Heap Lecture
Chapter 6

Pseudo-code is from Cormen et al. textbook.
Heap Basics

- **Structure:**
  - Nearly complete binary tree
  - Convenient array representation

- **HEAP Property:** (for MAX HEAP *)
  - Parent’s label not less than that of each child

* analogous property and operations for MIN HEAP
Operations on a Heap

assuming array representation

**MAX-HEAPIFY:**

- for a given node that is the root of a subtree, if both subtrees of that node are already HEAPs, MAX-HEAPIFY enforces the maximum HEAP PROPERTY via “downward swaps” so that the node together with its subtrees form a MAX HEAP

```
MAX-HEAPIFY(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] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Operations on a Heap

assuming array representation

- **BUILD-MAX-HEAP:**
  - builds a MAX HEAP from scratch using MAX-HEAPIFY

**BUILD-MAX-HEAP(A)**

1. \(A.\text{heap-size} = A.\text{length}\)
2. \(\text{for } i = \lfloor A.\text{length}/2 \rfloor \text{ downto } 1\)
3. \(\text{MAX-HEAPIFY}(A, i)\)
Building a Max Heap using
MAX-HEAPIFY vs. MAX-HEAP-INSERT

- **MAX-HEAPIFY**
  - swaps down
  - compares parent with both children before each swap

  - number of height levels
  - bound on number of nodes in this level

  \[ \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{h+1}} \leq \frac{n}{2} \]

  \[ = O\left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O(n2) = O(n) \]

- **MAX-HEAP-INSERT**
  - swaps up
  - compares parent with one child before each swap

  - bound on number of nodes in this level

  \[ = \left( O(n \lg n) \sum_{h=0}^\infty \frac{1}{2^h} \right) - \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^{h+1}} \right) \]

  \[ = O(n \lg n) \sum_{h=0}^\infty \frac{1}{2^h} = 2 \Rightarrow O(n \lg n) \]

Asymptotic worst-case running time of BUILD-MAX-HEAP using MAX-HEAPIFY is in \( O(n) \). However, using MAX-HEAP-INSERT the time would only be in \( O(n \lg n) \).
Operations on a Heap

assuming array representation

- **HEAPSORT:**
  - sorts an array by first using BUILD-MAX-HEAP then repeatedly swapping out root and calling MAX-HEAPIFY

**HEAPSORT(A)**

1. `BUILD-MAX-HEAP(A)`
2. `for i = A.length downto 2`
4. `A.heap-size = A.heap-size - 1`
5. `MAX-HEAPIFY(A, 1)`

```
16 14 10 8 7 9 3 2 4 1
1 2 3 4 5 6 7 8 9 10
```
Operations on a Max Heap

assuming array representation

- **MAX-HEAPIFY**:
  Asymptotic worst-case running time is in $O(\lg n)$. For a node at height $h$, time is in $O(h)$.

- **BUILD-MAX-HEAP**: 
  Asymptotic worst-case running time is in $O(n \lg n)$. However, this is a loose bound! Time is also in $O(n)$.

- **HEAPSORT**: 
  Asymptotic worst-case running time is in $O(n \lg n)$. 

$T(n) \leq T(2n/3) + \Theta(1)$ is in $O(\lg n)$ using Master Theorem

$\begin{align*}
\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{h+1}} & \leq \sum_{h=0}^{\infty} \frac{n}{2^{h+1}} = \frac{n}{1 - \frac{1}{2}} = 2n
\end{align*}$

$\begin{align*}
\sum_{h=0}^{\infty} \frac{h}{2^h} & = \sum_{h=0}^{\infty} \frac{2^h}{2^{h+1}} = \sum_{h=0}^{\infty} (2^h 2^{-1}) = \sum_{h=0}^{\infty} \frac{2^h}{2} = \sum_{h=0}^{\infty} \frac{1}{2} = 1
\end{align*}$

$\sum_{h=0}^{\infty} \frac{h}{2^h} = 1
$
Operations on a Max Heap

assuming array representation

- **PRIORITY QUEUE SUPPORT:**
  - **MAX-HEAP-INSERT**
    - adds new leaf to the tree and then “swaps up” to restore MAX HEAP PROPERTY
  - **HEAP- MAXIMUM**
    - MAX HEAP PROPERTY guarantees that maximum is at the root of a MAX HEAP
  - **HEAP- EXTRACT-MAX**
    - removes the maximum value from the root by swapping it out
    - restores MAX HEAP PROPERTY using MAX-HEAPIFY

**Applications:** Job Scheduling, Event Scheduling
Operations on a Max Heap
assuming array representation

- **PRIORITY QUEUE SUPPORT:**
  - **MAX-HEAP-INSERT**
    Asymptotic worst-case running time is in $O(lg n)$. For a node at height $h$, time is in $O(h)$.

- **HEAP- MAXIMUM**
  Asymptotic worst-case running time is in $O(1)$.

- **HEAP- EXTRACT-MAX**
  Asymptotic worst-case running time is in $O(lg n)$. For a node at height $h$, time is in $O(h)$.

**MAX-HEAP-INSERT**($A, key$)

1. $A.heap-size = A.heap-size + 1$
3. **HEAP-INCREASE-KEY**($A, A.heap-size, key$)

**HEAP-INCREASE-KEY**($A, i, key$)

1. if $key < A[i]$
2. error “new key is smaller than current key”
3. $A[i] = key$
4. while $i > 1$ and $A[PARENT(i)] < A[i]$
5. exchange $A[i]$ with $A[PARENT(i)]$
6. $i = PARENT(i)$

**HEAP-EXTRACT-MAX**($A$)

1. if $A.heap-size < 1$
2. error “heap underflow”
3. $max = A[1]$
5. $A.heap-size = A.heap-size - 1$
6. **MAX-HEAPIFY**($A, 1$)
7. return $max$