CS 61B Data Structures and Programming Methodology

Aug 7, 2008

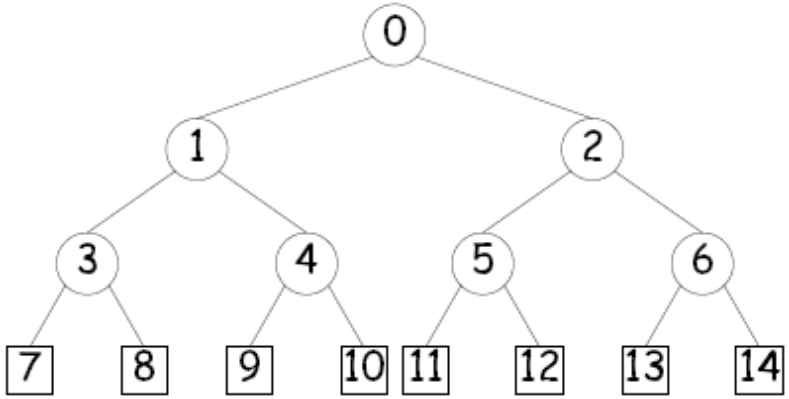
David Sun

Announcements

- Final will be held next Thursday 14th of Aug. Exact time TBA.
- Two review sessions next week, Tuesday and Wednesday during class with mock-exams.
- If you are taking CS61C and plan to take the CS61C exam on 14th of Aug, please send me an email with your name and student ID.

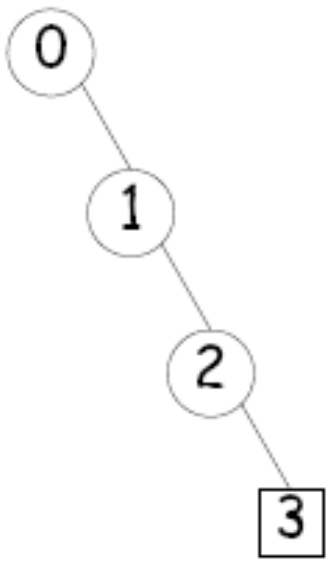
Recap: Binary Search Tree Running Times

- In a perfectly (full) balanced binary tree with height/depth *h*, the number of nodes $n = 2^{(h+1)} - 1$.
- Therefore, no node has depth greater than $log_2\ n$.
- The running times of `find()`, `insert()`, and `remove()` are all proportional to the depth of the last node encountered, so they all run in *O(log n)* worst-case time on a perfectly balanced tree.



Recap: Binary Search Tree Running Times

- What's the running time for this binary tree?
- The running times of `find()`, `insert()`, and `remove()` are all proportional to the depth of the last node encountered, but $d = n - 1$, so they all run in $O(n)$ worst-case time.



Recap: Binary Search Tree Running Times

- The Middle ground: reasonably well-balanced binary trees
  - Search tree operations will run in *O(log n)* time.
- You may need to resort to experiment to determine whether any particular application will use binary search trees in a way that tends to generate balanced trees or not.

2-3-4 Trees

- A 2-3-4 tree is perfectly balanced:
  - `find`, `insert`, and `remove` operations take *O(log n)* time, in the worst case.
- Every node in the tree has 2, 3, or 4 children, except leaves, which are all at the bottom level of the tree.
  - Each node stores 1, 2, or 3 entries, which determine how other entries are distributed among its children's subtrees.
- Each internal (non-leaf) node has one more child than keys.
  - For example, a node with keys [20, 40, 50] has four children.
- 2-3-4 tree invariant:
  - Eack key *k* in the subtree rooted at the first child satisfies *k <= 20*; at the second child, *20 <= k <= 40*; at the third child, *40 <= k <= 50*; and at the fourth child, *k >= 50*.

`find`

1. Start at the root. At each node, check for the key *k*. If *k* is found, we are done.
2. If it's not present, and if *k* < *k1,* where *k1* is the first key stored in the node, then look in the subtree rooted at *k1.* Otherwise, if *k1* < *k* < *k2,* then look in the subtree rooted at *k2.* Contin

`insert`

1. `insert()` walks down the tree in search of the key *k* (like `find()`).
2. If it find an entry with key *k*, it proceeds to that entry's "left child" and continues.
3. When `insert()` encounters a 3-key node, the middle key is ejected from the current node, and placed in the parent node. The other two keys in the 3-key node are split into two separate 1-key nodes, which are divided underneath the old middle key.

`insert`

- The splitting of every 3-key node is to make sure that:
  - there's room for the new key in the *leaf* node (if splitting happens at a leave node)
  - there's room for the ejected keys that are moved into the the *internal* nodes (if splitting happens at an internal node).
- Insert may increase the depth of the tree by one if the root of the tree was a 3-key node.

`remove`

1. Find a node containing key *k* using the same algorithm as `find()`.
2. Return `null` if *k* is not in the tree;
3. If *k* is found in a leaf node, remove it.
4. If *k* is found in an internal node, replace it with the entry with the *next higher key*.

`remove`

- Remove could potentially empty a 1-key leaf node, so we need to *eliminate* all 1-key nodes by turning it into either a 2-key node through *rotation* or a 3-key node through *fusion.*
- *Case 1 if there is an adjacent sibling with more than 1 key:*
  - Steal a key from this adjacent sibling.
  - To maintain the 2-3-4 tree invariant, perform a *rotation* by moving a key from the sibling to the parent and moving a key from the parent to the 1-key node.
  - If the moved key in the sibling has a subtree *S,* then that subtree is moved to the 1-key node (now a 2-key node).

`remove`

- *Case 2 if no adjacent siblings have more than 1 child:*
  - Steal a key from the parent and *fuse* the 1-key node, the key from the parent node, and the 1-key sibling node into one node.
  - Since the parent must be visited before the current node, it must have at least 2 keys (unless it's the root).

`remove`

- *Case 3 if the parent is the root and contains only one key, no adjacent sibling has more than one key:*
  - *Fuse* the current 1-key node, its 1-key sibling, and the 1-key root into one 3-key node that serves as the new root.
  - The only case where the depth of the tree decreases by one.

Running Time

- A 2-3-4 tree with depth *d* has between 2^d (if all the nodes are 1-key nodes) and 4^d (if all the nodes are 3-key nodes) leaves.
- If *n* is the total number of nodes, then $n >= 2^{(d+1)} - 1$. Taking the logarithm of both sides, *d* is in *O(log n)*.
- 2-3-4 tree operations are more complicated than corresponding operations for binary search trees, so the time required to process a *single* node is longer, but still in *O(1)* time.
- The number of nodes visited is proportional to the depth of the tree.
  - Hence, the running times of the `find()`, `insert()`, and `remove()` operations are in *O(d)* and hence in *O(log n)*, even in the worst case.
  - Compare this with the *Theta(n)* worst-case time of ordinary binary search trees.

Splay Tree

- A splay tree is a type of *balanced binary search tree*. Structurally, it is identical to an ordinary binary search tree; the only difference is in the algorithms for finding, inserting, and deleting entries.
- All splay tree operations run in *O(log n)* time on average, where *n* is the number of entries in the tree. Any single operation can take *Theta(n)* time in the worst case. But any sequence of *k* splay tree operations, with the tree initially empty and never exceeding *n* items, takes *O(k log n)* worst-case time.

Rotation

- Rotation is a procedure that keeps splay tree (and other balanced search trees) balanced.
- There are two types -- a left rotation and a right rotation -- and each is the other's reverse.
- The rotation procedure preserves the *binary search tree invariant*, i.e. after the rotation, the left subtree is less than or equal to the root and the right subtree is more than or equal to the root.

`find`

1. Walk down the tree until we find the entry with key k, or reach a dead end (just like binary search tree).
2. In both cases, splay the last visited node *X* (where the search ended) up the tree to become the new root via a sequence of rotations.
   1. Recently accessed entries are near the root of the tree, so if you access the same few entries repeatedly, accesses will be very fast.
   2. If *X* lies deeply down an unbalanced branch of the tree, the splay operation will improve the balance along that branch.

`find`

- Case 1 *X* is the right child of a left child
  - let *P* be the parent of *X*, and let *G* be the grandparent of *X*, first rotate *X* and *P* left and then rotate *X* and *G* right.
  - called the *zig-zag.*
- Case 1.1 *X* is a left child and *P* is a right child:
  - rotate *X* and *P* right, then *X* and *G* left.
- Case 2 *X* is the left child of a left child
  - start with the grandparent, and rotate *G* and *P* right. Then, rotate *P* and *X* right.
  - called the *zig-zig.*
- Case 2.1 *X* is the right child of a right child
  - rotate *G* and *P* left. rotate *P* and *X* left.

`find`

- Repeatedly apply zig-zag and zig-zig rotations to *X*.
  - each pair of rotations raises *X* two levels higher in the tree.
  - eventually, either *X* will reach the root (and we're done), or *X* will become the child of the root
- Case 3 *X's* parent *P* is the root:
  - rotate *X* and *P* so that *X* becomes the root. This is called "zig" case.

Readings

- Objects, Abstractions, Data Structures and Design
  - Chapter 11.1 – 11.3