CISC 320 Introduction to Algorithms
Fall 2005

Lecture 9
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 = root[T];
2. if x = nil or k = key[x]
3. then return x;
4. if k < 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

1. Red-black tree is a binary search tree. Every node is either red or black.
2. Root is always black
3. Every leaf (nil) must be black
4. If a node is red, then both children are black
5. 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.

Theorem: 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.
- b) if the root r has height h, then $h \leq 2 \cdot bh(r)$, 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. $O(\lg n)$
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.
step 3: check if red black tree properties are damaged. If yes, fix it by rotations. $O(?)$

RB-Insert($T$, $z$)
1. $y \leftarrow \text{nil}[T]$
2. $x \leftarrow \text{root}[T]$
3. while $x \neq \text{nil}[T]$
4. do $y \leftarrow x$
5. if key[$z$] < key[$x$]
6. then $x \leftarrow \text{left}[x]$
7. else $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. if $y = \text{nil}[T]$
10. then $\text{root}[T] \leftarrow z$
11. else if key[$z$] < key[$y$]
12. then $\text{left}[y] \leftarrow z$
13. else $\text{right}[y] \leftarrow z$
14. $\text{left}[z] \leftarrow \text{nil}[T]$
15. $\text{right}[z] \leftarrow \text{nil}[T]$
16. $\text{Color}[z] \leftarrow \text{RED}$
17. RB-Insert-Fixup($T$, $z$)

Case 1: red uncle

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[y] ← BLACK
6. color[p[z]] ← RED
7. color[p[p[z]]] ← RED
8. z ← 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 \(O(\log n)\), RB-Insert-Fixup runs in \(O(\log n)\) time. Therefore, the total time is

\[ T(n) = O(\log n) + O(\log n) = O(\log n) \]