Chapter 5
LINKED LIST

Data Structure & Algorithm

Compiled by :

Ankit Bhattarai
Email: ankit.bca@kathford.edu.np

Syllabus

Unit	Contents	Hours	Remarks
1.	Introduction to Data Structure	2	
2.	The Stack	3	
3.	Queue	3	
4.	List	2	
5.	Linked Lists	5	
6.	Recursion	4	
7.	Trees	5	
8.	Sorting	5	
9.	Searching	5	
10.	Graphs	5	
11.	Algorithms	5	

Credit : 3

Outlines

- Introduction
- Linked List as an ADT
- Dynamic Implementation
- Insertion & Deletion of Node To and From a List
- Insertion & Deletion of Node After and Before Nodes
- Linked Stacks & Queues
- Doubly Linked Lists & its Advantages

Introduction to Linked List

- Linked List is a collection of nodes where each node consists of at least two parts:

  i. **Information field or info field** : It contains the actual element to be stored.

  ii. **Linked or address field**: It contains one or two links that points to the next node or previous node in the list.

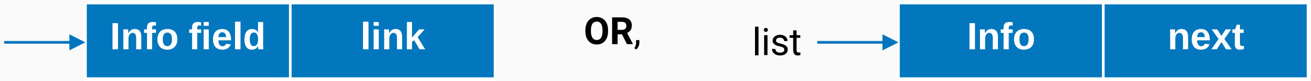
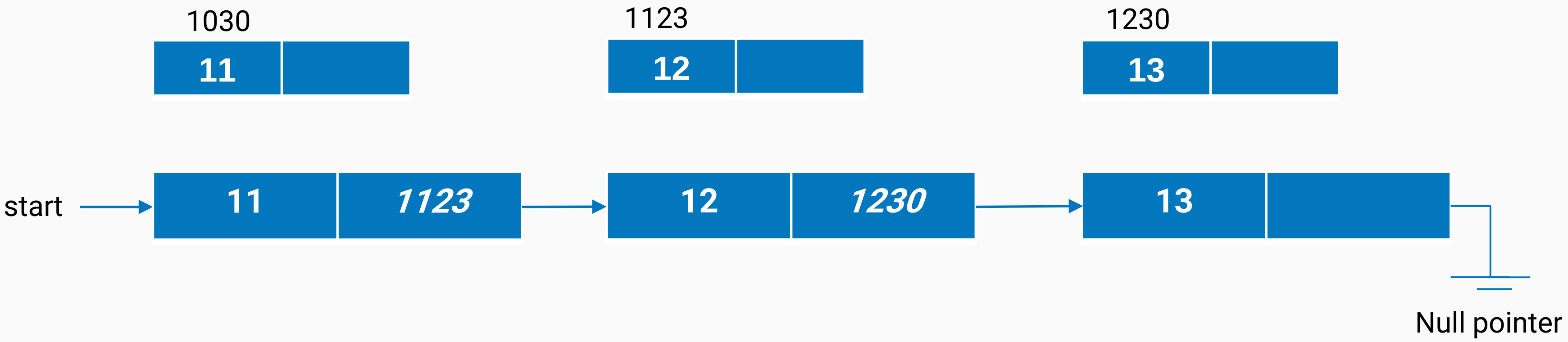


Figure. A node

✓ The memory for nodes of the linked list can be allocated dynamically whenever required.

Introduction to Linked List



Note:

- The first element of the list is known as **head** & the last element is known as **tail**.

Introduction: Basic Operations in Linked List

1. **Insertion:** This operation is creates a new node & inserts it into the list. A new node may be inserted at

✓ At the beginning of the list (insert Head)

✓ At the end of the List (insert Tail)

✓ At the specified position (insert After, insert Before, insert AtNthPos)

✓ If the list is empty, the node to be inserted will be the first node.

2. **Deletion** : This operation is deletes a node from the list. A node may be deleted at

✓ At the beginning of the list (delete Head)

✓ At the end of the List (delete Tail)

✓ At the specified position (delete After, delete Before, delete AtNthPos)

Introduction: Basic Operations in Linked List

3. Traversing

- The process is visiting all the nodes of the linked list from one end to the other is called traversing of the list.

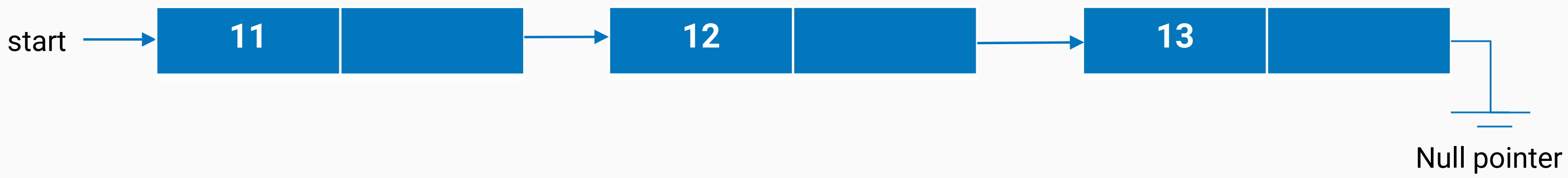
4. Searching:

- This operation checks if a particular data item is present in the list or not.

5. Display:

- This operation is used to print the content of the list.

Null Pointer/ External Pointer/ Empty List



- The linked field of last node contains NULL, rather than a valid address. It is called a **null pointer** & indicates the end of list.

- We use **external pointer or start** which points to the first node of the list. It enables us to access the entire linked list. This pointer must be updated whenever the first node of the list changes.

Null Pointer/ External Pointer/ Empty List

- If no nodes are present in the list, then it is called an **empty list** or a NULL list. The value of external pointer will be assigned NULL for an empty list in our case

*** start = NULL;**

Question : **Explain Linked List as an ADT.**

Values:

✓ A linked list contains a sequence of zero or more nodes.

✓ Each node consists of a general type T and the link field.

✓ It consists of a start pointer and null pointer which indicates start and end of the list respectively.

Linked List as an ADT

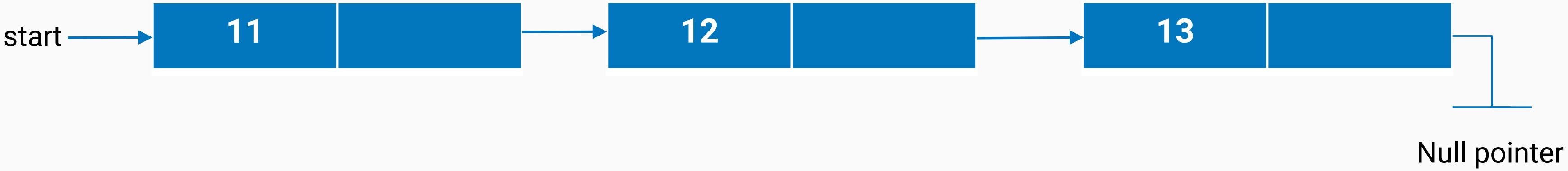
Operations:

i. makeEmpty(L): Make the linked list L an empty list in which start pointer points to the null pointer.
ii. InsertHead(): Insert a new data item at the beginning of the list.
iii. InsertTail(): Insert a new data item at the end of the list.
iv. Insertatposition(): Insert a new data item at the specified position of the list.
v. DeleteHead(): Delete the data item at the beginning of the list.
vi. Deletetail(): Delete the data item at the end of the list.
vii. Deleteatposition(): Delete the data item at the specified position of the list.

Types of Linked List

i. **Singly Linked List or Single Linked List (SLL)**

- It is a linear data structure.
- Only one address field is present in a node. This field stores the address of the following node i.e. it points to the next node.
- The address field of the last node consists of NULL pointer.



Types of Linked List: SLL

Node representation of singly linked list

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

Here, the type of information stored in ‘int’. Here, next is a pointer to the parent structure type.

Types of Linked List: DLL

ii. **Doubly Linked List (DLL)**

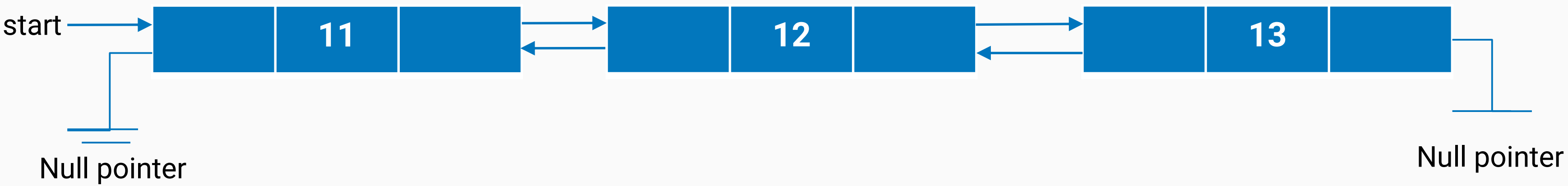
- Two address field are present in a node.
- One of the link stores the address of the following node while the next address field stores the address of the previous node.
- The next link of last node & previous link of first node are NULL

Previous node Address	**Info**	**Next node Address**

Types of Linked List: DLL

Node representation of doubly linked list

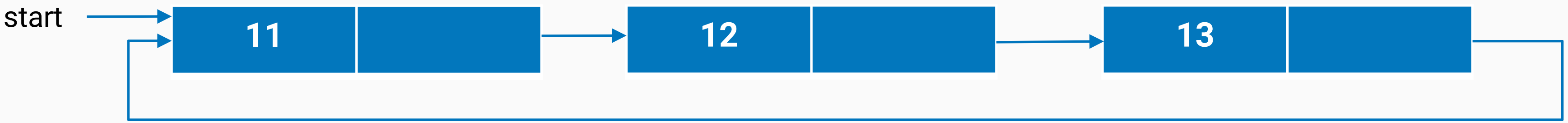
```
struct node {
          int info;
          struct node * next, * previous;
          } ;
```



Types of Linked List: CLL

iii. **Circular Linked List (CLL)**

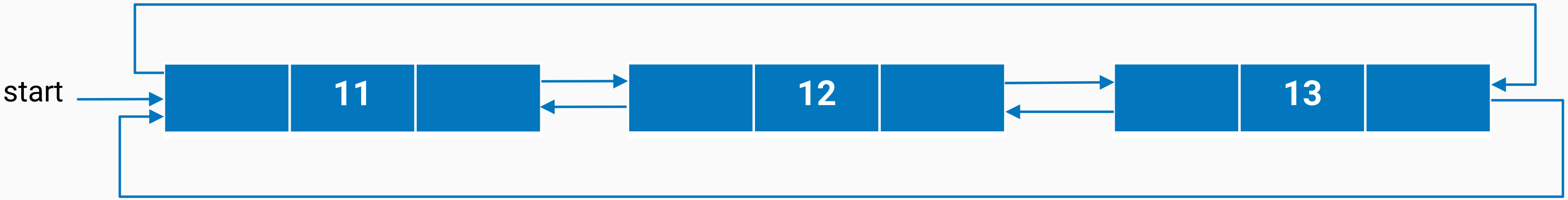
- Only one address field is present in a node. This field stores the address of the following node i.e. it points to the next node.
- The address of the last node consists of the address of the first node. So, the last node will point to the first node.



Types of Linked List: DCLL

iv. **Doubly circular Linked List (DCLL)**

- Two address fields are present in a node. One of the link stores the address of the following node while the next address field stores the address of previous node.
- The next link of last node points to the first node while the previous link of first node points to last node.



Linked list Overview

- The next address field of the last node in the list contains a special value, known as null, which is not a valid address. This null pointer is used to signal the end of a list.
- The list with no nodes on it is called the empty list or the null list.
- The nodes in a linked list are not stored contiguously in the computer’s memory
- We don’t have to shift any element while inserting/deleting in the list
- Memory for each node can be allocated dynamically whenever the need arises
- The no of items in the list is only limited by the computer’s memory

Dynamic Implementation of Linked List

- In Dynamic implementation the size of the structure in not fixed and can be modified during the operations performed on it.
- Linked List is used as a dynamic list.
- A linked list is a data structure is a dynamic structure. It is an ideal technique to store data when the user is not aware of the number of elements to be stored.
- **Linked list** is a series of data items with each item containing a pointer giving a location of next item in the list.

Dynamic Memory Allocation in C

- Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure is changed during the runtime.
- Four library functions that are use in C for dynamic memory allocation are: malloc(), calloc(), free() & realloc(). It is in <stdlib.h> file.
- **malloc():**
  - The name "malloc" stands for memory allocation.
  - **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size.
  - It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default ***garbage value***.

Dynamic Memory Allocation in C

- **malloc()**
  - *Syntax* : **ptr = (cast-type*) malloc(byte-size)**
  - Example: ptr = (int*) malloc(100 * sizeof(int));

  The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

- **free()**
  - free method in C is used to dynamically de-allocate the memory
  - It helps to reduce wastage of memory by freeing it.
  - *Syntax:* free(ptr);

Linked List Creation

Linked list can be created as:

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

ptr = (struct node *) malloc (sizeof(struct node));

- The struct declaration merely describes the format of the nodes and does not allocate storage.
- Here, the type of the information to be stored is ‘int’.
- next is a pointer to the parent structure type. This type of structure is called self-referential structures.
- Storage space for a node is created only when the function malloc() is called in the statement

Algorithm for insertion in singly linked list (SLL)

To insert a node in a list,

- ➢ allocate required memory to a node
- ➢ assign the data in its information field
- ➢ adjust the pointers to link the node at the required place

1. Algorithm for inserting a node at the beginning(head) of the SLL
2. Algorithm for inserting a node at the end(tail) of the SLL
3. Algorithm for inserting a node at the specified position of the SLL

1. Algorithm for inserting a node at the beginning(head) of the SLL

i. Start

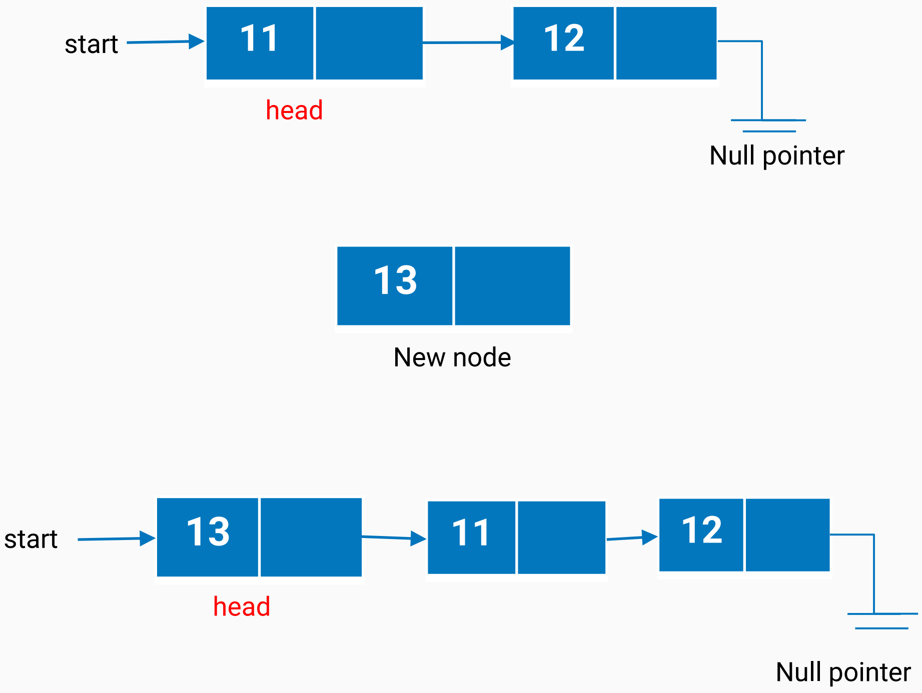
ii. Allocate memory for the new node

iii. Assign value

iv. If the list is currently empty, assign NULL to the link field of the new node. Otherwise, make the link field of the new node to point to the starting node of the linked list.

v. Then, set the external pointer (which is pointing to the starting node) to point to the new node.

vi. Stop



C code for inserting a node at the beginning(head) of the SLL

```
void begin_insert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }
}
```

2. Algorithm for inserting a node at the end(tail) of the SLL

i. Start

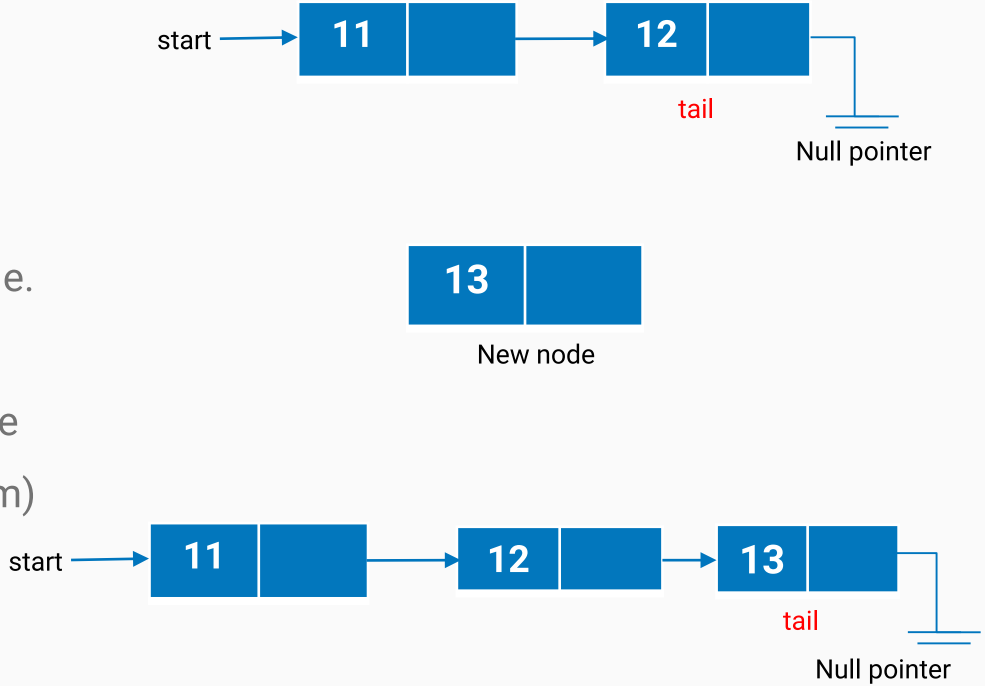
ii. Allocate memory for the new node

iii. Assign value

iv. Assign NULL to the link field of the new node.

v. Traverse the list until last node is reached. Then, insert the new node after the last node by storing the address of the new node (item) to link field of the last node.

vi. Stop



Code for inserting a node at the end(tail) of the SLL

```
void insert_at_the_end()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nLinked List Overflow");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
```

```
if(head == NULL)
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
```

3. Algorithm for inserting a node at the specified position of the SLL

i. Start

ii. Allocate memory for the new node

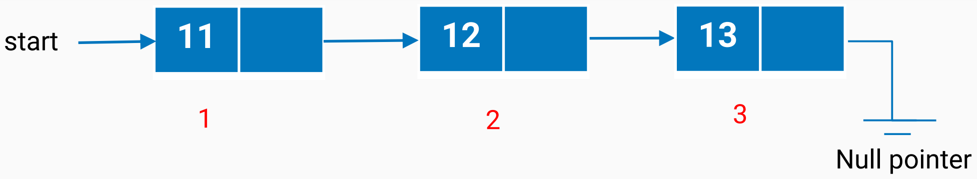
iii. Assign value

iv. Traverse the list till the given node ***(loc -1).***

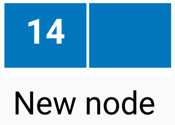
v. Assign next of the new node to point link field of **following node**. (e.g. data=12 is the following node)

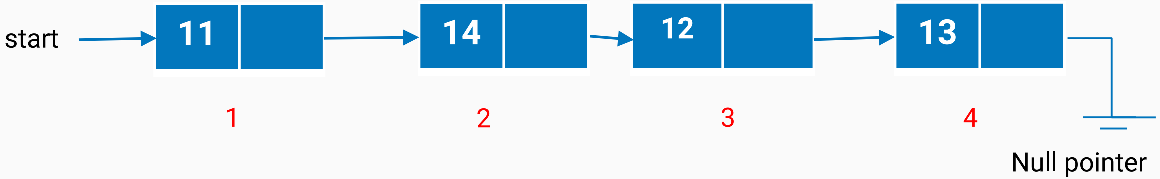
vi. Assign link field of given node to new node.

vii. Stop



Insert new data = 14 at position 2 of list.





Code for inserting a node after the specified position of the SLL

```
void position_insert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nLinked List OVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
for(i=1;i<loc-1;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\nCan't insert the value\n");
                return;
            }

        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode is inserted");
    }
}
```

Algorithm for delete in singly linked list (SLL)

To delete a node from a list

- ➢ Find out the node to be deleted.
- ➢ Adjust the pointers
- ➢ Free the node to be deleted

1. Algorithm for deleting a node at the beginning(head) of the SLL
2. Algorithm for deleting a node at the end(tail) of the SLL
3. Algorithm for deleting a node at the specified position of the SLL

1. Algorithm for deleting a node at the beginning(head) of the SLL

i. Start

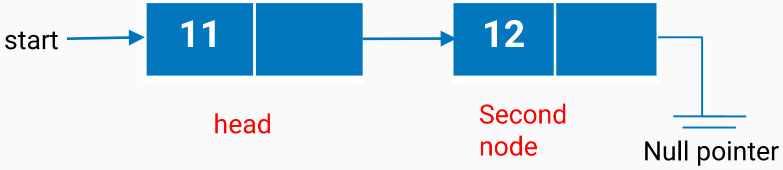
ii. If the list is empty, it is underflow.

Else,

move the start pointer to the second node

iii. Free the first node

iv. Stop



start

11

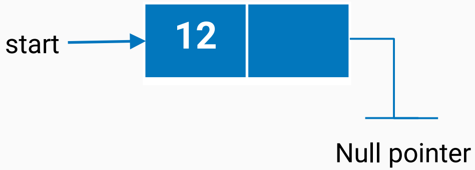
12

head

Second node

Null pointer

After deletion at the head, the start pointer points to the second node of the Linked List



start

12

Null pointer

Code for deleting a node at the beginning(head) of the SLL

```
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nLinked List is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ...\n");
    }
}
```

2. Algorithm for deleting a node at the end(tail) of the SLL

i. Start

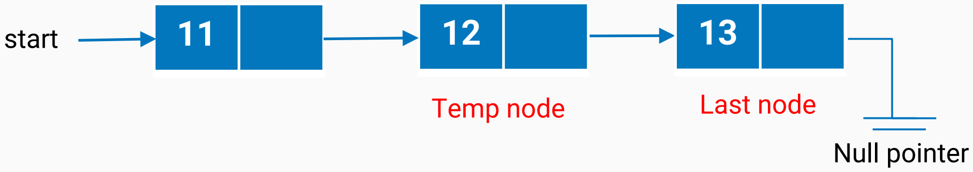
ii. If the list is empty, it is underflow

iii. Else, traverse until the last node is encountered.

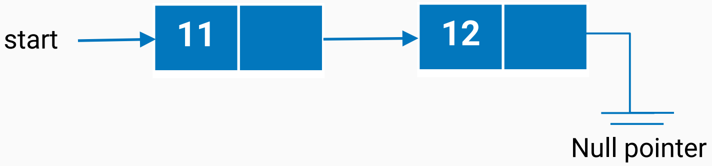
iv. Also keep track of the node, which is before the current node.

v. make the second last node (i.e. temp) to point to NULL and free the last node(i.e. current)

vi. Stop



After deletion of last node, the linked list will be



Code for deleting a node at the end(tail) of the SLL

```
void delete_at_the_end()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }

    else
    {
```

```
 ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ...\n");
    }
}
```

3. Algorithm for deleting a node at the specified position of the SLL

i. Start

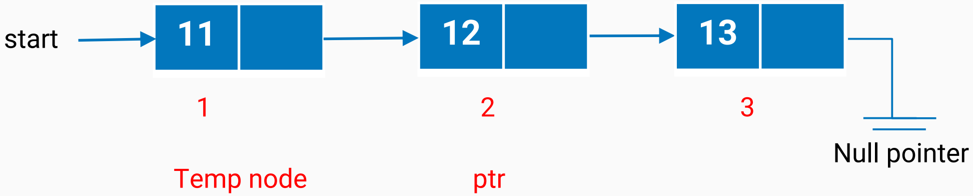
ii. If the list is empty, it is underflow

iii. Else, traverse until the specified node is encountered (ptr)

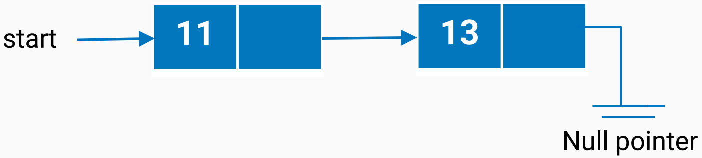
iv. Also keep track of the node, which is before the specified node(i.e temp node)

v. make the next of temp to point to next of ptr and free the node pointed by ptr.

vi. Stop



Delete data at position 2 of linked list.



Code for deleting a node at the specified position of the SLL

```
void position_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which   you
want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
 for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
```

```
  ptr1 ->next = ptr ->next;
   free(ptr);
   printf("\nDeleted node %d ",loc+1);
}
```

THANK YOU

Any Queries ?

ankit.bca@kathford.edu.np