Lab 4 – Report

**Introduction** – For this lab we are learning about B-Trees and with the given code provided to us fully understand how they work. We are to implement certain different methods to manipulate the B-Tree to get what I what from it in specific depths or if I want the a “branch” or a “leaf” (I named them “parents” and “children” respectfully) and to draw the tree correctly. We are also to draw the B-Tree in its correct order and understand it’s been drawn. With the methods that we are to make I must accept a variable and give the correct result. You might notice the pattern in all these methods, which is I check if it’s a child manipulate accordingly otherwise loop through the parent nodes to make sure to go through all the children if needed. The concept for me was easy to understand stand and a tad difficult to implement into code.

**draw**\_**tree** **–** This method takes a reference to the root and while being like the **draw\_tree** it accepts the same parameters to draw a b-tree with any numbers added into it. This method signature was slightly modified by adding the **getHeigh** method to get the height of the **BTreeNode**. Certain variables were kept the same such as **xm** which is the midpoint of the two other variables named **x0** (which is the farthest left-side value being 0) and **x1** (being the farthest right-side value being 100). Then its **yn** which is used to identify the properly different levels and **y** being the value which is the height of the graph being 100. This uses a similar structure to the rest of the method in a certain way, I first check if the node is a parent so I would use **!T.isLeaf** and inside this I would create a variable to keep track of where the set of trees are gonna be drawn a level under the root and another variable to add space between each node as this was a big issue with my code, I failed to take into consideration space between each of the parent nodes. I would have two variables take care of where first the next node is going to be drawn that are in the same level and the space between them, so they wouldn’t overlap. Next, I would have a **for loop** to draw each tree, the left side, the right side, and any tree that should appear on the middle. I would make an **if** statement for the left (**I == 0**), right (**I == T.n**), and any in between which aren’t **I == 0** and **I == T.n**. Inside each I would draw a line connecting each of the parents and I would have a recursive call that traverse through each of the node. Then at the end I would have what would draw the squares in their respective place. Using a for loop I would have a variable to take care of the separation of each of the square, so they wouldn’t overlap, and I would also have drawing tools to properly draw the square without having any overlapping text or drawings over it.

**getHeight –** This method takes a reference to the root and returns the height of the B-Tree. This method has a base that checks the node is a leaf I would **return** **0** and I then **return 1** which is the root being taken into consideration and adds 1 to it if it isn’t a leaf.

**bTreeToArray –** This method takes a reference to the root and puts all its contents into an array while already being sorted, I also pass the index, so I remember the position. I would make a base case checking **if it’s a child node** and I do this so that if a parent is picked, its children will all be inserted into the array in the correct order and increment the index for each child inserted. I then would do an **else** if they aren’t children. Inside this **else** it I would have a loop like the one made for the children to insert all parents into the array in the correct order and increment index every time a number is inserted correctly. I would also have a recursive call inside this loop to make sure I go through all the parents and the children of the parents which the base case will take care of. Depending on how the array is made, in my code I would do in my for loop base case **i < T.n** so I would have a left over child and parent so I then do another recursive call to make sure the leftover children are taken care of. I would then return the index to keep track of where I am in placing the values inside the B-Tree into the array.

**minAtDepth -** This method takes a reference to the root and returns the smallest element in the tree at a given depth, so I also pass the depth to keep track of our traversal through the B-Tree. I first make the base case which is **if the value of depth is 0**, I would return the leftmost element which is the smallest. Then I would do a recursive call which is just traversing through the left since the smallest value in a B-Tree is always at the left, and I would pass depth decreasing its value by 1 every time until the base case is met returning the smallest value at that depth.

**maxAtDepth -** This method takes a reference to the root and returns the highest element in the tree at a given depth, so I also pass the depth to keep track of our traversal through the B-Tree. I first make the base case which is **if the value of depth is 0**, I would return the rightmost element which is the highest. Then I would do a recursive call which is just traversing through the right since the largest value in a B-Tree is always at the right, and I would pass depth decreasing its value by **1** every time until the base case is met returning the largest value at that depth.

**nodesAtDepth -** This method takes a reference to the root and returns the number of nodes in the tree at a given depth, so I also pass the depth to keep track of our traversal through the B-Tree. I first make the base case which is **if the value of depth is 0** and if the criteria is met I would return **1** indicting I found one node at that depth. **Otherwise** I create a variable to remember how many nodes they’re at the depth I want, then I would do a for loop which is doing a recursive call which is just traversing parents in the B-Tree and its children, I would also pass depth decreasing its value by **1** every time until the base case is met and assign the recursive call to the variable created, also making sure to not forget any leftover children. Lastly, I return the variable created inside of your “Otherwise”.

**printAscendingAtDepth -** This method takes a reference to the root and prints all the nodes in the tree at a given depth, so I also pass the depth to keep track of our traversal through the B-Tree. I first make the base case which is **if the value of depth is 0** and I would print the items or “Children” at that depth. **Otherwise** I would do a for loop which is also doing a recursive call which is just traversing parents in the B-Tree and its children and have an extra recursive call outside of the loop so I don’t forget any children or parents, I would also pass depth decreasing its value by **1** every time until the base case is met printing all the nodes at the depth I want.

**ifChildrenAreFull -** This method takes a reference to the root and returns the number of leaf’s in the tree that are full. I first make a variable to keep track of how many children are full, then I make a base case which is **if it’s a leaf** and **its full**, I would return **1** indicting I found one children node at that is full. **Otherwise** I would do a for loop which is doing a recursive call which is just traversing parents in the B-Tree its children, adding what it returns to the variable created at the start, and another recursive call outside for any leftover parent nodes adding what it returns to the variable created at the start. Lastly, I would return the count which contains how many children are full.

**ifParentsAreFull-** This method takes a reference to the root and returns the number of nodes in the tree that are full (non-leaves). I first make a variable to keep track of how many parents are full, then I make a base case which is **if it’s not a leaf** and **its full**, I would return **1** indicting I found one parent node at that is full. **Otherwise** I would do a for loop which is doing a recursive call which is just traversing parents in the B-Tree its children, adding what it returns to the variable created at the start, and another recursive call outside for any leftover parent nodes adding what it returns to the variable created at the start. Lastly, I would return the count which contains how many nodes are full.

**printAtKNode –** This method takes a reference to the root and with a key print all the items that are in the same node as key. I first make a variable to keep track of whether we are going left or right depending on the value of key and the size of the node we are in. Then I would make an **if** statement that would be true if key wasn’t found and we would make another if I had found a child when we didn’t find the key and return immediately since it’s the end of the b-tree **otherwise** we would do another recursive call to traverse to the next node. **If** found the key I would then print out all the items in the node.

**depthOfK –** This method takes a reference to the root and given a key, return the depth at which it is found in the tree, of -1 if k is not in the tree. I first make a variable to keep track of whether we are going left or right depending on the value of key and the size of the node we are in. Then I make a while loop that searches for what they key is by adding one to **i**, in which if its less than the number of items in T and k is greater than the current items in t. Then we make an if statement to check if they key is found or not, if its not found and if it’s a child it will **return -1** otherwise it looks again depending on the value of **i**. If the statement is not true, that means **k** was found a zero is returned. Once the **d** gets a value from the recursive call, **d** is checked to see if the key was found. **If** the key was not found a -1 will be returned otherwise the sum of one and d will be returned as the depth which means the k was found.

**Experiments –** The following is in Nano Time. The following methods was used to create a random array with a size of 10,100,500,1000 to create results. Draw\_tree was omitted from the graph due it is messing up with the visual results.

public static int[] randomArray(int size, int range)//O(N)

{

int[] generatedArray = new int[size];//generate array

Random random = new Random();//random object to generate random numbers

for(int i = 0; i < generatedArray.length; i++)//populate array with random numbers

{

generatedArray[i] = random.nextInt(range);

}

return generatedArray;

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Items | getHeight | 1A | 2A | 2B | 2C | 2D | 2E | 2F | 2G | 2H | 2I |
| 10 | 6565 | 94933785 | 5835 | 14223 | 4011 | 2552 | 23704 | 8023 | 4011 | 3282 | 2817 |
| 100 | 5105 | 853270658 | 10940 | 12399 | 2553 | 2552 | 9117 | 8388 | 6929 | 3282 | 2443 |
| 500 | 4740 | 3711451634 | 31362 | 6929 | 5834 | 3647 | 1422 | 9846 | 104661 | 9846 | 125083 |
| 1000 | 5470 | 7232637113 | 53242 | 8387 | 1824 | 1458 | 35738 | 9847 | 72205 | 5835 | 22245 |

}

**Conclusion –** This lab was a bit challenging but completely understand once I get into the tracing and understanding how B-Tree work. My only issue was the drawing the tree as understanding the concept was okay and then drawing the tree by hand, but when drawing by code it gave me and from what I saw a lot of problems to other students. Other then the draw tree method everything was fun coding it. Understanding what exactly was wanted was another issue that came up for me, but I asked my peers and teacher assistant helped me understand what exactly it was asking for.

/\*\*

\* Objective - To use and understand binary search trees

\* Date Last Modified - 3/10/18

\* Course - Data Structures

\* Lab 3

\* Instructor - Olac Fuentues

\* Teaching Assistants - Zakia Al Kadri

\* @author Isaias Leos Ayala

\*/

public class BTreeTest

{

/\*\*

\* Method that returns the height of the B-Tree

\* @param T

\* @return height of the B-Tree

\*/

public static int getHeight(BTreeNode T)

{

if(T.isLeaf)

{

return 0;

}

return 1 + getHeight(T.c[0]);//Add up the height

}

/\*\*

\* Method for binary search trees as guide, write a method to display a B-tree given a reference to its

\* root.

\* @param T B-Tree

\* @param x0 x most left value

\* @param x1 x most right left value

\* @param y y coordinates

\* @param y\_inc how many levels are the tree going to be in

\*/

public static void draw\_tree(BTreeNode T, double x0, double x1, double y, double y\_inc)

{

double xm = (x0 + x1) / 2;

double yn = y - y\_inc;

if(!T.isLeaf)

{

double increase = (((x1 + xm) / 2.0) - ((x0 + xm) / 2.0)) / T.n;

double space = increase / 2.0;//space between children of different parent

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

{

if(i == 0)

{//Draw the left side of the tree

StdDraw.line(xm, y, ((x1 + xm) / 2), yn);//Take care of the line for the left children

draw\_tree(T.c[0], ((x0 + xm) / 2) - space, ((x0 + xm) / 2) + space, yn, y\_inc);

}

else if(i == T.n)

{//Draw the right side of the tree

StdDraw.line(xm, y, (x0 + xm) / 2, yn);//Take care of the line for the right children

draw\_tree(T.c[T.n], ((x1 + xm) / 2) - space, ((x1 + xm) / 2) + space, yn, y\_inc);

}

else if(i != 0 && i != T.n)

{//Draw any middle trees

StdDraw.line(xm, y, ((x0 + xm) / 2) + (increase \* i), yn);//Take care of the line

draw\_tree(T.c[i], ((x0 + xm) / 2) + (increase \* i) - space, ((x0 + xm) / 2) + (increase \* i) + space, yn, y\_inc);//Take care of the recursive call

}

}

}

int square = 5;

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

{//Draw the squares

int sep = i \* (square \* 2);

StdDraw.setPenColor(StdDraw.WHITE);//To clear the spot

StdDraw.filledSquare(xm + sep, y, square);//Clear the spot of where the squares are gonna appear

StdDraw.setPenColor(StdDraw.BLACK);//Reset back to default

StdDraw.square(xm + sep, y, square);//Draw the square

StdDraw.text(xm + sep, y, Integer.toString(T.item[i]));//Draw the text

}

}

/\*\*

\* Extracts the items in the B-tree into a sorted array

\* @param T

\* @param Array

\* @param index

\* @return

\*/

public static int bTreeToArray(BTreeNode T, int[] Array, int index)

{

if(T.isLeaf)

{//Extra that children

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

{

Array[index] = T.item[i];

index++;

}

return index;

}

else

{

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

{

index = bTreeToArray(T.c[i], Array, index);

Array[index] = T.item[i];//Assign the item to the array

index++;//Increment

}

index = bTreeToArray(T.c[T.n], Array, index);//Recursive call for leftover children

}

return index;

}

/\*\*

\* Return the minimum element in the tree at a given depth d

\* @param T B-Tree

\* @param depth the minimum depth your looking in

\* @return the lowest item in the given depth

\*/

public static int minAtDepth(BTreeNode T, int depth)

{

if(depth == 0)

{//if at the correct depth

return T.item[0];//return the leftmost value

}

else

{//if the correct depth is found return the smallest at that depth

return minAtDepth(T.c[0], depth - 1);

}

}

/\*\*

\* Return the maximum element in the tree at a given depth d

\* @param T B-Tree

\* @param depth the minimum depth your looking in

\* @return the highest item in the given depth

\*/

public static int maxAtDepth(BTreeNode T, int depth)

{

if(depth == 0)

{//if at the correct depth

return T.item[T.n - 1];//return the rightmost value

}

else

{//if the correct depth is found return the largest at that depth

return maxAtDepth(T.c[T.n], depth - 1);

}

}

/\*\*

\* Return the number of nodes in the tree at a given depth d.

\* @param T B-Tree

\* @param depth the minimum depth your looking in

\* @return how many nodes are at the given depth

\*/

public static int nodesAtDepth(BTreeNode T, int depth)

{

if(depth == 0)

{//if your at the depth

return 1;//you found the node at the depth

}

else

{

int numberN = 0;//keep track of the nodes

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

{

numberN += nodesAtDepth(T.c[i], depth - 1);//look through the parents

}

numberN += nodesAtDepth(T.c[T.n], depth - 1);//leftover parents

return numberN;//return the number of nodes that that depth

}

}

/\*\*

\* Print all the items in the tree at a given depth d

\* @param T B-Tree

\* @param depth the minimum depth your looking in

\*/

public static void printAscendingAtDepth(BTreeNode T, int depth)

{

if(depth == 0)

{//Print the nodes found at the depth required

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

{

System.out.print(T.item[i] + " ");

}

return;

}

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

{//look through the parents

printAscendingAtDepth(T.c[i], depth - 1);

}

printAscendingAtDepth(T.c[T.n], depth - 1);//leftover parents

}

/\*\*

\* Return the number of leafs in the tree that are full.

\* @param T B-Tree

\* @return the number of children that are full

\*/

public static int ifChildrenAreFull(BTreeNode T)

{

int count = 0;//keep track of the children that are full

if(T.isLeaf)

{//Make sure its a child

if(T.isFull())

{//Check if the child is full

return 1;

}

}

else

{

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

{

count += ifChildrenAreFull(T.c[i]);

}

count += ifChildrenAreFull(T.c[T.n]);

}

return count;

}

/\*\*

\* Return the number of non-leaves in the tree that are full.

\* @param T B-Tree

\* @return the number of parents that are full

\*/

public static int ifParentsAreFull(BTreeNode T)

{

int count = 0;//keep track of the parents that are full

if(!T.isLeaf)

{//Make sure it isn't a child

if(T.isFull())

{//Check if its full

return 1;

}

}

else

{

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

{

count += ifChildrenAreFull(T.c[i]);

}

count += ifChildrenAreFull(T.c[T.n]);

}

return count;

}

/\*\*

\* Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree.

\* @param T B-Tree

\* @param key

\* @return

\*/

public static int depthOfK(BTreeNode T, int key)

{

int i = 0;

while(i < T.n && key > T.item[i])

{

i++;

}

if(i == T.n || key < T.item[i])

{

if(T.isLeaf)

{

return -1;

}

else

{

int d = depthOfK(T.c[i], key);

if(d == -1)

{

return -1;

}

else

{

return 1 + d;

}

}

}

else

{

return 0;

}

}

/\*\*

\* Given a key k, print all the keys that are in the same node as k.

\* @param T B-Tree

\* @param key

\*/

public static void printAtKNode(BTreeNode T, int key)

{

int i = 0;

while(i < T.n && key > T.item[i])

{

i++;

}

if(i == T.n || key < T.item[i])

{

if(T.isLeaf)

{

return;

}

else

{

printAtKNode(T.c[i], key);

}

}

else

{

for(int j = 0; j < T.n; j++)

{

System.out.println(T.item[j] + " ");

}

}

}

public static void main(String[] args)

{

int x\_max = 400;

int y\_max = 400;

StdDraw.setCanvasSize(800, 800);

StdDraw.setXscale(0, x\_max);

StdDraw.setYscale(0, y\_max);

StdDraw.setPenColor(StdDraw.BLACK);

int[] S =

{

8, 9, 11, 4, 7, 12, 13, 17, 24, 15, 27, 28, 30, 33, 34, 37, 40, 42, 50,55

};

BTree B = new BTree(3);

//Build B-tree from array

for(int i = 0; i < S.length; i++)

{

B.insert(S[i]);

}

BTreeNode T = B.root;

int[] sortedArray = new int[S.length];

bTreeToArray(T, sortedArray, 0);

System.out.print("BTree to Sorted Array: ");

for(int i = 0; i < sortedArray.length; i++)

{

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

}

System.out.println("");

draw\_tree(T, 0, x\_max, y\_max - 5, (y\_max - 10.0) / getHeight(T));//draw tree

int currentDepth = 0;

int smallestNode = minAtDepth(T, currentDepth);

System.out.println("Smallest element at " + currentDepth + " depth: " + smallestNode);

int largestNode = maxAtDepth(T, currentDepth);

System.out.println("Largest element at " + currentDepth + " depth: " + largestNode);

System.out.print("Items at depth " + currentDepth + ": ");

printAscendingAtDepth(T, currentDepth);//8,12,28,34

System.out.println("");

int numOfChildren = ifChildrenAreFull(T);

System.out.println("Number of children nodes that are full: " + numOfChildren);

int numOfParents = ifParentsAreFull(T);

System.out.println("Number of children nodes that are full: " + numOfParents);

int numNodes = nodesAtDepth(T, currentDepth);

System.out.println("Number of Nodes at K Depth: " + numNodes);

int finding = 17;

int itemAtDepth = depthOfK(T, finding);

System.out.println(finding + " was found at depth: " + itemAtDepth);

}

}

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 provided inappropriate assistance to any student in the class.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_