Q 2. Discuss how a two-way stack can be developed using array and write sudo code for Push, Pop and display operations.

Two-Way Stack Using Array

A **two-way stack** is a data structure implemented within a single array where two stacks grow towards each other. This approach is space-efficient, as it allows the two stacks to share memory dynamically without requiring predefined space allocation for each stack.

Theory

1. **Structure**:
   - The array is divided logically, not physically.
   - **Stack 1** grows from the **leftmost index (0)** towards the **right**.
   - **Stack 2** grows from the **rightmost index (n-1)** towards the **left**.
2. **Pointers**:
   - top1: Points to the top of Stack 1, initialized to -1.
   - top2: Points to the top of Stack 2, initialized to n (size of the array).
3. **Key Operations**:
   - **Push Operation**:
     - Insert an element into a stack if there is space between top1 and top2.
   - **Pop Operation**:
     - Remove and return the top element from the respective stack.
4. **Space Utilization**:
   - Stacks grow towards each other, so space is dynamically adjusted based on the usage of both stacks.

Advantages

1. **Efficient Space Utilization**:
   - Memory is shared between two stacks, reducing wastage compared to separate arrays.
2. **Dynamic Space Allocation**:
   - The size of each stack adapts dynamically based on the growth of the other.
3. **Simple Implementation**:
   - Can be implemented with minimal changes to a single array data structure.

4. **Prevention of Fixed Stack Overflow**:
    - Prevents overflow in one stack as long as the other stack has unused space.

Disadvantages

1. **Complex Overflow Management**:
    - Requires constant checking of boundaries (top1 + 1 < top2) to prevent stack overflow.
2. **Limited Scalability**:
    - The size of the array is fixed; thus, it cannot handle large data growth without resizing.
3. **Manual Implementation**:
    - Requires careful programming to manage boundary conditions and avoid errors.
4. **Wastage of Space**:
    - When both stacks are not growing simultaneously, unused space in one side is inaccessible to the other.

Applications

1. **Memory Management**:
    - Useful in situations where memory is limited and shared between two tasks.
2. **Expression Parsing**:
    - Can be used to evaluate expressions where two stacks are required, e.g., one for operands and another for operators.
3. **Dual Functionality**:
    - Simultaneously maintaining two types of data (e.g., undo-redo stacks or two independent operations).
4. **Stack Simulation in Compilers**:
    - Useful in scenarios where compiler design requires two separate memory stacks (e.g., function call stack and temporary stack).

Example

Given:

An array of size 6.

Initial State:

Array: [_ , _ , _ , _ , _ , _]

top1 = -1

top2 = 6

Q 3. Convert the following infix expressions to postfix using stack. Clearly indicate the contents of stack. i) (A+B) *C- D*F +C ii) (A-5)*(b+C-D*E)/F

i) Expression: (A + B) * C - D * F + C

Initial Expression: (A + B) * C - D * F + C

Step-by-Step Conversion:

Symbol	**Stack**	**Postfix Expression**
(	(	
A	(	A
+	(, +	A
B	(, +	A B
)		A B +
*	*	A B +
C	*	A B + C
-	-	A B + C *
D	-	A B + C * D
*	-, *	A B + C * D
F	-, *	A B + C * D F
+	+	A B + C * D F * -
C	+	A B + C * D F * - C
End		A B + C * D F * - C +

Postfix Expression: A B + C * D F * - C +

ii) Expression: (A - 5) * (B + C - D * E) / F

Initial Expression: (A - 5) * (B + C - D * E) / F

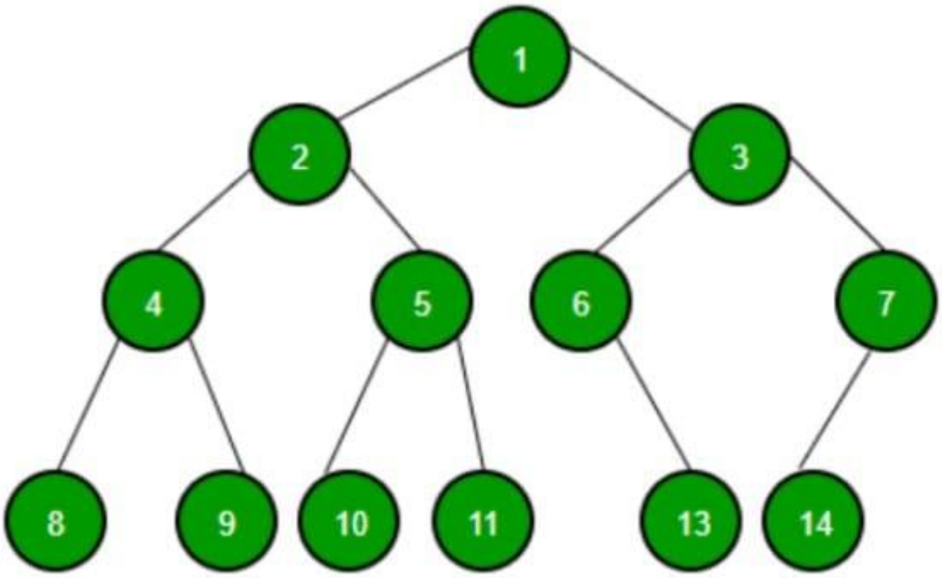
Step-by-Step Conversion:

Symbol	Stack	Postfix Expression
(	(	
A	(	A
-	(, -	A
5	(, -	A 5
)		A 5 -
*	*	A 5 -
(	*, (	A 5 -
B	*, (	A 5 - B
+	*, (, +	A 5 - B
C	*, (, +	A 5 - B C
-	*, (, -	A 5 - B C +
D	*, (, -	A 5 - B C + D
*	*, (, -, *	A 5 - B C + D
E	*, (, -, *	A 5 - B C + D E
)	*	A 5 - B C + D E * -
/	*, /	A 5 - B C + D E * -
F	*, /	A 5 - B C + D E * - F
End		A 5 - B C + D E * - F / *

Postfix Expression: A 5 - B C + D E * - F / *

Q 7. What is a Binary Tree? Explain the following operations on BinaryTree i) Inserting a node in to BT ii) Deletion a node from BT.

A binary tree is a type of **tree data structure** in which each node can have at most two child nodes, known as the left child and the right child. Each node of the tree consists of – data and pointers to the left and the right child.



Example of Binary Tree

Operations on Binary Tree

i) Insertion of a Node in a Binary Tree

Insertion in a binary tree is typically done in **level order** to maintain the tree structure. This ensures the tree remains as complete as possible.

Algorithm (Level Order Insertion):

1. Use a **queue** for level-order traversal.
2. Start with the root node:
   - If the **left child** of the current node is null, insert the new node as the left child and stop.
   - If the **right child** of the current node is null, insert the new node as the right child and stop.
   - If neither is null, enqueue both children for further processing.
3. Repeat until the new node is inserted.

Example:

Insert 8 into the following binary tree:

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

Steps:

1. Enqueue 1.
2. Check 1: Both children (2 and 3) exist. Enqueue 2 and 3.
3. Check 2: Both children (4 and 5) exist. Enqueue 4 and 5.
4. Check 3: Left child exists (6), but right child is null. Insert 8 as the right child of 3.

Updated Tree:

```
    1
  /  \
  2    3
 /\  /\
4   5 6   8
```

ii) Deletion of a Node in a Binary Tree

Deleting a node from a binary tree involves **replacing it with the deepest and rightmost node** in the tree to maintain the structure.

Algorithm:

1. Use **level-order traversal** to locate:
   - The **node to be deleted** (key).
   - The **deepest and rightmost node** in the tree.
2. Replace the value of the node to be deleted with the value of the deepest node.
3. Remove the deepest node from the tree.

Example:

Delete 5 from the following binary tree:

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

Steps:

1. Enqueue 1. Traverse level-order to locate 5 (node to delete) and 7 (deepest node).
2. Replace 5 with 7.
3. Delete the deepest node (7).

Updated Tree:

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

Advantages of Using a Binary Tree

1. **Efficient Searching**: Binary trees provide a structured way to search data (especially Binary Search Trees).
2. **Space Optimization**: Complete binary trees use minimal memory.
3. **Hierarchical Representation**: Useful for implementing hierarchical relationships (e.g., file systems, organizational charts).

Applications of Binary Trees

1. **Expression Trees**: Represent algebraic expressions.
2. **Priority Queues**: Implemented using binary heaps.
3. **Binary Search Trees**: Fast lookup, insertion, and deletion.
4. **Huffman Encoding**: Used in data compression.

Q 8. What is the use of threaded binary free? Give the node structure required for a threaded binary tree. Write pseudo code to find in-order successor of any node X in a threaded binary tree.

Threaded Binary Tree

A **threaded binary tree** is a type of binary tree where **null pointers** in the node structure are replaced with **threads**. These threads help in efficiently traversing the tree in **in-order** without requiring a stack or recursion. The threads point to the **in-order predecessor** or **in-order successor** of a node.

Uses of Threaded Binary Tree

1. **Efficient Traversal**: Allows in-order traversal without recursion or a stack, saving memory and execution time.
2. **Space Optimization**: Utilizes null pointers to store additional information (threads), reducing memory wastage.
3. **Faster Tree Navigation**: Directly access the next or previous node in in-order sequence.

Node Structure for a Threaded Binary Tree

A node in a threaded binary tree requires additional fields to store thread information.

Node Structure

```
struct Node {
    int data;         // Data stored in the node
    Node* left;       // Pointer to the left child (or in-order predecessor if threaded)
    Node* right;      // Pointer to the right child (or in-order successor if threaded)
    bool isLeftThread; // True if left pointer is a thread, False otherwise
    bool isRightThread; // True if right pointer is a thread, False otherwise
};
```

Pseudo Code for Finding In-Order Successor

```
FUNCTION findInOrderSuccessor(Node X)
   // Case 1: If the node has a right child and is not threaded
   IF X.isRightThread == FALSE THEN
      RETURN leftmostNode(X.right)
   ENDIF

   // Case 2: If the node is threaded
   RETURN X.right
END FUNCTION

FUNCTION leftmostNode(Node node)
   WHILE node.left IS NOT NULL AND node.isLeftThread == FALSE DO
      node = node.left
   END WHILE
   RETURN node
END FUNCTION
```

Q 13. What is topological Sorting? Illustrate with an example how topologicalS sorting is performed. List any two applications where topological sorting can be used.

What is Topological Sorting?

Topological sorting is a linear ordering of vertices in a **directed acyclic graph (DAG)** such that for every directed edge u→vu \to vu→v, the vertex uuu appears before vvv in the ordering. It is used to represent dependencies in tasks, processes, or systems.

- **Key points**:
  - Only works for **DAGs** (Directed Acyclic Graphs).
  - Shows the correct sequence of tasks or events based on dependencies.

How to Perform Topological Sorting

There are two common ways to perform topological sorting:

1. **Kahn's Algorithm** (using in-degree).

2. **DFS-based Approach** (using recursion).

We'll illustrate **Kahn's Algorithm** here:

Steps in Kahn's Algorithm:

1. Compute the **in-degree** (number of incoming edges) for each vertex.
2. Add all vertices with in-degree = 0 to a **queue**.
3. While the queue is not empty:
   - Remove a vertex from the queue and add it to the topological order.
   - For each neighbor of this vertex, reduce its in-degree by 1.
   - If any neighbor's in-degree becomes 0, add it to the queue.
4. Repeat until all vertices are processed.

Topological Sorting Example

Graph:

A → B → D

↓   ↑

→ C →

Steps:

1. **In-degree of nodes**:
   - A: 0
   - B: 1
   - C: 1
   - D: 2
2. **Start with nodes having in-degree 0**:
   - Add A to the result (in-degree 0).
3. **Process A**:
   - Reduce in-degrees of B and C:
     - B: 0, C: 0
   - Add B and C to the queue.
4. **Process B**:
   - Add B to the result.
   - Reduce in-degree of D: D: 1.

5. **Process C**:
    - Add C to the result.
    - Reduce in-degree of D: D: 0.
    - Add D to the queue.
6. **Process D**:
    - Add D to the result.

Topological Order: **A → B → C → D**

Applications of Topological Sorting

1. **Task Scheduling**:
    - Used to determine the order of tasks based on dependencies (e.g., task A must be done before task B).
2. **Course Prerequisites**:
    - Helps in finding the correct sequence of courses to take based on prerequisite requirements.
3. **Dependency Resolution**:
    - In build systems (like Makefile), used to compile files in the correct order.
4. **Critical Path Analysis**:
    - Used in project management to determine the order of tasks and the critical path.

Q 15. Illustrate with examples the Reheap up and Reheap down operations w.r.t. heaps. List any three applications of Heap.

Reheap Up and Reheap Down in Heaps

A **heap** is a complete binary tree where the value of each node satisfies the **heap property**:

- **Max Heap**: The value of each node is greater than or equal to its children.
- **Min Heap**: The value of each node is less than or equal to its children.

Reheap Up (or Bubble Up)

- This operation is used when a node is added to the heap (usually at the end).
- After insertion, the heap may violate the heap property. The Reheap Up operation fixes the violation by comparing the newly inserted node with its parent and swapping them if necessary.
- This process continues until the heap property is restored.

Reheap Down (or Bubble Down)

- This operation is used when a node is removed (typically the root in a max heap or min heap).
- After removal, the heap may violate the heap property. The Reheap Down operation fixes the violation by comparing the node with its children and swapping it with the larger (in a max heap) or smaller (in a min heap) child.
- This process continues until the heap property is restored.

Applications of Heaps

1. **Priority Queue**:
   - A heap is used to implement a priority queue, where each element has a priority. The element with the highest or lowest priority is always processed first. It's commonly used in task scheduling or managing events in simulations.
2. **Heap Sort**:
   - Heaps are used in the heap sort algorithm, which is an efficient way to sort a list of elements. It repeatedly extracts the maximum (or minimum) element from the heap and places it in the correct position.
3. **Graph Algorithms**:
   - Heaps are used in graph algorithms like **Dijkstra's algorithm** (for finding the shortest path) and **Prim's algorithm** (for finding the minimum spanning tree), where they help efficiently select the next node to process.

Q 16. Explain with example hash functions? . / Write short note on closed hashing and Open addressing.

Hash Functions

A **hash function** is a function that takes an input (or key) and returns a fixed-size string or number, typically used to map data to a specific location (index) in a hash table. The main purpose is to **distribute** the keys uniformly across the table to avoid collisions.

Example of a Hash Function:

Let's say we have a list of numbers and we want to store them in a hash table. The hash function could be as simple as taking the modulus of the number by the size of the table.

Hash function:
Hash(key) = key % table_size

For a hash table of size 10, if the input is 25, the hash value would be:
Hash(25) = 25 % 10 = 5
So, the value 25 would be stored at index 5.

Closed Hashing (Also called Open Addressing)

In **closed hashing**, all elements are stored **within** the hash table itself. When a collision occurs (i.e., when two keys hash to the same index), the algorithm finds another available slot within the table.

Types of Closed Hashing:

1. **Linear Probing**:
   If the desired index is already occupied, move to the next index in a linear fashion (i.e., index + 1) until an empty slot is found.

Example:

- Table: [ ] [ ] [ ] [ ] [ ]
- Insert 15, Hash(15) = 5 → Place 15 at index 5.
- Insert 25, Hash(25) = 5 → Slot 5 is full, so check slot 6.
- Insert 25 at index 6.

2. **Quadratic Probing**:
   If the index is occupied, move to index + 1^2, then + 2^2, and so on.

3. **Double Hashing**:
   Uses a second hash function to calculate the next available index when a collision occurs.

Open Addressing

Open addressing is a method where collisions are resolved by **searching for the next open slot** within the hash table itself. Unlike closed hashing, where new storage locations are allocated, open addressing looks for available slots inside the existing table.

Types of Open Addressing:

1. **Linear Probing**:
   As mentioned, if a slot is full, it checks the next slot (index + 1).

2. **Quadratic Probing**:
   Similar to linear probing, but instead of moving one slot ahead, it checks positions based on quadratic increments (i.e., index + 1^2, + 2^2, etc.).

3. **Double Hashing**:
   Uses a second hash function to calculate the next slot.

Key Differences:

- **Closed Hashing**: Stores the element in the hash table directly; handles collisions by probing or chaining.
- **Open Addressing**: Resolves collisions by finding another open slot within the hash table itself.

Q 17. Write Comparison of different file organizations (sequential, index sequential and Direct Access).

Comparison of Different File Organizations

File Organization	**Sequential**	**Index Sequential**	**Direct Access**
Access Method	Data is accessed in order, one by one.	Access via an index or sequentially.	Directly accessed using a key or address.
Speed of Access	Slow for searching, needs to read all records.	Faster than sequential, can use index for quick access.	Very fast, as data is accessed directly without search.
Data Structure	Stored in a sequential file (sorted or unsorted).	Stores data in a sequential file with an index file.	Data is stored in a direct access file (e.g., hash table).
Efficiency	Not efficient for random access.	More efficient than sequential for large data sets.	Very efficient for large datasets with frequent random access.
Modification	Difficult to insert, delete, or update.	Easier to modify using index.	Easy to modify, insert, or delete data.
Storage	Requires less storage space.	Requires extra storage for the index.	May require more storage due to index or direct address management.
Application	Best for processing records in order (e.g., logs).	Suitable for databases with frequent searches.	Ideal for systems needing fast retrieval (e.g., databases).
Examples	Simple file storage, batch processing.	Traditional databases, file systems.	Database management systems, memory-resident data.

Q 20. Explain chaining with replacement and chaining without replacement in hashing?

Chaining in Hashing

Chaining is a collision resolution technique in hashing where each hash table index points to a **linked list** (or other data structures) of elements that hash to the same index. It helps handle multiple keys that hash to the same location (i.e., collisions).

Chaining with Replacement

- **Definition**: In **chaining with replacement**, when a collision occurs, the new element **replaces** the existing element at the same index in the hash table.
- **How it works**: If two keys hash to the same index, instead of storing both keys in a linked list, the new key replaces the old key.
- **Use case**: This method is typically used when only one element should be stored at each hash table index.

Example:

- Assume a hash table of size 5 and a simple hash function hash(key) = key % 5.
- Inserting key 15:
  - hash(15) = 15 % 5 = 0 → Place 15 at index 0.
- Inserting key 5:
  - hash(5) = 5 % 5 = 0 → Replace 15 with 5 at index 0.

After insertion, the table looks like:

Index 0: 5

Index 1: -

Index 2: -

Index 3: -

Index 4: -

Chaining without Replacement

- **Definition**: In **chaining without replacement**, when a collision occurs, the new element is **added** to the linked list at the same index in the hash table, meaning both elements are stored.
- **How it works**: If two keys hash to the same index, the new key is **added to a linked list** or a chain at that index, preserving all elements that hash to the same location.
- **Use case**: This method is typically used when multiple elements can be stored at each index.

Example:

- Assume a hash table of size 5 and the same hash function hash(key) = key % 5.
- Inserting key 15:
  - hash(15) = 15 % 5 = 0 → Place 15 at index 0.
- Inserting key 5:
  - hash(5) = 5 % 5 = 0 → Add 5 to the list at index 0 (chain).

After insertion, the table looks like:

Index 0: 15 → 5

Index 1: -

Index 2: -

Index 3: -

Index 4: -

Key Differences:

Feature	**Chaining with Replacement**	**Chaining without Replacement**
Handling Collisions	New element replaces the existing one at the index.	New element is added to a linked list at the index.
Memory Usage	Uses less memory since only one element is stored per index.	Uses more memory to store a linked list at each index.
Efficiency	May lose data when collisions occur.	All elements are preserved, but requires more space.
Use Case	Suitable when only one element should be stored at each index.	Suitable for storing multiple elements at the same index.