UNIVERSITY OF INFORMATION TECHNOLOGY AND SCIENCES

Course Name: **Data Structure and Algorithm (DSA-1) Lab**

Course Code: **CSE0613212**

Lab Report

On

Searching Algorithm, Sorting Algorithm, LinkedList

Report Submitted By:

Md. Zahid Hasan

ID: **0432410005101016**

Section: **3A1**

Semester: **Spring 25**

Submitted To:

Md. Ismail	**Pabon Shaha**
Lecturer Department of CSE UITS	Lecturer Department of CSE UITS

Submission Date: **16 April,2025**

Table of Content

Program No.: 01

Title:

Linear Search

Objective:

Implement linear search to find a key in an array.

Explanation:

Linear Search is a sequential searching algorithm. It goes through each element of an array one by one until it finds the target value or reaches the end of the array.

Time Complexity:

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

Pseudo Code:

LinearSearch(arr, n, key):

```
1. for i = 0 to n-1
2.    if arr[i] == key
3.        return i
4. return -1
```

Code:

```
#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;
}

int main() {
    int arr[100], n, key;
    cout << "Enter size of array: ";
    cin >> n;
    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) cin >> arr[i];
    cout << "Enter key to search: ";
    cin >> key;
    int result = linearSearch(arr, n, key);
    if (result != -1)
```

```
        cout << "Element found at index: " << result << endl;
    else
        cout << "Element not found." << endl;
    return 0;
}
```

Sample Output:

Enter size of array: 5

Enter elements: 4 2 7 1 5

Enter key to search: 7

Element found at index: 2

Conclusion:

Linear search is simple but not efficient for large datasets. Time complexity: O(n).

PROGRAM 2

Title:

Binary Search

Objective:

Implement binary search for sorted arrays.

Explanation:

Binary search halves the array to find the target. Only works on sorted arrays.

Pseudo Code:

```
BinarySearch(arr, left, right, key):

1. while left <= right
2.     mid = (left + right) / 2
3.     if arr[mid] == key
4.         return mid
5.     else if arr[mid] < key
6.         left = mid + 1
7.     else
8.         right = mid - 1
9. return -1
```

Code:

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

int binarySearch(int arr[], int left, int right, int key) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) left = mid + 1;
        else right = mid - 1;
    }
     return -1;
}

int main() {
    int arr[100], n, key;
    cout << "Enter sorted array size: ";
    cin >> n;
    cout << "Enter sorted elements: ";
    for (int i = 0; i < n; i++) cin >> arr[i];
    cout << "Enter key to search: ";
    cin >> key;
    int result = binarySearch(arr, 0, n-1, key);
    if (result != -1)
        cout << "Element found at index: " << result << endl;
```

```
    else
        cout << "Element not found." << endl;
    return 0;
}
```

Sample Output:

Enter sorted array size: 6

Enter sorted elements: 1 3 5 7 9 11

Enter key to search: 9

Element found at index: 4

Conclusion:

Binary search is efficient but works only on sorted arrays. Time complexity: O (log n).

PROGRAM 3

Title:

Bubble Sort

Objective:

Implement bubble sort to sort an array.

Explanation:

Bubble sort compares adjacent elements and swaps them if they are in the wrong order. This is repeated until the array is sorted.

Pseudo Code:

```
BubbleSort(arr, n):

1. for i = 0 to n-1

2.    for j = 0 to n-i-2

3.       if arr[j] > arr[j+1]

4.          swap(arr[j], arr[j+1])
```

Code:

```
#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]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
}

int main() {
    int arr[100], n;
    cout << "Enter size of array: ";
    cin >> n;
    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) cin >> arr[i];
    bubbleSort(arr, n);
```

```
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Sample Output:

Enter size of array: 5

Enter elements: 64 34 25 12 22

Sorted array: 12 22 25 34 64

Conclusion:

Bubble sort is easy to implement but inefficient for large arrays. Time complexity: O(n^2).

PROGRAM 4

Title:

Insertion Sort

Objective:

Implement insertion sort to sort an array.

Explanation:

Insertion sort builds the final sorted array one item at a time. It places each element at its correct position by comparing with previous elements.

Pseudo Code:

```
InsertionSort(arr, n):

1. for i = 1 to n-1
2.     key = arr[i]
3.     j = i - 1
4.     while j >= 0 and arr[j] > key
5.         arr[j+1] = arr[j]
6.         j = j - 1
7.     arr[j+1] = key
```

Code:

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

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;
    }
}

int main() {
    int arr[100], n;
    cout << "Enter size of array: ";
    cin >> n;
     cout << "Enter elements: ";
```

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

    insertionSort(arr, n);

    cout << "Sorted array: ";

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

    cout << endl;

    return 0;

}
```

Sample Output:

Enter size of array: 6

Enter elements: 5 2 9 1 6 3

Sorted array: 1 2 3 5 6 9

Conclusion:

Insertion sort is efficient for small datasets and partially sorted arrays. Time complexity: O(n^2).

PROGRAM 5

Title:

Selection Sort

Objective:

Implement selection sort to sort an array.

Explanation:

Selection sort repeatedly selects the minimum element from the unsorted part and places it at the beginning.

Pseudo Code:

```
SelectionSort(arr, n):

1. for i = 0 to n-2
2.     minIndex = i
3.     for j = i+1 to n-1
4.         if arr[j] < arr[minIndex]
5.             minIndex = j
6.     swap(arr[i], arr[minIndex])
```

Code:

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

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}

int main() {
    int arr[100], n;
    cout << "Enter size of array: ";
    cin >> n;
    cout << "Enter elements: ";
```

```
    for (int i = 0; i < n; i++) cin >> arr[i];
    selectionSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Sample Output:

Enter size of array: 5

Enter elements: 29 10 14 37 13

Sorted array: 10 13 14 29 37

Conclusion: Selection sort is easy to understand but not very efficient. Time complexity: O(n^2).

PROGRAM 6

Title:

Singly Linked List

Objective:

Implement a singly linked list with insertion and traversal.

Explanation:

A singly linked list is a linear data structure where each element (node) points to the next. It allows dynamic memory allocation and efficient insertions/deletions.

Pseudo Code:

InsertSingly(head, value):

1. Create a new node with value

2. If head is NULL

3.   head = new node

4. Else

5.   Traverse to the last node

6.   Set last node's next to new node

PrintList(head):

1. While head is not NULL

2.   Print head->data

3.   Move head to head->next

Code:

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

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

void insert(Node*& head, int value) {
    Node* newNode = new Node{value, nullptr};
    if (!head) head = newNode;
    else {
        Node* temp = head;
        while (temp->next) temp = temp->next;
        temp->next = newNode;
    }
}

void print(Node* head) {
    while (head) {
```

```
        cout << head->data << " -> ";
        head = head->next;
    }
    cout << "NULL
";
}

int main() {
    Node* head = nullptr;
    int n, val;
    cout << "Enter number of nodes: ";
    cin >> n;
    cout << "Enter node values: ";
    for (int i = 0; i < n; i++) {
        cin >> val;
        insert(head, val);
    }
    print(head);
    return 0;
}
```

Sample Output:

Enter number of nodes: 4

Enter node values: 10 20 30 40

10 -> 20 -> 30 -> 40 -> NULL

Conclusion:

Singly linked lists are efficient for insertions and deletions. Traversal is one-directional. Time complexity of insertion at end.

PROGRAM 7

Title:

Doubly Linked List

Objective:

Implement a doubly linked list with insertion and traversal.

Explanation:

A doubly linked list is a linear data structure where each node contains links to both its previous and next nodes. It allows bidirectional traversal.

Pseudo Code:

InsertDoubly(head, value):

1. Create a new node with value

2. If head is NULL

3. head = new node

4. Else

5. Traverse to the last node

6. Set last node's next to new node

7. Set new node's prev to last node

PrintList(head):

1. While head is not NULL

2. Print head->data

3. Move head to head->next

Code:

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

struct DNode {
    int data;
    DNode* prev;
    DNode* next;
};

void insert(DNode*& head, int value) {
    DNode* newNode = new DNode{value, nullptr, nullptr};
    if (!head) head = newNode;
    else {
        DNode* temp = head;
        while (temp->next) temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }
}
void print(DNode* head) {
```

```
    while (head) {
        cout << head->data << " <-> ";
        head = head->next;
    }
    cout << "NULL
";
}
int main() {
    DNode* head = nullptr;
    int n, val;
    cout << "Enter number of nodes: ";
    cin >> n;
    cout << "Enter node values: ";
    for (int i = 0; i < n; i++) {
        cin >> val;
        insert(head, val);
    }
    print(head);
  return 0;
}
```

Sample Output:

Enter number of nodes: 4

Enter node values: 5 10 15 20

5 <-> 10 <-> 15 <-> 20 <->NULL

Conclusion:

Doubly linked lists support both forward and backward traversal. Time complexity of insertion at end: O(n).

PROGRAM 8

Title:

Circular Linked List

Objective:

Implement a circular linked list with insertion and traversal.

Explanation:

A circular linked list is a variation of a linked list where the last node points back to the first node, forming a loop. It is useful for applications requiring cyclic traversal.

Pseudo Code:

InsertCircular(head, value):

1. Create a new node with value

2. If head is NULL

3. head = new node

4. new node's next = head

5. Else

6. Traverse to the last node

7. last node's next = new node

8. new node's next = head

PrintList(head):

1. If head is NULL, return

2. Initialize temp = head

3. Do

4. Print temp->data

5. temp = temp->next

6. While temp != head

Code:

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

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

void insert(CNode*& head, int value) {
   CNode* newNode = new CNode{value, nullptr};
   if (!head) {
      head = newNode;
```

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

void print(CNode* head) {
    if (!head) return;
    CNode* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(back to head)
";
}
int main() {
    CNode* head = nullptr;
```

```
 int n, val;
   cout << "Enter number of nodes: ";
   cin >> n;
   cout << "Enter node values: ";
   for (int i = 0; i < n; i++) {
       cin >> val;
       insert(head, val);
   }
   print(head);
   return 0;
}
```

Sample Output:

Enter number of nodes: 4

Enter node values: 11 22 33 44

11 -> 22 -> 33 -> 44 -> (back to head)

Conclusion:

Circular linked lists efficiently support looping through data continuously. Time complexity of insertion at end: O(n)