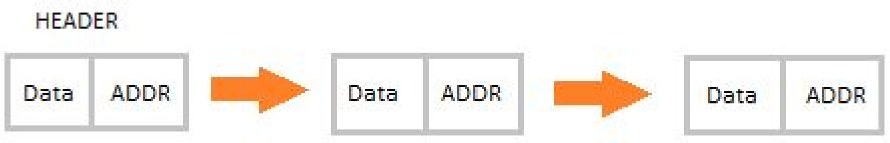
Linked Lists

Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

Linked List is a very commonly used linear data structure which consists of set of **nodes** in a sequence.

Each node has two fields known as data and link. **Data** field stores actual data and **link** contains address of next node to point to the next node.

Linked Lists are used to create trees and graphs.



A linked list is a collection of nodes ordered by links that are stored as part of node. They are not ordered by their physical placement in memory.

Chain is a single linked list that comprised of zero or more nodes and with 0 in last node. when the number of nodes is zero, chain is empty. the last node of chain has 0 link.

Problems with Arrays:

1. Prediction of size – Size of an array must be specified precisely at the beginning which may be difficult task in many practical applications.
2. Static memory allocation—Memory allotment is required at the compile time.
3. Wastage of space—inefficient usage of storage memory.
4. Inefficient implementation of insertions and deletions as they require shift operations.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Size of linked list can grow or shrink in size during the execution of a program.
- In Linked Lists we don't need to know the size in advance
- Stacks and queues can be easily executed.

- Linked List provides flexibility in allowing the items to be rearranged efficiently.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- Access time is linear O(n). No element can be accessed randomly; it has to access each node sequentially.
- Many basic operations—such as obtaining the last node of the list, finding a node that contains a given datum, or locating the place where a new node should be inserted—may require iterating through most or all of the list element
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement several other common abstract data types(data structures), including lists, stacks, queues, associative arrays, and S-expressions.
- Linked lists let you insert elements at the beginning and end of the list.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

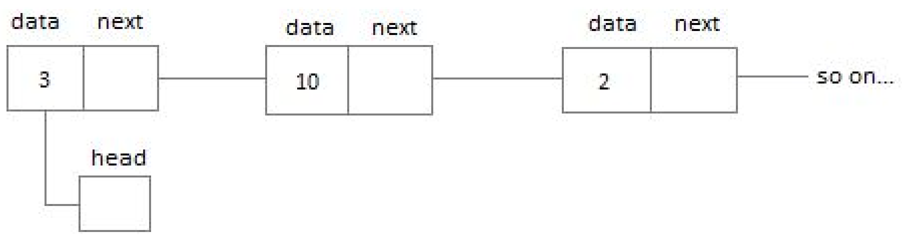
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List(Single or Double)

Let's know more about them and how they are different from each other.

Singly Linked List

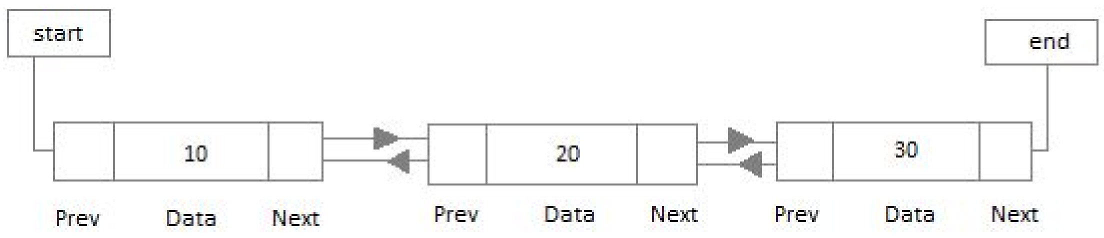
Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



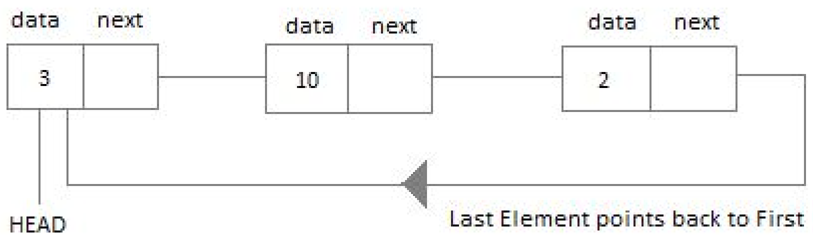
Doubly Linked List

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



Single Linked List

What is Linked List?

When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

What is Single Linked List?

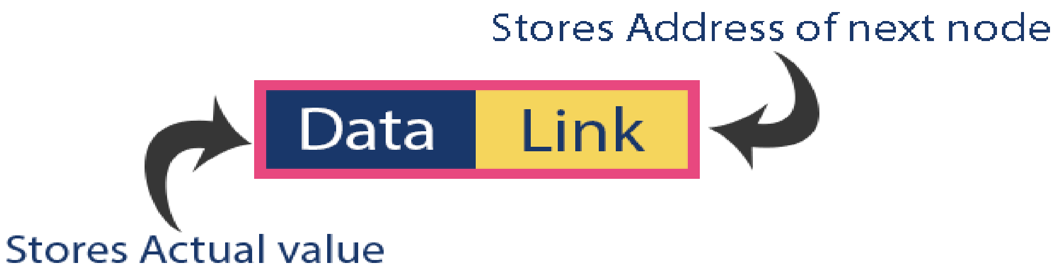
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows...



In a single linked list, the address of the first node is always stored in a reference node known as "head"(Some times it is also known as "front").
Note:-- Always next part (reference part) of the last node must be NULL.

A node is represented as

Struct node

{

Int data;

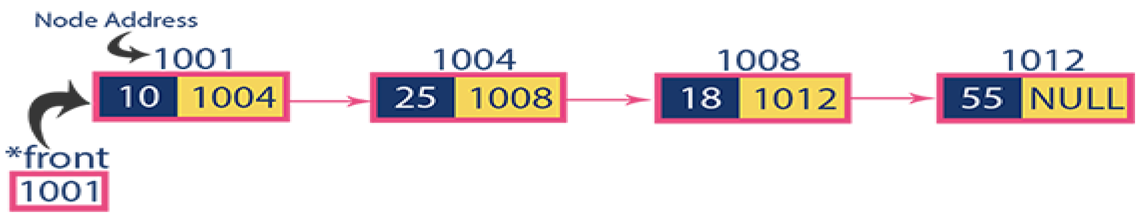
Struct node *next;

};

Int data refers to the data part of a node and Struct node *next is a pointer refers the next node in the list.

In a linked list, every node contains a pointer to another node which is of same type, called as “Self referential data type”, and structure is self referential structure.

Example:



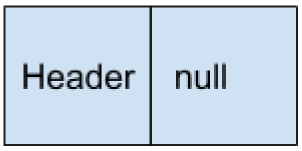
Operations for Chains

In a single linked list we perform the following operations...

- Insertion
- Deletion
- Display

Algorithm to create a node:

```
Nptr Createnode()

Begin

1.Allocate memory for header node.

   h=(nptr)(malloc(sizeof(struct node)));

2. Verify the memory allocation.

         If(h==NULL) then

                  Printf("Memory not Allocated")

                  Return;

3.Allocate the next of header as NULL.

         H→next=NULL;
```

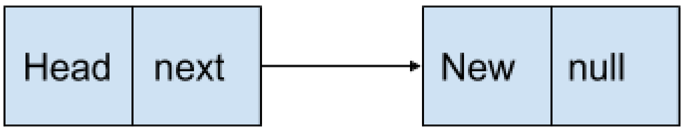


```
4.Return h.

End.
```

Algorithm to create a list:

```
Algorithm Createlist()

Begin

1.Create empty list.

   h=(nptr)(malloc(sizeof(struct node)));

   h→next=NULL;
```

```
    Temp=h;
2. Repeat the following steps while user enters positive choice.
        Read x;
3.Allocate the space for new node.
        new=(nptr)(malloc(sizeof(struct node)));
4. Assign x value to new node data and link to NULL.
      new->data=x;
      new->next=temp->next;
5. Link new node with previous node
          temp->next=new;
          temp=new;
6. End while loop
```



so on nodes can be created

End.

Algorithm to insert a node:

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.

Step 2: Declare all the user defined functions.

Step 3: Define a Node structure with two members data and next

Step 4: Define a Node pointer 'head' and set it to NULL.

Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

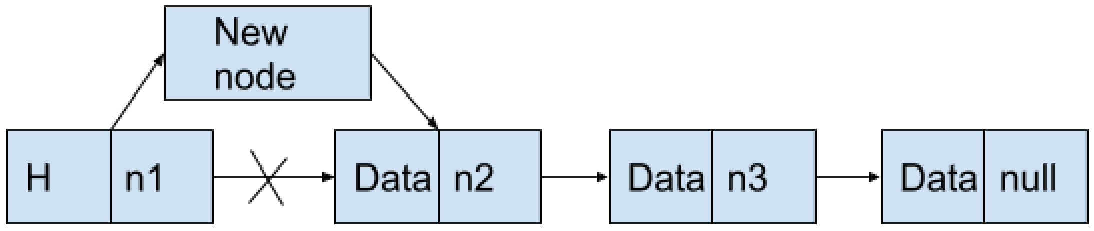
In a single linked list, the insertion operation can be performed in three ways. They are as follows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Insertion at Beginning of the list



We can use the following steps to insert a new node at beginning of the single linked list...

Step 1: Create a new node with given value.

Step 2: Check whether list is Empty (head == NULL)

Step3: If it is Empty then, create head node.

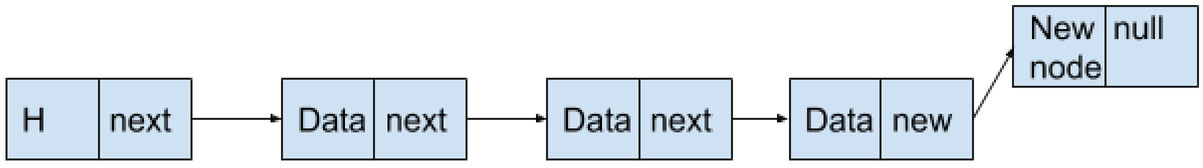
```
h=(nptr)(malloc(sizeof(struct node)));
H→next=NULL;
Temp=h;
```

Step4: If it is Not Empty then, create new node.

```
new=(nptr)(malloc(sizeof(struct node)));
new->data=x;
 new->next=h->next;
 h->next=new;
```

Step5: Return h.

Inserting At End of the list



We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a newNode with given value.

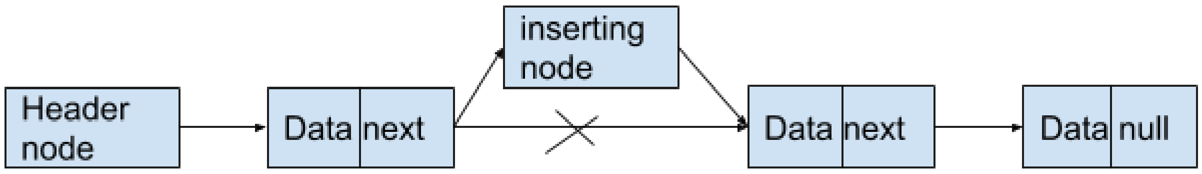
```
 new=(nptr)(malloc(sizeof(struct node)));

new->data=x;

 new->next=NULL;
```

Step 2: If list is Not Empty then, define a node pointer temp and initialize with head.

Step 3: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 4: Set temp → next = new.

Inserting At Specific location in the list (After a Node)



We can use the following steps to insert a new node after a node in the single linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Not Empty then, define a node pointer temp and initialize with head→next.

Step 5: Keep moving the temp node until it reaches to the node before/after which we want to insert the new Node . If it is before

```
{

  new->next=temp->next;

 tem p->next=new;

  printf("inserted");

  return;

}

else if it is after

{

  new->next=p->next->next;

  p->next->next=new;
```

```
        printf("inserted");
        return;
       }
```

Step 6: Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function.

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list is having only one node (temp → next == NULL)

Step 5: If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)

Step 6: If it is FALSE then set head = temp → next, and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

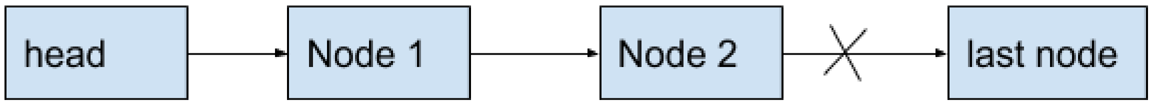
Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Check whether list has only one Node (temp1 → next == NULL)

Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)

Step 7: Finally, Set temp2 → next = NULL and delete temp1.



Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).

Step 8: If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).

Step 9: If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.

Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11: If temp1 is last node then set temp2 → next = NULL and
delete temp1 (free(temp1)).

Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 →
next and delete temp1 (free(temp1)).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5: Finally display temp → data with arrow pointing to NULL (temp → data --->
NULL).

SINGLE LINKED LIST PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
typedef struct node *nptr;
struct node
{
   int data;
   nptr next;
};

nptr createolist();
nptr createnode();
void displaylist(nptr);
void findnode(nptr);
void deletenode(nptr);
void insertnode(nptr);
void insertnodefirst(nptr);
void insertnodelast(nptr);

int main()
{

   nptr h,ho;
   int ch,d,x,n;
   while(1)
   {
        printf("\nMenu:\n1.Create Ordered List\n2.Create Node3.Delete an item\n4.Find an
item\n5.Print the List\n6.Insert Node\n7.Insert node at First\n8.Insert node at Last\n9.
Exit\n");
        printf("enter choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:ho=createolist();
                   displaylist(ho);
                   break;
            case 2:h=createnode();
                   break;
            case 3:printf("list before deletion\n");
                   displaylist(h);
                   deletenode(h);
                   printf("list after deletion\n");
                   displaylist(h);
                   break;
            case 4:findnode(h);
                  break;
            case 5:displaylist(h);
                   break;
            case 6: insertnode(h);
                   displaylist(h);
                   break;
            case 7: insertnodefirst(h);
                   displaylist(h);
                   break;
            case 8: insertnodelast(h);
                   displaylist(h);
                   break;
            case 9:return 0;

        }
    }
}
nptr createolist()
    {
    nptr h,new,temp;
    int x,d;
    h=(nptr)(malloc(sizeof(struct node)));
    h->next=NULL;
    temp=h;
    printf("\n\nh is %d", h);
    while(1)
    {

      printf("enter data,to stop enter negative number\n");
```

```
  scanf("%d",&x);
  if(x<=0)
    break;
  else
  {
   new=(nptr)(malloc(sizeof(struct node)));
   new->data=x;
   new->next=temp->next;
   temp->next=new;
   temp=new;
  }

}
return h;
}
nptr createnode()
{
nptr h;
h=(nptr)(malloc(sizeof(struct node)));
h->next=NULL;
return h;
}
void displaylist(nptr h)
{
    nptr p;
  if(h->next==NULL)
  {
      printf("empty list");
      return;
  }
  else
    {
    for(p=h->next;p!=NULL;p=p->next)
    printf("\n%d\t",p->data);
    }
}

void deletenode(nptr h)
{
nptr temp,p;
int d;
printf("enter data item to delete\n");
scanf("%d",&d);
  if(h->next==NULL)
  {
      printf("empty list");
      return;
  }
  else
    {
    for(p=h;p->next!=NULL;p=p->next)
    {

    if(p->next->data==d)
    {
        p->next=p->next->next;
        printf("data deleted");
        return;
    }
    }
    printf("Element not found\n");
}
}
void findnode(nptr h)
{
    nptr temp,p;
    int x;
    printf("enter data item to search\n");
    scanf("%d",&x);
  if(h->next==NULL)
  {
      printf("empty list");
      return;
  }
  else
    {
    for(p=h;p->next!=NULL;p=p->next)
    {

    if(p->next->data==x)
    {
        printf("data found at position %d",p->next);
        return;
    }
    }
    printf("Element not found\n");
    return;
}

}
void insertnode(nptr h)
{
    nptr p,new;
    int i,x,n;
    printf("enter data item to insert\n");
    scanf("%d",&x);
```

```
        printf("enter list item\n");
        scanf("%d",&n);
         if(h->next==NULL)
      {
         printf("empty list\n");
          return;
      }
      else
        {
           new=(nptr)(malloc(sizeof(struct node)));
           new->data=x;
           printf("enter before 1, After 2\n");
          scanf("%d",&i);

        for(p=h;p->next!=NULL;p=p->next)
        {

        if(p->next->data==n)
        {

          if(i==1)
          {
            new->next=p->next;
            p->next=new;
            printf("inserted");
            return;
          }
          else if(i==2)
          {
            new->next=p->next->next;
            p->next->next=new;
            printf("inserted");
            return;

          }
        }
        }
        printf("Element not found\n");
        return;
    }

    }
void insertnodefirst(nptr h)
{
    nptr new;
    int x;
    if(h==NULL)
    {
      h=(nptr)(malloc(sizeof(struct node)));
      h->next=NULL;
    }
    printf("enter data\n");
    scanf("%d",&x);
    new=(nptr)(malloc(sizeof(struct node)));
    new->data=x;
    new->next=h->next;
    h->next=new;
    return;

}
void insertnodelast(nptr h)
{
    nptr new,p;
    int x;
    if(h==NULL)
    {
      printf("Empty list");
      return;
    }
    printf("enter data\n");
    scanf("%d",&x);
    new=(nptr)(malloc(sizeof(struct node)));
    new->data=x;
    new->next=NULL;
    for(p=h;p->next!=NULL;p=p->next);
    p->next=new;
    return;

}
```

Additional Operations for Chains

1.Inverting/Reverse of a List:

Algorithm

Circular Linked List

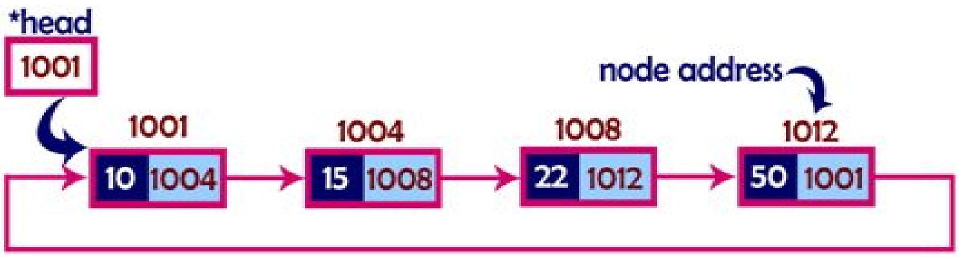
What is Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example



Operations for Circularly Linked Lists

In a circular linked list, we perform the following operations...

Insertion

Deletion

Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.

Step 2: Declare all the user defined functions.

Step 3: Define a Node structure with two members data and next

Step 4: Define a Node pointer 'head' and set it to NULL.

Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1: Create a new node with given value.

Step 2: Check whether list is Empty (head == NULL)

Step3: If it is Empty then, create head node.

```
h=(nptr)(malloc(sizeof(struct node)));

H→next=h;

Temp=h;
```

Step4: If it is Not Empty then, create new node.

```
new=(nptr)(malloc(sizeof(struct node)));

new->data=x;

 new->next=h->next;

 h->next=new;
```

Step5: Return h.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a newNode with given value.

```
 new=(nptr)(malloc(sizeof(struct node)));

new->data=x;

 new->next=h;
```

Step 2: If list is Not Empty then, define a node pointer temp and initialize with head.

Step 3: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to head).

Step 4: Set temp → next = new.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL).

Step 3: If it is Empty then, set head = newNode and newNode → next = head.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next == head).

Step 6: Set temp → next = newNode and newNode → next = head.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, set head = newNode and newNode → next = head.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether temp is reached to the last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7: If temp is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8: If temp is last node then set temp → next = newNode and newNode → next = head.

Step 8: If temp is not last node then set newNode → next = temp → next and temp → next = newNode.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with head.

Step 4: Check whether list is having only one node (temp1 → next == head)

Step 5: If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)

Step 6: If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head)

Step 7: Then set head = temp2 → next, temp1 → next = head and delete temp2.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Check whether list has only one Node (temp1 → next == head)

Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until temp1 reaches to the last node in the list. (until temp1 → next == head)

Step 7: Set temp2 → next = head and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node (temp1 → next == head)

Step 7: If list has only one node and that is the node to be deleted then set head = NULL and delete temp1 (free(temp1)).

Step 8: If list contains multiple nodes then check whether temp1 is the first node in the list (temp1 == head).

Step 9: If temp1 is the first node then set temp2 = head and keep moving temp2 to its next node until temp2 reaches to the last node. Then set head = head → next, temp2 → next = head and delete temp1.

Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next == head).

Step 11: If temp1 is last node then set temp2 → next = head and delete temp1 (free(temp1)).

Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5: Finally display temp → data with arrow pointing to head → data.

```
#include<stdio.h>
#include<conio.h>

void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();

struct Node
{
  int data;
  struct Node *next;
}*head = NULL;

void main()
{
  int choice1, choice2, value, location;
  clrscr();
  while(1)
  {
    printf("\n*********** MENU *************\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
    scanf("%d",&choice1);
    switch()
    {
      case 1: printf("Enter the value to be inserted: ");
                     scanf("%d",&value);
           while(1)
           {
                   printf("\nSelect from the following Inserting options\n");
                   printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter
your choice: ");
             scanf("%d",&choice2);
             switch(choice2)
             {
               case 1:       insertAtBeginning(value);
                             break;
               case 2:       insertAtEnd(value);
                             break;
               case 3:       printf("Enter the location after which you want to insert: ");
                             scanf("%d",&location);
                             insertAfter(value,location);
                             break;
               case 4:       goto EndSwitch;
               default: printf("\nPlease select correct Inserting option!!!\n");
             }
           }
      case 2: while(1)
           {
                   printf("\nSelect from the following Deleting options\n");
                   printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter
your choice: ");
             scanf("%d",&choice2);
             switch(choice2)
             {
               case 1:       deleteBeginning();
```

}

```
                                  break;
                 case 2:          deleteEnd();
                                  break;
                 case 3:          printf("Enter the Node value to be deleted: ");
                                  scanf("%d",&location);
                                  deleteSpecic(location);
                                  break;
                 case 4:          goto EndSwitch;
                 default: printf("\nPlease select correct Deleting option!!!\n");
               }
             }
             EndSwitch: break;
      case 3: display();
           break;
      case 4: exit(0);
      default: printf("\nPlease select correct option!!!");
    }
  }
}

void insertAtBeginning(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   if(head == NULL)
   {
     head = newNode;
     newNode -> next = head;
   }
   else
   {
     struct Node *temp = head;
     while(temp -> next != head)
        temp = temp -> next;
     newNode -> next = head;
     head = newNode;
     temp -> next = head;
   }
   printf("\nInsertion success!!!");
}
void insertAtEnd(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  if(head == NULL)
  {
    head = newNode;
    newNode -> next = head;
  }
  else
  {
    struct Node *temp = head;
    while(temp -> next != head)
      temp = temp -> next;
    temp -> next = newNode;
    newNode -> next = head;
  }
  printf("\nInsertion success!!!");
}
void insertAfter(int value, int location)
{
  struct Node *newNode;
```

```
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  if(head == NULL)
  {
    head = newNode;
    newNode -> next = head;
  }
  else
  {
    struct Node *temp = head;
    while(temp -> data != location)
    {
      if(temp -> next == head)
      {
        printf("Given node is not found in the list!!!");
        goto EndFunction;
      }
      else
      {
        temp = temp -> next;
      }
    }
    newNode -> next = temp -> next;
    temp -> next = newNode;
    printf("\nInsertion success!!!");
  }
  EndFunction:
}
void deleteBeginning()
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp = head;
    if(temp -> next == head)
    {
      head = NULL;
      free(temp);
    }
    else{
      head = head -> next;
      free(temp);
    }
    printf("\nDeletion success!!!");
  }
}
void deleteEnd()
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp1 = head, temp2;
    if(temp1 -> next == head)
    {
      head = NULL;
      free(temp1);
    }
    else{
      while(temp1 -> next != head){
        temp2 = temp1;
        temp1 = temp1 -> next;
      }
      temp2 -> next = head;
      free(temp1);
    }
    printf("\nDeletion success!!!");
  }
}
void deleteSpecific(int delValue)
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp1 = head, temp2;
    while(temp1 -> data != delValue)
    {
      if(temp1 -> next == head)
      {
        printf("\nGiven node is not found in the list!!!");
        goto FuctionEnd;
      }
      else
      {
        temp2 = temp1;
        temp1 = temp1 -> next;
      }
    }
    if(temp1 -> next == head){
      head = NULL;
      free(temp1);
    }
    else{
      if(temp1 == head)
      {
        temp2 = head;
        while(temp2 -> next != head)
          temp2 = temp2 -> next;
        head = head -> next;
        temp2 -> next = head;
        free(temp1);
      }
      else
      {
        if(temp1 -> next == head)
        {
          temp2 -> next = head;
        }
        else
        {
          temp2 -> next = temp1 -> next;
        }
        free(temp1);
      }
    }
    printf("\nDeletion success!!!");
  }
  FuctionEnd:
}
void display()
{
  if(head == NULL)
    printf("\nList is Empty!!!");
  else
  {
    struct Node *temp = head;
    printf("\nList elements are: \n");
    while(temp -> next != head)
    {
      printf("%d ---> ",temp -> data);
    }
    printf("%d ---> %d", temp -> data, head -> data);
  }
}
```

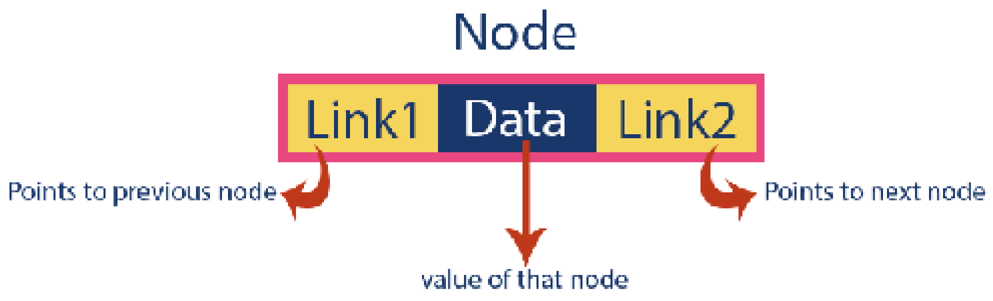
Double Linked List

What is Double Linked List?

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

Example

- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.

☀ In double linked list, the first node must be always pointed by head.
☀ Always the previous field of the head node must be NULL.
☀ Always the next field of the last node must be NULL.

Operations

In a double linked list, we perform the following operations...

Insertion

Deletion

Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1: Create a newNode with given value and newNode → previous as NULL.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode → next and newNode to head.

Step 4: If it is not Empty then, assign head to newNode → next and newNode to head.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty, then assign NULL to newNode → previous and newNode to head.

Step 4: If it is not Empty, then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6: Assign newNode to temp → next and temp to newNode → previous.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode → previous & newNode →
next and newNode to head.

Step 4: If it is not Empty then, define two node pointers temp1 & temp2 and
initialize temp1 with head.

Step 5: Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.

Step 7: Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → nextand newNode to temp2 → previous.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list is having only one node (temp → previous is equal to temp → next)

Step 5: If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 6: If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list has only one Node (temp → previous and temp → next both are NULL)

Step 5: If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7: Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

Step 5: If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node which is to be deleted then
set head to NULL and delete temp (free(temp)).

Step 8: If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).

Step 9: If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.

Step 10: If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).

Step11: If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp(free(temp)).

Step 12: If temp is not the first node and not the last node, then
set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Display 'NULL <--- '.

Step 5: Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node

Step 6: Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Polynomial Representation

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + …. + jx+ k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

<u>An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:</u>

- one is the coefficient
- other is the exponent

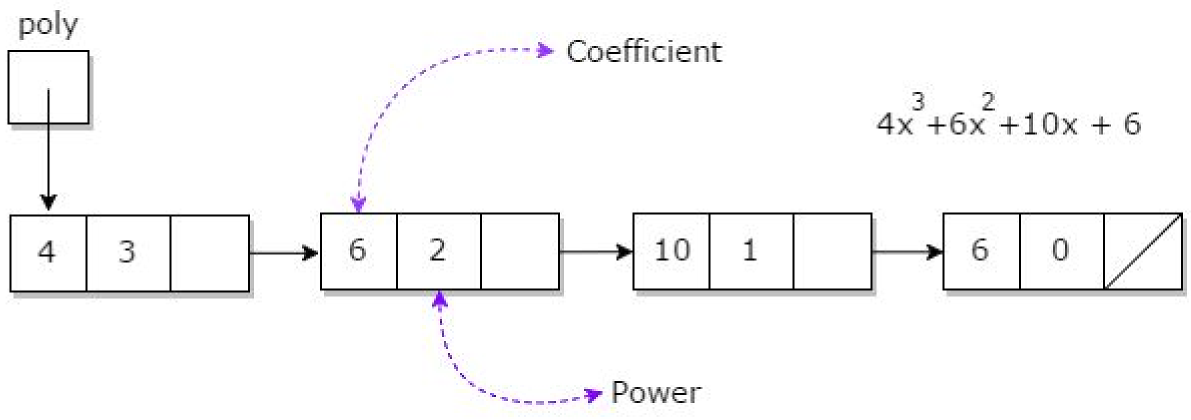
<u>Example:</u>

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

<u>Points to keep in Mind while working with Polynomials:</u>

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself

- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Representation of Polynomial Using Linked List

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

- The exponent part
- The coefficient part

```
typedef struct node *nptr;
struct node
{
   int coef;
   int expo;
   nptr next;
};
```

Creation of polynomial list and Addition of Polynomial Lists:

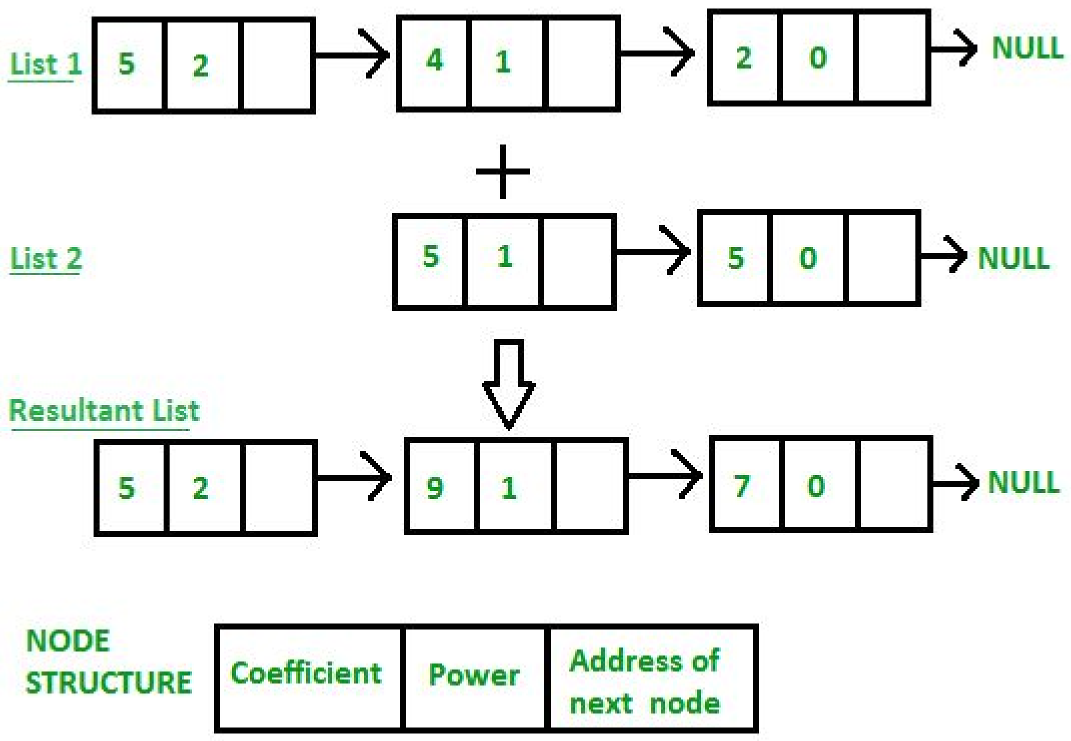
Algorithm:Creation of polynomial list for n terms

1. Create head node.
2. Read the number of terms in an expression (read n)
3. Repeat the following steps for n terms.

4. Create new node and add new node to list.

```
new=(nptr)(malloc(sizeof(struct node)));
        new->coef=co;
        new->expo=pow;
        new->next=temp->next;
        temp->next=new;
        temp=new;
```

Addition of two polynomial lists



Algorithm:

Let p and q be the two polynomials represented by linked lists.
Let r represent result list of the addition of two polynomials

```
1. while p and q are not null, repeat step 2.
2. If powers of the two terms are equal
        then insert the sum of the terms into the sum Polynomial
        if(p ->expo==q->expo)
        {
        r->co-eff=p->co-eff+q->co-eff
        r->expo=p->expo
        p=p->next
        q=q->next
        }
    Else if the expo of the polynomial p> expo of polynomial q
        Then insert the term from polynomial p into sum polynomial
         Advance p
        Else if(p ->expo>q->expo)
        {
        r->co-eff=p->co-eff
        r->expo=p->expo
        p=p->next
        }

    Else insert the term from q polynomial into sum polynomial
        Advance q
        {
        r->co-eff=q->co-eff
        r->expo=q->expo
        q=q->next
        }
```

3. copy the remaining terms from the non empty polynomial into the sum polynomial. The third step of the algorithm is to be processed till the end of the polynomials has not been reached.

```
        While(p!=NULL)
        {
        r->co-eff=p->co-eff
        r->expo=p->expo
        p=p->next
        }
        While(q!=NULL)
        {
        r->co-eff=q->co-eff
        r->expo=q->expo
        q=q->next
        }
```

Addition of two polynomials expressions:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node*nptr;
struct node
{
   int coef;
   int expo;
   nptr next;
};
nptr createplist();
void displaylist(nptr);
nptr polyadd(nptr,nptr);
int main()
{
   nptr p1,p2,p3;
   p1=createplist();
   displaylist(p1);
  p2=createplist();
  displaylist(p2);
  displaylist(p1);
  p3=polyadd(p1,p2);
  printf("polynomial list after addition\n");
  display(p3);
}
nptr createplist()
{
   nptr h,new,temp;
   int co,pow,n,i;
   h=(nptr)(malloc(sizeof (struct node)));
   h->next=NULL;
   temp=h;
   printf("\n\nh is %d",h);
   printf(“enter terms number n\n”);
   scanf(“%d”,&n);
   for(i=1;i<=n;i++)
   {
    printf(“enter coeff and power\n”);
    scanf(“%d%d”,&co,&pow);
    new=(nptr)(malloc(sizeof(struct node)));
    new->coef=co;
   new->expo=pow;
   new->next=temp->next;
   temp->next=new;
   temp=new;
   }
return h;
}
void display(nptr h)
{
   nptr p;
 if(h->next==NULL)
 {
  printf(“empty list”);
  return;
 }
 else
 {
 for(p=h->next;p!=NULL;p=p->next)
 {
  printf(“\n coef %d\t”,p->coef);
  printf(“expo %d\t”,p->expo);
 }
 }
 nptr polyadd(nptr p1,nptr p2)
```

```
{
nptr p3,p,q,r,new;
int c,pow;
p3=(nptr)(malloc(sizeof(struct node)));
p3->next=NULL;
r=p3;p=p1->next;q=p2->next;
while((p!=NULL)&&(q!=NULL))
{
  if(p->expo>q->expo)
{
 c=p->coef;
 pow=p->expo;
 p=p->next;
}
else if(q->expo>p->expo)
{
  c=q->coef;
  pow=q->expo;
  q=q->next;
}
else if(p->expo=q->expo)
{
  c=p->coef+q->coef;
  pow=p->expo;
  p=p->next;
  q=q->next;
}
new=(nptr)(malloc(sizeof(struct node)));
new->coeff=c;
new->expo=pow;
new->next=NULL;
r->next=new;
```

```
 r=new;
 }
 while(p!=NULL)
 {
   new=(nptr)(malloc(sizeof(struct node)));
   new->coef=p->coeff;
  new->expo=p->expo;
   new->next=NULL;
   r->next=new;
   r=new;
   p=p->next;
 }
while(q!=NULL)
{
  new=(nptr)(malloc(sizeof(struct node)));
  new->coeff=q->coeff;
  new->expo=q->expo;
  new->next=NULL;
  r->next=new;
  r=new;
 q=q->next;
}
return p3;
}
```

sparse matrix:

A matrix is an argument of m.n elements arranged as m rows and n columns.

The sparse matrix is a matrix with zeros as the dominating elements.The matrix which has very few non-zero entries is known as a sparse matrix.

Sparse Matrix

What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

Sparse Matrix Representations

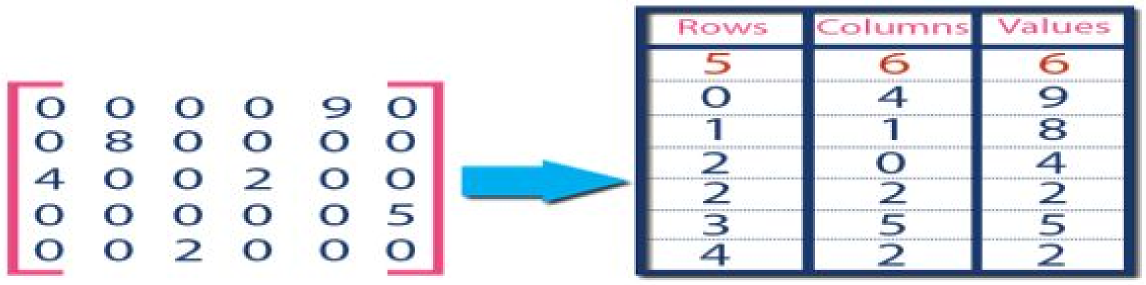
A sparse matrix can be represented by using TWO representations, those are as follows...

Triplet Representation

Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

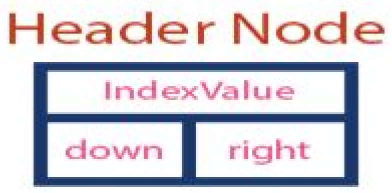
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

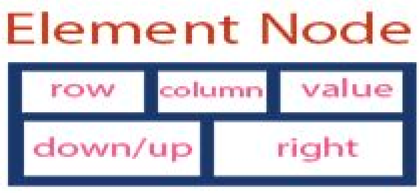
For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

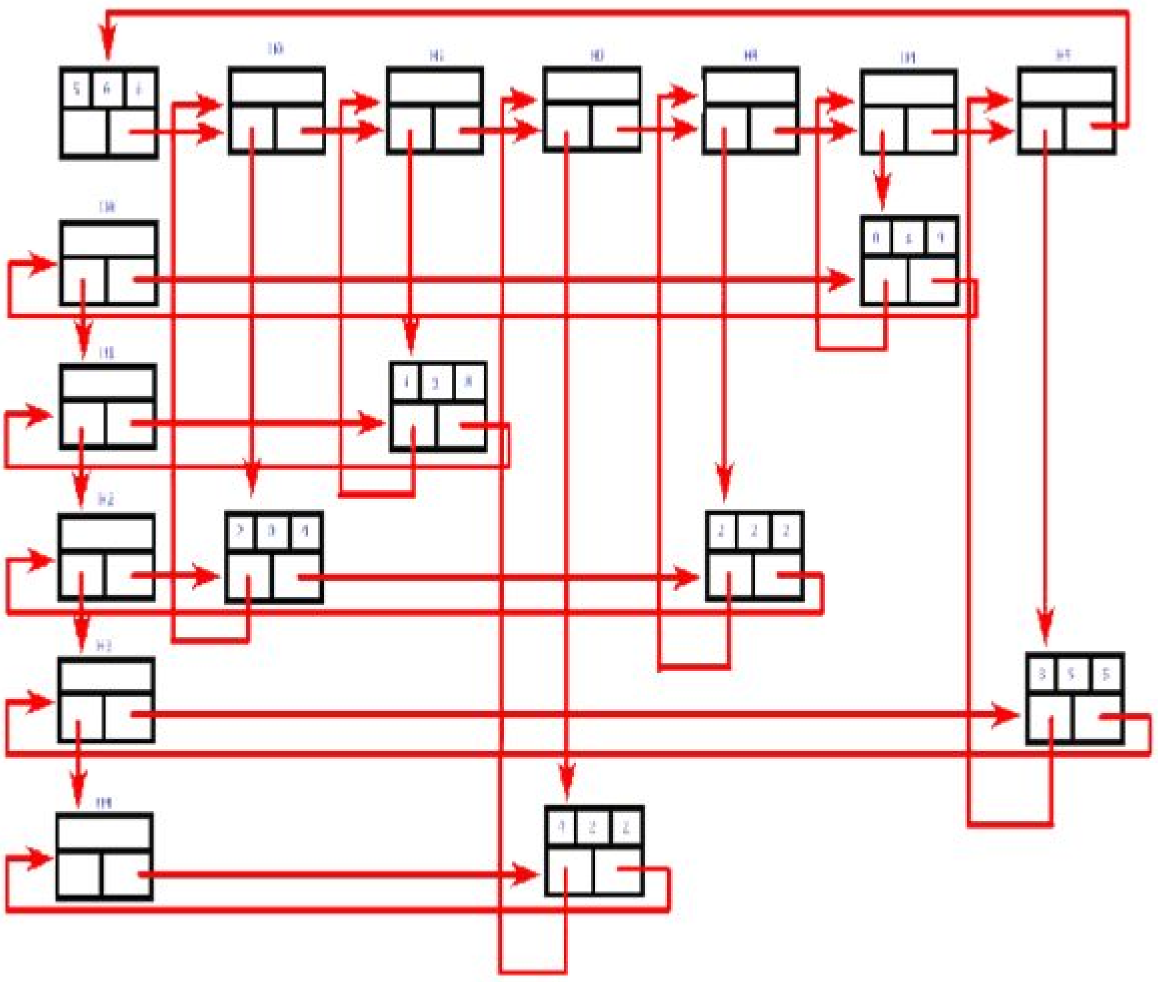
Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the image…





Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image…



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.