Last Time

- Balanced Binary Search Trees
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 \texttt{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

```
    16
   / | \
  14 10
 / | | | | |
 8 7 9 3
/ | \
2
```
Heap Example

```
    16
   / \
  14  10
 / \ / \  
8  7 9  3
/ \ / \  
2  4  
```
Heap Example

```
16
14 10
8 7 9 3
2 4 1
```
The Max heap property allows compact representation of a heap as an array.

- **Parent(i) = \lfloor i/2 \rfloor**
- **Left(i) = 2i**
- **Right(i) = 2i + 1**
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 $O(n^2)$. 
Binary Tree Example

```
A[i]  16  14  ∅  8  ∅  ∅  ∅  2  4  ∅
i    1   2   3   4   5   6   7   8   9  10
```

```
16  
   14
    8
     2  4
```

The **Max heap property** allows compact representation of a heap as an array.

- **Parent**($i$) = $\lfloor i/2 \rfloor$
- **Left**($i$) = $2i$
- **Right**($i$) = $2i + 1$
$2^{d-1}$ is the maximum number of nodes in a binary tree at a given depth $d$. 
2^{d-1} is the maximum number of nodes in a binary tree at a given depth $d$.

Base case $d = 1$ A tree of depth 1 has only a root. $2^0 = 1$. 
Representing a Heap as an array

- $2^{d-1}$ is the maximum number of nodes in a binary tree at a given depth $d$.
- **Base case** $d = 1$ A tree of depth 1 has only a root. $2^0 = 1$.
- **Inductive Step** The maximum number of nodes at depth $d + 1$ arises when each node at $d$ has the maximum number of children.
2^{d-1} is the maximum number of nodes in a binary tree at a given depth \( d \).

**Base case** \( d = 1 \) A tree of depth 1 has only a root. \( 2^0 = 1 \).

**Inductive Step** The maximum number of nodes at depth \( d + 1 \) arises when each node at \( d \) has the maximum number of children.

The maximum number of children is 2. Thus, the maximum number of nodes at depth \( d + 1 \) is double the maximum number at depth \( d \).

The maximum at depth \( d \) is \( 2^{d-1} \).

The maximum at depth \( d + 1 \) is \( 2 \cdot 2^{d-1} = 2^{(d+1)-1} (= 2^d) \).
The maximum number of nodes in a tree of depth $d$ is $2^d - 1$
The maximum number of nodes in a tree of depth $d$ is $2^d - 1$

- **Base Case** A tree with depth $d = 1$ only has one node, a root. $2^d - 1 = 2^1 - 1 = 2 - 1 = 1$. 
The maximum number of nodes in a tree of depth $d$ is $2^d - 1$

**Base Case** A tree with depth $d = 1$ only has one node, a root. $2^d - 1 = 2^1 - 1 = 2 - 1 = 1$.

**Inductive Step** The maximum number of nodes in a tree of depth $d + 1$ is the number of nodes in a tree of depth $d$ plus the maximum number of nodes at depth $d + 1$.

Max nodes in a tree of depth $d$ is $2^d - 1$

Max nodes at depth $d + 1$ is $2^{(d+1)−1} = 2^d$

$2^d + 2^d − 1 = 2 * 2^d − 1 = 2^{d+1} − 1$
The nodes at any complete depth $d$ can be uniquely indexed with the values between 1 and $2^d - 1$.

**Base Case** When $d = 1$ the tree has only a root. Thus the root is uniquely indexed by 1. $2^{1-1} = 2^0 = 1$ and $2^1 - 1 = 2 - 1 = 1$
The nodes at any complete depth $d$ can be uniquely indexed with the values between 1 and $2^d - 1$.

**Base Case** When $d = 1$ the tree has only a root. Thus the root is uniquely indexed by 1. $2^{1-1} = 2^0 = 1$ and $2^1 - 1 = 2 - 1 = 1$

**Inductive Step** Assume all nodes at depths $d$ or below are uniquely indexed between 1 and $2^d - 1$.

The $d + 1$ depth of the tree contains $2^{(d+1)-1} = 2^d$ nodes.

The range between $2^{(d+1)-1}$ and $2^{(d+1)} - 1$ contains $2^d$ indices.

$2^{d+1} - 1 - 2^{(d+1)-1} + 1 = 2^{d+1} - 2^d = 2 \times 2^d - 2^d = 2^d$

Therefore the range between $2^d$ and $2^{d+1} - 1$ can contain enough elements for depth $d + 1$ and does not overlap with the elements at lower depths.
Assuming we have a complete graph with $N$ nodes, can we arrange the elements compactly in an $N$ element array using the following relationships?

- $\text{Parent}(i) = \lfloor i/2 \rfloor$
- $\text{Left}(i) = 2i$
- $\text{Right}(i) = 2i + 1$

Does $j$ correspond to a node in the tree?

Does $j$ correspond to a unique node in the tree?
Does an index $j$ in the range $1$ and $2^d - 1$ correspond to a node in a complete tree?

Assume not. If there exists an index $j$ that does not correspond to a node.

Therefore $j$ is not $\text{LEFT}(i)$ nor $\text{RIGHT}(i)$ for all $1 \leq i \leq 2^d - 1$.

If $j = 1$ then $j$ corresponds to the root.

Otherwise, assume without loss of generality that there exists $i$ such that $2i \leq j$ and $i + 1$ such that $2(i + 1) \geq j$.

Therefore $2i \leq j \leq 2(i + 1)$.

Thus, $j$ can be $2i$ in which case it is $\text{LEFT}(i)$

Thus, $j$ can be $2i + 1$ in which case it is $\text{RIGHT}(i)$

Thus, $j$ can be $2i + 2 = 2(i + 1)$ in which case it is $\text{LEFT}(i+1)$
Representing a Heap as an array

- Does an index \( j \) in the range 1 and \( 2^d - 1 \) correspond to a unique node in a complete tree?
- Assume not. If there exists an index \( j \) that corresponds to two nodes.
- \( j \) must be the child of two different nodes, \( i \) and \( i' \) where \( i \neq i' \).
- Both Left children. \( 2i \neq 2i' \) if \( i \neq i' \).
- Both Right children. \( 2i + 1 \neq 2i' + 1 \) if \( i \neq i' \).
- Note \( 2i \) is even and \( 2i + 1 \) is odd
- Therefore \( 2i \neq 2i' + 1 \) for any integers \( i \) and \( i' \).
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

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

\[ \text{return} \ \ A[1] \]

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

\textbf{MaxHeapify}(A, i)

\begin{itemize}
\item \( l \leftarrow \text{Left}(i) \)
\item \( r \leftarrow \text{Right}(i) \)
\item \textbf{if} \( l \leq \text{size}(A) \) and \( A[l] > A[i] \) \textbf{then}
\hspace{1cm} \text{largest} \leftarrow l
\item \textbf{else}
\hspace{1cm} \text{largest} \leftarrow i
\item \textbf{end if}
\item \textbf{if} \( r \leq \text{size}(A) \) and \( A[r] > A[\text{largest}] \) \textbf{then}
\hspace{1cm} \text{largest} \leftarrow r
\item \textbf{end if}
\item \textbf{if} \( \text{largest} \neq i \) \textbf{then}
\hspace{1cm} \text{swap } A[i] \leftrightarrow A[\text{largest}]
\hspace{1cm} \text{MaxHeapify}(A, \text{largest})
\item \textbf{end if}
\end{itemize}
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)**

```
    16
   /  
  14   10
 /     /  
4      7   9
 |     /  
2     8   3
```

```
MaxHeapify Example

MaxHeapify(A, 4)
MaxHeapify Runtime

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

**BuildMaxHeap(A)**

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

- But this calls MaxHeapify on the leaves as well as internal nodes of the tree.
- The leaves of a heap are indexed by \([n/2] + 1\) through \(n\)
**BuildMaxHeap**

\[
\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1 \text{ do}
\]
\[\text{MaxHeapify}(A, i)\]
\[\text{end for}\]
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 \texttt{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 BuildMaxHeap

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

$\lfloor \log n \rfloor$. 
Runtime of **BuildMaxHeap**

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 $h + 1$. 
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 `MaxHeapify` on a node of height $h$ is $O(h)$. 
How many nodes can a heap of size $n$ have with height $h$?

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

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

The runtime of $\text{MaxHeapify}$ on a node of height $h$ is $O(h)$.

Thus $\text{BuildMaxHeap}$ takes:

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

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \cdot (1/2)^h = \frac{1/2}{(1 - 1/2)^2} = 2$$
Runtime of 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 MaxHeapify on a node of height $h$ is $O(h)$.
- Thus BuildMaxHeap takes:

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

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

\textbf{BuildMaxHeap}(A)

\begin{verbatim}
for \( i \leftarrow \lfloor n/2 \rfloor \) \textbf{downto} 1 do
    \texttt{MaxHeapify}(A,i)
end for
\end{verbatim}

- **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 \texttt{BuildMaxHeap}

\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 \texttt{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)**

\[
\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1 \text{ do} \\
\text{MaxHeapify(A,i)} \\
\text{end for}
\]

- **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**

**HEAPINCREASEKEY**(*A, i, key*)

1. \( A[i] \leftarrow key \)
2. **while** \( i > 1 \) **and** \( A[\text{Parent}(i)] < A[i] \) **do**
   - swap \( A[i] \leftrightarrow A[\text{PARENT}(i)] \)
   - \( i \leftarrow \text{PARENT}(i) \)
3. **end while**

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

MaxHeapInsert($A$, $key$)

$$size(A) \leftarrow size(A) + 1$$
$$A[size(A)] \leftarrow -\infty$$
HeapIncreaseKey($A$, $size(A)$, $key$)

- $O(\log n)$ - HeapIncreaseKey($A$, $i$, $key$)
Heap Extract Max

**HEAPEXTRACTMAX**(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 \( \text{MaxHeapify}(A, 1) \)
We can use a Heap to sort an array.

Turn the array into a heap using \texttt{BuildMaxHeap}

Position the \texttt{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 \texttt{BuildMaxHeap}
- Position the Maximum element \(n\) times to construct a sorted array.

\begin{algorithm}
\textbf{HeapSort}(A)
\begin{algorithmic}
\State \textbf{BuildMaxHeap}(A)
\For{\(i \leftarrow \text{size}(A)\) \textbf{downto} 2}
\State swap \(A[1] \leftrightarrow A[\text{size}(A)]\)
\State \(\text{size}(A) \leftarrow \text{size}(A) - 1\)
\State \textbf{MaxHeapify}(A, 1)
\EndFor
\end{algorithmic}
\end{algorithm}
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)**

```plaintext
BuildMaxHeap(A)
for $i \leftarrow \text{size}(A)$ downto 2 do
    swap $A[1] \leftrightarrow A[\text{size}(A)]$
    $\text{size}(A) \leftarrow \text{size}(A) - 1$
    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 (10/7)
  - Balanced Binary Search Trees
- For Next Class
  - Read 13.1, 13.2, 13.3, 13.4