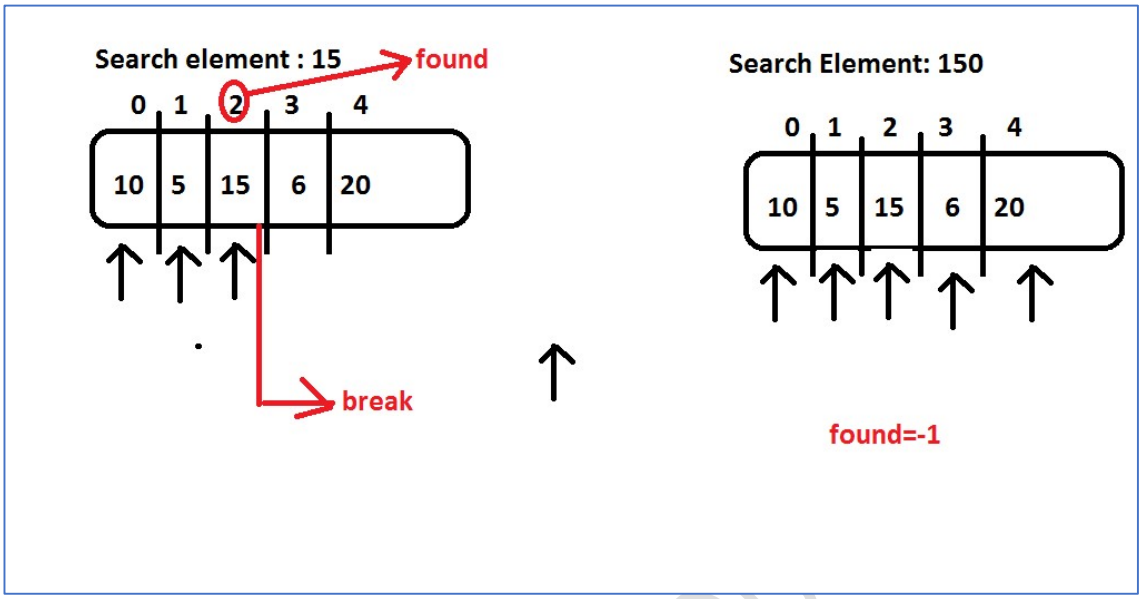
Linear Search




```c
/* Searching
    used to check whether an required element
    is available in the list or not
    Two searching algorithms
    1. Linear Search or sequential search
    2. Binary Search

    Result:
    -1 --> Not found
    >=0 --> Found at position
*/

# include<stdio.h>

void main(){

    int numlist[] = {10,5,15,6,20};
    int element=150;
    int result;
    int linearsearch(int[],int);

    result = linearsearch(numlist,element);

    if(result>=0)
    printf("\n %d is found at %d position",
    element,result);
    else
    printf("\n %d is not found",element);
}
```

```
int linearsearch(int p[],int ele){
    int found=-1,i;

    for(i=0;i<5;i++){
        if(p[i]==ele){
        found=i;
        break;
        }
    }

    return found;

}
```

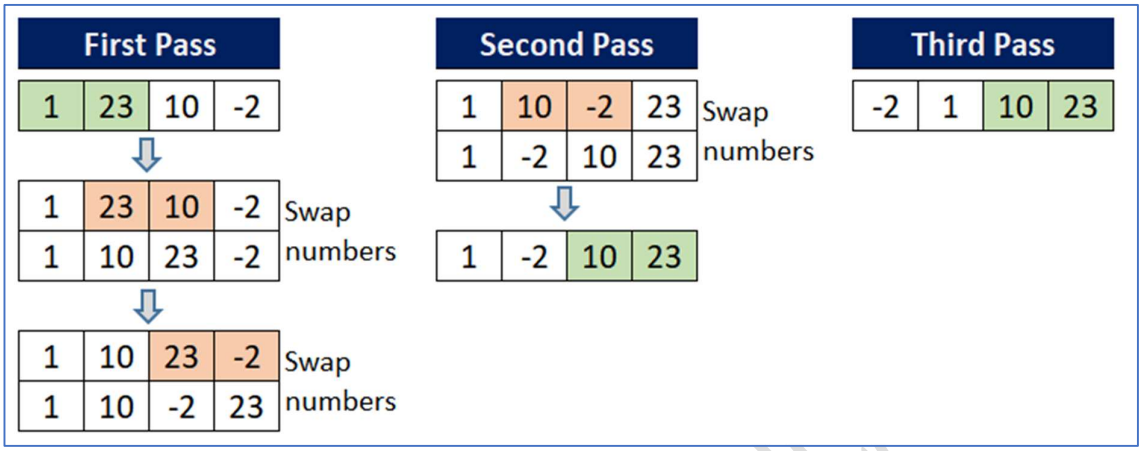
Binary Search

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

void main()
{
int binarysearch(int[],int,int);
int a[]={1,2,3,4,5,6,7,8,9,10};
int se,res;

clrscr();
printf("\n Enter the search element:");
scanf("%d",&se);
res=binarysearch(a,10,se);
if(res>=0)
printf("\n Element is found at:%d location",res);
else
printf("\n Element is not found");
}

int binarysearch(int p[],int s,int key)
{
int low=0;
int high=s;
int mid;
while(low<high)
{
  mid=(low+high)/2;
```

```
if(p[mid]==key)
return(mid+1);
else
if(key>p[mid])
low=mid+1;
else
high=mid-1;
}
return(-1);
}
```

Bubble Sort

Bubble sort is the simplest sorting algorithm and it is based on the idea that every adjacent elements are compared and swapped if found in wrong order.



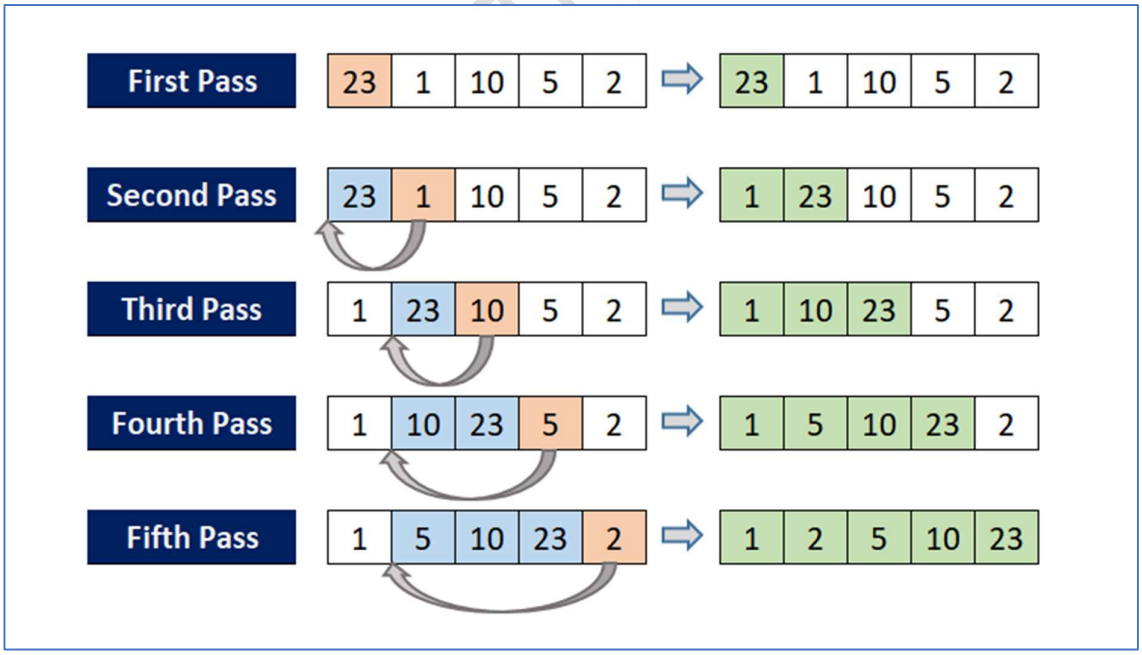
C program to perform bubble sort

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

void main()
{
int a[]={5,3,8,4,1};
int i;
void bubble_sort(int[],int);
clrscr();
printf("\n Elements in the list are:");
for(i=0;i<5;i++)
printf("\n %d",a[i]);
bubble_sort(a,5);
printf("\n Elements after sorting in the list are:");
for(i=0;i<5;i++)
printf("\n %d",a[i]);
}

void bubble_sort(int a[],int n)
{
int i,j,t,temp,k;

for(i=0;i<n;i++)
{
for(j=0;j<n-i-1;j++)
{
k=j+1;
```

```
if(a[j]>a[k])
{
temp=a[k];
a[k]=a[j];
a[j]=temp;
}
}
printf("\n Elements after %d iteration are:",i);
for(t=0;t<n;t++)
printf("%d->",a[t]);
}
}
```

Insertion Sort

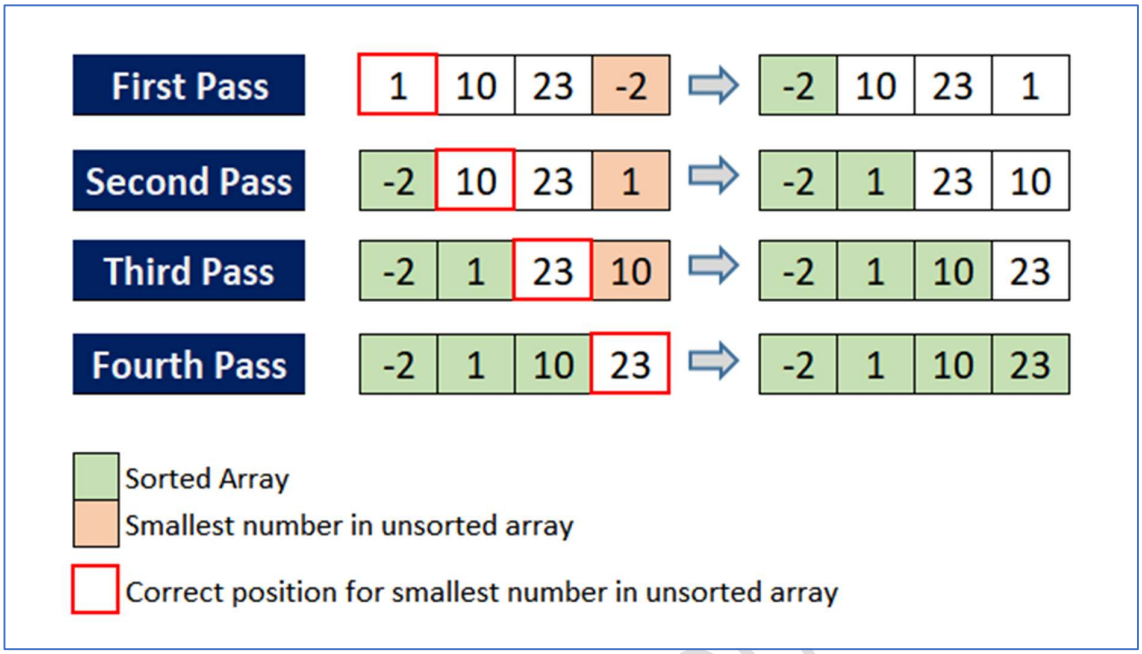
- Insertion sort is based on the idea of consuming one element from unsorted array and inserting it at the correct position in the sorted array.
- This will result into increasing the length of the sorted array by one and decreasing the length of unsorted array by one after each iteration.



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

void main()
{
int a[10]={2,5,9,8,7,1,6,3,4,10};
int i;
void insertion_sort(int[],int);
clrscr();

for(i=0;i<10;i++)
printf("%d->",a[i]);
insertion_sort(a,10);
printf("\n After sorting:");
for(i=0;i<10;i++)
printf("%d->",a[i]);
}
void insertion_sort(int a[],int size)
{
int i,j,item;
for(j=1;j<size;j++)
{
item=a[j];
i=j-1;
while(i>=0 && item<a[i])
{
a[i+1]=a[i];
i=i-1;
}
a[i+1]=item;
}

}
```

Selection Sort

- Selection sort is based on the idea of finding smallest or largest element in unsorted array and placing it at the correct position in the sorted array.
- This will result into increasing the length of the sorted array by one and decreasing the length of unsorted array by one after each iteration.

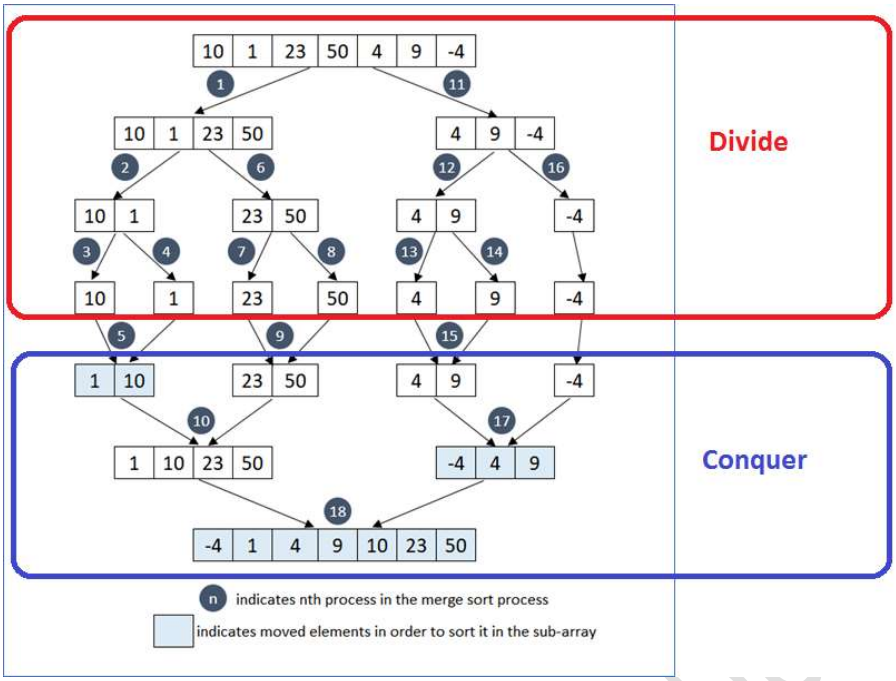



```
# include<stdio.h>
# include<conio.h>

void main()
{
int a[]={10,5,2,100,98,65,33,99,50,97};
int i;
void selection_sort(int[],int);

clrscr();
printf("\n Elements in the list are:");
for(i=0;i<10;i++)
printf("\n %d",a[i]);
selection_sort(a,10);
printf("\n Elements after sorting in the list are:");
for(i=0;i<10;i++)
printf("\n %d",a[i]);
}
void selection_sort(int a[],int n)
{
int i,index,j,large,t;
for(i=n-1;i>0;i--)
{
large=a[0];
index=0;
for(j=1;j<=i;j++)
if(a[j]>large)
```

```
{
large=a[j];
index=j;
}
a[index]=a[i];
a[i]=large;
printf("\n Elements after %d iteration",n-yi);
for(t=0;t<n;t++)
printf("%d->",a[t]);
}
}
```

Merge Sort

- Merge sort is a divide and conquer algorithm.
- It is based on the idea of dividing the unsorted array into several sub-array until each sub-array consists of a single element and merging those sub-array in such a way that results into a sorted array.
- The process step of merge sort can be summarized as follows:
    - Divide: Divide the unsorted array into several sub-array until each sub-array contains only single element.
    - Merge: Merge the sub-arrays in such way that results into sorted array and merge until achieves the original array.
    - Merging technique: the first element of the two sub-arrays is considered and compared. For ascending order sorting, the element with smaller value is taken from the sub-array and becomes a new element of the sorted array. This process is repeated until both sub-array are emptied and the merged array becomes sorted array.




```c
#include<stdio.h>
# include<conio.h>
#define MAX 50

void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);

int main(){

    int merge[MAX],i,n;
    clrscr();
    printf("Enter the total number of elements: ");
    scanf("%d",&n);

    printf("Enter the elements which to be sort: ");
    for(i=0;i<n;i++){
     scanf("%d",&merge[i]);
    }

    partition(merge,0,n-1);

    printf("\n After merge sorting elements are: ");
    for(i=0;i<n;i++){
     printf("%d ",merge[i]);
    }
```

```
   return 0;
}

void partition(int arr[],int low,int high){

    int mid;

    if(low<high){
     mid=(low+high)/2;
     printf("\n Partition on1:%d->%d",low,mid);
     partition(arr,low,mid);
     printf("\n Partition on2:%d->%d",mid+1,high);
     partition(arr,mid+1,high);
     printf("\n Partition on3:%d->%d->%d",low,mid,high);
     mergeSort(arr,low,mid,high);
    }
}

void mergeSort(int arr[],int low,int mid,int high){

    int i,m,k,l,temp[MAX];

    l=low;
    i=low;
    m=mid+1;

    while((l<=mid)&&(m<=high)){

         if(arr[l]<=arr[m]){
             temp[i]=arr[l];
             l++;
         }
         else{
             temp[i]=arr[m];
             m++;
         }
         i++;
    }

    if(l>mid){
         for(k=m;k<=high;k++){
             temp[i]=arr[k];
             i++;
         }
    }
    else{
         for(k=l;k<=mid;k++){
```

```
            temp[i]=arr[k];
            i++;
        }
    }

    for(k=low;k<=high;k++){
        arr[k]=temp[k];
    }
}
```

Quick Sort

- Quick sort is a divide and conquer algorithm.
- It is based on the idea of choosing one element as a pivot and partitioning the array around the pivot with left side of the pivot containing all elements less than the pivot and right side of the pivot containing all elements greater than the pivot.
- There are many ways to choose the pivot. Few of them are mentioned below:
    - First element of the array
    - Last element of the array
    - Random element of the array
    - Median index element of the array
- The quick sort has less space complexity as compared to merged sort because it do not use auxiliary array.

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

void swap_in_array(int*,int,int);
void main()
{
int a[]={10,9,8,7,6,5,4,3,2,1};
void quick_sort(int*,int,int);
int i;
clrscr();
printf("\n Elements in the array is:");
for(i=0;i<10;i++)
printf("%d ->",a[i]);

quick_sort(a,0,10);

printf("\n Elements in the array is:");
for(i=0;i<10;i++)
printf("%d ->",a[i]);
```

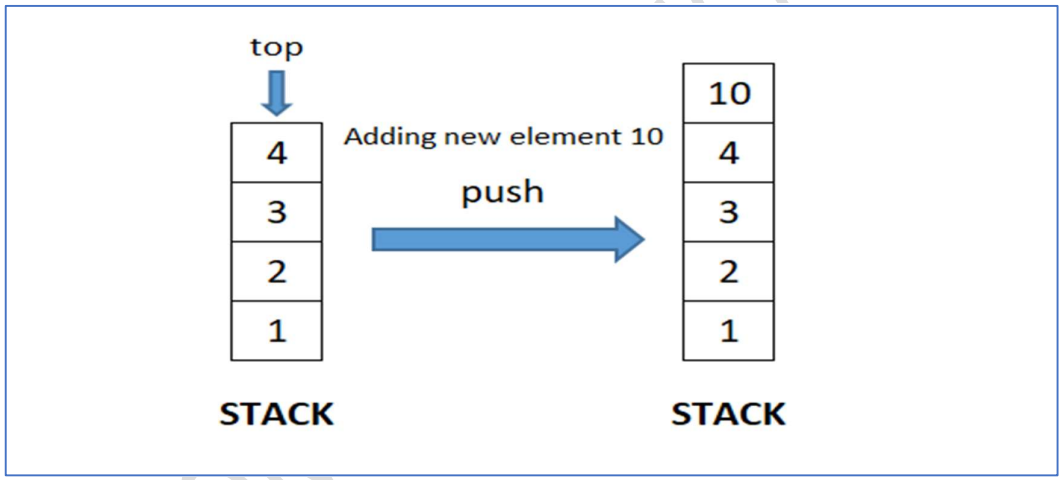
```
}

void quick_sort(int array[],int l,int r)
{
     int j,k;
     if(r<=l) return;
     j=partition(array,l,r);
     printf("\n Elements after partition1:");
    for(k=0;k<j-1;k++)
    printf("%d ->",array[k]);
    printf("\n Jth element is:%d",array[j]);
     printf("\n Elements after partition2:");
    for(k=j+1;k<r;k++)
    printf("%d ->",array[k]);
     quick_sort(array,l,j-1);
     quick_sort(array,j+1,r);
 }

 int partition(int array[], int l, int r)
 {
     int pivot,i,j;

     pivot=array[l];
     i=l+1;
     j=r;

     for(;;)
     {
     while((array[i] <= pivot) && (i <= r))i++;
     while((array[j] > pivot) && (j > l))j--;

     if(i < j)
         swap_in_array(array,i,j);
     else
         break;
     }
     swap_in_array(array,j,l);

 }

 void swap_in_array(int array[], int i, int j)
 {
     int tmp;
     tmp=array[i];
     array[i]=array[j];
     array[j]=tmp;
 }
```

Stack

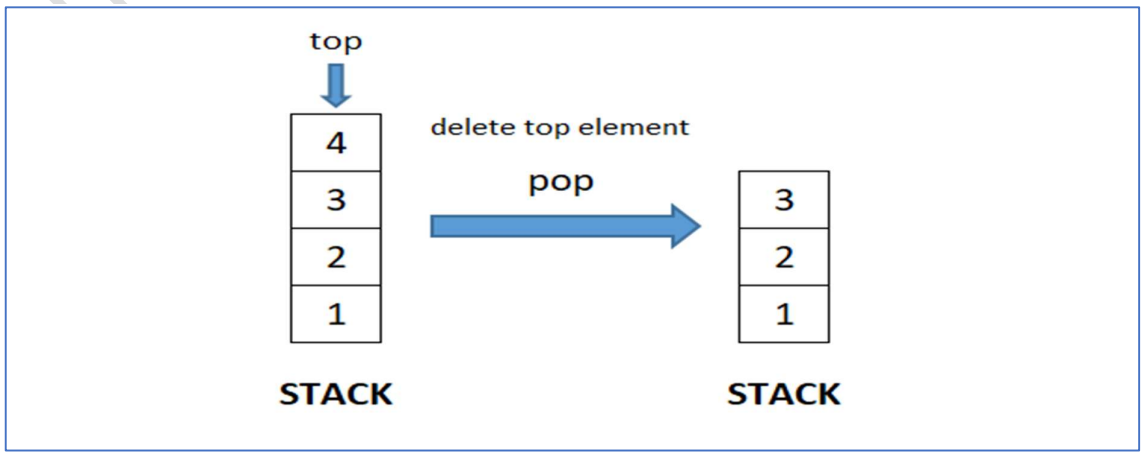
- A stack is a linear dynamic data structure that follows **Last-In/First-Out (LIFO)** principle.
- In a stack, addition of a new element and deletion of an element occurs at the same end which implies that the element which is added last in the stack will be the first to be removed from the stack.

Basic Operations of a Stack

- **isEmpty():** Checks whether the stack is empty or not.
- **size():** Returns the size of the stack.
- **topElement():** Returns the top element of the stack.
- **push(x):** Adds a new element 'x' at the top of the stack. Consequently, size of the stack increases by 1.



- **pop():** Deletes the top element of the stack. Consequently, size of the stack decreases by 1.



Implementation of Stack

```
// Stack using arrays

# include<stdio.h>
# include<conio.h>
# define max 10

// Global declarations
void push(int);
int pop();
void disp();
int st[max];
int top=-1;

void main()
{
int ele,ch;

clrscr();
clrscr();
do
{
printf("\n 1. push operation");
printf("\n 2. pop operation");
printf("\n 3. display operation");
printf("\n do u want to continue(press 0 to quit");
scanf("%d",&ch);
switch(ch)
{
case 1:
    printf("\n Enter the value:");
    scanf("%d",&ele);
    push(ele);
    break;
case 2:
    ele=pop();
    printf("\n Deleted element is:%d",ele);
    break;
case 3:
    disp();
    break;
default:
    exit(0);
}// end of switch case
}while(1);
}
```

```c
// Insert x into stack
void push(int x)
{
if(top==max-1)
{
printf("\n Stack is full");
return;
}
top++;
st[top]=x;
}
// Remove x from stack
int pop()
{
int x;
if(top==-1)
{
printf("\n Stack is empty");
return(-1);
}
x=st[top];
top--;
return(x);
}

// Display the stack list
void disp()
{
int i;
if(top==-1)
{
printf("\n Stack is empty");
return;
}
for(i=top;i>=0;i--)
printf("%d->",st[i]);
}
```

Queue

A queue is a linear dynamic data structure that follows **First-In/First-Out (FIFO)** principle. In a queue, addition of a new element and deletion of an

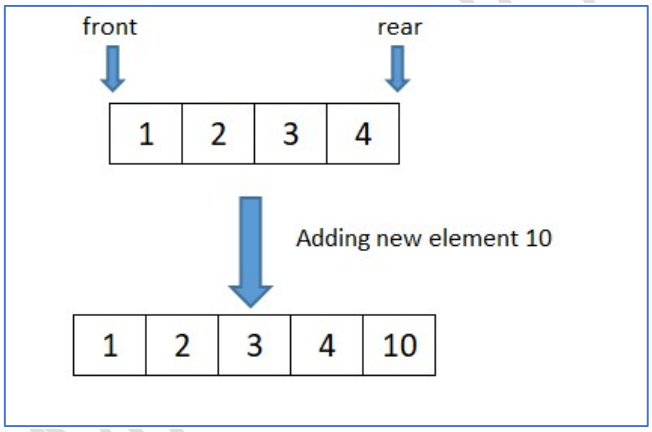
element occurs at different end which implies that the element which is added first in the queue will be the first to be removed from the queue.

Features of queue

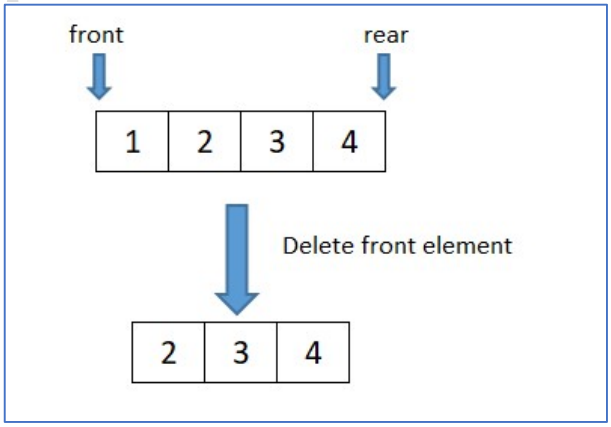
- It is a dynamic data structure.
- It has dynamic size.
- It uses dynamic memory allocation.

Basic Operations of a Queue

- **isEmpty():** Checks whether the queue is empty or not.
- **size():** Returns the size of the queue.
- **insert(x):** Adds a new element 'x' from the rear side of the queue. Consequently, size of the queue increases by 1.



- **remove():** Deletes the front element of the queue. Consequently, size of the queue decreases by 1.



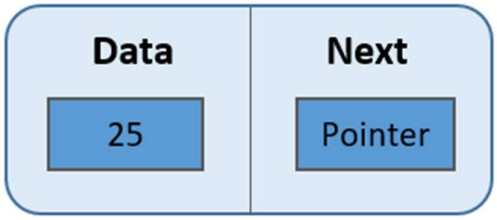
Implementation of Queue

```c
// Queue using arrays

# include<stdio.h>
# include<conio.h>
# define max 10

// Global declarations
void insert(int);
int del();
void disp();
int st[max];
int front=-1;
int rear=-1;

void main()
{
int ele,ch;

clrscr();
clrscr();
do
{
printf("\n Menu Driven for Queue using Arrays");
printf("\n 1. push operation");
printf("\n 2. pop operation");
printf("\n 3. display operation");
printf("\n do u want to continue(press 0 to quit");
scanf("%d",&ch);
switch(ch)
{
case 1:
    printf("\n Enter the value:");
    scanf("%d",&ele);
    insert(ele);
    break;
case 2:
    ele=del();
    printf("\n Deleted element is:%d",ele);
    break;
case 3:
    disp();
    break;
default:
    exit(0);
}// end of switch case
}while(1);
}
```

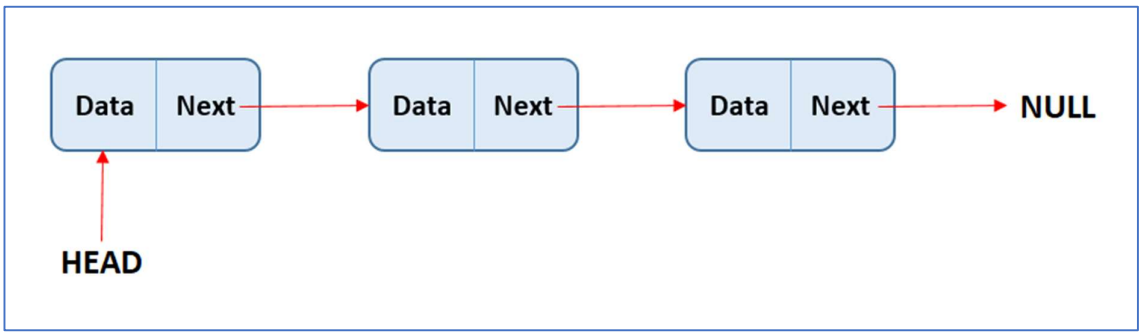
```c
// Insert x into stack
void insert(int x)
{
if(rear==max-1)
{
printf("\n Queue is full");
return;
}
rear++;
st[rear]=x;
if(front==-1)
front++;

}
// Remove x from stack
int del()
{
int x;
if(front==-1)
{
printf("\n Queue is empty");
return(-1);
}
x=st[front];
front++;
return(x);
}

// Display the stack list
void disp()
{
int i;
if(rear==-1)
{
printf("\n Stack is empty");
return;
}
for(i=front;i<=rear;i++)
printf("%d->",st[i]);
}
```

Linked List

A linked list is a linear data structure, in which the elements are stored in the form of a node. Each node contains two sub-elements. A data part that stores the value of the element and next part that stores the pointer to the next node as shown in the below image:



The first node also known as HEAD is always used as a reference to traverse the list. The last node points to NULL. Linked list can be visualized as a chain of nodes, where every node points to the next node.



Types of Linked List

The types of linked list are mentioned below:

- **Singly Linked List**: can be traversed only in forward direction.
- **Doubly Linked List**: can be traversed in forward and backward directions.
- **Circular Singly Linked List**: Last element contains link to the first element as next.
- **Circular Doubly Linked List**: Last element contains link to the first element as next and the first element contains link of the last element as previous.

How to create a node in single linked list

```
//node structure
struct Node {
  int data;
  struct Node* next;
};
```

Insert operation on Linked List

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

//node structure
struct Node {
  int data;
  struct Node* next;
};

// test the code
int main() {
  //create the head node with name MyList
  struct Node* MyList = NULL;

  //Add first node.
  struct Node* first;
  //allocate second node in the heap
  first = (struct Node*)malloc(sizeof(struct Node));
  first->data = 10;
  first->next = NULL;
  //linking with head node
  MyList = first;

  //Add second node.
  struct Node* second;
  //allocate second node in the heap
  second = (struct Node*)malloc(sizeof(struct Node));
  second->data = 20;
  second->next = NULL;
  //linking with first node
  first->next = second;

  //Add third node.
  struct Node* third;
  //allocate third node in the heap
  third = (struct Node*)malloc(sizeof(struct Node));
  third->data = 30;
```

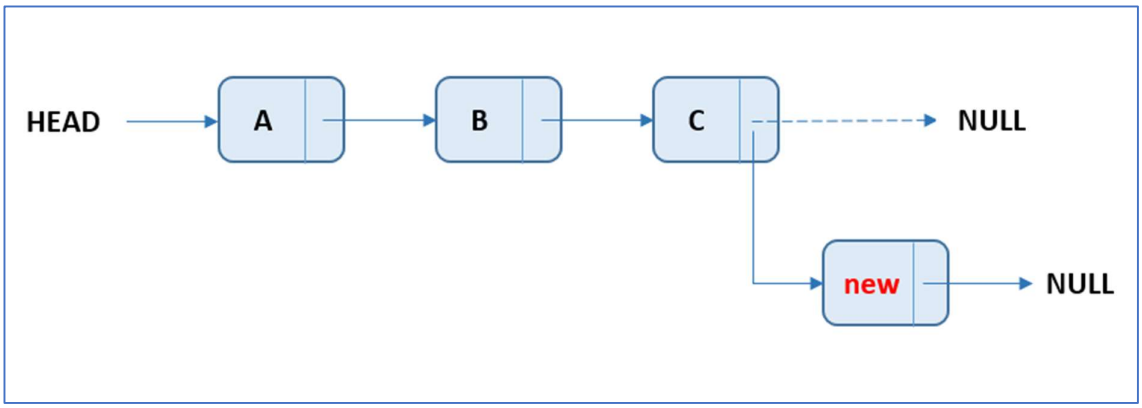
```
  third->next = NULL;
  //linking with second node
  second->next = third;

  return 0;
}
```

Display the content of list or traversal

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

//node structure
struct Node {
  int data;
  struct Node* next;
};

//display the content of the list
void PrintList(struct Node* head_ref) {
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
  //create the head node with name MyList
  struct Node* MyList = NULL;

  //Add first node.
  struct Node* first;
  //allocate second node in the heap
  first = (struct Node*)malloc(sizeof(struct Node));
  first->data = 10;
  first->next = NULL;
  //linking with head node
  MyList = first;
```

```
  //Add second node.
  struct Node* second;
  //allocate second node in the heap
  second = (struct Node*)malloc(sizeof(struct Node));
  second->data = 20;
  second->next = NULL;
  //linking with first node
  first->next = second;

  //Add third node.
  struct Node* third;
  //allocate third node in the heap
  third = (struct Node*)malloc(sizeof(struct Node));
  third->data = 30;
  third->next = NULL;
  //linking with second node
  second->next = third;

  //print the content of list
  PrintList(MyList);
  return 0;
}
```

Insert a new node at the end of the list

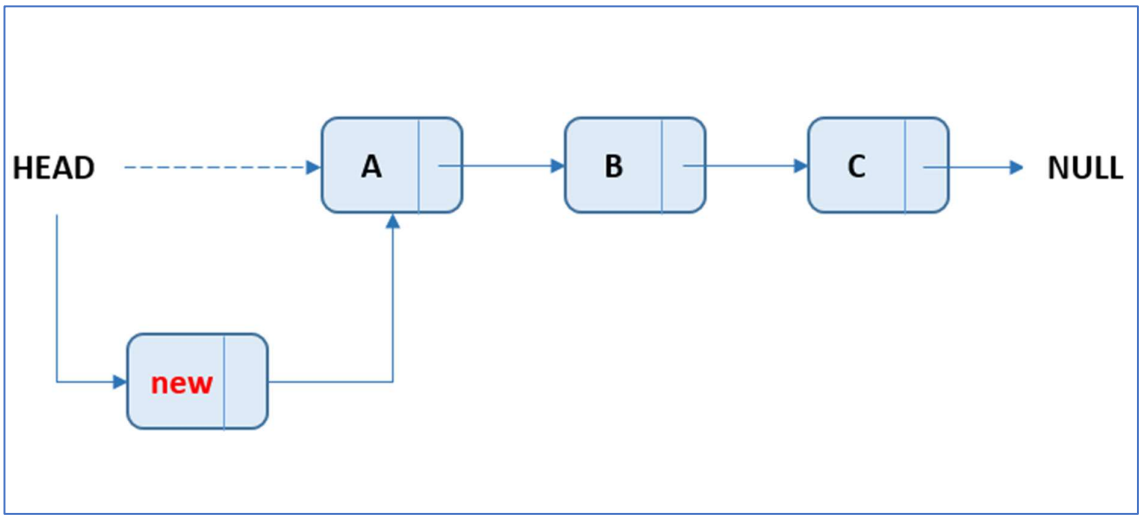



```
void push_back(struct Node** head_ref, int newElement) {

  //1. allocate node
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
  //2. assign data element
  newNode->data = newElement;

  //3. assign null to the next of new node
  newNode->next = NULL;

  //4. Check the Linked List is empty or not,
  //   if empty make the new node as head
  if(*head_ref == NULL) {
    *head_ref = newNode;
  } else {

    //5. Else, traverse to the last node
    temp = *head_ref;
    while(temp->next != NULL) {
      temp = temp->next;
    }

    //6. Change the next of last node to new node
    temp->next = newNode;
  }
}
#include <stdio.h>
#include <stdlib.h>

//node structure
struct Node {
  int data;
  struct Node* next;
};

//Add new element at the end of the list
void push_back(struct Node** head_ref, int newElement) {
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = NULL;
  if(*head_ref == NULL) {
    *head_ref = newNode;
  } else {
    temp = *head_ref;
    while(temp->next != NULL) {
      temp = temp->next;
    }
    temp->next = newNode;
  }
}
```

```
//display the content of the list
void PrintList(struct Node* head_ref) {
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
  struct Node* MyList = NULL;

  //Add three elements at the end of the list.
  push_back(&MyList, 10);
  push_back(&MyList, 20);
  push_back(&MyList, 30);
  PrintList(MyList);

  return 0;
}
```

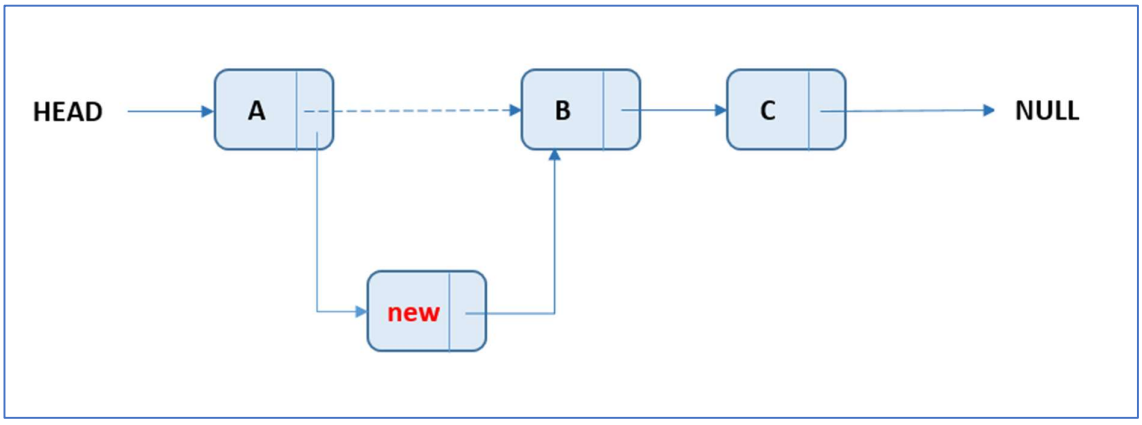
Insert a node at the beginning of the list




```
void push_front(struct Node** head_ref, int newElement) {

  //1. allocate a new node
  struct Node* newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));

  //2. assign data element
  newNode->data = newElement;

  //3. make next node of new node as head
  newNode->next = *head_ref;

  //4. make new node as head
  *head_ref = newNode;
}
#include <stdio.h>
#include <stdlib.h>

//node structure
struct Node {
  int data;
  struct Node* next;
};

//Add new element at the start of the list
void push_front(struct Node** head_ref, int newElement) {
  struct Node* newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = *head_ref;
  *head_ref = newNode;
}
```

```
//display the content of the list
void PrintList(struct Node* head_ref) {
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
  struct Node* MyList = NULL;

  //Add three elements at the start of the list.
  push_front(&MyList, 10);
  push_front(&MyList, 20);
  push_front(&MyList, 30);
  PrintList(MyList);

  return 0;
}
```

Insert a new node at the specified position



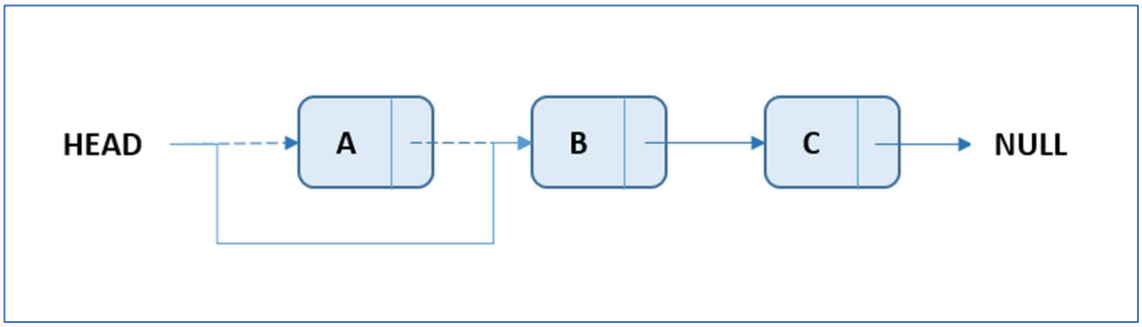

```
void push_at(struct Node** head_ref, int newElement, int position) {

  //1. allocate node to new element
```

```
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = NULL;

  //2. check if the position is > 0
  if(position < 1) {
    printf("\nposition should be >= 1.");
  } else if (position == 1) {

    //3. if the position is 1, make next of the
    //   new node as head and new node as head
    newNode->next = *head_ref;
    *head_ref = newNode;
  } else {

    //4. Else, make a temp node and traverse to the
    //   node previous to the position
    temp = *head_ref;
    for(int i = 1; i < position-1; i++) {
      if(temp != NULL) {
        temp = temp->next;
      }
    }

    //5. If the previous node is not null, make
    //   newNode next as temp next and temp next
    //   as newNode.
    if(temp != NULL) {
      newNode->next = temp->next;
      temp->next = newNode;
    } else {

      //6. When the previous node is null
      printf("\nThe previous node is null.");
    }
  }
}
#include <stdio.h>
#include <stdlib.h>

//node structure
struct Node {
  int data;
  struct Node* next;
};

//Add new element at the end of the list
```

```
void push_back(struct Node** head_ref, int newElement) {
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = NULL;
  if(*head_ref == NULL) {
    *head_ref = newNode;
  } else {
    temp = *head_ref;
    while(temp->next != NULL) {
      temp = temp->next;
    }
    temp->next = newNode;
  }
}

//Inserts a new element at the given position
void push_at(struct Node** head_ref, int newElement, int position) {
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = NULL;

  if(position < 1) {
    printf("\nposition should be >= 1.");
  } else if (position == 1) {
    newNode->next = *head_ref;
    *head_ref = newNode;
  } else {

    temp = *head_ref;
    for(int i = 1; i < position-1; i++) {
      if(temp != NULL) {
        temp = temp->next;
      }
    }

    if(temp != NULL) {
      newNode->next = temp->next;
      temp->next = newNode;
    } else {
      printf("\nThe previous node is null.");
    }
  }
}

//display the content of the list
void PrintList(struct Node* head_ref) {
```

```
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
  struct Node* MyList = NULL;

  //Add three elements in the list.
  push_back(&MyList, 10);
  push_back(&MyList, 20);
  push_back(&MyList, 30);
  PrintList(MyList);

  //Insert an element at position 2
  push_at(&MyList, 100, 2);
  PrintList(MyList);

  //Insert an element at position 1
  push_at(&MyList, 200, 1);
  PrintList(MyList);

  return 0;
}
```

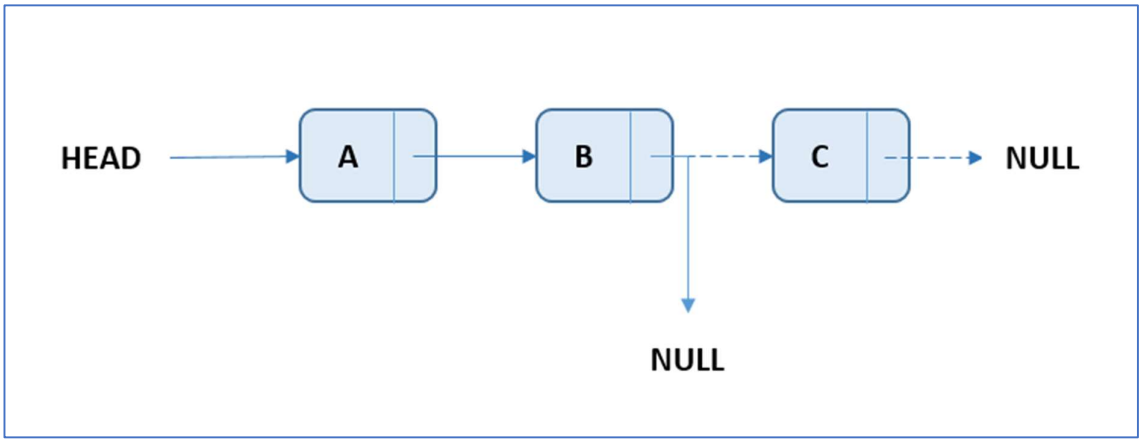
Delete the First Node of the list



```
void pop_front(struct Node** head_ref) {
  if(*head_ref != NULL) {

    //1. if head is not null, create a
    //    temp node pointing to head
    struct Node *temp = *head_ref;

    //2. move head to next of head
    *head_ref = (*head_ref)->next;

    //3. delete temp node
    temp = NULL;
  }
}
#include <stdio.h>
#include <stdlib.h>

//node structure
struct Node {
  int data;
  struct Node* next;
};

//Add new element at the end of the list
void push_back(struct Node** n, int newElement) {
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = NULL;
  if(*n == NULL) {
    *n = newNode;
  } else {
    temp = *n;
    while(temp->next != NULL) {
      temp = temp->next;
    }
    temp->next = newNode;
  }
}

//Delete first node of the list
void pop_front(struct Node** head_ref) {
  if(*head_ref != NULL) {
    struct Node *temp = *head_ref;
    *head_ref = (*head_ref)->next;
    temp = NULL;
  }
}
```

```
//display the content of the list
void PrintList(struct Node* head_ref) {
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
  struct Node* MyList = NULL;

  //Add four elements in the list.
  push_back(&MyList, 10);
  push_back(&MyList, 20);
  push_back(&MyList, 30);
  push_back(&MyList, 40);
  PrintList(MyList);

  //Delete the first node
  pop_front(&MyList);
  PrintList(MyList);
  return 0;
}
```

Delete the Last Node of the list

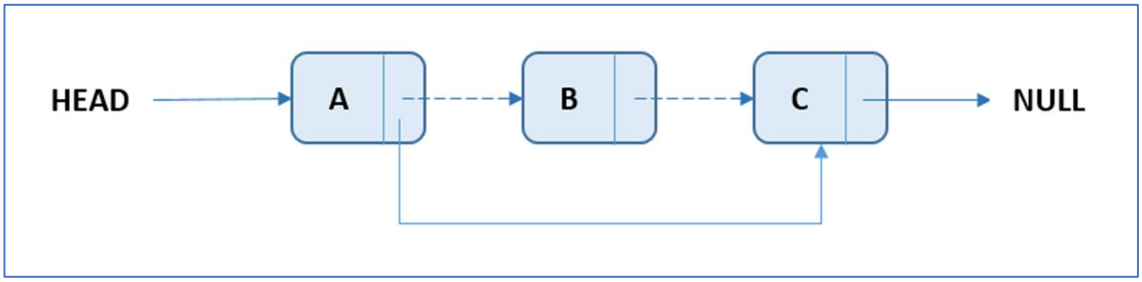



```
void pop_back(struct Node** head_ref) {
  if(*head_ref != NULL) {

    //1. if head in not null and next of head
    //   is null, release the head
    if((*head_ref)->next == NULL) {
      *head_ref = NULL;
    } else {

      //2. Else, traverse to the second last
      //   element of the list
      struct Node* temp = *head_ref;
      while(temp->next->next != NULL)
        temp = temp->next;

      //3. Change the next of the second
      //   last node to null and delete the
      //   last node
      struct Node* lastNode = temp->next;
      temp->next = NULL;
      free(lastNode);
    }
  }
}
#include <stdio.h>
#include <stdlib.h>

//node structure
struct Node {
  int data;
  struct Node* next;
};

//Add new element at the end of the list
void push_back(struct Node** n, int newElement) {
```

```
  struct Node *newNode, *temp;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = newElement;
  newNode->next = NULL;
  if(*n == NULL) {
    *n = newNode;
  } else {
    temp = *n;
    while(temp->next != NULL) {
      temp = temp->next;
    }
    temp->next = newNode;
  }
}

//Delete last node of the list
void pop_back(struct Node** head_ref) {
  if(*head_ref != NULL) {
    if((*head_ref)->next == NULL) {
      *head_ref = NULL;
    } else {
      struct Node* temp = *head_ref;
      while(temp->next->next != NULL)
        temp = temp->next;
      struct Node* lastNode = temp->next;
      temp->next = NULL;
      free(lastNode);
    }
  }
}

//display the content of the list
void PrintList(struct Node* head_ref) {
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
```

```
  struct Node* MyList = NULL;

  //Add four elements in the list.
  push_back(&MyList, 10);
  push_back(&MyList, 20);
  push_back(&MyList, 30);
  push_back(&MyList, 40);
  PrintList(MyList);

  //Delete the last node
  pop_back(&MyList);
  PrintList(MyList);
  return 0;
}
```

Delete the node from the given position




```
void pop_at(struct Node** head_ref, int position) {

  //1. check if the position is > 0
  if(position < 1) {
    printf("\nposition should be >= 1.");
  } else if (position == 1 && *head_ref != NULL) {

    //2. if the position is 1 and head is not null, make
    //   head next as head and delete previous head
    struct Node* nodeToDelete = *head_ref;
    *head_ref = (*head_ref)->next;
    free(nodeToDelete);
  } else {

    //3. Else, make a temp node and traverse to the
    //   node previous to the position
    struct Node *temp;
    temp = *head_ref;
    for(int i = 1; i < position-1; i++) {
      if(temp != NULL) {
```

```
      temp = temp->next;
    }
  }

  //4. If the previous node and next of the previous
  //   is not null, adjust links
  if(temp != NULL && temp->next != NULL) {
    struct Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    free(nodeToDelete);
  } else {

    //5. Else the given node will be empty.
    printf("\nThe node is already null.");
  }
 }
}
#include <stdio.h>
#include <stdlib.h>

//node structure
struct Node {
 int data;
 struct Node* next;
};

//Add new element at the end of the list
void push_back(struct Node** head_ref, int newElement) {
 struct Node *newNode, *temp;
 newNode = (struct Node*)malloc(sizeof(struct Node));
 newNode->data = newElement;
 newNode->next = NULL;
 if(*head_ref == NULL) {
  *head_ref = newNode;
 } else {
  temp = *head_ref;
  while(temp->next != NULL) {
   temp = temp->next;
  }
  temp->next = newNode;
 }
}

//Delete an element at the given position
void pop_at(struct Node** head_ref, int position) {
 if(position < 1) {
  printf("\nposition should be >= 1.");
 } else if (position == 1 && *head_ref != NULL) {
```

```
    struct Node* nodeToDelete = *head_ref;
    *head_ref = (*head_ref)->next;
    free(nodeToDelete);
  } else {
    struct Node *temp;
    temp = *head_ref;
    for(int i = 1; i < position-1; i++) {
      if(temp != NULL) {
        temp = temp->next;
      }
    }
    if(temp != NULL && temp->next != NULL) {
      struct Node* nodeToDelete = temp->next;
      temp->next = temp->next->next;
      free(nodeToDelete);
    } else {
      printf("\nThe node is already null.");
    }
  }
}

//display the content of the list
void PrintList(struct Node* head_ref) {
  struct Node* temp = head_ref;
  if(head_ref != NULL) {
    printf("The list contains: ");
    while (temp != NULL) {
      printf("%i ",temp->data);
      temp = temp->next;
    }
    printf("\n");
  } else {
    printf("The list is empty.\n");
  }
}

// test the code
int main() {
  struct Node* MyList = NULL;

  //Add three elements at the end of the list.
  push_back(&MyList, 10);
  push_back(&MyList, 20);
  push_back(&MyList, 30);
  PrintList(MyList);

  //Delete an element at position 2
  pop_at(&MyList, 2);
```

```
  PrintList(MyList);

  //Delete an element at position 1
  pop_at(&MyList, 1);
  PrintList(MyList);

  return 0;
}
```

Count the nodes

```
int countNodes(struct Node* head_ref) {

  //1. create a temp node pointing to head
  struct Node* temp = head_ref;

  //2. create a variable to count nodes
  int i = 0;

  //3. if the temp node is not null increase
  //   i by 1 and move to the next node, repeat
  //   the process till the temp becomes null
  while (temp != NULL) {
    i++;
    temp = temp->next;
  }

  //4. return the count
  return i;
}
```

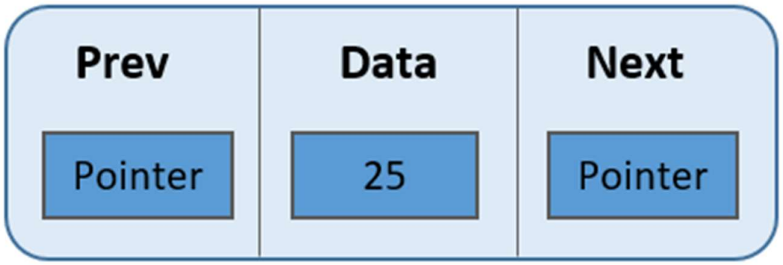
```
# include<stdio.h>
# include<conio.h>


// Program on Linked List
// Node defination
struct node
{
int data;
struct node *next;
```

```
};

struct node *fir=NULL;

void main()
{
struct node *t;
void insertlast(int);
void insertfirst(int);
void insertposition(int,int);
void display();
void delet(int);
clrscr();
insertlast(10);
insertlast(20);
display();
insertfirst(30);
display();
insertposition(5,3);
display();
delet(20);
display();
}

void insertlast(int d)
{
struct node *temp=(struct node*)malloc(sizeof(struct node)),*t1=fir;
temp->data=d;
temp->next=NULL;
if(t1==NULL)
fir=temp;
else
{
while(t1->next!=NULL)
t1=t1->next;
t1->next=temp;
}
}
void insertfirst(int d)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=d;
temp->next=fir;
fir=temp;
}
void insertposition(int d,int p)
{
```

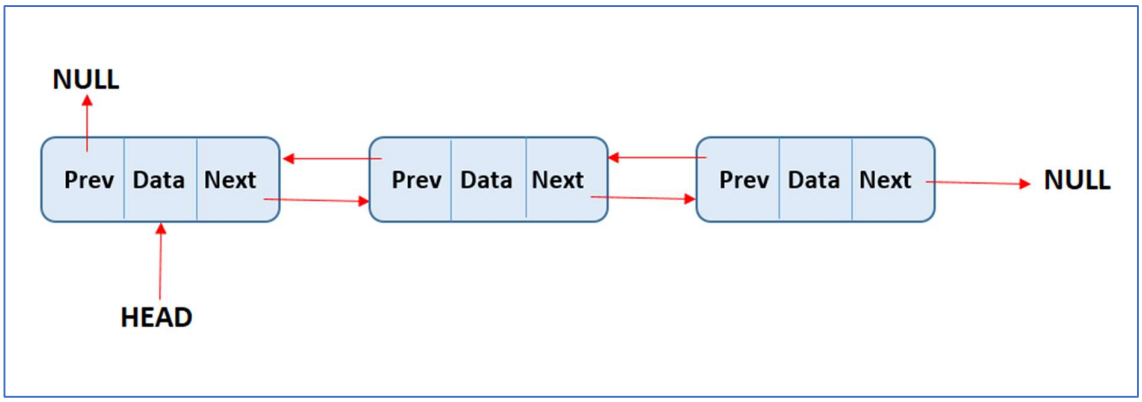
```
struct node *temp=(struct node*)malloc(sizeof(struct node));
int count=1;
struct node *t1,*t2;
temp->data=d;
temp->next=NULL;

t1=fir;
t2=fir->next;
while(count!=p)
{
t1=t1->next;
t2=t2->next;
count++;
}
t1->next=temp;
temp->next=t2;
}

void display()
{
struct node *t;
printf("\n");
for(t=fir;t!=NULL;t=t->next)
printf("%d->",t->data);
}
void delet(int k)
{
struct node *t=fir,*t1=fir,*t2=fir->next;

while(t->data!=k)
{
if(t==t1)
{
t=t->next;
t2=t2->next;
}
else
{
t1=t1->next;
t2=t2->next;
t=t->next;
}
}
t1->next=t2;
}
```

Doubly Linked List

A doubly linked list is a linear data structure, in which the elements are stored in the form of a node. Each node contains three sub-elements. A data part that stores the value of the element, the previous part that stores the pointer to the previous node, and the next part that stores the pointer to the next node as shown in the below image:



The first node also known as HEAD is always used as a reference to traverse the list. The previous of head node and next of last node points to NULL. A doubly linked list can be visualized as a chain of nodes, where every node points to previous and next node.



Implementation of Doubly Linked List

How to create a new Node?

```
//node structure
struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
```

```
};
```

```
# include<stdio.h>

struct dllnode{
    struct sllnode *prev;
    int data;
    struct sllnode *next;
};

/* Head is the pointer variable always refers
the starting node */
struct sllnode *head=NULL;

void main(){
    void insertingNodeEmptySLL(int);
    void insertingNodeNonEmptySLL(int);
    void displaySLLValues();
    /*create a temp node of type sllnode */
    insertingNodeEmptySLL(10);
    insertingNodeNonEmptySLL(20);
    insertingNodeNonEmptySLL(30);
    insertingNodeNonEmptySLL(40);
    //void displaySLLValues();

}

/* to insert a first node in the empty linked list*/
void insertingNodeEmptySLL(int value){
    struct sllnode *temp = (struct sllnode*)malloc(struct sllnode);

    /* insert element */
    temp->data=value;

    /* mark the next address value as null */
    temp->next=NULL;
    temp->prev=NULL;

    /* placing the temp node as a first node
    in the linked list */
    head=temp;

}

/* to insert the values in second and subsequent
```

```
nodes */
void insertingNodeNonEmptySLL(int value){

    struct dllnode *temp;

    /* insert element */
    temp->data=value;

    /* mark the next address value as null */
    temp->next=NULL;
    temp->prev=NULL;

    /* move the control to the last node of the
    linkedlist */

    struct dllnode *ctrl = head;
    while(ctrl->next!=NULL){
        ctrl=ctrl->next;
    }

    /* copy the address of tem to ctrl->next */
    ctrl->next=temp;
    temp->prev=ctrl;


}

void displayDLLValues(){

    /* ctrl is pointing first node of the list*/
    struct dllnode *ctrl = head;
    printf("\n Values are.......");
    /* the below condition is checking
    whether ctrl reached to the end of list or not*/
    while(ctrl!=NULL){
        /* fetching the data and displaying */
        printf("%5d --> ",ctrl->data);
        /*moving the control to the next node*/
        ctrl=ctrl->next;
    }

void searchForAnElement(int p){

    /* ctrl is pointing first node of the list*/
    struct dllnode *ctrl = head;
    int isFound=0;
    /* the below condition is checking
```

```
    whether ctrl reached to the end of list or not*/
    while(ctrl!=NULL){
    if(ctrl->data==p){
        isFound=1;
        break;
    }

    /*moving the control to the next node*/
        ctrl=ctrl->next;
    }

    if(isFound==0)
    printf("%d is Not found",p);
    else
    printf("%d is found",p);
}
```