Universidad Nacional Autónoma de México

Facultad de Ingeniería
Estructura De Datos y Algoritmos II

Proyecto 1:
Árboles Binarios

Profesor:
Edgar Tista García

Equipo: 8

Integrantes:
Barrios Aguilar Dulce Michelle
Chong Hernández Samuel
Mendoza Hernández Carlos Emiliano

Semestre 2023-1

4 de diciembre de 2022

1. Objetivo

Que el alumno implemente aplicaciones relacionadas con los árboles binarios y que desarrolle sus habilidades de trabajo en equipo y programación orientada a objetos.

2. Introducción

En las estructuras de datos lineales como las pilas o las colas, los datos se estructuran en forma secuencial es decir cada elemento puede ir enlazado al siguiente o al anterior. En las estructuras de datos no lineales o estructuras multi-enlazadas se pueden presentar relaciones más complejas entre los elementos; cada elemento puede ir enlazado a cualquier otro, es decir. puede tener varios sucesores y/o varios predecesores.

Ejemplos de estructuras de datos no lineales son los árboles.

Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación(rama) que impone una estructura jerárquica entre los nodos. Los árboles genealógicos y los organigramas son ejemplos de árboles.

Un árbol puede definirse formalmente de forma recursiva como:

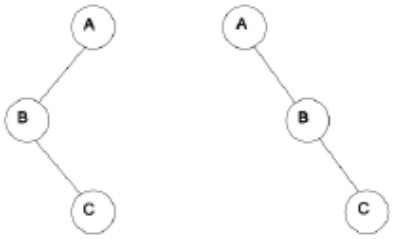
1. Un solo nodo es, por sí mismo un árbol. Ese nodo es también la raíz de dicho árbol.
2. Suponer que n es un nodo y que $A_1, A_2, ..., A_k$ son árboles con raíces $n_1, n_2, ..., n_k$ respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos $n_1, n_2, ..., n_k$. En dicho árbol, n es la raíz y $A_1, A_2, ..., A_k$ son subárboles de la raíz. Los nodos $n_1, n_2, ..., n_k$ reciben el nombre de hijos del nodo n.

La definición implica que cada nodo del árbol es raíz de algún subárbol contenido en el árbol principal.

2.1. Árboles binarios

Un árbol binario es un árbol de grado dos, esto es, cada nodo puede tener dos, uno o ningún hijo.

En los árboles binarios se distingue entre el subárbol izquierdo y el subárbol derecho de cada nodo. De forma que, por ejemplo, los dos siguientes árboles, a pesar de contener la misma información son distintos por la posición de los subárboles:



Se puede definir al árbol binario como un conjunto finito de nodos m nodos ($m \geq 0$), tal que:

1. Si $m = 0$, el árbol está vacío.
2. Si $m > 0$
   a) Existe un nodo raíz.
   b) El resto de los nodos se reparte en dos árboles binarios que se conocen como subárbol izquierdo y subárbol derecho de la raíz.

2.2. Aplicaciones de los árboles binarios

Una aplicación es la representación de otro tipo de árboles. Esto es importante porque resulta más complejo manipular nodos de grado variable (número variable de relaciones) que nodos de grado fijo. Entonces es posible establecer una relación de equivalencia entre cualquier árbol no binario y un árbol binario, es decir, obtener un árbol binario equivalente.

Otra aplicación de los árboles binarios es hallar soluciones a problemas cuyas estructuras son binarias, por ejemplo, las expresiones aritméticas y lógicas.

En este proyecto se profundizará en más aplicaciones de los árboles binarios en sus formas de heap, árbol binario de búsqueda balanceado y árbol de expresiones aritméticas.

3. Desarrollo

3.1. Árbol AVL

Los árboles AVL surgen con el objetivo de mantener la eficiencia en la operación de búsqueda en árboles binarios de búsqueda. La principal característica de éstos es la de realizar re-acomodos o balanceos, después de re-inserciones o eliminaciones de elementos.

Formalmente, se define un árbol balanceado (o AVL) como un árbol binario de búsqueda, en el cual se debe cumplir con la siguiente condición: Para todo nodo T del árbol, la altura de los subárboles izquierdo y derecho no deben diferir en más de una unidad.

$$|H_{R_I} - H_{R_D}| \leq 1$$

Donde H_{R_I} es la altura de la rama o subárbol izquierdo y H_{R_D} es la altura de la rama o subárbol derecho.

Inserción

Al insertar un elemento de un árbol balanceado se deben distinguir los siguientes casos:

1. Las ramas izquierda (RI) y derecha (RD) del árbol tienen la misma altura ($H_{R_I} = H_{R_D}$), por lo tanto:

   1.1 Si se inserta un elemento en RI, entonces H_{R_I} será mayor en una unidad a H_{R_D}
   1.2 Si se inserta un elemento en RD, entonces H_{R_D} será mayor en una unidad a H_{R_I}

2. Las ramas izquierda (RI) y derecha del árbol tienen altura diferente ($H_{R_I} \neq H_{R_D}$) :

   2.1 Si $H_{R_I} < H_{R_D}$:

      2.1.1 Si se inserta un elemento en RI, entonces H_{R_I} será igual a H_{R_D}
      2.1.2 Si se inserta un elemento en RD, entonces se rompe el criterio de equilibrio del árbol y es necesario estructurarlo.

   2.2 Si $H_{R_I} > H_{R_D}$:

      2.2.1 Si se inserta un elemento en RI, entonces se rompe el criterio de equilibrio del árbol y es necesario estructurarlo.
      2.2.2 Si se inserta un elemento en RD, entonces H_{R_D} será igual a H_{R_I}

Factor de equilibrio

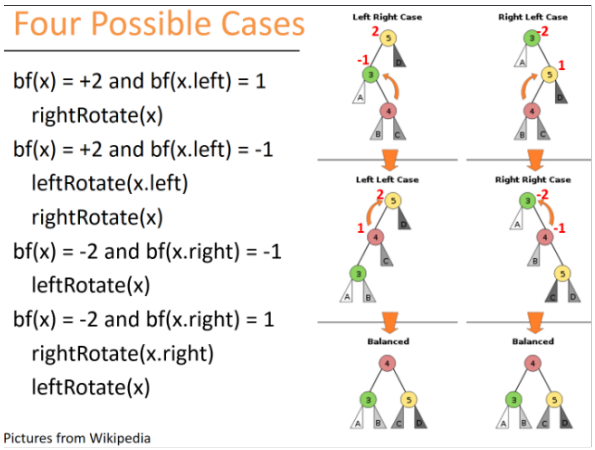
Para determinar si un árbol está balanceado o no, se debe manejar información relativa al equilibrio de cada nodo del árbol. Surge así el concepto del factor de equilibrio de un nodo (FE) que se define como la altura del subárbol derecho menos la altura del subárbol izquierdo.

$$FE = H_{R_I} - H_{R_D}$$

Reestructuración del árbol balanceado

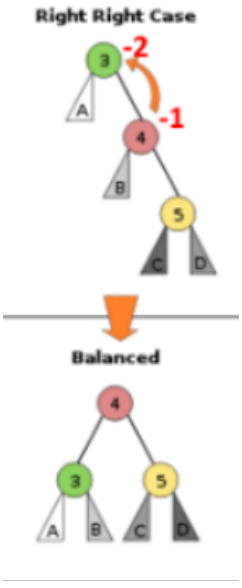
1. Seguir el camino de búsqueda del árbol, hasta localizar el lugar donde hay que insertar el elemento.
2. Calcular su FE, que será 0 en este estado.
3. Regresar por el camino de búsqueda calculando el FE de los distintos nodos visitados.
4. Si en alguno de los nodos se viola el criterio de equilibrio, entonces se debe reestructurar el árbol.
5. El proceso termina al llegar a la raíz, o cuando se realiza la reestructuración, en cuyo caso no es necesario determinar el FE de los nodos restantes.

Reestructurar el árbol significa rotar nodos de este para llevarlo a un estado de equilibrio. La rotación puede ser simple o compuesta. El primer caso involucra dos nodos y el segundo caso afecta a tres. Si la rotación es simple se puede realizar por las ramas derechas o por las ramas izquierdas. Si por otra parte la rotación es compuesta se puede realizar por las ramas derecha e izquierda o por las ramas izquierda y derecha.



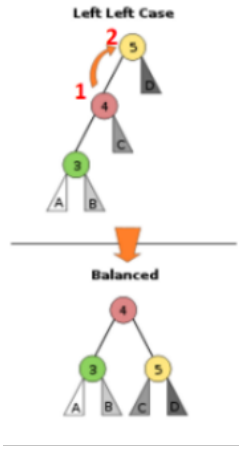
Rotación simple a la izquierda

FE = -2 y FE del hijo izquierdo = -1



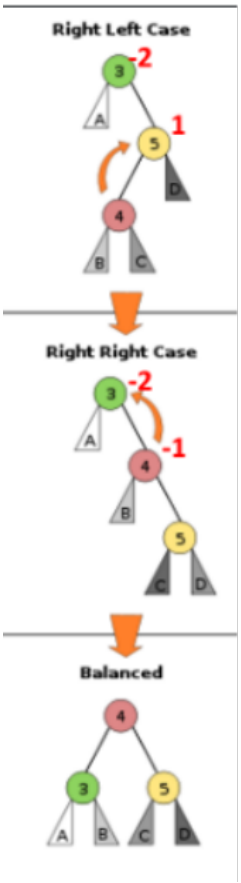
Rotación simple a la derecha

FE = 2 y FE del hijo derecho = 1



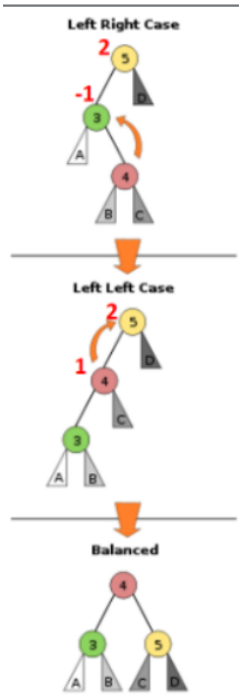
Rotación doble a la izquierda

FE = -2 y FE del hijo derecho = 1



Rotación doble a la derecha

FE = 2 y FE del hijo izquierdo = -1



Eliminación

Esta operación consiste en quitar un nodo del árbol sin violar el principio de equilibrio del árbol AVL. Para la eliminación se tienen los siguientes casos:

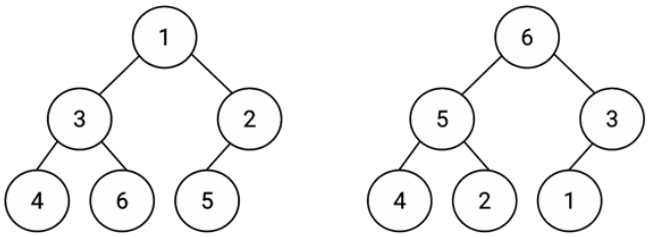
1. Si el elemento a eliminar es una hoja, se elimina.
2. Si el elemento a eliminar tiene un solo hijo, entonces se tiene que sustituir por ese hijo.
3. Si el elemento a eliminar tiene los dos hijos, entonces se tiene que sustituir por el predecesor.

Lo primero que se debe hacer es localizar la posición del nodo a eliminar en el árbol. Se elimina siguiendo los criterios establecidos anteriormente y se regresa por el camino de búsqueda calculando el FE de los nodos visitados. Si en alguno de los nodos se viola el criterio de equilibrio, entonces se debe reestructurar el árbol. El proceso termina cuando se llega a la raíz del árbol. En este proceso se puede llevar a cabo más de una rotación en el camino hacia atrás. Las reestructuraciones y sus casos son similares a la operación de inserción.

Búsqueda
La búsqueda en un árbol AVL es exactamente igual que para un árbol binario de búsqueda.

3.2. Heap

Un heap es un árbol binario completo en donde cada nodo tiene como máximo 2 hijos (0, 1, ó 2 hijos) donde para cada nodo que se pueda revisar, el valor del padre es mayor o menor según sus propiedades.

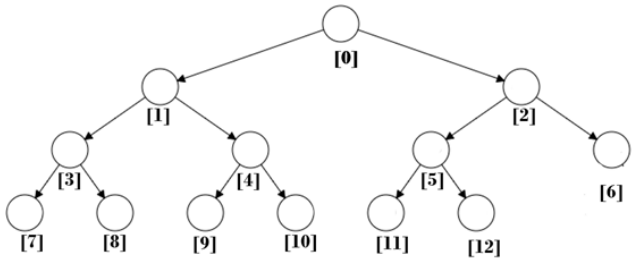


Otra manera de llamar a los heaps es por su traducción al español como "Monticulo", sin embargo, dentro de las estructuras de datos es mejor conocido como heap.

Un montículo cumple con las siguientes propiedades:

- Árbol binario completo (aquel en donde en todos los niveles, excepto el ultimo están completamente íntegros.
- En el último nivel, las claves están lo más a la izquierda posible
- Puede ser fácilmente representado como una matriz

Una manera didáctica de entender el funcionamiento del algoritmo es mediante el entendimiento de cada nodo que conforma el árbol como un arreglo:

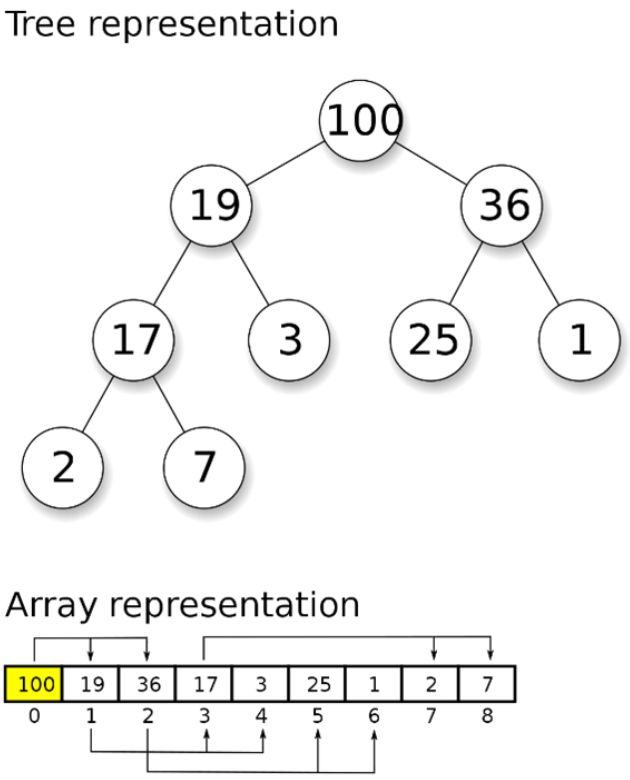


Donde, para cada nodo, existe un arreglo dada su posición dentro del árbol completo, comenzando su recorrido desde el nodo padre (o raíz), donde el primer nodo hijo que se encuentre lo más a la izquierda posible ocupará el siguiente lugar en el arreglo, siguiendo su recorrido hacia el nodo hijo derecho, esto de manera recursiva para los demás niveles del árbol.
Según el valor del padre o raíz, obtenemos la siguiente clasificación:

- MAX HEAP

  Para cada valor del nodo, el valor del padre es mayor.
  Si algún nodo se almacena en la posición "i", entonces su hijo izquierdo se almacena en 2i +1 y el hijo derecho en 2i + 2, en donde obtenemos el siguiente análisis:

  - $A[(i-1)/2]$ representa el nodo padre de A [i].
  - A [(2i +1)] representa el nodo hijo izquierdo de A [i].
  - A [2i + 2] devuelve el nodo hijo derecho de A [i].



- MIN HEAP Para cada valor del nodo, el valor del padre es menor.
  Si un nodo se almacena en la posición 'i', entonces su nodo hijo izquierdo se almacena en la posición 2i + 1 y luego el nodo hijo derecho está en la posición 2i + 2. La posición (i-1) / 2 devuelve su nodo padre.

Los árboles, incluyendo a los heaps, son estructuras de datos no lineales en donde podemos implementar las siguientes operaciones:

- Inserción
- Eliminación
- Búsqueda

Construcción de un heap

La *inserción* de claves para la construcción de un heap sigue los siguientes pasos:

- Insertar el nuevo elemento en la primera posición disponible lo más abajo y a la izquierda posible
- Verificar si el valor agregado es mayor que el nodo padre, en tal caso, intercambiar los elementos, en caso contrario finaliza la inserción.
- Repetir de manera recursiva, volviendo a verificar en la nueva posición.

En la etapa de construcción de un heap se hacen comparaciones/ intercambios con una parte de los elementos de la colección, es decir con un solo subárbol (derecho o izquierdo).

Para la *eliminación* de una clave se realiza lo siguiente:

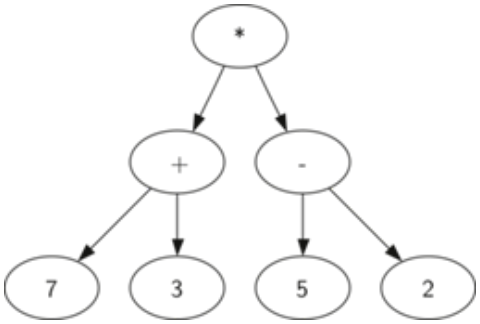
- Se reemplaza la raíz con ele elemento que ocupa la última posición del heap lo más abajo y a la derecha posible

- Verificamos si el valor de la nueva raíz es menor que el valor mas grande entre sus hijos, (si esto ocurre se realiza el intercambio, en caso contario finaliza la ejecución.
- Este proceso se repite recursivamente.

Es importante mencionar que el proceso de eliminación concluye cuando se verifica que el resto de la estructura conserve la integridad de un heap.

3.3. Árbol de expresiones aritmeticas

Los árboles de expresiones aritméticas son árboles binarios encargados de almacenar una expresión matemática, en el que la raíz y cada nodo perteneciente a la raíz del subárbol izquierdo como el subárbol derecho contienen a los operadores de dicha operación; mientras que los nodos hoja contienen a los operandos de la expresión. Antes de pasar a la forma de construcción de un árbol de expresión aritmética, se explicará un concepto fundamental: **Recorrido de árboles.**



Se denomina recorrido de un árbol al proceso que permite visitar todos los nodos del mismo por lo menos una vez.

Existen tres tipos de recorrido para un árbol:

1. Recorrido preOrden. (Notación polaca)
   a) Visitar nodo.
   b) Recorrer subárbol izquierdo.
   c) Recorrer subárbol derecho.
2. Recorrido inOrder.
   a) Recorrer subárbol izquierdo.
   b) Visitar nodo.
   c) Recorrer subárbol derecho.
3. Recorrido postOrden. (Notación polaca inversa)
   a) Recorrer subárbol izquierdo.
   b) Recorrer subárbol derecho.
   c) Visitar nodo.

Este es el algoritmo para realizar por cada tipo de recorrido, el cual se basa en la recursividad. La importancia de conocer estos recorridos es que es a través de ellos podemos construir nuestro árbol y resolver la expresión obtenida. Para construir un árbol binario a partir de una expresión aritmética es importante tomar en cuenta que la expresión a partir de la cual se construirá el árbol se encontrará en su notación infija (inOrder), además de que esta estará contenida en paréntesis con operaciones internas igualmente limitadas por paréntesis.

Construcción del árbol de expresión

Dicho lo anterior, se debe seguir el siguiente algoritmo para la construcción de un árbol de expresión aritmética:

Se recorrerá la expresión introducida analizando carácter por carácter. Matemáticamente los operadores mantienen una jerarquía que les da prioridad sobre otros operadores, esta regla se mantendrá en

la construcción del árbol para obtener el resultado correcto de la expresión introducida. Se utilizarán dos pilas, una para ir almacenando a los nodos operadores y otra para almacenar los subárboles de operaciones que se vayan formando a lo largo de la construcción del árbol. Si el carácter no corresponde a un operador (operando), se apilará en la pila de los subárboles hasta la aparición de uno; si corresponde a un operador se apilará en la pila de operadores, se le asignarán sus hijos izquierdo y derecho los cuales serán los operandos a operar, y posteriormente se apilará en la pila de expresiones como subárbol. Se deben ir ignorando los paréntesis en la formación del árbol, y el último elemento en la pila de subárboles que se desencola es el árbol resultante.

Resolución de la expresión

El propósito de crear este árbol es poder llegar a la resolución de la expresión introducida. Para lograrlo debemos obtener la notación postfija de la expresión, o sea, realizar el recorrido postOrden del árbol creado. El fin de obtener esta notación es aplicar el algoritmo de la notación polaca inversa que nos permitirá obtener el resultado de la expresión.

Dada la peculiar sintaxis de la notación postfija de la expresión en la cual primeramente vienen los operandos y posteriormente los operadores, el algoritmo de notación polaca inversa es similar al de la construcción del árbol. Para este se leerá la expresión en su notación postfija, se utilizará una pila en la que se irán almacenando los resultados de cada operación de acuerdo a la prioridad de los operadores. Este resultado se irá acumulando, hasta que de igual manera el último elemento en desencolar sea el resultado final.

4. Implementación

4.1. Árboles AVL

Clase NodoAVL

Esta clase tiene cuatro atributos, un método constructor y sus setters y getters.

Atributos

- int valor: Contiene el valor o clave del nodo.
- NodoAVL izq: Referencia al hijo izquierdo del nodo.
- NodoAVL der: Referencia al hijo derecho del nodo.
- int h: Contiene la altura del subárbol que se forma tomando este nodo como raíz. Este atributo sirve para facilitar el cálculo del factor de equilibrio de un nodo. Si se consulta este atributo tanto del hijo izquierdo como del hijo derecho de este nodo, simplemente se hace la resta para obtener el factor de equilibrio.

Constructor

NodoAVL (int valor): Inicializa un nodo con el valor especificado. La altura siempre se inicializa con 1 porque cada nuevo nodo se inserta como hoja, cuya altura es 1 al momento de la inserción.

Clase ArbolAVL

Esta clase tiene un atributo, dos constructores, 10 métodos internos (privados) y 4 métodos públicos.

Atributo

NodoAVL root: Referencia al nodo raíz del árbol.

Constructores

- ArbolAVL(): Inicializa un árbol vacío.
- ArbolAVL(int valor): Inicializa un árbol cuya raíz contiene el valor especificado.

Métodos

- public void vaciar(): Elimina la referencia de la raíz, perdiendo así las referencias de todos los nodos. Se utiliza para eliminar el árbol actual y permitir la creación de uno nuevo
- public void breadthFirst(): Realiza el recorrido BFS del árbol. Imprime cada nodo conforme se visita en el recorrido. Esta función es utilizada como manera de imprimir el árbol. Se apoya de la función privada visit.
- private void visit(NodoAVL n): Imprime el valor del nodo que recibe como parámetro. Este método interno se utiliza como manera de visitar un nodo en el recorrido BFS.
- public void buscar(int clave): Realiza la búsqueda de la clave especificada. Imprime un mensaje indicando si la clave buscada está o no está en el árbol. Este método funciona como una máscara del método recursivo privado buscarUtil.
- private buscarUtil (NodoAVL actual, int clave): Este método hace un recorrido recursivo del árbol, iniciando por la raíz. Funciona de la misma manera que la búsqueda para un árbol binario de búsqueda.

- public void insertar(int valor): Realiza la inserción de un nodo con el valor especificado, manteniendo las condiciones de equilibrio. Este método funciona como una máscara del método recursivo privado insertarUtil.
- private void insertarUtil(NodoAVL actual, int valor): Este método recursivo inicia sus llamados desde la raíz, sigue los siguientes pasos:

  1. Se realiza la inserción al igual que se hace para un árbol de búsqueda binario:

     1.1 Si el nodo especificado es nulo, se crea un nuevo nodo con el valor especificado y es regresado. Esto conforma el caso base del método recursivo. Tendremos que el nodo especificado es nulo cuando se inserte la raíz, o cuando se desee insertar un nuevo nodo y se haya encontrado su lugar en el árbol.

     1.2 Si el valor especificado es menor al valor del nodo actual (dentro de las llamadas recursivas), nos pasamos al hijo izquierdo del nodo actual y se hace el llamado recursivo de la función.

     1.3 Si el valor especificado es mayor al valor del nodo actual, nos pasamos al hijo derecho del nodo actual y se hace el llamado recursivo de la función.

     1.4 Si el valor especificado es igual al valor del nodo actual (esto es, que ya existe en el árbol), imprime un mensaje alertando al usuario que ese valor ya existe.

  2. Una vez colocado en el árbol el nuevo nodo, se empieza a salir de los llamados recursivos. Se recorre desde el padre del nodo insertado, subiendo hasta la raíz. Para cada nodo recorrido se actualiza su altura. Para actualizar la altura, se utiliza el método privado mayor y el método privado getAltura. La altura actualizada del nodo actual es la altura mayor de sus dos hijos más uno (porque en esta implementación se empieza a contar la altura desde 1).
  3. Se calcula el factor de equilibrio del nodo actual (siguiendo la idea de que estamos saliendo de las llamadas recursivas y recorriendo los nodos desde el padre del nodo insertado hasta la raíz). Se utiliza como apoyo el método privado getFE.
  4. Se revisa la condición de equilibrio del nodo actual. Se puede presentar uno de los siguientes casos:

     4.1 El FE del nodo actual es 2 y el FE de su hijo izquierdo es 1: se hace la rotación simple a la derecha.

     4.2 El FE del nodo actual es -2 y el FE de su hijo izquierdo es -1: se hace la rotación simple a la izquierda

     4.3 El FE del nodo actual es 2 y el FE de su hijo izquierdo es -1: se hace la rotación doble a la derecha.

     4.4 El FE del nodo actual es -2 y el FE de su hijo izquierdo es 1: se hace la rotación doble a la izquierda.

  5. La inserción finaliza cuando el nodo actual ha regresado hasta la raíz (fin de las llamadas recursivas).

- private int mayor(int a, int b): Devuelve el número mayor entre dos números especificados. Concretamente se utiliza para encontrar la mayor de las alturas de los hijos de un nodo.
- private getAltura(NodoAVL actual): Funciona para obtener la altura de un nodo, validando los casos en los que el nodo especificado es 0 (para evitar el NullPointerException).

- private int getFE(NodoAVL actual): Retorna el resultado de obtener la fórmula $FE = H_{R_I} - H_{R_D}$ aplicada al nodo actual.
- private NodoAVL rotacionD(NodoAVL actual): Se supone el caso donde el FE del nodo actual es 2, y el FE de su hijo izquierdo es 1. Para esta rotación se realiza lo siguiente:
  1. Se crea una nueva referencia para el hijo izquierdo del nodo actual, que se colocará como "nueva raíz" de este subárbol.
  2. Se guardar en un nodo temporal al hijo derecho de la nueva raíz
  3. Ahora se realiza la rotación, se hace el intercambio de la nueva raíz con el nodo actual: el nodo actual pasa a la posición del hijo derecho de la nueva raíz, y la referencia temporal pasa a ser el hijo izquierdo del nodo actual
  4. Se actualizan las alturas tanto del nodo actual como del nuevo.
  5. Se retorna la nueva raíz del subárbol.
- private NodoAVL rotacionI(NodoAVL actual): Se supone el caso donde el FE del nodo actual es -2, y el FE de su hijo derecho es -1. Para esta rotación se realiza lo siguiente:
  1. Se crea una nueva referencia para el hijo derecho del nodo actual, que se colocará como "nueva raíz" de este subárbol.
  2. Se guardara en un nodo temporal al hijo izquierdo de la nueva raíz.
  3. Ahora se realiza la rotación, se hace el intercambio de la nueva raíz con el nodo actual: el nodo actual pasa a la posición del hijo izquierdo de la nueva raíz, y la referencia temporal pasa a ser el hijo derecho del nodo actual.
  4. Se actualizan las alturas tanto del nodo actual como del nuevo.
  5. Se retorna la nueva raíz del subárbol.
- public void eliminar(int clave): Realiza la eliminación de un nodo con la clave especificada (se asume que la clave existe en el árbol), manteniendo las condiciones de equilibrio. Este método funciona como una máscara del método recursivo privado eliminarUtil.
- public NodoAVL eliminarUtil(NodoAVL actual, int clave): Este método recursivo inicia sus llamados desde la raíz y sigue los siguientes pasos:
  1. Se empieza la eliminación al igual que se hace para un árbol de búsqueda binario:

     1.1 Si el nodo especificado es nulo, se regresa como valor de retorno. Esto conforma el caso base del método recursivo (recordando que el árbol se reconstruye al llegar a la última llamada recursiva y regresar el recorrido de los nodos al ir saliendo de los llamados). Tendremos que el nodo especificado es nulo cuando el árbol está vacío, o cuando se haya recorrido todo el árbol y no se haya encontrado el valor (en este caso el árbol se reconstruirá tal y como estaba).

     1.2 Si la clave especificada es menor a la clave del nodo actual (dentro de las llamadas recursivas), nos pasamos al hijo izquierdo del nodo actual y se hace el llamado recursivo de la función

     1.3 Si la clave especificada es mayor a la clave del nodo actual, nos pasamos al hijo derecho del nodo actual y se hace el llamado recursivo de la función.

     1.4 Si el valor especificado es igual al valor del nodo actual (esto es, que ya se encontró el nodo a eliminar), se verifica cuál de los siguientes casos se presenta:

     1.4.1 El nodo a eliminar no tiene hijos (es una hoja). Se elimina directamente el nodo. Esto hace que terminen las llamadas recursivas

1.4.2 El nodo a eliminar tiene solo un hijo (izquierdo o derecho). Si tiene el hijo izquierdo, se crea un nodo temporal para guardar la referencia del hijo (subárbol) derecho. Caso contrario, se guarda en el nodo temporal la referencia del hijo (subárbol) izquierdo. El nodo actual pasa a ser la referencia de su hijo que se guardo en el nodo temporal.

1.4.3 El nodo a eliminar tiene sus dos hijos. Al igual que en los árboles binarios de búsqueda, se busca al predecesor del nodo (nodo más a la derecha del subárbol izquierdo). Para buscar al predecesor se utiliza el método privado getPredecesor. Luego, se copia la clave del predecesor en el nodo actual y se hace un llamado recursivo de eliminación en el hijo izquierdo del nodo actual para eliminar al predecesor.

2. Una vez se ha eliminado el nodo, se empieza a salir de los llamados recursivos. Se recorre desde el padre del nodo eliminado, subiendo hasta la raíz. Para cada nodo recorrido se actualiza su altura. Para actualizar la altura, se utiliza el método privado mayor y el método privado getAltura. La altura actualizada del nodo actual es la altura mayor de sus dos hijos más uno (porque en esta implementación se empieza a contar la altura desde 1).
3. Se calcula el factor de equilibrio del nodo actual (siguiendo la idea de que estamos saliendo de las llamadas recursivas y recorriendo los nodos desde el padre del nodo insertado hasta la raíz). Se utiliza como apoyo el método privado getFE.
4. Se revisa la condición de equilibrio del nodo actual. Los casos que se presentan son los mismos que para la inserción y se resuelven de la misma manera.
5. La eliminación termina cuando se ha regresado hasta la raíz en las llamadas recursivas.

- private NodoAVL getPredecesor(NodoAVL nodo): Recorre iterativamente los hijos derechos del nodo especficado (mientras los tenga). Regresa el nodo más a la derecha encontrado.

4.2. Heap

Clase Heap

Esta clase define los métodos necesarios para la creación de un MAX HEAP a partir de los elementos o claves ingresados por el usuario.

Atributos

- Nodo(). Para definir el valor que le será asignado a la variable.
- Nodo root. Para asignarle la raíz al árbol según el algoritmo.
- int val. Referencia al número que el usuario ingrese.

Constructores

- publicHeap().Constructor default del Heap.
- Heap(int val). Constructor que define un nuevo nodo según el valor asignado.

Heap(Nodo root). Constructor que define la instancia de la raíz del heap.
Métodos

- printHeap(). Se encarga de imprimir el heap dadas las claves ordenadas de acuerdo al método heapify(que se encarga de mantener las propiedades de un max heap, donde para todas las claves que se revisen, el valor del padre debe ser siempre mayor) también definido en esta clase.
- Heapify(). Se encarga de crear esta estructura de datos no lineal en donde reordenar las claves ingresadas para mantener las propiedades de una max heap, donde definimos al nodo izquierdo y derecho según su fórmula equivalente para realizar de manera recursiva el reordenamiento de la clave ingresada y asignarla al subárbol izquierdo o derecho.
    1. El algoritmo de este método analiza el primer hijo derecho cuyo índice viene dado por n/2 - 1
    2. Establece la clave que se está analizando como largest
    3. Si el hijo izquierdo es mayor que su padre entonces establece esta clave como mayor
    4. Si el hijo derecho es mayor que la clave que se está analizando entonces definimos a la clave que se encuentra en el hijo derecho como largest
    5. Se hace el intercambio con largest con el padre
    6. Este proceso se realiza de manera recursiva para todas las claves ingresadas.
- buildHeap(). Recorre la lista en la que los elementos se almacenan conforme se van ingresando mientras que los va ordenando de forma en que la raíz de cada heap sea mayor a sus hijos.
- Delete(). Encargado de la eliminación de una clave, ya sea raíz, hijo derecho o hijo izquierdo de acuerdo con las reglas de eliminación de un heap, donde para la eliminación de la raíz se reemplaza con el elemento que ocupa la última posición del heap lo más abajo y a la derecha posible, verificando si el valor de la nueva raíz es menor que el valor más grande entre sus hijos, en tal caso, se realiza un intercambio, repitiendo este proceso recursivo.
    1. Si el nodo a ser eliminado es una hoja, lo elimina directamente
    2. Si el nodo a eliminar es un hijo derecho o un hijo izquierdo, realiza un intercambio con ayuda del método heapify() con la clave que ocupa la última posición del heap lo más abajo y a la derecha posible,
- Insert(). Método que realiza la inserción de los elementos de acuerdo a las reglas de ordenamiento de un MAX Heap:

1. Definimos una variable que nos devuelva el tamaño de la lista.
2. Si es el primer elemento a insertar lo añade a la primera posición disponible
   - Si no se cumple, entonces inserta la clave ingresada en la primera posición disponible lo más abajo y a la izquierda posible
   - Se verifica si la clave es mayor que el padre, en tal caso intercambia los elementos haciendo una llamada al método que es el encargado de realizar laa asignaciónm.
3. Este proceso se realiza de manera recursiva

- defNodo(). Método que define a los nodos derecho e izquierdo, así como la raíz del heap, de acuerdo a las características de un MAX Heap, donde en una nueva lista asignamos a los hijos izquierdos y derechos de acuerdo con las operaciones necesarias para definir tanto hijo izquierdo como derecho en su inserción de claves.

  1. Inicializamos los nodos derecho e izquierdo respectivamente con el indice que tendrá el nodo, left (2*i) + 1 haciendo referencia al subárbol izquierdo, right (2*i) + 2 haciendo referencia al subárbol derecho.
  2. Le asignamos el índice que tendrán los hijos derecho e izquierdo
  3. Le asignamos el índice correspondiente a la raíz.

Clase Nodo

Esta clase define los getters y setters de los nodos, así como los valores de retorno necesarios para la ejecución exitosa del programa.

Atributos

- intValor(). Variable que le asignará un valor a un nuevo nodo
- Nodo izq : Inicializa el valor del nodo izquierdo en null.
- Nodo der : Inicializa el valor del nodo izquierdo en null.

Constructores

- Nodo(Nodo nodo): constructor default de la clase.
- Nodo(int data): inicializa los valores del nodo derecho e izquierdo en null.
- Nodo(int data, Nodo lt, Nodo rt): inicializa las variables a utilizar en null.

Métodos

- setIzq(Nodo izq). Método que define la instancia izquierda en el heap (Nodo izquierdo).
- setIzq(Nodo izq). Método que define la instancia derec en el heap (Nodo derecho)

4.3. Árbol de expresión aritmética

Conociendo los requerimientos para construir un árbol de expresión aritmética, primeramente, se dividió el caso en tres casos:

- **Mejor caso.** El usuario introduce una expresión aritmética correcta (operadores de un solo dígito, contenida en paréntesis con operaciones internas igualmente contenidas en paréntesis). Ejemplo:

  ((2*4+5)/(9-2)*(9+3+2))

- **Caso Promedio.** El usuario introduce un carácter erróneo. Ejemplo:

  ([9+3)/(2+1))

- **Peor caso.** El usuario introduce cualquier cosa excepto una expresión aritmética correcta. Ejemplo:

  Akd8)03l

Con base en estos casos se creará una excepción que arrojarán los métodos implementados.

A continuación, se realiza una descripción de la forma en la que se implementaron las clases del programa, describiendo sus métodos y atributos.

Clase NodoExpresion.

Es la clase encargada de construir los nodos del árbol.

Atributos

- **Object** valor: Es el valor que contendrá el nodo. De tipo Object para poder asignar cualquier referencia de objeto al dato.
- **NodoExpresion** izq: Corresponde al hijo izquierdo del nodo.
- **NodoExpresion** der: Corresponde al hijo derecho del nodo.

Constructores

- **NodoExpresion()** Constructor por defecto de un Nodo, asignando null a los hijos del nodo.
- **NodoExpresion** (Object Valor) Constructor para crear un Nodo asignando el dato que contendrá.

Métodos

- public void setIzq(NodoExpresion izq). Asigna el hijo izquierdo de un nodo.
- public void setDer(NodoExpresion der). Asigna el hijo derecho de un nodo.

Clase ArbolExpresión.

Es la clase encargada de crear el árbol de expresión, así como de resolver la expresión ingresada.

Atributos

- **NodoExpresion** root. Corresponde al nodo raíz del árbol

Constructores

- **ArbolExpresion()** Es el método constructor vacío por defecto.
- **ArbolExpresion** (String cadena) Método que a partir de una expresión ingresada creará el árbol de expresión correspondiente.

Métodos

- public void BFS(). Se encarga de realizar el recorrido por capas del árbol, visitando cada nodo por lo menos una vez y mostrándolo.
- static void visitar(NodoExpresion n). Se encarga de marcar y mostrar el nodo visitado.
- private static int Prioridad(char operador). Como se vio anteriormente, los operadores de una expresión aritmética tiene cierta prioridad unos con otros para ir realizando las operaciones en orden. Este método se encarga de asignar dicha prioridad dependiendo el operador. Prioridad ascendente: +, -, /, *.
- private static boolean Operador(char carácter). Método encargado de verificar si el carácter analizado es un operador.
- static NodoExpresion crearArbol(String expresion). Es el método encargado de realizar el proceso de conversión de la expresión a Árbol de expresión mediante el algoritmo explicado con anterioridad. Este método arroja una excepción en el caso de que la cadena contenga un carácter fuera del rango permitido para la expresión aritmética.

  1. Se lee la cadena y se analiza carácter por carácter.
  2. Se crea un nuevo nodo por cada carácter.
     a) Se verifica si el carácter se encuentra dentro del rango. De lo contrario se lanza un mensaje de error.
     b) Se verifica si el carácter es o no es un operador
  3. Se apilan los nodos
     a) Si el carácter corresponde a un paréntesis de apertura se apila en la pila de operadores.
     b) Si el carácter corresponde a un operando, se apila en la pila de expresiones.
     c) Si nos encontramos con un operador se comprueba si su prioridad es menor a la del último operador en la pila, si es así, a este operador se le asignan sus hijos desapilando los dos últimos operandos de la pila de expresiones. Hecho esto, se apila el nodo resultante.
     d) El proceso se repite para los demás operadores.
     e) Al encontrar un paréntesis que cierra la expresión se termina el proceso y se desapilan los paréntesis sin asignarse a alguna variable.
  4. Se retorna el nodo resultante que corresponde al árbol creado.

- static float resolver(String cadena). Tras obtener la notación postfija de la expresión, este método será el encargado de aplicar el algoritmo de notación polaca inversa con una pila.

  1. Se lee la cadena y analiza carácter por carácter.
     a) Se verifica si corresponde a un operador u operando por medio de su prioridad. Si esta es diferente de 0 indica que es un operador.
  2. Se apilan los caracteres.
     a) Si corresponde a un operando, se apila su referencia menos su valor ASCII para obtener su valor flotante.
     b) Si corresponde a un operador se desapilan dos elementos de la pila y de acuerdo al operador por medio del método operar se realiza la operación correspondiente, el resultado se apila.
     c) Se repite el proceso con los resultados que se encuentren en la pila.

3. Se retira el último elemento de la pila que corresponde al resultado final.

- private static float operar(char operador, float operando1, float operando2). Se encarga de comprobar qué operador se esta recibiendo para así aplicarlo.
- public static void postOrden(NodoExpresion raiz). Método encargado de realizar el recorrido postOrden del árbol.
- public static String posOrdenString(NodoExpresion raiz, String c). El método se encarga de regresar como cadena de caracteres al recorrido postOrden para así poder ser utilizado en la resolución de la expresión.

Clase excepcionCaracter.

Clase que hereda a la clase "Exception", encargada de controlar la excepción en el caso de que se meta un carácter incorrecto.

Constructores

- **excepcionCaracter()** Método constructor por defecto.
- **excepcionCaracter()** Método constructor del que se hereda atributo de la clase Exception.

5. Conclusiones

Barrios Aguilar Dulce Michelle

Entender inicialmente la metodología que sigue un conjunto finito de elementos puede llegar a complicarse al relacionarlo con la creación de una estructura no lineal como son los árboles, donde los nodos son precisamente este conjunto finito en el que puede ser relacionado de distintas maneras según las características del tipo de árbol en el que se implemente. El hecho de comprender la estructura que se sigue para la construcción de un árbol binario y además el trabajar con estas 3 implementaciones en las que analizamos diferencias muy marcadas entre ellas, expande el panorama hacia las diversas aplicaciones en las que podríamos llegar a utilizarlas para resolver diversos problemas relacionados con el manejo y la jerarquía de información en los que podemos obtener una representación de dicha información según las necesidades y requerimientos que se soliciten con un tiempo de acceso relativamente bajo. .

Chong Hernández Samuel
En clase se vieron variantes de los árboles binarios, sean estas el árbol binario de búsqueda, heaps y de expresión; con este proyecto y la tarea respecto a árboles nos podemos dar cuenta de la amplia variedad en cuanto a las variantes de árboles binarios y sus múltiples aplicaciones, indicándonos que esta estructura de datos es fundamental. Esto es importante, ya que como ingenieros en computación tendremos que idear soluciones para los problemas que se nos presenten y dada la estructura jerárquica de los árboles binarios es necesario que sepamos implementarlos y manejarlos para formular dichas soluciones. Puedo concluir que se cumple con el objetivo, ya que no solamente pudimos implementar variantes de árboles binarios para la solución del proyecto, sino que pudimos desarrollar nuestro trabajo en equipo a través del apoyo proporcionado por cada uno de los integrantes para sacar adelante cada aspecto requerido.

Mendoza Hernández Carlos Emiliano
Las estructuras de datos no lineales permiten aplicaciones más complejas que las no lineales. Una de estas estructuras, los árboles, en su forma binaria (con máximo dos hijos) da pie a diversas variaciones entre las que se destacan el heap, el árbol binario de búsqueda, el árbol AVL, el árbol de expresiones aritméticas, entre otros. Estos árboles se caracterizan por ser estructuras recursivas, y se crearon con el propósito de hacer operaciones en un tiempo razonable. En esta práctica fue posible hacer tres implementaciones básicas de estos árboles, sin embargo, sus aplicaciones en el mundo real son bastante amplias. Considero que si bien puede ser complejo implementar sus algoritmos, el tiempo en el que realizan sus operaciones hace que valga la pena su uso. Al mismo tiempo, se realizaron las implementaciones con un enfoque orientado a objetos. Por estas razones, considero que se cumplieron los objetivos del proyecto.

6. Referencias

1. Cairó, O., Guardati, S. (2010). Estructuras de datos. (3a. ed.) McGrawHill.
2. Edgar Tista García [Prof. Edgar Tista]. (2020 septiembre 29). Heapsort. [Video]. Recuperado de https://youtu.be/uFTLwNe3P1o
3. Edgar Tista García [Prof. Edgar Tista]. (2022 Febrero 13). ¿Qué es un heap?. [Video]. Recuperado de https://youtu.be/Gtwg5eus5Gc
4. Team, D. S. (2022, 2 junio). Implementación del Heap Mínimo y del Heap Máximo – Cómo funciona todo. DATA SCIENCE. https://datascience.eu/es/programacion/implementacion-del-heap-minimo-y-del-heap-maximo-como-funciona-todo/
5. Alonso, J. A. (2017, Enero 27). Notación Polaca Inversa. Exercitium. Recuperado diciembre 4, 2022, de https://www.glc.us.es/ jalonso/exercitium/notacion-polaca-inversa/
6. Árboles de expresion - ISC. Mendez-PORTAFOLIO. Google Sites: Sign-in. (n.d.). Recuperado Diciembre 4, 2022, de https://sites.google.com/site/iscmendezportafolio/arboles-de-expresion
7. Construcción de un árbol de expresión. (n.d.). Recuperado Diciembre 4, 2022, de https://www.techiedelight.com/es/ex tree/