Intro to Data Structures

Lecture #22 – Priority Queues & Heaps
November 16, 2014

Mark Stehlik

Outline for Today

- HW5 returned with grade sheets
- HW6 due Tuesday midnight (Pgh)
- Priority Queues
- Heaps
- What's left?
  - Maps & Sets
  - Hashing
  - stuff

Priority Queue

- A new Abstract Data Type (what ADT's have we seen so far in the course?)
- What's new about this one? Operations:
  - boolean isEmpty
  - add(AnyType) // AnyType must implement Comparable
  - AnyType peekMin() // could also be peekMax()
  - AnyType removeMin() // or removeMax()
- What does this remind you of?
- Applications – what is it used for?

Priority Queue implementations

- What data structure should we use to implement a Priority Queue?
- What data structures have we seen so far and how efficient will they be?
  - unordered array (or ArrayList)
  - sorted array (or ArrayList)
    - increasing order
    - decreasing order
  - LinkedList (increasing order)
  - BST

Efficiency of various implementations

	unordered ArrayList	increasing ArrayList	decreasing ArrayList	increasing LinkedList	BST
add	**O(1)**	**O(n)**	**O(n)**	**O(n)**	**O(log n) [w/c O(n)]**
peekMin	**O(n)**	**O(1)**	**O(1)**	**O(1)**	where would min be? **O(log n) [w/c O(n)]**
removeMin	**O(n)**	**O(n)**	**O(1)**	**O(1)**	**O(log n) [w/c O(n)]**

Can we do better?

- Yes, but we'll need a new data structure - a *binary heap*. A *binary heap* has two properties:
  - Shape (structure) property - it must be a complete binary tree (what was that?)
  - Order (heap) property - the parent of a node is <= its children (minHeap) or >= its children (maxHeap)
- Examples…

Can we do better?

- How does this work w/respect to implementing the operations of a priority queue?
  - add - add at end (maintain *shape* property) & heapify up (swap w/parent if necessary to maintain *order* property)
  - peekMin - always at the root
  - removeMin - remove root; move last leaf into root's position (maintain *shape*) & heapify down (swap w/smaller child if exists to maintain *order*)
  - some examples…
- what data structure should we use to implement the heap in order to minimize the cost of operations?

Efficiency of PQ ops

- Use the array implementation of a binary tree!
  - access to the place to insert the next leaf value - O(1)
  - heap with n nodes will have height log n
  - add - at end (maintain *shape* property) & heapify up (if less, swap w/parent to maintain *order* property): O(1) + O(log n) [most nodes added low; why?]
  - peekMin - return the value at the root: O(1)
  - removeMin - remove root; move last leaf into root's position (maintain *shape*) & heapify down (swap w/smaller child if exists to maintain *order*): O(1) + O(log n)

Is it better than the others?

- Yes…
  - O(1) for peekMin(), guaranteed O(log n) for add() and removeMin()
- So?
  - Fast implementation of Priority Queue
  - And?
    - If I add n integers to an initially empty minHeap and then remove items one at a time, what is true about the sequence of values removed?
    - They are in ascending order! This is heap sort!

Details of Heapsort…

- Time to add n integers into an empty minHeap?
  - n * O(log n) --> O(n log n)
- Time to remove n mins?
  - n * O(log n) --> O(n log n)
- Time to do both?
  - 2 O(n log n) --> O(n log n)
- And, unlike merge sort, it can be done in place (no auxiliary storage). How? Put the elements into a maxHeap, then removeMax, store in last open index, removeMax, store in next open index…