CS8391 - Data Structures
ANNA UNIVERSITY EXAMINATIONS

UNIT- I
PART A

1. **Define: data structure.**
   A data structure is a way of storing and organizing data in the memory for efficient usage. The way information is organized in the memory of a computer.
2. **Give few examples for data structures?**
   Arrays, stacks, queue, list, tree, graph, set, map, table and deque.
3. **What are the different types of data structures?**
   i) Primitive
   ii) Composite
   iii) Abstract
4. **What are primitive data types?**
   The basic building blocks for all data structures are called primitive data types.
   (e.g) int, float, char, double, Boolean
5. **What are composite data types?**
   Composite data types are composed of more than one primitive data type.
   (e.g) array,structure,union
6. **What is meant by an abstract data type?**
   An ADT is a mathematical model for a certain class of data structures that have similar behavior. (e.g) list, stack, queue
7. **How can we categorize data structures based on data access?**
   Linear – list, stack, queue
   Non-linear- heap, tree, graph
8. **State the difference between linear and non-linear data structures.**
   The main difference between linear and nonlinear data structures lie in the way they organize data elements.
   In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory.
   In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Due to this it might be difficult to be implemented in computer's linear memory.
9. **List a few real-time applications of data structures?**
   - Undo and redo feature - stack
   - Decision making - graph
   - Printer (printing jobs) – queue
   - Compilers – hash table
   - Directory structure- trees
   - Communication networks- graphs
10. **Define List.**

The general form of the list is a_1, a_2, a_3… a_n. The size of the list is 'n'. Any element in the list at the position i is defined to be at a_i, a_{i+1} the successor of a_i , and a_{i-1} is the predecessor of a_i. a_1 doesn't have predecessor and a_n doesn't have successor.

11. What are the various operations done on List ADT?

The operations done under List ADT are Print list, Insert, Delete, FindPrevious, Find k^{th}, Find, MakeEmpty, IsLast and IsEmpty.

12. What are the different ways to implement list?

- Array implementation of list
- Linked list implementation of list
- Cursor implementation of list

13. Arrays are not used to implement lists. Why?

- Requires that the list size to be known in advance
- Running time for insertions and deletions is slow

14. What are the advantages in the array implementation of list?

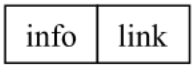
- Print list operation can be carried out at linear time
- Finding K^{th} element takes a constant time

15. What are the disadvantages in the array implementation of list?

The running time for insertions and deletions is so slow and the list size must be known in advance.

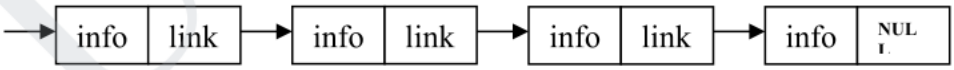
16. Define node.

A node consists of two fields namely an information field called INFO and a pointer field called LINK. The INFO field is used to store the data and the LINK field is used to store the address of the next field.



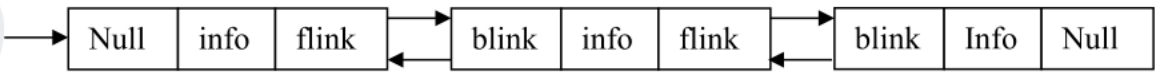
17. What is a linked list?

Linked list is series of nodes, which are not necessarily adjacent in memory. Each node contains a data element and a pointer to the next node.



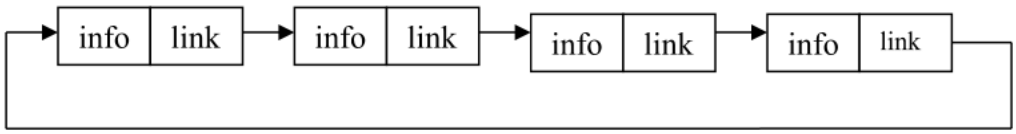
18. What is a doubly linked list?

In a doubly linked list, along with the data field there will be two pointers one pointing the next node(flink) and the other pointing the previous node(blink).



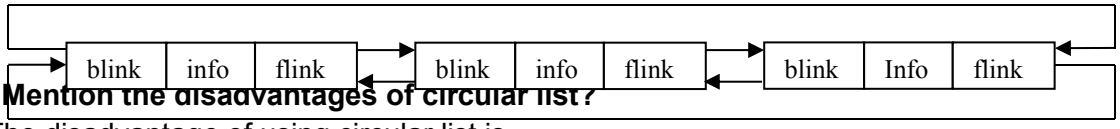
19. Define circularly linked list?

In a singly circular linked list the last node's link points to the first node of the list.



20. **Define double circularly linked list?**

In a circular doubly linked list the last node's forward link points to the first ode of the list, and the first node's back link points to the last node of the list.



21. **Mention the disadvantages of circular list?**

The disadvantage of using circular list is

- It is possible to get into an infinite loop.
- It is not possible to detect the end of the list.

22. **What is the need for the header?**

Header of the linked list is the first element in the list and it may store the number of elements in the list. It points to the first data element of the list.

Without header

- Insertion at the front of the list needs special coding & updating of the linked list address.
- Deletion at the front of the list also needs special coding & updating of the linked list address.

23. **List three applications that uses linked list?**

Three examples/applications that uses linked list are Polynomial representation, radix sort, & multi lists.

24. **What are the disadvantage of linked list over array?(Nov/Dec 2018)**

Nodes do not have their own address. Only the address of the first node is stored and in order to reach any node, we need to traverse the whole **list** from beginning to the desired node. As all Nodes don't have their particular address, BINARY SEARCH cannot be performed.

25. **State the advantages of ADT?(Nov/Dec 2018)**

- ADT is reusable, robust, and is based on principles of Object Oriented Programming (OOP) and Software Engineering (SE)
- An ADT can be re-used at several places and it reduces coding efforts
- Encapsulation ensures that data cannot be corrupted
- Working of various integrated operation cannot be tampered with by the application program
- ADT ensures a robust data structure

PART-B

1. Derive an ADT to perform insertion and deletion in a singly linked list.(8) **(Nov/Dec 18)**

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

```
struct node
{
   int data;
   struct node *next;
}*start=NULL,*q,*t;

int main()
{
   int ch;
   void insert_beg();
   void insert_end();
   int insert_pos();
   void display();
   void delete_beg();
   void delete_end();
   int delete_pos();

   while(1)
   {
      printf("\n\n---- Singly Linked List(SLL) Menu ----");
      printf("\n1.Insert\n2.Display\n3.Delete\n4.Exit\n\n");
      printf("Enter your choice(1-4):");
      scanf("%d",&ch);

      switch(ch)
      {
         case 1:
               printf("\n---- Insert Menu ----");
               printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at specified
position\n4.Exit");
               printf("\n\nEnter your choice(1-4):");
               scanf("%d",&ch);

               switch(ch)
               {
                  case 1: insert_beg();
                        break;
                  case 2: insert_end();
                        break;
                  case 3: insert_pos();
                        break;
                  case 4: exit(0);
                  default: printf("Wrong Choice!!");
               }
               break;

         case 2: display();
               break;

         case 3: printf("\n---- Delete Menu ----");
```

```
                printf("\n1.Delete from beginning\n2.Delete from end\n3.Delete from
specified position\n4.Exit");
                printf("\n\nEnter your choice(1-4):");
                scanf("%d",&ch);

                switch(ch)
                {
                   case 1: delete_beg();
                         break;
                   case 2: delete_end();
                         break;
                   case 3: delete_pos();
                         break;
                   case 4: exit(0);
                   default: printf("Wrong Choice!!");
                }
                break;
          case 4: exit(0);
                default: printf("Wrong Choice!!");
       }
    }
    return 0;
}

void insert_beg()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;

    if(start==NULL)       //If list is empty
    {
       t->next=NULL;
       start=t;
    }
    else
    {
       t->next=start;
       start=t;
    }
}

void insert_end()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;
```

```
    t->next=NULL;

    if(start==NULL)        //If list is empty
    {
        start=t;
    }
    else
    {
        q=start;
        while(q->next!=NULL)
        q=q->next;
        q->next=t;
    }
}

int insert_pos()
{
    int pos,i,num;
    if(start==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    printf("Enter position to insert:");
    scanf("%d",&pos);
    t->data=num;

    q=start;
    for(i=1;i<pos-1;i++)
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }

        q=q->next;
    }

    t->next=q->next;
    q->next=t;
    return 0;
}

void display()
{
    if(start==NULL)
```

```
    {
        printf("List is empty!!");
    }
    else
    {
        q=start;
        printf("The linked list is:\n");
        while(q!=NULL)
        {
            printf("%d->",q->data);
            q=q->next;
        }
    }
}

void delete_beg()
{
    if(start==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        q=start;
        start=start->next;
        printf("Deleted element is %d",q->data);
        free(q);
    }
}

void delete_end()
{
    if(start==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        q=start;
        while(q->next->next!=NULL)
        q=q->next;

        t=q->next;
        q->next=NULL;
        printf("Deleted element is %d",t->data);
        free(t);
    }
}

int delete_pos()
{
```

```
    int pos,i;

    if(start==NULL)
    {
       printf("List is empty!!");
       return 0;
    }

    printf("Enter position to delete:");
    scanf("%d",&pos);

    q=start;
    for(i=1;i<pos-1;i++)
    {
       if(q->next==NULL)
       {
          printf("There are less elements!!");
          return 0;
       }
       q=q->next;
    }

    t=q->next;
    q->next=t->next;
    printf("Deleted element is %d",t->data);
    free(t);

    return 0;
}
```

2. Design an algorithm to reverse the linked list. Trace it with an example?(8)

```
// Iterative C++ program to reverse
// a linked list
#include<iostream>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
    Node (int data)
    {
            this->data = data;
            next = NULL;
    }
};

struct LinkedList
```

```
{
    Node *head;
    LinkedList()
    {
            head = NULL;
    }


    /* Function to reverse the linked list */
    void reverse()
    {
            // Initialize current, previous and
            // next pointers
            Node *current = head;
            Node *prev = NULL, *next = NULL;


            while (current != NULL)
            {
                    // Store next
                    next = current->next;

                    // Reverse current node's pointer
                    current->next = prev;

                    // Move pointers one position ahead.
                    prev = current;
                    current = next;
            }
            head = prev;
    }

    /* Function to print linked list */
    void print()
    {
            struct Node *temp = head;
            while (temp != NULL)
            {
                    cout << temp->data << " ";
                    temp = temp->next;
            }
    }

    void push(int data)
    {
```

```
            Node *temp = new Node(data);
            temp->next = head;
            head = temp;
        }
    };

    /* Driver program to test above function*/
    int main()
    {
        /* Start with the empty list */
        LinkedList ll;
        ll.push(20);
        ll.push(4);
        ll.push(15);
        ll.push(85);

        cout << "Given linked list\n";
        ll.print();

        ll.reverse();

        cout << "\nReversed Linked list \n";
        ll.print();
        return 0;
    }
```

3. Write an algorithm for inserting and deleting an element from Circular linked list?(8)

```
#include<stdio.h>
struct node
{
    int data;
    struct node *next;
};
struct node *start;
void insertbeg(void)
{
    struct node *nn,*temp;int a;
    nn=(struct node *)malloc(sizeof(struct node));
    printf("enter data:");
    scanf("%d",&nn->data);
    a=nn->data;
    if(start==NULL)
    {
```

```
            nn->next=nn;
            start=nn;
        }
        else
        {
            temp=start;
            while(temp->next!=start)
            {
                temp=temp->next;
            }
            temp->next=nn;
            nn->next=start;
            start=nn;
        }
        printf("%d succ. inserted\n",a);
        return;
}
void insertend(void)
{
        struct node *nn,*lp;int b;
        nn=(struct node *)malloc(sizeof(struct node));
        printf("enter data:");
        scanf("%d",&nn->data);
        b=nn->data;
        if(start==NULL)
        {
            nn->next=nn;
            start=nn;
        }
        else
        {
            lp=start;
            while(lp->next!=start)
            {
                lp=lp->next;
            }
            lp->next=nn;
            nn->next=start;
        }
        printf("%d is succ. inserted\n",b);
        return;
```

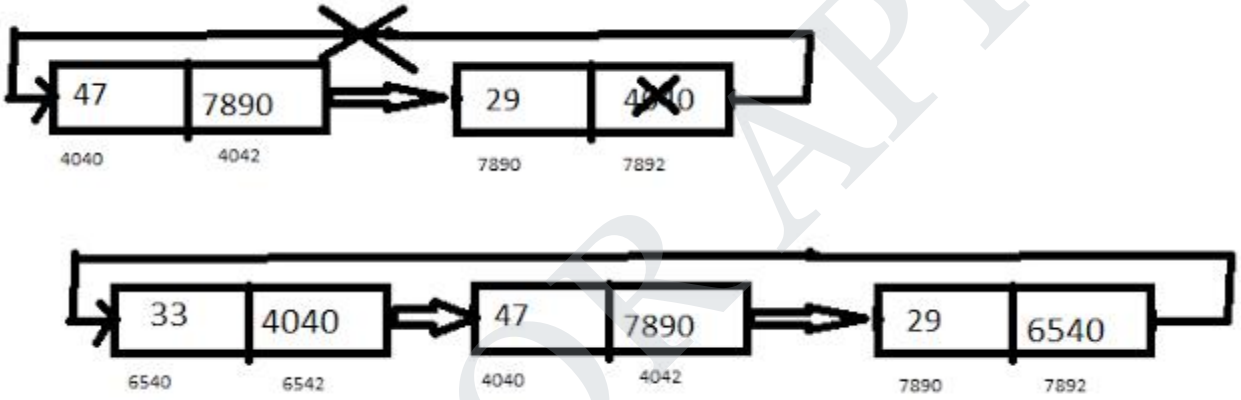
```
}
void insertmid(void)
{
    struct node *nn,*temp,*ptemp;int x,v;
    nn=(struct node *)malloc(sizeof(struct node));
    if(start==NULL)
    {
        printf("sll is empty\n"); return;
    }
    printf("enter data before which no. is to be inserted:\n");
    scanf("%d",&x);
    if(x==start->data)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

```
    int x;
    if(start==NULL)
    {
        printf("sll is empty\n");
        return;
    }
    printf("enter data to be deleted:");
    scanf("%d",&x);
    if(x==start->data)
    {
        pp=t=start;
if(start->next==start)
            {
  free(pp);
       start=NULL;
   printf("%d succ. deleted",x);
  return;
            }
        while(t->next!=start)
        {
            t=t->next;
        }
        t->next=start->next;
        start=start->next;
        free(pp);
        printf("%d is succ. deleted\n",x);
        return;
   }
    pt=start;
    t=start->next;
    while(t!=start&&t->data!=x)
    {
        pt=t;t=t->next;
    }
    if(t==start)
    {
        printf("%d does not exist\n",x);return;
    }
    else
    {
        pt->next=t->next;
```

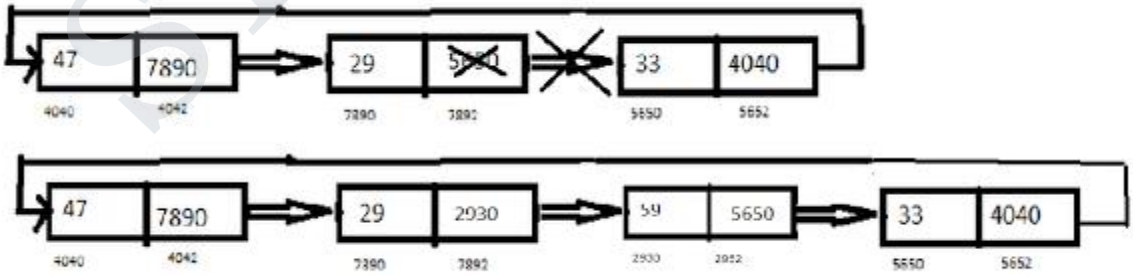
```
    }
    printf("%d is succ. deleted\n",x);
    free(t);
    return;
}
void display(void)
{
    struct node *temp;
    if(start==NULL)
    {
        printf("sll is empty\n");
        return;
    }
    printf("elements are:\n");
    temp=start;
    while(temp->next!=start)
    {
        printf("%d\n",temp->data);
        temp=temp->next;
    }
    printf("%d\n",temp->data);
    return;
}
void main()
{
    int c,a; start=NULL;
    do
    {
        printf("1:insert\n2:delete\n3:display\n4:exit\nenter choice:");
        scanf("%d",&c);
        switch(c)
        {
            case 1:
                printf("1:insertbeg\n2:insert end\n3:insert mid\nenter choice:");
                scanf("%d",&a);
                switch(a)
                {
                    case 1:insertbeg();break;
                    case 2:insertend();break;
                    case 3:insertmid();break;
                }
```

```
                break;
            case 2:deletion();break;
            case 3:display();break;
            case 4:printf("program ends\n");break;
            default:printf("wrong choice\n");
            break;
        }
    }while(c!=4);
}
```

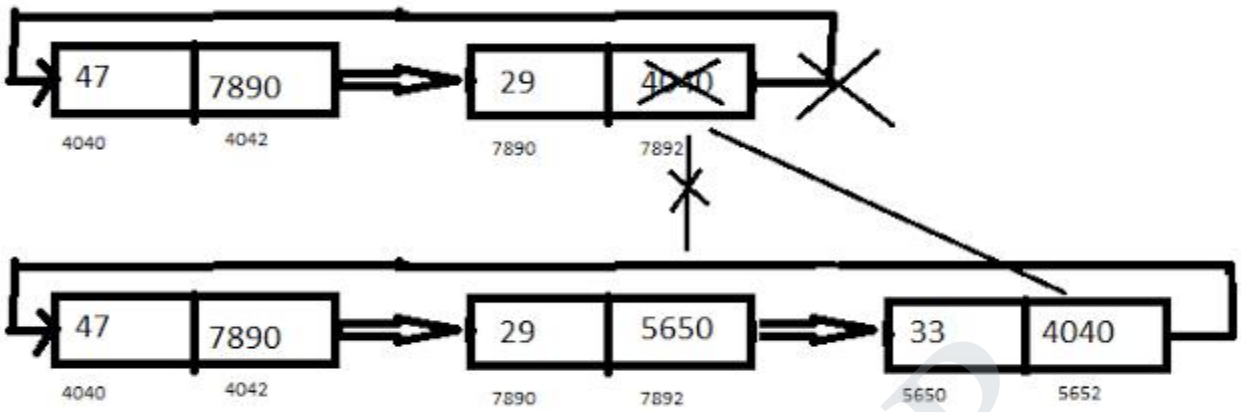
Detail Explanation:let us consider the code in insertbeg().we check that is linked list empty or not by checking value of Start!=NULL. (Note: Click on image for better view)



if start=null then the new created node is assign to Start else consider the code
temp=start;
while(temp->next!=start)
    temp=temp->next;
temp->next=nn;
 nn->next=start; start=nn;
assume the above image,we want to add 33 at the begg.so the temp pointer is traversed to the end of list,inserting the address of new node in temp->next and inserting address pointed by start in the new nodes next,make the start pointer to point new node.

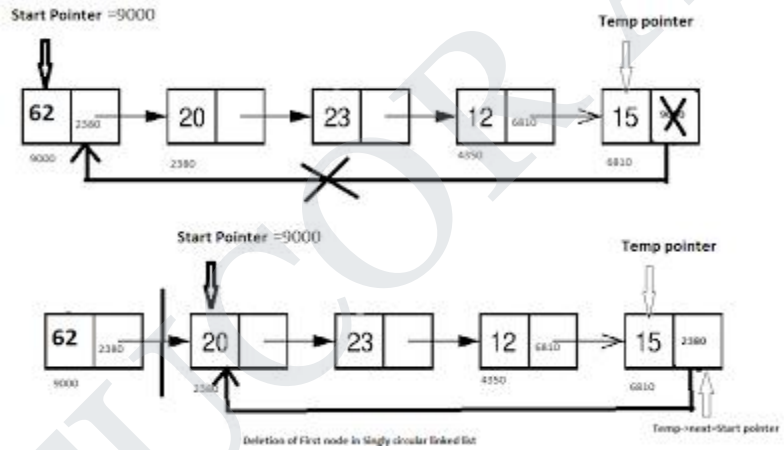


No Explanation is needed for inserting element at the mid,it remains the same as that of singly linked list

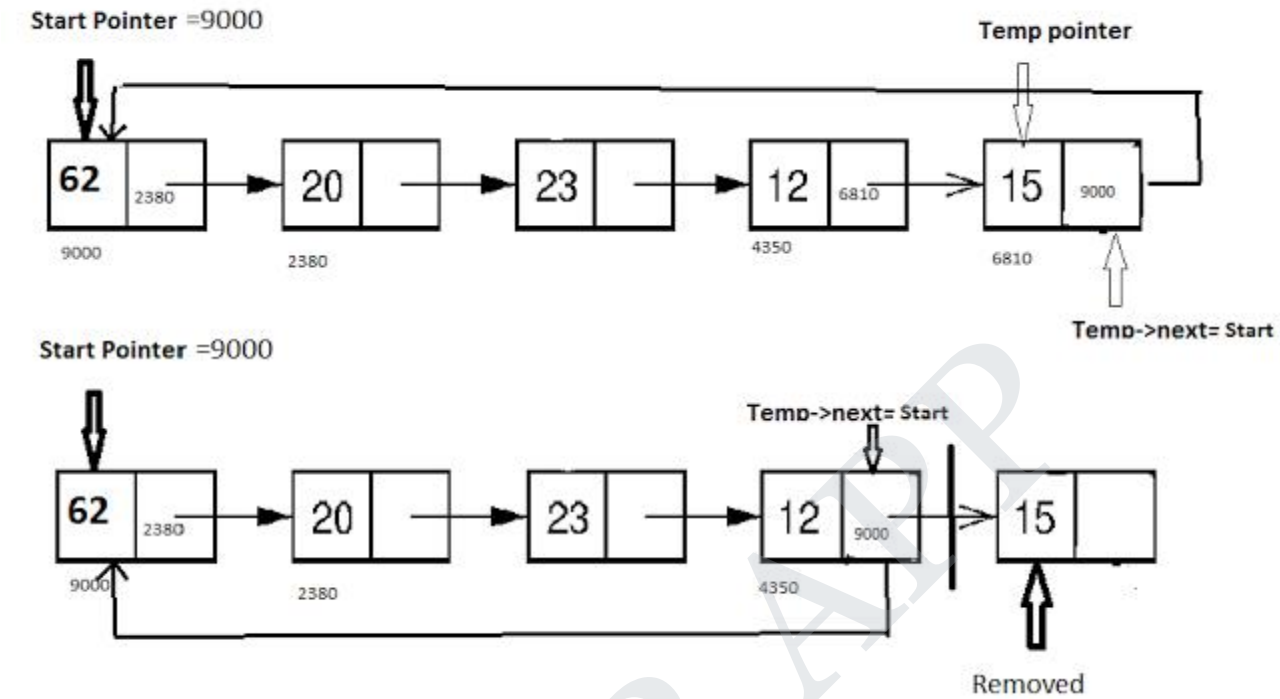


Inserting 33 at the end of list

Consider the case of inserting element at the end of singly circular linked list. Assume we wanted to add 33 at the end of list,we will traverse the temporary pointers till the temps next address is not start.So at this point we have last node pointed by temp.we will insert temp->next address in new nodes next and new nodes address in temp->next.



Deletion of First node.For deletion of first node again we have to traversed the temp pointer till the end of list so that address of start can be removed from temp->next and addrss of start's next is inserted in temp->next.Start pointer is made to point the address contained by start->next. Deletion of middle node is same as that of singly linked list



Deletion of Last Element in the singly circular linked list we have to traversed the temp pointer till the end of list and ptemp till the previous node.so the address contain by last node->next is inserted at next of node pointed by ptemp.

4. Write the algorithm for the deletion and reverse operations on doubly linked list. (8) (Nov 09)

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

struct node
{
  int element;
  struct node* flink;
  struct node* blink;
};

struct node* createlist(void)
{  struct node* l;
   l=( struct node*) malloc (sizeof (struct node));
   l->flink=NULL;
   l->blink=NULL;
       return l;
}
void insert(int x,list l, struct node* p)
{  struct node* tmpcell,q;
   tmpcell=( struct node*) malloc (sizeof (struct node));
   tmpcell->element=x;
   q=p->flink;
```

```
   tmpcell->flink=q;
   tmpcell->blink=p;
   p->flink=tmpcell;
   if (q)
       q->blink=tmpcell;
}
int isempty(struct node* l)
{  return l->flink==NULL; }
void delete1(int x, struct node* l)
{ struct node* p,q,r;
  p=findprevious(x,l);
  q=p->flink;
  r=q->flink;
  p->flink=r;
  if(r)
          r->blink=p;
}
position find(int x, struct node* l)
{ position p;
 p=l->flink;
 while(p!=NULL && p->element!=x)
   p=p->flink;
 return p;
}
position findprevious(elementtype x, list l)
{ position p;
  p=l;
  while(p->flink!=NULL && p->flink->element!=x)
     p=p->flink;
  return p;
}
void printlist(struct node* l)
{         struct node* p=l;
          while(p!=NULL)
          { p=p->flink;
            printf("%d",p->element);
          }
}
void main()
{
  int ch,data;
  struct node *prev,*p,*pos;
  struct node *l;
  l=createlist();
  while(1)
{ printf("\n1. Insert operation");
  printf("\n2. Delete an element");
  printf("\n3. Find operation");
  printf("\n4. Display List elements");
  printf("\n5. Quit");
  printf("\n Enter your choice");
```

```
 scanf("%d",&ch);
 switch(ch)
 {case 1:
          printf("\n Enter the element");
          scanf("%d",&data);
          prev=l;
          p=l->flink;
          while (p!=NULL && data>p->element)
          { prev=p;
            p=p->flink;
          }
          insert(data,l,prev);      break;
  case 2:
          printf("\n Enter the element to be Deleted");
          scanf("%d",&data);
          delete1(data,l);          break;
  case 4:
         printf("\n Enter the element to be searched");
         scanf("%d",&data);
         pos=find(data,l);
         if (pos==NULL)
           printf("\nElement not found");
         else
           printf("\nElement found ");
        break;
   case 5:
         printlist(l);        break;
  case 6:
         exit(0);
 }

 } // End of While loop
 }// End of Main
```

5. Write algorithms to perform the following in doubly linked list: (may 10)
   (i) To insert an element in the beginning, middle, end of the list. (8)
   (ii) To delete an element from anywhere in the list. (8)
   An element is a structure variable that contains an integer data field and a string data field.

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

struct node
{
  int element;
  struct node* flink;
  struct node* blink;
};

struct node* createlist(void)
```

```
{  struct node* l;
   l=( struct node*) malloc (sizeof (struct node));
   l->flink=NULL;
   l->blink=NULL;
       return l;
}
void insert(int x,list l, struct node* p)
{  struct node* tmpcell,q;
   tmpcell=( struct node*) malloc (sizeof (struct node));
   tmpcell->element=x;
   q=p->flink;
   tmpcell->flink=q;
   tmpcell->blink=p;
   p->flink=tmpcell;
   if (q)
       q->blink=tmpcell;
}
int isempty(struct node* l)
{  return l->flink==NULL; }
void delete1(int x, struct node* l)
{ struct node* p,q,r;
  p=findprevious(x,l);
  q=p->flink;
  r=q->flink;
  p->flink=r;
  if(r)
r->blink=p;
}
position find(int x, struct node* l)
{ position p;
 p=l->flink;
 while(p!=NULL && p->element!=x)
   p=p->flink;
 return p;
}
position findprevious(elementtype x, list l)
{ position p;
  p=l;
  while(p->flink!=NULL && p->flink->element!=x)
     p=p->flink;
  return p;
}
void printlist(struct node* l)
{          struct node* p=l;
           while(p!=NULL)
           { p=p->flink;
             printf("%d",p->element);
           }
}
void main()
{
```

```
 int ch,data;
 struct node *prev,*p,*pos;
 struct node *l;
 l=createlist();
 while(1)
{ printf("\n1. Insert operation");
 printf("\n2. Delete an element");
 printf("\n3. Find operation");
 printf("\n4. Display List elements");
 printf("\n5. Quit");
 printf("\n Enter your choice");
 scanf("%d",&ch);
 switch(ch)
 {case 1:
          printf("\n Enter the element");
          scanf("%d",&data);
          prev=l;
          p=l->flink;
          while (p!=NULL && data>p->element)
          { prev=p;
            p=p->flink;
          }
          insert(data,l,prev);      break;
  case 2:
          printf("\n Enter the element to be Deleted");
          scanf("%d",&data);
          delete1(data,l);          break;
  case 4:
         printf("\n Enter the element to be searched");
         scanf("%d",&data);
         pos=find(data,l);
         if (pos==NULL)
           printf("\nElement not found");
         else
           printf("\nElement found ");
        break;
   case 5:
         printlist(l);       break;
  case 6:
         exit(0);
 }

 } // End of While loop
 }// End of Main
```

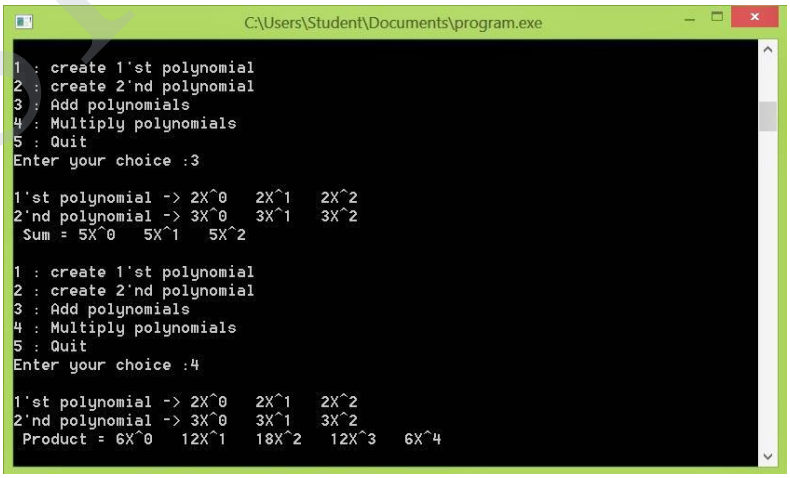
6. Write an algorithm to perform the following polynomial manipulation using linked list representation.**(Nov/Dec 2018)**
   Insertion, Deletion, Merge, Traversal

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

```
#include<conio.h>
#define MAX 17

typedef struct node
  {
 int coeff;
 struct node *next;
  }node;
node *  init();
void read(node *h1);
void print(node *h1);
node * add(node *h1,node *h2);
node * multiply(node *h1,node *h2);

/*Polynomial is stored in a linked list, ith node  gives coefficient of x^i .
  a polynomial 3x^2 + 12x^4 will be represented as (0,0,3,0,12,0,0,....)
*/

void main()
{
 node *h1=NULL,*h2=NULL,*h3=NULL;
 int option;
 do
 {
  printf("\n\n1 : create 1'st polynomial");
  printf("\n2 : create 2'nd polynomial");
  printf("\n3 : Add polynomials");
  printf("\n4 : Multiply polynomials");
  printf("\n5 : Quit");
  printf("\nEnter your choice :");
  scanf("%d",&option);
  switch(option)
  {
   case 1:h1=init();read(h1);break;
   case 2:h2=init();read(h2);break;
   case 3:h3=add(h1,h2);
          printf("\n1'st polynomial -> ");
          print(h1);
          printf("\n2'nd polynomial -> ");
          print(h2);
          printf("\n Sum = ");
          print(h3);
          break;
   case 4:h3=multiply(h1,h2);
```

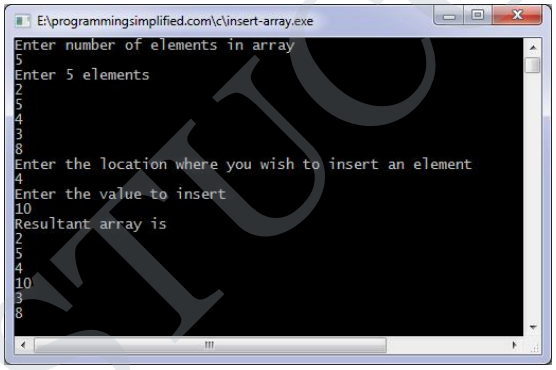
```
            printf("\n1'st polynomial -> ");
            print(h1);
            printf("\n2'nd polynomial -> ");
            print(h2);
            printf("\n Product = ");
            print(h3);
            break;
     }
    }while(option!=5);
}
void  read(node *h)
{
 int n,i,j,power,coeff;
 node *p;
 p=init();
 printf("\n Enter number of terms :");
 scanf("%d",&n);
 /* read n terms */
 for (i=0;i<n;i++)
 {       printf("\nenter a term(power  coeff.)");
  scanf("%d%d",&power,&coeff);
  for(p=h,j=0;j<power;j++)
      p=p->next;
  p->coeff=coeff;
 }
}
void print(node *p)
{
   int i;
 for(i=0;p!=NULL;i++,p=p->next)
  if(p->coeff!=0)
   printf("%dX^%d   ",p->coeff,i);
}
node * add(node *h1, node *h2)
{
    node *h3,*p;
    h3=init();
    p=h3;
    while(h1!=NULL)
     {
 h3->coeff=h1->coeff+h2->coeff;
 h1=h1->next;
 h2=h2->next;
 h3=h3->next;
```

```
    }
 return(p);
}
node * multiply(node *h1, node *h2)
{
 node *h3,*p,*q,*r;
 int i,j,k,coeff,power;
 h3=init();
 for(p=h1,i=0;p!=NULL;p=p->next,i++)
  for(q=h2,j=0;q!=NULL;q=q->next,j++)
     {
   coeff=p->coeff * q->coeff;
   power=i+j;
   for(r=h3,k=0;k<power;k++)
    r=r->next;
   r->coeff=r->coeff+coeff;
      }
  return(h3);
}

node * init()
{
    int i;
    node *h=NULL,*p;
    for(i=0;i<MAX;i++)
 {
  p=(node*)malloc(sizeof(node));
  p->next=h;
  p->coeff=0;
  h=p;
  } return(h);}
```



7. What are the various operations on array ? Write a procedure to insert an element in the middle of the array.

```
#include <stdio.h>

int main()
{
  int array[100], position, c, n, value;
  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d elements\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter the location where you wish to insert an element\n");
  scanf("%d", &position);
  printf("Enter the value to insert\n");
  scanf("%d", &value);
  for (c = n - 1; c >= position - 1; c--)
    array[c+1] = array[c];
  array[position-1] = value;
  printf("Resultant array is\n");
  for (c = 0; c <= n; c++)
    printf("%d\n", array[c]);
  return 0;}
```



8. Write a procedure to deleting the last node from a circular linked list

```
// C++ program to delete a given key from
// linked list.
#include <bits/stdc++.h>
using namespace std;
/* structure for a node */
class Node {
public:
    int data; Node* next; };
```

```
/* Function to insert a node at the beginning of
a Circular linked list */
void push(Node** head_ref, int data)
{
    // Create a new node and make head as next
    // of it.
    Node* ptr1 = new Node();
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the
    next of last node */
    if (*head_ref != NULL) {
            // Find the node before head and update
            // next of it.
            Node* temp = *head_ref;
            while (temp->next != *head_ref)
                    temp = temp->next;
            temp->next = ptr1;
    }
    else
            ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given
circular linked list */
void printList(Node* head)
{
    Node* temp = head;
    if (head != NULL) {
            do {
                    cout << temp->data << " ";
                    temp = temp->next;
            } while (temp != head);
    }

    cout << endl;
}

/* Function to delete a given node from the list */
void deleteNode(Node** head, int key)
{
```

```
        if (*head == NULL)
        return;
    if((*head)->data==key && (*head)->next==*head)
    {
        free(*head);
        *head=NULL; }
    Node *last=*head,*d;
        if((*head)->data==key) {
        while(last->next!=*head)
            last=last->next;
            last->next=(*head)->next;
        free(*head);
        *head=last->next; }
        while(last->next!=*head&&last->next->data!=key) {
        last=last->next; }
        if(last->next->data==key) {
        d=last->next;
        last->next=d->next;
        free(d); }
    else
        cout<<"no such keyfound"; }

int main()
{
    /* Initialize lists as empty */
    Node* head = NULL;
    /* Created linked list will be 2->5->7->8->10 */
    push(&head, 2);
    push(&head, 5);
    push(&head, 7);
    push(&head, 8);
    push(&head, 10);
    cout << "List Before Deletion: ";
    printList(head);
    deleteNode(&head, 7);
    cout << "List After Deletion: ";
    printList(head);
    return 0; }
```

1. **Define Stack**

   A Stack is an ordered list in which all insertions (Push operation) and deletion (Pop operation) are made at one end, called the top. The topmost element is pointed by top. The top is initialized to -1 when the stack is created that is when the stack is empty. In a stack S = (a1,…an), a1 is the bottom most element and element ai is on top of element ai-1. Stack is also referred as Last In First Out (LIFO) list.

2. **What are the various Operations performed on the Stack?**

   The various operations that are performed on the stack are

   CREATE(S) – Creates S as an empty stack.
   PUSH(S,X) – Adds the element X to the top of the stack.
   POP(S) – Deletes the top most elements from the stack.
   TOP(S) – returns the value of top element from the stack.
   ISEMTPTY(S) – returns true if Stack is empty else false.
   ISFULL(S) - returns true if Stack is full else false.

3. **How do you test for an empty stack?**

   The condition for testing an empty stack is top =-1, where top is the pointer pointing to the topmost element of the stack, in the array implementation of stack.In linked list implementation of stack the condition for an empty stack is the header node link field is NULL.

4. **Name two applications of stack? (Nov/Dec 2018)**

   Nested and Recursive functions can be implemented using stack.Conversion of Infix to Postfix expression can be implemented using stack.Evaluation of Postfix expression can be implemented using stack.

5. **Define a suffix expression.**

   The notation used to write the operator at the end of the operands is called suffix notation.
   Suffix notation format : operand operand operator
   Example: ab+, where a & b are operands and '+' is addition operator.

6. **What do you meant by fully parenthesized expression? Give eg.**

   A pair of parentheses has the same parenthetical level as that of the operator to which it corresponds. Such an expression is called fully parenthesized expression.

   Ex: (a+((b*c) + (d * e))

7. **Write the postfix form for the expression -A+B-C+D?**

   A-B+C-D+

8. **What are the postfix and prefix forms of the expression?**

   A+B*(C-D)/(P-R)

   Postfix form: ABCD-*PR-/+
   Prefix form: +A/*B-CD-PR

9. **Explain the usage of stack in recursive algorithm implementation?**

   In recursive algorithms, stack data structures is used to store the return address when a recursive call is encountered and also to store the values of all the parameters essential to the current state of the function.

10. **Define Queues.**

A Queue is an ordered list in which all insertions take place at one end called the rear, while all deletions take place at the other end called the front. Rear is initialized to -1 and front is initialized to 0. Queue is also referred as First In First Out (FIFO) list.

11. What are the various operations performed on the Queue?

The various operations performed on the queue are
CREATE(Q) – Creates Q as an empty Queue.
Enqueue(Q,X) – Adds the element X to the Queue.
Dequeue(Q) – Deletes a element from the Queue.
ISEMTPTY(Q) – returns true if Queue is empty else false.
ISFULL(Q) - returns true if Queue is full else false.

12. How do you test for an empty Queue?

The condition for testing an empty queue is rear=front-1. In linked list implementation of queue the condition for an empty queue is the header node link field is NULL.

13. Write down the function to insert an element into a queue, in which the queue is implemented as an array. (May 10)

Q – Queue
X – element to added to the queue Q
IsFull(Q) – Checks and true if Queue Q is full
Q->Size - Number of elements in the queue Q
Q->Rear – Points to last element of the queue Q
Q->Array – array used to store queue elements

```
void enqueue (int X, Queue Q) {
        if(IsFull(Q))
          Error ("Full queue");
        else      {
                Q->Size++;
                  Q->Rear = Q->Rear+1;
                  Q->Array[ Q->Rear ]=X;
                }
        }
```

14. Define Deque.

Deque stands for Double ended queue. It is a linear list in which insertions and deletion are made from either end of the queue structure.

15. Define Circular Queue.

Another representation of a queue, which prevents an excessive use of memory by arranging elements/ nodes $Q_1, Q_2, \ldots Q_n$ in a circular fashion. That is, it is the queue, which wraps around upon reaching the end of the queue

16. What are priority queue? What are the ways to implement priority queue? (Nov/Dec 2018)

To make all of the operations very efficient, we'll use a new data structure called a heap. Consider **implementing** a **priority queue** using an array, a linked list, or

a BST. For each, describe how each of the **priority queue** operations would be**implemented**, and what the worst-case time would be.

Part – B

1. Write an algorithm for Push and Pop operations on Stack using Linked list. (8)
   Implement a stack using singly linked list

Implement a stack using single linked list concept. all the single linked list operations perform based on Stack operations LIFO(last in first out) and with the help of that knowledge we are going to implement a stack using single linked list. using single linked lists so how to implement here it is linked list means what we are storing the information in the form of nodes and we need to follow the stack rules and we need to implement using single linked list nodes so what are the rules we need to follow in the implementation of a stack a simple rule that is last in first out and all the operations we should perform so with the help of a top variable only with the help of top variables are how to insert the elements let's see
A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. which is "head" of the stack where pushing and popping items happens at the head of the list. first node have null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.

The main advantage of using linked list over an arrays is that it is possible to implements a stack that can shrink or grow as much as needed. In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocate. so overflow is not possible.

Stack Operations:

1. **Push() :** Insert the element into linked list nothing but which is the top node of Stack.
2. **Pop() :** Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
3. **peek():** Return the top element.
4. **display():** Print all element of Stack.

```
// C program to Implement a stack
//using singly linked list
#include <stdio.h>
#include <stdlib.h> // Declare linked list node
struct Node {
        int data;
        struct Node* link;
};
struct Node* top;

void push(int data)
{
        // create new node temp and allocate memory
        struct Node* temp;
```

```
        temp = (struct Node*)malloc(sizeof(struct Node));

        // check if stack (heap) is full. Then inserting an element would
        // lead to stack overflow
        if (!temp) {
                printf("\nHeap Overflow");
        exit(1); }
        temp->data = data;     // put top pointer reference into temp link
        temp->link = top;
                top = temp; }
int isEmpty()
{
        return top == NULL;
}
int peek()
{
                if (!isEmpty(top))
                return top->data;
        else
                exit(EXIT_FAILURE);
}
void pop()
{
        struct Node* temp;
                if (top == NULL) {
                printf("\nStack Underflow");
                exit(1); }
        else {
                                temp = top;
                                top = top->link;
                                temp->link = NULL;
                                free(temp); }}
void display() // remove at the beginning
{
        struct Node* temp;

        // check for stack underflow
        if (top == NULL) {
                printf("\nStack Underflow");
                exit(1);
        }
        else {
                temp = top;
                while (temp != NULL) {

                                                printf("%d->", temp->data);

                        // assign temp link to temp
                        temp = temp->link;
                }
        }
```

```
}

// main function

int main(void)
{
        // push the elements of stack
        push(11);
        push(22);
        push(33);
        push(44);

        // display stack elements
        display();

        // print top elementof stack
        printf("\nTop element is %d\n", peek());

        // delete top elements of stack
        pop();
        pop();

        // display stack elements
        display();

        // print top elementof stack
        printf("\nTop element is %d\n", peek());
        return 0;
}
```

2. Explain the linked list implementation of stack ADT in detail?

```
/**
 * Stack implementation using linked list in C language.
 */

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>     // For INT_MIN

#define CAPACITY 10000  // Stack maximum capacity

// Define stack node structure
struct stack
{
    int data;
    struct stack *next;
} *top;

// Stack size
int size = 0;
```

```c
/* Function declaration to perform push and pop on stack */
void push(int element);
int  pop();


int main()
{
    int choice, data;

    while(1)
    {
        /* Menu */
        printf("-------------------------------------\n");
        printf("    STACK IMPLEMENTATION PROGRAM    \n");
        printf("-------------------------------------\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Size\n");
        printf("4. Exit\n");
        printf("-------------------------------------\n");
        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                printf("Enter data to push into stack: ");
                scanf("%d", &data);

                // Push element to stack
                push(data);
                break;

            case 2:
                data = pop();

                // If stack is not empty
                if (data != INT_MIN)
                    printf("Data => %d\n", data);
                break;

            case 3:
                printf("Stack size: %d\n", size);
                break;

            case 4:
                printf("Exiting from app.\n");
                exit(0);
                break;
```

```
            default:
                printf("Invalid choice, please try again.\n");
        }

        printf("\n\n");
    }


    return 0;
}



/**
 * Functiont to push a new element in stack.
 */
void push(int element)
{
    // Check stack overflow
    if (size >= CAPACITY)
    {
        printf("Stack Overflow, can't add more element to stack.\n");
        return;
    }

    // Create a new node and push to stack
    struct stack * newNode = (struct stack *) malloc(sizeof(struct stack));

    // Assign data to new node in stack
    newNode->data = element;

    // Next element after new node should be current top element
    newNode->next = top;

    // Make sure new node is always at top
    top = newNode;

    // Increase element count in stack
    size++;

    printf("Data pushed to stack.\n");
}


/**
 * Function to pop element from top of stack.
 */
int pop()
{
    int data = 0;
    struct stack * topNode;

    // Check stack underflow
```

```
    if (size <= 0 || !top)
    {
        printf("Stack is empty.\n");

        // Throw empty stack error/exception
        // Since C does not have concept of exception
        // Hence return minimum integer value as error value
        // Later in code check if return value is INT_MIN, then
        // stack is empty
        return INT_MIN;
    }

    // Copy reference of stack top to some temp variable
    // Since we need to delete current stack top and make
    // Stack top its next element
    topNode = top;

    // Copy data from stack's top element
    data = top->data;

    // Move top to its next element
    top = top->next;

    // Delete the previous top most stack element from memory
    free(topNode);

    // Decrement stack size
    size--;

    return data;
}
```

Output

```
-----------------------------------
    STACK IMPLEMENTATION PROGRAM
-----------------------------------
1. Push
2. Pop
3. Size
4. Exit
-----------------------------------
Enter your choice: 2
Stack is empty.


-----------------------------------
    STACK IMPLEMENTATION PROGRAM
-----------------------------------
```

3. Define an efficient representation of two stacks in a given area of memory with n words and explain.

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.
push1(int x) –> pushes x to first stack
push2(int x) –> pushes x to second stack

pop1() –> pops an element from first stack and return the popped element
pop2() –> pops an element from second stack and return the popped element

Implementation of *twoStack* should be space efficient.

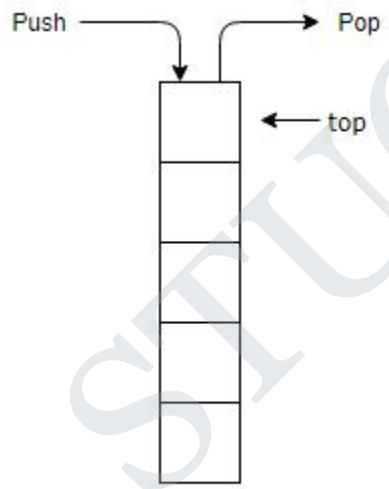
```
#include<iostream>
#include<stdlib.h>

using namespace std;

class twoStacks
{
        int *arr;
        int size;
        int top1, top2;
public:
twoStacks(int n) // constructor
{
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
}

// Method to push an element x to stack1
void push1(int x)
{
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
                top1++;
                arr[top1] = x;
        }
        else
        {
                cout << "Stack Overflow";
                exit(1);
        }
}

// Method to push an element x to stack2
void push2(int x)
{
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
                top2--;
```

```
                arr[top2] = x;
        }
        else
        {
                cout << "Stack Overflow";
                exit(1);
        }
}

// Method to pop an element from first stack
int pop1()
{
        if (top1 >= 0 )
        {
                int x = arr[top1];
                top1--;
                return x;
        }
        else
        {
                cout << "Stack UnderFlow";
                exit(1);
        }
}

// Method to pop an element from second stack
int pop2()
{
        if (top2 < size)
        {
                int x = arr[top2];
                top2++;
                return x;
        }
        else
        {
                cout << "Stack UnderFlow";
                exit(1);
        }
}
};


/* Driver program to test twStacks class */
int main()
{
        twoStacks ts(5);
        ts.push1(5);
        ts.push2(10);
        ts.push2(15);
        ts.push1(11);
```

```
        ts.push2(7);
        cout << "Popped element from stack1 is " << ts.pop1();
        ts.push2(40);
        cout << "\nPopped element from stack2 is " << ts.pop2();
        return 0;
}
```

4. Explain linear linked implementation of Stack and Queue?
    a. write an ADT to implement stack of size N using an array. The elements in the stack are to be integers. The operations to be supported are PUSH, POP and DISPLAY. Take into account the exceptions of stack overflow and stack underflow. (8)
    b. A circular queue has a size of 5 and has 3 elements 10,20 and 40 where F=2 and R=4. After inserting 50 and 60, what is the value of F and R. Trying to insert 30 at this stage what happens? Delete 2 elements from the queue and insert 70, 80 & 90. Show the sequence of steps with necessary diagrams with the value of F & R. (8 Marks)

A Stack is a linear data structure which allows adding and removing of elements in a particular order. New elements are added at the top of Stack. If we want to remove an element from the Stack, we can only remove the top element from Stack. Since it allows insertion and deletion from only one end and the element to be inserted last will be the element to be deleted first, hence it is called Last in First Out data structure (LIFO).



Stack Data Structure

Here we will define three operations on Stack,

- Push - it specifies adding an element to the Stack. If we try to insert an element when the Stack is full, then it is said to be Stack Overflow condition
- Pop - it specifies removing an element from the Stack. Elements are always removed from top of Stack. If we try to perform pop operation on an empty Stack, then it is said to be Stack Underflow condition.
- Peek - it will show the element on the top of Stack(without removing it).

Implementing Stack functionalities using Linked List

Stack can be implemented using both, arrays and linked list. The limitation in case of array is that we need to define the size at the beginning of the implementation. This makes our Stack static. It can also result in *"Stack overflow"* if we try to add elements after the array is full. So, to alleviate this problem, we use linked list to implement the Stack so that it can grow in real time.

First, we will create our Node class which will form our Linked List. We will be using this same Node class to implement the Queue also in the later part of this article.

Push an element into Stack

Now, our Stack and Node class is ready. So, we will proceed to Push operation on Stack. We will add a new element at the top of Stack.

Algorithm

- Create a new node with the value to be inserted.
- If the Stack is empty, set the next of the new node to null.
- If the Stack is not empty, set the next of the new node to top.
- Finally, increment the top to point to the new node.

The time complexity for *Push* operation is O(1). The method for Push will look like this.

Pop an element from Stack

We will remove the top element from Stack.

Algorithm

- If the Stack is empty, terminate the method as it is Stack underflow.
- If the Stack is not empty, increment top to point to the next node.
- Hence the element pointed by top earlier is now removed.

The time complexity for Pop operation is O(1). The method for Pop will be like following.

Peek the element from Stack

The peek operation will always return the top element of Stack without removing it from Stack.

Algorithm

- If the Stack is empty, terminate the method as it is Stack underflow.
- If the Stack is not empty, return the element pointed by the top.

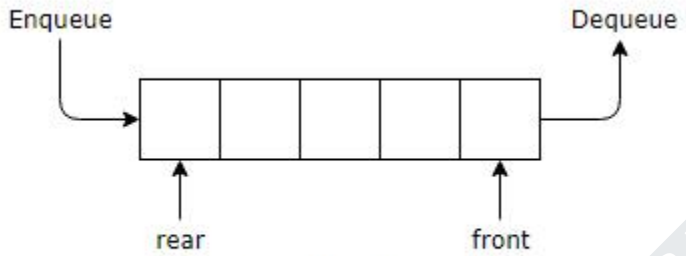
The time complexity for Peek operation is O(1). The Peek method will be like following.

Uses of Stack

- Stack can be used to implement back/forward button in the browser.
- Undo feature in the text editors are also implemented using Stack.
- It is also used to implement recursion.
- Call and return mechanism for a method uses Stack.
- It is also used to implement backtracking.

What is Queue?

A Queue is also a linear data structure where insertions and deletions are performed from two different ends. A new element is added from the rear of Queue and deletion of existing element occurs from the front. Since we can access elements from both ends and the element inserted first will be the one to be deleted first, hence Queue is called First in First Out data structure (FIFO).



Queue data structure

Here, we will define two operations on Queue.

- Enqueue - It specifies the insertion of a new element to the Queue. Enqueue will always take place from the rear end of the Queue.
- Dequeue - It specifies the deletion of an existing element from the Queue. Dequeue will always take place from the front end of the Queue.

Implementing Queue functionalities using Linked List

Similar to Stack, the Queue can also be implemented using both, arrays and linked list. But it also has the same drawback of limited size. Hence, we will be using a Linked list to implement the Queue.
The Node class will be the same as defined above in Stack implementation.
We will define LinkedListQueue class as below.
Here, we have taken two pointers - rear and front - to refer to the rear and the front end of the Queue respectively and will initialize it to null.

Enqueue of an Element

We will add a new element to our Queue from the rear end.

Algorithm

- Create a new node with the value to be inserted.
- If the Queue is empty, then set both front and rear to point to newNode.
- If the Queue is not empty, then set next of rear to the new node and the rear to point to the new node.

The time complexity for Enqueue operation is O(1). The Method for *Enqueue* will be like the following.

Dequeue of an Element

We will delete the existing element from the Queue from the front end.

Algorithm

- If the Queue is empty, terminate the method.
- If the Queue is not empty, increment front to point to next node.
- Finally, check if the front is null, then set rear to null also. This signifies empty Queue.

The time complexity for Dequeue operation is O(1). The Method for *Dequeue* will be like following.

Uses of Queue

- CPU scheduling in Operating system uses Queue. The processes ready to execute and the requests of CPU resources wait in a queue and the request is served on first come first serve basis.
- Data buffer - a physical memory storage which is used to temporarily store data while it is being moved from one place to another is also implemented using Queue.

5. Write the algorithm for converting infix expression to postfix (polish) expression?**(Nov/Dec 18)**

```
/* This program converts infix expression to postfix expression.
 * This program assume that there are Five operators: (*, /, +, -,^)
        in infix expression and operands can be of single-digit only.
 * This program will not work for fractional numbers.
 * Further this program does not check whether infix expression is
 valid or not in terms of number of operators and operands.*/

#include<stdio.h>
#include<stdlib.h>     /* for exit() */
#include<ctype.h>     /* for isdigit(char ) */
#include<string.h>

#define SIZE 100


/* declared here as global variable because stack[]
* is used by more than one fucntions */
char stack[SIZE];
int top = -1;

/* define push operation */

void push(char item)
{
        if(top >= SIZE-1)
        {
                printf("\nStack Overflow.");
        }
        else
        {
                top = top+1;
```

```
                stack[top] = item;
        }
}

/* define pop operation */
char pop()
{
        char item ;

        if(top <0)
        {
                printf("stack under flow: invalid infix expression");
                getchar();
                /* underflow may occur for invalid expression */
                /* where ( and ) are not matched */
                exit(1);
        }
        else
        {
                item = stack[top];
                top = top-1;
                return(item);
        }
}

/* define function that is used to determine whether any symbol is operator or not
(that is symbol is operand)
* this fucntion returns 1 if symbol is opreator else return 0 */

int is_operator(char symbol)
{
        if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol
=='-')
        {
                return 1;
        }
        else
        {
        return 0;
        }
}

/* define fucntion that is used to assign precendence to operator.
* Here ^ denotes exponent operator.
* In this fucntion we assume that higher integer value
* means higher precendence */

int precedence(char symbol)
{
        if(symbol == '^')/* exponent operator, highest precedence*/
        {
```

```
                return(3);
        }
        else if(symbol == '*' || symbol == '/')
        {
                return(2);
        }
        else if(symbol == '+' || symbol == '-')         /* lowest precedence */
        {
                return(1);
        }
        else
        {
                return(0);
        }
}

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
        int i, j;
        char item;
        char x;

        push('(');                              /* push '(' onto stack */
        strcat(infix_exp,")");                  /* add ')' to infix expression */

        i=0;
        j=0;
        item=infix_exp[i];          /* initialize before loop*/

        while(item != '\0')         /* run loop till end of infix expression */
        {
                if(item == '(')
                {
                        push(item);
                }
                else if( isdigit(item) || isalpha(item))
                {
                        postfix_exp[j] = item;                  /* add operand symbol to
postfix expr */
                        j++;
                }
                else if(is_operator(item) == 1)         /* means symbol is operator */
                {
                        x=pop();
                        while(is_operator(x)    ==    1    &&    precedence(x)>=
precedence(item))
                        {
                                postfix_exp[j] = x;                     /* so pop all higher
precendence operator and */
                                j++;
```

```
                        x = pop();                      /* add them to postfix
expresion */
                }
                push(x);
                /* because just above while loop will terminate we have
                oppped one extra item
                for which condition fails and loop terminates, so that one*/

                push(item);             /* push current oprerator symbol
onto stack */
        }
        else if(item == ')')        /* if current symbol is ')' then */
        {
                x = pop();              /* pop and keep popping until */
                while(x != '(')          /* '(' encounterd */
                {
                        postfix_exp[j] = x;
                        j++;
                        x = pop();
                }
        }
        else
        { /* if current symbol is neither operand not '(' nor ')' and nor
                operator */
                printf("\nInvalid infix Expression.\n");        /* the it is illegeal
symbol */
                getchar();
                exit(1);
        }
        i++;


        item = infix_exp[i]; /* go to next symbol of infix expression */
} /* while loop ends here */
if(top>0)
{
        printf("\nInvalid infix Expression.\n");       /* the it is illegeal  symbol
*/
        getchar();
        exit(1);
}
if(top>0)
{
        printf("\nInvalid infix Expression.\n");       /* the it is illegeal  symbol
*/
        getchar();
        exit(1);
}


postfix_exp[j] = '\0'; /* add sentinel else puts() fucntion */
```

```
        /* will print entire postfix[] array upto SIZE */

}

/* main function begins */
int main()
{
        char infix[SIZE], postfix[SIZE];        /* declare infix string and postfix string
*/

        /* why we asked the user to enter infix expression
        * in parentheses ( )
        * What changes are required in porgram to
        * get rid of this restriction since it is not
        * in algorithm
        * */
        printf("ASSUMPTION: The infix expression contains single letter variables
and single digit constants only.\n");
        printf("\nEnter Infix expression : ");
        gets(infix);

        InfixToPostfix(infix,postfix);              /* call to convert */
        printf("Postfix Expression: ");
        puts(postfix);                  /* print postfix expression */

        return 0;
}
```

6. Explain in detail about priority queue ADT in detail?

   Priority Queue | Set 1 (Introduction)

   Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

   In the below priority queue, element with maximum ASCII value will have the highest priority.

   A typical priority queue supports following operations.

   insert(item, priority): Inserts an item with given priority.

   getHighestPriority(): Returns the highest priority item.

   deleteHighestPriority(): Removes the highest priority item.

   How to implement priority queue?

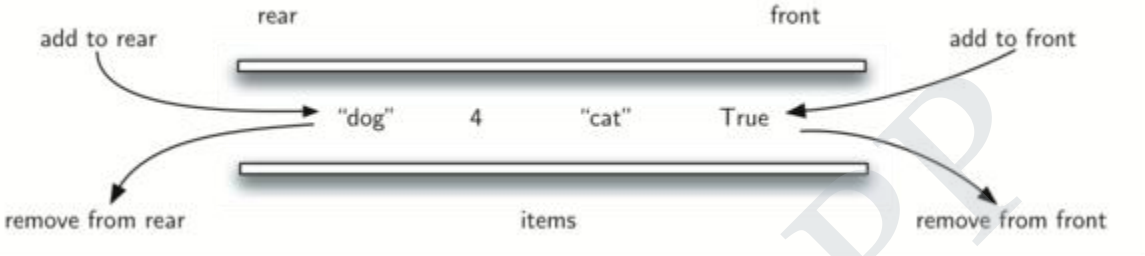
   Using Array: A simple implementation is to use array of following structure.

7. What is a DeQueue? Explain its operation with example?

A **deque**, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be

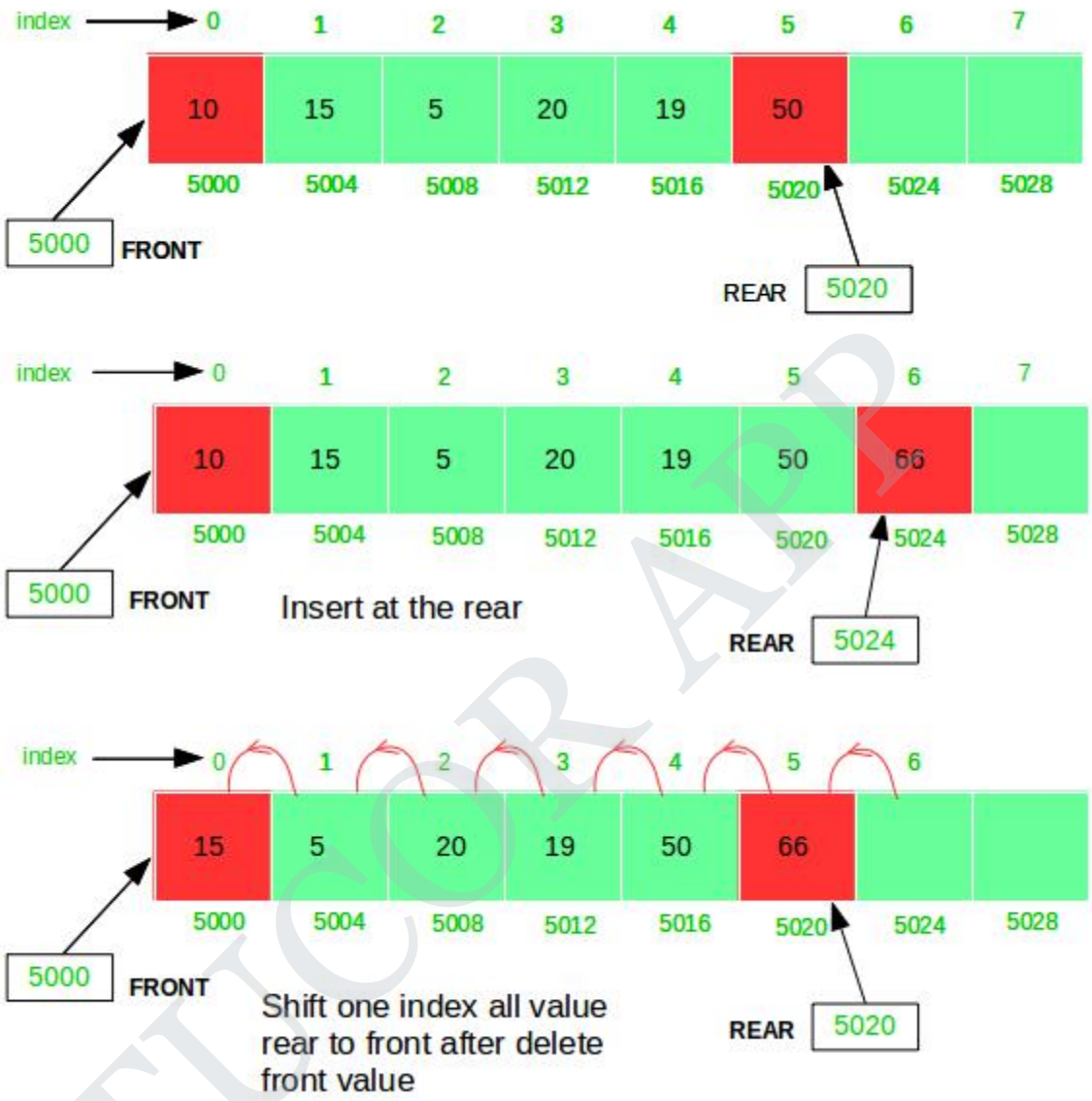
removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. Figure 1 shows a deque of Python data objects.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.



8. Explain the array implementation of queue ADT in detail?
   In queue, insertion and deletion happen at the opposite ends, so implementation is not as simple as stack.
   To implement a queue using array, create an array *arr* of size *n* and take two variables *front*and *rear* both of which will be initialized to *0* which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

1. **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear* < *n* which indicates that the array is not full then store the element at *arr[rear]* and increment *rear* by *1* but if *rear* == *n* then it is said to be an Overflow condition as the array is full.
2. **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear* > *0*. Now, element at *arr[front]* can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.
3. **Front:** Get the front element from the queue i.e. *arr[front]* if queue is not empty.
4. **Display:** Print all element of the queue. If the queue is non-empty, traverse and print all the elements from index *front* to *rear*.



Below is the implementation of a queue using an array:

```
// C++ program to implement a queue using an array
#include <bits/stdc++.h>
using namespace std;

struct Queue {
        int front, rear, capacity;
        int* queue;
        Queue(int c)
        {
                front = rear = 0;
                capacity = c;
                queue = new int;
        }

        ~Queue() { delete[] queue; }
```

```
// function to insert an element
// at the rear of the queue
void queueEnqueue(int data)
{
	// check queue is full or not
	if (capacity == rear) {
		printf("\nQueue is full\n");
		return;
	}

	// insert element at the rear
	else {
		queue[rear] = data;
		rear++;
	}
	return;
}

// function to delete an element
// from the front of the queue
void queueDequeue()
{
	// if queue is empty
	if (front == rear) {
		printf("\nQueue is empty\n");
		return;
	}

	// shift all the elements from index 2 till rear
	// to the left by one
	else {
		for (int i = 0; i < rear - 1; i++) {
			queue[i] = queue[i + 1];
		}

		// decrement rear
		rear--;
	}
	return;
}

// print queue elements
void queueDisplay()
{
	int i;
	if (front == rear) {
		printf("\nQueue is Empty\n");
		return;
	}
```

```
                // traverse front to rear and print elements
                for (i = front; i < rear; i++) {
                        printf(" %d <-- ", queue[i]);
                }
                return;
        }

        // print front of queue
        void queueFront()
        {
                if (front == rear) {
                        printf("\nQueue is Empty\n");
                        return;
                }
                printf("\nFront Element is: %d", queue[front]);
                return;
        }
};

// Driver code
int main(void)
{
        // Create a queue of capacity 5
        Queue q(4);

        // print Queue elements
        q.queueDisplay();

        // inserting elements in the queue
        q.queueEnqueue(20);
        q.queueEnqueue(30);
        q.queueEnqueue(40);
        q.queueEnqueue(50);

        // print Queue elements
        q.queueDisplay();

        // insert element in the queue
        q.queueEnqueue(60);

        // print Queue elements
        q.queueDisplay();

        q.queueDequeue();
        q.queueDequeue();

        printf("\n\nafter two node deletion\n\n");

        // print Queue elements
        q.queueDisplay();
```

```
        // print front of the queue
        q.queueFront();

        return 0;
}
```

9. Explain the addition and deletion operations performed on a circular queue with necessary algorithms.(8) **(Nov/Dec 18)**

Circular queue avoids the wastage of space in a regular queue implementation using arrays.

As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.

The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e.

if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

Queue operations work as follows:

- Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
- When initializing the queue, we set the value of FRONT and REAR to -1.
- On enqueing an element, we circularly increase the value of REAR index and place the new element in the position pointed to by REAR.
- On dequeueing an element, we return the value pointed to by FRONT and circularly increase the FRONT index.
- Before enqueing, we check if queue is already full.
- Before dequeuing, we check if queue is already empty.
- When enqueing the first element, we set the value of FRONT to 0.
- When dequeing the last element, we reset the values of FRONT and REAR to -1.

However, the check for full queue has a new additional case:

- Case 1: FRONT = 0 && REAR == SIZE - 1
- Case 2: FRONT = REAR + 1

The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

UNIT- III

1. Define tree?

Trees are non-liner data structure, which is used to store data items in a shorted sequence. It represents any hierarchical relationship between any data Item. It is a collection of nodes, which has a distinguish node called the root and zero or more non-empty sub trees T1, T2,….Tk. each of which are connected by a directed edge from the root.

2. Define Height of tree? (Nov/Dec 2018)

The height of n is the length of the longest path from root to a leaf. Thus all leaves have height zero. The height of a tree is equal to a height of a root.

3. Define Depth of tree?

For any node n, the depth of n is the length of the unique path from the root to node n. Thus for a root the depth is always zero.

4. What is the length of the path in a tree?

The length of the path is the number of edges on the path. In a tree there is exactly one path form the root to each node.

5. Define sibling? (Nov/Dec 2018)

Nodes with the same parent are called siblings.

6. Define binary tree?

A Binary tree is a finite set of data items which is either empty or consists of a single item called root and two disjoin binary trees called left sub tree max degree of any node is two.

7. What are the two methods of binary tree implementation?

Two methods to implement a binary tree are,

a. Linear representation.

b. Linked representation

8. What are the applications of binary tree?

Binary tree is used in data processing.

a. File index schemes

b. Hierarchical database management system

9. List out few of the Application of tree data-structure?

Ø The manipulation of Arithmetic expression

Ø Used for Searching Operation

Ø Used to implement the file system of several popular operating systems

Ø Symbol Table construction

Ø Syntax analysis

10. Define expression tree?

Expression tree is also a binary tree in which the leafs terminal nodes or operands and non-terminal intermediate nodes are operators used for traversal.

11. Define tree traversal and mention the type of traversals?

Visiting of each and every node in the tree exactly is called as tree traversal.

Three types of tree traversal

1. Inorder traversal
2. Preoder traversal
3. Postorder traversal.

12. Define in -order traversal?

In-order traversal entails the following steps;

a. Traverse the left subtree

b. Visit the root node

c. Traverse the right subtree

13. Define threaded binary tree.

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.

14. What are the types of threaded binary tree?

Right-in threaded binary tree
Left-in threaded binary tree
Fully-in threaded binary tree

15. Define Binary Search Tree. (Nov/Dec 2018)

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X and the values of all the keys in its right subtree are larger than the key value in X.

16.What is AVL Tree?

AVL stands for Adelson-Velskii and Landis.An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

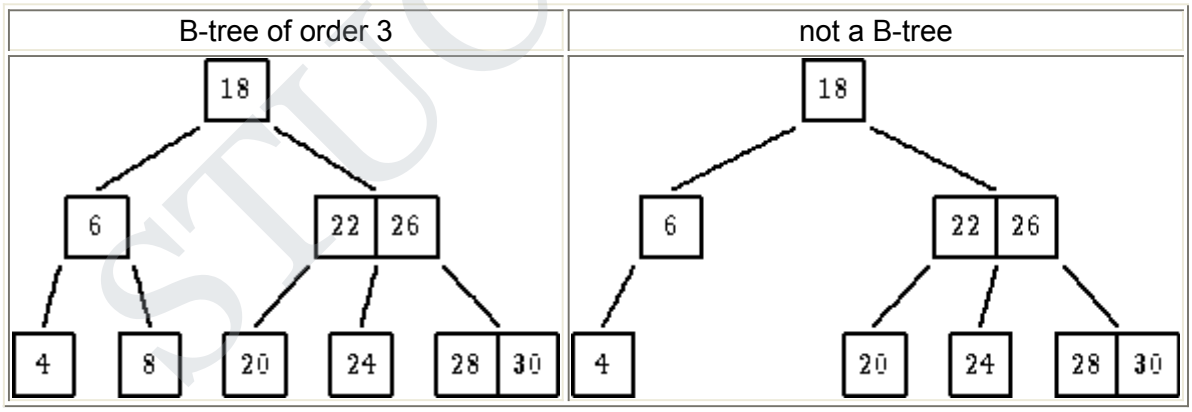
Search time is O(logn). Addition and deletion operations also take O(logn) time.

17. List out the steps involved in deleting a node from a binary search tree.

Deleting a node is a leaf node (ie) No children
Deleting a node with one child.
Deleting a node with two Childs.

18. What is 'B' Tree?

A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.



Important properties of a B-tree:

- B-tree nodes have many more than two children.
- A B-tree node may contain more than just a single element.

19. What is binomial heaps?

A binomial heap is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent

.
The number of trees in a binomial heap is O(log n).

20. Define complete binary tree.
If all its levels, possible except the last, have maximum number of nodes and if all the nodes in the last level appear as far left as possible.

21. State the complexity of binary search?

Class	**Search algorithm**
Best-case performance	O(1)
Average performance	O(log n)
Worst-case space complexity	O(1)

PART - B

1. **Explain the AVL tree insertion and deletion with suitable example.**
   Tree is one of the most important data structure that is used for efficiently performing operations like insertion, deletion and searching of values. However, while working with a large volume of data, construction of a well-balanced tree for sorting all data s not feasible. Thus only useful data is stored as a tree, and the actual volume of data being used continually changes through the insertion of new data and deletion of existing data. You will find in some cases where the NULL link to a binary tree to special links is called as threads and hence it is possible to perform traversals, insertions, deletions without using either stack or recursion. In this chapter, you will learn about the Height balance tree which is also known as the AVL tree.
   AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree.
   An AVL tree can be defined as follows:
   Let T be a non-empty binary tree with T_L and T_R as its left and right subtrees. The tree is height balanced if:

- T_L and T_R are height balanced

- h_L - h_R <= 1, where h_L - h_R are the heights of T_L and T_R

The Balance factor of a node in a binary tree can have value 1, -1, 0, depending on whether the height of its left subtree is greater, less than or equal to the height of the right subtree.

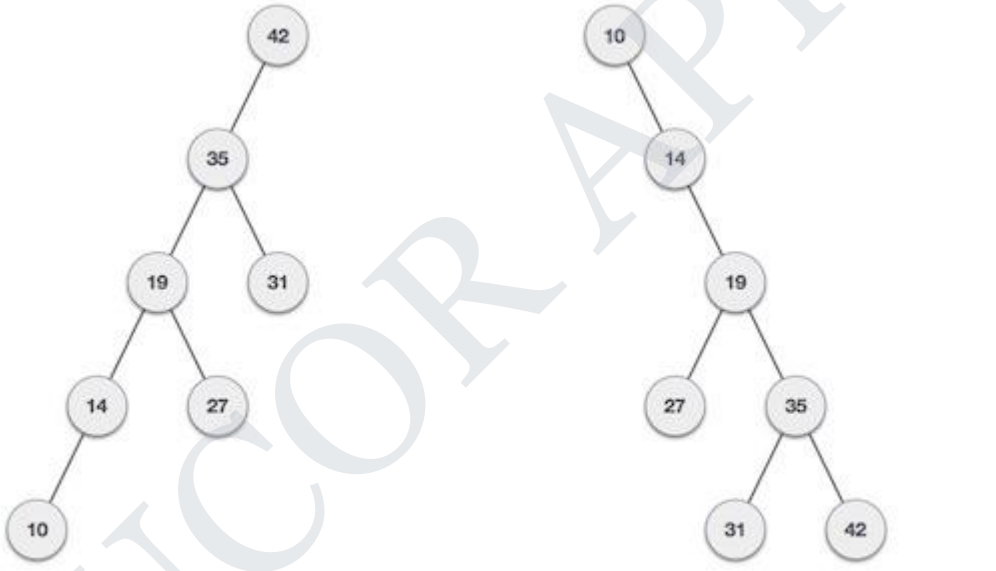
Advantages of AVL tree

Since AVL trees are height balance trees, operations like insertion and deletion have low time complexity. Let us consider an example:

If you have the following tree having keys 1, 2, 3, 4, 5, 6, 7 and then the binary tree will be like the second figure:

2. Describe the algorithms used to perform single and double rotation on AVL tree.

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



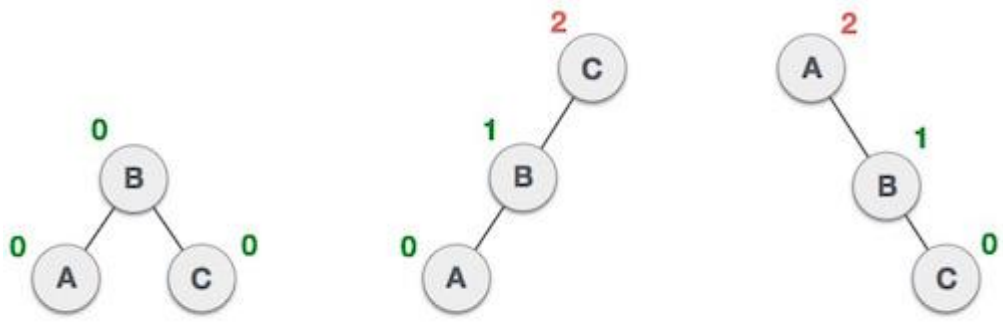
If input 'appears' non-increasing manner

If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Here we see that the first tree is balanced and the next two trees are not balanced −



Balanced Not balanced Not balanced

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

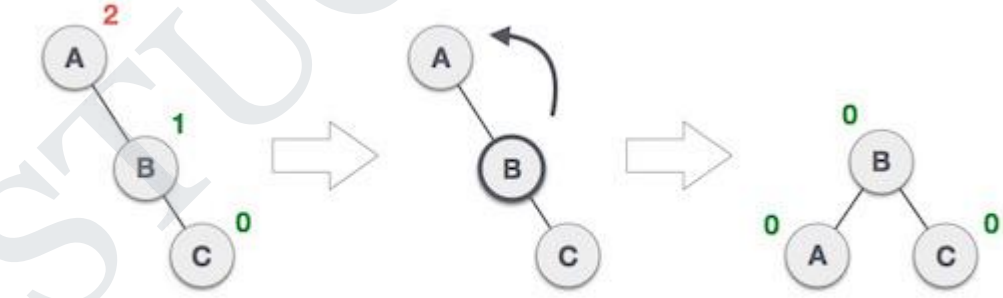
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −

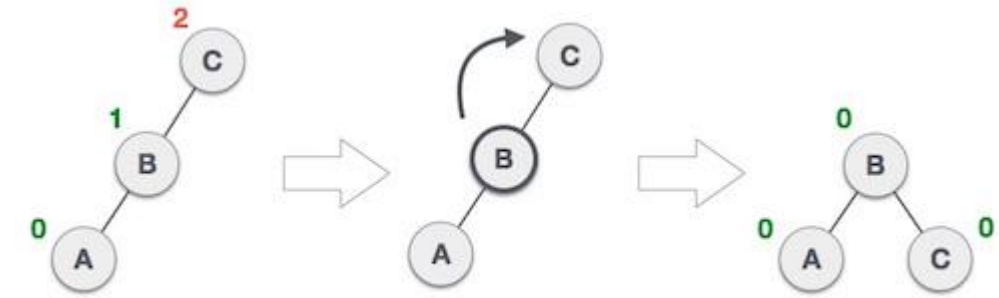


Right unbalanced tree Left Rotation Balanced

In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making Athe left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

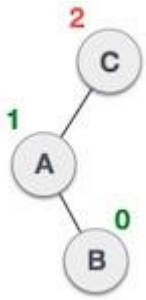
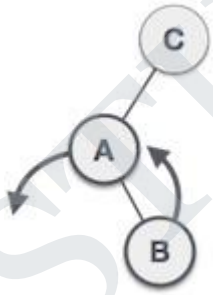
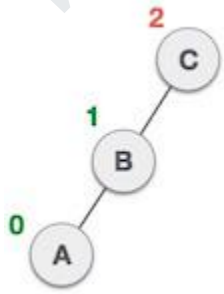


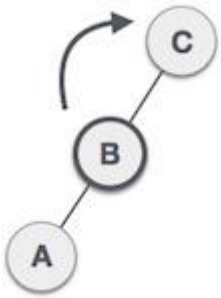
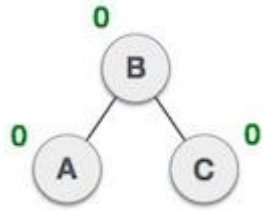
Left unbalanced Tree Right Rotation Balanced Tree

As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

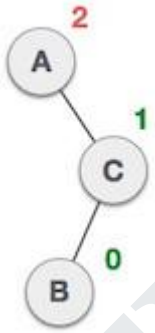
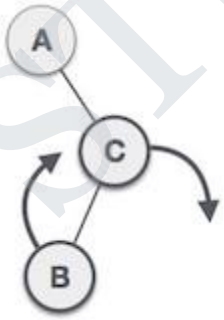
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

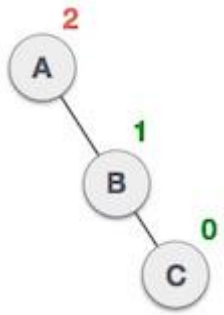
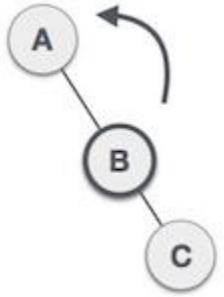
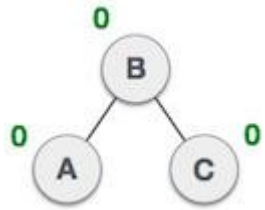
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

	We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.
	First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.

	Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.
	The tree is now balanced.

3. Explain about B-Tree with suitable example.

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

Properties of B-Tree

1) All leaves are at same level.

2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.

3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

4) All nodes (including root) may contain at most 2t – 1 keys.

5) Number of children of a node is equal to the number of keys in it plus 1.

6) All keys of a node are sorted in increasing order. The child between two keys k1 and

k2 contains all keys in the range from k1 and k2.
7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

4. Explain about B+ trees with suitable algorithm.(Nov/Dec 18)

In computer science, a **B+ tree** is a type of tree data structure. It represents sorted data in a way that allows for efficient insertion and removal of elements. It is a dynamic, multilevel index with maximum and minimum bounds on the number of keys in each node.
A B+ tree is a variation on a B-tree. In a B+ tree, in contrast to a B tree, all data are saved in the leaves. Internal nodes contain only keys and tree pointers. All leaves are at the same lowest level. Leaf nodes are also linked together as a linked list to make range queries easy.
The maximum number of keys in a record is called the order of the B+ tree.
The minimum number of keys per record is 1/2 of the maximum number of keys. For example, if the order of a B+ tree is *n*, each node (except for the root) must have between *n*/2 and *n* keys.
The number of keys that may be indexed using a B+ tree is a function of the order of the tree and its height.
For a *n*-order B+ tree with a height of *h*:

- maximum number of keys is
- minimum number of keys is

5. Write short notes on

1) Binomial heaps

The main application of Binary Heap is as implement priority queue. Binomial Heap is an extension of Binary Heap that provides faster union or merge operation together with other operations provided by Binary Heap.
A Binomial Heap is a collection of Binomial Trees
What is a Binomial Tree?
A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one as leftmost child or other.
A Binomial Tree of order k has following properties.
a) It has exactly 2k nodes.
b) It has depth as k.
c) There are exactly kCi nodes at depth i for i = 0, 1, . . . , k.
d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

2) Fibonacci heaps

Heaps are mainly used for implementing priority queue. We have discussed below heaps in previous posts.

Binary Heap
Binomial Heap

In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are amortized time complexities of Fibonacci Heap.

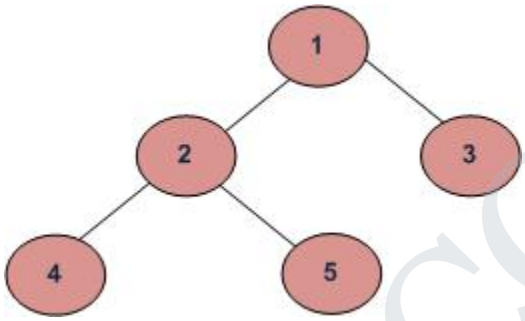
1) Find Min: Θ(1) [Same as both Binary and Binomial]
2) Delete Min: O(Log n) [Θ(Log n) in both Binary and Binomial]
3) Insert: Θ(1) [Θ(Log n) in Binary and Θ(1) in Binomial]
4) Decrease-Key: Θ(1) [Θ(Log n) in both Binary and Binomial]
5) Merge: Θ(1) [Θ(m Log n) or Θ(m+n) in Binary and Θ(Log n) in Binomial]

Like Binomial Heap, Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).

6. Explain the tree traversal techniques with an example.

Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5
Please see this post for Breadth First Traversal.

Inorder Traversal (Practice**):**

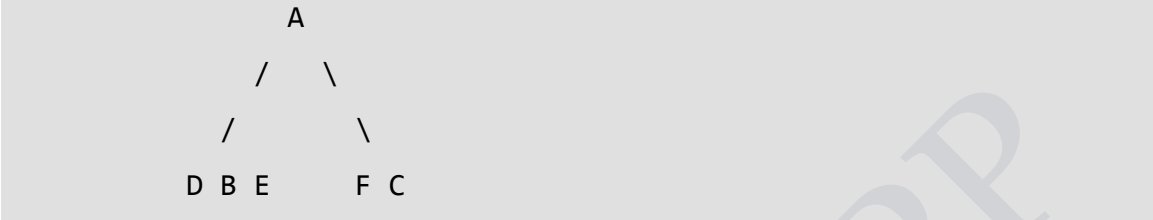
Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

7. **Construct an expression tree for the expression (a+b*c) + ((d*e+f)*g). Give the outputs when you apply inorder, preorder and postorder traversals.**

In a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now.



We recursively follow above steps and get the following tree.

Algorithm: buildTree()

1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.

2) Create a new tree node tNode with the data as picked element.

3) Find the picked element's index in Inorder. Let the index be inIndex.

4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.

5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.

6) return tNode.

8. How to insert and delete an element into a binary search tree and write down the code for the insertion routine with an example.

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

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

struct node* createNode(value){
    struct node* newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}



struct node* insert(struct node* root, int data)
{
    if (root == NULL) return createNode(data);

    if (data < root->data)
        root->left  = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}

void inorder(struct node* root){
    if(root == NULL) return;
```

```
    inorder(root->left);
    printf("%d ->", root->data);
    inorder(root->right);
}


int main(){
    struct node *root = NULL;
    root = insert(root, 8);
    insert(root, 3);
    insert(root, 1);
    insert(root, 6);
    insert(root, 7);
    insert(root, 10);
    insert(root, 14);
    insert(root, 4);

    inorder(root);
}
```

9. **What are threaded binary tree? Write an algorithm for inserting a node in a threaded binary tree.**

```
// Insertion in Threaded Binary Search Tree.
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    bool lthread;

    // True if right pointer points to predecessor
    // in Inorder Traversal
```

```
        bool rthread;
};

// Insert a Node in Binary Threaded Tree
struct Node *insert(struct Node *root, int ikey)
{
        // Searching for a Node with given value
        Node *ptr = root;
        Node *par = NULL; // Parent of key to be inserted
        while (ptr != NULL)
        {
          // If key already exists, return
          if (ikey == (ptr->info))
          {
                    printf("Duplicate Key !\n");
                    return root;
          }

          par = ptr; // Update parent pointer

          // Moving on left subtree.
          if (ikey < ptr->info)
          {
                    if (ptr -> lthread == false)
                              ptr = ptr -> left;
                    else
                              break;
          }

          // Moving on right subtree.
          else
          {
                    if (ptr->rthread == false)
                              ptr = ptr -> right;
                    else
                              break;
          }
        }

        // Create a new node
        Node *tmp = new Node;
        tmp -> info = ikey;
        tmp -> lthread = true;
        tmp -> rthread = true;

        if (par == NULL)
        {
          root = tmp;
          tmp -> left = NULL;
          tmp -> right = NULL;
        }
```

```
        else if (ikey < (par -> info))
        {
          tmp -> left = par -> left;
          tmp -> right = par;
          par -> lthread = false;
          par -> left = tmp;
        }
        else
        {
          tmp -> left = par;
          tmp -> right = par -> right;
          par -> rthread = false;
          par -> right = tmp;
        }

        return root;
}

// Returns inorder successor using rthread
struct Node *inorderSuccessor(struct Node *ptr)
{
        // If rthread is set, we can quickly find
        if (ptr -> rthread == true)
          return ptr->right;

        // Else return leftmost child of right subtree
        ptr = ptr -> right;
        while (ptr -> lthread == false)
          ptr = ptr -> left;
        return ptr;
}

// Printing the threaded tree
void inorder(struct Node *root)
{
        if (root == NULL)
          printf("Tree is empty");

        // Reach leftmost node
        struct Node *ptr = root;
        while (ptr -> lthread == false)
          ptr = ptr -> left;

        // One by one print successors
        while (ptr != NULL)
        {
          printf("%d ",ptr -> info);
          ptr = inorderSuccessor(ptr);
        }
}
```

```
// Driver Program
int main()
{
	struct Node *root = NULL;

	root = insert(root, 20);
	root = insert(root, 10);
	root = insert(root, 30);
	root = insert(root, 5);
	root = insert(root, 16);
	root = insert(root, 14);
	root = insert(root, 17);
	root = insert(root, 13);

	inorder(root);

	return 0;
}
```

10. **Create a binary search tree for the following numbers start from an empty binary search tree. 45,26,10,60,70,30,40 Delete keys 10,60 and 45 one after the other and show the trees at each stage.(Nov/Dec 18)**

```
// Insertion in Threaded Binary Search Tree.
#include<bits/stdc++.h>
using namespace std;

struct Node
{
	struct Node *left, *right;
	int info;

	// True if left pointer points to predecessor
	// in Inorder Traversal
	bool lthread;

	// True if right pointer points to predecessor
	// in Inorder Traversal
	bool rthread;
};

// Insert a Node in Binary Threaded Tree
struct Node *insert(struct Node *root, int ikey)
{
	// Searching for a Node with given value
	Node *ptr = root;
	Node *par = NULL; // Parent of key to be inserted
	while (ptr != NULL)
	{
	  // If key already exists, return
	  if (ikey == (ptr->info))
	  {
```

```
                printf("Duplicate Key !\n");
                return root;
        }

        par = ptr; // Update parent pointer

        // Moving on left subtree.
        if (ikey < ptr->info)
        {
                if (ptr -> lthread == false)
                        ptr = ptr -> left;
                else
                        break;
        }

        // Moving on right subtree.
        else
        {
                if (ptr->rthread == false)
                        ptr = ptr -> right;
                else
                        break;
        }
}

// Create a new node
Node *tmp = new Node;
tmp -> info = ikey;
tmp -> lthread = true;
tmp -> rthread = true;

if (par == NULL)
{
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
}
else if (ikey < (par -> info))
{
        tmp -> left = par -> left;
        tmp -> right = par;
        par -> lthread = false;
        par -> left = tmp;
}
else
{
        tmp -> left = par;
        tmp -> right = par -> right;
        par -> rthread = false;
        par -> right = tmp;
}
```

```
        return root;
}

// Returns inorder successor using rthread
struct Node *inorderSuccessor(struct Node *ptr)
{
        // If rthread is set, we can quickly find
        if (ptr -> rthread == true)
          return ptr->right;

        // Else return leftmost child of right subtree
        ptr = ptr -> right;
        while (ptr -> lthread == false)
          ptr = ptr -> left;
        return ptr;
}

// Printing the threaded tree
void inorder(struct Node *root)
{
        if (root == NULL)
          printf("Tree is empty");

        // Reach leftmost node
        struct Node *ptr = root;
        while (ptr -> lthread == false)
          ptr = ptr -> left;

        // One by one print successors
        while (ptr != NULL)
        {
          printf("%d ",ptr -> info);
          ptr = inorderSuccessor(ptr);
        }
}

// Driver Program
int main()
{
        struct Node *root = NULL;

        root = insert(root, 20);
        root = insert(root, 10);
        root = insert(root, 30);
        root = insert(root, 5);
        root = insert(root, 16);
        root = insert(root, 14);
        root = insert(root, 17);
        root = insert(root, 13);
```

```
        inorder(root);

        return 0;
}
```

UNIT- IV
PART A

1. **Write the definition of weighted graph?**
   A graph in which weights are assigned to every edge is called a weighted graph.
2. **Define Graph? (Nov/Dec 2018)**
   A graph G consist of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set of edges E to set of pairs of elements of V. It can also be represented as G=(V, E).
3. **Define adjacency matrix?**
The adjacency matrix is an n x n matrix A whose elements aij are given by
aij = 1 if (vi, vj) Exists =0 otherwise
4. **Define adjacent nodes?**
   Any two nodes, which are connected by an edge in a graph, are called adjacent nodes. For example, if an edge x E is associated with a pair of nodes
(u,v) where u, v V, then we say that the edge x connects the nodes u and v.
5. **What is a directed graph?**
   A graph in which every edge is directed is called a directed graph.
6. **What is an undirected graph?**
   A graph in which every edge is undirected is called an undirected graph.
7. **What is a loop?**
   An edge of a graph, which connects to itself, is called a loop or sling.
8. **What is a simple graph?**
   A simple graph is a graph, which has not more than one edge between a pair of nodes.
9. **What is a weighted graph?**
   A graph in which weights are assigned to every edge is called a weighted graph.
10. **Define indegree and out degree of a graph?**
    In a directed graph, for any node v, the number of edges, which have v as their initial node, is called the out degree of the node v.
    Outdegree: Number of edges having the node v as root node is the outdegree of the node v.
11. **Define path in a graph?**
    The path in a graph is the route taken to reach terminal node from a starting node.
12. **What is a simple path?**
    A path in a diagram in which the edges are distinct is called a simple path.
It is also called as edge simple.
13. **What is a cycle or a circuit?**
    A path which originates and ends in the same node is called a cycle or circuit.
14. **What is an acyclic graph?**
    A simple diagram, which does not have any cycles, is called an acyclic graph.
15. **What is meant by strongly connected in a graph?**

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

16. When a graph said to be weakly connected?

When a directed graph is not strongly connected but the underlying graph is connected, then the graph is said to be weakly connected.

17. Name the different ways of representing a graph? Give examples (Nov 10)

a. Adjacency matrix
b. Adjacency list

18. What is an undirected acyclic graph?

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph. It is also called as undirected forest.

19. What is meant by depth?

The depth of a list is the maximum level attributed to any element with in the list or with in any sub list in the list.

20. What is the use of BFS?

BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph. The shortest distance is the minimum number of edges traversed in order to travel from the start node the specific node being examined.

21. What is topological sort?

It is an ordering of the vertices in a directed acyclic graph, such that: If there is a path from u to v, then v appears after u in the ordering.

22. Write BFS algorithm

1. Initialize the first node's dist number and place in queue
2. Repeat until all nodes have been examined
3. Remove current node to be examined from queue
4. Find all unlabeled nodes adjacent to current node
5. If this is an unvisited node label it and add it to the queue
6. Finished.

23. Define biconnected graph?

A graph is called biconnected if there is no single node whose removal causes the graph to break into two or more pieces. A node whose removal causes the graph to become disconnected is called a cut vertex.

24. What are the two traversal strategies used in traversing a graph?

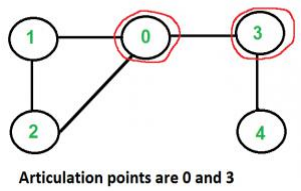
a. Breadth first search
b. Depth first search

25. **Articulation Points (or Cut Vertices) in a Graph**

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

Following are some example graphs with articulation points encircled with red color.



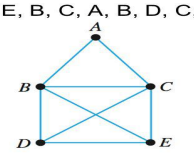
Articulation points are 0 and 3

26. An Euler path is a path that uses every edge of agraph exactly once. An Euler circuit is a circuit that uses every edge of a graph exactly once. ▸ An Eulerpath starts and ends at different vertices. ▸ An Euler circuit starts and ends at the same vertex. (Nov/Dec 2018)

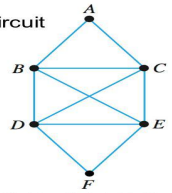
Examples

- Euler path

D, E, B, C, A, B, D, C, E



- Euler circuit

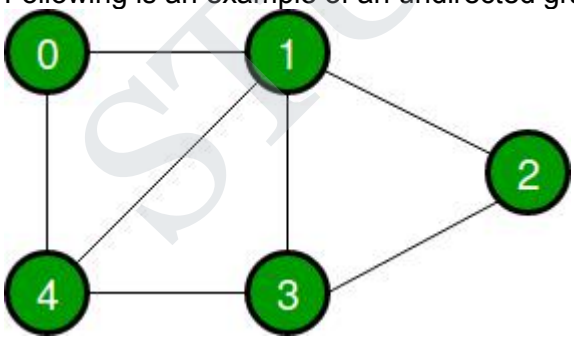


D, E, B, C, A, B, D, C, E, F, D

PART-B

1. Explain the various representation of graph with example in detail?

Graph is a data structure that consists of following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See this for more applications of graph.
Following is an example of an undirected graph with 5 vertices.



Following two are the most commonly used representations of a graph.
1. Adjacency Matrix
2. Adjacency List
There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.

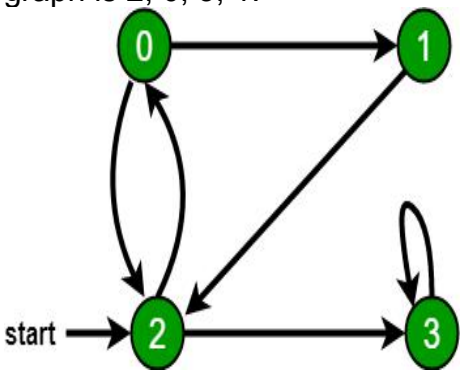
Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

```
// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v)
    {
            cout << "\n Adjacency list of vertex "
                 << v << "\n head ";
            for (auto x : adj[v])
            cout << "-> " << x;
            printf("\n");
    }
}

// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    printGraph(adj, V);
    return 0;
}
```

2. Explain Breadth First Search algorithm with example? (Nov/Dec 18)

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For

simplicity, it is assumed that all vertices are reachable from the starting vertex. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.




```
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V; // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
```

```
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
```

```
        g.addEdge(3, 3);

        cout << "Following is Breadth First Traversal "
                << "(starting from vertex 2) \n";
        g.BFS(2);

        return 0;
    }
```

3. Explain Depth first and breadth first traversal? (Nov/Dec 18)

Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visits each node before its children. It is most easy to program as a recursive routine:

```
preorder(node v)
{
visit(v);
for each child w of v
   preorder(w);
}
```

To turn this into a graph traversal algorithm, we basically replace "child" by "neighbor". But to prevent infinite loops, we only want to visit each vertex once. Just like in BFS we can use marks to keep track of the vertices that have already been visited, and not visit them again. Also, just like in BFS, we can use this search to build a spanning tree with certain useful properties.

```
dfs(vertex v)
{
visit(v);
for each neighbor w of v
   if w is unvisited
   {
   dfs(w);
   add edge vw to tree T
   }
}
```

The overall depth first search algorithm then simply initializes a set of markers so we can tell which vertices are visited, chooses a starting vertex x, initializes tree T to x, and calls dfs(x). Just like in breadth first search, if a vertex has several neighbors it would be equally correct to go through them in any order. I didn't simply say "for each unvisited neighbor of v" because it is very important to delay the test for whether a vertex is visited until the recursive calls for previous neighbors are finished.

The proof that this produces a spanning tree (the depth first search tree) is essentially the same as that for BFS, so I won't repeat it. However while the BFS tree is typically "short and bushy", the DFS tree is typically "long and stringy".

Just like we did for BFS, we can use DFS to classify the edges of G into types. Either an edge vw is in the DFS tree itself, v is an ancestor of w, or w is an ancestor of v. (These last two cases should be thought of as a single type, since they only differ by what order we look at the vertices in.) What this means is that if v and w are in different subtrees of v, we can't have an edge from v to w. This is because if such an edge existed and (say)

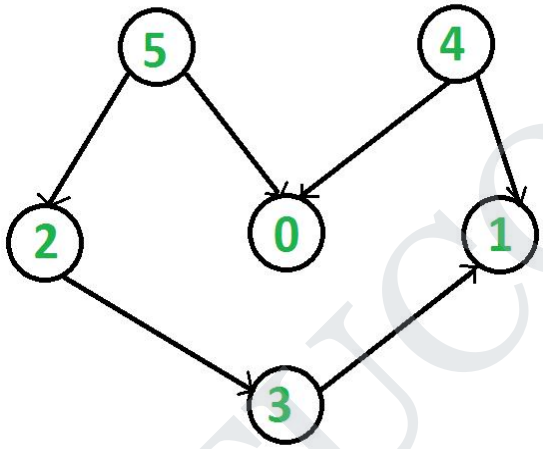
v were visited first, then the only way we would avoid adding vw to the DFS tree would be if w were visited during one of the recursive calls from v, but then v would be an ancestor of w.

As an example of why this property might be useful, let's prove the following fact: in any graph G, either G has some path of length at least k. or G has O(kn) edges.

4. What is topological sort? Write an algorithm to perform topological sort?(8) (Nov/Dec 18)

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS):
In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

Algorithm to find Topological Sorting:
We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

```
// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
```

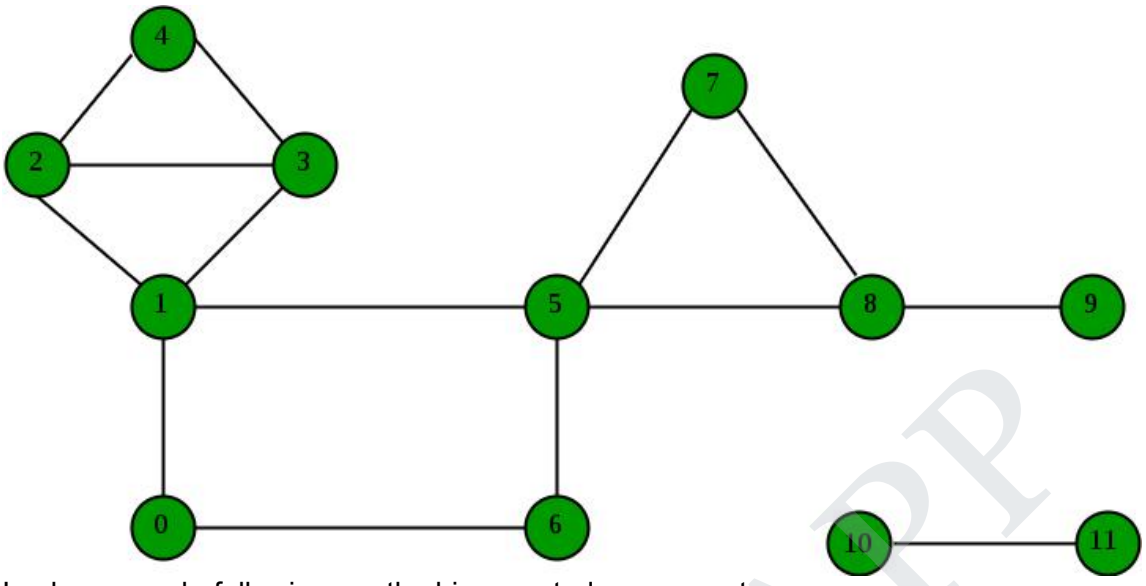
```
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
    int V; // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                                    stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                    topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}
```

```
// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
    if (visited[i] == false)
        topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given graph \n";
    g.topologicalSort();

    return 0;
}
```

5. (i) write an algorithm to determine the biconnected components in the given graph. (10) (may 10)

(ii)determine the biconnected components in a graph. (6)

A biconnected component is a maximal biconnected subgraph.
Biconnected Graph is already discussed here. In this article, we will see how to find biconnected component in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11

```
// A C++ program to find biconnected components in a given undirected graph
#include <iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;
int count = 0;
class Edge {
public:
        int u;
        int v;
        Edge(int u, int v);
};
Edge::Edge(int u, int v)
{
        this->u = u;
        this->v = v;
}

// A class that represents an directed graph
class Graph {
        int V; // No. of vertices
        int E; // No. of edges
        list<int>* adj; // A dynamic array of adjacency lists

        // A Recursive DFS based function used by BCC()
        void BCCUtil(int u, int disc[], int low[],
                                list<Edge>* st, int parent[]);
```

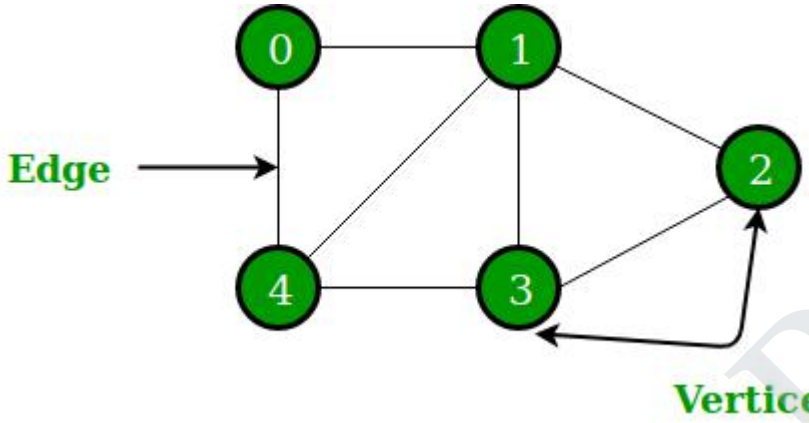
```
public:
        Graph(int V); // Constructor
        void addEdge(int v, int w); // function to add an edge to graph
        void BCC(); // prints strongly connected components
};

Graph::Graph(int V)
{
        this->V = V;
        this->E = 0;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w);
        E++;
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] -- >> earliest visited vertex (the vertex with minimum
// discovery time) that can be reached from subtree
// rooted with current vertex
// *st -- >> To store visited edges
void Graph::BCCUtil(int u, int disc[], int low[], list<Edge>* st,
                                        int parent[])
{
        // A static variable is used for simplicity, we can avoid use
        // of static variable by passing a pointer.
        static int time = 0;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;
        int children = 0;

        // Go through all vertices adjacent to this
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i) {
                int v = *i; // v is current adjacent of 'u'

                // If v is not visited yet, then recur for it
                if (disc[v] == -1) {
                        children++;
                        parent[v] = u;
                        // store the edge in stack
                        st->push_back(Edge(u, v));
                        BCCUtil(v, disc, low, st, parent);
```

```
                        // Check if the subtree rooted with 'v' has a
                        // connection to one of the ancestors of 'u'
                        // Case 1 -- per Strongly Connected Components Article
                        low[u] = min(low[u], low[v]);

                        // If u is an articulation point,
                        // pop all edges from stack till u -- v
                        if ((disc[u] == 1 && children > 1) || (disc[u] > 1 && low[v] >= disc[u]))
{
                                while (st->back().u != u || st->back().v != v) {
                                        cout << st->back().u << "--" << st->back().v << " ";
                                        st->pop_back();
                                }
                                cout << st->back().u << "--" << st->back().v;
                                st->pop_back();
                                cout << endl;
                                count++;
                        }
                }

                // Update low value of 'u' only of 'v' is still in stack
                // (i.e. it's a back edge, not cross edge).
                // Case 2 -- per Strongly Connected Components Article
                else if (v != parent[u]) {
                        low[u] = min(low[u], disc[v]);
                        if (disc[v] < disc[u]) {
                                st->push_back(Edge(u, v));
                        }
                }
        }
}

// The function to do DFS traversal. It uses BCCUtil()
void Graph::BCC()
{
        int* disc = new int[V];
        int* low = new int[V];
        int* parent = new int[V];
        list<Edge>* st = new list<Edge>[E];

        // Initialize disc and low, and parent arrays
        for (int i = 0; i < V; i++) {
                disc[i] = NIL;
                low[i] = NIL;
                parent[i] = NIL;
        }

        for (int i = 0; i < V; i++) {
                if (disc[i] == NIL)
                        BCCUtil(i, disc, low, st, parent);
```

```
                int j = 0;
                // If stack is not empty, pop all edges from stack
                while (st->size() > 0) {
                        j = 1;
                        cout << st->back().u << "--" << st->back().v << " ";
                        st->pop_back();
                }
                if (j == 1) {
                        cout << endl;
                        count++;
                }
        }
}

// Driver program to test above function
int main()
{
        Graph g(12);
        g.addEdge(0, 1);
        g.addEdge(1, 0);
        g.addEdge(1, 2);
        g.addEdge(2, 1);
        g.addEdge(1, 3);
        g.addEdge(3, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 2);
        g.addEdge(2, 4);
        g.addEdge(4, 2);
        g.addEdge(3, 4);
        g.addEdge(4, 3);
        g.addEdge(1, 5);
        g.addEdge(5, 1);
        g.addEdge(0, 6);
        g.addEdge(6, 0);
        g.addEdge(5, 6);
        g.addEdge(6, 5);
        g.addEdge(5, 7);
        g.addEdge(7, 5);
        g.addEdge(5, 8);
        g.addEdge(8, 5);
        g.addEdge(7, 8);
        g.addEdge(8, 7);
        g.addEdge(8, 9);
        g.addEdge(9, 8);
        g.addEdge(10, 11);
        g.addEdge(11, 10);
        g.BCC();
        cout << "Above are " << count << " biconnected components in graph";
        return 0;
}
```

6. Explain the various applications of Graphs.

A graph is a non-linear data structure, which consists of vertices(or nodes) connected by edges(or arcs) where edges may be directed or undirected.



- In **Computer science** graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

7. What are expression tree. Write the procedure for constructing an expression tree.

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:
Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)
Construction of Expression Tree:
Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.
1) If character is operand push that into stack
2) If character is operator pop two values from stack make them its child and

push current node again.
At the end only element of stack will be root of expression tree.

```
// C++ program for expression tree
#include<bits/stdc++.h>
using namespace std;

// An expression tree node
struct et
{
    char value;
    et* left, *right;
};

// A utility function to check if 'c'
// is an operator
bool isOperator(char c)
{
    if (c == '+' || c == '-' ||
                    c == '*' || c == '/' ||
                    c == '^')
        return true;
    return false;
}

// Utility function to do inorder traversal
void inorder(et *t)
{
    if(t)
    {
        inorder(t->left);
        printf("%c ", t->value);
        inorder(t->right);
    }
}

// A utility function to create a new node
et* newNode(int v)
{
    et *temp = new et;
    temp->left = temp->right = NULL;
    temp->value = v;
    return temp;
};

// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
    stack<et *> st;
    et *t, *t1, *t2;
```

```
        // Traverse through every character of
        // input expression
        for (int i=0; i<strlen(postfix); i++)
        {
                // If operand, simply push into stack
                if (!isOperator(postfix[i]))
                {
                        t = newNode(postfix[i]);
                        st.push(t);
                }
                else // operator
                {
                        t = newNode(postfix[i]);

                        // Pop two top nodes
                        t1 = st.top(); // Store top
                        st.pop();           // Remove top
                        t2 = st.top();
                        st.pop();

                        // make them children
                        t->right = t1;
                        t->left = t2;

                        // Add this subexpression to stack
                        st.push(t);
                }
        }

        // only element will be root of expression
        // tree
        t = st.top();
        st.pop();

        return t;
}

// Driver program to test above
int main()
{
        char postfix[] = "ab+ef*g*-";
        et* r = constructTree(postfix);
        printf("infix expression is \n");
        inorder(r);
        return 0;
}
```

UNIT – V

1. **What is meant by Sorting?**

Sorting is ordering of data in an increasing or decreasing fashion according to some linear relationship among the data items.

2. List the different sorting algorithms.

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Quick sort
- Radix sort
- Heap sort
- Merge sort

3. Why bubble sort is called so?

The bubble sort gets its name because as array elements are sorted they gradually "bubble" to their proper positions, like bubbles rising in a glass of soda.

4. State the logic of bubble sort algorithm.

The bubble sort repeatedly compares adjacent elements of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order.

5. What number is always sorted to the top of the list by each pass of the Bubble sort algorithm?

Each pass through the list places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs.

6. When does the Bubble Sort Algorithm stop?

The bubble sort stops when it examines the entire array and finds that no "swaps" are needed. The bubble sort keeps track of the occurring swaps by the use of a flag.

7. State the logic of selection sort algorithm.

It finds the lowest value from the collection and moves it to the left. This is repeated until the complete collection is sorted.

8. What is the output of selection sort after the 2nd iteration given the following sequence? 16 3 46 9 28 14

Ans: 3 9 46 16 28 14

8. How does insertion sort algorithm work?

In every iteration an element is compared with all the elements before it. While comparing if it is found that the element can be inserted at a suitable position, then space is created for it by shifting the other elements one position up and inserts the desired element at the suitable position. This procedure is repeated for all the elements in the list until we get the sorted elements.

9. What operation does the insertion sort use to move numbers from the unsorted section to the sorted section of the list?

The Insertion Sort uses the swap operation since it is ordering numbers within a single list.

11. How many key comparisons and assignments an insertion sort makes in its

worst case?

The worst case performance in insertion sort occurs when the elements of the input array are in descending order. In that case, the first pass requires one comparison, the second pass requires two comparisons, third pass three comparisons,….kth pass requires (k-1), and finally the last pass requires (n-1) comparisons. Therefore, total numbers of comparisons are:

f(n) = 1+2+3+……+(n-k)+…..+(n-2)+(n-1) = n(n-1)/2 = O(n2)

12. Which sorting algorithm is best if the list is already sorted? Why?

Insertion sort as there is no movement of data if the list is already sorted and complexity is of the order O(N).

13. Which sorting algorithm is easily adaptable to singly linked lists? Why?

Insertion sort is easily adaptable to singly linked list. In this method there is an array link of pointers, one for each of the original array elements. Thus the array can be thought of as a linear link list pointed to by an external pointer first initialized to 0. To insert the k^{th} element the linked list is traversed until the proper position for x[k] is found, or until the end of the list is reached. At that point x[k] can be inserted into the list by merely adjusting the pointers without shifting any elements in the array which reduces insertion time.

14. Why Shell Sort is known diminishing increment sort?

The distance between comparisons decreases as the sorting algorithm runs until the

last phase in which adjacent elements are compared. In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted.

15. Which of the following sorting methods would be especially suitable to sort a list L consisting of a sorted list followed by a few "random" elements?

Quick sort is suitable to sort a list L consisting of a sorted list followed by a few "random" elements.

10. Which sorting algorithm follows the divide-and-conquer strategy?

Quick sort and Merge sort

11. **What is the output of quick sort after the 3rd iteration given the following sequence?** 24 56 47 35 10 90 82 31

Pass 1:- (10) 24 (56 47 35 90 82 31)
Pass 2:- 10 24 (56 47 35 90 82 31)
Pass 3:- 10 24 (47 35 31) 56 (90 82)

12. Mention the different ways to select a pivot element.

The different ways to select a pivot element are

- Pick the first element as pivot
- Pick the last element as pivot
- Pick the Middle element as pivot
- Median-of-three elements
  - Pick three elements, and find the median x of these elements
  - Use that median as the pivot.
- Randomly pick an element as pivot.

13. What is divide-and-conquer strategy?

- Divide a problem into two or more sub problems
- Solve the sub problems recursively
- Obtain solution to original problem by combining these solutions

14. Compare quick sort and merge sort.

Quicksort has a best-case linear performance when the input is sorted, or nearly sorted. It has a worst-case quadratic performance when the input is sorted in reverse, or nearly sorted in reverse.Merge sort performance is much more constrained and predictable than the performance of quicksort. The price for that reliability is that the average case of merge sort is slower than the average case of quicksort because the constant factor of merge sort is larger.

15. What is the key idea of radix sort?

Sort the keys digit by digit, starting with the least significant digit to the most significant digit.

16. Define Searching.

Searching for data is one of the fundamental fields of computing. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set. Naturally, the use of a hash table or binary search tree will result in more efficient searching, but more often than not an array or linked list will be used. It is necessary to understand good ways of searching data structures not designed to support efficient search.

17. What is linear search?

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

24. What is Binary search?

A binary search, also called a dichotomizing search, is a digital scheme for locating a specific object in a large set. Each object in the set is given a key. The number of keys is always a power of 2. If there are 32 items in a list, for example, they might be numbered 0 through 31 (binary 00000 through 11111). If there are, say, only 29 items, they can be numbered 0 through 28 (binary 00000 through 11100), with the numbers 29 through31 (binary 11101, 11110, and 11111) as dummy keys.

26. Define hash function?

Hash function takes an identifier and computes the address of that identifier in the hash table using some function.

27. Why do we need a Hash function as a data structure as compared to any other data structure? (may 10)

Hashing is a technique used for performing insertions, deletions, and finds in constant average time.

28. What are the important factors to be considered in designing the hash function? (Nov 10)

- To avoid lot of collision the table size should be prime

- For string data if keys are very long, the hash function will take long to compute.

29. What are the problems in hashing?

a. Collision

b. Overflow

28. What do you mean by hash table?

The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value. Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell.

29. What do you mean by hash function?

A hash function is a key to address transformation which acts upon a given key to compute the relative position of the key in an array. The choice of hash function should be simple and it must distribute the data evenly. A simple hash function is hash_key=key mod tablesize.

30. What do you mean by separate chaining? (Nov/Dec 2018)

Separate chaining is a collision resolution technique to keep the list of all elements that hash to the same value. This is called separate chaining because each hash table element is a separate chain (linked list). Each linked list contains all the elements whose keys hash to the same index.

PART B

1. Write an algorithm to implement Bubble sort with suitable example.

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

```
// C program for implementation of Bubble sort
#include <stdio.h>
```

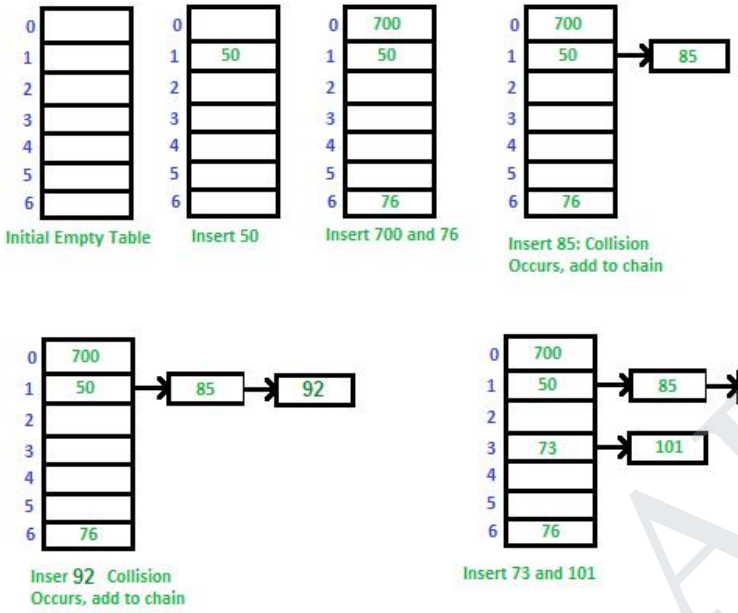
```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
int i, j;
for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                    swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
            printf("%d ", arr[i]);
    printf("n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

2. Explain any two techniques to overcome hash collision.

3. **Separate Chaining:**

   The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

4. Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:
1) Simple to implement.
2) Hash table never fills up, we can always add more elements to the chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:
1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2) Wastage of Space (Some Parts of hash table are never used)
3) If the chain becomes long, then search time can become O(n) in the worst case.
4) Uses extra space for links.

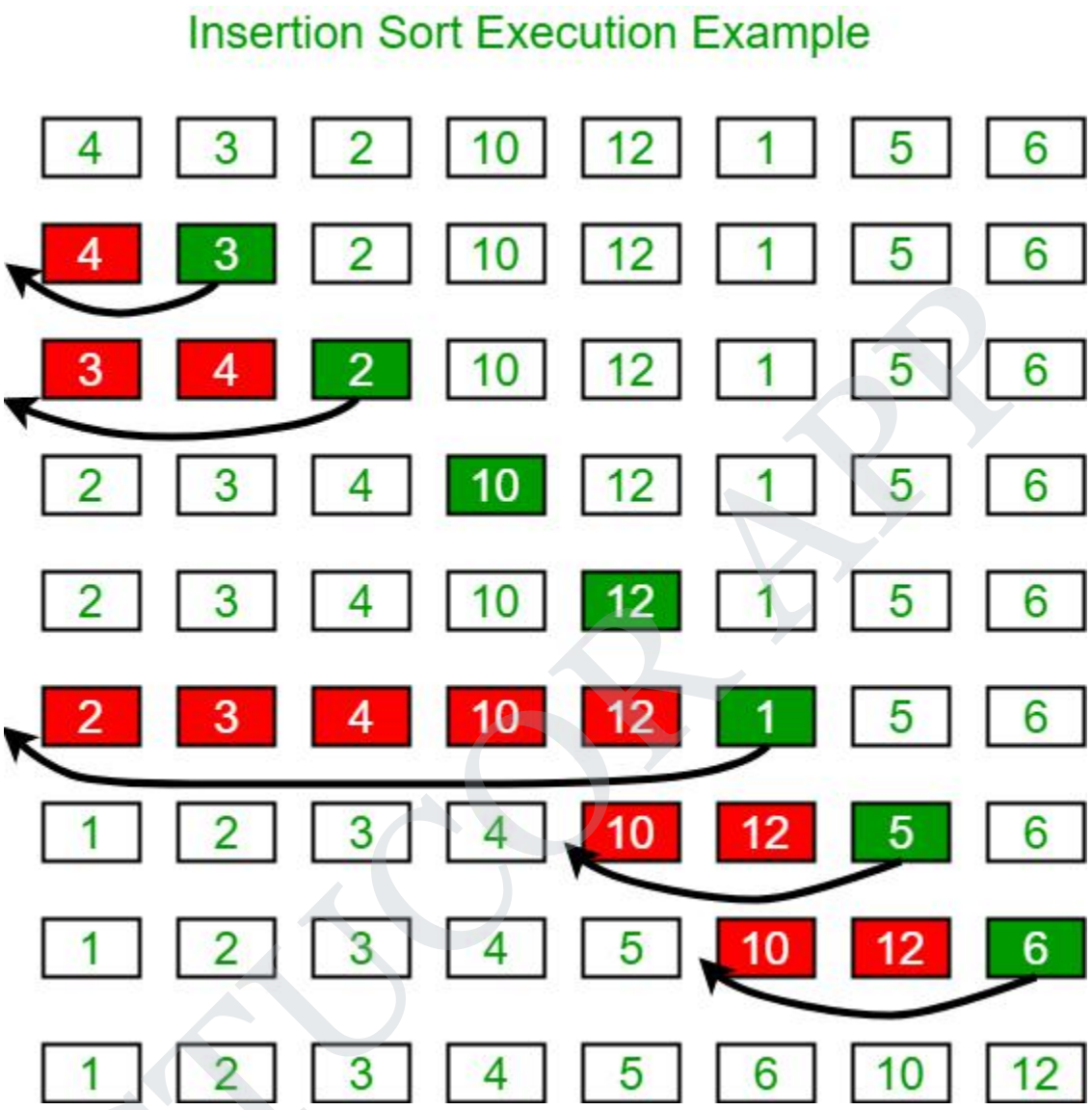
5. Write an algorithm to implement insertion sort with suitable example.

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Example:



6. Write an algorithm to implement selection sort with suitable example.

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```
// C++ program for implementation of selection sort
#include <bits/stdc++.h>
using namespace std;
```

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
            // Find the minimum element in unsorted array
            min_idx = i;
            for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                    min_idx = j;

            // Swap the found minimum element with the first element
            swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
            cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

7. Write an algorithm to implement radix sort with suitable example.

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.
Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

What if the elements are in range from 1 to n2?
We can't use counting sort because counting sort will take O(n2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?
Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The Radix Sort Algorithm

1) Do following for each digit i where i varies from least significant digit to the most significant digit.

……**a)** Sort input array using counting sort (or any stable sort) according to the i'th digit.

Example:

Original, unsorted list:
170, 45, 75, 90, 802, 24, 2, 66
Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]
170, 90, 802, 2, 24, 45, 75, 66

```
// C++ implementation of Radix Sort
#include<iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
        int mx = arr[0];
        for (int i = 1; i < n; i++)
                if (arr[i] > mx)
                        mx = arr[i];
        return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
        int output[n]; // output array
        int i, count[10] = {0};

        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
                count[ (arr[i]/exp)%10 ]++;

        // Change count[i] so that count[i] now contains actual
        // position of this digit in output[]
        for (i = 1; i < 10; i++)
                count[i] += count[i - 1];

        // Build the output array
        for (i = n - 1; i >= 0; i--)
        {
```

```
                output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
                count[ (arr[i]/exp)%10 ]--;
        }

        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to current digit
        for (i = 0; i < n; i++)
                arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
        // Find the maximum number to know number of digits
        int m = getMax(arr, n);

        // Do counting sort for every digit. Note that instead
        // of passing digit number, exp is passed. exp is 10^i
        // where i is current digit number
        for (int exp = 1; m/exp > 0; exp *= 10)
                countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
        for (int i = 0; i < n; i++)
                cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
        int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
        int n = sizeof(arr)/sizeof(arr[0]);
        radixsort(arr, n);
        print(arr, n);
        return 0;
}
```

8. Write an algorithm for binary search with suitable example. (Nov/Dec 18)

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.

- If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.

2. When a match is found, return the index of the element matched.
3. If no match is found, then return -1

9. Discuss the common collision resolution strategies used in closed hashing system.

closed hashing (open Addressing)

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

1. Liner Probing (this is prone to clustering of data + Some other constrains. Refer *Wiki*)
2. Quadratic probing (for more detail refer *Wiki*)
3. Double hashing (in short in case of collision another hashing function is used with the key value as an input to identify where in the open addressing scheme the data should actually be stored.)

A COMPARATIVE ANALYSIS OF CLOSED HASHING VS OPEN HASHING

Open Addressing	Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	No state data needs to be maintained (easier to maintain)
Uses space efficiently	Expensive on space

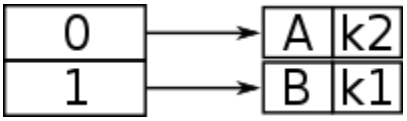
10. Explain Re-hashing and Extendible hashing.(Nov/Dec 18)

Extendible hashing is a type of hash system which treats a hash as a bit string and uses a trie for bucket lookup.[1] Because of the hierarchical nature of the system, re-hashing is an incremental operation (done one bucket at a time, as needed). This means that time-sensitive applications are less affected by table growth than by standard full-table rehashes.

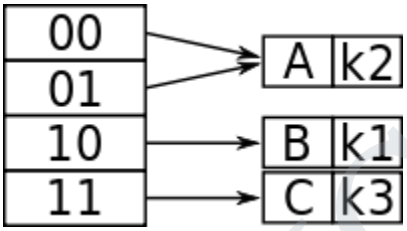
Extendible hashing was described by Ronald Fagin in 1979. Practically all modern filesystems use either extendible hashing or B-trees. In particular, the Global File System, ZFS, and the SpadFS filesystem use extendible hashing

Assume that the hash function returns a string of bits. The first i bits of each string will be used as indices to figure out where they will go in the "directory" (hash table). Additionally, i is the smallest number such that the index of every item in the table is unique.
Keys to be used:

Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k_1 and k_2, can be distinguished by the most significant bit, and would be inserted into the table as follows:



Now, if k_3 were to be hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because both k_3 and k_1 have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now k_1 and k_3 have a unique location, being distinguished by the first two leftmost bits. Because k_2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.