UTEP

Data Structure 2302

Lab3

Alcantar,Luis

TR 9:00-10:20

B-Tree is a different data structure were the BigO notation is not taken in consideration. When we talk about BigO notation we assume that all operations are equal (time). In fact with the B-Tree we what to make sure we do not use all the memory that we keep the tree as shot as possible (deep). When the computer runs out of memory it start to use disk space. Disk space is expensive a normal disk takes on average takes 1/120 of a second or 8.3 m on access time.

The problem for this lab is to implement several methods of the B-Tree, because is not possible to exactly know the deep of every leave in a B-Tree all the methods need to use recursion. The problem is to create two fields to keep track of the size and the number of nodes in the B-Tree. In particular the B-Tree methods require a combination of for loops and recursion, to be able to transverse the hole tree.

Methods:

The methods can be really simple or complicate as the method find max and fin min. Due the fact that the max element in the last position of the last leave, it can be easily return as well the min is in leave of the left in the first position. In the other hand methods that required checking every node are more complex because they need to be transverse the hole tree with out exception.

In a B-Tree is more easily to find an element because the data is sorted in a way that we can only visit a part of the tree. Due the fact most of the methods are recursively they are self-explanatory not as elegant as the methods should be but they are acceptable.

In this lab I did not perform any experiments I just make sure that all the methods work as indicated. The only experiment performs for this lab was to change the number of elements and the number, adding duplicates and negatives numbers.

In conclusion, a B-Tree is a very useful data structure wen we want to store a lot of elements on average we can expect logarithmic time on the most of the operations. In the other hand is not easy to implement, it is a complex data structure .The methods are pure recursion with a combination of for loops something in my opinion can be very expensive. The good thing of a B-Tree is that all the elements are sorted and it easier to find an element. In this lab we use different field and we include size and nodes to keep track of the size the number of elements, and the number of nodes .In fact it will be more useful to include this B-Tree in a hash table compare to a link List(chaining).

I certify that this project is entirely my own work. I wrote, debugged and tested the code being presented, performed the experiments and wrote the report. I also certify that I did not share my code or report or provide inappropriate assistance to any student in the class.

Code:

**import** java.util.Random;

**public** **class** BTreeTest {

**public** **static** **void** main(String[] args) {

Random generator = **new** Random();

BTree T = **new** BTree(3);

// for (int i=0;i<30;i++){

// T.insert(generator.nextInt(100));

// }

//T.print();

T.insert(23);

T.insert(22);

T.insert(21);

T.insert(20);

T.insert(19);

T.insert(18);

T.insert(17);

T.insert(16);

T.insert(15);

T.insert(14);

T.insert(13);

T.insert(12);

T.insert(11);

T.insert(9);

T.insert(8);

T.insert(4);

T.insert(23);

T.insert(23);

System.*out*.println("There are : "+T.size());

System.*out*.print("The size of the tree(field) : ");

T.printsize();

System.*out*.println("There heigh is : "+T.heigh());

System.*out*.print("There heigh is (field): ");

T.printHeight();

System.*out*.print("There Largest element : ");

T.largestElement();

System.*out*.println("There Smallest element : "+ T.smallerElement());

System.*out*.print("Desending order : " );

T.desending();

System.*out*.println("");

System.*out*.print("Find an element 19 ---->");

System.*out*.println(T.findK(19));

T.assending();

System.*out*.println("");

System.*out*.println("The numeber of nodes is : ->"+T.numberNodes());

System.*out*.print("The numeber of nodes is(Field) : ->");

T.printNodesField();

System.*out*.println("are all nodes full : "+ T.allFull());

System.*out*.println("thre are nodes with 4 elements : "+ T.nodesWithNElements(4));

System.*out*.println("The number of leaves in the tree : "+ T.numberLeaves());

System.*out*.println("The Nodes with the minimal number of keys : "+T.nodesWithMinimalNumeberOfElements());

System.*out*.println("The Nodes with the Maximum number of keys : "+T.numberOfNodeWithFullElements());

**int** n=2;

System.*out*.println("The number of kesy in deep 1 is : "+T.nodesInDeepN(n));

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **class** BTreeNode {

**public** **int**[] key;//array int

**public** BTreeNode[] c; //un array the nodetree

**boolean** isLeaf;

**public** **int** n;

**private** **int** T; //Each node has at least T-1 and at most 2T-1 keys

**public** BTreeNode(**int** t){

T = t;

isLeaf = **true**;

key = **new** **int**[2\*T-1];

c = **new** BTreeNode[2\*T];

n=0;

}

**public** **boolean** isFull(){

**return** n==(2\*T-1);

}

**public** **int** insert(**int** newKey){

// Instert new key to current node

// We make sure that the current node is not full by checking and

// splitting if necessary before descending to node

//System.out.println("inserting " + newKey); // Debugging code

**int** i=n-1;

**int** x =0;

**if** (isLeaf){

**while** ((i>=0)&& (newKey<=key[i])) {

**if**(newKey==key[i])

x=-1;

key[i+1] = key[i];

i--;

}

**if**(x>=0){

n++;

key[i+1]=newKey;

}

}

**else**{

**while** ((i>=0)&& (newKey<=key[i])) {

**if**(newKey==key[i])

x=-1;

**else**

i--;

}

**if**(x>=0){

**int** insertChild = i+1; // Subtree where new key must be inserted

**if** (c[insertChild].isFull()){

x++;

// The root of the subtree where new key will be inserted has to be split

// We promote the mediand of that root to the current node and

// update keys and references accordingly

//System.out.println("This is the full node we're going to break ");// Debugging code

//c[insertChild].printNodes();

//System.out.println("going to promote " + c[insertChild].key[T-1]);

n++;

c[n]=c[n-1];

**for**(**int** j = n-1;j>insertChild;j--){

c[j] =c[j-1];

key[j] = key[j-1];

}

key[insertChild]= c[insertChild].key[T-1];

c[insertChild].n = T-1;

BTreeNode newNode = **new** BTreeNode(T);

**for**(**int** k=0;k<T-1;k++){

newNode.c[k] = c[insertChild].c[k+T];

newNode.key[k] = c[insertChild].key[k+T];

}

newNode.c[T-1] = c[insertChild].c[2\*T-1];

newNode.n=T-1;

newNode.isLeaf = c[insertChild].isLeaf;

c[insertChild+1]=newNode;

//System.out.println("This is the left side ");

//c[insertChild].printNodes();

//System.out.println("This is the right side ");

//c[insertChild+1].printNodes();

//c[insertChild+1].printNodes();

**if** (newKey <key[insertChild]){

c[insertChild].insert(newKey); }

**else**{

c[insertChild+1].insert(newKey); }

}

**else**

c[insertChild].insert(newKey);

}

}

**return** x ;

}

**public** **void** print(){

//Prints all keys in the tree in ascending order

**if** (isLeaf){

**for**(**int** i =0; i<n;i++)

System.*out*.print(key[i]+" ");

System.*out*.println();

}

**else**{

**for**(**int** i =0; i<n;i++){

c[i].print();

System.*out*.print(key[i]+" ");

}

c[n].print();

}

}

**public** **void** printNodes(){

//Prints all keys in the tree, node by node, using preorder

//It also prints the indicator of whether a node is a leaf

//Used mostly for debugging purposes

**for**(**int** i =0; i<n;i++)

System.*out*.print(key[i]+" ");

System.*out*.println(isLeaf);

**if** (!isLeaf){

**for**(**int** i =0; i<=n;i++){

c[i].printNodes();

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **class** BTree {

**private** BTreeNode root;

**private** **int** T; //2T is the maximum number of childen a node can have

**private** **int** height;

**private** **int** size;

**private** **int** nodes;

**public** BTree(**int** t){

root = **new** BTreeNode(t);

T = t;

height = 0;

size=0;

nodes++;

}

/\*

\* The method print size prints the field size

\*/

**public** **void** printsize(){

System.*out*.println("Tree size is "+size);

}

/\*

\* The method print height prints the field height

\*/

**public** **void** printHeight(){

System.*out*.println("Tree height is "+height);

}

/\*

\* The method print nodes fild prints the field nodes

\*/

**public** **void** printNodesField(){

System.*out*.println("The number of nodes is "+nodes);

}

/\*

\* The method insert check if the root is full if is ful it splits and after checking it calls the method inset of the root

\* The method BTreeNOde.insert it returns a integer, ir the integet is 0 it means it perform an insertion

\* if the method returns -1 it means that it find a duplicate

\* more than o is the times the roots were splited

\*/

**public** **void** insert(**int** newKey){

**if** (root.isFull()){//Split root;

split();

height++;

nodes++;

}

**int** x =root.insert(newKey);

**if**(x>=0)

{

**if**(x==0)

{

size++;

}

**else**

{

size++;

nodes+=x;

}

}

}

/\*

\* a public method that call a private method with the root

\*/

**public** **int** numberLeaves()

{

**if**(root.n==0)

System.*out*.println("the root is empty");

**return** numberLeaves(root);

}

/\*

\* The methodnumberLeaves finds the number of leaves

\* base case: if is a leaf it returns one else

\* it transverse the tree with recursion

\*/

**private** **int** numberLeaves(BTreeNode t)

{

**if**(t.isLeaf)

**return** 1;

**else**

{

**int** total=0;

**for**(**int** i =0;i<=t.n;i++)

{

total+=numberLeaves(t.c[i]);

}

**return** total;

}

}

/\*

\* checks if the nodes are full

\*/

**public** **boolean** allFull()

{

**if**(root.n==0)

**return** **false**;

**else**

**return** allFull(root);

}

/\*

\* we check if the number of nodes is equal to t.n and we return true or false

\*

\*/

**private** **boolean** allFull(BTreeNode t)

{

**if**(t.isLeaf)

{

**if**(t.n==t.key.length)

**return** **true**;

}

**if**(t.n!=t.key.length)

**return** **false**;

**for**(**int** i=0;i<=t.n;i++)

{

**if**(!(allFull(t.c[i])))

**return** **false**;

}

**return** **true**;

}

/\*

\* to find the smaller element

\*/

**public** **int** smallerElement()

{

**if**(root.n ==0)

**return** -1;

**return** smallerElement(root);

}

/\*

\* it goes to the leave in position t.c[0 ] when it find the leave it return the key (t.key[0])

\*/

**private** **int** smallerElement(BTreeNode t )

{

**if**(t.isLeaf)

**return** t.key[0];

**return** smallerElement(t.c[0]);

}

/\*

\* @param integer the integer is used to compere the number of keys in each node

\*/

**public** **int** nodesWithNElements(**int** d)

{

**if**(d>=root.key.length||root.n==0)

**return** -1;

**return** nodesWithNElements(root,d);

}

/\*

\* compares each node with the integer d, and returns the number of elements that t.n ==d

\*/

**private** **int** nodesWithNElements(BTreeNode t,**int** d)

{

**if**(t.isLeaf)

{

**if**(t.n==d)

**return** 1;

**else**

**return** 0;

}

**if**(t.n==d)

**return** 1;

**int** total=0;

**for**(**int** i=0;i<=t.n;i++)

total+=nodesWithNElements(t.c[i],d);

**return** total;

}

/\*

\* print the value

\*/

**public** **void** print(){

// Wrapper for node print method

root.print();

}

/\*

\*

\*/

**private** **int** size(BTreeNode t)

{

**if**(t.isLeaf)

**return** t.n;

**else**

{

**int** total =t.n;

**for**(**int** i =0; i<=t.n;i++)

total += size(t.c[i]);

**return** total;

}

}

**public** **int** size()

{

**if**(root.n==0)

**return** -1;

**else**

**return** size(root);

}

/\*

\*

\*/

**public** **int** numberNodes()

{

**if**(root.n==0)

**return** -1;

**else**

**return** numberNodes(root);

}

**private** **int** numberNodes(BTreeNode t)

{

**if**(t.isLeaf)

**return** t.n;

**else**

{

**int** total=0;

**for**(**int** i = 0;i<=t.n;i++)

total+= 1 + numberNodes(t.c[i]);

**return** total;

}

}

/\*

\* find the key in the tree retrn true or false it the tree was find

\*/

**private** **boolean** findK(BTreeNode t,**int** k)

{

**if**(t.isLeaf)

{

**for**(**int** i =0;i<t.n;i++)

{

**if**(t.key[i]==k)

**return** **true**;

}

}

**else**{

**for**(**int** i = 0;i<=t.n;)

{

**if**(i==t.n)

**return** findK(t.c[t.n],k);

**if**(t.key[i]==k)

**return** **true**;

**if**(t.key[i]<k)

i++;

**else**

**return** findK(t.c[i],k);

}

}

**return** **false**;

}

/\*

\* check taht the root i not empty an call the metho find it was not empty

\*/

**public** **boolean** findK(**int** k)

{

**if**(root.n==0)

**return** **false**;

**else**

**return** findK(root,k);

}

/\*

\* prints the tree in assending order

\*/

**private** **void** assending(BTreeNode t)

{

**if**(t.isLeaf)

{

**for**(**int** i =0;i<t.n;i++)

{

System.*out*.print(t.key[i]+ " ");

}

}

**else**

{

**for**(**int** i =0;i<t.n;i++){

assending(t.c[i]);

System.*out*.print(t.key[i] + " ");

}

assending(t.c[t.n]);

}

}

/\*

\* The method assending that is public it checks if the tree is empty aftet that call the assendingwith the root as a parameter

\*/

**public** **void** assending()

{

**if**(root.n ==0)

System.*out*.println("the tree is empty");

assending(root);

}

/\*

\* the height a method that returns the height of the tree

\*/

**private** **int** heigh(BTreeNode t)

{

**int** x = t.isLeaf?0:1 +heigh(t.c[t.n]);

**return** x;

}

**public** **int** heigh()

{

**if**(root.n==0)

**return** -1;

**return** heigh(root);

}

/\*

\* prints the tree in desending order doing trought the loop

\*/

**private** **void** desending(BTreeNode t)

{

**if**(t.isLeaf)

{

**for**(**int** i=t.n-1;i>=0;i--)

System.*out*.print(t.key[i]+" ");

}

**else**

{

desending(t.c[t.n]);

**for**(**int** i =t.n-1;i>=0;i--){

System.*out*.print(t.key[i] + " ");

desending(t.c[i]);

}

}

}

/\*

\* it checks that the root is not empty

\*/

**public** **void** desending()

{

**if**(root.n==0)

System.*out*.println("The tree is empty!");

**else**

desending(root);

}

/\*

\* print lares element it goes to the end og the leaves recusively by t.c[0] and return the key at position key[t.n-1]

\*/

**private** **void** largestElement(BTreeNode t)

{

**if**(t.isLeaf)

System.*out*.println(t.key[t.n-1]);

**else**

largestElement(t.c[t.n]);

}

/\*

\* it checks thata the tree is not empty

\*/

**public** **void** largestElement()

{

**if**(root.n==0)

System.*out*.println("The tree is empty!");

**else**

largestElement(root);

}

/\*

\* it checks thata the tree is not empty

\*/

**public** **int** nodesWithMinimalNumeberOfElements()

{

**if**(root.n==0)

**return** -1;

**else**

**return** nodesWithMinimalNumeberOfElements(root);

}

/\*

\* finds the numbers of nodes with the minimal number of elements

\*/

**public** **int** nodesWithMinimalNumeberOfElements(BTreeNode t)

{

**if**(t.isLeaf){

**if**(t.n==(t.key.length/2))

**return** 1;

**else**

**return** 0;

}

**if**(t.n==(t.key.length/2))

**return** 1;

**int** total =0;

**for**(**int** i=0;i<=t.n;i++)

total+=nodesWithMinimalNumeberOfElements(t.c[i]);

**return** total;

}

/\*

\* it checks thata the tree is not empty

\*/

**public** **int** numberOfNodeWithFullElements()

{

**if**(root.n==0)

**return** -1;

**else**

**return** numberOfNodeWithFullElements(root);

}

/\*

\* finds the numbers of nodes with the maximun number of elements

\*/

**public** **int** numberOfNodeWithFullElements(BTreeNode t)

{

**if**(t.isLeaf){

**if**(t.n==(t.key.length))

**return** 1;

**else**

**return** 0;

}

**if**(t.n==(t.key.length))

**return** 1;

**int** total =0;

**for**(**int** i=0;i<=t.n;i++)

total+=numberOfNodeWithFullElements(t.c[i]);

**return** total;

}

**public** **int** nodesInDeepN(**int** n)

{

**if**(root.n==0)

**return** -1;

**else** **if**(n==1)

**return** root.n;

**else**

**return** nodesInDeepN(root,n);

}

**public** **int** nodesInDeepN(BTreeNode t,**int** n)

{

**if**(n==1)

{

**return** t.n;

}

**else**

{

**int** total =0;

**for**(**int** i=0;i<=t.n;i++)

total+=nodesInDeepN(t.c[i],n-1);

**return** total;

}

}

/\*

\*

\*/

**public** **void** printNodes(){

// Wrapper for node print method

**if**(root.n==0)

System.*out*.println("The tree is empty!");

**else**

root.printNodes();

}

**public** **void** split(){

// Splits the root into three nodes.

// The median element becomes the only element in the root

// The left subtree contains the elements that are less than the median

// The right subtree contains the elements that are larger than the median

// The height of the tree is increased by one

//System.out.println("Before splitting root");

//root.printNodes(); // Code used for debugging

BTreeNode leftChild = **new** BTreeNode(T);

BTreeNode rightChild = **new** BTreeNode(T);

leftChild.isLeaf = root.isLeaf;

rightChild.isLeaf = root.isLeaf;

leftChild.n = T-1;

rightChild.n = T-1;

**int** median = T-1;

**for** (**int** i = 0;i<T-1;i++){

leftChild.c[i] = root.c[i];

leftChild.key[i] = root.key[i];

}

leftChild.c[median]= root.c[median];

**for** (**int** i = median+1;i<root.n;i++){

rightChild.c[i-median-1] = root.c[i];

rightChild.key[i-median-1] = root.key[i];

}

rightChild.c[median]=root.c[root.n];

root.key[0]=root.key[median];

root.n = 1;

root.c[0]=leftChild;

root.c[1]=rightChild;

root.isLeaf = **false**;

//System.out.println("After splitting root");

//root.printNodes();

}

}