CISC 320 Introduction to Algorithms
Fall 2003

Lecture 8
Red-Black Trees

Binary Search Trees (BST)

key[x]: key stored at x.
left[x]: pointer to left child of x.
right[x]: pointer to right child of x.
p[x]: pointer to parent node of x.

binary-search-tree property:
for every node x:

\[ \text{key}[y] \leq \text{key}[x] \leq \text{key}[z] \]

where \( y \) is any node in the left subtree of \( x \), and
\( z \) is any node in the right subtree of \( x \).

e.g., two valid BSTs for the keys 2, 3, 4, 5, 7, 8.
Inorder-tree-walk(x)
1. if x ≠ nil
2. then inorder-tree-walk(left[x]);
3. print key[x];
4. inorder-tree-walk(right[x]);

It prints all elements in monotonically increasing order, in \( \Theta(n) \) time.

BST Search
Search(T, k)
1. \( x = \text{root}[T]; \)
2. if \( x = \text{nil} \) or \( k = \text{key}[x] \)
3. then return \( x \);
4. if \( k < \text{key}[x] \)
5. then return Search(left[x], k)
6. else return Search(right[x], k);

Time: \( O(h) \), where \( h \) is the tree height.
- for a balanced binary tree, \( h = \lg(n) \)
- worst-case: \( h = n \).
Rotation

Note: 1. inorder key ordering is unchanged after rotation: a, x, b, y, c
2. rotation takes O(1) time.
Is there a mechanism to automatically rotate whenever the tree is significantly unbalanced?
Red-Black Trees

Red-black tree is a binary search tree

- Every node is either red or black.
- Root and leaves (nil) are black
- If a node is red, then both children are black
- All paths from any node x to a descendant leaf have same number of black nodes.

Definition: black-height of a node x is the number of black nodes (excluding x) on any path from x to a descendant leaf.

A red-black tree

Intuition: if a red-black tree contains black nodes only, the tree is perfectly balanced, i.e., it is a complete binary tree. Presence of red nodes corrupts the balance, but not much, because of the restrictions imposed on red nodes.
Theorem 6.3  A red-black tree with \( n \) internal nodes has height at most \( 2 \lg(n+1) \).

**proof:**

a) for any node \( x \), its black-height denoted as \( bh(x) \), there are at least \( 2^{bh(x)} - 1 \) internal nodes under \( x \) (proof by induction).

b) if the root \( r \) has height \( h \), then \( h \leq 2^{bh(r)} \), because at least half of the nodes on any path from \( r \) to a leaf must be black. According to a), we have

\[
2^{bh(r)} - 1 \leq n \\
bh(r) \leq \lg(n+1) \\
h \leq 2\lg(n+1).
\]

QED

Therefore, a red-black tree can never be too off-balanced. As a result, searching a key in a red-black tree of \( n \) nodes takes \( O(2 \lg(n+1)) \) time.

How about insert and delete?
More work is needed for these operations since the red-black tree properties need to be maintained.
Insertion

step 1: locate where to insert, via an unsuccessful search. \(O(\log n)\)

step 2: insert. \(O(1)\)

new node is assigned red color. why?
- any new node potentially can unbalance the tree
- if red node gets a red child, RBT is broken.
  This alerts us to prevent from continuously adding nodes at a branch.
- a hard argument: otherwise the black-height of the tree is not conserved.

step 3: check if red-black-tree properties are damaged. If yes, fix it by rotations. \(O(?)\)

---

```
RB-Insert(T, z)
1. y ← nil[T]
2. x ← root[T]
3. while x ≠ nil[T]
4.   do y ← x
5.     if key[z] < key[x]
6.       then x ← left[x]
7.     else x ← right[x]
8.   pl[z] ← y
9. if y = nil[T]
10.  then root[T] ← z
11. else if key[z] < key[y]
12.   then left[y] ← z
13. else right[y] ← z
14. left[z] ← nil[T]
15. right[z] ← nil[T]
16. Color[z] ← RED
17. RB-Insert-Fixup(T, z)
```
1. Why C must be black? Because otherwise B and C already broke RBT.
2. Why C has to be red after B and D are changed to black? To maintain the black height for any node above C, say E.
3. If C is not root, its color change may propagate the problem up.
   if C is the root, we only need to recolor it as black.
Case 2: right child, black uncle

1. Does this left rotation at node B make the tree more balanced? Not yet.

Case 3: left child, black uncle

Note: 1. The right rotation at node C makes the tree more balanced
2. will not propagate further up.
Case 4, 5 and 6 are just mirror symmetric to cases 1, 2, and 3 respectively, and can be similarly handled.
RB-Insert-Fixup(T, z)
1. while color[p[z]] = RED
2.   do if p[z] = left[p[p[z]]]
3.     then y ← right[p[p[z]]]
4.     if color[y] = RED
5.       then color[p[z]] ← BLACK
6.       color[y] ← BLACK
7.       color[p[p[z]]] ← RED
8.     z ← p[p[z]]
9.     else if z = right[p[z]]
10.    then z ← p[z]
11.   LEFT-ROTATE(T, z)
12.   color[p[z]] ← BLACK
13.   color[p[p[z]]] ← RED
14. RIGHT-ROTATE(T, p[p[z]])
15. else (same as then clause with "right" and "left" exchanged)
16. Color[root[T]] ← BLACK

Time analysis for insertion

RB-Insert-Fixup either removes a red edge by constant time (cases 2, and 3) or propagates red edge one level up (never down), at most to the root, which is the worst case. As a red-black tree of n internal nodes can not be higher than O(lg n), RB-Insert-Fixup runs in O(lg n) time. Therefore, the total time is

\[ T(n) = O(lg n) + O(lg n) = O(lg n) \]
**deletion**

\[ \text{RB-Delete}(T, z) \] // to delete z from tree T
1. if left\([z]\) = nil\([T]\) or right\([z]\) = nil\([T]\); // z has just one child node or no child node
   then \( y \leftarrow z \) // assign z to y
2. else \( y \leftarrow \text{Tree-Successor}(z) \) // assign z’s successor to y when z has two children
3. if left\([y]\) \( \neq \) nil\([T]\)
   then \( x \leftarrow \text{left}[y] \)
   else \( x \leftarrow \text{right}[y] \)
4. \[ \text{if p}[x] \neq \text{nil}[T] \]
5. \[ \text{then x} \leftarrow \text{left}[y] \]
6. \[ \text{else x} \leftarrow \text{right}[y] \]
7. \[ \text{p}[x] \leftarrow \text{p}[y] \]
8. \[ \text{if p}[y] = \text{nil}[T] \]
9. \[ \text{then root}[T] \leftarrow x \]
10. \[ \text{else if y} = \text{left}[p[y]] \]
11. \[ \text{then left}[p[y]] \leftarrow x \]
12. \[ \text{else right}[p[y]] \leftarrow x \]
13. \[ \text{if y} \neq z \]
14. \[ \text{then key}[z] \leftarrow \text{key}[y] \]
15. \[ \text{copy y’s satellite data into z} \]
16. \[ \text{if color}(y) = \text{BLACK} \]
17. \[ \text{then RB-Delete-Fixup}(T, x) \]
18. Return y

---

```
(a)
```

```
(b)
```

```
(c)
```
If the spliced-out node was red, the tree remains as red-black tree, because
  - No red nodes would be made adjacent
  - Black-heights in the tree not changed

If the spliced-out node y is black, we must give y’s descendents another black ancestor. Let’s color y’s child x black.
  - If x had been a red node, this may solve the problem
  - If x had been a black node, we got a doubly black node

---

Case 1: the double black x has a red brother.
Case 2: x has a black brother w, and w has two black children
Case 3: x has a black brother w, and w has a red left child and black right child
Case 4: x has a black brother w, and w has a red right child, and the left child’s color can be either.
Case 1: X’s sibling w is red

1. X is doubly black.
2. X’s brother w must have black children, because B would have unequalled black-length paths otherwise.
3. After coloring w as black and left rotating on x’s parent, no new violation of RBT is created, and case 1 is converted to case 2, 3, or 4.

Case 2: x’s sibling w is black, and w has two black children

1. X is doubly black. Take one black off both x and w, then x becomes just black, and w becomes red. Under x’s parent, RBT is still conserved. Payoff is that x is singly black.
2. To ensure rbt not violated in nodes beyond this subtree, one black is added at x’s parent, which becomes the new x. The color problem is floated up.
Case 3: x’s sibling w is black, w has red left and black right

1. Switch w’s color with its left child, and right rotate on w.
2. No RBT violation is created for this branch from step 1
3. Case 3 is converted to case 4.

Case 4: x’s sibling w is black, and w has red right

1. Switch p[x]'s color with w, and left rotate on p[x]
2. Change w’s right child to black.
RB-Delete-Fixup(T, x)
1. While x ≠ root[T] and color[x] = BLACK
2. do if x = left[p[x]]
3. then w ← right[p[x]]
4. if color[w] = RED
5. then color[w] = BLACK
6. color[p[x]] = RED
7. Left-Rotate(T, p[x])
8. if color[left[w]] = BLACK and color[right[w]] = BLACK
9. then color[w] = RED
10. x ← p[x]
11. else if color[right[w]] = BLACK
12. then color[left[w]] = BLACK
13. color[w] = RED
14. Right-Rotate(T, w)
15. w ← right[p[x]]
16. color[w] ← color[p[x]]
17. color[p[x]] ← BLACK
18. color[right[w]] ← BLACK
19. Left-Rotate(T, p[x])
20. x ← root[T]
21. else (same as then clause with “right” and “left” exchanged)
22. color[x] ← BLACK
23.

- **Time analysis for RB-Delete(T, z)**
  - Locate z: O(lg n)
  - Fixup
    - Cases 1, 3, and 4: O(1)
    - Case 2: O(lg n)
  - Overall cost: O(lg n)