UNIT – II

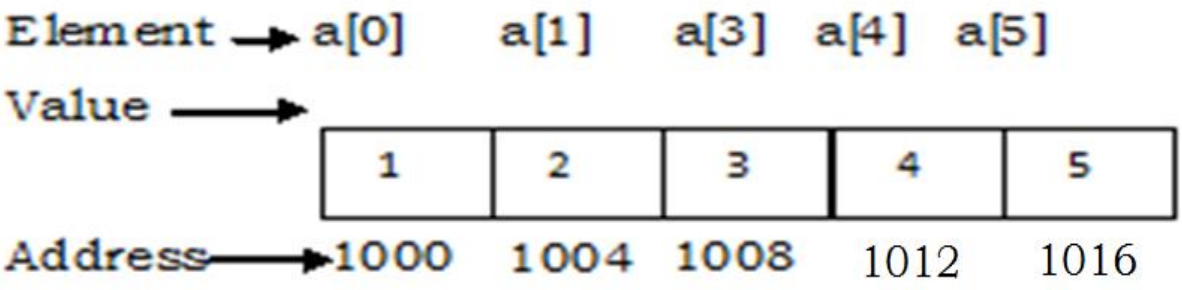
POINTERS

POINTER ARRAYS

- ✓ When an array is declared, the compiler allocates a base address and sufficient amount of memory to contain all the elements of the array in continuous memory locations.
- ✓ The base address is the location of the first element of the array denoted by **a[0]**.
- ✓ The compiler also defines the array name as constant pointer to the first element.
- ✓ For Example: -

int a [5] = {1, 2, 3, 4, 5};

- ✓ Here if the base address is 1000 for "a" and integer occupies 4 bytes then the five elements requires 20 bytes as shown below.



- ✓ The name of the array is "a" and it is defined as a constant pointer pointing to the first element of the array and it is a[0] whose base address is 1000 becomes the value of "a". It is represented as

a = &a[0] = 1000;

- ✓ If p is a pointer of integer type then p to point the array a is given by the assignment statement

p = a;

which is equivalent to

p = &a[0];

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

- ✓ Now it is possible to access every value of a using p++ to move from one element to another element as

P	**= &a[0]**	**=1000**
P+1	**= &a[1]**	**= 1004**
P+2	**= &a[2]**	**= 1008**
P+3	**= &a[3]**	**= 1012**
P+4	**= &a[4]**	**= 1016**

- ✓ The address of the element is calculated by using the formula

address of a[3] = base address + (3 * scale factor of int)

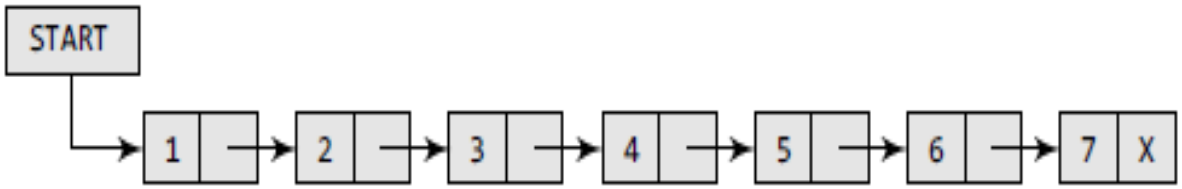
- ✓ When handling arrays we can use pointers to access the array elements. Hence *(p+3) gives the value of a[3].
- ✓ Pointers can also be used to manipulate two dimensional arrays. In one dimensional array "a" the expression.

***(a+i) or *(p+i)**

LINKED LIST

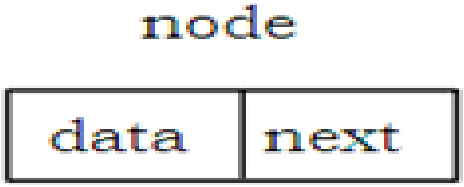
- ✓ It is a collection of linear list of data elements.
- ✓ The data elements are called **nodes**.
- ✓ Each node contains **two** parts: **data** and **link**.
- ✓ The data represents **integers** and link is a **pointer** that points to next node.
- ✓ The last node of the linked list is not connected to any node so it stores the value **NULL** in link part.
- ✓ Here NULL is defined as **-1**
- ✓ NULL pointer denotes **end** of the list.
- ✓ It contains pointer variable called start node that contains the address of first node in the list
- ✓ We can traverse the list starting from start node that contains first node address and in turn first node contains second node address and so on thus forming chain of nodes.
- ✓ If **start == NULL** then the list is empty.

- ✓ The diagrammatic representation of linked list is shown below:



Simple linked list

NODE REPRESENTATION



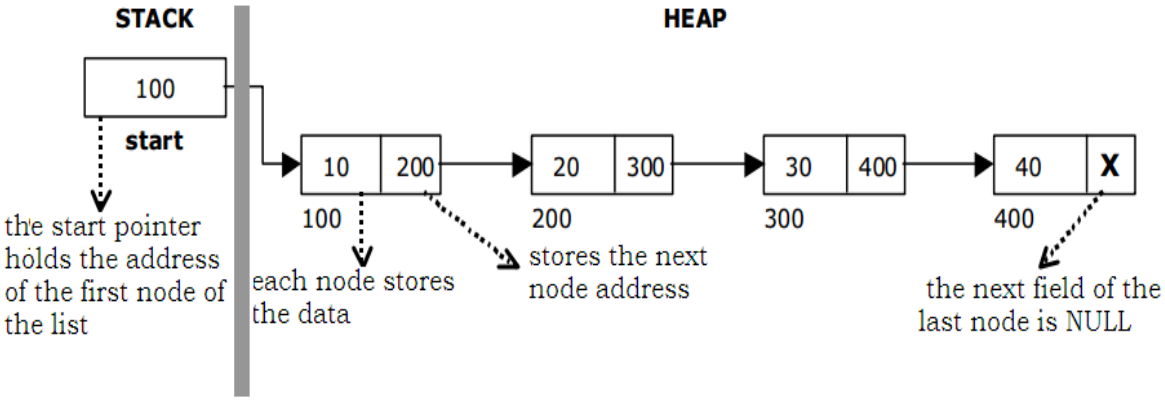
- ✓ In C language, the code for the linked list is

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

SINGLE LINKED LIST

- ✓ **"A single linked list is a linked list in which each node contains only one link pointing to the next node in the 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 **malloc()** and it is explicitly de-allocated using **free()**.
- ✓ The single linked list starts with a pointer to the **"start"** node.
- ✓ The single linked list is called as **linear list** or **chain**.

- ✓ The **traversing** of data can be in **one** direction only.



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.

ADT FOR SINGLE LINKED LIST

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 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

Traverse( ): Display all elements of the list

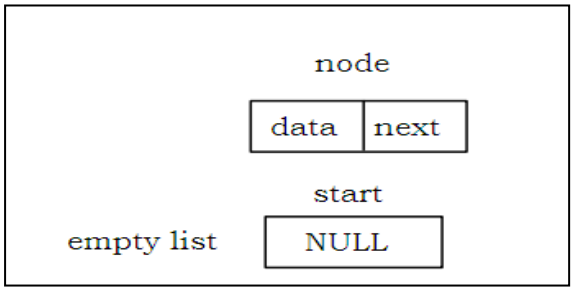
}

IMPLEMENTATION OF SINGLE LINKED LIST

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;
```



BASIC OPERATION PERFORMED ON SINGLE LINKED LIST

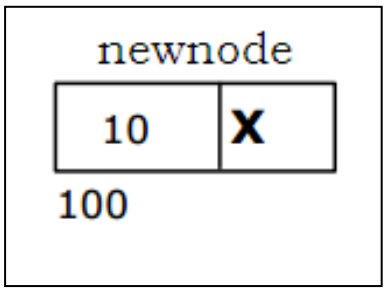
The different operations performed on the single linked list are listed as follows.

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching

Creating a node for Single Linked List

- ✓ 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 **malloc()** 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;
        printf("Enter data");
        scanf("%d", &newnode  -> data;
        newnode -> next = NULL;
        return newnode;
}
```



Creating a Singly 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().

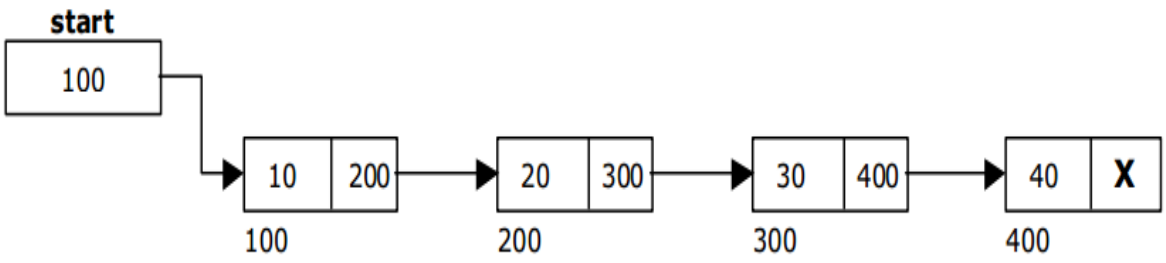
   newnode = getnode();

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 next field of the new node 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 new node 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)
{
```

```
        int i;
        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 important operations that can be done in a singly linked list is the insertion of a node.
- ✓ Memory is to be allocated for the newnode before reading the data.
- ✓ The newnode will contain empty data field and empty next field.
- ✓ The data field of the newnode is then stored with the information read from the user.
- ✓ The next field of the newnode is assigned to NULL.
- ✓ The newnode 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.**

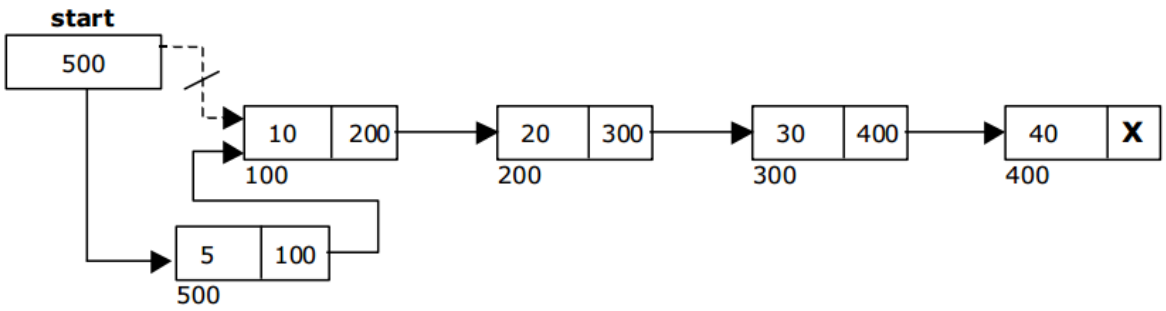
INSERTING A NODE AT THE BEGINNING

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

1. Get the newnode 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 -> 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;
        }
}
```

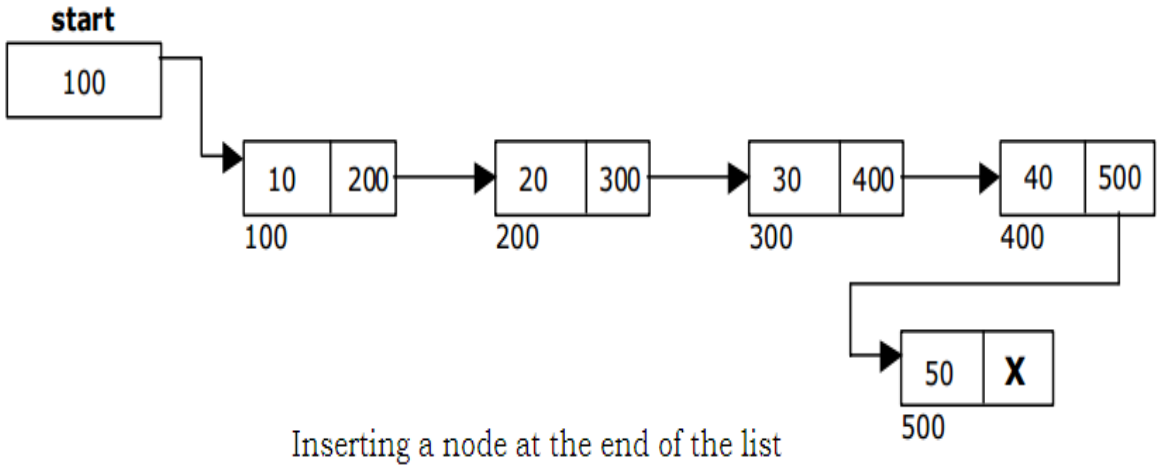

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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;
        }
}
```

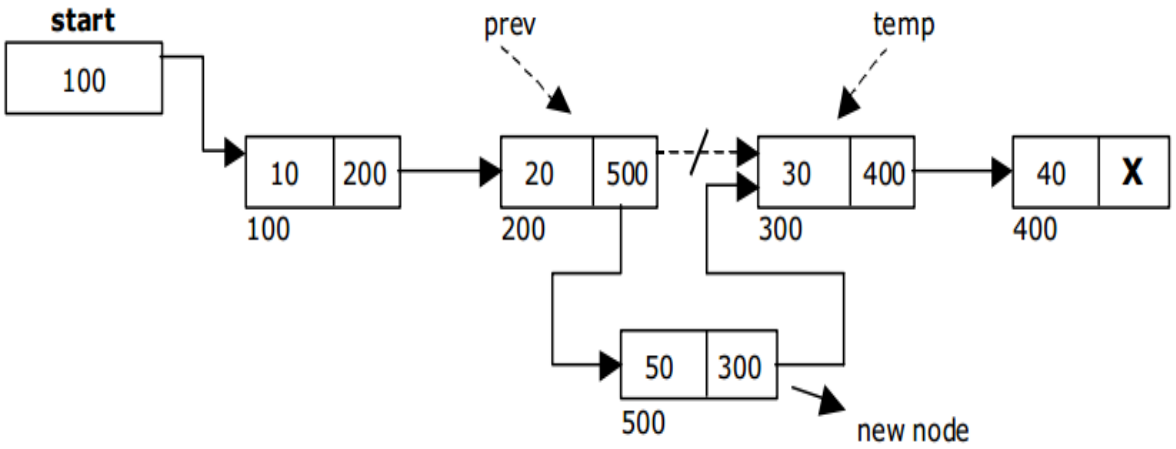
INSERTING A NODE AT SPECIFIED POSITION

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 upto 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();
        printf(" Enter the position");
        scanf("%d", &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
        {
                printf("%d", pos);
        }
}
```

DELETION OF A NODE

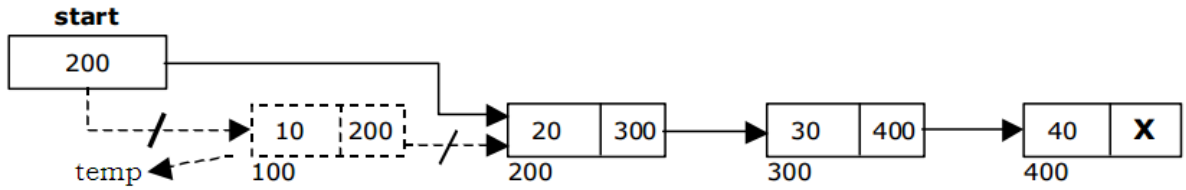
- ✓ Another operation that can be done in a singly linked list is the deletion of a node.
- ✓ Memory is to be released for the node to be deleted.
- ✓ It is done by using **free()** function.
- ✓ A node can be deleted from the list from three different places.
  - ✓ **Deleting a node at the beginning.**
  - ✓ **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 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(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)
        {
                printf(" Empty List ");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("Node deleted");
        }
}
```

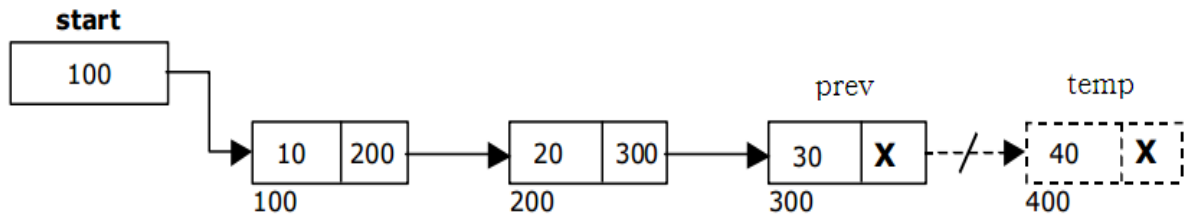

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DELETING A NODE AT THE END

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

1. If list is empty then display 'Empty List' 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;
free(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)
        {
                printf(" Empty List ");
                return ;
        }
        else
        {
                temp = start;
```

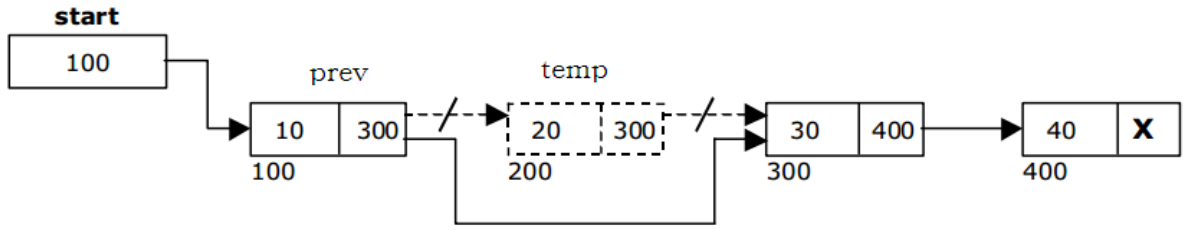
```
            prev = start;
            while(temp -> next != NULL)
            {
                  prev = temp;
                  temp = temp -> next;
            }
            prev -> next = NULL;
            free(temp);
            printf("Node deleted");
      }
}
```

DELETING A NODE AT SPECIFIED POSITION

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

1. If list is empty then display 'Empty List' 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;
            free(temp);
            printf("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()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf(" Empty List ");
                return ;
        }
        else
        {
                printf(" Enter position of node to delete ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf(" This node doesnot exist: ");
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
```

```
                    ctr ++;
                }
                prev -> next = temp -> next;
                free(temp);
                printf("Node deleted");
            }
            else
                printf("Invalid position");
        }
    }
```

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 the following steps.

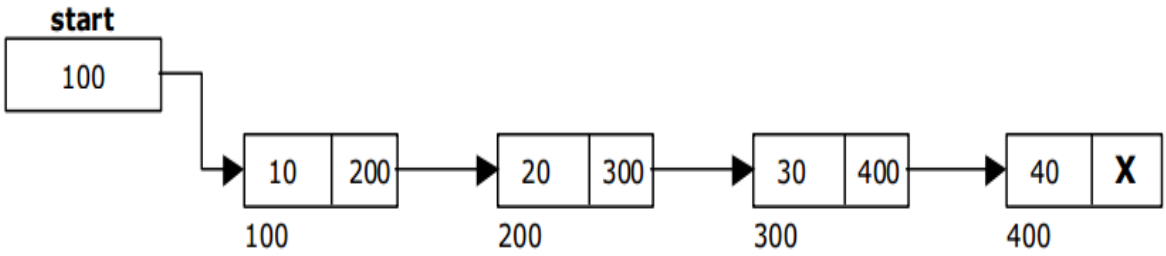
1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

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;
    printf(" The contents of List (Left to Right) ");
    if(start == NULL )
        printf(" Empty List ");
    else
    {
        while (temp != NULL)
        {
            printf("%d", temp -> data);
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
                temp = temp -> next;
            }
        }
        printf("%d", X);
}
```



Single Linked List with 4 nodes

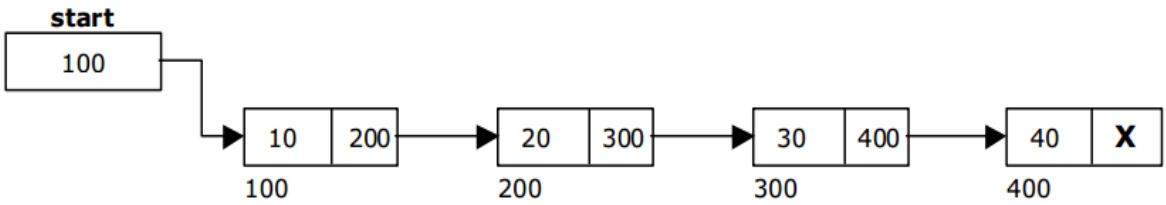
SEARCHING A NODE IN A SINGLE LINKED LIST

- ✓ Searching a single linked list means to find a particular element in the single linked list.
- ✓ A single linked list consists of nodes which are divided into two parts, the data part and the next part.
- ✓ So searching means finding whether a given value is present in the data part of the node or not.
- ✓ If it is present, then display element found otherwise element not found.

```
void search()
{
        node *temp;
        int value = 30;
        temp = start;
        if(start == NULL )
                printf(" Empty List ");
        else
        {
                while (temp != NULL)
                {
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
                    if(value = temp->data)
                    {
                        printf(" Element found ");
                        return;
                    }
                    temp = temp -> next;
                }
                printf(" Element not found ");
            }
        }
```



Single Linked List with 4 nodes

ADVANTAGES OF SINGLE LINKED LIST

- ✓ Insertions and Deletions can be done easily.
- ✓ It does not need movement of elements for insertion and deletion.
- ✓ The space is not wasted as we can get space according to our requirements.
- ✓ Its size is not fixed.
- ✓ It can be extended or reduced according to requirements.
- ✓ Elements may or may not be stored in consecutive memory available, even then we can store the data in computer.
- ✓ It is less expensive.

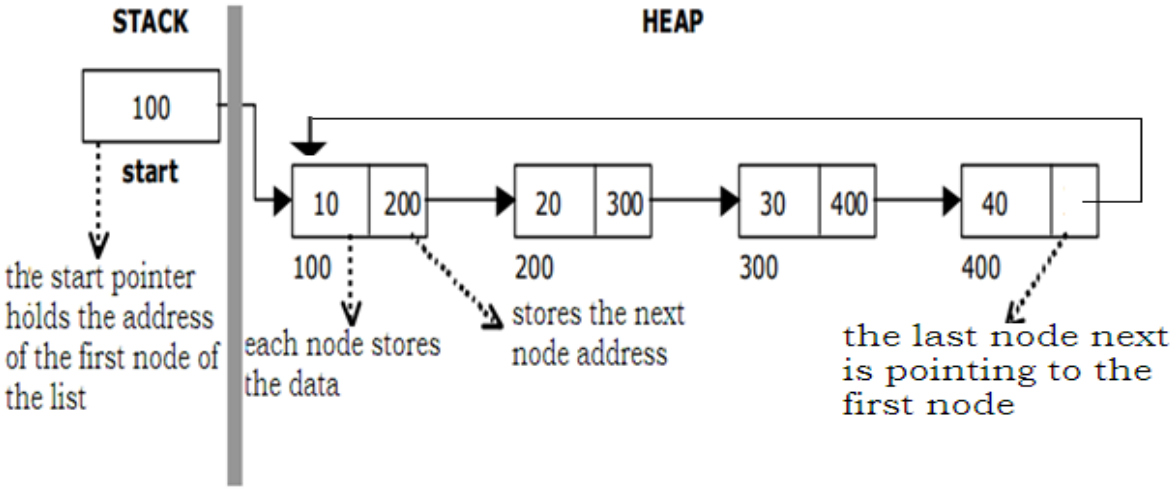
DISADVANTAGES OF SINGLE LINKED LIST

- ✓ It requires more space as pointers are also stored with information.
- ✓ Different amount of time is required to access each element.
- ✓ If we have to go to a particular element then we have to go through all those elements that come before that element.

- ✓ We cannot traverse it from last.
- ✓ It is not easy to sort the elements stored in the Single linked list.

CIRCULAR LINKED 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 contains the value of the node and the next part has the address of the next node.
- ✓ The last node of list has the next pointer pointing to the first node thus making circular traversal possible in the list. 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.



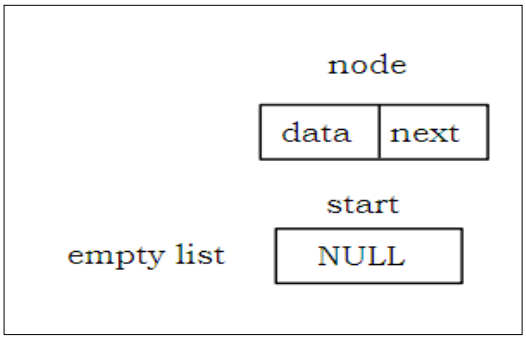
Circular Linked List Reresentation

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



BASIC OPERATION PERFORMED ON CIRCULAR LINKED LIST

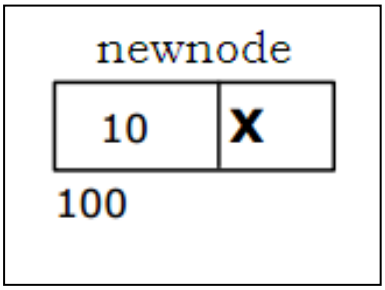
The operations on the circular linked list are listed as follows.

1. Creation
1. Insertion
2. Deletion
3. Traversing
4. 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 **malloc()** 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;
        printf(" Enter data ");
        scanf("%d", &newnode  -> data);
        newnode -> next = NULL;
```



Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

```
        return newnode;
    }
```

Creating a Circular 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.

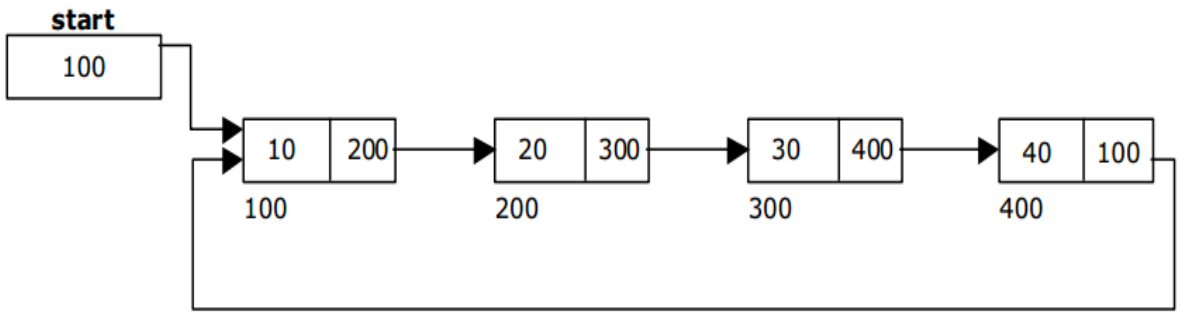
start = newnode;

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

```
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)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n ; i++)
    {
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

```
            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

- ✓ One operation performed on circular linked list is the insertion of a node.
- ✓ Memory is to be allocated for the newnode before reading the data.
- ✓ The newnode will contain empty data field and empty next field. The data field of the newnode is then stored with the information read from the user. The next field of the newnode is assigned to NULL.
- ✓ The newnode can then be inserted at three different positions:
    - ✓ **Inserting a node at the beginning.**
    - ✓ **Inserting a node at the end.**

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().

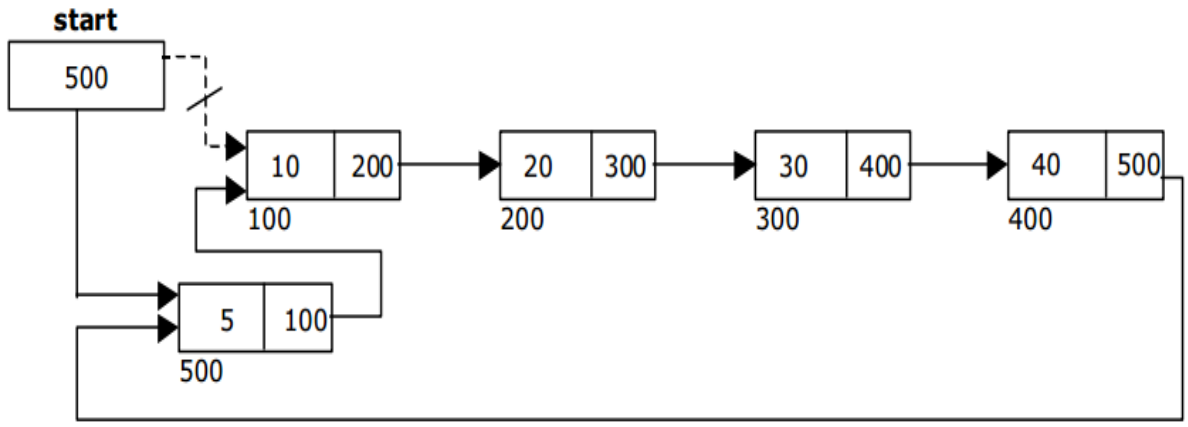
```
newnode = getnode();
```

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

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().

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

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

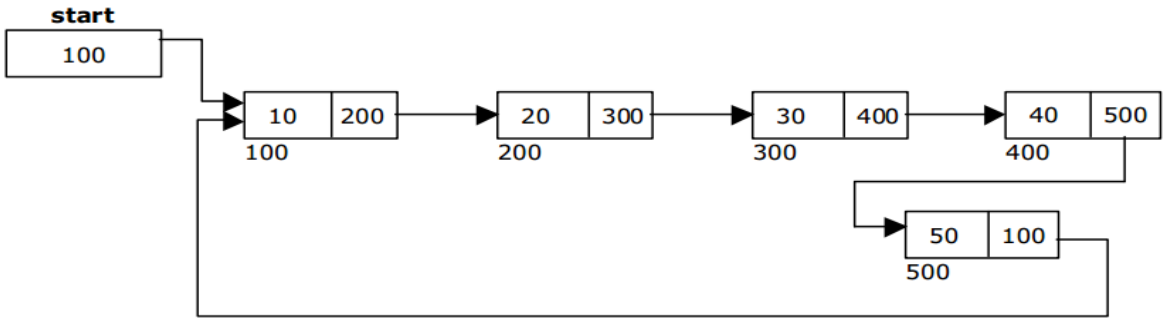
```
start = newnode;
newnode -> next = start;
```

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

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


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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



Inserting a node at the end

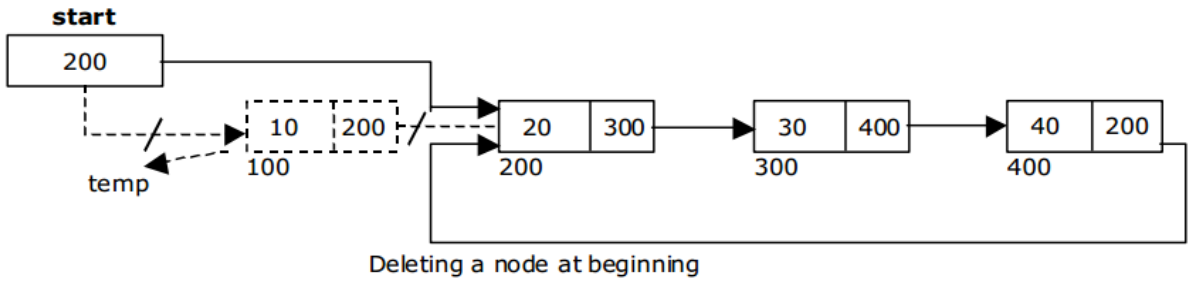
DELETING A NODE AT THE BEGINNING

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

1. If the list is empty, display a message 'Empty List'.
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

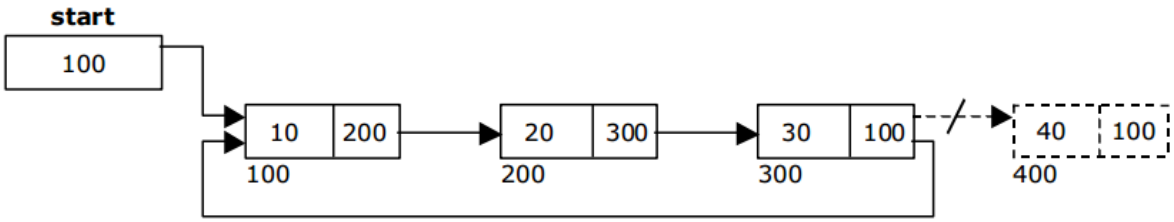
The following steps are followed to delete a node at the end of the list:

1. If the list is empty, display a message 'Empty List'.
2. If the list is not empty, follow the steps given below:

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
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.

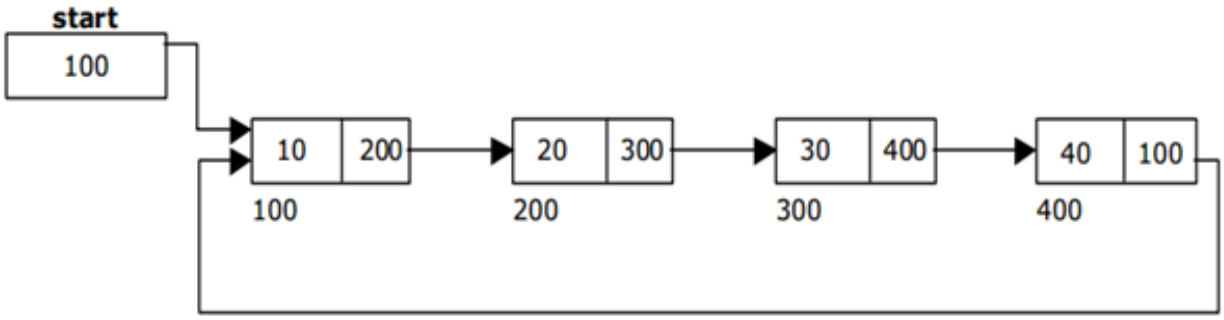
TRAVERSING A CIRCULAR LINKED LIST FROM LEFT TO RIGHT

- ✓ To display the list, we have to traverse (move) the circular linked list, node by node from the first node, until the end of the list is reached.
- ✓ Traversing a list involves the following 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;
        printf(" The contents of List (Left to Right)");
        if(start == NULL )
                printf(" Empty List ");
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

```
        else
        {
                do
                {
                        printf("%d", temp -> data);
                        temp = temp -> next;
                } while (temp != start);
        }
}
```



Circular Linked List with 4 nodes

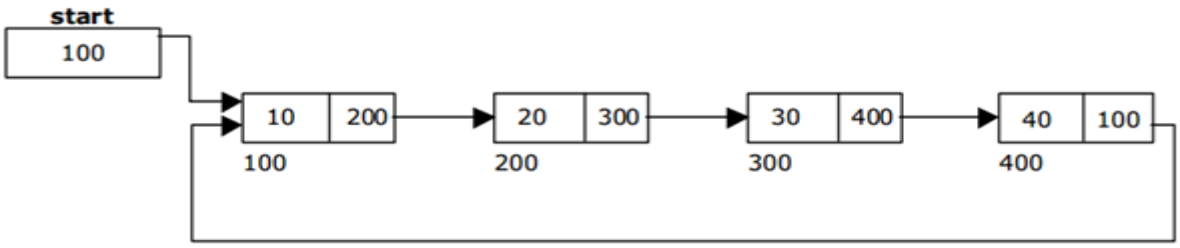
SEARCHING A NODE IN A CIRCULAR LINKED LIST

- ✓ Searching a circular linked list means to find a particular element in the circular linked list.
- ✓ A circular linked list consists of nodes which are divided into two parts, the data part and the next part.
- ✓ So searching means finding whether a given value is present in the data part of the node or not.
- ✓ If it is present, then display element found otherwise element not found.

```
void search()
{
        node *temp;
        int value = 30;
        temp = start;
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
        if(start == NULL )
                printf(" Empty List ");
        else
        {
                while (temp->next != start)
                {
                        if(value = temp->data)
                        {
                                printf(" Element found ");
                                return;
                        }
                        temp = temp -> next;
                }
                printf(" Element not found ");
        }
}
```



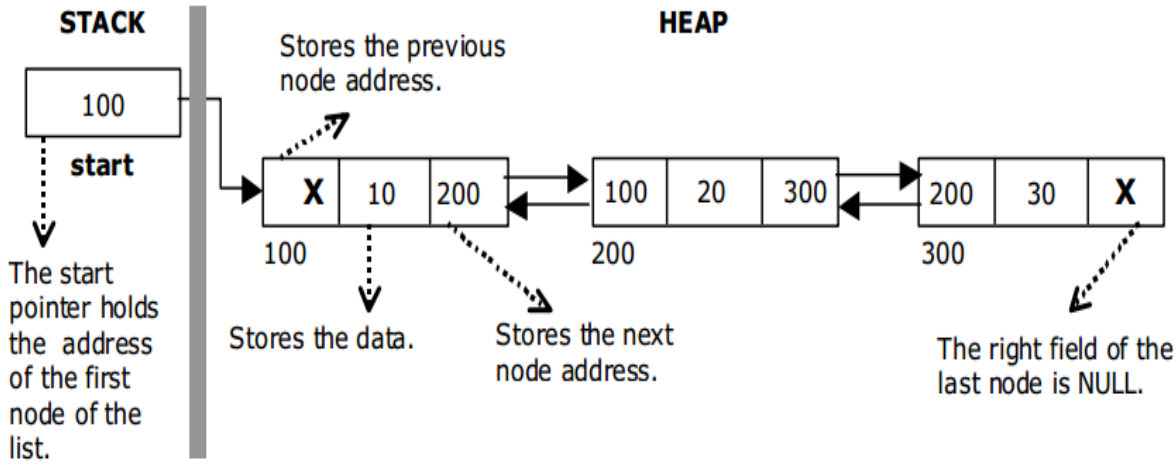
Circular Linked List with 4 nodes

DOUBLY LINKED LSITS

- ✓ 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**.



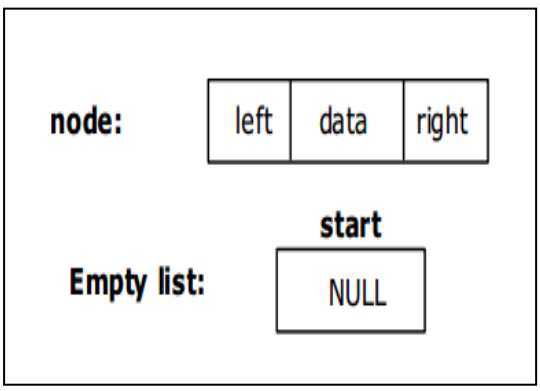
Doubly Linked List Representation

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 other nodes 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;
```



Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BASIC OPERATION PERFORMED ON DOUBLY LINKED LIST

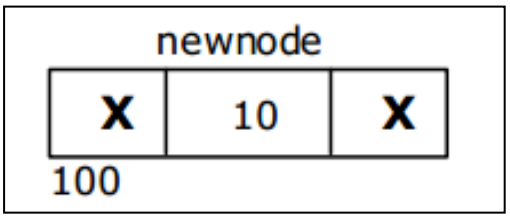
The different operations performed on the doubly linked list are listed as follows.

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Display

Creating a node for Doubly Linked List

- ✓ Creating a double 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 **malloc()** 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** fields to **NULL** and finally return the **node.**

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        printf(" Enter data ");
        scanf("%d", &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().

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

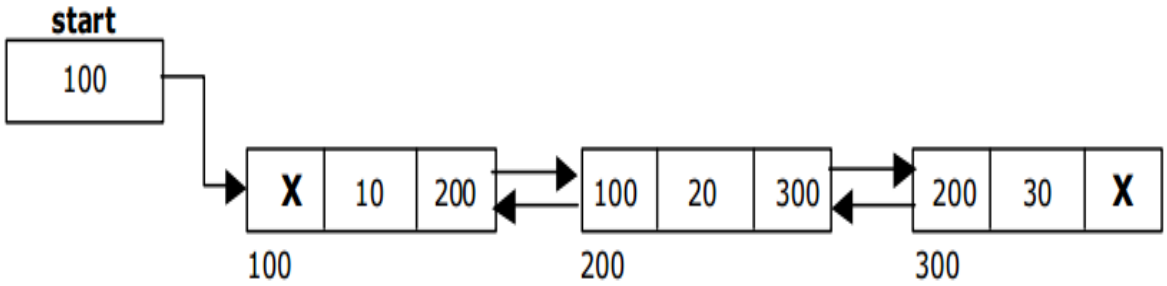
newnode = getnode();

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)
{
        int i;
        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

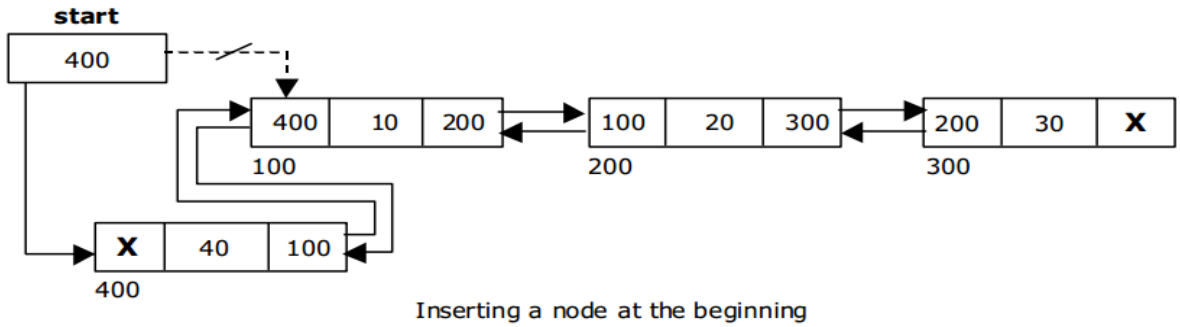
- ✓ One of the most important operation that can be done in a doubly linked list is the insertion of a node.
- ✓ Memory is to be allocated for the **newnode** before reading the data.
- ✓ The **newnode** will contain **empty data** field and **empty left** and **right fields.**
- ✓ The data field of the newnode is then stored with the information read from the user.
- ✓ The left and right fields of the newnode are set to **NULL.**
- ✓ The newnode 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 newnode 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;
```



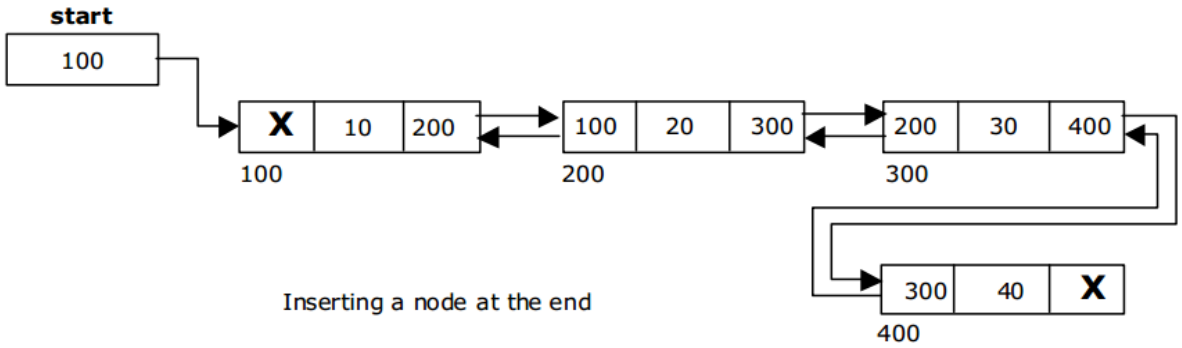
Inserting a node at the beginning

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 -> 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.

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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

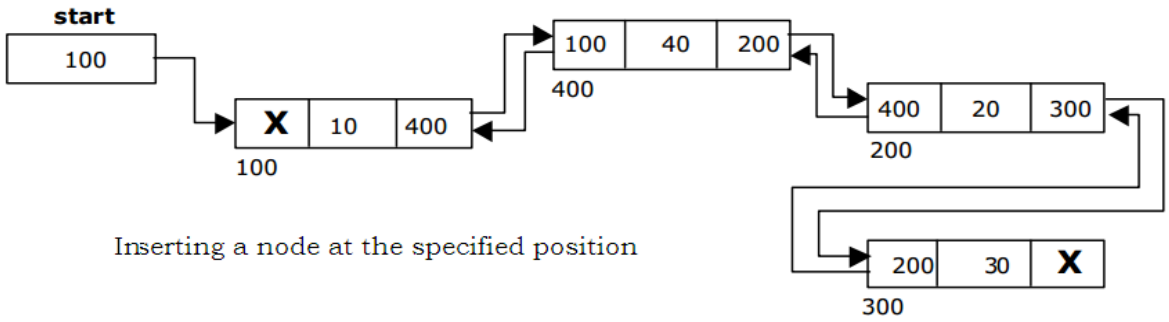
4. After reaching the specified position, follow the steps given below:

newnode -> left = temp;

newnode ->right = temp ->right;

temp -> right ->left = newnode;

temp -> right = newnode;



Inserting a node at the specified position

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 'Empty List' message.

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

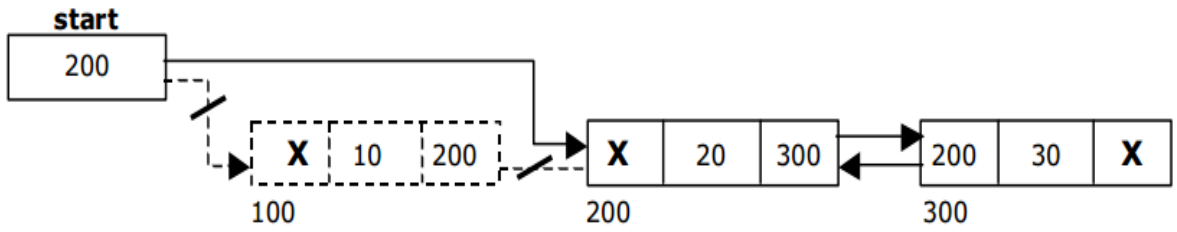
temp = start;

start = start -> right;

start -> left = NULL;

free(temp);

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



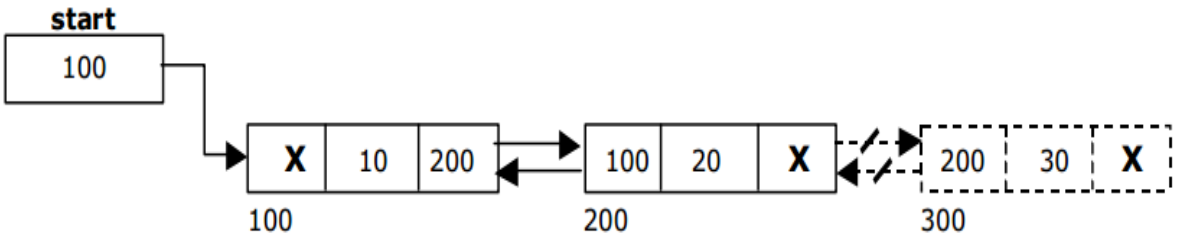
Deleting a node at beginning

DELETING A NODE AT THE END

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

1. If list is empty then display 'Empty List' 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;
free(temp);
```



Deleting a node at the end

DELETING A NODE AT SPECIFIED POSITION

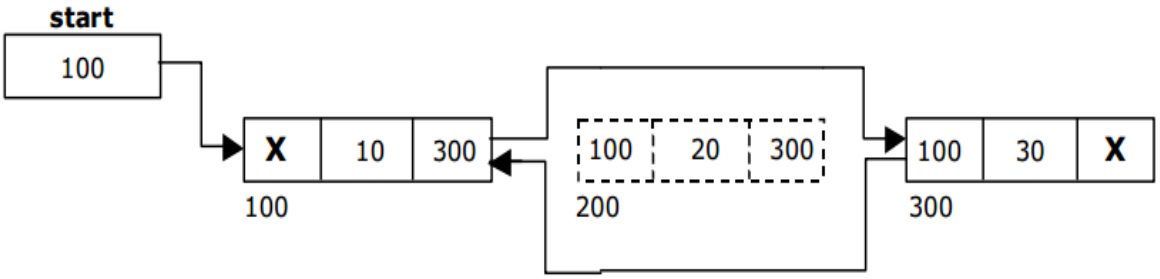
The following steps are followed, to delete a node from the specified position in the list.

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below.

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


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
        ctr = 1;
        while(ctr < pos)
        {
                temp = temp -> right;
                ctr++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
}
```



Deleting a node at the specified position

TRAVERSAL AND DISPLAYING A LIST

- ✓ To display the list, we have to traverse (move) the double linked list, node by node from the first node, until the end of the list is reached.
- ✓ To traverse double linked list from left to rightwe have the following steps:

  1. If list is empty then display **'Empty List'** message.
  2. If the list is not empty, follow the steps given below:

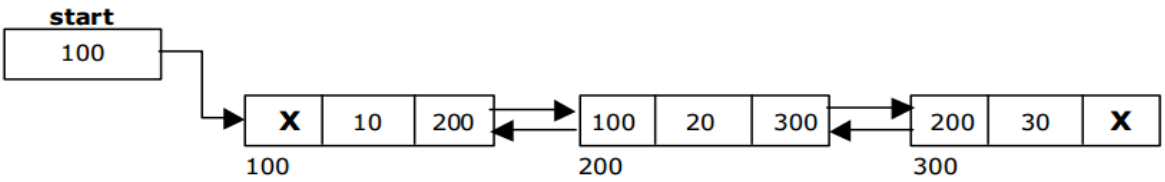
```
        temp = start;
        while(temp != NULL)
        {
                printf("%d", temp -> data);
                temp = temp -> right;
}
```

- ✓ To display the list, we have to traverse (move) the double linked list, node by node from the first node, until the end of the list is reached.

- ✓ The following steps are followed, to traverse a list from left to right:

  1. If list is empty then display 'Empty List' message.
  2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp!= NULL)
{
        printf("%d", temp -> data);
        temp = temp -> right;
}
```



Double Linked List with 3 nodes

COUNTING THE NUMBER OF NODES

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

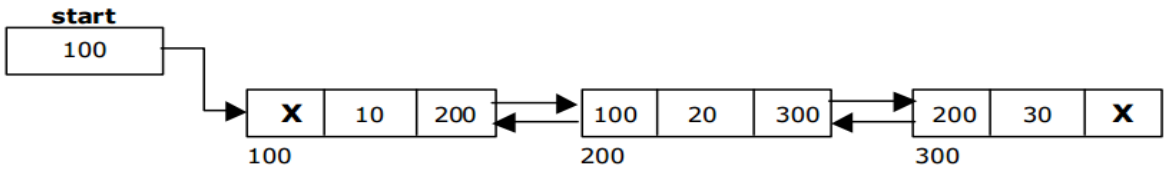
```
int countnode(node *start)
{
        if(start = = NULL)
                return 0;
        else
                return(1 + countnode(start ->right ));
}
```

SEARCHING A NODE IN A DOUBLE LINKED LIST

- ✓ Searching a double linked list means to find a particular element in the double linked list.
- ✓ A double linked list consists of nodes which are divided into two parts, the data part and the next part.
- ✓ So searching means finding whether a given value is present in the data part of the node or not.

- ✓ If it is present, then display element found otherwise element not found.

```
void search()
{
        node *temp;
        int value = 30;
        temp = start;
        if(start == NULL )
                printf(" Empty List ");
        else
        {
                while (temp->right != NULL)
                {
                        if(value = temp->data)
                        {
                                printf(" Element found ");
                                return;
                        }
                        temp = temp -> right;
                }
                printf(" Element not found ");
        }
}
```



Double Linked List with 3 nodes

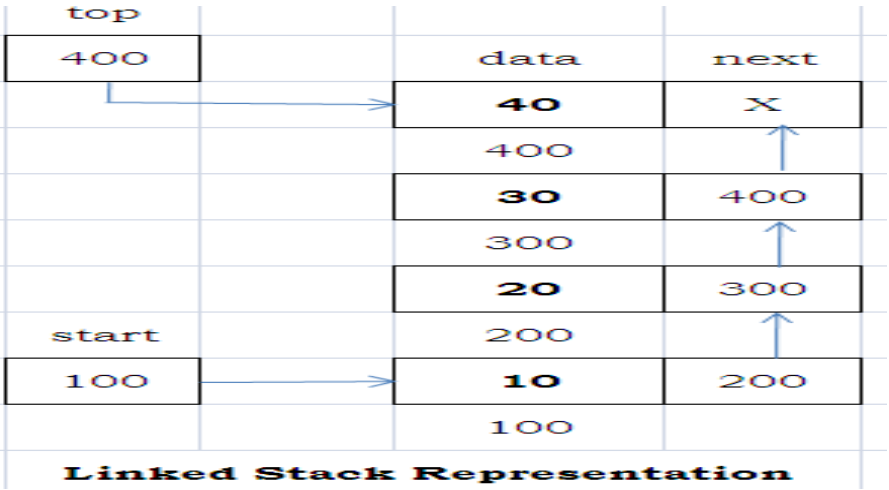
LINKED STACKS

- ✓ 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.
- ✓ The element that is inserted last is 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.

REPRESENTATION OF STACK USING LINKED LIST

- ✓ A stack is represented using an array is easy, but the drawback is that the array must be declared to have some fixed size.
- ✓ In case the stack is a very small or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- ✓ But if the array size cannot be determined in advance, then linked representation is used.
- ✓ The storage requirement of linked representation of the stack with n elements is O(n), and the time requirement for the operations is O(1).
- ✓ In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node.
- ✓ The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP.
- ✓ If TOP = NULL, then it indicates that the stack is empty.



Linked Stack Representation

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATIONS ON LINKED STACKS

- ✓ There are three possible operations performed on a stack. They are push, pop and peek.
  - ✓ **Push: Allows adding an element at the top of the stack.**
  - ✓ **Pop: Allows removing an element from the top of the stack.**
  - ✓ **Peek: it returns the value of topmost element of the stack**

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 Operation

Step 1: Allocate memory for the temporary node and name it as temp

Step 2: Set temp - > data = x

Step 3: if top = NULL

Set temp - > next = NULL

Set top = temp

else

Set temp - > next = top

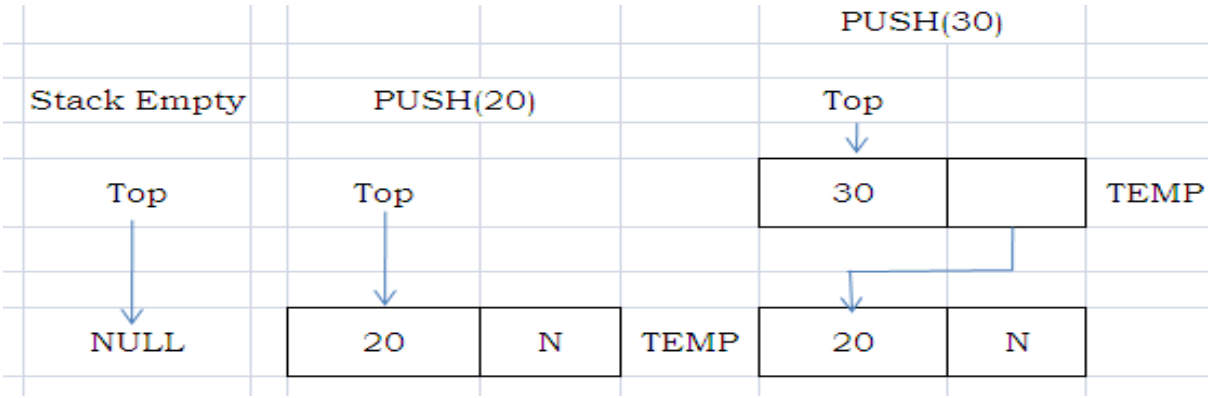
Set top = temp

Step 4: Exit

EXAMPLE

- ✓ The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
- ✓ To insert an element with value 20, we first check if top=NULL. Then we allocate memory for a newnode(temp), store the value in its data part and NULL in its next part.

- ✓ The newnode(temp) will then be called top. However, if top!=NULL, then we insert the newnode(temp) at the beginning of the linked stack and name this newnode(temp) as top.



Pop Operation

- ✓ The data in the topmost node of the stack is first stored in a variable called x.
- ✓ 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

Step 1: if top = NULL

display Underflow and goto step 6

Step 2: Set x = top - > data

Step 3: Set temp = top

Step 4: Set top = top - > next

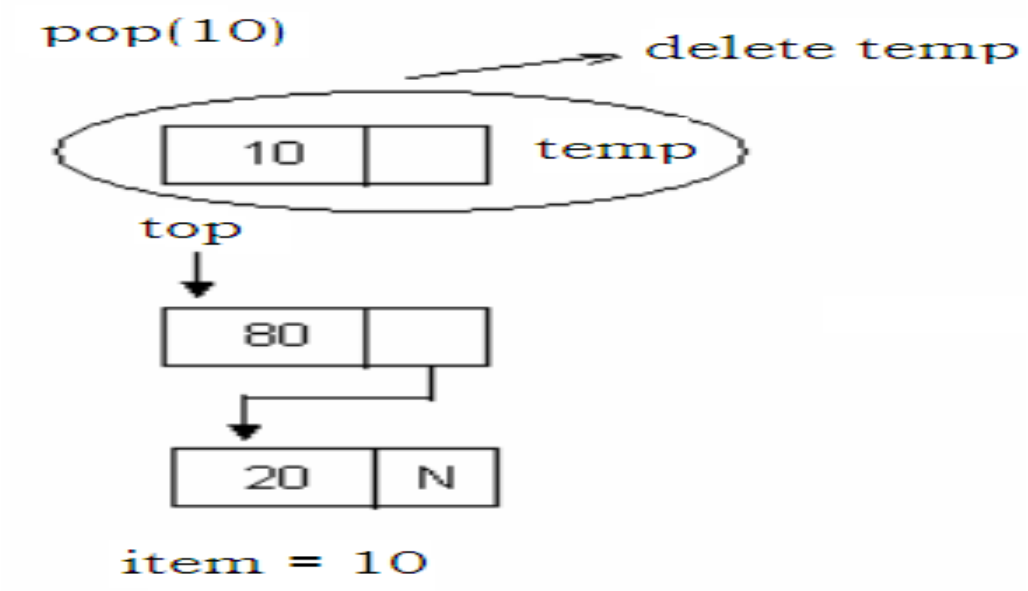
Step 5: free temp

Step 6: Exit

EXAMPLE:

- ✓ The pop operation is used to delete the topmost element from a stack. Before deleting the value, we must first check if top=NULL, then we display stack is empty and no more deletions can be done.
- ✓ If an attempt is made to delete a value from a stack that is already empty, an underflow message is printed.

- ✓ In case top!=NULL, then we will delete the node pointed by top, and make top point to the second element of the linked stack.



IMPLEMENTATION OF STACKS USING LINKED LIST

```
#include<stdio.h>
struct node
{
        int data;
        struct node *next;
}*top = NULL;
void push(int);
void pop();
void display();
int main(void)
{
        int choice, value;
        clrscr();
        printf("\n:: Stack using Linked List ::\n");
        while(1)
        {
                printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
                printf("Enter your choice: ");
```

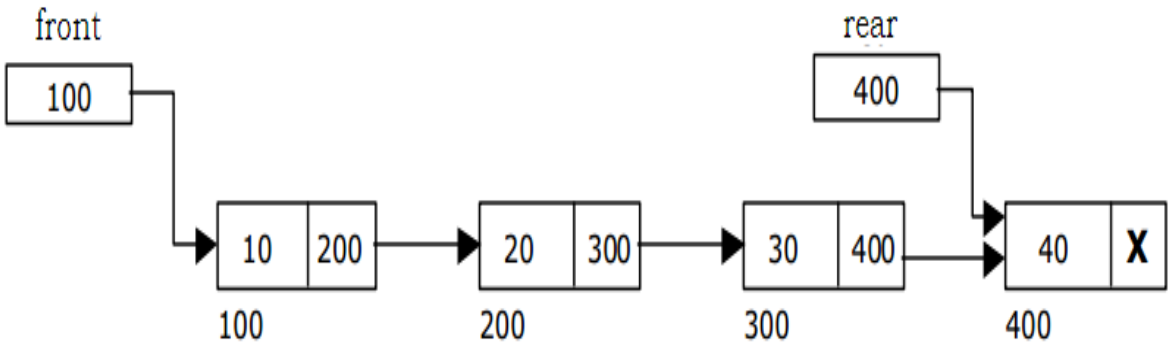
```
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:     printf("Enter the value to be insert: ");
                                    scanf("%d", &value);
                                    push(value);
                                    break;
                        case 2:     pop(); break;
                        case 3:     display(); break;
                        case 4:     exit(0);
                        default:    printf("\nWrong selection!!! Please try
                                    again!!!\n");
                }
        }
}
void push(int value)
{
        struct node *newnode;
        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = value;
        if(top == NULL)
                newnode->next = NULL;
        else
                newnode->next = top;
        top = newnode;
        printf("\nInsertion is Success!!!\n");
}
void pop()
{
        if(top == NULL)
                printf("\nStack is Empty!!!\n");
        else
```

```
            {
                struct node *temp = top;
                printf("\nDeleted element: %d", temp->data);
                top = temp->next;
                free(temp);
            }
}
void display()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else
    {
        struct node *temp = top;
        while(temp->next != NULL)
        {
                printf("%d--->",temp->data);
                temp = temp -> next;
        }
        printf("%d--->NULL",temp->data);
    }
}
```

LINKED QUEUES AND ITS REPRESENTATION

- ✓ 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 into 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 ON QUEUES USING LINKED LIST

Enqueue operation

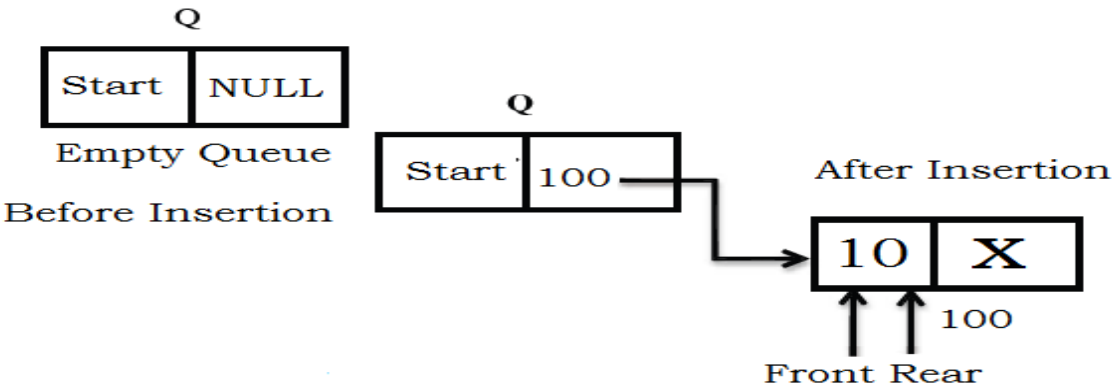
- ✓ In linked list representation of queue, the addition of new element to the queue takes place at the rear end.
- ✓ It is the normal operation of adding a node at the end of a list.

Algorithm for Enqueue(inserting an element)

Allocate memory for the new node and name it as temp

```
set newnode - > data = value
set newnode -> next = NULL
if (front = NULL) then
        set rear = front = newnode
        set rear - > next = front -> next = NULL
else
        set rear - > next = temp
        set rear = rear - > next
        set rear -> next = NULL
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

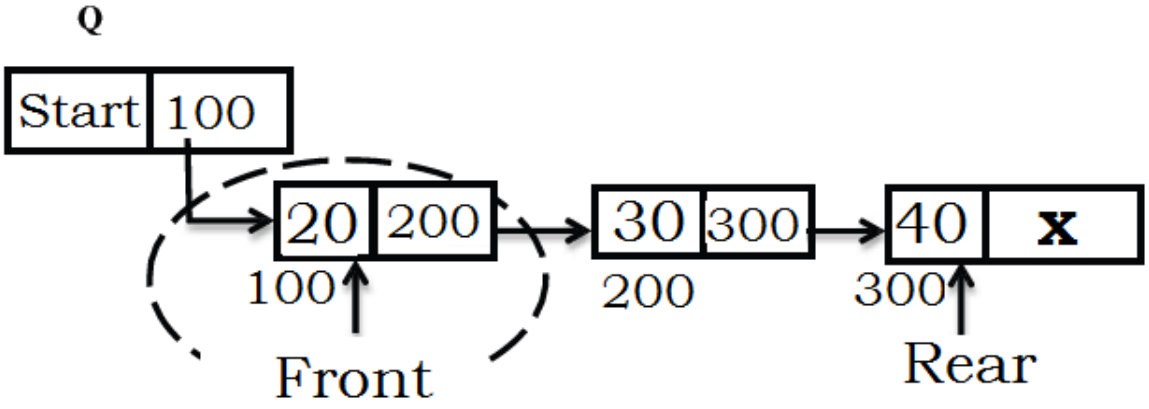


Dequeue operation

- ✓ The dequeue 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
                free(temp)
```



Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IMPLEMENTATION OF QUEUES USING LINKED LIST

```
#include<stdio.h>
#include <stdlib.h>
struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear = NULL;
node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
            rear = newnode;
        }
        else
        {
            rear -> next = newnode;
            rear = newnode;
        }
        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
            printf("\n\n\t Empty Queue..");
            return;
        }
        temp = front;
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp ->data);
        free(temp);
}
void displayQ()
{
        node *temp;
        if(front == NULL)
        {
            printf("\n\n\t\t Empty Queue ");
        }
        else
        {
            temp = front;
```

```
                printf("\n\n\n\t\t Elements in the Queue are: ");
                while(temp != NULL )
                {
                        printf("%5d ", temp -> data);
                        temp = temp -> next;
                }
        }
}
char menu()
{
        char ch;
        clrscr();
        printf("\n \t..Queue operations using pointers.. ");
        printf("\n\t ----------**********----------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}
int main(void)
{
        char ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                        insertQ();
```


Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


```
                    break;
            case '2' :
                    deleteQ();
                    break;
            case '3' :
                    displayQ();
                    break;
            case '4':
                    return;
        }
    } while(ch != '4');
    return 0;
}
```

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:

$5x^2 + 3x + 1$

$12x^3 + 4$

$4x6 + 10x4 – 5x + 3$

$5x^4 – 8x^3 + 2x^2 + 4x^1 + 9$

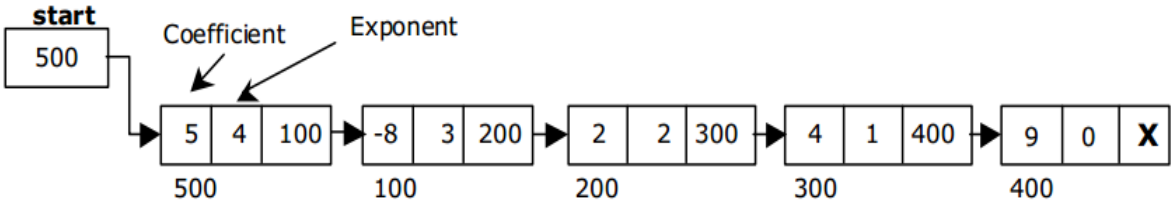
$23x^9 + 18x^7 – 41x^6 + 163x^4 – 5x + 3$

REPRESENTATION OF POLYNOMIALS

- ✓ It is not necessary to write terms of 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.

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

- ✓ 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 back up 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.

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

```
struct node
{
    int coeff;
    int pow;
    struct node *next;
};
void readpolynomial(struct node** poly)
{
    int coeff, exp, mterms;
    struct node* temp = (struct node *) malloc(sizeof(struct node));
    *poly = temp;
    do
    {
            printf("\n Coefficient: ");
            scanf("%d", &coeff);
            printf("\n Exponent: ");
            scanf(%d", &pow);
            temp -> coeff = coeff;
            temp -> pow = exp;
            temp -> next = NULL;
            printf("Have more terms: 1 for Y and 0 for N");
            scanf("%d", &mterms);
            if(mterms)
            {
                    temp -> next = (struct node *) malloc(sizeof(struct node));
                    temp -> next = NULL;
            }
    }while(mterms);
}
void displaypolynomial(struct node* poly)
{
    printf("\n Polynomial Expression is  ");
```

```
    while(poly!=NULL)
    {
        printf("%dX^%d", poly -> coeff, poly -> pow);
        poly = poly -> next;
        if(poly!=NULL)
            printf(" + ");
    }
}
void addpolynomial(struct node**result, struct node* first, struct node*
second)
{
    struct node* temp = (struct node *)malloc(sizeof(struct node));
    temp -> next = NULL;
    *result = temp;
    while(first && second)
    {
        if(first -> pow > second -> pow)
        {
            temp -> coeff = first -> coeff;
            temp -> pow = first -> pow;
            first = first -> next;
        }
        else if(first -> pow < second -> pow)
        {
            temp -> coeff = second -> coeff;
            temp -> pow = second -> pow;
            second = second -> next;
        }
        else
        {
            temp -> coeff = first -> coeff + second -> coeff;
            temp -> pow = first -> pow;
```

```
                first = first -> next;
                second = second -> next;
            }
            if(first && second)
            {
                temp->next = (struct Node*)malloc(sizeof(struct Node));
                temp = temp->next;
                temp->next = NULL;
            }
        }
        while(first || second)
        {
            temp -> next = (struct Node*)malloc(sizeof(struct Node));
            temp  = temp -> next;
            temp -> next = NULL;
            if(first)
            {
                temp -> coeff = first -> coeff;
                temp -> pow = first -> pow;
                first = first -> next;
            }
            else if(second)
            {
                temp -> coeff = second -> coeff;
                temp -> pow = second -> pow;
                second = second -> next;
            }
        }
    }
    int main(void)
    {
        struct node* first = NULL;
```

```
    struct node* second = NULL;
    struct node* result = NULL;
    printf("\nEnter the corresponding data:-\n");
    printf("\nFirst polynomial:\n");
    readpolynomial(&first);
    displaypolynomial(first);
    printf("\nSecond polynomial:\n");
    readpolynomial(&second);
    displaypolynomial(second);
    addpolynomials(&result, first, second);
    displaypolynomial(result);
    return 0;
}
```

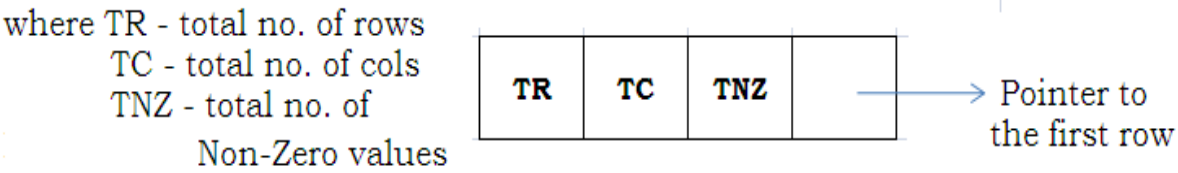
SPARSE MATRIX

- ✓ "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"

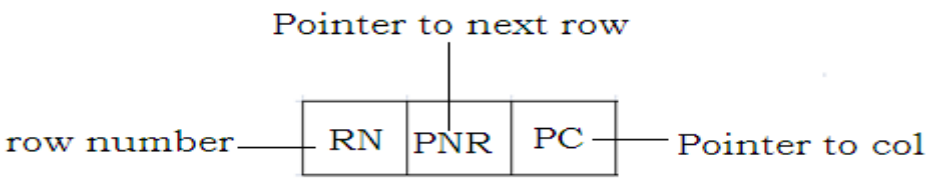
SPARSE MATRIX REPRESENTATION

For linked representation, we need three structures.

1. head node

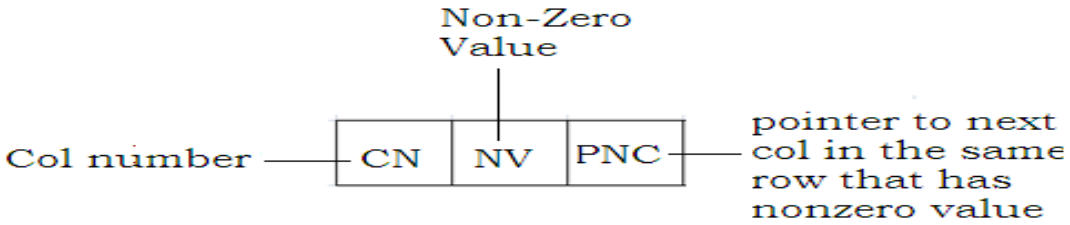


2. row node



Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

3. column node

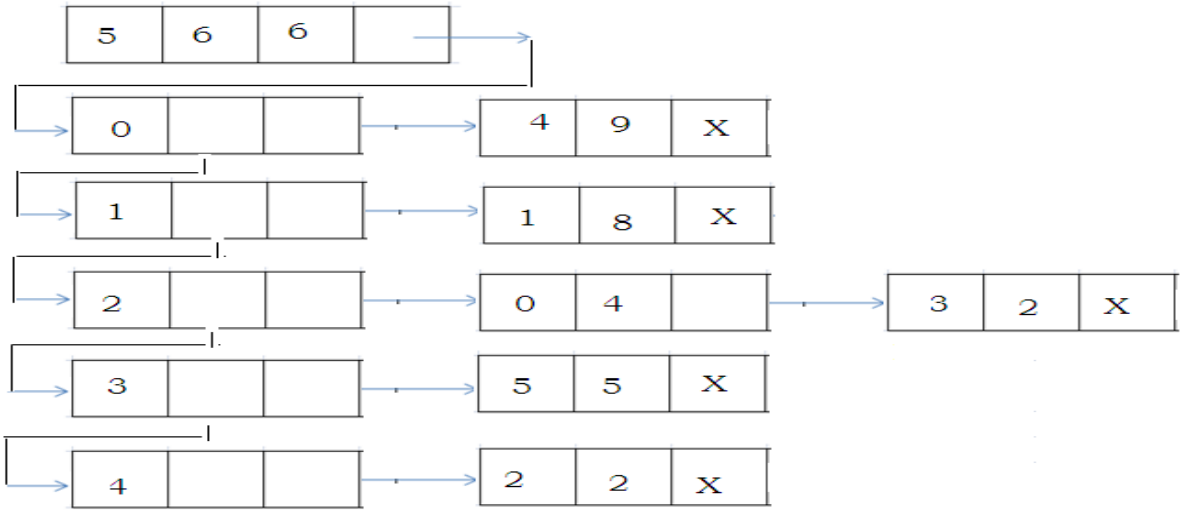


The matrix representation for the sparse matrix is shown below for example.

	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

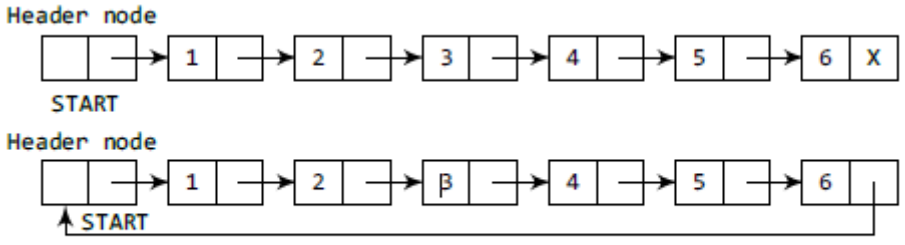
In the above matrix representation there are 5 rows, 6 columns and 6 non-zero values. The linked representation is as follows:



HEADER LINKED LIST

- ✓ A header linked list is a special type of linked list which contains a header node at the beginning of the list.
- ✓ In a header linked list, **START** will not point to the first node of the list but START will contain the address of the header node.

- ✓ The following are the two types of a header linked list:
  - ✓ ***Grounded header linked*** *list* which stores NULL in the next field of the last node.
  - ✓ ***Circular header linked list*** which stores the address of the header node in the next field of the last node. So header node will denote the end of the list.



Header linked list

- ✓ In other linked lists, if **START = NULL**, then it is an empty header linked list.
- ✓ Let us see how a grounded header linked list is stored in the memory. In a grounded header linked list, a node has two fields, DATA and NEXT.
- ✓ The DATA field will store the information part and the NEXT field will store the address of the node in sequence.
- ✓ Note that START stores the address of the header node. The NEXT field of the header node stores the address of the first node of the list.
- ✓ This node stores H. The corresponding NEXT field stores the address of the next node.
- ✓ Hence, we see that the first node can be accessed by writing **FIRST_NODE = START -> NEXT** and not by writing **START = FIRST_ NODE.**
- ✓ Let us now see how a circular header linked list is stored in the memory. The last node in this case stores the address of the header node (instead of –1).
- ✓ Hence, we see that the first node can be accessed by writing FIRST_NODE = START -> NEXT and not writing START = FIRST_NODE.

Algorithm for Insertion

Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 10
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET PTR = START -> NEXT
Step 5: SET NEW_NODE -> DATA = VAL
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7: SET PTR = PTR -> NEXT
Step 8: NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT

Algorithm for Deletion

Step 1: SET PTR = START->NEXT
Step 2: Repeat Steps 3 and 4 while
        PTR DATA != VAL
Step 3: SET PREPTR = PTR
Step 4: SET PTR = PTR -> NEXT
Step 5: SET PREPTR -> NEXT = PTR -> NEXT
Step 6: FREE PTR
Step 7: EXIT

Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

