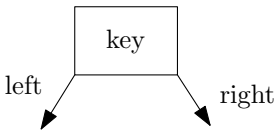
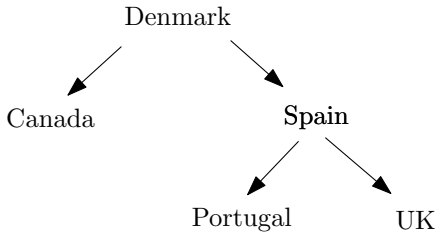
Binary Trees

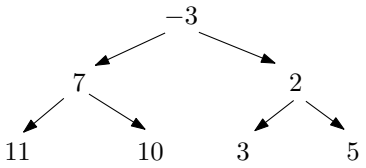
Binary trees are a data structure for storing and manipulating ordered data. There are several possible representations, but we will start by assuming that every binary tree has three fields:



- *key* is the part of the data that we use to order the nodes — every pair of keys must be comparable (total order).
- *left* is a pointer (reference) to the root of the left subtree.
- *right* is a pointer (reference) to the root of the right subtree.

Keys typically have data associated with them (e.g. names with phone numbers), which are stored in each node as additional fields. Many trees also store the parent of each node to allow easy traversal up the tree. The node with no parent is the *root*, and nodes with no children are *leaves*.





One way in which binary trees differ is how the key of a parent relates to the keys of its children. Minimum heaps (priority queues) are binary trees in which the key of a parent is less than (or equal to) the keys of its left and right children. This means that the minimum element is always at the root of the tree and it makes it easy to keep the tree balanced. Being balanced means that there are (roughly) as many nodes in the left subtree as in the right subtree, for every node in the tree. The advantage of being balanced is that the height of the tree is logarithmic in the number of nodes stored in it. Heaps are kept balanced by keeping the tree very close to a complete tree — that is, by having all of the leaves in the last two rows of the tree.

Lemma. *Let $l(v)$ denote the number of nodes in the left subtree of v, and let $r(v)$ denote the number of nodes in the right subtree of v. Suppose that there is some constant $c \geq 1$ such that $l(v) \leq c(r(v)+1)$ and $r(v) \leq c(l(v)+1)$, for every node v in the tree (that is, there are roughly less than c times as many children in the left subtree as the right subtree and vice versa). Then the height of the tree is $O(\log n)$, where n is the total number of nodes in the tree.*

Proof. Let $h(v)$ be the height of the subtree rooted at $h(v)$. Let $n(v)$ be the number of vertices in the subtree rooted at v. We prove inductively on $n(v)$ that $h(v) \leq d \log n(v) + 1$, for some positive constant d that depends on c.

Base case: $n(v) = 1$. Then v has no children and $h(v) = 1$. Now $d \log n(v) + 1 = d \log 1 + 1 = d \times 0 + 1 = 1$.

Inductive case: Suppose that inequality holds for all subtrees with less than $n(v)$ vertices. We now show that it holds for a subtree with $n(v)$ vertices. First we upperbound $l(v)$ and $r(v)$ in terms of $n(v)$ and c. Clearly, $l(v) + r(v) + 1 = n(v)$. So

$$\begin{aligned}
l(v) &\leq c(r(v)+1) \\
l(v)+(r(v)+1) &\leq c(r(v)+1)+(r(v)+1) \\
n(v) &\leq (c+1)(r(v)+1) \\
\frac{n(v)}{c+1} &\leq r(v)+1 \\
\frac{n(v)}{c+1}+l(v) &\leq l(v)+r(v)+1 = n(v) \\
l(v) &\leq n(v)-\frac{n(v)}{c+1} \\
&= \frac{(c+1)n(v)-n(v)}{c+1} = \frac{cn(v)}{c+1}
\end{aligned}$$

This upperbound also holds for $r(v)$ because $l(v)$ and $r(v)$ are symmetric in all of the inequalities that we used. Now $h(v)$ is one plus the maximum of the height of the left subtree and the right subtree. Using our inductive hypothesis

$$\begin{aligned}
h(v) &\leq \max\{d\log l(v)+1, d\log r(v)+1\}+1 \\
&\leq \max\left\{d\log\left(\frac{cn(v)}{c+1}\right)+1, d\log\left(\frac{cn(v)}{c+1}\right)+1\right\}+1 \\
&\leq d\log\left(\frac{cn(v)}{c+1}\right)+1+1 \\
&= d\log n(v)+d\log\left(\frac{c}{c+1}\right)+1+1 \\
&\leq d\log n(v)+1, \text{ if } d\log\left(\frac{c}{c+1}\right)+1 \leq 0
\end{aligned}$$

Recall that $c \geq 1$. Therefore $\frac{c}{c+1} < 1$. Hence $\log\left(\frac{c}{c+1}\right) < 0$. Thus $d = -\frac{1}{\log(c/(c+1))}$ is a positive constant satisfying our constraint, and our induction goes through. □

Although finding the minimum key in a heap is very fast, finding an arbitrary key value in a heap requires looking at most of the nodes. When this kind of random access is required, we construct binary trees with the following important properties:

- No two keys in the tree are equal.
- Every key α in the left subtree of a node N is less than N.key.
- Every key β in the right subtree of a node N is greater than N.key.

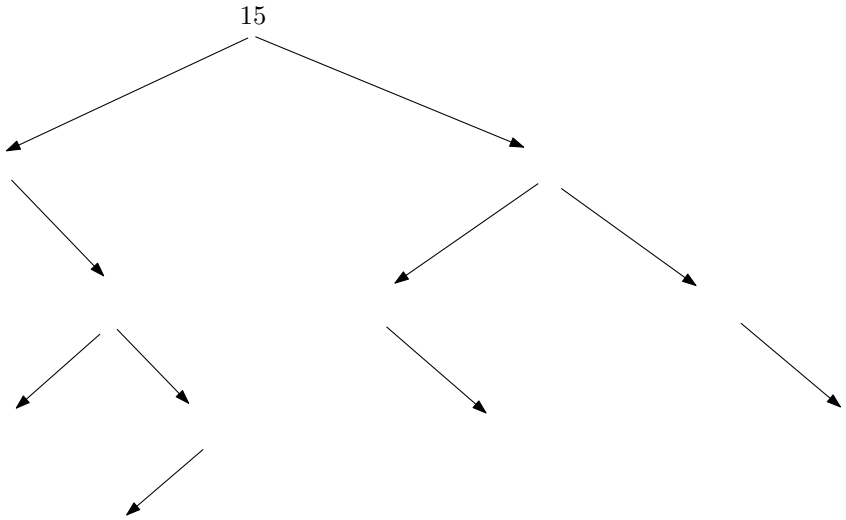
We can construct such binary trees by adding one node at a time using the following algorithm.

```
Algorithm TreeAdd(Tree T, Node N)
   if T = ∅ then return N
   if N.key < T.key then
      if T.left = ∅ then
         T.left ← N
      else
         TreeAdd(T.left, N)
   else if N.key > T.key then
      if T.right = ∅ then
         T.right ← N
      else
         TreeAdd(T.right, N)
   return T
```

We recursively search for the value in the tree and insert the value as a leaf.

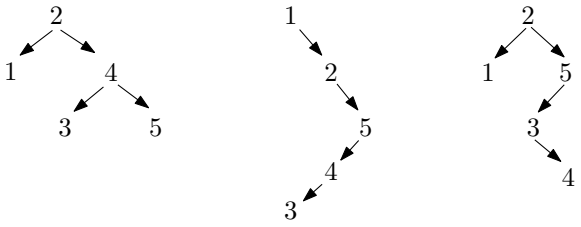
Can every tree be constructed this way?

What is the worst-case height of a tree constructed this way?



Add these values to the tree in their listed order: 4, 23, 17, 19, 8, 11, 25, 9, 24, 5, and 27.

Trees containing the same keys can differ substantially. Some trees are more balanced than others. The reason that we chose the ordering properties that we did was that they permit a simple divide-and-conquer method of locating a key.




```
Algorithm TreeFind(Tree T, Key k)
   if T = ∅ then
      return ∅
   if k < T.key then
      return TreeFind(T.left)
   if k = T.key then
      return T
   if k > T.key then
      return TreeFind(T.right)
```

Every recursive call decends a level in the tree, so the total number of recursive calls is upper-bounded by the height of the tree. The previous lemma guarantees that the search time is logarithmic in the size of the tree, if the tree is balanced.

What is the average-case search in a randomly constructed tree?

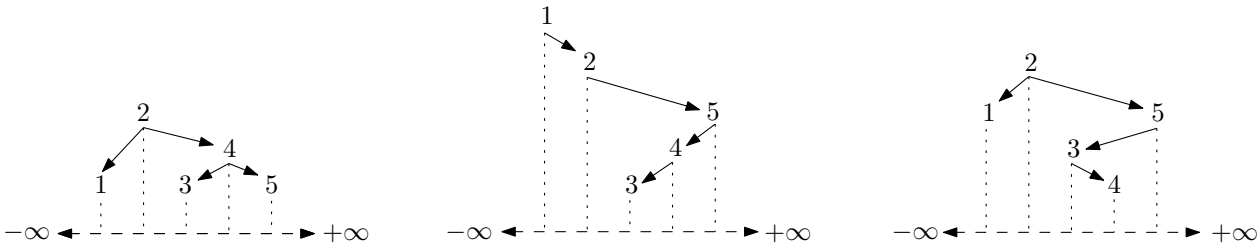
Lemma. *Consider a tree constructed by inserting keys. Assume that the order of key insertion is uniformly random (i.e. every permutation of the insertion order is equally likely). Then the expected query time is* $O(\log n)$*, where* n *is the tree size.*

Proof. We use a technique called backwards analysis. To use it, we first take a query key. Then we build the binary tree incrementally, and after each insertion, we track where the query key was found or where we dropped off the tree looking for it. Searching for 3.5 in the trees above would drop off of at 3, 3, and 4 (from left to right). Searching for 2.5 would drop off at 3 in each of the trees.

Clearly, inserting one more node in the binary tree only increases the number of recursive calls in the search algorithm because new nodes are inserted as leaves. Moreover, we only add one node, which results in at most one more recursive call (i.e. tree height increases by at most one). Let X_i be a 0/1 random variable: X_i is 1 if the number of recursive calls increases after the ith insertion and 0 otherwise. Then the number of recursive calls after n insertions is $\sum_{i=1}^{n} X_i$ and the average-case runtime is

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E\left[X_i\right]$$

by the linearity of expectation. To calculate the expectation of X_i, we must understand when the number of recursive calls increases. This is best seen by redrawing the above trees as shown below.



We now view each node as splitting the real number line. The particular structure of the tree does not affect the partition of the real number line.

- If the query key is contained in the tree after the ith insertion, the search stops at the node containing the key. So if $X_i = 1$, we need the ith insertion to be equal to the query key. Each of the i keys was equally like to be the ith insertion (backwards analysis), so the probability of this happening is $1/i$.
- If the query key is not contained in tree after the ith insertion, the search falls off at a node bounding the interval containing the query point. If $X_i = 1$, one of the bounding keys was inserted on the ith step. There are at most two keys bounding any interval, so the probability of this happening is at most $\frac{2}{i}$ because each of the i keys was equally likely to be the ith insertion (backwards analysis).

Therefore,

$$\sum_{i=1}^{n} E\left[X_i\right] \leq \sum_{i=1}^{n} \frac{2}{i} = 2 \sum_{i=1}^{n} \frac{1}{i}.$$

The harmonic series $\sum_{i=1}^{n} (1/i) \in \Theta(\log n)$ because $\log n = \int_1^x (1/y)\ dy$. □

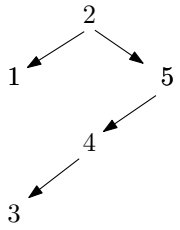
Note that the condition that every insertion order is equally likely is not the same as saying every tree is equally likely because different insertion orders can produce the same tree (e.g. $2, 1, 3$ and $2, 3, 1$). Also note that this result holds for queries that are in the tree. The run-time of such queries is asymptotically the depth of the node containing the query key. So the expected level of any given node is $\Theta(\log n)$.

Does this result tell us about the height of a random binary tree? Unfortunately no, because the height is determined by the worst-case query, which is biased through maximization (i.e. it ratchets up on the variance of the expectation).

Exercise: Write a `TreeMax` routine that returns the largest key in a tree.

We now show how to find the predecessor key of a node in a binary tree, which we will use to remove nodes shortly.

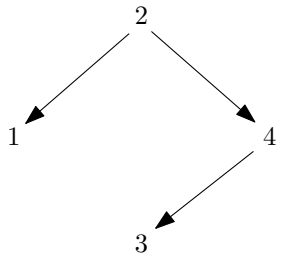
```
Algorithm TreePred(Tree T, Key k)
   v ←Find(T, k)
   if v = ∅ then
      return ∅
   if v.left≠ ∅ then
      return TreeMax(v.left)
   while (v.parent ≠ ∅ and v ≠ v.parent.right)
      v ← v.parent
   return v.parent
```



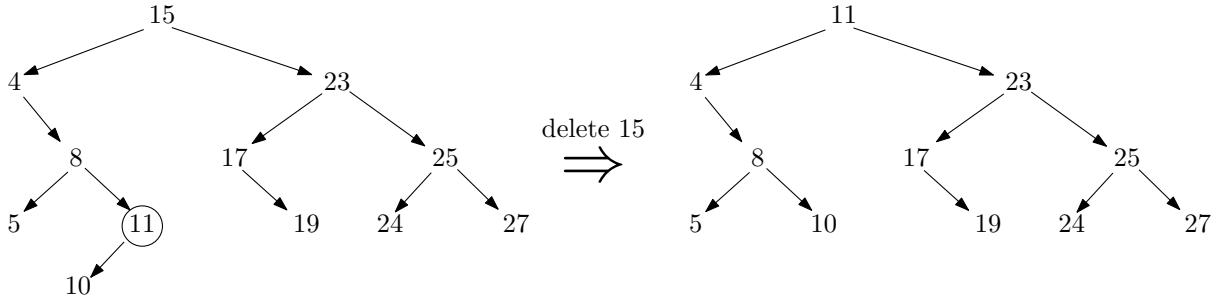
What is the asymptotic runtime of `TreePred`? It is dominated by `TreeFind`, `TreeMax`, and the `while` loop. All three of these take $O(h)$ time, where h is the height of the binary tree.

We now turn to deleting a node. In order to remove a node we have to find a place for its children. Consider the following cases:

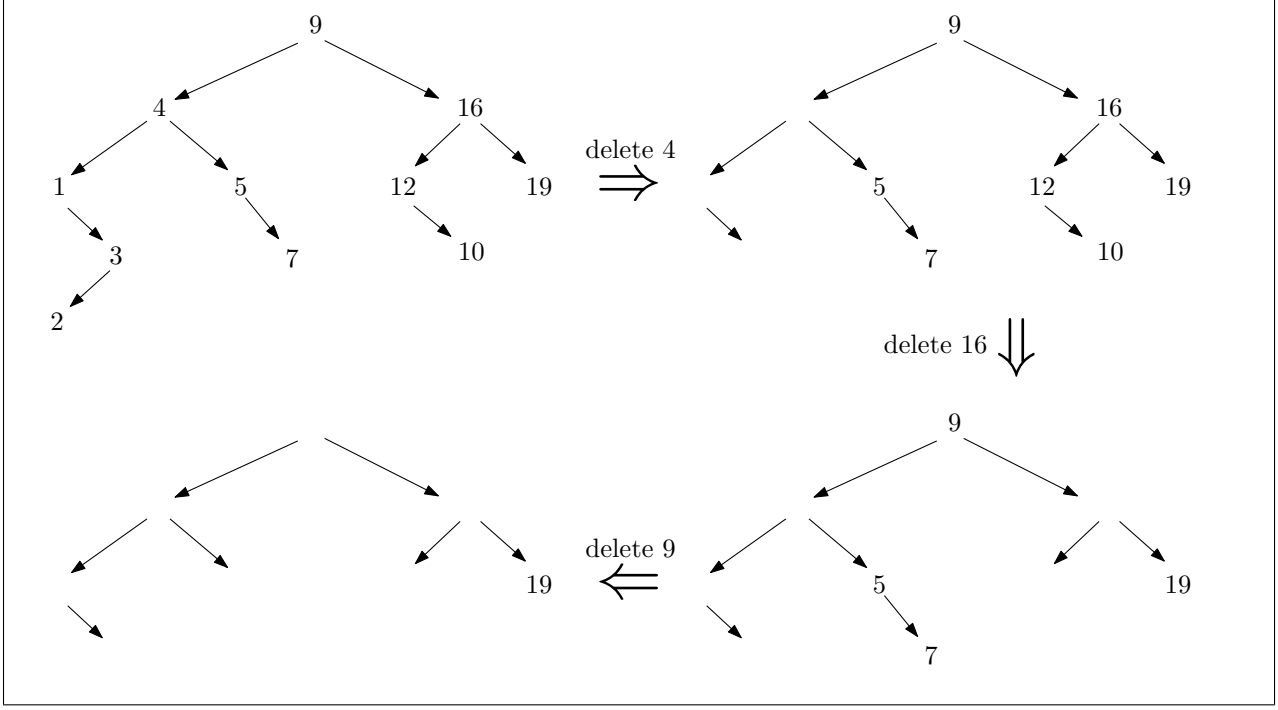
(1) If the node to delete has no children (3), delete the node.
(2) If the node to delete has one child (4), delete the node and put the child in its parent's place.
(3) If the node to delete has two children (2), removing the node leaves two dangling subtrees. A straightforward approach is to temporarily remove one of the child subtrees T. Then we can apply the previous case. To reattach T, we could iteratively add each node in T. However, the worst case of this approach is $|T| \times O(h)$ where, h is the height of the tree.



We can resolve case (3) in a better manner: replace the node n that we want to delete with its predecessor l. Recall that l is maximum element in n's left subtree. The ordering property of the subtree rooted at n tree is that $\alpha < n < \beta$, where α is any node in the left subtree of n and β is any node in the right subtree of n. The predecessor l of n has the property that $\alpha \leq l$ and $l < n < \beta$. So removing l and then changing n to l preserves the order. But if l has two children, we have only shifted case (3) elsewhere! Fortunately, l has no right child because l is the maximum element in the left subtree of n.



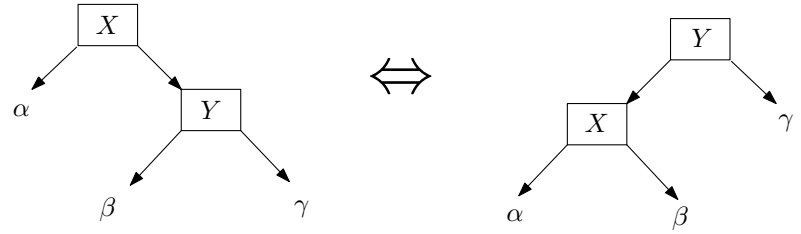
Practice:



What is the asymptotic complexity of this sophisticated deletion routine? We always do $O(h)$ to locate the node to delete.

- Case 1 and 2 use a small local fix that takes $\Theta(1)$ time.
- Case 3 takes $O(h)$ time to find the predecessor. Deleting the predecessor is $\Theta(1)$ time because we have already located it. Swapping the predecessor takes $\Theta(1)$ time. So the total is $O(h)$ time.

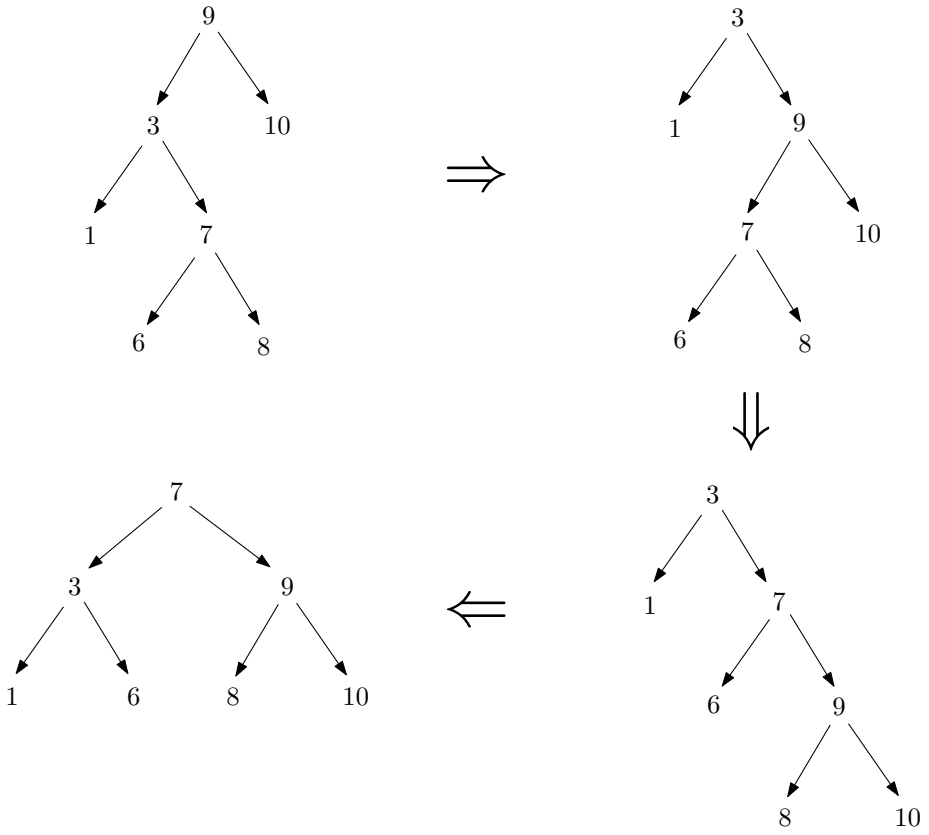
Regardless the case, the total runtime is $O(h)$. Notice that `TreeFind`, `TreeMax`, `TreePred`, and `TreeAdd` also take $O(h)$ time. Hence, good performance requires keeping the height low. This can be done by keeping the tree balanced. Two common schemes for doing this are AVL trees and red-black trees. They both use a rotation operation to rebalance the tree.



$\alpha < X < \boxed{\beta < Y < \gamma}$ $\qquad$ $\boxed{\alpha < X < \beta} < Y < \gamma$

The rotation operation has three very important properties.

(1) It preserves the ordering of subtrees with respect to their roots.
(2) It transfers one level between subtrees: it decreases the height of the right subtree by one and increases the height of the left subtree by one (or vice versa).
(3) It is local, so it only takes constant time.



Both AVL and red-black trees are fast but complicated and not obviously correct (lots of case analysis). So we study B-trees instead.