CS109A Notes for Lecture 2/28/96

Binary Trees

Every binary tree has two "slots" for children; it may have none, either one, or both.

- *Empty* (0-node) binary tree is possible.
- Equivalently, a node has left and right subtrees. Either or both may be null.

Data Structure for Binary Trees

A node is a record structure; a binary tree is a pointer to a node.

- `NULL` pointer represents the empty tree; otherwise a tree is a pointer to its root node.
- Nodes have fields **`leftchild`**, **`rightchild`**.
    - □ These point to roots of left/right subtrees, (left/right children).
    - □ They are null if left/right subtree is `NULL` (left/right child does not exist).
- Other fields within nodes are possible, e.g., label, pointer to parent.

Structural Induction on Binary Trees

- One important difference: basis is the empty tree, not a tree of one node.

Example:

- $S(T)$: In a binary tree T represented by left- and right-child pointers, there is one more `NULL` pointer than node.

Basis: If T is the empty tree, then there is a `NULL` pointer that represents the tree as a whole. There are no nodes, so $S(T)$ holds for $T =$ empty tree.

Induction: Let T not be empty and have left and right subtrees L and R.

- By the IH, L and R each have one more `NULL` than node. "Excess" $= 2$.

- However, T also has its root node, so the excess for T is 1, proving $S(T)$.

Binary Search Trees

- Labels at nodes, ordered by some $<$ comparison operator, e.g., ints, reals, strings.

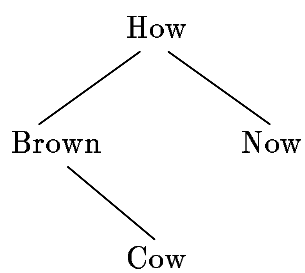
- If a node has label x, then every label in the right subtree is $> x$, and every label in the left subtree is $< x$.

- Supports *dictionary* = set with operations lookup, insert, delete.

  - ☐ Running time $= O(\log n)$ per operation on the average; $n =$ size of set.

- Supports *range queries* = find values between upper and lower limits.

Example:



Lookup

- Key point: label at root tells us which half of the tree we must search, either left or right.

  - ☐ Thus, on the average, we cut the size of the tree to search almost in half in $O(1)$ time. After average $O(\log n)$ steps, we are down to 1 element and are done.

- Searching for x at tree T:

Basis:

1. If T is empty, fail; x is not there.

2. If T has label x at the root, then found.

Induction: Let T have root label y. If $x < y$, lookup x on the left subtree of the root; if $x > y$ lookup x on the right subtree.

Insertion

Two approaches in C:

1. Insertion function gets tree (pointer to node) as argument and returns a revised tree including inserted element.

2. Insertion function gets pointer to tree (pointer to pointer to node) as argument and, when it needs to insert, creates a new node and makes the slot pointed to by its argument point to the new node.

- We'll sketch (1), typified by code in Fig. 5.35; (2) is typified by code of Fig. 5.38 for "delete."

Basis:

1. If T is `NULL` create a new node with label x and return a pointer to that node.

2. If x at root, no action needed so return T.

Induction: Let root of T have label y. If $x < y$, insert x into left subtree. The left-subtree pointer at the root of T becomes whatever tree is returned by recursive call. If $x > y$, do analogously at right.

Deletion

To delete x from tree T:

Basis: If T is empty, just return T; if x at root, delete root, fix up T (explained next), and return the fixed-up T.

Induction: If T has label y at root, delete x from left/right subtree if $x < y/x > y$. Replace left/right pointer by returned tree, and return the resulting T.

Fixup (Deletemin)

- If we need to delete the root of T, if it has one `NULL` subtree, just return the other subtree (even if it too is `NULL`).

- Otherwise, find the least element in the right subtree (by going down the leftmost path) and move it to the root of T.