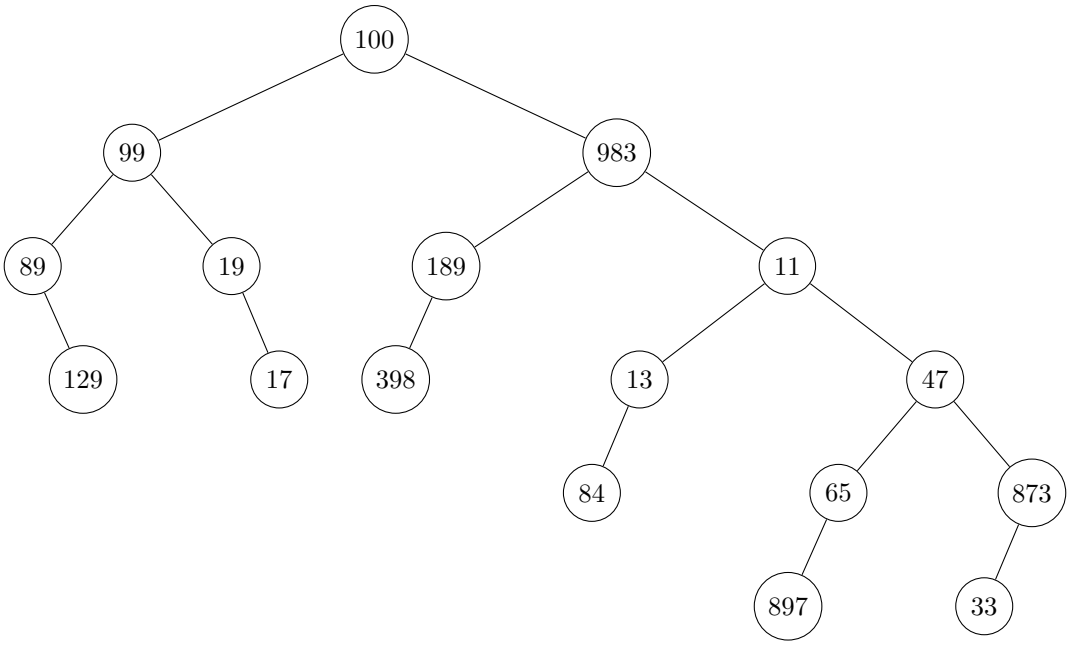


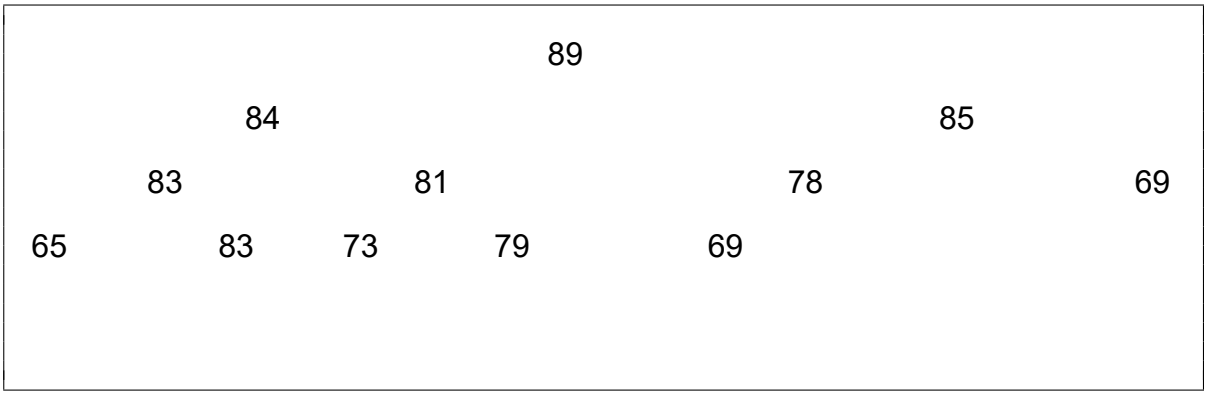

Figure 1: An example binary tree

## Binary Trees

1. For the tree shown in Figure 1,
   (a) Which node is the root? 100
   (b) Which nodes are leaves? 129, 17, 398, 84, 897, 33
   (c) What is the tree's height? 6
   (d) What is the result of preorder traversal through the tree?
   100, 99, 89, 129, 19, 17, 983, 189, 398, 11, 13, 84, 47, 65, 897, 873, 33
   (e) What is the result of postorder traversal through the tree?
   129, 89, 17, 19, 99, 398, 189, 84, 13, 897, 65, 33, 873, 47, 11, 983, 100
   (f) What is the result of inorder traversal through the tree?
   89, 129, 99, 19, 17, 100, 398, 189, 983, 84, 13, 11, 897, 65, 47, 33, 873

## Binary Heaps and Priority Queues

2. Draw the heap that results when the keys `69 65 83 89 81 85 69 83 84 73 79 78` are inserted, in that order, into an initially empty max-oriented heap. When dealing with equal keys during the swim down operation, take the left branch.




3. Suppose that the sequence `80 82 73 79 * 82 * * 73 * 84 * 89 * * * 81 85 69 * * * 85 * 69` (where a number means insert and an asterisk means remove the maximum) is applied to an initially empty priority queue. Give the sequence of numbers returned by repeated remove the maximum operations.

82 82 80 79 84 89 73 73 85 81 69 85

4. A colleague suggests that instead of using a priority queue to implement finding the maximum, you could instead use a linked list, but keep track of the maximmum value inserted so far. Then, when you want the maximum value, you could simply return it, implementing the find the maximum operation in constant time. Why won't this approach work?

The approach works fine if you only want to the maximum value inserted so far. You will not be able to find the next largest value or the one after and so on in constant time You can additionally keep track of the second next largest, third and so on, but this defeats the purpose. Or, doing searches also defeats the purpose with linear time.

5. For a max-oriented priority queue, suppose that a client calls `insert(...)` with an item that is larger than all items in the queue, and then immediately calls `removeMax(...)`. Is the resulting heap identical to the heap as it was before these operations? Assume that there are no duplicate keys.

Not usually. If the child of each node from the max/root to the inserted key was an only child, then yes, but this would not be a heap. A sufficient condition for a binary heap would be if each said child was larger than its sibling. I'm pretty sure this is a necessary condition as well and thus a logical equivalent.

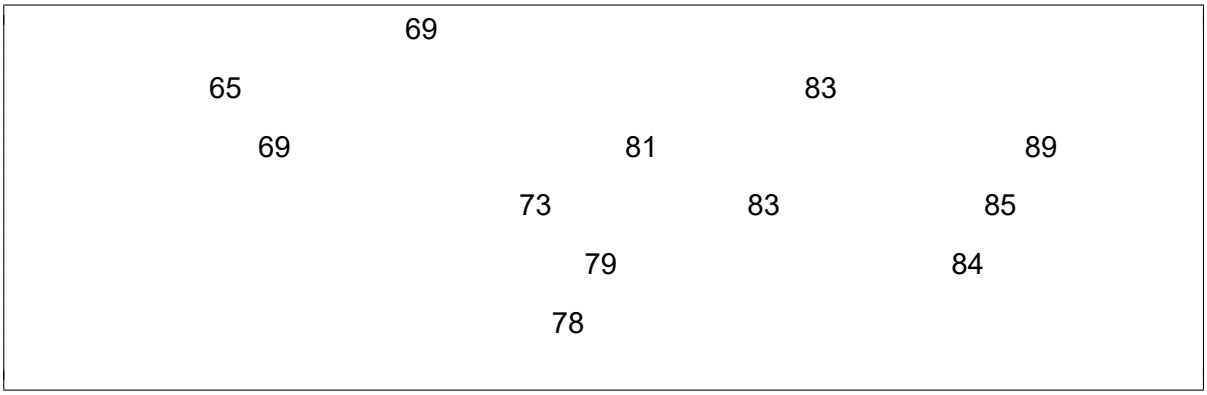
6. Describe an algorithm to find the smallest 1,000 elements in an unordered array of $n$ integers in $O(n)$ time.

Linear time (bottom-up level-wise heapify) build a min-oriented heap. Then extract the root, which is constant time, and reheapify, which is log_2(n-i) time for i from 0 to 999. So, the total time is linear.

If linear time build is off the table, we can build a max-oriented heap of the first 1,000 elements in constant time. Then, we insert (at worst log_2 1000 time) and remove root (at worst log_2 1000 time) n-1,000 times, for a total of at worst (n-1000)*2*log_2(1000), meaning linear time.

## Binary Search Trees

7. Draw the BST that results when you insert the keys `69 65 83 89 81 85 69 83 84 73 79 78`, in that order, into an initially empty tree. When dealing with equal keys, take the left branch.




8. Suppose that a BST has keys that are integers between 1 and 10, and we search for 5. Which sequence(s) below *cannot* be the sequence of keys examined? There may be more than one correct answer.

✓ A. 10, 9, 8, 7, 6, 5

✗ B. 4, 10, 8, 7, 9, 5 B. Since 5 is less than 7, the next comparison should be less than 7, not 9.

✓ C. 1, 10, 2, 9, 3, 8, 4, 7, 6, 5

✗ D. 2, 7, 3, 8, 4, 5 D. This order violates the order in which keys are inserted in a BST: 8 should be right of 7, not right of 3.

✓ E. 1, 2, 10, 4, 8, 5

9. A colleague decides to sort input data before inserting it into a binary search tree. How will this input sequence affect the runtime of the search operation?

This makes the BST a chain (no branching), meaning the at worst every vertex will be traversed, increasing log time to linear time.

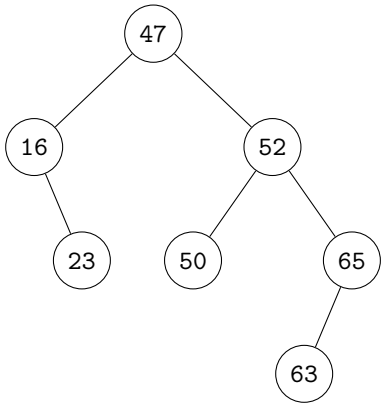
10. Suppose that we have an estimate ahead of time of how often search keys are to be accessed in a BST, and the freedom to insert them in any order that we desire. Should the keys be inserted into the tree in increasing order, decreasing order of likely frequency of access, or some other order? Explain your answer. They should be inserted in an order that minimizes the summation over f_i * w_i where f is the frequency of the i-th key and w_i is the weight of the i-th key, where w_i is the level of the i-th key and the root is the zeroth level. This means the keys with highest frequency should be at the top, and order of insert depends on the value of the key. If the value of the key is the frequency, then they should not be inserted in increasing or decreasing order since that will create a chain which adds 1 to the weight with each insertion; which illustrates that in general, the closer to a complete tree, the better.

11. Using the binary search tree property, devise an algorithm to find the $k$th smallest and $k$th largest value in a binary search tree. Expand the algorithm, using the same strategy, to find the largest $k$ and smallest $k$ values in the binary search tree.

For kth smallest, do inorder traversal and count to k. For kth largest, if we know how many vertices/nodes there are, then do inorder traversal up to n-k. If not, do inorder traversal in reverse (right traverse, enqueue, left traverse) and count to k. To expand, we don't really do anything different, as we are already visiting all k values on the way to the kth.

12. Suppose we have a binary search tree where in addition to `left` and `right` pointers, nodes also have `parent` pointers, upwards in the tree to their parents. Using this property, given pointers to two nodes, devise an algorithm to find the nearest common ancestor of the two nodes. For example, given the following tree, node `50` and `63` would have a nearest common ancestor of `52`. However, nodes `23` and `63`

share no ancestors until the root of the tree. In this case, the root, 47, is their nearest common ancestor. Finally, determine the time complexity of your algorithm.




Do inorder traversal (until the greater of the two) and starting with the lesser of the two count backwards from zero (or whatever) for each level up and forward for each level down (or vice versa) and the vertex/node at which the level count was minimum (max for the vice versa) is nearest common ancestor (or join or supremum). Otherwise, we can count from each of the two to the root. Whichever has the greater count, we can traverse it upward by the difference between the two counts. Then we can traverse both upward at the same time until their key values are equal.