DATA STRUCTURES

(MAY 2018)

Q.1

(a) Explain different types of data structure with example. (05)

➔ Data structures are generally categorized into two classes:

1. Primitive Non-primitive Data Structure
   Primitive: Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.
   Non-primitive: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non-linear data structures
2. Linear and Non-linear Structures
   Linear: If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.
   Example:
   1.Linked Lists

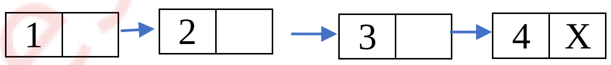


Fig.1 Simple Linked List

2. Stacks

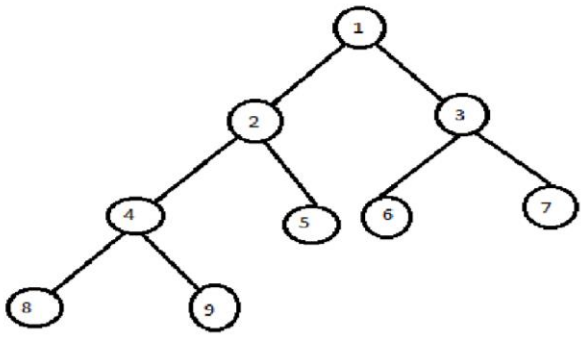
A	AB	ABC	ABCD	ABCDE			
0	1	2	3	TOP=4	5	6	7

Fig.2 Array representation of a stack

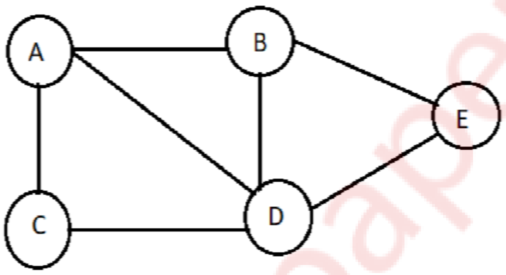
Non-Linear: if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Example:

1. Trees



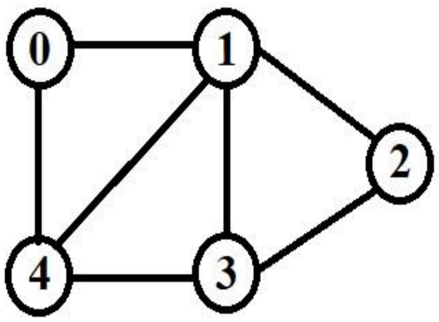
2. Graphs



(b) what is graph? Explain methods to represent graph. (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(digraph). 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.

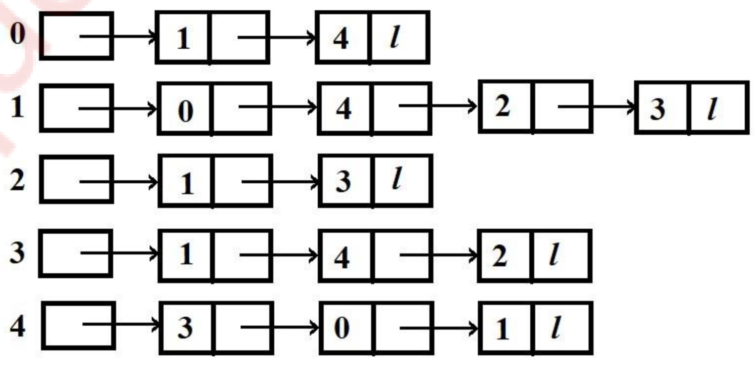


There are two most commonly used representations of a graph.

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) write a program in 'C' to implement Merge sort. (10)

➔ PROGRAM

```
#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);

void main()
{
int arr[size], i, n; printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements of the array: ");
for(i=0;i<n;i++)
{
scanf("%d", &arr[i]);
}
merge_sort(arr, 0, n-1);
printf("\n The sorted array is: \n");
 for(i=0;i<n;i++)
printf(" %d\t", arr[i]);
 getch();
}

void merge(int arr[], int beg, int mid, int end)
{
int i=beg, j=mid+1, index=beg, temp[size], k;
while((i<=mid) && (j<=end))
{
if(arr[i] < arr[j])
{
temp[index] = arr[i];
i++;
}
Else
{
temp[index] = arr[j];
j++;
}
index++;
}
if(i>mid)
{
while(j<=end)
{
temp[index] = arr[j];
j++;
index++;
}
}
Else
{
while(i<=mid)
{
temp[index] = arr[i];
i++;
index++;
}
}
for(k=beg;k<index;k++)
arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
int mid;
if(beg<end)
{
mid = (beg+end)/2;
merge_sort(arr, beg, mid);
merge_sort(arr, mid+1, end);
merge(arr, beg, mid, end);
}
}
```

OUTPUT:

```
Enter the number of elements in the array :5

Enter the elements of the array: 12  45   32   67   88

The sorted array is: 12  32 45  67 88
```

Q.2

(a) Write a program in 'C' to implement QUEUE ADT using Linked-List.Perform the following Operation. (10)
i) Insert node in the queue.
ii) Delete node from the list.
iii) display queue elements.

➔ **PROGRAM**

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
typedef struct node
{
int data;
struct node *link;
}
NODE;
NODE *front=NULL,*rear=NULL,*s,*ptr,*disply;
int main()
{
int no,item; char c;
printf("\n\tPROGRAM QUEUE USING LINKEDLIST");
do
{
printf("\t\t\t\tMENU");
printf("\n\t\t1.INSERT\n\t\t2.DELETE\n\t\t3.DISPLAY\n\t\t4.EXIT\n\t\t
Enter your choice: ");
scanf("%d",&no);
if(no==1)
{
ptr=(NODE*)malloc(sizeof(NODE));
printf("\t\tEnter the element: ");
scanf("%d",&ptr->data);
ptr->link=NULL;
if(rear==NULL)
{
front=ptr; rear=ptr;
}
else
{
rear->link=ptr; rear=ptr;
}
}
if(no==2)
{
if(front==NULL)
printf("\t\tStack is empty\n");
else
{
s=front;
printf("\t\tDeleted Element is %d\n",front->data);
front=front->link;
free(s);
if(front==NULL)
rear=NULL;
}
}
if(no==3)
{
if(front==NULL)
printf("\t\tQueue is empty\n");
else
{
printf("\t\tQueue elements are");
disply=front; while(disply!=NULL)
{
printf(" %d",disply->data);
disply=disply->link;
}
("\n");
}
}
if(no==4)
break;
printf("\t\tDo you want to continue(y/n) ");
scanf(" %c",&c);
}
while(c=='y'||c=='Y');
getch();
}
```

OUTPUT:

```
PROGRAM QUEUE USING LINKEDLIST
 MENU
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice:1
Enter the element:23
Do you want to continue(y/n): y
Enter your choice:1
Enter the element:44
Do you want to continue(y/n): y
Enter your choice:2
Deleted Element is 23
Do you want to continue(y/n): y
Enter your choice:3
Queue elements are 44
Do you want to continue(y/n): n
```

(b) Using Linear probing and Quadratic probing, insert following values in the hash table of size 10. Show how many collisions occur in each iteration 28, 55, 71, 67, 11, 10, 90, 44. (10)

➔

1. Linear Probing

	Empty Table	After 28	After 55	After 71	After 67	After 11	After 10	After 90	After 44
0							10	10	10
1				71	71	71	71	71	71
2						11	11	11	11
3								90	90
4									44
5			55	55	55	55	55	55	55
6									
7					67	67	67	67	67
8	28	28	28	28	28	28	28	28	28
9									

Number of collision=2

2. Quadratic Probing

	Empty table	After 28	After 55	After 71	After 67	After 11	After 10	After 90	After 44	
0							10	10	10	
1				71	71	71	71	71	71	
2						11	11	11	11	*
3								90	90	*
4										
5			55	55	55	55	55	55	55	
6										
7					67	67	67	67	67	
8		28	28	28	28	28	28	28	28	
9									44	*

Q.3

(a) Write a program in 'C' to evaluate postfix expression using STACK ADT. (10)

➔ Program

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=–1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
 {
float val;
char exp[100];
clrscr();
printf("\n Enter any postfix expression : ");
gets(exp);
val = evaluatePostfixExp(exp);
printf("\n Value of the postfix expression = %.2f", val);
getch();
return 0;
}
float evaluatePostfixExp(char exp[])
{
int i=0;
float op1, op2, value;
while(exp[i] != '\0')
{
if(isdigit(exp[i]))
push(st, (float)(exp[i]–'0'));
else
{
  op2 = pop(st);
  op1 = pop(st);
  switch(exp[i])
  {
        case '+':
              value = op1 + op2;
              break;
```

```
            case '–':
                   value = op1 – op2;
                   break;
            case '/':
                    value = op1 / op2;
                     break;
            case '*':
                     value = op1 * op2;
                      break;
            case '%':
                 value = (int)op1 % (int)op2;
                 break;
 }
  push(st, value);
    }
   i++;
  }
  return(pop(st));
  }
  void push(float st[], float val)
 {
  if(top==MAX–1)
   printf("\n STACK OVERFLOW");
  else
  {
   top++;  st[top]=val;
  }
 }
 float pop(float st[])
  {
 float val=–1; if(top==–1)
   printf("\n STACK UNDERFLOW");
  else
  {
  val=st[top];
   top—;
 }
 return val;
  }
```

```
OUTPUT:

Enter any postfix expression : 9 – ((3 * 4) + 8) / 4

Value of the postfix expression 9 3 4 * 8 + 4 / –
```

(b) Explain different types of tree traversals techniques with example. Also write recursive function for each traversal technique. (10)

➔ TRAVERSING A BINARY TREE

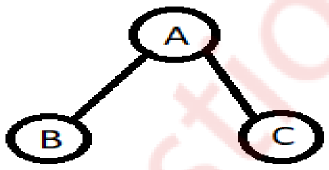
Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.

1. Pre-order Traversal

   - To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

   1. Visiting the root node.
   2. Traversing the left sub-tree, and finally
   3. Traversing the right sub-tree.

   - Example



   - The pre-order traversal of the tree is given as A, B, C. Root node first, the left sub-tree next, and then the right sub-tree.
   - Recursive Function

```
Void preorder (node * T) /* address of the root node is passed in T */
{
                if (T! = NULL)
                 {
                      printf("\n%d, T --> data);
                      preorder( T -->left);
                      preorder( T -->right);

                 }
}
```

2. In-order Traversal

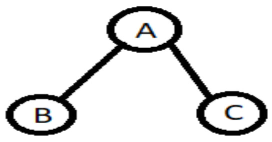
   - To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

   1. Traversing the left sub-tree.

   2. Visiting the root node, and finally

   3. Traversing the right sub-tree.

   - Example



   - The pre-order traversal of the tree is given as B,A,C . Left sub-tree first, the root node next, and then the right sub-tree.
   - Recursive Function

```
Void inorder (node * T) /* address of the root node is passed in T */
{
                if (T! = NULL)
                 {
                      inorder( T -->left);
                      printf(“\n%d, T --> data);
                      inorder( T -->right);

                 }
}
```

3. Post-order Traversal

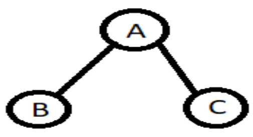
   - To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

   1. Traversing the left sub-tree.

   2. Traversing the right sub-tree, and finally

   3. Visiting the root node.

   - Example



- The pre-order traversal of the tree is given as B,C,A . Left sub-tree first, the right sub-tree and then the root node .
- Recursive Function

```
Void postorder (node * T) /* address of the root node is passed in T
*/
{
                 if (T! = NULL)
                  {
                       postorder( T -->left);
                       postorder( T -->right);
                       printf("\n%d, T --> data);


                  }
}
```

Q.4

(a) State advantages of Linked List over arrays. Explain different applications of Linked List. (10)

➔ Linked list over arrays

- Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations.
- Another point of difference between an array and a linked list is that a linked list does not allow random access of data
- Nodes in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.
- Another advantage of a linked list over an array is that we can add any number of elements in the list
- linked lists provide an efficient way of storing related data and performing basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing the address of next nodes.

Application of Linked list

- Linked lists can be used to represent polynomials and the different operations that can be performed on them
- we will see how polynomials are represented in the memory using linked lists.

1. Polynomial representation

- Let us see how a polynomial is represented in the memory using a linked list.
- Consider a polynomial 6x3 + 9x2 + 7x + 1. . Every individual term in a polynomial consists of two parts, a coefficient and a power.
- Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.
- Every term of a polynomial can be represented as a node of the linked list. Figure shows the linked representation of the terms of the above polynomial.

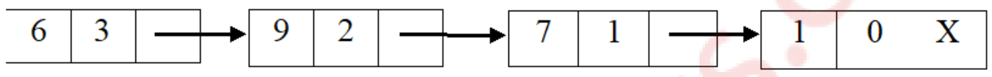


Figure. Linked representation of a polynomial

- Now that we know how polynomials are represented using nodes of a linked list.

(b) Write a program in 'C' to implement Circular queue using arrays. (10)

➔ <u>Program</u>

```
#include<stdio.h>
#include<conio.h>
#define size 5
int q[size],front=-1,rear=-1,i,element;
void insert(int ele);
int del();
void disp();

void main()
{
int ch,ele;
clrscr();
printf("\t ***** Main Menu *****");
printf("\n 1. insert \n2.delete \n3.display \n4.Exit\n");
do
{
printf("\n Enter your choice: \n \n");
scanf("%d",&ch);
switch(ch)
{
case 1:printf("\n Enter Element to Insert \n ");
scanf("%d",&ele);
insert(ele);
disp();
break;

case 2:ele=del();
scanf("\n %d is the deleted element \n ",ele);
disp();
break;

case 3:disp();
break;

case 4:break;
default:printf(" \n Invalid Statement \n");
}
}
while(ch!=4); getch();
}

void insert(int ele)
{
if(front==-1 && rear==-1)
{
front=rear=0;    q[rear]=ele;
}
else if((rear+1)%size==front)
{
printf("\n  Queue is Full \n");
}
else
{
rear=(rear+1)%size;
q[rear]=ele;
}
}

int del()
{
if(rear==-1 && front==-1)
{
printf("\n Queue is Empty \n");
}
```

```
else if(rear==front)
{
rear=-1;
front=-1;
printf("\n Queue is Empty \n");
}
else
{
element=q[front];    front=(front+1)%size;
}
return element;
}

void disp()
{
if(rear==-1 && front==-1)
{
printf("\n Queue is Empty \n");
}
else
{
for(i=front;i<(rear+1)%size;i++)
{
printf("\t %d",q[i]);
}
}
}
```

OUTPUT:

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

1. insert

2.delete

3.display

4.Exit

 Enter your choice: 1

Enter Element to Insert: 23

Enter your choice: 1

Enter Element to Insert: 45
Enter your choice: 2

23 is the deleted element

Enter your choice: 3

45

Enter your choice: 4
```

Q.5

(a) Write a program to implement singly Linked List. Provide the following operations:
i) Insert a node at the specified location.
ii) Delete a node from end
iii) Display the list **(10)**

➔ PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void InsertAtPosition(int value,int position);
void RemoveAtEnd();
void display();
int CheckEmpty();

struct Node{
int data;
struct Node *next;
}* head=NULL;

int main()
{
int value,choice;
char c;
do{
printf("Enter\n1-Insert\n2-Remove\n3-Display\n");
printf("Enter your choice");
scanf("%d",&choice);
switch(choice)
{
```

```
case 1:
{
int x;
printf("Enter \n1-Insert at Position\n");
scanf("%d",&x);
printf("Enter Value to be Inserted\n");
scanf("%d",&value);
switch(x)
{
case 1:
{
int position;
printf("Enter position to insert a value(counted from 0)\n");
scanf("%d",&position);
InsertAtPosition(value,position);
break;
}
default :
{
printf("Enter Valid Choice\n");
break;
}
}
break;
}

case 2:
{
int x;
printf("Enter \n1-Delete At End\ n");
scanf("%d",&x);
switch(x)
{
case 1:
{
RemoveAtEnd();
break;
}
default :
{
printf("Enter Valid Choice\n");
break;
}
}
break;
}

case 3:
{
display();
break;
}
default:
{
printf("Enter Valid Choice\n");
break;
}
}
printf("Enter 'Y' to continue else any letter\n");
fflush(stdin);
c=getche();
printf("\n");
}while(c=='Y' || c=='y');
return(0);
}

void InsertAtPosition(int value,int position){
struct Node *newNumber,*temp;
int count,flag;
newNumber = (struct Node*)malloc(sizeof(struct Node));
newNumber->data = value;
temp=head;
flag=CheckEmpty();
if(flag==1)
{
int flag1=0;
count=0;
while(temp!=NULL)
{
if(count==position-1)
{
flag1=1;
newNumber->next=temp->next;
temp->next=newNumber;
}
```

```
else
{
temp=temp->next;
}
count++;
}
if(flag1==0)
{
printf("Entered Position Not available\n");
}
else
{
printf("Given number %d is inserted at position %d
successfully\n",value,position);
}
}
else
{
printf("List is Empty\n");
}
}

void RemoveAtEnd()
{
int flag=CheckEmpty();
if(flag==1)
{
if(head->next==NULL)
{
head=NULL;
}
else
{
struct Node *temp=head,*temp1;
while(temp->next!=NULL)
{
temp1=temp;
temp=temp->next;
}
temp1->next=NULL;
free(temp);
}
}
```

```
else
{
printf("List Empty.Try again!\n");
}
}
void display()
{
int flag=CheckEmpty();
if(flag==1)
{
struct Node *temp;
temp=head;
while(temp->next!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
printf("%d",temp->data);
printf("\n");
}
else
{
printf("No List Available\n");
}
}

int CheckEmpty()
{
if(head==NULL)
return 0;
else
return 1;
}
```

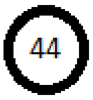
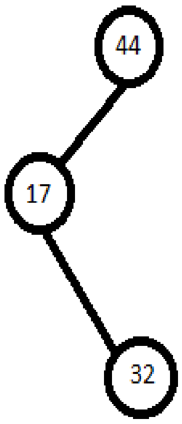
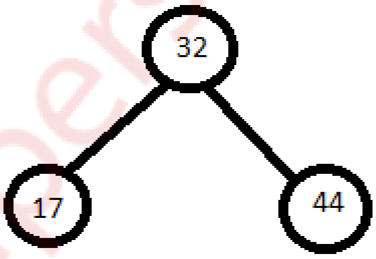
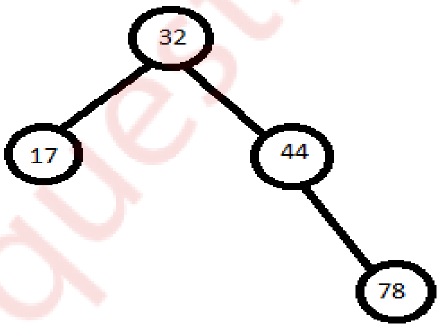
OUTPUT:

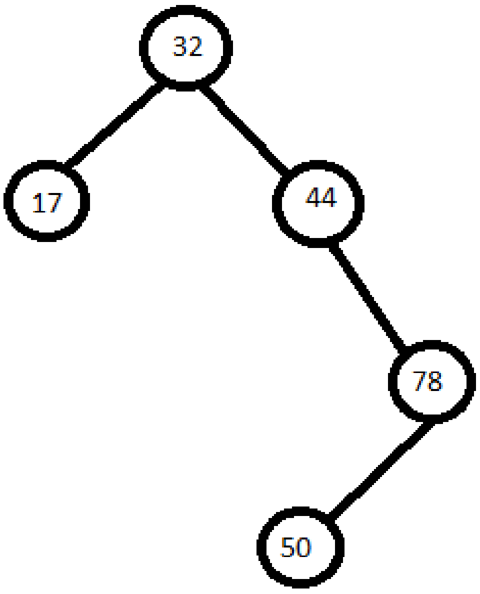
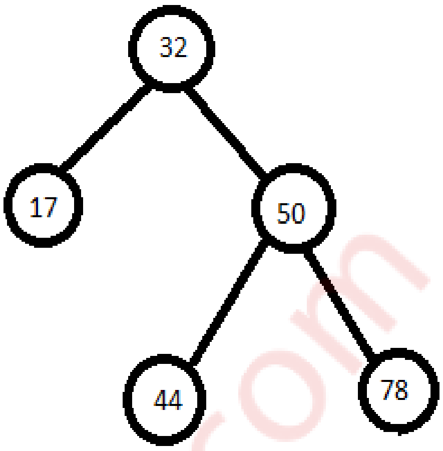
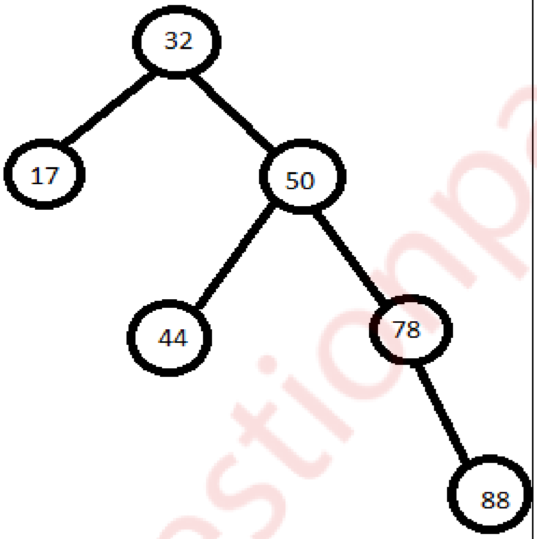
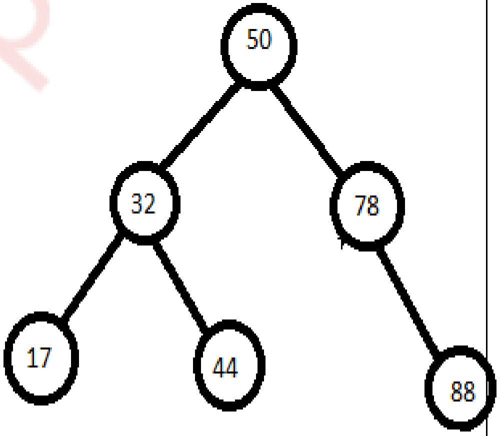
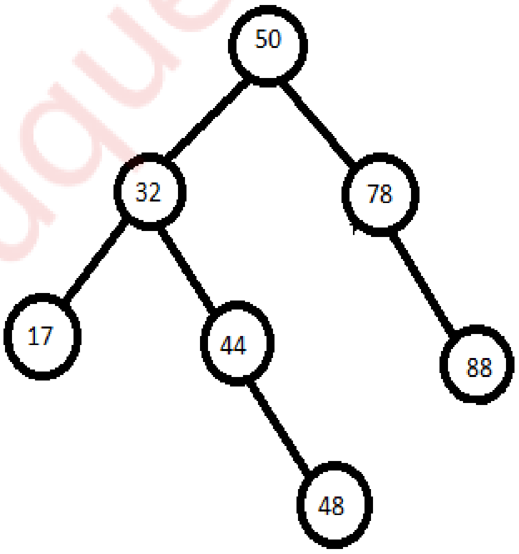
```
1.Insert
2.Remove
3.Display
Enter your choice:1
Enter
1-Insert at position
Enter value to be inserted
45
```

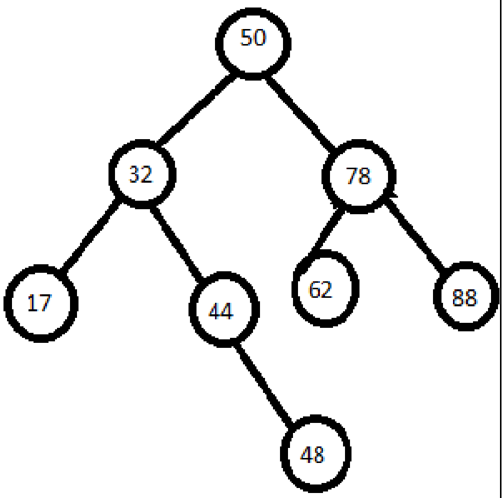
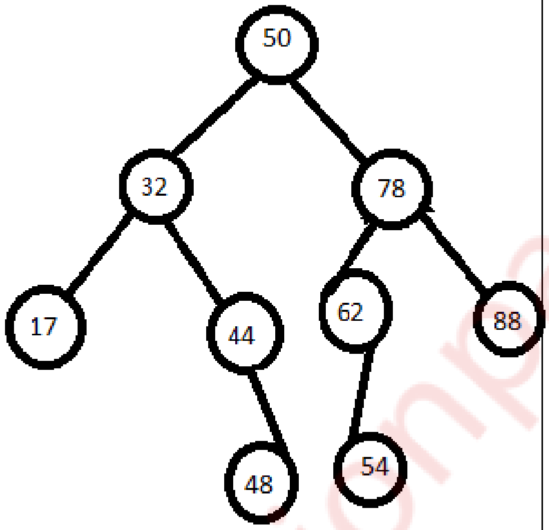
```
Enter position to insert a value(counted from 0)
2
Given number 45 is inserted at position 2 successfully
Enter ‘Y’ to continue else any letter
Y
Enter
1.Insert
2.Remove
3.Display
Enter your choice:3
45
Enter ‘Y’ to continue else any letter
Y
Enter
1.Insert
2.Remove
3.Display
Enter your choice:2
Enter
1-Delete at end
1
```

(b) Insert the following elements in AVL tree: 44, 17, 32, 78, 50, 88, 48, 62, 54. Explain different rotation that can be used. (10)

➔

Sr. no	Data to be insert	Tree after insertion	Tree after Rotation
1	44		
2	17,32		
3	78		

4	**50**	32 17 44 78 50	32 17 50 44 78
5	**88**	32 17 50 44 78 88	50 32 78 17 44 88
6	**48**	50 32 78 17 44 88 48	

7	**62**		
8	**54**		

Q.6 Explain the following (any two) **(20)**

(a) splay Tree and Trie

<u>Splay Tree</u>

- A splay tree consists of a binary tree, with no additional fields.
- When a node in a splay tree is accessed, it is rotated or 'splayed' to the root, thereby changing the structure of the tree.
- Since the most frequently accessed node is always moved closer to the starting point of the search (or the root node), these nodes are therefore located faster.
- A simple idea behind it is that if an element is accessed, it is likely that it will be accessed again.
- In a splay tree, operations such as insertion, search, and deletion are combined with one basic operation called splaying.

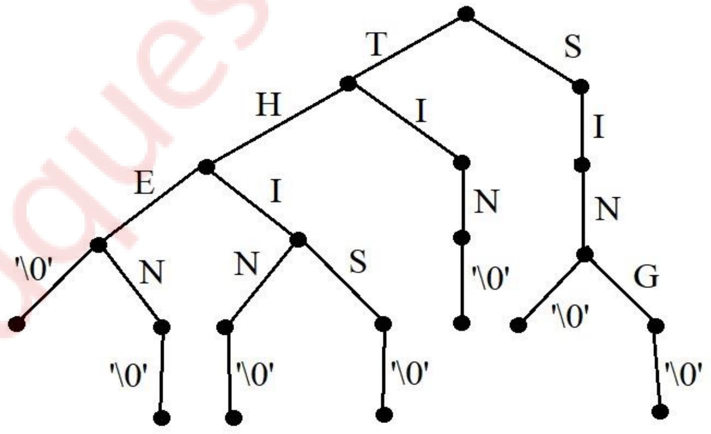
- Splaying the tree for a particular node rearranges the tree to place that node at the root.
- A technique to do this is to first perform a standard binary tree search for that node and then use rotations in a specific order to bring the node on top.

Advantages and Disadvantages of Splay Trees

- A splay tree gives good performance for search, insertion, and deletion operations.
- This advantage centres on the fact that the splay tree is a self-balancing and a self-optimizing data structure.
- Splay trees are considerably simpler to implement.
- Splay trees minimize memory requirements as they do not store any book-keeping data.
- Unlike other types of self-balancing trees, splay trees provide good performance.

<u>Trie</u>

- 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.

(b) Graph Traversal Techniques

➔ There are two standard methods of graph traversal.

1. Breadth-first search
   - Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes.
   - Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
   - That is, we start examining the node A and then all the neighbours of A are examined.

   Algorithm for Breadth-first search
   - Step 1: SET STATUS=1(ready state) for each node in G
   - Step 2: Enqueue the starting node A and set its STATUS=2 (waiting state)
   - Step 3: Repeat Steps4and5until QUEUE is empty
   - Step 4: Dequeueanode N. Process it and set its STATUS=3 (processed state).
   - Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2 (waiting state) [END OF LOOP]
   - Step 6: EXIT

   Applications of Breadth-First Search Algorithm
   - Breadth-first search can be used to solve many problems such as:
   - Finding all connected components in a graph G.
   - Finding all nodes within an individual connected component.
   - Finding the shortest path between two nodes, u and v, of an unweighted graph.
   - Finding the shortest path between two nodes, u and v, of a weighted graph.

2. Depth-first Search
   - The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
   - When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
   - depth-first search begins at a starting node A which becomes the current node.
   - Then, it examines each node N along a path P which begins at A.

- That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.

Algorithm for depth-first search

- Step 1: SET STATUS=1(ready state) for each node in G
- Step 2: Push the starting nodeAon the stack and set its STATUS=2(waiting state)
- Step 3: Repeat Steps4and5until STACK is empty
- Step 4: Pop the top node N. Process it and set its STATUS=3(processed state)
- Step 5: Push on the stack all the neighbours ofNthat are in the ready state (whose STATUS=1) and set their STATUS=2(waiting state) [END OF LOOP]
- Step 6: EXIT

(c) Huffman Encoding

 Huffman Code:

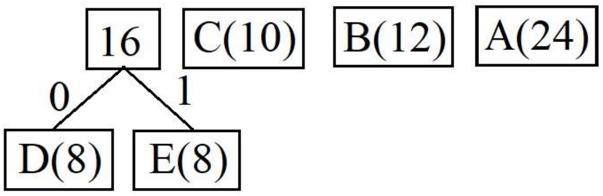
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages M1, M2, …Mn - Message is converted into a binary string.
- Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.
- 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

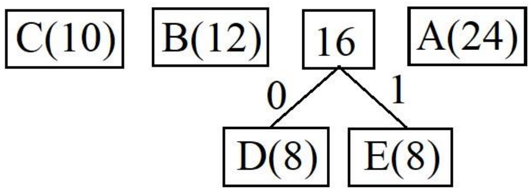
Arrange the message in ascending order according to their frequency

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

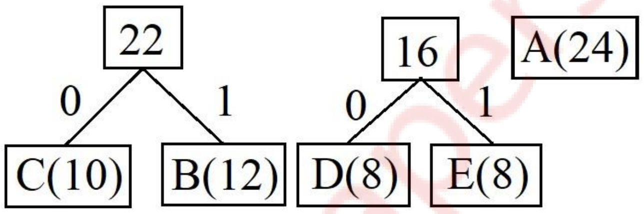
Merge two minimum frequency message



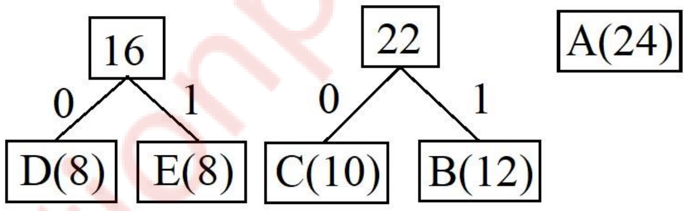
Rearrange in ascending order



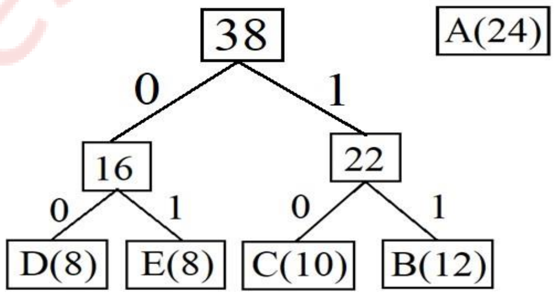
Merge two minimum frequency message



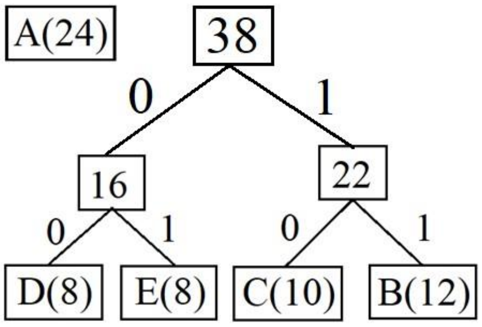
Rearrange in ascending order



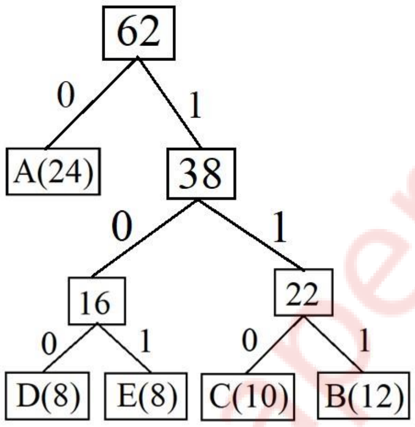
Merge two minimum frequency message



Again Rearrange in ascending order



Merge two minimum frequency message



Huffman code

A = 0
B = 111
C = 110
D= 100
E = 101

(d) Double Ended Queue

➔

- It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end.
- However, no element can be added and deleted from the middle
- In a deue, two pointers are maintained, LEFT and RIGHT, which point to either endof the deque.
- They include,
  Input restricted deque – In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
  The following operations are possible in an input restricted dequeues.

  i) Insertion of an element at the rear end and
  ii)Deletion of an element from front end
  iii)Deletion of an element from rear end

Output restricted deque- In this dequeue, deletions can be done only at one of the ends,while insertions can be done on both ends.

i) Deletion of an element at the front end
ii)Insertion of an element from rear end
iii)Insertion of an element from rear end

Operations on a dequeue

i. initialize(): Make the queue empty.
ii. empty(): Determine if queue is empty.
iii. full(): Determine if queue is full.
iv. enqueueF(): Insert at element at the front end of the queue.
v. enqueueR(): Insert at element at the rear end of the queue.
vi. dequeueF(): Delete the front end
vii. dequeueR(): Delete the rear end
viii. print(): print elements of the queue.

- **There are various methods to implement a dequeue.**
- Using a circular array
- Using a linked list
- Using a cicular linked list
- Using a doubly linked list
- Using a doubly cirular linked list

			29	37	45	54	63		
0	1	2	left = 3	4	5	6	right = 7	8	9

42	56						63	27	18
0	right = 1	2	3	4	5	6	left = 7	8	9

Fig. **Double-ended queues**

****