Recorridos de árboles.

Prof. Isabel Pita

Explicación de las transparencias de clase

Código desarrollado por el profesor A. Verdejo para la asignatura de Estructuras de Datos de la Facultad de Informática de la UCM.

12 de abril de 2021

1. Recorrido de árboles.

Recorrer un árbol consiste en visitar en cierto orden todos los nodos del árbol, haciendo algo con ellos. Dependiendo del orden en que se visitan los nodos podemos diferenciar los siguientes recorridos:

- Recorridos en profundidad. Los recorridos en profundidad, seleccionan una rama del árbol y avanzan por ella hasta que se termina, luego retroceden hasta encontrar una rama sin recorrer y avanzan por ella hasta recorrerla completa. Se da siempre prioridad a la rama izquierda sobre la derecha.
  - *Preorden*: se visita en primer lugar la raíz del árbol y, a continuación, se recorren en preorden el hijo izquierdo y el hijo derecho. R I D
  - *Inorden*: se recorre el hijo izquierdo, después se visita la raíz, y por último se recorre el hijo derecho. I R D
  - *Postorden*: primero se recorren los hijos izquierdo y derecho, en ese orden, y después se visita la raíz. I D R
- Recorrido por niveles o en anchura. Este recorrido recorre el árbol por niveles, primero la raíz, luego los nodos que están a una distancia 1 de la raíz, luego los nodos que están a una distancia 2 de la raíz etc. Este recorrido se utiliza para encontrar el nodo más cercano a la raíz que cumple una cierta propiedad.

1.1. Implementación de los recorridos

Todos los recorridos se encuentran implementados en la clase `bintree` que se encuentra en el campus virtual, en el apartado de árboles.

Recorridos en profundidad Los tres recorridos en profundidad se implementan de forma parecida.

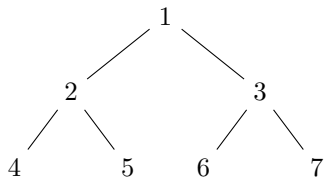
- Se define un método público sin parámetros que devuelve en un vector el valor de los nodos del árbol en el orden en que se recorre el árbol.
- La función pública llama a una función privada con dos parámetros, el primero es un *Link* (puntero inteligente) a la raíz del árbol que nos permite ir recorriendo toda la estructura del árbol, y el segundo es el vector de salida donde vamos construyendo el recorrido.
- La función privada tiene una implementación recursiva. El caso base se produce cuando el *Link* a la raíz es `nullptr`, es decir, cuando el árbol es vacío. En este caso no se hace nada, ya que no hay nada que añadir al recorrido. En el caso recursivo (cuando el *Link* no es vacío, recorremos la raíz del árbol, el hijo izquierdo y el hijo derecho en el orden que indica el recorrido).
- En *preorden* se añade el valor de la raíz al vector, después se recorre el hijo izquierdo y después el hijo derecho. Cada recorrido se hace de forma recursiva llamando a la misma función con el *Link* al hijo izquierdo o el *Link* al hijo derecho.
- En *inorden* se recorre el hijo izquierdo, después se añade el valor de la raíz al vector, y por último se recorre el hijo derecho.
- En *postorden* se recorre el hijo izquierdo, después el hijo derecho, y por último se añade el valor de la raíz al vector.

```
public:
  std::vector<T> preorder() const {
    std::vector<T> pre;
    preorder(raiz, pre);
    return pre;
 }

 std::vector<T> inorder() const {
    std::vector<T> in;
    inorder(raiz, in);
```

```
        return in;
    }

    std::vector<T> postorder() const {
        std::vector<T> post;
        postorder(raiz, post);
        return post;
    }

protected:
    static void preorder(Link a, std::vector<T> & pre) {
        if (a != nullptr) {
            pre.push_back(a->elem);
            preorder(a->left, pre);
            preorder(a->right, pre);
        }
    }

    static void inorder(Link a, std::vector<T> & in) {
        if (a != nullptr) {
            inorder(a->left, in);
            in.push_back(a->elem);
            inorder(a->right, in);
        }
    }

    static void postorder(Link a, std::vector<T> & post) {
        if (a != nullptr) {
            postorder(a->left, post);
            postorder(a->right, post);
            post.push_back(a->elem);
        }
    }
```

Los tres recorridos se pueden implementar de forma iterativa utilizando una pila para guardar la parte del árbol que habiendo sido ya recorrido no ha sido todavía visitada. El recorrido en inorden iterativo lo veremos un poco más adelante cuando se explique la implementación del iterador que recorre el árbol en inorden.

Recorrido en anchura La implementación del recorrido en anchura se realiza de forma iterativa. Dado el árbol:



El recorrido en anchura debe comenzar en la raíz. Después iremos al hijo izquierdo (nodo 2) , y después debemos ir al hijo derecho (nodo 3). Sin embargo, al acceder al nodo 3, perdemos el acceso a los hijos del nodo 2 (nodos 4 y 5). Debemos por lo tanto mantener en una estructura de datos los accesos a los nodos que vamos dejando pendientes de visitar. Observamos que cuando acabamos de visitar el nodo 1, aparecen como pendientes el 2 y el 3, de estos debe tratarse antes el nodo 2 que el 3 (primero que aparece es el primero que se debe tratar), por lo tanto el tipo de datos que utilizaremos es una cola.

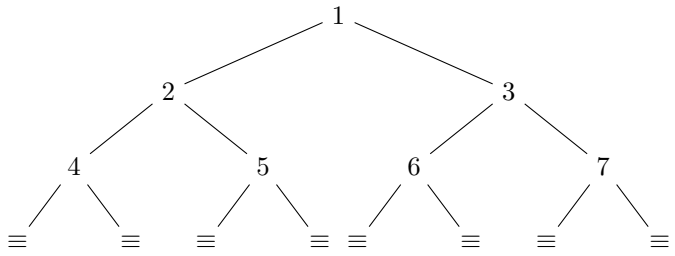
Si vemos la implementación, en la cola `pendientes` guardamos todos los *Link* de los nodos que tenemos pendientes de visitar. Cuando tratamos un nodo, guardamos en la cola el `Link` al hijo izquierdo y el `Link` al hijo derecho si no son nulos. Mientras haya `Links` en la cola sacamos el primero, lo tratamos y guardamos sus hijos para que sean tratados posteriormente.

```
    std::vector<T> levelorder() const {
```

```
    std::vector<T> levels;
    if (!empty()) {
        std::queue<Link> pendientes; // Cola de pendientes
        pendientes.push(raiz);  // Inicializamos la cola con el Link raiz
        while (!pendientes.empty()) {
            Link sig = pendientes.front();  // Obtenemos el primero de la cola
            pendientes.pop();
            levels.push_back(sig->elem);   // Lo tratamos
            if (sig->left != nullptr)   // Anadimos el hijo izquierdo si no es nulo
                pendientes.push(sig->left);
            if (sig->right != nullptr)  // Anadimos el hijo derecho si no es nulo
                pendientes.push(sig->right);
        }
    }
    return levels;
}
```

1.2. Reconstrucción de un árbol binario a partir de sus recorridos.

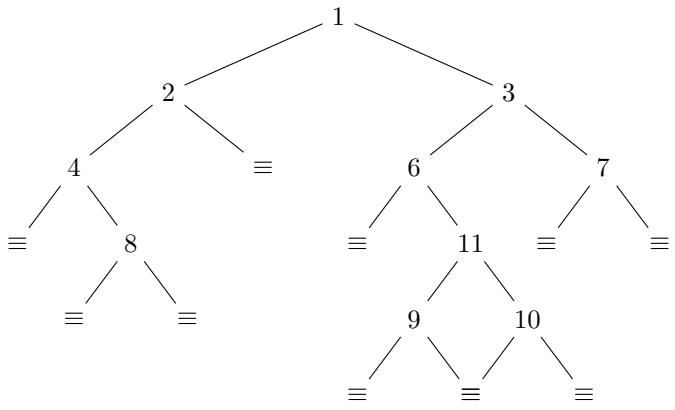
Dado un árbol



observamos que en su recorrido en preorden: 1 2 4 5 3 6 7, el primer nodo del recorrido es la raíz del árbol, a continuación se sitúan los valores de los nodos que pertenecen a su hijo izquierdo y a continuación los valores de los nodos de su hijo derecho.

Si tenemos un árbol binario completo [1], como el de la figura anterior, podemos obtener el árbol a partir de su recorrido en preorden. La raíz del árbol es el primer elemento del vector y la primera mitad de los elementos siguientes (2 4 5) forman su hijo izquierdo y la segunda mitad (3 6 7) los elementos de su hijo derecho.

Sin embargo, si el árbol binario no es completo



no podemos saber cuales son los valores del preorden que pertenecen a su hijo izquierdo y cuales a su hijo derecho. No podemos obtener por lo tanto un árbol binario que no sea completo a partir de su

[1] Un árbol binario es completo si es el árbol vacío o si la altura de sus dos hijos es la misma y además ambos hijos son árboles completos

preorden, porque no es único.

Ejercicio 1: Dibujar dos árboles binarios distintos, cuyo recorrido en preorden sea: 1 2 3 4.

¿Existe algún recorrido que permita calcular el número de elementos que pertenecen a la parte izquierda del árbol y el número de elementos que pertenecen a la parte derecha si todos los nodos tienen valores diferentes y sabemos el valor de la raíz?

Observad que se pide que todos los valores del árbol sean diferentes. Además sabemos cuál es la raíz del árbol.

El recorrido en inorden del árbol anterior es: 4 8 2 1 6 9 11 10 3 7. Sabiendo que la raíz del árbol es el valor 1, observamos que tenemos 3 valores a su izquierda (4 8 2) y 6 valores a su derecha (6 9 11 10 3 7). Los valores de la izquierda forman el árbol izquierdo y los valores de la derecha el árbol derecho.

Sabiendo el número de valores que pertenecen al árbol izquierdo y el número de valores que pertenecen al árbol derecho, podemos obtener a partir del preorden, el preorden del árbol izquierdo y el preorden del árbol derecho. Podemos también obtener el inorden del lado izquierdo y el inorden del árbol derecho y de esta forma construir recursivamente el árbol izquierdo y el árbol derecho.

Ejercicio 2: ¿Se pueden reconstruir un árbol binario a partir de los recorridos en postorden y en inorden?

Ejercicio 3: ¿Se puede reconstruir un árbol binario a partir de los recorridos en preorden y en postorden?. En caso negativo dar un contraejemplo.