Title: Heaps and AVL Tree

Author: Tolga Han Arslan

ID: 22003061

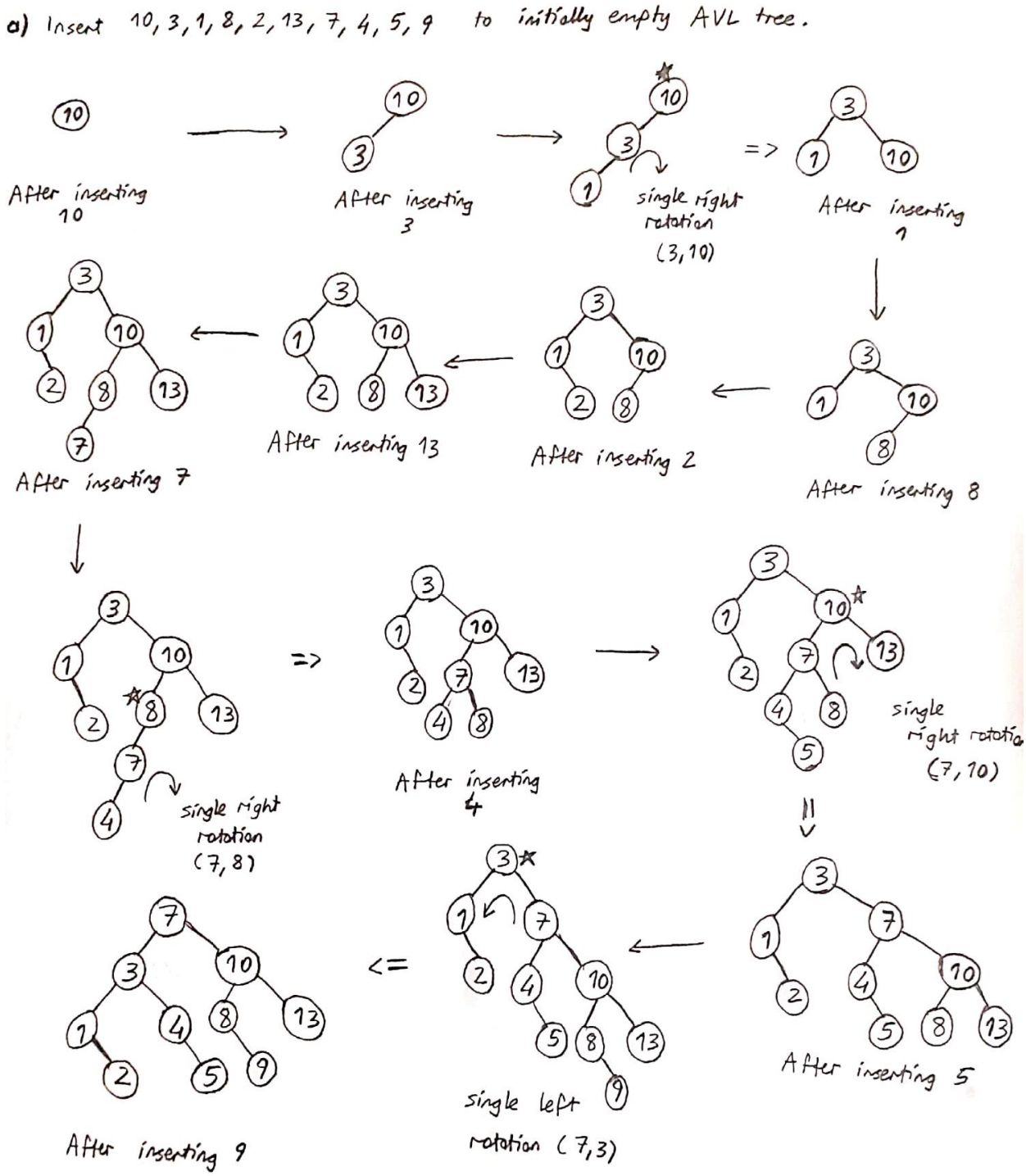
Section: 1

Assignment: 3

Description: HW3 Report

Question 1:

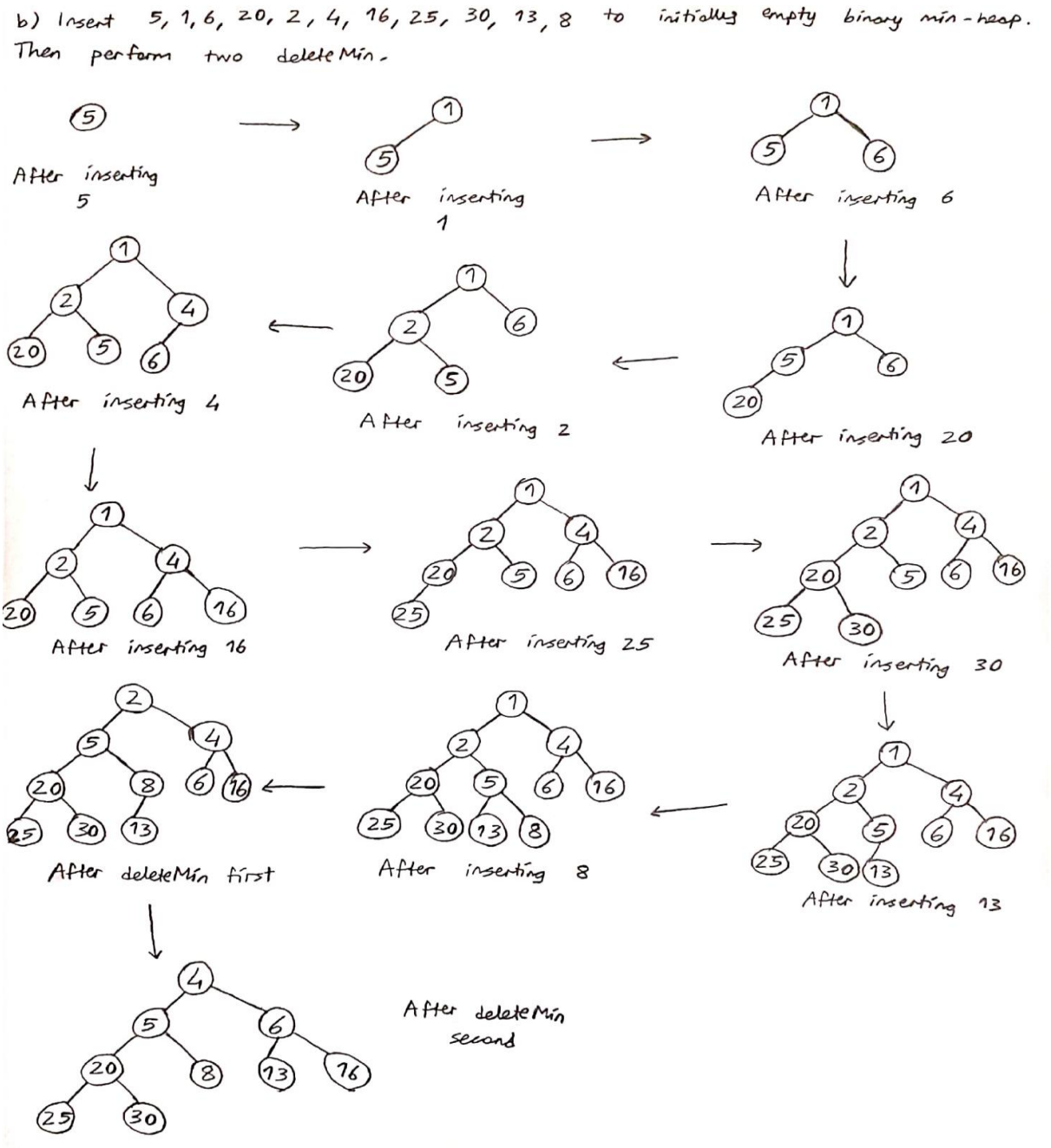
a) (10 points) Show the result of inserting 10, 3, 1, 8, 2, 13, 7, 4, 5 and 9 in that order into an initially empty AVL tree. Show the tree after each insertion, clearly labeling which tree is which.



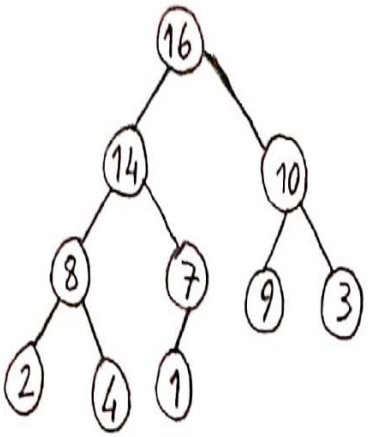
b) (10 points) This problem gives you some practice with the basic operations on binary min heaps. Make sure to check your work.

● Starting with an empty binary min heap, show the result of inserting, in the following order, 5, 1, 6, 20, 2, 4, 16, 25, 30, 13 and 8, one at a time, into the heap. By show, we mean, “draw the tree resulting from each insert.”

● Now perform two deleteMin operations on the binary min heap you constructed in part (a). Show the binary min heaps that result from these successive deletions, again by drawing the binary tree resulting from each delete.



c) (10 points) Consider a binary heap. Print the keys as encountered in preorder traversal. Is the output sorted? Justify your answer. Attempt the same question for inorder and postorder traversals.



Consider the binary max-heap given (from lecture slides).

Preorder: 16, 14, 8, 2, 4, 7, 1, 10, 9, 3

Inorder: 2, 8, 4, 14, 1, 7, 16, 9, 10, 3

Postorder: 2, 4, 8, 1, 7, 14, 9, 3, 10, 16

None of the traversal types resulted in a sorted output since heaps are not ordered similar to the binary search tree. Although parent nodes have always greater keys than its children in max-heap (reverse is true for min-heaps), there is no exact relation between left and right child. For example, node 8's left child 2 is smaller than right child 4 but node 10's left child 9 is greater than right child 3. Hence, there is no guarantee for sorted output in each traversal since child nodes are ordered in no particular order.

d) (10 points)

● Give a precise expression for the minimum number of nodes in an AVL tree of height h.

● What is the minimum number of nodes in an AVL tree of height 15?

height = 0 1 2 3 4 ...

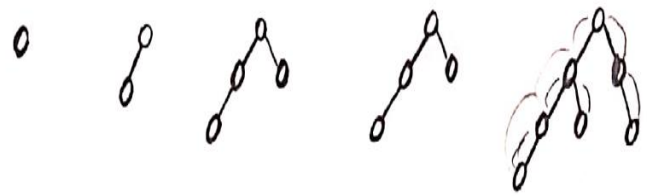
min. nodes = 0 1 2 4 7 ...

Min. no of nodes (height h)

$N(h) = 1 + N(h-1) + N(h-2)$

$N(0) = 0$

$N(1) = 1$ } base cases



$N(0) = 0$ $N(3) = 4$ $N(6) = 20$ $N(9) = 88$ $N(12) = 376$ $N(15) = 1596$

$N(1) = 1$ $N(4) = 7$ $N(7) = 33$ $N(10) = 143$ $N(13) = 609$

$N(2) = 2$ $N(5) = 12$ $N(8) = 54$ $N(11) = 232$ $N(14) = 986$

Minimum number of nodes in an AVL tree of height 15 = 1596.

e) (10 Points) Write a function in pseudocode that determines whether a given binary tree is a minheap.

```
Pseudocode that checks a given binary tree is a minheap
  isMinHeap ( inout treePtr : TreeNodePtr , inout index : int, in size : int )
  // Checks if the binary tree is complete and each node has higher key
  // than its parent (minheap)
    if( treePtr is NULL) { Empty tree, is a minheap }
  // Check if tree is complete by comparing size and index
    if ( index >= size ) { Not complete, is not a minheap }
  // check if node has higher value than its left or right child
    if ((treePtr -> leftChild -> item < treePtr -> item) AND treePtr -> leftChild
        is not NULL) OR ( treePtr -> rightChild -> item < treePtr -> item AND
        treePtr -> rightChild is not NULL)) {
           It is not a min-heap
        }
  // Check left and right subtree
    leftMinHeap = isMinHeap (treePtr -> leftChild, leftChildIndex, size)
    rightMinHeap = isMinHeap ( treePtr -> rightChild, rightChildIndex, size)
  // if both left and subtree are minheap, tree is a minheap
      if (leftMinHeap AND rightMinHeap)
          It is a minheap

treePtr = root                          leftChildIndex = 2 * index + 1
index = 0                               rightChildIndex = 2 * index + 2
size = total no of nodes
```

Question 2:

In this assignment a max-heap data structure is used to implement the heap sort algorithm. Max-heap is a complete binary tree that each node has greater key values than their respective children. On the other hand, there is no relation between child nodes key values, unlike binary search trees. This max-heap data structure has following member functions:

- isEmpty(): This function checks if the heap is empty.
- Insert(): This function is for inserting an item to heap, it puts the given item to the last location in the array and trickle it up to the its proper location in the heap. This procedure requires O(log(n)) comparisons since heap's height is approximately log(n).
- Maximum(): This function retrieves maximum value of the heap (root).
- popMaximum(): This function is for deleting maximum value from the heap. It retrieves and deletes the root item, moves the last item into the root, and trickles it down to its proper position by comparing parent and child nodes respectively. This procedure also requires O(log(n)) comparisons since heap's height is approximately log(n).
- heapRebuild(): This is the helper function for popMaximum function. It transforms semiheap into a heap.

In the heap sort algorithm, first step is creating a heap from the given array. This is done by inserting array elements to the heap structure. Then, the root item and last element in the heap will be swapped to place the maximum item to its correct position in the array. After this, last index of heap will be decremented and heapRebuild function converts the remaining semiheap into a heap by comparing and swapping child nodes with their parent nodes respectively. This step requires O(log(n)) comparisons since heap's height is

approximately log(n). Overall, this procedure is repeated until there is 1 item left in the heap part.

Input Size (n)	Comparison count theoretical	Comparison count experimental
1000	17954	12280
2000	39906	27665
3000	63810	43883
4000	87810	61327
5000	113618	78567

Figure 1.1: Theoretical and experimental comparison counts of heapsort

Heapsort is O(n*log(n)) algorithm since n items in the array must construct a heap and heap property must be maintained after swapping mentioned before. By looking at figure 1.1, theoretical values are compatible with this O(n*log(n)) growth rate. For example, when the input size is getting larger, comparison count growth is relative to the n*log(n) rate.

Theoretical comparison counts can be calculated by considering the heapsort algorithm as a two step process: constructing the heap and using rebuild function to converting semiheaps into a heap. For constructing, n elements must be inserted to heap and since the heap's height is approximately log(n), this leads to $\sum_{i=2}^{N} \log(i)$ comparisons. Variable i starts from 2 here since inserting root does not require any comparison. In the heapRebuild process, comparisons are in one path from node to the root. It requires approximately $\sum_{i=1}^{N} \log(i)$ comparisons, as similar reasoning of heap's height is approximately log(n).