Today

- Quiz 10
- Binary Heaps

Assignments

- HW 8 out
- HW 9 (optional) out
- Final Report due (see webpage for due date)
- Final Exam (see webpage for day/time)
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 and treesort, 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
- pair<K,V> find_min() ... 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 this
- 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 an $O(1)$ operation
- Q: What if we used a balanced tree instead? ... $O(\log n)$

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 $\Omega(1)$ (no trickle up) and worst is $O(\log n)$ (trickle up)
Removing the min key-value pair

Basic Idea: ... note we need to maintain a complete tree

- 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?

![Diagram of removing min key-value pair](image)

**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 $\Omega(1)$ (no trickle down), worst is $O(\log n)$ (trickle down)
Navigating to “last” node 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