Estructura de Datos

Estructuras de Datos No Lineales

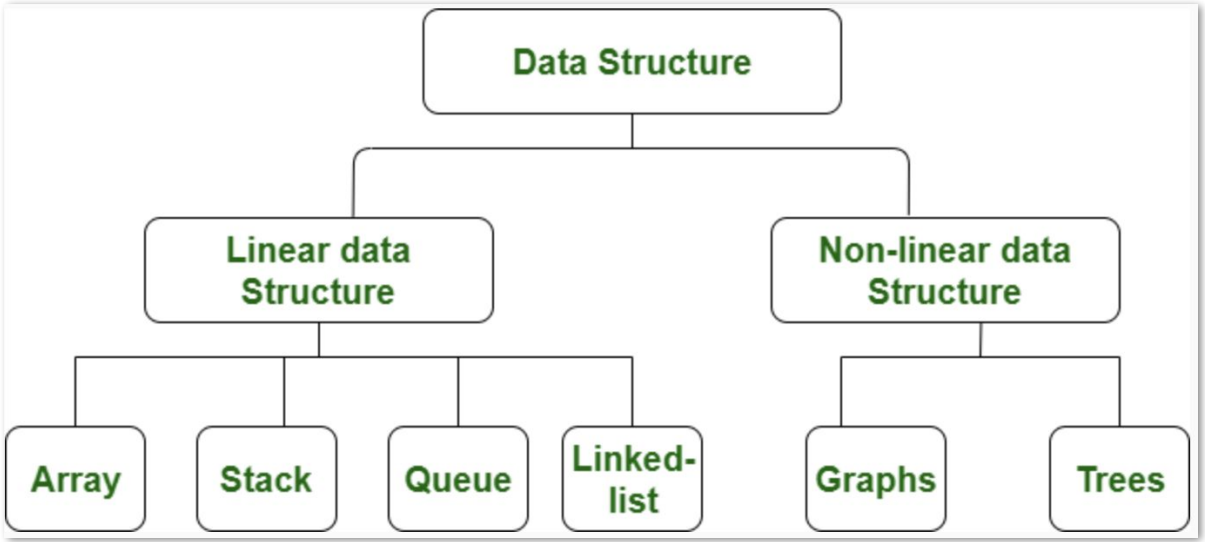
John Corredor Franco, PhD
Departamento de Ingeniería de Sistemas

Enero, 2024

➢ Agenda

- Estructuras de Datos
- Estructura de Datos No Lineal
- Propiedades
- Aplicaciones
- Árboles
  - Aplicaciones
  - Tipos de árboles
- Binary search tree
- Tablas Hash
  - Definición
  - Algoritmo en C++
  - Ejemplo
  - Resumen
  - Conclusiones

➢ Estructura de Datos No Lineal



Estructura de Datos: Diferencia entre lineales y no lineales

#	Estructura de Datos Lineal	Estructura de Datos No Lineal
1	Los elementos de datos se organizan en un orden lineal en el que todos y cada uno de los elementos se adjuntan a su adyacente anterior y siguiente.	Los elementos de datos se adjuntan de forma jerárquica.
2	Se trata como un solo nivel.	Se involucran múltiples niveles
3	Su implementación es fácil en comparación con la estructura de datos no lineal.	Si bien su implementación es compleja en comparación con la estructura de datos lineal.
4	Los elementos de datos se pueden recorrer en una sola ejecución.	Los elementos de datos no se pueden recorrer en una sola ejecución.

Estructura de Datos: Diferencia entre lineales y no lineales

#	Estructura de Datos Lineal	Estructura de Datos No Lineal
7	La memoria no se utiliza de manera eficiente.	La memoria se utiliza de manera eficiente.
8	Sus ejemplos son: • array, • pila, • cola, • lista enlazada, etc.	Sus ejemplos son: • árboles y grafos
9	Las aplicaciones de estructuras de datos lineales se encuentran principalmente en el desarrollo de software de aplicación.	Las aplicaciones de las estructuras de datos no lineales se encuentran en la inteligencia artificial y el procesamiento de imágenes.

➢ Estructura de Datos No Lineal

- La estructura de datos en la que los elementos de datos no están organizados secuencialmente se denomina estructura de datos no lineal. En otras palabras, los elementos de datos de la estructura de datos no lineal pueden estar conectados a más de un elemento para reflejar una relación especial entre ellos.
- Los elementos de datos se presentan en varios niveles, por ejemplo, en forma de árbol.
- En los árboles, los elementos de datos se disponen de forma jerárquica, mientras que en los grafos, los elementos de datos se disponen en orden aleatorio, utilizando las aristas y los vértices.
- Se necesitan varias ejecuciones para recorrer todos los elementos por completo. Es imposible recorrer toda la estructura de datos en una sola ejecución.
- Cada elemento puede tener múltiples caminos para llegar a otro elemento.

➢ Estructura de Datos No Lineal: Propiedades

- Se utiliza para almacenar los elementos de datos combinados siempre que no estén presentes en las posiciones de memoria contiguas.
- Es una forma eficaz de organizar y conservar adecuadamente los datos.
- Reduce el desperdicio de espacio de memoria proporcionando memoria suficiente para cada elemento de datos.
- Con la estructura de datos no lineal y se tiene múltiples opciones para atravesar de un nodo a otro.
- Los datos se almacenan aleatoriamente en la memoria.
- Es relativamente difícil de implementar.
- Múltiples niveles están involucrados.
- La utilización de la memoria es efectiva.

➢ Estructura de Datos No Lineal: Aplicaciones

- **Indexación de bases de datos:** Las estructuras de datos no lineales, como los árboles B y las tablas hash, se utilizan habitualmente para indexar bases de datos. Estas estructuras de datos proporcionan un acceso rápido a los datos de una base de datos, lo que permite ejecutar consultas con rapidez.
- **Infografía:** Las estructuras de datos no lineales, como los octrees y los árboles k-d, se utilizan en gráficos por ordenador para una indexación espacial y una detección de colisiones eficaces. Estas estructuras de datos permiten representar escenas complejas con rapidez y precisión.
- **Inteligencia artificial:** Las estructuras de datos no lineales, como los árboles de decisión y las redes neuronales, se utilizan en inteligencia artificial para tareas como la clasificación y la regresión. Estas estructuras de datos permiten aprender y modelar relaciones complejas entre entradas y salidas.
- **Enrutamiento de redes:** Las estructuras de datos no lineales, como las tablas de enrutamiento y las bases de datos de estado de enlace, se utilizan en los protocolos de enrutamiento de redes para determinar la mejor ruta para que los datos viajen a través de una red. Estas estructuras de datos permiten enrutar las redes de forma eficaz y fiable.

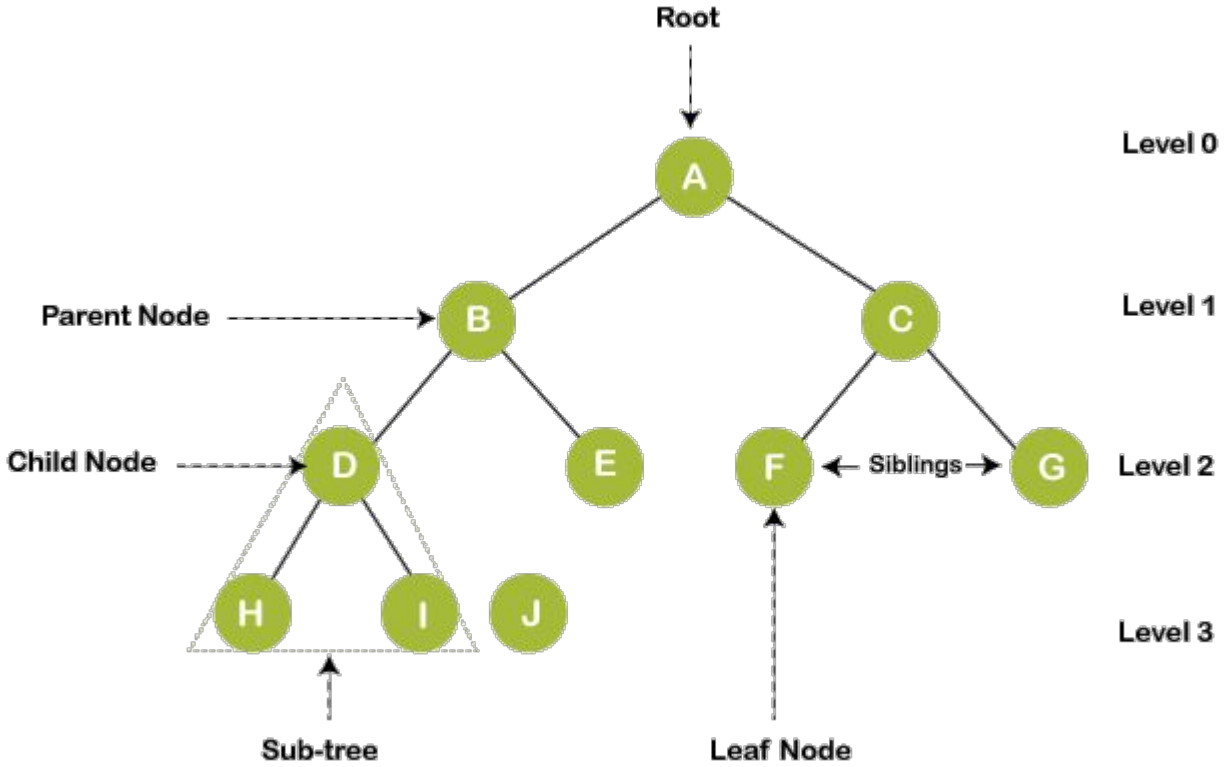
➢ Árboles

- El árbol es una estructura de datos no lineal que se compone de varios nodos. Los nodos de la estructura de datos en árbol están dispuestos en orden jerárquico.
- Consta de un nodo raíz al que corresponden varios nodos hijos, presentes en el siguiente nivel. El árbol crece por niveles, y los nodos raíz tienen nodos hijos limitados en función del orden del árbol.
- Por ejemplo, en el árbol binario, el orden del nodo raíz es 2, lo que significa que puede tener como máximo 2 hijos por nodo, no más que él.
- La estructura de datos no lineal no puede implementarse directamente, y se implementa utilizando la estructura de datos lineal como un array y una lista enlazada.
- El árbol en sí es una estructura de datos muy amplia y se divide en varias categorías como Árbol binario, Árbol de búsqueda binaria, Árboles AVL, Heap, max Heap, min-heap, etc.

➢ Árboles

- Una estructura de datos en árbol es una colección de objetos o entidades conocidos como nodos enlazados entre sí para representar o simular una jerarquía.
- Estos datos no se disponen en una ubicación contigua secuencial como hemos observado en un array, los elementos de datos homogéneos se colocan en la ubicación contigua de memoria para que la recuperación de los elementos de datos sea más sencilla.
- Una estructura de datos en árbol no es lineal porque no almacena secuencialmente. Se trata de una estructura jerárquica, ya que los elementos de un árbol se organizan en varios niveles.
- El nodo superior de la estructura de datos de árbol se conoce como nodo raíz. Cada nodo contiene datos de cualquier tipo. El nodo contiene el nombre del empleado en la estructura de árbol, por lo que el tipo de dato sería una cadena.
- Cada nodo contiene algunos datos y el enlace o referencia de otros nodos que pueden llamarse hijos.

Árboles


Root
A
Level 0
Parent Node
B
C
Level 1
Child Node
D
E
F
Siblings
G
Level 2
H
I
J
Level 3
Sub-tree
Leaf Node

➢ Árboles

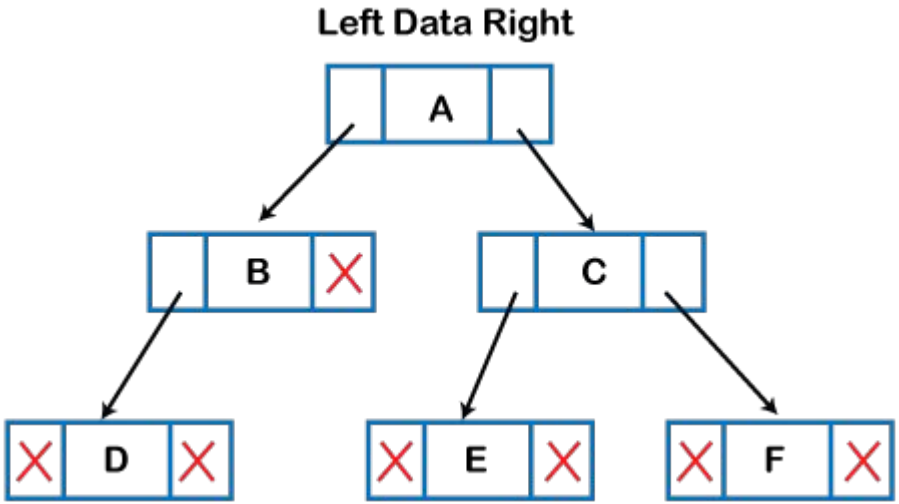
- **Root node**:
  - Es el nodo inicial del árbol a partir del cual crece cualquier árbol.
  - El nodo raíz está presente en el nivel 0 de cualquier árbol.
  - Dependiendo del orden del árbol, puede contener nodos hijos.
  - Por ejemplo, si en un árbol el orden es 3, entonces puede tener como máximo tres nodos hijos, y como mínimo, puede tener 0 nodos hijos.
- **Child node:**
  - El nodo hijo es el nodo que viene después del nodo raíz, que tiene un nodo padre y tiene algunos ancestros.
  - Es el nodo presente en el siguiente nivel de su nodo padre.
  - Por ejemplo, si algún nodo está presente en el nivel 5, es seguro que su nodo padre está presente en el nivel 4, justo por encima de su nivel.

➢ Árboles

- **Edge**:
  - Las aristas son el enlace entre el nodo padre y el nodo hijo se denomina enlace o conectividad de arista entre dos nodos.
- **Siblings:**
  - Los nodos que tienen el mismo nodo padre se llaman hermanos entre sí.
  - No son hermanos los nodos que tienen los mismos ancestros, y sólo el nodo padre debe ser el mismo, definido como hermano.
  - Por ejemplo, si el nodo A tiene dos nodos hijos, B y C, B y C se denominan hermanos.
- **Leaf node:**
  - El nodo hoja es el nodo que no tiene ningún nodo hijo. Es el punto final de cualquier árbol. También se conoce como nodo externo.

- **Internal nodes:**
  - Los nodos internos son nodos no hoja, que tienen al menos un nodo hijo con nodos hijos.
- **Degree of a node:**
  - El grado de un nodo se define como el número de nodos hijos.
  - El grado del nodo hoja siempre es 0 porque no tiene hijos, y el nodo interno siempre tiene al menos un grado, ya que contiene al menos un nodo hijo.
- **Altura del árbol**:
  - La altura del árbol se define como la distancia del nodo raíz al nodo hoja presente en el último nivel.
  - En otras palabras, la altura es el nivel máximo hasta el que se extiende el árbol.

Árboles: Implementación




```
struct node {
     int data;
     struct node *left;
     struct node *right;
}
```

➢ Árboles: Aplicaciones

Almacenamiento natural de datos jerárquicos: Los árboles se utilizan para almacenar los datos en la estructura jerárquica. Por ejemplo, el sistema de archivos. El sistema de archivos almacenados en la unidad de disco, el archivo y la carpeta están en la forma de los datos naturalmente jerárquicos y almacenados en forma de árboles.

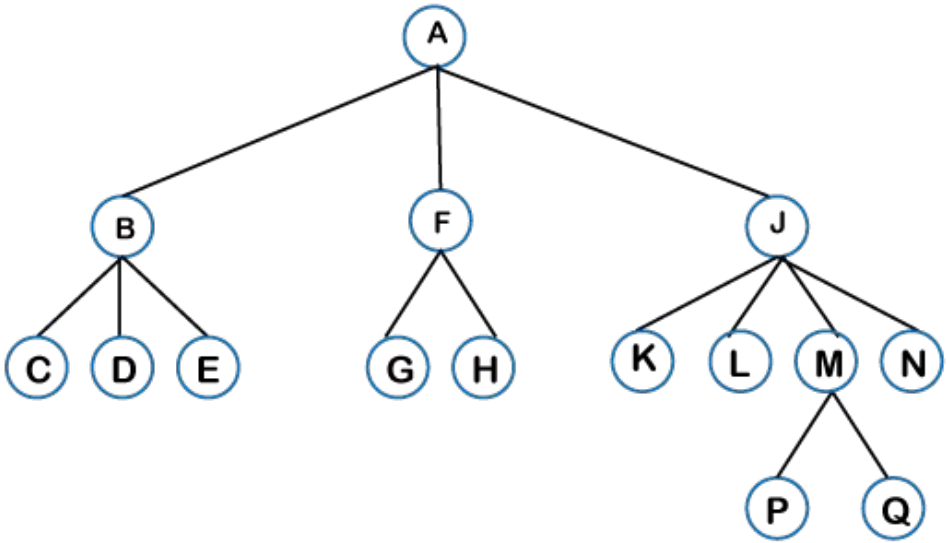
Organizar datos: Se utiliza para organizar los datos para una inserción, eliminación y búsqueda eficientes. Por ejemplo, un árbol binario tiene un tiempo logN para buscar un elemento.

Trie: Es un tipo especial de árbol que se utiliza para almacenar el diccionario. Es un método rápido y eficaz para la corrección ortográfica dinámica.

Heap: También es una estructura de datos en árbol que se implementa utilizando arrays. Se utiliza para implementar colas de prioridad.

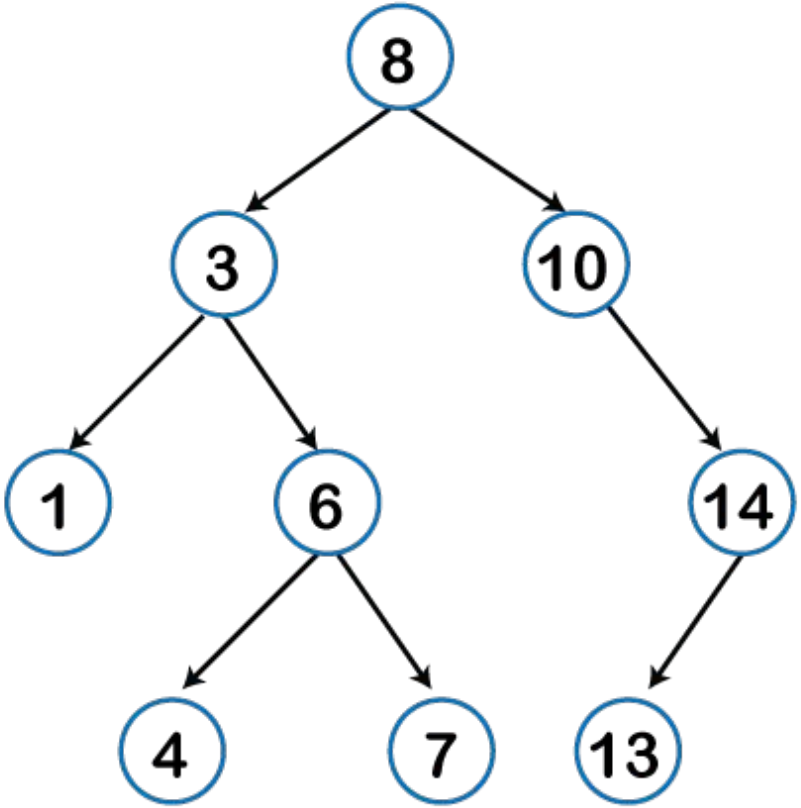
B-Tree y **B+Tree**: estructuras de datos en árbol utilizadas para implementar la indexación en bases de datos.

Tabla de rutas: La estructura de datos en árbol también se utiliza para almacenar los datos en tablas de enrutamiento en los routers.



General tree:

- Un nodo puede tener 0 o un máximo de **n** nodos.
- No se impone ninguna restricción al grado del nodo (el número de nodos que puede contener un nodo).
- Los hijos del nodo raíz se conocen como subárboles.
- En un árbol general puede haber n subárboles.
- En el árbol general, los subárboles no están ordenados, ya que los nodos del subárbol no pueden ordenarse.
- Cada árbol no vacío tiene una arista descendente, y estas aristas están conectadas a los nodos conocidos como nodos hijos.

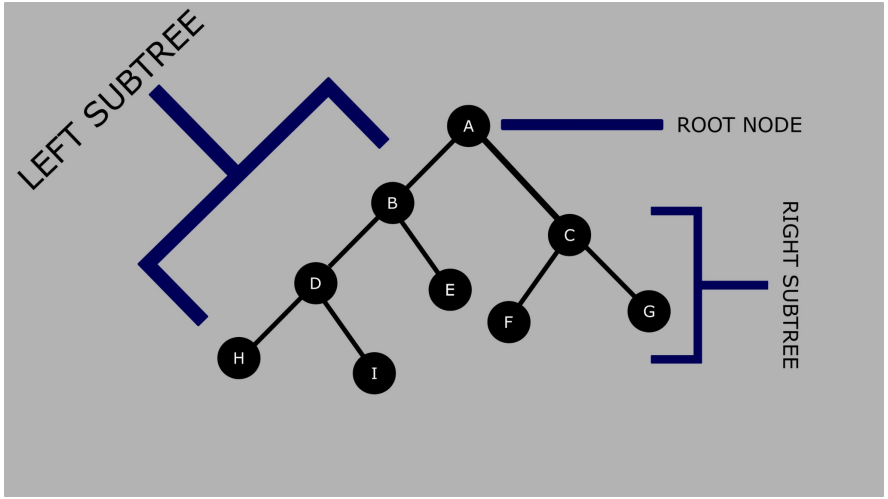


Binary tree:

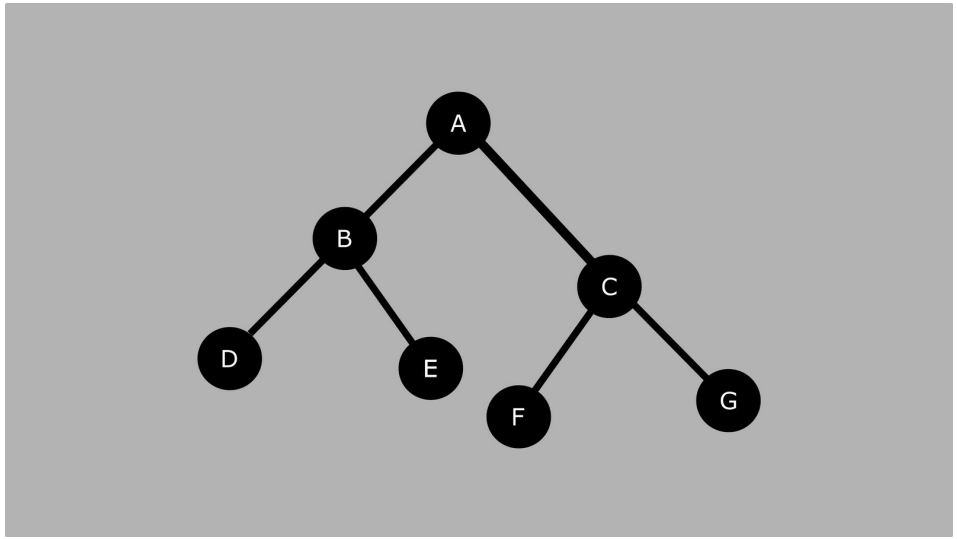
- Aquí, el propio nombre binario sugiere dos números, es decir, 0 y 1.
- En un árbol binario, cada nodo de un árbol puede tener dos nodos hijos como máximo.
- Aquí, máximo significa si el nodo tiene 0 nodos, 1 nodo o 2 nodos.

Binary Search Tree:

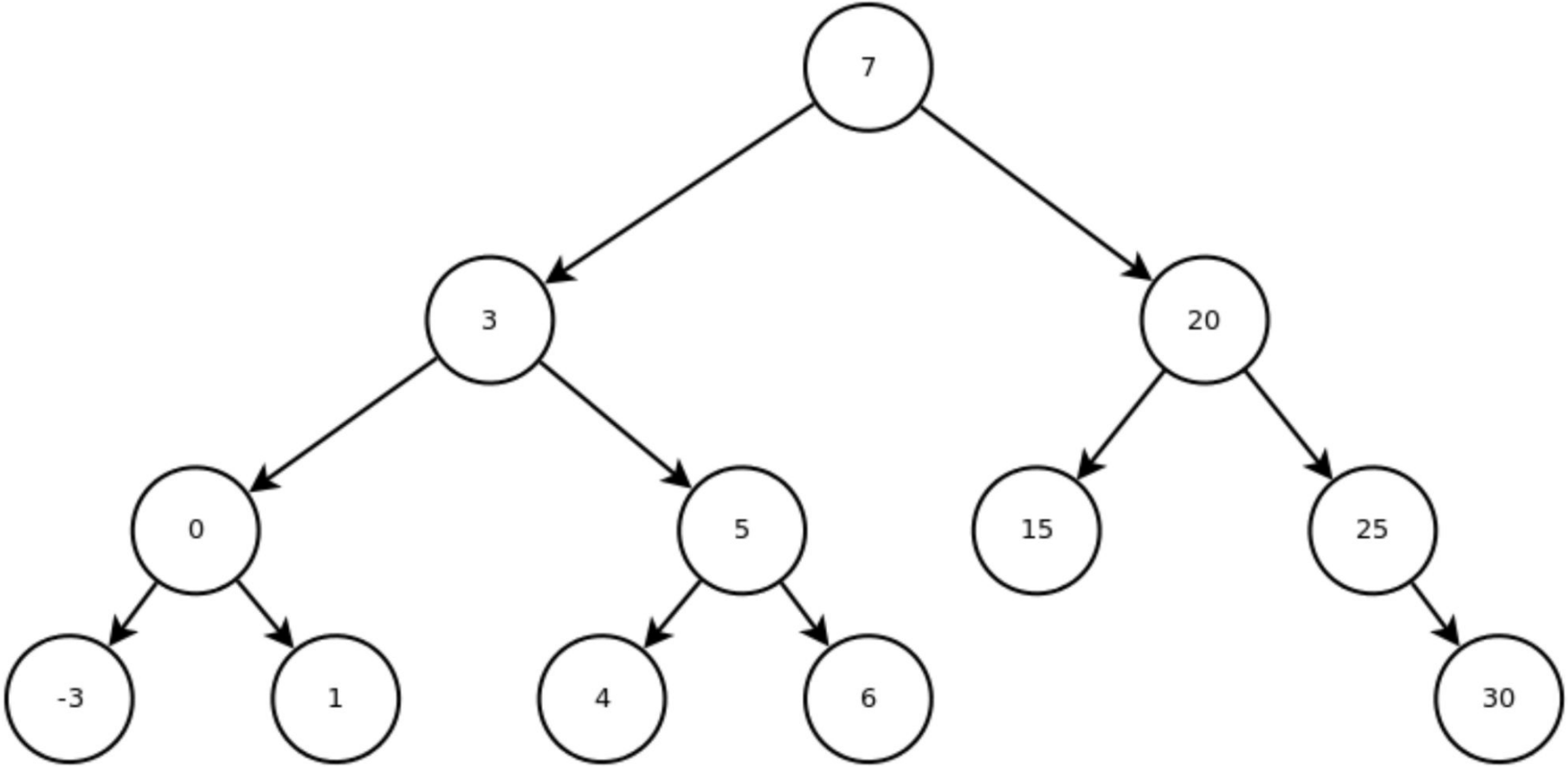
- Un árbol de búsqueda binario es un árbol binario formado por nodos. Cada nodo tiene una clave que indica su valor.
- El valor de los nodos del subárbol izquierdo es menor que el valor del nodo raíz. Y el valor de los nodos del subárbol derecho es mayor que el valor del nodo raíz.
- El nodo raíz es el nodo padre de ambos subárboles.



- **Inorder**, se pasa del subárbol izquierdo a la raíz y luego al subárbol derecho.
  - Inorder => Left, Root, Right. (LDR)
- **Preorden**, desde la raíz al subárbol izquierdo y luego al subárbol derecho.
  - Preorder => Root, Left, Right. (DLR)
- **Post orden**, se pasa del subárbol izquierdo al subárbol derecho y luego a la raíz.
  - Post order => Left, Right, Root. (LRD)



Inorder: D, B, E, A, F, C, G
Preorder: A, B, D, E, C, F, G
Pos order: D, E, B, F, G, C, A


7
3
20
0
5
15
25
-3
1
4
6
30

Estructura de Datos No Lineal (2)

- **Árboles**
  - Una estructura de datos de árbol consta de varios Nodos vinculados entre sí.
  - La estructura de un árbol es jerárquica que forma una relación como la del padre y el hijo.
  - La estructura del árbol está formada de manera que hay una conexión para cada relación de Node padre-hijo.
  - Solo debe existir una ruta entre la raíz y un Node en el árbol.
- **Grafos**
  - Consisten en una cantidad definida de vértices y aristas.
  - Los vértices o los Nodos están involucrados en el almacenamiento de datos y los bordes muestran la relación de los vértices.
  - La diferencia entre un grafo y un árbol es que en un grafo no existen reglas específicas para la conexión de los Nodos. Los problemas de la vida real como las redes sociales, las redes telefónicas, etc. se pueden representar a través de los grafos.

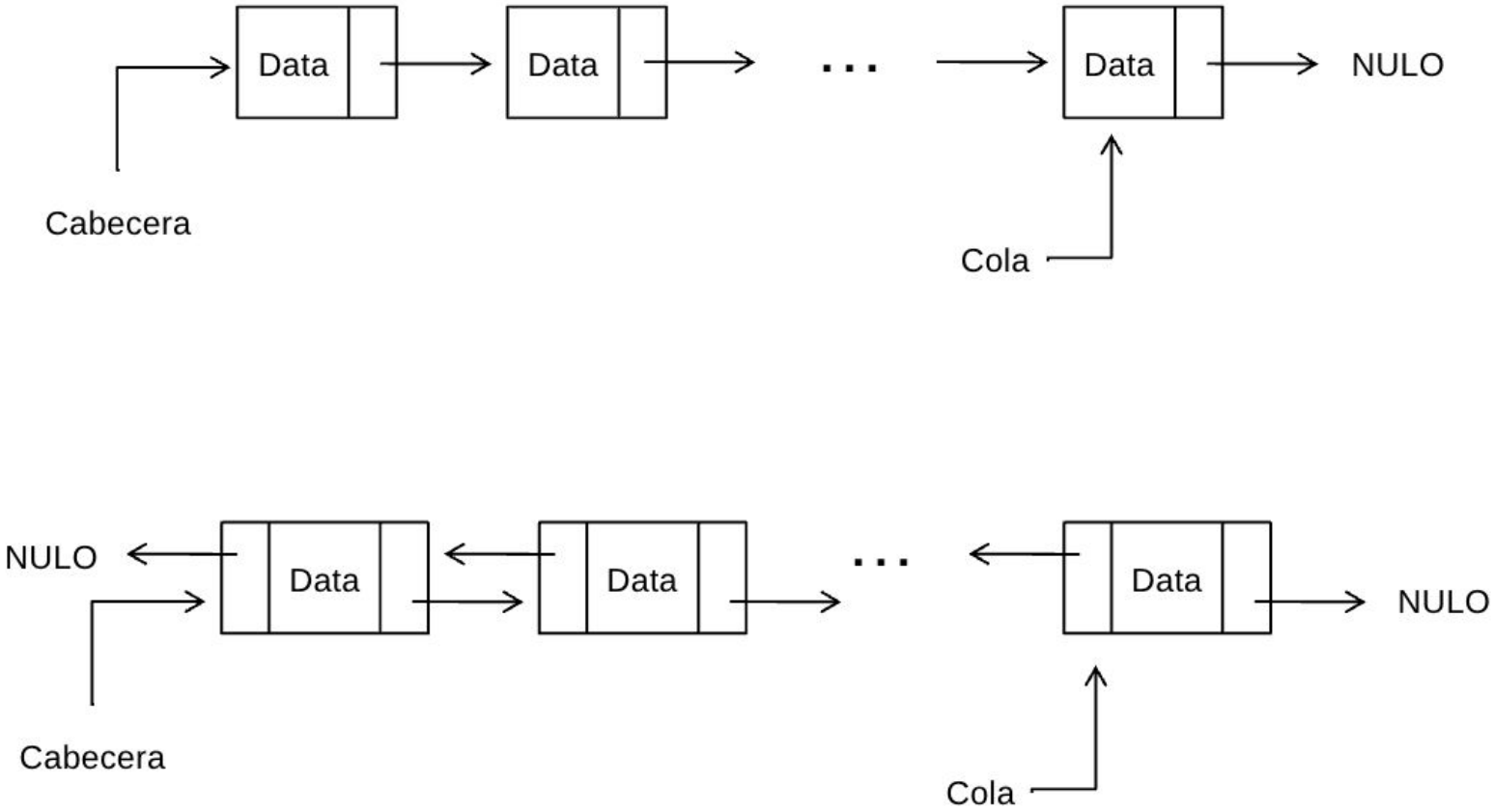
Listas Enlazadas

- Estructura formada por nodos que se enlazan entre sí partiendo de un nodo inicial cuya referencia lleva el nombre de cabecera y una posible referencia al nodo final o cola.
- Si cada nodo apunta sólo a su siguiente se dice que la lista enlazada es simple.
- Si también se incluye una referencia al nodo anterior, entonces la lista enlazada es doble .
- La mejor forma de entender estructuras enlazadas es manejar el diagrama de bloques y sus respectivos enlaces.
- Perder o no manejar bien un enlace provoca la pérdida de información a partir del bloque correspondiente.

Listas Enlazadas (2)

- Estructura formada por nodos que se enlazan entre sí partiendo de un nodo inicial cuya referencia lleva el nombre de cabecera y una posible referencia al nodo final o cola.
- Si cada nodo apunta sólo a su siguiente se dice que la lista enlazada es simple.
- Si también se incluye una referencia al nodo anterior, entonces la lista enlazada es doble .
- La mejor forma de entender estructuras enlazadas es manejar el diagrama de bloques y sus respectivos enlaces.
- Perder o no manejar bien un enlace provoca la pérdida de información a partir del bloque correspondiente.

Listas Enlazadas (3)



Listas Enlazadas: ejemplo

```
1  // Representa la data del nodo. Puede ser lo que se quiera.
2  // En este ejemplo se está usando el tipo int
3  //
4  typedef int DATA;
5
6  // Representa un nodo de una lista enlazada simple
7  //
8  class NodoS {
9      private:
10         DATA Dato;
11         NodoS *Sig;
12
13     public:
14         // Construye un nodo con un dato dado
15         //
16         NodoS(DATA d) { Dato = d, Sig = 0L; }
17
18         // Operaciones para obtener / cambiar el dato
19         //
20         DATA ObtDato() { return Dato; }
21         void AsgDato(DATA d) { Dato = d; }
22
23         // peraciones para cambiar / obtener la referencia al siguiente
24         //
25         NodoS *ObtSig() { return Sig; }
26         void AsgSig(NodoS *s) { Sig = s; }
27 };
```

Listas Enlazadas: ejemplo (2)

```
29  // Representa una lista enlazada simple con referencias a la cabecera y la cola
30  //
31  class ListaS {
32      private:
33          NodoS *Cab, *Col;
34          int N;
35
36          // Limpia la lista suprimiendo todos sus nodos
37          //
38          void Vaciar();
39
40      public:
41          ListaS() { Cab = Col = 0L; N = 0; }
42          ~ListaS() { Vaciar(); }
43
44          // Métodos para agregar un dato
45          //
46          bool AgregarPos(DATA d, int Pos);
47          bool AgregarComienzo(DATA d) { return AgregarPos(d, 0); }
48          bool AgregarFinal(DATA d) { return AgregarPos(d, N); }
49
50          // ...
```

Listas Enlazadas: ejemplo (3)

```
50          // ...
51
52          // Métodos para suprimir un dato
53          //
54          bool Suprimir(DATA d);
55          bool Suprimir(int Pos);
56
57          // Número de elementos en la lista
58          //
59          int NumElementos() { return N; }
60
61          // Si la lista está o no vacía
62          //
63          bool EstaVacia() { return N == 0; }
64
65          // Saber si un dato está o no en la lista
66          //
67          bool EstaDato(DATA d);
68
69          // Retornar el dato que está en una posición específica
70          //
71          DATA ObtData(int Pos);
72          DATA operator[](int Pos) { return ObtData(Pos); }
73  };
```

Listas Enlazadas: ejemplo (4)

```
70  // Representa un nodo de una lista enlazada doble
71  //
72  class NodoD {
73      private:
74          DATA Dato;
75          NodoD *Sig, *Ant;
76
77      public:
78          // Construye un nodo con un dato dado
79          //
80          NodoD(DATA d) { Dato = d, Sig = Ant = OL; }
81
82          // Operaciones para obtener / cambiar el dato
83          //
84          DATA ObtDato() { return Dato; }
85          void AsgDato(DATA d) { Dato = d; }
86
87          // peraciones para cambiar / obtener las referencia
88          //
89          NodoD *ObtSig() { return Sig; }
90          void AsgSig(NodoD *s) { Sig = s; }
91
92          NodoD *ObtAnt() { return Ant; }
93          void AsgAnt(NodoD *a) { Ant = a; }
94  };
```

Listas Enlazadas: ejemplo (5)

```
96  // Representa una lista enlazada doble
97  //
98  class ListaD {
99      private:
100         NodoD *Cab, *Col;
101         int N;
102
103         // Limpia la lista suprimiendo todos sus nodos
104         //
105         void Vaciar();
106
107     public:
108         ListaD() { Cab = Col = 0L; N = 0; }
109         ~ListaD() { Vaciar(); }
110
111         // Métodos para agregar un dato
112         //
113         bool AgregarPos(DATA d, int Pos);
114         bool AgregarComienzo(DATA d) { return AgregarPos(d, 0); }
115         bool AgregarFinal(DATA d) { return AgregarPos(d, N); }
116
117         // ...
```

Listas Enlazadas: ejemplo (6)

```
143  // Representa un conjunto
144  //
145  class Conjunto {
146      private:
147          ListaS S;
148
149      public:
150          // Agregar / suprimir elementos
151          //
152          void Agregar(DATA d) { S.AgregarFinal(d); }
153          void Suprimir(DATA d) { S.Suprimir(d); }
154
155          // Operaciones
156          //
157          Conjunto operator +(Conjunto S1);
158          Conjunto Union(Conjunto S1, Conjunto S2) { return S1 + S2; }
159
160          Conjunto operator *(Conjunto S1);
161          Conjunto Interseccion(Conjunto S1, Conjunto S2) { return S1 * S2; }
162
163          Conjunto operator -(Conjunto S1);
164          Conjunto Diferencia(Conjunto S1, Conjunto S2) { return S1 - S2; }
165
166          bool Pertenencia(DATA d) { return S.EstaDato(d); }
167          bool Vacio() { return S.EstaVacia(); }
168          int Cardinalidad() { return S.NumElementos(); }
169          bool SubConjunto(Conjunto S1);
170  };
```

Tablas Hash: Definición

- Una tabla hash es básicamente una estructura de datos que se utiliza para almacenar el par clave-valor.
- En C++, una tabla hash utiliza la función hash para calcular el índice en un array en el que el valor necesita ser almacenado o buscado.
- Este proceso de cálculo del índice se denomina hashing.
- Los valores en una tabla hash no se almacenan en orden y hay grandes posibilidades de colisiones en la tabla hash, que generalmente se resuelven mediante el proceso de encadenamiento (creación de una lista enlazada con todos los valores y las claves asociadas a ella).

Tablas Hash: Algoritmo en C++

- Inicializar el tamaño de la tabla a algún valor entero.
- Creación de una estructura de tabla hash hashTableEntry para la declaración de pares clave y valor.
- Crear el constructor de hashMapTable.
- Crear el hashFunction() y encontrar el valor hash que será un índice para almacenar los datos reales en la tabla hash utilizando la fórmula:

```
hash_value = hashFunction(key);
index = hash_value% (table_size)
```

Tablas Hash: Algoritmo en C++ (2)

- Funciones como:
  - **Insert()**
  - **searchKey()**
  - **Remove()**

  Se utilizan para la inserción del elemento en la clave, la búsqueda del elemento en la clave, y la eliminación del elemento en la clave, respectivamente.
- El destructor se llama para destruir todos los objetos de hashMapTable.

Tablas Hash: funcionamiento

Key	Index (usando una función hash)	Dato
12	12%10 = 2	23
10	10%10 = 0	34
6	6% 10 = 6	54
23	23 % 10 = 3	76
54	54 %10 = 4	75
82	81 %10 = 1	87

0	1	2	3	4	5	6	7	8	
34	87	23	76	75		54			

- Existen altas posibilidades de colisión ya que podría haber 2 o más claves que computen el mismo código hash resultando en el mismo índice de elementos en la tabla hash.

- Una colisión no puede ser evitada en el caso de hashing incluso si tenemos una tabla de gran tamaño. Podemos evitar una colisión eligiendo una buena función hash y un buen método de implementación.

Técnica básica de hashing abierto

- Aunque hay muchas técnicas de implementación usadas para ello como **Linear probing, open hashing**, etc.
- **La técnica básica de hashing abierto**, también llamada encadenamiento separado, se utiliza una lista enlazada para el encadenamiento de valores.
- Cada entrada en la tabla hash es una lista enlazada. Así, cuando se necesita una nueva entrada, el índice se calcula utilizando la clave y el tamaño de la tabla.
- Una vez calculado, se inserta en la lista correspondiente a ese índice. Cuando hay 2 o más valores que tienen el mismo valor hash/índice, se insertan las dos entradas correspondientes a ese índice enlazadas entre sí. Por ejemplo,

2 es el índice de la tabla hash recuperada mediante la función hash

12, 22, 32 son los valores de datos que se insertarán enlazados entre sí

2 —> 12 —->22 —->32

Ejemplo Tabla Hash en C++

```
#include <list>
class HashMapTable{
    // size of the hash table
    inttable_size;
    // Pointer to an array containing the keys
    list<int> *table;
public:
    // creating constructor of the above class containing all the methods
    HashMapTable(int key);
    // hash function to compute the index using table_size and key
    inthashFunction(int key) {
        return (key % table_size);
    }
    // inserting the key in the hash table
    void insertElement(int key);
    // deleting the key in the hash table
    void deleteElement(int key);
    // displaying the full hash table
    void displayHashTable();
};
```

Ejemplo Tabla Hash en C++ (2)

```
//creating the hash table with the given table size
HashMapTable::HashMapTable(intts){
    this->table_size = ts;
    table = new list<int>[table_size];
}
// insert function to push the keys in hash table
void HashMapTable::insertElement(int key){
    int index = hashFunction(key);
    table[index].push_back(key);
}
// delete function to delete the element from the hash table
void HashMapTable::deleteElement(int key){
    int index = hashFunction(key);
    // finding the key at the computed index
    list <int> :: iterator i;
    for (i = table[index].begin(); i != table[index].end(); i++){
        if (*i == key)
            break;
    }
```

Ejemplo Tabla Hash en C++ (2)

```
// removing the key from hash table if found
    if (i != table[index].end())
        table[index].erase(i);
    }
// display function to showcase the whole hash table
void HashMapTable::displayHashTable() {
    for (inti = 0; i<table_size; i++) {
        cout<<i;
        // traversing at the recent/ current index
        for (auto j : table[i])
            cout<< " ==> " << j;
            cout<<endl;
        }
    }
```

Ejemplo Tabla Hash en C++ (2)

```
// Main function
int main(){
    // array of all the keys to be inserted in hash table
    intarr[] = {20, 34, 56, 54, 76, 87};
    int n = sizeof(arr)/sizeof(arr[0]);
    // table_size of hash table as 6
    HashMapTableht(6);
    for (inti = 0; i< n; i++)
        ht.insertElement(arr[i]);
    // deleting element 34 from the hash table
    ht.deleteElement(34);
    // displaying the final data of hash table
    ht.displayHashTable();
    return 0;
}
```

Resumen del código ejemplo

- En el código, se crea un array para todas las claves que necesitan ser insertadas en la tabla hash.
- Se crean la clase y los constructores para hashMapTable para calcular la función hash utilizando la fórmula mencionada anteriormente.
- Se crea la lista como puntero al array de valores clave.
- Se crean funciones específicas para la inserción, borrado y visualización de la tabla hash y se llaman desde el método principal.
- Durante la inserción, si 2 o más elementos tienen el mismo índice, se insertan utilizando la lista uno tras otro.

Conclusiones

- Se explica claramente qué es una tabla hash en C++ y cómo se utiliza en los programas para almacenar los pares clave-valor.
- Las tablas hash se utilizan porque son muy rápidas para acceder y procesar los datos.
- Hay muchas posibilidades de colisiones al calcular el índice utilizando una función hash.
- Siempre debemos buscar métodos que nos ayuden a prevenir las colisiones.
- Así que hay que tener mucho cuidado al implementarlo en el programa.