Heapsort
Homework 1 posted, due Sunday Oct. 1
3

Radix sort

Use a stable sort to sort from the least significant digit to most

Psuedo code: (A=input)
for i = 1 to d
    stable sort of A on digit i
// i.e. use counting sort
Stable means you can draw lines without crossing for a single digit.
Radix sort

Run time?
Radix sort

Run time?

\[ O\left( \frac{b}{r} (n+2^r) \right) \]

b-bits total, r bits per 'digit'

d = \frac{b}{r} digits

Each count sort takes \( O(n + 2^r) \) runs

count sort d times...

\[ O( d(n+2^r)) = O( \frac{b}{r} (n + 2^r) ) \]
Radix sort

Run time?

if $b < \log(n)$, $\Theta(n)$
if $b \geq \log(n)$, $\Theta(n \log n)$
Heapsort
It is possible to represent binary trees as an array.
Binary tree as array

index 'i' is the parent of '2i' and '2i+1'
Is it possible to represent any tree with a constant branching factor as an array?
Binary tree as array

Is it possible to represent any tree with a constant branching factor as an array?

Yes, but the notation is awkward
Heaps

A max heap is a tree where the parent is larger than its children (A min heap is the opposite)
The idea behind heapsort is to:

1. Build a heap
2. Pull out the largest (root) and re-compile the heap
3. (repeat)
Heapsort

To do this, we will define subroutines:

1. Max-Heapify = maintains heap property

2. Build-Max-Heap = make sequence into a max-heap
Max-Heapify

Input: a root of two max-heaps
Output: a max-heap
Max-Heapify

Pseudocode Max-Heapify(A,i):
left = left(i)  // 2*i
right = right(i)  // 2*i+1
L = arg_max( A[left], A[right], A[i] )
if (L not i)
    exchange A[i] with A[L]
Max-Heapify(A, L)
// now make me do it!
Max-Heapify

Runtime?
Max-Heapify

Runtime?

Obviously (is it?): \( \lg n \)

\[ T(n) = T\left(\frac{2}{3} n\right) + O(1) \] // why?

Or...

\[ T(n) = T\left(\frac{1}{2} n\right) + O(1) \]
Master's theorem

Master's theorem: (proof 4.6)
For \( a \geq 1, b \geq 1, T(n) = a \cdot T(n/b) + f(n) \)

If \( f(n) \) is... (3 cases)
\( O(n^c) \) for \( c < \log_b a \), \( T(n) \) is \( \Theta(n^{\log_b a}) \)
\( \Theta(n^{\log_b a}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \cdot \log n}) \)
\( \Omega(n^c) \) for \( c > \log_b a \), \( T(n) \) is \( \Theta(f(n)) \)
Max-Heapify

Runtime?

Obviously (is it?): \(\lg n\)

\[ T(n) = T(2/3 \ n) + O(1) \quad // \quad \text{why?} \]
Or...

\[ T(n) = T(1/2 \ n) + O(1) = O(\lg n) \]
Build-Max-Heap

Next we build a full heap from an unsorted sequence

Build-Max-Heap(A)
for i = \text{floor}(A\text{.length}/2) \text{ to } 1
Heapify(A, i)
Build-Max-Heap

Red part is already Heapified
Build-Max-Heap

Correctness:
Base: Each alone leaf is a max-heap
Step: if A[i] to A[n] are in a heap, then Heapify(A, i-1) will make i-1 a heap as well
Termination: loop ends at i=1, which is the root (so all heap)
Build-Max-Heap

Runtime?
Build-Max-Heap

Runtime?

$O(n \lg n)$ is obvious, but we can get a better bound...

Show ceiling($n/2^{h+1}$) nodes at any height $'h'$
Build-Max-Heap

Heapify from height 'h' takes $O(h)$

$$\sum_{h=0}^{\log n} \lceil \frac{n}{2^{h+1}} \rceil \ O(h)$$

$$= O(n \sum_{h=0}^{\log n} \lceil \frac{h}{2^{h+1}} \rceil)$$

$$= O(n 4/2) = O(n)$$
Heapsort

Heapsort(A):
Build-Max-Heap(A)
for i = A.length to 2
    Swap A[1], A[i]
    A.heapsize = A.heapsize – 1
Max-Heapify(A, 1)
Heapsort
Heapsort

Runtime?
Heapsort

Runtime?

Run Max-Heapify $O(n)$ times
So... $O(n \lg n)$
Priority queues

Heaps can also be used to implement priority queues (i.e. airplane boarding lines)

Operations supported are: Insert, Maximum, Extract-Max and Increase-key
Priority queues

Maximum(A):
  return A[1]

Extract-Max(A):
  max = A[1]
  A.heapsize = A.heapsize – 1
  Max-Heapify(A, 1),    return max
Priority queues

Increase-key(A, i, key):
A[i] = key
while (i > 1 and A[floor(i/2)] < A[i])
    swap A[i], A[floor(i/2)]
i = floor(i/2)

Opposite of Max-Heapify... move high keys up instead of low down
Priority queues

Insert(A, key):
A.heapsize = A.heapsize + 1
A[A.heapsize] = -∞
Increase-key(A, A.heapsize, key)
Priority queues

Runtime?

Maximum =
Extract-Max =
Increase-Key =
Insert =
Priority queues

Runtime?

Maximum = $O(1)$
Extract-Max = $O(lg \ n)$
Increase-Key = $O(lg \ n)$
Insert = $O(lg \ n)$
### Sorting comparisons:

<table>
<thead>
<tr>
<th>Name</th>
<th>Average</th>
<th>Worst-case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion[s,i]</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge[s,p]</td>
<td>$O(n \lg n)$</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>Heap[i]</td>
<td>$O(n \lg n)$</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>Quick[p]</td>
<td>$O(n \lg n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Counting[s]</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
</tr>
<tr>
<td>Radix[s]</td>
<td>$O(d(n+k))$</td>
<td>$O(d(n+k))$</td>
</tr>
<tr>
<td>Bucket[s,p]</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Sorting comparisons:

https://www.youtube.com/watch?v=kPRA0W1kECg

Quick Sort (LR ptrs) - 454 comparisons, 670 array accesses, 1.00 ms delay

http://panthema.net/2013/sound-of-sorting