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).

**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).

MinHeap: y = 0.0048x - 10.173

MaxHeap: y = 0.002x - 0.928

BST: y = 2E-09x2 + 0.0067x - 21.629

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 = 7E-11x2 + 0.0002x - 0.0896 MinHeap DFS: y = 3E-11x2 + 0.0002x - 0.1351

MaxHeap BFS: y = 8E-05x - 0.0399 MaxHeap DFS: y = 0.0001x - 0.1447

BST BTS: y = 0.0012ln(x) - 0.0025

Above is a graph of observed search times and their trend line equations. Overall, these