S.Y. Diploma : Sem. III
[CO/CM/IF/CD/CW]

Data Structure using 'C'

Time: 3 Hrs.] Prelim Question Paper Solution **[Marks : 100**

Q.1(a) Attempt any SIX of the following : [12]

Q.1(a) (i) Define time complexity and Space complexity. [2]

(A) Time complexity

Time complexity of a program/algorithm is the amount of computer time that it needs to run to completion (execution).

Space complexity

Space complexity of a program/algorithm is the amount of memory that it needs to run to completion (execution).

Q.1(a) (ii) Define Push and Pop operations of stack. [2]

(A) **PUSH:** Push operation is used to insert an element at TOP of stack.

POP: Pop operation is used to remove an element from TOP of stack.

Q.1(a) (iii) Define Hashing. [2]

(A) Hashing is a technique used to compute memory address for performing insertion, deletion and searching of an element using hash function.

Q.1(a) (iv) List various sorting techniques. [2]

(A) Sorting Techniques

- Bubble sort
- Insertion sort
- Merge sort
- Shell sort
- Selection sort
- Quick sort
- Radix sort

Q.1(a) (v) Define Complete binary tree. [2]

(A) Complete Binary Tree

A binary tree in which every node has 2 children except the leaf nodes at level 'd' where 'd' is the depth of the tree.

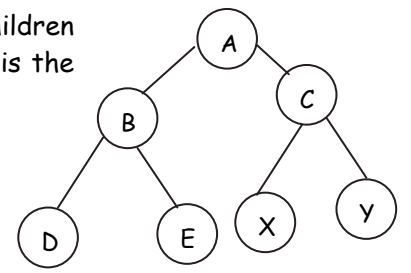
Hence at any level n, No of nodes = 2^n

For a complete binary tree of depth 'd',

Number of leaf nodes = 2^d

Number of non-leaf nodes = $2^d - 1$

Total number of nodes = $2^{d+1} - 1$

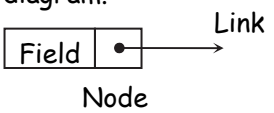


Q.1(a) (vi) Define Node and Pointers. **[2]**

(A) Unlike a sequential representation where successive items of list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. Another way of saying this is that when we want access elements in the list in the list in correct order, with each element we store the address of next element in that list. Thus, associated with each element there is a pointer to the next item. This pointer is often referred to as a link.

In general node is a collection of data, and link. Each item in a node is called as field. A field either contains a data or link.

Consider the following diagram.



Q.1(a) (vii) Define Data Structure. **[2]**

(A) A data stricture is a specialized format for organizing and storing data.

Q.1(a) (viii) List operations on trees. **[2]**

(A) **Operations on trees**

1) creation
2) traversal
3) deletion
4) insertion
5) compare
6) merge

Q.1(b) Attempt any TWO of the following : **[8]**

Q.1(b) (i) Explain different approaches to design an algorithm. **[4]**

(A) **(1) Top-Down Approach:** A top–down approach starts with identifying major components of system or program decomposing them into their lower level components & iterating unit desired level of module complexity is achieved. In this we start topmost module & incrementally and modules that is calls.

(2) Bottom-Up Approach: A bottom-up approach starts with designing most basic or primitive component & proceeds to higher level components. Starting from very bottom, operations that provide layer of abstraction are implemented.

Q.1(b) (ii) Explain circular queue with example. [4]

(A) Circular Queue

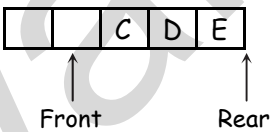
Circular queue are the queues implemented in circular form rather than in a straight line. Circular queues overcome the problem of unutilized space in linear queue implemented as an array. The main disadvantage of linear queue using array is that when elements are deleted from the queue, new elements cannot be added in their place in the queue, i.e. the position cannot be reused.

After rear reaches the last position, i.e. MAX-1 in order to reuse the vacant positions, we can bring rear back to the 0^{th} position, if it is empty, and continue incrementing rear in same manner as earlier. Thus rear will have to be incremented circularly. For deletion, front will also have to be incremented circularly.

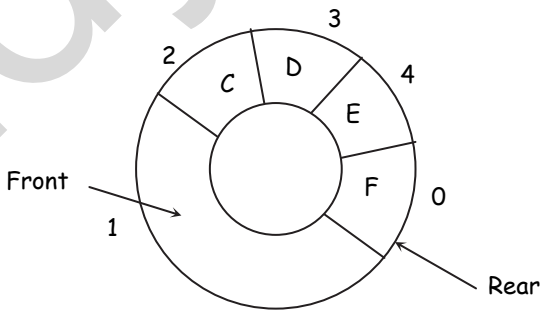
Rear can be incremented circularly by the following code.

```
If ((rear == MAX-1) and (front !=0)
        Rear =0;
Else
        Rear= rear +1;
```

Example: Assuming that the queue contains three elements.



Now we insert an element F at the beginning by bringing rear to the first position in the queue. This can be represented circularly as shown.



In the above example, if another element, G is added to the queue, i.e. rear and front coincide. But rear and front coincide even when the queue is full is

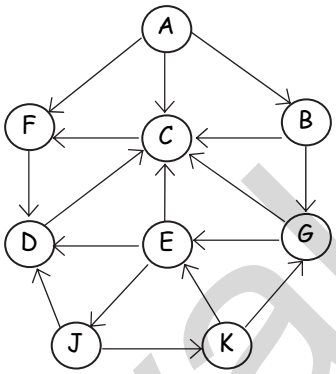
empty. Thus rear and front cannot be used for both i.e. to check for empty queue as well as the condition for a full queue.

The rear ==front condition is used to check for the empty queue since initially both are initialized to the same value. Thus to check for queue full condition there are three methods.

1) Use a counter to keep track of the number of elements in the queue. If this counter reaches to MAX the queue is full.
2) After remove operation rear = front, then set to -1. After add operation if rear = front then will say that queue is full.
3) By checking (rear + 1) % MAX== front.

Q.1(b) (iii) Describe Depth First Search. **[4]**

(A) DFS (Depth First Search)

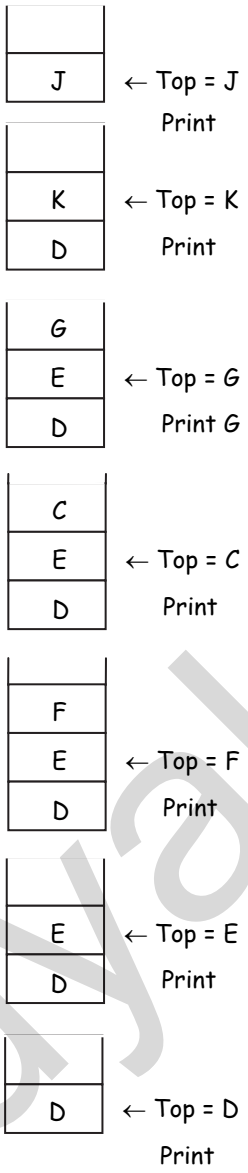


Traverse a graph till node 'J' is found using DFS.

Step 1:

edge	Adjacancy list
A	F, C, B
B	C, G
C	F
D	C
E	D, C, J
F	D
G	C, E
J	D, K
K	E, G

Step 2:



Step 3 : Hence, J – K – G – C – F – E – D are the precisely nodes which are reachable from J.

Q.2 Attempt any FOUR of the following : [16]

Q.2(a) Describe Big 'O' notation used in algorithm. [4]

(A) Asymptotic Comparison

For smaller values of n, the constants play significant role in the complexity values. Those constants are the byproduct of the assumptions done and are

required to be overcome. As the value of n increases, the effect of constants gets vanished. We can see functions fluctuating for the smaller values of n, attains a proper rate as the value of n is increased. Hence the comparison of the algorithms is done for the higher values of inputs, such comparison is known as Asymptotic Comparison. The functions used for the asymptotic comparison are known as asymptotic growth functions. Following are some of the asymptotic growth functions used in algorithm analysis.

1) Big O (Upper Bounding) Function
2) Big (Lower Bounding) Function
3) Theta (Order) Function

Asymptotic Growth Functions

Big O (Upper Bounding) Function

A function g(n) = O (f(n)) if and only if, there are 2 constants c and n_0 (both > 0) such that, $0 \leq g(n)$ c . f(n) for all $n < n_0$.

Suppose, $g(n) = 502n^2 + 3n - 7$

$f(n) = 715n^2 + 100n + 10$

Here by making C = 1 and $n_0 = 1$ we get, $0 \leq g(n) \leq c$. f (n). Hence g(n) = O (f (n)) and, it is said that g(n) is upper bounded by f(n). If $g(n) = 502n^2 + 3n - 7$ and $f(n) = n^3 + 200n^2 - 5$. Now we want to prove g(n) = O(f(n)) then C = 1 and $n_0 = 502$. If $f(n) = 0.3n^3 + 200n^2 - 5$ then we can say $f(n) = O(n^3)$. Hence the highest power term matters. Now suppose $g(n) = 502n^2 + 3n - 7$ and $f(n) = n^2 - 7n - 15$. Then we can show g(n) = 0(f(n)) by taking c = 200000 (say) and n = 1 i.e. $g(n) = O(n^2)$. But f(n) = 7n + 5 and $g(n) = 502n^2 + 3n - 7$. Now even if we take c = 200000, it cannot be proved that there is some n_0 such that $g(n) < c$ f(n) for all $n > n_0$. Hence $g(n) \neq O(f(n))$. The Big O function is used for the worst case complexity.

Q.2(b) Distinguish between Stack and Queue (Any 4 points). [4]

(A) [4 points each 1 mark]

	Stack	Queue
1)	In Stack insertion and deletion operations are performed at same end	In Queue insertion and deletion operations are performed at different end
2)	In stack the element which is inserted last is first to delete so it is called Last In first Out	In Queue the element which is inserted first is first to delete so it is called first In first Out
3)	In stack only one pointer is used called Top.	In queue two pointer is used called front, rear.
4)	In Stack Memory is not wasted	In Queue memory can be wasted/ unusable in case of linear queue.

5)	In computer system stack is for procedure calls	Queue is used for time/resource sharing system;
6)	Plate Counter at marriage reception is an example of stack	students standing in a line at fee counter is an example of queue

Q.2(c) Write a program for Selection sort. **[4]**

(A)

```
#include < stdio.h >
int main ()
{
        int array[100],n,i,j,temp,pos;
        printf("Enter the number of elements to be sorted: ");
        scanf("%d",&n);
        printf("enter the elements \n");
        for(i=0;i<n;i++)
{
        scanf("%d",&array[i]);
}
for(i=0;i<n;i++)
{
        pos=i;
            forj = i + 1; j < n; j++)
            {
                    if(array[j]<array[pos])
                    pos=j;
            }
        temp=array[i];
        array[i]=array[pos];
        array[pos]=temp;
}
printf("The Sorted List Is ");
for(i=0;i<n;i++)
    printf("%d ",array[i]);
 getch();
}
```

Q.2(d) Explain Linear search algorithm. **[4]**

(A) **Linear or Sequential Search**

Logic : Sequential search can be applied to any data i.e. stored in the form of array or the linked list.

- It accepts the data to be searched and starts comparing from the first record till either it reaches the record to be searched or the searched data is exhausted.

Algorithm :

Step 1: Establish an array a with n element i.e. a[n].
Step 2: Accept the value to be searched and call it as x.
Step 3: Compare the value of n with the current record and move to next record.
Step 4: Repeat steps till the match is found or there are no records left to be compared.
Step 5: Return the record if the match is found else signal failure.
Step 6: End

Limitations :

Generally if n record or n^{th} element is to be searched atleast (n – 1) comparison are required so the average performance of algorithm is given as

$f(n) = n - 1, O = (n)$

Q.2(e) Define terms: [4]

(i) Node (ii) Address (iii) Null Pointer (iv) Next Pointer

(A) **(i) Node :** A node of a tree is an item or information along with the branches to other nodes [1 mark]

(ii) Address : Address indicates location of node [1 mark]

(iii) Null pointer : A linked field of the last node contains Null rather than a valid address it indicates end of the list [1 mark]

(iv) Next Pointer for Linked list: [1 mark]
Which Hold the Address of the next node, which intern links the nodes together?

Q.2(f) Explain Priority queue with its types. [4]

(A) A priority queue is a queue in which the intrinsic ordering among the elements decides the result of its basic operations i.e. the ordering among the elements decides the manner in which Add and Delete operations will be performed.

In a priority queue,

1) Each element has a priority.
2) The elements are processed according to, higher priority element is processed before lower priority element and two elements of same priority are processed according to the order of insertion.

An example where priority queue are used is in operating systems. The operating system has to handle a large number of jobs. These jobs have to be properly scheduled. The operating system assigns priorities to each type of job. The jobs are placed in a queue and the job with the highest priority will be executed first.

Types of priority queues:

i) **Ascending priority queue:** This is a priority queue in which elements can be inserted in any order i.e. arbitrarily, but only the smallest element can be removed. This means that the element having smallest value has highest priority.

ii) **Descending priority queue:** This is priority queue in which elements can be inserted in any order i.e. arbitrarily, but only the largest element can be removed. This means that the element having largest value in the queue has highest priority.

Q.3 Attempt any FOUR of the following : [16]

Q.3(a) Explain Hash functions. [4]

(A) Hash Functions

The two principle criteria used in selecting a hash function are as follows: First, the function should be very easy and quick to compute. Second, the function should, as far as possible, uniformly divide the hash addresses throughout the available memory address range so that there are minimum number of collisions. To compute various hashing functions we assume that there is a file F of n records with a set k of keys which uniquely determine the records in F. Secondly, we assume that F is maintained in memory by a table T of m memory locations, and that L is a set of memory addresses of the locations in T.

Some of the popular hash functions are described below.

Division Method

In this method, we choose a number m (i.e., memory locations) larger than the number of n (i.e., number of records) of keys in K (i.e., keys which uniquely determine records). The number of m Is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions. The hash function **H** is defined by:

$$H(K) = K(\text{mod } m)$$

or

$$H(K) = K(\text{mod } m) + 1$$

Here K (mod m) denotes the remainder when K is divided by m. The second formula is used when we want the hash addresses to range from 1 lo m rather than from 0 lo m-1.

For example, a library has 68 books and they have been assigned 4–digit employee number. Assume L (memory addresses in the table) of 100 two digit addresses: 00, 01, 02.....,99. Applying the above hash function, say, for employee numbers:

3205, 7148, 2345

Here, we choose a prime number m dose to 99, such as 97 then —

H (3205) = 4
H (7148) = 67
H (2345) = 17

That is, dividing 3205 by 97 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, and dividing 2345 by 97 gives a remainder of 17.
In the other case where the memory addresses begin with 01 rather than 00, we choose the hash function:

H (3205) = 4 + 1 = 5
H (7148) = 67 + 1 = 68
H (2345) = 17 + 1 =18

MID Square Method

When the key **K** is squared then the hash function H is defined by

H (K) = I

where I is obtained by deleting digits from both ends of K^2 such that two digit address is left. We emphasize that the same position of K^2 must be used for all of the keys.

Considering the same example of employees as in finding the hash function through division method we get –

K	:	3205	7048	2345
K2	:	102 72 025	510 93 904	54 99 025
H(K)	=	72	93	99

In this example, the fourth and fifth digits, counting from the right, are chosen as the hash addresses.

Folding method

In this method, the key K is partitioned into number of parts, $K_1 \ldots K_r$ where each part, except possibly the last, has the same number of digits as required addresses. Then the parts are added together ignoring the last carry or most. The hash function defined as:

$H(K) = K_1 + K_2 + ...K_r$

where the leading digit carries, if any are ignored. Sometimes for extra 'milling' the even numbered parts, K_2 K_4..... are each reversed before addition.

Considering the same example used in defining the above two functions.
For example:

H (3205) = 32 + 05 = 37

H (7148) = 71+ 48 = 19

H (2345) = 23 + 45 = 68

Observe that the most significant digit 1 in H (7148) is ignored. Alternatively, one may want to reverse the second part before adding, thus, producing the following hash addresses,

H (3205) = 32 + 50 = 82

H (7148) = 71 + 34 = 55

H (2345) = 23 + 54 = 77

Q.3(b) Write a program for bubble sort [4]

(A) (Note: program can be written differently depending on the same logic. It may be without separate function) given solution is only code for separate function. Same logic from the code can be used in single program without function.)

```
bsort(int a[ ],int n)
{
inti,j,temp;
printf("\n\n BUBBLE SORT");
for(i=0;i<10;i++)
 {
for(j=0;j<10-i;j++)
 {
if (a[j]>a[j+1])
 {
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
 }
 }
 }
 }
```

Q.3(c) Convert the following expression P written in postfix notation into infix: [4]

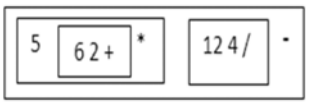
P: 5,6,2,+,*,12,4,/,-

Also evaluate P for final value.

(A) Postfix to infix form :

Postfix expression (P): 5, 6, 2, +,*, 12, 4, /,-

Step 1: Grouping of tokens as per the sequence of evaluation. k



Step 2: Operators are moved between the two operands of the group

5*(6 + 2) – 12/4

	Input	Stack	Operation
1	5, 6, 2, +, *, 12, 4, /, –	Empty	
2	6, 2, +, *, 12, 4, /, –	5	
3	2, +, *, 12, 4, /, –	5, 6	
4	+, *, 12, 4, /, –	5, 6, 2	
5	*, 12, 4, /, –	5, 8	6 + 2 = 8
6	12, 4, /, –	40	5 * 8 = 40
7	4, /, –	40, 12	
8	/, –	40, 12, 4	
9	–	40, 3	12/4 = 3
10	End	37	40–3=37

Q.3(d) Explain inorder, preorder and postorder traversal. [4]

(A) **1. Preorder traversal :** [1 mark]

In this traversal method first process root element, then left sub tree and then right sub tree.

Procedure:

Step 1: Visit root node

Step 2: Visit left sub tree in preorder

Step 3: Visit right sub tree in preorder

2. Inorder traversal : [1 mark]

In this traversal method first process left element, then root element and then the right element.

Procedure

Step 1: Visit left sub tree in inorder

Step 2: Visit root node
Step 3: Visit right sub tree in inorder

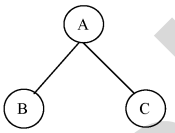
3. Postorder traversal : [1 mark]

In this traversal first visit / process left sub tree, then right sub tree and then the root element.

Procedure:

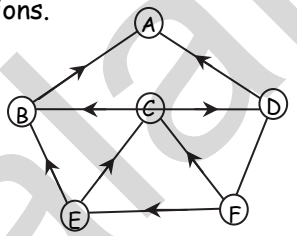
Step 1: Visit left sub tree in postorder
Step 2: Visit right sub tree in postorder
Step 3: Visit root node [1 mark]

Example : Preorder : A, B, C
Inorder : B, A, C
Postorder : B, C, A



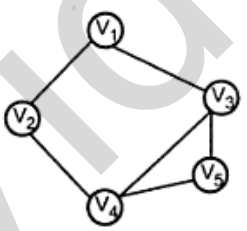
Q.3(e) Explain Graph representation with example. [4]

(A) There are several representations of graphs, but we will discuss the most commonly used representations.



Adjacency Matrix

Consider a graph G of n vertices and the matrix M. If there is an edge present between vertices V_i and V_j then M[i][j] = 1 else M[i][j] = 0. Note that for an undirected graph if M[i][j] = 1 then for M[j][i] is also 1. Here are some graphs shown by adjacency matrix.



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

Path Matrix

Let G be a simple directed graph with m nodes V_1, V_2, V_m. The path matrix or the reach ability matrix of G is the m-square matrix P = (P_{id}) denned as follows:

$$P_{ij} = \begin{Bmatrix} 1 \text{ if there is path from } V_i \text{ to } V_j \\ 0 \text{ otherwise.} \end{Bmatrix}$$

Suppose there is a path from V_i to V_j, then there must be a simple path form V_i, to V_j or there must be a cycle from V_i to V_j when $V_i = V_j$. Since G hat only m nodes, such a simple path must have length m–1 or less, or such a cycle must have length m or less. This means that there is a non-zero ij entry in the matrix B_m.

Incidence Matrix

Let 'G' be a graph with n vertices, "e" edges and no self loops. Consider a (n*e) matrix A = (aij) whose rows are equivalent to the 'n' vertices and the columns to e edges as follows:

$$aij = \begin{Bmatrix} 1 \text{ if } e_i \text{ is incident to Vertex} \\ 0 \text{ otherwise} \end{Bmatrix}$$

This kind of a matrix is known as the incidence matrix of a graph. Consider the graph given in figure 1.

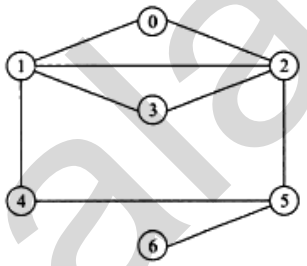


Fig. 1: Graph

The incidence matrix of above graph will be

	(0, 1)	(0, 2)	(1, 2)	(1, 3)	(1, 4)	(2, 3)	(2, 5)	(4, 5)	(5, 6)
0	1	1	0	0	0	0	0	0	0
1	–1	0	–1	1	1	0	0	0	0
2	0	–1	1	0	0	1	1	0	0
3	0	0	0	–1	0	–1	0	0	0
4	0	0	0	0	–1	0	0	1	0
5	0	–1	0	0	0	0	–1	–1	–1
6	0	0	0	0	0	0	0	0	1

The elements 'a_{ij}' of an incidence matrix of a directed graph are defined as follows:

a_{ij} = 1 if 'e_j' is Incident out of 'V_1'
= –1 if 'e_j" is incident into V_1
= 0 if 'e_j' is not incident to 'V_j'

Since every edge is incident to exactly two vertices, each column of A has two 1s for undirected graphs. The number of is in a row gives the out degree and the number of (-1) in a row gives the indegree of the tenet in a directed graph.

Q.3(f) Explain Height balanced tree with example. [4]

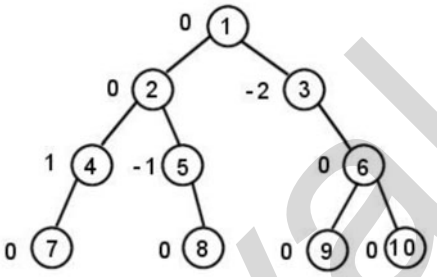
(A) Height balanced trees

AVL trees are binary search trees, which have the balance propriety. The balance property is true forany node and it states: "the height of the left subtree of any node differs from the height of the right subtree by 1".

The binary tree is balanced when all the balancing factors of all the nodes are – 1, 0 +1.

Formally, we can translate this to this: |hd – hs| ≤ 1, node X being any node in the tree, where hs andhd represent the heights of the left and the right subtrees.

Example



[2 marks]

Q.4 Attempt any FOUR of the following : [16]

Q.4(a) Define Recursion. Write 'C' program to find factorial of a number. [4]

(A) RECURSION

It is one of the design techniques where we try to replace the iterative code with the recursive one. It is the process of calling a function within itself till a terminating condition is not achieved. In many programs recursion makes the code compact. Some of the programs are basically recursive in nature so they should be solved recursively. The iterative solutions of such programs are quite tedious. Recursion though advantageous, proves inefficient with respect to time and space complexity. Calling the same function again and again leads to increase in execution time. For every next pass, the details of the previous pass are pushed on to the stack which increases the space requirements of a program. So recursion is always preferred when either there is no iterative solution for the problem or the iterative solution is tedious.

Program :

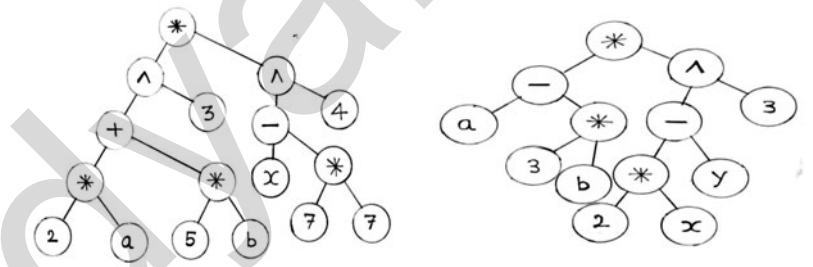
```
# icnldue <stdio.h>
int fact (int n)
{
    if (n < 0)
    printf ("Negative not allowed");
    else if (n = = 0 || n = = 1)
        return 1;
    else
        return n * fact (n – 1);
}

void main( )
{
    int a;
    printf ("Enter a number");
    scanf ("%d ", &a)
    printf ("Factorial of %d is %d ", a, fact(a));
}
```

Q.4(b) Construct expression tree for the following : [4]

(i) $(2a + 5b)^3(x - 7y)^4$ (ii) $(a-3b)(2x-y)^3$

(A)



Q.4(c) Explain Radix sort with example. [4]

(A) RADIX SORT

Logic :

- This algorithm uses the concept of queue for sorting the queue sequence.
- It generates queue for all possible digits i.e. it generates queue for digit 0 to 9.
- Starting from least significant digit the algorithm places the value inside the queue.

- This process is repeated till the data is analyzed for most significant digits.
- After analyzing for MSD (Most Significant Digit) the no. are placed in the order of appearance to the final sorted sequence.

Algorithm :

Step 1: Establish an array 'a' with 'n' element i.e. a[n]
Step 2: Let 'm' be the maximum number of digits in any of the elements.
Step 3: Initialize q from q[0] to q[9].
Step 4: Repeat step 5 and 6 m times.
Step 5: **Place** each element to **one** of the **10 q's** according to the next significant digit starting with least significant digits.
Step 6: Update array 'a' using the element from q[0] to q[9].
Step 7: End

Efficiency :

- For radix sort the amount of time required is directly depended on the no. of digits and the total no. of values in the sequence.
- If 'm' is the maximum number of digits and 'n' is the total no. of values then
  f(n) = m*n, so the efficiency is f(n) = O(m*n). The m approximates to log n so that O(m*n) approximates to O(n log n).

Q.4(d) What are the applications of Graph? [4]

(A) Application Of Graphs

Let us assume one input line containing four integers followed by any number of input lines with two integers each. The first integer on the first line, n, represents number of cities which, for simplicity, are numbered from 0 to n - 1. The second and third integers on that line are between 0 to n-1 and represent two cities. It is desired to travel from the first city to second using exactly nr roads, where nr is the fourth integer on the first input line. Each subsequent input line contains two integers representing two cities, indicating that there is a road from the first city to the second. The problem is to determine whether there is it path of required length by which one can travel from the first of the given cities to the second.

Following is the plan for solution:

Create a graph with the cities as nodes and the roads as arcs. To find the path of length nr from node A to node B, look for a node C such that an arc exists from A to C and a path of length nr - 1 exists from C to B. If these conditions are satisfied for some node C, the desired path exists, if the conditions are not satisfied for any node C, the desired path does not exist.

The traversal of a graph like Depth first traversal has many important applications such as finding the components of a graph, detecting cycles in an undirected graph, determining whether the graph is bi-connected etc.

Q.4(e) What are the characteristics of an algorithm? [4]

(A) The characteristics of a good algorithm are:

- Precision – the steps are precisely stated(defined).
- Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness – the algorithm stops after a finite number of instructions are executed.
- Input – the algorithm receives input.
- Output – the algorithm produces output.
- Generality – the algorithm applies to a set of inputs.

Q.4(f) Explain Binary search with example. [4]

(A) Binary search

Binary search can be performed only on sorted array. In this method first calculate mid position in an array and compare the mid position element with the search element. If a match is found the complete the search otherwise divide the input list into 2 parts. First part contains all the numbers less than mid elements and 2nd part contains all the numbers greater than mid element.

To calculate mid element perform (lower + upper) / 2.
The binary search performs the comparison till the element is found or division of list gives one element.

Example: Input list 0, 1, 2, 9, 10, 11, 15, 20, 46, 72

Key: 15

→ Iteration 1
Lower = 0 Upper = 9
mid = (lower + upper) / 2 = (0 + 9/2) = 4.5

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

↑
a[Mid] ! = 15
a[Mid] < KEY; Lower = mid + 1

→ Iteration 2
Lower = 5 Upper = 9
Mid = (Lower + Upper) / 2 = (5 + 9) / 2 = 7

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

↑
a[Mid] ! = 25
a[Mid] > KEY; Upper = mid – 1

→ Iteration 3
Lower = 5 Upper = 6
Mid = (Lower + Upper) / 2 = (5 + 6) / 2 = 5.5

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

↑
a[Mid] ! = 15
a[Mid] < KEY; Lower = mid + 1

→ Iteration 4
Lower = 6 Upper = 6
Mid = (Lower + Upper) / 2 = (6 + 6) / 2 = 6

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

↑
a[Mid] = 15
Number is found

Q.5 Attempt any TWO of the following : [16]

Q.5(a) Write an algorithm to insert new node at the beginning, at the middle and at the end of a singly linked list. [8]

(A)

```
# include <alloc.h>
# include <stdio.h>
# define NULL 0
struct node
    {
        int data;
        struct node *link;
    }
    struct node * head = NULL;
```

```
void insert ( );
void search ( );
void delete ( );
void display ( );
void create ( );
void main ( )
{
    int ch;
    while (1)
{
    pf ("1 : create\n");
    pf ("2 : insert\n");
    pf ("3 : delete\n");
    pf ("4 : display\n");
    pf ("5 : search\n");
    sf ("%d", & ch);
        if (ch = = 6)
            break;
    switch (ch)
    {   case 1 :    create ( );
                    break;
        case 2 :    insert ( );
                    break;
        case 3 :    delete ( );
                    break;
        case 4 :    display ( );
                    break;
        case 5 :    search ( );
                    break;
        default : pf ("wrong choice \n");
        }
    }
}

    void create ( )
    { int data; struct node * temp, *temp1;
    pf ("please enter the data … 0 to stop\n");
    sf ("%d", & data);

    while ( data ! = 0)
    {
```

```
        if ( head = = NULL)
        {
            temp = malloc (siteof (struct nodd));
            temp → data = data;
            temp → link = NULL;
            head = temp;
        }
    else
        {   temp1 = malloc (siteof (struct node));
            temp1  → data = data;
            temp1  → link = NULL;
            temp   → link = temp1;
            temp   =   temp1;
        }
    }
            return (head);
    }

    void display (struct node * head)
    {       struct node * temp;
        if (head = NULL)
            pf ("List empty \n");
    else
        { temp = head;
        while ( temp! = NULL)
            {
            pf ("%d"; temp → data);
            temp = temp → link;
            }
        }
    }
void insert ( )
{
pf("1 : insert at first position\n");
pf("2 : insert at middle position\n")
pf("3 : insert at last position\n");
pf("enter your choice\n");
sf("%d", & choice);

pf("enter data to be inserted\n");
sf("%d", & data);
```

```
    if (choice = = 1)
    {
        temp = malloc (siteof (
                                    struct node));
        temp → data = data;
        temp → link = head;
        head = temp;
    }
if (choice = = 2)
        { printf ("Enter data after which you want to insert data");
            sf ("%d", & data1);
    temp = head; found = 0;
    while (temp ! = NULL && ! found)
    {
        if (temp → data = = data1
                    found = 1;
    else
        temp = temp → link;
    }
if   (temp = = NULL)
        pf ("Data can not be inserted");
else
    {   temp 1 = malloc (siteof (struct node));
        temp 1 → data = data;
        temp 1 → link = temp → link;
        temp → link = temp;
    }
}
if (choice = = 3)
    {   temp = head;
        while ( temp → link ! = NULL)
                { temp = temp → link;
                }
            temp 1 = malloc (sizeof (struct node);
            temp 1 → data = data;
            temp 1 → link = NULL;
        temp  → link = temp1;
        }
    }
```

Q.5(b) Explain Linear Probing and Chaining techniques of Hashing with example. [8]

(A) Collision Resolution Methods

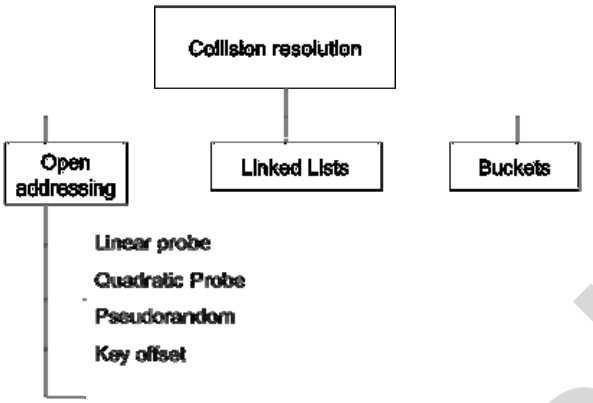


Fig. 1

In this section I will discuss the Linear probe and Quadratic probe

Linear Probe

In a linear probe when data cannot be stored in the home address we resolve the collision by adding 1 to the current address.

Example for Linear Probe collision Resolution

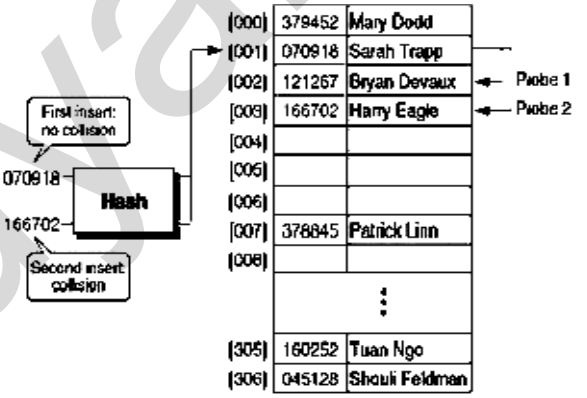


Fig. 2 : Linear Probe Collision Resolution

Quadratic Collision Resolution

In the quadratic probe, the increment is the collision probe number squared. Thus for the first probe we add 1^2, for the second collision probe we add 2^2 etc.

Table : Quadratic Collision Resolution Increments

Probe number	Collision location	Probe^2 and increment	New address
1	1	1^2 = 1	1 + 1 = 02
2	2	2^2 = 4	2 + 4 = 06
3	6	3^2 = 9	6 + 9 = 15
4	15	4^2 = 16	15 + 16 = 31
5	31	5^2 = 25	31 + 25 = 56
6	56	6^2 = 36	56 + 36 = 92
7	92	7^2 = 49	92 + 49 = 41
8	41	8^2 = 64	41 + 64 = 05
9	5	9^2 = 81	5 + 81 = 86
10	86	10^2 = 100	86 + 100 = 86

Q.5(c) Explain Breadth First Search with example. [8]

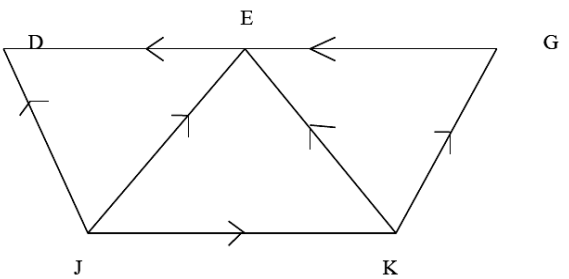
(A) Description of Algorithm:

BFS (Breadth first search) is a graph search algorithm that begins that specific node and explores on the neighbouring nodes. It is used to find minimum path P between two particular nodes.

The algorithm works as follows:

Initialize all the nodes to the ready state. Insert starting node in a queue and change its status to waiting state. Then remove front node (N) from queue and change its status to visited state. Find adjacent nodes of removed node (N). Insert them into the queue and change their status to waiting state. It continues the process of insertion and deletion of elements till queue becomes empty. To find path algorithm uses origin to keep track of nodes whose adjacent nodes are inserted into the queue. At the end origin is used to find path from source node to destination node.

Example: - Consider following graph G.
Find all nodes reachable from node K to D.

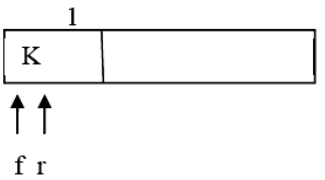


All nodes are in ready state.

Step 1- Initially insert node K in a queue and add NULL to orig

Front = 1 queue = K

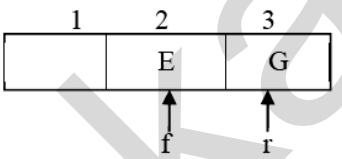
Rear = 1 orig = O



Step 2 – Remove front element K and change its states to visited. Find its adjacent nodes and insert them into queue if their states is ready.

Front = 2 queue = K, E, G

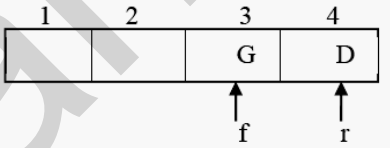
Rear = 3 orig = O, K, K



Step 3 – remove front element E and change its states to visited. Find its adjacent nodes and insert them into queue if their states is ready.

Front = 3 queue = K, E, G, D

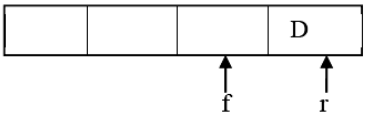
Rear = 4 orig = O, K, K, E



Step 4 – Remove front element G and change its states to visited. Find its adjacent nodes and insert them into queue if their state is ready. E is already visited.

Front = 4 queue = K, E, G, D

Rear = 4 orig = O, K, K, E



Step 5 - Remove front element D and change its states to visited. Find its adjacent nodes.

Path betweenand insert them into queuK to D is eK -> E -> D

and insert them into queue if their state is ready.

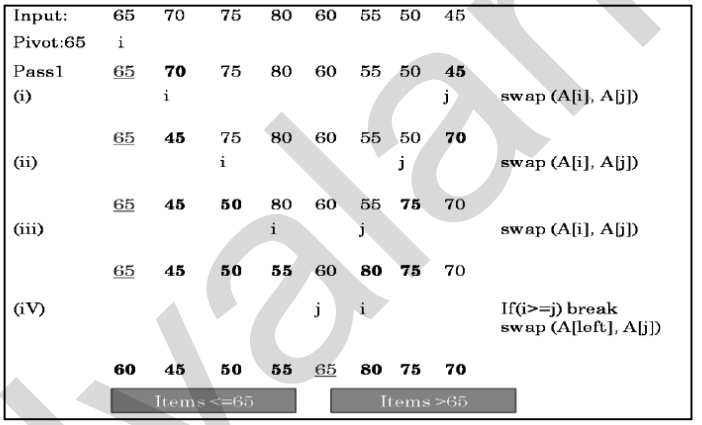
Q.6 Attempt any TWO of the following : [16]

Q.6(a) Explain Quick sort with example. [8]

(A) In quick sort, in each pass only one element is fixed at its final position in a list. For sorting, fix the first element as a pivot element. Use two index variables (i,j) with initial values of first index position and n-1 index position respectively. Compare a pivot element with i^{th} index element till you find a

number greater than pivot element. If the i^{th} element is less than pivot element then increment the index by one. If i^{th} element is greater than pivot element than mark the element and then compare pivot element with j^{th} element till you find a number less than pivot element. If the jth element is greater than the pivot element then decrement index by one. If i^{th} element is less than pivot element mark that element. After finding elements greater than and less than pivot element, interchange both the elements.

Again compare pivot element with i^{th} and j^{th} element till i and j are equal to each other or cross to each other. Then fix the pivot element at its position such a way that all elements preceding it should be less than it and all the elements following it should be greater than it. Then divide the list into two parts without including fix element. Each part of the list should be sorted with the same procedure till all elements are placed at their final position in sorted list.



After pass 1 all the elements which are less than or equal to 65 are before 65(pivot) and all the other elements which are greater than 65 are at the right side.

Now we need to repeat same process for left half and right half of the array. then we will get sorted array.

Q.6(b) Write a program to read an integer number. Print the reverse of this number. [8]

(A)

```
#include <stdio.h>
int main()
{
    int num,rev;
    printf( "\nEnter a number :" );
```

```
        scanf( "%d" ,&num);
        rev=reverse(num);
        printf( "\nAfter reverse the no is : %d" ,rev);
        return 0;
    }
    int sum=0,r;
    reverse( int num)
    {
        if (num)
        {
            r=num%10;
            sum=sum*10+r;
            reverse
            (num/10);
        }
        else
        return sum;
        return sum;
    }
```

Q.6(c) Write a program to implement Queue using array. **[8]**

(A) Program for Queue using array:

```
#include<stdio.h>
#include<conio.h>
#define MAX 50

struct queue
{
    int rear,front;
    int elems[MAX];
};

int empty(struct queue *q)
{
    if(q->rear < q->front)
        return 1;
    else
        return 0;
}

int full(struct queue *q)
```

```
{
        if(q->rear==MAX-1)
            return 1;
        else
            return 0;
}

void insert(struct queue *q,int x)
{
        if(full(q))
            printf("Queue is full");
        else
            q->elems[++(q->rear)]=x;
}
int removes(struct queue *q)
{
        if(empty(q))
            return -1;
        else
            return q->elems[(q->front)++];
}

void show(struct queue q)
{
        int i;
        for(i=q.front;i<=q.rear;i++)
            printf("\t%d",q.elems[i]);
}

void main()
{
        int ch,x;
        struct queue q;
        q.rear=-1;
        q.front=0;
        do
        {
            printf("\nMenu\t1.Insert\t2.Remove\t3.Show\t4.Exit");
            printf("enter ur choice");
            scanf("%d",&ch);
            switch(ch)
```

```
            {
                case 1:
                    printf("enter element");
                    scanf("%d",&x);
                    insert(&q,x);
                    break;

                case 2:
                    x=removes(&q);
                    if(x==-1)
                        printf("Queue is empty");
                    else
                        printf("%d is removed",x);
                    break;
                case 3:
                    printf("elements are:");
                    show(q);
                    break;

                case 4: exit(0);
            }
        } while(ch!=4);
        getch();
}
```

❑ ❑ ❑ ❑ ❑