Additional Questions and Answers related to Lecture 5

M.G.Noel A.S Fernando (PhD)

Professor

NSBM/Dept. of Information Systems Engineering, UCSC

Question 1

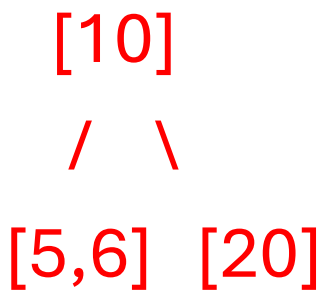
- Insert the following values in order :10, 20, 5, 6, 12, 30, 7, 17 (order is 3-4)
- We want to create a **3–4 B-tree** (also called a 2–3–4 tree) using the data set:
- 10, 20, 5, 6, 12, 30, 7, 17
- A 3–4 tree node can have **2, 3, or 4 children**, and **1–3 keys**.
- **Step 1**: Insert 10
- Tree is empty → 10 becomes the root.
- [10]

Question 1 cont..

- Step 2: Insert 20
- Root has 1 key → can accommodate another → insert 20.
- [10, 20]
- Step 3: Insert 5
- Root has 2 keys → can accommodate another → insert 5 in order.
- [5, 10, 20]

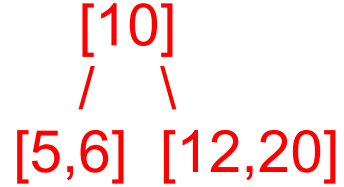
Question 1 cont..

- Step 4: Insert 6
  - Root has 3 keys → needs to **split** when inserting a 4th key.
  - Order keys: 5, 6, 10, 20
  - Middle key **10** moves up → becomes new root
  - Left child: [5, 6], Right child: [20]

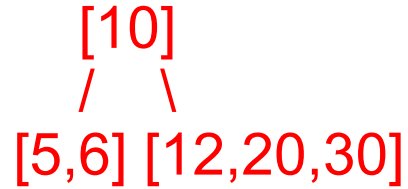


Question 1 Cont..

- Step 5: Insert 12
  12 > 10 → goes to right child [20]
  Right child has 1 key → can accommodate → insert 12 in order → [12, 20]

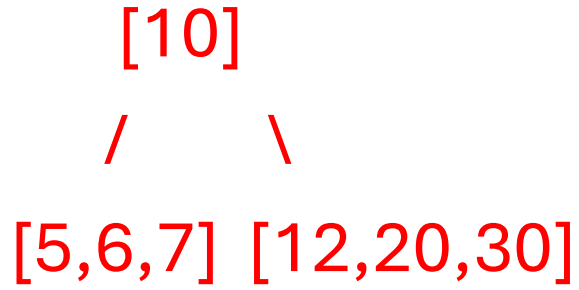


- Step 6:Insert 30
  - 30 > 10 → goes to right child [12, 20]
  - Right child has 2 keys → can accommodate → insert 30 → [12, 20, 30]



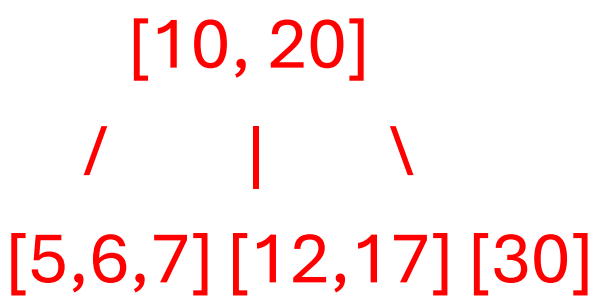
Question 1 Cont..

- Step 7: Insert 7
  - 7 < 10 → goes to left child [5,6]
  - Left child has 2 keys → can accommodate → insert 7 → [5,6,7]



Question 1 Cont..

- Step 8: Insert 17
  - 17 > 10 → goes to right child [12,20,30]
  - Right child has 3 keys → will **split**
  - Order: 12,17,20,30 → middle key 20 moves up to root
  - Root [10] becomes [10,20]
  - Children: Left → [12,17], Right → [30]
  - Left child remains [5,6,7]



Question 2

- Insert the following numbers into an initially empty B-tree to build a B-tree of order(m) =4
- [5,3,21,9,13,22,7,10,11,14,8,16]

Answer to the question 2

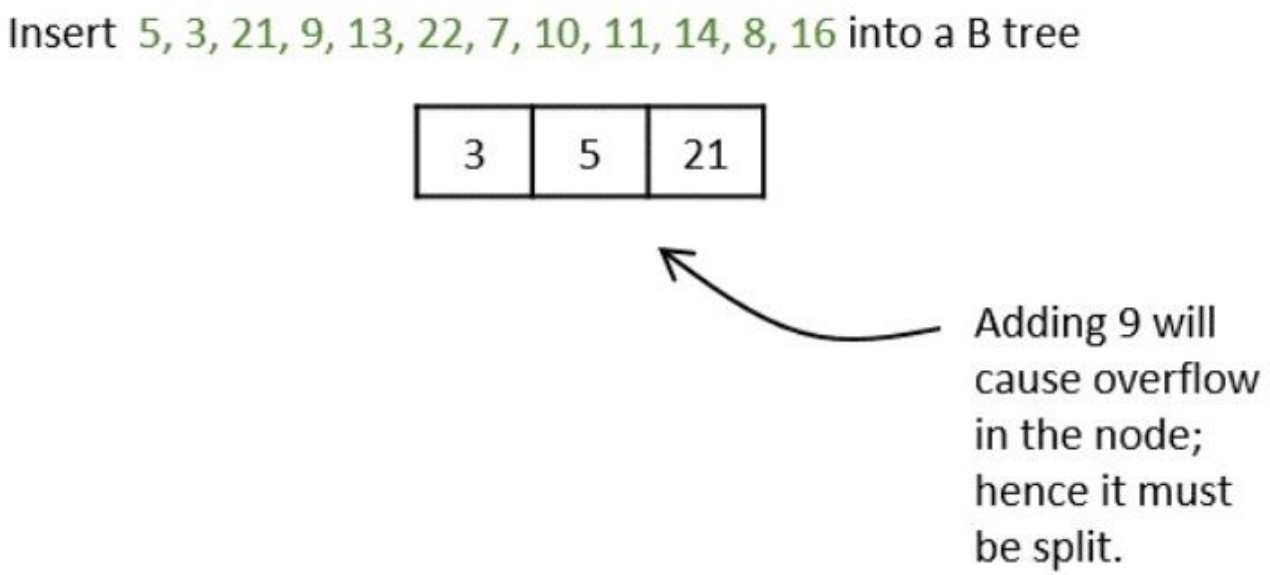
- **Step 1** – Calculate the maximum (m–1) and, minimum (⌈m/2⌉–1) number of keys a node can hold, where m is denoted by the order of the B Tree

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

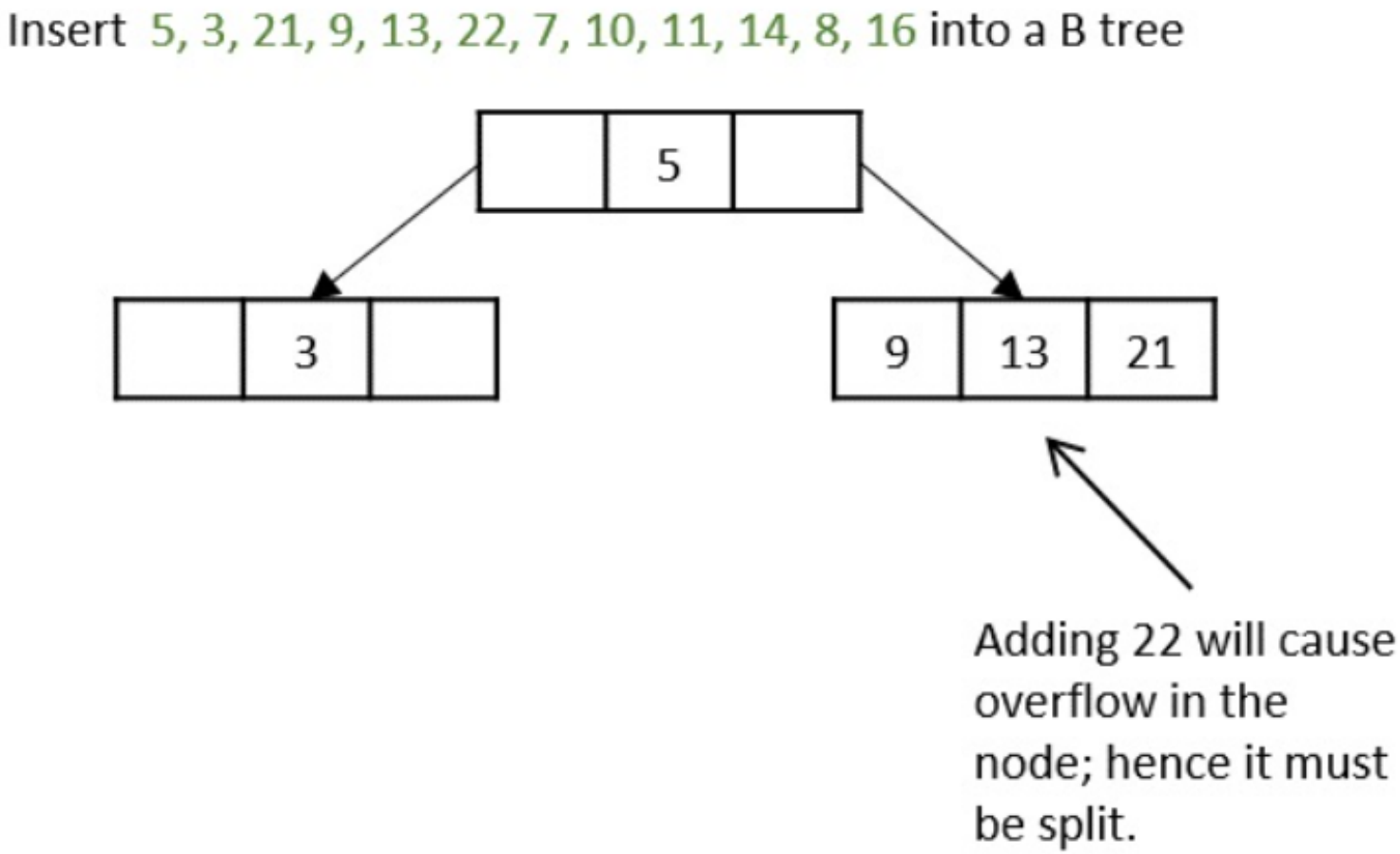
- Order (m) = 4
- Maximum Keys (m – 1) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil$) – 1 = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

Answer to the question Cont....

- **Step 2** – The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children.

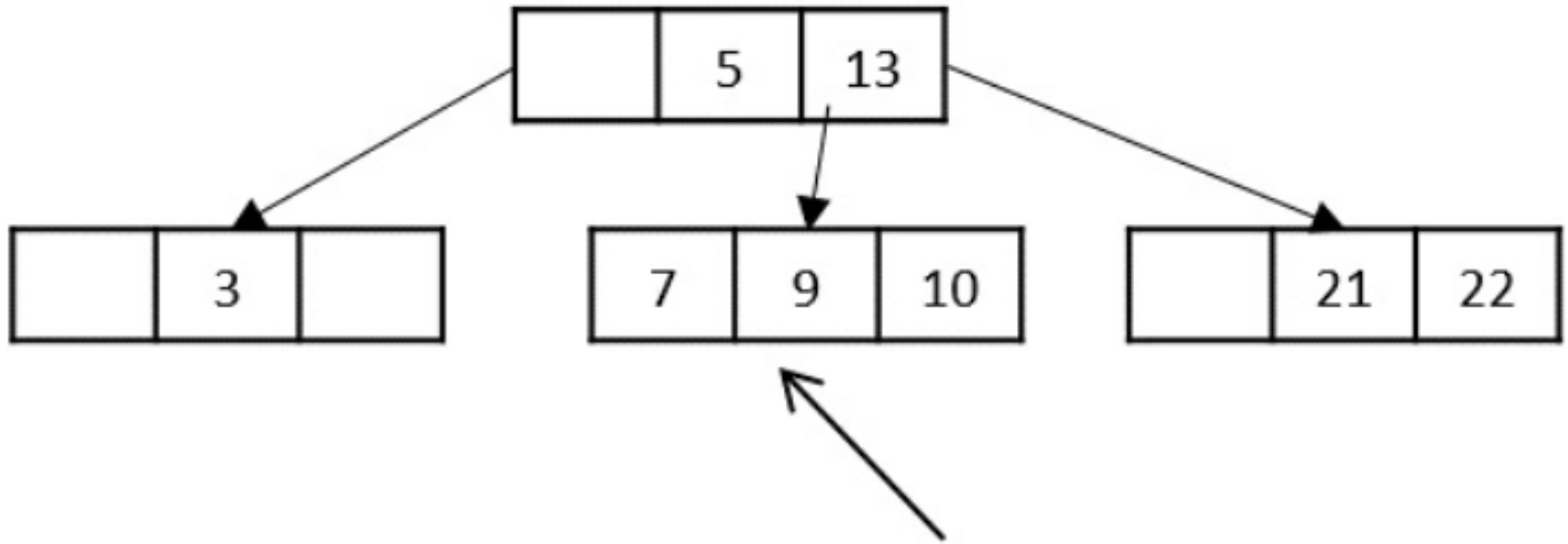


Step 3 – All the leaf nodes must be on the same level.

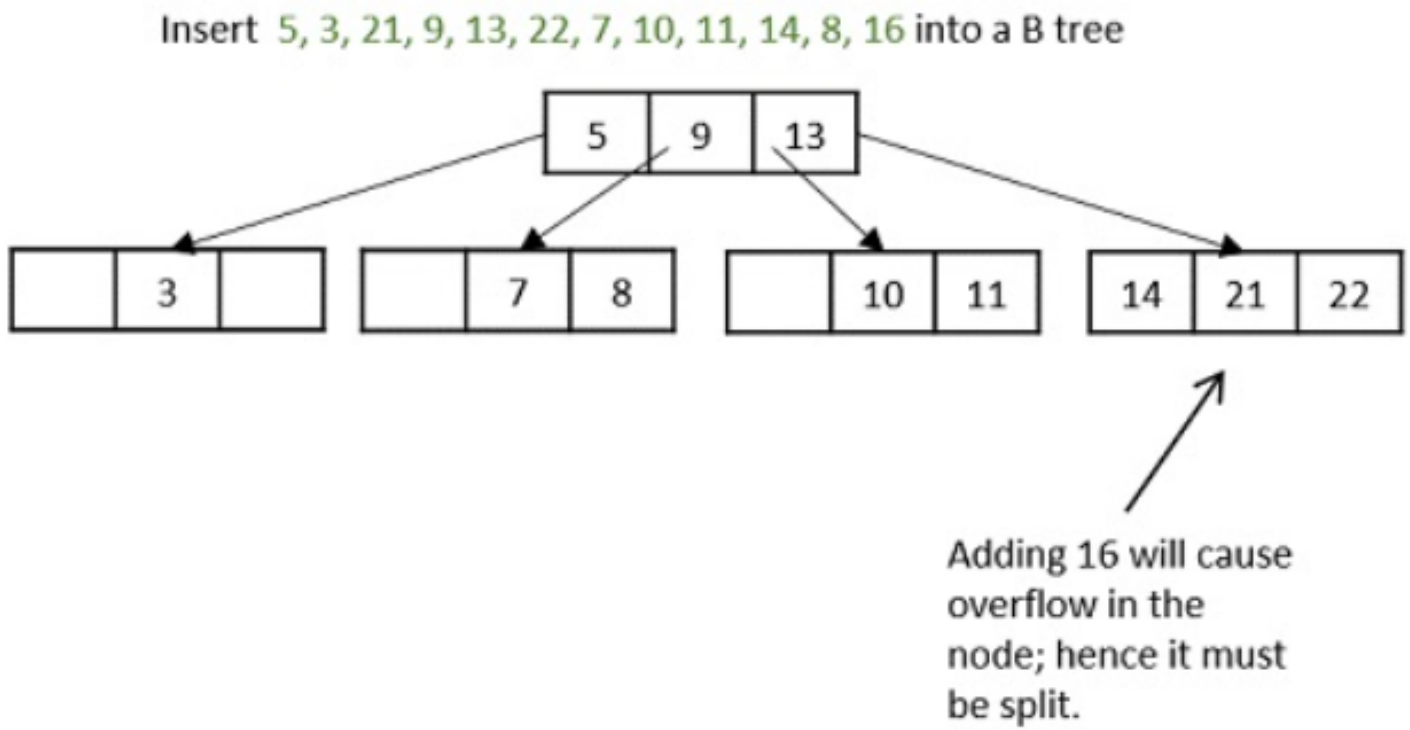


The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

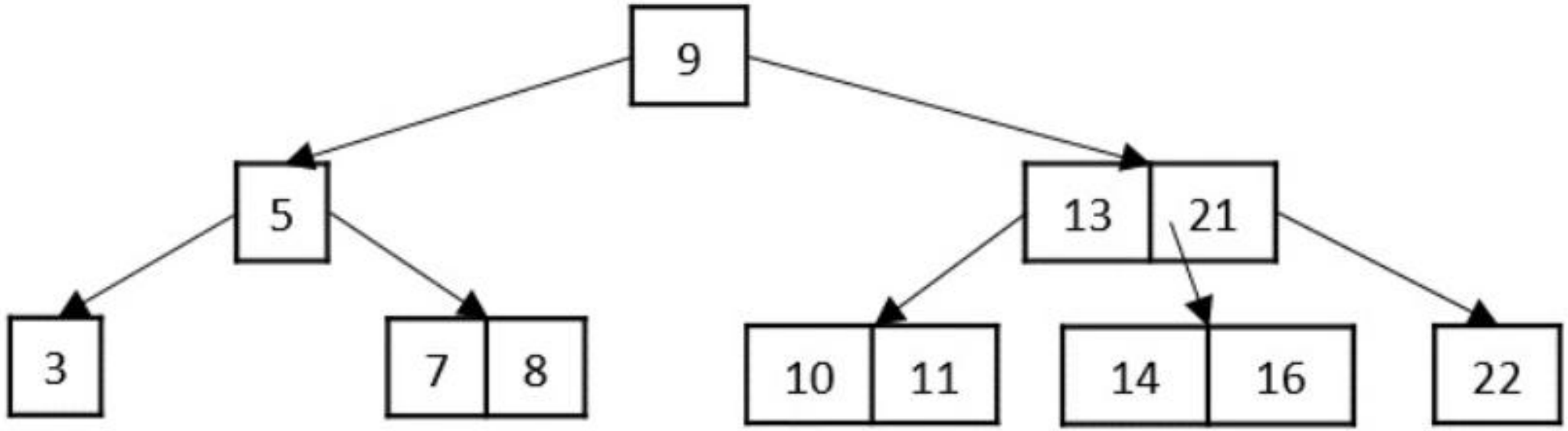


Adding 11 will cause overflow in the node; hence it must be split.



While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.

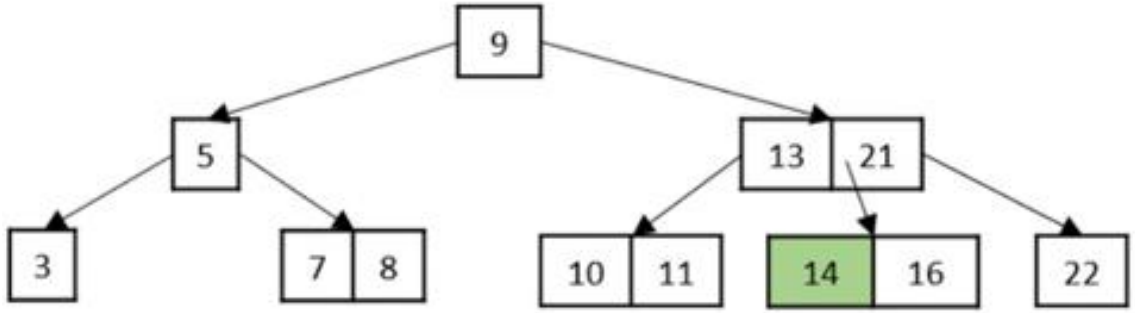
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



The final B tree after inserting all the elements is achieved.

Question 3

- Consider the following Tree



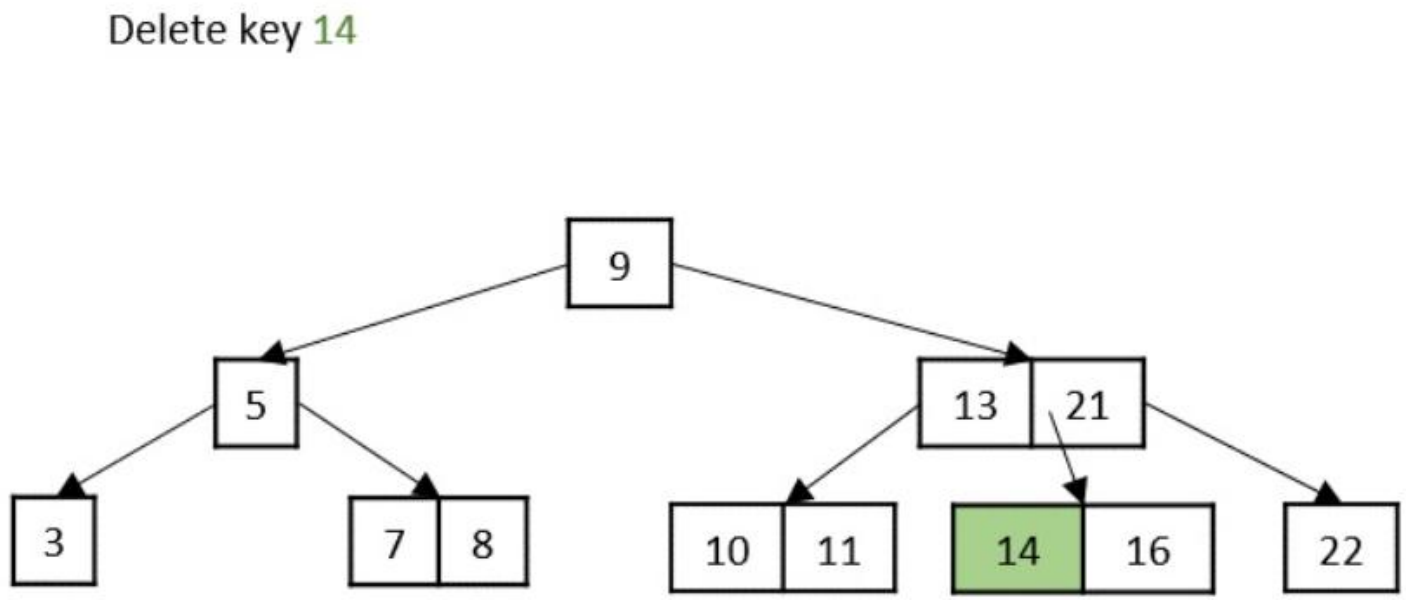
Perform the following operations:

1) Delete node 14
2) Delete node 3
3) Delete node 13
4) Delete node 5

- **Note:** All deletions should be performed on the original tree.

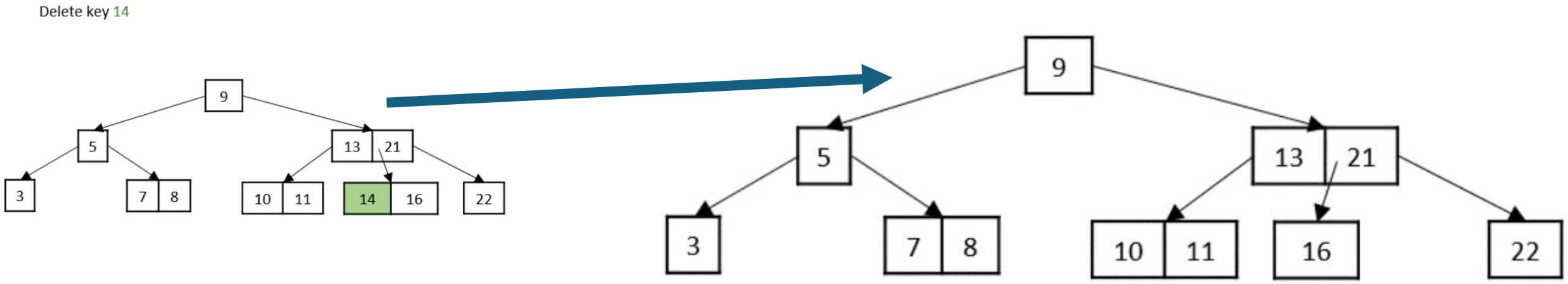
Deletion operation

1 – If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.



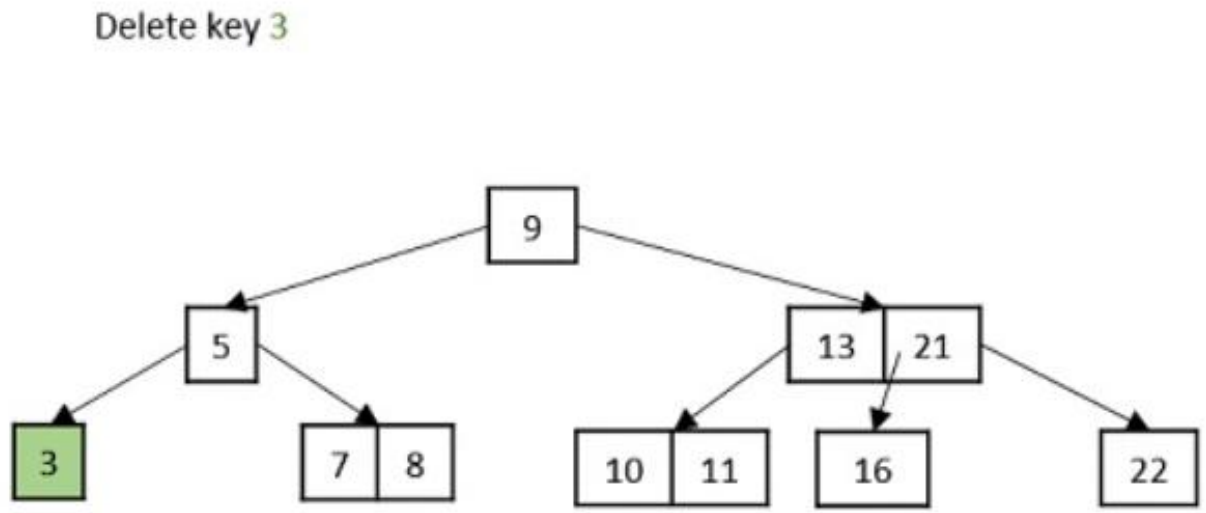
Deletion operation

1) – If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.



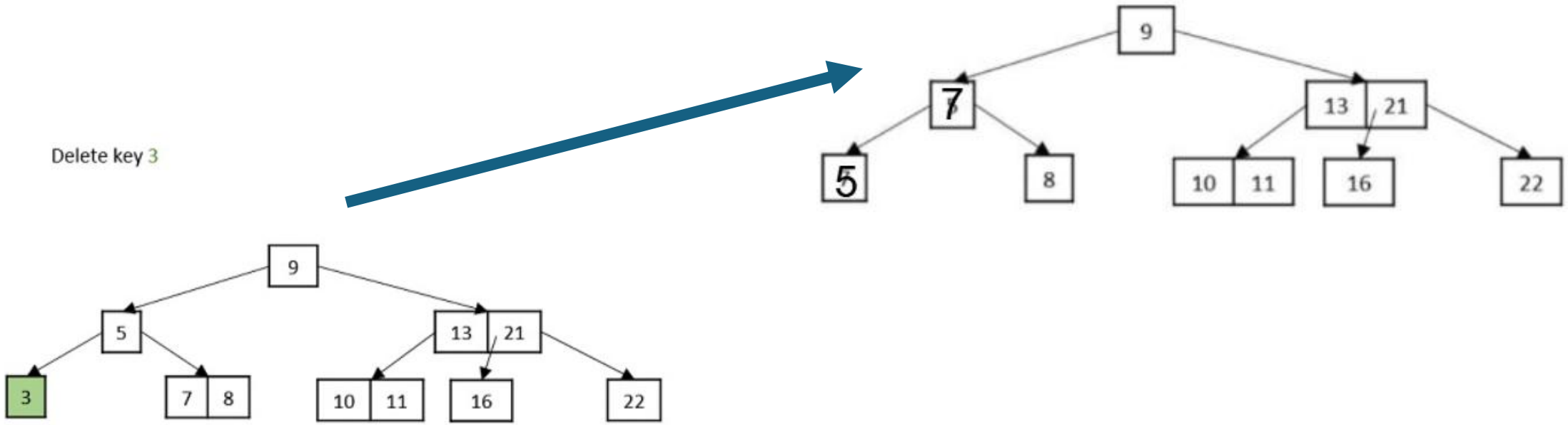
Deletion operation (cont..)

2) – If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling.



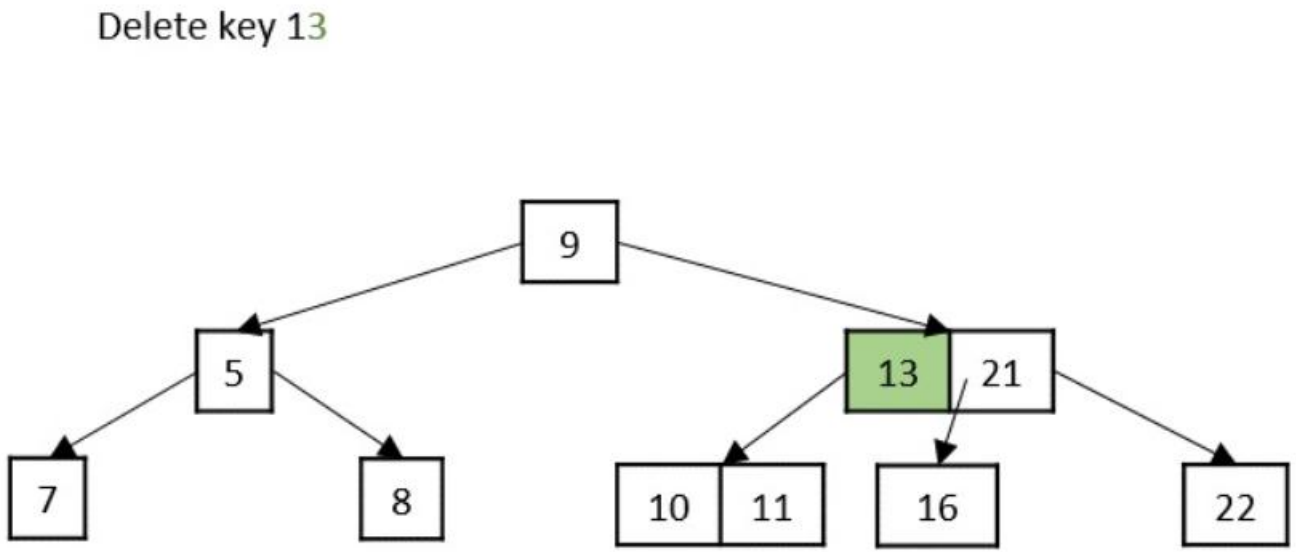
Deletion operation (cont..)

2)



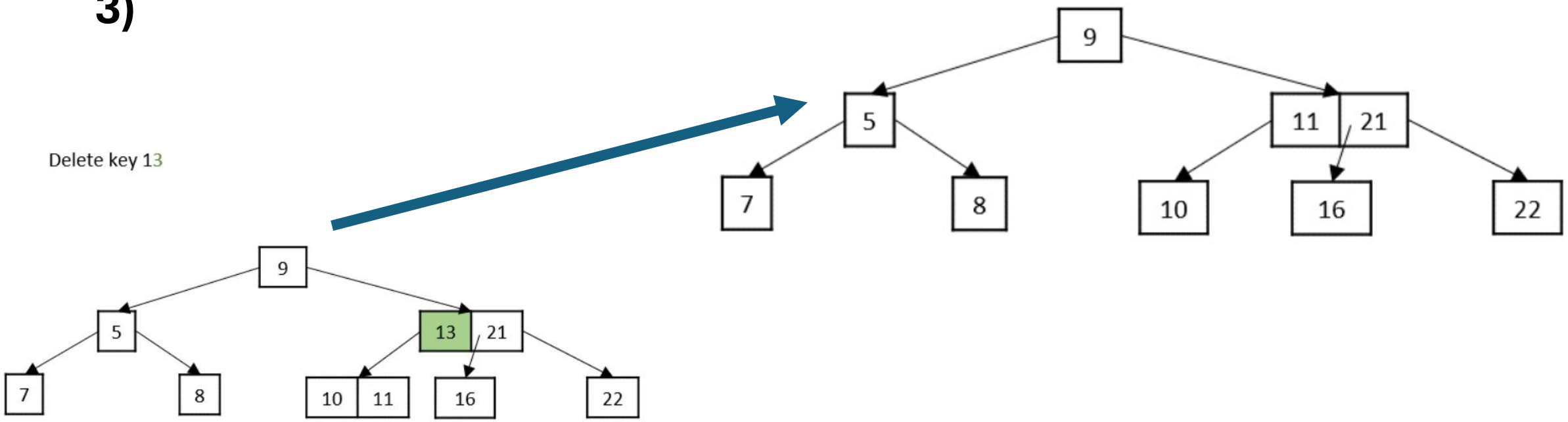
Deletion operation (cont..)

3) If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, there merged together.



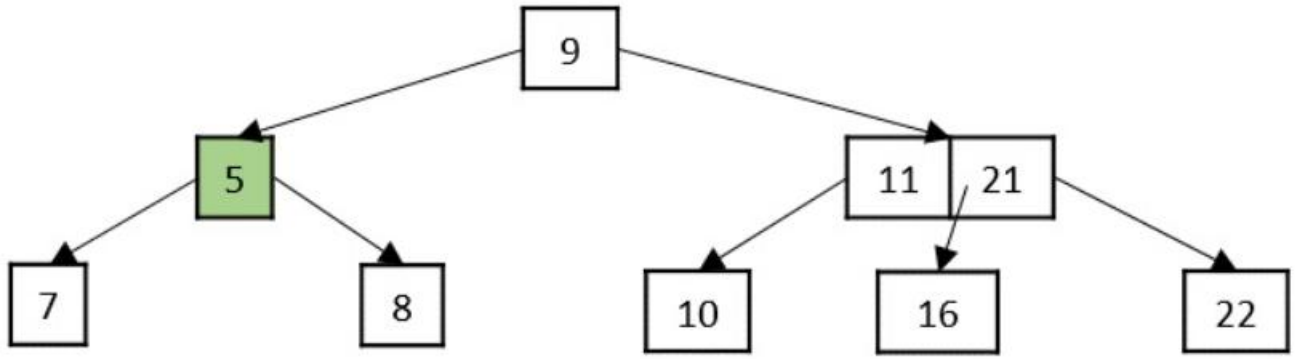
Deletion operation (cont..)

3)



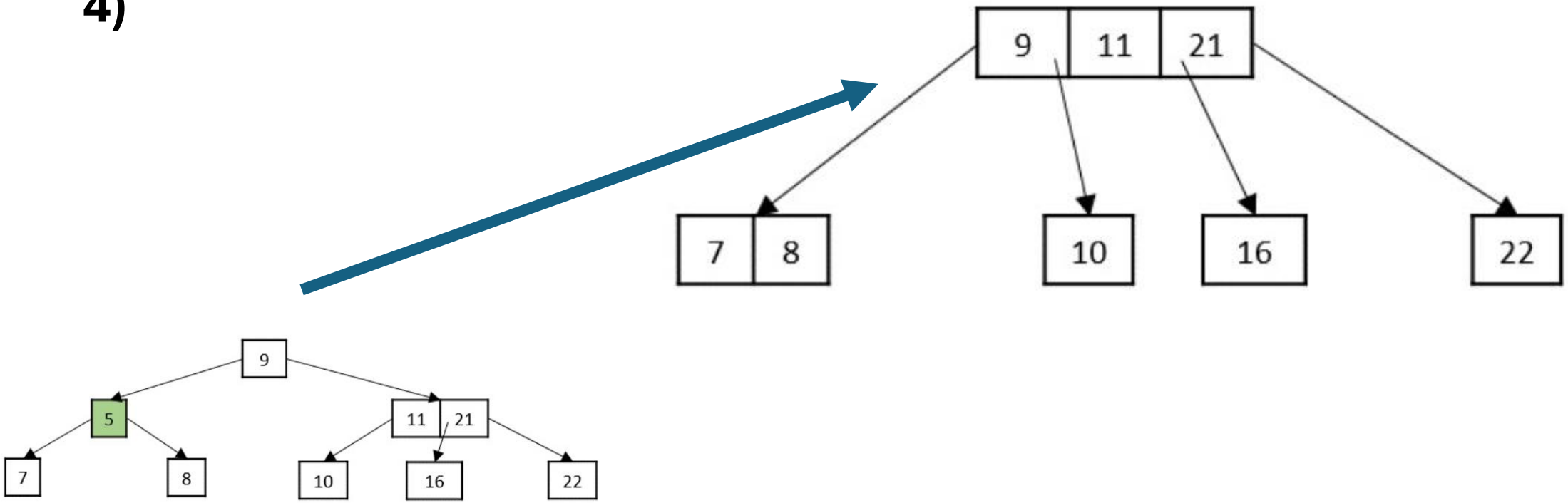
Deletion operation (cont..)

4)– If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent.



Deletion operation (cont..)

4)

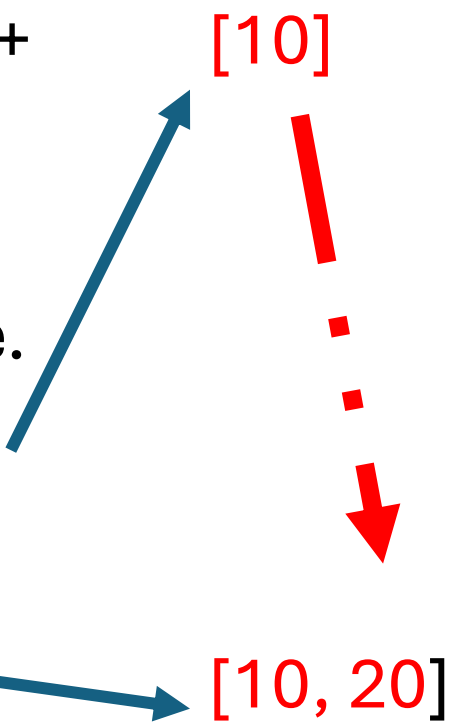


Question 4

- Consider the following set of numbers:
  10, 20, 5, 6, 12, 30, 7, 17
- **a)** Create a B+ Tree using the above dataset when order (m)=3 .
  Note: You may present the step-by-step procedure of B+ Tree construction, including explanations at each intermediate step.
- **b)** Why are the leaf nodes linked in a B+ Tree?

Answer : Question 4

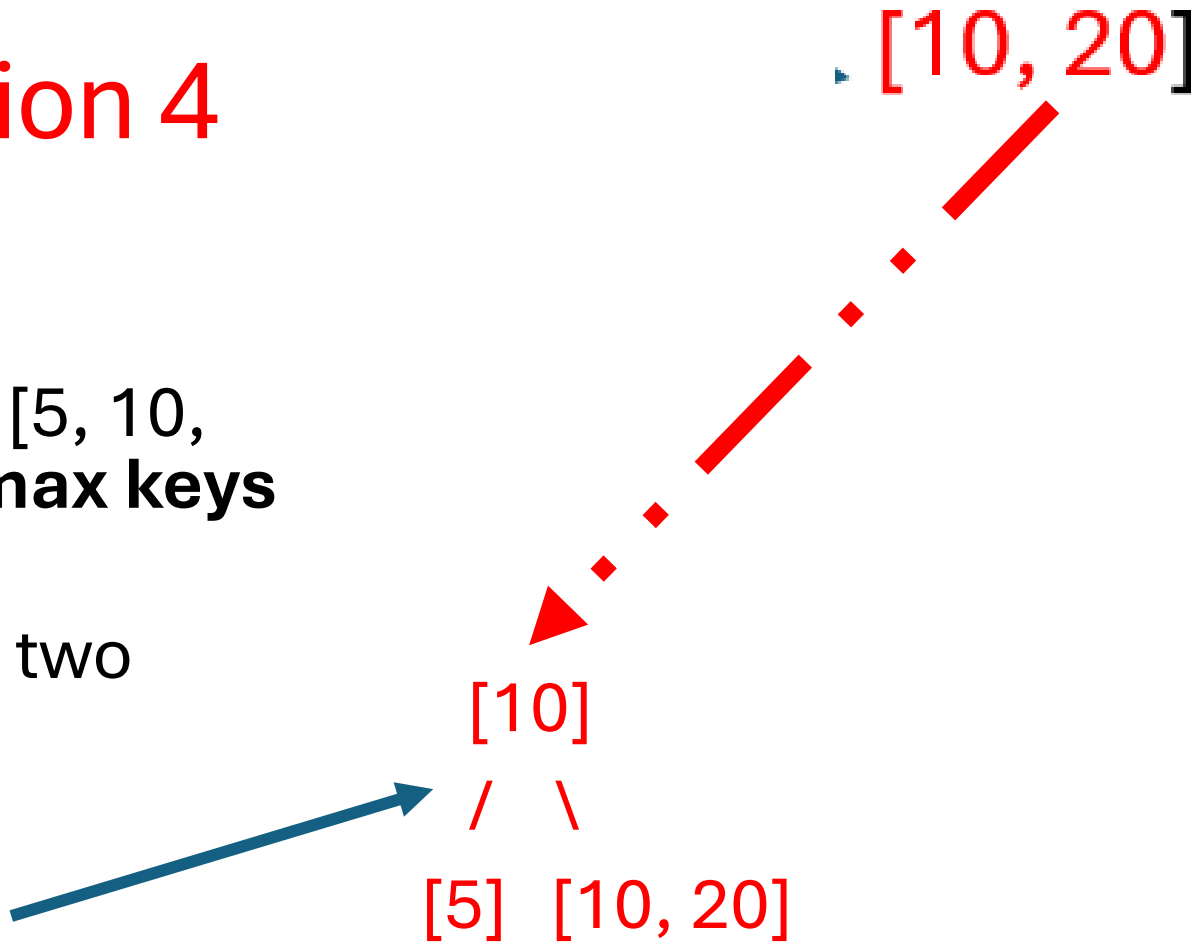
- Step-by-Step Insertion in B+ Tree (Order = 3)
- **Step 1: Insert 10**
- Start with an empty B+ Tree.
- Insert 10 in the **leaf node**.
- **Step 2: Insert 20**
- Add 20 to the same leaf.



Answer : Question 4

Step 3: Insert 5

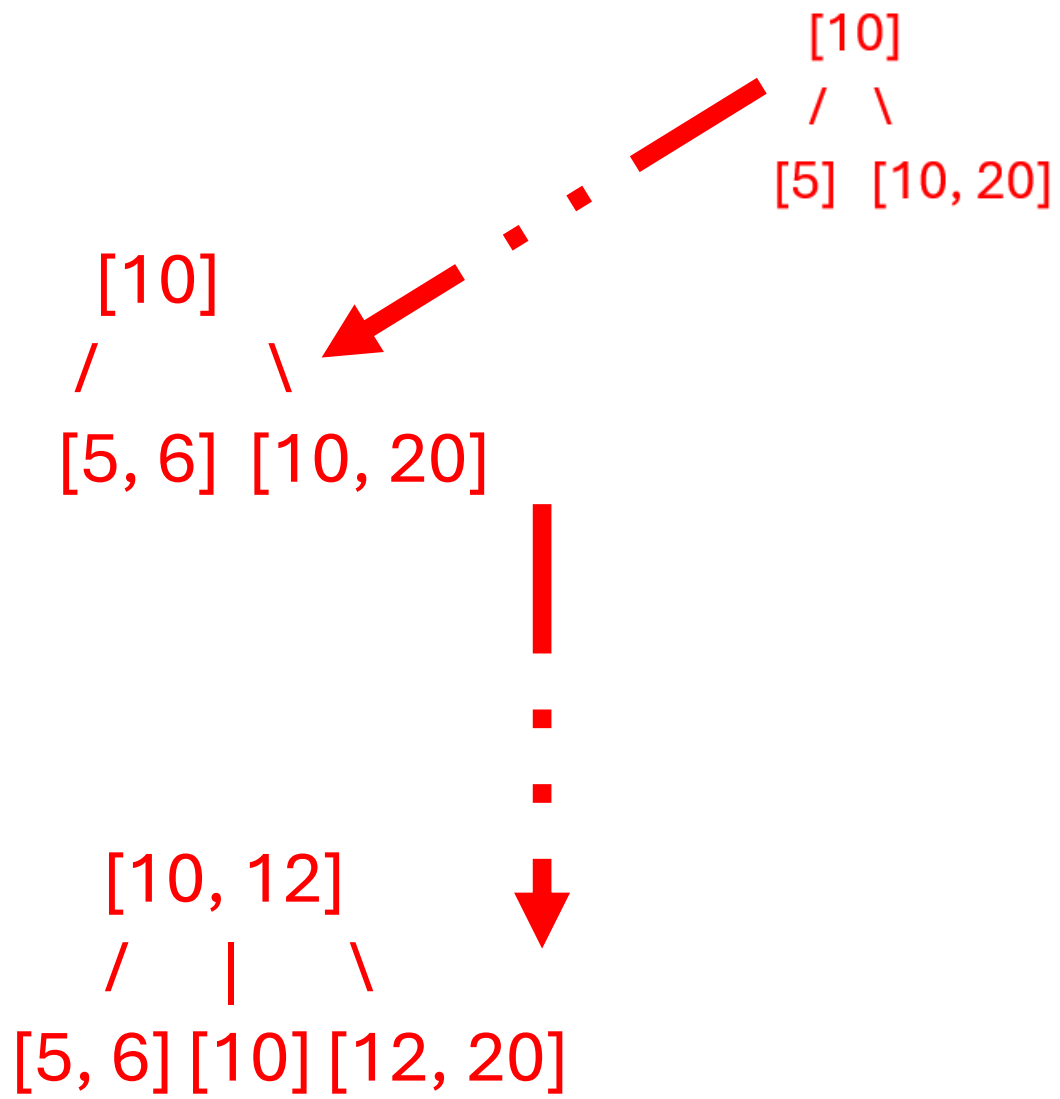
- Add 5 → Now leaf has [5, 10, 20], which **exceeds max keys = 2**.
- So, **split the leaf** into two nodes:
  - Left: [5]
  - Right: [10, 20]
- Push up the **first key of the right node (10)** to create the root.



Answer : Question 4

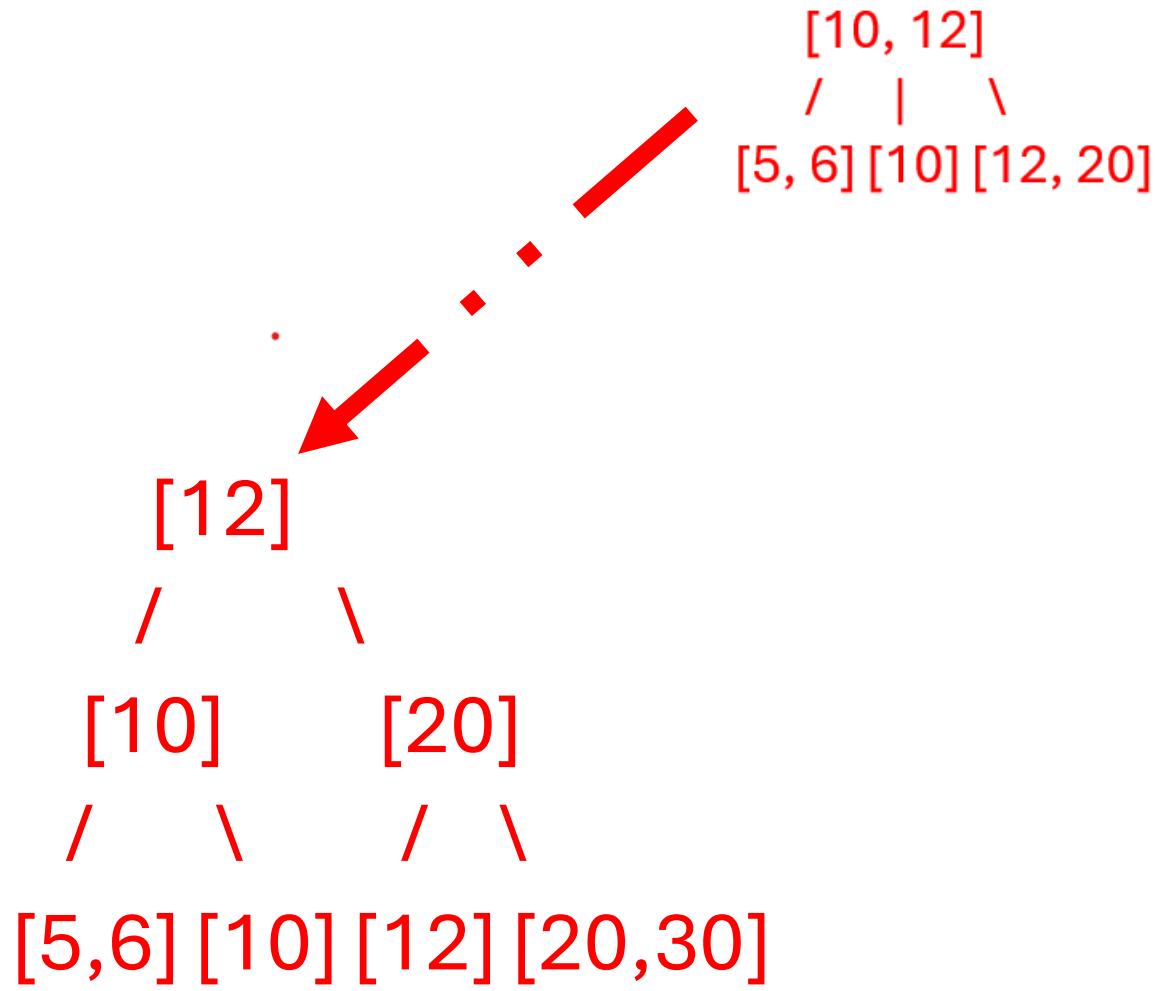
- **Step 4: Insert 6**
- 6 < 10 → Goes to [5]
- Insert into [5] → becomes [5, 6]

- **Step 5: Insert 12**
- 12 > 10 → Goes to [10, 20]
- Insert into [10, 20] → becomes [10, 12, 20] → **overflow!**
- Split:
  - Left: [10]
  - Right: [12, 20]
- Promote **12** to parent.
- Now root has [10, 12], with 3 children:



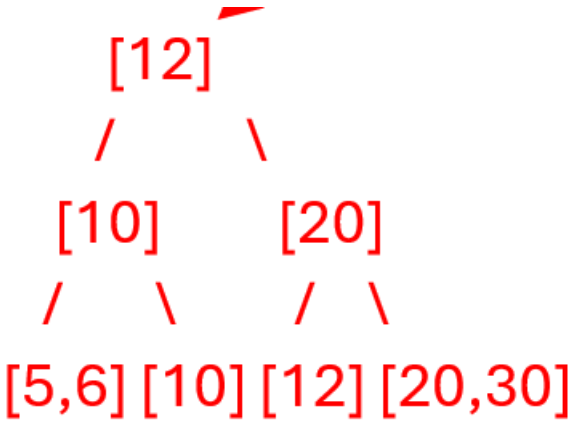
Answer : Question 4

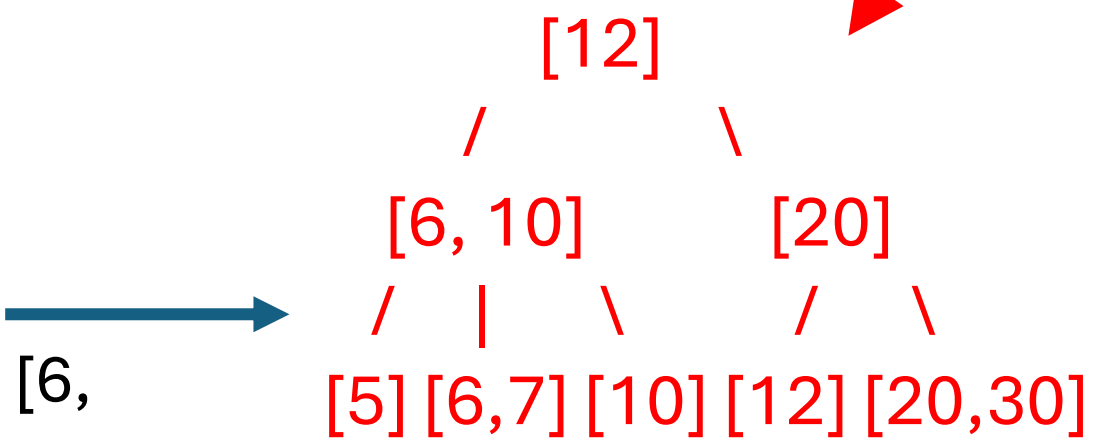
- **Step 6: Insert 30**
- 30 > 12 → Goes to [12, 20]
- Insert into [12, 20] → becomes [12, 20, 30] → **overflow!**
- Split:
  - Left: [12]
  - Right: [20, 30]
- Promote **20** → Insert into root [10, 12] → becomes [10, 12, 20] → **overflow!**
- Time to **split the root**



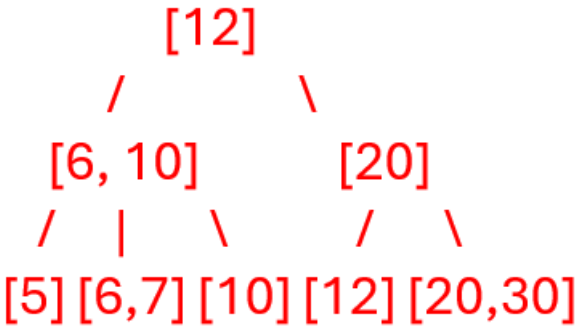
Answer : Question 4

- **Step 7: Insert 7**
- 7 < 12 → Left subtree
- 7 < 10 → [5, 6]
- Insert into [5, 6] → becomes [5, 6, 7] → **overflow!**
- Split:
  - Left: [5]
  - Right: [6, 7]
- Promote **6** to parent [10]
- Now update structure:
- Internal node [10] becomes [6, 10]

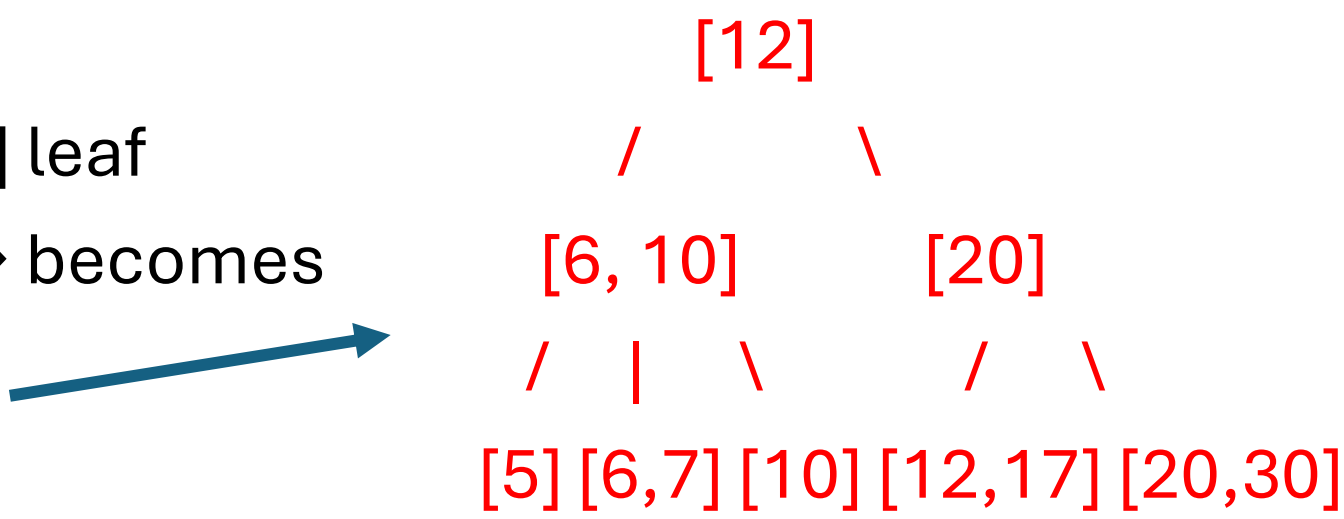




Answer : Question 4



- **Step 8: Insert 17**
- 17 < 20 → Go to [12] leaf
- Insert 17 into [12] → becomes [12, 17]



- Internal nodes direct the search and hold only keys, not data.
- Leaf nodes contain actual data and are linked sequentially.
- Splits propagate upward, and root split causes tree height increase.
- All data exists at **leaf level**; **internal nodes only guide**.

Answer : Question 4

(b)Why Are Leaf Nodes Linked in a B+ Tree?

- In a **B+ Tree**, **only the leaf nodes store the actual data** (records). To efficiently **support range queries**, **sequential access**, and **fast full-table scans**, **leaf nodes are connected using pointers (linked list fashion)**.
- Each **leaf node** has a **pointer to its right sibling** (and sometimes left).
- This allows traversal of **all data in sorted order without needing to go back up the tree**.

Lecture – 6
Graphs

M.G.Noel A.S Fernando (PhD)

Professor

NSBM/Dept. of Information Systems Engineering, UCSC

Introduction to Graph

- In **Data Structures and Algorithms (DSA)**, a **graph** is a **non-linear data structure** that consists of:

Vertices (or nodes) — the entities.

Edges (or arcs) — the connections or relationships between the vertices

- A graph **G** is represented as:

G=(V,E) Where:

V is a set of vertices.

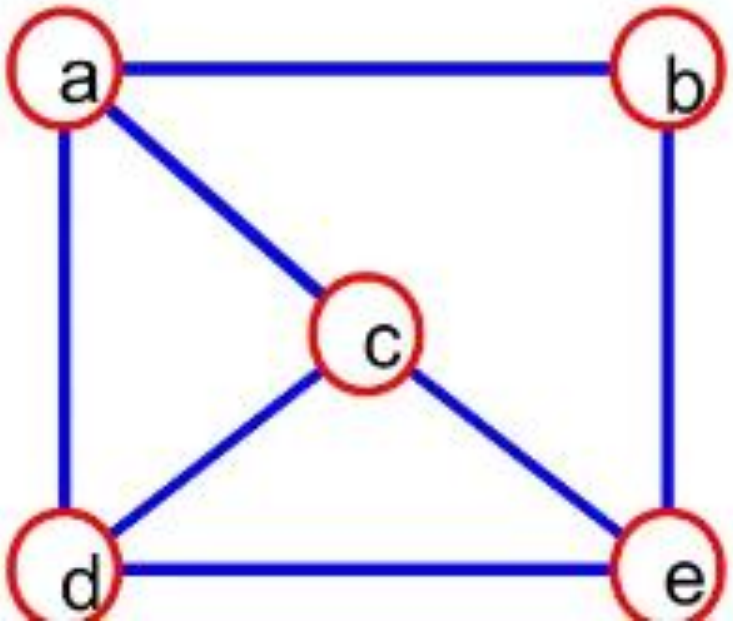
E is a set of edges, which are pairs of vertices.

What is a Graph?

- A graph G = (V,E) is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:



V= {a,b,c,d,e}

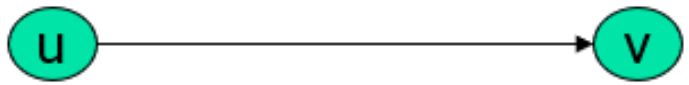
E= {(a,b),(a,c),
(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Applications in Graphs

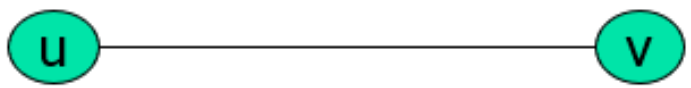
- **Computer Science Applications**
  - •**Data Structures**: Representing networks such as social networks, web pages (links), and file systems.
  - •**Routing Algorithms**: Dijkstra's, Bellman-Ford, and Floyd-Warshall for shortest paths in networks and more
- **Networking**
  - •**Computer Networks**: Routers and switches are modeled as graph nodes; connections are edges.
  - •**Internet Structure**: Web graph where web pages are vertices and hyperlinks are edges.
- **Social Networks**
  - •Friendship/Follow Relationships: Facebook, Twitter, LinkedIn all use graph models.
  - •Community Detection: Identifying clusters of people with common interests.
- **And more**

Different types of graphs

Directed: Ordered pair of vertices. Represented as (u, v) directed from vertex u to v.

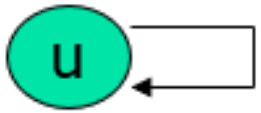


Undirected: Unordered pair of vertices. Represented as {u, v}. Disregards any sense of direction and treats both end vertices interchangeably.



Different types of graphs (cont..)

- **Loop:** A loop is an edge whose endpoints are equal i.e., an edge joining a vertex to it self is called a loop. Represented as {u, u} = {u}

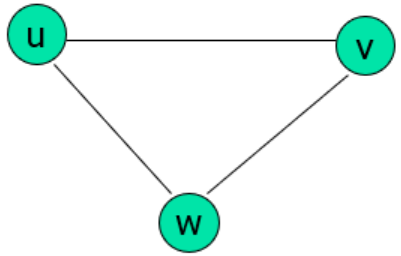


- **Multiple Edges:** Two or more edges joining the same pair of vertices.

Different types of graphs (cont..)

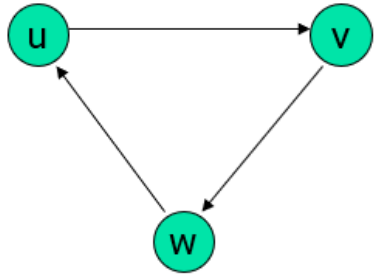
Simple (Undirected) Graph: consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges (undirected)

Representation Example: G(V, E), V = {u, v, w}, E = {{u, v}, {v, w}, {u, w}}



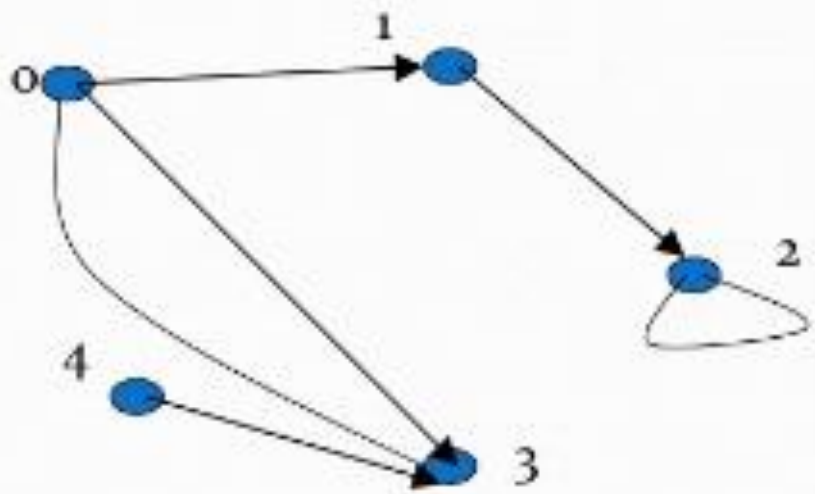
Directed Graph: G(V, E), set of vertices V, and set of Edges E, that are ordered pair of elements of V (directed edges)

Representation Example: G(V, E), V = {u, v, w}, E = {(u, v), (v, w), (w, u)}



Examples of Graphs

- V={0,1,2,3,4}
- E={(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)}



When (x,y) is an edge,
we say that x is *adjacent to* y, and y is *adjacent from* x.

0 is adjacent to 1.
1 is not adjacent to 0.
2 is adjacent from 1.

Terminology – 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 $\langle v_0, v_1 \rangle$ is an edge in a directed graph
  - v_0 is **adjacent to** v_1, and v_1 is **adjacent from** v_0
  - The edge $\langle v_0, v_1 \rangle$ is incident on v_0 and v_1

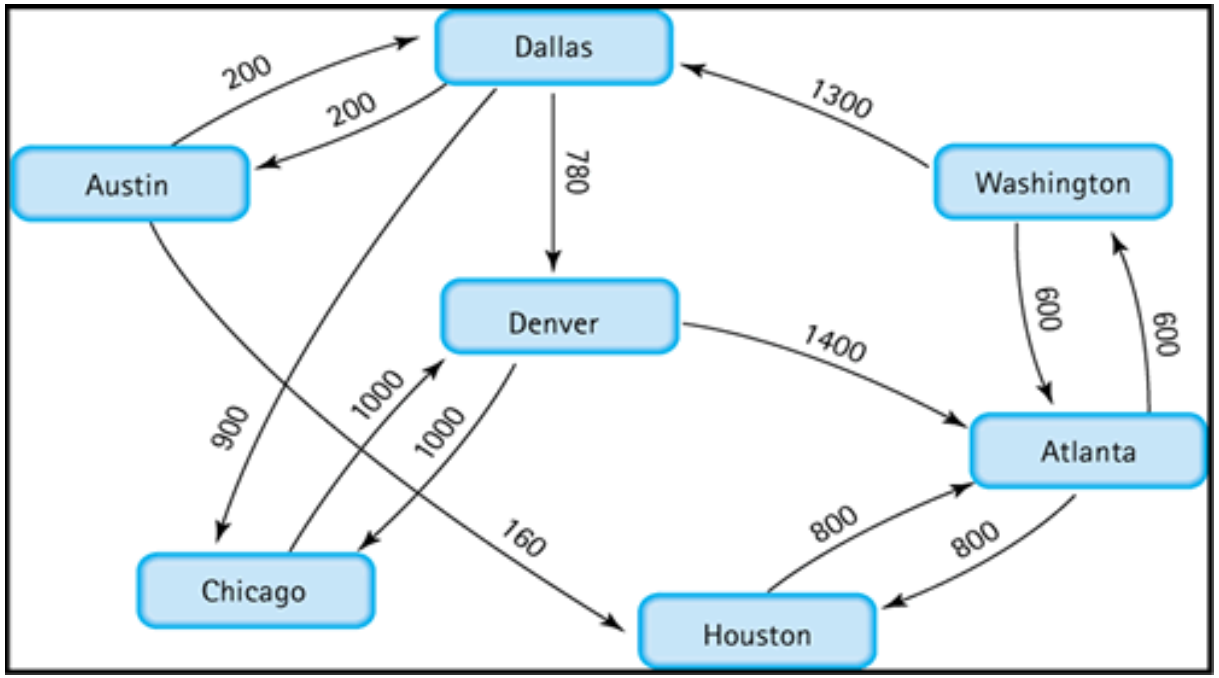
Graph –Example

- If numbers are associated with each arc of a graph, the graph is called a weighted graph or a network.

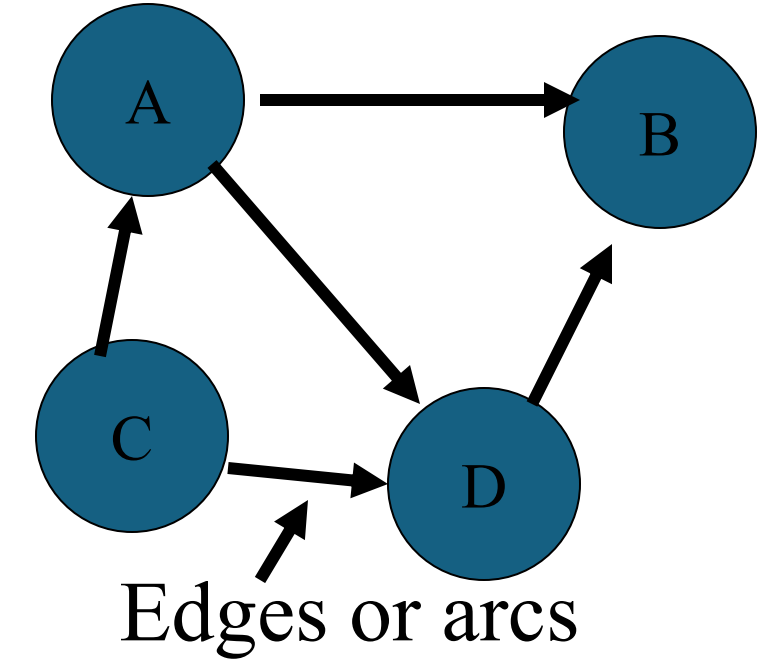
  Eg Nodes – towns

  Arcs - Road

  Weight – Distance



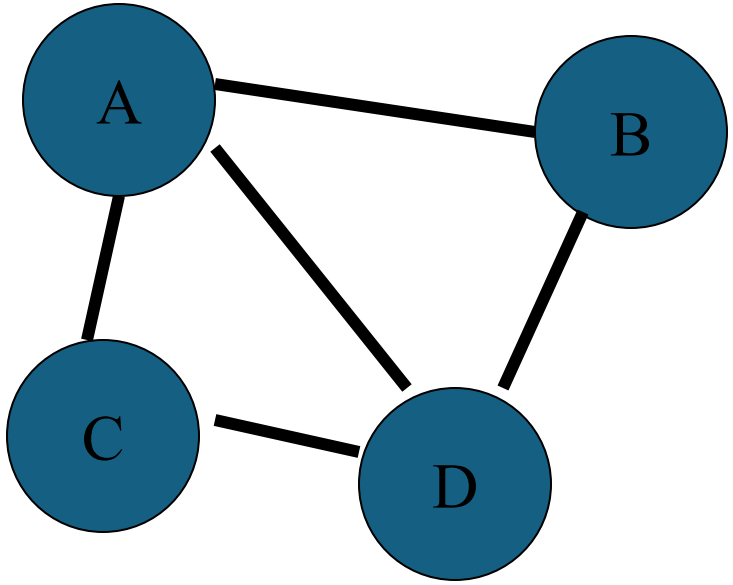
Examples



Nodes or vertex

Edges or arcs

Directed



Un-directed

Graph Terminology

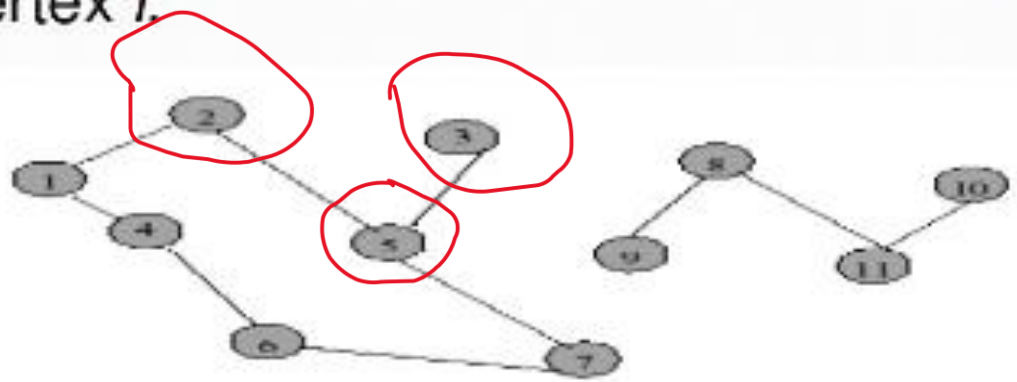
- Adjacent nodes: two nodes are adjacent if they are connected by an edge

5 is adjacent to 7
7 is adjacent from 5

- Path: a sequence of vertices that connect two nodes in a graph
- A simple path is a path in which all vertices, except possibly in the first and last, are different.
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph Terminology cont..

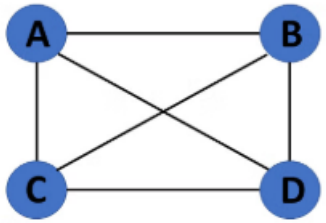
- A cycle is a simple path with the same start and end vertex.
- The **degree** of vertex *i* is the no. of edges incident on vertex *i*.



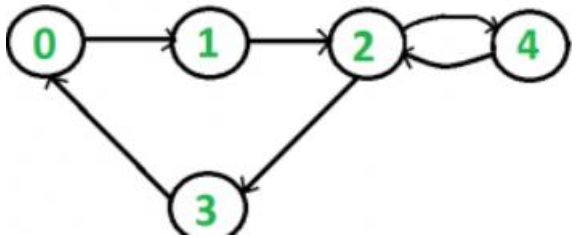
e.g., degree(2) = 2, degree(5) = 3, degree(3) = 1

Graph Terminology cont..

- Connected graph
  - A connected graph is a graph where every pair of vertices is joined by at least one path

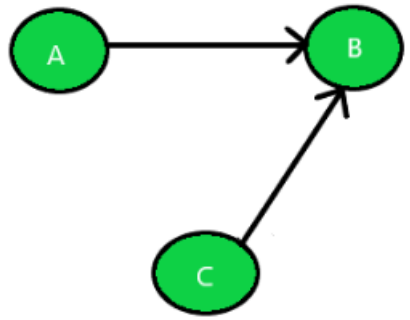


- Strongly connected directed graph
  - A strongly connected directed graph is a directed graph where every vertex can reach every other vertex.



Graph Terminology cont..

- Weekly connected graphs : This property describes a directed graph where all vertices are connected if you disregard the direction of the edges



- **Complete digraph:** is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction)

Example

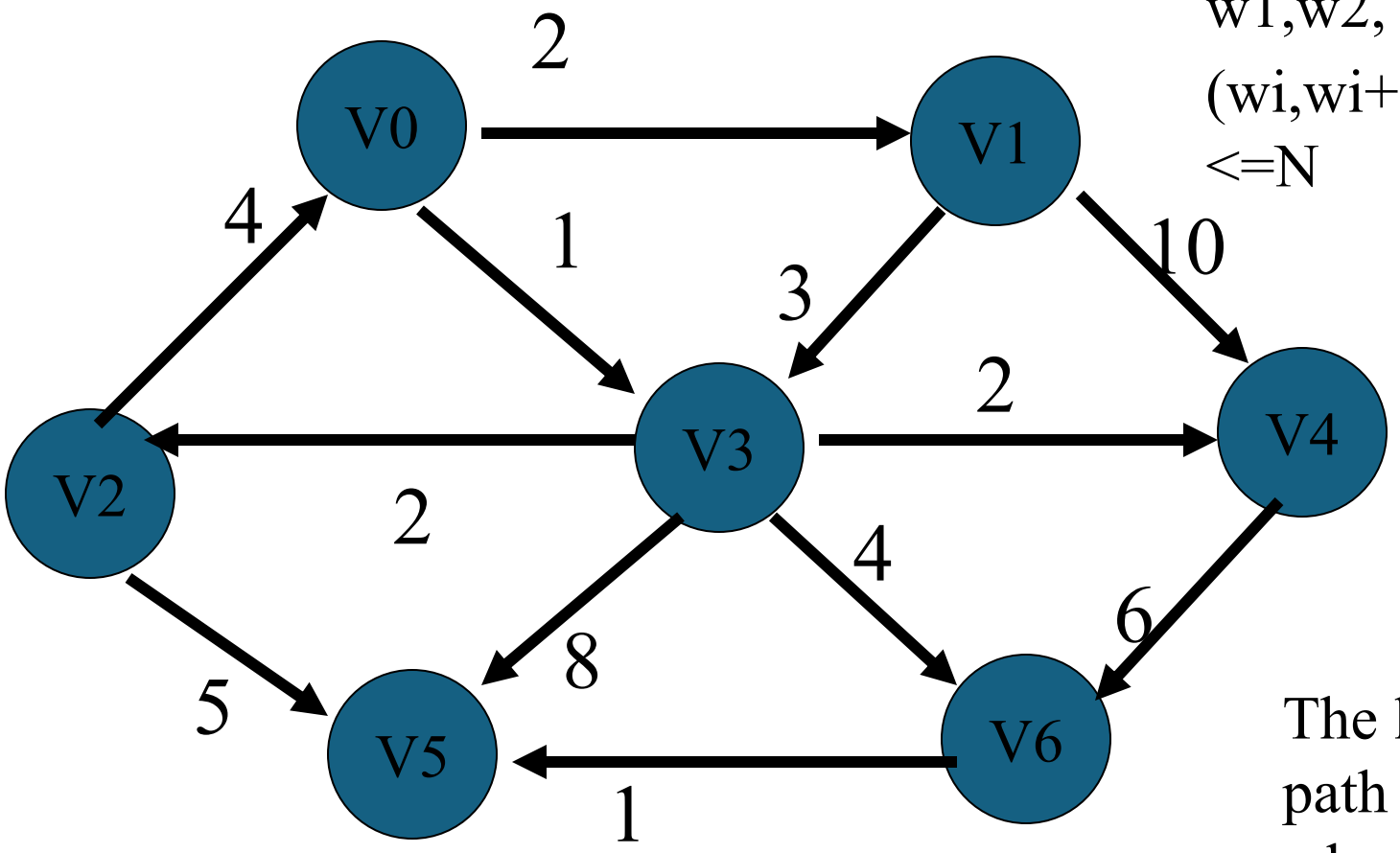
The graph G has the following 8 vertices.

V={
V0,V1,V2,V3,V4,V5,V6,V7 }

And 12 edges

E={(V0,V1,2),(V0,V3,1),(V1,V3,3),(V1.V4,10),V2,V4,2),(V3,V6,4),(V3,V5,8),(V3,V2,2),(V2,V0,4),(V2,V5,5),(V4,V6,6),(V6,V5,1)}

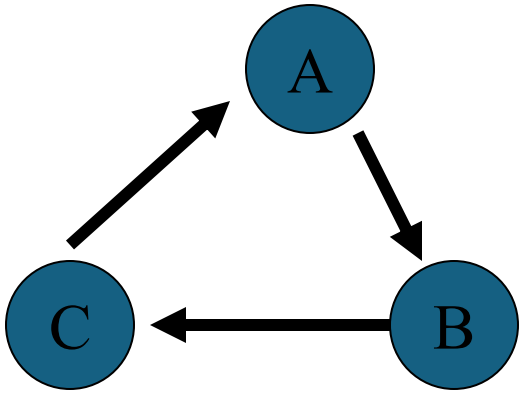
A path in a graph is a sequences of vertices w1,w2, …..wn such that (wi,wi+1) ε E for 1=< I <=N



A directed graph

The length of such a path is the number of edges on the path.(N-1). This is called un-weighted path length.

- The weighted path length is the sum of the costs of the edges on the path.
- As an example v0,v3,v5 is a path from vertex v0 to v5. The path length is two edges, and the weighted path length is 9.
- A path from a node to itself is called a cyclic. If a graph contains a cyclic, it is cyclic otherwise it is acyclic



Graph Implementations

- Graph data structures can be implemented in several ways depending on memory efficiency, type of operations, and use cases.
- The **main implementation methods** are:
- Adjacency Matrix
- Adjacency List
- Incidence Matrix
- Edge List/Table

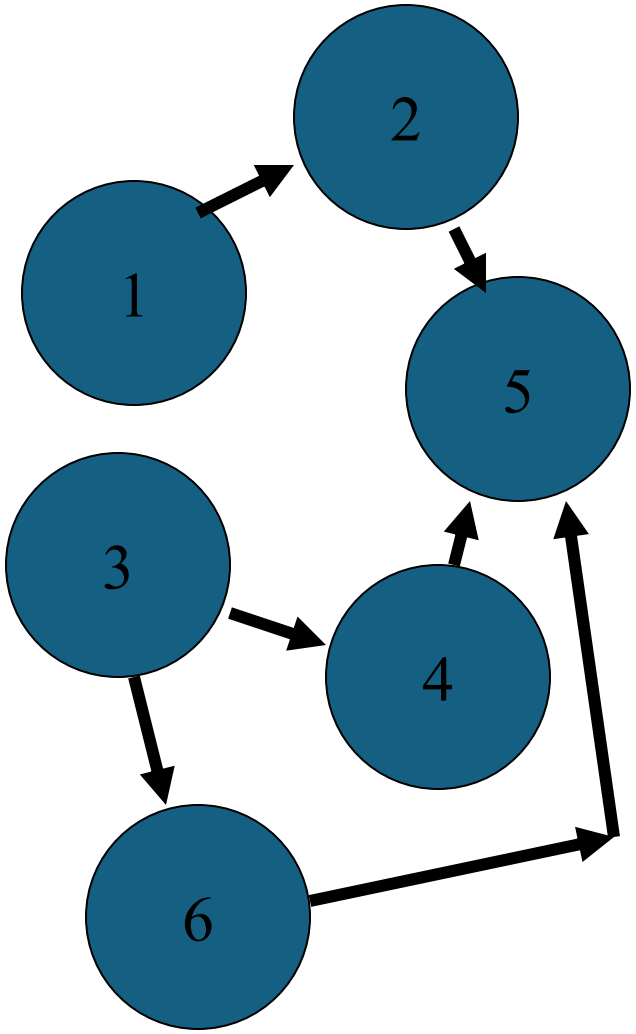
Adjacency Matrix in Graphs- (Array based implementation of graphs)

- An **adjacency matrix** is a **2D array (matrix) representation of a graph**.
- We can use a two-dimensional square array. The dimension of the array is equal to the maximum number of vertices that our graph can have.
- For a graph with V vertices, the adjacency matrix is a V × V matrix A where:
- For an **unweighted graph**:
  - A[i][j] = 1, if there is an edge between vertex i and vertex j
  - A[i][j] = 0, if there is no edge.
- For a **weighted graph**:
  - A[i][j] = w if there is an edge with weight w from i to j.
  - A[i][j] = 0 (or ∞ for directed graphs) if no edge exists

Adjacency Matrix in Graphs- (Array based implementation of graphs)

- In the simplest implementation of a graph, the array contains Boolean elements.
  In a graph represented by an array named *graph*, the element *graph[i][j]* is **true** if vertex *i* is connected to vertex *j* by an edge, and **false** if it is not.
  A purely array-based representation of a graph is called an **adjacency matrix**.
  The number of vertices in the graph is determined by the dimension of the array.

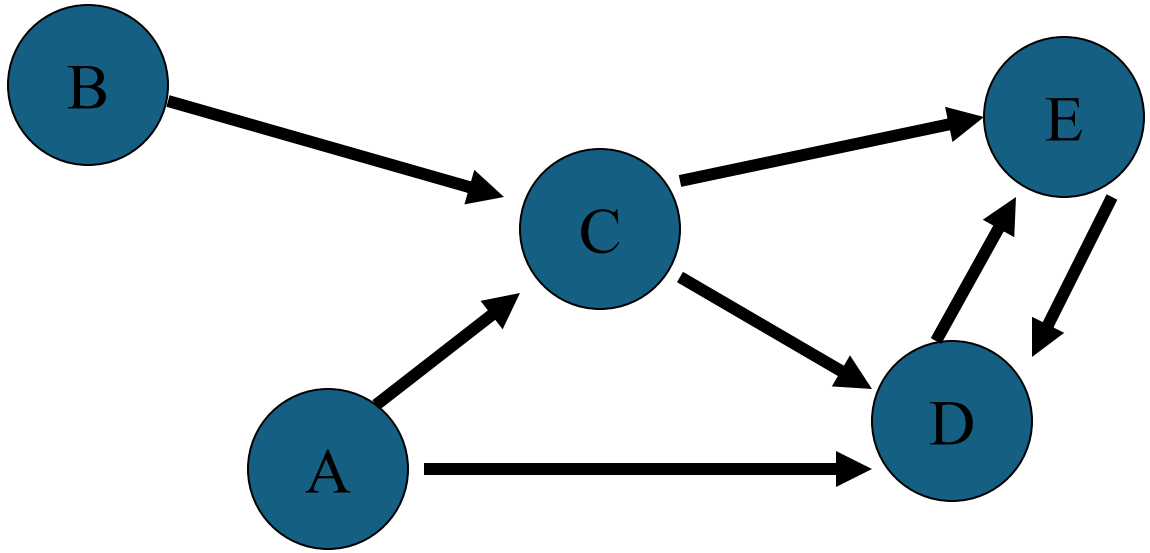
Consider the following unweighted directed graph



	1	2	3	4	5	6
1	F	T	F	F	F	F
2	F	F	F	F	T	F
3	F	F	F	T	F	T
4	F	F	F	F	T	F
5	F	F	F	F	F	F
6	F	F	F	F	T	F

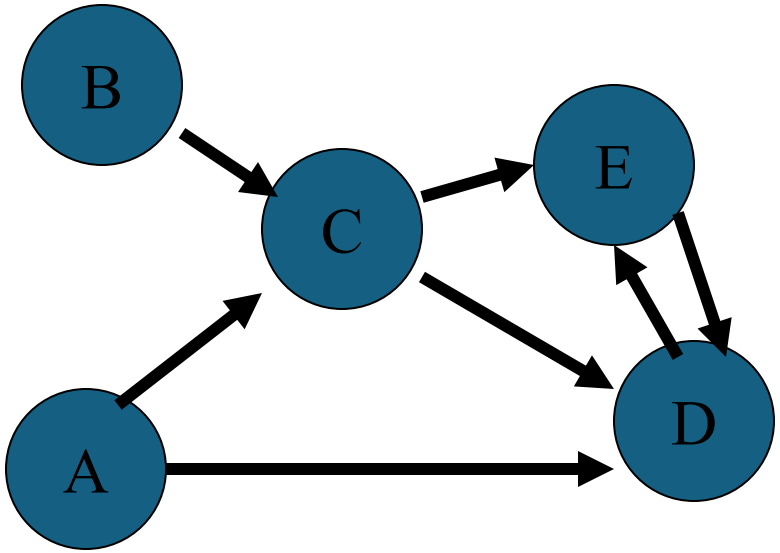
The graph is totally describing its adjacency matrix.

Boolean products of adjacency Matrices.



Adj[i][j] =1 or true then there
is an arc from node i to node j.
Adj[i][j] =0 or false otherwise

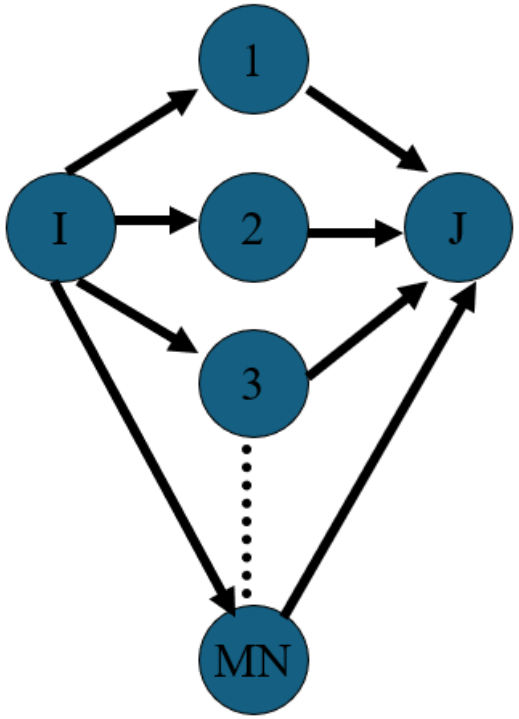
Let's us assume that a graph is completely describe its adjacency matrix



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

ADj

Adjacency values



- Consider the logical expression
- Adj[i][k] and adj[k][j]. Its value is true if and only if the values of both adj[I][k] and adj[k][j] are true.which implies that there is an arc from node I to node j.
- Thus, adj[I][k] and adj[k][j] equal to if and only if there is a path of length 2 from node I to node j passing through k..
- If adj[I][j] is true if and only if (path of length=2)
- Adj[I][1] and adj[1][j] or
- Adj[I][2] and adj[2][j] or
- Adj[I][3] and adj[3][j] or

...

Adj[I][MN] and adj[MN][j]

The value of this expression is true only if there is a path of length 2 from node I to node j passing through node 1, node 2 ..... MN

Adjacency matrices

Adj_2 is said to be Boolean product of adj with self

Adj=path matrix with length = 1 [adj]

Adj_2=path matrix with length = 2 [adj*adj]

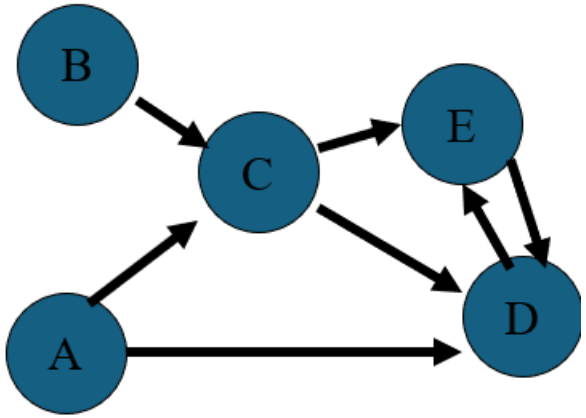
Adj_3=path matrix with length = 3 [adj_2*adj]

.....

Adj_l=path matrix with length = l [adj_{l-1}*adj

Question 1

- Consider the following directed graph.



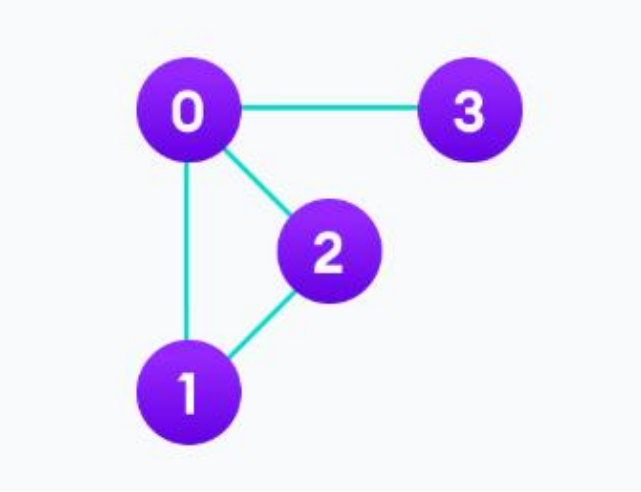
- Find the Adjacency matrices (adj, Adj_2,Adj_3,Adj_4 and Adj_5)
- Find the path matrix (transitive closure)

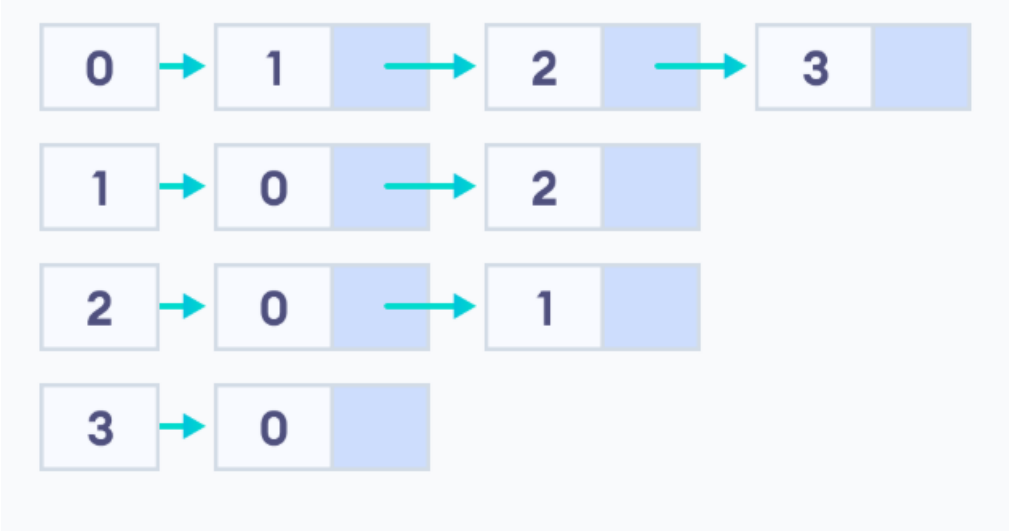
Advantages and disadvantages of adjacency matrix applications

Advantages	Disadvantages
Simple and Easy to Implement (Straightforward 2D array structure)	High Space Complexity (even if the graph has very few edges (sparse graph).
Constant-Time Edge Lookup	Inefficient for Sparse Graphs (Wastes memory when the number of edges E is much smaller than V^2)
Good for Dense Graphs (When the number of edges is close to the maximum possible (V^2), the matrix is efficient)	Slow to Traverse Neighbors
Supports Weighted Graphs (Can easily store edge weights instead of just 1/0)	Insertion/Deletion of Vertices is Costly (adding or removing a vertex requires resizing the matrix.)
Symmetry in Undirected Graphs (For undirected graphs, the matrix is symmetric, making it easier to process.)	Not Ideal for Certain Algorithms (Many graph algorithms (e.g., Dijkstra, BFS, DFS) are faster with adjacency)

Adjacency lists representation of graphs

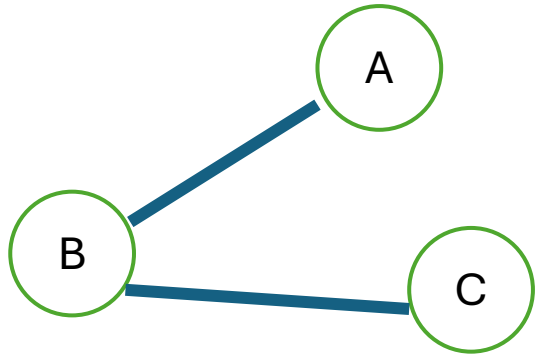
- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
- Example 1 – Undirected graph Adjacency List(Linked list of the graph)
-





Incidence definition

- In **graph theory**, the word **incident** describes the relationship between a **vertex** and an **edge.**
  - An **edge** is said to be **incident on** the **vertices** that form its endpoints.
  - A **vertex** is said to be **incident with** an **edge** if that edge is connected to the vertex.
- Example:
- graph $G = (V, E)$ with
- V={A,B,C} and
- $E = \{(A, B), (B, C)\}$



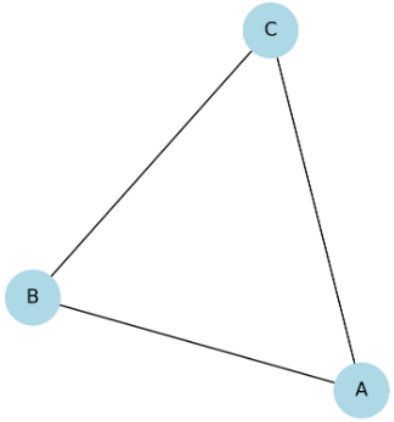
- The edge $(A, B)$is **incident** on vertices $A$and B.
-

Incidence Matrix (Graph Representation)

- An **incidence matrix** is a way to represent a graph using a **2D matrix** that shows the relationship between **vertices** and **edges**.
- Useful in **network flow problems, linear algebraic graph theory, and electrical circuits**.
- If a graph has V vertices and E Edges, its **incidence matrix** is a V×E matrix M , where:
  - Rows represent **vertices**.
  - Columns represent **edges**.
  - M[i][j]= 1(or sometimes −1) if vertex i is **incident** to edge j (i.e., edge j is connected to vertex i).
  - Otherwise, M[i][j]=0

Incidence Matrix (Graph Representation)

- Example
- Consider the following undirected graph.

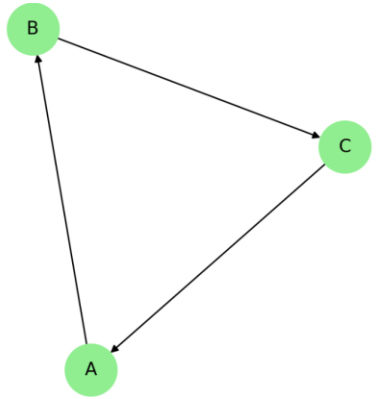


- Vertices: {A, B, C}
- edge: {e1 = A–B, e2 = B–C, e3 = C–A}

Vertex/Edge	e1 (A–B)	e2 (B–C)	e3 (C–A)
A	1	0	1
B	1	1	0
C	0	1	1

Incidence Matrix (Graph Representation)

- Example
- Consider the following directed graph.

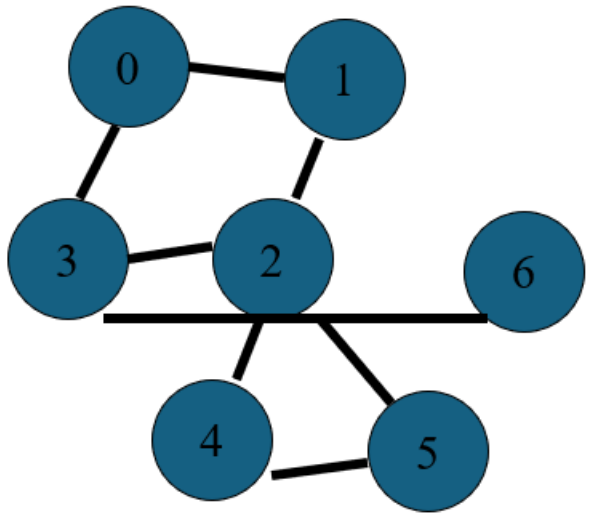


- Vertices: {A, B, C}
- Edges: {e1 = A → B, e2 = B → C, e3 = C → A}
- Incidence Matrix:

Vertex/Edge	e1 (A→B)	e2 (B→C)	e3 (C→A)
A	-1	0	1
B	1	-1	0
C	0	1	-1

Edge list/Table

Consider the following undirected graph



Node	Vertex list
0	1 3
1	0 2
2	1 3 4 5
3	0 2 6
4	2 5
5	2 4
6	3

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order.
- There are two commonly used methods of travelling a graph.
  - Depth First Traversal (DFT): use recursion stack
  - Breadth First Traversal (BFT): use queue

Depth-First Traversal/ Search (DFS/DFT)

DFS(v): visits all the nodes reachable from v in depth-first order

- Mark v as visited
- For each edge $v \to u$:
  - If u is not visited, call DFS(u)

- Use non-recursive version if recursion depth is too big (over a few thousands)
  - Replace recursive calls with a stack

Breadth-First Traversal/ Search (DFS/DFT)

BFS(v): visits all the nodes reachable from v in breadth-first order

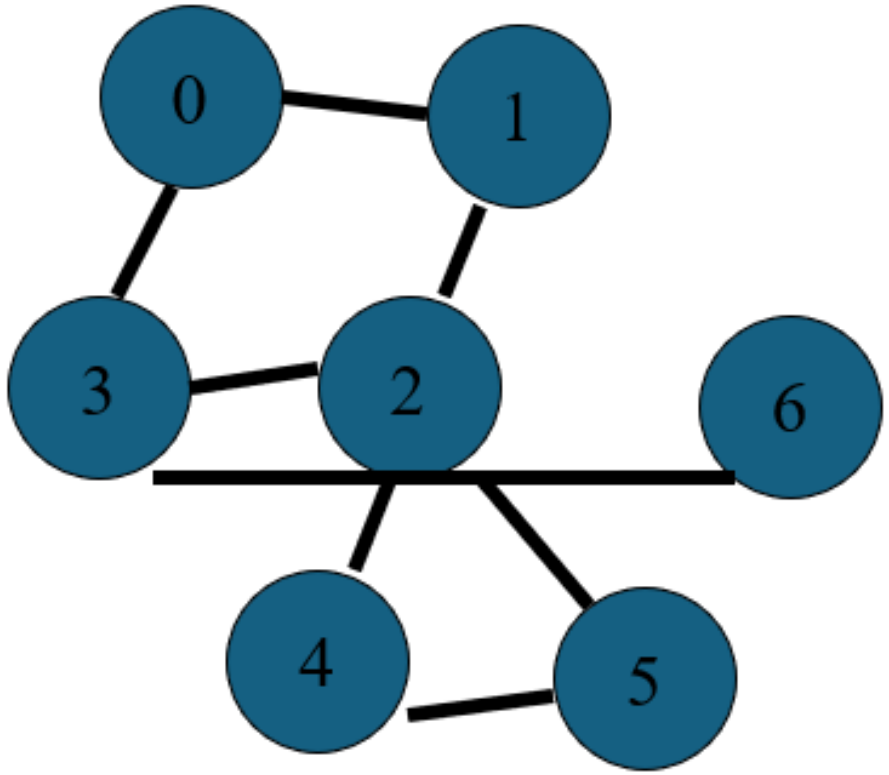
- Initialize a queue Q
- Mark v as visited and push it to Q
- While Q is not empty:
  - Take the front element of Q and call it w
  - For each edge $w \to u$:
    - If u is not visited, mark it as visited and push it to Q

Depth First Traversal (DFT)

- First, we construct an edge table for the graph.
- An edge table is built by listing all the vertices of the graph in a vertical column.
- Next to each entry in the column we list all vertices to which that entry is connected directly.

Question 2

- Consider the following Graph.
- Creating the edge table and traversing it using Depth First Traversal.

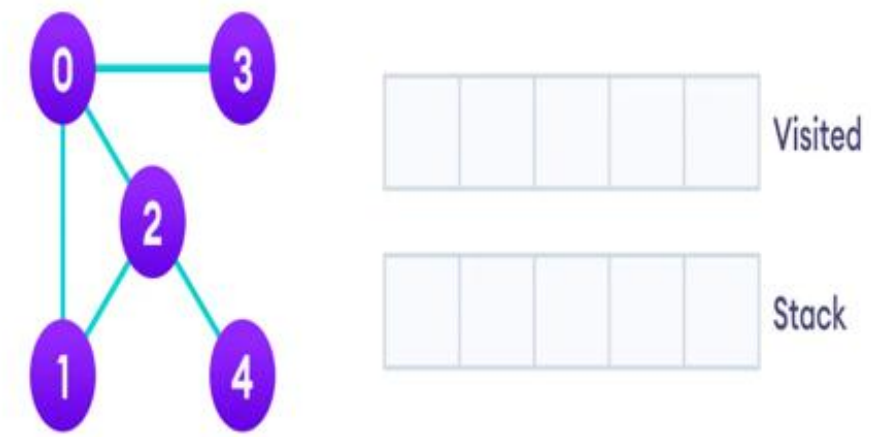


Implementation of Depth First Search/Traversal using a Stack

- **Depth First Search Algorithm**
- A standard DFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The DFS algorithm works as follows:
  - Start by putting any one of the graph's vertices on top of a stack.
  - Take the top item of the stack and add it to the visited list.
  - Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
  - Keep repeating steps 2 and 3 until the stack is empty.

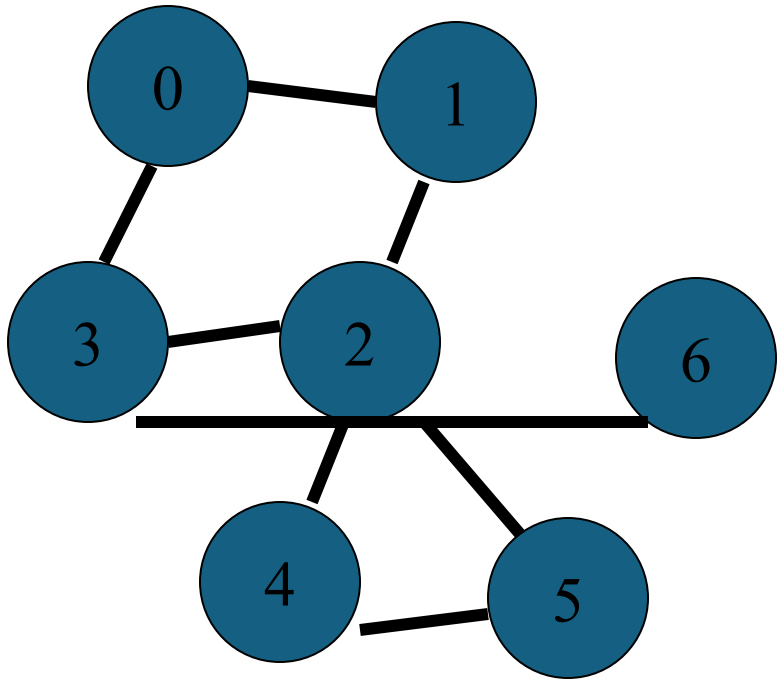
Question 3

- Consider the following Graph.



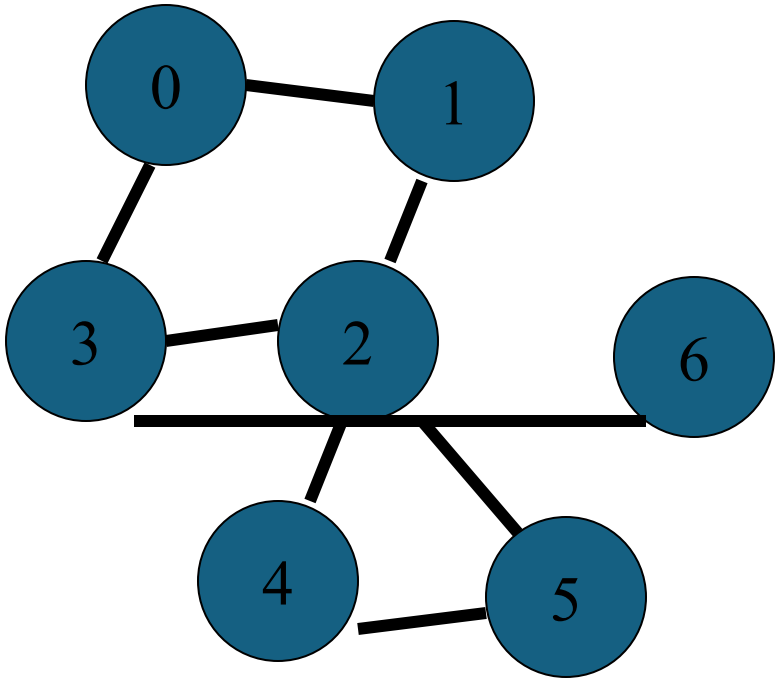
- Explain how to traverse the above graph using Depth First Traversal with a stack.
- Write the pseudocode algorithm for DFT/DFS

Breadth First Traversal



The idea here is to list all the vertices connected to each vertex before moving the on in the edge table.

Example (BFS/T)



The edge table for the above graph.

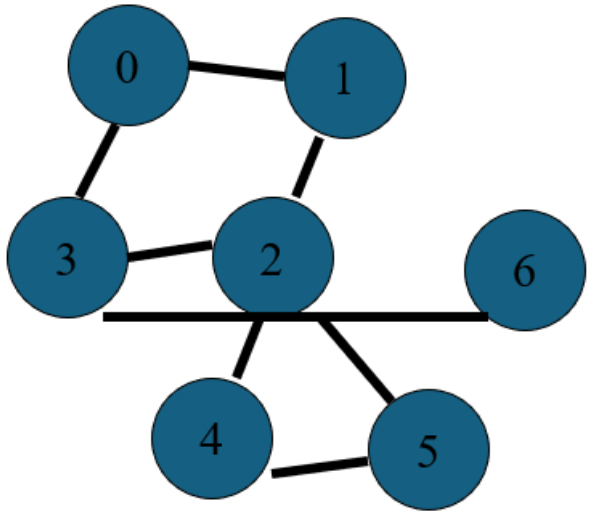
Node	Vertex list
0	1 3
1	0 2
2	1 3 4 5
3	0 2 6
4	2 5
5	2 4
6	3

Algorithm

1. Start at vertex V and mark it visited
2. Visit all the un-visited vertices adjacent to V
3. Visit all the un-visited adjacent to the vertices visited at the previous level
4. Continue this process until all the vertices are visited.

Question 4

- Consider the following graph.



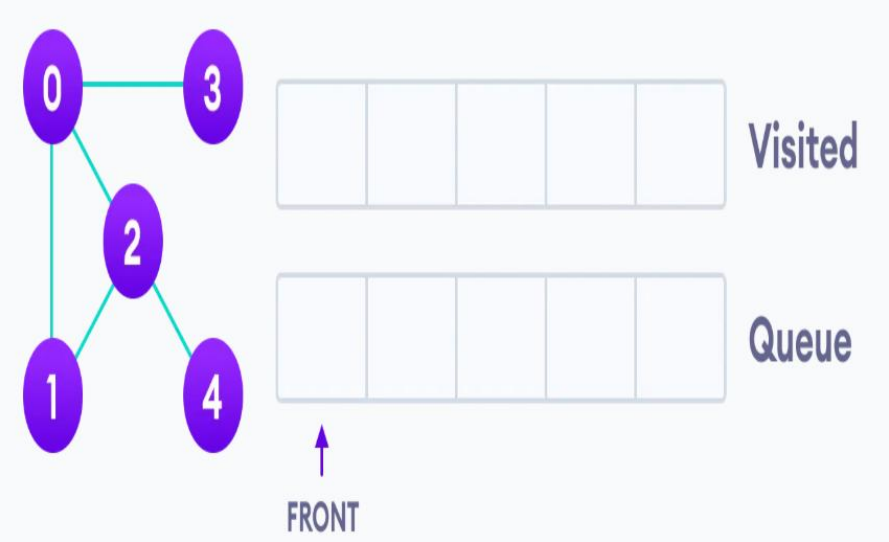
- Creating the edge table and traversing it using Breadth First Traversal.

Implementation of **BFS** Search/Traversal using a Queue

- The algorithm works as follows:
- **Step 1**: Start by putting any one of the graph's vertices at the back of a queue.
- **Step 2**: Take the front item of the queue and add it to the visited list.
- **Step 3**:Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
- **Step 4**: Keep repeating steps 2 and 3 until the queue is empty.

Question 5

- Consider the following Graph.



- Explain how to traverse the above graph using Breadth First Traversal with a queue.
- Write the pseudocode algorithm for BFT/BFS