Heaps and Priority Queues

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

February 17, 2019

Heaps and Priority Queues

The Priority Queue ADT

- A collection of items, each with a priority
- Exists in two variants:
  1. Min-priority queue:
     - insert(item): Add `item` to queue
     - minItem(): Return the item with the smallest priority
     - deleteMin(): Remove and return (pop) the item with the smallest priority from the queue
  2. Max-priority queue:
     - insert(item): Add `item` to queue
     - maxItem(): Return the item with the largest priority
     - deleteMax(): Remove and return (pop) the item with the largest priority from the queue
- No support for efficient random access, ordered iteration, etc.
- Optional advanced (but useful) operation:
  - changePriority(item, priority): Update priority of item already in the priority queue

Priority Queues: Examples

- For max-PQ: job pool:
  - Insert jobs to be executed with some "importance" (priority) parameter
  - Pull out most important job to perform next
- For min-PQ: scheduling:
  - Insert events with triggering times (e.g., at 12:05:00, launch task T)
  - Look at most imminent event to execute
- Common tool in algorithms (A* search, Dijkstra, Prim):
- Common pattern:
  1. Start with initial state S
  2. Discover some new states from S, add to PQ
  3. Select the best (min-cost) discovered state S' as our next state
  4. Discover new states from S', add to PQ
  5. Repeat until done

Questions

1. Assume we are using a max-PQ with objects with an item.priority field storing the priority value.
   How does the PQ operate in the following cases?
   1.1 If every new item has priority larger than all previous ones? (E.g., priority is a ticking "clock".)
      - pq.deleteMax() acts like stack.pop() – newest first
   1.2 If every new item has priority smaller than all previous ones?
      - pq.deleteMax() acts like queue.dequeue() – oldest first
2. Which data structures that we already know could provide good/bad implementations of a Priority Queue?
   Some options:
   - Same as a Queue (linked list/array): always possible, but bad worst-case performance
   - Self-balancing BST, e.g., AVL tree or Red/black tree: good worst-case bounds, very complex code.
   - Will see binary heaps data structure

Implementation 0: As Queue

- It's possible to implement PQ via a linked list:
  - Linked list contains items sorted by priority
  - minItem() and deleteMin() work in $\Theta(1)$ time
    (just look at / delete the head of the list)
  - insert(item): insert in correct position
    (up to $\Theta(n)$ work to insert into middle position)
- Array-based variant: circular buffer (very similar profile)
- These may be efficient if items arrive mostly in sequence
- ...but we want worst-case guarantees, which this doesn't give

Implementation 1: Trees

- We can easily support a Priority Queue with a balanced binary search tree:
  - `insert`: Exists, time $O(\log n)$
  - `minValue`: Exists, time $O(\log n)$
  - `deleteMin`: Exists via `min`+`delete`, time $O(\log n)$
- Even the operation changePriority(item, priority):
  - Delete old item, insert with new priority
  - $O(\log n)$ total time

Implementation 2: Heaps

- Canonical implementation of priority queues
- Operations: `insert`, `deleteMin` in $O(\log n)$ time, `minValue` in $O(1)$ time
- Can bootstrap: $O(n)$-time initialisation with n items

Why heaps?

Running times of tree and heap implementations nearly identical – why heaps?

- Heaps live in arrays – trees are linked structures (use nodes)
  - Heaps have better memory usage
  - Heaps are slightly faster in practice (tighter code)
- Simpler code
- Offer interesting extensions (beyond scope of this course)

We will see...

- [Now] Heaps: What is a heap? How is it represented in memory?
- [Now] Heaps: Implementing `insert`, `deleteMin`
- [Later] $O(n)$-time `makeHeap`
- [Later] The `heapsort` algorithm

Binary Heaps

Heaps

1. A heap is a tree with a special property
2. A binary heap is a heap represented as an array
   - Logically: Works like a tree
   - Physically: Lives in memory like an array (no pointers)
3. "Heap tree" property:
   - Every node in the tree is smaller than all its children
   - By induction: Smallest item in root
   - Not a binary search tree!

Binary heaps: Tree/array representation

- To map tree into array, each node gets a number (levelorder)
  - Root is number 0
  - The children of node i are number $2i + 1$ and $2i + 2$
  - Can access data for node i as `array[i]` (no links needed)
- A binary heap is always nearly complete
  - All nodes $0, 1, \dots n - 1$ exist (no "holes" in array)
  - Implies: All nodes have exactly two children, except possibly at last and second-last level

Main heap property

Heap property

Every node in the tree contains a smaller item than both its children

- By induction:
  - Smallest item always in the root
  - Largest item can be anywhere on leaf level (among $n/2$ options)
- More relaxed than a search tree – easier to maintain

Heap operations

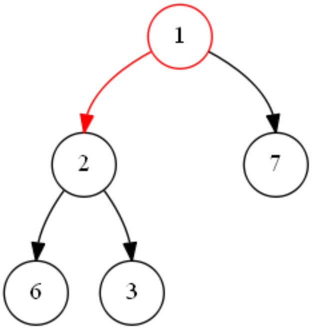
Presented algorithms

1. Primitive (underlying) heap operations:
   - `siftUp(index)` ("swim"), `siftDown(index)` ("sink")
2. Fundamental Priority Queue operations:
   - `min`, `insert`, `deleteMin`
3. `makeHeap` operation: Fast initialisation
4. `heapsort` sorting algorithm

Primitive 1: siftUp

- Case: Tree is almost a heap, except one item (a leaf) that is in the wrong place
- Need to "sift up": bubble item upwards until it finds its place
- Code:
  1. While `array[i] < array[parent(i)]`:
     1.1 Swap contents `array[i]` and `array[parent(i)]`
     1.2 Let `i=parent(i)` (break if trying to find parent of node 1)

Sift up: Visualisation



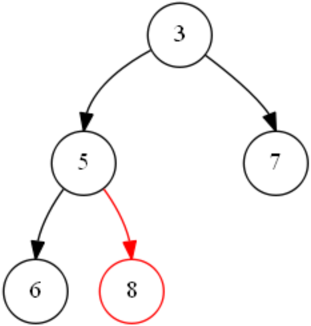
Sift up: Analysis

- Question 1: Why does it work? (Why is the result a heap?)
  - Assumed: Only problem is that node `i` is out of place
  - In swap, only shifted `i` and `parent(i)` around
  - Easy to check that `i` and `parent(i)` are both smaller than both their children after the swap
- Question 2: What is its running time?
  - Number of steps $\leq$ height of tree
  - Tree balanced $\Rightarrow$ $O(\log n)$ time

Primitive 2: Sift down

- Case: Tree is (again) almost a heap, except one item (maybe the root) is too large
- Need to "sift" or "bubble" item downwards until it finds its place
- Code `siftDown(i)`:
  1. Let `smallest` be smallest node of `i`, `leftChild(i)`, `rightChild(i)`
  2. If `smallest` $\neq$ `i`:
     2.1 Swap contents `array[i]` and `array[smallest]`
     2.2 Call `siftDown(smallest)` recursively

Sift down: Visualisation



Sift down: Analysis (brief)

1. Correctness: Similar to `siftUp` (somewhat longer arguments)
2. Running time: $O(\text{tree height}) = O(\log n)$.

Priority Queue operations

- Operation `min`:
  1. Return `array[0]`
- Operation `insert(item)`:
  1. Add `item` to the end of array
  2. Call `siftUp(n)` to put it in its place
- Operation `deleteMin`:
  1. Swap root with last item of tree
  2. "Forget" last item (let `n=n-1`)
  3. Call `siftDown(0)` to put other item in place

Priority Queue: Summary

- Priority Queue: Useful ADT for situations where we want to frequently access the "most urgent" / "best-looking" item
- Well-implemented (canonically) by the heap structure
- Advantages:
  - Small memory usage (no extra space for "links"/pointers or node objects)
  - Simple code, fast operations
- However: Need to understand heap structure ("tree as array")
- Extensions exist (binomial heap, Fibonacci heap) – technically, we only saw the binary heap