Graph Theory and Search Trees

Lecture VI for course in Data Structures & Algorithms for AI

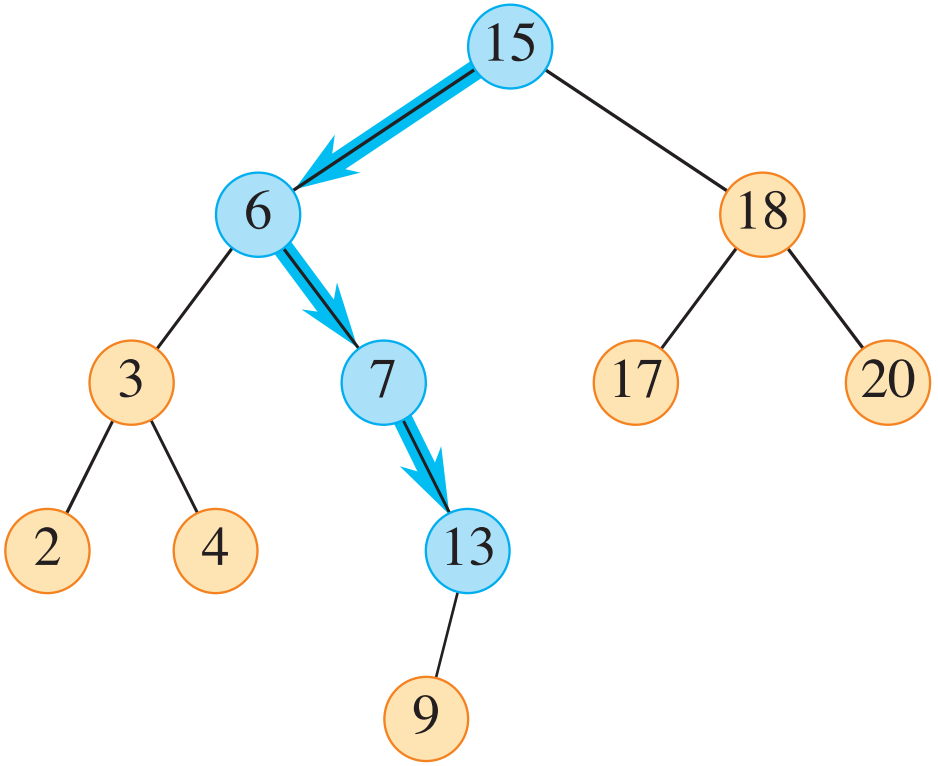
Lectures by Hannah Santa Cruz Baur.
Based on *Introduction to Algorithms,* Cormen et al. and material by Rafaella Mulas.

Homework (due with Assignment 3)

- Prove that in a heap, with nodes indexed by $1, \ldots, n$, starting with the root, the node with largest index to have no children, has index $\lfloor n/2 \rfloor$.

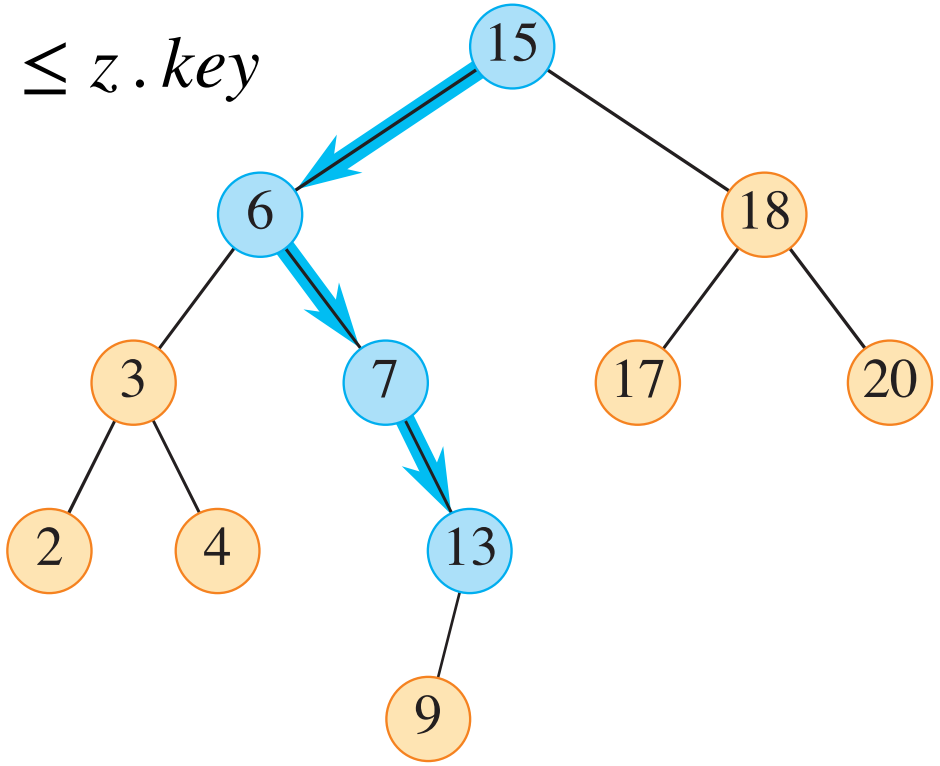
Binary Search Trees

- A **Binary Search Tree** is a data structure which can be viewed as a binary tree, satisfying a key property:



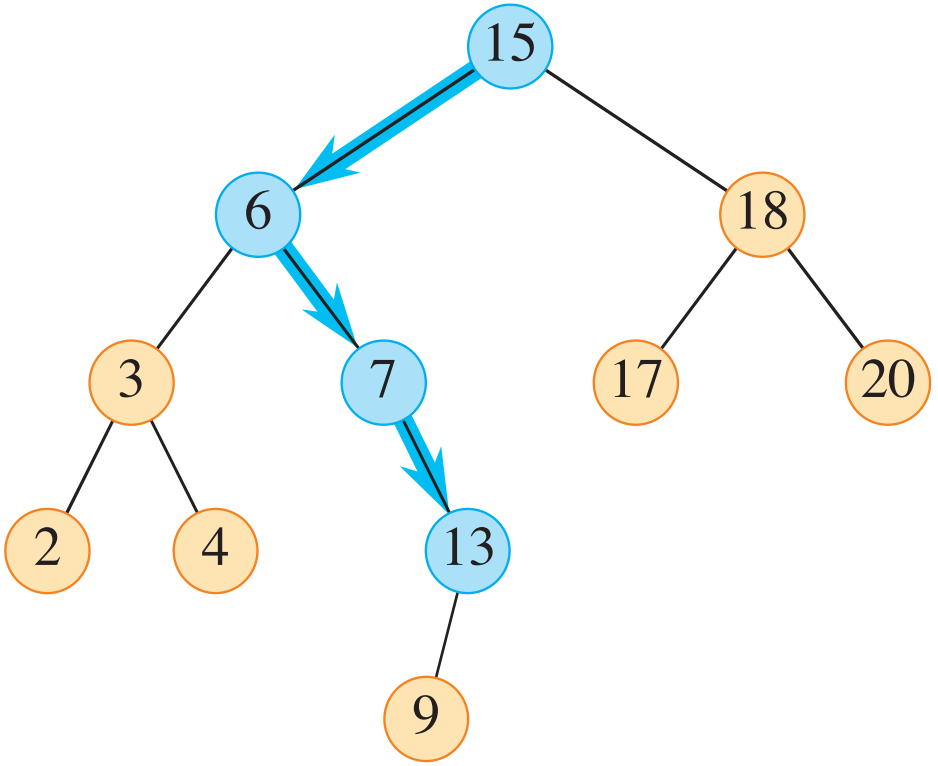
Binary Search Trees

- A **Binary Search Tree** is a data structure which can be viewed as a binary tree, satisfying a key property:
- For any node x, and nodes y and z in it's left and right subtrees respectively, it holds that:
- $y\,.\,key \leq x\,.\,key \leq z\,.\,key$



Binary Search Trees

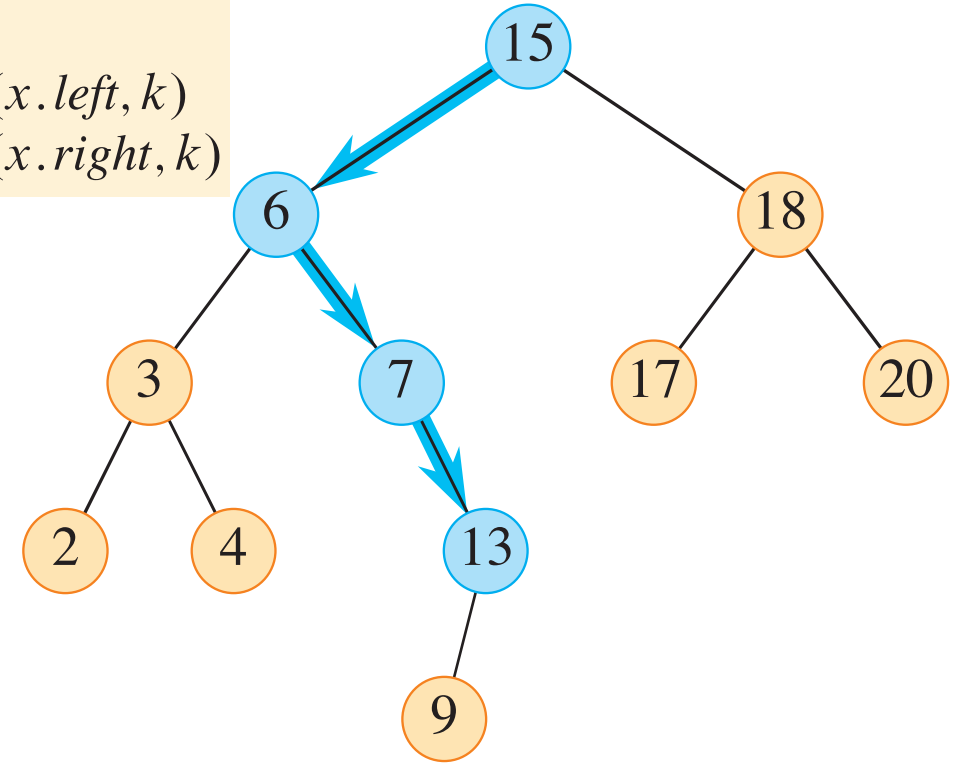
- The *binary search tree property* enables for an efficient search algorithm.



Binary Search Trees

- The *binary search tree property* enables for an efficient search algorithm.

```
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)
```



Binary Search Trees

- The *binary search tree property* enables for an efficient search algorithm.

```
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)
```

Which of the following sequences could not be the result of searching for 363 in a binary search tree?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

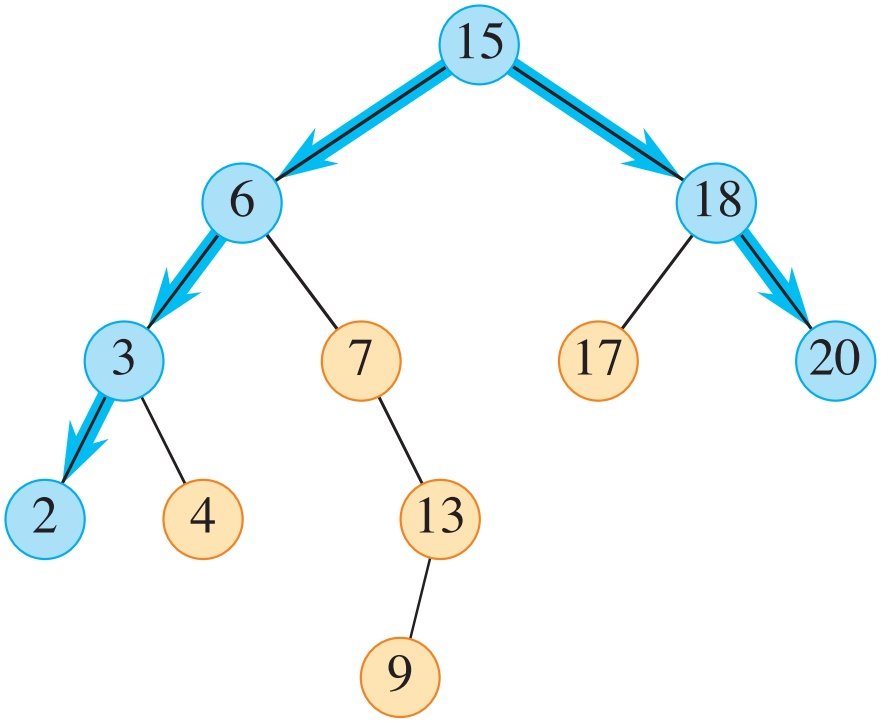
c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

Binary Search Trees

- The *binary search tree property* also enables for efficient minimum and maximum search algorithms.

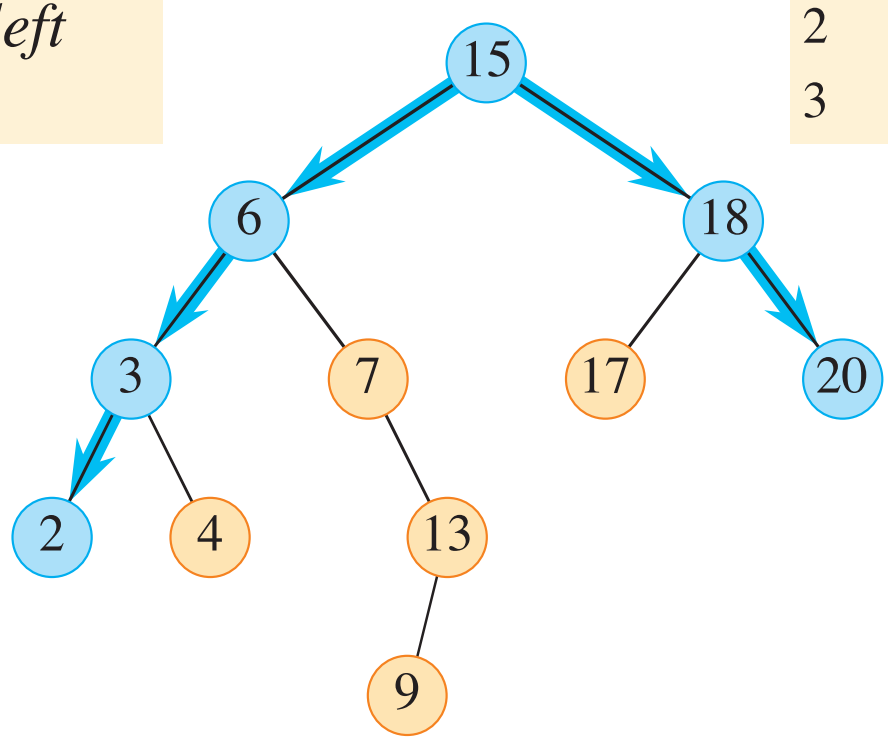


Binary Search Trees

- The *binary search tree property* also enables for efficient minimum and maximum search algorithms.

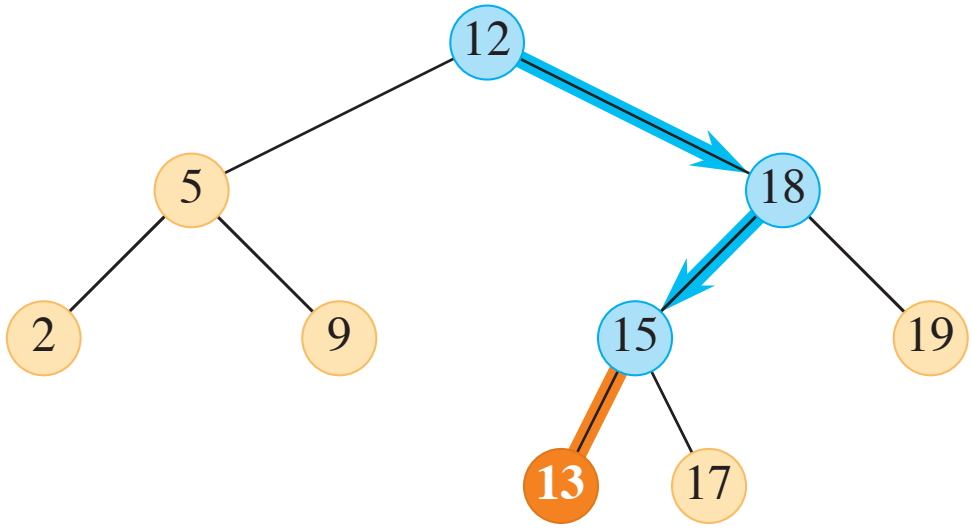
```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

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



Binary Search Trees

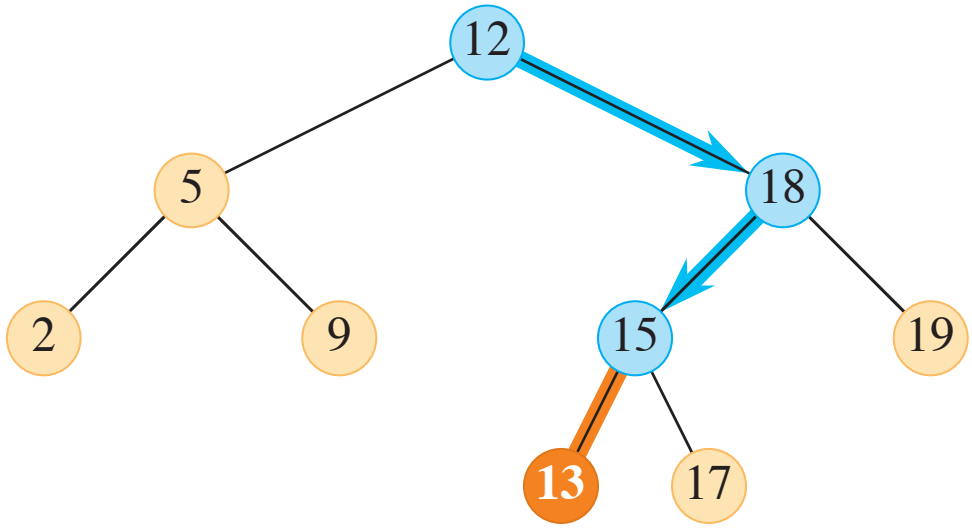
- To preserve the *binary search tree property,* the insertion operation adds new elements as leaves.



Binary Search Trees

- To preserve the *binary search tree property,* the insertion operation adds new elements as leaves.

```
TREE-INSERT(T, z)
 1   x = T.root
 2   y = NIL
 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
```

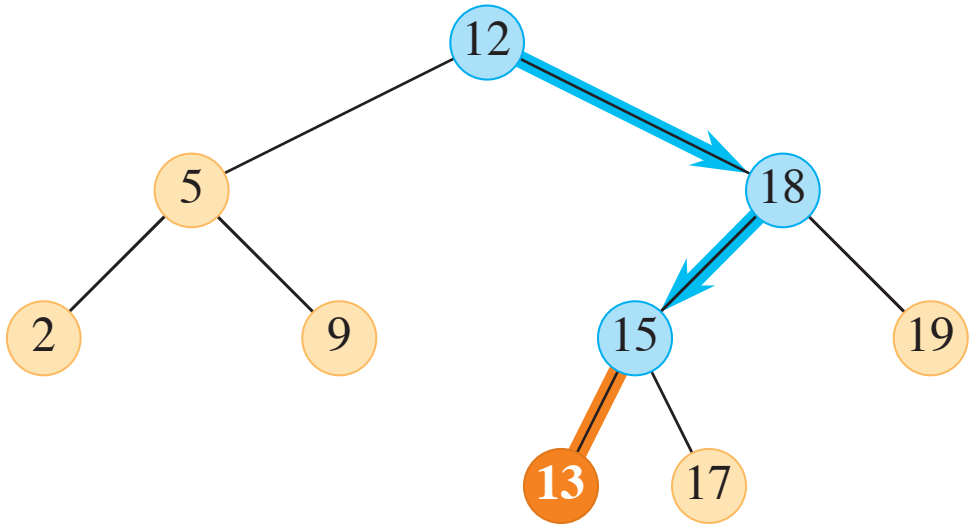


Binary Search Trees

- To preserve the *binary search tree property,* the insertion operation adds new elements as leaves.

```
TREE-INSERT(T, z)
 1   x = T.root
 2   y = NIL
 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
```

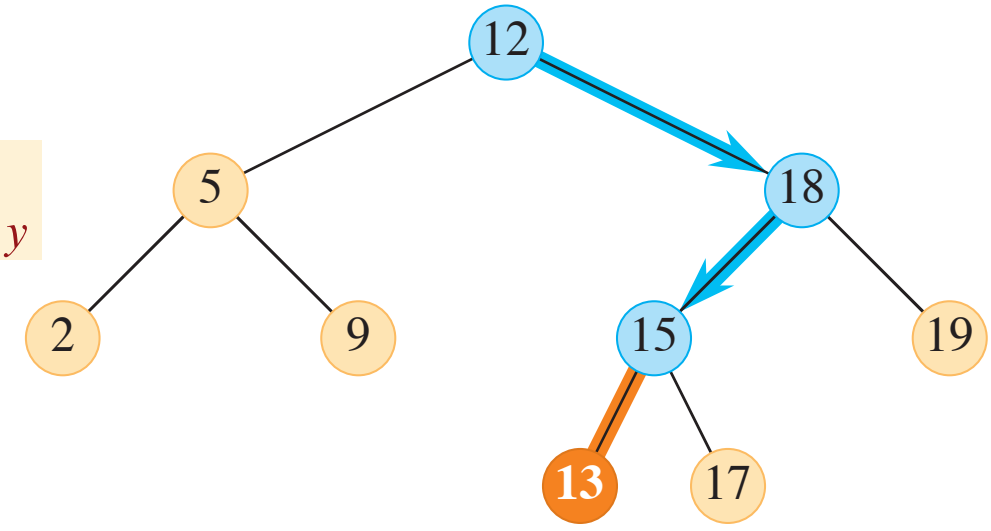
// node being compared with z
// y will be parent of z
// descend until reaching a leaf



Binary Search Trees

- To preserve the *binary search tree property,* the insertion operation adds new elements as leaves.

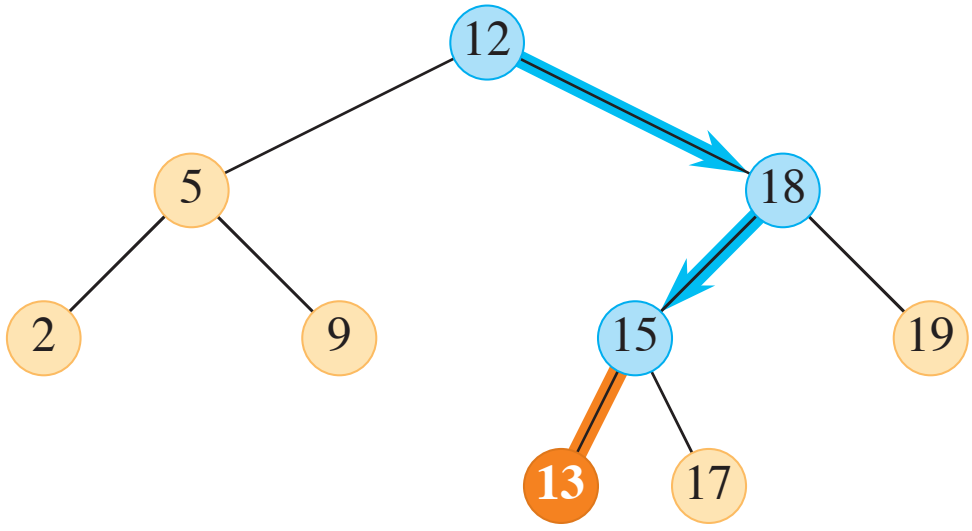
```
TREE-INSERT(T, z)
 1   x = T.root                  // node being compared with z
 2   y = NIL                     // y will be parent of z
 3   while x ≠ NIL               // descend until reaching a leaf
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y                     // found the location
 9   if y == NIL                 —insert z with parent y
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```



Binary Search Trees

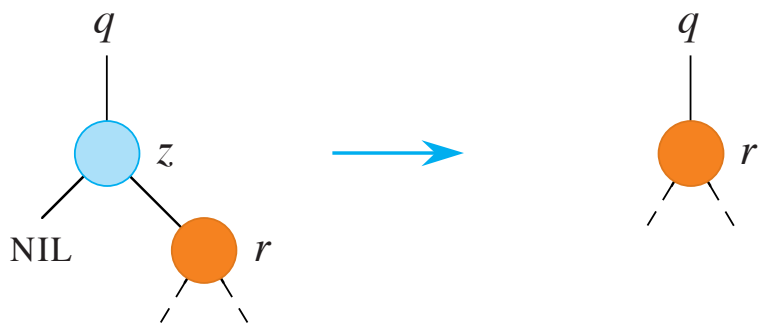
- To preserve the *binary search tree property,* the insertion operation adds new elements as leaves.

```
TREE-INSERT(T, z)
 1   x = T.root
 2   y = NIL
 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
```



Binary Search Trees

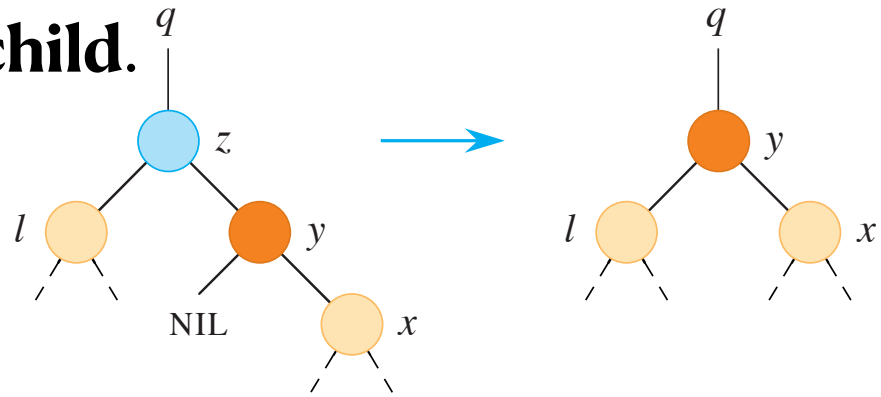
- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.
- To delete a node z, we distinguish four cases:
- If z has **no left child**, replace z with it's right child (could be NIL).



Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.
- To delete a node z, we distinguish four cases:
- If z has a **right & left child**, and it's **right child, has no left child**.

  Swap z with it's right child.



Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.
- To delete a node z, we distinguish four cases:
- If z has a **right & left child**, and it's **right child, has a left child,** find the **successor** of z:

  the node with smallest key, larger than $z.key$.

```
TREE-SUCCESSOR*(x)
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
    return y
```

Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.
- To delete a node z, we distinguish four cases:
- If z has a **right & left child**, and it's **right child, has a left child,** find the **successor** of z:

  the node with smallest key, larger than $z.key$.

```
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
```

Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.
- To delete a node z, we distinguish four cases:
- If z has a **right & left child**, and it's **right child, has a left child,** find the **successor** of z:

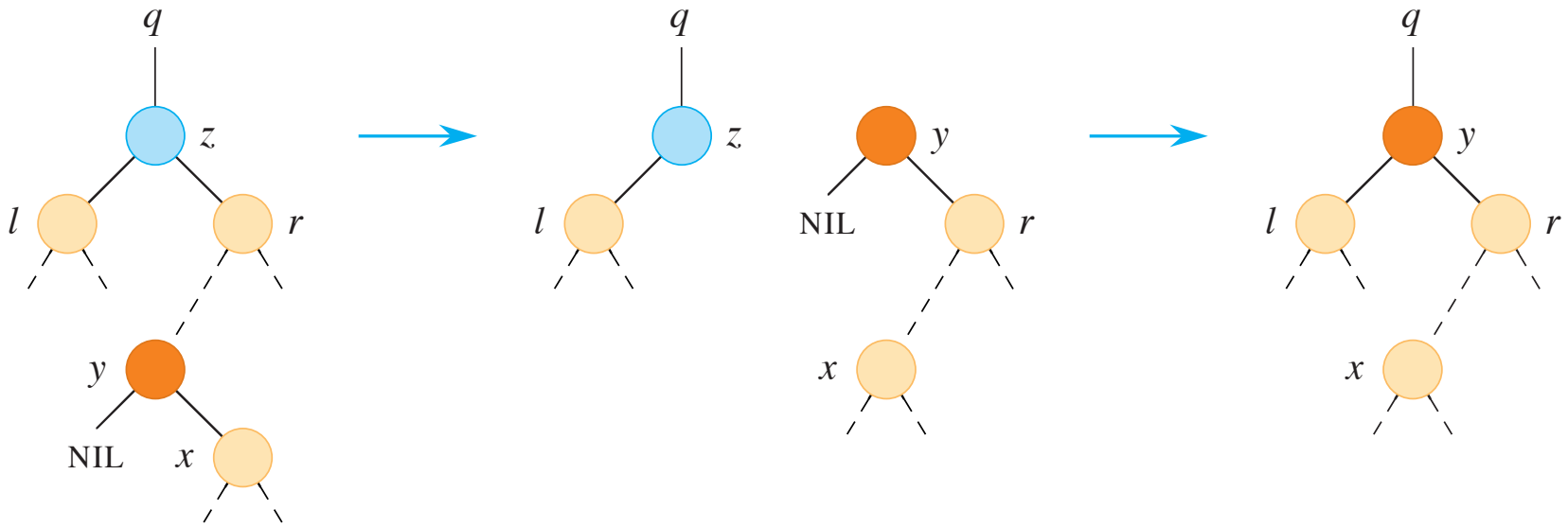
  the node with smallest key, larger than $z.key$.

  Notice the successor has no left child.

Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.
- To delete a node z, we distinguish four cases:
- If z has a **right & left child**, and it's **right child, has a left child,** find the **successor**, y, of z:

  Put the right child of y in y's position, and y in z's position.



Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)          // replace z by its right child
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)           // replace z by its left child
 5  else y = TREE-MINIMUM(z.right)         // y is z's successor
 6      if y ≠ z.right                     // is y farther down the tree?
 7          TRANSPLANT(T, y, y.right)      // replace y by its right child
 8          y.right = z.right              // z's right child becomes
 9          y.right.p = y                  //       y's right child
10      TRANSPLANT(T, z, y)                // replace z by its successor y
11      y.left = z.left                    // and give z's left child to y,
12      y.left.p = y                       //       which had no left child
```

Binary Search Trees

- To preserve the *binary search tree property,* the deletion operation needs to significantly alter the data structure.

```
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
```

```
TRANSPLANT(T, u, 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
```

Homework (due with Assignment 3)

- Is the operation of deletion in a binary search tree"commutative"?

  That is, would deleting x and then y, yield the same tree as deleting y and then x?

  Argue why it is commutative, or give a counterexample.

- Suppose that you construct a binary search tree by repeatedly inserting distinct values into the tree.

  Argue that the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.