CS 361, Lecture 19

Jared Saia
University of New Mexico

Outline

- Deletion in BSTs
- Probability Review
- Randomly built BSTs

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.

Example

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 $O(\log n)$

Probability Review

- We want to answer the question: "What will be the average depth of a node in a randomly built tree?"
- We can define a *random variable* which gives the depth of a node chosen uniformly at random in the tree.
- We want to compute the *expectation* of this random variable.
- (Note: Appendix C in the book gives a good review of probability theory. If you are confused, make sure you *read this appendix*)

Random Variable

- Recall that a random variable is a function from a sample space to the real numbers
- It associates a real number with each possible outcome of an experiment.
- For a random variable X and a real number x, $P(X = x)$ is the probability that the random variable X takes on the value x.

Example

- Consider the experiment of rolling two 6-sided die.
- There are 36 possible outcomes of this experiment $(6 * 6)$
- Define the *random variable* X to be the maximum of the two values showing on the dice
- Then we can say that $P(X = 3) = 5/36$ since X assigns the value of 3 to 5 of the 36 possible outcomes ((1,3),(2,3),(3,3),(3,2),(3,1))

Expectation

- A simple and useful summary of the distribution of a random variable is the "average" of the values it takes on
- The *expectation* (or *expected value*) of a random variable X is:

$$E(X) = \sum_x x * P(X = x)$$

Example

- Consider a game where you flip two coins
- You earn \$3 for each head but lose \$2 for each tail.
- Let X be a random variable representing your earnings. The expected value of X is

$$\begin{aligned} E(X) &= 6 * P(\text{2 H's}) + 1 * P(\text{1 H, 1 T}) - 4 * P(\text{2 T's}) \\ &= 6 * (1/4) + 1(1/2) - 4(1/4) \\ &= 1 \end{aligned}$$

Our Problem

- We want to answer the question: "What will be the average depth of a node in a randomly built tree?"
- Define the random variable X to be the depth of a node chosen uniformly at random in the tree
- X takes on n possible values, it takes on each value with probability $1/n$

Our Problem

- 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 $d(x,T)$
- Then

$$\begin{aligned} E(X) &= \frac{1}{n}\sum_{x\in T} d(x,T) \\ &= \frac{1}{n}P(T) \end{aligned}$$

- Thus we want to show that $P(T) = O(n \log n)$

Analysis

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

- 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 as the recurrence for randomized Quicksort
- Recall from hw problem 7-2, 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