Algorithm

Traversing in Linear Array

```
Start
Step 1: Read the value of n
Step 2: Declare an array arr of size n
Step 3: Repeat for i = 0 to n - 1
    Read arr[i]
Step 4: Repeat for i = 0 to n - 1
    Print arr[i]
End
```

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

int main() {
    int n;
    cin >> n;
    int arr[n];

    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Bubble Sort

```
Start
Step 1: Repeat for i = 0 to n - 2
Step 2:     Repeat for j = 0 to n - i - 2
Step 3:         If arr[j] > arr[j + 1] then
Step 4:             Swap arr[j] and arr[j + 1]
Step 5:         End of If Structure
Step 6:     End of inner loop
Step 7: End of outer loop
End
```

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

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]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
int main() {
    int n;
    cin >> n;
    int arr[n];

    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    bubbleSort(arr, n);

    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Time Complexity:

- Best Case: O(n) (when array is already sorted and optimized with a flag)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Worst Case Time Complexity Calculation for Bubble Sort

In the worst case, the array is sorted in reverse order. Every element needs to be compared and swapped.

- Outer loop runs from i = 0 to n-2 → (n-1) times
- Inner loop runs from j = 0 to n-i-2 → approximately n times in the first iteration, then n-1, then n-2, ... down to 1

Number of comparisons =
(n-1) + (n-2) + (n-3) + ... + 1
= Sum of first (n-1) natural numbers
= (n-1) * n / 2
= $(n^2 - n) / 2$

Ignoring lower order terms and constants, worst case time complexity is:
$O(n^2)$

Linear Search

```
Start
Step 1: Repeat for i = 0 to n - 1
Step 2:      If arr[i] == key then
Step 3:          Return i
Step 4:      End of If Structure
Step 5: End of loop
Step 6: Return -1
End
```

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

int linearSearch(int arr[], int n, int key) {
    for(int i = 0; i < n; i++) {
        if(arr[i] == key) {
            return i;
        }
    }
    return -1;
}
```

```cpp
int main() {
    int n;
    cin >> n;
    int arr[n];

    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int key;
    cin >> key;

    int result = linearSearch(arr, n, key);
    if(result != -1) {
        cout << "Found at index " << result << endl;
    } else {
        cout << "Not found" << endl;
    }

    return 0;
}
```

Time Complexity:

- Best Case: O(1) (key found at first position)
- Average Case: O(n)
- Worst Case: O(n)

Binary Search

```
BinarySearch(int arr[], int key, int size)
Start
Step 1: Set low = 0, high = size - 1
Step 2: While low ≤ high
Step 3:     mid = (low + high) / 2
Step 4:     If arr[mid] == key then
Step 5:         Return mid
Step 6:     Else if arr[mid] < key then
Step 7:         low = mid + 1
Step 8:     Else
Step 9:         high = mid - 1
Step 10:    End of If Structure
Step 11: End of loop
Step 12: Return -1
End
```

```
int BinarySearch(int arr[], int key, int size) {
    int low = 0, high = size - 1;
    while(low <= high) {
        int mid = low + (high - low) / 2;
        if(arr[mid] == key) {
            return mid;
        } else if(arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

```cpp
int main() {
    int n;
    cin >> n;
    int arr[n];
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    int key;
    cin >> key;
    int result = BinarySearch(arr, key, n);
    if(result != -1) {
        cout << "Found at index " << result << endl;
    } else {
        cout << "Not found" << endl;
    }
    return 0;
}
```

Time Complexity:

- Best Case: O(1)
- Average Case: O(log n)
- Worst Case: O(log n)

Computational Complexity of Linear Search

- **Maximum comparisons:** Visits all elements in the worst case
- **Number of comparisons:** n - 1 (where n = size of array)
- **Example:** For an array of 1024 elements, max comparisons = 1023

Computational Complexity of Binary Search

- **Maximum comparisons:** Array size halves each iteration
- **Number of comparisons:** $\log_2(n)$ (where n = size of array)
- **Example:** For a sorted array of 1024 elements, max comparisons = 10

Median number of an Array

```
FindMedian(int arr[], int n)
Start
Step 1: Sort the array arr[]
Step 2: If n is odd then
Step 3:     median = arr[n / 2]
Step 4: Else
Step 5:     median = (arr[(n / 2) - 1] + arr[n / 2]) / 2
Step 6: End of If Structure
Step 7: Return median
End
```

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

double FindMedian(int arr[], int n) {
    sort(arr, arr + n);
    if (n % 2 != 0) {
        return arr[n / 2];
    } else {
        return (arr[(n / 2) - 1] + arr[n / 2]) / 2.0;
    }
}
```

```
int main() {
    int n;
    cin >> n;
    int arr[n];
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    double median = FindMedian(arr, n);
    cout << "Median is: " << median << endl;
    return 0;
}
```

Insert an item into a linear array

```
InsertItem(int arr[], int n, int capacity, int item, int pos)
Start
Step 1: If n == capacity then
Step 2:     Return "Array is full"
Step 3: End of If Structure
Step 4: For i = n - 1 down to pos
Step 5:     arr[i + 1] = arr[i]
Step 6: End of loop
Step 7: arr[pos] = item
Step 8: n = n + 1
Step 9: Return n
End
```

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

int InsertItem(int arr[], int n, int capacity, int item, int pos) {
    if(n == capacity) {
        cout << "Array is full" << endl;
        return n;
    }
    for(int i = n - 1; i >= pos; i--) {
        arr[i + 1] = arr[i];
    }
    arr[pos] = item;
    n = n + 1;
    return n;
}
```

```cpp
int main() {
    int capacity;
    cin >> capacity;
    int arr[capacity];
    int n;
    cin >> n;
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    int item, pos;
    cin >> item >> pos;
    n = InsertItem(arr, n, capacity, item, pos);
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Time Complexity of Insert Item in Linear Array:

- **Best Case:** O(1)

  (When insertion is at the end of the array, no shifting needed)

- **Worst Case:** O(n)

  (When insertion is at the beginning, all elements need to be shifted)

- **Average Case:** O(n)

  (On average, about half of the elements are shifted)

Delete an existing item from the array

```
DeleteItem(int arr[], int n, int item)
Start
Step 1: Set pos = -1
Step 2: For i = 0 to n - 1
Step 3:     If arr[i] == item then
Step 4:         pos = i
Step 5:         Break
Step 6:     End of If Structure
Step 7: End of loop
Step 8: If pos == -1 then
Step 9:     Return n (item not found)
Step 10: End of If Structure
Step 11: For i = pos to n - 2
Step 12:    arr[i] = arr[i + 1]
Step 13: End of loop
Step 14: n = n - 1
Step 15: Return n
End
```

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

int DeleteItem(int arr[], int n, int item) {
    int pos = -1;
    for(int i = 0; i < n; i++) {
        if(arr[i] == item) {
            pos = i;
            break;
        }
    }
    if(pos == -1) {
        return n;
    }
    for(int i = pos; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    n = n - 1;
    return n;
}
```

```cpp
int main() {
    int n;
    cin >> n;
    int arr[n];
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    int item;
    cin >> item;
    n = DeleteItem(arr, n, item);
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Time Complexity:

- Best Case: O(n) (to find the item)
- Worst Case: O(n) (to find and shift elements)
- Average Case: O(n)

Representation of Records in memory: Parallel Array

```
StoreStudentInfo()
Start
Step 1: Define structure Student with id, name, and marks
Step 2: Declare array of Student
Step 3: For i = 0 to n - 1
Step 4:     Input student id, name, marks
Step 5: End of loop
Step 6: For i = 0 to n - 1
Step 7:     Output student id, name, marks
Step 8: End of loop
End
```

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

struct Student {
    int id;
    string name;
    float marks;
};
```

```cpp
int main() {
    int n;
    cin >> n;
    Student students[n];
    for(int i = 0; i < n; i++) {
        cin >> students[i].id;
        cin.ignore();
        getline(cin, students[i].name);
        cin >> students[i].marks;
    }
    for(int i = 0; i < n; i++) {
        cout << "ID: " << students[i].id << endl;
        cout << "Name: " << students[i].name << endl;
        cout << "Marks: " << students[i].marks << endl;
    }
    return 0;
}
```

Pointer

```
int x = 5;
int *p = &x;
int **q = &p;

**q = **q + 2;
```

`*p` → 5

`*q` → p

* `**q` → p → x

`**q = **q + 2` → `x = 5 + 2 = 7`

Answer is correct: B) 7

```
int a = 3, b = 9;
int *p1 = &a;
int *p2 = &b;
p1 = p2;
*p1 = 6;
```

`p1` now points to `b`

`*p1 = 6` → updates `b`

`a` remains unchanged

Answer is correct: A) a = 3, b = 6

```
int arr[] = {1, 2, 3, 4};
int *p = arr;

p += 2;
*p = 10;
```

`p` points to `arr[0]`

`p += 2` → `p` now points to `arr[2]`

`*p = 10` → `arr[2] = 10`

Answer is correct: B) 10

```
int a = 5;
int *p = &a;

cout << *(p + 1);
```

Accessing *(p + 1) means accessing memory after `a`

No guarantee what exists there

Answer is correct: D) Undefined behavior

```c
void update(int *p) {
    *p = *p + 10;
}

int x = 3;
update(&x);
```

`*p = 3 + 10` → `x = 13`

Answer is correct: B) 13

```c
void swap(int *a, int *b) {
    int *temp = a;
    a = b;
    b = temp;
}

int x = 1, y = 2;
swap(&x, &y);
```

This swaps pointer values **inside the function only**

`x` and `y` remain unchanged

Answer is correct: B) x = 1, y = 2

```c
int a = 7;
int *p = &a;

(*p)++;
```

`(*p)++` → `a = 7 + 1 = 8`

Answer is correct: B) 8

```c
int a = 1, b = 2;
int *p = &a;
int **q = &p;
*p = b;
**q = *p + 3;
```

Answer is correct: C) 5

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

int main() {
    int a = 3, b = 7, c = 2;

    int *p = &a;
    int *q = &b;
    int *r = &c;

    int **pp = &p;
    int **qq = &q;

    **pp = **pp + *r;       // a = a + c = 3 + 2 = 5
    *qq = *qq - *p;         // b = b - a = 7 - 5 = 2
    *r = *r + **qq;         // c = 2 + b = 2 + 2 = 4

    cout << a << " " << b << " " << c;
    return 0;
}
```

`a = 3, b = 7, c = 2`

`p = &a, q = &b, r = &c`

`pp = &p, qq = &q`

1. `**pp = **pp + *r;`

→ `a = a + c = 3 + 2 = 5`

→ Now `a = 5`

2. `*qq = *qq - *p;`

→ `b = b - a = 7 - 5 = 2`

→ Now `b = 2`

3. `*r = *r + **qq;`

→ `c = c + b = 2 + 2 = 4`

→ Now `c = 4`

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

void manipulate(int ***x, int **y, int *z) {
    ***x = **y + *z;        // Step 1: a = b + c
    **y = ***x - *z;        // Step 2: b = a - c
    *z = ***x - **y;        // Step 3: c = a - b
}
```

```cpp
int main() {
    int a = 5, b = 6, c = 2;

    int *p1 = &a;
    int *p2 = &b;
    int *p3 = &c;

    int **pp1 = &p1;
    int **pp2 = &p2;

    int ***ppp = &pp1;

    manipulate(ppp, pp2, p3);

    cout << a << " " << b << " " << c;
    return 0;
}
```

Output : 8 6 2

Pointer & Arrays

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

void PointersAndArrays(int *arr, int size) {
    int *ptr = arr;
    for (int i = 0; i < size; i++) {
        cout << *(ptr + i) << " ";
    }
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);

    PointersAndArrays(arr, size);

    return 0;
}
```

Program to describe the pointer to structures

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

struct Student {
    int id;
    char name[50];
    float marks;
};
```

```
void PointerToStructure(struct Student *ptr) {
    cout << "ID: " << ptr->id << endl;
    cout << "Name: " << ptr->name << endl;
    cout << "Marks: " << ptr->marks << endl;
}

int main() {
    struct Student s1 = {101, "Shaon Khan", 88.5};
    struct Student *p = &s1;

    PointerToStructure(p);

    return 0;
}
```

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

struct Student {
    int id;
    char name[50];
    float marks;
};

void UpdateAndDisplayStudents(Student **ptrArr, int n) {
    for (int i = 0; i < n; i++) {
        // Update marks: add i*5 to current marks
        ptrArr[i]->marks += i * 5;

        cout << "Student " << i+1 << " Details:\n";
        cout << "ID: " << ptrArr[i]->id << "\n";
        cout << "Name: " << ptrArr[i]->name << "\n";
        cout << "Updated Marks: " << ptrArr[i]->marks << "\n\n";
    }
}
```

```cpp
int main() {
    Student s1 = {101, "Shaon Khan", 80.0};
    Student s2 = {102, "Ayesha Rahman", 85.5};
    Student s3 = {103, "Imran Hossain", 90.0};

    // Array of pointers to Student
    Student *students[3] = {&s1, &s2, &s3};

    UpdateAndDisplayStudents(students, 3);

    return 0;
}
```

```
Student 1 Details:
ID: 101
Name: Shaon Khan
Updated Marks: 80
Student 2 Details:
ID: 102
Name: Ayesha Rahman
Updated Marks: 90.5
Student 3 Details:
ID: 103
Name: Imran Hossain
Updated Marks: 100
```

Linked List

InsertAtBeginning

```
Start
Step 1: Create a new node named newNode
Step 2: Set newNode->data = value
Step 3: Set newNode->next = head
Step 4: Set head = newNode
End
```

InsertAtEnd

```
Start
Step 1: Create a new node named newNode
Step 2: Set newNode->data = value
Step 3: Set newNode->next = nullptr
Step 4: If head == nullptr then
        Set head = newNode
        Exit
Step 5: Set temp = head
Step 6: Repeat Step 7 while temp->next != nullptr
Step 7: Set temp = temp->next
Step 8: Set temp->next = newNode
End
```

InsertAtPosition

```
Start
Step 1: If position < 1 then
        Print "Invalid position" and Exit

Step 2: If position == 1 then
        Call insertAtBeginning(value)
        Exit

Step 3: Create a new node named newNode
Step 4: Set newNode->data = value
Step 5: Set newNode->next = nullptr

Step 6: Set temp = head
Step 7: Repeat Step 8 from i = 1 to i < position - 1
        If temp == nullptr then
            Print "Position out of bounds" and Exit
Step 8: Set temp = temp->next

Step 9: Set newNode->next = temp->next
Step 10: Set temp->next = newNode
End
```

DeleteFromBeginning

```
Start
Step 1: If head == nullptr then
        Print "List is empty." and Exit

Step 2: Set temp = head
Step 3: Set head = head->next
Step 4: Delete temp
End
```

DeleteFromEnd

```
Start
Step 1: If head == nullptr then
        Print "List is empty." and Exit

Step 2: If head->next == nullptr then
        Delete head
        Set head = nullptr
        Exit

Step 3: Set temp = head
Step 4: Repeat Step 5 while temp->next->next != nullptr
Step 5: Set temp = temp->next

Step 6: Delete temp->next
Step 7: Set temp->next = nullptr
End
```

DeleteFromPosition

```
Start
Step 1: If head == nullptr then
        Print "List is empty." and Exit

Step 2: If position == 1 then
        Call deleteFromBeginning()
        Exit

Step 3: Set temp = head
Step 4: Repeat Step 5 from i = 1 to i < position - 1
        If temp == nullptr or temp->next == nullptr then
            Print "Position out of bounds." and Exit
Step 5: Set temp = temp->next

Step 6: Set delNode = temp->next
Step 7: If delNode == nullptr then
        Print "Position out of bounds." and Exit

Step 8: Set temp->next = delNode->next
Step 9: Delete delNode
End
```

Traverse

```
Start
Step 1: Set temp = head
Step 2: Repeat Steps 3 and 4 while temp != nullptr
Step 3: Print temp->data
Step 4: Set temp = temp->next
Step 5: Print "NULL"
End
```

Code

```
#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;
    }

    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }
```

```
void insertAtEnd(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr)
        temp = temp->next;

    temp->next = newNode;
}
```

```cpp
void insertAtPosition(int value, int position) {
    if (position < 1) {
        cout << "Invalid position.\n";
        return;
    }

    if (position == 1) {
        insertAtBeginning(value);
        return;
    }
```

```cpp
    Node* newNode = new Node(value);
    Node* temp = head;

    for (int i = 1; i < position - 1; ++i) {
        if (temp == nullptr) {
            cout << "Position out of bounds.\n";
            return;
        }
        temp = temp->next;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}
```

```cpp
void deleteFromBeginning() {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;
    head = head->next;
    delete temp;
}
```

```cpp
void deleteFromEnd() {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }

    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }

    Node* temp = head;
    while (temp->next->next != nullptr)
        temp = temp->next;

    delete temp->next;
    temp->next = nullptr;
}
```

```
void deleteFromPosition(int position) {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }

    if (position == 1) {
        deleteFromBeginning();
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1; ++i) {
        if (temp == nullptr || temp->next == nullptr) {
            cout << "Position out of bounds.\n";
            return;
        }
        temp = temp->next;
    }
```

```
    Node* delNode = temp->next;
    if (delNode == nullptr) {
        cout << "Position out of bounds.\n";
        return;
    }
    temp->next = delNode->next;
    delete delNode;
}
```

```cpp
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};
```

```cpp
int main() {
    LinkedList list;

    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.insertAtBeginning(5);
    list.insertAtPosition(25, 4);
    list.display();

    list.deleteFromBeginning();
    list.display();

    list.deleteFromEnd();
    list.display();

    list.deleteFromPosition(2);
    list.display();

    return 0;
}
```

Search Item

```
Start
Step 1: Set temp = head
Step 2: Set position = 1
Step 3: Repeat Step 4 and Step 5 while temp != nullptr
Step 4: If temp->data == target then
            Print "Item found at position", position
            Exit
Step 5: Set temp = temp->next
        Increment position by 1
Step 6: Print "Item not found"
End
```

Code

```cpp
#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;
    }
```

```
void insertAtEnd(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr)
        temp = temp->next;

    temp->next = newNode;
}

void searchItem(int target) {
    Node* temp = head;
    int position = 1;
    while (temp != nullptr) {
        if (temp->data == target) {
            cout << "Item found at position " << position << endl;
            return;
        }
        temp = temp->next;
        position++;
    }
```

```
        cout << "Item not found" << endl;
    }

    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};
```

Circular Linked List

Insert At Beginning

```
Start
Step 1: Create newNode
Step 2: If head == nullptr then
            newNode->next = newNode
            head = newNode
        Else
            temp = head
            While temp->next != head
                temp = temp->next
            newNode->next = head
            temp->next = newNode
            head = newNode
End
```

Insert At End

```
Start
Step 1: Create newNode
Step 2: If head == nullptr then
            newNode->next = newNode
            head = newNode
        Else
            temp = head
            While temp->next != head
                temp = temp->next
            temp->next = newNode
            newNode->next = head
End
```

Insert At Position

```
Start
Step 1: If position < 1 then
            Print "Invalid position"
            Exit
Step 2: If position == 1 then
            Call insertAtBeginning
            Exit
Step 3: Create newNode
Step 4: Set temp = head
Step 5: Loop from i = 1 to position - 2
            If temp->next == head then
                Print "Position out of bounds"
                Exit
            temp = temp->next
Step 6: newNode->next = temp->next
        temp->next = newNode
End
```

Delete From Beginning

```
Start
Step 1: If head == nullptr then
            Print "List is empty"
            Exit
Step 2: If head->next == head then
            Delete head
            head = nullptr
            Exit
Step 3: Set temp = head
        Set last = head
        While last->next != head
            last = last->next
        head = head->next
        last->next = head
        Delete temp
End
```

Delete From End

```
Start
Step 1: If head == nullptr then
            Print "List is empty"
            Exit
Step 2: If head->next == head then
            Delete head
            head = nullptr
            Exit
Step 3: Set temp = head
        Set prev = nullptr
        While temp->next != head
            prev = temp
            temp = temp->next
        prev->next = head
        Delete temp
End
```

Delete From Position

```
Start
Step 1: If head == nullptr then
            Print "List is empty"
            Exit
Step 2: If position == 1 then
            Call deleteFromBeginning
            Exit
Step 3: Set temp = head
        Set prev = nullptr
Step 4: Loop from i = 1 to position - 1
            If temp->next == head then
                Print "Position out of bounds"
                Exit
            prev = temp
            temp = temp->next
Step 5: prev->next = temp->next
        Delete temp
End
```

Display Circular Linked List

```
Start
Step 1: If head == nullptr then
            Print "List is empty"
            Exit
Step 2: Set temp = head
Step 3: Do
            Print temp->data
            temp = temp->next
        While temp != head
End
```

C++ Code

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

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

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

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

```
public:
    CircularLinkedList() {
        head = nullptr;
    }

    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            newNode->next = newNode;
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != head)
            temp = temp->next;
```

```
        newNode->next = head;
        temp->next = newNode;
        head = newNode;
    }

    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            newNode->next = newNode;
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != head)
            temp = temp->next;

        temp->next = newNode;
        newNode->next = head;
    }
```

```
    void insertAtPosition(int value, int position) {
        if (position < 1) {
            cout << "Invalid position.\n";
            return;
        }
        if (position == 1) {
            insertAtBeginning(value);
            return;
        }

        Node* newNode = new Node(value);
        Node* temp = head;
```

```cpp
        for (int i = 1; i < position - 1; ++i) {
            if (temp->next == head) {
                cout << "Position out of bounds.\n";
                return;
            }
            temp = temp->next;
        }

        newNode->next = temp->next;
        temp->next = newNode;
    }
```

```cpp
void deleteFromBeginning() {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }
    if (head->next == head) {
        delete head;
        head = nullptr;
        return;
    }

    Node* temp = head;
    Node* last = head;
    while (last->next != head)
        last = last->next;

    head = head->next;
    last->next = head;
    delete temp;
}
```

```cpp
void deleteFromEnd() {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }
    if (head->next == head) {
        delete head;
        head = nullptr;
        return;
    }

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

    while (temp->next != head) {
        prev = temp;
        temp = temp->next;
    }

    prev->next = head;
    delete temp;
}
```

```cpp
void deleteFromPosition(int position) {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }
    if (position == 1) {
        deleteFromBeginning();
        return;
    }

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

    for (int i = 1; i < position; ++i) {
        if (temp->next == head) {
            cout << "Position out of bounds.\n";
            return;
        }
        prev = temp;
        temp = temp->next;
    }

    prev->next = temp->next;
    delete temp;
}
```

```cpp
    void display() {
        if (head == nullptr) {
            cout << "List is empty.\n";
            return;
        }

        Node* temp = head;
        do {
            cout << temp->data << " -> ";
            temp = temp->next;
        } while (temp != head);

        cout << "(head)\n";
    }
};
```

Circular Queue

Array-based

Algorithm Enqueue

```
Start

Function: enqueue(int item)

Step 1: If count == MAXSIZE then
            Print "OVERFLOW: Queue is full"
            Exit from function

Step 2: If count == 0 then
            Set front = 0
            Set rear = 0
        Else if rear == MAXSIZE - 1 then
            Set rear = 0
        Else
            Set rear = rear + 1

Step 3: queue[rear] = item

Step 4: count = count + 1

Step 5: Print "Inserted: ", item

End
```

Dequeue

```
Start

Function: dequeue()

Step 1: If count == 0 then
            Print "UNDERFLOW: Queue is empty"
            Exit from function

Step 2: item = queue[front]

Step 3: If front == MAXSIZE - 1 then
            Set front = 0
        Else
            Set front = front + 1

Step 4: count = count - 1

Step 5: Print "Deleted: ", item

End
```

Display Function

```
Start

Function: display()

Step 1: If count == 0 then
            Print "Queue is empty"
            Exit from function

Step 2: Set index = front

Step 3: Repeat for i = 0 to count - 1
            Print queue[index]
            Set index = (index + 1) % MAXSIZE

End
```

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

#define MAXSIZE 5

class CircularQueue {
private:
    int queue[MAXSIZE];
    int front, rear, count;

public:
    CircularQueue() {
        front = -1;
        rear = -1;
        count = 0;
    }

    void enqueue(int item) {
        if (count == MAXSIZE) {
            cout << "OVERFLOW: Queue is full.\n";
            return;
        }
```

```
    if (count == 0) {
        front = rear = 0;
    } else if (rear == MAXSIZE - 1) {
        rear = 0;
    } else {
        rear = rear + 1;
    }

    queue[rear] = item;
    count++;
    cout << "Inserted: " << item << "\n";
}
```

```
void dequeue() {
    if (count == 0) {
        cout << "UNDERFLOW: Queue is empty.\n";
        return;
    }

    int item = queue[front];

    if (front == MAXSIZE - 1) {
        front = 0;
    } else {
        front = front + 1;
    }

    count--;
    cout << "Deleted: " << item << "\n";
}
```

```
    void display() {
        if (count == 0) {
            cout << "Queue is empty.\n";
            return;
        }

        cout << "Queue elements: ";
        int index = front;
        for (int i = 0; i < count; i++) {
            cout << queue[index] << " ";
            index = (index + 1) % MAXSIZE;
        }
        cout << "\n";
    }
};
```

Queue by Linked List

Enqueue

```
Start
Step 1: Create newNode
Step 2: Set newNode->data = value
Step 3: Set newNode->next = NULL
Step 4: If rear = NULL Then
          Set front = rear = newNode
          Exit
Step 5: Set rear->next = newNode
Step 6: Set rear = newNode
End
```

Dequeue

```
Start
Step 1: If front = NULL Then
          Print "Underflow"
          Exit
Step 2: Set temp = front
Step 3: Set front = front->next
Step 4: If front = NULL Then
          Set rear = NULL
Step 5: Delete temp
End
```

Display

```
Start
Step 1: If front = NULL Then
          Print "Empty"
          Exit
Step 2: Set temp = front
Step 3: While temp != NULL
          Print temp->data
          Set temp = temp->next
End
```

Code

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *next;
};

class myQueue {
private:
    Node *front;
    Node *rear;

public:
    myQueue() {
        front = rear = nullptr;
    }
```

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

void enqueue(int value) {
    Node *newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    if (rear == nullptr) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}
```

```
  void dequeue() {
      if (isEmpty()) {
          cout << "Underflow" << endl;
          return;
      }

      Node *temp = front;
      front = front->next;

      if (front == nullptr) {
          rear = nullptr;
      }

      delete temp;
  }
```

```
    void display() {
        cout << endl;
        if (isEmpty()) {
            cout << "Empty" << endl;
            return;
        }

        Node* temp = front;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

Stack

Push Algorithm

```
Start
Step 1: If top >= MAX - 1 Then
            Print "Stack Overflow!"
            Exit
Step 2: Increment top by 1
Step 3: Set array[top] = value
Step 4: Print value + " pushed into the stack."
End
```

Pop Algorithm

```
Start
Step 1: If top < 0 Then
            Print "Stack Underflow!"
            Exit
Step 2: Print array[top] + " popped from the stack."
Step 3: Decrement top by 1
End
```

Display Algorithm

```
Start
Step 1: If top < 0 Then
            Print "Stack is empty."
            Exit
Step 2: Print "Stack elements are: "
Step 3: For i from top down to 0
            Print array[i]
End
```

Code

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 100

class Stack {
private:
    int array[MAX];
    int top;

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

    void push(int value) {
        if (top >= MAX - 1) {
            cout << "Stack Overflow!" << endl;
        } else {
            top++;
            array[top] = value;
```

```
            cout << value << " pushed into the stack." << endl;
        }
    }

    void pop() {
        if (top < 0) {
            cout << "Stack Underflow!" << endl;
        } else {
            cout << array[top] << " popped from the stack." << endl;
            top--;
        }
    }
```

```
    void display() {
        if (top < 0) {
            cout << "Stack is empty." << endl;
        } else {
            cout << "Stack elements are: ";
            for (int i = top; i >= 0; i--) {
                cout << array[i] << " ";
            }
            cout << endl;
        }
    }
};
```

Infix to Postfix

Algorithm — Push Operation

```
Start
Step 1: If top >= MAX - 1 Then
            Print "Overflow"
            Exit
Step 2: Increment top by 1
Step 3: Set stk[top] = c
End
```

Algorithm — Pop Operation

```
Start
Step 1: If top == -1 Then
            Print "Underflow"
            Exit
Step 2: Return stk[top]
Step 3: Decrement top by 1
End
```

Algorithm — Peek Operation

```
Start
Step 1: If top == -1 Then
            Print "No elements left"
            Exit
Step 2: Return stk[top]
End
```

Algorithm — Precedence Function

```
Start
Step 1: Switch op
Step 2: Case '^' : Return 3
Step 3: Case '*' or '/' : Return 2
Step 4: Case '+' or '-' : Return 1
Step 5: Default : Return 0
End
```

Algorithm — Infix to Postfix Conversion

```
Start
Step 1: Initialize postfix as empty string
Step 2: For each character c in infix expression
Step 3: If c is operand (a-z, A-Z, 0-9)
            Append c to postfix
Step 4: Else if c == '('
            Push c to stack
Step 5: Else if c == ')'
            While top != -1 and peek() != '('
                Append pop() to postfix
            Pop '(' from stack
Step 6: Else // c is operator
            While top != -1 and precedence(peek()) >= precedence(c)
                Append pop() to postfix
            Push c to stack
Step 7: End For
Step 8: While top != -1
            Append pop() to postfix
Step 9: Print postfix expression
End
```

Code

```cpp
#include<bits/stdc++.h>
using namespace std;

#define MAX 100

char stk[MAX];
int top = -1;

void push(char c){
    if(top >= MAX-1){
        cout << "Overflow" << endl;
    } else {
        stk[++top] = c;
    }
}

char pop(){
    if(top == -1){
        cout << "Underflow" << endl;
    } else {
        return stk[top--];
    }
}
```

```cpp
char peek(){
    if(top == -1){
        cout << "No elements left" << endl;
    } else {
        return stk[top];
    }
}

int precedence(char op){
    switch(op){
        case '^': return 3;
        case '*':
        case '/': return 2;
        case '+':
        case '-': return 1;
        default : return 0;
    }
}
```

```
void infixToPostfix(string infix) {
    string postfix = "";

    for (int i = 0; i < infix.length(); i++) {
        char c = infix[i];

        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {
            postfix += c;
        }
        else if (c == '(') {
            push(c);
        }
        else if (c == ')') {
            while (peek() != '(' && top != -1) {
                postfix += pop();
            }
            if (peek() == '(') pop();
        }
        else {
            while (top != -1 && precedence(peek()) >= precedence(c)) {
                postfix += pop();
            }
            push(c);
        }
    }

    while (top != -1) {
        postfix += pop();
    }

    cout << "Postfix Expression: " << postfix << endl;
}

int main() {
    string infix;
    cout << "Enter Infix Expression (e.g., A+B*(C^D-E)): ";
    cin >> infix;
    infixToPostfix(infix);
    return 0;
}
```

Insertion Sort

Algorithm

```
Start
Step 1: For i = 1 to n-1 do
Step 2:     Set key = arr[i]
Step 3:     Set j = i - 1
Step 4:     While j >= 0 and arr[j] > key do
Step 5:         Set arr[j + 1] = arr[j]
Step 6:         Decrement j by 1
Step 7:     End While
Step 8:     Set arr[j + 1] = key
Step 9: End For
End
```

Code

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

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
}
```

Time Complexity Analysis

Case	Explanation	Complexity
Best Case	Array is already sorted. No shifts needed.	O(n)
Worst Case	Array is reverse sorted. Maximum shifts needed.	$O(n^2)$
Average Case	Random order of elements.	$O(n^2)$

Merge Sort

Algorithm for Merge Function

```
Start
Step 1: Calculate lengths: len1 = mid - start + 1, len2 = end - mid
Step 2: Create arrays left[len1], right[len2]
Step 3: Copy elements from arr[start..mid] to left[]
Step 4: Copy elements from arr[mid+1..end] to right[]
Step 5: Initialize i = 0, j = 0, k = start
Step 6: While i < len1 and j < len2 do
        If left[i] <= right[j] then arr[k] = left[i], i++, k++
        Else arr[k] = right[j], j++, k++
Step 7: Copy remaining elements of left[], if any, to arr[]
Step 8: Copy remaining elements of right[], if any, to arr[]
End
```

Algorithm for MergeSort Function

```
Start
Step 1: If start >= end then return
Step 2: Calculate mid = start + (end - start) / 2
Step 3: Call mergeSort(arr, start, mid)
Step 4: Call mergeSort(arr, mid+1, end)
Step 5: Call merge(arr, start, mid, end)
End
```

Code

```
void merge(int arr[], int start, int mid, int end) {
    int len1 = mid - start + 1;
    int len2 = end - mid;

    int left[len1], right[len2];

    for (int i = 0; i < len1; i++)
        left[i] = arr[start + i];
    for (int j = 0; j < len2; j++)
        right[j] = arr[mid + 1 + j];

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

    while (i < len1 && j < len2) {
        if (left[i] <= right[j])
            arr[k++] = left[i++];
        else
            arr[k++] = right[j++];
    }

    while (i < len1)
        arr[k++] = left[i++];
    while (j < len2)
        arr[k++] = right[j++];
}
```

```cpp
void mergeSort(int arr[], int start, int end) {
    if (start >= end)
        return;

    int mid = start + (end - start) / 2;
    mergeSort(arr, start, mid);
    mergeSort(arr, mid + 1, end);
    merge(arr, start, mid, end);
}

void display(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {12, 31, 35, 8, 32, 17};
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);
    display(arr, n);

    return 0;
}
```

Time Complexity of Merge Sort

Case	Explanation	Complexity
Best Case	Always divides the array and merges, irrespective of order	O(n log n)
Worst Case	Same as best, due to fixed division and merging steps	O(n log n)
Average Case	Same reasoning applies	O(n log n)

Quick Sort

Algorithm for Partition Function

```
Start
Step 1: Set index = start - 1
Step 2: Set pivot = arr[end]
Step 3: For j = start to end - 1 do
        If arr[j] <= pivot then
            index = index + 1
            Swap arr[j] and arr[index]
Step 4: index = index + 1
Step 5: Swap arr[index] and arr[end]
Step 6: Return index
End
```

Algorithm for QuickSort Function

```
Start
Step 1: If start < end then
Step 2:   pivotIndex = partition(arr, start, end)
Step 3:   quickSort(arr, start, pivotIndex - 1)
Step 4:   quickSort(arr, pivotIndex + 1, end)
Step 5: End If
End
```

Code

```
int partition(int arr[], int start, int end) {
    int index = start - 1;
    int pivot = arr[end];

    for (int j = start; j < end; j++) {
        if (arr[j] <= pivot) {
            index++;
            swap(arr[j], arr[index]);
        }
    }
    index++;
    swap(arr[index], arr[end]);
    return index;
}

void quickSort(int arr[], int start, int end) {
    if (start < end) {
        int pivotIndex = partition(arr, start, end);
        quickSort(arr, start, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, end);
    }
}
```

```cpp
int main() {
    int arr[] = {12, 3, 45, 23, 78, 1, 94};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

Time Complexity of Quick Sort

Case	Explanation	Complexity
Best Case	Pivot splits array into nearly equal halves each time	$O(n \log n)$
Worst Case	Pivot is always the smallest or largest element (highly unbalanced)	$O(n^2)$
Average Case	On average, pivot splits array in a balanced way	$O(n \log n)$

Heap

Algo. heapSort

```
Function heapSort(arr, size)
    For i = (size / 2) - 1 down to 0 do
        Call heapify(arr, size, i)
    End For

    For i = size - 1 down to 1 do
        Swap arr[0] and arr[i]
        Call heapify(arr, i, 0)
    End For
End Function
```

Algo. heapify

```
Function heapify(arr, size, i)
1.  largest = i
2.  left = 2 * i + 1
3.  right = 2 * i + 2

4.  If left < size and arr[left] > arr[largest] then
5.      largest = left
6.  End If

7.  If right < size and arr[right] > arr[largest] then
8.      largest = right
9.  End If

10. If largest != i then
11.     Swap arr[i] and arr[largest]
12.     Call heapify(arr, size, largest)
13. End If
End Function
```

Code

```
void heapify(int arr[], int size, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < size && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < size && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, size, largest);
    }
}
```

```
void heapSort(int arr[], int size) {
    for (int i = size / 2 - 1; i >= 0; i--) {
        heapify(arr, size, i);
    }

    for (int i = size - 1; i >= 1; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {4, 23, 4, 56, 1, 93, 56};
    int size = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, size);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity	Space Complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n) (due to recursion)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)