12.1 Binary Search Trees

A binary search tree (BST) is a BST for which each node contains fields:

\[
\text{key} \times 1, \text{left} \times 1, \text{right} \times 1, \text{title} (\text{+satellite}) \text{data}
\]

and satisfies the

**Binary Search Tree Properties**: for all nodes \(x\):

- if \(y\) is in the left subtree of \(x\), then \(\text{key}[y] \leq \text{key}[x]\).
- if \(y\) is in the right subtree of \(x\), then \(\text{key}[x] \leq \text{key}[y]\).

**Ex**: \(\text{root}[1] - 4\)

\(\text{root}[1] = \text{null}\) indicates an empty tree.
The fact property makes it possible to print out keys in ascending order.

InOrderTreeWalk(x)
1. if x ≠ nil
2. InOrderTreeWalk(left[x])
3. Print key[x]
4. InOrderTreeWalk(right[x])

To print full tree, call:

InOrderTreeWalk(root[T])

Run time: \( O(n) \) if T has n nodes.

See Chapter 5, P. 255.

A **pre-order tree walk** would print

key[x] before it prints the values in either subtree.

A **post-order tree walk** prints the

key[x] after it prints values in left & right subtrees.

Exercise: Write pseudo-code for these ops.
12.2 Queries

A tree can serve as an efficient structure for storing data (i.e., a database).

\begin{align*}
\text{TreeSearch}(x, k) \\
1.) & \text{ if } x = \text{nil} \text{ or } k = \text{key}[x] \\
2.) & \text{ return } x \\
3.) & \text{ if } k < \text{key}[x] \\
4.) & \text{ return TreeSearch(left[x], k)} \\
5.) & \text{ else} \\
6.) & \text{ return TreeSearch(right[x], k)}
\end{align*}

To search for a node \( x \) with key \( k \), we call

\begin{align*}
\text{TreeSearch(root[T], k)}
\end{align*}

If such a node exists (a pointer to) that node is returned, otherwise nil is returned.

\textbf{Note:} In many applications (e.g., databases) keys will be distributed. If \( T \) contains multiple keys, some node with matching key is returned.
The nodes encountered in TreeSearch form a downward path starting at the root. Run time is $\Theta(h)$ (worst case).

Site iterative version on p. 257.

**Tree Minimum** ($x$)
1.) while left[$x$] $\neq \text{nill}$
2.) $x \leftarrow \text{left}[x]$
3.) return $x$

**Tree Maximum** ($x$)
1.) while right[$x$] $\neq \text{nill}$
2.) $x \leftarrow \text{right}[x]$
3.) return $x$

Awareness of both algorithms follows from BST properties. The worst case run time is $\Theta(h)$ where $h = \text{height}(T)$.

The successor of a node $x$ is the next node to be processed after $x$ in an in order tree walk.
Successor (1) = 2
Successor (8) = 9
Successor (7) = 8

Problem 12.2-6.
If x has no right child & x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.
\textbf{TreeSuccessor}(x)

1. \textbf{if} \textbf{right}(x) \neq \textbf{nil}
2. \textbf{return} \textbf{TreeMinimum} (\textbf{right}(x))
3. \textbf{y} \leftarrow \textbf{right}(x)
4. \textbf{while} \ y \neq \textbf{nil} \ \textbf{do}
5. \quad \textbf{x} \leftarrow \textbf{y}
6. \quad \textbf{y} \leftarrow \textbf{left}(\textbf{y})
7. \textbf{return} \textbf{y}

To understand the 
operation of this algorithm, we consider two cases.

\textbf{CASE 1}: If \textbf{right successor} at \textbf{x}
\textbf{is not empty}, then \textbf{successor of x}
is the leftmost node in \textbf{x}'s right subtree, which is referenced on line 2.

\textbf{CASE 2}: If \textbf{right successor} of \textbf{x} is
\textbf{empty}, and \textbf{x} has a \textbf{successor} \textbf{y},
\textbf{then} \textbf{y} is the
\textbf{lowest ancestor} of \textbf{x} whose
\textbf{left child is also an ancestor of}
\textbf{x} (or \textbf{x} itself). (See Page 12.2-6)
\textbf{To find this node we climb up the}
\textbf{tree} from \textbf{x} until we find a
\textbf{node} which is the \textbf{left child}
of its parent.
12.3

**BST Insertion & Deletion**

The key to maintaining the BST property is to insert and delete elements carefully.

To insert a node \( z \) with key \( z.k = k \) into a BST \( T \), first set

\[
\begin{align*}
\text{key}[z] &\leftarrow k \\
\text{left}[z] &\leftarrow \text{NIL} \\
\text{right}[z] &\leftarrow \text{NIL}
\end{align*}
\]

Then

\[
\textbf{Insert}(T, z)
\]

1. \( y \leftarrow \text{NIL} \)
2. \( x \leftarrow \text{root}[T] \)
3. \( \text{while } x \neq \text{NIL} \)
4. \( y \leftarrow x \)
5. \( \text{if } \text{key}[z] \lt \text{key}[x] \)
6. \( x \leftarrow \text{left}[x] \)
7. \( \text{else} \)
8. \( x \leftarrow \text{right}[x] \)
9. \( \text{if } y = \text{NIL} \) \( (T \text{ was empty in this else}) \)
10. \( \text{root}[T] \leftarrow z \)
11. \( \text{else if } \text{key}[z] \lt \text{key}[y] \)
12. \( \text{left}[y] \leftarrow z \)
13. \( \text{else} \)
14. \( \text{right}[y] \leftarrow z \)
The pointer variable x tracks a path from the root down to a leaf, with y maintained as the parent of x. When x is set to NIL (in line 6 or 8), this NIL will occupy the position where z belongs, so p[x] is set to y.

If y is still NIL at this point, then the tree was empty to begin with (i.e. root[1] = NIL) so z becomes the root. If not, the only question is whether z should be the left or right child of y. This is decided in lines 12 - 15.

Insert tracks a path from the root down to a leaf, so runs in O(h) time.
Insert \((T, z)\) without key \(z\) equal to 5

The procedure for deleting a node is considered in three cases:

Case 1: if \(z\) has no children, modify node \(T\) to have \(NIL\) in place of its children \(z\). In diagram ex. delete (5)

Case 2: if \(z\) has a single child, split out \(z\) (link its child to \(z\) current parent) delete (6)
CASE 3: If z has two children, split out z's successor y (which has no left child by problem) and copy fields of y into fields of z (i.e. key \( \frac{1}{2} \), satellite data.)

**Example**

Delete (4), copy data in 5 into 4.

**Problem**

1. Tree-Delete
2. (P. 262)