Today
- Final Overview
- Heap Data Structure

Assignments
- HW 10 out (due by end of semester)
- Final Part 1 out (due by end of semester)
Final Exam Overview

2 parts (total of 20% of grade)

• Part 1 out – analysis (see website and piazza)
  – worth 40% of final exam

• Part 2 “in-class” exam
  – Wed, May 6th, 1:00–3:00
  – open book and notes
  – delivered via zoom and blackboard (must login to zoom by 1pm!)
  – worth 60% of final exam

Topics

• Cumulative

• Mix of questions from 2 exams (variations on questions)
  – Go back over last 2 exams
  – Go over quizzes
  – Go over lecture notes (exercises, examples, etc)

• Expect additional questions on:
  – Binary Search Trees
  – AVL Trees
  – Min/Max Heaps (from today’s lecture – as extra credit on exam)

Finally, be sure to complete all reading exercises in zybooks!!!
Heap Data Structure

Uses of Heaps ...

• Priority queues (ADT)
  – tasks (jobs) added to a list to be scheduled
  – each task assigned a priority
  – execute highest priority task from list first

• Sorting
  – Heapsort is another $O(n \log n)$ sorting algorithm
  – Taken an array, build a heap, get/remove items back in order
  – Like mergesort, trades space for efficiency

Two variants ...

• “Min” heaps have highest priority items as smallest
• “Max” heaps have highest priority items as largest

Primary functions for a Min Heap

• void insert(const K& key, const V& val) ... add key-value pair
• bool find_min(K& key, V& val) ... returns smallest key-value pair
• void remove_min() ... deletes smallest key-value pair
Implementing a Min Heap

- Array based, useful for heap sort ... we won’t talk about
- Pointer based, useful for priority queues ... what we’ll discuss

(Binary) Pointer-Based Min Heap

- A complete binary tree
- A different ordering constraint than a binary search tree ... 

Min Heap Ordering Constraint

- Every parent node has a smaller search key than children
- An empty tree is trivially a min heap

A valid heap

Not a valid heap

Not a valid heap
Finding the min key value

- Look at the root!
- This is worst-, best-, and average-case \( O(1) \)

Inserting key-value pairs

Basic Idea:

- Find next location in leaf-level of tree
- Insert a new node with key-value pair
- “Trickle up” the node’s values (recursively)

Exercise: Show the result of inserting the following (in order) into an empty heap:
\{80, 30, 70, 50, 40, 20, 60, 10\}

Q: What is the Big-O complexity of insert? ...

- \( O(\log n) \) in all cases (since have to traverse down to insert node)
- In an array, best is \( O(1) \) (no trickle up) and worst is \( O(\log n) \) (trickle up)
Removing the min key-value pair

Basic Idea:

- Find the leaf-level node to delete (like inorder successor)
- Copy the node’s values to the root and delete the node
- “Trickle down” the node’s values (recursively)
  - pick smallest child to trickle down with ... why?

Exercise: Show result of removing all of the (min) values
Q: What is the Big-O complexity of remove?

- $O(\log n)$ in all cases (traverse to delete, trickle down)
- In an array, best is $O(1)$ (no trickle down), worst is $O(\log n)$ (trickle down)
Navigating a path in a Heap

- Recall that a tree height of $h$, a full tree has $n = 2^h - 1$
  - in our examples, $h = 3$ so $2^3 - 1 = 7$
  - thus, our tree isn’t full since $n = 5$ and $5 < 7$

- Note that the leaf level has at most $2^{(h-1)}$ nodes
  - In our examples, since $h = 3$, $2^{(3-1)} = 4$ leaf nodes max
  - Since we are 2 nodes short of full, the second leaf node is the “last” leaf node

Basic sketch of finding the “last” node in a complete tree

```c
#include <math.h>

// pre: n >= 1
// where n is the number of nodes in the tree (size)
Node* last(int n, Node* subtree_root) {
    if (n == 1)
        return subtree_root;
    int h = ceil(log2(n+1));
    int unfilled = (pow(2, h) - 1) - n;
    if (unfilled == 0)
        return last(n - pow(2, h-1), subtree_root->left);
    int leaf_max = pow(2, h-1);
    int leaf_filled = leaf_max - unfilled;
    if ((leaf_max / 2) >= leaf_filled)
        return last(n - pow(2, h-2), subtree_root->left);
    else
        return last(n - pow(2, h-1), subtree_root->right);
}
```

This is the general idea, details will vary in your insert/remove implementation