UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

SEDE VICTOR LEVI SASSO

ESTRUCTURA DE DATOS II

INCLUYE PRUEBAS SUMATIVAS Y PRESENTACIONES DEL CONTENIDO

ELABORADO POR:

DR. CARLOS A. ROVETTO

AGOSTO 2021

Contenido

Índice de figuras

Introducción

Esta asignatura consta de dos módulos. En el módulo I se desarrollarán temas relacionados con las técnicas de estructuración de datos no lineales las cuales incluyen los árboles y los grafos. En la sección de los árboles veremos su representación, recorridos, operaciones, árboles en montón, entre otros. En el tópico de grafos se definirá qué es un grafo, se explicarán los tipos de grafos, su representación, las operaciones que se pueden realizar en los mismos y sus recorridos.

En el módulo II se desarrollarán temas relacionados con el ordenamiento y la búsqueda, en donde se explicará la Notación de la Gran O y los distintos algoritmos de ordenamiento y de búsqueda. También se desarrollará el tópico de la transformación de llaves y las distintas técnicas de cálculo de direcciones, sus comparaciones y los métodos para el manejo del problema de las colisiones.

Las estructuras de datos nos permiten la reutilización de componentes en programación. El mejoramiento en los procesos de información depende de una buena disposición de los datos y el manejo adecuado del espacio de memoria lo que se consigue con un buen diseño y utilización de las estructuras de datos.

Las estructuras de datos se pueden usar como herramientas para la evaluación de soluciones optimas de problemas y necesidades en el complejo mundo de la informática. Estas son la base para el diseño de aplicaciones importantes en distintas áreas de sistemas como lo son Bases de datos, Sistemas operativos, Compiladores, Redes, etc.

Capítulo I

1.1 Técnicas de estructuración de datos no lineales

1.1.1 Definición y conceptos

Un árbol es una estructura de datos no-lineal y dinámica. Es no-lineal debido a que a cada elemento pueden seguirle varios elementos y es dinámica porque su estructura puede cambiar durante su ejecución.

1.1.1.1 Árboles generales

Un árbol general es un árbol donde cada nodo puede tener cero o más hijos (un árbol binario es un caso especializado de un árbol general). Los árboles generales se utilizan para modelar aplicaciones como los sistemas de archivos.

Se puede definir a un árbol como un conjunto finito no vacío T de elementos, llamados nodos, tales que:

- Existe un nodo raíz.
- El resto de los nodos se distribuye en un número n de subconjuntos distintos.
- Cada uno de estos subconjuntos es un subárbol del nodo raíz.

En la Figura 1 se muestran cada uno de estos elementos.

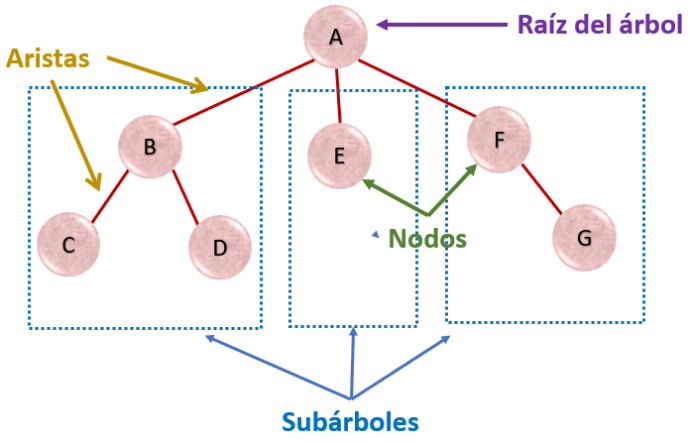


Figura 1. Árbol general. Elaboración propia.

Nodos del árbol: A, B, C, D, E, F

Nodo raíz: A

Raíz: es el primer nodo del árbol, se encuentra ubicado en la parte superior del árbol. Solo hay una raíz por árbol y una ruta desde el nodo raíz a cualquier nodo.

Aristas o ramas: son las líneas que unen dos nodos.

Padre: es el nodo que tiene hijos, es decir, nodos inferiores que se encuentran unidos al nodo superior por una arista. El único nodo que no tiene padre es el nodo raíz del árbol.

A cada nodo se le asocian uno o varios subárboles llamados descendientes o hijos.

Descendientes o hijos del nodo B: C, D

Hijo: el nodo debajo de un nodo dado (padre) conectado por su arista hacia abajo. Cada nodo puede tener un número arbitrario de hijos.

Hijos del nodo A: B, E, F. Estos a su vez conforman los subárboles del árbol general.

Nodos hermanos: son los sucesores o descendientes directos de un mismo nodo (hijos de un mismo padre).

Nodos hermanos (hijos de B): C, D

Hojas: son los nodos sin hijos. También se les llama nodos terminales, nodos de grado o nodos externos.

Nodos hojas: C, D, E, G

Nodos internos: son los nodos que tienen al menos un hijo.

Nodos internos: A, B, F

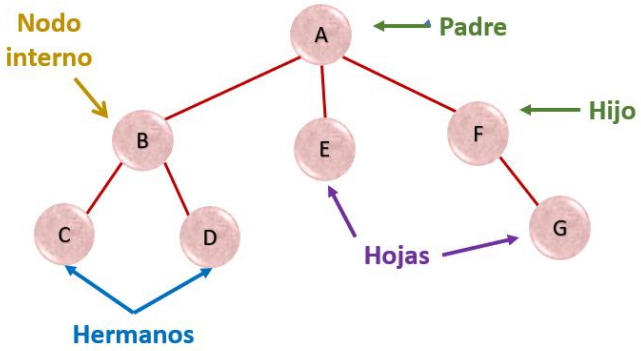


Figura 2. Conceptos de padre, hijo, hermanos, hojas, nodo interno. Elaboración propia.

Camino de un nodo: es la secuencia de aristas a través de las cuales se pasa desde el nodo raíz a un nodo.

Rama: Un camino que termina en una hoja.

Antecesores o predecesores de un nodo: son todos los nodos del camino que va desde la raíz del árbol hasta el nodo.

Antecesores de D: B, A

Descendientes o sucesores de un nodo: son aquellos nodos accesibles por un camino que comience en el nodo.

Descendientes del nodo E: H, I, J

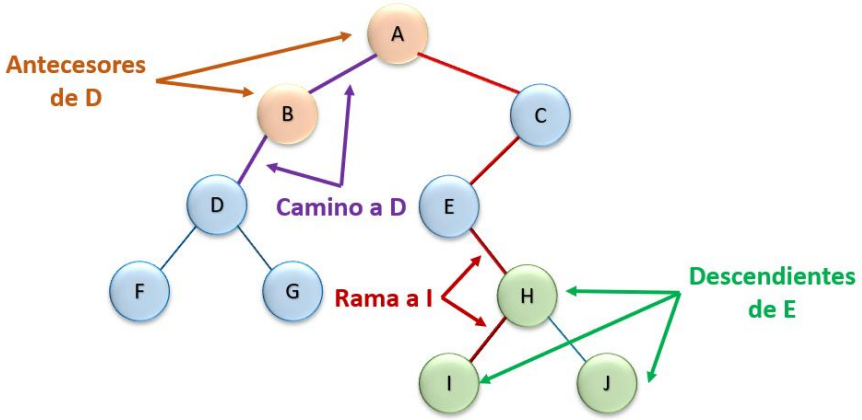


Figura 3. Conceptos de antecesores, descendientes, camino y rama. Elaboración propia.

Grado de un nodo: es el número de hijos que tiene el nodo. Así, el grado de un nodo hoja es cero. En la figura anterior el grado del nodo D es 2 y el grado del nodo E es 1.

Grado del árbol: es el mayor grado de sus nodos. Por ejemplo, el árbol binario es de grado 2 porque cada nodo tiene como mucho dos descendientes directos.

Niveles: es la distancia desde la raíz en la que se encuentra ubicado cada nodo. Cada nodo de un árbol tiene asignado un número de nivel de la siguiente forma la raíz tiene el número de nivel 0, y al resto de los nodos se le asigna un número de nivel que es mayor en 1 que el número de nivel del padre.

Nivel de un nodo: se refiere a la distancia del nodo desde la raíz del árbol.

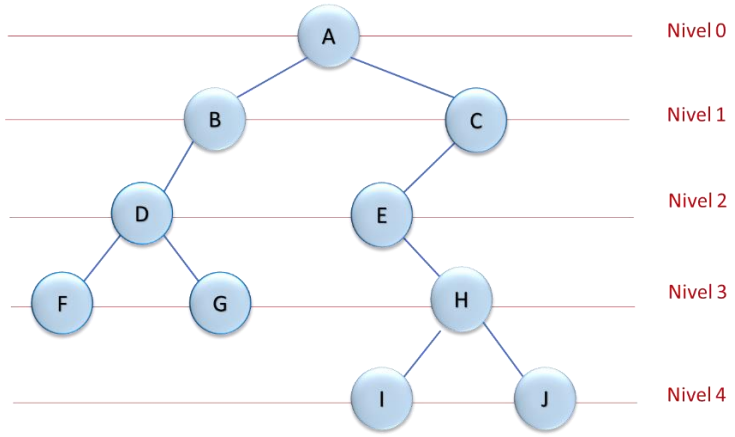


Figura 4. Conceptos de niveles del árbol. Elaboración propia.

Altura de un nodo: es la longitud del camino más largo que comienza en el nodo y termina en una hoja.

- La altura de un nodo hoja es 0
- La altura de un nodo es igual a la mayor altura de sus hijos + 1. Por ejemplo, la altura del nodo B en la figura anterior es 2.

Altura de un árbol: es la longitud de la rama más larga del árbol más uno. Equivale a 1 más que el mayor número de nivel del árbol.

Profundidad de un nodo: es la longitud del camino (único) que comienza en la raíz y termina en el nodo. También se denomina nivel.

- La profundidad de la raíz es 0
- La profundidad de un nodo es igual a la profundidad de su padre + 1

1.1.1.2 Árboles binarios

El árbol binario es un árbol en el cual cada nodo tiene como máximo dos hijos, uno a la izquierda y el otro a la derecha. En un **árbol binario de búsqueda** el hijo izquierdo, si existe, debe tener un valor menor que el valor de su padre y el hijo derecho, si existe, debe tener un valor mayor que el valor de su padre.

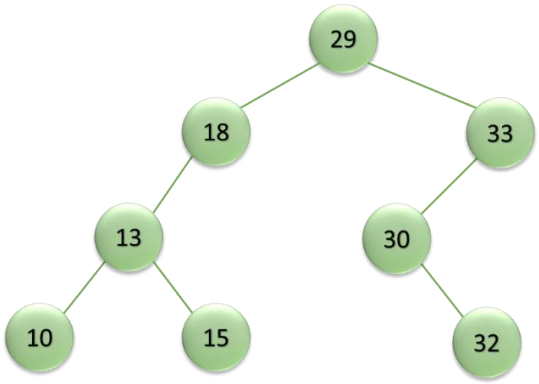


Figura 5. Árbol binario de búsqueda. Elaboración propia.

El número máximo de nodos en cualquier nivel N de un árbol binario es 2^N.

Árboles binarios distintos: dos árboles binarios son distintos cuando sus estructuras son diferentes.

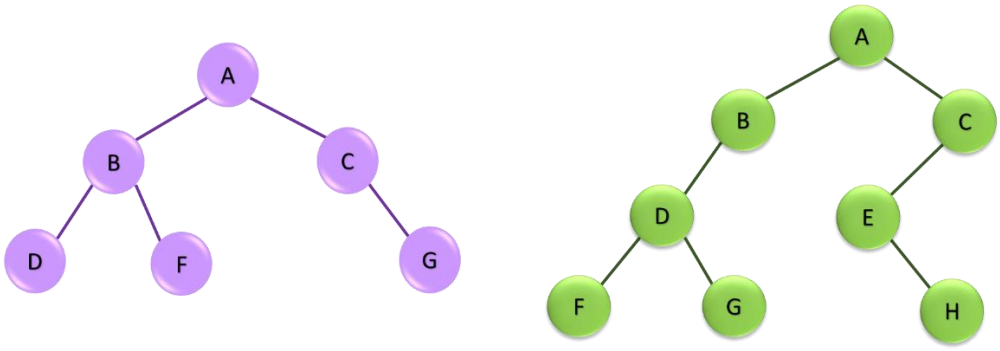


Figura 6. Árboles binarios distintos. Elaboración propia.

Árboles binarios similares: dos árboles binarios son similares si sus estructuras (forma) son idénticas pero la información que contienen sus nodos difiere entre sí.

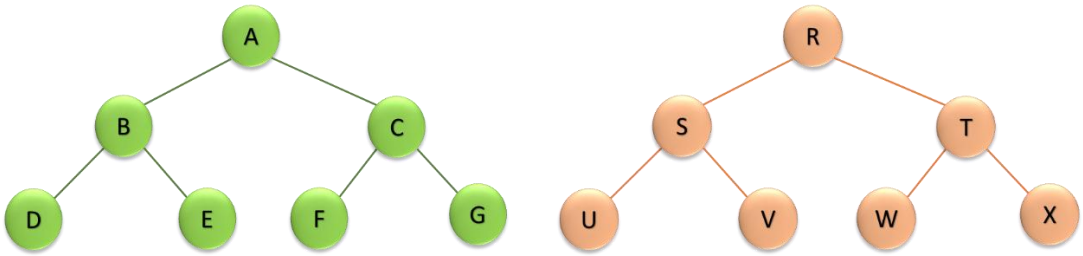


Figura 7. Árboles binarios similares. Elaboración propia.

Árboles binarios equivalentes: son aquellos que son similares y los nodos contienen la misma información.

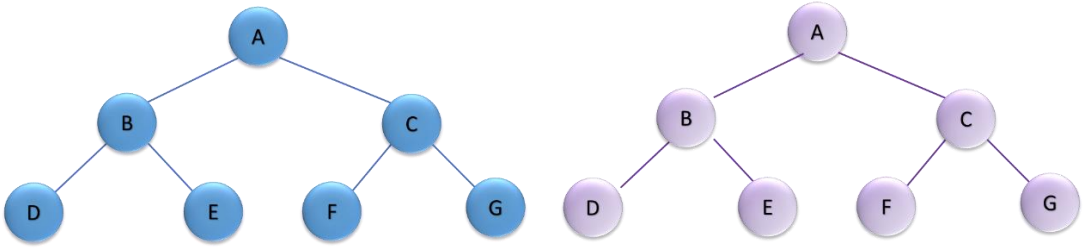


Figura 8. Árboles binarios equivalentes. Elaboración propia.

Árbol binario completo: el árbol binario es completo si todos sus niveles, excepto posiblemente el último, tienen el máximo número de nodos posibles y si todos los nodos del último nivel están situados lo más posible a la izquierda.

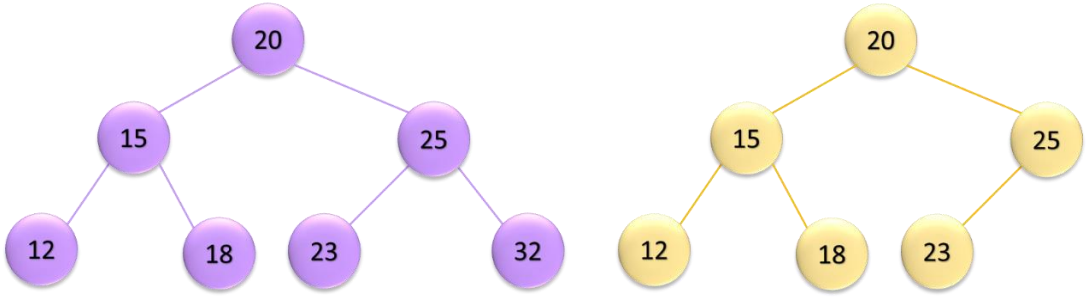


Figura 9. Árboles binarios de búsqueda completos. Elaboración propia.

Árboles binarios extendidos: árboles-2

Es un árbol binario en donde el número de hijos de cada nodo es igual al grado del árbol, en este caso, es 2. Si alguno de los nodos no cumple con esta condición, entonces se le deben agregar tantos nodos especiales como se requiera para llegar a cumplirla.

Para convertir el árbol binario de la siguiente figura a un árbol-2 añadimos a cada nodo, que no tenga dos hijos, los nodos externos necesarios para hacer cumplir dicha cantidad. Los nodos externos añadidos son representados por medio de un cuadrado.

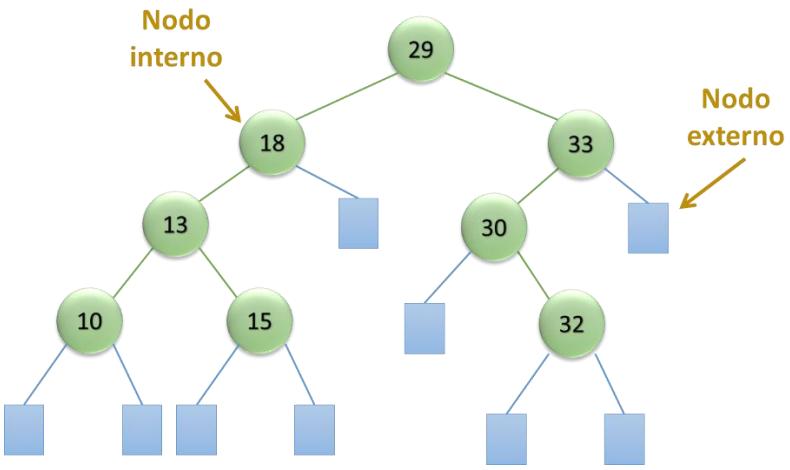


Figura 10. Árboles binarios de búsqueda extendidos. Elaboración propia.

1.1.2 Representación de árboles en memoria

Existen dos formas de representar un árbol binario en memoria:

- Por medio de punteros o enlazada
- Por medio de arreglos

La representación de punteros o enlazada es análoga a la forma en que se representan las listas enlazadas en memoria; la representación secuencial utiliza un arreglo simple.

1.1.2.1 Enlazada

Un árbol binario se puede representar en memoria de forma enlazada utilizando tres arreglos paralelos: INFO, IZQ y DER como se muestra en la figura de abajo y una variable puntero a la que llamaremos RAIZ.

Figura 11. Arreglos paralelos usados en la representación enlazada de un árbol binario de búsqueda. Elaboración propia.

A cada nodo N del árbol le corresponderá una posición K, tal que:

- INFO [K] contendrá los datos del nodo N.
- IZQ [K] contendrá la localización del hijo izquierdo del nodo N.
- DER [K] contendrá la localización del hijo derecho del nodo N.

La raíz contendrá la posición de la raíz R del árbol. Cuando el árbol está vacío, la RAIZ contendrá el valor nulo.

En el siguiente ejemplo se mostrará la representación enlazada en memoria de un árbol binario de búsqueda. Observe que cada nodo está dibujado con sus tres campos y que los subárboles vacíos están dibujados usando un diagonal / para las entradas nulas.

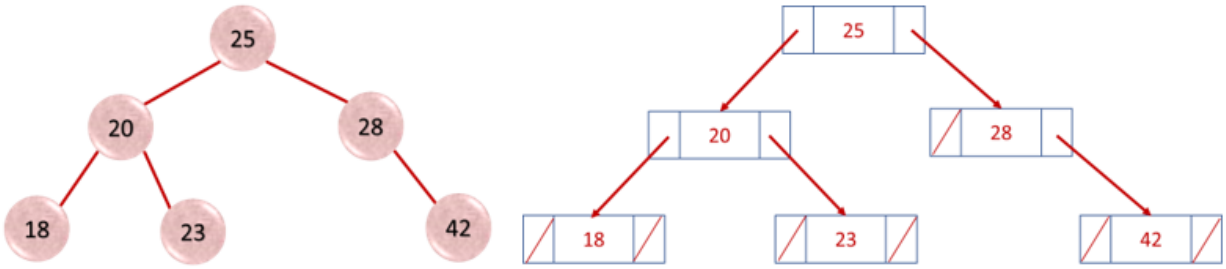


Figura 12. Representación enlazada de un árbol binario de búsqueda. Elaboración propia.

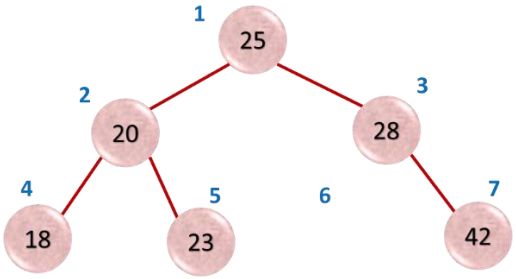
1.1.2.2 Secuencial

Esta representación usa un arreglo lineal al que llamaremos ARBOL de la forma siguiente:

- La raíz R del árbol se guarda en la posición del arreglo ARBOL [1].
- Si un nodo N está ubicado en la posición ARBOL [K], entonces sus hijos:
    - izquierdo está en la posición ARBOL [2*K]
    - derecho en la posición ARBOL [2*K+1]

Se usa nulo para indicar que el árbol o un subárbol está vacío, así ARBOL [1] = NULO indica que el árbol está vacío.

En el siguiente ejemplo se mostrará la representación secuencial en memoria de un árbol binario de búsqueda. Observe que cada nodo está ubicado en el arreglo en la misma posición que tiene en el árbol. Para una mejor comprensión de dichas posiciones se ha colocado el número que le corresponde a cada una de ellas al lado del nodo en el árbol.



ARBOL

POSICIÓN	1	2	3	4	5	6	7
VALOR	25	20	28	18	23		42

Figura 13. Representación Secuencial de un árbol binario de búsqueda. Elaboración propia.

1.1.3 Recorridos en un árbol binario

El recorrido en un árbol es el proceso de visitar todos los nodos de un árbol. Se clasifican los algoritmos de recorrido, dependiendo del orden en que se visitan los nodos.

Recorrido en Amplitud: se implementa utilizando la estructura de datos denominada cola. El recorrido empieza desde la raíz y atraviesa el árbol nivel por nivel, pasa por todos los nodos de un nivel antes de pasar a los nodos hijos.

Ejemplo

Utilizando el siguiente árbol escriba el recorrido por amplitud.

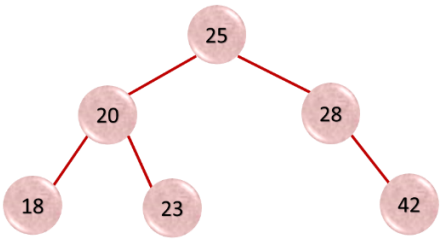
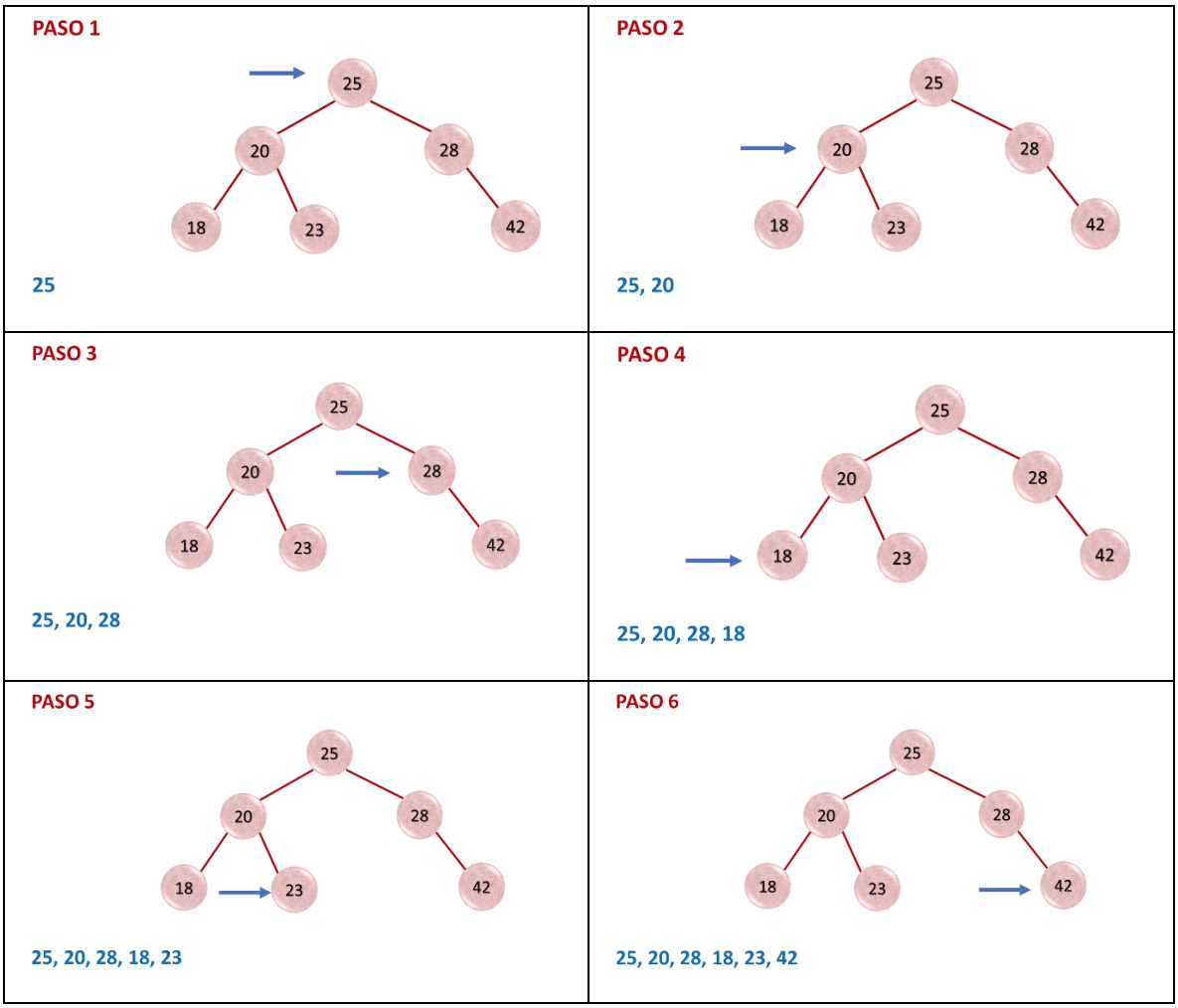


Figura 14. Árbol binario de búsqueda utilizado para realizar el recorrido en amplitud. Elaboración propia.

Solución



Recorrido en Profundidad: se implementa utilizando la estructura de datos denominada pila. El algoritmo empieza desde la raíz y visita a todos los nodos de una sola rama del

árbol. Cuando termina en esa rama, entonces hace una vuelta hacia atrás (denominado backtracking) y sigue por la otra rama.

Recorridos en profundidad

- Preorden
- Inorden
- Postorden

1.1.3.1 Preorden

ORDEN DEL RECORRIDO

- Raíz
- Subárbol izquierdo
- Subárbol derecho

Ejemplo

Utilizando el siguiente árbol escriba el recorrido por preorden.

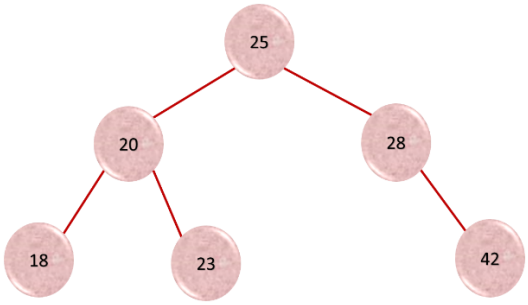
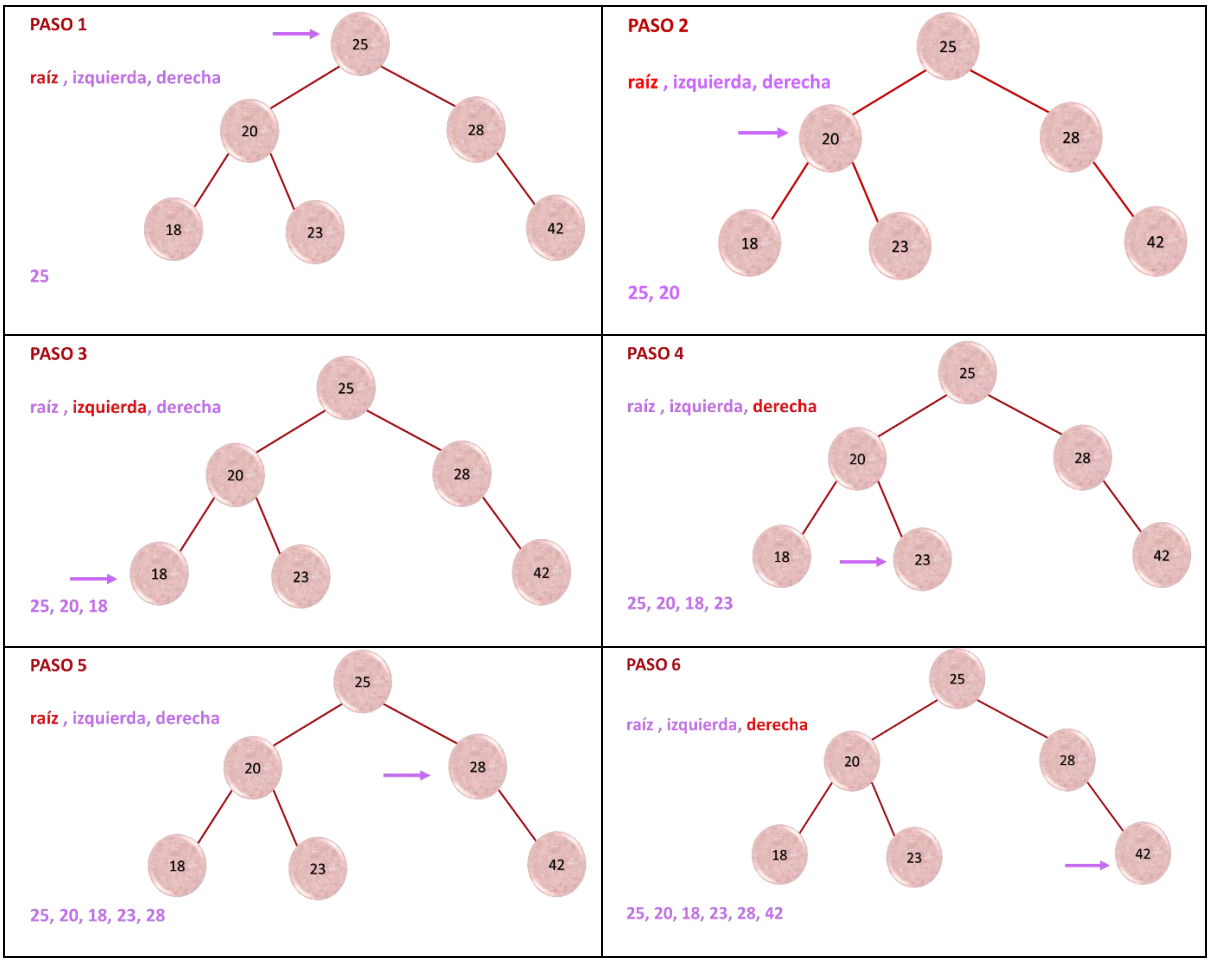


Figura 15. Árbol binario de búsqueda utilizado para realizar el recorrido en profundidad. Elaboración propia.

Solución



1.1.3.2 Inorden

ORDEN DEL RECORRIDO

- Subárbol izquierdo
- Raíz
- Subárbol derecho

Ejemplo

Utilizando el siguiente árbol escriba el recorrido por inorden.

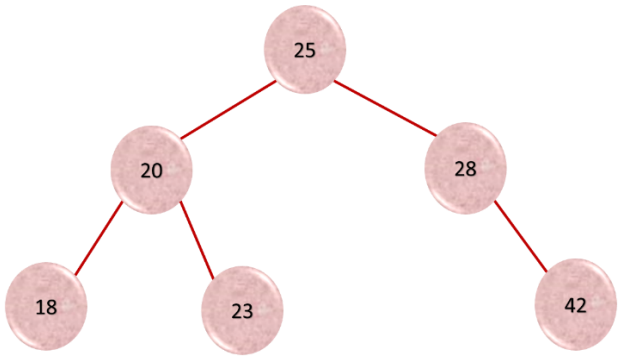
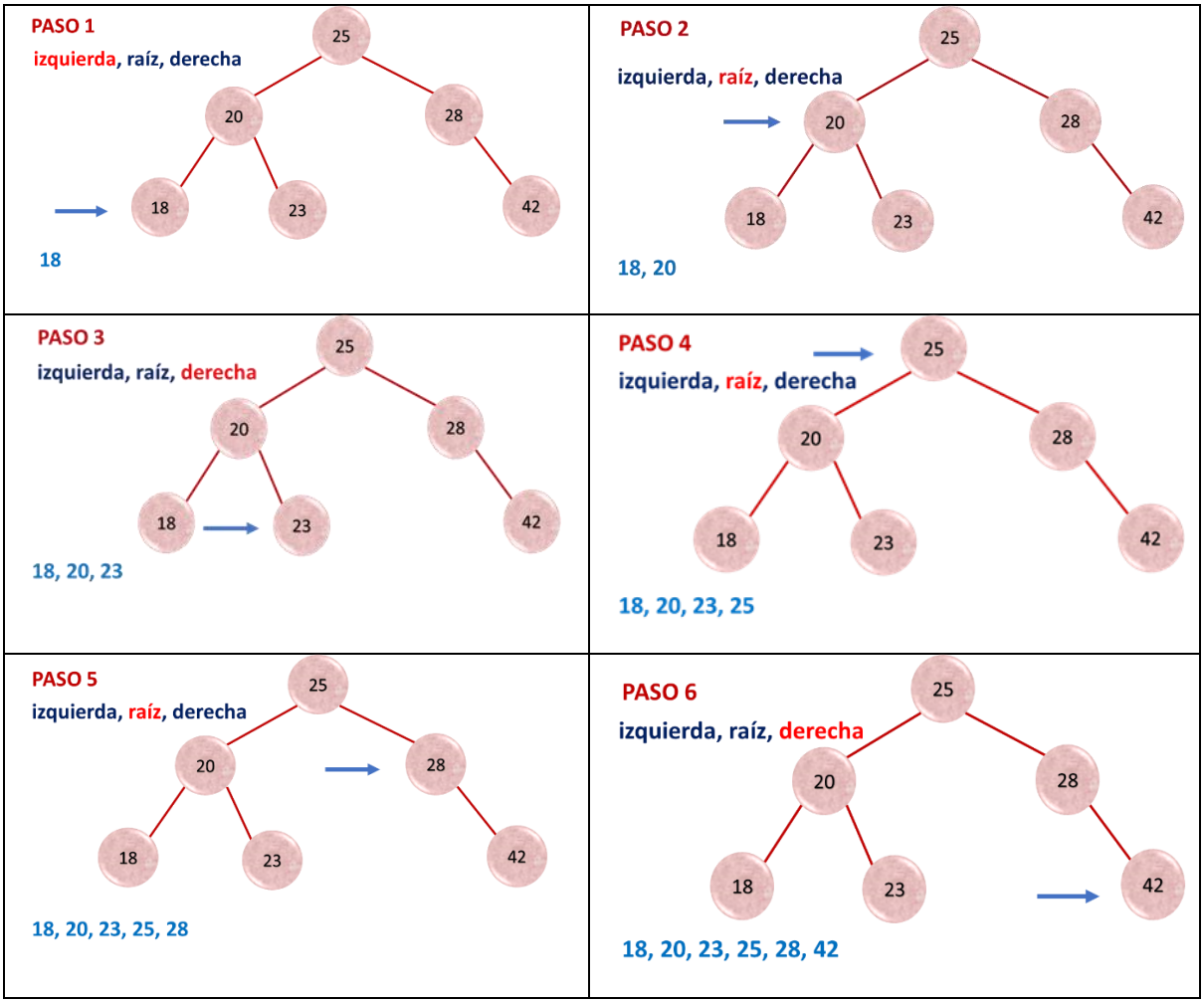


Figura 16. Árbol binario de búsqueda utilizado para realizar el recorrido en profundidad. Elaboración propia.

Solución



1.1.3.3 Postorden

ORDEN DEL RECORRIDO

- Subárbol izquierdo
- Subárbol derecho
- Raíz

Ejemplo

Utilizando el siguiente árbol escriba el recorrido por postorden.

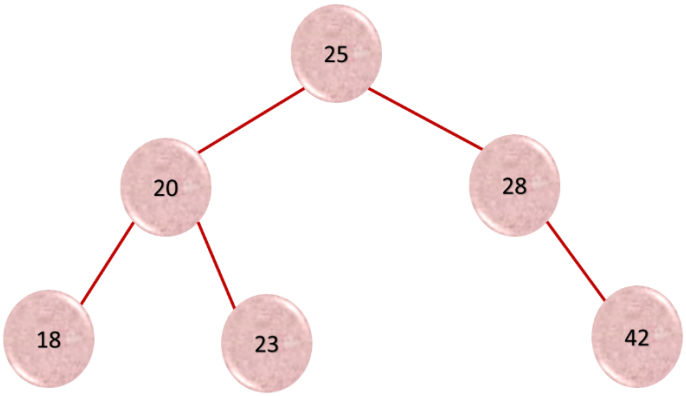
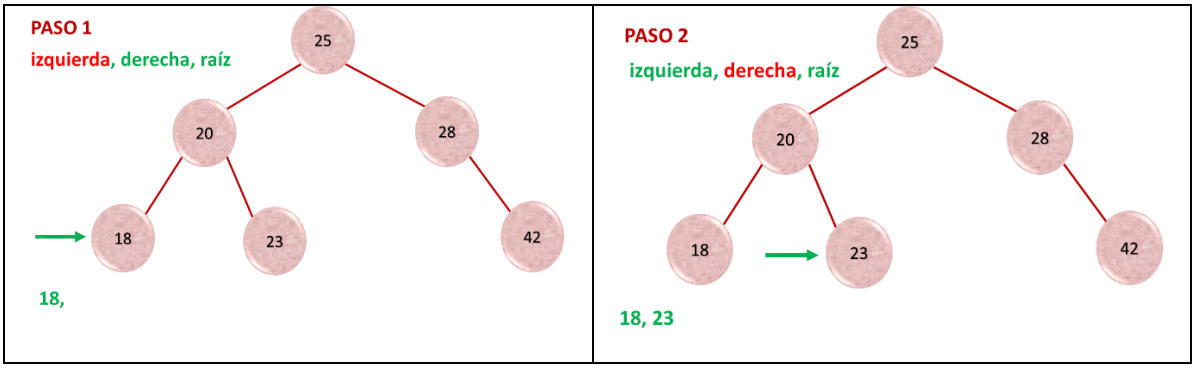
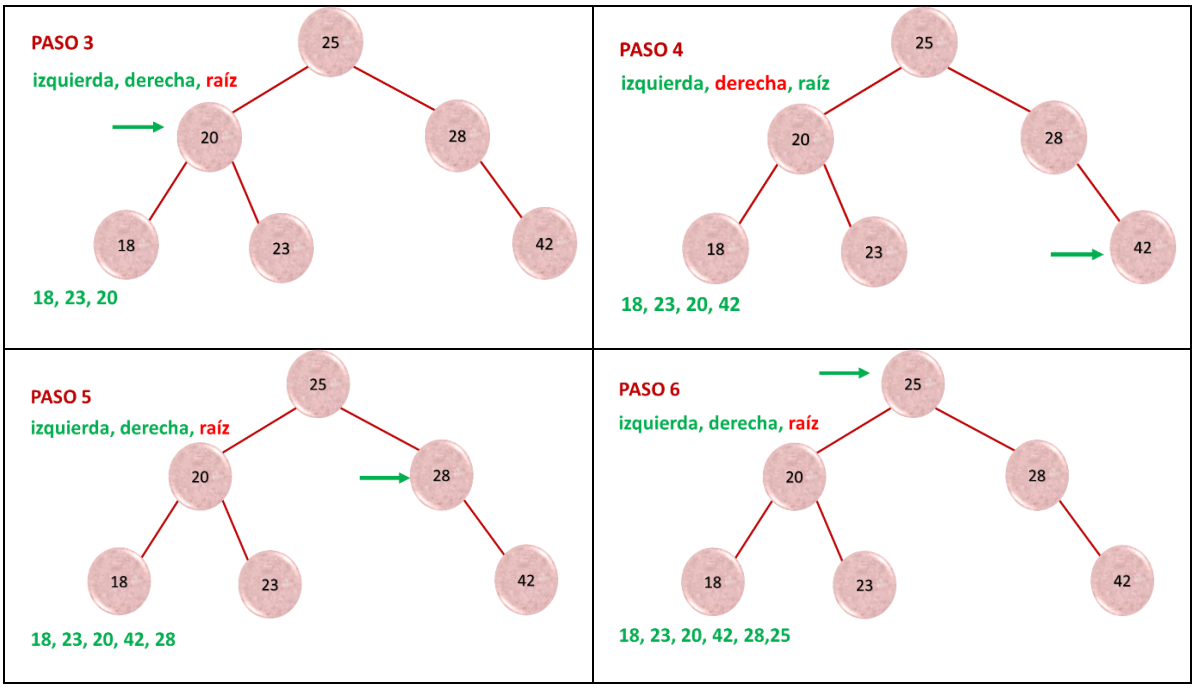


Figura 17. Árbol binario de búsqueda utilizado para realizar el recorrido en profundidad. Elaboración propia.

Solución





1.1.4 Operaciones sobre un árbol binario

En un árbol binario los nodos almacenan elementos cuyos valores son comparables mediante menor " < " y mayor " > " debido a que los mismos cumplen la propiedad de ordenación la cual indica que: "todo nodo es mayor que los nodos de su subárbol izquierdo, y menor que los nodos de su subárbol derecho".

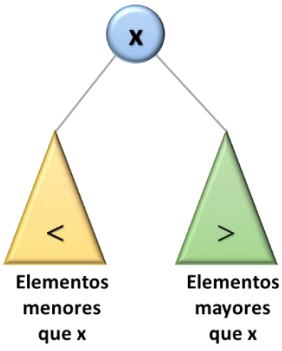


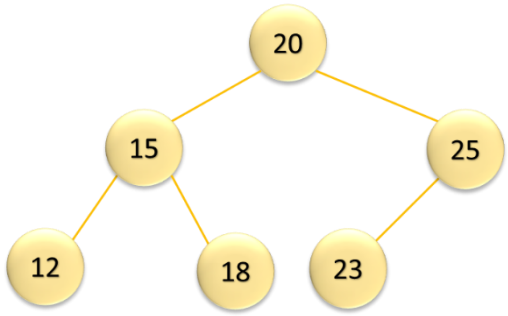
Figura 18. Propiedad de ordenación de los árboles binarios. Elaboración propia.

Para buscar un elemento, utilizando la propiedad antes mencionada, se parte de la raíz y se desciende escogiendo el subárbol izquierdo si el valor buscado es menor que el del nodo o el subárbol derecho si es mayor. Este proceso de búsqueda implica mover un puntero (**ptr**) a la derecha o a la izquierda hasta que se encuentra el valor deseado utilizando los siguientes criterios:

- Ubicar el puntero (**ptr**) en la raíz del árbol
- Comparar el valor de **ClaveNueva** (que es el valor que buscamos) con el **info** (**vnodo**) que tiene **ptr**.
    - Si **vnodo** = **ClaveNueva** se ha encontrado el nodo deseado
    - Si **vnodo** < **ClaveNueva** mover ptr al hijo izquierdo del nodo
    - Si **vnodo** > **ClaveNueva** mover ptr al hijo derecho del nodo
    - Continuar comparando hasta que se encuentra el nodo correcto, o hasta que nos caemos del árbol, donde **ptr** = Nulo

Ejemplo:

Buscar el valor de 18 en el siguiente árbol utilizando el Algoritmo de Búsqueda dado en el material complementario



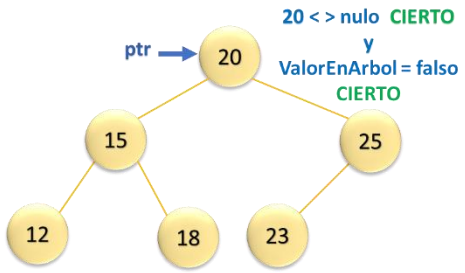
Solución

ClaveNueva = 18

Paso 1

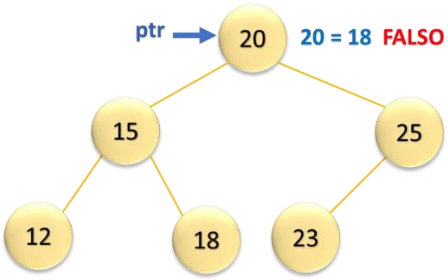
ptr ← Raízarbol
ValorEnArbol ← Falso

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer



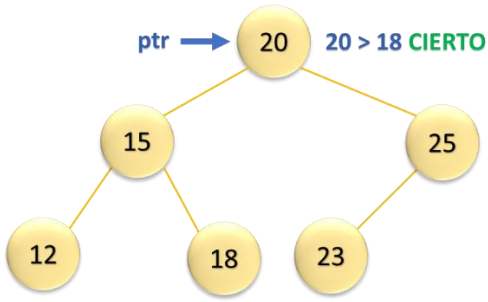
Paso 2

Si (ptr.info = ClaveNueva)



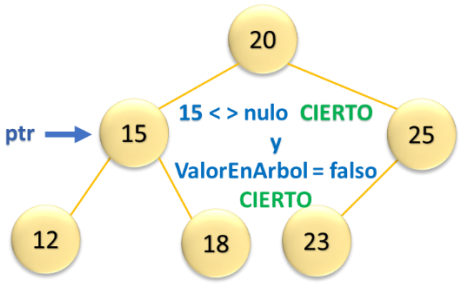
Paso 3

Si (ptr.info > ClaveNueva)



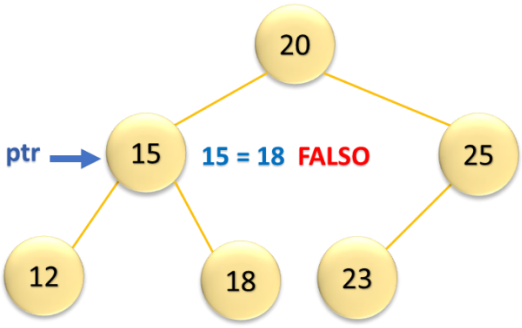
Paso 4

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer



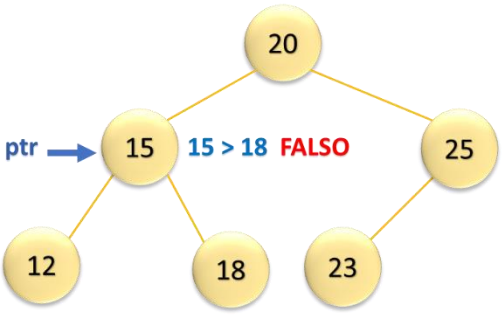
Paso 5

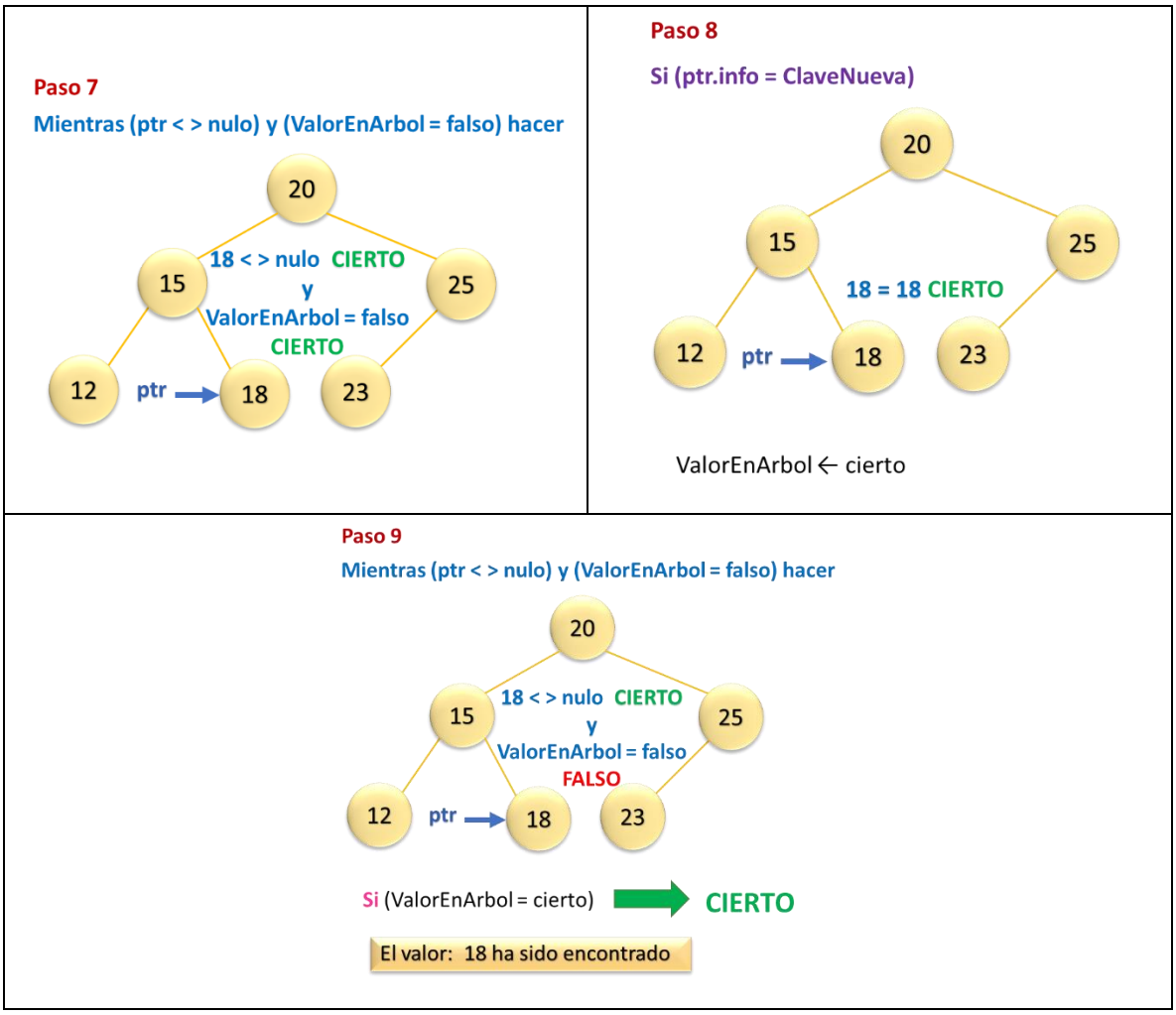
Si (ptr.info = ClaveNueva)



Paso 6

Si (ptr.info > ClaveNueva)





1.1.4.1 Inserción

Para insertar un nuevo nodo, se busca en el árbol y se inserta como nodo hoja en la posición en el árbol donde debería encontrarse utilizando los siguientes criterios:

- Ubicar el puntero (**ptr**) en la raíz del árbol y un puntero anterior (**ant**) a nulo.
- Mover a **ptr** a la derecha o a la izquierda a través del árbol, como en la operación de búsqueda y **ant** irá justo en la posición anterior que tenía **ptr**.
- Comparar el valor de **ClaveNueva** (que es el valor a insertar) con el **info** (**vnodo**) que tiene **ptr**.

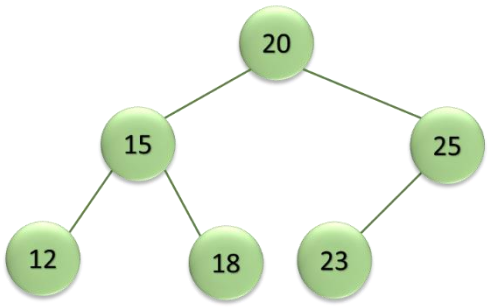
- Si **ptr** = Nulo y **ant** ≠ Nulo, **ptr** se cayó del árbol pero **ant** tiene la posición en la cual se debe insertar el nodo con **ClaveNueva**.
- Si **ant** = Nulo el árbol está vacío y el nodo a insertar con **ClaveNueva** es la raíz del árbol.

El algoritmo para la operación de inserción sobre un árbol binario de búsqueda específica tres tareas que deben ser realizadas:

- Crear un nodo para contener los nuevos datos
- Encontrar el lugar de inserción
- Cambiar los punteros para insertar el nuevo nodo dentro del árbol

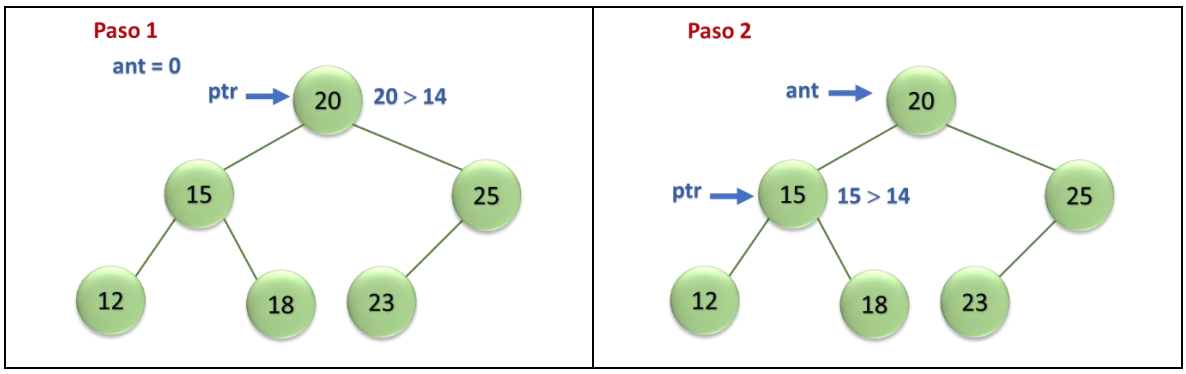
Ejemplo:

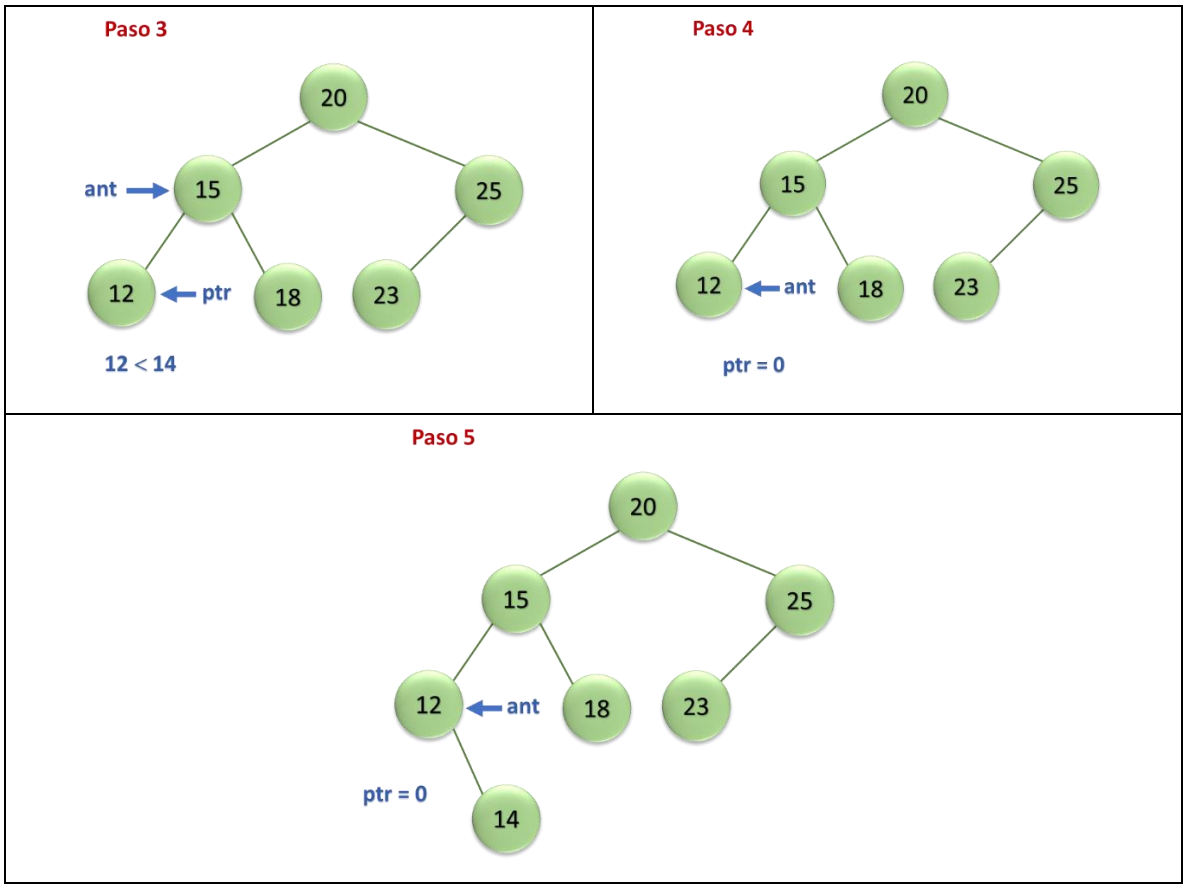
Insertar el valor de 14 en el siguiente árbol utilizando el Algoritmo de Inserción dado en el material complementario



Solución

ClaveNueva = 14





1.1.4.2 Eliminación

Para eliminar un nodo se busca en el árbol el valor que se desea eliminar (**ValorClave**) para suprimirlo. Esta operación se realiza en dos partes:

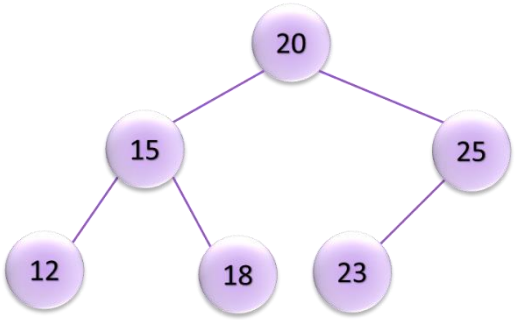
- Encontrar el nodo que contiene **ValorClave**
- Borrar el nodo del árbol

Tipos de eliminación

1. **Eliminación de una hoja (sin hijos):** en este caso se coloca el enlace del padre a nulo y se procede a deshacerse el nodo innecesario.

Ejemplo:

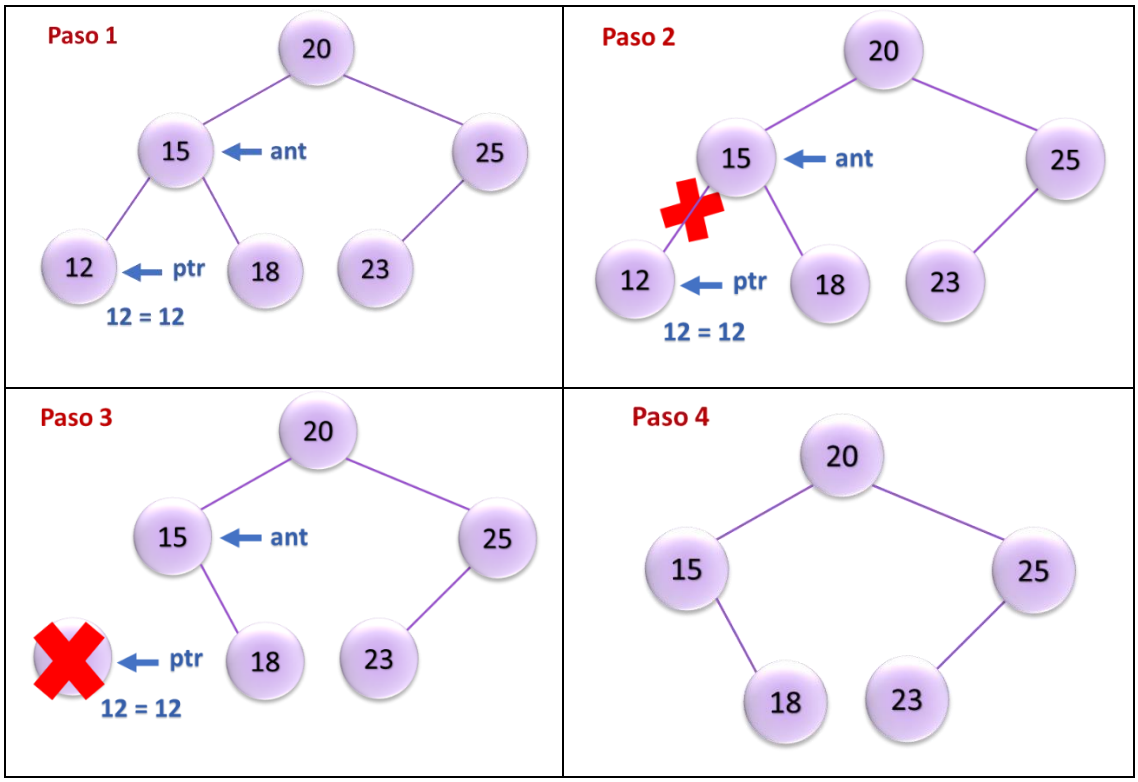
Eliminar el valor de 12 en el siguiente árbol utilizando el Algoritmo de Suprimir Nodo dado en el material complementario



Solución

ValorClave = 12

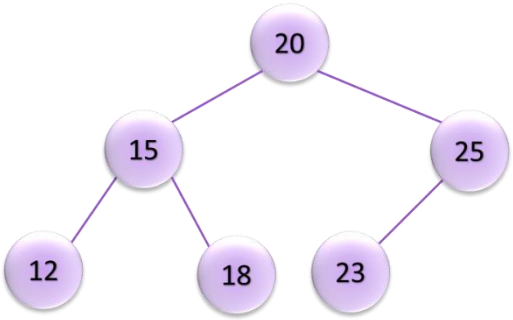
Después de buscar el valor a eliminar:



2. **Eliminación de un nodo con un solo hijo:** el puntero del padre debe saltar sobre el nodo que se va a suprimir y debe apuntar al hijo de dicho nodo (el nodo a suprimir). Luego se procede a deshacerse el nodo innecesario.

Ejemplo:

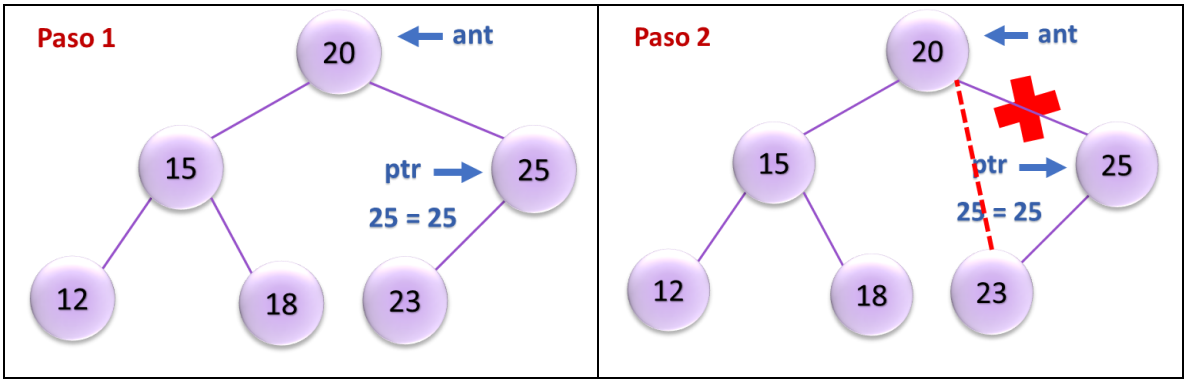
Eliminar el valor de 25 en el siguiente árbol utilizando el Algoritmo de Suprimir Nodo dado en el material complementario

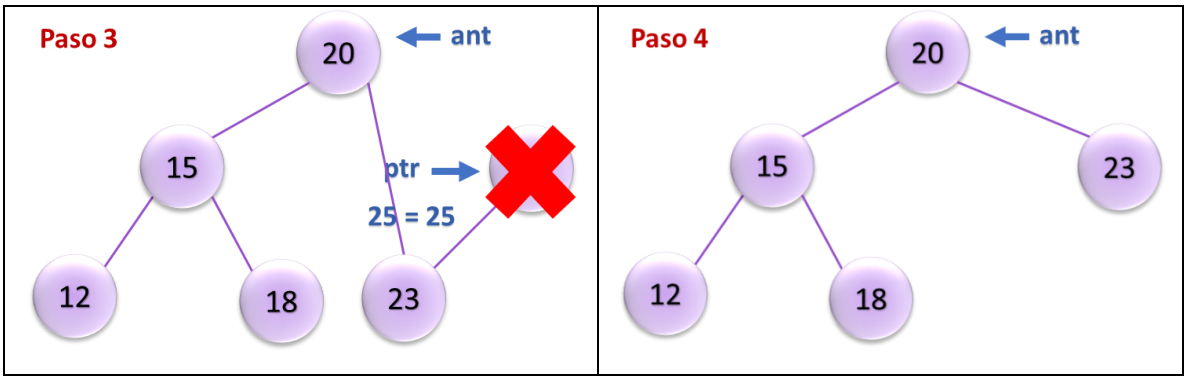


Solución

ValorClave = 25

Después de buscar el valor a eliminar:

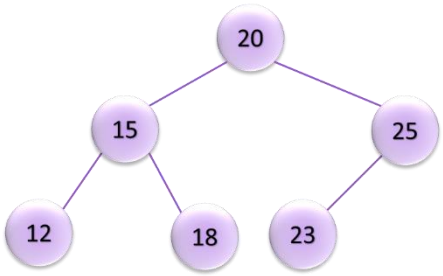




3. **Eliminación de un nodo con dos hijos:** para encontrar el predecesor inmediato, nos movemos una vez a la izquierda y luego tanto como podemos a la derecha. (Si el hijo izquierdo del nodo que queremos suprimir no tiene hijo derecho, entonces el propio hijo izquierdo se usa como el nodo que le reemplaza). A continuación, reemplazamos el valor a suprimir con el valor del nodo que le reemplaza. Luego podemos suprimir el nodo.

Ejemplo:

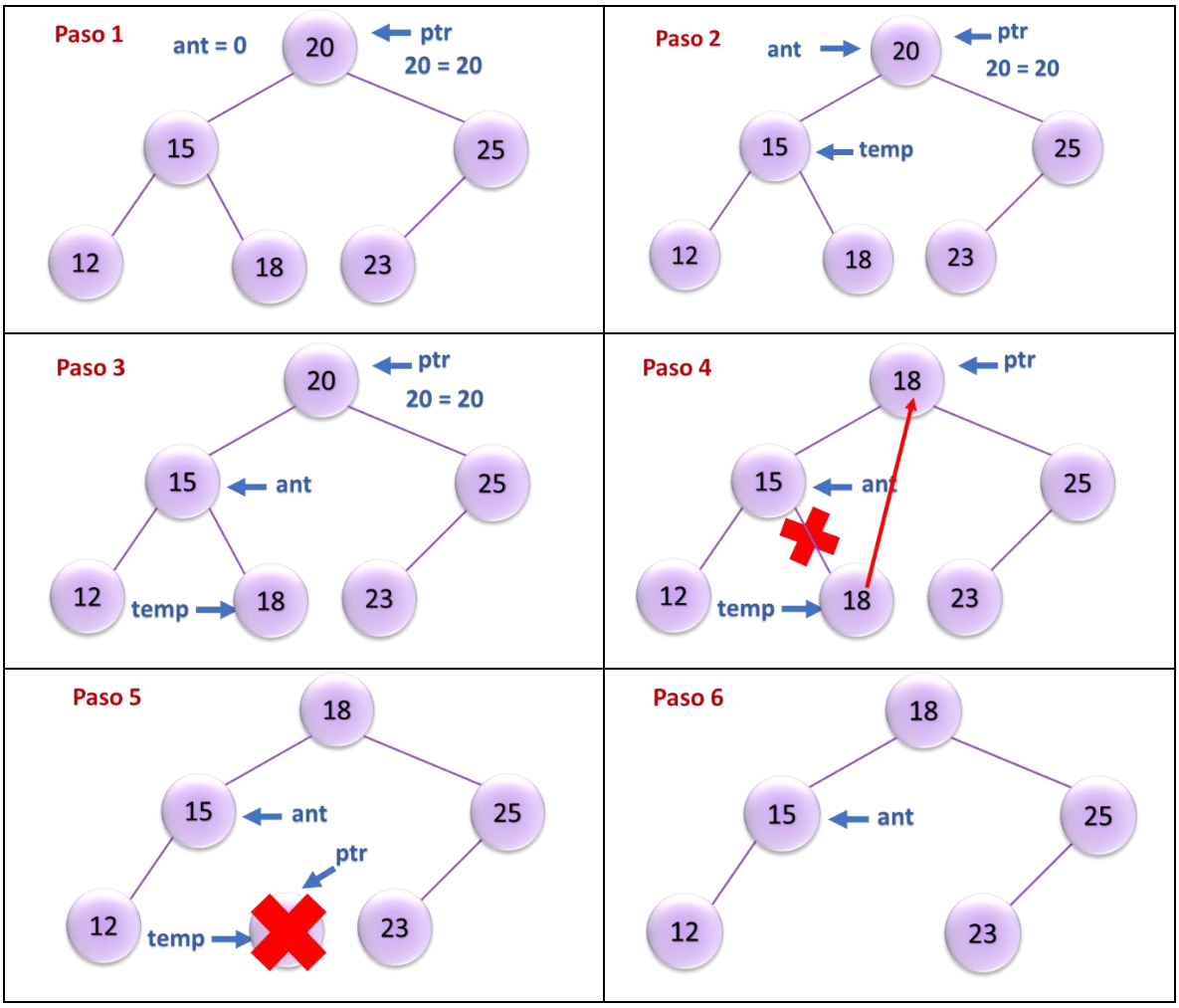
Eliminar el valor de 20 en el siguiente árbol utilizando el Algoritmo de Suprimir Nodo dado en el material complementario



Solución

ValorClave = 20

Después de buscar el valor a eliminar:



1.1.4.3 Algoritmo de Búsqueda

{

/* Busca en el árbol binario de búsqueda un nodo cuya clave es ClaveNueva, y devuelve los datos Info del nodo */

ptr ← Raízarbol

ValorEnArbol ← Falso

// Encontrar nodo del árbol que contiene ClaveNueva

// Al final del ciclo, si ClaveNueva está en el árbol, ptr apuntará a su nodo

```
Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
        Si (ptr.info = ValorClave)
                Entonces ValorEnArbol ← cierto
                Sino            //Continua buscando
                        Si (ptr.info > ClaveNueva)
                                Entonces ptr ← ptr.izquierdo
                                Sino ptr ← ptr.derecho
                         Fin-si
        Fin-si
Fin-mientras

// Si ClaveNueva se encontraba en el árbol, copia su parte info

Si (ValorEnArbol = cierto)
        Entonces Imprimir (“El valor: ”, ClaveNueva,  “ha sido encontrado”)
        Sino Imprimir (“El valor: ”, ClaveNueva,  “no ha sido encontrado”)
Fin-si
}
```

1.1.4.4 Algoritmo de Inserción

```
{
/* Construye nodo para contener InfoNodo, y lo inserta en el lugar apropiado del árbol
binario de búsqueda */

// Crea un nuevo nodo
Nuevo (Nuevonodo)
Nuevonodo.izquierdo ← nulo
```

```
Nuevonodo.derecho ← nulo
Nuevonodo.info ← InfoNodo
ClaveNueva ← Infonodo

// Busca el lugar de inserción
ptr ← RaízArbol
ant ← nulo
Mientras (ptr < > nulo) hacer        //Buscar hasta que ptr se caiga del árbol
        Ant ← ptr
        Si (ptr.Info > ClaveNueva)
                Entonces ptr ← ptr.izquierdo
                Sino ptr ← ptr.derecho
        Fin-si
Fin-mientras

/* Se encontró el lugar de inserción. Conectar punteros para enlazar el nuevo nodo al
árbol */

Si (ant = nulo)
        Entonces RaízArbol ← Nuevonodo        //Primer nodo en el árbol
        Sino            //Añadir al árbol ya existente
                Si (ant.info > ClaveNueva)
                        Entonces ant.izquierdo ← Nuevonodo
                        Sino ant.derecho ← Nuevonodo
                Fin-si
Fin-si
}
```

1.1.4.5 Algoritmo Suprimir Nodo

```
{
     // Encontrar nodo que contiene ValorClave
    // ValorClave es el valor a eliminar
ptr ← RaízArbol
ant ← nulo
Mientras (ptr.info < > ValorClave) hacer
   ant ← ptr
    si (ptr.info > ValorClave)
        entonces ptr ← ptr.izquierdo
        sino ptr ← ptr.derecho
    fin-si
Fin-mientras
// Caso de supresión de una hoja
Si (ptr.derecho = nulo) y (ptr.izquierdo = nulo)
   Entonces   si (ant = nulo)  // nodo ptr es el último nodo del árbol
                    Entonces RaízArbol ← nulo
                    Sino    // suprimir la hoja
                          Si (ant.derecho = ptr)
                             Entonces ant.derecho = nulo
                             Sino ant.izquierdo = nulo
                          Fin-si
                Fin-si
   Sino // caso de supresión de nodo con dos hijos
         Si (ptr.derecho < > nulo) y (ptr.izquierdo < > nulo)
            Entonces ant ← ptr
                        temp ← ptr.izquierdo
                        Mientras (temp.derecho < > nulo) hacer
                            ant ← temp
```

```
                    temp ← temp.derecho
                Fin-mientras
                ptr.info ← temp.info
                 si (ant = ptr)
                    entonces ant.izquierdo ← temp.izquierdo
                    sino ant.derecho ← temp.izquierdo
                 Fin-si
                 ptr ← temp
        sino   // caso en el que el nodo tiene un hijo
            si (ptr.derecho < > nulo)
                entonces    // hay un hijo derecho
                    si (ant = nulo)
                        entonces RaízArbol ← ptr.derecho
                        sino   si  (ant.derecho = ptr)
                                    entonces   ant.derecho ← ptr.derecho
                                     sino ant.izquierdo ← ptr.derecho
                                fin-si
                     fin-si
             sino  // hay un hijo izquierdo
                 si (ant = nulo)
                    entonces RaízArbol ← ptr.izquierdo
                    sino   si (ant.derecho = ptr)
                                entonces ant.derecho ← ptr.izquierdo
                                 sino ant.izquierdo ← ptr.izquierdo
                            fin-si
                  fin-si
              fin-si
          fin-si
fin-si
```

```
// Liberar nodo
liberar (ptr)
}
```

1.1.5 Árboles en montón

Un árbol en montón H (también llamado montículo) es usado para implementar colas de prioridad. Se dice que H es un árbol en montón, o **montón máx**, si cada nodo N de H tiene la siguiente propiedad: El valor de N es mayor o igual que el valor de cualquier hijo de N. (Un **montón min** se define análogamente: el valor de N es menor o igual que el valor de cualquier hijo de N).

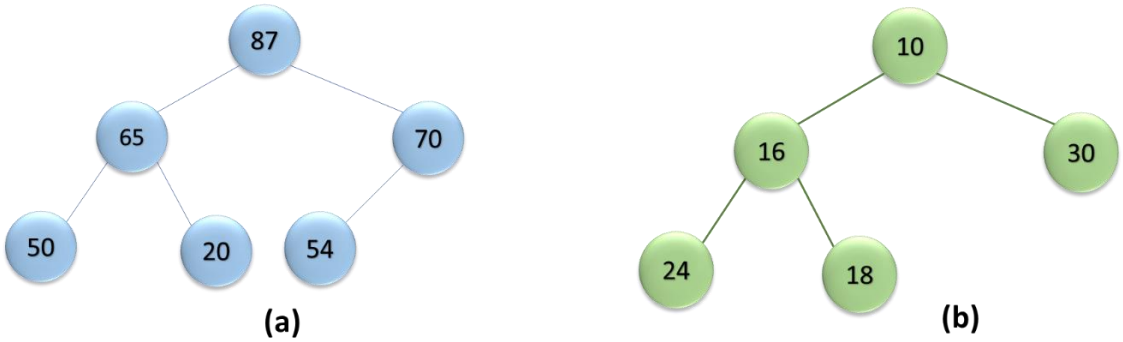


Figura 19. Árboles en montón: máximo (a) y mínimo (b). Elaboración propia.

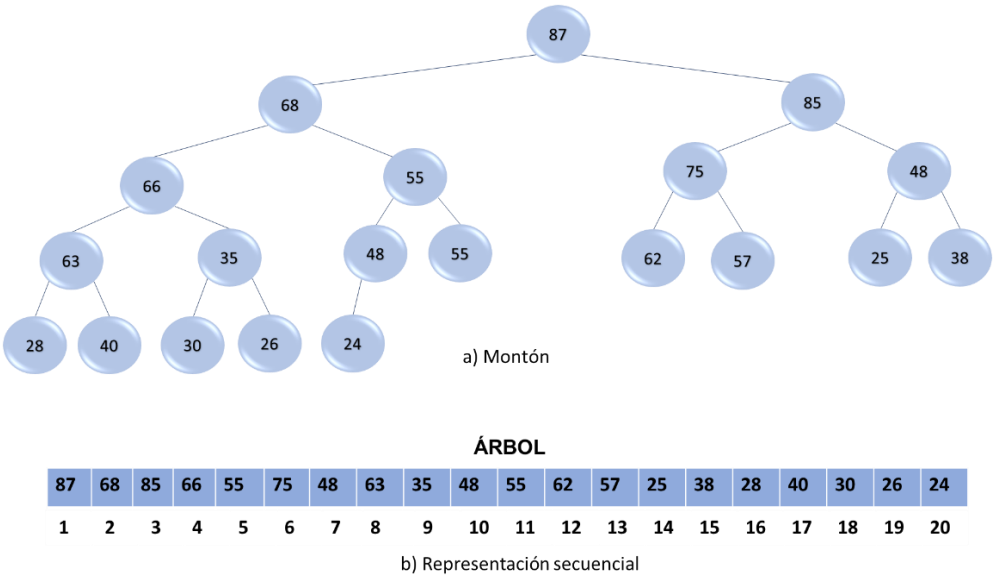


Figura 20. Árbol en montón máximo y su representación secuencial en memoria. Elaboración propia.

1.1.5.1 Inserción en un árbol en montón

Sea H un árbol en montón con N nodos e ITEM un nuevo nodo con información que deseamos insertar en el árbol. Insertamos ITEM en el montón H como sigue:

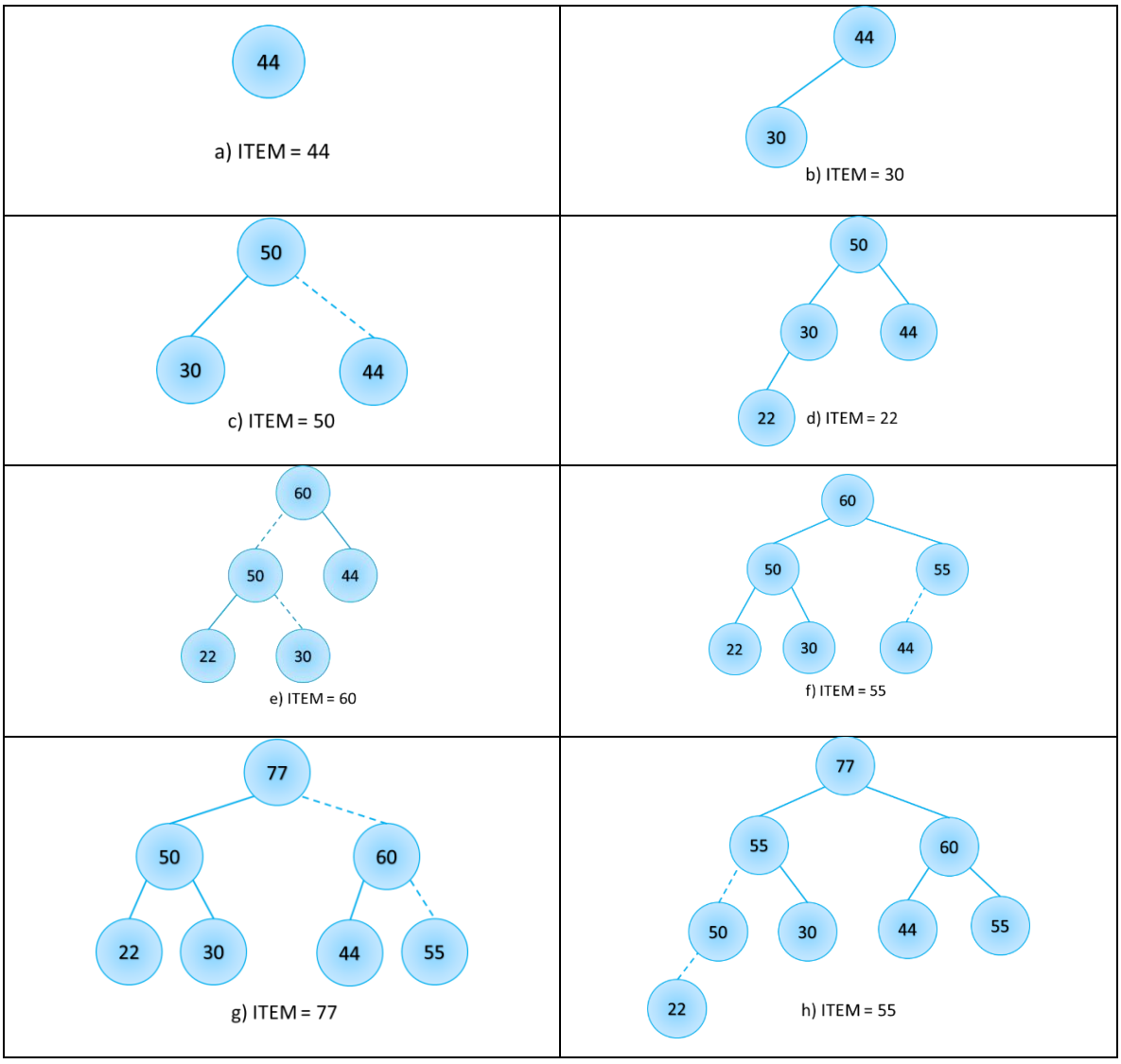
1) Primero se añade ITEM al final de H, de forma que H siga siendo un árbol completo, aunque no necesariamente un montón.
2) Entonces se hace subir a ITEM hasta su "lugar apropiado" en H para que H sea finalmente un montón.

Ejemplo:

Construir un árbol en montón máximo con la siguiente lista de números:

44, 30, 50, 22, 60, 55, 77, 55

Solución



1.1.5.2 Eliminación de la raíz de un montón

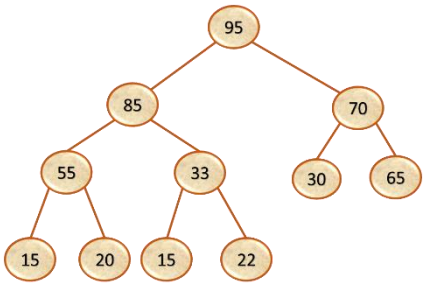
Sea H un árbol en montón con N nodos en donde se desea eliminar la raíz. Esta operación se lleva a cabo como sigue:

1) Asignar la raíz R a alguna variable ITEM.
2) Reemplazar el nodo R a eliminar con el último nodo L de H de forma que H sigue siendo completo, aunque no necesariamente un montón.

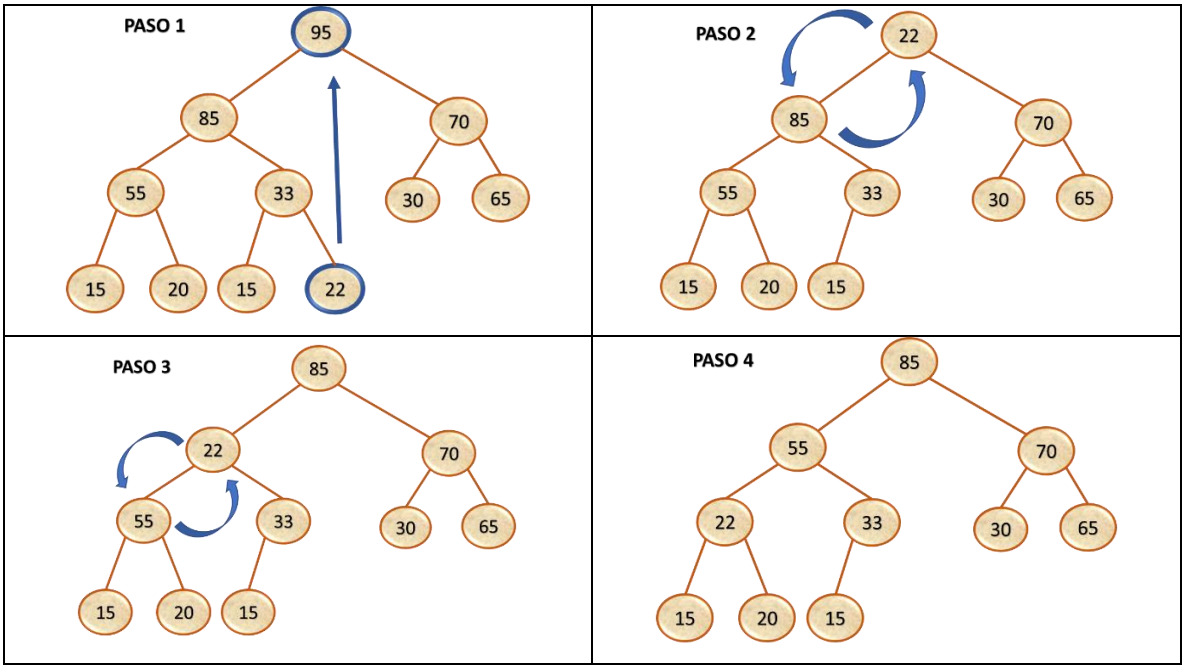
3) Reamontonar: hacer que L se mueva a su sitio adecuado en H para que H sea finalmente un montón.

Ejemplo:

Eliminar la raíz del siguiente árbol en montón.



Solución



1.1.6 Longitud de camino (Algoritmo de Huffman)

El algoritmo de Huffman se usa para la creación de códigos de Huffman, una técnica usada para la compresión de datos. Esta busca reducir la cantidad de datos o espacio de memoria para crear un mensaje, para ellos es necesario el uso de un alfabeto con n caracteres finitos, el tipo de símbolos que puede tener y las frecuencias que hay de cada

letra en el mensaje. Con esto se crea un código para cada una de las letras y finalmente se unen para crear la nueva versión del mensaje.

Este algoritmo trabaja con árboles binarios extendidos o árboles-2. En donde:

- **el número de nodos externos (N_E):** es 1 más que el número de nodos internos (N_I), $N_E = N_I + 1$
- **la longitud de camino interna (L_I):** es la suma de todas las longitudes de camino obtenidos sobre cada camino desde la raíz del árbol hasta un nodo interno.
- **la longitud de camino externa (L_E):** es la suma de todas las longitudes de camino obtenidos sobre cada camino desde la raíz del árbol hasta un nodo externo. También se puede encontrar si se conoce la longitud de camino interna L_I usando la siguiente fórmula: $L_E = L_I + 2n$, donde n es el número de nodos internos del árbol.
- la **longitud de camino con peso (externa) P:** se define como la suma de las longitudes de camino con sus pesos: $P = W_1L_1 + W_2L_2 + \ldots + W_nL_n$, donde W_iL_i denotan, respectivamente, el peso y la longitud del camino del nodo externo N_i.

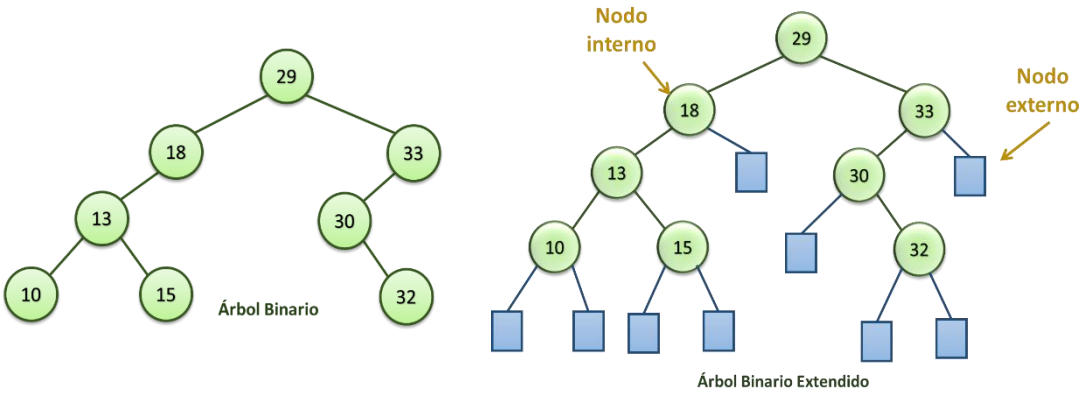


Figura 21. Ejemplo de Árbol Binario y Árbol Binario Extendido. Elaboración propia.

Ejemplo:

Encuentre en el siguiente árbol:

a) la cantidad de nodos internos N_I
b) la cantidad de nodos externos N_E
c) la longitud de camino interna L_I
d) la longitud de camino externa L_E
e) la longitud de camino con peso P

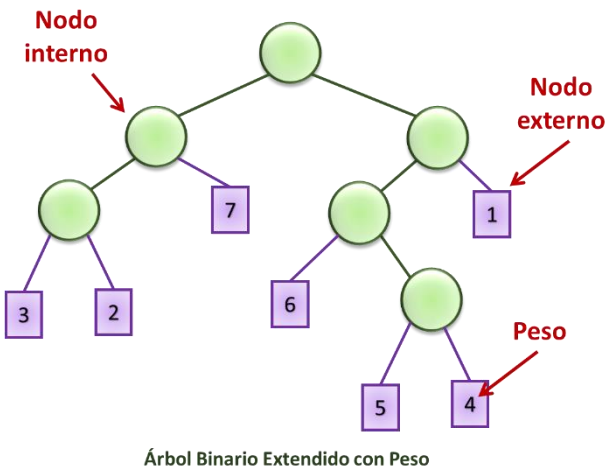


Figura 22. Árbol Binario Extendido con Peso. Elaboración propia.

Solución

a) $N_I = 6$

b) $N_E = N_I + 1 = 6+1$
$N_E = 7$

c) $L_I = 0 + 1 + 1 + 2 + 2 + 3$
$L_I = 9$

d) $L_E = 3 + 3 + 2 + 3 + 4 + 4 + 2$
$L_E = 21$

Usando la fórmula $L_E = L_I + 2n$
$L_E = 9 + 2(6)$
$L_E = 9 + 12$
$L_E = 21$

e) $P = 3*3 + 2*3 + 7*2 + 6*3 + 5*4 + 4*4 + 1*2$
$P = 85$

Construcción del árbol de mínima longitud de camino con peso mediante el Algoritmo de Huffman:

1. Se eligen dos de los subárboles con la menor combinación de pesos posible. (Recuerde que cada elemento pertenece a su propio subárbol).

2. Se unen para formar un nuevo subárbol con peso igual a la suma de ambos subárboles.
3. Se repiten los pasos anteriores hasta que se termine de formar el árbol.

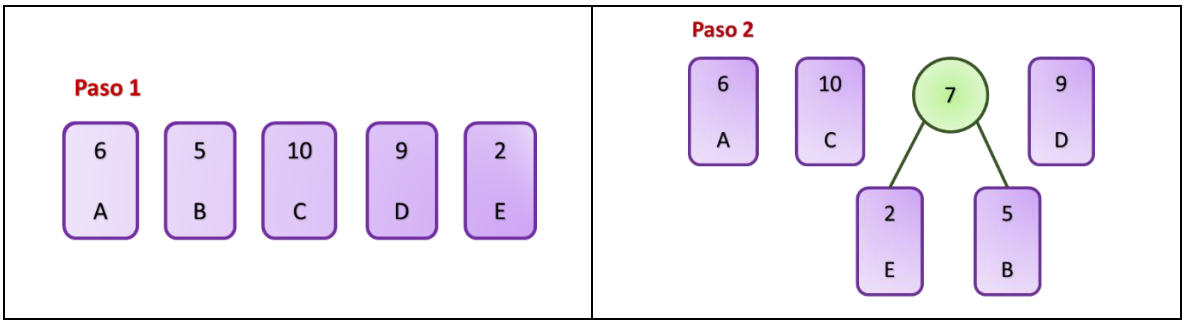
Ejemplo:

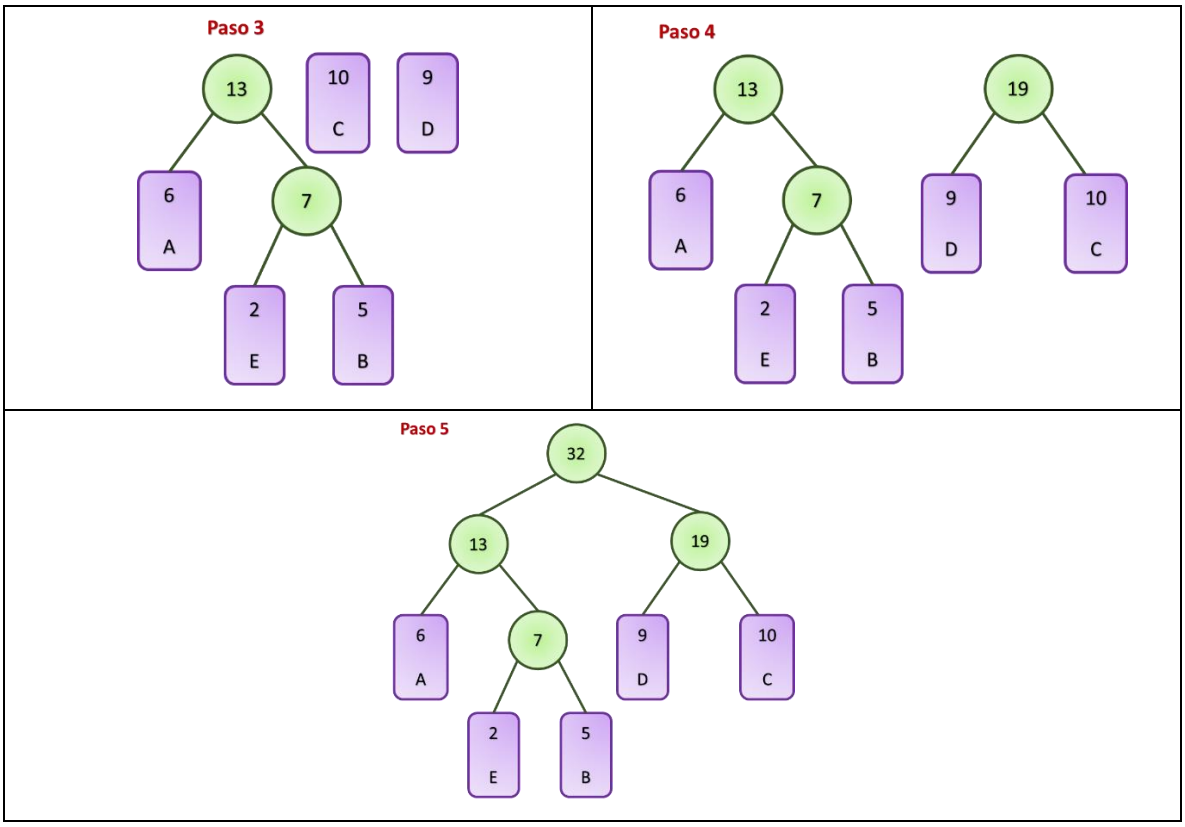
Construir un árbol de mínima longitud de camino con peso mediante el algoritmo de Huffman con los siguientes datos:

Dato:	A	B	C	D	E
Peso:	6	5	10	9	2

Solución

- Elegimos dos de los subárboles con la menor combinación de pesos posible: 2 y 5.
- Se unen para formar un nuevo subárbol con peso igual a la suma de ambos subárboles 7. Se debe colocar el peso inferior a la izquierda y el peso superior a la derecha.
- En cada paso unen los dos subárboles con menor peso.





1.1.7 Árboles binarios de expresión

Los árboles binarios de expresión son aquellos arboles utilizados para representar una expresión binaria en el cual la raíz del nodo contiene el operador y los dos hijos contienen los operandos. Cuando usamos un árbol binario para representar una expresión, los paréntesis no son necesarios para indicar la precedencia. Los niveles de los nodos del árbol indican implícitamente la precedencia relativa de evaluación.

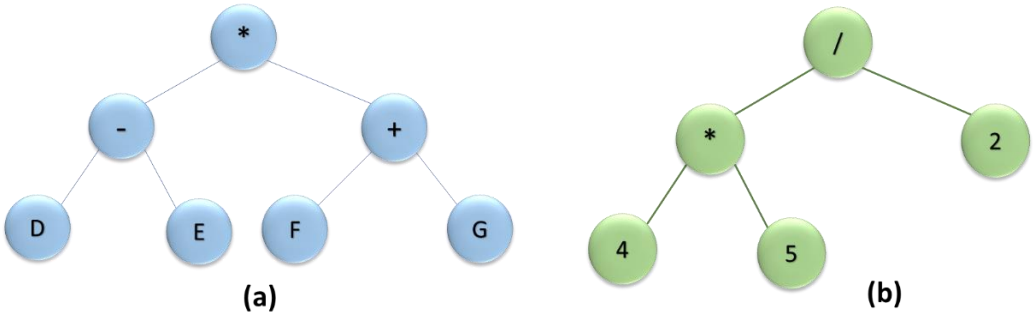


Figura 23. Árboles Binarios de Expresión. Elaboración propia.

1.1.7.1 Construcción de un árbol binario de expresión

El método que usaremos para construir el árbol es el siguiente: insertar nuevos nodos, cada vez moviéndonos hacia la izquierda hasta que pongamos un operando. Luego, vuelta atrás hasta el último operador, y poner el siguiente nodo a su derecha. Continuamos de la misma forma: si hemos insertado un nodo operador, ponemos el siguiente nodo a su izquierda; si hemos insertado un nodo operando, volvemos atrás y ponemos el siguiente nodo a la derecha del último operador.

Para poder llevar el control del último operador usaremos una pila en la cual vamos a guardar dicho operador, también se usará un indicador SigMov para indicar el lugar en el cual se debe colocar el siguiente nodo a la izquierda o a la derecha basándose en si el nodo actual contiene un operador o un operando. Para denotar el final de la expresión usaremos el carácter especial punto y coma (;).

Algoritmo ConstruirArbol

{

/* Construye el árbol binario de expresión que corresponde a la expresión en prefija */

ultimosímbolo ← ;

ObtenerSímb (CadenaPref, Símbolo) //Obtiene el primer símbolo

//Construye el nodo raíz

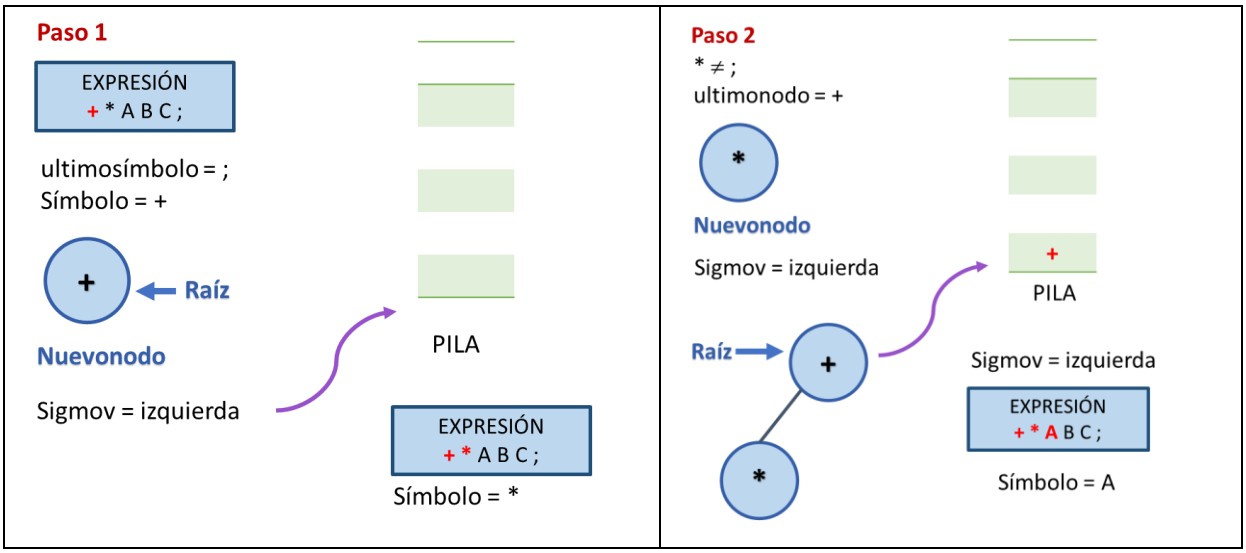
```
Nuevo (Nuevonodo)
Nuevonodo.info ← símbolo
Raíz ← Nuevonodo
Sigmov ← izquierda
LimpiarPila (Pila)
ObtenerSímb (CadenaPref, Símbolo)              //Obtiene el siguiente símbolo

// Añadir el símbolo siguiente al árbol
Mientras (símbolo < > ultimosímbolo) hacer
        // Guardar puntero al nodo anterior
        ultimonodo ← Nuevonodo
        //Construir el nuevo nodo
        Nuevo (Nuevonodo)
        Nuevonodo.info ← símbolo
        //Añadir el nuevo nodo al árbol
        Si (sigmov = izquierda)
          entonces   // Añadir a la izquierda del nodo (ultimonodo)
            {   ultimonodo.izquierda ← Nuevonodo
                Meter (Pila, ultimonodo)     }
           Sino   // Añadir a la derecha de Nodo (ultimonodo)
            {   Sacar (Pila, ultimonodo)
                 ultimonodo.derecha ← Nuevonodo   }
            Fin-si

      // Reinicializar sigmov según el tipo de símbolo
      Si ((símbolo = '+') ó (símbolo = '-') ó (símbolo = '*') ó (símbolo = '/'))
        Entonces  // es un operador
            sigmov ← izquierda
```

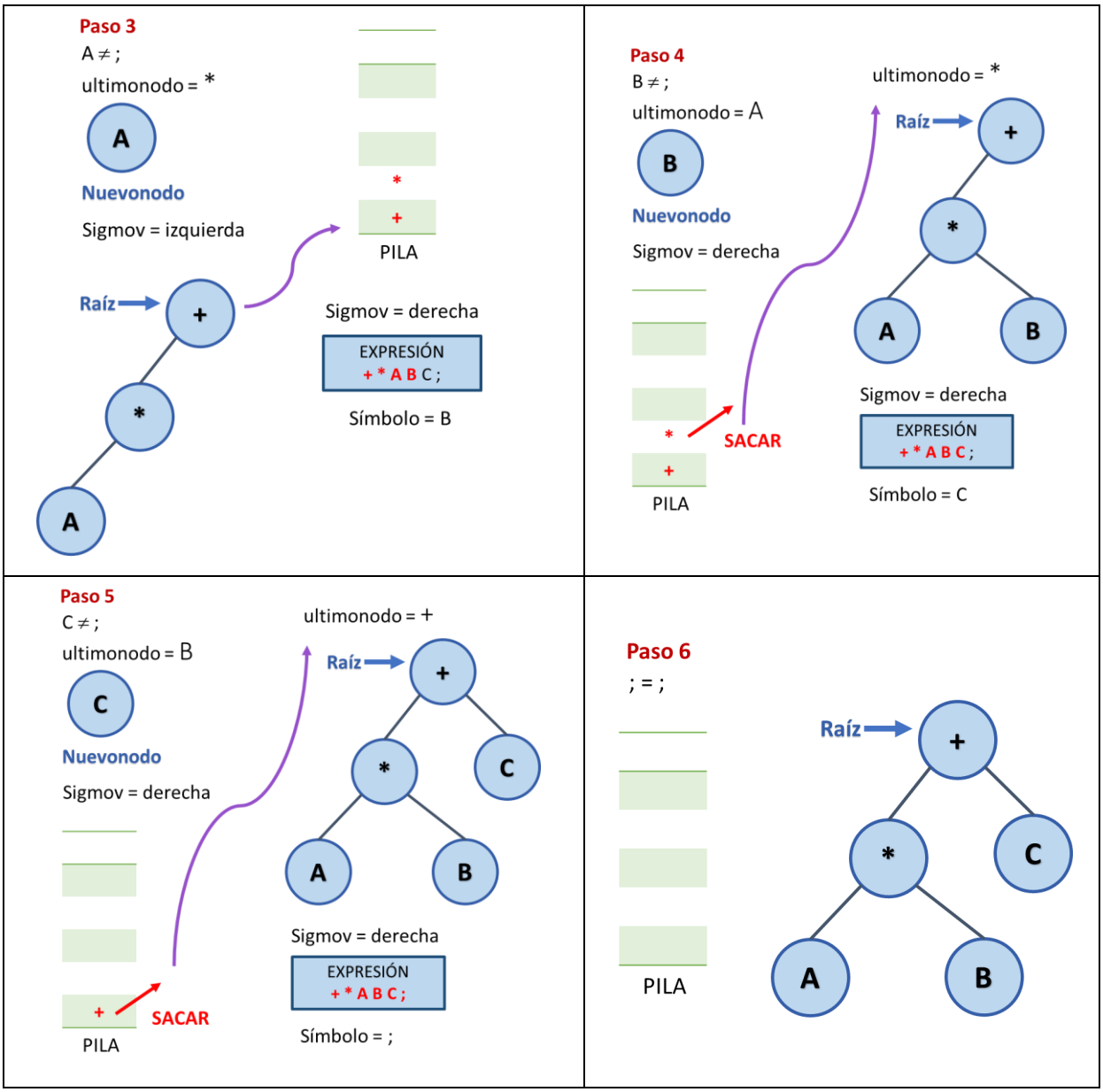
```
        Sino  // es un  operando
            {  Nuevonodo.izquierda ← nulo
               Nuevonodo.derecha ← nulo
               sigmov ← derecha   }
        Fin-si
        ObtenerSímb (CadenaPref, Símbolo)
Fin-Mientras
}
```

Ejemplo

Utilizando el Algoritmo ConstruirArbol dado en el material complementario, construya el árbol binario de expresión con la expresión prefija: + * A B C ;

Solución





1.2 Estructura de datos tipo grafo

1.2.1 Definición y conceptos

Un grafo está formado por un conjunto de nodos (llamados también vértices) y un conjunto de líneas llamadas aristas (o arcos) que conectan los diferentes nodos.

Definición formal de un grafo:

G = (V, A)

donde

- **V(G)** es un conjunto finito, no vacío de vértices que se especifican listando los nodos en notación conjunto, es decir, dentro de paréntesis o llaves.
- **A(G)** es un conjunto de aristas (pares de vértices) que se específica listando una secuencia de aristas. Cada arista se denota escribiendo los nombres de los dos nodos que conecta entre paréntesis, con una coma entre ellos.

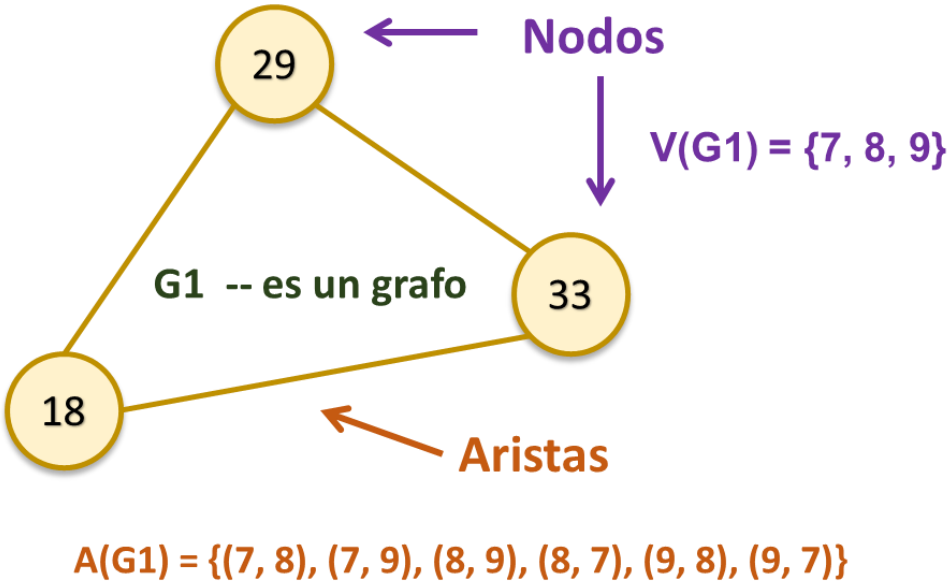


Figura 24. Grafo no dirigido G1. Elaboración propia.

❖ Clasificación de los grafos

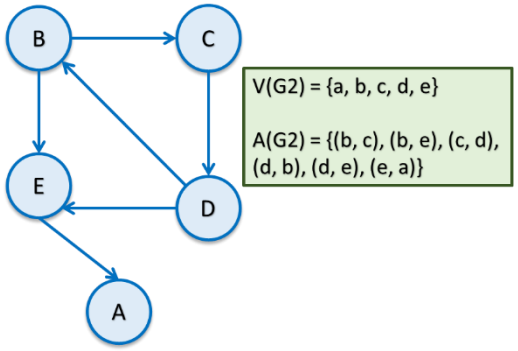


Figura 25. Grafo dirigido G2. Elaboración propia.

Grafo dirigido: la dirección de la línea es indicada por el nodo que se lista primero.

Grafo no dirigido: la relación entre los dos nodos es desordenada. Es decir, un nodo apunta al otro; ellos están simplemente conectados.

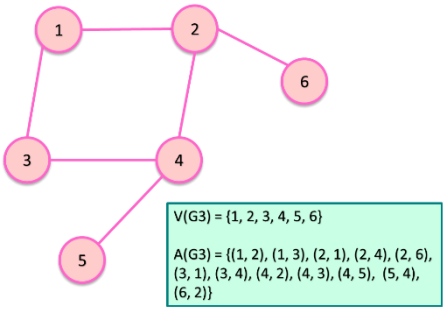


Figura 26. Grafo no dirigido G3. Elaboración propia.

❖ Tipos particulares de grafos:

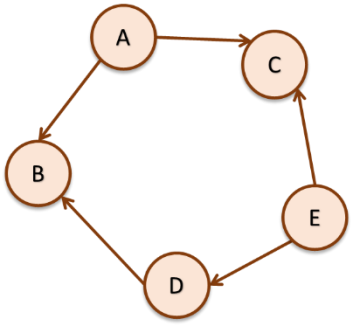


Figura 27. Grafo acíclico. Elaboración propia.

Grafo acíclico: es aquel grafo no contiene ningún ciclo simple.

Grafo cíclico: un grafo se dice cíclico si contiene algún ciclo simple.

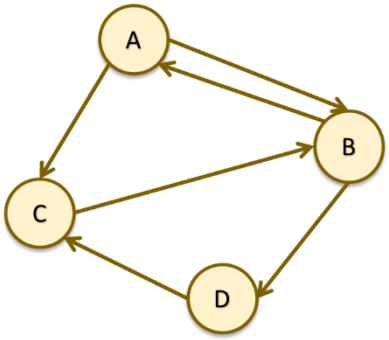


Figura 28. Grafo cíclico. Elaboración propia.

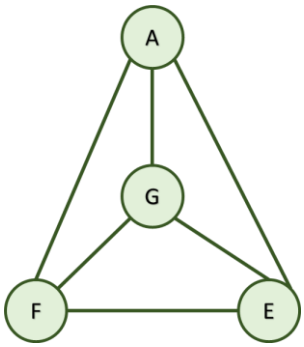


Figura 29. Grafo regular. Elaboración propia

Grafo regular: es un grafo cuyos vértices tienen el mismo grado.

Grafo plano: es un grafo que es posible dibujar en el plano sin que ningún par de aristas se crucen entre sí.

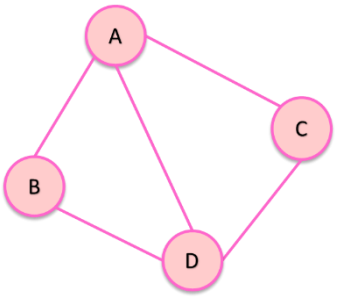


Figura 30. Grafo plano. Elaboración propia.

Grafo simple o simplemente grafo: es aquel que acepta una sola una arista uniendo dos vértices cualesquiera. Esto es equivalente a decir que una arista cualquiera es la única que une dos vértices específicos. Es la definición estándar de un grafo.

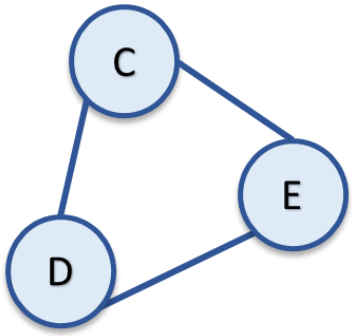


Figure 31. Grafo simple. Elaboración propia.

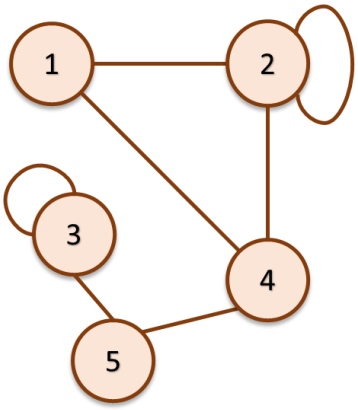


Figura 32. Multigrafo. Elaboración propia.

Multigrafo: son grafos que aceptan más de una arista entre dos vértices. Estas aristas se llaman múltiples o lazos. Los grafos simples son una subclase de esta categoría de grafos.

Grafo vacío: es el grafo cuyo conjunto de aristas es vacío.

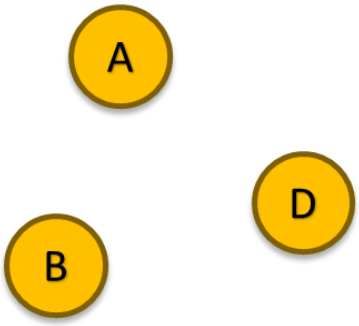


Figura 33. Grafo vacío. Elaboración propia.

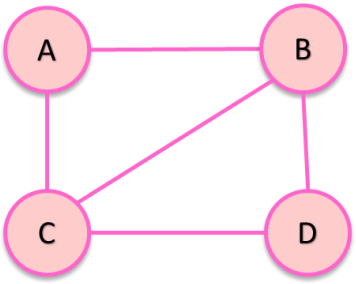


Figura 34. Grafo completo. Elaboración propia.

Grafo completo: un grafo es completo si cada vértice tiene un grado igual a n-1, donde n es el número de vértice que compone el grafo. Además, es un grafo simple en el que cada vértice es adyacente a cualquier todo otro vértice.

Grafo conexo: decimos que es un grafo conexo, si es posible formar un camino desde cualquier vértice a cualquier otro en el grafo.

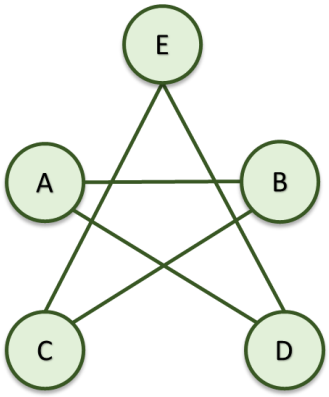


Figura 35. Grafo conexo. Elaboración propia.

Grafo bipartito: es cualquier grafo, cuyos vértices pueden ser divididos en dos conjuntos, tal que no haya aristas entre los vértices del mismo conjunto. Se ve que un grafo es bipartito si no hay ciclos de longitud impar.

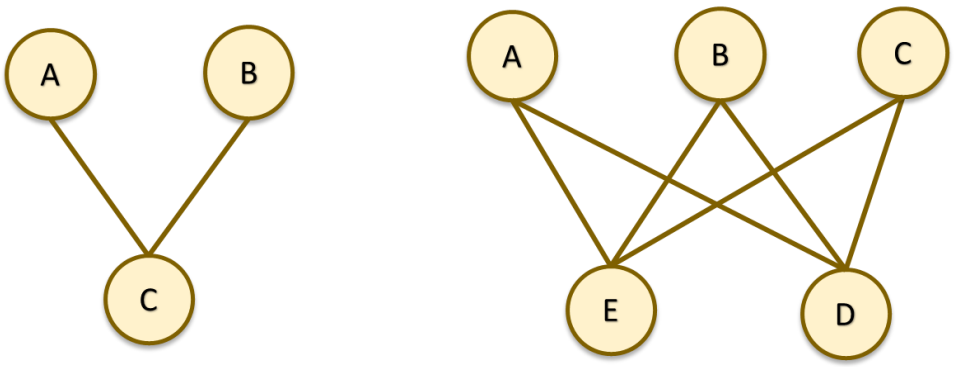


Figura 36. Grafo bipartito. Elaboración propia.

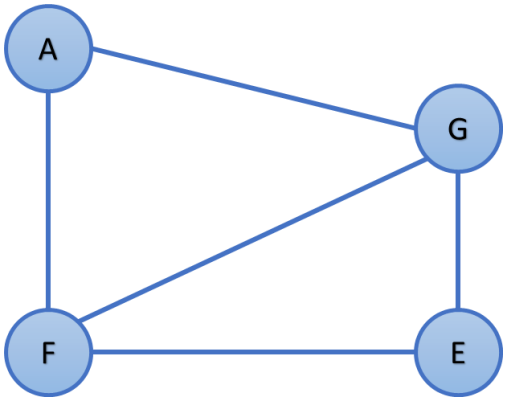


Figura 37. Grafo denso. Elaboración propia.

Grafo denso: un grafo denso es aquel grafo en el que el número de aristas está cercano al número de máximo de aristas.

Grafo con peso: es un grafo en el que cada arista transporta un valor. Los grafos con pesos se usan para representar situaciones en las que el valor de la conexión entre los vértices es importante, no sólo la existencia de la conexión.

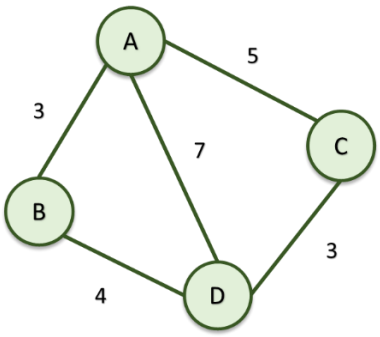


Figura 38. Grafo no dirigido con peso en las aristas. Elaboración propia.

❖ Otras definiciones

Grado total de un vértice: corresponde al número de aristas incidentes sobre el vértice, en donde, cada bucle lo cuenta dos veces. En base a esta definición podemos mencionar:

Vértice fuente: es un vértice con grado de entrada cero.

Vértice sumidero: es un vértice con grado de salida cero.

Vértice hoja: es un vértice con grado uno.

Vértice aislado: tiene grado cero.

Grado total de un vértice (Gr) en un grafo no dirigido: es igual al número de aristas que tiene el vértice.

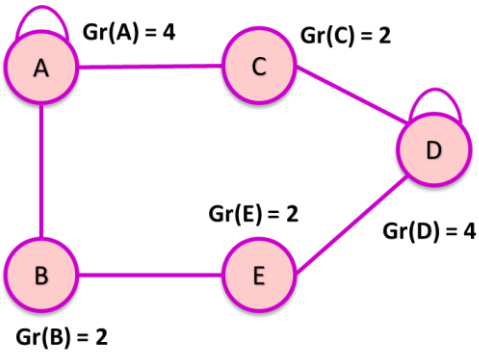


Figura 39. Grado de un vértice en un grafo. Elaboración propia.

En un grafo dirigido distinguimos entre grado de entrada (Ge) y grado de salida (Gs) de un vértice. A continuación, se explican cada uno de ellos.

El grado de entrada (Ge) de un vértice en un grafo dirigido: es el número de aristas que llegan al vértice.

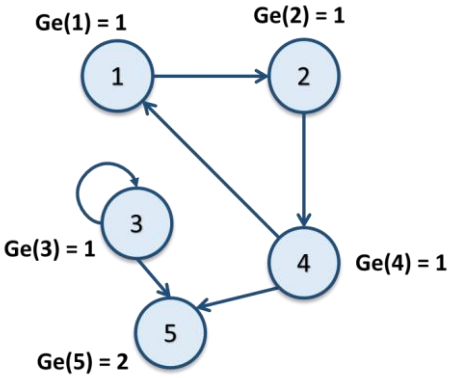


Figura 40. Grado de entrada de un vértice en un grafo dirigido. Elaboración propia.

El grado de salida (Gs) de un vértice en un grafo dirigido: corresponde al número de aristas que salen del vértice.

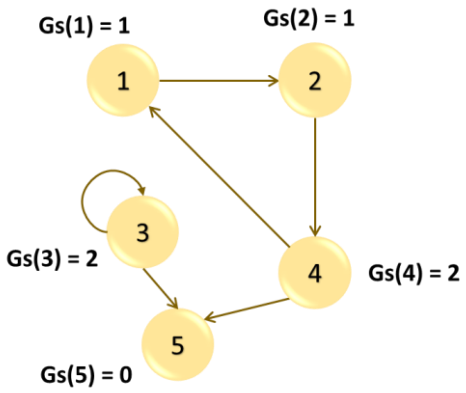


Figura 41. Grado de salida de un vértice en un grafo dirigido. Elaboración propia.

El grado total de un vértice (Gr) en un grafo dirigido: es la suma del grado entrante más el grado saliente.

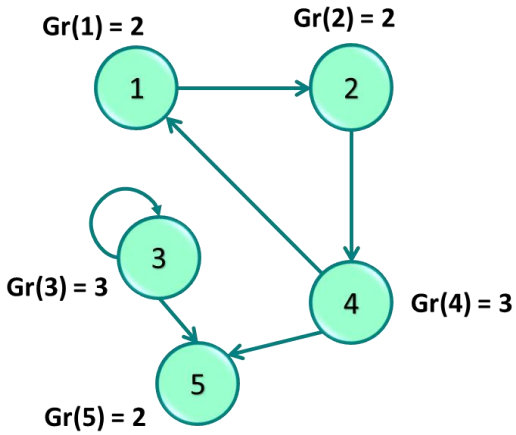


Figura 42. Grado total de un vértice en un grafo dirigido. Elaboración propia.

Camino: es una sucesión de vértices que conecta al vértice V_i con el vértice V_j. Es decir, debe haber una secuencia ininterrumpida de aristas desde V_i a través de cualquier número de nodos hasta V_j.

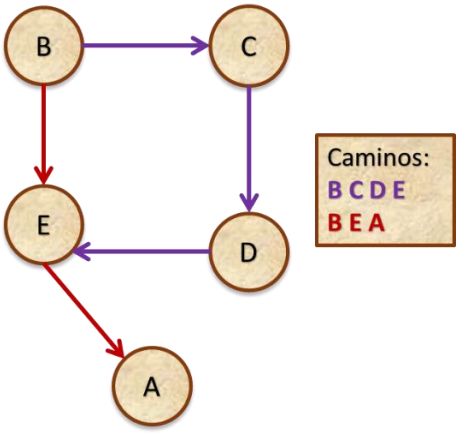


Figura 43. Caminos en el grafo. Elaboración propia.

1.2.2 Representación secuencial de grafos

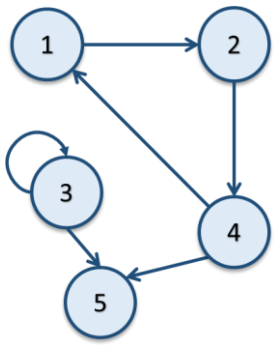
En esta sección se muestran dos formas de representar secuencialmente un grafo: la matriz de adyacencia y la matriz de caminos.

1.2.2.1 Matriz de adyacencia

Para un grafo con N nodos la matriz de adyacencia será una tabla con N filas y N columnas, en donde, el valor en la posición [i, j] de la tabla será 1 si existe una arista (V_i, V_j) perteneciente al conjunto de las aristas A, y 0 en otro caso. Si el grafo es un grafo con peso, la celda [i, j] contendrá el peso de la arista si la arista está en el conjunto A, y 0 en otro caso.

Ejemplo

Escriba la matriz de adyacencia para el siguiente grafo.

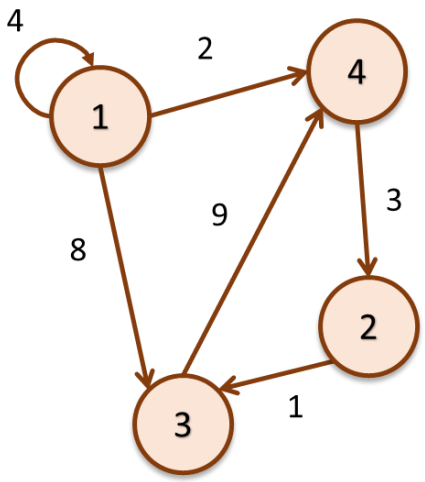


Solución

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	0
3	0	0	1	0	1
4	1	0	0	0	1
5	0	0	0	0	0

Ejemplo

Escriba la matriz de adyacencia para el siguiente grafo con peso.



Solución

	1	2	3	4
1	4	0	8	2
2	0	0	1	0
3	0	0	0	9
4	0	3	0	0

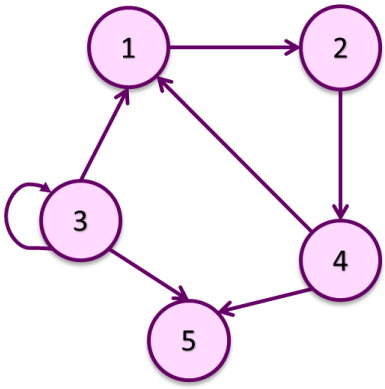
1.2.2.2 Matriz de caminos

Sea G un grafo dirigido simple con *m* nodos v_1, v_2 … v_m. La **matriz de caminos** o **matriz de alcance** de G es la matriz *m*-cuadrada P= V(i, j) definida como sigue:

$$P = \begin{cases} \mathbf{1} & \text{si hay un camino desde } V_i \text{ hasta } V_j \\ \mathbf{0} & \text{en otro caso} \end{cases}$$

Ejemplo

Escriba la matriz de caminos para el siguiente grafo.



Solución

	1	2	3	4	5
1	1	1	0	1	1
2	1	1	0	1	1
3	1	0	1	0	1
4	1	1	0	1	1
5	0	0	0	0	0

1.2.3 Algoritmo de Warshall (caminos mínimos)

Dado un grafo G(V, A) dirigido se puede aplicar el algoritmo de Warshall para resolver el problema de si hay o no algún camino que una a dos vértices cualquiera. Para esto

necesitamos que el valor de cada arista del grafo sea 0 o 1 indicando si existe camino o no entre los dos vértices que la definen.

Algoritmo de Warshall (Caminos Mínimos)

m = cantidad de nodos

{

Desde (i=1, i > m, i=i+1)

Desde (j=1, j>m, j=j+1)

Si (w[i, j] = 0)

Entonces Q_0 [i, j] = ∞

Sino Q_0 [i, j] = w[i, j]

Fin-si

Fin-desde

Fin-desde

Desde (k=1, k > m, k=k+1)

Desde (i=1, i > m, i=i+1)

Desde (j=1, j>m, j=j+1)

Q_k [i, j] = MIN (Q_{k-1} [i, j], Q_{k-1} [i, k] + Q_{k-1} [k, j])

Fin-desde

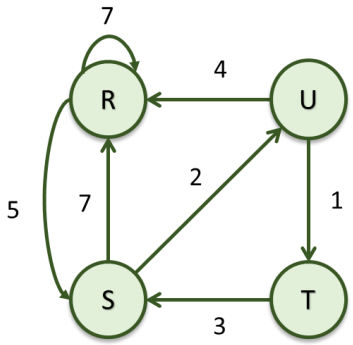
Fin-desde

Fin-desde

}

Ejemplo:

Encuentre la matriz de caminos mínimos para el siguiente grafo con peso. Se asume que v_1 = R, v_2 = S, v_3 = T y v_4 = U.



Solución

Paso 1

Primero se obtiene la matriz de pesos del grafo

$$W = \begin{pmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{pmatrix}$$

Paso 2

Utilizando la sección que se muestra del algoritmo de Warshall se obtiene a partir de la matriz de peso W la matriz Q_0

m = 4

```
m = cantidad de nodos
{
  Desde (i=1, i > m, i=i+1)
     Desde (j=1, j>m, j=j+1)
          Si (w[i, j] = 0)
               Entonces Q0 [i, j] = ∞
               Sino Q0 [i, j] = w[i, j]
          Fin-si
     Fin-desde
  Fin-desde
```

$$Q_0 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{pmatrix}$$

Nota: lo que hace esta parte del algoritmo es cambiar los ceros (0) por infinito (∞).

Paso 3

Utilizando la sección que se muestra del algoritmo de Warshall se obtiene la matriz Q_1 a partir de la matriz Q_0

```
Desde (k=1, k > m, k=k+1)
        Desde (i=1, i > m, i=i+1)
            Desde (j=1, j>m, j=j+1)
                Qk [i, j] = MIN (Qk-1 [i, j], Q k-1 [i, k] + Q k-1 [k, j])
            Fin-desde
        Fin-desde
Fin-desde
```

Note que la instrucción:

Q_k [i, j] = MIN (Q_{k-1} [i, j], Q $_{k-1}$ [i, k] + Q $_{k-1}$ [k, j])

Elige el valor mínimo resultante entre el valor ubicado en la posición Q_{k-1} [i, j] y el resultado de la suma de los dos valores ubicados en las posiciones Q $_{k-1}$ [i, k] + Q $_{k-1}$ [k, j]

Se mostrarán por fila los valores obtenidos para la matriz Q_1 utilizando el algoritmo de Warshall. Los cambios en los valores de la matriz se pueden observar en color rojo.

Calculando la fila 1 de la matriz Q_1

K	I	J	$Q_k [i, j] = MIN (Q_{k-1} [i, j], \quad Q_{k-1} [i, k] + Q_{k-1} [k, j])$
1	1	1	$Q_1 [1,1] = MIN (Q_0 [1,1], \quad Q_0 [1, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	1	2	$Q_1 [1,2] = MIN (Q_0 [1,2], \quad Q_0 [1, 1] + Q_0 [1, 2]) = MIN (5, 7+5) = 5$
1	1	3	$Q_1 [1,3] = MIN (Q_0 [1,3], \quad Q_0 [1, 1] + Q_0 [1, 3]) = MIN (\infty, 7+\infty) = \infty$
1	1	4	$Q_1 [1,4] = MIN (Q_0 [1,4], \quad Q_0 [1, 1] + Q_0 [1, 4) = MIN (\infty, 7+\infty) = \infty$

$$Q_1 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ & & & \end{pmatrix}$$

Calculando la fila 2 de la matriz Q_1

K	I	J	$Q_k [i, j] = MIN (Q_{k-1} [i, j], \quad Q_{k-1} [i, k] + Q_{k-1} [k, j])$
1	1	1	$Q_1 [1,1] = MIN (Q_0 [1,1], \quad Q_0 [1, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	1	2	$Q_1 [1,2] = MIN (Q_0 [1,2], \quad Q_0 [1, 1] + Q_0 [1, 2]) = MIN (5, 7+5) = 5$
1	1	3	$Q_1 [1,3] = MIN (Q_0 [1,3], \quad Q_0 [1, 1] + Q_0 [1, 3]) = MIN (\infty, 7+\infty) = \infty$
1	1	4	$Q_1 [1,4] = MIN (Q_0 [1,4], \quad Q_0 [1, 1] + Q_0 [1, 4) = MIN (\infty, 7+\infty) = \infty$
1	2	1	$Q_1 [2,1] = MIN (Q_0 [2,1], \quad Q_0 [2, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	2	2	$Q_1 [2,2] = MIN (Q_0 [2,2], \quad Q_0 [2, 1] + Q_0 [1, 2]) = MIN (\infty, 7+5) = 12$
1	2	3	$Q_1 [2,3] = MIN (Q_0 [2,3], \quad Q_0 [2, 1] + Q_0 [1, 3]) = MIN (\infty, 7+\infty) = \infty$
1	2	4	$Q_1 [2,4] = MIN (Q_0 [2,4], \quad Q_0 [2, 1] + Q_0 [1, 4]) = MIN (2, 7+\infty) = 2$

$$Q_1 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & 12 & \infty & 2 \\ & & & \end{pmatrix}$$

Calculando la fila 3 de la matriz Q_1

K	I	J	$Q_k [i, j] = MIN (Q_{k-1} [i, j], \quad Q_{k-1} [i, k] + Q_{k-1} [k, j])$
1	1	1	$Q_1 [1,1] = MIN (Q_0 [1,1], \quad Q_0 [1, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	1	2	$Q_1 [1,2] = MIN (Q_0 [1,2], \quad Q_0 [1, 1] + Q_0 [1, 2]) = MIN (5, 7+5) = 5$
1	1	3	$Q_1 [1,3] = MIN (Q_0 [1,3], \quad Q_0 [1, 1] + Q_0 [1, 3]) = MIN (\infty, 7+\infty) = \infty$
1	1	4	$Q_1 [1,4] = MIN (Q_0 [1,4], \quad Q_0 [1, 1] + Q_0 [1, 4) = MIN (\infty, 7+\infty) = \infty$
1	2	1	$Q_1 [2,1] = MIN (Q_0 [2,1], \quad Q_0 [2, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	2	2	$Q_1 [2,2] = MIN (Q_0 [2,2], \quad Q_0 [2, 1] + Q_0 [1, 2]) = MIN (\infty, 7+5) = 12$
1	2	3	$Q_1 [2,3] = MIN (Q_0 [2,3], \quad Q_0 [2, 1] + Q_0 [1, 3]) = MIN (\infty, 7+\infty) = \infty$
1	2	4	$Q_1 [2,4] = MIN (Q_0 [2,4], \quad Q_0 [2, 1] + Q_0 [1, 4]) = MIN (2, 7+\infty) = 2$
1	3	1	$Q_1 [3,1] = MIN (Q_0 [3,1], \quad Q_0 [3, 1] + Q_0 [1, 1]) = MIN (\infty, \infty+7) = \infty$
1	3	2	$Q_1 [3,2] = MIN (Q_0 [3,2], \quad Q_0 [3, 1] + Q_0 [1, 2]) = MIN (3, \infty+5) = 3$
1	3	3	$Q_1 [3,3] = MIN (Q_0 [3,3], \quad Q_0 [3, 1] + Q_0 [1, 3]) = MIN (\infty, \infty+\infty) = \infty$
1	3	4	$Q_1 [3,4] = MIN (Q_0 [3,4], \quad Q_0 [3, 1] + Q_0 [1, 4]) = MIN (\infty, \infty+\infty) = \infty$

$$Q_1 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & 12 & \infty & 2 \\ \infty & 3 & \infty & \infty \end{pmatrix}$$

Calculando la fila 4 de la matriz Q_1

K	I	J	$Q_k [i, j] = MIN (Q_{k-1} [i, j], \quad Q_{k-1} [i, k] + Q_{k-1} [k, j])$
1	1	1	$Q_1 [1,1] = MIN (Q_0 [1,1], \quad Q_0 [1, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	1	2	$Q_1 [1,2] = MIN (Q_0 [1,2], \quad Q_0 [1, 1] + Q_0 [1, 2]) = MIN (5, 7+5) = 5$
1	1	3	$Q_1 [1,3] = MIN (Q_0 [1,3], \quad Q_0 [1, 1] + Q_0 [1, 3]) = MIN (\infty, 7+\infty) = \infty$
1	1	4	$Q_1 [1,4] = MIN (Q_0 [1,4], \quad Q_0 [1, 1] + Q_0 [1, 4) = MIN (\infty, 7+\infty) = \infty$
1	2	1	$Q_1 [2,1] = MIN (Q_0 [2,1], \quad Q_0 [2, 1] + Q_0 [1, 1]) = MIN (7, 7+7) = 7$
1	2	2	$Q_1 [2,2] = MIN (Q_0 [2,2], \quad Q_0 [2, 1] + Q_0 [1, 2]) = MIN (\infty, 7+5) = 12$

1	2	3	Q_1 [2,3] = MIN (Q_0 [2,3], Q_0 [2, 1] + Q_0 [1, 3]) = MIN (∞, 7+∞) = ∞
1	2	4	Q_1 [2,4] = MIN (Q_0 [2,4], Q_0 [2, 1] + Q_0 [1, 4]) = MIN (2, 7+∞) = 2
1	3	1	Q_1 [3,1] = MIN (Q_0 [3,1], Q_0 [3, 1] + Q_0 [1, 1]) = MIN (∞, ∞+7) = ∞
1	3	2	Q_1 [3,2] = MIN (Q_0 [3,2], Q_0 [3, 1] + Q_0 [1, 2]) = MIN (3, ∞+5) = 3
1	3	3	Q_1 [3,3] = MIN (Q_0 [3,3], Q_0 [3, 1] + Q_0 [1, 3]) = MIN (∞, ∞+∞) = ∞
1	3	4	Q_1 [3,4] = MIN (Q_0 [3,4], Q_0 [3, 1] + Q_0 [1, 4]) = MIN (∞, ∞+∞) = ∞
1	4	1	Q_1 [4,1] = MIN (Q_0 [4,1], Q_0 [4, 1] + Q_0 [1, 1]) = MIN (4, 4+7) = 4
1	4	2	Q_1 [4,2] = MIN (Q_0 [4,2], Q_0 [4, 1] + Q_0 [1, 2]) = MIN (∞, 4+5) = 9
1	4	3	Q_1 [4,3] = MIN (Q_0 [4,3], Q_0 [4, 1] + Q_0 [1, 3]) = MIN (1, 4+∞) = 1
1	4	4	Q_1 [4,4] = MIN (Q_0 [4,4], Q_0 [4, 1] + Q_0 [1, 4]) = MIN (∞, 4+∞) = ∞

$$Q_1 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & 12 & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & 9 & 1 & \infty \end{pmatrix}$$

Paso 4

Aplicando el algoritmo de Warshall mostrado anteriormente, se obtienen las siguientes matrices Q_2, Q_3 y Q_4 que son mostradas a continuación.

$$Q_2 = \begin{pmatrix} 7 & 5 & \infty & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & 9 & 1 & 11 \end{pmatrix}$$

$$Q_3 = \begin{pmatrix} 7 & 5 & \infty & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & 4 & 1 & 6 \end{pmatrix}$$

$$Q_4 = \begin{pmatrix} 7 & 5 & 8 & 7 \\ 6 & 6 & 3 & 2 \\ 9 & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{pmatrix}$$

1.2.4 Representación enlazada de grafos

Otra forma de representar los grafos es utilizando las listas enlazadas.

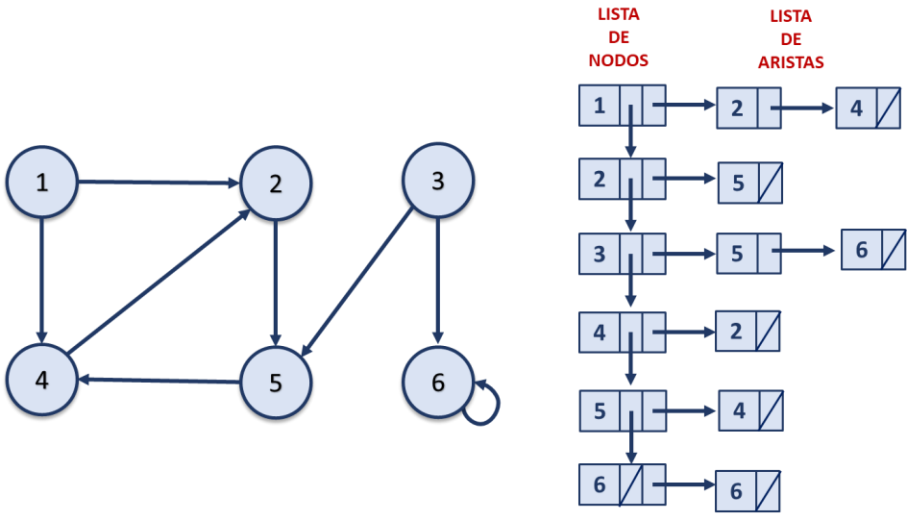


Figura 44. Representación enlazada de grafos. Elaboración propia.

En las listas enlazadas tenemos:

- **Lista de nodos:** lista que contiene los nombres de los nodos. Está formada por tres partes:
  - **Info:** el nombre o valor clave del nodo

- **Sig:** puntero al siguiente nodo de la lista nodo
- **Ady:** puntero al primer elemento de la lista de adyacencia del nodo que se mantiene en la lista de aristas

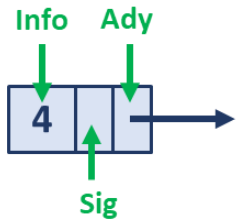


Figura 45. Representación de las partes de la lista de nodos. Elaboración propia.

- **Lista de aristas:** que contiene la(s) arista(s) a la(s) que el nodo está conectado. Está formada por dos partes:
  - **Dest:** campo que apunta al nodo destino de la arista
  - **Enl:** campo que enlaza las aristas del mismo nodo inicial

Figura 46. Representación de las partes de la lista de aristas. Elaboración propia.

En los grafos con peso se debe agregar un campo adicional en la lista de las aristas en donde se colocará el peso de las mismas.

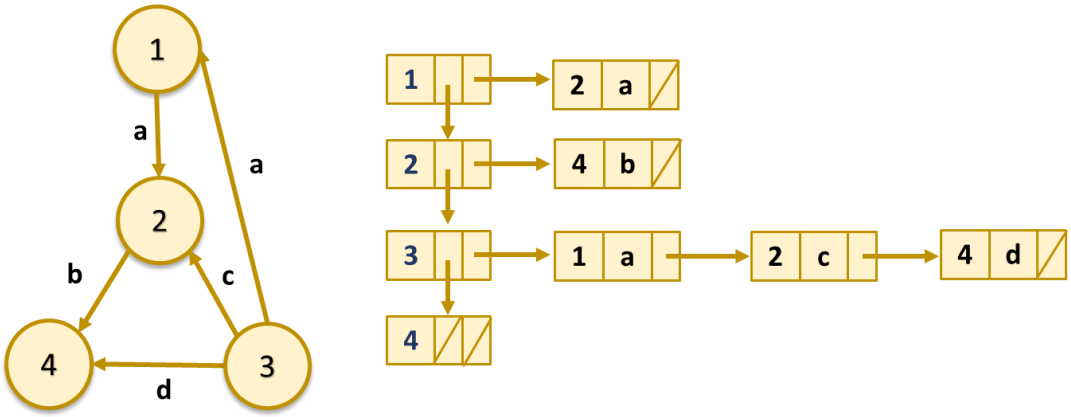


Figura 47. Representación enlazada de grafos con peso. Elaboración propia.

1.2.5 Operaciones sobre grafos

Dos operaciones que son comunes realizar sobre los grafos son añadir o insertar un nodo o arista al grafo y eliminar ya sea un nodo o una arista. En esta sección se explican estas operaciones utilizando los algoritmos que realizan dicha operación. Se inicia el tema presentado el funcionamiento de la búsqueda con el algoritmo de búsqueda.

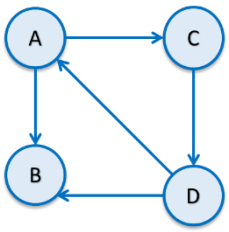
1.2.5.1 Búsqueda

Este algoritmo busca en la lista de nodos, un nodo cuyo valor se encuentra en la variable **ítem**. Al finalizar el algoritmo, se enviará un mensaje indicando si se encontró o no el nodo.

```
Algoritmo de Búsqueda
Inicio
Leer (ítem)  /* lectura del valor a buscar*/
pos = 0
ptr = principio
Mientras (ptr ≠ nulo) hacer
Si (ptr.info = ítem) entonces
                    pos = ptr
                    ptr = 0
          sino
                    ptr = ptr.sig
          fin si
fin mientras
si (pos = nulo) entonces
          imprimir ("No se ha encontrado el valor")
sino
          imprimir ("El valor:", ítem, "ha sido encontrado")
fin si
fin
```

Ejemplo:

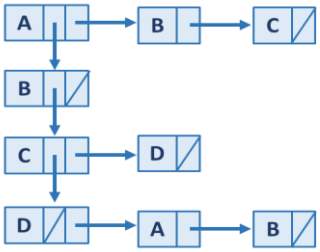
Busque el valor de C el siguiente grafo utilizando el algoritmo de búsqueda. La lista de nodos inicia en el grafo con principio = A.



Solución:

PASO 1

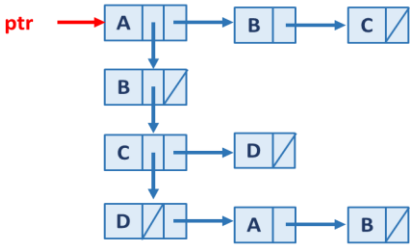
Hacer la representación enlazada del grafo



PASO 2

Leer (ítem)
pos = 0
ptr = principio

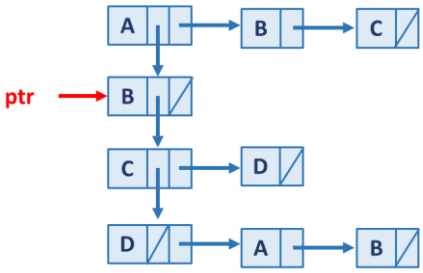
Item = C
Pos = 0



PASO 3

```
Mientras (ptr ≠ nulo) hacer
 Si (ptr.info = ítem) entonces
      pos = ptr
      ptr = 0
 sino
      ptr = ptr.sig
 fin si
fin mientras
```

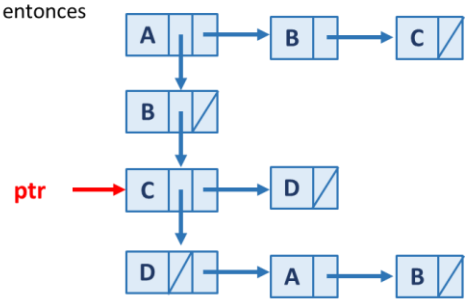
ptr ≠ nulo
A ≠ C



PASO 4

```
Mientras (ptr ≠ nulo) hacer
 Si (ptr.info = ítem) entonces
      pos = ptr
      ptr = 0
 sino
      ptr = ptr.sig
 fin si
fin mientras
```

ptr ≠ nulo
B ≠ C



PASO 5

```
Mientras (ptr ≠ nulo) hacer
 Si (ptr.info = ítem) entonces
      pos = ptr
      ptr = 0
 sino
      ptr = ptr.sig
 fin si
fin mientras
```

ptr ≠ nulo
C = C
pos = C
ptr = 0

PASO 6

```
si (pos = nulo) entonces
   imprimir ("No se ha encontrado el valor")
sino
   imprimir ("El valor:", ítem, "ha sido encontrado")
fin si
fin
```

pos ≠ nulo
El valor: C ha sido encontrado

1.2.5.2 Inserción

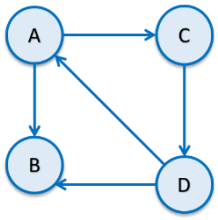
El algoritmo de inserción primero verifica si existen los nodos origen y destino en el grafo, en caso contrario los inserta en el grafo para luego proceder a insertar la arista correspondiente a dichos nodos.

Algoritmo de Inserción

```
inicio
/* lectura del vértice origen (ítemA) y destino (ítemB) */
leer(ítemA, ítemB)
/* búsqueda del vértice origen (ítemA) en el grafo */
pos = nulo
ptr = principio
Mientras (ptr ≠ nulo) hacer
Si (ptr.info = ítemA) entonces
                pos = ptr
                ptr = nulo
        sino
                ptr = ptr.enl
        fin si
fin mientras
si (pos = nulo) entonces
/* crea el nodo origen y lo agrega a la lista de nodos */
nuevo(nodo)
nodo.info = ítemA
nodo.enl = principio
nodo.sig = nulo
principio = nodo
pos = nodo
fin si
/* búsqueda del vértice destino (ítemB) en el grafo */
pos1 = nulo
ptr1 = principio
Mientras (ptr1 ≠ nulo) hacer
                pos1 = ptr1
                ptr1 = nulo
        sino
                ptr1 = ptr1.enl
        fin si
fin mientras
si (pos1 = nulo) entonces
/* crea el nodo destino y lo agrega a la lista de nodos */
nuevo(nodo)
nodo.info = ítemB
nodo.enl = principio
nodo.sig = nulo
principio = nodo
fin si
/* verifica si la arista existe, sino existe crea la arista y la
agrega a la lista de aristas */
mientras (pos.sig ≠ ítemB y pos.sig ≠ nulo) hacer
        pos = pos.sig
fin mientras
si (pos.sig = nulo) entonces
/* crea la arista y la agrega a la lista de aristas */
        nueva(arista)
        arista.info = ítemB
        arista.sig = nulo
        pos.sig = arista
fin si
fin
```

Ejemplo:

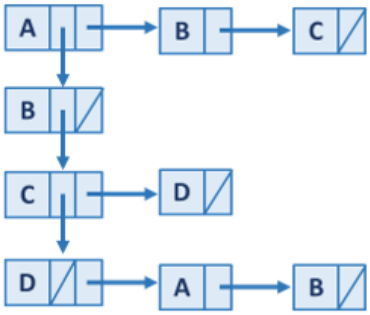
Insertar la arista (B, E) el siguiente grafo utilizando el algoritmo de inserción. La lista de nodos inicia en el grafo con principio = A.



Solución:

PASO 1

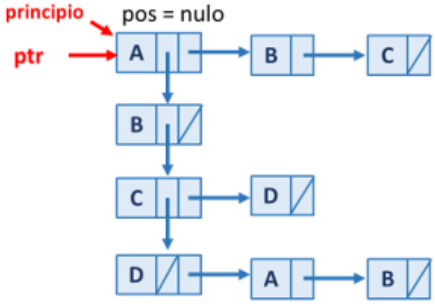
Hacer la representación enlazada del grafo



PASO 2

```
/* lectura del vértice origen
(ítemA) y destino (ítemB) */
leer(ítemA, ítemB)
/* búsqueda del vértice origen
(ítemA) en el grafo */
pos = nulo
ptr = principio
```

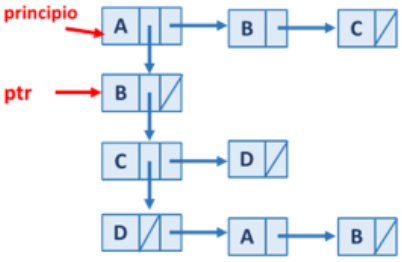
itemA = B
itemB = E
pos = nulo



PASO 3

```
Mientras (ptr ≠ nulo) hacer
  Si (ptr.info = ítemA) entonces
     pos = ptr
     ptr = nulo
  sino
     ptr = ptr.enl
  fin si
fin mientras
```

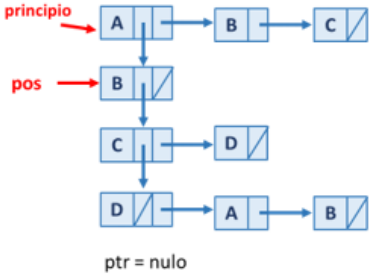
ptr ≠ nulo
A ≠ B



PASO 4

```
Mientras (ptr ≠ nulo) hacer
  Si (ptr.info = ítemA) entonces
     pos = ptr
     ptr = nulo
  sino
     ptr = ptr.enl
  fin si
fin mientras
```

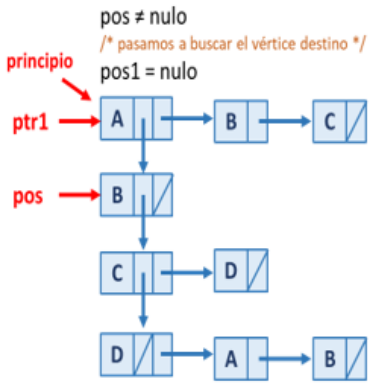
ptr ≠ nulo
B = B



ptr = nulo

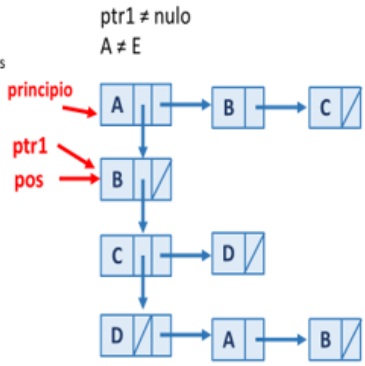
PASO 5

```
si (pos = nulo) entonces
/* crea el nodo origen y lo agrega
a la lista de nodos */
  nuevo(nodo)
  nodo.info = ítemA
  nodo.enl = principio
  nodo.sig = nulo
  principio = nodo
  pos = nodo
fin si
/* búsqueda del vértice destino
(ítemB) en el grafo */
pos1 = nulo
ptr1 = principio
```



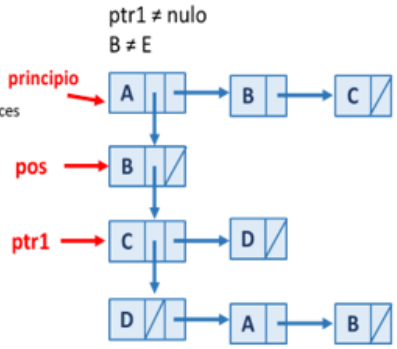
PASO 6

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
     pos1 = ptr1
     ptr1 = nulo
  sino
     ptr1 = ptr1.enl
  fin si
fin mientras
```



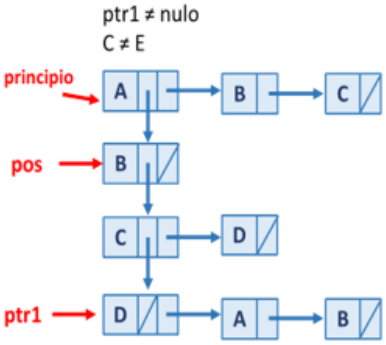
PASO 7

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
     pos1 = ptr1
     ptr1 = nulo
  sino
     ptr1 = ptr1.enl
  fin si
fin mientras
```



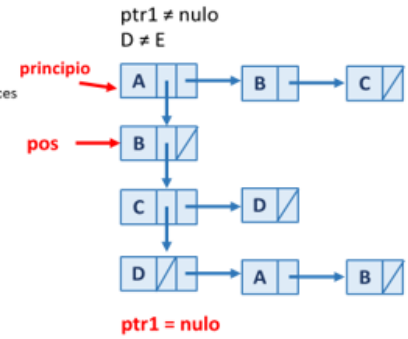
PASO 8

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
     pos1 = ptr1
     ptr1 = nulo
  sino
     ptr1 = ptr1.enl
  fin si
fin mientras
```



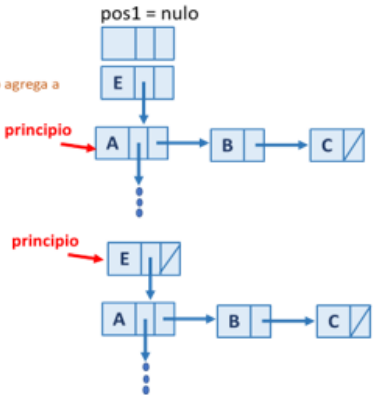
PASO 9

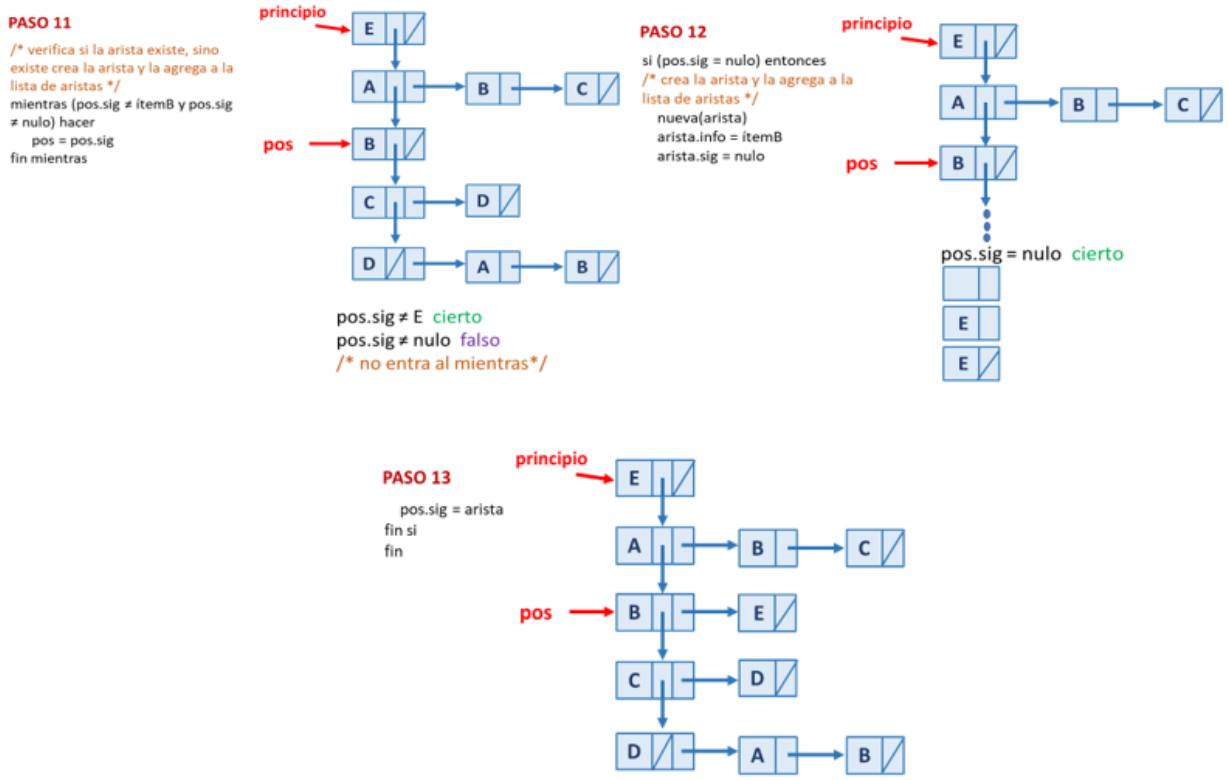
```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
     pos1 = ptr1
     ptr1 = nulo
  sino
     ptr1 = ptr1.enl
  fin si
fin mientras
```



PASO 10

```
si (pos1 = nulo) entonces
/* crea el nodo destino y lo agrega a
la lista de nodos */
  nuevo(nodo)
  nodo.info = ítemB
  nodo.enl = principio
  nodo.sig = nulo
  principio = nodo
fin si
```





1.2.5.3 Eliminación

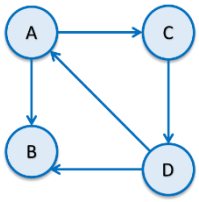
Para eliminar una arista del grafo debemos encontrar la posición del vértice origen y del destino, luego se busca la arista en la lista de aristas del nodo origen y se elimina de la misma.

Algoritmo de eliminación de una arista en el grafo

```
inicio
/* ítemA contiene el vértice origen, ítemB contiene el
vértice destino */
leer(ítemA, ítemB)
/* búsqueda del vértice origen */
pos = nulo
ptr = principio
mientras (ptr ≠ nulo) hacer
        si(ptr.info = ítemA) entonces
                pos = ptr
                ptr = nulo
        sino
                ptr = ptr.sig
        fin si
fin mientras
/* búsqueda de la arista */
pos1 = pos.ady
ptr1 = pos
mov = 0
mientras (pos1.dest ≠ ítemB y pos1.enl ≠ nulo) hacer
    ptr1 = pos1
    pos1 = pos1.enl
    mov = 1
fin mientras

si (pos1.dest = ítemB) entonces
  si (mov = 0) entonces
     ptr1.ady = pos1.enl
  sino
     ptr1.enl = pos1.enl
  fin si
fin si
liberar (pos1)
fin
```

Ejemplo:

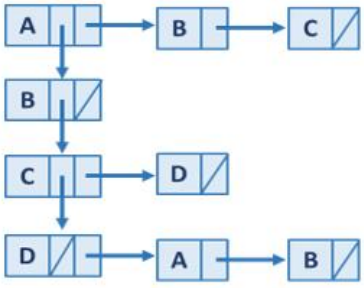
Eliminar la arista (A, B) el siguiente grafo utilizando el algoritmo de eliminación de una arista. La lista de nodos inicia en el grafo con principio = A.



Solución:

PASO 1

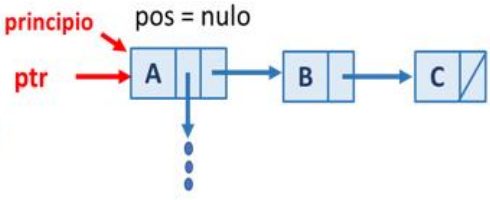
Hacer la representación enlazada del grafo



PASO 2

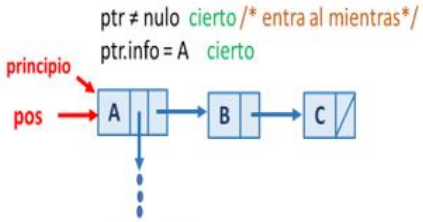
```
inicio
/* ítemA contiene el vértice
origen, ítemB contiene el vértice
destino */
leer(ítemA, ítemB)
/* búsqueda del vértice origen */
pos = nulo
ptr = principio
```

itemA = A
itemB = B
pos = nulo



PASO 3

```
mientras (ptr ≠ nulo) hacer
  si(ptr.info = ítemA) entonces
     pos = ptr
     ptr = nulo
  sino
     ptr = ptr.sig
  fin si
fin mientras
```



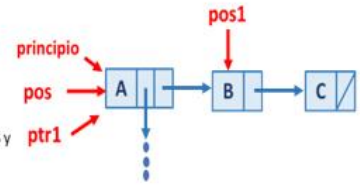
ptr ≠ nulo cierto /* entra al mientras*/
ptr.info = A cierto

ptr = nulo

ptr ≠ nulo falso /* sale del mientras*/

PASO 4

```
/* búsqueda de la arista */
pos1 = pos.ady
ptr1 = pos
mov = 0
mientras (pos1.dest ≠ ítemB y
pos1.enl ≠ nulo) hacer
    ptr1 = pos1
    pos1 = pos1.enl
    mov = 1
fin mientras
```

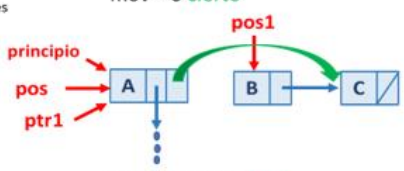


mov = 0
pos1.dest ≠ B falso
pos.enl ≠ nulo cierto
/* no entra al mientras*/

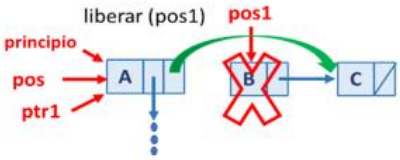
PASO 5

```
si (pos1.dest = ítemB) entonces
   si (mov = 0) entonces
      ptr1.ady = pos1.enl
   sino
      ptr1.enl = pos1.enl
   fin si
fin si
liberar (pos1)
fin
```

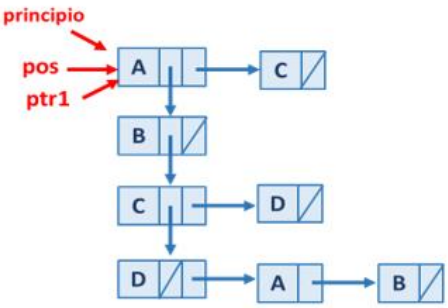
pos1.dest = B cierto
mov = 0 cierto



liberar (pos1)



PASO 6

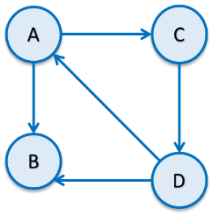


Algoritmo de eliminación de un nodo del grafo

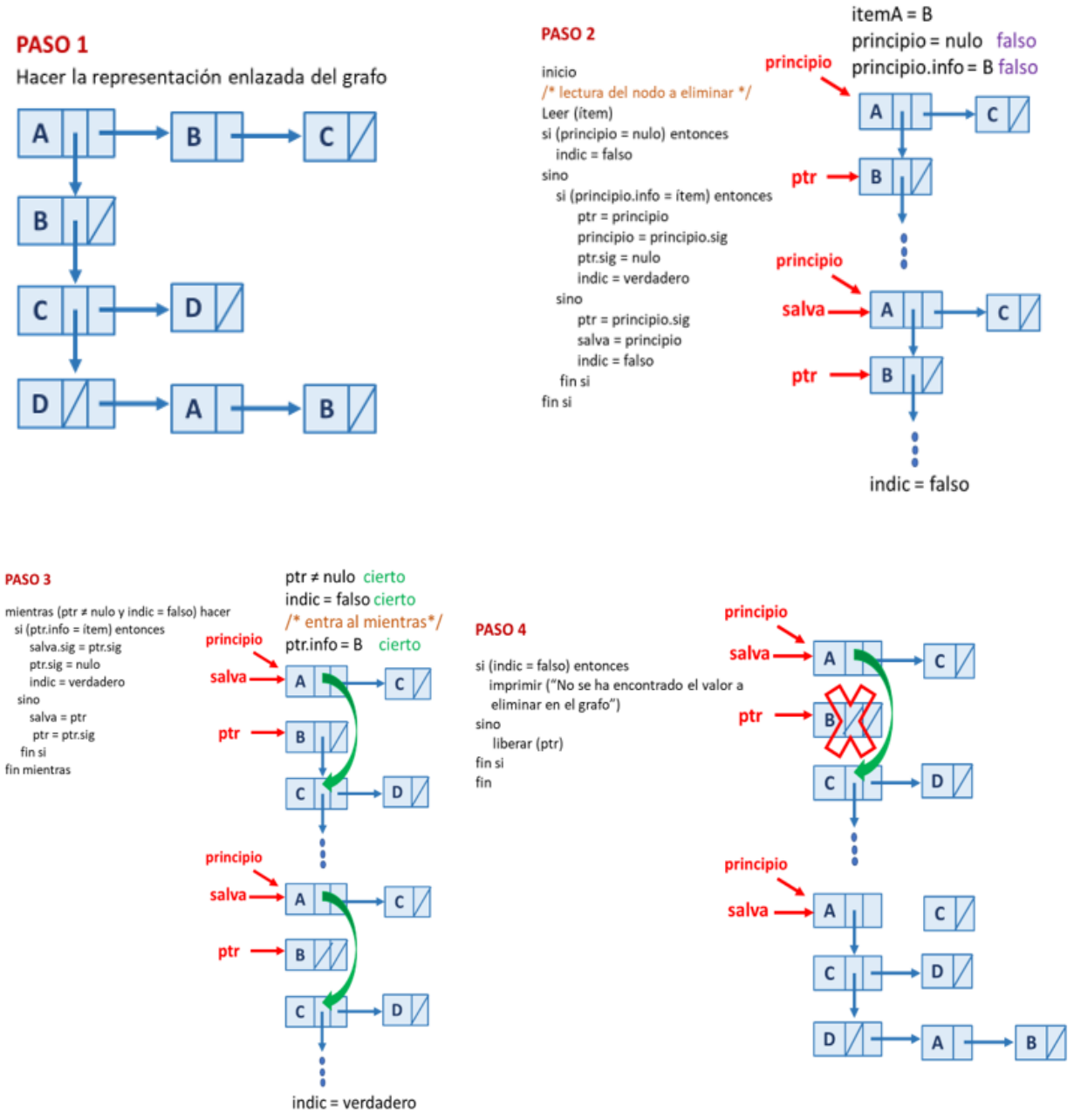
```
inicio
/* lectura del nodo a eliminar */
Leer (ítem)
inicio
/* lectura del nodo a eliminar */
Leer (ítem)
si (principio = nulo) entonces
   indic = falso
sino
   si (principio.info = ítem) entonces
        ptr = principio
        principio = principio.sig
        ptr.sig = nulo
        indic = verdadero
   sino
        ptr = principio.sig
        salva = principio
        indic = falso
    fin si
fin si
mientras (ptr ≠ nulo y indic = falso) hacer
  si (ptr.info = ítem) entonces
      salva.sig = ptr.sig
      ptr.sig = nulo
      indic = verdadero
  sino
      salva = ptr
       ptr = ptr.sig
   fin si
fin mientras
si (indic = falso) entonces
   imprimir ("No se ha encontrado el valor a eliminar en
   el grafo")
sino
    liberar (ptr)
fin si
fin
```

Ejemplo:

Eliminar el nodo B en el siguiente grafo utilizando el algoritmo de eliminación de un nodo. La lista de nodos inicia en el grafo con principio = A.



Solución:



1.2.6 Recorridos en un grafo

Hay dos enfoques básicos de recorridos: el recorrido en anchura y el recorrido en profundidad. A continuación, se explican cada uno de estos recorridos.

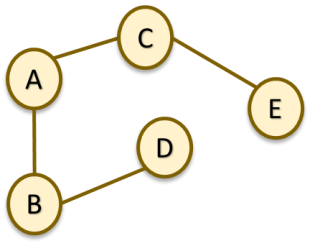
1.2.6.1 Anchura

En este recorrido se empieza en un nodo v: primero se visita v, luego se visitan todos sus adyacentes. Luego los adyacentes de estos y así sucesivamente. El algoritmo utiliza una cola de vértices. Las operaciones básicas que se utilizarán en la cola son:

- Sacar un elemento de la cola.
- Añadir a la cola sus adyacentes no visitados.

Ejemplo:

Realizar el recorrido en anchura en el siguiente grafo. La lista de nodos inicia en el grafo con Actual = A.



Solución:

PASO 1

Hacer un arreglo de visitados del grafo
Marcar cada nodo como no_visitado

NODOS	A	B	C	D	E
VISITADOS	F	F	F	F	F

Crear la Cola

COLA

PASO 2

Actual = A
Marcar actual como visitado

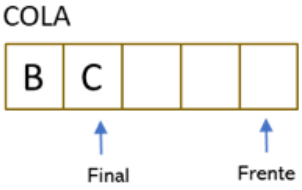
NODOS	A	B	C	D	E
VISITADOS	C	F	F	F	F

PASO 3

Marcar todos los vecinos de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	F

COLA

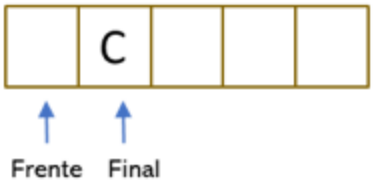


RECORRIDO:
A

PASO 4

Sacar de la Cola y asignar a Actual

COLA

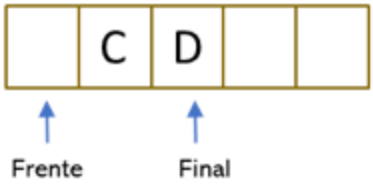


Actual = B

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	F

COLA



RECORRIDO:

A B

PASO 5

Sacar de la Cola y asignar a Actual

COLA

Actual = C

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C

COLA

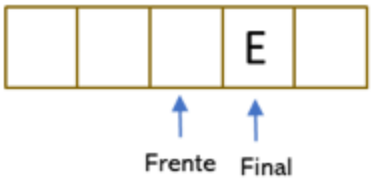
RECORRIDO:

A B C

PASO 6

Sacar de la Cola y asignar a Actual

COLA

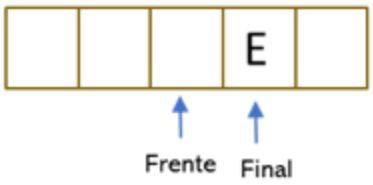


Actual = D

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C

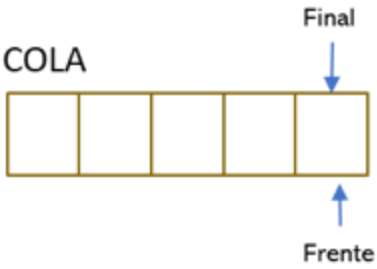
COLA



RECORRIDO:
A B C D

PASO 7

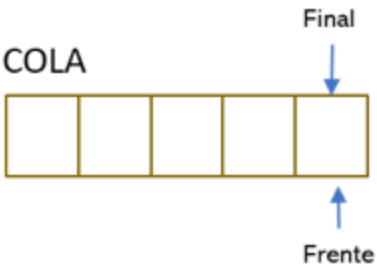
Sacar de la Cola y asignar a Actual



Actual = E

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C



RECORRIDO:
A B C D E

1.2.6.2 Profundidad

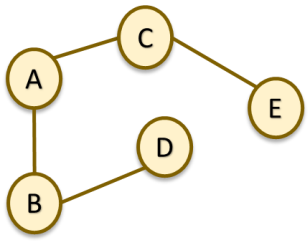
Consiste en alejarse todo lo posible del nodo origen para después empezar a visitar los nodos restantes a la vuelta.

En este tipo de recorrido hay que usar una pila para ir almacenando los nodos que nos falta visitar. Las operaciones básicas que se utilizarán en la pila son:

- Sacar un elemento de la pila.
- Añadir un elemento a la pila.

Ejemplo:

Realizar el recorrido en profundidad en el siguiente grafo. La lista de nodos inicia en el grafo con Actual = A.



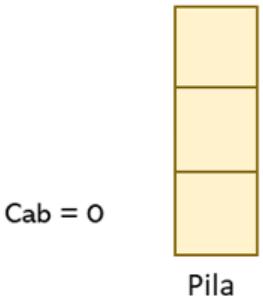
Solución:

PASO 1

Hacer un arreglo de visitados del grafo
Marcar cada nodo como no_visitado

NODOS	A	B	C	D	E
VISITADOS	F	F	F	F	F

Crear la Pila



PASO 2

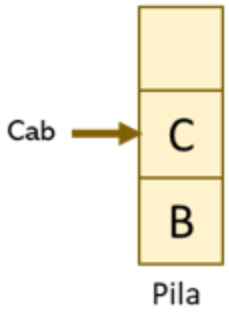
Actual = A
Marcar actual como visitado

NODOS	A	B	C	D	E
VISITADOS	C	F	F	F	F

PASO 3

Marcar todos los vecinos de actual como visitados y los metemos en la Pila

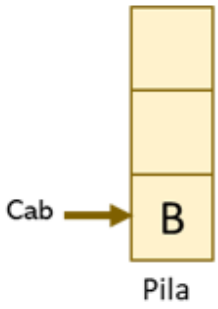
NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	F



RECORRIDO:
A

PASO 4

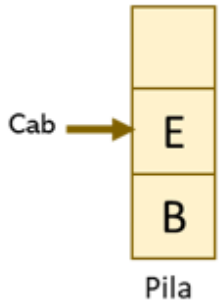
Sacar de la Pila y asignar a Actual



Actual = C

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

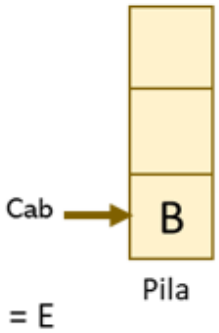
NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	C



RECORRIDO:
A C

PASO 5

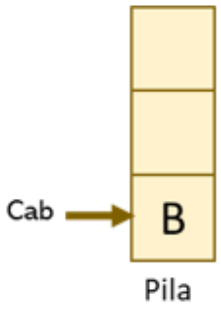
Sacar de la Pila y asignar a Actual



Actual = E

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

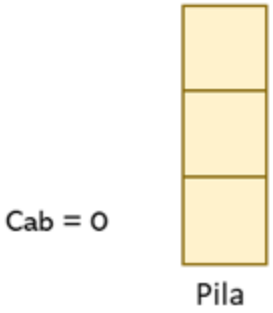
NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	C



RECORRIDO:
A C E

PASO 6

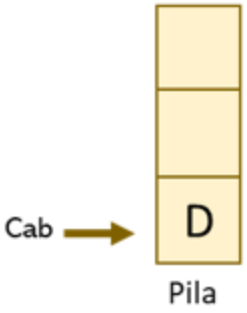
Sacar de la Pila y asignar a Actual



Actual = B

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

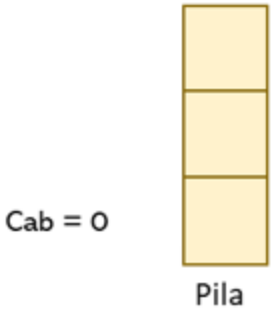
NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C



RECORRIDO:
A C E B

PASO 7

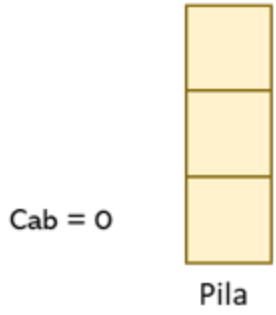
Sacar de la Pila y asignar a Actual



Actual = D

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C



RECORRIDO:
A C E B D

Capítulo II

2.1 Algoritmos de ordenamiento y búsqueda

2.1.1 Definición y conceptos

Para resolver un problema pueden existir varios algoritmos. Por tanto, es lógico elegir el "mejor". Si el problema es sencillo o no hay que resolver muchos casos, se podría elegir el más "fácil". Si el problema es complejo o existen muchos casos habría que elegir el algoritmo que menos recursos utilice. Los recursos más importantes son el tiempo de ejecución y el espacio de almacenamiento. Generalmente, el más importante es el tiempo.

Al hablar de la eficiencia de un algoritmo nos referiremos a lo "rápido" que se ejecuta. La eficiencia de un algoritmo dependerá, en general, del "tamaño" de los datos de entrada. La eficiencia no tiene unidades de medida y se usa la notación O(n) para describirla.

A la hora de analizar un algoritmo es necesario saber que pueden darse tres tipos de casos:

- **Caso mejor:** se trata de aquellos ejemplares del problema en los que el algoritmo es más eficiente; por ejemplo: multiplicar un número por cero, insertar en una lista vacía, ordenar un vector que ya está ordenado, etc. Generalmente este caso no nos interesa.
- **Caso peor:** se trata de aquellos ejemplares del problema en los que el algoritmo es menos eficiente (no siempre existe el caso peor). Ejemplos: insertar al final de una lista, ordenar un vector que está ordenado en orden inverso, etc. Este caso nos interesa mucho.
- **Caso medio:** se trata del resto de ejemplares del problema. Por ejemplo: multiplicar dos números enteros distintos de cero, insertar en una lista que no sea

el principio ni el final, ordenar un vector que no está ordenado ni en orden directo ni inverso, etc. Es el caso que más nos debería preocupar puesto que será el más habitual, sin embargo, no siempre se puede calcular.

2.1.2 Notación de la Gran O (Big O)

Podemos definir la eficiencia de un algoritmo como una función t(n). A la hora de analizar un algoritmo nos interesa, principalmente, la forma en que se comporta el algoritmo al aumentar el tamaño de los datos; es decir, cómo aumenta su tiempo de ejecución. Esto se conoce como eficiencia asintótica de un algoritmo y nos permitirá comparar distintos algoritmos puesto que deberíamos elegir aquellos que se comportarán mejor al crecer los datos.

La notación asintótica se describe por medio de una función cuyo dominio es el conjunto de números naturales, N. Ésta se describe mediante la notación O.

Se describe la notación O como:

$$O(g(n)) = \{f(n) \mid \exists\ c > 0, n_0 > 0,0 \quad \leq f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0\}$$

$$f(n): Z^+ \rightarrow R^+$$

$$g(n): Z^+ \rightarrow R^+$$

Los órdenes más utilizados en análisis de algoritmos, en orden creciente, son los siguientes (donde c representa una constante y n el tamaño de la entrada):

notación	***nombre***
O(1)	orden constante
O(log log n)	orden sublogarítmico
O(log n)	orden logarítmico
$O(\sqrt{n})$	orden sublineal
O(n)	orden lineal
O(n · log n)	orden lineal logarítmico
$O(n^c)$	orden potencial
$O(c^n)$, n > 1	orden exponencial
O(n!)	orden factorial
$O(n^n)$	orden potencial exponencial

Ejemplo:

Calcule el orden de complejidad para el siguiente algoritmo.

```
para i de 1 a n
  para j de 1 a n
     escribir i + j
  fin para
fin para
```

Solución:

Se calcularía como:

Paso 1:

```
para i de 1 a n
  para j de 1 a n
    escribir i + j
  fin para
fin para
```

➡ *O(1) -- **La complejidad de esta sentencia es***

Paso 2:

```
para i de 1 a n
  para j de 1 a n
    escribir i + j
  fin para
fin para
```

➡ ***O(n·1) = O(n) -- La complejidad del bucle es el producto del número de ejecuciones por***

Paso 3:

```
para i de 1 a n
  para j de 1 a n
     escribir i + j
  fin para
fin para
```

➡ ***O(n·n) = O(n^2) -- La complejidad del bucle es el producto del número de ejecuciones por***

Llegándose a la conclusión de que la complejidad es O(n^2).

2.1.3 Algoritmos de ordenamiento y su eficiencia

El ordenamiento o clasificación consiste en organizar los datos en orden ascendente o descendente. Se le denomina clave al elemento en base al cual se está ordenado un conjunto de datos. Este procedimiento organiza los datos en una secuencia que facilita la búsqueda.

Dos criterios se utilizan para evaluar la eficiencia de un algoritmo de ordenamiento:

- Tiempo necesario para ordenar los datos proporcionados.
- Espacio de memoria requerido para hacerlo.

A continuación, se tratarán los diferentes métodos de ordenamiento con sus respectivas eficiencias.

2.1.3.1 Selección

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo. Este algoritmo tiene una complejidad de $O(n^2)$.

```
Algoritmo OrdSelección
{
// n = número de elementos del arreglo
entero i, j, mindice, Aux;
i = 1;
Mientras (i<n) hacer
{  mindice = i;
   j = i+1;
   // Encontrar el índice del mínimo elemento
desordenado
   Mientras (j <= n) hacer
   {  si (Datos [ j ] < Datos [ mindice ]) entonces
        mindice = j;
      fin-si
      j = j + 1;
    }  // Fin-mientras
// Intercambiar el primer elemento desordenado con
el mínimo elemento desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
          i = i + 1;
}  // Fin-mientras
}
```

Ejemplo:

Ordene mediante el algoritmo de selección el siguiente arreglo.

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

Solución:

PASO 1

```
// n = número de elementos del arreglo
entero i, j, mindice, Aux;
i = 1;
Mientras (i<n) hacer
{  mindice = i;
   j = i+1;
   // Encontrar el índice del mínimo elemento desordenado
```

n = 5
i = 1
1 < 5 cierto
mindice = 1;
j = i+1 = 1 + 1 = 2

PASO 2

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

2 <= 5 cierto
5 < 9 cierto
mindice = 2

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 2+1 = 3

PASO 3

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

3 <= 5 cierto
4 < 9 cierto
mindice = 3

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 3+1 = 4

PASO 4

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

4 <=5 cierto
12 < 9 falso

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 4+1 = 5

PASO 5

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

5 <= 5 cierto
2 < 9 cierto
mindice = 5

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 5 +1 = 6
6 <= 5 falso

PASO 6

```
// Intercambiar el primer elemento desordenado con el mínimo elemento desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 9
A[1] = 2
A[5] = 9

2	5	4	12	9
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

i = 1 +1 = 2

PASO 7

```
Mientras (i<n) hacer
{ mindice = i;
  j = i+1;
  // Encontrar el índice del mínimo elemento
desordenado
  Mientras (j <= n) hacer
    { si (Datos [j] < Datos [mindice])
    entonces
       mindice = j;
     fin-si
      j = j + 1;
    } // Fin-mientras
```

2 < 5 cierto
mindice = 2;
j = i+1 = 2 + 1 = 3
3 <= 5 cierto
4 < 5 cierto
mindice = 3

j = 3 +1 = 4

PASO 8

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
     mindice = j;
    fin-si
     j = j + 1;
  } // Fin-mientras
```

4 <= 5 cierto
12 < 4 falso

j = 4 +1 = 5

PASO 9

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
     mindice = j;
    fin-si
     j = j + 1;
  } // Fin-mientras
```

5 <= 5 cierto
9 < 4 falso

j = 5 +1 = 6
6 <= 5 falso

PASO 10

```
// Intercambiar el primer elemento
desordenado con el mínimo elemento
desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 5
A[2] = 4
A[3] = 5

i = 2 +1 = 3

PASO 11

```
Mientras (i<n) hacer
{ mindice = i;
  j = i+1;
  // Encontrar el índice del mínimo elemento
desordenado
  Mientras (j <= n) hacer
    { si (Datos [j] < Datos [mindice])
    entonces
       mindice = j;
     fin-si
      j = j + 1;
    } // Fin-mientras
```

3 < 5 cierto
mindice = 3;
j = i+1 = 3 + 1 = 4
4 <= 5 cierto
12 < 5 falso

j = 4 +1 = 5

PASO 12

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
     mindice = j;
    fin-si
     j = j + 1;
  } // Fin-mientras
```

5 <= 5 cierto
9 < 5 falso

j = 5 +1 = 6
6 <= 5 falso

PASO 13

```
// Intercambiar el primer elemento desordenado con el mínimo elemento desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 5
A[3] = 5
A[3] = 5

i = 3 +1 = 4

PASO 14

```
Mientras (i<n) hacer
{ mindice = i;
  j = i+1;
  // Encontrar el índice del mínimo elemento desordenado
  Mientras (j <= n) hacer
    { si (Datos [j] < Datos [mindice])
    entonces
      mindice = j;
    fin-si
    j = j + 1;
    } // Fin-mientras
```

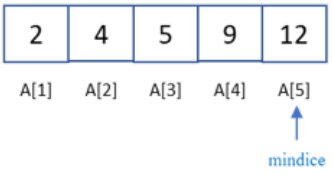
4 < 5 cierto
mindice = 4;
j = i+1 = 4 + 1 = 5
4 <= 5 cierto
9 < 12 cierto
mindice = 5

j = 5 +1 = 5
6 <= 5 falso

PASO 15

```
// Intercambiar el primer elemento desordenado con el mínimo elemento desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 12
A[4] = 9
A[5] = 12



i = 4 +1 = 5
5 < 5 falso

2.1.3.2 Inserción

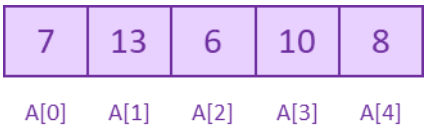
En este tipo de algoritmo los elementos que van a ser ordenados son considerados uno a la vez. Cada elemento es insertado en la posición apropiada con respecto al resto de los elementos ya ordenados.

Este procedimiento recibe el arreglo de datos a ordenar a[] y altera las posiciones de sus elementos hasta dejarlos ordenados de menor a mayor. N representa el número de elementos que contiene a[]. Los arreglos comienzan en la posición 0. Este algoritmo tiene una complejidad de $O(n^2)$.

```
Algoritmo Inserción
Inicio
entero i, j, n, A[];
Desde (i=1; i < n; i = i + 1)
   j= i
   Aux = A[i]
   Mientras (j > 0 y Aux  < A[j-1] )
      // Intercambiar el elemento
      A[ j ] = A[ j-1 ]
      j=j-1
   Fin Mientras
   A[ j ] = Aux;
Fin Desde
Fin
```

Ejemplo:

Ordene mediante el algoritmo de inserción el siguiente arreglo.



Solución:

PASO 1

```
Desde (i=1; i < n; i = i + 1)
   j= i
   Aux = A[i]
   Mientras (j > 0 y Aux  < A[j-1] )
      // Intercambiar el elemento
      A[ j ] = A[ j-1 ]
      j=j-1
   Fin Mientras
   A[ j ] = Aux;
Fin Desde
```

n = 5
i = 1
1 < 5 cierto
j = 1
Aux = 13
1 > 0 y 13 < 7 falso
A[1] = 13

PASO 2

```
Desde (i=1; i < n; i = i + 1)
   j= i
   Aux = A[i]
   Mientras (j > 0 y Aux  < A[j-1] )
      // Intercambiar el elemento
      A[ j ] = A[ j-1 ]
      j=j-1
   Fin Mientras
   A[ j ] = Aux;
Fin Desde
```

i = 2
2 < 5 cierto
j = 2
Aux = 6
2 > 0 y 6 < 13 cierto
A[2] = 13

j = 1

PASO 3

```
Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

1 > 0 y 6 < 7 cierto
A[1] = 7

7	7	13	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

j = 0
0 > 0 falso
A[0] = 6

6	7	13	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 4

```
Desde (i=1; i < n; i = i + 1)
  j= i
  Aux = A[i]
  Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

i = 3
3 < 5 cierto
j = 3
Aux = 10
3 > 0 y 10 < 13 cierto
A[3] = 13

6	7	13	13	8
A[0]	A[1]	A[2]	A[3]	A[4]

j = 2

PASO 5

```
Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

2 > 0 y 10 < 7 falso
A[2] = 10

6	7	10	13	8
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 6

```
Desde (i=1; i < n; i = i + 1)
  j= i
  Aux = A[i]
  Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

i = 4
4 < 5 cierto
j = 4
Aux = 8
4 > 0 y 8 < 13 cierto
A[4] = 13

6	7	10	13	13
A[0]	A[1]	A[2]	A[3]	A[4]

j = 3

PASO 7

```
Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

3 > 0 y 8 < 10 cierto
A[3] = 10

7	7	10	10	13
A[0]	A[1]	A[2]	A[3]	A[4]

j = 2
2 > 0 y 8 < 7 falso
A[2] = 8

6	7	8	10	13
A[0]	A[1]	A[2]	A[3]	A[4]

i = 5
5 < 5 falso

2.1.3.3 Burbuja

En el ordenamiento por burbuja se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados tomando el elemento mayor y recorriendo de posición en posición hasta ponerlo en su lugar. Esto se hace hasta que el arreglo este ordenado. Esta operación se hace n − 1 pasadas y n − 1 comparaciones en cada pasada,

por lo tanto, el número de comparaciones es (n − 1) * (n − 1) = n2 − 2n + 1, es decir, la complejidad es 0(n^2).

```
Algoritmo Burbuja
Inicio
i = 1;
cambiado=cierto
Mientras ((i < n) y (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
  // sube la burbuja del valor más pequeño no ordenado
  Mientras (j > i)  hacer
    // si el valor es menor que su predecesor, intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
     j = j - 1
   Fin-mientas
    i = i +1
  Fin-mientras
Fin
```

Ejemplo:

Ordene mediante el algoritmo de burbuja el siguiente arreglo.

19	5	16	12	8
A[1]	A[2]	A[3]	A[4]	A[5]

Solución:

PASO 1

```
i = 1;
cambiado=cierto
Mientras ((i < n)  y
   (cambiado = "cierto"))  hacer
   j = n;
  cambiado = falso
// sube la burbuja del valor más pequeño no ordenado
```

n = 5
i = 1
cambiado = cierto
1 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

PASO 2

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor, intercambiarlos.
   Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
        aux = Datos [ j ]
        Datos [ j ] = Datos [ j-1]
        Datos [ j-1] = aux
     Fin-si
   j = j – 1
Fin-mientas
```

5 > 1 cierto
8 < 12 cierto
cambiado = cierto
Aux = 8
A[5] = 12
A[4] = 8

19	5	16	8	12
A[1]	A[2]	A[3]	A[4]	A[5]

j = 4

PASO 3

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
    intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
        aux = Datos [ j ]
        Datos [ j ] = Datos [ j-1]
        Datos [ j-1] = aux
      Fin-si
   j = j – 1
Fin-mientas
```

4 > 1 cierto
8 < 16 cierto
cambiado = cierto
Aux = 8
A[4] = 16
A[3] = 8

19	5	8	16	12
A[1]	A[2]	A[3]	A[4]	A[5]

j = 3

PASO 4

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
    intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
        aux = Datos [ j ]
        Datos [ j ] = Datos [ j-1]
        Datos [ j-1] = aux
      Fin-si
   j = j – 1
Fin-mientas
```

3 > 1 cierto
8 < 5 falso

j = 2

PASO 5

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
    intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
        aux = Datos [ j ]
        Datos [ j ] = Datos [ j-1]
        Datos [ j-1] = aux
      Fin-si
   j = j – 1
Fin-mientas
i = i +1
Fin-mientras
```

2 > 1 cierto
5 < 19 cierto
cambiado = cierto
Aux = 5
A[2] = 19
A[1] = 5

5	19	8	16	12
A[1]	A[2]	A[3]	A[4]	A[5]

j = 1
1 > 1 falso
i = 2

PASO 6

```
Mientras ((i < n) y
  (cambiado = "cierto")) hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

2 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

PASO 7

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
    intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
        aux = Datos [ j ]
        Datos [ j ] = Datos [ j-1]
        Datos [ j-1] = aux
      Fin-si
   j = j – 1
Fin-mientas
```

5 > 2 cierto
12 < 16 cierto
cambiado = cierto
Aux = 12
A[5] = 16
A[4] = 12

5	19	8	12	16
A[1]	A[2]	A[3]	A[4]	A[5]

j = 4

PASO 8

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
    intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
        aux = Datos [ j ]
        Datos [ j ] = Datos [ j-1]
        Datos [ j-1] = aux
      Fin-si
   j = j – 1
Fin-mientas
```

4 > 2 cierto
12 < 8 falso
j = 3

PASO 9

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
       Fin-si
    j = j – 1
Fin-mientas
i = i +1
 Fin-mientras
```

3 > 2 cierto
8 < 19 cierto
cambiado = cierto
Aux = 8
A[3] = 19
A[2] = 8

5	8	19	12	16
A[1]	A[2]	A[3]	A[4]	A[5]

j = 2
2 > 2 falso
i = 3

PASO 10

```
Mientras ((i < n) y
  (cambiado = "cierto")) hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

3 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

PASO 11

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
       Fin-si
    j = j – 1
Fin-mientas
i = i +1
 Fin-mientras
```

5 > 3 cierto
16 < 12 falso
j = 4
4 > 3 cierto
12 < 19 cierto
cambiado = cierto
Aux = 12
A[4] = 19
A[3] = 12

5	8	12	19	16
A[1]	A[2]	A[3]	A[4]	A[5]

j = 3
3 > 3 falso
i = 4

PASO 12

```
Mientras ((i < n) y
  (cambiado = "cierto")) hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

4 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

PASO 13

```
Mientras (j > i) hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
       Fin-si
    j = j – 1
Fin-mientas
i = i +1
 Fin-mientras
```

5 > 4 cierto
16 < 19 cierto
cambiado = cierto
Aux = 16
A[5] = 19
A[4] = 16

5	8	12	16	19
A[1]	A[2]	A[3]	A[4]	A[5]

j = 4
4 > 4 falso
i = 5

PASO 14

```
Mientras ((i < n) y
  (cambiado = "cierto")) hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

5 < 5 y cambiado = cierto falso

2.1.3.4 Quicksort

Aplica la técnica divide y vencerás, al dividir el arreglo a ordenar en dos particiones separadas por un elemento central, denominado pivote o elemento de partición. Una característica de esta partición es que todos los elementos de la partición izquierda son

menores que todos los elementos de la partición derecha, luego se ordenan las dos particiones independientemente. El algoritmo tiene una complejidad de O(n log2 n).

```
Algoritmo quicksort(A, inf, sup)
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
  mientras A[j] > x hacer
    j = j - 1
  fin_mientras
  si i ≤ j entonces
    aux = A[i]
    A[i] = A[j]
    A[j] = aux
    i = i + 1
    j = j - 1
  fin_si
fin_mientras
si inf < j
  quicksort(A, inf, j)
fin_si
si i < sup
  quicksort(A, i, sup)
fin_si
Fin
```

Ejemplo:

Ordene mediante el algoritmo quicksort el siguiente arreglo.

7	10	13	12	8
A[1]	A[2]	A[3]	A[4]	A[5]

Solución:

PASO 1

```
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
```

inf = 1
sup = 5
i = 1
j = 5
x = A[(1+5)/2] = A[3] x = 13
1 ≤ 5 cierto mientras
7 < 13 cierto mientras
i = 2
10 < 13 cierto mientras
i = 3
13 < 13 falso mientras

PASO 2

```
mientras A[j] > x hacer
  j = j - 1
fin_mientras
si i ≤ j entonces
  aux = A[i]
  A[i] = A[j]
  A[j] = aux
  i = i + 1
  j = j - 1
fin_si
fin_mientras
```

8 > 13 falso mientras
1 ≤ 5 cierto
aux = 13
A[3] = 8
A[5] = 13

7	10	8	12	13
A[1]	A[2]	A[3]	A[4]	A[5]

i = 4
j = 4

PASO 3

```
mientras i ≤ j hacer
 mientras A[i] < x hacer
   i = i + 1
 fin_mientras
 mientras A[j] > x hacer
   j = j - 1
 fin_mientras
 si i ≤ j entonces
   aux = A[i]
   A[i] = A[j]
   A[j] = aux
   i = i + 1
   j = j - 1
 fin_si
fin_mientras
```

4 ≤ 4 cierto mientras
12 < 13 cierto mientras
i = 5
13 < 13 falso mientras
12 > 13 falso mientras
5 ≤ 4 falso

PASO 4

```
si inf < j
   quicksort(A, inf, j)
fin_si
si i < sup
   quicksort(A, i, sup)
fin_si
Fin
```

1 < 4 cierto
quicksort(A, 1, 4)

PASO 5

```
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
 mientras A[i] < x hacer
   i = i + 1
 fin_mientras
```

i = 1
j = 4
x = A[(1+4)/2] = A[2]
x = 10
1 ≤ 4 cierto mientras
7 < 10 cierto mientras
i = 2
10 < 10 falso mientras

PASO 6

```
mientras A[j] > x hacer
   j = j - 1
 fin_mientras
 si i ≤ j entonces
   aux = A[i]
   A[i] = A[j]
   A[j] = aux
   i = i + 1
   j = j - 1
 fin_si
fin_mientras
```

12 > 10 cierto mientras
j = 3
8 > 10 falso mientras
2 ≤ 3 cierto
aux = 10
A[2] = 8
A[3] = 10

7	8	10	12	13
A[1]	A[2]	A[3]	A[4]	A[5]

i = 3
j = 2
3 ≤ 2 falso mientras

PASO 7

```
si inf < j
   quicksort(A, inf, j)
fin_si
si i < sup
   quicksort(A, i, sup)
fin_si
Fin
```

1 < 2 cierto
quicksort(A, 1, 2)

PASO 8

```
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
 mientras A[i] < x hacer
   i = i + 1
 fin_mientras
```

i = 1
j = 2
x = A[(1+2)/2] = A[1]
x = 7
1 ≤ 2 cierto mientras
7 < 7 falso mientras

PASO 9

```
mientras A[j] > x hacer
    j = j - 1
fin_mientras
si i ≤ j entonces
    aux = A[i]
    A[i] = A[j]
    A[j] = aux
    i = i + 1
    j = j - 1
fin_si
fin_mientras
```

8 > 7 cierto mientras
j = 1
7 > 7 falso mientras
1 ≤ 1 cierto
aux = 7
A[1] = 7
A[1] = 7

7	8	10	12	13
A[1]	A[2]	A[3]	A[4]	A[5]

i = 2
j = 0
2 ≤ 0 falso mientras

PASO 10

```
si inf < j
    quicksort(A, inf, j)
fin_si
si i < sup
    quicksort(A, i, sup)
fin_si
Fin
```

1 < 0 falso
1 < 0 falso

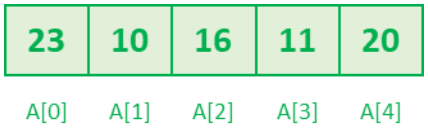
2.1.3.5 Heapsort

El algoritmo tiene un tiempo de ejecución de O(n log n), el mismo consiste en almacenar todos los elementos en un montículo y luego extraer el nodo que queda como raíz en iteraciones sucesivas obteniendo el conjunto ordenado.

```
HeapSort
n = cantidad de elementos del arreglo
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
fin HeapSort

heapify(arr[], m, j)
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

Ejemplo:

Ordene mediante el algoritmo heapsort el siguiente arreglo.



Solución:

PASO 1

```
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
```

n = 5
i = 5/2 - 1
i = 1
1 ≥ 0 cierto
heapify(A, 5, 1)

PASO 2

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 1
leftChild = 3
rightChild = 4
3 < 5 y 11 > 10 cierto
max = 3
4 < 5 y 20 > 11 cierto
max = 4
4 ≠ 1 cierto
swap = 10
A[1] = 20
A[4] = 10

heapify(A, 5, 4)

PASO 3

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 4
leftChild = 9
rightChild = 10
9 < 5 falso
10 < 5 falso
4 ≠ 4 falso

PASO 4

```
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
```

i = 0
0 ≥ 0 cierto
heapify(A, 5, 0)

PASO 5

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 5 y 20 > 23 falso
2 < 5 y 16 > 23 falso
0 ≠ 0 falso

PASO 6

```
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

i = -1
-1 ≥ 0 falso
k = 4
4 ≥ 0 cierto
tmp = 23
A[0] = 10
A[4] = 23

10	20	16	11	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 4, 0)

PASO 7

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 4 y 20 > 10 cierto
max = 1
2 < 4 y 16 > 20 falso
1 ≠ 0 cierto
swap = 10
A[0] = 20
A[1] = 10

20	10	16	11	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 4, 1)

PASO 8

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 1
leftChild = 3
rightChild = 4
3 < 4 y 11 > 10 cierto
max = 1
4 < 4 falso
1 ≠ 1 falso

PASO 9

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 3
3 ≥ 0 cierto
tmp = 20
A[0] = 11
A[3] = 20

11	10	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 3, 0)

PASO 10

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 3 y 10 > 11 falso
2 < 3 y 16 > 11 cierto
max = 2
2 ≠ 0 cierto
swap = 11
A[0] = 16
A[2] = 11

A[0] A[1] A[2] A[3] A[4]

heapify(A, 3, 2)

PASO 11

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 2
leftChild = 5
rightChild = 6
5 < 3 falso
6 < 3 falso
2 ≠ 2 falso

PASO 12

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 2
2 ≥ 0 cierto
tmp = 16
A[0] = 11
A[2] = 16

A[0] A[1] A[2] A[3] A[4]

heapify(A, 2, 0)

PASO 14

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 1
1 ≥ 0 cierto
tmp = 11
A[0] = 10
A[1] = 11

A[0] A[1] A[2] A[3] A[4]

heapify(A, 1, 0)

PASO 13

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 2 y 10 > 11 falso
2 < 2 falso
0 ≠ 0 falso

PASO 15

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 1 falso
2 < 1 falso
0 ≠ 0 falso

PASO 16

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 0
0 ≥ 0 cierto
tmp = 10
A[0] = 10
A[0] = 10

10	11	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 0, 0)

PASO 17

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 0 falso
2 < 0 falso
0 ≠ 0 falso

PASO 18

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = -1
-1 ≥ 0 falso

2.1.4 Otras consideraciones de eficiencia

La eficiencia de un algoritmo se mide en función de ejecución del mismo, lo cual depende, del tiempo que le tome a la computadora ejecutar las líneas de código del algoritmo. Esto depende de la velocidad de la computadora, el lenguaje de programación y el compilador entre otros factores.

Medimos el tiempo de ejecución de un algoritmo como una función del tamaño de su entrada, usualmente la cantidad de datos de entrada se especifica con la letra n. A la velocidad del crecimiento de la función con el tamaño de la entrada se le llama tasa de crecimiento del tiempo de ejecución.

El estudio del cambio en el rendimiento del algoritmo con el cambio en el orden del tamaño de entrada se define como análisis asintótico. Existen tres notaciones asintóticas

para medir la eficiencia de un algoritmo la notación O que se explicó en la sección 1.2, la notación Theta grande (θ) y notación Omega grande (Ω) que veremos en esta sección.

❖ Notación Theta (θ)

La notación theta encierra la función desde arriba y desde abajo. Dado que representa el límite superior e inferior del tiempo de ejecución de un algoritmo, se utiliza para analizar la complejidad de caso promedio de un algoritmo.

Para una función g (n), Θ (g (n)) viene dada por la relación:

$$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

La expresión anterior puede describirse como una función f (n) pertenece al conjunto Θ (g (n)) si existen constantes positivas c1 y c2 de modo que pueda intercalarse entre c1g (n) y c2g (n), por suficientemente grande n.

Si una función f(n) se encuentra entre $c_1 g(n)$ y $c_2 g(n)$ para todo $n \geq n_0$, entonces se dice que f(n) tiene un límite asintóticamente estrecho.

❖ Notación Omega (Ω)

La notación Omega representa el límite inferior del tiempo de ejecución de un algoritmo. Por lo tanto, proporciona la mejor complejidad de caso de un algoritmo.

$$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ tal que } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

La expresión anterior se puede describir como una función f(n) pertenece al conjunto Ω (g (n)) si existe una constante positiva c tal que se encuentra por encima de cg (n), para n suficientemente grande.

Para cualquier valor de n, el tiempo mínimo requerido por el algoritmo viene dado por Omega Ω (g (n)).

2.1.5 Algoritmos de búsqueda y su eficiencia

La búsqueda es una operación que tiene por objeto la localización de un elemento dentro de la estructura de datos. A menudo un programador estará trabajando con grandes cantidades de datos almacenados en arreglos y pudiera resultar necesario determinar si un arreglo contiene un valor que coincide con algún valor clave o buscado. En las siguientes secciones se presentan diferentes métodos de búsqueda en los arreglos.

2.1.5.1 Secuencial

La búsqueda secuencial es la técnica más simple para buscar un elemento en un arreglo. Consiste en recorrer el arreglo elemento a elemento e ir comparando con el valor buscado (clave). Se empieza con la primera casilla del arreglo y se observa una casilla tras otra hasta que se encuentra el elemento buscado o se han visto todas las casillas. El resultado de la búsqueda es un solo valor, y será la posición del elemento buscado o cero. Dado que el arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentre ya sea en el primer o en el último elemento. La eficiencia de la búsqueda secuencial es de O(n).

Algoritmo de Búsqueda Secuencial

```
leer(dato)
enc= falso
pos = 0
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras)
Si (enc = cierto) entonces
    Imprimir ("El valor se ha encontrado")
Sino
    Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

Ejemplo:

Utilizando el algoritmo de búsqueda secuencial, busque el valor de 6 en el siguiente arreglo.

7	13	6	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

Solución

PASO 1

```
leer(dato)
enc= falso
pos = 0
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras
```

n = 6
dato =6
enc = falso
pos = 0
0 < 6 y enc = falso cierto
7 = 6 falso
pos = 1

PASO 2

```
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras
```

1 < 6 y enc = falso cierto
13 = 6 falso
pos = 2

PASO 3

```
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras
```

2 < 6 y enc = falso cierto
6 = 6 cierto
enc = cierto
2 < 6 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
   Imprimir ("El valor se ha
encontrado")
Sino
   Imprimir ("El valor no se ha
encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

2.1.5.2 Binaria

La búsqueda binaria es el método más eficiente para encontrar elementos en un arreglo ordenado. El proceso comienza comparando el elemento central del arreglo con el valor buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor que el central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en el subarreglo superior, si el elemento buscado es menor que el contenido de la casilla central, se debe buscar en el subarreglo inferior. El orden de complejidad es $O(\log_2 n)$.

```
Algoritmo de Búsqueda Binaria
leer(dato)
inf = 0
sup = n-1
enc = falso
Mientras (inf <= sup y enc = falso) hacer
  centro = (sup + inf) / 2
  // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
        enc = cierto
  sino
        si (dato < a[centro]) entonces
              sup = centro – 1
        sino
              inf = centro + 1
        fin si
  fin si
Fin Mientras
Si (enc = cierto) entonces
        Imprimir ("El valor se ha encontrado")
Sino
        Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

Ejemplo

Utilizando el algoritmo de búsqueda binaria, busque el valor de 23 en el siguiente arreglo.

10	11	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

Solución

PASO 1

```
leer(dato)
inf = 0
sup = n-1
enc = falso
Mientras (inf <= sup y enc = falso) hacer
  centro = (sup + inf) / 2
 // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
     enc = cierto
  sino
     si (dato < a[centro]) entonces
        sup = centro – 1
     sino
        inf = centro + 1
     fin si
  fin si
Fin Mientras
```

n = 5
dato = 23
inf = 0
sup = 4
enc = falso
0 ≤ 4 y enc = falso cierto
centro = (4 + 0)/2
centro = 2
16 = 23 falso
23 < 16 falso
inf = 3

PASO 2

```
Mientras (inf <= sup y enc = falso) hacer
  centro = (sup + inf) / 2
 // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
     enc = cierto
  sino
     si (dato < a[centro]) entonces
        sup = centro – 1
     sino
        inf = centro + 1
     fin si
  fin si
Fin Mientras
```

3 ≤ 4 y enc = falso cierto
centro = (4 + 3)/2
centro = 3
20 = 23 falso
23 < 20 falso
inf = 4

PASO 3

```
Mientras (inf <= sup y enc = falso) hacer
  centro = ((sup - inf) / 2) + inf
 // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
     enc = cierto
  sino
     si (dato < a[centro]) entonces
        sup = centro – 1
     sino
        inf = centro + 1
     fin si
  fin si
Fin Mientras
```

4 ≤ 4 y enc = falso cierto
centro = (4 + 4)/2
centro = 4
23 = 23 cierto
enc = cierto
4 ≤ 4 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
  Imprimir ("El valor se ha encontrado")
Sino
  Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

2.1.5.3 En tabla

También se le conoce como Búsqueda Lineal. El algoritmo busca un dato (clave) K que está almacenado en una tabla T de índice único. La condición de búsqueda pueda ser determinada mediante una función booleana, enc que devuelve un valor cierto si y sólo si el dato satisface la condición. El algoritmo tiene una complejidad lineal de O(n).

Algoritmo Búsqueda Lineal

PASO 3

```
Mientras (inf <= sup y enc = falso) hacer
  centro = ((sup - inf) / 2) + inf
 // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
    enc = cierto
  sino
    si (dato < a[centro]) entonces
      sup = centro – 1
    sino
      inf = centro + 1
    fin si
  fin si
Fin Mientras
```

$4 \leq 4$ y enc = falso cierto
centro = (4 + 4)/2
centro = 4
23 = 23 cierto
enc = cierto
$4 \leq 4$ y enc = falso falso

```
enc = falso
i = 1
mientras (i <= n y enc = falso) hacer
 si (K = A[i]) entonces
   enc = cierto
 sino
   i = i + 1
 fin si
```

PASO 4

```
Si (enc = cierto) entonces
  Imprimir ("El valor se ha encontrado")
Sino
  Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

```
fin mientras
Si (enc = cierto) entonces
        Imprimir ("El valor se ha encontrado")
Sino
        Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

Ejemplo

Utilizando el algoritmo de búsqueda lineal o en tabla, busque el valor de 4 en el siguiente arreglo.

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

Solución

PASO 1

```
enc = falso
i = 1
mientras (i <= n y enc = falso) hacer
 si (K = A[i]) entonces
   enc = cierto
 sino
   i = i + 1
 fin si
fin mientras
```

n = 5
K = 4
enc = falso
i = 1
$1 \leq 5$ y enc = falso cierto
4 = 9 falso
i = 2

PASO 2

```
mientras (i <= n y enc = falso) hacer
 si (K = A[i]) entonces
   enc = cierto
 sino
   i = i + 1
 fin si
fin mientras
```

$2 \leq 5$ y enc = falso cierto
4 = 5 falso
i = 3

PASO 3

```
mientras (i <= n y enc = falso) hacer
 si (K = A[i]) entonces
  enc = cierto
 sino
   i = i + 1
  fin si
fin mientras
```

3 ≤ 5 y enc = falso cierto
4 = 4 cierto
enc = cierto
3 ≤ 5 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
  Imprimir ("El valor se ha encontrado")
Sino
 Imprimir ("El valor no se ha
encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

2.1.5.4 Directa de cadena

El algoritmo localiza dentro de una cadena de longitud n (denominada texto, arreglo t en el algoritmo) una subcadena más pequeña de longitud m (denominada patrón, arreglo p en el algoritmo) o un carácter. El algoritmo compara carácter a carácter los arreglos texto y patrón comenzando por el extremo izquierdo de ambos, si coinciden se compara el siguiente carácter, si no coinciden el proceso se reinicia comenzado en la posición siguiente a la que se inició la búsqueda.

```
Búsqueda cadena
// t[n] y p[m] son arreglos de caracteres
i = 0
j = 0
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j  = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
si (j = m) entonces
  Imprimir ("Se encontró la subcadena")
sino
  Imprimir ("No se encontró la subcadena")
fin si
fin
```

Ejemplo

Utilizando el algoritmo de búsqueda de cadena, busque la subcadena del arreglo P en el arreglo T.

A	D	E
P[0]	P[1]	P[2]

C	A	D	E	N	A
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]

Solución

PASO 1

```
i = 0
j = 0
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

n = 6
m = 3
i = 0
j = 0
0 < 6 y 0 < 3 cierto
C = A falso
i = 1
j = 0

PASO 2

```
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

1 < 6 y 0 < 3 cierto
A = A cierto
i = 2
j = 1

PASO 3

```
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

2 < 6 y 1 < 3 cierto
D = D cierto
i = 3
j = 2

PASO 4

```
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

3 < 6 y 2 < 3 cierto
E = E cierto
i = 4
j = 3
4 < 6 y 3 < 3 falso

PASO 5

```
si (j = m) entonces
  Imprimir ("Se encontró la
    subcadena")
sino
  Imprimir ("No se encontró la
    subcadena")
fin si
fin
```

3 = 3 cierto
Se encontró la
subcadena

2.1.5.5 De cadena Knuth-Morris-Pratt

También se le conoce como el Algoritmo KMP, busca las ocurrencias de un "patrón" dentro de un "texto" principal. El algoritmo usa la observación de cuando se produce una falta de coincidencia, la palabra en sí incluye información suficiente para determinar dónde podría comenzar la próxima coincidencia: con la cadena a localizar se precalcula una tabla de saltos (conocida como tabla de fallos) que después al examinar entre sí las cadenas, se utiliza para hacer saltos cuando se localiza un fallo. Con esto se evita el análisis más de una vez de los caracteres de la cadena donde se busca. El tiempo total de ejecución del algoritmo es O(n).

```
Algoritmo KMP(Text, Patron)
// m = largo del texto, arreglo T del texto
// n = largo del patrón, arreglo P del patrón
GenerateSuffixArray(Patron)
i = 0
j = 0
mientras (i < m) hacer
   si (Patron[j] = Text[i]) entonces
      j = j + 1
      i = i + 1
   fin si
   si (j = n) entonces
      Imprimir ("Incidencia encontrada en: ",
      i - j)
   sino
   si (i < m) y (Patron[j] ≠ Text[i]) entonces
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         i = i + 1
      fin si
   fin si
 fin si
fin mientras
```

```
GenerateSuffixArray(Patron)
i = 1
j = 0
S[0] = 0
// n = largo del patrón, arreglo P del patrón
mientras (i < n) hacer
   si (Patron[i] = Patron[j]) entonces
      S[i] = j + 1
      j = j + 1
      i = i + 1
   sino
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         S[i] = 0
         i = i + 1
      fin si
   fin si
fin mientras
```

Ejemplo

Utilizando el algoritmo KMP, busque la subcadena del arreglo P en el arreglo T.

A	A	B	A
P[0]	P[1]	P[2]	P[3]

F	B	A	A	B	A	D	C
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]

Solución

PASO 1

```
GenerateSuffixArray(Patron)
i = 1
j = 0
S[0] = 0
// n = largo del patrón, arreglo P del patrón
mientras (i < n) hacer
  si (Patron[i] = Patron[j]) entonces
    S[i] = j + 1
    j = j + 1
    i = i + 1
```

n = 4
i = 1
j = 0
S[0] = 0

0			
S[0]	S[1]	S[2]	S[3]

1 < 4 cierto
A = A cierto
S[1] = 1

0	1		
S[0]	S[1]	S[2]	S[3]

j = 1
i = 2

PASO 2

```
mientras (i < n) hacer
  si (Patron[i] = Patron[j]) entonces
    S[i] = j + 1
    j = j + 1
    i = i + 1
  sino
    si (j ≠ 0) entonces
      j = S[j-1]
    sino
      S[i] = 0
      i = i + 1
    fin si
  fin si
fin mientras
```

2 < 4 cierto
B = A falso
1 ≠ 0 cierto
j = 0
0 < 4 cierto
B = A falso
0 ≠ 0 falso
S[2] = 0

0	1	2	
S[0]	S[1]	S[2]	S[3]

i = 3

PASO 3

```
mientras (i < n) hacer
  si (Patron[i] = Patron[j]) entonces
    S[i] = j + 1
    j = j + 1
    i = i + 1
  sino
    si (j ≠ 0) entonces
      j = S[j-1]
    sino
      S[i] = 0
      i = i + 1
    fin si
  fin si
fin mientras
```

3 < 4 cierto
A = A cierto
S[3] = 1

0	1	2	1
S[0]	S[1]	S[2]	S[3]

j = 1
i = 4
3 < 4 falso

PASO 4 – Continua en el algoritmo KMP

```
i = 0
j = 0
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
```

m = 8
n = 4
i = 0
j = 0
0 < 8 cierto
A = F falso
0 = 4 falso

PASO 5

```
si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
 sino
   si (i < m)  y (Patron[j] ≠ Text[i])
   entonces
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         i = i + 1
      fin si
   fin si
 fin si
fin mientras
```

0 = 4 falso
0 < 8 y A ≠ F cierto
0 ≠ 0 falso
i = 1

PASO 6

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
     Imprimir ("Incidencia encontrada
     en: ", i - j)
     j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

1 < 8 cierto
A = B falso
0 = 4 falso
1 < 8 y A ≠ B cierto
0 ≠ 0 falso
i = 2

PASO 7

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
     Imprimir ("Incidencia encontrada
     en: ", i - j)
     j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

2 < 8 cierto
A = A cierto
j = 1
i = 3
1 = 4 falso
3 < 8 y A ≠ A falso

PASO 8

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
     Imprimir ("Incidencia encontrada
     en: ", i - j)
     j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

3 < 8 cierto
A = A cierto
j = 2
i = 4
2 = 4 falso
4 < 8 y B ≠ B falso

PASO 9

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

4 < 8 cierto
B = B cierto
j = 3
i = 5
3 = 4 falso
5 < 8 y A ≠ A falso

PASO 10

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

5 < 8 cierto
A = A cierto
j = 4
i = 6
4 = 4 cierto
Incidencia
encontrada en: 2
j = 1

PASO 11

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

6 < 8 cierto
A = D falso
1 = 4 falso
6 < 8 y A ≠ D cierto
1 ≠ 0 cierto
j = 0

PASO 12

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

6 < 8 cierto
A = D falso
0 = 4 falso
6 < 8 y A ≠ D cierto
0 ≠ 0 falso
i = 7

PASO 13

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

7 < 8 cierto
A = C falso
0 = 4 falso
7 < 8 y A ≠ C cierto
0 ≠ 0 falso
i = 8
7 < 8 falso

2.1.5.6 De cadena Boyer-Moore

El algoritmo preprocesa la cadena objetivo (patrón) que está siendo buscada. En su verificación, el algoritmo intenta comprobar si hay una coincidencia en una posición particular marchando hacia atrás. El algoritmo precalcula dos tablas para procesar la información que obtiene en cada verificación fallada: una tabla calcula cuantas posiciones hay por delante en la siguiente búsqueda basada en el valor del carácter que no coincide; la otra hace un cálculo similar basado en cuantos caracteres coincidieron satisfactoriamente antes del intento de coincidencia fallado. Estas dos tablas devuelven resultados que indican cuán lejos "saltar" hacia delante, por este motivo son llamada en algunas ocasiones "tablas de salto". El algoritmo se desplazará con el valor más grande de los dos valores de salto cuando no ocurra una coincidencia. El algoritmo Boyer-Moore presenta en el peor de los casos una complejidad de O(n).

Una simplificación del algoritmo que omite la "tabla primera" es el algoritmo Boyer-Moore-Horspool (BMH) que requiere, en el peor caso, mn comparaciones. El algoritmo BMH compara el patrón con el texto de derecha a izquierda, y se detiene cuando se encuentra una discrepancia con el texto. Cuando esto sucede, se desliza el patrón de manera que la letra del texto que estaba alineada con bm se quede alineada con algún bj, con j<m. Esta función sólo depende del patrón.

```
Algoritmo BMH
// m es el largo del patrón
// n es el largo del texto
// los índices comienzan desde 1
k = m
j = m
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
    k = k + (m - siguiente(T[k]))
    j = m
 fin si
fin mientras
Si (j = 0) entonces
   imprimir ("Se encontró el patrón")
sino
   imprimir ("No se encontró el patrón")
fin si
fin
```

```
siguiente (T[k])
   val = 0
   desde (i = 1, i < m, i = i + 1)
      si (T[k] = P[i]) entonces
         val = i
         i = m
      fin si
   fin desde
   retornar (val)
fin siguiente
```

Ejemplo

Utilizando el algoritmo BMH, busque la subcadena del arreglo P en el arreglo T.

P	R	U	E
P[1]	P[2]	P[3]	P[4]

T	R	S	U	P	R	U	E	A	C
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Solución

PASO 1

```
// m es el largo del patrón
// n es el largo del texto
// los índices comienzan desde 1
k = m
j = m
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] = P[j]) entonces
     j = j - 1
  sino
   k = k + (m - siguiente(T[k]))
   j = m
  fin si
fin mientras
```

m = 4
n = 10
k = 4
j = 4
4 ≤ 10 y 4 ≥ 1 cierto
T[4 – (4 – 4)] = T[4]
U = E falso
k = 4 + (4 – siguiente(T[4])

PASO 2

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
   si (T[k] = P[i]) entonces
      val = i
       i = m
   fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
U = P falso
i = 2
2 < 4 cierto
U = R falso
i = 3
3 < 4 cierto
U = U cierto
val = 3
i = 4
4 < 4 falso
retornar (3)

PASO 3

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] = P[j]) entonces
     j = j - 1
  sino
   k = k + (m - siguiente(T[k]))
   j = m
  fin si
fin mientras
```

k = 4 + (4 – 3)
k = 3
j = 4
3 ≤ 10 y 4 ≥ 1 cierto
T[3 – (4 – 4)] = T[3]
S = E falso
k = 3 + (4 – siguiente(T[3])

PASO 4

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
   si (T[k] = P[i]) entonces
      val = i
       i = m
   fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
S = P falso
i = 2
2 < 4 cierto
S = R falso
i = 3
3 < 4 cierto
S = U cierto
val = 3
i = 4
4 < 4 falso
retornar (0)

PASO 5

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

k = 4 + (4 – 0)
k = 7
j = 4
7 ≤ 10 y 4 ≥ 1 cierto
T[7 – (4 – 4)] = T[7]
U = E falso
k = 7 + (4 – siguiente(T[7])

PASO 6

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
   si (T[k] = P[i]) entonces
      val = i
      i = m
   fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
U = P falso
i = 2
2 < 4 cierto
U = R falso
i = 3
3 < 4 cierto
U = U cierto
val = 3
i = 4
4 < 4 falso
retornar (3)

PASO 7

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

k = 7 + (4 – 3)
k = 6
j = 4
6 ≤ 10 y 4 ≥ 1 cierto
T[6 – (4 – 4)] = T[6]
R = E falso
k = 6 + (4 – siguiente(T[6])

PASO 8

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
   si (T[k] = P[i]) entonces
      val = i
      i = m
   fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
R = P falso
i = 2
2 < 4 cierto
R = R cierto
val = 2
i = 4
4 < 4 falso
retornar (2)

PASO 9

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

k = 6 + (4 – 2)
k = 8
j = 4

PASO 10

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

8 ≤ 10 y 4 ≥ 1 cierto
T[8 – (4 – 4)] = T[8]
E = E cierto
j = 3

PASO 11

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

$8 \leq 10$ y $3 \geq 1$ cierto
$T[8 - (4 - 3)] = T[7]$
U = U cierto
j = 2

PASO 12

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

$8 \leq 10$ y $2 \geq 1$ cierto
$T[8 - (4 - 2)] = T[6]$
R = R cierto
j = 1

PASO 13

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

$8 \leq 10$ y $1 \geq 1$ cierto
$T[8 - (4 - 1)] = T[5]$
P = P cierto
j = 0

PASO 14

```
mientras (k ≤ n y j ≥ 1) hacer
 si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
 sino
  k = k + (m - siguiente(T[k]))
  j = m
 fin si
fin mientras
```

$8 \leq 10$ y $0 \geq 1$ falso
j = 0 cierto
Se encontró el patrón

2.2 Transformación de llaves

2.2.1 Definición y conceptos

El método llamado por transformación de claves o llaves (hash), permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta también con la ventaja de que el tiempo de búsqueda es prácticamente independiente del número de componentes del arreglo. Trabaja basándose en una función de transformación o función hash (H) que convierte una clave en una dirección (índice) dentro del arreglo.

dirección ← H(clave)

Cuando se tienen claves que no se corresponden con índices (por ejemplo, por ser alfanuméricas), o bien cuando las claves son valores numéricos muy grandes, debe

utilizarse una función hash que permita transformar la clave para obtener una dirección apropiada. Esta función hash debe de ser simple de calcular y debe de asignar direcciones de la manera más uniforme posible. Es decir, dadas dos claves diferentes, debe generar posiciones diferentes. Si esto no ocurre, por ejemplo:

$$H(K1) = d$$
$$H(K2) = d$$
$$y\ K1 \neq K2$$

hay una colisión. Se define, entonces, una colisión como la asignación de una misma dirección a dos o más claves distintas.

Por todo lo mencionado, para trabajar con este método de búsqueda debe elegirse previamente:

- Una función hash que sea fácil de calcular y que distribuya uniformemente las claves.
- Un método para resolver colisiones. Si estas se presentan se debe contar con algún método que genere posiciones alternativas.

Costos:

- Tiempo de procesamiento requerido para la aplicación de la función hash
- Tiempo de procesamiento y los accesos E/S requeridos para solucionar las colisiones.

A continuación, se enumeran las ventajas y desventajas de la función hash, así como también se brinda información acerca de los costos y la eficiencia que la misma posee.

Ventajas:

- Se pueden usar los valores naturales de la llave, puesto que se traducen internamente a direcciones fáciles de localizar

- Se logra independencia lógica y física, debido a que los valores de las llaves son independientes del espacio de direcciones
- No se requiere almacenamiento adicional para los índices.

Desventajas:

- No pueden usarse registros de longitud variable
- El archivo no está clasificado
- No permite llaves repetidas
- Solo permite acceso por una sola llave

Usos de la función hash

- Como un tipo de clave mediante la cual se ordena la lista
- Como un método de acceso al registro

La eficiencia de una función hash depende de:

1. La distribución de los valores de llave que realmente se usan
2. El número de valores de llave que realmente están en uso con respecto al tamaño del espacio de direcciones
3. El número de registros que pueden almacenarse en una dirección dada sin causar una colisión
4. La técnica usada para resolver el problema de las colisiones

2.2.2 Técnicas de cálculo de direcciones

No hay reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves que asegure la máxima uniformidad en la distribución de las mismas. Realizar un análisis de las principales características de las claves puede ayudar en la

elección de la función hash. A continuación, se detallan algunas de las funciones hash más utilizadas.

2.2.2.1 Hashing por residuo

Consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Suponga que se tiene un arreglo de N elementos, y K es la clave del dato a buscar. La función hash queda definida por la siguiente formula:

$$H(K) = (K \bmod N) + 1 \quad \text{(fórmula 2.1)}$$

En la fórmula anterior se puede observar que al residuo de la división se le suma 1, esto es para obtener un valor entre 1 y N.

Para lograr una mayor uniformidad en la distribución, N debe ser un número primo o divisible entre muy pocos números. Por lo tanto, si dado N éste no es un número primo, se tomará el valor primo más cercano.

Ejemplo

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 y K_2 dos claves a las que deben asignarse posiciones en el arreglo. Utilizando la función hashing por residuo, calcule las direcciones para K_1 = 7259 y K_2 = 9359.

Solución

Aplicando la fórmula 2.1 con N = 100, para calcular las direcciones correspondientes a K_1 y K_2.

$$H(K_1) = (7259 \bmod 100) + 1 = 60$$

$$H(K_2) = (9359 \bmod 100) + 1 = 60$$

Como H(K_1) es igual a H(K_2) y K_1 es distinto de K_2, se está ante una colisión. Se aplica ahora la fórmula 2.1 con N igual a un valor primo en vez de utilizar N igual a 100.

$$H(K_1) = (7259 \bmod 97) + 1 = 82$$

$$H(K_2) = (9359 \bmod 97) + 1 = 48$$

Con N = 97 se ha eliminado la colisión.

2.2.2.2 Hashing por cuadrado medio

Consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos a tomar queda determinado por el rango del índice. Sea K la clave del dato a buscar. La función hash queda definida por la siguiente fórmula:

H(K) = dígitos_centrales (K^2) + 1 (Fórmula 2.2)

La suma de una unidad a los dígitos centrales es para obtener un valor entre el 1 y N.

Ejemplo:

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 y K_2 dos claves a las que deben asignarse posiciones en el arreglo. Utilizando la función hashing por cuadrado medio, calcule las direcciones para K_1 = 7259 y K_2 = 9359.

Solución

Aplicando la fórmula 2.2 para calcular las direcciones a K_1 y K_2.

$$K_1^2 = (7259)^2 = 52693081$$

$$K_2^2 = (9359)^2 = 87590881$$

$H(K_1)$ = dígitos_centrales (52693081) + 1 = 94

$H(K_2)$ = dígitos_centrales (87590881) + 1 = 91

Como el rango de índices en el ejemplo varía de 1 a 100, se toman solamente los dos dígitos centrales del cuadrado de las claves.

2.2.2.3 Hashing por pliegue

Consiste en dividir la clave en partes de igual número de dígitos (la última puede tener menos dígitos) y operar con ellas, tomando como dirección los dígitos menos significativos. La operación entre las partes puede hacerse por medio de sumas o multiplicaciones. Sea K la clave del dato a buscar. K está formada por los dígitos d_1, d_2, …, d_n. La función hash queda definida por la siguiente formula:

$$H(K) = \text{dígmensig}((d_1 \ldots d_i) + (d_{i+1} \ldots d_j) + \ldots + (d_1 \ldots d_n)) + 1 \quad \text{(Fórmula 2.3)}$$

El operador que aparece en la fórmula operando las partes de la clave es el de suma, pero como se aclaró antes, puede ser el de la multiplicación. La suma de una unidad a los dígitos menos significativos (dígmensig) es para obtener un valor entre 1 y N.

Ejemplo:

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 y K_2 dos claves a las que deben asignarse posiciones en el arreglo. Utilizando la función hashing por pliegue, calcule las direcciones para K_1 = 7259 y K_2 = 9359.

Solución

Aplicando la fórmula 2.3 para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = dígmensig (72 + 59) + 1 = dígmensig (131) + 1 = 32

$$H(K_2) = \text{dígmensig } (93 + 59) + 1 = \text{dígmensig } (152) + 1 = 53$$

De la suma de las partes se toman solamente dos dígitos porque los índices del arreglo varían de 1 a 100.

2.2.3 Comparación entre las funciones Hash

Aunque alguna otra técnica pueda desempeñarse mejor en situaciones particulares, la técnica del residuo de la división proporciona el mejor desempeño. El método del medio del cuadrado puede aplicarse en archivos con factores de cargas bastantes bajas para dar generalmente un buen desempeño. El método de pliegues puede ser la técnica más fácil de calcular, pero produce resultados bastante erráticos, a menos que la longitud de la llave sea aproximadamente igual a la longitud de la dirección.

Si la distribución de los valores de llaves no es conocida, entonces el método del residuo de la división es preferible. Note que el hashing puede ser aplicado a llaves no numéricas. Las posiciones de ordenamiento de secuencia de los caracteres en un valor de llave pueden ser utilizadas como sus equivalentes “numéricos”. Alternativamente, el algoritmo hash actúa sobre las representaciones binarias de los caracteres.

Todas las funciones hash presentadas tienen destinado un espacio de tamaño fijo. Aumentar el tamaño del archivo relativo creado al usar una de estas funciones, implica cambiar la función hash, para que se refiera a un espacio mayor y volver a cargar el nuevo archivo.

2.2.4 Métodos para el manejo del problema de las colisiones

Como se definió anteriormente, tenemos una colisión cuando se asigna una misma dirección a dos o más claves distintas. La elección de un método adecuado para resolver

colisiones es tan importante como la elección de una buena función hash. Se está ante una colisión cuando la función obtiene una misma dirección para dos claves diferentes. Normalmente, cualquiera que sea el método elegido resulta costoso tratar las colisiones. Es por ello, que se debe hacer un esfuerzo por encontrar la función que ofrezca mayor uniformidad en la distribución de las claves.

La manera más natural de resolver el problema de las colisiones es reservar una casilla por clave, es decir, que aquellas se correspondan una a una con las posiciones del arreglo. Pero como ya se mencionó, esta solución puede tener un alto costo en memoria, por lo tanto, deben analizarse otras alternativas que permitan equilibrar el uso de memoria con el tiempo de búsqueda.

En esta sección se estudiarán los siguientes métodos:

- Prueba lineal
- Prueba cuadrática
- Doble dirección hash

❖ Prueba lineal

Consiste en, una vez detectada la colisión, recorrer el arreglo secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía. Se trata el arreglo como una estructura circular: el siguiente elemento después del último es el primero.

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del arreglo permanecerían vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial, perdiendo así las ventajas del método hash.

Ejemplo

Sea V un arreglo de 10 elementos. Calcule su dirección según la función Hash: **H(K) = (K mod 10) + 1** para las claves K que se muestran en la tabla. En caso de colisiones solucione el mismo por medio de la prueba lineal.

k	25	43	56	35	54	13	80	104

Solución

Paso 1

Calculamos la dirección para cada una de las claves K utilizando la función hash

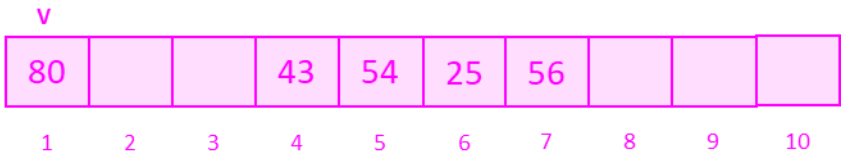
H(K) = (K mod 10) + 1

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Paso 2

Ubicamos todas las claves que no tengan colisiones en el arreglo (color verde), las que presenten colisiones se resolverán luego de terminar de ubicar las claves sin colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5



Paso 3

Ahora se procede a ubicar las claves con colisiones, estas se muestran en color rojo en la tabla.

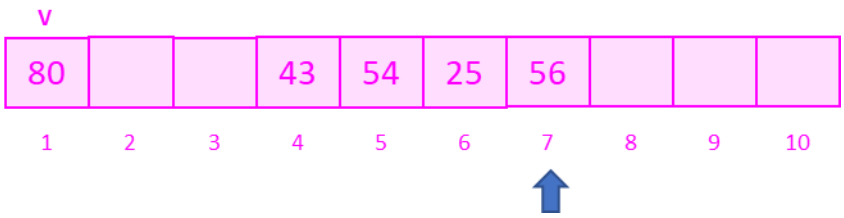
k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se recorre el arreglo secuencialmente a partir del punto de colisión hasta encontrar una posición vacía.

K = 35

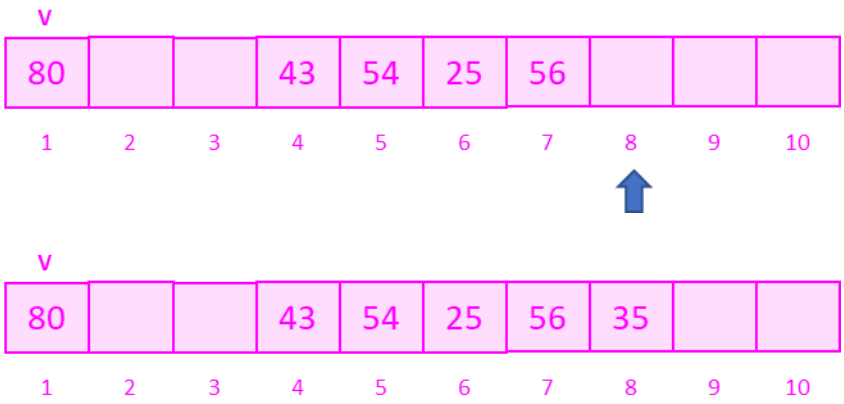
H(k) = 6

H'(k) = 7



la posición se encuentra ocupada, por lo tanto, seguimos buscando

H'(k) = 8



Paso 4

Se ubicar la siguiente clave con colisión.

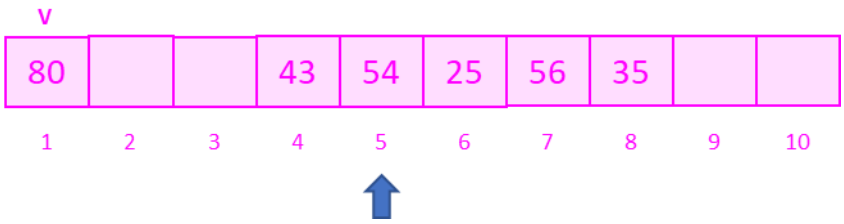
k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se recorre el arreglo secuencialmente a partir del punto de colisión hasta encontrar una posición vacía.

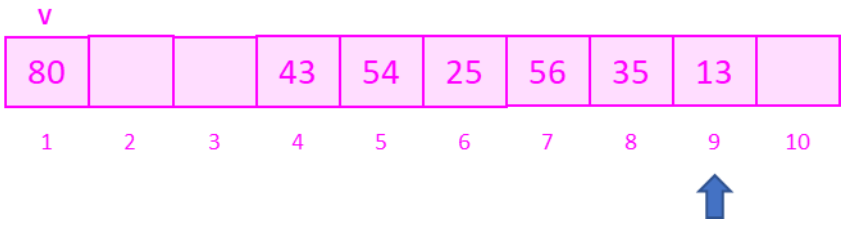
K = 13

H(k) = 4

H’(k) = 5



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo



Paso 5

Se procede de la misma forma con la última clave con colisión.

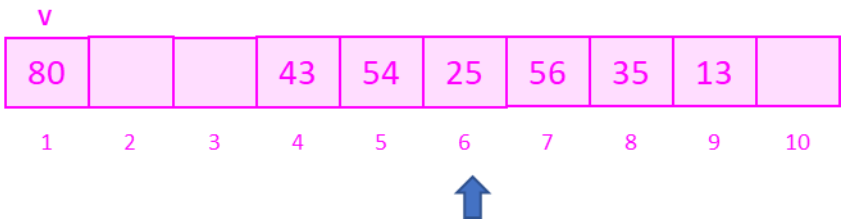
k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se recorre el arreglo secuencialmente a partir del punto de colisión hasta encontrar una posición vacía.

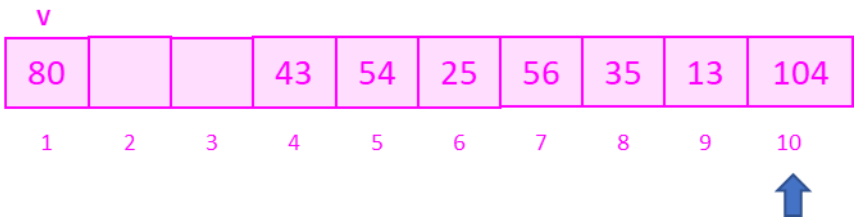
K = 104

H(k) = 5

H’(k) = 6



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo



❖ Prueba cuadrática

Este método es similar al de la prueba lineal. La diferencia consiste en que en el cuadrático las direcciones alternativas se generarán como D + 1, D + 4, D + 9, …, D + i2 en vez de D + 1, D + 2, …, D + i. Esta variación permite una mejor distribución de las claves colisionadas.

La principal desventaja de este método es que pueden quedar casillas del arreglo sin visitar. Además, como los valores de las direcciones varían en i2 unidades, resulta difícil determinar una condición general para detener el ciclo mientras. Este problema podría solucionarse empleando una variable auxiliar, cuyos valores dirijan el recorrido del arreglo, de tal manera, que se garantice que serán visitadas todas las casillas.

Ejemplo

Sea V un arreglo de 10 elementos. Calcule su dirección según la función Hash: **H(K) = (K mod 10) + 1** para las claves K que se muestran en la tabla. En caso de colisiones solucione el mismo por medio de la prueba cuadrática.

k	25	43	56	35	54	13	80	104

Solución

Paso 1

Calculamos la dirección para cada una de las claves K utilizando la función hash

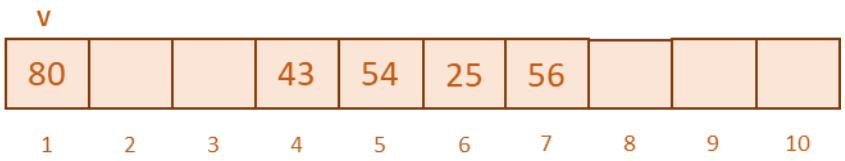
H(K) = (K mod 10) + 1

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Paso 2

Ubicamos todas las claves que no tengan colisiones en el arreglo (color verde), las que presenten colisiones se resolverán luego de terminar de ubicar las claves sin colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5



Paso 3

Ahora se procede a ubicar las claves con colisiones, estas se muestran en color rojo en la tabla.

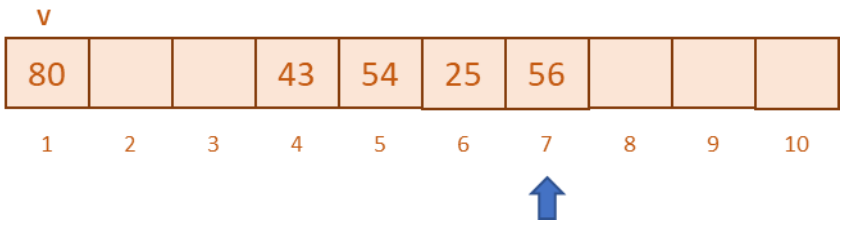
k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se calculan las direcciones alternativas con **D + i^2** a partir del punto de colisión hasta encontrar una posición vacía.

K = 35

H(k) = 6

H'(k) = 6 + 1^2 = 7



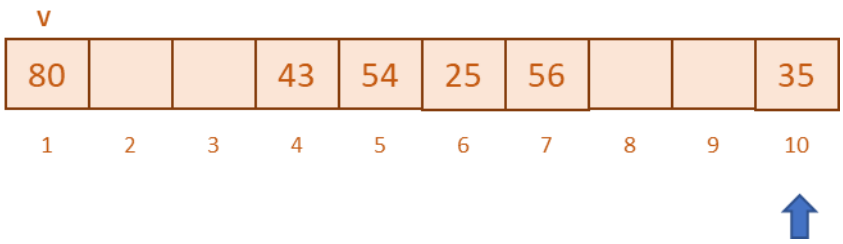
la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

Paso 4

K = 35

H(k) = 6

H'(k) = 6 + 2^2 = 6 + 4 = 10



Paso 5

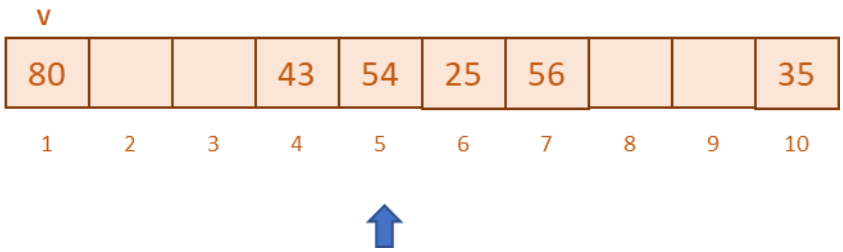
Se ubica la siguiente clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 13

H(k) = 4

H'(k) = 4 + 1^2 = 5



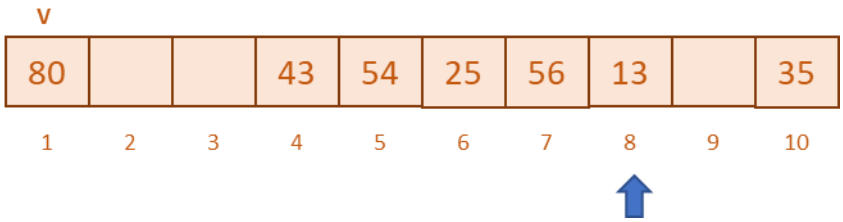
la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

Paso 6

K = 13

H(k) = 4

H'(k) = $4 + 2^2 = 4 + 4 = 8$



Paso 7

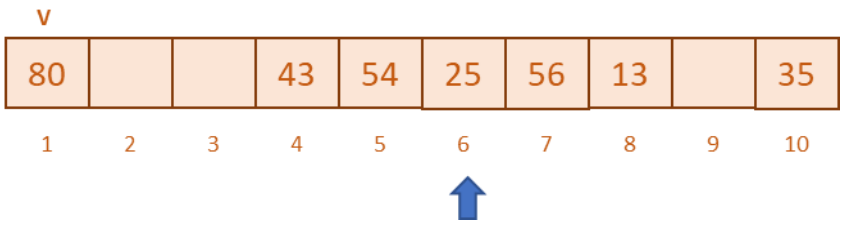
Se procede de la misma forma con la última clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 104

H(k) = 5

H'(k) = $5 + 1^2 = 6$



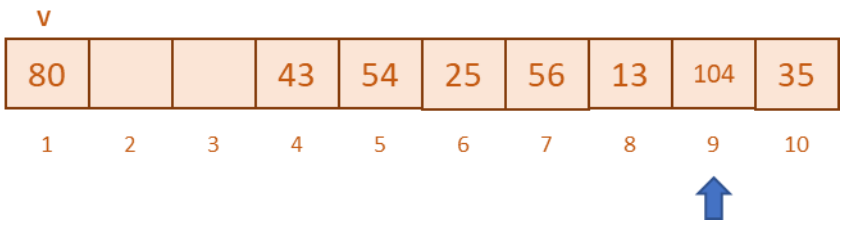
la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

Paso 8

K = 104

H(k) = 5

$H'(k) = 5 + 2^2 = 5 + 9$



❖ Doble dirección hash

Consiste en, una vez detectada la colisión, generar otra dirección aplicando la función hash a la dirección previamente obtenida. El proceso se detiene cuando el elemento es hallado, o bien cuando se encuentra una posición vacía.

La función hash que se aplica a las direcciones puede o no ser la misma que originalmente se aplicó a la clave. No existe una regla que nos permita decidir cuál será la mejor función que se puede emplear en el cálculo de las direcciones sucesivas.

Ejemplo

Sea V un arreglo de 10 elementos. Calcule su dirección según la función Hash: **H(K) = (K mod 10) + 1** para las claves K que se muestran en la tabla. En caso de colisiones solucione el mismo por medio de la doble dirección hash.

k	25	43	56	35	54	13	80	104

Solución

Paso 1

Calculamos la dirección para cada una de las claves K utilizando la función hash

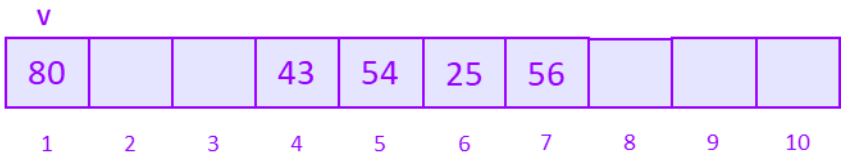
H(K) = (K mod 10) + 1

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Paso 2

Ubicamos todas las claves que no tengan colisiones en el arreglo (color verde), las que presenten colisiones se resolverán luego de terminar de ubicar las claves sin colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5



Paso 3

Ahora se procede a ubicar las claves con colisiones, estas se muestran en color rojo en la tabla.

k	25	43	56	35	54	13	80	104

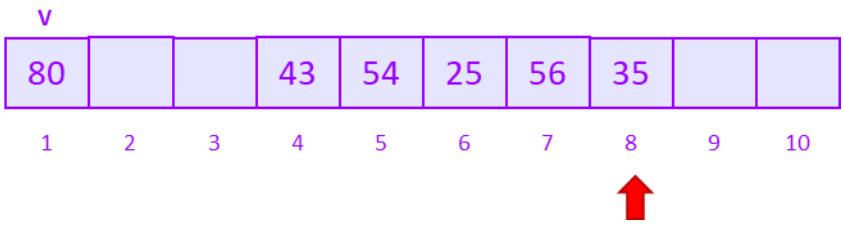
H(k)	6	4	7	6	5	4	1	5

Se calculan las direcciones alternativas con **H(D) + 2** a partir del punto de colisión hasta encontrar una posición vacía.

K = 35

H(k) = 6

H'(k) = 6 + 2 = 8



Paso 4

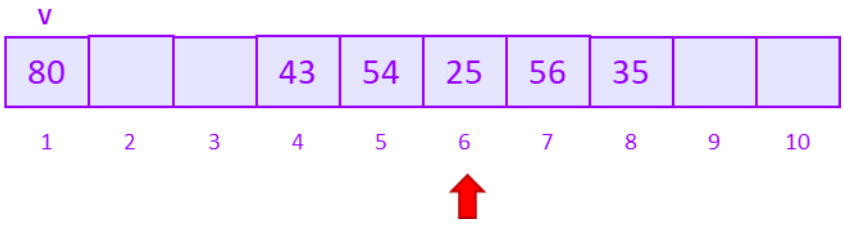
Se ubica la siguiente clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 13

H(k) = 4

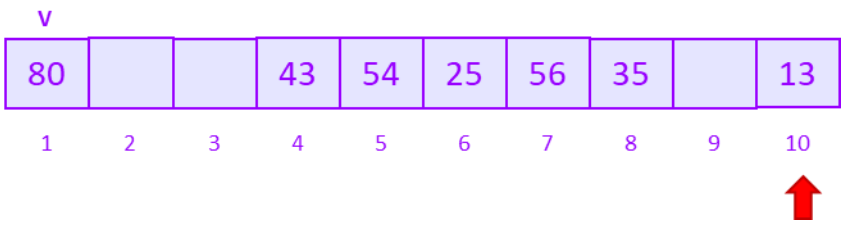
H'(k) = 4 + 2 = 6



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

H'(k) = 6 + 2 = 8 **-- posición ocupada**

H’(k) = 8 + 2 = 10



Paso 5

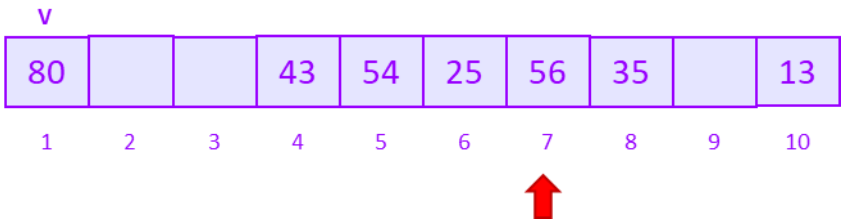
Se procede de la misma forma con la última clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 104

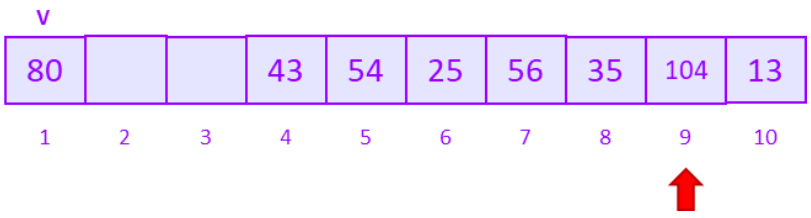
H(k) = 5

H’(k) = 5 + 2 = 7



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

H’(k) = 7 + 2 = 9



Bibliografía

Estructuras de Datos con C y C++. Yedidyah Langsam, Moshe J. Augenstein, Aaron M. tenenbaum. Editorial Prentice Hall. 1997. Segunda Edición.

Estructuras de Datos. Osvaldo Cairó, Silvia Guardati. Editorial Mc Graw Hill. 2006. Tercera Edición.

Estructuras de Datos orientada a objetos Algoritmos con C++. Silvia Guardati. Editorial Pearson. 2007. Primera Edición.

Fundamentos de Programación. Algoritmos y Estructura de Datos. Aguilar, Luis Joyanes Mc Graw-Hill.

Estructura de Datos en Java. Joyanes Aguilar, Luis; Zahonero Martínez, Ignacio. Edit. McGraw.

Estructuras de Datos. Osvaldo Cairó, Silvia Guardati. Editorial Mc Graw Hill. 2006. Tercera Edición.

Algoritmos y Estructuras de Datos. Wirth, Niklaus. Prentice Hall.

Estructura de Datos Teoría y Problemas. Lipschuts, Seymour. Mc Graw Hill.

Análisis de Algoritmos y Tecnología de Grafos. Abellanas, Lodares. Macrobit.

Estructura de Datos y Organización de Archivos. Loomis, Mary E. Prentice Hall.

Anexos 1: Pruebas Rápidas

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #1

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. Es el primer nodo de un árbol: _____.
2. Es la secuencia de aristas a través de las cuales se pasa desde el nodo raíz a un nodo: _____.
3. Es el número de hijos que tiene un nodo: _____.
4. Es la distancia desde la raíz en la que se encuentra ubicado un nodo: _____.
5. Son las líneas que unen a los nodos: _____.
6. Es la longitud del camino más largo que comienza en el nodo y termina en una hoja: _____.

II Parte: Desarrollo

1. Describa por qué los árboles son estructuras de datos no lineales y dinámicas.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #2

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Selección múltiple.

1. Cuando dos árboles binarios tienen estructuras idénticas pero la información que contienen los nodos difiere entre sí, se dice que son:
   a. Árboles binarios equivalentes
   b. Árboles binarios completos
   c. Árboles binarios similares

2. Tipo de árbol en el que el número de hijos de cada nodo es igual al grado del árbol:
   a. Árbol binario de búsqueda
   b. Árbol binario extendido
   c. Árbol general

3. Un árbol es una estructura de datos de tipo:
   a. No lineal y dinámica b. No lineal y dinámica c. Lineal y dinámica

II Parte: Desarrollo.

1. ¿Qué es un árbol binario?
2. Mencione las formas de representar árboles en memoria.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #3

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. El recorrido en _____ es aquel que empieza desde la raíz y visita a todos los nodos de una sola rama del árbol.
2. El recorrido en _____ es aquel que empieza desde la raíz y atraviesa el árbol nivel por nivel.
3. El recorrido en profundidad puede ser de tres tipos: _____, _____ y _____.
4. Nombre de la estructura de datos en la que se implementa el recorrido en profundidad: _____.
5. Nombre de la estructura de datos en la que se implementa el recorrido en amplitud: _____.

II Parte: Desarrollo.

1. Indique el orden del recorrido de cada uno de los tres tipos de recorridos en profundidad.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #4

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Cierto y Falso.

1. _____ El algoritmo de Huffman se usa para crear código de Huffman, que es una técnica usada para descomprimir datos.
2. _____ En un árbol binario, los valores almacenados en los nodos pueden compararse mediante menor (<) y mayor (>).
3. _____ El orden de recorrido del recorrido en preorden es: subárbol izquierdo – raíz – subárbol derecho.
4. _____ Los árboles en montón se utilizan para implementar colas de prioridad.
5. _____ El orden de recorrido del recorrido en postorden es: subárbol izquierdo – subárbol derecho – raíz.
6. _____ El orden de recorrido del recorrido en inorden es: raíz – subárbol izquierdo – subárbol derecho.

II Parte: Desarrollo.

1. ¿Qué son los árboles binarios de expresión?

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #5

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. Un grafo _____ es aquel en el que los vértices tienen el mismo grado.
2. Un grafo _____ es aquel en el que cada vértice tiene un grado igual a n-1, donde n es el número de vértices que componen el grafo.
3. Un grafo _____ es aquel en el que cada arista transporta un valor.
4. Un grafo _____ es aquel que no contiene ningún ciclo simple.
5. Un grafo _____ es aquel en el que los vértices pueden ser divididos en dos conjuntos, de modo que no haya aristas entre los vértices del mismo conjunto.

II Parte: Desarrollo.

1. Describa la definición formal de un grafo.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #6

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. La representación enlazada de grafos se hace mediante: _____.
2. Tres operaciones que pueden realizarse sobre grafos son: _____, _____ y _____.
3. Una lista de nodos está formada por tres partes: _____, _____ y _____.
4. Una lista de aristas está formada por dos partes: _____ y _____.
5. Dos formas de representar secuencialmente un grafo son: _____ y _____.

II Parte: Desarrollo.

1. ¿Cuáles son los enfoques básicos de recorridos en un grafo?

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #7

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. _____ El algoritmo del recorrido en profundidad en un grafo utiliza una cola para ir almacenando los nodos que falta visitar.
2. _____ El algoritmo de Warshall también se denomina algoritmo de caminos mínimos.
3. _____ La representación enlazada de grafos puede ser mediante una lista de aristas o una lista de nodos.
4. _____ El algoritmo del recorrido en anchura en un grafo utiliza una pila de vértices.
5. _____ La única forma de representar un grafo en la representación enlazada.

II Parte: Llene los espacios.

1. Un vértice _____ es aquel con grado de entrada cero.
2. Un vértice _____ es aquel que tiene grado cero.
3. Un vértice _____ es aquel que tiene grado uno.
4. Un vértice _____ es aquel con grado de salida cero.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #8

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. Dos criterios utilizados para evaluar la eficiencia de un algoritmo de ordenamiento son:
   a) _____
   b) _____
2. Mencione cuatro tipos de algoritmos de ordenamiento: _____, _____ y _____.
3. Mencione cuatro tipos de algoritmos de búsqueda: _____, _____, _____ y _____.

II Parte: Desarrollo.

1. Describa los tres tipos de casos que pueden darse al analizar un algoritmo.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #9

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Llene los espacios.

1. Tres notaciones utilizadas para definir la eficiencia de un algoritmo son: _____, _____ y _____.
2. Nombre de la notación que representa el límite inferior del tiempo de ejecución de un algoritmo: _____.
3. Nombre de la notación que representa el límite superior e inferior del tiempo de ejecución de un algoritmo: _____.
4. Método de ordenamiento en el que se aplica la técnica divide y vencerás, al dividir el arreglo a ordenar en dos particiones separadas por un elemento central: _____.
5. Método de ordenamiento en el que todos los elementos se almacenan en un montículo: _____.
6. Método de búsqueda en el que se empieza comparando el elemento central del arreglo con el valor buscado: _____.
7. Método de búsqueda en el que se recorre el arreglo elemento a elemento y se va comparando con el valor buscado: _____.

II Parte: Desarrollo.

1. Describa en qué consisten las operaciones de búsqueda y ordenamiento.

Universidad Tecnológica de Panamá

Facultad de Ingeniería en Sistemas Computacionales

Estructuras de Datos II

Prueba #10

Nombre: _____ Cédula: _____ Grupo: _____

Profesor: Dr. Carlos A. Rovetto Puntos Obtenidos: _____/

I Parte: Selección múltiple.

1. Tipo de hashing que consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección:
   a. Por pliegue b. Por residuo c. Por cuadrado medio

2. Tipo de hashing que consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo:
   a. Por pliegue b. Por residuo c. Por cuadrado medio

3. Tipo de hashing que consiste en dividir la clave en partes de igual número de dígitos y operar con ellas, tomando como dirección los dígitos menos significativos:
   a. Por pliegue b. Por residuo c. Por cuadrado medio

4. Método para el manejo de colisiones que consiste en recorrer el arreglo de forma secuencial a partir del punto de colisión, buscando el elemento:
   a. Doble direcciόh hash b. Prueba lineal c. Prueba cuadrática

5. Método para el manejo de colisiones que consiste en generar otra dirección aplicando la función hash a la dirección previamente obtenida:
   a. Doble direcciόh hash b. Prueba lineal c. Prueba cuadrática

II Parte: Desarrollo.

1. ¿Qué es la transformación de claves?

Anexos 2: Presentaciones

Estructura de Datos II

Autor: Dr. Carlos A. Rovetto

1

Capítulo I

2

Técnicas de estructuración de datos no lineales

Un árbol es una estructura de datos no-lineal y dinámica. Es no-lineal debido a que a cada elemento pueden seguirle varios elementos y es dinámica porque su estructura puede cambiar durante su ejecución.

3

Árboles generales

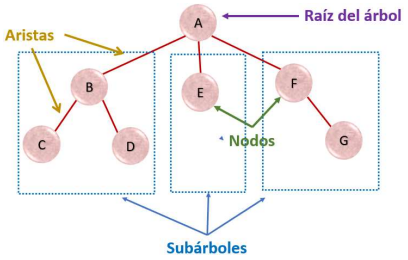
Un árbol general es un árbol donde cada nodo puede tener cero o más hijos (un árbol binario es un caso especializado de un árbol general). Los árboles generales se utilizan para modelar aplicaciones como los sistemas de archivos.

Se puede definir a un árbol como un conjunto finito no vacío T de elementos, llamados nodos, tales que:

- Existe un nodo raíz.
- El resto de los nodos se distribuye en un número n de subconjuntos distintos.
- Cada uno de estos subconjuntos es un subárbol del nodo raíz

4

- **Nodos del árbol:** A, B, C, D, E, F
- **Nodo raíz:** A
- **Raíz:** es el primer nodo del árbol, se encuentra ubicado en la parte superior del árbol. Solo hay una raíz por árbol y una ruta desde el nodo raíz a cualquier nodo.
- **Aristas o ramas:** son las líneas que unen dos nodos.



5

- **Padre:** es el nodo que tiene hijos, es decir, nodos inferiores que se encuentran unidos al nodo superior por una arista. El único nodo que no tiene padre es el nodo raíz del árbol.
- A cada nodo se le asocian uno o varios subárboles llamados descendientes o hijos.

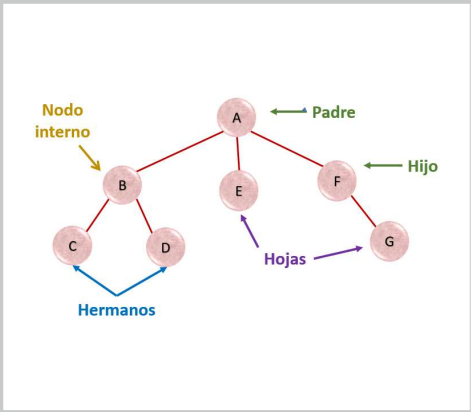
  Descendientes o hijos del nodo B: C, D
- **Hijo:** el nodo debajo de un nodo dado (padre) conectado por su arista hacia abajo. Cada nodo puede tener un número arbitrario de hijos.

  Hijos del nodo A: B, E, F. Estos a su vez conforman los subárboles del árbol general.
- **Nodos hermanos:** son los sucesores o descendientes directos de un mismo nodo (hijos de un mismo padre).

  Nodos hermanos (hijos de B): C, D
- **Hojas:** son los nodos sin hijos. También se les llama nodos terminales, nodos de grado o nodos externos.

  Nodos hojas: C, D, E, G
- **Nodos internos:** son los nodos que tienen al menos un hijo.

  Nodos internos: A, B, F

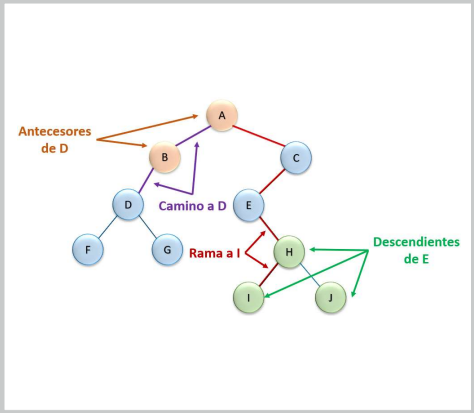


6

- **Camino de un nodo:** es la secuencia de aristas a través de las cuales se pasa desde el nodo raíz a un nodo.
- **Rama:** Un camino que termina en una hoja.
- **Antecesores o predecesores de un nodo:** son todos los nodos del camino que va desde la raíz del árbol hasta el nodo.

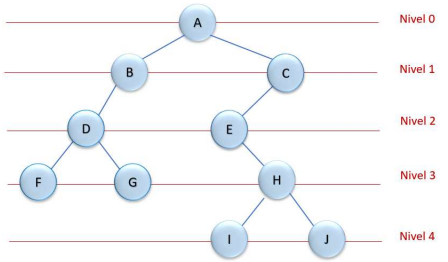
  Antecesores de D: B, A
- **Descendientes o sucesores de un nodo:** son aquellos nodos accesibles por un camino que comience en el nodo.

  Descendientes del nodo E: H, I, J



7

- **Grado de un nodo:** es el número de hijos que tiene el nodo. Así, el grado de un nodo hoja es cero. En la figura anterior el grado del nodo D es 2 y el grado del nodo E es 1.
- **Grado del árbol:** es el mayor grado de sus nodos. Por ejemplo, el árbol binario es de grado 2 porque cada nodo tiene como mucho dos descendientes directos.
- **Niveles:** es la distancia desde la raíz en la que se encuentra ubicado cada nodo. Cada nodo de un árbol tiene asignado un número de nivel de la siguiente forma la raíz tiene el número de nivel 0, y al resto de los nodos se le asigna un número de nivel que es mayor en 1 que el número de nivel del padre.
- **Nivel de un nodo:** se refiere a la distancia del nodo desde la raíz del árbol.
- **Altura de un nodo:** es la longitud del camino más largo que comienza en el nodo y termina en una hoja.
  - La altura de un nodo hoja es 0
  - La altura de un nodo es igual a la mayor altura de sus hijos + 1. Por ejemplo, la altura del nodo B en la figura es 2.
- **Altura de un árbol:** es la longitud de la rama más larga del árbol más uno. Equivale a 1 más que el mayor número de nivel del árbol.
- **Profundidad de un nodo:** es la longitud del camino (único) que comienza en la raíz y termina en el nodo. También se denomina nivel.
  - La profundidad de la raíz es 0
  - La profundidad de un nodo es igual a la profundidad de su padre + 1



8

Árboles binarios

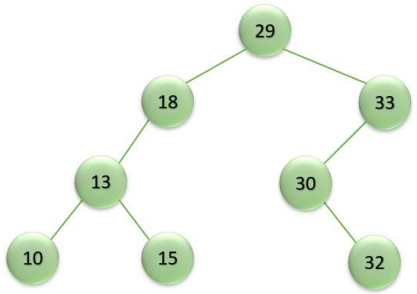
El árbol binario es un árbol en el cual cada nodo tiene como máximo dos hijos, uno a la izquierda y el otro a la derecha.

En un **árbol binario de búsqueda** el hijo izquierdo, si existe, debe tener un valor menor que el valor de su padre y el hijo derecho, si existe, debe tener un valor mayor que el valor de su padre.

9

Árbol binario de búsqueda

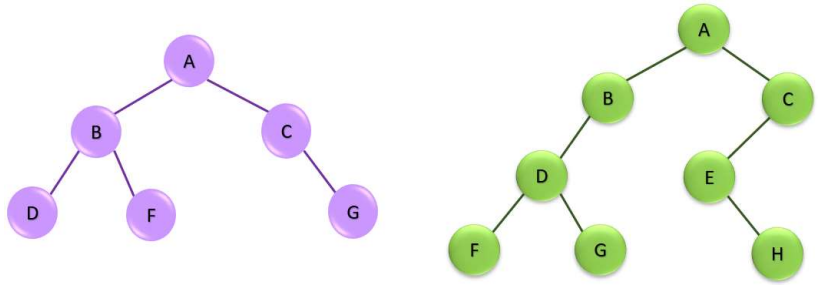
El número máximo de nodos en cualquier nivel N de un árbol binario es 2N.



10

Árboles binarios distintos

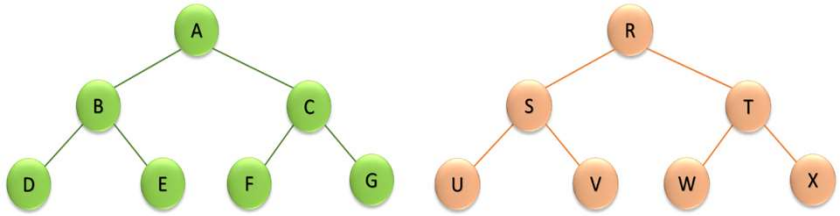
Dos árboles binarios son distintos cuando sus estructuras son diferentes.



11

Árboles binarios similares

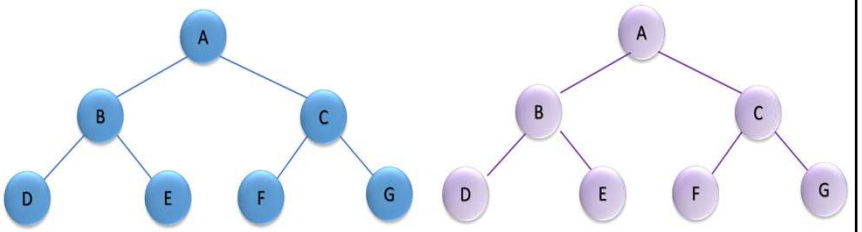
Dos árboles binarios son similares si sus estructuras (forma) son idénticas pero la información que contienen sus nodos difiere entre sí.



12

Árboles binarios equivalentes

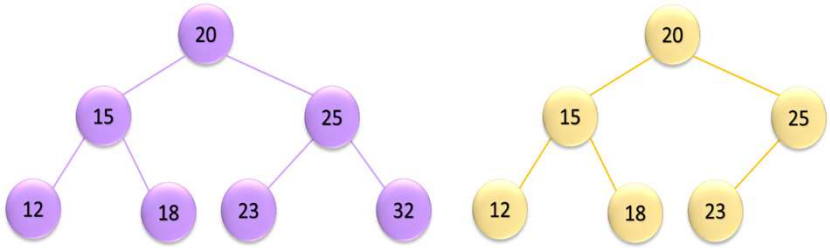
Son aquellos que son similares y los nodos contienen la misma información.



13

Árbol binario completo

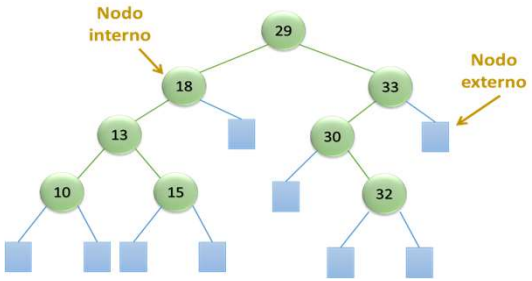
El árbol binario es completo si todos sus niveles, excepto posiblemente el último, tienen el máximo número de nodos posibles y si todos los nodos del último nivel están situados lo más posible a la izquierda.



14

Árboles binarios extendidos: árboles-2

Es un árbol binario en donde el número de hijos de cada nodo es igual al grado del árbol, en este caso, es 2.



15

Representación de árboles en memoria

- **Por medio de punteros o enlazada**: Es análoga a la forma en que se representan las listas enlazadas en memoria.
- **Por medio de arreglos o secuencial**: Utiliza un arreglo simple.

16

Representación enlazada

Un árbol binario se puede representar en memoria de forma enlazada utilizando tres arreglos paralelos: INFO, IZQ y DER (como se muestra en la figura) y una variable puntero a la que llamaremos RAIZ.

IZQ	INFO	DER

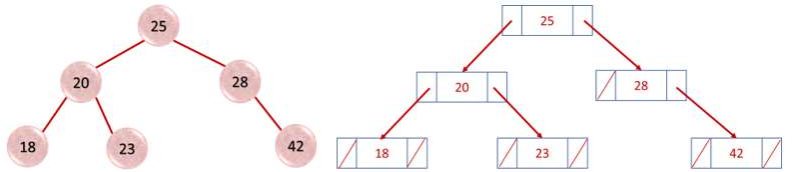
A cada nodo N del árbol le corresponderá una posición K, tal que:

- INFO [K] contendrá los datos del nodo N.
- IZQ [K] contendrá la localización del hijo izquierdo del nodo N.
- DER [K] contendrá la localización del hijo derecho del nodo N.
- La raíz contendrá la posición de la raíz R del árbol. Cuando el árbol está vacío, la RAIZ contendrá el valor nulo.

17

Representación enlazada

En el ejemplo se muestra la representación enlazada en memoria de un árbol binario de búsqueda. Observe que cada nodo está dibujado con sus tres campos y que los subárboles vacíos están dibujados usando un diagonal / para las entradas nulas.



18

Representación secuencial

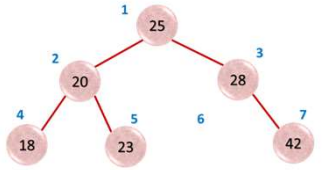
Esta representación usa un arreglo lineal al que llamaremos ARBOL.

- La raíz R del árbol se guarda en la posición del arreglo ARBOL [1].
- Si un nodo N está ubicado en la posición ARBOL [K], entonces sus hijos:
  - izquierdo está en la posición ARBOL [2*K]
  - derecho en la posición ARBOL [2*K+1]
- Se usa nulo para indicar que el árbol o un subárbol está vacío, así ARBOL [1] = NULO indica que el árbol está vacío.

19

Representación secuencial

En el siguiente ejemplo se mostrará la representación secuencial en memoria de un árbol binario de búsqueda. Observe que cada nodo está ubicado en el arreglo en la misma posición que tiene en el árbol. Para una mejor comprensión de dichas posiciones se ha colocado el número que le corresponde a cada una de ellas al lado del nodo en el árbol.



ARBOL

POSICIÓN	1	2	3	4	5	6	7
VALOR	25	20	28	18	23		42

20

Recorridos en un árbol binario

El recorrido en un árbol es el proceso de visitar todos los nodos de un árbol. Se clasifican los algoritmos de recorrido, dependiendo del orden en que se visitan los nodos.

- AMPLITUD
- PROFUNDIDAD
  - PREORDEN
  - INORDEN
  - POSTORDEN

21

Recorridos en un árbol binario

Recorrido en amplitud

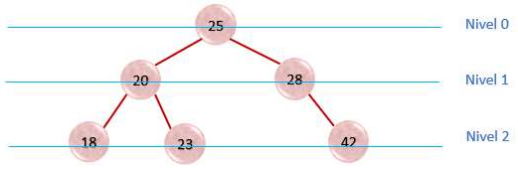
- Se implementa utilizando la estructura de datos denominada cola.
- El recorrido empieza desde la raíz y atraviesa el árbol nivel por nivel, pasa por todos los nodos de un nivel antes de pasar a los nodos hijos.

Recorrido en profundidad

- Se implementa utilizando la estructura de datos denominada pila.
- El algoritmo empieza desde la raíz y visita a todos los nodos de una sola rama del árbol. Cuando termina en esa rama, entonces hace una vuelta hacia atrás (denominado backtracking) y sigue por la otra rama.

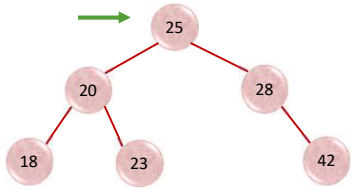
22

Ejemplo del recorrido en amplitud



23

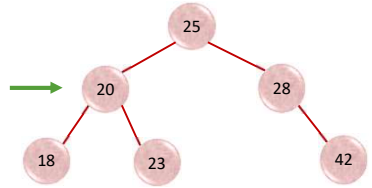
Ejemplo del recorrido en amplitud



25

24

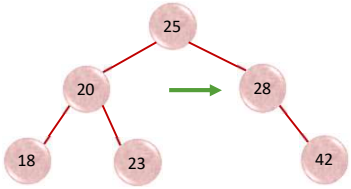
Ejemplo del recorrido en amplitud



25, 20

25

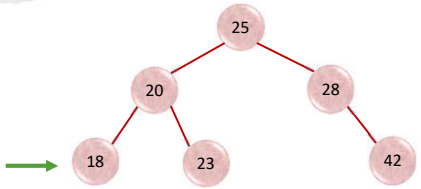
Ejemplo del recorrido en amplitud



25, 20, 28

26

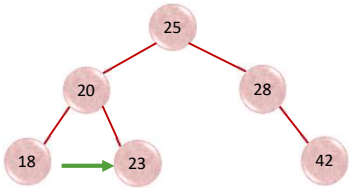
Ejemplo del recorrido en amplitud



25, 20, 28, 18

27

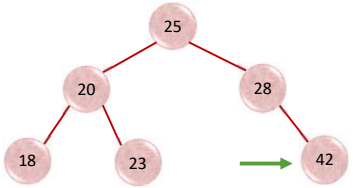
Ejemplo del recorrido en amplitud



25, 20, 28, 18, 23

28

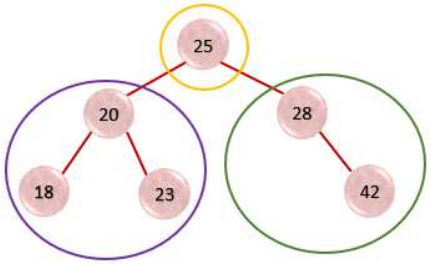
Ejemplo del recorrido en amplitud



25, 20, 28, 18, 23, 42

29

Ejemplos de los recorridos en profundidad

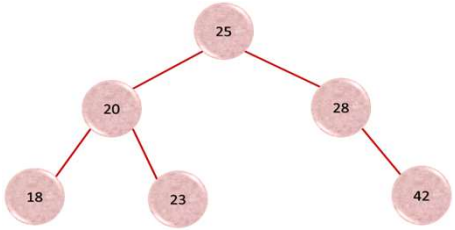


30

Recorrido preorden

Realiza el recorrido siguiendo el siguiente orden:

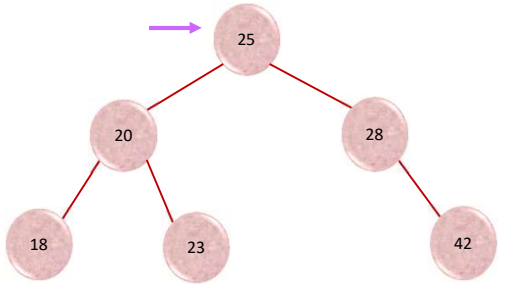
- **raíz**
- **subárbol izquierdo**
- **subárbol derecho**



31

Recorrido preorden

raíz , izquierda, derecha



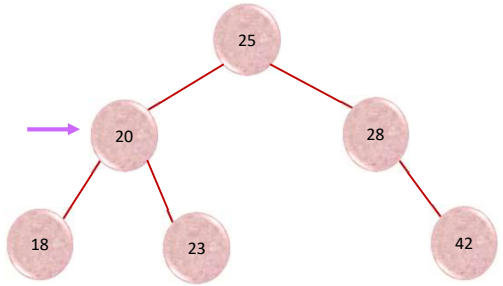
25

32

Recorrido preorden

raíz , izquierda, derecha

25, 20

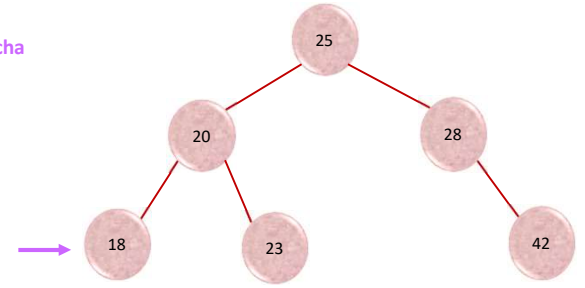


33

Recorrido preorden

raíz , **izquierda**, derecha

25, 20, 18

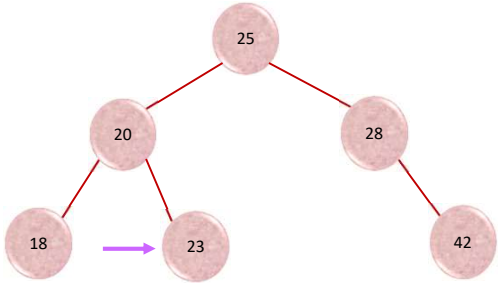


34

Recorrido preorden

raíz , izquierda, **derecha**

25, 20, 18, 23

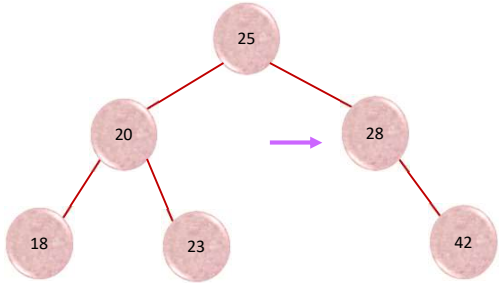


35

Recorrido preorden

raíz , izquierda, derecha

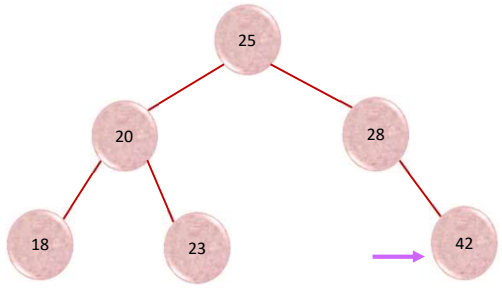
25, 20, 18, 23, 28



36

Recorrido preorden

raíz , izquierda, derecha



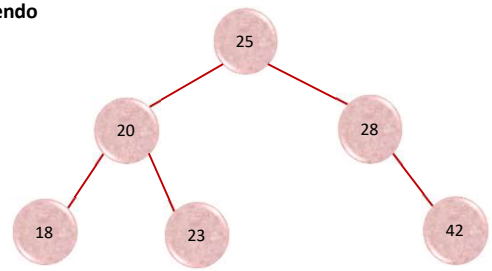
25, 20, 18, 23, 28, 42

37

Recorrido inorden

Realiza el recorrido siguiendo el siguiente orden:

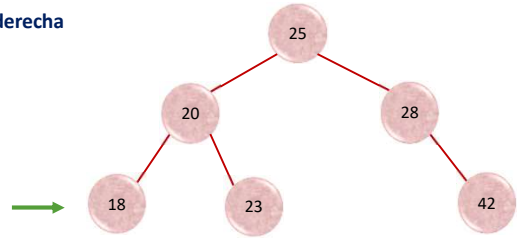
- **subárbol izquierdo**
- **raíz**
- **subárbol derecho**



38

Recorrido inorden

izquierda, raíz, derecha

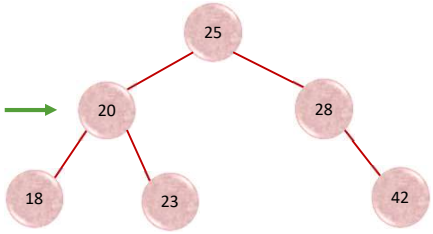


18

39

Recorrido inorden

izquierda, raíz, derecha

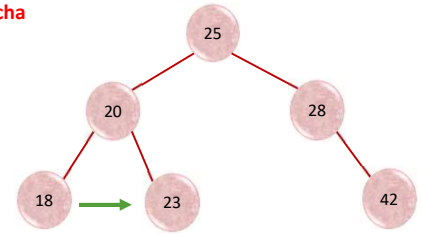


18, 20

40

Recorrido inorden

izquierda, raíz, derecha

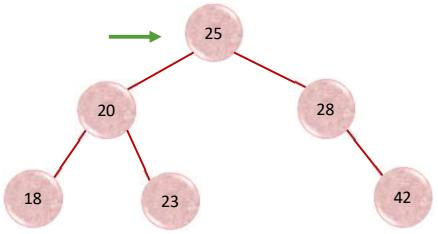


18, 20, 23

41

Recorrido inorden

izquierda, raíz, derecha

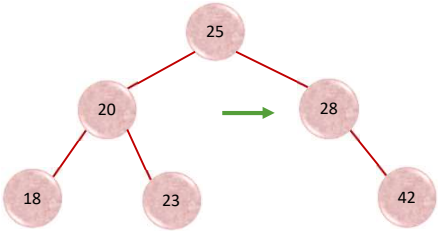


18, 20, 23, 25

42

Recorrido inorden

izquierda, raíz, derecha

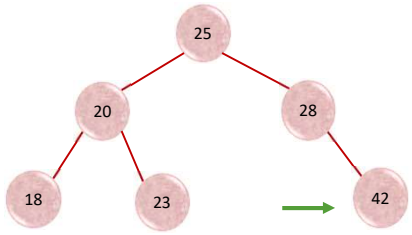


18, 20, 23, 25, 28

43

Recorrido inorden

izquierda, raíz, derecha



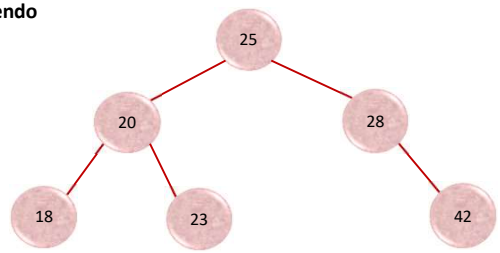
18, 20, 23, 25, 28, 42

44

Recorrido postorden

Realiza el recorrido siguiendo el siguiente orden:

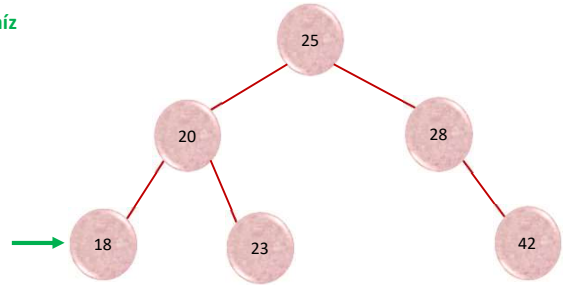
- **subárbol izquierdo**
- **subárbol derecho**
- **raíz**



45

Recorrido postorden

izquierda, derecha, raíz

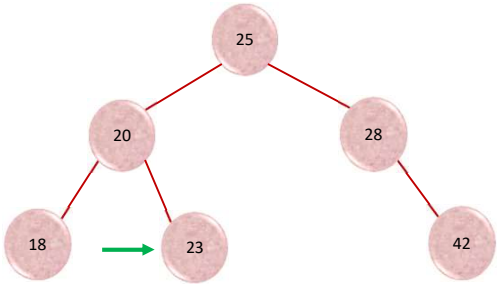


18,

46

Recorrido postorden

izquierda, derecha, raíz

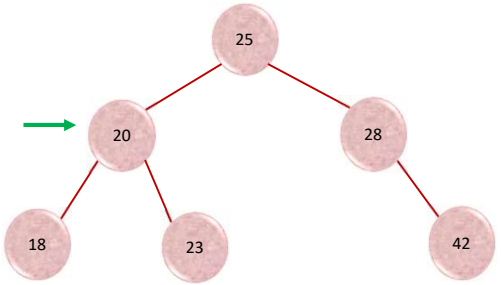


18, 23

47

Recorrido postorden

izquierda, derecha, raíz

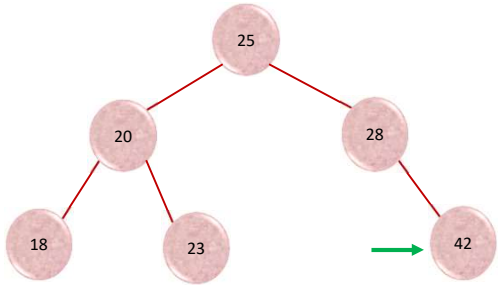


18, 23, 20

48

Recorrido postorden

izquierda, derecha, raíz

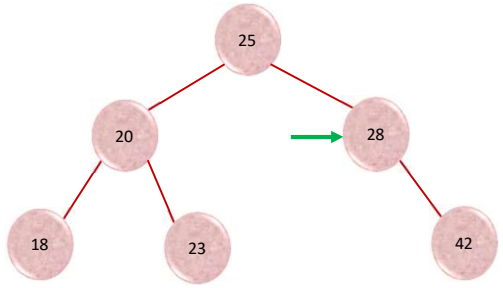


18, 23, 20, 42

49

Recorrido postorden

izquierda, derecha, raíz

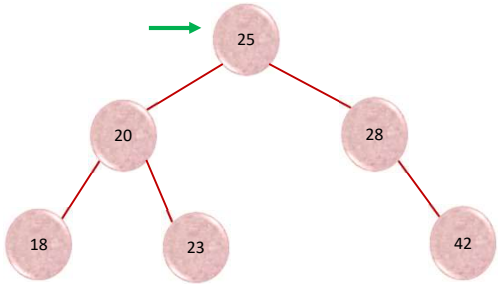


18, 23, 20, 42, 28

50

Recorrido postorden

izquierda, derecha, raíz

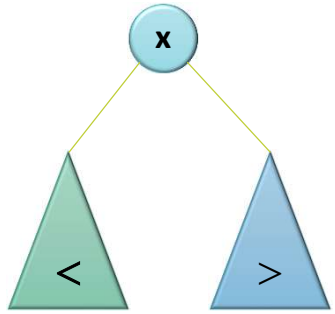


18, 23, 20, 42, 28,25

51

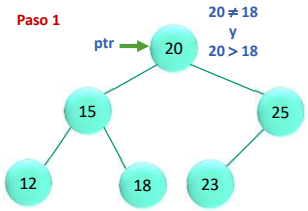
Operaciones sobre un árbol binario

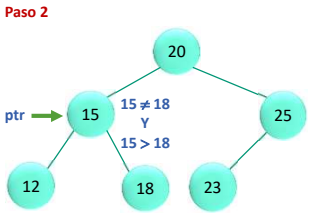
En un árbol binario los nodos almacenan elementos cuyos valores son comparables mediante menor " < " y mayor " > " debido a que los mismos cumplen la propiedad de ordenación la cual indica que: "todo nodo es mayor que los nodos de su subárbol izquierdo, y menor que los nodos de su subárbol derecho".



52

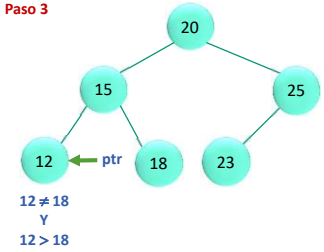
Búsqueda en un árbol binario

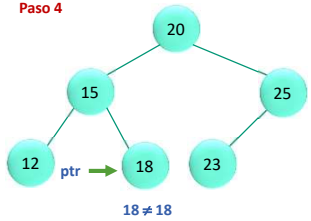




53

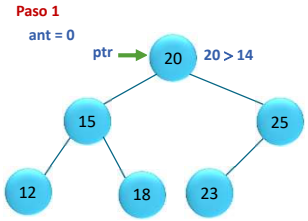
Búsqueda en un árbol binario

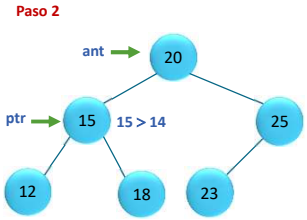




54

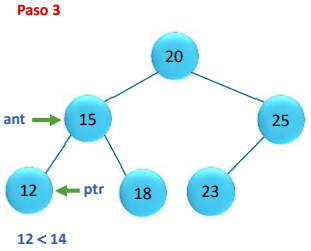
Inserción en un árbol binario

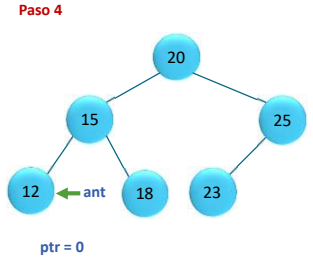




55

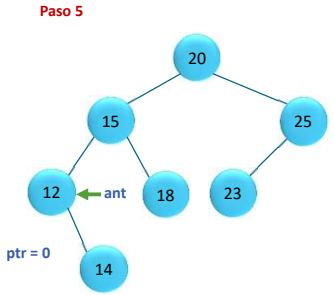
Inserción en un árbol binario





56

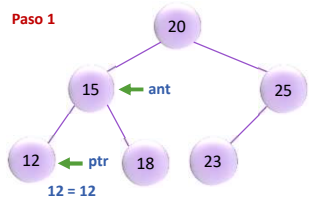
Inserción en un árbol binario

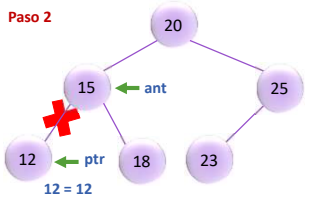


57

Eliminación en un árbol binario

ELIMINAR UNA HOJA

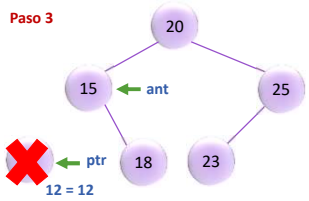


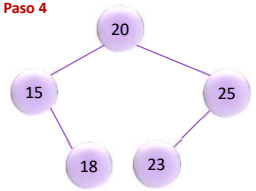


58

Eliminación en un árbol binario

ELIMINAR UNA HOJA

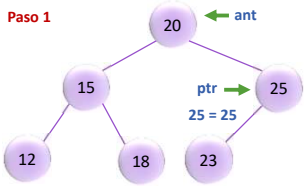


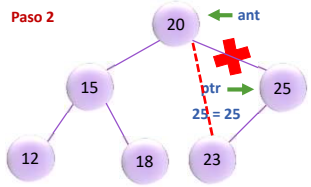


59

Eliminación en un árbol binario

ELIMINAR -- NODO CON UN SOLO HIJO

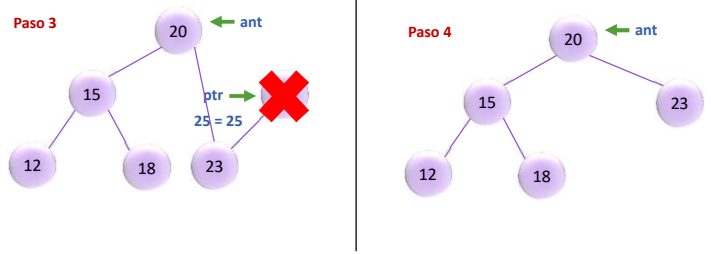




60

Eliminación en un árbol binario

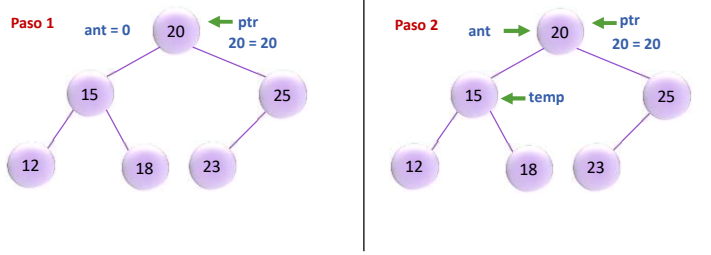
ELIMINAR -- NODO CON UN SOLO HIJO



61

Eliminación en un árbol binario

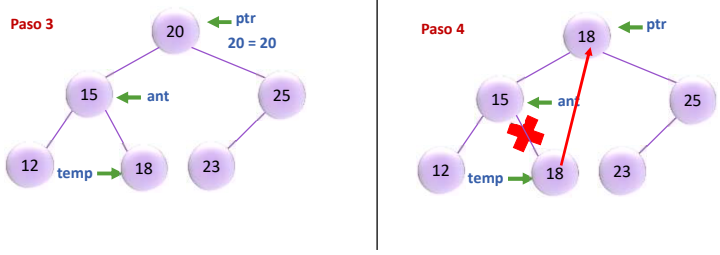
ELIMINAR -- NODO CON DOS HIJOS



62

Eliminación en un árbol binario

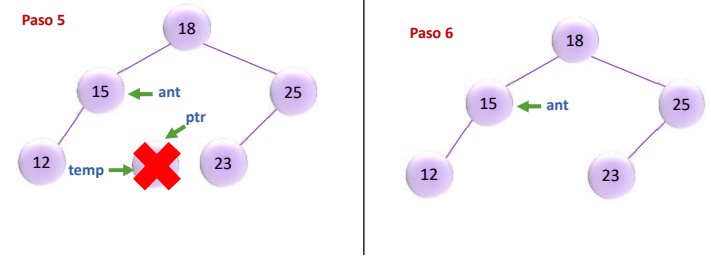
ELIMINAR -- NODO CON DOS HIJOS



63

Eliminación en un árbol binario

ELIMINAR -- NODO CON DOS HIJOS



64

Búsqueda en árbol binario

65

Algoritmo de Búsqueda – Variables utilizadas

ptr – variable puntero

Raízarbol – raíz del árbol

ValorEnArbol – variable booleana usada para indicar si el valor está en el árbol

ClaveNueva – valor que se está buscando

ptr.info – valor al que apunta el puntro ptr

ptr.izquierdo – variable usada para indicar que nos movemos a la izquierda del puntero ptr

ptr.derecho – variable usada para indicar que nos movemos a la derecha del puntero ptr

66

Algoritmo de Búsqueda

```
{

/* Busca en el árbol binario de búsqueda un nodo cuya clave
es ClaveNueva, y devuelve los datos Info del nodo */

ptr ← Raízarbol

ValorEnArbol ← Falso
```

67

Algoritmo de Búsqueda

```
// Encontrar nodo del árbol que contiene ClaveNueva

/* Al final del ciclo, si ClaveNueva está en el árbol, ptr apuntará
a su nodo */

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer

Si (ptr.info = ClaveNueva)
        Entonces ValorEnArbol ← cierto
        Sino     //Continua buscando
                 Si (ptr.info > ClaveNueva)
                          Entonces ptr ← ptr.izquierdo
                          Sino ptr ← ptr.derecho
                 Fin-si
Fin-si

Fin-mientras
```

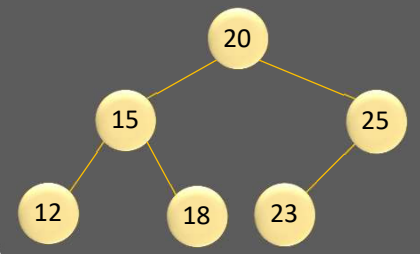
68

Algoritmo de Búsqueda

```
// Si ClaveNueva se encontraba en el árbol, copia su parte info
Si (ValorEnArbol = cierto)
Entonces Imprimir ("El valor: ", ClaveNueva, "ha sido encontrado")
Sino Imprimir ("El valor: ", ClaveNueva, "no ha sido encontrado")
Fin-si
}
```

69

EJEMPLO

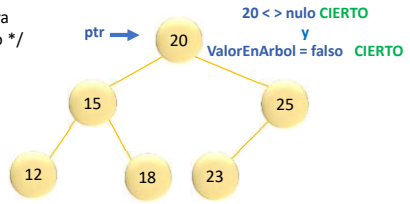


ClaveNueva = 18

70

Paso 1

```
{
/* Busca en el árbol binario de búsqueda un nodo cuya
clave es ValorClave, y devuelve los datos Info del nodo */
ptr ← Raízarbol
ValorEnArbol ← Falso

// Encontrar nodo del árbol que contiene ClaveNueva
/* Al final del ciclo, si ClaveNueva está en el árbol, ptr
apuntará a su nodo */
Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
```



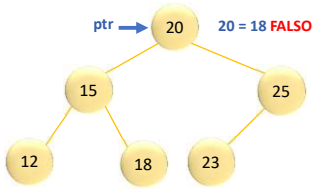
ValorEnArbol = FALSO

ClaveNueva = 18

71

Paso 2

```
{
/* Busca en el árbol binario de búsqueda un nodo cuya
clave es ValorClave, y devuelve los datos Info del nodo */
ptr ← Raízarbol
ValorEnArbol ← Falso
// Encontrar nodo del árbol que contiene ClaveNueva
/* Al final del ciclo, si ClaveNueva está en el árbol, ptr
apuntará a su nodo */
Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
Si (ptr.info = ClaveNueva)
```



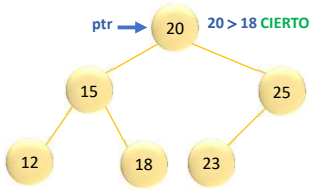
ValorEnArbol = FALSO

ClaveNueva = 18

72

Paso 3

{
/* Busca en el árbol binario de búsqueda un nodo cuya clave es ValorClave, y devuelve los datos Info del nodo */
ptr ← Raízarbol
ValorEnArbol ← Falso
// Encontrar nodo del árbol que contiene ClaveNueva
/* Al final del ciclo, si ClaveNueva está en el árbol, ptr apuntará a su nodo */
Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
Si (ptr.info = ClaveNueva)
Entonces ValorEnArbol ← cierto
Sino //Continua buscando
Si (ptr.info > ClaveNueva)
Entonces ptr ← ptr.izquierdo
Sino ptr ← ptr.derecho
Fin-si
Fin-si
Fin-mientras



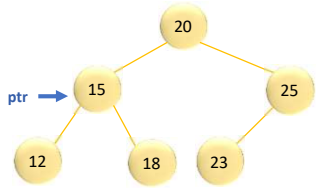
ValorEnArbol = FALSO

ClaveNueva = 18

73

Paso 3

{
/* Busca en el árbol binario de búsqueda un nodo cuya clave es ValorClave, y devuelve los datos Info del nodo */
ptr ← Raízarbol
ValorEnArbol ← Falso
// Encontrar nodo del árbol que contiene ClaveNueva
/* Al final del ciclo, si ClaveNueva está en el árbol, ptr apuntará a su nodo */
Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
Si (ptr.info = ClaveNueva)
Entonces ValorEnArbol ← cierto
Sino //Continua buscando
Si (ptr.info > ClaveNueva)
Entonces ptr ← ptr.izquierdo
Sino ptr ← ptr.derecho
Fin-si
Fin-si
Fin-mientras



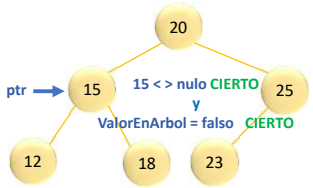
ValorEnArbol = FALSO

ClaveNueva = 18

74

Paso 4

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer



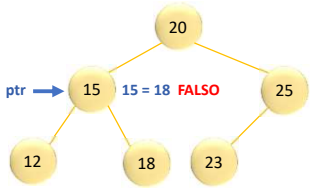
ValorEnArbol = FALSO

ClaveNueva = 18

75

Paso 5

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
Si (ptr.info = ClaveNueva)



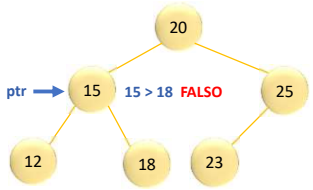
ValorEnArbol = FALSO

ClaveNueva = 18

76

Paso 6

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
  Si (ptr.info = ClaveNueva)
    Entonces ValorEnArbol ← cierto
    Sino //Continua buscando
    Si (ptr.info > ClaveNueva)



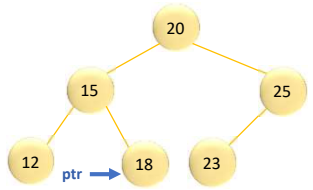
ValorEnArbol = FALSO

ClaveNueva = 18

77

Paso 6

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
  Si (ptr.info = ClaveNueva)
    Entonces ValorEnArbol ← cierto
    Sino //Continua buscando
      Si (ptr.info > ClaveNueva)
        Entonces ptr ← ptr.izquierdo
        Sino ptr ← ptr.derecho
      Fin-si
  Fin-si
Fin-mientras



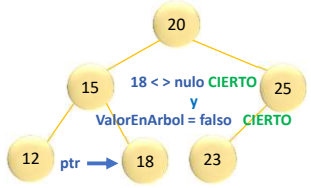
ValorEnArbol = FALSO

ClaveNueva = 18

78

Paso 7

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer



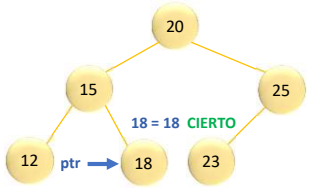
ValorEnArbol = FALSO

ClaveNueva = 18

79

Paso 8

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
  Si (ptr.info = ClaveNueva)
    Entonces ValorEnArbol ← cierto
    Sino //Continua buscando
      Si (ptr.info > ClaveNueva)
        Entonces ptr ← ptr.izquierdo
        Sino ptr ← ptr.derecho
      Fin-si
  Fin-si
Fin-mientras

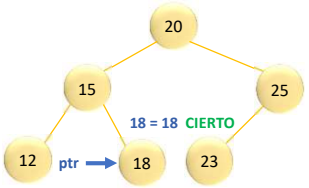


ClaveNueva = 18

80

Paso 8

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
Si (ptr.info = ClaveNueva)
Entonces ValorEnArbol ← cierto
Sino //Continua buscando
Si (ptr.info > ClaveNueva)
Entonces ptr ← ptr.izquierdo
Sino ptr ← ptr.derecho
Fin-si
Fin-si
Fin-mientras



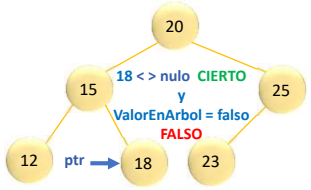
ValorEnArbol = CIERTO

ClaveNueva = 18

81

Paso 9

Mientras (ptr < > nulo) y (ValorEnArbol = falso) hacer
Si (ptr.info = ClaveNueva)
Entonces ValorEnArbol ← cierto
Sino //Continua buscando
Si (ptr.info > ClaveNueva)
Entonces ptr ← ptr.izquierdo
Sino ptr ← ptr.derecho
Fin-si
Fin-si
Fin-mientras



ValorEnArbol = CIERTO

ClaveNueva = 18

82

Paso 9

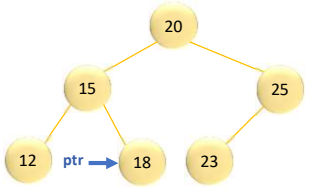
// Si ClaveNueva se encontraba en el árbol, copia su parte info

Si (ValorEnArbol = cierto)

Entonces Imprimir ("El valor: ", ClaveNueva, "ha sido encontrado")

Sino Imprimir ("El valor: ", ClaveNueva, "no ha sido encontrado")

Fin-si
}



ValorEnArbol = CIERTO

ClaveNueva = 18

83

Paso 9

El valor: 18 ha sido encontrado

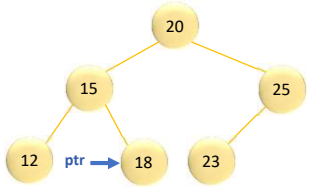
// Si ClaveNueva se encontraba en el árbol, copia su parte info

Si (ValorEnArbol = cierto)

Entonces Imprimir ("El valor: ", ClaveNueva, "ha sido encontrado")

Sino Imprimir ("El valor: ", ClaveNueva, "no ha sido encontrado")

Fin-si
}



ValorEnArbol = CIERTO

ClaveNueva = 18

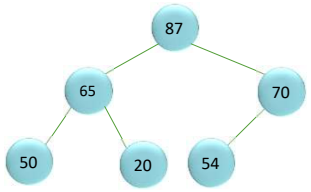
84

Árboles en montón

Un árbol en montón H (también llamado montículo) es usado para implementar colas de prioridad.

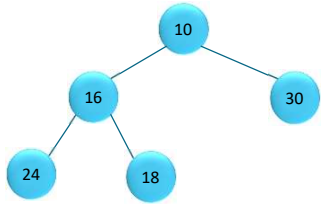
85

Árbol en montón máximo



El valor de N es mayor o igual que el valor de cualquier hijo de N.

Árbol en montón mínimo



El valor de N es menor o igual que el valor de cualquier hijo de N.

86

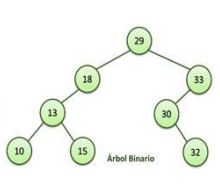
Algoritmo de Huffman

87

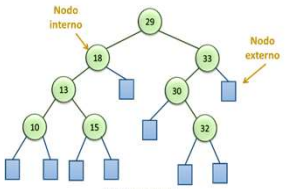
Algoritmo de Huffman

Este algoritmo trabaja con árboles binarios extendidos o árboles-2. En donde:

- **el número de nodos externos (N_E):** es 1 más que el número de nodos internos (N_I), $N_E = N_I + 1$
- **la longitud de camino interna (L_I):** es la suma de todas las longitudes de camino obtenidos sobre cada camino desde la raíz del árbol hasta un nodo interno.
- **la longitud de camino externa (L_E):** es la suma de todas las longitudes de camino obtenidos sobre cada camino desde la raíz del árbol hasta un nodo externo. También se puede encontrar si se conoce la longitud de camino interna L_I usando la siguiente fórmula: $L_E = L_I + 2n$, donde **n** es el número de nodos internos del árbol.
- la **longitud de camino con peso (externa) P:** se define como la suma de las longitudes de camino con sus pesos: $P = W_1L_1 + W_2L_2 + \ldots + W_nL_n$, donde W_iL_i denotan, respectivamente, el peso y la longitud del camino del nodo externo N_i.



Árbol Binario

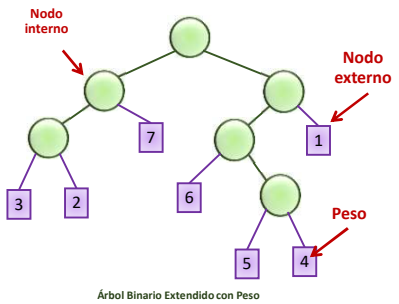


Árbol Binario Extendido

88

Construcción del árbol de mínima longitud de camino con peso mediante el Algoritmo de Huffman

1. Se eligen dos de los subárboles con la menor combinación de pesos posible. (Recuerde que cada elemento pertenece a su propio subárbol).
2. Se unen para formar un nuevo subárbol con peso igual a la suma de ambos subárboles.
3. Se repiten los pasos anteriores hasta que se termine de formar el árbol.

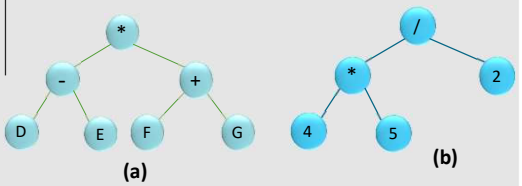


Árbol Binario Extendido con Peso

89

Árboles binarios de expresión

Los árboles binarios de expresión son aquellos árboles utilizados para representar una expresión binaria en el cual la raíz del nodo contiene el operador y los dos hijos contienen los operandos. Cuando usamos un árbol binario para representar una expresión, los paréntesis no son necesarios para indicar la precedencia. Los niveles de los nodos del árbol indican implícitamente la precedencia relativa de evaluación.



90

Construcción de un árbol binario de expresión

Paso 1

EXPRESIÓN
/ A - B C ;

ultimosímbolo = ;
Símbolo = /

/ ← Raíz

Nuevonodo

Sigmov = izquierda

PILA

EXPRESIÓN
/ A - B C ;

Símbolo = A

91

Construcción de un árbol binario de expresión

Paso 2

A ≠ ;
ultimonodo = /

A

Nuevonodo

Sigmov = izquierda

Raíz → /

A

PILA

/

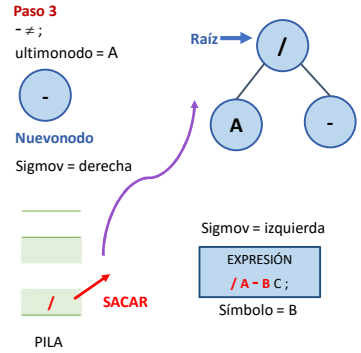
Sigmov = derecha

EXPRESIÓN
/ A - B C ;

Símbolo = -

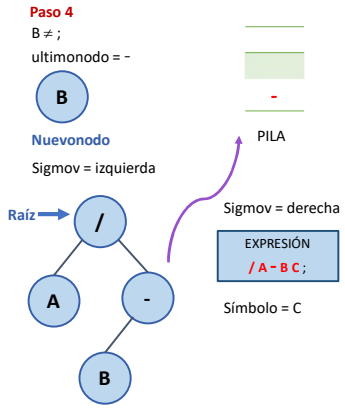
92

Construcción de un árbol binario de expresión



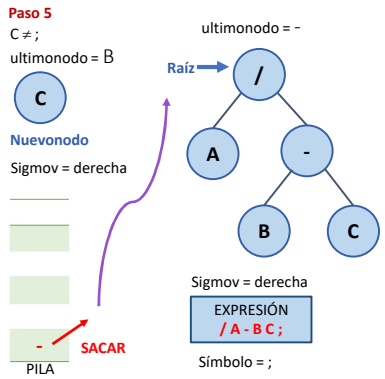
93

Construcción de un árbol binario de expresión



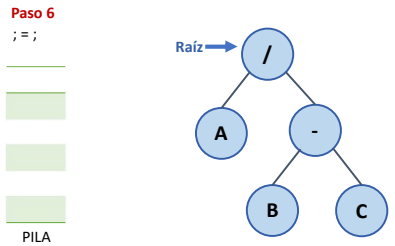
94

Construcción de un árbol binario de expresión



95

Construcción de un árbol binario de expresión



96

Estructuras de datos tipo grafo

Un grafo está formado por un conjunto de nodos (llamados también vértices) y un conjunto de líneas llamadas aristas (o arcos) que conectan los diferentes nodos.

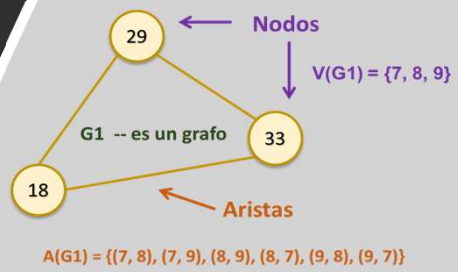
97

Definición formal de un grafo

G = (V, A)

donde

- **V(G)** es un conjunto finito, no vacío de vértices que se especifican listando los nodos en notación conjunto, es decir, dentro de paréntesis o llaves.
- **A(G)** es un conjunto de aristas (pares de vértices) que se especifica listando una secuencia de aristas. Cada arista se denota escribiendo los nombres de los dos nodos que conecta entre paréntesis, con una coma entre ellos.



A(G1) = {(7, 8), (7, 9), (8, 9), (8, 7), (9, 8), (9, 7)}

98

Clasificación de los grafos

Grafo dirigido

- La dirección de la línea es indicada por el nodo que se lista primero.

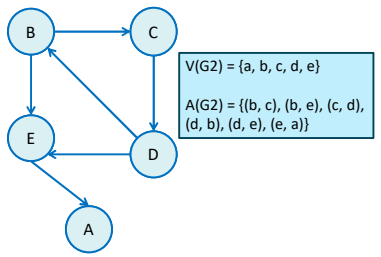
Grafo no dirigido

- La relación entre los dos nodos es desordenada. Es decir, un nodo apunta al otro; ellos están simplemente conectados.

99

Clasificación de los grafos

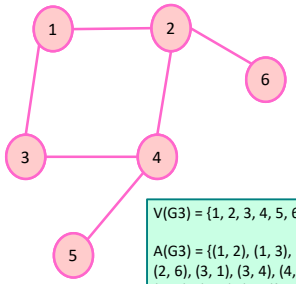
Grafo dirigido



V(G2) = {a, b, c, d, e}

A(G2) = {(b, c), (b, e), (c, d), (d, b), (d, e), (e, a)}

Grafo no dirigido



V(G3) = {1, 2, 3, 4, 5, 6}

A(G3) = {(1, 2), (1, 3), (2, 1), (2, 4), (2, 6), (3, 1), (3, 4), (4, 2), (4, 3), (4, 5), (5, 4), (6, 2)}

100

Tipos particulares de grafos

Grafo acíclico

- Es aquel grafo no contiene ningún ciclo simple.

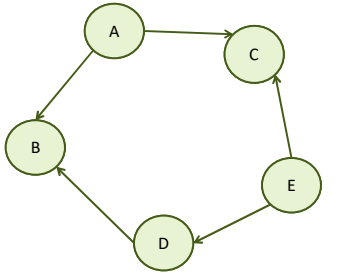
Grafo cíclico

- Un grafo se dice cíclico si contiene algún ciclo simple.

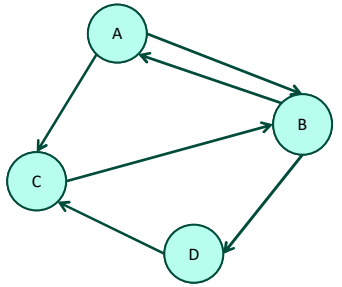
101

Tipos particulares de grafos

Grafo acíclico



Grafo cíclico



102

Tipos particulares de grafos

Grafo regular

- Es un grafo cuyos vértices tienen el mismo grado.

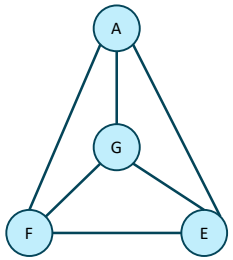
Grafo plano

- Es un grafo que es posible dibujar en el plano sin que ningún par de aristas se crucen entre sí.

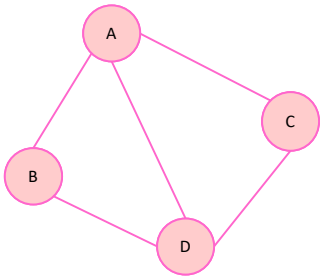
103

Tipos particulares de grafos

Grafo regular



Grafo plano



104

Tipos particulares de grafos

Grafo simple

- Es aquel que acepta una sola una arista uniendo dos vértices cualesquiera. Es la definición estándar de un grafo.

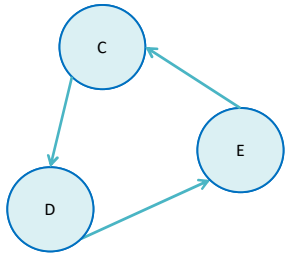
Multigrafo

- Son grafos que aceptan más de una arista entre dos vértices. Estas aristas se llaman múltiples o lazos. Los grafos simples son una subclase de esta categoría de grafos.

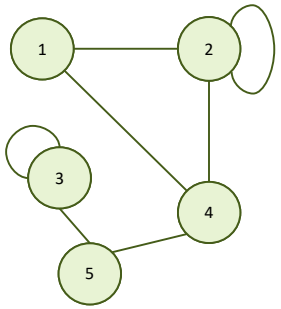
105

Tipos particulares de grafos

Grafo simple



Multigrafo



106

Tipos particulares de grafos

Grafo vacío

- Es el grafo cuyo conjunto de aristas es vacío.

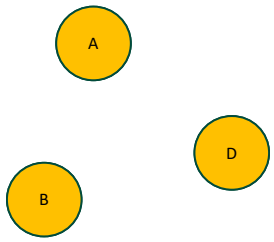
Grafo completo

- Un grafo es completo si cada vértice tiene un grado igual a n-1, donde n es el número de vértice que compone el grafo. Además, es un grafo simple en el que cada vértice es adyacente a cualquier todo otro vértice.

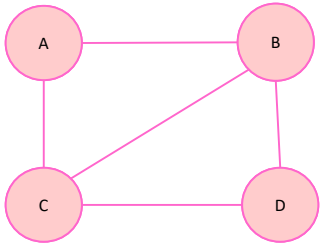
107

Tipos particulares de grafos

Grafo vacío



Grafo completo



108

Tipos particulares de grafos

Grafo conexo

- Un grafo es conexo si es posible formar un camino desde cualquier vértice a cualquier otro en el grafo.

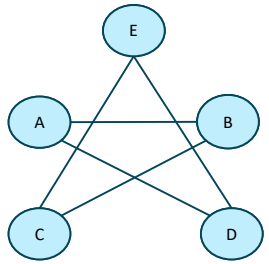
Grafo bipartito

- Es cualquier grafo, cuyos vértices pueden ser divididos en dos conjuntos, tal que no haya aristas entre los vértices del mismo conjunto. Se ve que un grafo es bipartito si no hay ciclos de longitud impar.

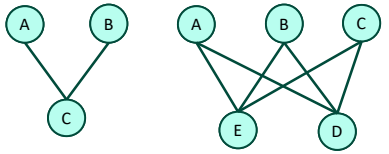
109

Tipos particulares de grafos

Grafo conexo



Grafo bipartito



110

Tipos particulares de grafos

Grafo denso

- Es aquel grafo en el que el número de aristas está cercano al número de máximo de aristas.

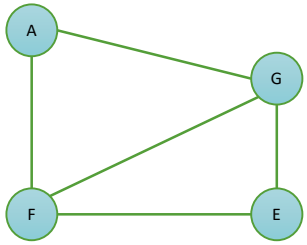
Grafo con peso

- Es un grafo en el que cada arista transporta un valor. Se usan para representar situaciones en las que el valor de la conexión entre los vértices es importante, no sólo la existencia de la conexión.

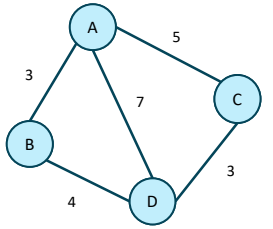
111

Tipos particulares de grafos

Grafo denso



Grafo con peso



Grafo no dirigido con peso en las aristas

112

Otras definiciones de grafos

Grado total de un vértice: corresponde al número de aristas incidentes sobre el vértice, en donde, cada bucle lo cuenta dos veces. En base a esta definición podemos mencionar:

- **Vértice fuente:** es un vértice con grado de entrada cero.
- **Vértice sumidero:** es un vértice con grado de salida cero.
- **Vértice hoja:** es un vértice con grado uno.
- **Vértice aislado:** tiene grado cero.

113

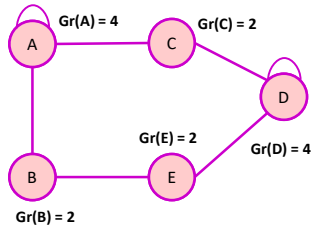
Otras definiciones de grafos

- **Grado total de un vértice (Gr) en un grafo no dirigido:** es igual al número de aristas que tiene el vértice.
- **El grado de entrada (Ge) de un vértice en un grafo dirigido:** es el número de aristas que llegan al vértice.
- **El grado de salida (Gs) de un vértice en un grafo dirigido:** corresponde al número de aristas que salen del vértice.
- **El grado total de un vértice (Gr) en un grafo dirigido:** es la suma del grado entrante más el grado saliente.
- **Camino:** es una sucesión de vértices que conecta al vértice V_i con el vértice V_j. Es decir, debe haber una secuencia ininterrumpida de aristas desde V_i a través de cualquier número de nodos hasta V_j.

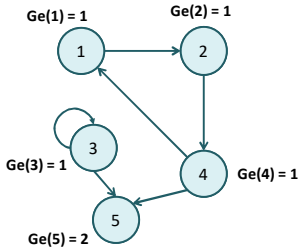
114

Otras definiciones de grafos

Grado total de un vértice (Gr) en un grafo no dirigido



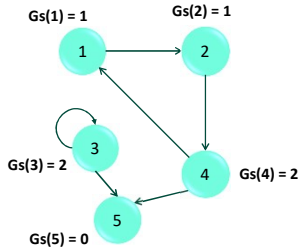
El grado de entrada (Ge) de un vértice en un grafo dirigido



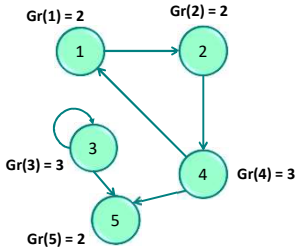
115

Otras definiciones de grafos

El grado de salida (Gs) de un vértice en un grafo dirigido



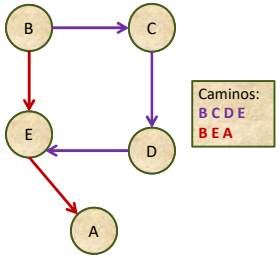
El grado total de un vértice (Gr) en un grafo dirigido



116

Otras definiciones de grafos

Camino



117

Representación secuencial de grafos

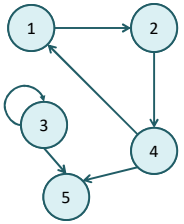
Matriz de adyacencia

Para un grafo con N nodos la matriz de adyacencia será una tabla con N filas y N columnas, en donde, el valor en la posición [i, j] de la tabla será 1 si existe una arista (V_i, V_j) perteneciente al conjunto de las aristas A, y 0 en otro caso. Si el grafo es un grafo con peso, la celda [i, j] contendrá el peso de la arista si la arista está en el conjunto A, y 0 en otro caso.

118

Representación secuencial de grafos

Ejemplo de matriz de adyacencia de un grafo

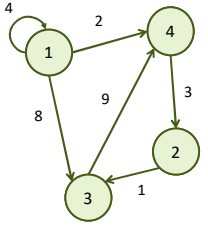


	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	0
3	0	0	1	0	1
4	1	0	0	0	1
5	0	0	0	0	0

119

Representación secuencial de grafos

Ejemplo de matriz de adyacencia de un grafo con peso



	1	2	3	4
1	4	0	8	2
2	0	0	1	0
3	0	0	0	9
4	0	3	0	0

120

Representación secuencial de grafos

Matriz de caminos

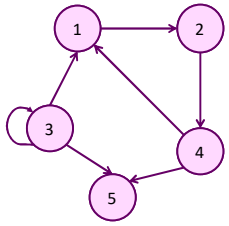
Sea G un grafo dirigido simple con m nodos v1, v2 … vm. La matriz de caminos o matriz de alcance de G es la matriz m-cuadrada P= V(i, j) definida como sigue:

$$P = \begin{cases} 1 & \text{si hay un camino desde } V_i \text{ hasta } V_j \\ 0 & \text{en otro caso} \end{cases}$$

121

Representación secuencial de grafos

Ejemplo de matriz de caminos de un grafo



	1	2	3	4	5
1	1	1	0	1	1
2	1	1	0	1	1
3	1	0	1	0	1
4	1	1	0	1	1
5	0	0	0	0	0

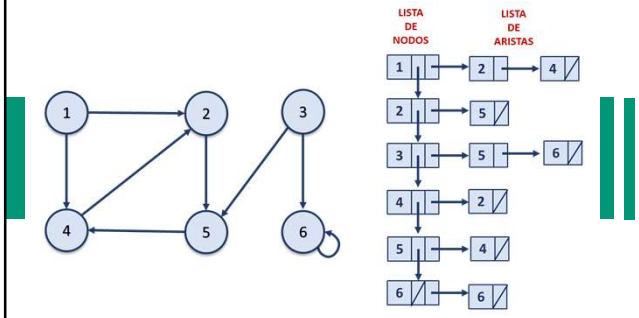
122

Algoritmo de Warshall (Caminos mínimos)

Dado un grafo G(V, A) dirigido se puede aplicar el algoritmo de Warshall para resolver el problema de si hay o no algún camino que una a dos vértices cualquiera. Para esto necesitamos que el valor de cada arista del grafo sea 0 o 1 indicando si existe camino o no entre los dos vértices que la definen.

```
m = cantidad de nodos
{
  Desde (i=1, i > m, i=i+1)
    Desde (j=1, j>m, j=j+1)
      Si (w[i, j] = 0)
        Entonces Q0 [i, j] = ∞
        Sino Q0 [i, j] = w[i, j]
      Fin-si
    Fin-desde
  Fin-desde
  Desde (k=1, k > m, k=k+1)
    Desde (i=1, i > m, i=i+1)
      Desde (j=1, j>m, j=j+1)
        Qk [i, j] = MIN (Qk-1 [i, j], Qk-1 [i, k] + Qk-1 [k, j])
      Fin-desde
    Fin-desde
  Fin-desde
}
```

123

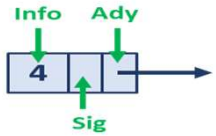


Representación enlazada de grafos

124

Lista de nodos

Lista que contiene los nombres de los nodos. Se conforma de tres partes:



Info
- El nombre o valor clave del nodo

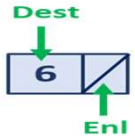
Sig
- Puntero al siguiente nodo de la lista nodo

Ady
- Puntero al primer elemento de la lista de adyacencia del nodo que se mantiene en la lista de aristas

125

Lista de aristas

Lista que contiene la(s) arista(s) a la(s) que el nodo está conectado. Está formada por dos partes:



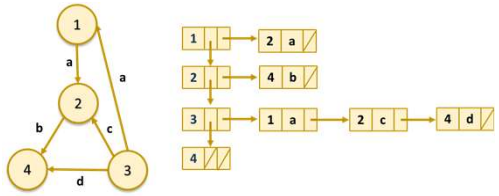
Dest
- Campo que apunta al nodo destino de la arista

Enl
- Campo que enlaza las aristas del mismo nodo inicial

126

Lista de aristas

En los grafos con peso se debe agregar un campo adicional en la lista de las aristas en donde se colocará el peso de las mismas.



127

Operaciones sobre grafos

- Búsqueda
- Inserción
- Eliminación

128

Algoritmo de búsqueda

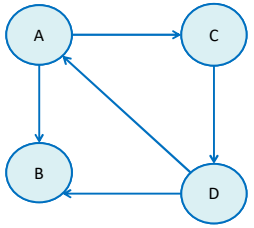
Algoritmo de Búsqueda

```
Inicio
Leer (ítem)  /* lectura del valor a buscar*/
pos = 0
ptr = principio
Mientras (ptr ≠ nulo) hacer
Si (ptr.info = ítem) entonces
                    pos = ptr
                    ptr = 0
          sino
                    ptr = ptr.sig
          fin si
fin mientras
si (pos = nulo) entonces
          imprimir ("No se ha encontrado el valor")
sino
          imprimir ("El valor:", ítem, "ha sido encontrado")
fin si
fin
```

129

Ejemplo del algoritmo de búsqueda

Busque el valor de C el siguiente grafo utilizando el algoritmo de búsqueda. La lista de nodos inicia en el grafo con principio = A.

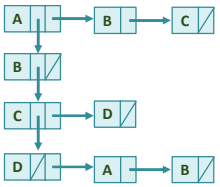


130

Ejemplo del algoritmo de búsqueda

PASO 1

Hacer la representación enlazada del grafo



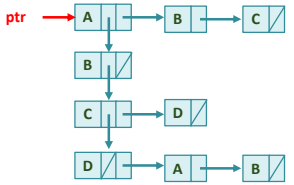
131

Ejemplo del algoritmo de búsqueda

PASO 2

Leer (ítem)
pos = 0
ptr = principio

Item = C
Pos = 0



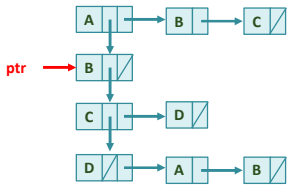
132

Ejemplo del algoritmo de búsqueda

PASO 3

```
Mientras (ptr ≠ nulo) hacer
 Si (ptr.info = ítem) entonces
      pos = ptr
      ptr = 0
 sino
      ptr = ptr.sig
 fin si
fin mientras
```

ptr ≠ nulo
A ≠ C



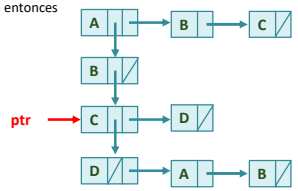
133

Ejemplo del algoritmo de búsqueda

PASO 4

```
Mientras (ptr ≠ nulo) hacer
 Si (ptr.info = ítem) entonces
      pos = ptr
      ptr = 0
 sino
      ptr = ptr.sig
 fin si
fin mientras
```

ptr ≠ nulo
B ≠ C



134

Ejemplo del algoritmo de búsqueda

PASO 5

```
Mientras (ptr ≠ nulo) hacer
 Si (ptr.info = ítem) entonces
      pos = ptr
      ptr = 0
 sino
      ptr = ptr.sig
 fin si
fin mientras
```

ptr ≠ nulo
C = C
pos = C
ptr = 0

PASO 6

```
si (pos = nulo) entonces
   imprimir ("No se ha encontrado el valor")
sino
   imprimir ("El valor:", ítem, "ha sido encontrado")
fin si
fin
```

pos ≠ nulo
El valor: C ha sido encontrado

135

Algoritmo de inserción

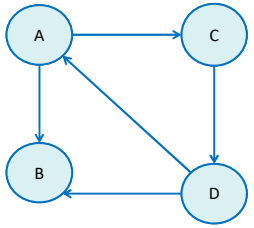
Algoritmo de Inserción

```
inicio
/* lectura del vértice origen (itemA) y destino (itemB) */
leer(itemA, itemB)
/* búsqueda del vértice origen (itemA) en el grafo */
pos = nulo
ptr = principio
Mientras (ptr ≠ nulo) hacer
Si (ptr.info = itemA) entonces
                pos = ptr
                ptr = nulo
        sino
                ptr = ptr.enl
        fin si
fin mientras
si (pos = nulo) entonces
/* crea el nodo origen y lo agrega a la lista de nodos */
nuevo(nodo)
nodo.info = itemA
nodo.enl = principio
nodo.sig = nulo
principio = nodo
pos = nodo
fin si
/* búsqueda del vértice destino (itemB) en el grafo */
pos1 = nulo
ptr1 = principio
Mientras (ptr1 ≠ nulo) hacer
                pos1 = ptr1
                ptr1 = nulo
        sino
                ptr1 = ptr1.enl
        fin si
fin mientras
si (pos1 = nulo) entonces
/* crea el nodo destino y lo agrega a la lista de nodos */
nuevo(nodo)
nodo.info = itemB
nodo.enl = principio
nodo.sig = nulo
principio = nodo
fin si
/* verifica si la arista existe, sino existe crea la arista y la
agrega a la lista de aristas */
mientras (pos.sig ≠ itemB y pos.sig ≠ nulo) hacer
        pos = pos.sig
fin mientras
si (pos.sig = nulo) entonces
/* crea la arista y la agrega a la lista de aristas */
        nueva(arista)
        arista.info = itemB
        arista.sig = nulo
        pos.sig = arista
fin si
fin
```

136

Ejemplo del algoritmo de inserción

Insertar la arista (B, E) el siguiente grafo utilizando el algoritmo de inserción. La lista de nodos inicia en el grafo con principio = A.

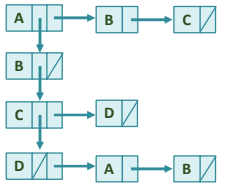


137

Ejemplo del algoritmo de inserción

PASO 1

Hacer la representación enlazada del grafo



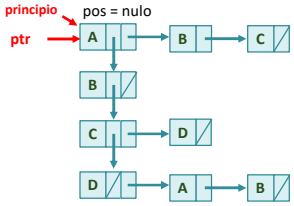
138

Ejemplo del algoritmo de inserción

PASO 2

```
/* lectura del vértice origen (ítemA) y destino (ítemB) */
leer(ítemA, ítemB)
/* búsqueda del vértice origen (ítemA) en el grafo */
pos = nulo
ptr = principio
```

itemA = B
itemB = E
pos = nulo



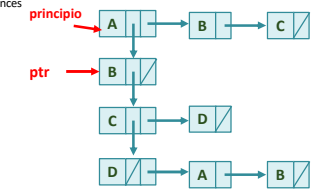
139

Ejemplo del algoritmo de inserción

PASO 3

```
Mientras (ptr ≠ nulo) hacer
  Si (ptr.info = ítemA) entonces
     pos = ptr
     ptr = nulo
  sino
     ptr = ptr.enl
  fin si
fin mientras
```

ptr ≠ nulo
A ≠ B

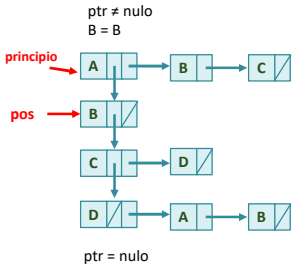


140

Ejemplo del algoritmo de inserción

PASO 4

```
Mientras (ptr ≠ nulo) hacer
  Si (ptr.info = ítemA) entonces
      pos = ptr
      ptr = nulo
  sino
      ptr = ptr.enl
  fin si
fin mientras
```

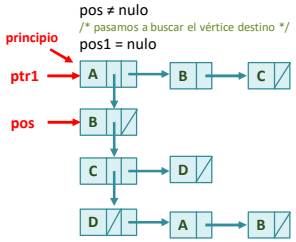


141

Ejemplo del algoritmo de inserción

PASO 5

```
si (pos = nulo) entonces
/* crea el nodo origen y lo agrega
a la lista de nodos */
  nuevo(nodo)
  nodo.info = ítemA
  nodo.enl = principio
  nodo.sig = nulo
  principio = nodo
  pos = nodo
fin si
/* búsqueda del vértice destino
(ítemB) en el grafo */
pos1 = nulo
ptr1 = principio
```

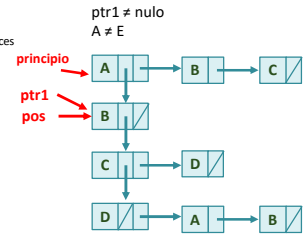


142

Ejemplo del algoritmo de inserción

PASO 6

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
      pos1 = ptr1
      ptr1 = nulo
  sino
      ptr1 = ptr1.enl
  fin si
fin mientras
```

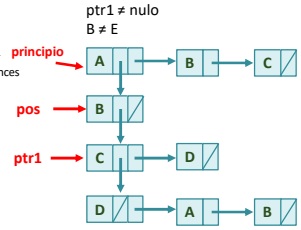


143

Ejemplo del algoritmo de inserción

PASO 7

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
      pos1 = ptr1
      ptr1 = nulo
  sino
      ptr1 = ptr1.enl
  fin si
fin mientras
```

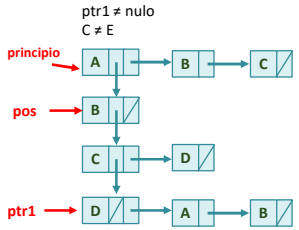


144

Ejemplo del algoritmo de inserción

PASO 8

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
     pos1 = ptr1
     ptr1 = nulo
  sino
     ptr1 = ptr1.enl
  fin si
fin mientras
```

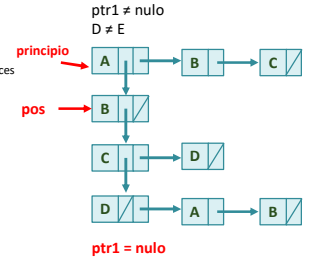


145

Ejemplo del algoritmo de inserción

PASO 9

```
Mientras (ptr1 ≠ nulo) hacer
  Si (ptr1.info = ítemB) entonces
     pos1 = ptr1
     ptr1 = nulo
  sino
     ptr1 = ptr1.enl
  fin si
fin mientras
```

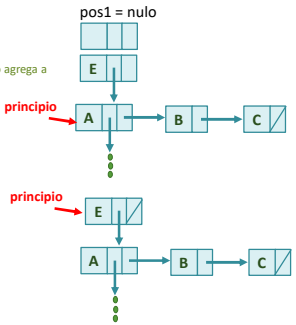


146

Ejemplo del algoritmo de inserción

PASO 10

```
si (pos1 = nulo) entonces
/* crea el nodo destino y lo agrega a
la lista de nodos */
  nuevo(nodo)
  nodo.info = ítemB
  nodo.enl = principio
  nodo.sig = nulo
  principio = nodo
fin si
```

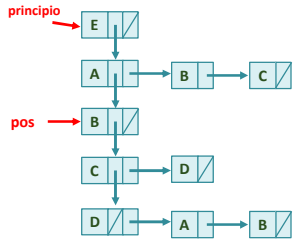


147

Ejemplo del algoritmo de inserción

PASO 11

```
/* verifica si la arista existe, sino
existe crea la arista y la agrega a la
lista de aristas */
mientras (pos.sig ≠ ítemB y pos.sig
≠ nulo) hacer
     pos = pos.sig
fin mientras
```



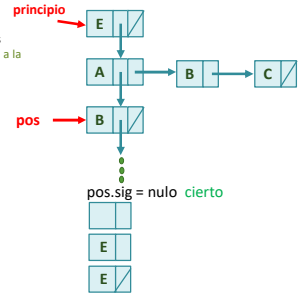
pos.sig ≠ E cierto
pos.sig ≠ nulo falso
/* no entra al mientras*/

148

Ejemplo del algoritmo de inserción

PASO 12

si (pos.sig = nulo) entonces
/* crea la arista y la agrega a la lista de aristas */
nueva(arista)
arista.info = ítemB
arista.sig = nulo

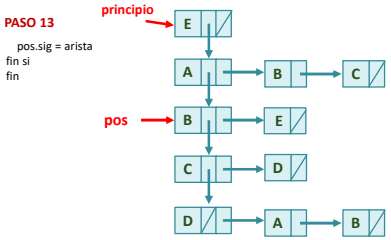


149

Ejemplo del algoritmo de inserción

PASO 13

pos.sig = arista
fin si
fin



150

Algoritmo de eliminación de una arista

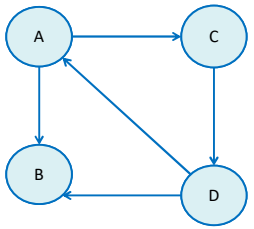
Algoritmo de eliminación de una arista en el grafo

```
inicio
/* ítemA contiene el vértice origen, ítemB contiene el vértice destino */
leer(itemA, itemB)
/* búsqueda del vértice origen */
pos = nulo
ptr = principio
mientras (ptr ≠ nulo) hacer
        si(ptr.info = itemA) entonces
                pos = ptr
                ptr = nulo
        sino
                ptr = ptr.sig
        fin si
fin mientras
/* búsqueda de la arista */
pos1 = pos.ady
ptr1 = pos
mov = 0
mientras (pos1.dest ≠ itemB y pos1.enl ≠ nulo) hacer
    ptr1 = pos1
    pos1 = pos1.enl
    mov = 1
fin mientras

si (pos1.dest = itemB) entonces
   si (mov = 0) entonces
      ptr1.ady = pos1.enl
   sino
      ptr1.enl = pos1.enl
   fin si
fin si
liberar (pos1)
fin
```

151

Ejemplo del algoritmo de eliminación de una arista

Eliminar la arista (A, B) el siguiente grafo utilizando el algoritmo de eliminación de una arista. La lista de nodos inicia en el grafo con principio = A.

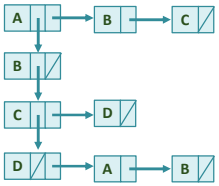


152

Ejemplo del algoritmo de eliminación de una arista

PASO 1

Hacer la representación enlazada del grafo



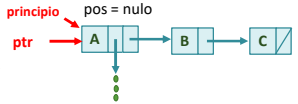
153

Ejemplo del algoritmo de eliminación de una arista

PASO 2

```
inicio
/* ítemA contiene el vértice origen, ítemB contiene el vértice destino */
leer(ítemA, ítemB)
/* búsqueda del vértice origen */
pos = nulo
ptr = principio
```

itemA = A
itemB = B
pos = nulo

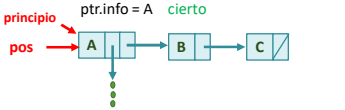


154

Ejemplo del algoritmo de eliminación de una arista

PASO 3

```
mientras (ptr ≠ nulo) hacer
  si(ptr.info = ítemA) entonces
     pos = ptr
     ptr = nulo
  sino
     ptr = ptr.sig
  fin si
fin mientras
```



ptr ≠ nulo cierto /* entra al mientras*/
ptr.info = A cierto

ptr = nulo

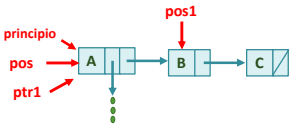
ptr ≠ nulo falso /* sale del mientras*/

155

Ejemplo del algoritmo de eliminación de una arista

PASO 4

```
/* búsqueda de la arista */
pos1 = pos.ady
ptr1 = pos
mov = 0
mientras (pos1.dest ≠ ítemB y
pos1.enl ≠ nulo) hacer
    ptr1 = pos1
    pos1 = pos1.enl
    mov = 1
fin mientras
```



mov = 0
pos1.dest ≠ B falso
pos.enl ≠ nulo cierto
/* no entra al mientras*/

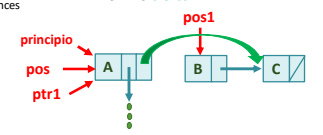
156

Ejemplo del algoritmo de eliminación de una arista

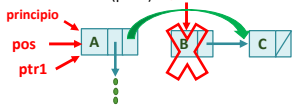
PASO 5

```
si (pos1.dest = ítemB) entonces
   si (mov = 0) entonces
      ptr1.ady = pos1.enl
   sino
      ptr1.enl = pos1.enl
   fin si
fin si
liberar (pos1)
fin
```

pos1.dest = B cierto
mov = 0 cierto



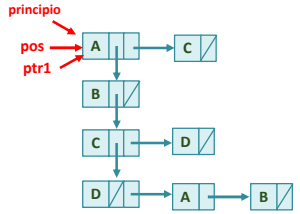
liberar (pos1) pos1



157

Ejemplo del algoritmo de eliminación de una arista

PASO 6



158

Algoritmo de eliminación de una nodo

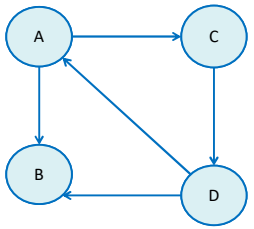
Algoritmo de eliminación de un nodo del grafo

```
inicio
/* lectura del nodo a eliminar */
Leer (ítem)
inicio
/* lectura del nodo a eliminar */
Leer (ítem)
si (principio = nulo) entonces
   indic = falso
sino
   si (principio.info = ítem) entonces
      ptr = principio
      principio = principio.sig
      ptr.sig = nulo
      indic = verdadero
   sino
      ptr = principio.sig
      salva = principio
      indic = falso
   fin si
fin si
mientras (ptr ≠ nulo y indic = falso) hacer
   si (ptr.info = ítem) entonces
      salva.sig = ptr.sig
      ptr.sig = nulo
      indic = verdadero
   sino
      salva = ptr
      ptr = ptr.sig
   fin si
fin mientras
si (indic = falso) entonces
   imprimir ("No se ha encontrado el valor a eliminar en
   el grafo")
sino
   liberar (ptr)
fin si
fin
```

159

Ejemplo del algoritmo de eliminación de un nodo

Eliminar el nodo B en el siguiente grafo utilizando el algoritmo de eliminación de un nodo. La lista de nodos inicia en el grafo con principio = A.

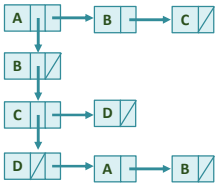


160

Ejemplo del algoritmo de eliminación de un nodo

PASO 1

Hacer la representación enlazada del grafo



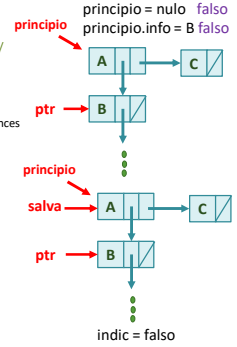
161

Ejemplo del algoritmo de eliminación de un nodo

PASO 2

```
inicio
/* lectura del nodo a eliminar */
Leer (ítem)
si (principio = nulo) entonces
   indic = falso
sino
   si (principio.info = ítem) entonces
        ptr = principio
        principio = principio.sig
        ptr.sig = nulo
        indic = verdadero
   sino
        ptr = principio.sig
        salva = principio
        indic = falso
   fin si
fin si
```

itemA = B
principio = nulo falso
principio.info = B falso



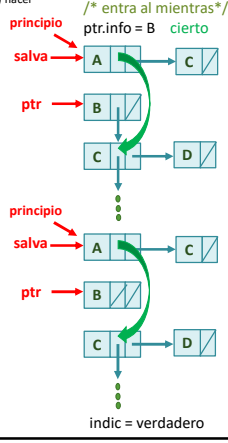
162

Ejemplo del algoritmo de eliminación de un nodo

PASO 3

```
mientras (ptr ≠ nulo y indic = falso) hacer
  si (ptr.info = ítem) entonces
       salva.sig = ptr.sig
       ptr.sig = nulo
       indic = verdadero
  sino
       salva = ptr
       ptr = ptr.sig
  fin si
fin mientras
```

ptr ≠ nulo cierto
indic = falso cierto
/* entra al mientras*/
ptr.info = B cierto



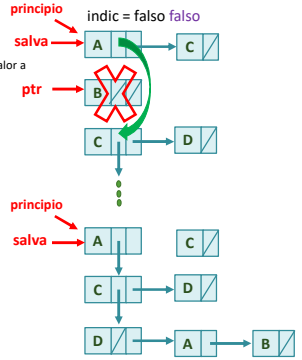
163

Ejemplo del algoritmo de eliminación de un nodo

PASO 4

```
si (indic = falso) entonces
   imprimir ("No se ha encontrado el valor a
   eliminar en el grafo")
sino
   liberar (ptr)
fin si
fin
```

indic = falso falso



164

Recorridos en un grafo

Recorrido en anchura

- En este recorrido se empieza en un nodo v: primero se visita v, luego se visitan todos sus adyacentes. Luego los adyacentes de estos y así sucesivamente. El algoritmo utiliza una cola de vértices.

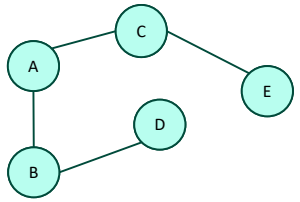
Recorrido en profundidad

- Consiste en alejarse todo lo posible del nodo origen para después empezar a visitar los nodos restantes a la vuelta. En este tipo de recorrido hay que usar una pila para ir almacenando los nodos que nos falta visitar.

165

Ejemplo del recorrido en anchura

Realizar el recorrido en anchura en el siguiente grafo. La lista de nodos inicia en el grafo con Actual = A.



166

Ejemplo del recorrido en anchura

PASO 1

Hacer un arreglo de visitados del grafo
Marcar cada nodo como no_visitado

NODOS	A	B	C	D	E
VISITADOS	F	F	F	F	F

Crear la Cola

COLA (Final, Frente)

PASO 2

Actual = A
Marcar actual como visitado

NODOS	A	B	C	D	E
VISITADOS	C	F	F	F	F

PASO 3

Marcar todos los vecinos de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	F

COLA: B C (Final, Frente)

RECORRIDO:
A

167

Ejemplo del recorrido en anchura

PASO 4

Sacar de la Cola y asignar a Actual

COLA: C (Frente, Final)

Actual = B

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	F

COLA: C D (Frente, Final)

RECORRIDO:
A B

PASO 5

Sacar de la Cola y asignar a Actual

COLA: D (Frente, Final)

Actual = C

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C

COLA: D E (Frente, Final)

RECORRIDO:
A B C

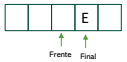
168

Ejemplo del recorrido en anchura

PASO 6

Sacar de la Cola y asignar a Actual

COLA

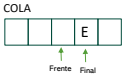


Actual = D

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C

COLA

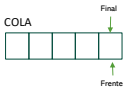


RECORRIDO:
A B C D

PASO 7

Sacar de la Cola y asignar a Actual

COLA

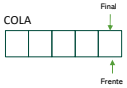


Actual = E

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Cola

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C

COLA

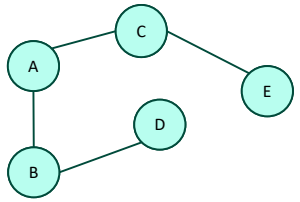


RECORRIDO:
A B C D E

169

Ejemplo del recorrido en profundidad

Realizar el recorrido en profundidad en el siguiente grafo.
La lista de nodos inicia en el grafo con Actual = A.



170

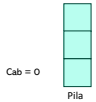
Ejemplo del recorrido en profundidad

PASO 1

Hacer un arreglo de visitados del grafo
Marcar cada nodo como no_visitado

NODOS	A	B	C	D	E
VISITADOS	F	F	F	F	F

Crear la Pila



PASO 2

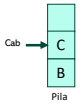
Actual = A
Marcar actual como visitado

NODOS	A	B	C	D	E
VISITADOS	C	F	F	F	F

PASO 3

Marcar todos los vecinos de actual como visitados y los metemos en la Pila

NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	F



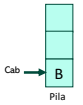
RECORRIDO:
A

171

Ejemplo del recorrido en profundidad

PASO 4

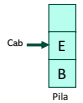
Sacar de la Pila y asignar a Actual



Actual = C

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

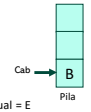
NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	C



RECORRIDO:
A C

PASO 5

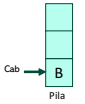
Sacar de la Pila y asignar a Actual



Actual = E

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

NODOS	A	B	C	D	E
VISITADOS	C	C	C	F	C



RECORRIDO:
A C E

172

Ejemplo del recorrido en profundidad

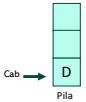
PASO 6

Sacar de la Pila y asignar a Actual

Actual = B

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

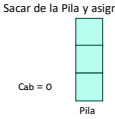
NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C



RECORRIDO:
A C E B

PASO 7

Sacar de la Pila y asignar a Actual



Actual = D

Marcar todos los vecinos no marcados de actual como visitados y los metemos en la Pila

NODOS	A	B	C	D	E
VISITADOS	C	C	C	C	C

RECORRIDO:
A C E B D

173

Capítulo II

174

Algoritmos de ordenamiento y búsqueda

Para resolver un problema pueden existir varios algoritmos. Por tanto, es lógico elegir el "mejor". Si el problema es sencillo o no hay que resolver muchos casos, se podría elegir el más "fácil". Si el problema es complejo o existen muchos casos habría que elegir el algoritmo que menos recursos utilice. Los recursos más importantes son el tiempo de ejecución y el espacio de almacenamiento. Generalmente, el más importante es el tiempo.

Al hablar de la eficiencia de un algoritmo nos referiremos a lo "rápido" que se ejecuta. La eficiencia de un algoritmo dependerá, en general, del "tamaño" de los datos de entrada. La eficiencia no tiene unidades de medida y se usa la notación O(n) para describirla.

175

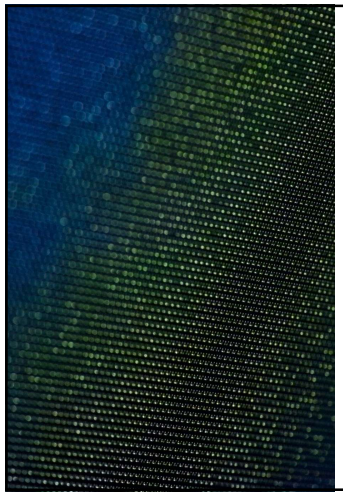
A la hora de analizar un algoritmo es necesario saber que pueden darse tres tipos de casos:

Caso mejor
- Se trata de aquellos ejemplares del problema en los que el algoritmo es más eficiente.
- Generalmente este caso no nos interesa.

Caso peor
- Se trata de aquellos ejemplares del problema en los que el algoritmo es menos eficiente (no siempre existe el caso peor).
- Este caso nos interesa mucho.

Caso medio
- Se trata del resto de ejemplares del problema.
- Es el caso que más nos debería preocupar puesto que será el más habitual, sin embargo, no siempre se puede calcular.

176



Consideraciones de eficiencia

La eficiencia de un algoritmo se mide en función de ejecución del mismo, lo cual depende, del tiempo que le tome a la computadora ejecutar las líneas de código del algoritmo. Esto depende de la velocidad de la computadora, el lenguaje de programación y el compilador entre otros factores.

Medimos el tiempo de ejecución de un algoritmo como una función del tamaño de su entrada, usualmente la cantidad de datos de entrada se especifica con la letra n. A la velocidad del crecimiento de la función con el tamaño de la entrada se le llama tasa de crecimiento del tiempo de ejecución.

El estudio del cambio en el rendimiento del algoritmo con el cambio en el orden del tamaño de entrada se define como análisis asintótico

177

Notaciones asintóticas para medir la eficiencia de un algoritmo

Notación de la Gran O (Big O)

- La notación asintótica se describe por medio de una función cuyo dominio es el conjunto de números naturales, N. Ésta se describe mediante la notación O.

Notación Theta (θ)

- Encierra la función desde arriba y desde abajo. Dado que representa el límite superior e inferior del tiempo de ejecución de un algoritmo, se utiliza para analizar la complejidad de caso promedio de un algoritmo.

Notación Omega (Ω)

- Representa el límite inferior del tiempo de ejecución de un algoritmo. Por lo tanto, proporciona la mejor complejidad de caso de un algoritmo.

178

Algoritmos de ordenamiento

El ordenamiento o clasificación consiste en organizar los datos en orden ascendente o descendente. Se le denomina clave al elemento en base al cual se está ordenado un conjunto de datos. Este procedimiento organiza los datos en una secuencia que facilita la búsqueda.

Dos criterios se utilizan para evaluar la eficiencia de un algoritmo de ordenamiento:

- Tiempo necesario para ordenar los datos proporcionados.
- Espacio de memoria requerido para hacerlo.

179

Algoritmos de ordenamiento

- Selección
- Inserción
- Burbuja
- Quicksort
- Heapsort

180

Ordenamiento por selección

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo. Este algoritmo tiene una complejidad de $O(n^2)$.

181

Algoritmo de ordenamiento por selección

```
Algoritmo OrdSelección
{
// n = número de elementos del arreglo
entero i, j, mindice, Aux;
i = 1;
Mientras (i<n) hacer
{  mindice = i;
  j = i+1;
  // Encontrar el índice del mínimo elemento
desordenado
  Mientras (j <= n) hacer
  {  si (Datos [ j ] < Datos [ mindice ]) entonces
       mindice = j;
     fin-si
     j = j + 1;
    }  // Fin-mientras
// Intercambiar el primer elemento desordenado con
el mínimo elemento desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
          i = i + 1;
}  // Fin-mientras
}
```

182

Ejemplo de ordenamiento por selección

Ordene mediante el algoritmo de selección el siguiente arreglo.

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

PASO 1

```
// n = número de elementos del
arreglo
entero i, j, mindice, Aux;
i = 1;
Mientras (i<n) hacer
{  mindice = i;
  j = i+1;
  // Encontrar el índice del
mínimo elemento desordenado
```

n = 5
i = 1
1 < 5 cierto
mindice = 1;
j = i+1 = 1 + 1 = 2

PASO 2

```
Mientras (j <= n) hacer
  {  si (Datos [j] < Datos [mindice]) entonces
       mindice = j;
     fin-si
     j = j + 1;
    }  // Fin-mientras
```

2 <= 5 cierto
5 < 9 cierto
mindice = 2

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 2+1 = 3

183

Ejemplo de ordenamiento por selección

PASO 3

```
Mientras (j <= n) hacer
  {  si (Datos [j] < Datos [mindice]) entonces
       mindice = j;
     fin-si
     j = j + 1;
    }  // Fin-mientras
```

3 <= 5 cierto
4 < 9 cierto
mindice = 3

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 3+1 = 4

PASO 4

```
Mientras (j <= n) hacer
  {  si (Datos [j] < Datos [mindice]) entonces
       mindice = j;
     fin-si
     j = j + 1;
    }  // Fin-mientras
```

4 <=5 cierto
12 < 9 falso

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 4+1 = 5

184

Ejemplo de ordenamiento por selección

PASO 5

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

5 <= 5 cierto
2 < 9 cierto
mindice = 5

j = 5 +1 = 6
6 <= 5 falso

PASO 6

```
// Intercambiar el primer elemento
desordenado con el mínimo elemento
desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 9
A[1] = 2
A[5] = 9

i = 1 +1 = 2

185

Ejemplo de ordenamiento por selección

PASO 7

```
Mientras (i<n) hacer
{ mindice = i;
  j= i+1;
  //Encontrar el índice del mínimo elemento
desordenado
  Mientras (j <= n) hacer
    { si (Datos [j] < Datos [mindice])
    entonces
      mindice = j;
    fin-si
    j = j + 1;
    } // Fin-mientras
```

2 < 5 cierto
mindice = 2;
j = i+1 = 2 + 1 = 3
3 <= 5 cierto
4 < 5 cierto
mindice = 3

j = 3 +1 = 4

PASO 8

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

4 <= 5 cierto
12 < 4 falso

j = 4 +1 = 5

186

Ejemplo de ordenamiento por selección

PASO 9

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

5 <= 5 cierto
9 < 4 falso

j = 5 +1 = 6
6 <= 5 falso

PASO 10

```
// Intercambiar el primer elemento
desordenado con el mínimo elemento
desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 5
A[2] = 4
A[3] = 5

i = 2 +1 = 3

187

Ejemplo de ordenamiento por selección

PASO 11

```
Mientras (i<n) hacer
{ mindice = i;
  j= i+1;
  //Encontrar el índice del mínimo elemento
desordenado
  Mientras (j <= n) hacer
    { si (Datos [j] < Datos [mindice])
    entonces
      mindice = j;
    fin-si
    j = j + 1;
    } // Fin-mientras
```

3 < 5 cierto
mindice = 3;
j = i+1 = 3 + 1 = 4
4 <= 5 cierto
12 < 5 falso

j = 4 +1 = 5

PASO 12

```
Mientras (j <= n) hacer
  { si (Datos [j] < Datos [mindice]) entonces
      mindice = j;
    fin-si
    j = j + 1;
  } // Fin-mientras
```

5 <= 5 cierto
9 < 5 falso

j = 5 +1 = 6
6 <= 5 falso

188

Ejemplo de ordenamiento por selección

PASO 13

```
// Intercambiar el primer elemento
desordenado con el mínimo elemento
desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 5
A[3] = 5
A[3] = 5

2	4	5	12	9
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

i = 3 +1 = 4

PASO 14

```
Mientras (i<n) hacer
{ mindice = i;
 j = i+1;
 // Encontrar el índice del mínimo elemento
desordenado
 Mientras (j <= n) hacer
 { si (Datos [j] < Datos [mindice])
 entonces
  mindice = j;
 fin-si
 j = j + 1;
 } // Fin-mientras
```

4 < 5 cierto
mindice = 4;
j = i+1 = 4 + 1 = 5
4 <= 5 cierto
9 < 12 cierto
mindice = 5

2	4	5	12	9
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

j = 5 +1 = 5
6 <= 5 falso

189

Ejemplo de ordenamiento por selección

PASO 15

```
// Intercambiar el primer elemento
desordenado con el mínimo elemento
desordenado
Aux = Datos [i];
Datos [i] = Datos [mindice];
Datos [mindice] = Aux;
i = i + 1;
} // Fin-mientras
}
```

Aux = 12
A[4] = 9
A[5] = 12

2	4	5	9	12
A[1]	A[2]	A[3]	A[4]	A[5]

mindice

i = 4 +1 = 5
5 < 5 falso

190

En este tipo de algoritmo los elementos que van a ser ordenados son considerados uno a la vez. Cada elemento es insertado en la posición apropiada con respecto al resto de los elementos ya ordenados.

Este procedimiento recibe el arreglo de datos a ordenar a[] y altera las posiciones de sus elementos hasta dejarlos ordenados de menor a mayor. N representa el número de elementos que contiene a[]. Los arreglos comienzan en la posición 0. Este algoritmo tiene una complejidad de $O(n^2)$.

Ordenamiento por inserción

191

Algoritmo Inserción

```
Inicio
entero i, j, n, A[];
Desde (i=1; i < n; i = i + 1)
   j= i
   Aux = A[i]
   Mientras (j > 0 y Aux  < A[j-1] )
      // Intercambiar el elemento
      A[ j ] = A[ j-1 ]
      j=j-1
   Fin Mientras
   A[ j ] = Aux;
Fin Desde
Fin
```

Algoritmo de ordenamiento por inserción

192

Ejemplo de ordenamiento por inserción

Ordene mediante el algoritmo de inserción el siguiente arreglo.

7	13	6	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 1

```
Desde (i=1; i < n; i = i + 1)
  j= i
  Aux = A[i]
  Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

n = 5
i = 1
1 < 5 cierto
j = 1
Aux = 13
1 > 0 y 13 < 7 falso
A[1] = 13

PASO 2

```
Desde (i=1; i < n; i = i + 1)
  j= i
  Aux = A[i]
  Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

i = 2
2 < 5 cierto
j = 2
Aux = 6
2 > 0 y 6 < 13 cierto
A[2] = 13

7	13	13	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

j = 1

193

Ejemplo de ordenamiento por inserción

PASO 3

```
Mientras (j > 0 y Aux  < A[j-1] )
  // Intercambiar el elemento
  A[ j ] = A[ j-1 ]
  j=j-1
 Fin Mientras
 A[ j ] = Aux;
Fin Desde
```

1 > 0 y 6 < 7 cierto
A[1] = 7

7	7	13	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

j = 0
0 > 0 falso
A[0] = 6

6	7	13	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 4

```
Desde (i=1; i < n; i = i + 1)
  j= i
  Aux = A[i]
  Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

i = 3
3 < 5 cierto
j = 3
Aux = 10
3 > 0 y 10 < 13 cierto
A[3] = 13

6	7	13	13	8
A[0]	A[1]	A[2]	A[3]	A[4]

j = 2

PASO 5

```
Mientras (j > 0 y Aux  < A[j-1] )
  // Intercambiar el elemento
  A[ j ] = A[ j-1 ]
  j=j-1
 Fin Mientras
 A[ j ] = Aux;
Fin Desde
```

2 > 0 y 10 < 7 falso
A[2] = 10

6	7	10	13	8
A[0]	A[1]	A[2]	A[3]	A[4]

194

Ejemplo de ordenamiento por inserción

PASO 6

```
Desde (i=1; i < n; i = i + 1)
  j= i
  Aux = A[i]
  Mientras (j > 0 y Aux  < A[j-1] )
    // Intercambiar el elemento
    A[ j ] = A[ j-1 ]
    j=j-1
  Fin Mientras
  A[ j ] = Aux;
Fin Desde
```

i = 4
4 < 5 cierto
j = 4
Aux = 8
4 > 0 y 8 < 13 cierto
A[4] = 13

6	7	10	13	13
A[0]	A[1]	A[2]	A[3]	A[4]

j = 3

PASO 7

```
Mientras (j > 0 y Aux  < A[j-1] )
  // Intercambiar el elemento
  A[ j ] = A[ j-1 ]
  j=j-1
 Fin Mientras
 A[ j ] = Aux;
Fin Desde
```

3 > 0 y 8 < 10 cierto
A[3] = 10

7	7	10	10	13
A[0]	A[1]	A[2]	A[3]	A[4]

j = 2
2 > 0 y 8 < 7 falso
A[2] = 8

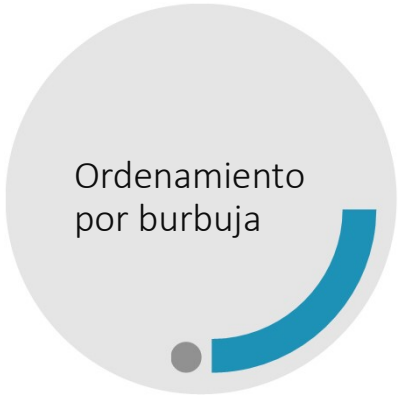
6	7	8	10	13
A[0]	A[1]	A[2]	A[3]	A[4]

i = 5
5 < 5 falso

195

Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados tomando el elemento mayor y recorriendo de posición en posición hasta ponerlo en su lugar. Esto se hace hasta que el arreglo este ordenado. Esta operación se hace n − 1 pasadas y n − 1 comparaciones en cada pasada, por lo tanto, el número de comparaciones es (n − 1) * (n − 1) = n2 − 2n + 1, es decir, la complejidad es $0(n^2)$.

Ordenamiento por burbuja



196

Algoritmo Burbuja

```
Inicio
i = 1;
cambiado=cierto
Mientras ((i < n) y (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
  // sube la burbuja del valor más pequeño no ordenado
  Mientras (j > i)  hacer
    // si el valor es menor que su predecesor, intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
        cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j - 1
  Fin-mientas
  i = i +1
 Fin-mientras
Fin
```

Algoritmo de ordenamiento por burbuja

197

Ejemplo de ordenamiento por burbuja

Ordene mediante el algoritmo de burbuja el siguiente arreglo.

19	5	16	12	8
A[1]	A[2]	A[3]	A[4]	A[5]

PASO 1

```
i = 1;
cambiado=cierto
Mientras ((i < n) y
  (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

n = 5
i = 1
cambiado = cierto
1 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

PASO 2

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
```

5 > 1 cierto
8 < 12 cierto
cambiado = cierto
Aux = 8
A[5] = 12
A[4] = 8

19	5	16	8	12
A[1]	A[2]	A[3]	A[4]	A[5]

j = 4

198

Ejemplo de ordenamiento por burbuja

PASO 3

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
```

4 > 1 cierto
8 < 16 cierto
cambiado = cierto
Aux = 8
A[4] = 16
A[3] = 8

19	5	8	16	12
A[1]	A[2]	A[3]	A[4]	A[5]

j = 3

PASO 4

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
```

3 > 1 cierto
8 < 5 falso

j = 2

199

Ejemplo de ordenamiento por burbuja

PASO 5

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
     Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
i = i +1
Fin-mientras
```

2 > 1 cierto
5 < 19 cierto
cambiado = cierto
Aux = 5
A[2] = 19
A[1] = 5

5	19	8	16	12
A[1]	A[2]	A[3]	A[4]	A[5]

j = 1
1 > 1 falso
i = 2

PASO 6

```
Mientras ((i < n) y
  (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

2 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

200

Ejemplo de ordenamiento por burbuja

PASO 7

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
```

5 > 2 cierto
12 < 16 cierto
cambiado = cierto
Aux = 12
A[5] = 16
A[4] = 12

5	19	8	12	16
A[1]	A[2]	A[3]	A[4]	A[5]

j = 4

PASO 8

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
```

4 > 2 cierto
12 < 8 falso
j = 3

201

Ejemplo de ordenamiento por burbuja

PASO 9

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
i = i +1
 Fin-mientras
```

3 > 2 cierto
8 < 19 cierto
cambiado = cierto
Aux = 8
A[3] = 19
A[2] = 8

5	8	19	12	16
A[1]	A[2]	A[3]	A[4]	A[5]

j = 2
2 > 2 falso
i = 3

PASO 10

```
Mientras ((i < n)  y
  (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

3 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

202

Ejemplo de ordenamiento por burbuja

PASO 11

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
i = i +1
 Fin-mientras
```

5 > 3 cierto
16 < 12 falso
j = 4
4 > 3 cierto
12 < 19 cierto
cambiado = cierto
Aux = 12
A[4] = 19
A[3] = 12

5	8	12	19	16
A[1]	A[2]	A[3]	A[4]	A[5]

j = 3
3 > 3 falso
i = 4

PASO 12

```
Mientras ((i < n)  y
  (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

4 < 5 y cambiado = cierto cierto
j = 5
cambiado = falso

203

Ejemplo de ordenamiento por burbuja

PASO 13

```
Mientras (j > i)  hacer
  // si el valor es menor que su predecesor,
     intercambiarlos.
    Si (Datos [ j ] < Datos [ j –1 ]) entonces
          cambiado = cierto
          aux = Datos [ j ]
          Datos [ j ] = Datos [ j-1]
          Datos [ j-1] = aux
      Fin-si
    j = j – 1
Fin-mientas
i = i +1
 Fin-mientras
```

5 > 4 cierto
16 < 19 cierto
cambiado = cierto
Aux = 16
A[5] = 19
A[4] = 16

5	8	12	16	19
A[1]	A[2]	A[3]	A[4]	A[5]

j = 4
4 > 4 falso
i = 5

PASO 14

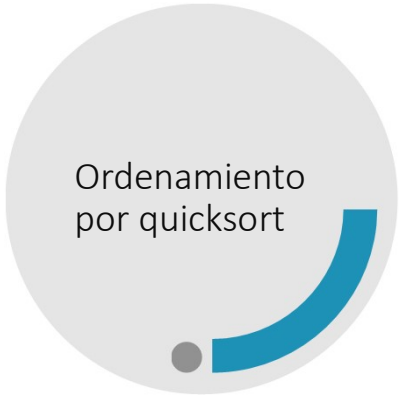
```
Mientras ((i < n)  y
  (cambiado = "cierto"))  hacer
  j = n;
  cambiado = falso
// sube la burbuja del valor más
pequeño no ordenado
```

5 < 5 y cambiado = cierto falso

204

Aplica la técnica divide y vencerás, al dividir el arreglo a ordenar en dos particiones separadas por un elemento central, denominado pivote o elemento de partición. Una característica de esta partición es que todos los elementos de la partición izquierda son menores que todos los elementos de la partición derecha, luego se ordenan las dos particiones independientemente. El algoritmo tiene una complejidad de O(n log2 n).

Ordenamiento por quicksort



205

```
Algoritmo quicksort(A, inf, sup)
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
  mientras A[j] > x hacer
    j = j - 1
  fin_mientras
  si i ≤ j entonces
    aux = A[i]
    A[i] = A[j]
    A[j] = aux
    i = i + 1
    j = j - 1
  fin_si
fin_mientras
si inf < j
  quicksort(A, inf, j)
fin_si
si i < sup
  quicksort(A, i, sup)
fin_si
Fin
```

Algoritmo de ordenamiento por quicksort

206

Ejemplo de ordenamiento por quicksort

Ordene mediante el algoritmo quicksort el siguiente arreglo.

7	10	13	12	8
A[1]	A[2]	A[3]	A[4]	A[5]

PASO 1

```
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
```

inf = 1
sup = 5
i = 1
j = 5
x = A[(1+5)/2] = A[3] x = 13
1 ≤ 5 cierto mientras
7 < 13 cierto mientras
i = 2
10 < 13 cierto mientras
i = 3
13 < 13 falso mientras

PASO 2

```
mientras A[j] > x hacer
  j = j - 1
fin_mientras
si i ≤ j entonces
  aux = A[i]
  A[i] = A[j]
  A[j] = aux
  i = i + 1
  j = j - 1
fin_si
fin_mientras
```

8 > 13 falso mientras
1 ≤ 5 cierto
aux = 13
A[3] = 8
A[5] = 13

7	10	8	12	13
A[1]	A[2]	A[3]	A[4]	A[5]

i = 4
j = 4

207

Ejemplo de ordenamiento por quicksort

PASO 3

```
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
  mientras A[j] > x hacer
    j = j - 1
  fin_mientras
  si i ≤ j entonces
    aux = A[i]
    A[i] = A[j]
    A[j] = aux
    i = i + 1
    j = j - 1
  fin_si
fin_mientras
```

4 ≤ 4 cierto mientras
12 < 13 cierto mientras
i = 5
13 < 13 falso mientras
12 > 13 falso mientras
5 ≤ 4 falso

PASO 4

```
si inf < j
  quicksort(A, inf, j)
fin_si
si i < sup
  quicksort(A, i, sup)
fin_si
Fin
```

1 < 4 cierto
quicksort(A, 1, 4)

PASO 5

```
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
```

i = 1
j = 4
x = A[(1+4)/2] = A[2]
x = 10
1 ≤ 4 cierto mientras
7 < 10 cierto mientras
i = 2
10 < 10 falso mientras

PASO 6

```
mientras A[j] > x hacer
  j = j - 1
fin_mientras
si i ≤ j entonces
  aux = A[i]
  A[i] = A[j]
  A[j] = aux
  i = i + 1
  j = j - 1
fin_si
fin_mientras
```

12 > 10 cierto mientras
j = 3
8 > 10 falso mientras
2 ≤ 3 cierto
aux = 10
A[2] = 8
A[3] = 10

7	8	10	12	13
A[1]	A[2]	A[3]	A[4]	A[5]

i = 3
j = 2
3 ≤ 2 falso mientras

208

Ejemplo de ordenamiento por quicksort

PASO 7

```
si inf < j
    quicksort(A, inf, j)
fin_si
si i < sup
    quicksort(A, i, sup)
fin_si
Fin
```

1 < 2 cierto
quicksort(A, 1, 2)

PASO 8

```
i = inf
j = sup
x = A[(inf+sup)div 2]
mientras i ≤ j hacer
  mientras A[i] < x hacer
    i = i + 1
  fin_mientras
```

i = 1
j = 2
x = A[(1+2)/2] = A[1]
x = 7
1 ≤ 2 cierto mientras
7 < 7 falso mientras

PASO 9

```
mientras A[j] > x hacer
  j = j - 1
fin_mientras
si i ≤ j entonces
  aux = A[i]
  A[i] = A[j]
  A[j] = aux
  i = i + 1
  j = j - 1
fin_si
fin_mientras
```

8 > 7 cierto mientras
j = 1
7 > 7 falso mientras
1 ≤ 1 cierto
aux = 7
A[1] = 7
A[1] = 7

7	8	10	12	13
A[1]	A[2]	A[3]	A[4]	A[5]

i = 2
j = 0
2 ≤ 0 falso mientras

PASO 10

```
si inf < j
    quicksort(A, inf, j)
fin_si
si i < sup
    quicksort(A, i, sup)
fin_si
Fin
```

1 < 0 falso
1 < 0 falso

El algoritmo tiene un tiempo de ejecución de O(n log n), el mismo consiste en almacenar todos los elementos en un montículo y luego extraer el nodo que queda como raíz en iteraciones sucesivas obteniendo el conjunto ordenado.

Ordenamiento por heapsort


```
HeapSort
n = cantidad de elementos del arreglo
desde (i = n / 2 - 1; i >= 0; i= i - 1)
      heapify(arr, n, i)
fin desde
desde (k = n - 1; k >= 0; k = k - 1)
      int tmp = arr[0]
      arr[0] = arr[k]
      arr[k] = tmp
      heapify(arr, k, 0)
fin desde
fin HeapSort

heapify(arr[], m, j)
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y arr[leftChild] > arr[max]) entonces
      max = leftChild
fin si
si (rightChild < m y arr[rightChild] > arr[max]) entonces
      max = rightChild
fin si
si (max ≠ j) entonces
      swap = arr[j]
      arr[j] = arr[max]
      arr[max] = swap
      heapify(arr, m, max)
fin si
fin heapify
```

Algoritmo de ordenamiento por heapsort

Ejemplo de ordenamiento por heapsort

Ordene mediante el algoritmo heapsort el siguiente arreglo.

23	10	16	11	20
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 1

```
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
```

n = 5
i = 5/2 - 1
i = 1
1 ≥ 0 cierto
heapify(A, 5, 1)

PASO 2

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 1
leftChild = 3
rightChild = 4
3 < 5 y 11 > 10 cierto
max = 3
4 < 5 y 20 > 11 cierto
max = 4
4 ≠ 1 cierto
swap = 10
A[1] = 20
A[4] = 10

23	20	16	11	10
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 5, 4)

PASO 3

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 4
leftChild = 9
rightChild = 10
9 < 5 falso
10 < 5 falso
4 ≠ 4 falso

Ejemplo de ordenamiento por heapsort

PASO 4

```
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
```

i = 0
0 ≥ 0 cierto
heapify(A, 5, 0)

PASO 5

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 5 y 20 > 23 falso
2 < 5 y 16 > 23 falso
0 ≠ 0 falso

PASO 6

```
desde (i = n / 2 - 1; i >= 0; i= i - 1)
    heapify(arr, n, i)
fin desde
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

i = -1
-1 ≥ 0 falso
k = 4
4 ≥ 0 cierto
tmp = 23
A[0] = 10
A[4] = 23

10	20	16	11	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 4, 0)

213

Ejemplo de ordenamiento por heapsort

PASO 7

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 4 y 20 > 10 cierto
max = 1
2 < 4 y 16 > 20 falso
1 ≠ 0 cierto
swap = 10
A[0] = 20
A[1] = 10

20	10	16	11	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 4, 1)

PASO 8

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 1
leftChild = 3
rightChild = 4
3 < 4 y 11 > 10 cierto
max = 1
4 < 4 falso
1 ≠ 1 falso

214

Ejemplo de ordenamiento por heapsort

PASO 9

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 3
3 ≥ 0 cierto
tmp = 20
A[0] = 11
A[3] = 20

11	10	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 3, 0)

PASO 10

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 3 y 10 > 11 falso
2 < 3 y 16 > 11 cierto
max = 2
2 ≠ 0 cierto
swap = 11
A[0] = 16
A[2] = 11

16	10	11	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 3, 2)

215

Ejemplo de ordenamiento por heapsort

PASO 11

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 2
leftChild = 5
rightChild = 6
5 < 3 falso
6 < 3 falso
2 ≠ 2 falso

PASO 12

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 2
2 ≥ 0 cierto
tmp = 16
A[0] = 11
A[2] = 16

11	10	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 2, 0)

216

Ejemplo de ordenamiento por heapsort

PASO 13

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 2 y 10 > 11 falso
2 < 2 falso
0 ≠ 0 falso

PASO 14

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 1
1 ≥ 0 cierto
tmp = 11
A[0] = 10
A[1] = 11

10	11	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 1, 0)

PASO 15

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 1 falso
2 < 1 falso
0 ≠ 0 falso

217

Ejemplo de ordenamiento por heapsort

PASO 16

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = 0
0 ≥ 0 cierto
tmp = 10
A[0] = 10
A[0] = 10

10	11	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

heapify(A, 0, 0)

PASO 17

```
max = j
leftChild = 2 * j + 1
rightChild = 2 * j + 2
si (leftChild < m y
arr[leftChild] > arr[max]) entonces
    max = leftChild
fin si
si (rightChild < m y
arr[rightChild] > arr[max]) entonces
    max = rightChild
fin si
si (max ≠ j) entonces
    swap = arr[j]
    arr[j] = arr[max]
    arr[max] = swap
    heapify(arr, m, max)
fin si
fin heapify
```

max = 0
leftChild = 1
rightChild = 2
1 < 0 falso
2 < 0 falso
0 ≠ 0 falso

PASO 18

```
desde (k = n - 1; k >= 0; k = k - 1)
    int tmp = arr[0]
    arr[0] = arr[k]
    arr[k] = tmp
    heapify(arr, k, 0)
fin desde
```

k = -1
-1 ≥ 0 falso

218

Algoritmos de búsqueda

El ordenamiento o clasificación consiste en organizar los datos en orden ascendente o descendente. Se le denomina clave al elemento en base al cual se está ordenado un conjunto de datos. Este procedimiento organiza los datos en una secuencia que facilita la búsqueda.

Dos criterios se utilizan para evaluar la eficiencia de un algoritmo de ordenamiento:

- Tiempo necesario para ordenar los datos proporcionados.
- Espacio de memoria requerido para hacerlo.

219

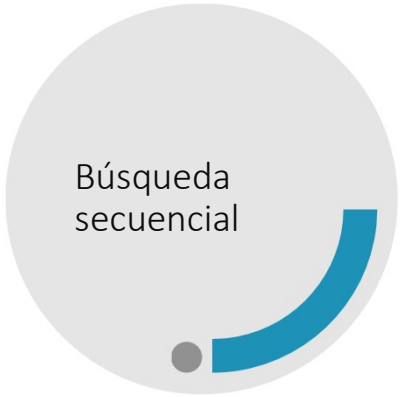
Algoritmos de búsqueda

- Secuencial
- Binaria
- En tabla
- Directa de cadena
- De cadena Knuth-Morris-Pratt
- De cadena Boyer-Moore

220

Es la técnica más simple para buscar un elemento en un arreglo. Consiste en recorrer el arreglo elemento a elemento e ir comparando con el valor buscado (clave). Se empieza con la primera casilla del arreglo y se observa una casilla tras otra hasta que se encuentra el elemento buscado o se han visto todas las casillas. El resultado de la búsqueda es un solo valor, y será la posición del elemento buscado o cero. Dado que el arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentre ya sea en el primer o en el último elemento. La eficiencia de la búsqueda secuencial es de O(n).

Búsqueda secuencial



221

Algoritmo de Búsqueda Secuencial

```
leer(dato)
enc= falso
pos = 0
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras)
Si (enc = cierto) entonces
    Imprimir ("El valor se ha encontrado")
Sino
    Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

Algoritmo de búsqueda secuencial

222

Ejemplo de búsqueda secuencial

Utilizando el algoritmo de búsqueda secuencial, busque el valor de 6 en el siguiente arreglo.

7	13	6	10	8
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 1

```
leer(dato)
enc= falso
pos = 0
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras
```

n = 6
dato =6
enc = falso
pos = 0
0 < 6 y enc = falso cierto
7 = 6 falso
pos = 1

PASO 2

```
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras
```

1 < 6 y enc = falso cierto
13 = 6 falso
pos = 2

223

Ejemplo de búsqueda secuencial

PASO 3

```
Mientras (pos < n y enc = falso)
Si (A[pos] = dato) entonces
  enc = cierto
sino
   pos = pos + 1
fin si
Fin Mientras
```

2 < 6 y enc = falso cierto
6 = 6 cierto
enc = cierto
2 < 6 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
    Imprimir ("El valor se ha
encontrado")
Sino
    Imprimir ("El valor no se ha
encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

224

Es el método más eficiente para encontrar elementos en un arreglo ordenado. El proceso comienza comparando el elemento central del arreglo con el valor buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor que el central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en el subarreglo superior, si el elemento buscado es menor que el contenido de la casilla central, se debe buscar en el subarreglo inferior. El orden de complejidad es $O(\log_2 n)$.

Búsqueda binaria

225

Algoritmo de Búsqueda Binaria

```
leer(dato)
inf = 0
sup = n-1
enc = falso
Mientras (inf <= sup y enc = falso) hacer
  centro = (sup + inf) / 2
   // División entera: se trunca la fracción
   si (a[centro] = dato) entonces
          enc = cierto
    sino
          si (dato < a[centro]) entonces
                sup = centro – 1
          sino
                inf = centro + 1
          fin si
    fin si
Fin Mientras
Si (enc = cierto) entonces
          Imprimir ("El valor se ha encontrado")
Sino
          Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

Algoritmo de búsqueda binaria

226

Ejemplo de búsqueda binaria

Utilizando el algoritmo de búsqueda binaria, busque el valor de 23 en el siguiente arreglo.

10	11	16	20	23
A[0]	A[1]	A[2]	A[3]	A[4]

PASO 1

```
leer(dato)
inf = 0
sup = n-1
enc = falso
Mientras (inf <= sup y enc = falso) hacer
  centro = (sup + inf) / 2
  // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
     enc = cierto
  sino
     si (dato < a[centro]) entonces
        sup = centro – 1
     sino
        inf = centro + 1
     fin si
  fin si
Fin Mientras
```

n = 5
dato = 23
inf = 0
sup = 4
enc = falso
0 ≤ 4 y enc = falso cierto
centro = (4 + 0)/2
centro = 2
16 = 23 falso
23 < 16 falso
inf = 3

227

Ejemplo de búsqueda binaria

PASO 2

```
Mientras (inf <= sup y enc = falso) hacer
  centro = (sup + inf) / 2
  // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
     enc = cierto
  sino
     si (dato < a[centro]) entonces
        sup = centro – 1
     sino
        inf = centro + 1
     fin si
  fin si
Fin Mientras
```

3 ≤ 4 y enc = falso cierto
centro = (4 + 3)/2
centro = 3
20 = 23 falso
23 < 20 falso
inf = 4

PASO 3

```
Mientras (inf <= sup y enc = falso) hacer
  centro = ((sup - inf) / 2) + inf
  // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
     enc = cierto
  sino
     si (dato < a[centro]) entonces
        sup = centro – 1
     sino
        inf = centro + 1
     fin si
  fin si
Fin Mientras
```

4 ≤ 4 y enc = falso cierto
centro = (4 + 4)/2
centro = 4
23 = 23 cierto
enc = cierto
4 ≤ 4 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
  Imprimir ("El valor se ha encontrado")
Sino
  Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

228

El algoritmo busca un dato (clave) K que está almacenado en una tabla T de índice único. La condición de búsqueda pueda ser determinada mediante una función booleana, enc que devuelve un valor cierto si y sólo si el dato satisface la condición. El algoritmo tiene una complejidad lineal de O(n).

Búsqueda lineal o en tabla

229

Algoritmo Búsqueda Lineal

```
enc = falso
i = 1
mientras (i <= n y enc = falso) hacer
  si (K = A[i]) entonces
    enc = cierto
  sino
    i = i + 1
fin si
```

PASO 3

```
Mientras (inf <= sup y enc = falso) hacer
  centro = ((sup - inf) / 2) + inf
  // División entera: se trunca la fracción
  si (a[centro] = dato) entonces
    enc = cierto
  sino
    si (dato < a[centro]) entonces
      sup = centro – 1
    sino
      inf = centro + 1
    fin si
  fin si
Fin Mientras
```

4 ≤ 4 y enc = falso cierto
centro = (4 + 4)/2
centro = 4
23 = 23 cierto
enc = cierto
4 ≤ 4 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
  Imprimir ("El valor se ha encontrado")
Sino
  Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

Algoritmo de búsqueda lineal o en tabla

230

Ejemplo de búsqueda lineal o en tabla

Utilizando el algoritmo de búsqueda lineal o en tabla, busque el valor de 4 en el siguiente arreglo.

9	5	4	12	2
A[1]	A[2]	A[3]	A[4]	A[5]

PASO 1

```
enc = falso
i = 1
mientras (i <= n y enc = falso) hacer
  si (K = A[i]) entonces
    enc = cierto
  sino
    i = i + 1
  fin si
fin mientras
```

n = 5
K = 4
enc = falso
i = 1
1 ≤ 5 y enc = falso cierto
4 = 9 falso
i = 2

PASO 2

```
mientras (i <= n y enc = falso) hacer
  si (K = A[i]) entonces
    enc = cierto
  sino
    i = i + 1
  fin si
fin mientras
```

2 ≤ 5 y enc = falso cierto
4 = 5 falso
i = 3

PASO 3

```
mientras (i <= n y enc = falso) hacer
  si (K = A[i]) entonces
    enc = cierto
  sino
    i = i + 1
  fin si
fin mientras
```

3 ≤ 5 y enc = falso cierto
4 = 4 cierto
enc = cierto
3 ≤ 5 y enc = falso falso

PASO 4

```
Si (enc = cierto) entonces
  Imprimir ("El valor se ha encontrado")
Sino
  Imprimir ("El valor no se ha encontrado")
Fin si
Fin
```

enc = cierto cierto
El valor se ha encontrado

231

El algoritmo localiza dentro de una cadena de longitud n (denominada texto, arreglo t en el algoritmo) una subcadena más pequeña de longitud m (denominada patrón, arreglo p en el algoritmo) o un carácter. El algoritmo compara carácter a carácter los arreglos texto y patrón comenzando por el extremo izquierdo de ambos, si coinciden se compara el siguiente carácter, si no coinciden el proceso se reinicia comenzado en la posición siguiente a la que se inició la búsqueda.

Búsqueda directa de cadena

232

Búsqueda cadena

```
// t[n] y p[m] son arreglos de caracteres
i = 0
j = 0
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j  = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
si (j = m) entonces
  Imprimir ("Se encontró la subcadena")
sino
  Imprimir ("No se encontró la subcadena")
fin si
fin
```

Algoritmo de búsqueda directa de cadena

233

Ejemplo de búsqueda directa de cadena

Utilizando el algoritmo de búsqueda de cadena, busque la subcadena del arreglo P en el arreglo T.

C	A	D	E	N	A
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]

A	D	E
P[0]	P[1]	P[2]

PASO 1

```
i = 0
j = 0
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

n = 6
m = 3
i = 0
j = 0
0 < 6 y 0 < 3 cierto
C = A falso
i = 1
j = 0

PASO 2

```
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

1 < 6 y 0 < 3 cierto
A = A cierto
i = 2
j = 1

234

Ejemplo de búsqueda directa de cadena

PASO 3

```
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

2 < 6 y 1 < 3 cierto
D = D cierto
i = 3
j = 2

PASO 4

```
mientras (i<n y j<m) hacer
  si (t[i] = p[j]) entonces
    i = i + 1
    j = j + 1
  sino
    i = i - j + 1
    j = 0
  fin si
fin mientras
```

3 < 6 y 2 < 3 cierto
E = E cierto
i = 4
j = 3
4 < 6 y 3 < 3 falso

PASO 5

```
si (j = m) entonces
  Imprimir ("Se encontró la
    subcadena")
sino
  Imprimir ("No se encontró la
    subcadena")
fin si
fin
```

3 = 3 cierto
Se encontró la subcadena

235

Busca las ocurrencias de un "patrón" dentro de un "texto" principal. El algoritmo usa la observación de cuando se produce una falta de coincidencia, la palabra en sí incluye información suficiente para determinar dónde podría comenzar la próxima coincidencia: con la cadena a localizar se precalcula una tabla de saltos (conocida como tabla de fallos) que después al examinar entre sí las cadenas, se utiliza para hacer saltos cuando se localiza un fallo. Con esto se evita el análisis más de una vez de los caracteres de la cadena donde se busca. El tiempo total de ejecución del algoritmo es O(n).

Búsqueda de cadena Knuth-Morris-Pratt (KMP)

236

Algoritmo de búsqueda KMP

```
Algoritmo KMP(Text, Patron)
// m = largo del texto, arreglo T del texto
// n = largo del patrón, arreglo P del patrón
GenerateSuffixArray(Patron)
i = 0
j = 0
mientras (i < m) hacer
   si (Patron[j] = Text[i]) entonces
      j = j + 1
      i = i + 1
   fin si
   si (j = n) entonces
      Imprimir ("Incidencia encontrada en: ",
      i - j)
   sino
      si (i < m) y (Patron[j] ≠ Text[i]) entonces
         si (j ≠ 0) entonces
            j = S[j-1]
         sino
            i = i + 1
         fin si
      fin si
   fin si
fin mientras
```

```
GenerateSuffixArray(Patron)
i = 1
j = 0
S[0] = 0
// n = largo del patrón, arreglo P del patrón
mientras (i < n) hacer
   si (Patron[i] = Patron[j]) entonces
      S[i] = j + 1
      j = j + 1
      i = i + 1
   sino
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         S[i] = 0
         i = i + 1
      fin si
   fin si
fin mientras
```

237

Ejemplo de búsqueda KMP

Utilizando el algoritmo KMP, busque la subcadena del arreglo P en el arreglo T.

	F	B	A	A	B	A	D	C
T[]	T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]

	A	A	B	A
P[]	P[0]	P[1]	P[2]	P[3]

PASO 1

GenerateSuffixArray(Patron)
i = 1
j = 0
S[0] = 0
// n = largo del patrón, arreglo P del patrón
mientras (i < n) hacer
  si (Patron[i] = Patron[j]) entonces
    S[i] = j + 1
    j = j + 1
    i = i + 1

n = 4
i = 1
j = 0
S[0] = 0

S[]	S[0]	S[1]	S[2]	S[3]
	0			

1 < 4 cierto
A = A cierto
S[1] = 1

S[]	S[0]	S[1]	S[2]	S[3]
	0	1		

j = 1
i = 2

PASO 2

```
mientras (i < n) hacer
   si (Patron[i] = Patron[j]) entonces
      S[i] = j + 1
      j = j + 1
      i = i + 1
   sino
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         S[i] = 0
         i = i + 1
      fin si
   fin si
fin mientras
```

2 < 4 cierto
B = A falso
1 ≠ 0 cierto
j = 0
0 < 4 cierto
B = A falso
0 ≠ 0 falso
S[2] = 0

S[]	S[0]	S[1]	S[2]	S[3]
	0	1	2	

i = 3

238

Ejemplo de búsqueda KMP

PASO 3

```
mientras (i < n) hacer
   si (Patron[i] = Patron[j]) entonces
      S[i] = j + 1
      j = j + 1
      i = i + 1
   sino
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         S[i] = 0
         i = i + 1
      fin si
   fin si
fin mientras
```

3 < 4 cierto
A = A cierto
S[3] = 1

S[]	S[0]	S[1]	S[2]	S[3]
	0	1	2	1

j = 1
i = 4
3 < 4 falso

PASO 4 – Continua en el algoritmo KMP

```
i = 0
j = 0
mientras (i < m) hacer
   si (Patron[j] = Text[i]) entonces
      j = j + 1
      i = i + 1
   fin si
```

m = 8
n = 4
i = 0
j = 0
0 < 8 cierto
A = F falso
0 = 4 falso

PASO 5

```
si (j = n) entonces
   Imprimir ("Incidencia encontrada
   en: ", i - j)
   j = S[j - 1]
sino
   si (i < m)  y (Patron[j] ≠ Text[i])
   entonces
      si (j ≠ 0) entonces
         j = S[j-1]
      sino
         i = i + 1
      fin si
   fin si
fin si
fin mientras
```

0 = 4 falso
0 < 8 y A ≠ F cierto
0 ≠ 0 falso
i = 1

239

Ejemplo de búsqueda KMP

PASO 6

```
mientras (i < m) hacer
   si (Patron[j] = Text[i]) entonces
      j := j + 1
      i := i + 1
   fin si
   si (j = n) entonces
      Imprimir ("Incidencia encontrada
      en: ", i - j)
      j = S[j - 1]
   sino
      si (i < m)  y (Patron[j] ≠ Text[i])
      entonces
         si (j ≠ 0) entonces
            j = S[j-1]
         sino
            i = i + 1
         fin si
      fin si
   fin si
fin mientras
```

1 < 8 cierto
A = B falso
0 = 4 falso
1 < 8 y A ≠ B cierto
0 ≠ 0 falso
i = 2

PASO 7

```
mientras (i < m) hacer
   si (Patron[j] = Text[i]) entonces
      j := j + 1
      i := i + 1
   fin si
   si (j = n) entonces
      Imprimir ("Incidencia encontrada
      en: ", i - j)
      j = S[j - 1]
   sino
      si (i < m)  y (Patron[j] ≠ Text[i])
      entonces
         si (j ≠ 0) entonces
            j = S[j-1]
         sino
            i = i + 1
         fin si
      fin si
   fin si
fin mientras
```

2 < 8 cierto
A = A cierto
j = 1
i = 3
1 = 4 falso
3 < 8 y A ≠ A falso

240

Ejemplo de búsqueda KMP

PASO 8

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

3 < 8 cierto
A = A cierto
j = 2
i = 4
2 = 4 falso
4 < 8 y B ≠ B falso

PASO 9

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

4 < 8 cierto
B = B cierto
j = 3
i = 5
3 = 4 falso
5 < 8 y A ≠ A falso

241

Ejemplo de búsqueda KMP

PASO 10

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

5 < 8 cierto
A = A cierto
j = 4
i = 6
4 = 4 cierto
Incidencia encontrada en: 2
j = 1

PASO 11

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

6 < 8 cierto
A = D falso
1 = 4 falso
6 < 8 y A ≠ D cierto
1 ≠ 0 cierto
j = 0

242

Ejemplo de búsqueda KMP

PASO 12

```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

6 < 8 cierto
A = D falso
0 = 4 falso
6 < 8 y A ≠ D cierto
0 ≠ 0 falso
i = 7

PASO 13

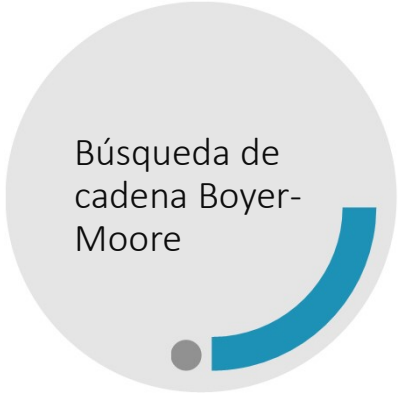
```
mientras (i < m) hacer
  si (Patron[j] = Text[i]) entonces
    j := j + 1
    i := i + 1
  fin si
  si (j = n) entonces
    Imprimir ("Incidencia encontrada
    en: ", i - j)
    j = S[j – 1]
  sino
    si (i < m)  y (Patron[j] ≠ Text[i])
    entonces
      si (j ≠ 0) entonces
        j = S[j-1]
      sino
        i = i + 1
      fin si
    fin si
  fin si
fin mientras
```

7 < 8 cierto
A = C falso
0 = 4 falso
7 < 8 y A ≠ C cierto
0 ≠ 0 falso
i = 8
7 < 8 falso

243

El algoritmo preprocesa la cadena objetivo (patrón) que está siendo buscada. En su verificación, el algoritmo intenta comprobar si hay una coincidencia en una posición particular marchando hacia atrás. El algoritmo precalcula dos tablas para procesar la información que obtiene en cada verificación fallada: una tabla calcula cuantas posiciones hay por delante en la siguiente búsqueda basada en el valor del carácter que no coincide; la otra hace un cálculo similar basado en cuantos caracteres coincidieron satisfactoriamente antes del intento de coincidencia fallado. Estas dos tablas devuelven resultados que indican cuán lejos "saltar" hacia delante, por este motivo son llamada en algunas ocasiones "tablas de salto". El algoritmo se desplazará con el valor más grande de los dos valores de salto cuando no ocurra una coincidencia. El algoritmo Boyer-Moore presenta en el peor de los casos una complejidad de O(n).

Búsqueda de cadena Boyer-Moore



244

Una simplificación del algoritmo que omite la "tabla primera" es el algoritmo Boyer-Moore-Horspool (BMH) que requiere, en el peor caso, mn comparaciones. El algoritmo BMH compara el patrón con el texto de derecha a izquierda, y se detiene cuando se encuentra una discrepancia con el texto. Cuando esto sucede, se desliza el patrón de manera que la letra del texto que estaba alineada con bm se quede alineada con algún bj, con j<m. Esta función sólo depende del patrón.

Búsqueda de cadena Boyer-Moore-Horspool (BMH)

245

Algoritmo de búsqueda BMH

```
Algoritmo BMH
// m es el largo del patrón
// n es el largo del texto
// los índices comienzan desde 1
k = m
j = m
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] = P[j]) entonces
    j = j - 1
  sino
    k = k + (m - siguiente(T[k]))
    j = m
  fin si
fin mientras
Si (j = 0) entonces
  imprimir ("Se encontró el patrón")
sino
  imprimir ("No se encontró el patrón")
fin si
fin
```

```
siguiente (T[k])
  val = 0
  desde (i = 1, i < m, i = i + 1)
    si (T[k] = P[i]) entonces
      val = i
      i = m
    fin si
  fin desde
  retornar (val)
fin siguiente
```

246

Ejemplo de búsqueda BMH

Utilizando el algoritmo BMH, busque la subcadena del arreglo P en el arreglo T.

P	R	U	E
P[1]	P[2]	P[3]	P[4]

T	R	S	U	P	R	U	E	A	C
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

PASO 1

```
// m es el largo del patrón
// n es el largo del texto
// los índices comienzan desde 1
k ← m
j ← m
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] = P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

m = 4
n = 10
k = 4
j = 4
4 ≤ 10 y 4 ≥ 1 cierto
T[4 – (4 – 4)] = T[4]
U = E falso
k = 4 + (4 – siguiente(T[4])

PASO 2

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
  si (T[k] = P[i]) entonces
    val = i
    i = m
  fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
U = P falso
i = 2
2 < 4 cierto
U = R falso
i = 3
3 < 4 cierto
U = U cierto
val = 3
i = 4
4 < 4 falso
retornar (3)

247

Ejemplo de búsqueda BMH

PASO 3

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] = P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

k = 4 + (4 – 3)
k = 3
j = 4
3 ≤ 10 y 4 ≥ 1 cierto
T[3 – (4 – 4)] = T[3]
S = E falso
k = 3 + (4 – siguiente(T[3])

PASO 4

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
  si (T[k] = P[i]) entonces
    val = i
    i = m
  fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
S = P falso
i = 2
2 < 4 cierto
S = R falso
i = 3
3 < 4 cierto
S = U cierto
val = 3
i = 4
4 < 4 falso
retornar (0)

PASO 5

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] = P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

k = 4 + (4 – 0)
k = 7
j = 4
7 ≤ 10 y 4 ≥ 1 cierto
T[7 – (4 – 4)] = T[7]
U = E falso
k = 7 + (4 – siguiente(T[7])

PASO 6

```
siguiente (T[k])
val = 0
desde (i = 1, i < m, i = i + 1)
  si (T[k] = P[i]) entonces
    val = i
    i = m
  fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
U = P falso
i = 2
2 < 4 cierto
U = R falso
i = 3
3 < 4 cierto
U = U cierto
val = 3
i = 4
4 < 4 falso
retornar (3)

248

Ejemplo de búsqueda BMH

PASO 7

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

k = 7 + (4 – 3)
k = 6
j = 4
6 ≤ 10 y 4 ≥ 1 cierto
T[6 – (4 – 4)] = T[6]
R = E falso
k = 6 + (4 – siguiente(T[6]))

PASO 8

```
siguiente (T[k])
val ← 0
desde (i ← 1, i < m, i ← i + 1)
  si (T[k] = P[i]) entonces
    val ← i
    i ← m
  fin si
fin desde
retornar (val)
fin siguiente
```

val = 0
i = 1
1 < 4 cierto
R = P falso
i = 2
2 < 4 cierto
R = R cierto
val = 2
i = 4
4 < 4 falso
retornar (2)

PASO 9

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

k = 6 + (4 – 2)
k = 8
j = 4

PASO 10

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

8 ≤ 10 y 4 ≥ 1 cierto
T[8 – (4 – 4)] = T[8]
E = E cierto
j = 3

249

Ejemplo de búsqueda BMH

PASO 11

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

8 ≤ 10 y 3 ≥ 1 cierto
T[8 – (4 – 3)] = T[7]
U = U cierto
j = 2

PASO 12

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

8 ≤ 10 y 2 ≥ 1 cierto
T[8 – (4 – 2)] = T[6]
R = R cierto
j = 1

PASO 13

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

8 ≤ 10 y 1 ≥ 1 cierto
T[8 – (4 – 1)] = T[5]
P = P cierto
j = 0

PASO 14

```
mientras (k ≤ n y j ≥ 1) hacer
  si (T[k - (m - j)] ≠ P[j]) entonces
    j ← j - 1
  sino
    k ← k + (m - siguiente(T[k]))
    j ← m
  fin si
fin mientras
```

8 ≤ 10 y 0 ≥ 1 falso
j = 0 cierto
Se encontró el patrón

250

Transformación de llaves o hash

El método llamado por transformación de claves o llaves (hash), permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta también con la ventaja de que el tiempo de búsqueda es prácticamente independiente del número de componentes del arreglo.

Trabaja basándose en una función de transformación o función hash (H) que convierte una clave en una dirección (índice) dentro del arreglo.

dirección ← H(clave)

251

Costos

- Tiempo de procesamiento requerido para la aplicación de la función hash
- Tiempo de procesamiento y los accesos E/S requeridos para solucionar las colisiones.

252

Ventajas

- Se pueden usar los valores naturales de la llave, puesto que se traducen internamente a direcciones fáciles de localizar.
- Se logra independencia lógica y física, debido a que los valores de las llaves son independientes del espacio de direcciones.
- No se requiere almacenamiento adicional para los índices.

253

Desventajas

- No pueden usarse registros de longitud variable
- El archivo no está clasificado
- No permite llaves repetidas
- Solo permite acceso por una sola llave

254

Técnicas de cálculo de direcciones

No hay reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves que asegure la máxima uniformidad en la distribución de las mismas. Realizar un análisis de las principales características de las claves puede ayudar en la elección de la función hash. A continuación, se detallan algunas de las funciones hash más utilizadas.

- Hashing por residuo
- Hashing por cuadrado medio
- Hashing por pliegue

255

Hashing por residuo

- Consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Suponga que se tiene un arreglo de N elementos, y K es la clave del dato a buscar.
- La función hash queda definida por la siguiente fórmula:

$$H(K) = (K \bmod N) + 1$$

- Para lograr una mayor uniformidad en la distribución, N debe ser un número primo o divisible entre muy pocos números. Por lo tanto, si dado N éste no es un número primo, se tomará el valor primo más cercano.

256

Ejemplo de hashing por residuo

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K1 y K2 dos claves a las que deben asignarse posiciones en el arreglo. Utilizando la función hashing por residuo, calcule las direcciones para K1 = 7259 y K2 = 9359.

Fórmula ➡ **H(K) = (K mod N) + 1**

Aplicando la fórmula con N = 100, para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = (7259 mod 100) + 1 = 60
$H(K_2)$ = (9359 mod 100) + 1 = 60

257

Ejemplo de hashing por residuo

Como $H(K_1)$ es igual a $H(K_2)$ y K_1 es distinto de K_2, se está ante una colisión. Se aplica ahora la fórmula 2.1 con N igual a un valor primo en vez de utilizar N igual a 100.

H(K1) = (7259 mod 97) + 1 = 82
H(K2) = (9359 mod 97) + 1 = 48

Con N = 97 se ha eliminado la colisión.

258

Hashing por cuadrado medio

- Consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos a tomar queda determinado por el rango del índice.
- Sea K la clave del dato a buscar. La función hash queda definida por la siguiente fórmula:

H(K) = dígitos_centrales (K^2) + 1

- La suma de una unidad a los dígitos centrales es para obtener un valor entre el 1 y N.

259

Ejemplo de hashing por cuadrado medio

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K1 y K2 dos claves a las que deben asignarse posiciones en el arreglo. Utilizando la función hashing por cuadrado medio, calcule las direcciones para K1 = 7259 y K2 = 9359.

Fórmula ➡ **H(K) = dígitos_centrales (K^2) + 1**

Aplicando la fórmula para calcular las direcciones a K1 y K2.

K12 = (7259)2 = 52693081
K22 = (9359)2 = 87590881

260

Ejemplo de hashing por cuadrado medio

H(K1) = dígitos_centrales (52693081) + 1 = 94
H(K2) = dígitos_centrales (87590881) + 1 = 91

Como el rango de índices en el ejemplo varía de 1 a 100, se toman solamente los dos dígitos centrales del cuadrado de las claves.

261

Hashing por pliegue

- Consiste en dividir la clave en partes de igual número de dígitos (la última puede tener menos dígitos) y operar con ellas, tomando como dirección los dígitos menos significativos. La operación entre las partes puede hacerse por medio de sumas o multiplicaciones.
- Sea K la clave del dato a buscar. K está formada por los dígitos d1, d2, …, dn. La función hash queda definida por la siguiente fórmula:

H(K) = dígmensig ((d1…di) + (di + 1…dj) + … + (d1…dn)) + 1

- El operador que aparece en la fórmula operando las partes de la clave es el de suma, pero como se aclaró antes, puede ser el de la multiplicación. La suma de una unidad a los dígitos menos significativos (dígmensig) es para obtener un valor entre 1 y N.

262

Ejemplo de hashing por pliegue

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K1 y K2 dos claves a las que deben asignarse posiciones en el arreglo. Utilizando la función hashing por pliegue, calcule las direcciones para K1 = 7259 y K2 = 9359.

Fórmula

H(K) = dígmensig ((d1…di) + (di + 1…dj) + … + (d1…dn)) + 1

Aplicando la fórmula para calcular las direcciones correspondientes a K1 y K2.

H(K1) = dígmensig (72 + 59) + 1 = dígmensig (131) + 1 = 32
H(K2) = dígmensig (93 + 59) + 1 = dígmensig (152) + 1 = 53

De la suma de las partes se toman solamente dos dígitos porque los índices del arreglo varían de 1 a 100.

263

Comparación entre las funciones hash

Aunque alguna otra técnica pueda desempeñarse mejor en situaciones particulares, la técnica del residuo de la división proporciona el mejor desempeño.

El método del medio del cuadrado puede aplicarse en archivos con factores de cargas bastantes bajas para dar generalmente un buen desempeño.

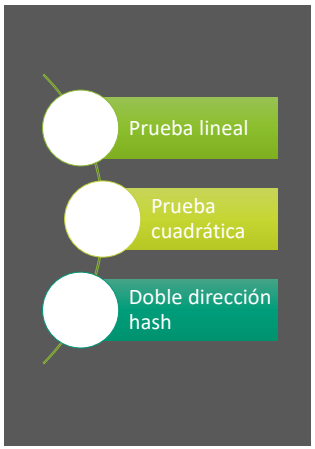
El método de pliegues puede ser la técnica más fácil de calcular, pero produce resultados bastante erráticos, a menos que la longitud de la llave sea aproximadamente igual a la longitud de la dirección.

Todas las funciones hash presentadas tienen destinado un espacio de tamaño fijo. Aumentar el tamaño del archivo relativo creado al usar una de estas funciones, implica cambiar la función hash, para que se refiera a un espacio mayor y volver a cargar el nuevo archivo.

264

Métodos para el manejo del problema de las colisiones

Tenemos una colisión cuando se asigna una misma dirección a dos o más claves distintas. La elección de un método adecuado para resolver colisiones es tan importante como la elección de una buena función hash. Se está ante una colisión cuando la función obtiene una misma dirección para dos claves diferentes. Normalmente, cualquiera que sea el método elegido resulta costoso tratar las colisiones. Es por ello, que se debe hacer un esfuerzo por encontrar la función que ofrezca mayor uniformidad en la distribución de las claves.



Prueba lineal

- Consiste en, una vez detectada la colisión, recorrer el arreglo secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía. Se trata el arreglo como una estructura circular: el siguiente elemento después del último es el primero.

- La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del arreglo permanecerían vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial, perdiendo así las ventajas del método hash.

Ejemplo de prueba lineal

Sea V un arreglo de 10 elementos. Calcule su dirección según la función Hash: H(K) = (K mod 10) + 1 para las claves K que se muestran en la tabla. En caso de colisiones solucione el mismo por medio de la prueba lineal.

k	25	43	56	35	54	13	80	104

Paso 1

Calcular la dirección para cada una de las claves K utilizando la función hash.

H(K) = (K mod 10) + 1

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Ejemplo de prueba lineal

Paso 2

Ubicar todas las claves que no tengan colisiones en el arreglo (color rojo), las que presenten colisiones se resolverán luego de terminar de ubicar las claves sin colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

v									
80			43	54	25	56			
1	2	3	4	5	6	7	8	9	10

Ejemplo de prueba lineal

Paso 3 Ubicar las claves con colisiones, estas se muestran en color rojo en la tabla.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se recorre el arreglo secuencialmente a partir del punto de colisión hasta encontrar una posición vacía.

K = 35

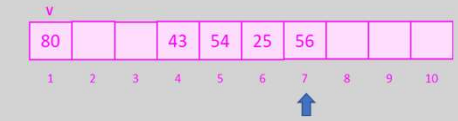
H(k) = 6

H'(k) = 7

269

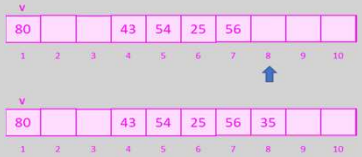
Ejemplo de prueba lineal

Paso 3 (continuación)



la posición se encuentra ocupada, por lo tanto, seguimos buscando

H'(k) = 8



270

Ejemplo de prueba lineal

Paso 4 Se ubicar la siguiente clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se recorre el arreglo secuencialmente a partir del punto de colisión hasta encontrar una posición vacía.

K = 13

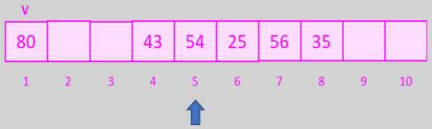
H(k) = 4

H'(k) = 5

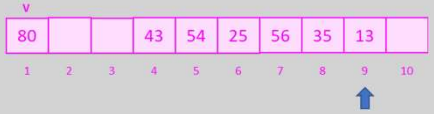
271

Ejemplo de prueba lineal

Paso 4 (continuación)



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo



272

Ejemplo de prueba lineal

Paso 5 Se procede de la misma forma con la última clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se recorre el arreglo secuencialmente a partir del punto de colisión hasta encontrar una posición vacía.

K = 104

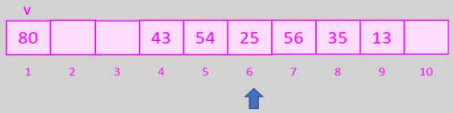
H(k) = 5

H'(k) = 6

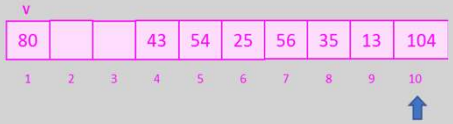
273

Ejemplo de prueba lineal

Paso 5 (continuación)



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo



274

Prueba cuadrática

- Este método es similar al de la prueba lineal. La diferencia consiste en que en el cuadrático las direcciones alternativas se generarán como D + 1, D + 4, D + 9, …, D + i2 en vez de D + 1, D + 2, …, D + i. Esta variación permite una mejor distribución de las claves colisionadas.

- La principal desventaja de este método es que pueden quedar casillas del arreglo sin visitar. Además, como los valores de las direcciones varían en i^2 unidades, resulta difícil determinar una condición general para detener el ciclo mientras. Este problema podría solucionarse empleando una variable auxiliar, cuyos valores dirijan el recorrido del arreglo, de tal manera, que se garantice que serán visitadas todas las casillas.

275

Ejemplo de prueba cuadrática

Sea V un arreglo de 10 elementos. Calcule su dirección según la función Hash: **H(K) = (K mod 10) + 1** para las claves K que se muestran en la tabla. En caso de colisiones solucione el mismo por medio de la prueba cuadrática.

k	25	43	56	35	54	13	80	104

Paso 1

Calculamos la dirección para cada una de las claves K utilizando la función hash .

H(K) = (K mod 10) + 1

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

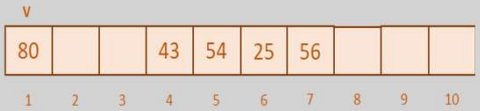
276

Ejemplo de prueba cuadrática

Paso 2

Ubicamos todas las claves que no tengan colisiones en el arreglo (color rojo), las que presenten colisiones se resolverán luego de terminar de ubicar las claves sin colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5



277

Ejemplo de prueba cuadrática

Paso 3 Ahora se procede a ubicar las claves con colisiones, estas se muestran en color rojo en la tabla.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se calculan las direcciones alternativas con D + i^2 a partir del punto de colisión hasta encontrar una posición vacía.

K = 35

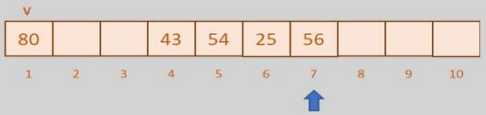
H(k) = 6

H'(k) = 6 + 1^2 = 7

278

Ejemplo de prueba cuadrática

Paso 3 (continuación)



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

279

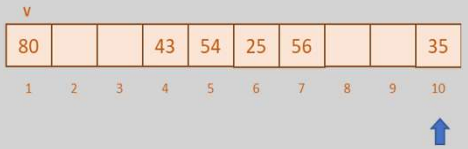
Ejemplo de prueba cuadrática

Paso 4

K = 35

H(k) = 6

H'(k) = 6 + 2^2 = 6 + 4 = 10



280

Ejemplo de prueba cuadrática

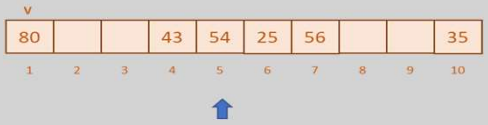
Paso 5 Se ubica la siguiente clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 13

H(k) = 4

H'(k) = 4 + 1^2 = 5



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

281

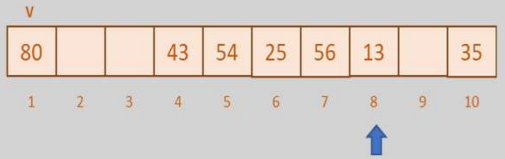
Ejemplo de prueba cuadrática

Paso 6

K = 13

H(k) = 4

H'(k) = 4 + 2^2 = 4 + 4 = 8



282

Ejemplo de prueba cuadrática

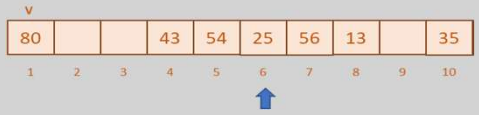
Paso 7 Se procede de la misma forma con la última clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 104

H(k) = 5

H'(k) = 5 + 1^2 = 6



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

283

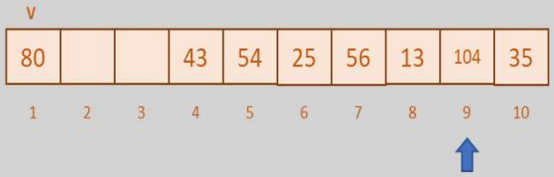
Ejemplo de prueba cuadrática

Paso 8

K = 104

H(k) = 5

H'(k) = 5 + 2^2 = 5 + 9



284

Doble dirección hash

- Consiste en, una vez detectada la colisión, generar otra dirección aplicando la función hash a la dirección previamente obtenida. El proceso se detiene cuando el elemento es hallado, o bien cuando se encuentra una posición vacía.

- La función hash que se aplica a las direcciones puede o no ser la misma que originalmente se aplicó a la clave. No existe una regla que nos permita decidir cuál será la mejor función que se puede emplear en el cálculo de las direcciones sucesivas.

285

Ejemplo de doble dirección hash

Sea V un arreglo de 10 elementos. Calcule su dirección según la función Hash: **H(K) = (K mod 10) + 1** para las claves K que se muestran en la tabla. En caso de colisiones solucione el mismo por medio de la doble dirección hash.

k	25	43	56	35	54	13	80	104

Paso 1

Calculamos la dirección para cada una de las claves K utilizando la función hash.

H(K) = (K mod 10) + 1

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

286

Ejemplo de doble dirección hash

Paso 2

Ubicamos todas las claves que no tengan colisiones en el arreglo (color rojo), las que presenten colisiones se resolverán luego de terminar de ubicar las claves sin colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

V

80			43	54	25	56			
1	2	3	4	5	6	7	8	9	10

287

Ejemplo de doble dirección hash

Paso 3 Ahora se procede a ubicar las claves con colisiones, estas se muestran en color rojo en la tabla.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

Se calculan las direcciones alternativas con **H(D) + 2** a partir del punto de colisión hasta encontrar una posición vacía.

K = 35

H(k) = 6

H'(k) = 6 + 2 = 8

V

80			43	54	25	56	35		
1	2	3	4	5	6	7	8	9	10

288

Ejemplo de doble dirección hash

Paso 4

Se ubica la siguiente clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 35

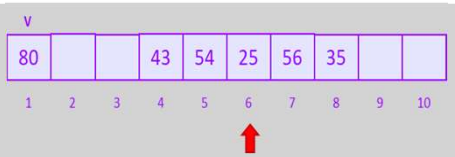
H(k) = 6

H'(k) = 6 + 2^2 = 6 + 4 = 10

289

Ejemplo de doble dirección hash

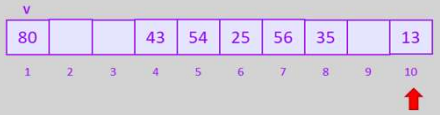
Paso 4 (continuación)



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

H'(k) = 6 + 2 = 8 -- posición ocupada

H'(k) = 8 + 2 = 10



290

Ejemplo de doble dirección hash

Paso 5

Se procede de la misma forma con la última clave con colisión.

k	25	43	56	35	54	13	80	104
H(k)	6	4	7	6	5	4	1	5

K = 104

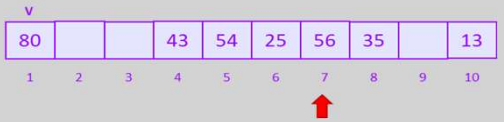
H(k) = 5

H'(k) = 5 + 2 = 7

291

Ejemplo de doble dirección hash

Paso 5 (continuación)



la posición se encuentra ocupada, por lo tanto, seguimos buscando hasta encontrar una posición vacía y se agrega en el arreglo

H'(k) = 7 + 2 = 9

292