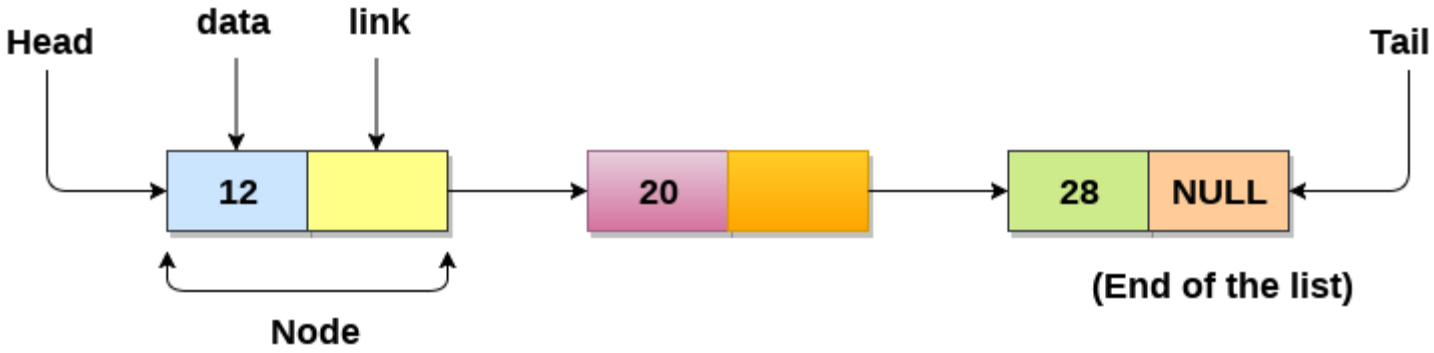
Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

Data Structur	Time Complexity	Space Compleit

e									y
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. };
6. struct node *head, *ptr;
7. ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after

		which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Insertion in singly linked list at beginning

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to inser a new node in the list at beginning.

- Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.

1. ptr = (struct node *) malloc(sizeof(struct node *));
2. ptr → data = item

- Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.

1. ptr->next = head;

- At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

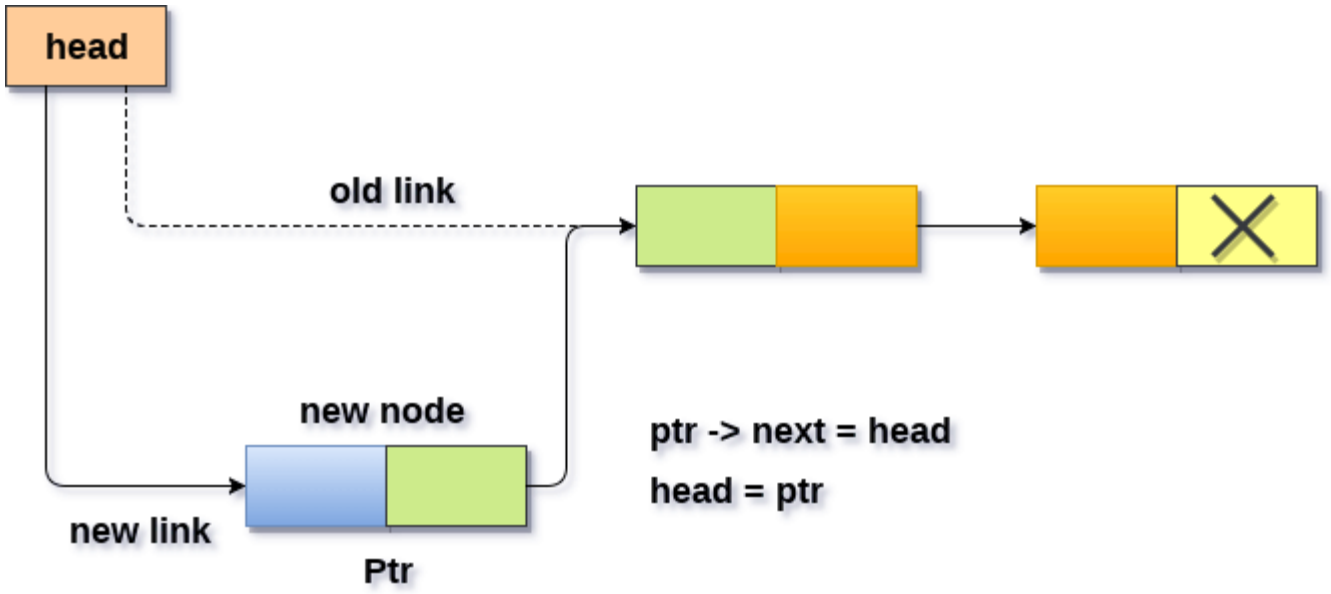
1. head = ptr;

Algorithm

- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 7
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR → NEXT
- **Step 4:** SET NEW_NODE → DATA = VAL
- **Step 5:** SET NEW_NODE → NEXT = HEAD
- **Step 6:** SET HEAD = NEW_NODE
- **Step 7:** EXIT



C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void beginsert(int);
4. struct node
5. {
6.     int data;
7.     struct node *next;
8. };
9. struct node *head;
10. void main ()
11. {
12.     int choice,item;
13.     do
14.     {
15.         printf("\nEnter the item which you want to insert?\n");
16.         scanf("%d",&item);
17.         beginsert(item);
18.         printf("\nPress 0 to insert more ?\n");
19.         scanf("%d",&choice);
20.     }while(choice == 0);
21. }
22. void beginsert(int item)
23.     {
24.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
```

```
25.         if(ptr == NULL)
26.         {
27.             printf("\nOVERFLOW\n");
28.         }
29.         else
30.         {
31.             ptr->data = item;
32.             ptr->next = head;
33.             head = ptr;
34.             printf("\nNode inserted\n");
35.         }
36.
37.     }
```

Output

```
Enter the item which you want to insert?
12

Node inserted

Press 0 to insert more ?
0

Enter the item which you want to insert?
23

Node inserted

Press 0 to insert more ?
2
```

Insertion in singly linked list at the end

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list

2. The node is being added to the end of the linked list

in the first case,

- The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.

```
1. ptr->data = item;
2.          ptr -> next = NULL;
```

Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.

```
Head = ptr
```

In the second case,

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.

```
1. Temp = head
```

- Then, traverse through the entire linked list using the statements:

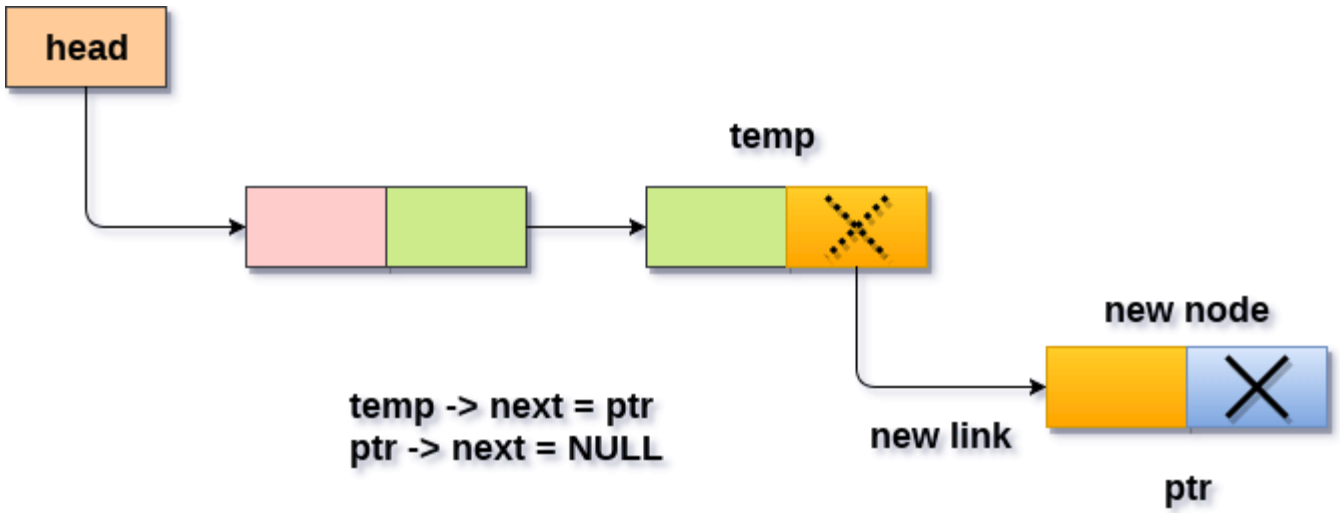
```
1. while (temp→ next != NULL)
2.          temp = temp → next;
```

- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**. We need to make the next part of the temp node (which is currently the last node of the list) point to the new node (ptr) .

```
1. temp = head;
2.       while (temp -> next != NULL)
3.       {
4.          temp = temp -> next;
5.       }
6.       temp->next = ptr;
7.       ptr->next = NULL;
```

Algorithm

- **Step 1:** IF PTR = NULL Write OVERFLOW
  Go to Step 1
  [END OF IF]
- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR - > NEXT
- **Step 4:** SET NEW_NODE - > DATA = VAL
- **Step 5:** SET NEW_NODE - > NEXT = NULL
- **Step 6:** SET PTR = HEAD
- **Step 7:** Repeat Step 8 while PTR - > NEXT != NULL
- **Step 8:** SET PTR = PTR - > NEXT
  [END OF LOOP]
- **Step 9:** SET PTR - > NEXT = NEW_NODE
- **Step 10:** EXIT



Inserting node at the last into a non-empty list

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void lastinsert(int);
4. struct node
5. {
```

```
6.      int data;
7.      struct node *next;
8.  };
9.  struct node *head;
10. void main ()
11. {
12.     int choice,item;
13.     do
14.     {
15.         printf("\nEnter the item which you want to insert?\n");
16.         scanf("%d",&item);
17.         lastinsert(item);
18.         printf("\nPress 0 to insert more ?\n");
19.         scanf("%d",&choice);
20.     }while(choice == 0);
21. }
22. void lastinsert(int item)
23.     {
24.         struct node *ptr = (struct node*)malloc(sizeof(struct node));
25.         struct node *temp;
26.         if(ptr == NULL)
27.         {
28.             printf("\nOVERFLOW");
29.         }
30.         else
31.         {
32.             ptr->data = item;
33.             if(head == NULL)
34.             {
35.                 ptr -> next = NULL;
36.                 head = ptr;
37.                 printf("\nNode inserted");
38.             }
39.             else
40.             {
41.                 temp = head;
42.                 while (temp -> next != NULL)
43.                 {
44.                     temp = temp -> next;
45.                 }
```

```
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
```

Output

```
Enter the item which you want to insert?
12

Node inserted
Press 0 to insert more ?
0

Enter the item which you want to insert?
23

Node inserted
Press 0 to insert more ?
2
```

Insertion in singly linked list after specified Node

- In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. This will be done by using the following statements.

```
emp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                return;
            }

        }
```

- Allocate the space for the new node and add the item to the data part of it. This will be done by using the following statements.

1. ptr = (struct node *) malloc (sizeof(struct node));
2. ptr->data = item;
    - Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp). This will be done by using the following statements.

1. ptr→ next = temp → next

   now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

1. temp ->next = ptr;

Algorithm

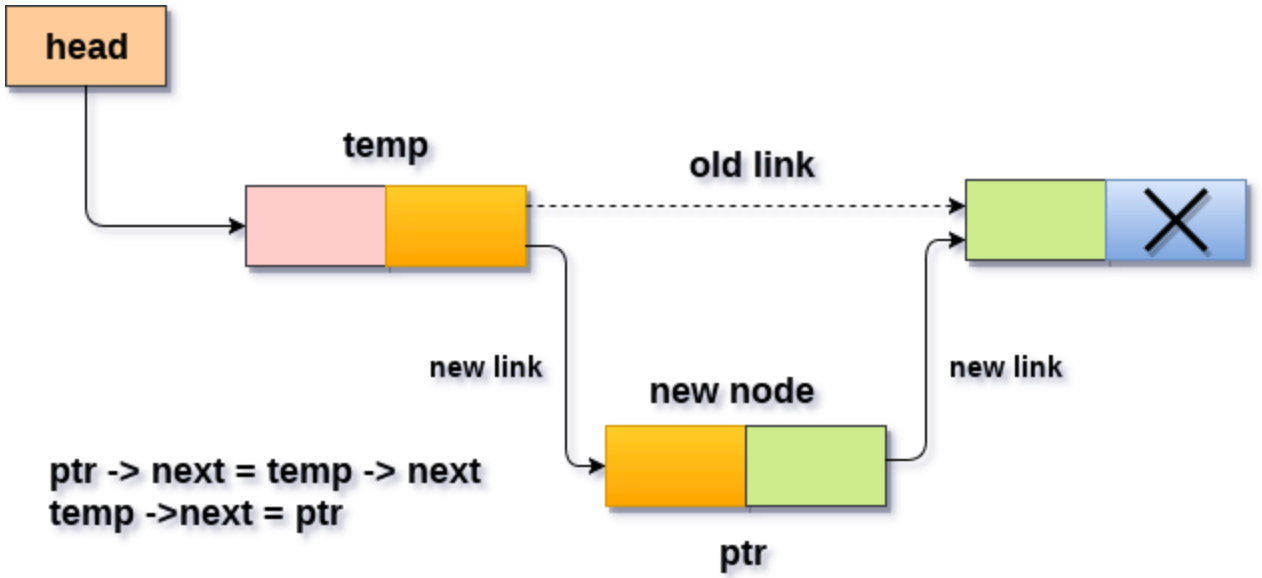
- **STEP 1:** IF PTR = NULL

  WRITE OVERFLOW
  GOTO STEP 12
  END OF IF

- **STEP 2:** SET NEW_NODE = PTR
- **STEP 3:** NEW_NODE → DATA = VAL
- **STEP 4:** SET TEMP = HEAD
- **STEP 5:** SET I = 0
- **STEP 6:** REPEAT STEP 5 AND 6 UNTIL I<loc< li=""></loc<>
- **STEP 7:** TEMP = TEMP → NEXT
- **STEP 8:** IF TEMP = NULL

  WRITE "DESIRED NODE NOT PRESENT"
  GOTO STEP 12
  END OF IF
  END OF LOOP

- **STEP 9:** PTR → NEXT = TEMP → NEXT
- **STEP 10:** TEMP → NEXT = PTR
- **STEP 11:** SET PTR = NEW_NODE
- **STEP 12:** EXIT



C Function

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  void randominsert(int);
4.  void create(int);
5.  struct node
6.  {
7.      int data;
8.      struct node *next;
9.  };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item,loc;
14.     do
15.     {
16.         printf("\nEnter the item which you want to insert?\n");
17.         scanf("%d",&item);
18.         if(head == NULL)
19.         {
20.             create(item);
21.         }
22.         else
23.         {
24.             randominsert(item);
25.         }
```

```
26.         printf("\nPress 0 to insert more ?\n");
27.         scanf("%d",&choice);
28.     }while(choice == 0);
29. }
30. void create(int item)
31. {
32.
33.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
34.         if(ptr == NULL)
35.         {
36.             printf("\nOVERFLOW\n");
37.         }
38.         else
39.         {
40.             ptr->data = item;
41.             ptr->next = head;
42.             head = ptr;
43.             printf("\nNode inserted\n");
44.         }
45. }
46. void randominsert(int item)
47.     {
48.         struct node *ptr = (struct node *) malloc (sizeof(struct node));
49.         struct node *temp;
50.         int i,loc;
51.         if(ptr == NULL)
52.         {
53.             printf("\nOVERFLOW");
54.         }
55.         else
56.         {
57.
58.             printf("Enter the location");
59.             scanf("%d",&loc);
60.             ptr->data = item;
61.             temp=head;
62.             for(i=0;i<loc;i++)
63.             {
64.                 temp = temp->next;
65.                 if(temp == NULL)
```

```
66.             {
67.                 printf("\ncan't insert\n");
68.                 return;
69.             }
70.
71.         }
72.         ptr ->next = temp ->next;
73.         temp ->next = ptr;
74.         printf("\nNode inserted");
75.       }
76.
77.    }
```

Output

```
Enter the item which you want to insert?
12

Node inserted

Press 0 to insert more ?
2
```

Deletion in singly linked list at beginning

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements.

1. ptr = head;

2. head = ptr->next;

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

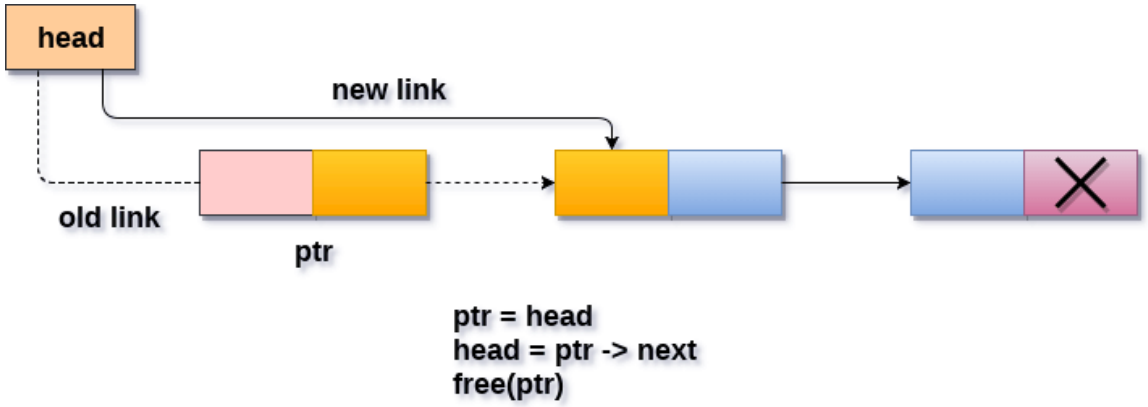
1. free(ptr)

Algorithm

- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 5
  [END OF IF]

- **Step 2:** SET PTR = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** EXIT



Deleting a node from the beginning

C function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void begdelete();
5. struct node
6. {
7.     int data;
```

```
8.      struct node *next;
9.  };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("\n1.Append List\n2.Delete node\n3.Exit\n4.Enter your choice?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
20.             case 1:
21.             printf("\nEnter the item\n");
22.             scanf("%d",&item);
23.             create(item);
24.             break;
25.             case 2:
26.             begdelete();
27.             break;
28.             case 3:
29.             exit(0);
30.             break;
31.             default:
32.             printf("\nPlease enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38.     {
39.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
40.         if(ptr == NULL)
41.         {
42.             printf("\nOVERFLOW\n");
43.         }
44.         else
45.         {
46.             ptr->data = item;
47.             ptr->next = head;
```

```
48.             head = ptr;
49.             printf("\nNode inserted\n");
50.         }
51.
52.     }
53. void begdelete()
54.     {
55.         struct node *ptr;
56.         if(head == NULL)
57.         {
58.             printf("\nList is empty");
59.         }
60.         else
61.         {
62.             ptr = head;
63.             head = ptr->next;
64.             free(ptr);
65.             printf("\n Node deleted from the begining ...");
66.         }
67.     }
```

Output

```
1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted

1.Append List
2.Delete node
3.Exit
4.Enter your choice?2

 Node deleted from the begining ...
```

Deletion in singly linked list at the end

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

1. ptr = head
2. head = NULL
3. free(ptr)

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

this all will be done by using the following statements.

```
ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
```

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free. It will be done by using the following statements.

```
ptr1->next = NULL;
        free(ptr);
```

Algorithm

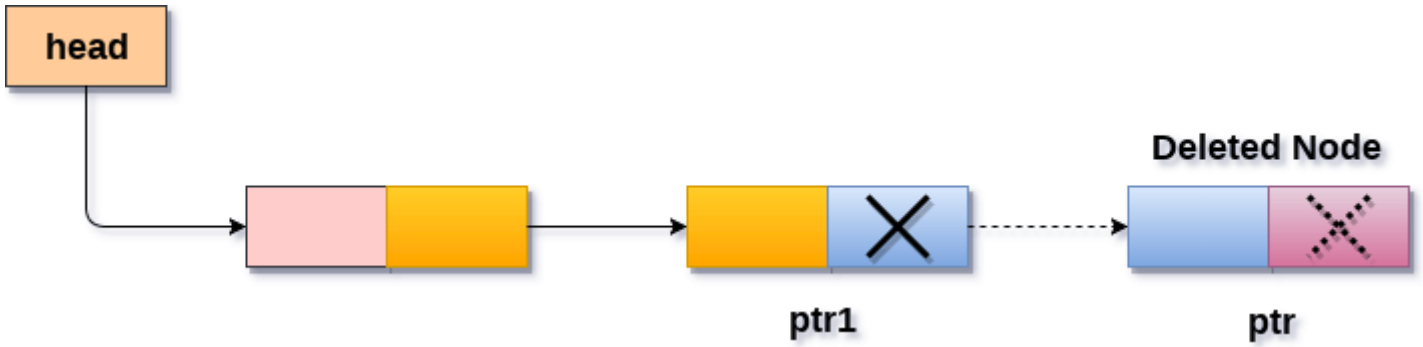
- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 8
  [END OF IF]

- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while PTR -> NEXT!= NULL
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT

  [END OF LOOP]

- **Step 6:** SET PREPTR -> NEXT = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT



Deleting a node from the last

C Function :

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  void create(int);
4.  void end_delete();
5.  struct node
6.  {
7.      int data;
8.      struct node *next;
9.  };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("\n1.Append List\n2.Delete node\n3.Exit\n4.Enter your choice?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
```

```
20.             case 1:
21.             printf("\nEnter the item\n");
22.             scanf("%d",&item);
23.             create(item);
24.             break;
25.             case 2:
26.             end_delete();
27.             break;
28.             case 3:
29.             exit(0);
30.             break;
31.             default:
32.             printf("\nPlease enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38.     {
39.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
40.         if(ptr == NULL)
41.         {
42.             printf("\nOVERFLOW\n");
43.         }
44.         else
45.         {
46.             ptr->data = item;
47.             ptr->next = head;
48.             head = ptr;
49.             printf("\nNode inserted\n");
50.         }
51.
52.     }
53. void end_delete()
54.     {
55.         struct node *ptr,*ptr1;
56.         if(head == NULL)
57.         {
58.             printf("\nlist is empty");
59.         }
```

```
60.     else if(head -> next == NULL)
61.     {
62.         head = NULL;
63.         free(head);
64.         printf("\nOnly node of the list deleted ...");
65.     }
66.
67.     else
68.     {
69.         ptr = head;
70.         while(ptr->next != NULL)
71.             {
72.                 ptr1 = ptr;
73.                 ptr = ptr ->next;
74.             }
75.             ptr1->next = NULL;
76.             free(ptr);
77.             printf("\n Deleted Node from the last ...");
78.         }
79.     }
```

Output

```
1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
12

Node inserted

1.Append List
2.Delete node
3.Exit
4.Enter your choice?2

Only node of the list deleted ...
```

Deletion in singly linked list after the specified node :

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.

Use the following statements to do so.

```
1. ptr=head;
2.         for(i=0;i<loc;i++)
3.         {
4.             ptr1 = ptr;
5.             ptr = ptr->next;
6.
7.             if(ptr == NULL)
8.             {
9.                 printf("\nThere are less than %d elements in the list..",loc);
10.                return;
11.            }
12.        }
```

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

This will be done by using the following statements.

```
1. ptr1 ->next = ptr ->next;
2.         free(ptr);
```

Algorithm

- **STEP 1:** IF HEAD = NULL

  WRITE UNDERFLOW
  GOTO STEP 10
  END OF IF

- **STEP 2:** SET TEMP = HEAD
- **STEP 3:** SET I = 0
- **STEP 4:** REPEAT STEP 5 TO 8 UNTIL I<loc< li=""></loc<>
- **STEP 5:** TEMP1 = TEMP
- **STEP 6:** TEMP = TEMP → NEXT

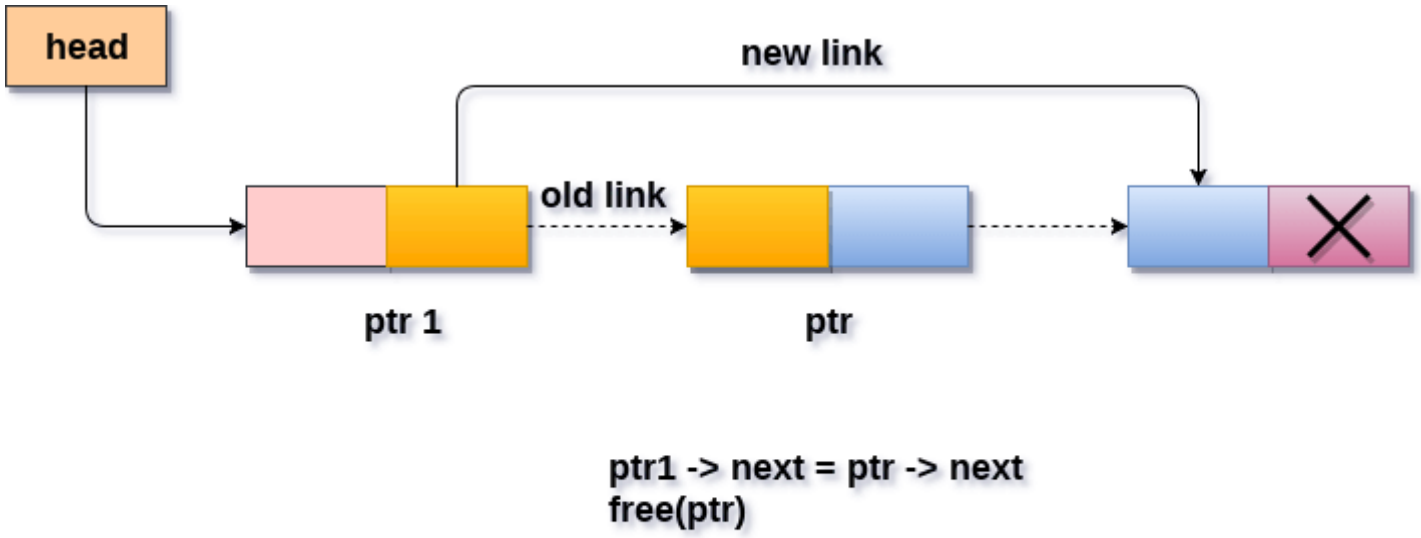
- **STEP 7:** IF TEMP = NULL

  WRITE "DESIRED NODE NOT PRESENT"
  GOTO STEP 12
  END OF IF

- **STEP 8:** I = I+1

  END OF LOOP

- **STEP 9:** TEMP1 → NEXT = TEMP → NEXT
- **STEP 10:** FREE TEMP
- **STEP 11:** EXIT



Deletion a node from specified position

C function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void delete_specified();
5. struct node
6. {
7.     int data;
8.     struct node *next;
9. };
```

```
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("\n1.Append List\n2.Delete node\n3.Exit\n4.Enter your choice?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
20.             case 1:
21.             printf("\nEnter the item\n");
22.             scanf("%d",&item);
23.             create(item);
24.             break;
25.             case 2:
26.             delete_specified();
27.             break;
28.             case 3:
29.             exit(0);
30.             break;
31.             default:
32.             printf("\nPlease enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38.     {
39.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
40.         if(ptr == NULL)
41.         {
42.             printf("\nOVERFLOW\n");
43.         }
44.         else
45.         {
46.             ptr->data = item;
47.             ptr->next = head;
48.             head = ptr;
49.             printf("\nNode inserted\n");
```

```
50.       }
51.
52.    }
53. void delete_specified()
54.    {
55.       struct node *ptr, *ptr1;
56.       int loc,i;
57.       scanf("%d",&loc);
58.       ptr=head;
59.       for(i=0;i<loc;i++)
60.       {
61.          ptr1 = ptr;
62.          ptr = ptr->next;
63.
64.          if(ptr == NULL)
65.          {
66.             printf("\nThere are less than %d elements in the list..\n",loc);
67.             return;
68.          }
69.       }
70.       ptr1 ->next = ptr ->next;
71.       free(ptr);
72.       printf("\nDeleted %d node ",loc);
73.    }
```

Output

```
1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
12

Node inserted

1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted

1.Append List
2.Delete node
```

```
3.Exit
4.Enter your choice?2
12

There are less than 12 elements in the list..

1.Append List
2.Delete node
3.Exit
4.Enter your choice?2
1

Deleted 1 node
```

Traversing in singly linked list

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
1. ptr = head;
2.       while (ptr!=NULL)
3.          {
4.             ptr = ptr -> next;
5.          }
```

Algorithm

- **STEP 1:** SET PTR = HEAD
- **STEP 2:** IF PTR = NULL

  WRITE "EMPTY LIST"
  GOTO STEP 7
  END OF IF

- **STEP 4:** REPEAT STEP 5 AND 6 UNTIL PTR != NULL
- **STEP 5:** PRINT PTR→ DATA
- **STEP 6:** PTR = PTR → NEXT

  [END OF LOOP]

- **STEP 7:** EXIT

C function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void traverse();
5. struct node
6. {
7.     int data;
8.     struct node *next;
9. };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("\n1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
20.             case 1:
21.             printf("\nEnter the item\n");
22.             scanf("%d",&item);
23.             create(item);
24.             break;
25.             case 2:
26.             traverse();
27.             break;
28.             case 3:
29.             exit(0);
30.             break;
31.             default:
32.             printf("\nPlease enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38.     {
```

```
39.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
40.         if(ptr == NULL)
41.         {
42.             printf("\nOVERFLOW\n");
43.         }
44.         else
45.         {
46.             ptr->data = item;
47.             ptr->next = head;
48.             head = ptr;
49.             printf("\nNode inserted\n");
50.         }
51.
52.     }
53. void traverse()
54.     {
55.         struct node *ptr;
56.         ptr = head;
57.         if(ptr == NULL)
58.         {
59.             printf("Empty list..");
60.         }
61.         else
62.         {
63.             printf("printing values . . . . .\n");
64.             while (ptr!=NULL)
65.             {
66.                 printf("\n%d",ptr->data);
67.                 ptr = ptr -> next;
68.             }
69.         }
70.     }
```

Output

```
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted
```

```
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
233

Node inserted

1.Append List
2.Traverse
3.Exit
4.Enter your choice?2
printing values . . . . .

233
23
```

Searching in singly linked list

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL

  WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF

- **STEP 4:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL

- **STEP 5:** if ptr → data = item

  write i+1
  End of IF

- **STEP 6:** I = I + 1
- **STEP 7:** PTR = PTR → NEXT

  [END OF LOOP]

- **STEP 8:** EXIT

C function

```
1.    #include<stdio.h>
2.  #include<stdlib.h>
3.  void create(int);
4.  void search();
5.  struct node
6.  {
7.     int data;
8.     struct node *next;
9.  };
10. struct node *head;
11. void main ()
12. {
13.    int choice,item,loc;
14.    do
15.    {
16.       printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
17.       scanf("%d",&choice);
18.       switch(choice)
19.       {
20.          case 1:
21.          printf("\nEnter the item\n");
22.          scanf("%d",&item);
23.          create(item);
24.          break;
25.          case 2:
26.          search();
27.          case 3:
```

```
28.             exit(0);
29.             break;
30.             default:
31.             printf("\nPlease enter valid choice\n");
32.         }
33.
34.     }while(choice != 3);
35. }
36.     void create(int item)
37.     {
38.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
39.         if(ptr == NULL)
40.         {
41.             printf("\nOVERFLOW\n");
42.         }
43.         else
44.         {
45.             ptr->data = item;
46.             ptr->next = head;
47.             head = ptr;
48.             printf("\nNode inserted\n");
49.         }
50.
51.     }
52. void search()
53. {
54.     struct node *ptr;
55.     int item,i=0,flag;
56.     ptr = head;
57.     if(ptr == NULL)
58.     {
59.         printf("\nEmpty List\n");
60.     }
61.     else
62.     {
63.         printf("\nEnter item which you want to search?\n");
64.         scanf("%d",&item);
65.         while (ptr!=NULL)
66.         {
67.             if(ptr->data == item)
```

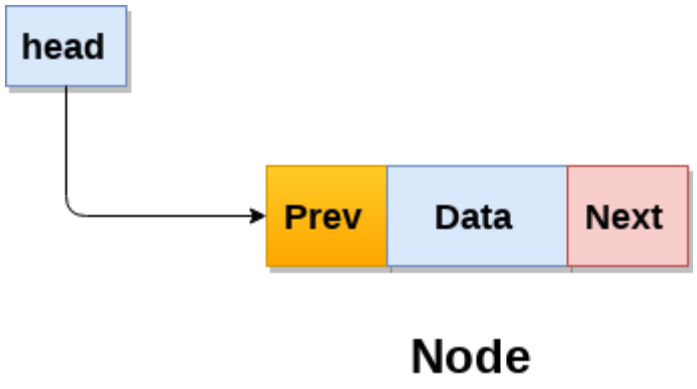
```
68.         {
69.             printf("item found at location %d ",i+1);
70.             flag=0;
71.         }
72.         else
73.         {
74.             flag=1;
75.         }
76.         i++;
77.         ptr = ptr -> next;
78.       }
79.       if(flag==1)
80.       {
81.           printf("Item not found\n");
82.       }
83.    }
84.
85. }
```

Output

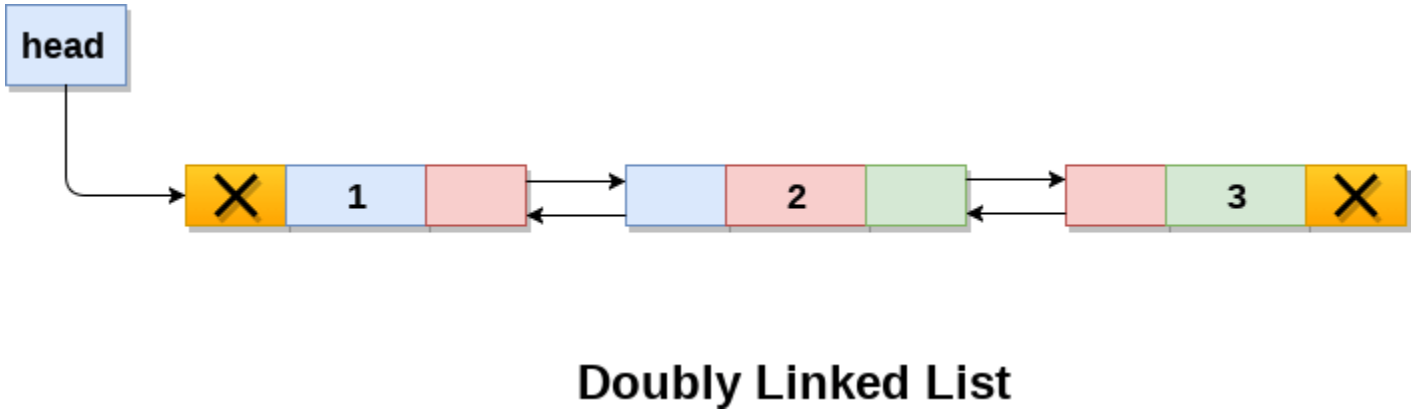
```
1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted

1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
34

Node inserted

1.Create
2.Search
3.Exit
4.Enter your choice?2

Enter item which you want to search?
34
item found at location 1
```

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
1. struct node
2. {
3.     struct node *prev;
4.     int data;
5.     struct node *next;
6. }
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

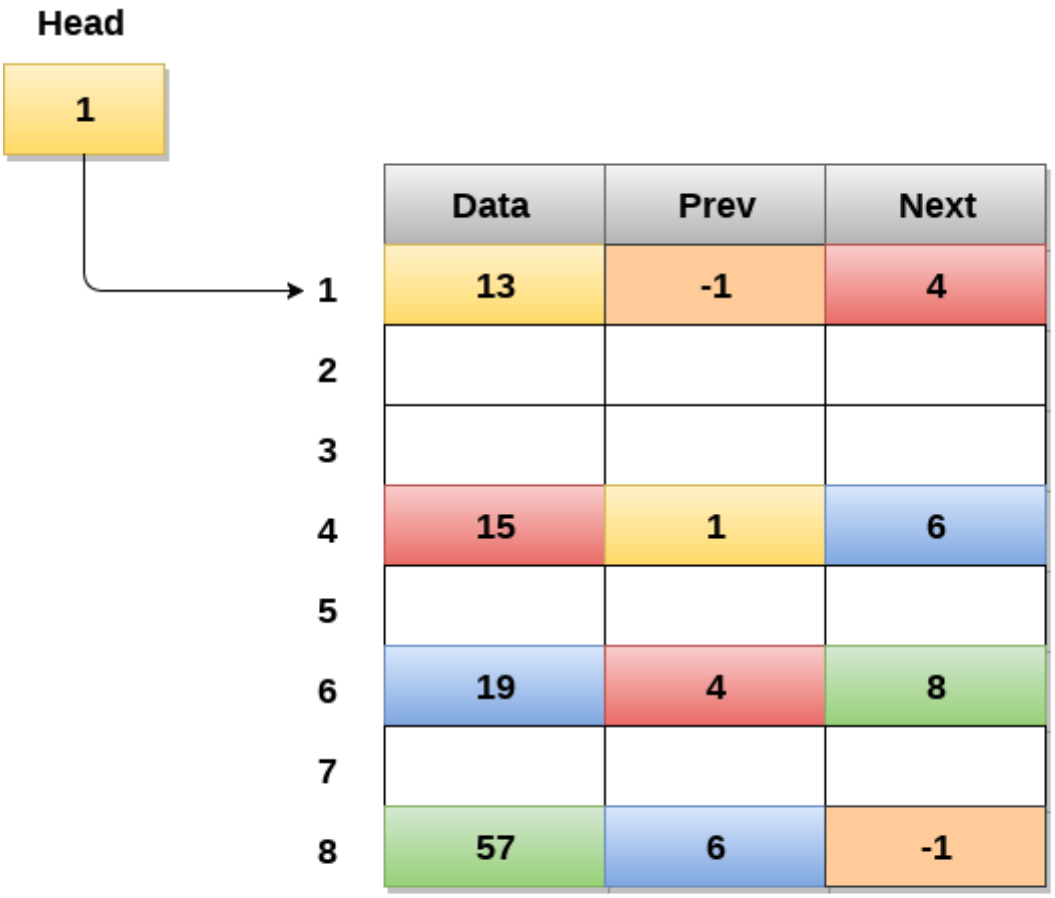
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

```
1. struct node
2. {
3.     struct node *prev;
4.     int data;
5.     struct node *next;
6. };
7. struct node *head;
```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description

1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory. This will be done by using the following statement.

1. ptr = (struct node *)malloc(sizeof(struct node));

- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

1. ptr->next = NULL;
2. ptr->prev=NULL;
3. ptr->data=item;
4. head=ptr;

- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.
- This will be done by using the following statements.

1. ptr->next = head;
2. head→prev=ptr;

Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

1. ptr→prev =NULL
2. head = ptr

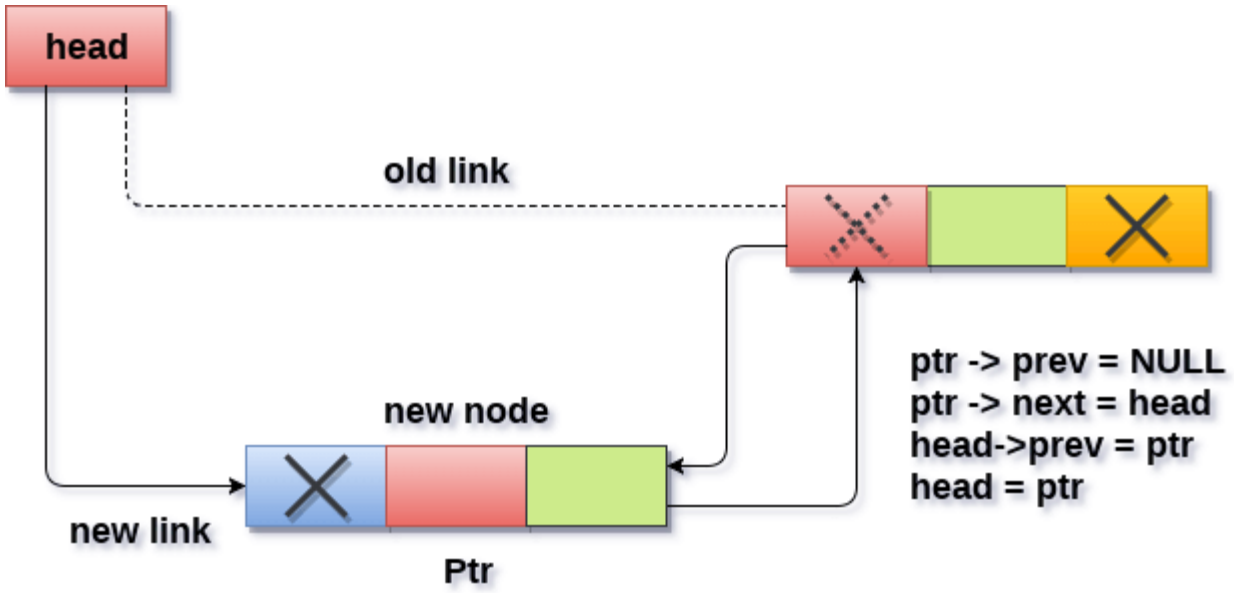
Algorithm :

- **Step 1:** IF ptr = NULL

  Write OVERFLOW
  Go to Step 9
  [END OF IF]

- **Step 2:** SET NEW_NODE = ptr
- **Step 3:** SET ptr = ptr -> NEXT

- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> PREV = NULL
- **Step 6:** SET NEW_NODE -> NEXT = START
- **Step 7:** SET head -> PREV = NEW_NODE
- **Step 8:** SET head = NEW_NODE
- **Step 9:** EXIT



Insertion into doubly linked list at beginning

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void insertbeginning(int);
4. struct node
5. {
6.     int data;
7.     struct node *next;
8.     struct node *prev;
9. };
10. struct node *head;
11. void main ()
```

```
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("\nEnter the item which you want to insert?\n");
17.         scanf("%d",&item);
18.         insertbeginning(item);
19.         printf("\nPress 0 to insert more ?\n");
20.         scanf("%d",&choice);
21.     }while(choice == 0);
22. }
23. void insertbeginning(int item)
24. {
25.
26.     struct node *ptr = (struct node *)malloc(sizeof(struct node));
27.     if(ptr == NULL)
28.     {
29.         printf("\nOVERFLOW");
30.     }
31.     else
32.     {
33.
34.
35.     if(head==NULL)
36.     {
37.         ptr->next = NULL;
38.         ptr->prev=NULL;
39.         ptr->data=item;
40.         head=ptr;
41.     }
42.     else
43.     {
44.         ptr->data=item;
45.         ptr->prev=NULL;
46.         ptr->next = head;
47.         head->prev=ptr;
48.         head=ptr;
49.     }
50. }
51.
```

52. }

Output

```
Enter the item which you want to insert?
12

Press 0 to insert more ?
0

Enter the item which you want to insert?
23

Press 0 to insert more ?
2
```

Output

Insertion in doubly linked list at the end

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.

1. ptr = (struct node *) malloc(sizeof(struct node));

- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

1. ptr->next = NULL;
2. ptr->prev=NULL;
3. ptr->data=item;
4. head=ptr;

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

```
1. Temp = head;
2. while (temp != NULL)
3. {
4. temp = temp → next;
5.     }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

```
1. temp→next =ptr;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

```
1. ptr → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
1. ptr → next = NULL
```

Algorithm

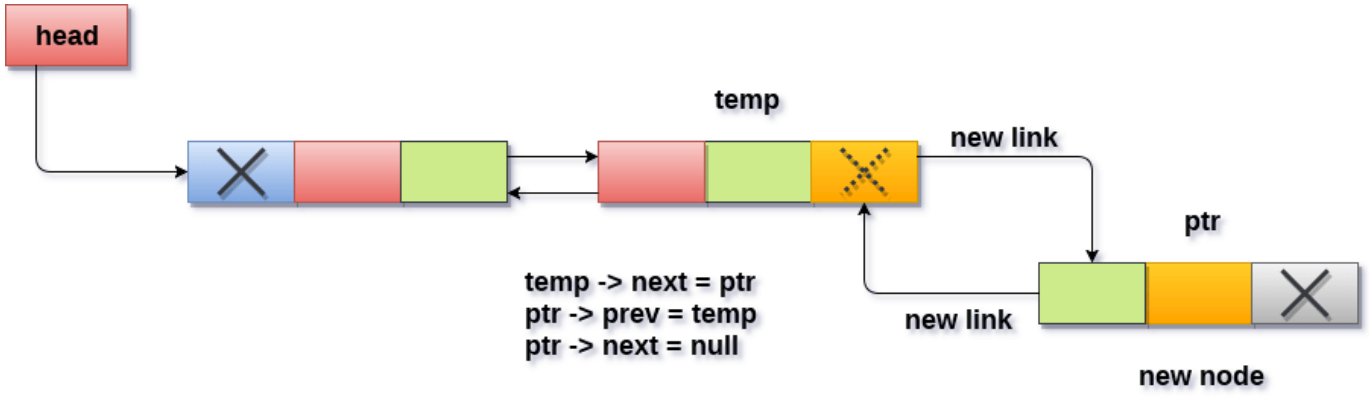
- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 11
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = NULL
- **Step 6:** SET TEMP = START
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != NULL
- **Step 8:** SET TEMP = TEMP -> NEXT

[END OF LOOP]

- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10C:** SET NEW_NODE -> PREV = TEMP
- **Step 11:** EXIT



Insertion into doubly linked list at the end

C Program

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void insertlast(int);
4. struct node
5. {
6.     int data;
7.     struct node *next;
8.     struct node *prev;
9. };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("\nEnter the item which you want to insert?\n");
17.         scanf("%d",&item);
18.         insertlast(item);
19.         printf("\nPress 0 to insert more ?\n");
```

```
20.         scanf("%d",&choice);
21.      }while(choice == 0);
22. }
23. void insertlast(int item)
24. {
25.
26.    struct node *ptr = (struct node *) malloc(sizeof(struct node));
27.    struct node *temp;
28.    if(ptr == NULL)
29.    {
30.        printf("\nOVERFLOW");
31.
32.    }
33.    else
34.    {
35.
36.         ptr->data=item;
37.        if(head == NULL)
38.        {
39.            ptr->next = NULL;
40.            ptr->prev = NULL;
41.            head = ptr;
42.        }
43.        else
44.        {
45.          temp = head;
46.          while(temp->next!=NULL)
47.          {
48.              temp = temp->next;
49.          }
50.          temp->next = ptr;
51.          ptr ->prev=temp;
52.          ptr->next = NULL;
53.        }
54. printf("\nNode Inserted\n");
55.
56.    }
57. }
```

Output

```
Enter the item which you want to insert?
12

Node Inserted

Press 0 to insert more ?
2
```

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

- Allocate the memory for the new node. Use the following statements for this.

1. ptr = (struct node *)malloc(sizeof(struct node));

- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.

```
1. temp=head;
2.     for(i=0;i<loc;i++)
3.    {
4.        temp = temp->next;
5.        if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentioned lo
   cation
6.        {
7.            return;
8.        }
9.    }
```

- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

1. ptr → next = temp → next;

make the **prev** of the new node ptr point to temp.

1. ptr → prev = temp;

   make the **next** pointer of temp point to the new node ptr.

1. temp → next = ptr;

   make the **previous** pointer of the next node of temp point to the new node.

1. temp → next → prev = ptr;

Algorithm

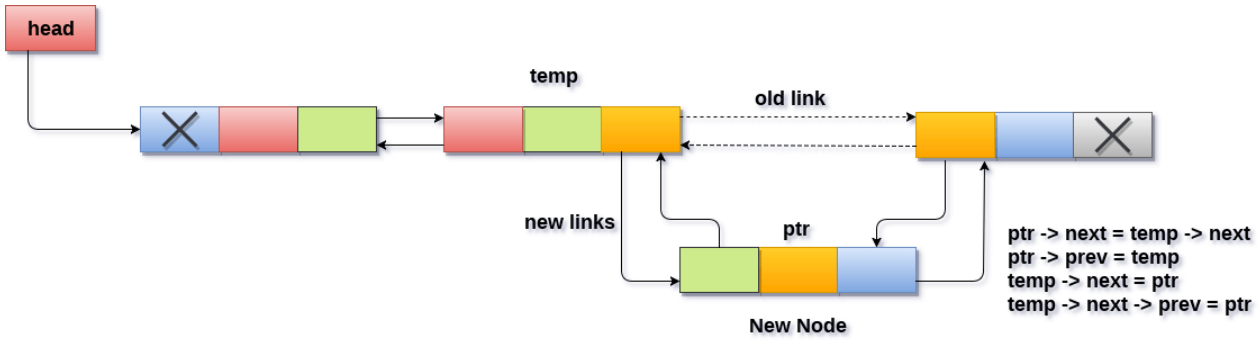
- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 15
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = START
- **Step 6:** SET I = 0
- **Step 7:** REPEAT 8 to 10 until I<="" li="">
- **Step 8:** SET TEMP = TEMP -> NEXT
- **STEP 9:** IF TEMP = NULL
- **STEP 10:** WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

  GOTO STEP 15
  [END OF IF]
  [END OF LOOP]

- **Step 11:** SET NEW_NODE -> NEXT = TEMP -> NEXT
- **Step 12:** SET NEW_NODE -> PREV = TEMP
- **Step 13 :** SET TEMP -> NEXT = NEW_NODE
- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** EXIT



Insertion into doubly linked list after specified node

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void insert_specified(int);
4. void create(int);
5. struct node
6. {
7.     int data;
8.     struct node *next;
9.     struct node *prev;
10. };
11. struct node *head;
12. void main ()
13. {
14.     int choice,item,loc;
15.     do
16.     {
17.         printf("\nEnter the item which you want to insert?\n");
18.         scanf("%d",&item);
19.         if(head == NULL)
20.         {
21.             create(item);
22.         }
23.         else
24.         {
25.             insert_specified(item);
26.         }
27.         printf("\nPress 0 to insert more ?\n");
28.         scanf("%d",&choice);
```

```
29.     }while(choice == 0);
30. }
31. void create(int item)
32.     {
33.    struct node *ptr = (struct node *)malloc(sizeof(struct node));
34.    if(ptr == NULL)
35.    {
36.        printf("\nOVERFLOW");
37.    }
38.    else
39.    {
40.
41.
42.    if(head==NULL)
43.    {
44.        ptr->next = NULL;
45.        ptr->prev=NULL;
46.        ptr->data=item;
47.        head=ptr;
48.    }
49.    else
50.    {
51.        ptr->data=item;printf("\nPress 0 to insert more ?\n");
52.        ptr->prev=NULL;
53.        ptr->next = head;
54.        head->prev=ptr;
55.        head=ptr;
56.    }
57.     printf("\nNode Inserted\n");
58. }
59.
60. }
61. void insert_specified(int item)
62. {
63.
64.    struct node *ptr = (struct node *)malloc(sizeof(struct node));
65.    struct node *temp;
66.    int i, loc;
67.    if(ptr == NULL)
68.    {
```

```
69.       printf("\n OVERFLOW");
70.   }
71.   else
72.   {
73.       printf("\nEnter the location\n");
74.       scanf("%d",&loc);
75.       temp=head;
76.       for(i=0;i<loc;i++)
77.       {
78.           temp = temp->next;
79.           if(temp == NULL)
80.           {
81.               printf("\ncan't insert\n");
82.               return;
83.           }
84.       }
85.       ptr->data = item;
86.       ptr->next = temp->next;
87.       ptr -> prev = temp;
88.       temp->next = ptr;
89.       temp->next->prev=ptr;
90.       printf("Node Inserted\n");
91.   }
92. }
```

Output

```
Enter the item which you want to insert?
12

Node Inserted

Press 0 to insert more ?
0

Enter the item which you want to insert?
90

Node Inserted
```

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

1. Ptr = head;
2. head = head → next;

now make the prev of this new head node point to NULL. This will be done by using the following statements.

1. head → prev = NULL

Now free the pointer ptr by using the **free** function.

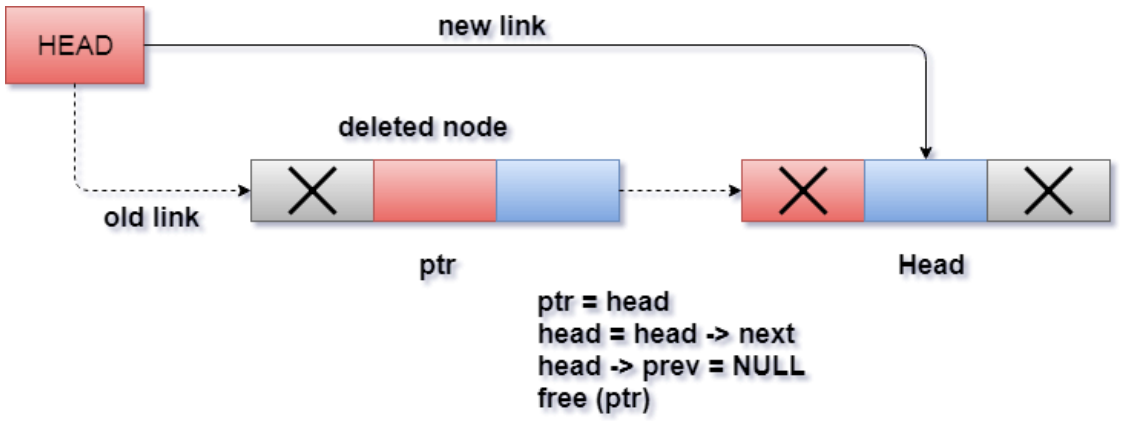
1. free(ptr)

Algorithm

- **STEP 1:** IF HEAD = NULL

  WRITE UNDERFLOW
  GOTO STEP 6

- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

C Function

```c
#include<stdio.h>
#include<stdlib.h>
void create(int);
void beginning_delete();
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
struct node *head;
void main ()
{
    int choice,item;
    do
    {
        printf("1.Append List\n2.Delete node from beginning\n3.Exit\n4.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
            case 2:
            beginning_delete();
            break;
            case 3:
            exit(0);
            break;
            default:
            printf("\nPlease enter valid choice\n");
        }

    }while(choice != 3);
}
void create(int item)
{

```

```
41.    struct node *ptr = (struct node *)malloc(sizeof(struct node));
42.    if(ptr == NULL)
43.    {
44.        printf("\nOVERFLOW\n");
45.    }
46.    else
47.    {
48.
49.
50.    if(head==NULL)
51.    {
52.        ptr->next = NULL;
53.        ptr->prev=NULL;
54.        ptr->data=item;
55.        head=ptr;
56.    }
57.    else
58.    {
59.        ptr->data=item;printf("\nPress 0 to insert more ?\n");
60.        ptr->prev=NULL;
61.        ptr->next = head;
62.        head->prev=ptr;
63.        head=ptr;
64.    }
65.     printf("\nNode Inserted\n");
66. }
67.
68. }
69. void beginning_delete()
70. {
71.     struct node *ptr;
72.     if(head == NULL)
73.     {
74.         printf("\n UNDERFLOW\n");
75.     }
76.     else if(head->next == NULL)
77.     {
78.         head = NULL;
79.         free(head);
80.         printf("\nNode Deleted\n");
```

```
81.    }
82.    else
83.    {
84.        ptr = head;
85.        head = head -> next;
86.        head -> prev = NULL;
87.        free(ptr);
88.        printf("\nNode Deleted\n");
89.    }
90. }
```

Output

```
1.Append List
2.Delete node from beginning
3.Exit
4.Enter your choice?1

Enter the item
12

Node Inserted
1.Append List
2.Delete node from beginning
3.Exit
4.Enter your choice?2

Node Deleted
1.Append List
2.Delete node from beginning
3.Exit
4.Enter your choice?
```

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.

- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
1. ptr = head;
2.       if(ptr->next != NULL)
3.       {
4.          ptr = ptr -> next;
5.       }
```

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

```
1. ptr → prev → next = NULL
```

free the pointer as this the node which is to be deleted.

```
1. free(ptr)
```

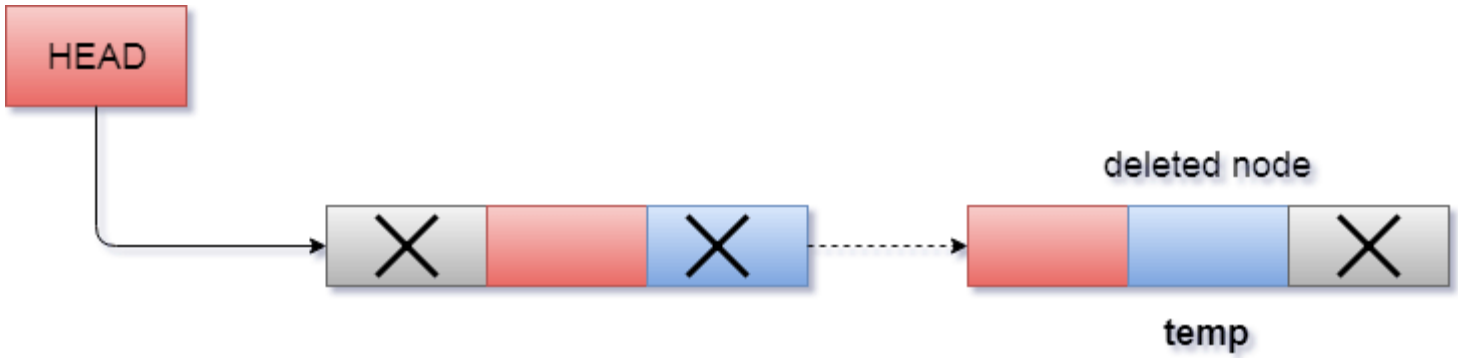
- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 7
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT

  [END OF LOOP]

- **Step 5:** SET TEMP ->PREV-> NEXT = NULL
- **Step 6:** FREE TEMP
- **Step 7:** EXIT



temp->prev->next = NULL
free(temp)

Deletion in doubly linked list at the end

C Function

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  void create(int);
4.  void last_delete();
5.  struct node
6.  {
7.      int data;
8.      struct node *next;
9.      struct node *prev;
10. };
11. struct node *head;
12. void main ()
13. {
14.     int choice,item;
15.     do
16.     {
17.         printf("1.Append List\n2.Delete node from end\n3.Exit\n4.Enter your choice?");
18.         scanf("%d",&choice);
19.         switch(choice)
20.         {
21.             case 1:
22.             printf("\nEnter the item\n");
```

```
23.             scanf("%d",&item);
24.             create(item);
25.             break;
26.             case 2:
27.             last_delete();
28.             break;
29.             case 3:
30.             exit(0);
31.             break;
32.             default:
33.             printf("\nPlease enter valid choice\n");
34.         }
35.
36.      }while(choice != 3);
37. }
38. void create(int item)
39. {
40.
41.    struct node *ptr = (struct node *)malloc(sizeof(struct node));
42.    if(ptr == NULL)
43.    {
44.        printf("\nOVERFLOW\n");
45.    }
46.    else
47.    {
48.
49.
50.    if(head==NULL)
51.    {
52.        ptr->next = NULL;
53.        ptr->prev=NULL;
54.        ptr->data=item;
55.        head=ptr;
56.    }
57.    else
58.    {
59.        ptr->data=item;
60.        ptr->prev=NULL;
61.        ptr->next = head;
62.        head->prev=ptr;
```

```
63.       head=ptr;
64.   }
65.    printf("\nNode Inserted\n");
66. }
67.
68. }
69. void last_delete()
70. {
71.    struct node *ptr;
72.    if(head == NULL)
73.    {
74.        printf("\n UNDERFLOW\n");
75.    }
76.    else if(head->next == NULL)
77.    {
78.        head = NULL;
79.        free(head);
80.        printf("\nNode Deleted\n");
81.    }
82.    else
83.    {
84.        ptr = head;
85.        if(ptr->next != NULL)
86.        {
87.            ptr = ptr -> next;
88.        }
89.        ptr -> prev -> next = NULL;
90.        free(ptr);
91.        printf("\nNode Deleted\n");
92.    }
93. }
```

Output

```
1.Append List
2.Delete node from end
3.Exit
4.Enter your choice?1

Enter the item
12

Node Inserted
```

```
1.Append List
2.Delete node from end
3.Exit
4.Enter your choice?1

Enter the item
90

Node Inserted
1.Append List
2.Delete node from end
3.Exit
4.Enter your choice?2

Node Deleted
```

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.

```
1. temp = head
```

- Traverse the list until we find the desired data value.

```
1. while(temp -> data != val)
2. temp = temp -> next;
```

- Check if this is the last node of the list. If it is so then we can't perform deletion.

```
1. if(temp -> next == NULL)
2.     {
3. return;
4.     }
```

- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
1. if(temp -> next -> next == NULL)
2.     {
```

```
3.          temp ->next = NULL;
4.      }
```

- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
1.  ptr = temp -> next;
2.          temp -> next = ptr -> next;
3.          ptr -> next -> prev = temp;
4.          free(ptr);
```

Algorithm

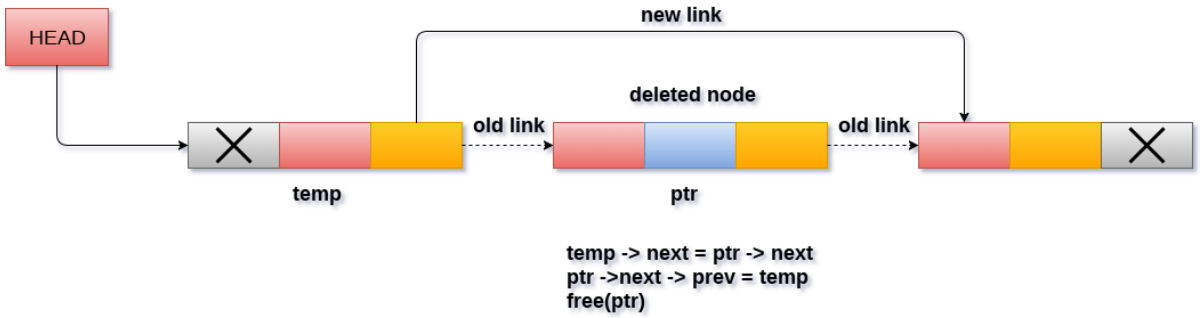
- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 9
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** Repeat Step 4 while TEMP -> DATA != ITEM
- **Step 4:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 5:** SET PTR = TEMP -> NEXT
- **Step 6:** SET TEMP -> NEXT = PTR -> NEXT
- **Step 7:** SET PTR -> NEXT -> PREV = TEMP
- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void delete_specified();
5. struct node
6. {
7.     int data;
8.     struct node *next;
9.     struct node *prev;
10. };
11. struct node *head;
12. void main ()
13. {
14.     int choice,item;
15.     do
16.     {
17.         printf("1.Append List\n2.Delete node\n3.Exit\n4.Enter your choice?");
18.         scanf("%d",&choice);
19.         switch(choice)
20.         {
21.             case 1:
22.             printf("\nEnter the item\n");
23.             scanf("%d",&item);
24.             create(item);
25.             break;
26.             case 2:
27.             delete_specified();
28.             break;
29.             case 3:
30.             exit(0);
31.             break;
32.             default:
33.             printf("\nPlease enter valid choice\n");
34.         }
35.
36.     }while(choice != 3);
37. }
38. void create(int item)
```

```
39. {
40.    struct node *ptr = (struct node *)malloc(sizeof(struct node));
41.    if(ptr == NULL)
42.    {
43.        printf("\nOVERFLOW\n");
44.    }
45.    else
46.    {
47.
48.
49.    if(head==NULL)
50.    {
51.        ptr->next = NULL;
52.        ptr->prev=NULL;
53.        ptr->data=item;
54.        head=ptr;
55.    }
56.    else
57.    {
58.        ptr->data=item;
59.        ptr->prev=NULL;
60.        ptr->next = head;
61.        head->prev=ptr;
62.        head=ptr;
63.    }
64.     printf("\nNode Inserted\n");
65. }
66.
67. }
68. void delete_specified( )
69. {
70.     struct node *ptr, *temp;
71.     int val;
72.     printf("Enter the value");
73.     scanf("%d",&val);
74.     temp = head;
75.     while(temp -> data != val)
76.     temp = temp -> next;
77.     if(temp -> next == NULL)
78.     {
```

```
79.         printf("\nCan't delete\n");
80.     }
81.     else if(temp -> next -> next == NULL)
82.     {
83.         temp ->next = NULL;
84.         printf("\nNode Deleted\n");
85.     }
86.     else
87.     {
88.         ptr = temp -> next;
89.         temp -> next = ptr -> next;
90.         ptr -> next -> prev = temp;
91.         free(ptr);
92.         printf("\nNode Deleted\n");
93.     }
94. }
```

Output

```
1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
12

Node Inserted
1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
23

Node Inserted
1.Append List
2.Delete node
3.Exit
4.Enter your choice?1

Enter the item
34

Node Inserted
1.Append List
2.Delete node
3.Exit
4.Enter your choice?2
Enter the value23

Node Deleted
```

Searching for a specific node in Doubly Linked List

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.

1. ptr = head

- declare a local variable I and assign it to 0.

1. i=0

- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm

- **Step 1:** IF HEAD == NULL

  WRITE "UNDERFLOW"
  GOTO STEP 8
  [END OF IF]

- **Step 2:** Set PTR = HEAD
- **Step 3:** Set i = 0
- **Step 4:** Repeat step 5 to 7 while PTR != NULL
- **Step 5:** IF PTR → data = item

return i
[END OF IF]

- **Step 6:** i = i + 1
- **Step 7:** PTR = PTR → next
- **Step 8:** Exit

C Function

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
struct node *head;
void main ()
{
    int choice,item,loc;
    do
    {
        printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
            case 2:
            search();
            case 3:
            exit(0);
            break;
            default:
```

```
32.             printf("\nPlease enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38. {
39.
40.     struct node *ptr = (struct node *)malloc(sizeof(struct node));
41.     if(ptr == NULL)
42.     {
43.         printf("\nOVERFLOW");
44.     }
45.     else
46.     {
47.
48.
49.     if(head==NULL)
50.     {
51.         ptr->next = NULL;
52.         ptr->prev=NULL;
53.         ptr->data=item;
54.         head=ptr;
55.     }
56.     else
57.     {
58.         ptr->data=item;printf("\nPress 0 to insert more ?\n");
59.         ptr->prev=NULL;
60.         ptr->next = head;
61.         head->prev=ptr;
62.         head=ptr;
63.     }
64.      printf("\nNode Inserted\n");
65. }
66.
67. }
68. void search()
69. {
70.      struct node *ptr;
71.      int item,i=0,flag;
```

```
72.    ptr = head;
73.    if(ptr == NULL)
74.    {
75.        printf("\nEmpty List\n");
76.    }
77.    else
78.    {
79.        printf("\nEnter item which you want to search?\n");
80.        scanf("%d",&item);
81.        while (ptr!=NULL)
82.        {
83.            if(ptr->data == item)
84.            {
85.                printf("\nitem found at location %d ",i+1);
86.                flag=0;
87.                break;
88.            }
89.            else
90.            {
91.                flag=1;
92.            }
93.            i++;
94.            ptr = ptr -> next;
95.        }
96.        if(flag==1)
97.        {
98.            printf("\nItem not found\n");
99.        }
100.           }
101.
102.       }
```

Output

```
1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
23

Node Inserted

1.Create
```

```
2.Search
3.Exit
4.Enter your choice?1

Enter the item
90

Node Inserted

1.Create
2.Search
3.Exit
4.Enter your choice?2

Enter item which you want to search?
90

item found at location 1
```

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

1. Ptr = head

then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

```
1. while(ptr != NULL)
2.         {
3.             printf("%d\n",ptr->data);
4.             ptr=ptr->next;
5.         }
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

- **Step 1:** IF HEAD == NULL

  WRITE "UNDERFLOW"
  GOTO STEP 6
  [END OF IF]

- **Step 2:** Set PTR = HEAD
- **Step 3:** Repeat step 4 and 5 while PTR != NULL

- **Step 4:** Write PTR → data
- **Step 5:** PTR = PTR → next
- **Step 6:** Exit

C Function

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
int traverse();
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
struct node *head;
void main ()
{
    int choice,item;
    do
    {
        printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
            case 2:
            traverse();
            break;
            case 3:
            exit(0);
            break;
            default:
            printf("\nPlease enter valid choice\n");
        }
```

```
35.
36.     }while(choice != 3);
37. }
38. void create(int item)
39. {
40.
41.    struct node *ptr = (struct node *)malloc(sizeof(struct node));
42.    if(ptr == NULL)
43.    {
44.        printf("\nOVERFLOW\n");
45.    }
46.    else
47.    {
48.
49.
50.    if(head==NULL)
51.    {
52.        ptr->next = NULL;
53.        ptr->prev=NULL;
54.        ptr->data=item;
55.        head=ptr;
56.    }
57.    else
58.    {
59.        ptr->data=item;printf("\nPress 0 to insert more ?\n");
60.        ptr->prev=NULL;
61.        ptr->next = head;
62.        head->prev=ptr;
63.        head=ptr;
64.    }
65.     printf("\nNode Inserted\n");
66. }
67.
68. }
69. int traverse()
70. {
71.     struct node *ptr;
72.     if(head == NULL)
73.     {
74.         printf("\nEmpty List\n");
```

```
75.    }
76.    else
77.    {
78.        ptr = head;
79.        while(ptr != NULL)
80.        {
81.            printf("%d\n",ptr->data);
82.            ptr=ptr->next;
83.        }
84.    }
85. }
```

Output

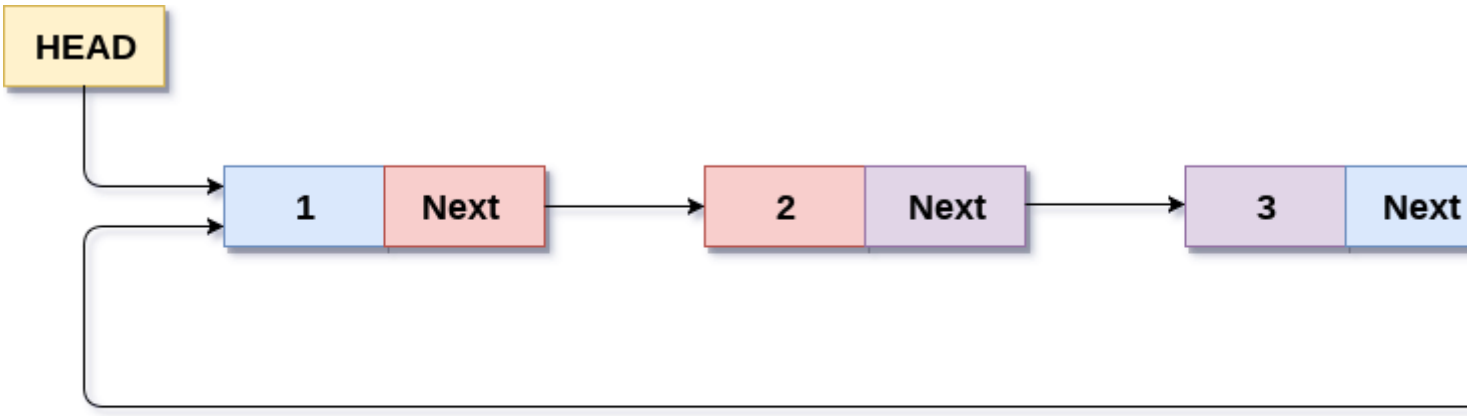
```
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
23

Node Inserted
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
23

Press 0 to insert more ?

Node Inserted
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
90

Press 0 to insert more ?

Node Inserted
1.Append List
2.Traverse
3.Exit
4.Enter your choice?2
90
23
23
```

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.



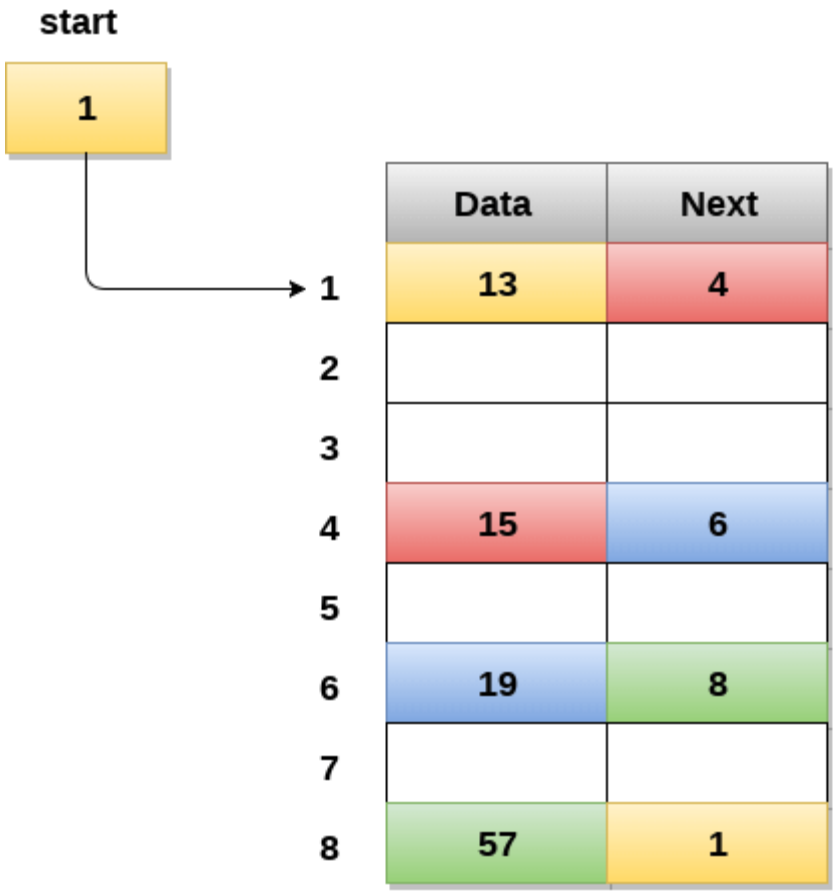
Circular Singly Linked List

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.

Insertion into circular singly linked list at beginning

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Firstly, allocate the memory space for the new node by using the malloc method of C language.

1. struct node *ptr = (struct node *)malloc(sizeof(struct node));

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just

inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
1. if(head == NULL)
2.         {
3.             head = ptr;
4.             ptr -> next = head;
5.         }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
1. temp = head;
2.             while(temp->next != head)
3.                 temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr.

This will be done by using the following statements.

```
1. temp -> next = ptr;
```

the next pointer of temp will point to the existing head node of the list.

```
1. ptr->next = head;
```

Now, make the new node ptr, the new head node of the circular singly linked list.

```
1. head = ptr;
```

in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

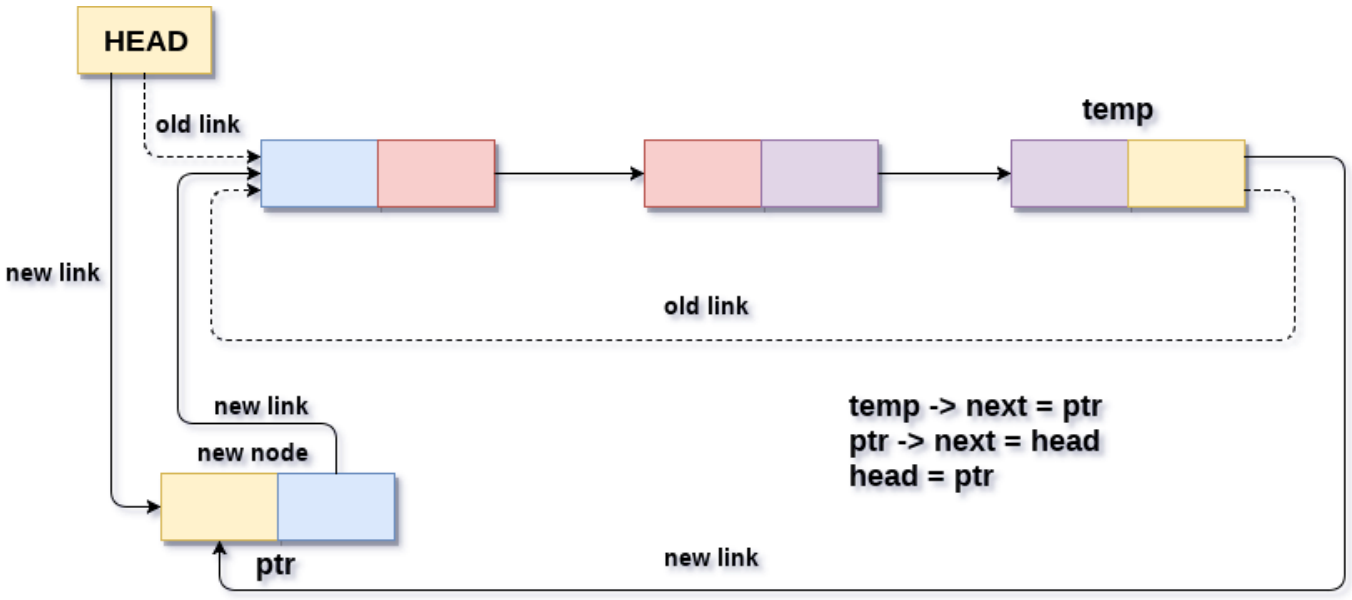
- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 11
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = HEAD
- **Step 6:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 7:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 8:** SET NEW_NODE -> NEXT = HEAD
- **Step 9:** SET TEMP → NEXT = NEW_NODE
- **Step 10:** SET HEAD = NEW_NODE
- **Step 11:** EXIT



Insertion into circular singly linked list at beginning

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void beg_insert(int);
4. struct node
5. {
6.     int data;
```

```
7.      struct node *next;
8.  };
9.  struct node *head;
10. void main ()
11. {
12.     int choice,item;
13.     do
14.     {
15.         printf("\nEnter the item which you want to insert?\n");
16.         scanf("%d",&item);
17.         beg_insert(item);
18.         printf("\nPress 0 to insert more ?\n");
19.         scanf("%d",&choice);
20.     }while(choice == 0);
21. }
22. void beg_insert(int item)
23. {
24.
25.     struct node *ptr = (struct node *)malloc(sizeof(struct node));
26.     struct node *temp;
27.     if(ptr == NULL)
28.     {
29.         printf("\nOVERFLOW");
30.     }
31.     else
32.     {
33.         ptr -> data = item;
34.         if(head == NULL)
35.         {
36.             head = ptr;
37.             ptr -> next = head;
38.         }
39.         else
40.         {
41.             temp = head;
42.             while(temp->next != head)
43.                 temp = temp->next;
44.             ptr->next = head;
45.             temp -> next = ptr;
46.             head = ptr;
```

```
47.         }
48.     printf("\nNode Inserted\n");
49.     }
50.
51. }
52.
```

Output

```
        Enter the item which you want to insert?
12

Node Inserted

Press 0 to insert more ?
0

Enter the item which you want to insert?
90

Node Inserted

Press 0 to insert more ?
2
```

Insertion into circular singly linked list at the end

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

- Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
1. struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
1. if(head == NULL)
2.         {
3.             head = ptr;
4.             ptr -> next = head;
5.         }
```

In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

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

At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore the existing last node i.e. **temp** must point to the new node **ptr**. This is done by using the following statement.

```
temp -> next = ptr;
```

The new last node of the list i.e. ptr will point to the head node of the list.

```
ptr -> next = head;
```

In this way, a new node will be inserted in a circular singly linked list at the beginning.

Algorithm

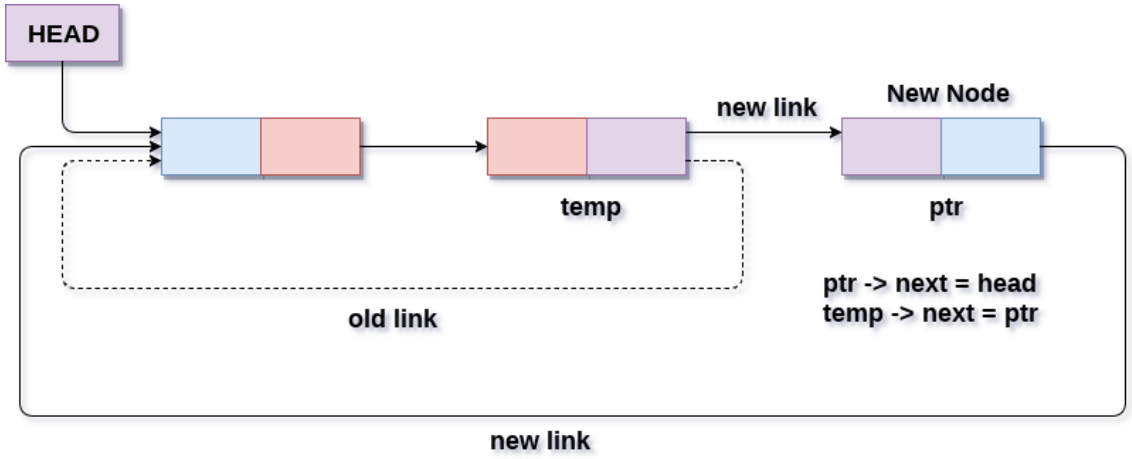
- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 1
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = HEAD
- **Step 6:** SET TEMP = HEAD
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 8:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10:** EXIT



Insertion into circular singly linked list at end

C Function

```
1.  void lastinsert(struct node*ptr, struct node *temp, int item)
2.  {
3.      ptr = (struct node *)malloc(sizeof(struct node));
4.      if(ptr == NULL)
5.      {
6.          printf("\nOVERFLOW\n");
7.      }
8.      else
9.      {
10.         ptr->data = item;
11.         if(head == NULL)
12.         {
13.             head = ptr;
14.             ptr -> next = head;
15.         }
16.         else
17.         {
18.             temp = head;
19.             while(temp -> next != head)
20.             {
21.                 temp = temp -> next;
22.             }
23.             temp -> next = ptr;
24.             ptr -> next = head;
```

```
25.         }
26.     }
27.
28. }
```

Insertion into circular singly linked list at the end

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

- Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
1. struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
1. if(head == NULL)
2.         {
3.             head = ptr;
4.             ptr -> next = head;
5.         }
```

In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
1. temp = head;
2.             while(temp->next != head)
3.                 temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore the existing last node i.e. **temp** must point to the new node **ptr**. This is done by using the following statement.

1. temp -> next = ptr;

   The new last node of the list i.e. ptr will point to the head node of the list.

1. ptr -> next = head;

   In this way, a new node will be inserted in a circular singly linked list at the beginning.

Algorithm

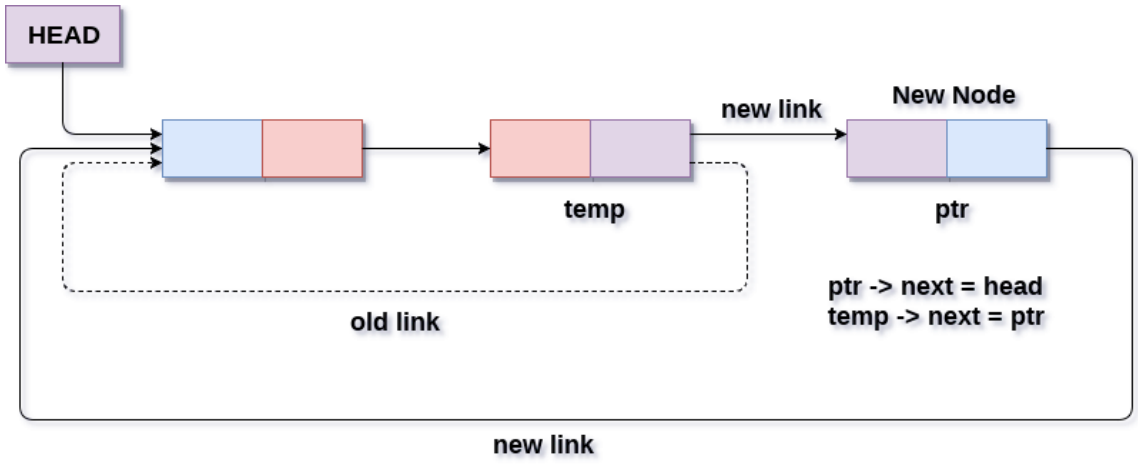
- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 1
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = HEAD
- **Step 6:** SET TEMP = HEAD
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 8:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10:** EXIT



Insertion into circular singly linked list at end

C Function

```
1.  void lastinsert(struct node*ptr, struct node *temp, int item)
2.  {
3.      ptr = (struct node *)malloc(sizeof(struct node));
4.      if(ptr == NULL)
5.      {
6.          printf("\nOVERFLOW\n");
7.      }
8.      else
9.      {
10.         ptr->data = item;
11.         if(head == NULL)
12.         {
13.             head = ptr;
14.             ptr -> next = head;
15.         }
16.         else
17.         {
18.             temp = head;
19.             while(temp -> next != head)
20.             {
21.                 temp = temp -> next;
22.             }
23.             temp -> next = ptr;
24.             ptr -> next = head;
25.         }
26.     }
27.
28. }
```

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

```
if(head == NULL)
    {
        printf("\nUNDERFLOW");
        return;
    }
```

Scenario 2(the list contains single element)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
    head = NULL;
    free(head);
}
```

Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined. The following sequence of code is used for this purpose.

```
ptr = head;
        while(ptr ->next != head)
        {
            preptr=ptr;
            ptr = ptr->next;
        }
```

now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

```
preptr->next = ptr -> next;
        free(ptr);
```

Algorithm

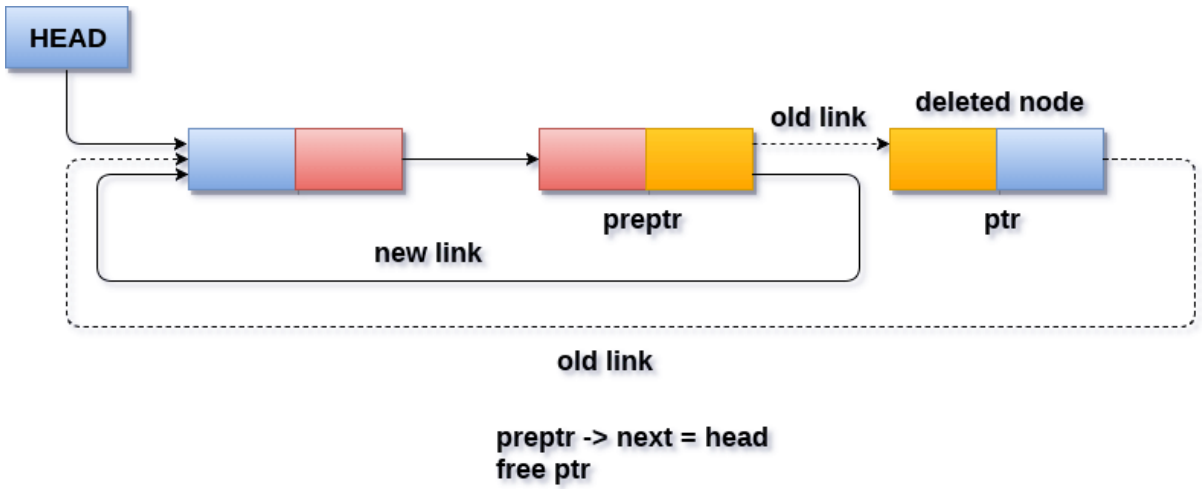
- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 8
  [END OF IF]

- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT

  [END OF LOOP]

- **Step 6:** SET PREPTR -> NEXT = HEAD
- **Step 7:** FREE PTR
- **Step 8:** EXIT



Deletion in circular singly linked list at end

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void last_delete();
5. struct node
6. {
7.     int data;
```

```
8.      struct node *next;
9.  };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("1.Append List\n2.Delete Node from end\n3.Exit\n4.Enter your choice?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
20.             case 1:
21.             printf("\nEnter the item\n");
22.             scanf("%d",&item);
23.             create(item);
24.             break;
25.             case 2:
26.             last_delete();
27.             break;
28.             case 3:
29.             exit(0);
30.             break;
31.             default:
32.             printf("\nPlease Enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38. {
39.
40.     struct node *ptr = (struct node *)malloc(sizeof(struct node));
41.     struct node *temp;
42.     if(ptr == NULL)
43.     {
44.         printf("\nOVERFLOW\n");
45.     }
46.     else
47.     {
```

```
48.         ptr -> data = item;
49.         if(head == NULL)
50.         {
51.             head = ptr;
52.             ptr -> next = head;
53.         }
54.         else
55.         {
56.             temp = head;
57.             while(temp->next != head)
58.                 temp = temp->next;
59.             ptr->next = head;
60.             temp -> next = ptr;
61.             head = ptr;
62.         }
63.     printf("\nNode Inserted\n");
64.     }
65.
66. }
67. void last_delete()
68. {
69. struct node *ptr, *preptr;
70.     if(head==NULL)
71.     {
72.         printf("\nUNDERFLOW\n");
73.     }
74.     else if (head ->next == head)
75.     {
76.         head = NULL;
77.         free(head);
78.         printf("\nNode Deleted\n");
79.     }
80.     else
81.     {
82.         ptr = head;
83.         while(ptr ->next != head)
84.         {
85.             preptr=ptr;
86.             ptr = ptr->next;
87.         }
```

```
88.         preptr->next = ptr -> next;
89.         free(ptr);
90.         printf("\nNode Deleted\n");
91.     }
92. }
```

Output

```
1.Append List
2.Delete Node from end
3.Exit
4.Enter your choice?1

Enter the item
90

Node Inserted
1.Append List
2.Delete Node from end
3.Exit
4.Enter your choice?2

Node Deleted
```

Searching in circular singly linked list

Searching in circular singly linked list needs traversing across the list. The item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned otherwise -1 is returned.

The algorithm and its implementation in C is given as follows.

Algorithm

- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL

  WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF

- **STEP 4:** IF HEAD → DATA = ITEM

WRITE i+1 RETURN [END OF IF]

- **STEP 5:** REPEAT STEP 5 TO 7 UNTIL PTR->next != head
- **STEP 6:** if ptr → data = item

  write i+1
  RETURN
  End of IF

- **STEP 7:** I = I + 1
- **STEP 8:** PTR = PTR → NEXT

  [END OF LOOP]

- **STEP 9:** EXIT

C Function

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
    int data;
    struct node *next;
};
struct node *head;
void main ()
{
    int choice,item,loc;
    do
    {
        printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
```

```
25.             case 2:
26.             search();
27.             case 3:
28.             exit(0);
29.             break;
30.             default:
31.             printf("\nPlease enter valid choice\n");
32.         }
33.
34.     }while(choice != 3);
35. }
36. void create(int item)
37. {
38.     struct node *ptr = (struct node *)malloc(sizeof(struct node));
39.     struct node *temp;
40.     if(ptr == NULL)
41.     {
42.         printf("\nOVERFLOW\n");
43.     }
44.     else
45.     {
46.         ptr->data = item;
47.         if(head == NULL)
48.         {
49.             head = ptr;
50.             ptr -> next = head;
51.         }
52.         else
53.         {
54.             temp = head;
55.             while(temp -> next != head)
56.             {
57.                 temp = temp -> next;
58.             }
59.             temp -> next = ptr;
60.             ptr -> next = head;
61.         }
62.     printf("\nNode Inserted\n");
63.     }
64.
```

```
65. }
66. void search()
67. {
68.     struct node *ptr;
69.     int item,i=0,flag=1;
70.     ptr = head;
71.     if(ptr == NULL)
72.     {
73.         printf("\nEmpty List\n");
74.     }
75.     else
76.     {
77.         printf("\nEnter item which you want to search?\n");
78.         scanf("%d",&item);
79.         if(head ->data == item)
80.         {
81.         printf("item found at location %d",i+1);
82.         flag=0;
83.         return;
84.         }
85.         else
86.         {
87.         while (ptr->next != head)
88.         {
89.             if(ptr->data == item)
90.             {
91.                 printf("item found at location %d ",i+1);
92.                 flag=0;
93.                 return;
94.             }
95.             else
96.             {
97.                 flag=1;
98.             }
99.             i++;
100.                    ptr = ptr -> next;
101.                }
102.                }
103.                if(flag != 0)
104.                {
```

```
105.                 printf("Item not found\n");
106.                 return;
107.             }
108.         }
109.
110.     }
```

Output

```
1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
12

Node Inserted

1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
23

Node Inserted

1.Create
2.Search
3.Exit
4.Enter your choice?2

Enter item which you want to search?
12
item found at location 1
```

Traversing in Circular Singly linked list

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **head**. The algorithm and the c function implementing the algorithm is described as follows.

Algorithm

- **STEP 1:** SET PTR = HEAD
- **STEP 2:** IF PTR = NULL

  WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF

- **STEP 4:** REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD
- **STEP 5:** PRINT PTR → DATA
- **STEP 6:** PTR = PTR → NEXT

  [END OF LOOP]

- **STEP 7:** PRINT PTR→ DATA
- **STEP 8:** EXIT

C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void traverse();
5. struct node
6. {
7.     int data;
8.     struct node *next;
9. };
10. struct node *head;
11. void main ()
12. {
13.     int choice,item;
14.     do
15.     {
16.         printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
20.             case 1:
21.             printf("\nEnter the item\n");
```

```
22.             scanf("%d",&item);
23.             create(item);
24.             break;
25.             case 2:
26.             traverse();
27.             break;
28.             case 3:
29.             exit(0);
30.             break;
31.             default:
32.             printf("\nPlease enter valid choice\n");
33.         }
34.
35.     }while(choice != 3);
36. }
37. void create(int item)
38. {
39.
40.     struct node *ptr = (struct node *)malloc(sizeof(struct node));
41.     struct node *temp;
42.     if(ptr == NULL)
43.     {
44.         printf("\nOVERFLOW");
45.     }
46.     else
47.     {
48.         ptr -> data = item;
49.         if(head == NULL)
50.         {
51.             head = ptr;
52.             ptr -> next = head;
53.         }
54.         else
55.         {
56.             temp = head;
57.             while(temp->next != head)
58.                 temp = temp->next;
59.             ptr->next = head;
60.             temp -> next = ptr;
61.             head = ptr;
```

```
62.         }
63.     printf("\nNode Inserted\n");
64.     }
65.
66. }
67. void traverse()
68. {
69.     struct node *ptr;
70.     ptr=head;
71.     if(head == NULL)
72.     {
73.         printf("\nnothing to print");
74.     }
75.     else
76.     {
77.         printf("\n printing values ... \n");
78.
79.         while(ptr -> next != head)
80.         {
81.
82.             printf("%d\n", ptr -> data);
83.             ptr = ptr -> next;
84.         }
85.         printf("%d\n", ptr -> data);
86.     }
87.
88. }
```

Output

```
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
23

Node Inserted
1.Append List
2.Traverse
3.Exit
4.Enter your choice?2

 printing values ...
23
```

APPLICATIONS OF LINKED LISTS – POLYNOMIALS

Representation of the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geq 0$.

Present each term as a node containing coefficient and exponent fields, as well as a pointer to the next term.

Assuming that the coefficients are integers, the type declarations are:

```
typedef struct polyNode *polyPointer;
typedef struct {
                    int coef;
                    int expon;
                    polyPointer link;
                } polyNode;
polyPointer a,b;
```

coef	expon	link

Figure shows how we would store the polynomials

$$a = 3x^{14} + 2x^8 + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^6$$

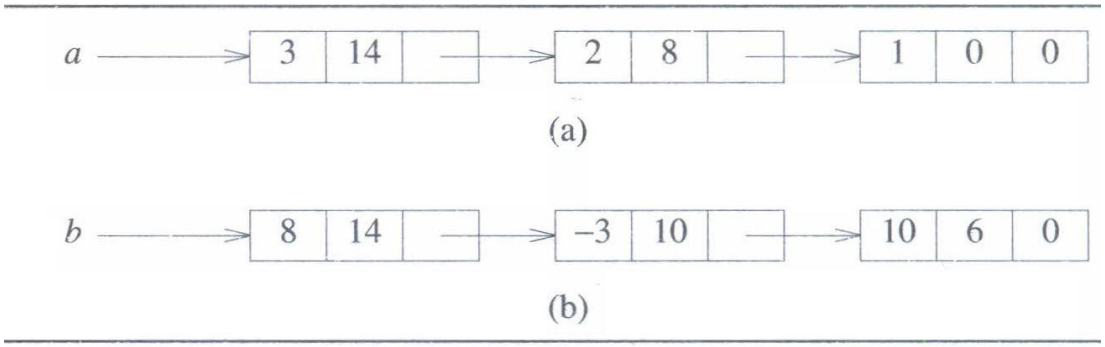


Figure: Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

Adding Polynomials

To add two polynomials, examine their terms starting at the nodes pointed to by ***a*** and ***b***.

- If the exponents of the two terms are equal, then add the two coefficients and create a new term for the result, and also move the pointers to the next nodes in ***a*** and ***b***.
- If the exponent of the current term in ***a*** is less than the exponent of the current term in ***b***, then create a duplicate term of ***b***, attach this term to the result, called ***c***, and advance the pointer to the next term in ***b***.
- If the exponent of the current term in ***b*** is less than the exponent of the current term in ***a***, then create a duplicate term of ***a***, attach this term to the result, called ***c***, and advance the pointer to the next term in ***a***

Below figure illustrates this process for the polynomials addition.

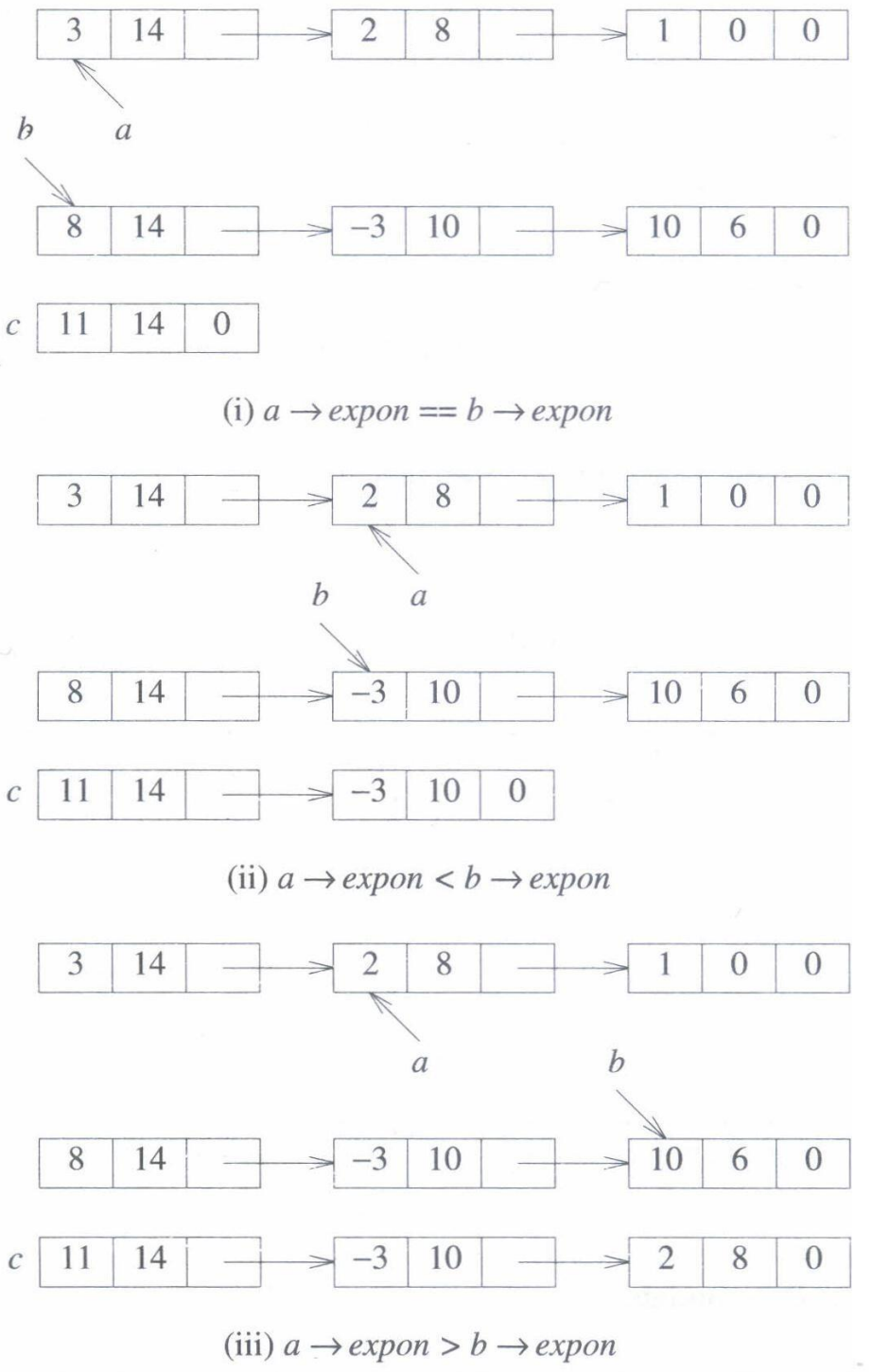


Figure: Generating the first three terms of $c = a + b$

The complete addition algorithm is specified by padd( )

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
  polyPointer c, rear, temp;
  int sum;
  MALLOC(rear, sizeof(*rear));
  c = rear;
  while (a && b)
    switch (COMPARE(a→expon,b→expon)) {
      case -1: /* a→expon < b→expon */
           attach(b→coef,b→expon,&rear);
           b = b→link;
           break;
      case 0: /* a→expon = b→expon */
           sum = a→coef + b→coef;
           if (sum) attach(sum,a→expon,&rear);
           a = a→link;  b = b→link; break;
      case 1: /* a→expon > b→expon */
           attach(a→coef,a→expon,&rear);
           a = a→link;
    }
  /* copy rest of list a and then list b */
  for (; a; a = a→link) attach(a→coef,a→expon,&rear);
  for (; b; b = b→link) attach(b→coef,b→expon,&rear);
  rear→link = NULL;
  /* delete extra initial node */
  temp = c; c = c→link;  free(temp);
  return c;
}
```

Program : Add two polynomials

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
  polyPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→coef = coefficient;
  temp→expon = exponent;
  (*ptr)→link = temp;
  *ptr = temp;
}
```

Program : Attach a node to the end of a list

Analysis of padd:

To determine the computing time of padd, first determine which operations contribute to thecost. For this algorithm, there are three cost measures:
(l) Coefficient additions
(2) Exponent comparisons
(3) Creation of new nodes for c

The maximum number of executions of any statement in padd is bounded above by m + n. Therefore, the computing time is O(m+n). This means that if we implement and run the algorithm on a computer, the time it takes will be $C_1m + C_2n + C_3$, where C1, C2, C3 are constants. Since any algorithm that adds two polynomials must look at each nonzero term at least once, padd is optimal to within a constant factor.

APPLICATIONS OF LINKED LISTS

Sparse Matrix

A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m*n matrix. Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Why do we need to use a sparse matrix instead of a simple matrix?

We can also use the simple matrix to store the elements in the memory; then why do we need to use the sparse matrix. The following are the advantages of using a sparse matrix:

- **Storage:** As we know, a sparse matrix that contains lesser non-zero elements than zero so less memory can be used to store elements. It evaluates only the non-zero elements.
- **Computing time:** In the case of searching n sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. The zeroes in the matrix are of no use to store zeroes with non-zero elements. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

Sparse Matrix Representation

The non-zero elements can be stored with triples, i.e., rows, columns, and value. The sparse matrix can be represented in the following ways:

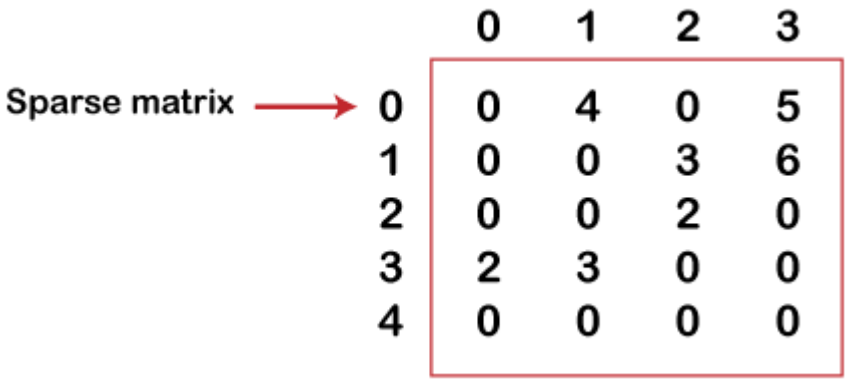
- Array representation
- Linked list representation

Array Representation

The 2d array can be used to represent a sparse matrix in which there are three rows named as:

1. Row: It is an index of a row where a non-zero element is located.
2. Column: It is an index of the column where a non-zero element is located.
3. Value: The value of the non-zero element is located at the index (row, column).

Let's understand the sparse matrix using array representation through an example.



As we can observe above, that sparse matrix is represented using triplets, i.e., row, column, and value. In the above sparse matrix, there are 13 zero elements and 7 non-zero elements. This sparse matrix occupies 5*4 = 20 memory space. If the size of the sparse matrix is increased, then the wastage of memory space will also be increased. The above sparse matrix can be represented in the tabular form shown as below:

Table Structure

Row	Column	Value
0 0 1 1 2 3	1 3 2 3 2 0	4 5 3 6 2 2
3	1	3

In the above table structure, the first column is representing the row number, the second

column is representing the column number and third column represents the non-zero value at index(row, column). The size of the table depends upon the number of non-zero elements in the sparse matrix. The above table occupies (7 * 3) = 21 but it more than the sparse matrix. Consider the case if the matrix is **8*8** and there are only 8 non-zero elements in the matrix then the space occupied by the sparse matrix would be 8*8 = 64 whereas, the space occupied by the table represented using triplets would be 8*3 = 24.

In the 0^{th} row and 1^{nd} column, 4 value is available. In the 0^{th} row and 3^{rd} column, value 5 is stored. In the 1^{st} row and 2^{nd} column, value 3 is stored. In the 1^{st} row and 3^{rd} column, value 6 is stored. In 2^{nd} row and 2^{nd} column, value 2 is stored. In the 3^{rd} row and 0^{th} column, value 2 is stored. In the 3^{rd} row and 1^{st} column, value 3 is stored.

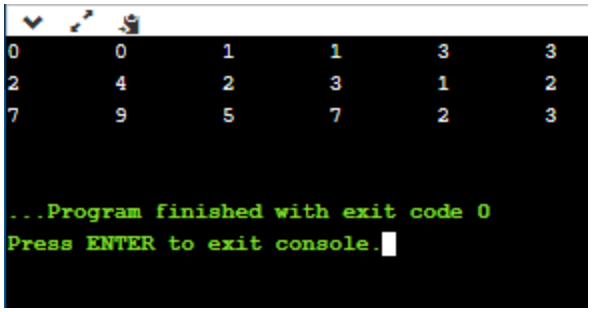
Array implementation of sparse matrix in C

```
1. #include <stdio.h>
2. int main()
3. {
```

```
4.      // Sparse matrix having size 4*5
5.      int sparse_matrix[4][5] =
6.      {
7.          {0 , 0 , 7 , 0 , 9 },
8.          {0 , 0 , 5 , 7 , 0 },
9.          {0 , 0 , 0 , 0 , 0 },
10.         {0 , 2 , 3 , 0 , 0 }
11.     };
12.    // size of matrix
13.     int size = 0;
14.     for(int i=0; i<4; i++)
15.     {
16.         for(int j=0; j<5; j++)
17.         {
18.             if(sparse_matrix[i][j]!=0)
19.             {
20.                 size++;
21.             }
22.         }
23.     }
24.    // Defining final matrix
25.     int matrix[3][size];
26.      int k=0;
27.    // Computing final matrix
28.     for(int i=0; i<4; i++)
29.     {
30.         for(int j=0; j<5; j++)
31.         {
32.             if(sparse_matrix[i][j]!=0)
33.             {
34.                 matrix[0][k] = i;
35.                 matrix[1][k] = j;
36.                 matrix[2][k] = sparse_matrix[i][j];
37.                 k++;
38.             }
39.       }
40.     }
```

```
41.   // Displaying the final matrix
42.    for(int i=0 ;i<3; i++)
43.    {
44.        for(int j=0; j<size; j++)
45.        {
46.            printf("%d ", matrix[i][j]);
47.            printf("\t");
48.        }
49.        printf("\n");
50.    }
51.    return 0;
52. }
```

Output

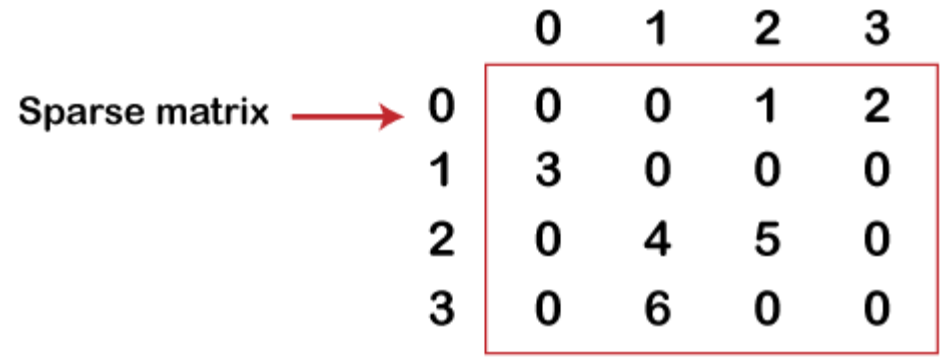


Linked List Representation

In linked list representation, linked list data structure is used to represent a sparse matrix. In linked list representation, each node consists of four fields whereas, in array representation, there are three fields, i.e., row, column, and value. The following are the fields in the linked list:

- Row: It is an index of row where a non-zero element is located.
- Column: It is an index of column where a non-zero element is located.
- Value: It is the value of the non-zero element which is located at the index (row, column).
- Next node: It stores the address of the next node.

Let's understand the sparse matrix using linked list representation through an example.

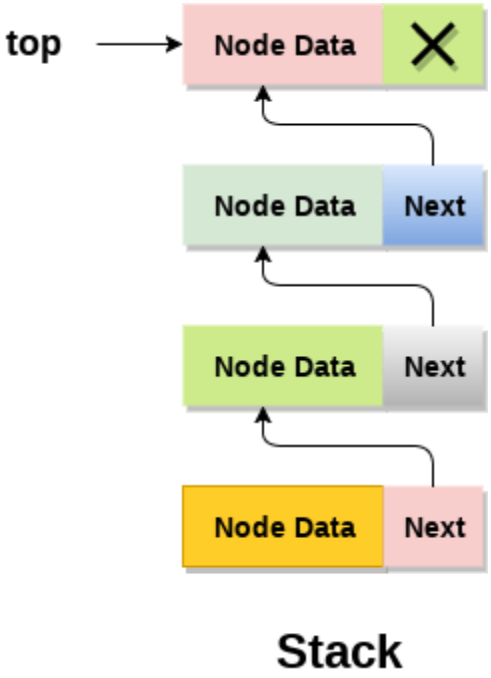


In the above figure, sparse represented in the linked list form. In the node, first field represents the index of row, second field represents the index of column, third field represents the value and fourth field contains the address of the next node.

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

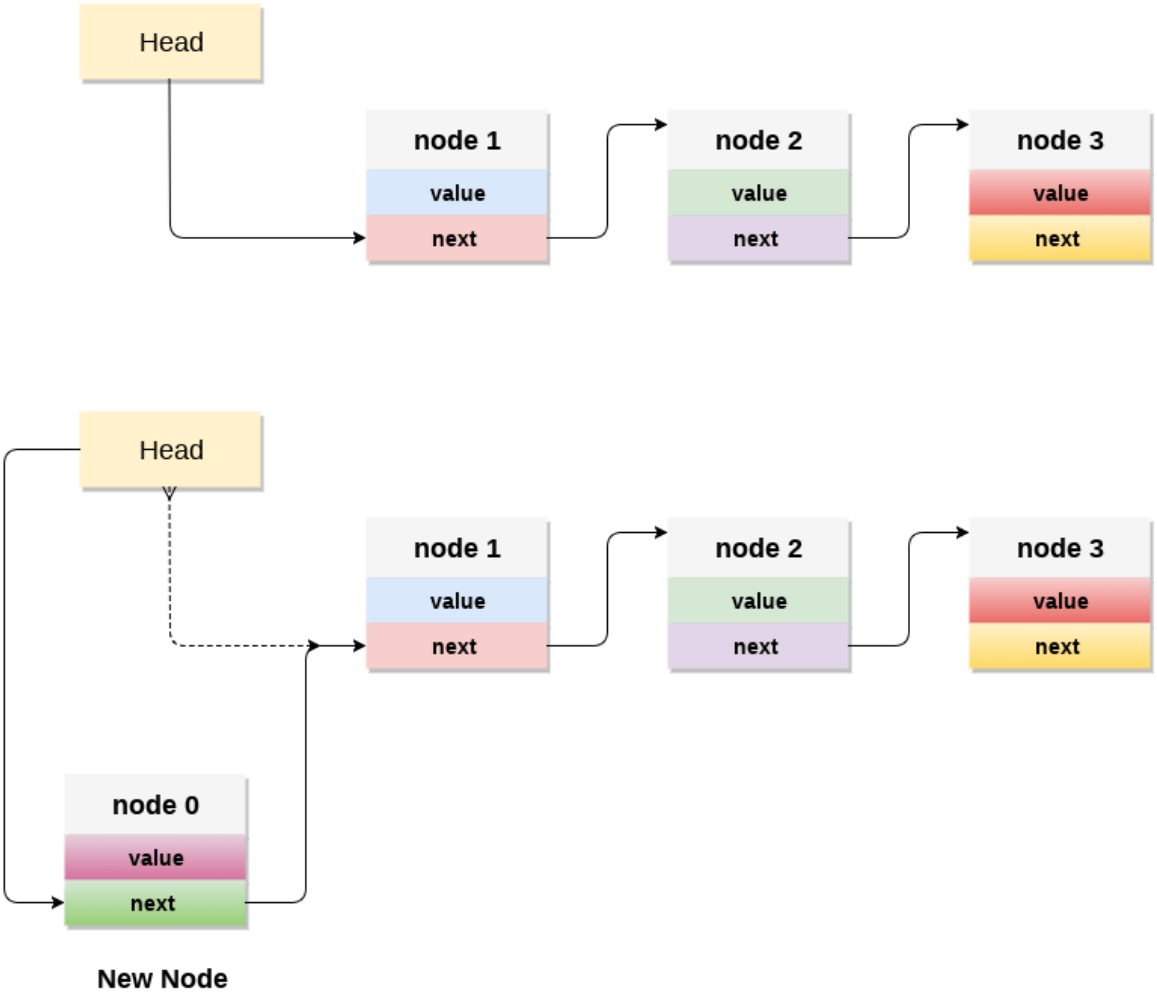
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.

2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)



C implementation :

```
1. void push ()
2. {
```

```
3.      int val;
4.      struct node *ptr =(struct node*)malloc(sizeof(struct node));
5.      if(ptr == NULL)
6.      {
7.          printf("not able to push the element");
8.      }
9.      else
10.     {
11.         printf("Enter the value");
12.         scanf("%d",&val);
13.         if(head==NULL)
14.         {
15.             ptr->val = val;
16.             ptr -> next = NULL;
17.             head=ptr;
18.         }
19.         else
20.         {
21.             ptr->val = val;
22.             ptr->next = head;
23.             head=ptr;
24.
25.         }
26.         printf("Item pushed");
27.
28.     }
29. }
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

30. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.

31. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
1.  void pop()
2.  {
3.      int item;
4.      struct node *ptr;
5.      if (head == NULL)
6.      {
7.          printf("Underflow");
8.      }
9.      else
10.     {
11.         item = head->val;
12.         ptr = head;
13.         head = head->next;
14.         free(ptr);
15.         printf("Item popped");
16.
17.     }
18. }
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

19. Copy the head pointer into a temporary pointer.
20. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

C Implementation

```
1. void display()
2. {
3.     int i;
4.     struct node *ptr;
5.     ptr=head;
6.     if(ptr == NULL)
7.     {
8.         printf("Stack is empty\n");
9.     }
10.    else
11.    {
12.        printf("Printing Stack elements \n");
13.        while(ptr!=NULL)
14.        {
15.            printf("%d\n",ptr->val);
16.            ptr = ptr->next;
17.        }
18.    }
19. }
```

Menu Driven program in C implementing all the stack operations using linked list :

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. void push();
4. void pop();
5. void display();
6. struct node
7. {
8. int val;
9. struct node *next;
10. };
11. struct node *head;
12.
13. void main ()
```

```
{
    int choice=0;
    printf("\n*********Stack operations using linked list*********\n");
    printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("\n\nChose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
    };
```

```
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
```

```
88.    {
89.        printf("Underflow");
90.    }
91.    else
92.    {
93.        item = head->val;
94.        ptr = head;
95.        head = head->next;
96.        free(ptr);
97.        printf("Item popped");
98.
99.    }
100.        }
101.        void display()
102.        {
103.            int i;
104.            struct node *ptr;
105.            ptr=head;
106.            if(ptr == NULL)
107.            {
108.                printf("Stack is empty\n");
109.            }
110.            else
111.            {
112.                printf("Printing Stack elements \n");
113.                while(ptr!=NULL)
114.                {
115.                    printf("%d\n",ptr->val);
116.                    ptr = ptr->next;
117.                }
118.            }
119.        }
```