CSCI 136
Data Structures & Advanced Programming

Lecture 25

Spring 2020

Profs Bill & Dan

Last Time

- Tries! (Such a cool data structure…)
  - Application: Lexicon (i.e., Lab 7)
- Priority Queues
  - Elements must be comparable (why?)
  - Ordered Vector implementation
    - Implements the interface, but performance is slow

Today

- Heaps
  - Implementation
    - Describe invariants
    - Describe general strategy
    - Walk through some examples (whiteboard)
    - Study the code

Priority Queues

- Recall from last recording: priority queues are ***not*** FIFO
  - We always dequeue the object with the **highest priority** regardless of when it was enqueued
  - We can define a min heap or a max heap
    - Min heap: smallest elements have highest priority
    - Max heap: largest elements have highest priority
- Where might we see prioritie queues in the "real world"?
  - ER triage, network routers, airplane boarding…

Priority Queues

- If data is returned/removed according to priority, then a heap can't store values that can't be sorted
  - Otherwise, how do we decide which elemen to prioritize?
- Like ordered structures (i.e., OrderedVectors and OrderedLists), PriorityQeues require values that are comparable

Reminder: PQ Interface

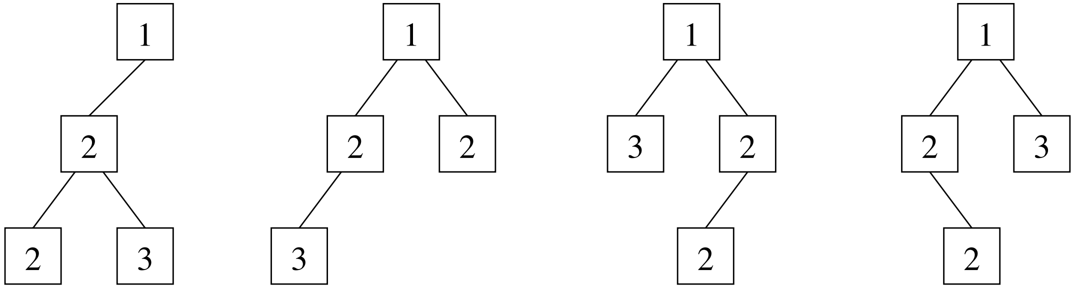
```
public interface PriorityQueue<E extends Comparable<E>> {
  public E getFirst(); // peeks at minimum element
  public E remove();   // removes + returns min element
  public void add(E value); // adds an element
  public boolean isEmpty();
  public int size();
  public void clear();
}
```

Implementing PQs

- An OrderedVector (`PriorityVector`)
- Details in book & discussed last lecture
  - 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
  - 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)
  - Recall from Binary Tree lecture 23:
    - 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: make tree modifications that preserve tree *completeness*, but may violate heap property. Then fix.
  - Question: what does a complete tree look like in an array representation?
  - 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”

Steps to insert into a PQ

1. Add new value as a leaf
2. “Percolate” the new value up the tree

```
while (value < parent’s value) {
        swap value with parent
}
```

- This operation preserves the heap property since new value was the only one violating heap property

Steps to remove from a PQ

1. Make a copy of the root node's value (highest priority). This will be our final result.
2. Replace the root node's value with the value in the rightmost leaf (removing the leaf). This violates the heap property.
3. "Push" *value* down through the tree to restore the heap property:

```
while (value > at least one child ) {
        Swap value with the smaller child
}
```

- This operation preserves the heap property since the "new root" was the only one violating heap property

Analyzing PQ Performance

- Insertion efficiency depends upon speed of:
  - Finding a place to add new node O(1)
  - Finding parent (to "percolate up" new node) O(1)
  - Tree height $O(\log_2 n)$

- Removal efficiency depends upon speed of
  - Finding a leaf O(1)
  - Finding locations of children (to "push down" new root) O(1)
  - Tree height $O(\log_2 n)$

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?

Next Class

- How to “heapify” a vector?
  - With some algorithmic analysis, we can decide between “Top-Down” and “Bottom-up”
- HeapSort
  - Idea
  - Performance
  - Advantages
- “Skew Heaps”
  - Brief overview