ADA LAB 1

1. **What is STL in C++? Write about containers in STL.**

The Standard Template Library (STL) is a powerful component of the C++ Standard Library that provides a set of common classes and interfaces for data structures and algorithms. It enables developers to write more efficient and reusable code by offering a collection of generic classes and functions. STL is primarily composed of four components: containers, algorithms, iterators, and function objects.

Containers are the fundamental part of STL, as they provide the means to store and manage collections of objects. They abstract the details of data storage and provide a uniform interface for accessing and manipulating the elements. There are four main types of containers in the C++ Standard Template Library (STL). Besides sequence containers, associative containers, and container adapters, there is a fourth type called unordered associative containers. Here's a brief overview of each type, including the unordered associative containers:

1. Sequence Containers

Sequence containers store elements in a specific order. The common sequence containers are:

- vector: A dynamic array allowing fast random access and efficient insertion/deletion at the end.
  Example: std::vector<int> vec = {1, 2, 3, 4};
- deque: A double-ended queue that allows fast insertion and deletion at both ends.
  Example: std::deque<int> deq = {1, 2, 3, 4};
- list: A doubly-linked list allowing fast insertion and deletion anywhere in the sequence.
  Example: std::list<int> lst = {1, 2, 3, 4};
- forward_list: A singly-linked list allowing fast insertion and deletion at the front.
  Example: std::forward_list<int> fwd_lst = {1, 2, 3, 4};
- array: A static array with fixed size, providing fast random access.
  Example: std::array<int, 4> arr = {1, 2, 3, 4};

2. Associative Containers

Associative containers store elements formed by a combination of a key value and a mapped value, and they allow fast retrieval based on keys. The common associative containers are:

- set: A collection of unique keys, sorted by keys.
  Example: std::set<int> my_set = {3, 1, 4, 2};

- multiset: Similar to a set but allows duplicate keys.
  Example: std::multiset<int> my_multiset = {3, 1, 4, 2, 3};
- map: A collection of key-value pairs with unique keys, sorted by keys.
  Example: std::map<int, std::string> my_map = {{1, "one"}, {2, "two"}};
- multimap: Similar to a map but allows duplicate keys.
  Example: std::multimap<int, std::string> my_multimap = {{1, "one"}, {1, "uno"}};

3. Unordered Associative Containers

Unordered associative containers store elements based on hash functions and allow average-case constant-time complexity for search, insert, and delete operations. The common unordered associative containers are:

- unordered_set: A collection of unique keys, not sorted.
  Example: std::unordered_set<int> my_unordered_set = {3, 1, 4, 2};
- unordered_multiset: Similar to an unordered_set but allows duplicate keys.
  Example: std::unordered_multiset<int> my_unordered_multiset = {3, 1, 4, 2, 3};
- unordered_map: A collection of key-value pairs with unique keys, not sorted.
  Example: std::unordered_map<int, std::string> my_unordered_map = {{1, "one"}, {2, "two"}};
- unordered_multimap: Similar to an unordered_map but allows duplicate keys.
  Example: std::unordered_multimap<int, std::string> my_unordered_multimap = {{1, "one"}, {1, "uno"}};

4. Container Adapters

Container adapters provide a different interface for the underlying sequence containers. The common container adapters are:

- stack: A last-in, first-out (LIFO) data structure.
  Example: std::stack<int> my_stack;
  my_stack.push(1);
  my_stack.push(2);
- queue: A first-in, first-out (FIFO) data structure.
  Example: std::queue<int> my_queue;
  my_queue.push(1);
  my_queue.push(2);
- priority_queue: A queue that allows retrieval of the largest element.
  Example: std::priority_queue<int> my_pq;
  my_pq.push(3);
  my_pq.push(1);

2. Explain Sequence, Adapter, Associative and unordered container in STL.

Sequence Containers

Sequence containers store elements in a specific order. The common sequence containers are:

- vector: A dynamic array allowing fast random access and efficient insertion/deletion at the end.
- deque: A double-ended queue that allows fast insertion and deletion at both ends.
  Example: std::deque<int> deq = {1, 2, 3, 4};
- list: A doubly-linked list allowing fast insertion and deletion anywhere in the sequence.
  Example: std::list<int> lst = {1, 2, 3, 4};
- forward_list: A singly-linked list allowing fast insertion and deletion at the front.
  Example: std::forward_list<int> fwd_lst = {1, 2, 3, 4};
- array: A static array with fixed size, providing fast random access.
  Example: std::array<int, 4> arr = {1, 2, 3, 4};

Associative Containers

Associative containers store elements formed by a combination of a key value and a mapped value, and they allow fast retrieval based on keys. The common associative containers are:

- set: A collection of unique keys, sorted by keys.
  Example: std::set<int> my_set = {3, 1, 4, 2};
- multiset: Similar to a set but allows duplicate keys.
  Example: std::multiset<int> my_multiset = {3, 1, 4, 2, 3};
- map: A collection of key-value pairs with unique keys, sorted by keys.
  Example: std::map<int, std::string> my_map = {{1, "one"}, {2, "two"}};
- multimap: Similar to a map but allows duplicate keys.
  Example: std::multimap<int, std::string> my_multimap = {{1, "one"}, {1, "uno"}};

Unordered Associative Containers

Unordered associative containers store elements based on hash functions and allow average-case constant-time complexity for search, insert, and delete operations. The common unordered associative containers are:

- unordered_set: A collection of unique keys, not sorted.
  Example: std::unordered_set<int> my_unordered_set = {3, 1, 4, 2};
- unordered_multiset: Similar to an unordered_set but allows duplicate keys.
  Example: std::unordered_multiset<int> my_unordered_multiset = {3, 1, 4, 2, 3};
- unordered_map: A collection of key-value pairs with unique keys, not sorted.
  Example: std::unordered_map<int, std::string> my_unordered_map = {{1, "one"}, {2, "two"}};

- unordered_multimap: Similar to an unordered_map but allows duplicate keys.
  Example: std::unordered_multimap<int, std::string> my_unordered_multimap = {{1, "one"}, {1, "uno"}};

Container Adapters

Container adapters provide a different interface for the underlying sequence containers. The common container adapters are:

- stack: A last-in, first-out (LIFO) data structure.
  Example: std::stack<int> my_stack;
  my_stack.push(1);
  my_stack.push(2);
- queue: A first-in, first-out (FIFO) data structure.
  Example: std::queue<int> my_queue;
  my_queue.push(1);
  my_queue.push(2);
- priority_queue: A queue that allows retrieval of the largest element.
  Example: std::priority_queue<int> my_pq;
  my_pq.push(3);
  my_pq.push(1);

3. Implement priority queue in C++.

```
#include <iostream>
#include <queue>
#include <vector>

int main() {
   std::priority_queue<int> pq;

   pq.push(10);
   pq.push(30);
   pq.push(20);
   pq.push(5);

   std::cout << "Size of priority queue: " << pq.size() << std::endl;

   std::cout << "Elements in priority queue (in descending order): ";
   while (!pq.empty()) {
      std::cout << pq.top() << " ";
      pq.pop();
   }
   std::cout << std::endl;

   return 0;
}
```

4. Implement stack using array and linked list in C++.

Stack Using Array

```
#include <iostream>

#define MAX_SIZE 100

class StackArray {
private:
    int top;
    int arr[MAX_SIZE];

public:
    StackArray() {
        top = -1;    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == MAX_SIZE - 1;
    }

    void push(int val) {
        if (isFull()) {
            std::cout << "Stack overflow!" << std::endl;
            return;
        }
        arr[++top] = val;
    }

    int pop() {
        if (isEmpty()) {
            std::cout << "Stack underflow!" << std::endl;
            return -1;
        }
        return arr[top--];
    }

    int peek() {
        if (isEmpty()) {
            std::cout << "Stack is empty!" << std::endl;
```

```
            return -1;
        }
        return arr[top];
    }
};

int main() {
    StackArray stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Top element: " << stack.peek() << std::endl;

    std::cout << "Popped element: " << stack.pop() << std::endl;
    std::cout << "Popped element: " << stack.pop() << std::endl;

    std::cout << "Top element after popping: " << stack.peek() << std::endl;

    return 0;
}
```

Stack Using Linked List

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

class StackLinkedList {
private:
    Node* top;

public:
    StackLinkedList() : top(nullptr) {}

    bool isEmpty() {
```

```
        return top == nullptr;
    }

    void push(int val) {
        Node* newNode = new Node(val);
        newNode->next = top;
        top = newNode;
    }

    int pop() {
        if (isEmpty()) {
            std::cout << "Stack underflow!" << std::endl;
            return -1;
        }
        int poppedValue = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return poppedValue;
    }
    int peek() {
        if (isEmpty()) {
            std::cout << "Stack is empty!" << std::endl;
            return -1;
        }
        return top->data;
    }
};

int main() {
    StackLinkedList stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Top element: " << stack.peek() << std::endl;

    std::cout << "Popped element: " << stack.pop() << std::endl;
    std::cout << "Popped element: " << stack.pop() << std::endl;

    std::cout << "Top element after popping: " << stack.peek() << std::endl;
    return 0;
}
```

5. Implement Binary Search Tree in C++.

```
#include <iostream>

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertNode(TreeNode* node, int value) {
        if (node == nullptr) {
            return new TreeNode(value);
        }

        if (value < node->data) {
            node->left = insertNode(node->left, value);
        } else if (value > node->data) {
            node->right = insertNode(node->right, value);
        }

        return node;
    }

    TreeNode* findMinNode(TreeNode* node) {
        while (node->left != nullptr) {
            node = node->left;
        }
        return node;
    }

    TreeNode* deleteNode(TreeNode* node, int value) {
        if (node == nullptr) {
            return nullptr;
        }

        if (value < node->data) {
```

```
            node->left = deleteNode(node->left, value);
        } else if (value > node->data) {
            node->right = deleteNode(node->right, value);
        } else {
            // Case 1: No child or one child
            if (node->left == nullptr) {
                TreeNode* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == nullptr) {
                TreeNode* temp = node->left;
                delete node;
                return temp;
            }

            // Case 2: Two children
            TreeNode* minRight = findMinNode(node->right);
            node->data = minRight->data;
            node->right = deleteNode(node->right, minRight->data);
        }

        return node;
    }

    void inOrderTraversal(TreeNode* node) {
        if (node != nullptr) {
            inOrderTraversal(node->left);
            std::cout << node->data << " ";
            inOrderTraversal(node->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int value) {
        root = insertNode(root, value);
    }

    void remove(int value) {
        root = deleteNode(root, value);
    }

    void traverseInOrder() {
```

```
        inOrderTraversal(root);
    }
};

int main() {
    BinarySearchTree bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    std::cout << "Inorder traversal of the BST: ";
    bst.traverseInOrder();
    std::cout << std::endl;

    bst.remove(30);
    std::cout << "Inorder traversal after removing 30: ";
    bst.traverseInOrder();
    std::cout << std::endl;

    return 0;
}
```