5. Estructuras de Datos: Árboles

Un **árbol** es una estructura de datos jerárquica compuesta por nodos, donde un nodo raíz inicia la estructura y cada nodo puede tener cero o más nodos hijos. Los árboles permiten modelar relaciones de tipo jerárquico y se utilizan ampliamente en algoritmos y sistemas de información.

—

0.1. Características de los Árboles Generales

- **Nodos y Aristas:** Cada elemento es un nodo; las conexiones entre nodos se llaman aristas.
- **Raíz:** Nodo inicial sin padre.
- **Hojas:** Nodos sin hijos.
- **Altura:** Longitud máxima desde la raíz hasta una hoja.
- **Grado:** Número de hijos de un nodo.
- **Subárbol:** Árbol cuya raíz es un nodo cualquiera del árbol original.
- **No requiere memoria contigua:** Los nodos se pueden asignar dinámicamente.

—

0.2. Operaciones Básicas en Árboles Generales

1. **Recorrido:** Existen diversos algoritmos para recorrer un árbol. Los más importantes son DFS (Depth-First Search) y BFS (Breadth-first search).
2. **Inserción:** Añadir un nuevo nodo como hijo de un nodo existente.
3. **Eliminación:** Remover un nodo y gestionar la posición de sus subárboles (varios casos dependiendo de la implementación).
4. **Búsqueda:** Localizar un nodo con un valor dado mediante recorrido.
5. **Altura y Profundidad:** Calcular niveles de los nodos y altura del árbol.

—

0.3. Ejemplo de Árbol Genérico en C++

En el siguiente ejemplo, implementamos un árbol en el que cada nodo puede tener un número indeterminado de nodos hijos. Esto se hace mediante una lista de punteros a nodos.

Listing 1: Definición básica de nodo de árbol genérico

```cpp
#include <iostream>
#include <vector>
using namespace std;

struct Node {
    int data;
    vector<Node*> children;
    Node(int x) : data(x) {}
};

void traverse(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    for (Node* child : root->children) {
        traverse(child);
    }
}

int main() {
    Node* root = new Node(1);
    root->children.push_back(new Node(2));
    root->children.push_back(new Node(3));
    root->children[0]->children.push_back(new Node(4));
    root->children[0]->children.push_back(new Node(5));

    traverse(root); // 1 2 4 5 3
    return 0;
}
```

—

0.4. Árboles Binarios

Un **árbol binario** es un caso especial donde cada nodo puede tener a lo sumo dos hijos: izquierdo y derecho. Permite estructuras ordenadas (BST), árboles equilibrados (AVL, Red-Black), heaps, etc.

Características

- **Dos hijos máximo:** Cada nodo tiene punteros `left` y `right`.
- **Profundidad y Altura:** Similar a árboles generales.

Implementación Básica en C++

Listing 2: Nodo y recorrido in-order de árbol binario

```cpp
#include <iostream>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
    Node(int k) : key(k), left(nullptr), right(nullptr)
        {}
};

void inOrder(Node* root) {
    if (!root) return;
    inOrder(root->left);
    cout << root->key << " ";
    inOrder(root->right);
}

int main() {
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(15);
    root->left->left = new Node(3);
    root->left->right = new Node(7);

    inOrder(root); // 3 5 7 10 15
    return 0;
}
```

—

Un árbol de búsqueda binaria es un árbol binario en los hijos de cada nodo cumplen que el valor del medio está entre el valor izquierdo y el derecho. Se llaman así porque la búsqueda en estos árboles será mucho más eficiente (Si llegas a un nodo y buscar un valor menor, vas por la rama izquierda, si buscas un valor mayor, vas por la rama derecha).

—

Este último no puede confundirse con un Heap, una estructura de datos parecida que es un árbol que cumple la siguiente propiedad: Cada nodo tiene un valor mayor o igual que el de sus nodos hijos (en el caso de un max-heap. En un min-heap sería al revés). Este tiene muchas utilidades en la ordenación.

0.5. Ventajas y Desventajas de Árboles

Ventajas:

- Representan jerarquías de forma natural.

- Búsqueda, inserción y eliminación eficientes en árboles de búsqueda binaria.

Desventajas:

- Implementación compleja según tipo de árbol.
- Peor caso lineal si no es de búsqueda binaria.

—

0.6. Aplicaciones Comunes

- Sistemas de archivos y jerarquías.
- Estructuras de decisión (árboles de decisión).
- Implementación de colecciones ordenadas (sets, maps).

—

Como consejo final, es común que los algoritmos que se hagan en árboles usen recursividad. Por ejemplo, para calcular la altura de un arbol, será uno mas que la altura máxima de todos sus hijos. Este tipo de razonamiento es muy útil para estos problemas.