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:
  - key[y] ≤ key[x] ≤ 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 O(n) time.

BST Search

1. x = root[T];
2. if x = nil or k < key[x]
3. then return nil;
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.
A red-black tree

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.

Theorem 6.3 A red-black tree with n internal nodes has height at most $2 \log(n+1)$.

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 $h \leq 2 \log(r)$, because at least half of the nodes on any path from r to a leaf must be black.

According to a), we have

$2^{bh(x)} - 1 \leq n$

$bh(r) \leq \log(n+1)$

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 $O(2 \log(n+1))$ 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.
  - 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. $O(?)$

---

### 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.

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(lgn), RB-Insert-Fixup runs in O(lgn) time. Therefore, the total time is

T(n) = O(lgn) + O(lgn) = O(lgn)

deletion

Case 1

Case 2

Case 3

RB-Delete(T, z)

1. white color[z][i] = RED
2. do if d[z] = left[z][d][i]
3. then y = right[z][d][i]
4. if color[y] = RED
5. then color[y] = BLACK
6. color[z][d][i] = RED
7. z = R4[y]
8. else if x = right[z][d][i]
9. then x = z
10. LEFT-ROTATE(T, z)
11. color[z][d][i] = BLACK
12. color[z][d][i] = RED
13. RIGHT-ROTATE(T, y[d][i])
14. else (same as then clause with “right” and “left” exchanged)
15. Color[rot][z] = BLACK

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

Case 3: x’s sibling w is black, w has red left and black right

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

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 unequaled 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 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.
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: $O(\lg n)$