**1.1** *Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?*

*pg 73*

## SOLUTION

You may want to start off with asking your interviewer if the string is an ASCII string or a Unicode string. This is an important question, and asking it will show an eye for detail and a deep understanding of Computer Science.

We'll assume for simplicity that the character set is ASCII. If not, we would need to increase the storage size, but the rest of the logic would be the same.

Given this, one simple optimization we can make to this problem is to automatically return `false` if the length of the string is greater than the number of unique characters in the alphabet. After all, you can't have a string with 280 unique characters if there are only 256 possible unique characters.

Our first solution is to create an array of boolean values, where the flag at index `i` indicates whether character `i` in the alphabet is contained in the string. If you run across this character a second time, you can immediately return `false`.

The code below implements this algorithm.

```
1  public boolean isUniqueChars2(String str) {
2     if (str.length() > 256) return false;
3
4     boolean[] char_set = new boolean[256];
5     for (int i = 0; i < str.length(); i++) {
6        int val = str.charAt(i);
7        if (char_set[val]) { // Already found this char in string
8           return false;
9        }
10       char_set[val] = true;
11    }
12    return true;
13 }
```

The time complexity for this code is `O(n)`, where n is the length of the string. The space complexity is `O(1)`.

We can reduce our space usage by a factor of eight by using a bit vector. We will assume, in the below code, that the string only uses the lower case letters a through z. This will allow us to use just a single `int`.

```
1  public boolean isUniqueChars(String str) {
2
3
4     int checker = 0;
5     for (int i = 0; i < str.length(); i++) {
6        int val = str.charAt(i) - 'a';
```

```
7         if ((checker & (1 << val)) > 0) {
8            return false;
9         }
10        checker |= (1 << val);
11     }
12     return true;
13  }
```

Alternatively, we could do the following:

1. Compare every character of the string to every other character of the string. This will take $O(n^2)$ time and $O(1)$ space.
2. If we are allowed to modify the input string, we could sort the string in $O(n \log(n))$ time and then linearly check the string for neighboring characters that are identical. Careful, though: many sorting algorithms take up extra space.

These solutions are not as optimal in some respects, but might be better depending on the constraints of the problem.

**1.2** *Implement a function void reverse(char* str) in C or C++ which reverses a null-terminated string.*

*pg 73*

**SOLUTION**

This is a classic interview question. The only "gotcha" is to try to do it in place, and to be careful for the `null` character.

We will implement this in C.

```
1  void reverse(char *str) {
2     char* end = str;
3     char tmp;
4     if (str) {
5        while (*end) { /* find end of the string */
6           ++end;
7        }
8        --end; /* set one char back, since last char is null */
9
10       /* swap characters from start of string with the end of the
11        * string, until the pointers meet in middle. */
12       while (str < end) {
13          tmp = *str;
14          *str++ = *end;
15          *end-- = tmp;
16       }
17    }
18 }
```

This is just one of many ways to implement this solution. We could even implement this

code recursively (but we wouldn't recommend it).

**1.3** *Given two strings, write a method to decide if one is a permutation of the other.*

*pg 73*

## SOLUTION

Like in many questions, we should confirm some details with our interviewer. We should understand if the anagram comparison is case sensitive. That is, is God an anagram of dog? Additionally, we should ask if whitespace is significant.

We will assume for this problem that the comparison is case sensitive and whitespace is significant. So, "god " is different from "dog".

Whenever we compare two strings, we know that if they are different lengths then they cannot be anagrams.

There are two easy ways to solve this problem, both of which use this optimization.

### Solution #1: Sort the strings.

If two strings are anagrams, then we know they have the same characters, but in different orders. Therefore, sorting the strings will put the characters from two anagrams in the same order. We just need to compare the sorted versions of the strings.

```
1  public String sort(String s) {
2     char[] content = s.toCharArray();
3     java.util.Arrays.sort(content);
4     return new String(content);
5  }
6
7  public boolean permutation(String s, String t) {
8     if (s.length() != t.length()) {
9        return false;
10    }
11    return sort(s).equals(sort(t));
12 }
```

Though this algorithm is not as optimal in some senses, it may be preferable in one sense: it's clean, simple and easy to understand. In a practical sense, this may very well be a superior way to implement the problem.

However, if efficiency is very important, we can implement it a different way.

### Solution #2: Check if the two strings have identical character counts.

We can also use the definition of an anagram—two words with the same character counts—to implement this algorithm. We simply iterate through this code, counting how many times each character appears. Then, afterwards, we compare the two arrays.

```
1  public boolean permutation(String s, String t) {
2     if (s.length() != t.length()) {
3        return false;
4     }
5
6     int[] letters = new int[256]; // Assumption
7
8     char[] s_array = s.toCharArray();
9     for (char c : s_array) { // count number of each char in s.
10       letters[c]++;
11    }
12
13    for (int i = 0; i < t.length(); i++) {
14       int c = (int) t.charAt(i);
15       if (--letters[c] < 0) {
16          return false;
17       }
18    }
19
20    return true;
21 }
```

Note the assumption on line 6. In your interview, you should always check with your interviewer about the size of the character set. We assumed that the character set was ASCII.

**1.4** *Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end of the string to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)*

*pg 73*

**SOLUTION**

A common approach in string manipulation problems is to edit the string starting from the end and work backwards. This is useful because we have extra buffer at the end, which allows us to change characters without worrying about what we're overwriting.

We will use this approach in this problem. The algorithm works through a two scan approach. In the first scan, we count how many spaces there are in the string. This is used to compute how long the final string should be. In the second pass, which is done in reverse order, we actually edit the string. When we see a space, we copy `%20` into the next spots. If there is no space, then we copy the original character.

The code below implements this algorithm.

```
1  public void replaceSpaces(char[] str, int length) {
2     int spaceCount = 0, newLength, i;
3     for (i = 0; i < length; i++) {
```

```
4        if (str[i] == ' ') {
5           spaceCount++;
6        }
7     }
8     newLength = length + spaceCount * 2;
9     str[newLength] = '\0';
10    for (i = length - 1; i >= 0; i--) {
11       if (str[i] == ' ') {
12          str[newLength - 1] = '0';
13          str[newLength - 2] = '2';
14          str[newLength - 3] = '%';
15          newLength = newLength - 3;
16       } else {
17          str[newLength - 1] = str[i];
18          newLength = newLength - 1;
19       }
20    }
21 }
```

We have implemented this problem using character arrays, since Java strings are immutable. If we used strings directly, this would require returning a new copy of the string, but it would allow us to implement this in just one pass.

**1.5** *Implement a method to perform basic string compression using the counts of repeated characters. For example, the string* `aabcccccaaa` *would become* `a2b1c5a3`. *If the "compressed" string would not become smaller than the original string, your method should return the original string.*

*pg 73*

**SOLUTION**

At first glance, implementing this method seems fairly straightforward, but perhaps a bit tedious. We iterate through the string, copying characters to a new string and counting the repeats. How hard could it be?

```
1  public String compressBad(String str) {
2     String mystr = "";
3     char last = str.charAt(0);
4     int count = 1;
5     for (int i = 1; i < str.length(); i++) {
6        if (str.charAt(i) == last) { // Found repeat char
7           count++;
8        } else { // Insert char count, and update last char
9           mystr += last + "" + count;
10          last = str.charAt(i);
11          count = 1;
12       }
13    }
14    return mystr + last + count;
```

```
15 }
```

This code doesn't handle the case when the compressed string is longer than the original string, but it otherwise works. Is it efficient though? Take a look at the runtime of this code.

The runtime is $O(p + k^2)$, where p is the size of the original string and k is the number of character sequences. For example, if the string is aabccdeeaa, then there are six character sequences. It's slow because string concatenation operates in $O(n^2)$ time (see **StringBuffer** in Chapter 1).

We can make this somewhat better by using a StringBuffer.

```
1  String compressBetter(String str) {
2     /* Check if compression would create a longer string */
3     int size = countCompression(str);
4     if (size >= str.length()) {
5        return str;
6     }
7
8     StringBuffer mystr = new StringBuffer();
9     char last = str.charAt(0);
10    int count = 1;
11    for (int i = 1; i < str.length(); i++) {
12       if (str.charAt(i) == last) { // Found repeated char
13          count++;
14       } else { // Insert char count, and update last char
15          mystr.append(last); // Insert char
16          mystr.append(count); // Insert count
17          last = str.charAt(i);
18          count = 1;
19       }
20    }
21
22    /* In lines 15 - 16 above, characters are inserted when the
23     * repeated character changes. We need to update the string at
24     * the end of the method as well, since the very last set of
25     * repeated characters wouldn't be set in the compressed string
26     * yet. */
27    mystr.append(last);
28    mystr.append(count);
29    return mystr.toString();
30 }
31
32 int countCompression(String str) {
33    if (str == null || str.isEmpty()) return 0;
34    char last = str.charAt(0);
35    int size = 0;
36    int count = 1;
37    for (int i = 1; i < str.length(); i++) {
38       if (str.charAt(i) == last) {
```

```
39          count++;
40        } else {
41          last = str.charAt(i);
42          size += 1 + String.valueOf(count).length();
43          count = 1;
44        }
45      }
46      size += 1 + String.valueOf(count).length();
47      return size;
48  }
```

This algorithm is much better. Note that we have added the size check in lines 2 through 5.

If we don't want to (or aren't allowed to) use a `StringBuffer`, we can still solve this problem efficiently. In line 2, we compute the end size of the string. This allows us to create a `char` array of the correct size, so we can implement the code as follows:

```
1   String compressAlternate(String str) {
2     /* Check if compression would create a longer string */
3     int size = countCompression(str);
4     if (size >= str.length()) {
5       return str;
6     }
7
8     char[] array = new char[size];
9     int index = 0;
10    char last = str.charAt(0);
11    int count = 1;
12    for (int i = 1; i < str.length(); i++) {
13      if (str.charAt(i) == last) { // Found repeated character
14        count++;
15      } else {
16        /* Update the repeated character count */
17        index = setChar(array, last, index, count);
18        last = str.charAt(i);
19        count = 1;
20      }
21    }
22
23    /* Update string with the last set of repeated characters. */
24    index = setChar(array, last, index, count);
25    return String.valueOf(array);
26  }
27
28  int setChar(char[] array, char c, int index, int count) {
29    array[index] = c;
30    index++;
31
32    /* Convert the count to a string, then to an array of chars */
33    char[] cnt = String.valueOf(count).toCharArray();
```

```
34
35    /* Copy characters from biggest digit to smallest */
36    for (char x : cnt) {
37       array[index] = x;
38       index++;
39    }
40    return index;
41 }
42
43 int countCompression(String str) { /* same as earlier */ }
```

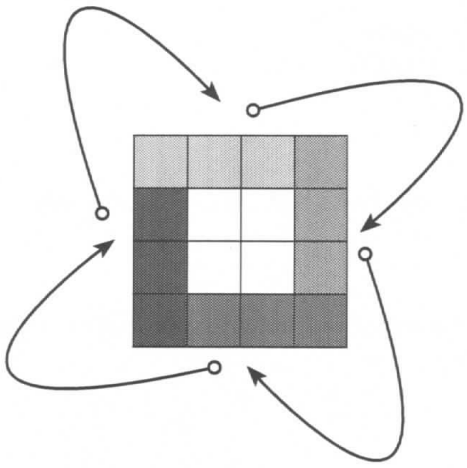
Like the second solution, the above code runs in O(N) time and O(N) space.

**1.6** *Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?*

*pg 73*

**SOLUTION**

Because we're rotating the matrix by 90 degrees, the easiest way to do this is to implement the rotation in layers. We perform a circular rotation on each layer, moving the top edge to the right edge, the right edge to the bottom edge, the bottom edge to the left edge, and the left edge to the top edge.



How do we perform this four-way edge swap? One option is to copy the top edge to an array, and then move the left to the top, the bottom to the left, and so on. This requires O(N) memory, which is actually unnecessary.

A better way to do this is to implement the swap index by index. In this case, we do the following:

```
1  for i = 0 to n
2     temp = top[i];
3     top[i] = left[i]
```

```
4     left[i] = bottom[i]
5     bottom[i] = right[i]
6     right[i] = temp
```

We perform such a swap on each layer, starting from the outermost layer and working our way inwards. (Alternatively, we could start from the inner layer and work outwards.)

The code for this algorithm is below.

```
1  public void rotate(int[][] matrix, int n) {
2     for (int layer = 0; layer < n / 2; ++layer) {
3        int first = layer;
4        int last = n - 1 - layer;
5        for(int i = first; i < last; ++i) {
6           int offset = i - first;
7           // save top
8           int top = matrix[first][i];
9
10          // left -> top
11          matrix[first][i] = matrix[last-offset][first];
12
13          // bottom -> left
14          matrix[last-offset][first] = matrix[last][last - offset];
15
16          // right -> bottom
17          matrix[last][last - offset] = matrix[i][last];
18
19          // top -> right
20          matrix[i][last] = top;
21       }
22    }
23 }
```

This algorithm is $O(N^2)$, which is the best we can do since any algorithm must touch all $N^2$ elements.

**1.7** *Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column are set to 0.*

*pg 73*

**SOLUTION**

At first glance, this problem seems easy: just iterate through the matrix and every time we see a cell with value zero, set its row and column to 0. There's one problem with that solution though: when we come across other cells in that row or column, we'll see the zeros and change their row and column to zero. Pretty soon, our entire matrix will be set to zeros.

One way around this is to keep a second matrix which flags the zero locations. We would then do a second pass through the matrix to set the zeros. This would take $O(MN)$ space.

Do we really need O(MN) space? No. Since we're going to set the entire row and column to zero, we don't need to track that it was exactly `cell[2][4]` (row 2, column 4). We only need to know that row 2 has a zero somewhere, and column 4 has a zero somewhere. We'll set the entire row and column to zero anyway, so why would we care to keep track of the exact location of the zero?

The code below implements this algorithm. We use two arrays to keep track of all the rows with zeros and all the columns with zeros. We then make a second pass of the matrix and set a cell to zero if its row or column is zero.

```
1  public void setZeros(int[][] matrix) {
2     boolean[] row = new boolean[matrix.length];
3     boolean[] column = new boolean[matrix[0].length];
4
5     // Store the row and column index with value 0
6     for (int i = 0; i < matrix.length; i++) {
7        for (int j = 0; j < matrix[0].length;j++) {
8           if (matrix[i][j] == 0) {
9              row[i] = true;
10             column[j] = true;
11          }
12       }
13    }
14
15    // Set arr[i][j] to 0 if either row i or column j has a 0
16    for (int i = 0; i < matrix.length; i++) {
17       for (int j = 0; j < matrix[0].length; j++) {
18          if (row[i] || column[j]) {
19             matrix[i][j] = 0;
20          }
21       }
22    }
23 }
```

To make this somewhat more space efficient, we could use a bit vector instead of a boolean array.

**1.8** *Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").*

*pg 74*

**SOLUTION**

If we imagine that `s2` is a rotation of `s1`, then we can ask what the rotation point is. For example, if you rotate `waterbottle` after `wat`, you get `erbottlewat`. In a rotation, we cut `s1` into two parts, `x` and `y`, and rearrange them to get `s2`.

```
s1 = xy = waterbottle
x = wat
y = erbottle
s2 = yx = erbottlewat
```

So, we need to check if there's a way to split s1 into x and y such that xy = s1 and yx = s2. Regardless of where the division between x and y is, we can see that yx will always be a substring of xyxy. That is, s2 will always be a substring of s1s1.

And this is precisely how we solve the problem: simply do `isSubstring(s1s1, s2)`.

The code below implements this algorithm.

```
1  public boolean isRotation(String s1, String s2) {
2    int len = s1.length();
3    /* check that s1 and s2 are equal length and not empty */
4    if (len == s2.length() && len > 0) {
5      /* concatenate s1 and s1 within new buffer */
6      String s1s1 = s1 + s1;
7      return isSubstring(s1s1, s2);
8    }
9    return false;
10 }
```

# Linked Lists

*Data Structures: Solutions* **Chapter 2**

**2.1** *Write code to remove duplicates from an unsorted linked list.*

*FOLLOW UP*

*How would you solve this problem if a temporary buffer is not allowed?*

*pg 77*

## SOLUTION

In order to remove duplicates from a linked list, we need to be able to track duplicates. A simple hash table will work well here.

In the below solution, we simply iterate through the linked list, adding each element to a hash table. When we discover a duplicate element, we remove the element and continue iterating. We can do this all in one pass since we are using a linked list.

```
1  public static void deleteDups(LinkedListNode n) {
2     Hashtable table = new Hashtable();
3     LinkedListNode previous = null;
4     while (n != null) {
5        if (table.containsKey(n.data)) {
6           previous.next = n.next;
7        } else {
8           table.put(n.data, true);
9           previous = n;
10       }
11       n = n.next;
12    }
13 }
```

The above solution takes `O(N)` time, where `N` is the number of elements in the linked list.

### Follow Up: No Buffer Allowed

If we don't have a buffer, we can iterate with two pointers: `current` which iterates through the linked list, and `runner` which checks all subsequent nodes for duplicates.

```
1  public static void deleteDups(LinkedListNode head) {
2     if (head == null) return;
3
4     LinkedListNode current = head;
5     while (current != null) {
6        /* Remove all future nodes that have the same value */
7        LinkedListNode runner = current;
8        while (runner.next != null) {
9           if (runner.next.data == current.data) {
10             runner.next = runner.next.next;
11          } else {
12             runner = runner.next;
13          }
14       }
```

```
15        current = current.next;
16    }
17 }
```

This code runs in O(1) space, but $O(N^2)$ time.

**2.2** *Implement an algorithm to find the kth to last element of a singly linked list.*

*pg 77*

## SOLUTION

We will approach this problem both recursively and non-recursively. Remember that recursive solutions are often cleaner but less optimal. For example, in this problem, the recursive implementation is about half the length of the iterative solution but also takes O(n) space, where n is the number of elements in the linked list.

Note that for this solution, we have defined k such that passing in k = 1 would return the last element, k = 2 would return to the second to last element, and so on. It is equally acceptable to define k such that k = 0 would return the last element.

### Solution #1: If linked list size is known

If the size of the linked list is known, then the kth to last element is the (length - k) th element. We can just iterate through the linked list to find this element. Because this solution is so trivial, we can almost be sure that this is not what the interviewer intended.

### Solution #2: Recursive

This algorithm recurses through the linked list. When it hits the end, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k, we know we have reached the kth to last element of the linked list.

Implementing this is short and sweet—provided we have a way of "passing back" an integer value through the stack. Unfortunately, we can't pass back a node and a counter using normal return statements. So how do we handle this?

*Approach A: Don't Return the Element.*

One way to do this is to change the problem to simply printing the kth to last element. Then, we can pass back the value of the counter simply through return values.

```
1  public static int nthToLast(LinkedListNode head, int k) {
2    if (head == null) {
3      return 0;
4    }
5    int i = nthToLast(head.next, k) + 1;
6    if (i == k) {
7      System.out.println(head.data);
8    }
```

```
9     return i;
10 }
```

Of course, this is only a valid solution if the interviewer says it is valid.

*Approach B: Use C++.*

A second way to solve this is to use C++ and to pass values by reference. This allows us to return the node value, but also update the counter by passing a pointer to it.

```
1  node* nthToLast(node* head, int k, int& i) {
2     if (head == NULL) {
3        return NULL;
4     }
5     node * nd = nthToLast(head->next, k, i);
6     i = i + 1;
7     if (i == k) {
8        return head;
9     }
10    return nd;
11 }
```

*Approach C: Create a Wrapper Class.*

We described earlier that the issue was that we couldn't simultaneously return a counter and an index. If we wrap the `counter` value with simple class (or even a single element array), we can mimic passing by reference.

```
1  public class IntWrapper {
2     public int value = 0;
3  }
4
5  LinkedListNode nthToLastR2(LinkedListNode head, int k,
6                             IntWrapper i) {
7     if (head == null) {
8        return null;
9     }
10    LinkedListNode node = nthToLastR2(head.next, k, i);
11    i.value = i.value + 1;
12    if (i.value == k) { // We've found the kth element
13       return head;
14    }
15    return node;
16 }
17
```

Each of these recursive solutions takes `O(n)` space due to the recursive calls.

There are a number of other solutions that we haven't addressed. We could store the counter in a static variable. Or, we could create a class that stores both the node and the counter, and return an instance of that class. Regardless of which solution we pick, we need a way to update both the node and the counter in a way that all levels of the recursive stack will see.

**Solution #3: Iterative**

A more optimal, but less straightforward, solution is to implement this iteratively. We can use two pointers, p1 and p2. We place them k nodes apart in the linked list by putting p1 at the beginning and moving p2 k nodes into the list. Then, when we move them at the same pace, p2 will hit the end of the linked list after LENGTH - k steps. At that point, p1 will be LENGTH - k nodes into the list, or k nodes from the end.

The code below implements this algorithm.

```
1  LinkedListNode nthToLast(LinkedListNode head, int k) {
2    if (k <= 0) return null;
3
4    LinkedListNode p1 = head;
5    LinkedListNode p2 = head;
6
7    // Move p2 forward k nodes into the list.
8    for (int i = 0; i < k - 1; i++) {
9      if (p2 == null) return null; // Error check
10     p2 = p2.next;
11   }
12   if (p2 == null) return null;
13
14   /* Now, move p1 and p2 at the same speed. When p2 hits the end,
15    * p1 will be at the right element. */
16   while (p2.next != null) {
17     p1 = p1.next;
18     p2 = p2.next;
19   }
20   return p1;
21 }
```

This algorithm takes O(n) time and O(1) space.

**2.3** *Implement an algorithm to delete a node in the middle of a singly linked list, given only access to that node.*

*pg 77*

**SOLUTION**

In this problem, you are not given access to the head of the linked list. You only have access to that node. The solution is simply to copy the data from the next node over to the current node, and then to delete the next node.

The code below implements this algorithm.

```
1  public static boolean deleteNode(LinkedListNode n) {
2    if (n == null || n.next == null) {
3      return false; // Failure
4    }
```

```
    LinkedListNode next = n.next;
    n.data = next.data;
    n.next = next.next;
    return true;
}
```

Note that this problem cannot be solved if the node to be deleted is the last node in the linked list. That's ok—your interviewer wants you to point that out, and to discuss how to handle this case. You could, for example, consider marking the node as dummy.

**2.4** *Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.*

*pg 77*

**SOLUTION**

If this were an array, we would need to be careful about how we shifted elements. Array shifts are very expensive.

However, in a linked list, the situation is much easier. Rather than shifting and swapping elements, we can actually create two different linked lists: one for elements less than x, and one for elements greater than or equal to x.

We iterate through the linked list, inserting elements into our `before` list or our `after` list. Once we reach the end of the linked list and have completed this splitting, we merge the two lists.

The code below implements this approach.

```
/* Pass in the head of the linked list and the value to partition
 * around */
public LinkedListNode partition(LinkedListNode node, int x) {
   LinkedListNode beforeStart = null;
   LinkedListNode beforeEnd = null;
   LinkedListNode afterStart = null;
   LinkedListNode afterEnd = null;

   /* Partition list */
   while (node != null) {
      LinkedListNode next = node.next;
      node.next = null;
      if (node.data < x) {
         /* Insert node into end of before list */
         if (beforeStart == null) {
            beforeStart = node;
            beforeEnd = beforeStart;
         } else {
            beforeEnd.next = node;
            beforeEnd = node;
```

```
21          }
22        } else {
23          /* Insert node into end of after list */
24          if (afterStart == null) {
25            afterStart = node;
26            afterEnd = afterStart;
27          } else {
28            afterEnd.next = node;
29            afterEnd = node;
30          }
31        }
32        node = next;
33      }
34
35      if (beforeStart == null) {
36        return afterStart;
37      }
38
39      /* Merge before list and after list */
40      beforeEnd.next = afterStart;
41      return beforeStart;
42  }
```

If it bugs you to keep around four different variables for tracking two linked lists, you're not alone. We can get rid of some of these, with just a minor hit to the efficiency. This drop in efficiency comes because we have to traverse the linked list an extra time. The big-O time will remain the same though, and we get shorter, cleaner code.

The second solution operates in a slightly different way. Instead of inserting nodes into the end of the before list and the after list, it inserts nodes into the front of them.

```
1   public LinkedListNode partition(LinkedListNode node, int x) {
2     LinkedListNode beforeStart = null;
3     LinkedListNode afterStart = null;
4
5     /* Partition list */
6     while (node != null) {
7       LinkedListNode next = node.next;
8       if (node.data < x) {
9         /* Insert node into start of before list */
10        node.next = beforeStart;
11        beforeStart = node;
12      } else {
13        /* Insert node into front of after list */
14        node.next = afterStart;
15        afterStart = node;
16      }
17      node = next;
18    }
19
20    /* Merge before list and after list */
```

```
21    if (beforeStart == null) {
22       return afterStart;
23    }
24
25    /* Find end of before list, and merge the lists */
26    LinkedListNode head = beforeStart;
27    while (beforeStart.next != null) {
28       beforeStart = beforeStart.next;
29    }
30    beforeStart.next = afterStart;
31
32    return head;
33 }
```

Note that in this problem, we need to be very careful about null values. Check out line 7 in the above solution. The line is here because we are modifying the linked list as we're looping through it. We need to store the next node in a temporary variable so that we remember which node should be next in our iteration.

**2.5** *You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.*

*FOLLOW UP*

*Suppose the digits are stored in forward order. Repeat the above problem.*

*pg 77*

**SOLUTION**

It's useful to remember in this problem how exactly addition works. Imagine the problem:

```
  6 1 7
+ 2 9 5
```

First, we add 7 and 5 to get 12. The digit 2 becomes the last digit of the number, and 1 gets carried over to the next step. Second, we add 1, 1, and 9 to get 11. The 1 becomes the second digit, and the other 1 gets carried over the final step. Third and finally, we add 1, 6 and 2 to get 9. So, our value becomes 912.

We can mimic this process recursively by adding node by node, carrying over any "excess" data to the next node. Let's walk through this for the below linked list:

```
  7 -> 1 -> 6
+ 5 -> 9 -> 2
```

We do the following:

1. We add 7 and 5 first, getting a result of 12. 2 becomes the first node in our linked list,

and we "carry" the 1 to the next sum.

```
List: 2 -> ?
```

2. We then add 1 and 9, as well as the "carry," getting a result of 11. 1 becomes the second element of our linked list, and we carry the 1 to the next sum.

```
List: 2 -> 1 -> ?
```

3. Finally, we add 6, 2 and our "carry," to get 9. This become the final element of our linked list.

```
List: 2 -> 1 -> 9.
```

The code below implements this algorithm.

```
1  LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2,
2                          int carry) {
3     /* We're done if both lists are null AND the carry value is 0 */
4     if (l1 == null && l2 == null && carry == 0) {
5        return null;
6     }
7
8     LinkedListNode result = new LinkedListNode(carry, null, null);
9
10    /* Add value, and the data from l1 and l2 */
11    int value = carry;
12    if (l1 != null) {
13       value += l1.data;
14    }
15    if (l2 != null) {
16       value += l2.data;
17    }
18
19    result.data = value % 10; /* Second digit of number */
20
21    /* Recurse */
22    if (l1 != null || l2 != null) {
23       LinkedListNode more = addLists(l1 == null ? null : l1.next,
24                                      l2 == null ? null : l2.next,
25                                      value >= 10 ? 1 : 0);
26       result.setNext(more);
27    }
28    return result;
29 }
```

In implementing this code, we must be careful to handle the condition when one linked list is shorter than another. We don't want to get a null pointer exception.

### Follow Up

Part B is conceptually the same (recurse, carry the excess), but has some additional complications when it comes to implementation:

1. One list may be shorter than the other, and we cannot handle this "on the fly." For example, suppose we were adding (1 -> 2 -> 3 -> 4) and (5 -> 6 -> 7). We need to know that the 5 should be "matched" with the 2, not the 1. We can accomplish this by comparing the lengths of the lists in the beginning and padding the shorter list with zeros.
2. In the first part, successive results were added to the tail (i.e., passed forward). This meant that the recursive call would be *passed* the carry, and would return the result (which is then appended to the tail). In this case, however, results are added to the head (i.e., passed backward). The recursive call must return the result, as before, as well as the carry. This is not terribly challenging to implement, but it is more cumbersome. We can solve this issue by creating a wrapper class called Partial Sum.

The code below implements this algorithm.

```
1  public class PartialSum {
2     public LinkedListNode sum = null;
3     public int carry = 0;
4  }
5
6  LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
7     int len1 = length(l1);
8     int len2 = length(l2);
9
10    /* Pad the shorter list with zeros - see note (1) */
11    if (len1 < len2) {
12       l1 = padList(l1, len2 - len1);
13    } else {
14       l2 = padList(l2, len1 - len2);
15    }
16
17    /* Add lists */
18    PartialSum sum = addListsHelper(l1, l2);
19
20    /* If there was a carry value left over, insert this at the
21     * front of the list. Otherwise, just return the linked list. */
22    if (sum.carry == 0) {
23       return sum.sum;
24    } else {
25       LinkedListNode result = insertBefore(sum.sum, sum.carry);
26       return result;
27    }
28 }
29
30 PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
31    if (l1 == null && l2 == null) {
32       PartialSum sum = new PartialSum();
33       return sum;
34    }
35    /* Add smaller digits recursively */
```

```
36    PartialSum sum = addListsHelper(l1.next, l2.next);
37
38    /* Add carry to current data */
39    int val = sum.carry + l1.data + l2.data;
40
41    /* Insert sum of current digits */
42    LinkedListNode full_result = insertBefore(sum.sum, val % 10);
43
44    /* Return sum so far, and the carry value */
45    sum.sum = full_result;
46    sum.carry = val / 10;
47    return sum;
48 }
49
50 /* Pad the list with zeros */
51 LinkedListNode padList(LinkedListNode l, int padding) {
52    LinkedListNode head = l;
53    for (int i = 0; i < padding; i++) {
54       LinkedListNode n = new LinkedListNode(0, null, null);
55       head.prev = n;
56       n.next = head;
57       head = n;
58    }
59    return head;
60 }
61
62 /* Helper function to insert node in the front of a linked list */
63 LinkedListNode insertBefore(LinkedListNode list, int data) {
64    LinkedListNode node = new LinkedListNode(data, null, null);
65    if (list != null) {
66       list.prev = node;
67       node.next = list;
68    }
69    return node;
70 }
```

Note how we have pulled `insertBefore()`, `padList()`, and `length()` (not listed) into their own methods. This makes the code cleaner and easier to read—a wise thing to do in your interviews!

**2.6** *Given a circular linked list, implement an algorithm which returns the node at the beginning of the loop.*

*pg 78*

**SOLUTION**

This is a modification of a classic interview problem: detect if a linked list has a loop. Let's apply the Pattern Matching approach.

### Part 1: Detect If Linked List Has A Loop

An easy way to detect if a linked list has a loop is through the `FastRunner` / `SlowRunner` approach. `FastRunner` moves two steps at a time, while `SlowRunner` moves one step. Much like two cars racing around a track at different steps, they must eventually meet.

An astute reader may wonder if `FastRunner` might "hop over" `SlowRunner` completely, without ever colliding. That's not possible. Suppose that `FastRunner` *did* hop over `SlowRunner`, such that SlowRunner is at spot `i` and `FastRunner` is at spot `i + 1`. In the previous step, SlowRunner would be at spot `i - 1` and FastRunner would at spot `((i + 1) - 2)`, or spot `i - 1`. That is, they would have collided.

### Part 2: When Do They Collide?

Let's assume that the linked list has a "non-looped" part of size k.

If we apply our algorithm from part 1, when will `FastRunner` and `SlowRunner` collide?

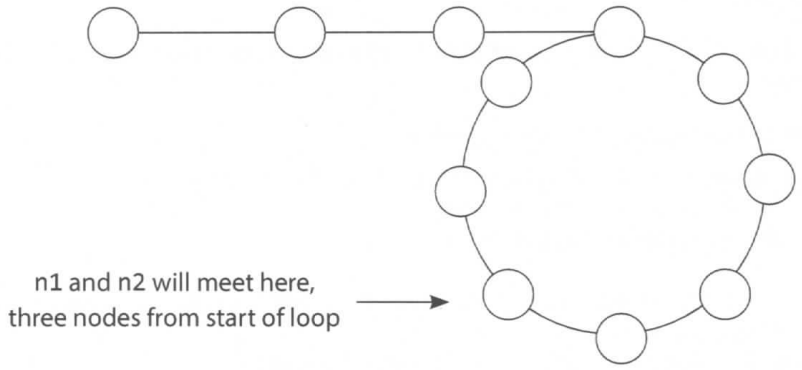
We know that for every p steps that `SlowRunner` takes, `FastRunner` has taken 2p steps. Therefore, when `SlowRunner` enters the looped portion after k steps, `FastRunner` has taken 2k steps total and must be 2k - k steps, or k steps, into the looped portion. Since k might be much larger than the loop length, we should actually write this as `mod(k, LOOP_SIZE)` steps, which we will denote as K.

At each subsequent step, `FastRunner` and `SlowRunner` get either one step farther away or one step closer, depending on your perspective. That is, because we are in a circle, when A moves q steps away from B, it is also moving q steps closer to B.

So now we know the following facts:

1. `SlowRunner` is 0 steps into the loop.
2. `FastRunner` is K steps into the loop.
3. `SlowRunner` is K steps behind `FastRunner`.
4. `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`.
5. `FastRunner` catches up to `SlowRunner` at a rate of 1 step per unit of time.

So, when do they meet? Well, if `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`, and `FastRunner` catches up at a rate of 1 step per unit of time, then they meet after `LOOP_SIZE - K` steps. At this point, they will be K steps before the head of the loop. Let's call this point `CollisionSpot`.




**Part 3: How Do You Find The Start of the Loop?**

We now know that `CollisionSpot` is `K` nodes before the start of the loop. Because `K = mod(k, LOOP_SIZE)` (or, in other words, `k = K + M * LOOP_SIZE`, for any integer `M`), it is also correct to say that it is `k` nodes from the loop start. For example, if node `N` is 2 nodes into a 5 node loop, it is also correct to say that it is 7, 12, or even 397 nodes into the loop.

Therefore, both `CollisionSpot` and `LinkedListHead` are `k` nodes from the start of the loop.

Now, if we keep one pointer at `CollisionSpot` and move the other one to `LinkedListHead`, they will each be `k` nodes from `LoopStart`. Moving the two pointers at the same speed will cause them to collide again—this time after `k` steps, at which point they will both be at `LoopStart`. All we have to do is return this node.

**Part 4: Putting It All Together**

To summarize, we move `FastPointer` twice as fast as `SlowPointer`. When `SlowPointer` enters the loop, after `k` nodes, `FastPointer` is `k` nodes into the linked list. This means that `FastPointer` and `SlowPointer` are `LOOP_SIZE - k` nodes away from each other.

Next, if `FastPointer` moves two nodes for each node that `SlowPointer` moves, they move one node closer to each other on each turn. Therefore, they will meet after `LOOP_SIZE - k` turns. Both will be `k` nodes from the front of the loop.

The head of the linked list is also `k` nodes from the front of the loop. So, if we keep one pointer where it is, and move the other pointer to the head of the linked list, then they will meet at the front of the loop.

Our algorithm is derived directly from parts 1, 2 and 3.

1. Create two pointers, `FastPointer` and `SlowPointer`.
2. Move `FastPointer` at a rate of 2 steps and `SlowPointer` at a rate of 1 step.
3. When they collide, move `SlowPointer` to `LinkedListHead`. Keep `FastPointer`

where it is.

4. Move `SlowPointer` and `FastPointer` at a rate of one step. Return the new collision point.

The code below implements this algorithm.

```
1  LinkedListNode FindBeginning(LinkedListNode head) {
2    LinkedListNode slow = head;
3    LinkedListNode fast = head;
4
5    /* Find meeting point. This will be LOOP_SIZE - k steps into the
6     * linked list. */
7    while (fast != null && fast.next != null) {
8      slow = slow.next;
9      fast = fast.next.next;
10     if (slow == fast) { // Collision
11       break;
12     }
13   }
14
15   /* Error check - no meeting point, and therefore no loop */
16   if (fast == null || fast.next == null) {
17     return null;
18   }
19
20   /* Move slow to Head. Keep fast at Meeting Point. Each are k
21    * steps from the Loop Start. If they move at the same pace,
22    * they must meet at Loop Start. */
23   slow = head;
24   while (slow != fast) {
25     slow = slow.next;
26     fast = fast.next;
27   }
28
29   /* Both now point to the start of the loop. */
30   return fast;
31 }
```

**2.7** *Implement a function to check if a linked list is a palindrome.*

*pg 78*

## SOLUTION

To approach this problem, we can picture a palindrome like `0 -> 1 -> 2 -> 1 -> 0`. We know that, since it's a palindrome, the list must be the same backwards and forwards. This leads us to our first solution.

### Solution #1: Reverse and Compare

Our first solution is to reverse the linked list and compare the reversed list to the original list. If they're the same, the lists are identical.

Note that when we compare the linked list to the reversed list, we only actually need to compare the first half of the list. If the first half of the normal list matches the first half of the reversed list, then the second half of the normal list must match the second half of the reversed list.

**Solution #2: Iterative Approach**

We want to detect linked lists where the front half of the list is the reverse of the second half. How would we do that? By reversing the front half of the list. A stack can accomplish this.

We need to push the first half of the elements onto a stack. We can do this in two different ways, depending on whether or not we know the size of the linked list.

If we know the size of the linked list, we can iterate through the first half of the elements in a standard for loop, pushing each element onto a stack. We must be careful, of course, to handle the case where the length of the linked list is odd.

If we don't know the size of the linked list, we can iterate through the linked list, using the fast runner / slow runner technique described in the beginning of the chapter. At each step in the loop, we push the data from the slow runner onto a stack. When the fast runner hits the end of the list, the slow runner will have reached the middle of the linked list. By this point, the stack will have all the elements from the front of the linked list, but in reverse order.

Now, we simply iterate through the rest of the linked list. At each iteration, we compare the node to the top of the stack. If we complete the iteration without finding a difference, then the linked list is a palindrome.

```
1  boolean isPalindrome(LinkedListNode head) {
2    LinkedListNode fast = head;
3    LinkedListNode slow = head;
4
5    Stack<Integer> stack = new Stack<Integer>();
6
7    /* Push elements from first half of linked list onto stack. When
8     * fast runner (which is moving at 2x speed) reaches the end of
9     * the linked list, then we know we're at the middle */
10   while (fast != null && fast.next != null) {
11     stack.push(slow.data);
12     slow = slow.next;
13     fast = fast.next.next;
14   }
15
16   /* Has odd number of elements, so skip the middle element */
17   if (fast != null) {
```

```
18        slow = slow.next;
19     }
20
21     while (slow != null) {
22        int top = stack.pop().intValue();
23
24        /* If values are different, then it's not a palindrome */
25        if (top != slow.data) {
26           return false;
27        }
28        slow = slow.next;
29     }
30     return true;
31  }
```

**Solution #3: Recursive Approach**

First, a word on notation: in the below solution, when we use the notation node Kx, the variable K indicates the value of the node data, and x (which is either f or b) indicates whether we are referring to the front node with that value or the back node. For example, in the below linked list, node 3b would refer to the second (back) node with value 3.

Now, like many linked list problems, you can approach this problem recursively. We may have some intuitive idea that we want to compare element 0 and element n, element 1 and element n-1, element 2 and element n-2, and so on, until the middle element(s). For example:

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

In order to apply this approach, we first need to know when we've reached the middle element, as this will form our base case. We can do this by passing in `length - 2` for the length each time. When the length equals 0 or 1, we're at the center of the linked list.

```
1  recurse(Node n, int length) {
2     if (length == 0 || length == 1) {
3        return [something]; // At middle
4     }
5     recurse(n.next, length - 2);
6     ...
7  }
```

This method will form the outline of the `isPalindrome` method. The "meat" of the algorithm though is comparing node `i` to node `n - i` to check if the linked list is a palindrome. How do we do that?

Let's examine what the call stack looks like:

```
1  v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
2     v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
3        v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
```

```
4          v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5          returns v3
6        returns v2
7      returns v1
8   returns ?
```

In the above call stack, each call wants to check if the list is a palindrome by comparing its head node with the corresponding node from the back of the list. That is:

- Line 1 needs to compare node `0f` with node `0b`
- Line 2 needs to compare node `1f` with node `1b`
- Line 3 needs to compare node `2f` with node `2b`
- Line 4 needs to compare node `3f` with node `3b`.

If we rewind the stack, passing nodes back as described below, we can do just that:

- Line 4 sees that it is the middle node (since `length = 1`), and passes back `head.next`. The value head equals node `3`, so `head.next` is node `2b`.
- Line 3 compares its head, node `2f`, to `returned_node` (the value from the previous recursive call), which is node `2b`. If the values match, it passes a reference to node `1b` (`returned_node.next`) up to line 2.
- Line 2 compares its head (node `1f`) to `returned_node` (node `1b`). If the values match, it passes a reference to node `0b` (or, `returned_node.next`) up to line 1.
- Line 1 compares its head, node `0f`, to `returned_node`, which is node `0b`. If the values match, it returns true.

To generalize, each call compares its head to `returned_node`, and then passes `returned_node.next` up the stack. In this way, every node `i` gets compared to node `n - i`. If at any point the values do not match, we return `false`, and every call up the stack checks for that value.

But wait, you might ask, sometimes we said we'll return a `boolean` value, and sometimes we're returning a node. Which is it?

It's both. We create a simple class with two members, a `boolean` and a node, and return an instance of that class.

```
1  class Result {
2     public LinkedListNode node;
3     public boolean result;
4  }
```

The example below illustrates the parameters and return values from this sample list.

```
1  isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
2     isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
3        isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
4           isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
```

```
5            isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
6            returns node 3b, true
7          returns node 2b, true
8        returns node 1b, true
9      returns node 0b, true
10 returns nobe 0b, true
```

Implementing this code is now just a matter of filling in the details.

```
1  Result isPalindromeRecurse(LinkedListNode head, int length) {
2    if (head == null || length == 0) {
3      return new Result(null, true);
4    } else if (length == 1) {
5      return new Result(head.next, true);
6    } else if (length == 2) {
7      return new Result(head.next.next,
8                        head.data == head.next.data);
9    }
10   Result res = isPalindromeRecurse(head.next, length - 2);
11   if (!res.result || res.node == null) {
12     return res;
13   } else {
14     res.result = head.data == res.node.data;
15     res.node = res.node.next;
16     return res;
17   }
18 }
19
20 boolean isPalindrome(LinkedListNode head) {
21   Result p = isPalindromeRecurse(head, listSize(head));
22   return p.result;
23 }
```

Some of you might be wondering why we went through all this effort to create a special `Result` class. Isn't there a better way? Not really—at least not in Java.

However, if we were implementing this in C or C++, we could have passed in a double pointer.

```
1  bool isPalindromeRecurse(Node head, int length, Node** next) {
2    ...
3  }
```

It's ugly, but it works.

# Stacks and Queues

*Data Structures: Solutions* **Chapter 3**

**3.1** *Describe how you could use a single array to implement three stacks.*

*pg 80*

## SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're ok with simply allocating a fixed amount of space for each stack, we can do that. This may mean though that one stack runs out of space, while the others are nearly empty.

Alternatively, we can be flexible in our space allocation, but this significantly increases the complexity of the problem.

### Approach 1: Fixed Division

We can divide the array in three equal parts and allow the individual stack to grow in that limited space. Note: we will use the notation "[" to mean inclusive of an end point and "(" to mean exclusive of an end point.

- For stack 1, we will use `[0, n/3)`.
- For stack 2, we will use `[n/3, 2n/3)`.
- For stack 3, we will use `[2n/3, n)`.

The code for this solution is below.

```
1  int stackSize = 100;
2  int[] buffer = new int [stackSize * 3];
3  int[] stackPointer = {-1, -1, -1}; // pointers to track top element
4
5  void push(int stackNum, int value) throws Exception {
6     /* Check if we have space */
7     if (stackPointer[stackNum] + 1 >= stackSize) { // Last element
8        throw new Exception("Out of space.");
9     }
10    /* Increment stack pointer and then update top value */
11    stackPointer[stackNum]++;
12    buffer[absTopOfStack(stackNum)] = value;
13 }
14
15 int pop(int stackNum) throws Exception {
16    if (stackPointer[stackNum] == -1) {
17       throw new Exception("Trying to pop an empty stack.");
18    }
19    int value = buffer[absTopOfStack(stackNum)]; // Get top
20    buffer[absTopOfStack(stackNum)] = 0; // Clear index
21    stackPointer[stackNum]--; // Decrement pointer
22    return value;
23 }
24
```

```
25 int peek(int stackNum) {
26    int index = absTopOfStack(stackNum);
27    return buffer[index];
28 }
29
30 boolean isEmpty(int stackNum) {
31    return stackPointer[stackNum] == -1;
32 }
33
34 /* returns index of top of stack "stackNum", in absolute terms */
35 int absTopOfStack(int stackNum) {
36    return stackNum * stackSize + stackPointer[stackNum];
37 }
```

If we had additional information about the expected usages of the stacks, then we could modify this algorithm accordingly. For example, if we expected Stack 1 to have many more elements than Stack 2, we could allocate more space to Stack 1 and less space to Stack 2.

**Approach 2: Flexible Divisions**

A second approach is to allow the stack blocks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary.

We will also design our array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning.

Please note that the code for this solution is far more complex than would be appropriate for an interview. You could be responsible for pseudocode, or perhaps the code of individual components, but the entire implementation would be far too challenging.

```
1  /* StackData is a simple class that holds a set of data about each
2   * stack. It does not hold the actual items in the stack. */
3  public class StackData {
4     public int start;
5     public int pointer;
6     public int size = 0;
7     public int capacity;
8     public StackData(int _start, int _capacity) {
9        start = _start;
10       pointer = _start - 1;
11       capacity = _capacity;
12    }
13
14    public boolean isWithinStack(int index, int total_size) {
15       /* Note: if stack wraps, the head (right side) wraps around
16        * to the left. */
17       if (start <= index && index < start + capacity) {
18          // non-wrapping, or "head" (right side) of wrapping case
```

```
19          return true;
20       } else if (start + capacity > total_size &&
21                   index < (start + capacity) % total_size) {
22          // tail (left side) of wrapping case
23          return true;
24       }
25       return false;
26    }
27 }
28
29 public class QuestionB {
30    static int number_of_stacks = 3;
31    static int default_size = 4;
32    static int total_size = default_size * number_of_stacks;
33    static StackData [] stacks = {new StackData(0, default_size),
34       new StackData(default_size, default_size),
35       new StackData(default_size * 2, default_size)};
36    static int [] buffer = new int [total_size];
37
38    public static void main(String [] args) throws Exception  {
39       push(0, 10);
40       push(1, 20);
41       push(2, 30);
42       int v = pop(0);
43       ...
44    }
45
46    public static int numberOfElements() {
47       return stacks[0].size + stacks[1].size + stacks[2].size;
48    }
49
50    public static int nextElement(int index) {
51       if (index + 1 == total_size) return 0;
52       else return index + 1;
53    }
54
55    public static int previousElement(int index) {
56       if (index == 0) return total_size - 1;
57       else return index - 1;
58    }
59
60    public static void shift(int stackNum) {
61       StackData stack = stacks[stackNum];
62       if (stack.size >= stack.capacity) {
63          int nextStack = (stackNum + 1) % number_of_stacks;
64          shift(nextStack); // make some room
65          stack.capacity++;
66       }
67
68       // Shift elements in reverse order
```

```
69      for (int i = (stack.start + stack.capacity - 1) % total_size;
70           stack.isWithinStack(i, total_size);
71           i = previousElement(i)) {
72         buffer[i] = buffer[previousElement(i)];
73      }
74
75      buffer[stack.start] = 0;
76      stack.start = nextElement(stack.start); // move stack start
77      stack.pointer = nextElement(stack.pointer); // move pointer
78      stack.capacity--; // return capacity to original
79   }
80
81   /* Expand stack by shifting over other stacks */
82   public static void expand(int stackNum) {
83      shift((stackNum + 1) % number_of_stacks);
84      stacks[stackNum].capacity++;
85   }
86
87   public static void push(int stackNum, int value)
88         throws Exception {
89      StackData stack = stacks[stackNum];
90      /* Check that we have space */
91      if (stack.size >= stack.capacity) {
92         if (numberOfElements() >= total_size) { // Totally full
93            throw new Exception("Out of space.");
94         } else { // just need to shift things around
95            expand(stackNum);
96         }
97      }
98      /* Find the index of the top element in the array + 1,
99       * and increment the stack pointer */
100     stack.size++;
101     stack.pointer = nextElement(stack.pointer);
102     buffer[stack.pointer] = value;
103  }
104
105  public static int pop(int stackNum) throws Exception {
106     StackData stack = stacks[stackNum];
107     if (stack.size == 0) {
108        throw new Exception("Trying to pop an empty stack.");
109     }
110     int value = buffer[stack.pointer];
111     buffer[stack.pointer] = 0;
112     stack.pointer = previousElement(stack.pointer);
113     stack.size--;
114     return value;
115  }
116
117  public static int peek(int stackNum) {
118     StackData stack = stacks[stackNum];
```

```
119       return buffer[stack.pointer];
120    }
121
122    public static boolean isEmpty(int stackNum) {
123       StackData stack = stacks[stackNum];
124       return stack.size == 0;
125    }
126 }
```

In problems like this, it's important to focus on writing clean, maintainable code. You should use additional classes, as we did with `StackData`, and pull chunks of code into separate methods. Of course, this advice applies to the "real world" as well.

**3.2** *How would you design a stack which, in addition to push and pop, also has a function `min` which returns the minimum element? Push, pop and `min` should all operate in O(1) time.*

*pg 80*

**SOLUTION**

The thing with minimums is that they don't change very often. They only change when a smaller element is added.

One solution is to have just a single `int` value, `minValue`, that's a member of the `Stack` class. When `minValue` is popped from the stack, we search through the stack to find the new minimum. Unfortunately, this would break the constraint that push and pop operate in O(1) time.

To further understand this question, let's walk through it with a short example:

```
push(5); // stack is {5}, min is 5
push(6); // stack is {6, 5}, min is 5
push(3); // stack is {3, 6, 5}, min is 3
push(7); // stack is {7, 3, 6, 5}, min is 3
pop(); // pops 7. stack is {3, 6, 5}, min is 3
pop(); // pops 3. stack is {6, 5}. min is 5.
```

Observe how once the stack goes back to a prior state (`{6, 5}`), the minimum also goes back to its prior state (5). This leads us to our second solution.

If we kept track of the minimum at each state, we would be able to easily know the minimum. We can do this by having each node record what the minimum beneath itself is. Then, to find the `min`, you just look at what the top element thinks is the min.

When you push an element onto the stack, the element is given the current minimum. It sets its "`local min`" to be the `min`.

```
1  public class StackWithMin extends Stack<NodeWithMin> {
2     public void push(int value) {
3        int newMin = Math.min(value, min());
```

```
4         super.push(new NodeWithMin(value, newMin));
5      }
6
7      public int min() {
8         if (this.isEmpty()) {
9            return Integer.MAX_VALUE; // Error value
10        } else {
11           return peek().min;
12        }
13     }
14  }
15
16  class NodeWithMin {
17     public int value;
18     public int min;
19     public NodeWithMin(int v, int min){
20        value = v;
21        this.min = min;
22     }
23  }
```

There's just one issue with this: if we have a large stack, we waste a lot of space by keeping track of the `min` for every single element. Can we do better?

We can (maybe) do a bit better than this by using an additional stack which keeps track of the `mins`.

```
1   public class StackWithMin2 extends Stack<Integer> {
2      Stack<Integer> s2;
3      public StackWithMin2() {
4         s2 = new Stack<Integer>();
5      }
6
7      public void push(int value){
8         if (value <= min()) {
9            s2.push(value);
10        }
11        super.push(value);
12     }
13
14     public Integer pop() {
15        int value = super.pop();
16        if (value == min()) {
17           s2.pop();
18        }
19        return value;
20     }
21
22     public int min() {
23        if (s2.isEmpty()) {
24           return Integer.MAX_VALUE;
```

```
25        } else {
26           return s2.peek();
27        }
28     }
29  }
```

Why might this be more space efficient? Suppose we had a very large stack and the first element inserted happened to be the minimum. In the first solution, we would be keeping n ints, where n is the size of the stack. In the second solution though, we store just a few pieces of data: a second stack with one element and the members within this stack.

**3.3** *Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure* `SetOfStacks` *that mimics this.* `SetOfStacks` *should be composed of several stacks and should create a new stack once the previous one exceeds capacity.* `SetOfStacks.push()` *and* `SetOfStacks.pop()` *should behave identically to a single stack (that is,* `pop()` *should return the same values as it would if there were just a single stack).*

*FOLLOW UP*

*Implement a function* `popAt(int index)` *which performs a pop operation on a specific sub-stack.*

*pg 80*

## SOLUTION

In this problem, we've been told what our data structure should look like:

```
1  class SetOfStacks {
2     ArrayList<Stack> stacks = new ArrayList<Stack>();
3     public void push(int v) { ... }
4     public int pop() { ... }
5  }
```

We know that `push()` should behave identically to a single stack, which means that we need `push()` to call `push()` on the last stack in the array of stacks. We have to be a bit careful here though: if the last stack is at capacity, we need to create a new stack. Our code should look something like this:

```
1  public void push(int v) {
2     Stack last = getLastStack();
3     if (last != null && !last.isFull()) { // add to last stack
4        last.push(v);
5     } else { // must create new stack
6        Stack stack = new Stack(capacity);
7        stack.push(v);
8        stacks.add(stack);
9     }
```

```
10 }
```

What should pop() do? It should behave similarly to push() in that it should operate on the last stack. If the last stack is empty (after popping), then we should remove the stack from the list of stacks.

```
1  public int pop() {
2     Stack last = getLastStack();
3     int v = last.pop();
4     if (last.size == 0) stacks.remove(stacks.size() - 1);
5     return v;
6  }
```

**Follow Up: Implement popAt(int index)**

This is a bit trickier to implement, but we can imagine a "rollover" system. If we pop an element from stack 1, we need to remove the *bottom* of stack 2 and push it onto stack 1. We then need to rollover from stack 3 to stack 2, stack 4 to stack 3, etc.

You could make an argument that, rather than "rolling over," we should be OK with some stacks not being at full capacity. This would improve the time complexity (by a fair amount, with a large number of elements), but it might get us into tricky situations later on if someone assumes that all stacks (other than the last) operate at full capacity. There's no "right answer" here; you should discuss this trade-off with your interviewer.

```
1  public class SetOfStacks {
2     ArrayList<Stack> stacks = new ArrayList<Stack>();
3     public int capacity;
4     public SetOfStacks(int capacity) {
5        this.capacity = capacity;
6     }
7
8     public Stack getLastStack() {
9        if (stacks.size() == 0) return null;
10       return stacks.get(stacks.size() - 1);
11    }
12
13    public void push(int v) { /* see earlier code */ }
14    public int pop() { /* see earlier code */ }
15    public boolean isEmpty() {
16       Stack last = getLastStack();
17       return last == null || last.isEmpty();
18    }
19
20    public int popAt(int index) {
21       return leftShift(index, true);
22    }
23
24    public int leftShift(int index, boolean removeTop) {
25       Stack stack = stacks.get(index);
26       int removed_item;
```

```
27        if (removeTop) removed_item = stack.pop();
28        else removed_item = stack.removeBottom();
29        if (stack.isEmpty()) {
30           stacks.remove(index);
31        } else if (stacks.size() > index + 1) {
32           int v = leftShift(index + 1, false);
33           stack.push(v);
34        }
35        return removed_item;
36     }
37  }
38
39  public class Stack {
40     private int capacity;
41     public Node top, bottom;
42     public int size = 0;
43
44     public Stack(int capacity) { this.capacity = capacity; }
45     public boolean isFull() { return capacity == size; }
46
47     public void join(Node above, Node below) {
48        if (below != null) below.above = above;
49        if (above != null) above.below = below;
50     }
51
52     public boolean push(int v) {
53        if (size >= capacity) return false;
54        size++;
55        Node n = new Node(v);
56        if (size == 1) bottom = n;
57        join(n, top);
58        top = n;
59        return true;
60     }
61
62     public int pop() {
63        Node t = top;
64        top = top.below;
65        size--;
66        return t.value;
67     }
68
69     public boolean isEmpty() {
70        return size == 0;
71     }
72
73     public int removeBottom() {
74        Node b = bottom;
75        bottom = bottom.above;
76        if (bottom != null) bottom.below = null;
```

```
77          size--;
78          return b.value;
79      }
80  }
```

This problem is not conceptually that tough, but it requires a lot of code to implement it fully. Your interviewer would not ask you to implement the entire code.

A good strategy on problems like this is to separate code into other methods, like a `leftShift` method that `popAt` can call. This will make your code cleaner and give you the opportunity to lay down the skeleton of the code before dealing with some of the details.

**3.4** *In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:*

*(1) Only one disk can be moved at a time.*

*(2) A disk is slid off the top of one tower onto the next rod.*

*(3) A disk can only be placed on top of a larger disk.*

*Write a program to move the disks from the first tower to the last using Stacks.*

*pg 81*

**SOLUTION**

This problem sounds like a good candidate for the Base Case and Build approach.

Let's start with the smallest possible example: `n = 1`.

Case `n = 1`. Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case `n = 2`. Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move

other disks to Tower 3.

Case n = 3. Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—we just repeat what we did in Step 1.

Case n = 4. Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

```
1  moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
2     /* Base case */
3     if (n <= 0) return;
4
5     /* move top n - 1 disks from origin to buffer, using destination
6      * as a buffer. */
7     moveDisks(n - 1, origin, buffer, destination);
8
9     /* move top from origin to destination
10    moveTop(origin, destination);
11
12    /* move top n - 1 disks from buffer to destination, using
13     * origin as a buffer. */
14    moveDisks(n - 1, buffer, destination, origin);
15 }
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
1  public static void main(String[] args) {
2     int n = 3;
3     Tower[] towers = new Tower[n];
4     for (int i = 0; i < 3; i++) {
5        towers[i] = new Tower(i);
6     }
7
```

```
8     for (int i = n - 1; i >= 0; i--) {
9        towers[0].add(i);
10    }
11    towers[0].moveDisks(n, towers[2], towers[1]);
12 }
13
14 public class Tower {
15    private Stack<Integer> disks;
16    private int index;
17    public Tower(int i) {
18       disks = new Stack<Integer>();
19       index = i;
20    }
21
22    public int index() {
23       return index;
24    }
25
26    public void add(int d) {
27       if (!disks.isEmpty() && disks.peek() <= d) {
28          System.out.println("Error placing disk " + d);
29       } else {
30          disks.push(d);
31       }
32    }
33
34    public void moveTopTo(Tower t) {
35       int top = disks.pop();
36       t.add(top);
37       System.out.println("Move disk " + top + " from " + index() +
38                          " to " + t.index());
39    }
40
41    public void moveDisks(int n, Tower destination, Tower buffer) {
42       if (n > 0) {
43          moveDisks(n - 1, buffer, destination);
44          moveTopTo(destination);
45          buffer.moveDisks(n - 1, destination, this);
46       }
47    }
48 }
```

Implementing the towers as their own object is not strictly necessary, but it does help to make the code cleaner in some respects.

**3.5** *Implement a MyQueue class which implements a queue using two stacks.*

*pg 81*

### SOLUTION

Since the major difference between a queue and a stack is the order (first-in first-out vs. last-in first-out), we know that we need to modify peek() and pop() to go in reverse order. We can use our second stack to reverse the order of the elements (by popping s1 and pushing the elements on to s2). In such an implementation, on each peek() and pop() operation, we would pop everything from s1 onto s2, perform the peek / pop operation, and then push everything back.

This will work, but if two pop / peeks are performed back-to-back, we're needlessly moving elements. We can implement a "lazy" approach where we let the elements sit in s2 until we absolutely must reverse the elements.

In this approach, stackNewest has the newest elements on top and stackOldest has the oldest elements on top. When we dequeue an element, we want to remove the oldest element first, and so we dequeue from stackOldest. If stackOldest is empty, then we want to transfer all elements from stackNewest into this stack in reverse order. To insert an element, we push onto stackNewest, since it has the newest elements on top.

The code below implements this algorithm.

```
1  public class MyQueue<T> {
2    Stack<T> stackNewest, stackOldest;
3
4    public MyQueue() {
5      stackNewest = new Stack<T>();
6      stackOldest = new Stack<T>();
7    }
8
9    public int size() {
10     return stackNewest.size() + stackOldest.size();
11   }
12
13   public void add(T value) {
14     /* Push onto stackNewest, which always has the newest
15      * elements on top */
16     stackNewest.push(value);
17   }
18
19   /* Move elements from stackNewest into stackOldest. This is
20    * usually done so that we can do operations on stackOldest. */
21   private void shiftStacks() {
22     if (stackOldest.isEmpty()) {
23       while (!stackNewest.isEmpty()) {
24         stackOldest.push(stackNewest.pop());
25       }
26     }
27   }
```

```
28
29    public T peek() {
30       shiftStacks(); // Ensure stackOldest has the current elements
31       return stackOldest.peek(); // retrieve the oldest item.
32    }
33
34    public T remove() {
35       shiftStacks(); // Ensure stackOldest has the current elements
36       return stackOldest.pop(); // pop the oldest item.
37    }
38 }
```

During your actual interview, you may find that you forget the exact API calls. Don't stress too much if that happens to you. Most interviewers are okay with your asking for them to refresh your memory on little details. They're much more concerned with your big picture understanding.

**3.6** *Write a program to sort a stack in ascending order (with biggest items on top). You may use at most one additional stack to hold items, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.*

*pg 81*

**SOLUTION**

One approach is to implement a rudimentary sorting algorithm. We search through the entire stack to find the minimum element and then push that onto a new stack. Then, we find the new minimum element and push that. This will actually require a total of three stacks: s1 is the original stack, s2 is the final sorted stack, and s3 acts as a buffer during our searching of s1. To search s1 for each minimum, we need to pop elements from s1 and push them onto the buffer, s3.

Unfortunately, we're only allowed one additional stack. Can we do better? Yes.

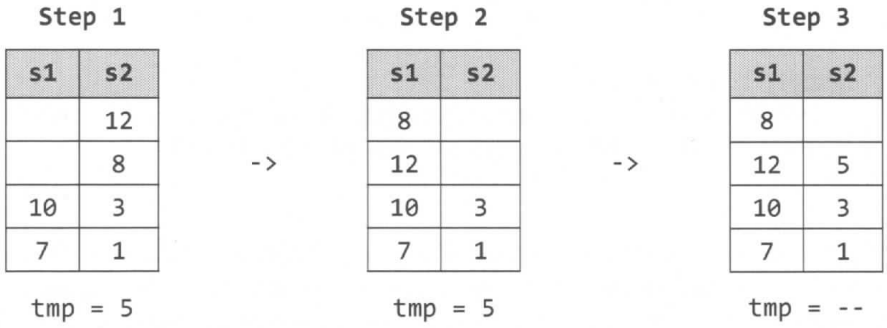
Rather than searching for the minimum repeatedly, we can sort s1 by inserting each element from s1 in order into s2. How would this work?

Imagine we have the following stacks, where s2 is "sorted" and s1 is not:

| s1 | s2 |
|---|---|
| | 12 |
| 5 | 8 |
| 10 | 3 |
| 7 | 1 |

When we pop 5 from s1, we need to find the right place in s2 to insert this number. In this case, the correct place is on s2 just above 3. How do we get it there? We can do this

by popping 5 from s1 and holding it in a temporary variable. Then, we move 12 and 8 over to s1 (by popping them from s2 and pushing them onto s1) and then push 5 onto s2.




Note that 8 and 12 are still in s1 -- and that's okay! We just repeat the same steps for those two numbers as we did for 5, each time popping off the top of s1 and putting it into the "right place" on s2. (Of course, since 8 and 12 were moved from s2 to s1 precisely *because* they were larger than 5, the "right place" for these elements will be right on top of 5. We won't need to muck around with s2's other elements, and the inside of the below `while` loop will not be run when `tmp` is 8 or 12.)

```
1  public static Stack<Integer> sort(Stack<Integer> s) {
2     Stack<Integer> r = new Stack<Integer>();
3     while (!s.isEmpty()) {
4        int tmp = s.pop(); // Step 1
5        while (!r.isEmpty() && r.peek() > tmp) { // Step 2
6           s.push(r.pop());
7        }
8        r.push(tmp); // Step 3
9     }
10    return r;
11 }
```

This algorithm is $O(N^2)$ time and $O(N)$ space.

If we were allowed to use unlimited stacks, we could implement a modified quicksort or mergesort.

With the mergesort solution, we would create two extra stacks and divide the stack into two parts. We would recursively sort each stack, and then merge them back together in sorted order into the original stack. Note that this would require the creation of two additional stacks per level of recursion.

With the quicksort solution, we would create two additional stacks and divide the stack into the two stacks based on a pivot element. The two stacks would be recursively sorted, and then merged back together into the original stack. Like the earlier solution, this one involves creating two additional stacks per level of recursion.

**3.7** *An animal shelter holds only dogs and cats, and operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog and dequeueCat. You may use the built-in `LinkedList` data structure.*

*pg 81*

**SOLUTION**

We could explore a variety of solutions to this problem. For instance, we could maintain a single queue. This would make dequeueAny easy, but dequeueDog and dequeueCat would require iteration through the queue to find the first dog or cat. This would increase the complexity of the solution and decrease the efficiency.

An alternative approach that is simple, clean and efficient is to simply use separate queues for dogs and cats, and to place them within a wrapper class called `Animal-Queue`. We then store some sort of timestamp to mark when each animal was enqueued. When we call dequeueAny, we peek at the heads of both the dog and cat queue and return the oldest.

```
1  public abstract class Animal {
2     private int order;
3     protected String name;
4     public Animal(String n) {
5        name = n;
6     }
7
8     public void setOrder(int ord) {
9        order = ord;
10    }
11
12    public int getOrder() {
13       return order;
14    }
15
16    public boolean isOlderThan(Animal a) {
17       return this.order < a.getOrder();
18    }
19 }
20
21 public class AnimalQueue {
22    LinkedList<Dog> dogs = new LinkedList<Dog>();
23    LinkedList<Cat> cats = new LinkedList<Cat>();
24    private int order = 0; // acts as timestamp
25
26    public void enqueue(Animal a) {
```

```
27        /* Order is used as a sort of timestamp, so that we can
28         * compare the insertion order of a dog to a cat. */
29        a.setOrder(order);
30        order++;
31
32        if (a instanceof Dog) dogs.addLast((Dog) a);
33        else if (a instanceof Cat) cats.addLast((Cat)a);
34     }
35
36     public Animal dequeueAny() {
37        /* Look at tops of dog and cat queues, and pop the stack
38         * with the oldest value. */
39        if (dogs.size() == 0) {
40           return dequeueCats();
41        } else if (cats.size() == 0) {
42           return dequeueDogs();
43        }
44
45        Dog dog = dogs.peek();
46        Cat cat = cats.peek();
47     if (dog.isOlderThan(cat)) {
48        return dequeueDogs();
49     } else {
50        return dequeueCats();
51     }
52
53     public Dog dequeueDogs() {
54        return dogs.poll();
55     }
56
57     public Cat dequeueCats() {
58        return cats.poll();
59     }
60  }
61
62  public class Dog extends Animal {
63     public Dog(String n) {
64        super(n);
65     }
66  }
67
68  public class Cat extends Animal {
69     public Cat(String n) {
70        super(n);
71     }
72  }
```