National University of Computer & Emerging Sciences, Karachi
Fall-2021 CS-Department
Final Examination (Sol)
December 30, 2021 Slot: 9:00AM to 12:00Noon

Course Code: CS 2001	**Course Name: Data Structures**
Instructor Name:	**Muhammad Rafi / Dr. Ali Raza/Shahbaz/ M Sohail/ Mubashra Fayyaz**
Student Roll No:	**Section No:**

- Return the question paper.
- Read each question completely before answering it. There are **7 questions and 5 pages.**
- In case of any ambiguity, you may make assumption. But your assumption should not contradict with any statement in the question paper.
- All the answers must be solved according to the sequence given in the question paper.
- Be specific, to the point while coding, logic should be properly commented, and illustrate with diagram where necessary.

Time: 180 minutes. **Max Marks**: 100 points

Arrays and Variants	
Question No. 1	**[Time: 20 Min] [Marks: 10]**

An interval is an ordered pair of (s_i,f_i) where s_i represents starting time and f_i represents final time of an interval i. You are given an instance ItervalArray of DynamicSafeArray<Interval> type. You need to perform the following two tasks:

a. You need to produce an integer array for representing ith smallest interval at the ith position. The SortedIntervals is also an instance of DynamicSafeArray<int>.
b. You need to produce another integer array for representing number of conflicts intervals (any pair of intervals if overlapping) for each sorted interval call it NoOfConflictsInSortedIntervals

For example, if the following array of intervals given to you, the required two arrays would contain the content given below:

IntervalArray = {(2,4), (5,9), (10,12), (3,9), (7,11)}
SortedInterval = {0, 2, 1, 4, 3}
NoOfConflictInSortedIntervals ={ 1, 1, 2, 3, 3}

```
DSA<int> sort(DSA <interval> arr)     {
   for(int i=0; i<arr.length()-1;i++)
        {
        for (int j=0;j<arr.length()-1;j++)
             {   int interval1=arr[j].second-arr[j].first;
                 int interval2=arr[j+1].second-arr[j+1].first;
                 if (interval1>interval2)
                  {    swap(arr[j], arr[j+1);   }

             }
        }
   return arr;
}
```

```
DSA<int> getindexSort(DSA<interval> arr)   {
   DSA<interval> sorted =sort(arr);
   DSA<int> indexessorted[arr.size];
   for(int i=0; i<arr.length() ;i++)
        {    indexessorted[i]=arr[i].index;
        }
   return indexessorted;
}
```

```
DSA<int> conflict(DSA<interval> arr) {
DSA<interval> sorted= sort(arr);
DSA<int> conflicts(arr.length())
   for(int i=0; i<arr.length();i++)
                              { int c=0;
                        for(int j=0;j<arr.length();j++)
             {
                  if (i==j) continue;
                  if(sorted[i].first>sorted[j].first &&
sorted[i].second<= sorted[j].second)
                  {    c++; }
                  else if (sorted[i].first<sorted[j].first &&
sorted[i].second>= second[j].second)   {     c++;     }
             }
        conflicts[sorted[i].index]=c;
   }
return conflicts;
}
```

Linked List and Variants	
Question No. 2	**[Time: 20 Min] [Marks: 10]**

You are given two singly linked lists L1 and L2. Write a function to Delete from list L1, nodes whose positions are to be found in an ordered list L2 possible empty. For instance, if L1 = (A B C D E) and L2 = (2 4 8), then the second and the fourth nodes are to be deleted from list L1 (the eighth node does not exist), and after deletion, L1= (A C E).

```
void deleteIT (list& l1, list& l2) {
     Node* temp1=l1.head;
     Node* temp1=l2.head;
     while (temp2! =NULL)
     {  if(temp2.data>l2.size())    return;
     while (i-1<temp1.next;
     i++;
     }
   Node* todelete=temp1.next;
   temp1.next= todelete. next;
   todelete.next=NULL;
   delete todelete;
   temp2=temp2.next;}
}
```

Stacks and Queues	
Question No. 3	**[Time: 20 Min] [Marks: 10]**

One FASTIAN suggested a very crafty implementation of combined Stack and Queue in an array. He used a DynamicSafeArray suggested during the course to use for it. He called it StackQ. The idea is to keep a queue (FIFO) from the right side of the array, while a stack (LIFO) from the left side of the array. The class implementation for this StackQ is having three indexes, one for top of the stack and two for both front and rear of the queue. The following diagram represents one instance of this data structures at some point in time.

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
12	76	43					15	8	7
		top ^					**rear^**		**front^**

Here is the class definition for some primitive functions for combined stack and queue.

```
template<class T>
class StackQ{

      private:
      DynamicSafeArray<T> *Data;
       unsigned int top;
       unsigned int front;
       unsigned int rear;

      public:

      StackQ();
      StackQ(int size);
      StackQ(constStackQ&rhs);
      StackQ operator=(constStackQ&rhs);
      ~StackQ();
      //Stack Primitive
      void Push(T element);
      void Pop();
      T Peek();
      bool IsfullStack();
      bool isEmptyStack();
      //Queue primitive
      bool IsFullQueue();
      bool IsEmptyQueue();
      void Enqueue(T element);
      void Dequeue();
      T Process();

};
```

Provide some valid implementation of all the primitives of stack and queue in this scenario. Your code must check all necessary conditions to support any primitive operation.

```
//Stack Primitive
     void Push(T element){
        if (top+1! =rear)
        {top++;
         data[top]=element;
        }}
     void Pop(){ if (top > -1) top--;}
     T Peek(){ if (top != -1) return Data[top];}
     bool IsfullStack(){return ((top+1==rear)||(top==size-1));}
     bool isEmptyStack(){ return (top == -1);}
//Queue primitive
     bool IsFullQueue(){ return ((rear==top+1)||(rear==0));}
     bool IsEmptyQueue(){ return (front == rear = size);}
     void Enqueue(T element){
      if((rear-1!=top) || (rear!=0)){
             rear--;
            data[rear]=element;}}
     void Dequeue(){ if ((front < size)&&(front >-1)) front--;}
     T Process(){ return Data[front];}
```

Recursion	
Question No. 4	**[Time: 20 Min] [Marks: 10]**

You are given a string of all unique characters for example PAIR. You need to write a recursive solution to count all possible DeArrange strings. A DeArrange string is a permutation of original string such that no character appears on it original position in the string. For the given example AIRP is a DeArrange string. Clearly identify the base and recursive case. You need to follow the given signature as well.

`int DeArrangeString(int stringLength).`

```
int DeArrangeString(int n)
{
  // Base cases
  if (n == 1) return 0;
  if (n == 2) return 1;
  // DeArrangeString(n) = (n-1)[DeArrangeString(n-1) + DeArrangeString(n-2)]
  return (n - 1) * (DeArrangeString(n - 1) + DeArrangeString(n - 2));
}
```

Sorting/Searching / Hashing	
Question No. 5	**[Time: 20 Min] [Marks: 10]**

a. Consider the situations and suggest the sorting algorithm known to you that is best suited for the given case. [5]

1. An Array of less than 500 integers when it is known that some of the elements of the collection is left shifted(misplaced).

Insertion Sort (as values are left shifted and insertion start the sorting from left will require less swaps).

2. A collection of records having large record size but small key for each record. The collection size is in hundreds.

Selection Sort (since, comparisons of keys are cheap and we only swap once for each item)

3. A collection on which the batches of data is continuously added.

Insertion sort is also very useful.

Merge Sort (We take sorted part as one batch and the incoming batch as another, we will sort the incoming batch first and then merge it with existing one.)

4. A collection in which a class responded with their choice of ice cream flavour from Mango, Vanilla, Pistachio and Butter Crunch.

Counting Sort (since we only have 5 indexes and their count)

5. A large collection of records with variable size.

Quick Sort (since, merge will be expensive for space and quick offers O (n log n) without using extra space).

b. Given an array A of n >= 10000 distinct positive integers. Write an algorithm that will output one element x of A such that x is not among the top 5 elements of A, neither is it among the bottom 5 elements of A. Note that the top and bottom 5 elements of A are the first 5 and the last 5 elements when A is sorted. Your algorithm should not take more than say 50 comparisons. Only comparisons count! All other arithmetic/memory operations are free. [5]

We can use Radix sort for this, as it is non-comparison sorting technique which uses extra space and arithmetic operations. Since, our arithmetic operations and memory are free Radix seems the best and only option. As in Count we will require the max (will take 10,000 comparisons).

Using arithmetic, we can generate an index which will be in range between 5 – 9994.

```
int getElement(int data [ ], int n) {

        register int d, j, k, factor;
        const int radix = 10;
```

```
    const int digits = 10; // the maximum number of digits for an int (2147483647)

    Queue<long> queues[radix]; // integer;

    for (d = 0, factor = 1; d < digits; factor *= radix, d++) {

        for (j = 0; j < n; j++)
            queues[(data[j] / factor) % radix].enqueue(data[j]);

        for (j = k = 0; j < radix; j++)
            while (!queues[j].empty())
                data[k++] = queues[j].dequeue();
    }

    return data [ 4 + ( rand ( ) % 9992 ) ];
}
```

<Note>

Another option will be to use partition method of the quick sort and decide very early.

Trees and Variants	
Question No. 6	**[Time: 50 Min] [Marks: 30]**

a. You are given a pair of Binary Trees with root pointers. Write a function that takes these pointers as input and decide whether they are mirror image of each other's or not. [5]

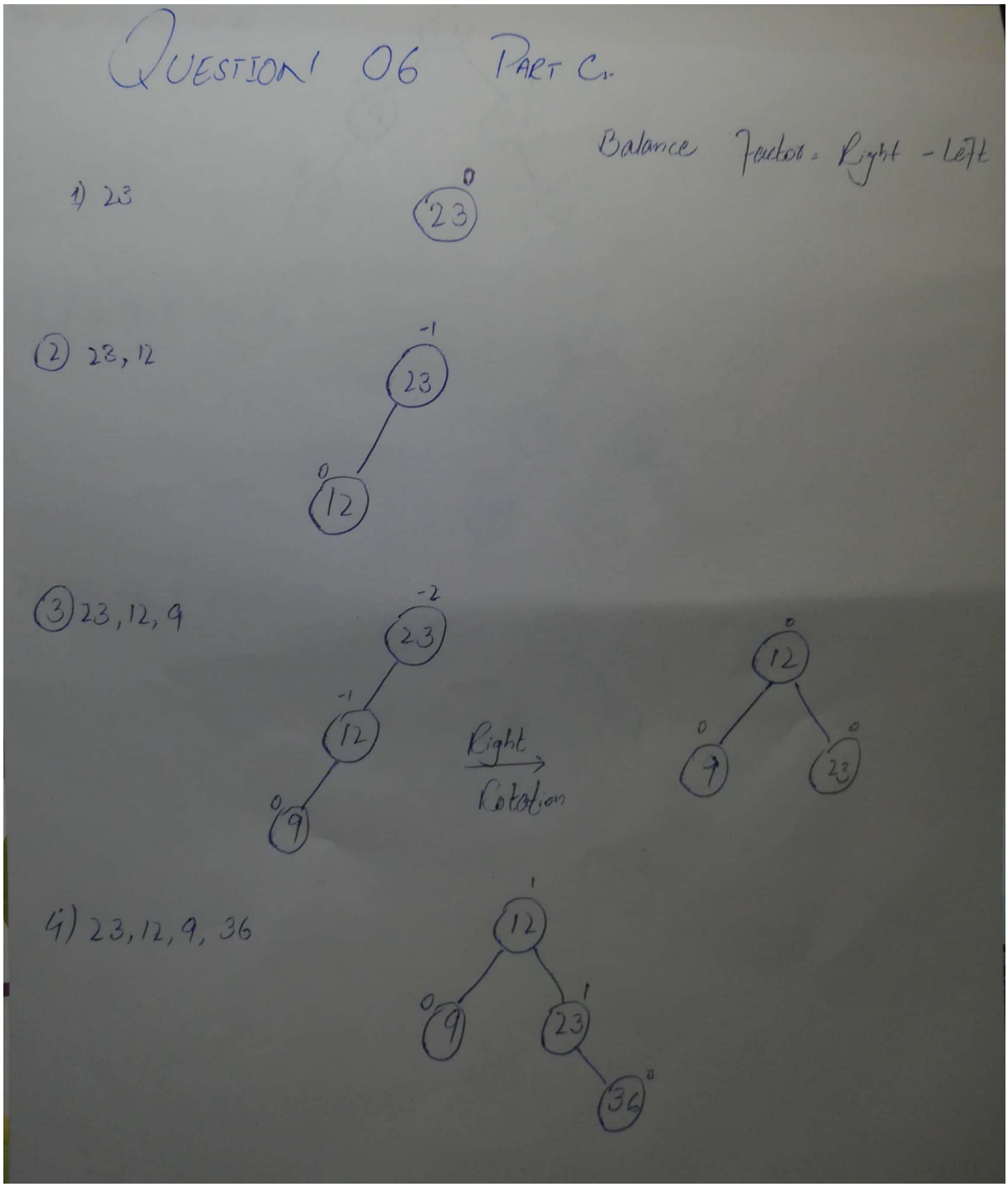
```
bool areMirror(Node* a, Node* b)
{
   /* Base case : Both empty */
   if (a==NULL && b==NULL)
      return true;

   // If only one is empty
   if (a==NULL || b == NULL)
      return false;

   /* Both non-empty, compare them recursively Note that in recursive calls,
      we pass left of one tree and right of other tree */
   return  a->data == b->data &&  areMirror(a->left, b->right) &&
         areMirror(a->right, b->left);
}
```

b. Write a function with the given signature, which take a Binary Search Tree as root pointer and return the second largest element present in the tree. [5]

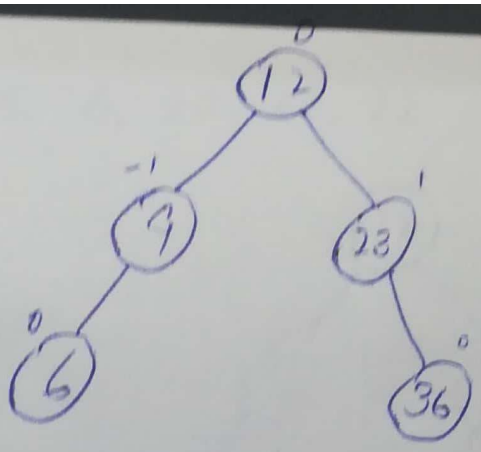
```
T& SecondLargest(BTNode<T> * root)
    BTNode<T> p = root;
    if (p == NULL || p->right == NULL)
        return p;
    for ( ; p-> right ->right != NULL; p = p->right) ;
    return p;
```

c. Insert the following numbers in an empty AVL tree and show each step with a small description of whether a rotation is performed / what sort of rotation is performed. [5]

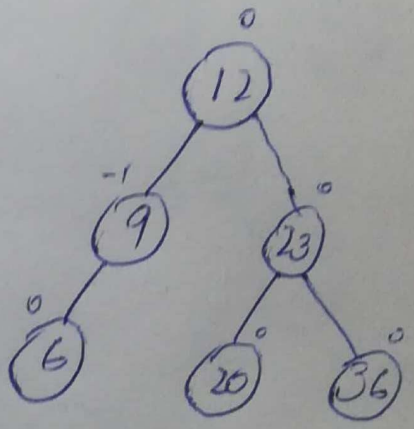
23, 12, 9, 36, 6, 20, 21, 22, 19


QUESTION 06 PART C.
Balance Factor = Right - Left
1) 23
23
0
2) 23, 12
23
12
-1
0
3) 23, 12, 9
23
12
9
-2
-1
0
Right
Rotation
12
9
23
0
0
0
4) 23, 12, 9, 36
12
9
23
36
1
0
1
0

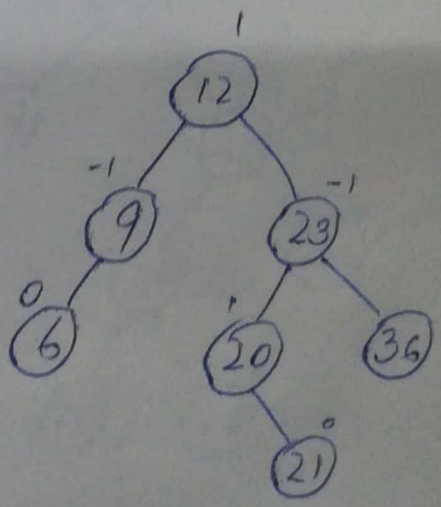
5) 23, 12, 9, 36, 6



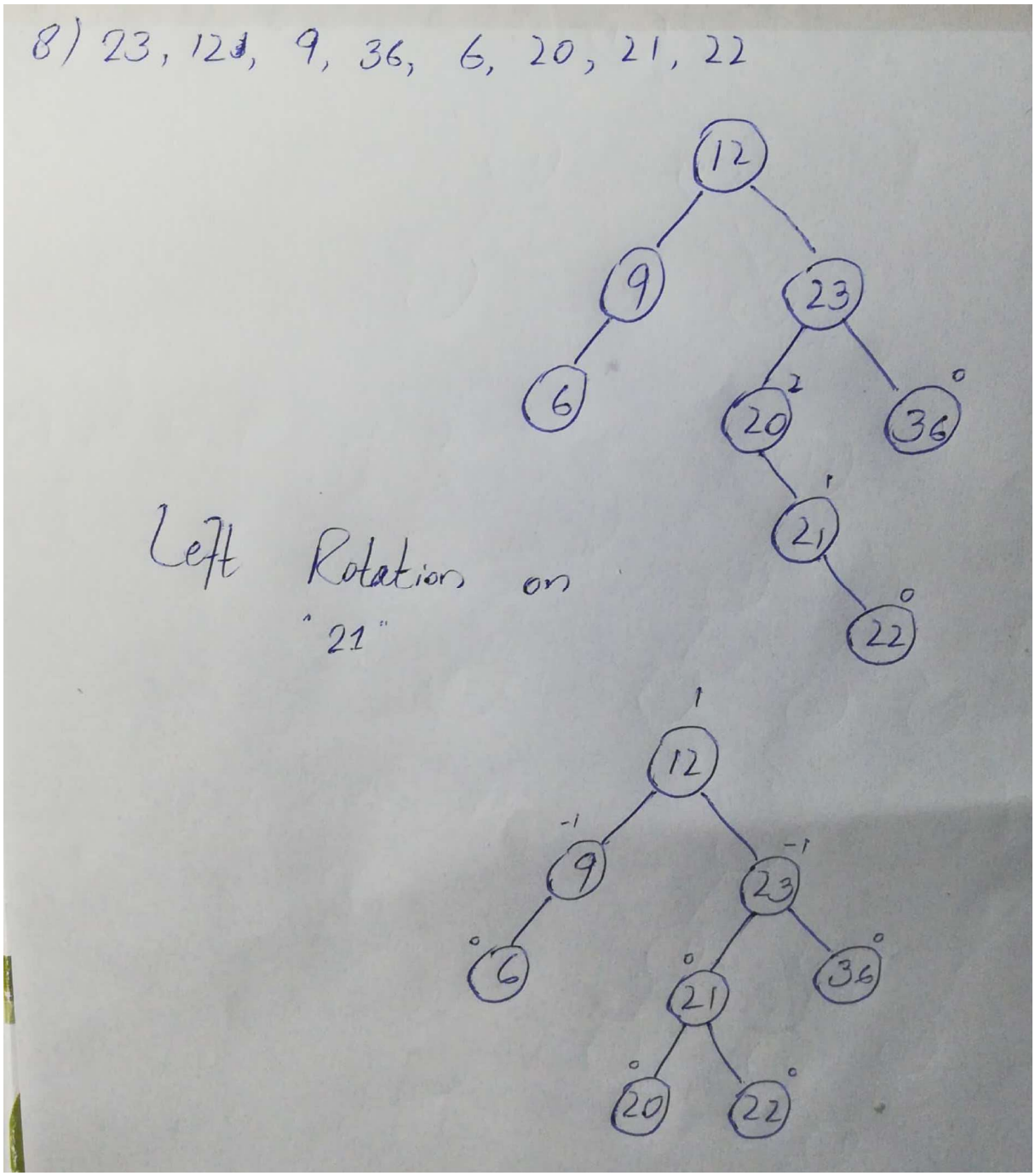
C) 23, 12, 9, 36, 6, 20

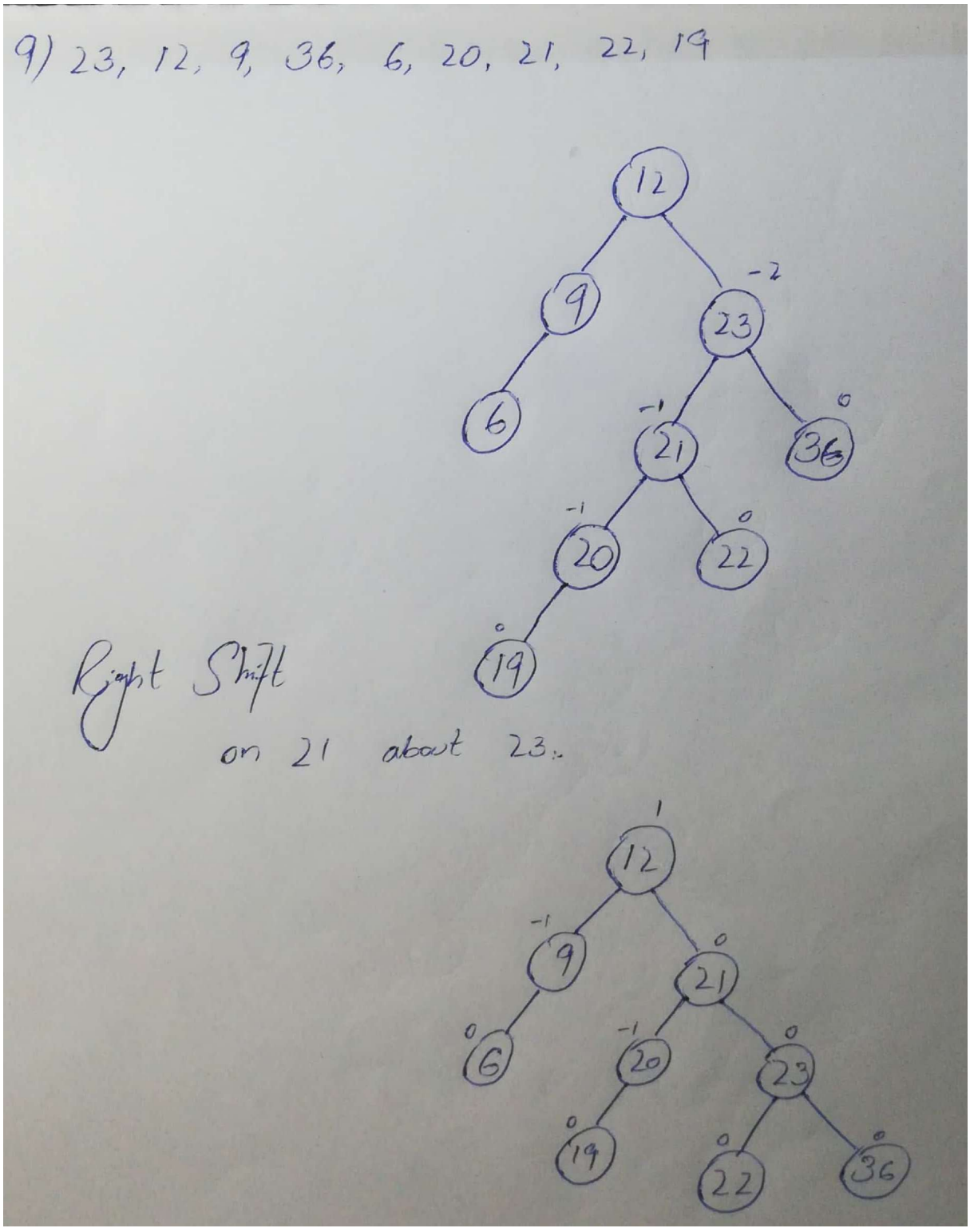


7) 23, 12, 9, 36, 6, 20, 21



8) 23, 12, 9, 36, 6, 20, 21, 22



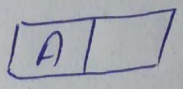


d. Insert the following characters in an empty B-Tree of order 2-3(3-way Tree) Tree and show each step with a small description. [5]

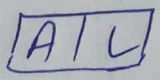
A L G O R I T H M S

B- Tree of Order 3.

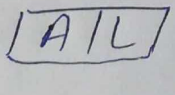
1) A



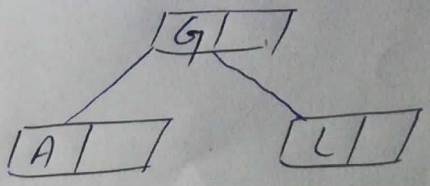
2) AL



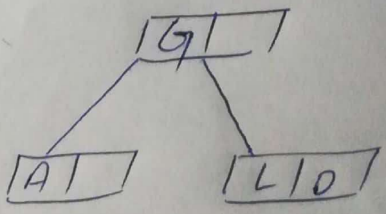
3) ALG



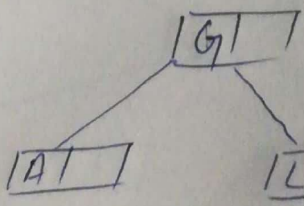
G [A G L]



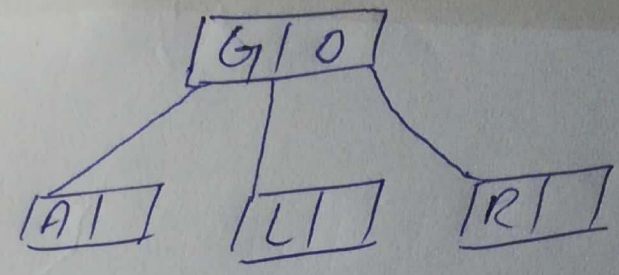
4) ALGO



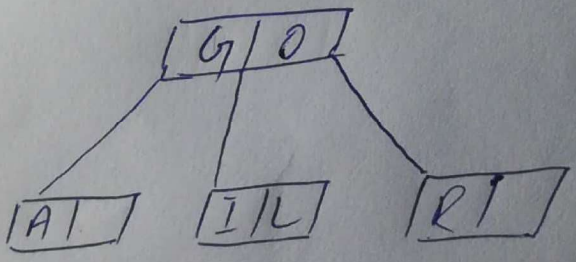
5) ALGOR



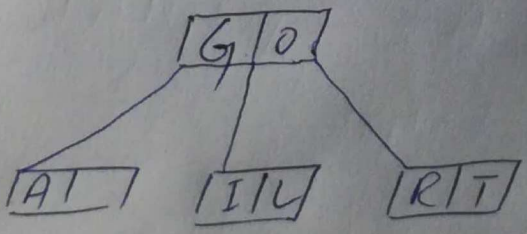
R [L O R]



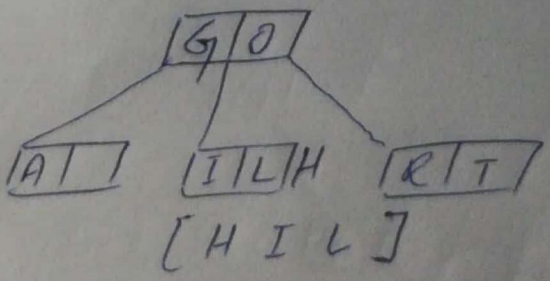
6) ALGORI

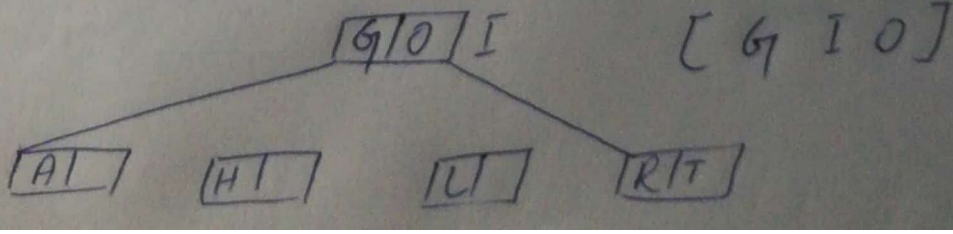


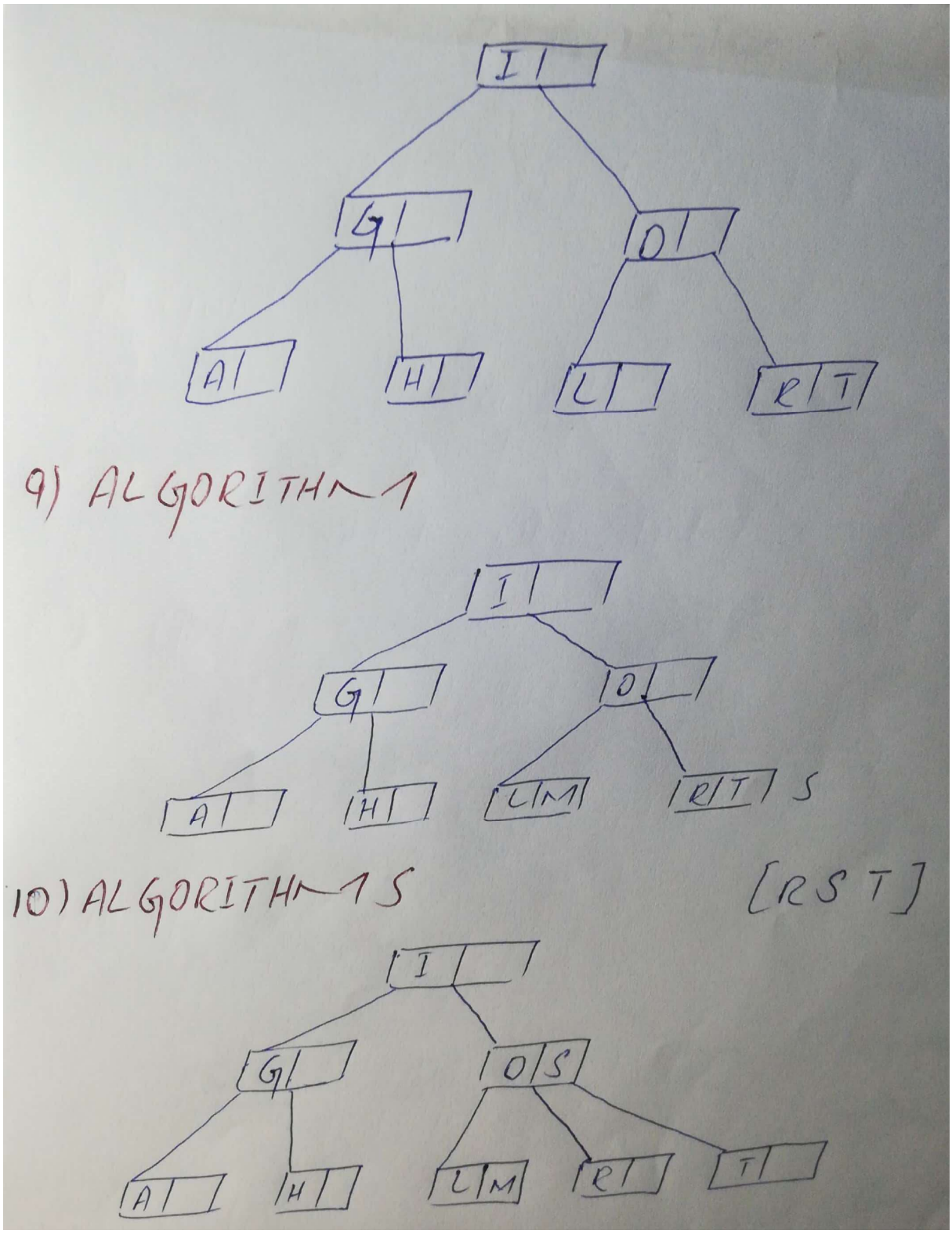
7) ALGORIT



8) ALGORITH

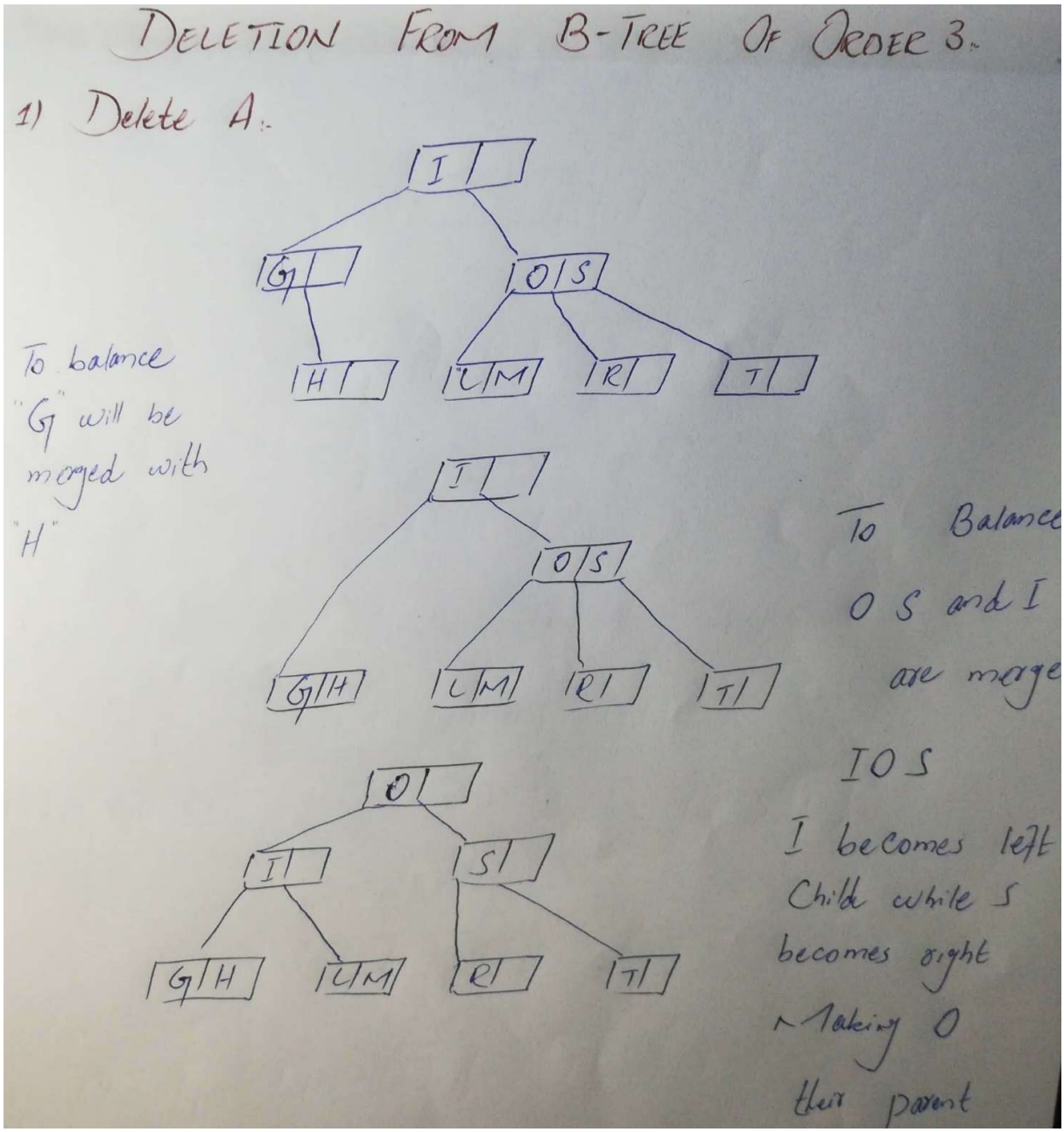




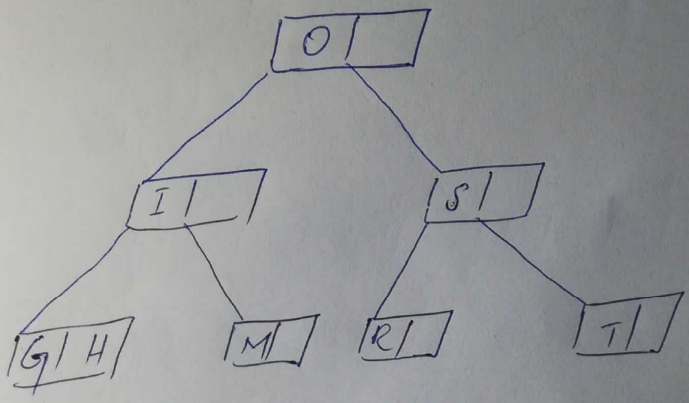


e. From the resultant 2-3 Tree obtained from part(d) above. Show each step after deleting the keys A, L, G and O in the given sequence respectively. [5]

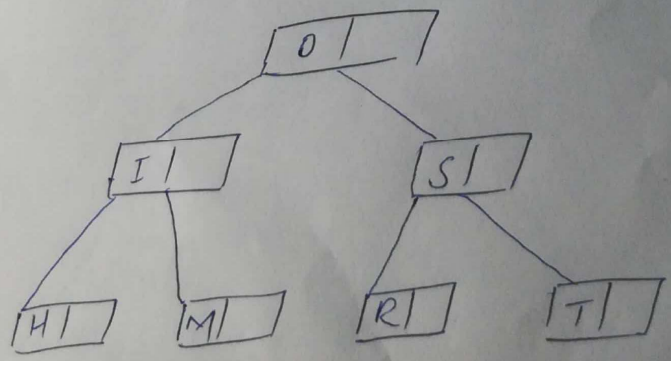
f.


DELETION FROM B-TREE OF ORDER 3.
1) Delete A.
I
G
O S
H
L M
R
T
To balance "G" will be merged with "H"
I
O S
G H
L M
R
T
To Balance
O S and I
are merge
I O S
I becomes left
Child while S
becomes right
taking O
their parent
O
I
S
G H
L M
R
T

2) DELETE L Does not violate the properties

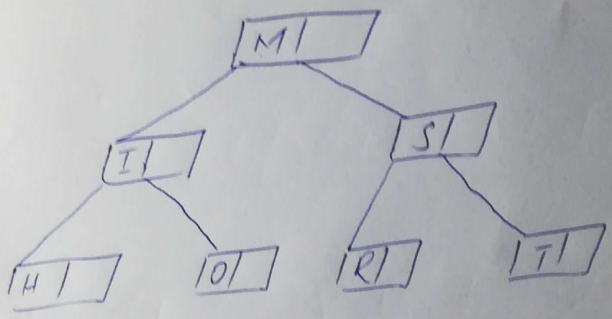


3) DELETE G : Similar to Step 2 :

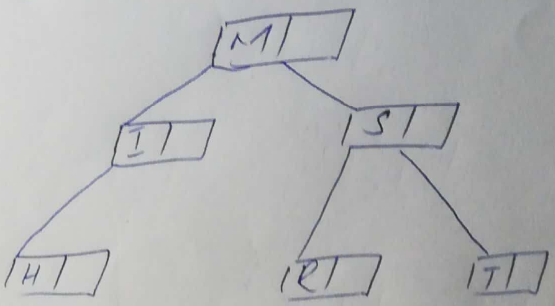


4) DELETE "O"

Since this is internal node, we find its Predecessor and replace the node with its predecessor. which is ^1

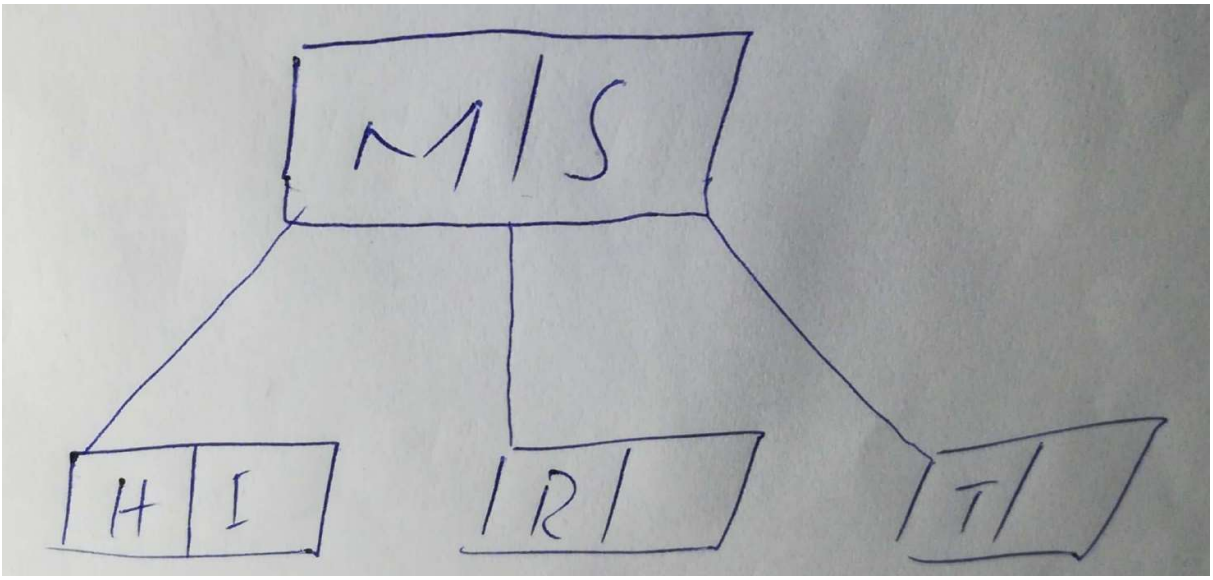


Delete "O"



To Balance H and I will be merged.

Similarly ^1 and S will be merged.



g. Illustrate at least three differences between B-Tree and B+-Trees. [5]

B-Trees	B+ Trees
Data is stored in leaf nodes as well as internal nodes.	Data is stored only in leaf nodes.
Searching is a bit slower as data is stored in internal as well as leaf nodes.	Searching is faster as the data is stored only in the leaf nodes.
Leaf nodes cannot be linked together (does not provide sequential access).	Leaf nodes are linked together to form a linked list (provides sequential access)
Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.

Graphs	
Question No. 7	**[Time: 30 Min] [Marks: 20]**

a. Outline at least three differences between Adjacency List and Adjacency Matrix of graph storage in memory. [5]

Adjacency Matrix	Adjacency List
Uses O(|v|^2) memory	O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex. Thus, overall space complexity is O(|V|+|E|)
It is slow to add a vertex; a complex operation O(|v|^2)	It is fast to add a vertex; compared to Adjacency matrix. Usually, O(1) operation (insert at front or at rear of linked list).
It is slow to delete a vertex; a complex operation O(|v|^2)	It is fast to delete a vertex; compared to Adjacency matrix. Usually, O(|v|) operation (due to traversing the linked list and deleting each edge).
Adding an edge requires setting matrix entry to 1. E.g. matrix[i][j]=1. So O(1) operation.	Adding an edge requires insert from front or insert from rear in the linked list. Also, O(1) operation.
Similarly, to remove set matrix entry to zero. So O(1) operation.	To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges which can be |v|-1. Thus, So, O(|V|) operation.
It is fast to look up and check for presence or absence of a specific edge between any two vertices O(1)	We need traverse through the linked list to check the presence or absence of an edge. So, O(|v|) operation.

b. Stepwise illustrate the Breadth First search path on the given graph below starting from node A. Show how each node is visited and intermediate data during the search. [5]

Solution:
Queue: enqueue from left dequeue from right.
Edges is a list of empty, initially empty.
num represents the order in which vertices are traversed.

1) Visit A (num=1)
2) Enqueue A, Queue={A}
3) Dequeue, A, Queue={}
4) Visit B (num=2)
5) Enqueue B, Queue={B}
6) Edges: { AB }
7) Visit H (num=3)
8) Enqueue H, Queue={H,B}
9) Edges: { AB, AH }
10) Dequeue, B, Queue={H}
11) Visit C (num=4)

12) Enqueue C, Queue={C,H}
13) Edges: { AB, AH, BC }
14) Dequeue, H, Queue={C}
15) Visit G (num=5)
16) Enqueue G, Queue={G,C}
17) Edges: { AB, AH, BC, HG }
18) Visit I (num=6)
19) Enqueue I, Queue={I,G,C}
20) Edges: { AB, AH, BC, HG, HI }
21) Dequeue, C, Queue={I,G}
22) Visit D (num=7)
23) Enqueue D, Queue={D,I,G}
24) Edges: { AB, AH, BC, HG, HI, CD }
25) Visit F (num=8)
26) Enqueue F, Queue={F,D,I,G}
27) Edges: { AB, AH, BC, HG, HI, CD, CF }
28) Dequeue, G, Queue={F,D,I}
29) Dequeue, I, Queue={F,D}
30) Dequeue, D, Queue={F}
31) Visit E (num=9)
32) Enqueue E, Queue={E,F}
33) Edges: { AB, AH, BC, HG, HI, CD, CF, DE }
34) Dequeue, F, Queue={E}
35) Dequeue, E, Queue={}

BFS vertex visit order = { A, B, H, C, G, I, D, F, E }
BFS Path = { AB, AH, BC, HG, HI, CD, CF, DE }

c. Apply Ford's Algorithm on the given graph below and find shortest path from source A to all others, Show each step of the algorithm. [5]

Solution:

Assuming using the following list of edges:

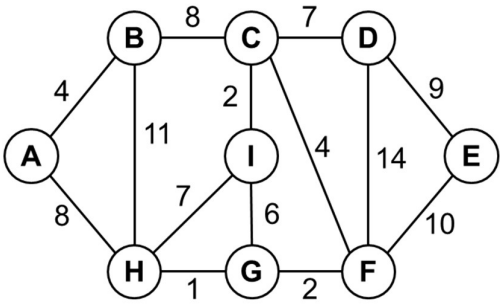
ab, ah, bc, bh, cd, cf, ci, de, df, ef, fg, gh, gi, hi

	Init	Iteration 1	Iteration 2
A	0		
B	Inf	4	
C	Inf	12	
D	Inf	19	
E	Inf	28	
F	Inf	16	
G	Inf	18	
H	Inf	8	
I	Inf	14	

No distance of any vertex changes in iteration 2 so algorithm stops.

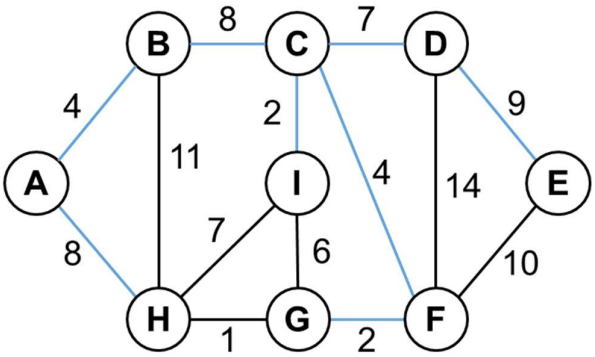
d. Draw the minimum spanning tree that would result from running Dijkstra's algorithm on the following graph. List the order in which edges are added to the tree. Also, mention which edges formed cycles. Use the following sequence of edges in the algorithm. [5]

Edges sequence: ab, ah, bc, bh, cd, cf, ci, de, df, ef, fg, gh, gi, hi



Solution:

Edges list: ab, ah, bc, cd, cf, ci, de, fg,
Edges formed cycle: bh, df, ef, gh, gi, hi



<The End.>