UNIT-5

Prepared by:

Manjula L, Assistant Professor

Dept. of CSE, MSRIT

Text Book: Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2008.

Contents

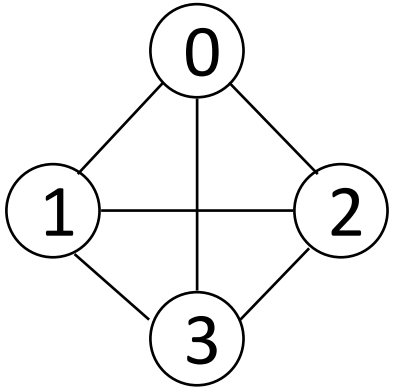
- Graphs: The Graph Abstract Data Type
- Elementary Graph Operations.
- Priority Queues: Single- and Double-Ended Priority Queues
- Leftist Trees.
- Efficient Binary Search Trees: AVL Trees.

Graphs

*A **Graph** is a **non-linear data structure** consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).*

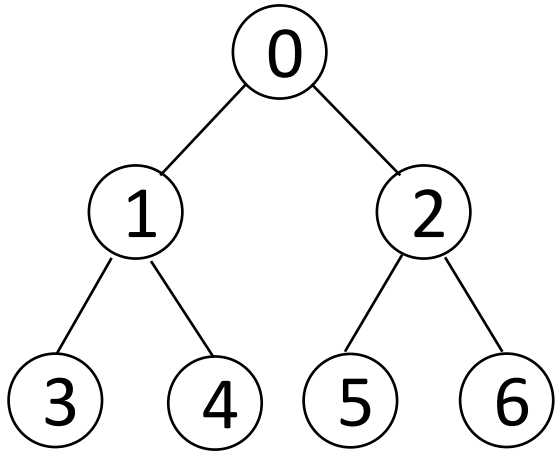
- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph
- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1> != <v_1, v_0>$

Examples for Graph

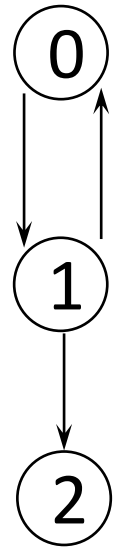


G_1

complete graph



G_2



incomplete graph

G_3

$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

$V(G_2)=\{0,1,2,3,4,5,6\}$ $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$ $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

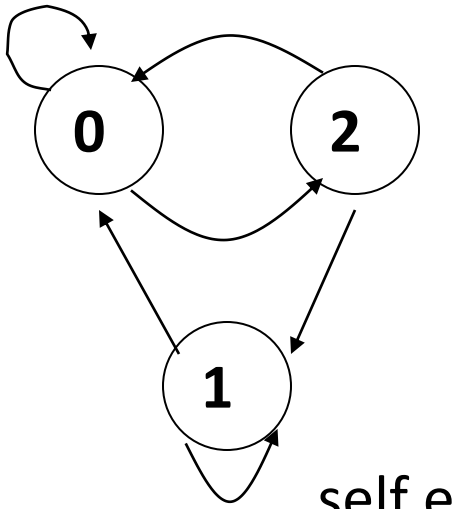
complete undirected graph: n(n-1)/2 edges

complete directed graph: n(n-1) edges

Adjacent and Incident

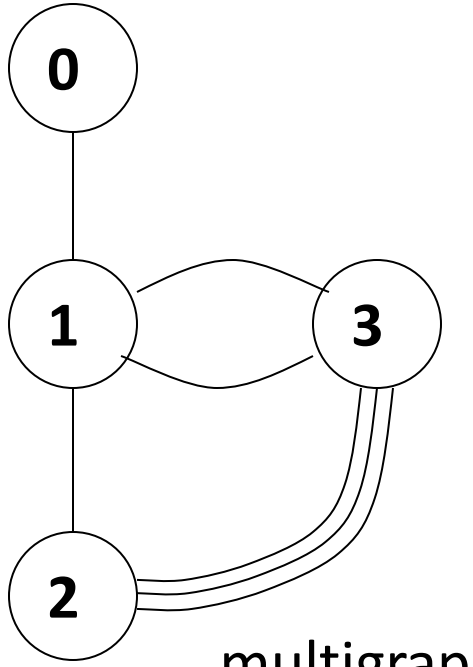
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

***Figure 6.3:**Example of a graph with feedback loops and a multigraph (p.260)



self edge

(a)



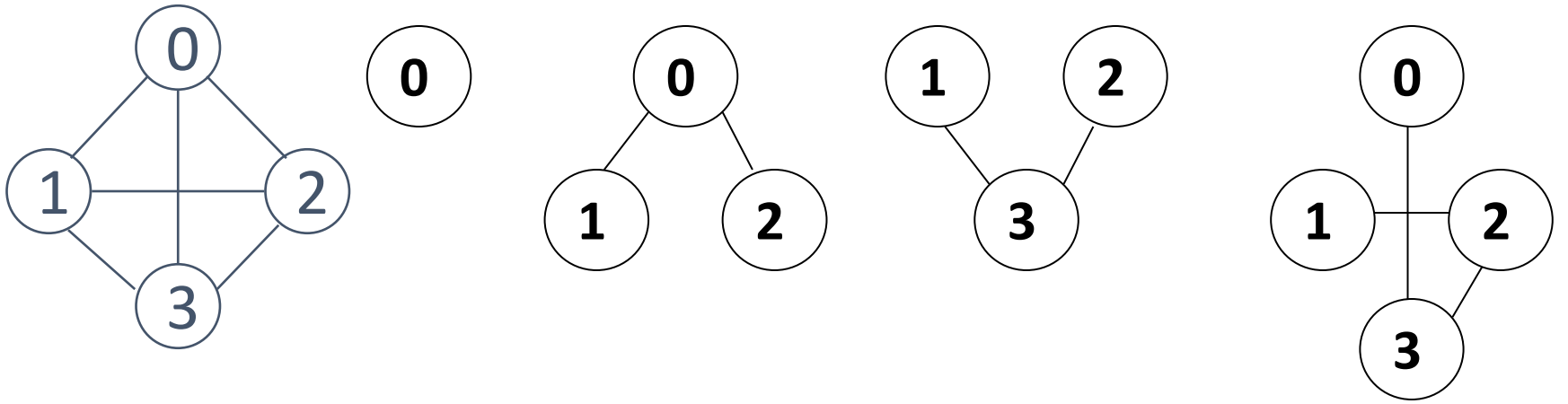
(b)

multigraph:
multiple occurrences
of the same edge

Subgraph and Path

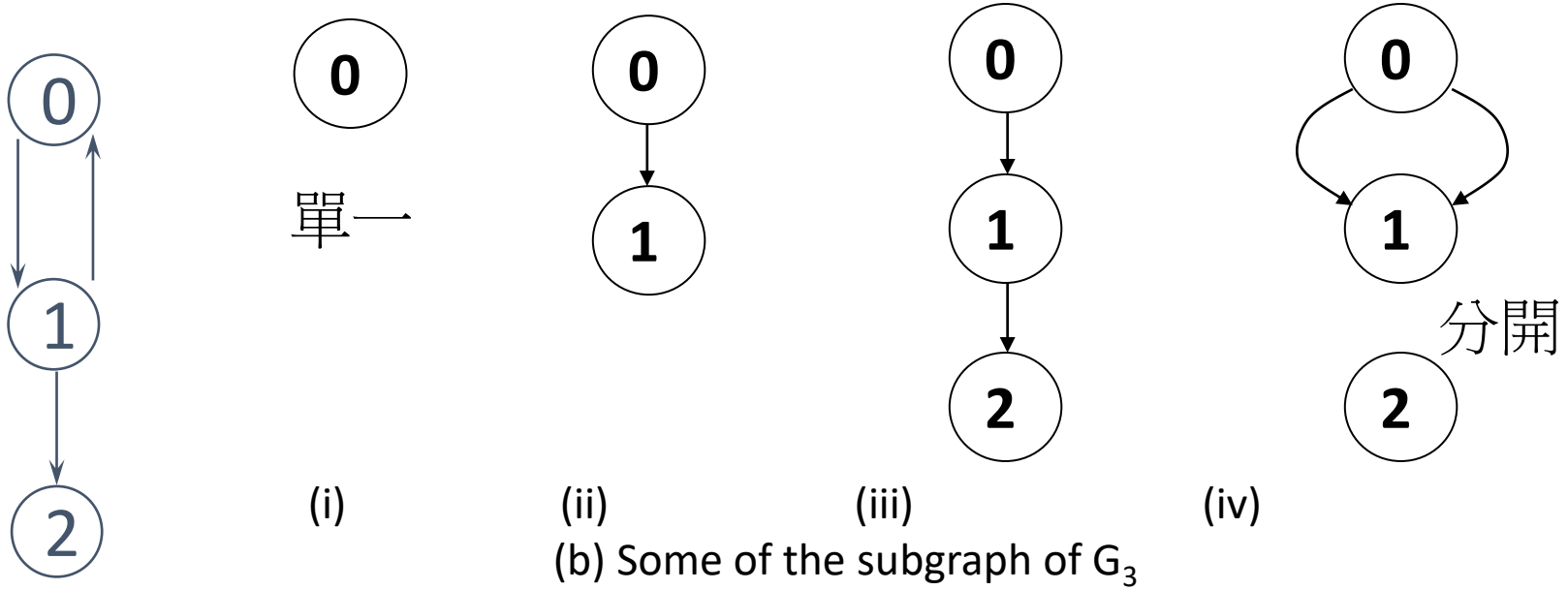
- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph
- The length of a path is the number of edges on it

Figure 6.4: subgraphs of G_1 and G_3 (p.261)



G_1 (i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1



(i) (ii) (iii) (iv)

(b) Some of the subgraph of G_3

G_3

Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
- An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

Connected Component

- A connected component of an undirected graph is a maximal connected subgraph.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i.
- A strongly connected component is a maximal subgraph that is strongly connected.

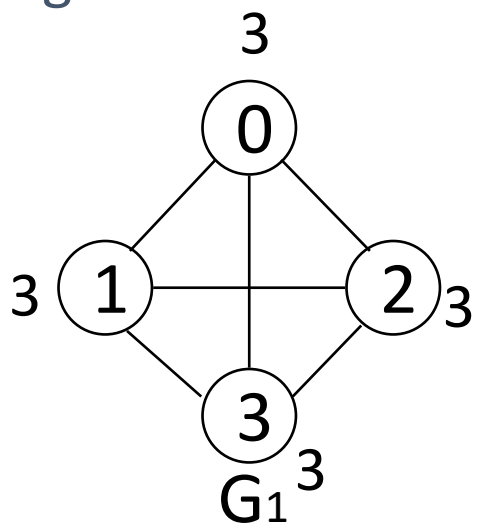
Degree

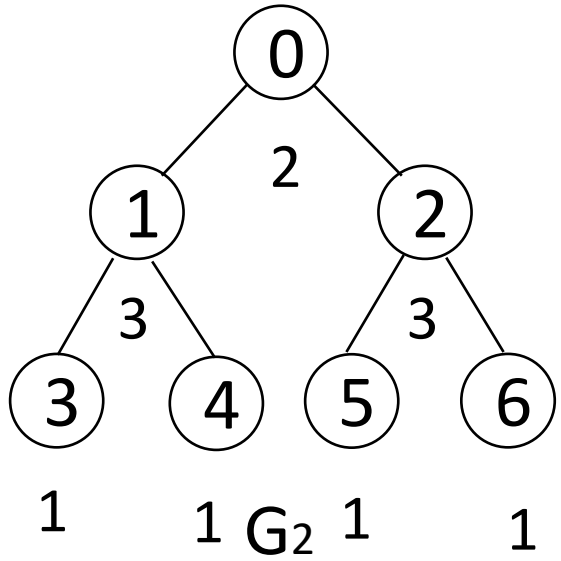
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i) / 2$$

undirected graph

degree

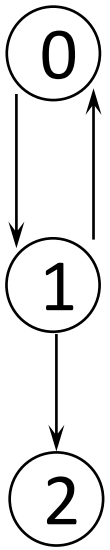




directed graph

in-degree

out-degree



in:1, out: 1

in: 1, out: 2

in: 1, out: 0

ADT for Graph

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, *v*, v_1 and v_2 $\in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE

else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

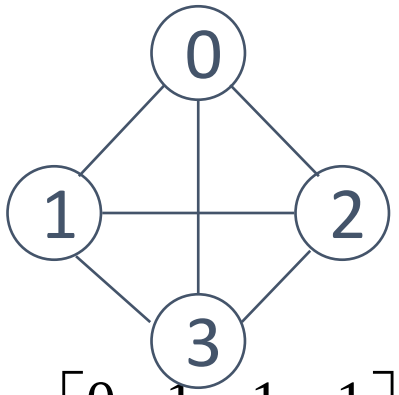
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

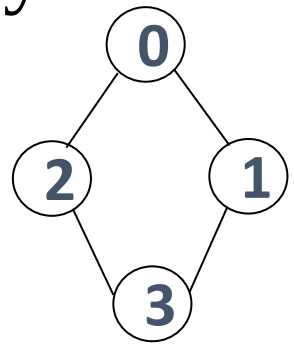
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

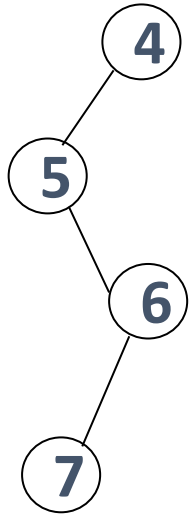
G_2

symmetric

undirected: $n^2/2$

directed: n^2





$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

Merits of Adjacency Matrix

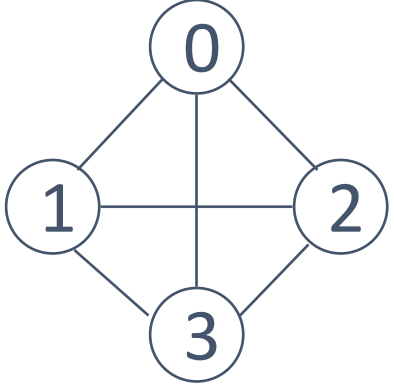
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

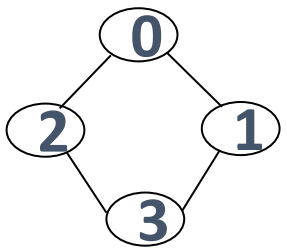
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

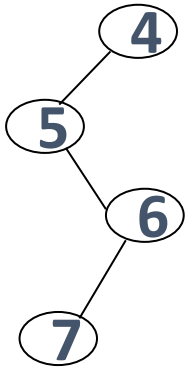
Data Structures for Adjacency Lists

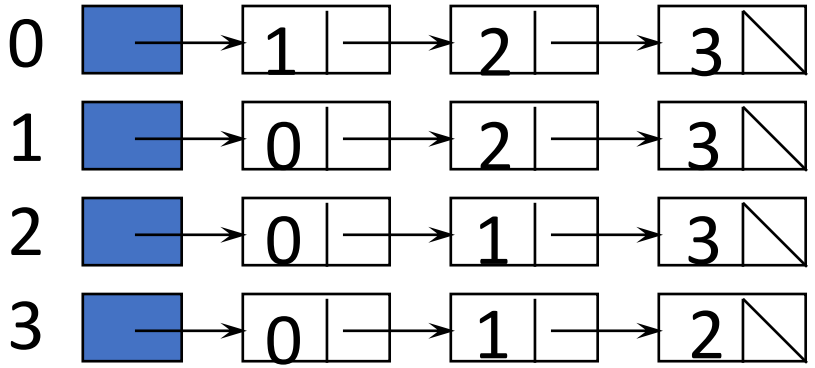
Each row in adjacency matrix is represented as an adjacency list.

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

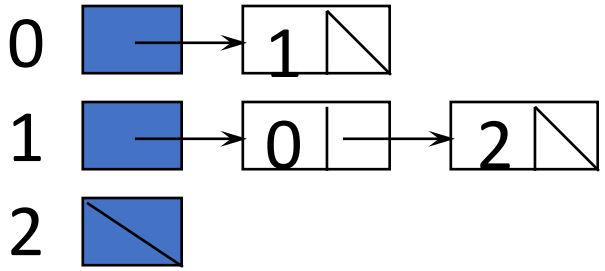




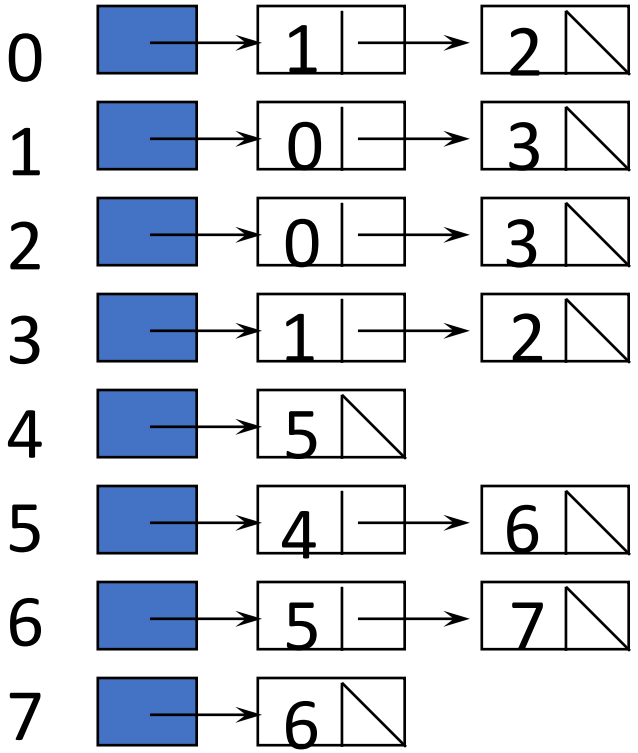




G_1



G_3



G_4

An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

Interesting Operations

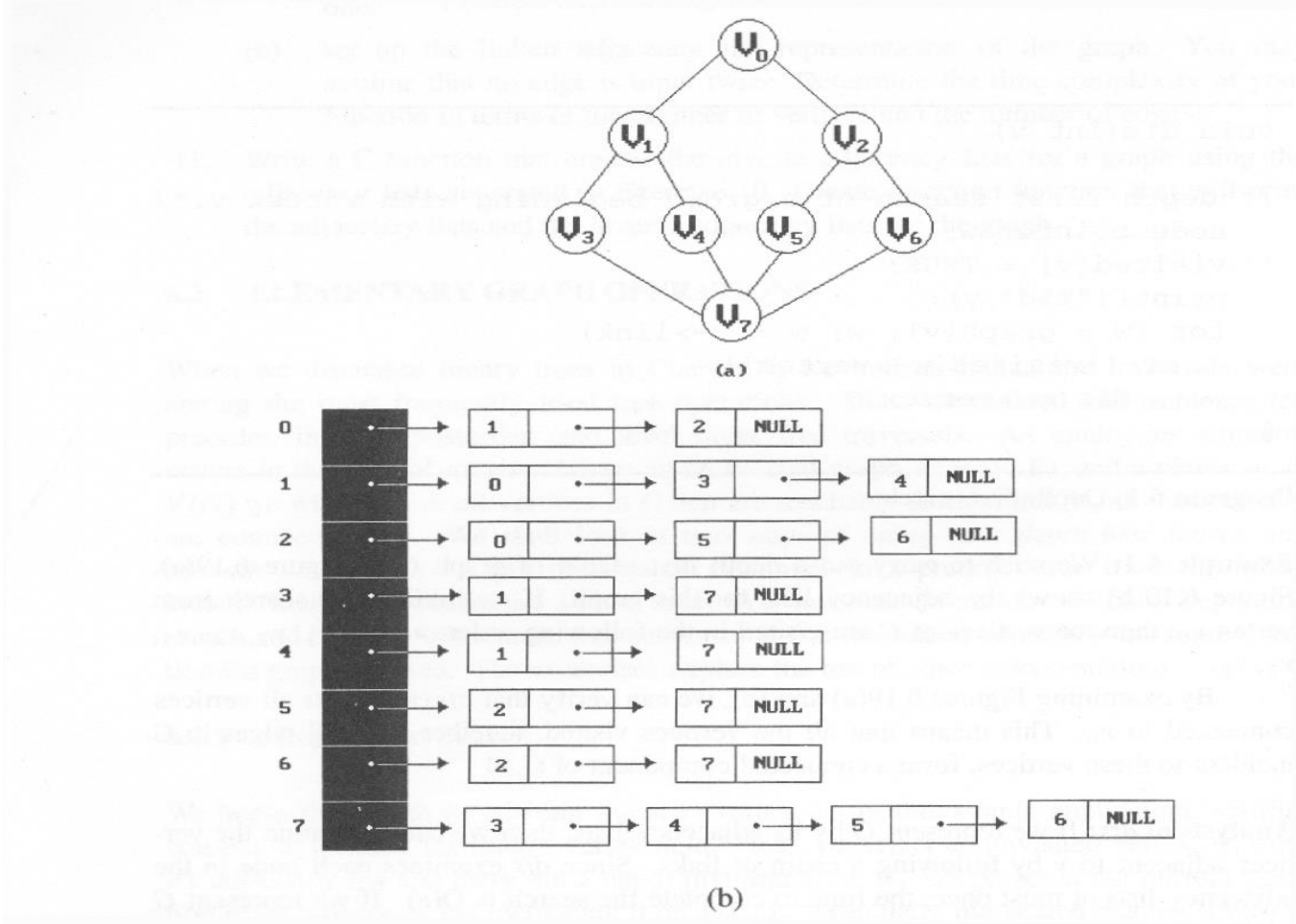
- degree of a vertex in an undirected graph
  - # of nodes in adjacency list
- # of edges in a graph
  - determined in O(n+e)
- out-degree of a vertex in a directed graph
  - # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - traverse the whole data structure

Some Graph Operations

- Traversal
  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.
  - Depth First Search (DFS)
    preorder tree traversal
  - Breadth First Search (BFS)
    level order tree traversal
- Connected Components
- Spanning Trees

*Figure 6.19:Graph *G* and its adjacency lists (p.274)

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%5d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Data structure
adjacency list: O(e)
adjacency matrix: $O(n^2)$

Breadth First Search

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(queue_pointer *,
          queue_pointer *, int);
int deleteq(queue_pointer *);
```

Breadth First Search (*Continued*)

```
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%5d", v);
  visited[v] = TRUE;
  addq(&front, &rear, v);
```

adjacency list: O(e)
adjacency matrix: $O(n^2)$

```
    while (front) {
      v= deleteq(&front);
      for (w=graph[v]; w; w=w->link)
        if (!visited[w->vertex]) {
          printf("%5d", w->vertex);
          addq(&front, &rear, w->vertex);
          visited[w->vertex] = TRUE;
        }
    }
}
```

Connected Components

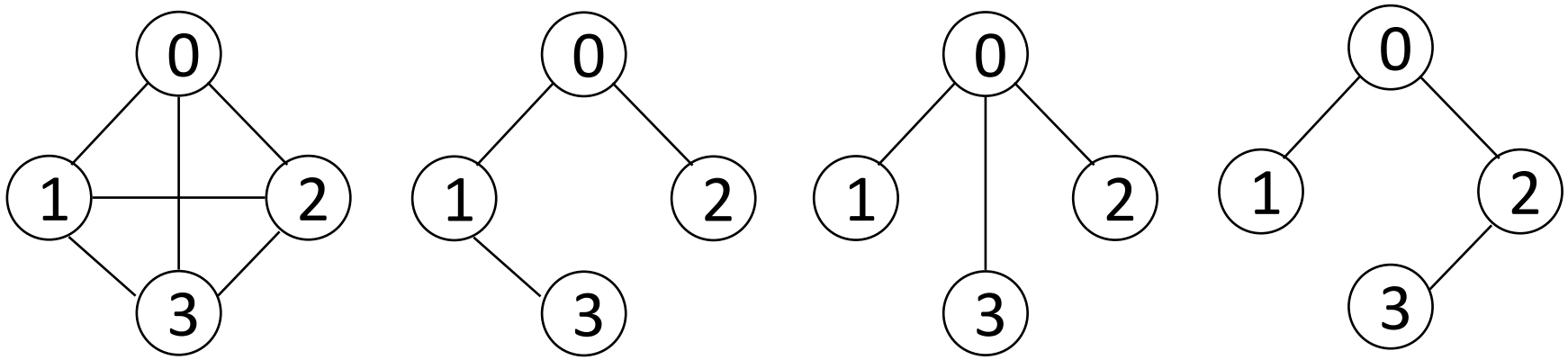
```
void connected(void)
{
    for (i=0; i<n; i++) {
        if (!visited[i]) {
            dfs(i);
            printf("\n");
        }
    }
}
```

adjacency list: O(n+e)
adjacency matrix: $O(n^2)$

Spanning Trees

- When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G
- A spanning tree is any tree that consists solely of edges in G and that includes all the vertices
- E(G): T (tree edges) + N (nontree edges)
  where T: set of edges used during search
  N: set of remaining edges

Examples of Spanning Tree



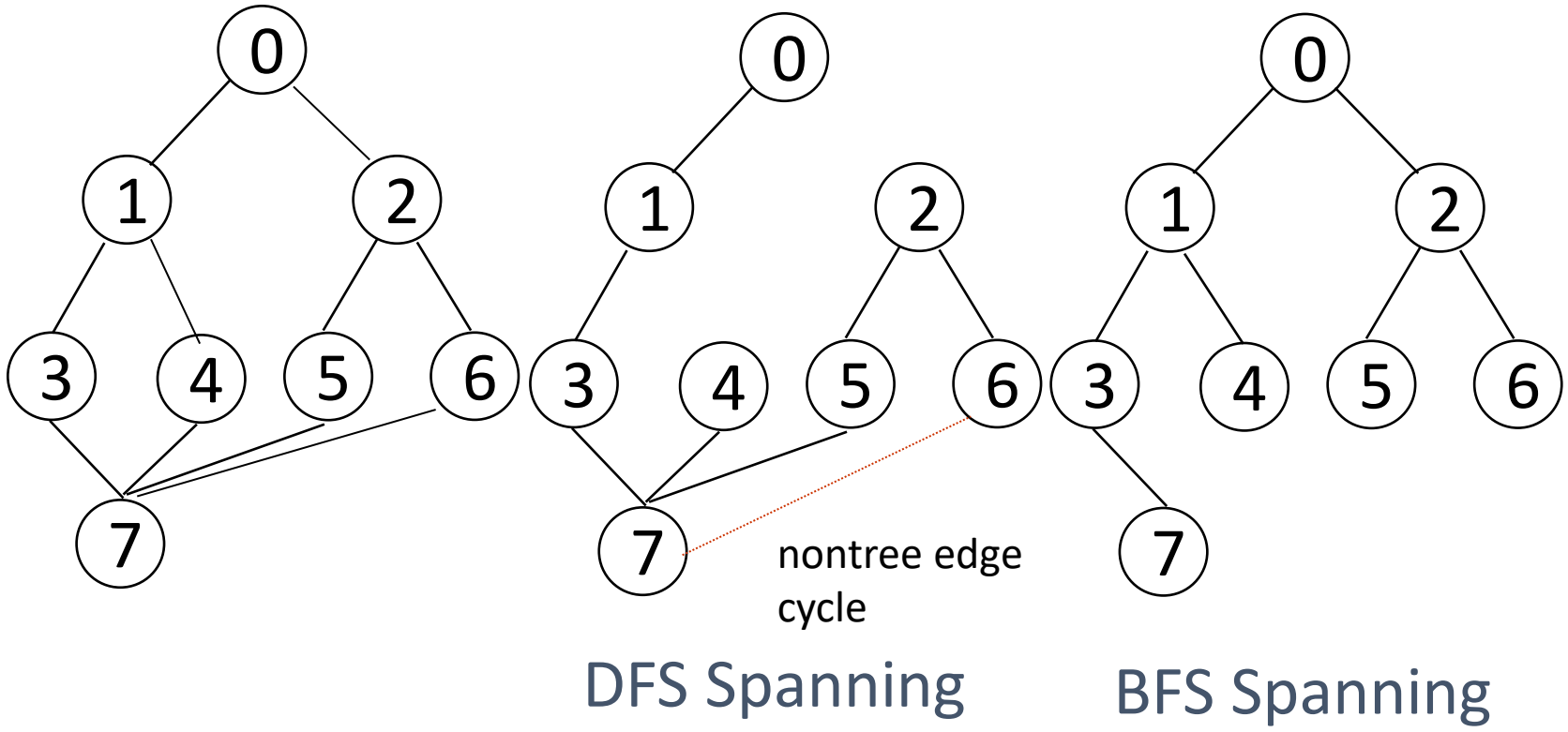
G_1

Possible spanning trees

Spanning Trees

- Either dfs or bfs can be used to create a spanning tree
  - When dfs is used, the resulting spanning tree is known as a depth first spanning tree
  - When bfs is used, the resulting spanning tree is known as a breadth first spanning tree
- While adding a nontree edge into any spanning tree, this will create a cycle

DFS VS BFS Spanning Tree



A spanning tree is a minimal subgraph, G’, of G such that V(G’)=V(G) and G’ is connected.

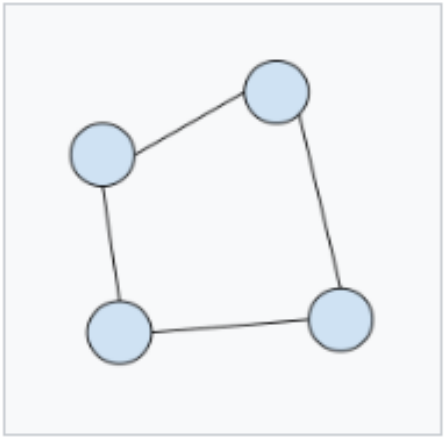
Any connected graph with n vertices must have at least n-1 edges.

Applications of the Spanning Tree

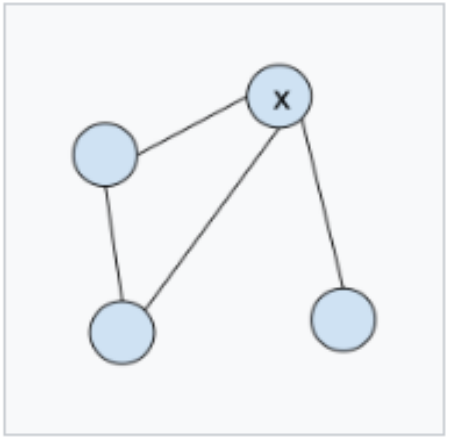
- A spanning tree is preferred to discover the shortest path to link all nodes of the graph. In this section, we will discuss some of the most famous and common applications of the spanning tree:
- A spanning tree is very helpful in the civil network zone and planning.
- It is used in network routing protocol.

Biconnected Components

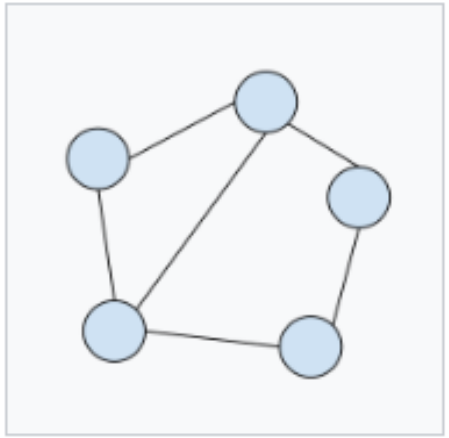
A **biconnected** undirected graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex (and its incident edges).



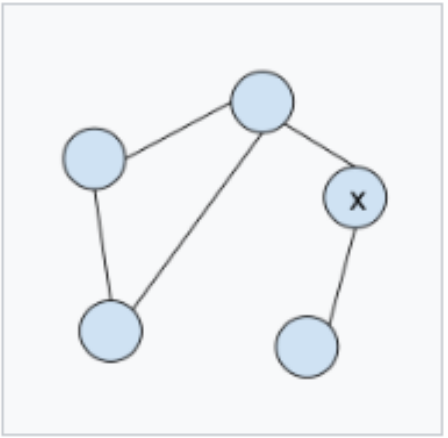
A biconnected graph on four vertices and four edges



A graph that is not biconnected. The removal of vertex x would disconnect the graph.



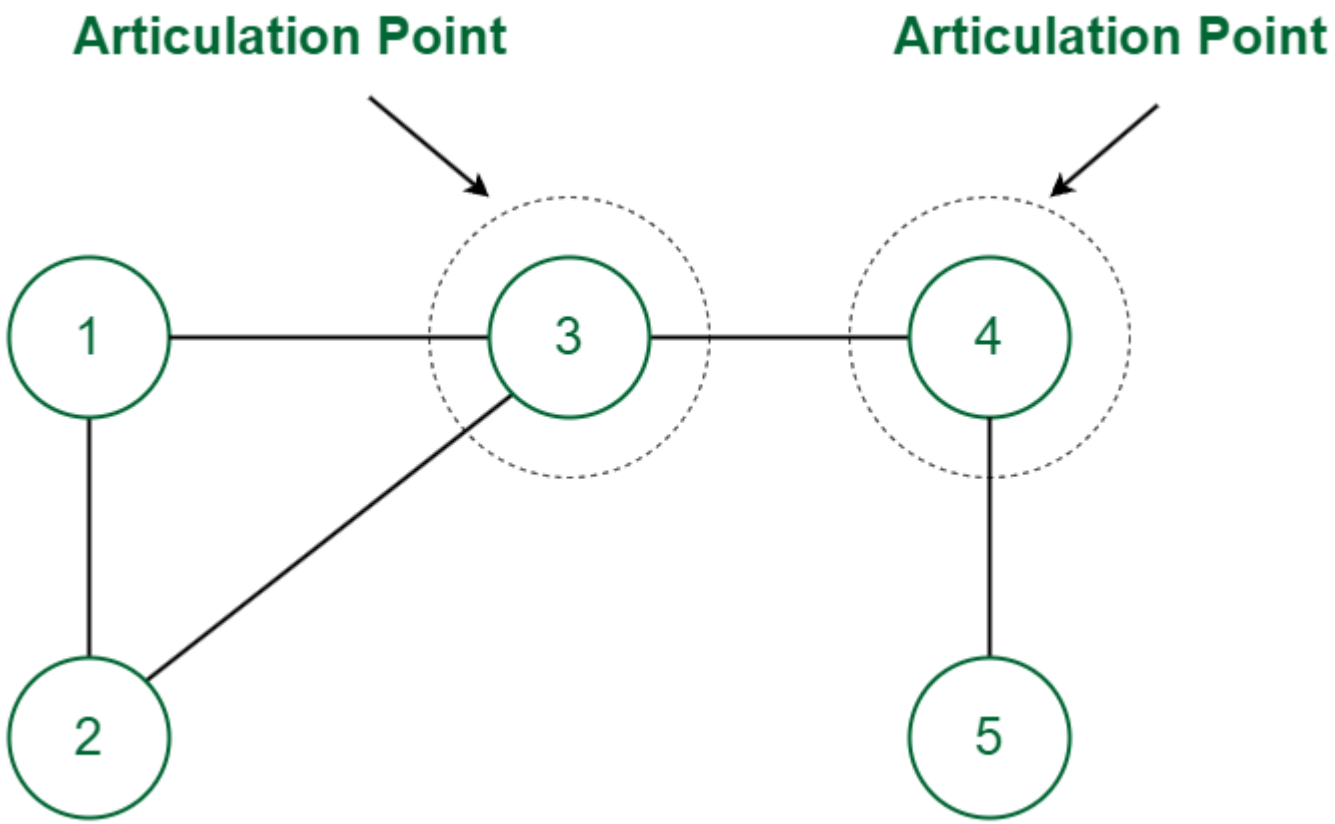
A biconnected graph on five vertices and six edges

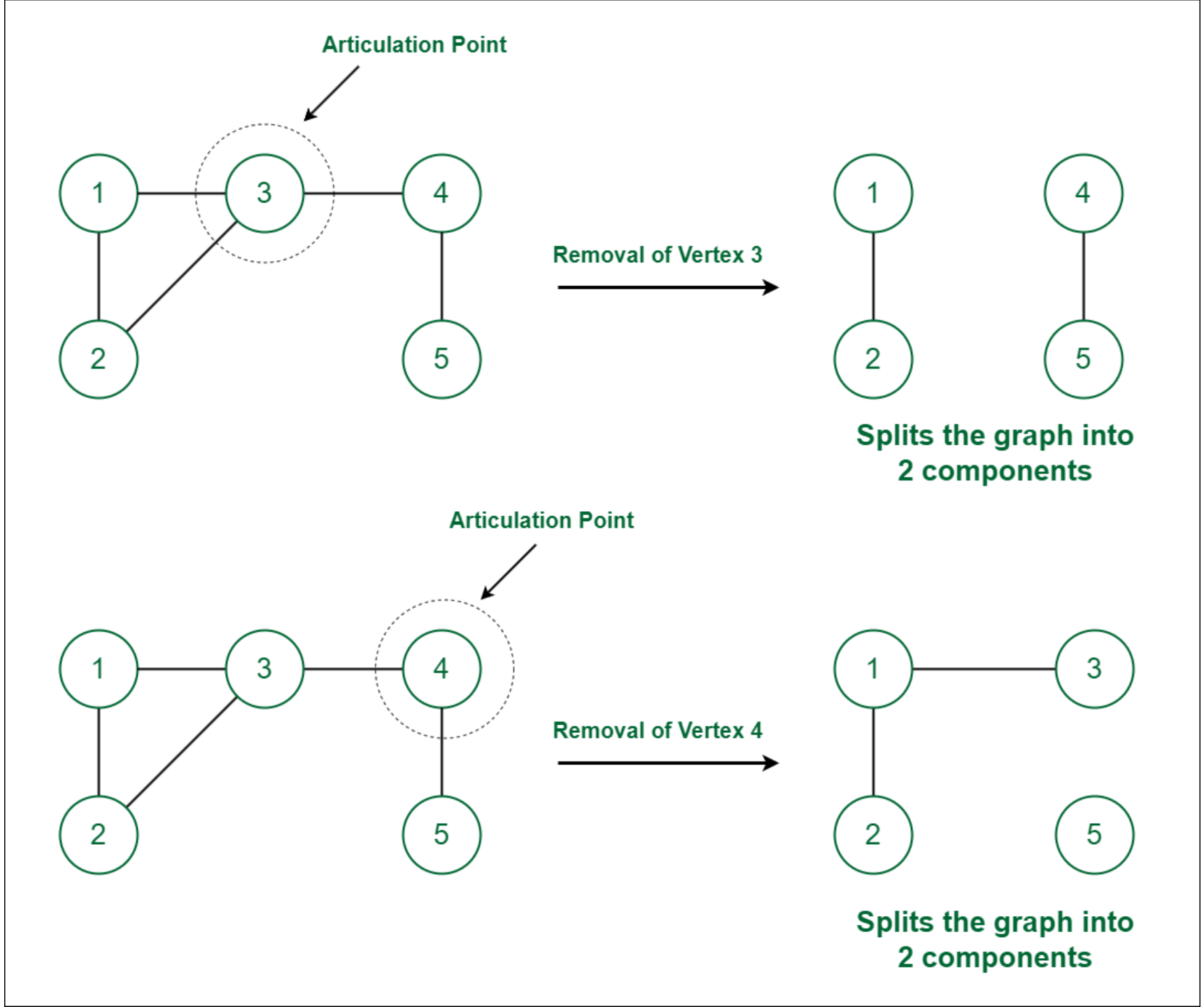


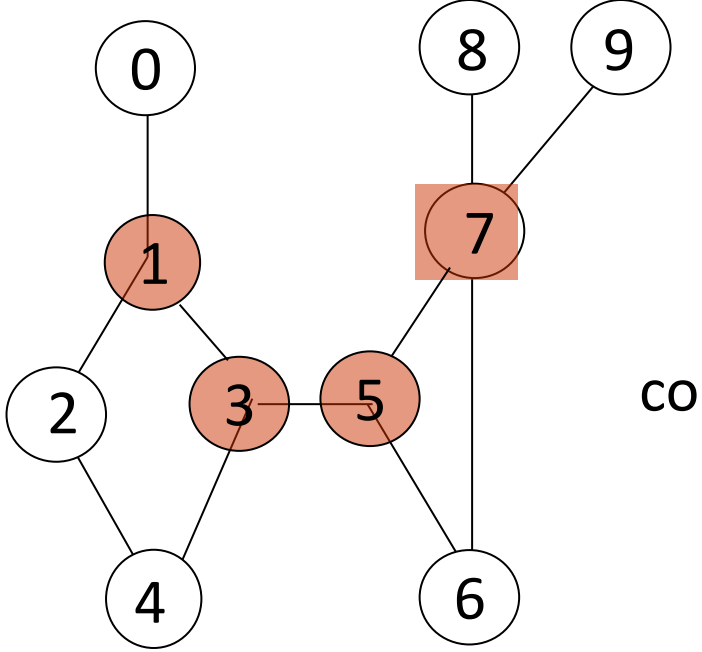
A graph that is not biconnected. The removal of vertex x would disconnect the graph.

Articulation Points (or Cut Vertices) in a Graph

- A vertex **v** is an **articulation point** (also called cut vertex) if removing **v** increases the number of connected components(graph count).
- Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks.

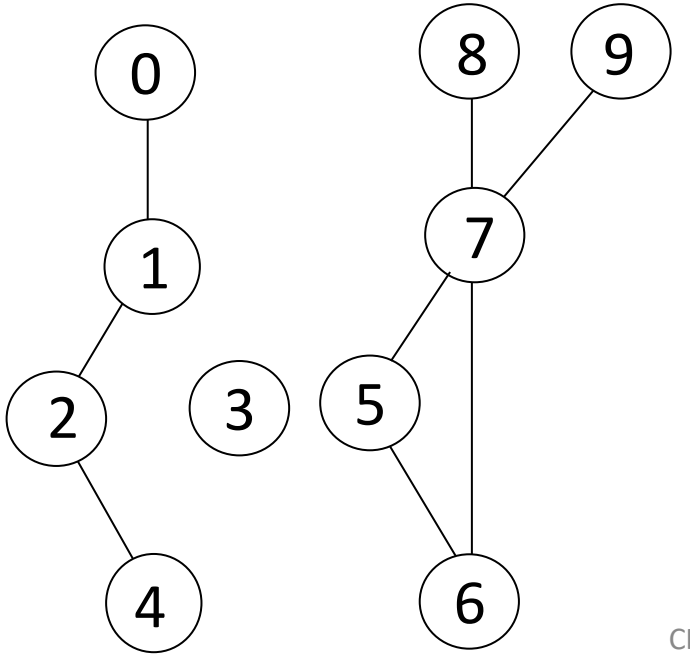

Articulation Point
Articulation Point
1
3
4
2
5


Articulation Point
1
3
4
2
5
Removal of Vertex 3
1
4
2
5
Splits the graph into
2 components
Articulation Point
1
3
4
2
5
Removal of Vertex 4
1
3
2
5
Splits the graph into
2 components

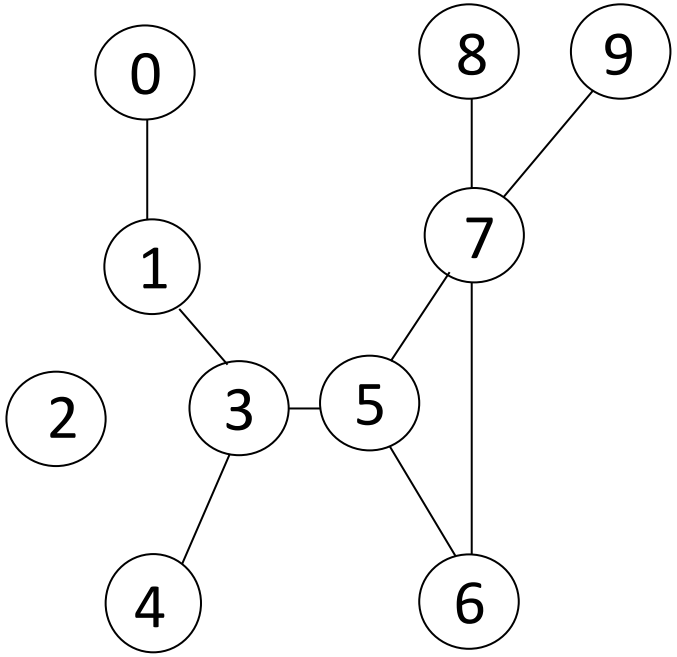


connected graph

two connected components



one connected graph



Priority Queues

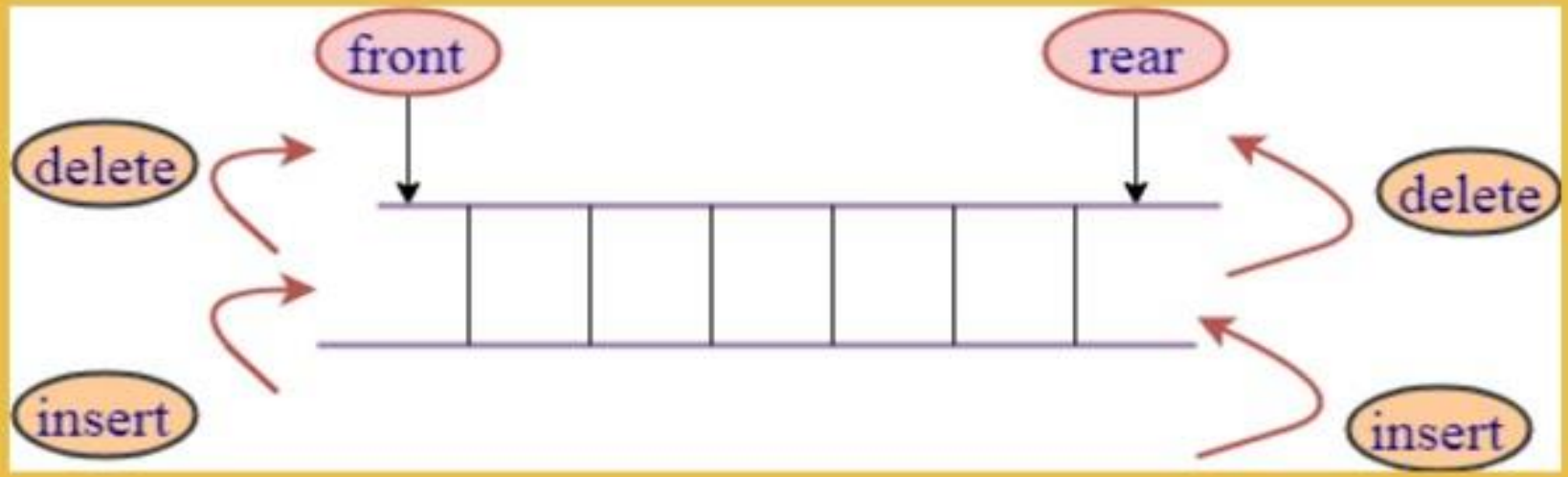
- A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.
- In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back
- Two Varieties:
  - Single-ended
  - Double-ended

Single-ended Priority Queue

Operations supported by single ended priority queue:

- SP1: Return an element with minimum/maximum priority.
- SP2: Insert an element with arbitrary priority.
- SP3: Delete an element with minimum /maximum priority.

Double ended Queue



Double-ended Priority Queue

Operations supported by Double ended priority queue:

- DP1: Return an element with minimum priority.
- DP2: Return an element with maximum priority.
- DP3: Insert an element with arbitrary priority.
- DP4: Delete an element with minimum priority.
- DP5: Delete an element with maximum priority.

DEPQ used as network buffer

The buffer holds the packets that are waiting their turn to be sent out over a network link, each packet as an associated priority.

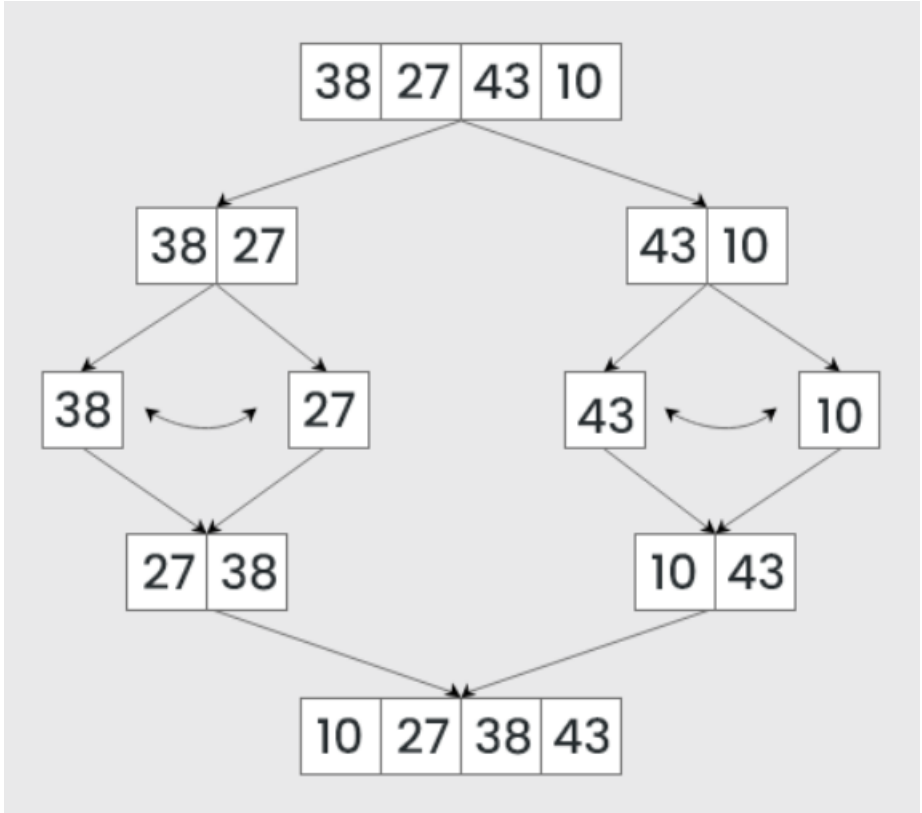
DeleteMax : When the network link become available, a packet with highest priority is deleted.

Insert : When the packet arrives to the network, added to the buffer.

DeleteMin: If the buffer is full, the packets with minimum priority are dropped.

Merge Sort-DEPQ

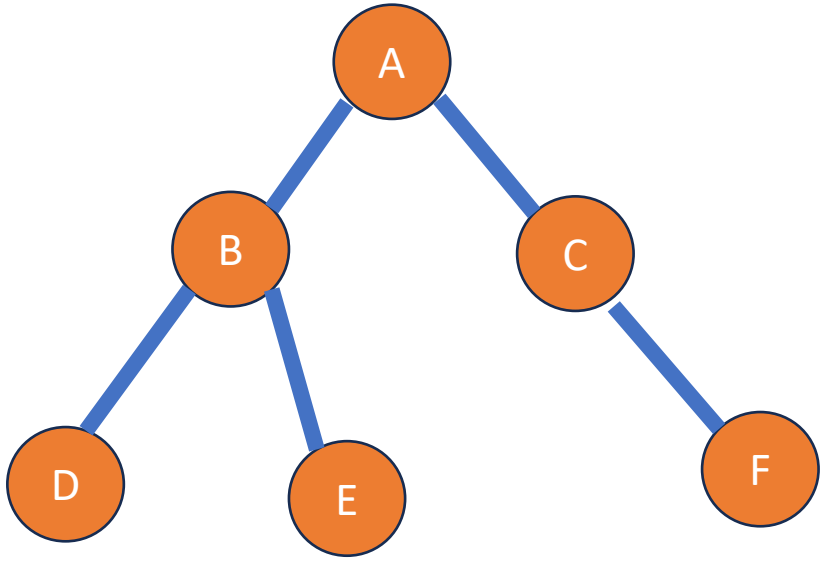
- Read in as many elements as will fit into an internal DEPQ. The elements in the DEPQ will eventually be the middle group of elements.
- Process the remaining element one at a time.
- Sort the left and right groups recursively.

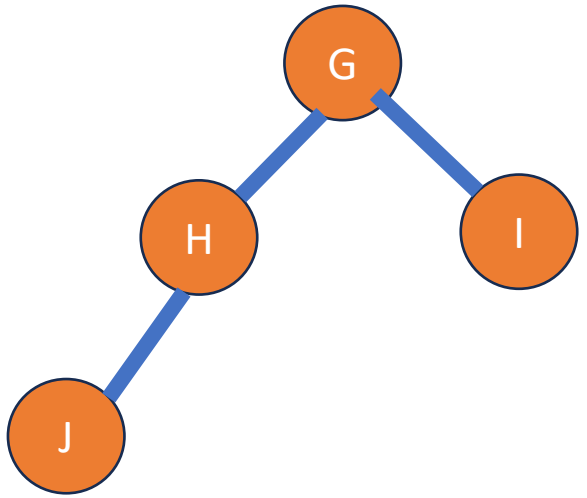


Leftist Tree

- Leftist tree provide an efficient implementation of meldable priority queues.,
- Let n be the total number of elements in the two priority queues that are to be melded, if heap are used to represent meldable queues, it takes o(n) time. Usage of leftist tree takes logarithmic time.
- It used concept of extended binary tree.
- Two Varieties:
  - Height Biased
  - Weight Biased

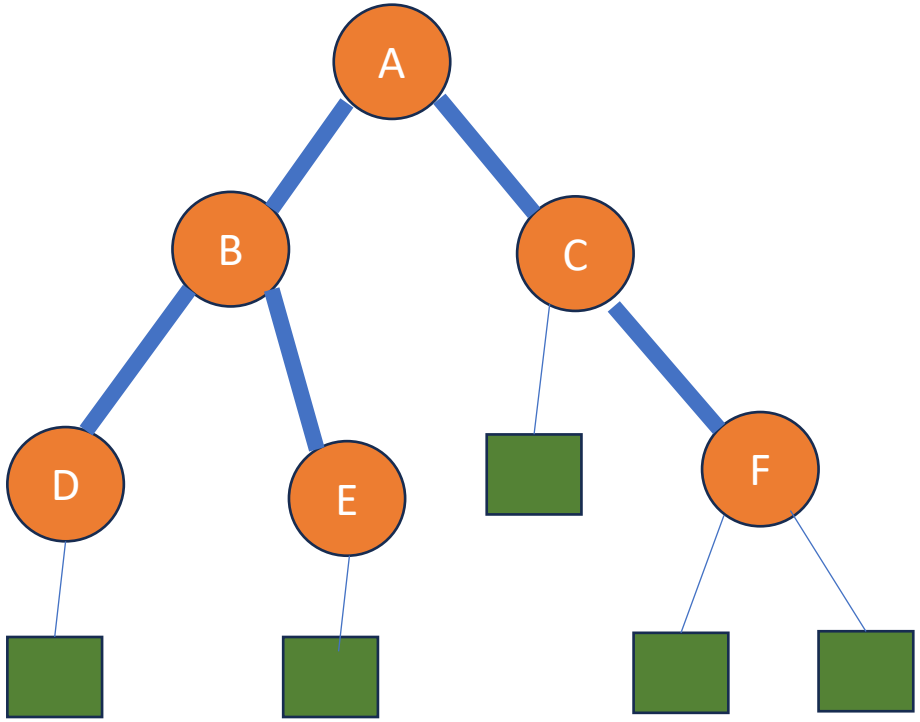
Height Biased Leftist Tree

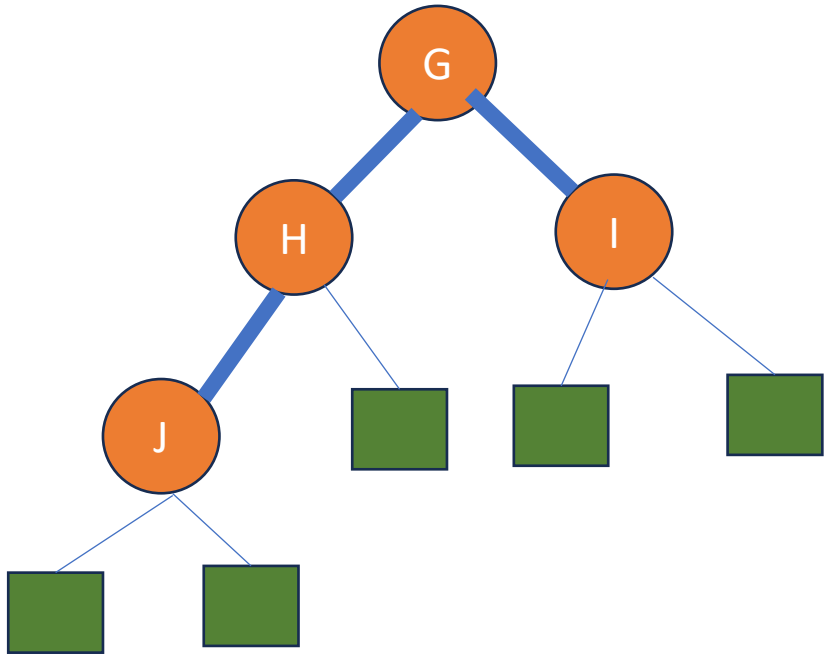




Binary trees

Height Biased Leftist Tree





Extended Binary trees

Shortest(x)

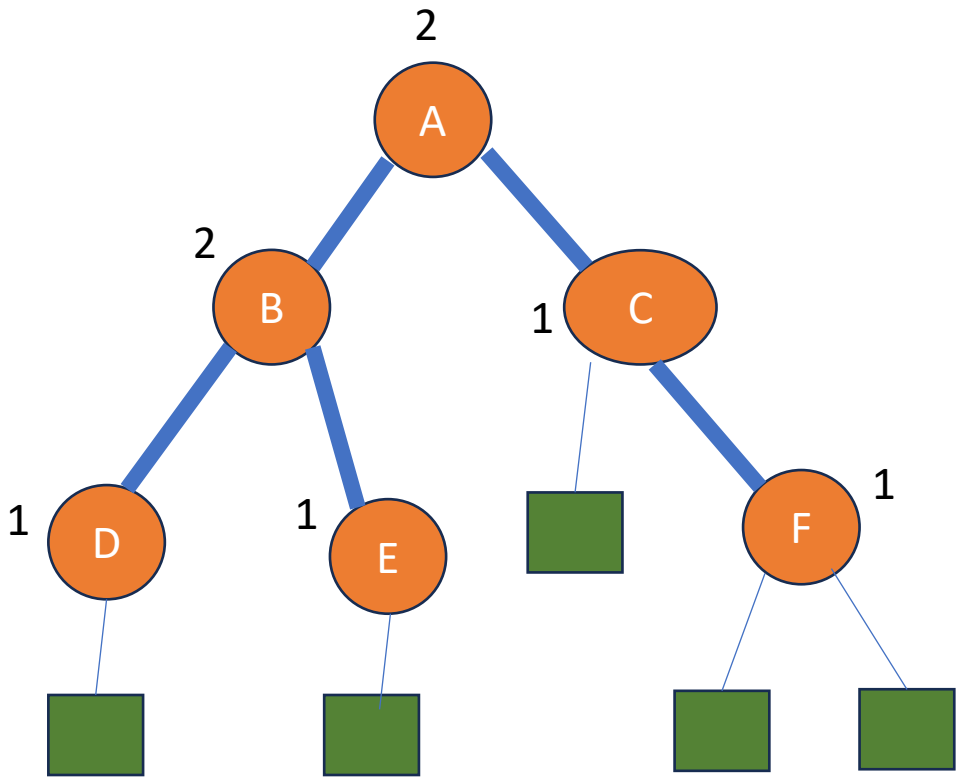
Now suppose s(x) be the length of a shortest path from node x to an external node in its subtree. If x is an external node, its s(x) value is 0. If the x is an internal node, the value is −

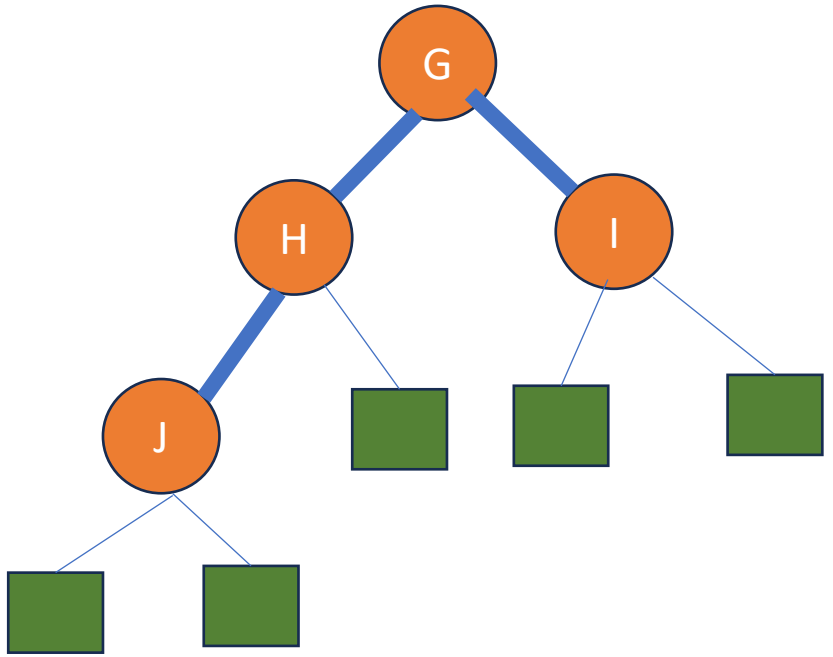
$$min\{s(L), s(R)\} + 1$$

- A leftist tree is a binary tree such that if it is not empty , then

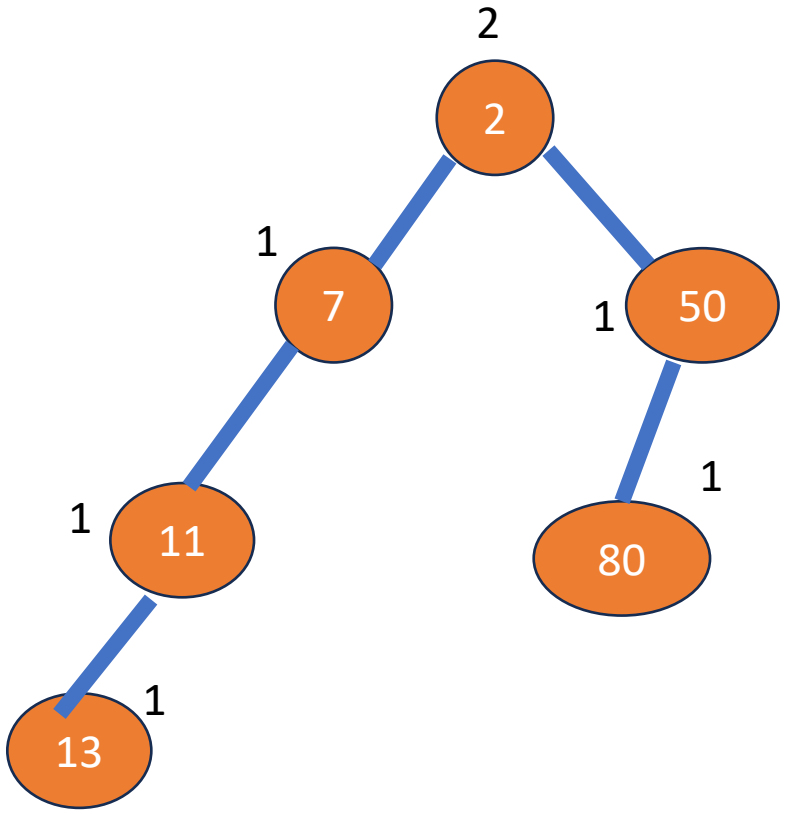
$$shortest(leftchild(x)) > shortest(rightchild(x))$$

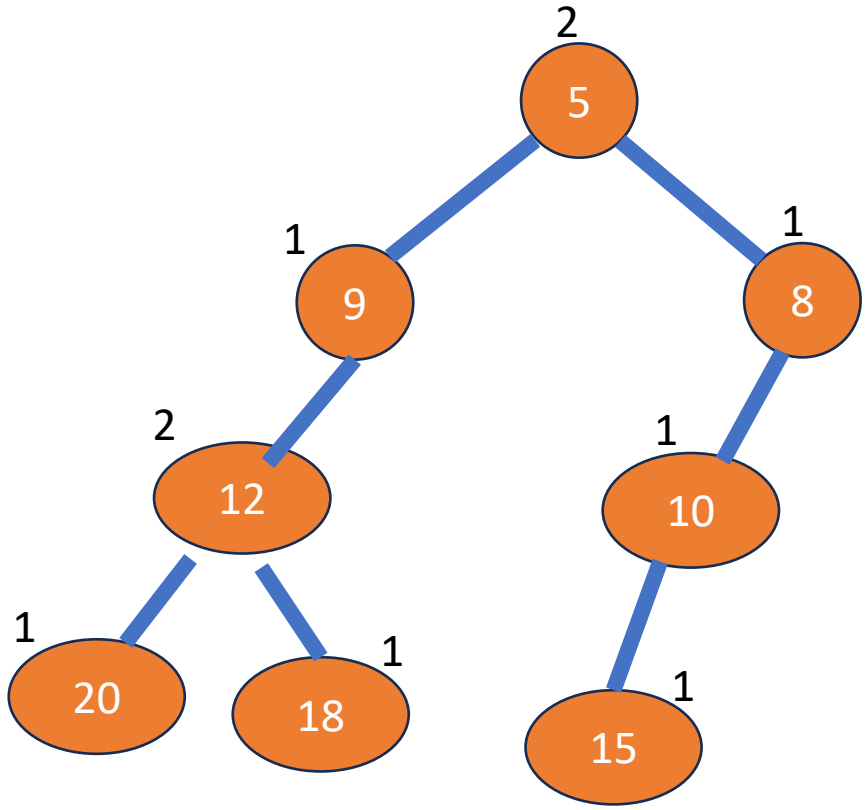
Height Biased Leftist Tree



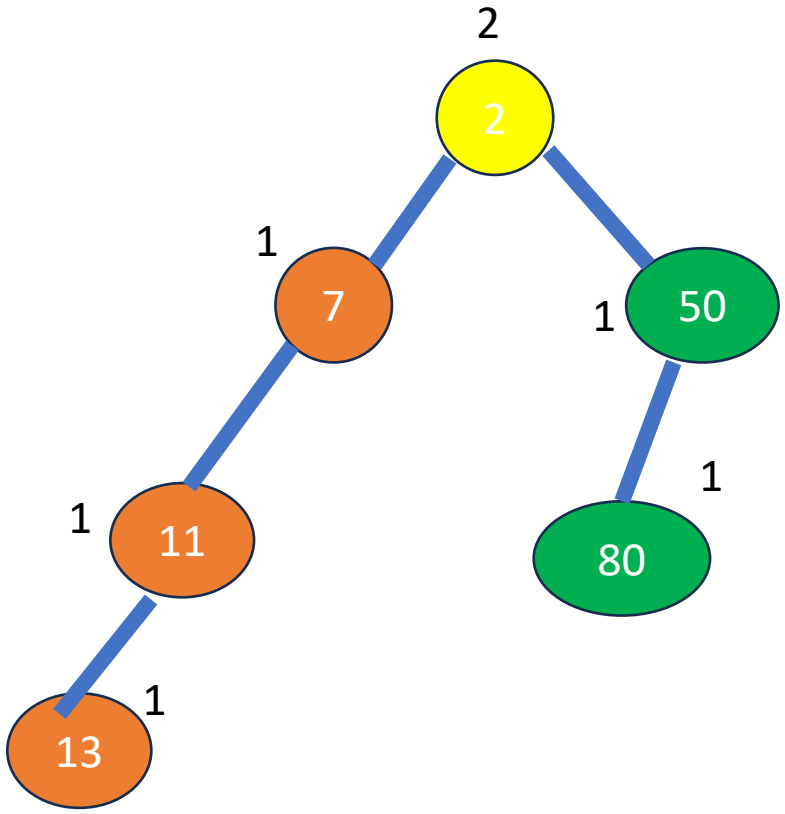


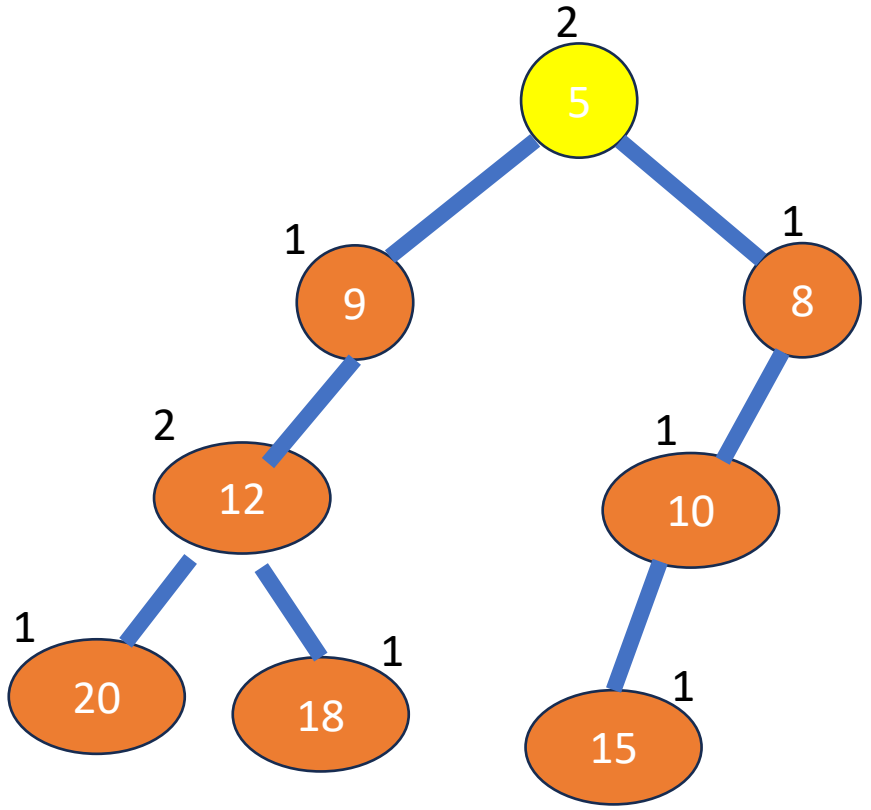
Min- Height Biased Leftist Tree





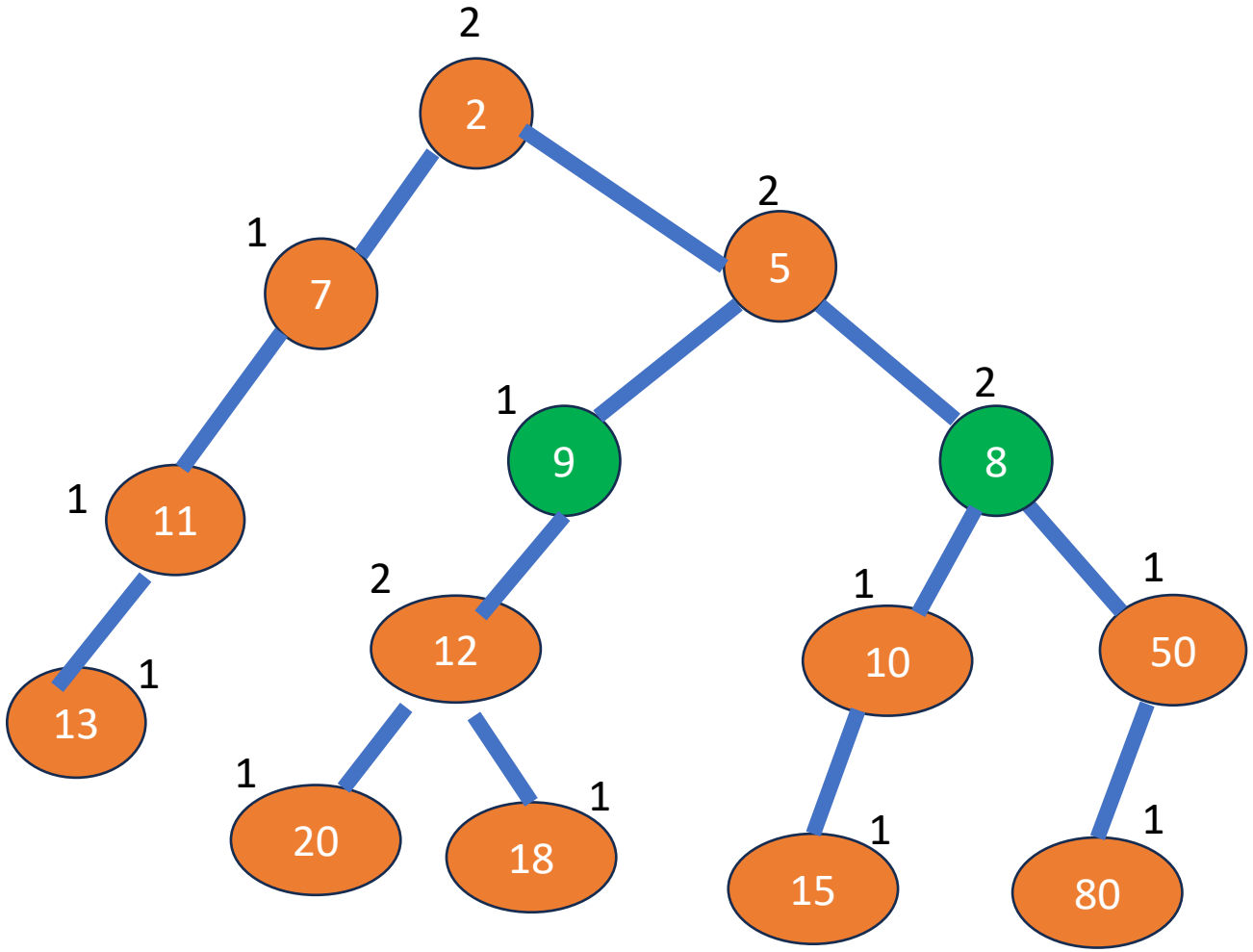
Min- Height Biased Leftist Tree





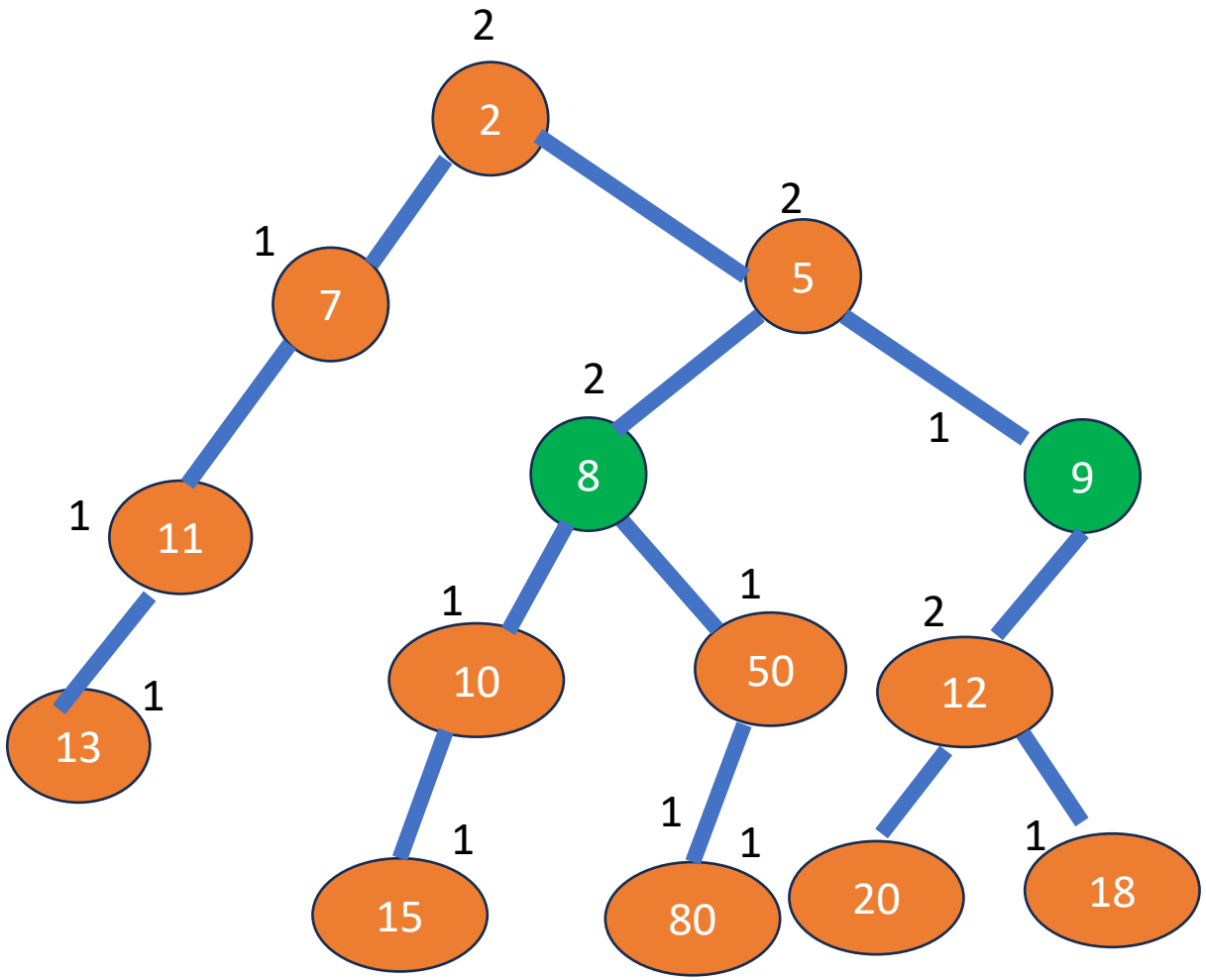
Compare root nodes 2 and 5, as 2<5 , 2 must be the root

Min- Height Biased Leftist Tree



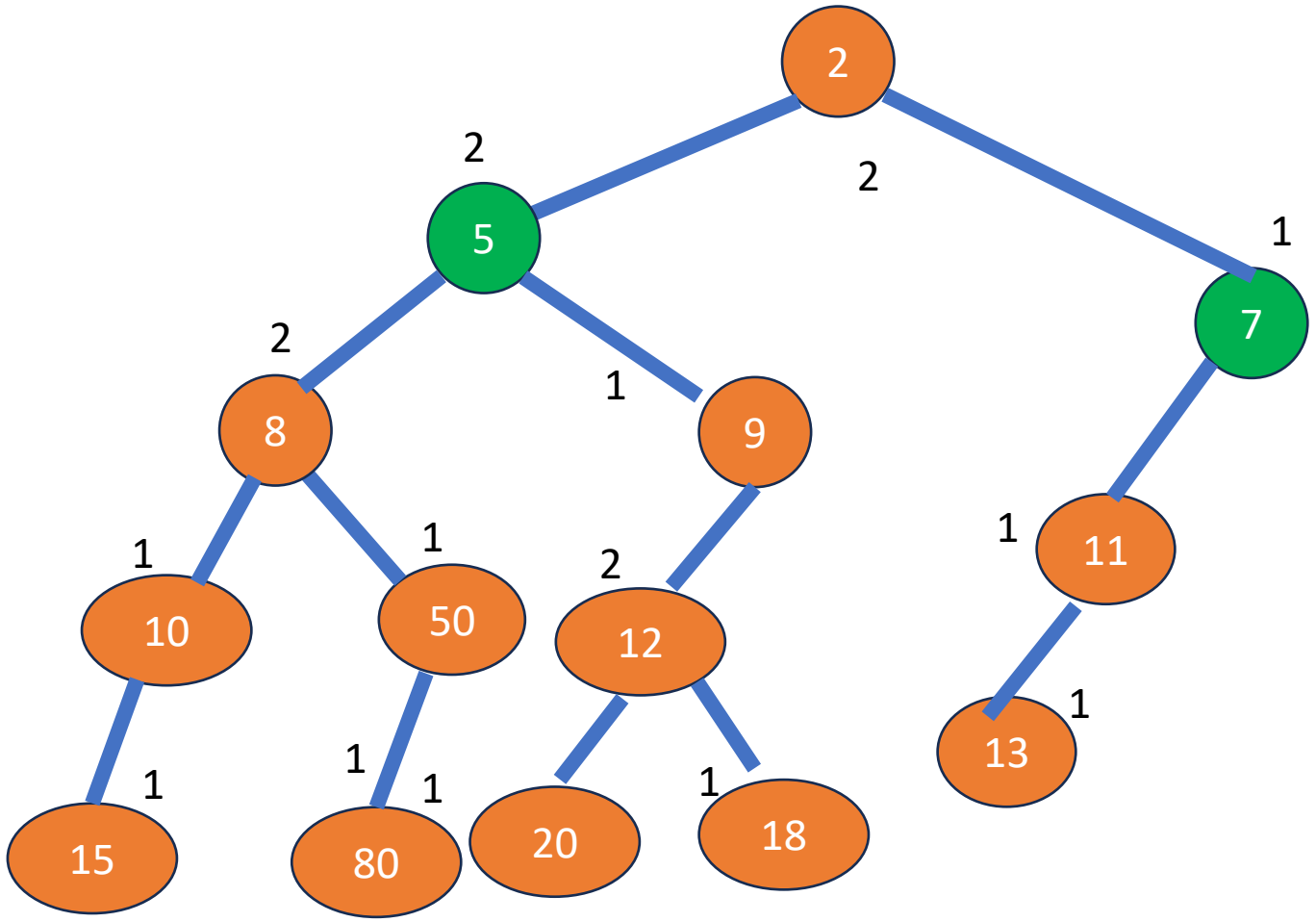
After adding , it can be observed that result is not leftist tree, for node 5, leftchild(s(5))<rightchild(s(5)) Swap left tree and right tree of node 5

Min- Height Biased Leftist Tree

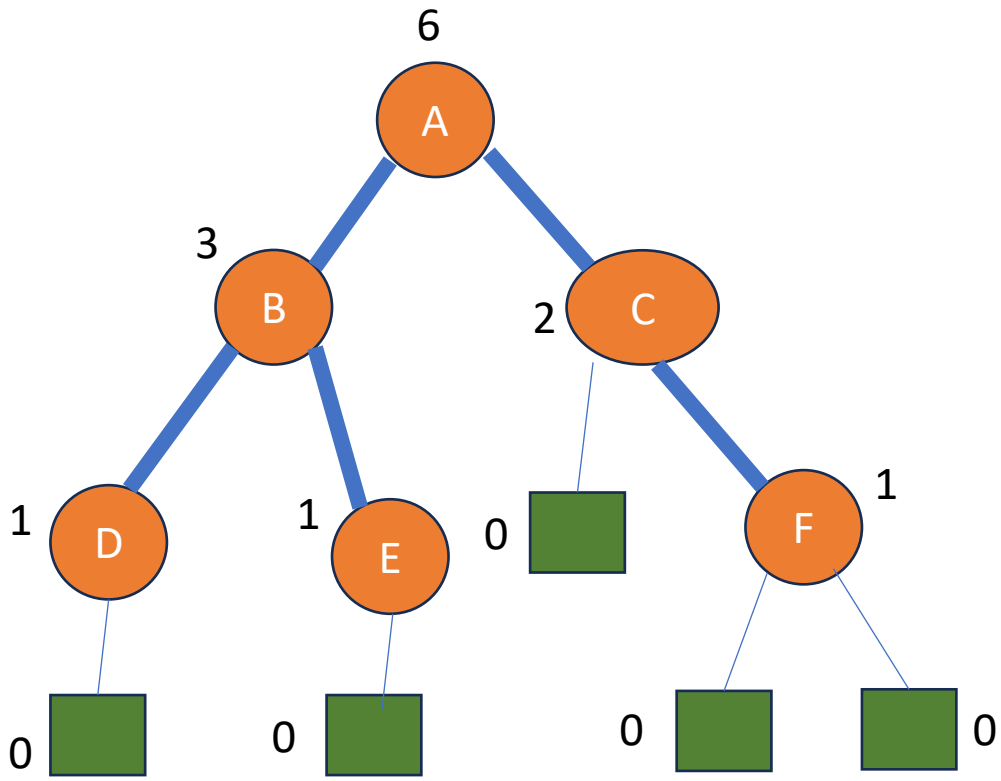


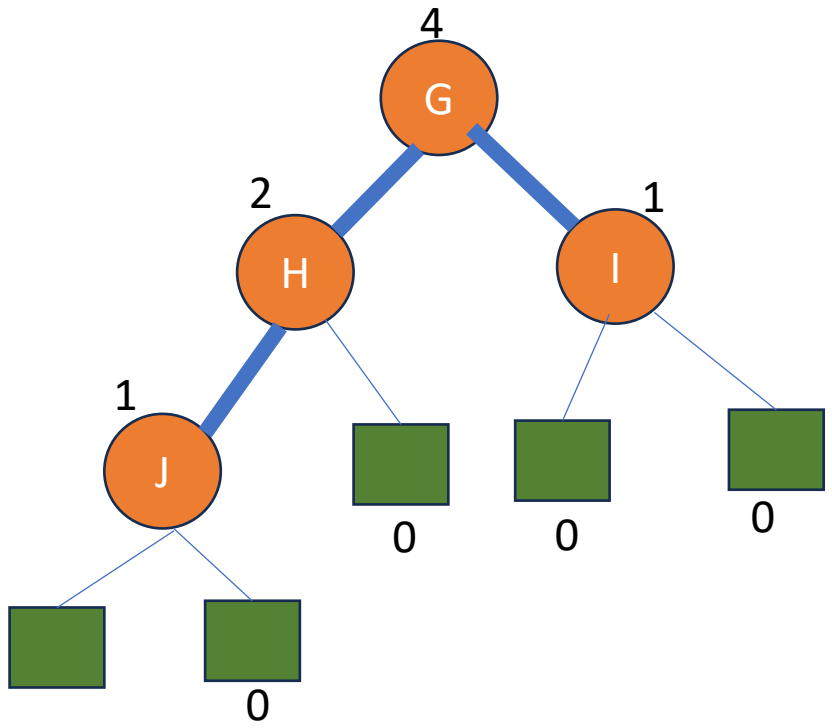
Still it needs swap as for node 2,
leftchild(s(2))<rightchild(s(2))
Swap left tree and right tree of node 2

Min- Height Biased Leftist Tree



Weight Biased Leftist Tree





0

If x is an external node, its weight(x) value is 0. If the x is an internal node, the value is =1+sum of weights of its children. Leftist property must also be satisfied. Only b is leftist

AVL Trees

- *An **AVL tree** defined as a self-balancing **Binary Search Tree** (BST) where the difference between heights of left and right subtrees for any node cannot be more than one**(balance factor).***

- The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

- The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

Balance factor

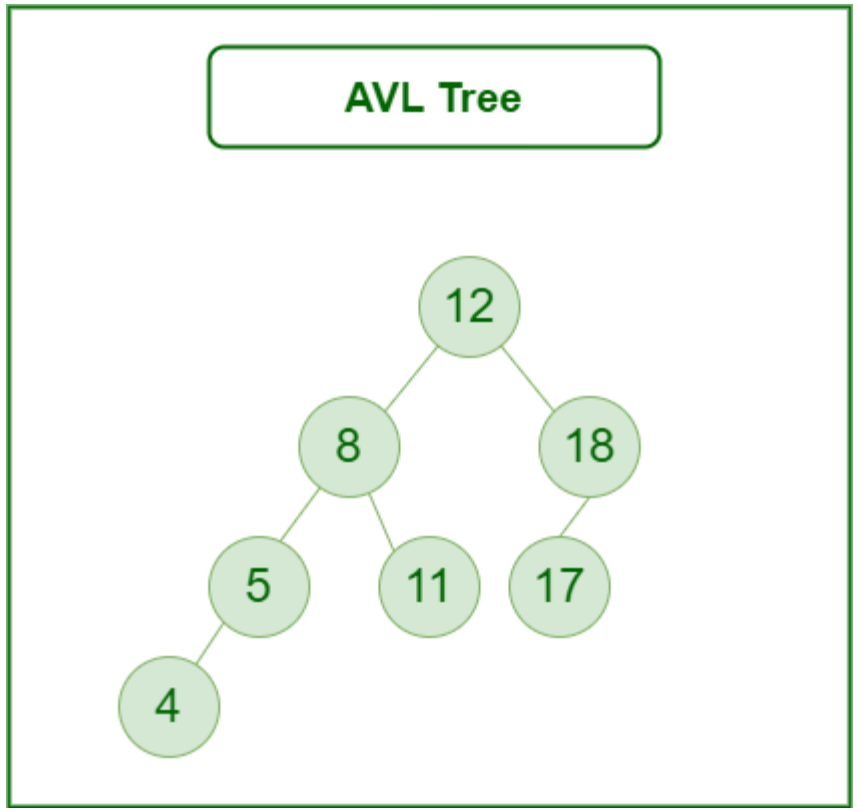
- *A Balance factor , BF(T) of a node T in a binary tree is defined to be hL-hR.*

hL-Height of Left subtree.

hR-Height of Right Subtree.

For any node T in an AVL tree,

BF(T)=-1,0,1

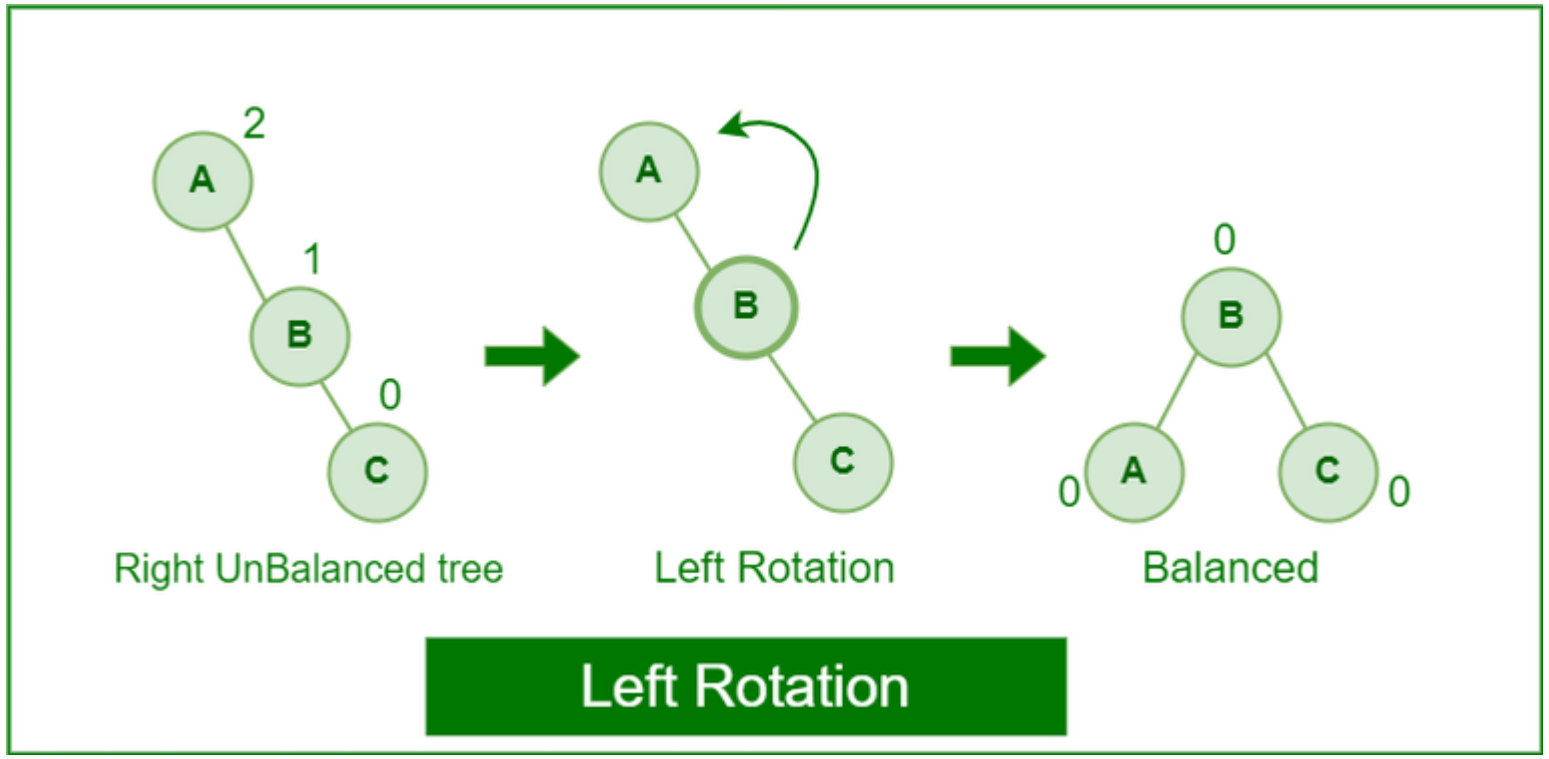


Rotating the subtrees in an AVL Tree:

- Left Rotation
- Right Rotation
- Left-Right Rotation
- Right-Left Rotation

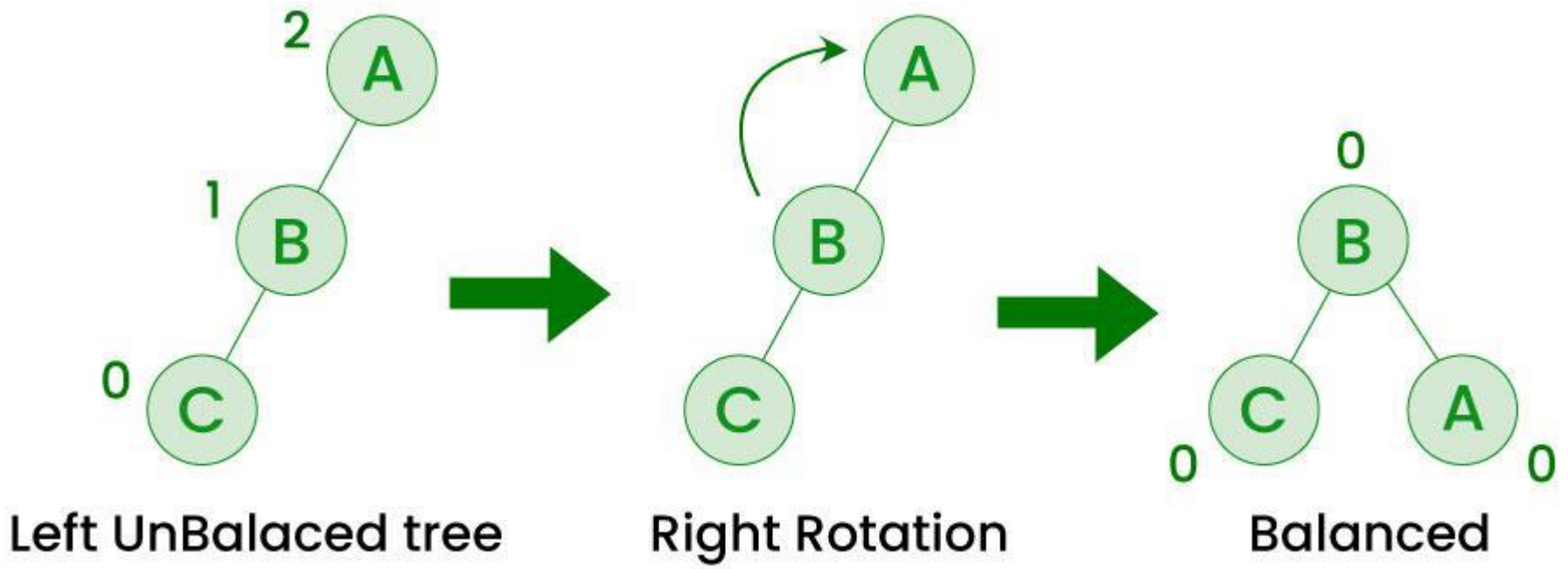
Left Rotation

- When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



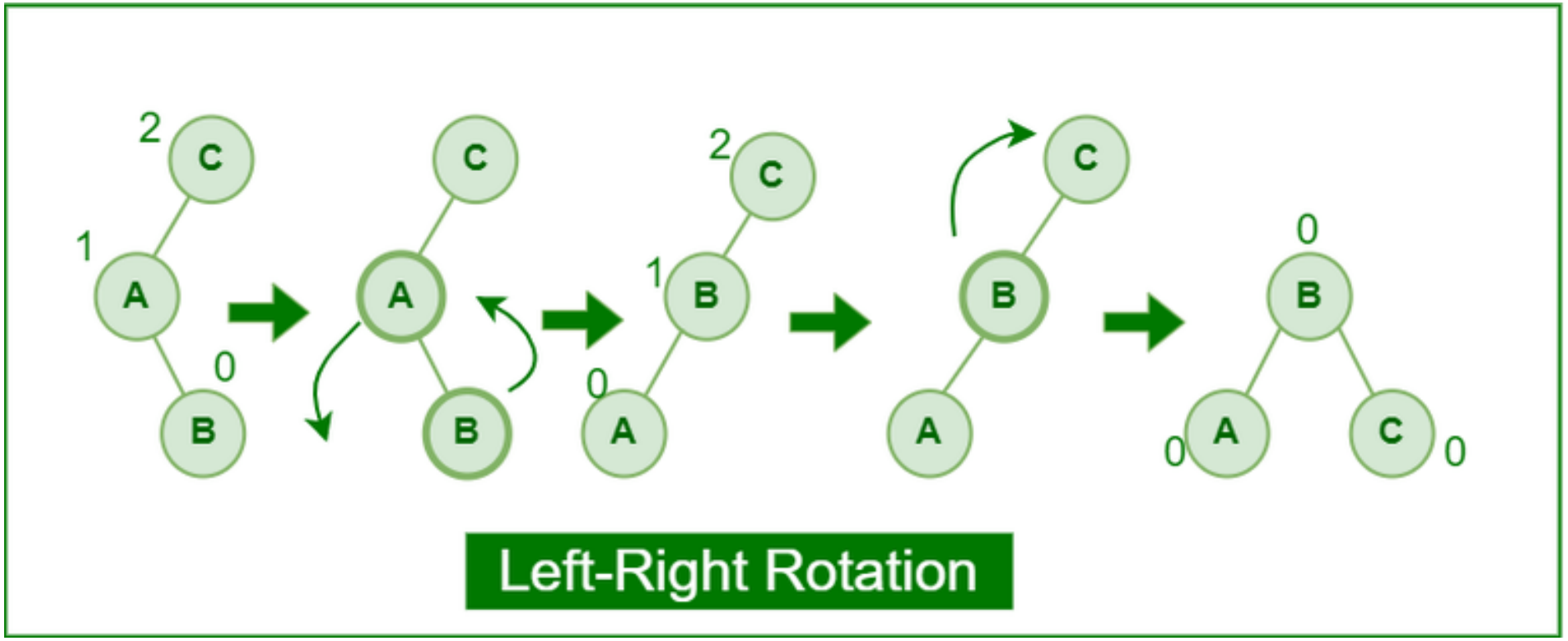
Right Rotation

- If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.

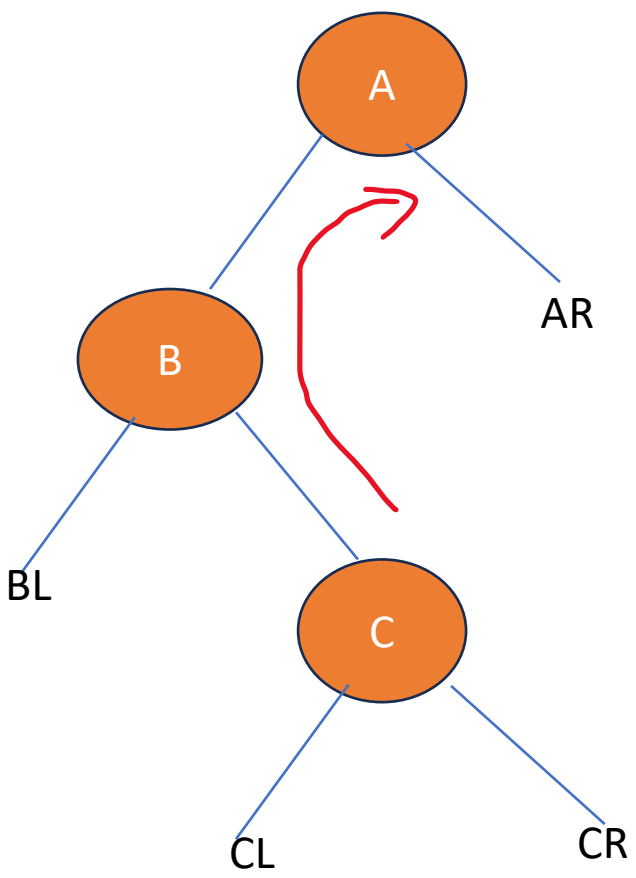


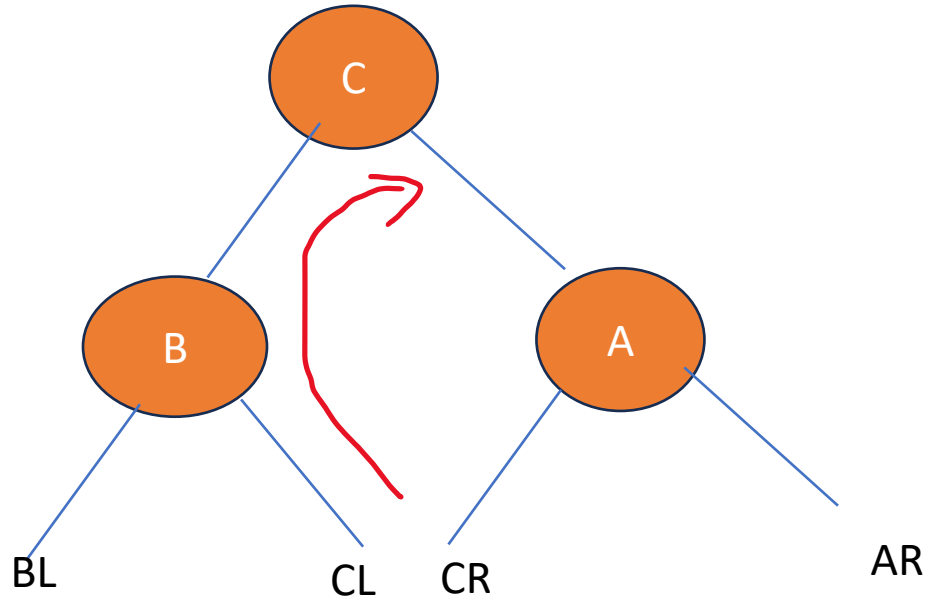
Left-Right Rotation

- A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



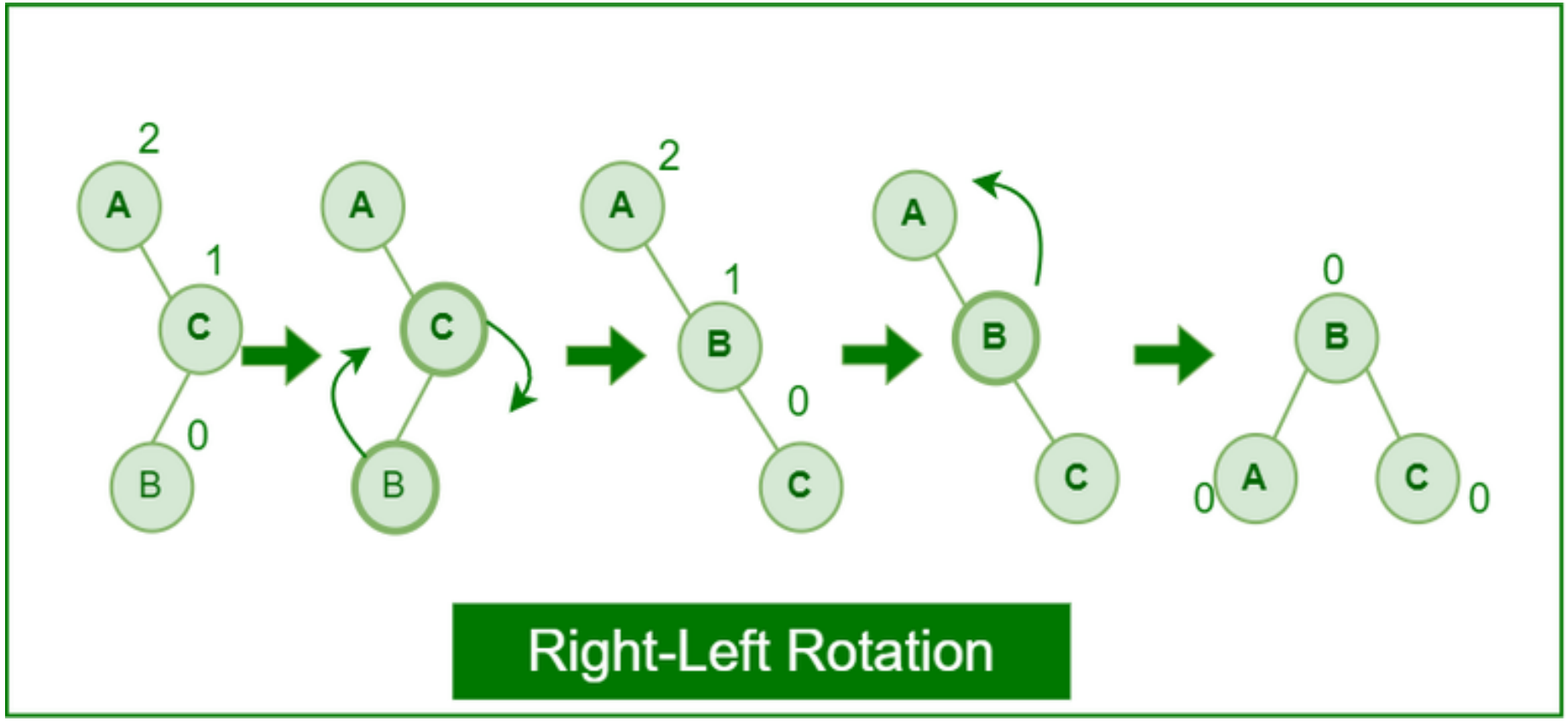
Left-Right Rotation – More Nodes



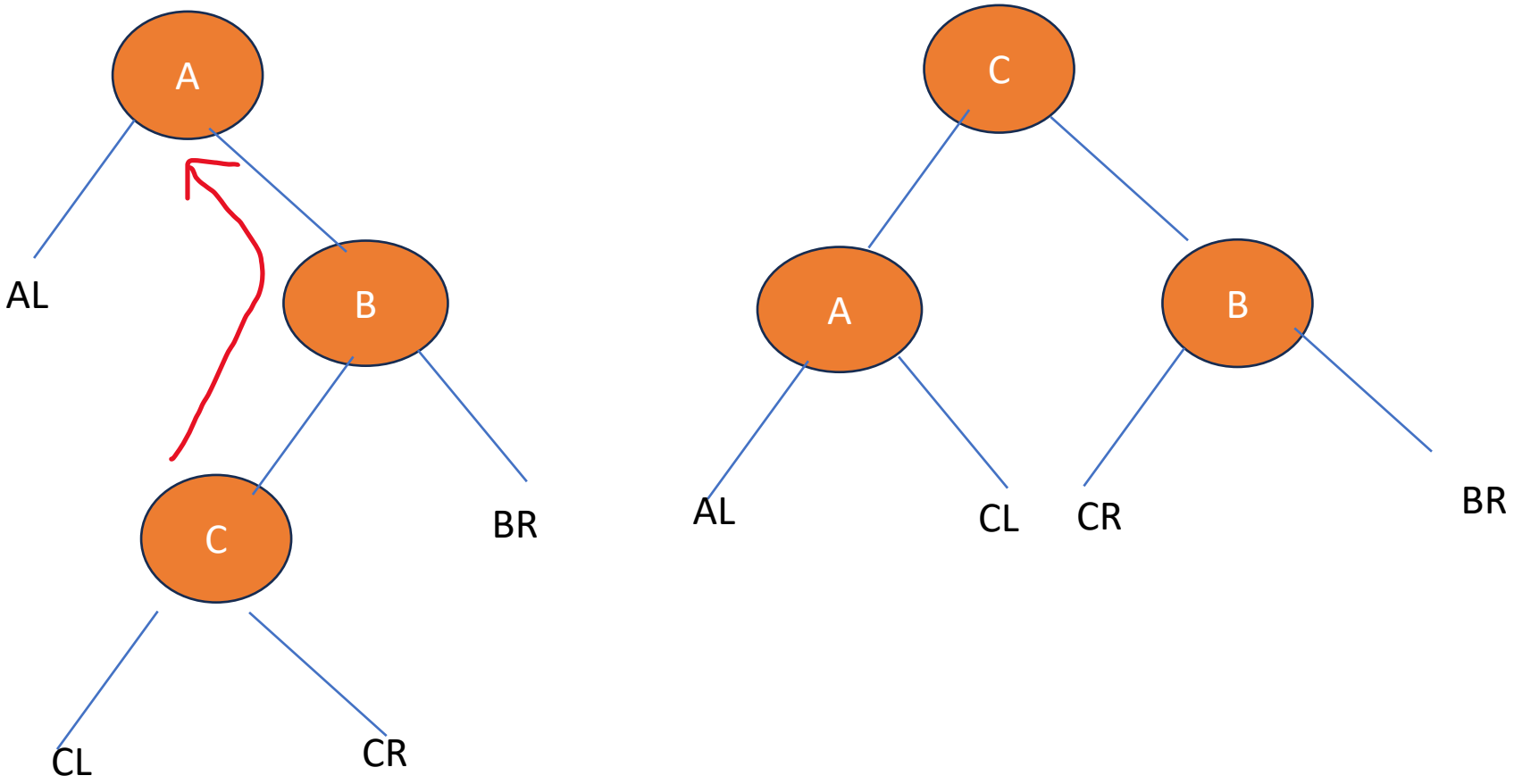


Right-Left Rotation

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.

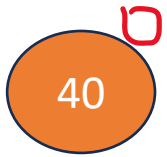


Left-Right Rotation – More Nodes



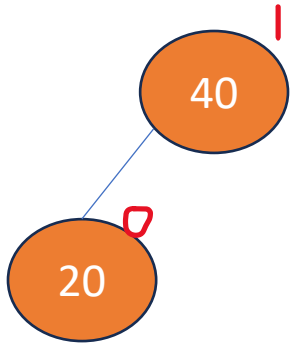
Construction of AVL Trees

40,20,10,25,30,22,50



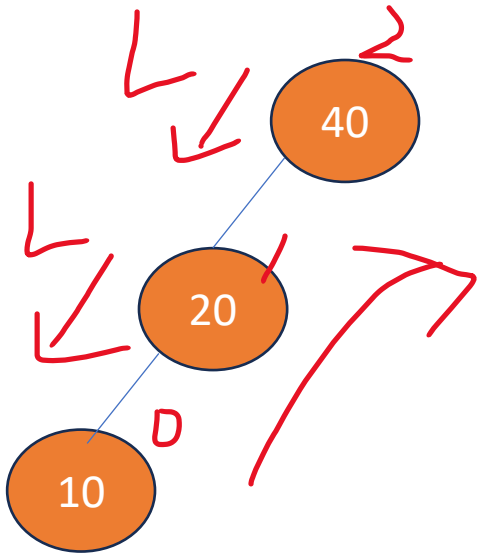
Construction of AVL Trees

40,20,10,25,30,22,50



Construction of AVL Trees

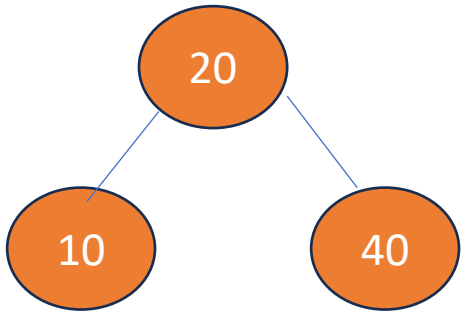
40,20,10,25,30,22,50



Imbalance , Apply LL Rotation

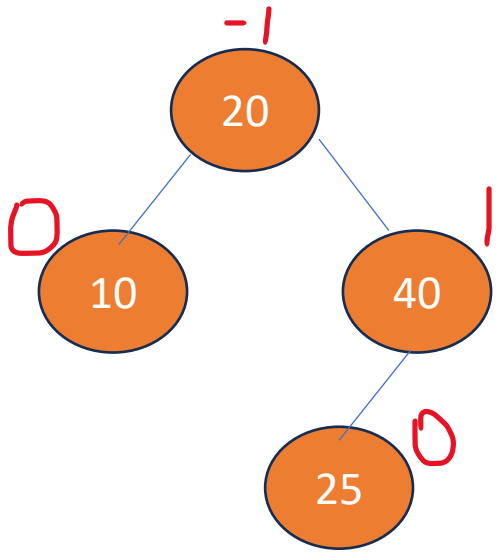
Construction of AVL Trees

40,20,10,25,30,22,50



Construction of AVL Trees

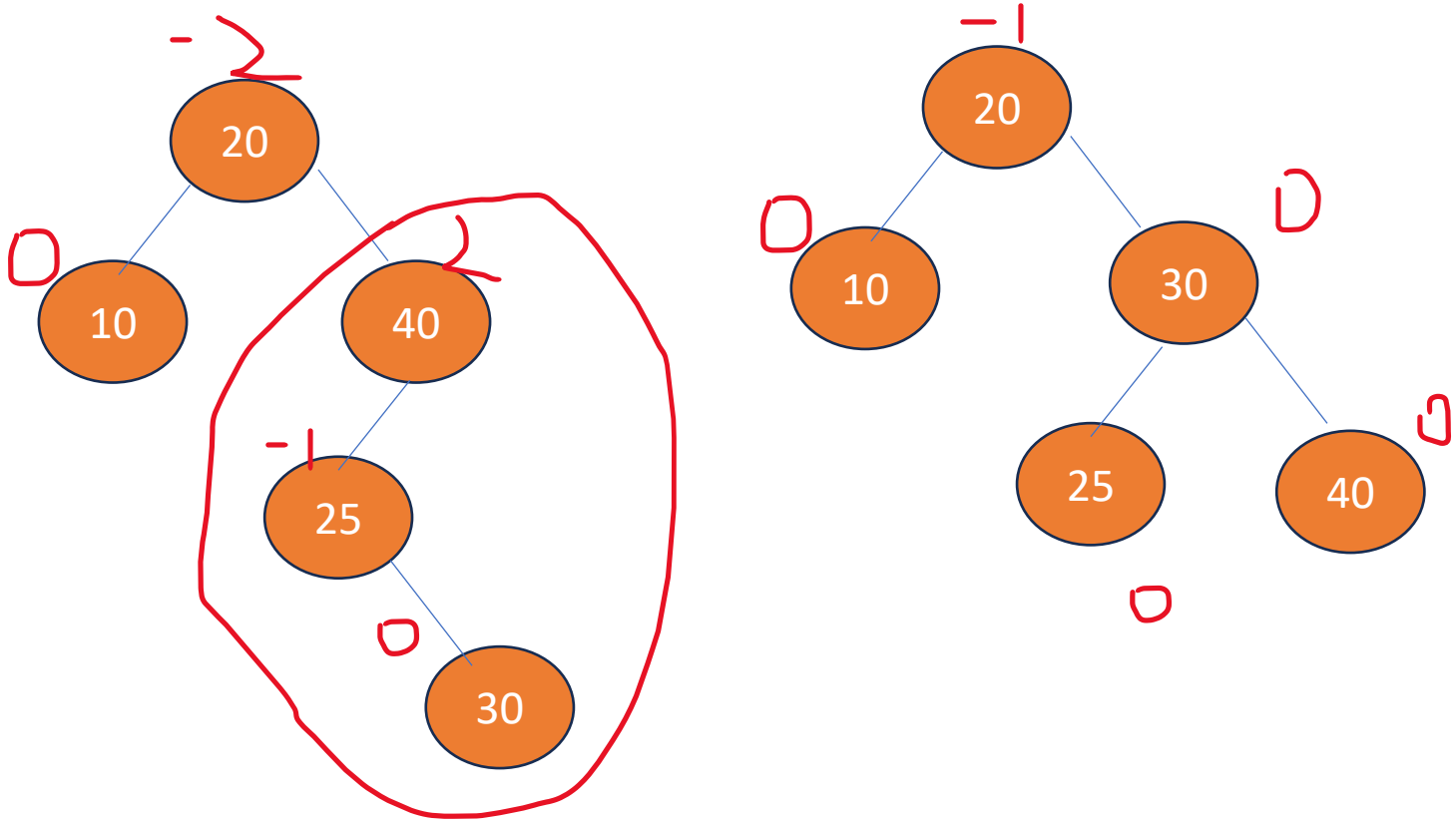
40,20,10,25,30,22,50



Construction of AVL Trees

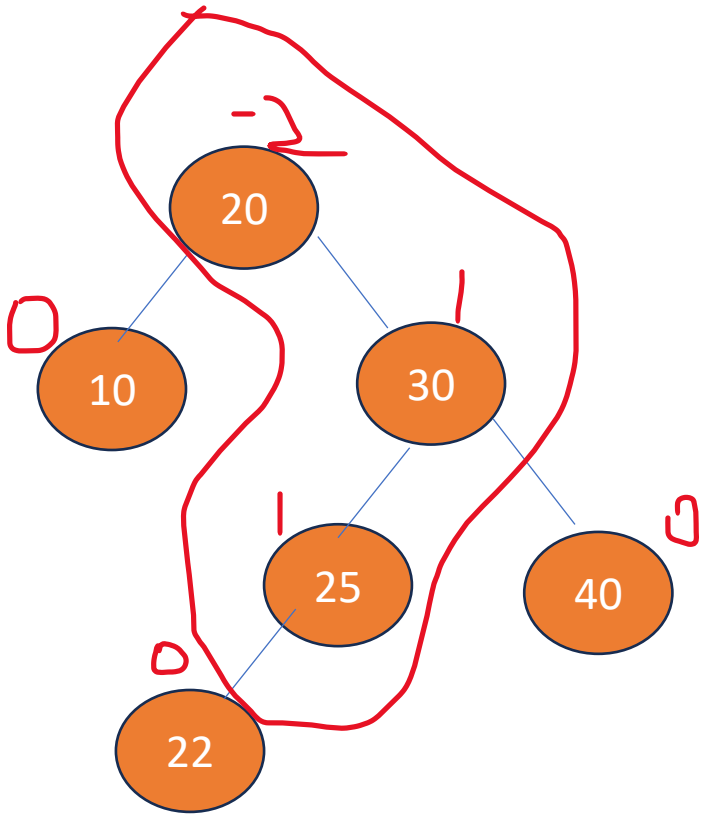
40,20,10,25,30,22,50

LR



Construction of AVL Trees

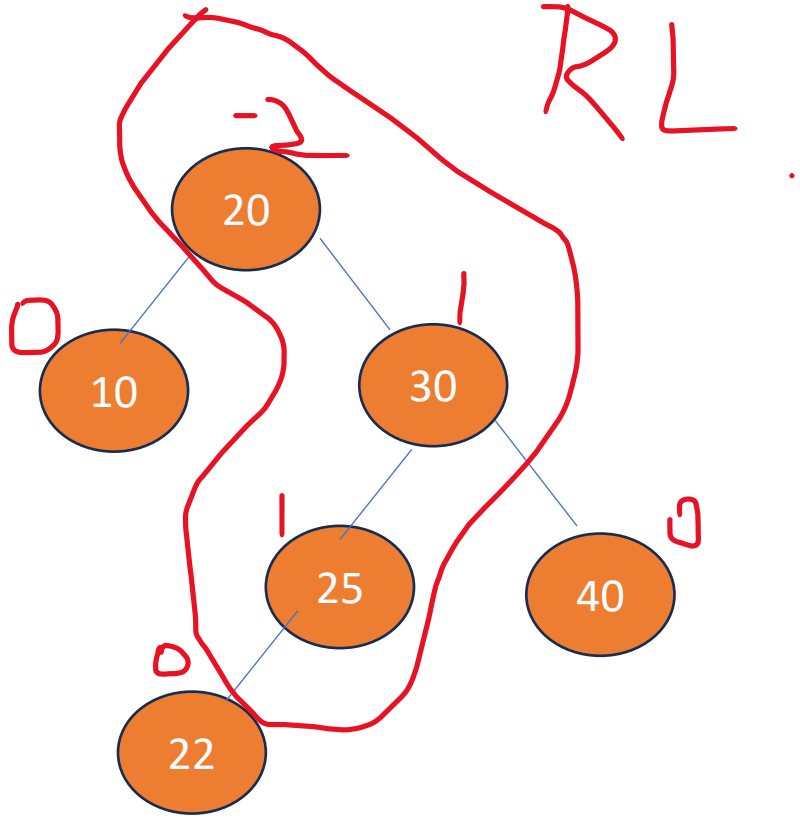
40,20,10,25,30,22,50

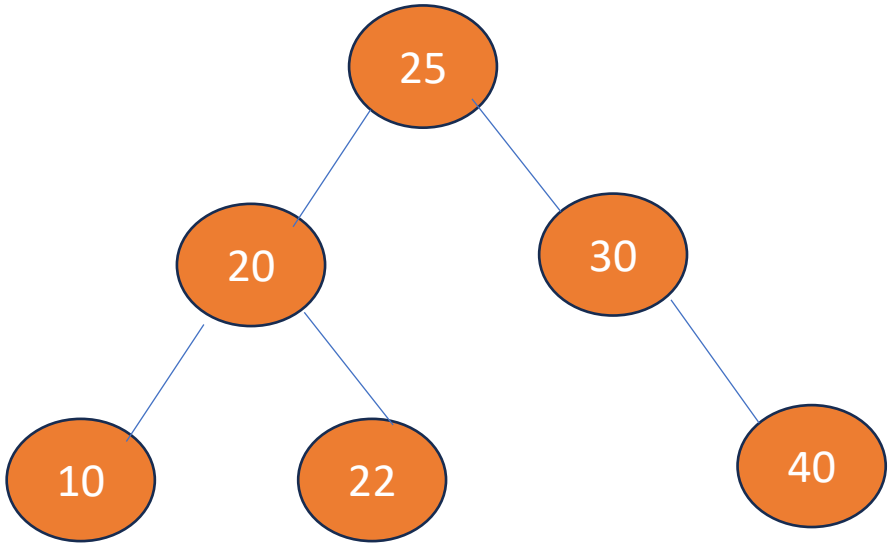


RL

Construction of AVL Trees

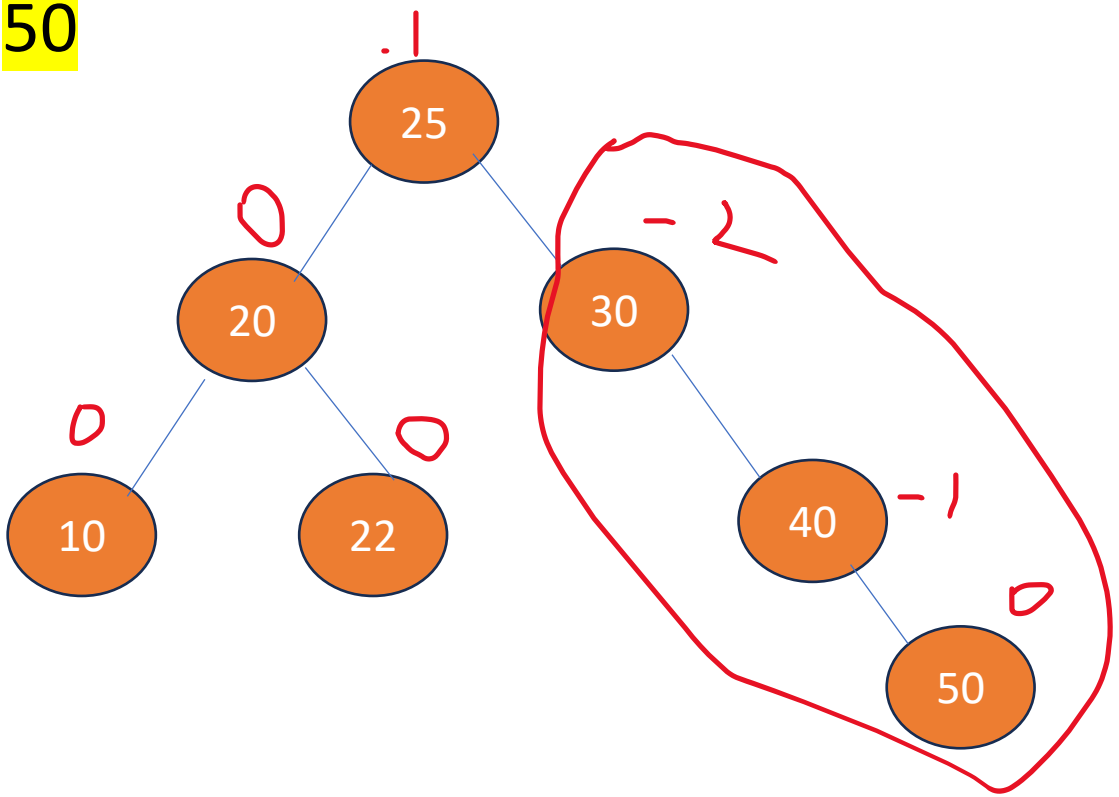
40,20,10,25,30,22,50





Construction of AVL Trees

40,20,10,25,30,22,50



Construction of AVL Trees

40,20,10,25,30,22,50