Chapter 12. Binary Search Trees

12.1-1.
Height 2 : Root 10, Left 4, Right 17, Left-Left 1, Left-Right 5, Right-Left 16, Right-Right 21.
Height 3 : Root 10, Left 4, Right 16, Left-Left 1, Left-Right 5, Right-Right 17, Right-Right-Right 21.
Height 4 : Root 5, Left 1, Right 10, Left-Right 4, Right-Right 16, Right-Right-Right 17, Right-Right-Right-Right 21.
Height 5 : Root 4, Left 1, Right 5, and 10-16-17-21 are skewed in chain of right child.
Height 5 : 1-4-5-10-16-17-21 are skewed in chain of right child.

12.1-2.
The binary search tree property guarantees that all nodes in the left subtree are smaller than the given node, and all nodes in the right subtree are larger. Whereas, the min-heap property only guarantees that the all nodes in the subtree is bigger than the given node, doesn't distinguishing left and right subtrees. Hence, the min-heap property can't be used to print out the keys in the sorted order in linear time, because we cannot get predecessor/successor in O(1) time.

12.1-3.

Algorithm 1: INORDER-ITERATIVE(T)

```
current = T.root;
while current ≠ NIL do
    if current.left == NIL then
        print current.key;
        current = current.right;
    else
        prev = current.left;
        while prev.right ≠ NIL and prev.right ≠ current do
            prev = prev.right;
        end
        if prev.right == NIL then
            prev.right = current;
            current = current.left;
        else
            prev.right = NIL;
            print current.key;
            current = current.right;
        end
    end
end
```

12.1-4.

Algorithm 2: PREORDER-RECURSIVE(x)

```
if x ≠ NIL then
    print x.key;
    PREORDER-RECURSIVE(x.left);
    PREORDER-RECURSIVE(x.right);
end
```

Algorithm 3: POSTORDER-RECURSIVE(x)

```
if x ≠ NIL then
    POSTORDER-RECURSIVE(x.left);
    POSTORDER-RECURSIVE(x.right);
    print x.key;
end
```

12.1-5.

If we can construct a binary search tree using a comparison-based algorithm with less than $\Omega(n \lg n)$ in the worst case, it means that we can get the sorted elements of the list of n elements less than $\Omega(n \lg n)$ in the worst case, a contradiction.

12.2-1.
c. is impossible, since 240 is the left child of 911 and therefore 912 belongs the left subtree of 911, a contradiction.
e. is also impossible, since 621 is the right child of 347 and therefore 299 belongs the right subtree of 347, a contradiction.

12.2-2.

Algorithm 4: TREE-MINIMUM(x)

```
if x.left ≠ NIL then
|  return TREE-MINIMUM(x.left);
else
|  return x;
end
```

Algorithm 5: TREE-MAXIMUM(x)

```
if x.right ≠ NIL then
|  return TREE-MAXIMUM(x.right);
else
|  return x;
end
```

12.2-3.

Algorithm 6: TREE-PREDECESSOR(x)

```
if x.left ≠ NIL then
|  return TREE-MAXIMUM(x.left);
end
y = x.parent;
while y ≠ NIL and x == y.left do
|  x = y;
|  y = y.parent;
end
return y;
```

12.2-4.
Root 2, Left 1, Right 3, Left-Right 5, Right-Right 4.
Searching for 4 gives $A = \{1, 5\}$, $B = \{2, 3, 4\}$, $C = \{\}$. Since $5 > 2, 3, 4$, this is a counterexample.

12.2-5.
If a node in a binary search tree has two children, then its succcessor is the minimum element of the right subtree, so it cannot have left child. Its predecessor is the maximum element of the left subtree, so it cannot have right child.

12.2-6.
First, we show that y must be an ancestor of x. If there is a common ancestor z of x and y, then $x < z < y$, so y cannot be the successor of x.
Next, y.left must be an ancestor of x, because x must be in the left subtree of y.
y.left.left must not be equal to x, because if it were then y cannot be the successor of x.
Therefore, y is the lowest ancestor of x whose left child is also an ancestor of x.

12.2-7.
We claim that the algorithm requires $O(n)$ time as it traverses each of the $n - 1$ edges at most twice (once going down and once going up). Consider the edge between any node u and its childern v. The only time the edge is traversed downward is in TREE-MINIMUM, the only time the edge is traversed upward is in TREE-SUCCESSOR when we look for the successor of a node that has no right subtree. Therefore the algorithm runs in $O(n)$ time. Since it requires $\Omega(n)$ time to do the n procedure calls, it runs in $\Theta(n)$ time.
12.2-8.
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 at most twice, hence it takes $O(k + h)$ time.

12.2-9.
If $x = y.left$, then $y.key$ is the smallest key in T larger than $x.key$, because x is a leaf.

If $x = y.right$, then $y.key$ is the largest key in T smaller than $x.key$, because x is a leaf.

12.3-1.

Algorithm 7: TREE-INSERT-RECURSIVE(parent, root, z)

```
if root == NIL then
    z.parent = parent;
    if z.key < parent.key then
        parent.left = z;
    else
        parent.right = z;
    end
else if z.key < root.key then
    TREE-INSERT-RECURSIVE(root, root.left, z);
else
    TREE-INSERT-RECURSIVE(root, root.right, z);
end
```

12.3-2.
The nodes examined in searching for a value are the nodes examined when the value was first inserted into the tree plus the node being searched.

12.3-3.
Worst case is $\Theta(n^2)$, occurs when a linear chain of nodes.
Best case is $\Theta(n \lg n)$, occurs when a binary tree of height $\Theta(\lg n)$.

12.3-4.
No. Consider the tree with root 2, left 1, right 4, right-left 3. If we delete 1-2, then we get root 4, left 3. If we delete 2-1, then we get root 3, left 4.

12.3-5.
TREE-SEARCH is unchanged.
We have to implement TREE-PARENT as follows:

Algorithm 8: TREE-PARENT(T, x)

```
if x == T.root then
|   return NIL;
end
y = TREE-MAXIMUM(x).succ;
if y == NIL then
|   y = T.root;
else
|   if x == y.left then
|   |   return y;
|   end
|   y = y.left;
end
while x ≠ y.right do
|   y = y.right;
end
return y;
```

TREE-INSERT becomes:

Algorithm 9: TREE-INSERT(T, z)

```
y = NIL;
x = T.root;
pred = NIL;
while x ≠ NIL do
    y = x;
    if z.key < x.key then
        x = x.left;
    else
        pred = x;
        x = x.right;
if y == NIL then
    T.root = z;
    z.succ = NIL;
else if z.key < x.key then
    y.left = z;
    z.succ = y;
    if pred ≠ NIL then
        pred.succ = z;
else
    y.right = z;
    z.succ = y.succ;
    y.succ = z;
```

TRANSPLANT becomes:

Algorithm 10: TRANSPLANT(T, u, v)

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

TREE-DELETE becomes:

Algorithm 11: TREE-DELETE(T, z)

```
if z == NIL then
    return;
pred = NIL;
if z.left ≠ NIL then
    pred = TREE-MAXIMUM(z.left);
else
    y = PARENT(T, z);
    x = z;
    while y ≠ NIL and x == y.left do
        x = y;
        y = PARENT(T, y);
    pred = y;
pred.succ = z.succ;
if z.left == NIL then
    TRANSPLANT(T, z, z.right);
else if z.right == NIL then
    TRANSPLANT(T, z, z.left);
else
    y = TREE-MINIMUM(z.right);
    if PARENT(T, y) ≠ z then
        TRANSPLANT(T, y, y.right);
        y.right = z.right;
    TRANSPLANT(T, z, y);
    y.left = z.left;
```

All of these three algorithms runs in $O(h)$ time.

12.3-6.

Algorithm 12: TREE-DELETE(T, z)

```
if z == NIL then
    return;
if z.left == NIL then
    TRANSPLANT(T, z, z.right);
else if z.right == NIL then
    TRANSPLANT(T, z, z.left);
else
    y = TREE-MAXIMUM(z.left);
    if y.p ≠ z then
        TRANSPLANT(T, y, y.left);
        y.left = z.left;
        y.left.p = y;
    TRANSPLANT(T, z, y);
    y.right = z.right;
    y.right.p = y;
```

Just randomly choosing y between TREE-MAXIMUM(z.left) and TREE-MINIMUM(z.right) is enough.

12.4-1.
Selecting 4 elements among $n+3$ elements is equivalent to setting one element and selecting remaining 3 elements among the succeeding elements, giving the equality.

12.4-2.
Consider a n-node binary search tree with $n-\sqrt{n \lg n}$ nodes forming a complete binary search tree and the other $\sqrt{n \lg n}$ nodes forming a single chain dangled at the bottom of the upper complete binary search tree part. The height of this tree is $\Theta(\lg(n-\sqrt{n \lg n}))+\sqrt{n \lg n}=\Theta(\sqrt{n \lg n})=\Omega(\lg n)$. There are $n-\sqrt{n \lg n}$ nodes having depth $\Theta(\lg n)$ and $\sqrt{n \lg n}$ nodes having depth at most $O(\lg n+\sqrt{n \lg n})$ in this tree. Hence, the average depth of a node is bounded below from $\frac{1}{n}\Theta((n-\sqrt{n \lg n}) \lg n)=\Omega(\lg n)$, bounded above from $\frac{1}{n}O(\sqrt{n \lg n}(\lg n+\sqrt{n \lg n})+(n-\sqrt{n \lg n}) \lg n)=O(\lg n)$, so it is $\Theta(\lg n)$.
Now we show that if the average depth of a node in an n-node binary search tree is $\Theta(\lg n)$, then the height of the tree is $O(\sqrt{n \lg n})$. If the height of

the tree is h, considering the path from the root to the leaf with depth h, the average depth of a node must be at least $\frac{1}{n}\sum_{d=0}^{h} d = \frac{1}{n}\Theta(h^2)$. Since the average depth is $\Theta(\lg n)$, h must be $O(\sqrt{n \lg n})$.

12.4-3.
With 3 distinct elements, there are 6 permutations but we have only 5 distinct binary search trees.

12.4-4.
A continuous real-valued functon is convex if its second derivative is nonnegative. For $f(x) = 2^x$, its second derivative is $(\lg 2)^2 2^x$, which is positive for all x, hence f is convex.

12.4-5.
A quicksort corresponds the following binary search tree: the initial pivot is the root node, the pivot of the left half is the root of the left subtree, the pivot of the right half is the root of the right subtree, and so on. The number of comparisons of the execution of quicksort equals the number of comparisons during the construction of the binary search tree by a sequence of insertions. Therefore we get the following argument: when running quicksort on a given permutation of n distinct elements, the running time is $O(nh)$, where h is the height of the corresponding binary search tree.
Now, let X_n be the random variable indicating the height of a randomly built binary search tree with n distinct keys. By the proof of Theorem 12.4,

$$\mathbb{E}[2^{X_n}] \leq \frac{1}{4}\binom{n+3}{3} = O(n^3).$$

By Exercise C.3-6,

$$P(X_n \geq (k+3)\lg n) = P(2^{X_n} \geq n^{3+k}) \leq \frac{\mathbb{E}[2^{X_n}]}{n^{3+k}} = O(\frac{1}{n^k}).$$

Therefore, for any constant $k > 0$, the probability that a permutation of n distinct elements yields quicksort with running time $\omega(n \lg n)$ is $O(\frac{1}{n^k})$.

12-1.
a. The nodes will form a single chain, yielding $\Theta(n^2)$ running time.
b. For each nodes, the difference between heights of the two subtrees is at

most 1 in this case. Hence the height of the tree becomes $\Theta(\lg n)$, yielding $\sum_{i=1}^{n} \lg i = \Theta(n \lg n)$.
c. The whole list would be insert once and it will be done, yielding $\Theta(n)$ running time.
d. Worst case happens if the choices are skewed, yielding $\Theta(n^2)$.
Expected running time is equivalent to expected running time of corresponding quicksort (see 12.4-5.), yielding $\Theta(n \lg n)$.

12-2.
To sort S, we first insert them into a radix tree, and use a preorder traversal. The output result is lexicographically sorted.
Correctness: In preorder traversal, any node's string is a prefix of all descendants' strings, and a node's left descendants precedes its right descendants, so it is correct.
Time complexity: Insertion of each string takes time proportional to its length, and since the sum of all the string lengths is n, the whole insertion takes $\Theta(n)$. The preorder walk is $\Theta(n)$, so the whole procedure takes $\Theta(n)$.

12-3.
a. Since $P(T) = \sum_{x \in T} d(x, T)$, the equality holds.
b.
$$P(T) = \sum_{x \in T} d(x, T) = \sum_{x \in T_L} d(x, T) + \sum_{x \in T_R} d(x, T)$$
$$= \sum_{x \in T_L} (d(x, T_L) + 1) + \sum_{x \in T_R} (d(x, T_R) + 1) = P(T_L) + P(T_R) + n - 1.$$

c. Since the root is equally likely to be any of n elements in the tree and the number of nodes in subtree T_L (and T_R) is equally likely to be any integer in the set $\{0, \cdots, n-1\}$. Combining with b., we get

$$P(T) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n - i - 1) + n - 1).$$

d. Since $P(0) = 0$ and each $P(k)$ occurs twice, we have

$$P(T) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n).$$

e. We use the recurrence relation. Suppose $P(k) = O(k\lg k)$ for all $k < n$. Then for some $a, b > 0$,

$$\begin{aligned}
P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) &\leq \frac{2}{n}\sum_{k=2}^{n-1}(ak\lg k + b) + \Theta(n) \\
&= \frac{2a}{n}\sum_{k=2}^{n-1} k\lg k + \frac{2b}{n}(n-2) + \Theta(n) \\
&\leq \frac{2a}{n}(\sum_{k=2}^{\lceil \frac{n}{2}\rceil - 1} k\lg k + \sum_{k=\lceil \frac{n}{2}\rceil - 1}^{n-1} k\lg k) + 2b + \Theta(n) \\
&< \frac{2a}{n}[(\lg n - 1)\sum_{k=2}^{\lceil \frac{n}{2}\rceil - 1} k + \lg n \sum_{k=\lceil \frac{n}{2}\rceil - 1}^{n-1} k] + 2b + \Theta(n) \\
&\leq \frac{2a}{n}[\frac{1}{2}n^2\lg n - \frac{1}{8}n^2] + 2b + \Theta(n) \\
&\leq an\lg n + b.
\end{aligned}$$

Therefore, $P(n) = O(n\lg n)$.
f. See 12.4-5.

12-4.
a. A binary tree with n nodes can be constructed with connecting two binary trees with k nodes and $n-1-k$ nodes and the parent (root), hence we get

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

b.

$$\begin{aligned}
xB(x)^2 + 1 = 1 + x(\sum_{n=0}^{\infty} b_n x^n)^2 &= 1 + x\sum_{n=1}^{\infty}\sum_{k=0}^{n} b_k b_{n-1-k} x^k \\
&= 1 + \sum_{n=1}^{\infty} b_n x^n = B(x).
\end{aligned}$$

Hence

$$B(x) = \frac{1}{2x}(1 - \sqrt{1-4x}).$$

c. The Taylor expansion of $f(x) = \sqrt{1-4x}$ gives

$$\sqrt{1-4x} = \sum_{n=0}^{\infty} \binom{\frac{1}{2}}{n}(-4x)^n = \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{4^n(2n-1)}\binom{2n}{n}(-4x)^n$$

$$= -\sum_{n=0}^{\infty} \frac{1}{2n-1}\binom{2n}{n}x^n.$$

Substituting this into b. gives

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

d. By Stirling's approximation,

$$b_n = \frac{1}{n+1}\binom{2n}{n} \simeq \frac{1}{n+1}\frac{\sqrt{4\pi n}(\frac{2n}{e})^{2n}}{2\pi n(\frac{n}{e})^{2n}} = \frac{4^n}{(n+1)\sqrt{\pi n}} = \frac{4^n}{\sqrt{\pi}n^{\frac{3}{2}}}(1 - \frac{1}{n+1})$$

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