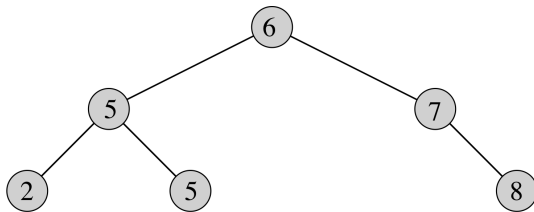


# Chapter 12: Binary Search Trees

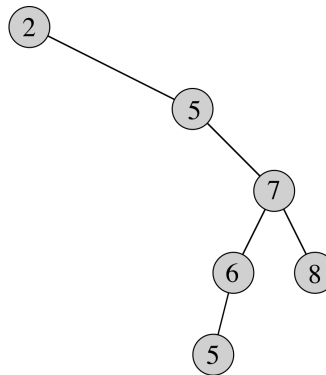
## What is a Binary Search Trees?

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( $x$ )

1. if  $x \neq \text{NIL}$
2.     INORDER-TREE-WALK( $x.\text{left}$ )     // Theta( $n$ )
3.     print  $x$
4.     INORDER-TREE-WALK( $x.\text{right}$ )

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:**

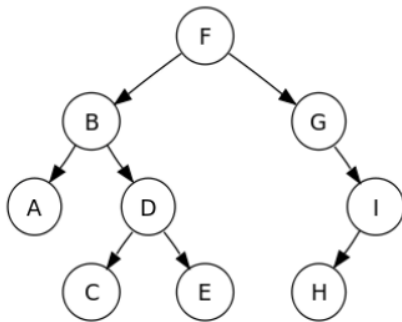
1. Write a recursive procedure for counting the number of nodes in a binary search tree.
2. Write a recursive procedure for finding the height of a binary search tree.

### Solution 12.1-3:

ITERATIVE-INORDER-TREE-WALK(x)

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

Example:



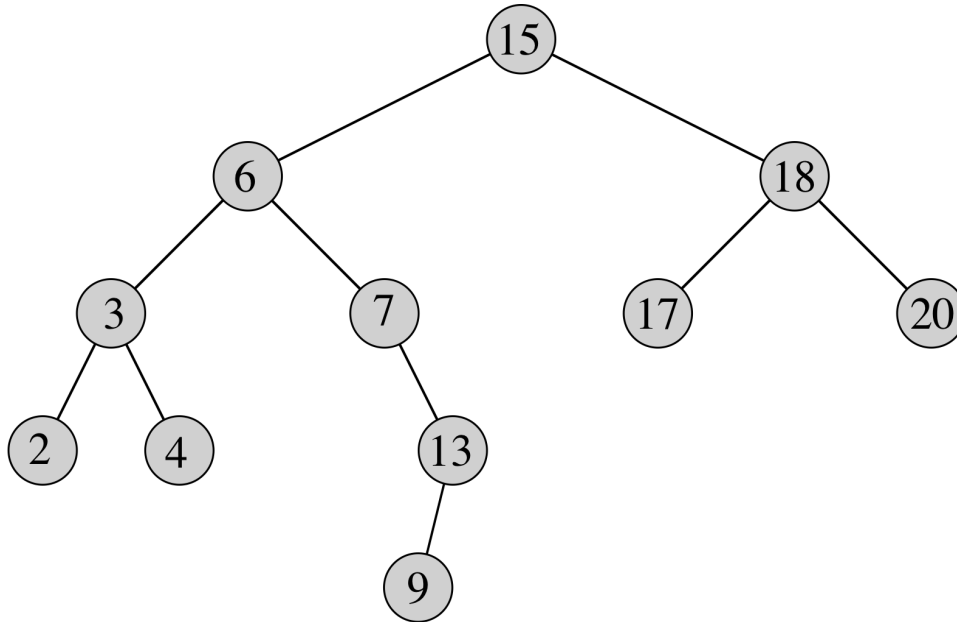
Step	Output	Stack
0		F,B,A
1	A	F,B
2	B	F,D,C
3	C	F,D
4	D	F,E
5	E	F
6	F	G
7	G	I,H
8	H	I
9	I	

## 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  $x == \text{null}$  or  $k == x.\text{key}$ 
2.   return  $x$ 
3. if  $k < x.\text{key}$ 
4.   return TREE-SEARCH( $x.\text{left}, k$ )
5. else return TREE-SEARCH( $x.\text{right}, k$ )
// 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  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2.   if  $k < x.\text{key}$ 
3.      $x = x.\text{left}$ 
4.   else  $x = x.\text{right}$ 
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

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

```

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

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

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.

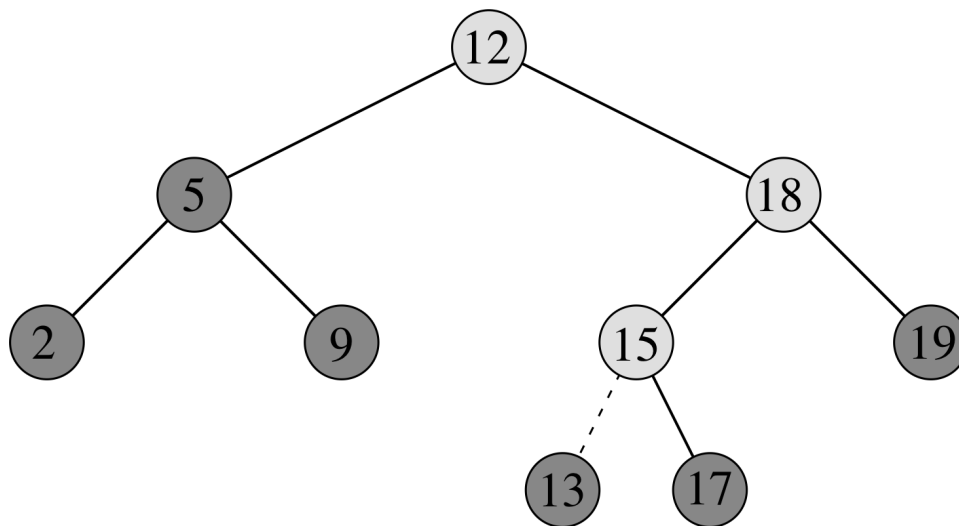
**Solve in class:** 12.2-3, 12.2-6.

## Insertion and Deletion

### Insertion

$\text{TREE-INSERT}(T, z)$ : insert a node  $z$  into a binary search tree  $T$ , where  $z.\text{key} = v$ ,  $z.\text{left} = z.\text{right} = z.p = \text{NIL}$  initially.

$\text{TREE-INSERT}$  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.  y = NIL                // y will track the parent of x
2.  x = root(T)            // x keeps track of the path for insertion
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  $O(h)$  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:

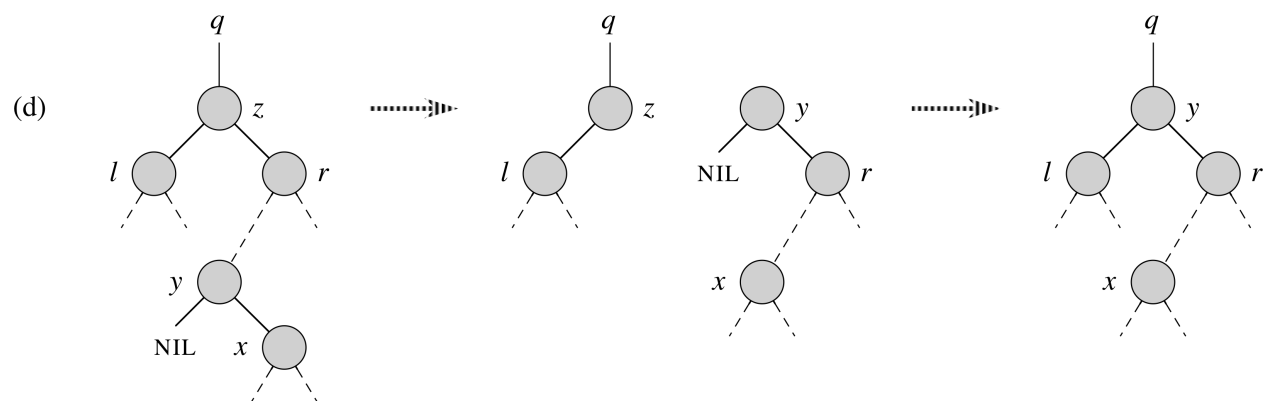
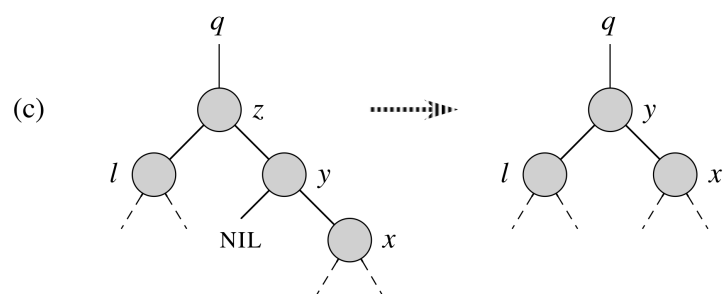
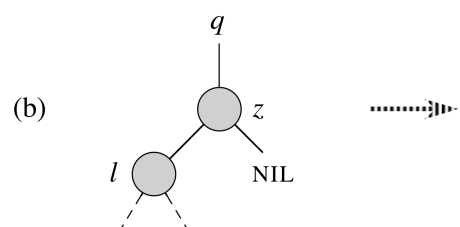
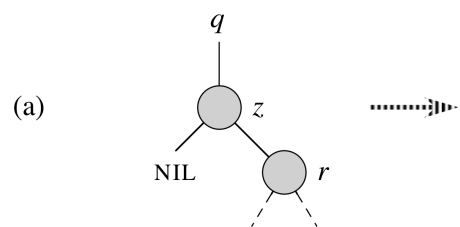
- What kind of BST do we get if we insert  $n$  elements that are already sorted in ascending order? What is its height?
- What kind of BST do we get if we insert  $n$  elements that are already sorted in descending order? What is its height?

## 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.



```

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.p != z
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

```

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>