Chapter - 06 : "Linked List"

MEMORY ALLOCATIONS IN C

There are two types of memory allocations possible in C:

(a) Static memory allocation (Compile-time allocation using arrays)
(b) **Dynamic memory allocation (Run-time allocation using pointers)**

Static memory allocation (using Arrays)

In **Static memory allocation**, the required amount of memory is allocated to the program elements at the start of the program. Here the memory to be allocated to the variable is fixed and is determined by the compiler at the compile time (if it is a single integer variable it allocates two bytes to it, if it is an array of five integer values it allocates ten bytes to it and if it is a single float type of variable compiler allocates four bytes to it. For example, consider the following declaration:

```
int x, y;
float A [5];
```

When the first statement is encountered, the compiler will allocate two bytes to each variable x and y of type int. The second statement results into the allocation of 20 bytes to the array **A** (5 * 4, where there are five elements and each element of float type takes four bytes). Note that as there is no bound checking in C for array boundaries, i.e., if you have declared an array of five elements, as above and by mistake you are intending to read more than five values in the array **A**, it will still work without error. For example, you are reading the above array as follows:

```
for (i=0; i<10; i++)
{
 scanf ("%f", &A[i]);
}
```

Though you have declared size of array 5, you are trying to read ten elements. However, the problem with this is that the extra elements added as a part of this array **"A"** are not allocated the consecutive memory location after the five elements, i.e. only the first five elements are stored in consecutive memory locations and the other elements are stored randomly at any unknown locations in the memory. Thus during accessing these extra elements would not be made available to the user, only the first five values can be accessible.

The second problem with static memory allocation is that if you store less number of elements than the number of elements for which you have declared memory, then the rest of the memory will be wasted (i.e. it is not made available to other applications and its status is set as allocated and not free). This leads to the wastage of memory.

Dynamic memory allocation (using pointers)

The concept of **Dynamic or Run-time memory allocation** helps us to overcome this problem in arrays, as well as allows us to be able to get the required portion of memory at run-time (or we say as the need arises). This is best suited type of allocation where we do not know the memory requirement in advance, which is the case with most of real-life problems. In other words, dynamic memory allocation gives flexibility for programmer. As well as it makes efficient use of memory by allocating the required amount of memory whenever needed, unlike static allocation where we declare the amount of memory to be allocated statically.

- **malloc ( ) function**

The **malloc( )** function allocates a block of memory in bytes. The user should explicitly give the block size it requires for the user. The **malloc( )** function is like a request to the RAM of the system to allocate memory, if the request is granted (i.e. if **malloc( )** function stays successful in allocating memory), returns a pointer to the first block of that memory. The type of the pointer it returns is **void**, which means that we can assign it any type of pointer. However if the **malloc( )** function fails to allocate the required amount of memory, it returns a **NULL**. The syntax of this function is as follows:

malloc (number of elements * size of each element);

For example,

```
int *ptr;
ptr= malloc (10 * size of each element);
```

where **size** represents the size of memory required in bytes (i.e. number of contiguous memory locations to be allocated). But the function **malloc ( )** returns a void pointer so a cast operator is required to change the returned pointer type according to our need, the above declaration would take the following form:

```
ptr_var = (type_cast *) malloc (size);
```

where **ptr_var** is the name of pointer that holds the starting address of allocated memory block, **type_cast** is the data type into which the returned pointer (of type void) is to be converted, and **size** specifies the size of allocated memory block in bytes.
For example,

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

After the execution of this statement, a consecutive memory blocks of 20 bytes (size of integer is 2 bytes, and we are requesting for 10 elements of type integer, therefore, 10 * 2 = 20 bytes) is allocated to the program. The address of first byte of this block is first converted into int and then assigned to the pointer ptr. Consider another example for type char,

```
char *ptr;
ptr = (char *) malloc (10 * sizeof (char));
```

Similarly, for allocation of memory to a structure variable, the following statements are required:

```
struct student
{
 char name[30];
 int roll_no;
 float percentage;
};
struct student *st_ptr;
```

For allocating of memory, the following statement is required:

```
st_ptr = (struct student *) malloc (sizeof (struct student));
```

After the execution of this statement, a contiguous block of memory of size 36 bytes (30 bytes for char type name, 2 bytes for integer roll_no and 4 bytes for float percentage) is allocated to **st_ptr**;

Note that to check that if the request made through the function **malloc ( )** to allocate the memory is rejected by system RAM (in case if required space is not available), the **malloc ( )** function returns a **NULL**. This can be done as follows:

```
int *ptr;
ptr = (int *) malloc (5 * sizeof (int));
if (ptr = = NULL)
{
 printf ("\nThe required amount of memory is not available");
 getch ();
 exit (0);
}
```

- **free ( ) function**

The **free ( )** function is used to de-allocate the previously allocated memory using malloc () functions. The syntax of this function is:

free (ptr_var);

where **ptr_var** is the pointer in which the address of the allocated memory block is assigned. The free function is used to return the allocated memory to the system RAM.

Need for Dynamic Data structures

The simplest one of data structures i.e. an array can be used only when their numbers of elements along with elements sizes are predetermined, since the memory is reserved before processing. For this reason, arrays are called static data structures. Now, consider a situation where exact numbers of elements are not known. In such a case, estimates might go wrong. Also during processing, i.e., either more memory is required or extra free spaces lie in the array. Another problem associated with arrays is complexity involved in insertions and deletions of elements.

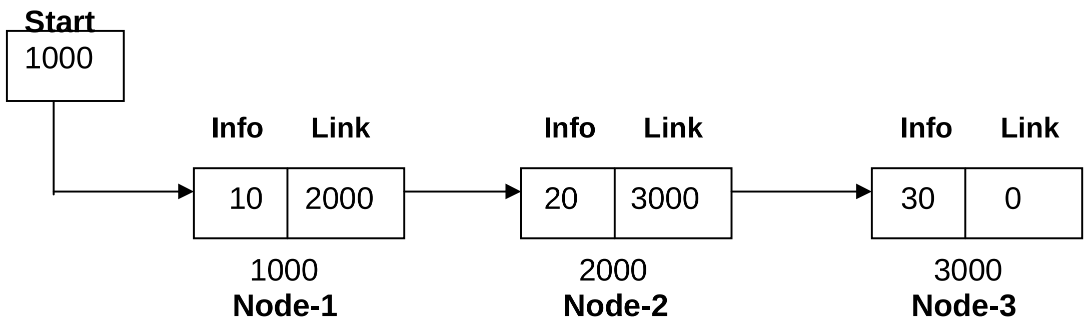
Linked lists(Dynamic data structures) overcome the drawbacks of arrays as in linked lists number of elements need not be predetermined, more memory can be allocated or released during the processing as and when required, making insertions and deletions much easier and simpler.

Dynamic memory allocation technique facilitates allocation of memory during the program execution itself using **malloc( )** function as and when required. Dynamic memory allocation also facilitates release of memory using **free( )** function, if memory is not required any more. Data structures like linked lists and trees use this technique for their memory allocation.

▪ LINKED LIST :

"***A linked list is a linear collection of data elements, called node pointing to the next nodes by means of pointers.***"

Each **node** is divided into two parts: the first part containing the information of the element, and the second part called the link or next pointer containing the address of the next node in the list.



OPERATIONS ON LINKED LISTS

The basic operations to be performed on the linked lists are as follows :

1. Creation **2. Insertion** **3. Deletion** **4. Traversing**
5. Searching **6. Concatenation** **7. Display**

Creation:

This operation is used to create a linked list.

Insertion:

This operation is used to insert a new node in the linked list at the specified position. A new node may be inserted

- At the beginning of a linked list
- At the end of a linked list
- At the specified position in a linked list.
- If the list itself is empty, then the new node is inserted as a first node.

Deletion:

This operation is used to delete a node from the linked list. A node may be deleted from the

- Beginning of a linked list
- End of a linked list.
- Specified position in the linked list.

Traversing:

It is a process of going through all the nodes of a linked list from one end to the other end. It we start traversing from the very first node towards the last node, it is called forward traversing. It we start traversing from the very last node towards the first node, it is called reverse traversing.

Searching:

If the desired element is found, we signal operation "SUCCESSFULL". Otherwise, we signal it as "UNSUCCESSFULL".

Concatenation:

It is a process of appending (joining) the second list to the end of the first list consisting of m nodes. When we concatenate two lists, the second list has n nodes, then the concatenated list will be having (m + n) nodes. The last node of the first linked list is modified so that it is now pointing to the first node in the second list.

Display:

This operation is used to print each and every node's information. We access each node from the beginning (or the specified position) of the list and output the data stored there.

TYPES OF LINKED LISTS

Basically, we can put linked lists into the following four types:

- ***Singly linked list (Linear linked list)***
- ***Doubly linked list***
- ***Circular linked list***
- ***Circular doubly linked list***

1. Singly linked list

A **singly linked list** is one in which all nodes are linked together in some sequential manner. Hence, it is also called **linear linked list**. Clearly, it has the beginning and the end. The problem with this list is that we cannot access the *predecessor node* (previous node) from the current node. This can be overcome by doubly linked lists. A linear linked list is shown in Fig. (1).

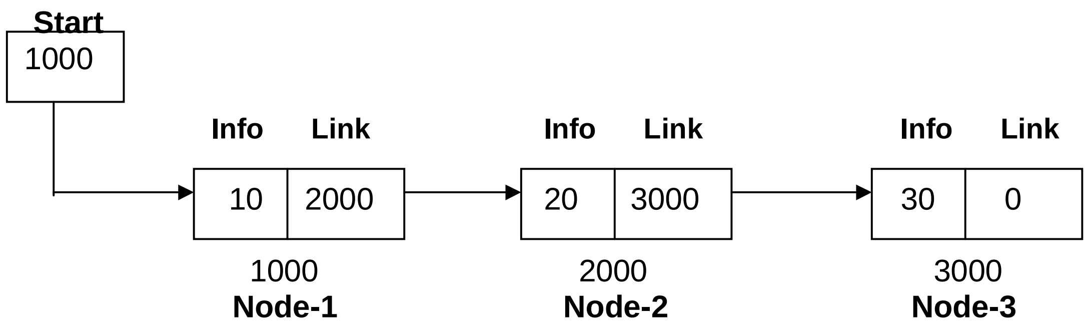


Fig. (1). A Singly (Linear) Linked list

2. Doubly linked list

A **Doubly linked list** is one in which all nodes are linked together by multiple links which help in accessing both the *successor node* (next node) and *predecessor node* (previous node) for any random node within the list. Therefore, each node in a doubly linked list points to the left node (previous) and the right node (next). This helps to traverse the list in the forward direction and backward direction. A doubly linked list is shown in Fig. (2).

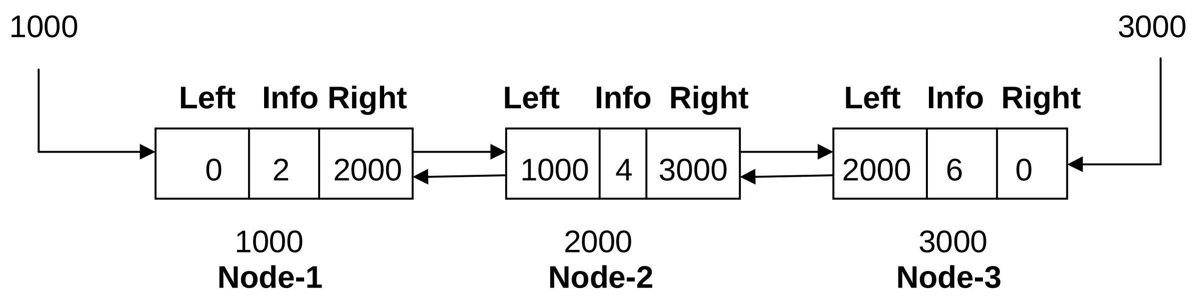


Fig. (2). A Doubly Linked list

3. <u>Circular linked list</u>

A **circular linked list** is one which has no beginning and no end. A singly linked list can be made circular linked list by simply storing the address of the very first node in the link field of the last node. A circular linked list is shown in Fig. (3).

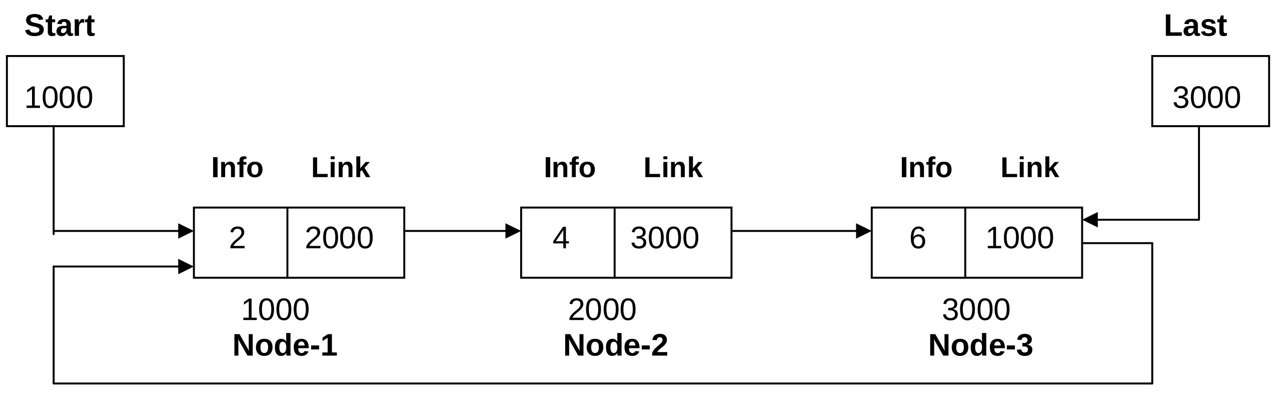


Fig. (3). A Circular Linked list

4. Circular doubly linked list

A **circular doubly linked list** is one which has both the successor pointer and predecessor pointer in circular manner. A Circular doubly linked list is shown in Fig. (4).

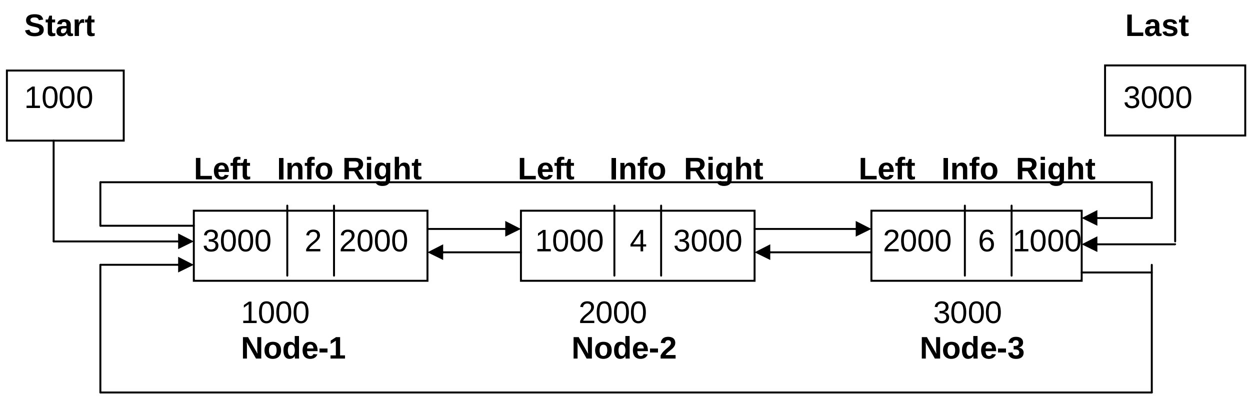


Fig. (4). A Circular Doubly Linked List

■ OPERATIONS OF SINGLY LINKED LISTS

(A) INSERTION

(I) Algorithm to insert node at the beginning of the singly linked list

Let PTR is the structure pointer which allocates memory for the new node at the beginning of the singly linked list & NUM is the element to be inserted into the linked list, INFO represents the information part of the new node and LINK represents the link or next pointer of the new node pointing to the address of next node. START represents the address of first node. Initially, before inserting first node in the singly linked list, START=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into singly linked list.
Step 3 : Set PTR->INFO = NUM
Step 4 : Set PTR->LINK=START;
Step 5 : Set START:=PTR
Step 6 : Exit
```

<u>Function to insert node at the beginning of the singly linked list</u>

```
void insert_beg()
{
 struct list *ptr;
 int num;
 ptr=(list *)malloc(sizeof(list));
 printf("\nEnter the element to be inserted in singly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->link=start;
 start=ptr;
}
```

(II) <u>Algorithm to insert node at the end of the singly linked list</u>

Let PTR is the structure pointer which allocates memory for the new node at the end of the singly linked list, TEMP is a structure pointer to modify the link part of the previous node & NUM is the element to be inserted into the linked list, INFO represents the information part of the new node and LINK represents the link or next pointer of the new node pointing to the address of next node. START represents the address of first node. Initially, before inserting first node in the singly linked list, START=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into singly linked list.
Step 3 : Set PTR->INFO = NUM
Step 4 : Set PTR->LINK := NULL
Step 5 : If START = NULL : then
              Set START := PTR.
         Else
           Set TEMP := START;
           Repeat while TEMP->LINK !=NULL
              Set TEMP := TEMP->LINK.
           [End of loop]
         Set TEMP-> LINK := PTR.
         [End of If Else Structure]
Step 6 : Exit
```

Function to insert node at the end of the singly linked list

```
void insert_end()
{
 struct list *ptr,*temp;
 int num;
 ptr=(list *)malloc(sizeof(list));
 printf("\nEnter the element to be inserted in singly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->link=NULL;
 if (start==NULL)
  start=ptr;
 else
 {
  temp=start;
  while(temp->link != NULL)
     temp=temp->link;
  temp->link=ptr;
 }
}
```

(II) Algorithm to insert node at a specific location in the singly linked list

Let PTR is the structure pointer which allocates memory for the new node at a specific location in a singly linked list, TEMP is a structure pointer to modify the link part of the previous node & NUM is the element to be inserted into the linked list, LOC is the location of the node to be inserted, INFO represents the information part of the new node and LINK represents the link or next pointer of the new node pointing to the address of next node. START represents the address of first node. Initially, before inserting first node in the singly linked list, START=NULL, F=0.

```
Step 1 : Read the location LOC where you want to insert the node.

Step 2 : If LOC > 1 : then
              Set TEMP := START.
                Repeat loop for I = 1 to LOC – 1
                  TEMP := TEMP->LINK.
                   If TEMP := NULL : then
                      Set F :=1 and Exit from loop.
                   [End of If Statement]
                [End of loop]
              If (F=1 OR (START=NULL AND LOC>1)) : then
                Write : “Total nodes in list are lesser than this position.”
                Write : “So Node cannot be inserted.” and return.
              [End of If structure]
          [End of Step 2 If Structure]
```

```
 Step 3 : Allocate memory for the new node using PTR.
Step 4 : Read NUM to be inserted into singly linked list.
Step 5 : Set PTR->INFO = NUM
Step 6 : If LOC :=1 : then
               Set PTR->LINK := START.
               Set START := PTR.
             Else
               Set PTR->LINK := TEMP->LINK.
               TEMP->LINK := PTR.
             [End of If Else Structure].
Step 7 : Exit
```

Function to insert node at specific location in the singly linked list

```
void insert_spe()
{
 struct list *ptr,*temp;
 int i,num,loc,f=0;
 printf("\nEnter the location where you want to store the value : ");
 scanf("%d",&loc);
 if(loc>1)
 {
 temp=start;
 for(i=1; i<loc-1;i++)
 {
  temp=temp->link;
   if(temp==NULL)
   {
    f=1;
    break;
   }
 }
 if(f==1 || (start==NULL && loc>1))
 {
    printf("\nTotal nodes in the list are lesser than this position. ");
    printf("So node cannot be inserted.\n");
    return;
 }
}
 ptr=(list *)malloc(sizeof(list));
 printf("\nEnter the element to be inserted in singly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(loc==1)
 {
  ptr->link=start;
  start=ptr;
 }
 else
 {
  ptr->link=temp->link;
  temp->link=ptr;
 }
}
```

(B) DELETION

(IV) Algorithm to delete node from the beginning of the singly linked list

Let PTR is the structure pointer which deallocates memory of the node at the beginning of the singly linked list & NUM is the element to be deleted from the linked list, INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node. START represents the address of first node.

```
Step 1 : If START := NULL : then
                Write : “List is empty” and return.
Step 2 : Set PTR := START.
Step 3 : Set NUM := PTR->INFO.
Step 4 : Write : “Deleted element from the beginning of the singly linked list : “,NUM.
Step 5 : Set START := START->LINK.
Step 6 : Deallocate memory of the node at the beginning of singly linked list using PTR.
Step7 : Exit
```

Function to delete node from the beginning of the singly linked list

```
void delete_beg()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 num=ptr->info;
 printf("\nThe deleted element from the beginning of singly linked list is : %d",num);
 start=start->link;
 free(ptr);
}
```

(V) Algorithm to delete node from the end of the singly linked list

Let PTR is the structure pointer which deallocates memory of the node from the end of the singly linked list, TEMP is a structure pointer to modify the LINK part of previous node & NUM is the element to be deleted from the linked list, INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node. START represents the address of first node.

```
Step 1 : If START := NULL : then
            Write : “List is empty” and return.
Step 2 : If START->LINK := NULL : then
           Set PTR := START.
           Set START := NULL.
         Else
            Set TEMP := START.
            Set PTR := PTR->LINK.
               Repeat loop while PTR->LINK!=NULL
                  Set TEMP := PTR.
                  Set  PTR := PTR->LINK.
               [End of loop]
            Set TEMP->LINK := NULL.
         [End of Step 2 If Else Structure]
Step 3 : Set NUM := PTR->INFO.
Step 4 : Write : “Deleted element from the end of the singly linked list : “,NUM.
Step 5 : Deallocate memory of the node at the end of singly linked list.
Step 6 : Exit
```

Function to delete node from the end of the singly linked list

```
void delete_end()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr,*temp;
 int num;
 if(start->link==NULL)
 {
  ptr=start;
  start=NULL;
 }
 else
 {
  temp=start;
  ptr=start->link;
  while(ptr->link!=NULL)
  {
   temp=ptr;
   ptr=ptr->link;
  }
  temp->link=NULL;
 }
 num=ptr->info;
 printf("\nThe deleted element from the singly linked list is : %d",num);
 free(ptr);
}
```

(V) <u>Algorithm to delete node from a specific location in the singly linked list</u>

Let PTR is the structure pointer which deallocates memory of the node from a specific location in the singly linked list, TEMP is a structure pointer to modify the LINK part of previous node & NUM is the element to be deleted from the linked list, INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node. START represents the address of first node.

```
Step 1 : If START := NULL : then
            Write : “List is empty” and return.
Step 2 : Set PTR := START.
Step 3 : Read the location LOC from where you want to delete the node.
Step 4 : If LOC := 1 : then
           START := PTR->LINK
         Else
             Repeat loop for I = 1 to LOC.
               Set TEMP := PTR.
               Set PTR := PTR->LINK.
                 If PTR = NULL : then
                   Write : “Total nodes in the list are lesser than this position”.
                   Write : “So node cannot be deleted” and return.
                 [End of If structure]
             [End of loop]
           Set TEMP->LINK := PTR->LINK.
         [End of Step 4 If Else structure]
Step 5 : Set NUM := PTR->INFO.
Step 6 : Write : “Deleted element from the singly linked list : “,NUM.
Step 7 : Deallocate memory of the node at the location LOC of singly linked list.
Step 8 : Exit
```

Function to delete node from a specific location in singly linked list

```
void delete_spe()
{
if(start==NULL)
{
 printf("\nList is empty\n");
 return;
}
struct list *ptr,*temp;
int num, loc;
ptr=start;
printf("\nEnter the location from where you want to delete the value : ");
scanf("%d",&loc);
 if(loc==1)
   start=ptr->link;
 else
 {
  for(int i=1;i<loc;i++)
  {
  temp=ptr;
  ptr=ptr->link;
   if(ptr==NULL)
   {
   printf("\nTotal nodes in the list are lesser than this position. ");
   printf("\nSo node cannot be deleted\n");
   return;
   }
  }
 temp->link=ptr->link;
 }
 num=ptr->info;
 printf("\nThe deleted element from the singly linked list is : %d",num);
 free(ptr);
}
```

PROGRAM – 1 : SINGLY LINKED LIST OPERATIONS

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

struct list
{
 int info;
 struct list *link;
};

struct list *start;

void initialize();
void insert_beg();
void insert_end();
void insert_spe();
void traverse();
void delete_beg();
void delete_end();
void delete_spe();

void main()
{
int choice;
initialize();
while(1)
{
 clrscr();
 printf("  IMPLEMENTATION OF LINKED LIST \n");
 printf("-----------------------------------\n");
 printf("1. Insertion at the beginning of list \n");
 printf("2. Insertion at the end of list \n");
 printf("3. Insertion at the specific location in list \n");
 printf("4. Deletion from beginning \n");
 printf("5. Deletion from end \n");
 printf("6. Deletion from specific location \n");
 printf("7. Traverse the list \n");
 printf("8. Exit \n");
 printf("-----------------------------------\n");
 printf("\nEnter your choice [1/2/3/4/5/6/7/8] : ");
 scanf("%d",&choice);
 switch(choice)
 {
 case 1 : insert_beg();
            break;
 case 2 : insert_end();
            break;
 case 3 : insert_spe();
            break;
 case 4 : delete_beg();
            break;
 case 5 : delete_end();
            break;
 case 6 : delete_spe();
            break;
 case 7 : traverse();
            break;
 case 8 : exit(0);
 default : printf("\nYou entered wrong choice. ");
 }
 getch();
 }
}
```

// Function to initialize Singly linked list

```
void initialize( )
{
  start=NULL;
}
```

// Function to insert node at the beginning of the singly linked list

```
void insert_beg()
{
 struct list *ptr;
 int num;
 ptr=(list *)malloc(sizeof(list));
 printf("\nEnter the element to be inserted in singly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->link=start;
 start=ptr;
}
```

```
// Function to insert node at the end of the singly linked list
void insert_end()
{
 struct list *ptr,*temp;
 int num;
 ptr=(list *)malloc(sizeof(list));
 printf("\nEnter the element to be inserted in singly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->link=NULL;
 if(start==NULL)
  start=ptr;
 else
 {
  temp=start;
  while(temp->link!=NULL)
     temp=temp->link;
  temp->link=ptr;
 }
}
// Function to insert node at specific location of the singly linked list
void insert_spe()
{
 struct list *ptr,*temp;
 int i,num,loc,f=0;
 printf("\nEnter the location where you want to store the value : ");
 scanf("%d",&loc);
 if(loc>1)
 {
 temp=start;
 for(i=1;i<loc-1;i++)
 {
  temp=temp->link;
   if(temp==NULL)
   {
    f=1;
    break;
   }
 }
 if(f==1 || (start==NULL && loc>1))
 {
    printf("\nTotal nodes in the list are lesser than this position. ");
    printf("So node cannot be inserted.\n");
    return;
 }
}
```

```
 ptr=(list *)malloc(sizeof(list));
 printf("\nEnter the element to be inserted in singly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(loc==1)
 {
  ptr->link=start;
  start=ptr;
 }
 else
 {
  ptr->link=temp->link;
  temp->link=ptr;
 }
}
```

// Function to delete node from the beginning of the singly linked list

```
void delete_beg()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 num=ptr->info;
 printf("\nThe deleted element from the singly linked list is : %d",num);
 start=start->link;
 free(ptr);
}
```

// Function to delete node from the end of the singly linked list

```
void delete_end()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr,*temp;
 int num;
```

```
  if(start->link==NULL)
 {
  ptr=start;
  start=NULL;
 }
 else
 {
  temp=start;
  ptr=start->link;
  while(ptr->link!=NULL)
  {
   temp=ptr;
   ptr=ptr->link;
  }
  temp->link=NULL;
 }
 num=ptr->info;
 printf("\nThe deleted element from the singly linked list is : %d",num);
 free(ptr);
}
```

// Function to delete node from a specific location in singly linked list

```
void delete_spe()
{
if(start==NULL)
{
 printf("\nList is empty\n");
 return;
}
struct list *ptr,*temp;
int num,loc;
ptr=start;
printf("\nEnter the location from where you want to delete the value : ");
scanf("%d",&loc);
 if(loc==1)
  start=ptr->link;
 else
 {
  for(int i=1;i<loc;i++)
  {
  temp=ptr;
  ptr=ptr->link;
   if(ptr==NULL)
   {
   printf("\nTotal nodes in the list are lesser than this position. ");
   printf("\nSo node cannot be deleted\n");
   return;
   }
  }
 temp->link=ptr->link;
 }
 num=ptr->info;
 printf("\nThe deleted element from the singly linked list is : %d",num);
 free(ptr);
}
```

// Function to traverse singly linked list

```
void traverse()
{
 printf("\nStart : %d",start);
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 ptr=start;
 printf("\nTraverse the list : \n");
 printf("\nPTR->INFO\tPTR->LINK\tPTR\n");
 while(ptr!=NULL)
 {
  printf("\n%d\t\t%d\t\t%d",ptr->info,ptr->link,ptr);
  ptr=ptr->link;
 }
}
```

OPERATIONS OF CIRCULAR LINKED LISTS

(A) INSERTION

(I) Algorithm to insert node at the beginning of the circular linked list

Let PTR is the structure pointer which allocates memory for the new node at the beginning of the circular linked list & NUM is the element to be inserted into the linked list, INFO represents the information part of the new node and LINK represents the link or next pointer of the new node pointing to the address of next node. START represents the address of first node & LAST represents the address of the last node. Initially , before inserting first node in the singly linked list, START=NULL, LAST=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into circular linked list.
Step 3 : Set PTR->INFO = NUM
Step 4 : If START = NULL : then
             Set START := LAST := PTR
             Set PTR->LINK := PTR.
         Else
             Set PTR->LINK :=START.
             Set START := PTR
             Set LAST->LINK := PTR
         [End of If Else Structure]
Step 5 : Exit
```

Function to insert node at the beginning of circular linked list

```
void insertbeg()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter the element to be inserted at the beginning of circular linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   start=last=ptr;
   ptr->link=ptr;
 }
 else
 {
   ptr->link=start;
   start=ptr;
   last->link=ptr;
 }
}
```

(II) Algorithm to insert node at the end of the circular linked list

Let PTR is the structure pointer which allocates memory for the new node at the beginning of the circular linked list & NUM is the element to be inserted into the linked list, INFO represents the information part of the new node and LINK represents the link or next pointer of the new node pointing to the address of next node. START represents the address of first node & LAST represents the address of the last node. Initially , before inserting first node in the singly linked list, START=NULL, LAST=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into circular linked list.
Step 3 : Set PTR->INFO = NUM
Step 4 : If START = NULL : then
             Set START := LAST := PTR
             Set PTR->LINK := PTR.
         Else
             Set LAST->LINK :=PTR.
             Set LAST := PTR
             Set PTR->LINK := START
         [End of If Else Structure]
Step 6 : Exit
```

Function to insert node at the end of the circular linked list

```
void insertend()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter the element to be inserted at the end of circular linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   start=last=ptr;
   ptr->link=ptr;
 }
 else
 {
   last->link=ptr;
   last=ptr;
   ptr->link=start;
 }
}
```

(B) DELETION

(III) Algorithm to delete node from the beginning of the circular linked list

Let PTR is the structure pointer which deallocates memory of the node at the beginning of the circular linked list & NUM is the element to be deleted from the linked list, INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node. START represents the address of first node, LAST represents the address of the last node.

```
Step 1 : If START := NULL : then
             Write : “List is empty” and return.
Step 2 : Set PTR := START.
Step 3 : Set NUM := PTR->INFO.
Step 4 : Write : “Deleted element from the beginning of the circular linked list : “,NUM.
Step 5 : If START = LAST : then
             Set START := LAST := NULL.
         Else
             Set START := START->LINK.
             Set LAST->LINK := START.
         [End of If else Structure]
Step 6 : Deallocate memory of the node at the beginning of circular linked list.
Step7 : Exit
```

Function to delete node from the beginning of circular linked list

```
void deletebeg()
{
 if(start==NULL)
 {
   printf("\nList is empty");
   return;
 }
 struct list *ptr;
 ptr=start;
 int num=ptr->info;
 printf("\nDeleted element from beginning of linked list is : %d",num);
 if(start==last)
   start=last=NULL;
 else
 {
  start=start->link;
  last->link=start;
 }
 free(ptr);
}
```

(IV) Algorithm to delete node from the end of the circular linked list

Let PTR is the structure pointer which deallocates memory of the node at the end of the circular linked list, TEMP is a structure pointer to modify the LINK part of previous node & NUM is the element to be deleted from the linked list, INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node. START represents the address of first node, LAST represents the address of the last node.

```
 Step 1 : If START := NULL : then
              Write : “List is empty” and return.
 Step 2 : Set PTR := START.
 Step 3 : If START = LAST : then
              Set START := LAST := NULL.
          Else
               Repeat loop while PTR->LINK != START
                  Set TEMP := PTR
                  Set PTR := PTR->LINK.
               [End of loop]
             Set TEMP->LINK := PTR->LINK.
             Set LAST := TEMP.
          [End of Step 3 If Else structure]
 Step 4 : Set NUM := PTR->INFO.
 Step 5 : Write : “Deleted element from the end of the circular linked list : “,NUM.
 Step 6 : Deallocate memory of the node at the beginning of circular linked list.
 Step7 : Exit
```

Function to delete node from the end of circular linked list

```
void deleteend()
{
 if(start==NULL)
 {
   printf("\nList is empty");
   return;
 }

 struct list *ptr,*temp;
 int num;
 ptr=start;

 if(start==last)
   start=last=NULL;
 else
 {
   while(ptr->link!=start)
   {
     temp=ptr;
     ptr=ptr->link;
   }
  temp->link=ptr->link;
  last=temp;
 }
 num=ptr->info;
 printf("\nDeleted element from the end of circular linked list : %d",num);
 free(ptr);
}
```

PROGRAM – 2 : CIRCULAR LINKED LIST OPERATIONS

```c
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct list
{
 int info;
 struct list *link;
};

struct list *start,*last;

void initialize();
void insertbeg();
void insertend();
void deletebeg();
void deleteend();
void traverse();

void main()
{
 int choice;
 initialize();
 while(1)
 {
   clrscr();
   printf("\nIMPLEMENTATION OF A CIRCULAR LINKED LIST");
   printf("\n---------------------------------------");
   printf("\n1. Insertion at beginning");
   printf("\n2. Insertion at end");
   printf("\n3. Deletion from beginning");
   printf("\n4. Deletion from end");
   printf("\n5. Traverse");
   printf("\n6. Exit");
   printf("\n---------------------------------------");
   printf("\n\nEnter your choice [1/2/3/4/5] : ");
   scanf("%d",&choice);
   switch(choice)
   {
     case 1 : insertbeg();
              break;
     case 2 : insertend();
              break;
     case 3 : deletebeg();
              break;
     case 4 : deleteend();
              break;
     case 5 : traverse();
              break;
     case 6 : exit(0);
     default : printf("\nInvalid choice");
   }
 getch();
}
}
```

// Function to initialize circular linked list

```
void initialize()
{
  start=last=NULL;
}
```

// Function to insert node at the beginning of circular linked list

```
void insertbeg()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));

 printf("\nEnter the element to be inserted at the beginning of circular linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
   start=last=ptr->link=ptr;
 else
 {
   ptr->link=start;
   start=ptr;
   last->link=ptr;
 }
}
```

// Function to insert node at the end of the circular linked list

```
void insertend()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter the element to be inserted at the end of circular linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   start=last=ptr;
   ptr->link=ptr;
 }
 else
 {
   last->link=ptr;
   last=ptr;
   ptr->link=start;
 }
}
```

// Function to delete node from the beginning of circular linked list

```
void deletebeg()
{
 if(start==NULL)
 {
   printf("\nList is empty");
   return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 num=ptr->info;
 printf("\nDeleted element from beginning of linked list is : %d",num);
 if(start==last)
   start=last=NULL;
 else
 {
  start=start->link;
  last->link=start;
 }
 free(ptr);
}
```

// Function to delete node from the end of circular linked list

```
void deleteend()
{
 if(start==NULL)
 {
   printf("\nList is empty");
   return;
 }
 struct list *ptr,*temp;
 int num;
 ptr=start;
 if(start==last)
   start=last=NULL;
 else
 {
  while(ptr->link!=start)
  {
   temp=ptr;
   ptr=ptr->link;
  }
 temp->link=ptr->link;
 last=temp;
 }
 num=ptr->info;
 printf("\nDeleted element from the end of circular linked list : %d", num);
 free(ptr);
}
```

// Function to traverse the circular linked list

```
void traverse()
{
 printf("\nStart : %d",start);
 printf("\nLast : %d",last);
if(start==NULL)
 {
   printf("\nList is empty");
   return;
 }
 struct list *ptr;
 ptr=start;
 printf("\n\nCircular linked list elements are : \n");
 printf("\nPTR->INFO\tPTR->LINK\tPTR\n");
 while(ptr->link!=start)
 {
   printf("\n%d\t\t%d\t\t%d",ptr->info,ptr->link,ptr);
   ptr=ptr->link;
 }
  printf("\n%d\t\t%d\t\t%d",ptr->info,ptr->link,ptr);
}
```

OPERATIONS OF DOUBLY LINKED LISTS

(A) INSERTION

(I) Algorithm to insert node at the beginning of the doubly linked list

Let PTR is the structure pointer which allocates memory for the new node at the beginning of the doubly linked list & NUM is the element to be inserted into the linked list, INFO represents the information part of the new node, LEFT represents the structure pointer of the new node pointing to the address of previous node and RIGHT represents the structure pointer of the new node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node. Initially , before inserting first node in the singly linked list, START=NULL, LAST=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into doubly linked list.
Step 3 : Set PTR->INFO = NUM
Step 4 : If START = NULL : then
             Set PTR->LEFT := PTR->RIGHT := NULL.
             Set START := LAST := PTR
         Else
             Set PTR->LEFT :=NULL.
             Set PTR->RIGHT := START.
             Set START->LEFT := PTR
             Set START := PTR.
         [End of If Else Structure]
Step 5 : Exit
```

Function to insert node at the beginning of doubly linked list

```
void insertbeg()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   ptr->left=ptr->right=NULL;
   start=last=ptr;
 }
 else
 {
   ptr->left=NULL;
   ptr->right=start;
   start->left=ptr;
   start=ptr;
 }
}
```

(II) Algorithm to insert node at the end of the doubly linked list

Let PTR is the structure pointer which allocates memory for the new node at the end of the doubly linked list & NUM is the element to be inserted into the linked list, INFO represents the information part of the new node, LEFT represents the structure pointer of the new node pointing to the address of previous node and RIGHT represents the structure pointer of the new node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node. Initially , before inserting first node in the singly linked list, START=NULL, LAST=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into doubly linked list.
Step 3 : Set PTR->INFO := NUM
Step 4 : Set PTR->RIGHT := NULL
Step 5 : If START = NULL : then
              Set PTR->RIGHT := NULL.
              Set START := LAST := PTR
           Else
              Set PTR->LEFT := LAST
              Set LAST->RIGHT := PTR
              Set LAST := PTR
            [End of If Else Structure]
Step 6 : Exit
```

Function to insert node at the end of doubly linked list

```
void insertend()
{
 struct list *ptr,*temp;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->right=NULL;
 if(start==NULL)
 {
   ptr->left=ptr->right=NULL;
   start=last=ptr;
 }
 else
 {
   ptr->left=last;
   last->right=ptr;
   last=ptr;
 }
}
```

(B) <u>DELETION</u>

(III) <u>Algorithm to delete node at the beginning of the doubly linked list</u>

Let PTR is the structure pointer which deallocates the memory of the node at the beginning of the doubly linked list & NUM is the element to be deleted from the linked list, INFO represents the information part of the node, LEFT represents the structure pointer of the deleted node pointing to the address of previous node and RIGHT represents the structure pointer of the deleted node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node.

```
Step 1 : If START := NULL : then
             Write : "List is empty" and return.
         [End of If Structure]
Step 2 : Set PTR := START.
Step 3 : If START = LAST : then
             Set START := LAST := NULL.
         Else
             Set START := START->RIGHT.
             Set START->LEFT := NULL.
         [End of If else Structure]
Step 4 : Set NUM := PTR->INFO.
Step 5 : Write : "Deleted element from the beginning of the Doubly linked list : ",NUM.
Step 6 : Deallocate memory of the node at the beginning of Doubly linked list.
Step7 : Exit
```

Function to delete node from the beginning of the doubly linked list

```
void deletebeg()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 if(start==last)
   start=last=NULL;
 else
 {
  start=start->right;
  start->left=NULL;
 }
 num=ptr->info;
 printf("\nDeleted element from the beginning of doubly linked list is : %d",num);
 free(ptr);
}
```

(IV) Algorithm to delete node at the end of the doubly linked list

Let PTR is the structure pointer which deallocates the memory of the node at the end of the doubly linked list & NUM is the element to be deleted from the linked list, INFO represents the information part of the node, LEFT represents the structure pointer of the deleted node pointing to the address of previous node and RIGHT represents the structure pointer of the deleted node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node.

```
Step 1 : If START := NULL : then
             Write : “List is empty” and return.
         [End of If Structure]
Step 2 : Set PTR := START.
Step 3 : If START = LAST : then
             Set START := LAST := NULL.
         Else
             Set LAST := LAST->LEFT.
             Set LAST->RIGHT := NULL.
         [End of If else Structure]
Step 4 : Set NUM := PTR->INFO.
Step 5 : Write : “Deleted element from the end of the Doubly linked list : “,NUM.
Step 6 : Deallocate memory of the node at the end of Doubly linked list.
Step7 : Exit
```

Function to delete node from the end of the doubly linked list

```
void deleteend()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=last;
 if(start==last)
   start=last=NULL;
 else
 {
  last=last->left;
  last->right=NULL;
 }
 num=ptr->info;
 printf("\nDeleted element from the end of the doubly linked list is : %d",num);
 free(ptr);
}
```

PROGRAM – 3 : DOUBLY LINKED LIST OPERATIONS

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

struct list
{
int info;
struct list *left,*right;
};

struct list *start=NULL,*last=NULL;

void insertbeg();
void insertend();
void deletebeg();
void deleteend();
void traverse();

void main()
{
int choice;
while(1)
{
clrscr();
 printf("  IMPLEMENTATION OF DOUBLY LINKED LIST \n");
 printf("------------------------------------\n");
 printf("1. Insertion at Beginning\n");
 printf("2. Insertion at End\n");
 printf("3. Deletion from beginning\n");
 printf("4. Deletion from end\n");
 printf("5. Traverse the list \n");
 printf("6. Exit \n");
 printf("------------------------------------\n");
 printf("\nEnter your choice [1/2/3/4/5/6] : ");
 scanf("%d",&choice);

  switch(choice)
  {
   case 1 : insertbeg();
              break;
   case 2 : insertend();
              break;
   case 3 : deletebeg();
              break;
   case 4 : deleteend();
              break;
   case 5 : traverse();
              break;
   case 6 : exit(0);
   default : printf("\nInvalid choice");
  }
 getch();
}
}
```

// Function to insert node at the beginning of doubly linked list

```
void insertbeg()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   ptr->left=ptr->right=NULL;
   start=last=ptr;
 }
 else
 {
   ptr->left=NULL;
   ptr->right=start;
   start->left=ptr;
   start=ptr;
 }
}
```

// Function to insert node at the end of doubly linked list

```
void insertend()
{
 struct list *ptr,*temp;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->right=NULL;
 if(start==NULL)
 {
   ptr->left=ptr->right=NULL;
   start=last=ptr;
 }
 else
 {
   ptr->left=last;
   last->right=ptr;
   last=ptr;
 }
}
```

// Function to delete node from the beginning of the doubly linked list

```
void deletebeg()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 if(start==last)
   start=last=NULL;
 else
 {
  start=start->right;
  start->left=NULL;
 }
 num=ptr->info;
 printf("\nDeleted element from the beginning of doubly linked list is : %d",num);
 free(ptr);
}
```

// Function to delete node from the end of the doubly linked list

```
void deleteend()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=last;
 if(start==last)
   start=last=NULL;
 else
 {
  last=last->left;
  last->right=NULL;
 }
 num=ptr->info;
 printf("\nDeleted element from the end of the doubly linked list is : %d",num);
 free(ptr);
}
```

// Function to traverse the Doubly Linked list

```
void traverse()
{
 printf("\nStart : %d",start);
 printf("\nLast : %d",last);
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 ptr=start;
 printf("\nTraverse the list : \n");
 printf("\nPTR->LEFT\tPTR->INFO\tPTR->RIGHT\tPTR\n");
 while(ptr!=NULL)
 {
  printf("\n%d\t\t%d\t\t%d\t\t%d",ptr->left,ptr->info,ptr->right,ptr);
  ptr=ptr->right;
 }
}
```

OPERATIONS OF CIRCULAR DOUBLY LINKED LISTS

(A) INSERTION

(I) Algorithm to insert node at the beginning of the circular doubly linked list

Let PTR is the structure pointer which allocates memory for the new node at the beginning of the circular doubly linked list & NUM is the element to be inserted into the circular doubly linked list, INFO represents the information part of the new node, LEFT represents the structure pointer of the new node pointing to the address of previous node and RIGHT represents the structure pointer of the new node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node. Initially , before inserting first node in the circular doubly linked list, START=NULL, LAST=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into circular doubly linked list.
Step 3 : Set PTR->INFO = NUM
Step 4 : If START = NULL : then
             Set PTR->LEFT := PTR->RIGHT := PTR.
             Set START := LAST := PTR
         Else
             Set PTR->LEFT :=LAST.
             Set PTR->RIGHT := START.
             Set START->LEFT := PTR
             Set LAST->RIGHT :=PTR
             Set START := PTR.
         [End of If Else Structure]
Step 5 : Exit
```

Function to insert node at the beginning of circular doubly linked list

```
void insertbeg()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in circular doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   ptr->left=ptr->right=ptr;
   start=last=ptr;
 }
 else
 {
  ptr->left=last;
  ptr->right=start;
  start->left=ptr;
  last->right=ptr;
  start=ptr;
 }
}
```

(II) <u>Algorithm to insert node at the end of the circular doubly linked list</u>

Let PTR is the structure pointer which allocates memory for the new node at the end of the circular doubly linked list & NUM is the element to be inserted into the circular doubly linked list, INFO represents the information part of the new node, LEFT represents the structure pointer of the new node pointing to the address of previous node and RIGHT represents the structure pointer of the new node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node. Initially , before inserting first node in the circular doubly linked list, START=NULL, LAST=NULL.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into circular doubly linked list.
Step 3 : Set PTR->INFO := NUM
Step 4 : If START = NULL : then
             Set PTR->LEFT := PTR->RIGHT := PTR.
             Set START := LAST := PTR
         Else
             Set PTR->RIGHT := START.
             Set PTR->LEFT := LAST.
             Set LAST->RIGHT := PTR
             Set START->LEFT :=PTR
             Set LAST := PTR.
         [End of If Else Structure]
Step 5 : Exit
```

Function to insert node at the end of circular doubly linked list

```
void insertend()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in circular doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   ptr->left=ptr->right=ptr;
   start=last=ptr;
 }
 else
 {
  ptr->right=start;
  ptr->left=last;
  last->right=ptr;
  start->left=ptr;
  last=ptr;
 }
}
```

(B) DELETION

(III) Algorithm to delete node from the beginning of the circular doubly linked list

Let PTR is the structure pointer which deallocates the memory of the node at the beginning of the circular doubly linked list & NUM is the element to be deleted from the linked list, INFO represents the information part of the node, LEFT represents the structure pointer of the deleted node pointing to the address of previous node and RIGHT represents the structure pointer of the deleted node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node.

```
Step 1 : If START := NULL : then
              Write : “List is empty” and return.
         [End of If Structure]
Step 2 : Set PTR := START.
Step 3 : If START = LAST : then
              Set START := LAST := NULL.
           Else
              Set START := START->RIGHT.
              Set START->LEFT := LAST.
              Set LAST->RIGHT := START.
          [End of If Else Structure]
Step 4 : Set NUM := PTR->INFO.
Step 5 : Write : “Deleted element from the Circular doubly linked list : “,NUM.
Step 6 : Deallocate memory of the node at the beginning of Circular doubly linked list.
Step7 : Exit
```

Function to delete node from the beginning of the circular doubly linked list

```
void deletebeg()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 if(start==last)
   start=last=NULL;
 else
 {
  start=start->right;
  start->left=last;
  last->right=start;
 }
 num=ptr->info;
 printf("\nDeleted element from the beginning of circular doubly linked list is : %d",num);
 free(ptr);
}
```

(IV) <u>Algorithm to delete node from the end of the circular doubly linked list</u>

Let PTR is the structure pointer which deallocates the memory of the node at the end of the circular doubly linked list & NUM is the element to be deleted from the linked list, INFO represents the information part of the node, LEFT represents the structure pointer of the deleted node pointing to the address of previous node and RIGHT represents the structure pointer of the deleted node pointing to the address of next node in the list. START represents the address of first node & LAST represents the address of the last node.

```
Step 1 : If START := NULL : then
             Write : “List is empty” and return.
         [End of If Structure]
Step 2 : Set PTR := START.
Step 3 : If START = LAST : then
             Set START := LAST := NULL.
         Else
             Set LAST := LAST->LEFT.
             Set LAST->RIGHT := START.
             Set START->LEFT := LAST
         [End of If else Structure]
Step 4 : Set NUM := PTR->INFO.
Step 5 : Write : “Deleted element from the end of the Circular doubly linked list : “,NUM.
Step 6 : Deallocate memory of the node at the end of Circular doubly linked list.
Step7 : Exit
```

// Function to delete node from the end of the circular doubly linked list

```
void deleteend()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=last;
 if(start==last)
   start=last=NULL;
 else
 {
   last=last->left;
   last->right=start;
   start->left=last;
 }
 num=ptr->info;
 printf("\nDeleted element from the end of the circular doubly linked list is : %d",num);
 free(ptr);  }
```

PROGRAM - 4 : CIRCULAR DOUBLY LINKED LIST OPERATIONS

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct list
{
 int info;
 struct list *left,*right;
};

struct list *start=NULL,*last=NULL;

void insertbeg();
void insertend();
void deletebeg();
void deleteend();
void traverse();

void main()
{
 int choice;
 while(1)
 {
  clrscr();
  printf("  IMPLEMENTATION OF CIRCULAR DOUBLY LINKED LIST \n");
  printf("-----------------------------------\n");
  printf("1. Insertion at Beginning\n");
  printf("2. Insertion at End\n");
  printf("3. Deletion from beginning\n");
  printf("4. Deletion from end\n");
  printf("5. Traverse the list \n");
  printf("6. Exit \n");
  printf("-----------------------------------\n");
  printf("\nEnter your choice [1/2/3/4/5/6] : ");
  scanf("%d",&choice);
  switch(choice)
  {
    case 1 : insertbeg();
             break;
    case 2 : insertend();
             break;
    case 3 : deletebeg();
             break;
    case 4 : deleteend();
             break;
    case 5 : traverse();
             break;
    case 6 : exit(0);
    default : printf("\nInvalid choice");
  }
  getch();
 }
}
```

// Function to insert node at the beginning of circular doubly linked list

```
void insertbeg()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in circular doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   ptr->left=ptr->right=ptr;
   start=last=ptr;
 }
 else
 {
  ptr->left=last;
  ptr->right=start;
  start->left=ptr;
  last->right=ptr;
  start=ptr;
 }
}
```

// Function to insert node at the end of circular doubly linked list

```
void insertend()
{
 struct list *ptr;
 int num;
 ptr=(struct list *)malloc(sizeof(struct list));
 printf("\nEnter element to be inserted in circular doubly linked list : ");
 scanf("%d",&num);
 ptr->info=num;
 if(start==NULL)
 {
   ptr->left=ptr->right=ptr;
   start=last=ptr;
 }
 else
 {
  ptr->right=start;
  ptr->left=last;
  last->right=ptr;
  start->left=ptr;
  last=ptr;
 }
}
```

// Function to delete node from the beginning of the circular doubly linked list

```
void deletebeg()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=start;
 if(start==last)
   start=last=NULL;
 else
 {
  start=start->right;
  start->left=last;
  last->right=start;
 }
 num=ptr->info;
 printf("\nDeleted element from the beginning of circular doubly linked list is : %d",num);
 free(ptr);
}
```

// Function to delete node from the end of the circular doubly linked list

```
void deleteend()
{
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 int num;
 ptr=last;
 if(start==last)
   start=last=NULL;
 else
 {
   last=last->left;
   last->right=start;
   start->left=last;
 }
 num=ptr->info;
 printf("\nDeleted element from the end of the circular doubly linked list is : %d",num);
 free(ptr);
}
```

//Function to display the elements of Circular Doubly Linked list

```
void traverse()
{
 printf("\nStart : %d",start);
 printf("\nLast : %d",last);
 if(start==NULL)
 {
  printf("\nList is empty");
  return;
 }
 struct list *ptr;
 ptr=start;
 printf("\nTraverse the list : \n");
 printf("\nPTR->LEFT\tPTR->INFO\tPTR->RIGHT\tPTR\n");
 while(ptr->right!=start)
 {
   printf("\n%d\t\t%d\t\t%d\t\t%d",ptr->left,ptr->info,ptr->right,ptr);
   ptr=ptr->right;
 }
  printf("\n%d\t\t%d\t\t%d\t\t%d",ptr->left,ptr->info,ptr->right,ptr);
}
```