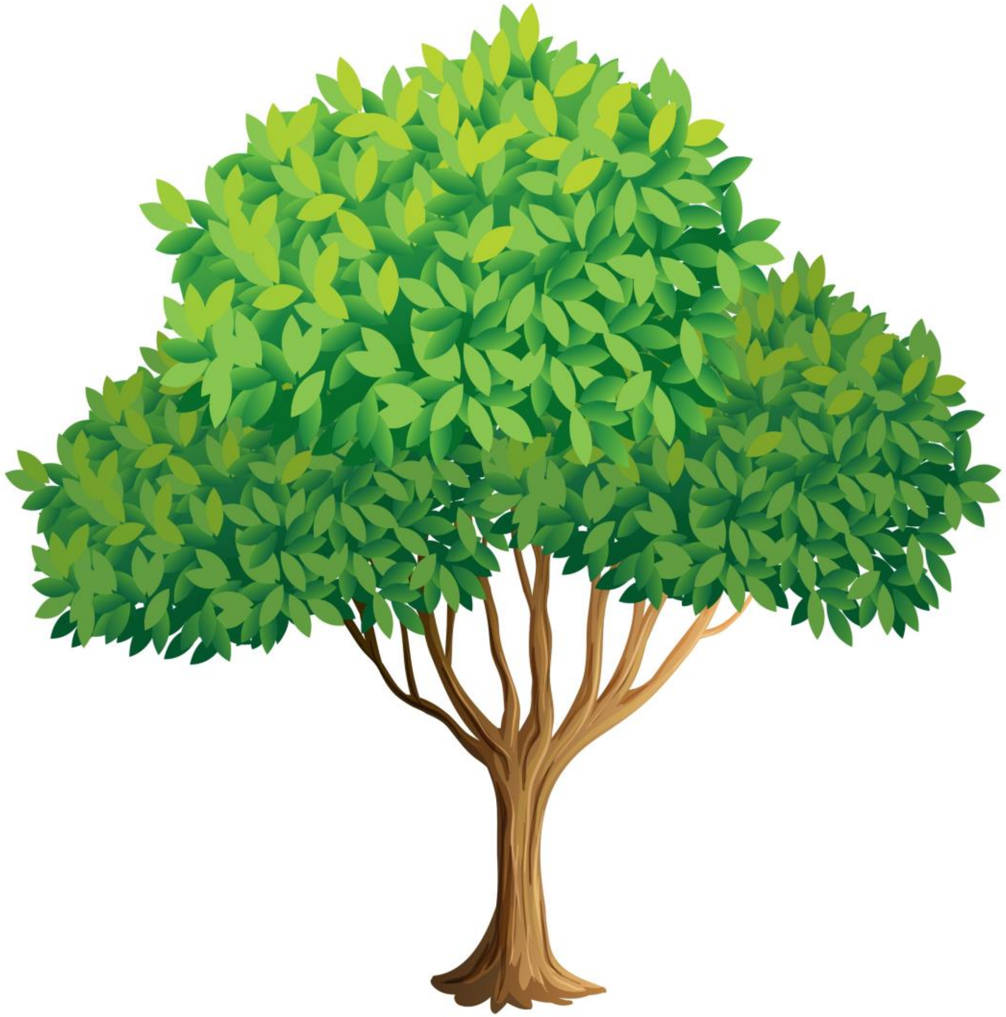


CSCI2100C Data Structures

Tutorial 09 – Trees

LI Muzhi 李木之

mzli@cse.cuhk.edu.hk

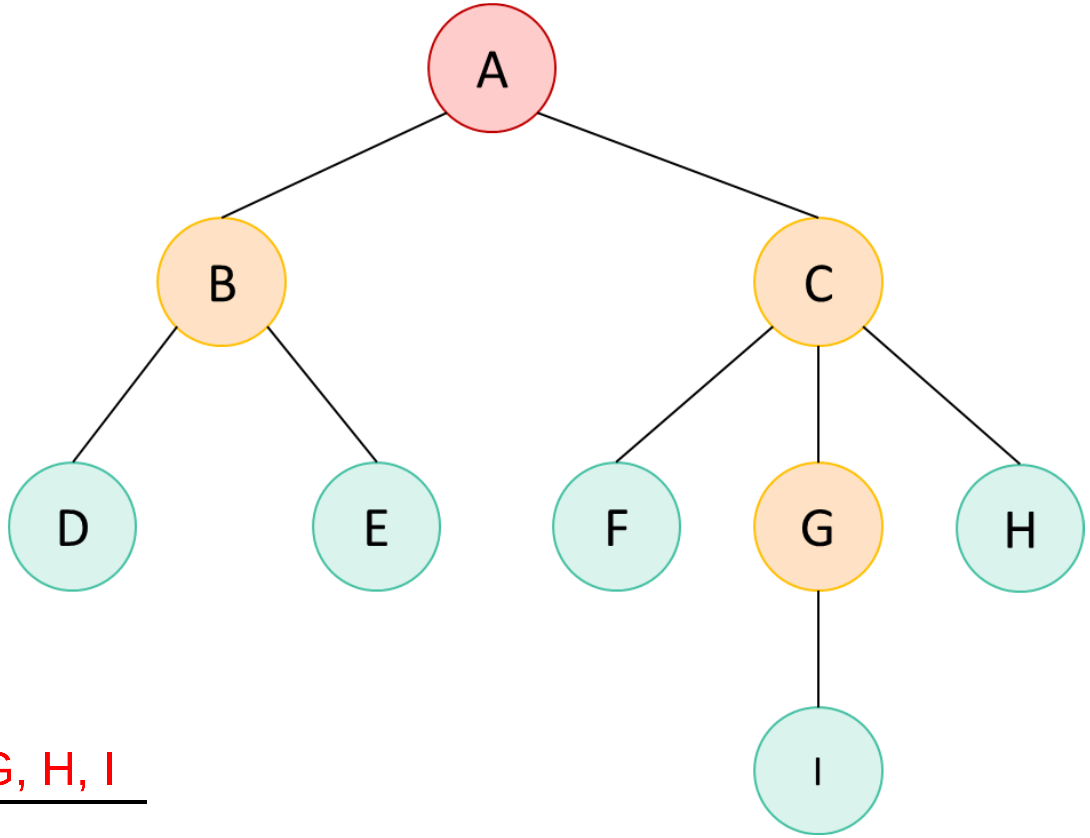
Slides adapted from Professor Leung Ho-Fung’s Lecture notes and LeetCode
Icons made by https://freepik.com

Written Exercise #1

Consider the tree on the right hand side

- Root of the tree: __A__
- Interior nodes of the tree: B, C, G
- Leaves of the tree: D, E, F, H, I
- Height of the tree: __4__

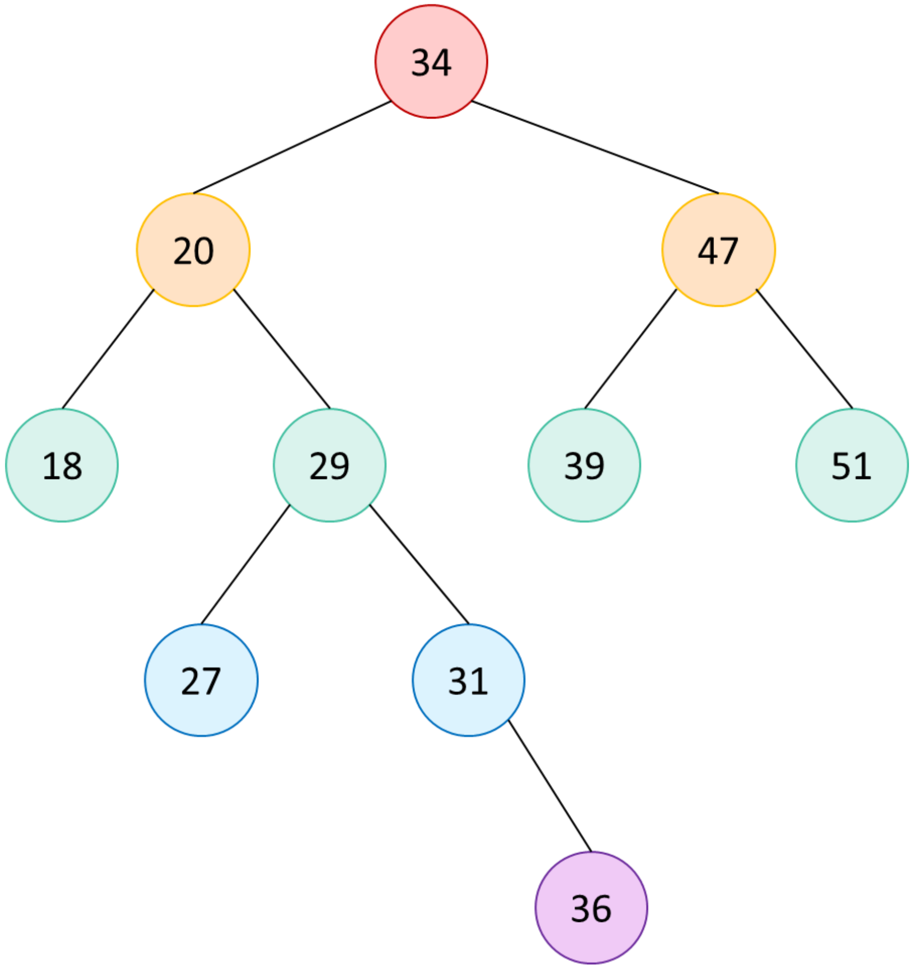
- State the parents of node C: __A__
- State the sibling of node F: G, H
- State the children of node C: F, G, H
- State the ascendent of node G: A, C
- State the descendent of node A: B, C, D, E, F, G, H, I
- Is this tree a binary tree? __No__



Written Exercise #2

Consider the tree on the right hand side

- Is this tree a binary tree? Yes
- Is this tree a Binary Search Tree? No
- Why? $36 > 34$ but 36 is in the left-subtree of 34
- Assume we change 36 to 33, is this tree a Balanced Binary Search Tree? No
- On which node the tree is not balanced? 20



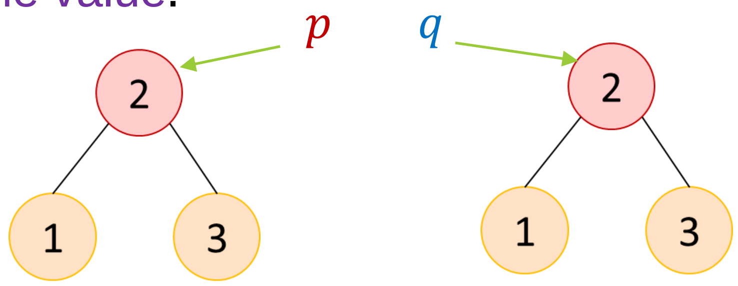
How to Write a Recursive Function

- Base case
- Recursive case (How to break down the problem)
- Assume the return value can correctly solve the sub-problem, think how to construct a solution from the solution of sub-problem

Your Assignment 2 shows that you need to practice more in writing recursive functions!

Programming Question #1 – Same Tree

- Given the roots of two binary trees p and q, write a function to check if they are the same or not.
- Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.



Tree A Tree B

Example 1: Tree A and Tree B are “Same Tree”

Tree C Tree D

Example 2: Tree C and Tree D are not “Same Tree”

```c
#include <stdbool.h>
bool isSameTree(BinaryTreeADT p, BinaryTree q) {
    //          Please finish this function
}
```

```c
typedef struct BinaryTreeCDT {
    TreeNodeADT rt;
    BinaryTreeADT lst;
    BinaryTreeADT rst;
} *BinaryTreeADT;
```

```c
typedef struct TreeNodeCDT{
    int val;
} *TreeNodeADT;
```

Programming Question #1 – Same Tree

- **Base case #1:** If p and q are both empty trees, they are same.
- **Case #2**: If in p and q, one of the tree is empty, the other is not, they are not same.
- **Claim**: If the left-subtrees and right-subtrees of p and q are identical, and the value stored in node p and q are same, the two trees are same.

```
bool isSameTree(BinaryTreeADT p, BinaryTree q) {
    if (TreeIsEmpty(p) && TreeIsEmpty(q)) {
        return true;
    }
    else if (TreeIsEmpty(p) || TreeIsEmpty(q)) {
        return false;
    }
    else if (isSameTree(LeftSubTree(p), LeftSubTree(q))
                && isSameTree(RightSubTree(p), RightSubTree(q))) {
        return GetNodeVal(Root(p)) == GetNodeVal(Root(q)) ;
    }
    else {
        return false;
    }
}
```

Note: The function parameter settings in LeetCode #100 is different from ours

Success Details ›

Runtime: 0 ms, faster than 100.00% of C online submissions for Same Tree.

Memory Usage: 6 MB, less than 31.61% of C online submissions for Same Tree.

Programming Question #2 – Invert Binary Tree

- Given a Binary Tree t, write a function to invert t as a mirror


```c
typedef struct BinaryTreeCDT {
    TreeNodeADT rt;
    BinaryTreeADT lst;
    BinaryTreeADT rst;
} *BinaryTreeADT;
```

```c
typedef struct TreeNodeCDT{
    int val;
} *TreeNodeADT;
```

```c
struct BinaryTreeADT invertTree(BinaryTreeADT t) {
    //  Please type your code here

}
```

Programming Question #2 – Invert Binary Tree

- **Base case:** If the tree is empty, return an empty tree.
- **Recursion:** Recursively calling invertTree function by left and right subtree to inverse left and right subtree

  Assign the left / right subtree to the right / left of the root.

```
BinaryTreeADT invertTree(BinaryTreeADT t) {
    if (TreeIsEmpty(t)) {
        return EmptyBinaryTree();
    }
    BinaryTreeADT lst = invertTree(RightSubTree(t));
    BinaryTreeADT rst = invertTree(LeftSubTree(t));
    return NonEmptyBinaryTree(Root(t), lst, rst);
}
```

Note: The function parameter settings in LeetCode are different from ours.
Directly copy these code into LeetCode does NOT work!

Success Details ›

Runtime: 0 ms, faster than 100.00% of C online submissions for Invert Binary Tree.

Memory Usage: 6 MB, less than 26.78% of C online submissions for Invert Binary Tree.

Programming Question #3 – Validate Binary Search Tree

- Given the $root$ of a Binary Tree, determine if it is a valid binary search tree (BST).
- A **valid BST** is defined as follows:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - Both the left and right subtrees must also be binary search trees.
- Constraints:
  - The number of nodes in the tree is in the range $[1, 10^4]$.
  - $-2^{31} \leq node\ value \leq 2^{31}$

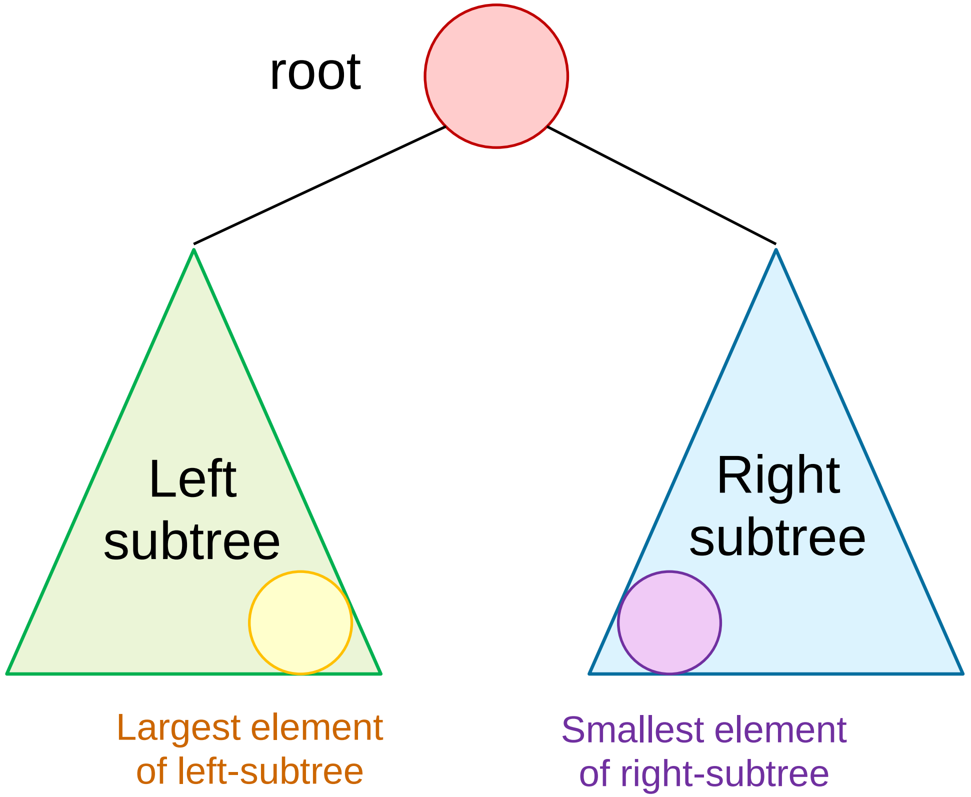
```c
#include <stdbool.h>
bool isValidBST(BinaryTreeADT t) {
    //  Please write your code here.
    //  You can write other auxiliary functions

}
```

```c
typedef struct TreeNodeCDT{
    int val;
} *TreeNodeADT;
```

```c
typedef struct BinaryTreeCDT {
    TreeNodeADT rt;
    BinaryTreeADT lst;
    BinaryTreeADT rst;
} *BinaryTreeADT;
```

Programming Question #3 – Validate Binary Search Tree

- **Base case:** If the tree is empty, it is a valid BST.
- **Case #2:** If either left or right subtree is not a valid BST, the BST is not valid.
- **Case #3:** Otherwise**,** If both left-subtree and right-subtree are empty, then the BST is valid.
- **Case #4:** If left-subtree / right-subtree is not empty, then the BST is valid if and only if the largest / smallest element of left / right subtree is smaller / greater than root element.



Programming Question #3 – Validate Binary Search Tree

- We may need two integer (pointers) to track the minimum and maximum value of current subtree.
- Therefore, we may need to create a supplementary recursive function.

```
bool validateBST(BinaryTreeADT t, int* min, int* max) {
    if (TreeIsEmpty(t)) {
        return true;
    }
    int left_min, left_max, right_min, right_max;
    if (!validateBST(LeftSubTree(t), &left_min, &left_max) || !validateBST(RightSubTree(t), &right_min, &right_max)) {
        return false;
    }
    if (TreeIsEmpty(LeftSubTree(t)) && TreeIsEmpty(RightSubTree(t))) {
        *min = GetNodeVal(t);
        *max = GetNodeVal(t);
        return true;
    }
```

"Pass by Reference"

- "left_min" is an integer, we can use '&' character to obtain its reference
- Note: type of "&left_min" is int* but not int

Programming Question #3 – Validate Binary Search Tree

Note: The function parameter settings in LeetCode are different from ours. Directly copy these code into LeetCode does NOT work!

Success   Details ›

Runtime: 8 ms, faster than 73.97% of C online submissions for Validate Binary Search Tree.

Memory Usage: 9.4 MB, less than 7.53% of C online submissions for Validate Binary Search Tree.

```
    else if (TreeIsEmpty(LeftSubTree(t))) {
      *min = GetNodeVal(t);
      *max = right_max;
      return GetNodeVal(t) < right_min;
    } else if (TreeIsEmpty(RightSubTree(t))) {
      *min = left_min;
      *max = GetNodeVal(t);
      return left_max < GetNodeVal(t);
    } else {
      *min = left_min;
      *max = right_max;
      return (left_max < GetNodeVal(t)) && (GetNodeVal(t) < right_min);
    }
}
```

```
bool isValidBST(BinaryTreeADT t){
    int min, max = 0;
    return validateBST(t, &min, &max);
}
```

- This method is called Deep First Search (DFS) or Post-order traversal.

Question

Do we have a better solution?

YES!

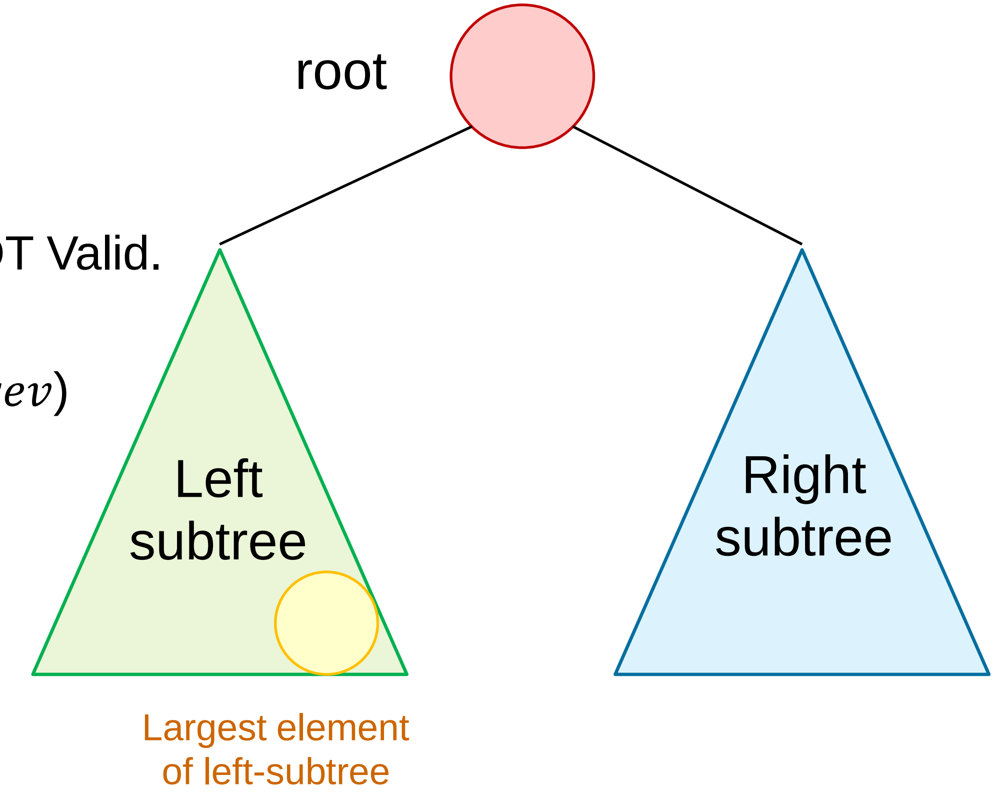
Programming Question #3 – Validate Binary Search Tree

- **Base case:** If the tree is empty, it is a valid BST.

Let's validate the BST from the left to the right subtree.

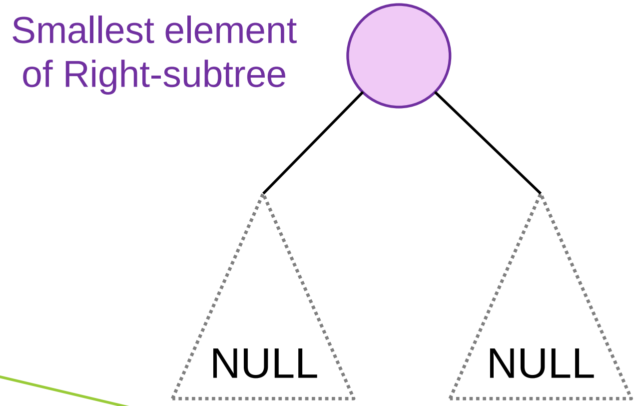
Use a variable $prev$ to track the largest element.

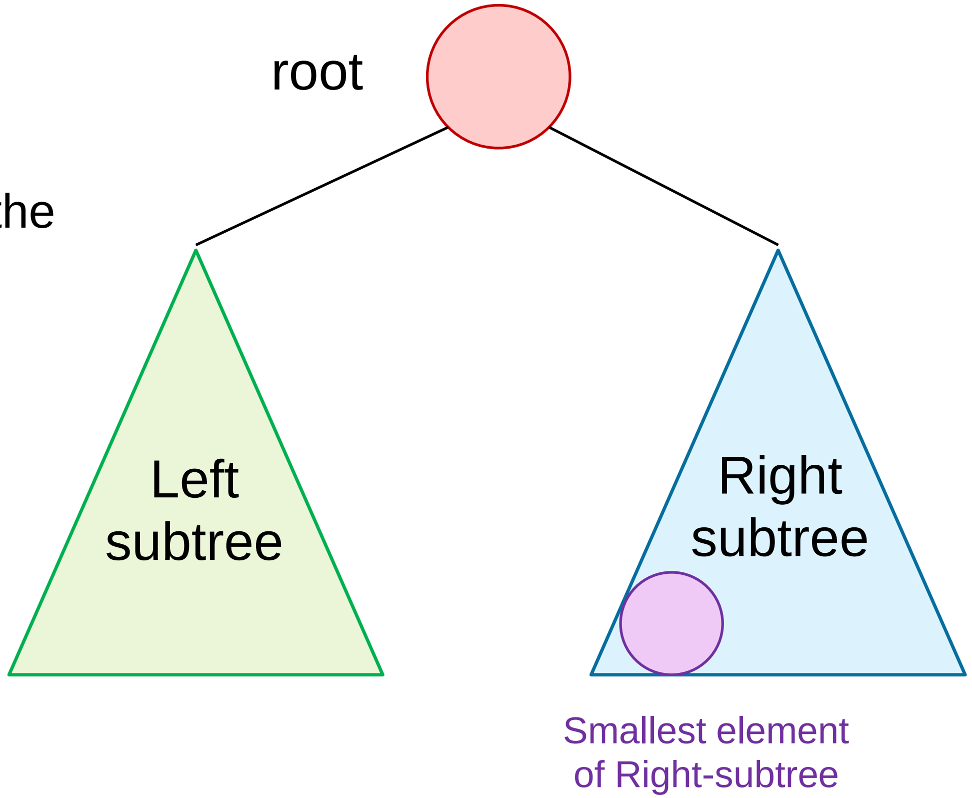
- If the left subtree is not a valid BST, then the BST is NOT Valid.
- Otherwise, If the largest element we have traversed ($prev$) is greater than root element, then the BST is not valid.
- Before checking the right subtree, mark $prev$ as the current root element.
- Return the right-subtree validation result.



How can we ensure the root element is smaller than the smallest element of right subtree?

- Before checking the right subtree, mark $prev$ as the current root element.
- Recall that our base case is TreeIsEmpty(t), consider the left-most subtree of right subtree shown below:





Question:
What's this?

For this subtree, if the largest element we have traversed ($prev$) is greater than root element, then the BST is not valid.

Intuitively you may come up with this solution

```c
bool validBST(BinaryTreeADT t, int* prev) {
    if (TreeIsEmpty(t)) {
        return true;
    }
    if (!validBST(LeftSubTree(t), prev)) {
        return false;
    }
    if (prev != NULL && GetNodeVal(t) <= *prev) {
        return false;
    }
    if (prev == NULL) {
        prev = (int*)malloc(sizeof(int));
    }
    *prev = GetNodeVal(t);
    return validBST(RightSubTree(t), prev);
}
```

```c
bool isValidBST(BinaryTreeADT t){
    int *prev = NULL;
    return validBST(t, prev);
}
```

Is this implementation appropriate? Why?

Programming Clinic

```
bool func2(int* pt2) {
    if (pt2 == NULL) {
        pt2 = (int*)malloc(sizeof(int));
    }
    *pt2 = 100;
    return pt2 == NULL;
}


int main() {
    func1();
    return 0;
}
```

```
Output:
In func1, pt is NULL
In func2, pt is NOT NULL
```

```
void func1() {
    int *pt = NULL;
    bool result = func2(pt);

    if (pt == NULL) {
        printf("In func1, pt is NULL\n" );
    } else {
        printf("In func1, pt is NOT NULL\n" );
    }

    if (result) {
        printf("In func2, pt is NULL\n" );
    } else {
        printf("In func2, pt is NOT NULL\n" );
    }
}
```

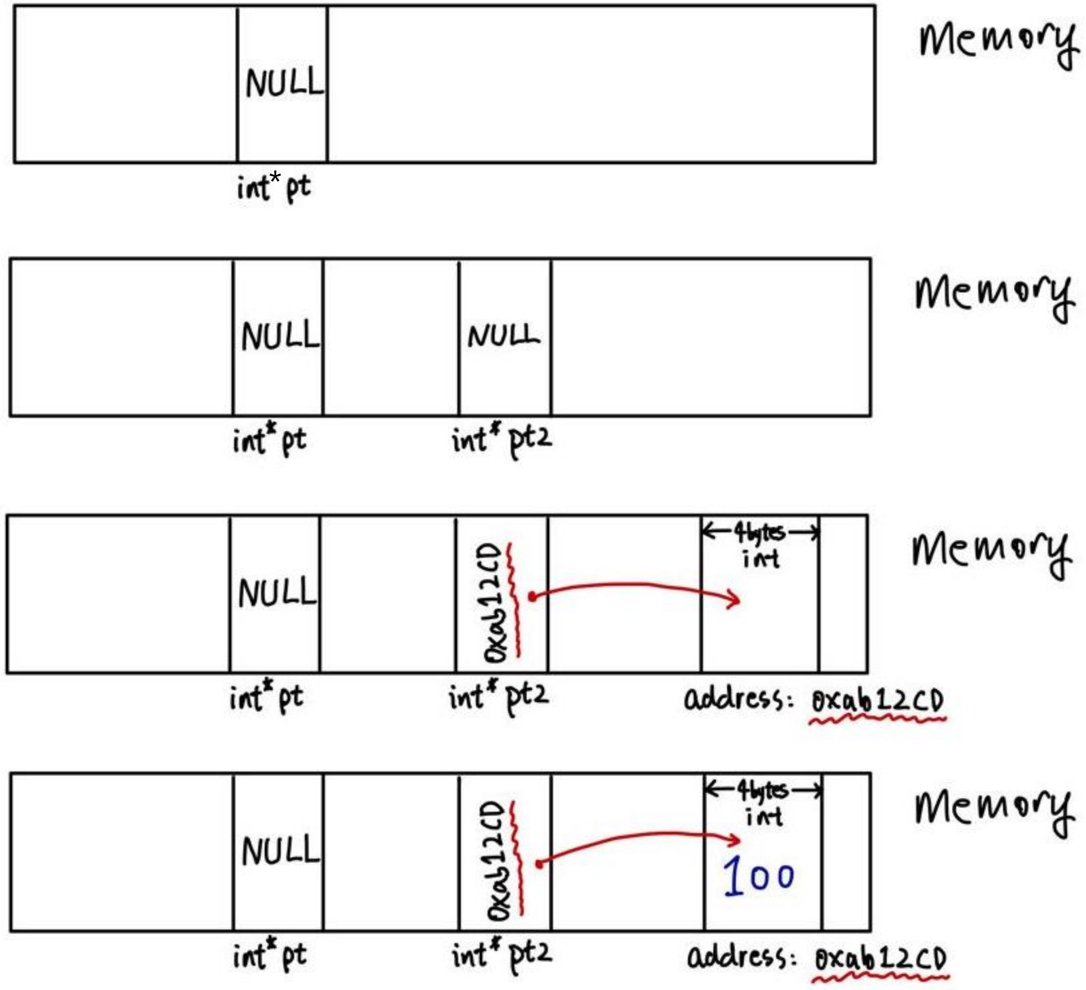
Pointer and Memory Storage

```
int *pt = NULL;
```

```
bool result = func2(pt);

bool func2(int* pt2) {
    …
}
```

```
pt2 = (int*)malloc(sizeof(int));
```

```
*pt2 = 100;
```



Programming Question #3 – Validate Binary Search Tree

```
int *prev = NULL;

bool validBST(BinaryTreeADT t) {
    if (TreeIsEmpty(p)) {
        return true;
    }
    if (!validBST(LeftSubTree(t))) {
        return false;
    }
    if (prev != NULL && GetNodeVal(t) <= *prev) {
        return false;
    }
    if (prev == NULL) {
        prev = (int*)malloc(sizeof(int));
    }
    *prev = GetNodeVal(t);
    return validBST(RightSubTree(t));
}
```

```
bool isValidBST(BinaryTreeADT t){
    prev = NULL;
    return validBST(t);
}
```

Note: The function parameter settings in LeetCode are different from ours.
Directly copy these code into LeetCode does NOT work!

Success   Details ›

Runtime: 8 ms, faster than 73.97% of C online submissions for Validate Binary Search Tree.

Memory Usage: 9 MB, less than 10.62% of C online submissions for Validate Binary Search Tree.

Homework

Please **re-do** the programming problem
by yourself <u>without</u> the help of my slides.