	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: TISTA GARCÍA EDGAR

Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS II

Grupo: 5

Número de Práctica(s): Guía práctica de estudio 9: Árboles Parte II

Integrante(s): MURRIETA VILLEGAS ALFONSO |VALDESPINO MENDIETA JOAQUÍN

Núm. De Equipo de cómputo empleado: 38

Semestre : 2019 - 1

Fecha de entrega: 16 DE OCTUBRE DE 2018

Observaciones:

CALIFICACIÓN: _____

ÁRBOLES. PARTE II

OBJETIVOS DE LA PRÁCTICA

- El estudiante conocerá e identificará las características de los árboles-B.

INTRODUCCIÓN

En los sistemas computacionales se cuenta con dos sistemas de almacenamiento en dos niveles, el almacenamiento en memoria principal y el almacenamiento secundario (como en discos magnéticos u otros periféricos). En los almacenamientos en memoria secundaria el costo es más significativo y viene dado por el acceso a este tipo de memoria, muy superior al tiempo necesario para acceder a cualquier posición de memoria principal. Entonces al tratarse de un dispositivo periférico y, por tanto, de acceso lento, resulta primordial minimizar en lo posible el número de accesos.

Para realizar un acceso a un disco (para lectura / escritura) se requiere de todo un proceso de movimientos mecánicos que toman un determinado tiempo, por lo que para amortizar el tiempo gastado en esperar por todos esos movimientos la información es dividida en páginas del mismo tamaño en bits que se encuentran en las llamadas pistas del disco, así cada lectura y/o escritura al disco es de una o más páginas. Los árboles-B (en inglés B-Tree), se utilizan cuando la cantidad de datos que se está trabajando es demasiado grande que no cabe toda en memoria principal. Así que, son árboles de búsqueda balanceados diseñados para trabajar sobre discos magnéticos u otros accesos o dispositivos de almacenamiento secundario.

En los algoritmos que utilizan árboles-B se copian las páginas necesarias desde el disco a memoria principal y una vez modificadas las escriben de regreso al disco. El tiempo de ejecución del algoritmo que utilice un árbol-B depende principalmente del número de lecturas y escrituras al disco, por lo que se requiere que estas operaciones lean y escriban lo más que se pueda de información y para ello cada nodo de un árbol-B es tan grande como el tamaño de una página completa lo que pone un límite al número de hijos de cada nodo. Para un gran árbol-B almacenado en disco, se tienen factores de ramificación de entre 50 y 2000, dependiendo del número de llaves o valores que tenga cada nodo (tamaño de la página). Un factor grande de ramificación reduce tanto el peso del árbol como el número de accesos al disco requerido para encontrar una llave o valor.

A continuación, se muestra un árbol-B con un factor de ramificación de 1001 y peso 2 que puede almacenar más de un billón de llaves. Dentro de cada nodo se encuentran el atributo, que indica el número de llaves que son 1000 por nodo.

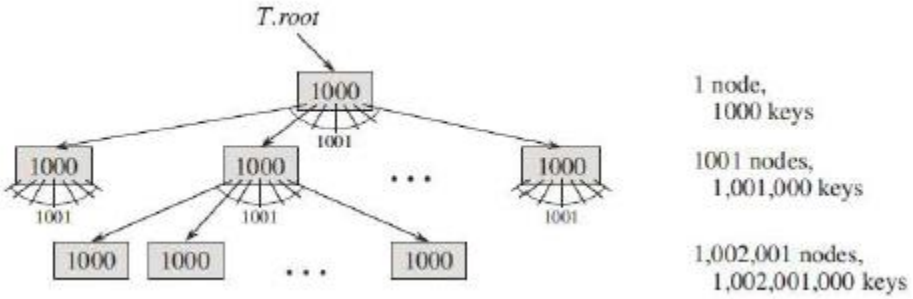

Imagen 1: Ejemplo de un árbol

PREVIO: 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.

Se puede definir al árbol binario como un conjunto finito de nodos (>= 0), tal que:

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

A continuación, se describen algunas características de los árboles binarios.

ÁRBOL BINARIO LLENO

Es un árbol binario lleno si cada nodo es de grado cero o dos. O bien, si es un árbol binario de profundidad que tiene $2 - 1$ nodos (es el número máximo de nodos).

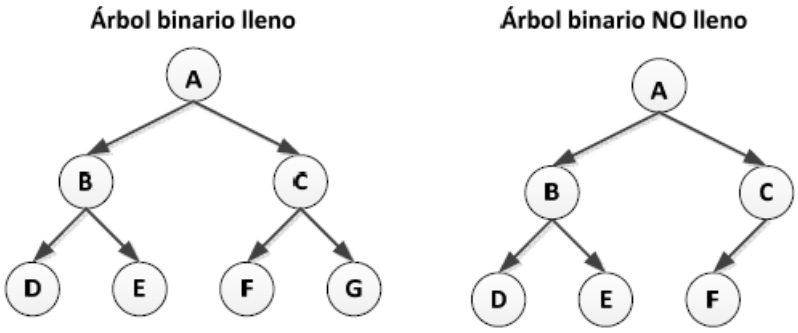

Imagen 2: Ejemplos de unos árboles

ÁRBOLES B

Un Árbol-B es un árbol multi-rama (con raíz T.Raiz) y tiene las siguientes propiedades:

1) Cada nodo tiene la forma

$$((x.h_1, x.llave_1), (x.h_2, x.llave_2), (x.h_3, x.llave_3), \ldots., (x.h_n, x.llave_n))$$

lo que indica que como mucho tiene hijos ($x.h_1$, $x.h_2$, $x.h_3$) y cuenta con las siguientes propiedades.

a. Tiene x.n llaves [o pares (llave, valor)] almacenadas en el nodo.

b. Puede ser hoja o nodo interno, se utiliza el atributo x.hoja que contiene un valor verdadero si es hoja y falso si es nodo interno.

2) Las llaves. separan los rangos de las llaves almacenadas en cada subárbol.

3) Todas las hojas tienen la misma profundidad y el árbol tiene profundidad h.

4) Existe una cota superior e inferior sobre el número de llaves que puede contener un nodo. Esta cota puede ser expresada en términos de un entero ≥ 2 llamado el grado mínimo del árbol-B:

a. Todo nodo que no sea la raíz debe tener al menos − 1 llaves.

b. Todo nodo interno que no sea la raíz debe tener al menos hijos. Si el árbol es vacío, la raíz debe tener al menos una llave. b. Todo nodo puede contener a lo más 2 − 1 llaves. Por lo tanto, un nodo interno puede tener a lo más 2 hijos. Se dice que el nodo está lleno si este contiene exactamente 2 – 1 llaves. El árbol-B más simple ocurre para cuando = 2. Todo nodo interno entonces tiene ya sea 2, 3, o 4 hijos, también llamado árbol 2-3-4.

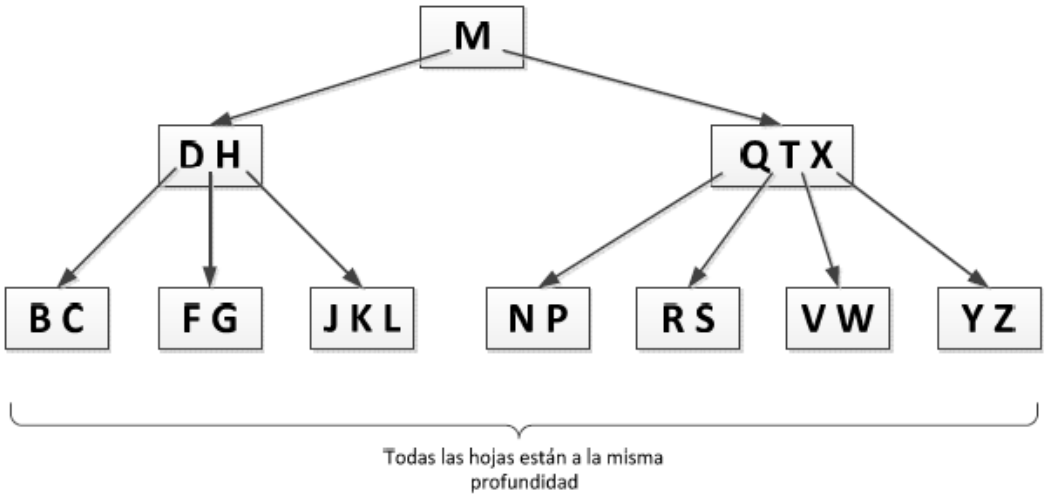

Imagen 3: Ejemplo de un árbol de altura 3 donde su grado mínimo es = 2 y cada nodo puede tener a lo más 2 − 1 llaves, en este caso 3 y al menos −1, que es 1.

OPERACIONES BÁSICAS:

Algunas de las operaciones básicas realizadas en árboles B son búsqueda, inserción y eliminación de una llave. En esta práctica se trabajará con la búsqueda y la inserción.

CREACIÓN DE ÁRBOLES

Para la creación del árbol, lo primero es crear un nodo vacío y después ir insertando llaves e ir creando otros nodos si es necesarios. Para ello en este documento se utilizan los procedimientos propuestos en [1] B-TREE- CREATE() para crear un nodo raíz vacío y B-TREE-INSERT() para agregar nuevas llaves. En ambas funciones se utiliza un procedimiento auxiliar ALLOCATE-NODE(), el cual asigna una página del disco a un nuevo nodo. Se asume que cada nodo creado no requiere una lectura a disco desde que todavía no se utiliza la información almacenada en el disco para ese nodo.
A continuación, se muestra el pseudocódigo de un árbol binario:

```
B-TREE-CREATE (T)
Inicio
        x=ALLOCATE-NODE()
        x.hoja=Verdadero
        x.n=0
        escibirDisco(x)
        T.raiz=x
Fin
```

INSERTAR LLAVES

La inserción es un poco más complicada que en un árbol binario. En un árbol-B no se puede solo crear una nueva hoja e insertarla porque daría lugar a ya no tener un árbol-B. Lo que se hace es insertar una nueva llave en un nodo hoja existente. Como no se puede insertar una llave en un nodo hoja que está lleno, se introduce una operación que divide el nodo lleno (que tiene $2t-1$ llaves) en dos, en torno a la llave del medio ($y.llave_i$). Los dos nodos van a tener $t-1$ llaves cada uno, y la llave del medio ($y.llave_i$) se moverá a su nodo padre para identificar el punto de división entre los dos nuevos árboles. Pero si el padre está lleno, también se debe dividir antes de que se inserte la nueva llave, por lo que el proceso se repetiría con los nodos más arriba hasta llegar al nodo raíz, en ese caso se genera un nuevo nodo raíz que contendrá solo el elemento desplazado hacia arriba de la antigua raíz.

De esta forma, un árbol-B crece por la raíz, de manera que, siempre que la altura se incrementa en 1, la nueva raíz sólo tiene un elemento.

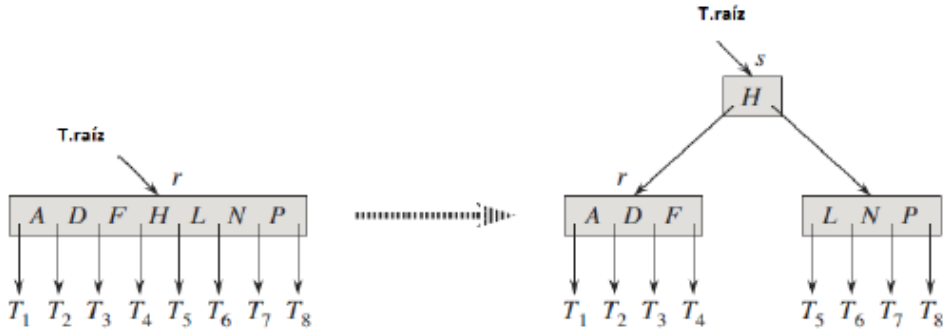

Imagen4: Desarrollo de un árbol B

Con base a el esquema anterior, se puede denotar que el método de inserción dentro de los árboles b es de la forma:

```
B-TREE-INSERT(T, k)
Inicio
        r = T.raiz
        Si r.n == 2t − 1
                s = ALLOCATE-NODO()
                T.raiz = s
                s.hoja = FALSO
                s.n = 0
                s.h1 = r
                B-TREE-SPLIT-CHILD(s, 1)
                B-TREE-INSERT-NONFULL(s, k)
        En otro caso
                B-TREE-INSERT-NONFULL(r, k)
        Fin Si
Fin
```

Este procedimiento tiene dos casos, primero cuando el nodo raíz está lleno (. == 2 − 1), éste se divide (con B-TREE-SPLIT-CHILD()) y se forma un nuevo nodo que será la nueva raíz ,esto hace que se incremente la altura del árbol. El segundo caso se da cuando la raíz no está llena y se puede insertar la llave con TREE-INSERT-NONFULL().

El procedimiento B-TREE-INSERT-NONFULL() es recursivo, de forma tal que asegura revisar los niveles de abajo del árbol para insertar la llave en el nodo y posición adecuada y también garantiza que el nodo que se analiza, si está lleno, sea dividido con el llamado a la función B-TREE-SPLIT-CHILD() cuando sea necesario.
A continuación, se muestra el pseudocódigo de la función previa:

```
B-TREE-INSERT-NONFULL(x,k)
Inicio
 i = x.n
 Si x.hoja == verdadero
        Mientras i ≥ 1  y k < x.llave_i
                x.llave_{i+1} = x.llave_i
                i = i − 1
        Fin Mientras
        x.llave_{i+1} = k
        x.n = x.n + 1
        EscribirADisco(x)
 En otro caso
        Mientras  i ≥ 1  y k < x.llave_i
                i = i − 1
        Fin Mientras
        i = i + 1
        Leer Del Disco(x.h_i)
        Si  x.h_i.x == 2t − 1
                B-TREE-SPLIT-CHILD(x. i)
                Si  k > x.llave_i
                        i = i + 1
                Fin Si
        Fin Si
        B-TREE-INSERT-NONFULL( x.h_i , k)
 Fin Si
Fin
```

BÚSQUEDA DE LLAVES

La búsqueda en un árbol-B se parece mucho a la búsqueda en un árbol binario, excepto que en lugar de hacerlo binario o de dos caminos, se hace una decisión de múltiples caminos de acuerdo al número de hijos que tiene el nodo.

Un algoritmo general para buscar la llave puede ser:

1- Seleccionar nodo actual igual a la raíz del árbol.
2- Comprobar si la clave se encuentra en el nodo actual:
   Si la clave está, termina algoritmo
   Si la clave no está:
      Si nodo actual es hoja, termina algoritmo
      Si nodo actual no es hoja y n es el número de claves en el nodo,
         Nodo actual == hijo más a la izquierda
         Nodo actual == hijo más a la derecha a la derecha
         Nodo actual == i-ésimo hijo
   Volver al paso 2.

Una manera de plantear el algoritmo es de forma recursiva, a continuación, se presenta un procedimiento para la búsqueda en árboles-B (propuesto en [1]) donde se toma como entrada un apuntador al nodo raíz de cada subárbol y una llave que se buscará dentro de cada subárbol. La llamada inicial al procedimiento se realiza con el nodo raíz y la llave a buscar, esto es, BTreeSearch(T.raiz ,k). Si la llave está dentro del árbol-B la función retorna el par ordenado, consistente de un nodo y un índice i tal que y.llavei = k. De otra forma el procedimiento regresa nulo.

```
B-TREE-SEARCH(x, k)
Inicio
        i=1
        Mientras i ≤ x.n y k > x.llave_i
                i=i+1
        Fin Mientras
        Si i ≤ x.n y k > x.llave_i
                retorna (x, i)
        Si no
                Si x.hoja==verdadero
                        Retorna Ninguno
                Si no
                        Leer-Disco(x.h_i)
                        Retorna B-TREE-SEARCH(x.h_i, k)
                Fin Si
        Fin Si
Fin
```

Actividad 1. Ejercicios del Manual

Para este apartado se pidió codificar el algoritmo de árboles presentado en el manual de prácticas, para ello a continuación se presenta la descripción y salida del programa.

DIAGRAMA DE CLASES - UML

Para llevar a cabo el algoritmo de árboles en Python fue necesario llevar a cabo el uso de clases dentro de este, es por ello que, mediante los conocimientos previos de POO, se tuvo primero que llevar a cabo un diagrama UML de clases para poder considerar los requisitos que debía tener el programa, a continuación, se muestra el diagrama:

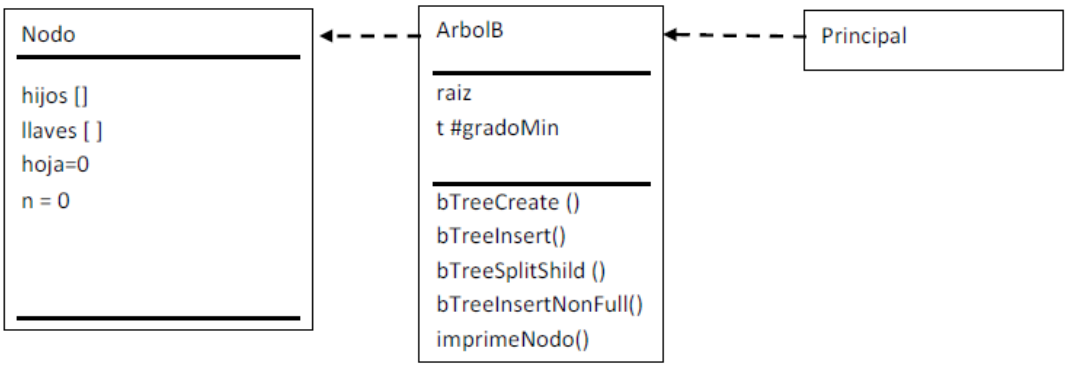

Imagen 5: Diagramas UML – Diagrama de Clases del código previo del algoritmo de Árbol B

SALIDA DEL PROGRAMA

A continuación, se muestran la salida del programa empleando el código dado en la práctica y a su vez los métodos que se tuvieron que llevar a cabo para hacer la búsqueda de elementos del siguiente árbol binario.

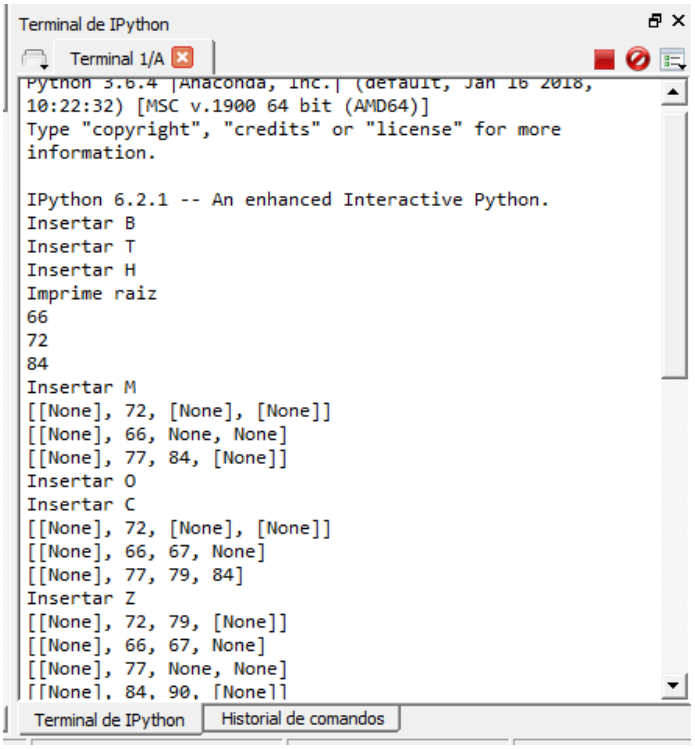

Imagen 7: Salida del programa empleando el algoritmo de árbol B en Python.

NOTA: La explicación de cada uno de los métodos realmente nos pareció innecesaria debido a que totalmente fue basado en los pseudocódigos de la práctica lo cual resulta un poco

Actividad 2. Ejercicios del laboratorio

2.A Compilar y ejecutar proyecto (Grafo propuesto y grafos de la clase)

Para este apartado simplemente se creó un proyecto en NetBeans donde la clase principal "practica9" la cual a través de su método main fuera la encargada de llamar a a través de instancias o directamente mediante métodos las distintas clases que a continuación se describirán:

DEFINICIÓN DE CLASES:

NodoB (véase en NodoB.java) – esta clase tiene como principal función armar el árbol b, esta contiene atributos como, listas de llaves y listas de hijos tipo NodoB, el orden el árbol, así como diversos índices que ayudaran en algunos métodos como es el caso de la inserción y división celular, que son tanto enteros como booleanos.

Aquí se declara un constructor que recibe el orden del árbol, además del índice que determina si el nodo es hoja o no, todos los demás atributos se inicializan de forma predeterminada.

AddNum-
Dentro de los métodos se inicia con el de addNum el cual recibe como parámetros el nodo donde se insertará el dato, el número y un índice que determinará donde ira en caso de que sea de nivel mayor a 0 En esta función se determinan 3 casos cuando el nodo raíz es hoja a su vez en ese caso solamente se inserta en la raíz, el segundo caso cuando es hoja el nodo, de esa manera se insertará en aquel nodo, en otro caso cuando se esté en un nodo no hoja y no raíz entonces mediante comparaciones del dato y las llaves se determinará en que nodo hijo tiene que ser insertado esto llamando a la función de manera recursiva dándole ahora como primer parámetro el nodo seleccionado

En todos los casos en la posibilidad de sobrepasar el índice impuesto que determina el número de elementos o llaves máximas, entonces hará división celular, en caso de ser raíz y hoja se llamará a la función de divisionPadre, en caso de nodo hoja solamente se llamará a la función divisionHijo.
Estos dándoles como parámetros el nodo que se va a dividir y el número que se tiene que insertar

División Padre- esta recibe el nodo raíz y el número que no alcanzo a insertarse el proceso de división es

- Crear dos nodos hijo
- Crear una lista auxiliar con los 5 elementos ordenados
- Crear sublistas de llaves de 0 al índice 1 para el hijo1 y de 3 a 4 para el hijo 2 y asignárselos
- Limpiar las claves del nodo raíz y agregar el elemento medio de la lista auxiliar
- Asignar la relación de los hijos al nodo padre es decir la raíz,
- Limpiar la lista de hijos y agregar los 2 hijos en dicha lista
- Modificar los índices de cantidad de claves 1 para la raíz y 2 para los hijos
- Modificar el estado de los nodos (si son hojas/raíz o no)

Esto quiere decir que el raíz solamente queda con el elemento medio y se crean dos hijos con las mitades correspondientes a las sublistas, después agregando las relaciones padre-hijo y modificando sus atributos para su correcto manejo posterior.

DivisionHijo- este recibe como parámetros el nodo que se va a dividir, el número que se insertara y el índice del nodo donde proviene.

Aquí se plantean 3 posibles casos de división que se aplicaran recursivamente dependiendo el caso que se encuentre el árbol, los tres siguen un proceso similar al de divisiónPadre aunque tienen algunas diferencias

El primer caso es cuando es un nodo hoja, la parte en la que se diferencia es en la asignación de relaciones padre-hijo, ya que aquí el nodo que se están tomando los datos es hoja por lo tanto este nodo se tiene que eliminar de la lista de su padre y los hijos tiene que agregarse en el lugar donde estaba el nodo y un lugar posterior. es por ello que se hace uso del índice para localizar donde se insertaran los nodos hijos.

En el segundo caso es cuando es un nodo intermedio, la parte en la que se diferencia es la relación de hijos del nodo y el padre del nodo,
Para ello se agregaron operaciones adicionales como

- Se tienen que generar dos sublistas de la lista de hijos del nodo que se divide
- Esas sublistas mediante un for se tiene tienen que cambiar las referencias de su antiguo padre al hijo que se está creando, ya que al crear sublistas de nodos estas continúan con sus atributos intactos es por ello que su padre se tiene que modificar
- Se tienen que sacar el índice donde estaba el nodo y actualizarlo, es por ello que se hace uso del indexOf para poder insértalos en el índice y su posterior.

En el tercer caso es cuando se vuelve a dividir el padre, pero ahora con hijos, al igual que el anterior se tienen que modificar las relaciones entre sus hijos al momento de generar las sublistas, de ahí en fuera es lo mismo que divisiónPadre, cabe aclarar que las referencias de los hijos de los hijos del padre permanecen intactas ya que no hay forma de que alteren sus relaciones ya establecidas con anterioridad

Además, cabe destacar que en los casos 1 y 2 se tiene una verificación en donde si el número medio a insertar en su padre excede el tamaño permitido entonces llamara a la función DivisionHijo para que haga el caso siguiente esta es una manera recursiva de llevar a cabo la división celular de manera modular, es decir por casos. También se agrega que los casos son métodos de la división.

ArbolB (véase en arbolB.java) – para esta clase solamente se tienen como atributos el orden del árbol y un NodoB que será la raíz

Dentro del constructor recibe el orden del árbol y se lo asigna, se declara un nodo con el orden y su respectivo 1 que indica que es hoja, a su vez de modifica su estado a que es raíz de igual manera.

Dentro de los métodos

Agregar- este método manda a llamar a du vez al método addNum del NodoB, este dándole como parámetros la raíz donde va a iniciar, el número y un índice que no indica nada.

Imprimir – este inicia con el nodo raíz solamente imprimiendo el nivel (0) y su lista de llaves ".key"
Posteriormente se llama a ImpUtil que recibe el nodo raíz y el nivel.

ImpUtil- este recorre todos los nodos hijos de la raiz en un for además de que va imprimiendo cada lista de llaves de cada nodo hijo, posteriormente si todavía hay hijos, se recorrerán los hijos de cada nodo, llamando de manera recursiva a ImpUtil, dándole como parámetros el nodo hijo y un nivel más.

Para hacer la lectura se tiene que observar que el algoritmo se recorre de izquierda a derecha de arriba abajo, y por cada nodo le corresponden 2 hijos del siguiente nivel, la impresión se hace en orden.

Buscar- recibe la raiz y el numero buscado, para ello se usó la idea de la impresión, sin embargo, lleva comparaciones para delimitar los rangos de búsqueda, primero se hace uso de un índice para ver en que nodo se tiene que buscar y de esa manera no hacer búsqueda lineal, en primera instancia se revisa la raíz, se compara el número con la lista de llaves, cada vez que el número sea mayor ese índice aumenta en 1, si se encuentra devuelve el nivel donde se encontró y un booleano, si no entonces se llama a BuscarUtil,

que manda como parámetros el nodo hijo donde se va a buscar(uso del índice generado) y el numero buscado, este método sigue la misma lógica

Donde radica es su llamada recursiva y la condición previa que debe cumplir ya que si no es esta manera entra en un nodo que no existe, es decir que no tiene que revisar los nodos hijo de los nodos hoja ya que son inexistentes.

En caso de no encontrar el numero solamente mandara un booleano, que será false.
En caso de encontrarlo mandará un mensaje indicando el nivel y devolverá un booleano que será true

Ejecución del programa /dibujo del árbol

Inserción de raíz / divisionPadre

```
se agrega: 1
0
5
nodo raiz [1]
se agrega: 2
1
5
nodo raiz [1, 2]
se agrega: 3
2                       list[1, 2, 3, 4, 5]
5                       dividido
nodo raiz [1, 2, 3]     padre [3]
se agrega: 4
3                       hijo 0 [1, 2]
5
nodo raiz [1, 2, 3, 4]  hijo2 1 [4, 5]
```

Inserción Hijo/división caso 1

```
se agrega: 6
aumenta indice de insercion
inserta en nodo 1
[4, 5]
2->5
padre[3]
hijo: [4, 5, 6]
se agrega: 7
aumenta indice de insercion
inserta en nodo 1
[4, 5, 6]
3->5
padre[3]
hijo: [4, 5, 6, 7]
se agrega: 8
aumenta indice de insercion
inserta en nodo 1
[4, 5, 6, 7]
4->5
entra caso 1
INDICE 1
padre [3]
list[4, 5, 6, 7, 8]
dividido
hijo 2 [4, 5]
hijo2 2 [7, 8]
```

Se han insertando números del 1 al 26

Inserción de 26 división caso 2

```
se agrega: 26
aumenta indice de insercion
inserta en nodo 1
[12, 15, 18, 21]
aumenta indice de insercion
aumenta indice de insercion
aumenta indice de insercion
aumenta indice de insercion
inserta en nodo 4
[22, 23, 24, 25]
4->5
entra caso 1
INDICE 4
padre [12, 15, 18, 21]
list[22, 23, 24, 25, 26]
dividido
hijo 2 [22, 23]
hijo2 2 [25, 26]
entra caso 2
list[12, 15, 18, 21, 24]
hijo1 [12, 15]
hijo2 [21, 24]
padre [9, 18]
```

La división caso 3 requiere un total mínimo de 56 elementos para realizarla por primera vez y más de 180 para una segunda.

Impresión del árbol /dibujo

```
nivel 0
nodo raiz-[9, 18]
nivel: 1
 nodo hijo [3, 6]
nivel: 1
 nodo hijo [12, 15]
nivel: 1
 nodo hijo [21, 24, 27]
nivel: 2
 nodo hijo [1, 2]
nivel: 2
 nodo hijo [4, 5]
nivel: 2
 nodo hijo [7, 8]
nivel: 2
 nodo hijo [10, 11]
nivel: 2
 nodo hijo [13, 14]
nivel: 2
 nodo hijo [16, 17]
nivel: 2
 nodo hijo [19, 20]
nivel: 2
 nodo hijo [22, 23]
nivel: 2
 nodo hijo [25, 26]
nivel: 2
 nodo hijo [28, 29, 30, 31]
```

Búsqueda
11 y 100

```
se busca 11
entra en hijo 1
entra en hijo 0
se encontro en nivel 2
true
se busca 100
entra en hijo 2
entra en hijo 3
false
```

CONCLUSIONES

MURRIETA VILLEGAS ALFONSO

A lo largo de esta práctica se aprendieron los principios y fundamentos de uno de los casos particulares más comunes de los grafos que son los árboles. En esta práctica se partió desde el concepto de básico de árbol para posteriormente abordar tanto los métodos de recorrido como un caso particular y aplicable en computadora que es el árbol B.
El uso de los árboles B sin duda alguna se da en casos útiles como por ejemplo guardar grandes cantidades de información ya que la misma estructura de es este permite esa propiedad (Cabe destacar que esto siempre dependerá del orden establecido del árbol).
Por otro lado, y para destacar aspectos fundamentales de la práctica, realmente se tuvo que realizar muchísimas pruebas para poder hacer el apartado de la eliminación y el reacomodo de los nodos y es que como se mencionó previamente existen diversos casos al momento de eliminar un nodo del árbol.
Por último, realmente esta práctica ha sido la más pesada de todas que se han realizado en EDA, el árbol B es sin duda una estructura que realmente conlleva muchísimas ventajas para el manejo eficiente de datos, pero realmente el trasfondo y manejo de este es sin duda complejo tanto de entender como de programar.

VALDESPINO MENDIETA JOAQUÍN

En conclusión, de la práctica se han logrado los objetivos, se ha logrado identificar las características y funciones de un árbol b, cabe destacar que su implementación ha sido de las más difíciles en cuestión de las practicas presentadas anteriormente, ya que esta involucra una lógica más avanzada, en tanto a la división celular, ya que esta requiere un estricto orden además de un cuidadoso uso de las referencias de los nodos, ya que aunque sea un mínimo detalle el árbol generado ya no será un árbol b, como fue una dificultad que se generó al realizar un cambio de referencias de padre, ya que al generar una sablista esta permanecía con el antiguo padre no con el padre nuevo, esto a hora de imprimirlo generaba un problema ya que el grafo era cíclico, por lo tanto perdía su estructura, una gran ventaja de un árbol b es la organización de los datos, de manera jerárquica, además de la búsqueda que asemeja a la búsqueda binaria, sin embargo como cualquier algoritmo a veces es difícil plantearlo, en este caso la implementación costo demasiado, por lo tanto podría considerarse una desventaja, también cabe aclarar que se modificó el uso de arreglos a ArrayList ya que se consideró que su implementación reduciría la

dificultad ya que esta tiene métodos que facilitan algunas operaciones que en tanto a un arreglo serían necesarias más implementaciones de métodos, en fin una práctica que involucro un constante trabajo para lograr una implementación.

REFERENCIAS

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). Problem Solving with Algorithms and Data Structures using Python. Segunda Edición.

Elba Karen Saenz García. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos 1I.* UNAM, Facultad de Ingeniería.