Types of Linked List

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

Singly Linked List

Singly linked list is a type of data structure that is made up of nodes that are created using self-referential structures. Each of these nodes contain two parts, namely the data and the reference to the next list node. Only the reference to the first list node is required to access the whole linked list

Node is represented as:

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

Common singly linked list operations

1. Search for a node in the ist

2. Add a node to the list

3. Remove a node from the list

Implementation

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

// Making a node struct containing a data int and a pointer
// to another node
struct Node {
  int data;
  Node *next;
};

class LinkedList
{
    // Head pointer
    Node* head;

  public:
    // default constructor. Initializing head pointer
```

```
LinkedList()
{
  head = NULL;
}

// inserting elements (At start of the list)
void insert(int val)
{
  // make a new node
  Node* new_node = new Node;
  new_node->data = val;
  new_node->next = NULL;

  // If list is empty, make the new node, the head
  if (head == NULL)
    head = new_node;
  // else, make the new_node the head and its next, the previous
  // head
  else
  {
    new_node->next = head;
    head = new_node;
  }
}

// loop over the list. return true if element found
bool search(int val)
{
  Node* temp = head;
  while(temp != NULL)
  {
    if (temp->data == val)
      return true;
    temp = temp->next;
  }
  return false;
}


void remove(int val)
{
  // If the head is to be deleted
  if (head->data == val)
  {
    delete head;
    head = head->next;
    return;
  }

  // If there is only one element in the list
  if (head->next == NULL)
  {
    // If the head is to be deleted. Assign null to the head
    if (head->data == val)
    {
      delete head;
      head = NULL;
      return;
    }
    // else print, value not found
```

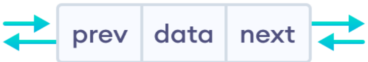
```
        cout << "Value not found!" << endl;
        return;
      }

      // Else loop over the list and search for the node to delete
      Node* temp = head;
      while(temp->next!= NULL)
      {
        // When node is found, delete the node and modify the pointers
        if (temp->next->data == val)
        {
          Node* temp_ptr = temp->next->next;
          delete temp->next;
          temp->next = temp_ptr;
          return;
        }
        temp = temp->next;
      }

      // Else, the value was neve in the list
      cout << "Value not found" << endl;
    }

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

int main() {

  LinkedList l;
  // inserting elements
  l.insert(6);
  l.insert(9);
  l.insert(1);
  l.insert(3);
  l.insert(7);
  cout << "Current Linked List: ";
  l.display();

  cout << "Deleting 1: ";
  l.remove(1);
  l.display();

  cout << "Deleting 13: ";
  l.remove(13);

  cout << "Searching for 7: ";
  cout << l.search(7) << endl;

  cout << "Searching for 13: ";
  cout << l.search(13) << endl;
}
```

Doubly Linked List

We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

A doubly linked list is a type of linked list in which each node consists of 3 components:

- `*prev` - address of the previous node
- `data` - data item
- `*next` - address of next node



Representation of Doubly Linked List

```
struct node {
   int data;
   struct node *next;
   struct node *prev;
}
```

Insertion on a Doubly Linked List

1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the End

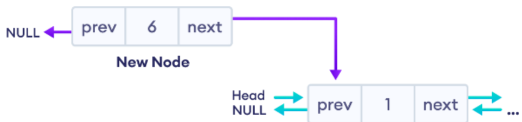
1. Insertion at the Beginning

1. Create a new node

- allocate memory for `newNode`
- assign the data to `newNode`.

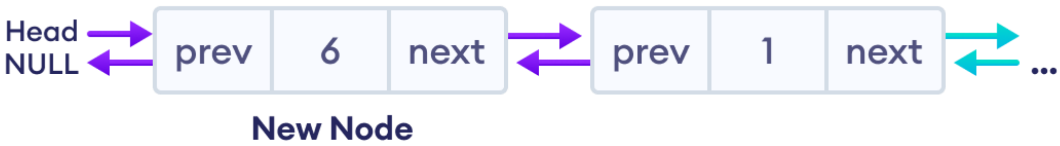
2. Set prev and next pointers of new node

- point `next` of `newNode` to the first node of the doubly linked list
- point `prev` to `null`



3. Make new node as head node

- Point `prev` of the first node to `newNode` (now the previous `head` is the second node)
- Point `head` to `newNode`



Insertion at Beginning

```
// insert node at the front
void insertFront(struct Node** head, int data) {

   // allocate memory for newNode
   struct Node* newNode = new Node;

   // assign data to newNode
   newNode->data = data;

   // point next of newNode to the first node of the doubly linked list
   newNode->next = (*head);

   // point prev to NULL
   newNode->prev = NULL;

   // point previous of the first node (now first node is the second node) to newNode
   if ((*head) != NULL)
      (*head)->prev = newNode;

   // head points to newNode
   (*head) = newNode;
}
```

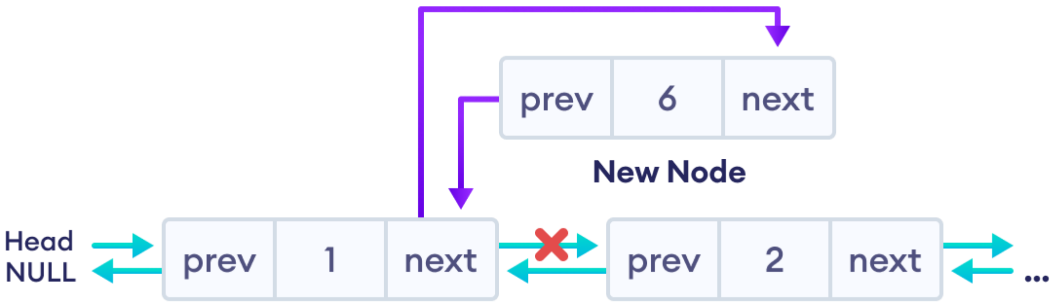
2. Insertion in between two nodes

1. Create a new node

- allocate memory for `newNode`
- assign the data to `newNode`.

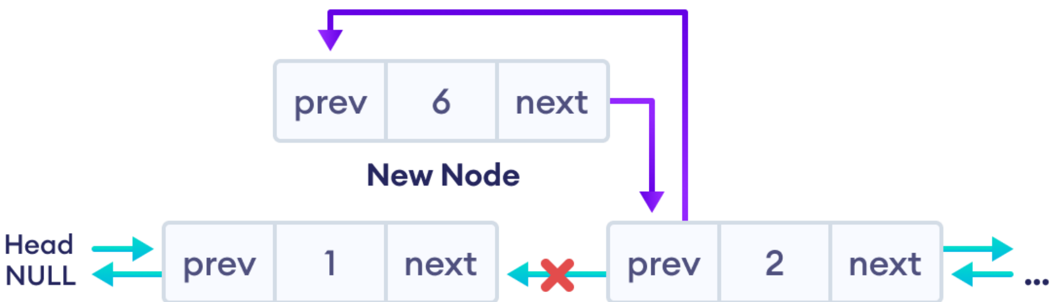
2. Set the next pointer of new node and previous node

- assign the value of `next` from previous node to the `next` of `newNode`
- assign the address of `newNode` to the `next` of previous node



3. Set the prev pointer of new node and the next node

- assign the value of `prev` of next node to the `prev` of `newNode`
- assign the address of `newNode` to the `prev` of next node



The final doubly linked list is after this insertion is:

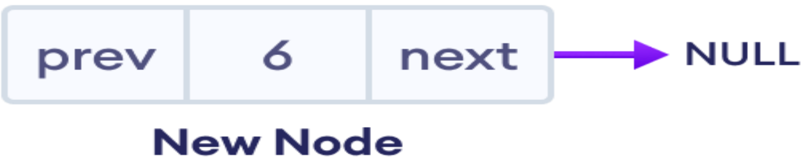

```
// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {

  // check if previous node is NULL
  if (prev_node == NULL) {
    cout << "previous node cannot be NULL";
    return;
  }

  // allocate memory for newNode
  struct Node* newNode = new Node;

  // assign data to newNode
  newNode->data = data;

  // set next of newNode to next of prev node
  newNode->next = prev_node->next;

  // set next of prev node to newNode
  prev_node->next = newNode;

  // set prev of newNode to the previous node
  newNode->prev = prev_node;

  // set prev of newNode's next to newNode
  if (newNode->next != NULL)
```

```
	newNode->next->prev = newNode;
}
```

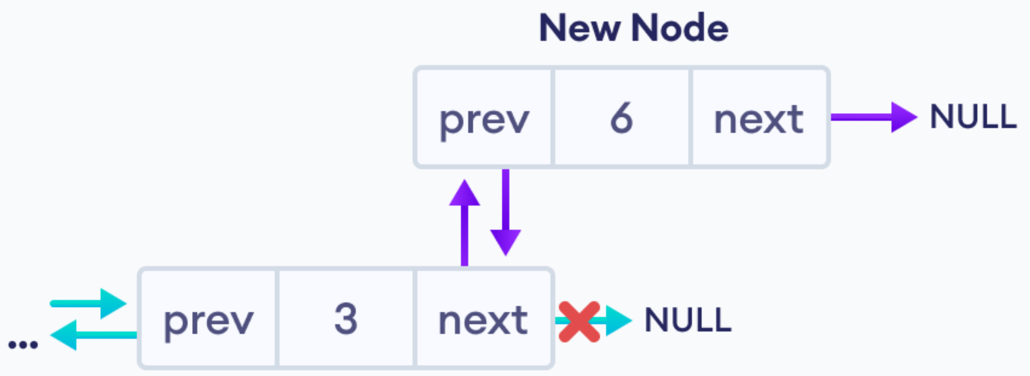
3. Insertion at the End

Let's add a node with value 6 at the end of the doubly linked list.

1. **Create a new node**



2. **Set prev and next pointers of new node and the previous node**



Code Insert at the End

```
// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
  // allocate memory for node
  struct Node* newNode = new Node;

  // assign data to newNode
  newNode->data = data;

  // assign NULL to next of newNode
  newNode->next = NULL;

  // store the head node temporarily (for later use)
  struct Node* temp = *head;

  // if the linked list is empty, make the newNode as head node
  if (*head == NULL) {
    newNode->prev = NULL;
    *head = newNode;
    return;
  }
  // if the linked list is not empty, traverse to the end of the linked list
  while (temp->next != NULL)
    temp = temp->next;
   // now, the last node of the linked list is temp
  // point the next of the last node (temp) to newNode.
  temp->next = newNode;
  // assign prev of newNode to temp
  newNode->prev = temp;
}
```

Doubly Linked List Complexity

Doubly Linked List Complexity	Time Complexity	Space Complexity
Insertion Operation	O(1) or O(n)	O(1)
Deletion Operation	O(1)	O(1)

Doubly Linked List Applications

1. Redo and undo functionality in software.
2. Forward and backward navigation in browsers.
3. For navigation systems where forward and backward navigation is required.

Singly Linked List Vs Doubly Linked List

Singly Linked List	**Doubly Linked List**
Each node consists of a data value and a pointer to the next node.	Each node consists of a data value, a pointer to the next node, and a pointer to the previous node.
Traversal can occur in one way only (forward direction).	Traversal can occur in both ways.
It requires less space.	It requires more space because of an extra pointer.
It can be implemented on the stack.	It has multiple usages. It can be implemented on the stack, heap, and binary tree.