DATA STRUCTURES

UNIT-2

Dr. Nagarathna
N
PROFESSOR
B.M.S. COLLEGE OF ENGINEERING

UNIT 2 : LINKED LIST

Dr. Nagarathna
N
PROFESSOR
B.M.S. COLLEGE OF ENGINEERING

UNIT 2

- **Linear list:** Singly linked list implementation, insertion, deletion and searching operations on linear list, circularly linked lists- insertion, deletion and searching operations for circularly linked lists, doubly linked list implementation, insertion, deletion and searching operations, maintaining directory of names, Manipulation of polynomials (addition), representing sparse matrices.

Linked List

- Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.
- Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.
- Linked Lists are used to create trees and graphs.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Applications of Linked Lists in the Real World:

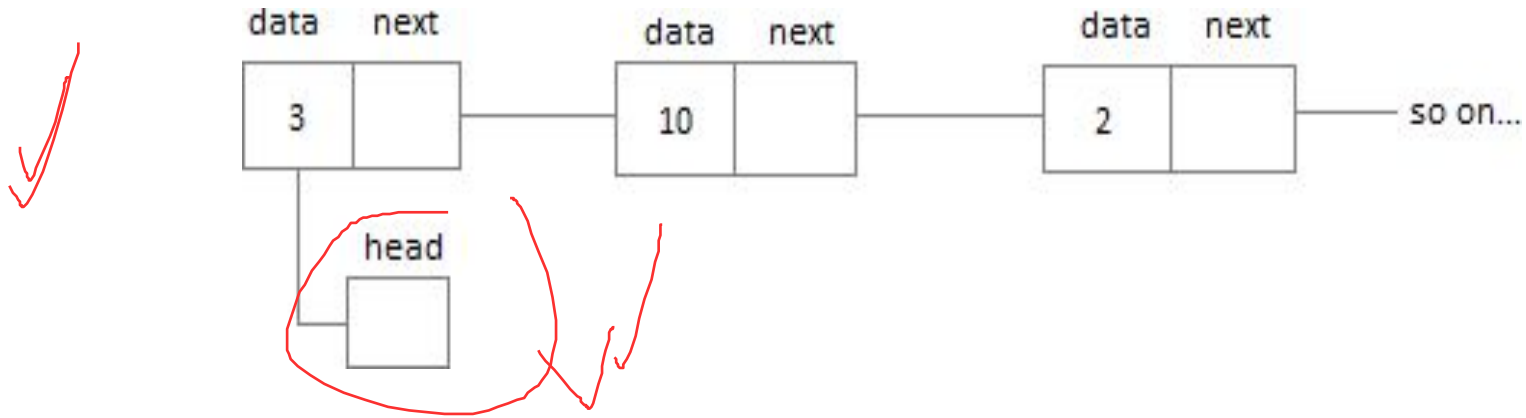
- In music players, we can create our song playlist and can play a song either from starting or ending of the list. And these music players are implemented using a linked list.
- We watch the photos on our laptops or PCs, and we can simply see the next or previous images easily. This feature is implemented using a linked list.
- You must be reading this article on your web browser, and in web browsers, we open multiple URLs, and we can easily switch between those URLs using the previous and next buttons because they are connected using a linked list.

Types of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List

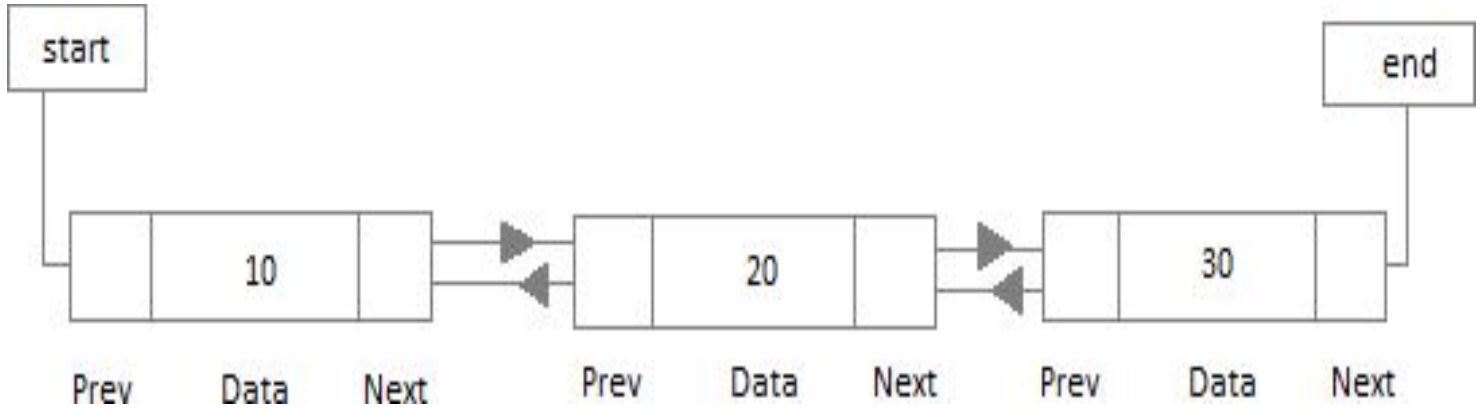
Singly Linked List

- Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. next, which points to the next node in the sequence of nodes.
- The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



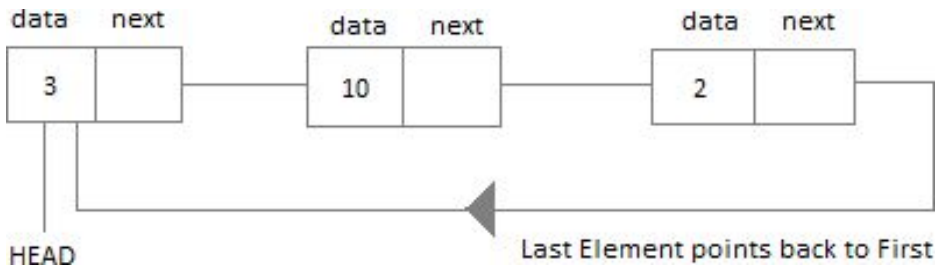
Doubly Linked List

- In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



Circular Linked List

- In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



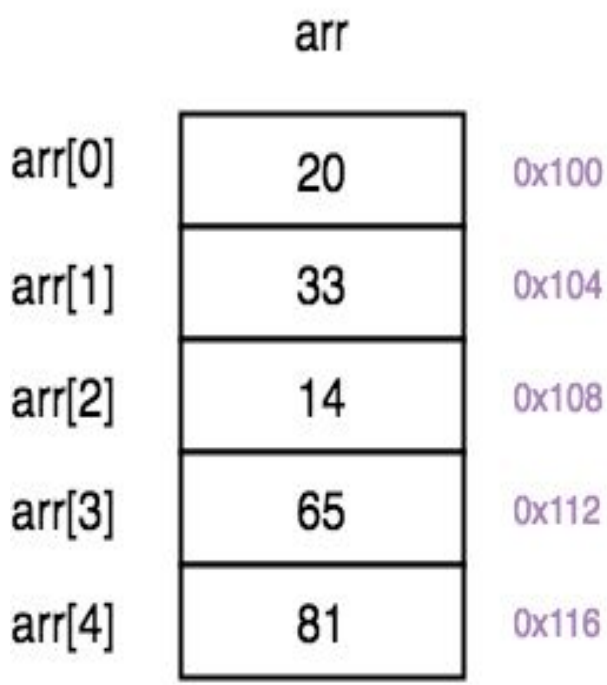
- **Applications of Singly Linked List :**
- The singly linked list is used to implement stack and queue.
- The undo or redo options, the back buttons, etc., that we discussed above are implemented using a singly linked list.
- During the implementation of a hash function, there arises a problem of collision, to deal with this problem, a singly linked list is used.

- **Application of Doubly Linked Lists :**
- The doubly linked list is used to implement data structures like a **stack**, **queue, binary tree,** and **hash table**.
- It is also used in algorithms of LRU (Least Recently used) and MRU(Most Recently Used) cache.
- The undo and redo buttons can be implemented using a doubly-linked list.
- The doubly linked list can also be used in the allocation and deallocation of memory.

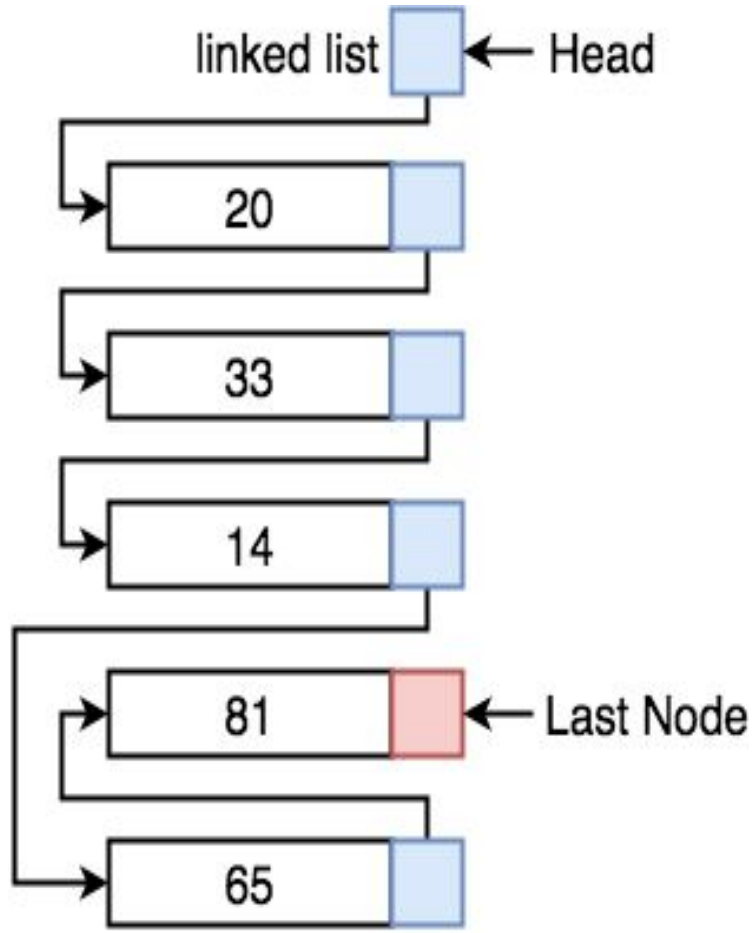
- **Applications of Circular Linked Lists :**
- The circular linked list can be used to implement queues.
- In web browsers, the back button is implemented using a circular linked list.
- In an operating system, a circular linked list can be used in scheduling algorithms like the **Round Robin algorithm**.
- The undo functionality that is present in applications like photo editors etc., is implemented using circular linked lists.
- Circular linked lists can also be used to implement advanced data structures like MRU (Most Recently Used) lists and Fibonacci heap.

Difference between Arrays and Linked List

S.No.	Arrays	Linked Lists
1.	Arrays is static data structure means its size can't be increased or decreased on runtime	Linked list is dynamic data structure means its size can be modified on runtime
2.	If we are confirm to use n fixed block and it is not going to change in a program, so arrays is better	If we are confirm to use n fixed block then linked list use extra space for pointer to next node and it waste 2*n bytes of memory space.
3.	Arrays is simpler to use	Linked list is a complex data structure and it is used basically for complex programming
4.	In arrays, insertion and deletion consequences as large amount of data movements	In linked list, insertion and deletion doesn't need so much data movements



Array representation



Dynamic Memory Allocation Functions

S.No.	Function Name	Meaning
1.	sizeof()	This function gives the size of its arguments in terms of bytes. The arguments can be variable, arrays, structure etc.
2.	malloc()	The malloc() function allocates a request size of bytes and returns a pointer to the first byte of the allocated space.
3.	calloc()	The calloc() function is used to allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
4.	realloc()	The realloc() function is used to reallocate space which is defined by the malloc() or calloc() so modifies the size of previously allocated space.
5.	free()	The free() function is used for efficient use of memory we can also release the memory space that is not required.

Example

- Syntax for malloc():

  ptr=(datatype *)malloc(specified-size);

- Example 1:

  int *ptr; ptr=(int*)malloc(10*sizeof(int));

- Example 2:

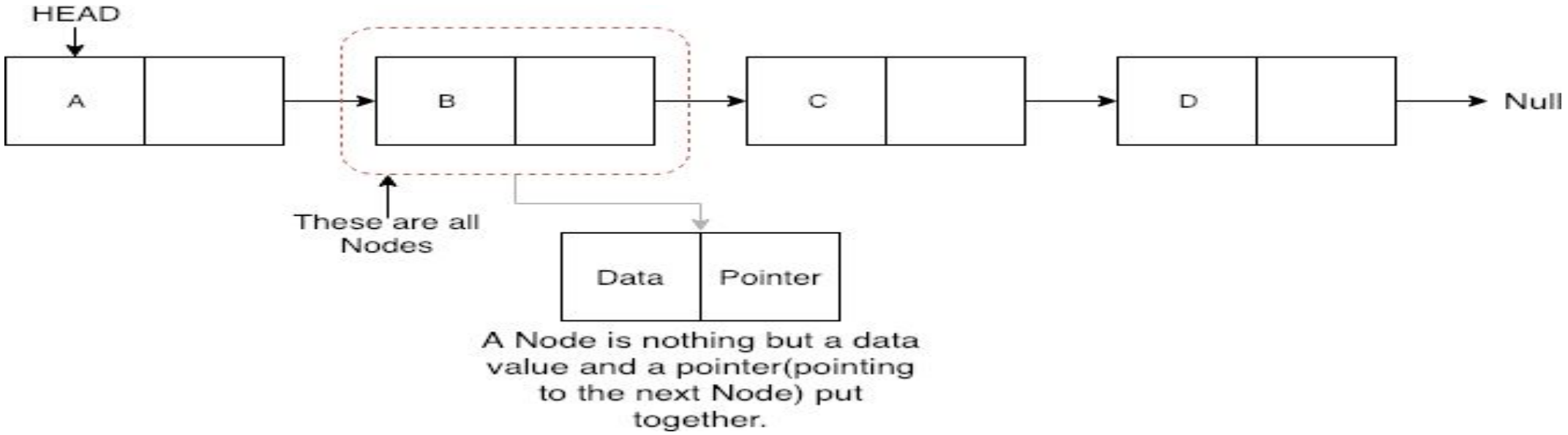
  struct *str;

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

NOTE: malloc and calloc functions return a void pointer, therefore, they can allocate a memory for any type of data. They are used to allocate memory at run time.

Singly Linked List

- What is a Node?
- A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.



Node Implementation

// A linked list node struct (Self-referential structures: structure that contain a reference to the data of its same type)

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

    typedef struct Books {
      char title[50];
      char author[50];
      char subject[100];
      int book_id;
      struct Books *add;
    } Book;
```

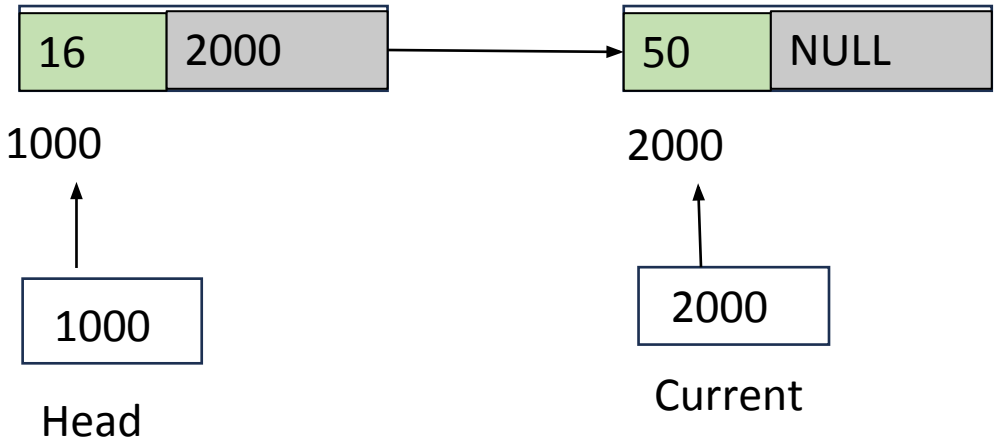
Node Implementation

```
#include<stdio.h>
#include<stdlib.h>
struct node
     {
          int data;
          struct node *link;
    };
    int main(){
     struct node *head=NULL;
    //A pointer to a structure is never itself a structure, but
    merely a variable that holds the address of a structure.
     head=(struct node *) malloc(sizeof(struct node));
     head->data=16;
     head->link=NULL;
     printf("%d", head->data);
     return 0;
    }
```

Node Implementation

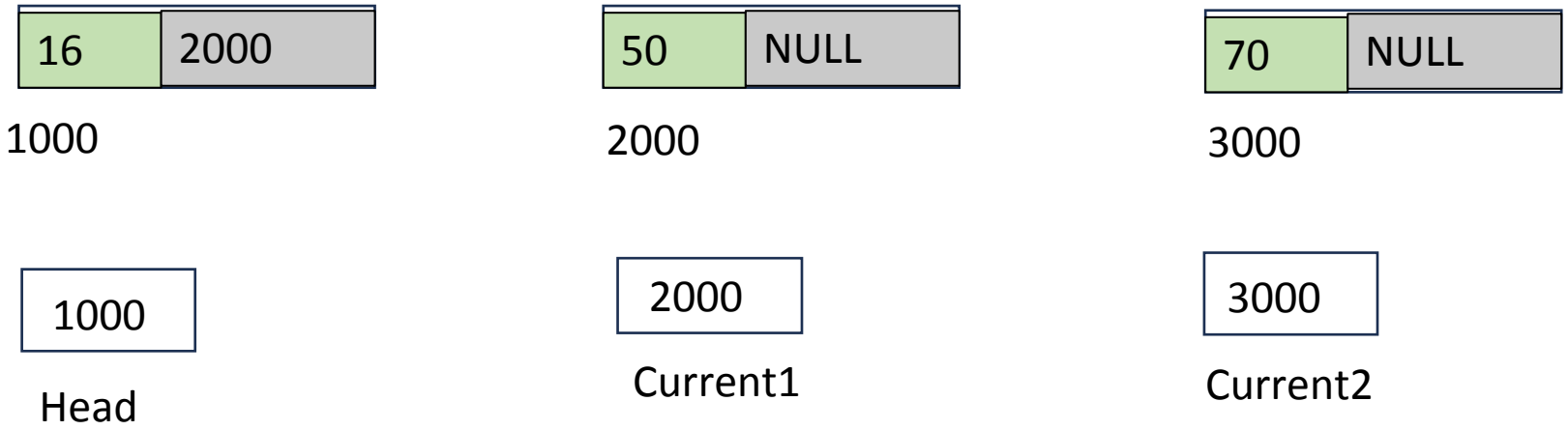
```
#include<stdio.h>
#include<stdlib.h>
struct node
     {
          int data;
          struct node *link;
    };
    int main(){
     struct node *head =(struct node *) malloc(sizeof(struct node));
    head->data=16;
     head->link=NULL;
     struct node *current =(struct node *) malloc(sizeof(struct node));
    current ->data=50;
     head->link= current;
     return 0;
    }
```

Node Implementation




```
        struct node *head =(struct node *) malloc(sizeof(struct node));
head->data=16;
  head->link=NULL;
  struct node *current =(struct node *) malloc(sizeof(struct node));
current ->data=50;
  head->link= current;
```

Node Implementation



Display the contents of the linked list

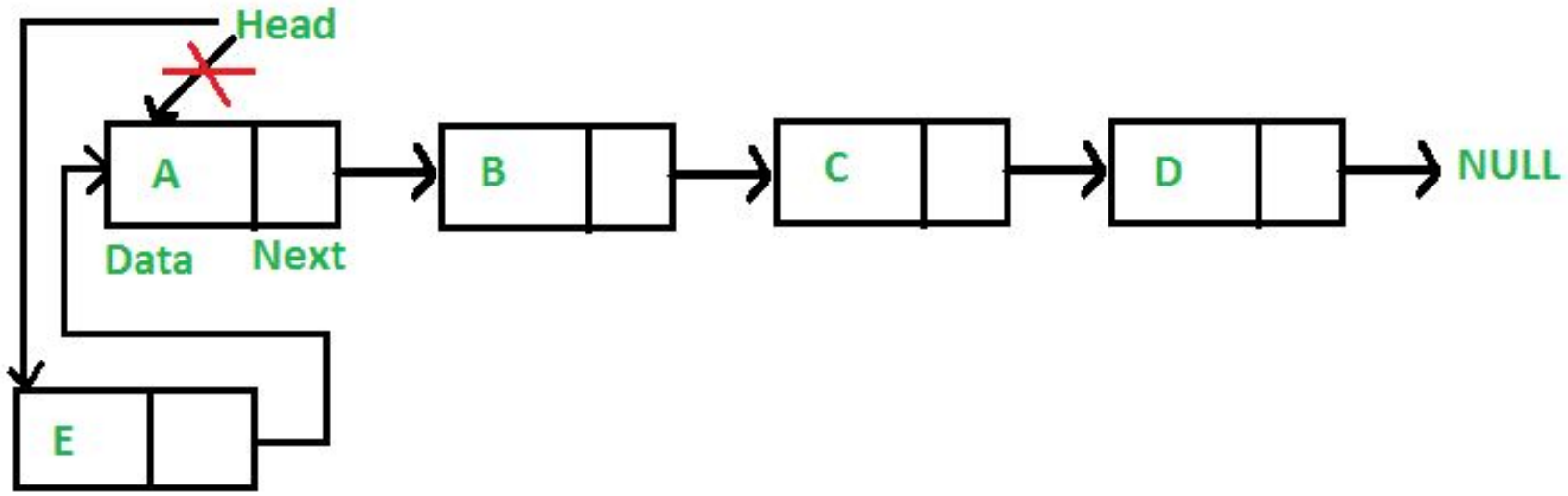
```
void print_list(struct node* head)
{   if (head==NULL)
        printf("Linked list is empty");
    struct node *ptr=head;
     while(ptr!=NULL)
        {printf("%d", ptr->data);
        ptr= ptr->link;
      }
}
```

Inserting a node

- A node can be added in three ways
  1) At the front of the linked list
  2) After a given node.
  3) At the end of the linked list.

To insert a node at the start/beginning/front of a Linked List, we need to:

Make the first node of Linked List linked to the new node
Remove the head from the original first node of Linked List
Make the new node as the Head of the Linked List.



```
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL, *current = NULL;

// display the list
void printList()
{
   struct node *p = head;

      //start from the beginning of list
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }

}
```

```
//insertion at the beginning
void insertatbegin(int data)
{
  //create a new node
  struct node *newnode = (struct node*) malloc(sizeof(struct node));
  newnode->data = data;

  // point it to old first node
  newnode->next = head;

  //point first to new first node
  head = newnode;
}
```

```
void main()
{
  insertatbegin(12);
 printList();
insertatbegin(22);
 printList();
  insertatbegin(32);
printList();
insertatbegin(42);
printList();
}
```

Output

[12 ->]

[22 -> 12 ->]

[32 -> 22 -> 12 ->]

[42 -> 32 -> 22 -> 12 ->]

```
//insertion at the end
void insertatend(int data)
   //create a newnode
   struct node *newnode;
   newnode= (struct node*) malloc(sizeof(struct node));
   newnode->data = data;
   struct node *save = head;

   // point it to last node by traversing
   while(save->next != NULL)
      save = save->next;

   //link  to new first node
   save->next = newnode;
}
```

```
void main()
{
 insertatbegin(12);
 printList();
insertatbegin(22);
 printList();
insertatbegin(32);
 printList();
insertatbegin(42);
printList();
 insertatend(30);
 printList();
 insertatend(44);
printList();
 insertatbegin(50);
   printList();
}
```

```
[ 12 ->]
[ 22 -> 12 ->]
[ 32 -> 22 -> 12 ->]
[ 42 -> 32 -> 22 -> 12 ->]
[ 42 -> 32 -> 22 -> 12 -> 30 ->]
[ 42 -> 32 -> 22 -> 12 -> 30 -> 44 ->]
[ 50 -> 42 -> 32 -> 22 -> 12 -> 30 -> 44 ->]
```

```
void insertafternode(struct node *list, int data)
{
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   lk->next = list->next;
   list->next = lk;
}
```

```
void main()
{
   insertatbegin(12);
   insertatbegin(22);
   insertatend(30);
   insertatend(44);
   insertatbegin(50);
   insertafternode(head->next->next, 33);
   printf("Linked List: ");
```

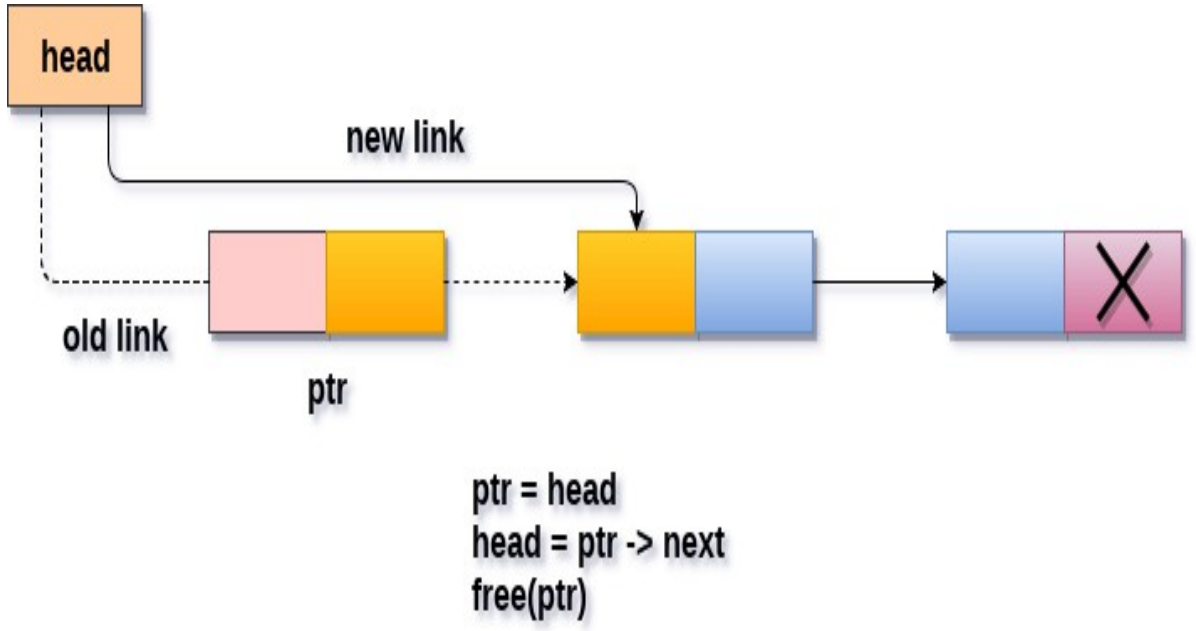
```
[ 12 ->]
[ 22 -> 12 ->]
[ 22 -> 12 -> 30 ->]
[ 22 -> 12 -> 30 -> 44 ->]
[ 50 -> 22 -> 12 -> 30 -> 44 ->]
 Linked List:
[ 50 -> 22 -> 12 -> 33 -> 30 -> 44 ->]
```

```
void deleteatbegin()
{
   head = head->next;
// how do we free the deleted node?
}

void deleteatend()
{
   struct node *linkedlist = head;
   while (linkedlist->next->next != NULL)
      linkedlist = linkedlist->next;
   linkedlist->next = NULL;
}
```

[12 ->]
[22 -> 12 ->]
[22 -> 12 -> 30 ->]
[22 -> 12 -> 30 -> 44 ->]
[50 -> 22 -> 12 -> 30 -> 44 ->]
Linked List:
[50 -> 22 -> 12 -> 33 -> 30 -> 44 ->]
Linked List after deleting first node:
[22 -> 12 -> 33 -> 30 -> 44 ->]
Linked List after deleting last node
[22 -> 12 -> 33 -> 30 ->]
Linked List after deletion of given node 12
[22 -> 33 -> 30 ->]

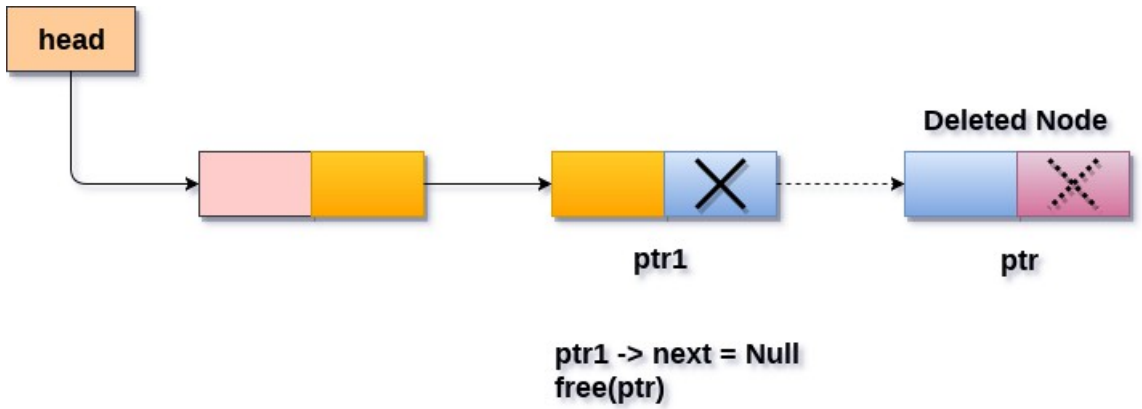
Delete a node at the front

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



Delete a node at the end

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

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



Singly Linked List Implementation

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *next;
};
typedef struct node *NODE;
NODE insertFront(NODE first);
NODE insertRear(NODE first);
NODE insertAfter(NODE first);
NODE insertBefore(NODE first);
NODE insertAtPos(NODE first);
NODE deleteFront(NODE first);
NODE deleteRear(NODE first);
NODE deleteAfterEle(NODE first);
NODE deleteBeforeEle(NODE first);
NODE deleteElement(NODE first);
NODE deletePos(NODE first);
void display(NODE first);
```

Singly Linked List Implementation

```
void main()
{
   NODE first=NULL;
   int choice;
   while(1)
   {
      printf("\n\n*****Singly linked list implementation*****");
      printf("\n 1. Insert Front \n 2. Insert rear \n 3. Insert After \n
                        4. Insert Before \n 5. Insert At Position \n
                        6. Delete Front \n 7. Delete Rear \n
                        8. Delete After \n 9. Delete Before \n 10. Delete Element \n
                        11. Delete At Position \n 12. Display \n 13. Exit");
      printf("\n\t**********");
      printf("\nEnter your choice ");
      scanf("%d",&choice);
```

```
*****Singly linked list implementation*****
 1. Insert Front
 2. Insert rear
 3. Insert After
 4. Insert Before
 5. Insert At Position
 6. Delete Front
 7. Delete Rear
 8. Delete After
 9. Delete Before
 10. Delete Element
 11. Delete At Position
 12. Display
 13. Exit
      **********
Enter your choice
```

Singly Linked List Implementation

```
switch (choice)
        {
        case 1: first=insertFront(first); break;
        case 2: first=insertRear(first); break;
        case 3: first=insertAfter(first); break;
        case 4: first=insertBefore(first); break;
        case 5: first=insertAtPos(first); break;
        case 6: first=deleteFront(first); break;
        case 7: first=deleteRear(first); break;
        case 8: first=deleteAfterEle(first); break;
        case 9: first=deleteBeforeEle(first); break;
        case 10: first=deleteElement(first); break;
        case 11: first=deletePos(first); break;
        case 12: display(first); break;
        case 13: printf("\n Program exits now"); exit(0);
        default: printf("enter valid choice");
        }
    }
}
```

Display

```
void display(NODE first)
{
   NODE cur;
   if(first==NULL)
   printf("No elements to display");
   else
   {
      cur=first;
      printf("\n Elements of Singly linked list are:\t");
      while(cur!=NULL)
      {
         printf("%d\t",cur->info);
         cur=cur->next;
      }
   }
}
```

Insert in Front

```
NODE insertFront(NODE first)
{
   NODE temp=NULL;
   temp=(NODE)malloc(sizeof(struct node));
   if (temp==NULL)
   {
      printf("Insufficient memory");
      return first;
   }
   printf("\n Enter element to be inserted");
   scanf("%d",&temp->info);
   temp->next=NULL;
   if(first==NULL)
   first=temp;
   else    {
      temp->next=first;
      first=temp;
      return first;
   }
}
```

Insert at Rear

```
NODE insertRear(NODE first)
{
   NODE temp=NULL,cur=NULL;
   temp=(NODE)malloc(sizeof(struct node));
   if (temp==NULL)
   {
      printf("Insufficient memory");
      return first;
   }
   printf("\n Enter element to be inserted");
   scanf("%d",&temp->info);
   temp->next=NULL;
   if(first==NULL)
        first=temp;
   else
   {
      cur=first;
      while(cur->next!=NULL)
      {
         cur=cur->next;
      }
      cur->next=temp;
   }    return first;   }
```

Insert After an item

```
NODE insertAfter(NODE first)
{
   NODE temp=NULL,cur=NULL;
   temp=(NODE)malloc(sizeof(struct node));
   if (temp==NULL)
   {
      printf("Insufficient memory");
      return first;
   }
   int ele,item;
   if(first==NULL)
   {
      printf("linked list is empty");
      return first;
   }
      printf("\n Enter element after which new node to be inserted");
      scanf("%d",&ele);
```

Insert After an item

```
cur=first;
      while(cur!=NULL&&cur->info!=ele)
         cur=cur->next;
      if(cur==NULL)
      {
         printf("Element not found");
         return first;
      }
      printf("\nEnter element to be inserted");
      scanf("%d",&item);
      temp->info=item;
      temp->next=NULL;
      temp->next=cur->next;
      cur->next=temp;
      return first;
}
```

```
Elements of Singly linked list are:   2     1     3

*****Singly linked list implementation*****
 1. Insert Front
 2. Insert rear
 3. Insert After
 4. Insert Before
 5. Insert At Position
 6. Delete Front
 7. Delete Rear
 8. Delete After
 9. Delete Before
 10. Delete Element
 11. Delete At Position
 12. Display
 13. Exit
      ***********
Enter your choice 3
 Enter element after which new node to be inserted 1
Enter element to be inserted5

Enter your choice 12

 Elements of Singly linked list are:   2     1     5     3
```

Insert Before an item

```
NODE insertBefore(NODE first)
{
   NODE temp=NULL, cur=NULL, prev=NULL;
   int ele,item;
   temp=(NODE)malloc(sizeof(struct node));
   if (temp==NULL)
   {
      printf("Insufficient memory");
      return first;
   }
   printf("\n Enter element before which new node to be inserted");
   scanf("%d",&ele);
```

Insert Before an item

Elements of Singly linked list are: 2 1 3

```
cur=first;
      while(cur!=NULL&&cur->info!=ele)
      {   prev=cur;
         cur=cur->next;       }
      if(cur==NULL)
      {     printf("Element not found");
          return first;       }
      printf("Element to be inserted");
      scanf("%d",&item);
      temp->info=item;
      temp->next=NULL;
      temp->next=cur;
      if(prev!=NULL)
         prev->next=temp;
      else
         first=temp;
      return first;
}
```

Insert at Position

```
NODE insertAtPos(NODE first)
{
   NODE temp=NULL,cur=NULL;
   temp=(NODE)malloc(sizeof(struct node));
   if (temp==NULL)
   {
      printf("Insufficient memory");
      return first;
   }
   int ele,pos;
   if(first==NULL)
   {
      printf("linked list is empty");
      return first;
   }
```

Insert at Position

```
printf("Enter pos at which new element to be inserted ");
    scanf("%d",&pos);
    printf("Enter element to be inserted at pos ");
    scanf("%d",&ele);
    if(pos==1)
    {
        first=insertFront(first);
        return first;
    }
```

Insert at Position

```
cur=first;
  int i=1;
  while(cur!=NULL&&i<pos-1)
  {
     cur=cur->next;
     i++;
  }
  if(cur==NULL)
     {
        printf("Element not found");
        return first;
     }
     temp->info=ele;
     temp->next=NULL;
     temp->next=cur->next;
     cur->next=temp;
     return first;
}
```

Elements of Singly linked list are: 2 6 1 5 3

*****Singly linked list implementation*****

1. Insert Front
2. Insert rear
3. Insert After
4. Insert Before
5. Insert At Position
6. Delete Front
7. Delete Rear
8. Delete After
9. Delete Before
10. Delete Element
11. Delete At Position
12. Display
13. Exit

****

Enter your choice 5

Enter pos at which new element to be inserted 4

Enter element to be inserted at pos 7

Enter your choice 12

Elements of Singly linked list are: 2 6 1 7 5 3

Delete Front

```
NODE deleteFront(NODE first)
{
    NODE temp=NULL;
    if(first==NULL)
    {
        printf("Linked List is empty, create Linked list");
        return first;
    }
    temp=first;
    first=first->next;
    printf("Element being deleted is %d",temp->info);
    free(temp);
    return first;
}
```

Delete Rear

```
NODE deleteRear(NODE first)
{
   NODE cur=NULL,prev=NULL;
   prev=(NODE)malloc(sizeof(struct node));
   if (prev==NULL)
   {
      printf("Insufficient memory");
      return first;
   }
   if(first==NULL)
   {
      printf("LL is empty");
      return first;
   }
```

Delete Rear

```
cur=first;
    prev=NULL;
    while(cur->next!=NULL)
    {
        prev=cur;
        cur=cur->next;
    }
    prev->next=NULL;
    printf("Element being deleted is %d",cur->info);
    free(cur);
    return first;
}
```

Delete Element

```
NODE deleteElement(NODE first)
{
    NODE cur = NULL, prev = NULL;
    int item;
    if(first==NULL)
    {
        printf("\nThe list is empty\n");
        return first;
    }
    printf("\nEnter the element to be deleted :");
    scanf("%d",&item);
```

Delete Element

```
cur=first;
   while(cur!=NULL && cur->info!=item)
   {   prev=cur;
       cur=cur->next;    }
   if(cur==NULL)
   {   printf("\nElement to be deleted doesnt exist in the list\n");
       return first;    }
   if(prev==NULL)
   {   first = deleteFront(first);
       return first;    }
   prev->next = cur->next;
   printf("\nElement being deleted is : %d\n", cur->info);
   free(cur);
   return first;
}
```

Delete at Position

```
NODE deletePos(NODE first)
{
   NODE cur = NULL, prev = NULL;
   int pos, k;
   if(first==NULL)
   {       printf("\nThe list is empty.. no elements to delete...\n");
          return first;
   }
   printf("\nEnter the position of element to be deleted :");
   scanf("%d",&pos);
   if(pos==1)
   {
      first = deleteFront(first);
      return first;
   }
```

Delete at Position

```
cur=first;
   k=1;
   while(cur!=NULL && k<pos)
   {  prev=cur;
      cur=cur->next;
      k++;
   }
   if(cur==NULL)
   {   printf("\nPosition doesnt exist in the list\n");
      return first;    }
   prev->next = cur->next;
   printf("\nElement being deleted is : %d\n", cur->info);
   free(cur);
   return first;
   }
```

Delete Before Element

```
NODE deleteBeforeEle(NODE first)
{  NODE cur = NULL, prev = NULL, pprev = NULL;
   int ele;
   if(first==NULL)
   {   printf("\nThe list is empty.. no elements to delete...\n");
       return first;
   }
   printf("\nEnter an element whose left element to be deleted :");
   scanf("%d",&ele);
   cur=first;
   while(cur!=NULL && cur->info!=ele)
   {    pprev = prev;
       prev=cur;
       cur=cur->next;
   }
```

Delete Before Element

```
 if(cur==NULL)
    {
       printf("\nElement doesnt exist in the list\n");
       return first;
    }
    if(pprev==NULL)
    {
       first = deleteFront(first);
       return first;
    }
    pprev->next = prev->next;
    printf("\nElement being deleted is : %d\n", prev->info);
    free(prev);
    return first;
}
```

Delete After Element

```
NODE deleteAfterEle(NODE first)
{
    NODE cur = NULL, temp = NULL;
    int ele;
    if(first==NULL)
    {   printf("\nThe list is empty.. no elements to delete...\n");
        return first;
    }
    printf("\nEnter an element whose right element to be deleted :");
    scanf("%d",&ele);
    cur=first;
    while(cur!=NULL && cur->info!=ele)
    {       cur=cur->next;
    }
```

Delete After Element

```
 if(cur==NULL)
    {
        printf("\nElement doesnt exist in the list\n");
        return first;
    }
    if(cur->next == NULL)
    {   printf("\nNo elements to delete after the given element...");
        return first;
    }
    temp = cur->next;
    cur->next = temp->next;
    printf("\nElement being deleted is : %d\n", temp->info);
    free(temp);
    return first;
}
```

Instructions to Students to be followed in each lab:

1. Each Student should write down the program in the observation book and get it evaluated by the respective lab faculty in-charge and then execute the program.

2. Each Student should bring the lab record with the programs and output written for the programs completed in their respective previous week and get it evaluated by the lab faculty in-charge. In the record book students should - Handwrite the Program - Pasting of the printout of the Output or Handwriting of the Output (Output should be written for all the cases).

3. Continuous Internal Evaluation for each lab is for 10 marks which includes execution of the program in the allotted lab time and showing the output. If Leetcode program is present for a particular lab, student needs to complete that also within the allotted lab slot only and show the output. Observation book needs to be corrected on the same day itself.

Note: wherever LeetCode program is present, the program to be executed will be shared during the particular lab week.

Note: Submission during the lab slot: 10 marks

Submission on the same day but after lab slot: 8 marks

Submission within 1 week the program was assigned: 6 marks

Submission within 2 weeks the program was assigned: 5 marks

Submission any later: 0 marks

Lab Program

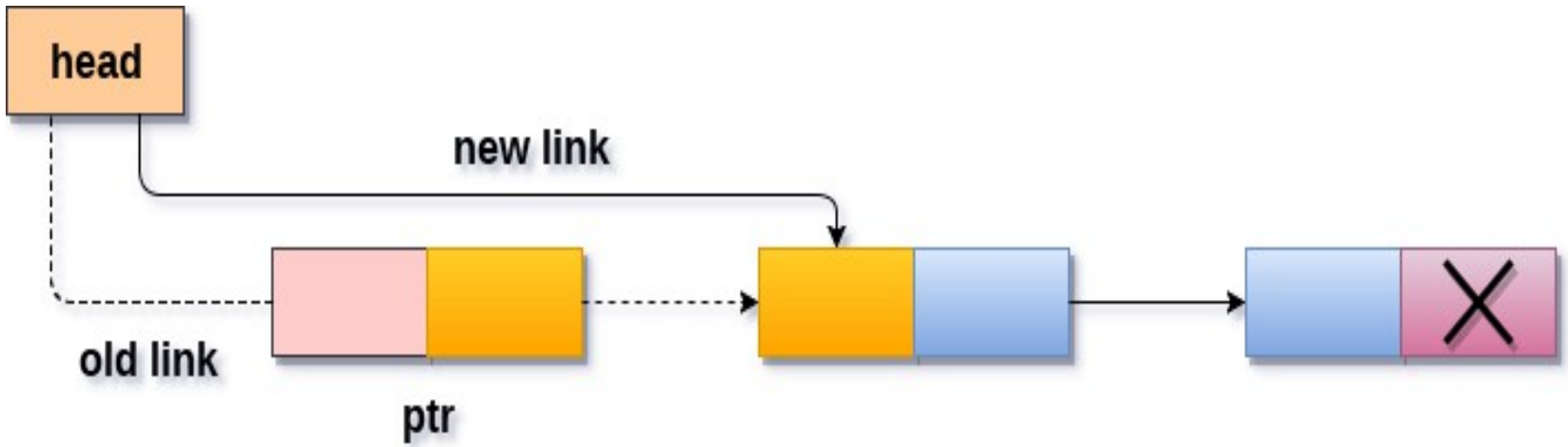
1. Write a program to implement Singly Linked List with following operations
    a) Create a linked list.
    b) Insertion of a node at first position, at any position and at end of list.
    c) Display the contents of the linked list.

2.Write a program to Implement Singly Linked List with following operations
    a) Create a linked list.
    b) Deletion of first element, specified element and last element in the list.
    c) Display the contents of the linked list.

Singly Linked List:Deleting a node

- A node can be deleted in three ways
  1) At the front of the linked list (Beginning)
  2) After a given node/specified position
  3) At the end of the linked list.

Delete a node at the front

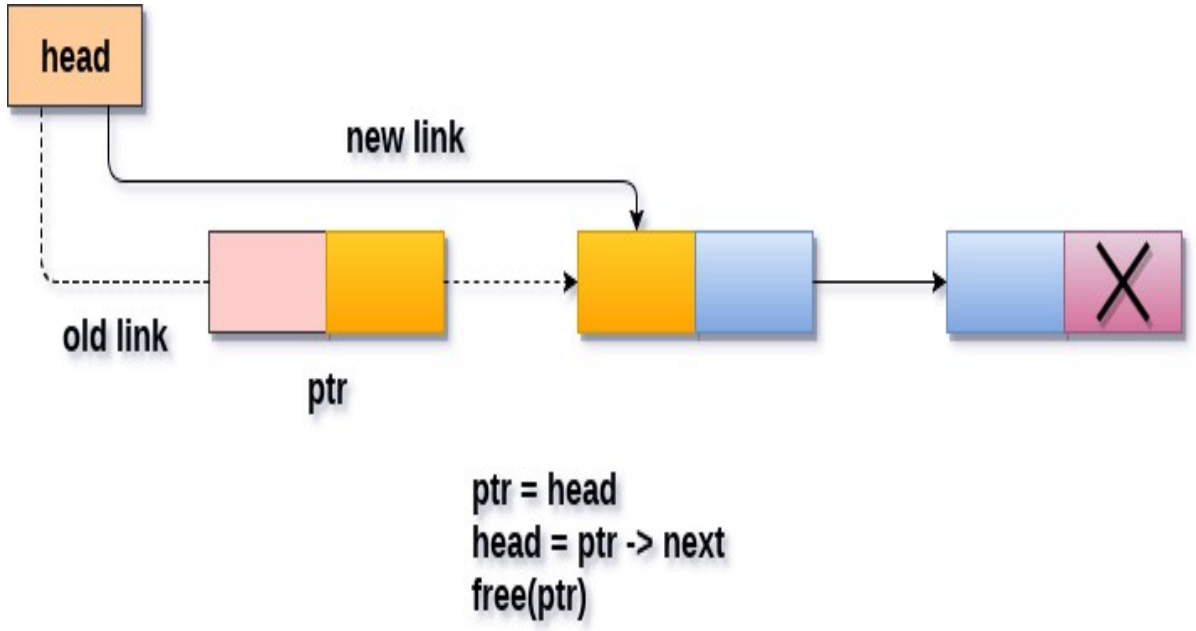



```
ptr = head
head = ptr -> next
free(ptr)
```

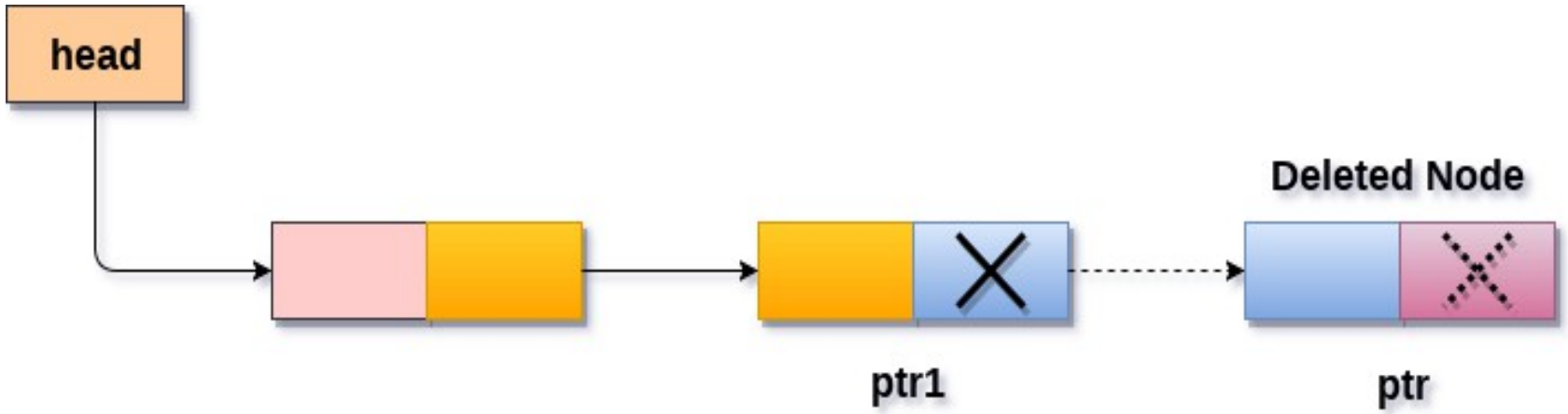
First check if the list is empty. If not empty continue
Point head to the next node i.e. second node
    temp = head
    head = head->next
Make sure to free unused memory
    free(temp); or delete temp;

Delete a node at the front

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

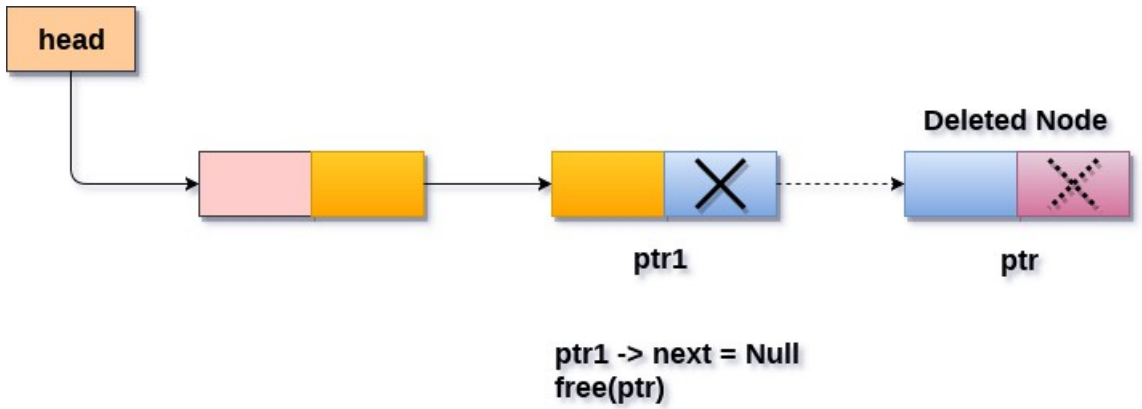


Delete a node at the end

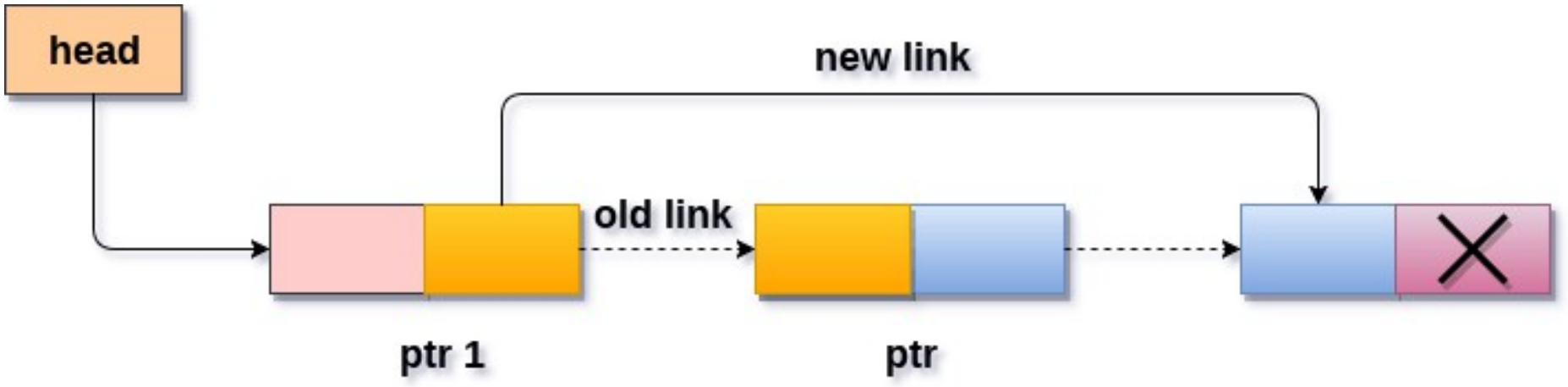


Delete a node at the end

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

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



Delete a node at specified position



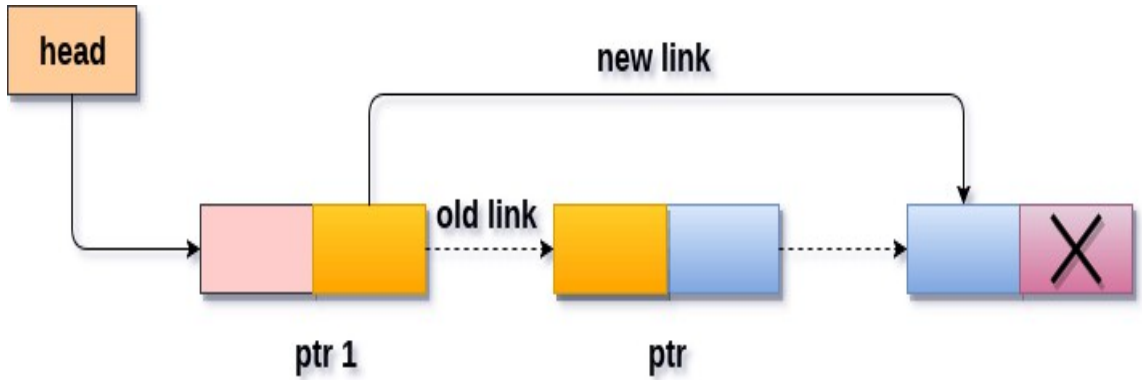
ptr1 -> next = ptr -> next
free(ptr)

Steps to delete a node from the linked list:

- Find the previous node of the node to be deleted.
- Change the **next** of the previous node.
- Free memory for the node to be deleted.

Delete a node at specified position

```
void delete_specified()
  {
    struct node *ptr, *ptr1; int loc,i;
    scanf("%d",&loc); ptr=head;
    for(i=0;i<loc;i++)
    {
      ptr1 = ptr;
      ptr = ptr->next;

      if(ptr == NULL)
      {
        printf("\nThere are less than %d elements in the
        list..\n",loc);
        return;
      }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted %d node ",loc);
  }
```



Singly Linked List Operations

- Search
- Count number of nodes
- Concatenation
- Merging
- Reversing

Search

Search for an element in the linked list:

- Initialize a node pointer, **current = head**.
- Do following while current is not NULL
- Get the key to be searched
- If the current value (i.e., **current->data**) is equal to the key being searched return true.
- Otherwise, move to the next node (**current = current->next**).
- If the key is not found, return false

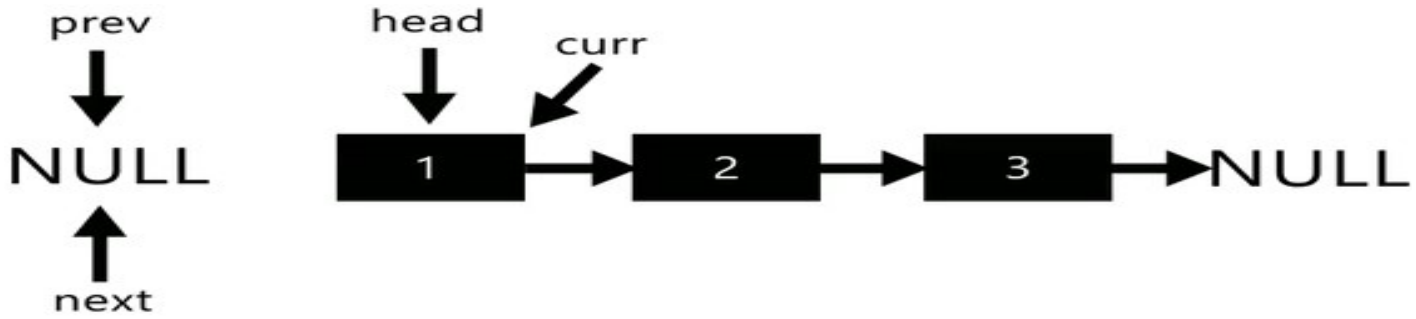
Search

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}
```

Count number of nodes

```
void Count_nodes(struct node* head )
{
/* temp pointer points to head */
     struct node* temp = head;
/* Initialize count variable */
     int count=0;
/* Traverse the linked list and maintain the count */
    while(temp != NULL)
     {
        temp = temp->next;
     /* Increment count variable. */
        count++;
    }
/* Print the total count. */
printf("\n Total no. of nodes is %d",count);
}
```

Reversing



Given a pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing the links between nodes.

Input: *Head of following linked list*
1->2->3->4->NULL
Output: *Linked list should be changed to,*
4->3->2->1->NULL

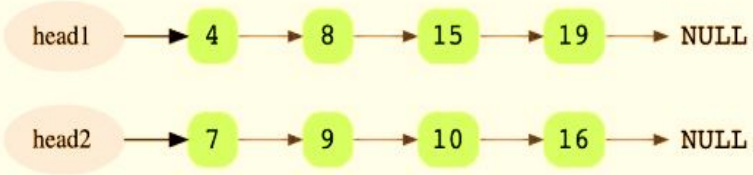
```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

Concatenation

```
void concatenate(struct node *a,struct node *b)
{
    if( a != NULL && b!= NULL )
    {
        if (a->next == NULL)
            a->next = b;
        else
            concatenate(a->next,b);
    }
    else
    {
        printf("Either a or b is NULL\n");
    }
}

struct node *concat( struct node *start1,struct node *start2)
{
        struct node *ptr;
        if(start1==NULL)
        {
                start1=start2;
                return start1;
        }
        if(start2==NULL)
                return start1;
        ptr=start1;
        while(ptr->link!=NULL)
                ptr=ptr->link;
        ptr->link=start2;
        return start1;
}
```

```
NodePtr merge_sorted(NodePtr head1, NodePtr head2) {

  // if both lists are empty then merged list is also empty
  // if one of the lists is empty then other is the merged list
  if (head1 == nullptr) {
    return head2;
  } else if (head2 == nullptr) {
    return head1;
  }


  NodePtr mergedHead = nullptr;
  if (head1->data <= head2->data) {
    mergedHead = head1;
    head1 = head1->next;
  } else {
    mergedHead = head2;
    head2 = head2->next;
  }

  NodePtr mergedTail = mergedHead;

  while (head1 != nullptr && head2 != nullptr) {
    NodePtr temp = nullptr;
    if (head1->data <= head2->data) {
      temp = head1;
      head1 = head1->next;
    } else {
      temp = head2;
      head2 = head2->next;
    }

    mergedTail->next = temp;
    mergedTail = temp;
  }

  if (head1 != nullptr) {
    mergedTail->next = head1;
  } else if (head2 != nullptr) {
    mergedTail->next = head2;
  }

  return mergedHead;
}
```

Consider two sorted linked lists as an example.



The merged linked list should look like this:

Lab Program

3. Write a program to Implement Singly Linked List with following operations
    a) Sort the linked list.
    b) Reverse the linked list.
    c) Concatenation of two linked lists