TDA 2 - ABB

[7541/9515] Algoritmos y Programación II

Segundo Cuatrimestre - 2021

Alumno:	**González Calderón, Julián**
Padrón:	107938
Email:	gonzalezcalderonjulian@gmail.com

1. Introducción

La idea general del trabajo practico era implementar el TDA ABB (árbol binario de búsqueda) utilizando nodos, cada nodo tenia la referencia a sus nodos hijos y para almacenar el árbol solo se guarda la referencia al nodo raíz. Además, los nodos estaban relacionados entre si. Los elementos a la izquierda eran menores a la raíz, mientras que los elementos a la derecha eran mayores.

Para seguir con los requisitos del trabajo practico, teníamos un ***contrato*** que explicaba las funciones que debíamos diseñar con sus debidas especificaciones.

Además de programar el ABB, nosotros debemos diseñar nuestras propias pruebas para verificar el correcto funcionamiento del mismo..

2. Teoría

- **¿Que es un árbol?**

  Un árbol es un tipo de dato abstracto compuesta por nodos, donde cada nodo tiene acceso a sus hijos. Los nodos pueden tener distintas relaciones entre si, así como tener distinto números de hijos/valores, dependiendo del tipo de árbol en cuestión.

- **¿Que es un árbol binario?**

  Un árbol binario es aquel que cada nodo tiene como máximo **dos** hijos. Uno izquierdo y uno derecho.

- **¿Que es un árbol binario de búsqueda?**

  Un árbol binario de búsqueda es aquel que, además de que cada nodo tenga un máximo de dos hijos, estos nodos están relacionados entre si. Los nodos mayores se encuentran en la rama izquierda mientras que los nodos menores se encuentran en la rama derecha. Esto permite buscar valores del árbol descartando parte del árbol a cada comparación. reduciendo la complejidad de la búsqueda de un función lineal a una logarítmica.

- **¿Cuales son las partes importantes de un árbol?**

  El árbol tiene distintas propiedades asignadas:

- **Nodo Padre:** Es la raíz del árbol principal
- **Nodos Hijos:** Son las raíces de los sub-árboles
- **Nodos Hojas:** Son aquellos nodos que no tienen hijos
- **Profundidad:** Cada nodo tiene una profundidad, definida como el numero de nodos que hay que atravesar para llegar al nodo en cuestión.

- **¿Cuales son los recorridos de un ABB?**

  Existen seis combinaciones posibles, pero nosotros vamos a estudiar las tres principales:

  - **In-Order (IND):** Accede primero al árbol izquierdo, luego al nodo, y por ultimo al árbol derecho. De esta forma los elementos se recorren de menor a mayor.
  - **Pre-Order (NID):** Accede primero al nodo, luego al árbol izquierdo, por ultimo al árbol derecho. Este recorrido es efectivo a la hora de clonar el árbol, ya que si insertamos elementos en ese orden llegamos al árbol original.
  - **Post-Order (IDN):** Accede primero al árbol izquierdo, luego al árbol derecho, por ultimo al nodo central, de esta forma recorremos primero las hojas de los arboles. Debido a esto, este recorrido es efectivo a la hora de destruir el árbol.

3. Detalles de implementación

Para utilizar este TDA, solamente es necesario compilar con el archivo fuente e incluir el archivo de cabecera en el *main.*

Algunas funciones importantes que merecen ser explicadas de forma detallada. Las funciones auxiliares las considero parte de la función original, por lo que las explico en el mismo apartado.

1. `abb_insertar()`

   Esta función recibe un elemento, recorre el árbol hasta que llega a un nodo ***NULL***, y lo inserta en esa posición.

   Para recorrer el árbol, parte del nodo padre. Si el elemento a insertar es mayor o igual entonces accede a su hijo derecho, de lo contrario accede al hijo izquierdo. Repite esta lógica hasta encontrar el nodo un nodo ***NULL***.

   Como la función es recursiva y devuelve el mismo nodo que la llamo. Entonces utiliza una variable externa para verificar si pudo insertar correctamente o no.

   La complejidad algorítmica de esta función es $O(\log n)$ en el mejor caso, y $O(n)$ en el peor.

2. `abb_quitar()`

   Esta función recibe un elemento y lo elimina del árbol. Para encontrar el elemento a eliminar sigue la misma lógica que el algoritmo anterior.

   - Si el elemento a eliminar es una hoja, simplemente lo elimina.
   - Si el elemento a eliminar tiene un hijo, Lo elimina y conecta el padre del nodo a eliminar con el hijo.

- Si el elemento a eliminar tiene dos hijos, entonces busca el mayor nodo del sub-árbol izquierdo del nodo a eliminar y lo remplaza.

Por los mismos motivos que la función anterior, esta utiliza una variable externa que almacena el elemento quitado.

La complejidad algorítmica de esta función es $O(\log n)$ en el mejor caso, y $O(n)$ en el peor.

3. `abb_buscar()`

   Esta función busca un determinado elemento en el árbol. Para esto usa el algoritmo mencionado anteriormente.

   Parte del nodo padre. Si el elemento buscado es mayor entonces accede a su hijo derecho y repite, de lo contrario accede al hijo izquierdo. Continua hasta encontrar el nodo buscado o llegar a un nodo ***NULL***.

   La complejidad algorítmica de esta función es $O(\log n)$ en el mejor caso, y $O(n)$ en el peor.

4. `abb_destruir()`

   Para destruir el árbol sin problemas, la función hace un recorrido post-order. De esta forma siempre visita hojas, lo que facilita su destrucción.

   La complejidad algorítmica de esta función es $O(n)$

5. `abb_destruir_todo()`

   Esta función es muy similar a la anterior, con la diferencia que aplica un destructor a cada elemento.

   La complejidad algorítmica de esta función es $O(n \cdot f(n))$, siendo $f(n)$ la complejidad de la función destructor.

6. `abb_con_cada_elemento()`

   Esta función recorre el árbol en el orden indicado y aplica una función a cada elemento. Utiliza una variable externa para contar la cantidad de veces que se aplico una función. Además, si alguna de las funciones devuelve ***FALSE***, entonces finaliza la ejecución prematuramente.

   Es una función recursiva, se llama a si misma con el nodo izquierdo y con el nodo derecha, respetando el orden pasado por parámetro.

   Para lograr esto, se guarda referencia a un puntero a ***BOOL*** inicialmente en ***FALSE***. Si en algún momento la función devuelve false, entonces se almacena el valor ***TRUE*** en el puntero, lo que le indica al resto de los llamados recursivos que ya no deben ejecutarse.

   La complejidad algorítmica de esta función es $O(n \cdot f(n))$, siendo $f(n)$ la complejidad de la función que se debe aplicar

7. `agregar_elemento()`

   Esta función recibe dos punteros void. El primero es el elemento que tengo que agregar al vector, El segundo es un vector de punteros void, donde cada puntero es un parámetro.

El primer parámetro es el vector, el segundo parámetro es el máximo del vector, el tercer parámetro es el tope del vector.

Si luego de agregar el elemento en el vector, se alcanzo el máximo, devuelve ***FALSE***.

8. `abb_recorrer()`

   Esta función utiliza la función anterior, llama a ***ABB_CON_CADA_ELEMENTO()*** con una función ***AGREGAR_ELEMENTO()*** que almacene elementos en un vector , pasando por parámetros el máximo y el tope del mismo. Si se alcanza el fin del vector, entonces la función devuelve ***FALSE***. De esta forma, se corta la ejecución.

   La complejidad algorítmica de esta función es $O(n)$

9. `extraer_maximo()`

   Esta función parte de un árbol y accede siempre al nodo de la derecha hasta llegar al final. Luego lo elimina y lo almacena en una variable externa.

   La complejidad algorítmica de esta función es $O(n \cdot f(n))$, siendo $f(n)$ la complejidad de la función que se debe aplicar

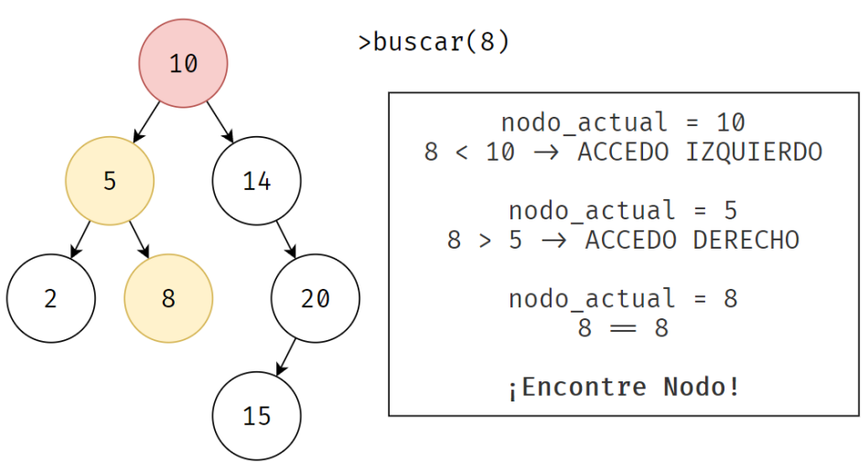
Las funciones restantes tienen complejidad $O(1)$:

- `abb_vacio()`
- `abb_tamanio()`
- `abb_crear()`

4. Diagramas

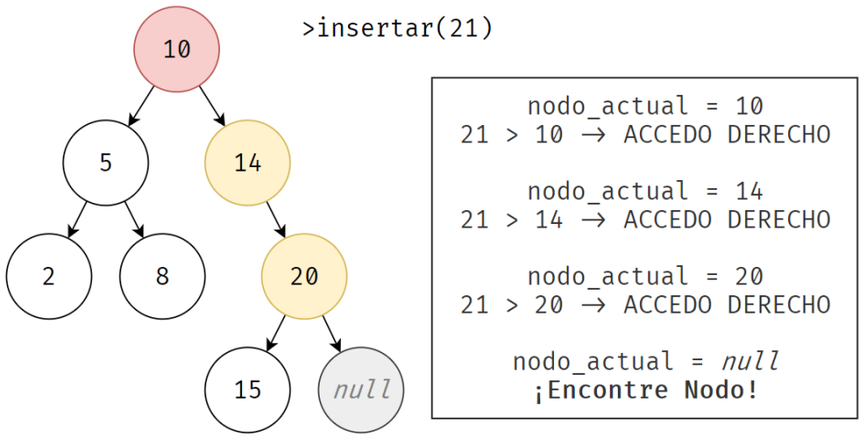
1. **Buscar Elemento**

   Partimos del nodo padre. Como el elemento a buscar **(8)** es menor que el nodo padre **(10)**, Accedemos al nodo izquierdo **(5)**. Luego comparamos **8** con el nodo actual **(5)**. Como es mayor accedemos al derecho. Si ahora comparamos, nodo actual **(8)** es el elemento buscado. Por lo que lo podemos devolver.



1. **Insertar Elemento**

   De forma similar al algoritmo anterior, recorremos el árbol hasta llegar a un nodo ***NULL***. Luego creamos un nodo y lo insertamos a la derecha del el ultimo nodo valido visitado **(20)**.



1. **Quitar Elemento**

   Buscamos el elemento anterior mediante el algoritmo de búsqueda. Como el elemento tiene dos hijos entonces aplicamos el algoritmo de eliminación necesario. Buscamos el mayor elemento del subárbol izquierdo al nodo a eliminar y lo usamos como remplazo del nodo a quitar.