Analysis of Algorithms I: Binary Search Trees

Xi Chen

Columbia University

Hash table: A data structure that maintains a subset of keys from a universe set $U = \{0, 1, \ldots, p-1\}$ and supports all three dictionary operations: Insert, Delete and Search. Has very good performance: Universal hashing and Perfect hashing. But what if we need a data structure that supports additional operations:

1. Minimum $(\cdot)$: return the smallest key;
2. Maximum $(\cdot)$: return the largest key;
3. Successor, Predecessor, Nearest matches: to be defined later

Starting from this class, we study binary search trees that support all these operations. We show that all these operations can be done in time linear in the height h of the tree.

Binary tree: A tree in which each internal node has at most two child nodes, usually referred to as its left and right child node. How to represent a binary tree? Check Section 10.4 of the text book. Each node x in the tree consists of three pointers:

1. $x.p$ points to its parent node (nil if x is the root);
2. $x.\ell$ and $x.r$ point to its left and right child nodes; nil if x has no left or right child; e.g., if x is a leaf then both are nil.

Binary search tree: A binary tree in which each node x has a key $x.k$ from the universe set U (or a pointer that leads to an object that has a key from U). Moreover, the keys satisfy the following binary-search-tree property: Let x be any node in the tree. Then for any node y in the left subtree of x, we have $y.k \leq x.k$. For any node z in the right subtree of x, we have $x.k \leq z.k$.

Note that we allow duplicate keys in a binary search tree, which sometimes makes things more complicated. We usually use n to denote the number of nodes and h to denote the height of the tree being considered.

Height of a binary search tree: length of the longest path from the root to a leaf. This parameter is crucial to the worst-case performance of a binary search tree. Note that two binary search trees may have exactly the same set of keys but have extremely different shape, one balanced and the other unbalanced (see the example on Page 287). In general, a binary search tree can be extremely unbalanced (e.g., a tree consists of a single path of length $n-1$) and has a height much larger than $O(\lg n)$ and thus, has bad worst-case performance. Now we start to describe operations that a binary search tree supports with $O(h)$ worst-case running time. In the next class, we will see how to maintain a balanced binary search tree with height bounded by $O(\lg n)$.

Let v be a node in the tree. Start with Min and Max:
Min(v): find a node in the subtree of v with the smallest key

1. while $v.\ell \neq$ nil
2. do $v = v.\ell$
3. return v

Keep moving to the left until v has no left child.

Max(v): find a node in the subtree of v with the largest key

1. while $v.r \neq$ nil
2. do $v = v.r$
3. return v

Symmetric. Just keep moving to the right child.

Given a node v, the keys in its subtree can be sorted in linear time:
Inorder(v): visit each node in the subtree of v in the sorted order

1. If $v.\ell \neq$ nil
2. Inorder$(v.\ell)$
3. print $v.k$
4. If $v.r \neq$ nil
5. Inorder$(v.r)$

The running time is linear in the number of nodes in the subtree of v. Check Theorem 12.1 on page 288. Use the following recurrence:

$$T(n) \leq \max_{0 \leq k \leq n} \Big(T(k) + T(n-k-1) \Big) + O(1)$$

and the substitution method.

Search (v, k): Find a node with key $= k \in U$ in the subtree of v

1. If $v.k = k$, return v
2. If $k < v.k$
3. If $v.\ell =$ nil, return nil; otherwise return Search $(v.\ell, k)$
4. If $k > v.k$
5. If $v.r =$ nil, return nil; otherwise return Search $(v.r, k)$

Check page 291 for an implementation without using recursions.

The following two operations assume that the keys are distinct. Successor(v): find the node with the smallest key $> v.k$. First, if the right subtree of v is nonempty, then the successor of v is the leftmost node in v's right subtree (or equivalently, the node with the smallest key in v's right subtree). Now if $v.r = \text{nil}$, examine the path from v back to the root. Find the lowest ancestor u of v whose left child is also an ancestor of v, and u is the successor of v. I leave both proofs of correctness as exercises. If no such u exists, it means v has the largest key in the tree and has no successor.

Deletion is easy (but keep in mind that we need to maintain the binary-search-tree property). If we want to delete a node v with no child, just remove it. If v has one child, then remove v and connect v's parent node with v's child. If v is the left child of $v.p$ then v's child becomes the new left child of $v.p$, and vice versa. The tricky case is when v has both left and right child nodes.

For this case, let $u = v.r$. We have $u \neq \text{nil}$ because v is assumed to have two children. Next, starting from u, keep going left until reaching a node w with no left child. Check that w has the smallest key in the subtree of u. Finally we remove w (using one of the first two cases because w does not have left child), and then replace the key of u by the key of w. Verify the correctness as how the binary-search-tree property is maintained.

Insertion is easier. We start by searching for the input key k in the binary tree. This finally leads us to a leaf u. If $u.k > k$, create a new node v as the left child of u and set $v.k = k$. If $u.k \leq k$, create a new node v as the right child of u and set $v.k = k$. Again, verify the correctness that the binary-search-tree property is well maintained.

While both insertion and deletion are easy, they may lead to highly unbalanced trees and they are responsible for very bad worst-case performance. For example, start with an empty tree and insert $1, 2, \ldots, n$. This results in a tree (a path actually) of height $n - 1$. In the next class we will see how to insert and delete "carefully" in order to maintain a balanced tree with $O(\lg n)$ height. This gives us a data structure that supports all operations in worst-case $O(\lg n)$ running time.