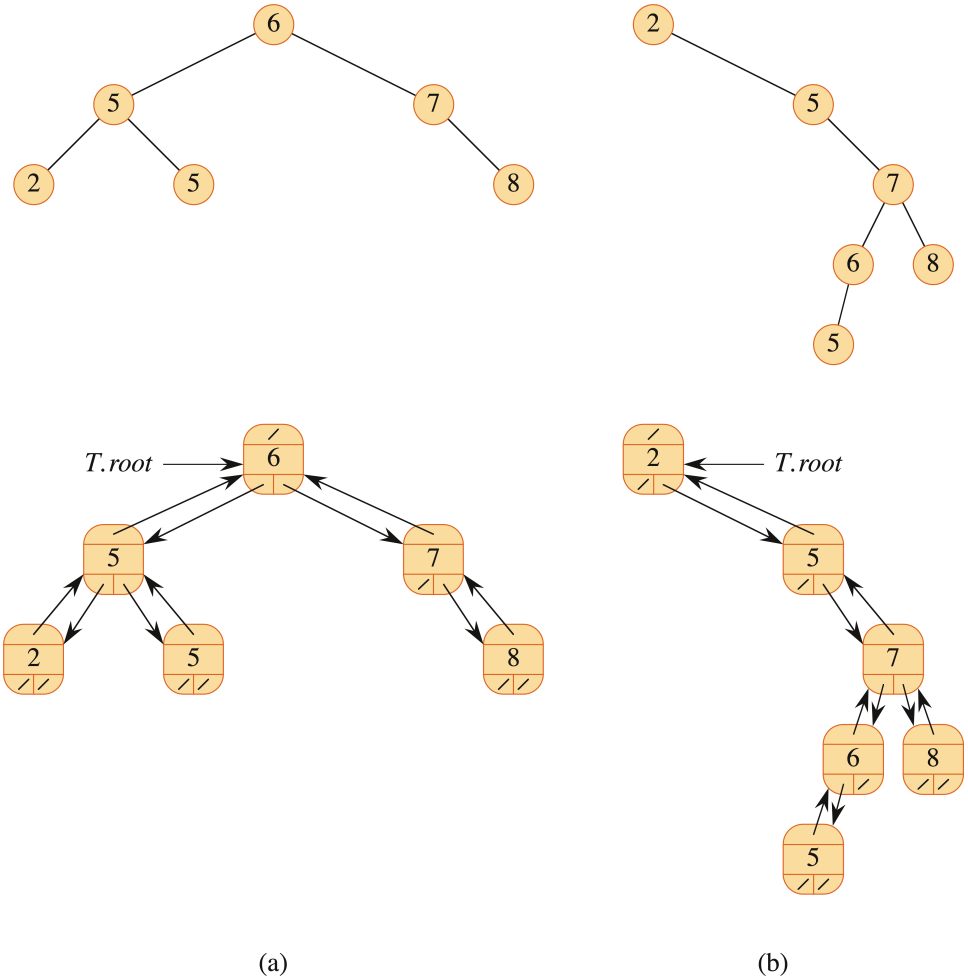
Chapter 13: Red-Black Trees

What is a Binary Search Tree (BST)?

A binary search tree is a binary tree with the following two properties:

1. If a node y is in the left subtree of a node x, then $y.key \leq x.key$.
2. If a node y is in the right subtree of a node x, then $y.key \geq x.key$.

Java: For more general objects implementing the `Comparable` interface, use $y.compareTo(x)$



(a) (b)

Inorder tree walk can print out all the keys in a binary search tree in a sorted order.

```
INORDER-TREE-WALK(x)
1. if x != NIL
2.    INORDER-TREE-WALK(x.left)
3.    print x
4.    INORDER-TREE-WALK(x.right)
```

We can similarly do a **preorder** and **postorder** walk or traversal. In preorder traversal, print the object in the node before traversing left and right subtrees. In postorder traversal, print the object in the node after traversing left and right subtrees. All of them take worst-case runtime of $\Theta(n)$.

Recommended Exercises: 12.1-1, 12.1-2, 12.1-4. *Challenging:* 12.1-3 (solved on next page)

Other Recommended Exercises:

1. Write a recursive procedure for counting the number of nodes in a binary search tree.

   Solution:

   ```
   COUNT-NODES(x)
   1. if x == NIL
   2.     return 0
   3. count = 1
   4. count = count + COUNT-NODES(x.left)
   5. count = count + COUNT-NODES(x.right)
   6, return count
   ```

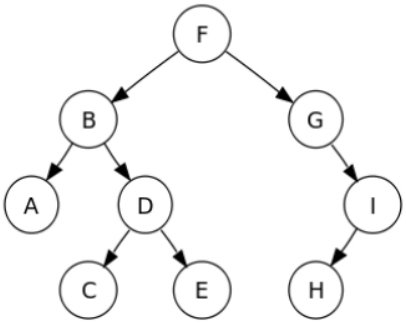
2. Write a recursive procedure for finding the height of a binary search tree.

You should write the procedure for the height on your own!

Solution 12.1-3:

```
ITERATIVE-INORDER-TREE-WALK(x)
1. initialize an empty stack S
2. while true
2.    //traverse to the leftmost leaf
3.    while x != NIL
4.       PUSH(S, x)
5.       x = x.left
6.    //if stack is empty, then we are done
7.    if EMPTY-STACK(S)
8.        return
9.    //pop the top element, print it and then add nodes
10.   //in the right subtree to the stack
11.   x = POP(S)
12.   print x.data
13.   x = x.right
```

Example:



Step	Output	Stack
0		F,B,A
1	A	F,B
2	B	F,D,C
3	C	F,D
4	D	F,E
5	E	F
6	F	G
7	G	I,H
8	H	I
9	I	

Querying a Binary Search Tree

Querying a Binary Search Tree: retrieve information from the tree without modifying the tree.

Query operations of a binary search tree include Search, Minimum, Maximum, Successor, Predecessor, ...

All of the above operations take $O(h)$, where h is the height of the tree.

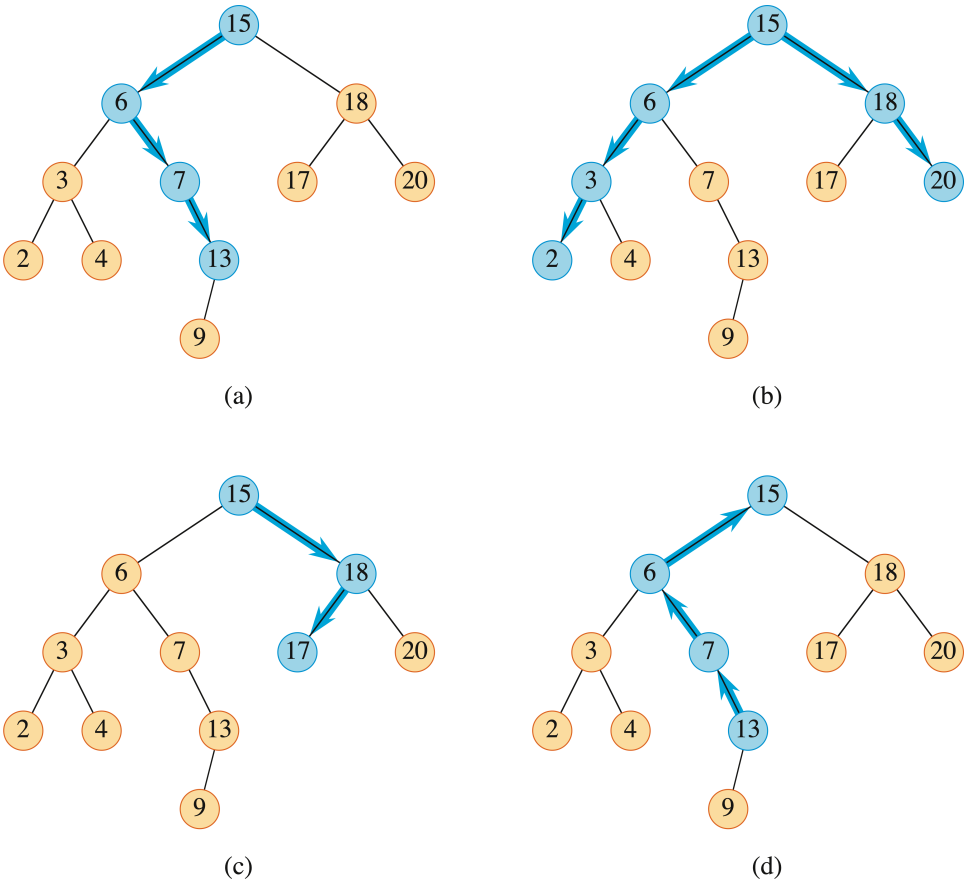


Figure (a) shows a search for key=13, (b) shows search for the min and max, (c) shows the search for the successor of 15, and (d) shows the search for the successor of 13.

```
TREE-SEARCH(x, k)
1. if x == NIL or k == x.key
2.    return x
3. if k < x.key
4.       return TREE-SEARCH(x.left, k)
5. else return TREE-SEARCH(x.right, k)
```

The nodes searched during the recursion form a path from the root downward. Thus, running time is $O(h)$.

```
ITERATIVE-TREE-SEARCH(x, k)
1. while x != NIL and k != x.key
2.       if k < x.key
3.            x = x.left
4.       else x = x.right
5. return x
```

```
TREE-MINIMUM(x)
1. while x.left != NIL
2.      x = x.left
3. return x
```

```
TREE-MAXIMUM(x)
1. while x.right != NIL
2.      x = x.right
3. return x
```

The successor of a node x is the node y, where

$$y = \begin{cases} \text{Minimum}(x.right) & \text{if } x.right \neq \text{NIL} \\ \text{The lowest ancestor of } x \text{ whose left child is also an ancestor of } x & \text{if } x.right = \text{NIL} \end{cases}$$

```
TREE-SUCCESSOR(x)
1. if x.right != NIL
2.    return TREE-MINIMUM(x.right)
3. else
4.    y = x.p
5.    while y != NIL and x == y.right
6.       x = y
7.       y = y.p
8.    return y
```

The worst-case runtime is $O(h)$, since we either follow a path downward (1st case) or a path upward (2nd case)

Recommended Exercises: 12.2-1, 12.2-2, 12.2-4, 12.2-5, 12.2-6.

Solve in class: 12.2-3.

Solution to Exercise 12.2-3: The predecessor of a node x is the node y, where

$$y = \begin{cases} \text{Maximum}(x.left) & \text{if } x.left \neq \text{NIL} \\ \text{The lowest ancestor of } x \text{ whose right child is also an ancestor of } x & \text{if } x.left = \text{NIL} \end{cases}$$

```
TREE-PREDECESSOR(x)
1. if x.left != NIL
2.    return TREE-MAXIMUM(x.left)
3. else
4.    y = x.p
5.    while y != NIL and x == y.left
6.       x = y
7.       y = y.p
8.    return y
```

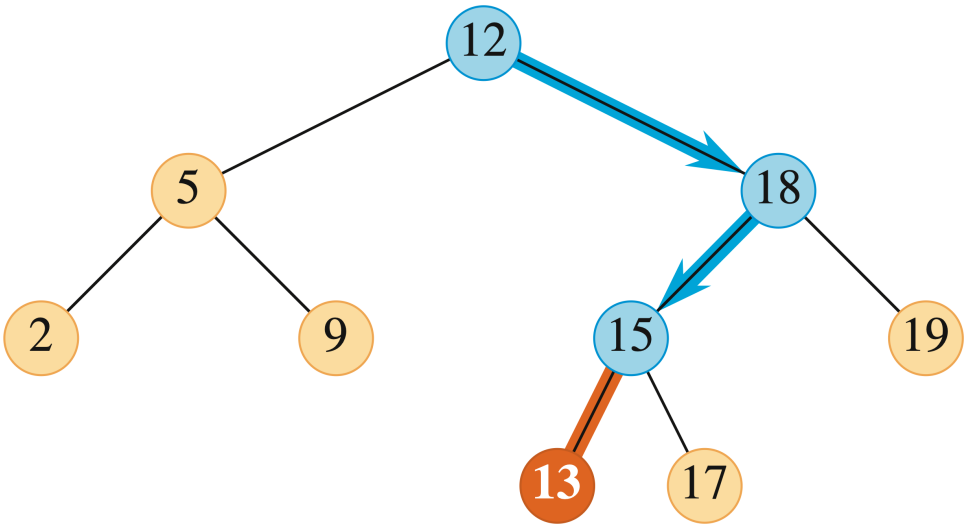
The worst-case runtime is $O(h)$, using same argument as for TREE-SUCCESSOR(x)

Insertion and Deletion

Insertion

TREE-INSERT(T, z): insert a node z into a binary search tree T, where $z.key = v$, $z.left = z.right = z.p = NIL$ (initially, but the pointers may change during the process of insertion)

TREE-INSERT always inserts a new node z as a leaf node.



```
TREE-INSERT(T, z)
// Insert node z, where z.key = v, z.left = NIL z.right = NIL
1.  x = T.root            // x keeps track of the path for insertion
2.  y = NIL               // y will track the parent of x
3.  while x != NIL
4.        y = x
5.        if z.key < x.key
6.             x = x.left
7.        else x = x.right
8.  z.p = y
9.  if y == NIL
10.        T.root = z
11. elseif z.key < y.key
12.           y.left = z
13. else y.right = z
```

Steps 3 - 7: find the position to insert the new node.

Steps 8 - 13: set the pointers to insert the new node.

Takes $O(h)$ worst-case runtime: trace downward from the root to a leaf to find the position to insert.

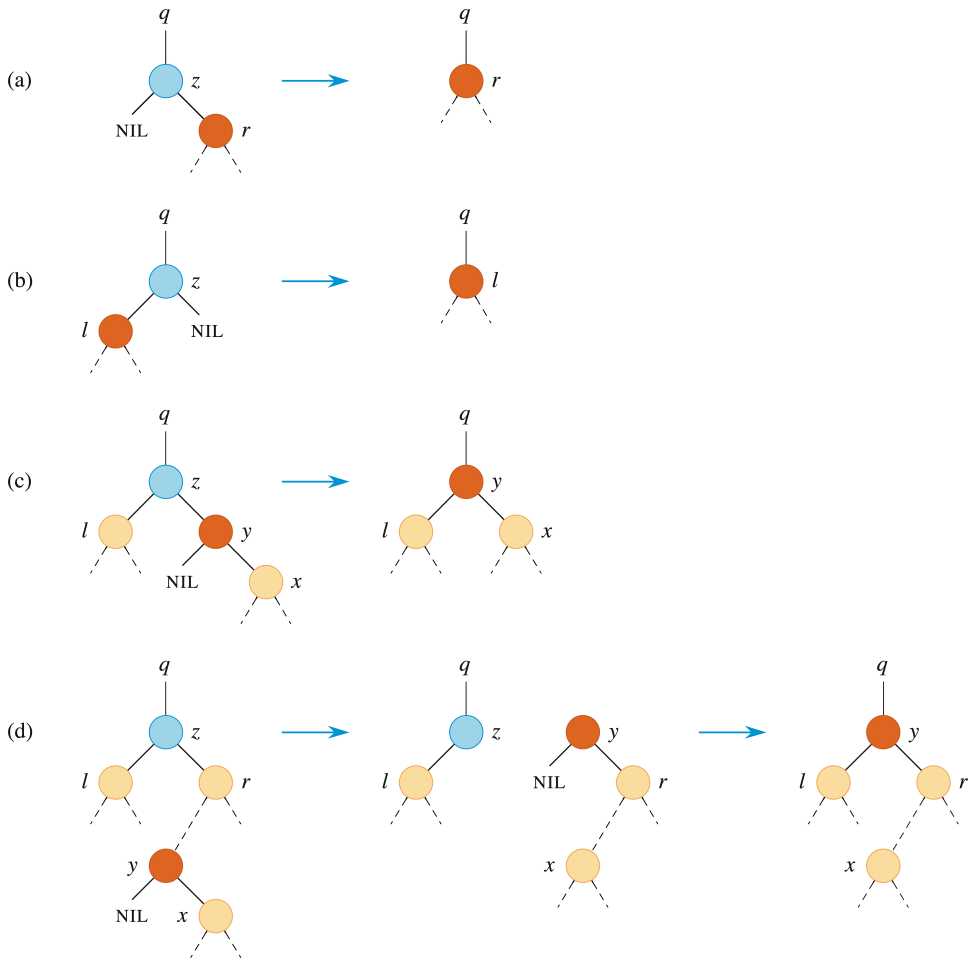
Give an example, using the animation in the references.

Questions:

- What kind of BST do we get if we insert n elements that are already sorted in ascending order? What is its height?
- What kind of BST do we get if we insert n elements that are already sorted in ascending order? What is its height?

Deletion

To delete a node z from a binary search tree, there are 4 cases to consider.



Case (a) If z has no left child, then replace z by its right child, which may or may not be NIL. This corresponds to part (a) in the figure.

Case (b) Otherwise, z has just a left child, we replace z by its left child. This corresponds to part (b) in the figure.

Case (c) and (d) Otherwise, z has both a left child and a right child. Find z's successor y, which lies in z's right subtree and has no left child. Splice node y out of its current location and replace z by y in the tree. How to do depends on the whether y is z's right child. This leads to the last two cases.

Case (c) If y is z's right child, then as in part (c) of the figure, replace z by y, leaving y's right child alone.

Case (d) Otherwise, y lies within z's right subtree but is not z's right child. In this case, as in part (d) of the figure, first replace y by its own right child, and then replace z by y.

```
TRANSPLANT(T, u, v)
//replace the subtree rooted at node u with the subtree rooted at node v
1.  if u.p == NIL
2.       T.root = v
3.  elseif u == u.p.left
4.       u.p.left = v
5.  else u.p.right = v
6.  if v != NIL
7.      v.p = u.p

TREE-DELETE(T, z)
1.  if z.left == NIL
2.      TRANSPLANT(T, z, z.right)
3.  elseif z.right == NIL
4.      TRANSPLANT(T, z, z.left)
5.  else y = TREE-MINIMUM(z.right)
6.       if y != z.right
7.          TRANSPLANT(T, y, y.right)
8.          y.right = z.right
9.          y.right.p = y
10.      TRANSPLANT(T, z, y)
11.      y.left = z.left
12.      y.left.p = y
```

Takes $O(h)$ worst-case runtime: case (a) or (b) take $\Theta(1)$, but case (c) and (d) takes $O(h)$.

Recommended Exercises: 12.3-1, 12.3-2, 12.3-3, 12.3-4, 12.3-5.

Solve in class: 12.3-3, 12.3-5.

Exercise 12.3-5: Is deletion commutative, in the senses that deleting x followed y gives us the same tree if we delete y followed by x? Argue why it is or give an counterexample.

Try deleting 5,3 and 3,5

```
 5
/ \
3  7
  /
 6
```

Randomly built binary search tree have an expected height of $O(\lg n)$, similar to the best-case. See below for an implementation (one simple, one generic) that runs an experiment to determine average height of randomly created binary search trees. This is an example of performance testing!

- See a simple implementation of a BST here: https://github.com/BoiseState/CS321-resources/tree/master/examples/binary-search-tree
- See a generic implementation of a BST here: https://github.com/BoiseState/CS321-resources/tree/master/examples/binary-search-tree-generic

But what if we have an unbalanced binary search tree (because it wasn't random...). How can we rebalance an existing binary search tree?

Recommended Exercise: We can do an inorder walk and then recursively build back a balanced tree. Develop pseudo-code for this procedure.

References

- TREE visualization and interactive explorer: https://visualgo.net/en/bst