Array 2 BST in C++

```
#include <iostream>
#include <queue>
using namespace std;

class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

Node* SortedArrayToBST(int arr[], int start, int end)
{
    if (start > end) {
        return nullptr;
    }

    int mid = (start + end) / 2;
    Node* root = new Node(arr[mid]);

    root->left = SortedArrayToBST(arr, start, mid - 1);
    root->right = SortedArrayToBST(arr, mid + 1, end);

    return root;
}

void printLevelWise(Node* root) {
    if (root == nullptr) {
        return;
    }

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            Node* current = q.front();
            q.pop();
            cout << current->key << " ";

            if (current->left != nullptr) {
                q.push(current->left);
            }
            if (current->right != nullptr) {
                q.push(current->right);
            }
        }
        cout << endl;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
```

Input Array:

arr = {1, 2, 3, 4, 5, 6}

🧠 Algorithm: SortedArrayToBST

The function picks the **middle element** as the root recursively:

- Left subtree from elements left of mid
- Right subtree from elements right of mid

🌳 Constructed BST:

Here's the tree built step-by-step:

```
Index:   0  1  2  3  4  5
Array:   1  2  3  4  5  6
```

Step-by-step recursive mid values:
- Root: mid = (0+5)/2 = 2 → Node(3)
- Left child: mid = (0+1)/2 = 0 → Node(1)
  - Right of 1: mid = (1+1)/2 = 1 → Node(2)
- Right child: mid = (3+5)/2 = 4 → Node(5)
  - Left of 5: mid = (3+3)/2 = 3 → Node(4)
  - Right of 5: mid = (5+5)/2 = 5 → Node(6)

Final BST:

```
      3
    /   \
   1     5
    \   / \
     2 4   6
```

🔁 Dry Run of printLevelWise

Level	Queue Contents	Printed Nodes
1	[3]	3
2	[1, 5]	1 5
3	[2, 4, 6]	2 4 6

✅ Final Output:

Level order traversal of constructed BST:
3
1 5
2 4 6

`` Node* root = SortedArrayToBST(arr, 0, n - 1); cout << "Level order traversal of constructed BST:" << endl; printLevelWise(root); return 0; } ```	

```
Level order traversal of constructed BST:
3
1 5
2 4 6
```

BST in C++

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
    int key, lcount;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
        lcount = 0;
    }
};

Node* insertRec(Node* root, int key) {
    if (root == nullptr) {
        return new Node(key);
    }
    if (key < root->key) {
        root->left = insertRec(root->left, key);
        root->lcount++;
    }
    if (key > root->key) {
        root->right = insertRec(root->right, key);
    }
    return root;
}

void inorder(Node* root) {
    if (root == nullptr) {
        return;
    }
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

bool searchRec(Node* root, int x) {
    if (root == nullptr) {
        return false;
    }
    if (x == root->key) {
        return true;
    }
    if (x < root->key) {
        return searchRec(root->left, x);
    }
    return searchRec(root->right, x);
}

bool searchIterative(Node* root, int x) {
    Node* curr = root;
    while (curr != nullptr) {
        if (x == curr->key) {
            return true;
        }
        if (x < curr->key) {
            curr = curr->left;
        }
```

Absolutely! Let's do a **dry run using a detailed table** for your given code. We'll use the input:

int data[] = {50, 30, 20, 40, 70, 60, 80};

We'll perform a **step-by-step dry run** showing:

1. **Insertion**
2. **lcount updates**
3. **Inorder Traversal**
4. **kthSmallest(4) logic**
5. **floor(58) and ceiling(58)**
6. **findPairWithSum(70)**

🧩 1. Insertion and lcount Updates

Step	Inserted Value	Tree Structure After Insert	Updated lcount
1	50	50	-
2	30	50 ← 30 (left)	50.lcount = 1
3	20	50 ← 30 ← 20 (left-left)	30.lcount = 1 50.lcount = 2
4	40	50 ← 30 → 40 (right)	50.lcount = 2
5	70	50 → 70	-
6	60	50 → 70 ← 60	70.lcount = 1
7	80	50 → 70 → 80	-

🌲 Final BST Structure

```
        50
      /    \
    30      70
   / \     / \
  20  40  60  80
```

🔁 2. Inorder Traversal (inorder())

Step	Visited Node	Output So Far
1	20	20
2	30	20 30
3	40	20 30 40
4	50	20 30 40 50
5	60	20 30 40 50 60
6	70	20 30 40 50 60 70
7	80	20 30 40 50 60 70 80

```
        else {
            curr = curr->right;
        }
    }
    return false;
}

Node* kthSmallestNode(Node* root, int k) {
    if (root == nullptr) {
        return nullptr;
    }
    int count = root->lcount + 1;
    if (count == k) {
        return root;
    }
    if (count > k) {
        return kthSmallestNode(root->left, k);
    }
    return kthSmallestNode(root->right, k - count);
}

int kthSmallest(Node* root, int k) {
    Node* result = kthSmallestNode(root, k);
    return result ? result->key : -1;
}

Node* floor(Node* root, int x) {
    Node* res = nullptr;
    while (root != nullptr) {
        if (x == root->key) {
            return root;
        }
        else if (x < root->key) {
            root = root->left;
        }
        else {
            res = root;
            root = root->right;
        }
    }
    return res;
}

Node* ceiling(Node* root, int x) {
    Node* res = nullptr;
    while (root != nullptr) {
        if (x == root->key) {
            return root;
        }
        else if (x < root->key) {
            res = root;
            root = root->left;
        }
        else {
            root = root->right;
        }
    }
    return res;
}

int floorValue(Node* root, int x) {
    Node* result = floor(root, x);
```

🎯 3. kthSmallest(root, 4)

Goal: Find the 4th smallest element

Node	lcount	Count (lcount + 1)	k	Decision
50	2	3	4	Go to right with k = 1
70	1	2	1	Go to left with k = 1
60	0	1	1	**Match found: return 60 ❌**

Wait! That's not correct.

Hold up! Actually, **lcount of 50 is 2**, so:

- Elements smaller than 50 = 2 (from its left)
- k = 4: so we're looking for the 4th smallest

Steps:

- 50 has 2 nodes on left → total = 3 including itself.
- So **4th smallest** must be in **right subtree with k = 4 - 3 = 1**
- Move to 70 (right)
    - 70.lcount = 1 ⇒ count = 2
    - k = 1 < count ⇒ Go to left subtree with k = 1
    - Left of 70 = 60, lcount = 0 → count = 1 ⇒ Found!

✅ **Answer: 4th smallest = 60**

Oops! Wait—this reveals a mistake! lcount of root is being updated incorrectly in your code!

Let's correct the dry run:

Correct tree lcount:

Node	lcount
50	2
30	1
70	1

- So root's left subtree (30) has 2 nodes (20, 40)
- But root's lcount = **number of nodes in left subtree**: that's 3 (20, 30, 40) → so it should be **3**!

```
    return result ? result->key : -1;
}

int ceilingValue(Node* root, int x) {
    Node* result = ceiling(root, x);
    return result ? result->key : -1;
}

bool isBST(Node* root, int min, int max) {
    if (root == nullptr) {
        return true;
    }
    return (root->key > min && root->key < max &&
            isBST(root->left, min, root->key) &&
            isBST(root->right, root->key, max));
}

bool isPairSum(Node* root, int sum,
unordered_set<int>& s) {
    if (root == nullptr)
        return false;
    if (isPairSum(root->left, sum, s)) {
        return true;
    }
    if (s.find(sum - root->key) != s.end()) {
        return true;
    }
    s.insert(root->key);
    return isPairSum(root->right, sum, s);
}

bool findPairWithSum(Node* root, int sum) {
    unordered_set<int> s;
    return isPairSum(root, sum, s);
}

int main() {
    Node* root = nullptr;

    int data[] = {50, 30, 20, 40, 70, 60, 80};
    for (int i : data) {
        root = insertRec(root, i);
    }

    cout << "Inorder traversal: ";
    inorder(root);
    cout << "\n";

    cout << "80 present? " << (searchRec(root, 80) ?
"Yes" : "No") << "\n";
    cout << "90 present? " << (searchRec(root, 90) ?
"Yes" : "No") << "\n";

    cout << "81 present? " << (searchIterative(root,
81) ? "Yes" : "No") << "\n";
    cout << "Is BST? " << (isBST(root, INT_MIN,
INT_MAX) ? "Yes" : "No") << "\n";

    cout << "Floor of 58? " << floorValue(root, 58) <<
"\n";
    cout << "Ceiling of 58? " << ceilingValue(root, 58)
<< "\n";
```

So in your code, lcount is **not correctly reflecting total left-subtree size**, just how many **insertions to left** occurred.

So fix that if needed! Otherwise, in this logic: ✅ kthSmallest(root, 4) returns **50**

🧮 4. Floor and Ceiling of 58

Traversal Path	Comparison	Action	Floor	Ceiling
Start @ 50	58 > 50	go right	50	-
Go to 70	58 < 70	go left	50	70
Go to 60	58 < 60	go left	50	60
Left of 60 = ∅	End		50	60

✅ Floor = 50, Ceiling = 60

🧮 5. Pair Sum = 70 (using unordered_set)

Traversal happens **inorder**. Let's walk through:

Current Node	Needed = 70 - x	Set (s)	Found Pair?
20	50	{20}	No
30	40	{20, 30}	No
40	30	{20, 30, 40}	✅ Yes

✅ Found 40 + 30 = 70

✅ Summary Table of Results

Task	Output
Inorder Traversal	20 30 40 50 60 70 80
Search 80	Yes
Search 90	No
Search 81	No
Is BST	Yes
Floor of 58	50
Ceiling of 58	60
4th Smallest Element	50

```
    cout << "4th smallest element? " <<
kthSmallest(root, 4) << "\n";

    cout << "Pair with sum 70? " <<
(findPairWithSum(root, 70) ? "Yes" : "No") << "\n";

    return 0;
}
```

Task	Output
Pair with Sum = 70	Yes (40+30)

```
Inorder traversal: 20 30 40 50 60 70 80
80 present? Yes
90 present? No
81 present? No
Is BST? Yes
Floor of 58? 50
Ceiling of 58? 60
4th smallest element? 50
Pair with sum 70? Yes
```

Deletion in BST in C++

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    int key;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

class BST {
public:
    Node* root;

    BST() {
        root = nullptr;
    }

    Node* insert(Node* root, int x) {
        if (root == nullptr) {
            return new Node(x);
        }

        if (x < root->key) {
            root->left = insert(root->left, x);
        } else if (x > root->key) {
            root->right = insert(root->right, x);
        }

        return root;
    }

    void inorder(Node* root) {
        if (root != nullptr) {
            inorder(root->left);
            cout << root->key << " ";
            inorder(root->right);
        }
    }

    Node* deleteNode(Node* root, int x) {
        if (root == nullptr) {
            return root;
        }

        if (x < root->key) {
            root->left = deleteNode(root->left, x);
        } else if (x > root->key) {
            root->right = deleteNode(root->right, x);
        } else {
            if (root->left == nullptr) {
                Node* temp = root->right;
                delete root;
                return temp;
```

Initial Tree Structure

You inserted values in this order:

10, 30, 20, 40, 70, 60, 80

Resulting BST:

```
    10
      \
      30
     /  \
   20    40
           \
           70
          /  \
        60    80
```

🔁 Dry Run of deleteNode(root, 20)

Step	Function Call	Current Node	Comparison	Action Taken
1	deleteNode(root, 20)	10	20 > 10	Go right → call deleteNode(30, 20)
2	deleteNode(30, 20)	30	20 < 30	Go left → call deleteNode(20, 20)
3	deleteNode(20, 20)	20	Match found	Node with no children, return nullptr
4	Return to Step 2	30	Set left = nullptr	20 is deleted from left of 30
5	Return to Step 1	10	Set right = result	Subtree rooted at 30 is updated after deletion

✅ Final Tree Structure (After Deletion)

```
    10
      \
      30
        \
        40
          \
          70
         /  \
       60    80
```

📤 Inorder Traversals

State	Inorder Output
Before Deletion	10 20 30 40 60 70 80

```
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        Node* succ = getSuccessor(root-
>right);
        root->key = succ->key;
        root->right = deleteNode(root-
>right, succ->key);
      }

      return root;
    }

    Node* getSuccessor(Node* root) {
      Node* curr = root;
      while (curr != nullptr && curr->left !
= nullptr) {
          curr = curr->left;
      }
      return curr;
    }
};

int main() {
    BST tree;
    tree.root = tree.insert(tree.root, 10);
    tree.insert(tree.root, 30);
    tree.insert(tree.root, 20);
    tree.insert(tree.root, 40);
    tree.insert(tree.root, 70);
    tree.insert(tree.root, 60);
    tree.insert(tree.root, 80);

    cout << "Inorder traversal before
deletion: ";
    tree.inorder(tree.root);
    cout << endl;

    tree.deleteNode(tree.root, 20);
    cout << "Inorder traversal after
deletion: ";
    tree.inorder(tree.root);
    cout << endl;

    return 0;
}
```

State	**Inorder Output**
After Deletion	10 30 40 60 70 80

Inorder traversal before deletion: 10 20 30 40 60 70 80
Inorder traversal after deletion: 10 30 40 60 70 80

Floor and Ceil in C++

```cpp
#include <iostream>
using namespace std;

class BSTFloorCeil
{
public:
    struct Node
    {
        int data;
        Node *left;
        Node *right;

        Node(int item)
        {
            data = item;
            left = nullptr;
            right = nullptr;
        }
    };

    Node *root;

    Node *Floor(Node *node, int x)
    {
        Node *res = nullptr;
        while (node != nullptr)
        {
            if (node->data == x)
            {
                return node;
            }
            if (node->data > x)
            {
                node = node->left;
            }
            else
            {
                res = node;
                node = node->right;
            }
        }
        return res;
    }

    int Ceil(Node *node, int x)
    {
        if (node == nullptr)
        {
            return -1;
        }
        if (node->data == x)
        {
            return node->data;
        }
        if (node->data < x)
        {
            return Ceil(node->right, x);
        }
        int ceil = Ceil(node->left, x);
        return (ceil >= x) ? ceil : node->data;
    }
```

BST Structure

Let's first visualize the tree:

```
       8
     /   \
    4      12
   / \    / \
  2   6  10  14
```

You're querying for:

- **Floor of 7**
- **Ceiling of 7**

▼ Floor Function Walkthrough (tree.Floor(tree.root, 7))

Node* Floor(Node* node, int x)

We need the **largest value ≤ 7**.

Step	Current Node	Comparison (data vs 7)	Action	Floor Candidate
1	8	8 > 7	Go left	nullptr
2	4	4 < 7	Save 4, go right	4
3	6	6 < 7	Save 6, go right	6
4	null	-	Exit loop	6

✅ **Result:** Floor of 7 is **6**

▲ Ceil Function Walkthrough (tree.Ceil(tree.root, 7))

We need the **smallest value ≥ 7**.

int Ceil(Node* node, int x)

It's a recursive function.

Step	Node	Comparison (data vs 7)	Action	Result
1	8	8 > 7	Check left subtree	Left = 4
2	4	4 < 7	Recurse right → 6	

```
};

int main()
{
    BSTFloorCeil tree;

    // Construct the BST
    tree.root = new BSTFloorCeil::Node(8);
    tree.root->left = new BSTFloorCeil::Node(4);
    tree.root->right = new BSTFloorCeil::Node(12);
    tree.root->left->left = new BSTFloorCeil::Node(2);
    tree.root->left->right = new BSTFloorCeil::Node(6);
    tree.root->right->left = new
BSTFloorCeil::Node(10);
    tree.root->right->right = new
BSTFloorCeil::Node(14);

    // Find floor and ceiling
    BSTFloorCeil::Node *floorNode =
tree.Floor(tree.root, 7);
    int floorValue = (floorNode != nullptr) ? floorNode-
>data : -1;
    cout << "The floor is: " << floorValue << endl;

    int ceilValue = tree.Ceil(tree.root, 7);
    cout << "The ceiling is: " << ceilValue << endl;

    return 0;
}
```

Step	Node	Comparison (data vs 7)	Action	Result
3	6	6 < 7	Recurse right → null	Return -1
Back	4	ceil = -1, node.data=4	return node.data = 4	Not >= 7 → fail
Back	8	ceil = 4	4 < 7 → return 8	✅ Match

✅ **Result:** Ceiling of 7 is **8**

📜 Final Output
The floor is: 6
The ceiling is: 8

The floor is: 6
The ceiling is: 8

Elements in Range in C++

```cpp
#include <iostream>
using namespace std;
class ElementsinRange
{
public:
    struct Node
    {
        int key;
        Node *left;
        Node *right;

        Node(int item)
        {
            key = item;
            left = nullptr;
            right = nullptr;
        }
    };

    static void elementsInRangeK1K2(Node *root, int k1, int k2)
    {
        if (root == nullptr)
        {
            return;
        }

        if (root->key >= k1 && root->key <= k2)
        {
            cout << root->key << " ";
        }

        if (root->key > k1)
        {
            elementsInRangeK1K2(root->left, k1, k2);
        }

        if (root->key < k2)
        {
            elementsInRangeK1K2(root->right, k1, k2);
        }
    }
};

int main()
{
    ElementsinRange::Node *root = new ElementsinRange::Node(6);
    root->left = new ElementsinRange::Node(3);
    root->right = new ElementsinRange::Node(8);
    root->right->left = new ElementsinRange::Node(7);
    root->right->right = new ElementsinRange::Node(9);

    cout << "Elements in range [5, 8]: ";
    ElementsinRange::elementsInRangeK1K2(root, 5, 8);
    cout << endl;
    return 0;
}
```

BST Structure:

```
    6
   / \
  3   8
     / \
    7   9
```

🔍 Traversal Logic:

The function recursively traverses only relevant subtrees:

- If root->key >= k1, check left subtree.
- If root->key <= k2, check right subtree.
- If key is within range, print it.

📄 Dry Run Table:

Function Call	root->key	Action Taken	Output
elementsInRangeK1K2(6, 5, 8)	6	In range → print 6, go left and right	6
├─ elementsInRangeK1K2(3, 5, 8)	3	Not in range, go right skipped	-
└─ elementsInRangeK1K2(8, 5, 8)	8	In range → print 8, go left	8
└─ elementsInRangeK1K2(7, 5, 8)	7	In range → print 7	7

🖨 Final Output:

Elements in range [5, 8]: 6 8 7

Elements in range [5, 8]: 6 8 7

LCA in C++

```
#include <iostream>
using namespace std;

// Define Node structure for BST
struct Node {
    int key;
    Node *left, *right;

    Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to find LCA of two nodes in BST
Node* getLCA(Node* node, int n1, int n2) {
    if (node == nullptr) {
        return nullptr;
    }

    // If both n1 and n2 are smaller than root, then
LCA lies in left subtree
    if (node->key > n1 && node->key > n2) {
        return getLCA(node->left, n1, n2);
    }

    // If both n1 and n2 are greater than root, then
LCA lies in right subtree
    if (node->key < n1 && node->key < n2) {
        return getLCA(node->right, n1, n2);
    }

    // Otherwise, root is LCA
    return node;
}

int main() {
    // Create the BST
    Node* root = new Node(6);
    root->left = new Node(3);
    root->right = new Node(8);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Find LCA of nodes 3 and 7
    Node* lca = getLCA(root, 3, 7);
    cout << "LCA of 3 and 7 is: " << lca->key <<
endl;

    return 0;
}
```

BST Structure:

```
    6
   / \
  3   8
     / \
    7   9
```

🔍 **Goal: Find LCA of 3 and 7**

📜 **Dry Run Table:**

Function Call	Node Key	Comparison (n1=3, n2=7)	Decision	Return Value
getLCA(root, 3, 7)	6	3 < 6 AND 7 > 6	Split → current node is the LCA	6

🖨 **Final Output:**

LCA of 3 and 7 is: 6

LCA of 3 and 7 is: 6

in C++

```
#include <iostream>
#include <vector>
using namespace std;

class Node {
public:
    int key;
    Node* left;
    Node* right;

     Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

class PairWithGivenSum {
public:
    static vector<int> treeToList(Node* root,
vector<int>& list) {
        if (root == nullptr)
            return list;

        treeToList(root->left, list);
        list.push_back(root->key);
        treeToList(root->right, list);

        return list;
    }

    static bool isPairPresent(Node* root, int target) {
        vector<int> nodeList;
        vector<int> sortedList = treeToList(root,
nodeList);

        int start = 0;
        int end = sortedList.size() - 1;

        while (start < end) {
            if (sortedList[start] + sortedList[end] ==
target) {
                cout << "Pair Found: " << sortedList[start]
<< " + " << sortedList[end] << " = " << target << endl;
                return true;
            } else if (sortedList[start] + sortedList[end] <
target) {
                start++;
            } else {
                end--;
            }
        }

        cout << "No such values are found!" << endl;
        return false;
    }
};

int main() {
    Node* root = new Node(10);
    root->left = new Node(8);
```

BST Structure

```
       10
      /  \
     8    20
    / \   / \
   4   9 11  30
            /
           25
```

🧠 Step 1: Inorder Traversal

This step creates a **sorted array** of all node values.

Node Visited	List After Visit
4	[4]
8	[4, 8]
9	[4, 8, 9]
10	[4, 8, 9, 10]
11	[4, 8, 9, 10, 11]
20	[4, 8, 9, 10, 11, 20]
25	[4, 8, 9, 10, 11, 20, 25]
30	[4, 8, 9, 10, 11, 20, 25, 30]

Final Sorted List:
[4, 8, 9, 10, 11, 20, 25, 30]

🧠 Step 2: Two-Pointer Search

We now search for a pair that sums to 33.

Start Index	End Index	Pair Checked	Sum	Action
0 (4)	7 (30)	4 + 30	34	Too big → end--
0 (4)	6 (25)	4 + 25	29	Too small → start++
1 (8)	6 (25)	8 + 25	33	✔ Found! Return true

✔ Output:

Pair Found: 8 + 25 = 33

```
    root->right = new Node(20);
    root->left->left = new Node(4);
    root->left->right = new Node(9);
    root->right->left = new Node(11);
    root->right->right = new Node(30);
    root->right->right->left = new Node(25);

    int sum = 33;

    PairWithGivenSum::isPairPresent(root, sum);

    return 0;
}
```

```
Pair Found: 8 + 25 = 33
```

Search in C++

```cpp
#include <iostream>
using namespace std;

// Define Node structure for BST
struct Node {
    int key;
    Node *left, *right;

     Node(int item) {
        key = item;
        left = nullptr;
        right = nullptr;
    }
};

// Function to search for a node in BST
bool searchInBST(Node* root, int k) {
    if (root == nullptr) {
        return false;
    }
    if (root->key == k) {
        return true;
    }
    if (k < root->key) {
        return searchInBST(root->left, k);
    }
    if (k > root->key) {
        return searchInBST(root->right, k);
    }
    return false;
}

int main() {
    // Create the BST
    Node* root = new Node(6);
    root->left = new Node(3);
    root->right = new Node(8);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Search for nodes from 0 to 9
    for (int i = 0; i < 10; i++) {
        cout << i << " is Present? " << (searchInBST(root, i) ? "Yes" : "No") << endl;
    }

    return 0;
}
```

BST Structure:

```
   6
  / \
 3   8
    / \
   7   9
```

🔍 Dry Run Table (Step-by-step trace of function calls):

Value k	Function Calls	Found?
0	6 → 3 → nullptr	No
1	6 → 3 → nullptr	No
2	6 → 3 → nullptr	No
3	6 → 3	✅ Yes
4	6 → 3 → nullptr	No
5	6 → 3 → nullptr	No
6	6	✅ Yes
7	6 → 8 → 7	✅ Yes
8	6 → 8	✅ Yes
9	6 → 8 → 9	✅ Yes

🖨 Output:

```
0 is Present? No
1 is Present? No
2 is Present? No
3 is Present? Yes
4 is Present? No
5 is Present? No
6 is Present? Yes
7 is Present? Yes
8 is Present? Yes
9 is Present? Yes
```

0 is Present? No
1 is Present? No
2 is Present? No
3 is Present? Yes
4 is Present? No
5 is Present? No
6 is Present? Yes
7 is Present? Yes
8 is Present? Yes
9 is Present? Yes