**50.004 – Introduction to Algorithms**

**Problem Set 1**

**Zhang Tianqin 1004878**

**Question 1**

Diagram

Description automatically generated

**Question 2**

(i) **function** Decrease key*(A, i, k)*

**Require:** A[1..n] is a max heap.

**Require:** *i* is an integer, and 1 ≤ *i* ≤ *n*

**Require:** *A*[*i*] *> k*

1. *A*[*i*] ← *k Time complexity:* ***O(1)***
2. **while** *i*  and (*A*[*i*]*<A*[left(*i*)] or *A*[*i*]*<A*[right(*i*)]) **do** *Time complexity*: ***O(logn)***
3. mindex ofthe larger of A[left(i)] and A[right(i)]
4. Swap*A*[*i*] and *A*[m]
5. *i* ←m

Best case scenario: The changed node’s key is slightly smaller than the original value and the max-heap property of the tree is still maintained, or the changed node is a leaf node that a smaller value won’t affect the heap property. In this case, there will be 1 step to assign value of k which will takes the time complexity of O(1).

The worst case scenario is that we give the root a value which is smaller than the smallest key of the binary tree. Then to maintain its max-heap property, the tree will need to shift the node down to the leaf at height 0, which gives the same operation as max\_heapify(A, 1). In this case, there are nodes to be visited since the maximum height of the tree is , which resulting in a time complexity of O().

Thus, this algorithm has a time complexity of O().

(ii) **function** build\_min\_heap(*A*):

**Require:** *A*[1*..n*] is a max heap.

**Require:** *i* is an integer, and 1 ≤ *i* ≤ *n*

1. 𝐴. heap\_size ← 𝐴. length *Time complexity:* ***O(1)***
2. for 𝑖 from [] to 1 **do**
3. min\_heapify(𝐴, 𝑖);

**function min\_heapify(𝑨, 𝒊):**

**Require:** *A*[1*..n*] is an array with n entries.

**Require:** *i* is an integer, and 1 ≤ *i* ≤ *n*

1. 𝑚 ← index of smallest among 𝐴[𝑖], 𝐴[left(𝑖)], 𝐴[right(𝑖)]. Assume there are k steps in total
2. If 𝑚 ≠ 𝑖, then do
3. Swap 𝐴[𝑖] and 𝐴[𝑚] ; min\_heapify(𝐴, 𝑚);

Time complexity: The height of the heap h must satisfy , where n is the number of

elements. Given the height h of a binary tree, there are at most nodes.

build\_min\_heap(A) takes a total of 1+ steps.

1+ (Based on the fact that

Hence, this algorithm has a time complexity of O(n).

**Question 3**

* 1. Yes.

Let nh be the minimum number of nodes in a non-empty BST that has height h. The height of its left and right subtrees differ by no more than 2:

nh nh-1 nh-3  nh-1 nh-3 2nh-3

nh  2nh-3  4nh-6  8nh-9  … h-3i  0 (if h is even)

nh  2nh-3  4nh-6  8nh-9  … h-3i  1(if h is odd)

n0=1, n1, nh  Take log for both sides: , h , which is h*O*(log *n*). Thus T is a self-balancing tree.

* 1. Yes.

Let nh be the minimum number of nodes in a non-empty BST that has height h. The number of nodes in its left subtree and its right subtree are each no more than 90% of the total number of nodes in its subtree:

nh-1 nh,nh-2 nh-1

By recursion, nh nh-i

nh n0 =1 (if h is even)🡪 nh 🡪 h h

nh n1 =2 (if h is even)🡪 nh 🡪 h h­/2

h*O*(log *n*), and T is a self-balancing tree.

1. The minimum depth of the AVL tree is (h is even) or (h is odd).

Diagram

Description automatically generated

Justification:

As the AVL tree shown on the left, for every node x, the height of its left child and the height of its right child differs at most by 1. Denote by 𝑑(ℎ). 𝑑(ℎ) is the minimum height of a leaf in an AVL tree of height ℎ. One subtree of the root necessarily has height ℎ−1, and the other one has height either ℎ−2 or ℎ−1. Therefore

To get the smallest depth *d(h)*, we expand the shorter path, which is the path that has smaller height : The number of levels L of the tree: . Thus, when h is even, the depth of the tree is . And when h is odd, the depth of the tree is .

**Question 4** (Binary search trees)**.**

1. False. The deletion of node for a BST follows rules:

1. If the node is a leaf, simply remove the leaf node, the BST property still holds.

2. If the node has one child, then “promote the child to replace the deleted node.

3. If the node has two children, replace the node with its descendants (either the successor or the predecessor) that can maintain the BST property.

Based on the rules of deletion, given a Binary search tree as shown above:

Diagram

Description automatically generated with medium confidence

If we delete node with key of 4 first (node a), the root node becomes the successor of node a, which is the leaf that has the key of 6. After that, we delete node with key of 3, since it is only a leaf, a simple deletion is done on this leaf and the final BST is obtained.

Diagram

Description automatically generated

However, if we delete node with key of 3 first, since it is only a leaf, a simple deletion of this leaf is made. After this, we delete the node with key of 4, we just “promote” the node with key of 7 and the final BST is obtained.

It is obvious that the two obtained BST are different. Hence, the sequence of deletion elements matters, and the statement is false.

1. **function** IN\_ORDER*(T)*

**Require:** T is a binary search tree with n elements

1. While T.root NIL **do** *While loop runs n times until all the nodes are deleted*
2. r T.root *Time complexity: O(1)*
3. min tree\_min(r) *Time complexity: O(height of r)*
4. print (min.key)
5. tree\_delete(T, min) *Time complexity: O(height of T)*

Time complexity: The while loop will run n times until all the nodes are deleted. The tree\_min(r) method has a time complexity of O(height of r) which is O(height of T) in the worst case as r is the root node of the BST. While the tree\_delete(T,min) method has a time complexity of O(height of T). Let h denotes the height of the binary tree,

Thus, this algorithm has a time complexity of T(n)=O(n

**Question 5**

Diagram

Description automatically generated

**Question 6**

(i) False. We might need to rebalance more than one nodes. As the below example shows:

Diagram

Description automatically generated

The original AVL tree follows the rule that heights of the two child subtrees of any node differ by at most one. Assume that we delete the node T5 from the given AVL tree, after the deletion of node T5, operation to rotate y is needed to maintain the AVL property as the node Y and node T4 has a height difference of 2. However, after the rotation, the height difference between node x and node T4 is still 2, which means another rotation is needed. In this case, there are two rotations to make. This is because after we do the left rotate(y), the longest path of the tree doesn’t change. So the height difference between the left subtree and right subtree is still more than 1 and hence more rotations are needed.

(ii) **function** TreeItemToList(T, r, l)

**Require:** T is a binary search tree with n elements

**Require:** r is the root node

**Require:** l is a list

1. if r NIL **do**
2. TreeItemToList (T, r.left, l)
3. l.add(r) Time Complexity: O(n) since there are n elements in the BST
4. TreeItemToList (T, r.right, l)

**function** SPLIT*(T, k)*

**Require:** T is a binary search tree with n elements

**Require:** k is a key of one node in T

1. Tleft BST;
2. Tright BST;
3. for element in l **do** Time Complexity: O(n) since there are n elements in the list l
4. if element.key k **do**
5. Tree\_insert (Tleft, element) Time Complexity: O(height of Tleft)
6. else **do**
7. Tree\_insert (Tright, element) Time Complexity: O(height of Tright)
8. **return** Tleft, Tright

**function** Tree\_insert*(T, x)*

**Require:** T is a binary search tree with n elements

**Require:** x is an element to insert, 𝑥. key ≠ NIL; 𝑥. left = 𝑥. right = NIL

1. 𝑦←NIL;𝑧←𝑇.root
2. while 𝑧 NIL **do**
3. 𝑦←𝑧
4. if 𝑥. key < z. key **do**
5. 𝑧 ← 𝑧. left
6. else **do**
7. 𝑧 ← 𝑧. right
8. 𝑥. parent ← 𝑦
9. if 𝑇. root = NIL **do**
10. 𝑇. root ← 𝑥
11. else if 𝑥. key < 𝑦. key **do**
12. 𝑦. left ← 𝑥
13. else **do**
14. 𝑦. right ← 𝑥

The algorithm works by firstly adding all the elements of the BST to a list which has a time complexity of O(n). Then it iterates every element (n comparisons to make) of the list and insert it accordingly to the two BTS trees. T<= and T>. The insertion of the elements use the method of tree\_insertion(T, x), which keeps the AVL property of both T<= and T>, and it takes a time complexity of O(height of T).

Hence, the time complexity of the algorithm is : O(n)+O(nlogn)O(nlogn)

**Question 7**

Diagram

Description automatically generated

Based on the property of d-ary, the item at position 0 of the array forms the root of the tree, the items at positions 1 through d are its children, the next items are its grandchildren. Thus, the parent of the item at position i (for any i > 0) is the item at position ⌊(i − 1)/d⌋ and its children are the items at positions di + 1 through di + d.

1. **function** MAX\_HEAPIFY(A, i, d)
2. i←j
3. for k from 0 up to d-1
4. if d **do**
5. j=d
6. if j
7. swap A[i], A[j]
8. MAX\_HEAPIFY(A, j, d)

The running time is O(d) because at each depth we are doing d loops, and we recurse to the depth of the tree. We compare the *i*th node and each of its children to find the maximum value for all of the nodes. Then if the maximum child is greater than the th node, we switch the two nodes and recurse on the child.

1. **function** BUILD\_MAX\_HEAP(A, d)

**Require:** A is an array of length n

**Require:** d is an integer

1. h←length(A)
2. for k from D-ary-parent(d, i) down to 1 **do**
3. MAX\_HEAPIFY(A, i, d)

BUILD\_MAX\_HEAP(A, d) takes a total of 1+ steps.

1+ (Based on the fact that

Hence, this algorithm has a time complexity of O().