Trees

CS 225 Course Staff

2022-10-25

Contents

Chapter 1

Data Structures in C++

Chapter 2

The pool of tears

```
std::cout<< "hello harsh" <<std::endl;
```

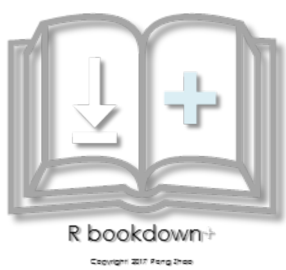

Figure 2.1: alt text or image title

Chapter 3

A caucus-race and a long tale

Chapter 4

Trees

You're either a botanist or computer scientist if you can talk about trees beyond the #savethetrees kind of narrative. Unlike trees in real life, Trees in computer science have a root node at the "top". So far we've talked about **linear** data structures. Linear data structures are ordered, meaning that there is a sequence with which our data is ordered and traversed. Trees are a **non-linear** data structure, meaning that the data isn't organized in a sequential manner. This means that you can visit all the elements on a tree in many different ways based on what type of problem you are trying to solve. For problems where we want to optimize for a certain outcome and don't care about order as much, trees are perfect.

If we take a very high level look at what trees and lists are composed of, it's just nodes and pointers, however, the difference with trees that they are **hierarchical** meaning that there is a top down organization. Trees are a hierarchical data structure with a certain set of properties that distinguish it from graphs. Trees are rooted, which means that there is a pointer to the root node and each child node can be reached via the root.

4.1 Basic tree terminology

(adapted from CS 173) * Vertex: "nodes"

- Path: sequence of edges
- Parents: Node **b, d, x** have Node **a** as their parent
- Children: **b, d, x,** are the children of **a**
- Siblings: **b, d, x,** are siblings of each other

- Ancestors: **u** has ancestors **l, d, a**
- Descendants: **x** has **s, m** as its descendants
- Leaves: Vertices with no children

4.2 Tree Property: Height

- **Computation of the tree height**
  - The length of the longest path from the root to the leaf (count edges).
  - If we want to compute recursively:

  height(T) = 1 + max(height(TL), height(TR)), where if height(null) = -1, which might be counter-intuitive but it follows the mathematical definition of tree height

4.3 Tree Property: Binary

- A binary tree is either
  - T = {TL, TR, r}, where TL, TR are binary trees
  - T = {} = $\emptyset$

4.4 Tree Property: Full

- A binary tree is full *if and only if*
  - Either: F = {}
  - Or: F = {TL, TR, r} where TL, TR both have either 0 or 2 children
- **Theorem**: A binary tree with n data items has n+1 null pointers.

4.5 Tree Property: Perfect

- A perfect tree Ph is defined by its height
  - Ph is a tree of height **h**, with
    - P-1 = {}
    - Ph = {r, Ph-1, Ph-1} when h>=0

4.6 Tree Property: Complete

(as defined in data structures) * A complete tree is * A perfect tree except for the last level

- All leaves must be pushed to the **left**
- Or, recursively, a complete tree **Ch** of height **h** is
  - C-1 = {}
  - Ch = {r, TL, TR} where
    - Either: TL = Ch-1 and TR = Ph-2 Or:TL = Ph-1 and TR = Ch-1

- Full does not imply perfect, so as complete does not imply perfect
- Not full implies not perfect, thus perfect implies full; perfect also implies complete too.

4.7 Tree Traversals

(practice them here: https://yongdanielliang.github.io/animation/web/BST.html) * Pre-Order: process the data first, then left child, then the right child * In-Order: left child, process the data, right child * Post-Order: left child, right child, process the data last

```
void BinaryTree<T>::preOrder(TreeNode * cur) {
    if (cur != NULL) {
        func(curr->data);
        preOrder(curr->left);
        preOrder(curr->right);
    }
}

void BinaryTree<T>::inOrder(TreeNode * cur) {
    if (cur != NULL) {
        preOrder(curr->left);
        func(curr->data);
        preOrder(curr->right);
    }
}

void BinaryTree<T>::inOrder(TreeNode * cur) {
    if (cur != NULL) {
```

```
        preOrder(curr->left);
        preOrder(curr->right);
        func(curr->data);
    }
}
```

4.8 Searching Trees

- BFS: breadth first search: visits nodes at each level (level-order traversal): use a queue
- DFS: depth first search: find the endpoint of the path quickly (in order, pre order or post order): use a stack
- Traversal vs Search: traverse visits every node vs search visits nodes until you find what you want

4.9 Delete and Insert