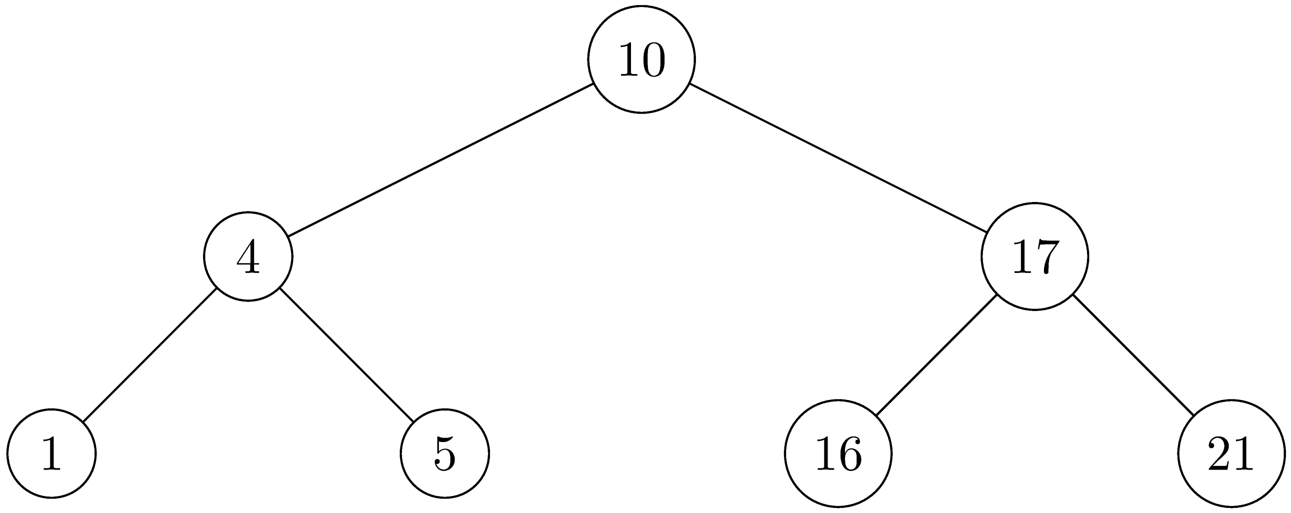
12 Binary Search Trees

12.1 What is a binary search tree?

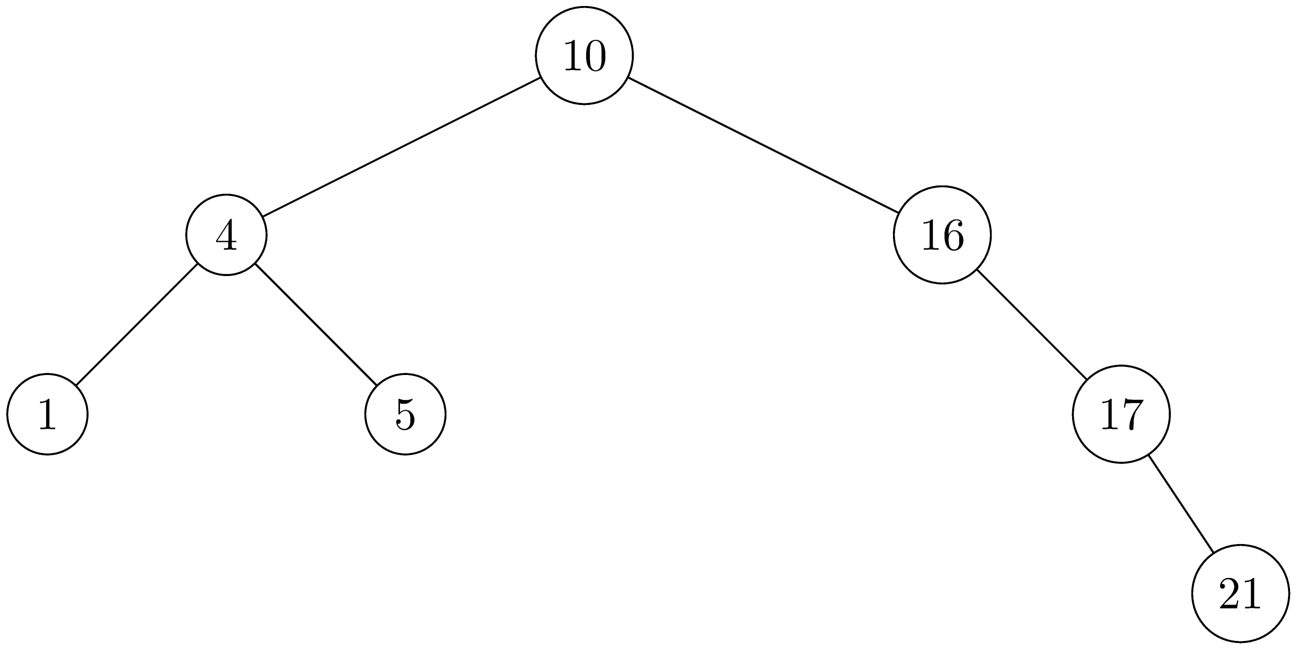
12.1-1

> For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

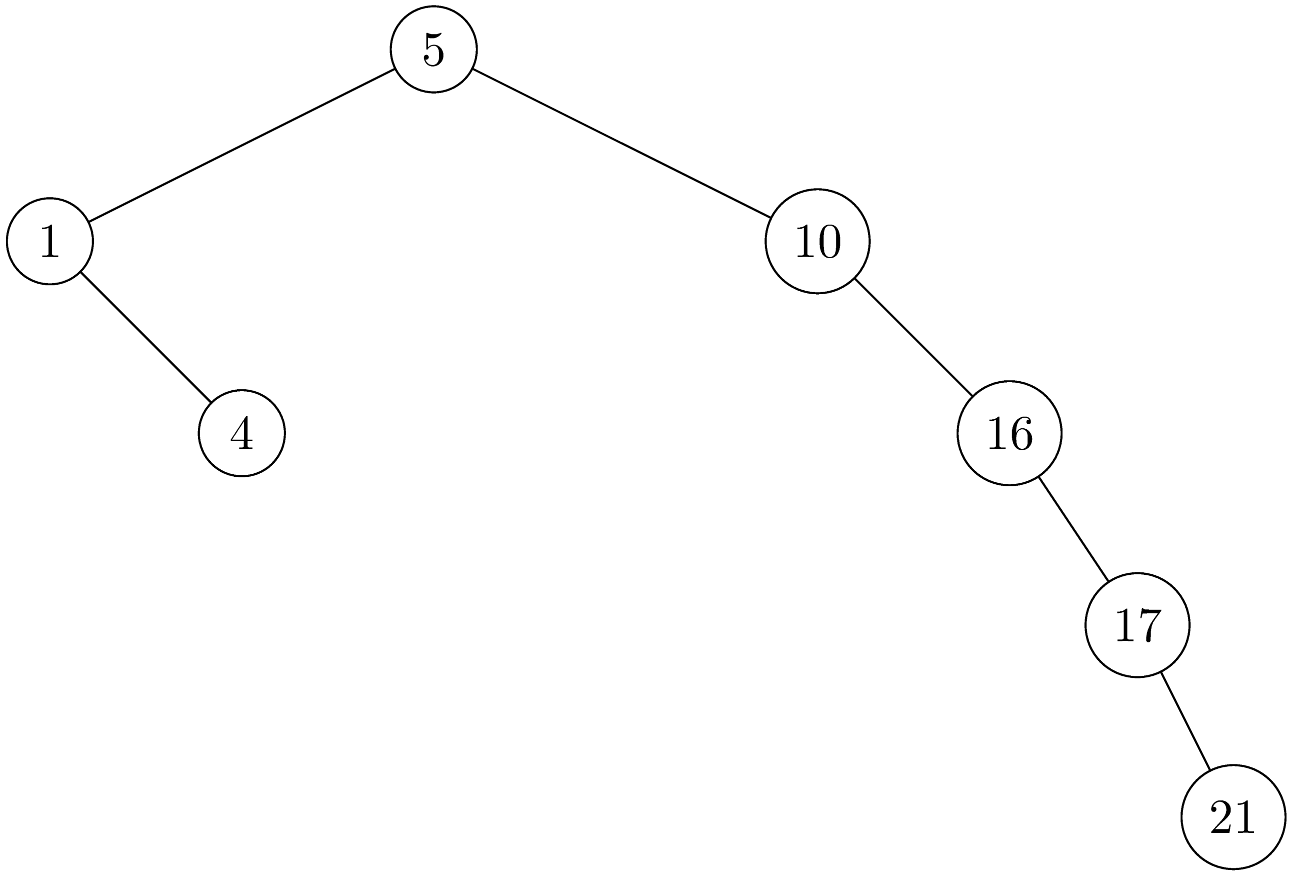
- height = 2:



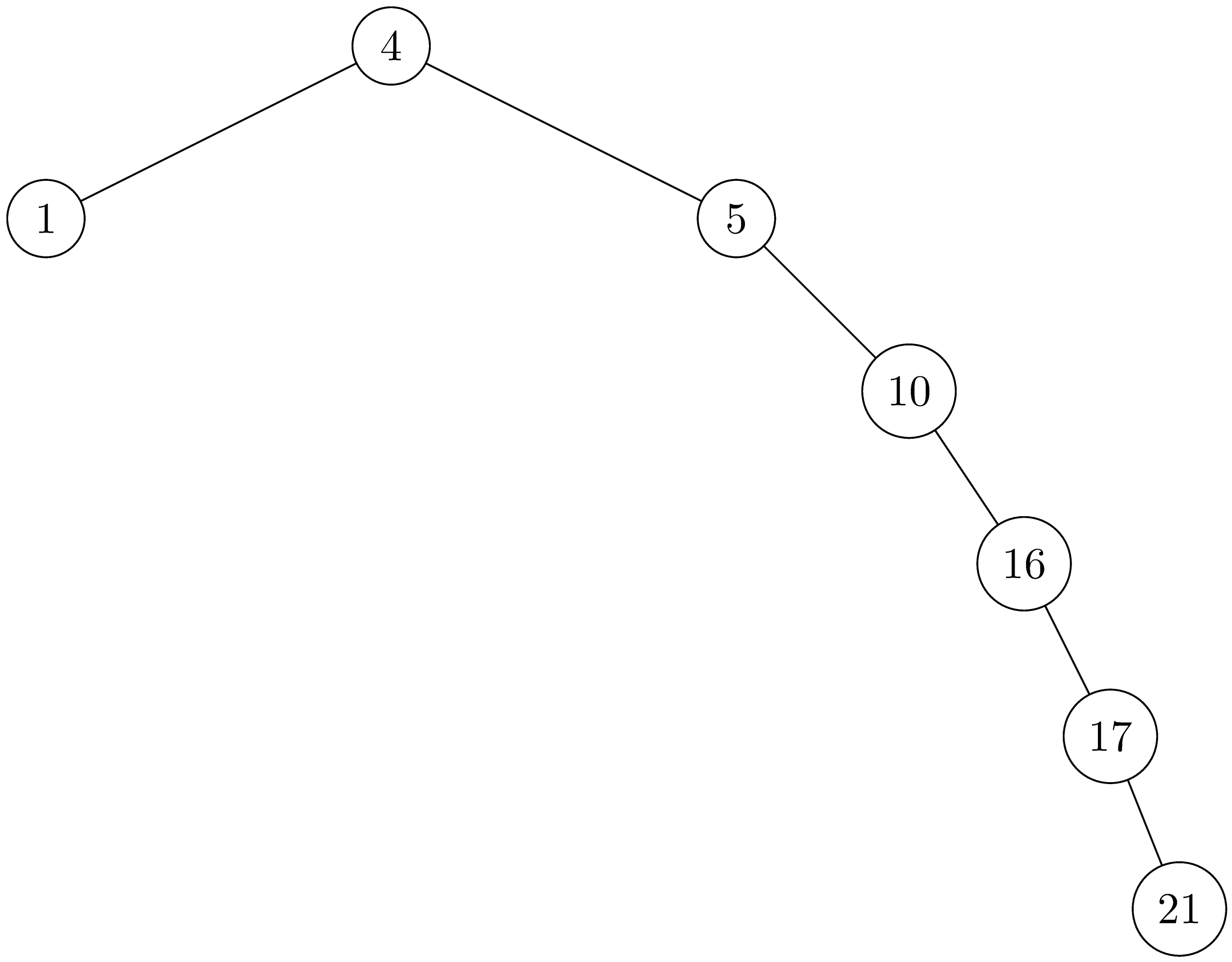
- height = 3:



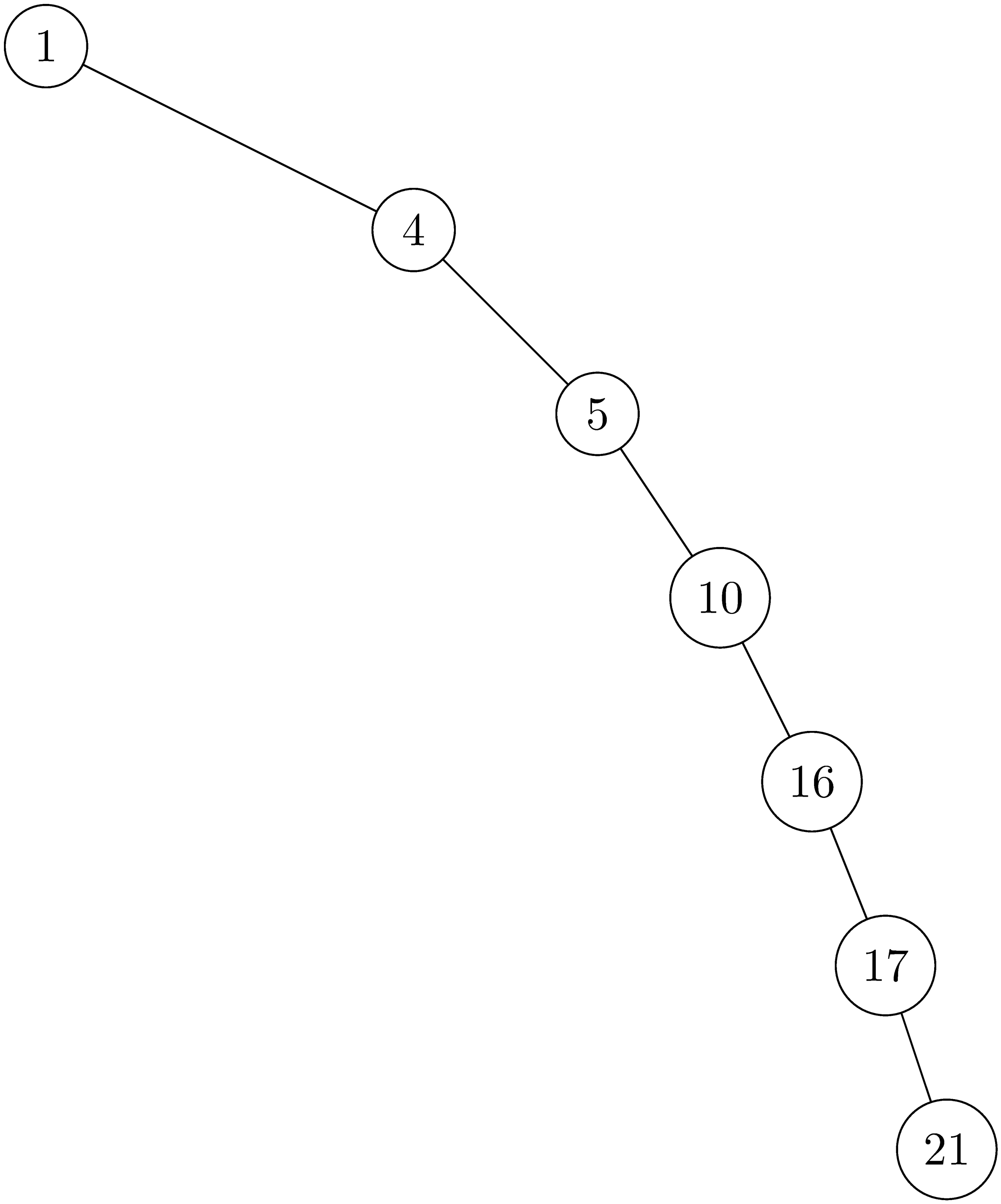
- height = 4:



- height = 5:



- height = 6:



12.1-2

> What is the difference between the binary-search-tree property and the min-heap property (see page 153)? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

- The binary-search-tree property guarantees that all nodes in the left subtree are smaller, and all nodes in the right subtree are larger.
- The min-heap property only guarantees the general child-larger-than-parent relation, but doesn't distinguish between left and right children. For this reason, the min-heap property can't be used to print out the keys in sorted order in linear time because we have no way of knowing which subtree contains the next smallest element.

12.1-3

> Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality.)

```
INORDER-TREE-WALK(T)
    let S be an empty stack
    current = T.root
    done = 0
    while !done
        if current != NIL
            PUSH(S, current)
            current = current.left
        else
            if !S.EMPTY()
                current = POP(S)
                print current
                current = current.right
            else done = 1
```

12.1-4

> Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

```
PREORDER-TREE-WALK(x)
    if x != NIL
        print x.key
        PREORDER-TREE-WALK(x.left)
        PREORDER-TREE-WALK(x.right)
```

```
POSTORDER-TREE-WALK(x)
    if x != NIL
        POSTORDER-TREE-WALK(x.left)
        POSTORDER-TREE-WALK(x.right)
        print x.key
```

12.1-5

> Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

Assume, for the sake of contradiction, that we can construct the binary search tree by comparison-based algorithm using less than $\Omega(n \lg n)$ time, since the inorder tree walk is $\Theta(n)$, then we can get the sorted elements in less than $\Omega(n \lg n)$ time, which contradicts the fact that sorting n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

12.2-1

> Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?
>
> **a.** $2, 252, 401, 398, 330, 344, 397, 363$.
>
> **b.** $924, 220, 911, 244, 898, 258, 362, 363$.
>
> **c.** $925, 202, 911, 240, 912, 245, 363$.
>
> **d.** $2, 399, 387, 219, 266, 382, 381, 278, 363$.
>
> **e.** $935, 278, 347, 621, 299, 392, 358, 363$.

- **c.** could not be the sequence of nodes explored because we take the left child from the 911 node, and yet somehow manage to get to the 912 node which cannot belong the left subtree of 911 because it is greater.
- **e.** is also impossible because we take the right subtree on the 347 node and yet later come across the 299 node.

12.2-2

> Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

```
TREE-MINIMUM(x)
    if x.left != NIL
        return TREE-MINIMUM(x.left)
    else return x
```

```
TREE-MAXIMUM(x)
    if x.right != NIL
        return TREE-MAXIMUM(x.right)
    else return x
```

12.2-3

> Write the TREE-PREDECESSOR procedure.

```
TREE-PREDECESSOR(x)
    if x.left != NIL
        return TREE-MAXIMUM(x.left)
    y = x.p
    while y != NIL and x == y.left
        x = y
        y = y.p
    return y
```

12.2-4

> Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

Search for 9 in this tree. Then $A = \{7\}$, $B = \{5, 8, 9\}$ and $C = \{\}$. So, since $7 > 5$ it breaks professor's claim.

12.2-5

> Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Suppose the node x has two children. Then it's successor is the minimum element of the BST rooted at $x.right$. If it had a left child then it wouldn't be the minimum element. So, it must not have a left child. Similarly, the predecessor must be the maximum element of the left subtree, so cannot have a right child.

12.2-6

> Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor

> of x. (Recall that every node is its own ancestor.)

First we establish that y must be an ancestor of x. If y weren't an ancestor of x, then let z denote the first common ancestor of x and y. By the binary-search-tree property, $x < z < y$, so y cannot be the successor of x.

Next observe that $y.left$ must be an ancestor of x because if it weren't, then $y.\text{right}$ would be an ancestor of x, implying that $x > y$. Finally, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z denote this lowest ancestor. Then z must be in the left subtree of y, which implies $z < y$, contradicting the fact that y is the successor of x.

12.2-7

> An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

To show this bound on the runtime, we will show that using this procedure, we traverse each edge twice. This will suffice because the number of edges in a tree is one less than the number of vertices.

Consider a vertex of a BST, say x. Then, we have that the edge between $x.p$ and x gets used when successor is called on $x.p$ and gets used again when it is called on the largest element in the subtree rooted at x. Since these are the only two times that that edge can be used, apart from the initial finding of tree minimum. We have that the runtime is $O(n)$. We trivially get the runtime is $\Omega(n)$ because that is the size of the output.

12.2-8

> Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

Suppose x is the starting node and y is the ending node. The distance between x and y is at most $2h$, and all the edges connecting the k nodes are visited twice, therefore it takes $O(k + h)$ time.

12.2-9

> Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.\text{key}$ is either the smallest key in T larger than $x.\text{key}$ or the largest key in T smaller than $x.\text{key}$.

- If $x = y.left$, then calling successor on x will result in no iterations of the while loop, and so will return y.
- If $x = y.\text{right}$, the while loop for calling predecessor (see exercise 3) will be run no times, and so y will be returned.

12.3 Insertion and deletion

12.3-1

> Give a recursive version of the TREE-INSERT procedure.

```
RECURSIVE-TREE-INSERT(T, z)
    if T.root == NIL
        T.root = z
    else INSERT(NIL, T.root, z)
```

```
INSERT(p, x, z)
    if x == NIL
        z.p = p
        if z.key < p.key
            p.left = z
        else p.right = z
    else if z.key < x.key
        INSERT(x, x.left, z)
    else INSERT(x, x.right, z)
```

12.3-2

> Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

Number of nodes examined while searching also includes the node which is searched for, which isn't the case when we inserted it.

12.3-3

> We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

- The worst-case is that the tree formed has height n because we were inserting them in already sorted order. This will result in a runtime of $\Theta(n^2)$.
- The best-case is that the tree formed is approximately balanced. This will mean that the height doesn't exceed $O(\lg n)$. Note that it can't have a smaller height, because a complete binary tree of height h only has $\Theta(2^h)$ elements. This will result in a rutime of $O(n \lg n)$. We showed $\Omega(n \lg n)$ in exercise 12.1-5.

12.3-4

> Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

No, giving the following courterexample.

- Delete A first, then delete B:

```
    A         C         C
  / \       / \         \
 B   D     B   D         D
    /
    C
```

- Delete B first, then delete A:

```
    A         A         D
  / \         \       /
 B   D         D     C
    /         /
    C         C
```

12.3-5

> Suppose that instead of each node x keeping the attribute $x.p$, pointing to x's parent, it keeps $x.succ$, pointing to x's successor. Give pseudocode for SEARCH, INSERT, and DELETE on a binary search tree T using this representation. These procedures should operate in time $O(h)$, where h is the height of the tree T. ($\textit{Hint:}$ You may wish to implement a subroutine that returns the parent of a node.)

We don't need to change SEARCH.

We have to implement PARENT, which facilitates us a lot.

```
PARENT(T, x)
    if x == T.root
        return NIL
    y = TREE-MAXIMUM(x).succ
    if y == NIL
        y = T.root
    else
        if y.left == x
            return y
        y = y.left
    while y.right != x
        y = y.right
    return y
```

```
INSERT(T, z)
    y = NIL
    x = T.root
    pred = NIL
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else
            pred = x
            x = x.right
    if y == NIL
        T.root = z
        z.succ = NIL
    else if z.key < y.key
        y.left = z
        z.succ = y
        if pred != NIL
            pred.succ = z
    else
        y.right = z
        z.succ = y.succ
        y.succ = z
```

We modify TRANSPLANT a bit since we no longer have to keep the pointer of p.

```
TRANSPLANT(T, u, v)
    p = PARENT(T, u)
    if p == NIL
        T.root = v
    else if u == p.left
        p.left = v
    else
        p.right = v
```

Also, we have to implement TREE-PREDECESSOR, which helps us easily find the predecessor in line 2 of DELETE.

```
TREE-PREDECESSOR(T, x)
    if x.left != NIL
        return TREE-MAXIMUM(x.left)
    y = T.root
    pred = NIL
    while y != NIL
        if y.key == x.key
```

```
                break
            if y.key < x.key
                pred = y
                y = y.right
            else
                y = y.left
        return pred
```

```
DELETE(T, z)
    pred = TREE-PREDECESSOR(T, z)
    pred.succ = z.succ
    if z.left == NIL
        TRANSPLANT(T, z, z.right)
    else if z.right == NIL
        TRANSPLANT(T, z, z.left)
    else
        y = TREE-MIMIMUM(z.right)
        if PARENT(T, y) != z
            TRANSPLANT(T, y, y.right)
            y.right = z.right
        TRANSPLANT(T, z, y)
        y.left = z.left
```

Therefore, all these five algorithms are still $O(h)$ despite the increase in the hidden constant factor.

12.3-6

> When node z in TREE-DELETE has two children, we could choose node y as its predecessor rather than its successor. What other changes to TREE-DELETE would be necessary if we did so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

Update line 5 so that y is set equal to $\text{TREE-MAXIMUM}(z.left)$ and lines 6-12 so that every $y.left$ and $z.left$ is replaced with $y.\text{right}$ and $z.\text{right}$ and vice versa.

To implement the fair strategy, we could randomly decide each time TREE-DELETE is called whether or not to use the predecessor or successor.

12.4 Randomly built binary search trees

12.4-1

> Prove equation (12.3).

Consider all the possible positions of the largest element of the subset of $n+3$ of size 4. Suppose it were in position $i+4$ for some $i \le n-1$. Then, we have that there are $i+3$ positions from which we can select the remaining three elements of the subset. Since every subset with different largest element is different, we get the total by just adding them all up (inclusion exclusion principle).

12.4-2

> Describe a binary search tree on n nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$. Give an asymptotic upper bound on the height of an n-node binary search tree in which the average depth of a node is $\Theta(\lg n)$.

To keep the average depth low but maximize height, the desired tree will be a complete binary search tree, but with a chain of length $c(n)$ hanging down from one of the leaf nodes. Let $k = \lg(n - c(n))$ be the height of the complete binary search tree. Then the average height is approximately given by

$$\frac{1}{n}\left[\sum_{i=1}^{n-c(n)} \lg i + (k+1) + (k+2) + \cdots + (k + c(n))\right] \approx \lg(n - c(n)) + \frac{c(n)^2}{2n}.$$

The upper bound is given by the largest $c(n)$ such that $\lg(n - c(n)) + \frac{c(n)^2}{2n} = \Theta(\lg n)$ and $c(n) = \omega(\lg n)$. One function which works is $\sqrt{n}$.

12.4-3

> Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (Hint: List the possibilities when $n = 3$.)

Suppose we have the elements $\{1, 2, 3\}$. Then, if we construct a tree by a random ordering, then, we get trees which appear with probabilities some multiple of $\frac{1}{6}$. However, if we consider all the valid binary search trees on the key set of $\{1, 2, 3\}$. Then, we will have only five different possibilities. So, each will occur with probability $\frac{1}{5}$, which is a different probability distribution.

12.4-4

> Show that the function $f(x) = 2^x$ is convex.

The second derivative is $2^x \ln^2 2$ which is always positive, so the function is convex

12.4-5 ⋆

> Consider $\text{RANDOMIZED-QUICKSORT}$ operating on a sequence of n distinct input numbers. Prove that for any constant $k > 0$, all but $O(1/n^k)$ of the $n!$ input permutations yield an $O(n \lg n)$ running time.

Let $A(n)$ denote the probability that when quicksorting a list of length n, some pivot is selected to not be in the middle $n^{1-k/2}$ of the numberes. This doesn't happen with probability $\frac{1}{n^{k/2}}$. Then, we have that the two subproblems are of size n_1, n_2 with $n_1 + n_2 = n - 1$, then

$$A(n) \leq \frac{1}{n^{k/2}} + T(n_1) + T(n_2).$$

Since we bounded the depth by $O(1/\lg n)$ let $\{a_{i,j}\}_i$ be all the subproblem sizes left at depth j,

$$A(n) \leq \frac{1}{n^{k/2}} \sum_j \sum_i \frac{1}{a}.$$

Problem 12-1 Binary search trees with equal keys

> Equal keys pose a problem for the implementation of binary search trees.
>
> **a.** What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?
>
> We propose to improve TREE-INSERT by testing before line 5 to determine whether $z.key = x.key$ and by testing before line 11 to determine whether $z.key = y.key$.
>
> If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)
>
> **b.** Keep a boolean flag $x.b$ at node x, and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time we visit x while inserting a node with the same key as x.
>
> **c.** Keep a list of nodes with equal keys at x, and insert z into the list.
>
> **d.** Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

a. Each insertion will add the element to the right of the rightmost leaf because the inequality on line 11 will always evaluate to false. This will result in the runtime being $\sum_{i=1}^{n} i \in \Theta(n^2)$.

b. This strategy will result in each of the two children subtrees having a difference in size at most one. This means that the height will be $\Theta(\lg n)$. So, the total runtime will be $\sum_{i=1}^{n} \lg n \in \Theta(n \lg n)$.

c. This will only take linear time since the tree itself will be height 0, and a single insertion into a list can be done in constant time.

d.

- **Worst-case:** every random choice is to the right (or all to the left) this will result in the same behavior as in the first part of this problem, $\Theta(n^2)$.

- **Expected running time:** notice that when randomly choosing, we will pick left roughly half the time, so, the tree will be roughly balanced, so, we have that the depth is roughly $\lg(n)$, $\Theta(n \lg n)$.

Problem 12-2 Radix trees

> Given two strings $a = a_0 a_1 \dots a_p$ and $b = b_0 b_1 \dots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either
>
> 1. there exists an integer j, where $0 \le j \le \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \dots j - 1$ and $a_j < b_j$, or
> 2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \dots, p$.
>
> For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.
>
> The ***radix tree*** data structure shown in Figure 12.5 stores the bit strings $1011, 10, 011, 100$, and 0. When searching for a key $a = a_0 a_1 \dots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence $0, 011, 10, 100, 1011$.

(Removed)

Problem 12-3 Average node depth in a randomly built binary search tree

> In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. Although this result is weaker than that of Theorem 12.4, the technique we shall use reveals a surprising similarity between the building of a binary search tree and the execution of $\text{RANDOMIZED-QUICKSORT}$ from Section 7.3.
>
> We define the ***total path length*** $P(T)$ of a binary tree T as the sum, over all nodes x in T, of the depth of node x, which we denote by $d(x, T)$.
>
> **a.** Argue that the average depth of a node in T is
>
> $$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$
>
> Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.
>
> **b.** Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then
>
> $$P(T) = P(T_L) + P(T_R) + n - 1.$$

> **c.** Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that
>
> $$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1).$$
>
> **d.** Show how to rewrite $P(n)$ as
>
> $$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n).$$
>
> **e.** Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$. At each recursive invocation of quicksort, we choose a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.
>
> **f.** Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

(Removed)

Problem 12-4 Number of different binary trees

> Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.
>
> **a.** Show that $b_0 = 1$ and that, for $n \ge 1$,
>
> $$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$
>
> **b.** Referring to Problem 4-4 for the definition of a generating function, let $B(x)$ be the generating function
>
> $$B(x) = \sum_{n=0}^{\infty} b_n x^n.$$
>
> Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is
>
> $$B(x) = \frac{1}{2x}(1 - \sqrt{1-4x}).$$
>
> The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k,$$

where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

c. Show that

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial expansion (C.4) to nonintegral exponents n, where for any real number n and for any integer k, we interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \ge 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n)).$$

a. A root with two subtree.

b.

$$\begin{aligned} B(x)^2 &= (b_0x^0 + b_1x^1 + b_2x^2 + \cdots)^2 \\ &= b_0^2x^0 + (b_0b_1 + b_1b_0)x^1 + (b_0b_2 + b_1b_1 + b_2b_0)x^2 + \cdots \\ &= \sum_{k=0}^{0} b_kb_{0-k}x^0 + \sum_{k=0}^{1} b_kb_{1-k}x^1 + \sum_{k=0}^{2} b_kb_{2-k}x^2 + \cdots \end{aligned}$$

$$\begin{aligned} xB(x)^2 + 1 &= 1 + \sum_{k=0}^{0} b_kb_{1-1-k}x^1 + \sum_{k=0}^{2} b_kb_{2-1-k}x^3 + \sum_{k=0}^{2} b_kb_{3-1-k}x^2 + \cdots \\ &= 1 + b_1x^1 + b_2x^2 + b_3x^3 + \cdots \\ &= b_0x^0 + b_1x^1 + b_2x^2 + b_3x^3 + \cdots \\ &= \sum_{n=0}^{\infty} b_nx^n \\ &= B(x). \end{aligned}$$

$$\begin{aligned} xB(x)^2 + 1 &= x \cdot \frac{1}{4x^2}(1 + 1 - 4x - 2\sqrt{1-4x}) + 1 \\ &= \frac{1}{4x}(2 - 2\sqrt{1-4x}) - 1 + 1 \\ &= \frac{1}{2x}(1 - \sqrt{1-4x}) \\ &= B(x). \end{aligned}$$

c. Let $f(x) = \sqrt{1-4x}$, the numerator of the derivative is

$$
\begin{aligned}
2 \cdot (1 \cdot 2) \cdot (3 \cdot 2) \cdot (5 \cdot 2) \cdots &= 2^k \cdot \prod_{i=0}^{k-2} (2k+1) \\
&= 2^k \cdot \frac{(2(k-1))!}{2^{k-1}(k-1)!} \\
&= \frac{2(2(k-1))!}{(k-1)!}.
\end{aligned}
$$

$$
f(x) = 1 - 2x - 2x^2 - 4x^3 - 10x^4 - 28x^5 - \cdots.
$$

The coefficient is $\frac{2(2(k-1))!}{k!(k-1)!}$.

$$
\begin{aligned}
B(x) &= \frac{1}{2x}(1 - f(x)) \\
&= 1 + x + 2x^2 + 5x^3 + 14x^4 + \cdots \\
&= \sum_{n=0}^{\infty} \frac{(2n)!}{(n+1)!n!} x \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1} \frac{(2n)!}{n!n!} x \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} x.
\end{aligned}
$$

$$
b_n = \frac{1}{n+1} \binom{2n}{n}.
$$

d.

$$
\begin{aligned}
b_n &= \frac{1}{n+1} \frac{(2n)!}{n!n!} \\
&\approx \frac{1}{n+1} \frac{\sqrt{4\pi n}(2n/e)^{2n}}{2\pi n(n/e)^{2n}} \\
&= \frac{1}{n+1} \frac{4^n}{\sqrt{\pi n}} \\
&= (\frac{1}{n} + (\frac{1}{n+1} - \frac{1}{n})) \frac{4^n}{\sqrt{\pi n}} \\
&= (\frac{1}{n} - \frac{1}{n^2+n}) \frac{4^n}{\sqrt{\pi n}} \\
&= \frac{1}{n}(1 - \frac{1}{n+1}) \frac{4^n}{\sqrt{\pi n}} \\
&= \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n)).
\end{aligned}
$$