UNIT – III

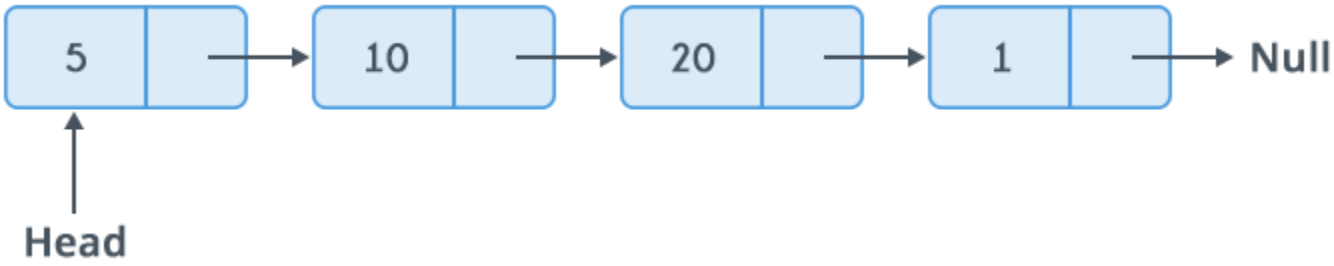
LINKED LIST BASED LINEAR DATA STRUCTURES

A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

**Node**:


A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

**Linked List**:




A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy.

Linked lists allocate memory for each element separately and only when necessary.

- **Advantages of linked lists:**

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

- **Disadvantages of linked lists:**

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

**Types of Linked Lists:**

Basically we can put linked lists into the following four items:

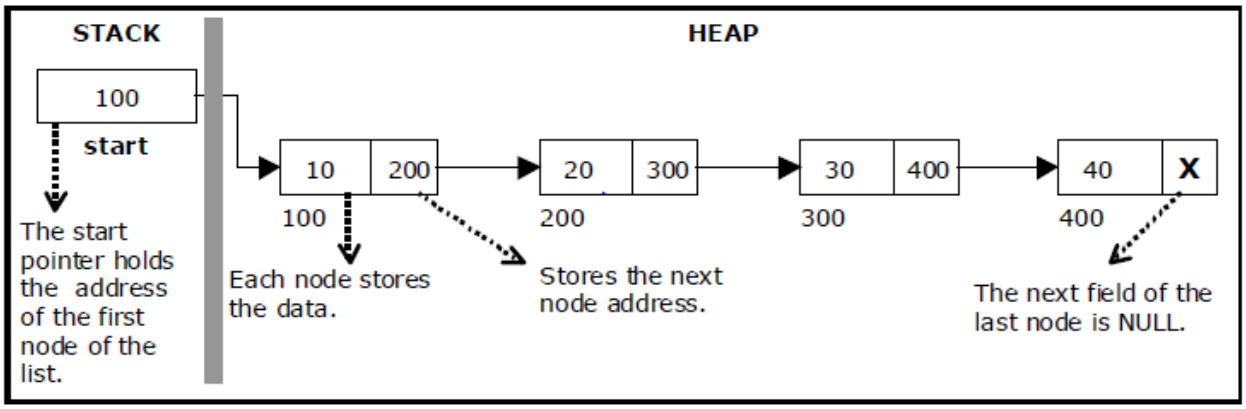
1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

1) A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.
2) A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list.
3) A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

4) A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

**Single 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 like the links in a chain.
- Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the “start” node.
- A single linked list is shown in following figure



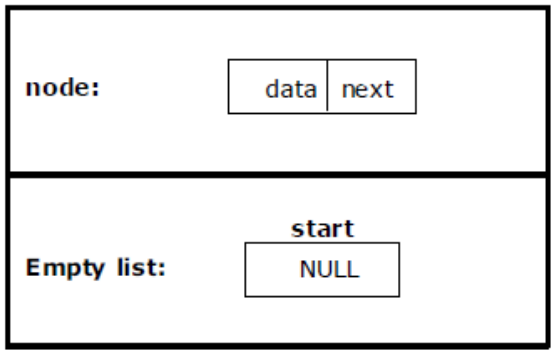

- The beginning of the linked list is stored in a "**start**" pointer which points to the first node.
- The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on.
- The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

**Implementation of Single Linked List:**

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do

- 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.
- Initialise the start pointer to be NULL.

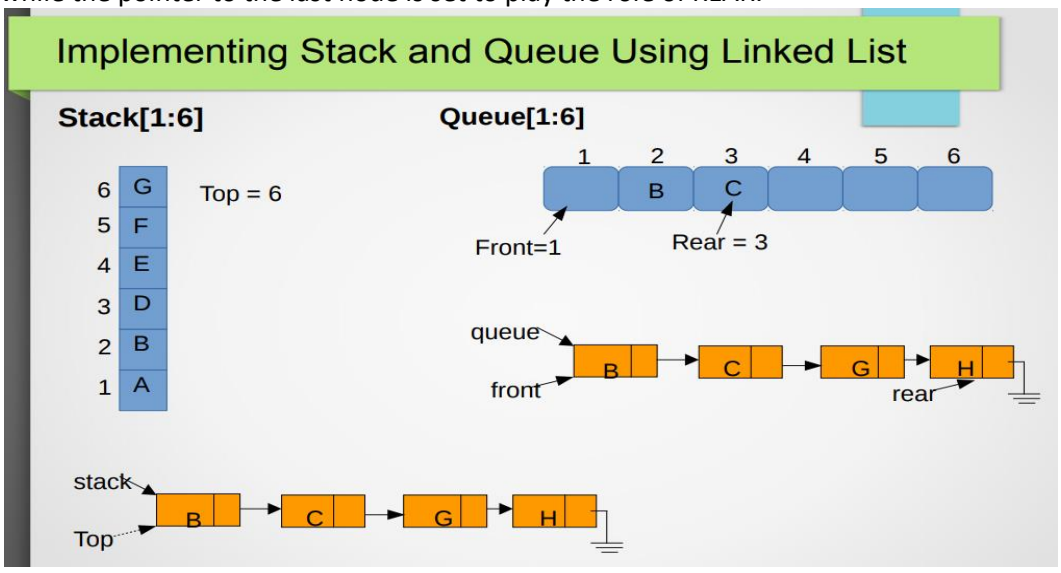
```
struct slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```




## LINKED STACKS AND QUEUES

- A linked stack is a linear list of elements commonly implemented as a singly linked list whose start pointer performs the role of the top pointer of a stack

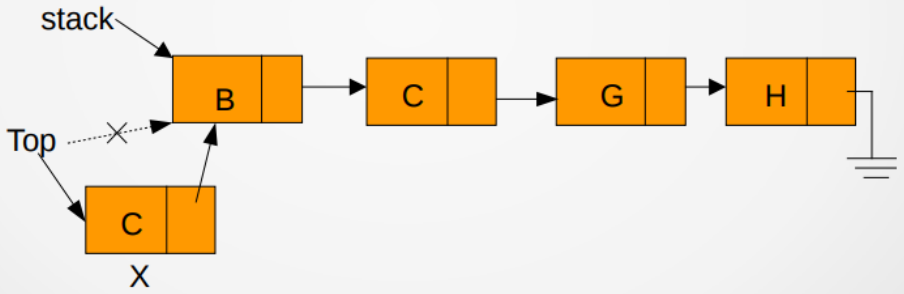
● A linked queue is also a linear list of elements commonly implemented as a singly linked list but with two pointers viz., FRONT and REAR. The start pointer of the singly linked list plays the role of FRONT while the pointer to the last node is set to play the role of REAR.



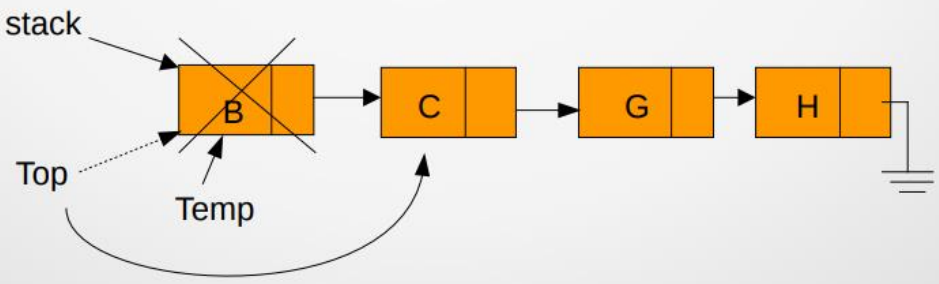
Operations of Linked Stack

## Operations on Linked Stack

- Push
  - GETNODE(X)
  - LINK(X) = Top
  - Top = X

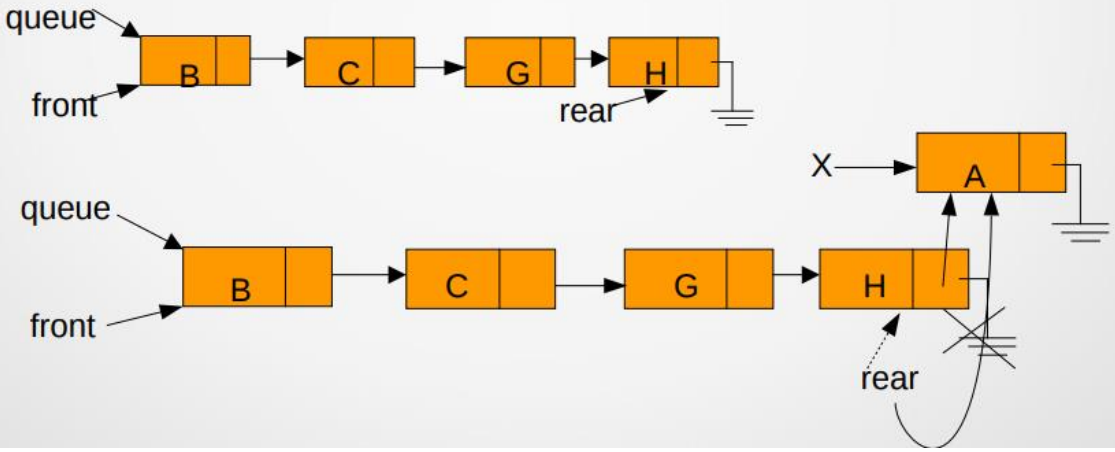



- Pop
  - Temp = Top
  - Item = DATA(Top)
  - Top = LINK(TOP)
  - RETURN(Temp)

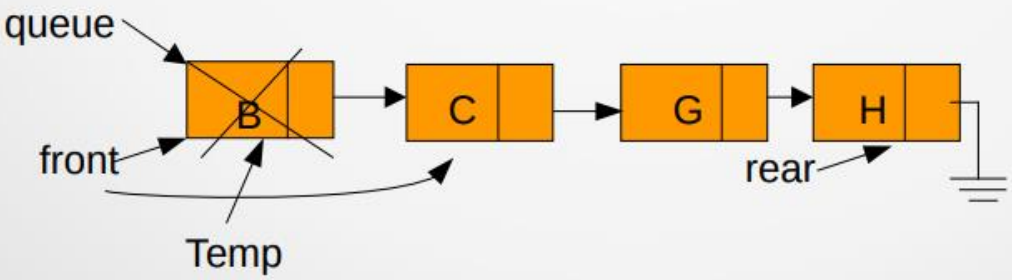


Operations on linked Queue

- Enqueue
  - LINK(rear) = X
  - Rear = X




- Temp = front
- front = Link(front)
- Item = DATA(Temp)
- RETURN(Temp)



POLYNOMIAL ADT

A polynomial is of the form:

Where, ci is the coefficient of the ith term and n is the degree of the polynomial

Some examples are:

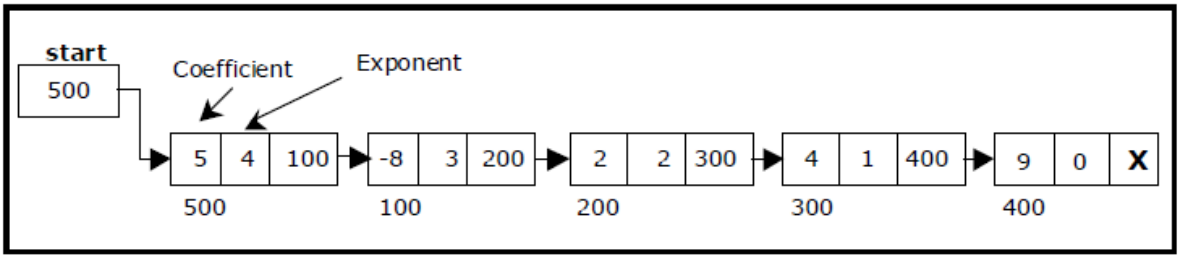
5x2 + 3x + 1

12x3 – 4x

5x4 – 8x3 + 2x2 + 4x1 + 9x0

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. 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 5x4 – 8x3 + 2x2 + 4x1 + 9x0




**Source code for polynomial creation with help of linked list:**

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>
struct link
{
float coef;
int expo;
struct link *next;
};
typedef struct link node;
node * getnode()
```

```
{
node *tmp;
tmp =(node *) malloc( sizeof(node) );
printf("\n Enter Coefficient : ");
fflush(stdin);
scanf("%f",&tmp->coef);
printf("\n Enter Exponent : ");
fflush(stdin);
scanf("%d",&tmp->expo);
tmp->next = NULL;
return tmp;
}
node * create_poly (node *p )
{
char ch;
node *temp,*newnode;
while( 1 )
{
printf ("\n Do U Want polynomial node (y/n): ");
ch = getche();
if(ch == 'n')
break;
newnode = getnode();
if( p == NULL )
p= newnode;
else
{
temp = p;
while(temp->next != NULL )
temp = temp->next;
temp->next = newnode;
}
}
return p;
}
```

```
void display (node *p)
{
node *t = p;
while (t != NULL)
{
printf("+ %.2f", t -> coef);
printf("X^ %d", t -> expo);
t=t -> next;
}
}
void main()
{
node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
clrscr();
printf("\nEnter First Polynomial..(in ascending-order of exponent)");
poly1 = create_poly (poly1);
printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
poly2 = create_poly (poly2);
clrscr();
printf("\n Enter Polynomial 1: ");
display (poly1);
printf("\n Enter Polynomial 2: ");
display (poly2);
getch();
}
```

**Addition of Polynomials:**

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.

**Source code for polynomial addition with help of linked list:**

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>
struct link
{
float coef;
int expo;
struct link *next;
};
typedef struct link node;
node * getnode()
{
node *tmp;}
```

DOUBLY LINKED LISTS

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 contains three fields:

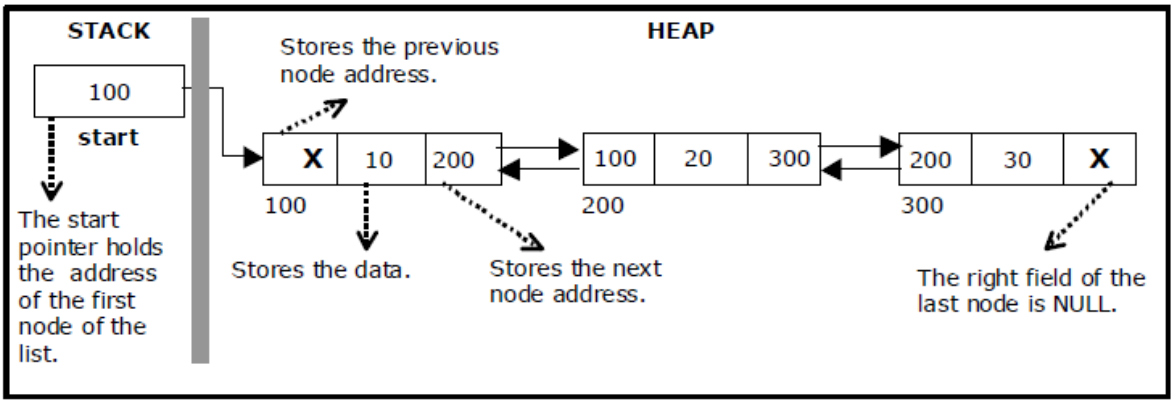
- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

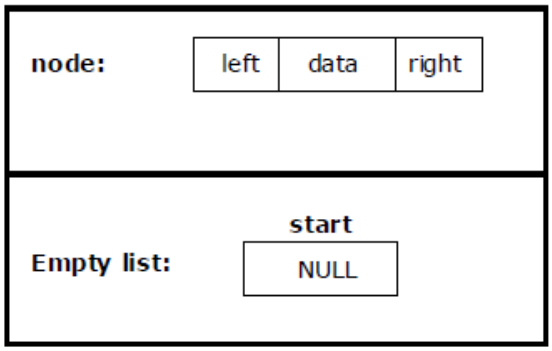
A double linked list




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. The following code gives the structure definition:

.

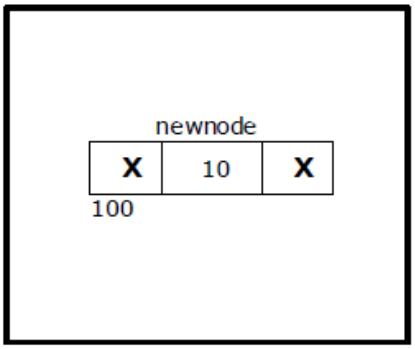
```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

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




**Creating a node for Double 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 structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL.

```
node* getnode()
{
      node* newnode;
      newnode = (node *) malloc(sizeof(node));
      printf("\n Enter data: ");
      scanf("%d", &newnode -> data);
      newnode -> left = NULL;
      newnode -> right = NULL;
      return newnode;
}
```

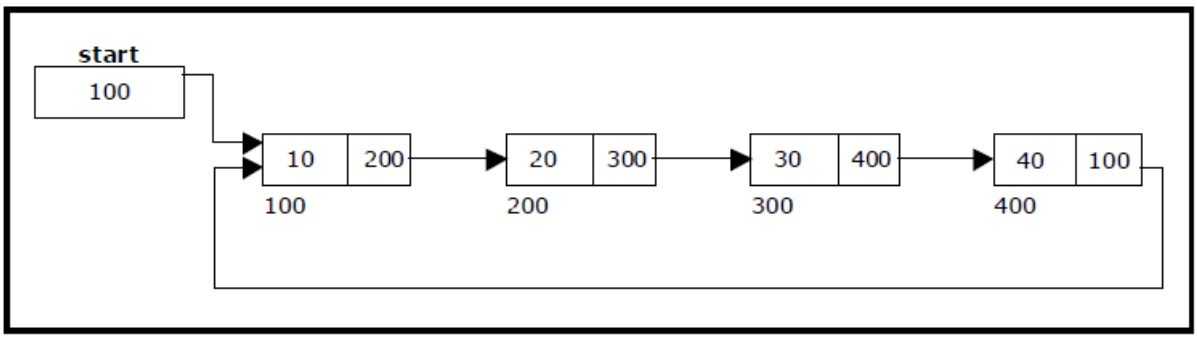



**Circular Single Linked List:**

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

It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure



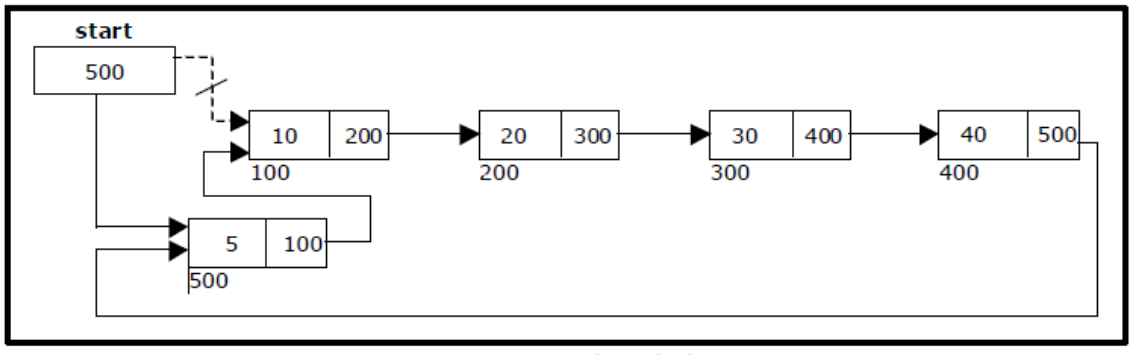

The basic operations in a circular single linked list are:
- Creation.
- Insertion.
- Deletion.
- Traversing.

**Creating a circular single Linked List with 'n' number of nodes:**
The following steps are to be followed to create 'n' number of nodes:

• Get the new node using getnode().
newnode = getnode();
• If the list is empty, assign new node as start.
start = newnode;
• If the list is not empty, follow the steps given below:
temp = start;
while(temp -> next != NULL)
temp = temp -> next;
temp -> next = newnode;
• Repeat the above steps ‘n’ times.
• newnode -> next = start;
The function createlist(), is used to create ‘n’ number of nodes:

**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:

• Get the new node using getnode().
newnode = getnode();
• If the list is empty, assign new node as start.
start = newnode;
newnode -> next = start;
• 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;



**Deleting a node at the beginning:**

The following steps are followed, to delete a node at the beginning of the list:
• If the list is empty, display a message 'Empty List'.
• 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;
• After deleting the node, if the list is empty then *start = NULL.*
The function cll_delete_beg(), is used for deleting the first node in the list.