# Practical session No. 5

## Trees

| Tree | ADT that stores elements hierarchically. Each node in the tree has a parent (except for the root), and zero or more children nodes. Access methods in the tree ADT:  
|      | - root() – returns the root of the tree.  
|      | - parent(v) – returns the parent of node v.  
|      | - children(v) – returns an iterator of the children of node v.  
|      | **Depth(node v)**  
|      | The depth of node v = the number of ancestors of v:  
|      | - depth(root) = 0  
|      | - depth(v ≠ root) = depth(v.parent) + 1  
|      | **Height(node v)**  
|      | The maximum height of a node v in T:  
|      | - height(v) = 0 , if v is a leaf.  
|      | - height(v) = 1 + maximum-height of a child of v, if v is not a leaf.  
|      | **Height(tree T)**  
|      | The height of the root of T  
| Binary Tree | Each node has at most two children, left-child and right-child.  
|      | - **Inorder** – inorder on the left sub tree, visit the current node and finally, inorder on the right sub tree.  
|      | - **Preorder** – visit the current node, preorder on its left sub tree and finally, preorder on its right sub tree.  
|      | - **Postorder** - postorder on the left sub tree, postorder on the right sub tree and finally, visit the current node.  
|      | - **Full Binary Tree** – A binary tree in which each node has exactly zero or two children.  
|      |  - A full binary tree with n leaves has n-1 inner nodes.  
|      |  - **Complete Binary Tree** - A binary tree in which every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.  
|      |  - A complete binary tree of height h has between $2^h$ and $2^{h+1}-1$ nodes.  
|      |  - The height of a complete binary tree with n nodes is $\lceil \log n \rceil$  
| k-Tree | Each node has k children at most. Representation option: 'left child - right sibling', each node has the following pointers:  
|      | - parent  
|      | - left-child - a pointer to the left most child.  
|      | - right-sibling - a pointer to the next sibling on the right. |
Question 1

For a binary tree $T$, let's define:

- $L(T)$ = Number of leaves in $T$.
- $D_2(T)$ = Number of nodes in $T$ that their degree is 2.

Prove that for a given binary tree $T$ with $n$ nodes, $D_2(T) = L(T) - 1$.

Solution:

Complete induction on the number of nodes in the tree:

**Induction base:**

$n = 1$ – a single node.

$D_2(T) = 0 = L(T)-1$

**Assumption:**

The theorem is true for a binary tree with $k < n$ nodes.

**Induction step:**

2 optional cases:

- **a. The root’s degree is 1**

  ![Diagram for Case a]

  **Case a:** The root's degree is 1.
  
  $D_2(T) = D_2(T_1)$
  
  $= L(T_1) - 1$
  
  $= L(T) - 1$

- **b. The root’s degree is 2**

  ![Diagram for Case b]

  **Case b:** The root's degree is 2:
  
  $L(T) = L(T_1) + L(T_2)$
  
  $D_2(T) = D_2(T_1) + D_2(T_2) + 1$ (the root)
  
  $= L(T_1) - 1 + L(T_2) - 1 + 1 = L(T) - 1$

Therefore, the theorem is true.

*Note:* In a full binary tree $T$, $L(T) =$ the number of inner nodes + 1.
**Question 2**

T is a binary tree with n nodes.
Any node x has the following fields:

- x.key - natural number
- x.left - pointer to the left child
- x.right - pointer to the right child
- x.val - natural number for general use.

maxPath(T) = the maximal sum of keys in a path from the root to a leaf.

a. Describe an algorithm for finding maxPath(T) in O(n).
b. Can you modify the algorithm so that it will print the maximal path as well?

**Example:**

```
1
/   \
2     4
/ \    / \ 
20 12  3  0
```

Max(T) = 23
Path = Left Left
Solution:

a. Finding maxPath(T).
Compute recursively for the left sub-tree and the right sub-tree and add the root's key to the maximal sum.

```java
maxPath( T )
    if (T = null)
        return 0
    lmax ← maxPath(left-sub-tree(T))
    rmax ← maxPath(right-sub-tree(T))
    m ← Max(lmax, rmax)
    return m + T.key
```

b. 

1. Finding maxPath(T).
   Same as before, only use x.val to remember actual path. x.val = 0 indicates that the maximal sum was accepted from the left sub-tree. x.val = 1 indicates the maximal sum was accepted from the right sub-tree.
2. Printing the path. For any node x, use x.val to decide which way to go next.

```java
PrintPath(T)
    While (T.left ≠ null or T.right ≠ null )
        if (T.val = 0)
            print("left ")
            T ← T.left
        else
            print("right ")
            T ← T.right
    T.val ← 0
    else
        T.val ← 1
    return m + T.key
```

Time complexity:

maxPath(T) – O(n) since every node was visited once.
PrintPath(T) – O(n) since every node in the path was visited once, and this number is bounded by n.
In total – O(n)
**Question 3**

Construct the tree from the following preorder and inorder travels:

**Preorder:** a,b,c,d,e,g,h,j,f.  
**Inorder:** c,d,b,a,h,g,j,e,f.

**Solution:**

From the preorder we can say that “a” is the root, that is why we can divide the inorder travel into two sub-trees [c,d,b] a [h,g,j,e,f] (in the preorder form it will be a [b,c,d] [e,g,h,j,f] )

```
  a
 /   \
|     |
|     |
|     c,d,b
|     h,g,j,e,f
```

In the left sub-tree from the preorder form ([b,c,d]) we can say that b is the root and again we could divide the left side in to additional two sub-trees [c,d] b [].

```
b
 /   \
|     |
|     |
|     c,d
|     h,g,j,e,f
```

In the same way we can also unfold c,d.

```
a
 /   \
|     |
|     |
|     b
|     h,g,j,e,f
```

```
b
 /   \
|     |
|     |
|     c
|     d
```

The next iterations will be:
Claim: from two travels on some tree where one of them is inorder we can reconstruct the tree.
One way of proving this claim is done using the process that was described in this question.

Question 4

Is it always possible to recreate a binary tree according to its Preorder and Postorder visits?
Prove if yes, or show a counter example if no.

Solution:

Not always.
Example: Preorder: AB   Postorder: BA
The tree can be one of the two:
# Binary Search Trees

<table>
<thead>
<tr>
<th>BST property</th>
<th>In a BST T, for any node x in T:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. If y is a node in the left sub-tree of x then key(y) &lt; key(x).</td>
</tr>
<tr>
<td></td>
<td>2. If y is a node in the right sub-tree of x then key(y) ≥ key(x).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minimal key</th>
<th>The key of the leftmost node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal key</td>
<td>The key of the rightmost node.</td>
</tr>
</tbody>
</table>

| Predecessor        | x is a node in a BST T. |
|--------------------| The predecessor of x is the node preceding x in an in-order traversal on T. |
|                    | If x has left child ⇒ the maximum in the left sub-tree of x. |
|                    | Otherwise, predecessor is the lowest ancestor of x whose right child is also ancestor of x. |

| Successor          | x is a node in a BST T. |
|--------------------| The successor of x is the node which succeeds x in an in-order traversal on T. |
|                    | If x has right child ⇒ the minimum in the right sub-tree of x. |
|                    | Otherwise, successor is the lowest ancestor of x whose left child is also ancestor of x. |

<table>
<thead>
<tr>
<th>Delete</th>
<th>Delete node x from a BST T:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If x is a leaf and the left child of his parent then x.parent.left ← null</td>
</tr>
<tr>
<td></td>
<td>If x is a leaf and the right child of his parent then x.parent.right ← null</td>
</tr>
<tr>
<td></td>
<td>If x has only one child, update x.parent to point to x's child instead of x. Update the parent of the child of x to be the former parent of x</td>
</tr>
<tr>
<td></td>
<td>If x has 2 children, replace x in node y, its successor (y has no left child) Note that y might have right child. If so, then apply the same process of deleting a node with only one child to y (see previous section).</td>
</tr>
</tbody>
</table>

**Example of BST:**

![Example of BST](image-url)
Question 5

Write a function to determine whether a given binary tree of distinct integers is a valid binary search tree. Assume that each node contains a pointer to its left child, a pointer to its right child, and an integer, but not a pointer to its parent.

Solution:

Note that it is not enough to write a recursive function that just checks if the left and right nodes of each node are less than and greater than the current node (and calls that recursively). You need to make sure that all the nodes of the sub tree starting at your current node are within the valid range of values allowed by the current node's ancestors. Therefore you can solve this recursively by writing a helper function that accepts a current node, the smallest allowed value, and the largest allowed value for that sub tree.

```java
isValid(Node root)
    return isValidHelper(root, -∞, ∞)

boolean isValidHelper(Node curr, int min, int max)
    if(curr.value < min || curr.value > max)
        return false
    if (curr.left != null && !isValidHelper(curr.left, min, curr.value))
        return false
    if (curr.right != null && !isValidHelper(curr.right, curr.value, max))
        return false
    return true
```

The running time of this algorithm is O(n).

Another possible solution would be to print the tree inorder, then go through the output and check that it is sorted. Running time is O(n).

Question 6

Given a BST T of size n, in which each node x has a unique key, and an additional field x.size - the number of the keys in the sub-tree rooted at x (including the root node x).

Suggest an O(h) time algorithm `Greater(T, k)` to find the number of keys that are strictly greater than k (h is the height of the binary search tree).
Solution:

```plaintext
Greater(T, k)
    keys ← 0
    while (T ≠ Null )
        if (k < T.key)
            keys ← keys + T.right.size + 1
            T ← T.left
        else if (k > T.key)
            T ← T.right
        else // k = T.key
            keys ← keys + T.right.size
            break
    return keys
```

Time Complexity: O(h)

Note: h=O(n) in the worst case and O(log n) in the average case (randomized tree).

Question 7

T is a BST of height h that contains one node with value a and one node with value b. Find an algorithm that returns all the keys in the range [a,b] (a<b) in O(h+k) time, where k is the number of keys in the range.

Solution:

**Range(a, b)**

1. Find nodes containing a and b in the tree, let $P_a$ and $P_b$ be the search path from the root to a and b respectively.
2. Let x be the node where $P_a$ and $P_b$ split.
3. for every node v in the search path from a to x do:
   - if (a < v)
     - print v
     - inorder(v.right)
4. for every node v in the search path from x to b do:
   - if (b > v)
     - inorder(v.left)
   - print v
**Time complexity:**

$P_a$ and $P_b$ are at most $O(h)$ long each.

In-order travel is $O(n)$ for a tree with $n$ nodes. We are printing $k$ numbers. So the inorder traversals cost $O(k)$. Total time $O(h+k)$

---

**Question 8**

There are two binary search trees $T_1$ and $T_2$ and their heights $h_1$ and $h_2$ respectively. All the keys in $T_1$ are smaller than all the keys in $T_2$. Assume all key values are distinct.

1. How can you merge $T_1$ and $T_2$ into one binary search tree in time $O(\min(h_1,h_2))$ so that the height of the merged tree will be $O(\max(h_1,h_2))$?
2. What would be the height of the merged tree?

**Solution:**

1. **case a:** $h_1 \leq h_2$
   - Extract from $T_1$ the element $v$ with the maximum key in $T_1$ in $O(h_1)$ time.
   - $v$.left $\leftarrow T_1$
   - $v$.right $\leftarrow T_2$
   - $v$ is the root of the new merged tree.

2. **case b:** $h_1 > h_2$
   - Extract from $T_2$ the element $v$ with the minimum key in $T_2$ in $O(h_2)$ time.
   - $v$.left $\leftarrow T_1$
   - $v$.right $\leftarrow T_2$
   - $v$ is the root of the new merged tree.

2. The height of the merged tree $= \max(h_1,h_2)+1$