Binary Search Trees
Binary Trees

- Recursive definition
  1. An empty tree is a binary tree
  2. A node with two child subtrees is a binary tree
  3. Only what you get from 1 by a finite number of applications of 2 is a binary tree.

Is this a binary tree?
Binary Search Trees

- View today as data structures that can support dynamic set operations.
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.

- Can be used to build
  - Dictionaries.
  - Priority Queues.

- Basic operations take time proportional to the height of the tree – $O(h)$. 
**BST – Representation**

- Represented by a linked data structure of nodes.
- \( root(T) \) points to the root of tree \( T \).
- Each node contains fields:
  - \( key \)
  - \( left \) – pointer to left child: root of left subtree.
  - \( right \) – pointer to right child: root of right subtree.
  - \( p \) – pointer to parent. \( p[root[T]] = NIL \) (optional).
Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of $n$ nodes in the worst case.
Tree Search

Tree-Search\( (x, k) \)
1. if \( x = \text{NIL} \) or \( k = \text{key}[x] \)
2. then return \( x \)
3. if \( k < \text{key}[x] \)
4. then return Tree-Search\( (\text{left}[x], k) \)
5. else return Tree-Search\( (\text{right}[x], k) \)

Running time: \( O(h) \)
Finding Min & Max

- The binary-search-tree property guarantees that:
  - The minimum is located at the left-most node.
  - The maximum is located at the right-most node.

<table>
<thead>
<tr>
<th>Tree-Minimum($x$)</th>
<th>Tree-Maximum($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. while $left[x] \neq NIL$</td>
<td>1. while $right[x] \neq NIL$</td>
</tr>
<tr>
<td>2. do $x \leftarrow left[x]$</td>
<td>2. do $x \leftarrow right[x]$</td>
</tr>
<tr>
<td>3. return $x$</td>
<td>3. return $x$</td>
</tr>
</tbody>
</table>

Q: How long do they take?
Predecessor and Successor

- Successor of node $x$ is the node $y$ such that $key[y]$ is the smallest key greater than $key[x]$.
- The successor of the largest key is NIL.
- Search consists of two cases.
  - If node $x$ has a non-empty right subtree, then $x$’s successor is the minimum in the right subtree of $x$.
  - If node $x$ has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - $x$’s successor $y$ is the node that $x$ is the predecessor of ($x$ is the maximum in $y$’s left subtree).
    - In other words, $x$’s successor $y$, is the lowest ancestor of $x$ whose left child is also an ancestor of $x$. 
Pseudo-code for Successor

```
Tree-Successor(x)

1. if right[x] ≠ NIL
2. then return Tree-Minimum(right[x])
3. y ← p[x]
4. while y ≠ NIL and x = right[y]
5. do x ← y
6. y ← p[y]
7. return y
```

Code for *predecessor* is symmetric.

**Running time:** $O(h)$
BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Insertion is easier than deletion.

**Tree-Insert(T, z)**

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

- Initialization: $O(1)$

- While loop in lines 3-7 searches for place to insert $z$, maintaining parent $y$. This takes $O(h)$ time.

- Lines 8-13 insert the value: $O(1)$

$\Rightarrow$ TOTAL: $O(h)$ time to insert a node.

```
Tree-Insert($T, z$)
1. $y \leftarrow$ NIL
2. $x \leftarrow root[T]$
3. while $x \neq$ NIL
4.     do $y \leftarrow x$
5.         if key[$z$] $<$ key[$x$]
6.             then $x \leftarrow left[x]$
7.         else $x \leftarrow right[x]$
8. $p[z] \leftarrow y$
9. if $y =$ NIL
10.     then $root[t] \leftarrow z$
11. else if key[$z$] $<$ key[$y$]
12.     then $left[y] \leftarrow z$
13. else $right[y] \leftarrow z$
```
Tree-Delete \((T, x)\)

if \(x\) has no children
  ♦ case 0
  then remove \(x\)

if \(x\) has one child
  ♦ case 1
  then make \(p[x]\) point to child

if \(x\) has two children (subtrees)
  ♦ case 2
  then swap \(x\) with its successor
  perform case 0 or case 1 to delete it

⇒ TOTAL: \(O(h)\) time to delete a node
Tree Deletion

CASE 0
Tree Deletion

CASE 1
Tree Deletion

CASE 2
Introduction to Computer Programming

Balanced Search Trees
- Red-black trees
- Height of a red-black tree
- Rotations
- Insertion
Balanced search trees

*Balanced search tree:* A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of $n$ items.

**Examples:**
- AVL trees
- Red-black trees
Red-black trees

This data structure requires an extra one-bit color field in each node.

Red-black properties:
1. Every node is either red or black.
2. The root and leaves (NIL’s) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node $x$ to a descendant leaf have the same number of black nodes = $\text{black-height}(x)$.
Example of a red-black tree

$h = 4$
Example of a red-black tree

1. Every node is either red or black.
Example of a red-black tree

2. The root and leaves (NIL’s) are black.
Example of a red-black tree

3. If a node is red, then its parent is black.
Example of a red-black tree

4. All simple paths from any node \( x \) to a descendant leaf have the same number of black nodes = \( \text{black-height}(x) \).
Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**

- Merge red nodes into their black parents.
Theorem. A red-black tree with n keys has height 
\[ h \leq 2 \ lg(n + 1). \]

Proof. (The book uses induction. Read carefully.)

Intuition:
• Merge red nodes into their black parents.
Height of a red-black tree

**Theorem.** A red-black tree with \( n \) keys has height

\[
h \leq 2 \log(n + 1).
\]

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
Theorem. A red-black tree with $n$ keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

Intuition:
• Merge red nodes into their black parents.
Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height

$$h \leq 2 \lg(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
**Theorem.** A red-black tree with \( n \) keys has height
\[
h \leq 2 \lg(n + 1).
\]

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth \( h' \) of leaves.
Proof (continued)

• We have $h' \geq h/2$, since at most half the leaves on any path are red.

• The number of leaves in each tree is $n + 1$
  \[ \Rightarrow n + 1 \geq 2^{h'} \]
  \[ \Rightarrow \lg(n + 1) \geq h' \geq h/2 \]
  \[ \Rightarrow h \leq 2 \lg(n + 1). \]
Query operations

**Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with $n$ nodes.
Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

• the operation itself,
• color changes,
• restructuring the links of the tree via “rotations”.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
Rotations

Rotations maintain the inorder ordering of keys:
• \( a \in \alpha, \; b \in \beta, \; c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c. \)

A rotation can be performed in \( O(1) \) time.
Insertion into a red-black tree

**IDEA:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
**Insertion into a red-black tree**

**Idea:** Insert \( x \) in tree. Color \( x \) red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert \( x = 15 \).
- Recolor, moving the violation up the tree.
Insertion into a red-black tree

**Idea:** Insert \( x \) in tree. Color \( x \) red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert \( x = 15 \).
- Recolor, moving the violation up the tree.
- **Right-Rotate(18).**
**Example:**
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- **RIGHT-ROTATE**(18).
- **LEFT-ROTATE**(7) and recolor.
**Insertion into a red-black tree**

**IDEA:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- **RIGHT-ROTATE(18).**
- **LEFT-ROTATE(7) and recolor.**
Pseudocode

**RB-INSERT**(*T, x*)

**TREE-INSERT**(*T, x*)

```
color[x] ← RED  ▶ only RB property 3 can be violated

while x ≠ root[*T*] and color[p[x]] = RED
    do if p[x] = left[p[p[x]]]
        then y ← right[p[p[x]]]  ▶ y = aunt/uncle of x
            if color[y] = RED
                then ⟨Case 1⟩
            else if x = right[p[x]]
                then ⟨Case 2⟩ ▶ Case 2 falls into Case 3
                    ⟨Case 3⟩
            else ⟨“then” clause with “left” and “right” swapped⟩

color[root[*T*]] ← BLACK
```
Graphical notation

Let △ denote a subtree with a black root.

All △’s have the same black-height.
Case 1

(Or, children of \(A\) are swapped.)

Recolor

Push \(C\)’s black onto \(A\) and \(D\), and recurse, since \(C\)’s parent may be red.
Case 2

\[
\text{LEFT-ROTATE}(A)
\]

Transform to Case 3.
Case 3

\textbf{RIGHT-ROTATE}(C)

\begin{align*}
\text{Done! No more violations of RB property 3 are possible.}
\end{align*}
Analysis

• Go up the tree performing Case 1, which only recolors nodes.
• If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

**Running time:** \(O(\log n)\) with \(O(1)\) rotations.

**RB-DELETE** — same asymptotic running time and number of rotations as **RB-INSERT** (see textbook).