TRIBHUVAN UNIVERSITY

Paschimanchal Campus
Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On:
Lab No: 6
Title:
Submitted By
Name: Anish Karki
Roll No: PAS080BCT007

Submitted To: HPB Sir
Date of submission: 31/07/2025

PRACTICAL-1
STACK USING LINKED LIST

THEORY

A **stack** is a linear data structure that follows the **Last In First Out (LIFO)** principle, where the element inserted last is removed first. The basic operations on a stack are:

- **Push**: Insert an element into the stack.
- **Pop**: Remove the top element from the stack.
- **Peek/Top**: View the top element without removing it.
- **isEmpty**: Check if the stack is empty.

When a **linked list** is used to implement a stack, each node of the list contains two parts:

- **Data**: The value stored.
- **Pointer**: A reference to the next node.

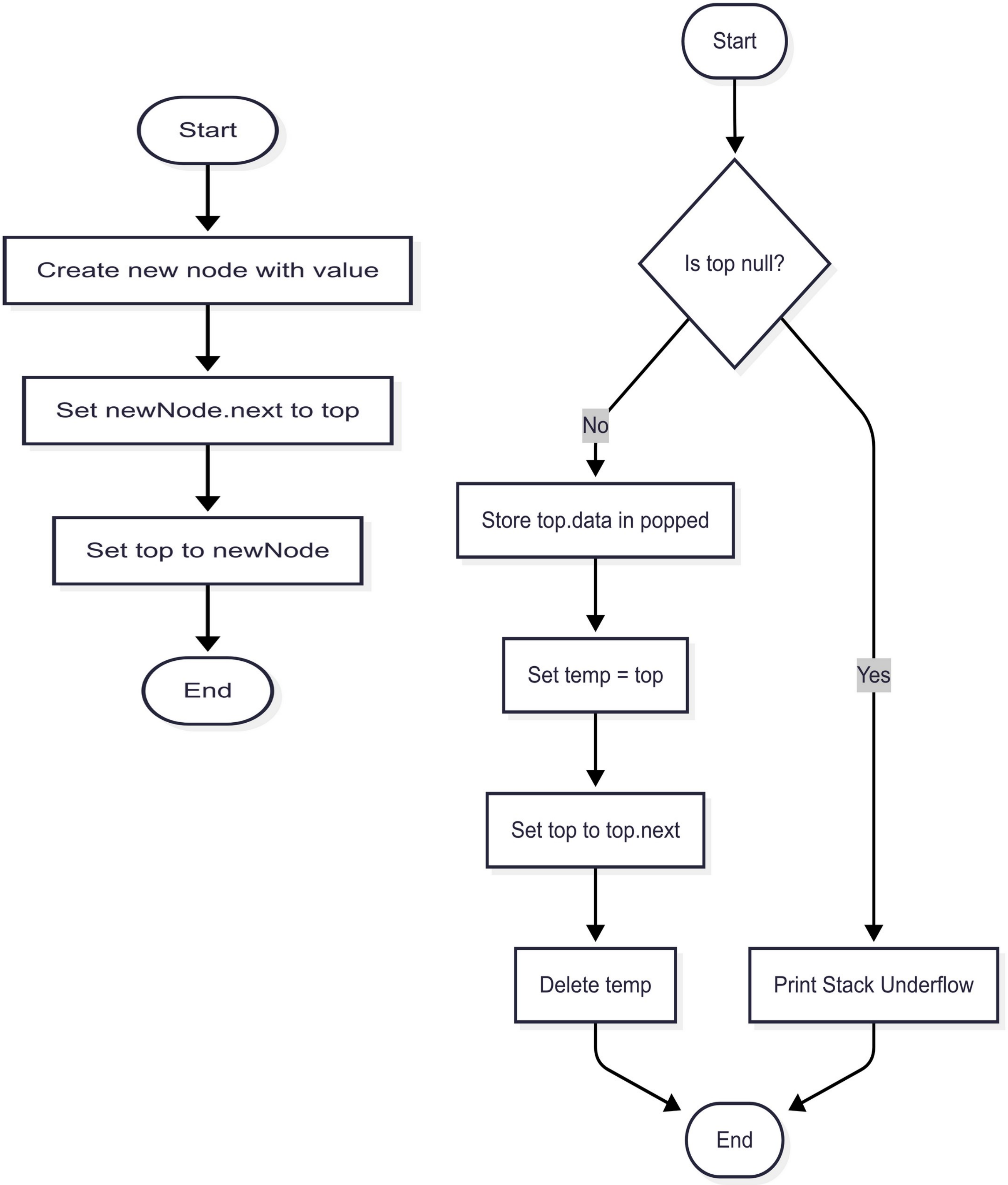
In this implementation:

- The **top of the stack** is represented by the **head** (or first node) of the linked list.
- The **Push operation** inserts a new node at the beginning of the list.
- The **Pop operation** removes the first node of the list.
- This method allows dynamic memory allocation and avoids stack overflow unless memory is full.

ALGORITHM

```
1.Start
2.Define a Node class with data and next pointer.
3.Define a Stack class with a top pointer.
4.isEmpty():
        •Return true if top is nullptr, else false.
5.push(val):
        •Create a new node with value val.
        •Set new node's next to top.
        •Update top to new node.
6.pop():
        •If stack is empty, print "Stack Underflow" and return -1.
        •Store top->data in a variable.
        •Move top to top->next.
        •Delete the old top node.
        •Return the stored value.
7.display():
        •Traverse from top to end.
        •Print each node's data.
8.In main:
        •Create a Stack object.
        •Push values onto the stack.
        •Display the stack.
        •Pop a value and display the stack again.
9.Stop
```

FLOWCHART


Start
Create new node with value
Set newNode.next to top
Set top to newNode
End
Start
Is top null?
No
Yes
Store top.data in popped
Set temp = top
Set top to top.next
Delete temp
Print Stack Underflow
End

CODE:

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

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

class Stack {
private:
    Node* top;
public:
    Stack() : top(nullptr) {}

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

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

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow" << endl;
            return -1;
        }
        int popped = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return popped;
    }

    void display() {
        Node* temp = top;
        cout << "Stack: ";
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.display();
    cout << "Popped: " << s.pop() << endl;
    s.display();

    return 0;
}
```

PRACTICAL-2
QUEUE USING LINKED LIST

THEORY

A **queue** is a linear data structure that follows the **First In First Out (FIFO)** principle, meaning the element inserted first is the one removed first. The basic operations of a queue are:

- **Enqueue**: Insert an element at the rear (end) of the queue.
- **Dequeue**: Remove an element from the front of the queue.
- **isEmpty**: Check if the queue is empty.
- **Front**: Access the element at the front without removing it.

When a **linked list** is used to implement a queue:

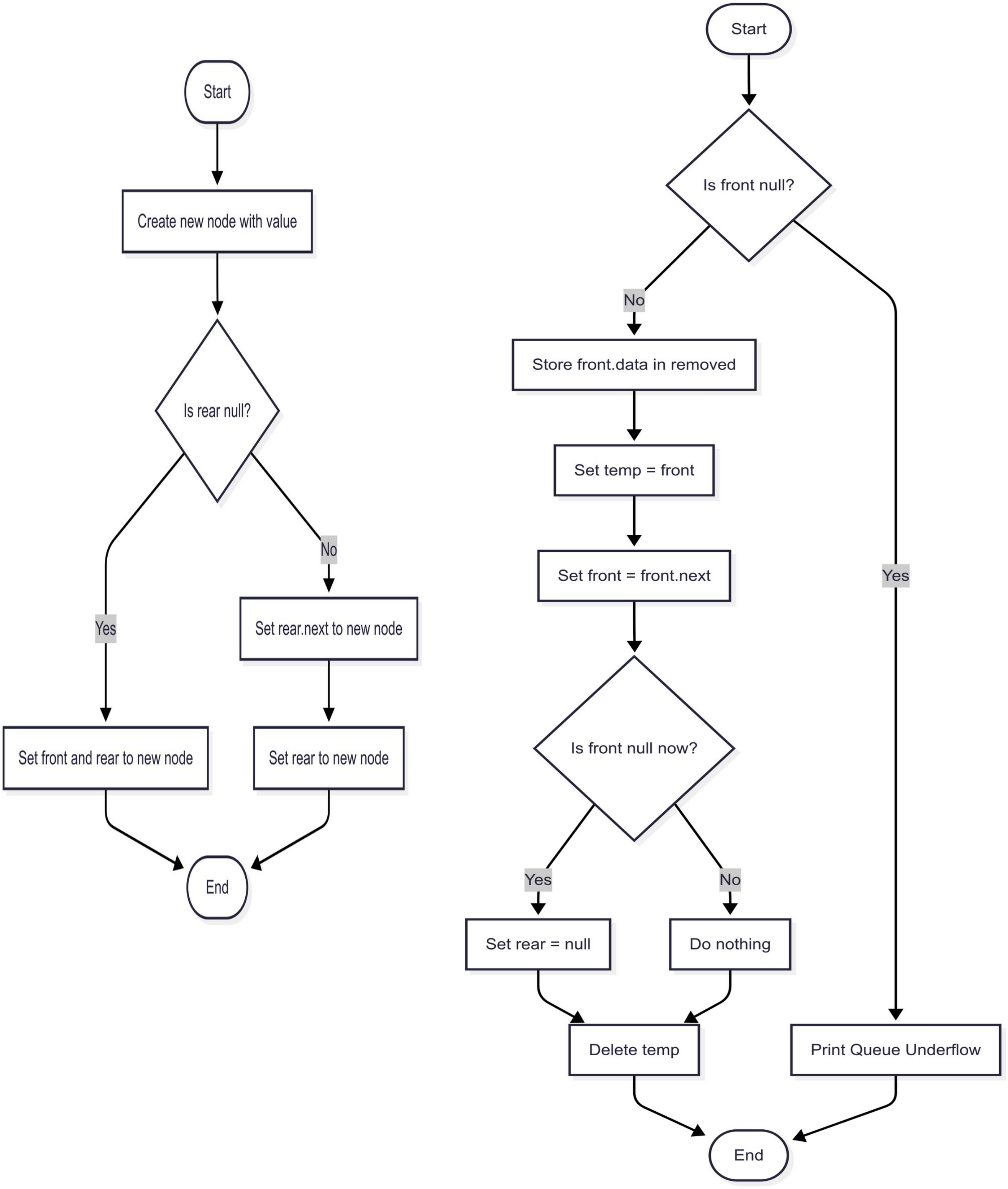
- Each node contains **data** and a **pointer to the next node**.
- The **front** of the queue is the node from where elements are removed.
- The **rear** of the queue is the node where new elements are added.

This dynamic implementation using linked list eliminates the limitation of a fixed-size queue, as seen in array-based implementations.

ALGORITHM

```
1.Start
2.Define a Node class with data and next pointer.
3.Define a Queue class with front and rear pointers.
4. isEmpty():
        •Return true if front is nullptr, else false.

5.enqueue(val):
        •Create a new node with value val.
        •If rear is nullptr, set both front and rear to new node.
        •Else, set rear->next to new node and update rear to new node.

6.dequeue():
        •If queue is empty, print "Queue Underflow" and return -1.
        •Store front->data in a variable.
        •Move front to front->next.
        •If front becomes nullptr, set rear to nullptr.
        •Delete the old front node.
        •Return the stored value.

7.display():
        •Traverse from front to end.
        •Print each node's data.

8.In main:
        •Create a Queue object.
        •Enqueue values into the queue.
        •Display the queue.
        •Dequeue a value and display the queue again.
9.Stop
```

FLOWCHART


Start
Create new node with value
Is rear null?
Yes
No
Set rear.next to new node
Set front and rear to new node
Set rear to new node
End
Start
Is front null?
No
Yes
Store front.data in removed
Set temp = front
Set front = front.next
Is front null now?
Yes
No
Set rear = null
Do nothing
Delete temp
Print Queue Underflow
End

CODE:

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

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

class Queue {
private:
    Node* front;
    Node* rear;
public:
    Queue() : front(nullptr), rear(nullptr) {}

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

    void enqueue(int val) {
        Node* newNode = new Node(val);
        if (rear == nullptr) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow" << endl;
            return -1;
        }
        int removed = front->data;
        Node* temp = front;
        front = front->next;
        if (front == nullptr)
            rear = nullptr;
        delete temp;
        return removed;
    }

    void display() {
        Node* temp = front;
        cout << "Queue: ";
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    cout << "Dequeued: " << q.dequeue() << endl;
    q.display();

    return 0;
}
```

PRACTICAL-3
LINEAR SEARCH

THEORY

Linear Search, also known as sequential search, is the simplest searching algorithm used to find the position of a target element (called the key) in a list or array.

In this method:

Each element in the array is checked one by one from the beginning.

The search continues until either the element is found or the end of the array is reached.

Steps:

1. Start from the first element of the array.
2. Compare each element with the key.
3. If a match is found, return its position (index).
4. If the loop ends without a match, the key is not present in the array.

Characteristics:

Works on both sorted and unsorted arrays.

Very easy to implement.

Not efficient for large datasets.

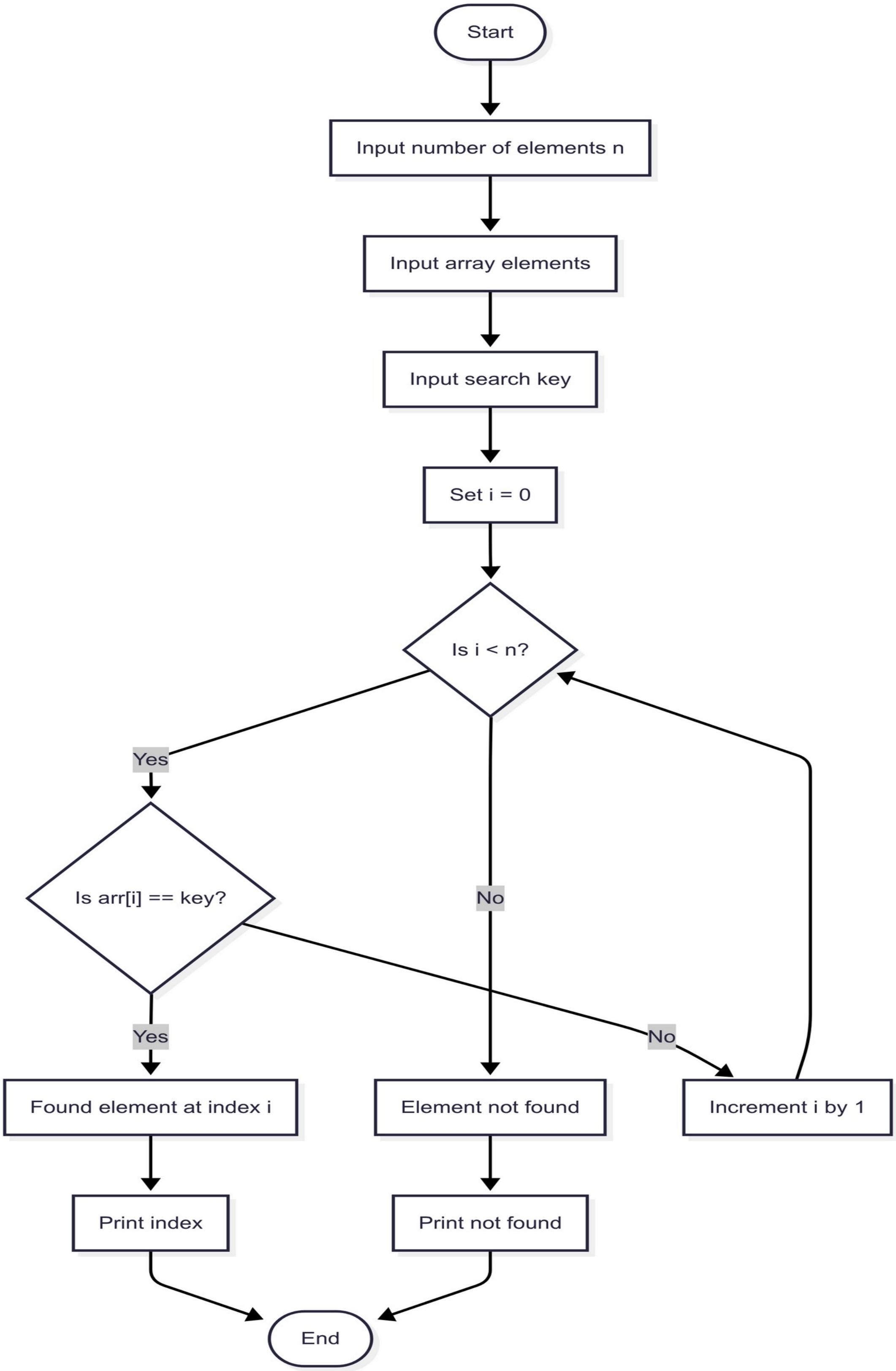
Time Complexity

The best case time complexity of linear search is O(1) when the element is found at the beginning of the array. The worst case time complexity is O(n) when the element is at the end or not present at all. The average case time complexity is also O(n), as it may require checking about half the elements on average.

ALGORITHM

```
1.Start
2.Input the number of elements n.
3.Declare an array arr of size n.
4.Input n elements into arr.
5.Input the search key key.
6.Set index i = 0.
7.Repeat steps 8-9 while i < n:
        1.If arr[i] == key, return i (element found).
        2.Increment i by 1.

8.If loop ends without finding the key, return -1 (element not found).
9.If result is not -1, print "Element found at index result".
10.Else, print "Element not found in the array".
11.Stop
```

FLOWCHART


Start
Input number of elements n
Input array elements
Input search key
Set i = 0
Is i < n?
Yes
No
Is arr[i] == key?
Yes
No
Found element at index i
Element not found
Increment i by 1
Print index
Print not found
End

CODE:

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

int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
    vector<int> arr = {2, 4, 6, 8, 10, 12, 14};
    int target = 10;

    int result = binarySearch(arr, target);

    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```

PRACTICAL-4
BINARY SEARCH

THEORY

Binary Search is an efficient searching algorithm used to find the position of a target element within a **sorted** array or list. It works by repeatedly dividing the search interval in half.
The process starts by comparing the target value to the middle element of the array:

- If the middle element is equal to the target, the search is successful.
- If the target is less than the middle element, the search continues in the left half.
- If the target is greater, the search continues in the right half.

This process is repeated until the target is found or the search interval becomes empty, indicating the target is not present.

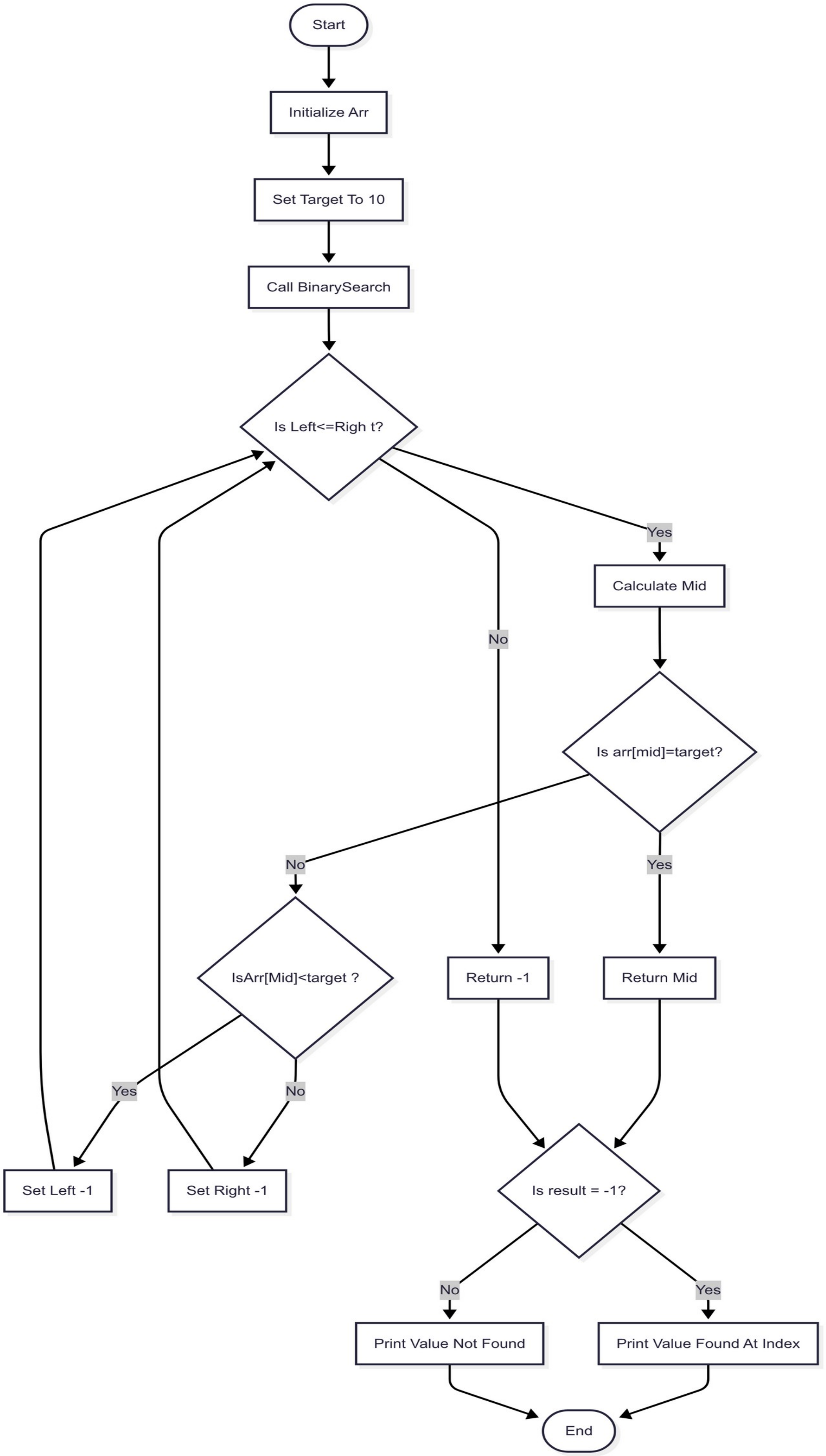
Binary search significantly reduces the search time compared to linear search by eliminating half of the remaining elements at each step.

Time Complexity:
The **time complexity** of binary search is **O(log n)** in the best, average, and worst cases. This is because the search space is halved with each comparison, drastically reducing the number of elements to check at every step until the target is found or the search ends.

ALGORITHM

```
1.Start
2.Initialize left = 0 and right = n - 1 (where n is the size of the array).
3.Repeat while left <= right:
        1.Calculate mid = left + (right - left) / 2.
        2.If arr[mid] == target, return mid (element found).
        3.If arr[mid] < target, set left = mid + 1 (search right half).
        4.If arr[mid] > target, set right = mid - 1 (search left half).

4.If loop ends, return -1 (element not found).
5.In main, call binary search and print result.
6.Stop
```

FLOWCHART


Start
Initialize Arr
Set Target To 10
Call BinarySearch
Is Left<=Righ t?
Yes
Calculate Mid
No
Is arr[mid]=target?
No
Yes
IsArr[Mid]<target ?
Return -1
Return Mid
Yes
No
Set Left -1
Set Right -1
Is result = -1?
No
Yes
Print Value Not Found
Print Value Found At Index
End

CODE:

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

int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
    vector<int> arr = {2, 4, 6, 8, 10, 12, 14};
    int target = 10;

    int result = binarySearch(arr, target);

    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```

CONCLUSION

In this lab, we successfully implemented the fundamental data structures stack and queue using linked lists. These implementations highlighted the advantages of dynamic memory allocation, allowing the data structures to grow and shrink as needed without a fixed size limitation. The linked list-based stack and queue efficiently performed insertion and deletion operations with constant time complexity.

Additionally, we explored two important searching algorithms: linear search and binary search. Linear search, although simple and applicable to both sorted and unsorted data, showed limited efficiency, especially with larger datasets. On the other hand, binary search demonstrated a much faster approach for searching within sorted arrays by systematically halving the search space, resulting in logarithmic time complexity.

Overall, this lab deepened our understanding of how data structures and algorithms function in practical scenarios and emphasized the importance of choosing appropriate methods based on the nature of data and the specific requirements for efficiency.