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**

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

→ 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 \( x = \text{Nil} \) or \( k = \text{key}[x] \)

then return \( x \)

If \( k < \text{key}[x] \)

then return Tree-Search\((\text{left}[x], k)\)

then return Tree-Search\((\text{right}[x], k)\)

Begins at the root, traces a path downward
Searching: Iterative
Recursive, can easily be made iterative

\[
\text{ITERATIVE-TREE-SEARCH}(x, k)\\
\begin{align*}
1 & \text{ while } x \neq \text{NIL and } k \neq \text{key}[x] \\
2 & \quad \text{do if } k < \text{key}[x] \\
3 & \quad \text{then } x \leftarrow \text{left}[x] \\
4 & \quad \text{else } x \leftarrow \text{right}[x] \\
5 & \text{ return } x
\end{align*}
\]
Minimum/Maximum

- Minimum: follow left
- Maximum: follow right

**Tree-Minimum(x)**

```plaintext
While left[x] ≠ Nil
    do x ← left[x]

return x
```

**Tree-Maximum(x)**

```
1 while right[x] ≠ NIL
2    do x ← right[x]
3    return x
```

**Correctness** guaranteed by the binary-search-tree property

**Complexity:** $O(h)$
**Minimum (Maximum):** follow left

**Tree-Minimum(x)**

While $left[x] \neq \text{Nil}$
  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 sorted order
(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)

$$\text{Tree-Successor}(x)$$

1. if $\text{right}[x] \neq \text{NIL}$
2. then return $\text{Tree-Minimum}(\text{right}[x])$
3. $y \leftarrow p[x]$
4. while $y \neq \text{NIL}$ and $x = \text{right}[y]$
5. do $x \leftarrow y$
6. $y \leftarrow p[y]$
7. 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 its 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

Input: a node \( z \), \( key[z] = v \),
\( left[z] = right[z] = \text{Nil} \)

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

\[
\text{Tree-Insert}(T, z)
\]

1. \( y \leftarrow \text{NIL} \)
2. \( x \leftarrow \text{root}[T] \)
3. while \( x \neq \text{NIL} \) do \( y \leftarrow x \)
4. if \( key[z] < key[x] \) then \( x \leftarrow \text{left}[x] \)
5. else \( x \leftarrow \text{right}[x] \)
6. \( p[z] \leftarrow y \)
7. if \( y = \text{NIL} \) then \( \text{root}[T] \leftarrow z \)
8. else if \( key[z] < key[y] \) then \( \text{left}[y] \leftarrow z \)
9. else \( \text{right}[y] \leftarrow z \)

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 \( \text{NIL} \), 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

Modify parent, $p[z]$, to replace $z$ with Nil
$z$ has a single child: 16

Splice out $z$ by making a new link between its parent and its child
$z$ has two children: 5

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...3 determines a node y to splice out (y = z or y = successor(z))
4...6 x is set to non-nil child of y (or to Nil)
7...13 splice out y by modifying pointers in p[y] and x
14...16 move the contents of z from y to z
17: return y so it can be recycled

Complexity: O(h)