Last Time

- Binary Search Trees
Today

- Heaps
- Maximum (and Minimum)
- Mean
- Median
Heaps

- 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

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

```
16
/   \
14   10
/ \
8   7 / \
2   4 1 9 3
```
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
Heap Example

\[
\begin{align*}
A[i] &= 16 & 14 & 10 & 8 & 7 & 9 & 3 & 2 & 4 & 1 \\
i &= 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{align*}
\]
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  ∅  1  2  3  4  5  6  7  8  9  10
The **Max heap property** allows compact representation of a heap as an array.

- `Parent(i) = ⌊i/2⌋`
- `Left(i) = 2i`
- `Right(i) = 2i + 1`
$2^{d-1}$ is the maximum number of nodes in a binary tree at a given depth $d$. 
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. $2^0 = 1$. 
- $2^{d-1}$ 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.
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.
  
  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 $2 \times 2^{d-1} = 2^{(d+1)-1} (= 2^d)$
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 $2^{(d+1)-1} = 2^d$
- $2^d + 2^d - 1 = 2 \times 2^d - 1 = 2^{d+1} - 1$
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 \( \text{LEFT}(i+1) \)
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.
The maximum value is always at the root of a max-heap.

\[ \text{MAXIMUM}(A) = \Theta(1) \]
Max Heapify

MaxHeapify(A, i)

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

MaxHeapify(A,1)
MaxHeapify Example

MaxHeapify(A, 2)
MaxHeapify Example

MaxHeapify(A, 4)

```
2  8  1
4   7   9  3
14  10
16
```
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: $T(n) \leq T(2n/3) + \Theta(1) = O(log n)$
Build Max Heap

\textbf{BuildMaxHeap}(A)

\begin{itemize}
    \item \textbf{for} \(i \leftarrow n\ \text{downto} \ 1\ \text{do}\)
    \item \textbf{MaxHeapify}(A,i)
    \item \textbf{end for}
\end{itemize}

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

for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do 
  MaxHeapify(A,i)
end for
We make \( n/2 \) 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 $h = 2$. 
What is the height of an $n$ element heap?
Runtime of BuildMaxHeap

- What is the height of an $n$ element heap?
- $\lfloor \log n \rfloor$. 
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 $h$?

\[
\left\lceil \frac{n}{2^{h+1}} \rightceil
\]

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 $h$?

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

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 `MaxHeapify` on a node of height $h$ is $O(h)$. 
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 $h + 1$.

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

Thus $\text{BuildMaxHeap}$ takes:

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

$$
\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = \frac{1/2}{(1 - 1/2)^2} = 2
$$
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 \texttt{BuildMaxHeap}

\begin{algorithm}
\textbf{BuildMaxHeap}(A)
\begin{align*}
\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1 & \text{ do} \\
\phantom{\text{for }} \text{MaxHeapify}(A,i) \\
\text{end for}
\end{align*}
\end{algorithm}

- **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 BuildMaxHeap**

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

\[
\begin{align*}
&\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1 \text{ do} \\
&\hspace{1cm} \text{MaxHeapify}(A, i) \\
&\text{end for}
\end{align*}
\]

- **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.

- **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 \( k > i \).
Correctness of \texttt{BuildMaxHeap}

\textbf{BuildMaxHeap}(A)

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

- \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.

- \textbf{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**($A, i, key$)

$A[i] \leftarrow key$

while $i > 1$ and $A[\text{Parent}(i)] < A[i]$ do

swap $A[i] \leftrightarrow A[\text{PARENT}(i)]$

$i \leftarrow \text{PARENT}(i)$

end while

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

**MaxHeapInsert**($A$, $key$)

```
size(A) ← size(A) + 1
A[size(A)] ← −∞
HeapIncreaseKey(A, size(A), key)
```

- $O(\log n)$ - **HeapIncreaseKey**($A$, $i$, $key$)
**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 \(\text{MaxHeapify}(A, 1)\)
- 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.
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
```
Heap Sort

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

\textbf{HeapSort}(A)

\begin{verbatim}
\textbf{BuildMaxHeap}(A)
\textbf{for} i \leftarrow \text{size}(A) \textbf{downto} 2 \textbf{do}
\hspace{1em} \text{swap} A[1] \leftrightarrow A[\text{size}(A)]
\hspace{1em} \text{size}(A) \leftarrow \text{size}(A) - 1
\hspace{1em} \textbf{MaxHeapify}(A, 1)
\textbf{end for}
\end{verbatim}

- \textbf{BuildMaxHeap}(A) = \(O(n)\)
- \textbf{MaxHeapify}(A) = \(O(\log n)\) – called \(n\) times.
- \textbf{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