Laboratorio Nro. 4
Tablas de Hash y Árboles

Alejandro Villada Toro
Universidad Eafit
Medellín, Colombia
avilladat@eafit.edu.co

Cristian Alzate Urrea
Universidad Eafit
Medellín, Colombia
calzateu@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

3.1

Porque los métodos de inserción y acceso(al primer y ultimo elemento) sea hacen en tiempo constante, es decir, su complejidad es O(1), y esto es muy practico para que el algoritmo funcione con una complejidad menor. Se obtuvo que la complejidad del algoritmo que calcula las colisiones entre las abejas es O(n*log(n)).

3.2

Para lograr crear un árbol genealógico cuya búsqueda e inserción se haga en tiempo logarítmico, utilizando arboles binarios de búsqueda, es necesario relacionar cada miembro de la familia, del árbol en cuestión, con un número, estos números tienen que ser muy precisos para que el árbol mantenga integridad, no es posible el auto balanceo, porque los nodos se moverían de posición, y podríamos terminar como el papá de un tío nuestro. De esta manera seria muy complicado implementar el árbol genealógico, porque es difícil y complejo encontrar los valores para que el árbol quede correcto y cada miembro que inserte quede en la posición indicada, ya que se debería revisar el número de cada miembro, y así la complejidad queda de *n* como el total de miembros del árbol. Por lo tanto, no creemos que sea eficiente.

3.3

PreOrderTreeToPosOrderTree

El metodo *preOrderTreeToPosOrderTree* permite imprimir un árbol en *posOrder* a partir de un arreglo cuyos elementos son los nodos de un árbol recorridos en *preOrder*. Recibe como parámetros un arreglo de enteros *preOrderTree* que contiene los nodos de un árbol recorridos en *preOrder.* El funcionamiento es siguiente: primero la raíz del árbol se vuelve el primer elemento del arreglo *this.root = new MyNode(preOrderTree[0]),* luego se añaden cada uno de los elementos del arreglo al árbol por medio de un ciclo que recorre todos los elementos del arreglo *insert(preOrderTree[i]).* Por último, se llama el metodo *printPosOrder* que es un metodo recursivo que imprime todos los elementos de un árbol en *posOrder,* su complejidad es n, que es el número de nodos del arbol.

PathSum

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

El método *pathSum* depende completamente del metodo *pathSumAux* por lo que este último es el que será explicado. El método *pathSumAux* verifica si la suma de los elementos de alguno de los caminos de raíz a hoja de un árbol es igual a un entero dado; es un metodo recursivo ya que se llama a si mismo durante su ejecución. Recibe como parámetros el entero *sum* en el que se guardara la suma de los nodos que comprenden el camino de raíz a hoja; el nodo *node* que sirve para ir recorriendo cada nodo del árbol en cuestión; un entero *target* que es el que se verifica; y una lista enlazada de enteros *paths* que nos dice si en un camino la suma de los nodos es igual a target *sum==target.* El funcionamiento es el siguiente: primero pregunta si *node* tiene dos hijos, es decir, que tanto el nodo de la izquierda como el de la derecha sean distintos de *null*, en ese caso se hacen dos llamados recursivos, el primero con *sum* igual a la suma de *sum* con el valor de *node sum + node.data, node* como el nodo izquierdo *node.left, target* y *paths* se mantienen igual. El segundo llamado con *sum* igual a la suma de *sum* con el valor de *node sum + node.data, node* como el nodo derecho *node.right, target* y *paths* se mantienen igual. En caso de que la condición no se cumpla, es decir, que *node* no tenga dos hijos, se pregunta si el nodo de la izquierda es distinto de *null node.left != null,* en caso afirmativo se hace un solo llamado recursivo con *sum* igual a la suma de *sum* con el valor de *node sum + node.data, node* como el nodo izquierdo *node.left, target* y *paths* se mantienen igual. Si el nodo de la izquierda si es igual a *null* entonces se pregunta si el nodo de la derecha es distinto a *null node.right != null*, en caso de que si, se hace un solo llamado recursivo con *sum* igual a la suma de *sum* con el valor de *node sum + node.data, node* como el nodo derecho *node.right, target* y *paths* se mantienen igual. Si ninguna condición se cumple indicaría que *node* no tienen hijos, este sería caso de parada; en cuyo caso se preguntaría si *sum* más el valor de *node* es igual a *target sum + node.data == target;* en caso afirmativo se le agrega el elemento 1 a *paths*, en caso contrario no se hace nada. Finalmente se retorna si *paths* no es vacío *return !paths.isEmpty().*

3.4

PreOrderTreeToPosOrderTree

La complejidad de este metodo está dada en función de n, que es el número de elementos del arreglo, que finalmente se convertirá en el número de elementos del árbol. La complejidad de la primera parte del código es *O(n*log(n))* ya que se tiene que recorrer cada elemento del arreglo e insertarlo al árbol, lo primero se hace *n* y el método *insert* tiene complejidad de *O(log(n)).* La complejidad del método *printPosOrder* es n, por lo tanto, la complejidad nos queda *O(n+n*log(n))*.

PathSum

Como es un método recursivo la complejidad está dada por el peor de los casos, que es donde se hacen dos llamados recursivos. La variable *n* es el número de nodos que tiene el árbol y la ecuación de recurrencia queda: *T(n) = T(n/2) + T(n/2) + C_1, al resolver la ecuación obtenemos C_1(n-1) + (C_1/2)n luego la complejidad está dada por:*
O(C_1(n-1) + (C_1/2)n) al aplicar las reglas de suma y producto la complejidad nos queda O(n).

3.5

PreOrderTreeToPosOrderTree

La variable *n* en el método *PreOrderTreeToPosOrderTree* representa el número de elementos del arreglo que recibe como parámetro *preOrderTree; que en el fondo es el número de nodos del árbol que recorrido.*

PathSum

La variable *n* en el método *pathSum representa el número de nodos que tiene el árbol.*

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

4) Simulacro de Parcial

4.1.
- ***4.1.1.*** *B*
- ***4.1.2.*** *D*

4.2.
- ***4.2.1.*** *C*

4.3.
- ***4.3.1.*** *false;*
- ***4.3.2.*** *0;*
- ***4.3.3.*** *a.izq, suma – a.data*
- ***4.3.4.*** *a.der, suma – a.data*

4.4.
- ***4.4.1.*** *C*
- ***4.4.2.*** *A*
- ***4.4.3.*** *D*
- ***4.4.4.*** *A*

4.5.
- ***4.5.1.*** *p.data == toInsert*
- ***4.5.2.*** *toInsert > p.data*

4.6.
- ***4.6.1.*** *D*
- ***4.6.2.*** *return 0;*
- ***4.6.3.*** *== 0*

4.7.
- ***4.7.1.*** *A*
- ***4.7.2.*** *B*

4.8.
- ***4.8.1.*** ***Consideramos que falta información***

4.9.
- ***4.9.1.*** *A*

4.10.
- ***4.10.1.*** ***Por definir.***

4.11.
- ***4.11.1.*** *B*
- ***4.11.2.*** *A*
- ***4.11.3.*** *B*

4.12.
- ***4.12.1.*** *i)*
- ***4.12.2.*** *a)*
- ***4.12.3.*** *b)*

4.13.
- ***4.13.1.*** *raiz.id*
- ***4.13.2.*** *D*

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

5) Lectura recomendada (opcional)

Mapa conceptual

6) Trabajo en Equipo y Progreso Gradual (Opcional)

Integrante	Fecha	Hecho	Haciendo	Por hacer
Cristian	22/10/2020	Primer vistazo del laboratorio		realización del laboratorio
Alejandro	22/10/2020	Primer vistazo del laboratorio		realización del laboratorio
Cristian	23/10/2020	La planeación del diseño del algoritmo para la solución del punto 1.1	Diseñando el algoritmo para solución del punto 1.1	implementación del algoritmo que da solución al punto 1.1
Alejandro	23/10/2020	La planeación de la implementación de los algoritmos que dan solucion a los puntos 2.1 y 2.2	Implementando los algoritmos que dan solucion a los puntos 2.1 y 2.2	Realización de pruebas que determinen la correcta ejecución de los algoritmos que dan solucion a los puntos 2.1 y 2.2
Cristian	24/10/2020	Avances en la implementación del algoritmo para la solución del punto 1.1	Implementando el algoritmo que da solucion al punto 1.1	Ajustes y pruebas que determinen la correcta ejecución del algoritmo que soluciona el punto 1.1
Alejandro	24/10/2020	Documentación de los métodos de los puntos 2.1 y 2.2, y realización de la primera parte del informe de laboratorio	Escribiendo la primera parte del informe de laboratorio. Corrigiendo y mejorando el algoritmo que da solución al punto 2.1 y 2.2	Realización del simulacro de parcial.

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

Cristian	25/10/2020	implementación del algoritmo que resuelve el punto 1.1, y realización del simulacro de parcial	Puliendo detalles en el algoritmo que da solución al algoritmo del punto 1.1. Leyendo y resolviendo las preguntas del simulacro del parcial entre los dos	Hacer la lectura del informe cada uno
Alejandro	25/10/2020	Realización del simulacro de parcial	Leyendo y resolviendo las preguntas entre los dos	Hacer la lectura del informe cada uno

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473