Binary Search Trees

ITX2010, CSX3003, IT2230

Data Structures and Algorithms, Information Structures

Learning Objectives

Students will be able to:

- Understand what binary search tree is.
- Examine basic operations on binary search tree takes time in proportional to the height of the tree.
- Understand how to manipulate binary search tree structure via insertion and deletion operation.
- Estimate a random binary search tree performance

Chapter Outline

1. What is a binary search tree?
2. Querying a binary search tree
3. Insertion and Deletion
4. Randomly built binary search trees

12.1

What is a binary search tree?

What is a binary search tree?

BST

- A binary search tree (BST) is a binary tree whose node contains: [1]
  - Left, right and p that point to the nodes corresponding to its left child, its right child and its parent, respectively.
  - The key is always stored and satisfied with the binary search tree property.

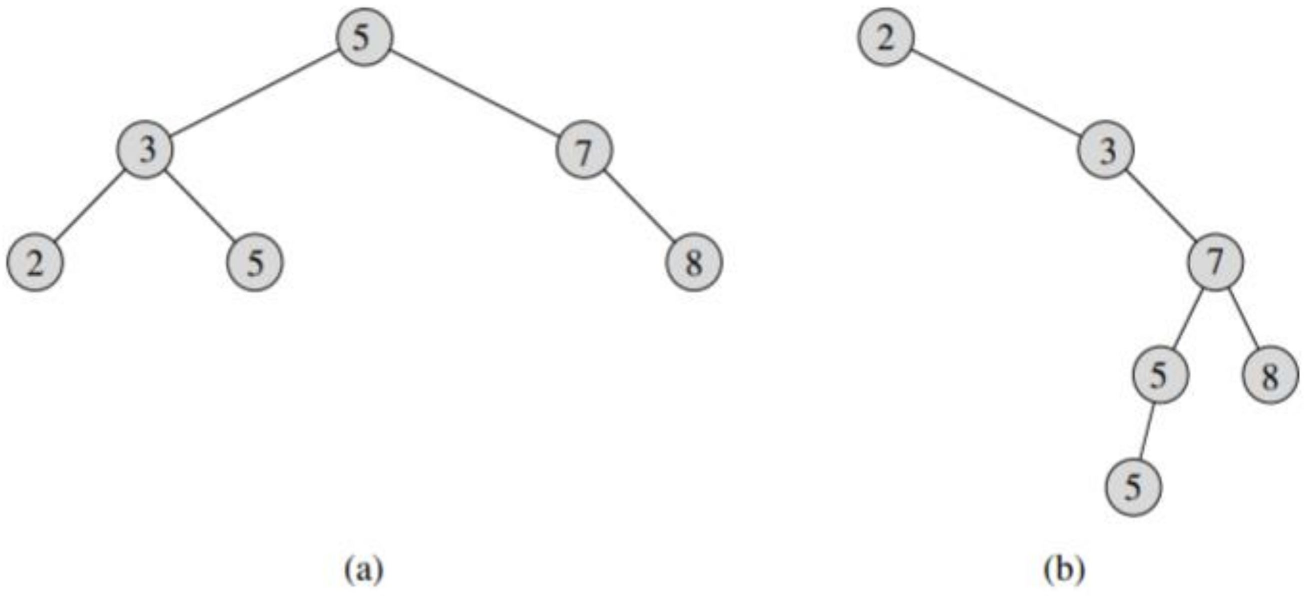


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys. [1]

What is a binary search tree?

BST Property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[x] \leq key[y]$. [1]

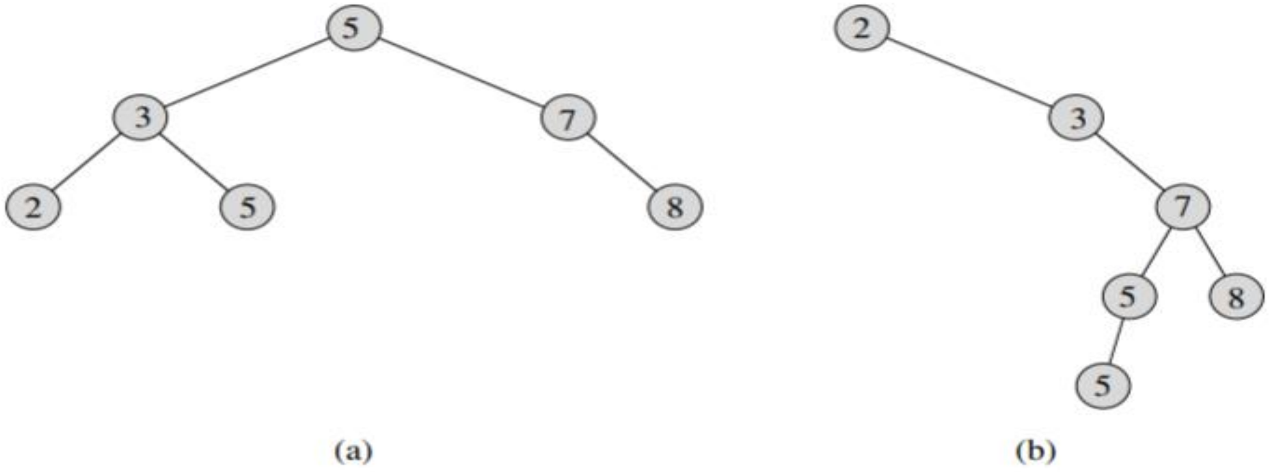


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys. [1]

1. It is a binary tree.
2. If y is a node in:
   - The left subtree of x, then key[y] ≤ key[x]
   - The right subtree of x, then key[x] ≤ key[y]
3. Both subtrees of each node are also BST.

What is a binary search tree?

BST Inorder Walk

- Print out all the keys in BST in sorted order by a simple recursive algorithm.

- The root of a subtree is printed between the values in its left subtree and those in its right subtree - Left->Root->Right

- It take Θ(n) time to walk n-node binary search tree! **[1]**

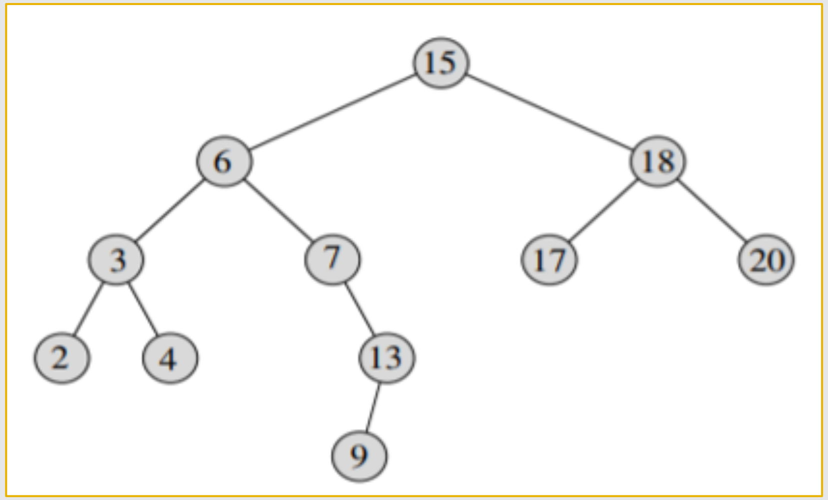
```
INORDER-TREE-WALK(x)
if x ≠ NIL
    then INORDER-TREE-WALK(left[x])
         print key[x]
         INORDER-TREE-WALK(right[x])
```

```
# Binary Search Tree
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
# Inorder traversal
def inorder(root): [3]
    if root is not None:
        # Traverse left
        inorder(root.left)
        # Traverse root
        print(str(root.key) + "->", end=' ')
        # Traverse right
        inorder(root.right)
# Search a node
def searchNode(root, key):...
# Find the inorder successor
def minValueNode(node):...
# Find the inorder successor
def minValueNode(node):...
# Insert a node
def insert(node, key):...
# Deleting a node
def deleteNode(root, key):...
#Main
root = None
root = insert(root, 15)
root = insert(root, 6)
root = insert(root, 18)
root = insert(root, 3)
root = insert(root, 7)
root = insert(root, 17)
root = insert(root, 20)
root = insert(root, 2)
root = insert(root, 4)
root = insert(root, 13)
root = insert(root, 9)
print("Inorder traversal: ", end=' ')
inorder(root)
```

```
Inorder traversal:  2-> 3-> 4-> 6-> 7-> 9-> 13-> 15-> The thread
  'MainThread' (0x1) has exited with code 0 (0x0).
17-> 18-> 20-> The program 'python.exe' has exited with code 0 (0x0).
```

INORDER-TREE-WALK(x) [1]

1 **if** $x \neq$ NIL
2 **then** INORDER-TREE-WALK($left[x]$)
3 print $key[x]$
4 INORDER-TREE-WALK($right[x]$)



What is a binary search tree?

Theorem 12.1 [1]

If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

- When n>0, a node x whose left subtree has k, right subtree has n-k-1 nodes,
- The time to perform function will plus d - for some positive constant d that reflects the time to execute function, exclusive of the time spent in recursive calls, as follows:

We use the substitution method to show that $T(n) = \Theta(n)$ by proving that $T(n) = (c+d)n + c$. For $n = 0$, we have $(c+d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$
\begin{aligned}
T(n) &= T(k) + T(n-k-1) + d \\
&= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= (c+d)n + c ,
\end{aligned}
$$

which completes the proof. ∎

12.2

Querying a binary search tree

Querying a binary search tree

BST Query Operations

- Searching for a key stored in BST.

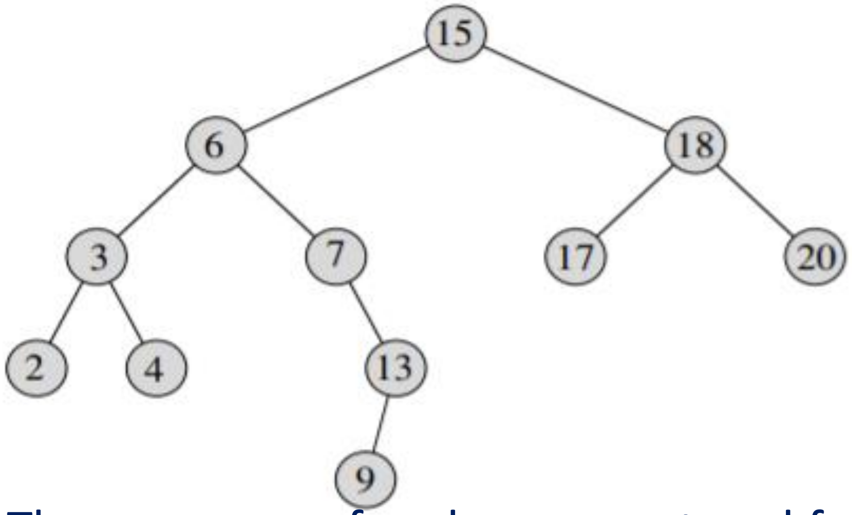
- Finding:
  - Minimum, Maximum
  - Successor and Predecessor in a sorted order determined by inorder tree walk (or inorder traversal).

- Each can be supported in time O(h) on a binary search tree of height h. [1]

Querying a binary search tree

BST Search

12.2 *Querying a binary search tree* [1] 257




```
TREE-SEARCH(x, k) [1]
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else return TREE-SEARCH(right[x], k)
```

```
ITERATIVE-TREE-SEARCH(x, k) [1]
while x ≠ NIL and k ≠ key[x]
    do if k < key[x]
          then x ← left[x]
          else x ← right[x]
return x
```

The sequence of nodes encountered forms a path downward from the root, the function run in O(h) time on a tree of height h.

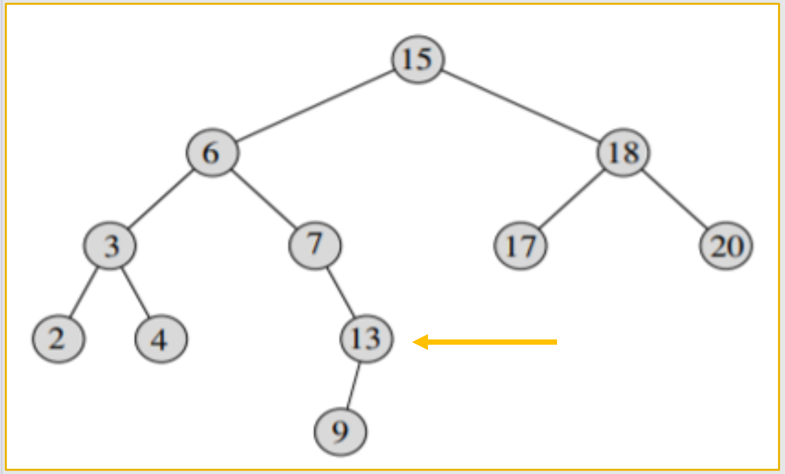
Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

```python
# Binary Search Tree
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
# Inorder traversal
def inorder(root):...
# Search a node
def searchNode(root, key):
    # Return if the tree is empty
    if (root is None):
        return print("Seaching is ended.")
    if (key == root.key):
            return print("\nSearch node:", root.key, "is found.")
    if key < root.key:
        return searchNode(root.left, key)
    else:
        return searchNode(root.right, key)
# Find the inorder successor
def minValueNode(node):...
# Find the inorder successor
def minValueNode(node):...
# Insert a node
def insert(node, key):...
# Deleting a node
def deleteNode(root, key):...
#Main
root = None
root = insert(root, 15)
root = insert(root, 6)
root = insert(root, 18)
root = insert(root, 3)
root = insert(root, 7)
root = insert(root, 17)
root = insert(root, 20)
root = insert(root, 2)
root = insert(root, 4)
root = insert(root, 13)
root = insert(root, 9)
print("Inorder traversal: ", end=' ')
inorder(root)
searchNode(root, 13)
```

```
Inorder traversal:  2-> 3-> 4-> 6-> 7-> 9-> 13-> 15-> 17-> 18-> 20->
Search node: 13 is found.
The thread 'MainThread' (0x1) has exited with code 0 (0x0).
The program 'python.exe' has exited with code 0 (0x0).
```

TREE-SEARCH(x, k) [1]

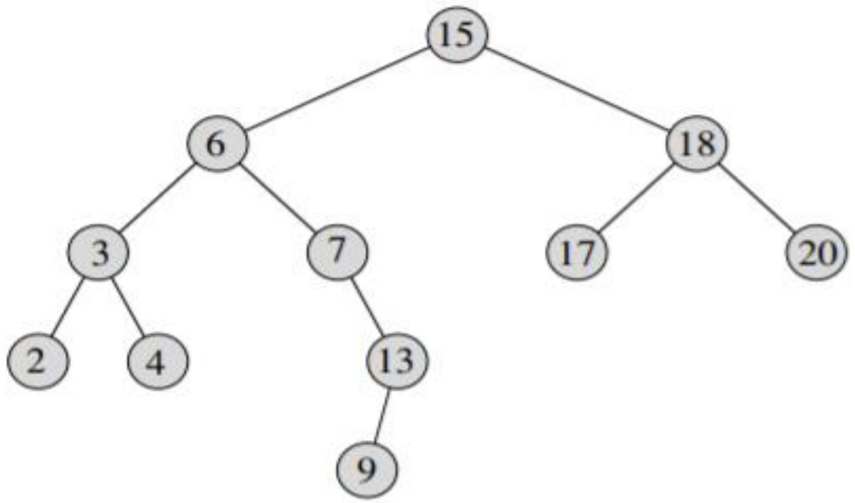
```
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else return TREE-SEARCH(right[x], k)
```



Querying a binary search tree

BST Maximum

12.2 Querying a binary search tree [1]



TREE-MAXIMUM(x) [1]

while $right[x] \neq$ NIL
  do $x \leftarrow right[x]$
return x

Like search function, the function run in O(h) time on a tree of height h.

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

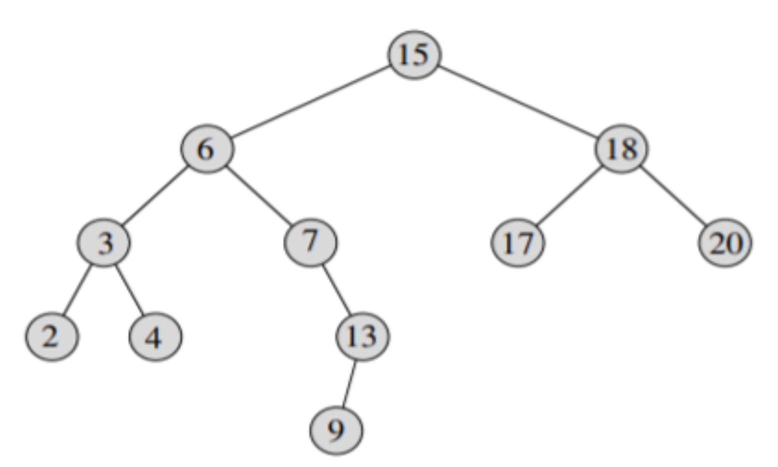
12.2

```python
# Binary Search Tree
class Node:
    def __init__(self, key):...
# Inorder traversal
def inorder(root):...
# Search a node
def searchNode(root, key):...
# Find the inorder successor
def minValueNode(node):...
# Find the maximum value node in BST
def maxValueNode(node):
    current = node
    # Find the leftmost leaf
    while(current.right is not None):
        current = current.right
    return print("Maximum value node
      is",current.key)
# Insert a node
def insert(node, key):...
# Deleting a node
def deleteNode(root, key):...
#Main
root = None
root = insert(root, 15)
root = insert(root, 6)
root = insert(root, 18)
root = insert(root, 3)
root = insert(root, 7)
root = insert(root, 17)
root = insert(root, 20)
root = insert(root, 2)
root = insert(root, 4)
root = insert(root, 13)
root = insert(root, 9)
print("Inorder traversal: ", end=' ')
inorder(root)
searchNode(root, 13)
maxValueNode(root)
```

```
Inorder traversal:  2-> 3-> 4-> 6-> 7-> 9-> 13-> 15-> 17-> 18-> 20->
Search node: 13 is found.
Maximum value node is 20
The thread 'MainThread' (0x1) has exited with code 0 (0x0).
The program 'python.exe' has exited with code 0 (0x0).
```

TREE-MAXIMUM(x) [1]

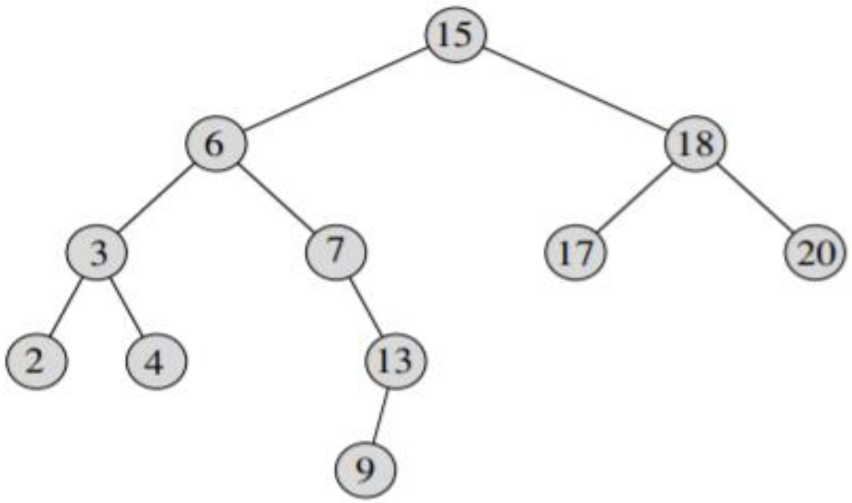
1 **while** $right[x] \neq$ NIL
2   **do** $x \leftarrow right[x]$
3 **return** x



Querying a binary search tree

BST Minimum

12.2 Querying a binary search tree [1]



TREE-MINIMUM(x) [1]

```
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x
```

Like search function, the function run in O(h) time on a tree of height h.

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

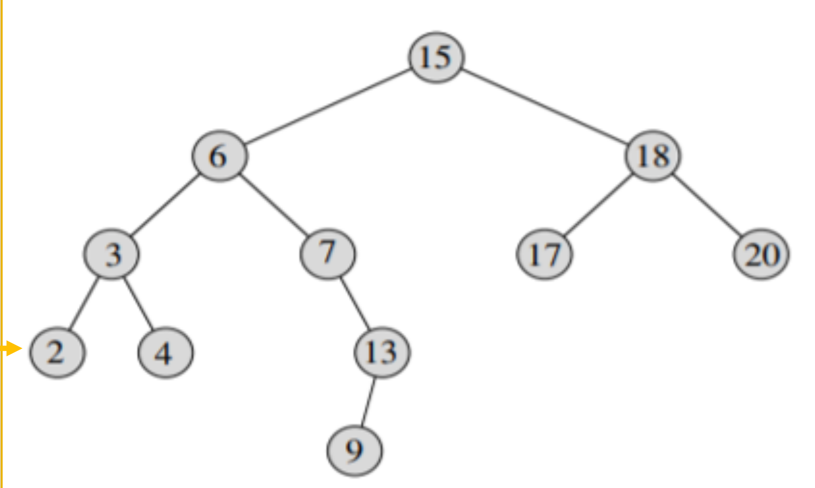
12.2

```python
# Binary Search Tree
class Node:
    def __init__(self, key):...
# Inorder traversal
def inorder(root):...
# Search a node
def searchNode(root, key):...
# Find the inorder successor
def minValueNode(node): [3]
    current = node
    # Find the leftmost leaf
    while(current.left is not None):
        current = current.left
    return print("Minumum value node is",current.key)
# Find the maximum value node in BST
def maxValueNode(node):...
# Insert a node
def insert(node, key):...
# Deleting a node
def deleteNode(root, key):...
#Main
root = None
root = insert(root, 15)
root = insert(root, 6)
root = insert(root, 18)
root = insert(root, 3)
root = insert(root, 7)
root = insert(root, 17)
root = insert(root, 20)
root = insert(root, 2)
root = insert(root, 4)
root = insert(root, 13)
root = insert(root, 9)
print("Inorder traversal: ", end=' ')
inorder(root)
searchNode(root, 13)
maxValueNode(root)
minValueNode(root)
```

```
Inorder traversal:  2-> 3-> 4-> 6-> 7-> 9-> 13-> 15-> 17-> 18-> 20->
Search node: 13 is found.
Maximum value node is 20
Minumum value node is 2
The thread 'MainThread' (0x1) has exited with code 0 (0x0).
The program 'python.exe' has exited with code 0 (0x0).
```

TREE-MINIMUM(x) [1]

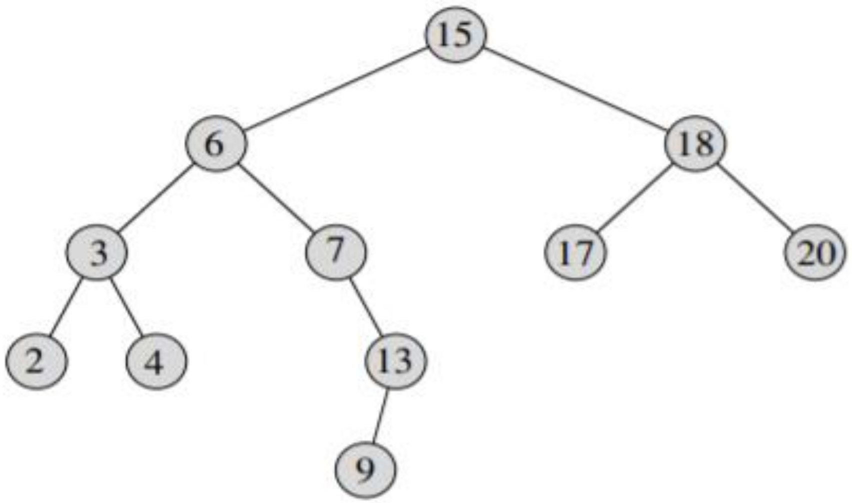
1. **while** $left[x] \neq$ NIL
2. **do** $x \leftarrow left[x]$
3. **return** x



Querying a binary search tree

BST Successor in inorder walk

12.2 Querying a binary search tree [1] 257




```
TREE-SUCCESSOR(x) [1]
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
    do x ← y
       y ← p[y]
return y
```

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

- If all keys are distinct, the successor of a node x is the node with the smallest key greater than key[x].
- BST allows to determine the successor of a node without ever comparing keys.
- The running time on a tree of height h is O(h), since we either follow a path up or down the tree.

```python
def searchNode(root, key):...
# Find the inorder successor
def minValueNode(node):
    current = node
    # Find the leftmost leaf
    while(current.left is not None):
        current = current.left
    return print("Minumum value node is",current.key)
# Find the maximum value node in BST
def maxValueNode(node):
    current = node
    # Find the leftmost leaf
    while(current.right is not None):
        current = current.right
    return print("Maximum value node is",current.key)
def successorNode(node):
    current =node
    if current.right is None:
        return maxValueNode(current.left),print(" -> is the successor of the node", current.key)
    else:
        return minValueNode(current.right),print(" -> is the successor of node", current.key)
# Insert a node
def insert(node, key):...
# Deleting a node
def deleteNode(root, key):...
#Main
root = None
root = insert(root, 15)
root = insert(root, 6)
root = insert(root, 18)
root = insert(root, 3)
root = insert(root, 7)
root = insert(root, 17)
root = insert(root, 20)
root = insert(root, 2)
root = insert(root, 4)
root = insert(root, 13)
root = insert(root, 9)
print("Inorder traversal: ", end=' ')
inorder(root)
searchNode(root, 13)
maxValueNode(root)
minValueNode(root)
successorNode(root)
```

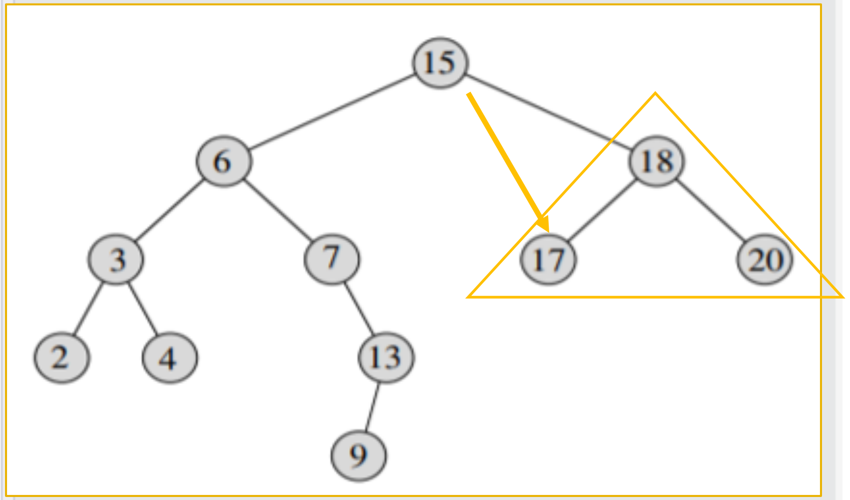
```
Inorder traversal:  2-> 3-> 4-> 6-> 7-> 9-> 13-> 15-> 17-> 18-> 20->

Search node: 13 is found.
Maximum value node is 20
Minumum value node is 2
Minumum value node is 17
 -> is the successor of node 15
The thread 'MainThread' (0x1) has exited with code 0 (0x0).
The program 'python.exe' has exited with code 0 (0x0).
```

12.2

TREE-SUCCESSOR(x) [1]

if $right[x] \neq$ NIL
  then return TREE-MINIMUM$(right[x])$
$y \leftarrow p[x]$
while $y \neq$ NIL and $x = right[y]$
  do $x \leftarrow y$
    $y \leftarrow p[y]$
return y



12.3

Insertion and Deletion

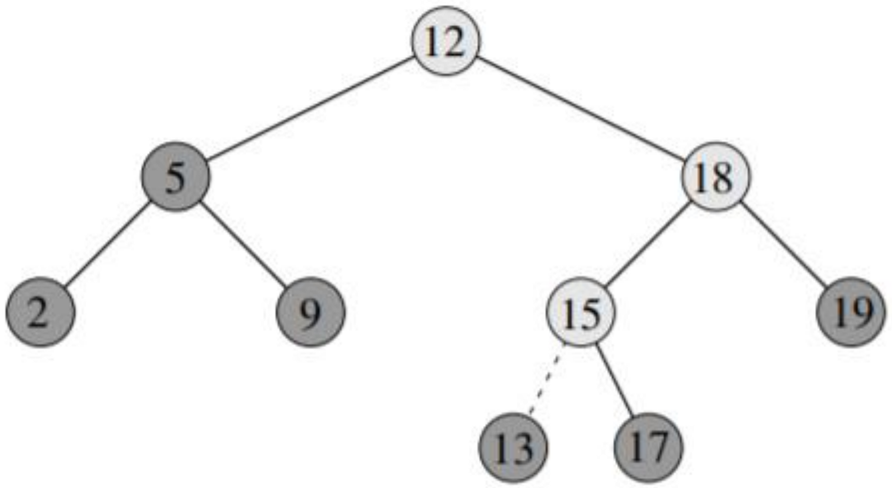
Insertion and Deletion

- They cause the dynamic set represented by a binary search tree to change.
- The structure must be modified in such a way that binary search tree property continues to hold.
- Both can be supported in time O(h) on a binary search tree of height h. [1]

Querying a binary search tree

BST Insertion

262 *Chapter 12 Binary Search Trees*




```
TREE-INSERT(T, z) [1]
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else  x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                          ▷ Tree T was empty
   else  if key[z] < key[y]
            then left[y] ← z
            else  right[y] ← z
```

Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item. [1]

```python
# Insert a node
def insert(node, key): [3]
    # Return a new node if the tree is empty
    if node is None:
        return Node(key)
    # Traverse to the right place and insert the node
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    return node

# Deleting a node
def deleteNode(root, key):...
#Main
root = None
root = insert(root, 12)
root = insert(root, 5)
root = insert(root, 18)
root = insert(root, 2)
root = insert(root, 9)
root = insert(root, 15)
root = insert(root, 19)
root = insert(root, 17)
root = insert(root, 13)
print("Inorder traversal: ", end=' ')
inorder(root)
```

```
Inorder traversal:  2-> 5-> 9-> 12-> 13-> 15-> 17-> 18-> 19-> The thread
  'MainThread' (0x1) has exited with code 0 (0x0).
The program 'python.exe' has exited with code 0 (0x0).
```

TREE-INSERT(T, z) [1]

$y \leftarrow$ NIL
$x \leftarrow root[T]$
while $x \neq$ NIL
    do $y \leftarrow x$
        if $key[z] < key[x]$
            then $x \leftarrow left[x]$
            else $x \leftarrow right[x]$
$p[z] \leftarrow y$
if $y =$ NIL
    then $root[T] \leftarrow z$ ▷ Tree T was empty
    else if $key[z] < key[y]$
            then $left[y] \leftarrow z$
            else $right[y] \leftarrow z$

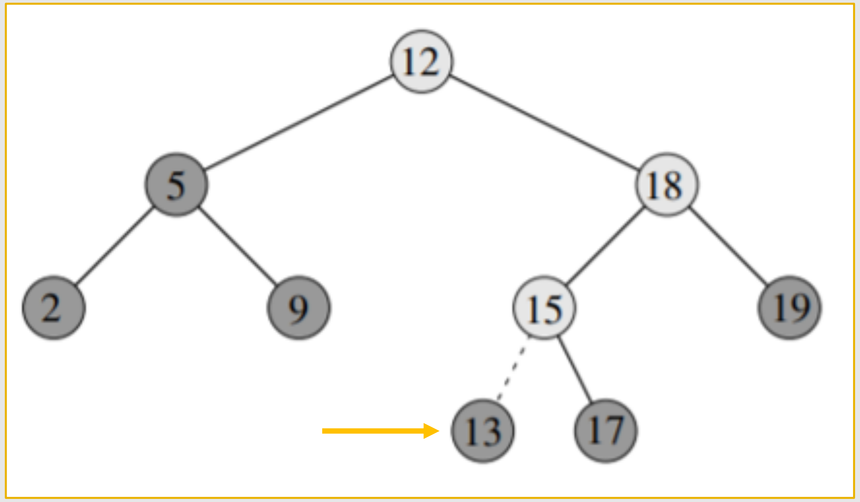


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item. [1]

Querying a binary search tree

BST Deletion

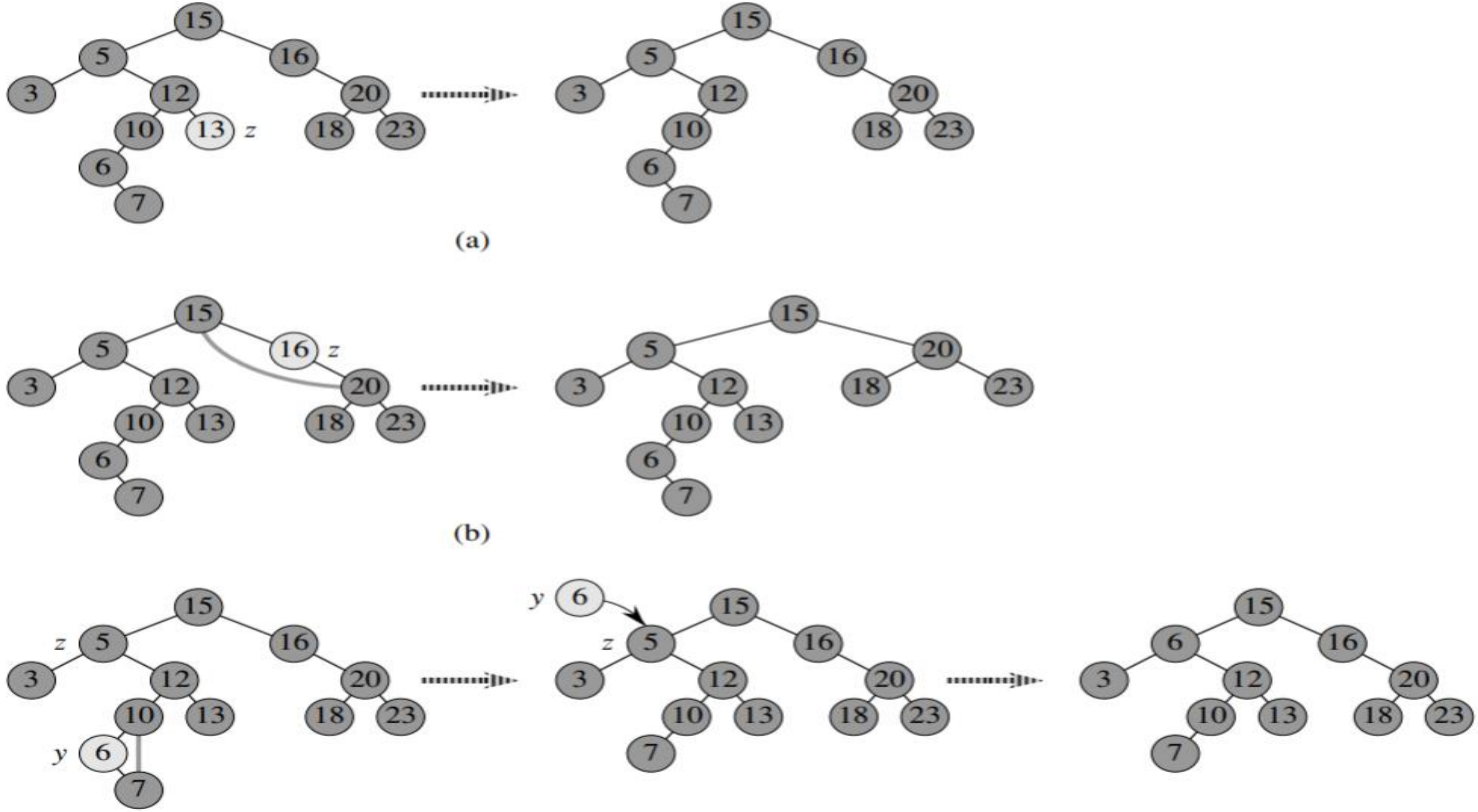
```
TREE-DELETE(T, z)   [1]
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
         copy y's satellite data into z
return y
```

[1]

12.3

12.3 Insertion and deletion

Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data. [1]



12.3

```python
# Deleting a node
def deleteNode(root, key):
    # Return if the tree is empty
    if root is None:
        return root
    # Find the node to be deleted
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)
    else:
        # If the node is with only one child or no child
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        # If the node has two children,
        # place the inorder successor in position of the node to be deleted
        temp = minValueNode(root.right)
        root.key = temp.key
        # Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)
    return root

#Main
root = None
root = insert(root, 15)
root = insert(root, 5)
root = insert(root, 16)
root = insert(root, 3)
root = insert(root, 12)
root = insert(root, 20)
root = insert(root, 10)
root = insert(root, 18)
root = insert(root, 23)
root = insert(root, 6)
root = insert(root, 7)
print("\nInorder traversal: ", end=' ')
inorder(root)
deleteNode(root,13)
print("\nInorder traversal after delete node 13:", end=' ')
inorder(root)
deleteNode(root,16)
print("\nInorder traversal after delete node 16:", end=' ')
inorder(root)
deleteNode(root,6)
print("\nInorder traversal after delete node 6:", end=' ')
inorder(root)
```

```
Inorder traversal:  3-> 5-> 6-> 7-> 10-> 12-> 15-> 16-> 18-> 20-> 23-> The thread
  'MainThread' (0x1) has exited with code 0 (0x0).

Inorder traversal after delete node 13: 3-> 5-> 6-> 7-> 10-> 12-> 15-> 16-> 18-> 20-> 23->
Inorder traversal after delete node 16: 3-> 5-> 6-> 7-> 10-> 12-> 15-> 18-> 20-> 23->
Inorder traversal after delete node 6: 3-> 5-> 7-> 10-> 12-> 15-> 18-> 20-> 23-> The progra
  'python.exe' has exited with code 0 (0x0).
```

12.3 Insertion and deletion 263

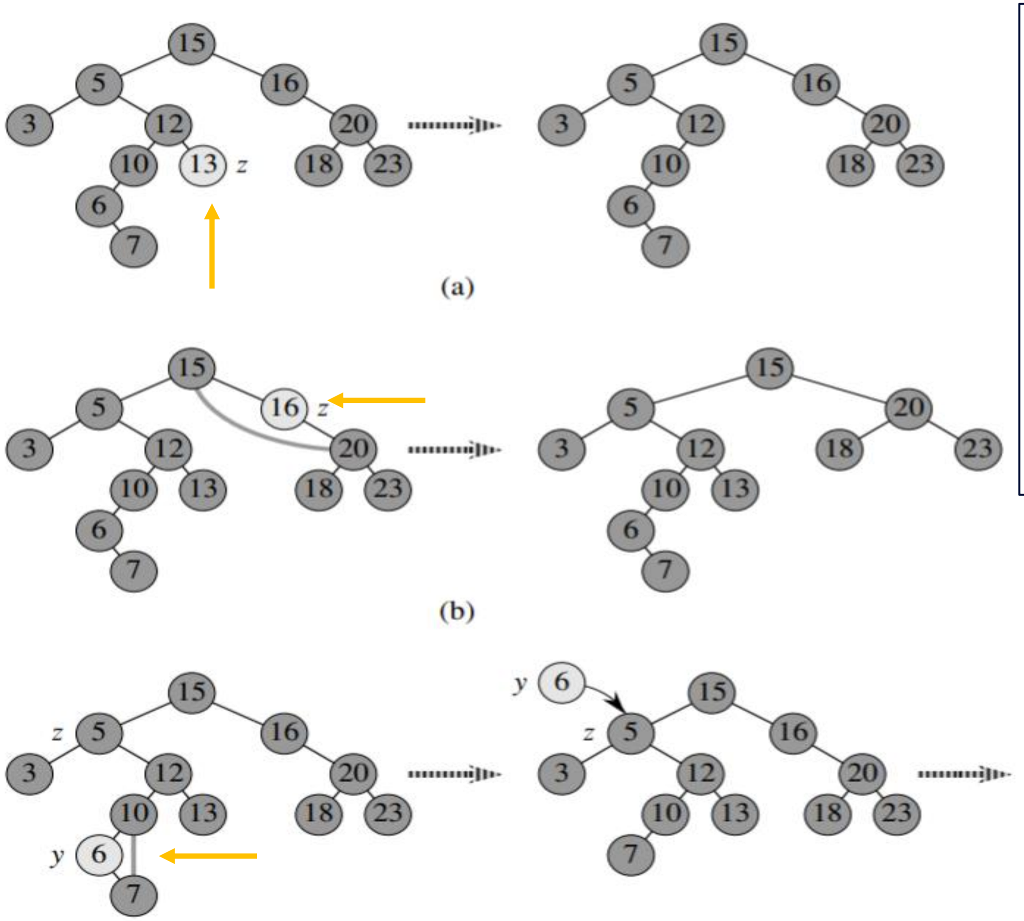


Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data. [1]

```
TREE-DELETE(T, z)  [1]
 if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
 if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
 if x ≠ NIL
    then p[x] ← p[y]
 if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
 if y ≠ z
    then key[z] ← key[y]
         copy y's satellite data into z
 return y
```

12.4

Randomly built binary search trees

Randonly built binary search trees

- The height of binary search tree varies, however, as items are inserted and deleted.
- If the items are inserted in strictly increasing order, the tree will be a chain with height n-1 !
- When the tree is created by insertion alone, the analysis becomes more traceable – called as a randomly built binary search tree on n keys. [1]

[1]

Randonly built binary search trees

- When inserting the keys in random order into an initially empty binary search tree, [1]
- Given the height of a randomly built binary search on n keys by X_n and we define:
  - The exponential height $Y_n = n^{X_n}$
  - R_n denote the random variable that holds a key
- If $R_n = i$, then the left subtree of the root is a randomly built binary search tree on n-i keys.
  - Because the height of a binary tree is one more than the larger of the heights of the two subtrees of the root, the exponential height of a binary tree is twice the larger of the exponential heights of the two subtrees of the root.

Randonly built binary search trees

- If R_n = i, then the left subtree of the root is a randomly built binary search tree on n-i keys: [1]

$R_n = i$, we therefore have that

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) .$$

As base cases, we have $Y_1 = 1$, because the exponential height of a tree with 1 node is $2^0 = 1$ and, for convenience, we define $Y_0 = 0$.

Next we define indicator random variables $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, where

$$Z_{n,i} = \mathrm{I}\{R_n = i\} .$$

Because R_n is equally likely to be any element of $\{1, 2, \ldots, n\}$, we have that $\Pr\{R_n = i\} = 1/n$ for $i = 1, 2, \ldots, n$, and hence, by Lemma 5.1,

$$\mathrm{E}[Z_{n,i}] = 1/n , \tag{12.1}$$

for $i = 1, 2, \ldots, n$. Because exactly one value of $Z_{n,i}$ is 1 and all others are 0, we also have

Randonly built binary search trees

- If R_n = i, then the left subtree of the root is a randomly built binary search tree on n-i keys: **[1]**

$$Y_n = \sum_{i=1}^{n} Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))\ .$$

We will show that $E[Y_n]$ is polynomial in n, which will ultimately imply that $E[X_n] = O(\lg n)$.

The indicator random variable $Z_{n,i} = I\{R_n = i\}$ is independent of the values of Y_{i-1} and Y_{n-i}. Having chosen $R_n = i$, the left subtree, whose exponential height is Y_{i-1}, is randomly built on the $i-1$ keys whose ranks are less than i. This subtree is just like any other randomly built binary search tree on $i-1$ keys. Other than the number of keys it contains, this subtree's structure is not affected at all by the choice of $R_n = i$; hence the random variables Y_{i-1} and $Z_{n,i}$ are independent. Likewise, the right subtree, whose exponential height is Y_{n-i}, is randomly built on the $n-i$ keys whose ranks are greater than i. Its structure is independent of the value of R_n, and so the random variables Y_{n-i} and $Z_{n,i}$ are independent. Hence,

Randomly built binary search trees

- If R_n = i, then the left subtree of the root is a randomly built binary search tree on n-i keys: [1]

$$
\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(by independence)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(by equation (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}[\max(Y_{i-1}, Y_{n-i})] && \text{(by equation (C.21))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(by Exercise C.3-4)}\ .
\end{aligned}
$$

Randonly built binary search trees

- If R_n = i, then the left subtree of the root is a randomly built binary search tree on n-i keys: [1]

$i=1$

Each term $E[Y_0], E[Y_1], \ldots, E[Y_{n-1}]$ appears twice in the last summation, once as $E[Y_{i-1}]$ and once as $E[Y_{n-i}]$, and so we have the recurrence

$$E[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] . \qquad (12.2)$$

Using the substitution method, we will show that for all positive integers n, the recurrence (12.2) has the solution

$$E[Y_n] \le \frac{1}{4} \binom{n+3}{3} .$$

In doing so, we will use the identity

Randonly built binary search trees

- If R_n = i, then the left subtree of the root is a randomly built binary search tree on n-i keys: [1]

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4} . \qquad (12.3)$$

(Exercise 12.4-1 asks you to prove this identity.)

For the base case, we verify that the bound

$$1 = Y_1 = \mathrm{E}[Y_1] \le \frac{1}{4}\binom{1+3}{3} = 1$$

holds. For the substitution, we have that

Randonly built binary search trees

$$
\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\
&= \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(by the inductive hypothesis) [1]} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(by equation (12.3)) [1]} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3}.
\end{aligned}
$$

Randonly built binary search trees

We have bounded $E[Y_n]$, but our ultimate goal is to bound $E[X_n]$. As Exercise 12.4-4 asks you to show, the function $f(x) = 2^x$ is convex (see page 1109). Therefore, we can apply Jensen's inequality (C.25), which says that [1]

$$2^{E[X_n]} \le E[2^{X_n}] = E[Y_n] ,$$

to derive that

$$\begin{aligned} 2^{E[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\ &= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\ &= \frac{n^3 + 6n^2 + 11n + 6}{24} . \end{aligned}$$

Taking logarithms of both sides gives $E[X_n] = O(\lg n)$. Thus, we have proven the following:

Randonly built binary search trees

We have bounded $E[Y_n]$, but our ultimate goal is to bound $E[X_n]$. As Exercise 12.4-4 asks you to show, the function $f(x) = 2^x$ is convex (see page 1109). Therefore, we can apply Jensen's inequality (C.25), which says that

$$2^{E[X_n]} \le E\left[2^{X_n}\right] = E[Y_n] ,$$

to derive that

$$\begin{aligned} 2^{E[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\ &= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\ &= \frac{n^3 + 6n^2 + 11n + 6}{24} . \end{aligned}$$

Taking logarithms of both sides gives $E[X_n] = O(\lg n)$. Thus, we have proven the following:

Theorem 12.4
The expected height of a randomly built binary search tree on n keys is $O(\lg n)$. ■ [1]

References

Texts | Integrated Development Environment (IDE)

[1] Introduction to Algorithms, Second Edition, Thomas H. C., Charles E. L., Ronald L. R., Clifford S., The MIT Press, McGraw-Hill Book Company, Second Edition 2001.

[2] https://visualstudio.microsoft.com/

[3] https://www.programiz.com/dsa/binary-search-tree