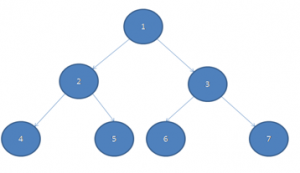
Amazon interview Questions:

1. Zig-zag traversal of a binary tree:  
   Given a binary tree as:  
     
   Make an algo that prints 1 3 2 4 5 6 7 (and more)  
   **Solution:** use a queue and a stack. Perform a level-order traversal using the Queue (condition: while queue is not empty). Keep track of the level by using a wrapper structure.   
   Assume the level of the root is 1  
   If the current level is an odd number, just print when you dequeue and don’t use the stack.   
   If the level is an even number, push onto the stack when you dequeue. Then, once the level *changes*, pop and print all from the stack. Doing so ensures that every even-numbered level gets printed in the reverse order it would during regular level-order traversal. You might need to pop all from the stack after the loop.  
   Code: [in C++](https://github.com/ARDivekar/Algorithms/blob/master/Interview%20Practice/Amazon/zigzag%20Binary%20Tree%20traversal.cpp)
2. Rotate a matrix by 90 degrees: this is a common question, and it’s also in Cracking the Coding Interview.  
   Basically, if it’s a square matrix, you do this: rotate the outermost ring first (i.e. the four overlapping arrays on the edges) by 90 degrees. Then you recursively do the same for the inner rings, from the outside in.   
   For an mxn matrix, if m!=n, then there’s a problem of actual space: you have to store it in a completely different set of arrays. So, might as well just copy it.
3. Rotate k alternate nodes of a linked list:  
   Example1:  
   Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3   
   Output: 3->2->1->6->5->4->8->7->NULL.   
   Example2:  
   Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5  
   Output: 5->4->3->2->1->8->7->6->NULL.  
     
   **Solution:** The simplest solution is disconnect every k nodes, pass it to a function that reverses linked lists, then reconnect it. This basically goes over the list twice. It requires constant auxiliary space (not even for the call stack). The time is O(n).  
   An iterative solution can also be done by reversing as we traverse the list. We keep a temporary head pointer for the local head, and effectively reverse the first ‘k’ nodes of the linked list starting from that head. The same can be done recursively, and is implemented [here](http://www.geeksforgeeks.org/reverse-a-list-in-groups-of-given-size/).  
   Code: in C++
4. Find the first non-repeating character in a string:  
   This is simple if we have a constant character range (it may be a very large constant, like Unicode, but constant nonetheless): just use a Hashmap of the counts of each character. To get the *first* non-repeating character, use a doubly linked list that links all the characters as they are seen. Eg: “this is a call” will be linked as “t->h->i->s->a>c>l”. We only add a new node at the end of the LL if a character has not been seen before (verified with the hashmap). At the end, we go over the linked list once, from the head, and eliminate nodes which have characters that have counts more than one. So, in our pruned linked list, we only have characters which appear once, in chronological order. Since we want the first non-repeating, we can return the head, but we can also get the last, second-last etc. with this method. This entire solution is O(N) time, where N is the length of our string.   
     
   This problem can be extended by saying that we have an infinite stream of characters. The above solution doesn’t exactly work for this case, as there is no ‘stop’ point after which we prune our linked list. So, we prune on the go: if the character in the head node appears for the second time, then we remove it and go to the next. We check the next, to verify that it appears only once. If it does not, we remove it too. So on, until we reach a non-repeating character or have an empty list. The good thing about this method is that since a node removed from the list cannot enter back into it at some later point, we will have at most |characters| number of nodes to prune, which is constant.  
     
   Another extension of the problem is when we have a non-constant number of characters, but still want to have constant time i.e. O(1). Luckily, finding the first non-repeating one is the part of the problem that does not change. What changes is how we use our hash table. Having a non-constant number of characters means that we will probably have to use something like linear probing with table doubling, or make linked lists of the hash table. I prefer the first method (linear probing) as it uses less space (no pointers), and because the linked list-hash table will get very slow if we grossly underestimate the number of unique characters we have. Table doubling ensures that linear probing is fast on average, even if it is expensive on occasion, even if we have a LOT of characters.
5. Longest palindromic substring: find the longest palindrome (i.e. same backwards and forwards) in a string.  
   **Solution:** via geeks for geeks in O(n^2) time but O(1) space: consider each index of the array to be the centre of the palindrome. So, starting at that index, grow to the left and the right one character by one, until it is no longer a palindrome or you hit an edge of the string. Store the start and end points of the longest palindrome with that index as the centre. Do this for each index.   
   Note: when checking the centre points, there are two cases: an even-numbered palindrome and an odd-numbered one. The even-numbered one requires we take that as the centre index, and then one before it, check if they are the same character, and *then* grow. Else, we use the odd-length palindrome, where we just grow from the index.   
   Side note: maybe something like Rabin-Karp will let you do this faster.
6. Special stack: push, pop and min in O(1): push and pop are easy. Min is the problem: we want to *repeatedly* be able to get the min in O(1) time, while also maintaining the O(1) time for push and pop. This makes things tricky. But understand one thing: we can only access the top of the stack. So, we don’t need to save the minimum of all the elements: we just need to save the minimum for all the elements *currently in the stack*.  
   So, we use two stacks: one for the actual data and one auxiliary stack, which, stores the minimum seen **so far**. So, (assuming we append at the end) Aux[i] stores the minimum of stack S[0…i]. If S[i] had been greater than S[i-1], Aux[i]=Aux[i-1],i.e. we store repeat minimums in the Auxiliary stack. Thus, it stores the *minimum seen so far*. [For more detail](http://www.geeksforgeeks.org/design-and-implement-special-stack-data-structure/).  
   We can space-optimize the Auxiliary stack by compressing the repeated minimums into only one, and when popping from the special stack, check whether to pop from the compressed auxiliary stack also.
7. Left view of a Binary Tree: a level order traversal wherein you only print the first (leftmost) node on each level.   
   Similarly, right view of tree.  
   Doing this with level-order is easy enough. We can also do this recursively with an inorder traversal: since we recursively go left in an inorder traversal, we keep track of the lowest level we have ‘handled’ so far. If the node we are on is at a lower level, we print it and reset the lowest level we have handled so far.   
   On [geeksforgeeks](http://www.geeksforgeeks.org/print-left-view-binary-tree/).
8. Given two linked lists of digits to represent numbers, make a linked list that is the sum of the two numbers.  
   Eg: 1 -> 2 -> 3 -> 4 and 4->3   
   print 1 - > 2 -> 7 -> 7
9. Find the point of looping in a linked list:   
   Solution:  
   The most straightforward implementation is with a hashtable that stores the addresses of the nodes. That’s O(N) space.   
   It’s possible to do it in O(1) space using the slow pointer-fast pointer method (this is called Floyd’s cycle): from the head of the LL, start a slow pointer and a fast pointer (which is 2x the slow pointer). Go until they meet (they will). From the meet point and from the head, start two slow pointers. The point where they meet is the start of the loop. You can prove all this mathematically quite simply.
10. Given a BST, replace each node with the sum of the values of all the nodes that are greater than that node. Only constraint being that you are not allowed to use any global or static variable.  
    Solution:
11. Given an array of numbers find a triplet that satisfies the given condition.   
    Condition: a[i] < a[j] < a[k] where i < j < k.   
    If there are more than one triples, print them all.  
    Solution:  
    This question has a rather surprising conundrum. Ask the interviewer to clarify what s/he means by ‘all’ triples.  
    For a given element at index ‘i’ in the array A[0…n-1], it is possible to know if a triple *exists* with that element as the central element, in O(n) time for all triples. However getting *all* the triples that exist is an O(n^2) task, and generating them is an O(n^3) task (because, there can be at most O(n^3) triples, for a sorted array of unique elements).  
    Both the solutions follow the same pattern: maintain an array smaller[0…n-1] and greater[0…n-1]. These are assumed to be initialized to -1. If there exists an element A[i] that has an element smaller than it in A[0…i-1], then we set smaller[i] as the index of that element. Similarly, we set greater[i] to be the index of an element, in A[i+1…n-1], if it exists. Then, we iterate through A[i] a second time, and if smaller[i]!=-1 and greater[i]!=-1, we print the triple( A[ smaller[i] ] , A[i], A[ greater[i] ] ).   
    So, how do we do this? The answer: we store the min we have seen so far, to set smaller. Concretely:  
    for(i=0; i < n; i++)  
     if(a[i] <= arr[min])  
     min=i  
     smaller[i]=-1  
     else smaller[i]=min

We do a similar thing to set greater, only we start from the other end of the array. Then, we do a final, third traversal to print the triples. The runtime is O(n) with O(n) auxiliary space.  
  
Extending this problem, we may be asked to print **all** triples that exist. There are O(n^3) such triples: for a sorted array of unique elements, a every element with the set of all elements before it, and every element with the set of all elements after it, forms a triple. Just printing all these triples would take O(n^3) time. The funny thing is, we can store all of them in O(n^2) time and O(n^2) space: instead of an array of integers of smaller[i], use an array of hash tables (hash tables to avoid repeat values). Do the same for greater[i]. Now, go over the array exactly O(N^2) times in two nested loops, and if A[j] < A[i] && j<i, append j to smaller[i]. If j>i && A[i]<A[j], append j to greater[i]. Then, for each i, loop through the left and right hash tables (nested) and print all triples. This printing step is O(n^3).