Linked list Data Structure

A linked list is a linear data structure that includes a series of connected nodes (not in consecutive memory locations like array). Here, each node stores the data and the address of the next node. For example,

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.

Linked lists can be of multiple types: singly, doubly, and circular linked list. Today we will focus on the singly linked list.

Creating a node class:

This C++ code defines a class called Node, which is commonly used to create nodes in a singly linked list.

The class consists of two data members.

- The variable val will store the data (or value) associated with the node.
- Node *next;: Declares a pointer member variable named next within the Node class. This pointer is used to point to the next node in the linked list. Initially, it is set to NULL to indicate that there is no next node.

Node(int val): Defines a constructor for the Node class. This constructor takes an integer val as a parameter and initializes the val member variable with the value passed as an argument. It also sets the next pointer to NULL.

So, this Node class is used to create individual nodes for a singly linked list, where each node contains an integer value (val) and a pointer to the next node (next).

Sample Code:

```
class Node
{
        public:
        int val;
        Node *next;
                Node(int val)
                {
                        this->val=val;
                        next=NULL;
                }
};
```

Creating the LinkedList class:

class LinkedList: Defines a class named LinkedList to represent a singly linked list. It has also two data members.

- int length:This variable will store the current length or number of nodes in the linked list.
- Node* head: Declares a pointer member variable named head within the LinkedList class. This pointer is used to point to the first node in the linked list. Initially, it is set to NULL to indicate that the list is empty.

LinkedList(): Defines a constructor for the LinkedList class. This constructor is responsible for initializing the linked list.

length=0;: Initializes the length member variable to zero, indicating that the list is initially empty.

head=NULL;: Initializes the head pointer to NULL, indicating an empty list with no nodes.

So, this LinkedList class provides a basic structure for managing a singly linked list. It keeps track of the list's length and maintains a pointer to the first node (head) in the list.

Sample Code:

```
class LinkedList
{
        int length;
        Node* head;
        public:
                LinkedList()
                {
                        length=0;
                        head=NULL;
                }
};
```

Check if the list is empty:

In case the list is empty the head node always points to NULL, which is initialized in the constructor of the LinkedList class.

```
bool isEmpty()
{
        if(head==NULL)
        {
                return true;
        }
        else
        {
                return false;
        }
}
```

Insertion in the List:

The insert function in the LinkedList class is designed to insert a new node with a given value at a specified position within the linked list. It begins by checking if the position provided (pos) is valid, which means it must be within the range of 1 to (length+1), where "length" represents the current number of nodes in the list. If the position is invalid, it prints an error message and exits. If the position is 1, indicating insertion at the beginning of the list, it creates a new node (temp) with the given value and makes it the new head of the list by updating the head pointer and setting its "next" pointer to the previous head. For positions other than 1, it iterates through the list to find the node immediately before the desired position (pos-1) and inserts the new node (temp) between this node and the next node, by connecting through pointers. Finally, it increments the length variable to reflect the addition of the new node.

Sample Code:

```
void insert(int pos, int val)
{
	if(pos<1 || pos> (length+1))
	{
		cout<<"Invalid position"<<endl;
		return;
	}
	else if(pos==1) // insert at top
	{
		Node *temp=new Node(1);
		temp->next=head;
		head=temp;
	}
	else // insert at any other location
	{
		Node *curr=head;
		for(int i=1; i<(pos-1); i++)
		{
			curr=curr->next;
		}
		Node *temp=new Node(val);
		temp->next=curr->next;
		curr->next=temp;
		}
	length++;
	}
```

Printing the List

The printList function in the LinkedList class is responsible for traversing the linked list and printing the values of each node.

This function starts by creating a temporary pointer curr and initializing it to point to the head of the linked list. It then enters a while loop, which continues as long as it reached the end of the list, which we can identify when the curr becomes NULL. Inside the loop, it prints the value (val) of the current node pointed to by curr. After printing the value, it updates curr to point to the next node in the list (curr->next) to continue traversing the list. This process repeats until curr becomes NULL, meaning it has reached the end of the list.

Sample Code:

```
void printList()
{
        Node *curr=head;
        while(curr!=NULL)
        {
                cout<<curr->val<<" ";
                curr=curr->next;
        }
        cout<<endl;
}
```

Searching in the List:

This function starts by initializing a boolean variable found to false, indicating that the element hasn't been found yet. It then creates a temporary pointer curr and initializes it to point to the head of the linked list. Next, it enters a while loop, which continues as long as curr is not NULL. Inside the loop, it checks if the value (val) of the current node pointed to by curr is equal to the value being searched for. If a match is found (i.e., curr->val equals val), it prints "Element found" to indicate that the desired value was found in the list and sets found to true to record the finding. The loop continues, allowing the function to search the

entire list. After exiting the loop, it checks the found variable; if it's still false, it means the element was not found in any of the nodes, so it prints "Element not found" to indicate that the value was not present in the list.

Sample Code:

```
void search(int val)
{
	bool found=false;
	Node *curr=head;
	while(curr!=NULL)
	{
		if(curr->val==val)
		{
			cout<<"Element found"<<endl;
			found=true
		}
		curr=curr->next;
	}
	if(found==false)
	{
		cout<<"Element not found"<<endl;
	}
}
```

Main Code:

```
Int main()
{
        LinkedList l;
        l.insert(1, 1);
        l.printList();
        l.insert(2, 3);
        l.printList();
        l.insert(3, 4);
        l.printList();
        l.insert(3, 5);
        l.printList();
        l.deleteNode(2);
        l.printList();
}
```

Sample Output:

```
1
1 3
1 3 4
1 3 5 4
1 5 4
```

Note that the delete, and update methods should be implemented on your own.