```
public class Trie {
  // The root of this trie.
  private TrieNode root;

  /* Takes a list of strings as an argument, and constructs a trie that stores
   * these strings. */
  public Trie(ArrayList<String> list) {
    root = new TrieNode();
    for (String word : list) {
      root.addWord(word);
    }
  }


  /* Takes a list of strings as an argument, and constructs a trie that stores
   * these strings. */
  public Trie(String[] list) {
    root = new TrieNode();
    for (String word : list) {
      root.addWord(word);
    }
  }

  /* Checks whether this trie contains a string with the prefix passed in as
   * argument. */
  public boolean contains(String prefix, boolean exact) {
    TrieNode lastNode = root;
    int i = 0;
    for (i = 0; i < prefix.length(); i++) {
      lastNode = lastNode.getChild(prefix.charAt(i));
      if (lastNode == null) {
        return false;
      }
    }
    return !exact || lastNode.terminates();
  }

  public boolean contains(String prefix) {
    return contains(prefix, false);
  }

  public TrieNode getRoot() {
    return root;
  }
}
```

The Trie class uses the TrieNode class, which is implemented below.

```
public class TrieNode {
  /* The children of this node in the trie.*/
  private HashMap<Character, TrieNode> children;
  private boolean terminates = false;

  /* The character stored in this node as data.*/
  private char character;

  /* Constructs an empty trie node and initializes the list of its children to an
   * empty hash map. Used only to construct the root node of the trie. */
```

```
11    public TrieNode() {
12       children = new HashMap<Character, TrieNode>();
13    }
14
15    /* Constructs a trie node and stores this character as the node's value.
16     * Initializes the list of child nodes of this node to an empty hash map. */
17    public TrieNode(char character) {
18       this();
19       this.character = character;
20    }
21
22    /* Returns the character data stored in this node. */
23    public char getChar() {
24       return character;
25    }
26
27    /* Add this word to the trie, and recursively create the child
28     * nodes. */
29    public void addWord(String word) {
30       if (word == null || word.isEmpty()) {
31          return;
32       }
33
34       char firstChar = word.charAt(0);
35
36       TrieNode child = getChild(firstChar);
37       if (child == null) {
38          child = new TrieNode(firstChar);
39          children.put(firstChar, child);
40       }
41
42       if (word.length() > 1) {
43          child.addWord(word.substring(1));
44       } else {
45          child.setTerminates(true);
46       }
47    }
48
49    /* Find a child node of this node that has the char argument as its data. Return
50     * null if no such child node is present in the trie. */
51    public TrieNode getChild(char c) {
52       return children.get(c);
53    }
54
55    /* Returns whether this node represents the end of a complete word. */
56    public boolean terminates() {
57       return terminates;
58    }
59
60    /* Set whether this node is the end of a complete word.*/
61    public void setTerminates(boolean t) {
62       terminates = t;
63    }
64 }
```

Hints

XIII

Interviewers usually don't just hand you a question and expect you to solve it. Rather, they will typically offer guidance when you're stuck, especially on the harder questions. It's impossible to totally simulate the interview experience in a book, but these hints are designed to get you closer.

Try to solve the questions independently when possible. But it's okay to look for some help when you are really struggling. Again, struggling is a normal part of the process.

I've organized the hints somewhat randomly here, such that all the hints for a problem aren't adjacent. This way you won't accidentally see the second hint when you're reading the first hint.

I

Hints for Data Structures

#1. 1.2 Describe what it means for two strings to be permutations of each other. Now, look at that definition you provided. Can you check the strings against that definition?

#2. 3.1 A stack is simply a data structure in which the most recently added elements are removed first. Can you simulate a single stack using an array? Remember that there are many possible solutions, and there are tradeoffs of each.

#3. 2.4 There are many solutions to this problem, most of which are equally optimal in runtime. Some have shorter, cleaner code than others. Can you brainstorm different solutions?

#4. 4.10 If T2 is a subtree of T1, how will its in-order traversal compare to T1's? What about its pre-order and post-order traversal?

#5. 2.6 A palindrome is something which is the same when written forwards and backwards. What if you reversed the linked list?

#6. 4.12 Try simplifying the problem. What if the path had to start at the root?

#7. 2.5 Of course, you could convert the linked lists to integers, compute the sum, and then convert it back to a new linked list. If you did this in an interview, your interviewer would likely accept the answer, and then see if you could do this without converting it to a number and back.

#8. 2.2 What if you knew the linked list size? What is the difference between finding the Kth-to-last element and finding the Xth element?

#9. 2.1 Have you tried a hash table? You should be able to do this in a single pass of the linked list.

#10. 4.8 If each node has a link to its parent, we could leverage the approach from question 2.7 on page 95. However, our interviewer might not let us make this assumption.

#11. 4.10 The in-order traversals won't tell us much. After all, every binary search tree with the same values (regardless of structure) will have the same in-order traversal. This is what in-order traversal means: contents are in-order. (And if it won't work in the specific case of a binary search tree, then it certainly won't work for a general binary tree.) The pre-order traversal, however, is much more indicative.

#12. 3.1 We could simulate three stacks in an array by just allocating the first third of the array to the first stack, the second third to the second stack, and the final third to the third stack. One might actually be much bigger than the others, though. Can we be more flexible with the divisions?

#13. 2.6 Try using a stack.

#14. 4.12 Don't forget that paths could overlap. For example, if you're looking for the sum 6, the paths `1->3->2` and `1->3->2->4->-6->2` are both valid.

#15. 3.5 One way of sorting an array is to iterate through the array and insert each element into a new array in sorted order. Can you do this with a stack?

#16. 4.8 The first common ancestor is the deepest node such that p and q are both descendants. Think about how you might identify this node.

#17. 1.8 If you just cleared the rows and columns as you found 0s, you'd likely wind up clearing the whole matrix. Try finding the cells with zeros first before making any changes to the matrix.

#18. 4.10 You may have concluded that if `T2.preorderTraversal()` is a substring of `T1.preorderTraversal()`, then `T2` is a subtree of `T1`. This is almost true, except that the trees could have duplicate values. Suppose `T1` and `T2` have all duplicate values but different structures. The pre-order traversals will look the same even though `T2` is not a subtree of `T1`. How can you handle situations like this?

#19. 4.2 A minimal binary tree has about the same number of nodes on the left of each node as on the right. Let's focus on just the root for now. How would you ensure that about the same number of nodes are on the left of the root as on the right?

#20. 2.7 You can do this in `O(A+B)` time and `O(1)` additional space. That is, you do not need a hash table (although you could do it with one).

#21. 4.4 Think about the definition of a balanced tree. Can you check that condition for a single node? Can you check it for every node?

#22. 3.6 We could consider keeping a single linked list for dogs and cats, and then iterating through it to find the first dog (or cat). What is the impact of doing this?

#23. 1.5 Start with the easy thing. Can you check each of the conditions separately?

#24. 2.4 Consider that the elements don't have to stay in the same relative order. We only need to ensure that elements less than the pivot must be before elements greater than the pivot. Does that help you come up with more solutions?

#25. 2.2 If you don't know the linked list size, can you compute it? How does this impact the runtime?

#26. 4.7 Build a directed graph representing the dependencies. Each node is a project and an edge exists from A to B if B depends on A (A must be built before B). You can also build it the other way if it's easier for you.

#27. 3.2 Observe that the minimum element doesn't change very often. It only changes when a smaller element is added, or when the smallest element is popped.

#28. 4.8 How would you figure out if p is a descendent of a node n?

#29. 2.6 Assume you have the length of the linked list. Can you implement this recursively?

#30. 2.5 Try recursion. Suppose you have two lists, A = `1->5->9` (representing 951) and B = `2->3->6->7` (representing 7632), and a function that operates on the remainder of the lists (`5->9` and `3->6->7`). Could you use this to create the `sum` method? What is the relationship between `sum(1->5->9, 2->3->6->7)` and `sum(5->9, 3->6->7)`?

#31. 4.10 Although the problem seems like it stems from duplicate values, it's really deeper than that. The issue is that the pre-order traversal is the same only because there are null nodes that we skipped over (because they're null). Consider inserting a placeholder value into the pre-order traversal string whenever you reach a null node. Register the null node as a "real" node so that you can distinguish between the different structures.

#32. 3.5 Imagine your secondary stack is sorted. Can you insert elements into it in sorted order? You might need some extra storage. What could you use for extra storage?

#33. 4.4 If you've developed a brute force solution, be careful about its runtime. If you are computing the height of the subtrees for each node, you could have a pretty inefficient algorithm.

#34. 1.9 If a string is a rotation of another, then it's a rotation at a particular point. For example, a rotation of `waterbottle` at character 3 means cutting `waterbottle` at character 3 and putting the right half (`erbottle`) before the left half (`wat`).

#35. 4.5 If you traversed the tree using an in-order traversal and the elements were truly in the right order, does this indicate that the tree is actually in order? What happens for duplicate elements? If duplicate elements are allowed, they must be on a specific side (usually the left).

#36. 4.8 Start with the root. Can you identify if root is the first common ancestor? If it is not, can you identify which side of root the first common ancestor is on?

#37. 4.10 Alternatively, we can handle this problem recursively. Given a specific node within T1, can we check to see if its subtree matches T2?

#38. 3.1 If you want to allow for flexible divisions, you can shift stacks around. Can you ensure that all available capacity is used?

#39. 4.9 What is the very first value that must be in each array?

#40. 2.1 Without extra space, you'll need $O(N^2)$ time. Try using two pointers, where the second one searches ahead of the first one.

#41. 2.2 Try implementing it recursively. If you could find the (K-1)th to last element, can you find the Kth element?

#42. 4.11 Be very careful in this problem to ensure that each node is equally likely and that your solution doesn't slow down the speed of standard binary search tree algorithms (like `insert`, `find`, and `delete`). Also, remember that even if you assume that it's a balanced binary search tree, this doesn't mean that the tree is full/complete/perfect.

#43. 3.5 Keep the secondary stack in sorted order, with the biggest elements on the top. Use the primary stack for additional storage.

#44. 1.1 Try a hash table.

#45. 2.7 Examples will help you. Draw a picture of intersecting linked lists and two equivalent linked lists (by value) that do not intersect.

#46. 4.8 Try a recursive approach. Check if p and q are descendants of the left subtree and the right subtree. If they are descendants of different subtrees, then the current node is the first common ancestor. If they are descendants of the same subtree, then that subtree holds the first common ancestor. Now, how do you implement this efficiently?

#47. 4.7 Look at this graph. Is there any node you can identify that will definitely be okay to build first?

#48. 4.9 The root is the very first value that must be in every array. What can you say about the order of the values in the left subtree as compared to the values in the right subtree? Do the left subtree values need to be inserted before the right subtree?

#49. 4.4 What if you could modify the binary tree node class to allow a node to store the height of its subtree?

#50. 2.8 There are really two parts to this problem. First, detect if the linked list has a loop. Second, figure out where the loop starts.

#51. 1.7 Try thinking about it layer by layer. Can you rotate a specific layer?

#52. 4.12 If each path had to start at the root, we could traverse all possible paths starting from the root. We can track the sum as we go, incrementing `totalPaths` each time we find a path with our target sum. Now, how do we extend this to paths that can start anywhere? Remember: Just get a brute-force algorithm done. You can optimize later.

#53. 1.3 It's often easiest to modify strings by going from the end of the string to the beginning.

#54. 4.11 This is your own binary search tree class, so you can maintain any information about the tree structure or nodes that you'd like (provided it doesn't have other negative implications, like making `insert` much slower). In fact, there's probably a reason the interview question specified that it was your own class. You probably need to store some additional information in order to implement this efficiently.

#55. 2.7 Focus first on just identifying if there's an intersection.

#56. 3.6 Let's suppose we kept separate lists for dogs and cats. How would we find the oldest animal of any type? Be creative!

#57. 4.5 To be a binary search tree, it's not sufficient that the `left.value <= current.value < right.value` for each node. Every node on the left must be less than the current node, which must be less than all the nodes on the right.

#58. 3.1 Try thinking about the array as circular, such that the end of the array "wraps around" to the start of the array.

#59. 3.2 What if we kept track of extra data at each stack node? What sort of data might make it easier to solve the problem?

#60. 4.7 If you identify a node without any incoming edges, then it can definitely be built. Find this node (there could be multiple) and add it to the build order. Then, what does this mean for its outgoing edges?

#61. 2.6 In the recursive approach (we have the length of the list), the middle is the base case: `isPermutation(middle)` is true. The node `x` to the immediate left of the middle: What can that node do to check if `x->middle->y` forms a palindrome? Now suppose that checks out. What about the previous node `a`? If `x->middle->y` is a palindrome, how can it check that `a->x->middle->y->b` is a palindrome?

#62. 4.11 As a naive "brute force" algorithm, can you use a tree traversal algorithm to implement this algorithm? What is the runtime of this?

#63. 3.6 Think about how you'd do it in real life. You have a list of dogs in chronological order and a list of cats in chronological order. What data would you need to find the oldest animal? How would you maintain this data?

#64. 3.3 You will need to keep track of the size of each substack. When one stack is full, you may need to create a new stack.

#65. 2.7 Observe that two intersecting linked lists will always have the same last node. Once they intersect, all the nodes after that will be equal.

#66. 4.9 The relationship between the left subtree values and the right subtree values is, essentially, anything. The left subtree values could be inserted before the right subtree, or the reverse (right values before left), or any other ordering.

#67. 2.2 You might find it useful to return multiple values. Some languages don't directly support this, but there are workarounds in essentially any language. What are some of those workarounds?

#68. 4.12 To extend this to paths that start anywhere, we can just repeat this process for all nodes.

#69. 2.8 To identify if there's a cycle, try the "runner" approach described on page 93. Have one pointer move faster than the other.

#70. 4.8 In the more naive algorithm, we had one method that indicated if `x` is a descendent of n, and another method that would recurse to find the first common ancestor. This is repeatedly searching the same elements in a subtree. We should merge this into one `firstCommonAncestor` function. What return values would give us the information we need?

#71. 2.5 Make sure you have considered linked lists that are not the same length.

#72. 2.3 Picture the list `1->5->9->12`. Removing 9 would make it look like `1->5->12`. You only have access to the 9 node. Can you make it look like the correct answer?

#73. 4.2 You could implement this by finding the "ideal" next element to add and repeatedly calling `insertValue`. This will be a bit inefficient, as you would have to repeatedly traverse the tree. Try recursion instead. Can you divide this problem into subproblems?

#74. 1.8 Can you use $O(N)$ additional space instead of $O(N^2)$? What information do you really need from the list of cells that are zero?

#75. 4.11 Alternatively, you could pick a random depth to traverse to and then randomly traverse, stopping when you get to that depth. Think this through, though. Does this work?

#76. 2.7 You can determine if two linked lists intersect by traversing to the end of each and comparing their tails.

#77. 4.12 If you've designed the algorithm as described thus far, you'll have an $O(N \log N)$ algorithm in a balanced tree. This is because there are N nodes, each of which is at depth $O(\log N)$ at worst. A node is touched once for each node above it. Therefore, the N nodes will be touched $O(\log N)$ time. There is an optimization that will give us an $O(N)$ algorithm.

#78. 3.2 Consider having each node know the minimum of its "substack" (all the elements beneath it, including itself).

#79. 4.6 Think about how an in-order traversal works and try to "reverse engineer" it.

#80. 4.8 The `firstCommonAncestor` function could return the first common ancestor (if p and q are both contained in the tree), p if p is in the tree and not q, q if q is in the tree and not p, and `null` otherwise.

#81. 3.3 Popping an element at a specific substack will mean that some stacks aren't at full capacity. Is this an issue? There's no right answer, but you should think about how to handle this.

#82. 4.9 Break this down into subproblems. Use recursion. If you had all possible sequences for the left subtree and the right subtree, how could you create all possible sequences for the entire tree?

#83. 2.8 You can use two pointers, one moving twice as fast as the other. If there is a cycle, the two pointers will collide. They will land at the same location at the same time. Where do they land? Why there?

#84. 1.2 There is one solution that is `O(N log N)` time. Another solution uses some space, but is `O(N)` time.

#85. 4.7 Once you decide to build a node, its outgoing edge can be deleted. After you've done this, can you find other nodes that are free and clear to build?

#86. 4.5 If every node on the left must be less than or equal to the current node, then this is really the same thing as saying that the biggest node on the left must be less than or equal to the current node.

#87. 4.12 What work is duplicated in the current brute-force algorithm?

#88. 1.9 We are essentially asking if there's a way of splitting the first string into two parts, `x` and `y`, such that the first string is `xy` and the second string is `yx`. For example, `x = wat` and `y = erbottle`. The first string is `xy = waterbottle`. The second string is `yx = erbottlewat`.

#89. 4.11 Picking a random depth won't help us much. First, there's more nodes at lower depths than higher depths. Second, even if we re-balanced these probabilities, we could hit a "dead end" where we meant to pick a node at depth 5 but hit a leaf at depth 3. Re-balancing the probabilities is an interesting , though.

#90. 2.8 If you haven't identified the pattern of where the two pointers start, try this: Use the linked list 1->2->3->4->5->6->7->8->9->?, where the ? links to another node. Try making the ? the first node (that is, the 9 points to the 1 such that the entire linked list is a loop). Then make the ? the node 2. Then the node 3. Then the node 4. What is the pattern? Can you explain why this happens?

#91. 4.6 Here's one step of the logic: The successor of a specific node is the leftmost node of the right subtree. What if there is no right subtree, though?

#92. 1.6 Do the easy thing first. Compress the string, then compare the lengths.

#93. 2.7 Now, you need to find where the linked lists intersect. Suppose the linked lists were the same length. How could you do this?

#94. 4.12 Consider each path that starts from the root (there are N such paths) as an array. What our brute-force algorithm is really doing is taking each array and finding all contiguous subsequences that have a particular sum. We're doing this by computing all subarrays and their sums. It might be useful to just focus on this little subproblem. Given an array, how would you find all contiguous subsequences with a particular sum? Again, think about the duplicated work in the brute-force algorithm.

#95. 2.5 Does your algorithm work on linked lists like 9->7->8 and 6->8->5? Double check that.

#96. 4.8 Careful! Does your algorithm handle the case where only one node exists? What will happen? You might need to tweak the return values a bit.

#97. 1.5 What is the relationship between the "insert character" option and the "remove character" option? Do these need to be two separate checks?

#98. 3.4 The major difference between a queue and a stack is the order of elements. A queue removes the oldest item and a stack removes the newest item. How could you remove the oldest item from a stack if you only had access to the newest item?

#99. 4.11 A naive approach that many people come up with is to pick a random number between 1 and 3. If it's 1, return the current node. If it's 2, branch left. If it's 3, branch right. This solution doesn't work. Why not? Is there a way you can adjust it to make it work?

#100. 1.7 Rotating a specific layer would just mean swapping the values in four arrays. If you were asked to swap the values in two arrays, could you do this? Can you then extend it to four arrays?

#101. 2.6 Go back to the previous hint. Remember: There are ways to return multiple values. You can do this with a new class.

#102. 1.8 You probably need some data storage to maintain a list of the rows and columns that need to be zeroed. Can you reduce the additional space usage to `O(1)` by using the matrix itself for data storage?

#103. 4.12 We are looking for subarrays with sum `targetSum`. Observe that we can track in constant time the value of $\texttt{runningSum}_i$, where this is the sum from element 0 through element `i`. For a subarray of element `i` through element `j` to have sum `targetSum`, $\texttt{runningSum}_{i-1}$ + `targetSum` must equal $\texttt{runningSum}_j$ (try drawing a picture of an array or a number line). Given that we can track the `runningSum` as we go, how can we quickly look up the number of indices `i` where the previous equation is true?

#104. 1.9 Think about the earlier hint. Then think about what happens when you concatenate `erbottlewat` to itself. You get `erbottlewaterbottlewat`.

#105. 4.4 You don't need to modify the binary tree class to store the height of the subtree. Can your recursive function compute the height of each subtree while also checking if a node is balanced? Try having the function return multiple values.

#106. 1.4 You do not have to—and should not—generate all permutations. This would be very inefficient.

#107. 4.3 Try modifying a graph search algorithm to track the depth from the root.

#108. 4.12 Try using a hash table that maps from a `runningSum` value to the number of elements with this `runningSum`.