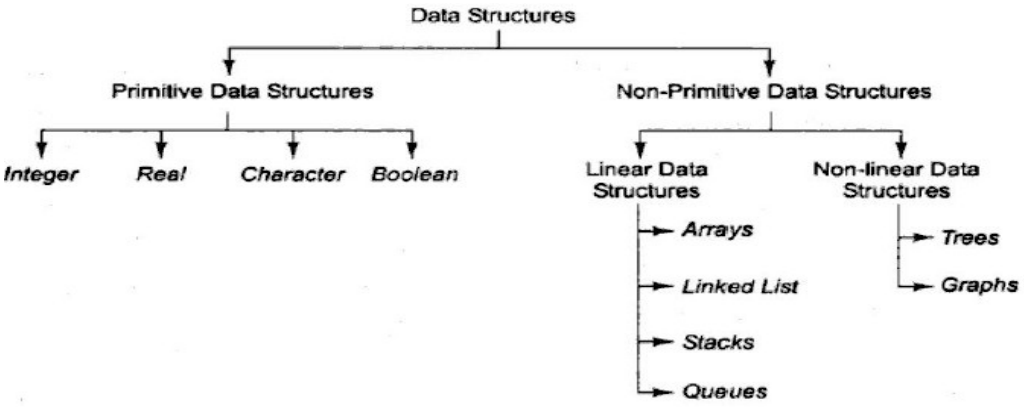
UNIT-I

Introduction to Data structure

Data structure is a method of organizing a large amount of data more efficiently so that any operation on that data becomes easy

A **data structure** is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure.

A **data structure** is a special way of organizing and storing data in a computer so that it can be used efficiently.



Abstract Data Types

Abstract Data type (ADT) is a type for objects whose behaviour is defined by a set of value and a set of operations.
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation independent view.
The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type.

Ex:
List ADT
A list contains elements of same type arranged in sequential order and following operations can be performed on the list.
get() – Return an element from the list at any given position.
insert() – Insert an element at any position of the list.
remove() – Remove the first occurrence of any element from a non-empty list.
removeAt() – Remove the element at a specified location from a non-empty list.
replace() – Replace an element at any position by another element.
size() – Return the number of elements in the list.
isEmpty() – Return true if the list is empty, otherwise return false.
isFull() – Return true if the list is full, otherwise return false.

Linear Data Structures

If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.
Ex:
Arrays
List (Linked List)
Stack
Queue

Linked List

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

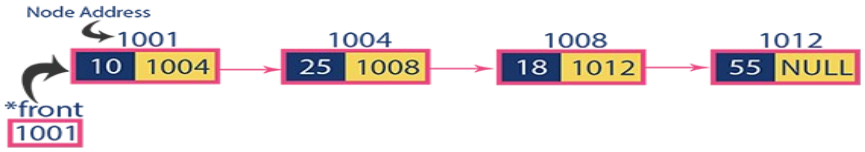
Single Linked List

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.
In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.
The graphical representation of a node in a single linked list is as follows...

Note:

1. In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head" or “start”).
2. Always next part (reference part) of the last node must be NULL.

Ex:



Operations on Single Linked List

The following operations are performed on a Single Linked List

- **Insertion**
- **Deletion**
- **Display**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program.
Step 2 - Declare all the **user defined functions**.
Step 3 - Define a **Node** structure with two members **data** and **next**
Step 4 - Define a Node pointer '**head**' and set it to **NULL**.
Step 5 - Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

```
struct node   //Each node in list will contain data and next pointer
{
int data;
struct node *next;
}*start=NULL;
```

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are 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 single 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 **newNode→next = NULL** and **head = newNode.**
Step 4 - If it is **Not Empty** then, set **newNode→next = head** and **head = newNode.**

```
void insertbeg()
{
    struct node *nn;
    int a;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("enter data:");
    scanf("%d",&nn->data);
    a=nn->data;
    nn->next=NULL;
    if(start==NULL)         //checking if List is empty
    {
            printf("\n list is empty, so new node inserted as start node\n");
            start=nn;
    }
    else    //list contains elements
    {
        nn->next=start;
        start=nn;
    }
    printf("\n %d succesfully inserted\n",a);
}
```

Inserting At End of the 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 → next** as **NULL.**
Step 2 - Check whether list is **Empty(head == NULL).**
Step 3 - If it is **Empty** then, set **head = newNode**.
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** is equal to **NULL**).
Step 6 - **Set temp → next = newNode.**

```
void insertend()
{
    struct node *nn,*t;
    int b;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("\n enter data:");
    scanf("%d",&nn->data);
    b=nn->data;
    nn->next=NULL;
    if(start==NULL)    //checking if List is empty
    {
            printf("\n list is empty, so new node inserted as start node\n");
            start=nn;
    }
    else    //list contains elements
    {
        t=start;
        while(t->next!=NULL)
        {
            t=t->next;
        }
        t->next=nn;
     }
    printf("%d is succesfully inserted\n",b);
}
```

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 single 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 **newNode → next = NULL** and **head = newNode**.
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 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 → next = temp → next' and 'temp → next = newNode'**

```
void insertmid()
{
    struct node *nn,*t=start;
    int x,v;
    nn=(struct node *)malloc(sizeof(struct node));
    if(start==NULL)  //checking if List is empty
    {
        printf("sll is empty\n"); return;
    }
    printf("\n enter data:");
    scanf("%d",&nn->data);
    v=nn->data;
    printf("enter data after which no. is to be inserted:\n");
    scanf("%d",&x);
    while(t!=NULL && t->data!=x)
    {
            t=t->next;
              }
              if(t==NULL)
              {
                        printf("\n data does not exist",x);
                        return;
              }
              else
              {
                nn->next=t->next;
                t->next=nn;
                printf("%d succesfully inserted\n",v);
              }
}
```

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are 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 single 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 → next == NULL**)
Step 5 - If it is **TRUE** then set **head = NULL** and delete **temp** (Setting **Empty** list conditions)
Step 6 - **If it is FALSE then set head = temp → next, and delete temp.**

```
void deletebeg()
{
        struct node *t=start;
        if(start==NULL)  //list is empty
        {
        printf("\n sll is empty");
        return;
        }
        else
        {
        if(start->next==NULL) //list contains single element
        {
        start=NULL;
        }
        else   //list contains more than one element
        {
        start=start->next;
        t->next=NULL;
        }
}
printf("\n%d is successfully deleted",t->data);
free(t);
}
```

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(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 == NULL**)
Step 5 - If it is **TRUE**. Then, set **head = 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 → next == NULL**)
Step 7 - **Finally, Set temp2 → next = NULL and delete temp1.**

```
void deleteend()
{
        struct node *t=start,*t1;
        int x;
        if(start==NULL)  //list is empty
        {
        printf("\n sll is empty");
        }
        else
        {
        if(start->next==NULL) //list contains single element
        {
        start=NULL;
        }
        else  // list contains more elements
        {
        t1=t->next;
        while(t1->next!=NULL)
        {
                t=t->next;
                t1=t1->next;
        }
        t->next=NULL;
        printf("\n%d is successfully deleted",t1->data);
        free(t1);
        }
}
}
```

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(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 or not
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 move the **head** to the next node (**head = head → next**) and delete **temp1**.
Step 10 – If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
Step 11 – If **temp1** is last node then set **temp2 → next = NULL** 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)).**

```
void deletemid()
{
        struct node *t=start,*t1;
        int x;
        if(start==NULL)  //list is empty
        {
        printf("\n sll is empty");
        }
        else
        {
        if(start->next==NULL) //list contains single element
        {
        start=NULL;
        }
        else
        {
          printf("enter data to be deleted:");
          scanf("%d",&x);
          t1=start->next;
         while(t1!=NULL && t1->data!=x)
        {
            t=t->next;
            t1=t1->next;
         }
          if(t1==NULL)
          {
                printf("\n data does not exist",x);
          }
          else
          {
              t->next=t1->next;
              t1->next=NULL;
             printf("\n%d is successfully deleted",t1->data);
             free(t1);
          }
          }
}
}
```

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is **Empty(head == 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 **head**.
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).**

```
void display()
{
    struct node *t;
    if(start==NULL)
    {
        printf("sll is empty\n");
        return;
    }
    printf("elements are:\n");
    t=start;
    while(t!=NULL)
    {
        printf("--> %d",t->data);
        t=t->next;
    }
    return;
}
```

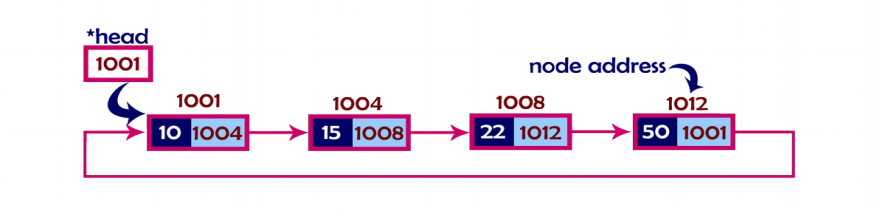
Circular Linked List

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

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.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

Ex



Operations

In a circular linked list, we perform the following operations...

- **Insertion**
- **Deletion**
- **Display**

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program.
Step 2 - Declare all the **user defined** functions.
Step 3 - Define a **Node** structure with two members **data** and **next**
Step 4 - Define a Node pointer '**head**' and set it to **NULL**.
Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

```
struct node   //Each node in list will contain data and next pointer
{
int data;
struct node *next;
}*last=NULL;
```

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are 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 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 (until '**temp → next == head**').
Step 6 - Set '**newNode → next = head**', '**head = newNode**' and '**temp → next = head**'.

```
void insertbeg()
{
    struct node *nn;
    int a;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("enter data:");
    scanf("%d",&nn->data);
    a=nn->data;
    nn->next=NULL;
    if(last==NULL)          //checking if List is empty
    {
                printf("\n list is empty, so new node inserted as start node\n");
                last=nn;
                last->next=nn;
    }
    else    //list contains elements
    {
        nn->next=last->next;
        last->next=nn;
    }
    printf("\n %d succesfully inserted\n",a);
}
```

Inserting At End of the 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.**

```
void insertend()
{
    struct node *nn,*t;
    int b;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("\n enter data:");
    scanf("%d",&nn->data);
    b=nn->data;
    nn->next=NULL;
    if(last==NULL)    //checking if List is empty
    {
                printf("\n list is empty, so new node inserted as start node\n");
                last=nn;
                last->next=nn;
    }
    else    //list contains elements
    {
        nn->next=last->next;
        last->next=nn;
        last=last->next;
     }
    printf("%d is succesfully inserted\n",b);
}
```

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 9 - If temp is not last node then set **newNode → next = temp → next** and **temp → next = newNode.**

```
void insertmid()
{
    if(last==NULL)   //checking if List is empty
    {
        printf("cll is empty\n");
        return;
    }
    else
    {
              if(last->next==last)     //list contains one element
              {
              printf("cll contains only one node, So insert begin or end of cll");
              }
              else       // list contains more than one element
              {
                   struct node *nn,*t=last;
                   nn=(struct node *)malloc(sizeof(struct node));
                                int x,v;
                                printf("\n enter data:");
                              scanf("%d",&nn->data);
                  v=nn->data;
                  printf("enter data after which no. is to be inserted:\n");
                            scanf("%d",&x);
                    while(t->next!=last && t->data!=x)
                       {
              t=t->next;
                        }
                       if(t!=last)
                        {
                             nn->next=t->next;
                             t->next=nn;
              printf("%d succesfully inserted\n",v);
                        }
                       else
                        {
                             printf("\n data does not exist or it is last node please choose other node",v);

                        }
                       }
               }
}
```

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are 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 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.**

```
void deletebeg()
{
                if(last==NULL)  //list is empty
                {
                printf("\n cll is empty");
                }
                else
                {
                if(last->next==last) //list contains single element
                {
                struct node *t=last;
                last=NULL;
                t->next=NULL;
                printf("\ncll contains only one element and %d is successfully deleted",t->data);
                free(t);
                }
                else   //list contains more than one element
                {
                struct node *t=last->next;
                last->next=t->next;
                t->next=NULL;
                printf("\n%d is successfully deleted",t->data);
                free(t);
                }
  }
}
```

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.**

```
void deleteend()
{
                if(last==NULL)  //list is empty
                {
                printf("\n cll is empty");
                }
                else
                {
                if(last->next==last) //list contains single element
                {
                struct node *t=last;
                last=NULL;
                t->next=NULL;
                printf("\ncll contains one element and %d is successfully deleted",t->data);
                free(t);
                }
                else  // list contains more elements
                {
                struct node *t=last->next,*t1;
                while(t->next!=last)
                t=t->next;
                t->next=last->next;
                t1=last;
                last=t;
                t1->next=NULL;
                printf("\n%d is successfully deleted",t1->data);
                free(t1);
                }
}
}
```

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)).**

```
void deletemid()
{
        if(last==NULL)  //list is empty
        {
        printf("\n cll is empty");
        }
        else
        {
        struct node *t=last,*t1;
        if(last->next==last) //list contains single element
        {
        last=NULL;
        t->next=NULL;
        printf("\ncll contains one element and %d is successfully deleted",t->data);
        free(t);
        }
        else  //list contains more elements
        {
         int x;
                printf("enter data to be deleted:");
         scanf("%d",&x);
         t1=t->next;
        while(t1->data!=x && t1!=last)
        {
                t=t->next;
                t1=t1->next;
                }
                 if(t->next==last)
                {
                printf("\n data does not exist or it is last node please choose other node",x);
        }
        else
        {
                t->next=t1->next;
                        t1->next=NULL;
                        printf("\n%d is successfully deleted",t1->data);
                        free(t1);
        }
        }
}
}
```

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

Step 1 - Check whether list is **Empty(head == 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 **head**.
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 **head → data.**

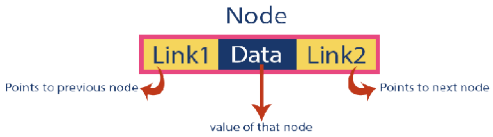
```
void display()
{
      if(last==NULL)
      {
                  printf("\ncll is empty");
      }
      else
      {
       struct node *t,*t1;
       t=last->next;
       t1=t;
       printf("elements are:\n");
       while(t->next!=t1)
          {
           printf("--> %d",t->data);
           t=t->next;
          }
         printf("--> %d",t->data);
      }
      }
```

Double Linked List

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

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 cannot traverse back. We can solve this kind of problem by using a double linked list.

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure.

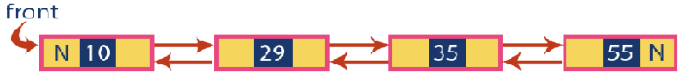


Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Note:

1. In double linked list, the first node must be always pointed by head.
2. Always the previous field of the first node must be NULL.
3. Always the next field of the last node must be NULL.

Ex:



Operations:

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

Step 1 - Include all the header files which are used in the program.
Step 2 - Declare all the user defined functions.
Step 3 - Define a Node structure with three members **previous**, **next** and **data**

Step 4 - Define a Node pointer '**head**' and set it to **NULL.**
Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

```
struct node   //Each node in list will contain data, previous and next pointer
{
    struct node *prev;
    int data;
    struct node *next;
}*start=NULL;
```

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**.
Step 2 - Check whether list is **Empty(head == NULL)**
Step 3 - If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
Step 4 - If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

```
void insertbeg()
{
    struct node *nn;
    int a;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("enter data:");
    scanf("%d",&nn->data);
    a=nn->data;
    nn->next=nn->prev=NULL;
    if(start==NULL)         //checking if List is empty
    {
      printf("\n dll is empty, so new node inserted as start node\n");
      start=nn;
    }
    else    //list contains elements
    {
        nn->next=start;
        start->prev=nn;
        start=nn;
    }
    printf("\n %d succesfully inserted\n",a);
}
```

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(head == NULL)**
Step 3 - If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **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** is equal to **NULL**).
Step 6 - Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

```
void insertend()
{
    struct node *nn,*t;
    int b;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("\n enter data:");
    scanf("%d",&nn->data);
    b=nn->data;
    nn->next=nn->prev=NULL;
    if(start==NULL)    //checking if List is empty
    {
            printf("\n dll is empty, so new node inserted as start node\n");
            start=nn;
    }
    else    //list contains elements
    {
        t=start;
        while(t->next!=NULL)
        {
            t=t->next;
        }
        t->next=nn;
        nn->prev=t;
     }
    printf("%d is succesfully inserted\n",b);
}
```

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.
Step7- Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → Previous**, **temp2** to **new Node → next** and **newNode** to **temp2→ previous.**

```
void insertmid()
{
    struct node *nn,*t=start;
    int x,v;
    nn=(struct node *)malloc(sizeof(struct node));
     if(start==NULL)  //checking if List is empty
    {
        printf("dll is empty\n");
        return;
    }
                printf("\n enter data:");
                scanf("%d",&nn->data);
                v=nn->data;
                nn->next=NULL;
                nn->prev=NULL;
                printf("enter data after which no. is to be inserted:\n");
                scanf("%d",&x);
                while(t!=NULL && t->data!=x)
                {
                t=t->next;
                                }
                                if(t==NULL)
                                {
                                                printf("\n data does not exist",x);
                                                return;
                                }
                                else
                                {
                                                if(t->next==NULL)  // dll contains one element
                                                {
                                                                t->next=nn;
                                                nn->prev=t;
                                }
                                else
                                {
                                                t->next->prev=nn;
                                                                nn->prev=t;
                                                                nn->next=t->next;
                                                                t->next=nn;
                                }
                                                printf("%d succesfully inserted\n",v);
                                }
}
```

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.**

```
void deletebeg()
{
                struct node *t=start;
                if(start==NULL)  //list is empty
                {
                printf("\n dll is empty");
                return;
                }
                else
                {
                if(start->next==NULL) //list contains single element
                {
                start=NULL;
                }
                else   //list contains more than one element
                {
                start=start->next;
                t->next=NULL;
                start->prev=NULL;
                }
}
printf("\n%d is successfully deleted",t->data);
free(t);
}
```

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.**

```
void deleteend()
{
                struct node *t=start;
                int x;
                if(start==NULL)  //list is empty
                {
                printf("\n dll is empty");
                return;
                }
                else
                {
                if(start->next==NULL) //list contains single element
                {
                start=NULL;
                printf("dll contains only one element and %d is deleted",t->data);
                free(t);
                }
                else  // list contains more elements
                {
                while(t->next!=NULL)
                {
                                t=t->next;
                }
                t->prev->next=NULL;
                t->prev=NULL;
                printf("\n%d is successfully deleted",t->data);
                free(t);
                }
}
}
```

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)).**

```
void deletemid()
{
                struct node *t=start;
                int x;
                if(start==NULL)  //list is empty
                {
                printf("\n dll is empty");
                return;
                }
                else
                {
                if(start->next==NULL) //list contains single element
                {
                start=NULL;
                printf("dll contains only one element and %d is deleted",t->data);
                }
                else
                {
                  printf("enter data to be deleted:");
                  scanf("%d",&x);
                  while(t!=NULL && t->data!=x)
                {
                t=t->next;
                }
                if(t==NULL)
                  {
                                printf("\n data does not exist",x);
                  }
                  else
                  {
                                if(t->next==NULL)  // deleting last element in dll
                                {
                                                printf("\ndeleting last element in dll");
                                deleteend();
                                                return;
                }
                if(t->prev==NULL)   //deleting first element in dll
                {
                                printf("\ndeleting first element in dll");
                                deletebeg();
                                return;
                }
                else    //deleting middle element in dll
                {
                                t->prev->next=t->next;
                                                t->next->prev=t->prev;
                                                t->next=t->prev=NULL;
                }
                   printf("\n%d is successfully deleted",t->data);
                   free(t);
                  }

                }
}
}
```

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

Step 1 - Check whether list is **Empty(head == 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 **head**.
Step 4 - Display '**NULL <--- **'.
Step 5 - Keep displaying **temp → data** with an arrow (<===>) until **temp** reaches to the last node
Step 6 - Finally, display **temp → data** with arrow pointing to **NULL (temp → data ---> NULL).**

```
void display()
{
    struct node *t;
    if(start==NULL)
    {
        printf("dll is empty\n");
        return;
    }
    printf("elements are:\n");
    t=start;
    while(t!=NULL)
    {
        printf("<--> %d",t->data);
        t=t->next;
    }
}
```

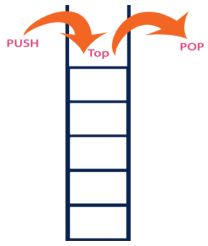
Stack ADT

Stack is a linear data structure in which the operations are performed based on LIFO principle.

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end.

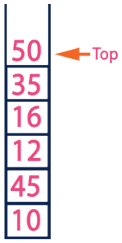
In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) or FILO (First In Last Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

Ex:
If we want to create a stack by inserting elements like 10, 45, 12, 16, 35 and 50. Then 10 becomes the bottom-most element and 50 is the topmost element. The last inserted element 50 is at Top of the stack as shown in the image below...



Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

Stack data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a stack is implemented using an array, that stack can organize an only limited number of elements. When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

Stack Implementation Using Array

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values.

This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**.

Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
Step 2 - Declare all the **functions** used in stack implementation.
Step 3 - Create a one dimensional array with fixed size (**int stack[SIZE]**)
Step 4 - Define a integer variable **'top'** and initialize with **'-1'. (int top = -1)**
Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

```
#include<stdio.h>
#include<stdlib.h>
#define max 10
int top=-1, stack[max];
void push();
void pop();
void display();
void main()
{
int choice;
while(1)
{
printf("\n\n***** Stack MENU *****\n");
printf("1.push\n2.pop\n3.display\n4.exit\n");
printf("enter your choice:\n");
scanf("%d",&choice);
switch(choice)
{
case 1: push();break;
case 2: pop();break;
case 3: display();break;
case 4:exit(0);
default: printf("Wrong selection!!! Try again!!!\n");
}
}
}
```

PUSH (value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack.

We can use the following steps to push an element on to the stack...

Step 1 - Check whether **stack** is **FULL. (top == SIZE-1)**
Step 2 - If it is **FULL,** then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
Step 3 - If it is **NOT FULL,** then increment **top value** by one **(top++)** and set stack[top] to value **(stack[top] = value).**

```
void push()
{
int element;
if(top==max-1)                                  //stack is full
printf("Stack is Full!!! Insertion is not possible!!!(stack overflows)\n");
else                                    //stack is not full
{
printf("enter the element\n");
scanf("%d",&element);
stack[++top]=element;
printf("\nInsertion success!!!");
}
}
```

POP() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter.

We can use the following steps to pop an element from the stack...

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)
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 delete **stack[top]** and decrement top value by one **(top--).**

```
void pop()
{
int element;
if(top==-1)     //stack is empty
printf("Stack is Empty!!! Deletion is not possible!!!(stack underflows)\n");
else                                    //stack contains elements
{
printf("the element %d at position %d is deleted\n",stack[top],top);
top=top-1;
}
}
```

Display() - Displays the elements of a Stack

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

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)
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 variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
Step 4 - Repeat above step until i value becomes '**0**'.

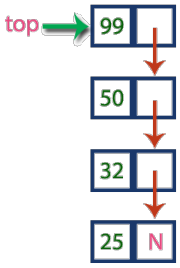
```
void display()
{
int i;
if(top==-1)     //stack is empty
printf("Stack is Empty!!!(stack underflows)\n");
else                                    //stack contains elements
{
printf("***Stack elements are***\n");
for(i=top;i>=0;i--)
{
printf("%d\n",stack[i]);
}
}
}
```

Stack Using 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**.

Ex:



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.

Stack Operations using Linked List

To implement a stack using a 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 **next**.

Step 3 - Define a **Node** pointer '**top**' and set it to **NULL**.

Step 4 - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

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

struct node
{
   int data;
   struct node *next;
}*top=NULL,*top1,*temp;
```

Push(value) - Inserting an element into the Stack

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

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether stack is **Empty** (**top** == **NULL**)

Step 3 - If it is **Empty**, then set **newNode** → **next** = **NULL**.

Step 4 - If it is **Not Empty**, then set **newNode** → **next** = **top**.

Step 5 - Finally, set **top = newNode.**

```
void push()
{
int element;
printf("enter the element\n");
scanf("%d",&element);

   if (top == NULL)        //stack is empty
   {
                top =(struct node *)malloc(sizeof(struct node));
                top->next = NULL;
                top->data = element;
   }
   else         //stack contains elements
   {
                temp =(struct node *)malloc(sizeof(struct node));
                temp->next = top;
                temp->data = element;
                top = temp;
   }

}
```

Pop() - Deleting an Element from a Stack

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 → next**'.
Step 5 - Finally, **delete 'temp'. (free(temp)).**

```
void pop()
{
  if (top == NULL)        //stack is empty
  {
                printf("Error :stack underflow\n");
                return;
  }
  else          //stack contains elements
  top1 = top;
  top = top->next;
  printf("Popped value : %d\n", top1->data);
  top1->next=NULL;
  free(top1);
}
```

Display() - Displaying stack of elements

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 → next** != **NULL**).
Step 5 - Finally! **Display 'temp → data ---> NULL'.**

```
void display()
{
  if (top == NULL)              //stack is empty
  {
                printf("Stack is empty\n");
                return;
  }
  top1 = top;
  printf("***Stack elements are***\n");
  while (top1 != NULL)        //stack contains elements
  {
                printf("%d\n", top1->data);
                top1 = top1->next;
  }
}
```

Stack Applications

Three applications of stacks are presented here. These examples are central to many activities that a computer must do and deserve time spent with them.

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

Expressions

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into 3 types. They are as follows...

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

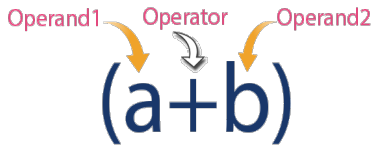
Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Ex:



Postfix Expression

In postfix expression, operator is used after operands. We can say that **"Operator follows the Operands"**.

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Ex:

Prefix Expression

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Ex:

Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Infix to Postfix Conversion

Any expression can be represented using three types of expressions (Infix, Postfix, and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Ex:

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

Step 1 - The Operators in the given Infix Expression : **= , + , ***

Step 2 - The Order of Operators according to their preference : * , + , =

Step 3 - Now, convert the first operator * ----- D = A + B C *

Step 4 - Convert the next operator + ----- D = A BC* +

Step 5 - Convert the next operator = ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

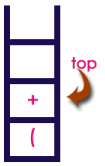
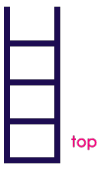
1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is **operand**, then directly print it to the result (Output).
3. If the reading symbol is **left parenthesis '('**, then Push it on to the Stack.
4. If the reading symbol is **right parenthesis ')'**, then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is **operator (+ , - , * , / etc.,)**, then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Ex:

Consider the following Infix Expression...

(A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as follows...

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY	EMPTY
(	Push '(' 	EMPTY
A	No operation Since 'A' is OPERAND 	A
+	'+' has low priority than '(' so, PUSH '+' 	A
B	No operation Since 'B' is OPERAND 	A B
)	POP all elements till we reach '(' POP '+' POP '(' 	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*' 	A B +
(	PUSH '(' 	A B +
C	No operation Since 'C' is OPERAND 	A B + C
-	'-' has low priority than '(' so, PUSH '-' 	A B + C
D	No operation Since 'D' is OPERAND 	A B + C D
)	POP all elements till we reach '(' POP '-' POP '(' 	A B + C D -
$	POP all elements till Stack becomes Empty	A B + C D - *

The final Postfix Expression is as follows...

A B + C D - *

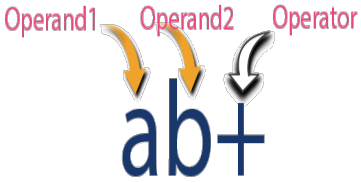
Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Ex:



Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is **operand**, then push it on to the Stack.
3. If the reading symbol is **operator (+ , - , * , / etc.,)**, then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

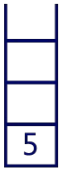
Ex:

Consider the following Expression...

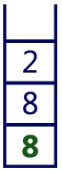
Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

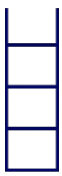
Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result) 
		value1 = pop(); // 3
		value2 = pop(); // 5
		result = 5 + 3; // 8
		Push(8)
		(5 + 3)

8	push(8) 	(5 + 3)
2	push(2) 	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result) 
		value1 = pop(); // 2
		value2 = pop(); // 8
		result = 8 - 2; // 6
		Push(6)
		(8 - 2)
		(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result) 
		value1 = pop(); // 6
		value2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3) * (8 - 2)

$ End of Expression	result = pop() 	Display (result) **48** As final result

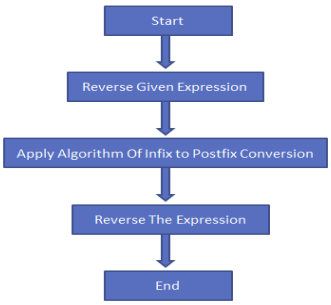
Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

```
//infix to postfix conversion
#include<stdio.h>
char infix[50],post[50];
int top=0, stack[20];
void postfix();
void push(int);
char pop();

void main()
{
  printf("\n Enter the infix expression:");
  scanf("%s",infix);
  postfix();
}
void postfix()
{
  int i,j=0;
  for(i=0; infix[i]!=0; i++)
  {
   switch(infix[i])
   {
    case '+': while(stack[top] >= 1)
                        post[j++]=pop();
                        push(1);
                        break;
    case '-': while(stack[top] >= 1)
                        post[j++]=pop();
                        push(2);
                        break;
    case '*': while(stack[top] >= 3)
                        post[j++]=pop();
                        push(3);
                        break;
    case '/': while(stack[top] >= 3)
                        post[j++]=pop();
                        push(4);
                        break;
    case '^': while(stack[top] >= 4)
                        post[j++]=pop();
                        push(5);
                        break;
    case '(': push(0);
                        break;
    case ')': while(stack[top] != 0)
                        post[j++]=pop();
                        top--;
                        break;
    default: post[j++]=infix[i];
    }
  }
  while(top>0)
  {
    post[j++]=pop();

  }
  printf("\n Postfix Expression is => %s\n", post);
}

 void push(int ele)
  {
   top++;
   stack[top]=ele;
  }
  char pop()
  {
   int el;
   char e;
   el=stack[top];
   top--;
   switch(el)
   {
    case 1:e='+';
                        break;
    case 2:e='-';
                        break;
    case 3:e='*';
                        break;
    case 4:e='/';
                        break;
    case 5:e='^';
                        break;
   }
   return(e);
}
```

Infix to Prefix Conversion

Step 1. Push “)” onto STACK, and add “(“ to end of the A
Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
Step 3. If an operand is encountered add it to BStep
4. If a right parenthesis is encountered push it onto STACKStep
5. If an operator is encountered then:
a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
b. Add operator to STACK
Step 6. If left parenthesis is encontered then
a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
b. Remove the left parenthesisStep
7. Exit



Expression = **(A+B^C)*D+E^5**
Step 1. Reverse the infix expression.
5^E+D*)C^B+A(
Step 2. Make Every '(' as ')' and every ')' as '('
5^E+D*(C^B+A)
Step 3. Convert expression to postfix form.

A+(B*C-(D/E-F)*G)*H

Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
*(C^B+A)	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

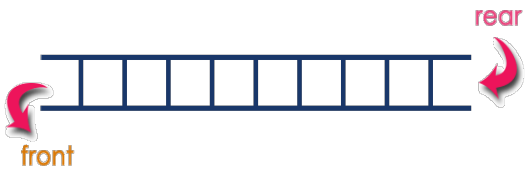
Step 4. Reverse the expression.
+*+A^BCD^E5
Result
+*+A^BCD^E5

```
//infix to prefix conversion
#include<stdio.h>
#include<string.h>

char infix[50],post[50],*pre,*rev;
int top=0, stack[20];
void postfix();
void push(int);
char pop();

int main()
{
  printf("\n Enter the infix expression:");
  scanf("%s",infix);
  rev=strrev(infix);
  postfix();
  return 0;
}
void postfix()
{
  int i,j=0;
  for(i=0; rev[i]!=0; i++)
  {
   switch(rev[i])
   {
    case '+': while(stack[top] >= 1)
                        post[j++]=pop();
                        push(1);
                        break;
    case '-': while(stack[top] >= 1)
                        post[j++]=pop();
                        push(2);
                        break;
    case '*': while(stack[top] >= 3)
                        post[j++]=pop();
                        push(3);
                        break;
    case '/': while(stack[top] >= 3)
                        post[j++]=pop();
                        push(4);
                        break;
    case '^': while(stack[top] >= 4)
                        post[j++]=pop();
                        push(5);
                        break;
    case '(': push(0);
                        break;
    case ')': while(stack[top] != 0)
                        post[j++]=pop();
                        top--;
                        break;
    default: post[j++]=rev[i];
   }
  }
  while(top>0)
{
    post[j++]=pop();

}
 pre=strrev(post);
printf("\n Prefix Expression is => %s", pre);
  }

 void push(int ele)
  {
   top++;
   stack[top]=ele;
  }
  char pop()
  {
   int el;
   char e;
   el=stack[top];
   top--;
   switch(el)
   {
    case 1:e='+';
                        break;
    case 2:e='-';
                        break;
    case 3:e='*';
                        break;
    case 4:e='/';
                        break;
    case 5:e='^';
                        break;
   }
   return(e);
}
```

Queue ADT

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and

deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.



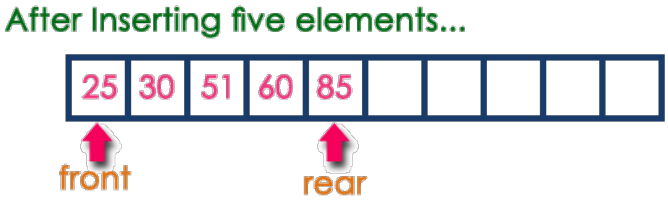
In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

Queue data structure can be defined as follows...

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Ex:
Queue after inserting 25, 30, 51, 60 and 85.



Operations on a Queue

The following operations are performed on a queue data structure...

1. enQueue(value) - (To insert an element into the queue)
2. deQueue() - (To delete an element from the queue)
3. display() - (To display the elements of the queue)

Queue data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

When a queue is implemented using an array, that queue can organize an only limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

Queue Datastructure Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables '**front**' and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

Step 1 - Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
Step 2 - Declare all the **user defined functions** which are used in queue implementation.
Step 3 - Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
Step 4 - Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

```
#define max 6
int queue[max],front=-1,rear=-1;
void add();
void del();
void display();
void main()
{
int choice;
while(1)
{
printf("enter the choice\n");
printf("\n1.Add\n2.Delete\n3.Display\n4.Exit\n");
printf("\nenter the choice:");
scanf("%d",&choice);
switch(choice)
{
case 1: add();break;
case 2: del();break;
case 3: display();break;
case 4:exit(0);
default: printf("you have entered wrong choice\n");
}
}
}
```

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1 - Check whether **queue** is **FULL**. (**rear == SIZE-1**)
Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
Step 3 - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

```
void add()
{
int element;
if(rear==max-1)
printf("queue is full\n");
else
{
if(front==-1)
front=0;
printf("enter the element\n");
scanf("%d",&element);
queue[++rear]=element;
}
}
```

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)
Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

```
void del()
{
int element;
if(front==-1)
printf("queue is empty\n");
else
{
printf("the element%d at position%d is deleted\n",queue[front-1],++front);
if(front>rear)
{
front=-1;
rear=-1;
}
}
}
```

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)
Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
Step 4 - Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

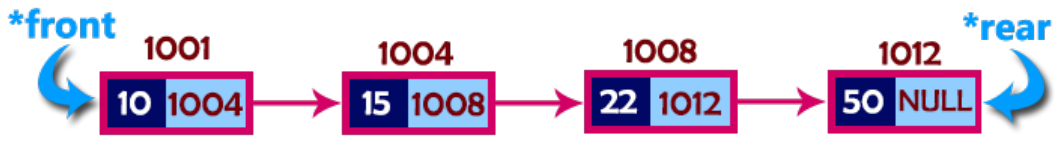
```
void display()
{
int i;
if(front==-1)
printf("queue is an empty\n");
else
{
printf("Queue elements:\n");
for(i=front;i<=rear;i++)
printf("%d\t",queue[i]);
}
}
```

Queue Using 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**'.

Ex:



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 **next**.

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.

```
struct Node
{
  int data;
  struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Queue Implementation using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
                case 1: printf("Enter the value to be insert: ");
                 scanf("%d", &value);
                 insert(value);
                 break;
                case 2: delete(); break;
                case 3: display(); break;
                case 4: exit(0);
                default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
```

enQueue(value) - Inserting an element into the Queue

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

Step 1 - Create a **newNode** with given value and set '**newNode** → **next**' to **NULL**.
Step 2 - Check whether queue is **Empty** (**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** → **next** = **newNode** and **rear** = **newNode**.

```
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
```

deQueue() - Deleting an Element from Queue

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** = **front** → **next**' and delete '**temp**' (**free(temp)**).

```
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    front = front -> next;
    printf("\nDeleted element: %d\n", temp->data);
    free(temp);
  }
}
```

display() - Displaying the elements of Queue

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

Step 1 - Check whether queue is **Empty** (**front** == **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 → next** != **NULL**).
Step 5 - Finally! Display '**temp → data** ---> **NULL**'.

```
void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
                printf("%d--->",temp->data);
                temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }
}
```