ASSIGNMENT FOR DATA STRUCTURES (EC2022E)

BADHON DATTA PROTTOY

ROLL: B230101EC

EC01

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

1.a)

```cpp
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;
void bubbleSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
int main() {
 int n = 100000;
 vector<int> arr(n);
 srand(time(0));
 for (int &x : arr) x = rand() % 10000;
 clock_t start = clock();
 bubbleSort(arr);
 clock_t end = clock();
 cout << "Bubble Sort Time for " << n << " elements: "
 << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;
 return 0;
}
```

Output:

```
Bubble Sort Time for 100000 elements: 53.842 seconds
Bubble Sort Time for 10000 elements: 1.195 seconds
Bubble Sort Time for 1000 elements: 0.006 seconds
Bubble Sort Time for 100 elements: 0 seconds
```

1.b)

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

```cpp
using namespace std;

void merge(vector<int> &arr, int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];

    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[j++];
}

void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    int n = 100000;
    vector<int> arr(n);
```

```
    srand(time(0));

    for (int &x : arr)
        x = rand() % 10000;

    clock_t start = clock();
    mergeSort(arr, 0, arr.size() - 1);
    clock_t end = clock();

    cout << "Merge Sort Time for " << n << " elements: "
         << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    return 0;
}
```

Output:

```
Merge Sort Time for 100000 elements: 0.062 seconds
Merge Sort Time for 10000 elements: 0.005 seconds
Merge Sort Time for 1000 elements: 0 seconds
Merge Sort Time for 100 elements: 0 seconds
```

1.c)

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

using namespace std;

void insertionSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i], j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
```

```
    }
}

int main() {
    int n = 100000;
    vector<int> arr(n);

    srand(time(0));

    for (int &x : arr)
        x = rand() % 10000;

    clock_t start = clock();
    insertionSort(arr);
    clock_t end = clock();

    cout << "Insertion Sort Time for " << n << " elements: "
         << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    return 0;
}
```

Output:

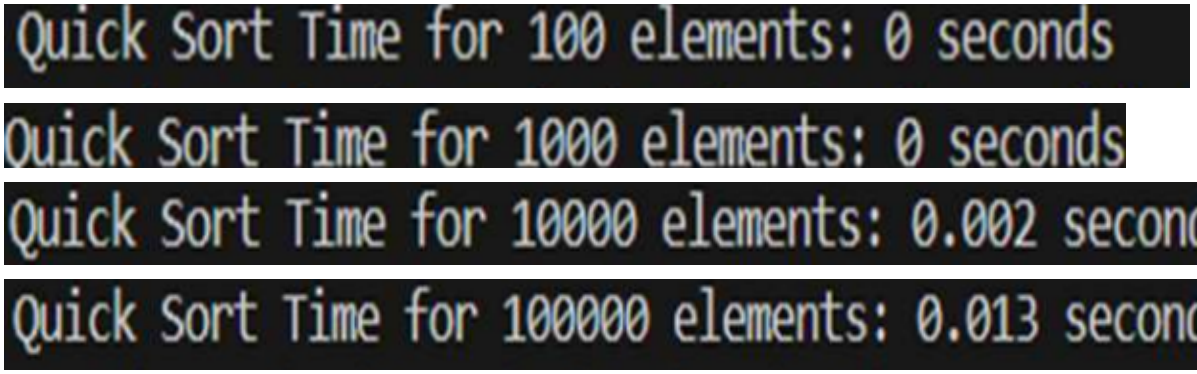
```
Insertion Sort Time for 100 elements: 0 seconds
Insertion Sort Time for 1000 elements: 0 seconds
Insertion Sort Time for 10000 elements: 0.094 seconds
Insertion Sort Time for 100000 elements: 9.614 seconds
```

1.d)

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

using namespace std;
```

```cpp
int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) swap(arr[++i], arr[j]);
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int n = 100000;
    vector<int> arr(n);

    srand(time(0));

    for (int &x : arr)
        x = rand() % 10000;

    clock_t start = clock();
    quickSort(arr, 0, arr.size() - 1);
    clock_t end = clock();

    cout << "Quick Sort Time for " << n << " elements: "
         << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    return 0;
}
```

Output:



Complexity Justification

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)

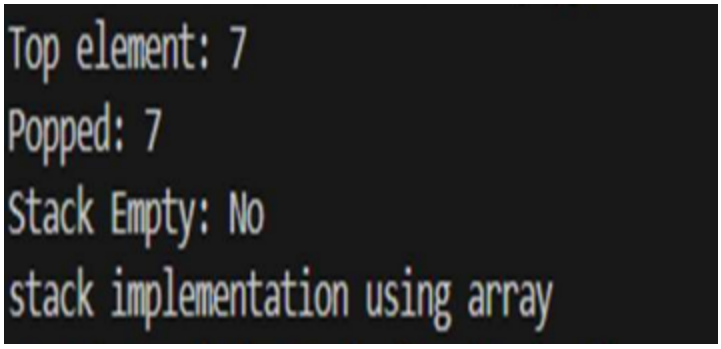
Bubble and Insertion Sort are slow for large datasets due to O(n^2) complexity. Merge and Quick Sort are much faster, operaϴng at O(nlogn) complexity. Quick Sort is the best for most cases but has O(n^2) worst case when elements are already sorted in reverse order.

2a1)

```
#include <iostream>
using namespace std;

class StackArray {
    int top;
    int arr[1000];

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

    void push(int x) {
        if (top >= 999) {
            cout << "Stack Overflow\n";
            return;
        }
        arr[++top] = x;
    }

    int pop() {
        if (top < 0) {
```

```
            cout << "Stack Underflow\n";
            return -1;
        }
        return arr[top--];
    }

    int peek() {
        if (top < 0) return -1;
        return arr[top];
    }

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

int main() {
    StackArray s;
    s.push(5);
    s.push(6);
    s.push(7);

    cout << "Top element: " << s.peek() << endl;
    cout << "Popped: " << s.pop() << endl;
    cout << "Stack Empty: " << (s.isEmpty() ? "Yes" : "No") << endl;
    cout << "Stack implementation using array" << endl;

    return 0;
}
```

Output



2a2)

```cpp
#include <iostream>
using namespace std;

class QueueArray {
    int front, rear, arr[1000];

public:
    QueueArray() {
        front = rear = -1;
    }

    void enqueue(int x) {
        if (rear >= 999) {
            cout << "Queue Overflow\n";
            return;
        }
        if (front == -1) front = 0;
        arr[++rear] = x;
    }

    int dequeue() {
        if (front == -1 || front > rear) {
            cout << "Queue Underflow\n";
            return -1;
        }
        return arr[front++];
    }

    int peek() {
        if (front == -1 || front > rear) return -1;
```

```
        return arr[front];
    }

    bool isEmpty() {
        return front == -1 || front > rear;
    }
};

int main() {
    QueueArray q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << "Front element: " << q.peek() << endl;
    cout << "Dequeued: " << q.dequeue() << endl;
    cout << "Queue Empty: " << (q.isEmpty() ? "Yes" : "No") << endl;
    cout << "Queue implementation using array" << endl;

    return 0;
}
```

Output:

```
Front element: 1
Dequeued: 1
Queue Empty: No
Queue implementation using array
```

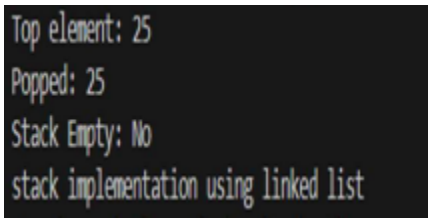
2b1)

```
#include <iostream>
using namespace std;

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

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

```
class StackLinkedList {
    Node* top;

public:
    StackLinkedList() { top = nullptr; }

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

    int pop() {
        if (!top) {
            cout << "Stack Underflow\n";
            return -1;
        }
        int val = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return val;
    }

    int peek() {
        return (top ? top->data : -1);
    }

    bool isEmpty() {
        return top == nullptr;
    }
};

int main() {
    StackLinkedList s;
    s.push(23);
    s.push(24);
    s.push(25);

    cout << "Top element: " << s.peek() << endl;
    cout << "Popped: " << s.pop() << endl;
    cout << "Stack Empty: " << (s.isEmpty() ? "Yes" : "No") << endl;
    cout << "Stack implementation using linked list" << endl; // Fixed text
```

```
    return 0;
}
```

Output:



2b2)

```
#include <iostream>
using namespace std;

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

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

class QueueLinkedList {
    Node *front, *rear;

public:
    QueueLinkedList() {
        front = rear = nullptr;
    }

    void enqueue(int x) {
        Node* newNode = new Node(x);
        if (!rear) {
            front = rear = newNode;
            return;
        }
        rear->next = newNode;
        rear = newNode;
    }

    int dequeue() {
        if (!front) {
```

```
            cout << "Queue Underflow\n";
            return -1;
        }
        int val = front->data;
        Node* temp = front;
        front = front->next;
        if (!front) rear = nullptr; // If queue becomes empty, reset rear
        delete temp;
        return val;
    }

    int peek() {
        return (front ? front->data : -1);
    }

    bool isEmpty() {
        return front == nullptr;
    }
};

int main() {
    QueueLinkedList q;
    q.enqueue(41);
    q.enqueue(42);
    q.enqueue(43);

    cout << "Front element: " << q.peek() << endl;
    cout << "Dequeued: " << q.dequeue() << endl;
    cout << "Queue Empty: " << (q.isEmpty() ? "Yes" : "No") << endl;
    cout << "Queue implementation using linked list" << endl; // Fixed text

    return 0;
}
```

Output:

```
Front element: 41
Dequeued: 41
Queue Empty: No
Queue implementatiuon using linked list
```

3

```
#include <iostream>
```

```
using namespace std;

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

class LinkedList {
private:
    Node* head;
public:
    LinkedList() { head = nullptr; }

    // a. Insert at the beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        newNode->next = head;
        head = newNode;
    }

    // b. Insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next) temp = temp->next;
        temp->next = newNode;
    }

    // c. Insert at a specific position
    void insertAtPosition(int val, int pos) {
        if (pos == 1) {
            insertAtBeginning(val);
            return;
        }
        Node* newNode = new Node(val);
        Node* temp = head;
        for (int i = 1; temp && i < pos - 1; i++)
            temp = temp->next;
```

```
    if (!temp) {
        cout << "Position out of range\n";
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

// d. Sort linked list (Bubble Sort)
void sortList() {
    if (!head || !head->next) return;
    bool swapped;
    do {
        swapped = false;
        Node* temp = head;
        while (temp->next) {
            if (temp->data > temp->next->data) {
                swap(temp->data, temp->next->data);
                swapped = true;
            }
            temp = temp->next;
        }
    } while (swapped);
}

// e. Delete a node
void deleteNode(int val) {
    if (!head) return;
    if (head->data == val) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }
    Node* temp = head;
    while (temp->next && temp->next->data != val)
        temp = temp->next;
    if (!temp->next) {
        cout << "Value not found\n";
        return;
    }
    Node* toDelete = temp->next;
    temp->next = temp->next->next;
    delete toDelete;
```

```
}

// f. Update node value
void updateValue(int oldVal, int newVal) {
    Node* temp = head;
    while (temp) {
        if (temp->data == oldVal) {
            temp->data = newVal;
            return;
        }
        temp = temp->next;
    }
    cout << "Value not found\n";
}

// g. Search for an element
bool search(int val) {
    Node* temp = head;
    while (temp) {
        if (temp->data == val) return true;
        temp = temp->next;
    }
    return false;
}

// h. Display linked list
void display() {
    Node* temp = head;
    while (temp) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}

// i. Reverse linked list
void reverse() {
    Node* prev = nullptr, *curr = head, *next = nullptr;
    while (curr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
```

```
        head = prev;
    }

    // Destructor to free memory
    ~LinkedList() {
        while (head) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
};

int main() {
    LinkedList ll;

    cout << "a. Insert at Beginning:\n";
    ll.insertAtBeginning(10);
    ll.insertAtBeginning(5);
    ll.display();

    cout << "b. Insert at End:\n";
    ll.insertAtEnd(20);
    ll.insertAtEnd(30);
    ll.display();

    cout << "c. Insert at Position (3rd position):\n";
    ll.insertAtPosition(15, 3);
    ll.display();

    cout << "d. Sorting the List:\n";
    ll.sortList();
    ll.display();

    cout << "e. Deleting Node with value 15:\n";
    ll.deleteNode(15);
    ll.display();

    cout << "f. Updating value 20 to 25:\n";
    ll.updateValue(20, 25);
    ll.display();

    cout << "g. Searching for 25: ";
    cout << (ll.search(25) ? "Found" : "Not Found") << endl;
```

```
    cout << "h. Displaying the Linked List:\n";
    ll.display();

    cout << "i. Reversing the Linked List:\n";
    ll.reverse();
    ll.display();

    return 0;
}
```

Output:

```
a. Insert at Beginning:
5 -> 10 -> NULL
b. Insert at End:
5 -> 10 -> 20 -> 30 -> NULL
c. Insert at Position (3rd position):
5 -> 10 -> 15 -> 20 -> 30 -> NULL
d. Sorting the List:
5 -> 10 -> 15 -> 20 -> 30 -> NULL
e. Deleting Node with value 15:
5 -> 10 -> 20 -> 30 -> NULL
f. Updating value 20 to 25:
5 -> 10 -> 25 -> 30 -> NULL
g. Searching for 25: Found
h. Displaying the Linked List:
5 -> 10 -> 25 -> 30 -> NULL
i. Reversing the Linked List:
30 -> 25 -> 10 -> 5 -> NULL
PS D:\CODE\Assignment3>
```