**Assignment 4**

2017-11-10

By: Bryan Greener

Language: C#

**Introduction**

**File Contents**

**a4.sln**

This is the main solution file that can be opened in Visual Studio. The solution was created in Visual Studio Community 2017 using C#.

**Algorithm Analysis.docx**

This is a MS Word document that you are currently reading. This document contains algorithm analysis both on a pseudocode and graphical level.

**Analysis Data.xlsx**

This is a MS Excel document that I used to generate the graphs used in the analysis word document. The values in this document were pulled from tests while running the program.

**a4.xml**

XML Documentation file that can be opened in browser in order to see a Javadoc style set of comments for this program.

**BENCHMARK 2017-11-09 RUN1.txt**

Old benchmark file that had benchmark data from an older set of tests where I was using a different method of searching MinHeap.

**BENCHMARK 2017-11-09 RUN1.txt**

This is the most up to date benchmark that has been run on the program using my final algorithms.

**namelist.txt**

This file contains the names given to us to be used as a sample set of data. This file is a backup and is not actually read in by the program.

**namelistSMALL.txt**

This file contains a small set of sample data that I used to determine if my trees were being built correctly. Again, this is only a backup file that isn’t read in by the program.

**\a4\bin\Debug\**

This folder contains files generated at runtime.

**\a4\bin\Debug\[1, 10, ... , 1000000].txt**

These are text files that are filled with random first and last names for benchmark tests.

**\a4\bin\Debug\a4.exe**

Run this executable to run the application. This opens in a windows console window.

**\a4\bin\Debug\namelist.txt and namelistSMALL.txt**

These two files are the sample data that gets passed into the program. Currently the program is only set up to read in namelist.txt so please put any first/last name sets into that file.

**\a4\bin\Debug\output.txt**

This is a neat output containing all tree traversals and searches including node positions for each of the data types.

**Instructions**

1. Please navigate to \a4\bin\Debug\ and run a4.exe. This will open a console window with a UI for you to navigate through.
2. Printing all items will give you the tree traversals requested however they get sent to \a4\bin\Debug\output.txt since they would fill up the console window otherwise.
3. Searching for a name from the UI displays the result to the console window since it doesn’t print off a book worth of data.
4. Please read the file contents above for any further clarification.

**Algorithm Analysis**

**MinHeap**

**Insert(Node B, Value val)**

Create NEW tempNode

IF(B == null)

return new Node(Value = val, subtree = 1)

IF(B.LeftChild == null OR B.RightChild == null)

IF(B.LeftChild == null)

tempNode = B.LeftChild = Insert(B.LeftChild, val)

tempNode.Parent = B

ELSE

tempNode = B.RightChild = Insert(B.RightChild, val)

tempNode.Parent = B

ELSE IF(B.LeftChild.Subtree < B.RightChild.Subtree)

B.LeftChild = Insert(B.LeftChild, val)

ELSE

B.RightChild = Insert(B.RightChild, val)

WHILE(tempNode.Parent != null AND tempNode.LastName < B.LastName)

Swap tempNode and B nodes

tempNode = B

B.Subtree = B.LeftChild.Subtree + B.RightChild.Subtree + 1

My method of inserting is slightly different from the example given in class. I instead insert a new node when the current position is null instead of returning from the statement when at null. On my way down the heap, I am always checking if either child is empty and if so, I then check which one is null, favoring left child if both are null. Otherwise, if neither child is empty, I compare the subtree sizes of the children and move toward which ever subtree is smaller in order to keep the tree balanced. Then after inserting, I move back up the tree with the new node until it is not a smaller value than its parent.

This operation is theoretically O(h) where h is the height of the tree and in cases where the tree is balanced it is O(log(n)) time. My implementation takes O(h) time to travel down for an insert then another O(h) time to travel back up the heap with the new node. Since 2h is the same as h and this tree maintains its balance, the total time for insertion is O(log(n)) where n is the number of nodes in the tree.

**UpHeapify(Node B)**

IF(B.Parent == null) return B

IF(B.Value < B.Parent.Value)

Swap B and B.Parent nodes

Upheapify(B)

ELSE Return B

This algorithm is pretty standard as it just takes a node up the heap until the node’s value is not less than its parent. This algorithm will run in O(h) time and since the tree is balanced it runs in O(log(n)) time.

**DownHeapify(Node B)**

IF(left subtree < right subtree)

IF(B value < left value)

Swap B and B.LeftChild nodes

DownHeapify(B)

ELSE return B

ELSE IF(right subtree > left subtree)

IF(B value < right value)

Swap B and B.RightChild nodes

DownHeapify(B)

ELSE return B

ELSE return B

In order to move a node downward while maintaining balance, this algorithm has to both compare the subtree sizes and verify that it maintains our minheap by keeping the current node larger than both its children. This algorithm also runs in O(h) time but since our heap is balanced, the time complexity is O(log(n)) again.

**BFS(Node B, String search)**

Enqueue B

WHILE queue isn’t empty

B = Dequeue

IF(B.Value == search) return temp

IF(B.LeftChild != null) Enqueue(B.LeftChild)

IF(B.RightChild != null) Enqueue(B.RightChild)

Since this algorithm moves from left to right along each row— and since it is possible that our search item is that the far bottom right node of the heap— the worst case time complexity comes out to O(n) where n is the total number of nodes in the heap.

**DFS(Node B, String search)**

IF(B != null)

IF(B.value == search) return B

ELSE

B = DFS(B.LeftChild, search)

IF(B == null) B = DFS(B.RightChild, search)

Return B

ELSE return null

This algorithm is a bit more standard since it is very similar to a tree traversal. Again, it is possible that the search node will be the very last node visited and therefore the time complexity is O(n).

**MaxHeap**

**Insert(Value val)**

Int last = last non-null index in Heap[]

Heap[last] = new Node(Value = val);

Int temp = last

last++

WHILE(Temp.Parent >= 0 AND Temp.Value > Temp.Parent.Value)

Swap temp and temp.Parent

Temp = Temp.Parent

This algorithm is close to the insert algorithm explained in class but has a couple differences. It starts by inserting a value at the last non null index in the heap, updating the pointer to the next insert position, then maintains heap by making sure that parent value isn’t larger than the current node. Since inserting to the available pointer is O(1) and traversing back up the heap is O(h) which, since this is a balanced heap, is O(log(n)), the total time complexity for an insert is O(log(n)).

Both the BFS and DFS algorithms used are exactly the same as in the MinHeap implementation however instead of referencing nodes, they reference pointer integers in the array. These pointers use helper methods for finding the parent, left child, and right child using the exact same function explained in class to get those indexes.

**Binary Search Tree**

All of the algorithms used in my binary search tree implementation are standard in the sense that they recursively move left or right depending on the comparison between the current node and the search/insert value. Because of this and because the search tree is not being balanced, both search and insert have a best case time complexity of O(log(n)) and a worst case of O(n).

**All Implementations**

In each of the implementations of trees used in this assignment I used helper methods for preorder, inorder, and postorder travsersals as well as a helper method that traverses each of the trees and assigns each node an x and a y value corresponding to their position in the tree. The tree traversal algorithms are standard recursive traversals however the AssignXY is more similar to DFS.

**AssignXY(Node B)**

Enqueue B

int x = 0, y = 0

WHILE queue isn’t empty

B = Dequeue

IF(B.Y != y)

y++

B.X = x = 1

IF(B.LeftChild != null)

B.LeftChild.Y = B.Y + 1

Enqueue(B.LeftChild)

IF(B.RightChild != null)

B.RightChild.Y = B.Y + 1

Enqueue(B.RightChild)

This method creates a queue just like in the Breadth First Search methods that were used earlier. It just checks if nodes are on the same row as each other and if they are then it increments x (position in row). Since it is so similar to the BFS we used earlier, this algorithm also runs at O(n).

**Graphical Analysis**

**MinHeap**: y = 0.0052x + 1.8349

**MaxHeap**: y = 0.0026x - 14.662

**BST**: y = 1E-09x2 + 0.0079x - 25.358

Above are the equations for each line in the “Build Times for Different Data Structures” graph. These lines show that each of these algorithms is close to the theoretical time complexity.

In this case, MinHeap’s trendline follows O(n) however it is more likely following O(nlog(n)) since our insertion alone costs O(log(n)) and we have to insert n items. In order to see a more pronounced nlogn line we would have to test for a much larger n which my computer cannot handle.

MaxHeap is a bit more sporadic and doesn’t show a very clear trendline however it should theoretically be following O(nlog(n)) as shown in the algorithm analysis section of this document. The observed runtime difference between this and the MinHeap using a binary tree is likely due to array accesses being faster than traversing nodes.

This implementation of a binary search tree also follows the theoretical time complexity closely. The trendline that I used shows the runtime to be somewhere around O(n2) which makes sense since this is not a balanced binary search tree. Since the BST isn’t balanced an insert in the worst case could take O(n) time for each of the n items thus giving us a total time complexity of O(n2).

**MinHeap BFS**: y = 0.0003x - 0.9604 **MinHeap DFS**: y = 0.0002x - 1.1882

**MaxHeap BFS**: y = 0.0002x - 0.487 **MaxHeap DFS**: y = 0.0001x + 0.1855

**BST**: y = 0.0012ln(x) - 0.0027

Above is a graph of observed search times and their trend line equations. Overall, each of these lines is very close to the expected time complexity. For both MinHeap search types and for both MaxHeap search types, the formula given by excel shows this as linear however the runtime is probably closer to nlogn however that trend would only be visible for much larger values of n. And again the BST search line is nowhere to be seen since it is running at logn time so it is buried at the bottom of the graph.