CMSC 420: Trees

Justin Wyss-Gallifent

January 30, 2023

1 Overview ......................................................... 2
2 Types of Trees .................................................... 2
3 Tree Storage ....................................................... 2
4 Binary Tree Traversals ........................................... 3
5 Threaded Binary Trees ............................................ 3
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 level 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 every left child’s key is smaller than the parent’s key and every right child’s key is larger than the parent’s key.

3 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. Instead what we can do is this: Each node has two pointers, one to the first child, if any, and one to the first sibling, if any. In this way a parent can link to the first child when can then go on to link to its siblings. The benefit is that we can have as many children as we like, the downside is that we cannot go from a parent directly to every child, rather we have to go to the first child and follow the sibling links.
- 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.
4 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 need 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 \( T_R \).

**Example 4.1.** Consider the tree shown here:

![Binary Tree Diagram]

We have the following. We have 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

5 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.

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).

**Example 5.1.** Here are the threads for the above tree:

![Tree with threads](image)

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.