CSCI 136
Data Structures &
Advanced Programming

Lecture 23

Fall 2018

Instructor: Bills

Administrative Details

- Lab 8: Simulations
  - You will simulate two queuing strategies
  - You can work with a partner
  - Time spent on lab before Wed. is time well-spent!
- Problem Set 3 is online
  - Due this Friday at beginning of class

Last Time

Improving Huffman's Algorithm

- Priority Queues & Heaps
  - A "somewhat-ordered" data structure
    - Conceptual structure
    - Efficient implementations

Today

- Finishing up with heaps
  - HeapSort
  - Alternative Heap Structures
- Binary Search Tree: A New Ordered Structure
  - Definitions
  - Implementation

Recap: Implementing Heaps

- Features
  - Represent as a full binary tree stored in an array
    - We always add in next available array slot (left-most available spot in binary tree (see percolate method)
    - We always remove using "final" leaf (see pushDown method)
  - *Heap Invariant becomes*
    - data[i] <= data[2i+1]; data[i]<=data[2i+2] (or kids might be null)
  - When elements are added and removed, do small amount of work to "re-heapify"
    - Finding a node's child or parent takes constant time, as does finding "final" leaf or next slot for adding
    - Since this heap corresponds to a full binary tree, the depth of the tree is O(log n), so percolate/pushDown takes O(log n) time!

Heapifying A Vector (or array)

- Method I: Top-Down
  - Assume V[0...k] satisfies the heap property
  - Now call percolate on item in location k+1
  - Then V[0..k+1] satisfies the heap property
- Method II: Bottom-up
  - Assume V[k..n] satisfies the heap property
  - Now call pushDown on item in location k-1
  - Then V[k-1..n] satisfies heap property
- Check out the demos at visualgo.net

Top-Down vs Bottom-Up

- Top-down heapify: elements at depth d may be swapped d times: Total # of swaps is at most

$$\sum_{d=0}^{h} d2^d = (h-1)2^{h+1} + 2 = (\log n - 1)2n + 2$$

- This is O(n log n)
- Some intuition: most of the elements are in the lowest levels of the tree, so each of them might have to move to root: O(log n) swaps per element

Top-Down vs Bottom-Up

- Bottom-up heapify: elements at depth d may be swapped h-d times: Total # of swaps is at most

$$\sum_{d=0}^{h} (h-d)2^d = 2^{h+1} - h - 2 = 2n - \log n + 2$$

  - This is O(n) --- beats top-down!
  - Some intuition: most of the elements are in the lowest levels of the tree, so each of them will only be pushed down (swapped) a small number of times   SO COOL!!!

Some Sums

$$\sum_{d=0}^{d=k} 2^d = 2^{k+1} - 1$$

$$\sum_{d=0}^{d=k} r^d = (r^{k+1} - 1)/(r - 1)$$

$$\sum_{d=0}^{d=k} d * 2^d = (k - 1) * 2^{k+1} + 2$$

$$\sum_{d=0}^{d=k} (k - d) * 2^d = 2^{k+1} - k - 2$$

All of these can be proven by (weak) induction.

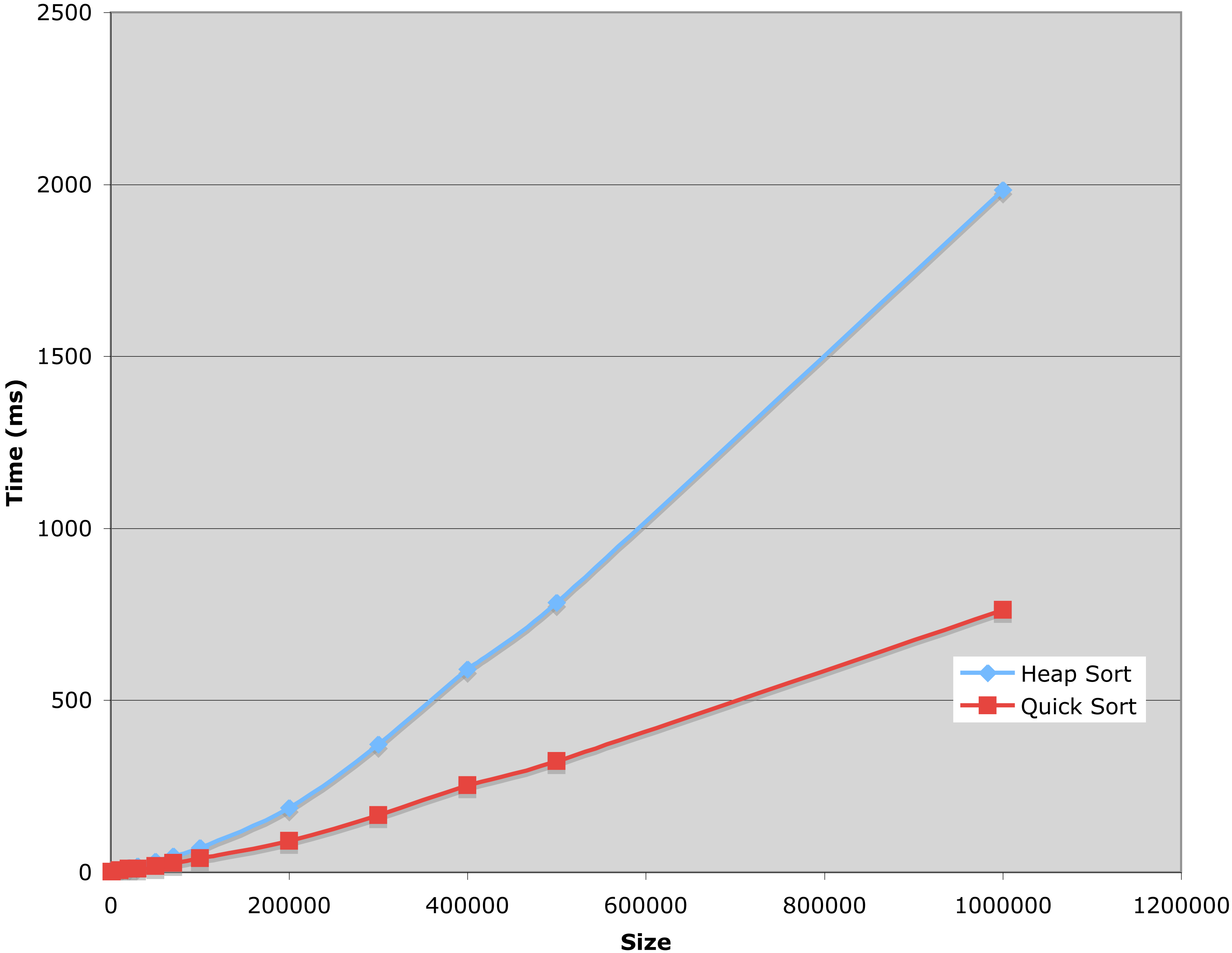
Try these to hone your skills

The second sum is called a geometric series. It works for any $r \neq 1$

HeapSort

- Heaps yield another O(n log n) sort method
- To HeapSort a Vector "in place"
  - Perform bottom-up heapify on the reverse ordering: that is: highest rank/lowest priority elements are near the root (low end of Vector)
  - Now repeatedly remove elements to fill in Vector from tail to head
    - For(int i = v.size() – 1; i > 0; i--)
      - RemoveMin from v[0..i] // v[i] is now not in heap
      - Put removed value in location v[i]

Heap Sort vs QuickSort



Why Heapsort?

- Heapsort is slower than Quicksort in general
- Any benefits to heapsort?
  - *Guaranteed* O(n log n) runtime
- Works well on mostly sorted data, unlike quicksort
- Good for incremental sorting

More on Heaps

- Set-up: We want to build a *large* heap. We have several processors available.
- We'd like to use them to build smaller heaps and then merge them together
- Suppose we can share the array holding the elements among the processors.
  - How long to merge two heaps?
  - How complicated is it?
- What if we use BinaryTrees for our heaps?

Mergeable Heaps

- We now want to support the additional *destructive* operation merge(heap1, heap2)
- Basic idea: heap with larger root somehow points into heap with smaller root
- Challenges
  - Points how? Where?
  - How much reheapifying is needed
  - How deep do trees get after many merges?

Skew Heap

- Don't force heaps to be complete BTs?
- Develop recursive merge algorithm that keeps tree shallow over time
- Theorem: Any set of m SkewHeap operations can be performed in O(m log n) time, where n is the total number of items in the SkewHeaps
- Let's sketch out merge operation....

Skew Heap: Merge Pseudocode

```
SkewHeap merge(SkewHeap S, SkewHeap T)
    if either S or T is empty, return the other
    if T.minValue < S.minValue
        swap S and T        (S now has minValue)
    if S has no left subtree, T becomes its left subtree
    else
        let temp point to right subtree of S
        left subtree of S becomes right subtree of S
        merge(temp, T) becomes left subtree of S
    return S
```

Tree Summary

- Trees
  - Express hierarchical relationships
  - Tree structure captures relationship
    - i.e., ancestry, game boards, decisions, etc.
- Heap
  - Partially ordered tree based on item priority
  - Node invariants: parent has higher priority than each child
  - Provides efficient PriorityQueue implementation