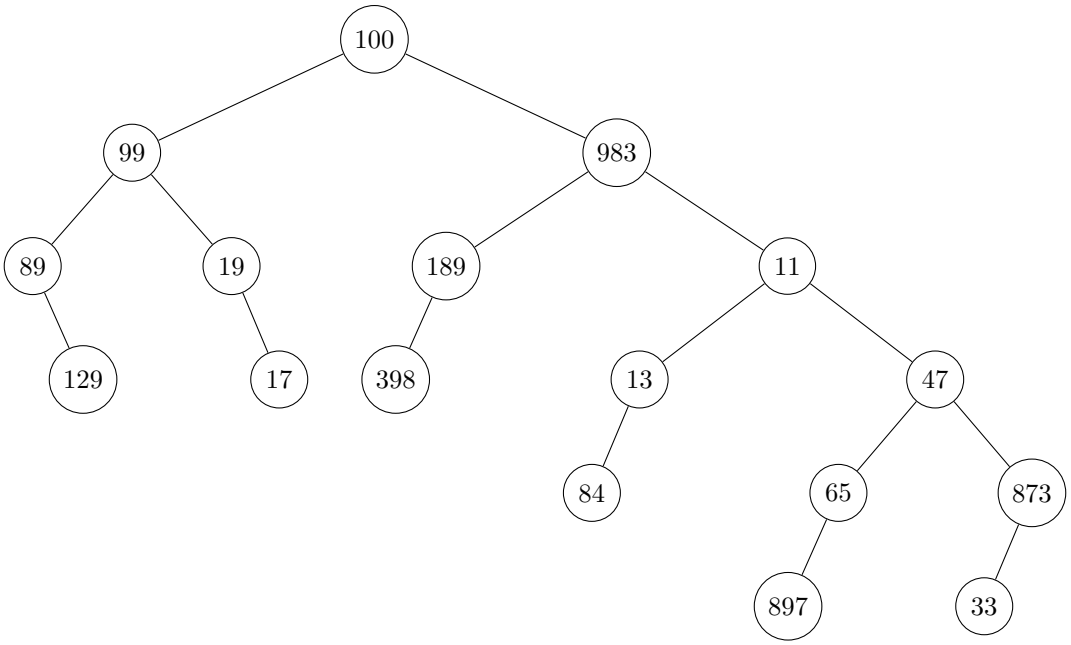


Figure 1: An example binary tree

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? **5**
   (d) What is the result of preorder traversal through the tree?

   Solution: 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?

   Solution: 129, 89, 17, 19, 99, 398, 189, 84, 13, 897, 65, 33, 873, 47, 11, 983, 100

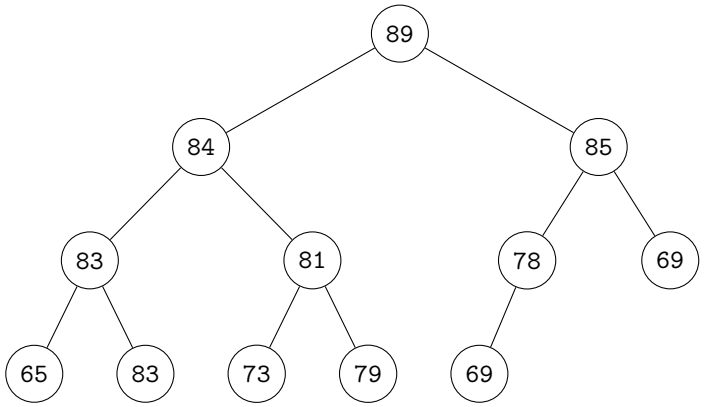
   (f) What is the result of inorder traversal through the tree?

   Solution: 129, 89, 99, 17, 19, 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.

Solution:



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.

Solution: `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?

Solution: This approach won't work because it only keeps track of the maximum value. After it has been retrieved, the data structure has no way to easily find the next maximum value in the remaining set of elements. One would have to do a linear scan through the linked list to find the next maximum element, resulting in an $O(n)$ time complexity (the worst possible time complexity for searching).

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.

Solution: Yes, the resulting heap will be identical to the heap as it was before the `insert(...)` and `removeMax(...)` operations. When a new largest number is inserted, it is inserted at the next open position and follows a path up the binary heap, swapping with elements all the way to the root. This is the same path that the element in the rightmost leaf node will follow down when the largest element is deleted.

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

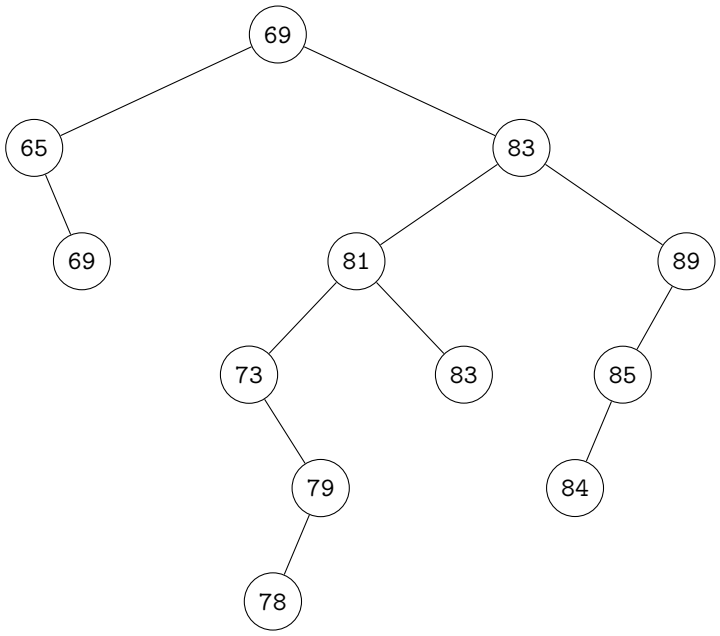
Solution: Create a max-heap with total capacity of 1,000 elements. Iterate through the array of n elements. For each element, if the max-heap is under capacity, simply insert the current element into the max-heap. If the max-heap is at capacity and if the current element is less than the max-heap's root, remove the maximum from the max-heap and insert the current element. Repeat for

all elements. The elements within the heap after the iteration is complete will be the smallest 1,000 elements within the array of n integers.

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.

Solution:



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?

   A. 10, 9, 8, 7, 6, 5

   B. 4, 10, 8, 7, 9, 5

   C. 1, 10, 2, 9, 3, 8, 4, 7, 6, 5

   D. 2, 7, 3, 8, 4, 5

   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?

Solution: Sorted input inserted into a binary search tree is its worst case. The colleague will essentially end up creating a linked list and receive no benefit of the $O(\log_2 n)$ lookup time that the binary search tree can provide.

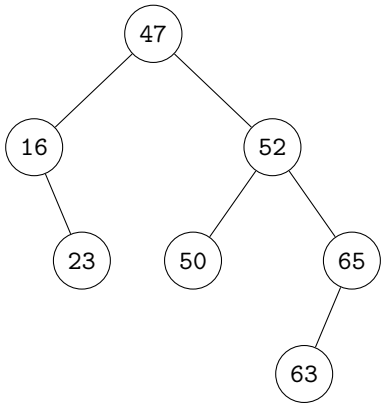
10. Suppose that we have an estimate ahead of time of how often search keys are to be accessed in a BST, and the freedom to insert them in any order that we desire. Should the keys be inserted into the tree in increasing order, decreasing order of likely frequency of access, or some other order? Explain your answer.

Solution: Ideally, we would have keys that are most likely to be accessed as close to the root of the tree as possible. This ensures the fewest number of hops during tree traversal, especially for very large trees. If we know a particular key is going to be accessed very often, we should insert that key into the tree sooner rather than later so as to keep it as close to the root of the tree as possible. Therefore, we should insert the keys in decreasing order of frequency of access. This way, the most frequently accessed keys will remain at the top of the binary search tree.

11. Using the binary search tree property, devise an algorithm to find the kth smallest and kth 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.

Solution: A binary search tree already contains some element of ordering within it. To get the smallest k elements from the binary search tree, we simply need to use in order traversal and only visit a maximum of k nodes. The kth smallest element will be the last element we visit. To get the largest k nodes, we need to do the same but use reverse in order traversal. The kth largest element will be the last element we visit.

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.



Solution: Given pointers to two nodes, `a` and `b`, determine each node's depth by navigating the `parent` links to the root. Once the depth of each node is known, bring the deep node to the same level as the shallow node via the `parent` links. Once the two pointers are at the same level in the tree, move the pointers up the levels of the tree together via the `parent` nodes. If the two pointers ever point to the same node, we have found our nearest common ancestor.

This algorithm walks up the tree a total of three times (once to determine `a`'s level, another time to determine `b`'s level, and then finally walking up with `a` and `b` together). The time complexity of the algorithm, then, would be $O(\log_2 n)$.

Another method is to take advantage of the binary search property. Given two nodes, `a` and `b`, we know that, starting from the root node, the nearest common ancestor will have a value in between `a` and `b`. Starting from the root, keep track of the `current` node we are looking at. If the `current` node is less than `min(a, b)`, update `current` to `current`'s right node. Alternatively, if the `current` node

is greater than `max(a, b)`, update `current` to `current`'s left node. When `current` is in between `a` and `b`, we have found our nearest common ancestor.