**CS3301 – Data Structures**

**UNIT I - LISTS (Data Structures)**

**Question 1:**

**Explain the Array-based and Linked list implementations of the List ADT. Compare their advantages and disadvantages.**

**Answer:**

**List ADT (Abstract Data Type):** A list is a collection of elements arranged in a linear order. List ADT supports operations such as insertion, deletion, traversal, and searching.

**1. Array-based List**

**:**

* Uses a contiguous block of memory to store elements.
* Elements are accessed via index.

**Operations:**

* Insertion at end: O(1)
* Insertion at beginning or middle: O(n)
* Deletion: O(n)
* Access: O(1)

**Advantages:**

* Fast access using index.
* Simple implementation.

**Disadvantages:**

* Fixed size.
* Inefficient insertions and deletions in the middle.

**Example Code (in C-like pseudocode):**

int arr[100];

arr[0] = 10;

arr[1] = 20;

**2. Linked List Implementation:**

* Consists of nodes where each node contains data and a pointer to the next node.
* Types: Singly Linked List, Doubly Linked List, Circular Linked List.

**Operations:**

* Insertion at beginning: O(1)
* Insertion at end: O(n) or O(1) with tail pointer
* Deletion: O(n)
* Access: O(n)

**Advantages:**

* Dynamic size.
* Efficient insertion/deletion.

**Disadvantages:**

* Slower access time.
* Extra memory for pointers.

**Comparison Table:**

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| Memory Allocation | Static | Dynamic |
| Access Time | O(1) | O(n) |
| Insertion/Deletion | Inefficient | Efficient |
| Memory Overhead | Less | More (pointers) |

**Question 2:**

**Describe the Singly Linked List, Circularly Linked List, and Doubly Linked List with their advantages, disadvantages, and applications.**

**Answer:**

**1. Singly Linked List:**

* Each node contains data and a pointer to the next node.
* Last node points to NULL.

**Operations:** Insertion, deletion, traversal.

**Advantages:**

* Easy to implement.
* Efficient for insertion/deletion.

**Disadvantages:**

* Cannot traverse backward.
* Time-consuming search.

**Applications:**

* Dynamic memory allocation.
* Implementation of stacks and queues.

**2. Circularly Linked List:**

* Last node points to the first node.
* No NULL in the list.

**Advantages:**

* Efficient circular traversal.
* Useful in applications like round-robin scheduling.

**Disadvantages:**

* Complexity in terminating traversal.

**Applications:**

* Multiplayer games.
* CPU scheduling.

**3. Doubly Linked List:**

* Each node contains data, a pointer to the next node, and a pointer to the previous node.

**Advantages:**

* Bi-directional traversal.
* Efficient deletion of nodes.

**Disadvantages:**

* More memory usage.
* Complex implementation.

**Applications:**

* Browser history navigation.
* Undo-redo functionality.

**Question 3:**

**What is Polynomial ADT? How is it implemented using a Linked List? Write code to add two polynomials.**

**Answer:**

**Polynomial ADT:**

* A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients.
* Example: 5x^3 + 4x^2 + 2x + 7

**Node Structure:**

struct PolyNode {

int coeff;

int exp;

struct PolyNode\* next;

};

**Implementation Steps:**

1. Represent each term as a node.
2. Add like terms by comparing exponents.

**Code to Add Two Polynomials (Pseudocode):**

struct PolyNode\* addPoly(struct PolyNode\* p1, struct PolyNode\* p2) {

struct PolyNode\* result = NULL;

struct PolyNode\*\* lastPtr = &result;

while (p1 && p2) {

if (p1->exp > p2->exp) {

\*lastPtr = p1;

p1 = p1->next;

} else if (p1->exp < p2->exp) {

\*lastPtr = p2;

p2 = p2->next;

} else {

p1->coeff += p2->coeff;

\*lastPtr = p1;

p1 = p1->next;

p2 = p2->next;

}

lastPtr = &((\*lastPtr)->next);

}

\*lastPtr = (p1) ? p1 : p2;

return result;

}

**Advantages:**

* Dynamic handling of terms.
* Efficient operations with sparse polynomials.

**Applications:**

* Computer algebra systems.
* Symbolic mathematics and compiler design.

**UNIT II: STACKS AND QUEUES - 16 MARK QUESTIONS AND ANSWERS**

**Question 1: Explain Stack ADT in detail with operations and applications.**

**Answer:**

A **Stack** is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle. The element added last is accessed first.

**Operations on Stack:**

1. **Push(x):** Adds an element x to the top of the stack.
2. **Pop():** Removes the topmost element of the stack.
3. **Peek()/Top():** Returns the topmost element without removing it.
4. **isEmpty():** Checks whether the stack is empty.
5. **isFull():** (For array-based implementation) Checks if the stack is full.

**Array-based Implementation:**

* Use an array and an integer top to indicate the current top element.
* Initialize top = -1.

**Linked List Implementation:**

* Each node has a data and a next pointer.
* Push: Insert at the beginning.
* Pop: Delete the node from the beginning.

**Applications of Stack:**

1. **Reversing a string.**
2. **Checking for balanced parentheses.**
3. **Function call management in recursion.**
4. **Expression evaluation and conversion.**
5. **Undo features in text editors.**

**Question 2: Explain the conversion of infix to postfix expression using a stack with an example.**

**Answer:**

**Infix Expression:** Operators are written between operands: A + B

**Postfix Expression:** Operators are written after operands: AB+

**Algorithm to Convert Infix to Postfix:**

1. Initialize an empty stack for operators.
2. Scan the infix expression from left to right.
3. If the character is an operand, add it to the postfix expression.
4. If the character is (, push it onto the stack.
5. If the character is ), pop and add to postfix until ( is found.
6. If the character is an operator, pop all higher or equal precedence operators from the stack and then push the current operator.
7. After the end of the expression, pop all remaining operators from the stack.

**Example:** Infix: (A+B)\*C

* Step 1: Read ( → push
* Step 2: Read A → output: A
* Step 3: Read + → push
* Step 4: Read B → output: AB
* Step 5: Read ) → pop + → output: AB+, pop (
* Step 6: Read \* → push
* Step 7: Read C → output: AB+C
* Step 8: End → pop \* → output: AB+C\*

**Postfix: AB+C\***

**Question 3: Explain Queue ADT and its operations. Describe circular queue and its applications.**

**Answer:**

A **Queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. The element inserted first is removed first.

**Operations on Queue:**

1. **Enqueue(x):** Add element x at the rear.
2. **Dequeue():** Remove and return element from the front.
3. **isEmpty():** Check if queue is empty.
4. **isFull():** (For array) Check if queue is full.
5. **Front():** Return front element.

**Types of Queue:**

* **Simple Queue**: Standard FIFO behavior.
* **Circular Queue**: Last position connects to the first.
* **DeQueue (Double-ended Queue):** Insert/delete from both ends.

**Circular Queue:**

* Prevents wastage of space.
* Rear = (rear + 1) % size
* Front = (front + 1) % size

**Example:** If size = 5, and front = 4, rear = 4, inserting a new element sets rear = 0 (wrap around).

**Applications of Queue:**

1. **CPU scheduling.**
2. **Printer spooling.**
3. **Handling of requests in web servers.**
4. **Simulation of real-world queues like bank lines.**

**UNIT III - TREES**

**1. Explain the Tree Abstract Data Type (ADT) and various types of tree traversals with examples.**

**Answer:**

A **Tree** is a non-linear data structure consisting of nodes connected by edges. It represents hierarchical relationships between elements.

**Tree ADT Properties:**

* Root: Topmost node.
* Parent: A node with children.
* Child: A node descended from a parent.
* Leaf: A node with no children.
* Degree: Number of children of a node.
* Height: Longest path from root to a leaf.

**Tree Traversals:**

Tree traversal is the process of visiting each node in a tree data structure, exactly once.

**1. Inorder Traversal (Left, Root, Right):**

# Example Tree:

# A

# / \

# B C

# / \

# D E

Inorder: D B E A C

**2. Preorder Traversal (Root, Left, Right):**

Preorder: A B D E C

**3. Postorder Traversal (Left, Right, Root):**

Postorder: D E B C A

Applications:

* Preorder: Used in expression trees and copying trees.
* Inorder: Gives sorted order in Binary Search Tree.
* Postorder: Used for deleting trees.

**2. Explain Binary Search Tree (BST) ADT and operations: insertion, deletion, and search with examples.**

**Answer:**

A **Binary Search Tree (BST)** is a binary tree where each node has a key and satisfies:

* Left child has a key < root.
* Right child has a key > root.

**Operations:**

**1. Insertion:**

* Start from root and compare the key.
* Place the node in left/right based on value.

**2. Search:**

* If value < root, go left; if > root, go right.
* If match found, return node.

**3. Deletion:**

* Case 1: Leaf Node – Remove it.
* Case 2: One Child – Replace with child.
* Case 3: Two Children – Replace with in-order predecessor/successor.

**Example BST:**

50

/ \

30 70

/ \ / \

20 40 60 80

Insert 25: goes to left of 30. Delete 70: replace with 80.

BST is useful for efficient searching, insertion, and deletion: O(log n) average-case.

**3. Explain AVL Trees. How does rotation help in balancing AVL Trees?**

**Answer:**

AVL (Adelson-Velsky and Landis) Tree is a self-balancing Binary Search Tree.

**Properties:**

* Balance Factor = height(left subtree) - height(right subtree)
* Valid AVL Tree: balance factor in {-1, 0, +1}

**Rotations:**

Used to rebalance the tree after insertion/deletion.

**1. Left Rotation:**

* When node is inserted in the right-right position.

**2. Right Rotation:**

* When node is inserted in left-left position.

**3. Left-Right Rotation:**

* Inserted in left-right. First left rotation on left child, then right on root.

**4. Right-Left Rotation:**

* Inserted in right-left. First right on right child, then left on root.

**Example:**

Insert in AVL Tree: 10, 20, 30

10

\

20

\

30

Not balanced => Left Rotation

Result:

20

/ \

10 30

AVL Trees maintain O(log n) complexity.

**4. What is an Expression Tree? Construct and evaluate an expression tree for a given infix expression.**

**Answer:**

An **Expression Tree** is a binary tree representing expressions. Internal nodes are operators, leaves are operands.

**Example Infix Expression:**

(A + B) \* (C - D)

**Postfix (for tree construction):**

A B + C D - \*

**Tree:**

\*

/ \

+ -

/ \ / \

A B C D

**Evaluation:**

Traverse postorder:

* Evaluate +: A + B
* Evaluate -: C - D
* Evaluate \*: (A + B) \* (C - D)

**Applications:**

* Used in compilers and calculators to evaluate expressions.

**5. Explain the concept of Heaps and Priority Queues. How is a binary heap used to implement a priority queue?**

**Answer:**

**Heap:**

A Heap is a complete binary tree with the heap property:

* **Max Heap**: Parent >= Children
* **Min Heap**: Parent <= Children

**Priority Queue:**

A priority queue assigns priority to elements and serves highest priority first. Implemented using Binary Heaps.

**Operations:**

1. **Insert** – Add to the end and heapify up.
2. **Delete Max/Min** – Replace root with last and heapify down.

**Example Min Heap:**

10

/ \

20 15

/ \

30 40

**Applications:**

* CPU Scheduling
* Graph algorithms like Dijkstra’s
* Event Simulation

Time Complexity:

* Insertion: O(log n)
* Deletion: O(log n)

**UNIT IV: MULTIWAY SEARCH TREES AND GRAPHS – 16 MARK QUESTION AND ANSWER**

**Question:** Explain in detail the following:

1. B-Trees
2. B+ Trees
3. Graph representation techniques
4. BFS and DFS traversals
5. Minimum Spanning Tree (Prim's and Kruskal's algorithms)

**Answer:**

**1. B-Trees:** A B-Tree is a self-balancing search tree in which nodes can have multiple children. It maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. It is widely used in databases and file systems.

**Properties of B-Trees (Order m):**

* Each node can have at most m children.
* Each node can have at most m-1 keys.
* All leaves are at the same level.
* A non-leaf node with k children contains k-1 keys.

**Operations:**

* **Search**: Starts from the root and moves down the tree.
* **Insert**: Add the key and if the node overflows, split it.
* **Delete**: Remove the key and merge nodes if necessary.

**Advantages:**

* Balanced tree with fewer levels.
* Efficient disk access due to block-level storage.

**2. B+ Trees:** A B+ Tree is an extension of the B-Tree which also maintains sorted data. In B+ Trees, all values are found at the leaf level and internal nodes only store keys for navigation.

**Properties:**

* Internal nodes only store keys.
* Leaf nodes store keys and pointers to actual data.
* Leaves are linked to form a linked list (used for range queries).

**Advantages over B-Trees:**

* Better for range queries.
* Faster search as internal nodes do not hold data.
* Used in databases and file indexing systems.

**3. Graph Representation Techniques:** Graphs can be represented using:

**a. Adjacency Matrix:**

* A 2D array where matrix[i][j] = 1 if there is an edge from vertex i to vertex j.
* Space complexity: O(V^2)

**b. Adjacency List:**

* Each vertex maintains a list of adjacent vertices.
* Space complexity: O(V + E)

**Comparison:**

* Matrix is suitable for dense graphs.
* List is better for sparse graphs.

**4. Graph Traversals:**

**a. Breadth-First Search (BFS):**

* Uses a queue to explore neighbors level by level.
* Useful for finding the shortest path in unweighted graphs.

**Algorithm:**

1. Start at a vertex, mark it visited.
2. Enqueue it.
3. While the queue is not empty:
   * Dequeue a vertex.
   * Visit all its unvisited neighbors and enqueue them.

**b. Depth-First Search (DFS):**

* Uses a stack (recursion) to explore as far as possible along each branch.

**Algorithm:**

1. Start at a vertex, mark it visited.
2. Recursively visit unvisited neighbors.

**5. Minimum Spanning Tree (MST):** A minimum spanning tree connects all the vertices in a graph with the minimum total edge weight and no cycles.

**a. Prim’s Algorithm:**

* Start with one vertex and add the smallest edge connecting to an unvisited vertex.
* Use a priority queue.
* Time Complexity: O(E log V)

**b. Kruskal’s Algorithm:**

* Sort all edges in ascending order.
* Use Disjoint Set (Union-Find) to avoid cycles.
* Add edge if it doesn’t form a cycle.
* Time Complexity: O(E log E)

**Applications:**

* Network design.
* Circuit design.

**UNIT V: SEARCHING, SORTING AND HASHING TECHNIQUES - 16 MARK QUESTIONS AND ANSWERS**

**Q1. Explain different sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, and Shell Sort. Provide example steps.**

**Answer:**

1. **Bubble Sort**:
   * Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
   * Time Complexity: O(n^2)
   * Example:
     + Input: [5, 3, 8, 4, 2]
     + Pass 1: [3, 5, 4, 2, 8]
     + Pass 2: [3, 4, 2, 5, 8]
     + Pass 3: [3, 2, 4, 5, 8]
     + Pass 4: [2, 3, 4, 5, 8]
2. **Selection Sort**:
   * Selects the smallest element from the unsorted list and swaps it with the element at the beginning.
   * Time Complexity: O(n^2)
   * Example:
     + Input: [5, 3, 8, 4, 2]
     + After 1st selection: [2, 3, 8, 4, 5]
     + After 2nd: [2, 3, 8, 4, 5]
     + After 3rd: [2, 3, 4, 8, 5]
     + After 4th: [2, 3, 4, 5, 8]
3. **Insertion Sort**:
   * Builds the sorted array one element at a time.
   * Time Complexity: O(n^2)
   * Example:
     + Input: [5, 3, 8, 4, 2]
     + Step 1: [3, 5, 8, 4, 2]
     + Step 2: [3, 5, 8, 4, 2]
     + Step 3: [3, 4, 5, 8, 2]
     + Step 4: [2, 3, 4, 5, 8]
4. **Shell Sort**:
   * Improves insertion sort by comparing elements that are far apart.
   * Time Complexity: Best case O(n log n), Worst case O(n^2)
   * Example:
     + Input: [5, 3, 8, 4, 2], with gaps like 3, 1
     + Gaps reduce over iterations, and insertion sort is applied.

**Q2. Describe Merge Sort algorithm with an example. Also write its advantages.**

**Answer:**

* **Merge Sort** is a divide and conquer algorithm:
  + Divide the array into halves.
  + Sort each half recursively.
  + Merge the sorted halves.

**Steps:**

* Input: [5, 3, 8, 4, 2, 7]
  + Divide: [5, 3, 8] and [4, 2, 7]
  + Sort: [3, 5, 8] and [2, 4, 7]
  + Merge: [2, 3, 4, 5, 7, 8]

**Time Complexity**:

* Best, Average, Worst: O(n log n)

**Advantages**:

* Efficient for large datasets
* Stable sort (does not change the order of equal elements)

**Q3. Explain Linear and Binary Search with their advantages and limitations.**

**Answer:**

1. **Linear Search**:
   * Checks each element until the desired element is found.
   * Time Complexity: O(n)
   * Advantage: Works on unsorted data.
   * Limitation: Slow for large datasets.

**Example**:

* Input: [10, 20, 30, 40], Search: 30 → Found at index 2.

1. **Binary Search**:
   * Works on sorted arrays. Repeatedly divides the array in half.
   * Time Complexity: O(log n)
   * Advantage: Much faster than linear search on sorted data.
   * Limitation: Requires sorted array.

**Example**:

* Input: [10, 20, 30, 40, 50], Search: 30
  + Mid = 2 → 30 == 30 → Found

**Q4. Explain Hashing with different hash functions and collision resolution techniques.**

**Answer:**

* **Hashing**: Technique to map data to a fixed-size table using a hash function.

**Hash Functions**:

1. Division Method: h(k) = k mod m
2. Multiplication Method
3. Mid-square Method

**Collision Resolution**:

1. **Separate Chaining**: Use linked list to store multiple elements at same index.
2. **Open Addressing**:
   * Linear Probing
   * Quadratic Probing
   * Double Hashing

**Example**:

* Hash table size = 10
* Keys = [23, 43, 13, 27]
* Hash = key % 10
  + 23 % 10 = 3 → index 3
  + 43 % 10 = 3 → collision → use chaining

**Q5. What is Rehashing and Extendible Hashing? Explain with examples.**

**Answer:**

1. **Rehashing**:
   * Process of resizing the hash table when the load factor is too high.
   * All existing keys are hashed into new table with larger size.

**Example**:

* Original table size = 5, keys = [10, 20, 30, 40, 50]
* All positions filled → resize to 10
* Rehash all elements with new size

1. **Extendible Hashing**:
   * Uses a directory of pointers to buckets.
   * Directory size increases dynamically.
   * Reduces overflow chains.

**Example**:

* Start with 2-bit directory
* On overflow → double the directory size
* Buckets split dynamically

**Advantages**:

* Avoids performance degradation due to collision
* Dynamically adjusts to dataset size