

## Loop Invariant Review \_\_\_\_\_

- **Termination** shows that if the loop invariant is true after the last iteration of the loop, then the algorithm is correct
- The termination condition is different than induction

## Choosing Loop Invariants \_\_\_\_\_

- Q: How do we choose the right loop invariant for an algorithm?
- A1: There is no standard recipe for doing this. It's like choosing the right guess for the solution to a recurrence relation.
- A2: Following is one possible recipe:

Answers \_\_\_\_\_

- Study the algorithm and list what important invariants seem true during iterations of the loop - it may help to simulate the algorithm on small inputs to get this list of invariants
- 2. From the list of invariants, select one which seems strong enough to prove the correctness of the algorithm
- 3. Try to show Initialization, Maintenance and Termination for this invariant. If you're unable to show all three properties, go back to the step 1.

Answers \_\_\_\_\_

- To show: If key k exists in the tree, Tree-Search returns the elem with key k, otherwise Tree-Search returns nil.
- Loop Invariant: If key k exists in the tree, then it exists in the subtree rooted at node x

• Initialization: Before the first iteration, x is the root of the entire tree, therefor if key k exists in the tree, then it exists in the subtree rooted at node x

8

9

## Maintenance \_\_\_\_\_

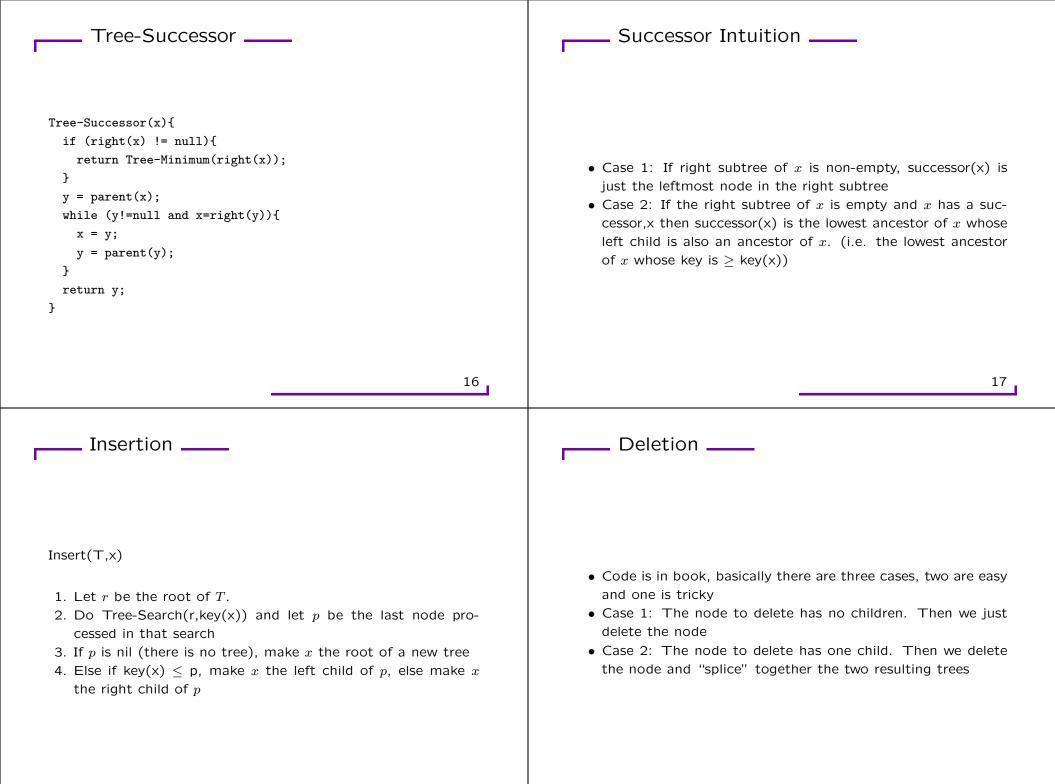
- Maintenance: Assume at the beginning of the procedure, it's true that if key k exists in the tree that it is in the subtree rooted at node x. There are three cases that can occur during the procedure:
  - Case 1: key(x) is k. In this case, the procedure terminates and returns x, so the invariant continues to hold
  - Case 2: k < key(x). In this case, by the *BST Property*, all keys in the subtree rooted on the right child of x are greater than k (since key(x) > k). Thus, if k exists in the subtree rooted at x, it must exist in the subtree rooted at left(x).
  - Case 3:k>key(x). In this case, by the BST Property, All keys in the subtree rooted on the right child of x are less than k (since key(x)<k). Thus, if k exists in the subtree rooted at x, it must exist in the subtree rooted at right(x).</li>

• By the loop invariant, we know that when the procedure terminates, if k is in the tree, then it is in the subtree rooted at x. If k is in fact in the tree, then x will never be nil, and so the procedure will only terminate by returning a node with key k. If k is not in the tree, then the only way the procedure will terminate is when x is nil. Thus, in this case also, the

procedure will return the correct answer.

Termination \_\_\_\_\_

| 12  | 13  |
|---|---|
| Tree Min/Max  | Successor   |
| <ul> <li>Tree Minimum(x): Return the leftmost child in the tree rooted at x</li> <li>Tree Maximum(x): Return the rightmost child in the tree rooted at x</li> </ul> | <ul> <li>The successor of a node x is the node that comes after x in the sorted order determined by an in-order tree walk.</li> <li>If all keys are distinct, the successor of a node x is the node with the smallest key greater than x</li> </ul> |



Case 3 Analysis \_\_\_\_\_ • All of these operations take O(h) time where h is the height Case 3: The node, x to be deleted has two children of the tree • If n is the number of nodes in the tree, in the worst case, h1. Swap x with Successor(x) (Successor(x) has no more than 1 is O(n)• However, if we can keep the tree balanced, we can ensure child (why?)) that  $h = O(\log n)$ 2. Remove x, using the procedure for case 1 or case 2. • Red-Black trees can maintain a balanced BST 20 21 Randomly Built BST \_\_\_\_\_ \_ Analysis \_\_\_\_\_ • What if we build a binary search tree by inserting a bunch of "Shut up brain or I'll poke you with a Q-Tip" - Homer Simpson elements at random? • Q: What will be the average depth of a node in such a • Note that the average depth of a node in T is randomly built tree? We'll show that it's  $O(\log n)$ • For a tree T and node x, let d(x,T) be the depth of node x  $\frac{1}{n}\sum_{x\in T}d(x,T) = \frac{1}{n}P(T)$ in T• Define the total path length, P(T), to be the sum over all • Thus we want to show that  $P(T) = O(n \log n)$ nodes x in T of d(x,T)

Analysis \_\_\_\_\_

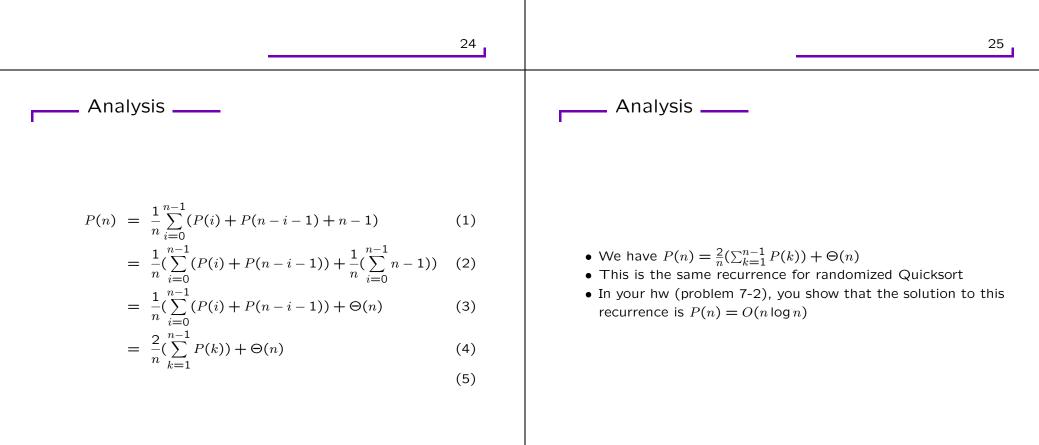
\_\_\_ Analysis \_\_\_\_\_

- Let  $T_l$ ,  $T_r$  be the left and right subtrees of T respectively. Let n be the number of nodes in T
- Then  $P(T) = P(T_l) + P(T_r) + n 1$ . Why?

- Let P(n) be the expected total depth of all nodes in a randomly built binary tree with n nodes
- Note that for all i,  $0 \le i \le n 1$ , the probability that  $T_l$  has i nodes and  $T_r$  has n i 1 nodes is 1/n.

27

• Thus  $P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n-1)$ 



26

Take Away \_\_\_\_\_ Take Away \_\_\_\_\_ • P(n) is the expected total depth of all nodes in a randomly • The expected average depth of a node in a randomly built built binary tree with n nodes. binary tree is  $O(\log n)$ • We've shown that  $P(n) = O(n \log n)$ • This implies that operations like search, insert, delete take • There are *n* nodes total expected time  $O(\log n)$  for a randomly built binary tree • Thus the expected average depth of a node is  $O(\log n)$ 29 28 \_ What to do? \_\_\_\_\_ Warning! • In many cases, data is not inserted randomly into a binary • A Red-Black tree implements the dictionary operations in search tree such a way that the height of the tree is always  $O(\log n)$ , • I.e. many binary search trees are not "randomly built" where n is the number of nodes • For example, data might be inserted into the binary search • This will guarantee that no matter how the tree is built that tree in almost sorted order all operations will always take  $O(\log n)$  time • Then the BST would not be randomly built, and so the • Next time we'll see how to create Red-Black Trees expected average depth of the nodes would not be  $O(\log n)$