Divide and Conquer

The divide-and-conquer approach is an important algorithm design paradigm to analyze and solve complicated problems. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more similar subproblems until the subproblems become simple enough to be solved directly. The solutions to the subproblems are then combined to compose a solution to the original problem.

Take the problem to convert a binary search tree to a sorted double-linked list as an example. The problem is to convert a binary search tree; it can be divided into two subproblems to convert the left and right subtrees. After two subtrees are converted to two sorted double-linked lists, they are connected to the root node and the original tree is converted. More details are available in the section *Binary Search Trees and Double-linked Lists*.

The divide-and-conquer algorithm also works when it reduces a problem to only one subproblem. For example, if the problem is to permute characters in a string, it can be divided into a subproblem to permute all characters excluding the first one. Please refer to the section *Permutation and Combination* for more details about this problem.

Traversal Sequences and Binary Trees

Question 61 Please build a binary tree with a pre-order traversal sequence and an in-order traversal sequence. All elements in these two given sequences are unique.

For example, if the input pre-order traversal sequence is {1, 2, 4, 7, 3, 5, 6, 8} and in-order traversal order is {4, 7, 2, 1, 5, 3, 8, 6}, the built tree is shown in Figure 6-13.

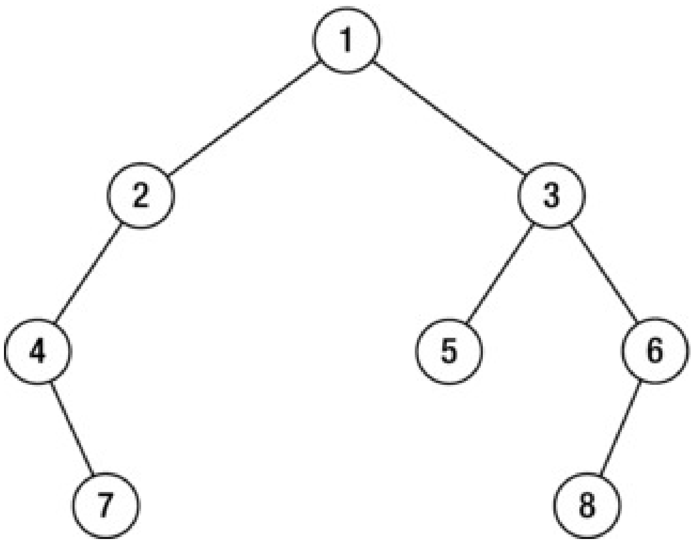

Figure 6-13. *The binary tree built from pre-order traversal sequence {1, 2, 4, 7, 3, 5, 6, 8} and in-order traversal sequence {4, 7, 2, 1, 5, 3, 8, 6}*

The root value is the first number in the pre-order traversal sequence of a binary tree, but it is in the middle of the in-order traversal sequence. Therefore, we have to scan the in-order traversal sequence to get the root value. As shown in Figure 6-14, the first number in the pre-order traversal sequence (number 1) is the root value. The location of root value is found if the whole in-order traversal sequence is scanned.

In the in-order traversal sequence, all numbers at the left side of the root value are for nodes in the left subtree, and all numbers at the right side of the root value correspond to nodes in the right subtree. According to the in-order traversal sequence in Figure 6-14, we find that there are three nodes in the left subtree and four in the right subtree. Therefore, the next three numbers behind the root value in the pre-order traversal sequence are for the left subtree, and the following four are for the right subtree.

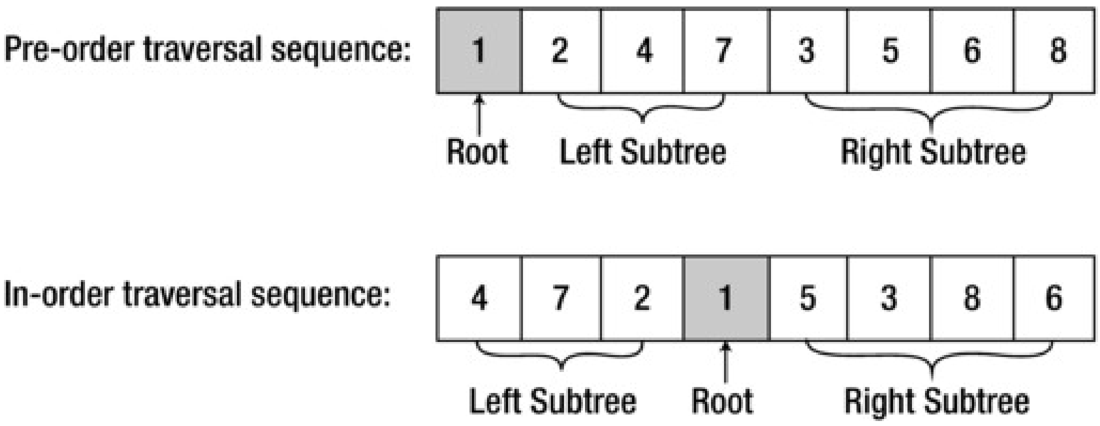

Figure 6-14. *Find the root node, the nodes in the left subtree, and the nodes in the right subtree in the pre-order traversal sequence and in-order traversal sequence.*

After the pre-order and in-order traversal sequences for both left and right subtrees are identified, subtrees can be built with recursion, similar to the process of building the whole tree. The sample code is shown in Listing 6-16 based on the divide-and-conquer approach.

Listing 6-16. *C++ Code to Build Binary Trees from Traversal Sequences*

```
    BinaryTreeNode* Construct(int* preorder, int* inorder, int length) {
    if(preorder == NULL || inorder == NULL || length <= 0)
        return NULL;

        return ConstructCore(preorder, preorder + length - 1,
        inorder, inorder + length - 1);
}

BinaryTreeNode* ConstructCore(int* startPreorder, int* endPreorder, int* startInorder, int* endInorder) {

    // The first number in the pre-order traversal sequence is the root value
    int rootValue = startPreorder[0];
    BinaryTreeNode* root = new BinaryTreeNode();
    root->m_nValue = rootValue;
    root->m_pLeft = root->m_pRight = NULL;

    if(startPreorder == endPreorder) {
        if(startInorder == endInorder
            && *startPreorder == *startInorder)
            return root;
        else
            throw std::exception("Invalid input.");
    }

    // Get the root value in the in-order traversal sequence
    int* rootInorder = startInorder;
    while(rootInorder <= endInorder && *rootInorder != rootValue)
        ++ rootInorder;

    if(rootInorder == endInorder && *rootInorder != rootValue)
        throw std::exception("Invalid input.");

    int leftLength = rootInorder - startInorder;
    int* leftPreorderEnd = startPreorder + leftLength;
    if(leftLength > 0) {
        // Build left subtree
        root->m_pLeft = ConstructCore(startPreorder + 1,
            leftPreorderEnd, startInorder, rootInorder - 1);
    }
    if(leftLength < endPreorder - startPreorder) {
        // Build rigth subtree
        root->m_pRight = ConstructCore(leftPreorderEnd + 1,
            endPreorder, rootInorder + 1, endInorder);
    }

    return root;
}
```

In the function `ConstructCore`, first the root value is gotten in the pre-order traversal sequence, and then it is located in the in-order traversal sequence. After the subsequence for the left and right subtrees are partitioned in the pre-order and in-order traversal sequences, the function `ConstructCore` is called recursively to build the left and right subtrees.

Source Code:

`061_ConstructBinaryTree.cpp`

Test Cases:

- Functional Cases (Sequences for normal binary trees, including some full binary trees; no binary trees can be built from the given sequences)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, or all nodes in a binary tree only have left/right subtrees)
- Cases for Robustness (The pointers to one or two arrays are `NULL`)

Question 62 How do you serialize and deserialize binary trees?

As discussed before, a binary tree can be built from a pre-order traversal sequence and an in-order traversal sequence. Inspired by this conclusion, if a binary is serialized to two number arrays for the pre-order and in-order traversal sequences, it can be rebuilt from these two arrays during deserialization.

This solution works, but has two disadvantages. One disadvantage is that there should not be any duplicated numbers in the array; otherwise, it causes problems during deserialization. The other disadvantage is that it has to read all numbers in the two traversal sequences before deserialization begins, so it has to wait for a long time to start deserialization if data comes from a stream. Let's explore other alternatives.

If serialization of a binary tree starts from the root node, the corresponding deserialization can start once the root value is received. Therefore, the pre-order traversal algorithm is applied to serialize a binary tree, which visits the root node first. When a `NULL` pointer is reached during traversal, it is serialized as a special character (such as a '$'). Additionally, node values should be separated by another character (such as a comma ','). According to these serialization rules, the binary tree in Figure 6-15 is serialized as a sequence "1,2,4,$,$,$,3,5,$,$,6,$,$". (See Listing 6-17.)

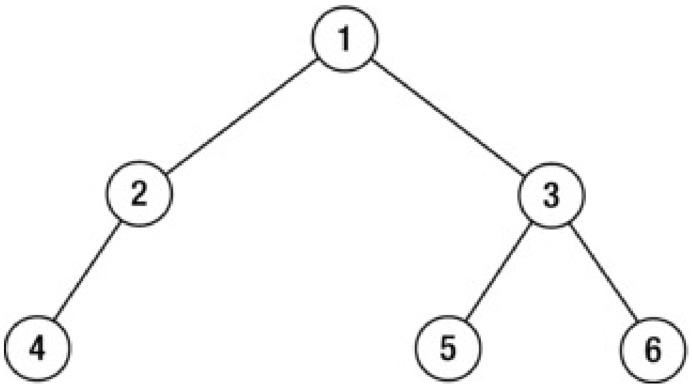

Figure 6-15. A binary tree whose serialization sequence is "1,2,4,$,$,$,3,5,$,$,6,$,$"

Listing 6-17. C++ Code to Serialize Binary Trees

```
void Serialize(BinaryTreeNode* pRoot, ostream& stream) {

 if(pRoot == NULL) {

     stream << "$,";

     return;

 }


 stream << pRoot->m_nValue << ',';

    Serialize(pRoot->m_pLeft, stream);

 Serialize(pRoot->m_pRight, stream);

}
```

Let's move on to analyze how to deserialize a binary tree. Take the serialization sequence "1,2,4,$,$,$,3,5,$,$,6,$,$" as an example. It first reads a number 1 from the stream, which is the root value of a binary tree. Another number 2 is read. Since the binary tree is serialized with the pre-order traversal algorithm, the number 2 is taken as the left child of the root node. Similarly, the next number, 4, is deserialized as the left child of the node with value 2. The following data read from the stream are two '$'s, which means that the node with value 4 is a leaf and does not have children nodes. It returns back to the parent node, the node with value 2, and continues to deserialize its right child. The right child of the node with value 2 is `NULL` because the next character read from the stream is also a '$'.

It returns back to the root node to deserialize the right subtree. A number 3 is read, so the right child of the root node contains value 3. The following data from the stream is another number 5, which is taken as the left child value of the node with value 3. The following two '$'s indicate that the node with value 5 is a leaf node. It continues to deserialize the right child of the node with value 3, which is another leaf node with value 6 because the next data from the stream is "6,$,$". It stops deserialization because all children of all non-leaf nodes have been constructed.

Listing 6-18 is the sample code used to deserialize a binary tree in C++.

Listing 6-18. C++ Code to Deserialize Binary Trees

```
void Deserialize(BinaryTreeNode** pRoot, istream& stream) {

 int number;

 if(ReadStream(stream, &number)) {

     *pRoot = new BinaryTreeNode();

     (*pRoot)->m_nValue = number;

     (*pRoot)->m_pLeft = NULL;

     (*pRoot)->m_pRight = NULL;
```

```
        Deserialize(&((*pRoot)->m_pLeft), stream);

        Deserialize(&((*pRoot)->m_pRight), stream);

    }

}
```

The function `ReadStream` in this code reads a token (a number or a '$') from a stream at every step. When a token received from the stream is a number, it returns `true`.

Source Code:

`062_SerializeBinaryTree.cpp`

Test Cases:

- Functional Cases (Normal binary trees, including some full binary trees)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, or all nodes in a binary tree only have left/right subtrees)
- Cases for Robustness (The pointer to the root node of a binary tree is `NULL`)

Question 63 Please check whether it is possible for an array to be the post-order traversal sequence of a binary search tree. All numbers in the input array are unique.

For example, the array {5, 7, 6, 9, 11, 10, 8} is the post-order traversal sequence of the binary search tree in Figure 6-16. However, there is not a binary search tree whose post-order traversal sequence is {7, 4, 6, 5}.

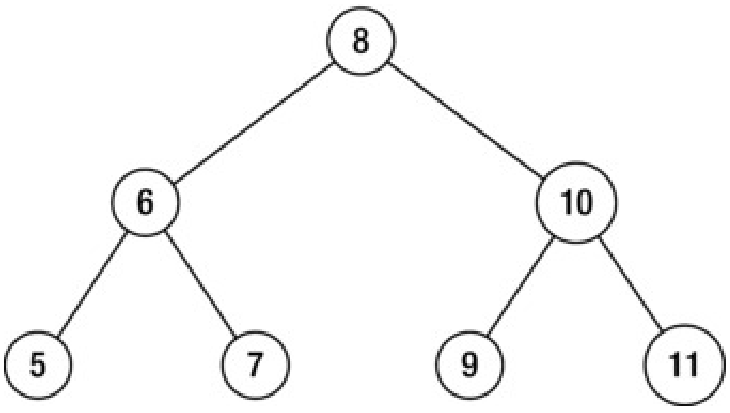

Figure 6-16. *A binary search tree built from a post-order traversal sequence {5, 7, 6, 9, 11, 10, 8}*

The last number in a post-order traversal sequence is the value of the root node. Other numbers in the sequence can be partitioned into two parts. The first numbers at the left side in the array, which are less than the value of the root node, are values of nodes in the left subtree; the following numbers, which are greater than the value of the root node, are values of nodes in the right subtree.

Take the input {5, 7, 6, 9, 11, 10, 8} as an example. The last number 8 in this sequence is the root value. The first three numbers (5, 7, and 6), which are less than 8, are values of nodes in the left subtree. The following three numbers (9, 11, and 10), which are greater than 8, are values of nodes in the right subtree.

The solution continues to construct the left subtree and right subtree according to the two sub-arrays with the same strategy. In the subsequence {5, 7, 6}, the last number 6 is the root value of the left subtree. The number 5 is the value of the left child since it is less than 6, and 7 is the value of the right child since it is greater than 6. Meanwhile, the last number 10 in the subsequence {9, 11, 10} is the root value of the right subtree. The number 9 is the value of the left child, and 11 is the value of right child accordingly.

Let's analyze another array {7, 4, 6, 5}. The last number 5 is the value of the root node. Since the first number 7 is greater than 5, there are no nodes in the left subtree, and the numbers 7, 4, 6 should be values of nodes in the right subtree. However, notice that the number 4, a value in the right subtree, is less than the root value 5. It violates the definition of binary search trees. Therefore, there are no binary search trees with the post-order traversal sequence {7, 4, 6, 5}.

It is not difficult to write code after we have examined this strategy. Some sample code is shown in Listing 6-19.

Listing 6-19. *C++ Code to Verify a Post-Order Traversal Sequence*

```
bool VerifySquenceOfBST(int sequence[], int length) {

    if(sequence == NULL || length <= 0)

        return false;


    int root = sequence[length - 1];


        // nodes in left sub-tree are less than root node

    int i = 0;

    for(; i < length - 1; ++ i) {

        if(sequence[i] > root)

            break;

    }
    // nodes in right sub-tree are greater than root node
    int j = i;
    for(; j < length - 1; ++ j) {
        if(sequence[j] < root)
            return false;
    }

    // Is left sub-tree a binary search tree?
    bool left = true;
    if(i > 0)
        left = VerifySquenceOfBST(sequence, i);

    // Is right sub-tree a binary search tree?
    bool right = true;
    if(i < length - 1)
        right = VerifySquenceOfBST(sequence + i, length - i - 1);

    return (left && right);
}
```

Source Code:

`063_SequenceOfBST.cpp`

Test Cases:

- Functional Cases (Post-order traversal sequences correspond to binary search trees, including full binary search trees; post-order traversal sequences do not correspond to any binary search trees)
- Boundary Cases (Special post-order traversal sequences where elements are increasingly or decreasingly sorted; there is only one element in the sequence)
- Cases for Robustness (The pointer to the array of traversal sequence is `NULL`)

Binary Search Trees and Double-Linked Lists

Question 64 How do you convert a binary search tree into a sorted double-linked list without creating any new nodes? It is only allowed that you can reconnect links between existing nodes.
For example, after reconnecting links in the binary search tree in Figure 6-17(a), it gets a sorted double-linked list in Figure 6-17(b).

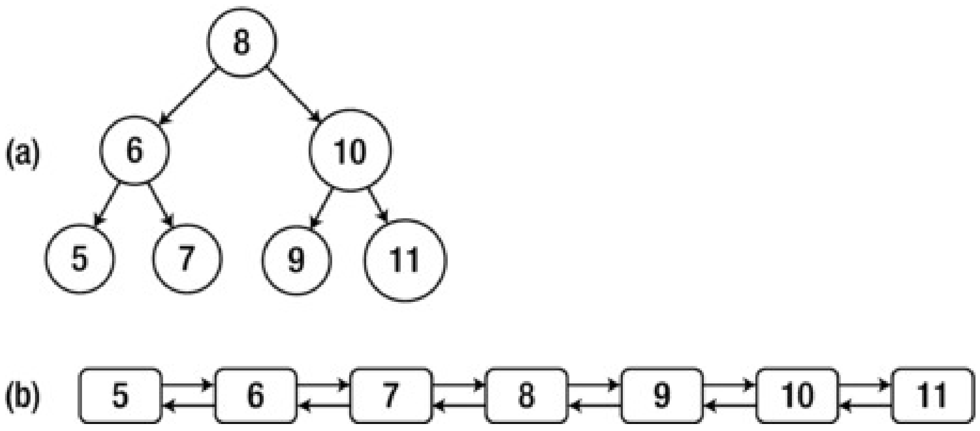

Figure 6-17. *A binary search tree and its converted sorted double-linked list*

Every node in a binary search tree has two links to its children nodes (some links might be `NULL` pointers). Every node in a double-linked list also has two links, one to the preceding node and the other to the next node. Additionally, the binary search tree is also a data structure for sorting, so a binary search tree can be converted to a sorted double-linked list in theory. The link to the left child in a binary search tree is connected to the preceding node in the converted double-linked list, and the link to the right child is connected to the next node.

Based on Divide and Conquer

If a binary search tree is scanned with the in-order traversal algorithm, nodes are traversed in increasing order of values. When the root node of the sample tree is visited, the tree can be viewed in three parts: the root value with value 8, the left subtree rooted at the node with value 6, and the right subtree rooted at the node with value 10. According to the sorting requirement, the root node with value 8 should be linked to the node in the left subtree with the maximal value (the node with value 7), as well as the node in the right subtree with the minimum value (the node with value 9), as shown in Figure 6-18.

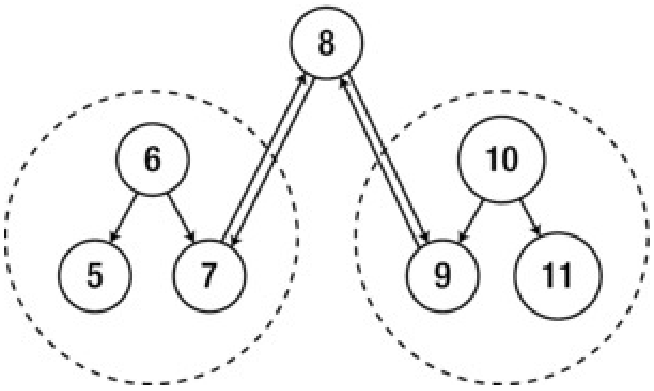

Figure 6-18. Divide a binary tree into three parts: root node, left subtree, and right subtree. After the left subtree and right subtree are converted to double-linked lists, they are connected to the root node and the whole binary search tree is converted to a sorted double-linked list.

Since the tree is scanned with the in-order traversal algorithm, the left subtree is converted when the root node is visited, and the last node in the converted double-linked list has the maximum value so far (node 7). The root node is connected to the list then, and it becomes the new last node in the list. We continue to convert the right subtree and connect the root node to the node with the minimum value in the right subtree (node 9). How do you convert the left and right subtrees? The process should be similar to the process to convert the whole tree, so it can be solved with recursion, as shown in Listing 6-20.

***Listing 6-20.** C++ Code to Convert a BSTree to a Double-Linked List (Version 1)*

```
  BinaryTreeNode* Convert(BinaryTreeNode* pRootOfTree) {
    BinaryTreeNode *pLastNodeInList = NULL;
    ConvertNode(pRootOfTree, &pLastNodeInList);

    // pLastNodeInList points to the last node in the doule-linked list,
    // but we are going to return the head node.
    BinaryTreeNode *pHeadOfList = pLastNodeInList;
    while(pHeadOfList != NULL && pHeadOfList->m_pLeft != NULL)
        pHeadOfList = pHeadOfList->m_pLeft;

    return pHeadOfList;
}

void ConvertNode(BinaryTreeNode* pNode, BinaryTreeNode** pLastNodeInList) {
    if(pNode == NULL)
        return;

    BinaryTreeNode *pCurrent = pNode;

    if (pCurrent->m_pLeft != NULL)
        ConvertNode(pCurrent->m_pLeft, pLastNodeInList);

    pCurrent->m_pLeft = *pLastNodeInList;
    if(*pLastNodeInList != NULL)
        (*pLastNodeInList)->m_pRight = pCurrent;

    *pLastNodeInList = pCurrent;

    if (pCurrent->m_pRight != NULL)
        ConvertNode(pCurrent->m_pRight, pLastNodeInList);
}
```

In this code, `pLastNodeInList` always points to the last node with the maximal value in the converted double-linked list. When the root node with value 8 is visited, the left subtree has been converted and `pLastNodeInList` points to the node with 7. The root node is linked to the list, and then `pLastNodeInList` points the root node with value 8. It takes `pLastNodeInList` as a parameter to continue converting the right subtree, and `pLastNodeInList` is connected to the node with value 9.

Based on Node Rotations

Node rotations are fundamental operations to maintain self-balancing binary trees, such as red-black trees. Let's borrow the ideas of node rotations to convert a binary search tree to a sorted double-linked list.

The conversion process begins from the root node. If the root node of a binary search tree has a left child, a right rotation occurs. As shown in Figure 6-19(b), the tree is rotated clockwise at the root node, 8. After the right rotation, node 6 (the left child of the original root node) becomes the new root, and node 7 (the right child of node 6) becomes the left child of node 8. Other links in the tree remain unchanged.

The tree is rotated clockwise at the new root node because it still has a left child. The left child node becomes the new root with a right rotation (Figure 6-19(c)). Since the new root, node 5, does not have a left child, no more rotations at the root are needed now. It traverses the tree along links to right children until the first node with a left child is found, that is, the node with value 8.

It rotates clockwise at node 8 (Figure 6-19(d)). Node 7, the left child of node 8, becomes the parent of node 8, and it is also linked to a right child of node 6, which is the original parent of node 8.

It continues to traverse on the tree along links to right children, and another node with left child (node 10) is found. The operation to rotate at node 10 is similar to the preceding steps, as shown in (Figure 6-19 (e)).

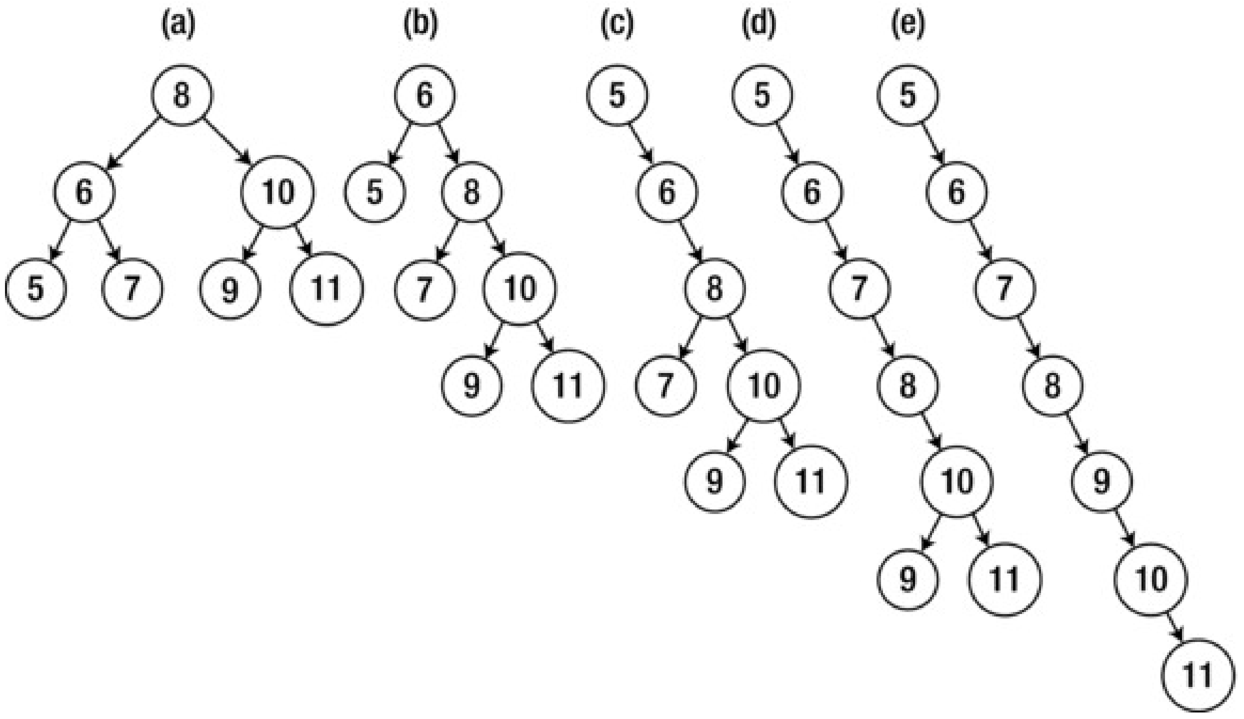

Figure 6-19. *The rotation process to convert a binary search tree to a sorted list. (a) The original binary search tree. (b) The tree with a right rotation at node 8. (c) The tree with a right rotation at node 6. (d) The tree with a right rotation at node 8. (e) The tree becomes a list with a right rotation at node 10.*

As shown in Figure 6-19(e), all nodes in the tree are linked as a single-linked list. Each child node is linked as a right child of its parent and links to left children are `NULL` pointers. In order to get a double-linked list, the last step is to link the parent node in the tree as the preceding node in the list.

The whole process to convert a binary search tree to a double-linked list can be implemented with C++, as shown in Listing 6-21.

Listing 6-21. *C++ Code to Convert a BSTree to a Double-Linked List (Version 2)*

```
BinaryTreeNode* Convert_solution2(BinaryTreeNode* pRoot) {
    BinaryTreeNode* pNode = pRoot;

        BinaryTreeNode* pHead = NULL;
    BinaryTreeNode* pParent = NULL;

    while(pNode != NULL) {
        while(pNode->m_pLeft != NULL) {
            // right rotation
            BinaryTreeNode* pLeft = pNode->m_pLeft;
            pNode->m_pLeft = pLeft->m_pRight;
            pLeft->m_pRight = pNode;
            pNode = pLeft;

            if(pParent != NULL)
                pParent->m_pRight = pNode;
        }

        if(pHead == NULL)
            pHead = pNode;
        pParent = pNode;

        pNode = pNode->m_pRight;

    }


    // build double-linked list

    BinaryTreeNode* pNode1 = pHead;

    if(pNode1 != NULL) {

        BinaryTreeNode* pNode2 = pNode1->m_pRight;

        while(pNode2 != NULL) {

            pNode2->m_pLeft = pNode1;


            pNode1 = pNode2;

            pNode2 = pNode2->m_pRight;

        }

    }


    return pHead;

}
```

Source Code:

```
    064_ConvertBinarySearchTree.cpp
```

Test Cases:

- Functional Cases (Normal binary search trees, including full binary search trees)
- Boundary Cases (Special binary search trees, such as a tree with only one node, or all nodes in a binary search tree only have left/right subtrees)
- Cases for Robustness (The pointer to the root node of a binary search tree is `NULL`)

Permutation and Combination

Question 65 Please print all permutations of a given string. For example, print "abc", "acb", "bac", "bca", "cab", and "cba" when given the input string "abc".

For many candidates, it is not a simple problem to get all permutations of a set of characters. In order to solve such a problem, candidates might try to divide it into simple subproblems. An input string is partitioned into two parts. The first part only contains the first characters, and the second part contains others. As shown in Figure 6-20, a string is divided into two parts with different background colors.

This solution gets permutations of a given string with two steps. The first step is to swap the first character with the following characters one by one. The second step is to get permutations of the string excluding the first character. Take the sample string "abc" as an example. It gets permutations of "bc" when the first character is 'a'. It then swaps the first character with 'b', gets permutations of "ac", and finally gets permutation of "ba" after swapping the first character with 'c'.

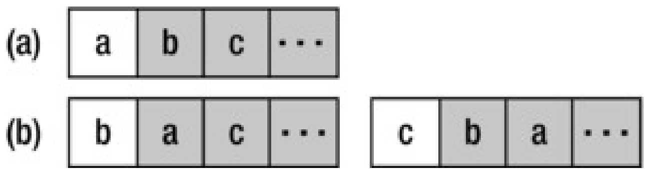

Figure 6-20. *The process to get permutations of a string. (a) A string is divided into two parts of which the first part only contains the first character, and the second part contains others (with gray background). (b) All characters in the second part are swapped with the first character one by one.*

The process to get permutations of a string excluding the first character is similar to the process to get permutations of a whole string. Therefore, it can be solved recursively, as shown in Listing 6-22.

Listing 6-22. *C Code to Get Permutations of a String*

```
   void Permutation(char* pStr) {

    if(pStr == NULL)

        return;


    PermutationCore(pStr, pStr);

}


void PermutationCore(char* pStr, char* pBegin) {
    char *pCh = NULL;
    char temp;

    if(*pBegin == '\0') {
        printf("%s\n", pStr);
    }
    else {
        for(pCh = pBegin; *pCh != '\0'; ++ pCh) {
            temp = *pCh;
               *pCh = *pBegin;
            *pBegin = temp;

            PermutationCore(pStr, pBegin + 1);

            temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;
        }
    }
}
```

Source Code:

```
065_StringPermutation.c
```

Test Cases:

- Functional Cases (The input string contains one or more characters)
- Cases for Robustness (The string is empty; the pointer to the string is `NULL`)

Question 66 How many distinct ways are available to place eight queens on a chessboard, where there are no two queens that can attack each other? That is to say, there are no two queens located at the same row, same column, or same diagonal.
For example, a solution in Figure 6-21 places eight queens on an 8×8 chessboard where any queen cannot attack another.

An array `columnIndex` [8] is defined, of which the i^{th} number stands for the column index of the queen at the i^{th} row. The eight numbers in the array are initialized as numbers 0, 1, 2, 3, 4, 5, 6, 7, and then we try to get all permutations of the array. Since the array is initialized with eight different numbers, any two queens are in different columns. It is only necessary to check whether there are two queens on the same diagonal.

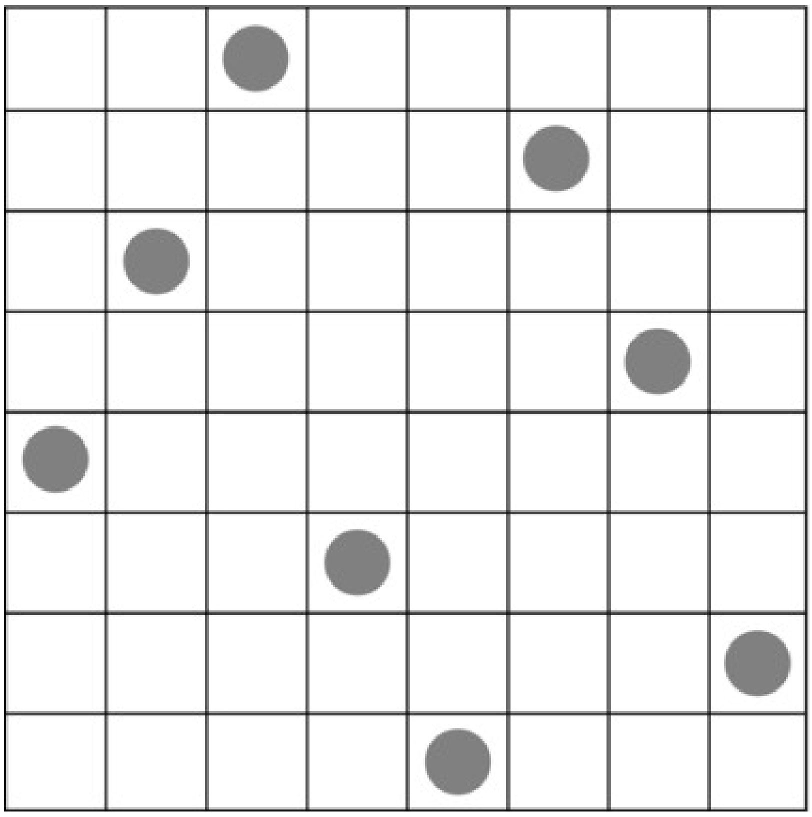

Figure 6-21. *Place eight queens (denoted as dots) on a chessboard where no two queens attack each other*

Therefore, the eight-queen puzzle can be solved with the code in Listing 6-23.

Listing 6-23. *C Code for the Eight Queens Puzzle*

```
   int EightQueen() {

    int columnIndex[8] = {0, 1, 2, 3, 4, 5, 6, 7};

    int count = 0;

    Permutation(columnIndex, 8, 0, &count);


    return count;

}


void Permutation(int columnIndex[], int length, int index, int* count) {

    int i, temp;


    if(index == length) {

        if(Check(columnIndex, length) != 0) {

            (*count)++;

        }

    }

    else {

        for(i = index; i < length; ++ i) {

            temp = columnIndex[i];

            columnIndex[i] = columnIndex[index];

            columnIndex[index] = temp;


            Permutation(columnIndex, length, index + 1, count);


            temp = columnIndex[index];

            columnIndex[index] = columnIndex[i];

            columnIndex[i] = temp;

        }

    }

}
```

```
/* If there are two queens on the same diagonal, it returns 0,
   otherwise it returns 1. */
int Check(int columnIndex[], int length) {
    int i, j;

    for(i = 0; i < length; ++ i) {
        for(j = i + 1; j < length; ++ j) {
            if((i - j == columnIndex[i] - columnIndex[j])
                || (j - i == columnIndex[i] - columnIndex[j]))
            return 0;
        }
    }

    return 1;
}
```

Source Code:

```
066_EightQueens.c
```

Question 67 There are *n* arrays. A permutation is generated when an element is selected from each array. How do you generate all permutations of *n* arrays? For example, permutations for the 3 arrays {1, 2}, {3, 4}, {5, 6} are: {1, 3, 5}, {1, 3, 6}, {1, 4, 5}, {1, 4, 6}, {2, 3, 5}, {2, 3, 6}, {2, 4, 5}, and {2, 4, 6}.

Similar to finding permutations of a string, it divides the *n* input arrays into two parts: the first array and the remaining *n*-1 arrays. After an element is selected from the first array, it continues to find permutations of the remaining arrays. The process needed to find permutations of the next *n*-1 arrays is identical to the process to find permutations of all *n* arrays, so it can be solved recursively, as shown in Listing 6-24.

Listing 6-24. *Java Code to Permute Arrays*

```
   void permute(ArrayList<int[]> arrays) {
    Stack<Integer> permutation = new Stack<Integer>();
    permuteCore(arrays, permutation);
}

void permuteCore(ArrayList<int[]> arrays, Stack<Integer> permutation) {
    if(permutation.size() == arrays.size()) {
        System.out.println(permutation);
        return;
    }

    int[] array = arrays.get(permutation.size());
    for(int i = 0; i < array.length; ++i) {
        permutation.push(array[i]);
        permuteCore(arrays, permutation);
        permutation.pop();
    }
}
```

Source Code:

```
067_ArrayPermutation.java
```

Test Cases:

- Functional Cases (Some normal arrays)
- Boundary Cases (There is only one array; there is only one number in each array)

Question 68 Please generate all combinations of a given string. For example, combinations of a given string "abc" are "a", "b", "c", "ab", "ac", "bc", and "abc".

Based on Divide and Conquer

Suppose the length of a given string is *n* and we are going to find one of its combinations with *m* characters. All characters are divided into two parts: the first character and the other *n*-1 characters. There are two choices for the first characters. The first choice is to insert the first character in the combination, and it continues to select *m*-1 characters from the remaining *n*-1 characters in the string. The other choice is to ignore the first character and select *m* characters from the next *n*-1 characters. Therefore, combinations of a given string can be generated recursively, as shown in Listing 6-25.

Listing 6-25. *Java Code for String Combinations (Version 1)*

```
 void combination(String str) {

  Stack<Character> result = new Stack<Character>();

  for(int i = 1; i <= str.length(); ++ i) {

      combination(str, 0, i, result);

  }

}


void combination(String str, int index, int number, Stack<Character> result) {

    if(number == 0) {

        System.out.println(result);

        return;

    }


    if(index == str.length())

        return;


    // select the character str[index]

    result.push(str.charAt(index));

    combination(str, index + 1, number - 1, result);

    result.pop();


    // ignore the character str[index]

    combination(str, index + 1, number, result);

}
```

Since a combination may contain one characters, two characters, …, or *n* characters for a given string with length *n*, there is a loop in the method `combination(String str)` .

Based on Bit Operations

Some characters are selected that compose a combination. A set of bits is utilized, where each bit stands for a character in the string. If the i^{th} character is selected for a combination, the i^{th} bit is `true` ; otherwise, it is `false` . For instance, three bits are used for combinations of the string "abc". If the first two characters 'a' and 'b' are selected to compose a combination "ab", the corresponding bits are {1, 1, 0}. Similarly, bits corresponding to another combination "ac" are {1, 0, 1}. We are able to get all combinations of a string with length *n* if we can get all possible combinations of *n* bits.

A number is composed of a set of bits. All possible combinations of *n* bits correspond to numbers from 1 to 2^n-1. Therefore, each number in the range between 1 and 2^n-1 corresponds to a combination of a string with length *n*. For example, the number 6 is composed of bits {1, 1, 0}, so the first and second characters are selected in the string "abc" to generate the combination "ab". Similarly, the number 5 with bits {1, 0, 1} corresponds to the combination "ac".

A new solution is found that generates combinations with *n* characters based on a set of bits corresponding to numbers from 1 to 2^n-1. The sample code is shown in Listing 6-26.

Listing 6-26. *Java Code for String Combinations (Version 2)*

```
 void combination_solution2(String str) {

  BitSet bits = new BitSet(str.length());

  while(increment(bits, str.length()))

      print(str, bits);

}


boolean increment(BitSet bits, int length) {

    int index = length - 1;


    while(index >= 0 && bits.get(index)) {

        bits.clear(index);

        --index;

    }
    if(index < 0)
        return false;

    bits.set(index);
    return true;
}

void print(String str, BitSet bits) {
    for(int i = 0; i < str.length(); ++i) {
        if(bits.get(i))
            System.out.print(str.charAt(i));
    }

    System.out.println();
}
```

The method `increment` increases a number represented in a set of bits. The algorithm clears 1 bits from the rightmost bit until a 0 bit is found. It then sets the rightmost 0 bit to 1. For example, in order to increase the number 5 with bits {1, 0, 1}, it clears 1 bits from the right side and sets the rightmost 0 bit to 1. The bits become {1, 1, 0} for the number 6, which is the result of increasing 5 by 1.
Source Code:

`068_StringCombination.java`

Test Cases:

- Functional Cases (Some normal strings)
- Boundary Cases (An empty string)