Binary Search Trees (8)

1 BINARY SEARCH TREES

1.1 Binary-Search-Tree Definition

An n-node binary tree.
Not necessarily complete.
All nodes j (other than leaves) satisfy the "binary-search-tree property": $\text{key}(i) \leq \text{key}(j) \leq \text{key}(k)$. Here, i is in j's left subtree, and k is j's right subtree.
In terms of the array representation: $T_v[i] \leq T_v[j] \leq T_v[k]$.
Example...
Note: not enough that the property just holds for immediate children. Must hold for the entire family line.

1.2 Some Properties of Binary Search Trees

Height is $O(n)$ (imbalanced) and $\Omega(\log n)$ (balanced).
Where are largest and smallest elements?
In what sense does each position in the tree represent a range?

1.3 Finding a Key

Search for node with key x. Start with i as the root.

```
typedef int** Tree;

#define NULL 0

int value = 0;
int parent = 1;
int left = 2;
int right = 3;

int Find( Tree T, int x, int i ) {
 if( i == NULL ) {
   return NULL;
 }
```

```
 if( x == T[value][i] ) {
   return i;
 }
 if( x < T[value][i] ) {
   return Find(T, x, T[left][i]);
 }
 return Find(T, x, T[right][i]);
}
```

Running time in terms of height of tree? How does this compare to doing the same thing in a heap?
How find minimum? Running time? Compared to heap?
Easily extended to insert x if it is not found.

1.4 Listing Items in Order

```
void Sort_Tree( Tree T, int i ) {
 if( i == NULL ) {
   return ;
 }
 Sort_Tree(T, T[left][i]);
 cout << T[value][i] << endl;;
 Sort_Tree(T, T[right][i]);
 return;
}
```

Running time? How prove this?

1.5 Where is the Successor?

We can derive a simple (?) rule for determining the node in the binary search tree that immediately follows node i in the sorted order (returns `NULL` for max element in tree):

```
int Successor( Tree T, int i ) {
  int j;
  if( T[right][i] != NULL ) {
    j = T[right][i];
    while( T[left][j] != NULL ) // T[leftcolor][j] = black;
      j = T[left][j];
    return j;
  }

  j = T[parent][i];
  while( T[parent][j] != NULL && j == isrightchild(T, j))
    j = T[parent][j]; // T[rightcolor][j] = black;
```

```
  return T[parent][j];
}
```

Running time in terms of the height of the tree?
How do test for being a right child?
How could this be used to delete an element from the tree?
How could this be used to sort?

1.6 Deleting

We want to delete a node i.

- If i has zero child, delete it.
- If i has one child, delete i and move i's child into i's place.
- If i has two children, let j be the successor of i. Delete j in place (it has at most one child, so that's easy). Now, move j into i's place.

Why is this well defined? In particular, how do we know that j has at most one child?

1.7 Successor Tree Walk

```
void Sort_Tree_Succ( Tree T, int i ) {
 i = Find_Min(T, i);

 while (i != NULL) {
  cout << T[value][i] << endl;;
  i = Successor( T, i );
 }
}
```

How implement `Find_Min`?
How analyze the algorithm?

1.8 Successor Tree Walk: Analysis

$$\Phi(T, i) = n - \text{rank}(i) + \sum_j (\chi(T_{lc}[j] = \text{white}) + \chi(T_{rc}[j] == \text{white}))$$

Each operation increments the rank of i, and, on each iteration of a while loop, makes at least one edge "black". (This can include "Find Min" as well.)
Upper bound on maximum value of potential function is $3n$.

1.9 Successive Insertions and Deletions

We know that nearly balanced trees are best because find, insertion, and deletion all run in $O(h)$, which is $O(\log n)$ if the tree isn't too stringy.
But, even if we start off with a nice balanced tree, it might not be balanced anymore after a sequence of insertions and deletions.
Sugar Pine: unbalanced
Coulter Pine: balanced

2 ROTATIONS

2.1 Complete Binary Search Tree

Given a list of numbers, we could create a well-balanced binary search tree.

- Sort A.
- Pick i as the appropriate halfway point. Let i be the root.
- Make a complete binary search tree out of $A[1]$ through $A[i-1]$ and make it the left subtree. Do the same with $A[i+1]$ through $A[n]$ and make it the right subtree.

If we choose i to be $\lfloor (n-1)/2 \rfloor$, what is the height of the tree? How prove?
Running time?
How can we choose i so that the resulting tree is complete (i.e., deepest level is "left justified")?

2.2 Recovering from Bad Luck

Sorting is a rather drastic way to make a binary search tree balanced.
Sometimes we can get by with more "local" adjustments.

2.3 Rotation

Swap a node and its parent.
Fix links to the kids to maintain the binary-search-tree property.
HW: Write pseudocode.
Running time in terms of the height of the tree?
What happens to the depths of nodes?

2.4 Nearly Balanced Trees

There are a collection of algorithms that use special rules to decide which rotations to apply when inserting and deleting elements to ensure that the binary search tree stays *nearly* balanced.

Nearly balanced means, for example, that the number of nodes in the left or right subtree is never fewer than 1/3 of the total number of nodes. This holds for all levels.
$T(n) \leq T(2n/3) + 1$... what is $T(n)$?
Example algorithms: red-black trees, 2-3 trees.
Insertion, deletion, find, all in $O(\log n)$, worst case.

2.5 A Tradeoff

The worst-case algorithms are complicated to implement but relatively simple to analyze. Next we'll talk about splay trees... relatively simple to implement, but harder to analyze. Gets us $O(\log n)$ insert, delete, and find, but only averaged over a sequence of operations. But good constant factors and simple implementation is a win.

3 LISTING LARGEST ELEMENTS

Let's think about solving the following problem. A web search engine matches queries against all the documents in a database and computes a score for each. It then needs to present the best documents to the user, in order.

Step 0 : Formalize problem.

Step 1 : Propose algorithms.

Step 2 : Prove correctness.

Step 3 : Prove running-time bounds.