DATA STRUCTURES

(DEC 2018)

Q 1

a) **What are various operations possible on data structures?** **(05)**

➔ The data appearing in our data structure is processed by means of certain operations. In fact, the particular data structure that once chooses for a given situation depends largely on the frequency with which specific operations are performed:

1. <u>Insertion</u>: Insertion means addition of a new data element in a data structure.
2. <u>Deletion</u>: Deletion means removal of a data element from a data structure if it is found.
3. <u>Searching</u>: Searching involves searching for the specified data element in a data structure.
4. <u>Traversal</u>: Traversal of a data structure means processing all the data elements present in data structure.
5. <u>Sorting</u>: Arranging data elements of a data structure in a specified order is called sorting.
6. <u>Merging</u>: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

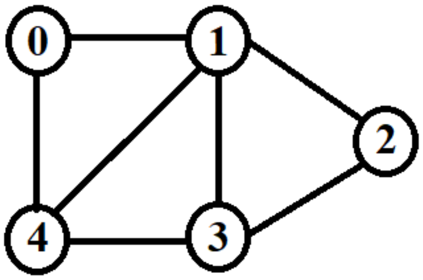
b) **What are different ways of representing a Graph data structure on a computer?** **(05)**

➔ 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.

Following is an example of undirected graph with 5 vertices.

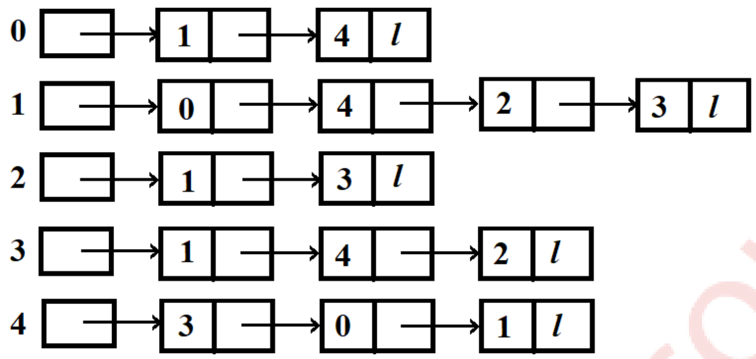


There are two most commonly used representations of a graph.

1. 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.
   - Following is adjacency matrix representation of the above graph.

	0	**1**	**2**	**3**	**4**
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

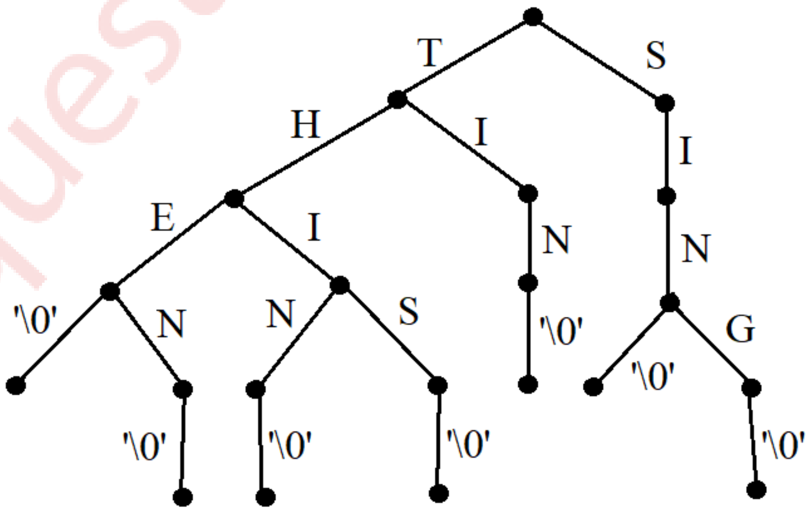
2. Adjacency List
   - An array of lists is used. Size of the array is equal to the number of vertices.
   - Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex.
   - This representation can also be used to represent a weighted graph.
   - The weights of edges can be represented as lists of pairs.
   - Following is adjacency list representation of the above graph.



c) Describe Tries with an example. **(05)**

➔ - A trie is a tree-like data structure whose nodes store the letters of an alphabet. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree.

- A trie is a tree of degree P ≥ 2.
- Tries re useful for sorting words as a string of characters.
- In a trie, each path from the root to a leaf corresponds to one word.
- Root node is always null.
- To avoid confusion between words like THE and THEN, a special end marker symbol '\0' is added at the end of each word.
- Below fig shows the trie of the following words (THE, THEN, TIN, SIN, THIN, SING)



- Most nodes of a trie has at most 27 children one for each letter and for '\0'
- Most nodes will have fewer than 27 children.
- A leaf node reached by an edge labelled '\0' cannot have any children and it need not be there.

d) Write a function in C to implement binary search. (05)

➔ Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

C function to implement binary search:

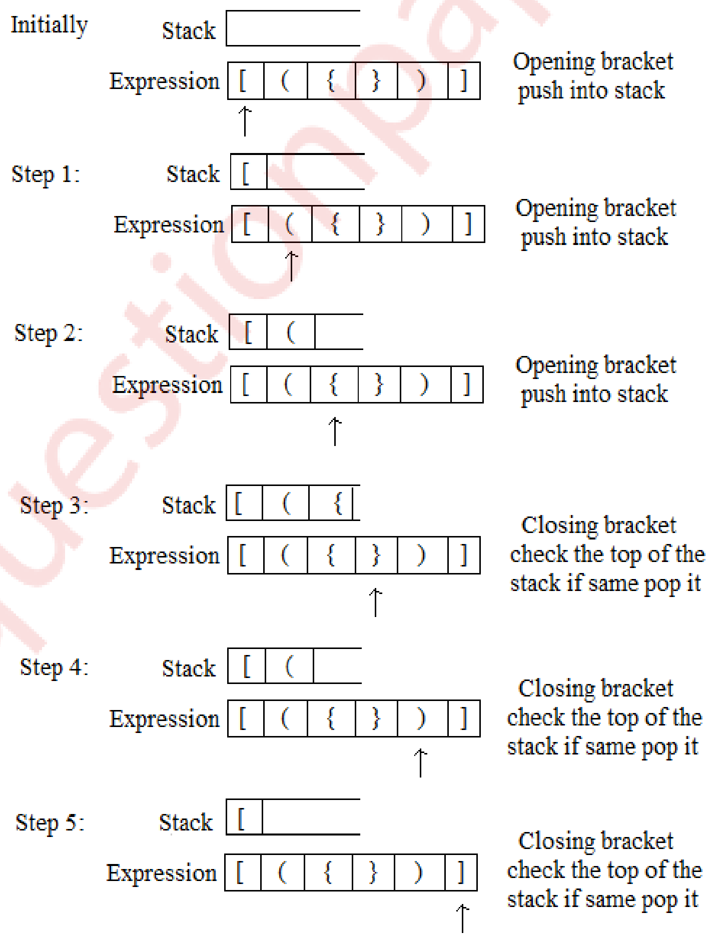
```
int binary_search(int sorted_list[], int low, int high, int element)
{
   int middle;
   while (low <= high)
   {
      middle = low + (high - low)/2;
      if (element > sorted_list[middle])
         low = middle + 1;
      else if (element < sorted_list[middle])
         high = middle - 1;
      else
         return middle;
   }
   return -1;
}
```

Q 2

a) **Use stack data structure to check well-formedness of parentheses in an algebraic expression. Write C program for the same. (10)**

➔ An expression is said to be well formed with respect to parenthesis:

1. If every opening parenthesis has a closing parenthesis.
2. If we count number of parenthesis of left parenthesis and right parenthesis then at no time, count of right parenthesis should exceed the count of left parenthesis.
3. A stack is used to validate an expression using simple rules by scanning the expression from left to right.
   - If opening bracket is found then push it on the stack.
   - If closing bracket is found then check the top of the stack if it is same then pop
   - If stack is empty then string is valid or else invalid.


Initially
Stack
Expression [({ })]
Opening bracket push into stack
Step 1:
Stack [
Expression [({ })]
Opening bracket push into stack
Step 2:
Stack [(
Expression [({ })]
Opening bracket push into stack
Step 3:
Stack [({
Expression [({ })]
Closing bracket check the top of the stack if same pop it
Step 4:
Stack [(
Expression [({ })]
Closing bracket check the top of the stack if same pop it
Step 5:
Stack [
Expression [({ })]
Closing bracket check the top of the stack if same pop it

C program to check well-formedness of parentheses

```
#include<stdio.h>
#include<string.h>
#define MAX 20
#define true 1
#define false 0

int top = -1;
int stack[MAX];

/*Begin of push*/
char push(char item)
{
        if(top == (MAX-1))
                printf("Stack Overflow\n");
        else
        {
                top=top+1;
                stack[top] = item;
        }
}

/*Begin of pop*/
char pop()
{
        if(top == -1)
                printf("Stack Underflow\n");
        else
                return(stack[top--]);
}

main()
{
        char exp[MAX],temp;
        int i,valid=true;
        printf("Enter an algebraic expression : ");
        gets(exp);

        for(i=0;i<strlen(exp);i++)
        {
                if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
                        push( exp[i] );
                if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
                        if(top == -1)    /* stack empty */
                                valid=false;
                        else
                        {
                                temp=pop();
                                if( exp[i]==')' && (temp=='{' || temp=='[') )
                                        valid=false;
                                if( exp[i]=='}' && (temp=='(' || temp=='[') )
                                        valid=false;
                                if( exp[i]==']' && (temp=='(' || temp=='{') )
                                        valid=false;
                        }
        }
        if(top>=0) /*stack not empty*/
                valid=false;

        if( valid==true )
                printf("Valid expression\n");
        else
                printf("Invalid expression\n");
}
```

Output:

```
Enter an algebraic expression: {([])}
Valid expression
Enter an algebraic expression: (){}
Valid expression
Enter an algebraic expression: (){[[[)
Invalid expression
```

b) Give the frequency for the following symbols, compute the Huffman code for each symbol. (10)

Symbol	**A**	**B**	**C**	**D**	**E**
Frequency	**24**	**12**	**10**	**8**	**8**

➔ Huffman Code:

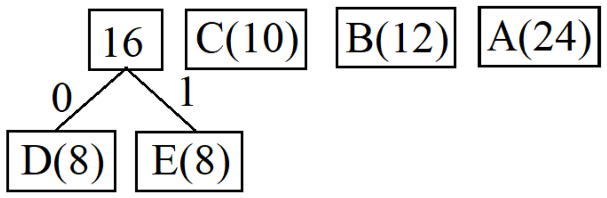
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages M1, M2, …Mn
- Message is converted into a binary string.
- Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.
- It use patterns of zeros and ones in communication system these are used at sending and receiving end.
- suppose there are n standard message M1, M2, ……Mn. Then the frequency of each message is considered, that is message with highest frequency is given priority for the encoding.
- The tree is called encoding tree and is present at the sending end.
- The decoding tree is present at the receiving end which decodes the string to get corresponding message.
- The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree.
- Example

Symbol	A	B	C	D	E
Frequency	24	12	10	8	8

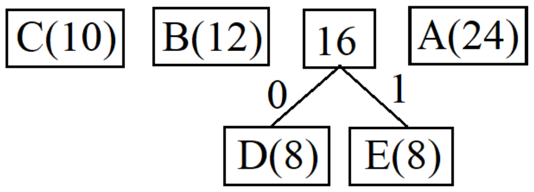
Arrange the message in ascending order according to their frequency

$\boxed{D(8)}$ $\boxed{E(8)}$ $\boxed{C(10)}$ $\boxed{B(12)}$ $\boxed{A(24)}$

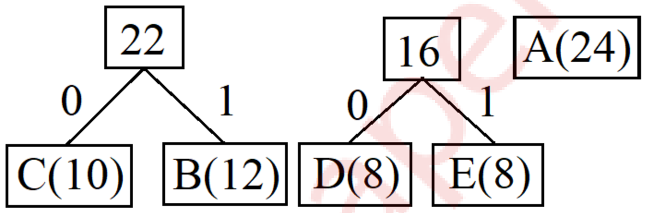
Merge two minimum frequency message



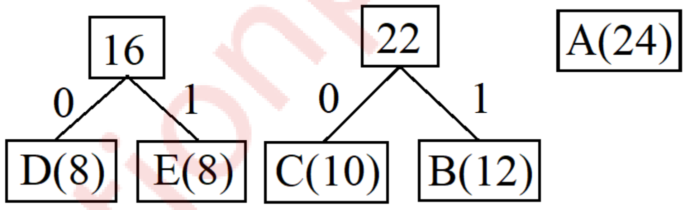
Rearrange in ascending order



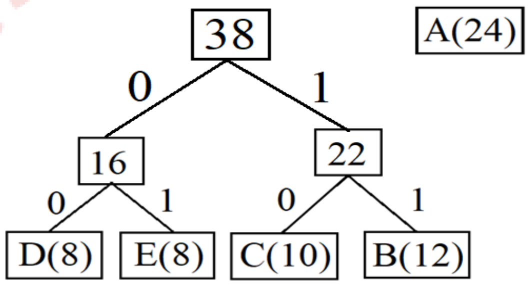
Merge two minimum frequency message



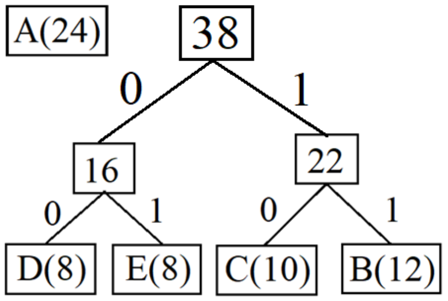
Rearrange in ascending order



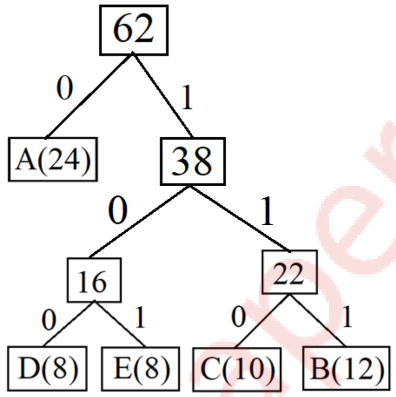
Merge two minimum frequency message



Again Rearrange in ascending order



Merge two minimum frequency message



Huffman code

A = 0

B = 111

C = 110

D = 100

E = 101

Q 3

a) **Write a C program to implement priority queue using arrays. The program should perform the following operations (12)**
i) Inserting in a priority queue
ii) Deletion from a queue
iii) Displaying contents of the queue

➔ Program:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 30

typedef struct pqueue
{
   int data[MAX];
   int rear,front;
}pqueue;

void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
int dequeue(pqueue *p);
void display(pqueue *p);

void main()
{
   int x,op,n,i;
   pqueue q;
   initialize(&q);

   do
   {
      printf("\n1)Create \n2)Insert \n3)Delete \n4)Print \n5)EXIT");
      printf("\nEnter Choice: ");
      scanf("%d",&op);
```

```
switch (op) {
  case 1: printf("\nEnter Number of Elements");
      scanf("%d",&n );
      initialize(&q);
      printf("Enter the data");

      for(i=0; i<n; i++)
      {
        scanf("%d",&x);
        if(full(&q))
        {
          printf("\nQueue is Full..");
          exit(0);
        }
        enqueue(&q,x);
      }
      break;

  case 2: printf("\nEnter the element to be inserted");
      scanf("%d\n",&x);
      if(full(&q))
      {
        printf("\nQueue is Full");
        exit(0);
      }
      enqueue(&q,x);
      break;

  case 3: if(empty(&q))
      {
        printf("\nQueue is empty..");
        exit(0);
      }

      x=dequeue(&q);
      printf("\nDeleted Element=%d",x);
      break;
```

```
            case 4: display(&q);
                    break;
            default: break;
        }
    }while (op!=5);
}

void initialize(pqueue *p)
{
    p->rear=-1;
    p->front=-1;
}

int empty(pqueue *p)
{
    if(p->rear==-1)
        return(1);

    return(0);
}

int full(pqueue *p)
{
    if((p->rear+1)%MAX==p->front)
        return(1);

    return(0);
}

void enqueue(pqueue *p, int x)
{
    int i;
    if(full(p))
        printf("\nOverflow");
    else
    {
        if(empty(p))
        {
            p->rear=p->front=0;
            p->data[0]=x;
        }
        else
        {
            i=p->rear;

            while(x>p->data[i])
            {
                p->data[(i+1)%MAX]=p->data[i];
                i=(i-1+MAX)%MAX;  //anticlockwise  movement  inside  the
queue
                if((i+1)%MAX==p->front)
                    break;
            }

            //insert x
            i=(i+1)%MAX;
            p->data[i]=x;

            //re-adjust rear
            p->rear=(p->rear+1)%MAX;
        }
    }
}

int dequeue(pqueue *p)
{
    int x;

    if(empty(p))
    {
        printf("\nUnderflow..");
    }
    else
    {
        x=p->data[p->front];
        if(p->rear==p->front)   //delete the last element
            initialize(p);
        else
            p->front=(p->front +1)%MAX;
    }

    return(x);
}

void display(pqueue *p)
{
    int i,x;

    if(empty(p))
    {
        printf("\nQueue is empty..");
    }
    else
    {
        i=p->front;
        while(i!=p->rear)
        {
            x=p->data[i];
            printf("\n%d",x);
            i=(i+1)%MAX;
        }

        //prints the last element
        x=p->data[i];
        printf("\n%d",x);
    }
}
```

Output:

```
1)Create
2)Insert
3)Delete
4)Display
5)EXIT
Enter Choice: 1

Enter Number of Elements4
Enter the data9
12
4
6

1)Create
2)Insert
3)Delete
4)Display
5)EXIT
Enter Choice: 4

12
9
6
4
1)Create
2)Insert
3)Delete
4)Display
5)EXIT
Enter Choice: 3

Deleted Element=12
1)Create
2)Insert
3)Delete
```

```
4)Display
5)EXIT
Enter Choice: 5
```

b) What are expression trees? What are its advantages? Derive the expression tree for the following algebraic expression (08)

(a + (b/c)) * ((d/e) - f)

➔ Expression Tree: Compilers need to generate assembly code in which one operation is executed at a time and the result is retained for other operations.

- Therefore, all expression has to be broken down unambiguously into separate operations and put into their proper order.
- Hence, expression tree is useful which imposes an order on the execution of operations.
- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand
- Parentheses do not appear in expresion trees, but their intent remains intact in tree representation.

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

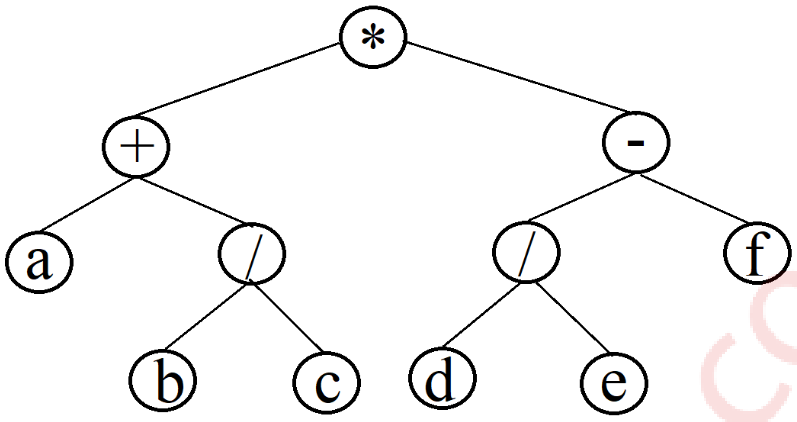
2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Advantage:

1. Expression trees are using widely in LINQ to SQL, Entity Framework extensions where the runtime needs to interpret the expression in a different way (LINQ to SQL and EF: to create SQL, MVC: to determine the selected property or field).
2. Expression trees allow you to build code dynamically at runtime instead of statically typing it in the IDE and using a compiler.

Expression Tree: (a + (b/c)) * ((d/e) - f)



Q 4

a) Write a C program to represent and add two polynomials using linked list. (12)

➔ Program:

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

typedef struct node
{
    int coef;
    int exp;
    struct node* next;
} node;

void get_input(node** head)
{
    node* temp,*ptr;
    ptr = *head;
    temp = (node*)malloc(sizeof(node));

    printf("\nEnter coef : ");
    scanf("%d",&(temp->coef));
    printf("\nEnter exp : ");
    scanf("%d",&(temp->exp));

    if(NULL == *head)
    {
        *head = temp;
        (*head)->next = NULL;
    }
    else
    {
        while(NULL != ptr->next)
        {
            ptr = ptr->next;
        }
        ptr->next = temp;
        temp->next = NULL;
    }
}

void display(node* head)
{
    while(head->next != NULL)
    {
        printf("(%d.x^%d)+",head->coef,head->exp);
        head = head->next;
    }
    printf("(%d.x^%d)",head->coef,head->exp);
    printf("\n");
}

void add(node* poly, node* poly1, node* poly2 ) //poly==result
{
    while(poly1->next && poly2->next)
    {
        if(poly1->exp > poly2->exp)
        {
            poly->coef = poly1->coef;
        poly->exp = poly1->exp;
        poly1 = poly1->next;
    }
    else if(poly1->exp < poly2->exp)
    {
        poly->coef = poly2->coef;
        poly->exp = poly2->exp;
        poly2 = poly2->next;
    }
    else
    {
        poly->coef = poly1->coef + poly2->coef;
        poly->exp = poly1->exp;
        poly1 = poly1->next;
        poly2 = poly2->next;
    }
    poly->next = (node*)malloc(sizeof(node));
    poly = poly->next;
    poly->next = NULL;
}

while(poly1->next || poly2->next)
{
    if(poly1->next)
    {
        poly->coef = poly1->coef;
        poly->exp= poly1->exp;
        poly1 = poly1->next;
    }
    if(poly2->next)
    {
        poly->coef = poly2->coef;
        poly->exp= poly2->exp;
        poly2 = poly2->next;
    }
    poly->next = (node*)malloc(sizeof(node));
    poly = poly->next;
```

```
        poly->next = NULL;
    }
}

int main()
{
    node* head1 = NULL;
    node* head2 = NULL;
    node* head = (node*)malloc(sizeof(node));
    int ch;

    do
    {
        get_input(&head1);
        printf("\nEnter more node in poly1? (1,0) :");
        scanf("%d",&ch);
    }while(ch);
    do
    {
        get_input(&head2);
        printf("\nEnter more node in poly2? (1,0) :");
        scanf("%d",&ch);
    }while(ch);

    add(head,head1,head2);
    display(head1);
    display(head2);
    display(head);

    return 0;
}
```

Output:

```
Enter coef: 6
Enter exp: 2
Enter more node in poly1? (1/0): 1
Enter coef: 4
Enter exp: 1
Enter more node in poly1? (1/0): 1
Enter coef: 2
Enter exp: 0
Enter more node in poly? (1/0): 0
Enter coef: 4
Enter exp: 2
Enter more node in poly2? (1/0): 1
Enter coef: 2
Enter exp: 1
Enter more node in poly2? (1/0): 1
Enter coef: 3
Enter exp: 0
Enter more node in poly2? (1/0): 0
(6.x^2)+(4.x^1)+(2.x^0)
(4.x^2)+(2.x^1)+(3.x^0)
(10.x^2)+(6.x^1)+(5.x^0)
```

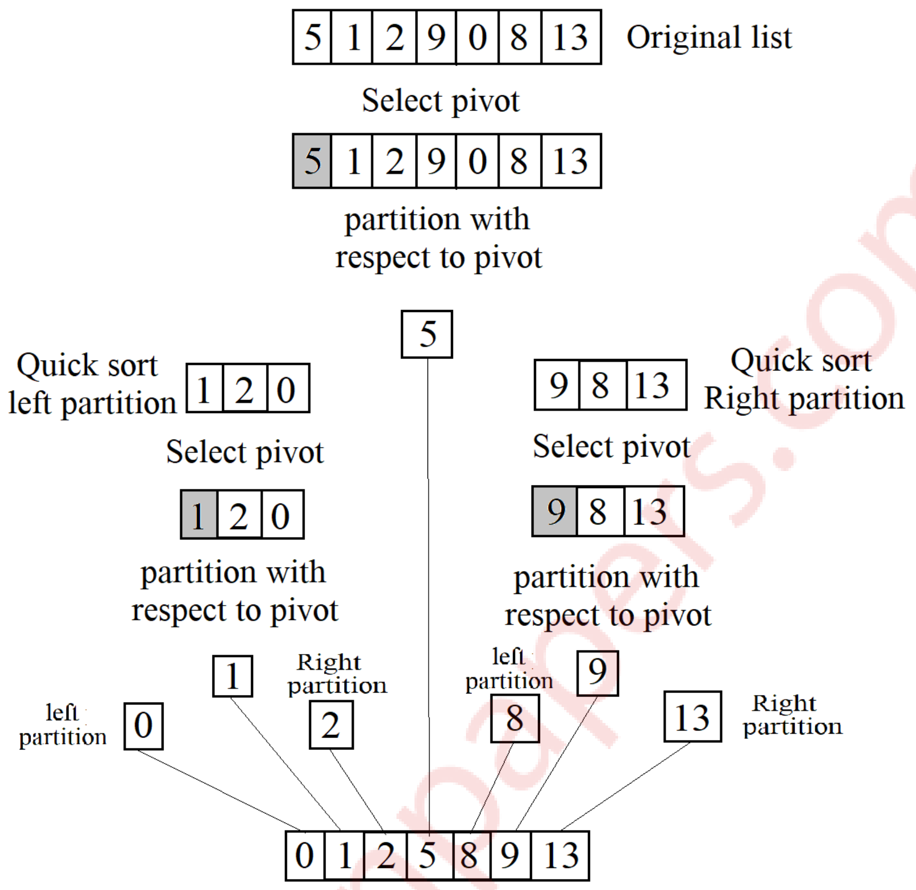
b) How does the Quicksort technique work? Give C function for the same. (08)

➔ Quick Sort:

- Quick sort is the fastest internal sorting algorithm with the time and space complexity = O(n log n).

- The basic algorithm to sort an array a[] of n elements can be describe recursively as follows:

1. If n <= 1, then return
2. Pick any element V in array a[]. This element is called as pivot.
   Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i \leq V$ left of V. if the place of the V after re-arrangement is j, all elements with value less than V, appear in a[0], a[1] … a[j-1] and all those with value greater than V appear in a[j+1] … a[n-1]
3. Apply quick sort recursively to a[0] … a[j-1] and to a a[j+1] … a[n-1]
   Entire array will thus be sorted by as selecting an element V.
   a) Partitioning the array around V.
   b) Recursively, sorting the left partition.
   c) Recursively, sorting the right partition.



<u>C function for partition</u>

```
int partition(int a[], int l, int u)
{
 int v,i,j,temp;
 v=a[l];
 i=l;
j=u+1;
do;
{
        do
             i++;
        while(a[i]<v && i<=u);
              do
                        j--;
              while(v<a[j]);
                                    if(i<j)
```

```
                  {
                          temp=a[i];
                           a[i]=a[j];
                           a[j]=temp;
                   }
         }while(i<j);
       a[l]=a[j];
       a[j]=v;
       return(j);
    }
```

<u>C function for Quick Sort</u>

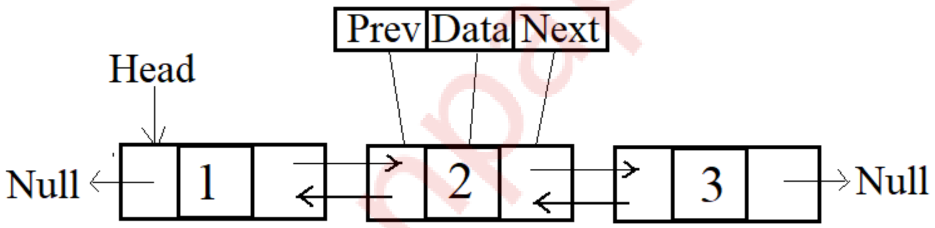
```
void quick_sort(int a[], int l, int u)
{
   int j;
   if(l<u)
   {
         j=partition(a, l, u);
         quick_sort(a, l, j-1);
         quick_sort(a, j+1, u);
   }
 }
```

Q 5

a) What is a doubly linked list? Give C representation for the same. (05)

➔ Doubly linked list:

1. Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.
2. Every nodes in the doubly linked list has three fields: LeftPointer, RightPointer and Data.
   LeftPointer = Left pointer points towards the left node.
   RightPointer = Right pointer points towards the right node.
   Data = Node which stores the data.
3. The last node has a next link with value NULL, marking the end of the list, and the first node has a previous link with the value NULL. The start of the list is marked by the head pointer.



C Representation of Doubly Linked List

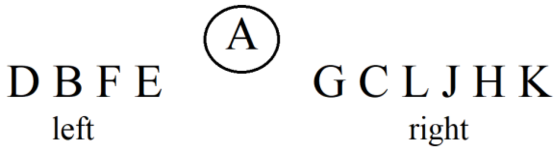
Structure of doubly link list will contain three fields LeftPointer (prev), RightPointer (next), and the data as shown below

```
struct Node  {
        int data;
        struct Node* next;
        struct Node* prev;
};
```

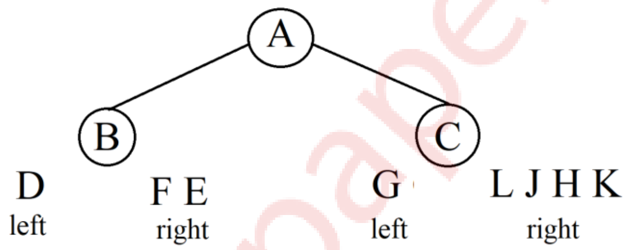
b) Given the postorder and inorder traversal of a binary tree, construct the original tree: (10)
Postorder: D E F B G L J K H C A
Inorder: D B F E A G C L J H K

➔ Construction of Tree:

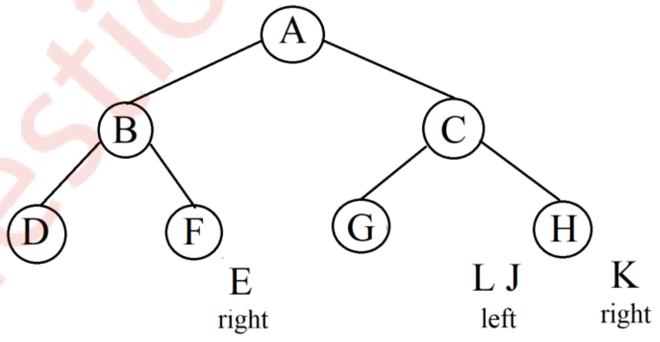
Step1: Select last element from the postorder as root node. So element A becomes root node. Divide the inorder into left and right with respect to root node A.



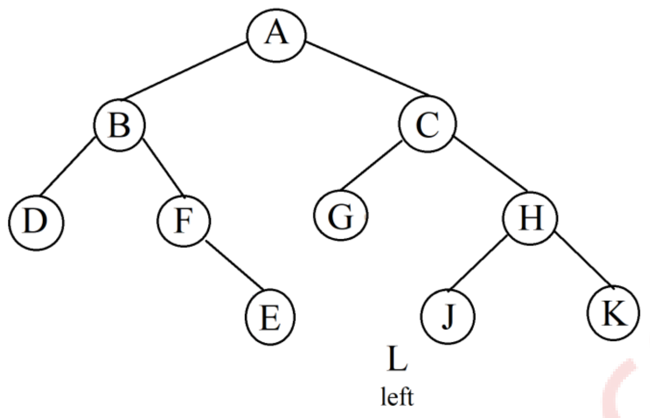
Step 2: Traverse element D B F E from postorder as B comes in last B becomes child node of A and similarly traverse G C L J H K in postorder C comes at last C becomes child node of A. Again with respect to B and C divide element into left and right as shown below



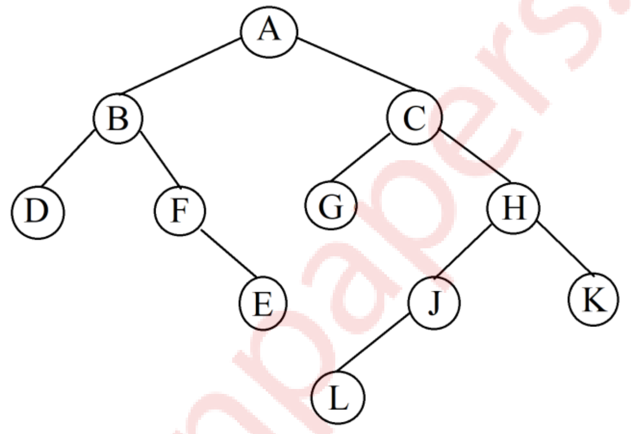
Step 3: As D is single it becomes child node of B and for left node of B Traverse F E in postorder F comes at last so F becomes child node of B. similarly, G and H become child node of C as shown below.



Step 4: As E is single it becomes child node of F. Traverse L J which is left element of node H in postorder as J comes last J becomes child node of H as K is single it becomes another child node.



Step 5: As L is simgle it becomes child node of J



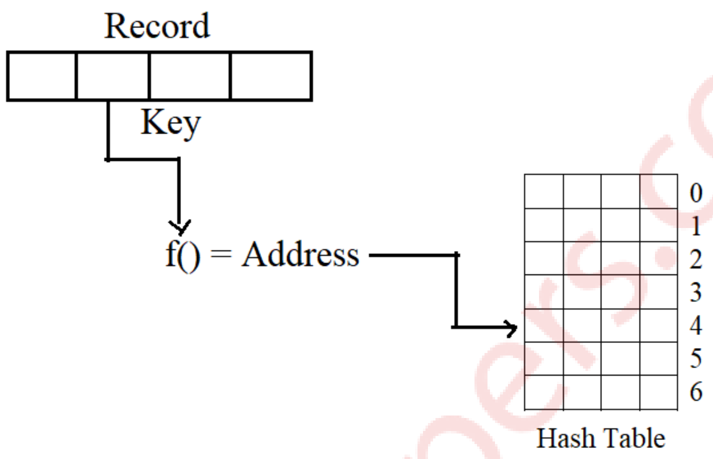
Original Tree

c) **What is hashing? What properties should a good hash function demonstrate? (05)**

➔ Hashing:

- Hashing is a technique by which updating or retrieving any entry can be achieved in constant time O(1).
- In mathematics, a map is a relationship between two sets. A map M is a set of pairs, where each pair is in the form of (key, value). For a given key, its corresponding value can be found with the help of a function that maps keys to values. This function is known as the hash function.
- So, given a key k and a hash function h, we can compute the value/location of the value v by the formula v = h(k).
- Usually the hash function is a division modulo operation, such as h(k)=k mod size, where size is the size of the data structure that holds the values.
- Hashing is a way with the requirement of keeping data sorted.
- In best case time complexity is of constant order O(1) in worst case O(n)

- Address or location of an element or record, x, is obtained by computing some arithmetic function f.f(key) gives the address of x in the table.
- Table used for storing of records is known as hash table.
- Function f(key) is known as hash function.



Mapping of records in hash table

Properties of good hash function:

1. A good hash function avoids collisions.
2. A good hash function tends to spread keys evenly in the array.
3. A good hash function is easy to compute.
4. The hash function should generate different hash values for the similar string.
5. The hash function is a perfect hash function when it uses all the input data.

Q 6

a) Given an array int a[] = {69, 78, 63, 98, 67, 75, 66, 90, 81}. Calculate address of a[5] if base address is 1600. (02)

➔

Address	1600	1604	1608	1612	1616	1620	1624	1628	1632
Elements	69	78	63	98	67	75	66	90	81
Array	0	1	2	3	4	5	6	7	8

Address of A [I] = B + W * (I – LB)

Where, B = Base address 1600 (given)
W = Storage Size of one element stored in the array (in byte) = 4
I = Subscript of element whose address is to be found = 5 (given)
LB = Lower limit / Lower Bound of subscript, if not specified assume 0

$$\begin{aligned}\text{Address of } A[5] &= 1600 + 4 * (5 - 0) \\ &= 1600 + 4 * 5 \\ &= 1600 + 20 \\ A[5] &= 1620\end{aligned}$$

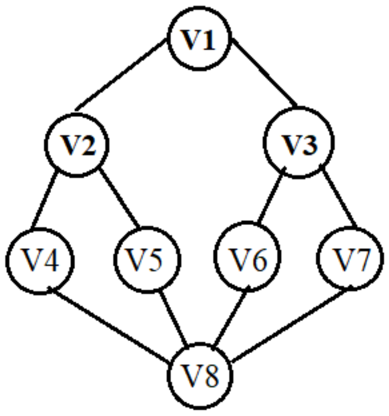
One can verify it from table too A[5] has element 75 stored at address 1620.

b) Give C function for Breadth First Search Traversal of a graph. Explain the code with an example. (10)

➔ C function for Breadth First Search

```
void BFS(int V)
{
    q : a queue type variable;
    initialize q;
    visited[v] = 1; // mark v as visited
    add the vertex V to queue q;
    while(q is not empty)
    {
        v ← delete an element from the queue;
        for all vertices w adjacent from V
        {
            if(!visited[w])
            {
                visited[w] = 1;
                add the vertex w to queue q;
            }
        }
    }
}
```

Example:



Queue	Visited[]	Vertex visited	Action
Null	1 2 3 4 5 6 7 8 0 0 0 0 0 0 0 0	-	-
V1	1 2 3 4 5 6 7 8 1 0 0 0 0 0 0 0	V1	Add (q, v1) visit (V1)
V2 V3	1 2 3 4 5 6 7 8 1 1 1 0 0 0 0 0	V1 V2 V3	Delete (q), add and visit adjacent vertices
V3 V4 V5	1 2 3 4 5 6 7 8 1 1 1 1 1 0 0 0	V1 V2 V3 V4 V5	Delete (q), add and visit adjacent vertices
V4 V5 V6 V7	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 0	V1 V2 V3 V4 V5 V6 V7	Delete (q), add and visit adjacent vertices
V5 V6 V7 V8	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q), add and visit adjacent vertices
V6 V7 V8	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q)
V7 V8	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q), add and visit adjacent vertices
V8	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	V1 V2 V3 V4 V5 V6 V7 V8	Delete (q)
Null	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	V1 V2 V3 V4 V5 V6 V7 V8	Algorithm terminates as the queue is empty

c) Write a C program to implement a singly linked list. The program should be able to perform the following operations: (08)
i) Insert a node at the end of the list
ii) Deleting a particular element
iii) Display the linked list

➔ Program:

```
#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_end();
   int delete_pos();
   void display();

   while(1)
   {
      printf("\n\n---- Singly Linked List(SLL) Menu ----");
      printf("\n1.Insert at end \n2.Delete specific node \n 3.Display
\n4.Exit\n\n");
      printf("Enter your choice(1-4):");
      scanf("%d",&ch);

      switch(ch)
      {
         case 1: insert_end();
                 break;
```

```
          case 2: delete_pos();
                     break;

          case 3: display();
                    break;

          case 4: exit(0);
                    default:
                    printf("Wrong Choice!!");
        }
    }
    return 0;
}

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 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;
}

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;
        }
    }
}
```

Output:

```
---- Singly Linked List(SLL) Menu ----
1.Insert at end
2.Delete specific node
3.Display
4.Exit

Enter your choice (1-4): 1
Enter data: 2

---- Singly Linked List(SLL) Menu ----
1.Insert at end
2.Delete specific node
3.Display
4.Exit

Enter your choice (1-4): 1
Enter data: 3

---- Singly Linked List(SLL) Menu ----
1.Insert at end
2.Delete specific node
3.Display
4.Exit
```

```
Enter your choice (1-4): 1
Enter data: 4

---- Singly Linked List(SLL) Menu ----
1.Insert at end
2.Delete specific node
3.Display
4.Exit

Enter your choice (1-4): 3
The linked list is:
2 -> 3->4->

---- Singly Linked List(SLL) Menu ----
1.Insert at end
2.Delete specific node
3.Display
4.Exit

Enter your choice (1-4): 2
Enter position to delete: 2
Deleted element is 3

---- Singly Linked List(SLL) Menu ----
1.Insert at end
2.Delete specific node
3.Display
4.Exit

Enter your choice (1-4): 3
The linked list is:
2 -> 4->
```

****