DATA STRUCTURE AND ALGORITHM

LECTURER: ER. SAROJ GHIMIRE
QUALIFICATION : MSC.CSIT, COMPUTER ENGINEERING

Lincoln University College

Overview

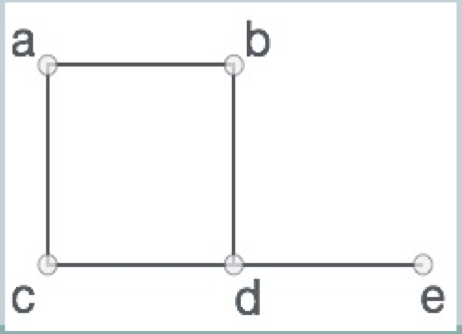
- Concept and representation of graphs, graphs traversal, minimum spanning tree: kruskal algorithm
- Shortest path algorithm: dijksrtra algorithm
- Definition of tree, tree height, level and depth; basic operation in binary tree
- Tree traversals, binary search tree, AVL tree, application of tree

Objectives

- To learn what a graph is and how it is used.
- To implement the graph abstract data type using multiple internal representations.
- To see how graphs can be used to solve a wide variety of problems
- Define trees as data structures
- Define the terms associated with trees
- Discuss tree traversal algorithms Discuss

Concept and representation of graphs

- Concept of graphs
  - A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
  - Graph is a non linear data structure
  - A graph is a data structure that consists of the following two components:
    - A finite set of vertices also called as nodes.
    - A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost
    - In this graph, V = {a, b, c, d, e} E = {ab, ac, bd, cd, de}



Concept and representation of graphs(contd)

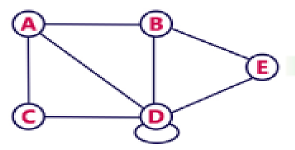
- Representation of graphs
  - In graph theory, a graph representation is a technique to store graph into the memory of computer.
  - To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.
  - There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

Concept and representation of graphs(contd)

1. Adjacency Matrix

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a n X n matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

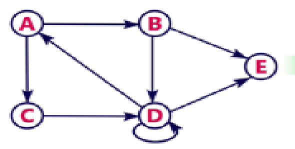
Undirected graph representation



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph represenation

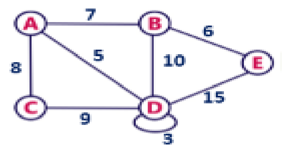
See the directed graph representation:



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

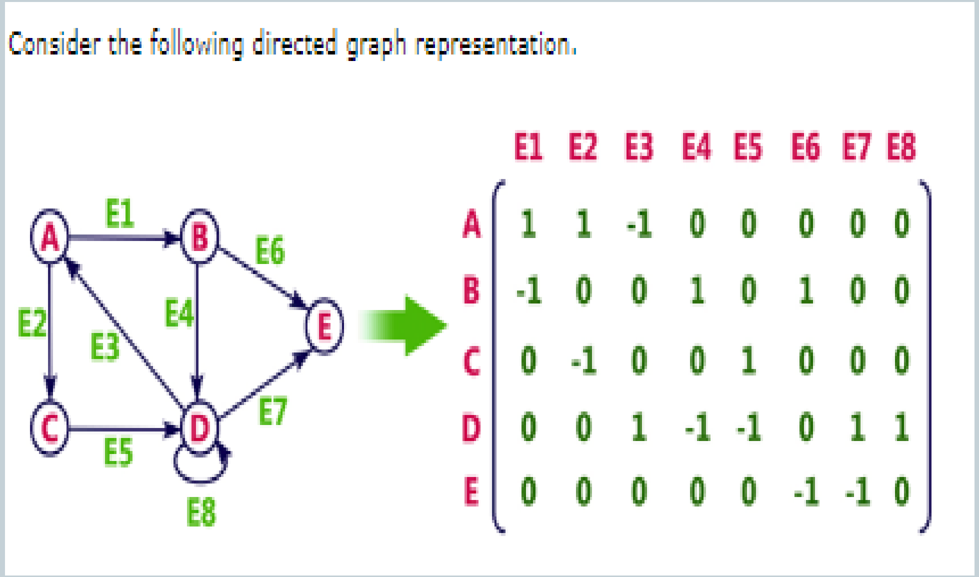
Undirected weighted graph represenation



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 7 & 8 & 5 & 0 \\ B & 7 & 0 & 0 & 10 & 6 \\ C & 8 & 0 & 0 & 9 & 0 \\ D & 5 & 10 & 9 & 3 & 15 \\ E & 0 & 6 & 0 & 15 & 0 \end{array}$$

Concept and representation of graphs(contd)

2. Incidence matrix
   - In Incidence matrix representation, graph can be represented using a matrix of size:
   - Total number of vertices by total number of edges.
   - It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.
   - This matrix is filled with either 0 or 1 or -1. Where,
     - 0 is used to represent row edge which is not connected to column vertex.
     - 1 is used to represent row edge which is connected as outgoing edge to column vertex.
     - -1 is used to represent row edge which is connected as incoming edge to column vertex



Consider the following directed graph representation.

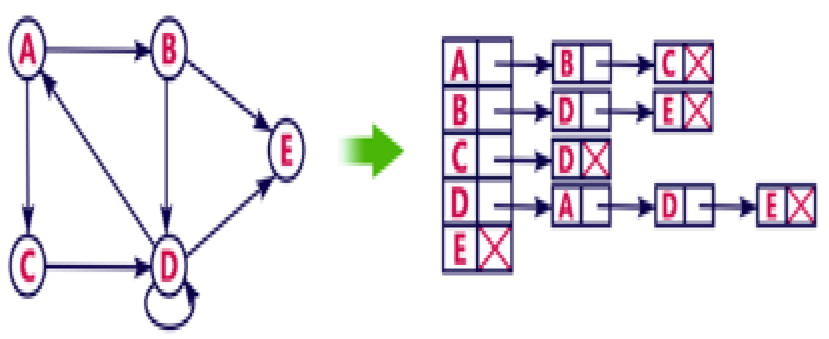
$$\begin{array}{c|cccccccc}
 & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\
\hline
A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0
\end{array}$$

Concept and representation of graphs(contd)

3. Adjacency List

- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbors of v.

Let's see the following directed graph representation implemented using linked list:



Graphs Traversal

- **Graph traversal** is a technique used to search for a vertex in a graph. It is also used to decide the order of vertices to be visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. This means that, with graph traversal, we can visit all the vertices of the graph without getting into a looping path. There are two graph traversal techniques:

1. **DFS (Depth First Search)**

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

1. Define a Stack of size total number of vertices in the graph.
2. Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
3. Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
4. Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
5. When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
6. Repeat steps 3, 4 and 5 until stack becomes Empty.
7. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

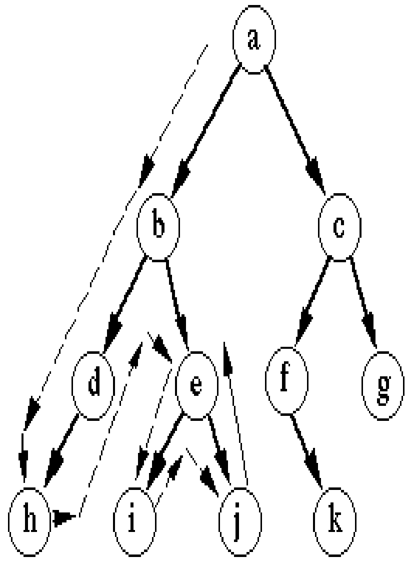
Graphs Traversal(contd)

2. **BFS (Breath First Search)**

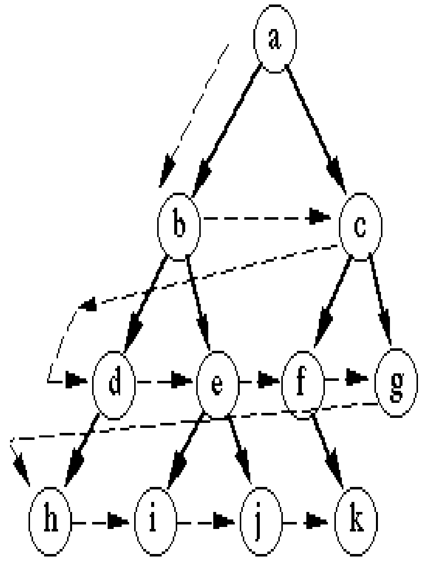
   BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal

   1. Define a Queue of size total number of vertices in the graph.
   2. Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
   3. Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
   4. When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
   5. Repeat steps 3 and 4 until queue becomes empty.
   6. When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

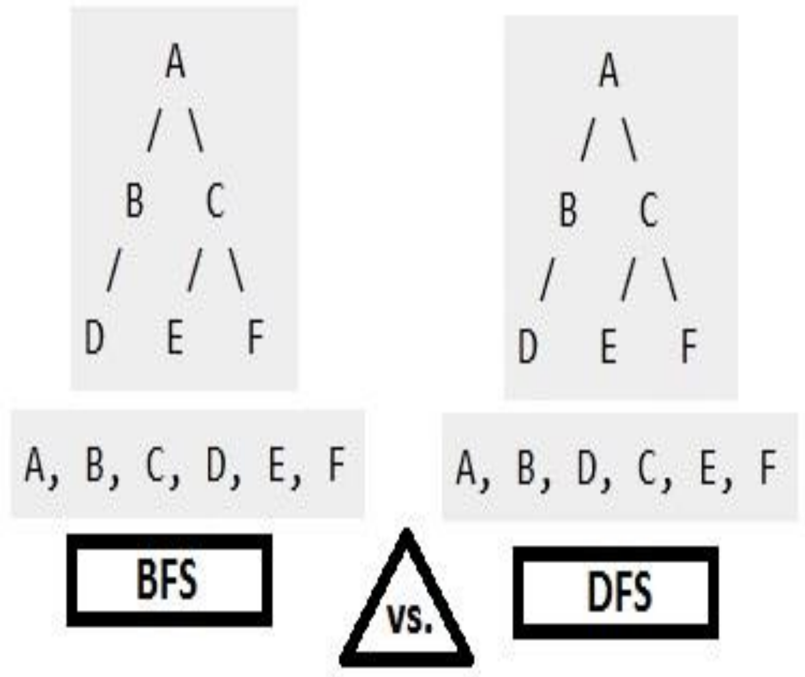
Example of **DFS VS BFS** on next page



Depth-first search



Breadth-first search



Minimum Spanning Tree: Kruskal Algorithm

- A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Application in the design of networks, Handwriting recognition, Image segmentation
  - **Kruskal's Algorithm**

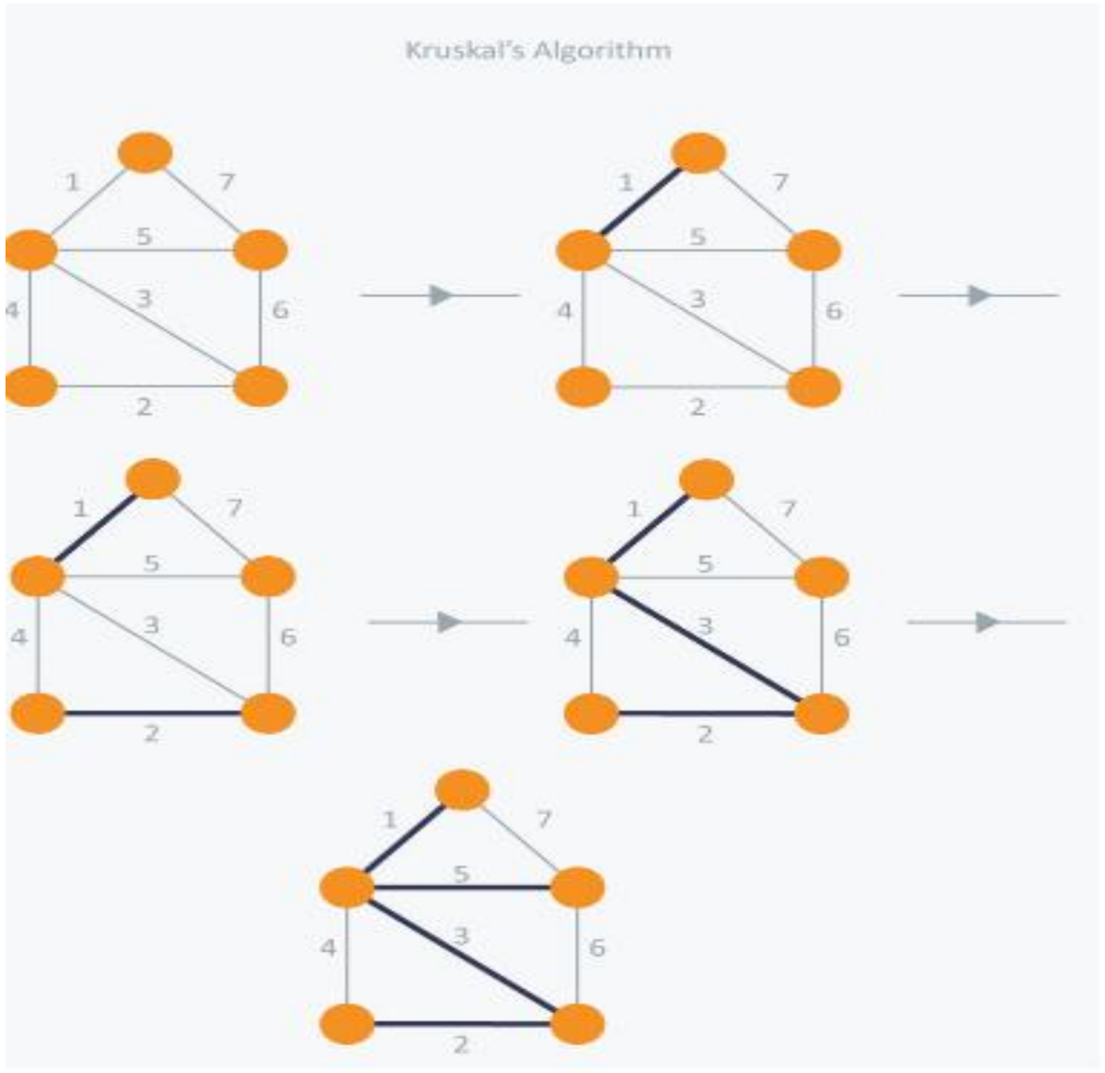
    Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

    Algorithm Steps:

    i. Sort the graph edges with respect to their weights.

    ii. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.

    iii. Only add edges which doesn't form a cycle , edges which connect only disconnected components.


Kruskal's Algorithm
1
7
5
3
4
6
2

Shortest Path Algorithm: Dijksrtra Algorithm

- The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

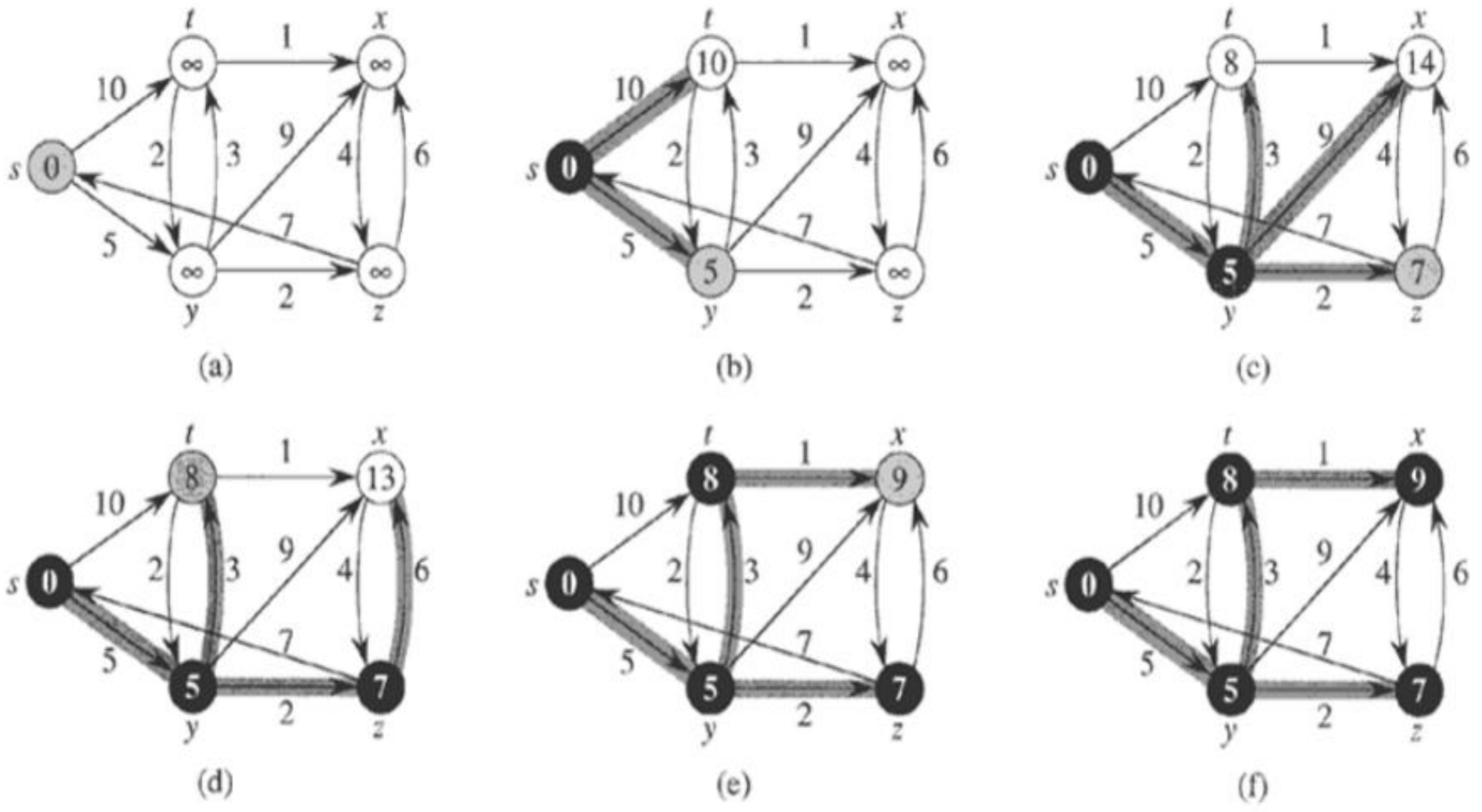
Dijksrtra Algorithm:

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). T

1. set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current
2. find minimum weight/distance from source to adjacent node that is unvisited repeat process until all node are visited .
3. Finally display the shortest path from source to destination.

The Dijkstra's Algorithm - An Example

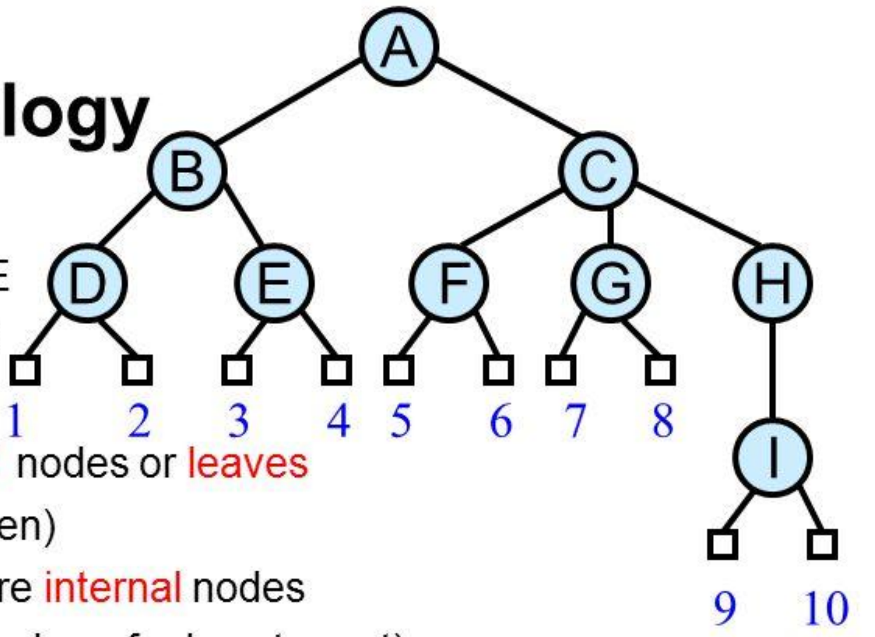


Definition Of Tree, Tree Height, Level And Depth

- A tree is a connected graph without any circuits **OR** If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.
- Terminology of tree
  - ✓ **Root:** In a tree data structure, the first node is called as **Root Node**
  - ✓ **Edge: In** a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.
  - ✓ **Parent:** In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**.
  - ✓ **Child:** In a tree data structure, the node which is descendant of any node is called as **CHILD Node**
  - ✓ **Siblings:** In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**
  - ✓ **Leaf:** In a tree data structure, the node which does not have a child is called as **LEAF Node.** In a tree, leaf node is also called as '**Terminal**' node.
  - ✓ **Internal Nodes :**In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.
  - ✓ **Degree:** In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree.**
  - ✓ **Level:** In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on..
  - ✓ **Height:** In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**.
  - ✓ **Depth:** In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**

Definition Of Tree, Tree Height, Level And Depth

Tree Terminology



- A is the root node
- B is the parent of D and E
- D and E are children of B
- (C ---- F) is an edge
- 1, 2,…,9, 10 are external nodes or leaves (i.e., nodes with no children)
- A, B, C, D, E, F, G, H, I are internal nodes
- depth (level) of E is 2 (number of edges to root)
- height of the tree is 4 (max number of edges in path from root)
- degree of node B is 2 (number of children)

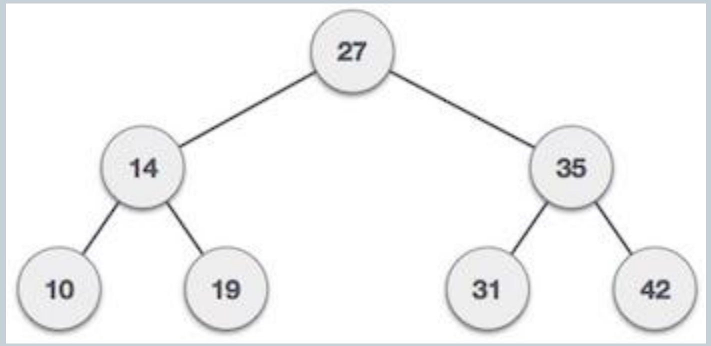
15/65

BASIC OPERATION IN BINARY TREE

- Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value

Tree node :

```
struct node {
        int data;
        struct node *leftChild;
        struct node *rightChild;
};
```



- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair (left and right)
- The basic operations that can be performed on a binary search tree data structure, are **Insert, Search Preorder Traversal, In order Traversal and Post order Traversal** –

Basic Operation In Binary Tree

- **Insert:** The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left sub tree and insert the data. Otherwise, search for the empty location in the right sub tree and insert the data

```
Algorithm
If root is NULL
  then create root node
return
If root exists then
  compare the data with node.data
  while until insertion position is located
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
  endwhile
  insert data
end If
```

- **Search:** Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left sub tree. Otherwise, search for the element in the right sub tree. Follow the same algorithm for each node.

```
Algorithm
If root.data is equal to search.data
  return root
else
  while data not found
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
    If data found
      return node
  endwhile
  return data not found
end if
```

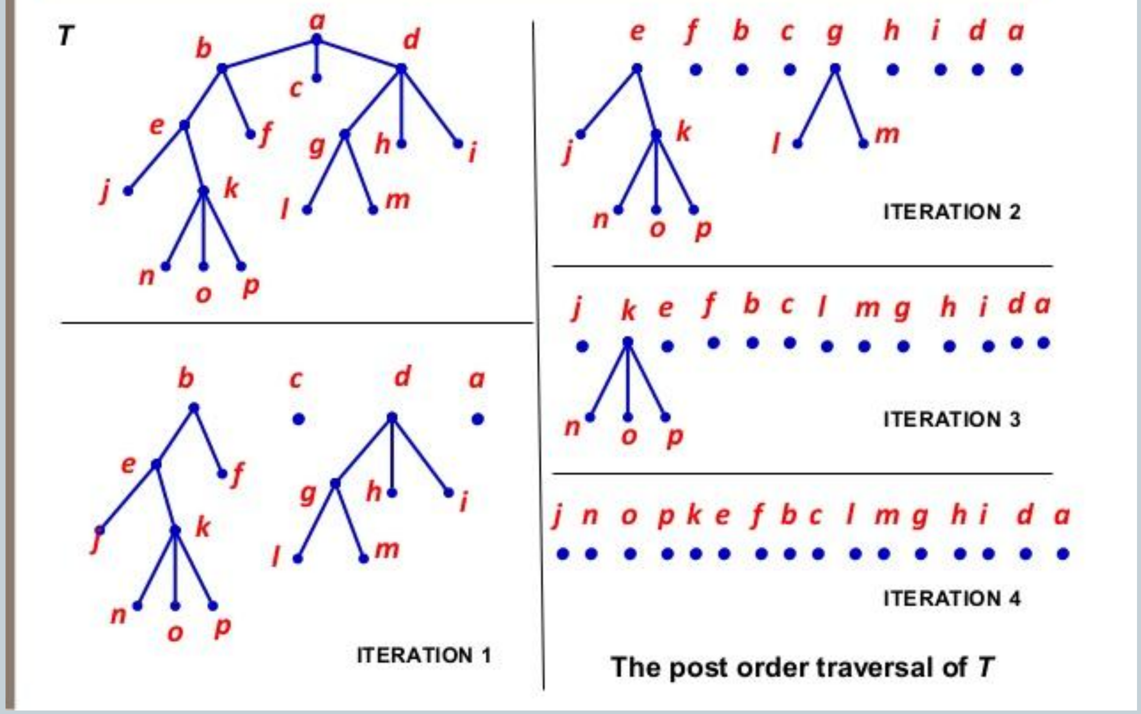
Tree Traversal

- A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

1. **In order Traversal: In the case of in order traversal, the root of each sub tree is visited after its left sub tree has been traversed but before the traversal of its right sub tree begins. The steps for traversing a binary tree in in order traversal are:**
   - **Visit the left sub tree, using in order.**
   - **Visit the root.**
   - **Visit the right sub tree, using in order**
2. **Preorder Traversal: In a preorder traversal, each root node is visited before its left and right sub trees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:**
   - **Visit the root.**
   - **Visit the left sub tree, using preorder.**
   - **Visit the right sub tree, using preorder**
3. **Post order Traversal: In a post order traversal, each root is visited after its left and right sub trees have been traversed. The steps for traversing a binary tree in post order traversal are:**
   - **Visit the left sub tree, using post order.**
   - **Visit the right sub tree, using post order**
   - **Visit the root**

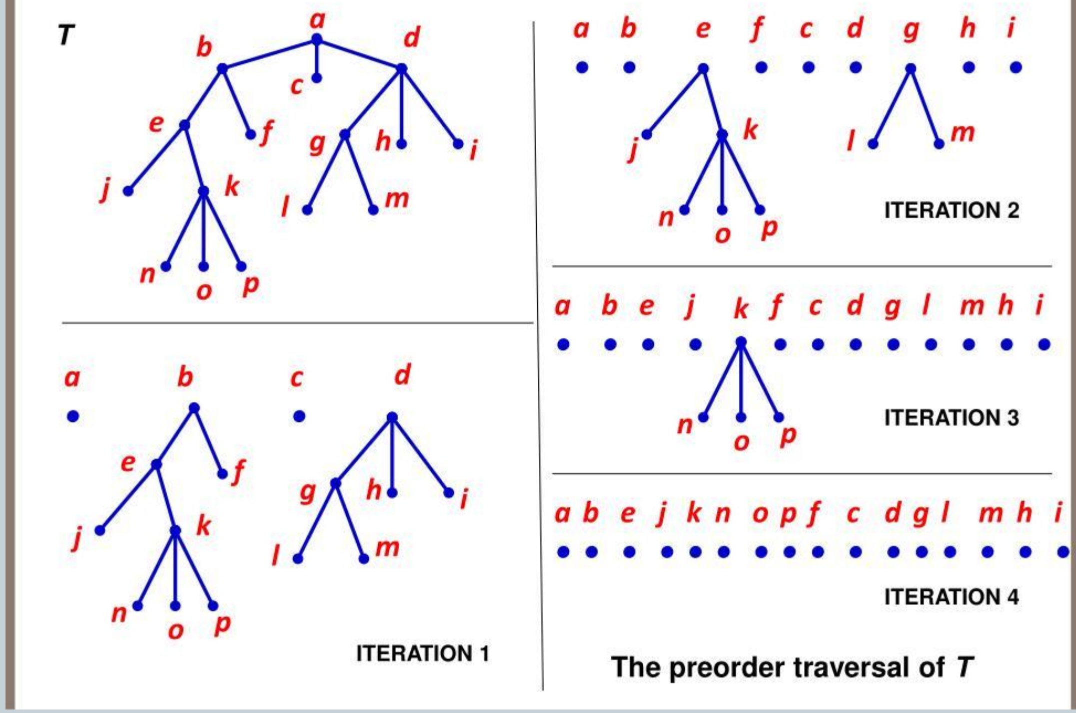
Tree Traversal

EXAMPLE: Postorder Traversal


T
a b c d e f g h i j k l m n o p
ITERATION 1
e f b c g h i d a
ITERATION 2
j k e f b c l m g h i d a
ITERATION 3
j n o p k e f b c l m g h i d a
ITERATION 4

The post order traversal of *T*

EXAMPLE: Preorder Traversal


T
a b c d e f g h i j k l m n o p
ITERATION 1
a b e f c d g h i
ITERATION 2
a b e j k f c d g l m h i
ITERATION 3
a b e j k n o p f c d g l m h i
ITERATION 4

The preorder traversal of *T*

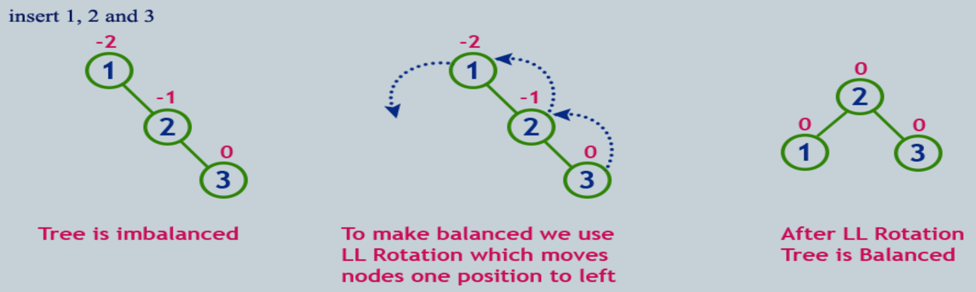
AVL Tree

- **AVL** tree is a height-balanced binary search tree. That means, an **AVL** tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right sub trees of every node in the tree is either -1, 0 or +1.
- In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains extra information known as balance factor.
- In **AVL** tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.
- Rotation operations are used to make the tree balanced

AVL Tree

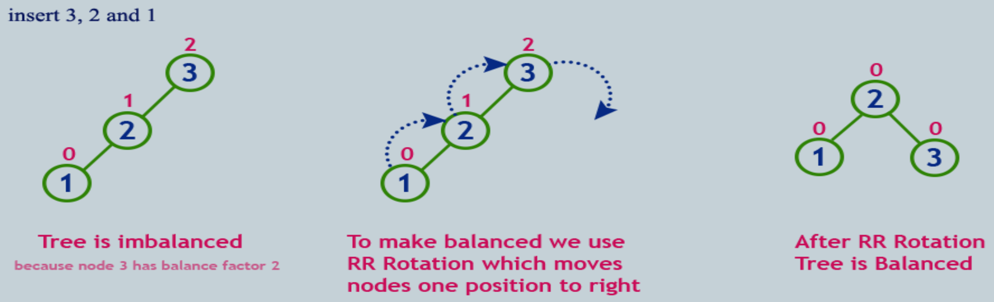
- **Single Left Rotation (LL Rotation)**

  In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree.


insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
-2
1
-1
2
0
3
To make balanced we use LL Rotation which moves nodes one position to left
0
2
0
1
0
3
After LL Rotation Tree is Balanced

- **Single Right Rotation (RR Rotation)**

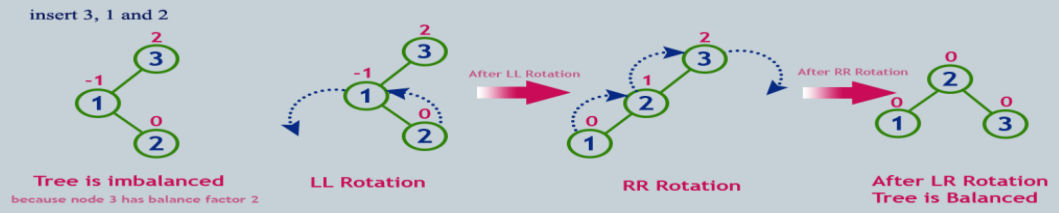
  In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree..


insert 3, 2 and 1
2
3
1
2
0
1
Tree is imbalanced
because node 3 has balance factor 2
2
3
1
2
0
1
To make balanced we use RR Rotation which moves nodes one position to right
0
2
0
1
0
3
After RR Rotation Tree is Balanced

AVL Tree

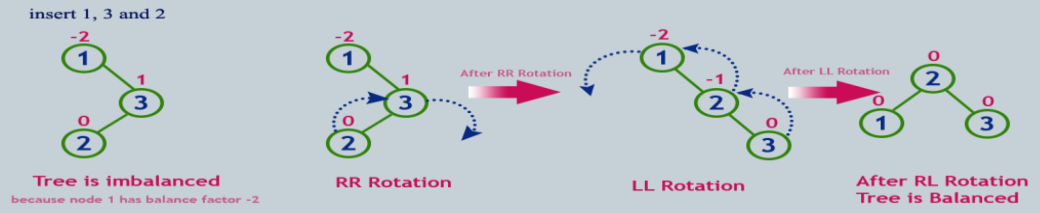
- **Left Right Rotation (LR Rotation)**

  The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree..



- **Right Left Rotation (RL Rotation)**

  The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree..

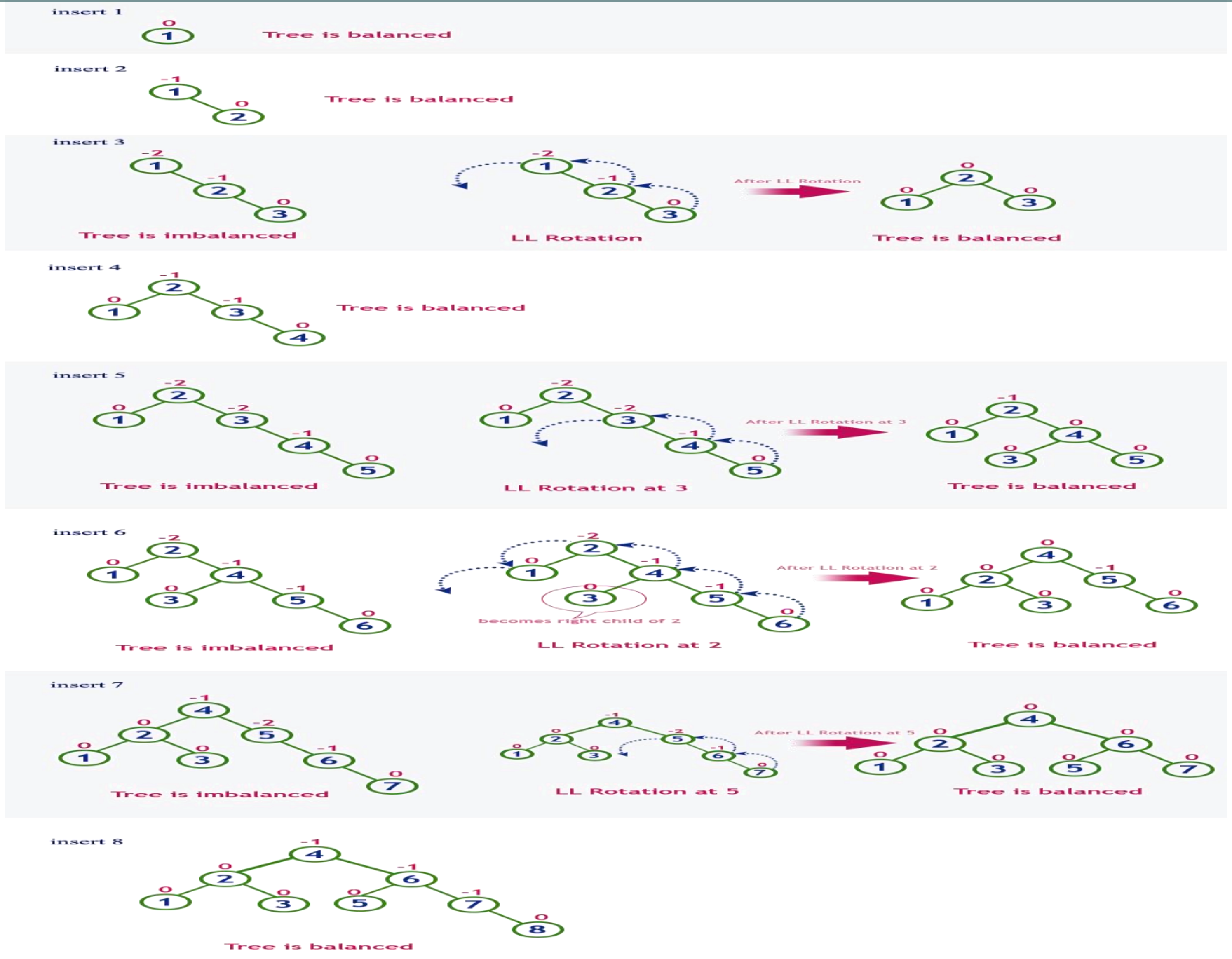


AVL Tree

Insertion Operation in AVL Tree

- ✓ In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...
  i. Insert the new element into the tree using Binary Search Tree insertion logic.
  ii. After insertion, check the Balance Factor of every node.
  iii. If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
  iv. If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8. On next page


insert 1
Tree is balanced
insert 2
Tree is balanced
insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced
insert 4
Tree is balanced
insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced
insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced
insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced
insert 8
Tree is balanced

Application Of Tree

- Represent organization
- Represent computer file systems
- Networks to find best path in the Internet
- Chemical formulas representation
- Manipulate hierarchical data.
- Make information easy to search (see tree traversal).
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms

Further Readings

- “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
- “Algorithms Unlocked” by Thomas H. Cormen