CS 361, Lecture 18

Jared Saia
University of New Mexico

Outline

- Class Evaluation
- Binary Trees

Binary Search Tree Property

- Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y)≤key(x). If y is a node in the right subtree of x then key(x)≤key(y)

Search in BT

```
Tree-Search(x,k){
  if (x=nil) or (k = key(x)){
    return x;
  }
  if (k<key(x)){
    return Tree-Search(left(x),k);
  }else{
    return Tree-Search(right(x),k);
  }
}
```

Analysis

- Let h be the height of the tree
- The run time is $O(h)$
- Correctness???

Previous In-Class Exercise

- Q1: What is the loop invariant for Tree-Search?
- Q2: What is Initialization?
- Q3: Maintenance?
- Q4: Termination?

Loop Invariant Review

A useful tool for proving correctness is loop invariants. Three things must be shown about a loop invariant

- **Initialization:** Invariant is true before first iteration of loop
- **Maintenance:** If invariant is true before iteration i, it is also true before iteration $i + 1$
- **Termination:** When the loop terminates, the invariant gives a property which can be used to show the algorithm is correct

Loop Invariant Review

- When **Initialization** and **Maintenance** hold, the loop invariant is true prior to every iteration of the loop
- Similar to mathematical induction: must show both base case and inductive step
- Showing the invariant holds before the first iteration is like the base case. Showing the invariant holds from iteration to iteration is like the inductive step

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:
  1. 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*

Answers

- 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

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).

Termination

- 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.

Tree Min/Max

- Tree Minimum(x): Return the leftmost child in the tree rooted at x
- Tree Maximum(x): Return the rightmost child in the tree rooted at x

Successor

- The successor of a node x is the node that comes after x in the sorted order determined by an in-order tree walk.
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x
- We can compute the successor of a node in $O(\log n)$ time

Tree-Successor

```
Tree-Successor(x){
  if (right(x) != null){
    return Tree-Minimum(right(x));
  }
  y = parent(x);
  while (y!=null and x=right(y)){
    x = y;
    y = parent(y);
  }
  return y;
}
```


Successor Intuition

- Case 1: If right subtree of x is non-empty, successor(x) is just the leftmost node in the right subtree
- Case 2: If the right subtree of x is empty and x has a successor, then successor(x) is the lowest ancestor of x whose left child is also an ancestor of x. (i.e. the lowest ancestor of x whose key is $\geq$ key(x))

Insertion

Insert(T,x)

1. Let r be the root of T.
2. Do Tree-Search(r,key(x)) and let p be the last node processed in that search
3. If p is nil (there is no tree), make x the root of a new tree
4. Else if key(x) $\leq$ p, make x the left child of p, else make x the right child of p

Deletion

- Code is in book, basically there are three cases, two are easy and one is tricky
- Case 1: The node to delete has no children. Then we just delete the node
- Case 2: The node to delete has one child. Then we delete the node and "splice" together the two resulting trees

Case 3

Case 3: The node, x to be deleted has two children

1. Swap x with Successor(x) (Successor(x) has no more than 1 child (why?))
2. Remove x, using the procedure for case 1 or case 2.

Analysis

- All of these operations take $O(h)$ time where h is the height of the tree
- If n is the number of nodes in the tree, in the worst case, h is $O(n)$
- However, if we can keep the tree *balanced*, we can ensure that $h = O(\log n)$
- Red-Black trees can maintain a balanced BST

Randomly Built BST

- What if we build a binary search tree by inserting a bunch of elements at random?
- Q: What will be the average depth of a node in such a 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 in T
- Define the total path length, $P(T)$, to be the sum over all nodes x in T of $d(x, T)$

Analysis

"Shut up brain or I'll poke you with a Q-Tip" - Homer Simpson

- Note that the average depth of a node in T is
$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T)$$
- Thus we want to show that $P(T) = O(n \log n)$

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?

Analysis

- 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, $0 \leq i \leq n-1$, the probability that T_l has i nodes and T_r has $n-i-1$ nodes is $1/n$.
- Thus $P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1)$

Analysis

$$
\begin{aligned}
P(n) &= \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) && (1)\\
&= \frac{1}{n}(\sum_{i=0}^{n-1}(P(i) + P(n-i-1)) + \frac{1}{n}(\sum_{i=0}^{n-1} n - 1)) && (2)\\
&= \frac{1}{n}(\sum_{i=0}^{n-1}(P(i) + P(n-i-1)) + \Theta(n) && (3)\\
&= \frac{2}{n}(\sum_{k=1}^{n-1} P(k)) + \Theta(n) && (4)\\
& && (5)
\end{aligned}
$$

Analysis

- We have $P(n) = \frac{2}{n}(\sum_{k=1}^{n-1} P(k)) + \Theta(n)$
- This is the same recurrence for randomized Quicksort
- In your hw (problem 7-2), you show that the solution to this recurrence is $P(n) = O(n \log n)$

Take Away

- $P(n)$ is the expected total depth of all nodes in a randomly built binary tree with n nodes.
- We've shown that $P(n) = O(n \log n)$
- There are n nodes total
- Thus the expected average depth of a node is $O(\log n)$

Take Away

- The expected average depth of a node in a randomly built binary tree is $O(\log n)$
- This implies that operations like search, insert, delete take expected time $O(\log n)$ for a randomly built binary tree

Warning!

- In many cases, data is not inserted randomly into a binary search tree
- I.e. many binary search trees are not "randomly built"
- For example, data might be inserted into the binary search tree in almost sorted order
- Then the BST would not be randomly built, and so the expected average depth of the nodes would not be $O(\log n)$

What to do?

- A Red-Black tree implements the dictionary operations in such a way that the height of the tree is always $O(\log n)$, where n is the number of nodes
- This will guarantee that no matter how the tree is built that all operations will always take $O(\log n)$ time
- Next time we'll see how to create Red-Black Trees