CS 225: Data Structures

Assignment 7

Group D1

Last Modified on April 15, 2022

Members	
Name	**Student ID**
Li Rong	3200110523
Zhong Tiantian	3200110643
Zhou Ruidi	3200111303
Jiang Wenhan	3200111016

Please turn over for our answers.

Exercise 4 is written by LI Rong.

Ex. 1 **Our Answer.** (This part is written by ZHONG Tiantian)

(i) Since in an AVL tree the left child is always smaller than its parent, and the right bigger, we can perform a recursive traverse to do this job. Note that the minimum should always exist as a left child of a node, and the maximum as a right child.

Our algorithm is described in Function MINIMUM and Function MAXIMUM.

Function MINIMUM(node)

```
if leftChild = Null then
    return node
end
return MINIMUM(leftChild)
```

Function MAXIMUM(node)

```
if rightChild = Null then
    return node
end
return MAXIMUM(rightChild)
```

(ii) Define a function CheckAVL, which computes the depth of the tree with root node and returns True if the tree rooted with node is an AVL tree.

Labelling the depth. By running function FindDepth, we label each node with their depth, assuming the root node has depth 0. Then iterating each node recursively by Depth-First Search and find the depth for all the nodes.

Compare the depth. By running function CompareDepth, we use Breadth-First Search to traverse the tree. When encountered with a leaf node, the function compares its depth with the maximum depth, i.e. the height of the tree. If the difference is larger than 1, the tree then can be determined not satisfying AVL tree properties.

See Algorithm 1.

Procedure FindDepth(node, parentDepth)

```
/* Find depth of each node using Depth-Frist Search */
depth[node] ← parentDepth + 1
if depth[node] > maxDepth then
    maxDepth ← depth[node]
end
if node is a leaf child then
    exit
end
if node has a left child then
    call FindDepth(leftChild[node],depth[node])
end
if node has a right child then
    call FindDepth(rightChild[node],depth[node])
end
```

Algorithm 1: AVL-Check

```
if root does not exist, i.e. an empty tree then
    return False
end
depth[root] ← 1
call FindDepth(root,depth[root])
append root to queue
while queue ≠ ∅ do
    node ← the front element in queue
    pop queue's front element
    if node is a leaf node then
        if depth[node] – maxDepth ≥ 2 then
            return false
        end
    end
    else
        append node's children to queue
    end
end
return true
```

Ex. 2 **Our Answer.** (This part is written by JIANG Wenhan, ZHONG Tiantian)

(i) Applying a search sequence from root to one leaf node (fixed directoin), then we return to the root and begin another search from root to another leaf node. Repeat such operation until all leaf nodes are traversed.

A structured description is provided in Algorithm 2.

Algorithm 2: Depth-Frist Search

```
// Initialize an empty list
list_of_elements ← []

Procedure DepthFirstSearch(node)
    append node to list_of_elements
    if node is a leaf child then
        exit
    end
    if node has a left child then
        append key[node] to list_of_elements
        call DepthFirstSearch(left_child[node])
    end
    if node has a right child then
        append key[node] to list_of_elements
        call DepthFirstSearch(right_child[node])
    end
/* Main */
begin
    call DepthFirstSearch(root)
    return list_of_elements
end
```

(ii) Applying a search sequence firstly among all child nodes of root node and add the children to the queue. The queue records the nodes to visit in the following steps.

Retrieve the top element (denoting it with e) in the queue, and **append** all the child nodes of e to the queue. Popping e, and repeat the operation until the queue is empty.

A structured description is provided in Algorithm 3.

Algorithm 3: Breadth-First Search

```
append root to visit_queue
while visit_queue ≠ ∅ do
    node ← the front element in visit_queue
    pop visit_queue's front element
    append node's children (if any) to visit_queue
end
return result_list
```

(iii) Create a queue recording the element in AVL tree in an ascending sequence. To get such a queue, we can simply call a BFS or a DFS. After this step the length of queue can be used to determine the median of the tree.

Note that since we are performing an inorder transversal, the elements in result list must be ordered ascending.

Algorithm 4: Find Median

```
visit_queue ← []
call Search(root) // whatever BreadthFirstSearch() or DepthFirstSearch()
if length is an odd number then
    return visit_queue[length/ 2 + 1]
end
else
    return 1/2 · (visit_queue[length/ 2] + visit_queue[length/ 2 + 1])
end
```

(iv) Still insist on Depth-First Search. But we will make the "digging" process stop when reaching the boundaries.

You may refer to Algorithm 5.

Algorithm 5: Range

```
// Initialize an empty list
list_of_elements ← []
Procedure DepthFirstSearch(node, shouldAppend)
    if node does not exist then
        exit
    end
    if shouldAppend = false ∧ key[node] > x then
        shouldAppend ← true
    end
    if shouldAppend = true ∧ key[node] < x then
        exit
    end
    call DepthFirstSearch(left_child[node],shouldAppend)
    if shouldAppend = true then
        append key[node] to list_of_elements
    end
    call DepthFirstSearch(right_child[node],shouldAppend)
/* Main */
begin
    call DepthFirstSearch(root, false)
    return list_of_elements
end
```

Ex. 3 **Our Answer.** (This part is written by ZHOU Ruidi)

(i) We consider the case for a single layer and the depth of a tree.

A Single Layer. For each level, we use binary search with $(b-1)$ nodes (since a root/vertex has at most b children); thus the comparison times for a single level is

$$\log(b-1) < \log b$$

Depth of the Tree. Because the minimum number of child nodes of a vertex is a, and all the leaf nodes lies in the same layer, the depth of the tree, H, is

$$H \leq \log_a(n-1) \leq \log_a(n-1) + (2 - \log_a 2)$$

Thus the total times is at most

$$\begin{aligned} T &\leq \log(b-1) \times H \\ &\leq \log(b-1) \times \log_a(n-1) \\ &\leq \lceil \log b \rceil \Big((\log_a(n-1) + (2 - \log_a 2) \Big) \end{aligned}$$

Hence proved.

(ii) From Ex. 3i we have the total number of comparison is

$$T = \lceil \log b \rceil \Big((\log_a(n-1) + (2 - \log_a 2) \Big)$$

By the O-notation,

$$2 \log b \in O(\log b) \tag{1}$$

At the same time, we know that because a is a constant,

$$\log_a \frac{n-1}{2} \in O(\log n) \tag{2}$$

Plus b is a constant, it's obvious that

$$\log b \cdot \log_a \frac{n-1}{2} \in O(\log n)$$

Hence adding equation (1) and (2) will produce the result

$$O(\log b) + O(\log n)$$