DATA STRUCTURES

(MAY 2019)

Q.1

(a) Explain Linear and Non-Linear data structures. (5)

➔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

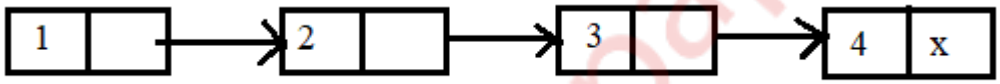


Fig.1

Simple Linked List

2. Stacks

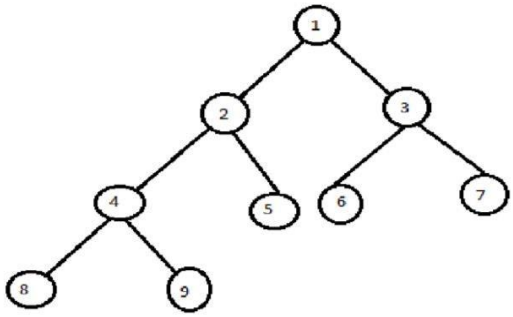
A	AB	ABC	ABCD	ABCDE			
0	1	2	3	TOP=4	5	6	7

Fig.2 Array representation of a stack

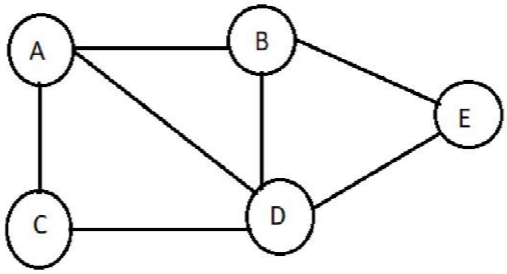
Non-Linear: if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Example:

1. Trees



2. Graphs



(b) Explain Priority Queue with example. (5)

➔Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low If two elements have the same priority, they are served according to their order in the queue.
- A typical priority queue supports following operations.
- insert(item, priority): Inserts an item with given priority.
- getHighestPriority(): Returns the highest priority item.
- deleteHighestPriority(): Removes the highest priority item.
- When arrays are used to implement a priority queue, then a separate queue for each priority numberis maintained. Each of these queues will be implemented using circulararrays or circular queues. Every individual queue will have itsown FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue wil be allocated the same amount of space.
- FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number.

Example:

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1				A	
2	B	C	D		
3				E	F
4	I			G	H

Priority Queue matrix

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1				A	
2	B	C	D		
3	R			E	F
4	I			G	H

Priority Queue Matrix after insertion

Of an element

- To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element.
- . In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed cc first.

(c) Write a Programe in 'c' to impliment Quick sort. (10)

Program:

```
#include<stdio.h>
#include<conio.h>
void quicksort(int number[25],int first,int last)
{
int i,j,pivot,temp;
if(first<last)
{
pivot=first;
i=first;
j=last;
```

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

}
}
temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);
}
}
int main()
{
int i,count,number[25];
printf("How many elements are u going to enter?");
scanf("%d",&count);
```

```
printf("enter %d element:",count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("Order of sorted elements");
for(i=0;i<count;i++)
scanf("%d",&number[i]);
return 0;
}
```

OUTPUT:

```
How many elements are u going to enter?4
enter 4 element:34
56
22
31
Order of sorted elements
22
31
34
56
```

Q.2

(a) Write a programe to impliment Circular Lined list Provide the following oparation: (10)

(i) Insert a node

(ii) Delete a node

(iv) Display the list

Program:

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

```
#include <stdlib.h>
#include <stdbool.h>
struct node
{
int data;
int key;
struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;
bool isEmpty()
{
return head == NULL;
}
int length()
{
int length = 0;
if(head == NULL)
{
return 0;
}
current = head->next;
while(current != head)
{
length++;
current = current->next;
}
return length;
}
```

```
void insertFirst(int key, int data)
{
struct node *link = (struct node*) malloc(sizeof(struct node));
link->key = key;
link->data = data;
if (isEmpty())
{
head = link;
head->next = head;
}
else
{
link->next = head;
head = link;
}
}
struct node * deleteFirst()
{
struct node *tempLink = head;
if(head->next == head)
{
head = NULL;
return tempLink;
}
head = head->next;
return tempLink;
}
void printList()
{
```

```
struct node *ptr = head;
printf("\n[ "); if(head != NULL)
{
while(ptr->next != ptr)
{
printf("(%d,%d) ",ptr->key,ptr->data);
ptr = ptr->next;
}
}
printf(" ]");
}
main()
{
insertFirst(1,10);
insertFirst(2,20);
insertFirst(3,30);
insertFirst(4,1);
insertFirst(5,40);
insertFirst(6,56);
printf("Original List: ");
printList();
while(!isEmpty())
{
struct node *temp = deleteFirst();
printf("\nDeleted value:");
printf("(%d,%d) ",temp->key,temp->data);
}
printf("\nList after deleting all items: ");
printList();
```

}

OUTPUT:

```
Original List:

[ (6,56) (5, 40) (4,1) (3,30) (2,20)  (1,10) ]

Deleted value: (6,56)

Deleted value: (5, 40)

Deleted value: (4,1)

Deleted value:  (3,30)

Deleted value: (2,20)

Deleted value: (1,10)

nList after deleting all items:
[   ]
```

(b) Explain Threaded Binary tree in detail (10)

➔ Threaded Binary Tree:

- A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.
- In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both.
- For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node.
- **These special pointers are called threads and binary trees containing threads are called threaded trees.**
- There are many ways of threading a binary tree and each type may vary according to the way the tree is traversed.
- 1. One-way Threading
  2. Two-way Threading

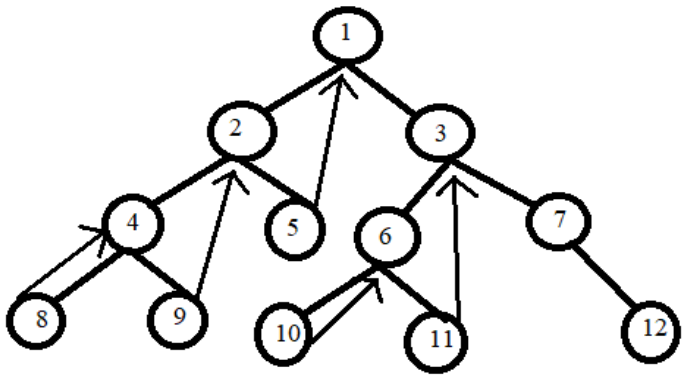


Fig1. Binary tree with one-way threading

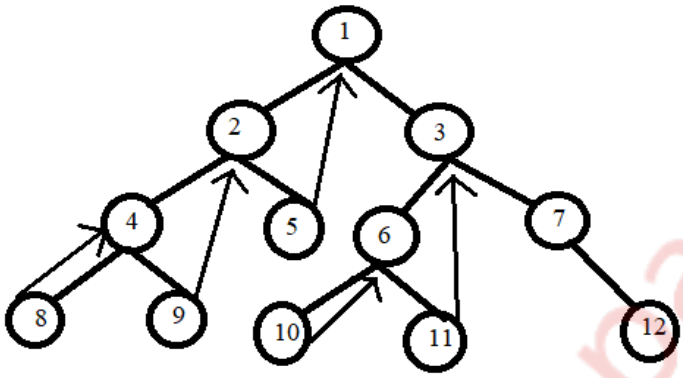


Fig2. Binary tree with two-way threading

- Apart from this, a threaded binary tree may correspond to one-way threading or a twoway threading.
- In one-way threading, a thread will appear either in the right field or the left field of the node.
- A one-way threaded tree is also called a single-threaded tree.
- one-way threaded tree is called a rightthreaded binary tree.
- In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node.
- A two-way threaded binary tree is also called a fully threaded binary tree.
- Advantages of Threaded Binary Tree:
    1. It enables linear traversal of elements in the tree.
    2. Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
    3. It enables to find the parent of a given element without explicit use of parent pointers.

4. Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.

- we see the basic difference between a binary tree and a threaded binary tree is that in binary trees a node stores a NULL pointer if it has no child and so there is no way to traverse back.

Q.3

(a) Explain Huffman Encoding with suitable example (10)

Huffman Code:

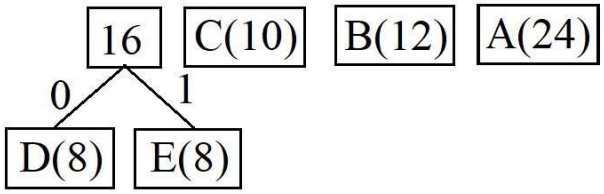
- Huffman code is an application of binary trees with minimum weighted external path length is to obtain an optimal set for messages M1, M2, …Mn - Message is converted into a binary string.
- Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.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

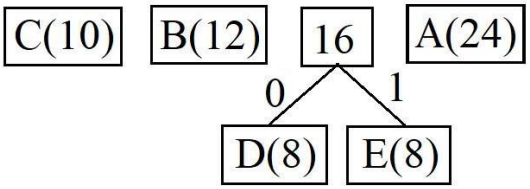
Arrange the message in ascending order according to their frequency

D(8) E(8) C(10) B(12) A(24)

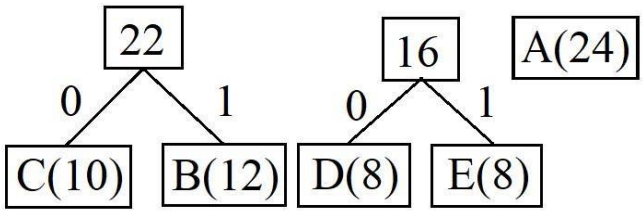
Merge two minimum frequency message



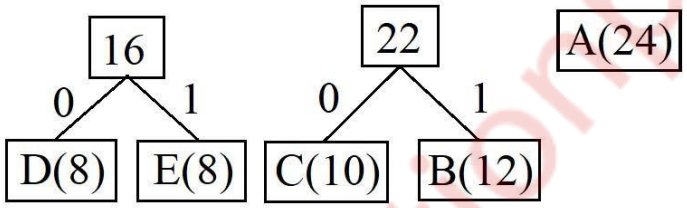
Rearrange in ascending order



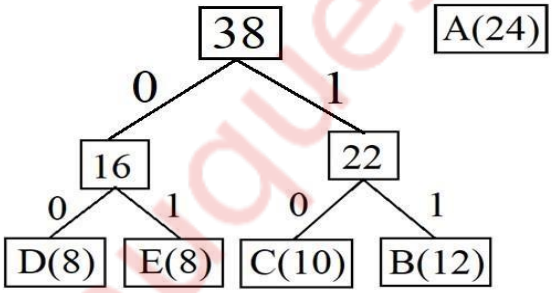
Merge two minimum frequency message



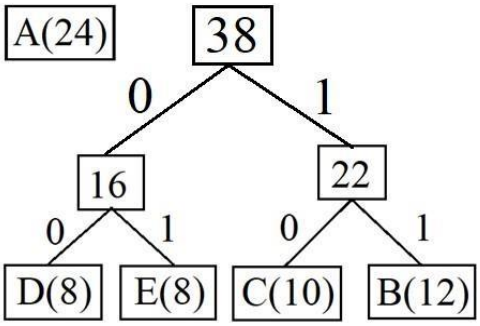
Rearrange in ascending order



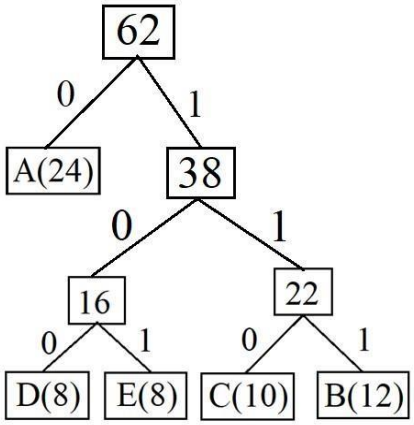
Merge two minimum frequency message



Again Rearrange in ascending order



Merge two minimum frequency message



Huffman code

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

(b) Write a program in 'C' to check for balanced parenthesis in an expression using stack. (10)

➔ Program

```
#include <stdio.h>
#include <string.h>
#define MAXSIZE 100
#define TRUE 1
#define FALSE 0
struct Stack {
int top;
```

```
int array[MAXSIZE];
} st;
void initialize() {
st.top = -1;
}
int isFull() {
if(st.top >= MAXSIZE-1)
return TRUE;
else
return FALSE;
}
int isEmpty() {
if(st.top == -1)
return TRUE;
else
return FALSE;
}
void push(int num) {
if (isFull())
printf("Stack is Full...\n");
else {
st.array[st.top + 1] = num;
st.top++;
}
}
int pop() {
if (isEmpty())
printf("Stack is Empty...\n");
else {
st.top = st.top - 1;
return st.array[st.top+1];
}
}
int main() {
char inputString[100], c;
int i, length;
initialize();
printf("Enter a string of paranthesis\n");
gets(inputString);
length = strlen(inputString);
for(i = 0; i < length; i++){
if(inputString[i] == '{')
push(inputString[i]);
```

```
else if(inputString[i] == '}')
pop();
else {
printf("Error : Invalid Character !! \n");
return 0;
}
}

if(isEmpty())
printf("Valid Paranthesis Expression\n");
else
printf("InValid Paranthesis Expression\n");

return 0;
}
```

OUTPUT:

```
Enter a string of paranthesis
{{{}{}{{}}{}}}
Valid Paranthesis Expression

Enter a string of paranthesis
{{{}{}{}}}{}{}{{}
InValid Paranthesis Expression
```

Q.4

(a) Write a program in 'C' to implement Queue using array. (10)

➔ Program

```
#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
```

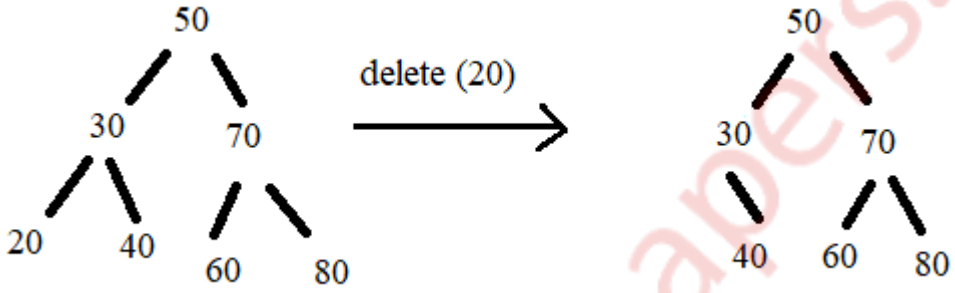
```
Enter your choice: 2
23 is the deleted element
Enter your choice: 3
```

```
45
Enter your choice: 4
```

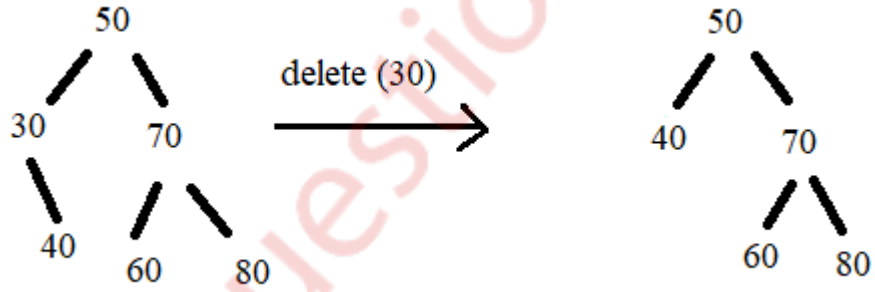
(b) Explain different cases for deletion of a node in binary search tree. Write function for each case (10)

➔ When we delete a node, three possibilities arise.

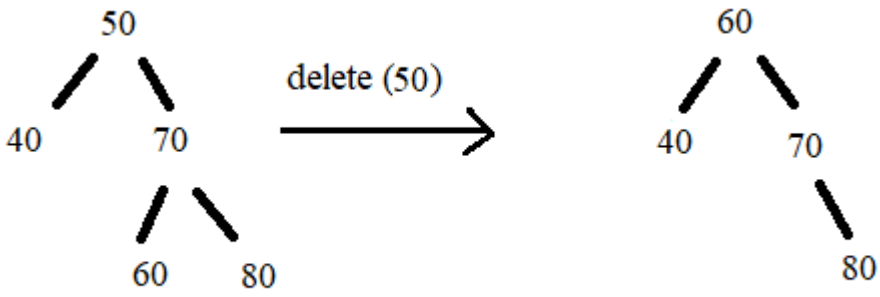
1) ***Node to be deleted is leaf:*** Simply remove from the tree.



2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



- The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.
- C Function:

```
void deletion(Node*& root, int item)
{
Node* parent = NULL;
Node* cur = root;
search(cur, item, parent);
if (cur == NULL)
return;
if (cur->left == NULL && cur->right == NULL)
{
if (cur != root)
{
if (parent->left == cur)
parent->left = NULL;
else
parent->right = NULL;
}
else
root = NULL;
free(cur);
}
else if (cur->left && cur->right)
{
Node* succ  = findMinimum(cur- >right);
int val = succ->data;
deletion(root, succ->data);
cur->data = val;
 }
```

```
else
{
Node* child = (cur->left)? Cur- >left: cur->right;
if (cur != root)
{
if (cur == parent->left)
parent->left = child;
else
parent->right = child;
}
else
root = child;
free(cur);
}
}

Node* findMinimum(Node* cur)
{
while(cur->left != NULL) {
cur = cur->left;
}
return cur;
}
```

Q.5

(a) Write a program in 'C' to implement Stack using Linked-List. Perform the following operation:

(i) Push

(ii) Pop

(iii) Peek

(iii) Display the stack contents **(10)**

PROGRAM:

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

```
{
int info;
struct node *ptr;
}*top,*top1,*temp;
int topelement();
void push(int data);
void pop();
void display();
void peek();
void create();
int count = 0;
void main()
{
int no, ch, e;
printf("\n 1 - Push");
printf("\n 2 - Pop");
printf("\n 3 - Peek");
printf("\n 4 - Exit");
printf("\n 5 - Dipslay");
create();
while (1)
{
printf("\n Enter choice : ");
scanf("%d", &ch);
switch (ch)
{
case 1:
printf("Enter data : ");
scanf("%d", &no);
```

```
push(no);
break;
case 2:
pop();
break;
case 3:
if (top == NULL)
printf("No elements in stack");
else
{
e = topelement();
printf("\n Top element : %d", e);
}
break;
case 4:
exit(0);
case 5:
display();
break;
default :
printf(" Wrong choice, Please enter correct choice  ");
break;
}
}
}
void create()
{
top = NULL;
}
```

```
void push(int data)
{
if (top == NULL)
{
top =(struct node *)malloc(1*sizeof(struct node));
top->ptr = NULL;
top->info = data;
}
else
{
temp =(struct node *)malloc(1*sizeof(struct node));
temp->ptr = top;
temp->info = data;
top = temp;
}
count++;
}
void display()
{
top1 = top;
if (top1 == NULL)
{
printf("Stack is empty");
return;
}
while (top1 != NULL)
{
printf("%d ", top1->info);
top1 = top1->ptr;
```

```
}
}
void peek()
{
top1 = top;
if (top1 == NULL)
{
printf("\n Error : Trying to pop from empty stack");
return;
}
else
top1 = top1->ptr;
printf("\n Popped value : %d", top->info);
free(top);
top = top1;
count--;
}
int topelement()
{
return(top->info);
}
```

OUTPUT:

```
1- Push
2- Pop
3- Peek
4- Display
5- Exit
Enter choice : 1
Enter data : 32

Enter choice : 1
Enter data : 34

Enter choice : 2
Popped value : 34

Enter choice : 3
Peek element : 32

Enter choice : 4
32

Enter choice : 5
```

(b) Explain Depth First search (DFS) Traversal with an example. Write the recursive function for DFS (10)

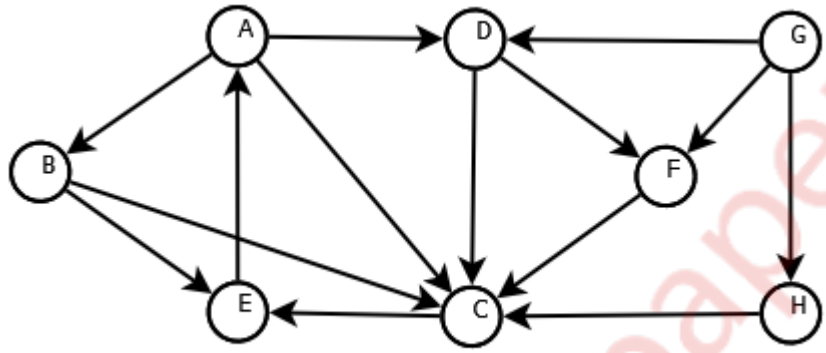
➔Depth-first Search

- 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

Example:



- Adjacency list for G:
  - A: B, C, D
  - B: C, E
  - C: E
  - D: C, F
  - E: A
  - F: C
  - G: D, F, H
  - H: C

Recursive Function:

```
void Graph::DFSUtil(int v, bool visited[])
{
visited[v] = true;
cout << v << " ";

// Recur for all the vertices adjacent
// to this vertex
```

```
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
if (!visited[*i])
DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
// Mark all the vertices as not visited
bool *visited = new bool[V];
for (int i = 0; i < V; i++)
visited[i] = false;

// Call the recursive helper function
// to print DFS traversal
DFSUtil(v, visited);
}
```

Q.6. Write Short notes on (any two) (20)

(a) Application of Linked-List –Polynomial addition

➔ 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.

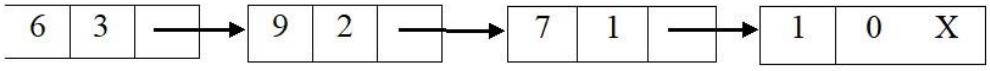


Figure. Linked representation of a polynomial

- Now that we know how polynomials are represented using nodes of a linked list.
- Example:
  Input:
  1st number = 5x^2 + 4x^1 + 2x^0
  2nd number = 5x^1 + 5x^0
  Output:
  5x^2 + 9x^1 + 7x^0
  Input:
  1st number = 5x^3 + 4x^2 + 2x^0
  2nd number = 5x^1 + 5x^0
  Output:
  5x^3 + 4x^2 + 5x^1 + 7x^0

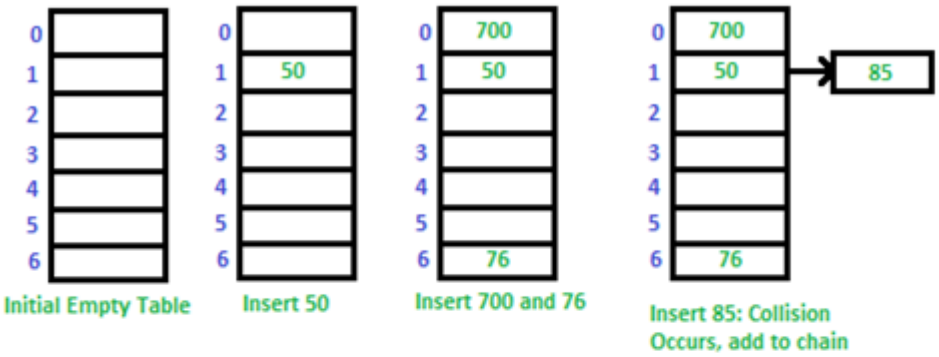
(b) Collision Handling technique

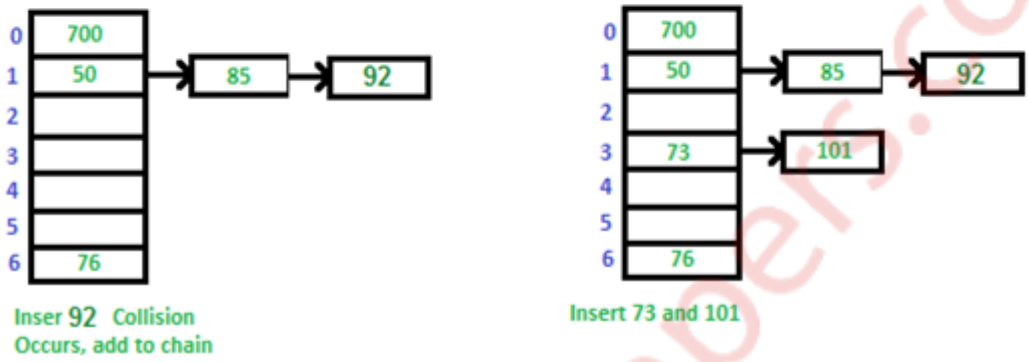
➔ There are mainly two methods to handle collision:

1) Separate Chaining
2) Open Addressing

1. Separate Chaining

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
- To handle collisions, the hash table has a technique known as **separate chaining**. **Separate chaining** is defined as a method by which linked lists of values are built in association with each location within the hash table when a collision occurs.
- The concept of separate chaining involves a technique in which each index key is built with a linked list. This means that the table's cells have linked lists governed by the same hash function.
- Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.





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

2. Open Addressing

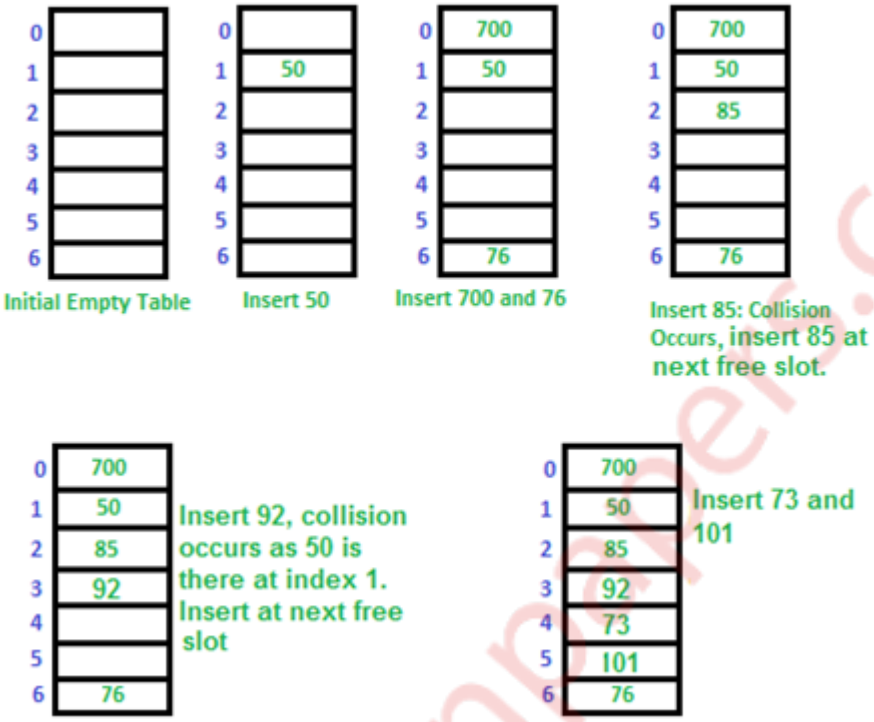
- Like separate chaining, open addressing is a method for handling collisions.
- In Open Addressing, all elements are stored in the hash table itself.
- So at any point, size of the table must be greater than or equal to the total number of keys
- In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell.
- This technique is called linear probing.

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let **hash(x)** be the slot index computed using hash function and **S** be the table size

a) Qudratic Probing: Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

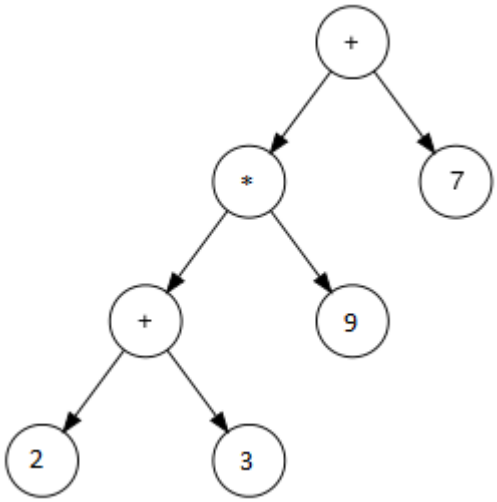
- Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



(c) Expression Tree

➔ Expression Tree

- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- A binary expression tree is a specific kind of a binary tree used to represent expressions.
- The leaves of the binary expression tree are operands, such as constants or variable names, and the other nodes contain operators.
- Assume the set of possible operators are {'+', '-', '*', '/'}. The set of possible operands are ['0' - '9']. See the figure below, which is the binary expression tree for the expression (in in-fix notation): (((2+3)*9)+7).

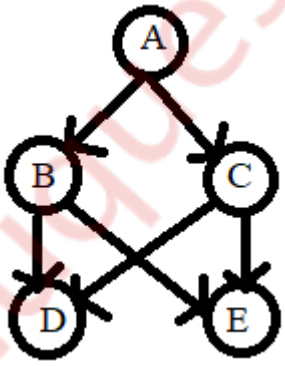


- **Construction of Expression Tree:**
  Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.
  1) If character is operand push that into stack
  2) If character is operator pop two values from stack make them its child and push current node again.
  At the end only element of stack will be root of expression tree
- **Algorithm For Expression tree**
  Let t be the expression tree
  If t is not null then
  If t.value is operand then
  Return t.value
  A = solve(t.left)
  B =solve(t.right)
  // calculate applies operate 't.value'
  // on A and B, and returns value
  Return calculate (A, B, t.value)

(d) Topological Sorting

- Topological sort of a directed acyclic graph (DAG) G is defined as a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.
- A topological sort of a DAG G is an ordering of the vertices of G such that if G contains an edge (u, v), then u appears before v in the ordering
- Note that topological sort is possible only on directed acyclic graphs that do not have any cycles.

- For a DAG that contains cycles, no linear ordering of its vertices is possible.
- In simple words, a topological ordering of a DAG G is an ordering of its vertices such that any directed path in G traverses the vertices in increasing order.
- Topological sorting is widely used in scheduling applications, jobs, or tasks. The jobs that have to be completed are represented by nodes, and there is an edge from node u to v if job u must be completed before job v can be started.
- A topological sort of such a graph gives an order in which the given jobs must be performed.
- The two main steps involved in the topological sort algorithm include:
  1.Selecting a node with zero in-degree
  2.Deleting N from the graph along with its edges
- **Algorithm For topological Sorting**
  Step 1: Find the in-degree INDEG(N) of every node in the graph
  Step 2: Enqueue all the nodes withazero in-degree
  Step 3: Repeat Steps4and5until the QUEUE is empty
  Step 4: Remove the front nodeNof the QUEUE by setting FRONT=FRONT+1
  Step 5: Repeat for each neighbourMof node N: a) Delete the edge from N to M by setting INDEG(M)=INDEG(M)-1 b) IF INDEG(M) =,then Enqueue M, that is, addMto the rear of the queue [END OF INNER LOOP] [END OF LOOP]
  Step 6: Exit
- Example:



Topological sort can be given as:

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