**CS3301 – Data Structures**

**UNIT I – LISTS**

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

**1. Explain the List Abstract Data Type (ADT) and its Array-based Implementation.**

**Answer:** The List ADT is a sequential collection of elements where each element has a specific position. It supports operations like insertion, deletion, traversal, searching, and updating.

**Operations of List ADT:**

* Insert
* Delete
* Search
* Traverse
* Update

**Array-Based Implementation:**

* In this approach, the list is implemented using arrays.
* Elements are stored in contiguous memory locations.
* We maintain a variable n to track the current number of elements in the list.

**Advantages:**

* Random access is possible.
* Easy to implement.

**Disadvantages:**

* Fixed size.
* Insertion and deletion in the middle require shifting elements.

**Example:**

int list[100];

int n = 0;

list[n++] = 10;

**2. Describe the Singly Linked List with operations and C code.**

**Answer:** A singly linked list is a collection of nodes where each node contains data and a pointer to the next node.

**Structure:**

struct Node {

int data;

struct Node\* next;

};

**Operations:**

* **Insert at beginning:** O(1)
* **Insert at end:** O(n)
* **Delete node:** O(n)
* **Traversal:** O(n)

**Advantages:**

* Dynamic size.
* Efficient insertion and deletion.

**Disadvantages:**

* No random access.
* More memory due to pointers.

**Example - Insertion at beginning:**

void insert(struct Node\*\* head, int value) {

struct Node\* newNode = (struct Node\*) malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = \*head;

\*head = newNode;

}

**3. Compare Singly Linked List, Circular Linked List and Doubly Linked List.**

**Answer:**

| **Feature** | **Singly Linked List** | **Circular Linked List** | **Doubly Linked List** |
| --- | --- | --- | --- |
| Next pointer | Points to next node | Points to next (last points to head) | Has both next and prev |
| Back traversal | Not possible | Not possible | Possible |
| Last node | Points to NULL | Points to head | Points to NULL |
| Memory | Less (1 pointer) | Less (1 pointer) | More (2 pointers) |

**Use Cases:**

* **Singly LL**: Stack, simple list traversal
* **Circular LL**: Round-robin scheduling
* **Doubly LL**: Navigation (e.g., browser history)

**4. Explain Polynomial ADT using Linked List with an example.**

**Answer:** Polynomial ADT stores polynomials in a structured format where each term is a node.

**Structure of a term node:**

struct Term {

int coeff;

int exp;

struct Term\* next;

};

**Example: 3x^2 + 5x + 6**

* Node1: coeff=3, exp=2
* Node2: coeff=5, exp=1
* Node3: coeff=6, exp=0

**Operations:**

* **Insertion in order of exponent**
* **Addition of two polynomials**

**Addition Logic:**

* Traverse both polynomials.
* Add coefficients if exponents match.
* Insert remaining terms as-is.

**5. Explain Radix Sort and its application using an example.**

**Answer:** **Radix Sort** is a non-comparative integer sorting algorithm that sorts data with integer keys by processing individual digits.

**Steps:**

1. Find the maximum number to know the number of digits.
2. Sort each digit from least significant to most using Counting Sort.

**Example:** Given: 170, 45, 75, 90, 802, 24, 2, 66

* Pass 1 (unit place): [170, 90, 802, 2, 24, 45, 75, 66]
* Pass 2 (tens place): [802, 2, 24, 45, 66, 170, 75, 90]
* Pass 3 (hundreds place): [2, 24, 45, 66, 75, 90, 170, 802]

**Advantages:**

* Efficient for fixed-length integers.
* Time complexity: O(nk) where k is the number of digits.

**Applications:**

* Used in scenarios with large number sets with bounded digits like phone numbers, student IDs.

**UNIT II: STACKS AND QUEUES - 13 MARK QUESTIONS AND ANSWERS**

**1. Explain Stack ADT and its operations. Illustrate with examples.**

**Answer:** A **Stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle. The element that is inserted last is the first to be removed.

**Operations on Stack:**

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

**Array-Based Implementation:**

#define SIZE 100

int stack[SIZE];

int top = -1;

void push(int x) {

if(top == SIZE-1) printf("Stack Overflow\n");

else stack[++top] = x;

}

int pop() {

if(top == -1) printf("Stack Underflow\n");

else return stack[top--];

}

**Applications:**

* Reversing a string
* Expression evaluation
* Function call stack

**2. Explain the algorithm for converting Infix expression to Postfix expression using stack. Give an example.**

**Answer:** Infix expressions (like A + B) are difficult to evaluate directly, so they are converted to **Postfix (Reverse Polish Notation)** using stacks.

**Algorithm:**

1. Initialize an empty stack and an empty result string.
2. Scan the infix expression from left to right.
3. If the token is an operand, add it to the result.
4. If the token is an operator:
   * Pop operators from the stack with greater or equal precedence.
   * Push the current operator onto the stack.
5. If token is '(', push it to stack.
6. If token is ')', pop from stack to result until '(' is found.
7. Pop all remaining operators from the stack to result.

**Example:** Infix: A + B \* C Postfix: A B C \* +

**3. Describe Queue ADT and its basic operations. Differentiate between Linear and Circular Queue.**

**Answer:** A **Queue** is a linear data structure that follows the **FIFO (First In First Out)** principle.

**Operations:**

1. **Enqueue** – Adds an element at the rear end.
2. **Dequeue** – Removes an element from the front.
3. **Front/Peek** – Returns the front element.
4. **isEmpty** – Checks if the queue is empty.
5. **isFull** – Checks if the queue is full (in array implementation).

**Linear Queue Issues:**

* Insertion at rear and deletion at front
* Problem of unused space as front moves forward

**Circular Queue:**

* Overcomes the space wastage issue.
* Uses modulo arithmetic to wrap around

**Example:**

#define SIZE 5

int cq[SIZE];

int front = -1, rear = -1;

void enqueue(int val) {

if ((rear + 1) % SIZE == front) printf("Queue Full\n");

else {

if (front == -1) front = 0;

rear = (rear + 1) % SIZE;

cq[rear] = val;

}

}

**4. Explain the applications of stack with examples (Balancing Symbols and Expression Evaluation).**

**Answer:** **Applications of Stack:**

1. **Balancing Symbols:** Used to check if every opening symbol (like '(', '[', '{') has a corresponding closing symbol.

**Example:** Expression: {[()()]} → Balanced Algorithm:

* Push opening brackets
* Pop and match with closing brackets

1. **Expression Evaluation:**

* Postfix expressions can be evaluated using a stack
* Operands are pushed; operators pop two operands, apply the operator, and push result

**Example:** Postfix: 23\*54\*+9- Step-by-step stack operation yields result: 17

**5. What is DeQueue? Explain types of DeQueue and their operations.**

**Answer:** A **DeQueue (Double Ended Queue)** allows insertion and deletion at both front and rear ends.

**Types of DeQueue:**

1. **Input Restricted DeQueue** – Insertion only at one end (rear), deletion from both ends.
2. **Output Restricted DeQueue** – Deletion only at one end (front), insertion from both ends.

**Operations:**

* **InsertFront(x)**
* **InsertRear(x)**
* **DeleteFront()**
* **DeleteRear()**

**Applications:**

* Job scheduling
* Palindrome checking
* Browser history (back and forward)

**Example Code:**

#define SIZE 5

int dq[SIZE];

int front = -1, rear = -1;

void insertFront(int val) { ... }

void deleteRear() { ... }

DeQueue offers more flexibility than stack or queue and is useful in a variety of real-time applications.

**UNIT III: TREES**

**13-MARK QUESTIONS AND DETAILED ANSWERS**

**1. Explain the different tree traversal methods with examples.**

**Answer:** Tree traversal refers to the process of visiting all the nodes of a tree in a specific order. There are three main types of depth-first traversals and one breadth-first traversal:

**a) Inorder Traversal (Left, Root, Right):**

* Traverse the left subtree
* Visit the root
* Traverse the right subtree

**Example:** For a binary tree with root A, left child B, right child C:

A

/ \

B C

Inorder: B A C

**b) Preorder Traversal (Root, Left, Right):**

* Visit the root
* Traverse the left subtree
* Traverse the right subtree

Preorder: A B C

**c) Postorder Traversal (Left, Right, Root):**

* Traverse the left subtree
* Traverse the right subtree
* Visit the root

Postorder: B C A

**d) Level Order Traversal:**

* Visit nodes level by level from top to bottom, left to right using a queue.

Level Order: A B C

**2. Explain Binary Search Tree (BST) and its operations with examples.**

**Answer:** A **Binary Search Tree** is a binary tree where each node has a key, and:

* Keys in the left subtree are smaller than the root.
* Keys in the right subtree are greater than the root.

**Operations:**

**a) Insertion:**

* Insert nodes recursively by comparing keys.

**b) Deletion:**

* Case 1: Node is a leaf
* Case 2: Node has one child
* Case 3: Node has two children – replace with inorder successor/predecessor.

**c) Searching:**

* Traverse left or right subtree based on comparison.

**Example:** Insert 50, 30, 20, 40, 70, 60, 80

50

/ \

30 70

/ \ / \

20 40 60 80

**3. Explain AVL Trees and the various types of rotations used for balancing.**

**Answer:** An **AVL Tree** is a self-balancing binary search tree where the difference in height between left and right subtree (balance factor) is at most 1.

**Balance Factor (BF) = Height(Left Subtree) - Height(Right Subtree)**

**Rotations to balance AVL Trees:**

**a) Left-Left (LL) Rotation:**

* Occurs when a node is inserted into the left subtree of the left child.

**b) Right-Right (RR) Rotation:**

* Occurs when a node is inserted into the right subtree of the right child.

**c) Left-Right (LR) Rotation:**

* Left rotation followed by right rotation.

**d) Right-Left (RL) Rotation:**

* Right rotation followed by left rotation.

**Example:** Insertions: 30 → 20 → 10 → triggers LL rotation.

**4. What is an expression tree? Construct an expression tree and show inorder, preorder and postorder traversals.**

**Answer:** An **expression tree** is a binary tree used to represent expressions. Internal nodes are operators, and leaf nodes are operands.

**Example Expression:** (a + b) \* (c - d)

**Tree:**

\*

/ \

+ -

/ \ / \

a b c d

**Traversals:**

* Inorder: a + b \* c - d
* Preorder: \* + a b - c d
* Postorder: a b + c d - \*

**5. Explain the concept of heaps and how they are used to implement priority queues.**

**Answer:** A **heap** is a complete binary tree that satisfies the heap property:

* **Max Heap**: Parent node is always greater than or equal to its children.
* **Min Heap**: Parent node is always less than or equal to its children.

**Priority Queue:**

* A queue where each element has a priority, and elements with higher priority are dequeued first.
* Implemented using heaps.

**Operations:**

* **Insert**: Add element at the end and heapify up.
* **Delete Max/Min**: Replace root with last element and heapify down.

**Example:** Insert 10, 20, 5 into a max heap:

Initial: 10 → insert 20 → [20, 10] → insert 5 → [20, 10, 5]

**Application:**

* Task scheduling
* Dijkstra’s shortest path algorithm

**UNIT IV: MULTIWAY SEARCH TREES AND GRAPHS - 13 MARK QUESTIONS AND ANSWERS**

**Question 1: Explain the structure and operations of B-Trees. How are insertions and deletions handled in B-Trees?**

**Answer:** A B-Tree is a self-balancing multi-way search tree in which nodes can have more than two children. It is designed to work well on storage systems like disks where reading/writing large blocks of data is more efficient than single elements.

**Properties of B-Trees:**

* All leaves are at the same level.
* A node with n keys has n+1 children.
* Keys in each node are sorted.
* The keys act as separation values which divide the subtrees.
* Every internal node has at least \u2071 (minimum degree) children (except the root).

**Insertion in B-Trees:**

* Begin at the root and find the appropriate leaf node.
* Insert the key in sorted order.
* If the node overflows (i.e., has more than 2t - 1 keys), split the node and promote the middle key to the parent.
* Repeat the split recursively up to the root if necessary.

**Deletion in B-Trees:**

* Delete from leaf node directly.
* If the key is in an internal node:
  + Replace it with predecessor or successor and delete recursively.
  + If child node has fewer than t keys, rebalance using rotations or merging.

**Question 2: Describe the structure and advantages of B+ Trees. How do they differ from B-Trees?**

**Answer:** A B+ Tree is an extension of a B-Tree where all records are stored at the leaf level and internal nodes store only keys.

**Structure:**

* All keys are stored in the leaves.
* Internal nodes guide search using indexing keys.
* Leaf nodes are linked for efficient range searches.

**Differences from B-Trees:**

* B+ Tree stores data only at leaf nodes; B-Tree stores data in internal and leaf nodes.
* B+ Trees support efficient range queries.
* B+ Trees use more space due to duplicated keys in internal nodes.

**Advantages:**

* Better suited for databases and file systems.
* Faster search operations due to fixed-height tree.

**Question 3: Describe the different graph representations and their advantages.**

**Answer:** Graphs can be represented in several ways:

1. **Adjacency Matrix:**
   * A 2D matrix where matrix[i][j] = 1 if there is an edge between vertex i and j.
   * Pros: Fast edge lookup (O(1)).
   * Cons: High space usage (O(V^2)).
2. **Adjacency List:**
   * Array of lists. Each vertex has a list of its adjacent vertices.
   * Pros: Space-efficient for sparse graphs (O(V + E)).
   * Cons: Slower edge lookup (O(V)).
3. **Incidence Matrix:**
   * Rows represent vertices and columns represent edges.
   * Pros: Useful for directed graphs and edge analysis.
   * Cons: Rarely used due to complexity.

**Question 4: Explain Depth First Search (DFS) and Breadth First Search (BFS) with examples.**

**Answer:** **DFS (Depth First Search):**

* Explores as far as possible along each branch before backtracking.
* Implemented using a stack (recursion or explicit).
* Useful in topological sorting, cycle detection.

**Example:** Given: A -> B -> C, A -> D

* DFS(A): A, B, C, D

**BFS (Breadth First Search):**

* Explores all neighbors before moving to the next level.
* Implemented using a queue.
* Useful in shortest path, level order traversal.

**Example:** Given: A -> B -> C, A -> D

* BFS(A): A, B, D, C

**Question 5: Discuss Prim's and Kruskal's algorithm for Minimum Spanning Tree (MST) with examples.**

**Answer:** **Minimum Spanning Tree (MST):**

* A spanning tree of a connected graph with the minimum total edge weight.

**Prim’s Algorithm:**

* Start from any vertex.
* Grow the MST by adding the smallest edge connecting the tree to a new vertex.
* Use priority queue.
* Time Complexity: O(E log V)

**Kruskal’s Algorithm:**

* Sort all edges by weight.
* Add the smallest edge to the MST if it doesn't form a cycle.
* Use Union-Find for cycle detection.
* Time Complexity: O(E log E)

**Example (for both):** Graph with vertices A, B, C, D and edges:

* A-B (1), B-C (4), A-C (3), C-D (2)
* Kruskal: Pick edges A-B, C-D, A-C → Total = 1+2+3 = 6
* Prim (from A): A-B, A-C, C-D → Total = 1+3+2 = 6

**UNIT V: SEARCHING, SORTING AND HASHING TECHNIQUES**

**1. Explain and compare Linear Search and Binary Search algorithms with examples.**

**Linear Search:**

* It is a simple search algorithm that checks every element in the list until the desired element is found or the list ends.
* Time Complexity: O(n)
* Best case: O(1), Worst case: O(n)

**Example:**

int linearSearch(int arr[], int n, int x) {

for (int i = 0; i < n; i++) {

if (arr[i] == x) return i;

}

return -1;

}

**Binary Search:**

* Used for sorted arrays. It repeatedly divides the array in half to find the target element.
* Time Complexity: O(log n)
* Best case: O(1), Worst case: O(log n)

**Example:**

int binarySearch(int arr[], int l, int r, int x) {

while (l <= r) {

int mid = l + (r - l) / 2;

if (arr[mid] == x) return mid;

if (arr[mid] < x) l = mid + 1;

else r = mid - 1;

}

return -1;

}

**Comparison:**

| **Criteria** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Array Type | Unsorted/Sorted | Sorted Only |
| Time Complexity | O(n) | O(log n) |
| Implementation | Simple | Slightly complex |

**2. Describe and differentiate various sorting techniques with examples.**

**Bubble Sort:**

* Repeatedly swaps adjacent elements if they are in wrong order.
* Time Complexity: O(n^2)

**Selection Sort:**

* Finds the minimum element and places it at the beginning.
* Time Complexity: O(n^2)

**Insertion Sort:**

* Builds sorted array one item at a time.
* Time Complexity: O(n^2)

**Shell Sort:**

* Generalization of Insertion sort. Uses gap sequences to sort elements.
* Time Complexity: Depends on gap, best case O(n log n)

**Merge Sort:**

* Divides the array into halves and merges them in sorted order.
* Time Complexity: O(n log n)

**Example (Merge Sort):**

void merge(int arr[], int l, int m, int r) {

// merge logic

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

**Comparison Table:**

| **Sort** | **Best** | **Average** | **Worst** | **Stable** | **In-place** |
| --- | --- | --- | --- | --- | --- |
| Bubble | O(n) | O(n^2) | O(n^2) | Yes | Yes |
| Selection | O(n^2) | O(n^2) | O(n^2) | No | Yes |
| Insertion | O(n) | O(n^2) | O(n^2) | Yes | Yes |
| Shell | O(n log n) | Varies | O(n^2) | No | Yes |
| Merge | O(n log n) | O(n log n) | O(n log n) | Yes | No |

**3. Explain Hashing and its techniques. Compare Separate Chaining and Open Addressing.**

**Hashing:**

* Hashing is a technique to map keys to values using hash functions.
* Hash function: Converts a key into an index.

**Techniques:**

1. **Separate Chaining:**
   * Collisions handled using linked lists.
   * Easy to implement, but uses extra space.
2. **Open Addressing:**
   * All elements stored in hash table only.
   * On collision, searches for next available slot.
   * Includes Linear Probing, Quadratic Probing, Double Hashing.

**Example (Separate Chaining):**

struct Node {

int key;

Node\* next;

};

**Comparison:**

| **Criteria** | **Separate Chaining** | **Open Addressing** |
| --- | --- | --- |
| Space | More (extra lists) | Less (no extra memory) |
| Insertion | Easier | May need probing |
| Deletion | Easier | Complex |

**4. Explain Open Addressing with Linear Probing, Quadratic Probing, and Double Hashing.**

**Linear Probing:**

* When a collision occurs, check next slot (i+1).
* h(k, i) = (h'(k) + i) % m

**Quadratic Probing:**

* Check i^2-th position: h(k, i) = (h'(k) + c1\*i + c2\*i^2) % m
* Reduces clustering.

**Double Hashing:**

* Uses a second hash function: h(k, i) = (h1(k) + i \* h2(k)) % m
* Least clustering.

**Advantages:**

* Space efficient
* Fast lookup in average cases

**Disadvantages:**

* Deletion is difficult
* Table resizing is complex

**5. What is Extendible Hashing? Explain with structure and example.**

**Extendible Hashing:**

* A dynamic hashing technique that grows the directory and buckets as needed.
* Uses a directory of pointers to buckets.
* Hash function returns binary string; directory uses first ‘i’ bits to index.

**Features:**

* Avoids overflow chains
* Directory doubles when needed
* Efficient disk-based indexing

**Example:**

* Initially: Global Depth = 1
* After inserts and splits: Directory size doubles, and pointers adjusted.

**Advantages:**

* Dynamic resizing
* Efficient space usage

**Disadvantages:**

* Directory may grow large
* More complex than static hashing