

CSCI 136

Data Structures & Advanced Programming

Lecture 25
Spring 2018
Profs Bill & Jon

Administrative Details

- Lab 8 Today
 - Partners (fill out the form)
 - Optional extensions are challenging!
- Pre-registration info session:
 - Friday @2:30pm
- Sigma Xi talks this week:
 - Thursday + Friday at 4:15pm

Last Time

- Heaps
 - Some Implementation details

Today

- Heaps
 - Finish Implementation details
 - Some analysis + proofs
- Heapsort
- Binary Search Trees

Implementing Heaps

- Strategy: perform tree modifications that always preserve tree *completeness*, but may violate heap property. Then fix.
 - 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”
 - pushDownRoot(): recursively swaps large element down the tree
 - percolateUp(): recursively swaps small element up the tree

VectorHeap Summary

- Let's look at VectorHeap code....
- Add/remove are both $O(\log n)$
- Data is not *completely* sorted
 - “**Partial**” order is maintained on all root-to-leaf paths
- Note: `VectorHeap(Vector<E> v)`
 - Takes an unordered Vector and uses it to construct a heap
 - How?

Heapifying A Vector (or array)

Problem: You are given a Vector V that is not a valid heap, and you want to “heapify” V

- Method I: Top-Down
 - Assume $V[0..k]$ satisfies the heap property
 - Now call `percolateUp` on item in location $k+1$
 - Then $V[0..k+1]$ satisfies the heap property



----->
Grow heap one element at a time

Practice Top-Down

Input:

- `int a[6] = {7, 5, 9, 1, 2, 5, 4}`
 0 1 2 3 4 5 6

```
for (int i = 0; i < a.length; i++)  
    percolateUp(a, i);
```

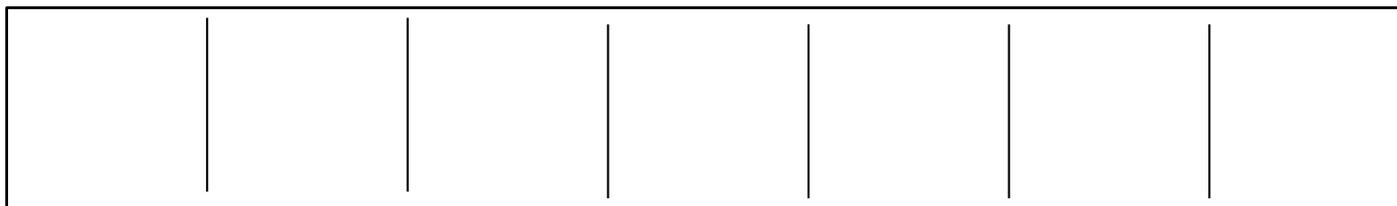
Result: a is a valid heap!

- `a = [1 | 2 | 4 | 7 | 5 | 9 | 5]`
 0 1 2 3 4 5 6

Heapifying A Vector (or array)

Problem: You are given a Vector V that is not a valid heap, and you want to “heapify” V

- 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



← - - - - -
Grow heap one element at a time

Practice Bottom-Up

Input:

- `int a[6] = {7, 5, 9, 1, 2, 5, 4}`
 0 1 2 3 4 5 6

```
for (int i = a.length-1; i > 0; i--)  
    pushDownRoot(a, i);
```

Result: a is a valid heap!

- `a = [1 | 2 | 4 | 5 | 7 | 5 | 9]`
 0 1 2 3 4 5 6

Let's Compare

- Which is faster: Top down or Bottom Up?
 - Think about a binary tree: Where do most of the nodes live?
 - Given that most of the nodes are leaves, should we percolateUp or pushDown?

Some Sums (for your toolbox)

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

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

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

Try these to hone your skills

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

The second sum is called a geometric series. It works for any $r \neq 0$

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

Top-Down vs Bottom-Up

- Top-down heapify: (percolate up) elements at depth d may be swapped d times: Total # of swaps is

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

(recall: $h = \log n$)

- 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: (push down) elements at depth d may be swapped $h-d$ times: Total # of swaps is

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

- 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!!!**

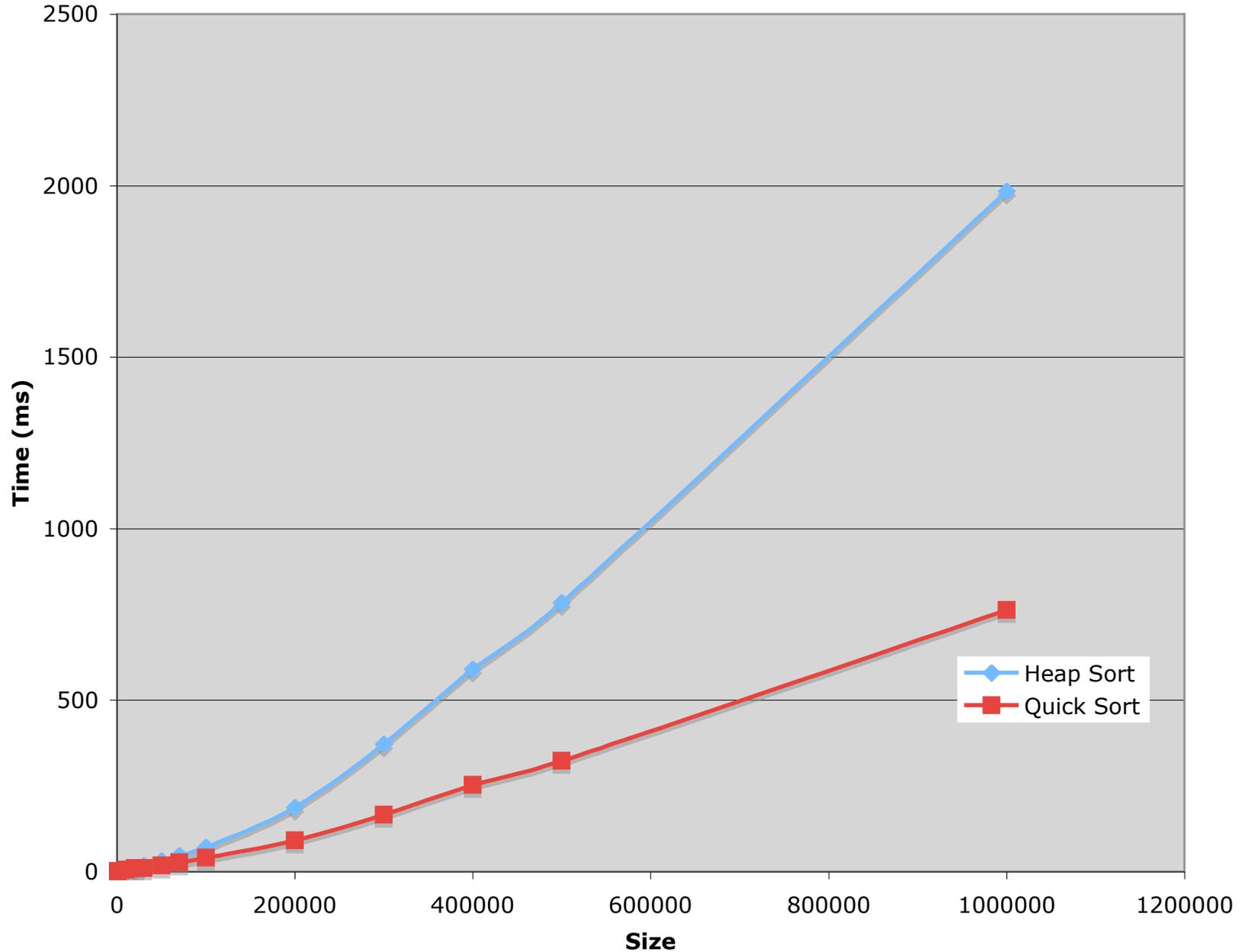
HeapSort

- “Advanced” Selection Sort (review Sel. Sort?)
- Strategy:
 - Make a *max-heap*: array[0...n]
 - array[0] is largest value
 - array[n] is rightmost leaf
 - Take the largest value (array[0]) and swap it with the rightmost leaf (array[n])
 - Call pushDownRoot on array[0] array[0...n-1]
 - Now our heap is one element smaller, but largest element is at end of array.
 - Repeat until array is sorted

HeapSort

- Another $O(n \log n)$ sort method
- Heapsort is not *stable*
 - The relative ordering of elements is not preserved in the final sort
 - Why?
 - There are multiple valid heaps given the same data
- Heapsort can be done *in-place*
 - No extra memory required!!!
 - Great for resource-constrained environments

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

- Heaps are not necessarily complete BTs
 - Rather than use Vector as underlying data structure, use BT
- Details in book, but...
 - The merge algorithm 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

Skew Heap: Merge Pseudocode

SkewHeap merge(SkewHeap S, SkewHeap T)

if either S or T is empty, return the other **Case 1**

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

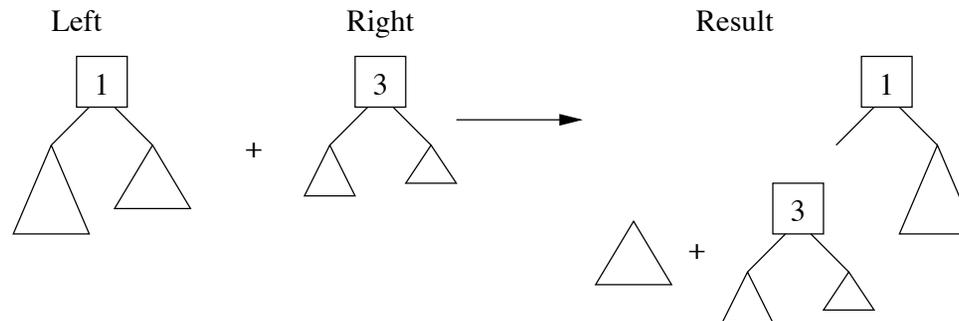
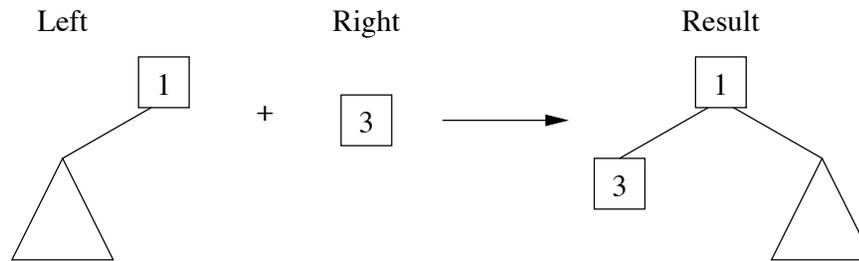
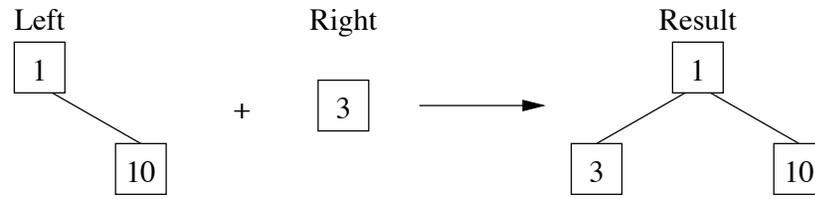
return S

Case 2

Case 3

(recurse)

Skew Heap: Merge Examples



Tree Summary

- Trees
 - Express hierarchical relationships
 - Level ordering captures the 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