Red-Black Trees
Red-Black Trees

Definition. A red-black tree is a binary search tree with the following properties:

- Every node is either red or black;
- The root is black;
- Every leaf is NIL and is black;
- If a node is red, then both its children are black;
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

(Note: Every node in a binary tree is either a leaf or has BOTH a left AND right child.)
Red-Black Tree: Example
Red-Black Tree: Example
**NOT a Red-Black Tree**

This tree is **NOT** a red-black tree. Why not?
Red-Black Tree Exercise

Color the following tree so that it is a red-black tree:
Red-Black Tree Height

**Definition.** A red-black tree satisfies the following properties:

- Every node is either red or black;
- The root is black;
- Every leaf is \texttt{NIL} and is black;
- If a node is red, then both its children are black;
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

**Theorem.** A red-black tree with $n$ internal nodes has height at most $2 \log_2(n + 1)$. 
Theorem 1. A complete binary search tree of height $h$ has $2^{h+1} - 1$ nodes.

Proof. Level $i$ has $2^i$ nodes ($i = 0, 1, \ldots, h$).

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^h = 2^{h+1} - 1.$$
Red-Black Tree Size

**Theorem 2.** A red-black tree of height $h$ has at least $2^{\lceil h/2 \rceil} - 1$ internal nodes.

**Proof.** (By Dr. Y. Wang.)
Let $T$ be a red-black tree of height $h$.
Remove the leaves of $T$ forming a tree $T'$ of height $h - 1$.
Let $r$ be the root of $T'$.
Since no child of a red node is red and $r$ is black, the longest path from $r$ to a leaf has at least $\lceil h/2 \rceil$ black nodes.
Since every path from $r$ to a leaf has the same number of black nodes, every path from $r$ to a leaf has at least $\lceil h/2 \rceil$ black nodes.
Thus, $T'$ contains a complete binary tree of height at least $\lceil h/2 \rceil - 1$.
(Note: height = Number of EDGES on longest path from root to leaf.)
By Theorem 1, tree $T'$ has at least $2^{\lceil h/2 \rceil - 1 + 1} - 1$ nodes so tree $T$ has at least $2^{\lceil h/2 \rceil} - 1$ internal nodes. \qed
Red-Black Tree Height

**Theorem.** A red-black tree with \( n \) internal nodes has height at most \( 2 \log_2(n + 1) \).

**Proof.** Let \( h \) be the height of a red-black tree with \( n \) nodes. By Theorem 2, \( n \geq 2^{\lceil h/2 \rceil} - 1 \).

Thus, \( \log_2(n + 1) \geq \lceil h/2 \rceil \geq h/2 \) so \( h \leq 2 \log_2(n + 1) \). \( \square \)
Red-Black Tree: Insert
Red-Black Tree Insert

function RBLocateParent(T, z)
/* Return future parent of z in tree */
1  y ← NIL;
2  x ← T.root;
3  while (x is not a leaf) do
4     y ← x;
5     if (z.key < x.key) then x ← x.left;
6     else x ← x.right;
7  end
8  return (y);
function RBTreeInsert(T, z)
1  y ← RBLocateParent(T, z);
2  z.parent ← y;
3  if (y = NIL) then T.root ← z; /* tree T was empty*/
4  else if (z.key < y.key) then y.left ← z;
5  else y.right ← z;
6  z.left ← leaf;
7  z.right ← leaf;
8  z.color ← Red;
9  RBInsertFixup(T,z);
Insert Fixup: Case I

If the parent and “uncle” of $z$ are Red:

- Color the parent of $z$ Black;
- Color the uncle of $z$ Black;
- Color the grandparent of $z$ Red;
- Repeat on the grandparent of $z$. 
Red-Black Tree Insert Fixup: Case I
function Sibling(x)
/* Return sibling of x */
1 if (x.parent = NIL) then error “Root has no siblings.”;
2 p ← x.parent;
3 if (p.left = x) then return (p.right);
4 else return (p.left);
Red-Black Tree Insert Fixup: Case I

function RBInsertFixupA(T, alters z)
1 while (z ≠ T.root) and (z.parent.color ≠ Black) do
2     y ← Sibling(z.parent);
3     if (y.color = Black) then return;
4     z.parent.color ← Black;
5     y.color ← Black;
6     z ← z.parent.parent;
7     z.color ← Red;
8 end
Insert Fixup: Case III

If the parent of $z$ is red and its “uncle” is black:
If $z$ is a left child and its parent is a left child:

- Right Rotate on the grandparent of $z$;
- Color the parent of $z$ Black;
- Color the sibling of $z$ red.
Red-Black Tree Insert Fixup: Case III
Red-Black Tree Insert Fixup: Case III

\[
\text{function } \text{RBInsertFixupC}(T, \text{alters } z) \\
1 \text{ if } (z = T.\text{root}) \text{ or } (z.\text{parent}.\text{color} = \text{Black}) \text{ then return;} \\
2 \quad x \leftarrow z.\text{parent}; \\
3 \quad w \leftarrow x.\text{parent}; \\
4 \quad \text{if } (z = x.\text{left}) \text{ and } (x = w.\text{left}) \text{ then} \\
5 \quad \quad \text{RightRotate}(T, w); \\
6 \quad \quad x.\text{color} \leftarrow \text{Black}; \\
7 \quad \quad w.\text{color} \leftarrow \text{Red}; \\
8 \quad \text{else if } (z = x.\text{right}) \text{ and } (x = w.\text{right}) \text{ then} \\
9 \quad \quad \text{Handle same as above with “right” and “left” exchanged} \\
10 \quad \ldots
\]
Insert Fixup: Case II

If the parent $x$ of $z$ is red and its “uncle” is black:
If $z$ is a right child and its parent $x$ is a left child:

- $z \leftarrow x$
- Left Rotate on $x$;
- Apply algorithm for Case III.
Red-Black Tree Insert Fixup: Case II
Red-Black Tree Insert Fixup: Case II

![Diagram of Red-Black Tree Insert Fixup Case II](image)

**function** `RBInsertFixupB(T, alters z)`

1. if \((z = T\text{.root})\) or \((z\text{.parent.color} = \text{Black})\) then return;
2. \(x \leftarrow z\text{.parent};\)
3. \(w \leftarrow x\text{.parent};\)
4. if \((z = x\text{.right})\) and \((x = w\text{.left})\) then
   5. \(z \leftarrow x;\)
   6. `LeftRotate(T,x);`
7. else if \((z = x\text{.left})\) and \((x = w\text{.right})\) then
   8. Handle same as above with “right” and “left” exchanged
9. ...
function RBInsertFixup(T, z)
1  RBInsertFixupA (T,z);
2  RBInsertFixupB (T,z);
3  RBInsertFixupC (T,z);
4  T.root.color ← Black;
Red-Black Tree Insert Fixup
function RBInsertFixup(T, z)
1 RBInsertFixupA (T,z);
2 RBInsertFixupB (T,z);
3 RBInsertFixupC (T,z);
4 T.root.color ← Black;