Heaps: Makeheap and Heapsort

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

February 18, 2019

Heaps: Heapsort

Heaps (reminder)

- A heap is a binary tree represented (stored) in a normal array
- The heap property: Every node in the tree contains a smaller value than both its children
- Example: Which of the following two arrays is a heap?
  1. $[1, 7, 3, 9, 8, 5]$
  2. $[1, 4, 9, 5, 6, 7]$
- 1. is a heap and 2. is not
- To remember:
  1. Array representation, tree drawing
  2. Fundamental repair operations: siftUp, siftDown (sink/swim)
  3. Implement min, insert, deleteMin using these

Plan

1. Makeheap: bootstrap heap from n items in $\mathcal{O}(n)$ time
2. Heapsort sorting algorithm
3. Discussion: decreaseKey, increaseKey?

Heaps: Fast initialisation

- Assume we have an array (e.g., `int[] array`) of n items we want to put in a heap
- Could make n calls to `insert` (takes $O(n \log n)$ time)
- Faster version: `makeHeap` algorithm:
    1. Start with `n` numbers in an array in arbitrary order
    2. For `i=n-1` down to `0`:
        2.1 Call `siftDown(i)`

makeHeap

- Question 1: Correctness?
  1. Code ends up doing "passes" level-by-level (from the bottom)
  2. After level ℓ is done, if `i` is in level $\ell - 1$, then inside the subtree rooted in `i`, only `i` is out of place
  3. So the `siftDown` call is valid: It "fixes" a tree with only one node out of place
- Question 2: Running time?
  - Intuition: Time for `siftDown(i)` is bounded by height of tree under `i`
  - "Almost all" nodes have very small trees under them ($n/2$ leaves, $n/4$ height 1, ...)

★Running time of makeHeap

- Assume tree has 31 nodes (1+2+4+8+16). Then:
  - 16 leaves: May skip these.
  - Number of iterations bounded by

    $$1+1+1+1+1+1+1+1+2+2+2+2+3+3+4=26$$

- Reordered:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+$$
$$0+0+0+0+0+0+0+0+1+1+1+1+1+1+1+$$
$$0+0+0+0+0+0+0+0+0+0+0+0+1+1+1+$$
$$0+0+0+0+0+0+0+0+0+0+0+0+0+0+1$$

- Technically, the running time is (see course book):

$$\sum_{i=1}^{\log n} O(i) \cdot \frac{n}{2^i} = O(n).$$

Heapsort

- Will show popular application of heaps: `heapsort`.
- In-place worst-case $O(n \log n)$-time sorting algorithm (but usually not the fastest algorithm in practice)
- Uses max-heaps (as what we saw, except inverted: largest item in root)

Heapsort code

- Code for `heapsort(int[] array)`, array length n:
  1. Call `makeHeap` to turn `array` into heap (in-place).
  2. For `i = n-1` down to 0:
     - 2.1 Swap `array[0]` with `array[i]`
     - 2.2 "Forget" item `array[i]` in heap (decrease "size" parameter)
     - 2.3 Call `siftDown(0)` to put "new root" in place
- We use a max-heap: The first thing that happens, max. element placed in `array[n]`
- Next step, second largest element in `array[n-1]`, etc.
- Time $O(n) + n \cdot O(\log n) = O(n \log n)$

Features of heapsort

- Theoretically efficient: $\mathcal{O}(n \log n)$ worst-case time
- Cheap memory use: No auxiliary data, need only constant number of "index" variables (technically $\mathcal{O}(\log n)$ space)
- Not very good in practice
  - siftUp, siftDown "jumps around" in memory too much if the array is large
  - Both quicksort and mergesort have "tighter" loops

Discussion: Modify key value

- A common operation in algorithms is to change (increase/decrease) the priority of an item in the priority queue (assume max-heap)
- Claim: To change the priority of the item in tree/array position i is easy:
  1. increaseKey at position i: Change priority, call siftUp
  2. decreaseKey at position i: Change priority, call siftDown
- Issue: We want to provide increaseKey(item, value), not redincreaseKey(position, value). Fixes?
  - Suggestion: Augment with map: $\text{Map}\langle \text{Items}, \text{Positions} \rangle$
  - Update your map for every modification to the array (e.g., every swap)

Data Structures – Wrap-up

Data structures wrap-up

- We saw three “advanced” data structures:
  1. Binary search trees (unbalanced, self-balancing)
  2. Hash tables
  3. Heaps
  4. Also “basics” like sorted/unsorted array, linked lists
- Each with their uses...
  - Self-balancing binary search trees: Good all-rounder structure
  - Hash tables: Very useful implementation of Set and Map functionality
  - Heaps: For Priority Queues
  - Arrays: Honestly, probably the most common structure you will actually use!

Data structures wrap-up, pt. 2

- Binary Search Trees:
  - In "plain" version, not very good performance...
  - With self-balancing (rotations: AVL-trees or red-black trees), perform almost any reasonable operation in time $O(\log n)$
- Hash tables:
  - Set-operations $O(1)$ time, no support for min/max/successor
- Heaps: Basically, for implementing the Priority Queue ADT
  - Useful concept on its own (job scheduler, message queue, etc.)
  - Also useful inside other algorithms (e.g. A*-search: Keep P.Queue of most promising path candidates)
- Arrays: Good if data small or well-behaved, e.g., data arrives almost in order