Q1

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <chrono>
#include <fstream>
#include <random>

using namespace std;
using namespace chrono;

namespace sort
{
    void BubbleSort(int* arr, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
            for (int j = 0; j < n - i - 1; j++)
            {
                if (arr[j] > arr[j + 1])
                {
                    std::swap(arr[j], arr[j + 1]);
                }
            }
        }
    }

    // Function to merge two subarrays using a single temporary array
    void merge(int arr[], int temp[], int left, int mid, int right)
    {
        int i = left, j = mid + 1, k = left;
        while (i <= mid && j <= right)
        {
            if (arr[i] <= arr[j])
            {
                temp[k++] = arr[i++];
            }
            else
            {
                temp[k++] = arr[j++];
            }
        }
        while (i <= mid)
```

```
        {
            temp[k++] = arr[i++];
        }
        while (j <= right)
        {
            temp[k++] = arr[j++];
        }
    }

    // Iterative Merge Sort Function with pointer swapping
    void MergeSort(int arr[], int n)
    {
        int* temp = new int[n];
        int* src = arr;
        int* dest = temp;

        for (int currSize = 1; currSize < n; currSize *= 2)
        {
            for (int left = 0; left < n; left += 2 * currSize)
            {
                int mid = std::min(left + currSize - 1, n - 1);
                int right = std::min(left + 2 * currSize - 1, n - 1);
                merge(src, dest, left, mid, right);
            }
            std::swap(src, dest); // Swap the pointers instead of copying
elements
        }

        if (src != arr)
        {
            for (int i = 0; i < n; i++)
            {
                arr[i] = src[i];
            }
        }
        delete[] temp;
    }

    void QuickSort(int* arr, int n)
    {
        using namespace std;
        stack<pair<int, int>> s;
        s.push({ 0, n - 1 });

        while (!s.empty())
        {
            int low = s.top().first;
            int high = s.top().second;
```

```
            s.pop();

            if (low >= high)
                continue;

            int pivot = arr[high];
            int i = low - 1;
            for (int j = low; j < high; j++)
            {
                if (arr[j] < pivot)
                {
                    i++;
                    swap(arr[i], arr[j]);
                }
            }
            swap(arr[i + 1], arr[high]);
            int p = i + 1;

            // Push right and left subarrays to stack
            s.push({ low, p - 1 });
            s.push({ p + 1, high });
        }
    }

    void insertionSort(int arr[], int n)
    {
        for (int i = 1; i < n; i++)
        {
            int key = arr[i];
            int j = i - 1;

            // Move elements that are greater than key one position ahead
            while (j >= 0 && arr[j] > key)
            {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

}

void printArray(int arr[], int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}

// Function to generate a random array
vector<int> generate_random_array(int size)
{
    vector<int> arr(size);
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> dist(0, 1000000);
    generate(arr.begin(), arr.end(), [&]()
        { return dist(gen); });
    return arr;
}

// Function to measure the execution time of a sorting algorithm
using SortFunction = void (*)(int*, int);

double measure_sort_time(SortFunction sort_func, int size, int trials = 3)
{
    double total_time = 0.0;
    for (int i = 0; i < trials; i++)
    {
        vector<int> arr = generate_random_array(size);
        auto start = high_resolution_clock::now();
        sort_func(arr.data(), size);
        auto end = high_resolution_clock::now();
        total_time += duration<double, milli>(end - start).count();
    }
    return total_time / trials;
}

int main()
{
    vector<SortFunction> algorithms = { sort::QuickSort, sort::MergeSort,
sort::BubbleSort , sort::insertionSort};
    vector<string> algorithm_names = { "QuickSort", "MergeSort", "BubbleSort",
"InsertionSort"};

    int min_size = 100;     // Minimum number of elements
    int max_size = 100000; // Maximum number of elements
    int steps = 3;         // Number of points between min and max

    ofstream file("sorting_results.csv");
    file << "Elements";
    for (const auto& name : algorithm_names)
        file << "," << name;
```

```
    file << endl;

    for (int i = 0; i <= steps; i++)
    {
        int size = min_size * pow(max_size / min_size, (double)i / steps);
        file << size;
        cout << "Testing size: " << size << endl;

        for (auto& sort_func : algorithms)
        {
            double avg_time = measure_sort_time(sort_func, size);
            file << "," << avg_time;
        }
        file << endl;
    }
    file.close();

    cout << "Results saved in sorting_results.csv" << endl;
    return 0;
}
```

Results

Time in milliseconds

Elements	QuickSort	MergeSort	BubbleSort	InsertionSort
100	0.00393333	0.00256667	0.00583333	0.00176667
158	0.00926667	0.00723333	0.0216333	0.00673333
251	0.00843333	0.00656667	0.0247667	0.00813333
398	0.0127333	0.0110333	0.054	0.0191667
630	0.021	0.0181	0.125967	0.0459333
1000	0.0631667	0.0587667	0.303233	0.1139
1584	0.0570333	0.0533	0.727733	0.276367
2511	0.164433	0.127133	1.79013	0.6929
3981	0.274733	0.257867	4.42333	1.7003
6309	0.430367	0.2646	11.0198	4.26983
10000	0.5328	0.4254	27.0893	10.851
15848	0.759067	0.681567	67.4676	27.051
25118	1.25153	1.1399	169.429	68.4845
39810	2.48763	1.8654	412.454	168.152
63095	3.54997	3.07447	1100.91	418.434
100000	4.96023	5.0366	4852.21	1053.18

The results obtained from the actual execution times do not perfectly match the theoretical complexity analysis due to factors such as caching differences, memory access patterns, and compiler optimizations.

Both Quicksort and Merge Sort have (O(n \log n)) complexity, and their actual execution times grow at a similar rate. The time taken by Quicksort and Merge Sort is much lower than Bubble Sort and Insertion Sort for larger (n). Quicksort is slightly faster than Merge Sort for smaller (n) but becomes slightly slower for (n = 100000), possibly due to its recursive nature or pivot selection.

The time taken by Bubble Sort follows an (O(n^2)) pattern, increasing dramatically as (n) grows. At (n = 100000), Bubble Sort takes 5277.39 seconds, which is orders of magnitude larger than Quicksort and Merge Sort.

Insertion Sort, similar to Bubble Sort, follows (O(n2)) complexity. However, it performs better than Bubble Sort because it involves fewer swaps. At (n = 100), it is the fastest, but it quickly becomes impractical for large (n).

Q2 a

Stack using array

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

template <typename T>
class Stack
{
private:
    int top;
    int capacity;
    T *arr;

public:
    // Constructor
    Stack(int size)
    {
        capacity = size;
        arr = new T[capacity];
        top = -1;
    }

    // Destructor
    ~Stack()
    {
        delete[] arr;
```

```
    }

    // Push operation
    void push(T value)
    {
        if (top == capacity - 1)
        {
            throw overflow_error("Stack Overflow");
        }
        arr[++top] = value;
    }

    // Pop operation
    T pop()
    {
        if (isEmpty())
        {
            throw underflow_error("Stack Underflow");
        }
        return arr[top--];
    }

    // Peek operation
    T peek()
    {
        if (isEmpty())
        {
            throw underflow_error("Stack is Empty");
        }
        return arr[top];
    }

    // Check if stack is empty
    bool isEmpty()
    {
        return top == -1;
    }

    // Check if stack is full
    bool isFull()
    {
        return top == capacity - 1;
    }
};

// Main function for testing
int main()
{
```

```
    cout << "Testing Stack implementation" << endl;
    Stack<int> stack(3);

    // Test pushing elements
    cout << "Pushing elements: 10, 20, 30" << endl;
    stack.push(10);
    stack.push(20);
    stack.push(30);

    // Test overflow condition
    cout << "Attempting to push 40 (should cause overflow)" << endl;
    try
    {
        stack.push(40);
    }
    catch (const exception &e)
    {
        cout << "Caught exception: " << e.what() << endl;
    }

    // Test peek operation
    cout << "Top element (should be 30): " << stack.peek() << endl;

    // Test popping elements
    cout << "Popping elements: " << stack.pop() << " " << stack.pop() << endl;
    cout << "Top element after popping (should be 10): " << stack.peek() <<
endl;

    // Test popping until empty
    cout << "Popping last element: " << stack.pop() << endl;
    cout << "Attempting to pop from empty stack (should cause underflow)" <<
endl;
    try
    {
        stack.pop();
    }
    catch (const exception &e)
    {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

Results

```
Testing Stack implementation
Pushing elements: 10, 20, 30
Attempting to push 40 (should cause overflow)
Caught exception: Stack Overflow
Top element (should be 30): 30
Popping elements: 20 30
Top element after popping (should be 10): 10
Popping last element: 10
Attempting to pop from empty stack (should cause underflow)
Caught exception: Stack Underflow
```

Q2 b

Stack using linked list(the implementation form 3rd question)

```
template <typename T>
class Stack
{
private:
    LinkedList<T> list;

public:
    Stack() {}

    void push(T val)
    {
        list.insertAtFront(val);
    }

    T pop()
    {
        if (isEmpty())
        {
            throw std::out_of_range("Stack is empty");
        }
        T val = list[0];
        list.deleteAt(0);
        return val;
    }

    T peek()
    {
        if (isEmpty())
        {
            throw std::out_of_range("Stack is empty");
        }
```

```
        return list[0];
    }

    bool isEmpty()
    {
        return list.getSize() == 0;
    }

    void printStack()
    {
        list.printList();
    }
};



int main()
{
    Stack<int> stack;

    std::cout << "Pushing values onto stack...\n";
    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.printStack();

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

    std::cout << "Popping values from stack...\n";
    std::cout << "Popped: " << stack.pop() << "\n";
    std::cout << "Popped: " << stack.pop() << "\n";
    stack.printStack();

    std::cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") <<
"\n";

    std::cout << "Popped: " << stack.pop() << "\n";
    std::cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") <<
"\n";

    try
    {
        std::cout << "Attempting to pop from an empty stack...\n";
        stack.pop();
    }
    catch (const std::out_of_range& e)
    {
        std::cerr << "Exception caught: " << e.what() << "\n";
    }
```

```
    return 0;
}
```

Results

```
Pushing values onto stack...
30 -> 20 -> 10 -> NULL
Top element (peek): 30
Popping values from stack...
Popped: 30
Popped: 20
10 -> NULL
Is stack empty? No
Popped: 10
Is stack empty? Yes
Attempting to pop from an empty stack...
Exception caught: Stack is empty
```

Q2 c

Queue using array

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

template <typename T>
class Queue {
private:
    T *arr;
    int front, rear, size, capacity;

public:
    Queue(int capacity) {
        this->capacity = capacity;
        arr = new T[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    ~Queue() {
        delete[] arr;
    }

    void enqueue(T value) {
        if (isFull()) {
```

```
            throw overflow_error("Queue is full!");
        }
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        size++;
    }

    void dequeue() {
        if (isEmpty()) {
            throw underflow_error("Queue is empty!");
        }
        front = (front + 1) % capacity;
        size--;
    }

    T peek() {
        if (isEmpty()) {
            throw underflow_error("Queue is empty!");
        }
        return arr[front];
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
    }

    void display() {
        if (isEmpty()) {
            throw underflow_error("Queue is empty!");
        }
        cout << "Queue elements: ";
        for (int i = 0; i < size; i++) {
            cout << arr[(front + i) % capacity] << " ";
        }
        cout << endl;
    }
};

int main() {
    try {
        Queue<int> q(5);

        cout << "Enqueuing elements: 10, 20, 30, 40, 50" << endl;
        q.enqueue(10);
```

```
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);
        q.display();

        cout << "Attempting to enqueue 60 (should throw exception)" << endl;
        try {
            q.enqueue(60);
        } catch (const exception &e) {
            cerr << "Exception: " << e.what() << endl;
        }

        cout << "Dequeuing two elements." << endl;
        q.dequeue();
        q.dequeue();
        q.display();

        cout << "Front element: " << q.peek() << endl;

        cout << "Dequeuing all elements." << endl;
        q.dequeue();
        q.dequeue();
        q.dequeue();

        cout << "Attempting to dequeue from empty queue (should throw
exception)" << endl;
        try {
            q.dequeue();
        } catch (const exception &e) {
            cerr << "Exception: " << e.what() << endl;
        }
    } catch (const exception &e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Result

```
Enqueuing elements: 10, 20, 30, 40, 50
Queue elements: 10 20 30 40 50
Attempting to enqueue 60 (should throw exception)
Exception: Queue is full!
Dequeuing two elements.
Queue elements: 30 40 50
Front element: 30
Dequeuing all elements.
Attempting to dequeue from empty queue (should throw exception)
Exception: Queue is empty!
```

Q2 d

Queue using linked list(the implementation form 3rd question)

```
template <typename T>
class Queue
{
private:
    LinkedList<T> list;

public:
    void enqueue(T val)
    {
        list.append(val);
    }

    bool dequeue()
    {
        return list.deleteAt(0);
    }

    T front()
    {
        if (isEmpty())
        {
            throw std::out_of_range("Queue is empty");
        }
        return list[0];
    }

    bool isEmpty()
    {
        return list.getSize() == 0;
    }
```

```
    size_t size()
    {
        return list.getSize();
    }

    void printQueue()
    {
        list.printList();
    }
};

int main()
{
    Queue<int> q;
    std::cout << "Enqueueing elements: 10, 20, 30, 40" << std::endl;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.printQueue();

    std::cout << "Front element: " << q.front() << std::endl;
    std::cout << "Queue size: " << q.size() << std::endl;

    std::cout << "Dequeuing two elements" << std::endl;
    q.dequeue();
    q.dequeue();
    q.printQueue();

    std::cout << "Front element after dequeue: " << q.front() << std::endl;
    std::cout << "Queue size after dequeue: " << q.size() << std::endl;

    std::cout << "Dequeuing remaining elements" << std::endl;
    q.dequeue();
    q.dequeue();

    std::cout << "Queue empty? " << (q.isEmpty() ? "Yes" : "No") << std::endl;
    return 0;
}
```

Result

```
Enqueueing elements: 10, 20, 30, 40
10 -> 20 -> 30 -> 40 -> NULL
Front element: 10
Queue size: 4
Dequeuing two elements
30 -> 40 -> NULL
Front element after dequeue: 30
Queue size after dequeue: 2
Dequeuing remaining elements
Queue empty? Yes
```

Q3

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

template <typename T>
class Node
{
public:
    T data;
    Node *next;
    Node(T val, Node *n = NULL) : data(val), next(n) {}
};

template <typename T>
class LinkedList
{
private:
    size_t size;
    Node<T> *head;
    Node<T> *tail;

public:
    LinkedList() : size(0), head(nullptr), tail(nullptr) {}

    LinkedList(const T *arr, size_t len) : size(0), head(nullptr),
tail(nullptr)
    {
        for (size_t i = 0; i < len; ++i)
        {
            append(arr[i]);
        }
    }
```

```
    ~LinkedList()
    {
        while (head)
        {
            Node<T> *temp = head->next;
            delete head;
            head = temp;
        }
    }

    void append(T val)
    {
        Node<T> *newNode = new Node<T>(val);
        if (!tail)
        {
            head = tail = newNode;
        }
        else
        {
            tail->next = newNode;
            tail = newNode;
        }
        size++;
    }

    void insertAtFront(T val)
    {
        head = new Node<T>(val, head);
        if (!tail)
            tail = head;
        size++;
    }

    bool insertAt(size_t pos, T val)
    {
        if (!head || pos >= size)
            return false;

        Node<T> *temp = head;
        for (size_t i = 0; i < pos - 1; ++i)
        {
            temp = temp->next;
        }
        temp->next = new Node<T>(val, temp->next);
        if (temp == tail)
            tail = temp->next;

        size++;
```

```
        return true;
    }

    bool deleteAt(size_t pos)
    {
        if (!head || pos >= size)
            return false;

        Node<T> *temp = head;
        if (pos == 0)
        {
            head = head->next;
            delete temp;
            if (!head)
                tail = nullptr;
            size--;
            return true;
        }

        Node<T> *prev = nullptr;
        for (size_t i = 0; i < pos; ++i)
        {
            prev = temp;
            temp = temp->next;
        }
        prev->next = temp->next;
        if (temp == tail)
            tail = prev;
        delete temp;
        size--;
        return true;
    }

    size_t getSize()
    {
        return size;
    }

    T &operator[](const int index)
    {
        if (index >= 0 && index < size)
        {
            Node<T> *temp = head;
            for (int i = 0; i < index; i++)
            {
                temp = temp->next;
            }
            return temp->data;
```

```
        }
        else
        {
            throw std::out_of_range("Index out of range");
        }
    }

    std::vector<size_t> find(T element)
    {
        std::vector<size_t> indices;
        Node<T> *temp = head;
        for (int i = 0; i < size; i++)
        {
            if (temp->data == element)
            {
                indices.push_back(i);
            }
            temp = temp->next;
        }

        return indices;
    }

    void reverse()
    {
        Node<T> *prev = nullptr, *current = head, *next = nullptr;

        std::swap(head, tail);
        for (int i = 0; i < size; i++)
        {
            next = current->next;
            current->next = prev;
            prev = current;
            current = next;
        }
    }

    void printList() const
    {
        Node<T> *temp = head;
        while (temp)
        {
            std::cout << temp->data << " -> ";
            temp = temp->next;
        }
        std::cout << "NULL" << std::endl;
    }
```

```
void sort()
{
    head = mergeSort(head);

    // Update tail
    if (!head)
    {
        tail = nullptr;
        return;
    }
    Node<T> *temp = head;
    while (temp->next)
    {
        temp = temp->next;
    }
    tail = temp;
}

Node<T> *mergeSortedLists(Node<T> *left, Node<T> *right)
{
    if (!left)
        return right;
    if (!right)
        return left;

    if (left->data < right->data)
    {
        left->next = mergeSortedLists(left->next, right);
        return left;
    }
    else
    {
        right->next = mergeSortedLists(left, right->next);
        return right;
    }
}

Node<T> *mergeSort(Node<T> *head)
{
    if (!head || !head->next)
        return head;

    Node<T> *slow = head, *fast = head->next;
    while (fast && fast->next)
    {
        slow = slow->next;
        fast = fast->next->next;
    }
```

```
        Node<T> *mid = slow->next;
        slow->next = nullptr;

        Node<T> *left = mergeSort(head);
        Node<T> *right = mergeSort(mid);

        return mergeSortedLists(left, right);
    }
};

int main()
{
    // Test default constructor
    LinkedList<int> list;

    // Test append function
    list.append(10);
    list.append(20);
    list.append(30);
    list.append(40);
    list.append(50);
    list.append(30);
    std::cout << "List after append: ";
    list.printList();

    // Test insertAtFront
    list.insertAtFront(5);
    std::cout << "List after inserting at front: ";
    list.printList();

    // Test insertAt
    list.insertAt(2, 15);
    std::cout << "List after inserting 15 at position 2: ";
    list.printList();

    // Test deleteAt
    list.deleteAt(3);
    std::cout << "List after deleting element at position 3: ";
    list.printList();

    // Test operator[]
    try
    {
        std::cout << "Element at index 2: " << list[2] << std::endl;
    }
    catch (const std::out_of_range &e)
    {
```

```
        std::cerr << e.what() << std::endl;
    }

    // Test find function
    std::vector<size_t> indices = list.find(30);
    std::cout << "Indices of element 30: ";
    for (size_t index : indices)
    {
        std::cout << index << " ";
    }
    std::cout << std::endl;

    // Test reverse function
    list.reverse();
    std::cout << "List after reversing: ";
    list.printList();

    // Test sorting function
    list.sort();
    std::cout << "List after sorting: ";
    list.printList();

    return 0;
}
```

Result

```
List after append: 10 -> 20 -> 30 -> 40 -> 50 -> 30 -> NULL
List after inserting at front: 5 -> 10 -> 20 -> 30 -> 40 -> 50 -> 30 -> NULL
List after inserting 15 at position 2: 5 -> 10 -> 15 -> 20 -> 30 -> 40 -> 50 -> 30 -> NULL
List after deleting element at position 3: 5 -> 10 -> 15 -> 30 -> 40 -> 50 -> 30 -> NULL
Element at index 2: 15
Indices of element 30: 3 6
List after reversing: 30 -> 50 -> 40 -> 30 -> 15 -> 10 -> 5 -> NULL
List after sorting: 5 -> 10 -> 15 -> 30 -> 30 -> 40 -> 50 -> NULL
```