Trees

Both arrays and lists are linear data structures, so it is difficult to utilize them to organize a hierarchical representation of objects. To overcome this limitation, a new data type named a tree was introduced, which consists of a set of nodes and links among them.

Trees are commonly used during our daily programming. In trees, all nodes except the root node have a parent node, and all nodes except leaves have one or more children nodes.

Interview questions about trees are usually not easy because there are many pointer operations on trees. If interviewers would like to examine candidates' capacity to handle complex pointer operations, they are likely to employ questions about trees.

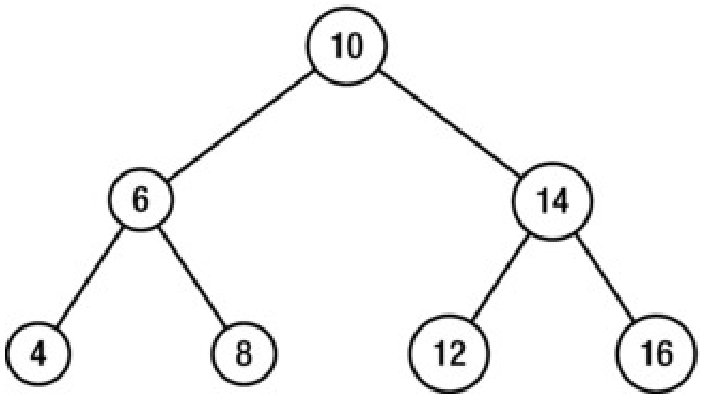

Figure 3-11. *A sample binary tree with seven nodes*

Most trees referred to during interviews are binary trees, where every node has two children at most. The most important operation on trees is traversal, which is to visit nodes in some order. The most common traversal algorithms include the following:

- *Pre-order traversal:* The root of a binary tree is visited first, then its left children, and finally its right children. The pre-order traversal sequence of the binary tree in Figure 3-11 is 10, 6, 4, 8, 14, 12, 16.
- *In-order traversal:* Left children of a binary tree are visited first, then its root, and finally its right children. The in-order traversal sequence of the binary tree in Figure 3-11 is 4, 6, 8, 10, 12, 14, 16.
- *Post-order traversal:* Left children of a binary tree are visited first, then its right children, and finally its root. The post-order traversal sequence of the binary tree in Figure 3-11 is 4, 8, 6, 12, 16, 14, 10.
- *Breadth-first traversal:* Nodes in the first level are traversed, then nodes in the second level, …, and finally nodes in the bottom level. Nodes in the same level are usually traversed from left to right. The breadth-first traversal sequence of the binary tree in Figure 3-11 is 10, 6, 14, 4, 8, 12, 16.

The first three traversal algorithms in the preceding list can be implemented with both recursion and iteration, and recursive implementations are more concise than iterative ones. Candidates should be very familiar with these six implementations and be able to implement them with bug-free code in a short period of time.

There are many interview questions about tree traversal. For example, questions about "Subtrees" (Question 50), "Path with Sum in Binary Trees" (Question 60), and "Depth of Binary Trees" (Question 85) are in this category. The interview question "Build a Binary Tree from Traversal Sequences" (Question 61) is about the characteristics of traversals.

Usually, a queue is utilized for breadth-first traversal algorithms. There are also many interesting interview questions on this topic, which are discussed in the section *Print Binary Trees Level by Level*.

There are some special binary trees such as binary search trees. In a binary search tree, all nodes in the left subtree are not greater than the root node, and all nodes in the right subtree are not less than the root node. Binary search trees are closely related to the binary search algorithm, where it costs O(log*n*) time to find a value among *n* nodes.

The tree in Figure 3-11 is actually a binary search tree. "Binary Search Tree Verification" (Question 19) and "Binary Search Tree and Double-Lined List" (Question 64) are examples of interview questions about binary search trees.

Another category of binary trees is the heap. There are two kinds of heaps: max heaps and min heaps. The value in the root node is the maximum in a max heap, while the value in the root node is the minimum in a min heap. If it is required to find the maximal or minimal value, you may consider employing heaps. Please refer to interview questions "Median in a Stream" (Question 69) and "Minimal *k* Numbers" (Question 70) for more details.

Next Nodes in Binary Trees

Question 18 Given a node in a binary tree, please implement a function to retrieve its next node in the in-order traversal sequence. There is a pointer to the parent node in each tree node.

The tree in Figure 3-12 is a binary tree whose in-order traversal sequence is *d*, *b*, *h*, *e*, *i*, *a*, *f*, *c*, *g*. Let's take it as an example to analyze how to get the next node in a binary tree.

If a node has a right child, its next node is the most left child in its right subtree. That is to say, it moves to the right child and then traverses along the links to the left child as much as possible. For example, the next node of node *b* is node *h*, and the next node of node *a* is node *f*.

If a node does not have a right child, its next node is its parent if it is the left child of its parent. For instance, the next node of node *d* is node *b*, and the next node of node *f* is node *c*.

It is more complex to get the next node of a node that does not have a right child and is the right child of its parent. It traverses along the links to parents until it reaches a node that is the left child of its parent. The parent is the next node if such a node exists.

In order to get the next node of node *i*, it traverses along the link to the parent and reaches node *e* at first. Since node *e* is not the left child of its parent, it continues to traverse and reaches at node *b*, which is the left child of its parent. Therefore, the parent of node *b*, which is node *a*, is the next node after node *i*.

It is a similar process to get the next node of node g. It first traverses the link to the parent and reaches node *c*. It continues to traverse because node *c* is not a left child of its parent, and it reaches node *a*. Because node *a* does not have a parent, node *g* is the last in the binary tree and it does not have a next node.

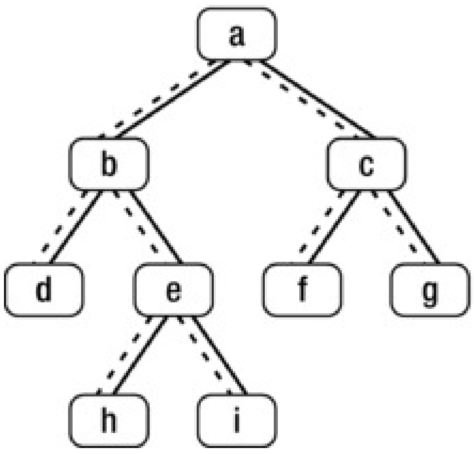

Figure 3-12. *A binary tree with nine nodes. Links from children to parents are drawn with dashed arrows.*

The C++ code to get the next node of a given node is found in Listing 3-24.

Listing 3-24. *C++ Code to Get the Next Node in a Binary Tree*

```
    BinaryTreeNode* GetNext(BinaryTreeNode* pNode) {
    if(pNode == NULL)
        return NULL;

    BinaryTreeNode* pNext = NULL;
    if(pNode->m_pRight != NULL) {
        BinaryTreeNode* pRight = pNode->m_pRight;
        while(pRight->m_pLeft != NULL)
            pRight = pRight->m_pLeft;

        pNext = pRight;
    }
    else if(pNode->m_pParent != NULL) {
        BinaryTreeNode* pCurrent = pNode;
        BinaryTreeNode* pParent = pNode->m_pParent;
        while(pParent != NULL && pCurrent == pParent->m_pRight) {
            pCurrent = pParent;
            pParent = pParent->m_pParent;
        }

        pNext = pParent;
    }

    return pNext;
}
```

Source Code:

```
018_NextNode.cpp
```

Test Cases:

- Input different kinds of binary trees, such as full binary trees or binary trees, in which all nodes only have right subtrees or left subtrees
- The next node is in the right subtree of the input node, the parent, or the skip-level ancestors
- The input node does not have a next node
- The input node of tree root is `NULL`

Binary Search Tree Verification

Question 19 How do you verify whether a binary tree is a binary search tree? For example, the tree in Figure 3-13 is a binary search tree.

The binary search tree is a specific type of binary tree that has an important characteristic: each node is greater than or equal to nodes in its left subtree, and less than or equal to nodes in its right subtree. We can verify binary search trees bases on this characteristic, as shown in Figure 3-13.

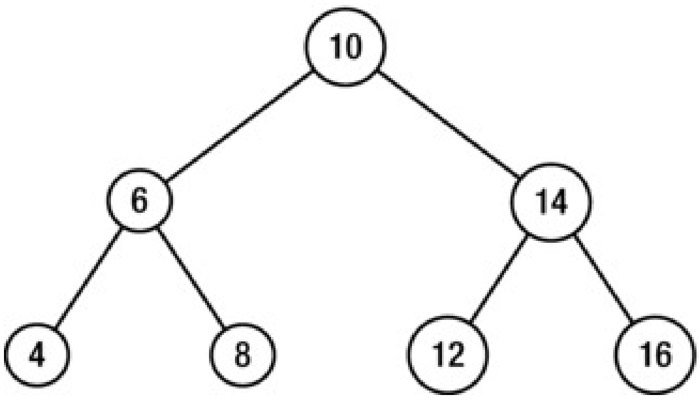

Figure 3-13. *A sample binary search tree with seven nodes*

Verify Value Range of Each Node

If a binary search tree is scanned with a pre-order traversal algorithm, the value in a root node is accessed first. After the root node is visited, it begins to scan nodes in the left subtree. The value of left subtree nodes should be less than or equal to the value of the root node. If a value of a left subtree node is greater than the value of the root node, it violates the definition of the binary search tree. Similarly, it also violates the definition of the binary search tree when a value of a right subtree node is less than the root node value because the value of the right subtree nodes should be greater than or equal to the root node value.

Therefore, when it visits a node in binary search tree, it narrows the value range of the left subtree and right subtree under the current visited node. All nodes are visited with the pre-order traversal algorithm, and their value is verified. If a value in any node is beyond its corresponding range, it is not a binary search tree.

The code in Listing 3-25 is implemented based on this pre-order traversal solution.

Listing 3-25. *C++ Code for Binary Search Tree Verification (Version 1)*

```
   bool isBST_Solution1(BinaryTreeNode* pRoot) {

       int min = numeric_limits<int>::min();

    int max = numeric_limits<int>::max();

    return isBSTCore_Solution1(pRoot, min, max);

}


bool isBSTCore_Solution1(BinaryTreeNode* pRoot, int min, int max) {

    if(pRoot == NULL)

        return true;


    if(pRoot->nValue < min || pRoot->nValue > max)

        return false;


    return isBSTCore_Solution1(pRoot->pLeft, min, pRoot->nValue)

        && isBSTCore_Solution1(pRoot->pRight, pRoot->nValue, max);

}
```

In this code, the value of each node should be in the range between `min` and `max`. The value of the current visited node is the maximal value of its left subtree and the minimal value of its right subtree, so it updates the `min` and `max` parameters and verifies subtrees recursively.

Increasing In-Order Traversal Sequence

The first solution is based on the pre-order traversal algorithm. Let's have another try using in-order traversal. The in-order traversal sequence of the binary search tree in Figure 3-13 is 4, 6, 8, 10, 12, 14, and 16. It is noticeable that the sequence is increasingly sorted.

Therefore, a new solution is available: nodes in a binary tree are scanned with the in-order traversal algorithm and compare values of each node against the value of the previously visited node. If the value of the previously visited node is greater than the value of the current node, it breaks the definition of a binary search tree.

This solution might be implemented in C++, as shown in Listing 3-26.

Listing 3-26. *C++ Code for Binary Search Tree Verification (Version 2)*

```
   bool isBST_Solution2(BinaryTreeNode* pRoot) {

    int prev = numeric_limits<int>::min();

    return isBSTCore_Solution2(pRoot, prev);

}
```

```
bool isBSTCore_Solution2(BinaryTreeNode* pRoot, int& prev) {

    if(pRoot == NULL)

        return true;



        // previous node

    return isBSTCore_Solution2(pRoot->pLeft, prev)

        // current node

        && (pRoot->nValue >= prev)

        // next node

        && isBSTCore_Solution2(pRoot->pRight, prev = pRoot->nValue);

}
```

The parameter `prev` of the function `isBSTCore_Solution2` is the value of the previously visited node in the pre-order traversal sequence.

Source Code:

`019_VerrifyBinarySearchTrees.cpp`

Test Cases:

- Binary trees (such as full binary trees or binary trees in which all nodes only have right subtrees or left subtrees) are binary search trees
- Binary trees (such as full binary trees or binary trees in which all nodes only have right subtrees or left subtrees) are not binary search trees
- Special binary trees, including a binary tree that has only one node, or the input node of tree root is `NULL`

Question 20 Please implement a function to get the largest size of all binary search subtrees in a given binary tree. A subtree inside a tree *t* is a tree consisting of a node and all of its descendants in *t*. The size of a tree is defined as the number of nodes in the tree.
For example, the largest binary search subtree in the binary tree of Figure 3-14 contains three nodes, which are node 9, node 8, and node 10.

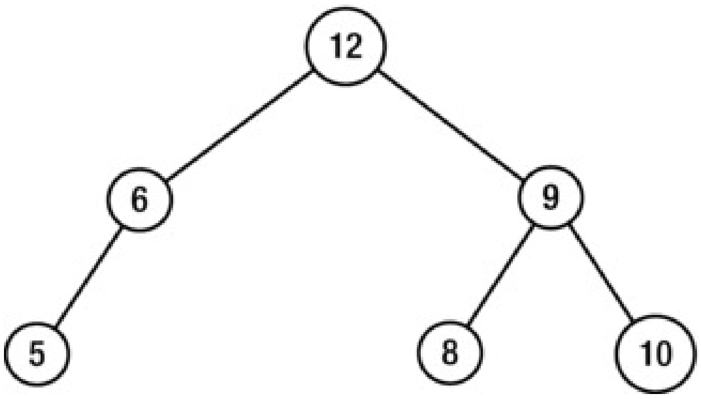

Figure 3-14. *A sample binary tree with six nodes in which the largest binary search subtree has three nodes (node 9, node 8, and node 10)*

There are two subproblems in this problem. One subproblem is how to verify whether a subtree is a binary search tree, and the other is how to find the size of a binary search subtree.

The whole tree may not be a binary tree even though some subtrees are binary search trees. Therefore, the solution has to verify binary search subtrees in bottom-up order. When the left subtree or right subtree under a node is not a binary search tree, the tree under the node cannot be a binary search. If both left subtree and right subtree are binary search trees and the value of the node is between the maximum in the left subtree and the minimum in the right subtree, the node and its descendants compose a binary search tree.

For example, node 6 and its children compose a binary search tree, and node 9 and its children compose another binary search tree in Figure 3-14. However, the tree rooted at node 12 is not a binary search tree because its value is greater than the minimal value in its right children.

The solution finds the size of a binary search subtree also in bottom-up order. When it visits a node, it finds the largest size of binary search trees in its left subtree (*size1*) and right subtree (*size2*). If both left and right subtrees are binary search trees and the value of the current visited node is inside the corresponding range, the node and its descendants compose a binary search tree whose size is *size1+size2+1*. If the current visited node and its descendants cannot compose a binary search tree, the largest size of binary search subtree is the maximal value between *size1* and *size2*.

The size of the subtree rooted at node 6 in Figure 3-14 is 2, and the size of the subtree rooted at node 9 is 3. Because the whole tree rooted at node 12 is not a binary search tree, the largest size of the binary search subtree is 3, the maximum between 2 and 3. (See Listing 3-27.)

Listing 3-27. *C++ Code to Get the Largest Size of Binary Search Subtrees*

```
int LargestBST(BinaryTreeNode* pRoot) {

    int min, max, largestSize;

    LargestBST(pRoot, min, max, largestSize);
    return largestSize;
}

bool LargestBST(BinaryTreeNode* pRoot, int& min, int& max, int& largestSize) {
    if(pRoot == NULL) {
        max = 0x80000000;
        min = 0x7FFFFFFF;
        largestSize = 0;
        return true;
    }

    int minLeft, maxLeft, leftSize;
    bool left = LargestBST(pRoot->pLeft, minLeft, maxLeft, leftSize);

    int minRight, maxRight, rightSize;
    bool right  = LargestBST(pRoot->pRight, minRight, maxRight, rightSize);

    bool overall = false;
    if(left && right && pRoot->nValue >= maxLeft && pRoot->nValue <= minRight)
    {
        largestSize = leftSize + rightSize + 1;
        overall = true;

        min = (pRoot->nValue < minLeft) ? pRoot->nValue : minLeft;
        max = (pRoot->nValue > maxRight) ? pRoot->nValue : maxRight;
    }
    else
        largestSize = (leftSize > rightSize) ? leftSize : rightSize;

    return overall;
}
```

Source Code:

`020_LargestBinarySearchSubtrees.cpp`

Test Cases:

- A whole binary tree is a binary search tree
- The left or right subtree under a certain node is a binary search tree, but the node, the left subtree, and the right subtree do not form a binary search tree as a whole
- Special binary trees, including a binary tree that has only one node, or the input node of tree root is `NULL`