DATA STRUCTURES AND ITS APPLICATIONS

Balanced Trees

Sandesh B. J
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Balanced Trees

Sandesh B. J
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Balanced Trees

In this lecture you will be able to learn:

- **Why trees becomes unbalanced?**
- **Why we need to balance the tree?**
- **AVL Tree**
- **How do we balance the unbalanced trees using tree rotation techniques**
- **Different tree Rotation techniques**
  - ✓ **Left rotation, right rotation**
  - ✓ **Left-right rotation and right-left rotations**

Balanced Trees

Why Balanced Trees?

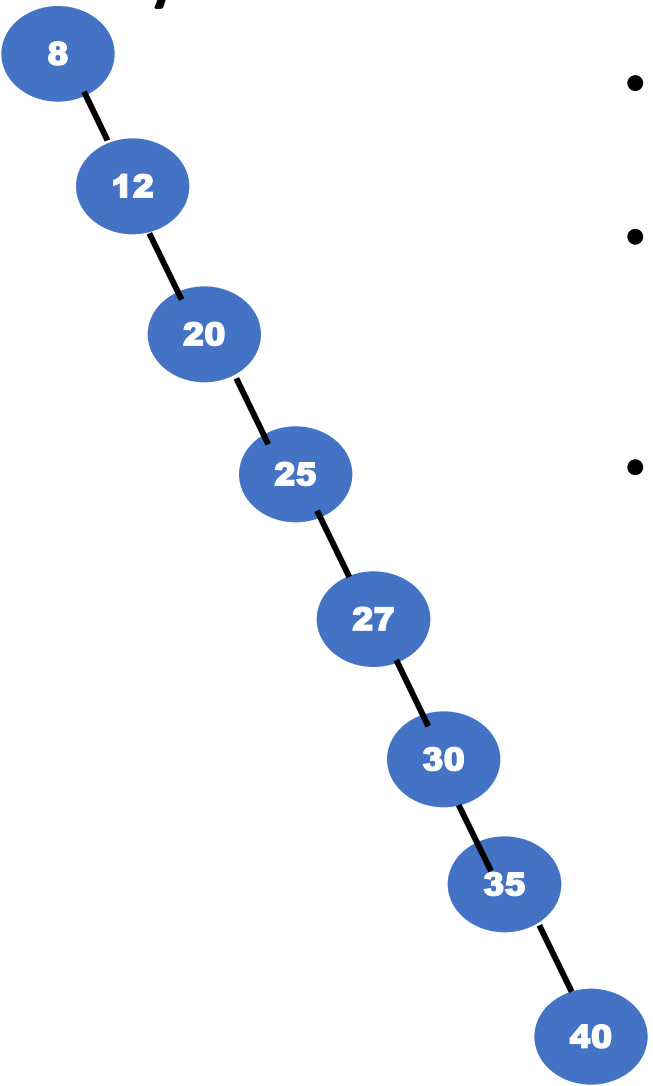
- Binary Search Tree(BST) – Data Structure used to implement the dictionary.

- What do we gain by implementing dictionary using BST instead of array ?

Complexity of Binary Search Tree

Operations	Average	Worst
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Search	O(log(n))	O(n)

Why Balanced trees?



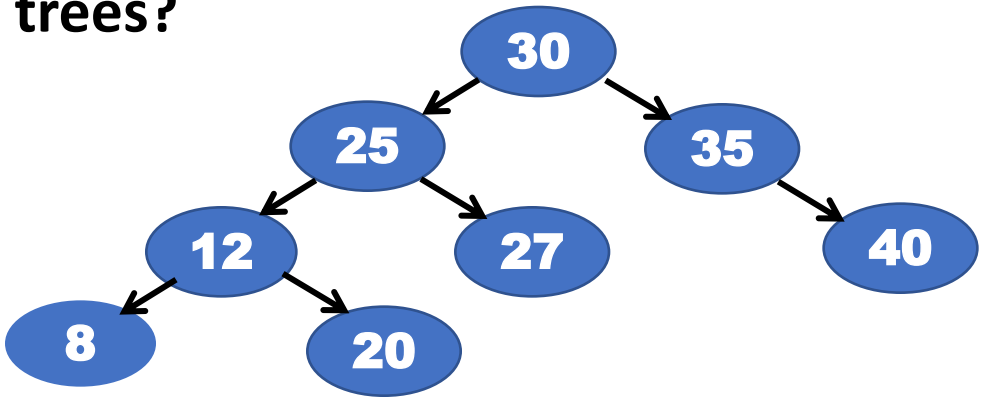
- Height of the BST depends on the order of insertion of elements into tree
- What happens if we insert elements given in increasing order?
  - Insert 8, 12, 20, 25, 27, 30, 35, 40
- Severely unbalanced

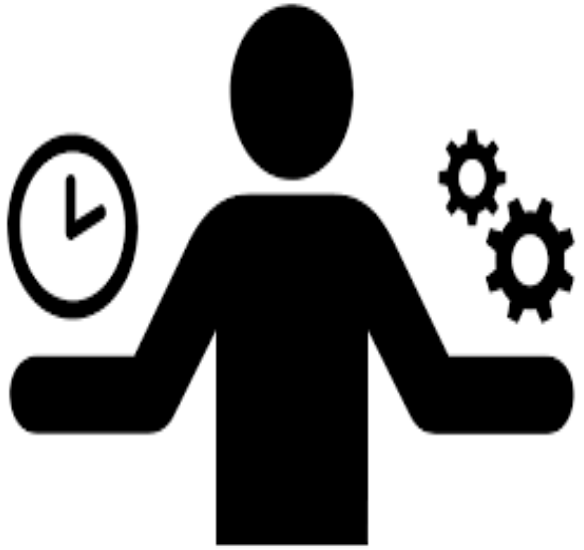
Disadvantages of Binary search trees

- The search and insertion algorithm does not ensure that the tree remains balanced
- The degree of balance dependent on the order in which the keys are inserted
- Tree can attain a height which can be as large as n-1
- Time taken for most of the operations worst case O(n)

Balanced Trees

Why Balanced trees?





- we can construct the binary search tree in which the height of the tree is always log(n)

Complexity of Balanced Binary Search Tree

Operations	Average	Worst
Insert	O(log(n))	O(log(n))
Delete	O(log(n))	O(log(n))
Search	O(log(n))	O(log()n)

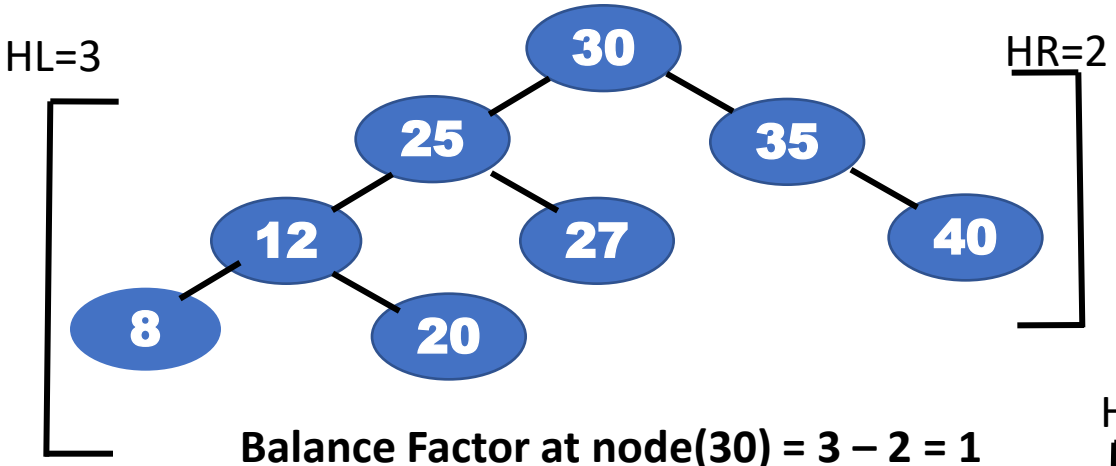
AVL Tree-Balanced Binary Search Trees

- AVL tree was invented in 1962 by two Russian mathematicians G.M. Adel'son-Vel'skii and E.M. Landis(AVL)

- An AVL tree is a binary search tree in which, for every node, the difference between the heights of the left and right subtrees, called the balance factor is either 0 or +1 or -1

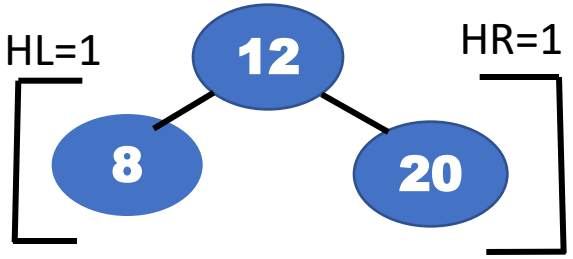
The Balance factor of any node:
Balance Factor = Height(left subtree) – Height(right subtree)

Balanced Binary Search Trees(AVL)

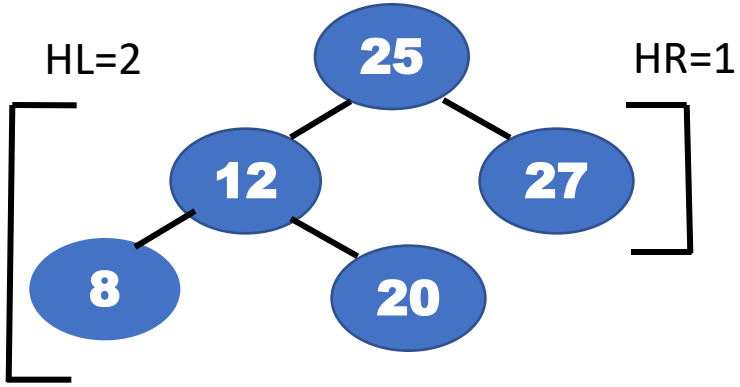


Balance Factor = HL - HR

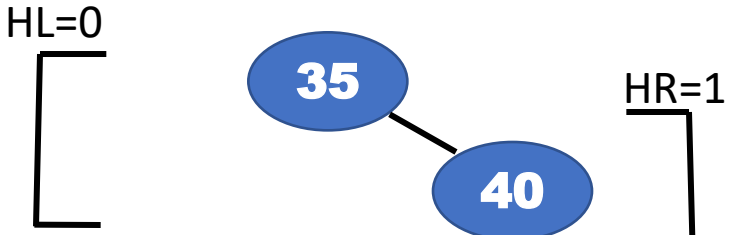
Balance Factor at node(30) = 3 – 2 = 1



Balance Factor at node(12) = 1– 1 = 0

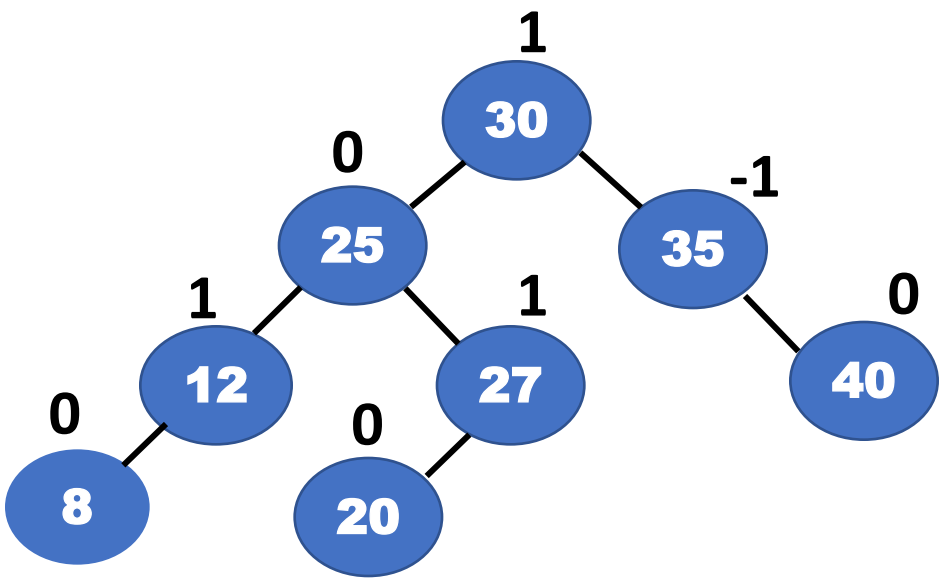


Balance Factor at node(25) = 2 – 1 = 1

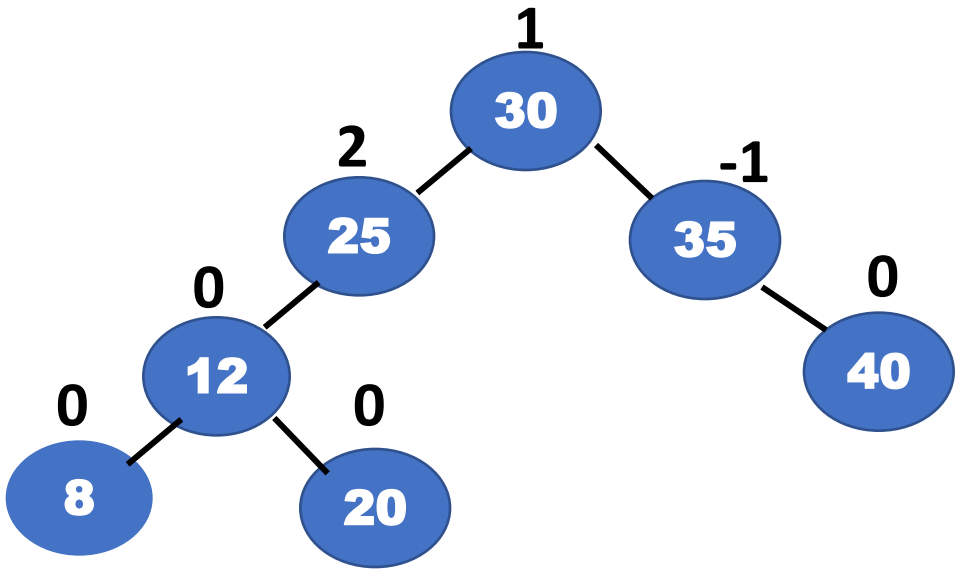


Balance Factor at node(35) = 0 – 1 = -1

Balanced Binary Search Trees(AVL)



AVL Tree



Not an AVL tree

- Node in balanced binary tree has balance of
  1 – Height(left subtree) > Height(right subtree)
  0 – Height(left subtree) = Height(right subtree)
  -1 – Height(left subtree) < Height(right subtree)

Rotations – AVL Tree

- The AVL tree may become unbalanced after insert and delete operations

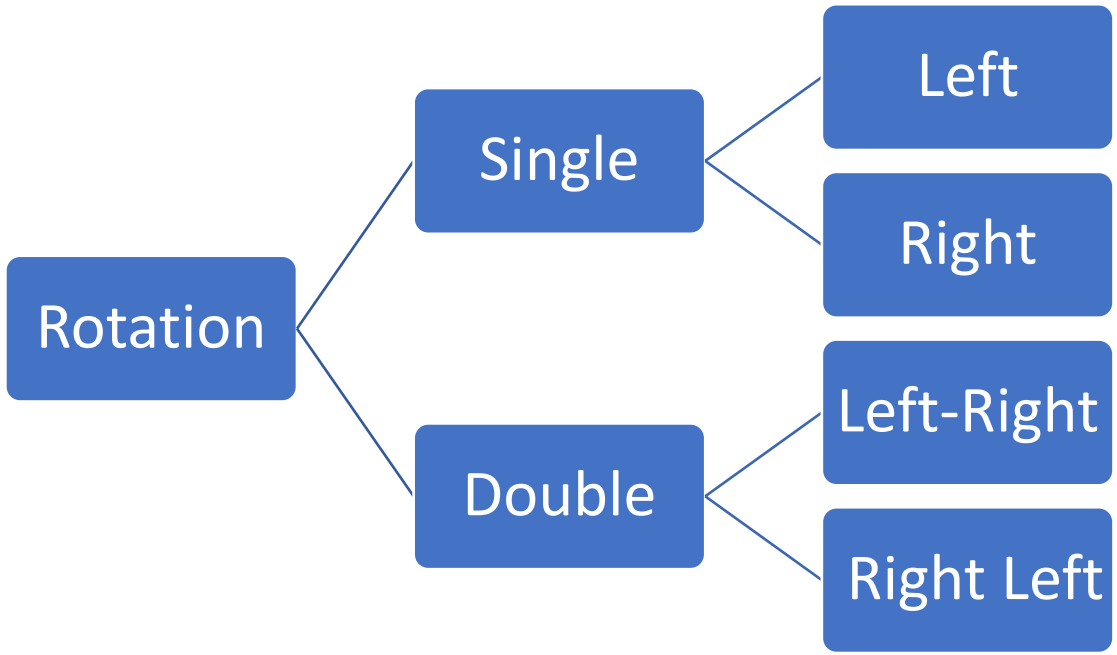
- If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four *rotations.*

- **Rotation** in a AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2

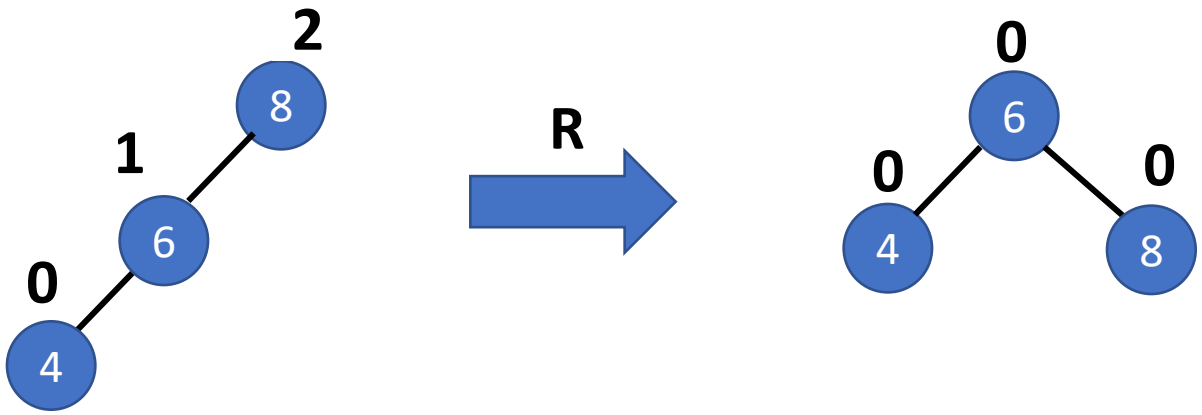
- In case there are several such nodes, The rotation is always performed for a subtree rooted at "unbalanced" node closest to the newly inserted leaf node.

Rotations – AVL Tree

- **Different types of Rotation:**



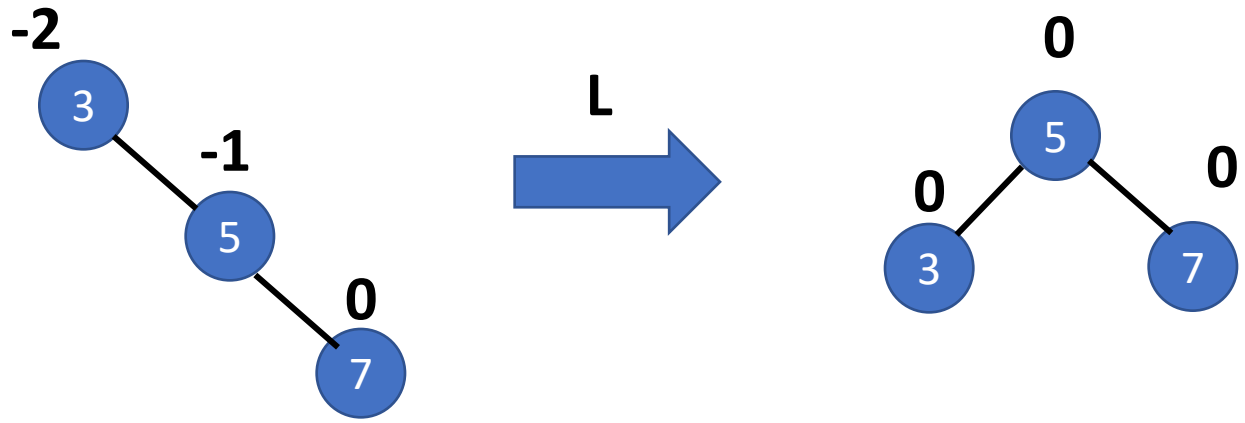
Single Right Rotation (R-Rotation)



- Root of the tree has balance of +1 before the insertion
- New key is inserted to the left of left child (Left-Left-case)
- To Balance the tree we need to perform the right rotation

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

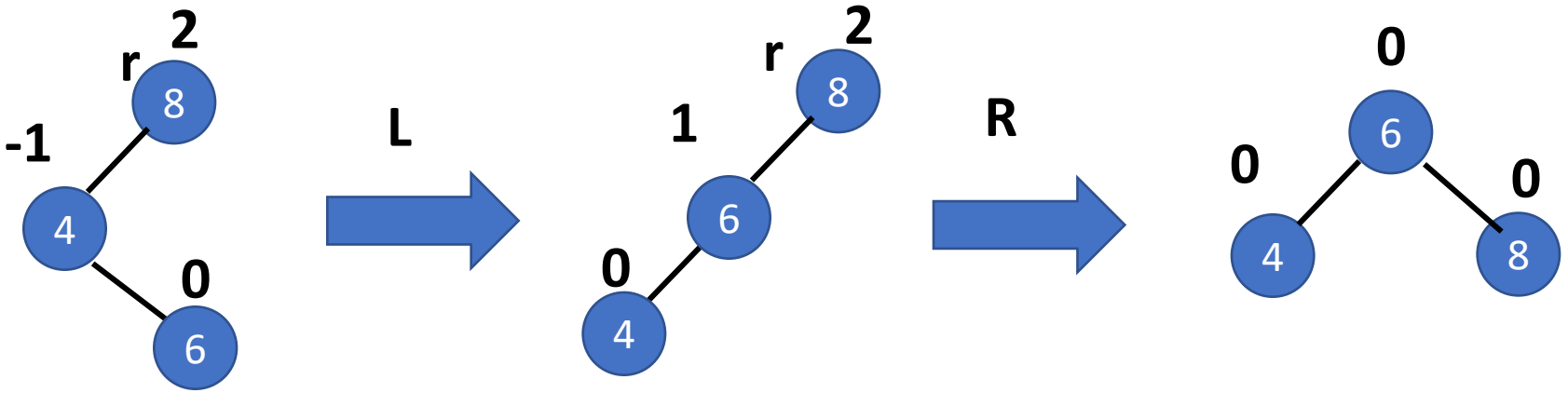
Single Left Rotation:



- Root of the tree had balance of -1 before the insertion
- New key is inserted to the right of right child (Right-Right-case)
- To Balance the tree we need to perform Left Rotation

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

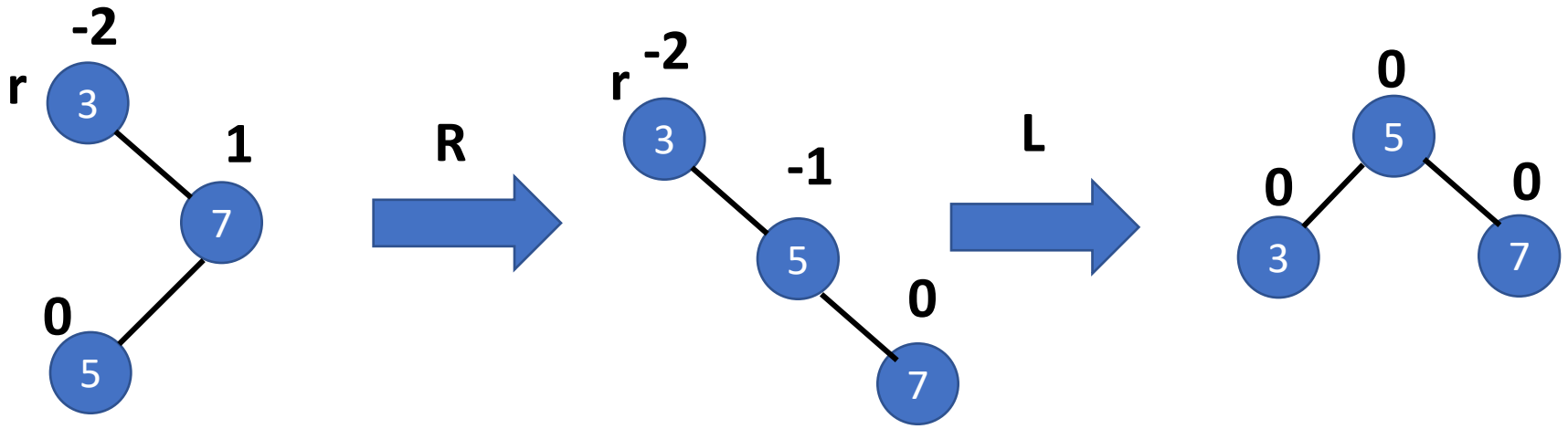
Double Left Right Rotation:



- Root r had the balance of +1 before the insertion
- New key is inserted to the right of left child (Left-Right-case)
- We perform the left rotation of left subtree of root r
- Right rotation of the new tree rooted at r

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

Double Right Left Rotation:

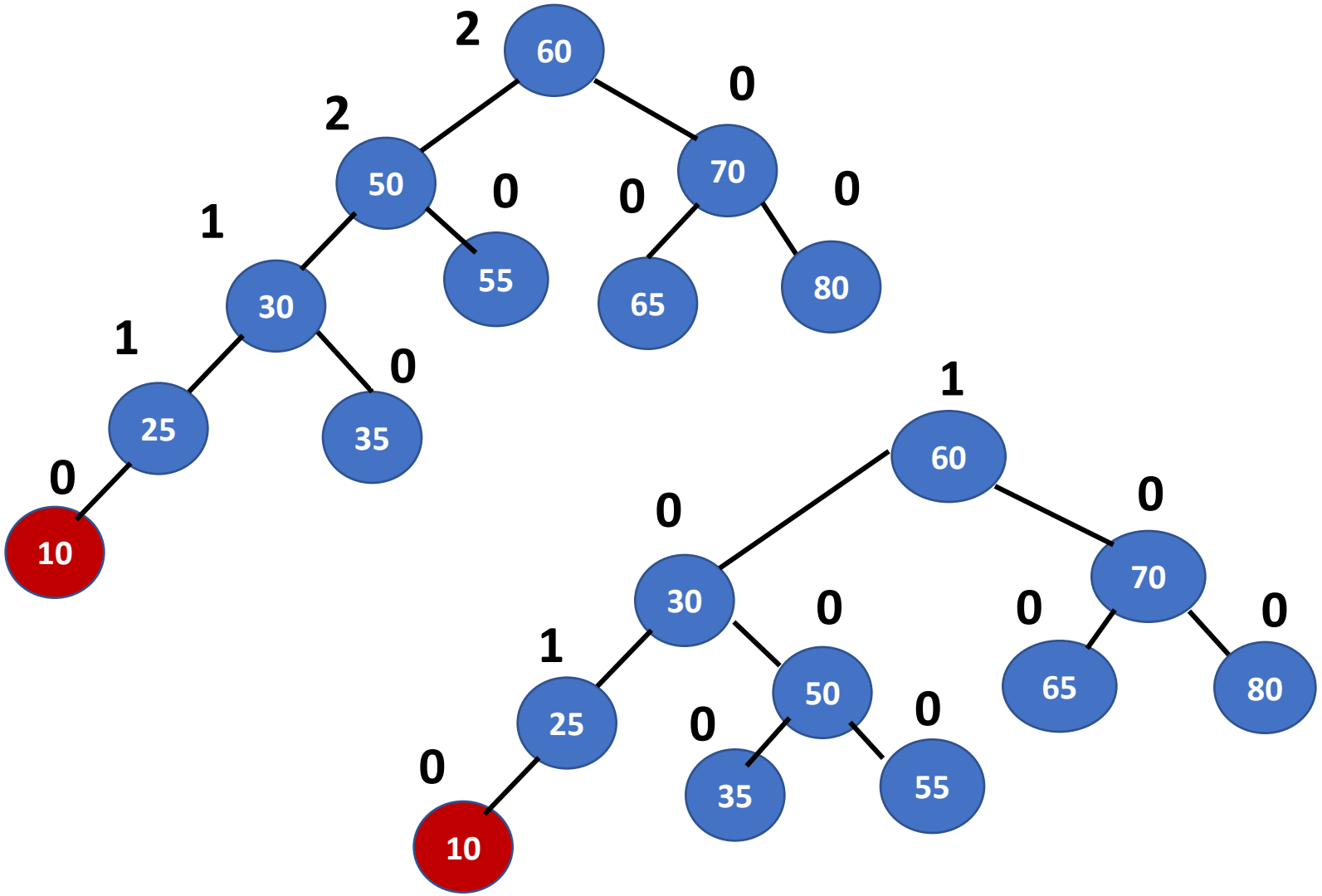


- Root r had the balance of -1 before the insertion
- New key is inserted to the left of right child (Right-Left-case)
- We perform the right rotation of right subtree of root r
- Left rotation of the new tree rooted at r

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

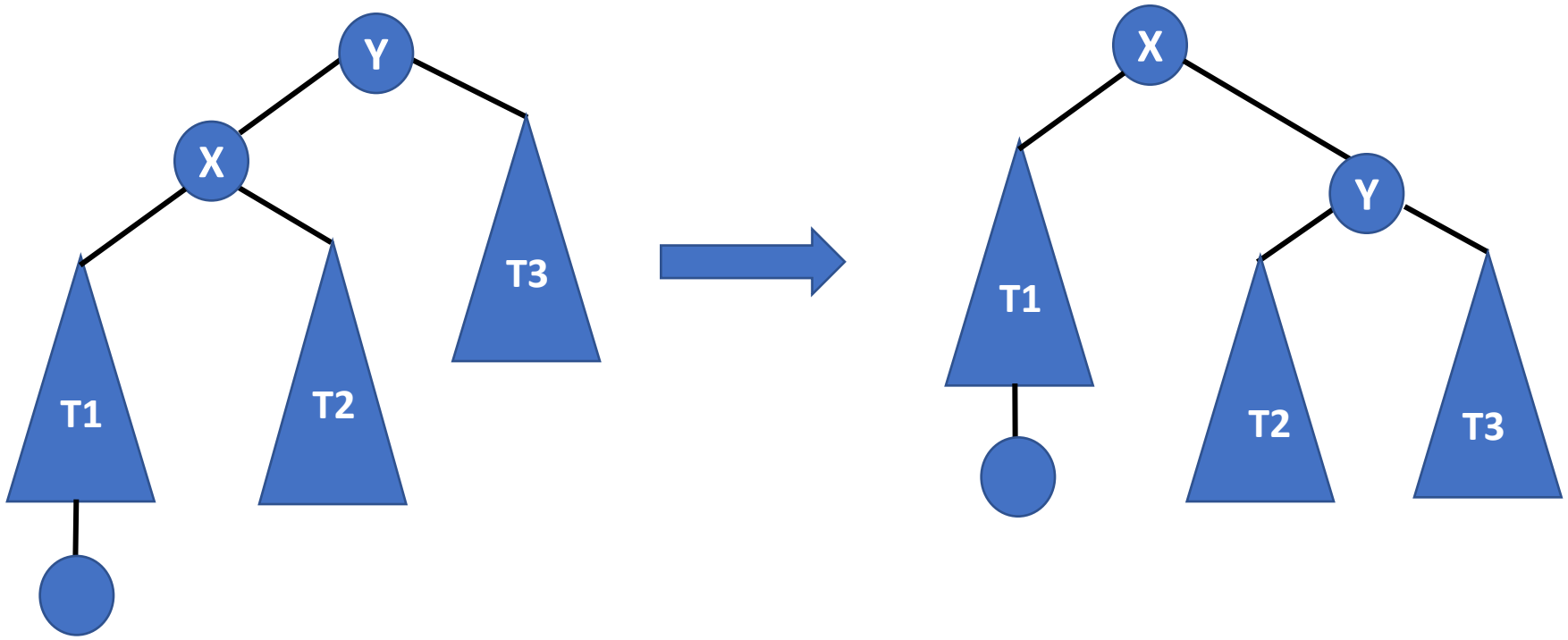
DATA STRUCTURES AND ITS APPLICATIONS

Example – Rotations in AVL Trees



Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

General form of Right-Rotation

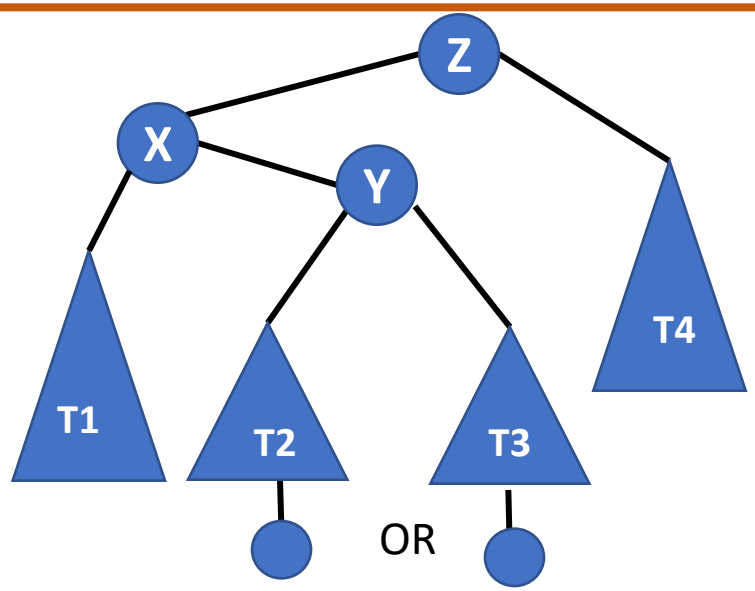


- keys(T1) < X < keys(T2) < Y < keys(T3)

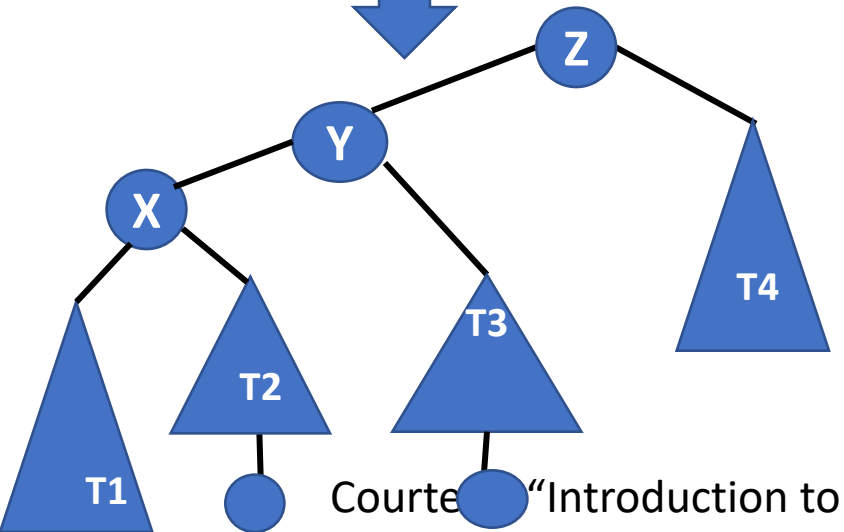
Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

DATA STRUCTURES AND ITS APPLICATIONS

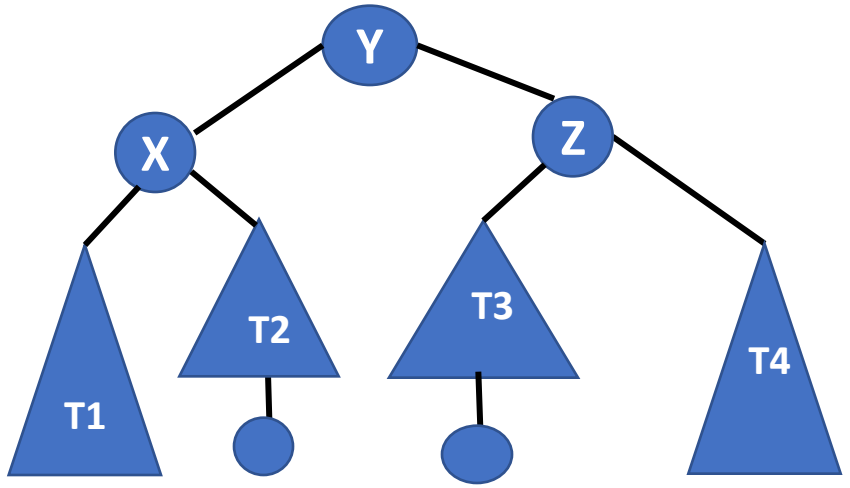
General form of Left – Right Rotation



OR



- keys(T1) < X <keys(T2) < Y < keys(T3) < Z < keys(T4)



Courte "Introduction to Design and Analysis of Algorithms" By Anany Levitin

THANK YOU

Sandesh B. J
Department of Computer Science & Engineering

sandesh_bj@pes.edu

+91 80 6618 6623

DATA STRUCTURES AND ITS APPLICATIONS

Balanced Trees

Sandesh B. J
Department of Computer Science & Engineering

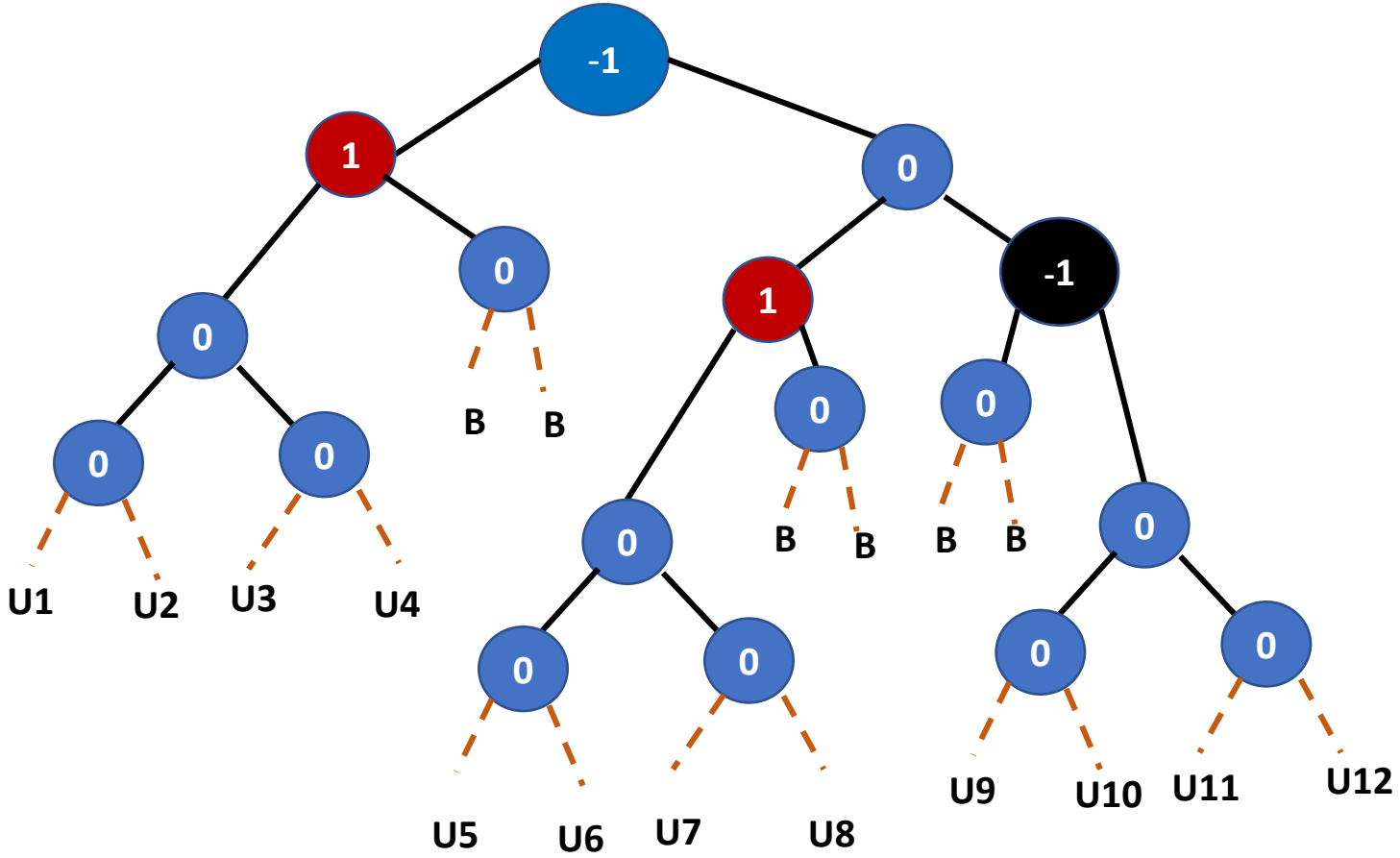
DATA STRUCTURES AND ITS APPLICATIONS

Balanced Trees

Sandesh B. J
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

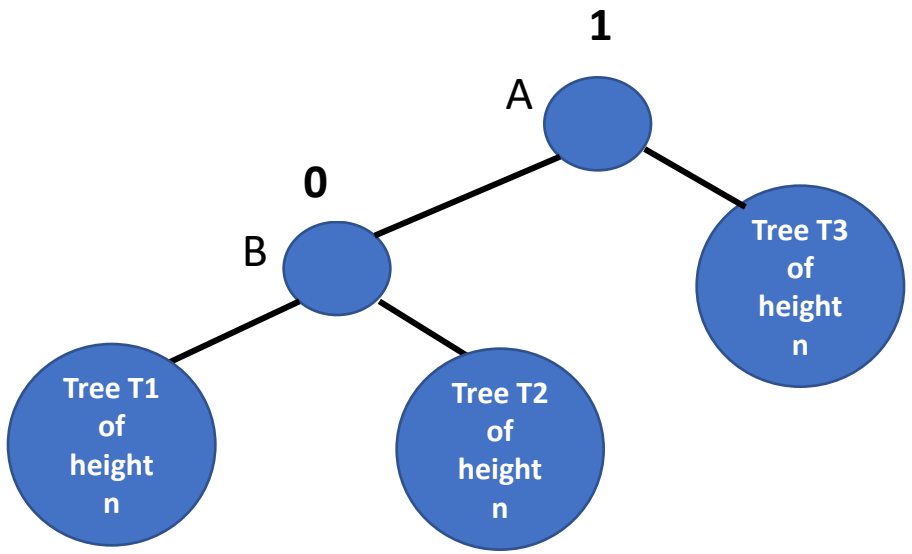
Possible insertion into AVL tree



- Unbalanced insertions are indicated by **U**
- Balanced insertions are indicated by **B**

Courtesy: “Data Structures using c and c++ ” By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

AVL tree Insertions



Balance factor(A) = (n+1) – n =1

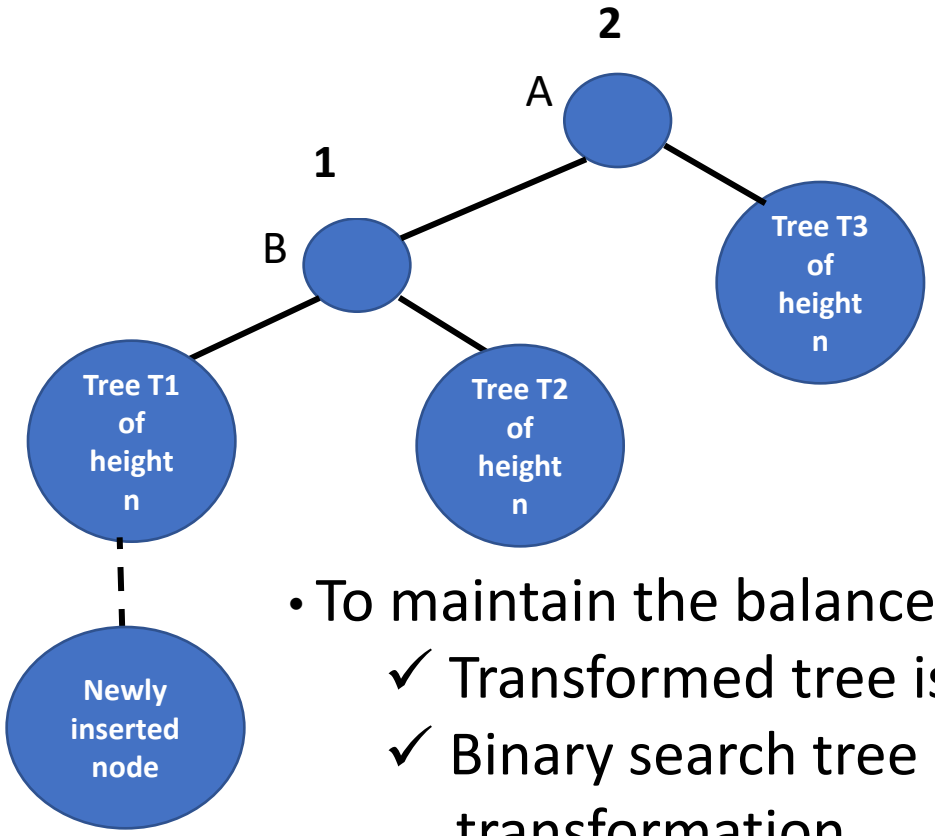
Balance factor(B) = n – n = 0

- Let us consider A is the youngest ancestor which becomes unbalanced
- Balance factor of A should be 1 before insertion
- A should have a left child B with the balance factor of 0

Courtesy: “Data Structures using c and c++ ” By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

Unbalanced Tree after inserting a node to left subtree

- Newly inserted node is left descendent of node A
- Changing the balance **B to 1 and A to 2**
- A is the youngest ancestor of the new node to become unbalanced

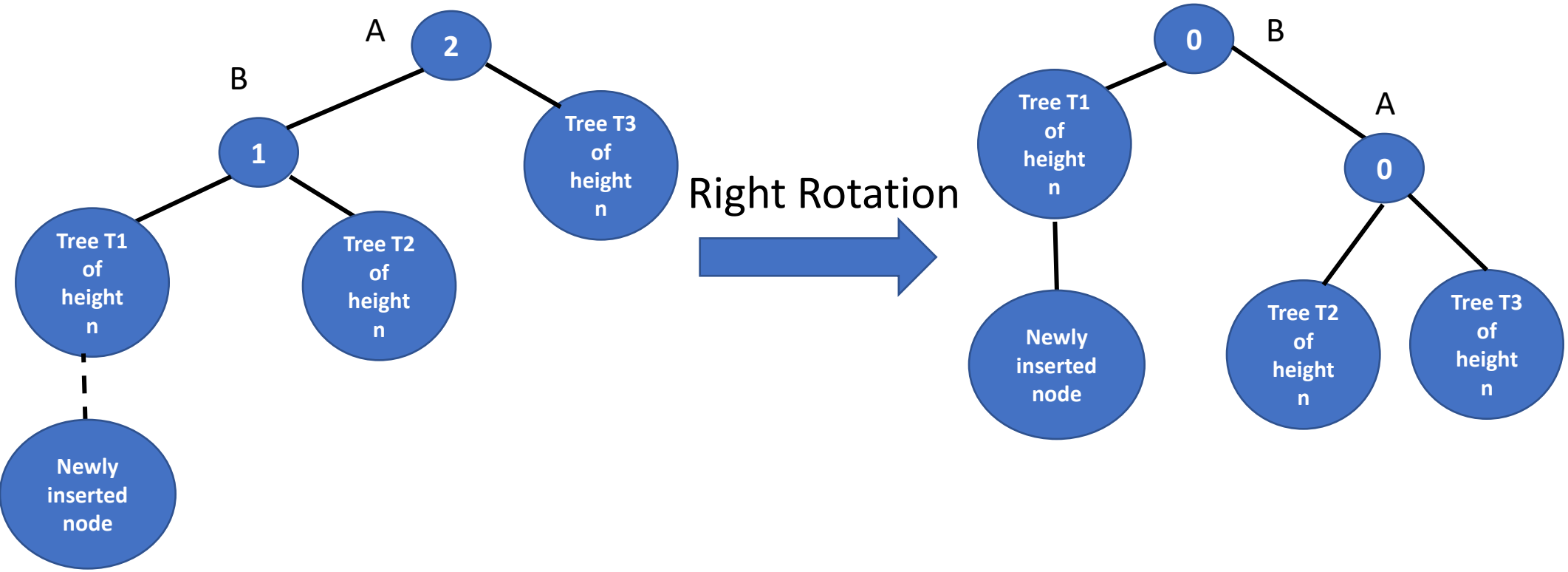


- To maintain the balance : Tree needs to be transformed
  - ✓ Transformed tree is balanced
  - ✓ Binary search tree property is maintained after transformation

Courtesy: "Data Structures using c and c++ " By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

Transformed Balanced Tree after Rotations

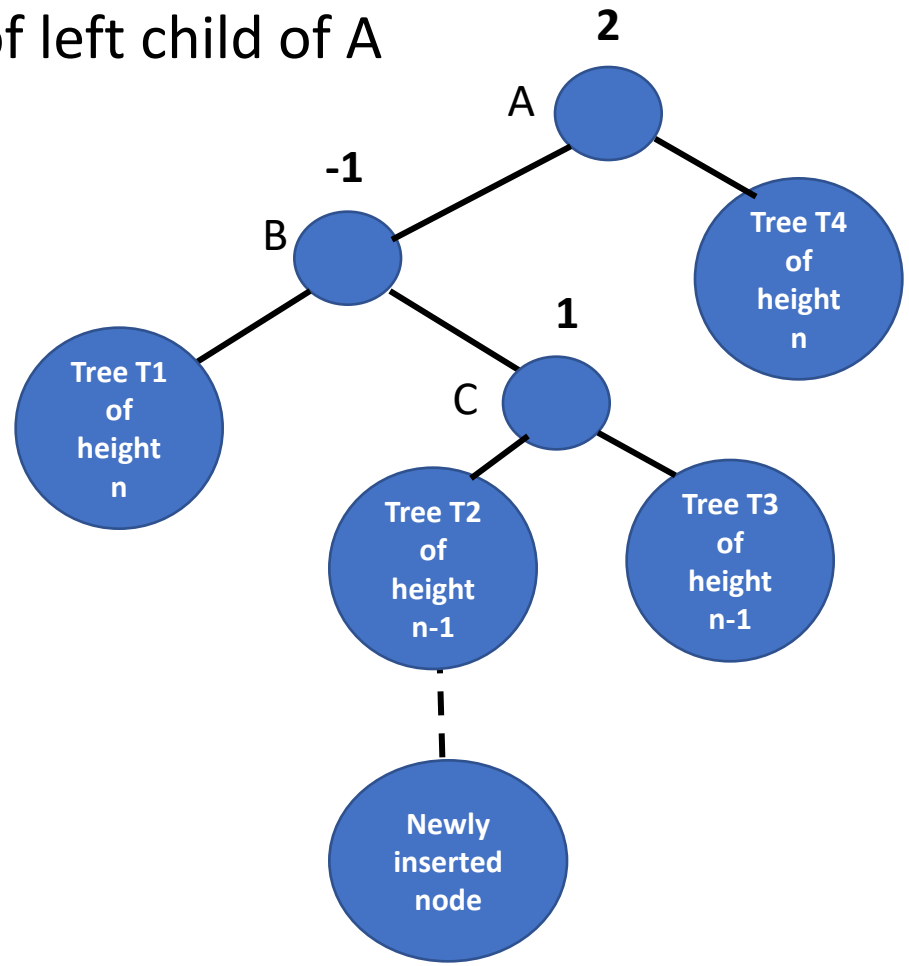
- To maintain a balance we need to rotate sub tree B rooted at node A



Courtesy: “Data Structures using c and c++ ” By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

Unbalanced Tree after inserting a node to right subtree

- Newly inserted node is left descendent of the node A
- New node is inserted into right subtree of left child of A

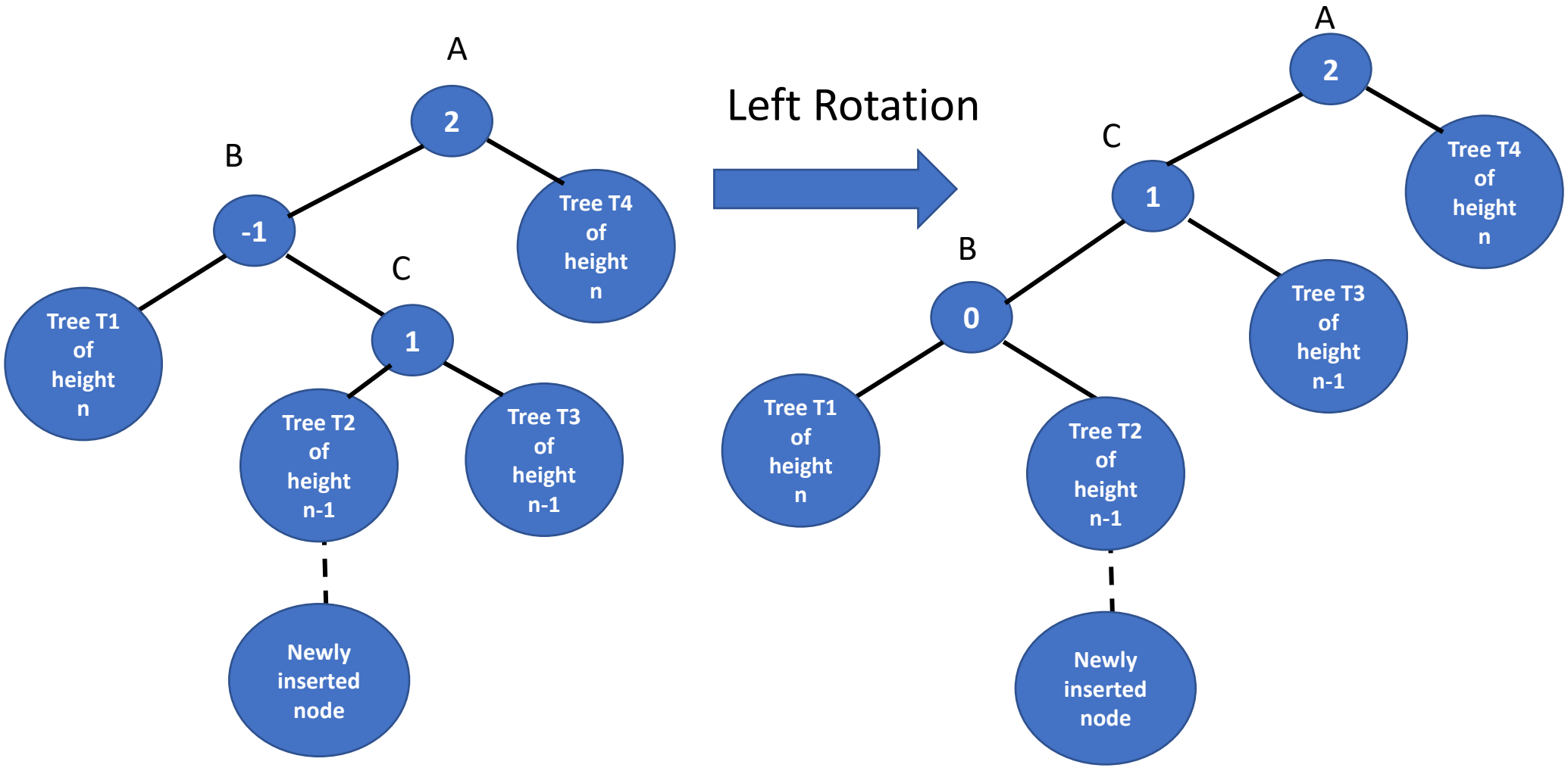


Balance factor(C)= n-(n-1) = 1
Balance factor(B)= n-(n+1) = -1
Balance factor(A)= n+2-n= 2

Courtesy: "Data Structures using c and c++ " By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

DATA STRUCTURES AND ITS APPLICATIONS

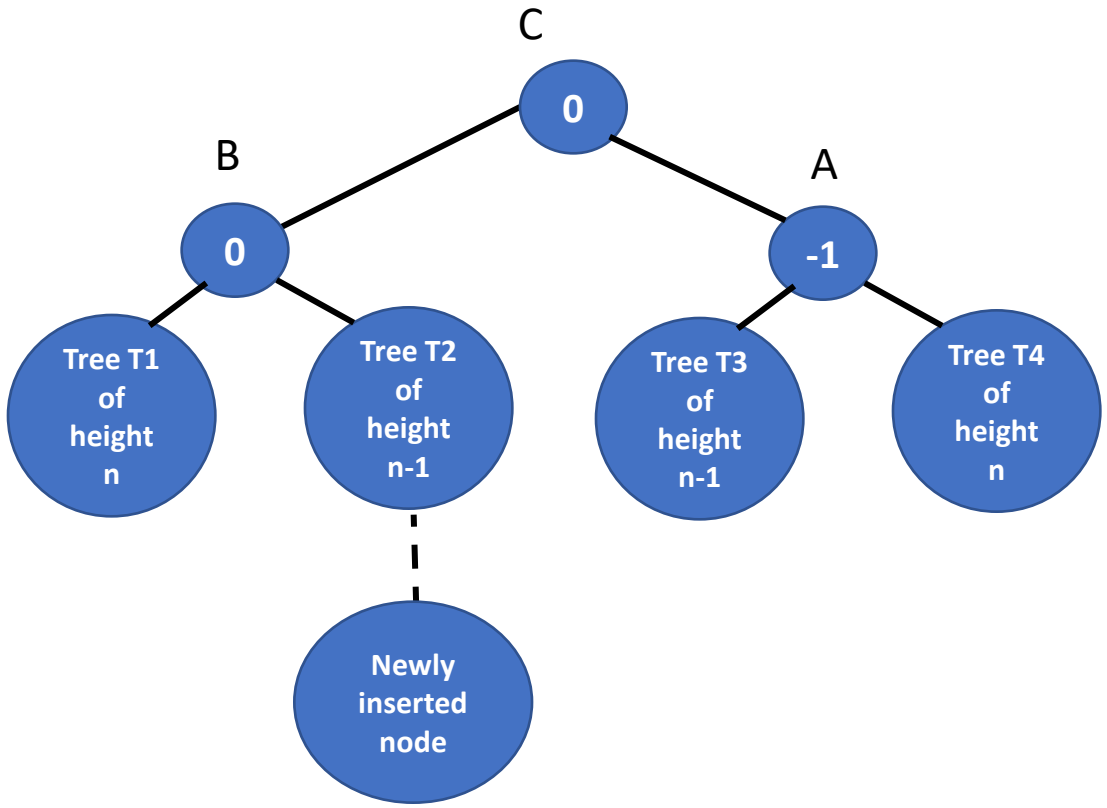
Transformed Balanced tree after Rotations



Courtesy: “Data Structures using c and c++ ” By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

Transformed Balanced tree after Rotations

Right Rotation



Courtesy: “Data Structures using c and c++ ” By Y Langsam, M. J. Augenstein and Aron M. Tenenbaum

AVL tree Insertion

- Insertion in AVL tree is performed using standard BST Insertion
- If tree becomes unbalanced, we rebalance the tree using left or right rotation
- If node X is inserted into balanced BST
- we need to find the youngest ancestor which becomes unbalanced

Four cases:

- IF(Balance factor of node) == 2 – unbalanced node(U)
  - ✓ case-1 : Left-Left case
    - IF((newly inserted key) < (key in the left subtree' root))
  - ✓ case-2: Left-Right case
    - IF((newly inserted key) > (key in the left subtree' root))

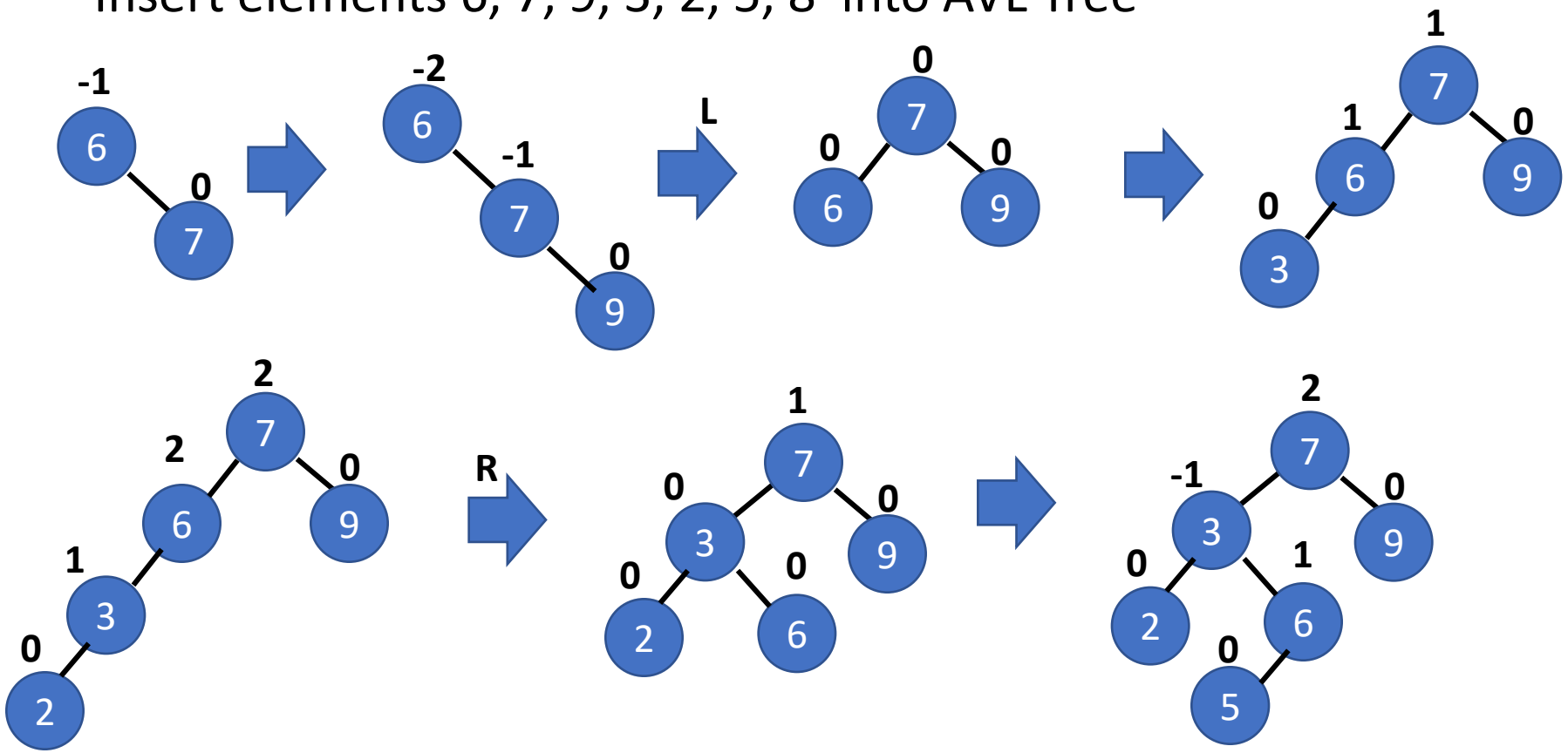
AVL tree Insertion

Four cases:

- IF((Balance factor of node)) == -2 – unbalanced node(U)
  - ✓ Case 3: Right-Right case
    - IF((newly inserted key) > (key in the right subtree' root))
  - ✓ Case 4:Right-Left case
    - IF((newly inserted key) < (key in the right subtree' root))

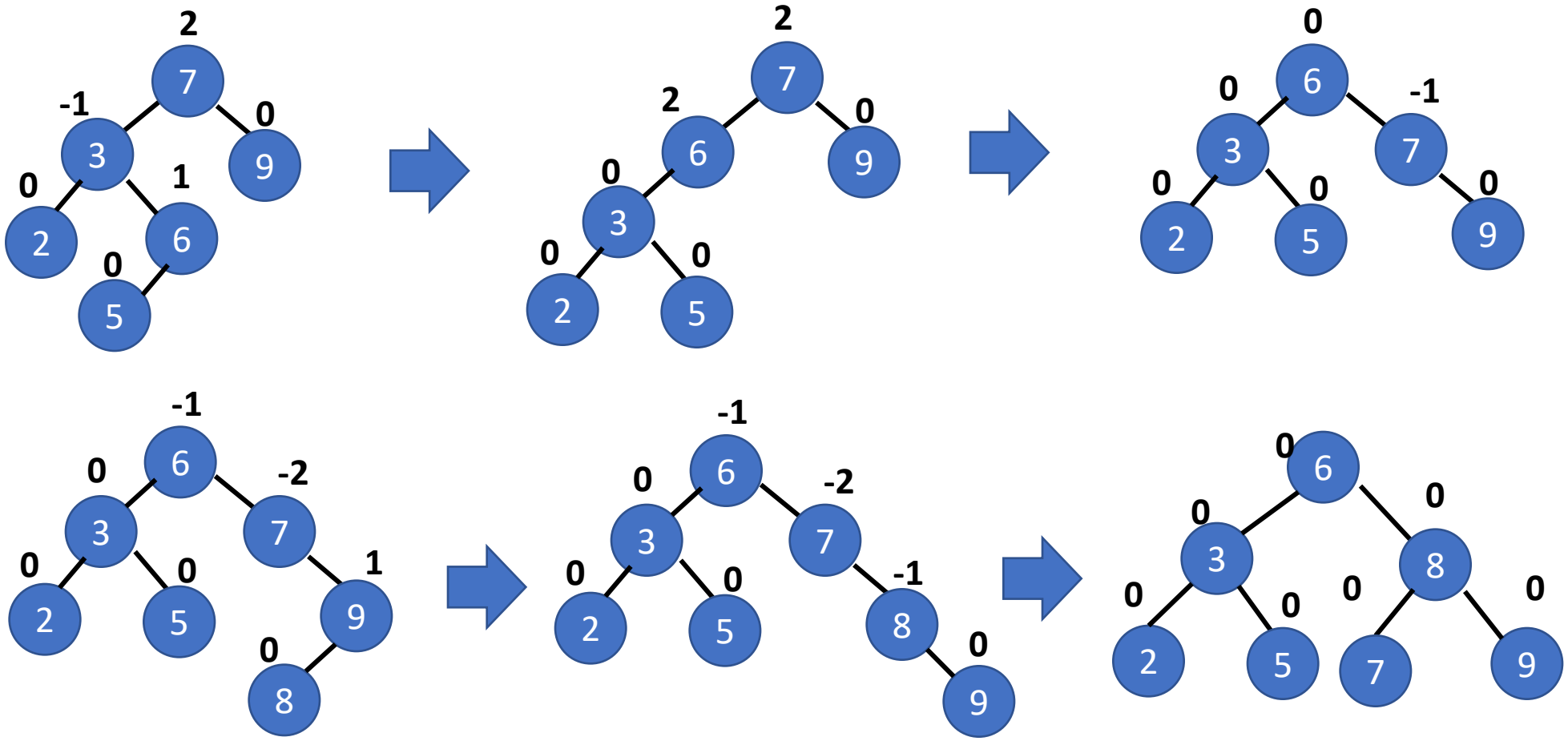
Examples – AVL Tree Insertions

- Insert elements 6, 7, 9, 3, 2, 5, 8 into AVL Tree



Example – AVL Tree Insertions

- Insert elements 6, 7, 9, 3, 2, 5, 8 into AVL Tree



Deletions in AVL tree

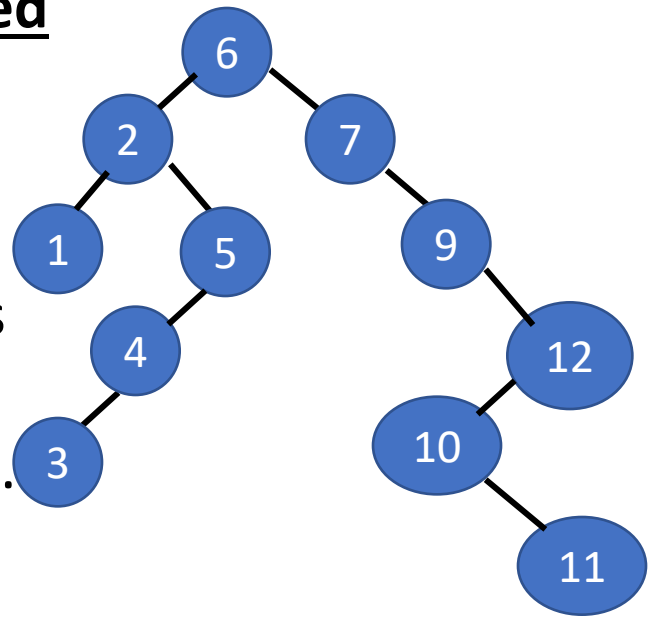
- Deletion in AVL tree is performed using standard BST Deletion
- If tree becomes unbalanced, we rebalance the tree using left or right rotation

BST Deletion: 3- case: Node to be delted

Case 1: Does not have any children

Case 2: has either left or right subtrees

Case 3: has both left and right subtrees.

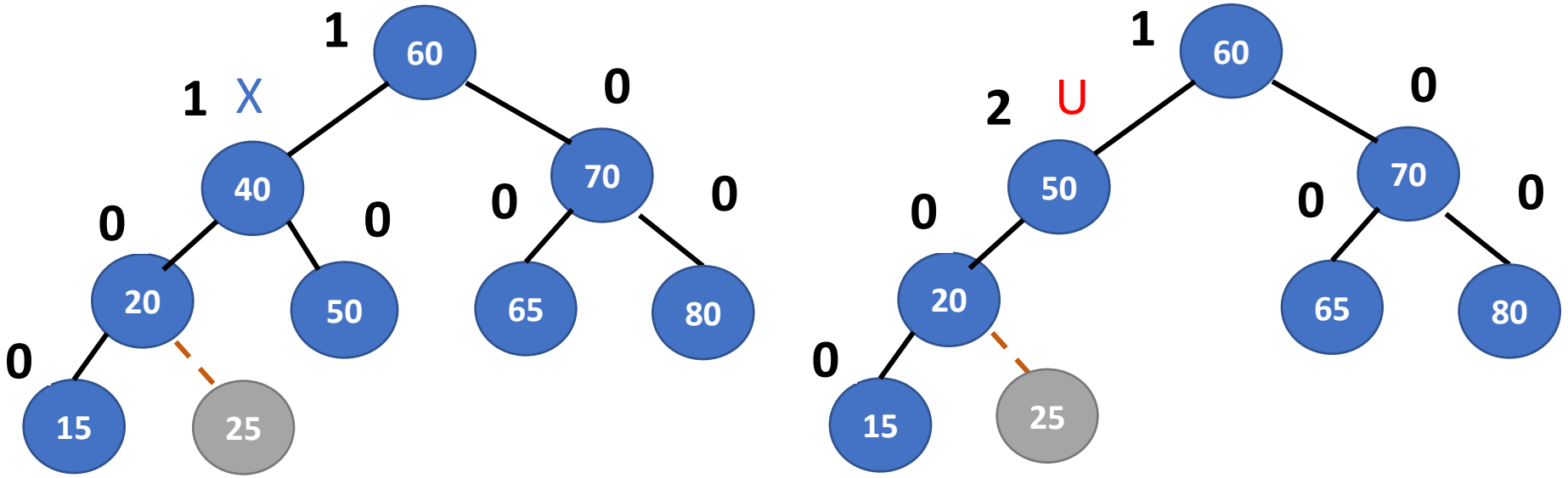


Deletions in AVL tree

- If node X is deleted from the BST
- we need to find the youngest ancestor which becomes unbalanced

Four cases: Case-1

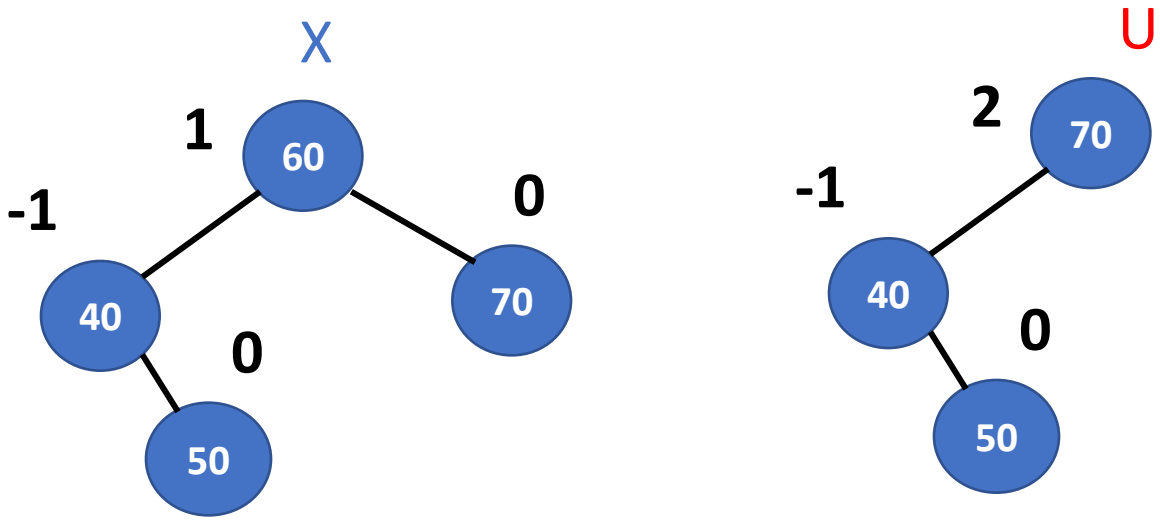
- IF((Balance factor of a node) == 2) - unbalanced node(U)
  - ✓ Left-Left case:
    - IF(Balance factor of left subtree's root) >= 0



Deletions in AVL tree

Case-2:

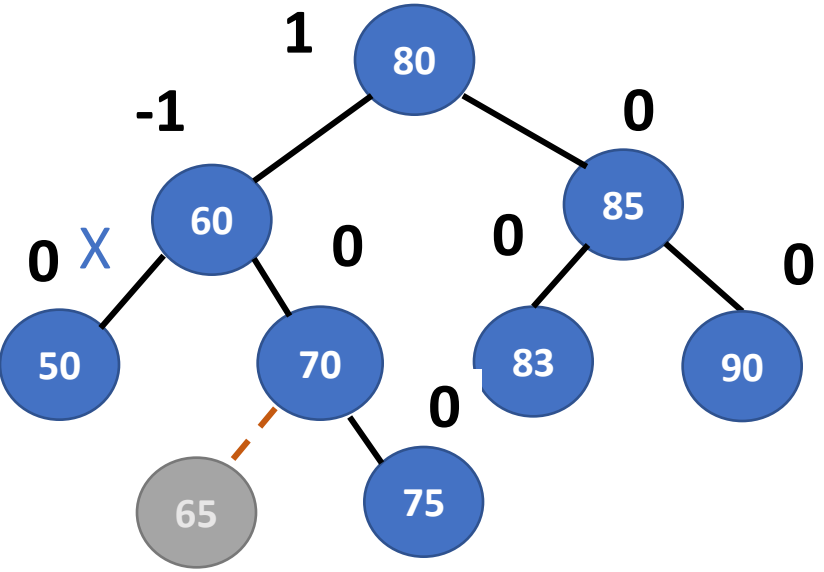
- IF(Balance factor of a node == 2) – unbalanced node(U)
  - ✓Left-Right case
    - IF(Balance factor of left subtree's root) < 0

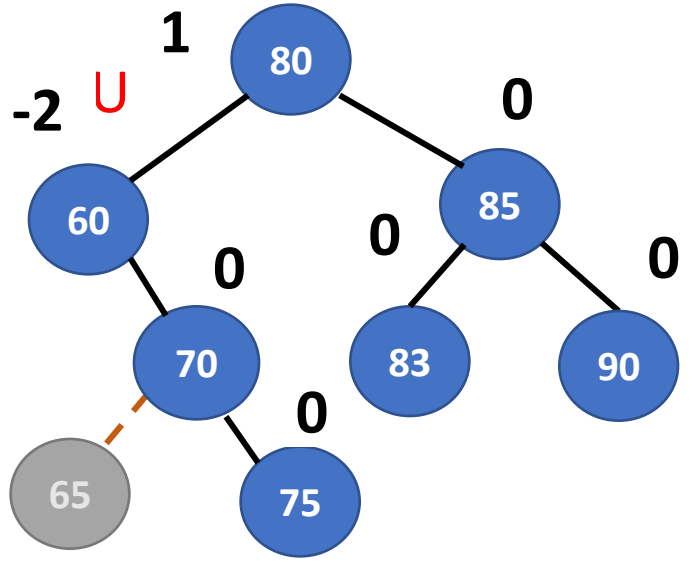


Deletions in AVL tree

Case-3:

- If ((Balance factor of a node) == -2) unbalanced node(U)
  - ✓Right-Right case
    - IF(Balance factor of Right subtree's root) <= 0

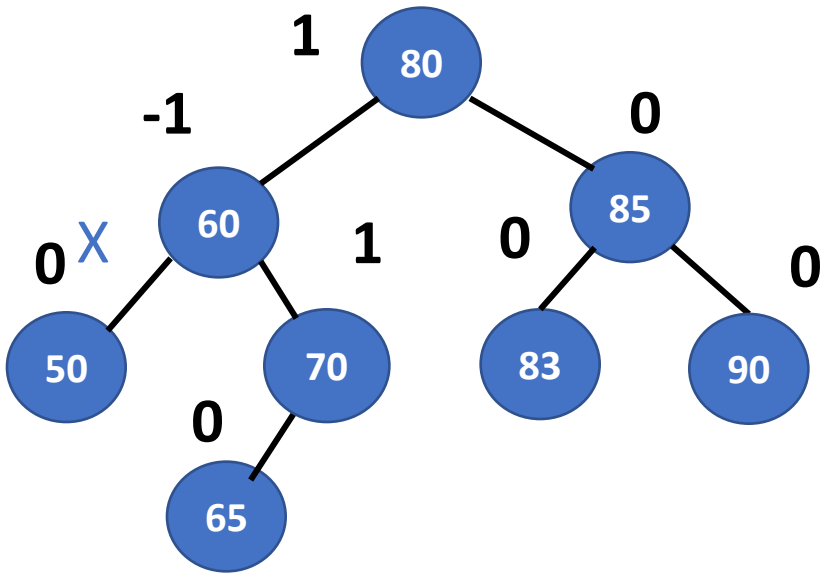


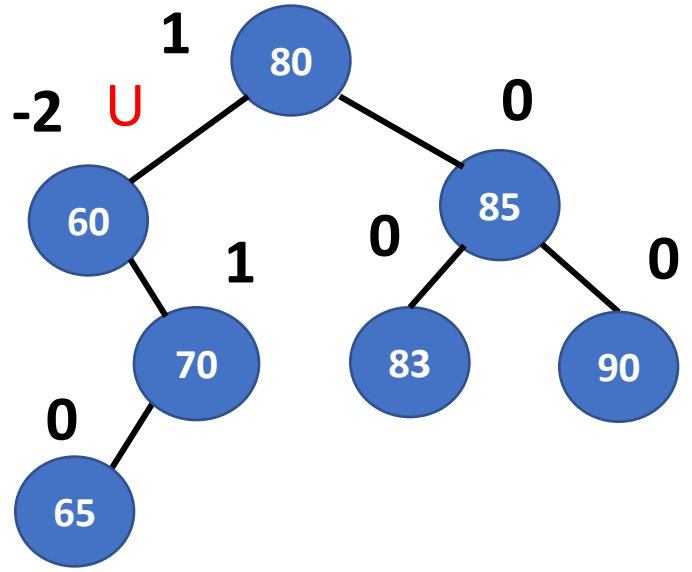


Case-4:

IF((Balance factor of unbalanced node) == -2) unbalanced node(U)

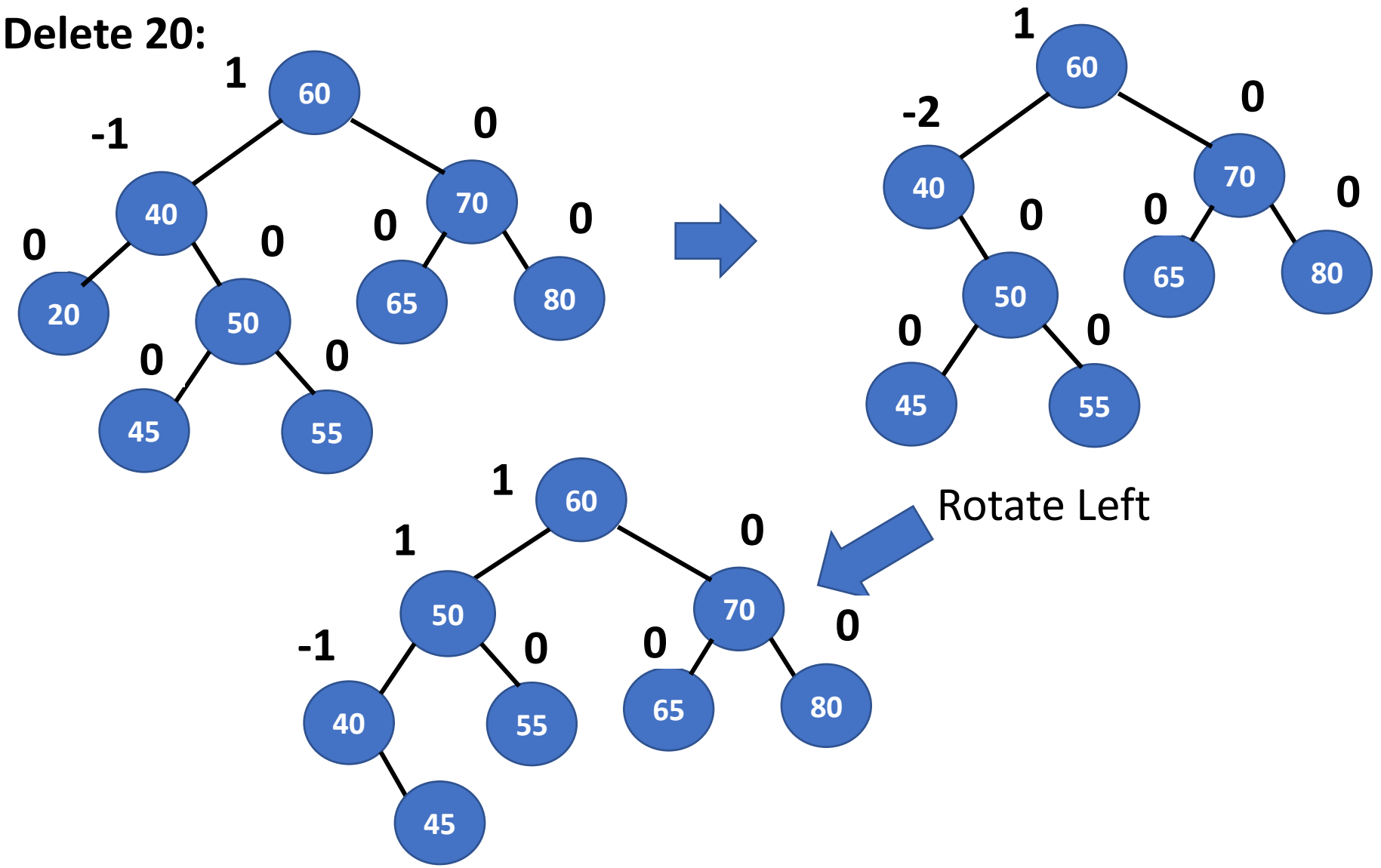
- ✓Right-Left case
  - IF(Balance factor of Right sub tree's root) > 0





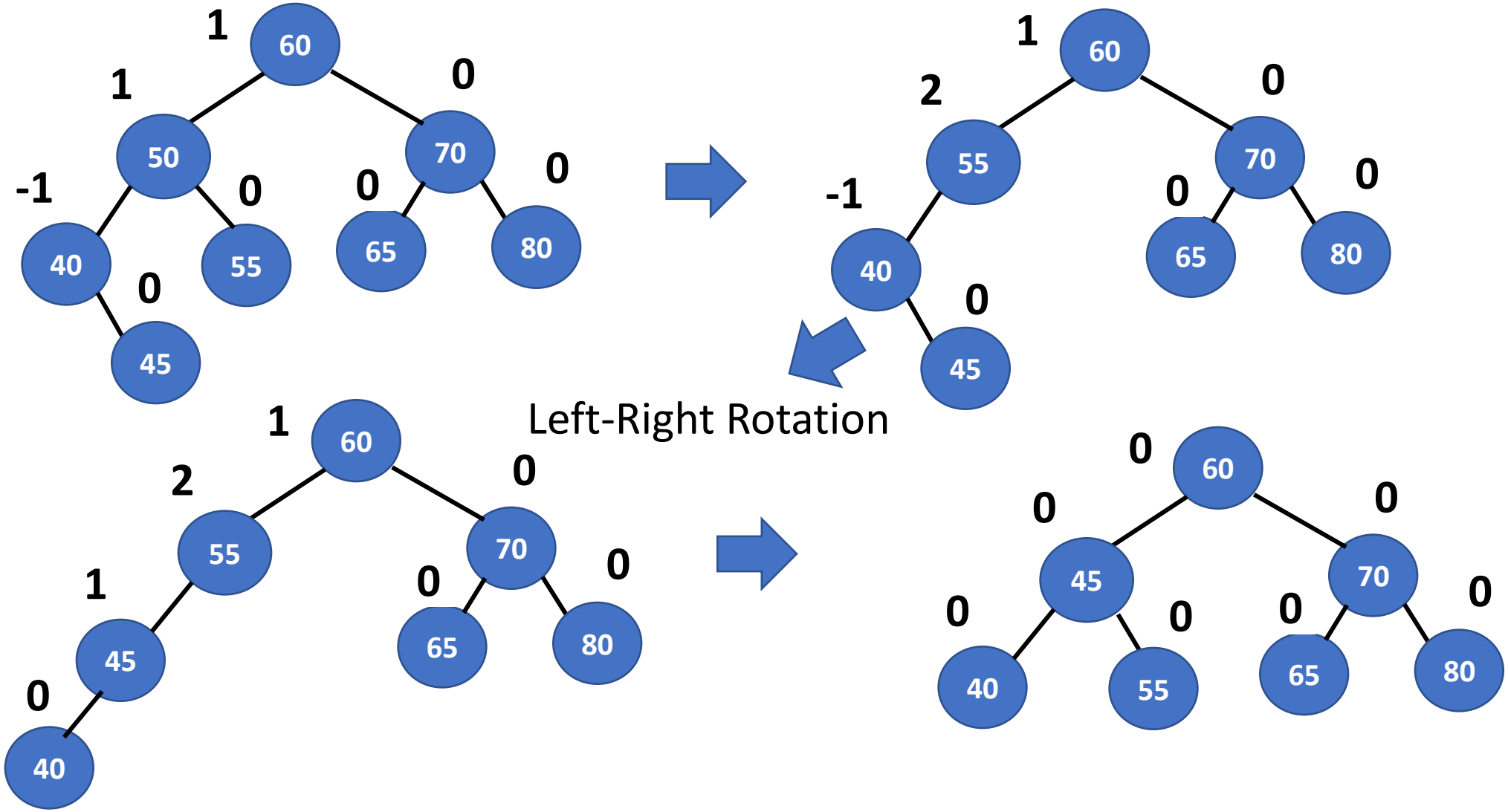
Example – Deletions in AVL tree

Delete 20:



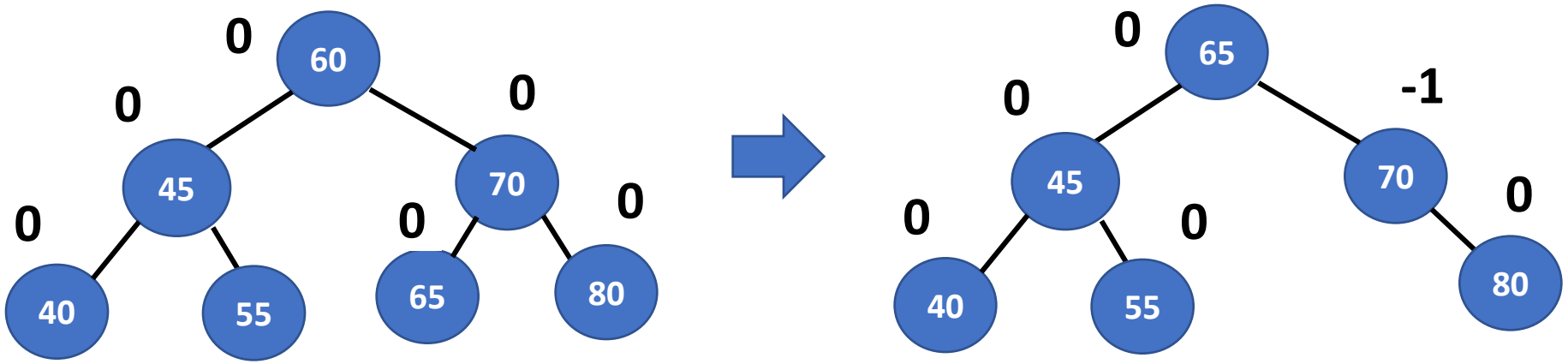
Example – Deletions in AVL tree

Delete 50:

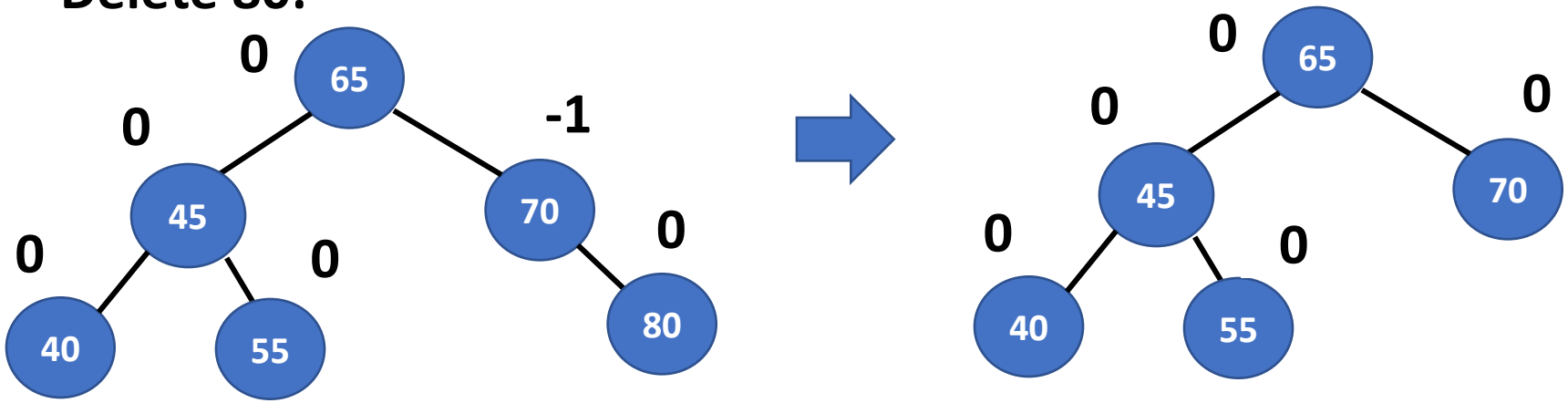


Example – Deletions in AVL tree

Delete 60:

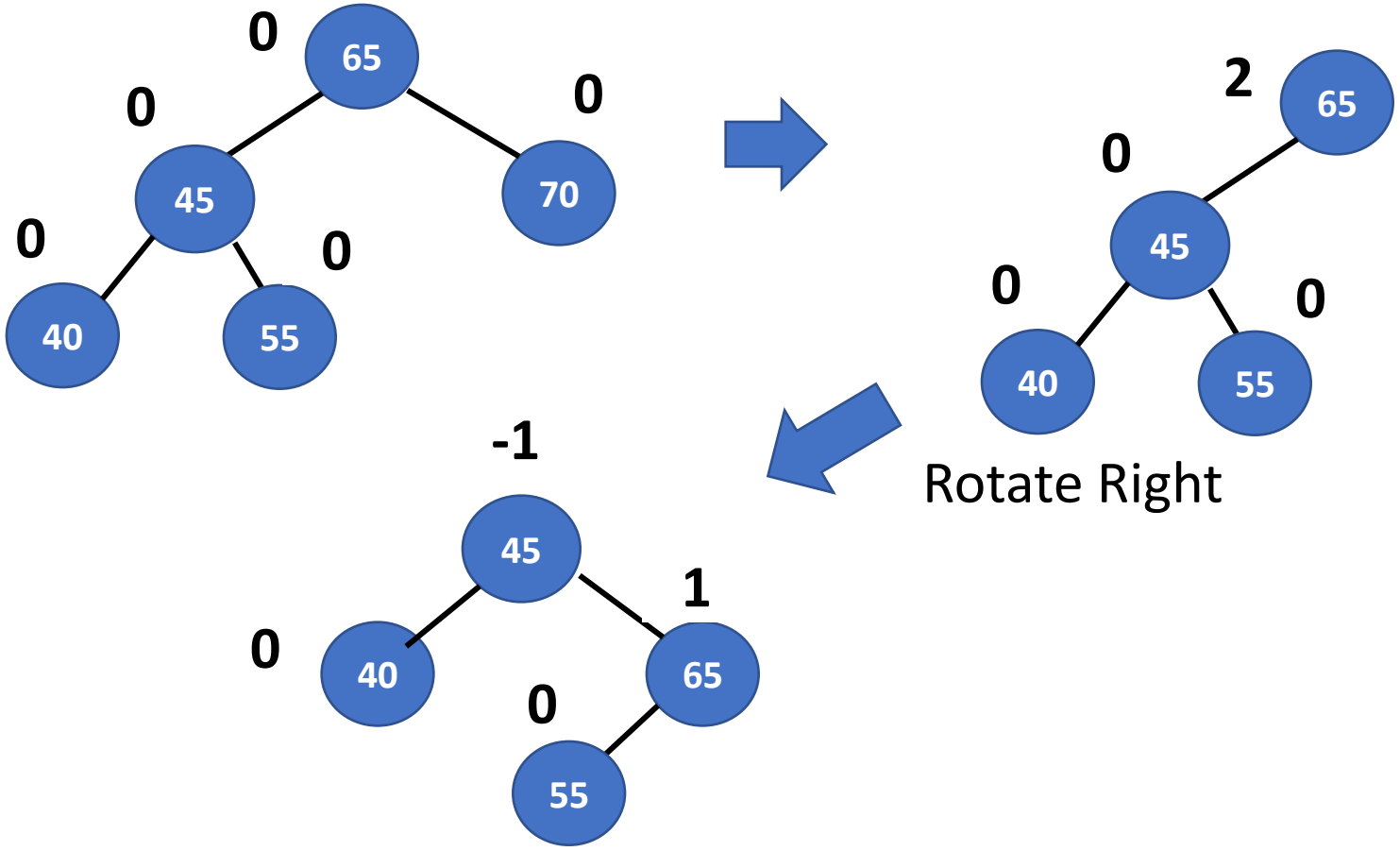


Delete 80:



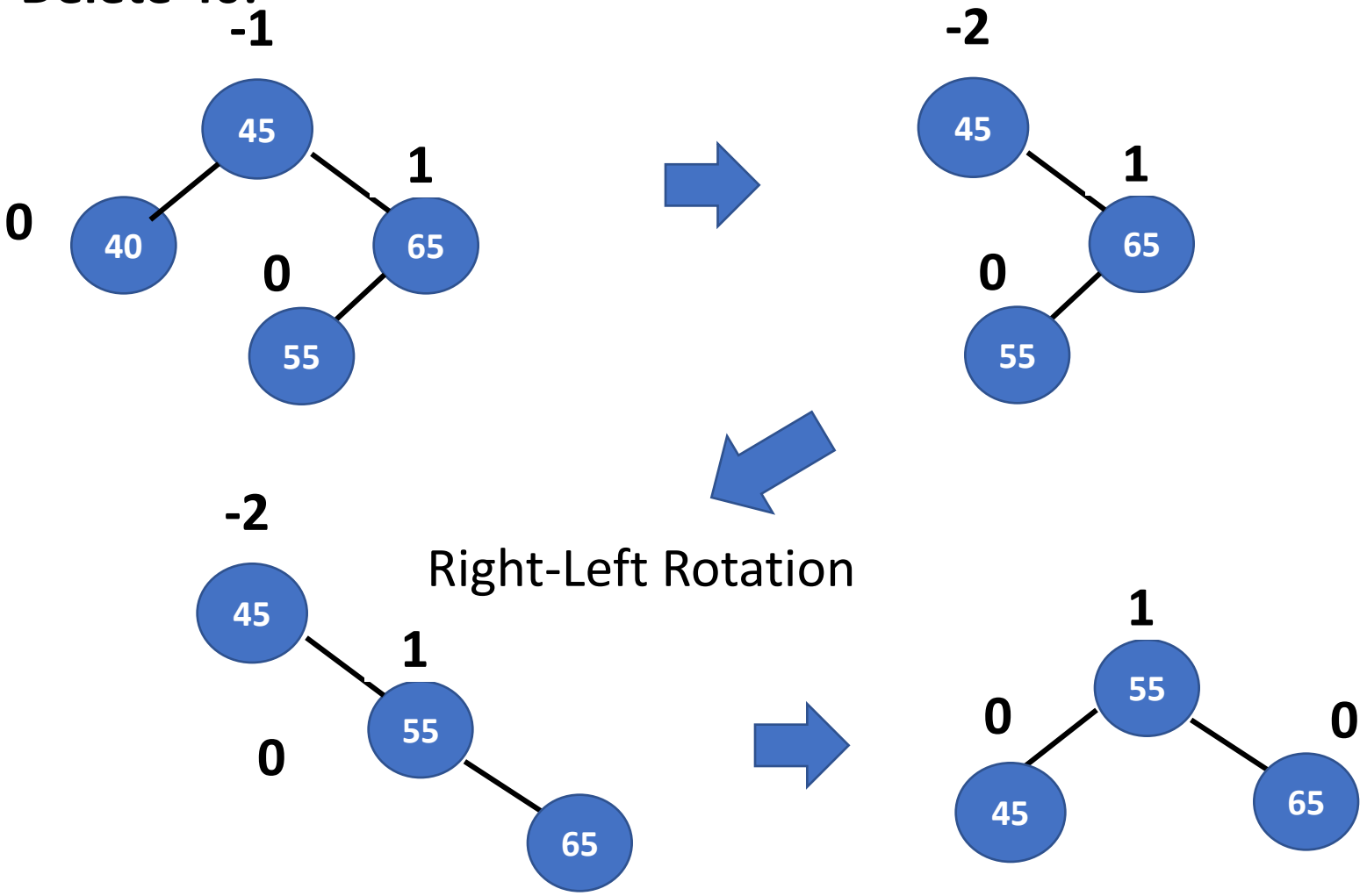
Example – Deletions in AVL tree

Delete 70:



Example – Deletions in AVL tree

Delete 40:



THANK YOU

Sandesh B. J

Department of Computer Science & Engineering

Sandesh_bj@pes.edu

+91 80 6618 6649

DATA STRUCTURES AND ITS APPLICATIONS

Graphs

Sandesh B. J & Saritha

Department of Computer Science & Engineering

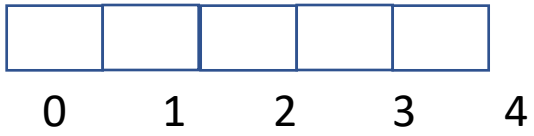
DATA STRUCTURES AND ITS APPLICATIONS

Graphs

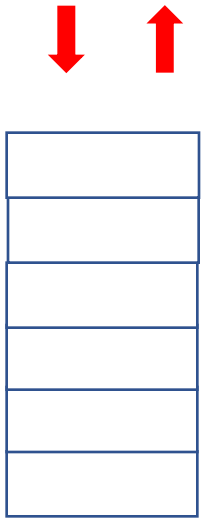
Saritha
Department of Computer Science & Engineering

Linear data structures

Array

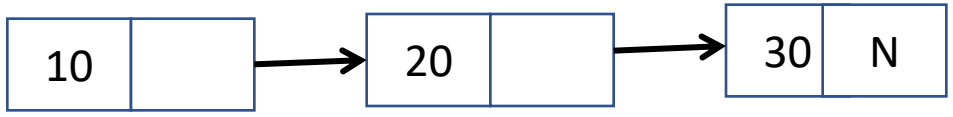


Stack



Queue

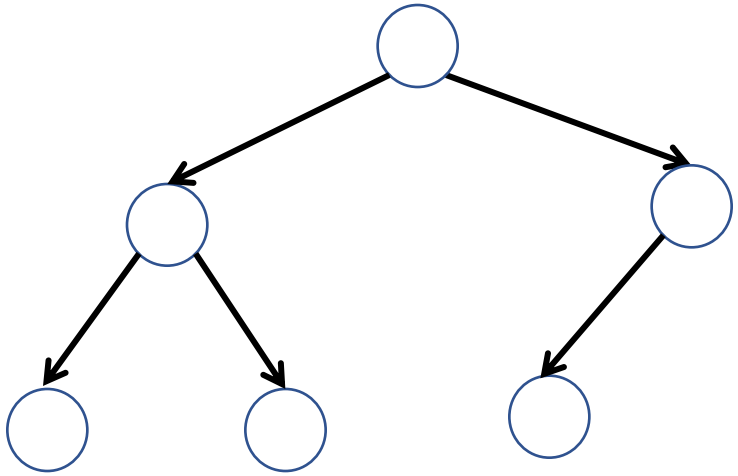
Linked List



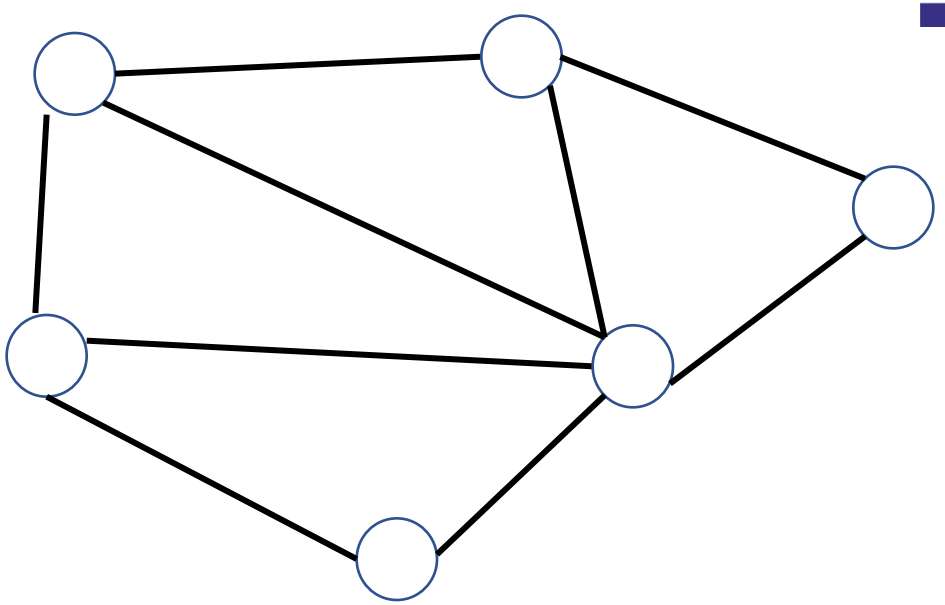
DATA STRUCTURES AND ITS APPLICATIONS

Introduction to Graphs

Non-Linear Data Structure



Tree



Graph

In a tree with N nodes there are N-1 edges

Introduction to graph

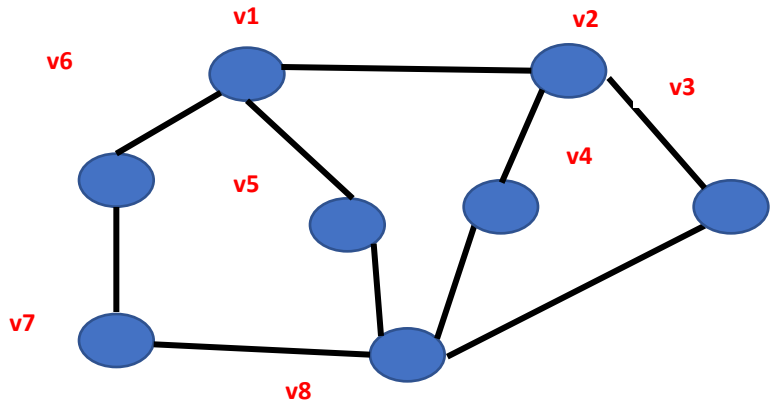
Graphs

- A Graph is a data structure that consists of set of vertices and a set of edges that relate the node to each other.
- The set of edges represents the relationship among the vertices.
- A graph G is defined as

G=(V,E)

V: finite nonempty set of vertices

E: a set of edges



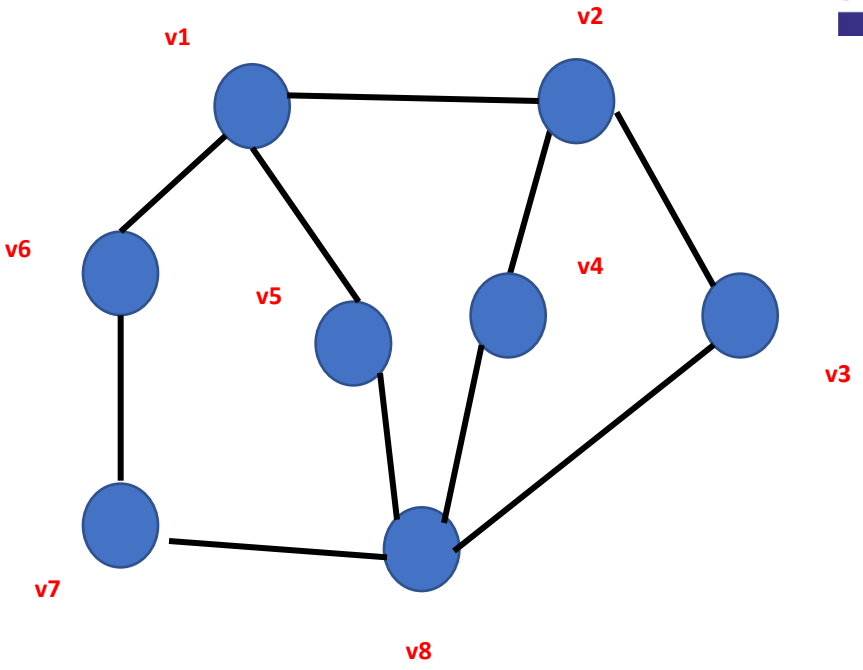
V = { v1,v2,v3,v4,v5,v6,v7,v8}

DATA STRUCTURES AND ITS APPLICATIONS

Representation of Edge

V = { v1,v2,v3,v4,v5,v6,v7,v8}

E={(v1,v2),(v2,v3),(v2,v4),(v4,v8),(v1,v5),(v1,v6),(v6,v7),(v5,v8)}



Types of Graphs

Undirected Graph:

- A graph is undirected, when the pair of vertices representing any edge is unordered.

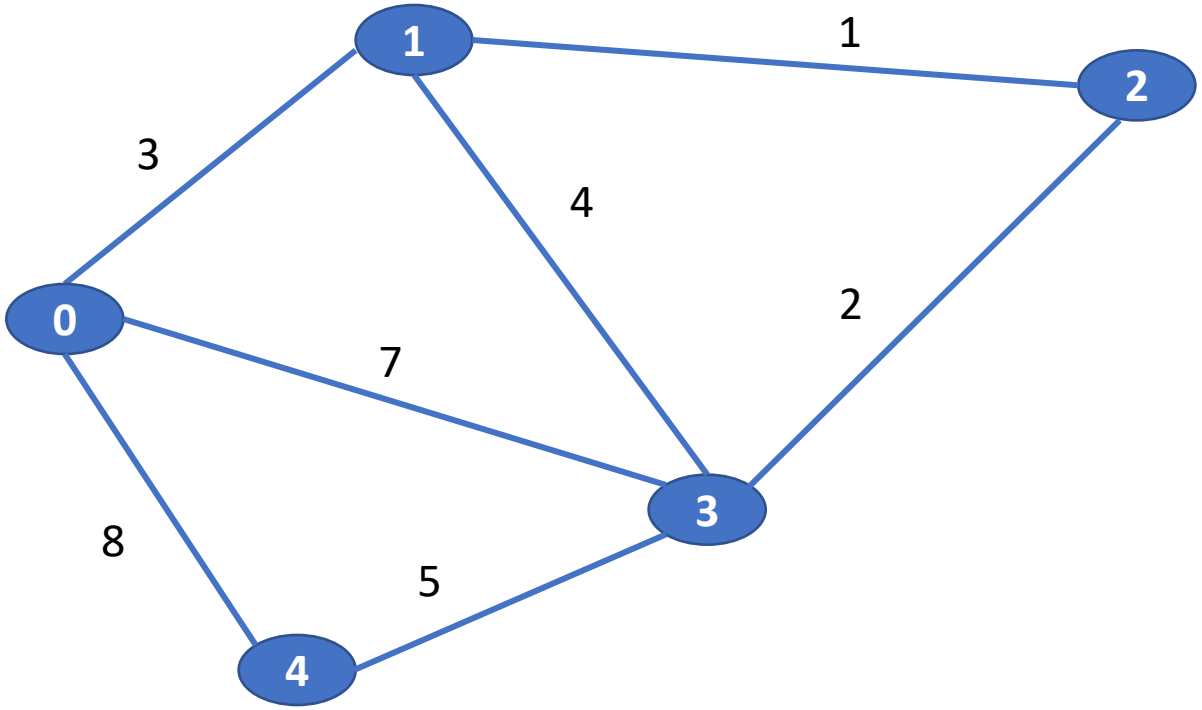
Directed Graph:

- A graph with all directed edges is called diagraph or directed graph.

Types of Graphs

Weighted Graph:

• A weighted graph is a graph where each edge has a numerical value called weight.



Graph terminologies

Adjacent Nodes :

- A node n is adjacent to node m if there is an edge from m to n.
- if n is adjacent to m, then n is called the **successor** of m and m is called the **predecessor** of n.

For example : a is adjacent to b

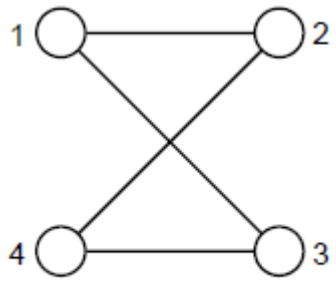
Path:

Path is a sequence of vertices that connect two nodes in a graph.

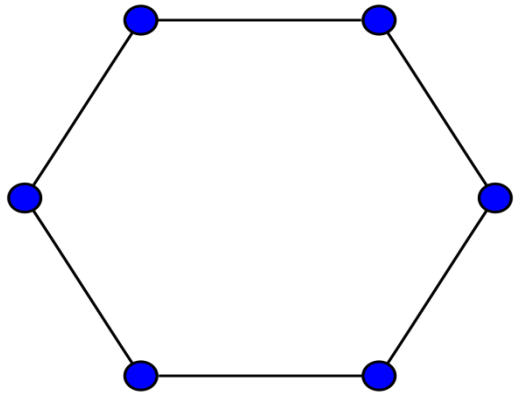
Graph terminologies

Cycle :

- A path from node to itself is called a cycle or cycle is path in which first and last vertices are same. A graph with at-least one cycle is called cyclic graph. For example the below graph are cyclic graphs



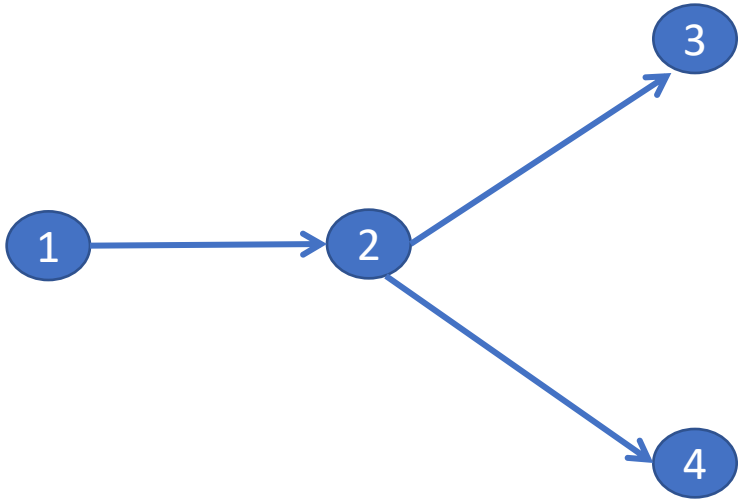
Cycle



Graph terminologies

Acylic :

- A graph with no cycles is called acyclic graph. A directed acyclic graph is called dag. For example below graph is a directed acylic graph



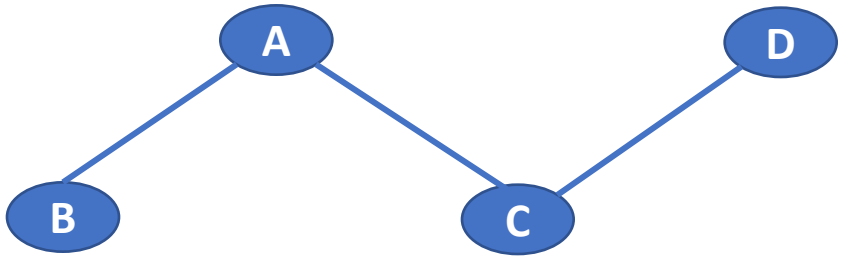
Graph terminologies

Incident:

A node n is incident to an edge x, if node is one of the two nodes the edge connects.

Degree:

The degree of vertex i is the number of edges incident on vertex i.



degree(A)=2,degree of(D)=1

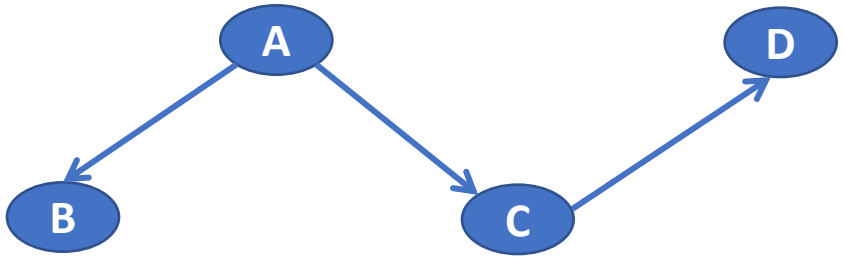
Graph terminologies

In-degree:

- In-degree of vertex i is the number of edges incident to i.

Out-degree:

- Out-degree of vertex i is the number of edges incident from i.



Out-degree(A)=2,in-degree of(A)=0

Out-degree(c)=1,in-degree of (c)=1

Properties of Graph

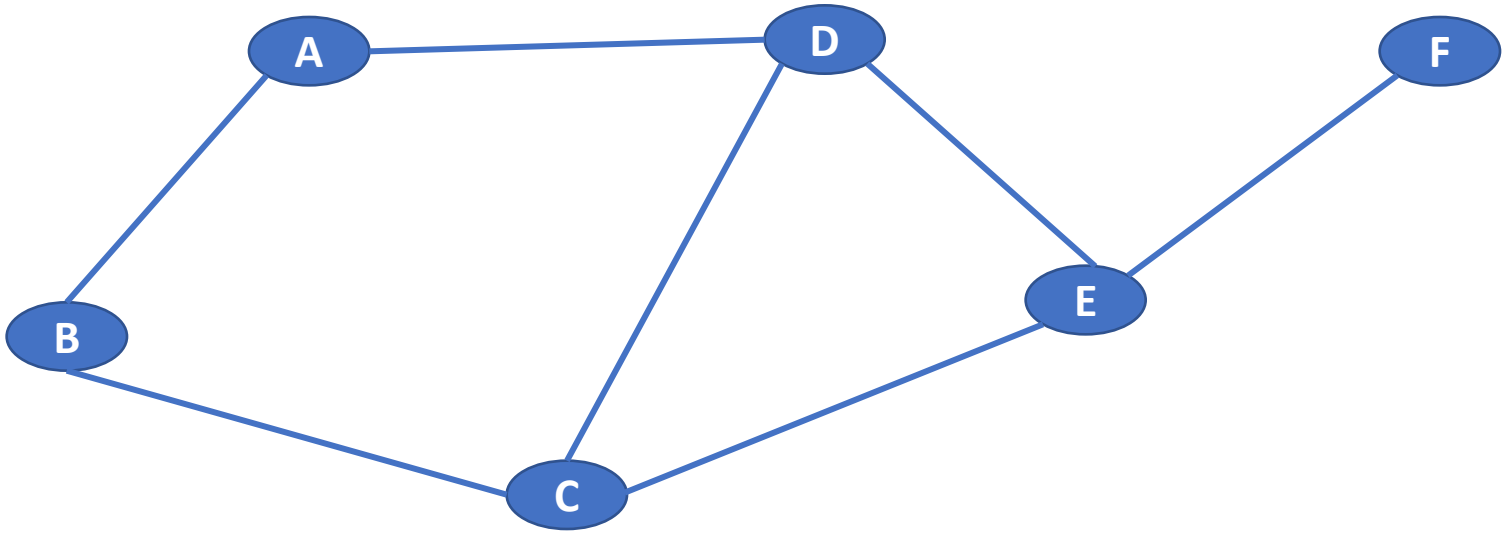
Directed graph:

- The number of possible pairs in an m vertex graph is m*(m-1)
- The number of edges in an directed graph is m*(m-1) since the edge(u, v) is not the same as the edge(v, u)
- The number of edges in an directed graph is<=m*(m-1)

Properties of Graph

Undirected graph:

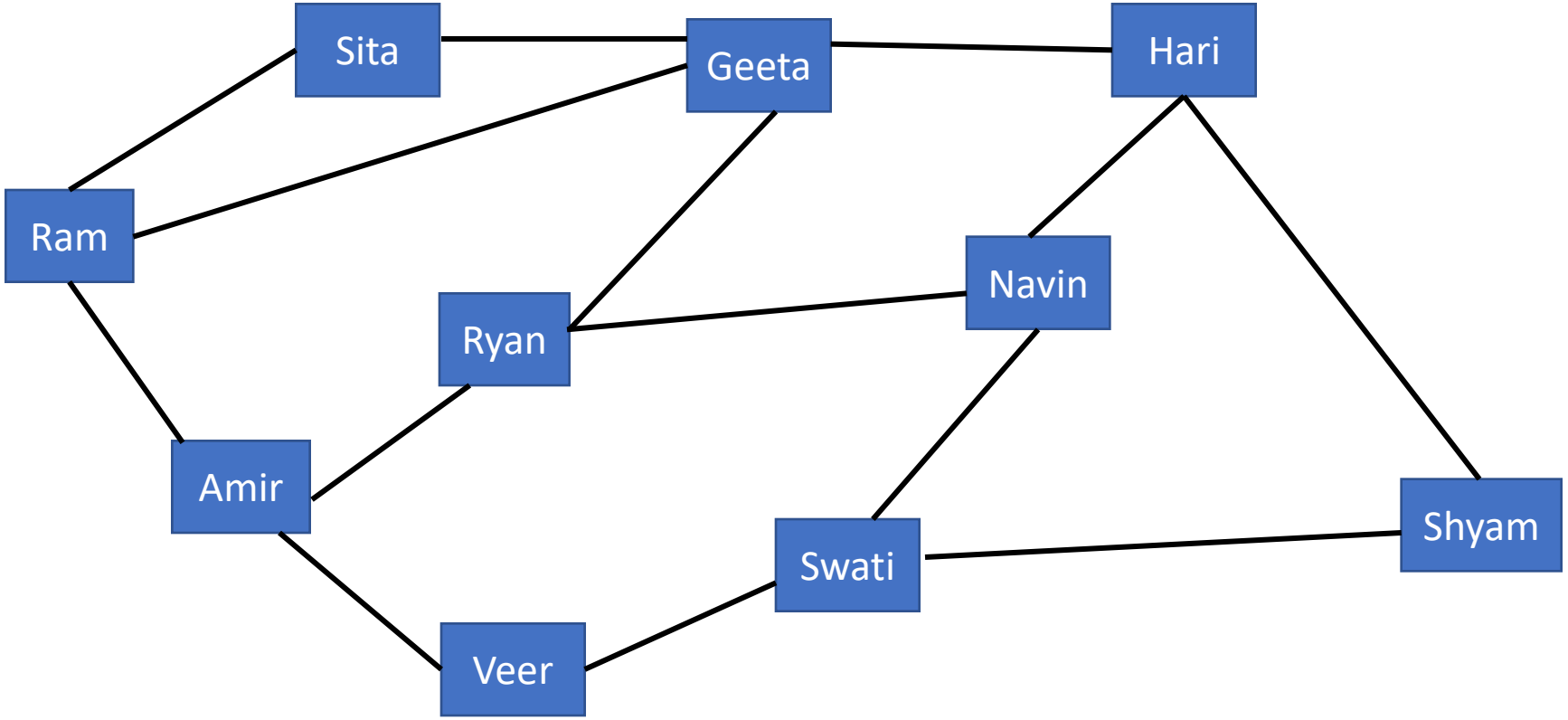
- The number of possible pairs in an m vertex graph is m*(m-1)
- The number of edges in an undirected graph is m*(m-1)/2 since the edge(u, v) is same as the edge(v, u)



DATA STRUCTURES AND ITS APPLICATIONS

Applications of graph

Social Networking sites



THANK YOU

Saritha

Department of Computer Science & Engineering

Saritha.k@pes.edu

9844668963

DATA STRUCTURES AND ITS APPLICATIONS

Graph Representation

Sandesh B. J
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Graph Representation

Sandesh B. J
Department of Computer Science & Engineering

Graph Representation

Learning objectives:

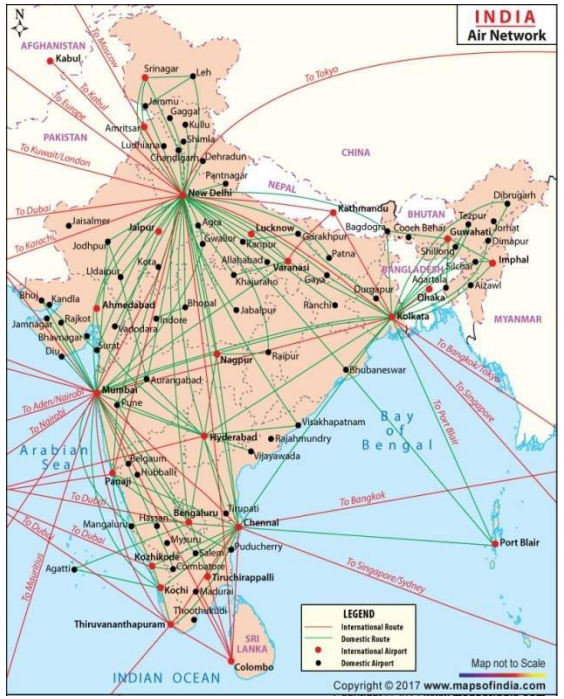
- Why we need to represent the structure of the graph in computer memory?
- How do we represent the graph ?
- Data structure used for the representation:
  - ✓Adjacency Matrix and Adjacency List
- Representation : Directed, Undirected and Weighted graphs
- Implementation details of representation using ‘c’
- Multilinked Representation

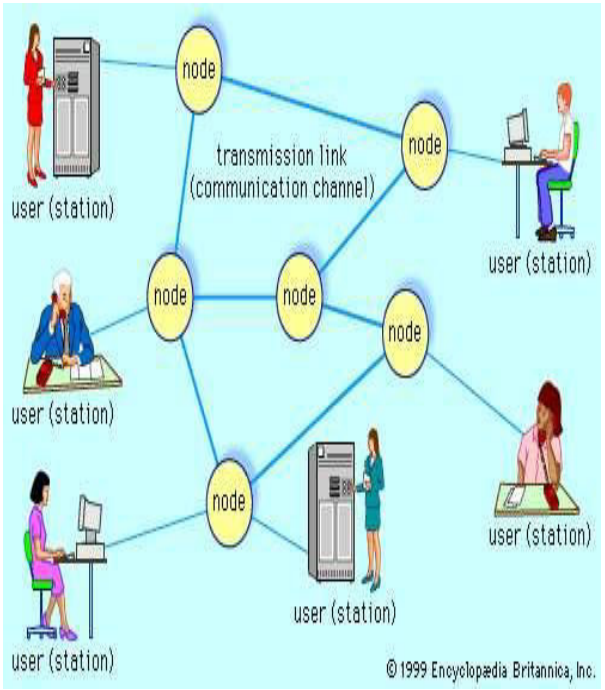
DATA STRUCTURES AND ITS APPLICATIONS

Graph Representation

PES UNIVERSITY ONLINE

Why Graph Representation?





Graph Representation

- How do we store mathematical structure of a graph in the computer memory?

- What types of data structures are used to represent the graph?

- Information Required to represent the graph
  - set of vertices of the graph
  - for each vertex neighbours of the vertex(edge information)

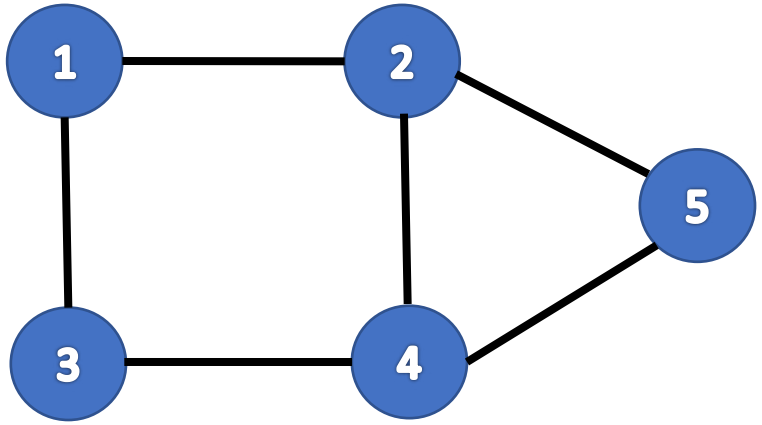
Graph Representation

- Depending on the density of edges, ease of use and types of operation performed , Graphs can be represented by

  - ✓ Adjacency Matrix
    - Two Dimensional Array
  - ✓ Adjacency List
    - Linked List

Adjacency Matrix Representation – Undirected graph

- Adjacency matrix = n x n matrix M, graph of n vertices/nodes
- **M[i][j] = 1** if (i, j) is an edge
- **M[i][j] = 0 ,** no edge between the pair of vertices i and j



Undirected Graph

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

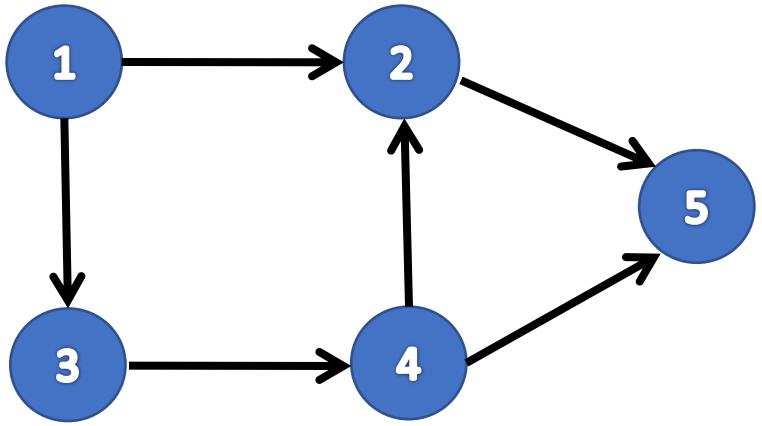
Main diagonal

Note:

- For undirected graph, M is symmetric i.e M[i][j] = M[j][i]
- Assume no edge from node to itself. So diagonal elements has value 0

Adjacency Matrix Representation – Directed graph

- In directed graph edge is directed
- In directed graph edge(i,j) is not equal to edge(j,i)
- Adjacency matrix is asymmetric



Directed Graph

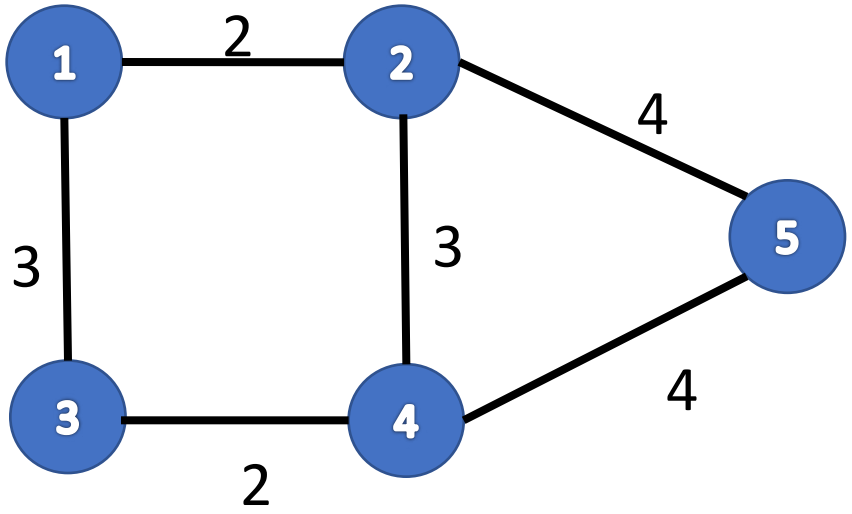
	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	0	1	0	0	1
5	0	0	0	0	0

Main diagonal

DATA STRUCTURES AND ITS APPLICATIONS

Adjacency Matrix Representation – weighted graph

- In the weighted graph distance or cost between the nodes are represented on the edge
- cost/distance value specified on the edge between adjacent nodes are stored in the adjacency matrix



Weighted Graph

	1	2	3	4	5
1	0	2	3	0	0
2	2	0	0	3	4
3	3	0	0	2	0
4	0	3	2	0	4
5	0	4	0	4	0

Drawbacks of Adjacency Matrix Representation

- Number of nodes in the graph needs to be known in prior

- In case graph needs to be updated dynamically as the program proceeds new matrix must be created for each addition or deletion

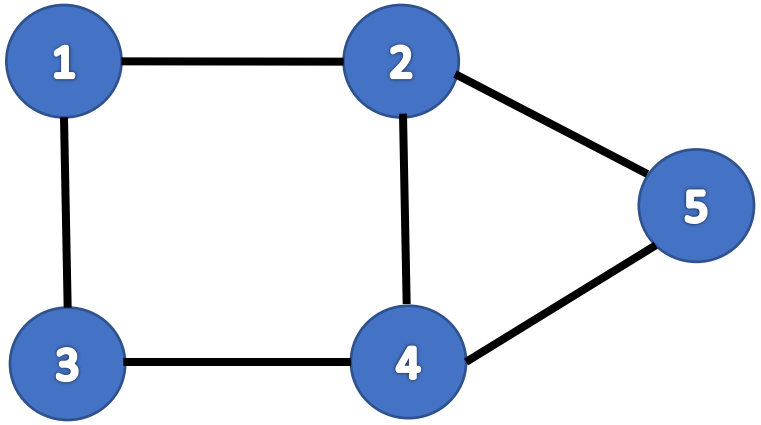
- To detect presence of edge between pair of nodes takes constant time O(1) but it takes $O(v^2)$ to visit all the neighbouring nodes of each node.

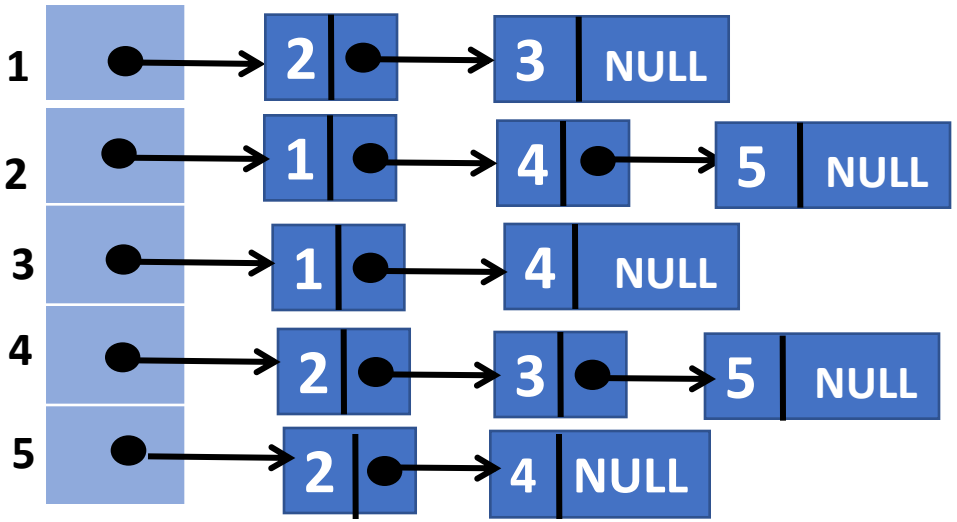
- Adjacency matrix becomes sparse in case graph has very few edges.

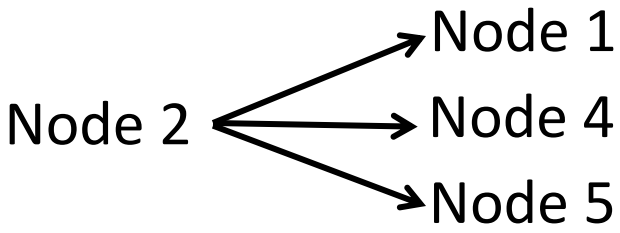
- space complexity is $O(v^2)$. v^2 locations are required for graph with v nodes

Adjacency list representation – Undirected graph

- Each node maintains
  - linked list of its neighbours

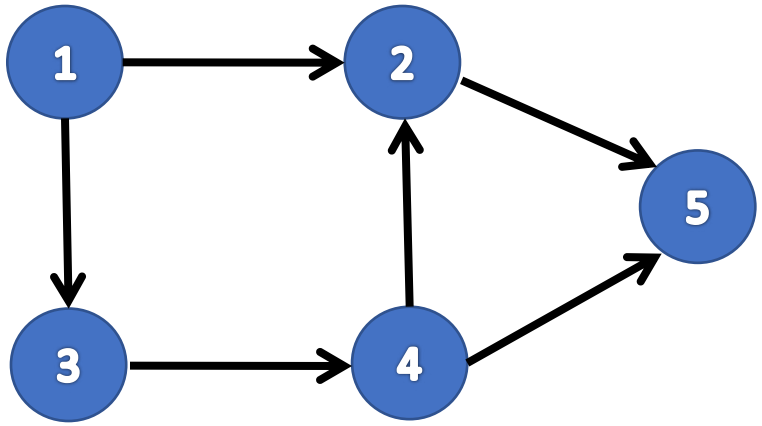


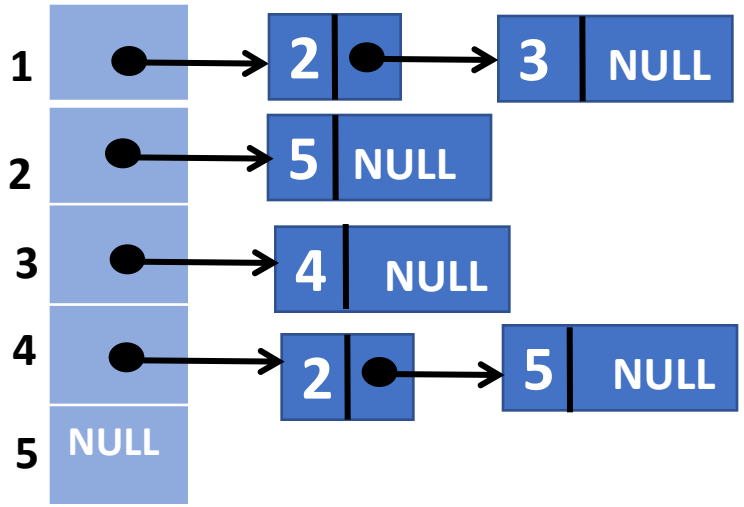




DATA STRUCTURES AND ITS APPLICATIONS

Adjacency list representation – Directed graph



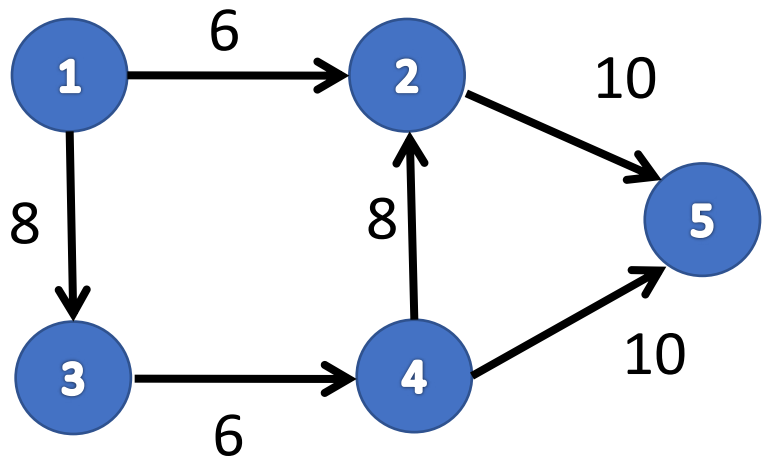


Directed Graph

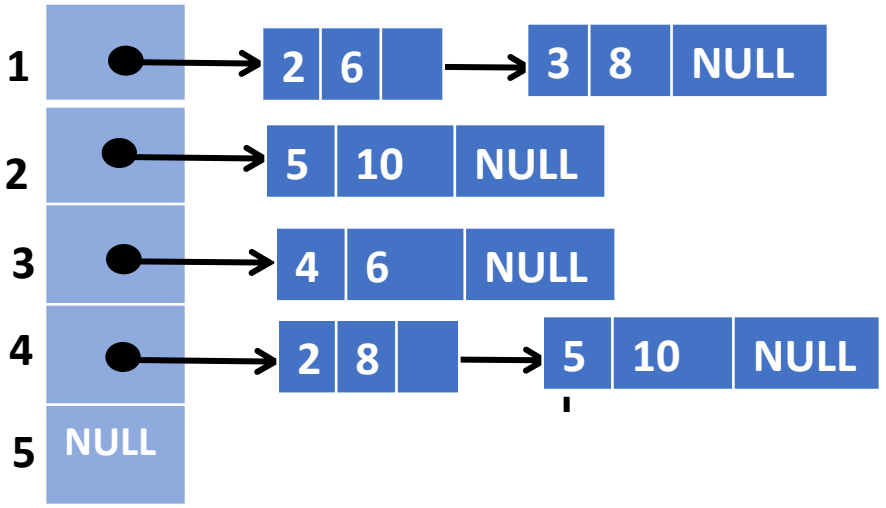
Node 2 ⟶ Node 5

Adjacency matrix representation – weighted graph

- In weighted graph the distance or cost between the nodes are represented on the edge
- Along with the information of adjacent node , cost/distance value is stored in each node of the linked list.



Weighted Graph



Pros and Cons of Adjacency List Representation

- Space Complexity of Adjacency list is O(V+E), because it stores the information of edges that actually exists in the graph
- In case of low density edges, the adjacency matrix becomes sparse using adjacency list is better for representation
- To detect the presence of edge between two nodes, we need to traverse the linked list of the node.

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of graph using adjacency matrix

Data structure to represent the adjacency matrix:

- To represent only the existence of edge between nodes

```
#define maxnodes 50
adj[maxnodes][maxnodes];
```

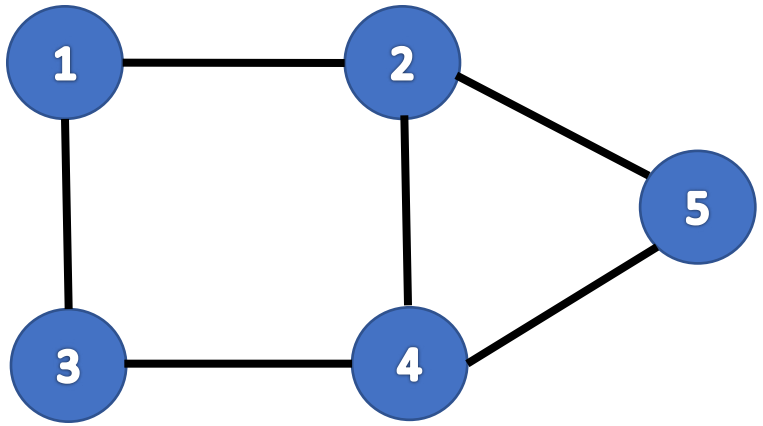
```
Typedef Boolean AdjecencyMatrix[maxnodes][maxnodes]

Typedef struct graph
{
    int n;     /* number of vertices in the graph */
    AdjacencyMatrix adj;
}Graph;
```

Implementation of Graph using Adjacency Matrix

To check the presence of edge between pair of nodes:

```
if ( adj[i][j] == 1)   */ i and j represents the vertex in a graph */
        then "edge exists between the pair of nodes"
else
          " there is  no edge between the pair of nodes"
```



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of graph using adjacency matrix:

To add an edge from node1 to node2:

```
void join(int adj[][MAXNODES],int node1,int node2)
{
        adj[node1][node2]=TRUE;
}
```

To delete an edge from node1 to node2 if it exists:

```
void remv(int adj[][MAXNODES],int node1,int node2)
{
        adj[node1][node2]=FALSE;
}
```

Implementation of adjacency matrix:

To check whether arc exists between node1 and node2:

```
int adjacent(int adj[][MAX],int node1,int node2)
{
        return((if(adj[node1][node2])==TRUE)?TRUE:FALSE);
}
```

C-Representation of graphs- Adjacency Matrix

```
#define MAXNODES 50
struct node
{
        //information associated with each node
};
struct arc
{
        int adj;// information associated with each edge
};
struct graph
{
        struct node nodes[MAXNODES];
        struct arc arcs[MAXNODES][MAXNODES];
}
struct graph g;
```

DATA STRUCTURES AND ITS APPLICATIONS

C representation of weighted graph

- Weighted graph with fixed number of nodes is declared as

```
struct arc{
      int adj;
      int weight;
  };
  struct arc g[maxnodes][maxnodes];
```

To add an edge from node1 to node2:

```
void  joinwt(struct arc g[][maxnodes],int node1,int node2, int wt)
{
  g[node1][node2].adj = TRUE;
  g[node1][node2].weight= wt;
}
```

DATA STRUCTURES AND ITS APPLICATIONS

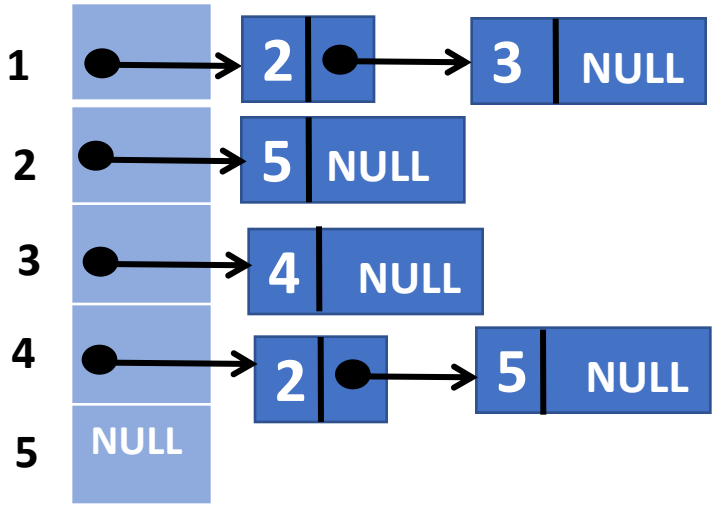
Implementation of adjacency List Representation:

```
struct node
{
   int node;
   struct node *nextedge;
};
```

NODE info	Nextedge

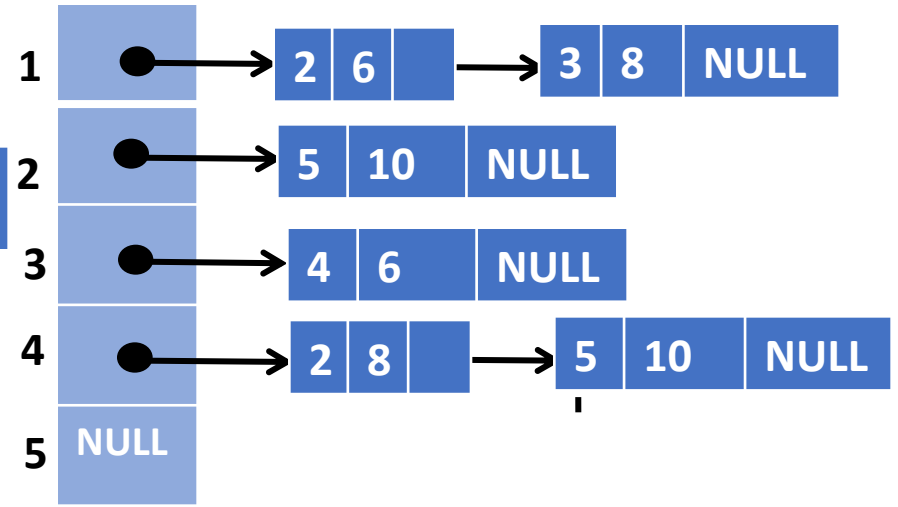
To create a node list:

```
 struct node  *nodelist;
 for(i=0; i < no_of_nodes; i++)
    nodelist[i]= NULL;
```



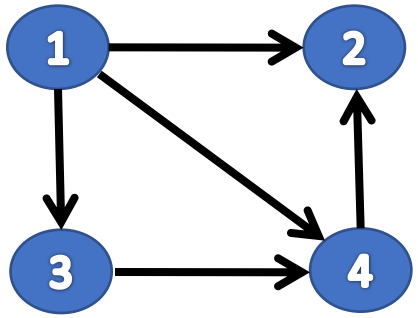

```
struct node
{
   int node;
   int weight;
   struct node *nextedge;
};
```

nodeinfo	weight	nextedge



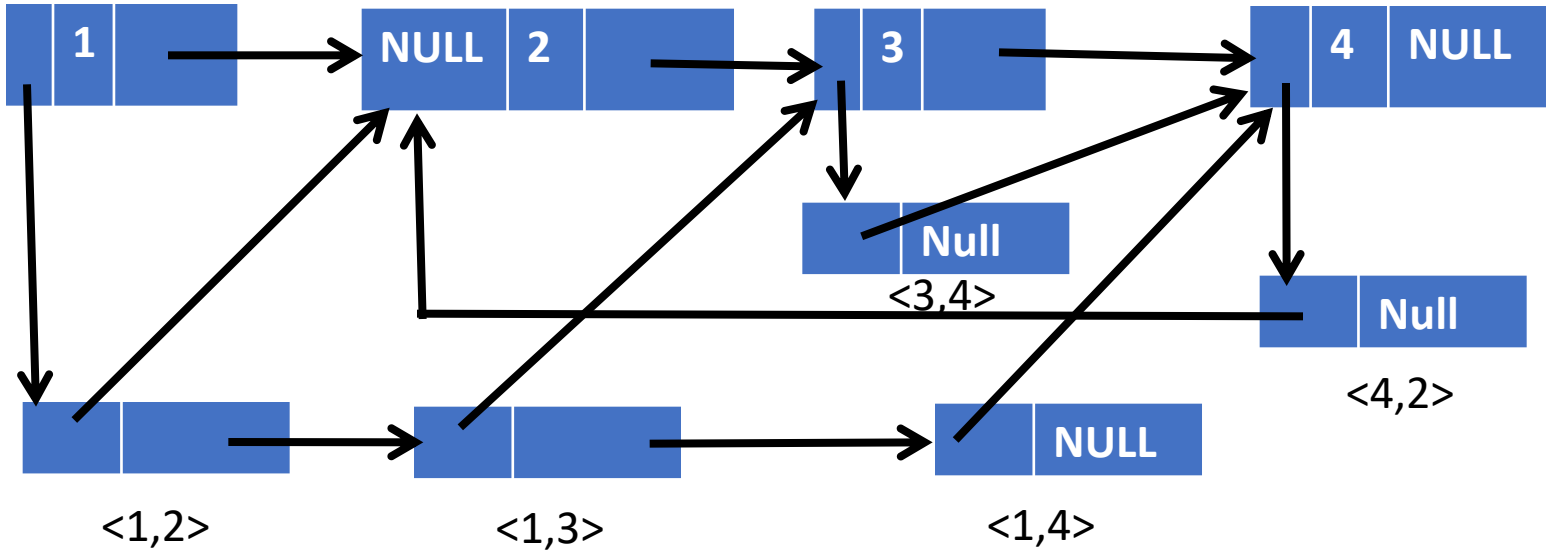
DATA STRUCTURES AND ITS APPLICATIONS

Multilinked structure representation



Sample header node representing the graph node

A sample list node representing edge



DATA STRUCTURES AND ITS APPLICATIONS

Multilinked structure representation

Dynamic Implementation:

```
struct nodetype
{
  int info;
  struct nodetype  *ptr;
  struct node type *next;
};
struct nodetype *nodeptr;
```

Sample header node representing the graph node

A sample list node representing edge

- Note that header node and list node have different formats and must be represented by different structures.
- In case of weighted graph in which each list node contains info field to store the weight of the edge the same dynamic implementation could be used for both types of nodes.

THANK YOU

Sandesh B. J

Department of Computer Science & Engineering

sandesh_bj@pes.edu

+91 80 6618 6623

DATA STRUCTURES AND ITS APPLICATIONS

Graph Traversal Methods

Sandesh B. J
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Graph Traversals Methods

Sandesh B. J
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Graph Traversal Methods - Topics

- Why we need to traverse the graph?
- Traversing the graph using different techniques
  - ✓ Depth First Search and Breadth First Search Traversal
- Algorithms for Depth First and Breadth First Search Traversal

Graph Traversals

- **Graph traversal**(also known as graph search) refers to the process of visiting/investigating (checking or updating) each vertex of the graph in some systematic order -Wiki

- Traversal can start in any arbitrary vertex

- Traversals are classified by the order in which the vertices are visited

Methods to Traverse the Graph

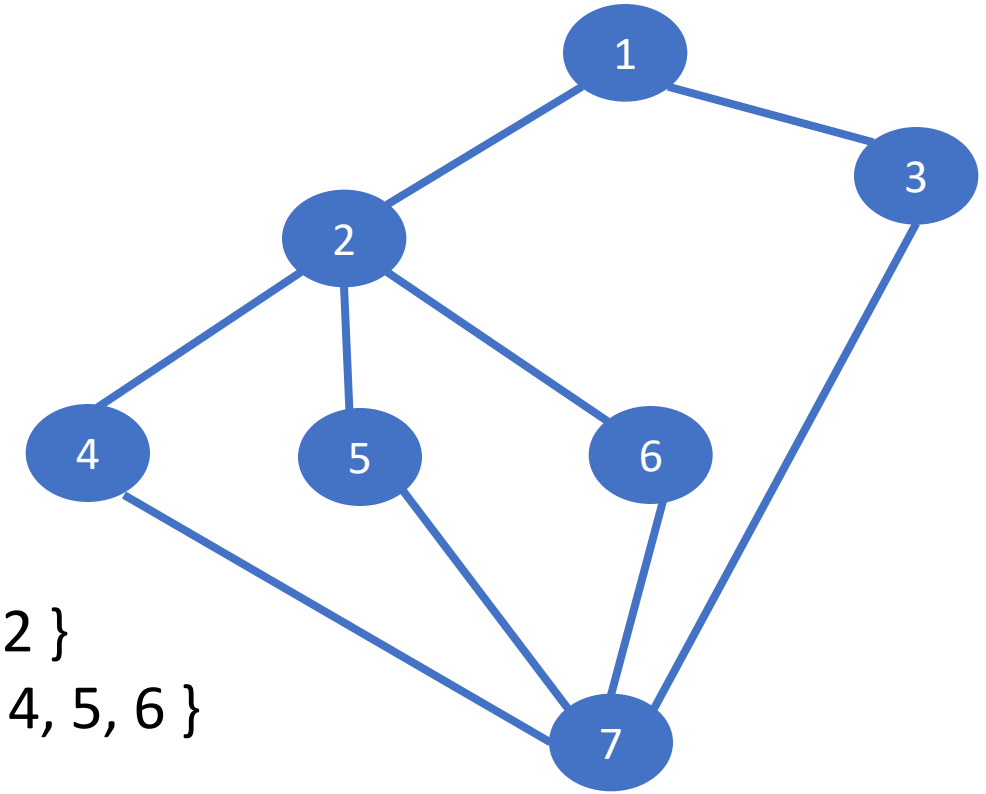
- Depth First Search
- Breadth First Search

Depth first search (DFS)

- Visits all the nodes related to one neighbour before visiting the other neighbours and its related nodes. Once it reaches dead end, it backtracks along previously visited node and continue traversing from there.
- Analogues to pre-order traversal of an ordered tree
- Uses stack behaviour, hence implemented using recursive algorithm

Depth First Search Traversal

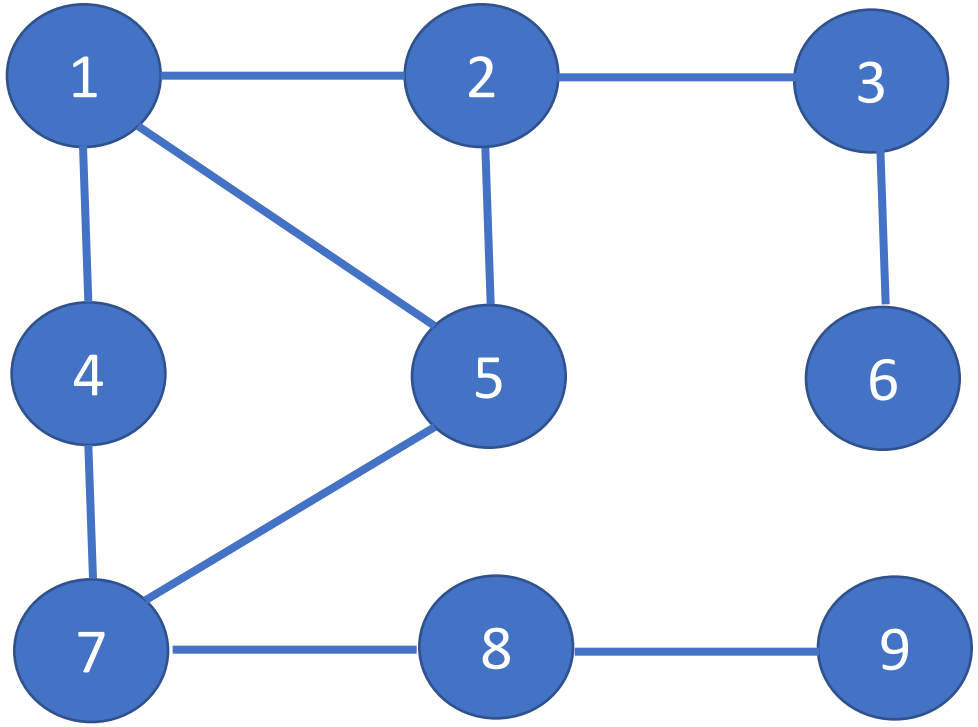
- DFS traverse the depth of the graph, until it cannot go any further at which point it backtracks and continues



V = { 2 }
W= { 4, 5, 6 }

Traversal order : 1 , 2, 4 , 7, 3, 5, 6

Depth First Search Traversal

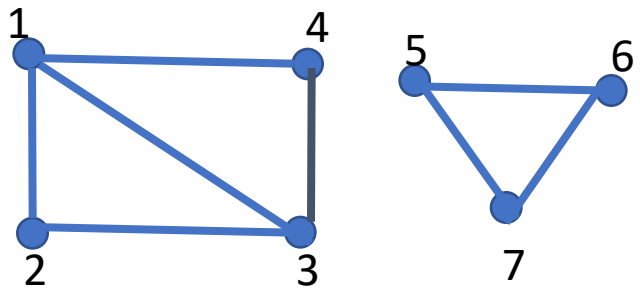


1 2 3 6 5 7 4 8 9

Difficulties in the graph traversals

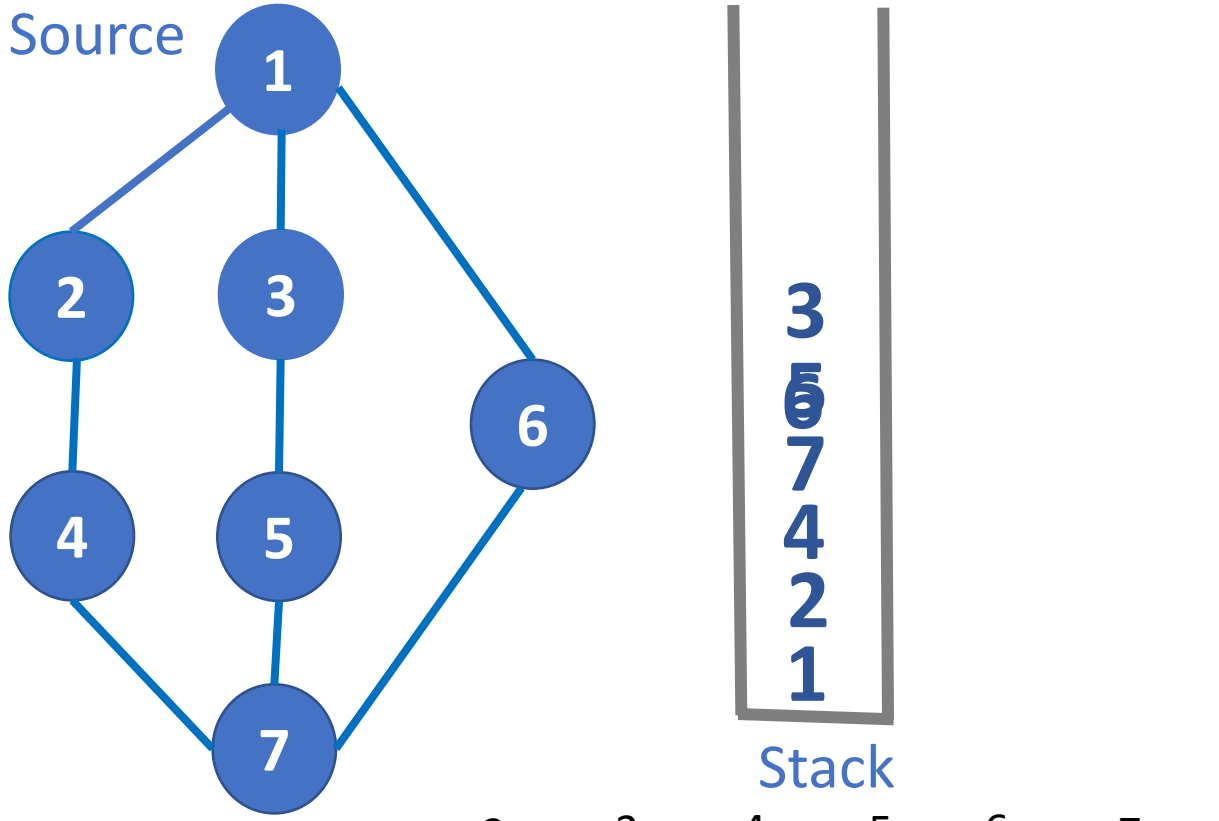
- **Graph may contain cycles**
  - Traversal algorithm may reach the same vertex second time.
  - To prevent the infinite recursion , we introduce Boolean valued array **visited**
  - Set the **visited[v] to true** once node v is visited
  - Check the **visited[w]** before processing any node w

- **Graph may not be connected:**
  - Traversal algorithm may fail to reach all the nodes from a single starting point



DATA STRUCTURES AND ITS APPLICATIONS

DFS Traversal – Using Stack



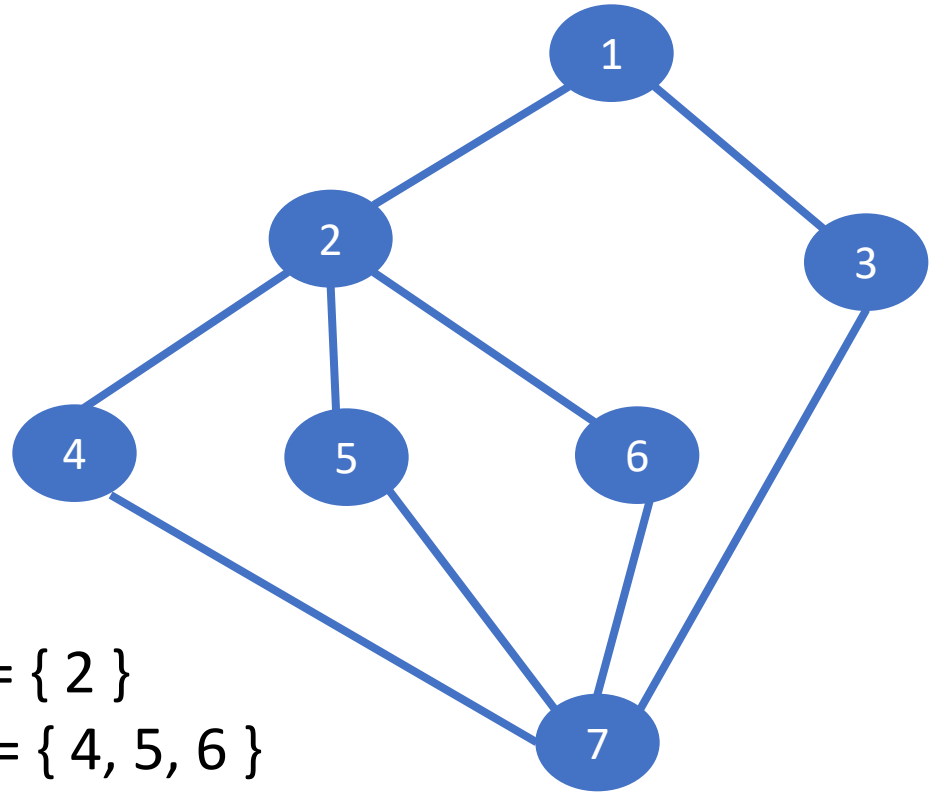
Breadth first search (BFS)

- Explores all the neighbour nodes in first level from an arbitrary node, before moving to next level of neighbouring nodes.

- Uses Queue behaviour

Breadth First Search Traversal

- Analogous to level-by-level traversal of ordered tree

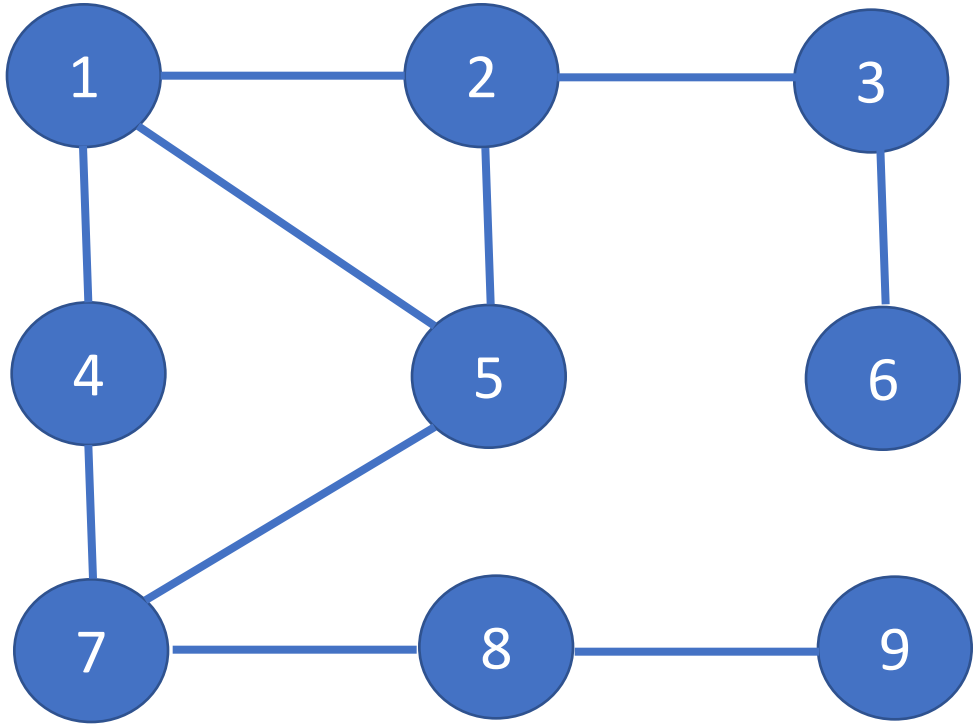


V = { 2 }
W= { 4, 5, 6 }

Traversal order : 1 , 2, 3, 4, 5, 6 , 7

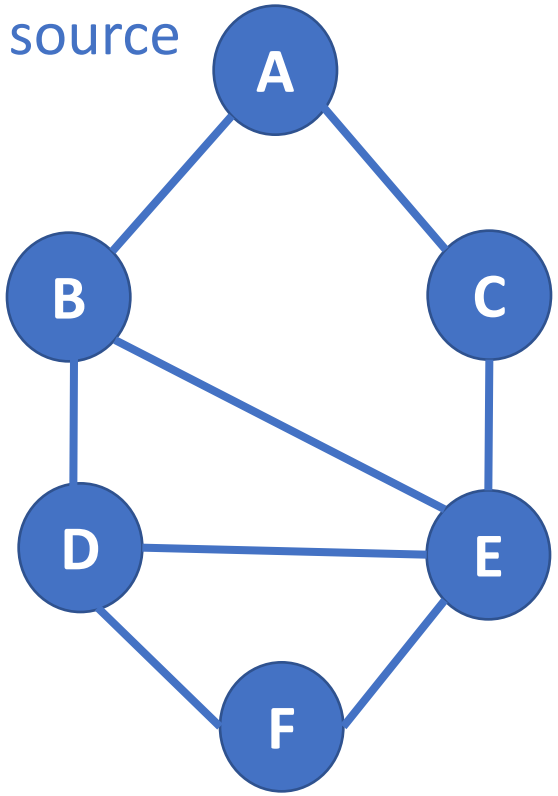
DATA STRUCTURES AND ITS APPLICATIONS

Breadth first search Traversal



1 2 4 5 3 7 6 8 9

BFS Traversal – Using Queue



front

Queue

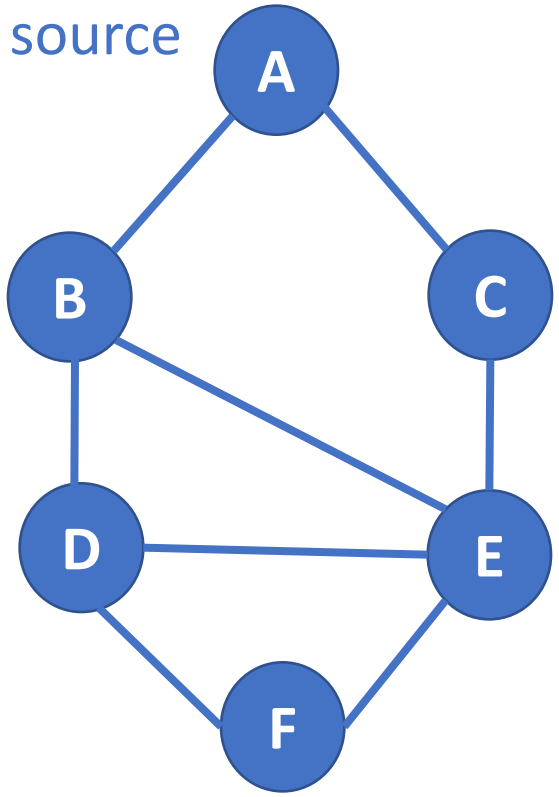
Remove_Q

Visited

A B C D E F

Visited nodes A B C D E

BFS – Traversal – Using Queue

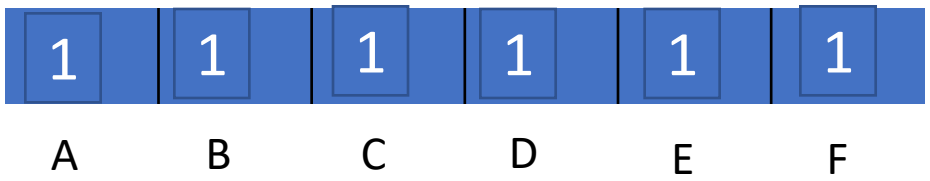


front

Queue

Remove_Q

Visited



Visited nodes A B C D E F

DFS - Algorithm

```
Algorithm DFS (Graph G )
//Implements DFS traversal for given graph
// Input Graph G = (V, E)
//Output Graph G with vertices marked as visited
 mark each vertex in V with 0  as a mark of being “unvisited”
     for each vertex v in V do
          if v is marked with o
               dfs(v)
```

```
dfs(v)
// visits recursively all the unvisited  vertices connected to
    vertex v by a path
For each vertex w in V adjacent to v do
    if w is marked with o
       mark w as visited
       dfs(w)
```

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

BFS - Algorithm

```
Algorithm BFS (Graph G )
//Implements BFS traversal for given graph
// Input Graph G = (V, E)
//Output Graph G with vertices marked as visited
 mark each vertex in V with 0  as a mark of being “unvisited”
     for each vertex v in V do
          if v is marked with 0
               bfs(v)
```

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

BFS - Algorithm

```
bfs(v)
// visits recursively all the unvisited  vertices connected to
    vertex v by a path
While the queue is not empty
   for each vertex w in V adjacent to front vertex do
      if w is marked with 0
         mark w as visited
         add w to queue
         remove the front vertex from the queue
```

Courtesy: “Introduction to Design and Analysis of Algorithms” By Anany Levitin

THANK YOU

Sandesh B. J

Department of Computer Science & Engineering

sandesh_bj@pes.edu

+91 80 6618 6623

DATA STRUCTURES AND ITS APPLICATIONS

Graphs

Saritha

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Representation of Network Topology

Saritha

Department of Computer Science & Engineering

Applications: Network Topology

Graph data structure is mainly in Computer Networks, Telecommunication, Electronic Circuits and Transport Networks.

Networking uses the Notation G(N,L) instead of G(V,E) for a graph where N is the set of nodes and L is the set of links.

DATA STRUCTURES AND ITS APPLICATIONS

Applications: Network Topology

- Topology is the order in which nodes and edges are arranged in the network.
- How the computers are connected or related to one another in a computer.
- There are 2 types of Topology
    1. Physical
    2. Logical

DATA STRUCTURES AND ITS APPLICATIONS

Applications: Network Topology

- 1.Ring Topology
- 2.Star Topology
- 3.Mesh Topology
- 4.BusTopology

DATA STRUCTURES AND ITS APPLICATIONS

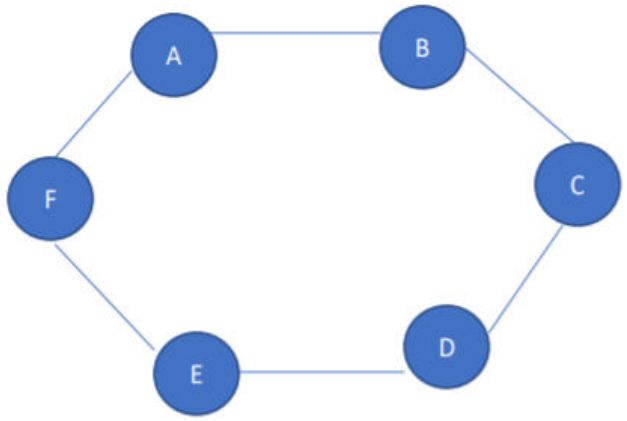
Representation of Graph

1.Adjacency Matrix

2.Adjacency List

Applications: Network Topology

1. Ring topology (cycle): A cycle graph is a simple graph which has two degrees of vertices.

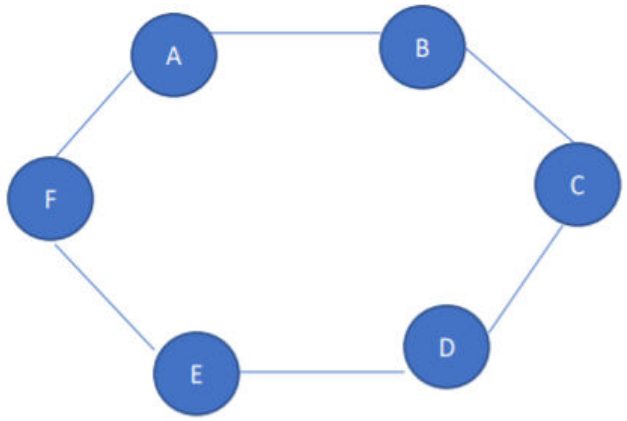


Ring topology

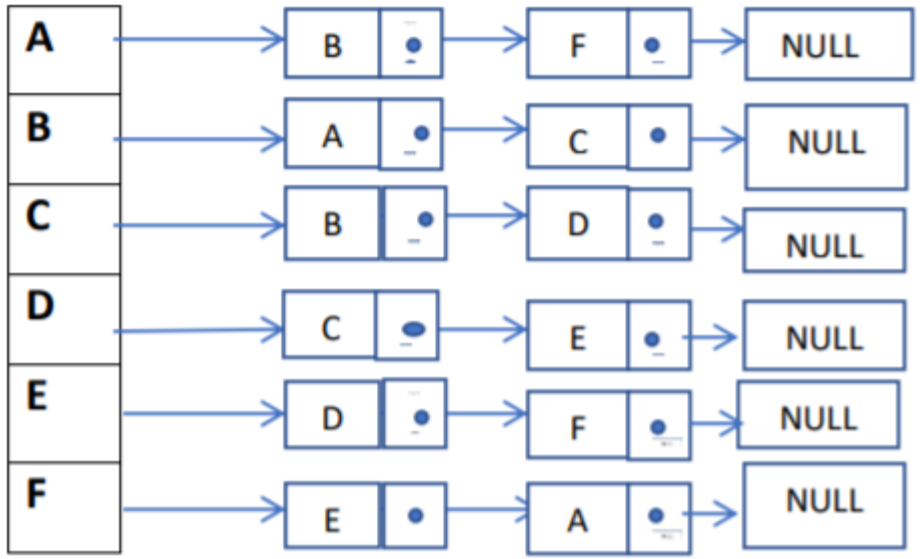
	A	B	C	D	E	F
A	0	1	0	0	0	1
B	1	0	1	0	0	0
C	0	1	0	1	0	0
D	0	0	1	0	1	0
E	0	0	0	1	0	1
F	1	0	0	0	1	0

DATA STRUCTURES AND ITS APPLICATIONS

Applications: Network Topology

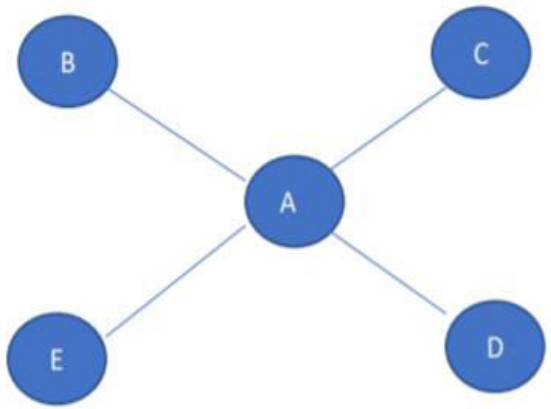


Ring topology



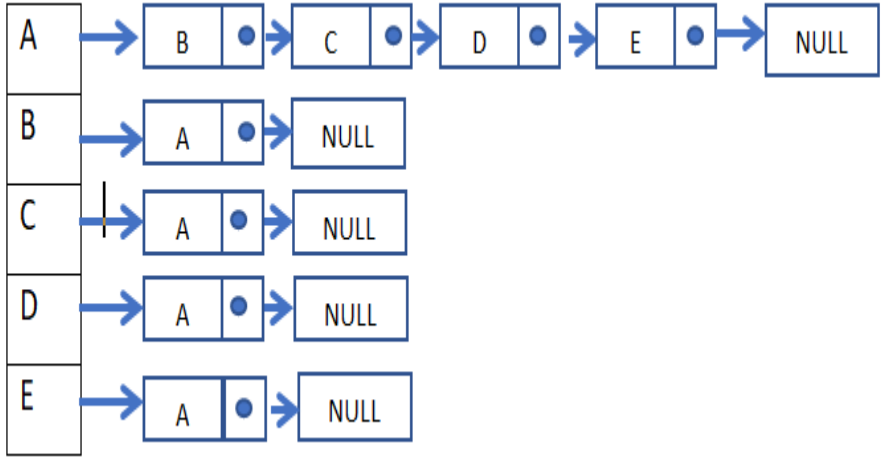
Applications: Network Topology

2.Star topology: Star topology is a network topology in the form of merging from the central vertex to each vertex .



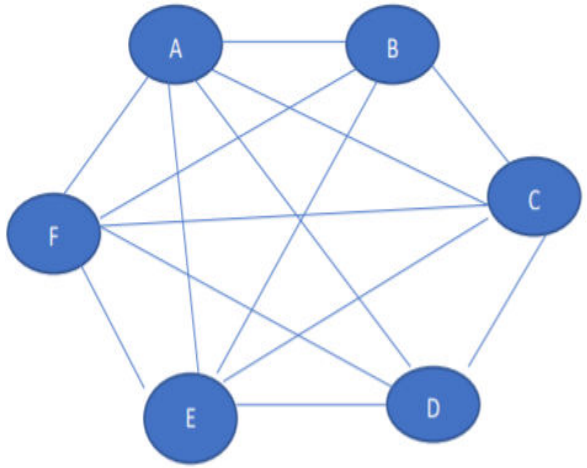
Star topology

	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	0	0
D	1	0	0	0	0
E	1	0	0	0	0



Applications: Network Topology

3.**Mesh topology**: Mesh Topology is a complete graph in which all the vertex is connected to all other vertices

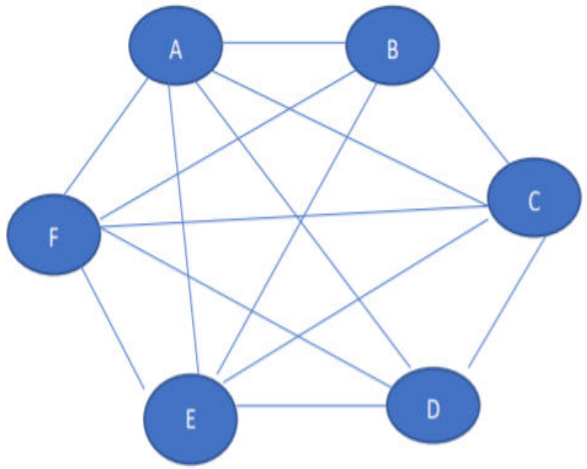


Mesh Topology

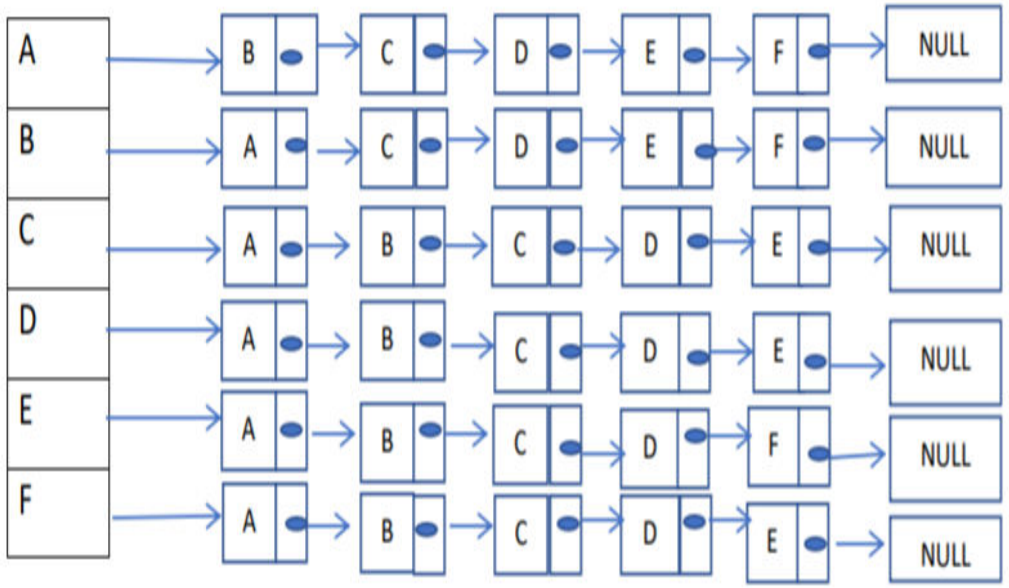
	A	B	C	D	E	F
A	0	1	1	1	1	1
B	1	0	1	1	1	1
C	1	1	0	1	1	1
D	1	1	1	0	1	1
E	1	1	1	1	0	1
F	1	1	1	1	1	0

DATA STRUCTURES AND ITS APPLICATIONS

Applications: Network Topology



Mesh Topology

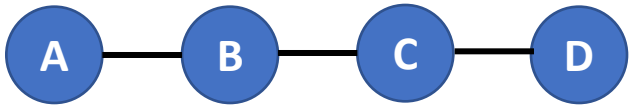


Bus Topology

A Graph G with V vertices is said to represent a bus topology if

1.Every node except the starting and ending node has degree 2 and starting and ending node have degree 1.

2. Number of edges=Number of vertices -1

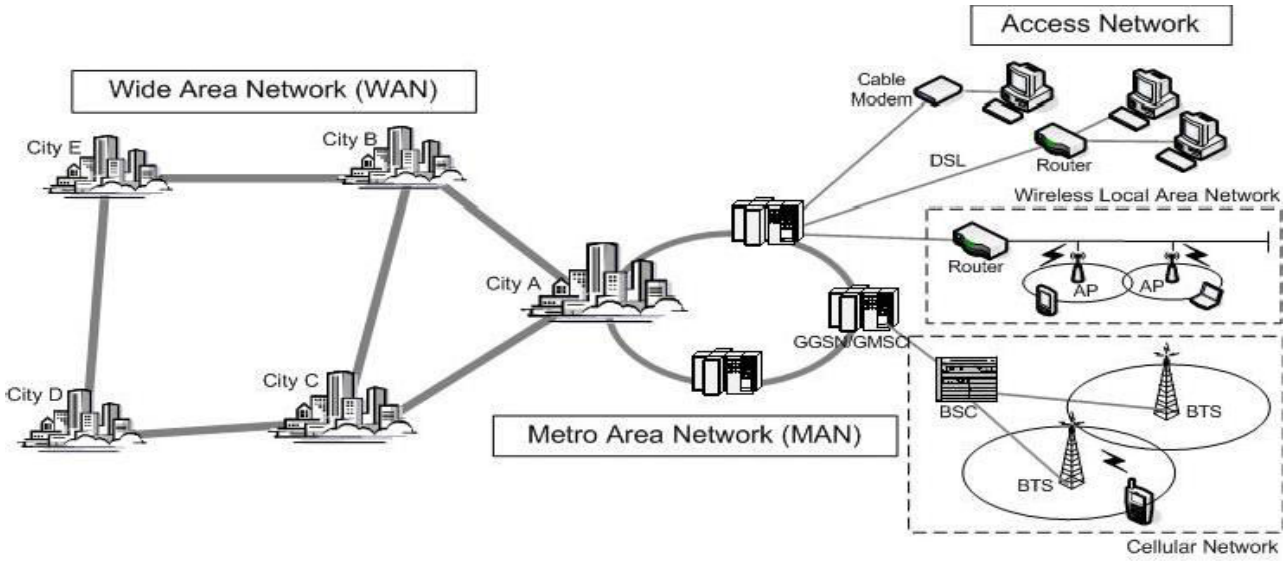


	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	0

DATA STRUCTURES AND ITS APPLICATIONS

Applications: Network Topology

Most networks a mix of rings, mesh – depending on network type, cost/traffic/reliability



Picture Courtesy: ComputerNetworkTopology.com

THANK YOU

Saritha

Department of Computer Science & Engineering

Saritha.k@pes.edu

9844668963

DATA STRUCTURES AND ITS APPLICATIONS

Graphs

Saritha
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Finding all the paths from a given source to destination

Saritha
Department of Computer Science & Engineering

Applications of BFS and DFS

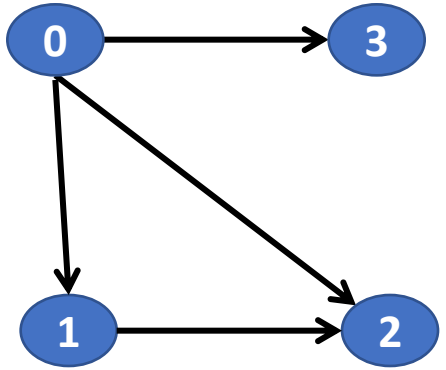
Application of DFS

- Detecting whether a cycle exist in graph.
- Finding a path in a network
- Topological Sorting: Used for job scheduling
- To check whether a graph is strongly connected or not

Applications of BFS and DFS

Path

- Sequence of edges that allows the user to go from vertex A to vertex B



- The paths from vertex 0 to vertex 2:
- 1.0->1->2
- 2.0->2

Finding all the paths from the given source to destination

- **Methodology**
- 1.Start from any traversal Method from a given source node
- 2.Store all the visited vertices in an array
- 3.Once the destination vertex is reached, print all the contents of Array.

Function to print all the paths using DFS traversal method

```
//Function to print all the paths from a given source to destination
void path_find(int source,int destination)
{
    int visited[10]  //An array to store the vertices as visited or not
    int path[10]    //An array to store the path
    int count=0;
    for(int i=0;i<n;i++)
        visited[i]=0   //initilize all the vertices as not visited.
    printallpaths(source,destination,visited,path);
}
```

Function to print all the paths using DFS traversal method

```
void printallpaths(int u,int d,int visited[10],int path[10])
{
   visited[u]=1;//Mark the current node and and store it in the array path
   path[count]=u;
   count++;
   if(u==d) //if the current vertex is same as the destination then print the array
which has stored the path
    {
      for(i=0;i<count;i++)
            printf("%d",path[i]);
     }
   else // if the current vertex is not the destination
   {
     for(NODE temp=a[u];temp!=NULL;temp=temp->link)
```

Function to print all the paths using DFS traversal method

```
if(!visited[temp->data])
{
   printallpaths(temp->data,d,visited,path);
}
}
count--  //Remove the current vertex from the path[] and mark it as
unvisited
Visited[u]=0;
}
```

DATA STRUCTURES AND ITS APPLICATIONS

Function to read adjacency list

```
void read_adjacency_list(NODE a[],int n)
{
   int ele,m;
   for(int i=0;i<n;i++)
   {
      printf("enter the number of nodes adjacent to %d:",i);
       scanf("%d",&m);
       if(m==0)
        continue;
       printf("enter the nodes adjacent to %d:",i);
       for(int j=0;j<m;j++)
      {
        scanf("%d",&ele);
       a[i]=insert_rear(ele,a[i]); // insert at rear end
       }   } }
```

Function to insert a node at rear end of the list

Function to insert a node at rear end

```
NODE insert_rear(int v,NODE head)
 {
    NODE temp;
    NODE cur;
    temp=getnode(); // create a node
    temp->info=v;
     temp->link=NULL;
    if(head==NULL)
          return temp;
     cur=head;
   while(cur->link!=NULL)
   {
      cur=cur->link;
    }
   cur->link=temp;
 return(head); }
```

Connectivity of the graph using Adjacency list

Function to create a node

```
NODE getnode()
 {
  NODE temp;
  temp=(NODE)malloc(sizeof(struct node)); //Dynamic allocation
  if(temp==NULL)
  {
     printf("out of memory");
      return;
   }
  return temp;
 }
```

THANK YOU

Saritha

Department of Computer Science & Engineering

Saritha.k@pes.edu

9844668963

DATA STRUCTURES AND ITS APPLICATIONS

Graphs

Saritha
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

B-trees

Saritha

Department of Computer Science & Engineering

Case study: B-tree

Indexing:

- Indexing is a technique used in data structure to access the records quickly from the database file.
- An Index is a small table containing two columns one for primary key and the second column.

B-tree

- A B-tree is a tree data structure that keeps data in its node in sorted order and performs the operations like searching, insertions, and deletions in logarithmic amortized time.
- B-tree is an m-way search tree in which each node, with the possible exception of the root, is at-least half full.

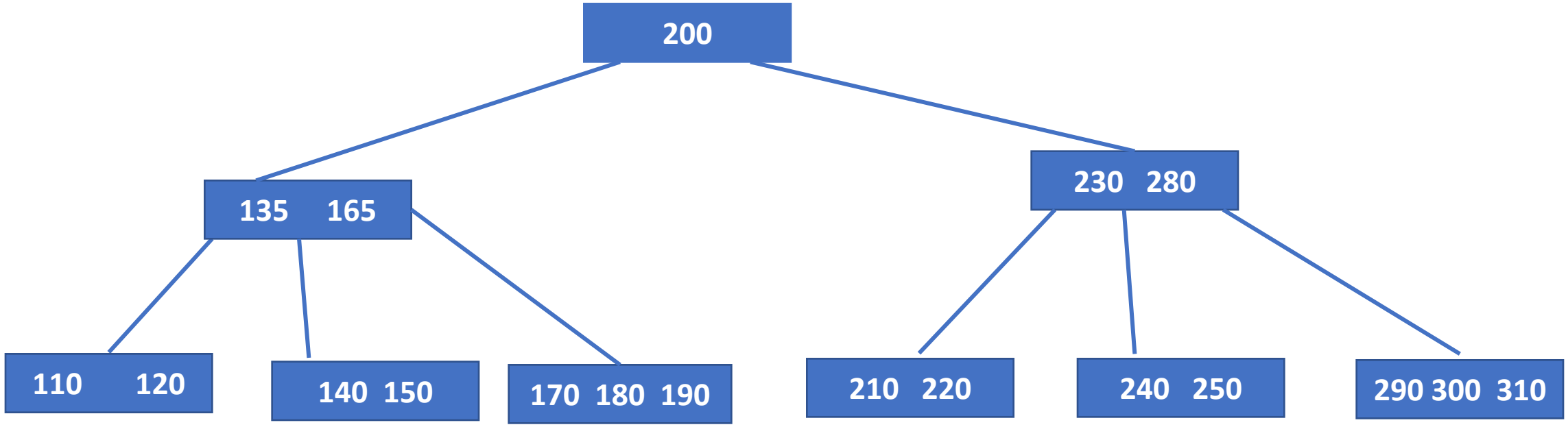
Properties of B-tree

- All the leaves created are at same level.
- B-Tree is determined by a number of degree m. The value of m depends upon the block size on the disk .
- The left subtree of the node has lesser values than the right side of the subtree.
- The maximum number of child nodes(a root node as well as its child nodes can contain) is calculated by :m – 1
- Every node except the root node must contain minimum keys of[m/2]-1
- The maximum number of child nodes a node can have is equal to its degree that is m.
- The minimum children a node can have is half of the order that is m/2

DATA STRUCTURES AND ITS APPLICATIONS

Case study: B-tree

Following is an example of B-Tree of minimum order 5



Case study: B-tree

Why use B-Tree

- B-tree reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.
- B-tree is efficient for reading and writing from large blocks of data

Search

Let Key be the element to be searched .
Start from the root and recursively traverse down.
If the value of key is lesser than the root value,
then search left subtree,
if the value of key is greater than the root value,
then search the right subtree.
If the node has the found key,
then return the node.
If the key is not found in the node,
then traverse down to the child with a greater key.
If key is not found in the tree,
then return NULL.

Insertion

- B-tree insertion takes place at leaf node.
- Locate the leaf node for the data being inserted.
- If the node is not full that is fewer than m-1 entries, the new data is simply inserted in the sequence of node.
- When the leaf node is full, we say overflow condition.
- Overflow requires that the leaf node be split into 2 nodes, each containing half of the data.
- To split the node, create a new node and copy the data from the end of the full node to the new node.
- After the data has been split, the new entry is inserted into either the original or the new node depending on its key value.
- Then the median data entry is inserted into parent node.

Insertion

Consider a sequence of integers and minimum degree ‘t’ of the tree is 3. The maximum number of keys the node can hold is 2t-1=5

Initially root=NULL

Insert 1:

Insert 2:

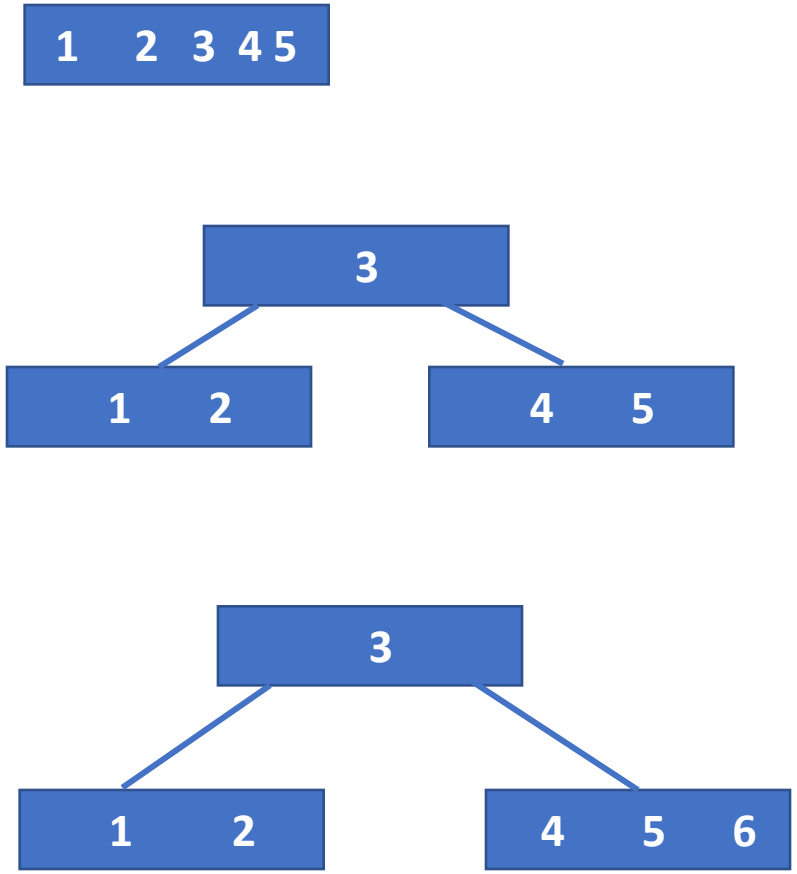
Insert 3:

Insert 4:

Insert 5:

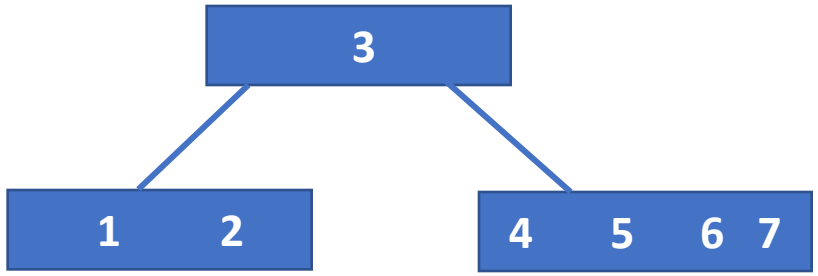
Insertion

Insert 6:

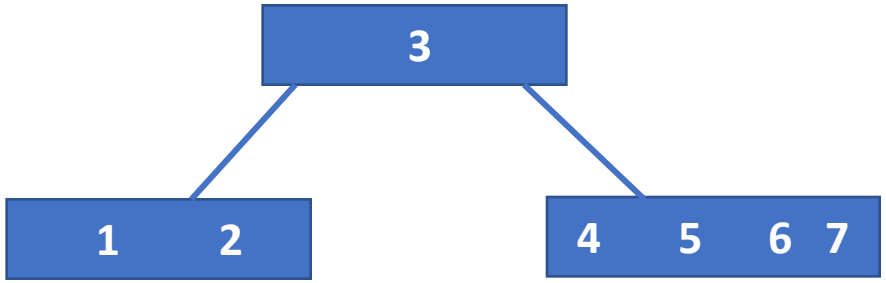


Insertion

Insert 7:

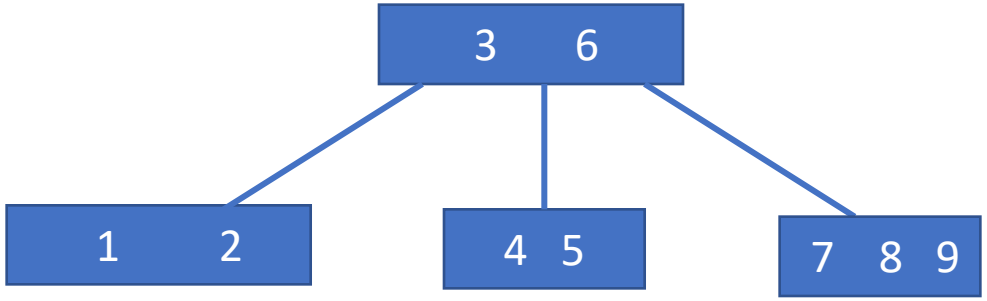


Insert 8:



Insertion

Insert 9:



Insertion Algorithm

1.Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2.If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3.Else, if the leaf node contains m-1 keys, then follow the following steps.
    a)Insert the new element in the increasing order of elements.
    b)Split the node into the two nodes at the median.
    c)Push the median element upto its parent node.
    d)If the parent node also contain m-1 number of keys, then split it too by following the same steps.

B-tree Deletion

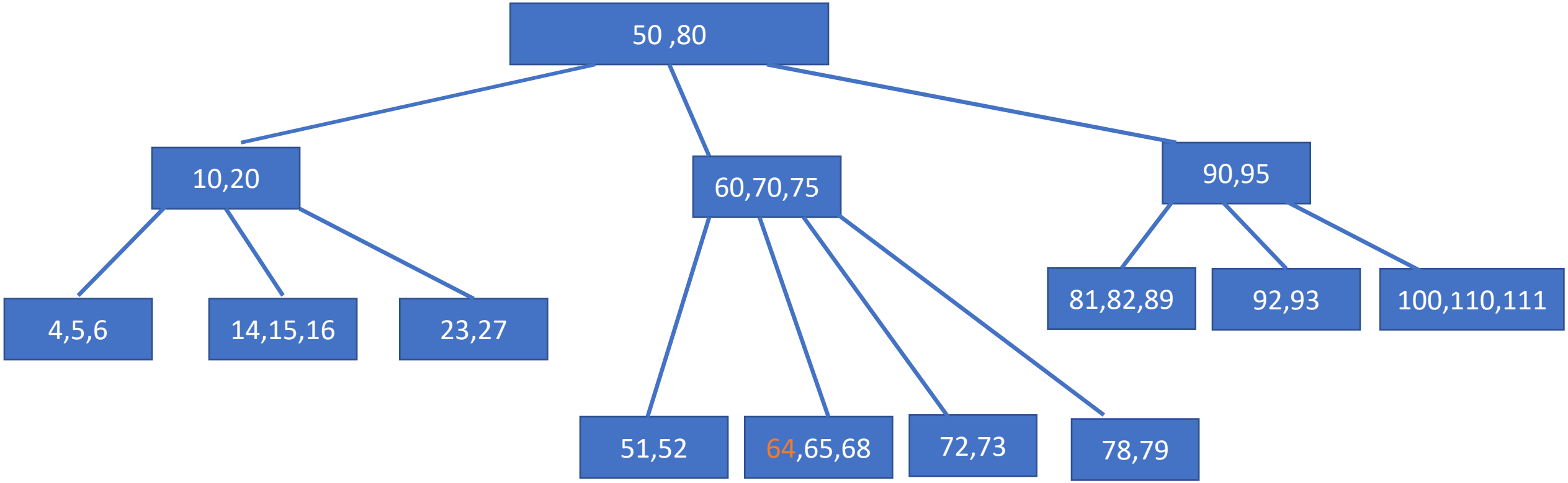
- When deleting a node from B-tree , there are three considerations
- 1. data to be deleted are actually present in the tree.
- 2. if the node does not have enough entries after the deletion, then correct the structural deficiency.
- A deletion that results in a node with fewer than minimum number of entries is an underflow
- 3. Deletion should takes place only from leaf node.
- If the data to be deleted are in internal node, find a data entry to take their place.

DATA STRUCTURES AND ITS APPLICATIONS

Deletion

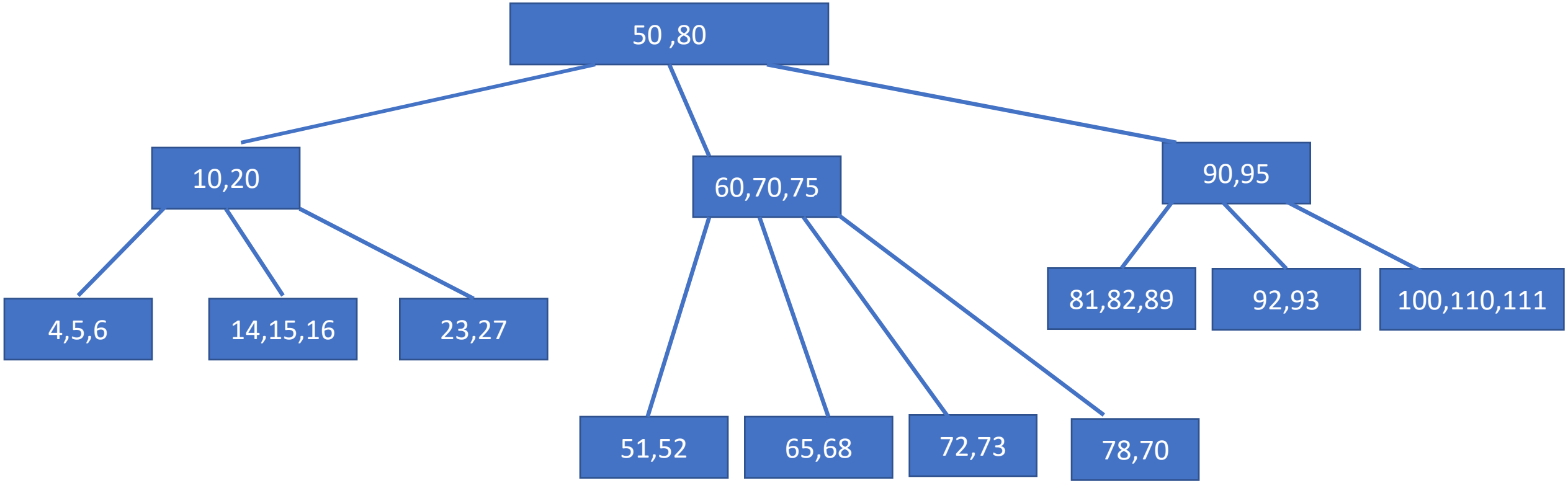
Consider a tree of order 5

Delete 64



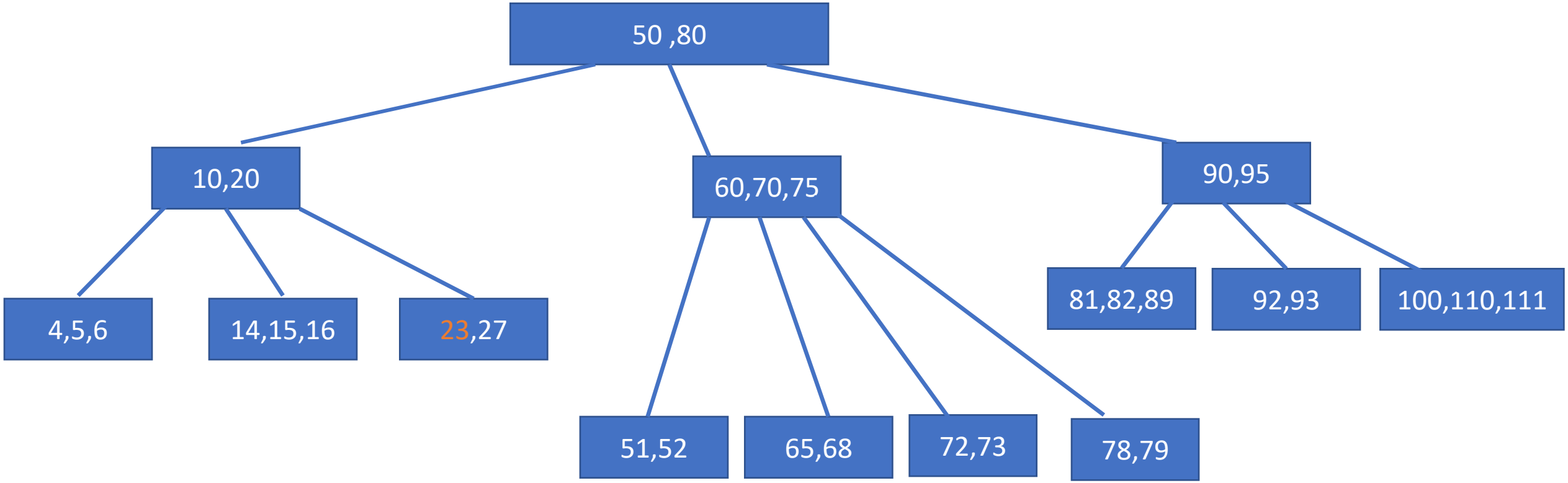
Deletion

After deleting 64



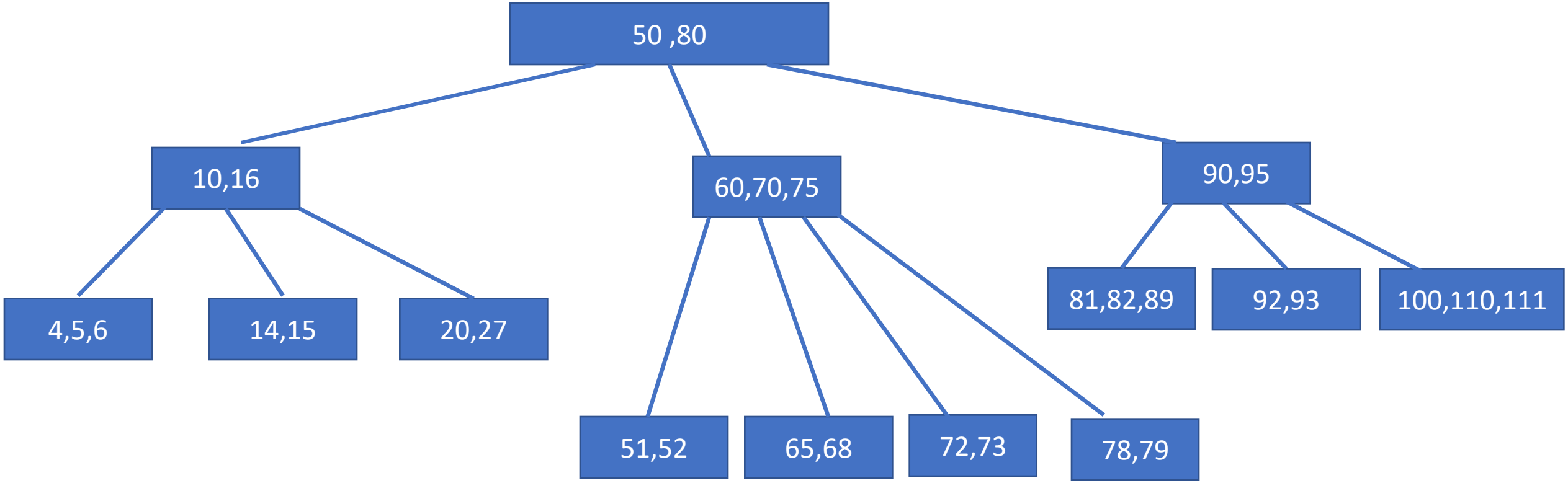
Deletion

Delete 23



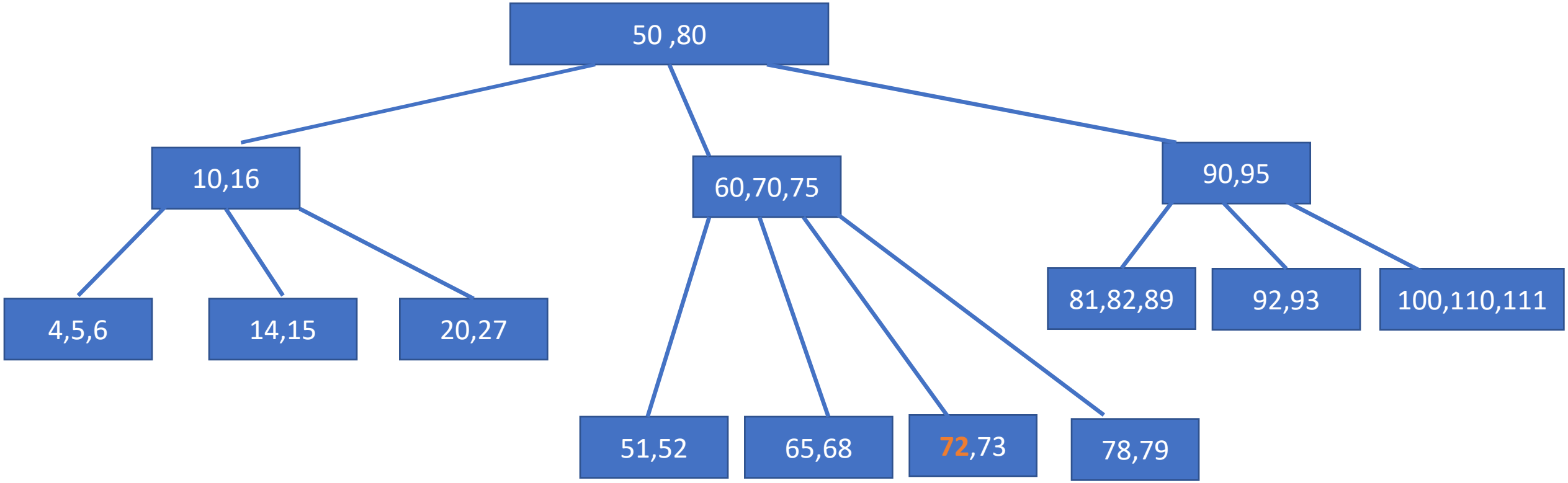
Deletion

After deleting 23



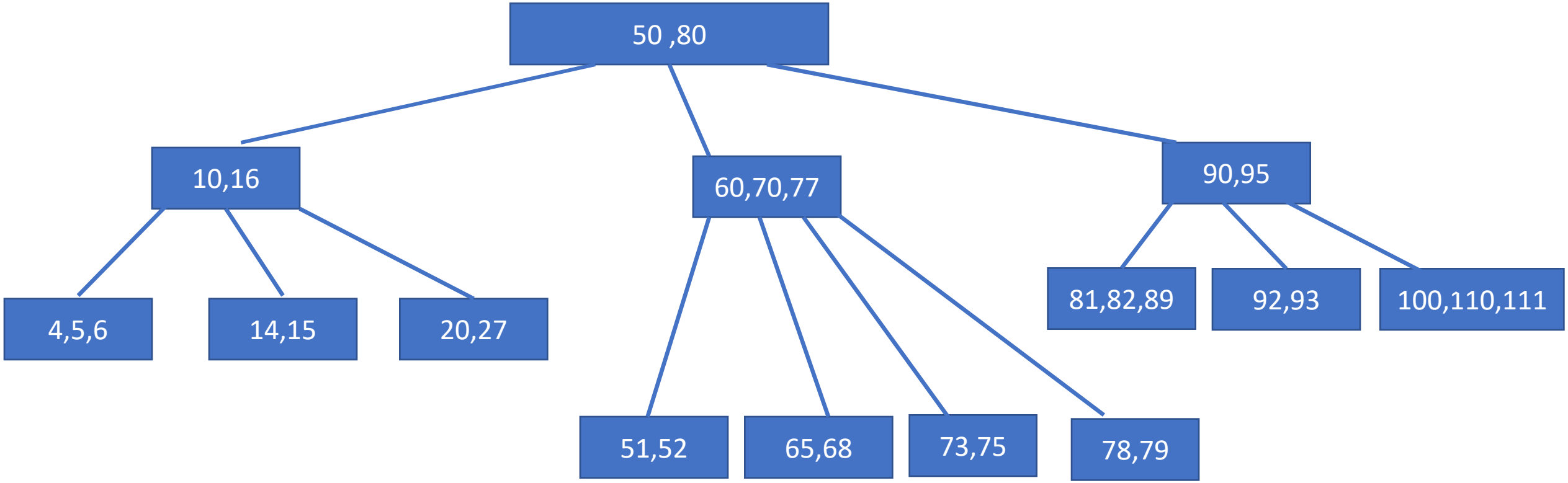
Deletion

Delete 72



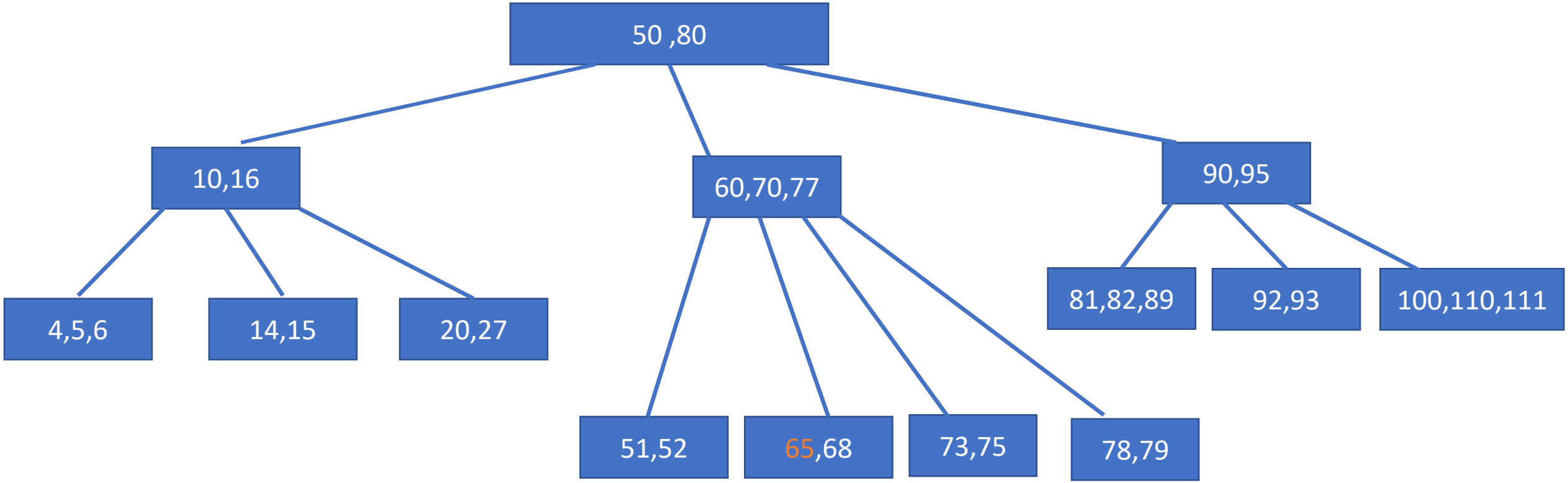
Deletion

After deleting 72



Deletion

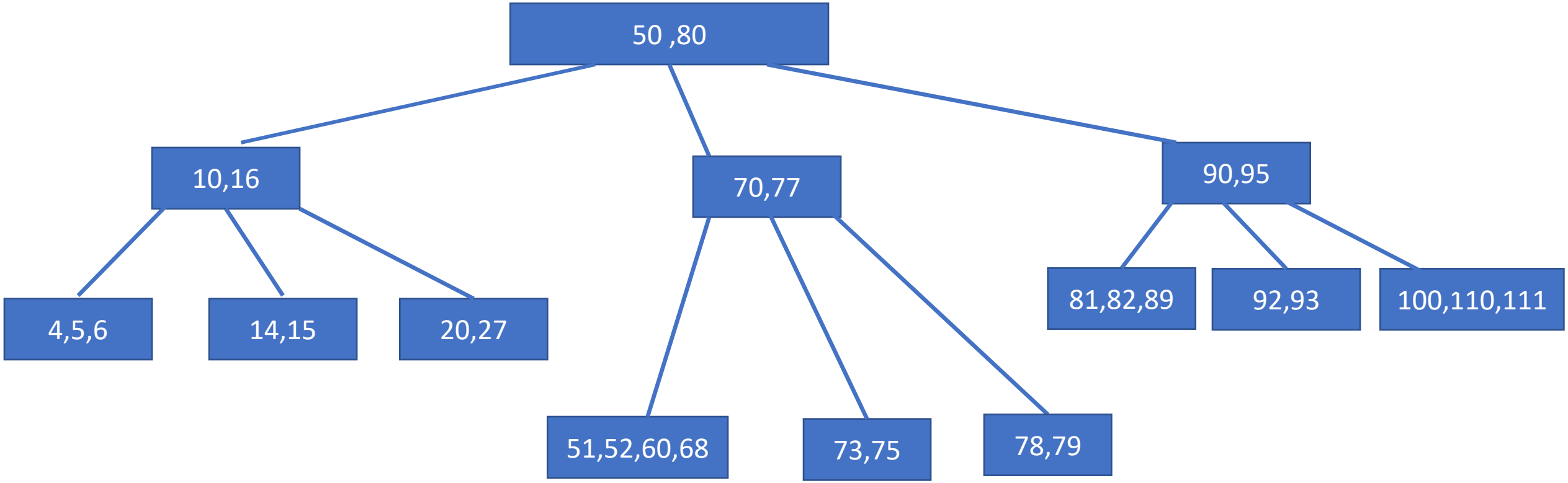
Delete 65



DATA STRUCTURES AND ITS APPLICATIONS

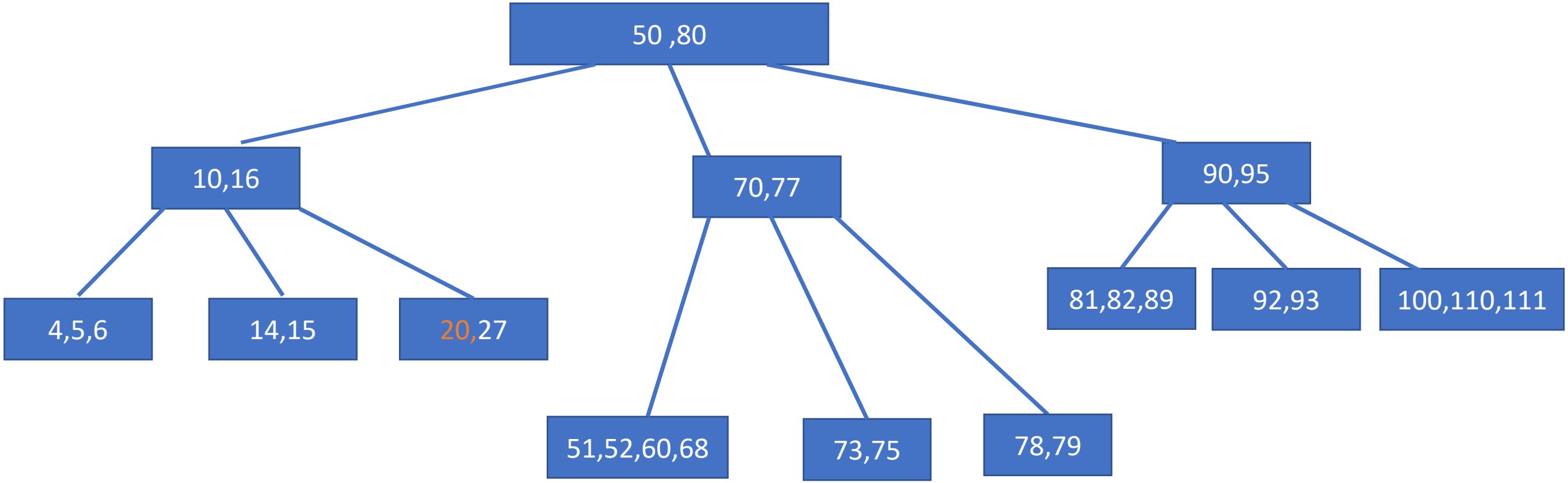
Deletion

After deleting 65



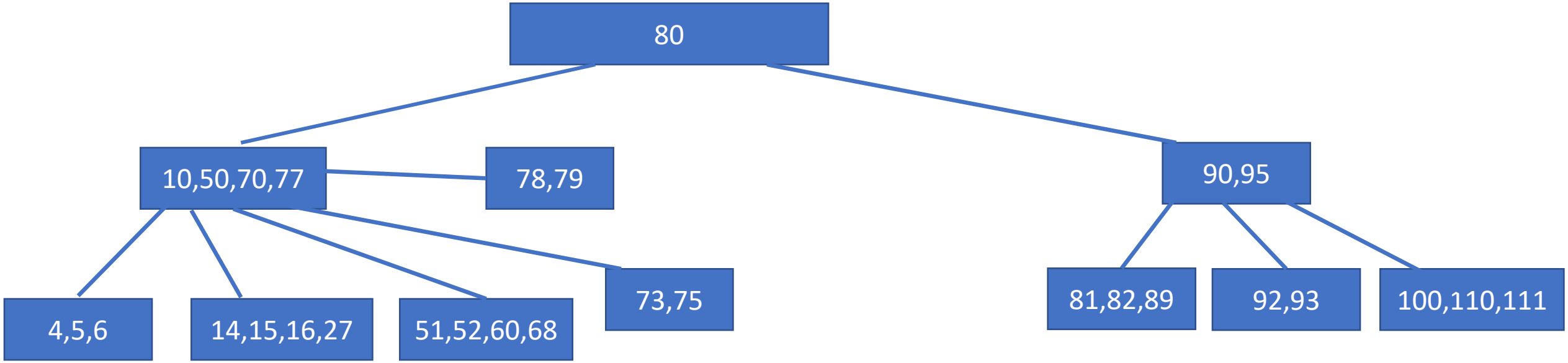
Deletion

Delete 20



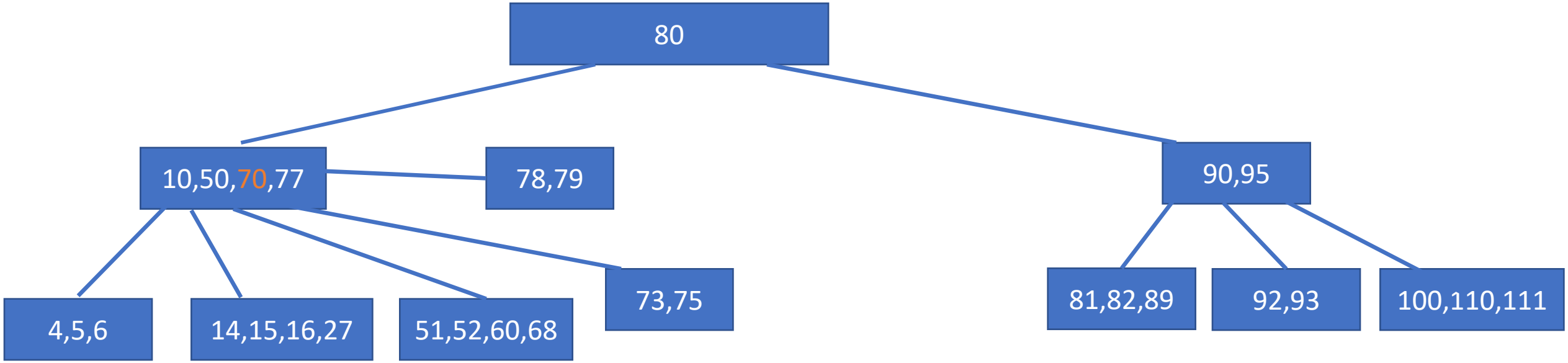
Deletion

After deleting 20



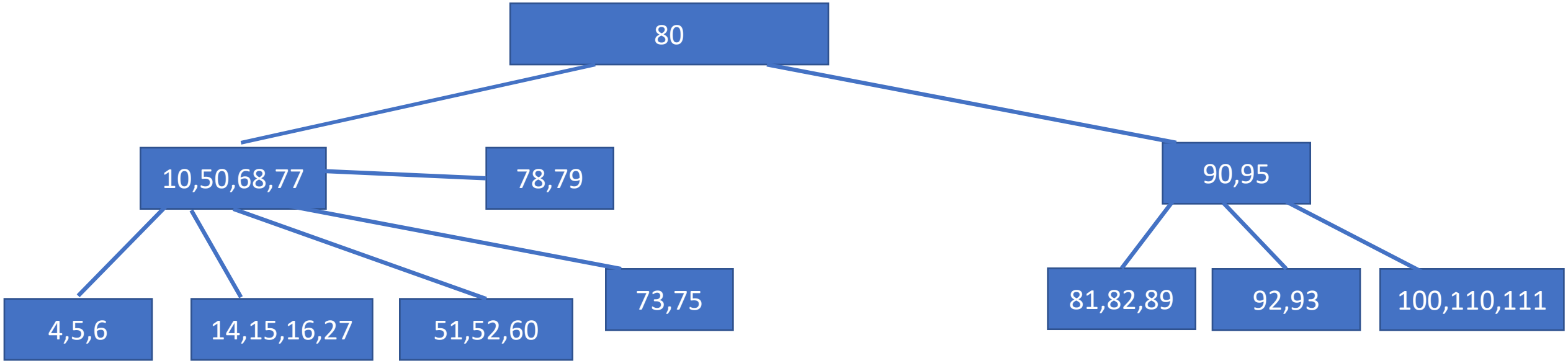
Deletion

Delete 70



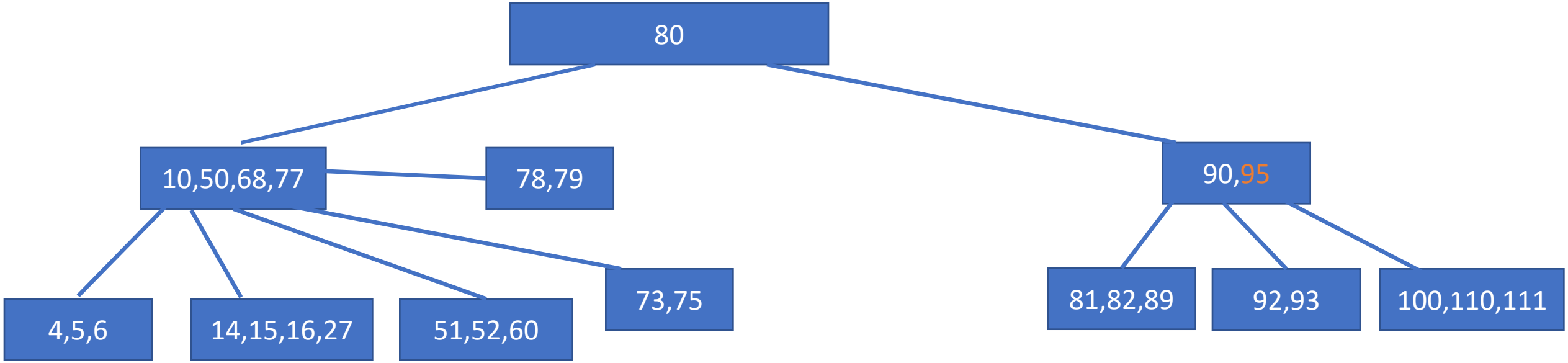
Deletion

After deleting 70



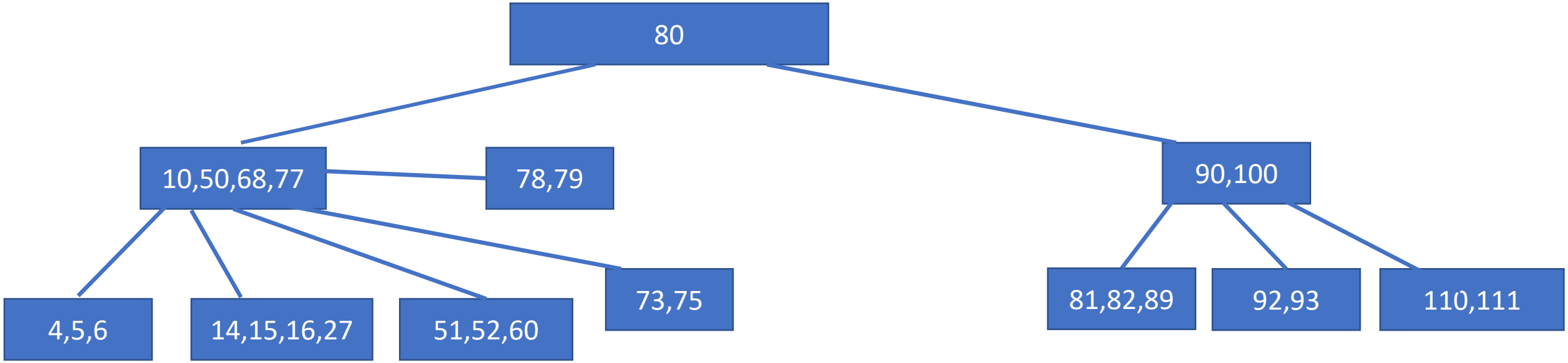
Deletion

Delete 95



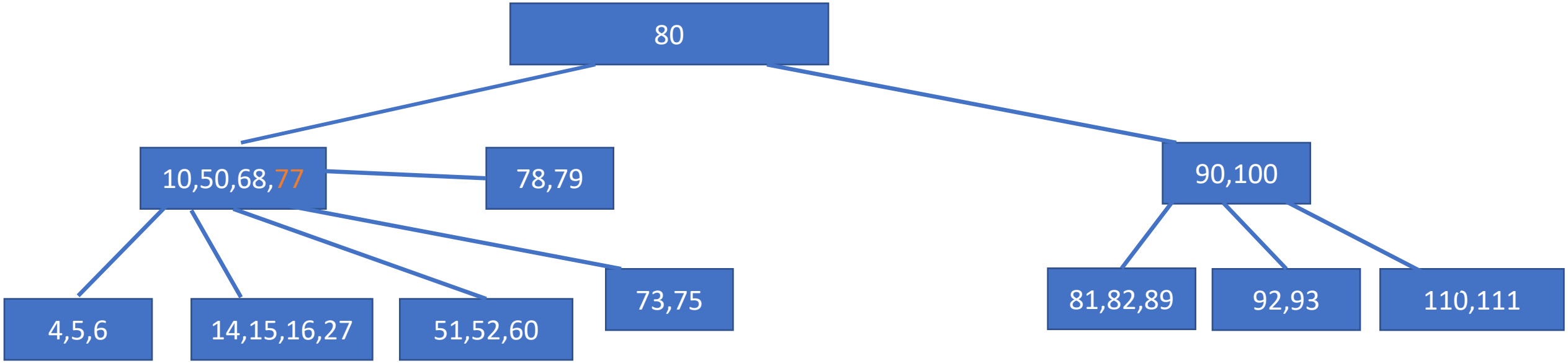
Deletion

After deleting 95



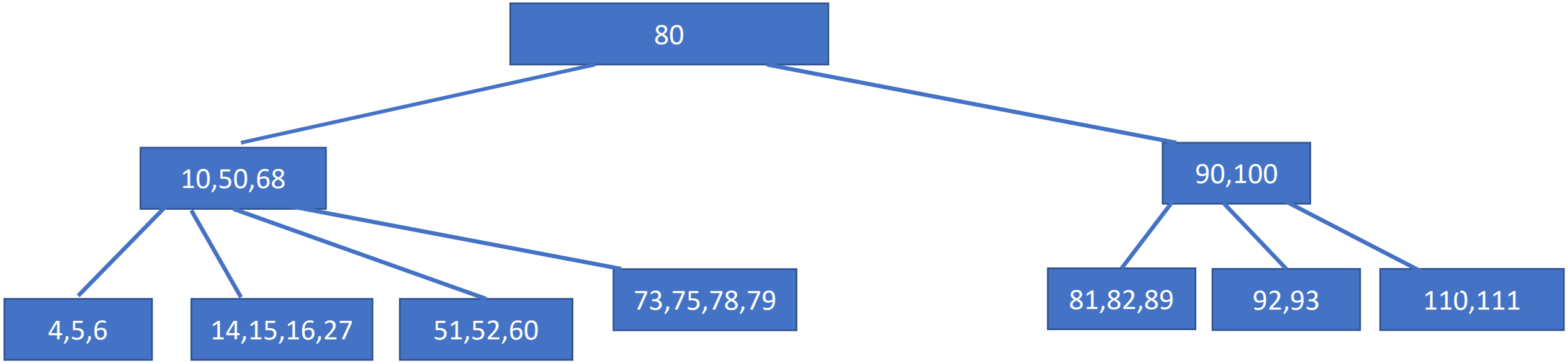
Deletion

Delete 77



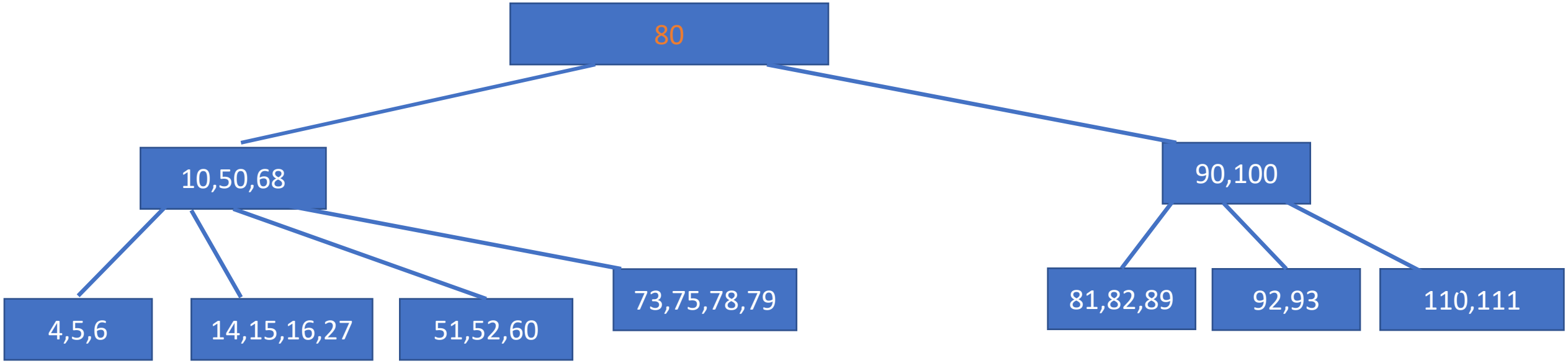
Deletion

After deleting 77



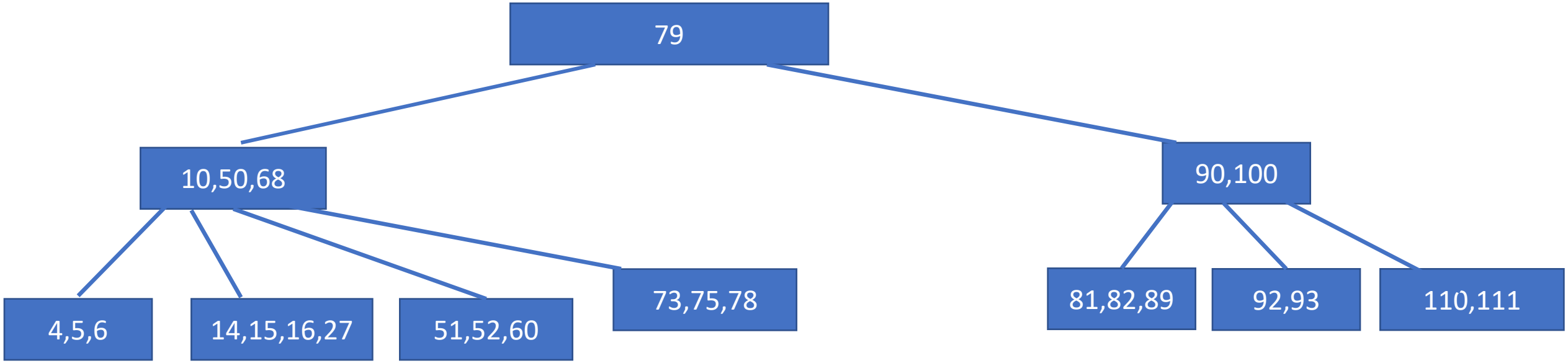
Deletion

Delete 80



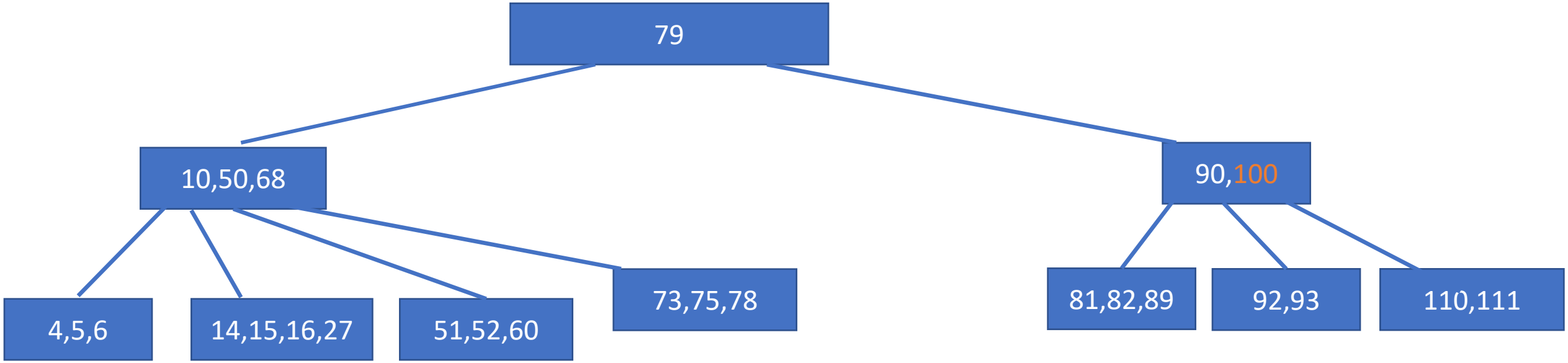
Deletion

After Deleting 80



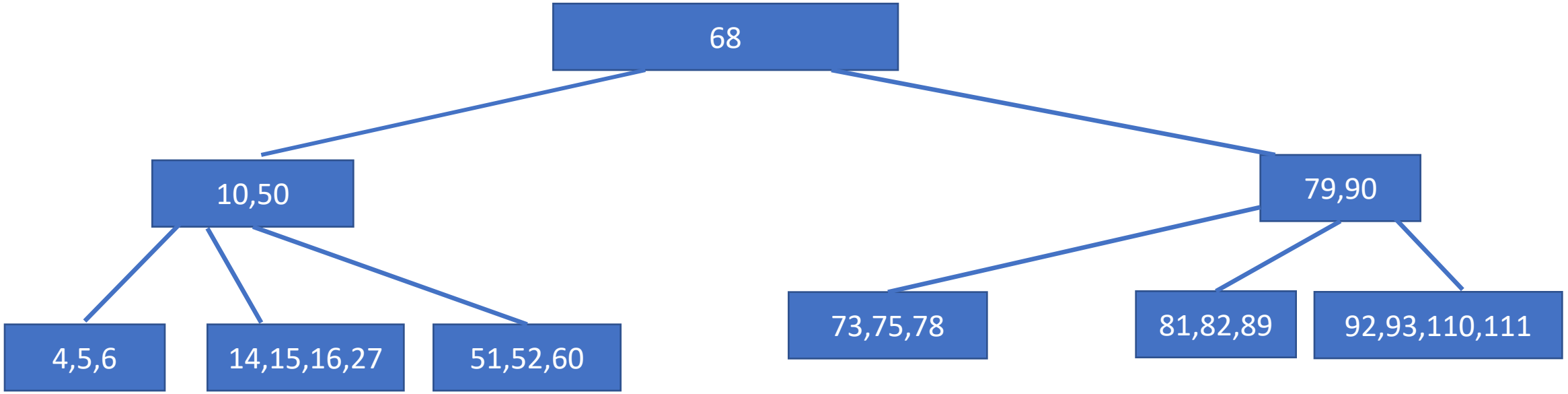
Deletion

Delete 100



Deletion

After deleting 100



Deletion Algorithm

1.Locate the leaf node.
2.If there are more than m/2 keys in the leaf node then delete key
3.If the leaf node doesn't contain m/2 keys then
    a)If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    b)If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4.If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5.If parent is left with less than m/2 nodes then, apply the above process on the parent too.

THANK YOU

Saritha

Department of Computer Science & Engineering

Saritha.k@pes.edu

9844668963

DATA STRUCTURES AND ITS APPLICATIONS

Graphs

Saritha
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Connectivity of the graph

Saritha

Department of Computer Science & Engineering

Applications of BFS

Application of BFS

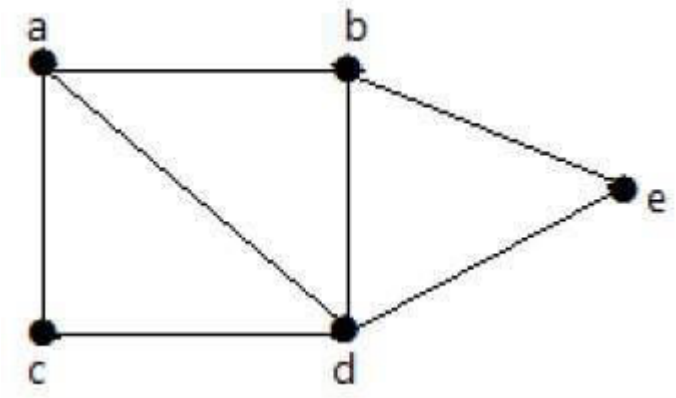
- Finding the shortest path
- Social Networking websites like twitter, Facebook etc.
- GPS Navigation system
- Web crawlers
- Finding a path in network
- In Networking to broadcast the packets

Connectivity of graph

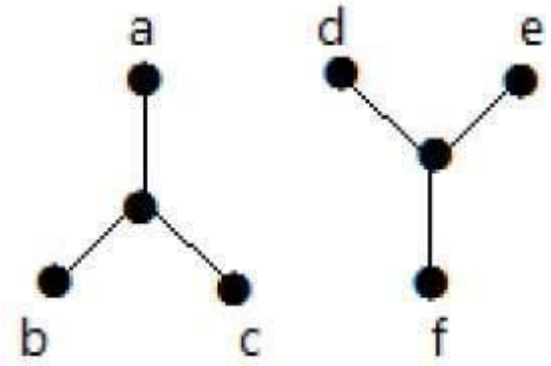
Connectivity of graph

- Connectivity refers to connection between two or more nodes or things

Connected graph



Disconnected graph

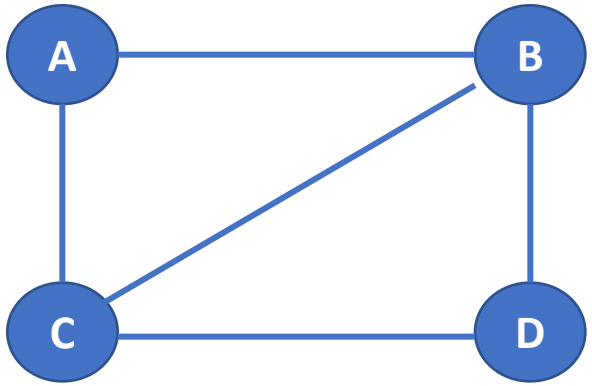


Connectivity of the graph

- A graph is connected if there is a path between every pair of vertices.
- The graph that is not connected is called disconnected graph.
- In connected graph there is no unreachable vertex.
- In the area of Information Technology the connectivity refers to internet connectivity through which various devices are connected to global network

Connectedness in the Direct graph using adjacency matrix

- To check whether a graph is connected or not, we traverse the graph using either bfs traversal or dfs traversal method
- After the traversal if there is at-least one node which is not marked as visited then that graph is disconnected graph
- For example: Below graph is connected



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Connectedness in the Direct graph using adjacency matrix

Procedure to check the whether a graph is connected or not using adjacency matrix

- Read the adjacency matrix .
- Create a visited [] array. Start DFS/BFS traversal method from any arbitrary vertex and mark the visited vertices in the visited[] array.
- Once DFS/BFS is completed check the visited [] array. if there is at-least one vertex which is marked as unvisited then the graph is disconnected otherwise it is connected.

Connectivity of the graph using Adjacency Matrix

Function to read adjacency matrix:

```
void read_ad_matr(int graph[][],int n)
{
   int i,j;
   for(i=0;i<n;i++)
    {
      for(j=0;j<n;j++)
        {
          scanf("%d", &graph[i][j]);//reading the values into two dimensional array
         }
    }
}
```

Connectivity of the graph using Adjacency Matrix

Function to traverse the graph using DFS

```
void traverse(int u, int visited[])
{
   visited[u] = 1; //mark v as visited
   for(int v = 0; v<n; v++)
  {
     if(graph[u][v])
     {
       if(!visited[v])
       traverse(v, visited);
     }
  }
}
```

Connectivity of the graph using Adjacency Matrix

Function to traverse the graph using bfs:

```
void traverse(int s, int visited[],int n,int graph[][])
 {
   int f,r,v,q[10];
   f=0,r=-1;
   q[++r]=s;
   visited[s]=1;// mark the nodes as visited
   while(f<=r)
   {
      s=q[f++];
      for(v=0;v<n;v++)
     {
      if(graph[s][v]==1) //adjacent nodes
       {
```

Connectivity of the graph using Adjacency Matrix


```
    if(visited[v]==0) //nodes not visited
     {
       visited[v]=1; //mark as visited
       q[++r]=v;
     }
}
 }
   }
}
```

Connectivity of the graph using Adjacency Matrix

Function to check whether the graph is connected or not

```
int isConnected()
{
   int vis[NODE];   //for all vertex u as start point, check whether all nodes are visible or not
 for(int u=0; u < NODE; u++)
    {
      for(int i = 0; i<NODE; i++)
          vis[i] = 0; //initialize as no node is visited
      traverse(u, vis); // any traversal method either bfs or dfs
       for(int i = 0; i<NODE; i++)
       {
          if(!vis[i]) //if there is a node, not visited by traversal, graph is not connected
          return 0;
        }
    }
 return 1;}
```

THANK YOU

Saritha

Department of Computer Science & Engineering

Saritha.k@pes.edu

9844668963