CSCI 136
Data Structures & Advanced Programming

Lecture 24

Spring 2018

Profs Bill & Jon

Administrative Details

- Lab 8 Posted: Super Lexicon
  - Implement a Trie data structure
    - Trie: A tree of letters
  - Efficiently solve a problem using trees
  - lexicon.html
  - Partners (fill out the form)
  - Optional extensions are challenging!

- Pre-registration info session: Friday @2:30pm

Last Time

- Breadth-First and Depth-First Search
- Application: Huffman Encoding
- Priority Queues

Today

- Heaps
  - Implementation
  - Some analysis + proofs
- Heapsort

Priority Queues

- Always dequeue object with **highest priority** (smallest rank) regardless of when it was enqueued
- Data can be received/inserted in any order, but it is always returned/removed according to priority
- Like ordered structures (i.e., OrderedVectors and OrderedLists), PQs require comparisons of values

PQ Interface

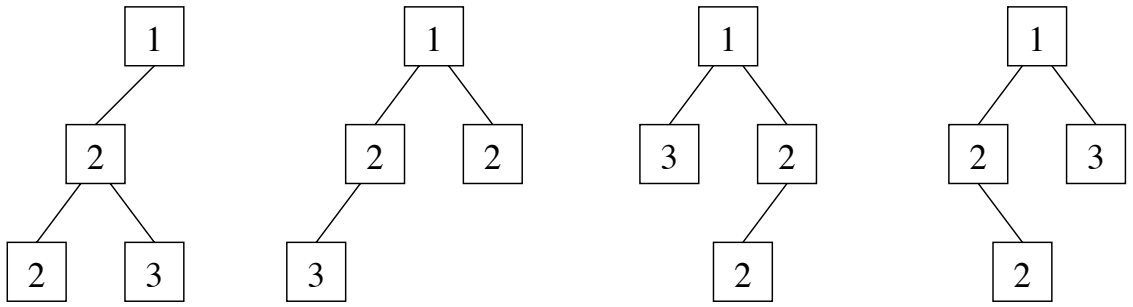
```
public interface PriorityQueue<E extends Comparable<E>> {
   public E getFirst(); // peeks at minimum element
   public E remove();   // removes + returns min element
   public void add(E value); // adds an element
   public boolean isEmpty();
   public int size();
   public void clear();
}
```

Implementing PQs

- An OrderedVector (`PriorityVector`)
  - Like a normal `Vector`, but no `add(int i)`
    - Instead, `add(Object o)` places `o` at proper location according to the ordering of all objects in the `Vector`
  - `O(n)` to add/remove from vector
  - Details in book…
  - Can we do better than `O(n)`?
- A Heap! (`VectorHeap`)
  - Partially ordered binary tree
  - $O(\log_2 n)$ to add/remove from heap

Heap

- A heap is a special type of tree
  - Root holds smallest (highest priority) value
  - Subtrees are also heaps (this is important!)
- Values increase in priority (decrease in rank) from leaves to root (from descendant to ancestor)
- *Heap Invariant for nodes:* For each child of each node
  - `node.value() <= child.value()` // if child exists
- Several valid heaps for same data set (no unique representation)



Implementing Heaps

- `VectorHeap`
  - Use conceptual array representation of BT (`ArrayTree`), but use extensible Vector instead of array (makes adding elements easier)
  - Note:
    - Root of tree is location 0 of Vector
    - Children of node in location i are in locations 2i+1 (left) and 2i+2 (right)
    - Parent of node i is in location (i-1)/2
      - Remember: dividing Integers truncates the result
  - *Heap Invariant* becomes
    - data[i] <= data[2i+1]; data[i]<=data[2i+2] (or kids might be null)

Implementing Heaps

- Strategy: tree modifications that always preserve tree *completeness*, but may violate heap property. Then fix.
  - Add/remove never add gaps to array
    - We always add in next available array slot (left-most available spot in binary tree)
    - We always remove using "final" leaf
  - When elements are added and removed, do small amount of work to "re-heapify"

Inserting into a PQ

- Add new value as a leaf
- "Percolate" it up the tree
  - while (value < parent's value) swap with parent
- This operation preserves the heap property since new value was the only one violating heap property
- Efficiency depends upon speed of
  - Finding a place to add new node
  - Finding parent
  - Tree height

Removing From a PQ

- Get value from root node (highest priority)
- Find a leaf, delete it, put its *data* in the root
- "Push" *data* down through the tree
  - while (*data.value* > value of (at least) one child)
    - Swap *data* with data of **smaller** child
- This operation preserves the heap property
- Efficiency depends upon speed of
  - Finding a leaf
  - Finding locations of children
  - Height of tree

VectorHeap Summary

- Let's look at VectorHeap code....

- Add/remove are both O(log n)
- Data is not completely sorted
  - "Partial" order is maintained: all root-to-leaf paths
- Note: VectorHeap(Vector<E> v)
  - Takes an unordered Vector and uses it to construct a heap
  - How?