ROLL NO.: 21C6051

BRANCH: COMPUTER SCIENCE

SECTION: 'A'

SEMESTER: 6^{th} SEMESTER

SUBJECT: DESIGN & ANALYSIS OF ALGORITHM

SUBJECT CODE: 6CERC1

SUBMITTED TO:
DR. G L PRAJAPATI

SUBMITTED BY:
VAIBHAV MALVIYA

Contents

1.Comparison of all sorting techniques

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// Bubble Sort
void bubbleSort(std::vector<int>& data) {
    int n = data.size();
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < n - 1; i++) {
            if (data[i] > data[i + 1]) {
                std::swap(data[i], data[i + 1]);
                swapped = true;
            }
        }
    } while (swapped);
}

// Selection Sort
void selectionSort(std::vector<int>& data) {
    int n = data.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (data[j] < data[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(data[i], data[minIndex]);
    }
}

// Insertion Sort
void insertionSort(std::vector<int>& data) {
    int n = data.size();
    for (int i = 1; i < n; i++) {
        int key = data[i];
        int j = i - 1;
        while (j >= 0 && data[j] > key) {
```

```cpp
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = key;
    }
}

// Merge Sort
void merge(std::vector<int>& data, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    std::vector<int> leftArr(n1);
    std::vector<int> rightArr(n2);

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++) {
        leftArr[i] = data[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = data[mid + 1 + j];
    }

    /* Merge the temporary arrays back into data[left..right]*/
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            data[k] = leftArr[i];
            i++;
        }
       else {
            data[k] = rightArr[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of leftArr, if there are any */
    while (i < n1) {
        data[k] = leftArr[i];
        i++;
        k++;
    }
```

```cpp
    /* Copy the remaining elements of rightArr, if there are any */
    while (j < n2) {
        data[k] = rightArr[j];
        j++;
        k++;
    }
}

void mergeSort(std::vector<int>& data, int left, int right) {
    if (left < right) {
        // Find the middle point
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(data, left, mid);
        mergeSort(data, mid + 1, right);

        // Merge the sorted halves
        merge(data, left, mid, right);
    }
}

// Quick Sort
int partition(std::vector<int>& data, int low, int high) {
    int pivot = data[high]; // pivot element can be chosen differently
    int i = (low - 1); // index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (data[j] <= pivot) {
            i++;
            std::swap(data[i], data[j]);
        }
    }
    std::swap(data[i + 1], data[high]);
    return (i + 1);
}

void quickSort(std::vector<int>& data, int low, int high) {
    if (low < high) {
        // pi is partitioning index, data[p] is now at right place
        int pi = partition(data, low, high);

        // Recursively sort elements before and after partition
        quickSort(data, low, pi - 1);
```

```
        quickSort(data, pi + 1, high);
    }
}
```

Comparisons

Algorithm	Time Complexity (Average/Worst)	Space Complexity	Stability	Comments
Bubble Sort	O(n^2)	O(1)	Yes	Inefficient for large datasets.
Selection Sort	O(n^2)	O(1)	Yes	Similar performance to Bubble Sort.
Insertion Sort	O(n^2) (worst) O(n) (nearly sorted)	O(1)	Yes	Can be useful for small or partially sorted data.
Merge Sort	O(n log n)	O(n)	Yes	Generally efficient, good for large datasets.
Quick Sort (average)	O(n log n)	O(log n)	No	Efficient on average, but worst-case O(n^2).
Quick Sort (worst)	O(n^2)	O(log n)	No	Can occur with a poorly chosen pivot.
std::sort (C++ STL)	O(n log n) (average)	O(log n)	Yes (implementation dependent)	Efficient for most sorting needs.

2. Creation of Stack

```
#include <iostream>

class Stack {
private:
    int arr[100]; // Array to store stack elements (adjust size as needed)
    int top;

public:
    Stack() {
        top = -1; // Initialize top to -1 (empty stack)
```

```cpp
    }

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

    bool isFull() const {
        return top == 99; // Adjust based on array size
    }

    void push(int data) {
        if (isFull()) {
            std::cout << "Stack overflow\n";
            return;
        }
        arr[++top] = data; // Increment top after assignment
    }

    int pop() {
        if (isEmpty()) {
            std::cout << "Stack underflow\n";
            return -1; // Or throw an exception
        }
        return arr[top--]; // Decrement top after returning element
    }

    int peek() const {
        if (isEmpty()) {
            std::cout << "Stack is empty\n";
            return -1; // Or throw an exception
        }
        return arr[top];
    }
};

int main() {
    Stack myStack;

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

    std::cout << myStack.pop() << std::endl; // Output: 30
    std::cout << myStack.peek() << std::endl; // Output: 20
```

```
    return 0;
}
```

3. Creation of Tree

```
#include <iostream>

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

    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

class Tree {
private:
    Node* root;

public:
    Tree() {
        root = nullptr;
    }

    // Function to insert a node (various insertion strategies can be
implemented)
    void insert(int value) {
        Node* newNode = new Node(value); // Create a new node
        if (root == nullptr) {
            root = newNode; // Set as root for an empty tree
            return;
        }

        // Implement insertion logic here (e.g., recursive or iterative)
        // This example uses a simple recursive approach for illustration
        insertHelper(root, newNode);
    }

    // Helper function for recursive insertion (can be adapted for other
strategies)
```

```
    void insertHelper(Node* current, Node* newNode) {
        if (newNode->data < current->data) {
            if (current->left == nullptr) {
                current->left = newNode;
                return;
            }
            insertHelper(current->left, newNode);
        } else {
            if (current->right == nullptr) {
                current->right = newNode;
                return;
            }
            insertHelper(current->right, newNode);
        }
    }

    // Additional methods for tree operations can be added here (e.g., traversal)
};

int main() {
    Tree myTree;

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

    // Implement additional functionalities as needed (e.g., traversal to
print the tree)
    return 0;
}
```

4. Creation of Graph

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

```
class Node {
public:
```

```cpp
    int data;
    std::vector<int> neighbors; // List of neighboring vertices

    Node(int value) {
        data = value;
    }
};
class Graph {
private:
    int numVertices;
    std::vector<Node*> adjList; // Array of adjacency lists

public:
    Graph(int num) {
        numVertices = num;
        adjList.resize(num); // Allocate space for adjacency lists
        for (int i = 0; i < num; i++) {
            adjList[i] = new Node(i); // Create nodes with vertex data (optional)
        }
    }

    // Function to add an edge (undirected by default)
    void addEdge(int src, int dest) {
        adjList[src]->neighbors.push_back(dest);
// Add destination to source's neighbor list
// For directed graphs, uncomment the line below to add an edge from dest to src
// adjList[dest]->neighbors.push_back(src);
    }

// Additional methods for graph operations can be added here (e.g., traversal)
};
int main() {
    int numVertices = 5;
    Graph myGraph(numVertices);

    myGraph.addEdge(0, 1);
    myGraph.addEdge(0, 4);
    myGraph.addEdge(1, 2);
    myGraph.addEdge(1, 3);
    myGraph.addEdge(2, 2); // Self-loop (optional)
    myGraph.addEdge(3, 0);

   // Implement additional functionalities as needed (e.g., traversal to print the graph)

    return 0;
```

```
}
```

5. Creation of linked list

```
#include <iostream>

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

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class LinkedList {
private:
    Node* head; // Pointer to the head (first) node

public:
    LinkedList() {
        head = nullptr;
    }

    // Function to insert a node at the beginning of the list
    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Function to insert a node at the end of the list (alternative
implementations exist)
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            return;
        }

        Node* temp = head;
```

```cpp
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }

    // Function to delete a node with a specific value (assuming no duplicates)
    void deleteNode(int value) {
        if (head == nullptr) {
            return; // List is empty
        }

        Node* temp = head;
        // Handle deletion of the head node
        if (temp->data == value) {
            head = head->next;
            delete temp;
            return;
        }

        // Traverse and find the node to delete
        while (temp->next != nullptr && temp->next->data != value) {
            temp = temp->next;
        }

        if (temp->next == nullptr) {
            return; // Value not found
        }

        // Delete the node (assuming it's found)
        Node* deleteNode = temp->next;
        temp->next = deleteNode->next;
        delete deleteNode;
    }

    // Additional methods for linked list operations can be added here (e.g., searching, traversal)

   // Function to print the linked list (optional)
    void printList() {
        Node* temp = head;
        while (temp != nullptr) {
            std::cout << temp->data << " -> ";
            temp = temp->next;
        }
        std::cout << "NULL\n";
```

```
    }
};

int main() {
    LinkedList myList;

    myList.insertAtBeginning(50);
    myList.insertAtEnd(30);
    myList.insertAtBeginning(20);
    myList.insertAtEnd(40);

    myList.printList(); // Output: 20 -> 50 -> 30 -> 40 -> NULL

    myList.deleteNode(50);

    myList.printList(); // Output: 20 -> 30 -> 40 -> NULL

    return 0;
}
```

6. Comparison of all above data structures.

Feature	Stack	Tree	Graph	Linked List
Structure	LIFO (Last In, First Out)	Hierarchical (parent-child)	Network of nodes (connected)	Linear sequence of nodes
Operations	Push, Pop, Peek	Insert, Delete, Search, Traversal (preorder, inorder, postorder)	Add vertex, Add edge, Traversal (DFS, BFS)	Insert, Delete, Search, Traversal
Underlying Data	Array or Linked List	Nodes with data and pointers (left, right)	Nodes with data and neighbor lists	Nodes with data and next pointers
Average Time Complexity	O(1) for Push/Pop	O(log n) for Insert/Delete (balanced), O(n) for worst-case	O(V + E) for Traversal (V: vertices, E: edges)	O(n) for most operations (depends on list length)

Space Complexity	O(1) or O(n)	O(n)	O(V + E)	O(n)
Applications	Undo/redo, Function calls, Expression evaluation	Hierarchical data (file systems, organizational charts)	Modeling relationships (social networks, maps)	Dynamic data (e.g., shopping carts, music playlists)

7. Create any one shopping cart or employee management system using any data structures .

```
#include <iostream>
#include <string>

class Product {
public:
    int id;
    std::string name;
    double price;

    Product(int id, const std::string& name, double price) :
        id(id), name(name), price(price) {}
};

class Node {
public:
    Product product;
    Node* next;

    Node(const Product& product) : product(product), next(nullptr) {}
};

class ShoppingCart {
private:
    Node* head;
```

```cpp
public:
    ShoppingCart() {
        head = nullptr;
    }

    // Add a product to the cart
    void addToCart(const Product& product) {
        Node* newNode = new Node(product);
        newNode->next = head;
        head = newNode;
    }

    // Remove a product from the cart by ID
    void removeFromCart(int id) {
        if (head == nullptr) {
            return; // Cart is empty
        }

        Node* temp = head;
        Node* prev = nullptr;

        while (temp != nullptr && temp->product.id != id) {
            prev = temp;
            temp = temp->next;
        }

        if (temp == nullptr) {
            return; // Product not found
        }

        if (prev == nullptr) { // Removing head node
            head = head->next;
        } else {
            prev->next = temp->next;
        }
        delete temp;
    }

    // Calculate the total cart price
    double getTotalPrice() const {
        double total = 0.0;
        Node* temp = head;
        while (temp != nullptr) {
            total += temp->product.price;
            temp = temp->next;
```

```cpp
        }
        return total;
    }

    // Print the cart contents (optional)
    void printCart() const {
        if (head == nullptr) {
            std::cout << "Cart is empty.\n";
            return;
        }

        std::cout << "Cart Items:\n";
        Node* temp = head;
        while (temp != nullptr) {
            std::cout << " - ID: " << temp->product.id
            << ", Name: " << temp->product.name
            << ", Price: $" << temp->product.price << std::endl;
            temp = temp->next;
        }
        std::cout << "Total Price: $" << getTotalPrice() << std::endl;
    }
};

int main() {
    ShoppingCart cart;

    // Add some products to the cart
    cart.addToCart(Product(1, "Shirt", 19.99));
    cart.addToCart(Product(2, "Jeans", 39.95));
    cart.addToCart(Product(3, "Hat", 14.50));

    // Print the cart contents
    cart.printCart();

    // Remove a product from the cart
    cart.removeFromCart(2);

    // Print the cart contents again
    cart.printCart();

    return 0;
}
```