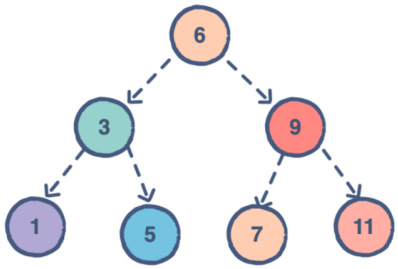
TDA Abb



[7541/9515] Algoritmos y Programación II
Primer cuatrimestre de 2022

Alumno:	Davies, Alen
Número de padrón:	107084
Email:	adavies@fi.uba.ar

Índice

1. Introducción **2**

2. Árboles **2**
2.1. Árboles Binarios 2
2.1.1. Operaciones 2
2.1.2. Recorridos 3
2.2. Árboles Binarios de Búsqueda 3

3. Implementación TDA Abb **3**
3.1. Crear 4
3.2. Insertar 4
3.3. Quitar 4
3.4. Recorrer 5
3.5. Buscar 5
3.6. Destruir 6
3.7. Iterador interno 6

4. Compilación y ejecución **6**

1. Introducción

Para este TP se pide implementar un TDA Abb utilizando nodos simplemente enlazados. Para ello se brindan las firmas de las funciones a implementar en archivos .c y .h contenidos en una carpeta src. También se pide la creación de un iterador interno para recorrer el árbol.
El TDA entregado deberá compilar y pasar las pruebas de la cátedra en Chanutron sin errores y sin pérdida de memoria.
Además de la carpeta src que contiene los archivos con las firmas de las funciones a implementar, se brinda tambien un archivo de ejemplo para verificar que el TDA funcione correctamente.
Otra de las cosas pedidas para este TP es la implementación de las pruebas y para ello se incluye un archivo pruebas.c que debe ser completado con las pruebas de las primitivas del TDA.

2. Árboles

Los árboles son estructuras de datos muy similares a las listas enlazadas, en el sentido que tienen punteros que apuntan a otros elementos, pero tienen una estructura lógica de tipo ramificada. Cada elemento del árbol denominado nodo contiene un valor.
Un árbol es una estructura de datos no lineal en la que cada nodo puede apuntar a uno o varios nodos.
Una forma de definir a los árboles es de forma recursiva. Los árboles son una colección de nodos que posee un nodo principal llamado raíz, el cual puede estar conectado a subárboles, cada uno de ellos con su raíz conectada al nodo raíz. A cada nodo que está conectado a una raíz se los llama nodo hijo y al nodo raíz de estos se lo llama nodo padre. A los nodos que poseen el mismo padre se los llama nodos hermanos. A los nodos que no tienen hijos se los llama nodos hoja.

2.1. Árboles Binarios

Es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos.
Los árboles binarios están relacionados con las operaciones de búsqueda ya que cuando se realiza una operación de búsqueda uno debe saber dónde seguir buscando, si a la derecha o a la izquierda de un de terminado valor y los árboles binarios al tener el nodo raíz solamente conectado a dos subárboles, nos permite tener la noción de izquierda y derecha.

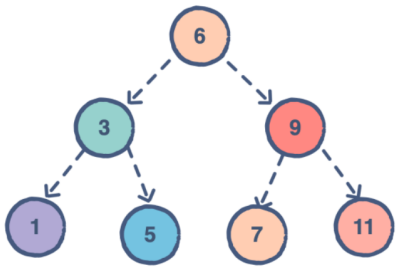
2.1.1. Operaciones

Las operaciones básicas de un árbol binario son:

- Crear.
- Destruir.
- Vacío.
- Insertar.
- Quitar.
- Buscar.
- Recorrer.

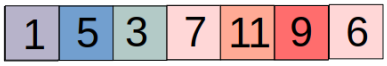
2.1.2. Recorridos

Recorrer un árbol implica pasar por cada uno de los nodos. Existen tres formas de recorrer un árbol binario.

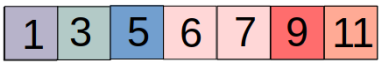


- Preorden: Primero se visita en nodo actual, luego el sub-árbol izquierdo y luego el derecho.

- Postorden: Primero se visita el sub-árbol izquierdo, luego al sub-árbol derecho y por último al nodo actual.



- Inorden: Primero se visita el sub-árbol izquierdo, luego el nodo actual y por último el sub-árbol derecho.



2.2. Árboles Binarios de Búsqueda

Un árbol binario de búsqueda (ABB) es un árbol binario en el cual tenemos un orden y cada nodo del árbol posee un valor único.
Las claves mayores se insertan en los subárboles derechos, Las claves menores se insertan en los subárboles izquierdos y Ambos subárboles también son árboles de búsqueda binaria.
Las operaciones de los árboles binarios de búsqueda son las mismas que las de los árboles binarios.

3. Implementación TDA Abb

A continuación se detallarán las operaciones más importantes con su complejidad y su implementación en el TDA.
Para este TP se nos proporcionó la siguiente estructura de nodo, la cual se debió utilizar para implementar el TDA:

```c
typedef struct nodo_abb {
        void *elemento;
        struct nodo_abb *izquierda;
        struct nodo_abb *derecha;
} nodo_abb_t;
```

Y la siguiente estructura de árbol:

```c
typedef struct abb {
        nodo_abb_t *nodo_raiz;
        abb_comparador comparador;
        size_t tamanio;
} abb_t;
```

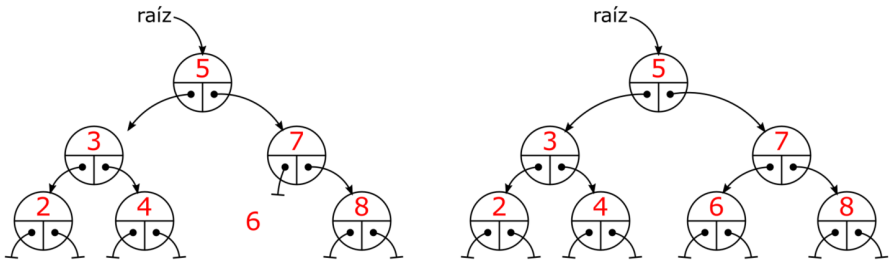
Para la implementación de este TDA, realicé varias funciones auxiliares que me permitieran utilizar la recursividad desde la raíz del árbol ya que las primitivas provistas recibían al árbol como parámetro. Esto lo realicé para las operaciones: insertar, quitar, buscar y destruir.
También hice una función para cada tipo de recorrido.

3.1. Crear

Para crear un árbol se reserva memoria para la estructura y se le asigna una función comparadora que no puede ser nula.
Su complejidad es O(1) ya que se realiza una sola vez.

3.2. Insertar

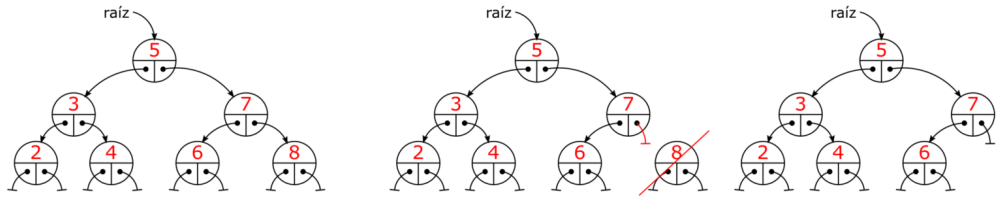
Para insertar un nuevo elemento en el árbol se debe compara el elemento a insertar con la raíz. Si es mayor avanzo hacia el subárbol derecho, si es menor hacia el izquierdo.
Repetir el paso anterior hasta encontrar un elemento con igual valor o llegar al final del sub-árbol donde debe insertarse el nuevo elemento.
Cuando se llega al final creo un nuevo nodo, asignando null a los punteros izquierdo y derecho del mismo.
Luego coloco el nuevo nodo como hijo izquierdo o derecho del anterior según sea su valor.
La complejidad es O(log(n)) ya que en cada comparación se excluyen la mitad de los subárboles.
Insertamos el 6.



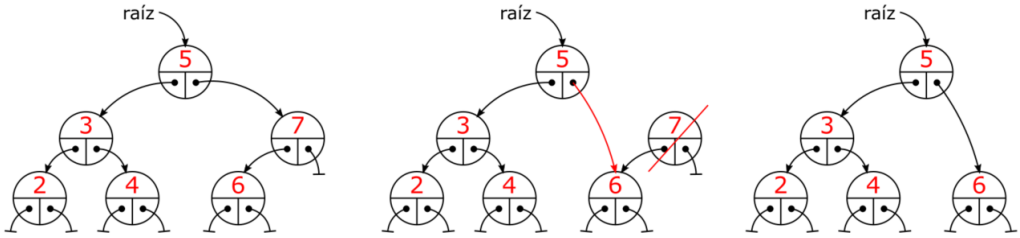
3.3. Quitar

La operación de eliminación de un nodo es también una extensión de la operación de búsqueda. Bajamos por el árbol buscando el nodo a borrar, una vez encontrado hay diferentes casos a tener en cuenta:

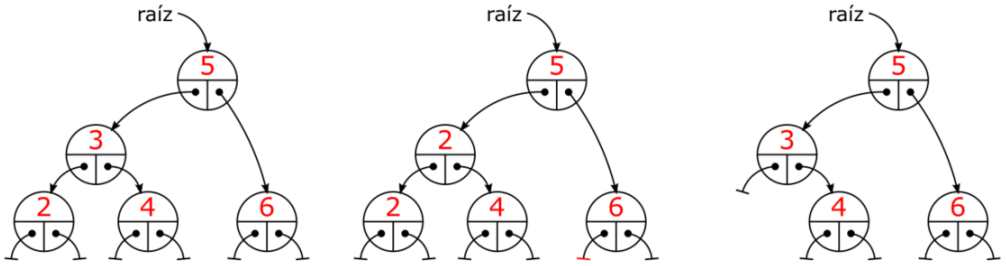
- Caso 1: Si no tiene hijos, podemos borrarlo directamente.
  Quitamos el 8.



- Caso 2: Si tiene un solo hijo, primero el padre del que queremos borrar se enlaza con el hijo del que queremos borrar, luego borramos el elemento.
  Quitamos el 7.



- Caso 3: Si tiene dos hijos, se sustituye el elemento por el predecesor inorden (el más grande del subárbol izquierdo). Para localizar este elemento se debe parar en el hijo izquierdo y avanzar siguiendo siempre la rama derecha de cada nodo hasta llegar al final. El nodo encontrado es el que va a reemplazar al nodo a borrar. Luego ya podemos quitar el elemento.
  Quitamos el 3.



La complejidad es O(log(n)) ya que en cada comparación se excluyen la mitad de los subárboles donde podría encontrarse el elemento a borrar.

3.4. Recorrer

Para recorrer un árbol primero se visita el nodo actual, si es una hoja finaliza el recorrido. Sino se recorre el subárbol derecho y el izquierdo. El orden de estos pasos depende del tipo de recorrido especificado por parámetro a la función.
La complejidad es O(n) ya que se visitan todos los elementos del árbol.

3.5. Buscar

La búsqueda de un elemento comienza en la raíz del árbol. Luego se compara el elemento buscado con el elemento de la raíz. Si el elemento buscado es mayor, se realiza lo anterior pero con el subárbol derecho y si es menor con el izquierdo.
La complejidad es O(log(n)) ya que en cada comparación se excluyen la mitad de los nodos donde se podría encontrar el elemento buscado.

3.6. Destruir

Para destruir el árbol se debe recorrer cada nodo destruyendo cada uno y por último se destruye el árbol. Su complejidad es O(n) ya que se recorren todos los elementos del árbol.

3.7. Iterador interno

En éste TP se pedia implementar un iterador interno que permitiera recorrer los elementos del árbol y que devuelva la cantidad de elementos recorridos.
La función del iterador interno, itera el árbol con el recorrido especificado por parámetro llamando a la función correspondiente a ese recorrido.

4. Compilación y ejecución

Dentro del .zip entregado, se encuentra un makefile el cual contiene los comandos para compilar el programa, testear las pruebas y errores de memoria.