CIS 313

week of Oct 8

third week of the term
priority queues

• chapter 6
• abstract operations (implementation independent)
• maintains a set $S$ of elements
• operations
  • insert($x$)
  • max (or returnMax)
  • extractMax (removes it)
  • increaseKey($x,k$) (set key of $x$ to a new larger value)
  • -OR- insert, min, extractMin, decreaseKey
can sort with priority queue

```plaintext
PQSort(array A)  //array A has n elements
create PQ Q
for i=1 to n
    Q.insert(A[i])
for i = n down to 1
    A[i] = Q.extractMax
```

cannot analyze time without implementation
unordered list implementation of PQ

- simple
- insert(x) is $O(1)$
- extractMax is $O(n)$
- What does PQSort look like?
  - selection sort
  - time $O(n^2)$, work done in second loop
ordered list implementation of PA

• also simple
• insert(x) is $O(n)$
• extractMax is $O(1)$
• What does PQSort look like?
  • insertion sort
  • time $O(n^2)$, work done in first loop
binary heap implementation of PQ

• most common implementation
• operations are $O(\log n)$
• uses a binary tree structure
• except that the tree is stored in an array with no pointers
• it is an *implicit* tree, children and parents inferred from location in array

• PQSort becomes *heapsort*
binary heap

- stored in array
- item located in position $i$
  - parent in location $\lfloor i/2 \rfloor$
  - left child in position $2i$
  - right child in position $2i + 1$
- tree is complete
  - all nodes have two children, except maybe parent of “last” one
- tree maintains heap property
  - value stored at location $i$ is greater than or equal to values stored in both its children

height of tree is $O(\log n)$
binary heap insertion

- put new value $x$ at end of array, extending its size by 1
- value $x$ is now viewed as being at the bottom of the tree
- if $x$ violates heap property (if larger than parent), swap with parent
- repeat until no violation
- time is proportional to height of tree, which is $O(lg n)$

- text handles this differently, they insert $-\infty$ and then use heap-increase-key to the new value
pseudo-code for insert

```plaintext
insert(x):

heapsize++
A[heapsize]=x

i = heapsize
while i>1 and A[i]>A[parent(i)]
    swap A[i] and A[parent(i)]
i = parent(i)
```

sometimes called “sift-up” or “bubble-up”
Binary Heap: Insert Operation

![Binary Heap Diagram]

- Viewed as a binary tree:
  - Before insertion: 1, 2, 3, 4, 5, 6, 7
  - After insertion: 16, 11, 12, 8, 10, 9, 14

- Viewed as an array:
  - Before insertion: 1, 2, 3, 4, 5, 6, 7
  - After insertion: 16, 11, 14, 8, 10, 9, 12
heap extract-max (deletion)

• similar but element moves down
• idea: remove and return root (in location 1 of the tree)
• move rightmost element into that empty location …
• … and reduce the heapsize
• tree shape is maintained but root location may violate heap property
• note: rest of tree still has heap property
• swap node with larger (why) of its children
• repeat while heap property violated until leaf hit
• called "sift-down" or "bubble-down"
text algorithm

MAX-HEAPIFY(A, i)
    // Input: A: an array where the left and right children of i root heaps (but i may not), i: an array index
    // Output: A modified so that i roots a heap
    // Running Time: O(log n) where n = heap-size[A] – i
1 l ← LEFT(i)
2 r ← RIGHT(i)
4    largest ← l
5 else largest ← i
7    largest ← r
8 if largest ≠ i
9    exchange A[i] and A[largest]
10 MAX-HEAPIFY(A, largest)
first attempt at sorting

1. for each element $x$, insert $x$ into a heap
   - time per insert $O(lg\ n)$, total $O(n\ lg\ n)$
   - this can be made much faster
   
2. while the heap is not empty, extract-max
   - output is a sorted list (reversed)
   - each extract-max is $O(lg\ n)$, total $O(n\ lg\ n)$
   - cannot be made faster

BUILDHEAP uses deletion idea to get linear overall time
buildheap code

**Build-Max-Heap(A)**

// Input: A: an (unsorted) array
// Output: A modified to represent a heap.
// Running Time: O(n) where n = length[A]
1 heap-size[A] ← length[A]
2 for i ← [length[A]/2] downto 1
3 MAX-HEAPIFY(A, i)

**time analysis**

if tree has height H=\lg n
- all nodes at level k take time H-k to sift down
- there are 2^k nodes at level k
- total time is \(\sum_{0}^{H} 2^k (H - k)\)
- can show this is at most 2n
now heapsort

\[
\textbf{Heap-Sort}(A)
\]

// \textit{Input}: A: an (unsorted) array
// \textit{Output}: A modified to be sorted from smallest to largest
// \textit{Running Time}: $O(n \log n)$ where $n = \text{length}[A]
1 \textbf{Build-Max-Heap}(A)
2 \textbf{for} i = \text{length}[A] \textbf{downto} 2
3 \textbf{exchange} A[1] \text{ and } A[i]
4 \text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1
5 \textbf{Max-Heapify}(A, 1)