Graph Algorithms

CS 225 Course Staff

2022-10-26

Contents

Chapter 1

Data Structures in C++

hugs

Welcome to the Course Staff produced coursebook to pair with CS 225! We hope this really helps you learn more and improves your code. Report any issues at our Github repo and we'll be sure to credit you as well :)

Chapter 2

C++ Introduction

Chapter 3

Arrays and Lists

Chapter 4

Trees

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

Chapter 5

Binary Search Trees

Chapter 6

AVL Trees

Chapter 7

Heaps

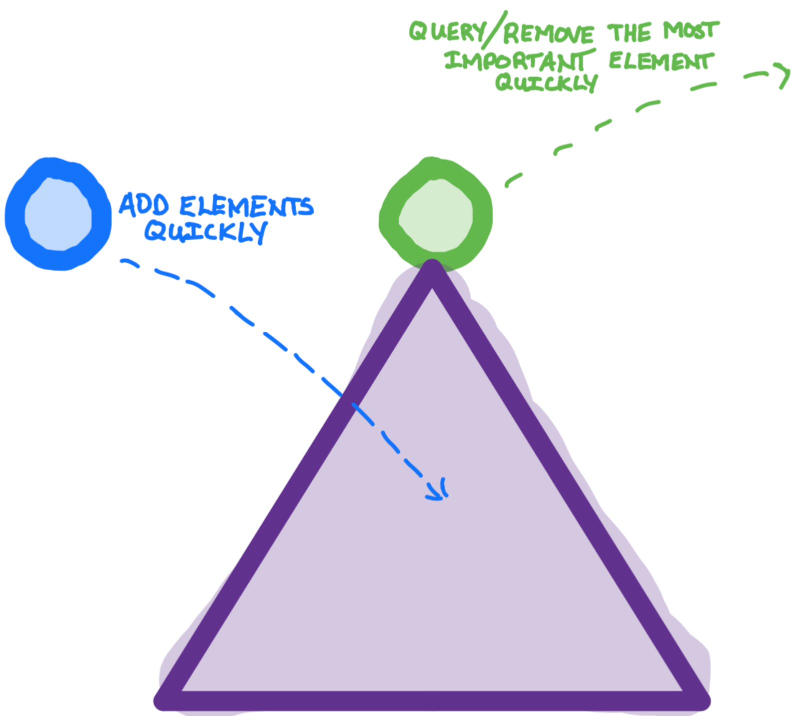

Figure 7.1: Heaps - Add elements quickly and query/remove the most important element quickly

7.1 Uses

- getting the smallest/largest item each time in succession
- maintaining top or bottom k elements, getting the median of large datasets
- sorting data via heap sort

7.1.1 Pre reqs of the data

- Has to be orderable
- Has to have > implemented

7.1.2 ADT implementation functions

- insert
- remove
- isEmpty

7.2 min heap vs max heap

- min heap is smallest at top and higher at the bottom
- max heap is the largest at top and goes smaller at the bottom
- the logic is basically the same in either case, just inverted - we'll do min heap here but the similar prinicples apply to max heap quite easily

7.3 Array based implementation

- the simplest way to do it is with arrays that has each level contiguous
- it makes swaps and indexing easy
- not having to deal with pointers as much - we're used to arrays

7.3.1 Compare to other implementations

- unsorted

7.4 insert() - Heapify up

- Add a bottom

7.5 Heapify down

7.6 Build heap

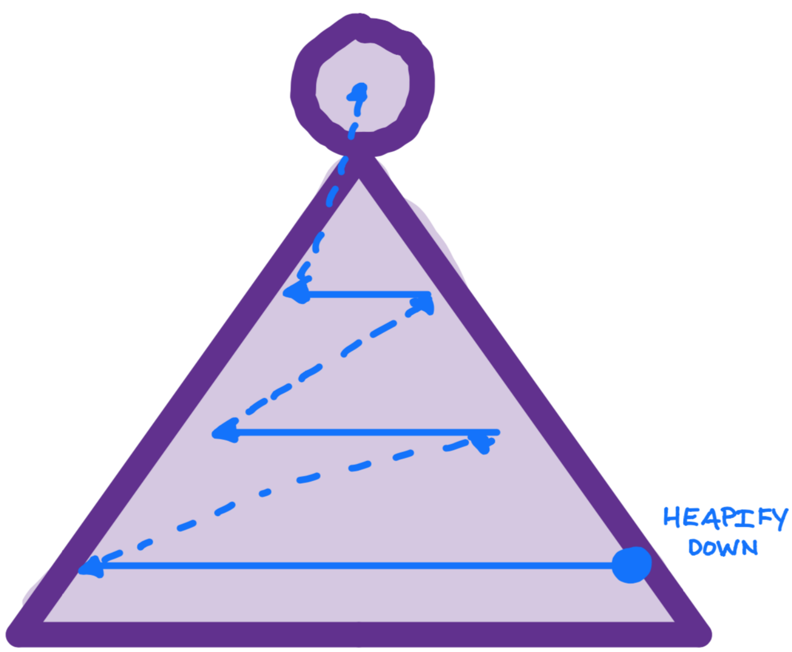

Figure 7.2: To build a Heap in linear time we heapify down from the bottom to the top

7.6.1 Recursive proof

7.7 Heap Sort

7.8 Priority Queue

7.9 See also:

- Learning to Love Heaps Long Medium Post by Vaidehi Joshi
- Introduction to a Heap Video Series by Paul Programming
- Old CS 225 resources page by Eddie Huang

Chapter 8

Disjoint Sets

Chapter 9

B Trees

Chapter 10

Hashing

Chapter 11

Graphs

Chapter 12

Graph Algorithms