DSA Assignment

Page No.
Date : / /

Name : Slok Regmi
Roll no : 221642
Software (Morning)

Q 1) What are ADT ? Write an ADT for natural numbers.

=> ADT stands for Abstract - Data - Type . It is a high - level description of a set of operations that can be performed on a data - structure, without specifying how these operations are implemented. In other words, an A.D.T provides a mathematical model for a certain type of data, defining what operations can be performed on that data and what the behaviour of those operations should be.

The key idea behind ADTs is to separate the logical properties of the data structure from its implementation details. This allows programmers to use & understand data structure at a higher level.

Example : Stack, Queue, List, Set.

ADT

```
class NaturalNumber {
    unsigned int value;
public:
    void add(unsigned int ~~value~~ num) {}
    void substract(unsigned int num) {}
```

Page No.
Date : / /

```
void divide (unsigned int num, unsigned int val) {}
bool isEqual (unsigned int num, unsigned int val) {}
bool isZero (unsigned int num) {}
bool isEven (unsigned int num) {}
bool isOdd (unsigned int num) {}
```

Q2) Differentiate between data type and data structure. ~~What~~ What are the two parts of definition. Explain.

Data Type	Data Structure
i) It is a category that defines the type of data and its operation	i) It is a way of organizing and storing data efficiently.
ii) It is relatively simple and predefined	ii) Complexity varies based on the organization and functionality
iii) It is more fundamental i.e. individual elements.	iii) It is more high level i.e. collection of elements.
iv) It contains basic operations like addition and subtraction	iv) It consists complex operations like insertion, deletion.
v) Eg: Integer, Float, String	v) Eg: Array, List, Stack

Page No.
Date : / /

Two parts of Definition:

⇒ Data Type:

1. Representation:

Describes how the values of the data type are stored in the memory. It includes details about the size of the memory block allocated for ~~the~~ each variable of that type.

2. Operations:

Specifies the set of operations that can be performed on variables of that data type. This includes arithmetic, logical, comparison operations.

⇒ Data Structure:

1. Logical Structure:

Describes how the data is organized and the relationships between the elements. It includes information about how the individual pieces of data are related to each other, their order and any constraints on the data.

2. Operations:

Specifies the set of operations that can be performed on the date structure. These operation could include insertion, deletion, traversal, searching, sorting, among others.

Page No.
Date : / /

Q. 3) Discuss the concept of ADT. Represent rational numbers as an ADT.

⇒ An Abstract Data Type is a high-level description of a set of operations that can be performed on a data structure, without specifying how these operations are implemented. It provides a mathematical model for a certain type of data defining what operations can be performed on that data and what the behaviour of those operations should be. The key idea is to ~~spe~~ separate the logical properties of the data structure from its implementation details, allowing for flexibility and modularity in software design.

Represent ADT for rational numbers

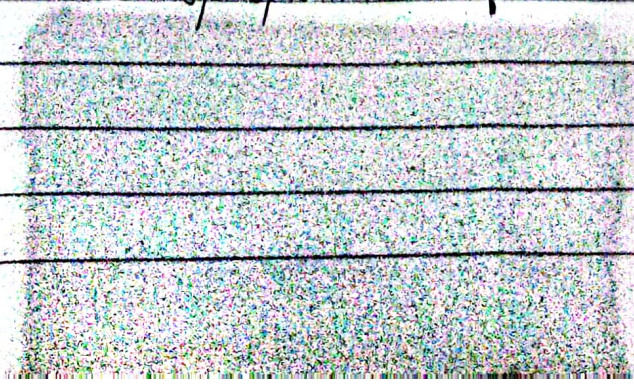
```
class RationalNumbers{
    int numerator, denomenator;
RationalNumber add (const RationalNumber & other) const;
RationalNumber sub( const RationalNumber & other) const;
RationalNumber mul (const RationalNumber & other) const;
RationalNumber div (const RationalNumber & other) const;

bool isEqual (const RationalNumber & other) const;
bool isGreater (const RationalNumber & other) const;
bool isSmaller (const RationalNumber & other) const;
int getNum ()const;
int getDeno ()const;
};
```

Page No.
Date : / /

Q. 4) Differentiate between ADT and C++ classes.

⇒ ADT	C++ classes.
i) It is abstraction that define set of values and set of operations on these values.	i) It is self-contained component which consists of methods and properties to make certain type of data useful.
ii) It is user defined data type.	ii) It is instance of class
iii) ADT is not necessarily an OOP concept.	iii) Classes and its objects is an OOP concepts.
iv) It allocates memory when data is stored.	iv) When we instantiate an object then memory is allocated.
v) Common example include lists, stacks, sets etc.	v) Objects have states and behaviours Test t = new Test();
vi) ADT is made up of primitive data types	vi) An object is an abstract type with the addition of polymorphism.
vii) It is a type for objects whose behaviour is defined by a set of values and a set of operations.	vii) It is a basic unit of Object-Oriented Programming



Page No.
Date : / /

Q.5) Explain the different type of primitive operations that we can perform on data structures.

⇒ Primitive operations on data structure refer to the fundamental operations that can be performed on various types of data structures. These operations are essential building blocks for manipulating and managing data within a data structure.

1. Access / Retrieve :
   ⇒ Retrieve the value of a specific element within the data structure.
   Eg: Accessing an array element by index, retrieving a node's value.

2. Search :
   ⇒ Find the location or presence of a particular element within the data structure.
   Eg: Searching for a key in a binary search tree, looking for a item.

3. Insertion :
   ⇒ Add a new element to the data structure
   Eg: Inserting a new node in a linked list, adding new element.

4. Deletion :
   ⇒ Remove an element from the data structure.
   Eg: Deleting a node from a linked list.

5. Traversal :
   ⇒ Visiting and processing each element in a data structure.
   Eg: Traversing all nodes in a tree, iterating through all elements

Page No.
Date : / /

6. Sorting:
⇒ Arrange the element in a specific order within the data structure.
Eg: Sorting a array in ascending /descending order.

7) Counting:
⇒ Determine the number of elements in the data structure.
Eg: Counting the number of nodes in atree.

Q. 6) Why data structure is needed? Explain advantages of abstraction.

⇒ The main importance why the data structure is needed is because:

i) Organizing of data:
⇒ Data structure provide a way to organize and store data efficiently. They define how data is arrange, accessed and modified.

ii) Efficient Retrieval & Modification:
⇒ Properly designed data structures enables efficient retrieval and modification of data, minimizing the time complexity of operations.

iii) Optimized Memory Usage:
⇒ Data structure helps optimize memory usage by providing organized storage mechanisms. They allocate and manage memory in a way that reduces wastage.

iv) Algorithm Design:
⇒ Data structure ~~from the basic~~ form the basis for algorithm

design. Algorithms often rely on specific data structure to achieve optimal performance and solve problem efficiently.

v) Complex problem solving :
⇒ Complex problems can be broken down into simpler sub problems using appropriate data structures, fasciliating a more systematic and efficient approach.

The advantages of abstraction in Data Structure are:

1) Simplification of Complexity :
⇒ Abstraction hides the underlying complexity of data structures and operations, allowing developer to focus on high-level design and functionality without dealing with intricate details

ii) Ease of understanding :
⇒ Abstraction provides a conceptual framework that makes it easier for developers to understand and reason about data structure.

iii) ~~Modularity & Reusability~~ : Flexibility & Adaptability :
⇒ Abstraction allows for different implementations of the same abstract data type. This flexibility enables developers to choose the most suitable implementation based on specific requirements and constraints

Page No.
Date : / /

Q.7) What is the condition of stack underflow & overflow. Write down the module for stack push and pop operation.

=> Stack UnderFlow:

Stack underflow occurs when you attempt to pop an element from an empty stack. In a stack, elements are removed in a last in first out (LIFO) manner. If you try to pop (remove) an element from an empty stack, there is no element to retrieve, resulting stack underflow.

Eg:

```
if stack is empty:
    raise StackUnderFlowError
else:
    element = stack.pop()
```

Stack Overflow:

Stack overflow occurs when you try to push an element onto a stack that has reached its maximum capacity. Stacks are often implemented using a fixed-size array or a linked list with a finite amount of memory allocated for elements. If you attempt to push (add) an element to a full stack, there is no space available, resulting in stack overflow.

Eg:

```
if stack is full:
    raise StackOverFlowError
else:
    stack.push(element)
```

Page No.
Date : / /

Module:

```
bool is_full () const {
    return top == max_size - 1;
}
bool is_empty () const {
    return top == -1;
}
void push (int element) {
    if (is_full()) {
        throw overflow_error("Stack Overflow");
    } else {
        data[++top] = element;
    }
}
void pop () {
    if (is_empty()) {
        throw underflow_error("Stack Underflow");
    } else {
        return data[top--];
    }
}
```

(Q 8) What is Data Structure? Show stack of converting to post fix
P + Q - (R * S / T + U) - V * W

> A data structure is a way of organizing and storing data to perform operations efficiently. It defines the relationship between the data and operations that can be performed on that data. Data structure provides a means to manage and organize large amount of information.

Page No.
Date : / /

Expression (Scanned)	Stack	Post Fix
P	Empty	P
+	+	P
Q	+	PQ
-	-	PQ+
(	(-	PQ+
R	(-	PQ+R
*	*(-	PQ+R
S	*(-	PQ+RS
/	/(-	PQ+RS*
T	/(-	PQ+RS*T
+	+(-	PQ+RS*T/
U	+(-	PQ+RS*T/U
)	-	PQ+RS*T/U+
-	-	PQ+RS*T/U+-
V	-	PQ+RS*T/U+-V
*	*-	PQ+RS*T/U+-V
W	*-	PQ+RS*T/U+-VW
Empty	Empty	PQ+RS*T/U+-VW*-

The required postfix expression: PQ+RS*T/U+-VW*-

Page No.
Date : / /

Q.9) What are the advantages of post fix expression over infix expression. Convert the infix expression to postfix.

=> Some advantages of postfix expression are:

i) Elimination of parenthesis:
→ In postfix notation, there is no need for parenthesis to indicate the order of operations. The postfix notation inherently captures the precedence and ~~assoca~~ associativity of operators.

ii) Simplified ~~Parsing~~ & Evaluation:
→ Post fix expressions are well-suited for evaluation using a stack based algorithm. The operands are pushed onto the stack and operators trigger the necessary operations by popping operands from stack.

iii) Compact Representation:
→ Post fix often provides a more compact representation of mathematical expression, especially those with multiple operators.

iv) Reduction of operand movement:
In infix operations, the movement of operands due to parenthesis and operator precedence can get complex. Postfix expression on the other hand involves a sequential involvement.

Page No.
Date : / /

~~Pf~~ (A+B*C/D)+E*F-(G*H+I-J)

Scanned ~~Opr~~	Stack	Postfix
(	(	Empty
A	*(	A
+	+(	A
B	+(	AB
*	*+(	AB
C	*+(	ABC
/	/+(	ABC*
D	/+(	ABC*D
)	Empty	ABC*D/+
+	+	ABC*D/+
E	+	ABC*D/+E
*	*+	ABC*D/+E
F	*+	ABC*D/+EF
-	-	ABC*D/+EF*+
(	(-	ABC*D/+EF*+
G	(-	ABC*D/+EF*+G
*	*(-	ABC*D/+EF*+G
H	*(-	ABC*D/+EF*+GH
+	+(-	ABC*D/+EF*+GH*
I	+(-	ABC*D/+EF*+GH*I
-	-(-	ABC*D/+EF*+GH*I+
J	-(-	ABC*D/+EF*+GH*I+J
Empty	Empty	ABC*D/+EF*+GH*I+J--

∴ Reqd postfix: ABC*D/+EF*+GH*I+J--

Page No.
Date : / /

Q.10) What are the limitation of Linear queue? Algorithm to enqueue & dequeue elements in the circular queue.

⇒ There are many limitation of linear queue. They are

i) Fixed size:

⇒ Many implementations of linear queues have a fixed size, meaning they can only hold a specific no. of elements

ii) Memory wastage:

⇒ If the size of the linear queue is fixed and set to accomodate the maximum expected number of elements, memory may get wasted.

iii) Queue Full and Queue Empty Condition:

⇒ Linear queues can face issues with determining whether the queue is full or empty, especially in implementation with fixed sizes.

Algorithm for enqueue: circular queue

⇒ Check if the queue is full

⇒ ↳ If (rear + 1) % MAX_SIZE == Front

{

Display full }

⇒ If queue is empty {

front = rear = 0;

}

⇒ else

rear = (rear+1) % MAX_SIZE

Page No.
Date : / /

→ queue [rear] = element.

Dequeue of Circular Queue

1. Check if queue is empty:
   if front == -1
   { Display underflow error }
2. Otherwise:
   element = queue [front]
   front = (front + 1) %. MAX_SIZE.
3) If front = rear then
   front = rear = -1
4) Return element

Q. 11) Explain priority queues Write an algorithm of insertion in linear queue.

⇒ A priority queue is an abstract data type that stores elements along with associated priorities and allows elements with higher priorities to be served before elements with lower priorities. It is a queue-based data structure where each element is assigned a priority, and the element with the highest priority is dequeued first. Priority queues are often used in scenarios where the order in which elements are processed is determined by their priority rather than their arrival time.

Page No.
Date : / /

Algorithm for insertion:

1) If rear == maxSize - 1;
   Display "Queue is full"
2) Otherwise
   rear ++;
   queue[rear] = value;
   Display "Value Enqueue"

Q.12) Define list. What are the primitive operation performed on list? Explain the dynamic implementation of list with examples.

⇒ List is a linear data structure that stores a sequence of elements. Each element in the list is associated with an index or position, allowing for easy traversal and manipulation. Lists can be implemented using various data structures like arrays or linked list.

Primitive operation performed on list are:

i) Insertion:
   → Adding an element to the list. This can be done at the beginning (prepend) or at end (append) of list.

ii) Deletion:
   Removing an element from the list. This can involve removing an element at a specific position, by value, or from beginning.

iii) Access (Read):

→ Retrieving the value of an element at a specific position in the list.

iv) Search:

→ Finding the position or presence of a specific element in the list.

In dynamic implementation of list, the size of the list is not fixed and can change dynamically during program execution.

Eg:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node * next;
Node (int val) : data (val), next (nullptr) {}
};
class LinkedList {
  private :
    Node * head;
public:
    LinkedList () : head (nullptr) {}
    void prepend (int value) {
        Node * newNode = new Node (value);
        newNode -> next = head;
        head = newNode;
    }
```

```
void display () const {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
}
int main() {
    LinkedList mylist;
    mylist.prepend(20);
    mylist.prepend(30);
    mylist.display();
    return 0;
}
```

Q.13) What is double linked list? Write an algorithm for deletion operation in double linked list.

⇒ A doubly linked list is a type of linked list where each node contains a data element and two pointers, one pointing to the next node in the sequence and another pointing to the previous node.

Algorithm for deletion operation is double linked

1) IF head == NULL
   Then return "Empty List"
2) IF del == head
   THEN head = del -> next

Page No.
Date : / /

IF head != NULL
THEN head -> prev = NULL

3 ELSE IF del -> next == NULL
THEN del -> prev -> next = NULL

4 Else
THEN del -> prev -> next = del -> next
del -> next -> prev = del -> prev

5) Free del.

Q. 14) How circular linked list can be implemented using singly linked list? Explain.

⇒ A circular linked list can be implemented using a single linked list ensuring that the last node's next pointer points back to the first node, creating a loop or circular structure. In other words, the next pointer of the last node in the list points to the first node, closing the loop.

Code eg:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node * next;
```

```
    Node (int value) : data (value), next (nullptr) {}
};
class CircularLinkedList {
    private:
        Node * head;
public:
    CircularLinkedList (): head (nullptr) {}

    void insert (int value) {
        Node * newNode = new Node (value);
        if (!head) {
            head = newNode;
            head -> next = head;
        } else {
            Node * last = head;
            while (last -> next != head) {
                last = last -> next;
            }
            last -> next = newNode;
            newNode -> next = head;
        }
    }
    void display () const {
        if (!head) {
            cout << "List is empty";
            return;
        }
        Node * current = head;
        do {
```

```
cout << current -> data << " ";
current = current -> next;
} while (current != head);
cout << endl;
} };
void deleteNode (int value) {
    if (!head) {
        cout << "Empty";
    return;
    }
    Node *current = head;
    Node * prev = nullptr;
    do {
        if (current -> data == value) {
            if (current == head) {
                head = head -> next;
            } else {
                prev -> next = current -> next;
            }
        ~~delete current;~~ delete current;
        return;
    }
    prev = current;
    current = current -> next;
} while (current != head);
    cout << "Can't delete";
} }
int main () {
    ~~Circular~~ CircularLinkedList myList;
```

```
myList.insert(10);
myList.insert(20);
myList.display();
myList.deleteNode(20);
return 0;
}
```

Q.15) Write a algorithm to evaluate arithmetic expression in Postfix string. Apply the algorithm to evaluate : AB+C-BA+C$-

=> Algorithm

1. Initialize an empty stack to hold operands
2. Scan the postfix expression from left to right.
3. For each symbol in postfix expression:
   - → If the symbol is operand, push onto stack
   - → If symbol is an operator.
     - ↳ Pop required num of operands from stack
     - ↳ Perform operation and push to stack
4) After scanning the entire expression, the result should be the only element left in the stack

AB+C-BA+C$-

(Assume: A=1, B=2, C=3)

=> let expression be

1 2 + 3 - 2 1 + 3 $

Page No.
Date : / /

Scanned 8	Stack.	Operation	Calculation.
1	1	push	
2	2 1	push	
+	3	pop (2 element) & evaluate.	3 2+1=3
3	3 3	push	
-	0	pop (2 element) & evaluate	3-3=0.
2	2 0	push	
1	1 2 0	push	
+	3 0	pop (2 element) & evaluate	1+2=3
3	3 3 0	push.	
$	9 0	pop (2 element) & evaluate	3^3=9
-	0-9		

∴ -9 is the final evaluated answer.

Page No.
Date: / /

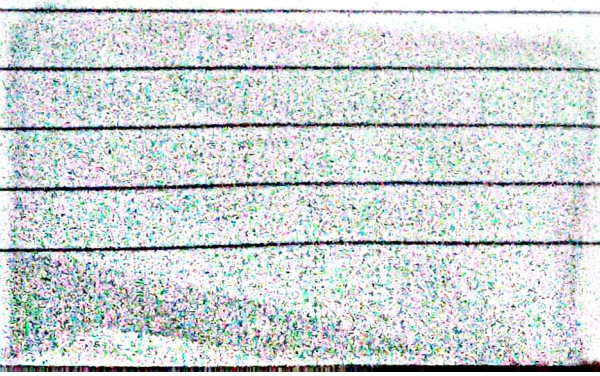
Q.No 16) How can we implement stack as a linked list. Use appropriate algorithm to show operation of stack using linked list.

⇒ To implement a stack using a linked list, we used a class to represent the nodes of the linked list and another class for the stack itself.

Using appropriate algorithm:

```
#include <iostream>
Using namespace std;
class Node {
  public :
    int data;
    Node * next;
Node (int value) : data (value), next (nullptr) {}
};
class Stack {
  private :
    Node * top;
public :
    Stack () : top (nullptr) {}
void push (int value) {
      Node * new_node = new Node (value);
    new_node -> next = top;
      top = new_node;
       cout << "pushed";
    }
int pop () {
```

```
int popped_value = top -> data;
    Node * temp = top;
    top = top -> next;
    delete temp;
     return popped_value;
  }
void display () {
        Node * current = top;
        while (current != nullptr ) {
            cout << current -> data ;
            current = current -> next;
          }
       cout << "Nullptr";
  } };
int main () {
     Stack stack;
    stack.push (1);
    stack.push (2);
    stack.push (3);
    stack.display ();
 return 0;
}
```



Page No.
Date : / /

Q.17) Evaluate given exp using postfix notation: A * (B+C) - (D/E)

→ ~~Trans~~ Converting infix notation to postfix notation

~~Infix~~ Scanned.	Stack	Postfix
A	Empty	A
*	*	A
(	(*	A
B	(*	AB
+	+ (*	AB
C	+ (*	ABC
)	*	ABC+
-	-	ABC+*
(	(-	ABC+*
D	(-	ABC+*D
/	/ (-	ABC+*D
E	/ (-	ABC+*DE
)	-	ABC+*DE/
Empty	Empty	ABC+*DE/-

We know that..

A = 5, B = 6, C = 2, D = 12, E = 4.

So:

5 6 2 + * 12 4 / -

Page No.
Date : / /

Scanned	Stack	Operation	Calculation
5	5	push.	
6	6 5	push.	
2	2 6 5	push	
+	8 5	pop 2 elemed & add. push.	~~2+6~~ 6+2=8
*	40.	pop 2 elemt & perform operatin	5 * 8 = 40
12	12 40	push	
4	4 12 40	push.	
/	3 40	pop 2 elemet & perform operation	12/4 = 3.
-	37.	pop 2 elemt & perform operatin.	40-3

∴ Final answer = 37 Ans

Page No.
Date : / /

Q. 18) Write the algorithm for insertion in singly linked list at the beginning at the end and at specified position.

⇒ Insert as first:

1) temp = (SLL *) malloc (sizeof (SLL));
2) temp → info = data;
3) if (First = NULL)
   temp → next = NULL
   first = temp
4) else.
   temp → next = first.
   & First = temp

⇒ Insert as last.:

1) temp = (SLL *) malloc (sizeof (SLL))
2) temp → info = data
3) temp → next = NULL
4) p = first
5) while (p → next ≠ NULL)
   p = p → next;
6) End while
7) p → next = temp

⇒ Insert at n^{th} position:

1) temp = (SLL *) malloc (sizeof(SLL));
2) temp → info = value
3) p = first
4) while (n > 2)
   p = p → next

n--

5) End while.

6) temp -> next = p -> next

7) p -> next = temp.

Q.# 19) What is double ended queue? Explain the insertion algorithm in ~~deque~~ double ended queue.

=) A double ended queue, often abbreviated as deque, is a data structure that allows insertion and deletion of elements from both ends, i.e. the front and the rear. It ~~compr~~ combines the properties of a stack (Last In, First Out) and a queue (First In, First Out)

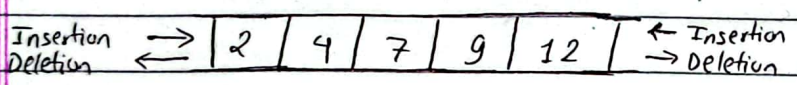


Fig: Structure of deque.

There are two types of double ended queue:

i) Input Restricted Deque:

→ An element can be only added from one end but delete ~~only~~ from both the ends.

ii) Output Restricted Deque:

→ An element can be deleted from one end but & insert from both the ~~e~~ ends.

Page No.
Date : / /

=> In deque there are 2 cases:
Case:1 Insert at Front ~~node~~ ~~Node~~ :

1) If ~~queue~~ != full (rear + 1) % MAX_SIZE == Front
   Display array is full.
2) ~~Else~~ If (front == -1)
   front = rear = 0.
3) Else if (front == 0)
   front = MAX_SIZE - 1;
4) Else
   front--;
5) ~~Else~~ Que [front] = data

Case 2: Insert at Rear.

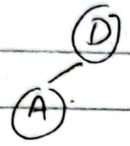
1) If (rear + 1) % MAX_SIZE == ~~FRONT~~ Front
   Display Array is full.
2) If (front == -1)
   front = rear = 0.
3) Elseif (rear == Max_size - 1)
   R. rear = 0
4) Else
   rear++.
5) Que [rear] = data

Page No.
Date : / /

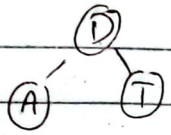
20) Draw a BST from the string DATASTRUCTURE & ~~post~~ traverse in post & pre order.

⇒ D.

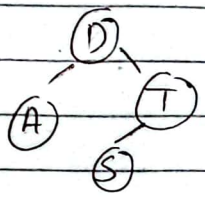
⇒ Insert A.



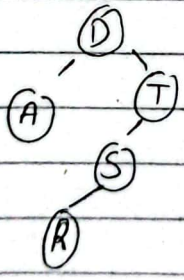
⇒ Insert T



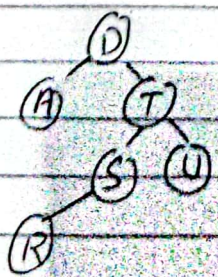
⇒ Insert S.



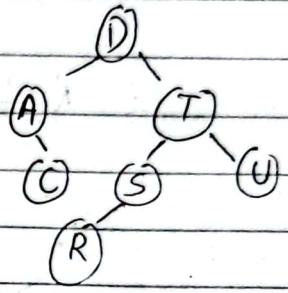
⇒ Insert R.



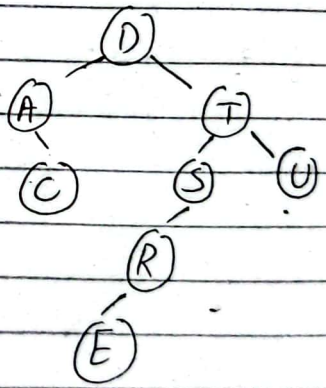
⇒ Insert U



⇒ Insert C.



⇒ Insert E.



Page No.
Date : / /

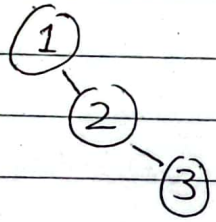
Preorder Traversal:

D, A, C, T, S, R, E, U.

Postorder Traversal:

C, A, E, R, S, ~~I U, D~~ U, T, D

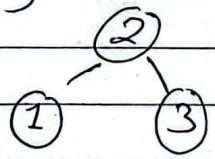
Q.21) Construct the AVL tree from the following 1, 2, 3, 5, 7, 9, 10, 12



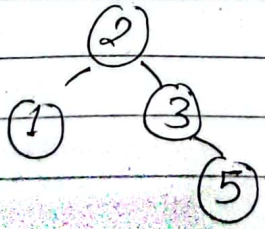
⇒ Inserting 1, 2, 3

B.f of 1 = -2

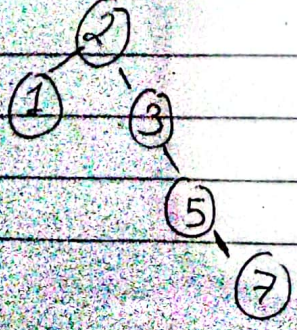
Applying L-L traversal



Inserting 5



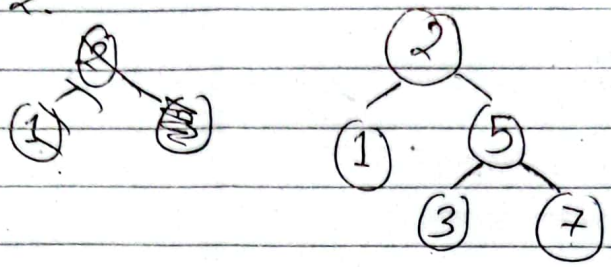
Inserting 7:



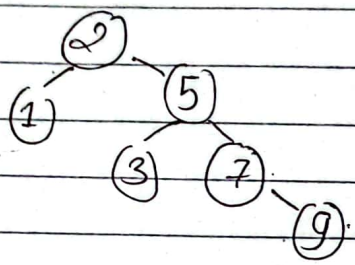
Page No.
Date : / /

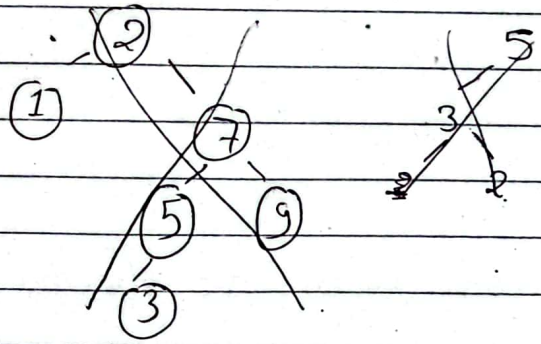
B.f of 3 : -2.

So:

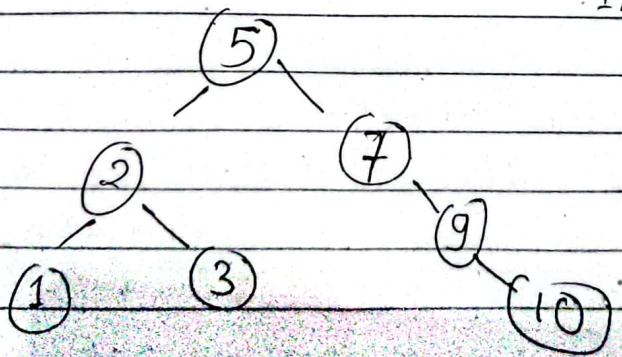


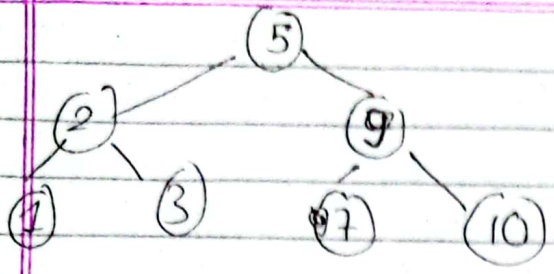
Inserting 9



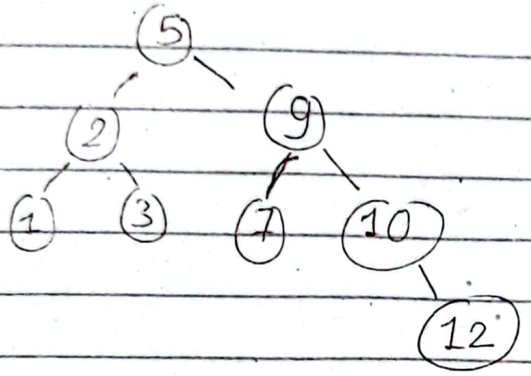


Insert 10





Inserting 12



= ∴ Ans

Q.22) Define recursion and explain the algorithm which is used to solve the TOH problem.

→ Recursion is a programming concept where a function calls itself directly or indirectly to solve a problem. In a recursive ~~function~~ solution, the problem is divided into smaller sub-problems that are ~~smil~~ similar to original problem. Each ~~res~~ recursive call works on a smaller instance of the problem until a base case is reached, at which point the solution is straightforward.

Page No.
Date : / /

TOH problem:

Tower of Hanoi is a classic problem that involves moving the stack of disks from one rod to another, subject to constraints

i) Only one disk can be moved at a time.

ii) A disk can only be moved to the top of another rod if it's smaller than the disk already on that rod.

iii) Only the top disk from the rod can be moved in each operation.

Recursive Algorithm:

1) Initialize the number of disk as n, and three rod as source, target and auxiliary.

2) The base case (n == 1) handles moving a single disk from the source to the target.

3) The recursive steps move "n-1" disk from the source to the auxiliary, move the n^{th} disk from the source to the target, and finally move the "n-1" disk from the auxiliary to the target.

Page No.
Date : / /

Q.23 Differences between recursion & iteration. WAP algorithm to find fibonacci series.

⇒

	Recursion	Iteration.
i)	A function calls itself directly or indirectly to solve a problem.	i) Repeated Execution of a set of statements using loops
ii)	The flow of control is handled by the function calls and the call stack	ii) The flow of control is managed explicitly using loops.
iii)	Requires a base case to terminate the recursive calls and prevent infinite recursion.	iii) Relies on loop conditions to control when the iteration steps.
iv)	May use more memory due to call stack, especially for deep recursion.	iv) Generally uses less memory as it doesn't rely on function calls and call stack
v)	Well suited for problems with a natural recursive structure.	v) Well suited for problems that can be expressed more naturally with loops.

Page No.
Date : / /

Algorithm:

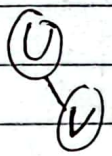
i) => ~~let n~~ Initialize 'n' as the no. of of fibonacci series you want do display.

ii) The base ~~clas~~ cases handles the first two ~~positio~~ position in the fibonacce sequence i.e. 0 & 1 where it returns 0 & 1 respectively

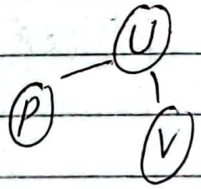
iii) The recursive calculates the Fibonacci number for position 'n' by adding the $(n-1)^{th}$ & $(n-2)^{th}$ Fibonacci number.

Q.24) Given letters are to be inserted in order into an empty binary search tree: U, V, P, Q, M, N, O, R, K, W, C, D. Find the different tree ~~travel~~ traversal of this binary tree.

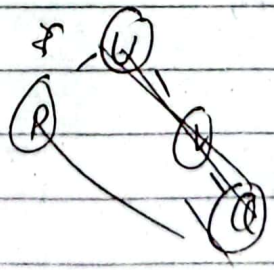
Insert V.

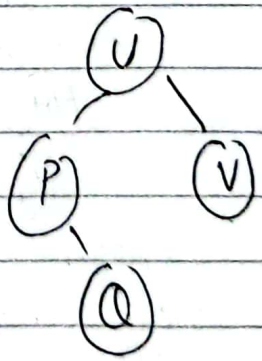


Insert P.



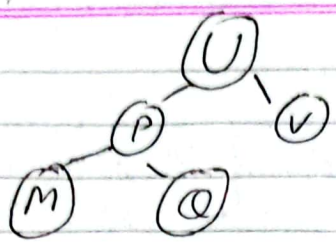
Insert Q



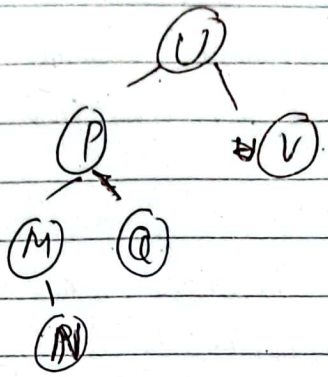


Page No.
Date : / /

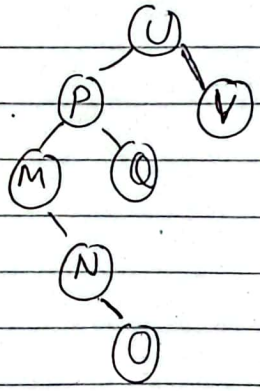
Insert M



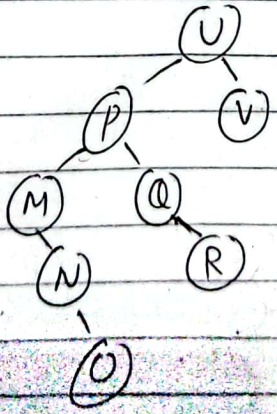
Insert N.



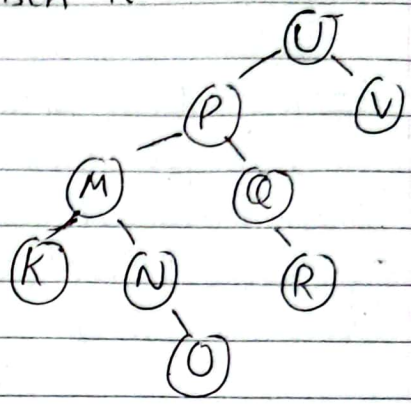
Insert O.



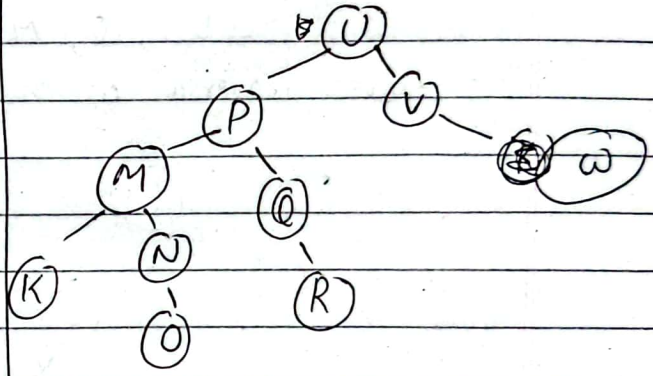
Insert R.



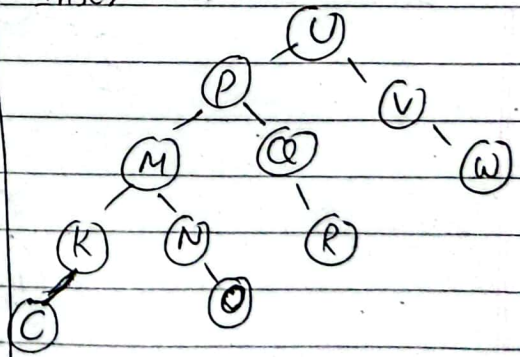
Insert K



Insert W.

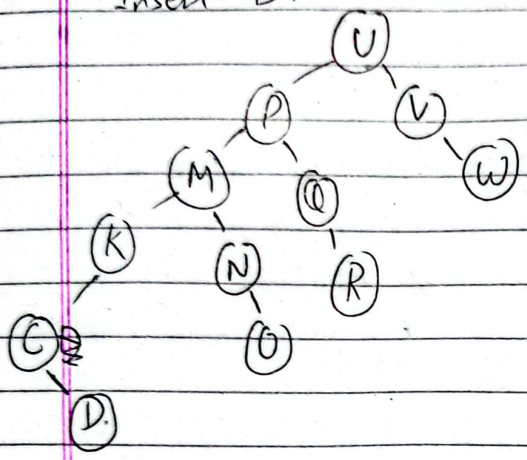


Insert C:



Page No.
Date : / /

Insert D.



Post Order Traversal:

⇒ D, C, K, O, N, M, R, Q, P, W, V, U

Pre order Traversal

⇒ U, P, M, K, C, D, N, O, Q, R, V, W.

- Inorder

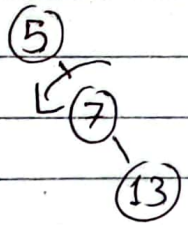
⇒ C, D, K, M, N, O, P, Q, R, U, V, W

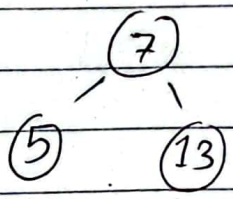
Page No.
Date : / /

Q.25) What is a balanced tree ? Create a AVL tree from
5, 7, 13, 9, 6, 3, 14, 10, 4

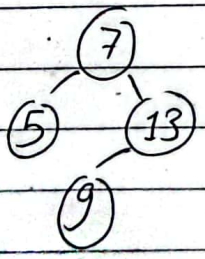
=> A balanced tree is a type of binary tree in which the depth of the left and right subtrees of every node differs by no more than one. This ensures that the tree remains relatively balanced and helps maintain efficient search, insertion & deletion operations.

Insert 5, 7, 13

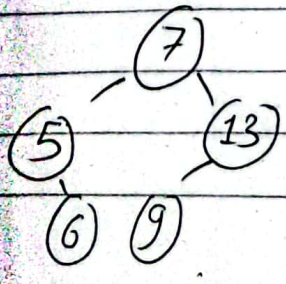




Inserting 9.

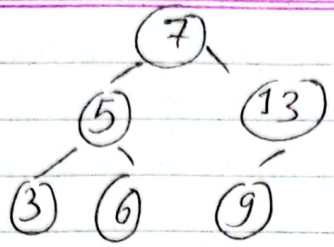


Inserting 6.

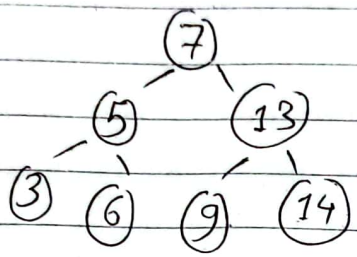


Page No.
Date : / /

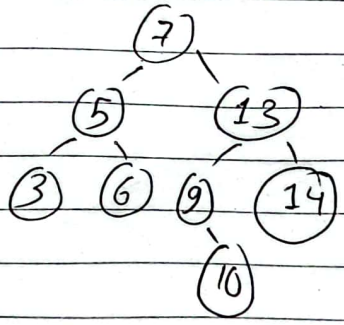
Inserting 3.



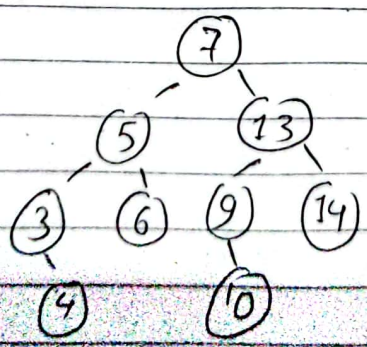
Inserting 14.



Inserting 10.



Inserting 4.



// Answer

Page No.
Date : / /

Q.26) What is B-tree? Create a 3-order & 5 order B-tree following data:
3, 2, 5, 9, 6, 11, 33, 23, 7.

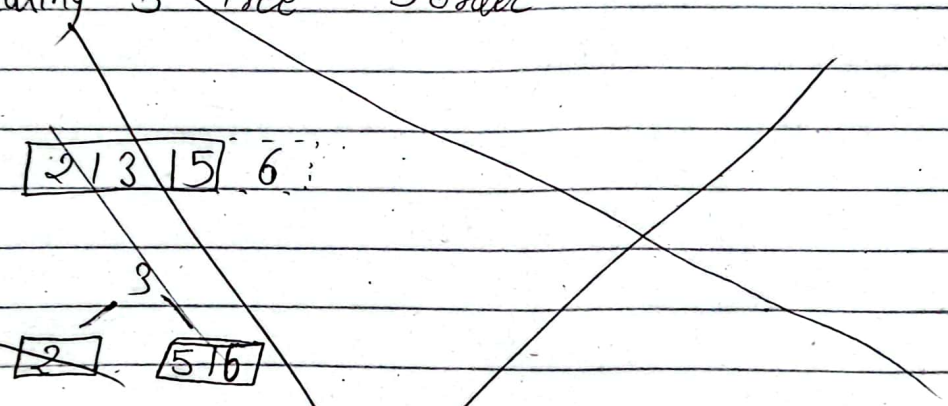
⇒ A B-Tree or Balanced Tree, is a self-balancing tree data structure that maintains sorted data and allows search, insertion & deletion in logarithmic time.

The key feature of a B-tree is its ability to keep the tree balanced during operations, ensuring that the height of tree remains logarithmic with respect to the number of elements in the tree.

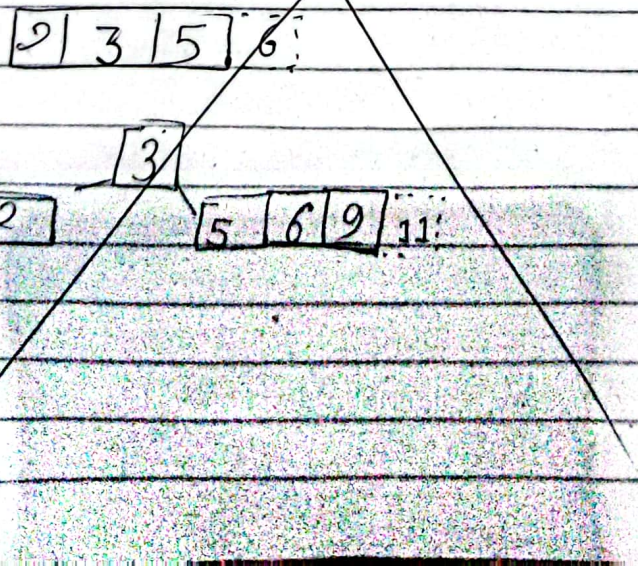
~~Sorting~~ we get: 2, 3, 5, ~~9~~, 6, 9, ~~11~~, 33, 23, ~~7~~

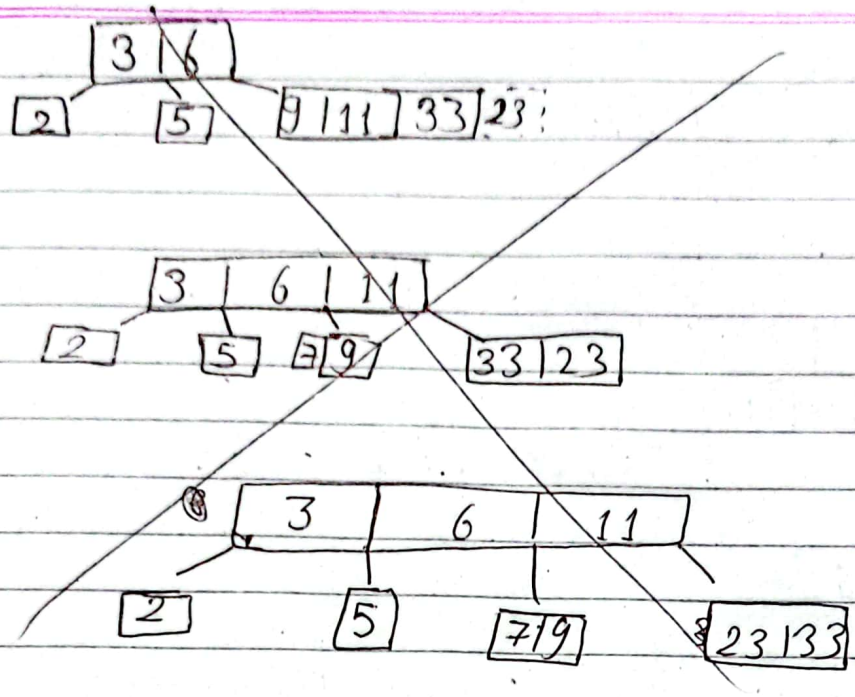
~~6~~, Creating B-Tree 3 order

Inserting:



Inserting:

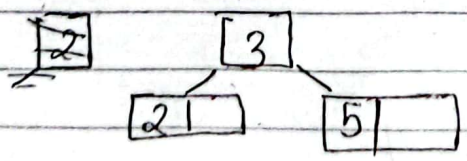




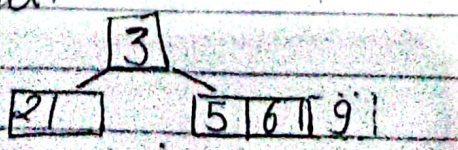
⇒ Order of 3 i.e. a leaf node can have n-1. i.e. 2 children at most.

Inserting 3, 2, 5.

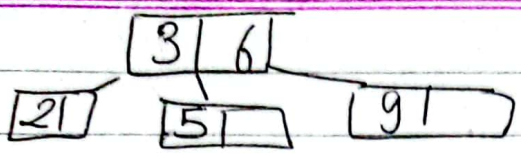
Since leaf can ~~can~~ have only 2 children breaking from 3



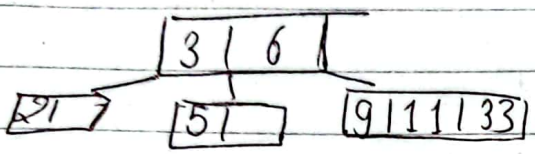
Inserting 9 4 6 we get:



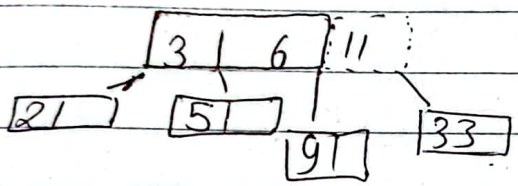
Breaking at 6



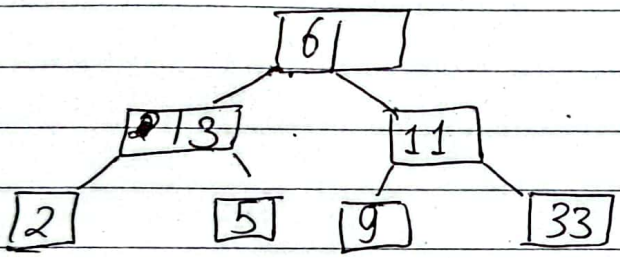
Inserting 11 & 33 we get



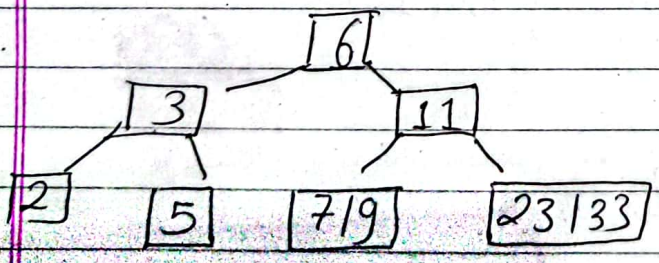
Breaking at 11



Breaking at 6 we get.



Inserting 33 and 7 we get.



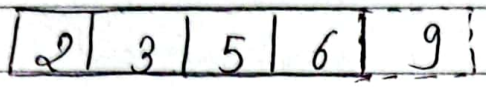
Order of 3.

Page No.
Date : / /

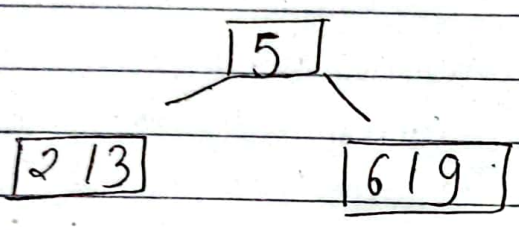
Order of 5

~~Taking~~ Since the leaf can have only n-1 ie. 4 nodes/children

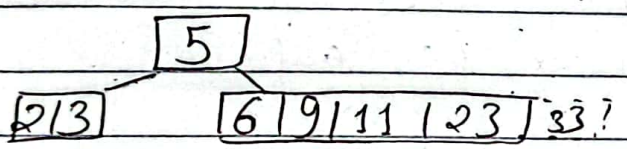
Inserting 3, 2, 5, 9, 6



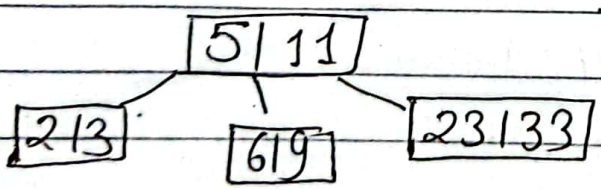
Breaking at 5 we get:



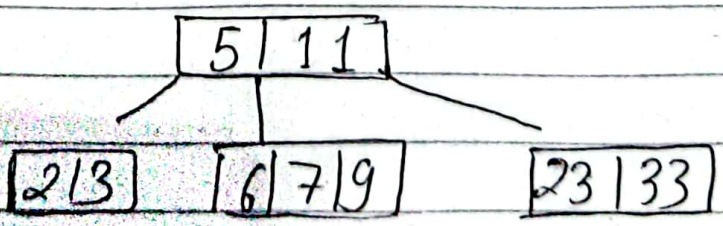
Inserting 11, 33, 23 we get:



Breaking at 11 we get.



Inserting 7 we get



is the order of 5

Page No.
Date : / /

Q.27) Write a short note on Game tree.

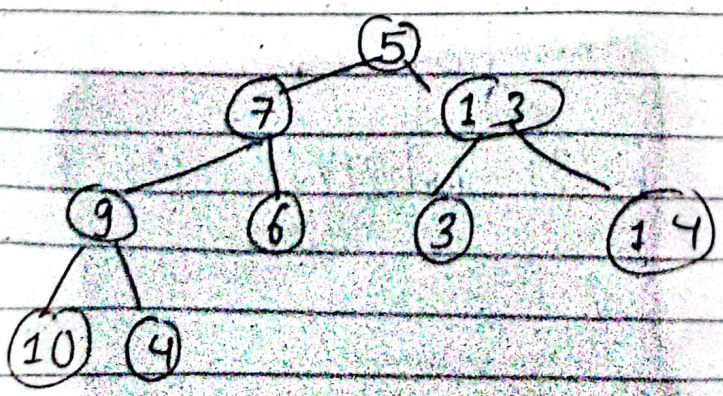
> A game tree is a tree-like structure used to represent the possible moves and outcomes in a sequential, two-player, or multiplayer game. It is commonly employed in algorithmic analysis to determine optimal strategies and outcomes in games of perfect information, where all players have complete knowledge of the game state.

The nodes in a game tree represent different game states, and the edges between nodes correspond to legal moves or transitions between states. The tree is rooted at the initial game state, and each level of the tree represents a turn taken by one of the players. The leaves of the tree correspond to terminal game states, where the game ends, and the associated values represent the outcome or utility for the players.

Q.28) Sort using heap sort:

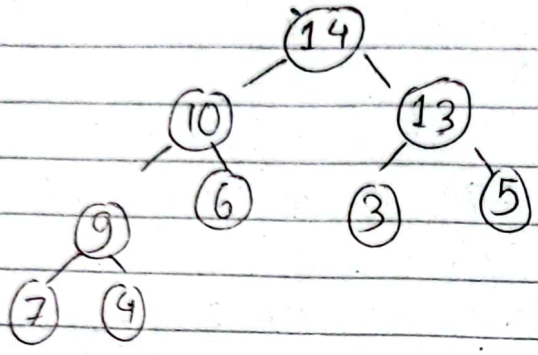
5, 7, 13, 9, 6, 3, 14, 10, 4

Putting in binary tree we get.



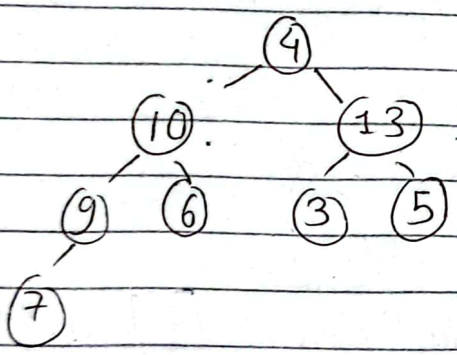
Page No.
Date : / /

Converting to max heapify

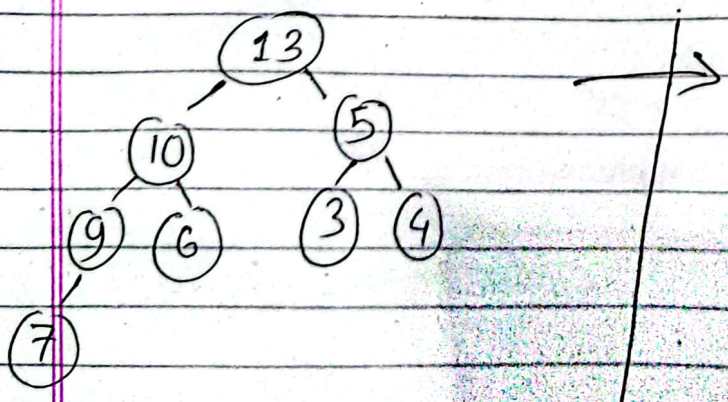


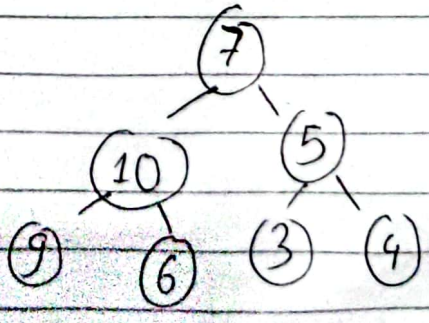
~~Subsht~~ Removing the root and replacing with last node.

Sorted Array [14]



Converting to max heapify. / Removing root & replacing with last node.

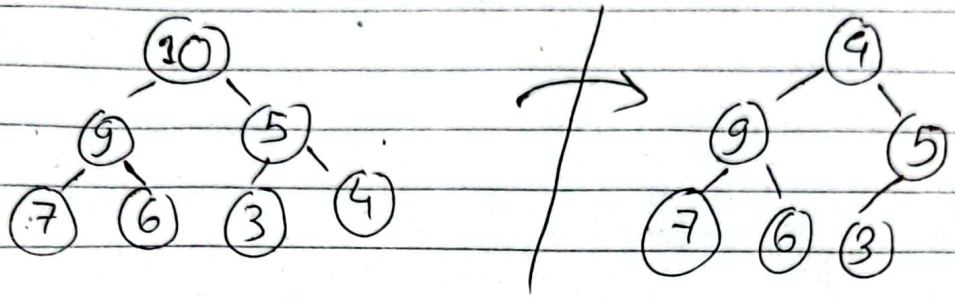




Sorted Array : [13] [14]

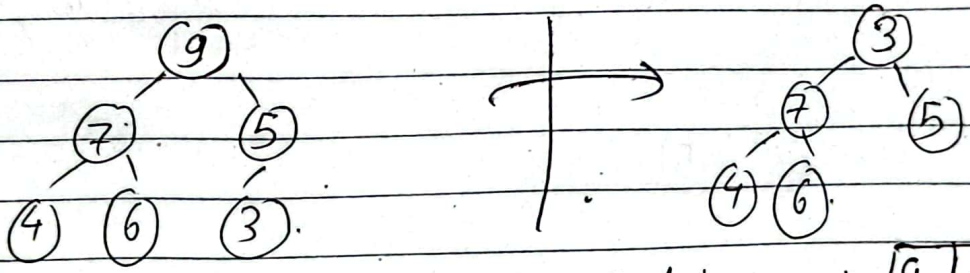
Page No.
Date : / /

Converting to max heapify & replacing ~~node~~ root to last node.



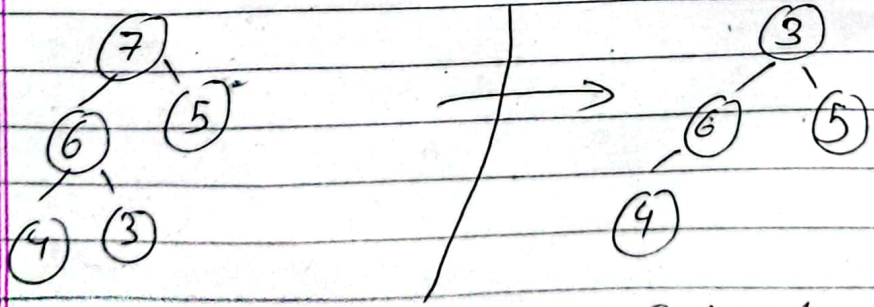
Sorted Array : ~~14~~ | 10 | 13 | 14 |

Converting to max heapify & Replacing root to last node.



Sorted array : | 9 | 10 | 13 | 14 |

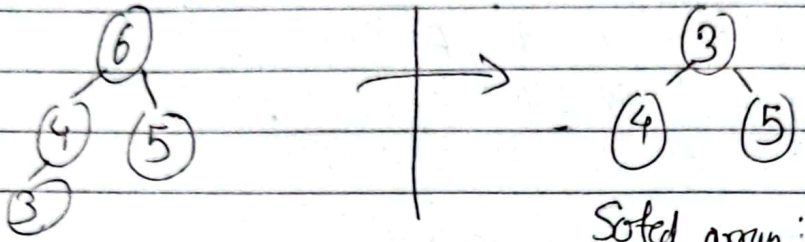
Converting to max heapify & Replacing root to last node



Sorted Array : | 7 | 9 | 10 | 13 | 14 |

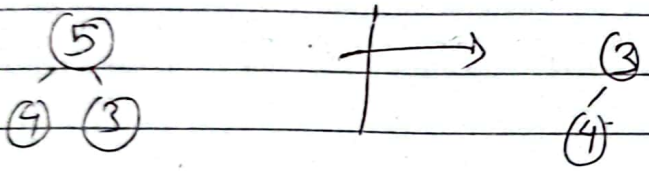
Page No.
Date : / /

Converting to max heapify & root to last.



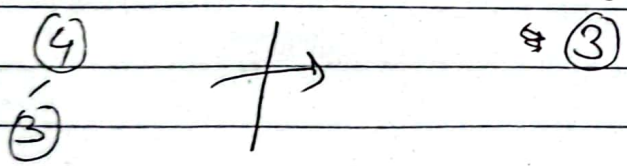
Sorted array: 6 | 7 | 9 | 10 | 13 | 14.

Converting to max heapify & root to last replacing.



5 | 6 | 7 | 9 | 10 | 13 | 14

Converting to max heapify.



∴ Sorted array is: 3 | 4 | 5 | 6 | 7 | 9 | 10 | 13 | 14.

Page No.
Date : / /

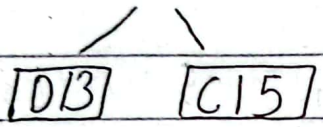
Q.29) For the given frequency of data, create a huffman tree

Item	A	B	C	D	E	F	G
Weight	15	10	5	3	7	12	25

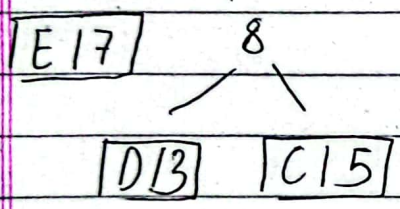
-> Sorting weights in ~~reverse~~ ascending order

D:3 C:5 E:7 B:10 F:12 A:15 G:25

E:7 8 B:10 F:12 A:15 G:25

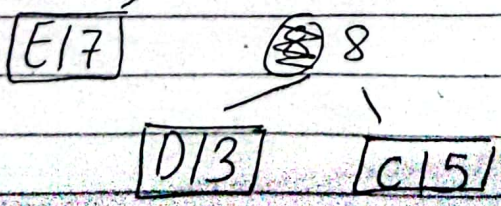


15 B:10 F:12 A:15 G:25

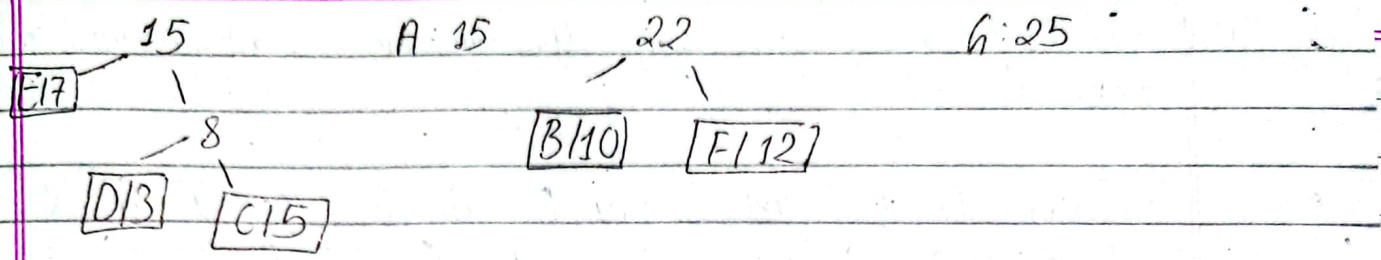


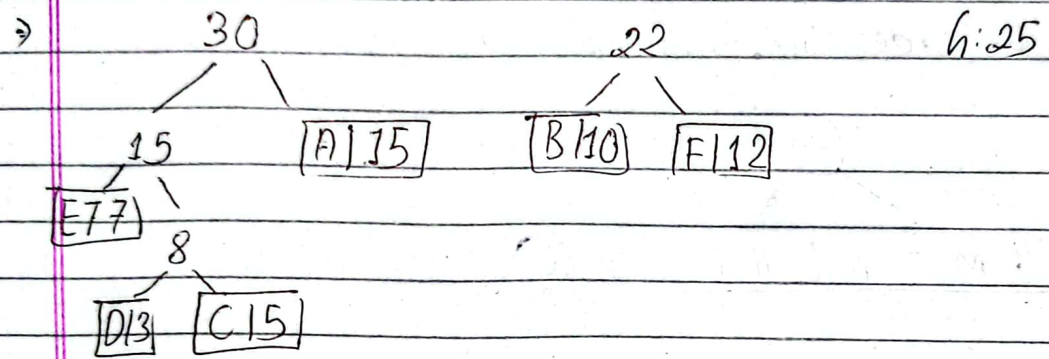
Arranging in ascending

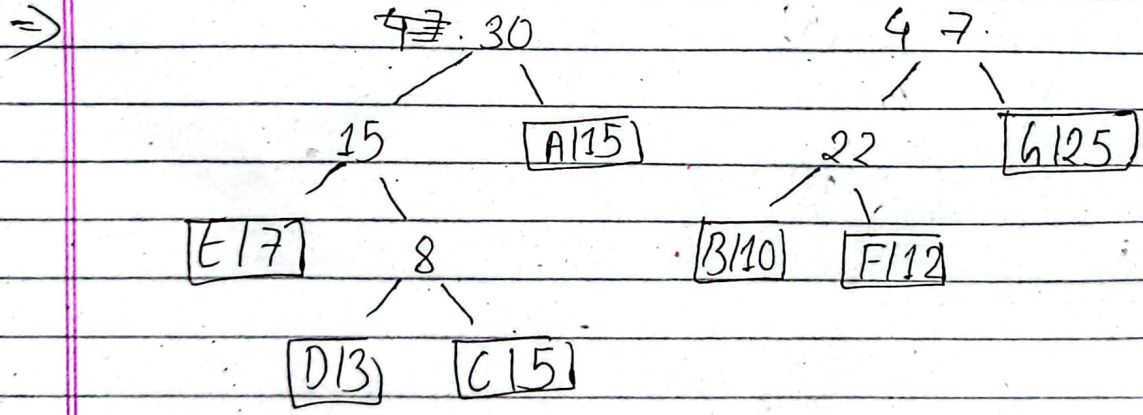
B:10 F:12 15 A:15 G:25

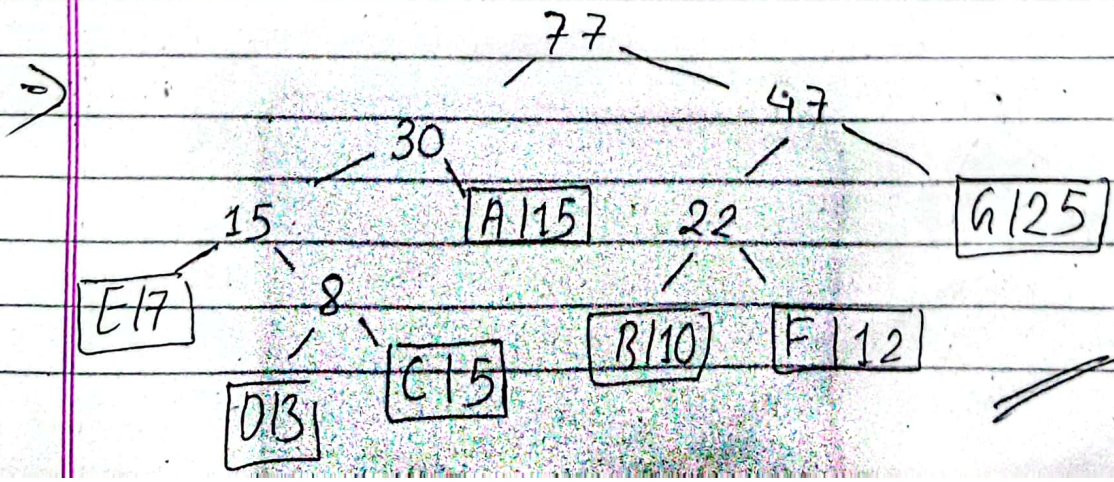


Page No.
Date : / /









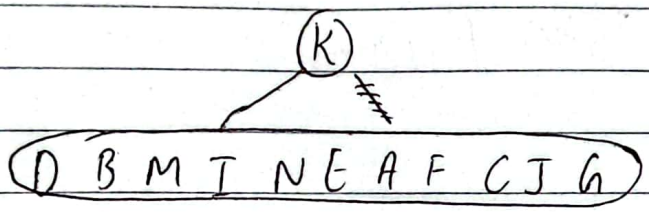
Huffman Tree.

Page No.
Date : / /

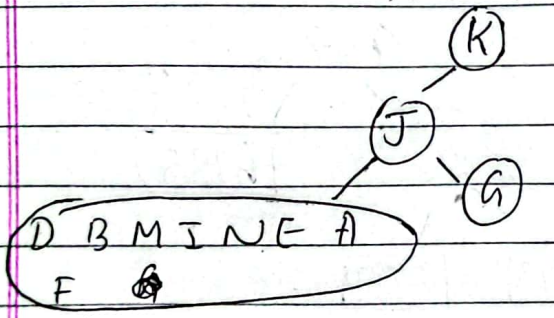
Q. 30) Construct binary tree using the following inorder & post order traversal.

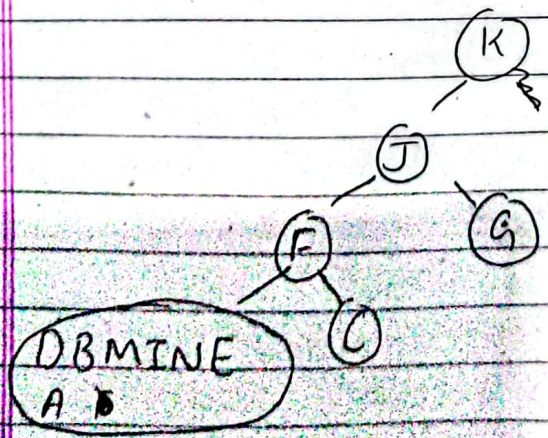
LVR Inorder : D B M I N E A F C J G K
LRV Postorder : A B D E I M N C F G J K

From post order we know root = K i.e.

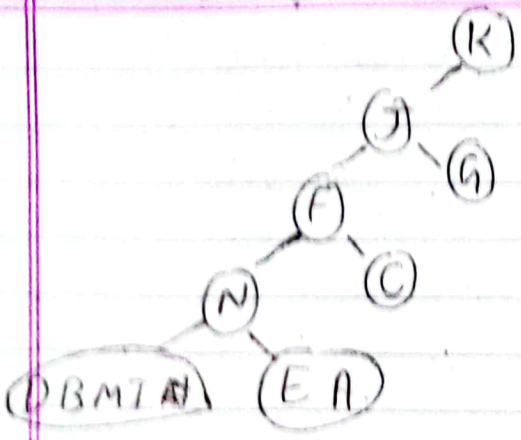


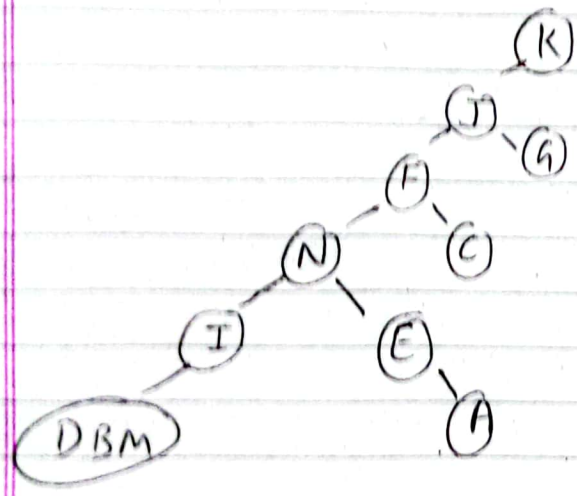
* ~~Now~~ Since these elements are in left
The root of these elements are:

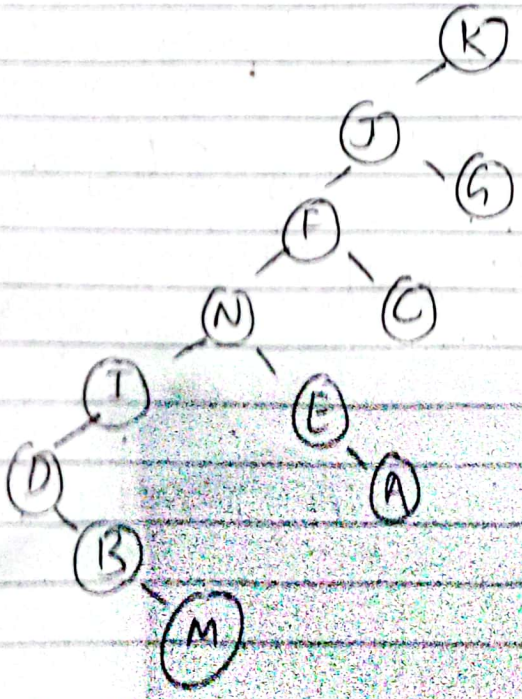




Page No.
Date : / /







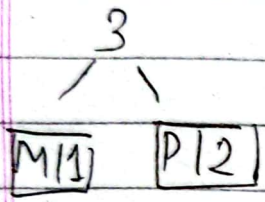
Page No.
Date : / /

Q.31) Construct the ~~tree~~ tree for MISSISSIPPI

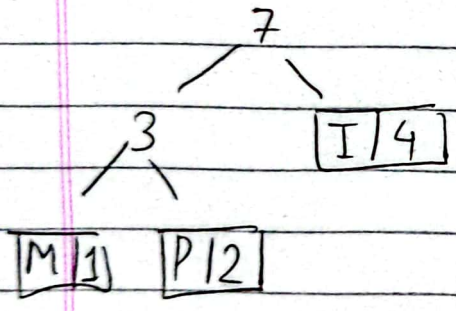
Calculating the frequency of M, S, I, P we get:

M: 1 I: 4 S: 4 P: 2

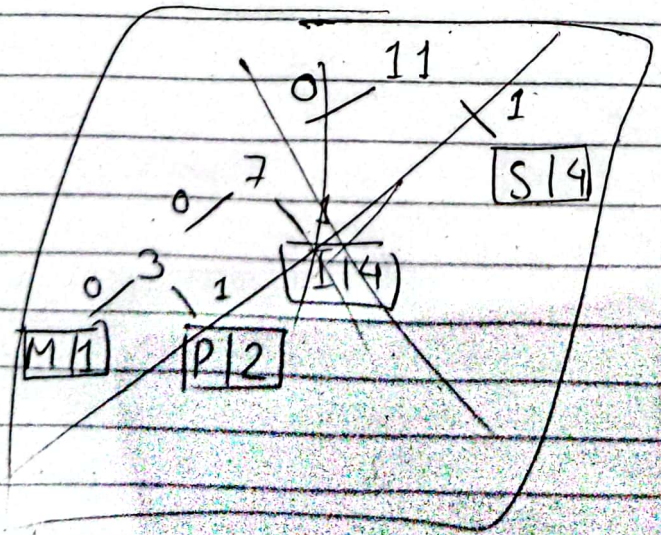
M: 1 P: 2 I: 4 S: 4

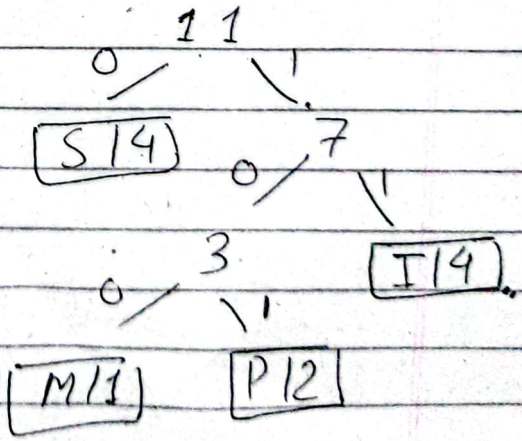


I: 4 S: 4



S: 4





Page No.
Date : / /

Writing Code for M, I, S, P we get.

M : 000	M : 100
P : 001	P : 101
I : ~~01~~ 11	I : 11
S : ~~1~~ 0	S : 0

3, 10,

~~Date~~

(Q.32) Write an algorithm for bubble sort and sort the data using bubble sort

5, 7, 13, 9, 6, 3, 14, 10, 4

i) ⇒ Initialize array arr of ~~sort~~ the unsorted array & n as the no of elements in array

ii) ⇒ The loop runs through the entire list and in each iteration, another nested loop compares adjacent elements and swaps them if in wrong order.

iii) After each pass of outer loop the largest unsorted elements settels at end of list.

iv) This is repeated till sorted ~~elem~~ array is formed.

Page No.
Date : / /

5 | 7 | 13 | 9 | 6 | 3 | 14 | 10 | 4

→ ~~5 7~~ Pass 1:

- Comparing ~~7 swap~~ 5 & 7 since 5 < 7 not swapping

5, 7, 13, 9, 6, 3, 14, 10, 4.

Comparing: 7 & 13 since 13 > 7 not swapping

5, 7, 13, 9, 6, 3, 14, 10, 4

Comparing: 13 & 9. since 13 > 9 swapping.

5, 7, 9, 13, 6, 3, 14, 10, 4.

Comparing 13 & 6 since: 13 > 6 swapping:

5, 7, 9, 6, 13, 3, 14, 10, 4.

Comparing 13 & 3 since 13 > 3 swapping

5, 7, 9, 6, 3, 13, 14, 10, 4.

Comparing 13 & 14 since. 13 < 14 not swapping

→ 5, 7, 9, 6, 3, 13, 14, 10, 4

Comparing 14 & 10 since 14 > 10 swapping

5, 7, 9, 6, 3, 13, 10, 14, 4.

Pass: 5.

5 3 6 7 4 | 9 | 10 | 13 | 14

→ 3 5 6 7 4 | 9 | 10 | 13 | 14

→ 3 5 6 7 4 | 9 | 10 | 13 | 14

→ 3 5 6 7 4 | 9 | 10 | 13 | 14

→ 3 5 6 4 | 7 | 9 | 10 | 13 | 14

Pass 6.

→ 3 5 6 4 | 9 | 10 | 13 | 14

→ 3 5 6 4 | 9 | 10 | 13 | 14

→ 3 5 4 | 6 | 9 | 10 | 13 | 14

Pass 7:

→ | 3 | 4 | 5 | 6 | 9 | 10 | 13 | 14 |

Sorted Array is achieved

Q.33) What is quick sort? Sort the array 35, 15, 40, 1, 60, 20, 55, 25, 50, 20.

→ Quick Sort is highly effective, efficient and widely used sorting algorithm that follows the divide-and-conquer paradigm. It is known for its average-case time complexity of O(nlog n) making it faster on average compared to other sorting algorithm.

Page No.
Date : / /

| 35 | 15 | 90 | 1 | 60 | 20 | 55 | 25 | 50 | 20 |

Let pivot value = 35

Down = 35

Up = 20

| 35 | 15 | 40 | 1 | 60 | 20 | 55 | 25 | 50 | 20 |

⇒ since: d[index] < u[index]. swapping u & d

| 35 | 15 | 20 | 1 | 60 | 20 | 55 | 25 | 50 | 40 |

Swapping u & d.

⇒ | 35 | 15 | 20 | 1 | 25 | 20 | 55 | 60 | 50 | 40 |

Swapping pivot with up we get.

⇒ | 20 | 5 | 20 | 1 | 25 | 35 | 55 | 60 | 50 | 40 |

Taking array left of pivot:

| 20 | 5 | 20 | 1 | 25 |

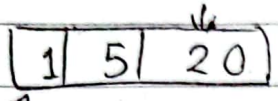
Taking pivot 20 we get.

| 20 | 5 | 20 | 1 | 25 |

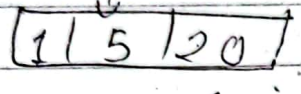
Swapping up & pivot value

| 1 | 5 | 20 | 20 | 25 |

Since there is only one elemt to ~~left~~ right so it is sorted.
So * again taking left array.

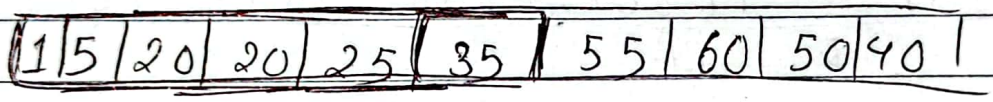
→ 

Taking pivot 5.

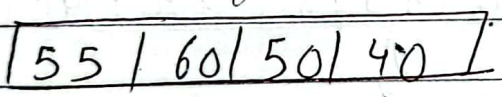


Replacing pivot & up pointer value.

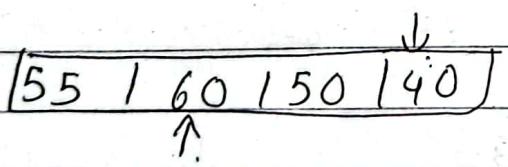
Therefor the left of the 35 is completely sorted and the array becomes:



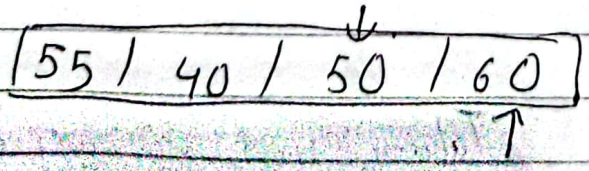
* Taking right of array:



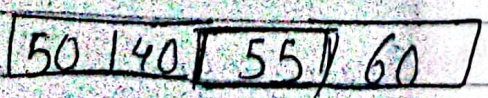
Pivot value = 55



Swaping up & down.



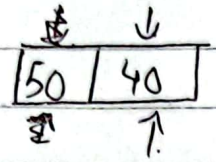
Swapping pivot & up.



Page No.
Date : / /

Taking left of pivot.
i.e. 50 | 40

pivot = 50.



Swapping pivot & up.

∴ 40 | 50

Hence all the array are sorted so combining all array the final sorted array is.

| 1 | 5 | 20 | 20 | 25 | 35 | 40 | 50 | 55 | 60 |

Ans.

Q.34) Differentiate between linear and binary. Write an algorithm for implementing binary search.

→

	Linear Search	Binary Search.
i)	It sequentially checks each element.	i) Divides the search interval in half.
ii)	It works on unsorted list.	ii) It only works sorted list.
iii)	It is suitable for small database.	iii) It is efficient for large dataset.
iv)	O(n) is worst case time complexity.	iv) O(log n) - worst case.

There are ~~typ~~ binary search approach.

1) Iterative Binary Search Algorithm:

i) Sort the array first if not sorted.
ii) Initialize two index high & low where low is the index of first element and high is the index of last element.
iii) Perform avg operation to calculate mid value i.e mid = low + high / 2
iv) if (x == mid) then
    (4.1) Return mid.
v) Else if x < mid data must be on the left so
    high = mid - 1
vi) Else if (x > mid) data must be on right side so
    low = ~~mid++~~ mid + 1;
vii) Repeat from step (3) till answer is found.

2) Recursive Binary Search Algorithm:

i) Sort the array if not sorted.
ii) Initialize two index high & low where low is the is the index of first & last elements
iii) Call a function with parameters array, low, high & value.
iv) Calculate the avg value of the array.
v) If (arr[m] == x)
    return x;
vi) If arr[mid] > x.
    Increase the value of front to mid - 1 and

recursively call itself.

iii) Else

change the value of low to mid + 1 and recursively call itself.

iv) Repeat this till answer is not found.

Q.35) How can you resolve a collision? Using divisive method, hash the given key values in a hash table of size 11 using linear, quadratic and double hashing.

76, 26, 37, 59, 21, 65.

→ We can resolve a collision by 2 methods i.e.

i) Chaining:

In this process the table slot points to a linked list of elements that hash to the same index. When collision occurs new key value is added to linked list

ii) Open Addression:

i) Linear Probing:

When collision occurs it searches for the next available slot.

(ii) Quadratic Probing:

When collision occurs it searches slot but within an interval

(iii) Double Hashing:

A seconday hash function is used to determine the step size between probing.

Page No.
Date : / /

Qudratic Probing :

(Hash (x) + i^2) mod TABLE_SIZE

10	76
9	
8	59
7	
6	
5	37
4	26
3	65
2	
1	
0	21

for 76 = h(x) = 76 %. 11 = 10.

26 = h(x) = 26 %. 11 = 4

37 = h(x) = 37 %. 11 = 4 [collision]

(37+1) %. 11 = 5

59 = h(x) = 59 %. 11 = 4 [collision]

(59+1) %. 11 = 5 [collision]

(59+4) %. 11 = 8

21 = h(x) = 21 %. 11 = 10 [collision]

= (21+1) %. 11 = 0

65 = h(x) = 65 %. 11 = 10 [collison]

= 66 %. 11 = 0 [collision]

= (65+4) %. 11 = 3

Page No.
Date : / /

Linear Probing : $h(x) = (\text{Hash}(x) + i) \text{ MOD TABLE\_SIZE}$

10	76.
9	
8	
7	
6	
5	37
4	26
3	
2	
1	65
0.	21

for 76 = 76 %. 11 = 10.

26 = h(x) = 26 %. 11 = 4.

37 = h(x) = 37 %. 11 = 4 [Collision]

h(x) = (37+1) %. 11 = 5.

59 = h(x) = 59 %. 11 = 4 [Collision]

= (59+1) %. 11 = 5 [Collision]

= (59+2) %. 11 = 6.

21 = h(x) = 21 %. 11 = 10 [Collision]

= 21+1 %. 11 = 0.

65 = h(x) = 65 %. 11 = 10 [Collision]

h(x) = (65+1) %. 11 = 0 [Collision]

h(x) = (65+2) %. 11 = 1

Page No.
Date : / /

Double Hashing.

$h(x) = [h_1(x) + i * h_2(x)] \% TABLE\_SIZE$

where

$h_1(x) =$ Hash(x) % TABBLE_SIZE.

$h_2(x) = R - (x \bmod R)$

where R is prime slightly less than Table size

10	75
9	37.
8	59.
7	~~59~~
6	21.
5	
4	~~26~~
3	65
2	
1	
0	

75. $h(x) = 75 \% 11 = 10$

~~25~~ $26 = h(x) = 26 \% 11 = 4$

$37 = h(x) = 37 \% 4$ [collision]

$= h(x) = 4 + 5 \% 11 = 9.$

$59 = 59 \% 11 = ~~10~~ 4$ [collision]

$h(x) = 4 + ~~34~~ \% 11 = ~~28~~$

$21 = 21 \% 11 = 10$ [collision]

$h(x) = ~~8~~ 10 + 7 \% 11 = 6.$

$65 = h(x) = ~~65~~ 65 \% 11 = 10$ [collision.

$= h(x) = 10 + 5 \% 11 = 4$ [collision]

$= 10 + 10 \% 11 = ~~8~~ 9.$ [collision]

$= 10 + 15 \% 11 = ~~4~~$ ~~[collision]~~

$= 3$