CMSC 420: Trees

Justin Wyss-Gallifent

September 6, 2023

1 Overview

We look at trees, lots of them, because trees are not just useful but because they form a foundation for many other data structures.

2 Types of Trees

We mention a few types of trees that we'll see frequently.

- Binary trees are trees in which each node has at most two children.
- Complete binary trees are binary trees in which every level is full, except possibly the lowest level, and in the lowest leval all the nodes are on the left.
- Perfect binary trees are binary trees in which every level is full. Note that a perfect binary tree will have $2^k - 1$ nodes for some k.
- Binary search trees are binary trees in which the left subtree of a node contains only keys smaller than the node's key and the right subtree of a node contains only keys larger than the node's key.

3 Tree Definitions

Definition 3.0.1. The *height* of a tree equals the number of generations minus one. For example if there is only a root and its children then the height is 1, whereas a tree with a root, its children, and its grandchildren then the height is 2. If the tree consists of just one node then the height is 0. and for consistency an empty tree (no nodes) will have height -1. This may seem odd but it makes many calculations cleaner.

Definition 3.0.2. The term *level* is not well-defined when it comes to trees. Sometimes the levels go from the top down and start at 0 whereas sometimes they go from the top down and start at 1. In some trees (like red-black and AA trees) are even weirder - the levels go from the bottom up and start at 1, except the null children of the leaves have level 0, and sometimes nodes are the same level as their children.

Which is to say make sure you know what *level* means in a particular context.

4 Tree Storage

There are many ways to store trees. A few examples are:

- Using child pointers. This is perhaps the most common way to represent a tree. For trees in which a node can have some maximum number of children we can define each node as having that many children and use

Null pointers to indicate when a child is missing. However for trees in which a node can have any number of children this won't work.

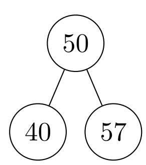
- If a node can have any number of children we could use a list, or a linked list, with each entry pointing to a child.
- Using parent pointers. In such a case each node will have a pointer to its parent and the root's parent pointer will be NULL. These are used in disjoint set data structures.
- Using a nested object, such as with JSON.
- A binary tree can be stored in a list where each node has a corresponding index in the list. Typically the binary tree will be 1-indexed with the root node being index 1 and the corresponding list (typically 0-indexed) will simply not use the 0 entry.

5 Binary Tree Traversals

Suppose we want to traverse a binary tree. In CMSC351 we saw how to do this with a breadth-first traversal and a depth-first traversal. Here are three new ways. All of these are mostly clearly managed with recursion.

- Preorder Traversal: Visit the root, then preorder traverse T_L and then T_R. Preorder traversal is useful for copying the tree since it generates the root node first, which we needs so that we can attach any children.
- Postorder Traversal: Postorder traverse T_L and then T_R and then the root. Postorder traversal if we need to delete an entire tree node-by-node, since it deletes the children before their parent in a sensible way for cleanup.
- Inorder Traversal: Inorder traverse T_L then the root then T_R. We will see that for binary search trees the inorder traversal yields the keys in increasing order.

Example 5.1. Here is a really easy example. Consider the tree shown here:

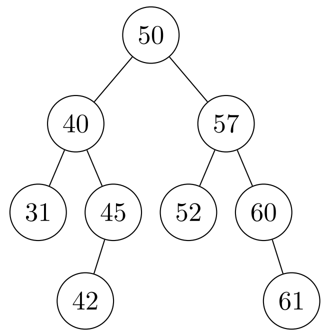


We have the following:

- Preorder: 50,40,57
- Postorder: 40,57,50
- Inorder: 40,50,57

Note that this is a binary search tree (if you're not sure what this is, that's fine) and the inorder traversal is in increasing order.

Example 5.2. Consider the tree shown here:



We have the following. In this example we have also included BFT and DFT with the assumption that left links are followed first.

- Preorder: 50,40,31,45,42,57,52,60,61
- Postorder: 31,42,45,40,52,61,60,57,50
- Inorder: 31,40,42,45,50,52,57,60,61
- Breadth-First Traverse: 50,40,57,31,45,52,60,42,61
- Depth-First Traverse: 50,40,31,45,42,57,52,60,61

Note that this is a binary search tree (if you're not sure what this is, that's fine) and the inrder traversal is in increasing order.

6 Threaded Binary Trees

In a binary tree any missing child corresponds to a null pointer. We might wonder if there's a better use for this space. One way is to use the pointers in some other way.

Example 6.1. For example, suppose we're doing frequent inorder traversals of this tree. Each left-child null pointer can point to that node's inorder predecessor and each right-child null pointer can point to that node's inorder successor. Call these special pointers *threads*. We will need to assign a flag to all child pointers indicating whether they go to a real child or follow a thread.

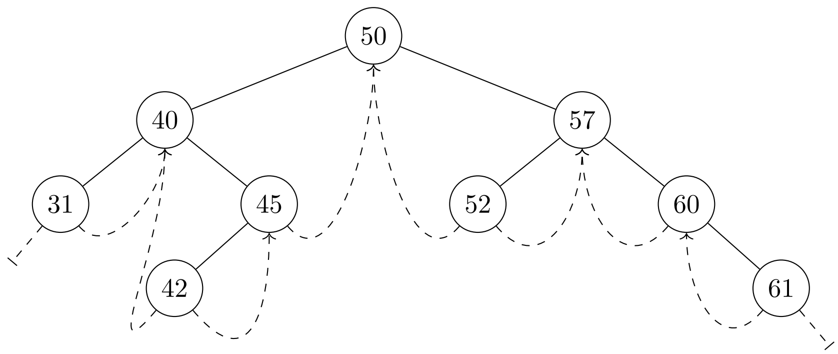
This allows for easy inorder traversal of the tree as follows. Suppose we are at some node x and want its inorder successor.

- If x's right-child pointer is a thread, follow it.
- If x's right-child pointer is not a thread then we have to systematically find the next largest key, so go to x's right child and follow left children as far as possible (possibly not at all).

Likewise suppose we are at some node x and want its inorder predecessor.

- If x's left-child pointer is a thread, follow it.
- If x's left-child pointer is not a thread then we have to systematically find the next smallest key, so go to x's left child and follow right children as far as possible (possibly not at all).

Here are the threads for the above tree:



Observe that the node with key 31 has no inorder predecessor because it is the first key in the inorder traversal. Similarly the node with key 61 has no inorder successor because it is the last key in the inorder traversal.

Note that for example 45's inorder successor is 50 via the thread but for 50's inorder successor we go right to 57 and then left as far as we can to 52. This is verified by the fact that 50 is 52's inorder predecessor.

We could do the same thing using preorder or postorder predecessors and successors.