Heapsort

Textbook: Chapter 7, Section 7.1, 7.2 and 7.3

CSCE310: Data Structures and Algorithms
www.cse.unl.edu/~choueiry/S01-310/

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
choueiry@cse.unl.edu, Tel: (402)472-5444
• Insertion sort  \( \sqrt{\cdot} \)
  Sorts in \( \Omega(n), O(n^2) \)
  Sorts in place, in array

• Merge-sort  \( \sqrt{\cdot} \)
  Sorts \( \Theta(n \lg n) \)
  Uses space, external to array

• Heapsort,  \( \leftarrow \)
  Sorts \( O(n \lg n) \)
  Sorts in place, in array
  Constant number of array elements stored outside input array at any time
  Uses \textbf{heap}  \( \leftarrow \) new data structure
(Binary) **Heap**: Array that can be viewed as a **complete** binary tree

- Node in the tree $\leadsto$ element in array storing same value
- Tree complete: filled on all levels, except possibly lowest level (from left up to a point)
- $A$ is the array, $heap - size[A], A[1 \ldots length[A]], heap - size[A] \leq length[A]$, elements in $A$ beyond $A[heap - size[A]]$ are not elements of the heap
• Root of tree is $A[1]$

• Node index $i$, indices of parent/children: $Parent(i)$, $Left(i)$, and $Right(i)$

• Index of $Parent(i)$ is $\lfloor i/2 \rfloor$

• Index of left child, $Left(i)$ is $2i$

• Index of right child, $Right(i)$ is $2i + 1$

• Binary representation: shift left one bit (+ one), shift right one bit
• **Heap property:** \( \forall i, A[\text{Parent}(i)] \geq A[i] \), except root
  Value of a node at most value of its parent

• Largest element in a heap is stored at the root

• Subtrees rooted at a node contain smaller values than the node’s

Exercise: 7.1-6. Is \( \langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle \) a heap?
• **Height** of a node: \#edges on longest simple path from node to a leaf

• Height of the tree = height of its root

• Heap of \( n \) elements, height is \( \Theta(lg \ n) \)

• Basic operations (remember?) on heaps run in time proportional to height, thus \( O(lg \ n) \)

Exercises: 7.1-1 and 7.1-2
Five basic procedures

1. **Heapify** maintains heap property
   runs in $O(\lg n)$

2. **Build-Heap** produces a heap from an unordered input array
   linear in time

3. **Heapsort** sorts an array in place
   runs in $O(n \lg n)$

4. **Extract-Max** and 5. **Insert** allow heap to be used as priority
   queue
   run in $O(\lg n)$
**Heapify** maintains heap property

**Input:** An array $A$, and an index $i$ in array
Assumption: Binary tree rooted at $Left(i)$ and $Right(i)$ are heaps
but $A[i]$ smaller than its children (i.e., heap property violated)

**Output:** heap $A$, heap property restored
$A[i]$ pushed down so that subtree rooted at $i$ becomes a heap
**Heapify** $(A, i)$

1. $l \leftarrow \text{Left}(i)$
2. $r \leftarrow \text{Right}(i)$
3. if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ then $\text{largest} \leftarrow l$
4. else $\text{largest} \leftarrow i$
5. if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ then $\text{largest} \leftarrow r$
6. if $\text{largest} \neq i$
7. then exchange $A[i] \leftarrow A[\text{largest}]$
8. $\text{Heapify}(A, \text{largest})$

Each call stores in $\text{largest}$ index of largest of
$A[i], A[\text{Left}(i)], A[\text{Right}(i)]$
If $A[i]$ is largest, we have a heap!
Otherwise, $A[i]$ swapped with $A[\text{largest}] \rightarrow i$ satisfies heap property
However, $\text{largest}$ may violate heap property $\rightarrow$ recursive call
**HEAPIFY**($A, i$)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
   then $\text{largest} \leftarrow l$
4. elif $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$
   then $\text{largest} \leftarrow r$
5. if $\text{largest} \neq i$
   then exchange $A[i] \leftarrow A[\text{largest}]
   \text{HEAPIFY}(A, \text{largest})$

**Exercise: 7.2-1. Heapify($A, 3$) on**

$A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$
Running time of Heapify

\textsc{Heapify}(A, i)
1 \hspace{1em} l \leftarrow \text{Left}(i)
2 \hspace{1em} r \leftarrow \text{Right}(i)
3 \hspace{1em} \text{if } l \leq \text{heap-size}[A] \text{ and } A[l] > A[i]
4 \hspace{1em} \text{then } \text{largest} \leftarrow l
5 \hspace{1em} \text{else } \text{largest} \leftarrow i
6 \hspace{1em} \text{if } r \leq \text{heap-size}[A] \text{ and } A[r] > A[\text{largest}]
7 \hspace{1em} \text{then } \text{largest} \leftarrow r
8 \hspace{1em} \text{if } \text{largest} \neq i
9 \hspace{1em} \text{then exchange } A[i] \leftarrow A[\text{largest}]
10 \hspace{1em} \text{\textsc{Heapify}}(A, \text{largest})

Subtree of size $n$, rooted at node $i$:

- $\Theta(1)$ to fix up relationship between $A[i], A[\text{Left}(i), A[\text{Right}(i)]$
- time to Heapify a subtree of at most $2n/3$ nodes
- Running time $T(n) \leq T(2n/3) + \Theta(1)$
- Solution: case 2 of Master theorem $T(n) = O(\log n)$
- Alternatively, in terms of $h$, $T(n) = O(h)$
Building a heap: use Heapify

Converts array $A[1 \ldots n]$ ($n = \text{length}[A]$) into a heap.

Elements in subarray $A[[n/2]] + 1 \ldots n$ are all leaves, 1-element heap, no need to heapify them (they’ll stay where there are)

Heapify the remaining elements, from $\lfloor n/2 \rfloor$ downto 1

\begin{verbatim}
BUILD-HEAP(A)
1 heap-size[A] ← length[A]
2 for i ← \lfloor length[A]/2 \rfloor downto 1
do HEAPIFY(A, i)
\end{verbatim}

Build a heap with $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
Running time of **Build-Heap**

Simple upper bound on running time:

- Each call to **Heapify** costs $O(lg\ n)$
- There are at most $O(n)$ calls
- Running time at most $O(n\ lg\ n)$

- Upper bound correct, but not tight:
  tighter upper bound can be computed.

Observation: Cost of call to **Heapify** depends on height of node and heights of most nodes are small!

One can prove running time is in $O(n)$