Lab 3 – Report

**Introduction –** For this lab we are reviewing how binary search trees worked and implement certain different methods to create binary trees from sorted arrays, order it, and create sorted arrays from binary search trees. There were given methods for us to use and to understand how the binary search trees work and how to properly manipulate them to help us progress with the lab. We are also learning different ways to search binary search trees, recursively and iteratively, and print the nodes at a certain depth of the tree.

**inOrderWithoutRecursion –** We are to create a inorder travel method that can’t be done recursively while also using stacks. We would assign the given node and create a new one to have a copy of their reference passed. We would then populate the empty stack if the root isn’t empty and push the node into the stack and then assign the next node to the left child. After traversing to the left we would then assign the start printing out the value and then going to the right and repeat this step of going left until null then going right printing out the binary search tree inorder.

**searchIterative –** We are to create a search method that can’t be done recursively. We would first pass the reference of T and the integer we are trying to find. While the Professor has given a recursive search method we are able to copy and modify the structure of the search method given. How it works is it would continue to run while the node isn’t empty. Then we would check if the item inside of the node is the same as the item we are trying to find and return the node. If we can’t find it on the first loop we would check if the item being searched is smaller than the current item inside of the node and if it is we would continue going to the left side and if it isn’t bigger then we would go to the right side of the binary search tree since the item being searched would probably be bigger. If the item isn’t found, we would just turn the null.

**arrayToTree –** We are to create a balanced binary search tree from a sorted array. With the given parameters would be the sorted array, the beginning of the array which would be 0 and the end of the array. We are to assign the middle index number of the array to the start of the binary search tree. Then make a recursive call and return a node that would assign the left node within the parameters we are to pass the middle the beginning and the middle index number position subtracting one. We would do the same thing to the right side by doing a recursive call and returning a node to assign it to the right child, the given parameters would then be the opposite of the left side which is the middle adding by one and the end.

**treeToArray –** Converts the Binary Search Tree to sorted array. This method copies how the inorder method is structure works. The given parameters would be the reference to the given node, an empty array, and the index to keep track of where we are in the array. That if the T isn’t null I would make a recursive call that would take care of the left side then assign the integer inside of the current node and assign it to the array first and increase the index position to each time it’s called then go for the right side. Then I would return the index, which in this case its **i**, to not lose track of where the index is.

**printNodesAtDepth –** This method would print the nodes at the given depth. The given parameters would be the reference to the given node and the depth of where to search, which would be **k,** the structure of this would be that if the given node is null we would end the method. We check if the integer that holds what the depth is 0 we would print the item inside of the current node. This is important to remember as it takes care of what to print at the certain depth since there would be one recursive calls for the left side and one for the right side while also passing the depth and subtracting by one to known that we went through a level. With each recursive call there would have the base case of checking if the depth is 0 and printing the item inside as previously stated.

**count –** This method is used to check how many nodes are inside of the binary search tree. The parameters would be a reference to the given node. It would check if T is null as a base case and return 0 as null wouldn’t be counted. Then we would return 1 plus a recursive call to check the left side which would return a number and then the same thing to the right side and return 1 if there are any nodes. This would add all the nodes from the left and right sides.

**Experimental Results**: The following is in Nano Time. The following two methods were used to create a random array with a size of 10,100,1000,10000 to create 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;

}

public static int[] sortArray(int size)//O(N)

{

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

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

{

generatedArray[i] = i;

}

return generatedArray;

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | inOrderWithoutRecursion | searchIterative | arrayToTree | Count | treeToArray | printNodesAtDepth |
| 10 | 269493 | 3282 | 8387 | 3647 | 3188 | 2553 |
| 100 | 1231135 | 4376 | 21516 | 10940 | 11305 | 156079 |
| 1000 | 10179466 | 6200 | 143316 | 97368 | 83875 | 79499 |
| 10000 | 54745720 | 21881 | 725333 | 97368 | 412810 | 69288 |

**Conclusion:** I was able to understand how to properly modify, count, convert binary search tree from sorted array and back. Traversing through the binary search tree using inorder traversal and print whats inside the binary search tree at a certain depth. This was a great review for me as it helped me remember the aspects of the binary search trees.

import java.util.Scanner;

import java.util.Stack;

/\*\*

\* Objective - To use and understand binary search trees

\* Date Last Modified - 2/26/18

\* Course - Data Structures

\* Lab 3

\* Instructor - Olac Fuentues

\* Teaching Assistants - Zakia Al Kadri

\* @author Isaias Leos Ayala

\*/

public class bst\_ops

{

/\*\*

\* Insert a node into the Binary Search Tree

\* @param T Binary Search Tree

\* @param i integer your inserting

\* @return

\*/

public static bstNode insert(bstNode T, int i)

{

if(T == null)//if null

{

T = new bstNode(i);

}

else if(i < T.item)//to the left

{

T.left = insert(T.left, i);

}

else//to the right

{

T.right = insert(T.right, i);

}

return T;

}

/\*\*

\* Recursive method that searches for integer in a binary search tree

\* @param T Binary Search Tree

\* @param i integer being searched

\* @return

\*/

public static bstNode search(bstNode T, int i)

{

if(T == null || T.item == i)//if item is found

{

return T;

}

if(i < T.item)//search left if true

{

return search(T.left, i);

}

else//search right if false

{

return search(T.right, i);

}

}

/\*\*

\* Find the minimum integer in the Binary Search Tree

\* @param T Binary Search Tree

\* @return smallest value

\*/

public static bstNode min(bstNode T)

{

if(T.left == null)

{

return T;

}

return min(T.left);

}

/\*\*

\* Find the height of the Binary Search Tree

\* @param T Binary Search Tree

\* @return

\*/

public static int height(bstNode T)

{

if(T == null)

{

return -1;

}

int hl = height(T.left);//height of the left side

int hr = height(T.right);//height of the right side

if(hl > hr)

{

return 1 + hl;

}

return 1 + hr;

}

/\*\*

\* Deletes a node from the Binary Search Tree

\* @param T Binary Search Tree

\* @param i integer your deleting

\* @return

\*/

public static bstNode delete(bstNode T, int i)

{

if(T == null)

{

return null;

}

if(i < T.item)

{

T.left = delete(T.left, i);

}

else if(i > T.item)

{

T.right = delete(T.right, i);

}

else // i == T.item

if(T.left == null && T.right == null)

{

T = null;

}

else if(T.left == null && T.right != null)

{

T = T.right;

}

else if(T.left != null && T.right == null)

{

T = T.left;

}

else

{

bstNode succ = min(T.right);

T.item = succ.item;

T.right = delete(T.right, succ.item);

}

return T;

}

/\*\*

\* A form of traversing the Binary Search Tree

\* @param T Binary Search Tree

\*/

public static void inOrder(bstNode T)

{//Eg. 1,2,3,4,5

if(T != null)

{//Check left then check right if empty

inOrder(T.left);

System.out.print(T.item + " ");

inOrder(T.right);

}

}

/\*\*

\* A form of traversing the Binary Search Tree

\* Iterative Version

\* Eg. 1,2,3,4,5

\* @param T

\*/

public static void inOrderWithoutRecursion(bstNode T)

{

Stack<bstNode> stack = new Stack<>();//Create Generic Stack

bstNode root = T;//Make a copy to the reference

while(!stack.isEmpty() || root != null)//check if its empty

{

if(root != null)

{//not empty push the left side to the stack

stack.push(root);

root = root.left;

}

else

{//go to the right side

bstNode node = stack.pop();

System.out.print(node.item + " ");//Print the node value

root = node.right;

}

}

}

/\*\*

\* Iterative method that searches for integer in a binary search tree

\* @param T Binary Search Tree

\* @param i integer being searched

\* @return

\*/

public static bstNode searchIterative(bstNode T, int i)

{

bstNode root = T;

while(root != null)

{

if(root.item == i)//If item is found

{

return root;

}

if(i < root.item)//if smaller

{//check the left side

root = root.left;

}

else//if bigger

{//check the right side

root = root.right;

}

}

return root;//return the node if found

}

/\*\*

\* Converts the sorted array to a Binary Search Tree

\* @param T Binary Search Tree

\* @param array given sorted array

\* @param i index of the array

\* @return

\*/

public static int treeToArray(bstNode T, int[] array, int i)

{

if(T != null)

{

i = treeToArray(T.left, array, i);//insert left side

array[i] = T.item;

i++;

i = treeToArray(T.right, array, i);//insert right side

}

return i;

}

/\*\*

\* Creates a Balanced Binary Search Tree from a sorted array

\* @param array

\* @param begin

\* @param end

\* @return

\*/

public static bstNode arrayToTree(int array[], int begin, int end)

{

if(begin > end)

{

return null;

}

int mid = (begin + end) / 2;//Middle element

bstNode node = new bstNode(array[mid]);//Make middle element root

//Make the left subtree

node.left = arrayToTree(array, begin, mid - 1);//make it left child

//Make the right subtree

node.right = arrayToTree(array, mid + 1, end);//make it right child

return node;

}

/\*\*

\* Prints the nodes at the given depth

\* @param T

\* @param k Depth

\*/

public static void printNodesAtDepth(bstNode T, int k)

{

if(T == null)

{

return;

}

if(k == 0)

{

System.out.print(T.item + " ");

return;

}

else if(T.right != null || T.left != null)

{

printNodesAtDepth(T.left, k - 1);

printNodesAtDepth(T.right, k - 1);

}

}

/\*\*

\* Counts how many nodes are in the tree

\* @param T Binary Tree Node

\* @return

\*/

public static int count(bstNode T)

{

if(T == null)

{

return 0;

}

return 1 + count(T.left) + count(T.right);//Count left and right side

}

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

{

if(T == null)

{

return;

}

double xm = (x0 + x1) / 2;

double yn = y - y\_inc;

if(T.left != null)

{

StdDraw.line(xm, y, (x0 + xm) / 2, yn);

draw\_tree(T.left, x0, xm, yn, y\_inc);

}

if(T.right != null)

{

StdDraw.line(xm, y, (x1 + xm) / 2, yn);

draw\_tree(T.right, xm, x1, yn, y\_inc);

}

StdDraw.setPenColor(StdDraw.WHITE);

StdDraw.filledCircle(xm, y, 3);

StdDraw.setPenColor(StdDraw.BLACK);

StdDraw.circle(xm, y, 3);

StdDraw.text(xm, y, Integer.toString(T.item));

}

/\*\*

\* Pause the animation until the user presses ENTER

\*/

public static void pause()

{

Scanner s = new Scanner(System.in);

System.out.println("Press enter to continue.....");

s.nextLine();

StdDraw.clear();

}

public static void main(String[] args)

{

int x\_max = 100;

int y\_max = 100;

StdDraw.setXscale(0, x\_max);

StdDraw.setYscale(0, y\_max);

StdDraw.setPenColor(StdDraw.BLACK);

int[] Array1 =

{

10, 16, 13, 17, 20, 19, 15, 4, 14, 8, 9, 5, 2, 3, 7, 12, 18, 1, 6, 11

};

bstNode firstBST = null;

for(int i = 0; i < Array1.length; i++)

{//insert nodes into tree

firstBST = insert(firstBST, Array1[i]);

}

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

System.out.print("inOrder Iterative: ");

inOrderWithoutRecursion(firstBST);

int finding = 10;

bstNode found = searchIterative(firstBST, finding);//Search for the integer in A

System.out.println("");

System.out.println("Item Searched: " + found.item);

int[] Array2 =

{

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 22, 33, 44, 55, 66, 77, 88, 99, 1010

};//sorted array

pause();//pause tree drawing

bstNode secondBST = arrayToTree(Array2, 0, Array2.length - 1);

draw\_tree(secondBST, 0, x\_max, y\_max - 5, (y\_max - 10.0) / height(secondBST));//draw the binary tree

int size = count(firstBST);

int[] sortedArray = new int[size - 1];

bstNode secondBSTCopy = secondBST;

treeToArray(secondBSTCopy, sortedArray, 0);//array to tree

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

{

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

}//print the sorted array from the binary tree

System.out.println("");//new line

int depth = 4;

System.out.print("Depth Searched: " + depth);

printNodesAtDepth(secondBST, depth);//print nodes at certain depth

}

}

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: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_