Lecture 8

Today:

- Data Structures
  - Priority Queues
  - P.Q.'s from Heaps
  - Heapsort

Admin:

- Quiz 1 Announcement Posted
  - Quiz Review Tuesday
    - 4-6 pm in 4.370
    - 1 two-sided cheat sheet
      - 8.5 x 11, handwritten.
Data Structures

New paradigm for algorithm design

- Outsource Data Management -

Algorithm

Data Structure

update instructions

response
**Data Structure:** specified by collection of actions (queries/updaters) it supports.

**Algorithm:**
- Should only use prescribed actions — even if it knows how data is stored inside structure.
- Should be correct no matter how queries are processed.
- Should be efficient if data structure is efficient.
**Trivial Examples**

- **Array**: 
  - CREATE(n) → \( A[1..n] \)
  - VALUE(A, i) → \( A[i] \)
  - UPDATE(A, i, x) : \( A[i] \leftarrow x \)

- **Stack**: 
  - CREATE() → S
  - Push(S, x) → push x onto S;
  - Pop(S) → returns top of S;
  - Empty?(S) → is S = \( \emptyset \) ?

- **Queue**: 
  - CREATE() → Q;
  - Add(Q, x) → adds x to end of Q
  - Extract(Q) → removes front element from Q & returns it;
Today's Data Structure

Priority Queue:

CREATE-Q(A); Returns new Q containing elements of array A, keyed by A[i].

EXTRACT-MAX(Q); finds largest element in Q; removes it & returns it.

MAX(Q); returns largest element in Q;

INSERT(Q,x); inserts element x into Q;

EMPTY?(Q); returns true if Q=∅;
Using a Priority Queue To Sort

\[ \text{P-Q-Sort} \ (A[1..n]) \]

\[ \textbf{Q} \leftarrow \text{CREATE} \ (A) ; \]

\[ i \leftarrow n ; \]

\[ \text{While not Empty (Q) do} \]

\[ B[i] \leftarrow \text{EXTRACT-MAX} \ (Q) ; \]

\[ i \leftarrow i - 1 ; \]

\[ \text{return} \ (B) ; \]

- Always correct sorting algorithm;
- Running time depends on CREATE, EXTRACT-MAX, EMPTY?
Naive Implementations of Priority Queue

1) Unsorted Array:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>EXTRACT-MAX</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>MAX</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Empty?</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

Time to sort = $\Theta(n^2)$

2) Sorted Array:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>EXTRACT-MAX</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

Time to sort: $\Theta(n \log n)$
Heaps

- A clever implementation of Priority Queues.

  CREATE  - $\Theta(n)$
  EXTRACT-MAX  - $\Theta(\log n)$
  MAX  - $\Theta(1)$
  INSERT  - $\Theta(\log n)$
  EMPTY  - $\Theta(1)$

- Sorting using Heaps (called HEAPSORT) is a new algorithm in time $\Theta(n\log n)$ in comparison model.
Looking behind the well

- Heap is an array, viewed as a binary tree.

- At any point of time, a heap with $n$ elements is an array $H[1..n]$. 

```
    root
    /  \
   /    \
H[1]   \
   /  \
   / \
```

H[$2i$], H[$2i+1$]
- If \( i < \frac{n}{2} \) then left child of \( H[i] \) is \( H[2i] \) and right child is \( H[2i+1] \).

- **Heap Property**: Always Maintained

  \[ H[i] \geq H[2i], \ H[2i+1] \]

- **Example**: Heap containing \( \{1, 2, 3, 4, 5, 6, 7, 8\} \)
Another heap could be

```
5
  
7
  6
  4
  1
  2
```

How to maintain heaps with heap property?

Key ingredient: `HEAPIFY(i);`
Assumes subtree at $2i$ and $2i+1$ form valid heaps & delivers valid heap at $i$
Example

heapify

Exchange 2 <-> 11

< than 2 <-> 4

Not yet a heap
Pseudo code

\[ \text{HEAPIFY}[i]; \]

\text{if}\ H[2i+1] < H[2i], \ j \leftarrow 2i
\text{else}\ j \leftarrow 2i+1

\text{if}\ H[i] > H[j] \text{ then done}
\text{else Exchange } H[i] \leftrightarrow H[j]

\text{HEAPIFY}[j]; \]
Running time: Clearly $O(\log n)$.

Better bound $\Theta(\log \frac{n}{i})$;

Can now create heap

\[
\begin{align*}
\text{CREATE} & (A[1..n]) \ ; \\
H[\cdot] & \leftarrow A[\cdot] \ ; \\
\text{for } i & \leftarrow \frac{n}{2} \ \text{down to } 1 \ \text{do} \\
& \text{HEAPIFY}[i] \ ;
\end{align*}
\]
Example: \{1, 2, 3, 4, 5, 6, 7, 8\}
\[ \text{Time complexity} \]
\[ = \sum_{i=1}^{\frac{n}{2}} \log \frac{n}{i} = \Theta(n) \]
**Extract-Max**

Simple!
\[ x \leftarrow H[1]; \]
\[ H[1] \leftarrow H[n]; \]
\[ n \leftarrow n-1; \]
\[ \text{HEAPIFY}[1]; \]
\[ \text{Return} \ (x); \]

Time complexity = \( \Theta(\log n) \)

**Heap Insert:** Exercise.
Moral of this story:

- Sometimes a good way to design an algorithm is to design a good data structure & use it.
- There is a variety of clever data structures out there....
- First thing you think of may not be the best.