CHAPTER 3

Data Structures

The primary purpose of most computer programs is to store, retrieve, and process information. Consequently, data structures and algorithms that manipulate information are at the heart of computer science. For this reason, many interviewers often focus a number of their questions on these aspects. This chapter will cover typical interview questions about the most common data structures, including arrays, strings, lists, trees, stacks, and queues.

Arrays

Arrays might be the most simple data structure. Elements are sequentially stored in continuous memory in arrays. When an array is created, its size should be specified. Even though it may only store one element at first, the size is required because we have to allocate memory for all of the elements. Since there may be vacancies in arrays, they are not efficient in memory utilization.

In order to improve space efficiency, dynamic arrays were developed. The class `vector` in the standard template library (STL) of C++ is one such example. Memory is allocated for a few elements in dynamic arrays at first. When the number of elements is greater than the capacity of a dynamic array, more memory is allocated (the capacity doubles when it has to enlarge the capacity of a `vector` in C++), existing elements are copied to the newly allocated space, and the previous memory is released. It reduces waste in memory, but many extra operations are required to enlarge capacity, so it has negative impact on time efficiency. Therefore, it is better to reduce the times needed to enlarge the capacity of dynamic arrays. The type `ArrayList` in both C# and Java is similar to `vector` in C++.

Because memory allocation for arrays is sequential, it only costs O(1) time to access to an element based on its index, and it is very efficient. A simple hash table can be implemented with an array to utilize its advantage of time efficiency. Each index is treated as a key and every element in an array is treated as a value, so an index and its corresponding element form a pair of key and value. Many problems can be solved with such a hash table, and examples are illustrated in the section *Hash Tables for Characters*. It is a practical solution especially when built-in hash tables are not available in some programming languages such as C/C++.

Arrays and pointers are closely related to each other in C/C++ and also different from each other. The C code in Listing 3-1 shows the relationship between them. What is the output of this code?

Listing 3-1. *C Code about Arrays and Pointers*

```
int GetSize(int data[]) {
    return sizeof(data);
}

int _tmain(int argc, _TCHAR* argv[]) {
    int data1[] = {1, 2, 3, 4, 5};
    int size1 = sizeof(data1);

    int* data2 = data1;
    int size2 = sizeof(data2);

    int size3 = GetSize(data1);

    printf("%d, %d, %d", size1, size2, size3);
}
```

The output should be "20, 4, 4" in a 32-bit system. `data1` is an array, and `sizeof(data1)` gets its size. There are five integers in the array, and each integer occupies four bytes, so the total size of array is 20 bytes.

The name of an array is the address of the first element in the array, so `data2` points to the first element of an array with the statement `data2 = data1`. `data2` is declared as a pointer, and the `sizeof` operator returns 4 for any pointers in a 32-bit system.

When an array is passed as a parameter in C/C++, the compiler treats it as a pointer. What gets passed is the address of the first element in an array, rather than the whole array. Therefore, the result of `sizeof(data)` is also 4 in the function `GetSize` even though `data` is declared as an array in the parameter list.

Duplication in an Array

■ **Question 5** An array contains *n* numbers ranging from 0 to *n*-2. There is exactly one number duplicated in the array. How do you find the duplicated number? For example, if an array with length 5 contains numbers {0, 2, 1, 3, 2}, the duplicated number is 2.

Suppose that the duplicated number in the array is *m*. The sum of all numbers in the array, denoted as *sum1*, should be the result of 0+1+…+(*n*-2)+*m*. It is not difficult to get the sum result of 0+1+…+(*n*-2), which is denoted as *sum2*. The duplicated number *m* is the difference between *sum1* and *sum2*. The corresponding code in Java is shown in Listing 3-2.

Listing 3-2. Java Code to Get a Duplicated Number in an Array

```
int duplicate(int numbers[]) {
    int length = numbers.length;

    int sum1 = 0;
    for(int i = 0; i < length; ++i) {
        if(numbers[i] < 0 || numbers[i] > length - 2)
            throw new IllegalArgumentException("Invalid numbers.");

        sum1 += numbers[i];
    }

    int sum2 = ((length - 1) * (length - 2)) >> 1;

    return sum1 - sum2;
}
```

Source Code:

```
    005_Duplication.java
```

Test Cases:

- Normal case: an array with size *n* has a duplication
- Boundary case: an array {0, 0} with size 2
- Some numbers are out of the range of 0 to *n*-2 in an array of size *n*

■ **Question 6** An array contains *n* numbers ranging from 0 to *n*-1. There are some numbers duplicated in the array. It is not clear how many numbers are duplicated or how many times a number gets duplicated. How do you find a duplicated number in the array? For example, if an array of length 7 contains the numbers {2, 3, 1, 0, 2, 5, 3}, the implemented function (or method) should return either 2 or 3.

A naive solution for this problem is to sort the input array because it is easy to find duplication in a sorted array. As we know, it costs O(*n*log*n*) time to sort an array with *n* elements.

Another solution is the utilization of a hash set. All numbers in the input array are scanned sequentially. When a number is scanned, we check whether it is already in the hash set. If it is, it is a duplicated number. Otherwise, it is inserted into the set. The data structure `HashSet` in Java is quite helpful in solving this problem. Even though this solution is simple and intuitive, it has costs: O(*n*) auxiliary memory to accommodate a hash set. Let's explore a better solution that only needs O(1) memory.

Indexes in an array with length *n* are in the range 0 to *n*-1. If there were no duplication in the *n* numbers ranging from 0 to *n*-1, we could rearrange them in sorted order, locating the number *i* as the i^{th} number. Since there are duplicate numbers in the array, some locations are occupied by multiple numbers, but some locations are vacant.

Now let's rearrange the input array. All numbers are scanned one by one. When the i^{th} number is visited, first it checks whether the value (denoted as *m*) is equal to *i*. If it is, we continue to scan the next number. Otherwise, we compare it with the m^{th} number. If the i^{th} number equals the m^{th} number, duplication has been found. If not, we locate the number *m* in its correct place, swapping it with the m^{th} number. We continue to scan, compare, and swap until a duplicated number is found.

Take the array {2, 3, 1, 0, 2, 5, 3} as an example. The first number 2 does not equal its index 0, so it is swapped with the number with index 2. The array becomes {1, 3, 2, 0, 2, 5, 3}. The first number after swapping is 1, which does not equal its index 0, so two elements in the array are swapped again and the array becomes {3, 1, 2, 0, 2, 5, 3}. It continues to swap since the first number is still not 0. The array is {0, 1, 2, 3, 2, 5, 3} after swapping the first number and the number with index 3. Finally, the first number becomes 0.

Let's move on to scan the next numbers. Because the following three numbers, 1, 2 and 3, are all equal to their indexes, no swaps are necessary for them. The following number, 2, is not the same as its index, so we check whether it is the same as the number with index 2. Duplication is found since the number with index 2 is also 2.

With an understanding of the detailed step-by-step analysis, it is time to implement code. Sample code in Java is shown in Listing 3-3.

Listing 3-3. Java Code to Get a Duplicated Number in an Array

```java
int duplicate(int numbers[]) {
    int length = numbers.length;
    for(int i = 0; i < length; ++i) {
        if(numbers[i] < 0 || numbers[i] > length - 1)
            throw new IllegalArgumentException("Invalid numbers.");
    }

    for(int i = 0; i < length; ++i) {
        while(numbers[i] != i) {
            if(numbers[i] == numbers[numbers[i]]) {
                return numbers[i];
            }

            // swap numbers[i] and numbers[numbers[i]]
            int temp = numbers[i];
            numbers[i] = numbers[temp];
            numbers[temp] = temp;
        }
    }

    throw new IllegalArgumentException("No duplications.");
}
```

It throws two exceptions in the code to make the code complete and robust. If there are any numbers out of the range between 0 and *n*-1, the first exception is thrown. If there is no duplication in the array, the second exception is thrown. It is important for candidates to write complete and robust code during interviews.

Source Code:

```
006_Duplication.java
```

Test Cases:

- Normal cases: an array with size *n* has one or more duplicated numbers
- Boundary cases: the array {0, 0} with size 2
- Some numbers are out of the range from 0 to *n*-1 in an array of size *n*
- No duplication in the array

Search in a 2-D Matrix

■ **Question 7** In a 2-D matrix, every row is increasingly sorted from left to right, and the last number in each row is not greater than the first number of the next row. A sample matrix follows. Please implement a function to check whether a number is in such a matrix or not. It returns `true` if it tries to find the number 7 in the sample matrix, but it returns `false` if it tries to find the number 12.

```
1  3  5
7  9  11
13 15 17
```

There are many solutions for this problem. The naive solution with brute force is to scan all numbers in the input matrix. Obviously, it costs O(*mn*) time if the size of the matrix is $m \times n$.

Since each row in the matrix is sorted and the first number of a row is guaranteed to be greater than or equal to the last number of the preceding row, the matrix can be viewed as a 1-D sorted array. If all rows in the sample matrix are concatenated in top down order, it forms a sorted array {1, 3, 5, 7, 9, 11, 13, 15, 17}. The binary search algorithm is suitable for such a scenario, as shown in Listing 3-4.

Listing 3-4. Java Code to Search in a Sorted Matrix

```
boolean find(int matrix[][], int value) {
    int rows = matrix.length;
    int cols = matrix[0].length;
    int start = 0;
    int end = rows * cols - 1;

    while (start <= end) {
        int mid = start + (end - start) / 2;
        int row = mid / cols;
        int col = mid % cols;
        int v = matrix[row][col];

        if (v == value)
            return true;

        if (v > value)
            end = mid - 1;
        else
            start = mid + 1;
    }

    return false;
}
```

If there are *m* rows and *n* columns in a matrix, the time efficiency for the binary search algorithm is O(log*mn*).

Source Code:

`007_FindInSortedMatrix.Java`

Test Cases:

- The matrix contains the target value (including cases where the target value is the maximum or minimum in the matrix)
- The matrix does not contain the target value (including cases where the target is larger than the maximum or less than the minimum)
- Special matrices, including matrices with only one row, only one column, or with only one element

---

■ **Question 8** In a 2-D matrix, every row is increasingly sorted from left to right, and every column is increasingly sorted from top to bottom. Please implement a function to check whether a number is in such a matrix or not. For example, all rows and columns are increasingly sorted in the following matrix. It returns `true` if it tries to find number 7, but it returns `false` if it tries to find number 5.

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

Different from the previous problem, the first number in a row may be less than the last number of the preceding row. For instance, the first number in the second row (the number 2) is less than the last number of the first row (the number 9). Therefore, we cannot utilize the binary search algorithm on the 2-D matrix as a whole.

Since each row is sorted, it improves efficiency if a binary search is utilized. It costs O(log*n*) time for a binary search on *n* numbers, so the overall time efficiency is O(*m*log*n*) for an *m*×*n* matrix. This solution does not fully take advantage of characteristics of an input matrix: all rows are sorted, and all columns are also sorted. More efficient solutions can be found if we fully utilize these characteristics.

Binary Search on a Diagonal

Because all rows and all columns in an input matrix are sorted, numbers on the diagonal from the top left corner to the bottom right corner are also sorted. Therefore, the binary search algorithm can be applied on numbers on the diagonal. If the target value is on the diagonal, it is done. Otherwise, it gets the greatest number on the diagonal that is less than the target value.

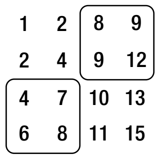

Figure 3-1. *Find 7 in a 2-D Matrix. The number 4 on the diagonal is located first, which is the greatest number less than the target 7. The whole matrix is split into four sub-matrices by the number 4, and it continues to search in two of them in the rounded rectangles.*

The greatest number that is less than the target value on the diagonal splits the whole matrix into four sub-matrices, and it continues to search in two of them. Searching in a sub-matrix is similar to searching in a matrix, so it can be solved recursively.

As shown in Figure 3-1, the target value 7 is not on the diagonal from the top left corner to the bottom right corner. The greatest number less than 7 on the diagonal (the number 4) is found first. Since the target number is greater than 4, it may appear in the northeast area or southwest area (numbers in rounded rectangle). All numbers in the northwest area should be less than 7, and numbers in the southeast area should be greater than 7.

Listing 3-5 provides sample code in Java based on the solution just discussed.

Listing 3-5. *Java Code to Search in a Partially Sorted Matrix (Version 1)*

```
boolean find_solution1(int matrix[][], int value) {
    int rows = matrix.length;
    int cols = matrix[0].length;
    return findCore(matrix, value, 0, 0, rows - 1, cols - 1);
}

boolean findCore(int matrix[][], int value, int row1, int col1, int row2, int col2) {
    if(value < matrix[row1][col1] || value > matrix[row2][col2])
        return false;

    if(value == matrix[row1][col1] || value == matrix[row2][col2])
        return true;

    int copyRow1 = row1, copyRow2 = row2;
    int copyCol1 = col1, copyCol2 = col2;

    int midRow = (row1 + row2) / 2;
    int midCol = (col1 + col2) / 2;

    // find the last element less than value on diagonal
    while((midRow != row1 || midCol != col1)
            && (midRow != row2 || midCol != col2)) {
        if(value == matrix[midRow][midCol])
            return true;

        if(value < matrix[midRow][midCol]) {
            row2 = midRow;
            col2 = midCol;
        }
        else {
            row1 = midRow;
            col1 = midCol;
        }

        midRow = (row1 + row2) / 2;
        midCol = (col1 + col2) / 2;
    }

    // find value in two sub-matrices
    boolean found = false;
    if(midRow < matrix.length - 1)
        found = findCore(matrix, value, midRow + 1, copyCol1, copyRow2, midCol);
    if(!found && midCol < matrix[0].length - 1)
        found = findCore(matrix, value, copyRow1, midCol + 1, midRow, copyCol2);

    return found;
}
```

If the diagonal length of the input matrix is l, the time efficiency of the solution can be calculated with the equation $T(l)=2T(l/2)+\log l$. According to the master theory, $T(l)=O(l)$. Additionally, the length l in a $m\times n$ matrix is $l=O(m+n)$. Therefore, the time efficiency of the solution above is $O(m+n)$.

Removing a Row or a Column at Each Step

When complicated problems are encountered during interviews, an effective method is to use examples to simplify complexity. We can also start from some examples to solve this problem. Let's analyze the step-by-step process to find the number 7 out of the sample matrix in the problem description.

First, we choose the number 9 at the top right corner of the matrix. Since 9 is greater than 7, and 9 is the first number, also the least one, in the fourth column, all numbers in the fourth column should be greater than 7. Therefore, it is not necessary to search in the last column any more, and it is safe to just focus on the other three columns, as shown in Figure 3-2(a).

The number at the top right corner of the remaining matrix is 8, which is also greater than 7, so the third column can also be removed. Let's just focus on the remaining two columns, as shown in Figure 3-2(b).

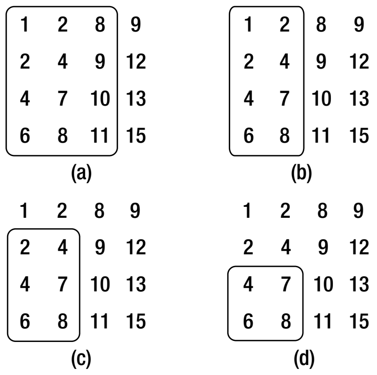

Figure 3-2. Find 7 in a 2-D Matrix. Numbers in the rounded rectangle are the focus of the next round of searching. (a) The number 9 at the top right corner is greater than the target value 7, so the column containing 9 is removed. (b) 8 is also greater than 7, so we remove the column containing 8. (c) The number 2 at the top right corner is less than 7, so the row containing 2 is removed. (d) 4 is less than 7 too, so the row containing 4 is removed.

The number 2 is at the top right corner of the remaining matrix with only two columns. Since 2 is less than 7, the target number 7 may be at the right side of 2 or below 2 according to the sorting rules. All columns at the right side of the number 2 have been removed, so it is safe to ignore them. Therefore, the target number 7 should be below the number 2, and the row containing 2 can also be removed. It is only necessary to search in a 3×2 sub-matrix (Figure 3-2(c)).

Similarly to the previous step, the row containing the number 4 can be removed as well because 4 is less than 7. The sub-matrix left with two rows and two columns is shown in Figure 3-2(d).

The number 7 at the top right corner of the remaining 2×2 sub-matrix equals the target number, so the target has been found and we stop searching here.

The following rules can be summarized based on the detailed analysis step-by-step. The number at the top right corner is selected in each round of searching, and it is compared with the target value. When it is the same as the target value, it stops to search. If it is greater than the target value, the last column in the remaining sub-matrix is removed. If it is less than the target value, the first row in the remaining sub-matrix is removed. Therefore, it reduces the sub-matrix by a row or a column if the target value is not at the top right corner.

It is not difficult to develop code after clearly understanding the searching process. Some sample code in Java is found in Listing 3-6.

Listing 3-6. Java Code to Search in a Partially Sorted Matrix (Version 2)

```
boolean find_solution2(int matrix[][], int value) {
    boolean found = false;
    int row = 0;
    int col = matrix[0].length - 1;

    while(row < matrix.length && col >= 0) {
        if(matrix[row][col] == value) {
            found = true;
            break;
        }
        if(matrix[row][col] > value)
            --col;
        else
            ++row;
    }

    return found;
}
```

Since a row or a column is removed in each round of searching, it costs O(*m*+*n*) time for a matrix with *m* rows and *n* columns.

In the previous analysis, the number at the top right corner is selected in each round of searching. Similarly, we can also select the number at the bottom left corner. Please try using the numbers at the bottom left corner if you are interested. However, numbers at the top left corner or bottom right corner are not appropriate choices. Let's take numbers at the top left corner as a quick example. The number 1 is at the top left corner of the original matrix. Since the target value 7 is greater than 1, it may be at the right side of 1 or be in rows below 1. Neither a row nor a column can be removed based on this comparison.

Source Code:

`008_FindInPatiallySortedMatrix.Java`

Test Cases:

- The matrix contains the target value (including cases where the target value is the maximum or minimum in the matrix)
- The matrix does not contain the target value (including cases where the target is larger than the maximum or less than the minimum)
- Special matrices, including matrices with only one row, only one column, or with only one element

String

A string, which is composed of a sequence of characters, is quite an important data structure. Many programming languages have special rules for strings for the optimization purpose, some of which are highlighted in C/C++, C#, and Java.

Strings in C/C++

All literal strings in C/C++ end with a special character '\0', so it is easy to find the end of a string. However, there is an extra cost for the special character, and it is easy to make mistakes. This is easy to see in Listing 3-7.

Listing 3-7. C/C++ Code for End of a String

```
char str[10];
strcpy(str, "0123456789");
```

A character array with length 10 is declared first, and the content of a string “0123456789” is copied into it. It seems that the string “0123456789” only has 10 characters, but its actual length is 11 because there is an extra character ‘\0’ at its end. The length should be at least 11 for a character array to accommodate the string.

Strings in C#

Strings are encapsulated in a class `System.String` in C#, whose contents are immutable. When we try to modify the content of a string, a new instance will be created. There are many interview questions about immutable strings. For example, what is the final content of `str` in the C# code in Listing 3-8?

Listing 3-8. C# Code for Immutable Strings

```
String str = "hello";
str.ToUpper();
str.Insert(0, " WORLD");
```

Although there are two operations, `ToUpper` and `Insert`, on `str`, it keeps the original content “hello” unchanged. When we try to modify the content of a string, the modified result is in return value.

If there are multiple editing operations on a string, multiple temporary instances will be created, and it has a negative impact on both time and space efficiencies. A new class related to string, `StringBuilder`, is defined to accommodate the modified result. Usually, `StringBuilder` is a better choice if we continue modifying strings many times.

Similar to editing strings, a new instance will be created when we try to assign a literal string to another string. An example is shown in Listing 3-9.

Listing 3-9. C# Code to Assign Strings

```
void ValueOrReference(Type type) {
    String result = "The type " + type.Name;

    if (type.IsValueType)
        Console.WriteLine(result + " is a value type.");
    else
        Console.WriteLine(result + " is a reference type.");
}

void ModifyString(String text) {
    text = "world";
}

void Main(string[] args) {
    String text = "hello";
    ValueOrReference(text.GetType());
    ModifyString(text);

    Console.WriteLine(text);
}
```

This example checks whether the class `String` is a value type and reference type first. Since it is defined as `public sealed class String {...}`, it is a reference type.

It assigns a new string “world” to `text` in the method `ModifyString`. A new string instance with content “world” is created here, and it is referenced by `text`. The variable `text` references the original string outside the method `ModifyString` because `text` is not marked as `ref` or `out` in the argument list. Therefore, the output for `text` is still “hello”. If the expected output is “world”, we have to mark the parameter `text` with `ref` or `out`.

Strings in Java

A section of memory is allocated for literal strings in Java. When a string is created once, it can be referenced by other instances. This optimization mechanism avoids recreation, but it makes identity and equality tests more complicated and confusing. Please read the code in Listing 3-10. What is the result?

Listing 3-10. Java Code for Equality and Identity of Strings

```
void testEquality() {
    String str1 = "Hello world.";
    String str2 = "Hello world.";

    if (str1 == str2)
        System.out.print("str1 == str2\n");
    else
        System.out.print("str1 != str2\n");

    if(str1.equals(str2))
        System.out.print("str1 equals to str2\n");
    else
        System.out.print("str1 doesn't equal to str2\n");

    String str3 = new String("Hello world.");
    String str4 = new String("Hello world.");

    if (str3 == str4)
        System.out.print("str3 == str4\n");
    else
        System.out.print("str3 != str4\n");

    if(str3.equals(str4))
        System.out.print("str3 equals to str4\n");
    else
        System.out.print("str3 doesn't equal to str4\n");
}
```

When the first line of code `String str1 = "Hello world."` executes, a string "Hello world." is created, and the variable `str1` references to it. Another string "Hello world." will not be created again when the next line of code executes because of optimization. The variable `str2` also references the existing "Hello world.".

The operator == checks the identity of the two objects (whether two variables reference the same object). Since `str1` and `str2` reference the same string in memory, they are identical to each other. The method `equals` checks equality of two objects (whether two objects have the same content). Of course, the contents of `str1` and `str2` are the same.

When code `String str3 = new String("Hello world.")` executes, a new instance of `String` with content "Hello world." is created and it is referenced by the variable `str3`. Then another instance of `String` with content "Hello world." is created again, and referenced by `str4`. Since `str3` and `str4` reference two different instances, they are not identical, but their contents are the same.

Therefore, the output contains four lines:

```
str1 == str2
str1 equals to str2
str3 != str4
str3 equals to str4
```

Replace Blanks in a String

■ **Question 9** Please implement a function to replace each blank in a string with "%20". For instance, it outputs "We%20are%20happy." if the input is "We are happy.".

If a URL parameter contains some special characters in web programming, such as blanks and '#'s, it may block servers from retrieving correct parameter information. Therefore, it is necessary to convert these characters into strings understandable by servers. The conversion rule is to append the ASCII value of a special character to a '%'. For example, the ASCII value for a blank is 32, 0x20 in hexadecimal, so a blank is converted to "%20". Take '#' as another example: its ASCII value is 35, 0x23 in hexadecimal, so it is replaced with "%23".

There are library methods to replace a piece of string with another, such as `String.Replace` in C# and `String.replaceAll` in Java. However, usually it is not the interviewers' intention to ask candidates to just call existing methods in libraries. They are more interested to know whether candidates have the ability to implement the replacement functionality.

A string with blanks will become longer if each blank is replaced with three characters: '%', '2', and '0'. If the replacement is on the original string, the longer converted string may overlap with memory behind the string. If it is allowed to create a new string and replace blanks on the new string, we can allocate enough memory to accommodate the longer converted string. Since there are two options, candidates should ask interviewers to clarify their requirements. Let's suppose the requirement is to replace blanks on the original string, and there is enough vacant memory behind it.

Replace from Left to Right in O(n^2) Time

An intuitive solution is to scan a string from beginning to end and replace each blank when it is met. Because a character is replaced with three, we must move characters behind blanks; otherwise, two characters will be overlapped.

Let's take a string "We are happy." as an example. We employ grids to visualize characters in a string, as shown in Figure 3-3(a).

When the first blank is replaced, the string becomes the content of Figure 3-3(b). We have to move the characters in the area in the light gray background. When the second blank is replaced, it becomes the content in Figure 3-3(c). Note that characters in "happy." are moved twice.

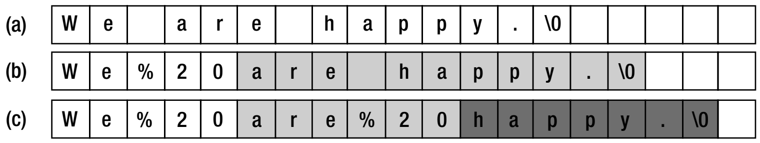
(a) W e a r e h a p p y . \0
(b) W e % 2 0 a r e h a p p y . \0
(c) W e % 2 0 a r e % 2 0 h a p p y . \0

Figure 3-3. Replace every blank in "We are happy." with "%20" from left to right. (a) This is the original string, "We are happy.". (b) Replace the first blank with "%20". It requires you to move characters with the light gray background. (c) Replace the second blank with "%20". It requires you to move characters with the dark gray background again.

Supposing the length of the string is n. O(n) characters are moved for each blank, so the total time efficiency is O(n^2) to replace O(n) blanks.

Interviewers may not be satisfied when they are told of this solution. What they expect is a more efficient solution. In the previous analysis, there are many characters moved multiple times. Is it possible to reduce the number of movements? Fortunately, the answer is yes. Let's have a try by replacing blanks from the end to the beginning.

Replace from Right to Left in O(*n*) Time

The number of blanks in the original string is gotten when it is scanned, and then the length of the converted string can also be gotten. The length increases by two when a blank is replaced, so the length after conversion is obtained by adding the original length to double the number of blanks. Take the string "We are happy." as an example again. Its length is 14 (including '\0'), and the length of the replaced string is 18 since there are two blanks in the original string.

Characters are copied or replaced from right to left at this time. Two pointers, P_1 and P_2, are declared. P_1 is initialized to the end of original string, and P_2 is initialized to the end of replaced string, as shown in Figure 3-4(a). The character pointed to by P_1 is copied to where P_2 points, and both pointers are moved backward until P_1 points to a blank (Figure 3-4(b)). Characters with the light gray background are copied. When a blank is met, P_1 moves backward one cell. Three characters "%20" are inserted to where P_2 points, and then P_2 moves backward three cells, as shown in Figure 3-4(c).

This process continues until it meets a blank again (Figure 3-4(d)). Similarly, P_1 moves backward one cell. Three characters "%20" are inserted to where P_2 points, and then P_2 moves backward three cells (Figure 3-4(e)). All blanks have been replaced because P_1 and P_2 overlap with each other.

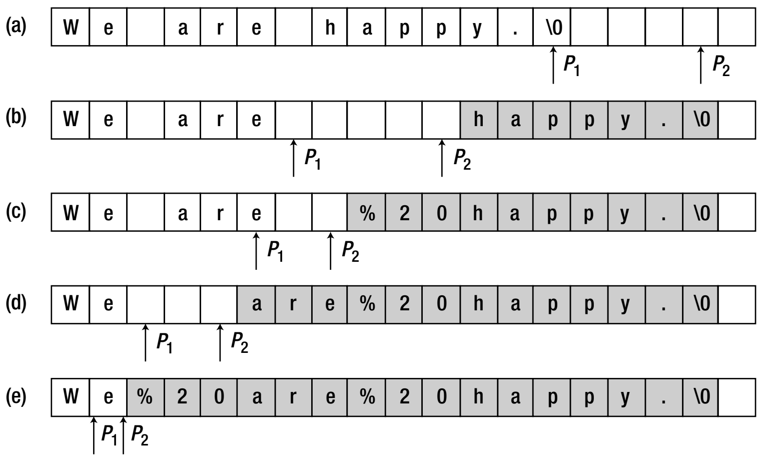

Figure 3-4. *Replace every blank in "We are happy." with "%20" backward. (a)* P_1 *is initialized to the end of the original string, and* P_2 *is initialized to the end of the replaced string. (b) Copy the character pointed to by* P_1 *to where* P_2 *points until the first blank is met. (c) "%20" is inserted where* P_2 *points.* P_1 *moves one cell, and* P_2 *moves three cells backward. (d) Copy the character pointed to by* P_1 *to where* P_2 *points until the second blank is met. (e) "%20" is inserted to where* P_2 *points.* P_1 *moves one cell, and* P_2 *moves three cells backward. All blanks have been replaced because* P_1 *overlaps with* P_2.

Note that all characters are copied (moved) once at most, so its time efficiency is O(n) and it is faster than the first solution.

It is time to implement code when your solution is accepted by interviewers. Some sample code is shown in Listing 3-11.

Listing 3-11. *C Code to Replace Blanks with "%20"*

```
/*capacity is total capacity of a string, which is longer than its actual length*/
void ReplaceBlank(char string[], int capacity) {
    int originalLength, numberOfBlank, newLength;
    int i, indexOfOriginal, indexOfNew;

    if(string == NULL || capacity <= 0)
        return;

    /*originalLength is the actual length of string*/
    originalLength = numberOfBlank = i = 0;
    while(string[i] != '\0') {
        ++ originalLength;

        if(string[i] == ' ')
            ++ numberOfBlank;

        ++ i;
    }
    /*newLength is the length of the replaced string*/
    newLength = originalLength + numberOfBlank * 2;
    if(newLength > capacity)
        return;

    indexOfOriginal = originalLength;
    indexOfNew = newLength;
    while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal) {
        if(string[indexOfOriginal] == ' ') {
            string[indexOfNew --] = '0';
            string[indexOfNew --] = '2';
            string[indexOfNew --] = '%';
        }
        else {
            string[indexOfNew --] = string[indexOfOriginal];
        }

        -- indexOfOriginal;
    }
}
```

Source Code:

```
    009_ReplaceBlanks.c
```

Test Cases:

- A string contains some blanks (including cases where some blanks are at the beginning or end of a string, or inside a string; some blanks are continuous)
- A string does not contain any blanks
- Special strings (such as empty strings, a string with only a blank, a string with only some continuous blanks, the input string pointer is `NULL`)

■ **Question 10** Given two sorted arrays, denoted as *array1* and *array2*, please merge them into *array1* and keep the merged array sorted. Suppose there is sufficient vacant memory at the end of *array1* to accommodate elements of *array2*.

All elements in an array are sequential, so some elements will be shifted when a new element is inserted. Supposing *array1* has *m* elements and *array2* has *n* elements. Since O(*m*) elements will be shifted when an element of *array2* is inserted to *array1*, it costs O(*mn*) time to merge two arrays via insertions.

Inspired by the solution of the previous problem, it is better to copy and move elements from right to left. The last two elements of these two arrays are compared, and the greater one is copied to the location with index (*m*+*n*-1). It continues to compare and copy until no numbers in *array2* are left. Sample code in C is shown in Listing 3-12.

Listing 3-12. C Code to Merge Sorted Arrays

```
// Supposing there is enough memory at the end of array1,
// in order to accommodate numbers in array2
void merge(int* array1, int length1, int* array2, int length2) {
    int index1, index2, indexMerged;

    if(array1 == NULL || array2 == NULL)
        return;

    index1 = length1 - 1;
    index2 = length2 - 1;
    indexMerged = length1 + length2 - 1;

    while(index1 >= 0 && index2 >= 0) {
        if(array1[index1] >= array2[index2])
            array1[indexMerged--] = array1[index1--];
        else
            array1[indexMerged--] = array2[index2--];
    }

    while(index2 >= 0)
        array1[indexMerged--] = array2[index2--];
}
```

Since only one element in *array1* or *array2* is copied and moved once in each step, the overall time complexity is O(*m*+*n*).

Source Code:

```
010_MergeSortedArrays.c
```

Test Cases:

- Merge two sorted arrays (including cases where there are duplicated numbers in two arrays)
- Special arrays (including cases where one or two pointers to arrays are `NULL`)

■ **Tip** If each element is shifted multiple times while merging two arrays (strings) from left to right, it may improve performance if elements are copied and moved from right to left.

String Matching

Regular expressions are an important topic in text processing, and many programming languages provide libraries to support them. For example, Java has a package `java.util.regex` and C# has a namespace `System.Text.RegularExpressions` for regular expressions. However, interviewers usually disallow candidates to employ library utilities to solve problems related to regular expressions, and candidates have to implement matching mechanisms to demonstrate their coding capabilities.

■ **Question 11** How do you implement a function to match regular expressions with '.' and '*' in patterns? The character '.' in a pattern matches a single character, and '*' matches zero or any number of characters preceding it. Matching means that a string fully matches the pattern where all characters in a string match the whole pattern. For example, the string "aaa" matches the pattern "a.a" and the pattern "ab*ac*a". However, it does not match the pattern "aa.a" nor "ab*a".

Our solution matches a character after another from a string and a pattern. Let's first analyze how to match a character. When the character *ch* in the pattern is a '.', it matches whatever character is in the string. If the character *ch* is not a '.' and the character in the string is *ch*, they match each other. When the first characters in a string and a pattern are matched, we continue to match the remaining string and pattern.

It is easy to match when the second character in the remaining pattern is not a '*'. If the first character in the remaining string matches the first character in the remaining pattern, it advances the string and pattern and continues to match next characters; otherwise, it returns `false` directly.

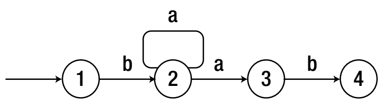
$$\begin{tikzcd} {} \arrow[r] & 1 \arrow[r, "b"] & 2 \arrow[loop above, "a"] \arrow[r, "a"] & 3 \arrow[r, "b"] & 4 \end{tikzcd}$$

Figure 3-5. *The nondeterministic finite automaton for the pattern ba*ab. There are two choices when it enters the state 2 with an input 'a': it advances to the state 3 or returns back to the state 2.*

It is more complex when the second character in the remaining pattern is a '*' because there might be multiple matching choices. One choice is to advance the pattern by two characters because a '*' may match zero characters in a string. In such a case, a '*' and its preceding character are ignored. If the first character in the remaining string matches the character before the '*' in the pattern, it may advance forward in the string and it has two choices in the pattern: it may advance the pattern by two characters or keep the pattern unchanged.

As shown in the nondeterministic finite automaton of Figure 3-5, it has two choices in the state 2 with an input 'a': it may advance to the state 3 (advance on the pattern by two characters) or return back to state 2 (keeping the pattern unchanged for the next round of matching).

The solution can be implemented based on recursion, as shown in Listing 3-13.

Listing 3-13. *C++ Code for Simple Regular Expression Matching*

```
bool match(char* string, char* pattern) {
    if(string == NULL || pattern == NULL)
        return false;

    return matchCore(string, pattern);
}
```

```
bool matchCore(char* string, char* pattern) {
    if(*string == '\0' && *pattern == '\0')
        return true;

    if(*string != '\0' && *pattern == '\0')
        return false;

    if(*(pattern + 1) == '*') {
        if(*pattern == *string || (*pattern == '.' && *string != '\0'))
                    // move on the next state
            return matchCore(string + 1, pattern + 2)
                    // stay on the current state
                || matchCore(string + 1, pattern)
                    // ignore a '*'
                || matchCore(string, pattern + 2);
        else
                    // ignore a '*'
            return matchCore(string, pattern + 2);
    }

    if(*string == *pattern || (*pattern == '.' && *string != '\0'))
        return matchCore(string + 1, pattern + 1);

    return false;
}
```

Source Code:

`011_SimpleRegularExpression.cpp`

Test Cases:

- A string matches or does not match a pattern (including cases where there are '.' and/or '*' in the pattern)
- Special inputs (including cases where a string or pattern is empty, pointers to a sting or pattern are `NULL`, or an invalid pattern with '*' is at the beginning)

■ **Question 12** How do you check whether a string stands for a number or not? Numbers include positive and negative integers and floats.
For example, strings "+100.", "5e2", "-.123", "3.1416", and "-1E-16" stand for numbers, but "12e", "1a3.14", "1.2.3", "+-5", and "12e+5.4" do not.

A numeric string follows this format:

`[sign]integral-digits[.[fractional-digits]][e|E[sign]exponential-digits].`

Elements in square brackets '[' and ']' are optional. The *sign* element is a negative sign symbol ('-') or a positive sign symbol ('+'). There is only one leading sign at most. The *integral-digits* element is a series of digits ranging from 0 to 9 that specify the integral part of the number. It can be absent if the string contains the *fractional-digits* element, which is also a series of digits ranging from 0 to 9 that specify the fractional part of a number. The ‘e’ or ‘E’ character indicates that the value is represented in exponential (scientific) notation, which is another series of digits ranging from 0 to 9 specified in *exponential-digits*.

First, it checks whether the leading character is the positive or negative sign symbol, and then it moves on to check the following substring. If the number in the string is a float value, there is a floating point (‘.’). Additionally, there might be an exponential notation at the end of an integer or a float number if there is an ‘e’ or `E' in the string. The overall process to verify a numeric string is shown in Listing 3-14.

Listing 3-14. *C++ Code to Verify Numeric Strings*

```
bool isNumeric(char* string) {
    if(string == NULL)
        return false;

    if(*string == '+' || *string == '-')
        ++string;
    if(*string == '\0')
        return false;

    bool numeric = true;

    scanDigits(&string);
    if(*string != '\0') {
        // for floats
        if(*string == '.') {
            ++string;
            scanDigital(&string);

            if(*string == 'e' || *string == 'E')
                numeric = isExponential(&string);
        }
        // for integers
        else if(*string == 'e' || *string == 'E')
            numeric = isExponential(&string);
        else
            numeric = false;
    }

    return numeric && *string == '\0';
}
```

The function `scanDigits` scans a segment of a string that only contains digital characters ranging from `0' to `9', as implemented in Listing 3-15.

Listing 3-15. *C++ Code to Scan Digits*

```
void scanDigits(char** string) {
    while(**string != '\0' && **string >= '0' && **string <= '9')
        ++(*string);
}
```

The function `isExponential` is to verify exponential notation at the end of a string. Exponential notation begins with an `e' or `E' and may have a sign symbol following it. Therefore, the function `isExponential` can be implemented, as shown in Listing 3-16.

Listing 3-16. *C++ Code to Verify an Exponential Notation*

```
bool isExponential(char** string) {
    if(**string != 'e' && **string != 'E')
        return false;

    ++(*string);
    if(**string == '+' || **string == '-')
        ++(*string);

    if(**string == '\0')
        return false;

    scanDigits(string);
    return (**string == '\0') ? true : false;
}
```

Source Code:

```
    012_NumericStrings.cpp
```

Test Cases:

- Numeric/Non-numeric strings with/without a sign symbol
- Numeric/Non-numeric strings with/without some fractional digits
- Numeric/Non-numeric strings with/without exponential notation
- Special inputs (including cases where a string is empty, the input pointer to a sting is `NULL`)

Linked Lists

Arrays are quite useful in all programming languages. However, they also have some limitations. They require creating a new array with bigger size and copying the existing elements from the old array to the new one when the capacity of an array is overrun. Additionally, they have to shift some elements in an array when a new element is inserted because memory of an array is sequentially allocated. Such limitations can be overcome by dynamic structures, such as linked lists.

It is not necessary to know the size of a list when it is created, so it is treated as a dynamic data structure. Rather than allocate memory for all elements when a list is initialized, memory is allocated for each node on demand when it is inserted. The space efficiency of lists is better than arrays because there is no vacant memory in lists.

Memory allocation of a list is not continuous because nodes are inserted dynamically and their memory is not allocated at the same time. It costs O(n) time to get the i^{th} node in a list since it has to traverse nodes one by one starting from the head node. It only takes O(1) time to get the i^{th} element in an array. Therefore, time efficiency to search lists is not as good as for arrays.

Linked lists are the most frequently met data structures during interviews. It only takes about 20 lines of code to create a list, insert a node into a list, or delete a node from a list. Compared to other complex data structures, such as hash tables and graphs, lists are more suitable for interviews due to their moderate code size. Additionally, lots of pointer operations are required to handle a list. Candidates without qualified programming abilities cannot implement complete and robust code related to lists. Moreover, lists are also flexible and challenging interview questions can be constructed with them. Therefore, many interviewers like questions related to lists.

Most lists met during interviews are single-linked lists, where each node has a link to its successor. For example, “Print a List from Tail to Head” (Question 13), “Delete a Node from a List in O(1) Time” (Question 43) , “k^{th} Node from the End” (Question 47), “Reverse Lists” (Question 48), and “First Intersection Node of Two Lists” (Question 82) are all about single-linked lists.

Not only are single-linked lists popular for interviews, but other types of lists are also frequently met:

- Usually the tail node in a single-linked list does not have a successor. If every node in a finite list has a successor, a loop is formed. The section *Loop in List* discusses lists with loops.
- If there is also a link to a predecessor besides a link to a successor in each node of a list, it is a double-linked list. The interview question “Binary Search Trees and Double-Linked Lists” (Question 64) is in this category.
- A complex list is composed if each node has a link to any other node (including the node itself). Please refer to the interview question “Clone Complex Lists” (Question 54) for more details on the complex list.

Print Lists from Tail to Head

■ **Question 13** Please implement a function to print a list from its tail to head.

When meeting this question, many candidates' intuition is to reverse links between all nodes and to reverse the direction of a list. It fulfills the requirement if all nodes in the reversed list are printed from head to tail. However, it has to modify the structure of the input list. Is it OK to do so? It depends on the requirement in an interviewers’ minds. You should ask your interviewers to clarify their requirement before you describe your solution.

■ **Tip** If you are going to modify the input data, you should ask interviewers whether it is allowed to do so.

Usually, printing is a read-only operation, and it is not allowed to modify the content to be printed. Supposing interviewers disallow to reverse the direction of lists here.

Nodes in a list are linked from the head to the tail, but the printing order is from the tail to the head. That is to say, the first node in the list is the last one to be printed, and the last node is the first one to be printed. It is typical “Last In, First Out,” so a stack can help to solve this problem. Every node is pushed into a stack when it is visited. After all nodes are visited and pushed into the stack, they are printed from the top of the stack and popped one by one. The printing order is opposite to the order in the list.

The sample code in Listing 3-17 is based on `stack` in the C++ standard template library.

Listing 3-17. *C++ Code to Print a List Reversely (Iterative Version)*

```
void PrintListReversingly_Iteratively(ListNode* pHead) {
    std::stack<ListNode*> nodes;
    ListNode* pNode = pHead;
    while(pNode != NULL) {
        nodes.push(pNode);
        pNode = pNode->m_pNext;
    }
    while(!nodes.empty()) {
        pNode = nodes.top();
        printf("%d\t", pNode->m_nValue);
        nodes.pop();
    }
}
```

It can be solved with a stack and since recursion is essentially equivalent to stacks, so it can also be solved recursively. To print a list in reverse, the next nodes are printed first when a node is visited, and then the currently visited node is printed. The recursive code is shown in Listing 3-18.

Listing 3-18. *C++ Code to Print a List Reversely (Recursive Version)*

```
void PrintListReversingly_Recursively(ListNode* pHead) {
    if(pHead != NULL) {
        if (pHead->m_pNext != NULL) {
            PrintListReversingly_Recursively(pHead->m_pNext);
        }
        printf("%d\t", pHead->m_nValue);
    }
}
```

The previous recursive code looks more concise, but it has a limitation: the recursive calls may have too many levels to cause call stack overflow errors when the list is very long. The iterative solution with an explicit stack is more robust. More discussion about recursion and iteration are available in the section *Recursion and Iteration.*

Source Code:

`013_PrintListsReversely.cpp`

Test Cases:

- Lists with multiple nodes or only one node
- The head node of a list is `NULL`

Sort Lists

■ **Question 14** Please implement a function to sort a given list.

The most efficient sorting algorithm in general, quicksort, is applicable to arrays because elements in arrays can be accessed in O(1) time based on indexes. It takes more time to locate a node in a list, so we have to utilize other algorithms to sort lists.

Let's take a list with four nodes as an example to analyze the process of the insert sort algorithm (Figure 3-6(a)). A list is split into two parts. The first part contains nodes already sorted, and the second part is not sorted yet. The node pointed by P_1 is the last sorted node, which is initialized to the first node, and the node pointed to by P_2 is the next one to be sorted.

When node 1 is the next node to be sorted, there is only one node (node 2) in the sorted list. Because the value 1 is less than the value in the list head, node 1 becomes the new head of the sorted list, and node 2 is linked to the next node of the previous node 1, which is node 4 (Figure 3-6(b)). Node 2 is still the last node in the sorted list, so it does not move P_1, and only moves P_2 to the next node of node 2.

The next node to be sorted is node 4. It traverses from the head node in order to find an appropriate location in the sorted list. Since the value 4 is greater than the value 2 in the last node of the sorted list, it is not necessary to relink nodes. Node 4 becomes the new last node of the sorted list, so it is pointed to by P_1 (Figure 3-6(c)).

Now the next node to be sorted is node 3. It traverses from the head node again in order to find an appropriate location in the sorted list. Node 3 is linked between node 2 and node 4. Node 4 is still the last node in the sorted list. Since there are no nodes left after node 4, the whole list is sorted (Figure 3-6(d)).

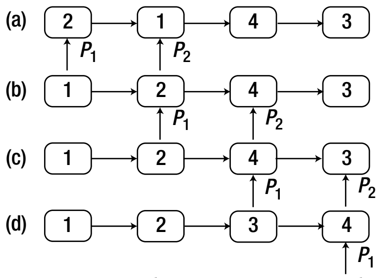

Figure 3-6. *The process to sort a list with four nodes.* P_1 *points to the last node in the sorted list, and* P_2 *points to the next node to be inserted into the sorted list, which always follows* P_1*. (a)* P_1 *is initialized to the first node, and* P_2 *is initialized to the second one. (b) The node with value 2 is inserted into the sorted list. (c) The node with value 4 is inserted into the sorted list. (d) The node with value 3 is inserted into the sorted list.*

The insert sort algorithm for lists can be implemented in C++, as shown in Listing 3-19.

Listing 3-19. *C++ Code to Sort a List*

```
void Sort(ListNode** pHead) {
    if(pHead == NULL || *pHead == NULL)
        return;
    ListNode* pLastSorted = *pHead;
    ListNode* pToBeSorted = pLastSorted->m_pNext;
    while(pToBeSorted != NULL) {
        if(pToBeSorted->m_nValue < (*pHead)->m_nValue) {
            pLastSorted->m_pNext = pToBeSorted->m_pNext;
            pToBeSorted->m_pNext = *pHead;
            *pHead = pToBeSorted;
        }
        else {
            ListNode* pNode = *pHead;
            while(pNode != pLastSorted
                && pNode->m_pNext->m_nValue < pToBeSorted->m_nValue) {
                pNode = pNode->m_pNext;
            }

            if(pNode != pLastSorted) {
                pLastSorted->m_pNext = pToBeSorted->m_pNext;
                pToBeSorted->m_pNext = pNode->m_pNext;
                pNode->m_pNext = pToBeSorted;
            }
            else
                pLastSorted = pLastSorted->m_pNext;
        }

        pToBeSorted = pLastSorted->m_pNext;
    }
}
```

Because it has to scan O(n) nodes on the sorted list in order to find an appropriate location to insert a new node, it costs O(n^2) time to sort a list with n nodes.

Source Code:

`014_SortLists.cpp`

Test Cases:

- Sort a list with multiple nodes, with/without duplicated values
- The input list has only one node
- The input head node of a list is `NULL`

■ **Question 15** Please implement a function to merge two sorted lists into a single sorted list. For example, the merged list of two sorted lists, L_1 and L_2 in Figure 3-7, is L_3.

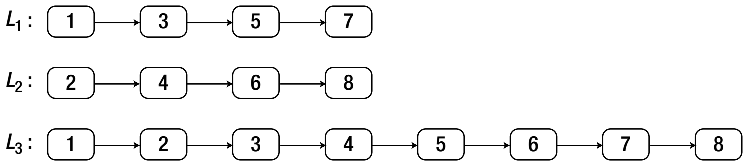

***Figure 3-7.** The lists* L_1 *and* L_2 *are two sorted lists. They compose the list* L_3 *when* L_1 *and* L_2 *are merged.*

The process begins to merge lists from head nodes because traversal on lists begins from head nodes. The head node of L_1 with value 1 is the head node of the merged list because its value is less than the value in the head of node of L_2, as shown in Figure 3-8(a).

It continues to merge the nodes remaining in L_1 and L_2 (the nodes in the rectangles with dashed edges in Figure 3-8). The steps to merge are similar to the ones before because nodes remaining in the two lists are still sorted. It continues to compare values in the head nodes of the two lists. Because the value in the head node of L_2 is less than the value of head node in the remaining L_1, it links the head node of L_2 to the tail of the merged list, as shown in Figure 3-8(b).

When the head node of one list with a lower value than the other is linked to the tail of the merged list, the nodes remaining in two lists are still increasingly sorted, so the steps to merge are the same as before. It is a typical recursive process, so let's implement it based on recursion.

It is important to know when to stop in recursion. It exits recursion when one of these two lists is empty. When L_1 is an empty list, the merged list is L_2. Similarly, the merged list is L_1 if L_2 is empty. If both lists are empty, the merged list is obviously empty.

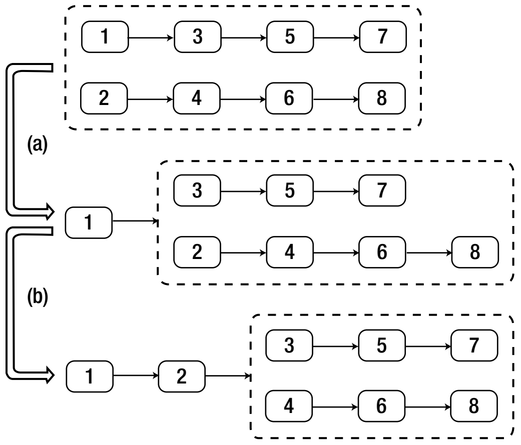

***Figure 3-8.** The process to merge two sorted lists. (a) The head of* L_1 *becomes the head of the merged list because the value in the head node of* L_1 *is less than the value in the head node of* L_2*. (b) The value in the head node of* L_2 *is less than the value in the head node of the remaining* L_1*, so the head node of* L_2 *is connected to the tail of the merged list.*

It is time to write code with careful analysis. A piece of sample code is shown in Listing 3-20.

Listing 3-20. C++ Code to Merge Sorted List

```
ListNode* Merge(ListNode* pHead1, ListNode* pHead2) {
    if(pHead1 == NULL)
        return pHead2;
    else if(pHead2 == NULL)
        return pHead1;

    ListNode* pMergedHead = NULL;

    if(pHead1->m_nValue < pHead2->m_nValue) {
        pMergedHead = pHead1;
        pMergedHead->m_pNext = Merge(pHead1->m_pNext, pHead2);
    }
    else {
        pMergedHead = pHead2;
        pMergedHead->m_pNext = Merge(pHead1, pHead2->m_pNext);
    }

    return pMergedHead;
}
```

Sorted lists can also be merged iteratively. Actually, the iterative solution is more robust with long lists. If you feel interested, please try to implement your own iterative version.

Source Code:

```
015_MergeSortedLists.cpp
```

Test Cases:

- There are/are not duplicated values in the two arrays
- One or two input list head nodes are `NULL`

Loop in List

■ **Question 16** How do you check whether there is a loop in a linked list? For example, the list in Figure 3-9 contains a loop.

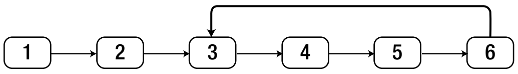

Figure 3-9. A list with a loop

This is a popular interview question. It can be solved with two pointers, which are initialized at the head of list. One pointer advances once at each step, and the other advances twice at each step. If the faster pointer meets the slower one again, there is a loop in the list. Otherwise, there is no loop if the faster one reaches the end of list.

The sample code in Listing 3-21 is implemented based on this solution. The faster pointer is `pFast`, and the slower one is `pSlow`.

Listing 3-21. *C++ Code to Check Whether a List Contains a Loop*

```
bool HasLoop(ListNode* pHead) {
    if(pHead == NULL)
        return false;

    ListNode* pSlow = pHead->m_pNext;
    if(pSlow == NULL)
        return false;

    ListNode* pFast = pSlow->m_pNext;
    while(pFast != NULL && pSlow != NULL) {
        if(pFast == pSlow)
            return true;

        pSlow = pSlow->m_pNext;

        pFast = pFast->m_pNext;
        if(pFast != NULL)
            pFast = pFast->m_pNext;
    }

    return false;
}
```

Source Code:

`016_LoopsInLists.cpp`

Test Cases:

- There is a loop in a list (including cases where there are one/multiple nodes in a loop, or a loop contains all nodes in a list)
- There is not a loop in a list
- The input node of the list head is `NULL`

■ **Question 17** If there is a loop in a linked list, how do you get the entry node of the loop? The entry node is the first node in the loop from the head of a list. For instance, the entry node of the loop in the list of Figure 3-9 is the node with value 3.

Inspired by the solution of the preceding problem, we can also solve this problem with two pointers.

Two pointers P_1 and P_2 are initialized to point to the head of a list. If there are *n* nodes in the loop, the first pointer move forward *n* steps, and then two pointers move together with same speed. When the second pointer reaches the entry node of the loop, the first one travels around the loop and returns back to the entry node.

Let's take the list in Figure 3-9 as an example. P_1 and P_2 are first initialized to point to the head node of the list (Figure 3-10(a)). There are four nodes in the loop of the list, so P_1 moves four steps ahead and reaches the node with value 5 (Figure 3-10(b)). Then these two pointers move two steps, and they meet at the node with value 3, which is the entry node of the loop, as shown in Figure 3-10(c).

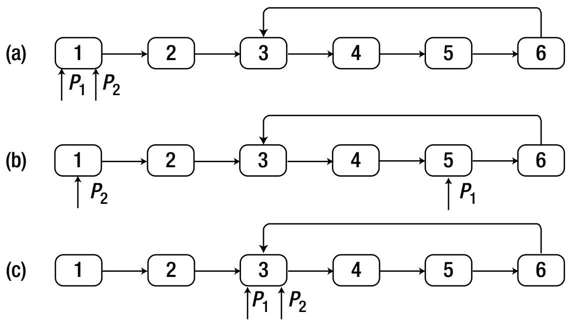

Figure 3-10. *Process to find the entry node of a loop in a list. (a) Pointers* P_1 *and* P_2 *are initialized at the head of list. (b) The pointer* P_1 *moves four steps ahead since there are four nodes in the loop. (c) Both* P_1 *and* P_2 *move ahead till they meet at the entry node of the loop.*

The only problem is how to figure out the number of nodes inside a loop. Let's go back to the solution of the previous question. Two pointers are employed, and the faster one meets the slower one if there is a loop. The meeting node should be inside the loop. Therefore, we can move forward from the meeting node and count nodes in the loop until we arrive at the meeting node again.

The function `MeetingNode` in Listing 3-22 finds the meeting node of two pointers if there is a loop in a list, which is a minor modification of the previous `HasLoop`.

Listing 3-22. *C++ Code to Get a Meeting Node in a Loop*

```
ListNode* MeetingNode(ListNode* pHead) {
    if(pHead == NULL)
        return NULL;

    ListNode* pSlow = pHead->m_pNext;
    if(pSlow == NULL)
        return NULL;

    ListNode* pFast = pSlow->m_pNext;
    while(pFast != NULL && pSlow != NULL) {
        if(pFast == pSlow)
            return pFast;

        pSlow = pSlow->m_pNext;

        pFast = pFast->m_pNext;
        if(pFast != NULL)
            pFast = pFast->m_pNext;
    }

    return NULL;
}
```

The function `MeetingNode` returns a node in the loop when there is a loop in the list. Otherwise, it returns `NULL`.

After finding the meeting node, it counts nodes in a loop of a list, as well as finding the entry node of the loop with the sample code, as shown in Listing 3-23.

Listing 3-23. *C++ Code to Get a Meeting Node in a Loop*

```
ListNode* EntryNodeOfLoop(ListNode* pHead) {
    ListNode* meetingNode = MeetingNode(pHead);
    if(meetingNode == NULL)
        return NULL;

    // get the number of nodes in loop
    int nodesInLoop = 1;
    ListNode* pNode1 = meetingNode;
    while(pNode1->m_pNext != meetingNode) {
        pNode1 = pNode1->m_pNext;
        ++nodesInLoop;
    }

    // move pNode1
    pNode1 = pHead;
    for(int i = 0; i < nodesInLoop; ++i)
        pNode1 = pNode1->m_pNext;

    // move pNode1 and pNode2
    ListNode* pNode2 = pHead;
    while(pNode1 != pNode2){
        pNode1 = pNode1->m_pNext;
        pNode2 = pNode2->m_pNext;
    }

    return pNode1;
}
```

Source Code:

`017_EntryNodeInLoopsInLists.cpp`

Test Cases:

- There is a loop in a list (including cases where there are one/multiple nodes in a loop, or a loop contains all nodes in a list)
- There is not a loop in a list
- The input node of the list head is `NULL`

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.

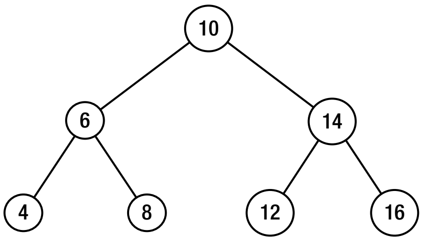

Figure 3-11. *A sample binary tree with seven nodes*

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.

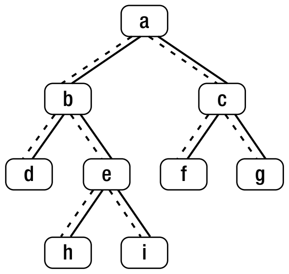

Figure 3-12. A binary tree with nine nodes. Links from children to parents are drawn with dashed arrows.

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.

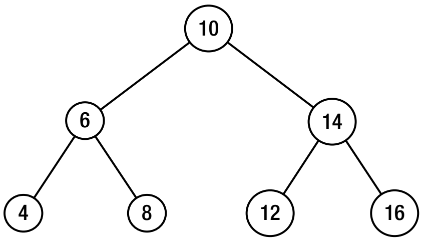

Figure 3-13. *A sample binary search tree with seven nodes*

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.

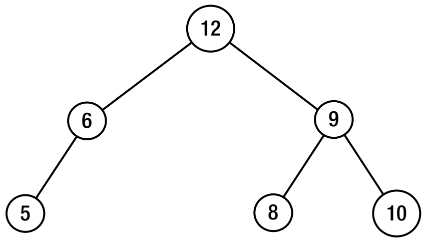

***Figure 3-14.** A sample binary tree with six nodes in which the largest binary search subtree has three nodes (node 9, node 8, and node 10)*

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`

Stack and Queue

A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data, which follows the rule "Last In, First Out." That is to say, the last element pushed into a stack will be the first one to be popped. Because of this rule, stacks are very useful when data have to be stored and then retrieved in reverse order. The stack is a common data structure in computer-related domains. For instance, operating systems create a stack for each thread to store function parameters, return addresses, and local variables.

We will discuss the characteristics of push and pop sequences in the interview question "Push and Pop Sequence of Stacks" (Question 56). Moreover, stacks are closely related to recursion. More details will be covered in the section *Recursion and Iteration*.

A queue is another important data structure where both ends are used: one for inserting new elements and the other for removing them. Different from stacks, queues follow the rule "First in, First Out," where the first enqueued element will be the first one to be dequeued. It is necessary to utilize queues for breadth-first traversal algorithms.

Usually, elements in stacks or queues are not sorted and it costs O(n) time to get the minimal or maximal element. Special design and implementation are needed if it is required to get the minimum or maximum in O(1) time. A detailed discussion is available in the sections *Stack with min Function* and *Maximum in a Queue*.

Even though the rules for stacks and queues are opposite, what is interesting is that a stack can be implemented with two queues, and a queue can be implemented with two stacks.

Build a Queue with Two Stacks

■ **Question 21** Please design a queue with two stacks and implement methods to enqueue and dequeue items.

It is necessary to implement a queue that follows the rule "First In, First Out" with two stacks that follow the rule of "Last In, First Out." The two stacks inside a queue are denoted as *stack1* and *stack2*.

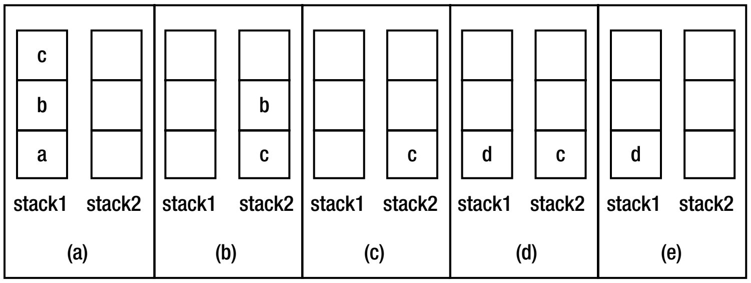

Figure 3-15. The operations on a queue with two stacks. (a) Enqueue the three elements a, b, c *one by one. (b) Dequeue the element* a. *(c) Dequeue the element* b *from the front end. (d) Enqueue another element* d. *(e) Dequeue the element* c.

Let us analyze the process required to enqueue and dequeue elements via some examples. First, an element *a* is appended. Let's push it into *stack1*. There is an element {*a*} in *stack1* and *stack2* that is empty. It continues to append two more elements, *b* and *c*, pushing them into *stack1*. There are three elements {*a, b, c*} in *stack1* now, where *c* is on top and *stack2* is still empty (as shown in Figure 3-15(a)).

We then delete an element from the queue. According to the "First in, First out" rule, the first element to be deleted is *a* since it was appended before *b* and *c*. The element *a* is stored in *stack1*, and it is not on the top of a stack. Therefore, it cannot be popped off directly. Note that *stack2* has not been used, and it is time for us to utilize it. If elements are popped from *stack1* and pushed into *stack2* one by one, the order of elements in *stack2* is in the reverse order of *stack1*. After three popping and pushing operations, *stack1* becomes empty and there are three elements {*c, b, a*} in *stack2*. The element *a* can be popped off now since it is on top of *stack2*. Now there are two elements left {*c, b*} in *stack2* and *b* is on top (as shown in Figure 3-15(b)).

How do we continue deleting more elements from the tail of the queue? The element *b* was inserted into the queue before *c*, so it should be deleted when there are two elements *b* and *c* left in queue. It can be popped off since it is on top of *stack2*. After the popping operation, *stack1* remains empty and there is only an element *c* in *stack2*, as shown in Figure 3-15(c).

It is time to summarize the steps to delete an element from a queue. The top of *stack2* can be popped off when *stack2* is not empty since it is the first element inserted into the queue. When *stack2* is empty, we pop all elements from *stack1* and push them into *stack2* one by one. The first element in a queue is pushed into the bottom of *stack1*. It can be popped out directly after popping and pushing operations since it is on top of *stack2*.

Let us enqueue another element, *d*, and it is pushed into *stack1*, as shown in Figure 3-15(d). If we are going to dequeue an element, the element on top of *stack2*, which is *c*, needs to be deleted. The element *c* is indeed inserted into the queue before the element *d*, so it is a reasonable operation to delete *c* before *d*. The final status of the queue is shown in Figure 3-15(e).

We can write code after we have gotten clear ideas about the process necessary to enqueue and dequeue elements. Some sample code in C# is shown in Listing 3-28.

Listing 3-28. *C# Code to Implement a Queue with Two Stacks*

```
public class QueueWithTwoStacks<T> {
    public void Enqueue(T item) {
        stack1.Push(item);
    }
    public T Dequeue() {
        if (stack2.Count == 0) {
            while (stack1.Count > 0) {
                T item = stack1.Peek();
                stack1.Pop();
                stack2.Push(item);
            }
        }

        if (stack2.Count == 0)
            throw new InvalidOperationException("Queue is Empty");

        T head = stack2.Peek();
        stack2.Pop();

        return head;
    }

    private Stack<T> stack1 = new Stack<T>();
    private Stack<T> stack2 = new Stack<T>();
}
```

Source Code:

`021_QueueWithTwoStacks.cs`

Test Cases:

- Insert elements into an empty queue and then delete them
- Insert elements into a non-empty queue and then delete them
- Enqueue and dequeue multiple elements continuously

Build a Stack with Two Queues

■ **Question 22** Please design a stack with two queues and implement the methods to push and pop items.

Similar to the analysis for the previous question, we employ some examples to simulate a stack with two queues, denoted as *queue1* and *queue2*.

An element *a* is pushed into the stack at first. Since its two queues are empty, we could choose any one of them to hold the first element. Supposing *a* is enqueued into *queue1*. It continues to enqueue two more elements, *b* and *c*, into *queue1*. *queue1* has three elements {*a*, *b*, *c*}, and *a* is at its head, *c* is at its tail, as shown in Figure 3-16(a).

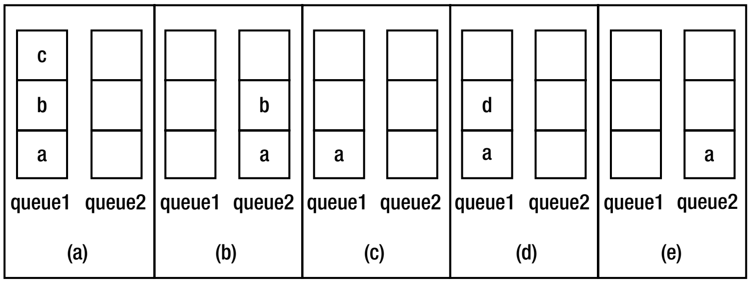

Figure 3-16. *The operations on a stack with two queues. (a) Push the three elements* a, b *and* c *one by one. (b) Pop the element* c. *(c) Pop the element* b *from the top. (d) Push another element* d. *(e) Pop the element* d.

Let us pop an element from the stack. The last element *c* should be popped according to the "Last In, First Out" rule. Only an element at the head of a queue can be deleted, but *c* is at the tail of *queue1*, so it can't be dequeued directly. The two elements *a* and *b* are deleted from queue1 one by one and inserted into *queue2*. Because elements before *c* have been dequeued from *queue1* and *c* is at the head, it can be dequeued now. The status of the stack after *c* is popped is shown in Figure 3-16(b). Similarly, the status of the stack after *b* is popped is shown in Figure 3-16(c).

How about pushing the element *d* into the stack? The item *d* is at the tail of *queue1* after it is enqueued (Figure 3-16(d)). The item *d* is inserted into *queue1* because it is not empty. We should keep one queue empty to accommodate elements of the other queue during popping operations.

If we are going to pop another element, the last pushed element *d* will be popped. Since *d* is not at the head of *queue1*, elements before it have to be dequeued and then enqueued into *queue2*. When *d* is at the head of *queue1*, it is deleted, as shown in Figure 3-16(e).

After we have clear ideas about how to handle pushing and popping operations in a stack with two queues, we can implement such a data structure. The sample code in C# is shown in Listing 3-29.

Listing 3-29. *C# Code to Implement a Stack with Two Queues*

```
public class StackWithTwoQueues<T> {
    public void Push(T item) {
        if (queue2.Count != 0)
            queue2.Enqueue(item);
        else
            queue1.Enqueue(item);
    }

    public T Pop() {
        if (queue1.Count == 0 && queue2.Count == 0)
            throw new InvalidOperationException("Stack is Empty");

        Queue<T> emptyQueue = queue1;
        Queue<T> nonemptyQueue = queue2;
        if (queue1.Count > 0) {
            emptyQueue = queue2;
            nonemptyQueue = queue1;
        }
        while (nonemptyQueue.Count > 1)
            emptyQueue.Enqueue(nonemptyQueue.Dequeue());

        return nonemptyQueue.Dequeue();
    }

    private Queue<T> queue1 = new Queue<T>();
    private Queue<T> queue2 = new Queue<T>();
}
```

Source Code:

```
    022_StackWithTwoQueues.cs
```

Test Cases:

- Insert elements into an empty stack and then delete them
- Insert elements into a non-empty stack and then delete them
- Push and pop multiple elements continuously

Summary

This chapter discusses many coding interview questions about data structures, which are always the focus of technical interviews. It covers common data structures, such as arrays, strings, linked lists, trees, stacks, and queues. It is necessary for candidates to master these structures.

Arrays and strings are two fundamental data structures that store numbers and characters in continuous memory. Many interview questions about arrays are related to search and sort algorithms. Linked lists and trees may be the most frequently discussed data structures during interviews. Candidates should pay much attention to robustness since there are many pointer operations on lists and trees. Stacks are related to recursion, and queues are related to bread-first-search in graphs. Stacks and queues can help us solve many algorithm problems if we have a deep understanding of them.