Unit 4-6

Data Structures Unit 4-6 Answers

Unit 4: Linked List

1. What is linked list? Explain types of linked list with the help of example.

A linked list is a linear data structure where elements are stored in nodes, and each node contains data and a reference (pointer) to the next node. Unlike arrays, linked lists don't store elements in contiguous memory locations.

Types of Linked Lists:

1. **Singly Linked List**: Each node contains data and a pointer to the next node. The last node points to NULL. Example: 10 → 20 → 30 → NULL
2. **Doubly Linked List**: Each node has data, a pointer to the next node, and a pointer to the previous node. Example: NULL ← 10 ↔ 20 ↔ 30 → NULL
3. **Circular Linked List**: The last node points back to the first node, forming a circle. Example: 10 → 20 → 30 → (back to 10)
4. **Circular Doubly Linked List**: Combines features of doubly and circular linked lists.

Linked lists provide dynamic memory allocation, efficient insertion/deletion at the beginning, but require sequential access to elements and use extra memory for storing pointers.

2. What is one-way (singly) linked list? Explain representation of singly linked list in memory with example.

A singly linked list is a linear data structure where each node contains two parts: data and a pointer to the next node. It allows traversal in only one direction (forward).

Structure of Node:

```
struct Node {
    int data;
    struct Node* next;
}
```

Memory Representation Example: Consider nodes with values 10, 20, 30 stored at memory addresses 1000, 2000, 3000:

```
Address  | Data | Next
---------|------|------
```

```
1000     | 10   | 2000
2000     | 20   | 3000
3000     | 30   | NULL
```

The head pointer stores address 1000. Each node's next pointer contains the address of the following node, creating a chain. The last node's next pointer is NULL, indicating the end of the list. This non-contiguous memory allocation allows dynamic size adjustment during runtime.

3. Write an algorithm or pseudo code to perform following operations on singly Linked List

a. Traversing a linked list:

```
TRAVERSE(HEAD)
1. SET PTR = HEAD
2. REPEAT WHILE PTR ≠ NULL
3.    PRINT PTR→DATA
4.    SET PTR = PTR→NEXT
5. END
```

b. Searching in unsorted list:

```
SEARCH_UNSORTED(HEAD, ITEM)
1. SET PTR = HEAD
2. REPEAT WHILE PTR ≠ NULL
3.    IF PTR→DATA = ITEM THEN
4.       RETURN PTR
5.    SET PTR = PTR→NEXT
6. RETURN NULL
```

c. Searching in sorted list:

```
SEARCH_SORTED(HEAD, ITEM)
1. SET PTR = HEAD
2. REPEAT WHILE PTR ≠ NULL AND PTR→DATA ≤ ITEM
3.    IF PTR→DATA = ITEM THEN
4.       RETURN PTR
5.    SET PTR = PTR→NEXT
6. RETURN NULL
```

d. Insert at beginning:

```
INSERT_BEGINNING(HEAD, ITEM)
1. CREATE NEW_NODE
2. SET NEW_NODE→DATA = ITEM
```

```
3. SET NEW_NODE→NEXT = HEAD
4. SET HEAD = NEW_NODE
```

4. What is two-way (doubly) linked list?

A doubly linked list is a linear data structure where each node contains three parts: data, a pointer to the next node, and a pointer to the previous node. This bidirectional linking allows traversal in both forward and backward directions.

Structure of Node:

```
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
}
```

Advantages:

- Bidirectional traversal
- Efficient deletion when node reference is given
- Easy insertion before a given node
- Better navigation in applications like browsers

Disadvantages:

- Extra memory overhead for previous pointer
- More complex insertion/deletion operations
- Additional pointer maintenance required

The first node's previous pointer is NULL, and the last node's next pointer is NULL, marking the boundaries of the list.

5. Write an algorithm to perform following operations on doubly Linked List

a. Traversing a linked list (Forward):

```
TRAVERSE_FORWARD(HEAD)
1. SET PTR = HEAD
2. REPEAT WHILE PTR ≠ NULL
3.     PRINT PTR→DATA
4.     SET PTR = PTR→NEXT
5. END
```

b. Searching in unsorted list:

```
SEARCH_UNSORTED(HEAD, ITEM)
1. SET PTR = HEAD
2. REPEAT WHILE PTR ≠ NULL
3.     IF PTR→DATA = ITEM THEN
4.         RETURN PTR
5.     SET PTR = PTR→NEXT
6. RETURN NULL
```

c. Searching in sorted list:

```
SEARCH_SORTED(HEAD, ITEM)
1. SET PTR = HEAD
2. REPEAT WHILE PTR ≠ NULL AND PTR→DATA ≤ ITEM
3.     IF PTR→DATA = ITEM THEN
4.         RETURN PTR
5.     SET PTR = PTR→NEXT
6. RETURN NULL
```

6. What is queue? Write algorithms to perform enqueue (insert) and dequeue (delete) operations on linear queue using linked list.

A queue is a linear data structure that follows First-In-First-Out (FIFO) principle. Elements are inserted at the rear and removed from the front.

ENQUEUE Operation:

```
ENQUEUE(FRONT, REAR, ITEM)
1. CREATE NEW_NODE
2. SET NEW_NODE→DATA = ITEM
3. SET NEW_NODE→NEXT = NULL
4. IF REAR = NULL THEN
5.     SET FRONT = REAR = NEW_NODE
6. ELSE
7.     SET REAR→NEXT = NEW_NODE
8.     SET REAR = NEW_NODE
9. END
```

DEQUEUE Operation:

```
DEQUEUE(FRONT, REAR)
1. IF FRONT = NULL THEN
2.     PRINT "Queue Underflow"
3.     RETURN
4. SET ITEM = FRONT→DATA
5. SET TEMP = FRONT
```

```
6. SET FRONT = FRONT→NEXT
7. IF FRONT = NULL THEN
8.    SET REAR = NULL
9. DELETE TEMP
10. RETURN ITEM
```

7. What is stack? Write algorithms to perform push and pop operations on stack using linked list.

A stack is a linear data structure that follows Last-In-First-Out (LIFO) principle. Elements are inserted and removed from the same end called the top.

PUSH Operation:

```
PUSH(TOP, ITEM)
1. CREATE NEW_NODE
2. SET NEW_NODE→DATA = ITEM
3. SET NEW_NODE→NEXT = TOP
4. SET TOP = NEW_NODE
5. PRINT "Item pushed successfully"
```

POP Operation:

```
POP(TOP)
1. IF TOP = NULL THEN
2.    PRINT "Stack Underflow"
3.    RETURN
4. SET ITEM = TOP→DATA
5. SET TEMP = TOP
6. SET TOP = TOP→NEXT
7. DELETE TEMP
8. RETURN ITEM
```

Advantages of linked list implementation:

- Dynamic size allocation
- No stack overflow (except memory limit)
- Efficient memory utilization
- Simple implementation of push/pop operations

8. Write an algorithm to perform following operations on circular Linked List

a. Create a linked list:

```
CREATE_CIRCULAR_LIST(LAST)
1. SET LAST = NULL
```

```
2. PRINT "Empty circular list created"
```

b. Insert into the list:

```
INSERT_CIRCULAR(LAST, ITEM, POS)
1. CREATE NEW_NODE
2. SET NEW_NODE→DATA = ITEM
3. IF LAST = NULL THEN
4.    SET NEW_NODE→NEXT = NEW_NODE
5.    SET LAST = NEW_NODE
6. ELSE
7.    SET NEW_NODE→NEXT = LAST→NEXT
8.    SET LAST→NEXT = NEW_NODE
9.    IF POS = "END" THEN
10.      SET LAST = NEW_NODE
11. END
```

c. Delete from that list:

```
DELETE_CIRCULAR(LAST, ITEM)
1. IF LAST = NULL THEN RETURN
2. SET PTR = LAST→NEXT, PREV = LAST
3. REPEAT
4.    IF PTR→DATA = ITEM THEN
5.       SET PREV→NEXT = PTR→NEXT
6.       IF PTR = LAST THEN SET LAST = PREV
7.       DELETE PTR
8.       RETURN
9.    SET PREV = PTR, PTR = PTR→NEXT
10. UNTIL PTR = LAST→NEXT
```

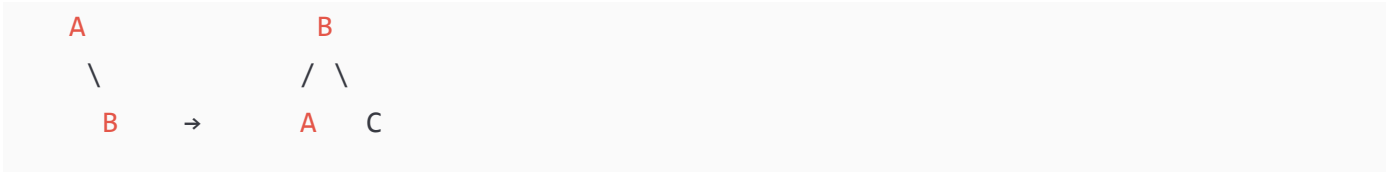
Unit 5: Trees

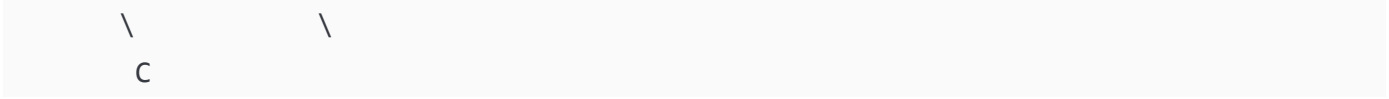
1. What is AVL tree? Write all the rotation operations of it.

An AVL tree is a self-balancing binary search tree where the height difference between left and right subtrees of any node is at most 1. It's named after Adelson-Velsky and Landis. The balance factor of each node is calculated as height(left subtree) - height(right subtree).

Rotation Operations:

1. Left Rotation (LL Rotation): Used when right subtree is heavier and right child has right subtree.





2. Right Rotation (RR Rotation): Used when left subtree is heavier and left child has left subtree.

3. Left-Right Rotation (LR Rotation): Combination of left rotation on left child, then right rotation on root.

4. Right-Left Rotation (RL Rotation): Combination of right rotation on right child, then left rotation on root.

2. Construct Binary Search Tree of following data and apply all the tree traversal techniques on that tree. Data:- 11,6,8,19,4,10,5,17,43,49,31,60

BST Construction: Starting with 11 as root, insert elements following BST property (left < root < right):

Tree Traversals:

Preorder (Root-Left-Right): 11, 6, 4, 5, 8, 10, 19, 17, 43, 31, 49, 60

Inorder (Left-Root-Right): 4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 49, 60

Postorder (Left-Right-Root): 5, 4, 10, 8, 6, 17, 31, 60, 49, 43, 19, 11

Note: Inorder traversal of BST gives sorted sequence.

3. Explain sequential (array) representation of binary tree. Explain preorder and inorder traversal in binary tree with example.

Sequential Representation: Binary trees can be represented using arrays where nodes are stored level by level. For a node at index i:

- Left child is at index 2i+1
- Right child is at index 2i+2
- Parent is at index (i-1)/2

Example Tree:

```
      A
     / \
    B   C
   / \   \
  D   E   F
```

Array Representation: [A, B, C, D, E, -, F] Index: [0, 1, 2, 3, 4, 5, 6]

Preorder Traversal (Root-Left-Right):

1. Visit root node
2. Recursively traverse left subtree
3. Recursively traverse right subtree Result: A, B, D, E, C, F

Inorder Traversal (Left-Root-Right):

1. Recursively traverse left subtree
2. Visit root node
3. Recursively traverse right subtree Result: D, B, E, A, C, F

4. Write an algorithm to insert an element in heap tree. Construct heap tree (maxheap) using following data: 23, 55, 46, 35, 10.

Algorithm for Insertion in Max Heap:

```
INSERT_HEAP(HEAP, N, ITEM)
1. SET N = N + 1
2. SET HEAP[N] = ITEM
3. SET PTR = N
4. REPEAT WHILE PTR > 1
5.    SET PAR = PTR / 2
6.    IF HEAP[PAR] >= HEAP[PTR] THEN EXIT
7.    SWAP HEAP[PAR] with HEAP[PTR]
8.    SET PTR = PAR
9. END
```

Max Heap Construction:

Insert 23: [23] Insert 55: [55, 23] (heapify up) Insert 46: [55, 23, 46] Insert 35: [55, 35, 46, 23] (heapify up) Insert 10: [55, 35, 46, 23, 10]

Final Max Heap:

```
      55
     /  \
   35    46
  /  \
 23   10
```

Array representation: [55, 35, 46, 23, 10]

5. Define AVL tree. With suitable example explain single rotations required for rebalancing in AVL trees after insertion operation.

An AVL tree is a height-balanced binary search tree where the balance factor (height difference between left and right subtrees) of every node is -1, 0, or +1. When this property is violated after insertion, rotations are performed to restore balance.

Single Rotations:

1. Left Rotation (LL Case): When insertion occurs in the right subtree of the right child.

Example: Insert 30, 20, 10 in sequence

```
30         20
 \   →    /  \
  20     10   30
 /
10
```

2. Right Rotation (RR Case): When insertion occurs in the left subtree of the left child.

Example: Insert 10, 20, 30 in sequence

```
 10         20
  \   →    /  \
   20     10   30
    \
     30
```

These single rotations maintain the BST property while restoring the AVL balance condition, ensuring O(log n) time complexity for all operations.

6. What is Binary tree? Write algorithms for preorder, inorder and post order traversal in binary tree.

A binary tree is a hierarchical data structure where each node has at most two children, referred to as left child and right child. It consists of nodes connected by edges, with one node designated as the root.

Preorder Traversal (Root-Left-Right):

```
PREORDER(ROOT)
1. IF ROOT ≠ NULL THEN
2.    PRINT ROOT→DATA
3.    CALL PREORDER(ROOT→LEFT)
4.    CALL PREORDER(ROOT→RIGHT)
5. END
```

Inorder Traversal (Left-Root-Right):

```
INORDER(ROOT)
1. IF ROOT ≠ NULL THEN
2.    CALL INORDER(ROOT→LEFT)
3.    PRINT ROOT→DATA
4.    CALL INORDER(ROOT→RIGHT)
5. END
```

Postorder Traversal (Left-Right-Root):

```
POSTORDER(ROOT)
1. IF ROOT ≠ NULL THEN
2.    CALL POSTORDER(ROOT→LEFT)
3.    CALL POSTORDER(ROOT→RIGHT)
4.    PRINT ROOT→DATA
5. END
```

For the given tree, traversals would be: Preorder: A,B,D,E,C,F; Inorder: D,B,E,A,C,F; Postorder: D,E,B,F,C,A

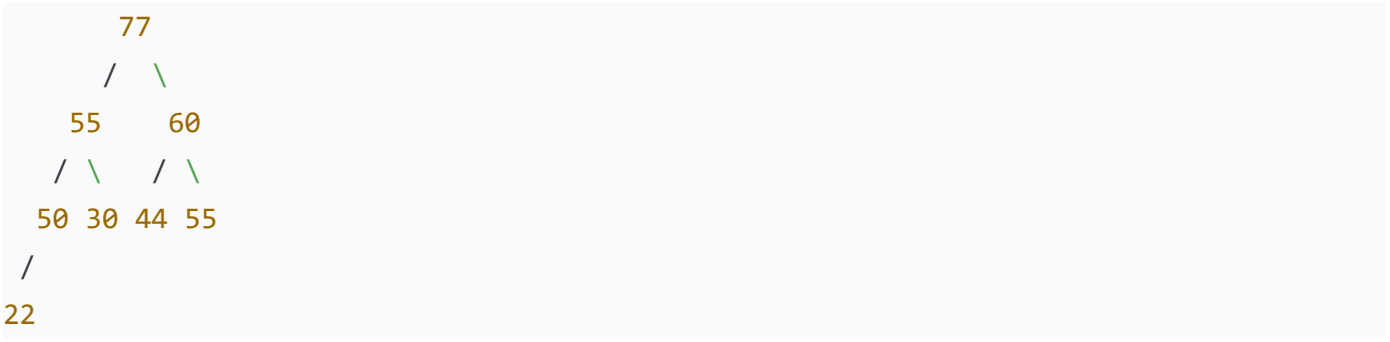
7. What is Heap tree? Construct heap tree (maxheap) using following data: 44, 30, 50, 22, 60, 55, 77, 55.

A heap tree is a complete binary tree that satisfies the heap property. In a max heap, the parent node is always greater than or equal to its children. In a min heap, the parent is smaller than or equal to its children.

Max Heap Construction Process:

Insert 44: [44] Insert 30: [44, 30] Insert 50: [50, 30, 44] (heapify up) Insert 22: [50, 30, 44, 22] Insert 60: [60, 50, 44, 22, 30] (heapify up) Insert 55: [60, 50, 55, 22, 30, 44] (heapify up) Insert 77: [77, 50, 60, 22, 30, 44, 55] (heapify up) Insert 55: [77, 55, 60, 50, 30, 44, 55, 22] (heapify up)

Final Max Heap:



8. Construct the binary search tree (BST) from the following elements: 45, 20, 80, 40, 10, 90, 70 Also, show pre-order and post-order traversal for the same.

BST Construction: Insert elements following BST property (left < root < right):

Construction Steps:

1. Insert 45 (root)
2. Insert 20 (left of 45)
3. Insert 80 (right of 45)
4. Insert 40 (right of 20)
5. Insert 10 (left of 20)
6. Insert 90 (right of 80)
7. Insert 70 (left of 80)

Pre-order Traversal (Root-Left-Right): 45, 20, 10, 40, 80, 70, 90

Post-order Traversal (Left-Right-Root): 10, 40, 20, 70, 90, 80, 45

The BST maintains the property that for any node, all values in the left subtree are smaller and all values in the right subtree are larger.

9. What is AVL tree? Explain all the rotations in AVL tree. Construct AVL tree for the following data: 1, 2, 3, 4, 5, 6

An AVL tree is a self-balancing BST where the height difference between left and right subtrees is at most 1 for every node.

Rotation Types:

1. Left Rotation (LL): When right subtree is heavy **2. Right Rotation (RR):** When left subtree is heavy **3. Left-Right Rotation (LR):** Left rotation on left child, then right rotation on root **4. Right-Left Rotation (RL):** Right rotation on right child, then left rotation on root

AVL Tree Construction:

Insert 1: [1] Insert 2: [1, 2] Insert 3: [2, 1, 3] (Left rotation needed) Insert 4: [2, 1, 3, 4] Insert 5: [2, 1, 4, 3, 5] (Right-Left rotation) Insert 6: [4, 2, 5, 1, 3, 6] (Left rotation)

Final AVL Tree:

All nodes maintain balance factor between -1 and 1.

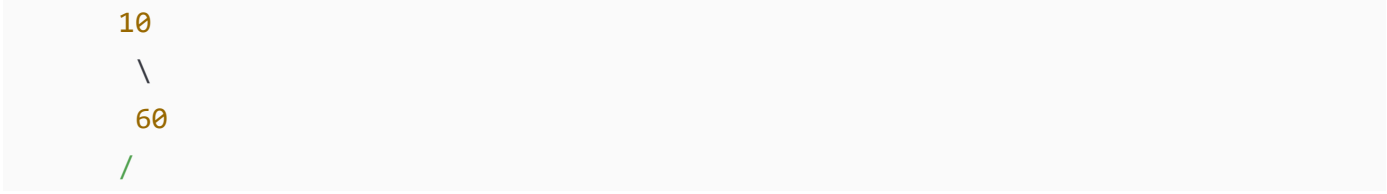
10. Construct the binary search tree from the following elements: 10, 60, 40, 28, 14, 50, 6.

BST Construction Process:

Starting with 10 as root, insert each element following BST property:

1. Insert 10 (root)
2. Insert 60 (right of 10)
3. Insert 40 (left of 60)
4. Insert 28 (left of 40)
5. Insert 14 (left of 28)
6. Insert 50 (right of 40)
7. Insert 6 (left of 14)

Final BST:



```
      40
     / \
   28   50
   /
  14
 /
6
```

Properties:

- Left subtree values < root value
- Right subtree values > root value
- In-order traversal gives sorted sequence: 6, 10, 14, 28, 40, 50, 60
- Tree is right-skewed, showing importance of balanced trees like AVL

11. Construct the binary search tree from the following elements: 5, 2, 8, 4, 1, 9, 7 Also show preorder, inorder and postorder traversal for the same.

BST Construction:

Insert elements following BST property:

```
      5
     / \
    2   8
   / \ / \
  1  4 7  9
```

Construction Steps:

1. Insert 5 (root)
2. Insert 2 (left of 5)
3. Insert 8 (right of 5)
4. Insert 4 (right of 2)
5. Insert 1 (left of 2)
6. Insert 9 (right of 8)
7. Insert 7 (left of 8)

Traversals:

Preorder (Root-Left-Right): 5, 2, 1, 4, 8, 7, 9

Inorder (Left-Root-Right): 1, 2, 4, 5, 7, 8, 9

Postorder (Left-Right-Root): 1, 4, 2, 7, 9, 8, 5

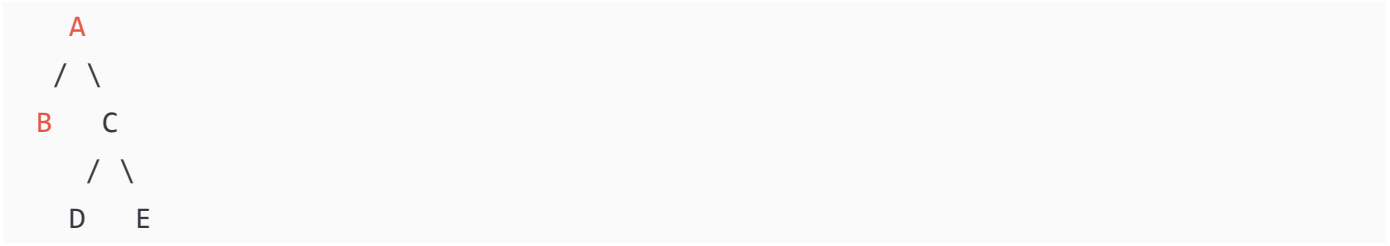
Note: Inorder traversal of BST always gives elements in sorted order, which is a key property used for validation.

12. Define Binary Tree. What are its types? Explain with suitable figures.

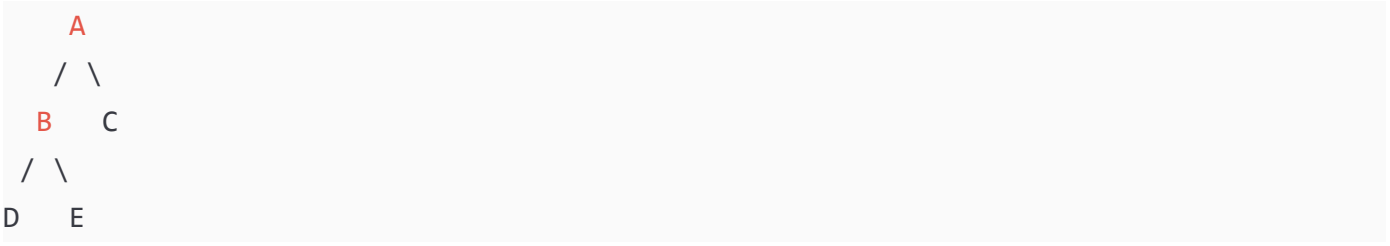
A binary tree is a hierarchical data structure in which each node has at most two children, commonly referred to as left child and right child. It consists of nodes connected by edges, with one node designated as the root.

Types of Binary Trees:

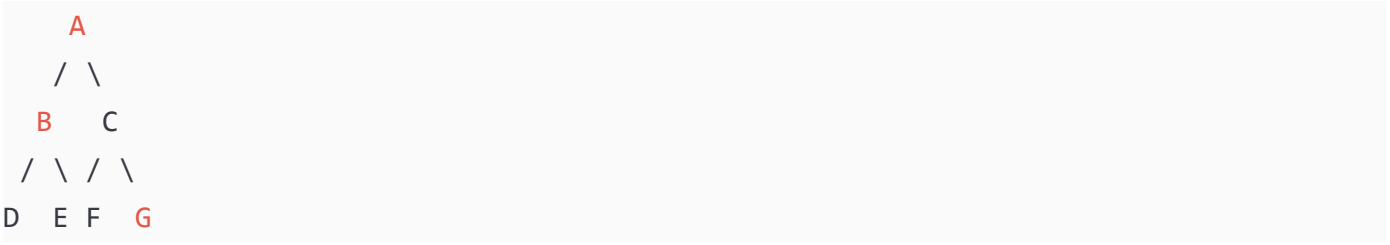
1. Full Binary Tree: Every node has either 0 or 2 children.



2. Complete Binary Tree: All levels are filled except possibly the last level, which is filled from left to right.



3. Perfect Binary Tree: All internal nodes have two children and all leaves are at the same level.



4. Balanced Binary Tree: Height difference between left and right subtrees is at most 1 for every node.

5. Degenerate Tree: Each internal node has only one child (essentially a linked list).

13. Define the following terms with respect to Trees:

i) Root: The topmost node of a tree that has no parent. It serves as the starting point for accessing all other nodes in the tree.

ii) Subtree: A tree formed by a node and all its descendants. Any node in a tree can be considered as the root of its subtree.

iii) Level of node: The distance from the root to a particular node. The root is at level 0, its children are at level 1, and so on.

iv) Depth of Tree: The maximum level of any node in the tree, also known as the height of the tree. It represents the longest path from root to any leaf.

v) Siblings: Nodes that share the same parent are called siblings. They exist at the same level in the tree structure.

Example:

```
      A    (Root, Level 0)
     / \
    B   C  (Siblings, Level 1)
   / \
  D   E    (Siblings, Level 2)
```

Here, A is root, B and C are siblings, depth is 2, and subtree rooted at B contains nodes B, D, E.

14. Explain with suitable example how binary tree can be represented using: i) Array ii) Linked List

i) Array Representation: Binary trees can be stored in arrays using level-order traversal. For a node at index i:

- Left child is at index 2i+1
- Right child is at index 2i+2
- Parent is at index (i-1)/2

Example Tree:

```
      A
     / \
    B   C
   / \   \
  D   E   F
```

Array: [A, B, C, D, E, -, F] **Index:** [0, 1, 2, 3, 4, 5, 6]

Advantages: Simple indexing, no extra memory for pointers Disadvantages: Memory wastage for incomplete trees

ii) Linked List Representation: Each node is represented as a structure containing data and pointers to left and right children.

Structure:

```
struct Node {
    char data;
    struct Node* left;
    struct Node* right;
};
```

Advantages: Dynamic memory allocation, no memory wastage Disadvantages: Extra memory for pointers, complex implementation

15. What is binary search tree? How to binary search tree in array.

A Binary Search Tree (BST) is a binary tree that maintains the following property: for every node, all values in the left subtree are smaller than the node's value, and all values in the right subtree are greater than the node's value.

BST Properties:

- Left subtree contains only nodes with values less than root
- Right subtree contains only nodes with values greater than root
- Both left and right subtrees must also be BSTs
- No duplicate values are allowed

Array Representation of BST: BSTs can be stored in arrays using level-order traversal, similar to complete binary trees. However, this may lead to sparse arrays for unbalanced BSTs.

Example BST:

```
      5
     / \
    3   8
   / \ / \
  2  4 7  9
```

Array Representation: [5, 3, 8, 2, 4, 7, 9]

Advantages: In-order traversal gives sorted sequence, efficient searching (O(log n) average case)
Applications: Database indexing, expression parsing, priority queues

16. Explain following types of tree: a. Binary tree b. Full binary Tree c. Complete Binary Tree d. Strict Binary Tree

a. Binary Tree: A tree where each node has at most two children (left and right child). It's the general form with no specific constraints on structure.

```
   A
  / \
```

```
  B    C
 /      \
D        E
```

b. Full Binary Tree (Also called Proper Binary Tree): Every node has either 0 or 2 children. No node has exactly one child.

```
    A
   / \
  B    C
 / \
D   E
```

c. Complete Binary Tree: All levels are completely filled except possibly the last level, which is filled from left to right.

```
    A
   / \
  B    C
 / \ /
D  E F
```

d. Strict Binary Tree (Same as Full Binary Tree): Every internal node has exactly two children. Leaf nodes have no children.

Key Differences:

- Binary tree: General case, any structure allowed
- Full/Strict: Internal nodes must have exactly 2 children
- Complete: Levels filled left to right, used in heaps

Unit 6: Graph

1. Draw the picture of the directed graph specified below and obtain adjacency matrix and adjacency list.

Given: G=(V,E) V(G)={1,2,3,4,5,6} E(G)={(1,2),(2,3),(3,4),(5,1),(5,6),(2,6),(1,6),(4,6),(2,4)}

Directed Graph:

```
1 → 2 → 3 → 4
↑   ↓   ↓   ↓
5 → 6 ← ← ← ←
```

i) Adjacency Matrix:

```
  1 2 3 4 5 6
1 0 1 0 0 0 1
2 0 0 1 1 0 1
3 0 0 0 1 0 0
4 0 0 0 0 0 1
5 1 0 0 0 0 1
6 0 0 0 0 0 0
```

ii) Adjacency List:

```
1: [2, 6]
2: [3, 4, 6]
3: [4]
4: [6]
5: [1, 6]
6: []
```

Each row in adjacency matrix represents outgoing edges from that vertex. Adjacency list shows all vertices directly reachable from each vertex.

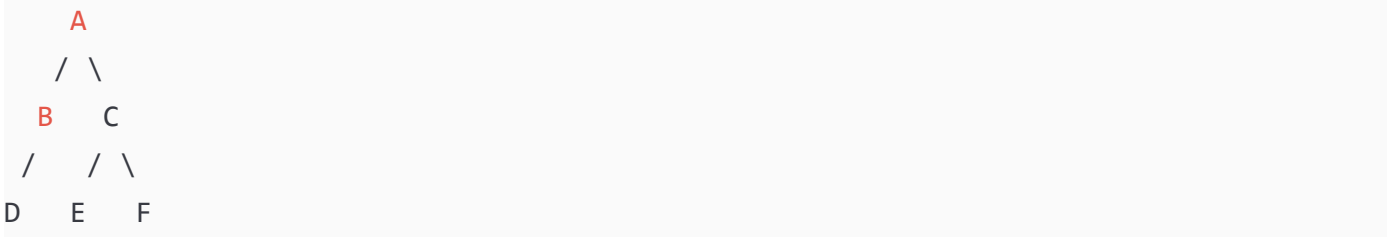
2. Explain any one following term of Graph with help of suitable example: i) Breadth First Search ii) Depth First Search

Breadth First Search (BFS): BFS explores graph level by level, visiting all neighbors of current vertex before moving to next level. It uses a queue data structure and finds shortest path in unweighted graphs.

Algorithm:

1. Start from source vertex, mark it visited
2. Add source to queue
3. While queue is not empty:
    - Dequeue a vertex
    - Visit all unvisited neighbors
    - Mark neighbors as visited and enqueue them

Example Graph:



BFS Traversal starting from A:

- Visit A, enqueue B, C
- Visit B, enqueue D
- Visit C, enqueue E, F
- Visit D (no unvisited neighbors)
- Visit E, F

Order: A, B, C, D, E, F

Applications: Shortest path, level-order traversal, web crawling, social networking features.

3. What is graph? Explain all types of graph with the help of example.

A graph is a collection of vertices (nodes) connected by edges. It's represented as G = (V, E) where V is set of vertices and E is set of edges.

Types of Graphs:

1. Directed Graph: Edges have direction (arrows).

```
A → B → C
```

2. Undirected Graph: Edges have no direction.

```
A — B — C
```

3. Weighted Graph: Edges have associated weights/costs.

```
A —5— B —3— C
```

4. Simple Graph: No self-loops or multiple edges between same vertices.

5. Complete Graph: Every pair of vertices is connected by an edge.

```
A — B
|\ /|
| X |
|/ \|
C — D
```

6. Connected Graph: Path exists between every pair of vertices.

7. Cyclic Graph: Contains at least one cycle.

8. Acyclic Graph: Contains no cycles (trees are acyclic connected graphs).

Applications: Social networks, transportation systems, computer networks, dependency resolution.

Data Structures Unit 6 Continued Answers

4. What is graph? Explain depth first traversal (DFS) in graph with example. (Continued)

Depth First Search (DFS): DFS explores graph by going as deep as possible along each branch before backtracking. It uses a stack (either explicit or recursion stack) and is useful for detecting cycles and topological sorting.

Algorithm:

1. Start from source vertex, mark it visited
2. For each unvisited neighbor of current vertex:
    - Recursively apply DFS
    - Mark neighbor as visited

Example Graph:

DFS Traversal starting from A:

- Visit A
- Go to B, visit B
- Go to D, visit D (no unvisited neighbors, backtrack)
- Backtrack to A, go to C, visit C
- Go to E, visit E (backtrack)
- Go to F, visit F

Order: A, B, D, C, E, F

Applications: Cycle detection, topological sorting, pathfinding, maze solving, strongly connected components.

5. Explain representation of graph using adjacency matrix and adjacency list with example. List applications of graph.

Adjacency Matrix: A 2D array where entry (i,j) is 1 if there's an edge from vertex i to vertex j, otherwise 0.

Example Graph:

```
  1 — 2
  |   |
  3 — 4
```

Adjacency Matrix:

```
  1 2 3 4
1 0 1 1 0
2 1 0 0 1
3 1 0 0 1
4 0 1 1 0
```

Adjacency List: Array of lists where each index contains list of adjacent vertices.

Adjacency List:

```
1: [2, 3]
2: [1, 4]
3: [1, 4]
4: [2, 3]
```

Comparison:

- Matrix: O(V²) space, O(1) edge lookup, O(V²) for sparse graphs
- List: O(V+E) space, O(degree) edge lookup, efficient for sparse graphs

Applications of Graphs: Social networks, GPS navigation, computer networks, web page ranking, dependency resolution, circuit design, scheduling problems, recommendation systems.

6. Explain Breadth First Search (BFS) traversal in graph with example.

Breadth First Search (BFS): BFS explores graph level by level, visiting all vertices at current depth before moving to vertices at next depth level. It uses a queue data structure.

Algorithm:

```
BFS(Graph, start_vertex):
1. Create queue Q and mark start_vertex as visited
2. Enqueue start_vertex to Q
3. While Q is not empty:
   a. Dequeue vertex v from Q
   b. Process v
   c. For each unvisited neighbor u of v:
      - Mark u as visited
      - Enqueue u to Q
```

Example Graph:

```
    1
   / \
  2   3
 /|   |\
4 5   6 7
```

BFS Traversal from vertex 1:

- Start: Queue = [1], Visited = {1}
- Process 1: Queue = [2,3], Visited = {1,2,3}
- Process 2: Queue = [3,4,5], Visited = {1,2,3,4,5}
- Process 3: Queue = [4,5,6,7], Visited = {1,2,3,4,5,6,7}
- Process 4,5,6,7: Queue becomes empty

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

Applications: Shortest path in unweighted graphs, level-order processing, web crawling.

7. What is Graph? With suitable example describe how graph is represented using adjacency list.

A graph is a collection of vertices (nodes) and edges that connect pairs of vertices. It's a fundamental data structure used to model relationships between objects.

Graph Components:

- Vertices (V): Set of nodes
- Edges (E): Set of connections between vertices
- Represented as G = (V, E)

Adjacency List Representation: An array of linked lists where each index represents a vertex and contains a list of its adjacent vertices.

Example Graph:

```
   A
  /|\
 B | C
  \|/
   D
```

Vertices: {A, B, C, D} **Edges:** {(A,B), (A,C), (A,D), (B,D), (C,D)}

Adjacency List:

```
A: [B, C, D]
B: [A, D]
C: [A, D]
D: [A, B, C]
```

Advantages:

- Space efficient for sparse graphs: O(V + E)
- Easy to iterate over neighbors
- Dynamic addition/deletion of vertices

Disadvantages:

- Checking if edge exists takes O(degree) time
- Not cache-friendly for dense graphs

8. What do you mean by adjacency matrix and adjacency list? Give the adjacency matrix and adjacency list for the graph shown below:

Adjacency Matrix: A 2D square matrix of size V×V where V is number of vertices. Entry (i,j) is 1 if there's an edge between vertex i and vertex j, otherwise 0. For weighted graphs, it stores the weight instead of 1.

Adjacency List: An array of lists where index i contains a list of all vertices adjacent to vertex i. More space-efficient for sparse graphs.

Properties:

- Adjacency Matrix: $O(V^2)$ space, O(1) edge lookup
- Adjacency List: O(V+E) space, O(degree) edge lookup

For Undirected Graphs:

- Matrix is symmetric
- Each edge (u,v) appears twice in adjacency list

For Directed Graphs:

- Matrix may not be symmetric
- Edge (u,v) appears only in u's adjacency list

Trade-offs:

- Use matrix for dense graphs or when frequent edge queries needed
- Use list for sparse graphs or when memory is constrained

Note: The actual graph image was not provided in the question, so specific matrix and list cannot be generated.

10. Define with an example: i) Undirected Graph ii) Directed Graph iii) Weighted Graph

i) Undirected Graph: A graph where edges have no direction. The edge (u,v) is the same as (v,u), meaning you can traverse in both directions.

Example:

```
A — B
|   |
C — D
```

Edges: {(A,B), (A,C), (B,D), (C,D)} Can move from A to B and also from B to A.

ii) Directed Graph (Digraph): A graph where edges have direction, indicated by arrows. Edge (u,v) allows movement only from u to v.

Example:

```
A → B
↓   ↓
C → D
```

Edges: {(A,B), (A,C), (B,D), (C,D)} Can move from A to B but not from B to A.

iii) Weighted Graph: A graph where each edge has an associated numerical value (weight/cost) representing distance, time, or cost.

Example:

```
A –5– B
|     |
3     2
|     |
C –4– D
```

Edges with weights: {(A,B,5), (A,C,3), (B,D,2), (C,D,4)}

Applications: Road networks (distances), flight routes (costs), network latency (time).

11. What is graph? Explain following terminology of graph with example:

A graph is a mathematical structure consisting of vertices (nodes) connected by edges, used to model pairwise relationships between objects.

a. Degree of Node: The number of edges incident to a vertex. In directed graphs:

- In-degree: Number of incoming edges
- Out-degree: Number of outgoing edges

Example:

```
A — B — C
    |
    D
```

Degree of A = 1, Degree of B = 3, Degree of C = 1, Degree of D = 1

b. Connected Graph: An undirected graph where there exists a path between every pair of vertices. All vertices are reachable from any starting vertex.

Example of Connected Graph:

```
A — B
|   |
C — D
```

c. Complete Graph: A graph where every pair of distinct vertices is connected by exactly one edge. A complete graph with n vertices has n(n-1)/2 edges.

Example (Complete graph with 4 vertices):

```
A — B
|\ /|
| X |
|/ \|
C — D
```

Every vertex is directly connected to every other vertex.

12. Draw the picture of the directed graph specified below and obtain adjacency matrix and adjacency list.

Given: G=(V,E) V(G)={A, B, C, D, E} E(G)={(A,B), (A,C), (A,D), (B,C), (D,C), (D,E), (E,C)}

Directed Graph:

```
A → B → C ← E
↓       ↗   ↑
D ————————↗
```

Adjacency Matrix:

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

Adjacency List:

```
A: [B, C, D]
B: [C]
C: []
D: [C, E]
E: [C]
```

Properties:

- Total vertices: 5
- Total edges: 7
- Vertex C has highest in-degree (4)
- Vertex C has out-degree 0 (sink vertex)
- Vertices B and E have out-degree 1

13. What is sparse matrix? Explain it in details.

A sparse matrix is a matrix in which most of the elements are zero. It's considered sparse when the number of zero elements is significantly greater than non-zero elements, typically when zeros comprise more than half the matrix.

Characteristics:

- Large number of zero elements
- Wasteful to store all elements in standard 2D array
- Special storage techniques used to save memory
- Common in scientific computing, graphics, networks

Example Sparse Matrix:

```
[0  0  3  0  0]
[0  0  0  0  4]
[0  2  0  0  0]
[1  0  0  0  0]
[0  0  0  5  0]
```

Only 5 non-zero elements out of 25 total elements.

Storage Methods:

1. **Array Representation:** Store only non-zero elements with their positions
2. **Linked List:** Chain of nodes containing row, column, and value
3. **Coordinate Format (COO):** Arrays for row indices, column indices, and values

Advantages of Special Storage:

- Reduced memory usage
- Faster operations on non-zero elements
- Efficient for large matrices with few non-zero elements

Applications: Finite element analysis, image processing, graph algorithms, machine learning.

14. Explain linked representation of sparse matrix.

Linked representation of sparse matrices uses linked lists to store only non-zero elements, saving significant memory space compared to standard 2D array representation.

Node Structure:

```
struct Node {
    int row;
    int col;
    int value;
    struct Node* next;
};
```

Representation Methods:

1. Single Linked List: All non-zero elements stored in a single linked list, typically ordered by row-major or column-major sequence.

2. Array of Linked Lists: Separate linked list for each row, making row-wise operations efficient.

Example Sparse Matrix:

```
[0  0  3  0]
[0  4  0  0]
[2  0  0  5]
```

Linked Representation (Row-wise):

```
Row 0: (0,2,3) → NULL
Row 1: (1,1,4) → NULL
Row 2: (2,0,2) → (2,3,5) → NULL
```

Advantages:

- Dynamic memory allocation
- Memory efficient for very sparse matrices
- Easy insertion and deletion of elements
- Suitable when sparsity pattern changes frequently

Disadvantages:

- Extra memory for storing pointers
- Sequential access required
- Complex implementation compared to arrays