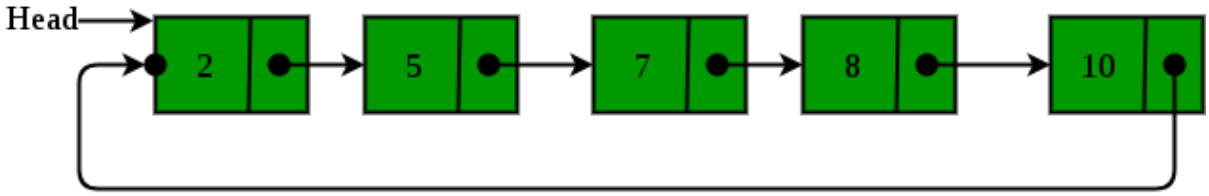
```
  if(isEmpty()) {
     //make it the last link
     last = link;
  } else {
     //make link a new last link
     last->next = link;


     //mark old last node as prev of new link
     link->prev = last;
  }

  //point last to new last node
  last = link;
}
```

## Circular Linked Lists: all operations their algorithms and the complexity analysis:

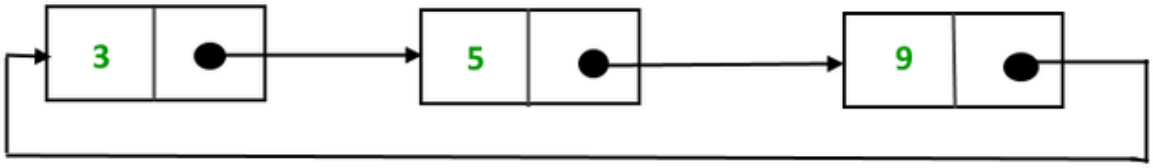
### What is Circular linked list?

*The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.*




**There are generally two types of circular linked lists:**

- **Circular singly linked list:** In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.




*Representation of Circular singly linked list*

- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.


*Representation of circular doubly linked list*

**Note:** We will be using the singly circular linked list to represent the working of the circular linked list.

**Representation of circular linked list:**

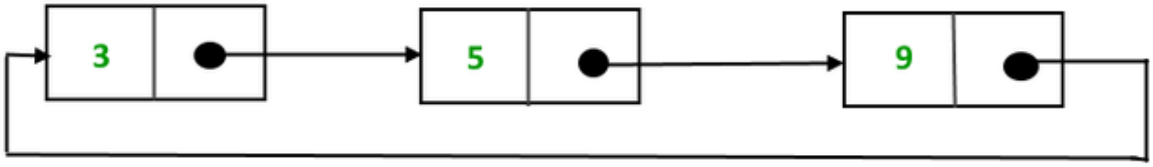
Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

Node representation of a Circular Linked List:

- C++

```
// Class Node, similar to the linked list
class Node{
    int value;

  // Points to the next node.
    Node next;
}
```

Example of Circular singly linked list:




*Example of  circular linked list*

The above  Circular singly linked list can be represented as:

- C++

```
// Initialize the Nodes.
Node one = new Node(3);
Node two = new Node(5);
Node three = new Node(9);

// Connect nodes
one.next = two;
two.next = three;
three.next = one;
```

**Explanation:** In the above program one, two, and three are the node with values 3, 5, and 9 respectively which are connected in a circular manner as:

- **For Node One:** The Next pointer stores the address of Node two.

- **For Node Two:** The Next stores the address of Node three
- **For Node Three:** The Next points to node one.

**Operations on the circular linked list:**
We can do some operations on the circular linked list similar to the singly linked list which are:

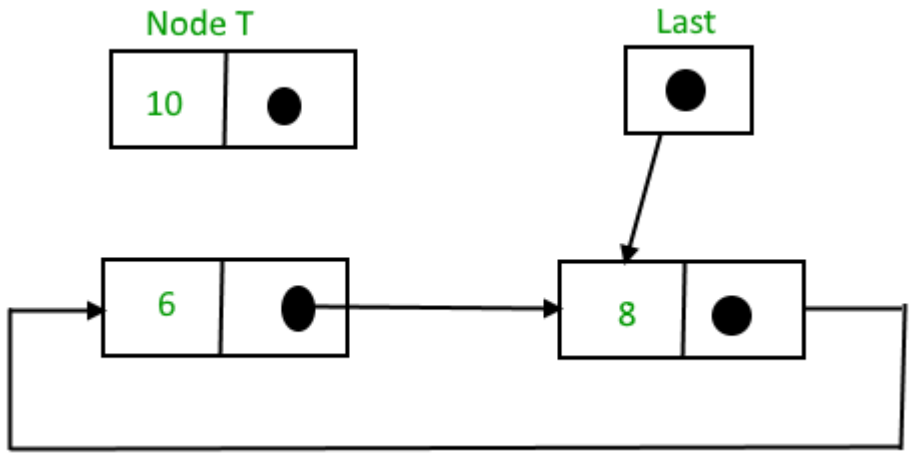
1. Insertion
2. Deletion

1. **Insertion in the circular linked list:**
A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

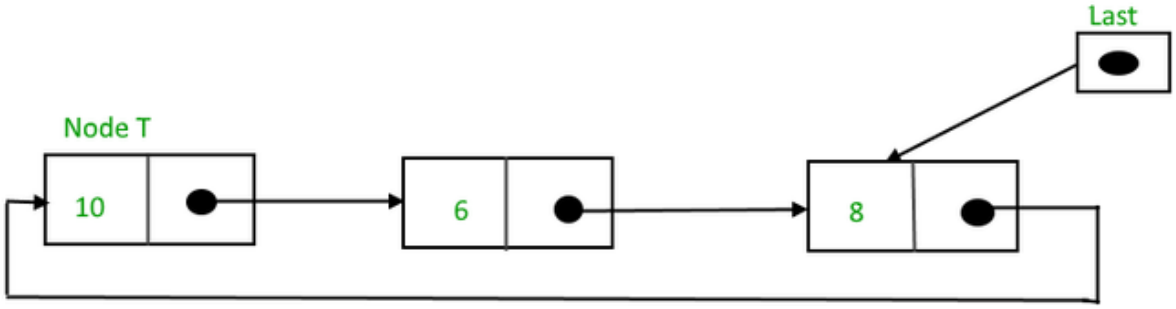
1) **Insertion at the beginning of the list:** To insert a node at the beginning of the list, follow these steps:

- Create a node, say T.
- Make T -> next = last -> next.
- last -> next = T.




*Circular linked list before insertion*

And then,




*Circular linked list after insertion*

Below is the code implementation to insert a node at the beginning of the list:

- C++

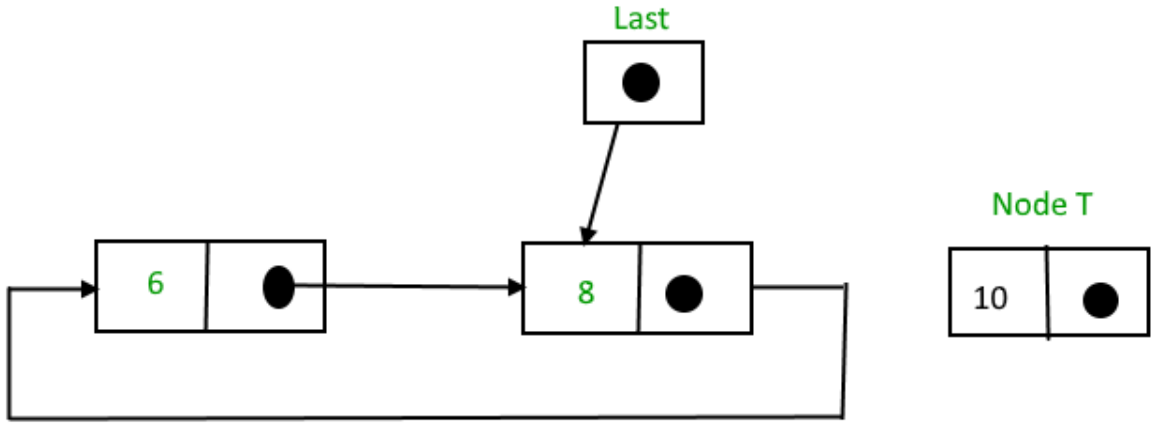
```
struct Node *addBegin(struct Node *last, int data)
{
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;

return last;
}
```

**Time complexity:** O(1) to insert a node at the beginning no need to traverse list it takes constant time
**Auxiliary Space:** O(1)

2) **Insertion at the end of the list:** To insert a node at the end of the list, follow these steps:

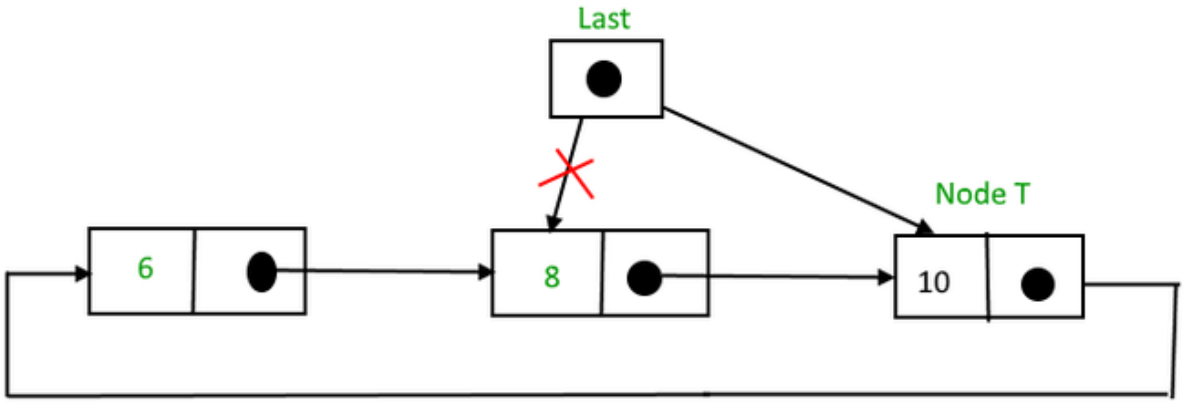
- Create a node, say T.
- Make T -> next = last -> next;
- last -> next = T.
- last = T.

Before insertion,




*Circular linked list before insertion of node at the end*

After insertion,




*Circular linked list after insertion of node at the end*

Below is the code implementation to insert a node at the beginning of the list:

- C++

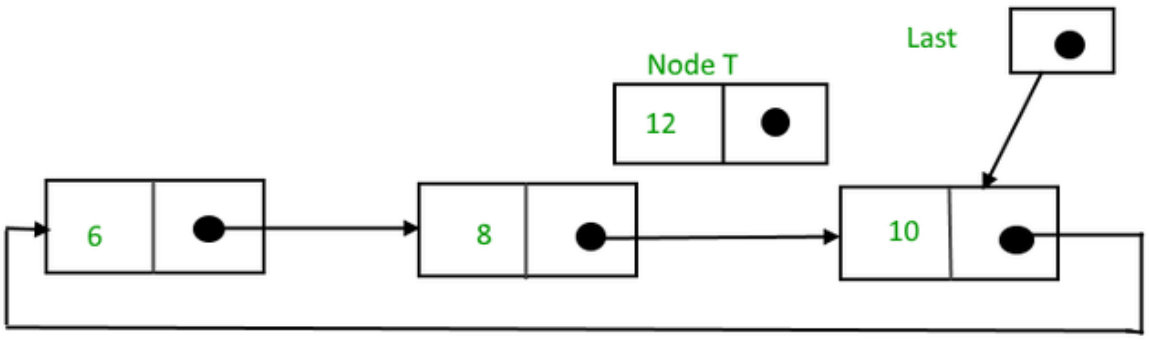
```
struct Node *addEnd(struct Node *last, int data)
{
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;
last = temp;

return last;
```

```
}
```

**Time Complexity:** O(1) to insert a node at the end of the list. No need to traverse the list as we are utilizing the last pointer, hence it takes constant time.
**Auxiliary Space:** O(1)
3) **Insertion in between the nodes:** To insert a node in between the two nodes, follow these steps:

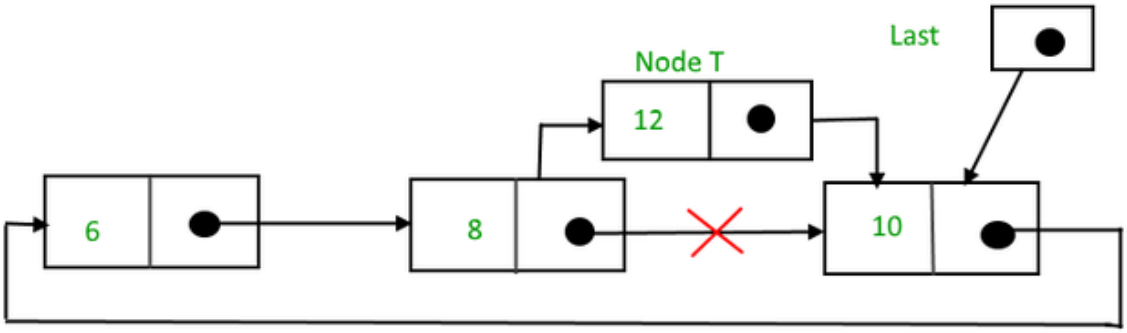
- Create a node, say T.
- Search for the node after which T needs to be inserted, say that node is P.
- Make T -> next = P -> next;
- P -> next = T.

Suppose 12 needs to be inserted after the node has the value 10,




*Circular linked list before insertion*

After searching and insertion,




*Circular linked list after  insertion*

Below is the code to insert a node at the specified position of the List:

- C++

```
struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
    return NULL;

    struct Node *temp, *p;
    p = last -> next;

    // Searching the item.
    do
    {
        if (p ->data == item)
        {
            // Creating a node dynamically.
            temp = (struct Node *)malloc(sizeof(struct Node));

            // Assigning the data.
            temp -> data = data;

            // Adjusting the links.
```

```
            temp -> next = p -> next;

            // Adding newly allocated node after p.
            p -> next = temp;

            // Checking for the last node.
            if (p == last)
                last = temp;

            return last;
        }
        p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

**Time Complexity:** O(N)
**Auxiliary Space:** O(1)

2. Deletion in a circular linked list:

**1) Delete the node only if it is the only node in the circular linked list:**

- Free the node's memory
- The last value should be NULL A node always points to another node, so NULL assignment is not necessary. Any node can be set as the starting point. Nodes are traversed quickly from the first to the last.

**2) Deletion of the last node:**

- Locate the node before the last node (let it be temp)
- Keep the address of the node next to the last node in temp
- Delete the last memory
- Put temp at the end

**3) Delete any node from the circular linked list:** We will be given a node and our task is to delete that node from the circular linked list.

**Algorithm:**

**Case 1**: List is empty.

- If the list is empty we will simply return.

**Case 2**:List is not empty

- If the list is not empty then we define two pointers **curr** and **prev** and initialize the pointer **curr** with the **head** node.
- Traverse the list using **curr** to find the node to be deleted and before moving to curr to the next node, every time set prev = curr.
- If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr).

- If the list has more than one node, check if it is the first node of the list. Condition to check this( curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr.
- If curr is not the first node, we check if it is the last node in the list. Condition to check this is (curr -> next == head).
- If curr is the last node. Set prev -> next = head and delete the node curr by free(curr).
- If the node to be deleted is neither the first node nor the last node, then set prev -> next = curr -> next and delete curr.

Below is the implementation for the above approach:

- C++

```
// C++ program to delete a given key from
// linked list.
#include <bits/stdc++.h>
using namespace std;

// Structure for a node
class Node {
public:
    int data;
    Node* next;
};

// Function to insert a node at the
// beginning of a Circular linked list
void push(Node** head_ref, int data)
{

    // Create a new node and make head
    // as next of it.
    Node* ptr1 = new Node();
```

```
ptr1->data = data;
ptr1->next = *head_ref;

// If linked list is not NULL then
// set the next of last node
if (*head_ref != NULL) {

    // Find the node before head and
    // update next of it.
    Node* temp = *head_ref;
    while (temp->next != *head_ref)
        temp = temp->next;
    temp->next = ptr1;
}
else

    // For the first node
    ptr1->next = ptr1;

*head_ref = ptr1;
```

```
}

// Function to print nodes in a given
// circular linked list
void printList(Node* head)
{
    Node* temp = head;
    if (head != NULL) {
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
    }

    cout << endl;
}

// Function to delete a given node
// from the list
void deleteNode(Node** head, int key)
```

```
{

    // If linked list is empty
    if (*head == NULL)
        return;

    // If the list contains only a
    // single node
    if ((*head)->data == key && (*head)->next == *head) {
        free(*head);
        *head = NULL;
        return;
    }

    Node *last = *head, *d;

    // If head is to be deleted
    if ((*head)->data == key) {

        // Find the last node of the list
```

```
    while (last->next != *head)
        last = last->next;

    // Point last node to the next of
    // head i.e. the second node
    // of the list
    last->next = (*head)->next;
    free(*head);
    *head = last->next;
    return;
}

// Either the node to be deleted is
// not found or the end of list
// is not reached
while (last->next != *head && last->next->data != key) {
    last = last->next;
}

// If node to be deleted was found
```

```
    if (last->next->data == key) {
        d = last->next;
        last->next = d->next;
        free(d);
    }
    else
        cout << "no such keyfound";
}

// Driver code
int main()
{
    // Initialize lists as empty
    Node* head = NULL;

    // Created linked list will be
    // 2->5->7->8->10
    push(&head, 2);
    push(&head, 5);
    push(&head, 7);
```

```
    push(&head, 8);
    push(&head, 10);

    cout << "List Before Deletion: ";
    printList(head);

    deleteNode(&head, 7);

    cout << "List After Deletion: ";
    printList(head);

    return 0;
}
```

**Output**

List Before Deletion: 10 8 7 5 2

List After Deletion: 10 8 5 2

**Time Complexity:** O(N), Worst case occurs when the element to be deleted is the last element and we need to move through the whole list.

**Auxiliary Space:** O(1), As constant extra space is used.

**Advantages of Circular Linked Lists:**

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of a queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and the front can always be obtained as next of last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like the Fibonacci Heap.

**Disadvantages of circular linked list:**

- Compared to singly linked lists, circular lists are more complex.
- Reversing a circular list is more complicated than singly or doubly reversing a circular list.
- It is possible for the code to go into an infinite loop if it is not handled carefully.
- It is harder to find the end of the list and control the loop.

**Applications of circular linked lists:**

- Multiplayer games use this to give each player a chance to play.
- A circular linked list can be used to organize multiple running applications on an operating system. These applications are iterated over by the OS.

**Why circular linked list?**

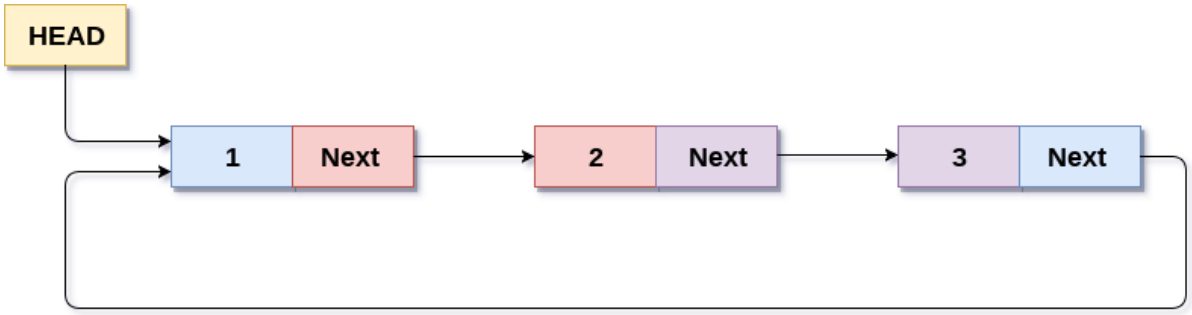
- A node always points to another node, so NULL assignment is not necessary.
- Any node can be set as the starting point.
- Nodes are traversed quickly from the first to the last.

**Circular Singly Linked List**

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



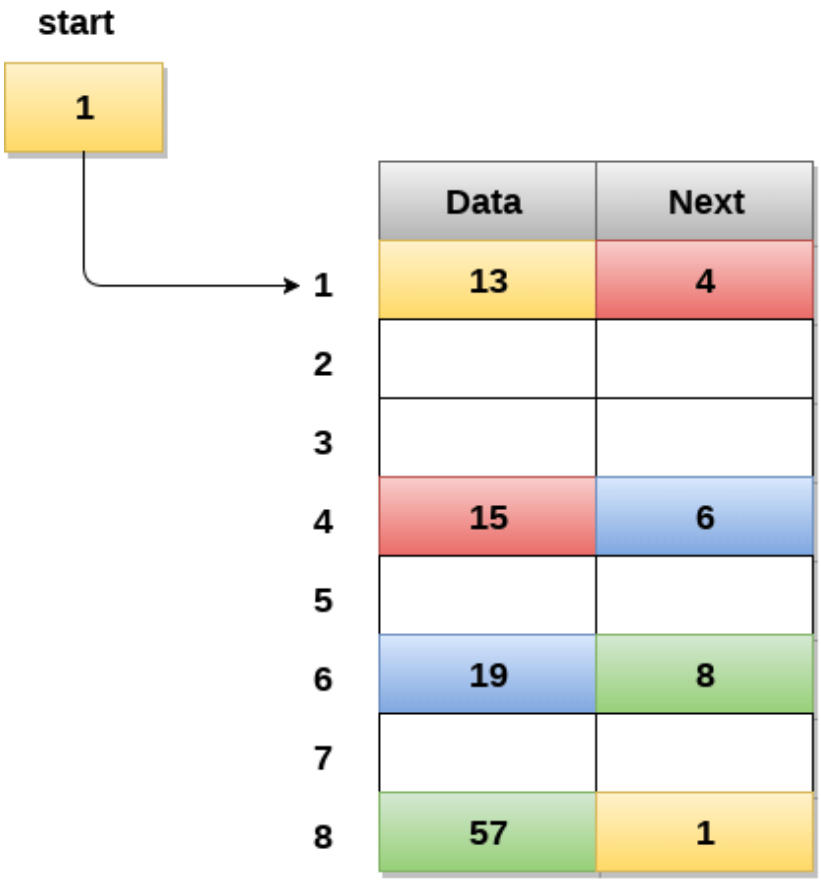

**Circular Singly Linked List**

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

**Memory Representation of circular linked list:**

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.




**Memory Representation of a circular linked list**

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

**Insertion**

| SN | Operation | Description |
|---|---|---|

| | | |
|---|---|---|
| 1 | Insertion at beginning | Adding a node into circular singly linked list at the beginning. |
| 2 | Insertion at the end | Adding a node into circular singly linked list at the end. |

**Deletion & Traversing**

| SN | Operation | Description |
|---|---|---|
| 1 | Deletion at beginning | Removing the node from circular singly linked list at the beginning. |
| 2 | Deletion at the end | Removing the node from circular singly linked list at the end. |
| 3 | Searching | Compare each element of the node with the given item and return the location at |

| | | |
|---|---|---|
| | | which the item is present in the list otherwise return null. |
| 4 | Traversing | Visiting each element of the list at least once in order to perform some specific operation. |

Menu-driven program in C implementing all operations
**on circular singly linked list**

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *head;
9.
10.      void beginsert ();
11.      void lastinsert ();
12.      void randominsert();
13.      void begin_delete();
14.      void last_delete();
15.      void random_delete();
16.      void display();
17.      void search();
```

```
18.     void main ()
19.     {
20.         int choice =0;
21.         while(choice != 7)
22.         {
23.             printf("\n*********Main
    Menu*********\n");
24.             printf("\nChoose one option from the following
    list ...\n");
25.             printf("\n==========================
    ====================\n");
26.             printf("\n1.Insert in begining\n2.Insert at
    last\n3.Delete from Beginning\n4.Delete from
    last\n5.Search for an element\n6.Show\n7.Exit\n");
27.             printf("\nEnter your choice?\n");
28.             scanf("\n%d",&choice);
29.             switch(choice)
30.             {
31.                 case 1:
32.                 beginsert();
33.                 break;
34.                 case 2:
35.                 lastinsert();
36.                 break;
37.                 case 3:
38.                 begin_delete();
39.                 break;
40.                 case 4:
41.                 last_delete();
42.                 break;
43.                 case 5:
44.                 search();
45.                 break;
```

```
46.                 case 6:
47.                 display();
48.                 break;
49.                 case 7:
50.                 exit(0);
51.                 break;
52.                 default:
53.                 printf("Please enter valid choice..");
54.             }
55.         }
56.     }
57.     void beginsert()
58.     {
59.         struct node *ptr,*temp;
60.         int item;
61.         ptr = (struct node *)malloc(sizeof(struct node));
62.         if(ptr == NULL)
63.         {
64.             printf("\nOVERFLOW");
65.         }
66.         else
67.         {
68.             printf("\nEnter the node data?");
69.             scanf("%d",&item);
70.             ptr -> data = item;
71.             if(head == NULL)
72.             {
73.                 head = ptr;
74.                 ptr -> next = head;
75.             }
76.             else
77.             {
78.                 temp = head;
```

```
79.                 while(temp->next != head)
80.                     temp = temp->next;
81.                 ptr->next = head;
82.                 temp -> next = ptr;
83.                 head = ptr;
84.             }
85.             printf("\nnode inserted\n");
86.         }
87.
88.     }
89.     void lastinsert()
90.     {
91.         struct node *ptr,*temp;
92.         int item;
93.         ptr = (struct node *)malloc(sizeof(struct node));
94.         if(ptr == NULL)
95.         {
96.             printf("\nOVERFLOW\n");
97.         }
98.         else
99.         {
100.            printf("\nEnter Data?");
101.            scanf("%d",&item);
102.            ptr->data = item;
103.            if(head == NULL)
104.            {
105.                head = ptr;
106.                ptr -> next = head;
107.            }
108.            else
109.            {
110.                temp = head;
111.                while(temp -> next != head)
```

```
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }

        printf("\nnode inserted\n");
    }

}

void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }

    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
```

```
145.            printf("\nnode deleted\n");
146.
147.        }
148.    }
149.    void last_delete()
150.    {
151.        struct node *ptr, *preptr;
152.        if(head==NULL)
153.        {
154.            printf("\nUNDERFLOW");
155.        }
156.        else if (head ->next == head)
157.        {
158.            head = NULL;
159.            free(head);
160.            printf("\nnode deleted\n");
161.
162.        }
163.        else
164.        {
165.            ptr = head;
166.            while(ptr ->next != head)
167.            {
168.                preptr=ptr;
169.                ptr = ptr->next;
170.            }
171.            preptr->next = ptr -> next;
172.            free(ptr);
173.            printf("\nnode deleted\n");
174.
175.        }
176.    }
177.
```

```
178.    void search()
179.    {
180.        struct node *ptr;
181.        int item,i=0,flag=1;
182.        ptr = head;
183.        if(ptr == NULL)
184.        {
185.            printf("\nEmpty List\n");
186.        }
187.        else
188.        {
189.            printf("\nEnter item which you want to
    search?\n");
190.            scanf("%d",&item);
191.            if(head ->data == item)
192.            {
193.            printf("item found at location %d",i+1);
194.            flag=0;
195.            }
196.            else
197.            {
198.            while (ptr->next != head)
199.            {
200.                if(ptr->data == item)
201.                {
202.                    printf("item found at location %d ",i+1);
203.                    flag=0;
204.                    break;
205.                }
206.                else
207.                {
208.                    flag=1;
209.                }
```

```
210.                i++;
211.                ptr = ptr -> next;
212.            }
213.            }
214.            if(flag != 0)
215.            {
216.                printf("Item not found\n");
217.            }
218.        }
219.
220.    }
221.
222.    void display()
223.    {
224.        struct node *ptr;
225.        ptr=head;
226.        if(head == NULL)
227.        {
228.            printf("\nnothing to print");
229.        }
230.        else
231.        {
232.            printf("\n printing values ... \n");
233.
234.            while(ptr -> next != head)
235.            {
236.
237.                printf("%d\n", ptr -> data);
238.                ptr = ptr -> next;
239.            }
240.            printf("%d\n", ptr -> data);
241.        }
242.
```

```
243.    }
```

**Output:**

```
*********Main Menu*********

Choose one option from the following list ...

==========================================
===

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
1

Enter the node data?10
```

node inserted

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search for an element

6.Show

7.Exit

Enter your choice?

2

Enter Data?20

node inserted

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search for an element

6.Show

7.Exit

Enter your choice?

2

Enter Data?30

node inserted

**********Main Menu**********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search for an element

6.Show

7.Exit

Enter your choice?

3

node deleted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search for an element

6.Show

7.Exit

Enter your choice?

4

node deleted

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search for an element

6.Show

7.Exit

Enter your choice?

5

```
Enter item which you want to search?
20
item found at location 1
**********Main Menu**********

Choose one option from the following list ...

==========================================
===

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
6
```

printing values ...

20

**********Main Menu**********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search for an element

6.Show

7.Exit

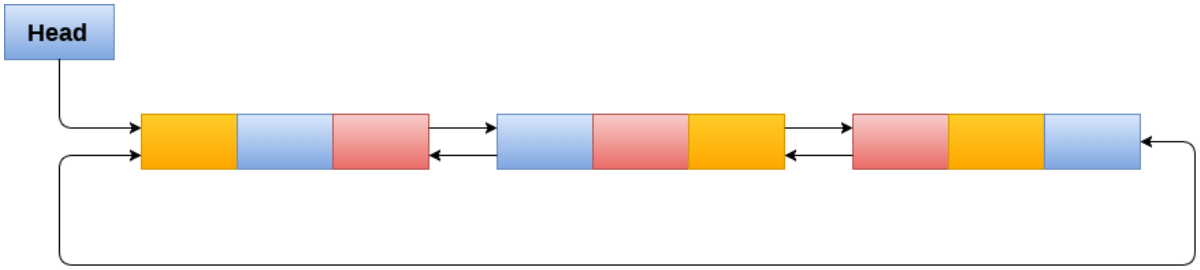
Enter your choice?

7

## Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



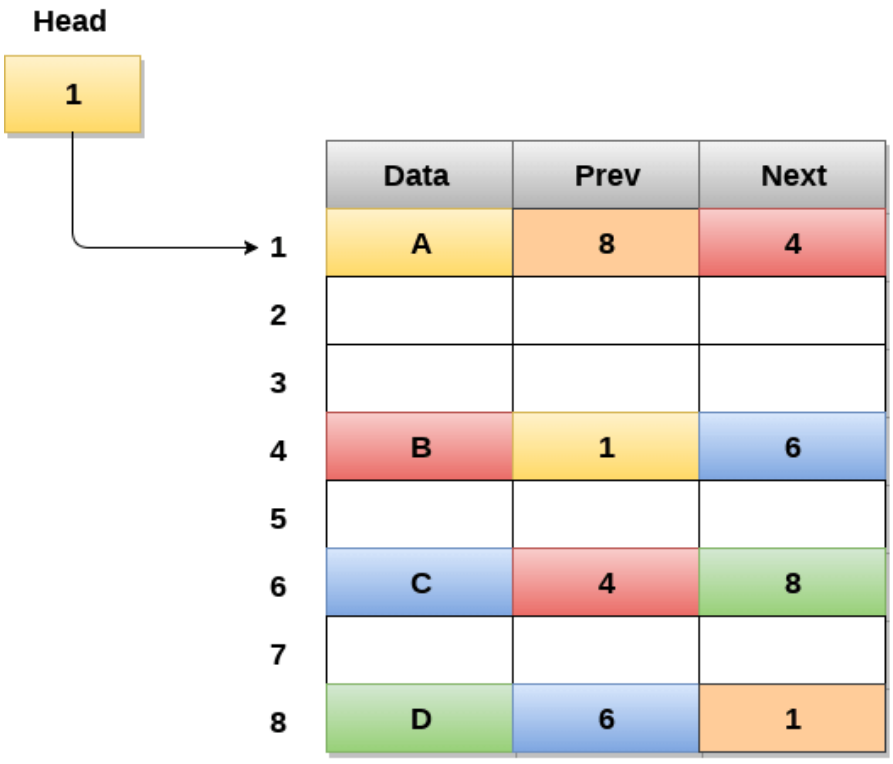

**Circular Doubly Linked List**

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1.

In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.




**Memory Representation of a Circular Doubly linked list**

Operations on circular doubly linked list :
There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list is described in the following table.

| SN | Operation | Description |
|---|---|---|

| | | |
|---|---|---|
| 1 | Insertion at beginning | Adding a node in circular doubly linked list at the beginning. |
| 2 | Insertion at end | Adding a node in circular doubly linked list at the end. |
| 3 | Deletion at beginning | Removing a node in circular doubly linked list from beginning. |
| 4 | Deletion at end | Removing a node in circular doubly linked list at the end. |

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.
C program to implement all the operations on circular doubly linked list

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     struct node *prev;
```

```
6.    struct node *next;
7.    int data;
8. };
9. struct node *head;
10.     void insertion_beginning();
11.     void insertion_last();
12.     void deletion_beginning();
13.     void deletion_last();
14.     void display();
15.     void search();
16.     void main ()
17.     {
18.     int choice =0;
19.         while(choice != 9)
20.         {
21.             printf("\n*********Main
  Menu*********\n");
22.             printf("\nChoose one option from the following
  list ...\n");
23.             printf("\n==========================
  ====================\n");
24.             printf("\n1.Insert in Beginning\n2.Insert at
  last\n3.Delete from Beginning\n4.Delete from
  last\n5.Search\n6.Show\n7.Exit\n");
25.             printf("\nEnter your choice?\n");
26.             scanf("\n%d",&choice);
27.             switch(choice)
28.             {
29.                 case 1:
30.                 insertion_beginning();
31.                 break;
32.                 case 2:
33.                         insertion_last();
```

```
34.                 break;
35.                 case 3:
36.                 deletion_beginning();
37.                 break;
38.                 case 4:
39.                 deletion_last();
40.                 break;
41.                 case 5:
42.                 search();
43.                 break;
44.                 case 6:
45.                 display();
46.                 break;
47.                 case 7:
48.                 exit(0);
49.                 break;
50.                 default:
51.                 printf("Please enter valid choice..");
52.             }
53.          }
54.      }
55.      void insertion_beginning()
56.      {
57.        struct node *ptr,*temp;
58.        int item;
59.        ptr = (struct node *)malloc(sizeof(struct node));
60.        if(ptr == NULL)
61.        {
62.            printf("\nOVERFLOW");
63.        }
64.        else
65.        {
66.         printf("\nEnter Item value");
```

```
67.          scanf("%d",&item);
68.          ptr->data=item;
69.         if(head==NULL)
70.         {
71.           head = ptr;
72.           ptr -> next = head;
73.           ptr -> prev = head;
74.         }
75.         else
76.         {
77.            temp = head;
78.          while(temp -> next != head)
79.          {
80.             temp = temp -> next;
81.          }
82.          temp -> next = ptr;
83.          ptr -> prev = temp;
84.          head -> prev = ptr;
85.          ptr -> next = head;
86.          head = ptr;
87.         }
88.         printf("\nNode inserted\n");
89.       }
90.
91.       }
92.       void insertion_last()
93.       {
94.         struct node *ptr,*temp;
95.         int item;
96.         ptr = (struct node *) malloc(sizeof(struct node));
97.         if(ptr == NULL)
98.         {
99.            printf("\nOVERFLOW");
```

```
100.        }
101.        else
102.        {
103.            printf("\nEnter value");
104.            scanf("%d",&item);
105.             ptr->data=item;
106.            if(head == NULL)
107.            {
108.                head = ptr;
109.                ptr -> next = head;
110.                ptr -> prev = head;
111.            }
112.            else
113.            {
114.               temp = head;
115.               while(temp->next !=head)
116.               {
117.                   temp = temp->next;
118.               }
119.               temp->next = ptr;
120.               ptr ->prev=temp;
121.               head -> prev = ptr;
122.             ptr -> next = head;
123.              }
124.        }
125.          printf("\nnode inserted\n");
126.    }
127.
128.    void deletion_beginning()
129.    {
130.        struct node *temp;
131.        if(head == NULL)
132.        {
```

```
133.            printf("\n UNDERFLOW");
134.        }
135.        else if(head->next == head)
136.        {
137.            head = NULL;
138.            free(head);
139.            printf("\nnode deleted\n");
140.        }
141.        else
142.        {
143.            temp = head;
144.            while(temp -> next != head)
145.            {
146.                temp = temp -> next;
147.            }
148.            temp -> next = head -> next;
149.            head -> next -> prev = temp;
150.            free(head);
151.            head = temp -> next;
152.        }
153.
154.    }
155.    void deletion_last()
156.    {
157.        struct node *ptr;
158.        if(head == NULL)
159.        {
160.            printf("\n UNDERFLOW");
161.        }
162.        else if(head->next == head)
163.        {
164.            head = NULL;
165.            free(head);
```

```
166.            printf("\nnode deleted\n");
167.        }
168.        else
169.        {
170.            ptr = head;
171.            if(ptr->next != head)
172.            {
173.                ptr = ptr -> next;
174.            }
175.            ptr -> prev -> next = head;
176.            head -> prev = ptr -> prev;
177.            free(ptr);
178.            printf("\nnode deleted\n");
179.        }
180.    }
181.
182.    void display()
183.    {
184.        struct node *ptr;
185.        ptr=head;
186.        if(head == NULL)
187.        {
188.            printf("\nnothing to print");
189.        }
190.        else
191.        {
192.            printf("\n printing values ... \n");
193.
194.            while(ptr -> next != head)
195.            {
196.
197.                printf("%d\n", ptr -> data);
198.                ptr = ptr -> next;
```

```
199.            }
200.            printf("%d\n", ptr -> data);
201.        }
202.
203.    }
204.
205.    void search()
206.    {
207.        struct node *ptr;
208.        int item,i=0,flag=1;
209.        ptr = head;
210.        if(ptr == NULL)
211.        {
212.            printf("\nEmpty List\n");
213.        }
214.        else
215.        {
216.            printf("\nEnter item which you want to
    search?\n");
217.            scanf("%d",&item);
218.            if(head ->data == item)
219.            {
220.            printf("item found at location %d",i+1);
221.            flag=0;
222.            }
223.            else
224.            {
225.            while (ptr->next != head)
226.            {
227.                if(ptr->data == item)
228.                {
229.                    printf("item found at location %d ",i+1);
230.                    flag=0;
```

```
231.                 break;
232.             }
233.             else
234.             {
235.                 flag=1;
236.             }
237.             i++;
238.             ptr = ptr -> next;
239.         }
240.         }
241.         if(flag != 0)
242.         {
243.             printf("Item not found\n");
244.         }
245.       }
246.
247.    }
```

**Output:**

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

1

Enter Item value123

Node inserted

**********Main Menu**********

Choose one option from the following list ...

==========

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

2

Enter value234

node inserted

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

1

Enter Item value90

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

2

Enter value80

node inserted

**********Main Menu**********

Choose one option from the following list ...

==========
===

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

3

**********Main Menu**********

Choose one option from the following list ...

==========
===

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

4

node deleted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

6

printing values ...

123

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

5

Enter item which you want to search?

123

item found at location 1

**********Main Menu**********

Choose one option from the following list ...

==========

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

7