

CSC 263 Lecture 3

June 1, 2004

For another reference, see chapters 12 and 13 of the textbook (CLRS).

1 Dictionaries

A dictionary is an important abstract data type (ADT). It represents the following object and operations:

Object: Sets where each element x has a value $key(x)$ which comes from a totally-ordered universe (this just means that for any two keys a and b , either $a > b$, $a < b$, or $a = b$).

Operations (S is a set, x is an element, and k is a key):

- **ISEMPTY(S):** check whether set S is empty or not
- **SEARCH(S,k):** return some x in S s.t. $key(x) = k$, or NIL if no such x exists
- **INSERT(S,x):** insert x in S
- **DELETE(S,x):** remove x from S

There are many possible data structures that could implement a dictionary. We list some of them with their worst-case running times for **SEARCH**, **INSERT**, **DELETE**.

DATA STRUCTURE	SEARCH	INSERT	DELETE
unsorted singly linked list	n	1	n
unsorted doubly linked list	n	1	1
sorted array	$\log n$	n	n
hash table	n	n	n
binary search tree	n	n	n
balanced search tree	$\log n$	$\log n$	$\log n$

2 Binary Search Trees (BSTs)

A binary tree is a BST if it satisfies the

BST Property: For every node x , if node y is in the left subtree of x , then $key(x) \geq key(y)$. If node y is in the right subtree of x , then $key(x) \leq key(y)$.

We'll see how to use this property to help search for a particular key. We also have to check that we can insert and delete elements and still maintain the property.

```

SEARCH (BST root R, key k):
  if R = NIL then
    return NIL
  else if ( k = key(R) ) then
    return R
  else if ( k < key(R) ) then
    return SEARCH ( left(R), k )
  else if ( k > key(R) ) then
    return SEARCH ( right(R), k )

```

In the worst case, we'll start at the root of the tree and follow the longest path in the tree and then find that there is no node with key k . Since the length of the longest path in the tree is the definition of the *height* of the tree, this takes time $\Theta(\text{height of tree})$. For a tree with n nodes, the height can be n (if there are no right children, for instance)! So the worst-case running time (that is, for the worst tree and the worst k) is $\Theta(n)$.

```

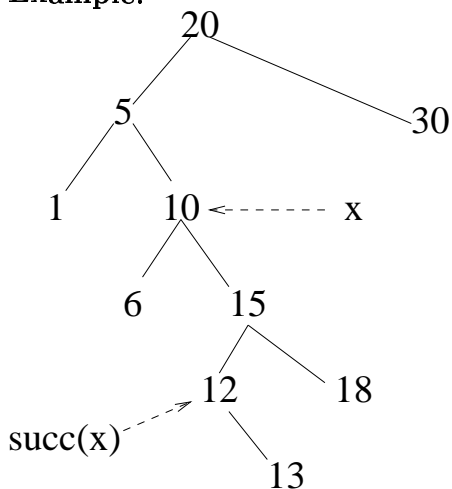
INSERT ( BST root R, node x ):
  if R = NIL then
    R := x
  else if ( key(x) < key(R) ) then
    INSERT ( left(R), x )
  else if ( key(x) > key(R) ) then
    INSERT ( right(R), x )
  else if ( key(x) = key(R) ) then
    /* depends on application */

```

x will always be added as a leaf. Again we might have to follow the longest path from the root to a leaf and then insert x , so in the worst-case, INSERT takes time $\Theta(n)$.

The DELETE operation is more complicated, so we describe it at a higher level. For one thing, we need to define the *successor* of x , $\text{succ}(x)$. This is the node whose key-value is the lowest of those higher than $\text{key}(x)$.

Example:



Notice that $\text{succ}(x)$ is the left-most node in the right subtree of x (that is, starting from x 's right child, go left until there are no left children to follow). This is always true if x has a right child (why?).

Now, DELETE (BST root R, node x) has three cases:

1. If x has no children, simply remove it by setting x to *NIL*.
2. If x has one child y and z is the parent of x , then we remove x and make y the appropriate child of z .
3. If x has two children, then things get a little more complicated. Let A and B be the left and right subtrees of x , respectively. First we find $\text{succ}(x)$. Then we set $\text{key}(x)$ to $\text{key}(\text{succ}(x))$ and DELETE $\text{succ}(x)$ (using case 1 or 2). By the definition of $\text{succ}(x)$, we know that everything in A has key less than or equal to $\text{key}(\text{succ}(x))$. Let B' be B with $\text{succ}(x)$ removed. Everything in B' must have key greater than or equal to $\text{key}(\text{succ}(x))$. So the **BST Property** is still maintained.

Again, if x is the root and $\text{succ}(x)$ is the leaf at the end of the longest path in the tree, then we'll take time $\Theta(n)$ in the worst-case to find $\text{succ}(x)$. Since everything else we do takes constant time, the worst-case running time is $\Theta(n)$.

Notice that the running times for these operations all depend on the height of the tree. If we had some guarantee that the tree's height was smaller (in terms of the number of nodes it contains), then we would be able to support faster operations.

3 Red-Black Trees

A *red-black tree* is a BST that also satisfies the following three properties:

1. Every node x is either red or black ($\text{color}(x) = \text{red}$ or $\text{color}(x) = \text{black}$).
2. Every child of a red node is black.
3. For every node x , any path from x to a leaf contains the same number of black nodes (this number is the *black height* of x , $BH(x)$).

To make things work out easier, we'll consider every node with a key value to be an internal node and the *NIL* values at the bottom of the tree will be the leaves and will be colored black.

3.1 Red-Black Trees Are Short

These three extra properties guarantee that the tree is approximately balanced and therefore pretty shallow. More precisely

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

In order to show this, we first show

Lemma: For any node x in a red-black tree, the number of internal nodes in the subtree rooted at x is at least $2^{BH(x)} - 1$.

Proof of Lemma: We use induction on the height of x . The base case is when x has height 0 (that is, it has no children). Certainly $BH(x)$ is 0 if its regular height is 0, so x must have at least $2^0 - 1 = 0$ children. This is true.

Now consider x with arbitrary height. Let's assume the lemma is true for all nodes of height less than the height of x . Then, for any x , the lemma is true for each of its children (remember we

are considering *NIL* to be a valid child). Each child must have black height at least $BH(x) - 1$, so there must be $2^{BH(x)-1}$ internal nodes in the subtrees rooted at each child. Then, including x , we have $2(2^{BH(x)-1} - 1) + 1 = 2^{BH(x)} - 1$ nodes in the subtree rooted at x .

Now we can easily prove the theorem:

Proof of Theorem: Assume the height of the red-black tree is h . Property 2 says that on any path from the root to leaf, at least half the nodes are black. So the black height of the root must be at least $h/2$. If n is the number of internal nodes, then we know from the lemma that $n \geq 2^{h/2} - 1$, so $2 \log(n + 1) \geq h$.

3.2 Operations on Red-Black Trees

Now we can use the same **SEARCH** routine to search the tree in time $O(\log n)$. **INSERT** and **DELETE** will also take time $O(\log n)$, but they might cause problems since they can violate the color properties of the tree. For instance, if we use the regular **BST INSERT**, then we'll add the new node at the bottom of the tree (so both its children are *NIL*). Then we have to decide whether to make it red or black. If we make it black, we'll certainly violate property 3 of red-black trees. If we make it red, we don't have to worry about property 3, but we might violate property 2 (if its parent is red).

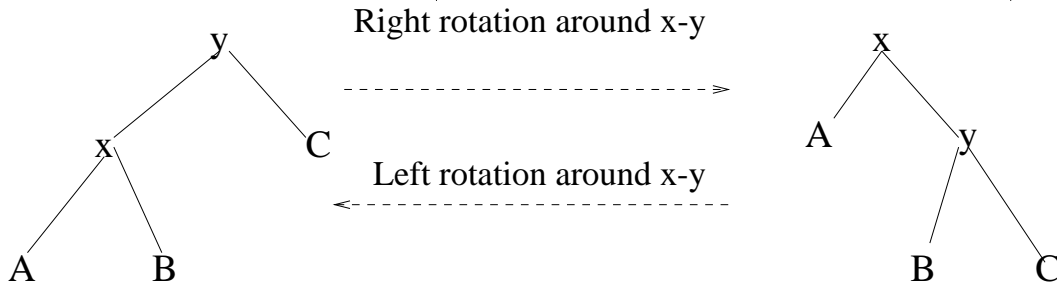
We'll use the following procedure to insert a node x into a red-black tree:

```
Red-BlackINSERT (R, x):
  INSERT ( R, x )
  color (x) := red
  If property 2 violated, fix tree
```

Property 2 can be violated only if $parent(x)$ is colored red. If $parent(x) = R$, the root of the tree, then we can just recolor R to be black. This won't violate property 3 for any node since there is nothing above R . If $parent(x) \neq R$, then we have three cases. We can assume $parent(parent(x))$ is colored black since otherwise we would be violating property 2 before we even inserted x .

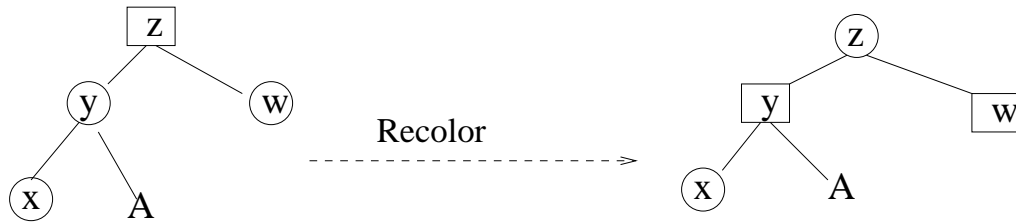
Keep in mind that we might need to apply the fixing operations multiple times before the tree is completely fixed. Hence, even though x starts off with both children *NIL*, we might have moved it upwards in the tree using previous fixing operations, so x might in general have non-*NIL* children.

Before we describe the three cases, we need to know how to rotate a subtree. A *rotation* is simply a transformation of the form (x and y are nodes, A , B , and C are subtrees):



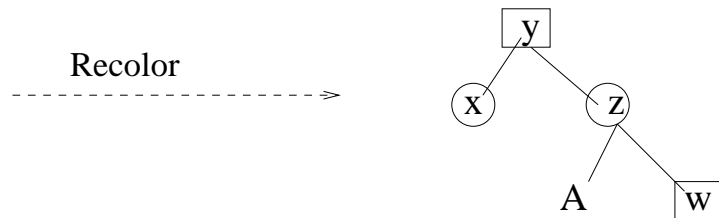
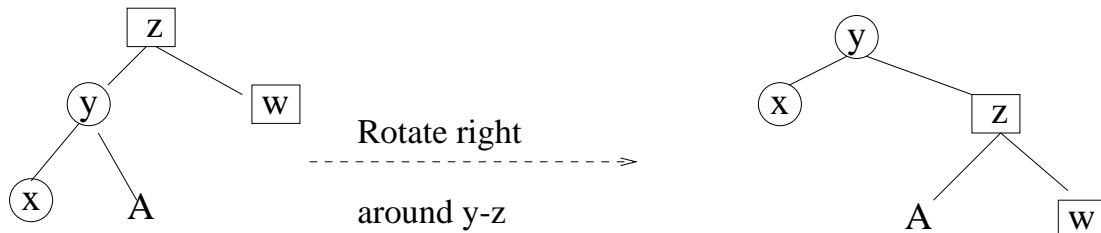
Now we can do the three cases. In each picture, the object on the left is a subtree of the entire red-black tree. It might have nodes above it and below it. A and B stand for subtrees themselves. Squares represent black nodes and circles represent red nodes. In every case, we assume that the subtree on the left does not violate property 3. Based on this assumption, you should check that the final subtree on the right also does not violate property 3.

- **Case 1:** x 's "uncle" is red.



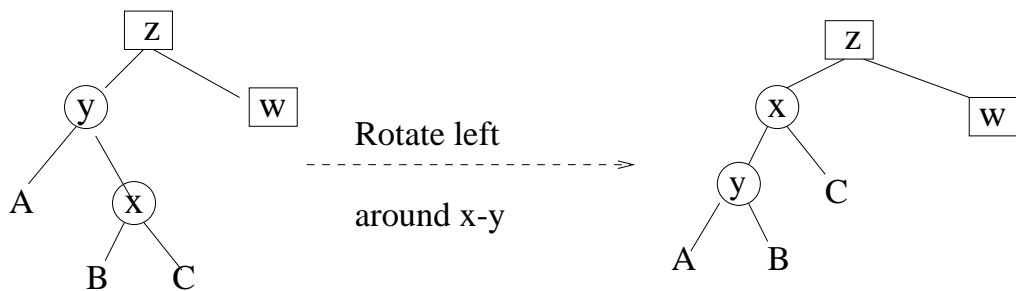
The problem here is that z 's parent might be red, so we still have a violation of property 2. But notice that we have moved the conflict upwards in the tree. Either we can keep applying case 1 until we reach the root, or we can apply case 2 or case 3 and end the fix up process. If we reach the root by applying case 1 (in other words, $parent(z)$ is the root and $parent(z)$ is red, then we just change $parent(z)$ to black.

- **Case 2:** x 's uncle is not red (it's black or does not exist) and $key(x) \leq key(y) \leq key(z)$ (or $key(x) \geq key(y) \geq key(z)$).



Now there are no violations of either property 2 or property 3, so we are finished.

- **Case 3:** x 's uncle is not red and $key(y) \leq key(x) \leq key(z)$ (or $key(y) \geq key(x) \geq key(z)$).



Now we can apply case 2 using y as x and we're done.

In the worst-case, we might have to apply case 1 until we move the red-red conflict all the way up to the root starting from the bottom of the tree. This takes time $O(\log n)$ since the height of the

tree is $O(\log n)$. Combined with the $O(\log n)$ time to do the INSERT, we find that **Red-BlackINSERT** takes time $O(\log n)$.

We'll save **Red-BlackDELETE** for next time.