Trabajo Práctico — TDA ABB

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Alumno:
Vainstein Aranguren, Tomás

Número de padrón:
109043

Email:
tvainstein@fi.uba.ar

1. Introducción

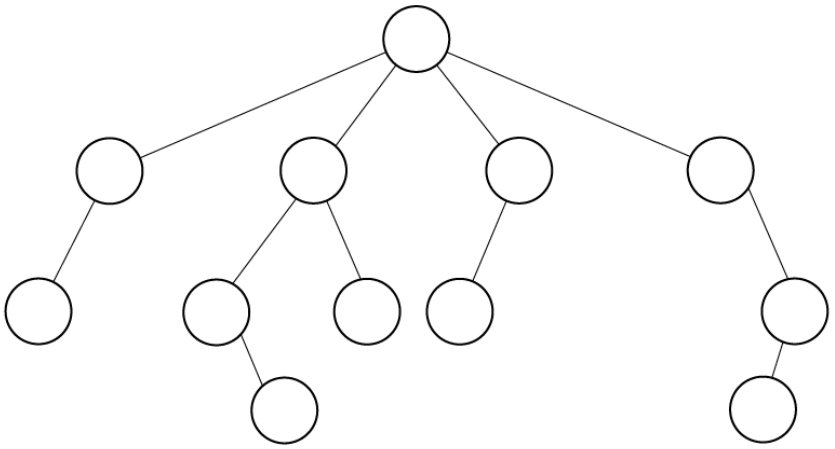
Para este TP se pidió armar el código de un TDA ABB que permita el ingreso y borrado de datos de tipo no determinado, utilizando funciones como insertar, quitar, buscar, recorrer e iterar.
Aparte, se crearon pruebas para facilitar el desarrollo del código utilizando la biblioteca pa2m.h.

2. Teoría

1. Árbol:

   Un árbol N-ario es un conjunto de distintos nodos que almacenan datos de tipo no definido y tiene como característica una cantidad no establecida de estos mismos nodos en unión entre cada uno de ellos, incluyendo la raíz. Sin embargo, debe respetarse la jerarquía del mismo, en donde cada nodo no puede tener unión con otro en su mismo nivel, estableciendo únicamente una unión entre padres (nodos con menor recorrido hasta la raíz) e hijos (nodos en un nivel inferior).
   No hay forma de definir la complejidad de cada una de sus operaciones, pues no hay forma de determinar la cantidad de elementos que puede tener almacenado un árbol de forma genérica.

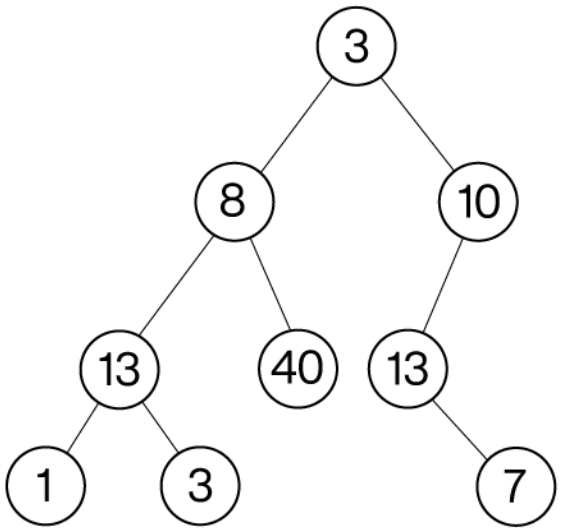
Árbol n-ario



2. Árbol binario:

Un árbol binario, al igual que el árbol genérico, es una colección de nodos que almacenan datos de tipo no definido, excepto que tiene como característica principal que la cantidad de nodos unidos entre padres e hijos no puede ser mayor a dos (por eso mismo se refiere al término binario).

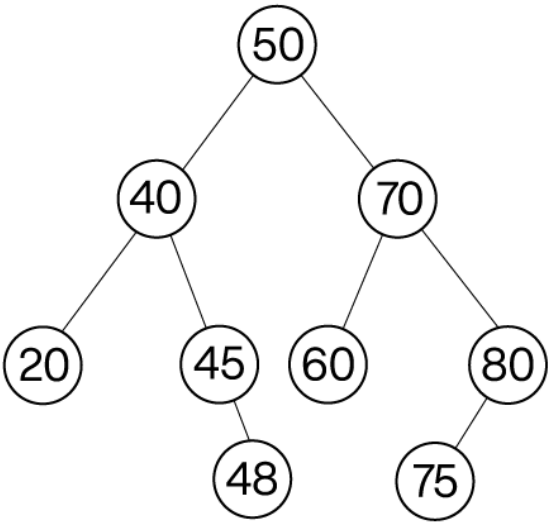
Árbol binario



3. Árbol binario de búsqueda:

Por último, el árbol binario de búsqueda (o ABB), comparte la mismas características que un árbol binario, a cambio de que en este tipo de árbol, los elementos se insertan de forma ordenada.
Se pueden tomar distintas convenciones para ordenar el árbol a medida que se van insertando elementos en el mismo.
En el TDA implementado, se insertaron los elementos menores a la izquierda y los elementos mayores a la derecha, como se puede observar en el ejemplo debajo.
Tener los elementos ordenados de cierta forma, simplifica la complejidad de las distintas operaciones que se pueden realizar con el árbol, sabiendo que se puede obtener un valor menor o mayor dentro de cada mitad del ABB con cada recorrido.
En el mejor de los casos, la complejidad de un ABB va a ser O(n log(n)) y en el peor de los casos O(n), siendo O(log (n)) el promedio.

Árbol binario de búsqueda



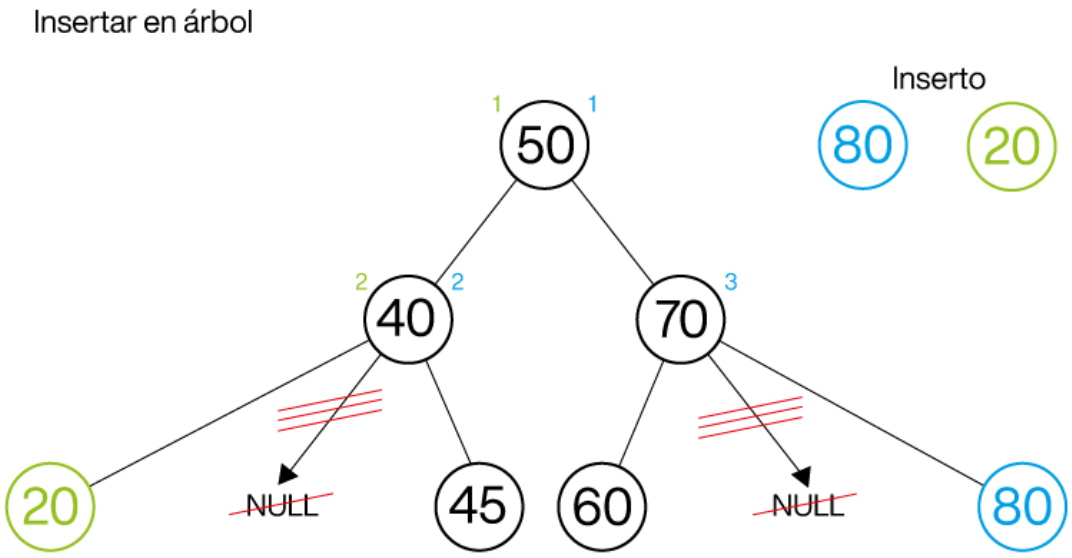
3. Detalles de implementación

Para empezar a programar las funciones del ABB primero hice diagramas para estudiar el comportamiento de cada una de las mismas, y cómo se comportan los elementos insertados o eliminados.
Para crear el ABB utilicé la función *abb_crear que inicializa los campos del árbol en 0, NULL y con el comparador recibido por parámetro. La función devuelve NULL en caso de no recibir el comparador o si hubo un fallo al reservar la memoria.
Luego utilicé la función de *nuevo_nodo_arbol que se encarga de devolver un nodo de árbol con sus campos ya inicializados e insertar el elemento recibido por parámetro. Esta se usa en la función de *abb_insertar, la cual recorre recursivamente el ABB e insertando el nuevo nodo creado en su posición correspondiente utilizando la función de comparador recibida por parámetro para acomodar los nodos menores en las ramas izquierdas y los mayores en las ramas derechas.
Para quitar elementos del árbol, implementé la función *abb_quitar de forma recursiva para ir recorriendo el árbol comenzando por el nodo raíz, hasta que la función comparador determine que el elemento actual y el recibido por parámetro son coincidentes. En el caso de que sea así, se verifica si el nodo con el elemento a borrar tiene un hijo izquierdo, un hijo derecho o ambos hijos. En el caso de que se tenga un sólo hijo, se libera la memoria del nodo y la función devuelve el hijo del eliminado, mientras que sí el nodo a liberar tiene dos hijos, se llama a la función recursiva *extraer_mas_derecho que se encarga de liberar la memoria del nodo y posicionar el predecesor inorden del mismo reemplazando esa posición.
La función recursiva *abb_buscar se encarga de devolver un puntero al elemento que se está buscando en caso de que el mismo se encuentre en el ABB. Se utiliza el comparador para recorrer el ABB, verificando si el elemento que se está buscando es menor (se recorre para la izquierda) o mayor (se recorre para la derecha) a cada nodo

del árbol. En caso de que se encuentre el elemento buscado, el comparador devuelve el valor 0 y la función devuelve el puntero.
Para destruir el ABB, se utilizan las funciones de *abb_destruir_todo y *abb_destruir, en donde se utiliza una función destructora en cada elemento del árbol o nada más se libera el árbol utilizando un destructor NULL, respectivamente.
Para iterar el árbol, se determina el tipo de recorrido de acuerdo al recibido por parámetro y se recorre recursivamente acordemente, siendo inorden (primero el nodo izquierda, luego el nodo raíz y luego el nodo derecha), preorden (nodo raíz, nodo izquierda, nodo derecha) o postorden (nodo izquierda, nodo derecha, nodo raíz). Se deja de iterar cuando la función recibida devuelve false, y se devuelve la cantidad de iteraciones.
Se hace algo similar al recorrer el ABB para insertar en un array, recorriendo recursivamente el árbol de acuerdo al recorrido recibido, guardando el elemento de cada respectivo nodo ordenadamente en el array y aumentando la cantidad de iteraciones, sirviendo como condición de corte que la cantidad de elementos almacenados sea igual al tamaño predeterminado del array. Cuando se termina de recorrer, se devuelven la cantidad de iteraciones.
Para compilar el TDA ABB, se utilizó el archivo makefile adjuntado en la entrega.
Los comandos en cuestión son:
- *make ejemplo*, para compilar el trabajo.
- *make valgrind-ejemplo*, para ejecutar el programa con Valgrind.
- *make pruebas*, para compilar el archivo de pruebas.
- *make valgrind-pruebas*, para ejecutar las pruebas con Valgrind.

3.1. Detalles de abb_insertar

El objetivo de esta función es insertar los elementos ordenadamente en el ABB, siguiendo la convención de insertar los elementos menores del lado izquierdo y los elementos mayores del lado derecho.
La función recibe el árbol y el elemento que se va a insertar (el cuál puede ser de valor NULL). Luego, se comienza a recorrer recursivamente iniciando por el nodo raíz, utilizando la función recursiva *abb_insertar_nodo, en donde se crea el elemento en caso de que el nodo recibido sea NULL y se utiliza el comparador para verificar si el elemento a insertar es menor o igual a 0 (se avanza para la izquierda) o mayor o igual a 0 (se avanza para la derecha). En caso de ser insertado exitosamente el elemento, se devuelve el nodo recibido y la función de abb_insertar devuelve el arbol. Se aumenta la cantidad de elementos en el árbol siempre y cuando la función de *nuevo_nodo_arbol no devuelva NULL.



Los números de cada color correspondiente del diagrama representan el recorrido de comparación que cada nodo de ese mismo color realiza antes de insertarse en el ABB.

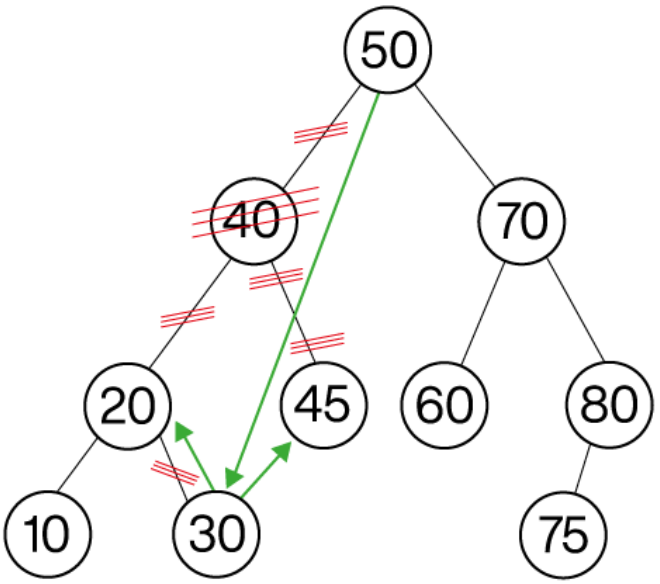
3.2. Detalles de abb_quitar

La función de abb_quitar se encarga de borrar el elemento recibido por parámetro del ABB. Para hacerlo, se utilizan las funciones recursivas de abb_quitar_nodo y extraer_mas_derecho.
En las primeras líneas de la función se verifica que el ABB no esté vacío, pues no se puede eliminar si no hay cantidad de elementos. Luego, se inicializa una variable void* elemento_quitado en NULL, la cual se va a usar para almacenar el valor del elemento quitado del ABB luego de que se encuentre en el mismo.
Para comenzar a recorrer se utiliza el nodo raíz, con el cual se verifica con la función comparador si el elemento a borrar es mayor, menor o igual. En el caso de que sea menor, se avanza para la izquierda, o para la derecha si es mayor.
El proceso se repite hasta que se verifique el valor de el elemento a quitar y el valor del elemento del nodo actual coincidan (comparacion == 0), caso en el que se chequea si el nodo tiene un sólo hijo derecho, un sólo hijo izquierdo, o ambos hijos de valor diferente a NULL.
En caso de que haya un único hijo derecho, se libera la memoria del nodo recibido, se reduce el tamaño del ABB y se reemplaza con ese hijo. Lo mismo sucede en el caso de un hijo izquierdo.
Distinto es el caso en que haya dos hijos, pues se utiliza la función extraer_mas_derecho para reemplazar el nodo borrado con su predecesor inorden, liberar la memoria y reducir el tamaño del árbol.
Finalmente, se verifica si el elemento eliminado es distinto de NULL y se devuelve el puntero. Sino la función devuelve NULL.

Quitar de árbol

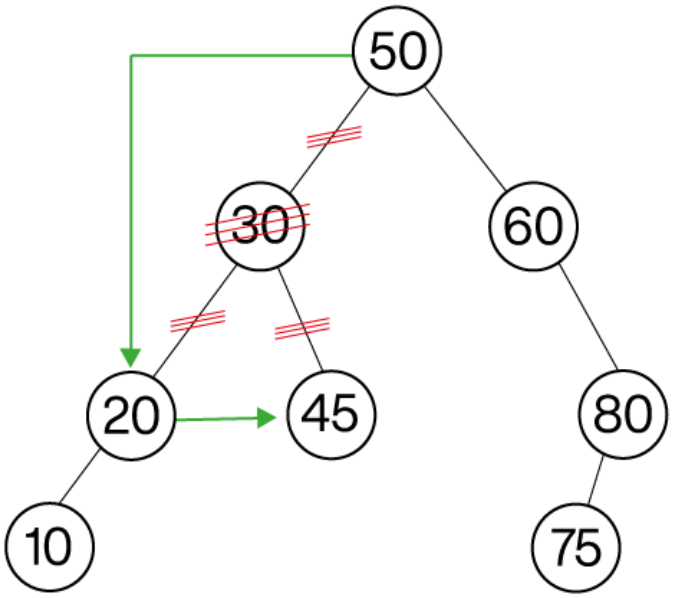
Quito

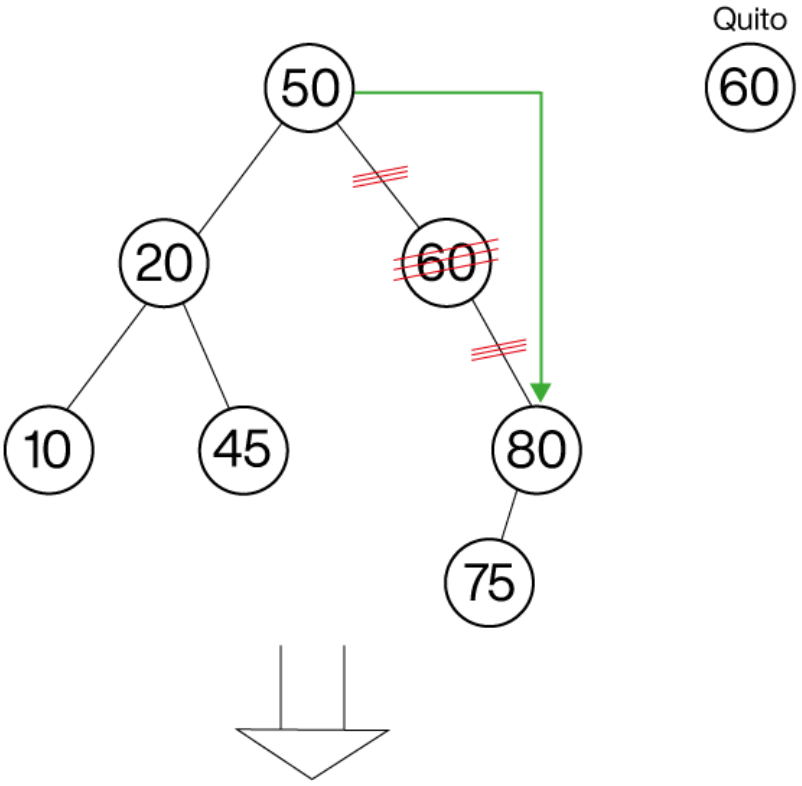
40

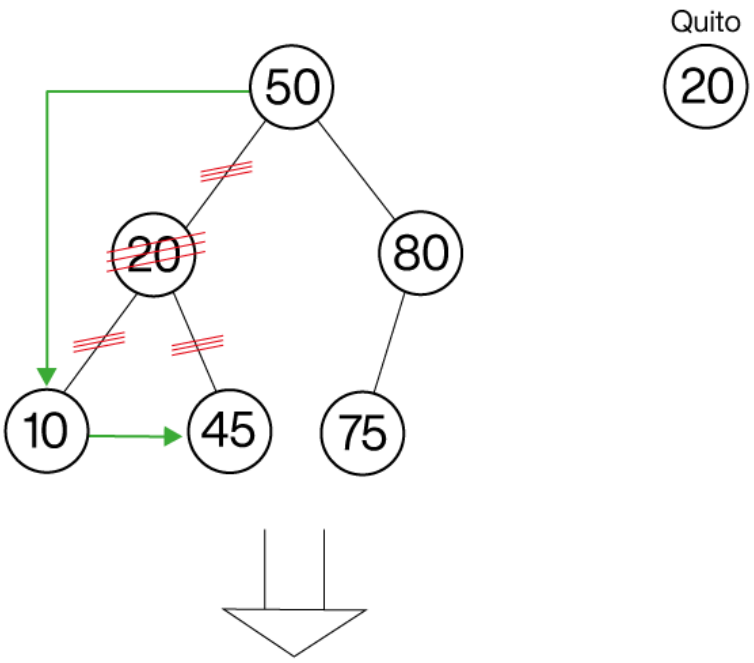


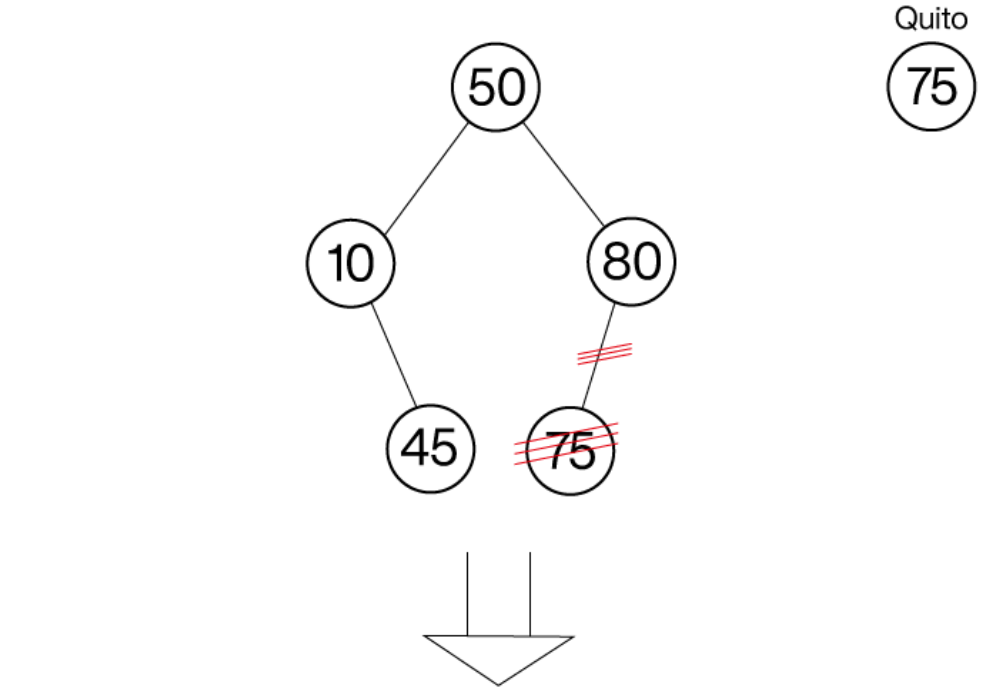
Quito

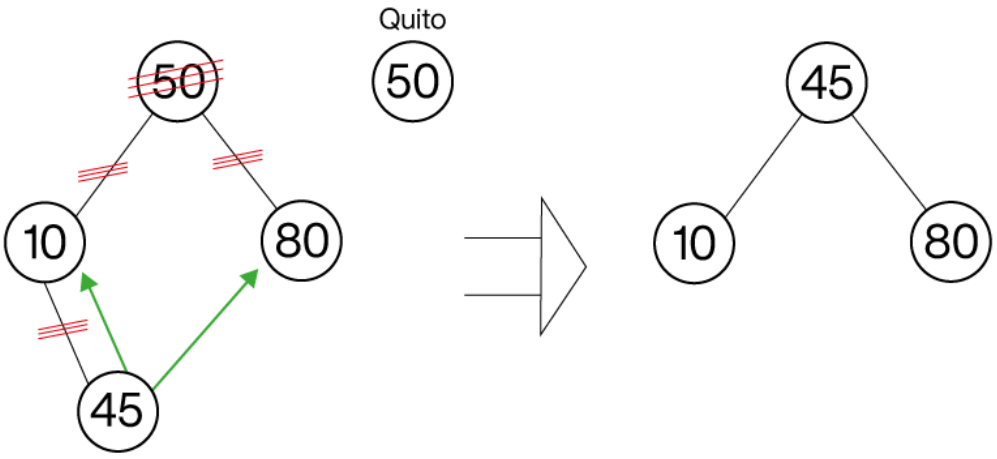
30




Quito
60
50
20
60
10
45
80
75

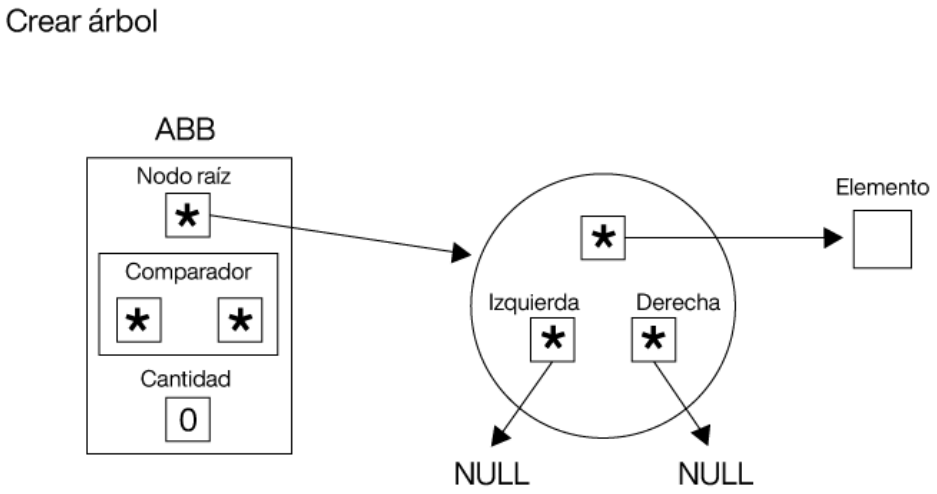

Quito
20
50
20
80
10
45
75



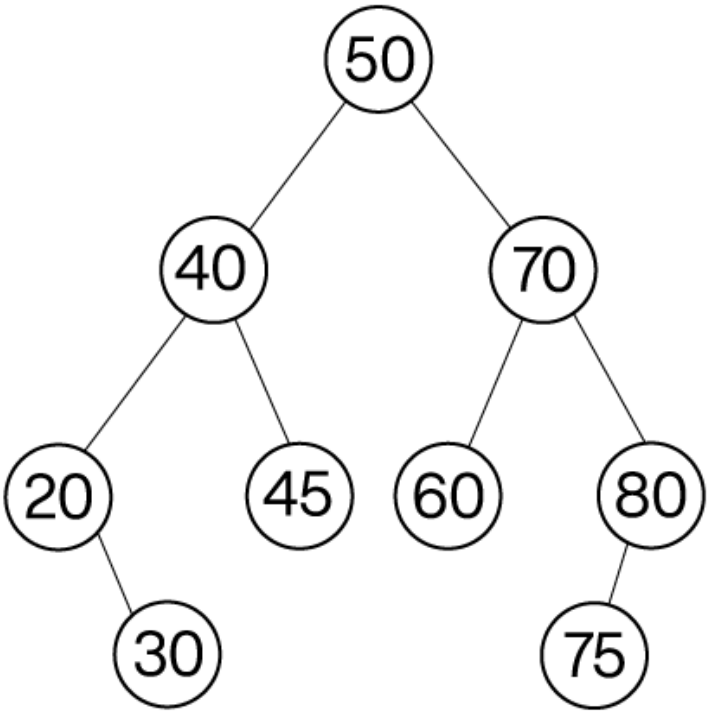


En los diagramas se omite que los nodos sin flechas tienen izquierda y derecha apuntando a NULL para mejorar la legibilidad de los mismos.

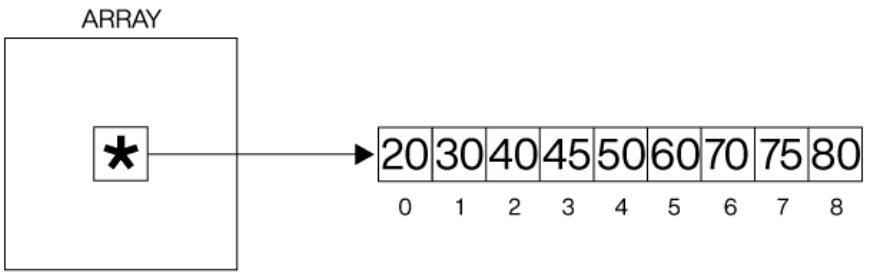
3.3. Detalles de abb_crear



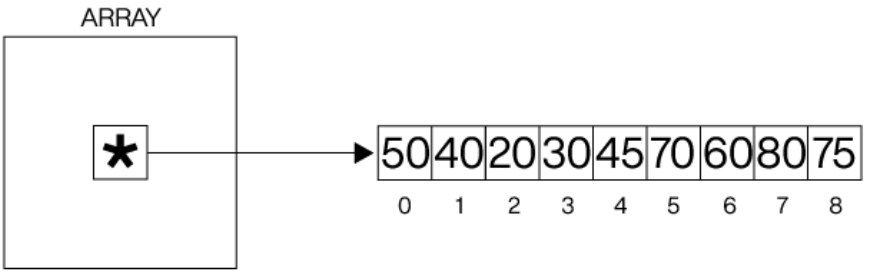
3.4. Detalles de abb_recorrer



INORDEN:



PREORDEN:



POSTORDEN: