THIRD SEMESTER EXAMINATION 2009-10

DATA STRUCTURES USING 'C'

Time : 3 Hours *Total Marks : 100*

Note: Attempt all questions.

Q.1. (a) Explain the different ways of analysing algorithm.

Ans. To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted list to which we apply binary search has n elements, and we can guarantee that each lookup of an element in the list can be done in unit time, then at most $\log_2 n + 1$ time units are needed to return an answer.

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as one step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.

Q.1.(*b*) Write an efficient algorithm to find the *k*th element in a sequence of *n* elements.

Ans. Traversal of 1D array

```
for(i=0;i<=n;i++)
if(i==k)
printf("\n %d",a[i]);
```

Q.1.(c) Write an algorithm which obtains the transpose of $n \times n$ sequence matrix onto itself.

Ans. Algorithm for transpose of matrix

```
for(i=0;i<n;i++)
for(j=0;j<n;j++)
t=a[i][j];
a[i][j]=a[j][i];
a[j][i]=t;
```

Q.1.(d) Write the transvering algorithm for a linear array.

Ans. for(i=0;i<n;i++)
printf("\n %d",a[i]);

Algorithm:

1. Repeat step 1 for i=1 to n
2. print a[i].

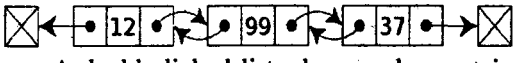
Q.1.(e) Write an algorithm and a C function to reverse a single linked list.

Ans. Reverse a Linked List

```
void reverse(struct node **kbr)
{
    struct node *temp,*p,*q;
    temp=*kbr;
    p=temp->next;
    q=temp;
    while(p!=NULL)
    {
        temp=p;
        p=p->next;
        temp->next=q;
        q=temp;
    }
    (*kbr)->next=NULL;
    *kbr=q;
}
```

Q.1.(f) What is double linked list? What are the advantage and disadvantage of double linked list?

Ans. Doubly Linked List : In computer science, a doubly-linked list is a linked data structure that consists of a set of data records, each having two special *link* fields that contain references to the previous and to the next record in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders.



A doubly-linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

The two links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is sometimes simpler because there is no need to keep track of the address of the previous node.

Advantages: Two pointers forward and backward

Disadvantage:

- Require extra memory
- Require more time in insertion and deletion

Q.2. Attempt any four parts: (5×4=20)

Q.2.(a) Write deletion algorithm for a steak. What is its complexity?

Ans. Algorithm For POP operation in stack

Pop(s):

1. if stack is empty then report underflow
2. item=stack[top];
3. top=top – 1;
4. end

Complexity is O(1).

Q.2.(b) Write an efficient algorithm which converts in-fix expressions into post fix expression.

Ans. Infix to postfix conversion

- Scan the Infix string from left to right.
- Initialise an empty stack.
- If the scanned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty, push the character to stack.
    - If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

Repeat this step till all the characters are scanned.

- (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.

Q.2.(d) Write a C program to implement a queue using linked list.

Ans. /* Program that implements queue as a linked list. */

```
#include <stdio.h>
#include <conio.h>
struct node
{
    int data;
    struct node *link;
};
struct queue
{
    struct node *front;
    struct node *rear;
};

void initqueue ( struct queue * );
void addq ( struct queue *, int );
int delq ( struct queue * );
void delqueue ( struct queue * );
void main( )
{
    struct queue a;
    int i;
    clrscr( );
    initqueue ( &a );
    addq ( &a, 11 );
    addq ( &a, -8 );
    addq ( &a, 23 );
    addq ( &a, 19 );
    addq ( &a, 15 );
    addq ( &a, 16 );
    addq ( &a, 28 );
    i = delq ( &a );
    printf ( "\nItem extracted: %d", i );
    i = delq ( &a );
    printf ( "\nItem extracted: %d", i );
    i = delq ( &a );
    printf ( "\nItem extracted: %d", i );
    delqueue ( &a );
    getch( );
}

/* initialises data member */
void initqueue ( struct queue *q )
{
    q -> front = q -> rear = NULL;
}
/* adds an element to the queue */
void addq ( struct queue *q, int item )
{
    struct node *temp;
    temp = ( struct node * ) malloc (
sizeof ( struct node ) );
    if ( temp == NULL )
        printf ( "\nQueue is full." );
    temp -> data = item;
    temp -> link = NULL;
    if ( q -> front == NULL )
    {
        q -> rear = q -> front = temp;
        return;
    }

    q -> rear -> link = temp;
    q -> rear = q -> rear -> link;
}

/* removes an element from the queue */
int delq ( struct queue *q )
{
    struct node *temp;
    int item;

    if ( q -> front == NULL )
    {
        printf ( "\nQueue is empty." );
        return NULL;
    }

    item = q -> front -> data;
    temp = q -> front;
    q -> front = q -> front -> link;
    free ( temp );
    return item;
}

/* deallocates memory */
void delqueue ( struct queue *q )
{
    struct node *temp;

    if ( q -> front == NULL )
        return;

    while ( q -> front != NULL )
    {
        temp = q -> front;
        q -> front = q -> front -> link;
        free ( temp );
    }}
```

Q.2.(e) Give short notes one:

(i) Dequeue

(ii) Priority Queues.

Ans. In computer science, a double-ended queue (often abbreviated to deque, pronounced *deck*) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Distinctions and sub-types: This differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

- An input-restricted deque is one where deletion can be made from both ends, but input can only be made at one end.
- An output-restricted deque is one where input can be made at both ends, but output can be made from one end only.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques, and can be implemented using deques.

Priority queue

Priority queue is an abstract data type in computer programming that supports the following three operations:

InsertWithPriority: Add an element to the queue with an associated priority

- **GetNext:** Remove the element from the queue that has the *highest priority*, and return it (also known as "PopElement(Off)", or "GetMinimum")
- **PeekAtNext (optional):** Look at the element with *highest priority* without removing it.

Effect of different data structures

The designer of the priority queue should take into account what sort of access pattern the priority queue will be subject to, and what computational resources are most important to the designer. The designer can then use various specialized types of heaps.

There are a number of specialized heap data structures that either supply additional operations or outperform the above approaches. The binary heap uses O(log *n*) time for both operations, but allows peeking at the element of highest priority without removing it in constant time. Binomial heaps add several more operations, but require O(log *n*) time for peeking. Fibonacci heaps can insert elements, peek at the maximum priority element, and increase an element's priority in amortized constant time (deletions are still O(log *n*)).

While relying on a heap is a common way to implement priority queues, for integer data faster implementations exist (this can even apply to datatypes that have finite range, such as floats):

- When the set of keys is {1, 2, ..., C}, a data structure by van Emde Boas supports the *minimum*, *maximum*, *insert*, *delete*, *search*, *extract-min*, *extract-max*, *predecessor* and *successor* operations in $O(\log\log C)$ time, but has a space cost for small queues of about $O(2^{m/2})$, where m is the number of bits in the priority value.

An algorithm by Fredman and Willard implements the *minimum* operation in O(1) time and *insert* and *extract-min* operations in $O(\sqrt{\log n})_{\text{time}}$.

For applications that do many "peek" operations for every "extract-min" operation, the time complexity for peek can be reduced to O(1) in all tree and heap implementations by caching the highest priority element after every insertion and removal. (For insertion this adds at most constant cost, since the newly inserted element need only be compared to the previously cached minimum element. For deletion, this at most adds an additional "peek" cost, which is nearly always cheaper than the deletion cost, so overall time complexity is not affected by this change.

Q.2.(f) What is recurrsione? Write a C program to solve Tower of Hanoi problem.

Ans. Recursion is a technique in which a function calls itself.

In this there exists a base criteria. Also in every step call should be closer to the base criteria.

Program for tower of Hanoi:

```
#include<stdio.h>
#include<conio.h>
Void hanoi(int n, char beg,char aux,
                                char end)
{ if(n==1)
  Printf("\n move disk from %c to
                              %c,beg,end);
else
hanoi(n-1;beg,end ,aux);
hanoi(1,beg,aux,end);
hanoi(n-1,aux,beg,end);
}
void main()
{int n;
char a,b,c;
printf("\n enter no of disks");
scanf("%d",&n);
hanoi(n,a,b,c);
getch();
}
```

Q.3. Attempt any two parts: (2×10=20)

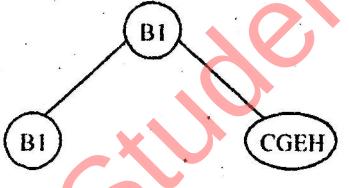
(a)(i) If the in-order traversal of a binary tree is B, I, D, A, C, G, E, H, F and its port-order traversal is I, D, B, G, C, H, F, F, A. Determine the binary tree.

Ans. Given In order : B,I,D,A,C,G,E,H,F

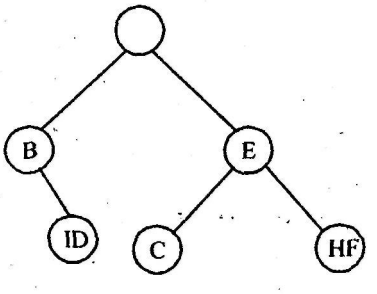
Post order: I,D,B,G,C,H,F,E,A

Creation of binary tree:

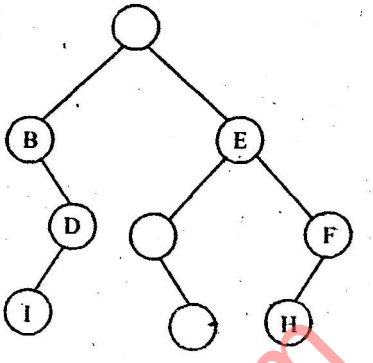
Step1



Step 2



Step3

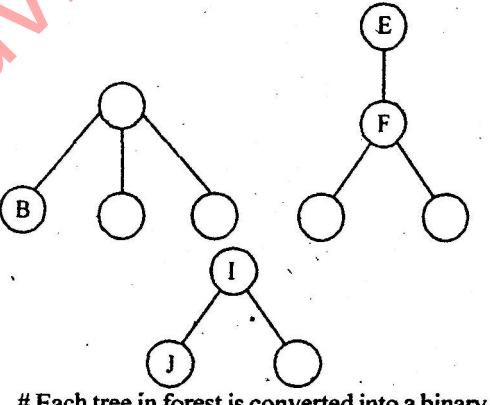


Final binary tree

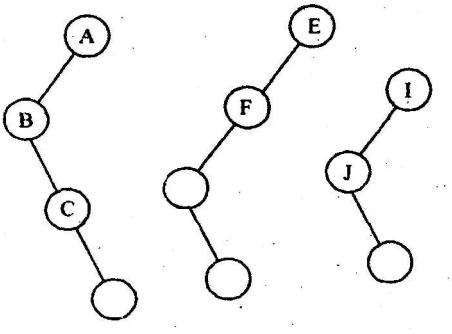
(a) (ii) Write an algorithm to convert a forest in to a binary tree.

Ans. **Forest :** Forest is defined as a set of trees . it should be clear that right child of equivalent binary tree will always be null. A root does not have a sibling.

When a forest is transformed into a binary tree, root will have a right child. Right child of the root will be next tree in a forest eg: forest with 3 trees.



Each tree in forest is converted into a binary tree using leftmost child right sibling relation

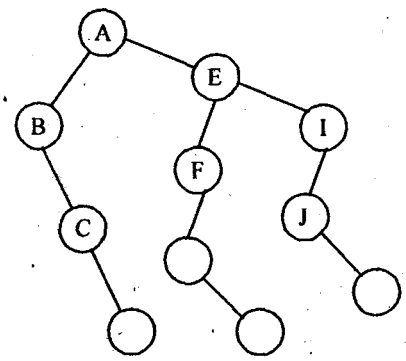


Each tree of forest is represented by its corresponding binary tree

These three trees can be combined by

1. tree with root node E is node the right child of node A

2. Tree with root node I is the node right child of node E



(b) What is a binary search tree? Write a C program to insert new notes to a binary search tree and delete a given node from a binary search tree.

Ans. Binary search tree: Binary Search tree is a binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. This is called binary-search-tree property.

/*Insertion and Deletion in Binary Search Tree*/

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
struct node
{
int info;
struct node *lchild;
struct node *rchild;
}*root;

main()
{
int choice,num;
root=NULL;
while(1)
{
printf("\n");
printf("1.Insert\n");
printf("2.Delete\n");
printf("6.Display\n");
printf("7.Quit\n");
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Enter the number to be inserted : ");
scanf("%d",&num);
insert(num);
break;
case 2:
printf("Enter the number to be deleted : ");
scanf("%d",&num);
del(num);
break;
case 6:
display(root,1);
break;
case 7:
exit();
default:
printf("Wrong choice\n");
}/*End of switch */
}/*End of while */
}/*End of main()*/

find(int item,struct node **par,struct node
**loc)
{
struct node *ptr,*ptrsave;
if(root==NULL) /*tree empty*/
{
*loc=NULL;
*par=NULL;
```

```
return;
}
if(item==root->info) /*item is at root*/
{ .
*loc=root;
*par=NULL;
return;
}
/*Initialize ptr and ptrsave*/
if(iteminfo)
ptr=root->lchild;
else
ptr=root->rchild;
ptrsave=root;

while(ptr!=NULL)
{
if(item==ptr->info)
{ *loc=ptr;
*par=ptrsave;
return;
}
ptrsave=ptr;
if(iteminfo)
ptr=ptr->lchild;
else
ptr=ptr->rchild;
}/*End of while */
*loc=NULL; /*item not found*/
*par=ptrsave;
}/*End of find()*/

insert(int item)
{ struct node *tmp,*parent,*location;
find(item,&parent,&location);
if(location!=NULL)
{
printf("Item already present");
return;
}
tmp=(struct node *)malloc(sizeof(struct
node));
tmp->info=item;
tmp->lchild=NULL;
tmp->rchild=NULL;

if(parent==NULL)
root=tmp;
else
if(iteminfo)
parent->lchild=tmp;
else
parent->rchild=tmp;
}/*End of insert()*/

del(int item)
{
struct node *parent,*location;
if(root==NULL)
{
printf("Tree empty");
return;
}

find(item,&parent,&location);
if(location==NULL)
{
printf("Item not present in tree");
return;
}
if(location->lchild==NULL && location-
>rchild==NULL)
case_a(parent,location);
if(location->lchild!=NULL && location-
>rchild==NULL)
case_b(parent,location);
if(location->lchild==NULL && location-
>rchild!=NULL)
case_b(parent,location);
if(location->lchild!=NULL && location-
>rchild!=NULL)
```

```
case_c(parent,location);
free(location);
}/*End of del()*/
case_a(struct node *par,struct node *loc )
{
if(par==NULL) /*item to be deleted is root
node*/
root=NULL;
else
if(loc==par->lchild)
par->lchild=NULL;
else
par->rchild=NULL;
}/*End of case_a()*/
case_b(struct node *par,struct node *loc)
{
struct node *child;
/*Initialize child*/
if(loc->lchild!=NULL) /*item to be deleted
has lchild */
child=loc->lchild;
else /*item to be deleted has rchild */
child=loc->rchild;
if(par==NULL ) /*Item to be deleted is root
node*/
root=child;
else
if( loc==par->lchild) /*item is lchild of its
parent*/
par->lchild=child;
else /*item is rchild of its parent*/
par->rchild=child;
}/*End of case_b()*/
case_c(struct node *par,struct node *loc)
{
struct node *ptr,*ptrsave,*suc,*parsuc;
/*Find inorder successor and its parent*/
ptrsave=loc;
ptr=loc->rchild;
while(ptr->lchild!=NULL)
{
ptrsave=ptr;
ptr=ptr->lchild;
}
suc=ptr;
parsuc=ptrsave;
if(suc->lchild==NULL && suc-
>rchild==NULL)
case_a(parsuc,suc);
else
case_b(parsuc,suc);
if(par==NULL) /*if item to be deleted is root
node */
root=suc;
else
if(loc==par->lchild)
par->lchild=suc;
else
par->rchild=suc;
suc->lchild=loc->lchild;
suc->rchild=loc->rchild;
}/*End of case_c()*/

display(struct node *ptr,int level)
{
int i;
if ( ptr!=NULL )
{
display(ptr->rchild, level+1);
printf("\n");
for (i = 0; i < level; i++)
printf(" ");
printf("%d", ptr->info);
display(ptr->lchild, level+1);
}/*End of if*/
}/*End of display()*/
```

(c) Write short notes one:

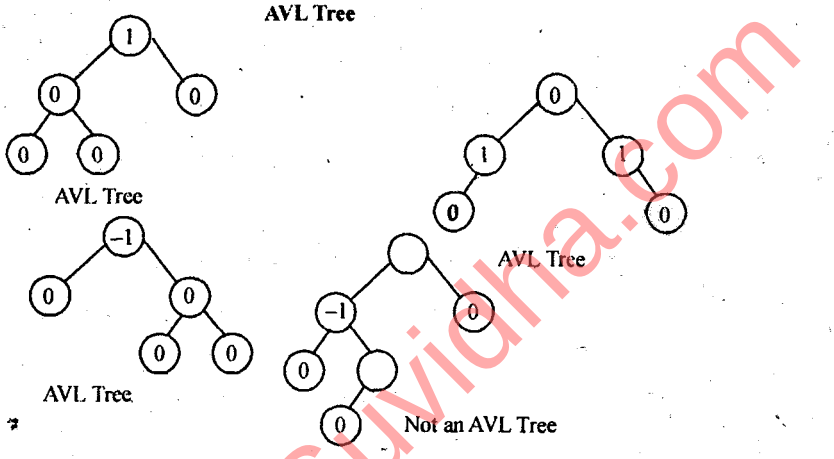
(i) Height balance tree

(ii) Thread binary tree

Ans. (i) Height Balance Tree A height balance tree is also known as AVL tree

* **AVL tree definition:** a binary search tree in which the maximum difference in the height of any node's right and left sub-trees is 1 (called the balance factor) balance factor = height(left) – height(right)
* AVL trees are usually not perfectly balanced
  however, the biggest difference in any two branch lengths will be no more than one level
* The balance factor will be 1 for left high,
  – 1 for right high
  0 for balanced node
* So AVL tree each node have three values of balance factor which are -1, 0, 1.

We can say that the absolute value of balance factor should be less than or equal to 1. Balance factor = $H(T_L) - H(T_R)$.

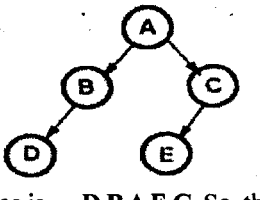


(ii) Threaded binary tree A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

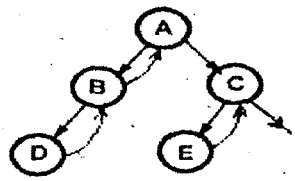
The node structure for a threaded binary tree varies a bit and its like this —

```
struct NODE
{
 struct NODE *leftchild;
 int node_value;
 struct NODE *rightchild;
 struct NODE *thread;
}
```

Let's make the Threaded Binary tree out of a normal binary tree



INORDER traversal for the above tree is — **D B A E C**. So, the respective Threaded Binary tree will be



B has no right child and its inorder successor is A and so a thread has been made in between them. Similarly, for D and E. C has no right child but it has no inorder successor even, so it has a hanging thread.

Q.4. Attempt any TWO parts: **(2×10=20)**

(a) (i) Obtain the minimum number of entries that can be made in a B-tree of order m and of levels *l*.

***(ii)* Use merge - sort algorithm to sort the following elements 15, 10, 5, 20, 25, 30, 40, 35.**

Ans. (a) (i) Minimum no of entries that can be made in tree of order m 7 and level l

No of nodes at level 0=1
No of nodes of level 1=2
Minimum No of nodes at level 2=2 * ((m+1)/2)
Minimum No of nodes at level 3=2 * $((m+1)/2)^2$
Minimum No of nodes at level 4=2 * $((m+1)/2)^3$

Minimum No of nodes at level l=2 *$((m+1)/2)^{l-1}$
Thus Minimum No of nodes in B –tree at level l =
1+2+2 * ((m+1)/2)+ 2 * $((m+1)/2)^2$ +————————————————+2 *$((m+1)/2)^{l-1}$
=1+2[1+(m+1)/2+ $(m+1)/2)^2$ +————————————————+$(m+1)/2)^{l-1}$]
=1+4* $(1-(m+1)/2)^l)/(1-m)$
Since each intermediate node should have (m-1)/2 keys
So minimum entries = 1+4* $(1-(m+1)/2)^l)/(1-m)$ *(m-1)/2 = 1+2[$((m+1)/2))^l$ – 1]

Ans. (ii) Merge sort : 15 10 5 20 25 30 40 35
Step 1 :- 15 10 5 20 25 30 40 35
10 15 5 20 25 30 35 40
Step 2 :- 10 15 5 20 25 30 35 40
5 10 15 20 25 30 35 40
Step 3 :- 5 10 15 20 25 30 35 40
5 10 15 20 25 30 35 40

(*b*) How can you find shortest path between two nodes in a graph by Dijkstra's algorithm? Explain by suitable diagram and algorithm.

Ans. Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s):

1. INITIALIZE SINGLE-SOURCE (G, s)

2. S ? { } // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ? V[G]
4. while priority queue Q is not empty do
5. *u* ? EXTRACT_MIN(Q) // Pull out new vertex
6. S ? S ∪ {*u*}
   // Perform relaxation for each vertex *v* adjacent to u
7. for each vertex v in Adj[u] do
8. Relax (*u*, *v*, *w*)

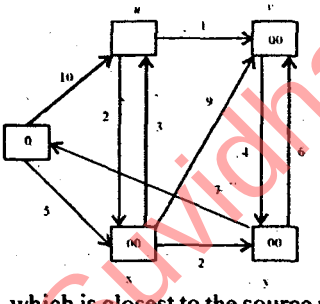
The relaxation procedure checks whether the current best estimate of the shortest distance to v (**d[v]**) can be improved by going through **u** (*i.e.* by making **u** the predecessor of **v**):

```
relax( Node u, Node v, double w[][] )
 if d[v] > d[u] + w[u,v] then
 d[v] := d[u] + w[u,v]
 pi[v] := u
```

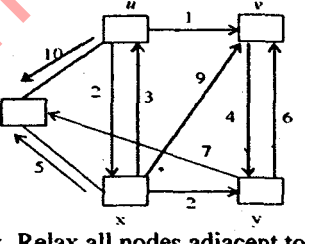
Analysis: Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.

Example: Step by Step operation of Dijkstra algorithm.

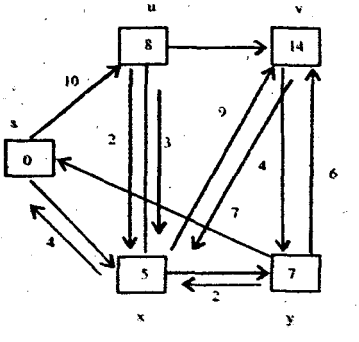
Step1. Given initial graph G=(V, E). All nodes nodes have infinite cost except the source node, s, which has 0 cost.



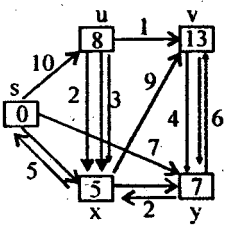
Step 2. First we choose the node, which is closest to the source node, s. We initialize d[s] to 0. Add it to S. Relax all nodes adjacent to source, s. Update predecessor (see red arrow in diagram below) for all nodes updated.



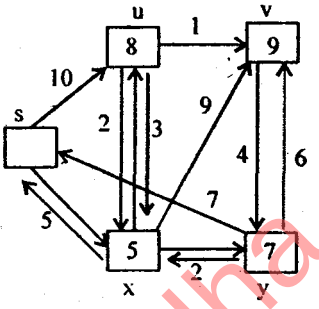
Step 3. Choose the closest node, x. Relax all nodes adjacent to node x. Update predecessors for nodes u, v and y (again notice red arrows in diagram below).



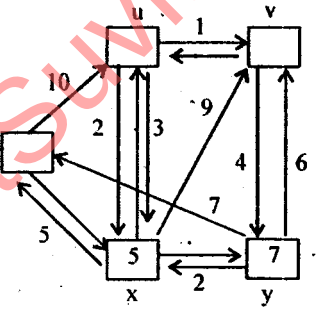
Step 4. Now, node y is the closest node, so add it to S. Relax node v and adjust its predecessor (red arrows remember!).



Step 5. Now we have node u that is closest. Choose this node and adjust its neighbor node v.



Step 6. Finally, add node v. The predecessor list now defines the shortest path from each node to the source node, s.



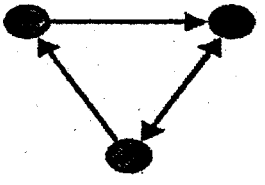
(c) What is a graph? Differentiate between (i) undirected and directed graph (ii) Cycle and Hamiltonian cycle.

Ans. Graph: A vertex (node) is a stand-alone object. Represented by a circle.

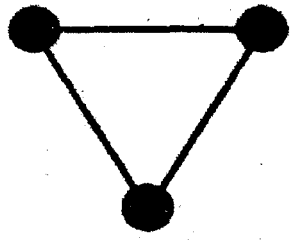
An edge (link) is an object connecting two vertices. Represented by either an arrow or a line.

Directed Graphs: A *directed graph* (or *digraph*) is a graph with *directed edges*.

Edges have directions so they are represented by arrows.Each edge *leaves* a vertex and *enters* a vertex.



Undirected graphs : An undirected graph is a graph with undirected edges. Edges have no directions so they are represented by lines.Self-loops are forbidden. Edge (u,v) is the same as edge (v,u).

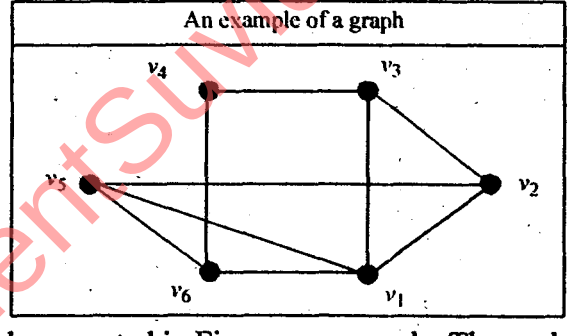


The number of edges with one endpoint on a given vertex is called that vertex's **degree**. In a directed graph, the number of edges that point *to* a given vertex is called its **in-degree**, and the number that point *from* it is called its **out-degree**. In an undirected graph, The out-degree and the in-degree are not defined. Only the degree of a vertex is defined.

Cycle and Hamiltonian cycle:

Cycle: A path is known as closed path cycle if $V_0=V_n$ otherwise if all the nodes are distinct then path is known as simple path.

Hamiltonian cycle: Hamiltonian cycle is closed path/ walk in which each vertex is traversed only once except for the starting vertex , as the walk terminates at the starting vertex



Consider the graph presented in Fig. as an example. The cycle $v_1v_6v_4v_3v_2v_5v_1$ is a Hamiltonian cycle. Of course, there are many other cycles that are not Hamiltonian, for example, $v_1v_6v_5v_1$ or the loop $v_2v_5v_1v_2v_5v_1v_2$.

Q.5 Attempt any two questions: **(2×10=20)**

(a) Write down the algorithm for bubble sort and explain how you can sort an unsorted array of integers by using quick-sort. Find out the time complexity of your algorithm.

Ans. Bubble sort Algorithm:

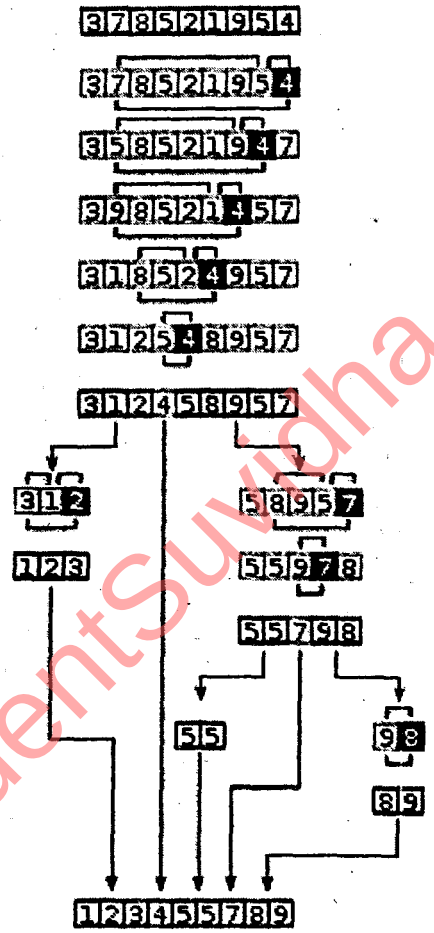
```
for i = 0 to n
 for j =0 to n-i
      ifA[j]<A[j+1]
        swap(A,i,j)
```

The QUICKSORT Example: Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.

Recursively sort the sub-list of lesser elements and the sub-list of greater elements. The base case of the recursion are lists of size zero or one, which are always sorted.



Full example of quicksort on a random set of numbers. The boxed element is the pivot. It is always chosen as the last element of the partition.

Complexity:-

	Worst	Average	Best
Bubble sort:	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick sort:	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$

(b) Define hash function. State different types of hash function. Give their algorithm and explain them by suitable diagram.

Ans. Hash Function: The basic idea in hashing is the transformation of key into the corresponding location in the hash table. This is done by hash function.

A function can be defined as a function that take key as input and transforms it into a hash index.

It is usually denoted by H

$H:K \rightarrow M$

Where H is a hash function

K is a set of keys

M is a set of memory addresses

Sometimes, such a function H may not generate distinct valuesIt is possible that two different keys K_1 and K_2 will yield the same hash address. This situation is called Hash Collision.

Different Hash Functions:

- Division Remainder Method
- Mid Square Method
- Folding Method
- Multiplication method

Division Remainder Method: In Division Remainder Method, key k to be mapped into of the m slots in the hash table is divided by m and the remainder of this division is taken as index into the hash table.

$h(k) = k \bmod m$ *range 0 to m-1*

$h(k) = k \bmod m + 1$ *range 1 to m*

Example: Consider a hash table with 9 slots i.e m=9, then the has function will map the key 132 to slot 6.

$h(132) = 132 \bmod 9 = 6$

Means address of data or key 132 will store in 6th position of Hash table

Mid Square Method: It is operates in two steps:

The square of the key value k is taken.

In second step, the hash value is obtained by deleting digits from ends of the squared value i.e. k^2. it is important to note that same position of k^2 must be used for all keys. Thus, the hash function is $h(k) = s$

Where s is obtained by deleting digits from both sides of k^2.

Consider the hash table with 100 slots i.e. m =100, and key values k=3205, 7148, 2345

K	3205	7148	2345
k^2	10272025	51093904	5499025
h(k)	72	93	90

Folding Method: It is also operates in two steps:

In the first step, the key value k is divided into number of parts, k1,k2,k3……..kr, where each part has same number of digits except the last part, which can have lesser digits.

In second step, these parts are added together and the hash value is obtained by ignoring the last carry, if any.

For example, if the hash table has 1000 slots, each part will have three digits, and the sum of these parts after ignoring the last carry will also be three digit number in the range 0 to 999.

Multiplication method: The multiplication method operates in two steps:

The key value k is multiplied by a constant A in the range $0<A<1$ and extract the fractional part of the value kA.

The fractional value obtained above is multiplied by m and the floor of the result is taken as the hash value. That is Hash Function is $h(k) = m(kA \bmod 1)$

Although this method works with any value of A. knuth has suggested in his study that the following value of A is likely to work reasonable well. $A = (\sqrt{5} - 1)/2 = 0.6180339887\ldots$

Consider a hash table with 10000 slots i.e. m= 10000, will map the key 123456 to slot 41 since

$$h(123456) = 10000 \times (123456 \times 0.61803\ldots \bmod 1) = 41.151 = 41$$

(c) Write shorts notes on:

(i) B+ Tree

(ii) Garbage collection.

Ans. (i) B+ Tree: The **B-tree** is the classic disk-based data structure for indexing records based on an ordered key set. The **B⁺-tree** (sometimes written B+-tree, B+tree, or just B-tree) is a variant of the original B-tree in which all records are stored in the leaves and all leaves are linked sequentially. The B+-tree is used as a (dynamic) indexing method in relational database management systems. B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned without visiting the higher nodes at all.

A B + -Tree consists of one or more blocks of data, called *nodes*, linked together by pointers. The B + -Tree is a tree structure. The tree has a single node at the top, called the *root node*. The root node points to two or more blocks , called *child nodes*. Each child nodes points to further child nodes and so on.

- The B + -Tree consists of two types of (1) *internal nodes* and (2) *leaf nodes*:
- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data in the database using *data pointers*.

Order of a B + -Tree: The *order* of a B + -Tree is the number of keys and pointers that an internal node can contain. An order size of *m* means that an internal node can contain *m-1* keys and *m* pointers.

(ii) Garbage Collection: In computer science, **garbage collection (GC)** is a form of automatic memory management. It is a special case of *resource management*, in which the limited resource being managed is memory. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program.

Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of the two approaches, and other techniques such as stack allocation and region inference can carve off parts of the problem.

Garbage collection does not traditionally manage limited resources other than memory that typical programs use, such as network sockets, database handles, user interaction windows, and file and device descriptors. Methods used to manage such resources, particularly destructors, may suffice as well to manage memory, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources.