Solution/ model answer to VTU question paper – Dec 2018
3rd Sem-CSE/ISE - DSA – 17CS33

Dr. P. N. Singh, Professor(CSE), CMRIT, Bangalore

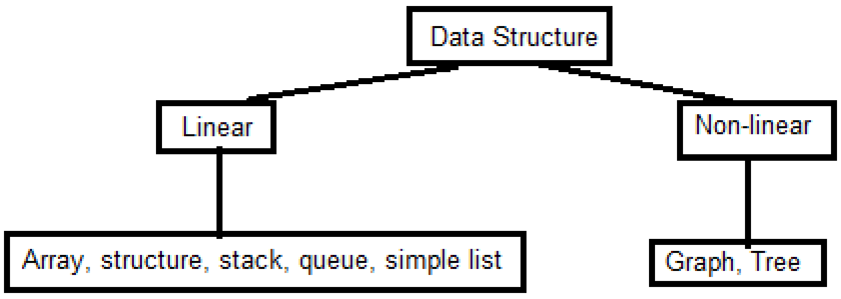
Module 1

Q 1 a. Define data structure. List & explain data structure operations. (5 marks)

Ans:
A structure represents name, type, size and format. A data structure is a represents to organize and store data in terms of name, type, size and & format in computers memory. Data may contain a single element or sometimes it may be a set of elements. Whether it is a single element or **multiple** elements but it must be organized in a particular way in the computer memory system.

General data structure types include the array, lists, vectors, the file, the record, the table, the tree, graphand so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and operated with appropriate ways.

Types of data structure:



Data Structure operations:

- Addition/Insertion: To add data at beginning or end or to insert data between two data
- Sorting: To sort data either in ascending or descending order to make the search operation faster
- Search: To search data within given collection by different ways
- Modification: To modify existing data with new data
- Deletion: To delete existing data

Q 1 b. Write the Bubble sort algorithm. (4 marks)

Ans:
In Bubble sort adjacent elements(1st with 2nd, 2nd with 3rd, 3rd with 4th, ...) are compared for swapping for order. In first pass "heaviest" highest/largest element is sunk down at the bottom most place. In second pass 2nd highest element goes to last second position. (This comparison may be started from last element towards first element also.) In each pass "lighter" element "bubbles up" and "heavier" item sinks down. For that the technique is known as Bubble sort.

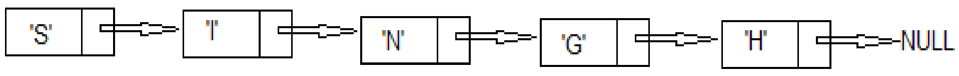
Algorithm for Bubble sort
Input array a of n elements

```
1. Start
2. For x = 0 to n-1
      a. For y = 0 to n-x-1
```

```
                    i.  If a[y] > a[y+1] then
                            1.  temp=a[y]
                            2.  a[y]=a[y+1]
                            3.  a[y+1]=temp
                    ii. Endif
            b.  Next y
        3.  Next x
        4.  End
Output sorted array a
```

Q 1 c. List & explain in detail, three types of structures used to store strings (10 Marks)

```
Structures used to store strings:
i.     Fixed length: Array of characters (where last character
       is put by default NULL):
       char name[] = 'SINGH';
       char name[5]={'S','I','N','G',H'};
ii.    Variable length: Pointer to strings
       char *name = "Dr. P. N. Singh";
iii.   struct type (character by character)
       struct str
       {
              char c;
        };
       sruct str name[5];
       name[0].c = 'S';
       name[1].c = 'I';
       name[2].c = 'N';
       name[3].c = 'G';
       name[4].c = 'H';

iv.    Structure may be extended to linked list

       struct str
       {
              char c;
               struct str *next;
       };
```



OR

Q 2 a. Explain Dynamic memory Allocation (5marks)

Ans:

Dynamic Memory allocation and de-allocation:
Memory can be allocated/ de-allocated at run-time as and when required. C has four in-built functions for the same with header file stdlib.h
malloc(), calloc() and realloc() to allocate and free() to de-allocate

malloc()
The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size in bytes and return a pointer of type void which can be casted into pointer of any form.
Syntax of malloc()
```
ptr=(cast-type*)malloc(byte-size);
```
example:
```
ptr=(int*)malloc(100*sizeof(int));
```

calloc()
The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.
Syntax for calloc():
```
ptr=(cast-type*)calloc(n,element-size);
```
example:
```
ptr=(int*)calloc(25,sizeof(int));
```

realloc()
If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().
Syntax/example:
```
ptr=realloc(ptr, newsize);
```

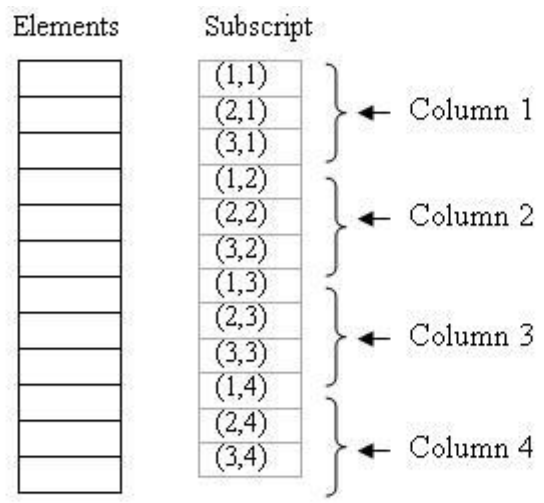
free()
Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.
Syntax/example:
```
free(ptr);
```

Q 2 b. Explain about representation of 2 dimensional arrays in memory (5 Marks)

Ans:
A 2D array's elements are stored in continuous memory locations. It can be represented in memory using any of the following two ways:

A. Column-Major Order
B Row-Major Order

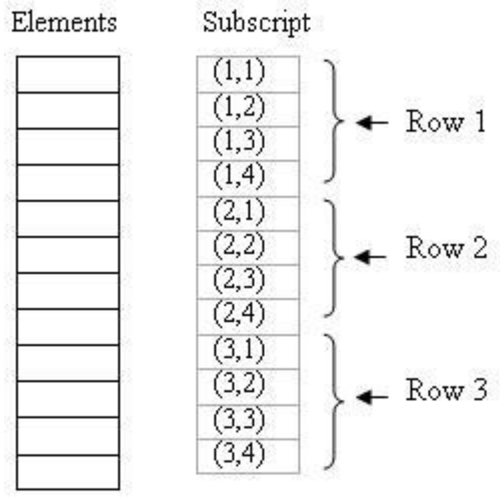
A. Column-Major Order:
In this method the elements are stored column wise, i.e. m elements of first column are stored in first m locations, m elements of second column are stored in next m locations and so on. E.g.
A 3 x 4 array will stored as below:



B. Row-Major Order:

In this method the elements are stored row wise, i.e. n elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on. E.g.

A 3 x 4 array will stored as below:



Q2c. What do you mean by pattern matching? Let P& T be strings with lengths R and S respectively and are stored as arrays with on character per element. Write a pattern matching algorithm that finds index P in T. Also discuss about this algorithm. (10 Marks)

Ans:

```
1   Start
2   Input string P[R], T[S] // P of length R and T of length S
3   For i = 1 to S-R   //Matching character from 1st position to R-Sth position
        3.1 j=1
        3.2 k=i
        3.3 Do while P[j] = T[k] AND J < R // loop matching up to length R
                3.3.1   k=k+1
                3.3.2   j=J+1
        3.4 Endwhile
```

```
            3.5  If j = R then        // if total number of matched character = R
                    3.5.1   found=TRUE
                    3.5.2   break  //terminate the loop if found
            3.6  ELSE
                    3.6.1   found=FALSE
            3.7  ENDIF
4   NEXT i  // again to start from next character of T
5   If found = TRUE then
            5.1  Print "index of p in T", i
6   Else
            6.1  Print "Not found"
7   END.
```

Discussion about algorithm:

This algorithm is known as "Brute Force Algorithm"

Implementation as dry run of the algorithm

Assume P is sub string of R characters i.e. 4 and T is string of S characters i.e 20. When first character P will match then we will continuously compare up to length of P

char P[]= "Nath";

char T[]="Dr. Paras Nath Singh";

We will have to find starting index of P in T

```
Dr. Paras Nath Singh
N                              //N compared with D  - Not matched
 N                            // N compared with r -  Not matched
  N                           // N compared with .  - Not matched
   N                          // Not Matched with blank  -  Not matched
    N                         // Not Matched with P -  not matched
     N                        // N compared with a - Not Matched
      N                       // N compared with r – not matched
       N                              // N compared with a – Not matched
        N                     // N compared with s – Not matched
         N                    // N compared with blank – Not matched
          Nath                // N matches, a matches, t matches h matches
```

Thus total 4 characters matched i.e. equivalent to length of P & it is 11th iteration of I outer loop.

So, index of P in T is 11

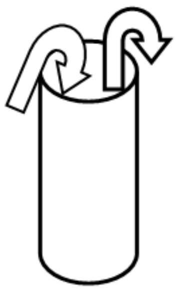
Module – 2

Q 3 a. Define stack. Write the procedure for two basic operations associated with stack (5 marks)

Ans:

A stack is a LIFO(Last in first out) data structure having one end open only to insert or delete items. The item inserted at last will be deleted at first.

A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.



A stack is generally implemented with only two basic principle operations

- Push : adds an item to a stack on “top”.
- Pop: extracts the most recently pushed item from the stack from “top”.

Procedure/Functions:

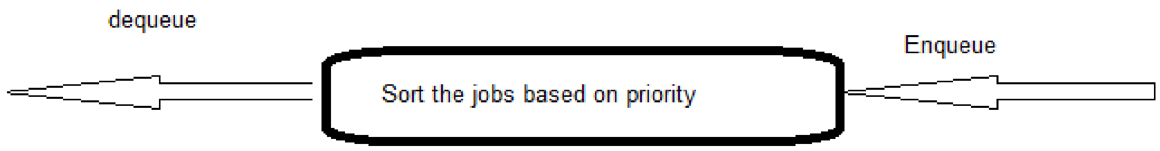
```
/*MAX is the size of stack & item is to be pushed incrementing
the top variable */

void push(int stack[], int item)
{
  if (top == (MAX - 1))
         printf("Stack is full/Overflow");
  else
  {
      top=top+1;
      stack[top]=item;
  }
 }

 /* function to pop the elements off the stack */
 int pop(int stack[])
 {
  int ret;
   if(top == -1)
   {
        printf("The stack is empty/underflow"  );
        return (NULL);
   }
   else
   {
       ret = stack[top];
       top = top-1;
       return (ret);
    }
  }
```

Q.3.b. Write a short note on priority queue. (5 marks)

Ans:

A Priority Queue is a queue but different from a normal queue, because instead of being a “first-in-first-out”, values come out in order by priority. It is an abstract data type that captures the idea of a container whose elements have “priorities” attached to them.

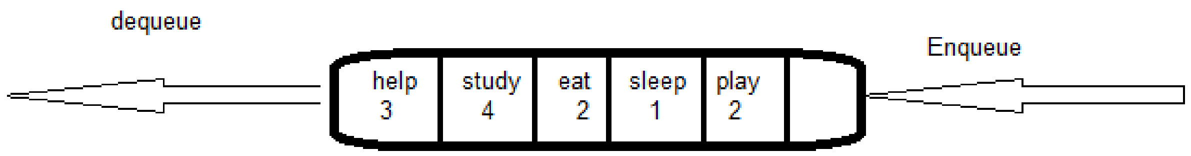
Jobs are rescheduled in the queue based on their priority number. In general a job least priority number will have highest priority. A job with higher priority is processed/de-queued before a job with lower priority.
Jobs/Items with the similar priority will be scheduled on FCFS (First Come First Serve) basis.



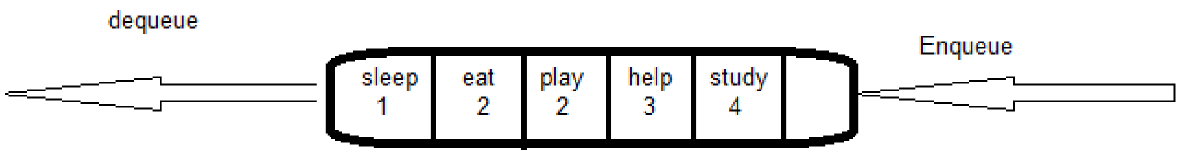
Priority Queue

Abstract of Priority queue:

```
struct pq
{
        char jname[20];
        int priority
}jobs[5];

jobs[0].jname="help";
jobs[0].priority=3;
jobs[1].jname="study";
jobs[1].priority=3;
jobs[2].jname="eat";
jobs[2].priority=2;
jobs[3].jname="sleep";
jobs[3].priority=1;
jobs[4].jname="play";
jobs[4].priority=2;
```



Before processing job/items are rescheduled based on priority:



Q 3 b. Define recursion. What are the properties of recursive procedure? Write recursive procedure for i) Tower of Hanoi ii) Factorial of a number (10 marks)

Ans:

Recursion is a process where a function calls itself repeatedly.

Recursion is repeated application of a procedure or definition.

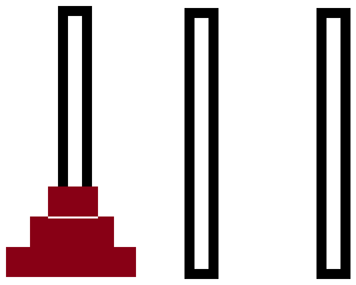
Properties of a recursive procedure:

There are 2 properties of a recursive procedure:

a. Base criteria − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
b. Progressive approach − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

i) Recursive procedure for Tower of Hanoi:

Problem Definition:



This is world-wide famous problem where complexity grows exponentially – O(2^n)
There are 3 pegs/towers say "left"(source) , "middle"(auxiliary) & "right"(destination). Number of disks (here 3) are in left peg in descending order of their size. Problem is to move all diskettes from left to right (with help of middle) with following conditions.

a. One disk at a time
b. No larger disk on smaller disk

```
1. Procedure TOH(disks, left, right, middle) //disks is integer & left, right middle are strings
2. If disks>0 then
     I.   TOH(disks-1, left, middle, right) // recursion  to alter the name of the pegs
     II.  Print "Move disk " + disks + " from" + left + "to" + right
     III. TOH(disks-1, middle, right, left) // recursion again
3. Endif
4. End of procedure TOH
```

ii. Recursive Procedure for Factorial of a number:

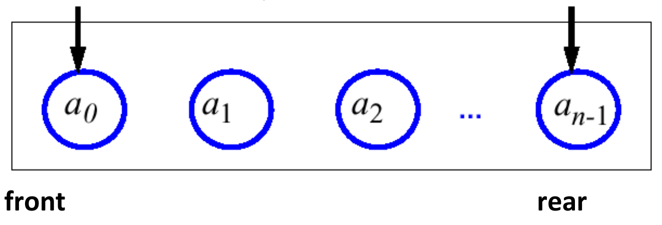
```
1. Procedure factorial(number)
2. If number = 0 OR number = 1
     a. Return 1
3. Else
     a. Return number*factorial(number-1)
4. Endif
5. End Procedure Factorial
```

Function in C:

```
long int factorial(int n)
{
    if (n==0 || n==1)     Return 1;
    else return  n*factorial(n-1);
}
```

Q 4 a. Define Queues. Write QINSERT and QDELETE procedures for queues using arrays. (10 marks)
Ans:

- ✓ A queue is a FIFO data structure
- ✓ A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- ✓ Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- ✓ Elements are inserted at the rear (enqueued) and removed from the front (dequeued)



QINSERT and QDELETE procdures/functions

```
void QINSERT(int q[], int item)
{
  if(rear==MAX-1)
            printf("Queue is full\n");
  else
            {
              rear++;
              q[rear]=item;
              if(front==-1) front=0; /*for first time*/
            }
 }

void QDELETE(int q[])
 {
            if(front==-1)
             printf("Queue is empty\n");
            else
             {
                        q[front]=0;
                        front++;
             }
             if(front==rear+1)  front=rear=-1;  /*when all items deleted */
 }
```

Q 4 b. Write the postfix form of the following expression (5 marks)
A+ (B*c – D/E ↑ F) * G) *H

Ans:
A+ (B*C – D/E ↑ F) * G) *H
↑ symbol is used for exponentiation. One right parenthesis after F) is assumed extra

- **A +(BC* - D/ EF↑ * G) *H**
- **A +(BC* - DEF↑/ * G)*H**
- **A+(BC* - DEF↑/G*) *H**
- **A + BC*DEF↑/G*- * H**
- **A + BC*DEF↑/G*-H***

A BC*DEF↑/G*-H*+
Converted to postfix expression

Q 4 c. Write a note on Ackermann's function (5 marks)

Ans:
Ackerman's function is a classic example of recursive function. **It is well defined total computable function but not primitive recursive.** Earlier before 1995 the concept was that all computable functions are primitive recursive also.
Ackerman's function grows exponentially:

Ackerman's A(x,y) function is defined for integer x and y:

$$a(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ a(x-1,1) & \text{if } y = 0 \\ a(x-1,\ a(x,y-1)) & \text{otherwise} \end{cases}$$

We can implement in C function:

```
int ackermann(int x, int y)
{
      if(x==0)  return y+1;
      if(y==0) return ackermann(x-1,1);
      return ackermann(x-1,ackermann(x,y-1));
}
```

If x=1 and y = 2, the result will be 4 and total function calls in recursion will be 10 times.

Module 3

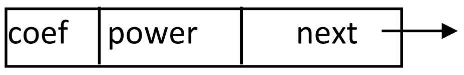
Q 5 a. Write the following algorithm for singly linked list.

- **(i) Inserting ITEM as the first node in the list**
- **(ii) Delting the node with the givem ITEM of information..**

Q 5 b. Write the node structure for linked representation of polynomial. Write the function to add two polynomials represented using linked list.

Ans:

```
/* node structure for linked representation of a polynomial */
typedef struct ntype
{
        int coef;
       int power;
       struct ntype *next;
 }node *p1, *p2, *p3;
```



Linked representation of a polynomial node

Function to add two polynomials:

```
void addpoly(node *p1, node *p2, node **p3)
 {
  int po;
  float co;
  while ( ( p1 != (node *) NULL &&  p2 != (node *) NULL) )
           {
            if(p1->power > p2->power)
             {
                 co = p1->coeff;
                 po = p1->power;
                 p1 = p1->next;
             }
           else if ( p1 -> power < p2 -> power)
           { co = p2->coeff;
             po = p2->power;
             p2 = p2 -> next;
           }
           else
           { co = p1 -> coeff + p2 -> coeff;
            po = p1 -> power;
            p1 = p1 -> next;
            p2 = p2 -> next;
           }
          if (co != 0)
           addnode(p3, co, po);
       }
   if (p1 == (node *) NULL)
   {
        for (; p2 != (node *) NULL; p2 = p2 -> next)
        addnode(p3, p2->coeff,p2->power);
   }
  else if (p2 == (node *) NULL)
       {
           for (; p1 != (node *) NULL; p1 = p1 -> next)
           addnode(p3, p1->coeff,p1->power);
       }
 }
```

OR

Q 6a. Write the functions perform the following: (10 Marks)

i. Inverting a singly linked list

ii. Concatenating the singly linked list

iii. Finding the length of a circular list.

Ans:

i. Function for Inverting a singly linked list:

```
void rev(node *list)
 {
    node *prevnode, *tempnode, *nextnode;
    /* Being careful for pointers.
       First store address of next node to
     other pointer variable
 then alter the path. - Dr. P. N. Singh  */
    nextnode=list->next;
    prevnode=list;
    prevnode->next=NULL;

    while(nextnode->next!=NULL)
   {
         tempnode=nextnode;
         nextnode=nextnode->next;
         /* again first storing the address
         And then altering the path*/
         tempnode->next=prevnode;
```

```
void addnode(node **ptr, int c, int p)
{
 node *newnode;
 newnode = (node *) malloc(sizeof(node));
 newnode->coeff=c;
 newnode->power=p;
 if(*ptr == (node *) NULL)
 {
   newnode->next = (node *) NULL;
    *ptr = newnode;
 }
 else
 { newnode->next = *ptr;  *ptr = newnode;
  }
}
```

```
            prevnode=tempnode;
        }
    }
```

ii. Concatenating the singly linked list

/*list1 & list2 are two singly linked list and ther pointer name is next */

```
void concatenate(struct node *list1, struct node *list2)
{
   If(a->next == NULL)
        list1->next =list2;
       else
    concatenate (list1->next, list2);
}
```

iii. Finding the length of a circular list.

Ans:

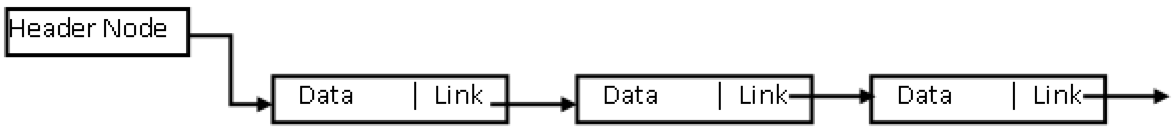
```
int count_nodes(struct node *cq)
{
   int k = 1;
   struct node *temp = cq;
   while (temp->next != cq)
   /*till pointer does not point to first node/cq */
   {
      k++;
      temp = temp->next;
   }
   return (k); /* returning total number of nodes */
}
```

Q 6 b. Write a note on header linked list (5 marks)

Ans:

A header linked list is linked list that always contain a special note at the front of the list, this special node is called headed node.

A Pointer first keeps the address of header node.



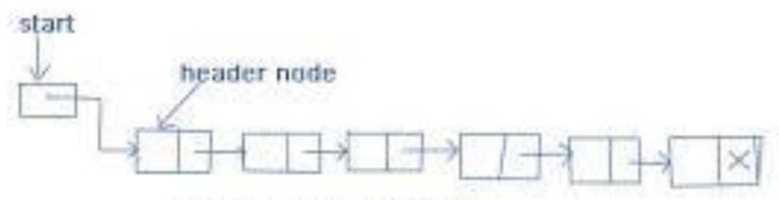
The header node does not contain actual data item included in the list but usually contains some useful information about the entire linked list.

Such as

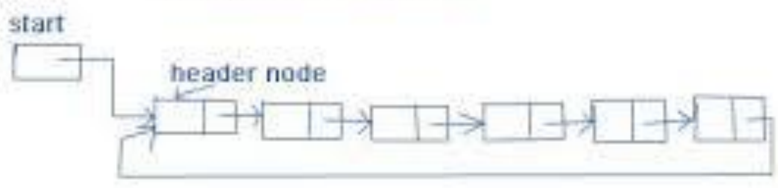
- Total number of nodes in the list .
- pointer to the last node in the list or to the last node accessed the header node in the list is never deleted that always point to the first node in the list.

General types of header linked lists are:

- Grounded header list: Where the last node contains the NULL pointer.
- Circular Header list: Where the last node contains the address of header node. In case of empty list the pointer of header node points to itself.



(a) Grounded header list



(b) Circular header list.

Q 6 c. For the given sparse matrix, write the diagrammatic linked list representation
(5 marks)

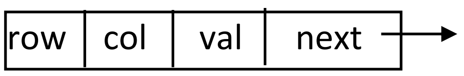
Ans:

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{pmatrix}$$

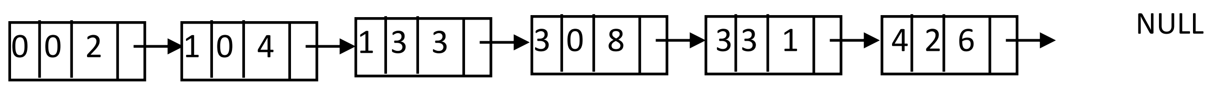
Ans:
A sparse matrix has relative number of elements 0. Representing a sparse matrix using 2-D array takes substantial amount of space with no use.
A sparse matrix can be represented by triplets with maximum column of 3 and each row will have corresponding row number, column number of non-zero elements and non-zero element itself.

Those triples can also be represented by a singly linked list having members row, col, val (non-zero) and address of next node:
Structure definition:

```
struct spmlist
{
        int row,col,val;
        struct spmlist *next;
};
```



Triplets in given sparse matrix: 0 0 2,   1 0 4,   1 3 3,   3 0 8,   3 3 1,   4 2 6
Now linked representation



Q 7 a. What is a tree? Write the routines to traverse the given string using

i) Pre-order traversal

ii) In-order traversal

iii) Post-order traversal **(10 marks)**

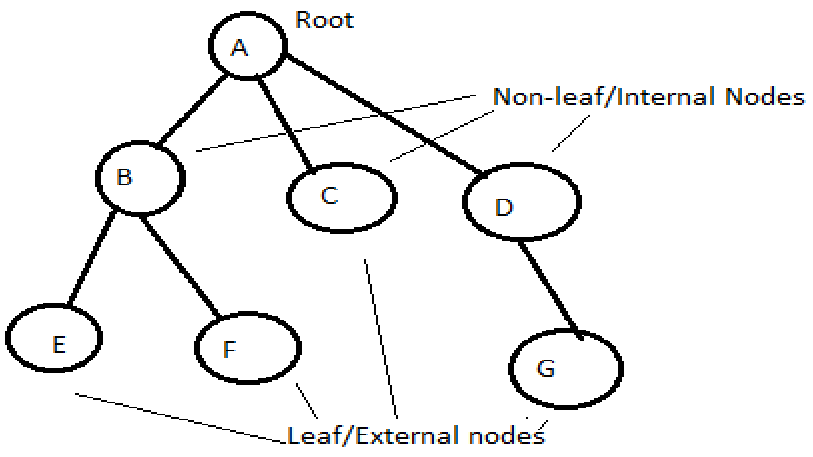
Ans:

A tree is a non-linear data structure that simulates a hierarchical tree structure stating with a root node and connected with subtree or children nodes.

Contrary to a physical tree, the root is usually depicted at the top of the tree structure and the leaves (leaf nodes) are depicted at the bottom.

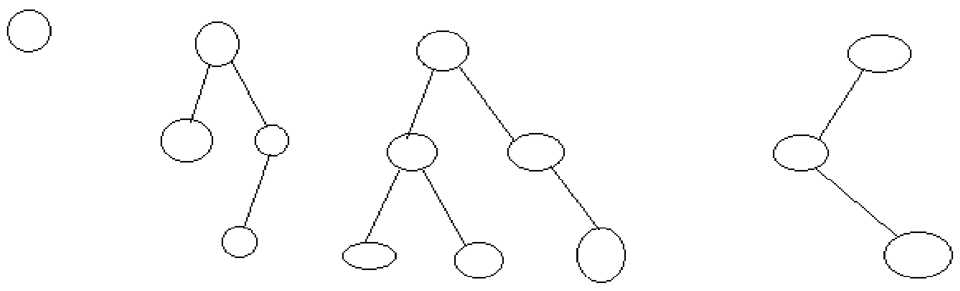
Leaf/External nodes will not have branch/child. Non-leaf/Internal nodes will have branch/child. Internal nodes are also known as parent node to their children node.

Children of same parent are known as siblings. In given diagram B,C & D are siblings. E and F are also siblings.



(i) Binary Tree:

A Binary tree is a tree where a node may have 0, 1 or 2 (maximum) children (called left and right child). All sub-trees of Binary tree are also Binary tree.



Binary Trees

Ans:

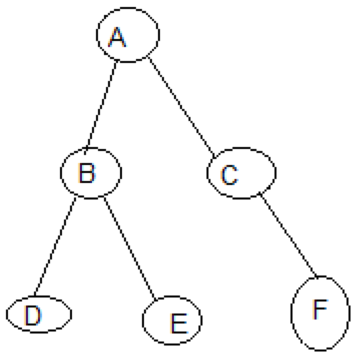
Pre-Order Traversal of a Binary Tree:

- Visit the Root Node
- Visit Left Sub-tree
- Visit Right Subtree

Routine/Recursive Function for Pre-order Traversal (Assuming that Binary tree structure has 3 members str, left and right pointers to the structure):

```
struct  tree
{
    char c;
    struct tree *left;
    struct tree *right;
};
type struct tree TREE;
TREE *bintree;
```

Assuming that given string characters are in the Binary tree

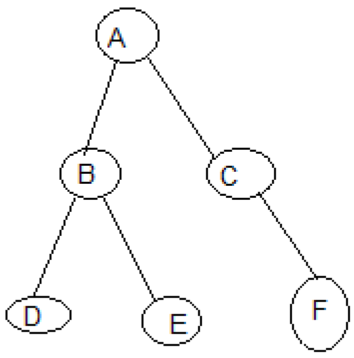


Pre-Order Traversal routine of a Binary Tree:

- Visit Left Sub-tree
- Visit the Root Node
- Visit Right Subtree

Pre-order traversal of given Binary tree
ABDECF

```
/*Recursive function of Pre-order Traversal*/
void preorder(TREE *bintree)
  {
   if(tree)
    {
              printf(" %s",bintree->str);
              preorder(bintree->left);
              preorder(bintree->right);
    }
  }
```

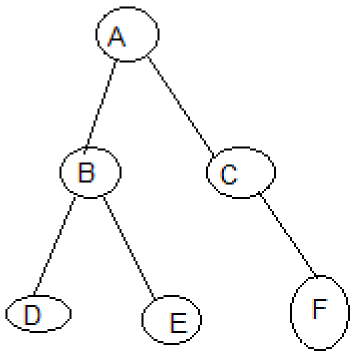


In-Order Traversal routine of a Binary Tree:

- Visit the Root Node
- Visit Left Sub-tree
- Visit Right Subtree

In-order traversal of given Binary tree
DBEACF

```
/* Recursive function of in-order traversal */
void inorder(TREE *bintree)
  {
   if(tree)
    {
               inorder(bintree->left);
               printf(" %s",bintree->str);
               inorder(bintree->right);
    }
  }
```



Post-Order Traversal routine of a Binary Tree:

- Visit Left Sub-tree
- Visit Right Subtree
- Visit the root node

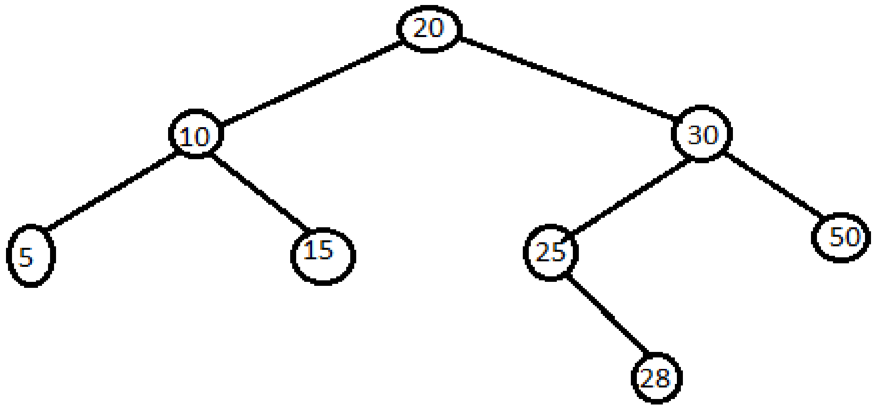
Post-order traversal of given Binary tree
DEBAFC

```
/* Recursive function of in-order traversal */

void postorder(TREE *bintree)
  {
   if(tree)
   {
            postorder(bintree->left);
            postorder(bintree->right);
            printf(" %s",bintree->str);
   }
  }
```

Q7 b. What is a Binary Search Tree? Write algorithm to implement for recursive search or iterative search for a Binary Search Tree. (10 marks)

Ans:

Binary Search Tree: A Binary Search tree is a Binary Tree in which left nodes contain less value than parent and right nodes contain greater value than parents. All sub trees of Binary Search Tree are also Binary Search Tree.
In general a BST does not contain duplicate values.



A Binary Search Tree

Algorithm for recursive search in a Binary Search Tree:

Input : A pointer to **BST & skey**(Binary Search Tree with data and left and right pointer & search key value)

Output: found / not-found flag counting number of hits/comparisons/for locations

```
1. hits=0
2. found = false
3. function searchbst(BST *p,)
   3.1. if p=NULL then
```

```
        3.1.1.   return found // false
    3.2.  endif
    3.3.   hits=hits+1
    3.4.   If skey = p->data then
        3.4.1.   found = true
        3.4.2.   return found  // true
    3.5.   else if  skey  < p->data then
        3.5.1.   searchbst(p->left) // recursion to search left sub-tree
        3.5.2.   else
        3.5.3.   searchbst(p->right) // recursion to search right sub-tree
    3.6.   Endif
4.  End function searchbst
Display the found status and count of hits
```

Iterative search for binary search tree:

```
element* iterSearch(treePointer tree, int k)
{
   /* return a pointer to the element whose key is k, if there is no such element, return NULL. */
  while (tree)
   {
       if (k == tree->data.key)
            return &(tree->data);
       if (k < tree->data.key)
             tree = tree->leftChild;
       else
             tree = tree->rightChild;
   }
   return NULL;
```

Note: If h is the height of the binary search tree, iterSearch: O(h)However, search has an additional stack space requirement which is O(h)

Q 8 a. Write the routines for, (i) Copying Binary Tree, (ii) testing for equality of binary trees.
(10 Marks)

Ans:
First defining structure:

```
typedef struct TREE
{
             int data;
            struct TREE *left;
            struct TREE *right;
 }TREE;
```

(i) **Routine (in C) for Copying Binary Tree**

```
void t_cpy(NODE *t1,NODE *t2)
{
    int val,opt=0;
    NODE *temp;
    if(t1==NULL || t2==NULL)
    {
       printf("Can not copy !\n");
    }
    inorder(t1);

    printf("\nEnter the node value where tree 2 should be copied\n");
    scanf("%d",&val);
    temp=t1;
    while(temp!=NULL)
    {
       if(val<temp->data)
          temp=temp->llink;
       else
          temp=temp->rlink;
    }
    if(temp->llink!=NULL || temp->rlink!=NULL)
       printf("Not possible to copy tree to this node\n");
    else
    {
       printf("Copy tree to \n 1.Left Node \n 2.Right Node\n Enter your choice : ");
       scanf("%d",&opt);
       if(opt==1)
       {
          temp->llink=t2;
       }
       else if(opt==2)
       {
          temp->rlink=t2;
       }
       else
          printf("Invalid choice\n");
    }
    printf("Tree1 after copying is\n");
    inorder(temp);
}
```

(ii) Routine for testing equality of Binary Trees

/* Two binary trees are equal if their topologies are same and corresponding nodes contain identical values */

```
int compare( TREE *p1, TREE *p2)
 {
        While(p1 || p2)             /* while both are not NULL */
        {
                If((p1 && !p2) || (p2 && !p1) /* if one of the tree points to NULL & other doesn't */
                        return (0);         /* returns false */
                If( p1->data != p2->data)
                        return (0);        /* returns  false */
                p1=p1->next;
                p2=p2->next;
        }
        return (1);        /* returns true if not returned by while loop */
 }
```

Q 8 b. List the rules to construct threads. Write the routines for inorder traversal of a threaded binary tree. (10 marks)

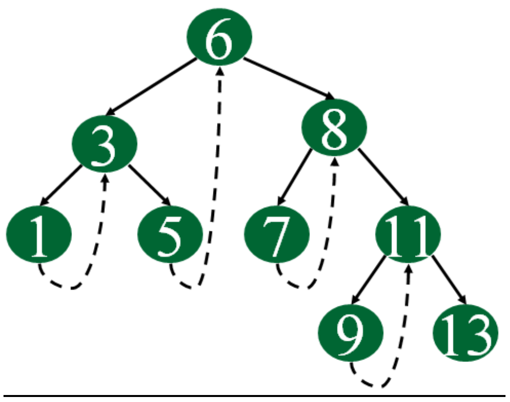
Ans:

Level-order scheme visit the root first, then the root's left child, followed by the root's right child. All the node at a level are visited before moving down to another level.

Two methods/rules:

i. Use of parent field to each node.
ii. Use of two bits per node to represents binary trees as threaded binary trees.

A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).



<u>In-order threaded Binary Tree</u>

<u>Routines/Functions for inorder traversal of a threaded binary tree</u>

```
/* Utility function to find leftmost node in a tree rooted with n */
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
       return NULL;

    while (n->left != NULL)
        n = n->left;

    return n;
}

 /* C code to do inorder traversal in a threaded binary tree */

void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);
        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
}
```

Module 5

Q 9 a. Write an Algorithm for an insertion sort. Also discuss about complexity of insertion sort.

Ans:

Method:

It is picking the element to insert, shifting the grater element and inserting the picked element at correct position. Actually these operation starts from second elements from the beginning.

Insertion sort always maintains two zones in the array to be sorted: *sorted* and *unsorted*. At the beginning the sorted zone consist of one element. On each step the algorithms expand it by one element inserting the first element from the unsorted zone in the proper place in the sorted zone and shifting all larger elements one slot down. It is an algorithms that many people intuitively use for sorting cards and it is very easy to illustrate on the deck of cards. Here is an example (sorted zone is in blue, unsorted is in red):

5 | 3 1 7 0 -> 3 5 | 1 7 9 -> 1 3 5 | 7 9 -> 1 3 5 7 | 9 -> 1 3 5 7 9

Algorithm:

Input : Array a of size: size

```
1. for x = 1 to size   // from second element for zero based arrays
    a. picked=a[x];
    b. y=x;
    c. while(y>0 and picked < a[y-1])
       do
        i. a[y]=a[y-1];
        ii. y=y-1;
    d. endwhile
    e. a[y]=picked;
2. next x
```

Output: sorted array a:

Discussion on complexity:

If the inversion count is O(n), then the time complexity of insertion sort is O(n). ...

The worst case occurs when the array is sorted in reverse order.

So the worst case time complexity of insertion sort is O(n^2).

Q 9b. Write an algorithm for : i) Breadth First Search II) Depth First Search. (10 marks)

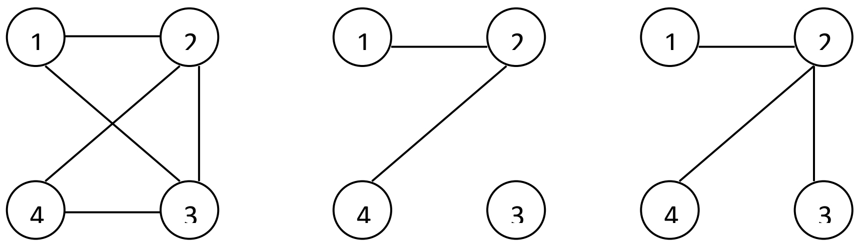
Ans:

OR

Q 10 a. Define graph. Explain in detail about directed graphs. (10 marks)

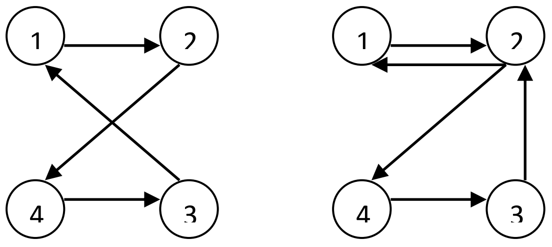
A graph is a set of vertices and edges which connect them. A graph is also known as finite set of ordered pairs.
We write: **G = (V,E)** where **V** is the set of vertices and the set of edges,
E = { (v_i,v_j) } where **v_i** and **v_j** are in **V**.
These pairs are called the edges of G. If e = (v,w) is an edge with vertices v and w, then v and w are said to lie on e, and e is said to be incident with v and w.
Paths : A *path*, p, of length, k, through a graph is a sequence of connected vertices:
p = <v_0,v_1,...,v_k> where, for all **i** in (0,**k**-1: **(v_i,v_{i+1})** is in **E**.

Undirected graph or graph – Pairs are unordered. They may be connected or disconnected.



Connected & Cycle **Disconnected** **Tree**
Various kinds of Undirected Graph

Directed graph or digraph – Pairs are ordered. A directed graph is called strongly connected if



There is a directed path from any vertex to any other vertex.

Applications of graphs are in circuit network, transport network, program flow diagram etc.

A directed graph may be

Symmetric directed graphs are directed graphs where all edges are bidirected (that is, for every arrow that belongs to the digraph, the corresponding inversed arrow also belongs to it).

Simple directed graphs are directed graphs that have no loops (arrows that connect vertices to themselves) and no multiple arrows with same source and target nodes. As already introduced, in case of multiple arrows the entity is usually addressed as directed multigraph. Some authors describe digraphs with loops as loop-digraphs

Q10 b. Explain in detail about static and dynamic hashing (10 Marks)
Ans:
Hashing is an efficient technique to directly search the location of desired data on the disk without using index structure. Data is stored at the data blocks whose address is generated by using hash function. The memory location where these records are stored is called as data block or data bucket.
Static hashing:

- Single hash function h(k) on key k
- Desirable properties of a hash function
    - Uniform: Total domain of keys is distributed uniformly over the range
    - Random: Hash values should be distributed uniformly irrespective of distribution of keys O(1) search
- Example of hash functions: h(k) = k MOD m (size of hash table)
- Collision resolution
- Chaining
    - Load factor
    - Primary pages and overflow pages (or buckets)
    - Search time more for overflow buckets
- Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Problems of static hashing

- Fixed size of hash table due to fixed hash function
- **Primary/secondary Clustering (We should keep size of hash table a prime number to reduce clustering)**
- May require rehashing of all keys when chains or overflow buckets are full

Dynamic hashing:

In this hashing scheme the set of keys can be varied & the address space is allocated dynamically. If a file F is collection of record a record R is key+data stored in pages (buckets) then space utilization:

Number of records/(Number of pages*Page capacity)

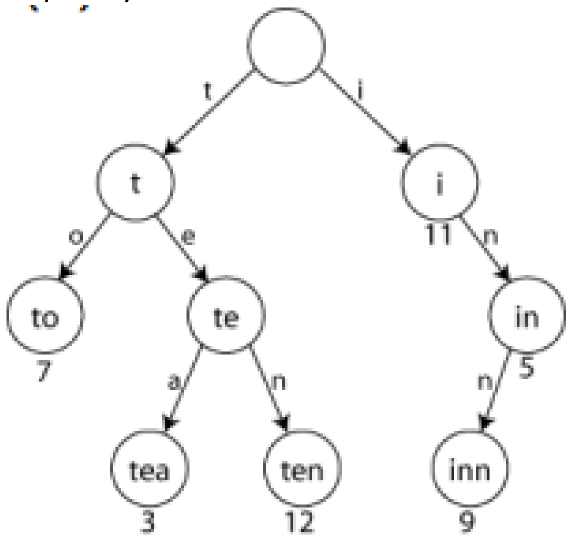
- Hash function modified dynamically as number of records grow
- Needs to maintain determinism
- Extendible hashing
- Linear hashing

Again: Dynamic hashing Hash function modified dynamically as number of records grow Needs to maintain determinism Extendible hashing and Linear hashing

- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- $h_0(k)$ produces index of primary page
- Particular access structure for binary trees
- Family of functions : $g(k) = \{ h_1(k),....h_i(k),....\}$

- Each $h_i(k)$ produces a bit
- At level i, $h_i(k)$ is 0, take left branch, otherwise right branch Example: bit representation

Trie: A Trie (retrieved from re**trie**ve) is a Binary tree in which an identifier is located by its bits sequence. Here key lookup is faster. Looking up a key of length m takes maximum O(m) time complexity in worst case.


t
i
t
i
11
o
e
n
to
7
te
in
5
a
n
n
tea
3
ten
12
inn
9