CL2001 Data Structures Lab	Lab 03 Linked list and types of linked list

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

Lab Content

1. Linked List
2. Singly Linked List
3. Doubly Linked List
4. Circular Linked List

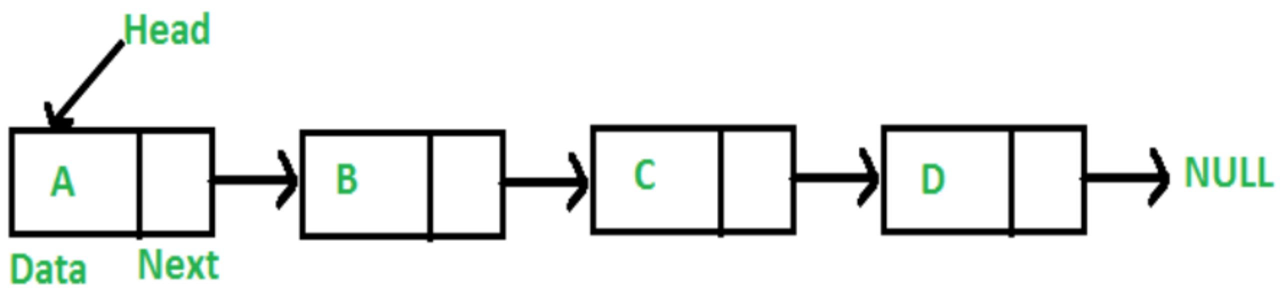
Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

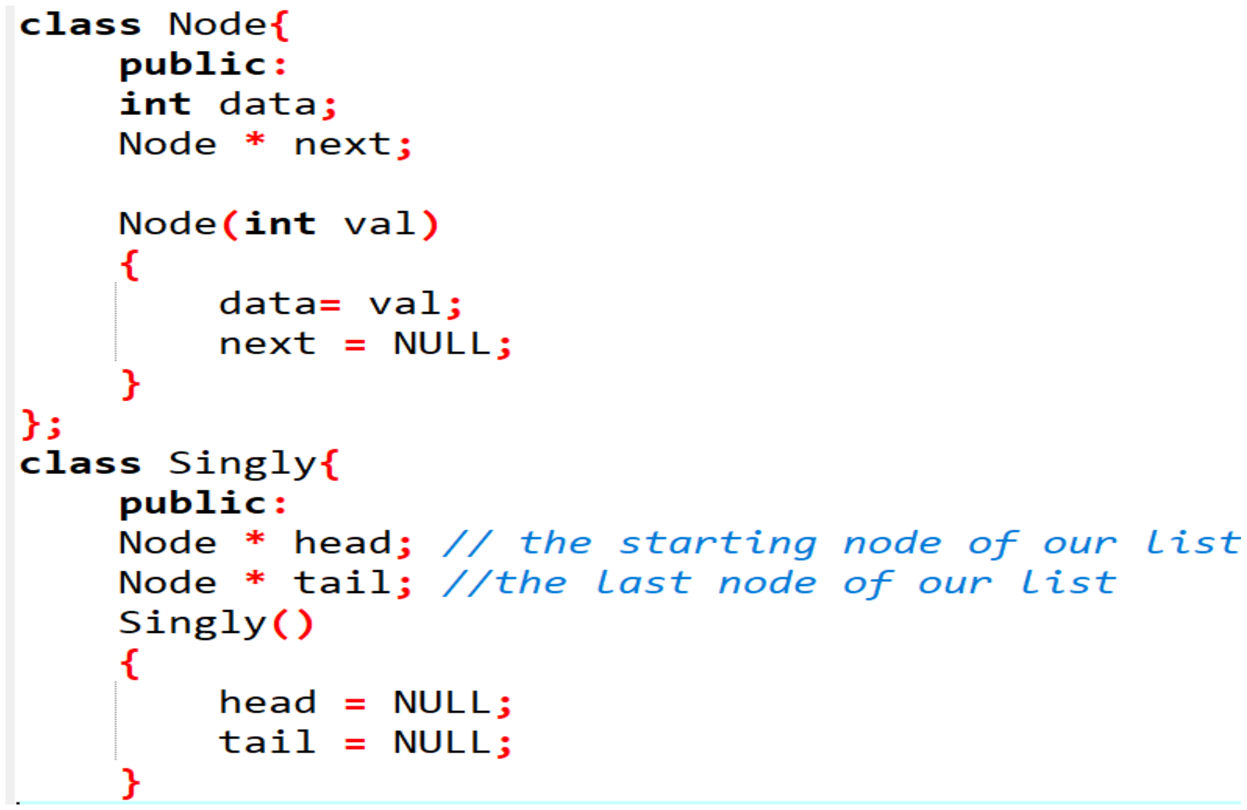
A singly linked list is indeed a type of linked list where each element (node) consists of two parts:

- Data: This part of the node holds the actual value or data that you want to store in the list. It can be any data type, such as integers, characters, or custom data structures.

- Pointer to the Next Node: This part of the node contains a reference or pointer to the next node in the list. This pointer helps maintain the structure of the linked list and allows you to traverse the list in a unidirectional manner, usually from the head (the first node) to the last node. The last node typically points to NULL to indicate the end of the list.



Creating a Linked List: Node and Singly Class

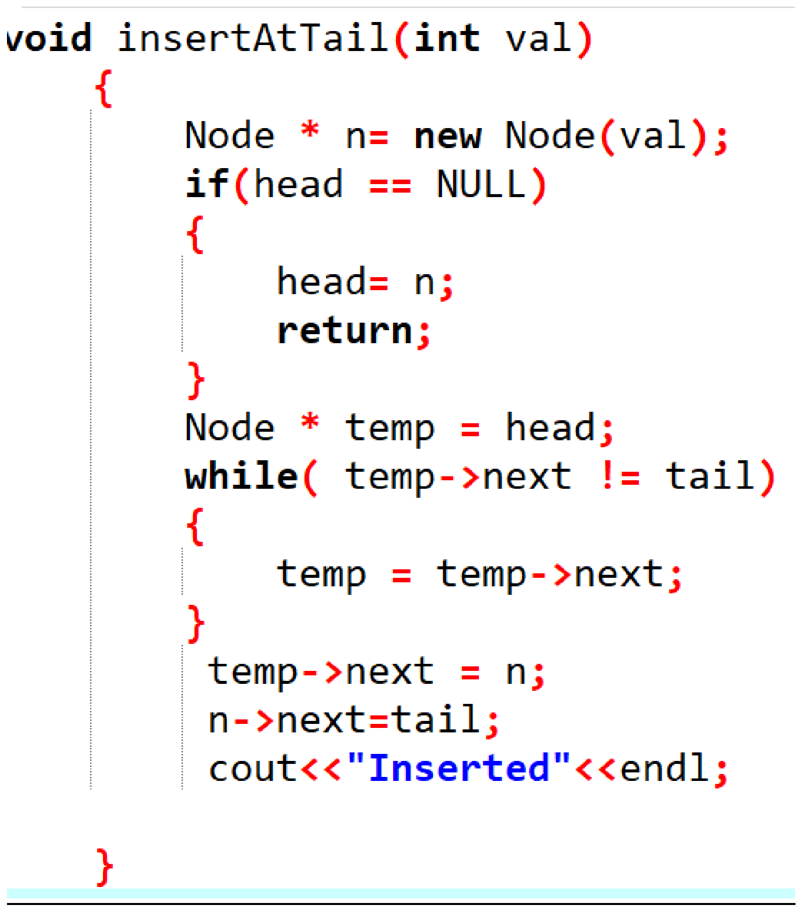


```
class Node{
    public:
    int data;
    Node * next;

    Node(int val)
    {
        data= val;
        next = NULL;
    }
};
class Singly{
    public:
    Node * head; // the starting node of our list
    Node * tail; //the last node of our list
    Singly()
    {
        head = NULL;
        tail = NULL;
    }
```

Adding Node to Linked List(Front,End and anywhere)

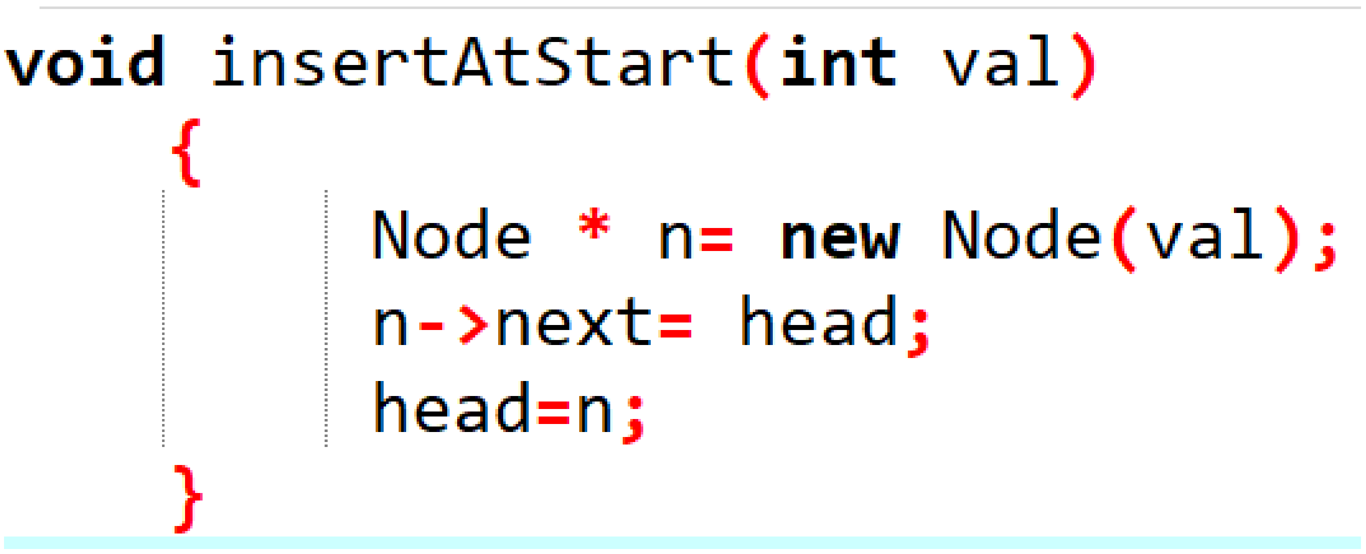
Adding a node to a linked list involves inserting an element, which consists of data and a reference to the next node, either at the front (beginning), the end (tail), or at any specified position within the list.

<u>InsertionAtTail: Append</u>



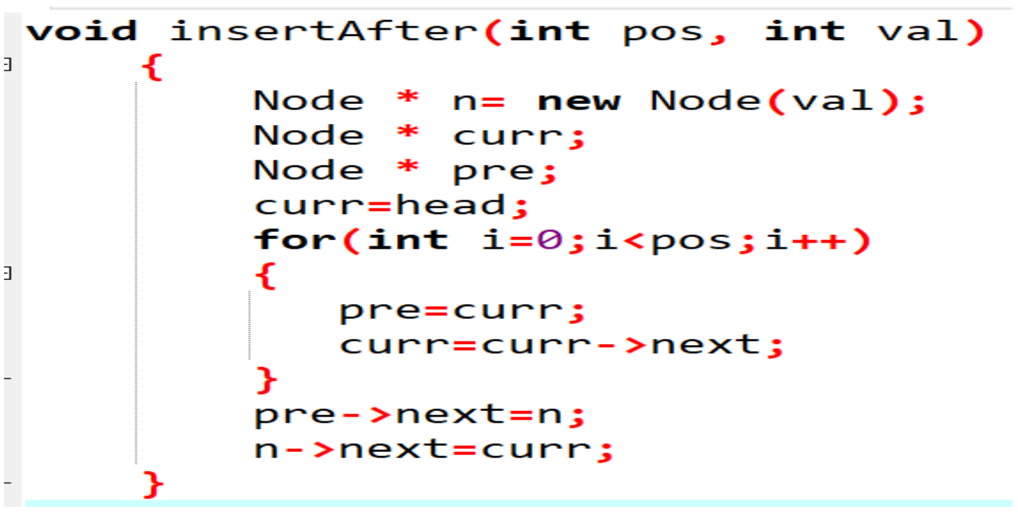
```
void insertAtTail(int val)
    {
        Node * n= new Node(val);
        if(head == NULL)
        {
            head= n;
            return;
        }
        Node * temp = head;
        while( temp->next != tail)
        {
            temp = temp->next;
        }
         temp->next = n;
         n->next=tail;
         cout<<"Inserted"<<endl;

    }
```

InsertionAtStart: Append



```
void insertAtStart(int val)
    {
            Node * n= new Node(val);
            n->next= head;
            head=n;
    }
```

InsertAfter/Insert at any Position



```
void insertAfter(int pos, int val)
    {
        Node * n= new Node(val);
        Node * curr;
        Node * pre;
        curr=head;
        for(int i=0;i<pos;i++)
        {
            pre=curr;
            curr=curr->next;
        }
        pre->next=n;
        n->next=curr;
    }
```

Searching for a Key

```
bool Search(int key)

    {
        Node * temp = head;
        while( temp!= NULL)
        {
            if( temp->data == key)
            {
                cout<<"\nFound"<<endl;
                return true;
            }
            temp = temp->next;

        }
        cout<<"\n Not Found"<<endl;
        return false;
    }
```

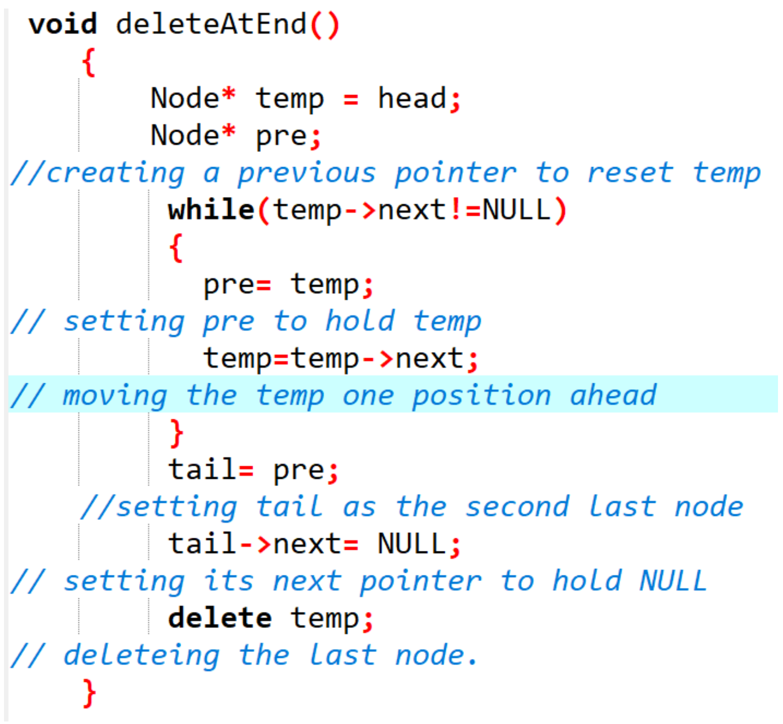
Displaying the Linked List/Traversal

```
void Display()

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

Removal of Nodes: Deletion

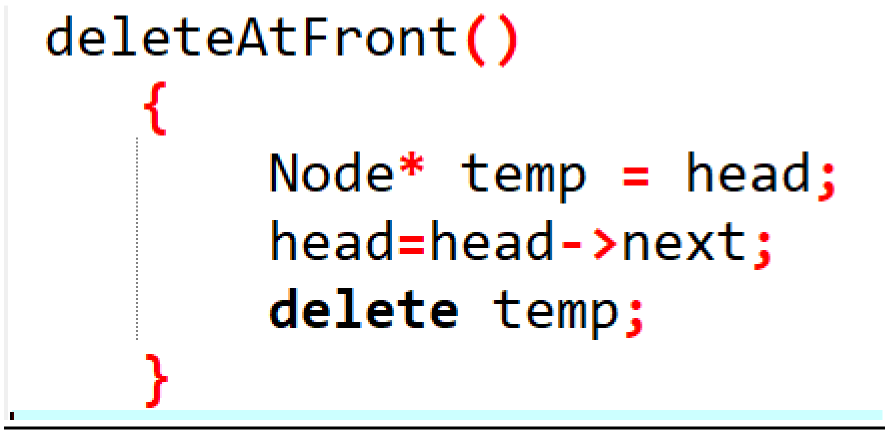
Performing a deletion operation on a linked list involves removing an element, either from the front (beginning), the end (tail), or from any specified position within the list, effectively altering the list's structure

DeletionFromEnd



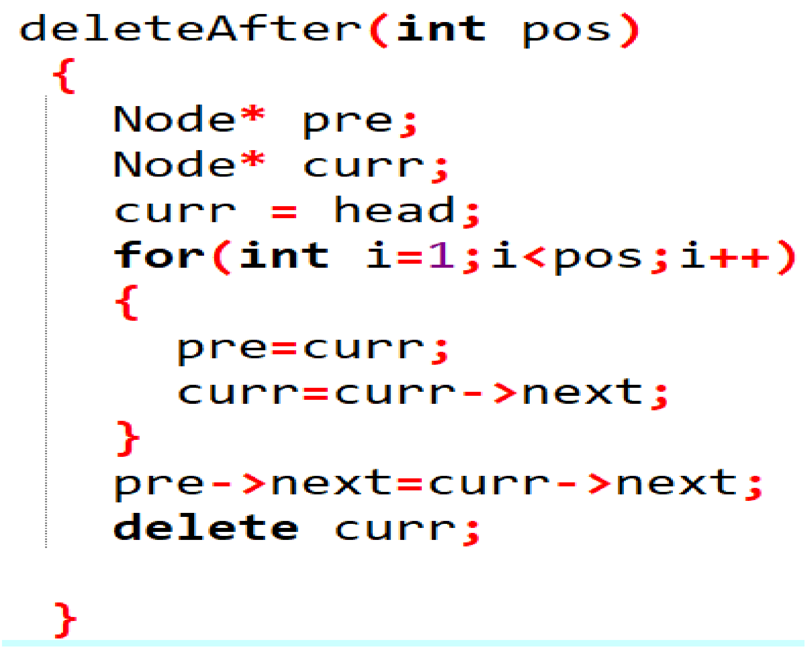
```
void deleteAtEnd()
    {
        Node* temp = head;
        Node* pre;
//creating a previous pointer to reset temp
         while(temp->next!=NULL)
         {
           pre= temp;
// setting pre to hold temp
           temp=temp->next;
// moving the temp one position ahead
         }
         tail= pre;
    //setting tail as the second last node
         tail->next= NULL;
// setting its next pointer to hold NULL
         delete temp;
// deleteing the last node.
    }
```

DeletionFromStart



```
deleteAtFront()
    {
        Node* temp = head;
        head=head->next;
        delete temp;
    }
```

DeleteAfter/ Delete at any position



```
deleteAfter(int pos)
 {
    Node* pre;
    Node* curr;
    curr = head;
    for(int i=1;i<pos;i++)
    {
      pre=curr;
      curr=curr->next;
    }
    pre->next=curr->next;
    delete curr;

 }
```

Doubly Linked List:

A Doubly Linked List is a linear data structure similar to a singly linked list, but with an important difference: each node in a doubly linked list contains two pointers instead of one. It has a pointer to the next node as well as a pointer to the previous node. This bidirectional connectivity allows traversal in both forward and backward directions.

Creation of Doubly Linked List

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

   Node() : data(0), next(NULL), prev(NULL) {}

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

class DoublyLinkedList {
public:
   Node* head;
   Node* tail;
```

```
    DoublyLinkedList() : head(NULL), tail(NULL) {}

    void insertAtEnd(int val) {
        Node* newNode = new Node(val);

        if (tail == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
};
```

Circular Linked List and Helper Functions:

A Circular Linked List is a data structure in which elements, known as nodes, are connected in a circular fashion. Unlike a regular singly linked list, where the last node points to null, in a circular linked list, the last node points back to the first node, creating a loop.
Here's a list of common helper functions that are often implemented in a Circular Linked List:

Append: Add a new node to the end of the circular linked list.
Insert: Add a new node at a specific position in the circular linked list.
Delete: Remove a node with a given value from the circular linked list.
Search: Find a node with a specific value in the circular linked list.
Display: Print the elements of the circular linked list.
Reverse: Reverse the order of nodes in the circular linked list.

Creation of Circular Linked List

```
class Node{
    public:
    int data;
    Node * next;
    Node()
    {
        data =0;
        next=NULL;
    }


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

};
```

```
class Circular{
   public:
   Node * head;
   Node * tail;

   void insertatend( int val)
   {
      Node* n= new Node(val);
      if(head == NULL)
      {
         head=n;
         tail=n;
         tail->next=head;

      }
      tail->next=n;
      tail=tail->next;
      tail->next=head;

   }
   void insertATFront( int val)
   {
      Node * n = new Node(val);
      tail->next= n;
      n->next=head;
      head=n;
```

Lab Tasks:

1. Palindrome Check in Singly Linked List

Write a program to check if a singly linked list is a **palindrome**.

- Input: A singly linked list of integers.
- Output: `true` if the list reads the same forward and backward, `false` otherwise.

---

2. Merge Two Sorted Singly Linked Lists

Implement a function to **merge two sorted singly linked lists** into one sorted list.

- Example:
  - List A: `1 → 3 → 5`
  - List B: `2 → 4 → 6`
  - Output: `1 → 2 → 3 → 4 → 5 → 6`
- Bonus: Solve without creating new nodes (rearrange pointers).

---

3. Reverse in Groups of K

Reverse nodes of a singly linked list in groups of size **k**.

- Example:
  - Input: `1 → 2 → 3 → 4 → 5 → 6 → 7 → 8`, with `k = 3`
  - Output: `3 → 2 → 1 → 6 → 5 → 4 → 8 → 7`

---

4. Josephus Problem using Circular Linked List

Use a circular linked list to solve the **Josephus problem**:

- N people stand in a circle, eliminating every k-th person until only one survives.
- Input: `N = 7, k = 3`
- Output: Position of survivor.
- Hint: Use circular traversal and deletion.

---

5. Convert Between Linked List Types

Implement functions for conversion:

1. Convert a singly linked list into a **doubly linked list**.
2. Convert a singly linked list into a **circular linked list**.

- Demonstrate both conversions with sample input lists.

---

6. Flatten a Multilevel Linked List

Each node has:

- `next` pointer (normal linked list connection).

- `child` pointer (points to another linked list).

Write a function to **flatten** the structure so that all nodes appear in a single-level list.

- Example:
- `1 → 2 → 3`
- `     |`
- `     4 → 5`

  Output: `1 → 2 → 4 → 5 → 3`