**Practical session No. 6**

**AVL Trees**

<table>
<thead>
<tr>
<th>Height-Balance Property</th>
<th>For every internal node ( v ) of a tree ( T ), the height of the children nodes of ( v ) differ by at most 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVL Tree</td>
<td>Any binary search tree that satisfies the Height-Balance property.</td>
</tr>
<tr>
<td>AVL Interface</td>
<td>The AVL interface supports the following operations in ( O(\log n) ): insert, search, delete, maximum, minimum, predecessor and successor.</td>
</tr>
<tr>
<td>AVL Height</td>
<td><strong>Lemma:</strong> The height of an AVL tree storing ( n ) keys is ( O(\log n) )</td>
</tr>
</tbody>
</table>

**Example of AVL:**

```
  44
 /  \
17   78
 /   /  \
 32  50   88
 /     /     \
2     48     62
```

[Diagram of AVL tree with keys 44, 17, 78, 32, 50, 88, 48, 62, and node heights]
Pictorial description of how rotations rebalance an AVL tree. The numbered circles represent the nodes being rebalanced. The lettered triangles represent subtrees which are themselves balanced AVL trees. A blue number next to a node denotes possible balance factors (those in parentheses occurring only in case of deletion).
Question 1

A node in a binary tree is an only-child if it has a parent node but no sibling node (Note: The root does not qualify as an only child). The "loneliness-ratio" of a given binary tree T is defined as the following ratio:

\[ LR(T) = \frac{\text{(The number of nodes in T that are only children)}}{\text{(The number of nodes in T)}}. \]

a. Prove that for any nonempty AVL tree T we have that \( LR(T) \leq 1/2 \).

b. Is it true for any binary tree T, that if \( LR(T) \leq 1/2 \) then \( \text{height}(T) = O(\log n) \)?

c. Is it true for any binary tree T, that if there are \( \Theta(n) \) only-children, all of which are leaves, then \( \text{height}(T) = O(\log n) \)?

Solution:

a. In an AVL tree just the leaves may be only-children, and therefore for every only-child in T, there exists a unique parent node that is not an only-child. The total number of only-children in T is at most \( n/2 \), which means that \( LR(T) \leq (n/2)/n = 1/2 \).

b. No. Given that \( LR(T) \leq 1/2 \), there may be \( n/2 \) only-children. This allows for a tree that is more than \( n/2 \) in depth.

c. No. There can be a tree such that \( \text{height}(T) = \Theta(n) \) though all only-children are leaves. Example:
**Question 2**

(a) Insert the following sequence of elements into an AVL tree, starting with an empty tree: 10, 20, 15, 25, 30, 16, 18, 19.

(b) Delete 30 in the AVL tree that you got.

**Solution:**

(a) Red dashed line signifies first part of double rotate action.

(b).
Question 3

In the class we have seen an implementation of AVL tree where each node $v$ has an extra field $h$, the height of the sub-tree rooted at $v$. The height can be used in order to balance the tree. For AVL trees with $n$ nodes, $h=O(\log n)$ thus requires $O(\log\log n)$ extra bits.

1. How can we reduce the number of extra bits necessary for balancing the AVL tree?
2. Suggest an algorithm for computing the height of a given AVL tree given in the representation you suggested in 1.

Solution:

1. In addition to the regular BST fields, each node $x$ will have 2-bits balance mark:
   
   $00 \equiv '/'$ – $h(x\text{.left}) > h(x\text{.right})$
   
   $01 \equiv '=\cdot'$ – $h(x\text{.left}) = h(x\text{.right})$
   
   $10 \equiv '\\cdot'$ - $h(x\text{.left}) < h(x\text{.right})$

2. We will follow the path from the root to the deepest leaf by following the ‘balance’ property. If a sub tree is balanced to one side, the deepest leaf resides on that side.

```java
int calcHeight(T)
    if T=null
        return -1
    if T.balance='/' or T.balance='=\cdot'
        return 1 + calcHeight( T.left )
    else
        return 1 + calcHeight( T.right )
```

Time complexity for height $h$ – $O(h)$ (in the previous representation we got the height of the tree in $O(1)$ and in a regular BST in $O(n)$)
**Question 4**

Level-order traversal of a tree is defined as visiting the nodes of the tree such that nodes with smaller depth are visited first (if the depth is equal then visit from left to right).

a. What is the sequence of level-order traversal in the following tree:

```
    F
   / \  
  B   H
 /   /  
A   D   G
 /  /   /  
C  E   J  I
```

b. Give an algorithm that prints the values of the nodes according to Level-order traversal.

c. Given an AVL tree T, is it always possible to build the same tree by a sequence of BST-insert and delete operations (with no rotations)?

**Solution:**

a. **Level-order traversal** - visit every node on a level before going to a lower level (this is also called Breadth-first traversal)

F, B, H, A, D, G, J, C, E, I

b. 

```java
PrintLevelOrder(AVLNode T) {
    if (T ≠ null) queue Q ← T
    while(!Q.isEmpty())
        n ← Q.dequeue()
        print(n.val)
        if (T.left ≠ null) Q.enqueue(n.left)
        if (T.right ≠ null) Q.enqueue(n.right)
}
```

c. Yes, by inserting the nodes according to the tree levels.

```java
rebuildAVL(AVL T) {
    queue Q ← nodesByLevels(T)
    newT ← new BST
    while(!Q.isEmpty())
        n ← Q.dequeue()
        newT.insert(n)
}
Question 5

Suggest an ADT containing integers that supports the following actions. Explain how these actions are implemented.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init()</td>
<td>Initialize the ADT</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert(x)</td>
<td>Insert x into the ADT, if it is not in ADT yet</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Delete(x)</td>
<td>Delete x from the ADT, if exists</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Delete_in_place(i)</td>
<td>Delete from the ADT an element, which is in the i&lt;sup&gt;th&lt;/sup&gt; place (as determined by the order of insertion) among the elements that are in the ADT at that moment.</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Get_place(x)</td>
<td>Return the place (which is determined by the order of insertion) of x among the elements that are in the ADT at that moment. If x does not exist, return -1.</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

For example, for the following sequence of actions:
Insert(3), Insert(5), Insert(11), Insert(4), Insert(7), Delete(5)

Get_place(7) returns 4, and Delete_in_place(2) will delete 11 from the tree.
Solution:

The ADT consists of 2 AVL trees.
- T1 stores the elements by their key.
- T2 stores the elements by the order of insertion (using a running counter \( \text{count} \)). To this tree new element will be inserted as the greatest from all elements in this tree. The nodes of this tree store also the number of elements in their subtree – \( x.\text{size} \). Each node has a pointer to his father.
- There are pointers between all the trees connecting the nodes with the same key.

For the example above the trees are:

\text{Insert}(3), \text{Insert}(5), \text{Insert}(11), \text{Insert}(4), \text{Insert}(7):

\begin{align*}
\text{Init()} & \quad \text{– initialize 2 empty trees. } \text{count} \leftarrow 0 \\
\text{Insert}(x) & \quad \text{– } \text{count} \leftarrow \text{count} + 1. \\
& \quad \text{Insert an element by key into T1, insert the element as the biggest to T2 (with } \text{count} \text{ as the key), and update the pointers. In T2 update the field } x.\text{size} \text{ in the insertion path. (The insertion is as in AVL tree)} \\
\text{Delete}(x) & \quad \text{– find the element in T1 (regular search), and delete it from both the trees. In T2, go up from the deleted element to the root and update } x.\text{size} \text{ for all the nodes in this path. (The deletion is as in AVL tree)}
\end{align*}
**Delete_by_place(i)** – find the $i^{th}$ element in $T2$ in the following way:

$x ← T2.root$

if ($x.size<i$) return //less than $i$ elements, do nothing

while ($x! = null$)
  if $x.left = null$
    $z ← 0$
  else
    $z ← x.left.size$

  if ($i ≤ z$)
    $x ← x.left$
  else if ($i = z + 1$)
    Delete($x$) and break
  else // $i > z+1$
    $i ← i - (z + 1)$
    $x ← x.right$

**Get_place(x)** – find $x$ in $T1$ (if it’s not there, return -1). Go by its pointer to $T2$, then calculate the index of $x$ in the tree – Go up from $x$ to the root. In this path sum the number of nodes that are in the left subtree of the nodes in the path that are smaller than $x$:

$place ← 1$

if ($x.left ≠ null$) $place ← place + x.left.size$

while ($x.parent ≠ null$)
  if ($x$ is left_child of $x.parent$)
    $x ← x.parent$
  else // $x$ is right_child
    $place ← place + 1 + x.parent.left.size$
    $x ← x.parent$

return $place$
**Question 6**

A clerk wants to save a list of his task records. Each task has a unique ID, and for each task he would like to be able to mark it as completed.

Suggest a data structure that supports the following operations in $O(\log n)$ time in the worst case, where $n$ is the number of tasks in the data structure:

1. **Insert(k, t)** - Insert a new task $t$ with id = $k$ to the data structure, at first mark the task as not completed.
2. **Update(k)** – Update task with ID = $k$ to be completed.
3. **FindDiff(k)** – Find the difference between the number of completed and incomplete ($|\#\text{of completed} - \#\text{of incomplete}|$) among all the tasks with ID smaller than $k$.

**Solution:**

The data structure is an AVL tree $T$ where each node $x$ represents a task and has the following fields (in addition to the regular fields of a node in an AVL tree):

1. $x.ID$ – task ID number (the search key for $T$)
2. $x.task$ – the task record
3. $x.isCompleted$ – 1 (completed) or 0 (incomplete)
4. $x.num_completed$ – the number of all completed tasks in the subtree rooted at $x$ (including $x$).
5. $x.num_incomplete$ – the number of all incomplete tasks in the subtree rooted at $x$ (including $x$).

**Insert (k, t)**

- create new node $x$
- $x.ID \leftarrow k$
- $x.task \leftarrow t$
- $x.isCompleted \leftarrow 0$ // (incomplete task)
- $x.num_incomplete \leftarrow 1$ // (a new node is always inserted as a leaf)
- $x.num_completed \leftarrow 0$
- AVL-Insert($x$) // with rotations that update new fields where necessary.
- for every node $v$ in the path from $x$ to the root do:
  - $v.num_incomplete \leftarrow v.num_incomplete + 1$

**Time complexity**: $O(\log n)$
Update (k)
- \( x \leftarrow AVL\text{-}search(k) \quad \text{// O(log}n\text{)} \)
- if \( x\text{.isCompleted} = 0 \)
  \[ x\text{.isCompleted} \leftarrow 1 \]
  for every node \( v \) in the path from \( x \) to the root do:
  \[ v\text{.num_incomplete} \leftarrow v\text{.num_incomplete} - 1 \]
  \[ v\text{.num_completed} \leftarrow v\text{.num_completed} + 1 \]
\[ \text{O(log}n\text{)} \]

\[ \text{Time complexity: O(log}n\text{)} \]

FindDiff (k)
- \( \text{sum}_c \leftarrow 0 \)
- \( \text{sum}_i \leftarrow 0 \)
- \( T \leftarrow \text{the root of the tree} \)
- // search for a node with ID = k:
  while (\( T \neq \text{NULL} \))
    if (\( T\text{.ID} < k \))
      \[ \text{sum}_c \leftarrow \text{sum}_c + T\text{.left.num_completed} \]
      \[ \text{sum}_i \leftarrow \text{sum}_i + T\text{.left.num_incomplete} \]
      if (\( T\text{.isCompleted} = 1 \))
        \[ \text{sum}_c \leftarrow \text{sum}_c + 1 \]
      else
        \[ \text{sum}_i \leftarrow \text{sum}_i + 1 \]
      \[ T \leftarrow T\text{.right} \]
    else
      \[ T \leftarrow T\text{.left} \]
  return \(| \text{sum}_c - \text{sum}_i | \)
\[ \text{O(log}n\text{)} \]

\[ \text{Time complexity: O(log}n\text{)} \]