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.

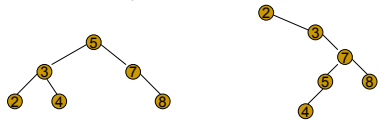
binary-search-tree property:

for every node x:

$$key[y] \le key[x] \le key[z]$$

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)
    if x ≠ nil
      then inorder-tree-walk(left[x]);
           print key[x];
           inorder-tree-walk(right[x]);
```

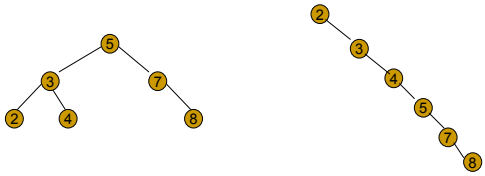
It prints all elements in monotonically increasing order, in Θ(n) time.

BST Search

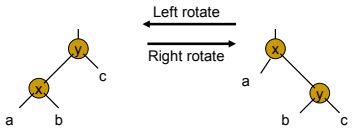
```
Search(T, k)
    x = root[T];
    if x = nil or k = key[x]
      then return x;
    if k < key[x]
      then return Search(left[x], k)
      else return Search(right[x], k);
```


Time: O(h), where h is the tree height.

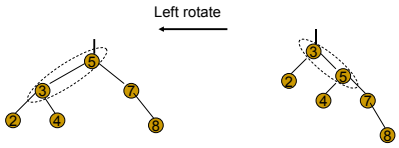
- for a balanced binary tree, h = lg(n)
- worst-case: h = n.



- Rotation

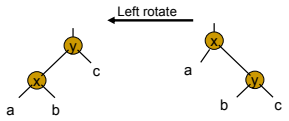


Note: 1. inorder key ordering is unchanged after rotation: a, x, b, y, c
2. rotation takes O(1) time.



- Left-rotate(T,x)

```
y ← right[x]
right[x] ← left[y]
if left[y] ≠ nil
   then p[left[y]] ← x
p[y] ← p[x]
if p[x] = nil
  then root[T] ← y
  else if x = left[p[x]]
            then left[p[x]] ← y
            else right[p[x]] ← y
left[y] ← x
p[x] ← y
```



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.

- 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: 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 $h \le 2\, bh(r)$, because at least half of the nodes on any path from r to a leaf must be black. According to a), we have

$$2^{bh(r)} - 1 \le n$$

$$bh(r) \le \lg(n+1)$$

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

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)

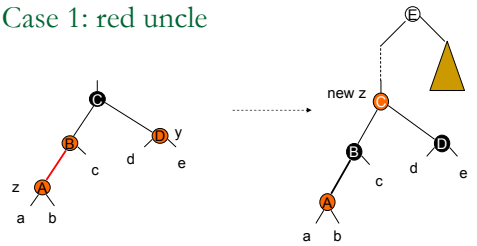
new node is assigned red color. why?

- 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)
    y ← nil[T]
    x ← root[T]
    while x ≠ nil[T]
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
    p[z] ← y
    If y = nil[T]
      then root[T] ← z
      else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
    left[z] ← nil[T]
    right[z] ← nil[T]
    Color[z] ← RED
    RB-Insert-Fixup(T, z)
```

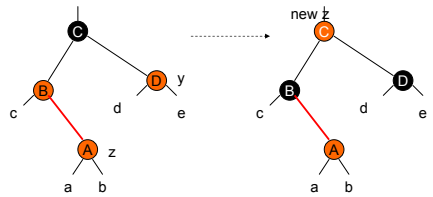

Case 1: red uncle



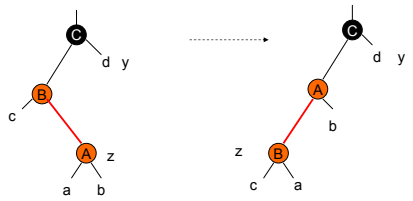
1. Why C must be black? Because otherwise B and C already broke RBT.
2. Why C has to be red after B and D are changed to black? To maintain the black height for any node above C, say E.
3. If C is not root, its color change may propagate the problem up.

   if C is the root, we only need to recolor it as black.

Case 1: red uncle

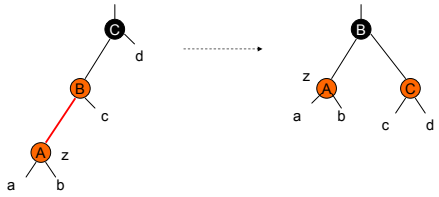


Case 2: right child, black uncle



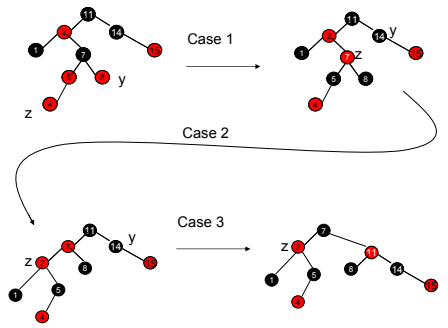
1. Does this left rotation at node B make the tree more balanced? Not yet.

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)
    while color[p[z]] = RED
      do if p[z] = left[p[p[z]]]
            then y ← right[p[p[z]]]
              if color[y] = RED
                then color[p[z]] ← BLACK
                     color[y] ← BLACK
                     color[p[p[z]]] ← RED
                     z ← p[p[z]]
              else if z = right[p[z]]
                      then z ← p[z]
                          LEFT-ROTATE(T, z)
                    color[p[z]] ← BLACK
                    color[p[p[z]]] ← RED
                    RIGHT-ROTATE(T, p[p[z]])
          else ( same as then clause with "right" and "left" exchanged)
    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

$$T(n) = O(\lg n) + O(\lg n) = O(\lg n)$$