Min Priority Heap (Chapter 6)

- Just ensure value at node is greater than everything in subtree below it
- Use array to store complete tree
  
Basic idea (using max instead of min):

If you want a lower amount of overhead (smaller constant):

If you want min and insert in \( O(\log n) \) and do not need search
Max-Heapify

\begin{algorithm}
\textbf{Max-Heapify} (A, i)
\begin{algorithmic}[1]
\STATE \textbf{if } A[i] \neq \text{largest} \\textbf{then}
\STATE \quad \text{\textbf{if } } A[i] \text{ < } \text{ largest} \\textbf{then}
\STATE \quad \quad i = \text{largest}
\STATE \quad \textbf{else if } A[i] \geq \text{ largest} \\textbf{then}
\STATE \quad \quad i = \text{largest}
\STATE \textbf{end if}
\STATE \textbf{if } A[largest] \neq \text{ largest} \\textbf{then}
\STATE \quad \textbf{if } A[largest] < A[\text{left}(i)] \\text{ and } A[largest] < A[\text{right}(i)]
\STATE \quad \quad i = \text{largest}
\STATE \textbf{end if}
\STATE \textbf{end if}
\STATE \textbf{end if}
\STATE \textbf{exchange } A[i] \text{ with } A[largest]
\STATE \textbf{Max-Heapify} (A, largest)
\end{algorithmic}
\end{algorithm}

---

Accessing Parent and Children

- Textbook starts array at 1
- Accessing parent?
- Left child?
- Right child?
- Accessing parent? (where root is 0)
- Maximum height of tree with n nodes?
```c
if (heap-size < 1)
    error "heap underflow"

max

A:

heap-size

D

AŒ1

AŒA:

heap-size

=: NUL

M

AX

HEAPIFY

A; 1/

return

max

MAX-HEAPIFY(A, i)

V. heap-size = V. heap-size - 1

if V. heap-size = [1] V.

[1] V. = max

error "heap underflow"

1 if V. heap-size > 1

HEAP-EXTRACT-MAX(A)
```

---

**Extracting Max**

- Running time?
- Why that starting point?
- A. heap-size versus A. length

```c
MAX-HEAPIFY(A, i)

for i = length/2 downto 1
    A. heap-size = A. length

BUILD-MAX-HEAP(A)
```
How can we use this to insert?

```plaintext
heap-increase-key(A, i, key)
```

1. `heap-increase-key` takes an array `A`, an index `i`, and a new key.
2. If the new key is smaller than the current key, a `ERROR` is indicated.
3. Otherwise, the key at `A[i]` is exchanged with the key at its parent's index, and the process repeats.
4. The exchange occurs as long as `A[i]` is greater than its parent's key.
5. The process continues until the `key` is in the correct position.
6. The function returns when the operation is completed.

The code snippet outlines the algorithm for increasing the key at a specified index in a max heap.