CS 3310 Data and File Structure

Instructor Dr.Ajay K. Gupta

Name: Mariam Ghali

**Tree and Heap**

**Phase 1: objective**

1. The main goal of this assignment is to practice building trees and travers it and build the heaps.
2. Develop a high-performance solution
3. See the difference between Heaps and Binary search tree
4. Practice two different types of searching DFS depth first search and BFS breadth first search and see the effect of those 2 searches on the max and min heap
5. The difference between implicit and explicit implementation for the trees.

**Phase 2: Description**

The program contains 6 classes

1. The main class that will create the objects for the three more classes and it will read the file name “nameList” or List.txt for the seek of testing. In this file we should search for our name using 5 different kind of search in the three main data structure we have: maxHeap, minHeap, and BST. In MaxHeap and MinHeap we should search based on DFS and BFS. In the BST we will search using the idea of the BST itself. IF the vales we are searching for is smaller than the root, search in the left child if it’s exist. If the value is bigger search on the right child. In addition, the main class will calculate the time that each method consumes.
2. The Node class will create the proberties or the information we need to know about the node such as: the parent of the node, the left child, the right child, and string array to get the first and the last name, the last thing is an int size to keep track with the size of the tree.
3. In the node class I am also creating the three-main way to travers a tree: Preorde, inorder and postorder. In the preorder, we visit the current node “parent or root if it is the start of the tree” then the left child then the right child. In the postorder, we visit the left child then the right child then the parent. In the inorder, traversal we visit the left , then the parent then the right child.
4. In the MaxHeap class, the implementation of the maxheap is implicit implementation, which means that it uses an array. the insert method will add each name we read from the file in the tree. While we are adding the names we happify of the biggest values” name” will be at the root” the first position in the array”. in the Maxheap the tree should be a complete tree. After we add all the values and happifying it, we add the three-main traversal to print the tree. Preorder, postorder and inorder. Then there are the 2-main search method: BFS and DFS.
5. In the BFS the maxheap use the queue to go to the tree level by level. First, we visit the root and add it to the queue, then e check if the queue is not empty, we remove the vales and recursively call the function to the leftchild then to the right child and repeat the same process again. In the DFS we use the stack and t goes in depth instead of level by level. So, it moves from the root to the left most child then until it reach the leaf then it move back to the parent then to the right child and so on. Using stack will be the reverse technical of queue so it can pop up the child before the parent itself. Both DFS and BFS should return the value we searched for, its parent, left child, right child , position and the level
6. In the fullname class is a way to get the number of the names we have an having access to the first and last name.
7. In the min heap, same as max heap but instead we heapifiy by the smallest value.in addition we call the main traversal of the tree, postOrde, inorder and preorder. Minheap is and explicit implementation for the tree which means that we add node to the tree not using the array implementation.
8. In the name search class, we search in the minheap tree for our names in the two main ways we have: BFS and DFS. In the BFS we use Queue as we did in the max heap but instead of using index for the array we use nodes. We start from the root then we move either to the left of the right. We compare the value we are searching for with the value in the tree if we found it we return its value, its parent, left child, right child , position and the level. Same with the depth instead if searching level by level we search with the depth of the tree.
9. In the Binary search tree we use the explicit implementation for the trees “using nodes”. BSt is complelety different from heap, It doesn’t have to be complete tree or balanced because of the way we implement it.f So basically BST is if the we add the values based on comparing if the value we are adding is smaller than the root, then add to the left child if it’s exist. If the value is bigger than the root we add to the right child. Same as searching, if the value we are searching for is bigger than the root so recursively call the right child if it is smaller call the left child.
10. **Pseudocode**
11. Main method
    1. try

Declare int numberOf line to zero

Declare buffer writer

Declare a buffer for reading file

Declare double variables for time

Create objects for the classes

read the file Lsit.txt

WHILE(not the end of the file )

Split the names by the tap

call the happify method for minheap

call the happify method for the maxheap

call the insert method for BST

end of WHILE

call the searchBFS method for minHeap

call the searchDFS method for minheap

call the searchBFS method for maxheap

call the searchDFS method for maxheap

call the searchBST method for BST

close file reader

close file write

catch

file not found exception

1. in the node

Declare a node for parent

Declare a node for left child

Declare a node for the right child

Declare a int for the size of the tree

Declare a string array of the first and last name

Preorder()

Visit parent or the current nore

Left child

Right child

Inorder()

Left child

Parent

Right child

postOrder()

Left child

Parent

Right child

1. Max heap class

Declare int lastindex == zero

Declare array of the object fullname

Declare an object of buffer writer

Declare a contractor

searchBFS(String first ,String last)

For i=0 until I < the length of the aarray

IF the value are the same

Print to a file the values we found

IF the index of the parent are not out of bound

Print the value of the parent

If the index of the left child is not out of bounds

Print the value in the left child

IF the index of the right child is not out of bound

Print the value in the right child

Level is the floor of the log i base 2

Position is index+1 – 2 to the power of level

Print Position and level

SearchDFS(String first,String last)

Decalare Stack

Declare int index == zero

If the index > arrayLenght or there is no value for array[index]

Return

Add the first element to the stack

WHILE the stack is not empty

Pop the last element we add

IF the value are the same

Print the value

IF the index of the parent are not out of bound

Print the value of the parent

If the index of the left child is not out of bounds

Print the value in the left child

IF the index of the right child is not out of bound

Print the value in the right child

Level is the floor of the log i base 2

Position is index+1 – 2 to the power of level

Print Position and level

END OF IF

IF the index of the left child are not out of bound or the value at that index is not null

Recursive call the function with the left child

IF the index of the right child are not out of bound or the value at that index is not null

Recursive call the function with the right child

Add(Sting first, String last )

IF the lastAddindex >= array length

Return -1

ELSE

Add the value to the array

Return the last index we added;

END OF IF

Heapif(index)

Declare a String array of length 2 for the last and first name

IF the index is < 0 or index> arraylength

Return -1

IF index of parent < zero

Return;

If ( value of parent is smaller than the child )

Swap the values

END OF IF

PreorderTraversal(index)

If the array is empty

Return;

Print myself

If the left child is out of bounds

Return

If array(index of left child) is not null

Recursive call for preoder with the left child

If array(index of right child) is not null

Recursive call for preoder with the right child

PostorderTraversal(index)

If the array is empty

Return;

If the left child is out of bounds

Return

If array(index of left child) is not null

Recursive call for preoder with the left child

If array(index of right child) is not null

Recursive call for preoder with the right child

Print myself

InorderTraversal(index)

If the array is empty

Return;

If the left child is out of bounds

Return

If array(index of left child) is not null

Recursive call for preoder with the left child

Print myself

If array(index of right child) is not null

Recursive call for preoder with the right child

1. MinHeap class

Node insert(Node temp, String first , String last)

IF root ==null

Create a root

Add the value to the data array

If the left and the right child of temp is not null

IF the module of the size of the sub tree is zero

Increase the size of the subtree

Return a recursive call for left child of the temp

ELSE

Increase the size of the subtree

Return a recursive call for left child of the temp

END OF IF

IF left child of the temp node is null

Create a node and make it the left child

Assign the first and the last name to the node

Return the left child ot the node temp

ELSE

Create a node and make it the right child

Assign the first and the last name to the node

Return the right child to the node temp

upHeapify(Node x)

Declare a string array with size 2 for the first and the last name

IF the parent of the ode is not null

IF value of the parent > value of the node

Swap

Recursive call for the parent to check the next parent

ENF OF IF

END OF IF

postOrder()

if root is not null

call postoder method from node starts from root

preOrder()

if root is not null

call preoder method from node starts from root

inOrder()

if root is not null

call inoder method from node starts from root

1. fullName

Declare String for last and first name

Contactors for initializing first and last name

1. BinarySearchTree

Contactors for buffer writer

Node insert(node temp, last , first )

IF root == null

Create a root for the tree

IF the value we need to add > the parent

IF the right child to the parent is empty

Create a right child

Assign the value of the last and first name

ELSE

Return Inset (right child ,last,first)

END OF IT

ELSE

IF the LEFT child to the parent is empty

Create a left child

Assign the value of the last and first name

ELSE

Return Inset (left child ,last,first)

searchBFS

IF valueof temp == first name

Print the name

Print the parent

Print the left child IF exists

Print the right child IF exist

Print the level and the position

ELSE IF the value of the temp > first

Recursive call with the left child

Else if the value of the temp < last

Recursive call with the right child

END OF IT

postOrder()

if root is not null

call postoder method from node starts from root

preOrder()

if root is not null

call preoder method from node starts from root

inOrder()

if root is not null

call inoder method from node starts from root

**Thermotical analysis**

We have 2 different kinds of implementing the tree explicit and implicit. In the project, we made explicit implementation for min heap and BST and implicit for maxheap. The min heap takes a constant time to insert and search where the value should be but it has o(h) for heapifying, time proportional to the height of the tree. For traversing the tree either with preorder, inorder or post order for both min and max heap, the time complexity is linear O(n). The time complexity for the searching a value in the min heap is linear to O(n). BFS for in heap using queue to go level by level of the tree and DFS using stack.

In the maxheap, it uses the array as an implementation. The time complexity for insertion is constant but the time complexity if heapifying is propositional to the height of the tree O(h). For both DFS and BFS the time complexity is O(n) because every time we add a value we have to compare it with it is parent until we get the largest value at the root of the tree.

In the BST, inserting a value requires to visit each node greater than or smaller than depends of the value itself before we insert. So, if the value is greater than the root we will recursively look at the right child, If the value is smaller than the root we look at the left child we will continue until we find that the next left or the right child is null so we create the node and add it. So, the time complexity of insertion is proportional to the height of the tree. The height of the BST can be huge constant because the BST trees are not complete or balanced trees. For the searching value of the BST it is like insert instead we actually compare the value we are searching for with the data in the tree. So, the time complexity of searching it proportional to the height of the tree O(h). based on the result of the three data structure, As the input increases, searching for a value usng BST is the best data structure we could use. This is also proved by using the empirical analysis.

**The empirical analyses**

It agrees with the thermotical analysis by showing that the graph of the three data structure with the different kinds of searching. Based on the graph, as the input increases, the BST seems to be the best for searching the name. Then it shoes that implicit implementation using Maxheap seems to be little faster than the min heap. The min heap for searching is similar to max heap but it takes more time however it is still linear.