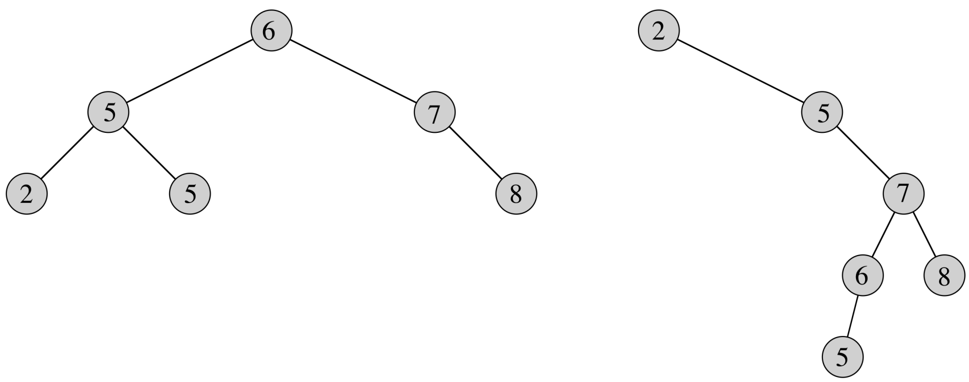
Chapter 12: Binary Search Trees

What is a Binary Search Trees?

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 $y.key > x.key$.



(a) (b)

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

```
INORDER-TREE-WALK(x)
if x != NIL
    INORDER-TREE-WALK(x.left)     // Theta(n)
    print x
    INORDER-TREE-WALK(x.right)
```

We can similarly do a **preorder** and **postorder** walk.

Recommended Exercises: 12.1-1, 12.1-2, 12.1-4. *Challenging:* 12.1-3

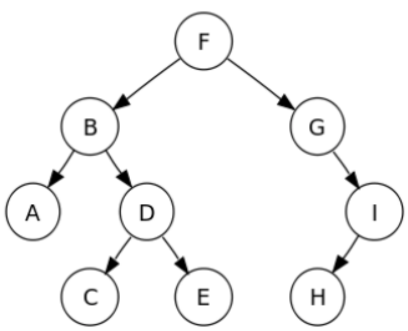
Other Recommended Exercises:

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

Solution 12.1-3:

```
ITERATIVE-INORDER-TREE-WALK(x)
initialize an empty stack S
while true
    //traverse to the leftmost leaf
    while x != NIL
        PUSH(S, x)
        x = x.left
    //if stack is empty, then we are done
    if EMPTY-STACK(S)
         return
    //pop the top element, print it and then add nodes
    //in the right subtree to the stack
    x = POP(S)
    print x.data
    x = x.right
```

Example:



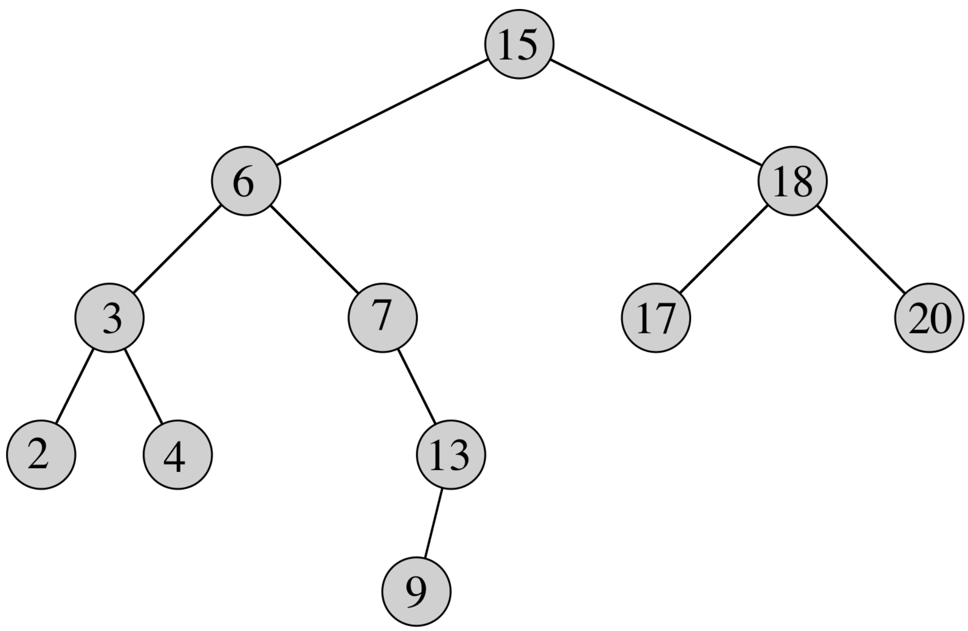
Step	Output	Stack
0		F,B,A
1	A	F,B
2	B	F,D,C
3	C	F,D
4	D	F,E
5	E	F
6	F	G
7	G	I,H
8	H	I
9	I	

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.




```
TREE-SEARCH(x, k)
1. if x == null 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                    // O(h)
3. return x
```

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

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

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

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

// O(h): since we either follow a path downward (1st case) or
//       a path upward (2nd case)
```

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

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

Example:

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

// O(h): same argument as TREE-Successor(x)
```

Recommended Exercises: 12.2-1, 12.2-2, 12.2-3, 12.2-4, 12.2-5, 12.2-6.

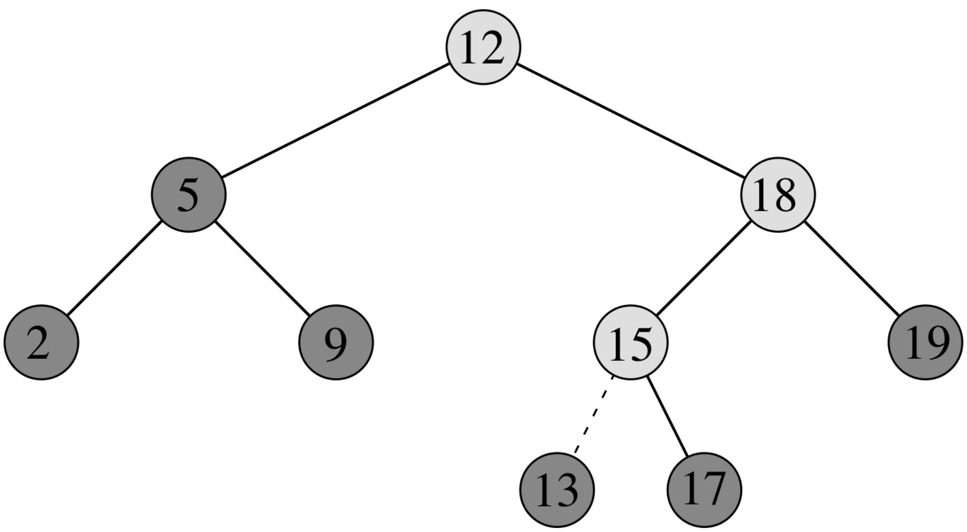
Solve in class: 12.2-3, 12.2-6.

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.

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
    y = NIL                    // y will track the parent of x
    x = root(T)                // x keeps track of the path for insertion
    while x != NIL
          y = x
          if z.key < x.key
               x = x.left
          else x = x.right
    z.p = y
    if y == NIL
           T.root = z
    elseif z.key < y.key
              y.left = z
         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 $O(h)$ time: trace downward from the root to a leaf to find the position to insert.

Give an example, using the animation in the references.

Questions:

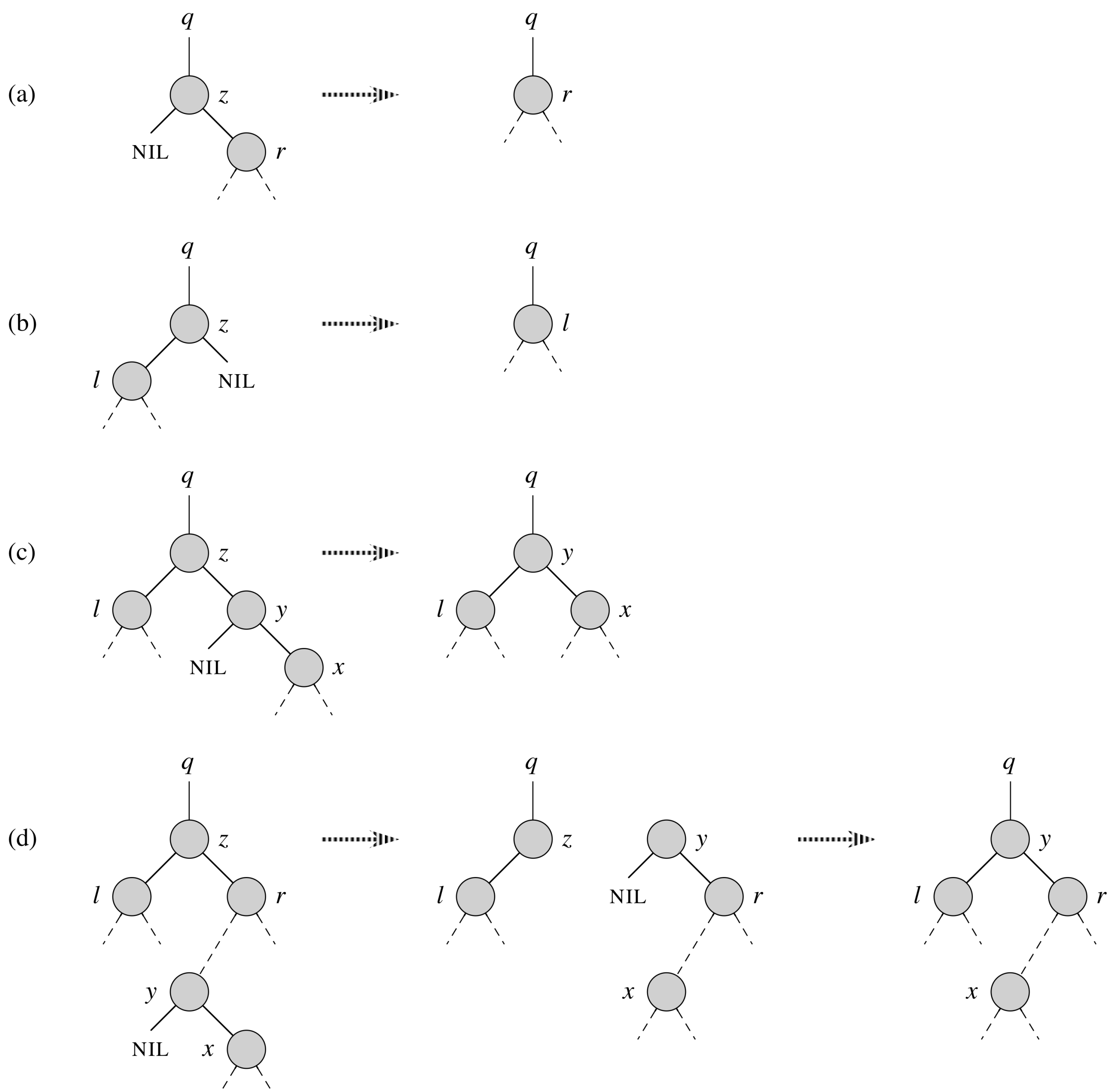
- What kind of BST do we get if we insert n elements that are already sorted in ascending order? What is its height?
- What kind of BST do we get if we insert n elements that are already sorted in ascending order? What is its height?

Deletion

To delete a node z from a binary search tree, there are 3 cases to consider.

1. If z does not have any children, then modify the parent $z.p$: replace z with NIL as $z.p$'s child.
2. If z has only one child, then take out z by making a new link between its child and its parent.
3. If z has two children, then take out z's successor y (y has no left child) and copy the contents of y to z.

These get refined into four cases in the code, shown visually on the next page.


(a)
q
z
NIL
r
q
r
(b)
q
z
l
NIL
q
l
(c)
q
z
l
y
NIL
x
q
y
l
x
(d)
q
z
l
r
y
NIL
x
q
z
l
y
NIL
r
x
q
y
l
r
x

```
TRANSPLANT(T, u, v)
//replace the subtree rooted at node u with the subtree rooted at node v
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v != NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
     if y.p != z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
     TRANSPLANT(T, z, y)
     y.left = z.left
     y.left.p = y
```

Take $O(h)$ time: case 1 or 2 take $\Theta(1)$, but case 3 takes $O(h)$.

Recommended Exercises: 12.3-1, 12.3-2, 12.3-3, 12.3-4, 12.3-6.

Solve in class: 12.3-3, 12.3-6

Randomly built binary search tree have an expected height of $O(\lg n)$, similar to the best-case.

How can we balance 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