Binary Search Trees

Textbook, Chapter 13, Sections 13.2 and 13.3

For Section 13.1, refer to Handout on Trees

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

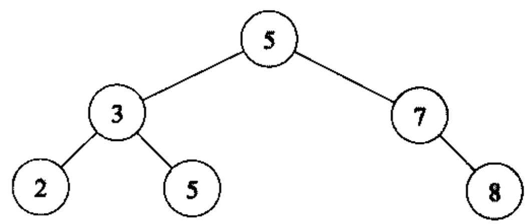
Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

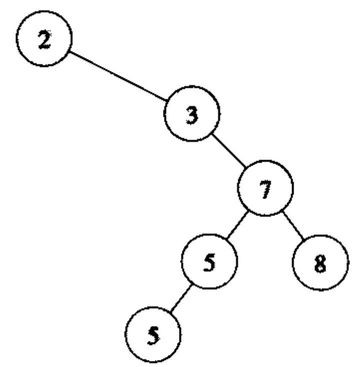
`choueiry@cse.unl.edu`, Tel: (402)472-5444

Binary search trees

is a binary tree that satisfies the binary-search-tree property



(a)



(b)

For any node x,

- 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 $key[x] \leq key[y]$

In-order tree traversal prints the key in a sorted order

Operations

Given a binary-search tree, we may want to:

1. **Queries:** Search, Minimum, Maximum, Successor, Predecessor
2. **Modifications:** Insertion, Deletion

$\longrightarrow$ All operations in $O(h)$, h height of the tree

Searching: recursive

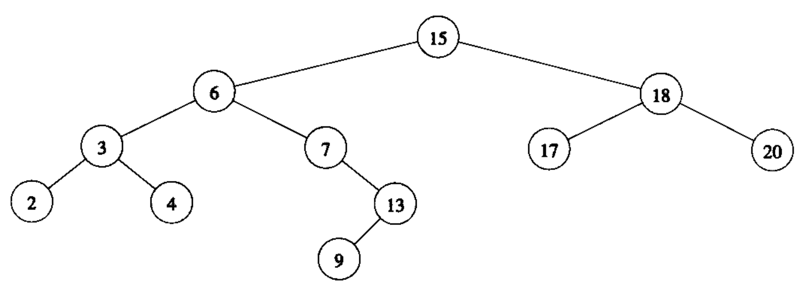
Input: pointer to the root, k

Output: pointers to node whose key is k, `Nil` otherwise

Tree-Search(x, k)

If $x =$ `Nil` or $k = key[x]$
  then return x
If $k < key[x]$
  then return Tree-Search($left[x], k$)
  then return Tree-Search($right[x], k$)

Begins at the root, traces a path downward



Searching: Iterative

Recursive, can easily be made iterative

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

Minimum/Maximum

- Minimum: follow left
- Maximum: follow right

Tree-Minimum(x)
While $left[x] \neq$ `Nil`
  do $x \leftarrow left[x]$
return x

Tree-Maximum(x)
while $right[x] \neq$ NIL
  do $x \leftarrow right[x]$
return x

Correctness guaranteed by the binary-search-tree property
Complexity: $O(h)$

Minimum (Maximum): follow left

Tree-Minimum(x)
While $left[x] \neq$ `Nil`
    do $x \leftarrow left[x]$
return x

Correctness:

x **has no left tree:** since every key in the right subtree has a value at least as large as $key[x]$
then, return x

x **has a left tree:** since no key in the right subtree has a value smaller than $key[x]$
and every key in the left subtree has a value not larger than $key[x]$

So, the minimum should be found in the subtree rooted at $left[x]$

Successor/Predecessor

Find successor/predecessor in the <u>sorted order</u>
(sorted order is determined by the in-order tree walk)

Assuming, all keys are distinct, and given a node x

- successor of x is the smallest key that is greater than the key of x
- predecessor of x is the greatest key that is smaller than the key of x

Successor, predecessor can be determined **without ever comparing keys**!

Successor

Input: node x

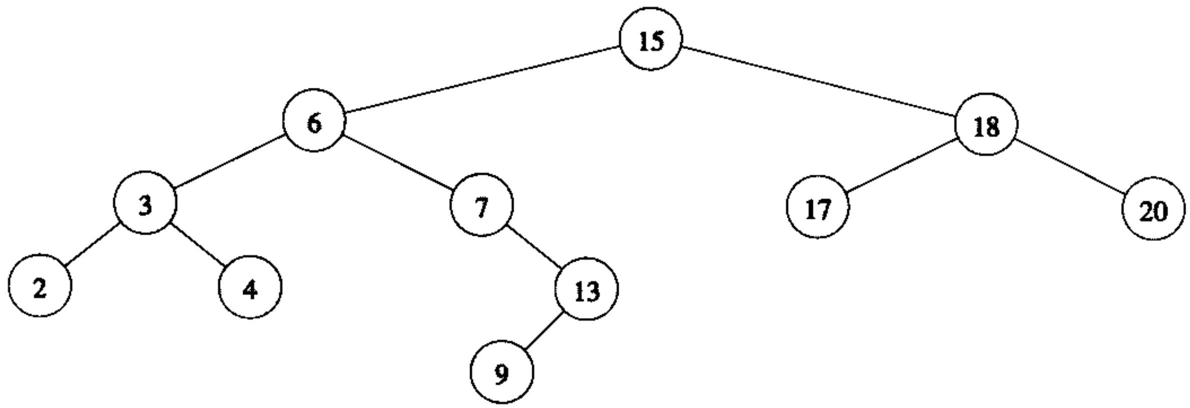
Output: its successor if it exists, `Nil` otherwise

(i.e., x has the largest key)

```
TREE-SUCCESSOR(x)
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
```

2 cases:

1. if the right subtree of x is not empty, then successor is..
2. otherwise, and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x



successor(15)=17, successor(6)=7, successor(7)=9, etc.
successor(13)=15

Complexity: either going down, or up the tree, $O(h)$

Important note

If a node has two children:

- Its successor is in its right tree
- its predecessor is in it left tree

Further

- Its successor cannot have a left child
  such a child would come between the node and its successor
  it comes after the node: it is in the node's right subtree
  it comes before the successor: it is in the left subtree
  not possible!!
- Its predecessor cannot have a right child

Insertion/Deletion

Modify the tree
Careful for preserving binary-search-tree property

Insertion: easy

Deletion: more intricate

Insertion $O(h)$

Input: a <u>node</u> z, $key[z] = v$,
$left[z] = right[z] =$`Nil`

Output: T, some fields of z are modified, z inserted in correct position

```
TREE-INSERT(T, z)
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
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```

begins at root, traces a path downward
x traces the path, y follows (maintains parent of x)
Pointers go left or right depending on whether how keys of x and z compare
Until x is `Nil`, this is where we want to put z, as a child of y

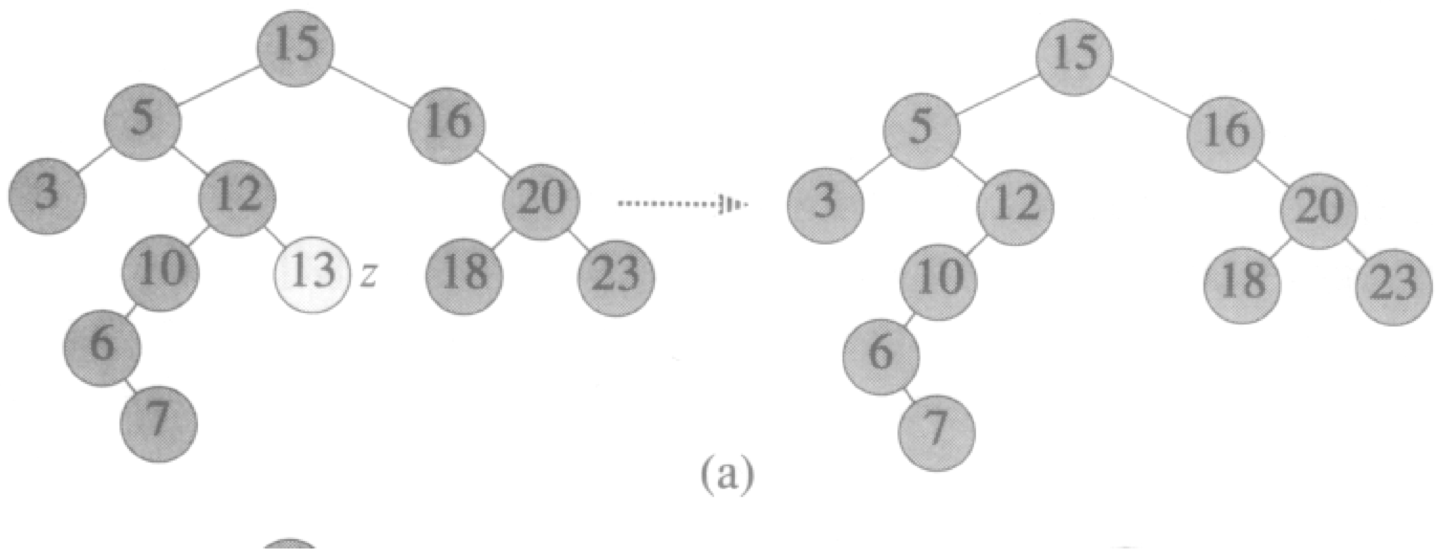
Deletion

Input: a point to node
Output: modified tree

Considers three cases:

1. z has no children
2. z has a single child
3. z has 2 children

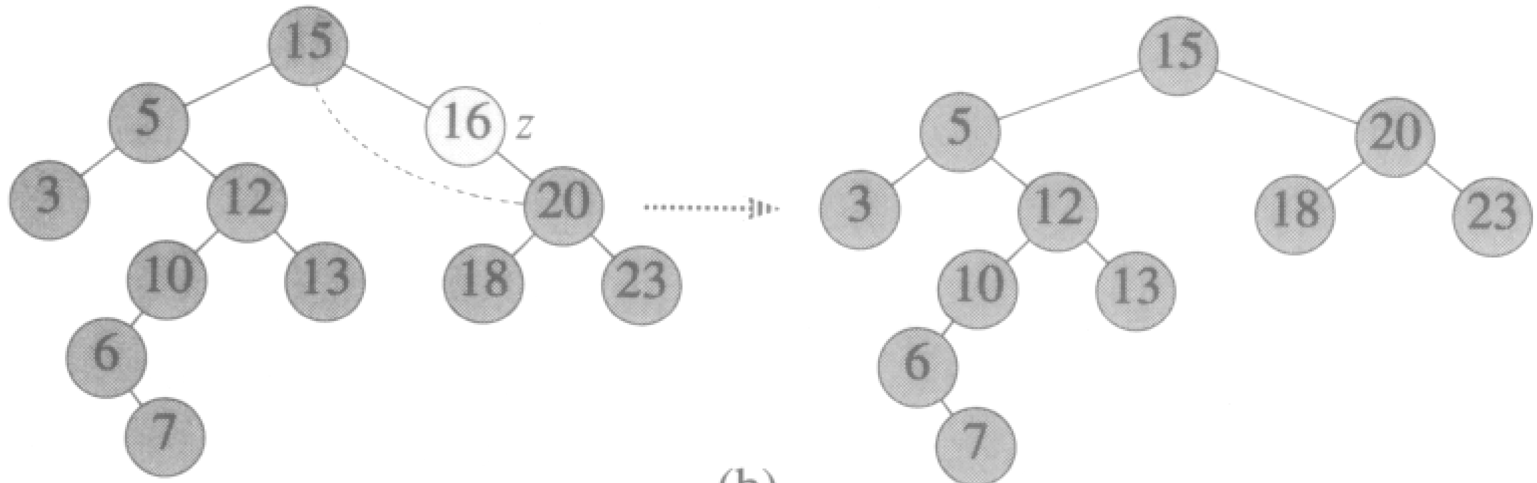
z **has no children**: 13



(a)

Modify parent, $p[z]$, to replace z with `Nil`

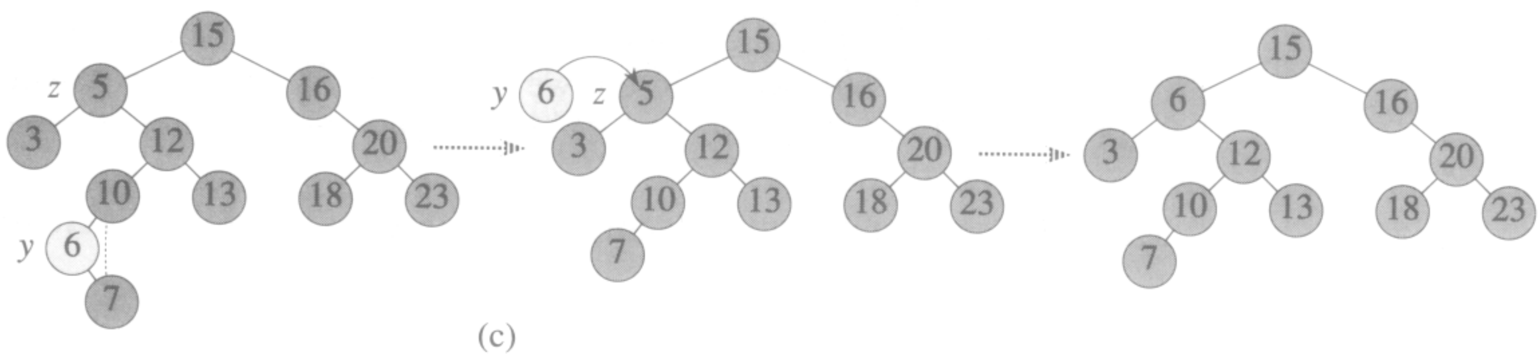
z **has a single child**: 16



(b)

Splice out z by making a new link between its parent and its child

z **has two children**: 5



(c)

Splice out $successor(z) = y$ (y cannot have a left child)
replace the content of z with the contents of y

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2      then y ← z
 3      else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5      then x ← left[y]
 6      else x ← right[y]
 7  if x ≠ NIL
 8      then p[x] ← p[y]
 9  if p[y] = NIL
10      then root[T] ← x
11      else if y = left[p[y]]
12              then left[p[y]] ← x
13              else right[p[y]] ← x
14  if y ≠ z
15      then key[z] ← key[y]
16              ▷ If y has other fields, copy them, too.
17  return y
```

$1 \ldots 3$ determines a node y to splice out ($y = z$ or $y = successor(z)$)

$4 \ldots 6$ x is set to non-nil child of y (or to Nil)

$7 \ldots 13$ splice out y by modifying pointers in $p[y]$ and x

$14 \ldots 16$ move the contents of z from y to z

17: return y so it can be recycled

Complexity: $O(h)$