NEPAL COLLEGE OF INFORMATION TECHNOLOGY

Balkumari, Lalitpur

(Affiliated to Pokhara University)

A Lab Report On

Subject:- Analysis and Design of Algorithm

Lab Report # 1

Title:Standard Template Library

Submitted by:
Name: -Anshu Thapa
Roll No: - 03
Faculty;-Science and Technology
Semester: -4th

Submitted to:
Instructor: - Ashok Basnet
Department of;-
Software Engineering
Submission Date:2024/05/28

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

Ans. STL stands for Standard Template Library which is a powerful feature of C++ that provides predefined templates for handling data structures and algorithms. Containers are fundamental components of the C++ Standard Template Library (STL), designed to store collections of objects. They provide various ways to organize, manage, and access data efficiently. STL containers can be broadly categorized into sequence containers, associative containers, and container adapters. Each type of container offers specific characteristics and functionalities tailored to different needs.

A. Sequence Containers

Sequence containers maintain the order of elements, allowing for direct access to elements through positions.

- vector:- Provides random access to elements, supports efficient insertion and deletion at the end, but can be costly for insertion and deletion in the middle.
- deque:- Supports fast insertion and deletion at both the beginning and the end, with random access to elements.
- list:-Allows efficient insertion and deletion at any position but does not support random access (sequential access only).
- forward_list:- Similar to list but with forward-only traversal, which makes it more memory efficient.
- array:- Provides fast access to elements with constant time complexity but the size cannot be changed once initialized.

B. Associative Containers

Associative containers manage data structures that allow for fast retrieval based on keys.

- set:- Automatically sorts elements, provides logarithmic time complexity for insertion, deletion, and search operations.
- multiset:- Automatically sorts elements, supports multiple instances of the same element.
- map:-Stores key-value pairs in a sorted order based on keys.
- Multimap:- Similar to map but allows multiple values for the same key.

C. Unordered Associative Containers

Unordered associative containers use hash tables for storage, providing average constant time complexity for insertion, deletion, and search. They do not maintain elements in any particular order.

- unordered_set:-Stores unique elements without any particular order.

- unordered_multiset:-Similar to unordered_set but allows duplicate elements.
- unordered_map:-Stores key-value pairs with unique keys.
- unordered_multimap:-Similar to unordered_map but allows multiple values for the same key.

2. Explain sequence,Adapter,Associative and Unordered in STL containers.

Ans. Sequence Containers are used to store elements in a particular order. They provide efficient random access to elements and allow the insertion and deletion of elements at any position. Sequence Containers in C++ are defined in the header file "<vector>" and "<deque>".Sequence Containers can store various data types, such as integers, characters, strings, and user-defined data types. They are implemented as dynamic arrays, linked lists, and arrays of fixed-size elements. These containers provide several functions to manipulate the elements, such as accessing elements, inserting and deleting elements, and searching for elements.

Unordered containers in the Standard Template Library (STL) use hash tables for storage, providing average constant time complexity for insertion, deletion, and search operations. They do not maintain elements in any particular order, which distinguishes them from ordered associative containers like set and map.

Associative containers in the Standard Template Library (STL) store elements as key-value pairs and maintain them in a sorted order based on the keys. These containers allow fast retrieval of elements based on the keys, typically providing logarithmic time complexity for insertion, deletion, and search operations.

3. Implement priority Queues in C++.

Ans. #include <iostream>

#include <queue>

int main() {

std::priority_queue<int> pq;

pq.push(30);

pq.push(10);

pq.push(50);

pq.push(20);

while (!pq.empty()) {

int top_element = pq.top();

std::cout << top_element << " ";

pq.pop();

}

std::cout << std::endl;

return 0;

}

4. Implement stock using array and linked list in c++

Ans. #include <iostream>
#include <string>
using namespace std;

```
struct Stock {
    string symbol;
    double price;
    int quantity;
};

class StockArray {
private:
    Stock* stocks;
    int maxSize;
    int currentIndex;

public:
    StockArray(int size) : maxSize(size), currentIndex(0) {
        stocks = new Stock[size];
    }


    ~StockArray() {
        delete[] stocks;
    }

        Node* next;
};

class StockLinkedList {
private:
    Node* head;

public:
```

```
    StockLinkedList() : head(nullptr) {}

    ~StockLinkedList() {
        Node* current = head;
        while (current != nullptr) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
    }

    void addStock(const Stock& newStock) {void addStock(const Stock& newStock) {
        if (currentIndex < maxSize) {
            stocks[currentIndex++] = newStock;
        } else {
            cout << "Array is full!" << endl;
        }
    }

    void displayStocks() {
        for (int i = 0; i < currentIndex; ++i) {
            cout << "Symbol: " << stocks[i].symbol << ", Price: $" << stocks[i].price << ", Quantity:
" << stocks[i].quantity << endl;
        }
    }
};

struct Node {
    Stock data;

        Node* newNode = new Node{newStock, nullptr};


     if (head == nullptr) {
            head = newNode;
        } else {
```

```
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
    }

    void displayStocks() {
        Node* current = head;
        while (current != nullptr) {
            cout << "Symbol: " << current->data.symbol << ", Price: $" << current->data.price << ",
Quantity: " << current->data.quantity << endl;
            current = current->next;
        }
    }
};

int main() {
    StockArray stockArray(5);
    stockArray.addStock({"AAPL", 150.25, 10});
    stockArray.addStock({"GOOG", 2500.75, 5});
    cout << "Stocks using array:" << endl;
    stockArray.displayStocks();

    StockLinkedList stockList;
    stockList.addStock({"AAPL", 150.25, 10});
    stockList.addStock({"GOOG", 2500.75, 5});
    cout << "\nStocks using linked list:" << endl;
    stockList.displayStocks();

    return 0;
}
```

5. Implement Binary Search trees in C++.

Ans. #include <iostream>

```
using namespace std;

struct Node {
    int key;

  Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

class BST {
private:
    Node* root;

    Node* insertHelper(Node* root, int key) {
        if (!root) return new Node(key);
        if (key < root->key) root->left = insertHelper(root->left, key);
        else if (key > root->key) root->right = insertHelper(root->right, key);
        return root;
    }

    bool searchHelper(Node* root, int key) {
        if (!root) return false;
        if (root->key == key) return true;
        else if (key < root->key) return searchHelper(root->left, key);
        else return searchHelper(root->right, key);
    }

    void inorderHelper(Node* root) {
        if (!root) return;
        inorderHelper(root->left);
        cout << root->key << " ";
        inorderHelper(root->right);
    }
```

```
public:
    BST() : root(nullptr) {}

    void insert(int key) {
        root = insertHelper(root, key);
    }

    bool search(int key) {
        return searchHelper(root, key);
    }

    void inorder() {
        inorderHelper(root);
        cout << endl;

 }
};

int main() {
    BST bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    cout << "Inorder traversal of the BST: ";
    bst.inorder();

    int keyToSearch = 40;
    if (bst.search(keyToSearch)) {
        cout << keyToSearch << " is found in the BST." << endl;
    } else {
        cout << keyToSearch << " is not found in the BST." << endl;
```

```
    }

    return 0;
}
```