CS 42—Trees

Thursday, October 18, 2018

Summary

Today, we'll learn about another inductive data structure: **trees**. The main reason to learn about trees is to see another example of how recursive algorithms and inductive data structures work together. We'll also see examples of analysis, using recurrence relations.

Trees—especially **balanced Binary Search Trees (BSTs)**—are important data structures in computer science, and you'll see more of them if you go on to learn more CS.

Trees: terminology and properties

tree: an inductive data structure that has a unique **root** (at the top of the tree).

A non-empty tree is made of **nodes**, connected by **edges**. The edges are directed: they communicate a one-way relationship between the **parent** node and its **children**. In a tree, every node has exactly one parent, except the root node, which has no parent. Two nodes are **siblings** if they share a parent.

Each node has a **key**—the value stored in that node.

A **leaf** is a node with no children.

A **path** is a sequence of zero or more edges from one node to another. A tree's structure means that there is one and only one path from the root to each node in the tree.

The **ancestors** of a node N are are the nodes along the path from N to the root. The **descendants** of a node N are all the nodes along paths from N to leaves of the tree. A **subtree** of a tree T is a node in T and all of its descendants.

The **height** of a tree is the length of (i.e., the number of edges in) the longest path from the root to a leaf. The height of an empty tree is -1 (although, this is not a universally held belief).

The **depth** of a node N is the length of the path from the root to N.

A **binary tree** is a tree where every node has at most two children.

A **binary search tree (BST)** is a binary tree where every key in a node N's left subtree is less than N's key and every node in N's right subtree is greater than N's key. We will assume that the BST contains no duplicate keys.

A **balanced binary search tree** is a BST where every subtree is about about the same size as its sibling.

A **perfect binary tree** is one where all the leaves have the same depth and all non-leaf nodes have two children.

interface: the operations that supported by a data structure—the "what" of the data structure

implementation: the simpler data structures and the algorithms used to make the structure's interface available—the "how" of the data structure

BST algorithms

Traversals

A **traversal** of a tree is an algorithm that visits all nodes in the tree. Here are three kinds of traversals:

preorder: Visit the root, preorder traverse its left subtree, preorder traverse its right subtree

inorder: Inorder traverse the left subtree, visit the root, inorder traverse the right subtree.

postorder: Postorder traverse the left subtree, postorder traverse the right subtree, visit the root

`find`

Given a BST *values* and a number *i*:

find(i, values):

  If the tree is empty, return false.

  Let key be the value at the root of the tree.

  If key is i, return true.

  If i < key, call find on the left subtree.

  If i > key, call find on the right subtree.

`insert`

Given a BST *values* and a number *i* (which is not yet in the BST):

insert(i, values):

  Look for i in values.

  Insert i as a leaf where it should be.

Worst-case inputs

Worst-case inputs for an algorithm are the "pathological" ones: the inputs that would be the most expensive to compute.

We usually *don't* think of worst-case inputs in terms of size. Instead, the size is fixed, and we talk about other qualities of the input that make it bad. For example, the worst-case tree of size N is a "stick" of size N, i.e., a list.

When reasoning about a problem, we often like to ask: "What's the cost for the worst-case input to this problem?" because the answer gives us a sense of how hard the problem might be.

Next Tuesday: No class—take a break!