Linked List

Charles Aunkan Gomes
Lecturer, Dept. of CSE
United International University
charles@cse.uiu.ac.bd

Linked List

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the **data** and the **address** of the next node. For example:

You might have played the game Treasure Hunt, where each clue includes the information about the next clue. That is how the linked list operates.

Types of Linked List

There are three common types of Linked List.

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

We will be discussing about Singly Linked List now.

Basic Operation

- Traverse: Go through the list starting from first and print.
- Insert: Add a new node in the first, last or interior of the list.
- Delete: Delete a node from the first, last or interior of the list.
- Search: Search a node containing particular value in the linked list.

Singly Linked List

Each node in singly linked list consists:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

```
struct node{
      int value;
      struct node *next;
}
```

You have to start somewhere, so we give the address of the first node a special name called HEAD. Also, the last node in the linked list can be identified because its next portion points to NULL.

Traversal of Singly Linked List

- Set a node “current” to head.
- Check if the current node is null:
  - Access the data of current node
  - Move to the next node
- Repeat the previous step until the current node is null, indicating the end of the linked list.

Traversal of Singly Linked List

```
void printList() {
   Node* current = head;

   while (current != nullptr) {
      cout << current->data << " -> ";
      current = current->link;
   }
   cout << "NULL" << endl;
}
```

Insertion in Singly Linked List

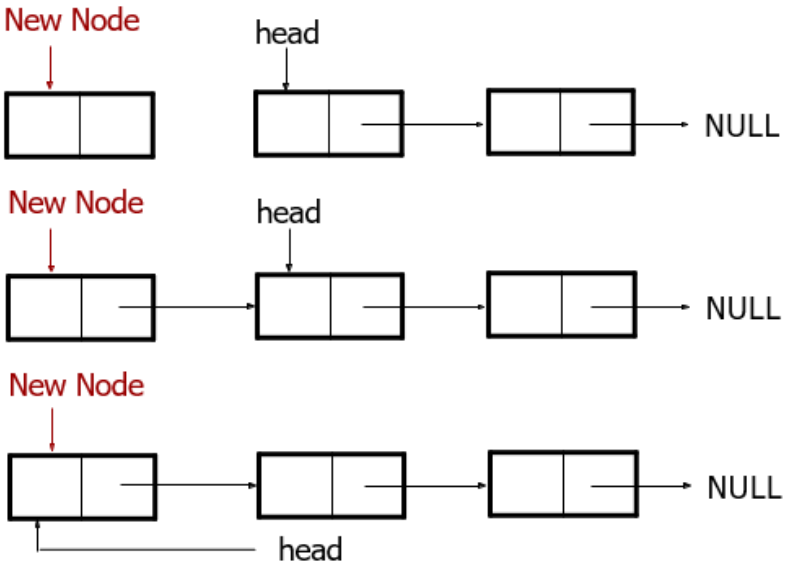
Insertion in a singly linked list involves adding a new node at a specific position. Depending on where you want to insert the new node, there are different cases:

Common Types of Insertion in a Singly Linked List:

1. Insert at the Beginning (Head)
2. Insert at the End (Tail)
3. Insert at a Specific Position

Insert at the Beginning

- Create a New Node
- Set the New Node's link to the Current Head.
- Update the Head to Point to the New Node.



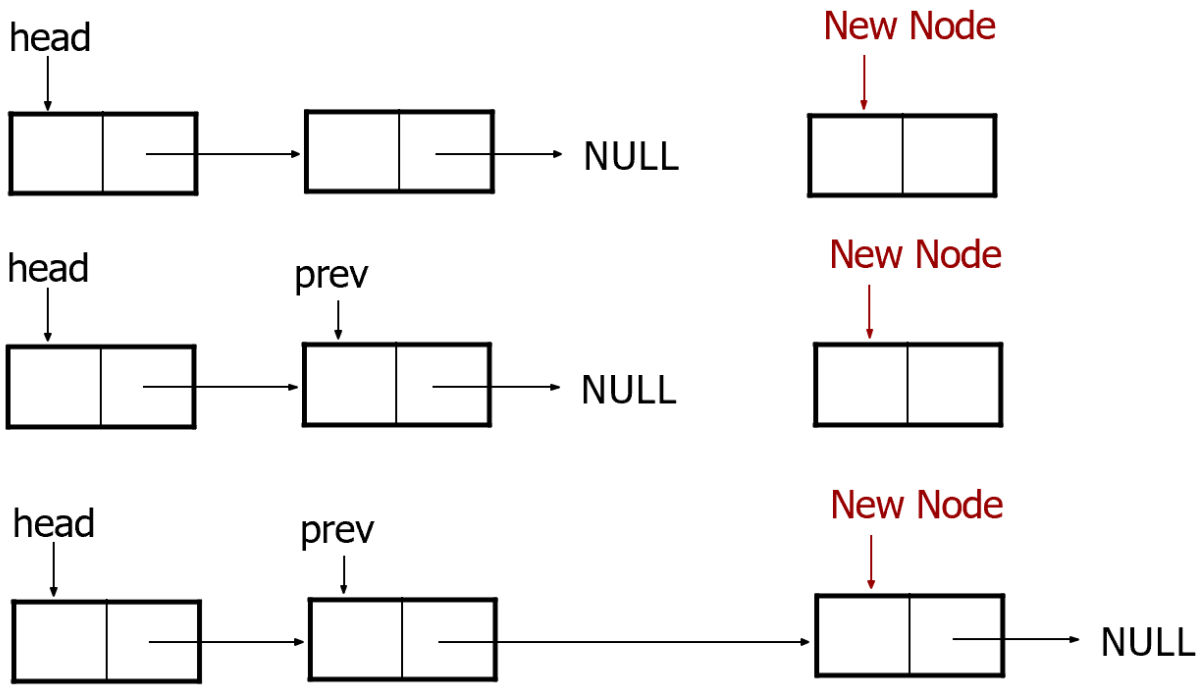
Insert at the Beginning

```
void insertFirst(int data) {
    Node* newNode = new Node(data);
    newNode->link = head;
    head = newNode;
}
```

Insert at the End

- Create a New Node
- Check if the List is Empty:
  - If the list is empty (the head is null), set the head to the new node.
- Traverse to the Last Node.
- Update the Last Node's next Pointer to New Node.

Insert at the End

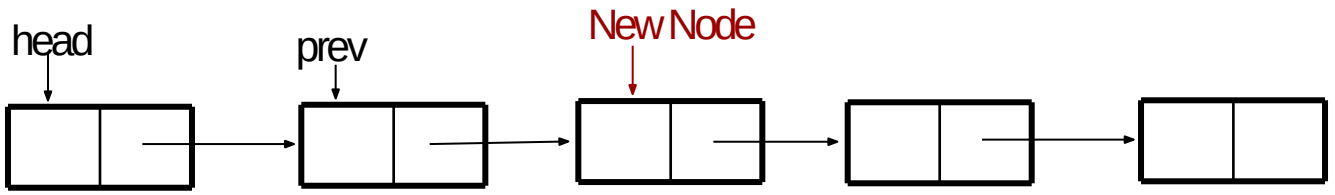


Insert at the End

```
void insertLast(int data) {
   Node* newNode = new Node(data);
   if (head == nullptr) {
      head = newNode;
      return;
   }
   Node* current = head;
   while (current->link != nullptr) {
      current = current->link;
   }
   current->link = newNode;
}
```

Insert at a Specific Position

- Create a new node with the required data.
- Check if the position is valid (>= 0).
- If position is 0, insert at the beginning.
- Traverse to the node before the desired position.
- Insert the new node by updating the appropriate next pointers.



Insert at a Specific Position

```
void insertAt(int data, int position) {
    if (position < 0) {
        cout << "Invalid position." << endl;
        return;
    } else if (position == 0) {
        insertFirst(data);
        return;
    } else {
        Node* newNode = new Node(data);
        Node* current = head;
        int currentPosition = 0;
```

Insert at a Specific Position

```
        while (current != nullptr && currentPosition < position - 1) {
            current = current->link;
            currentPosition++;
        }
        if (current == nullptr) {
            cout << "Position exceeds the length of the list. Node not inserted." << endl;
            delete newNode;
            return;
        } else {
            newNode->link = current->link;
            current->link = newNode;
        }
    }
}
```

Deletion from Linked List

Deletion can be done as follows:

1. Delete the first node (Head node)
2. Delete the last node (Tail node)
3. Delete a Node at a Specific Position

Delete the first node

- Check if the list is empty. If empty, return or show an error.
- Initialize temp pointer to point to the head node of the list.
- Move the head pointer to the second node of the list.
- Remove the node that is pointed by the temp pointer.

Delete the first node

```
void deleteFirst() {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete." << endl;
        return;
    }
    Node* temp = head;
    head = head->link;
    delete temp;
}
```

Delete the last node

- Check if the list is empty (if head == null), then return or show an error.
- Check if the list has only one node (head->next == null), then delete it by setting head = null.
- Traverse to the second-to-last node, then set its next pointer to null, effectively removing the last node.

Delete the last node

```
void deleteLast() {
   if (head == nullptr) {
      cout << "The list is empty. Nothing to delete." << endl;
      return;
   } else if (head->link == nullptr) {
      delete head;
      head = nullptr;
      return;
   }
   Node* current = head;
   while (current->link->link != nullptr) {
      current = current->link;
   }
   delete current->link;
   current->link = nullptr;
}
```

Delete a Node at a Specific Position

1. Initialize pointer current to point to the first node of the list, while the pointer prev has a value of null.
2. Traverse the entire list until the pointer cur points to the node that contains value of x, and prev points to the previous node.
3. Link the node pointed by pointer prev to the node after the cur's node.
4. Remove the node pointed by cur.

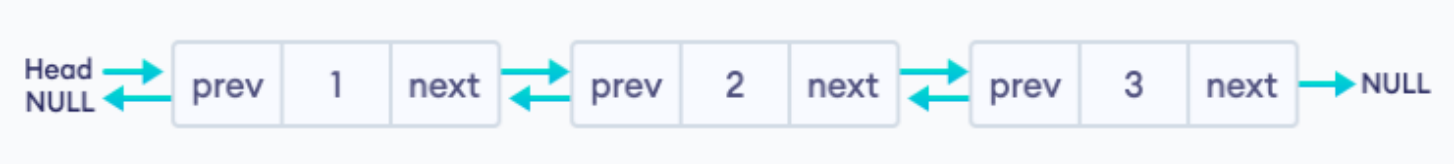
Delete a Node at a Specific Position

```
void deleteItem(int item) {
    Node* current = head;
    Node* prev = nullptr;
    while (current != nullptr) {
        if (current->data == item) {
            if (prev != nullptr) {
                prev->link = current->link;
            } else {
                head = current->link;
            }
            delete current;
            return;
        }
        prev = current;
        current = current->link;
    }
    cout << "Item not found. Node not deleted." << endl;
}
```

Doubly Linked List

Each node in a doubly linked list consists of:

1. A data item
2. Two references (or links):
   - A reference (or pointer) to the next node: This points to the node that comes after the current node, just like in a singly linked list.
   - A reference (or pointer) to the previous node: This points to the node that comes before the current node, which is the key difference between a singly and doubly linked list.



Traversal of Doubly Linked List

Traversal in a doubly linked list allows you to visit all the nodes in either forward direction (from head to tail) or backward direction (from tail to head).

1. Forward Traversal: Same as singly linked list.
2. Backward Traversal:
   - Initialize current to the tail node.
   - Move backward through the list by following the prev pointer from one node to the previous node.
   - When current is nullptr, the traversal is complete.

Traversal of Doubly Linked List

```
void printBackward() {
   Node* current = head;
   while (current != nullptr && current->next != nullptr) {
      current = current->next;
   }
   cout << "Doubly linked list (backward): NULL";
   while (current != nullptr) {
      cout << " <- " << current->data;
      current = current->prev;
   }
   cout << endl;
}
```

Insert First in Doubly Linked List

```
void insertFirst(int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}
```

Insert Last in Doubly Linked List

```
void insertLast(int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* current = head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
}
```

Insert Any in Doubly Linked List

```
void insertAt(int data, int position) {
    Node* newNode = new Node(data);
    if (position == 0) {
        newNode->next = head;
        if (head != nullptr) {
            head->prev = newNode;
        }
        head = newNode;
        return;
    }

    Node* current = head;
    int currentPosition = 0;
    while (current != nullptr && currentPosition < position - 1) {
        current = current->next;
        currentPosition++;
    }
```

Insert Any in Doubly Linked List

```
if (current == nullptr) {
    cout << "Position is out of bounds!" << endl;
  } else {
    newNode->next = current->next;
    newNode->prev = current;

    if (current->next != nullptr) {
      current->next->prev = newNode;
    }
    current->next = newNode;
  }
}
```

Delete First in Doubly Linked List

```
void deleteFirst() {
    if (head == nullptr) {
        cout << "List is empty. Cannot delete the first node." << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    if (head != nullptr) {
        head->prev = nullptr;
    }
    delete temp;
}
```

Delete Last in Doubly Linked List

```
void deleteLast() {
   if (head == nullptr) {
      cout << "List is empty. Cannot delete the last node." << endl;
      return;
   } else if (head->next == nullptr) {
      delete head;
      head = nullptr;
      return;
   } else {
      Node* current = head;
      while (current->next != nullptr) {
         current = current->next;
      }

      current->prev->next = nullptr;
      delete current;
   }
}
```

Delete Any in Doubly Linked List

```
void deleteItem(int item) {
   Node* current = head;
   while (current != nullptr) {
      if (current->data == item) {
         if (current->prev != nullptr) {
            current->prev->next = current->next;
         } else {
            head = current->next;
         }
         if (current->next != nullptr) {
            current->next->prev = current->prev;
         }
         delete current;
         return;
      }
      current = current->next;
   }
   cout << "Item not found. Node not deleted." << endl;
}
```

THANK YOU