Lecture: Trees-1

TDRK12 Data Structures, 7.5 credits

Vladimir Tarasov

7 February 2022

School of Engineering, Jönköping University

Jönköping University

Table of Contents

Trees

Definition

A **tree** is *recursively* defined as

- *a set of one or more nodes* where one node is designated as the **root** of the tree and
- all the remaining nodes can be partitioned into *non-empty sets* each of which is a **sub-tree** of the root.

Types of Trees

- General Trees
- Forests
- Binary Trees
- Expression Trees
- Tournament Trees

Root node The root node `R` is the *topmost node* in the tree. If `R = NULL`, then it means the tree is empty.

Sub-trees If the root node `R` is not `NULL`, then the trees T_1, T_2, and T_3 are called the sub-trees of `R`.

Leaf node A node that has *no children* is called the leaf node or the terminal node.

Edge The line connecting a node to any of its successors.

Path A *sequence of consecutive edges* is called a path.

Ancestor node An ancestor of a node is any *predecessor node* on *the path from root* to that node.

Descendant node A descendant node is any *successor node* on *any path* from the node to *a leaf node*

- To store simple as well as complex data.
- To implement other types of data structures like hash tables, sets, and maps.
- Trees are widely used for information storage and retrieval in symbol tables.
- Trees are used in compiler construction, database design, and file system directories.

- A self-balancing tree, Red-black tree is used in kernel scheduling to preempt massively multi-processor computer operating system use.
- B-trees are used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.

General Trees

- General trees are data structures that store elements *hierarchically*.
- The top node of a tree is the *root node* and each node, except the root, has a *parent*.
- A node in a general tree (except the leaf nodes) may have zero or more *sub-trees*.
- General trees which have 3 sub-trees per node are called *ternary trees*.
- The number of sub-trees for any node may be *variable*.
    - A node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

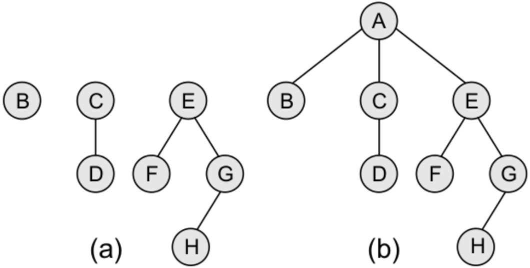
JÖNKÖPING UNIVERSITY

Definition

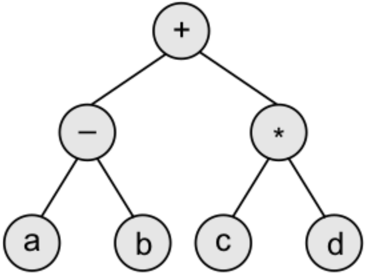
A forest is a *disjoint union of trees*:

- Obtained by deleting the root and the edges connecting the root node to nodes at level 1.
- Can be converted into a tree by adding a single node as the root node of the tree.
- Can also be defined as an *ordered set* of zero or more *general trees*.

- Every node of a tree is the root of some sub-tree.
- All the sub-trees immediately below a node form a forest.
- A forest may be empty because it is a set, and sets can be empty.



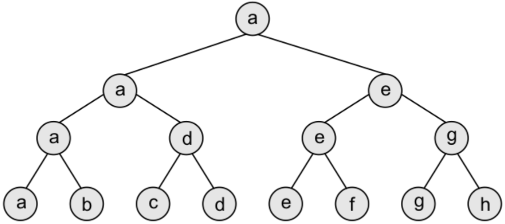
- Binary trees are widely used to store *algebraic expressions*.
- For example, consider the algebraic expression *Exp*:
  - $Exp = (a - b) + (cd)$
- This expression can be represented using a binary tree



Definition

In a tournament tree (also called a *selection tree*):

- Each *external node* represents a player
- Each *internal node* represents the winner of the match
  - between the players represented by its children



- Such tournament trees are also called *winner trees*
  - They record the winner at each level
- A *loser tree* records the loser at each level.

Binary Trees

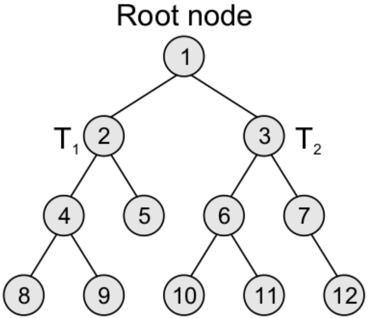
Definition

A binary tree is a data structure which is defined as a collection of elements called nodes:

- The topmost element is called the *root node*
- Each node has *0, 1, or at the most 2* children

Each node contains:

- a data element,
- a "left" pointer which points to the *left child*,
- a "right" pointer which points to the *right child*.



- The root element is pointed by a "root" pointer.
- If `root = NULL`, then the tree is empty.

Parent If N is any node in T that has *left successor* S_1 and *right successor* S_2, then N is called the *parent* of S_1 and S_2. Every node other than the root node has a parent

Sibling All nodes that are at the same level and share the same parent are called *siblings*

Level number Every node in the binary tree is assigned a *level number*. The root node is defined to be at level 0. The left and right child of the root node have a level number 1. Every node is at one level higher than its parents.

Binary Tree Terminology

Degree The number of children that a node has. The degree of a leaf node is zero.

In-degree The number of edges arriving at that node. The root has zero in-degree.

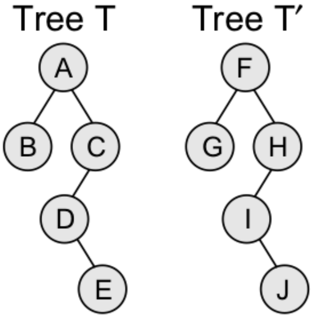
Out-degree The number of edges leaving that node.

Depth The length of the path from the root to the node N. The depth of the root node is zero.

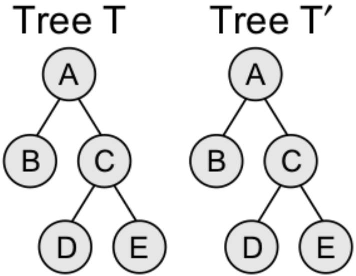
Height of a tree: The total number of nodes on the path from the *root* node to the *deepest node* in the tree.

- A tree with only a root node has a height of 1.
- A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. This is because every level will have at least one node and can have at most 2 nodes.
- The height of a binary tree with n nodes is at least $\log_2(n+1)$ and at most n.

Similar binary trees: Two binary trees T and T' are said to be similar if both trees have the *same structure*.



Copies of binary trees: Two binary trees T and T' are said to be copies if they have *similar structure and same content* at the corresponding nodes.

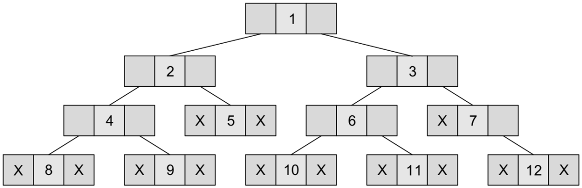


Linked Representation of Binary Trees

- In computer's memory, a binary tree can be maintained either using a linked representation or using sequential representation.
- In *linked representation* of binary tree, every node will have three parts:
  - the data element,
  - a pointer to the left node and
  - a pointer to the right node.

Binary tree node

```
struct node
{
    struct node* left;
    int data;
    struct node* right;
}
```

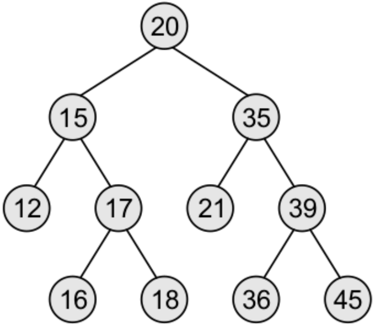


Sequential Representation of Binary Trees

- Sequential representation of trees is done using a single or one dimensional array.
- It is very inefficient as it requires a lot of memory space.
- A sequential binary tree follows the rules given below:
  - One dimensional array called `TREE` is used.
  - The root of the tree will be stored in the first location. That is, `TREE[1]` will store the data of the root element.
  - The children of a node K will be stored in location $(2 * K)$ and $(2 * K + 1)$.
  - The maximum size of the array `TREE` is given as $(2^h - 1)$, where h is the height of the tree.
  - An empty tree or sub-tree is specified using `NULL`. If `TREE[1] = NULL`, then the tree is empty.

Sequential Representation of Binary Trees

A binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.



1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

Definition

Traversing a binary tree is the process of *visiting each node* in the tree *exactly once* in a *systematic* way.

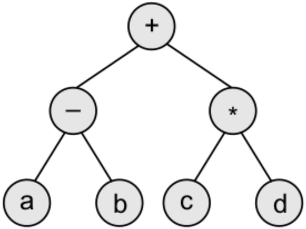
There are three different algorithms for tree traversals, which differ in the order in which the nodes are visited:

- Pre-order algorithm
- In-order algorithm
- Post-order algorithm

Algorithm for pre-order traversal

```
Step 1: Repeat Steps 2 to 4
        while TREE != NULL
Step 2:    Write TREE -> DATA
Step 3:    PREORDER(TREE -> LEFT)
Step 4:    PREORDER(TREE -> RIGHT)
        [END OF LOOP]
5: END
```

- Pre-order traversal is also called as *depth-first traversal.*
- Pre-order traversal algorithms are used to extract a *prefix notation* from an expression tree. For example:
  `+ − a b * c d`



JÖNKÖPING UNIVERSITY

Algorithm for in-order traversal

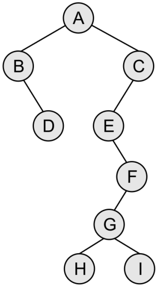
```
Step 1: Repeat Steps 2 to 4
        while TREE != NULL
Step 2:    INORDER(TREE -> LEFT)
Step 3:    Write TREE -> DATA
Step 4:    INORDER(TREE -> RIGHT)
        [END OF LOOP]
5: END
```

In-order traversal is also called as *symmetric traversal*.

In-order traversal algorithm is usually used to display the elements of a binary search tree

An example of in-order traversal:

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



Post-order Traversal

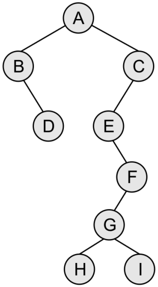
Algorithm for post-order traversal

```
Step 1: Repeat Steps 2 to 4
        while TREE != NULL
Step 2:    POSTORDER(TREE -> LEFT)
Step 3:    POSTORDER(TREE -> RIGHT)
Step 4:    Write TREE -> DATA
        [END OF LOOP]
5: END
```

Post-order traversals are used to extract postfix notation from an expression tree.

An example of post-order traversal:

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



Binary Search Trees

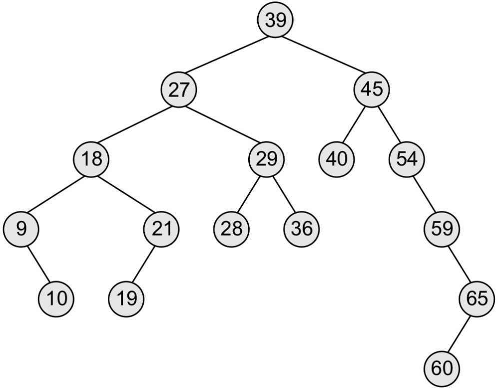
Definition

A binary search tree (BST), also known as an *ordered binary tree*, is a binary tree in which the nodes are arranged in order:

- All nodes in the *left* sub-tree have a value *less than* that of the *root* node.
- All nodes in the *right* sub-tree have a value *equal to or greater than* the *root* node.
- The same rule is applicable to every sub-tree in the tree.

Binary Search Trees

- BSTs are efficient in *searching* elements.
- BSTs are widely used in *dictionary problems*
    - where the code always inserts and searches the elements that are indexed by some key value.
- A binary search tree may or may not contain *duplicate values*, depending on its implementation



Operations on Binary Search Trees

1. Searching for a node in a binary search tree
2. Inserting a new node in a binary search tree
3. Deleting a node from a binary search tree
4. Determining the height of a binary search tree
5. Determining the number of nodes
6. Finding the mirror image of a binary search tree
7. Finding the smallest node of a binary search tree

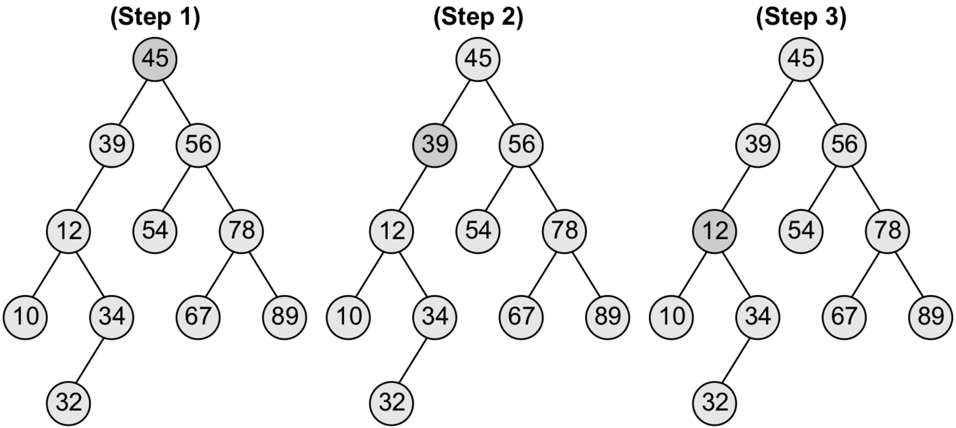
Searching for a Node in a BST

Algorithm to Search a Value in a BST

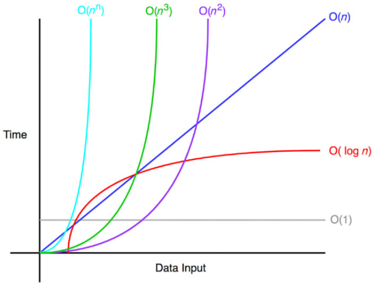
```
searchElement (TREE, VAL)
Step 1: IF TREE->DATA = VAL OR TREE = NULL, then
              Return TREE
        ELSE
            IF VAL < TREE->DATA
                Return searchElement(TREE->LEFT, VAL)
            ELSE
                Return searchElement(TREE->RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: End
```

- Since the nodes are ordered in BSTs, we eliminate half of the sub-tree from the search process as at every step.
- BSTs also speed up the insertion and deletion operations

Searching for a Node with value 12 in a BST



- The average running time of a search operation in a BST is $O(\log_2 n)$
- The worst case time to search for an element in a BST is $O(n)$
  - Occurs when the tree is a linear chain of nodes
- In a sorted array, searching can be done in $O(\log_2 n)$ time, but insertions and deletions are quite expensive
- In a linked list, inserting and deleting elements is easier, but searching for an element is done in $O(n)$ time.



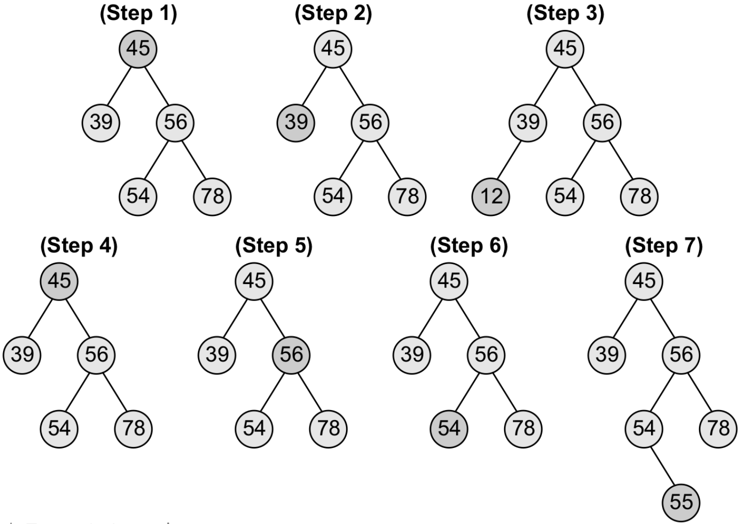
Inserting in a New Node in Binary Search Trees

Algorithm to insert a node in a BST

```
Insert (TREE, VAL)
Step 1: IF TREE = NULL, then
            Allocate memory for TREE
            SET TREE->DATA = VAL
            SET TREE->LEFT = TREE ->RIGHT = NULL
        ELSE
            IF VAL < TREE->DATA
                SET TREE->LEFT = Insert(TREE->LEFT, VAL)
            ELSE
                SET TREE->RIGHT =Insert(TREE->RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: End
```

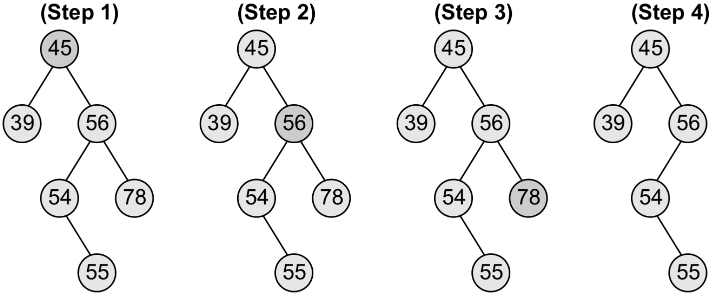
- The new node is added by following the rules of the binary search trees.
- The insertion requires time proportional to the height of the tree in the worst case.
- It takes $O(log_2 n)$ time to execute in the average case and $O(n)$ time in the worst case.

Inserting Node with Values 12 and 55 in a BST



- Care should be taken that the properties of the BSTs do not get violated and nodes are not lost in the process.
- The deletion of a node involves any of the three cases.

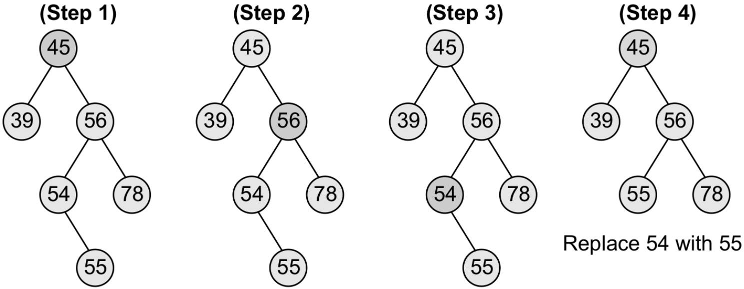
Case 1: Deleting a node (78) that has no children



Delete node 78

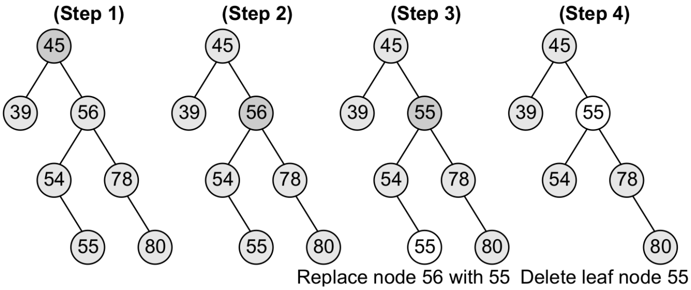
Case 2: Deleting a node (54) with one child (either left or right).

- The node's child is set to be the child of the node's parent.
- In other words, replace the node with its child.



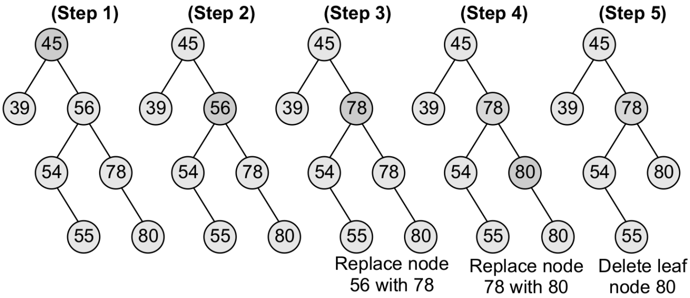
Case 3: Deleting a node (56) with two children.

- Replace the node's value with its *in-order predecessor* (largest value in the left sub-tree) or *in-order successor* (smallest value in the right sub-tree).
- Node 56 is replaced by its in-order predecessor.



Case 3: Deleting a node with two children.

- Replace the node's value with its *in-order predecessor* (largest value in the left sub-tree) or *in-order successor* (smallest value in the right sub-tree).
- Node 56 is replaced by its in-order successor.



Algorithm to delete a values from a BST

```
Delete (TREE, VAL)
Step 1: IF TREE = NULL, then
            Write "VAL not found in the tree"
        ELSE IF VAL < TREE->DATA
            SET TREE->LEFT = Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE->DATA
            SET TREE->RIGHT = Delete(TREE->RIGHT, VAL)
        ELSE IF TREE->LEFT AND TREE->RIGHT
            SET TEMP = findLargestNode(TREE->LEFT)
            SET TREE->DATA = TEMP->DATA
            SET TREE->LEFT = Delete(TREE->LEFT, TEMP->DATA)
        ELSE
            SET TEMP = TREE
            IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
                SET TREE = NULL
            ELSE IF TREE->LEFT != NULL
                SET TREE = TREE->LEFT
            ELSE
                SET TREE = TREE->RIGHT
        [END OF IF]
             FREE TEMP
    [END OF IF]
Step 2: End
```

- Deletion requires time proportional to the height of the tree in the worst case, $O(n)$.
- It takes $O(log_2 n)$ time to execute in the average case

JÖNKÖPING UNIVERSITY

Algorithm to determine the height of a BST

```
Height (TREE)
Step 1: IF TREE = NULL, then
            Return 0
        ELSE
            SET LeftHeight = Height(TREE->LEFT)
            SET RightHeight = Height(TREE->RIGHT)
        IF LeftHeight > RightHeight
            Return LeftHeight + 1
        ELSE
            Return RightHeight + 1
        [END OF IF]
    [END OF IF]
Step 2: End
```

- In order to determine the height of a BST, we will calculate the height of the left and right sub-trees.
- Whichever height is greater, 1 is added to it.

Determining the Number of Nodes in a BST

Algorithm to determine the number of nodes in a BST

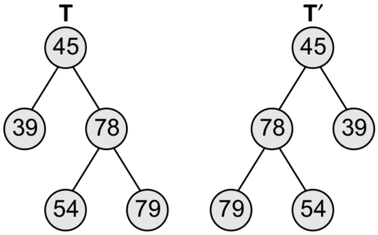
```
totalNodes (TREE)
Step 1: IF TREE = NULL, then
            Return 0
        ELSE
            Return totalNodes(TREE->LEFT)  +
                   totalNodes(TREE->RIGHT) + 1
        [END OF IF]
Step 2: End
```

- To calculate the total number of nodes in a BST, count the number of nodes in the left sub-tree and the right sub-tree ans add 1.

Algorithm to obtain the mirror image of a BST

```
MirrorImage (TREE)
Step 1: IF TREE != NULL , then
            MirrorImage(TREE->LEFT)
            MirrorImage(TREE->RIGHT)
            SET TEMP = TREE->LEFT
            SET TREE->LEFT = TREE->RIGHT
            SET TREE->RIGHT = TEMP
        [END OF IF]
Step 2: End
```

Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree.



Algorithm to find the smallest node in a bst

```
findSmallestElement (TREE)
Step 1: IF TREE = NULL OR TREE->LEFT = NULL, then
            Return TREE
        ELSE
            Return findSmallestElement(TREE->LEFT)
        [END OF IF]
Step 2: End
```

- The basic property of a BST states that the smaller value will occur in the left sub-tree.
- If the left sub-tree is NULL, then the value of root node will be smallest as compared with nodes in the right sub-tree.

AVL Trees

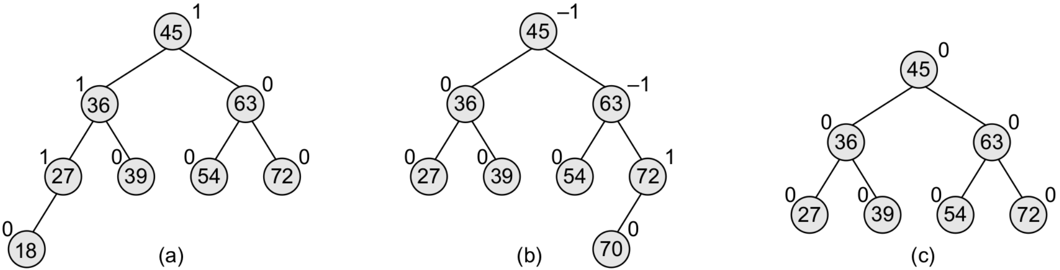
Definition

- AVL tree is a *self-balancing* binary search tree in which the heights of the two sub-trees of a node may differ by at most one.
- In the structure, AVL tree stores an additional variable called the BalanceFactor.

- AVL tree is also known as a *height-balanced tree*.
- The key advantage of using an AVL tree is that it takes $O(\log_2 n)$ time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to $O(\log_2 n)$).

Balance Factor of AVL Trees

- The balance factor of a node
  Balance factor = Height (left sub-tree) – Height (right sub-tree)
- A BST in which every node has a balance factor of -1, 0 or 1 is said to be *height balanced*. A node with any other balance factor is considered to be unbalanced and requires rebalancing.
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called *Left-heavy tree*.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called *Right-heavy tree*.



(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Searching for a Node in an AVL Tree

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes $O(\log_2 n)$ time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

Inserting a Node in an AVL Tree

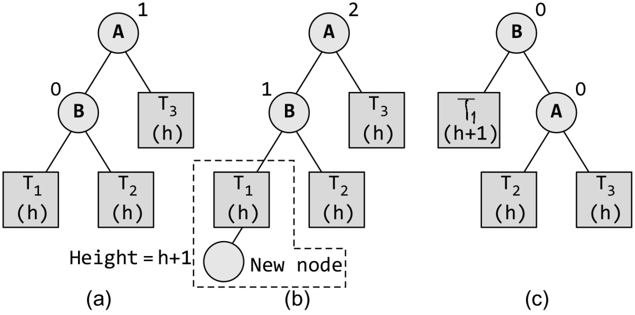
- Since an AVL tree is also a variant of BST, insertion is also done in the same way.
- The new node is always inserted as the leaf node.
- But the step of insertion is usually followed by an additional step of rotation.
- Rotation is done to restore the balance of the tree.
- If after insertion of the new node, the balance factor of every node is still -1, 0 or 1, then rotations are not needed.

Inserting a Node in an AVL Tree

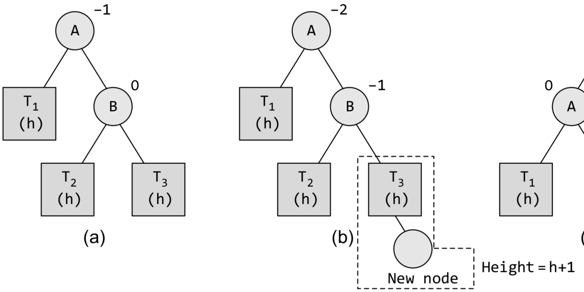
- The new node is inserted as the leaf node, so it will always have balance factor equal to zero.
- The nodes whose balance factors will change are those which lie on the *path* between the *root* of the tree and the *newly inserted* node.
- The possible changes which may take place in any node on the path are as follows:
  - Initially the node was either left or right heavy and after insertion has become balanced.
  - Initially the node was balanced and after insertion has become either left or right heavy.
  - Initially the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree thereby creating an unbalanced sub-tree. Such a node is said to be a *critical node*.

- To perform rotation, we need to find the critical node.
- Critical node is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.
- The second task is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations and their application depends on the position of the inserted node with reference to the critical node.

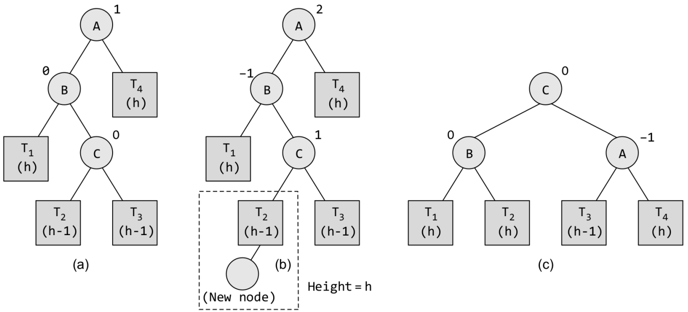
- **LL rotation:** the new node is inserted in the *left sub-tree* of the *left sub-tree* of the critical node
- Node B becomes the root, with T_1 and A as its left and right child. T_2 and T_3 become the left and right sub-trees of A .



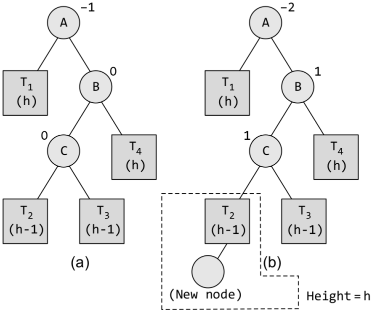
- **RR rotation:** the new node is inserted in the *right sub-tree* of the *right sub-tree* of the critical node
- Node B becomes the root, with A and T_3 as its left and right child. T_1 and T_2 become the left and right sub-trees of A.



- **LR rotation:** the new node is inserted in the *right sub-tree* of the *left sub-tree* of the critical node
- Node C becomes the root, with B and A as its left and right children. Node B has T_1 and T_2 as its left and right sub-trees and T_3 and T_4 become the left and right sub-trees of node A .



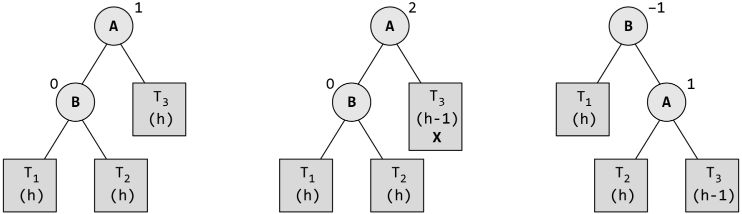
- **RL rotation:** the new node is inserted in the *left sub-tree* of the *right sub-tree* of the critical node
- Node C becomes the root, with A and B as its left and right children. Node A has T_1 and T_2 as its left and right sub-trees and T_3 and T_4 become the left and right sub-trees of node B .



Deleting a Node from an AVL Tree

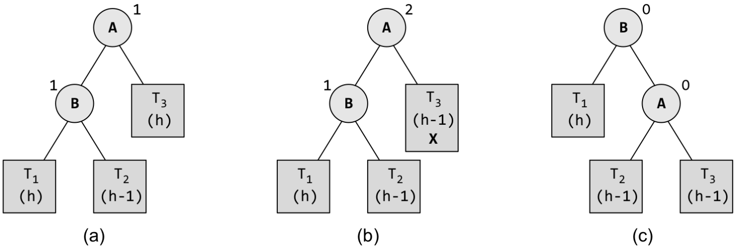
- Deletion of a node in an AVL tree is similar to that of binary search trees.
- Deletion may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations.
- There are two classes of rotation that can be performed on an AVL tree after deleting a given node: R rotation and L rotation.
  - If the node to be deleted is present in the left sub-tree of the critical node, then L rotation is applied else
  - if node is in the right sub-tree, R rotation is performed.
- Further there are three categories of L and R rotations.
  - The variations of L rotation are: L-1, L0 and L1 rotation.
  - Correspondingly for R rotation, there are R0, R-1 and R1 rotations.

- Let B be the root of the left or right sub-tree of A (critical node).
- **R0 rotation** is applied if *the balance factor of B is 0*.
- Node B becomes the root, with T_1 and A as its left and right child. T_2 and T_3 become the left and right sub-trees of A .

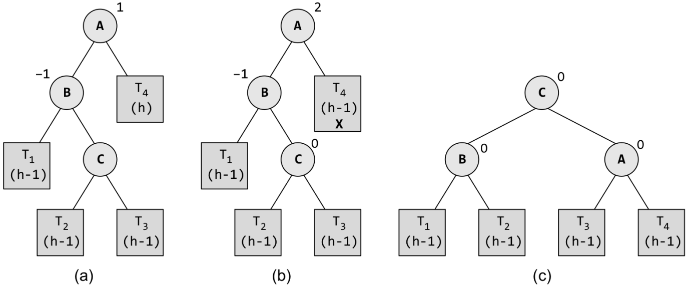


(a) (b) (c)

- Let B be the root of the left or right sub-tree of the critical node.
- **R1 rotation** is applied if *the balance factor of B is 1*.
- Node B becomes the root, with T_1 and A as its left and right child. T_2 and T_3 become the left and right sub-trees of A .



- Let B be the root of the left or right sub-tree of the critical node.
- **R-1 rotation** is applied if *the balance factor of B is -1.*
- Node C becomes the root, with T_1 and A as its left and right child. T_2 and T_3 become the left and right sub-trees of A .



QUESTIONS ?