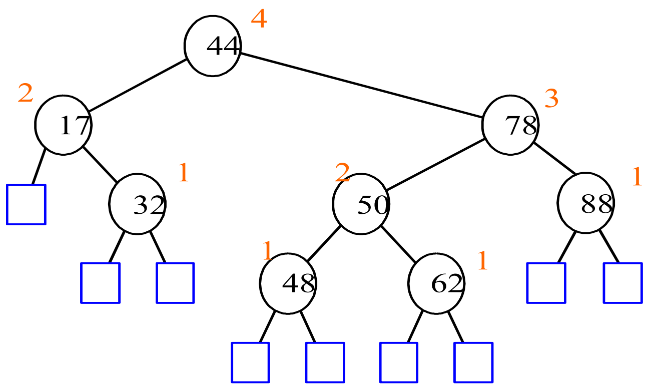
Practical session No. 6

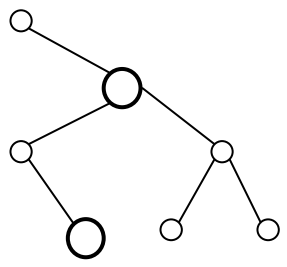
AVL Trees

Height-Balance Property	For every internal node *v* of a tree *T*, the height of the children nodes of *v* differ by at most 1.
AVL Tree	Any binary search tree that satisfies the Height-Balance property. Thus, it has $\Theta(\log n)$ height, which implies $\Theta(\log n)$ worst case search and insertion times.
AVL Interface	The AVL interface supports the following operations in *O(log n):* insert, search, delete, maximum, minimum, predecessor and successor.
AVL Height	**Lemma:** The height of an AVL tree storing *n* keys is *O(logn)*

Example of AVL:



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) = (The number of nodes in T that are only children) / (The number of nodes in T).

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

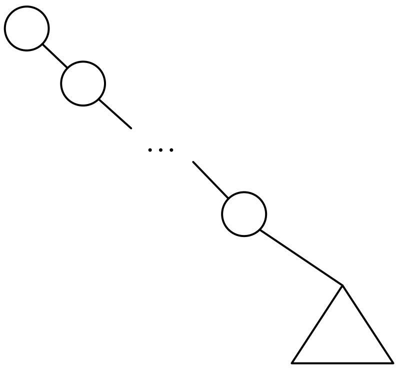
b. Is it true for any binary tree T, that if LR(T)≤1/2 then height(T)=O(lg*n*)?

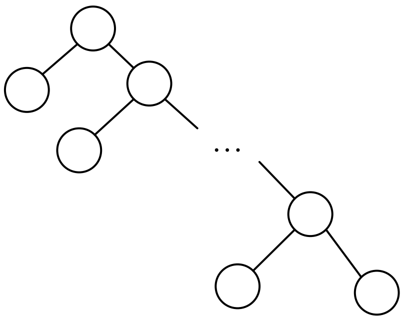
c. Is it true for any binary tree T, that if there are Θ(n) only-children, all of which are leaves, then height(T)=O(lg*n*)?

Solution:

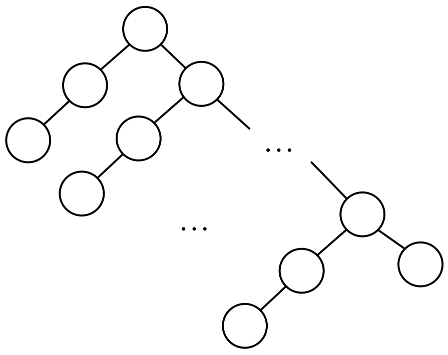
a. In an AVL tree just the leaves may be only-children, and therefore for every only-child in T, there exists a unique parent node that is not an only-child. The total number of only-children in T is at most n/2, which means that LR(T)≤ (n/2)/n = 1/2.

b. No.
Given that LR(T)≤1/2, there may be n/2 only-children. This allows for a tree that is more than n/2 in depth. Also, for every full Binary tree T, LR(T)=0.





c. No. There can be a tree such that height(T)=Θ(n) though all only-children are leaves.
Example:



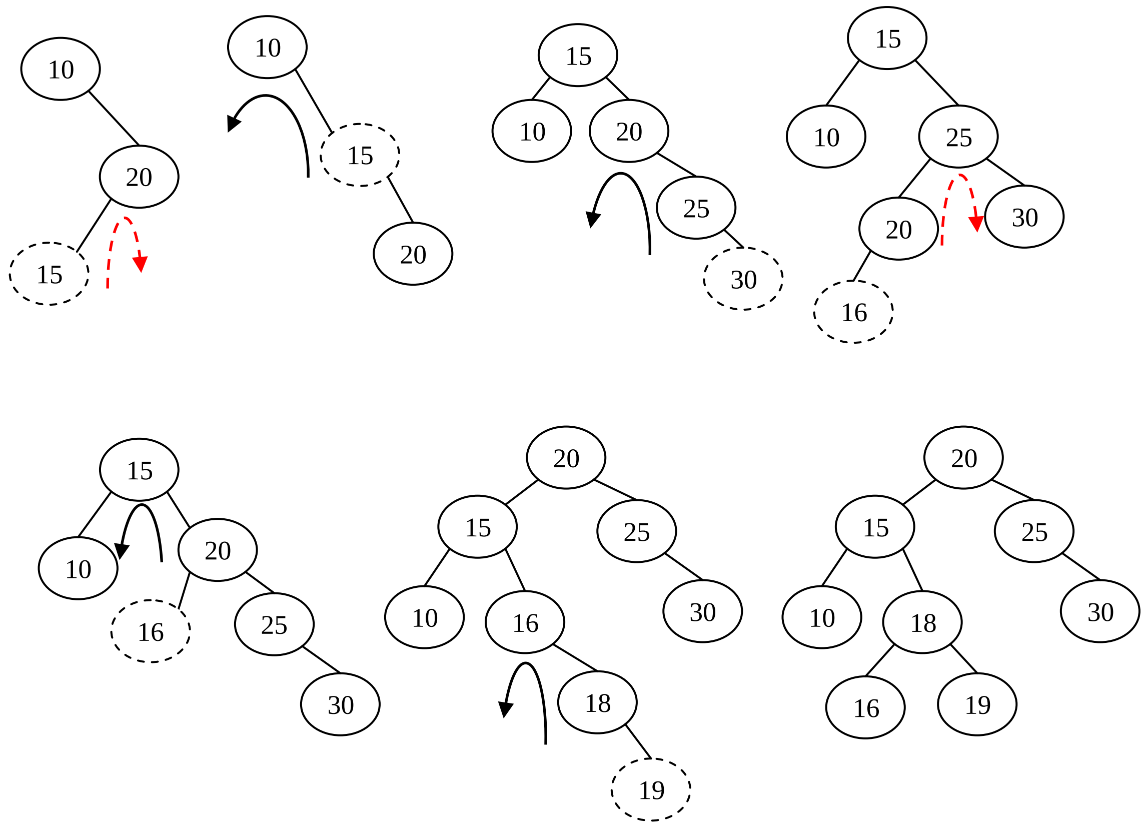
Question 2

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

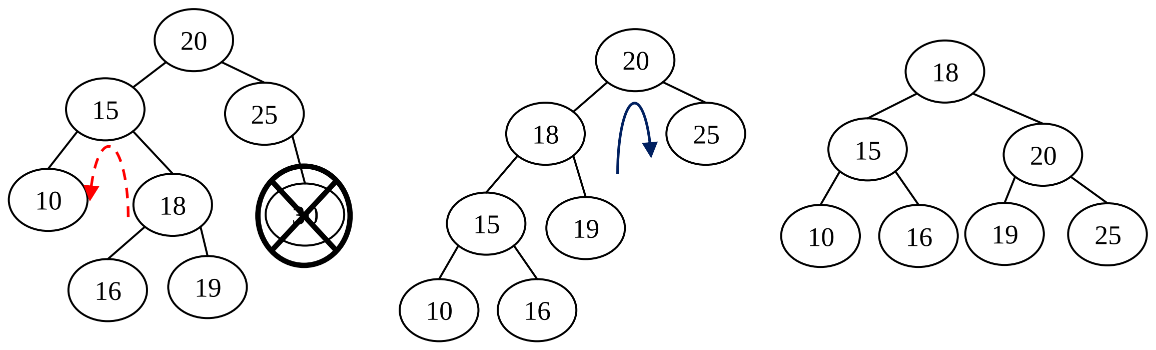
(b) Delete 30 in the AVL tree that you got.

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(logn) thus requires O(loglogn) 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 ≡ ’/’ – h(x.left) > h(x.right)
   01 ≡ ’–‘ – h(x.left) = h(x.right)
   10 ≡ ’\’ - h(x.left) < h(x.right)

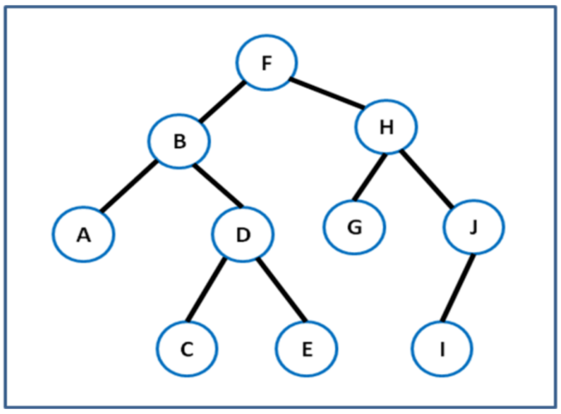
2. We will follow the path from the root to the deepest leaf by following the ‘balance’ property. If a sub tree is balanced to one side, the deepest leaf resides on that side.

```
calcHeight(T)
    if T=null
        return -1
    if T.balance=’/’ or T.balance=’-‘
        return 1 + calcHeight( T.left )
    else
        return 1 + calcHeight( T.right )
```

Time complexity – 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

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



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

```
rebuildAVL(AVL T)
    queue Q ← nodesByLevels(T)
    newT ← new BST
    while( ! Q.isEmpty())
        n ← Q.dequque()
        newT.insert(n)
```

Question 5

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

Init()	Initialize the ADT	O(1)
Insert(x)	Insert *x* into the ADT, if it is not in ADT yet	O(log n)
Delete(x)	Delete *x* from the ADT, if exists	O(log n)
Delete_in_place(i)	Delete from the ADT an element, which is in the i^{th} place (as determined by the order of insertion) among the elements that are in the ADT at that moment.	O(log n)
Get_place(x)	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.	O(log n)

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.

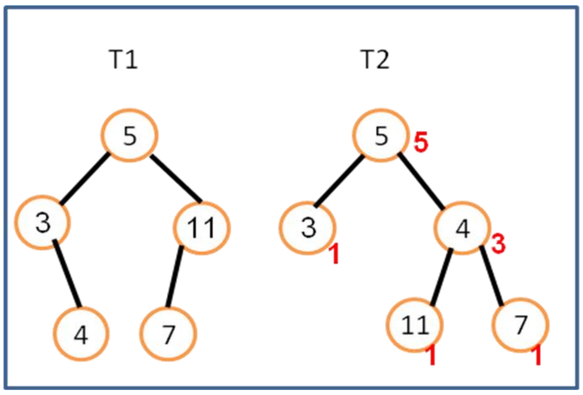
Solution:

The ADT consists of 2 AVL trees.

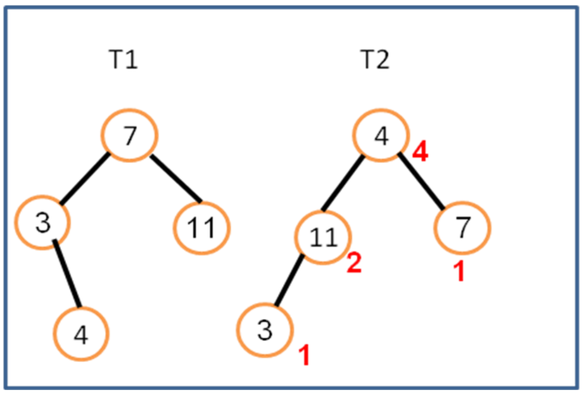
- T1 stores the elements by their key.
- T2 stores the elements by the order of insertion (using a running counter). To this tree new element will be inserted as the greatest from all elements in this tree. The nodes of this tree store also the number of elements in their subtree – x.size. Each node 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:

Insert(3), Insert(5), Insert(11), Insert(4), Insert(7):



Delete(5):



Init() – initialize 2 empty trees

Insert(x) – insert an element by key into T1, insert the element as the biggest to T2, and update the pointers. Update the in T2 the field x.size in the insertion path. (The insertion is as in AVL tree)

Delete(x) – 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.size for all the nodes in this path. (The deletion is as in AVL tree)

Delete_by_place(i) – find the ith element in T2 in the following way:

```
    x←T2.root
    if x.size<i return
    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, 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
    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

An electrician wants to represent a list of her clients' records (by their ID).
For each client we would like to mark whether he is a man or she is a woman.

Suggest a data structure that supports the following operations in O(log n) time in the worst case, where n is the number of persons (men and women) in the data structure when the operation is executed:

1. Insert(k,c) - Insert a new client c with id = k to the data structure, at first mark the client as a woman.
2. Update(k) – Update client with ID = k to be a man.
3. FindDiff(k) – Find the difference between the number of women and the number of men (| #of women - #of men |) among all the clients with ID smaller than k.

Solution:

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

1. x.ID - ID number (the search key for T)
2. x.client - the client record
3. x.gender- 1 (woman) or 0 (man)
4. x.women- the number of all women in the subtree rooted at x (including x).
5. x.men - the number of all men in the sub tree rooted at x (including x).

<u>Insert (k, c)</u>

```
•   create new node x                                        ⎫
•   x.ID ← k                                                 ⎪
•   x.client ← c                                             ⎬ O(1)
•   x.gender ← 1      //(woman)                              ⎪
•   x.women ← 1       // (a new node is always inserted as a leaf)
•   x.men ← 0                                                ⎭
•   AVL-Insert(x)  // With rotations that update new fields  ⎫
                   // where necessary.                       ⎬ O(logn)
•   for every node v in the path from x to the root do:      ⎪
        v.women ← v.women + 1                                ⎭
```

<u>Time complexity</u> : O(logn)

Update (k)

```
•  x ← AVL-search(k)    //  O(logn)
•  if x.gender = 1
       x.gender ← 0 (man)                                } O(1)
      for every node v in the path from x to the root do:  ⎫
          v.women ← v.women – 1                              ⎬ O(logn)
          v.men ← v.men +1                                   ⎭
```

Time complexity : O(logn)

FindDiff (k)

```
•  sum-m ← 0                          ⎫
•  sum-w ← 0                          ⎬ O(1)
•  T ← the root of the tree           ⎭
•  // search for a node with ID = k:
   while (T != NULL)                          ⎫
       if (T.ID < k )                         ⎪
           sum-m ← sum-m+T.left.men           ⎪
           sum-w ← sum-w+T.left.women         ⎪
           if (T.gender = 1)                  ⎪
               sum-w = sum-w + 1              ⎬ O(logn)
           else                               ⎪
               sum-m = sum-m + 1              ⎪
           T ← T.right                        ⎪
       else                                   ⎪
            T ← T.left                        ⎭
•  return |sum-w – sum-m|
```

Time complexity : O(logn)

Binary Search Trees (BST) -

Question 1

a. Given a BST T with unique keys, write an algorithm that prints the k smallest keys in T in O(*h*+*k*) time, where *h* is the height of T. If there are less than k nodes in T the algorithm should print all T's keys.

b. Let *x* be an arbitrary node in a given BST T. Write an algorithm that finds and prints *k* successors of *x* in O(*h*+*k*) time, where *h* is the height of T.
Example: If we run the tree contains all the numbers in the range 1…100 and key(x)=10 then we would want to print the numbers [11…11+k-1]. If k=4 this would mean [11…14].

Solution:

In the following algorithms, we will use a local variable *z* to hold the number of nodes already printed of the *k* that we want to print.

a. We'll Use a modified version of inorder to scan the tree.
First we'll print recursively print the z smallest keys from x.left. If we haven't printed k yet we'll print the root's key and then print the (k-z) remaining keys from the tree rooted in x.right.

```
inorder(x, k)
  if ( x = null )
      return 0
  z←inorder(x.left ,k)
  if ( z<k )
    print(x.key)
    z←z+1
    if ( z< k )
      z←z+inorder(x.right ,k-z)
  return z
```

b. If the node has a right sub-tree, then the first few successors must be there. The rest of the successors will be the successors of the first ancestor that is larger than *x*.

The **printSucc**(x,k) function prints at most k successors of x in the BST:

```
printSucc(x,k)
  if (k=0)
     return
  z ← inorder(x.right , k)
  if  (z < k )
    y←findSuccAncestor(x)
    print(y.key)
    z←z+1
    if  (z < k )
      printSucc(y, k-z)
```

The **findSuccAncestor**(x) Finds the first ancestor that is larger than *x*:

```
findSuccAncestor(x)
  y ← x.parent
  while ((y != NULL) & (x = y.right))
        x ← y;
        y ← y.parent
   return y
```

Time complexity:

findSuccAncestor – O(*h*) time

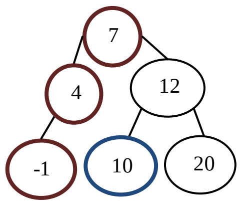
Total calls to inorder – O(*k*) time

Total time - O(*h*+*k*).

Question 2

T is a BST containing integer number values. For a given real value x **SumPaths(T,x)** is the number of paths in T or any subtree of T from the root down to some node for which the sum of path node keys is x.

a. Write an algorithm which calculates **SumPaths(T,x)**.
b. What is the running time of your algorithm?



Sumpaths(T,10)=2

Solution:

a. We'll use 2 functions.

- The first, **SumPathsRoot(T,x)**, will return the number of paths from the root down to some node for which the sum of path node keys is x.
- The second, **SumPaths(T,x)**, will return the value we want. It will first call **SumPathsRoot(T,x)** to get the number of desired paths beginning at the root T, and then call itself recursively to on T.left and T.right to find the desired paths beginning below T.

```
SumPaths(T,x)
  if (T = NULL)
    return 0
  y←SumPathsRoot(T,x)
  xL←SumPaths(T.left, x)
  xR←SumPaths(T.right, x)
  return y + xL + xR
```

```
SumPathsRoot(T,x)
  if (T = NULL)
    return 0
  xL←SumPathsRoot(T.left, x–T.key)
  xR←SumPathsRoot(T.right, x–T.key)
  if (x – T.key = 0)
    xL←xL+1
  return xL+xR
```

b. The **SumPaths(T,x)** algorithm's runtime is represented by the following formula:

$$\mathrm{T}(n)=\mathrm{T}(k)+\mathrm{T}(n-k-1)+1+\mathrm{S}(n)$$

with k being the size of the subtree rooted in T.left, and S(n) being the runtime of the **SumPathsRoot(T,x)** algorithm.

The **SumPathsRoot(T,x)** runtime is represented by the following formula

$$\mathrm{S}(n)=\mathrm{S}(k)+\mathrm{S}(n-k-1)+1$$

With k being the size of the subtree rooted in T.left. Assuming that S(0)=0 for simplicity (meaning that S(1)=1 for a leaf, which makes sense) we will prove by induction that S(n)=n.

The base of induction is covered by S(0)=0. We'll assume that S(m)=m for m<n.

$$\mathrm{S}(n)=\mathrm{S}(k)+\mathrm{S}(n-k-1)+1=k+n-k-1+1=n$$

We have $\mathrm{T}(n)=\mathrm{T}(k)+\mathrm{T}(n-k-1)+1+\mathrm{S}(n)\leq\mathrm{T}(k)+\mathrm{T}(n-k-1)+cn$

By induction $\mathrm{T}(n)=\Theta(n^2)$. Our algorithm runs in $\Theta(n^2)$ time.