CMR College of Engineering & Technology
Department of CSE
DATA STRUCTURES & ALGORITHMS LAB
(Common to ECE, CSE, EEE & IT)

Week	Name of the Program
1	A) Write a C program to perform the following operations on the given array (i) Insert element in specific position in to array (ii) Delete random element from array (iii) Reverse the array elements
2	A) Write a C program to implement Single linked list (i) Insertion (ii) Deletion (iii)Display B) Write a C program to implement Circular linked list (i) Insertion (ii) Deletion. (iii)Display
3	A) Write a C program to implement Doubly linked list (i) Insertion (ii) Deletion. (iii)Display B) Write C programs to implement Stack ADT using (i) Array (ii) Linked List
4	A) Write a C program that uses stack operations to convert a given infix expression in to its postfix equivalent. (Display the role of stack). B) Write a C program for Evaluation of postfix expression.
5	A) Write C programs to implement Queue ADT using (i) Array (ii) Linked List
6	Write a C program to implement Binary search tree (i) Insertion (ii) deletion (iii)Traversals
7	Write a C program to implement binary search tree Non - recursively traversals (i) Pre- Order (ii) Post –Order (iii)In-Order
8	(A) Write a C Program to Check if a Given Binary Tree is an AVL Tree or Not (B) Write a C program to find height of a Binary tree (C) Write a C program to count the number of leaf nodes in a tree.
9	Write a C program for implementing Graph traversal (i) DFS (ii) BFS
10	A) Write a C program to implement different hash methods B) Write a C program to implement the following collision resolving (i) Quadratic probing. (ii) Linear Probing

11	Write C programs for implementing the following Sorting methods and display the important steps. (i) Quick Sort (ii) Heap sort
12	Write a C program for implementing pattern matching algorithms (i) Knuth-Morris-Pratt (ii) Brute Force

Reference Books:

1. Ellis Horowitz, Sartaj Sahni, Fundamentals of Data Structures in C, Second Edition Universities Press.
2. Thomas H. Cormen Charles E. Leiserson, Introduction to Algorithms, PHI Learning Pvt. Ltd. Third edition.
3. Algorithms, Data Structures, and Problem Solving with C++", Illustrated Edition by Mark Allen Weiss, Addison-Wesley Publishing Company
4. E.Balagurusamy Data Structures Using C, McGraw Hill Education; First edition

Q1: Write a C program to perform the following operations on the given array

(i) Insert element in specific position in to array

(ii) Delete random element from array

(iii) Reverse the array elements

An array is a collection of items stored at contiguous memory locations.

Algorithm:
Step 1: Start
Step 2: Read number of elements
Step 3: Read Array of elements
Step 4: enter your choice to insert/ delete/ reverse the Given array
Step 5: Stop

Program

```
#include <stdio.h> #include
 <stdlib.h>
 void Insert(int [],int*,int,int ); void Delet(int
 [],int*,int); void Rever(int [],int);
 int main()
 {
 int a[10];
 int ch,pos,len,opt; int ele,n,i;
 printf("Enter the size of array\n"); scanf("%d",&n);
 printf("Enter the elements\n"); for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
do
{
printf("1:Insert element in specific position in to array\n"); printf("2:Delete
random element from array\n"); printf("3.Reverse the array elements\n");
printf("Enter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("Enter position\n");
scanf("%d",&pos);
printf("Enter Element to insert\n");
scanf("%d",&ele); Insert(a,&n,pos,ele);
break; case 2:
printf("Enter any position ");
scanf("%d",&pos);
Delet(a,&n,pos); break;
case 3:
Rever(a,n); break; default:
printf("Invalid option\n");
}
```

```
printf("\n Do you want to continue\n"); printf("\n press
1:YES 2:No\n"); scanf("%d",&opt);
}while(opt!=2); return 0;
}

void Insert(int a[],int *n,int p,int e)
{
int i,j; j=*n-1;
while(j>=(p-1))
{
a[j+1]=a[j]; j--;
}
a[p-1]=e;
(*n)++;
printf("Resultant array is\n");
for(i=0;i<(*n);i++)
printf("%d ",a[i]);
}
void Delet(int a[],int *n,int p)
{
int i,j; j=p;
while(j<=*n)
{
a[j-1]=a[j]; j++;
}
(*n)--;
for(i=0;i<*n;i++) printf("%d",a[i]);
}

void Rever(int a[],int l)
{
int i;
for(i=l-1;i>=0;i--)
{
        printf("%d ",a[i]);
}
}
```

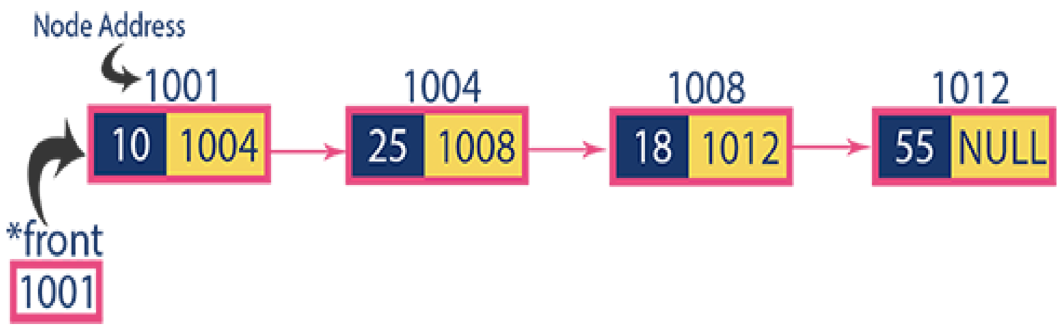
Q2. Write a C program to implement Single linked list

Linked List is a linear data structure in which every data item is represented as Node containing two or more slots.

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 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 node in a single linked list is as follows...

Example



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 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 **'start'** 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.

Insertion

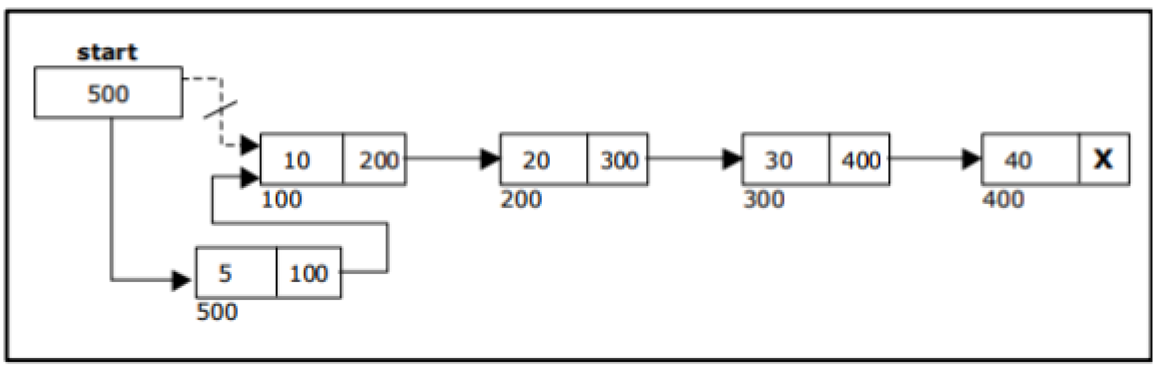
In a single linked list, the insertion operation can be performed in three ways. They are as follows...

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...

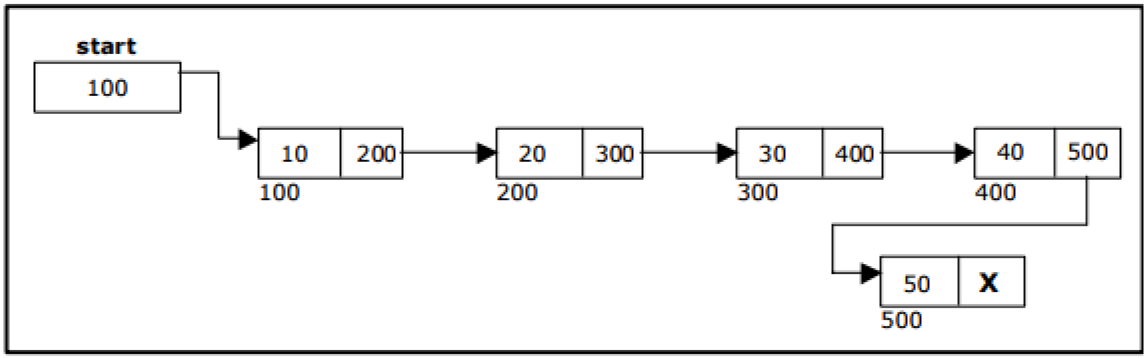
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **start** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **start** and **start**= **newNode**.



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** (**start** = = **NULL**).
- **Step 3 -** If it is **Empty** then, set **start** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is not equal to **NULL**).
- Step 6 - Set temp → next = newNode.



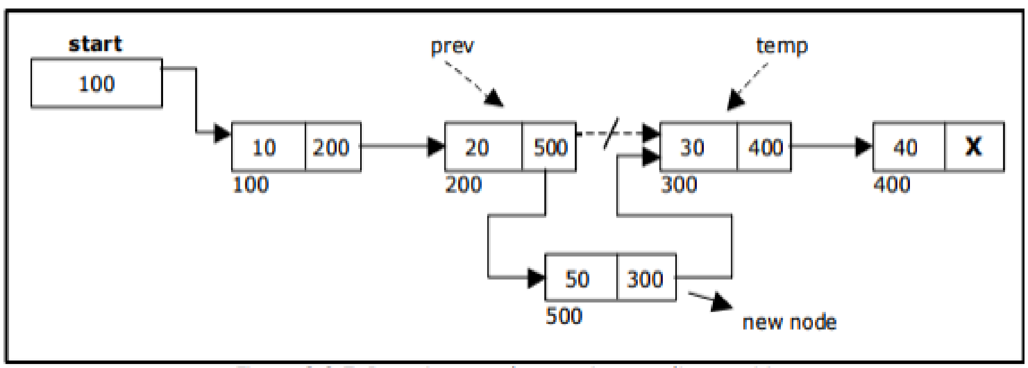
Inserting at middle of the list

Step 1 - Create a newNode with given value

Step 2 - Check whether list is **Empty** (**start** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode

Step 6-Finally, Set p->next=newnode, newnode->next=temp



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

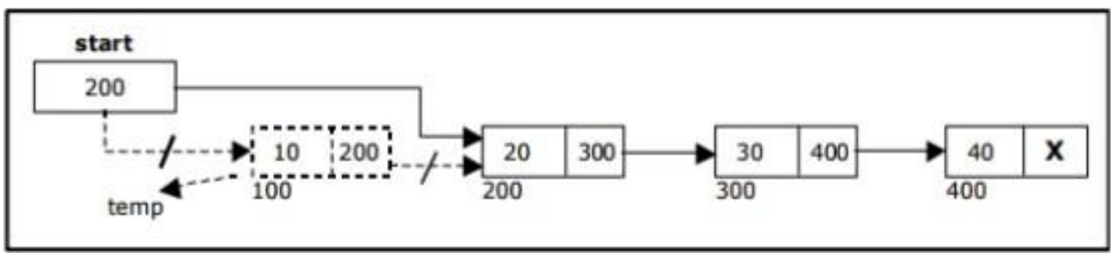
We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is **Empty** (**start** == **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 **start**. **Step 4 -** Check whether list is having only one node (**temp →** **next** == **NULL**)
Step 5 - If it is **TRUE** then set **start**= **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **start**= **temp → next**, and delete **temp(free(temp))**.



Deleting from End of the list

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

Step 1 - Check whether list is **Empty** (**start** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

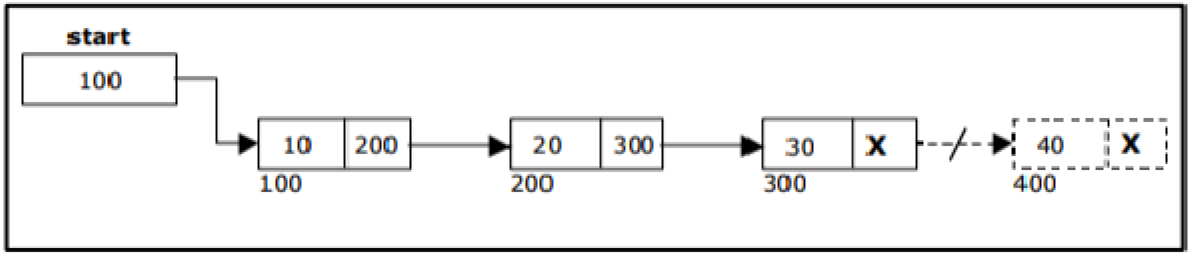
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp'** and '**temp1'** and initialize '**temp**' with **start**.

Step 4 - Check whether list has only one Node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **start** = **NULL** and delete **temp**. And terminate the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp1 = temp** ' and move **temp** 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 **temp1 → next** = **NULL** and delete **temp(free(temp))**.



Deleting a Specific Node from the list

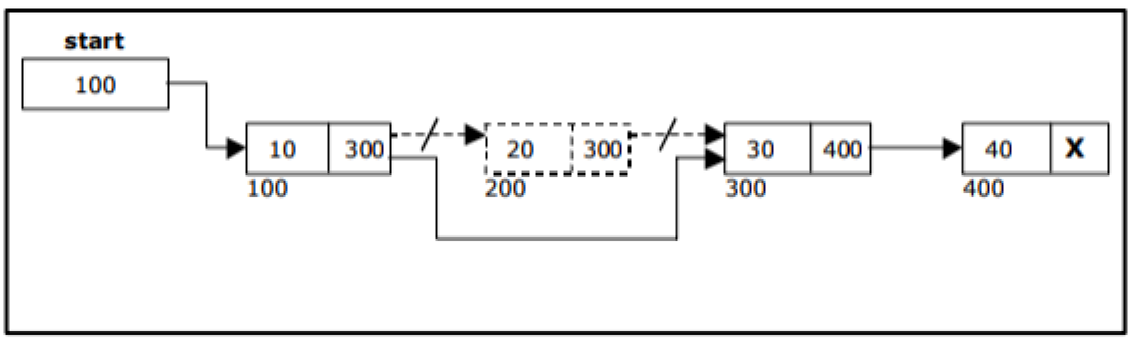
Step 1 - Create a newNode with given value

Step 2 - Check whether list is **Empty** (**start** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to delete the Node

Step 6-Finally, Set p->next=temp->next

Step 7-delete temp(free(temp)).



Traversing

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.
- The function traverse () is used for traversing and displaying the information stored in the list from left to right.

Program to implement Single linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 9)
   {
      printf("\n\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
      5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         beginsert();
         break;
         case 2:
         lastinsert();
         break;
         case 3:
         randominsert();
         break;
         case 4:
         begin_delete();
         break;
         case 5:
         last_delete();
         break;
         case 6:
         random_delete();
         break;
```

```c
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            ptr -> next = NULL;
```

```c
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }

        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
void begin_delete()
{
```

```
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ...\n");
    }
}
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }

    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ...\n");
    }
}

void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
```

```c
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}
```

```c
void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("Nothing to print");
    }
    else
    {
        printf("\nprinting values . . . . .\n");
        while (ptr!=NULL)
        {
            printf("\n%d",ptr->data);
            ptr = ptr -> next;
        }
    }
}
```

Output:

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
1

Node inserted

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

Choose one option from the following list ...
```

```
===============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
2

Enter value?
2

Node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
3

Enter element value1

Enter the location after which you want to insert 1

Node inserted

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

Choose one option from the following list ...
```

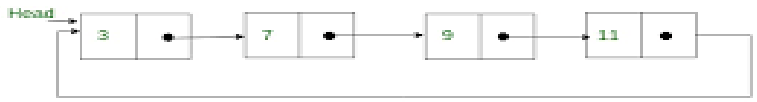
```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
8

printing values . . . . .

1
2
1

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
2

Enter value?
123
Node inserted
```

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
1234

Node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
4

Node deleted from the begining ...

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

Choose one option from the following list ...

===============================================
```

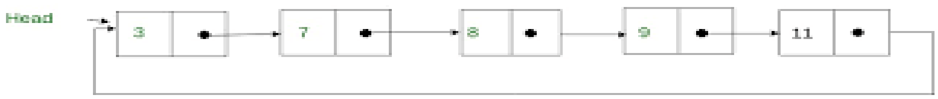
```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
5

Deleted Node from the last ...

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
6

Enter the location of the node after which you want to perform deletion
1

Deleted node 2
```

2B) Write a C program to implement Circular linked list
i) Insertion ii) Deletion. iii)Display

Circular Singly linked list is similar to Single linked list, but the last node of the list contains a pointer to the first node of the list.
We can have circular singly linked list as well as circular doubly linked list.



After inserting 8, the above CLL should be changed to the following




```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
    int choice =0;
    while(choice != 7)
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete from last\n5.Search for an element\n6.Show\n7.Exit\n");
        printf("\nEnter your choice?\n");
```

```c
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            begin_delete();
            break;
            case 4:
            last_delete();
            break;
            case 5:
            search();
            break;
            case 6:
            display();
            break;
            case 7:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
```

```
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
```

```
            ptr -> next = head;
        }

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

}

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

    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

    }
}
void last_delete()
{
    struct node *ptr, *preptr;
    if(head==NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if (head ->next == head)
    {
        head = NULL;
        free(head);
```

```c
        printf("\nnode deleted\n");

    }
    else
    {
        ptr = head;
        while(ptr ->next != head)
        {
            preptr=ptr;
            ptr = ptr->next;
        }
        preptr->next = ptr -> next;
        free(ptr);
        printf("\nnode deleted\n");

    }
}

void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        if(head ->data == item)
        {
        printf("item found at location %d",i+1);
        flag=0;
        }
        else
        {
        while (ptr->next != head)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
```

```c
            break;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      }
      if(flag != 0)
      {
         printf("Item not found\n");
      }
   }

}

void display()
{
   struct node *ptr;
   ptr=head;
   if(head == NULL)
   {
      printf("\nnothing to print");
   }
   else
   {
      printf("\n printing values ... \n");

      while(ptr -> next != head)
      {

         printf("%d\n", ptr -> data);
         ptr = ptr -> next;
      }
      printf("%d\n", ptr -> data);
   }

}
```

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

Choose one option from the following list ...

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

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

Enter your choice?
1

Enter the node data?10

node inserted

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

Choose one option from the following list ...

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

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

Enter your choice?
2

Enter Data?20

node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Delete from Beginning
```

4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
2

Enter Data?30

node inserted

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

Choose one option from the following list ...

===

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

Enter your choice?
3
node deleted

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

Choose one option from the following list ...

===

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

Enter your choice?
4

node deleted

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

Choose one option from the following list ...

===

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

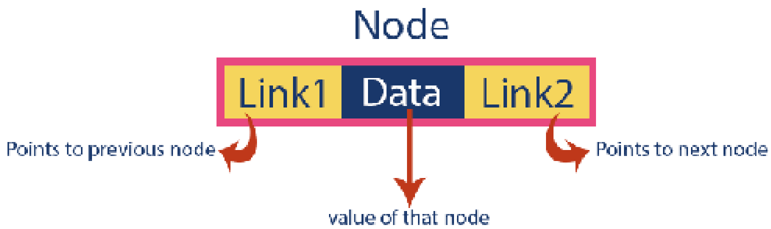
Enter your choice?
5

Enter item which you want to search?
20
item found at location 1

Q3. A) Write a C program to implement Doubly linked list
i) Insertion ii) Deletion iii) Display

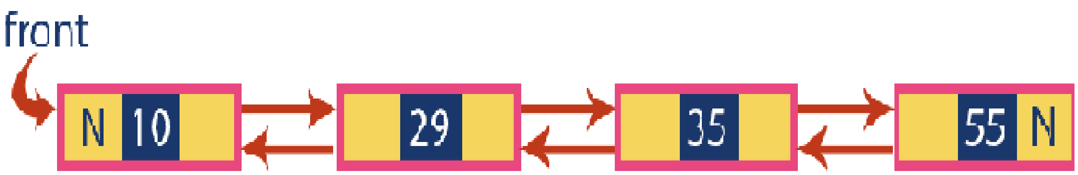
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

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.

Example



Operations on Double Linked List

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

1. Insertion
2. Deletion
3. Display

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** (**start** = = **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **start**.
- **Step 4 -** If it is **not Empty** then, assign **start** to **newNode → next** and **newNode** to **start**.

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** (**start = = NULL**)
- **Step3-** 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**

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

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

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

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

4. Deleting from Beginning of the list
5. Deleting from End of the list
6. 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** (**start == 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 **start**.

Step 4 - Check whether list is having only one node (**temp → next ==**

NULL)
Step 5 - If it is **TRUE** then set **start**= **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **start**= **temp → next**, and delete **temp(free(temp))**.

Deleting from End of the list

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

Step 1 - Check whether list is **Empty** (**start** == **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 **'temp'** and **'temp1'** and initialize '**temp**' with **start**.

Step 4 - Check whether list has only one Node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **start** = **NULL** and delete **temp**. And terminate the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp1 = temp** ' and move **temp** 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 **temp1 → next** = **NULL** and delete **temp(free(temp))**.

Deleting a Specific Node from the list

Step 1 - Create a newNode with given value

Step 2 - Check whether list is **Empty** (**start** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode → next** = **NULL** and **start** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to delete the Node

Step 6-Finally, Set p->next=temp->next
temp->next->prev=p

Step 7-delete temp(free(temp)).

Traversing

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.
- The function traverse () is used for traversing and displaying the information stored in the list from left to right.

Program to implement doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
   while(choice != 9)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete
from Beginning\n
      5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n")
;
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         insertion_beginning();
         break;
         case 2:
               insertion_last();
         break;
         case 3:
         insertion_specified();
         break;
         case 4:
         deletion_beginning();
         break;
         case 5:
         deletion_last();
         break;
         case 6:
         deletion_specified();
```

```c
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insertion_beginning()
{
   struct node *ptr;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
       printf("\nOVERFLOW");
   }
   else
   {
    printf("\nEnter Item value");
    scanf("%d",&item);

   if(head==NULL)
   {
       ptr->next = NULL;
       ptr->prev=NULL;
       ptr->data=item;
       head=ptr;
   }
   else
   {
       ptr->data=item;
       ptr->prev=NULL;
       ptr->next = head;
       head->prev=ptr;
       head=ptr;
   }
   printf("\nNode inserted\n");
}

}
```

```
void insertion_last()
{
  struct node *ptr,*temp;
  int item;
  ptr = (struct node *) malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
     printf("\nEnter value");
     scanf("%d",&item);
      ptr->data=item;
     if(head == NULL)
     {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
     }
     else
     {
       temp = head;
       while(temp->next!=NULL)
       {
          temp = temp->next;
       }
       temp->next = ptr;
       ptr ->prev=temp;
       ptr->next = NULL;
       }

     }
    printf("\nnode inserted\n");
   }
void insertion_specified()
{
  struct node *ptr,*temp;
  int item,loc,i;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
  }
  else
  {
     temp=head;
     printf("Enter the location");
     scanf("%d",&loc);
     for(i=0;i<loc;i++)
     {
```

```c
        temp = temp->next;
        if(temp == NULL)
        {
            printf("\n There are less than %d elements", loc);
            return;
        }
    }
    printf("Enter value");
    scanf("%d",&item);
    ptr->data = item;
    ptr->next = temp->next;
    ptr -> prev = temp;
    temp->next = ptr;
    temp->next->prev=ptr;
    printf("\nnode inserted\n");
  }
}
void deletion_beginning()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        head = head -> next;
        head -> prev = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }

}
void deletion_last()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
```

```
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }
}
void deletion_specified()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
```

```c
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }

}
```

Output

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
```

```
8.Show
9.Exit

Enter your choice?
8

 printing values...

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value12

Node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value123
```

```
Node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value1234

Node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
1234
123
12

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

```
Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
2

Enter value89

node inserted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
3
Enter the location1
Enter value12345

node inserted

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

Choose one option from the following list ...

===============================================
```

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
1234
123
12345
12
89

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
4

node deleted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
```

```
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
5

node deleted

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
123
12345

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
```

```
6

 Enter the data after which the node is to be deleted : 123

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
123

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

Choose one option from the following list ...

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

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
7

Enter item which you want to search?
123

item found at location 1
```

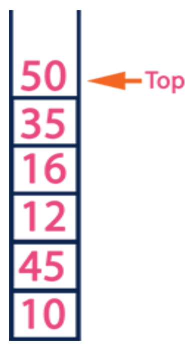
3B. i) Write C programs to implement Stack ADT using Array

Stack is a linear data structure.

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

Example

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



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

Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array stores only 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 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.

- **Step1 -** 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.

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**).

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- -**).

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 3 -** Repeat above step until **i** value becomes '0'.

Program to implement Stack using Array

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
```

```c
switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
```

```
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Stack Implementation Using Single Linked List

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;

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

```c
            break;
        }
        case 3:
        {
            display();
            break;
        }
        case 4:
        {
            printf("Exiting....");
            break;
        }
        default:
        {
            printf("Please Enter valid choice ");
        }
    };
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");
```

```
    }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```

Lab 4:

A. Write a C program that uses stack operations to convert a given infix expression in to its postfix equivalent. (Display the role of stack).

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 popped and print each popped symbol to the result.

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

Example

Consider the following Infix Expression...

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define SIZE 50
char s[SIZE];
int top=-1;
push(char elem)
{
    s[++top]=elem;
}

char pop()
{
    return(s[top--]);
}

int pr(char elem)
{
    switch(elem)
    {
    case '#': return 0;
    case '(': return 1;
    case '+':
    case '-': return 2;
    case '*':
    case '/': return 3;
    }
}

void main()
{
    char infx[50],pofx[50],ch,elem;
    int i=0,k=0;
    clrscr();
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infx);
    push('#');
      while( (ch=infx[i++]) != '\0')
    {
    if( ch == '(') push(ch);
    else
    if(isalnum(ch)) pofx[k++]=ch;
      else
    if( ch == ')')
    {
       while( s[top] != '(')
     pofx[k++]=pop();
       elem=pop();
    }
```

```
  else
  {
     while( pr(s[top]) >= pr(ch) )
    pofx[k++]=pop();
     push(ch);
  }
  }
  while( s[top] != '#')
 pofx[k++]=pop();
  pofx[k]='\0';
  printf("\n\nGiven Infix Expn: %s  Postfix Expn: %s\n",infx,pofx);
  getch();
}
```

OUTPUT:

Enter the Infix Expression = A*(B+c)/D

Given Infix Expn: A*(B+c)/D   Postfix Expn: ABC+*D/

4B. Write a C program for Evaluation of postfix expression.

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

Example

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.

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

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

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

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			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)
		48	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

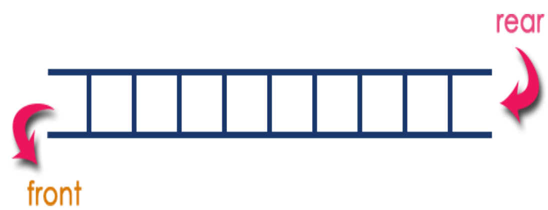
Postfix Expression 5 3 + 8 2 - * value is 48

```c
#include<stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define SIZE 40
int pop();
void push(int);
char postfix[SIZE];
int stack[SIZE], top = -1;
int main()
{
	int i, a, b, result, pEval;
	char ch;

	for(i=0; i<SIZE; i++)
	{
		stack[i] = -1;
	}
	printf("\nEnter a postfix expression: ");
	scanf("%s",postfix);
	for(i=0; postfix[i] != '\0'; i++)
	{
		ch = postfix[i];
		if(isdigit(ch))
		{
			push(ch-'0');
		}
		else if(ch == '+' || ch == '-' || ch == '*' || ch == '/')
		{
			b = pop();
			a = pop();

			switch(ch)
			{
				case '+': result = a+b;
					break;
				case '-': result = a-b;
					break;
				case '*': result = a*b;
					break;
				case '/': result = a/b;
					break;
			}

			push(result);
		}

	}
	pEval = pop();

	printf("\nThe postfix evaluation is: %d\n",pEval);
```

```
		return 0;
	}
	void push(int n)
	{
		if (top < SIZE -1)
		{
			stack[++top] = n;
		}
		else
		{
			printf("Stack is full!\n");
			exit(-1);
		}
	}
	int pop()
	{
		int n;
		if (top > -1)
		{
			n = stack[top];
			stack[top--] = -1;
			return n;
		}
		else
		{
			printf("Stack is empty!\n");
			exit(-1);
		}
	}
```

Lab 5: Write C programs to implement Queue ADT using
i) Array ii) Linked List

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 of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the

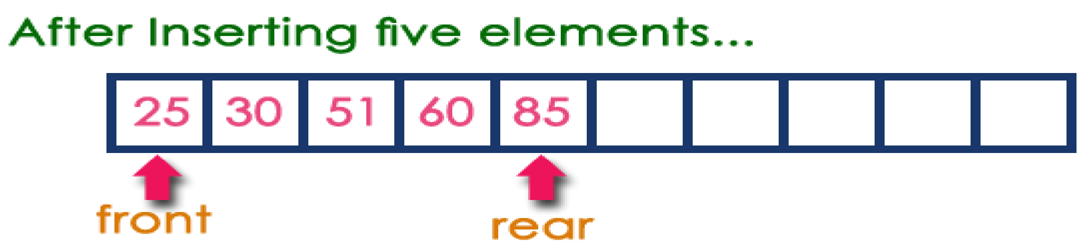


insertion operation is performed at a position which is known as '**rear**' and

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

Queue Data structure 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.

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 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**.

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**).

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**)

Program to implement Queue using Array

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
   int choice;
   while(choice != 4)
   {
      printf("\n*************************Main Menu*****************************\n");
      printf("\n===========================================================
==\n");
      printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
      printf("\nEnter your choice ?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(0);
         break;
         default:
         printf("\nEnter valid choice??\n");
```

```c
        }
    }
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");

}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;

    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
```

```
        front = front + 1;
      }
      printf("\nvalue deleted ");
   }


}



void display()
{
   int i;
   if(rear == -1)
   {
      printf("\nEmpty queue\n");
   }
   else
   {   printf("\nprinting values .....\n");
      for(i=front;i<=rear;i++)
      {
         printf("\n%d\n",queue[i]);
      }
   }
}
```

Output:

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

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

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
123

Value inserted

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

==============================================
```

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

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

==

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

************************Main Menu****************************
==

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3
printing values .....

90
************************Main Menu****************************
==

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4

5.ii) Implementation of Queue using Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
   int choice;
   while(choice != 4)
   {
      printf("\n***********Main Menu*** ************\n");
      printf("\n===========================\n");
      printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
      printf("\nEnter your choice ?");
      scanf("%d",& choice);
      switch(choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
```

```c
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}



void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

```
void delete ()
{
   struct node *ptr;
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
   }
}
void display()
{
   struct node *ptr;
   ptr = front;
   if(front == NULL)
   {
      printf("\nEmpty queue\n");
   }
   else
   {   printf("\nprinting values .....\n");
      while(ptr != NULL)
      {
         printf("\n%d\n",ptr -> data);
         ptr = ptr -> next;
      }
   }
}
```

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

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
123
```

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

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
90

***********Main Menu*** ************
==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

123

90

***********Main Menu*** ************
==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit
Enter your choice ?2

***********Main Menu*** ************
==============================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90
```

program to implement Binary search tree

i) Insertion ii) deletion iii) Traversals

Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

The following operations are performed on a binary Search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)**time complexity.

The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left sub tree.
- **Step 6:** If search element is larger, then continue the search process in right sub tree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity.

In binary search tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows...

- **Step 1:** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is **Empty**, then set set **root** to **newNode**.
- **Step 4:** If the tree is **Not Empty**, then check whether value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5:** If newNode is **smaller** than **or equal** to the node, then move to its **left** child. If newNode is **larger** than the node, then move to its **right** child.
- **Step 6:** Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).
- **Step 7:** After reaching a leaf node, then isert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree has follwing three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: **Deleting a leaf node**

We use the following steps to delete a leaf node from BST...

- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: **Deleting a node with one child**

We use the following steps to delete a node with one child from BST...

- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node using **free** function and

terminate the function.

Case 3: **Deleting a node with two children**

We use the following steps to delete a node with two children from BST...

- **Step 1:** **Find** the node to be deleted using **search operation**
- **Step 2:** If it has two children, then find the **largest** node in its **left sub tree** (OR) the **smallest** node in its **right sub tree**.
- **Step 3:** **Swap** both **deleting node** and node which found in above step.
- **Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.

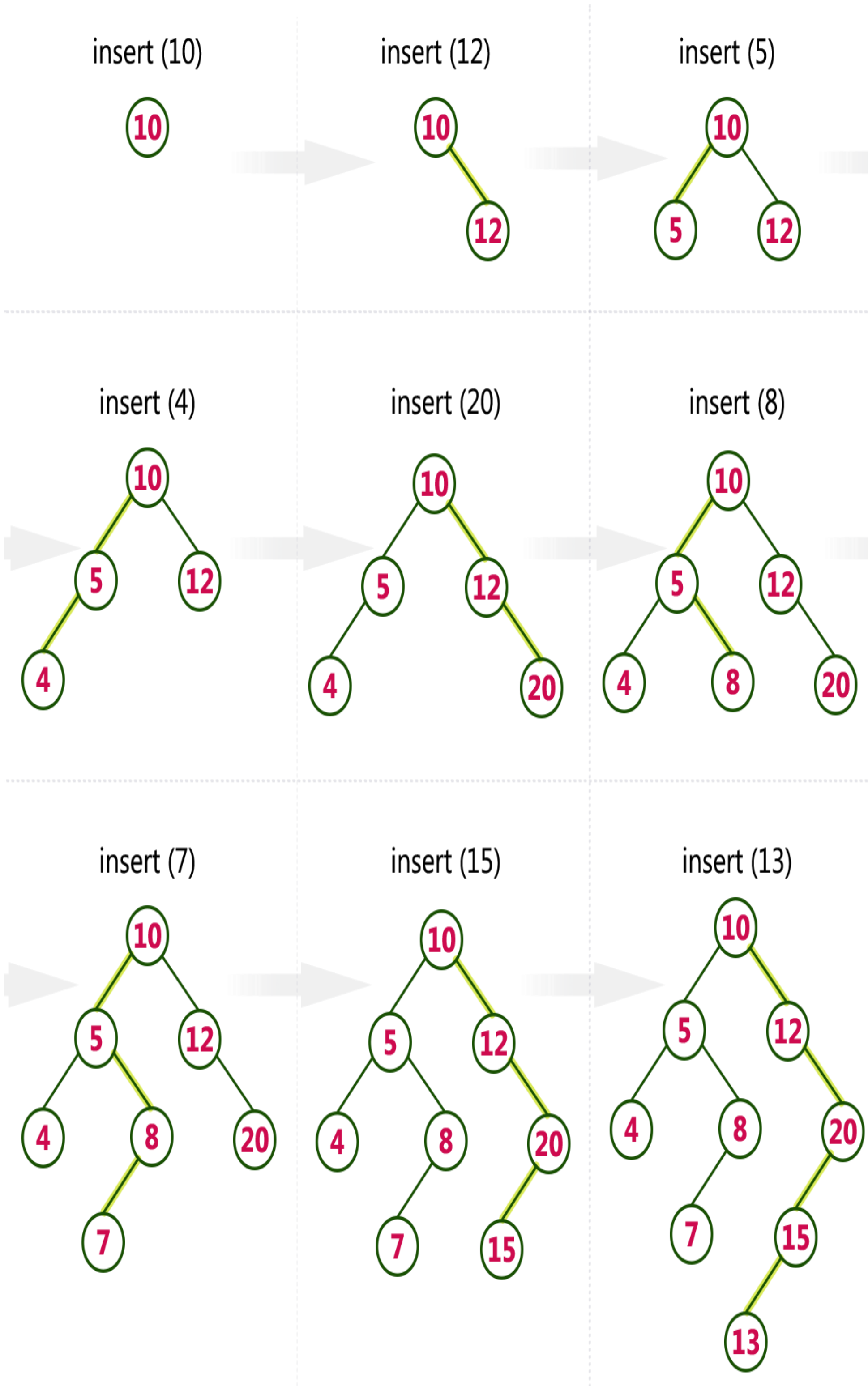
Tree Traversals:

- In **Pre-order** the parent node visited first then the left child node and at last the right child node.
- In **In-order** the left child node visited first then parent node and at last the right child node.
- In **Post-order** the left child node visited first then the right child node and in last parent node.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13


insert (10)
insert (12)
insert (5)
insert (4)
insert (20)
insert (8)
insert (7)
insert (15)
insert (13)

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

struct node
{
		int info;
		struct node *lchild;
		struct node *rchild;
}*root;



void find(int item,struct node **par,struct node **loc)
{
		struct node *ptr,*ptrsave;

		if(root==NULL)  /*tree empty*/
		{
				*loc=NULL;
				*par=NULL;
				return;
		}
		if(item==root->info) /*item is at root*/
		{
				*loc=root;
				*par=NULL;
				return;
		}
		/*Initialize ptr and ptrsave*/
		if(item<root->info)
				ptr=root->lchild;
		else
				ptr=root->rchild;
		ptrsave=root;

		while(ptr!=NULL)
		{
				if(item==ptr->info)
				{	*loc=ptr;
						*par=ptrsave;
						return;
				}
				ptrsave=ptr;
				if(item<ptr->info)
						ptr=ptr->lchild;
				else
						ptr=ptr->rchild;
		 }/*End of while */
		 *loc=NULL;   /*item not found*/
		 *par=ptrsave;
```

```
}/*End of find()*/

void insert(int item)
{       struct node *tmp,*parent,*location;
        find(item,&parent,&location);
        if(location!=NULL)
        {
                printf("Item already present");
                return;
        }

        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=item;
        tmp->lchild=NULL;
        tmp->rchild=NULL;

        if(parent==NULL)
                root=tmp;
        else
                if(item<parent->info)
                        parent->lchild=tmp;
                else
                        parent->rchild=tmp;
}/*End of insert()*/


void case_a(struct node *par,struct node *loc )
{
        if(par==NULL) /*item to be deleted is root node*/
                root=NULL;
        else
                if(loc==par->lchild)
                        par->lchild=NULL;
                else
                        par->rchild=NULL;
}/*End of case_a()*/

void case_b(struct node *par,struct node *loc)
{
        struct node *child;

        /*Initialize child*/
        if(loc->lchild!=NULL) /*item to be deleted has lchild */
                child=loc->lchild;
        else            /*item to be deleted has rchild */
                child=loc->rchild;

        if(par==NULL )   /*Item to be deleted is root node*/
                root=child;
        else
                if( loc==par->lchild)   /*item is lchild of its parent*/
```

```
                        par->lchild=child;
                else            /*item is rchild of its parent*/
                        par->rchild=child;
}/*End of case_b()*/

void case_c(struct node *par,struct node *loc)
{
        struct node *ptr,*ptrsave,*suc,*parsuc;

        /*Find inorder successor and its parent*/
        ptrsave=loc;
        ptr=loc->rchild;
        while(ptr->lchild!=NULL)
        {
                ptrsave=ptr;
                ptr=ptr->lchild;
        }
        suc=ptr;
        parsuc=ptrsave;

        if(suc->lchild==NULL && suc->rchild==NULL)
                case_a(parsuc,suc);
        else
                case_b(parsuc,suc);

        if(par==NULL) /*if item to be deleted is root node */
                root=suc;
        else
                if(loc==par->lchild)
                        par->lchild=suc;
                else
                        par->rchild=suc;

        suc->lchild=loc->lchild;
        suc->rchild=loc->rchild;
}/*End of case_c()*/
int del(int item)
{
        struct node *parent,*location;
        if(root==NULL)
        {
                printf("Tree empty");
                return 0;
        }

        find(item,&parent,&location);
        if(location==NULL)
        {
                printf("Item not present in tree");
                return 0;
        }
```

```
		if(location->lchild==NULL && location->rchild==NULL)
			case_a(parent,location);
		if(location->lchild!=NULL && location->rchild==NULL)
			case_b(parent,location);
		if(location->lchild==NULL && location->rchild!=NULL)
			case_b(parent,location);
		if(location->lchild!=NULL && location->rchild!=NULL)
			case_c(parent,location);
		free(location);
}/*End of del()*/

int preorder(struct node *ptr)
{
		if(root==NULL)
		{
			printf("Tree is empty");
			return 0;
		}
		if(ptr!=NULL)
		{
			printf("%d  ",ptr->info);
			preorder(ptr->lchild);
			preorder(ptr->rchild);
		}
}/*End of preorder()*/

void inorder(struct node *ptr)
{
		if(root==NULL)
		{
			printf("Tree is empty");
			return;
		}
		if(ptr!=NULL)
		{
			inorder(ptr->lchild);
			printf("%d  ",ptr->info);
			inorder(ptr->rchild);
		}
}/*End of inorder()*/

void postorder(struct node *ptr)
{
		if(root==NULL)
		{
			printf("Tree is empty");
			return;
		}
		if(ptr!=NULL)
		{
```

```
                    postorder(ptr->lchild);
                    postorder(ptr->rchild);
                    printf("%d  ",ptr->info);
          }
}/*End of postorder()*/

void display(struct node *ptr,int level)
{
          int i;
          if ( ptr!=NULL )
          {
                    display(ptr->rchild, level+1);
                    printf("\n");
                    for (i = 0; i < level; i++)
                              printf("    ");
                    printf("%d", ptr->info);
                    display(ptr->lchild, level+1);
          }/*End of if*/
}/*End of display()*/
main()
{
          int choice,num;
          root=NULL;
          while(1)
          {
                    printf("\n");
                    printf("1.Insert\n");
                    printf("2.Delete\n");
                    printf("3.Inorder Traversal\n");
                    printf("4.Preorder Traversal\n");
                    printf("5.Postorder Traversal\n");
                    printf("6.Display\n");
                    printf("7.Quit\n");
                    printf("Enter your choice : ");
                    scanf("%d",&choice);

                    switch(choice)
                    {
                     case 1:
                              printf("Enter the number to be inserted : ");
                              scanf("%d",&num);
                              insert(num);
                              break;
                     case 2:
                              printf("Enter the number to be deleted : ");
                              scanf("%d",&num);
                              del(num);
                              break;
                     case 3:
                              inorder(root);
                              break;
```

```
                    case 4:
                            preorder(root);
                            break;
                    case 5:
                            postorder(root);
                            break;
                    case 6:
                            display(root,1);
                            break;
                    case 7:
          break;
                    default:
                            printf("Wrong choice\n");
                    }/*End of switch */
            }/*End of while */
}/*End of main()*/
```

8A. Write a C Program to Check if a Given Binary Tree is an AVL Tree or Not

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 50
```

```
struct node
{
	struct node *lchild;
	int info;
	struct node *rchild;
};
struct node *insert(struct node *ptr, int ikey);
int height(struct node *ptr);
int isAVL(struct node *ptr);
struct node *insert(struct node *ptr, int ikey );
int main( )
{
	struct node *root=NULL,*copy_root=NULL, *root1=NULL;
	root = insert(root, 6);
	root = insert(root, 3);
	root = insert(root, 8);
	root = insert(root, 7);
	root = insert(root, 1);
	root = insert(root, 4);
	root = insert(root, 9);
	root = insert(root, 10);
	root = insert(root, 11);
	if( isAVL(root) )
		printf("AVL\n");
	else
		printf("Not AVL\n");
}/*End of main( )*/
int isAVL(struct node *ptr)
{
	int h_l, h_r, diff;
	if(ptr == NULL)
		return 1;
	h_l = height(ptr->lchild);
	h_r = height(ptr->rchild);
	diff = h_l>h_r ? h_l-h_r : h_r-h_l;
	if( diff<=1 &&  isAVL(ptr->lchild) && isAVL(ptr->rchild) )
		return 1;
	return 0;
}


int height(struct node *ptr)
{
	int h_left, h_right;

	if (ptr == NULL) /*Base Case*/
		return 0;
	 h_left =  height(ptr->lchild);
	h_right = height(ptr->rchild);
```

```
        if (h_left > h_right)
                return 1 + h_left;
        else
                return 1 + h_right;
}/*End of height()*/
struct node *insert(struct node *ptr, int ikey )
{
        if(ptr==NULL)
        {
                ptr = (struct node *) malloc(sizeof(struct node));
                ptr->info = ikey;
                ptr->lchild = NULL;
                ptr->rchild = NULL;
        }
        else if(ikey < ptr->info) /*Insertion in left subtree*/
                ptr->lchild = insert(ptr->lchild, ikey);
        else if(ikey > ptr->info) /*Insertion in right subtree */
                ptr->rchild = insert(ptr->rchild, ikey);
        else
                printf("Duplicate key\n");
        return(ptr);
}/*End of insert( )*/
```

8(B) Write a C program to find height of a Binary tree

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

```
int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Height of tree is %d", maxDepth(root));

    getchar();
    return 0;
}
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int maxDepth(struct node* node)
{
   if (node==NULL)
       return 0;
   else
   {
       /* compute the depth of each subtree */
       int lDepth = maxDepth(node->left);
       int rDepth = maxDepth(node->right);

       /* use the larger one */
       if (lDepth > rDepth)
           return(lDepth+1);
       else return(rDepth+1);
   }
}




/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
```

```
                            malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}
```

8(C) Write a C program to count the number of leaf nodes in a tree

```
#incude<stdio.h>
#include<stdlib.h>
/* A binary tree node has data,
pointer to left child and
a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
 struct node* newNode(int) ;
unsigned int getLeafCount(struct node* );

int main()
{
    /*create a tree*/
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

/*get leaf count of the above created tree*/
cout << "Leaf count of the tree is : "<<
                getLeafCount(root) << endl;
return 0;
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                    malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
```

```
return(node);
}

  /* Function to get the count
of leaf nodes in a binary tree*/
unsigned int getLeafCount(struct node* node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
    else
        return getLeafCount(node->left)+
            getLeafCount(node->right);
}
```

Output:
The leaf count of binary tree is : 3

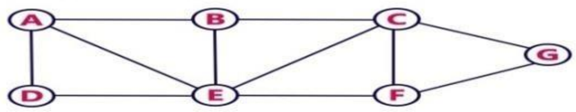
9A. Write a C Program to implement Breadth First Search (BFS)

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit

that vertex and insert it into the Queue.

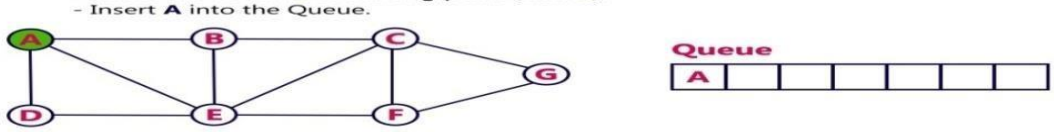
- **Step 3:** Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



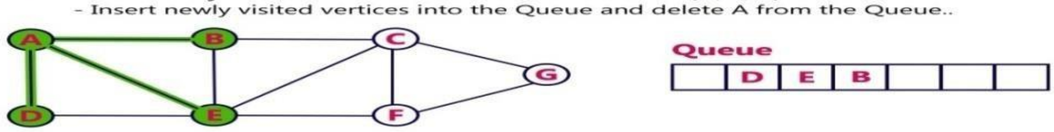
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



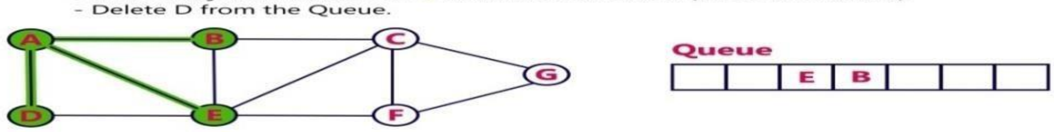
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



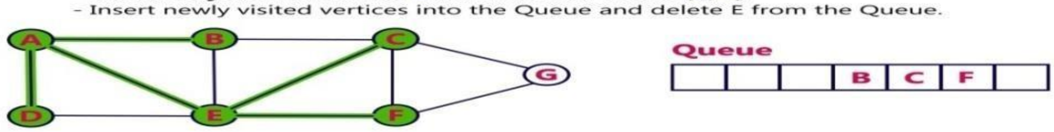
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



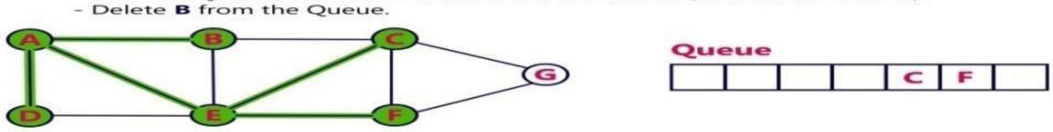
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



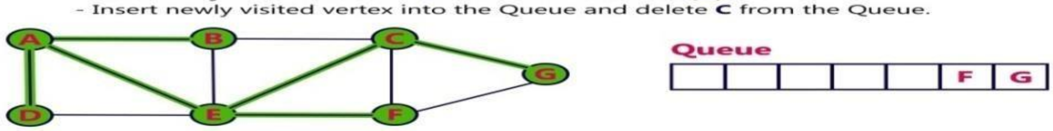
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



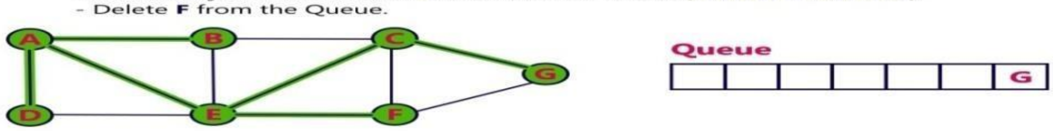
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



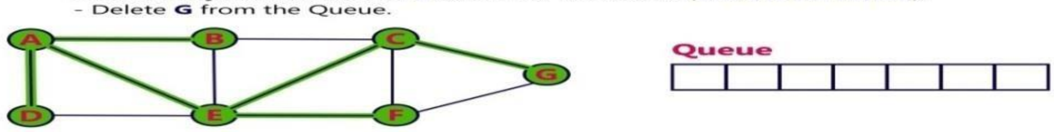
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

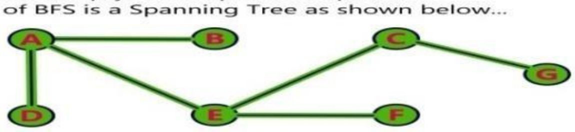


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



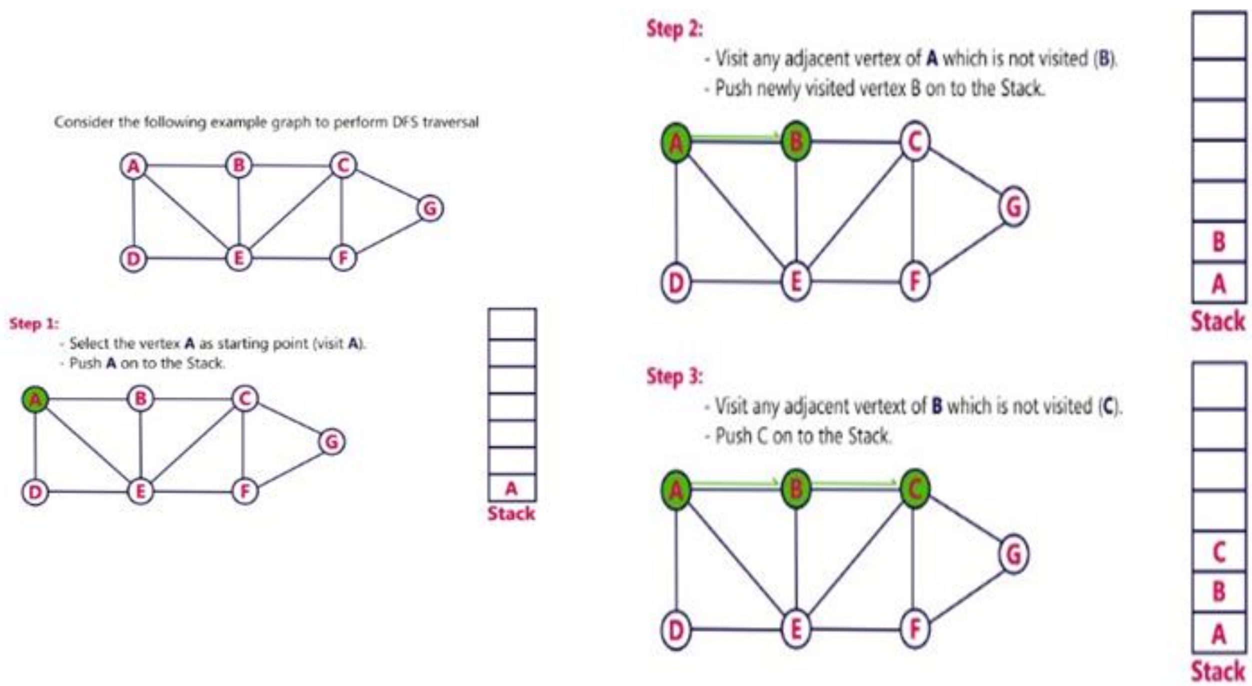
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



```c
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int);
void main()
 {
      int v;
      clrscr();
      printf("\n Enter the number of vertices:");
      scanf("%d",&n);
      for (i=1;i<=n;i++)
  {
            q[i]=0;
            visited[i]=0;
      }
      printf("\n Enter graph data in matrix form:\n");
      for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
         scanf("%d",&a[i][j]);
      printf("\n Enter the starting vertex:");
      scanf("%d",&v);
      bfs(v);
      printf("\n The node which are reachable are:\n");
      for (i=1;i<=n;i++)
        if(visited[i])
         printf("%d\t",i); else
         printf("\n Bfs is not possible");
      getch();
}
void bfs(int v)
{
      for (i=1;i<=n;i++)
        if(a[v][i] && !visited[i])
         q[++r]=i;
      if(f<=r)
 {
            visited[q[f]]=1;
            bfs(q[f++]);
      }
}
```

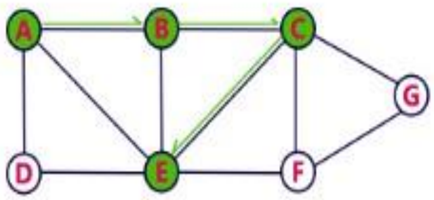
9B. Write a C Program to implement Depthth First Search (DFS)

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph



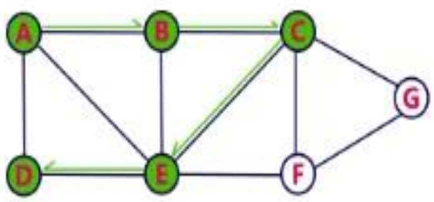
Step 4:

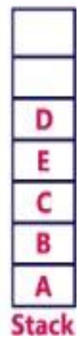
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



Step 5:

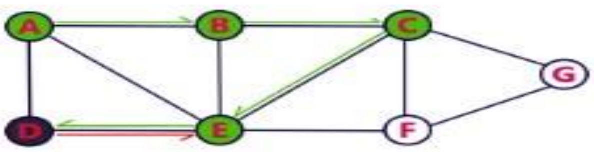
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

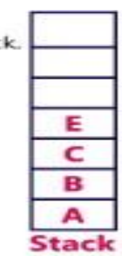




Step 6:

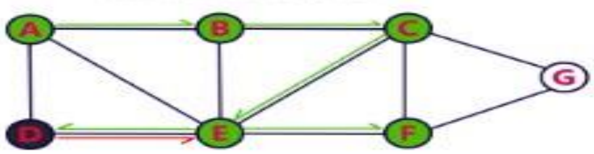
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

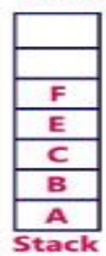




Step 7:

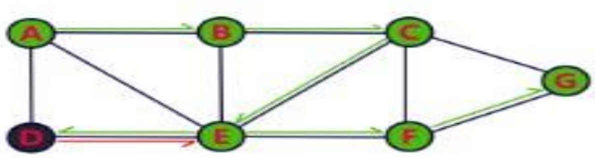
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

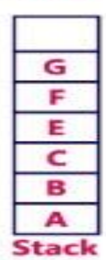




Step 8:

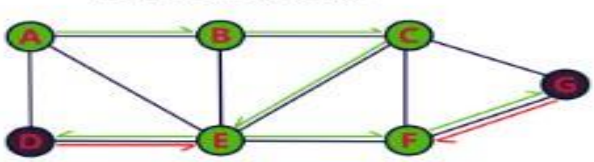
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

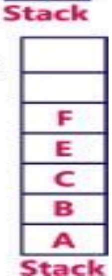




Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.





```c
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int );
void main()
 {
      int i,j,count=0;
      clrscr();
      printf("\n Enter number of vertices:");
      scanf("%d",&n);
      for (i=1;i<=n;i++) {
                reach[i]=0;
                for (j=1;j<=n;j++)
                   a[i][j]=0;
      }
      printf("\n Enter the adjacency matrix:\n");
      for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
         scanf("%d",&a[i][j]);
      dfs(1);
      printf("\n");
      for (i=1;i<=n;i++) {
                if(reach[i])
                   count++;
      }
      if(count==n)
        printf("\n Graph is connected"); else
        printf("\n Graph is not connected");
      getch();
}
void dfs(int v) {
      int i;
      reach[v]=1;
      for (i=1;i<=n;i++)
        if(a[v][i] && !reach[i]) {
                printf("\n %d->%d",v,i);
                dfs(i);
      }
}
```

10. A) write a c program to implement different hash methods in c

```
#include<stdio.h>
#include<stdlib.h>
 struct data
{
    int key;
    int value;
};
 struct data *array;
int capacity = 10;
int size = 0;
 int hashcode(int);
int get_prime(int);
int if_prime(int);
void init_array();
void remove_element(int);
void remove_element(int);
void display();
int size_of_hashtable();

void main()
{
    int choice, key, value, n, c;
    clrscr();
    init_array();
    do {
                printf("\n Implementation of Hash Table in C \n\n");
                printf("MENU-:  \n1.Inserting item in the Hash Table"
                          "\n2.Removing item from the Hash Table"
                             "\n3.Check the size of Hash Table"
                          "\n4.Display a Hash Table"
                      "\n\n Please enter your choice -:");

                scanf("%d", &choice);
                switch(choice)
             {
                case 1:
                     printf("Inserting element in Hash Table\n");
                     printf("Enter key -:\t");
                     scanf("%d", &key);
                     insert(key);
```

```
            break;
        case 2:
            printf("Deleting in Hash Table \n Enter the key to delete-:");
            scanf("%d", &key);
            remove_element(key);
            break;
        case 3:
            n = size_of_hashtable();
            printf("Size of Hash Table is-:%d\n", n);
            break;
        case 4:
            display();
            break;
        default:
             printf("Wrong Input\n");
        }

        printf("\n Do you want to continue-:(press 1 for yes)\t");
        scanf("%d", &c);

  }while(c == 1);
  getch();
 }
/* this function gives a unique hash code to the given key */
int hashcode(int key)
{
  return (key % capacity);
}


/* it returns prime number just greater than array capacity */
int get_prime(int n)
{
  if (n % 2 == 0)
     {
        n++;
  }
  for (; !if_prime(n); n += 2);

  return n;
}

/* to check if given input (i.e n) is prime or not */
```

```
int if_prime(int n)
{
    int i;
    if ( n == 1  ||  n == 0)
        {
                return 0;
    }
    for (i = 2; i < n; i++)
        {
                if (n % i == 0)
            {
                        return 0;
                }
    }
    return 1;
}

void init_array()
{
    int i;
    capacity = get_prime(capacity);
    array = (struct data*) malloc(capacity * sizeof(struct data));
    for (i = 0; i < capacity; i++)
        {
                array[i].key = 0;
                array[i].value = 0;
    }
}



/* to insert a key in the hash table */
void insert(int key)
{
    int index = hashcode(key);
    if (array[index].value == 0)
        {
                /*  key not present, insert it  */
                array[index].key = key;
                array[index].value = 1;
                size++;
                printf("\n Key (%d) has been inserted \n", key);
    }
    else if(array[index].key == key)
```

```
    {
            /*  updating already existing key  */
            printf("\n Key (%d) already present, hence updating its value \n", key);
            array[index].value += 1;
    }
    else
    {
            /*  key cannot be insert as the index is already containing some other key  */
            printf("\n ELEMENT CANNOT BE INSERTED \n");
    }
}

/* to remove a key from hash table */
void remove_element(int key)
{
    int index  = hashcode(key);
    if(array[index].value == 0)
    {
            printf("\n This key does not exist \n");
    }
    else {
            array[index].key = 0;
            array[index].value = 0;
            size--;
            printf("\n Key (%d) has been removed \n", key);
    }
}




/* to display all the elements of a hash table */
void display()
{
    int i;
    for (i = 0; i < capacity; i++)
    {
            if (array[i].value == 0)
            {
                    printf("\n Array[%d] has no elements \n");
            }
            else
            {
```

```
                    printf("\n array[%d] has elements -:\n key(%d) and value(%d) \t", i, array[i].key, array[i].value);
                }
    }
}

int size_of_hashtable()
{
    return size;
}
```

11A. Implementation of Quick Sort in C

Quick sort works in the following manner:

1. Taking the analogical view in perspective, consider a situation where one had to sort the papers bearing the names of the students, by name. One might use the approach as follows:

a. Select a splitting value, say L. The splitting value is also known as **Pivot**.

b. Divide the stack of papers into two. A-L and M-Z. It is not necessary that the piles should be equal.

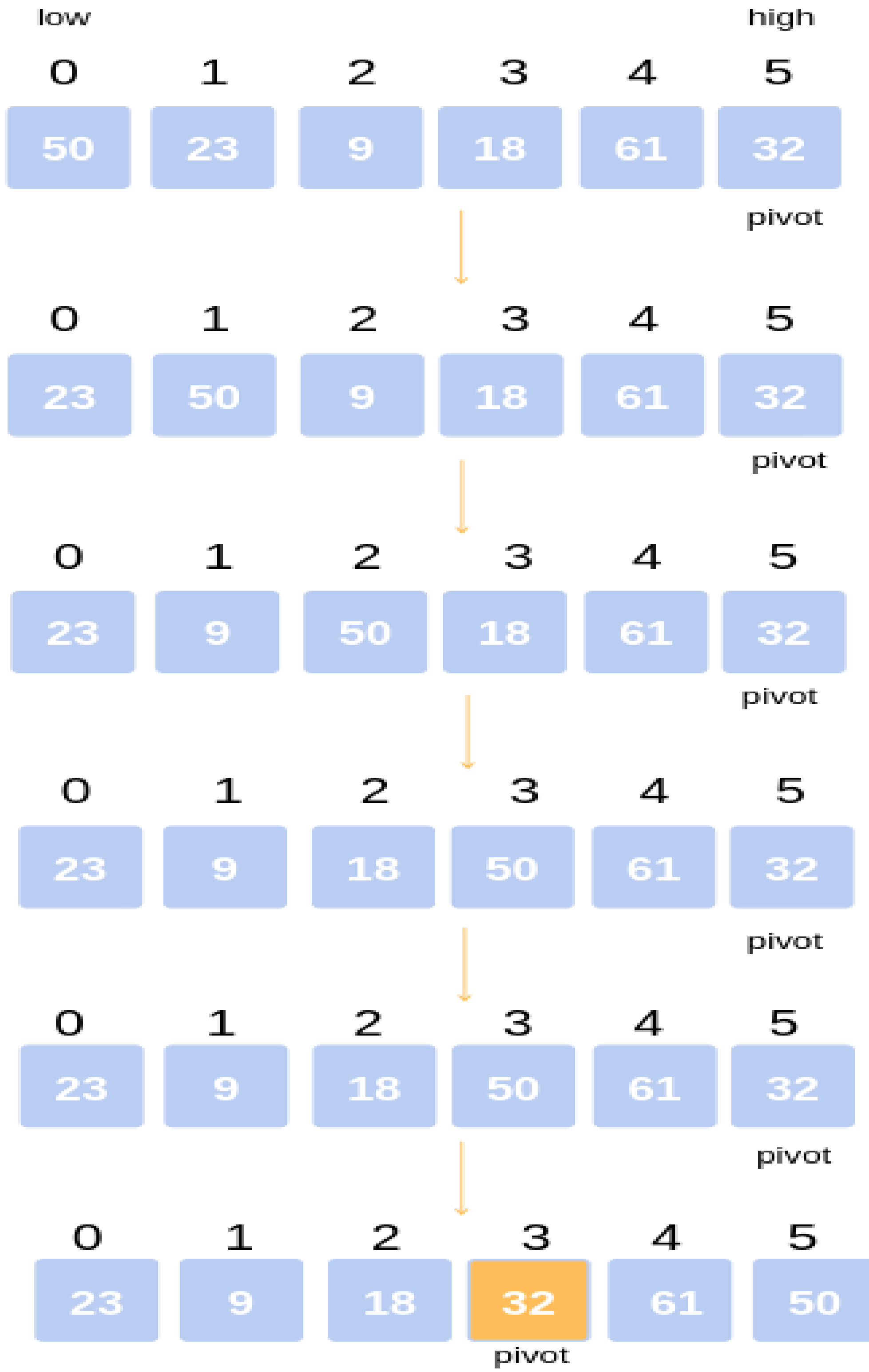
c. Repeat the above two steps with the A-L pile, splitting it into its significant two halves. And M-Z pile, split into its halves. The process is repeated until the piles are small enough to be sorted easily.

d. Ultimately, the smaller piles can be placed one on top of the other to produce a fully sorted and ordered set of papers.

2. The approach used here is **recursion** at each split to get to the single-element array.
3. At every split, the pile was divided and then the same approach was used for the smaller piles.
4. Due to these features, quick sort is also called as ***partition exchange sort.***

An example might come in handy to understand the concept.

Consider the following array: 50, 23, 9, 18, 61, 32

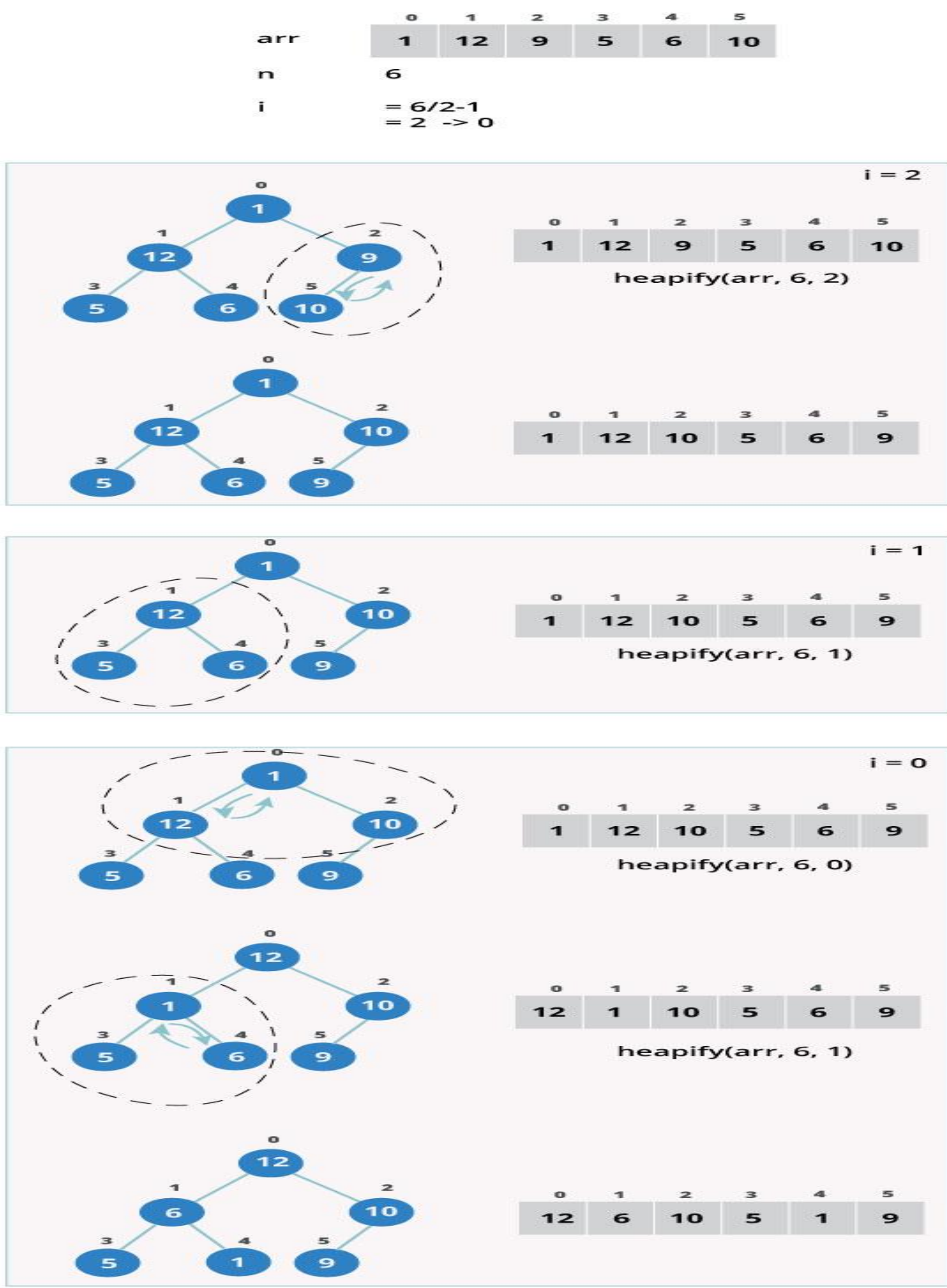

low
high
0 1 2 3 4 5
50 23 9 18 61 32
pivot
0 1 2 3 4 5
23 50 9 18 61 32
pivot
0 1 2 3 4 5
23 9 50 18 61 32
pivot
0 1 2 3 4 5
23 9 18 50 61 32
pivot
0 1 2 3 4 5
23 9 18 50 61 32
pivot
0 1 2 3 4 5
23 9 18 32 61 50
pivot

```c
#include<stdio.h>
void quicksort(int [],int, int);

int main()
{
  int i, count, number[25];

  printf("How many elements are u going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);


  quicksort(number,0,count-1);

  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);

  return 0;
}
void quicksort(int number[25],int first,int last)
{
  int i, j, pivot, temp;

  if(first<last){
    pivot=first;
    i=first;
    j=last;
    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);
  }
}
```

11B. Implementation Heap Sort in C

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap.**
- **Step 3 -** Delete the root element from Min Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

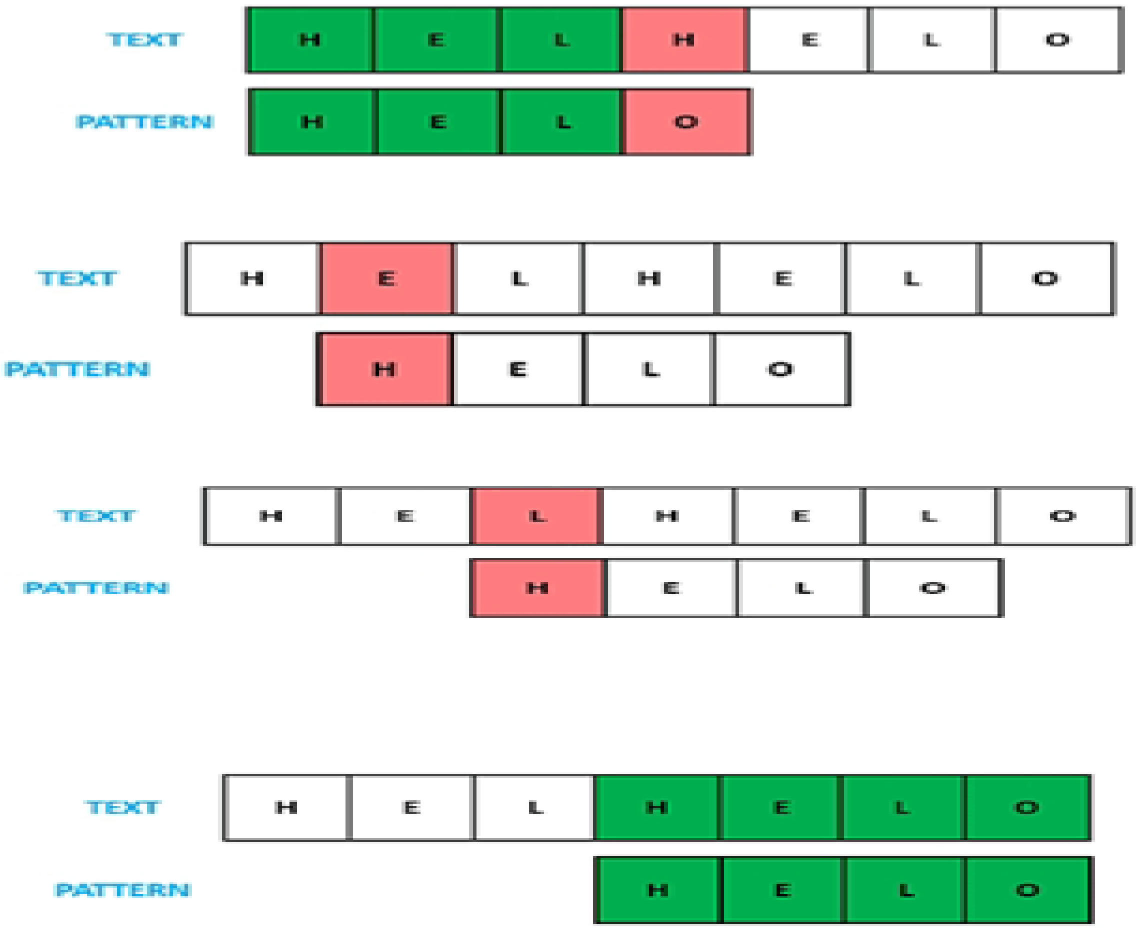


```c
#include<stdio.h>
int temp;
 void heapify(int [], int, int);
void heapSort(int [], int);

void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);

printf("printing sorted elements\n");
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
}

void heapify(int arr[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < size && arr[left] >arr[largest])
largest = left;

if (right < size && arr[right] > arr[largest])
largest = right;

if (largest != i)
{
temp = arr[i];
    arr[i]= arr[largest];
    arr[largest] = temp;
heapify(arr, size, largest);
}
}

void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
```

```
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
temp = arr[0];
    arr[0]= arr[i];
    arr[i] = temp;
heapify(arr, i, 0);
}
}
```

12A. Implement brute-force method of string matching in C

The principle of brute-force search is quite simple. Comparing the characters from left to right continuously (it is crucial because faster approaches work differently). The algorithm checks whether the actual character in the text matches the give character in the pattern. So, the concrete illustration how brute-force substring search works is as follows.




```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
 int i,j,k,n,m,flag=0;
 char t[40],p[30];
 clrscr();
 printf("Enter text: ");
 gets(t);
 printf("\nEnter pattern: ");
 gets(p);
 n=strlen(t);
```

```
 m=strlen(p);
 for(i=0;i<=n-m;i++)
 {
  j=0;
  while(j<m && p[j]==t[j+i])
   {
     j++;
     if(j==m)
     {
      flag=1;
      k=i+1;
     }
     else
      flag=0;
   }
 }



 if(flag==1)
    printf("\nPattern found at position: %d\n ",k);
 else
    printf("\nPattern not found in text \n");
  getch();
}
```

output:

```
Enter text: mca programme

Enter pattern: programme

Pattern found at position: 5
```

12B. KMP matching algorithm in C

Algorithm: Knuth-MorrisPratt

Step 1. n = T.length
Step 2. m=P.length3
Step 3. T = Computer-Prefix-function(p)
Step 4. q = 0
Step 5. for i = 1to n
Step 6. while q>0 and P[q+1]=/ T[i]
Step 7. q = 3.14[q]
Step 8. if P[q+1]== T[i]
Step 9. q = q+1
Step 10. if q == m
Step 11. Print ”Pattern occurs with shift” i-m
Step 12. q=3.14[q]

Example:

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

```
#include<iostream>
#include<string.h>
using namespace std;
void prefixSuffixArray(char* , int , int*);
void KMPAlgorithm(char*, char*);
```

```
int main()
 {
```

```
   char text[] = "xyztrwqxyzfg";
   char pattern[] = "xyz";
   printf("The pattern is found in the text at the following index : \n");
   KMPAlgorithm(text, pattern);
   return 0;
}
void prefixSuffixArray(char* pat, int M, int* pps)
 {
```

```
   int length = 0;
   pps[0] = 0;
   int i = 1;
   while (i < M)
{
      if (pat[i] == pat[length])
     {
         length++;
         pps[i] = length;
         i++;
      }
     else
       {
         if (length != 0)
         length = pps[length - 1];
         else
         {
            pps[i] = 0;
            i++;
         }
       }
   }
}
```

```
void KMPAlgorithm(char* text, char* pattern)
 {
   int M = strlen(pattern);
   int N = strlen(text);
   int pps[M];
   prefixSuffixArray(pattern, M, pps);
   int i = 0;
   int j = 0;
   while (i < N)
   {
      if (pattern[j] == text[i])
     {
         j++;
         i++;
      }
      if (j == M)
     {
         printf("Found pattern at index %d\n", i - j);
         j = pps[j - 1];
```

```
        }
        else if (i < N && pattern[j] != text[i])
        {
            if (j != 0)
            j = pps[j - 1];
            else
            i = i + 1;
        }
    }
}
```

Output:

```
Found pattern at index 0
Found pattern at index 7
```