DSA LAB

SUBMITTED TO: Mr. Rasikh Ali

SUBMITTED BY: RUKHSAR SHAHBAZ

ROLL NO: BSSEM-S24-058

Lab Manual

Lab 1

- **Pointers:**

  Task 1:

  Create a program that declares an integer variable and a pointer to it. Modify the value of the variable using the pointer and display both the variable and pointer values.

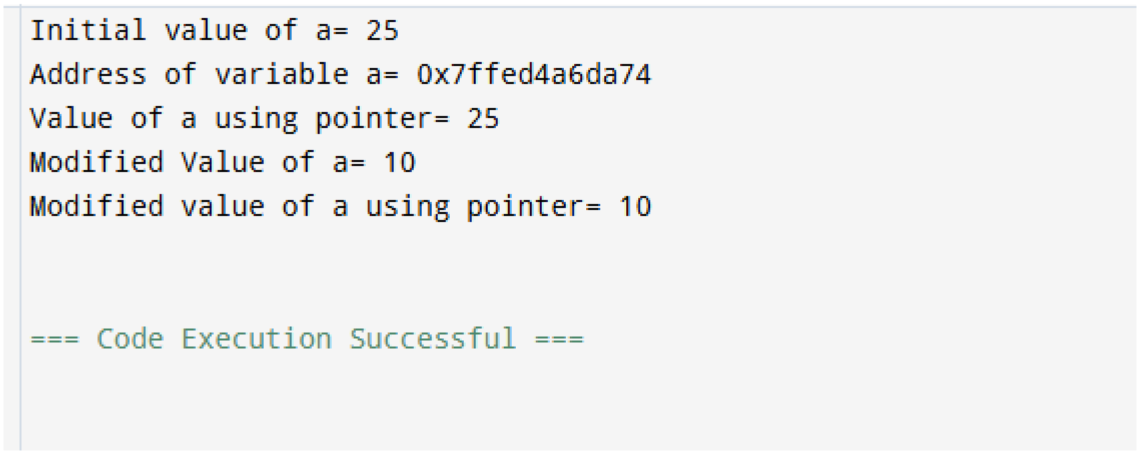
Code:

```
#include <iostream>
using namespace std;
int main(){
	int a = 25;
	int *ptr = &a;
	cout << "Initial value of a= " << a << endl;
	cout << "Address of variable a= " << ptr << endl;
	cout << "Value of a using pointer= "<< *ptr << endl;
	*ptr = 10;
	cout << "Modified Value of a= " << a << endl;
	cout << "Modified value of a using pointer= " << *ptr << endl;
	return 0;
}
```

Explanation of Task:

This program illustrates the use of pointers in C++. It initializes an integer variable **`a`** with value **`25`** first and then a pointer **`ptr`** which holds the address of **`a`.** Through the use of the pointer, the program prints out the original value of **`a`,** its memory location, and the value pointed to. Next, the pointer is used to change the value of `a` by dereferencing it **(`*ptr = 10`).** Lastly, the program outputs the changed value of **`a`,** demonstrating that modifying the value through the pointer also changes the original variable. This illustrates how pointers facilitate direct memory access and manipulation.

Output:



```
Initial value of a= 25
Address of variable a= 0x7ffed4a6da74
Value of a using pointer= 25
Modified Value of a= 10
Modified value of a using pointer= 10


=== Code Execution Successful ===
```

Lab 2

- **Big O Notation (Loops and Arrays)**

Task 1:

Implement a function that finds the maximum value in an array of size n. Determine its time complexity and explain why it is O(n).

Code:

```
#include <iostream>
using namespace std;
int findMax(int arr[], int num){
        int max = arr[0];
        for(int i=1; i<num; i++){
                if(arr[i] > max){
                        max = arr[i]; }
        }
        return max;
}
int main(){
        int arr[] = {3, 4, 1, 6, 0};
        int size = sizeof(arr)/ sizeof(arr[0]);
        cout << "Maximum value in array= " << findMax(arr, size) << endl;
```

```
    return 0;
}
```

Explanation of Task:

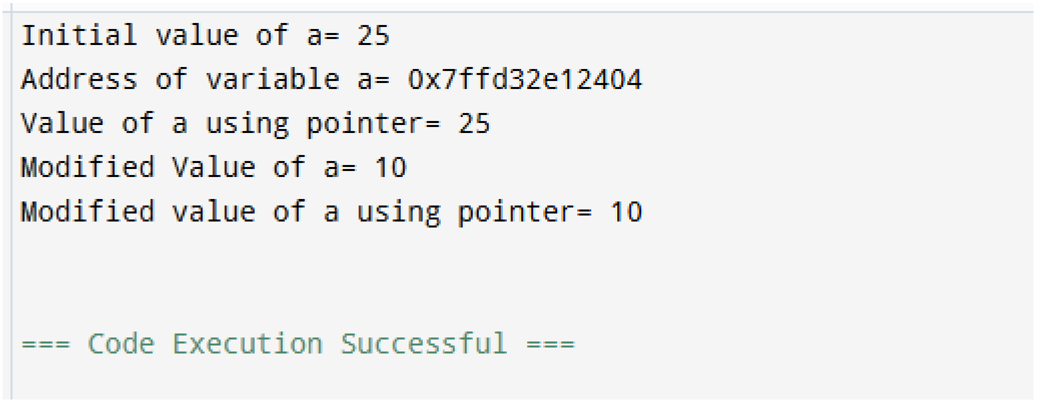
This C++ code illustrates how pointers function by holding and changing the address of a variable. It starts with declaring an integer variable **`a`** and setting it to value **`25`.** A pointer variable **`ptr`** is declared and given the address of `a` using the address-of operator **(`&`).**

The program then outputs the original value of **`a`,** the memory address of **`a`,** and the value accessed via the pointer with the dereference operator **(`*`).** As **`ptr`** contains the address of `a`, dereferencing **`ptr`** **(`*ptr`)** yields the same value as `a`.

Then, the program changes the value of `a` through the pointer by setting `*ptr = 10`. Because `ptr` is pointing to `a`, this alteration directly modifies `a`. The new value is then printed, indicating that `a` is now `10`, and dereferencing `ptr` also yields **`10`.**

This is an example of how pointers give direct access to memory so that variables can be changed without their names. It brings out important concepts like storing memory addresses, dereferencing, and changing values using pointers, which are basic in C++ programming for effective memory management and dynamic data structures.

Output:



```
Initial value of a= 25
Address of variable a= 0x7ffd32e12404
Value of a using pointer= 25
Modified Value of a= 10
Modified value of a using pointer= 10


=== Code Execution Successful ===
```

Lab 3

- **Singly Linked List (Insert at End, Insert at Start)**
  Task 1:
  Implement a singly linked list with functions to insert a node at the start and at the end. Display the list after each insertion.

Code:

```
#include <iostream>
using namespace std;
class Node{
        public:
                int data;
           Node* next;
           Node(int data){
                this->data = data;
                next = NULL;
                }
};
class Linkedlist{
        public:
                Node* head;
                Linkedlist(){
                        head =  NULL;
                }
                void insert_at_start(int data){
                        Node* newNode = new Node(data);
                        newNode->next = head;
                        head = newNode;


                }
                void insert_at_end(int data){
```

```
            Node* newNode = new Node(data);
            if(head == NULL){
                head = newNode;
            }else{
            Node* temp = head;
            while(temp->next != NULL){
                temp = temp->next;
            }
            temp->next = newNode;
        }
        }
        void display(){
            Node* temp = head;
            while(temp!=NULL){
                cout << temp->data << "->";
                temp = temp->next;
            }
            cout << "NULL" << endl;
        }
};
int main(){
    Linkedlist list;
    list.insert_at_start(56);
    cout << "Insertion at start" << endl;
    list.display();
    list.insert_at_end(41);
    cout << "Insertion at end" << endl;
    list.display();
    cout << "Insertion at start" << endl;
    list.insert_at_start(15);
    list.display();
```

```
        cout << "Insertion at start" << endl;

        list.insert_at_end(10);

        list.display();

}
```

Explanation of Task:

This is a C++ code implementing a basic ****singly linked list**** with operations such as inserting at the beginning and end, and printing the list. It has two classes: `Node` for each node of the linked list, and **`Linkedlist`** to deal with the list.

The `Node` class has two fields: `**data**` to hold the value and `**next**`, a pointer to the next node. Its constructor sets these values, making `next` `NULL` by default.

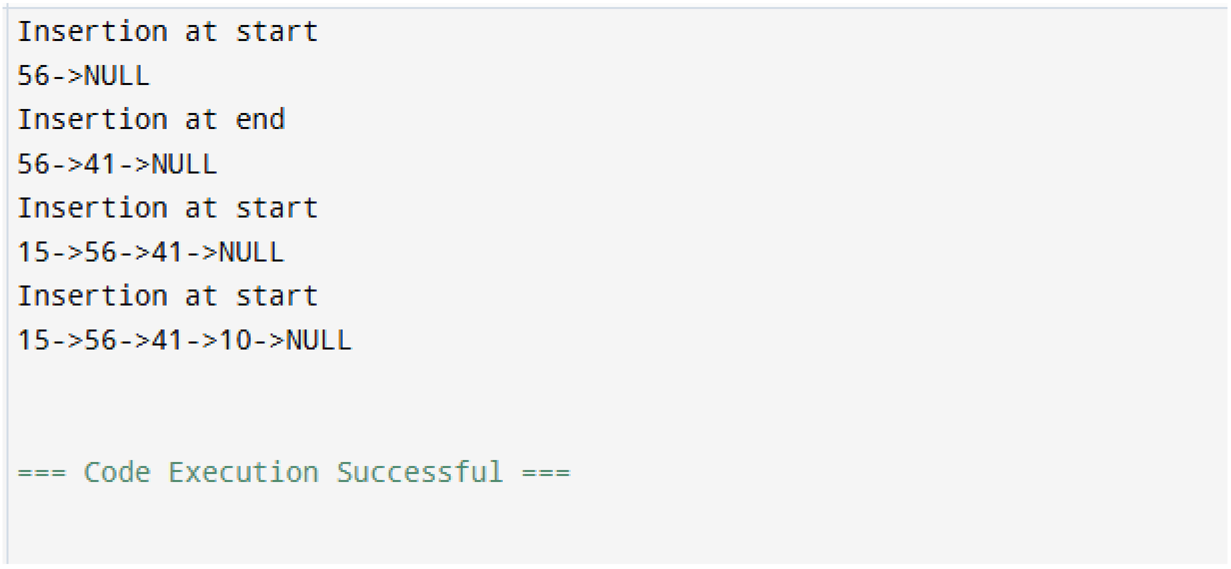
The `**Linkedlist**` class takes care of the linked list operations. It has a pointer `head`, which holds the first node. The constructor sets `head` to `NULL`, meaning the list is empty. The `**insert_at_start**` operation creates a new node, points its `next` to the existing `head`, and changes `head` to the new node, inserting at the front. The `**insert_at_end**` operation finds the last node in the list and appends the new node at the end. If the list is not empty, the new node is set as the `head`.

The `display` function traverses the list, printing the data of each node and an arrow **(`->`)** until it comes to `NULL`, marking the end of the list.

Inside the `main` function, an object of `**Linkedlist**` is made. Insertions are done under different circumstances to show insertion both at the beginning and at the end. Each time an insertion is done, the list is displayed to present the new structure.

This code illustrates basic linked list operations, `new`-based memory allocation, and dynamic node control. It efficiently inserts at ends and offers an organized structure to add more functionalities such as deletion or searching.

Output:



```
Insertion at start
56->NULL
Insertion at end
56->41->NULL
Insertion at start
15->56->41->NULL
Insertion at start
15->56->41->10->NULL


=== Code Execution Successful ===
```

Lab 4

- **Singly Linked List (Insert at Specific Location)**

  Task 1:

  Write a function to insert a node at a specific position in a singly linked list, ensuring valid position handling.

Code:

```
#include <iostream>

using namespace std;

class Node{

        public:

                int data;

          Node* next;

          Node(int data){

                this->data = data;

                next = NULL;

                }
```

```
};
class Linkedlist{
        public:
                Node* head;
                Linkedlist(){
                        head =  NULL;
                }
                void insert_at_start(int data){
                        Node* newNode = new Node(data);
                        newNode->next = head;
                        head = newNode;

                }
                void insert_at_end(int data){
                        Node* newNode = new Node(data);
                        if(head == NULL){
                                head = newNode;
                        }else{
                        Node* temp = head;
                        while(temp->next != NULL){
                                temp = temp->next;
                        }
                        temp->next = newNode;
                }
        }
          void insert_at_pos(int pos, int data){
                if(pos < 1){
                        cout << "Invalid Position! " << endl;
                        }
                        else if(pos == 1){
                                insert_at_start(data);
                        }else{
                                Node* newNode = new Node(data);
                                Node* temp = head;
                                for(int i=1; i<pos-1; i++){
                                    temp = temp->next;
                                    if(temp == NULL){
                                        cout << "Invalid Position! " << endl;
                                        delete newNode;
                                        return;
                                    }
                                }
                                newNode->next = temp->next;
                                temp->next = newNode;
                        }
                }
                void display(){
                        Node* temp = head;
                        while(temp!=NULL){
                                cout << temp->data << "->";
                                temp = temp->next;
                        }
                        cout << "NULL" << endl;
                }
};
int main(){
        Linkedlist list;
        list.insert_at_start(56);
        cout << "Insertion at start" << endl;
        list.display();
        list.insert_at_end(41);
        cout << "Insertion at end" << endl;
```

```
        list.display();

        cout << "Insertion at start" << endl;

        list.insert_at_start(15);

        list.display();

        cout << "Insertion at start" << endl;

        list.insert_at_end(10);

        list.display();

        cout << "Linked list after insertion at any position" << endl;

        list.insert_at_pos(2, 20);

        list.insert_at_pos(4, 12);

        list.insert_at_pos(6, 22);

        list.display();

        return 0;

}
```

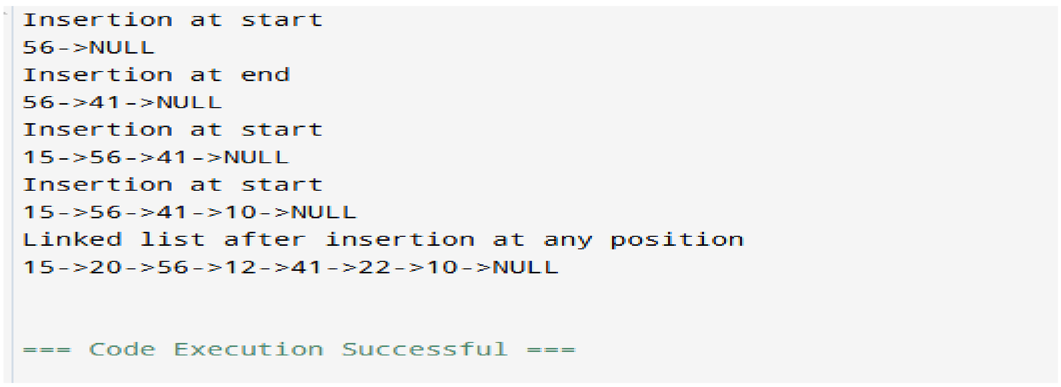
Explanation of Task:

This C++ code realizes a ****singly linked list**** with insertions at the beginning, end, and particular location. The `Node` class defines elements as `data` and `next` pointers. The **`Linkedlist`** class oversees operations by keeping a `head` pointer.

The **`insert_at_start()`** function inserts a node at the start, whereas `insert_at_end()` inserts a node at the end. The **`insert_at_pos()`** function inserts a node at a given position with valid placement. The **`display()`** function prints and traverses the list.

In **`main()`,** there are multiple insertions that show how elements can be dynamically inserted. The program efficiently demonstrates basic linked list operations and pointer-based memory handling in C++.

Output:



```
Insertion at start
56->NULL
Insertion at end
56->41->NULL
Insertion at start
15->56->41->NULL
Insertion at start
15->56->41->10->NULL
Linked list after insertion at any position
15->20->56->12->41->22->10->NULL


=== Code Execution Successful ===
```

Lab 5

- **Singly Linked List (Display Nodes)**

  Task 1:

  Implement functions to display the first node, last node, Nth node, and centre node of a singly linked list.

Code:

```
#include <iostream>

using namespace std;

class Node{

        public:

                int data;

          Node* next;

          Node(int data){

                this->data = data;

                next = NULL;

                }

};


class Linkedlist{

        public:

                Node* head;

                Linkedlist(){

                        head =  NULL;

                }
```

```
            void insert_at_start(int data){
                        Node* newNode = new Node(data);
                        newNode->next = head;
                        head = newNode;


            }
            void insert_at_end(int data){
                        Node* newNode = new Node(data);
                        if(head == NULL){
                                    head = newNode;
                        }else{
                        Node* temp = head;
                        while(temp->next != NULL){
                                    temp = temp->next;
                        }
                        temp->next = newNode;
            }
}
   void insert_at_pos(int pos, int data){
            if(pos < 1){
                        cout << "Invalid Position! " << endl;
                        }
                        else if(pos == 1){
                                    insert_at_start(data);
                        }else{
                                    Node* newNode = new Node(data);
                                    Node* temp = head;
                                    for(int i=1; i<pos-1; i++){
                                                temp = temp->next;
                                                if(temp == NULL){
                                                            cout << "Invalid Position! " << endl;
```

```
                    delete newNode;
                    return;
                }
            }
            newNode->next = temp->next;
            temp->next = newNode;
        }
}
void display_first(){
        if(head != NULL){
            cout << "First Node: " << head->data << endl;
        }
}
void display_last(){
        if(head == NULL){
            cout << "List is empty" << endl;
        }else{
            Node* temp = head;
            while(temp->next != NULL){
                temp = temp->next;
            }
            cout << "Last Node: " << temp->data << endl;
        }
}
void display_nth(int n){
        Node* temp = head;
        for(int i=1; temp!=NULL && i<n; i++){
            temp = temp->next;
        }
        if(temp != NULL){
            cout << "Nth Node: " << temp->data << endl;
```

```
                                }
                        }
                  void display_center() {
            if (!head) {
              cout << "List is empty" << endl;
              return;
            }
         int count = 0;
         Node* temp = head;
           while (temp) {
              count++;
              temp = temp->next;
            }
          temp = head;
           for (int i = 0; i < count / 2; i++) {
             temp = temp->next;
          }
         cout << "Center: " << temp->data << "\n";
                        }
         void display(){
                                Node* temp = head;
                                while(temp!=NULL){
                                        cout << temp->data << "->";
                                        temp = temp->next;
                                }
                                cout << "NULL" << endl;
                        }
};
int main(){
            Linkedlist list;
   list.insert_at_end(13);
```

```
    list.insert_at_start(6);
    list.insert_at_start(5);
    list.insert_at_end(8);
    list.insert_at_pos(4, 10);
    list.insert_at_pos(6, 8);
    list.display();
    list.display_first();
    list.display_last();
    list.display_nth(5);
    list.display_center();
}
```

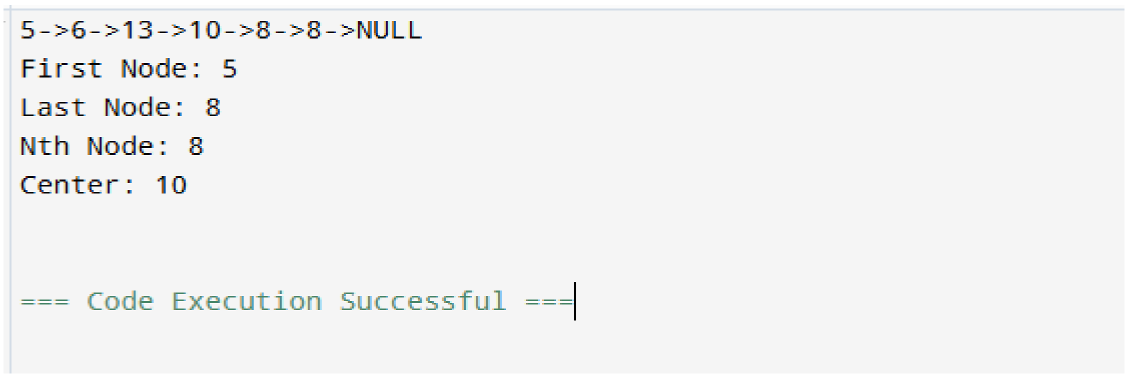
Explanation of Code:

This C++ program implements a singly linked list with additional functions to retrieve specific nodes. The Node class stores data and a pointer to the next node. The Linkedlist class manages the list and provides insertion functions (insert_at_start, insert_at_end, insert_at_pos) to add elements at different positions.
Additional functions:

- **display_first():** Displays the first node.
- **display_last():** Displays the last node.
- **display_nth(n):** Displays the nth node.
- **display_center():** Finds and displays the middle node.
- **display():** Prints the entire list.

In main(), various insertions are performed, followed by calls to display functions to retrieve specific nodes. The program efficiently demonstrates linked list traversal and manipulation.

Output:



```
5->6->13->10->8->8->NULL
First Node: 5
Last Node: 8
Nth Node: 8
Center: 10


=== Code Execution Successful ===
```

Lab 6

- **Singly Linked List (Delete Nodes)**

  <u>Task 1:</u>

  Implement functions to delete the first node, last node, Nth node, and centre node of a singly linked list.

Code:

```
#include <iostream>
using namespace std;
class Node{
        public:
                int data;
          Node* next;
          Node(int data){
                this->data = data;
                next = NULL;
                }
};
class Linkedlist{
        public:
                Node* head;
                Linkedlist(){
                        head =  NULL;
                }
                void insert_at_start(int data){
                        Node* newNode = new Node(data);
                newNode->next = head;
                head = newNode;

        }
        void insert_at_end(int data){
                Node* newNode = new Node(data);
                if(head == NULL){
                        head = newNode;
                }else{
                Node* temp = head;
                while(temp->next != NULL){
                        temp = temp->next;
                }
                temp->next = newNode;
        }
}
   void insert_at_pos(int pos, int data){
        if(pos < 1){
                cout << "Invalid Position! " << endl;
                }
                else if(pos == 1){
                        insert_at_start(data);
                }else{
                        Node* newNode = new Node(data);
                        Node* temp = head;
                        for(int i=1; i<pos-1; i++){
                                temp = temp->next;
                                if(temp == NULL){
                                        cout << "Invalid Position! " << endl;
                                        delete newNode;
                                        return;
                        }
                }
                newNode->next = temp->next;
                temp->next = newNode;
        }
}
void delete_first(){
        if(head == NULL){
                cout << "List is empty" << endl;
        }else{
                Node* temp = head;
                head = head->next;
                delete temp;
        }
}
void delete_last(){
        if(head == NULL){
                cout << "List is empty" << endl;
        }else if(head->next == NULL){
                delete head;
                head = NULL;
        }else{
                Node* temp = head;
                while(temp->next != NULL && temp->next->next != NULL){
                        temp = temp->next;
                }
                delete temp->next;
                temp->next = NULL;
        }
}
void delete_nth(int n){
        if(head == NULL || n<1){
                cout << "Invalid position!" << endl;
        }else if(n == 1){
                delete_first();
        }else{
                Node* temp = head;
                for(int i=1; temp->next&&i<n-1; i++){
                        temp= temp->next;
                }
                if(temp->next == NULL){
                        return;
                }else{
                        Node* todelete = temp->next;
                        temp->next = temp->next->next;
                        delete todelete;
                }
        }
}
void delete_center(){
        if(head==NULL||head->next==NULL){
                delete_first();
        }else{
                int count = 0;
                Node* temp = head;
                while(temp){
                        count++;
                        temp = temp->next;
                }
                int mid=count/2;
                temp=head;
                for(int i=1; i<mid; i++){
                        temp = temp->next;
                    }
                    Node* todelete = temp->next;
                    temp->next = temp->next->next;
                    delete todelete;
                }
            }
            void display(){
                Node* temp = head;
                while(temp!=NULL){
                    cout << temp->data << "->";
                    temp = temp->next;
                }
                cout << "NULL" << endl;
            }
};
int main(){
    Linkedlist list;
    list.insert_at_start(7);
    list.insert_at_start(3);
    list.insert_at_end(9);
    list.insert_at_end(10);
    list.insert_at_pos(4, 11);
    list.insert_at_pos(6, 2);
    cout << "Linked list before deletion" << endl;
    list.display();
    list.delete_first();
    list.delete_last();
    list.delete_center();
    list.delete_nth(5);
    cout << "Linked list after deletion" << endl;
```

```
        list.display();
        return 0;
}
```

Explanation of Task:

This C++ program shows a ****singly linked list**** in which insertion and deletion operations can happen. The `Node` class includes `data` and a next pointer to another node. The **`Linkedlist`** class, on the other hand, takes care of the whole list and has the following:

Insertion Methods:

- **`insert_at_start(data)`:** Insert a node at the beginning.
- **`insert_at_end(data)`:** Insert a node at the end.
- **`insert_at_pos(pos, data)`**: Insert a node at a specific position.

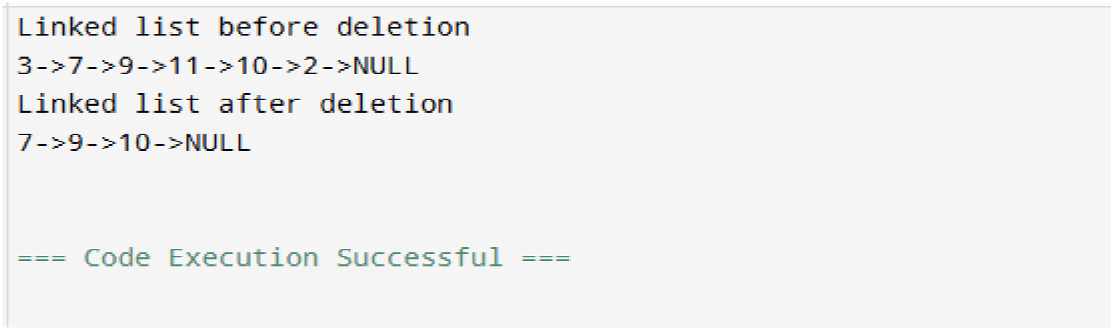
Deletion Methods:

- **`delete_first()`:** Delete the first node.
- **`delete_last()`:** Delete the last node.
- **`delete_nth(n)`:** Delete the node at the nth position.
- **`delete_center()`:** Delete the middle node.

Execution in `main()`:

1. Insert nodes into the linked list at random positions.
2. Show the linked list before performing deletions.
3. Perform various deletions.
4. Print out the new list after deletions.

Output:



```
Linked list before deletion
3->7->9->11->10->2->NULL
Linked list after deletion
7->9->10->NULL


=== Code Execution Successful ===
```

Lab 7

- **Doubly Linked List (Insert & Display Nodes)**

Task 1:

Implement functions to insert node at first, last, Nth location, and centre of a doubly linked list. And display in order and display in reverse order.

Code:

```
#include <iostream>
using namespace std;
class Node;
Node* head = NULL;
class Node {
public:
   int data;
   Node* next;
   Node* prev;

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

   void insert_at_start(int data) {
      Node* newNode = new Node(data);
      if (head == NULL) {
         head = newNode;
      } else {
         newNode->next = head;
         head->prev = newNode;
         head = newNode;
      }
   }
   void insert_at_last(int data) {
      Node* newNode = new Node(data);
      if (head == NULL) {
         head = newNode;
         return;
      }
      Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
   }
   void insert_at_pos(int data, int pos) {
      if (pos < 1) {
         cout << "Invalid position!" << endl;
         return;
      }
      if (pos == 1) {
         insert_at_start(data);
         return;
      }
      Node* temp = head;
      Node* newNode = new Node(data);
      for (int i = 1; i < pos - 1; i++) {
         if (temp == NULL) {
             cout << "Invalid Position!" << endl;
             return;
         }
         temp = temp->next;
      }
      if (temp == NULL) {
            cout << "Invalid Position!" << endl;
            return;
        }
        newNode->next = temp->next;
        if (temp->next != NULL) {
            temp->next->prev = newNode;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
void insert_at_center(int data) {
        Node* temp = head;
        int count = 0;
        while (temp != NULL) {
            count++;
            temp = temp->next;
        }
        int mid = (count / 2) + 1;
        insert_at_pos(data, mid);
    }
    void display() {
        Node* temp = head;
        if (temp == NULL) {
            cout << "Linked list is empty" << endl;
            return;
        }
        while (temp != NULL) {
            cout << temp->data;
            if (temp->next != NULL) cout << "->";
            temp = temp->next;
        }
        cout << endl;
    }

    void display_reverse() {
        if (head == NULL) {
            cout << "Linked list is empty" << endl;
            return;
        }
        Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        while (temp != NULL) {
            cout << temp->data;
            if (temp->prev != NULL) cout << "->";
            temp = temp->prev;
        }
        cout << endl;
    }
};
int main() {
    Node node(0);
    node.insert_at_start(8);
    node.insert_at_start(13);
    node.insert_at_last(14);
    node.insert_at_last(9);
    node.insert_at_pos(10, 4);
    node.insert_at_center(17);
    node.display();
    node.display_reverse();
    return 0;
}
```

Explanation of Task:

This C++ program implements a **Doubly Linked List (DLL),** where each node has pointers to both next and previous nodes, allowing for forward and backward traversal.

Key Functionalities:

-Insertion:

- **`insert_at_start(data)`:** Creates an element at the head of the linked list.
- **`insert_at_last(data)`:** Creates an element at the tail of the linked list.
- **`insert_at_pos(data, pos)`:** Add an element at a specified position in the list.
- **`insert_at_center(data)`:** Insert at the center of the list.

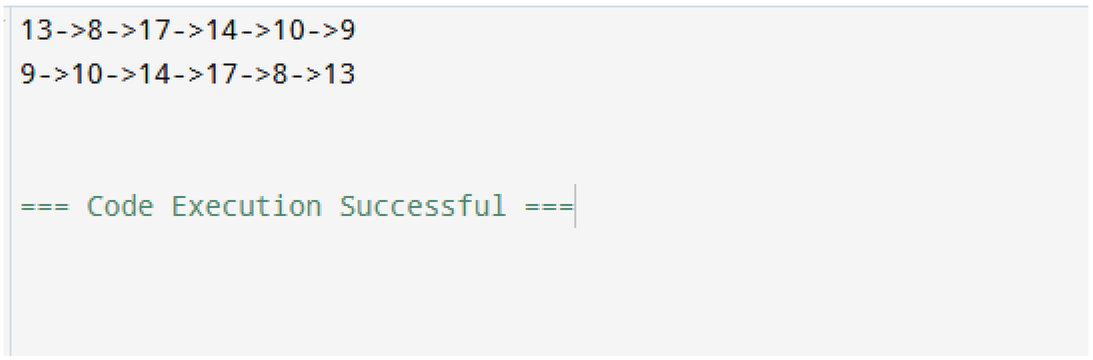
Display:

- **display():** to see the list in forward order
- **display_reversed():** to see the list in reverse order

Execution in `main()`:

Nodes were inserted starting from the beginning, at the end, at any given position, and in the center.
The linked list is then printed out in both forward and backward order.

Output:



```
13->8->17->14->10->9
9->10->14->17->8->13


=== Code Execution Successful ===
```

Lab 8

- **Merge two LinkedLists**

Task 1:

Create 2 Singly LinkedLists and Merge them and display them.

Code:

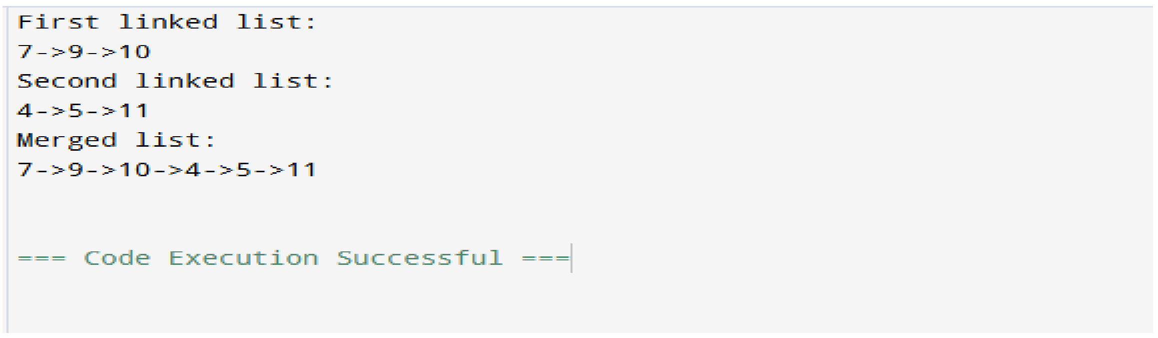
```
#include <iostream>
using namespace std;
class Node;
Node* head1 = NULL;
Node* head2 = NULL;
Node* mergehead = NULL;
class Node {
public:
   int data;
   Node* next;
   Node(int data) {
      this->data = data;
      this->next = NULL;
   }
   void insert(Node*& head, int data) {
      Node* newNode = new Node(data);
      if (head == NULL) {
         head = newNode;
      } else {
         Node* temp = head;
         while (temp->next != NULL) {
            temp = temp->next;
         }
         temp->next = newNode;
      }
   }
   void display(Node* head) {
      if (head == NULL) {
         cout << "List is empty" << endl;
      }
      Node* temp = head;
```

```
        while (temp != NULL) {
            cout << temp->data;
            if (temp->next != NULL) cout << "->";
            temp = temp->next;
        }
        cout << endl;
    }
    Node* merge(Node* h1, Node* h2) {
        if (h1 == NULL){
                        return h2;
                        }
        if (h2 == NULL) {
              return h1;
                       }
                        Node* temp = h1;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = h2;

        return h1;
    }
};
int main() {
    Node obj(0);

    obj.insert(head1, 7);
    obj.insert(head1, 9);
    obj.insert(head1, 10);
    cout << "First linked list: " << endl;
    obj.display(head1);
    obj.insert(head2, 4);

    obj.insert(head2, 5);

    obj.insert(head2, 11);

    cout << "Second linked list: " << endl;

    obj.display(head2);


    mergehead = obj.merge(head1, head2);

    cout << "Merged list: " << endl;

    obj.display(mergehead);


    return 0;

}
```

Explanation of Task:

This C++ program merges two singly linked lists. The `Node` class has methods ascending a linked list from end to end, methods to display the list, and methods for merging two lists.
Insertion means pushing nodes in either **`head1` or `head2`.**
Merge: This connects the last node of `head1` to the first node of `head2`.
Display: It prints the content of the linked lists.
In **`main()`,** creation of two lists, the merging of the lists, and their displaying.

Output:



```
First linked list:
7->9->10
Second linked list:
4->5->11
Merged list:
7->9->10->4->5->11


=== Code Execution Successful ===
```

Task 2:

Create 2 Double LinkedLists and Merge them and display them.

Code:

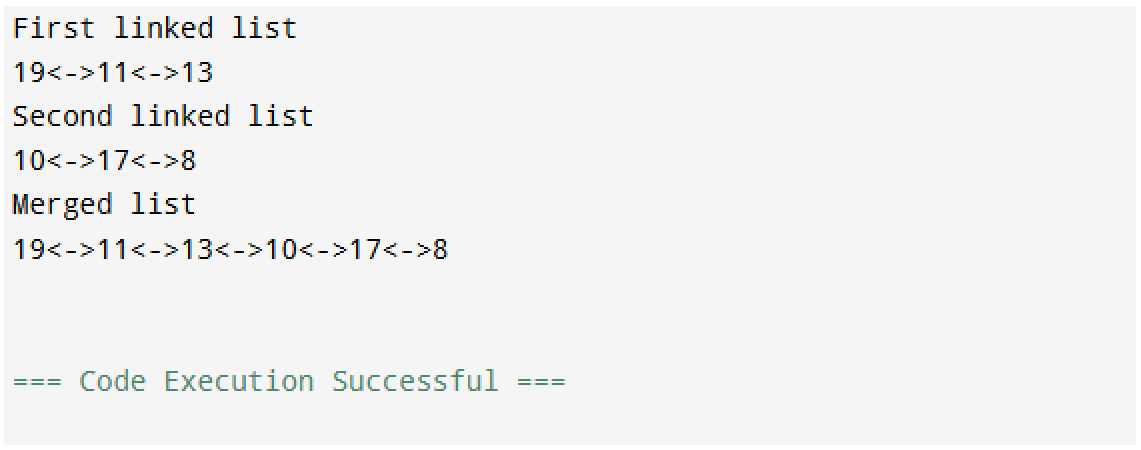
```
#include <iostream>
using namespace std;
class Node;
Node* head1 = NULL;
Node* head2 = NULL;
Node* mergehead = NULL;
class Node {
public:
  int data;
  Node* next;
  Node* prev;
  Node(int data) {
    this->data = data;
    this->next = NULL;
    this->prev = NULL;
  }
  void insert(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == NULL) {
      head = newNode;
    } else {
      Node* temp = head;
      while (temp->next != NULL) {
        temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
    }
  }
  void display(Node* head) {
    if (head == NULL) {
      cout << "Linked list is empty" << endl;
      return;
    }
    Node* temp = head;
    while (temp != NULL) {
      cout << temp->data;
      if (temp->next != NULL) cout << "<->";
      temp = temp->next;
    }
    cout << endl;
  }
  Node* merge(Node* h1, Node* h2) {
    if (h1 == NULL) return h2;
    if (h2 == NULL) return h1;
```

```
        Node* temp = h1;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = h2;
        if (h2 != NULL) {
            h2->prev = temp;
        }
        return h1;
    }
};
int main() {
    Node obj(0);

    obj.insert(head1, 19);
    obj.insert(head1, 11);
    obj.insert(head1, 13);
    cout << "First linked list " << endl;
    obj.display(head1);

    obj.insert(head2, 10);
    obj.insert(head2, 17);
    obj.insert(head2, 8);
    cout << "Second linked list " << endl;
    obj.display(head2);

    mergehead = obj.merge(head1, head2);
    cout << "Merged list" << endl;
    obj.display(mergehead);

    return 0;
}
```

Explanation of Code:

The program is built in C++ where two doubly linked lists are to be merged. Node Class: It exposes the structure for nodes having **`data`,** `next`, and **`prev`** pointers. Insertion: Nodes for the `head1` or `head2` are inserted at the end by traversing and linking the new node. Merge: It merges the `head1` with the `head2` by linking the last node of `head1` to the head node of `head2`. Display: All lists will be shown with the `<->` separator. In main, it is supposed to create two lists, merge them and display them.

Output:



```
First linked list
19<->11<->13
Second linked list
10<->17<->8
Merged list
19<->11<->13<->10<->17<->8


=== Code Execution Successful ===
```

Lab 9

- **Circular Linked List (Insert & Display Nodes)**

  Task 1:

  Implement functions to insert node at first, last, Nth location, and centre of a circular linked list. And display in order and display in reverse order.

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;
```

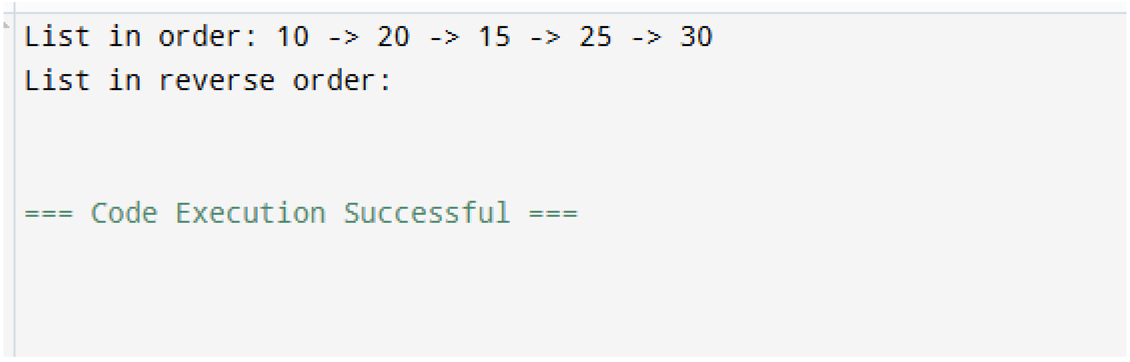
```
}
void insertAtFirst(int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        head->next = head;
    } else {
        Node* temp = head;
        while (temp->next != head) temp = temp->next;
        newNode->next = head;
        temp->next = newNode;
        head = newNode;
    }
    display();
}
void insertAtLast(int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        head->next = head;
    } else {
        Node* temp = head;
        while (temp->next != head) temp = temp->next;
        temp->next = newNode;
        newNode->next = head;
    }
    display();
}
void insertAtNth(int val, int pos) {
    Node* newNode = new Node(val);
    if (pos == 1 || !head) {
        insertAtFirst(val);
        return;
    }
    Node* temp = head;
    for (int i = 1; temp->next != head && i < pos - 1; i++) {
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    display();
}
void insertAtCenter(int val) {
    if (!head || head->next == head) {
        insertAtFirst(val);
        return;
    }
    Node* slow = head, *fast = head;
    while (fast->next != head && fast->next->next != head) {
        slow = slow->next;
        fast = fast->next->next;
    }
    insertAtNth(val, 3);
}
void display() {
    if (!head) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " -> ";
```

```
            temp = temp->next;
        } while (temp != head);
        cout << "(back to head)" << endl;
    }
    void displayReverse() {
        if (!head) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        int arr[100], i = 0;
        do {
            arr[i++] = temp->data;
            temp = temp->next;
        } while (temp != head);

        cout << "Reverse: ";
        for (int j = i - 1; j >= 0; j--) {
            cout << arr[j] << " -> ";
        }
        cout << "(back to head)" << endl;
    }
};
int main() {
    CircularLinkedList clist;
    clist.insertAtFirst(10);
    clist.insertAtLast(30);
    clist.insertAtNth(20, 2);
    clist.insertAtCenter(25);
    cout << "Circular Linked List: ";
    clist.display();
    clist.displayReverse();


    return 0;
}
```

Explanation of Code:

The code constructs a circular-linked list with functions that allow their use for inserting new nodes into it. It provides the ability to insert:
1) At the beginning of the list;
2) At the end;
3) At the Nth position;
4) In the center. Likewise, it displays the list bidirectionally, that is, from the normal side and the other reverse side. The main function calls the above insertions and shows both ways.

Output



```
List in order: 10 -> 20 -> 15 -> 25 -> 30
List in reverse order:


=== Code Execution Successful ===
```