TRIBHUVAN UNIVERSITY

Paschimanchal Campus

Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On:
Lab No: 5
Title:
Submitted By
Name: Anish Karki
Roll No: PAS080BCT007

Submitted To: HPB Sir
Date of submission: 09/07/2025

PRACTICAL-1
INSERTION OF SINGLE LINKED LIST

THEORY

Insertion in a singly linked list refers to the process of adding a new node to the list at a desired position. A singly linked list is a linear data structure where each element (node) contains two parts: the data and a pointer to the next node.

There are typically three cases of insertion in a singly linked list:

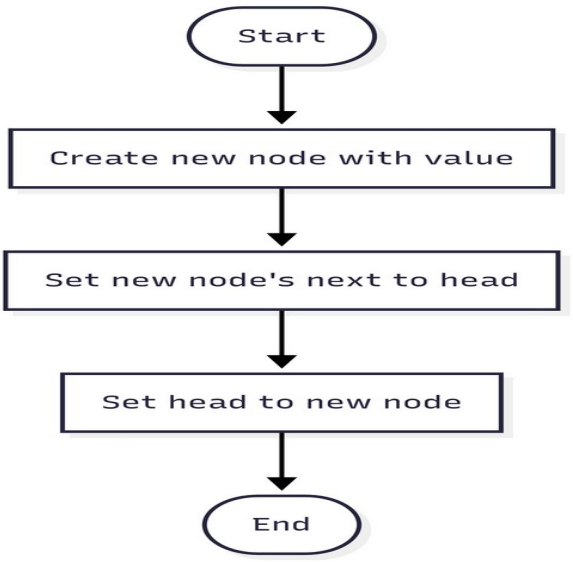
1. **Insertion at the Beginning**:
   A new node is created, and its pointer is set to point to the current head node. Then the head pointer is updated to this new node.
   Time Complexity: **O(1)**
2. **Insertion at the End**:
   A new node is created, and the list is traversed until the last node is reached. The last node's next pointer is set to the new node, and the new node's next pointer is set to NULL.
   Time Complexity: **O(n)**
3. **Insertion at a Specific Position**:
   The list is traversed to the node after which the new node needs to be inserted. The new node's next pointer is set to the next of that node, and the current node's next pointer is updated to point to the new node.
   Time Complexity: **O(n)**

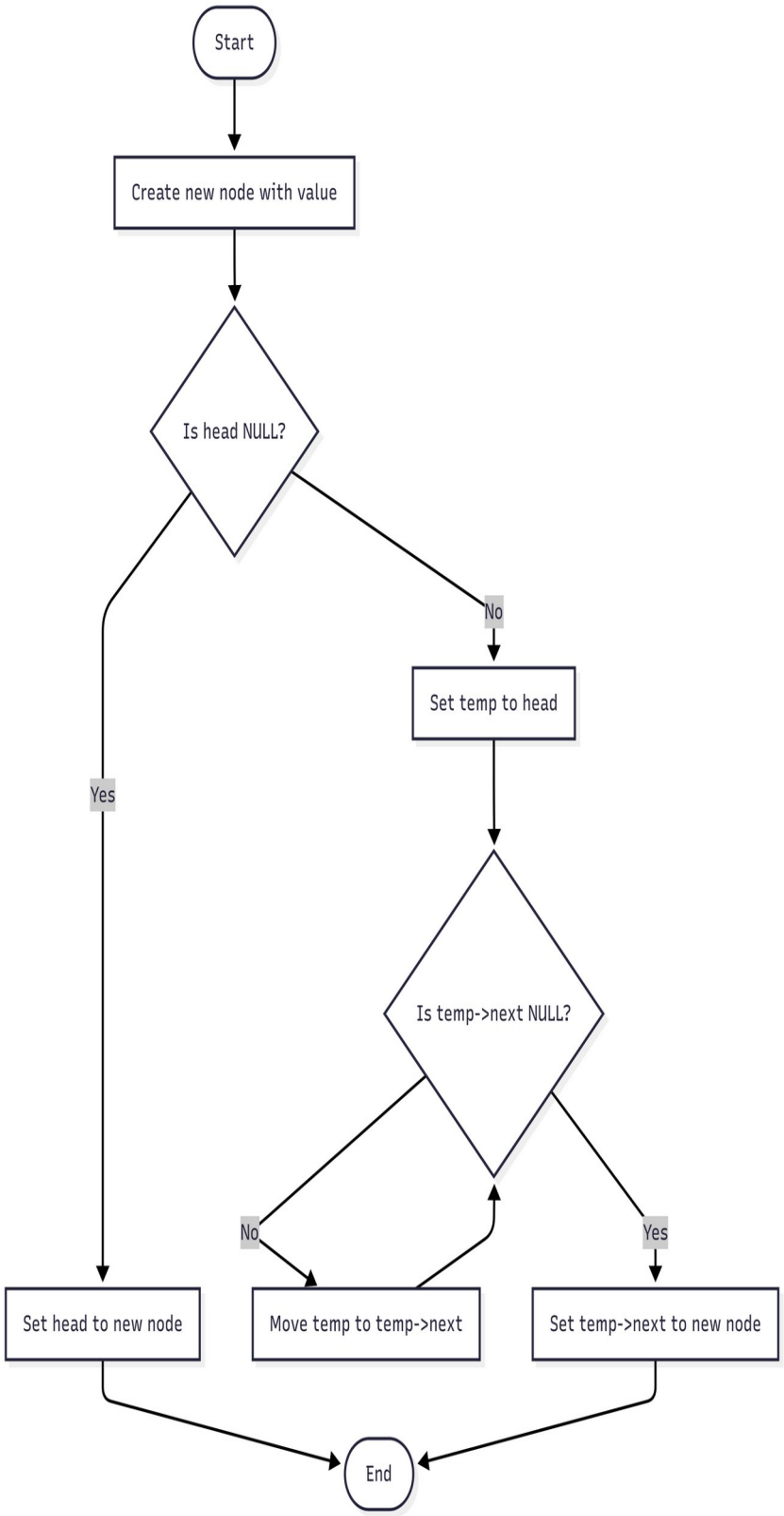
Each insertion operation requires pointer manipulation and, except for insertion at the beginning, may require traversal of the list to reach the correct position. Proper memory allocation and null-pointer checks are important to avoid runtime errors.

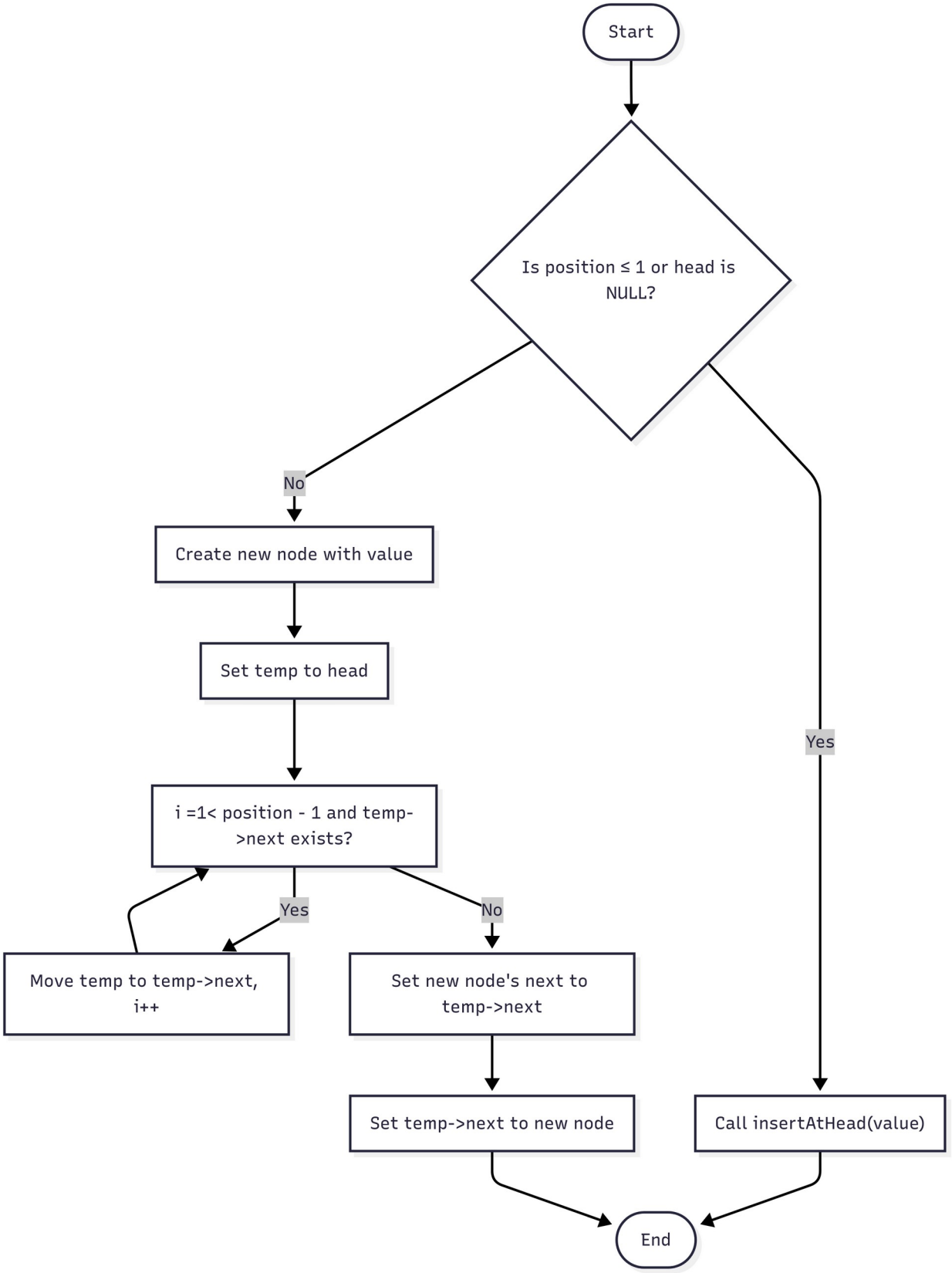
ALGORITHM

1. Start
2. Create a Node class with data and next pointer.
3. Create a LinkedList class with a head pointer.
4. insertAtHead(val):
    1. Create a new node with value val.
    2. Set new node's next to current head.
    3. Update head to new node.
5. insertAtEnd(val):
    1. Create a new node with value val.
    2. If head is NULL, set head to new node.
    3. Else, traverse to the last node.
    4. Set last node's next to new node.
6. insertAtPosition(val, position):
    1. If position is 1 or list is empty, call insertAtHead(val).
    2. Else, create a new node with value val.
    3. Traverse to the node at position-1 or last node.
    4. Set new node's next to current node's next.
    5. Set current node's next to new node.
7. display():
    1. Traverse from head to end.
    2. Print each node's data.
8. In main:
    1. Create a LinkedList object.
    2. Insert values at end, head, and specific position.
    3. Display the list.
9. Stop

FLOWCHART


Start
Create new node with value
Set new node's next to head
Set head to new node
End


Start
Create new node with value
Is head NULL?
No
Yes
Set temp to head
Is temp->next NULL?
No
Yes
Set head to new node
Move temp to temp->next
Set temp->next to new node
End


Start
Is position ≤ 1 or head is NULL?
No
Yes
Create new node with value
Set temp to head
i =1< position - 1 and temp->next exists?
Yes
No
Move temp to temp->next, i++
Set new node's next to temp->next
Set temp->next to new node
Call insertAtHead(value)
End

CODE:

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

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

  Node(int data) {
    this->data = data;
    this->next = NULL;
  }
};

class LinkedList {
  Node* head;

public:
  LinkedList() : head(NULL) {}

  void insertAtHead(int val) {
    Node* newnode = new Node(val);
    newnode->next = head;
    head = newnode;
  }

  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;
  }
```

```
  void insertAtPosition(int val, int position) {
    if (position <= 1 || !head) {
      insertAtHead(val);
      return;
    }
    Node* newnode = new Node(val);
    Node* temp = head;
    for (int i = 1; i < position - 1 && temp->next; i++) {
      temp = temp->next;
    }
    newnode->next = temp->next;
    temp->next = newnode;
  }

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

int main() {

  LinkedList list;
  list.insertAtEnd(10);
  list.insertAtEnd(20);
  list.insertAtHead(5);
  list.insertAtPosition(15, 3);
  cout << "Linked list after insertions: ";
  list.display();
  return 0;

}
```

PRACTICAL-2
CREATION OF DOUBLE LINKED LIST

THEORY

A **doubly linked list** is a linear data structure made up of nodes, where each node contains three fields:

- **Data**: stores the actual value.
- **Prev**: a pointer to the previous node.
- **Next**: a pointer to the next node.

In contrast to a singly linked list, a doubly linked list allows traversal in **both forward and backward** directions because of the additional prev pointer in each node.

Steps for Creation:

1. **Define the Node Structure**: Each node must have three parts — data, prev, and next.
2. **Initialize the Head**: Start with the head pointer as NULL (indicating the list is empty).
3. **Create New Nodes**:
    - Dynamically allocate memory for each new node.
    - Assign data to the node.
    - Set prev and next pointers appropriately.
4. **Link the Nodes**:
    - For the first node, both prev and next are NULL.
    - For other nodes, update the previous node's next to point to the new node, and the new node's prev to point back to the previous node.

Time Complexity:

- Creating a doubly linked list with n nodes takes **O(n)** time.

ALGORITHM

Steps:

1. Start
2. Define a Node class with:
    - •data (integer)
    - •prev (pointer to previous node)
    - •next (pointer to next node)
3. In main:
    - •Create a new node head with data 1.
    - •Create a new node second with data 2.
    - •Create a new node third with data 3.
    - •Set head->next to second.
    - •Set second->prev to head.
    - •Set second->next to third.
    - •Set third->prev to second.
4. Stop

CODE:

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

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

  Node(int num) : data(num), prev(NULL), next(NULL) {}
};

int main() {
  Node* head = new Node(1);
  Node* second = new Node(2);
  Node* third = new Node(3);

  head->next = second;
  second->prev = head;
  second->next = third;
  third->prev = second;

  return 0;

}
```

CONCLUSION

The practical implementation of singly and doubly linked lists in this lab session provided a deeper understanding of dynamic data structures. By performing insertion in a singly linked list, I learned how to manipulate pointers to efficiently add elements at different positions. Similarly, creating a doubly linked list helped me understand bidirectional node linkage using both `prev` and `next` pointers. These exercises reinforced key concepts such as memory allocation, pointer handling, and node connectivity. Overall, the lab enhanced my conceptual clarity and hands-on skills in working with linear data structures, which are fundamental in many areas of computer science and software development.