Last Time

- Binary Search Trees
Today

- Heaps
- Maximum (and Minimum)
- Mean
- Median
Recall: Binary Search Trees are constructed with $O(n \log n)$

- **Search** $O(\log n)$, **Insert** $O(\log n)$, **Delete** $O(\log n)$
- **Maximum** $O(\log n)$
- The data structure can be augmented to speed up **Maximum**.
Heaps are used when **maximum** is going to be heavily used.

Heaps are Binary Trees.

**Max-Heap Property**

- Given a Heap with height $h$, the top $h - 1$ levels of the heap must be complete.
- Heaps have the property that $T.key > T.right.key$ and $T > T.left.key$
Heap Example

```
16
  14  10
   8  7  9  3
```
Heap Example

![Heap Example Diagram]
Heap Example

```
16
14   10
  8   7   9   3
  2   4
```
Heap Example
The **Max heap property** allows compact representation of a heap as an array.

- \text{Parent}(i) = \left\lfloor i/2 \right\rfloor
- \text{Left}(i) = 2i
- \text{Right}(i) = 2i + 1
Heap Example

A[i] 16 14 10 8 7 9 3 2 4 1

i 1 2 3 4 5 6 7 8 9 10
This representation of a Heap as an array can be applied to any Binary Tree.

However, the **max-heap property** guarantees that this representation will be **compact**.

- This is due to the property that the top $height - 1$ levels of the tree are complete.

An arbitrary Binary Tree has a worst-case array size of $O(n^2)$. 
Heap Operations

- **Maximum** - Return the maximum.
- **MaxHeapify** - Given that the children of $i$ are max-heaps, maintain the **max-heap property**.
- **BuildMaxHeap** - Given an unsorted array, construct a max-heap.
- **MaxHeapInsert** - Insert an element into a max-heap.
- **HeapExtractMax** - Remove and return the maximum element from a max-heap.
- **HeapIncreaseKey** - Increase the value of an element in the max-heap. Used in **priority queues**.
- **HeapSort** - Use a max-heap to sort an array.
Heap Maximum

**Maximum(A)**

```markdown
return A[1]
```

- The maximum value is always at the root of a max-heap.
- $\text{Maximum}(A) = \Theta(1)$
Max Heapify

**MaxHeapify**\( (A, i) \)

\[
\begin{align*}
l & \leftarrow \text{Left}(i) \\
r & \leftarrow \text{Right}(i) \\
\text{if } l \leq \text{size}(A) \text{ and } A[l] > A[i] \text{ then} & \\
& \quad \text{largest} \leftarrow l \\
\text{else} & \\
& \quad \text{largest} \leftarrow i \\
\text{end if} \\
\text{if } r \leq \text{size}(A) \text{ and } A[r] > A[\text{largest}] \text{ then} & \\
& \quad \text{largest} \leftarrow r \\
\text{end if} \\
\text{if } \text{largest} \neq i \text{ then} & \\
& \quad \text{swap } A[i] \leftrightarrow A[\text{largest}] \\
& \quad \text{MaxHeapify} (A, \text{largest}) \\
\text{end if}
\end{align*}
\]
MaxHeapify Example

MaxHeapify(A, 1)

4
16 10
14 7 9 3
2 8 1
MaxHeapify Example

MaxHeapify(A,2)
MaxHeapify Example

MaxHeapify(A, 4)

[Diagram of a heap with numbers 16, 14, 10, 7, 9, 3, 4, 2, 8, 1]
MaxHeapify Example

MaxHeapify(A,4)
MaxHeapify Runtime

- MaxHeapify runtime is $\Theta(height) = \Theta(log n)$.
- $height$ of a max-heap is $\Theta(log n)$
- OR... Runtime: $T(n) \leq T(2n/3) + \Theta(1) = O(log n)$
建堆

BuildMaxHeap(A)

for i ← n downto 1 do
    MaxHeapify(A, i)
end for

- 这会调用MaxHeapify函数来堆化叶子节点以及树的内部节点。
- 堆的叶子节点是通过 $\lfloor n/2 \rfloor + 1$ 从 $n$ 所指定的。
Heap Example

A[i] 16 14 10 8 7 9 3 2 4 1

i 1 2 3 4 5 6 7 8 9 10
Build Max Heap

**BuildMaxHeap**

\[\text{BuildMaxHeap}(A)\]

\[
\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1 \text{ do}
\]

MaxHeapify(A,i)

end for
We make \( \frac{n}{2} \) calls to a function that is \( O(\log n) \), so \( O(n \log n) \).
We make $n/2$ calls to a function that is $O(\log n)$, so $O(n \log n)$.

A good guess, and true. However, it’s not a tight bound.

The runtime of $\text{MaxHeapify}$ depends on the height of the node $O(h)$, and most nodes have a small height. While $h = O(\log n)$, $h$ is usually much smaller than $\log n$.

- Twice as many nodes have $h = 1$ than have $h = 2$. 

Runtime of \textbf{BuildMaxHeap}

- What is the height of an $n$ element heap?
What is the height of an $n$ element heap?

$\lceil \log n \rceil$. 
How many nodes can a heap of size $n$ have with height $h$?
How many nodes can a heap of size $n$ have with height $h$?

$\left\lceil \frac{n}{2^{h+1}} \right\rceil$

This requires a slightly tricky proof. By induction show that the number of leaves ($h = 0$) is $\left\lceil \frac{n}{2} \right\rceil$. Then show that it holds for $h + 1$. 
Runtime of **BuildMaxHeap**

- How many nodes can a heap of size \( n \) have with height \( h \)?
  \[ \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \]

- This requires a slightly tricky proof. By induction show that the number of leaves (\( h = 0 \)) is \( \left\lceil \frac{n}{2} \right\rceil \). Then show that it holds for \( h + 1 \).

- The runtime of **MaxHeapify** on a node of height \( h \) is \( O(h) \).
Runtime of \texttt{BuildMaxHeap}

- How many nodes can a heap of size $n$ have with height $h$?
  - $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
- This requires a slightly tricky proof. By induction show that the number of leaves ($h = 0$) is $\left\lfloor \frac{n}{2} \right\rfloor$. Then show that it holds for $h + 1$.
- The runtime of \texttt{MaxHeapify} on a node of height $h$ is $O(h)$.
- Thus \texttt{BuildMaxHeap} takes:

$$
\sum_{h=0}^{\lceil \log n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^{h+1}} \right) = O \left( n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right)
$$

$$
\sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \cdot (1/2)^h = \frac{1/2}{(1 - 1/2)^2} = 2
$$
Runtime of \texttt{BuildMaxHeap}

- How many nodes can a heap of size \( n \) have with height \( h \)?
  \[
  \left\lceil \frac{n}{2^{h+1}} \right\rceil
  \]
- This requires a slightly tricky proof. By induction show that the number of leaves \((h = 0)\) is \( \left\lceil \frac{n}{2} \right\rceil \). Then show that it holds for \( h + 1 \).
- The runtime of \texttt{MaxHeapify} on a node of height \( h \) is \( O(h) \).
- Thus \texttt{BuildMaxHeap} takes:

  \[
  \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}} \right) = O \left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)
  \]

  \[
  O \left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n \cdot 2) = O(n)
  \]
**Correctness of BuildMaxHeap**

**BuildMaxHeap(A)**

```
for i ← ⌊n/2⌋ downto 1 do
    MaxHeapify(A, i)
end for
```

- **Loop Invariant** At the start of each iteration of the for loop, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.
- **Initialization** $i = \lfloor n/2 \rfloor$ Each node $\lfloor n/2 \rfloor + 1, \ldots, n$ is a leaf, and thus the root of a max-heap.
Correctness of \textbf{BuildMaxHeap}

\begin{tabular}{|l|}
\hline
\textbf{BuildMaxHeap}(A) \\
\hline
\textbf{for} \( i \leftarrow \lfloor n/2 \rfloor \) \textbf{downto} 1 \textbf{do} \hfill \\
\hspace{2cm} \textbf{MaxHeapify}(A,i) \hfill \\
\textbf{end for} \\
\hline
\end{tabular}

\begin{itemize}
    \item \textbf{Loop Invariant} At the start of each iteration of the for loop, each node \( i + 1, i + 2, \ldots, n \) is the root of a max-heap.
    \item \textbf{Maintenance} The children of \( i \) have indices \( \text{LEFT}(i) \) \( i \) and \( \text{RIGHT}(i) > i \), and are thus roots of max-heaps. Therefore \( \text{MaxHeapify}(A,i) \) will make \( i \) the root of a max-heap, and preserve the max-heap property for all nodes \( k > i \).
\end{itemize}
Correctness of BuildMaxHeap

**BuildMaxHeap(A)**

```
for i ← ⌊n/2⌋ downto 1 do
    MaxHeapify(A,i)
end for
```

- **Loop Invariant** At the start of each iteration of the for loop, each node \( i + 1, i + 2, \ldots, n \) is the root of a max-heap.
- **Termination** When the for loop finishes \( i = 0 \). Thus each node \( 1, 2, \ldots, n \) is the root of a max-heap. Specifically, node 1 is.
**Heap Increase Key**

\[ \text{HeapIncreaseKey}(A, i, key) \]

\[
A[i] \leftarrow key \\
\text{while } i > 1 \text{ and } A[\text{Parent}(i)] < A[i] \text{ do} \\
\quad \text{swap } A[i] \leftrightarrow A[\text{Parent}(i)] \\
\quad i \leftarrow \text{Parent}(i) \\
\text{end while}
\]

- \( O(\log n) \) - Heap traversal
Max Heap Insert

\[ \text{MaxHeapInsert}(A, \text{key}) \]

- \( \text{size}(A) \leftarrow \text{size}(A) + 1 \)
- \( A[\text{size}(A)] \leftarrow -\infty \)
- \( \text{HeapIncreaseKey}(A, \text{size}(A), \text{key}) \)

- \( O(\log n) - \text{HeapIncreaseKey}(A, i, \text{key}) \)
**Heap Extract Max**

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

\[
\begin{align*}
max & \leftarrow A[1] \\
size(A) & \leftarrow size(A) - 1 \\
\text{MAXHEAPIFY}(A, 1) \\
\text{return} & \quad max
\end{align*}
\]

- $O(\log n)$ - from $\text{MAXHEAPIFY}(A, 1)$
Heap Sort

- We can use a Heap to sort an array.
- Turn the array into a heap using **BuildMaxHeap**.
- Position the **Maximum** element $n$ times to construct a sorted array.
Heap Sort

- We can use a Heap to sort an array.
- Turn the array into a heap using $\text{BuildMaxHeap}$
- Position the $\text{Maximum}$ element $n$ times to construct a sorted array.

$\text{HeapSort}(A)$

$\text{BuildMaxHeap}(A)$

for $i \leftarrow \text{size}(A)$ downto 2 do

  swap $A[1] \leftrightarrow A[\text{size}(A)]$

  $\text{size}(A) \leftarrow \text{size}(A) - 1$

  $\text{MaxHeapify}(A, 1)$

end for
Heap Sort

- We can use a Heap to sort an array.
- Turn the array into a heap using **BuildMaxHeap**
- Position the Maximum element \( n \) times to construct a sorted array.

### HeapSort(A)

**BuildMaxHeap(A)**

```plaintext
for i ← size(A) downto 2 do
    size(A) ← size(A) − 1
    MaxHeapify(A, 1)
end for
```

- **BuildMaxHeap(A) = O(n)**
- **MaxHeapify(A) = O(log n – called \( n \) times.**
- **HeapSort(A) = O(n log n).**
Bye

Next time (10/6)
- Operations on Data Streams

For Next Class
- Homework 5 Due
- Read 13.1, 13.2, 13.3, 13.4