UNIT -3

Pointers:

A variable that stores the address of another variable is called pointer

Ex : int i=3;
j=&i;

A pointer is a variable that represents the location (address) of a data item, such as a variable or an Array element.

Declaring Pointer variables

data_type *ptr_name;

Here, data_type is the data type of the value that the pointer will point to.

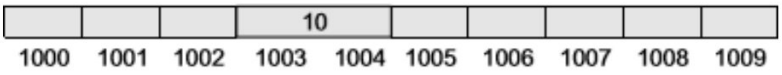
int *pnum;
char *pch;
float *pfnum;

A pointer variable is declared to point to a variable of the specified data type. Although all these pointers (pnum, pch, and pfnum) point to different data types, they will occupy the same amount of space in the memory. But how much space they will occupy will depend on the platform where the code is going to run.

Declare an integer pointer variable and start using it in our program code.

int x= 10;
int *ptr;
ptr = &x;

- In the above statement, ptr is the name of the pointer variable.
- The * informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable.
- An integer pointer variable, therefore, 'points to' an integer variable.
- In the last statement, ptr is assigned the address of x.
- The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr.



Memory representation

Since x is an integer variable, it will be allocated 2 bytes. Assuming that the compiler assigns it memory locations 1003 and 1004, the address of x (written as &x) is equal to 1003, that is the starting address of x in the memory. When we write, ptr = &x, then ptr = 1003.

```
#include <stdio.h>
int main()
{
        int num,*pnum;
        pnum = &num;
        printf("\n Enter the number : ");
        scanf("%d", &num);
        printf("\n The Address of number that was entered is : %d", pnum);
         printf("\n The number that was entered is : %d", *pnum);
        return 0;
}
```

Output

Enter the number : 10
The Address of number that was entered is :1030
The number that was entered is : 10

What will be the value of ***(&num)**? It is equivalent to simply writing **num**.

NULL POINTERS

null pointer which is a special pointer value and does not point to any value. This means that a null pointer does not point to any valid memory address. To declare a null pointer, you may use the predefined constant **NULL** which is defined in several standard header files including <stdio.h>, <stdlib.h>, and <string.h>.

int *ptr = NULL;

You can always check whether a given pointer variable stores the address of some variable or contains NULL by writing

```
if (ptr == NULL)
{
        Statement block;
}
```

You may also initialize a pointer as a null pointer by using the constant 0

```
int *ptr,
ptr = 0;
```

NULL is a preprocessor macro, which typically has the value or replacement text **0**. However, to avoid ambiguity, it is always better to use NULL to declare a null pointer

GENERIC POINTERS

A generic pointer is a pointer variable that has void as its data type. The void pointer, or the generic pointer, is a special type of pointer that can point to variables of any data type. It is declared like a normal pointer variable but using the void keyword as the pointer's data type.

For example,

```
void *ptr;
```

```
#include <stdio.h>
int main()
{
        int x=10;
        char ch = 'A';
        void *gp;
        gp = &x;
        printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
        gp = &ch;
        printf("\n Generic pointer now points to the character= %c", *(char*)gp);
        return 0;
}
```

Output

Generic pointer points to the integer value = 10
Generic pointer now points to the character = A

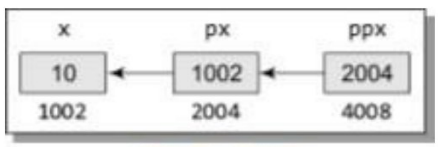
It is always recommended to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking.

Pointer to Pointers

Use pointers that point to pointers. The pointers in turn point to data or even to other pointers. To declare pointers to pointers, just add an asterisk * for each level of reference.

For example, consider the following code:

```
int x=10;
int *px, **ppx;
px = &x;
ppx = &px;
```



Pointer to pointer

printf("\n %d", **ppx);
Then, it would print 10, the value of x.

ARRAYS OF POINTERS

An array of pointers can be declared as
int *ptr[10];
The above statement declares an array of 10 pointers where each of the pointer points to an integer variable.
For example, look at the code given below.

```
int *ptr[10];
int p = 1, q = 2, r = 3, s = 4, t = 5;
ptr[0] = &p;
ptr[1] = &q;
ptr[2] = &r;
ptr[3] = &s;
ptr[4] = &t;
```

Output of the following statement
printf("\n %d", *ptr[3]);
The output will be 4 because ptr[3] stores the address of integer variable s and *ptr[3] will therefore print the value of s that is 4.
The address of three individual arrays in the array of pointers:

```
int main()
{
        int arr1[]={1,2,3,4,5};
        int arr2[]={0,2,4,6,8};
        int arr3[]={1,3,5,7,9};
        int *parr[3] = {arr1, arr2, arr3};
        int i; for(i = 0;i<3;i++)
                printf(«%d», *parr[i]);
        return 0;
}
```

Output
1 0 1
.In the for loop, parr[0] stores the base address of arr1 (or, &arr1[0]). So writing *parr[0] will print the value stored at &arr1[0]. Same is the case with *parr[1] and *parr[2].

APPLICATIONS

1Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs.
2Pointers are used to pass arrays and strings as function arguments.
3 Pointers provide an alternate way to access the individual elements of an array.
4Pointers enable the programmers to return multiple data items from a function via function arguments

Linked List representation
(NODE REPRESENTATION OF LINKED LIST):

A linked list, is a linear collection of data elements. These data elements are called **nodes**. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

Fig: Simple Linked List

Linked list in which every node contains two parts, an **Data** and a **Poin**ter to the next-node. The left part of the node which contains data may include a simple data type, an array, or a structure.The right part of the node contains a pointer to the next node (or address of the next node in sequence).

The last node will have no next node connected to it, so it will store a special value called NULL.In the above fig the NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list. Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a **self-referential data type.**

Linked lists contain a pointer variable **START** that stores the address of the first node in the list.We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

Implement a linked list using the following code:

Node representation:

```
struct node
{
        int data;
        struct node *link;
};
    Struct node * root;
```

Root= (struct node *) malloc(Size of (Struct node))

SINGLY LINKED LISTS:

A singly linked list is the simplest type of linked list in which every node contains some **data and a pointer** to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list **allows traversal of data only in one way.**

Single Linked List

a. Traversing a Single Linked List

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.

A linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node.

For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

Algorithm:

- **Step 1 -** Check whether list is **Empty** (**root** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **root**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **NULL**(**temp → data-> NULL**).

PROGRAM:

```
If(root==NULL)
{
Printf("List is empty");
}
Else
{
   temp = root;
   while(temp!=NULL)
   {
    Printf("%d", temp  -> data);
    temp = temp -> link;
    }
}
```

b. Searching for a value in a Linked List

Searching a linked list means to find a particular element in the linked list. A linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

Algorithm

```
Step 1: [INITIALIZE] SET temp = root
Step 2: Repeat Step3 while temp != NULL
Step 3:                 if VAL=Temp-->DATA
                                Print "element is found"
                                Go To Step 4
                        ELSE
                        SET temp=temp → link
                           [END OF IF]
        [END OF LOOP]
   Step 4: EXIT
```

program:

```
Void search(char val  ,struct node *root)
{
Struct Node * temp=root;
While(temp!=NULL)
 {
   If(temp->data==val)
```

```
    {
    Printf ("element is found");
    Break;
    }
Temp=Temp->link;
}
```

A. Inserting a new node in a Linked List

We can a new node is added into an already existing linked list. We will take 3 cases

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: Inserting a Node at given position in a Linked List
Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

1. Inserting a Node at the Beginning of a Linked List:

Consider the linked list shown in Fig. suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.

Algorithm to insert a new node at the beginning

We can use the following steps to insert a new node at beginning of the single linked list...

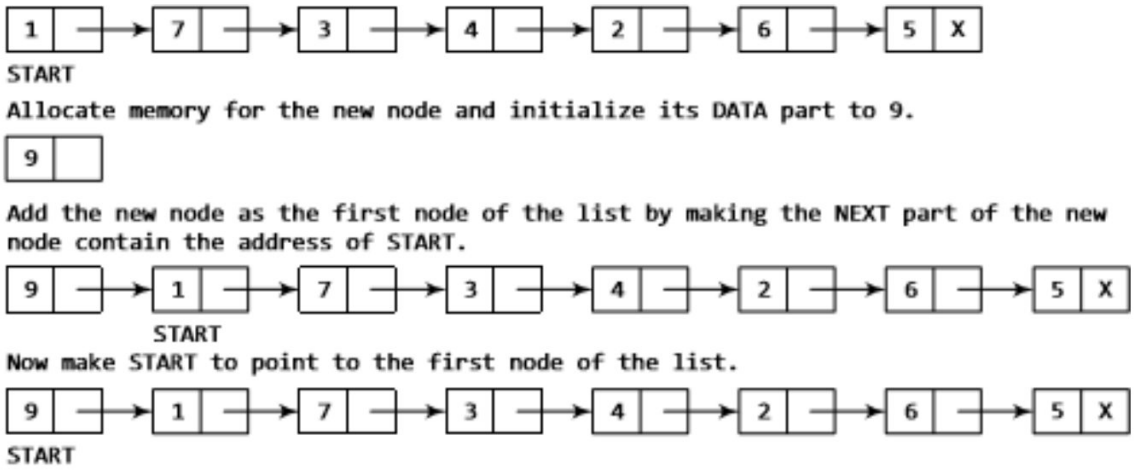


Figure Inserting an element at the beginning of a linked list

- **Step 1 -** Create a **NewNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**Root**== **NULL**)
- **Step 3 -** If it is **Empty** then, set **NewNode→LINK**= **NULL** and **NewNode→data** = **value**.
- **Step 4 - S**et **Root** = **newNode**

2.Inserting a Node at the End of a Linked List

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **Link** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**Root** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **root** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **root**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **link** is equal to **NULL**).
- **Step 6 -** Set **temp** → **link** = **newNode**.

3. Inserting a Node at given position in a Linked List

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**root** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode** →**link** = **NULL** and **root** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **root**
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode** → **link** = **temp** → **link** and '**temp** → **link** = **newNode**'

Deleting a Node from a Linked List

We can delete the node from an already existing linked list. We will consider three cases

Case 1: The first node is deleted.

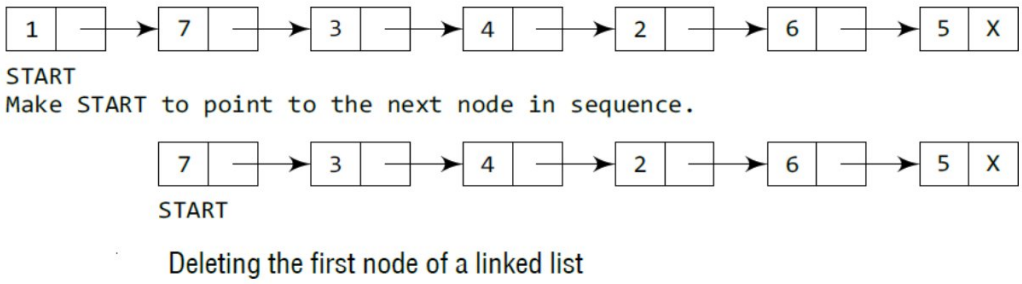
Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete.

When we delete a node from a linked list, we actually have to free the memory occupied by that node. The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

1. Deleting the First Node from a Linked List: Consider the linked list in Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



Deleting the first node of a linked list

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**Root** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **root**.
- **Step 4 -** Check whether list is having only one node (**temp** → **link** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **Root** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **Root** = **temp** → **link**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**Root** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **Root**.
- **Step 4 -** Check whether list has only one Node (**temp1** → **Link** == **NULL**)
- **Step 5 -** If it is **TRUE**. Then, set **Root** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **link** == **NULL**)
- **Step 7 -** Finally, Set **temp2** → **link** = **NULL** and delete **temp1**, **temp2.**

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- **Step 1 -** Check whether list is **Empty** (**Root** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **Root**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node to be deleted, then set **Root= NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == Root**).
- **Step 9 -** If **temp1** is the first node then move the **root** to the next node (**root = temp1 → link**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → link == NULL**).
- **Step 11 -** If **temp1** is last node then set **temp2 → link = NULL** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → link = temp1 → link** and delete **temp1** (**free(temp1)**).

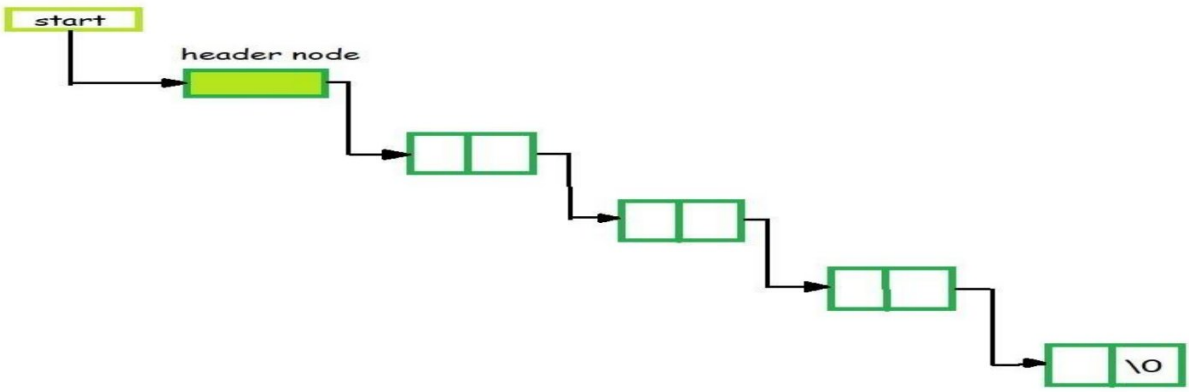
Header Linked Lists

A *header node* is a special node that is found at the *beginning* of the list. A list that contains this type of node, is called the header-linked list. This type of list is useful when information other than that found in each node is needed.

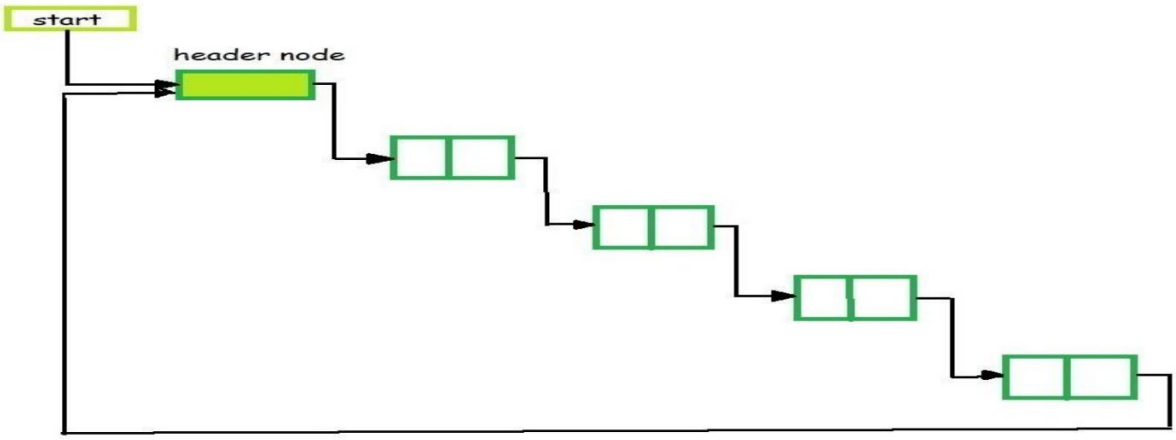
For example, suppose there is an application in which the number of items in a list is often calculated. Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.

Types of Header Linked List

1. **Grounded Header Linked List**
   It is a list whose *last node* contains the *NULL* pointer. In the header linked list the **start** pointer always points to the header node. **start -> next = NULL** indicates that the grounded header linked list is *empty*. The operations that are possible on this type of linked list are *Insertion, Deletion, and Traversing*.

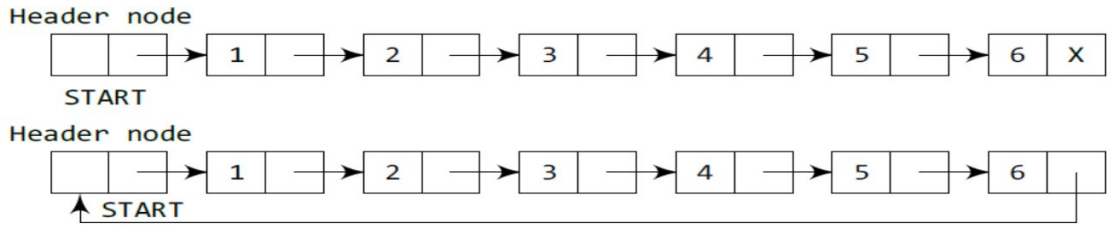


2. **Circular Header Linked List**
   A list in which *last node* points back to the *header node* is called circular Header linked list. The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because last node of a circular linked list does **not contain** the **NULL** pointer. The possible operations on this type of linked list are *Insertion, Deletion and Traversing*.

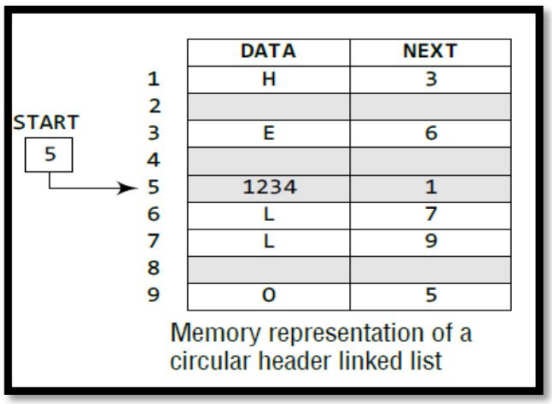


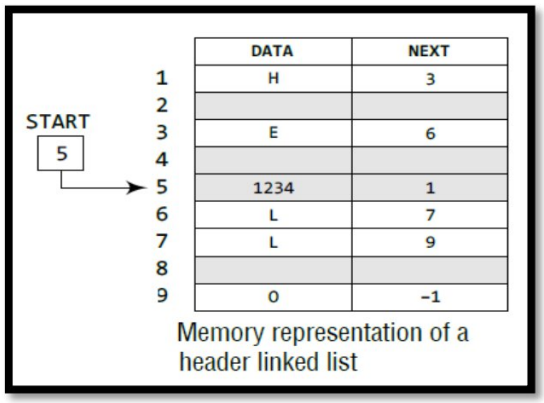
1. *Grounded header linked list* which stores NULL in the next field of the last node.
2. *Circular header linked list* which stores the address of the header node in the next field of the last node. Here, the header node will denote the end of the list.

The following shows both the types of header linked lists.



Header linked list






```
Step 1: SET PTR = START -> NEXT
Step 2: Repeat Steps 3 and 4 while PTR != START
Step 3:            Apply PROCESS to PTR -> DATA
Step 4:            SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: EXIT
```

Algorithm to traverse a circular header linked list

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

Algorithm to insert a new node in a circular header linked list

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

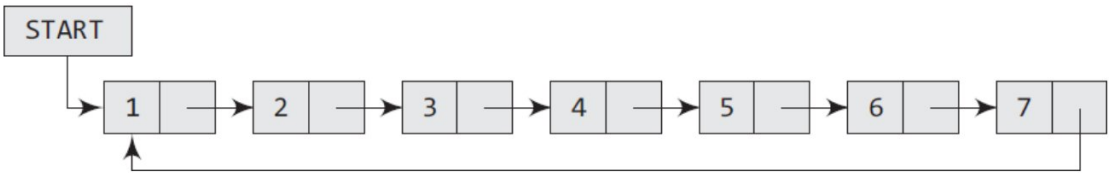
Algorithm to delete a node from a circular header linked list

There is actually just one small difference between these algorithms We have a header list and a circular header list, we also have a two-way (doubly) header list and a circular two-way (doubly) header list. The algorithms to perform all the basic operations will be exactly the same except that the first node will be accessed by writing START -> NEXT instead of START.

CIRCULAR LINKED LISTs

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.



Circular linked list

Traversing:

In a conventional linked list, we traverse the list from the root node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again.

temp =root;

```
if (root != NULL)
 {
     // Keep printing nodes till we reach the first node again
     do
     {
         printf("%d ", temp->data);
         temp = temp->link;
     }
     while (temp != root);
 }
```

a. .Searching for a value in a Circular 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 element is not found is returned.

Algorithm

```
Void search(char val  ,struct node *root)
{
Struct Node * temp=root;
Int val;
Do
{
If(temp->data==val)
    {
    Printf ("element is found");
    Break;
    }
Temp=Temp->link;
}
While(temp!=Root)

}
```

Inserting a New Node in a Circular Linked List

A new node is added into an already existing linked list. We will take 3 cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning of the circular linked list.

Case 2: The new node is inserted at the end of the circular linked list.

Case 3:Inserting At Specific location in the list (After a Node)

1.Inserting a Node at the Beginning of a Circular Linked List

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**Root** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **root** = **newNode** and **newNode→link** = **root** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**root**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp** → **next** == **root**').
- **Step 6 -** Set '**newNode** → **link** =**root**', '**root** = **newNode**' and '**temp** → **link** = **root**'.

2.Inserting a Node at the End of a Circular Linked List

We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** == **head**).
- **Step 6 -** Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

3.Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after

which we want to insert the newNode).

- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.

Deleting a Node from a Circular Linked List

A node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion are same as that given for singly linked lists.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head == NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1 → next == head**)
- **Step 5 -** If it is **TRUE** then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head**)
- **Step 7 -** Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head == NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next == head**)
- **Step 5 -** If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)
- **Step 7 -** Set **temp2 → next = head** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head == NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next == head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head = NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next = head** and

delete **temp1**.

- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next = head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1** (**free(temp1)**).

DOUBLY LINKED LISTS

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list.

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Fig.

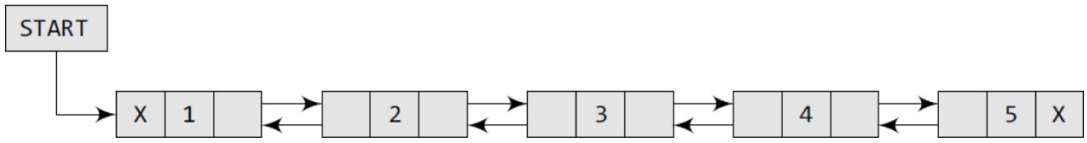


Figure : Doubly linked list

In C, the structure of a doubly linked list can be given as,

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

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

A doubly linked list calls for more space per node and more expensive basic operations. However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a doubly linked list is that it makes searching twice as efficient. Let us view how a doubly linked list is maintained in the memory.

START = 1

	DATA	PREV	NEXT
1	H	−1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	−1

Figure: Memory representation of a doubly linked list

In the figure, we see that a variable START is used to store the address of the first node. In this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field. We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list. When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **previous** as **NULL** and
- **newNode** → **next** as **NULL**
- **Step 2 -** Check whether list is **Empty** (**root** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **newNode** to **root**.
- **Step 4 -** If it is **not Empty** then, create **temp** and assign **new node** and
  Root to **temp** → **next** and
  temp to **root-> previous** and
  temp to **root**

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**Root** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode** → **previous** and **newNode** to **root**
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **root**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp** → **next** and **temp** to **newNode** → **previous**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode** → **previous** & **newNode** → **next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1** → **next** to **temp2**, **newNode** to **temp1** → **next**, **temp1** to **newNode** → **previous**, **temp2** to **newNode** → **next** and **newNode** to **temp2** → **previous**.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp** → **previous** is equal to **temp** → **next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp** → **next** to **head**, **NULL** to **head** → **previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp** → **previous** and **temp** → **next** both

are **NULL**)

- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head == NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5 -** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

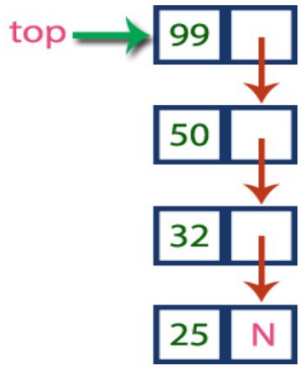
APPLICATIONS OF LINKED LISTS

Polynomials and Sparse Matrix are two important applications of arrays and linked lists. A polynomial is composed of different terms where each of them holds a coefficient and an exponent.

STACK USING SINGLE LINKED LIST

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

NODE Representation:

```
Struct node
{
Int data;
Struct node * link;
};
Struct node * top =NULL;
```

Stack Operations using Linked List

push(value) - Inserting an element into the Stack

```
Void push()
{
Struct node * temp;
temp= (struct node*) malloc (size of (struct node));
printf("enter node data");
scanf("%d",&temp->data);
temp->link=top;
top=temp;
}
```

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value and create **temp** assign **newnode**
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **temp** → **link** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **temp** → **link** = **top**.
- **Step 5 -** Finally, set **top** = **temp**

pop() - Deleting an Element from a Stack

```
Void pop()
{
Struct node * temp;
If(top==null)
{
Printf("stack is empty");
}
else
{
temp=  top;
top = top → link
temp->link = NULL;
free(temp);
}
```

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top** → **link**'.
- **Step 5 -** Then set '**temp->link** = **NULL;**
- **Step 6 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

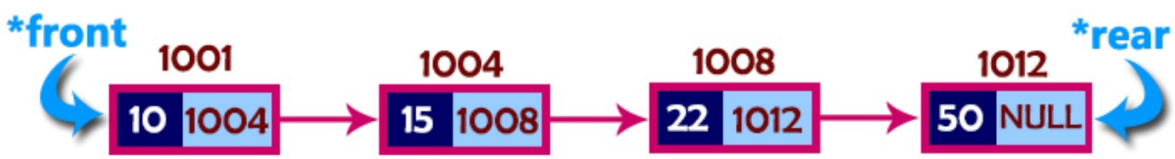
```
Void traverse ()
{
Struct node * temp;
If(top==null)
{
Printf("stack is empty");
}
else
{
temp=  top;
while(temp!=null)
{
printf("%d", temp->data);
temp=temp->link;
}
}
}
```

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp** → **data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp** → **link** != **NULL**).
- **Step 5 -** Finally! Display '**temp** → **data** ---> **NULL**'.

QUEUE USING SINGLE LINKED LIST

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **Link**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

Node representation:

```
Struct node
{
Int data;
Struct node * link;
}
Struct node *front =null;
Struct node * rear;=null;
Newnode=(struct node*)malloc(size of(struct node));
```

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

```
If (front==null && rear==null)
{
Front=rear=newnode;
}
else
{
rear → link = newNode;
rear = newNode;
}
```

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**front=== NULL** &&**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear → link** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

```
struct node *temp;
temp=front;
If (front==null && rear==null)
{
Printf("queue is empty Deletion is not  possible");
}
else
{
front=temp->link;
temp->link=null;
Free(temp);
}
```

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **temp → link**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

```
struct node *temp;
If (front==null && rear==null)
{
Printf("queue is empty");
}
else
{
Temp=front;
While(temp!=NULL)
{
Printf("%d",temp->data);
Temp=temp → link
}
}
```

- **Step 1 -** Check whether queue is **Empty** (**front == NULL&&rear==NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp**!= **NULL**).
- **Step 5 -** Finally! Display '**temp → data**.

The Polynomial Abstract Data type :-

The Polynomial is a sum of terms $c \cdot x^e$, where c is a coefficient, e is the exponent, and x is a variable.

A Polynomial is one of the examples of an ordered list.

Ex :- $7x^2 + 3x + 4$

Operations:

When we think of a Polynomial as an ADT The basic operation as follows.

- → creation of Polynomical
- → addition of two Polynomials
- → Substration of two Polynomials
- → Multiplication of two Polynomials

Definition of Polynomial :-

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

where $a_n, a_0 \cdots$ are numbers

x – variable

Ex : $3x^2 - 5x + 4$

Term = $3x^2 - 5x + 4$

Variable = $3x^2 - 5x + 4 \Rightarrow x$

Coefficient of $3x^2 - 5x + 4$

$\Rightarrow 3, -5, 4$

Exponent of $3x^2 - 5x + 4$

$3x^2 \Rightarrow 2$

$5x \Rightarrow 1$

Checking wheather given term polynomial or not

Ex :

① $3x^2 - 5x + 4$

All of the variables have Integer exponents That are Positive This is Polynomical.

② $10x + 1$

All of the variables have Integer exponents That are Positive This is Polynomical.

Scanned with CamScanner

$5x^9 + 5x^7 - 10) * 2x$

Since All the Variables have Integer Exponents that are Positive This is a Polynomial.

(4) $9x^{-5} + 6$

Not a Polynomial because a term has a negative Exponent

(5) $7x^{1/2} + 3$

not a Polynomial a term has a fraction Exponent.

Degree of Polynomial:-

The Degree is the value of the greatest Exponent.

Ex :- (1) $3x^2 + 5x + 100$ → Degree is "2"

(2) $10x + 1$ → Degree is "1"

(3) $5x^9 + 5x^7 - 10$ → Degree is "9"

Polynomial Representation:-

A Polynomial is an Expression that contains coefficient & Exponent, variables

Ex:- $P(x) = 4x^3 + 6x^2 + x + 9$

A Polynomial Thus may be represented using arrays or linked lists. Array representation assumes That the Exponents of the given Expression are arranged from highest Value (degree) to "0" (Zero) which is represented by the Sub Script of the array begining with "0". The coefficients of the respective Exponents are placed at an appropriate index in the array.

Scanned with CamScanner

Example :-

This representation is very efficient with respect to operations such as store(), retrieve() as it requires constant time.

The conventional algorithms of addition, substraction, multiplication and so on can be used for this representation very efficiently.

Ex1 :- $P(x) = 2x^4 + 3x^3 + x^2 - 2x + 5$

Index	0	1	2	3	4
coefficient	2	3	1	-2	5
Exponent	4	3	2	1	0

Polynomical representation.

Ex-2 :- Represent the $P(x) = 11x^8 + 5x^6 + x^5 + 2x^4 - 3x^2 + x + 10$ The above expression / Term In Polynomical representation array

Index	0	1	2	3	4	5	6	7	8
Coefficient	11	0	5	1	2	0	-3	1	10
Exponent	8	7	6	5	4	3	2	1	0

Scanned with CamScanner

nomial addition :-

ict two Polynomials A and B be

$A = 4x^9 + 8x^6 + 5x^3 + x^2 + 4x$

$B = 3x^7 + x^3 - 2x + 5$

Then $C = A + B = 4x^9 + 3x^7 + 8x^6 + 6x^3 + x^2 + 2x + 5$

The Polynomials A and B are to be added to get The resultant Polynomial C. Here we assume that The two Polynomials are in descending order of Their Exponents.

⇒ let us revise A and B are adding two Polynomials. Let i, j, k be The Three indicies to keep track of The current term of The Polynomials, A, B, C respectively being Processed. Initially, it tracks The fisst term.

The major steps Involved can be listed as follows

⇒ if The Exponents of The two terms of Polynomials A and B are Equal, Then The coffieients are added, and The new term of ~~The~~ Polynomials is stored in The resulltant Polynomial 'c' and advance i, j, k to track to The next term.

⇒ if The ~~term~~ Exponent of term Indicated by 'i' in A is less Than The Exponent of The current term specified by j of B, Then copy The current term of B Pointed by j in The location Pointed by k in Polynomial 'c'. The Pointers j and k are advanced to The next term.

Scanned with CamScanner

→ if The Exponent of the term Pointed by j in B is less than The Exponent of The current term Pointed by i of A, Then copy The current term of a Pointed by i in The location Pointed by k in Polynomical C. Advance The Pointer i and k to the next term.

Each time a new term is generated, its coefficient and Exponent fields are set accordingly. The resultant term then is attached to the end of The Polynomial C. The current term of Polynomical C is indicated by k

Ex:- $A = 4x^9 + 8x^6 + 5x^3 + x^2 + 4x$

$B = 3x^7 + x^3 - 2x + 5$

add The above 2 Polynomials Then

$C = A + B = 4x^9 + 3x^7 + 8x^6 + 6x^3 + x^2 + 2x + 5$

index (i)	0	1	2	3	4
coefficient	4	8	5	1	4
Exponent	9	6	3	2	1

A

index (j)	0	1	2	3
coefficient	3	1	-2	5
Exponent	7	3	1	0

B

index (k)	0	1	2	3	4	5	6
Coefficient	4	3	8	6	1	2	5
Exponent	9	7	6	3	2	1	0

C = A+B

Scanned with CamScanner

Sparse Matrix :-

In Computer Programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' colums and 'n' rows represents a "mxn" matrix.

There may be a situation in which a matrix contains more number of zero values than non-zero values. Such Matrix is known as sparse matrix.

" Sparse Matrix is a matrix which contains very few non-zero elements"

When a sparse Matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix.

Ex:- Consider a matrix of size 100x100 containing only 10 non-zero elements. In This matrix only 10 spaces are filled with non-zero values and remaining spaces of Matrix are filled with zero. That means, totally we allocate 100x100x4 = 40000 bytes of space to store This integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times

* " Main use of sparse matrix is reduce The Scanning time"

Scanned with CamScanner

Sparse Matrix Representations:-

a Sparse Matrix can be represented by using two representation Those are as follows

1. 3 column representation (Triplet Representation) (or) 3-Tuple Representation
2. Linked Representation.

1. 3 column Representation:-

In This representation, we consider only non-zero values along with their row and column index values. In This representation, The 0th row stores total rows, total columns and total non zero values in the matrix.

Ex:- Consider a matrix of size 5×6 containing 6 number of non-zero values. This matrix can be represented as shown below

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}_{5\times 6} \Rightarrow$$

3 column representation

row	column	values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

> Header row

In The above example matrix There are only 6 non-zero elements (Those are 9, 8, 4, 2, 5, 2) and matrix size is (5×6). we represent This matrix as shown in the above Image. Here The first row in The right side table is filled with values 5, 6, 6 which Indicates that it is a Sparse matrix with 5 rows, 6 columns, 6 non zero values second row is filled with 0, 4, 9 which Indicates The value in The matrix 0th row, 4th column is 9. In The same way remaing non zero values filled same Pattern

Scanned with CamScanner

2) Sparse Matrix Linked List Representation :-

In linked list each node has four fields

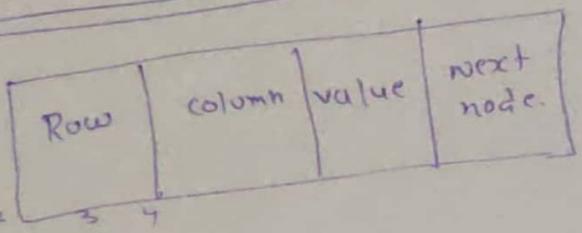
Row : Index of row, where non zero elements is located.

Column : Index of column, where non zero elements is located.

Value : Value of non zero elements located at index. - (row, column).

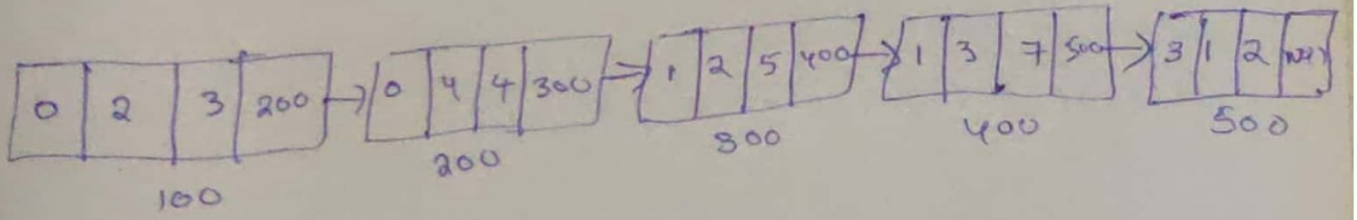
Next node : Address of the next node.

Node structure :-



Ex :-

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	0	0	0



Scanned with CamScanner