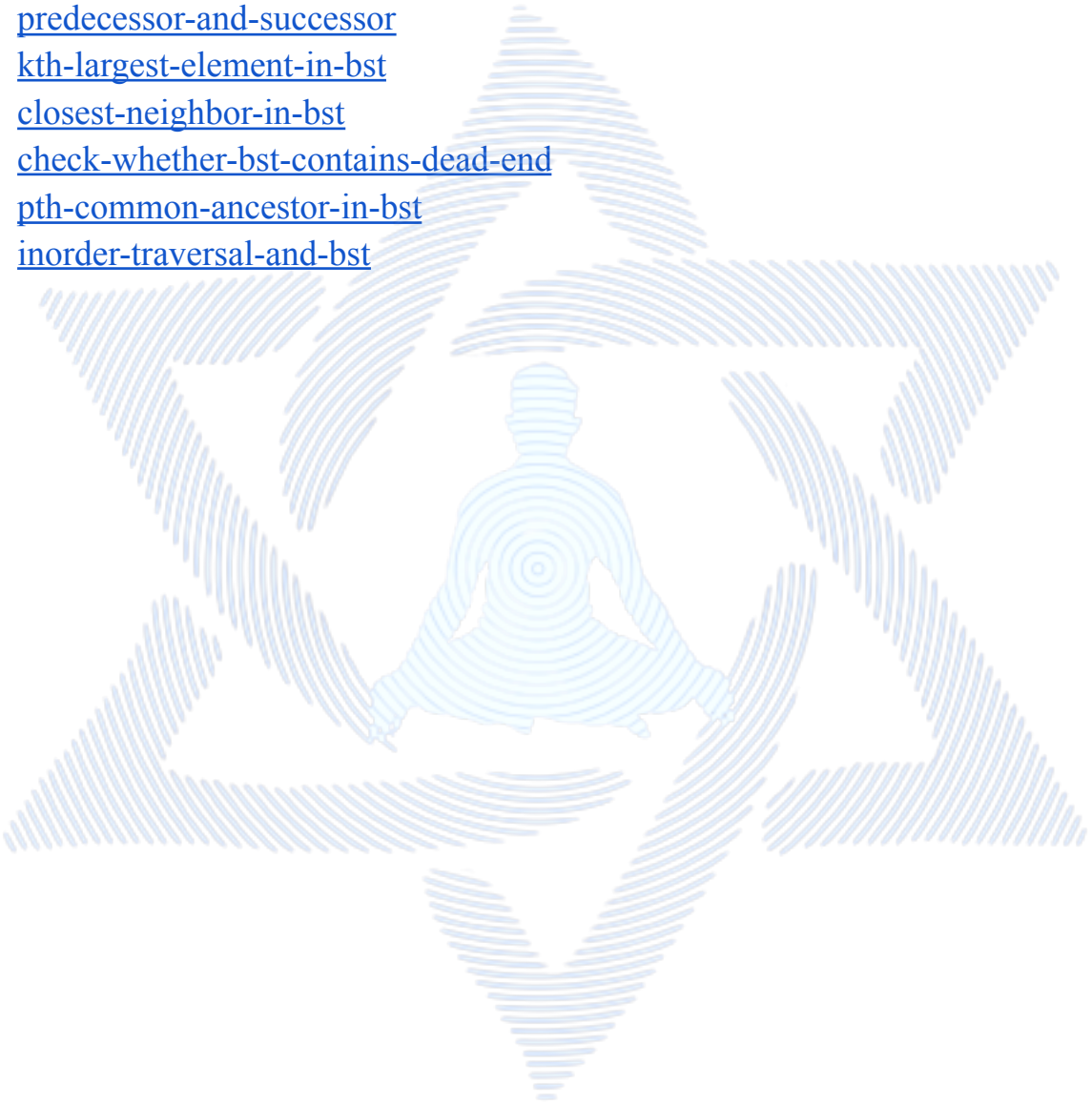
Trees : Level 3
Lesson 2

1. Delete-nodes-greater-than-k
2. predecessor-and-successor
3. kth-largest-element-in-bst
4. closest-neighbor-in-bst
5. check-whether-bst-contains-dead-end
6. pth-common-ancestor-in-bst
7. inorder-traversal-and-bst



Delete-nodes-greater-than-k

Given a **BST** and a **value k**, the task is to delete the nodes having values **greater than or equal to x**.

Example 1:

Input:

```
   4
  / \
 1   9
```

k = 2

Output:

1

Expected Time Complexity: O(Size of tree)

Expected Auxiliary Space: O(1).

Constraints:

1 <= T <= 100 1 <= N <= 103

1 <= A[] <= 103 1 <= x <= N

```
1.  Node* deleteNode(Node* root, int k)
2.  {
3.    if(root)
4.    {
5.      if(root->data>=k)
6.        return deleteNode(root->left,k);
7.      if(root->data<k)
8.      {
9.        root->right=deleteNode(root->right,k);
10.       return root;
11.     }
12.   }
13.   else return NULL;
14. }
```

Predecessor and Successor

There is a BST given with a root node with the key part as integer only.

You need to find the inorder successor and predecessor of a given key.

In case, if either of the predecessor or successor is not found print -1.

Input:

The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case contains n denoting the number of edges of the BST. The next line contains the edges of the BST. The last line contains the key.

Output:

Print the predecessor followed by the successor for the given key. If the predecessor or successor is not found print -1.

Constraints:

1<=T<=100

1<=n<=100

1<=data of node<=100

1<=key<=100

Example:

Input:

2
6
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R
65
6
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R
100

Output:

60 70

80 -1

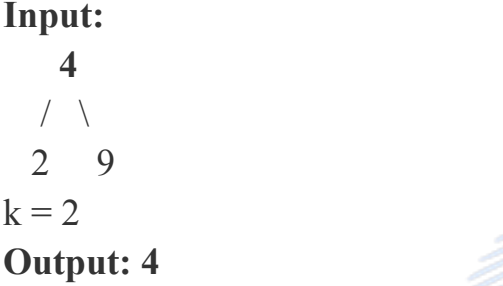
```
1.  void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
2.  {
3.
4.  // Your code goes here
5.    if(!root)
6.       return;
7.    findPreSuc(root->left,pre,suc,key);
8.    if(root->key<key)
9.       pre=root;
10.   else if(root->key>key && !suc)
11.      suc=root;
12.   findPreSuc(root->right,pre,suc,key);
13. }
```

Kth largest element in BST

Given a Binary search tree. Your task is to complete the function which will return the Kth largest element without doing any modification in the Binary Search Tree.

Example 1:

Input:



k = 2

Output: 4

Example 2:

Input:

K = 1

Output: 10

Expected Time Complexity: O(H + K).

Expected Auxiliary Space: O(H)

Constraints:

1 <= N <= 1000

1 <= K <= N

```
1.    int count(Node* root)
2.    {
3.      if(root)
4.          return count(root->left)+count(root->right)+1;
5.      return 0;
6.    }
7.
8.    int kthLargest(Node *root, int K)
9.    {
10.     //Your code here
11.     int r=count(root->right);
12.     if(K==r+1)
13.         return root->data;
14.     else if(K<=r)
15.         return kthLargest(root->right, K);
16.     else
17.         return kthLargest(root->left, K-(r+1));
18.   }
```

Closest Neighbor in BST

Given a binary search tree and a number N, find the greatest number in the binary search tree that is less than or equal to N.

Example 1 :

Input : N = 24
Output : 21
Explanation : The greatest element in the tree which is less than or equal to 24, is 21. (Searching will be like 5->12->21)

Example 2 :

Input : N = 4
Output : 3
Explanation : The greatest element in the tree which is less than or equal to 4, is 3. (Searching will be like 5->2->3)

Expected Time Complexity: O(Height of the BST)

Expected Auxiliary Space: O(Height of the BST).

Constraints:

1<=N<=103

```
1.    int findMaxForN(Node* root, int N)
2.    {
3.       if(!root)
4.           return -1;
5.       if(N==root->key)
```

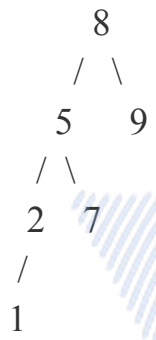
```
6.          return N;
7.      if(N<root->key)
8.          return findMaxForN(root->left,N);
9.      if(N>root->key)
10.     {
11.         if(root->right && root->right->key<=N)
12.             return findMaxForN(root->right,N);
13.         else
14.         {
15.             if(root->right)
16.             {
17.                 int k=findMaxForN(root->right->left,N);
18.                 if(k>root->key)
19.                     return k;
20.             }
21.             return root->key;
22.         }
23.     }
24. }
```

Check whether BST contains Dead End

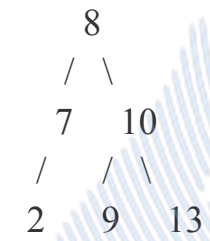
Given a Binary search Tree that contains positive integer values greater then 0. The task is to complete the function **isDeadEnd** which returns true if the BST contains a dead end else returns false. Here Dead End means, we are not able to insert any element after that node.

Examples:

Input :



Output : Yes
Explanation : Node "1" is the dead End because after that we can't insert any element.
Input :



Output : Yes
Explanation : We can't insert any element at node 9.

Input:

The first line of the input contains an integer 'T' denoting the number of test cases. Then 'T' test cases follow. Each test case consists of three lines. First line of each test case contains an integer N denoting the no of nodes of the BST . Second line of each test case consists of 'N' space separated integers denoting the elements of the BST. These elements are inserted into BST in the given order.

Output:

The output for each test case will be 1 if the BST contains a dead end else 0.

Constraints:

1<=T<=100

1<=N<=200

Example(To be used only for expected output):

Input:

2

6

8 5 9 7 2 1

6

8 7 10 9 13 2

Output:

1

1

```
bool isDead(Node* root, int mi, int ma)
{
  if(!root)
    return false;
  if(root->data-1==mi && root->data+1==ma)
    return true;
  return isDead(root->left,mi,root->data) || isDead(root->right,root->data,ma);
}
/*You are required to complete below method */
bool isDeadEnd(Node *root)
{
  //Your code here
  return isDead(root,0,INT_MAX);
}
```

pth common ancestor in BST

Given a Binary Search Tree and two node values x and y present in the BST. Your task is to find the pth (p >= 1) common ancestor of these two nodes x and y.

The 1st common ancestor is the lowest common ancestor. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

All the elements of BST are non-negative and there is no duplicate entry in BST. Also, x and y are unequal.

You are required to complete the function **pthCommonAncestor()**.

Return -1 if the pth common ancestor doesn't exist for a given BST.

Input:

The first line of input contains a single integer T denoting the number of test cases. Then T test cases follow. Each test case consists of three lines. The first line of each test case consists of two integers N and p, denoting the number of elements in the BST and the pth common ancestor to be found, respectively. The second line of each test case consists of N space-separated integers denoting the elements in the BST. The third line of each test case consists of two integers x and y as described in the problem statement.

Output:

You are required to complete the function **pthCommonAncestor()** which takes the root of the tree and two integers x and y(as described in the problem statement) as the arguments. The function returns the value of pth common ancestor to be found.

Constraints:

1 <= T <= 1000

2 <= N <= 1000

1 <= p <= 100

Example:

Input:

2

6 2

50 30 70 60 55 65

55 65

5 2

6 3 8 1 4

1 4

Output:

70

6

```
1.    NODE* lca(int x, int y,NODE *root, vector<int> &p)
2.    {
3.      if(!root)
4.                  return NULL;
5.      p.push_back(root->data);
6.      if((root->data>x && root->data<y) || root->data==x || root->data==y)
7.                  return root;
8.      if(root->data>x)    return lca(x,y,root->left,p);
9.      if(root->data<x)    return lca(x,y,root->right,p);
10.   }
11.   int pthCommonAncestor(int x,int y,NODE *root,int p,vector<int> &vec){
12.     lca(x,y,root,vec);
13.     if(vec.size()>=p)
14.        return vec[vec.size()-p];
15.     return -1;
16.   }
```

Inorder Traversal and BST

Given an array **arr** of size **N**, write a program that returns **1** if array represents Inorder traversal of a BST, else returns **0**.

Note: All keys in BST must be unique.

Example 1:

Input:
N = 3
arr = {2, 4, 5}

Output: 1

Explanation: Given arr representing inorder traversal of a BST.

Example 2:

Input:
N = 3
arr = {2, 4, 1}

Output: 0

Explanation: Given arr is not representing inorder traversal of a BST.

Expected Time Complexity: O(N)

Expected Auxiliary Space: O(1)

Constraints:

$1 \leq N \leq 10^3$

$1 \leq arr[i] \leq 10^3$

```
class Solution{
  public:
  int isRepresentingBST(int arr[], int N)
  {
    // code here
    for(int i=0; i<N-1; i++)
    {
      if(arr[i]>=arr[i+1]) return false;
    }
    return true;
  }
};
```