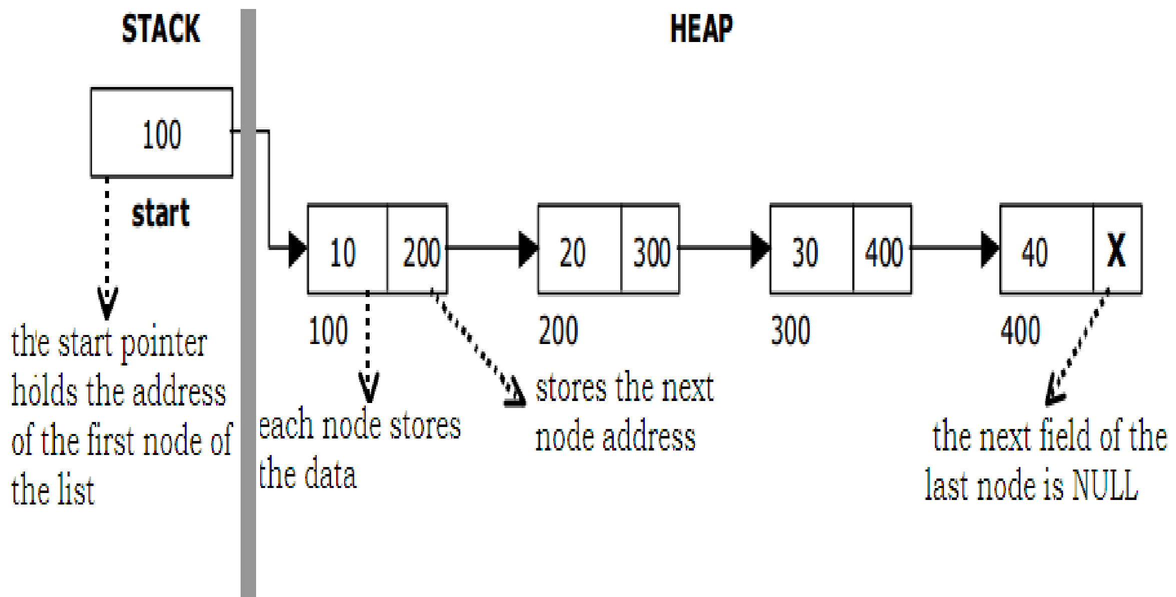
UNIT-3
LINKED LIST

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together. Each node contains two fields-a "**data**" field to store element, and a "**next**" field which is a pointer used to connect to the next node. Each node is allocated in the heap using **new()** and it is explicitly de-allocated using **delete()**. The front of the list is a pointer to the **"start"** node. The single linked list is called as linear list or chain.



Single Linked List Representation

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node and so on. The last node in the list has its next field set to NULL to mark the end of the list.

"Asinglylinkedlistisalinkedlistinwhicheachnodecontainsonlyonelinkpointingtothenextnodeinthelist".

AbstractDataType SlinkedList
{
instances:
finite collection of zero or more elements linked by pointers

operations:

Count( ): Count the number of elements in the list.
Addatbeg(x): Add x to the beginning of the list.
Addatend(x): Add x at the end of the list.
Insert(k, x):Insert xjust after kth element.
Delete(k): Delete the kth element.
Search(x): Return the position of x in the list otherwise
return-1,if not found
Display( ): Display all elements of the list

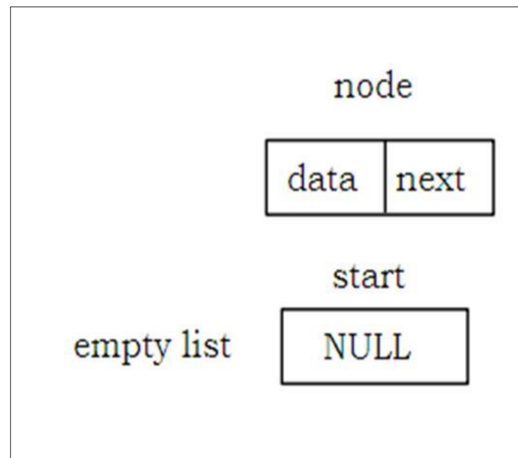
}

Implementation of SingleLinkedList

Before writing the code to build the list, we need to create a **start** node, used to create and access other nodes in the linked list.

- ✓ Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- ✓ Initialize the start pointer to be NULL.

```
Struct slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;
node *start = NULL;
```



Basicoperationperformedonsinglelinkedlist

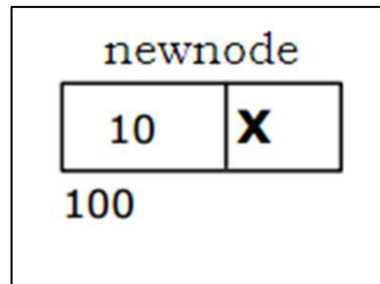
The different operations performed on the single linked list are listed as follows.

- ✓ Creation
- ✓ Insertion
- ✓ Deletion
- ✓ Traversing
- ✓ Display

Creating a node for SingleLinkedList

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the new () function. The function getnode(), is used for creating a node, after allocating memory for the node, the Information for the node at a path as to be read from the user and set next field to NULL and finally return the node.

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        cout<< Enter data;
        cin>>newnode-> data
        newnode -> next = NULL;
        return newnode;
}
```



Creating a Singly Linked List with„ n" number of nodes

The following steps are to befollowed to create „n" number of nodes.

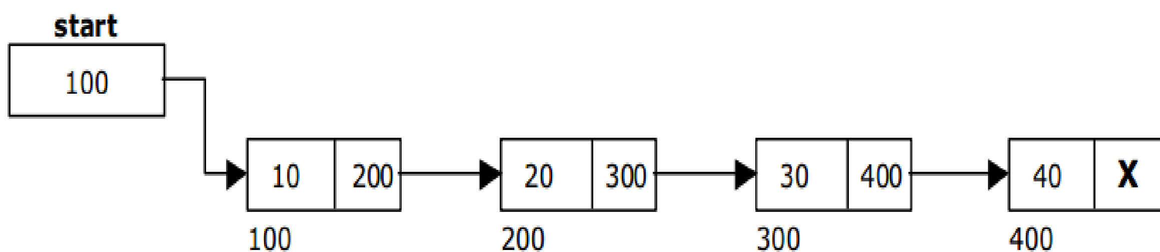
1. Get the new node using getnode().

    newnode = getnode();

2. If the list is empty, assign new node as start.

    start = newnode;

3. If the list is not empty, follow the steps given below.
    - ✓ The next field of the newnode is made to point the first node (i.e. start node) in the list by assigning the address of the first node.
    - ✓ The start pointer is made to point the newnode by assigning the address of the new node.
4. Repeat the above steps „n" times.



Single Linked List with 4 nodes

The function createlist(), is used to create „n" number of nodes

```
Void createlist(int n)
{
        inti;
        node *newnode;
        node *temp;
        for(i = 0; i< n ; i++) {
                newnode = getnode();
                if(start = = NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                                temp -> next = newnode;
                        }
        }
}
```

INSERTION OF A NODE

One of the most import operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL.

The new node can then be inserted at three different places namely:

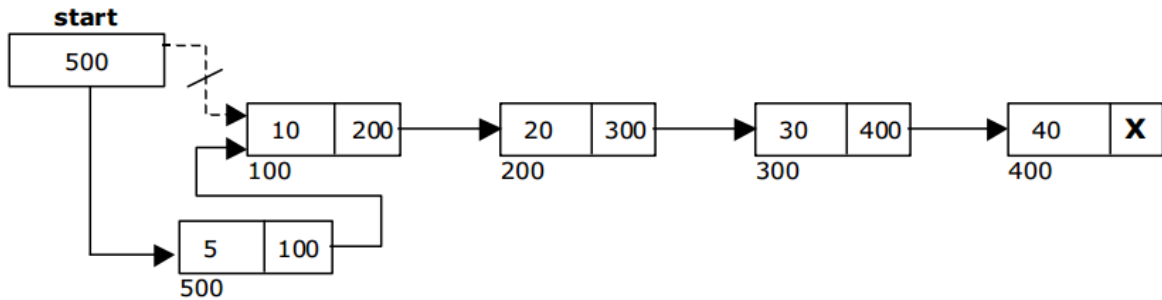
- ✓ Inserting a node at the beginning.
- ✓ Inserting a node at the end.
- ✓ Inserting a node at specified position.

INSERTING A NODE AT THE BEGINNING

The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using getnode() then new node = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty, follow the steps given below:

```
newnode -> next = start;
start = newnode;
```



Inserting a node at the begining of the list

The function insert_at_beg(),is used for inserting a node at the beginning.

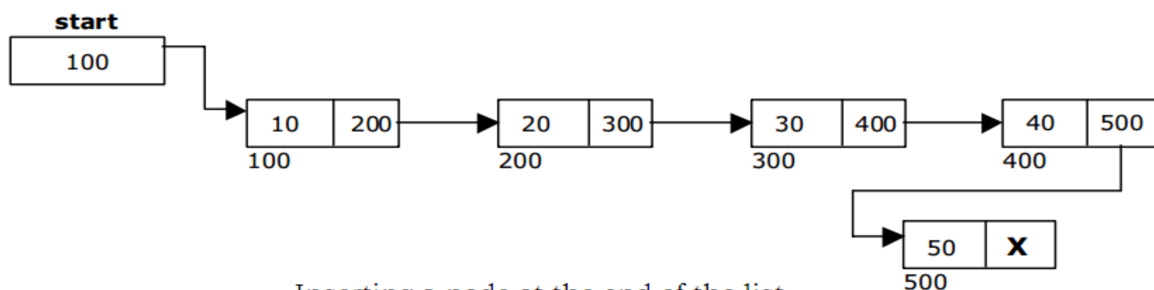
```
Void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode-> next = start;
                start = newnode;
        }
}
```

INSERTING A NODE AT THE END

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```



Inserting a node at the end of the list

The function insert_at_end(), is used for inserting a node at the end.

```
Void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
        }
}
```

INSERTING A NODE AT SPECIFIED POSITION

The following steps are followed, to insert a new node in an intermediate position in the list:

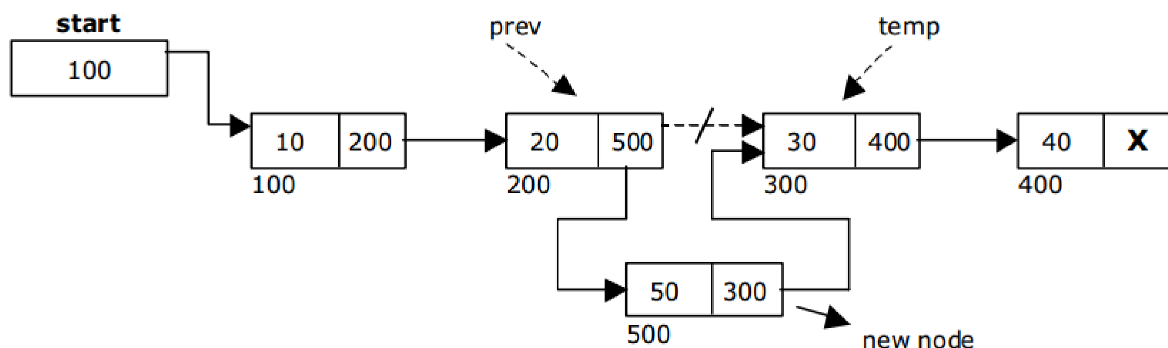
1. Get the new node using getnode() then newnode = getnode();

2.Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

3.Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer up to the specified position followed by prev pointer.

4. After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode-> next = temp;
```



Inserting a node at specified position

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
Void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        cout<< Enter the position;
        cin>>pos;
        nodectr = countnode(start);
        if(pos> 1 &&pos<nodectr) {
                temp = prev = start;
                while(ctr<pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                cout<<pos; }
}
```

DELETION OF A NODE

Another operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places.

- ✓ Deleting at beginning.
- ✓ Deleting a node at the end.
- ✓ Deleting a node at specified position.

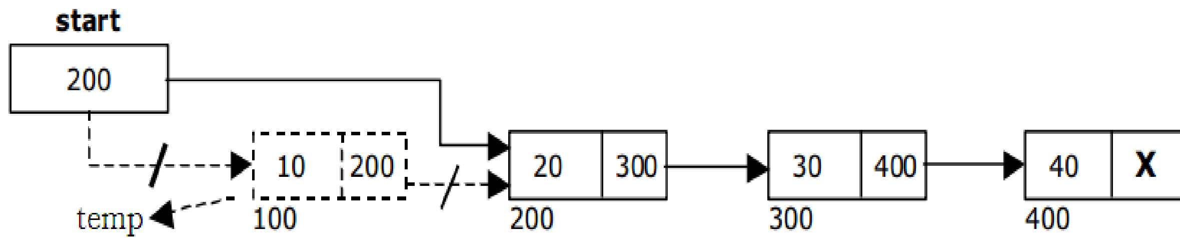
DELETING A NODE AT THE BEGINNING

The following steps are followed, to delete a node at the beginning of the list:

1.If list is empty then display „EmptyList" message.2.

If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
delete temp;
```



deleting a node at the begining

The function delete_at_beg(), is used for deleting the first node in the list.

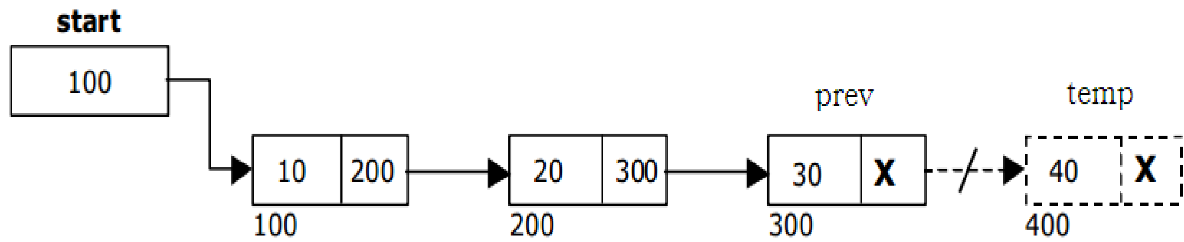
```
Void delete_at_beg()
{
	node *temp;
	if(start == NULL)
	{
		cout<< No nodes are exist;
		return ;
	}
	else
	{
		temp = start;
		start = temp -> next;
		delete temp;
		cout<<Node deleted;
	}
}
```

DELETING A NODE AT THE END

The following steps are followed todelete a node at the endof the list:

1. If list is empty then display „EmptyList" message.
2. If the list is not empty, follow the steps given below:

```
	temp = prev = start;
	while(temp -> next != NULL)
	{
		prev = temp;
		temp = temp -> next;
	}
	prev -> next = NULL;
	delete temp;
```



deleting a node at the end

The function delete_at_last(),is used for deleting the last node in the list.

```
Void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                cout<<Empty List;
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                delete temp;
                cout<<Node deleted;
        }
}
```

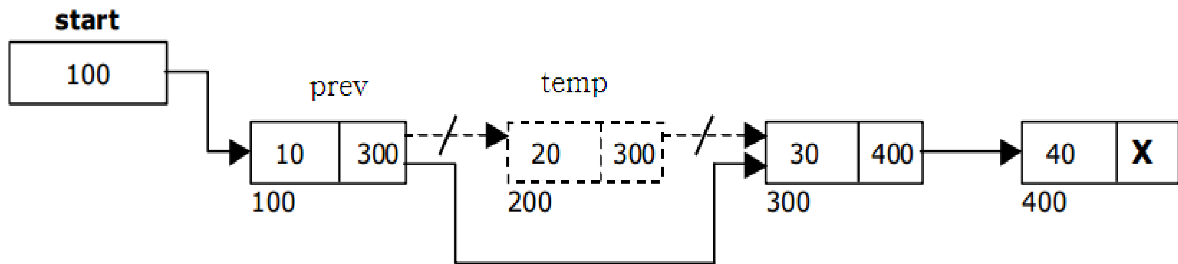
DELETING A NODE AT SPECIFIED POSITION

The following steps are followed, to delete node from the specified position in the list.

1. If list is empty then display „EmptyList" message
2. If the list is not empty, follow the steps given below.

```
        if(pos> 1 &&pos<nodectr) {
                temp = prev = start;
                ctr = 1;
                while(ctr<pos) {
```

```
            prev = temp;
            temp = temp -> next;
            ctr++;
        }
        prev -> next = temp -> next;
        delete temp;
        cout<<node deleted;
    }
```



deleting a node at the specified position

The function delete_at_mid(),is used for deleting the specified position node in the list.

```
Void delete_at_mid()
{
        intctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                cout<<Empty List;
                return ;
        }
        else
        {
                cout<<Enter position of node to delete;
                cin>>pos;
                nodectr = countnode(start);
                if(pos>nodectr)
                {
                        cout<<This node does not exist;
                }
                if(pos> 1 &&pos<nodectr) {
                        temp = prev = start;
                        while(ctr<pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr ++;
                        }
                        prev -> next = temp -> next;
```

```
                    delete temp;
                    cout<<Node deleted;
               }
               else
               {
                    cout<<Invalid position;
                    getch();
               }
          }
     }
```

TRAVERSAL AND DISPLAYING A LIST (LEFT TO RIGHT)

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves thefollowing steps.

1. Assign the address of start pointer to a temp pointer.

2. Display the information from the data field of each node.

The function *traverse* () is used for traversing and displaying the information stored in the list from left to right.

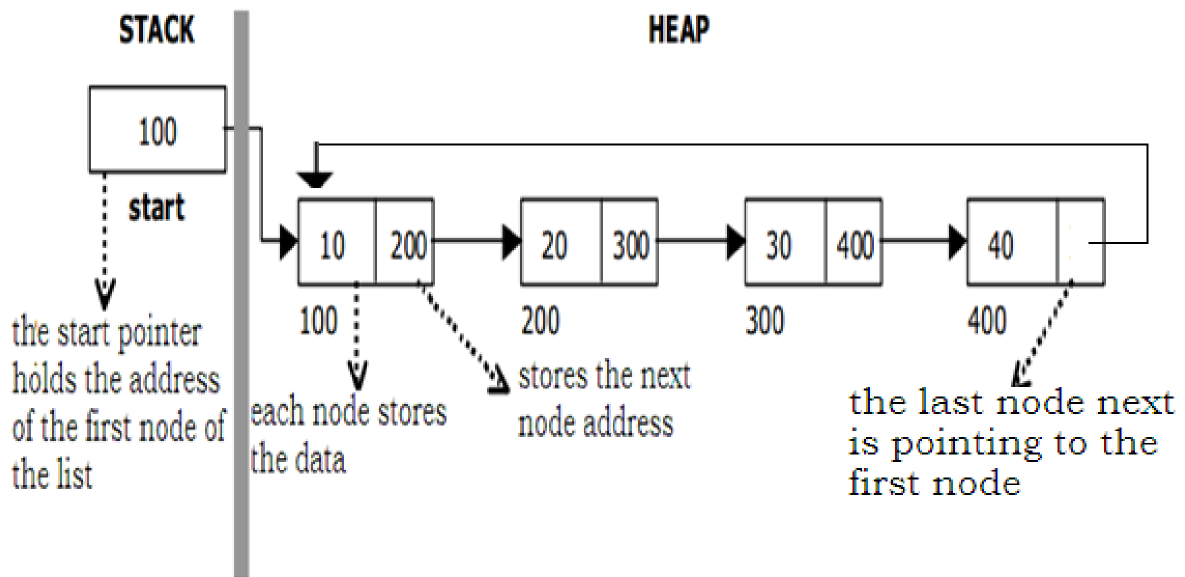
```
void traverse()
{
     node *temp;
     temp = start;
     cout<<The contents of List (Left to Right);
     if(start == NULL )
          cout<<Empty List;
     else
     {
          while (temp != NULL)
          {
               cout<<temp -> data;
               temp = temp -> next;
          }
     }
     cout<<X;
}
```

CIRCULAR LISTS

Circular linked list is a linked list which consists of collection of nodes each of which has two parts, namely the data part and the next part. The data part holds the value of the element and the next part has the address of the next node. The last node of list has the next pointing to the first node thus making the circular traversal possible in the list.

It is just a single linked list in which the next field of the last node points back to the address of the first node. A circular linked list has no beginning

and no end. In circular linked list no null pointers are used, hence all pointers contain valid address.



Circular Linked List Reresentation

Abstract DataType CLinkedList

{

Instances:

Finite collection of zero or more elements linked by pointers

operations:

Count( ): Count the number of elements in the list.
Addatbeg(x): Add x to the beginning of the list.
Addatend(x): Add x at the end of the list.
Insert(k, x):Insert x just after kth element.
Delete(k): Delete the kth element.
Search(x): Return the position of x in the list otherwise return -1 if not found
Display( ): Display all elements of the list

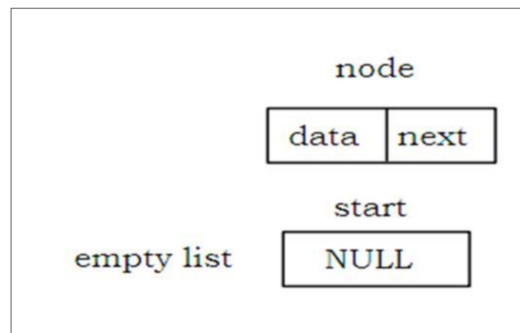
}

Implementation of Circular Linked List

Before writing the code to build the list, we need to create a **start** node, used to create and access othernodes in the linked list.

- ✓ Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- ✓ Initialize the start pointer to be NULL.

```
struct clinklist
{
        int data;
        struct clinklist* next;
};
typedef struct clinklist node;
node *start = NULL;
```



Basic operation performed on single linked list

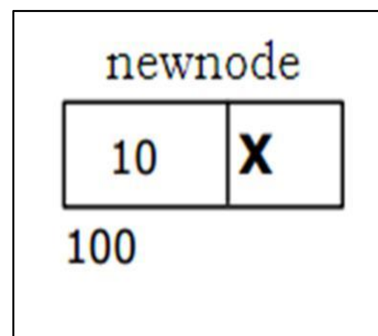
The different operations performed on the circular linked list are listed as follows.

- ✓ Creation
- ✓ Insertion
- ✓ Deletion
- ✓ Traversing
- ✓ Display

Creating a node for Circular Linked List

Creating a circular linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the new() function. The function getnode(),is used for creating a node, after allocating memory for the node, the information for the node data part has to be read from the user and set next field to NULL and finally return the node.

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        cout<< Enter data;
        cin>>newnode-> data
        newnode -> next = NULL;
        return newnode;
}
```



Creating a Circular Linked List with „n" number of nodes

The following steps are to befollowed to create „n" number of nodes.

1. Get the new node using getnode().

    newnode = getnode();

2. If the list is empty, assign new node as start.

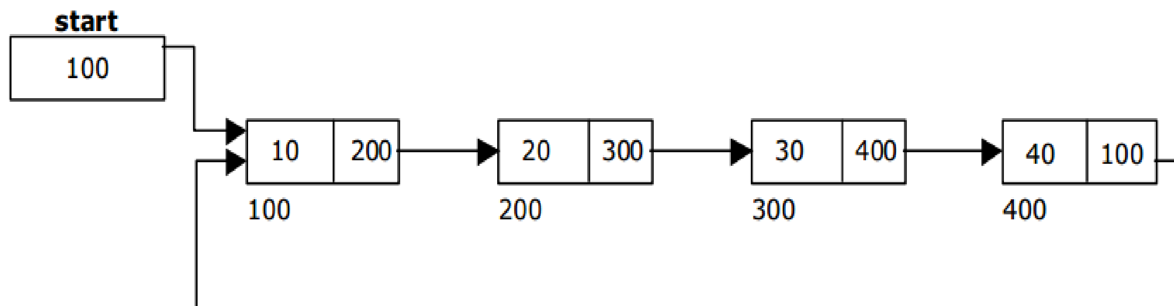
    start = newnode;

3. If the list is not empty, follow the steps given below.

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

4. Repeat the above steps „n"

times. 5. newnode -> next = start;



Circular Linked List with 4 nodes

The function createlist(), is used to create „n" number of nodes

```
Void createlist(int n)
{
        inti;
        node *newnode;
        node *temp;
        for(i = 0; i< n ; i++) {
                newnode = getnode();
                if(start = = NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                                temp -> next = newnode;
```

```
                }
                newnode -> next = start;
        }
}
```

INSERTING A NODE AT THE BEGINNING

The following steps are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode ().

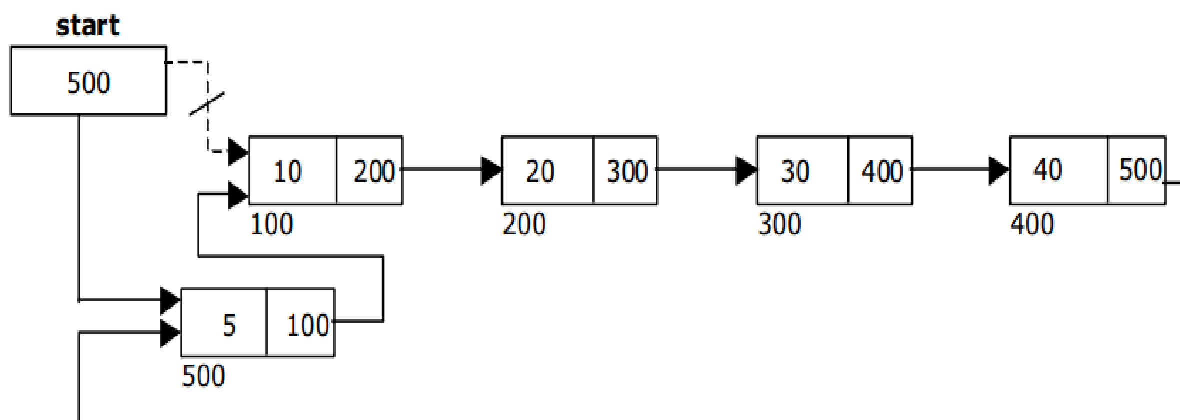
```
newnode = getnode();
```

2. If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

3. If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next != start)
        last = last -> next;
newnode -> next = start;
start = newnode;
last -> next = start;
```



Inserting a node at the beginning

INSERTINGANODEATTHEEND

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode().

```
newnode = getnode();
```

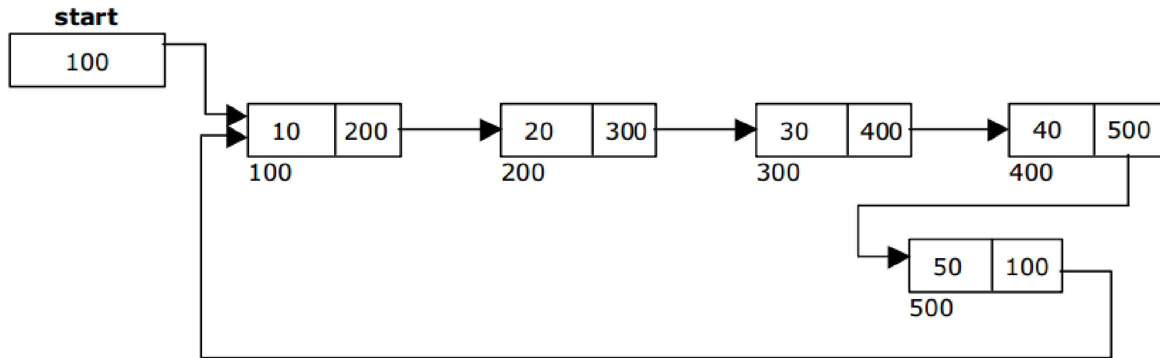
2. If the list is empty, assign new node as start.

```
start = newnode;
```

```
newnode -> next = start;
```

3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != start)
        temp = temp -> next;
temp -> next = newnode;
newnode -> next = start;
```



Inserting a node at the end

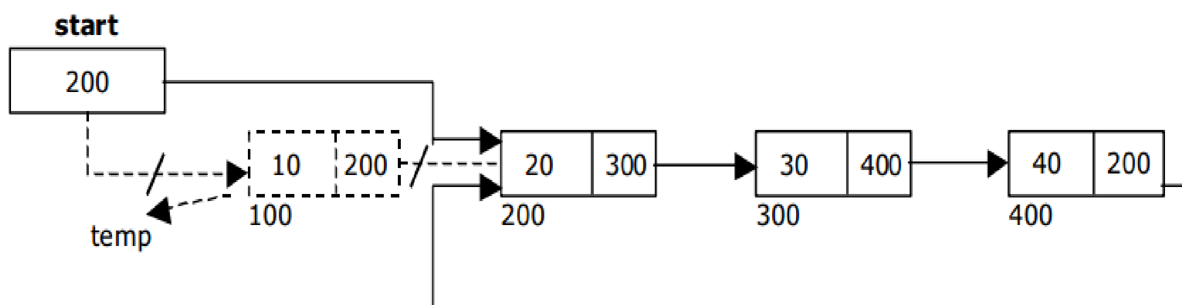
DELETING A NODE AT THE BEGINNING

The following steps are followed, to delete a node at the beginningof the list:

1. If the list is empty, display a message „EmptyList" .
2. If the list is not empty, follow the steps given below:

```
last = temp = start;
while(last -> next != start)
        last = last -> next;
start = start -> next;last
-> next = start;
```

3. After deleting the node, if the list is empty then**start=NULL.**



Deleting a node at beginning

DELETING A NODE AT THE END

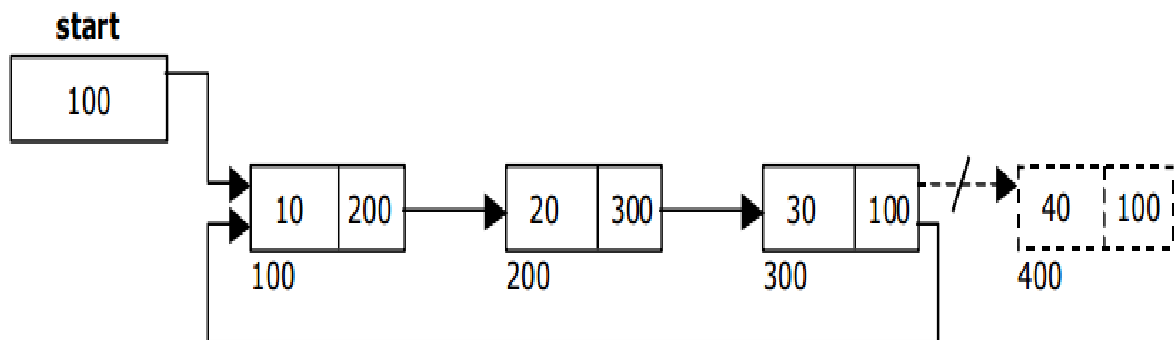
The following steps are followed todelete a node at the end of the list:

1. If the list is empty, display a message „EmptyList" .

2. If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

4. After deleting the node, if the list is empty then**start=NULL**



Deleting a node at the end.

TRAVERSING A CIRCULAR SINGLE LINKED LIST FROM LEFT TO RIGHT

The following steps are followed, totraverse a list from left to right:

1. If list is empty then display „EmptyList" message.2.

If the list is not empty, follow the steps given below:

```
temp = start;
do
{
        cout<<temp -> data;
        temp = temp -> next;
} while(temp != start);.
```

AVAILABLE SPACE LISTS

The collection of free nodes present in the memory is called list of available space or avail list or free poolers to rage pooler simply avail. If we want to insert a node in to the linked list we take a node from this avail list. If a node is deleted then that node is added to the avail list. Here AVAIL is a pointer that points to the first node of the free list.

The function getnode(),like new()in C++,returns a pointer to a new node taken from the available space list and also removes this node from the available space list. It is defined as

if(AVAIL ==NULL)

then empty list
return

else

x = AVAIL;
AVAIL = AVAIL-> next;
return x

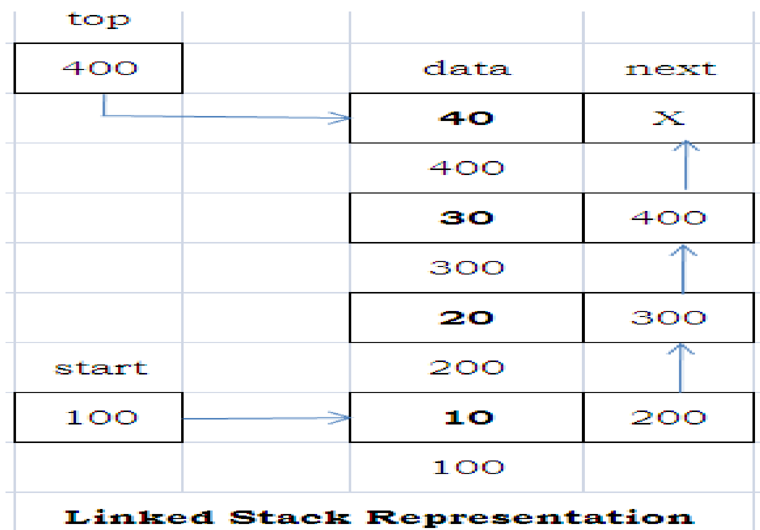
The function release(),like delete()in C++,causes the node pointed to by x to be returned to the available space list. It is defined as

x -> next = AVAIL;
AVAIL = x;

LINKED STACKS

A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is known as top of stack. When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop.

Stack is also called as Last-In-First-Out(LIFO)list which means that the last element that is inserted will be the first element to be removed from the stack. Stack can be implemented using linked list and the same operations can be performed at the end of the list using top pointer.



Linked Stack Representation

Push operation

Create a temporary node and store the value of x in the data part of the node. Now make next part of temp point to Top and then top point to temp. That will make the newnode as the topmost element in the stack.

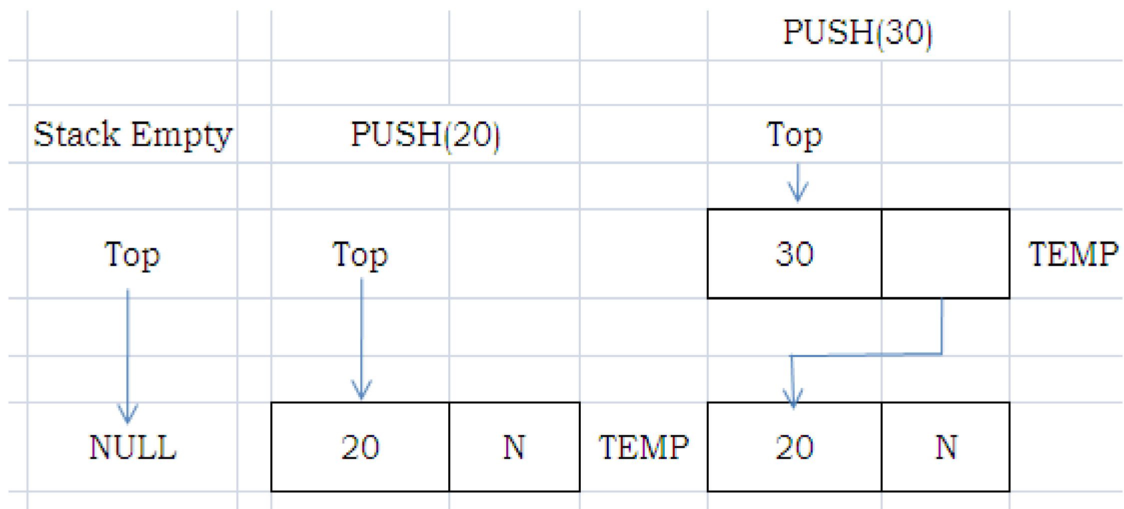
Algorithm for PUSH

1.temp -> data = x

2. temp -> next = top

3. top = temp

4. exit

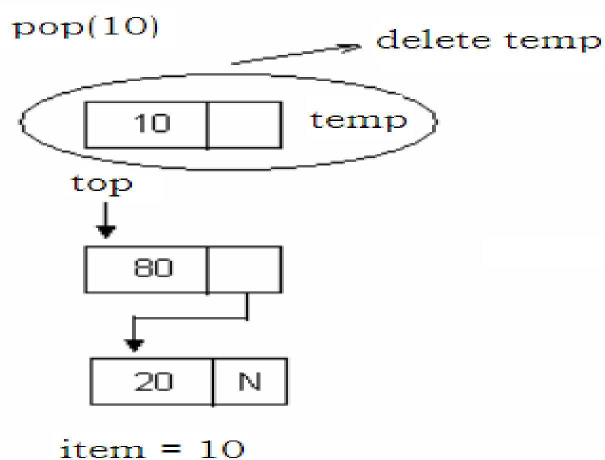


Pop Operations

The data in the top most node of the stack is first stored in a variable called item. Then a temporary pointer is created to point to top.The top is now safely moved to the next node below it in the stack. Temp node is deleted and the item is returned.

Algorithm for POP Operation

```
1. if top = NULL
        stack empty
        return

   else
        x = top -> data
        temp = top
        top = top -> next
        delete temp
        return x

2. exit
```



C++program to illustrate about stacks using linked lists

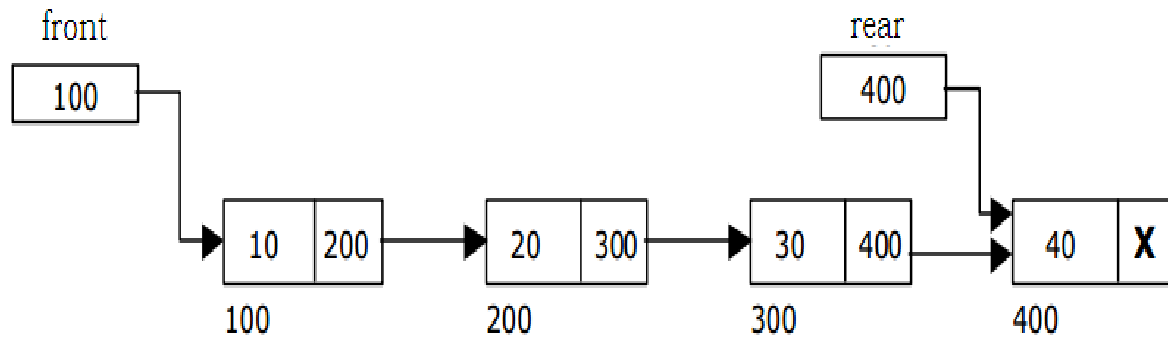
```
#include <iostream.h>
#include <conio.h>

class stack
{
	private:
		struct node
		{
			int data;
			node *link;
		};
		node *top;
	public:
		stack();
		~stack();
		void push(int x);
		int pop();
		void display();
	};
	stack::stack()
	{
			top=NULL;
	}
	stack::~stack()
	{
		node *temp;while
		(top!=NULL) {
			temp=top->link;
			delete top;
			top=temp;
		}
	}
	void stack::push(int x)
	{
		node *temp;
		temp=new node;
		temp->data=x;
		temp->link=top;
		top=temp;
	}
	int stack::pop()
	{
		if (top==NULL)
		{
			cout<<"\nStack is empty!";
			return NULL;
```

```
		}
		node *temp=top;
		int item=temp->data;
		top=temp->link;
		delete temp;
		return item;
	}
	void stack::display()
	{
		node *temp=top;
		while (temp!=NULL)
		{
			cout<<"\n"<<temp->data;
			temp=temp->link;
		}
	}
void main()
{
	clrscr();
	stack s;
	int n;
	s.push(10);
	s.push(20);
	s.push(30);
	s.push(40);
	s.display();
	n=s.pop();
	cout<<"\nPopped item:"<<n;
	n=s.pop();
	cout<<"\nPopped item:"<<n;
	s.display();
	getch();
}
```

LINKED QUEUES

Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end. The end at which the deletion of an element take place is called front, and the end at which insertion of a new element can take place is called rear. The deletion or insertion of elements can take place only at the front or rear end called dequeue and enqueue.

The first element that gets added in to the queue is the first one to get removed from the queue. Hence the queue is referred to as First-In-First-Out list(FIFO).We can perform the similar operations on two ends of the list using two pointers front pointer and rear pointer.



Linked Queue Representation

Operations onQueues

Enqueue

In linked list representation of queue, the addition of new element to the queue takes place at the rear end. It is the normal operation of adding a node at the end of a list.

Algorithm for Enqueue (inserting an element)

```
If( front = NULL )then
        rear = front = temp
        return
end if

        rear - > next = temp
        rear = rear - > next
```

Dequeue operation

The delq() operation deletes the first element from the front end of the queue. Initially it is checked, if the queue is empty. If it is not empty, then return the value in the node pointed by front, and moves the front pointer to the next node.

Algorithm for Dequeue(deleting an element)

```
if (front = NULL)
        display  "Queue is empty"
        return
else
        while (front != NULL)

                temp = front

                front = front - > next

                delete temp

        end while

end if
```

C++program to illustrate about queues using linked lists

```
#include <iostream.h>
#include <conio.h>
class queue
{
	private:
		struct node
		{
			int data;
			node *link;
		};
		node *front, *rear;
	public:
		queue();
		~queue();
		void addq(int x);
		int delq();
		void display();
	};
	queue::queue()
	{
			front=rear=NULL;
	}
	queue::~queue()
	{
		node *temp;
		while (front!=NULL)
		{
			temp=front->link;
			delete front;
			front=temp;
		}
	}
	void queue::addq(int x)
	{
		node *temp;
		temp=new node;
		temp->data=x;
		temp->link=NULL;
		if (front==NULL)
			front=rear=temp;
		else
		{
			rear->link=temp;
			rear=temp;
		}
	}
```

```
        int queue::delq()
        {
                int item;
                node *temp;
                if (front==NULL)
                {
                        cout<<"Queue is empty!";
                        return NULL;
                 }
                        temp=front->link;
                        item=front->data;
                        delete front;
                        front=temp;return
                        item;
        }
        void queue::display()
        {
                 node *temp;
                 temp=front;
                while (temp!=NULL)
                {
                        cout<<temp->data<<"\t";
                        temp=temp->link;
                }
        }
void main()
{
        clrscr();
        queue q;
        q.addq(50);
        q.addq(40);
        q.addq(90);
        q.display();
        cout<<endl;
        int i=q.delq();
        cout<<endl;
        cout<<i<<" deleted!";
        cout<<endl;
        q.display();
        i=q.delq();
        cout<<endl;
        cout<<i<<" deleted!";
        cout<<endl;
        i=q.delq();
        cout<<i<<" deleted!";
        cout<<endl;
        i=q.delq();
        getch();
}
```

POLYNOMIALS

A polynomial is of the form

$$\sum_{i=0}^{n} c_i x^i$$

Where, ci is the coefficient of the I th term and n is the degree of the polynomial. Some examples are:

$5x^2 + 3x + 1$

$12x^3 + 4$

4x6 + 10x4 – 5x +3

$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$

$23x^9 + 18x^7 - 41x^6 + 163x^4 - 5x + 3$

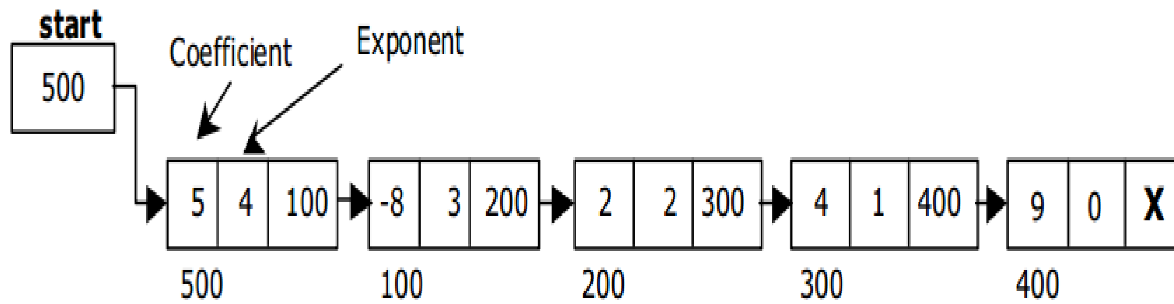
Polynomial as Abstract Data Type

The abstract data type of apolynomial is defined as follows:

```
class polynomial
{
        instances:p(x)=a0x0+a1x1+..........+anxn,is a set of ordered pairs of
        <ei,ai> where aiЄcoefficients and eiЄexponents, eiare integers>=0
        operations:
                polynomial();
                int operator!();
                coeffiecient coef(exponent e);
                exponent Leadexp();
                polynomial  Add(polynomial poly);
                polynomial  Multiply(polynomial poly);
};
```

REPRESENTATION OF POLYNOMIALS

It is not necessary to write term so if the polynomials in decreasing order of degree. In other words the two polynomials 1+x and x+1 are equivalent. The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures . A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$



Single Linked List for the polynomial $F(x) = 5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$

Advantages

- ✓ Save space
- ✓ Easy to maintain
- ✓ Do not need to allocate memory size initially

Disadvantages

- ✓ It is difficult to backup to the start of the list
- ✓ It is not possible to jump to the beginning of the list from the end of the list

POLYNOMIAL ADDITION

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials then we add the coefficients otherwise we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

- ✓ Read two polynomials.
- ✓ Add them.
- ✓ Display the resultant polynomial.

C++ program to illustrate about adding polynomials using linked lists

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>

struct poly
{
	int coef;
	int pow;
```

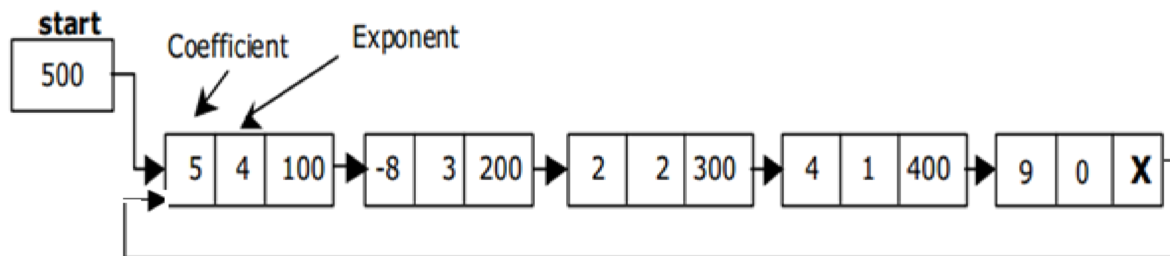
```
        poly *next;
};

class add2poly()
{
                poly *poly1, *poly2, *poly3;
        public:
                add2poly()
                {
                        poly1=poly2=poly3=NULL;
                }
                void addpoly();
                void display();
};
void add2poly :: addpoly()
{
        inti, p;
        poly *new1 = NULL, *end = NULL;
        cout<<"enter the highest degree power for x";
        cin>>p;
        cout<<"first polynomial";
        for(i=p; i>=0; i--)
        {
                new1 = new poly;
                 new1 ->pow = p;
                cout<<"enter the coeffients for the degree"<<i;
                cin>>new1 ->coef;
                new1 -> next = NULL;
                if(poly1 == NULL)
                        poly1 = new;
                else
                        end -> next = new1;
                        end = new;
        }
        cout<<"second polynomial";
        end = NULL;
        for(i=p; i>=0; i--)
        {
                new1 = new poly;
                 new1 ->pow = p;
                cout<<"enter the coeffients for the degree"<<i;
                cin>>new1 ->coef;
                new1 -> next = NULL;
                if(poly2 == NULL)
                        poly2 = new;
                else
                        end -> next = new1;
                        end = new;
        }
```

```
	poly *p1 = poly1, *p2 = poly2;
	end = NULL;
	while(*p1!=NULL&& *p2!=NULL) {
		if(p1 -> pow == p2 -> pow)
		{
			new1 = new poly;
			new1 ->pow = p--;
			new1 ->coef = p1 ->coef + p2->coef;
			new1 -> next = NULL;
			if(poly3 ==NULL)
				poly3 = new1;
			else
				end -> next = new1;
			end = new;
		}
		p1 = p1 -> next;
		p2 = p2 -> next;
	}
}
void add2poly :: display()
{
	poly *t = poly3;
	cout<<"after addition";
	while(t! = NULL)
	{
		cout.setf(ios :: showpos);
		cout<<t ->coef;
		cout.unsetf(ios :: showpos);
		cout<<"x"<<t -> pow;
		t = t -> next;
	}
}
main()
{
	add2poly obj;
	obj.addpoly();
	obj.display();
}
```

CIRCULARLIST REPRESENTATION OF POLYNOMIALS

A single linked list in which last node next has null is called a chain. It is possible to free all nodes of a polynomial more efficiently then the list structure is modified such that the last node next points to the first node in the list called circular list. The circular list representation of polynomial

$5x^4 – 8x^3 + 2x^2 + 4x^1 + 9$



If a node is no longer in use we free that node so that node can be reused later. An efficient erase algorithm is used for freeing the nodes. The list is maintained for the freed nodes. When we need a node ,we examine this list. If the list is empty, we need to use new function to create a new node. If the list is not empty, then we may use any one of the node.

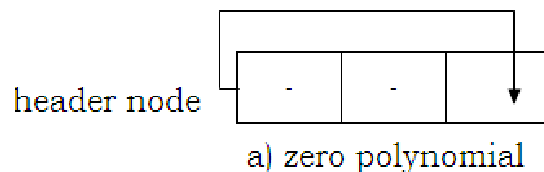
```
void erase(polypointer *ptr)
{
        polypointer temp;
        while(*ptr)
        {
                temp = *ptr;
                *ptr = *ptr -> next;
                delete temp;
        }
}
```

The list of free nodes is called list of available space or avail list or avail.Availabe a variable of type polypointer and always points to the first node in the list. Initially,avail is set to NULL.Here instead of using new and delete we use getnode() and retnode().

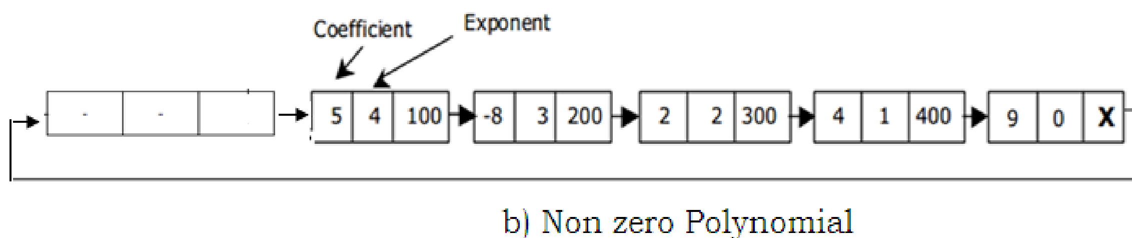
```
Polypointe rgetnode()
{
        polypointer node;
        if(avail)
        {
                node = avail;
                avail = avail -> next;
        }
        else
                node = new poly pointer;
                return node;
}
Void retnode(polypointer node)
{
        node -> next = avail;
        avail = node;
}
```

We can erase circular list in fixed amount of time irrespective of how many number of nodes it may contain .By using erase algorithm, we have the problem of handling zero polynomials. To avoid the zero polynomial, we use header node in to each polynomial. The coefficient and exponent fields of the header is not relevant .The diagrammatic representation of zero and non–zero polynomials are as follows.

```
Void cerase(polypointer *ptr)
{
        polypointer temp;
        if(*ptr)
        {
                temp = *ptr -> next;
                *ptr -> next = avail;
                avail = temp;
                *ptr = null;
        }
}
```



a) zero polynomial



b) Non zero Polynomial

EQUIVALENCECLASSES

A relation ≡, over a set S, is said tobe equivalence relation over S if and only if it is reflexive, symmetric and transitive over S.

1.for any polygon x, x≡x,xi selectrically equivalent to its elfthen≡is reflexive.

2.for any two polygons xandy,x≡y implies y≡x then the relation≡is symmetric

3.for any three polygons x,y and x, x≡y, y≡z implies x≡z then the relation ≡ is transitive

The examples of equivalence relations are numerous.For example "equal to" relationship is an equivalence relation.

1. x = x

2. x = y implies y = x

3. x = y, y = z implies x = z

The equivalence relation is to partition set "S" in to equivalent classes such that the two members x and y of "S" are in the same equivalence class if and only if x≡y.For example there are 12 variables numbered from 1 to 12 with pairs as follows:

1 ≡ 5, 4 ≡ 2, 7 ≡ 11, 9≡ 10, 8 ≡ 5, 7 ≡ 9, 4 ≡ 6, 3 ≡ 12and 12 ≡ 1

The equivalence classes over the equivalence relation is as follows. The 12 variables partitioned into 3 classes.

{ 1, 3, 5, 8, 12}, { 2, 4, 6},{ 7, 9, 10, 11}

The algorithm for equivalence classes defined works in two phases. In the first phase the equivalence pair(i,j) are readded and stored. In second phase,we begin at 1 and find all pairs of (1,j).The values 1 and j are in the same class.By transitivity,all the pairs of the form(j,k)implies k is in the same equivalence class.

```
void equivalence()
{
        initialize;
        while(there are more pairs)
        {
                read the next pair (i,j);
                process this pair;
        }
        initialize  the output;
        do
        {
                output a new equivalence class;
        }while(not done);
}
```

The inputs n and m represents number of objects and number of related pairs. The data structure used here is array to hold these pairs. As we have problem of using arrays we go linked lists .Each node will have data and next parts.We will use largest one dimensional array seq(n) and outputting the object we need out(i).

```
void equivalence()
{
        initialize  seq to null and out to true;
        while(there  are more pairs)
        {
                 read the next pair (i,j);
                 put j on the seq(i) list;
                 put i on the seq(j) list;
        }
```

```
          for(i= 0; i<n; i++)
                    if(out(i))
                    {
                              out(i) = false;
                              output this equivalence class;
                    }
}
```

SEQ	1	2	3	4	5	6	7	8	9	10	11	12
DATA	12	4	12	6	8	4	9	5	7	9	7	1
LINK		0	0			0		0		0	0	
DATA	5			2	1		11		10			3
LINK	0			0	0		0					0

SPARSE MATRIX

"A matrix that contains very few number of non-zero elements is called sparse matrix"

"A matrix that contains more number of zero values when compared with non-zero values is called a sparse matrix"

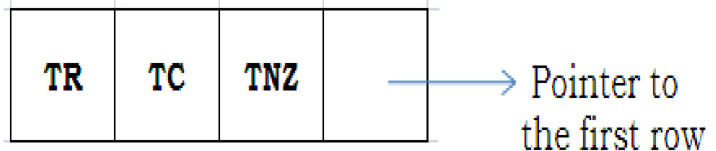
```
abstract  datatype  sparsematrix
{
          instances:
                    a set of triples (row, col, value) where row, col are two
                    integers and value comes from the set item.
          operations:
                    for all a,bϵ sparse matrix , xϵ item, i,j, maxcol, maxrowϵ
                    index
                    sparsematrix create(maxrow, maxcol);
                    sparesematrix matrixtranspose(a);
                    psparsematrix matrixadd(a, b);
                    sparsematrix matrixmultiply(a, b);
}
```

SPARSE MATRIX REPRESENTATION

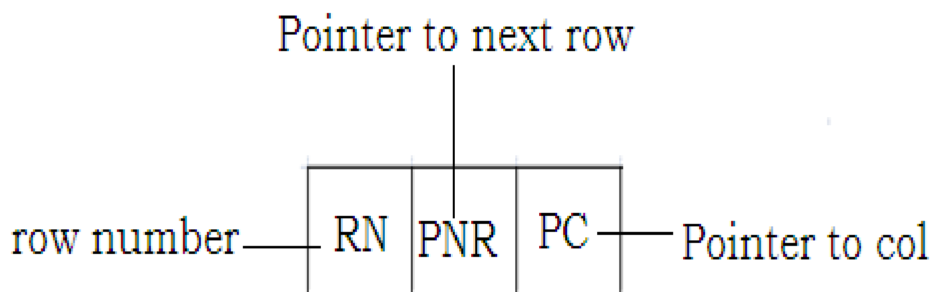
For linked representation, we need three structures.

1. head node

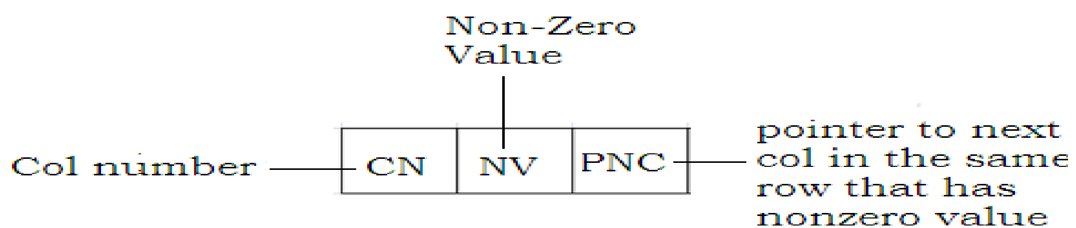
where TR - total no. of rows
TC - total no. of cols
TNZ - total no. of
Non-Zero values



2. row node



3. column node

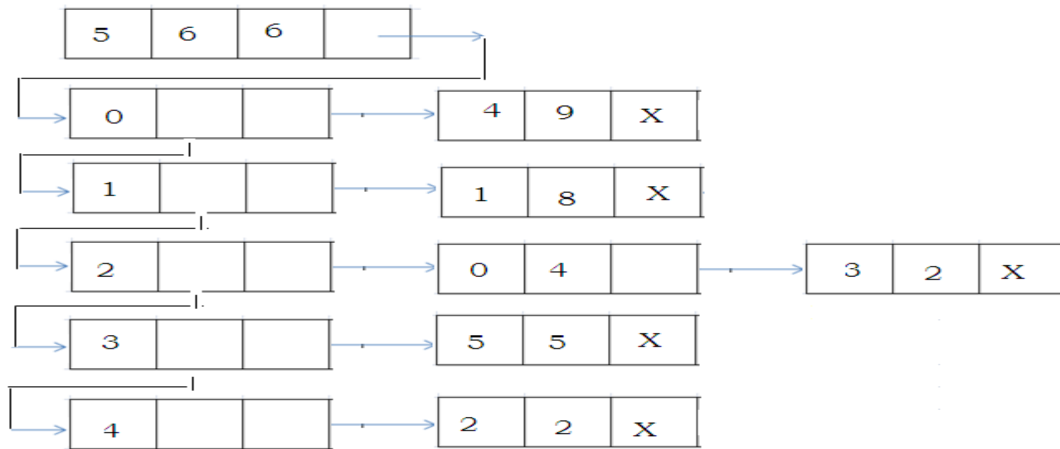


The matrix representation for the sparse matrix is shown below for example.

	c0	c1	c2	c3	c4	c5
r0	0	0	0	0	9	0
r1	0	8	0	0	0	0
r2	4	0	0	2	0	0
r3	0	0	0	0	0	5
r4	0	0	2	0	0	0

5 x 6

In the above matrix representation there are 5 rows, 6 columns and 6 non-zero values. Thelinked representation is as follows:



SPARSEMATRIX INPUT

The sparse matrix input is to read the input lines and obtain linked representation of sparse matrix. The first line of the sparse input represents number of rows ,number of columns and number of non-zero terms. This line is followed by the number of non-zero terms lines .They will be of the form(row,col,value).The order is of row wise and within row maintains column wise. For example, let us consider the sparse matrix of the form

	c0	c1	c2	c3	c4	c5
r0	0	0	0	0	9	0
r1	0	8	0	0	0	0
r2	4	0	0	2	0	0
r3	0	0	0	0	0	5
r4	0	0	2	0	0	0

5 x 6

	rows	cols	value
r1	5	6	6
r2	0	4	9
r3	1	1	8
r4	2	0	4
r5	2	3	2
r6	3	5	5
r7	4	2	2

The node representation is as shown below and the set up for a_{ij} is as follows:

DOWN	ROW	COL	RIGHT
VALUE			

Node

	i	j	
a_{ij}			

set up for a_{ij}

ProcedureMREAD(A)

```
1. read (n, m, r)

2. p = max(m, n)

3. fori = 1 to p do
       call GETNODE(x)
       HDNODE(i) = x
       ROW(x) = COL(x) = 0
       RIGHT(x) = VALUE(x) = x

4. current_row = 1

       LAST= HDNODE(1)

5. fori = 1 to r do

       read(rrow, ccol, val)

       if(rrow>current_row) then
           RIGHT(LAST) = HDNODE(current_row)
           current_row = rrow
           LAST= HDNODE(rrow)
           call GETNODE(x)
           ROW(x) = rrow
           COL(x) = ccol
           VALUE(x) =
           valRIGHT(LAST)
           = x LAST= x
           DOWN(VALUE(HDNODE(ccol))) = x
           VALUE(HDNODE(ccol)) = x

6. if r = 0 then RIGHT(LAST) = HDNODE(current_row)

7. fori = 1 to m do

       DOWN(VALUE(HDNODE(i))) = HDNODE(i)

8. call GETNODE(A)

       ROW(A) = n

       COL(A) = m

9. fori = 1 to p-1 do

       VALUE(HDNODE(i)) = HDNODE(i + 1)

10. if p= 0 then VALUE(A) = A

     else VALUE(HDNODE(p)) = A

           VALUE(A) = HDNODE(1)
```

DELETING A SPARSEMATRIX

The deletion of all nodes from the sparse matrix is returned to the systemmemory space .It is possible to delete only one node at a time . This

Node is to returned to the available space list .Similarly all the nodes are deleted node by node and returned to available space list.

ProcedureMERASE(A)

1. RIGHT(A) = AV

   AV = A
   NEXT= VALUE(A)

2. while(NEXT= A) do

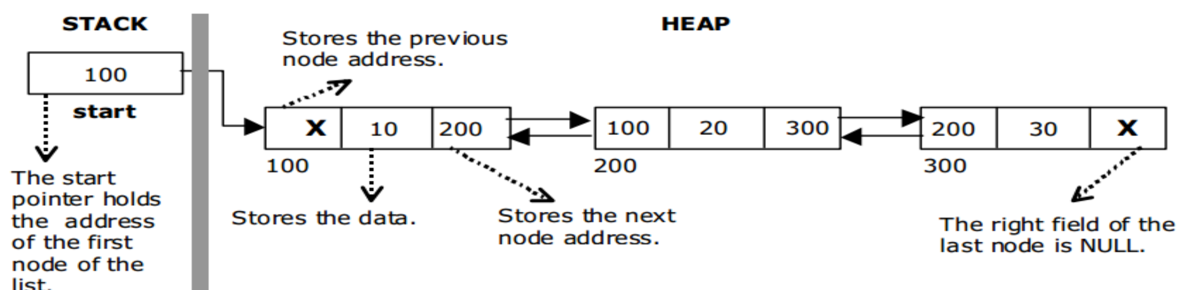
   T= RIGHT(NEXT)
   RIGHT(NEXT) = AV
   AV = T
   NEXT= VALUE(NEXT)

DOUBLYLINKED LISITS

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position .It provides bi-directional traversing .Each node has three fields namely

- ✓ Left link
- ✓ Data
- ✓ Right link

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.



Doubly Linked List Representation

AbstractDataType DoublyLinkedList

{

instances:

finite collection of zero or more elements linked by two pointers, one pointing the previous node and the other pointing to the next node.

operations:

Count(): Count the number of elements in the list.

Addatbeg(x): Add x to the beginning of the list.

Addatend(x): Add x at the end of the list.

Insert(k, x):Insert xjust after kth element.

Delete(k): Delete the kth element.

Search(x): Return the position of x in the list otherwise return-1 if not found

Display( ): Display all elements of the list

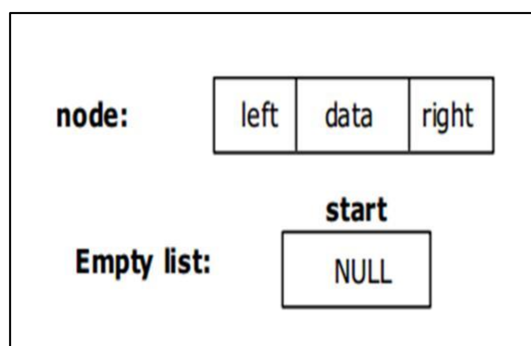
}

Implementation of Doubly Linked List

Before writing the code to build the list, we need to create a **start** node**,** used to create and access othernodes in the linked list.

- ✓ Creating a structure with one data item and a right pointer, which will be pointing to next node of the list and left pointer pointing to the previous node. This is called as self-referential structure.
- ✓ Initialize the start pointer to be NULL.

```
Struct dlinklist
{
        struct dlinklist * left;
        int data;
        struct dlinklist * right;
};
typedef struct dlinklist node;
node *start = NULL;
```



Basic operation performed on doublylinkedlist

The different operations performed on the doubly linked list are listed as follows.

- ✓ Creation
- ✓ Insertion
- ✓ Deletion
- ✓ Traversing
- ✓ Display

Creating a node for DoublyLinkedList

Creating a doubly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the new() function.

The function getnode(),is used for creating a node, after allocating memory for the node, the information for the node data part has to be read from the user and set left and right field to NULL and finally return the node.

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        cout<< Enter data;
        cin>>newnode-> data
        newnode -> left = NULL;
        newnode -> right = NULL;
        return newnode;
}
```

Creating a Doubly Linked List with „n" number of nodes

The following steps are to be followed to create „n" number of

nodes. 1. Get the new node using getnode().

```
newnode = getnode();
```

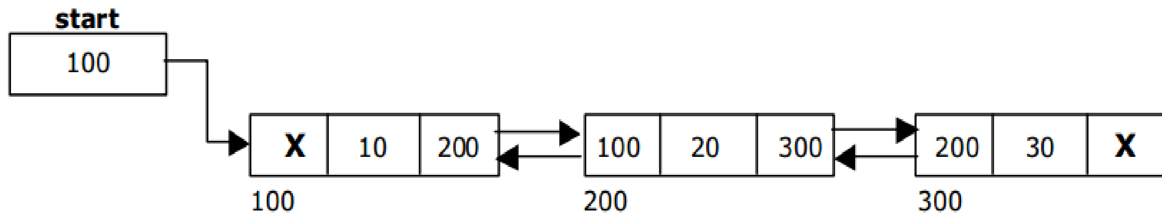
2. If the list is empty, assign new node as start.

```
start = newnode;
```

3. If the list is not empty, follow the steps given below.

- ✓ The left field of the new node is made to point the previous node.
- ✓ The previous nodes right field must be assigned with address of the new node.

4. Repeat the above steps „n" times.



Double Linked List with 3 nodes

The function createlist(), is used to create „n" number of nodes

```
Void createlist(int n)
{
        inti;
        node *newnode;
        node *temp;
        for(i = 0; i< n ; i++) {
                newnode = getnode();
                if(start = = NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> right != NULL)
                        {
                                temp = temp -> right;
                        }
                        temp -> right = newnode;
                        newnode -> left = temp;
                }
        }
}
```

INSERTION OF A NODE

One of the most important operations that can be done in a doubly linked list is the insertion of a node .Memory is to be allocated for the new node before reading the data .The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user .The left and right fields of the new node are set to NULL. The new node can then be inserted at three different places namely:

- ✓ Inserting a node at the beginning.
- ✓ Inserting a node at the end.
- ✓ Inserting a node at specified position.

INSERTING A NODE AT THE BEGINNING

The following steps are to be followed to insert a newnode at the beginning of the list:

1. Get the new node using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty, follow the steps given below:

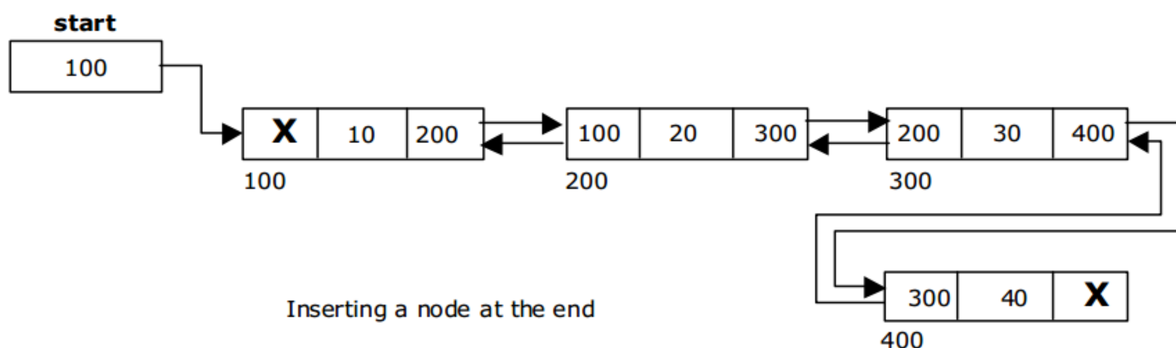
```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```

INSERTING A NODE AT THE END

The following steps are followed to insert a newnode at the end of the list:

1. Get the new node using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
        temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```



Inserting a node at the end

INSERTING A NODE AT SPECIFIED POSITION

The following steps are followed, to insert a new node in an intermediate position in the list:

1. Get the new node using getnode() then newnode = getnode();

2.Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

3.Store the starting address (which is in start pointer)in temp and prev pointers. Then traverse the temp pointer up to the specified position followed by prev pointer.

4. After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode ->right= temp->right;
temp -> right ->left = newnode;
temp-> right= newnode;
```

DELETION OF A NODE

Another operation that can be done in a doubly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places.

- ✓ Deleting a node at the beginning.
- ✓ Deleting a node at the end.
- ✓ Deleting a node at specified position

DELETING A NODE AT THE BEGINNING

The following steps are followed ,to delete a node at the beginning of the list:

1.If list is empty then display „EmptyList" message.2.

If the list is not empty, follow the steps given below:

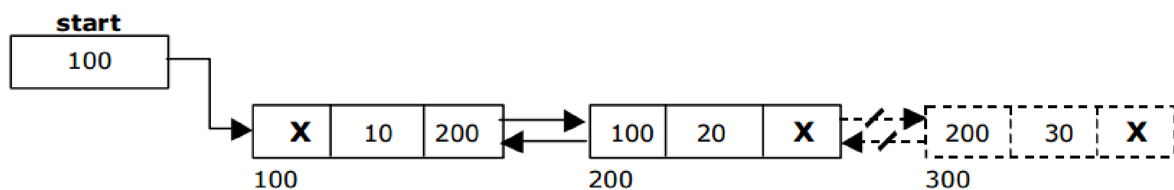
```
temp = start;
start = start -> right;
start -> left = NULL;
delete temp;
```

DELETING A NODE AT THE END

The following steps are followed todelete a node at the end of the list:

1. If list is empty then display „EmptyList" message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp ->right;
}
temp –> left -> right = NULL;
delete temp;
```



Deleting a node at the end

DELETING A NODE AT SPECIFIED POSITION

Thefollowingstepsarefollowed,todeleteanodefromthespecified position in the list.

1. If list is empty then display „EmptyList" message
2. If the list is not empty, follow the steps given below.

```
if(pos> 1 &&pos<nodectr) {
        temp = start;ctr
        = 1;
        while(ctr<pos) {
                temp = temp -> right;
                ctr++;
        }
        temp -> right -> left = temp ->left;
        temp -> left -> right = temp -> right;
        delete temp;
}
```

TRAVERSAL AND DISPLAYING A LIST(LEFT TO RIGHT)

To display the information, you have to traverse the list, node by node from the first node ,until the end of the list is reached .The function *traverse_left_right*() is used for traversing and displaying the information stored in the list from left to right. The following steps are followed, to traverse a list from left to right:

1. If list is empty then display „EmptyList‟ message.2.

If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
        cout<<temp -> data;
        temp = temp -> right;
}
```

TRAVERSAL AND DISPLAYING A LIST (RIGHT TO LEFT)

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() is used for traversing and displaying the information stored in the list from right to left .The following steps are followed, to traverse a list from right to left:

1. If list is empty then display „EmptyList‟ message.2.

If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
        temp = temp -> right;
while(temp != NULL)
{
        cout<< temp -> data;
        temp = temp -> left;
}
```

COUNTING THE NUMBER OF NODES

The following code will count the number of nodes exist in the list (using recursion).

```
Int countnode(node *start)
{
        if(start = = NULL)
                return 0;
        else
                return(1 + countnode(start ->right ));
}
```

REPRESENTATIONOFGENERALIZEDLISTS

The list A is written as A={$_{0,1,2,\ldots\ldots\ldots n-1}$}.The capital letters are used to represent the names of the lists. The length of the list is indicated by n. the lower case letters are used to represent the atoms. If n>=0, the n_0 is the head of A and tail of A is $_{1,2,\ldots\ldots\ldots n-1}$.Some examples of generalized lists are listed as follows:

1. D = ( ) indicates null list or empty list, itslength is zero

2.A=(a,(b,c))indicates a list of length two, its first element is an a to m and its second element is a linear list (b, c)

3.B=(A,A,())indicates a list of length 3 whose first two elements are the list A and third is a null list.

4.C=(a,C)indicates a list of length 2,Ccorresponds to the infinite list C =(a, (a, (a,……..).

```
Class polynode
{
        polynode *link
        int exp;
        triple trio;
        union
        {
                char var1;
                polynode *dlink;
                int coef;
        };
};
```

In this representation, there three types of nodes depending on the value of trio. If trio==var, then the node is head node ,the field var1 is used to indicate the name of the variable and exp=0.If trio==ptr,then coefficient it self is a list pointed by the dlink field .If trio==no, then coefficient is an integer and is stored in the coef field and exp is based on the list.

Every generalized list can be represented by using the node structure as follows.

```
Enum boolean {false, true};

class genlist;

Class genlistnode
{
	Friend class genlist;
	genlist node *link
	boolean tag
	union
	{
		char data;
		genlistnode *dlink;
	};
};

classgenlist {
	private:
		genlistnode *first;
};
```

RECURSIVE ALGORITHMS FOR LISTS

When a data object is defined recursively, it is easy to describe recursive algorithms that work on the data objects recursively. A recursive algorithm consists of two components. The first one is the recursive function itself called workhorse and the second one is the function that call the recursive function at the top level is called driver.

When a recursive is used to implement a class operation, we require two class member functions. The driver is declared as public member function and workhorse is declared as a private member function. The different types of recursive algorithms are

1.**Copying a list**–produces an exact copy of a non-recursive list "*l*" in which no sublists are shared.

driver

```
void genlist :: copy(const genlist&l) {
	first = copy(l, first);
}
```

workhorse

```
genlistnode *genlist :: copy(genlistnode *p)
{
	genlistnode *q = 0;
	if(p)
	{
		q = new genlistnode;
		q -> tag = p -> tag; if(!p -> tag)
```

```
                q -> data = p -> data;
            else
                q ->dlink = copy(p ->dlink); q
                -> link = copy(p -> link);
        }
        Return q;
    }
```

2. List Equality–determines whether two lists are identical or not. To be identical, the lists must have same structure and same data in corresponding data members.

driver

```
int operator ==(const genlist&l, const genlist&m) {
        return equal(l, first, m, first);
}
```

workhorse

```
int equal(genlistnode *s, genlistnode *t)
{
        int x;
        if((!s) &&(!t))
                return 1;
        if(s && t &&(s -> tag == t -> tag))
        {
                if(!s -> tag)
                        if(s -> data == t -> data)
                                x = 1;
                        else
                                x = 0;
                else
                        x = equal(s ->dlink, t ->dlink);
                if(x)
                        return equal(s -> link, t-> link); }
        return 0;
}
```

3.List Depth–computes the depth of the list .if the list is empty then the depth of the list is zero.

$$\text{depth}(s) = \begin{cases} 0 & \text{if } s \text{ is an atom} \\ 1 + \max\{\text{depth}(x_1, x_2 \ldots\ldots x_n)\} & \text{if } s \text{ is the list } (x_1, x_2, \ldots\ldots x_n) \end{cases}$$

driver

```
int genlist :: depth()
{

        return depth(first);
}
```

workhorse

```
int genlist :: depth(genlistnode *s)
{
	if(*s)
		return 0;
	genlistnode *p = s;
	int m = 0;
	while(p)
	{
		if(p -> tag)
		{
			int n = depth(p ->dlink);
			if(m < n)
				m = n;
		}
		p = p -> link;
	}
	return m + 1;
}
```