08-0: **Priority Queue ADT**  
Operations  
- Add an element / key pair  
- Return (and remove) element with smallest key  

Keys are “priorities”, with smaller keys having a “better” priority

08-1: **Priority Queue ADT**  
Operations  
- Add an element / key pair  
- Return (and remove) element with smallest key

Implementation:  
- Sorted Array  
  - Add Element  
  - Remove Smallest Key

08-2: **Priority Queue ADT**  
Operations  
- Add an element / key pair  
- Return (and remove) element with smallest key

Implementation:  
- Sorted Array  
  - Add Element $O(n)$  
  - Remove Smallest Key $O(1)$ (using circular array)

08-3: **Priority Queue ADT**  
Operations  
- Add an element / key pair  
- Return (and remove) element with smallest key

Implementation:  
- Binary Search Tree  
  - Add Element  
  - Remove Smallest Key

08-4: **Priority Queue ADT**  
Operations  
- Add an element / key pair  
- Return (and remove) element with smallest key
Implementation:

- Binary Search Tree
  - Add Element \( O(\lg n) \)
  - Remove Smallest Key \( O(\lg n) \)

If the tree is balanced

08-5: **Priority Queue ADT**

Operations

- Add an element / key pair
- Return (and remove) element with smallest key

Implementation:

- Binary Search Tree
  - Add Element \( O(n) \)
  - Remove Smallest Key \( O(n) \)

Computer Scientists are Pessimists  

(Murphy was right)

08-6: **Heap Definition**

- Complete Binary Tree
- Heap Property
  - For every subtree in a tree, each value in the subtree is \( \geq \) value stored at the root of the subtree

08-7: **Heap Examples**

```
    1
   / \
  2   4
 / \   /
7  3 14 15
/ \   /
9  8  5  4
```

Valid Heap

08-8: **Heap Examples**
08-9: **Heap Insert**

- What is the only place we can insert an element in a heap, and maintain the complete binary tree property?

08-10: **Heap Insert**

- What is the only place we can insert an element in a heap, and maintain the complete binary tree property?
  - “End” of the tree – as a child of the shallowest leaf that is farthest to the left
  - Will the resulting tree still be a heap?

08-11: **Heap Insert**

- What is the only place we can insert an element in a heap, and maintain the complete binary tree property?
  - “End” of the tree – as a child of the shallowest leaf that is farthest to the left
  - Inserting an element at the “end” of the heap may break the heap property
  - Swap the value up the tree (examples)

08-12: **Heap Insert**

- Running time for Insert?

08-13: **Heap Insert**

- Running time for Insert?
  - Place element at end of tree: \(O(1)\) (We’ll see a clever way to find the “end” of the tree in a bit)
  - Swap element up the tree: \(O(\text{height of tree})\) (Worst case, swap all the way up to the root)
    - Height of a Complete Binary Tree with \(n\) nodes?

08-14: **Heap Insert**

- Running time for Insert?
  - Place element at end of tree: \(O(1)\) (We’ll see a clever way to find the “end” of the tree in a bit)
  - Swap element up the tree: \(O(\text{height of tree})\) (Worst case, swap all the way up to the root)
• Height of a Complete Binary Tree with \( n \) nodes = \( \Theta(\lg n) \)

• Total running time: \( \Theta(\lg n) \) in the worst case

08-15: **Heap Remove Smallest**

• Finding the smallest element is easy – at the root of the tree
• Removing the Root of the heap is hard
• What element is easy to remove? How could this help us?

08-16: **Heap Remove Smallest**

• Finding the smallest element is easy – at the root of the tree
• Removing the Root of the heap is hard
• Removing the element at the “end” of the heap is easy
  • Copy last element of heap into root
  • Remove the last element
  • Problem?

08-17: **Heap Remove Smallest**

• Finding the smallest element is easy – at the root of the tree
• Removing the Root of the heap is hard
• Removing the element at the “end” of the heap is easy
  • Copy last element of heap into root
  • Remove the last element
  • May break the heap property

08-18: **Heap Remove Smallest**

• Finding the smallest element is easy – at the root of the tree
• Removing the Root of the heap is hard
• Removing the element at the “end” of the heap is easy
  • Copy last element of heap into root
  • Remove the last element
  • Push the root down, until heap property is satisfied

08-19: **Heap Remove Smallest**

• Running time for remove smallest?

08-20: **Heap Remove Smallest**

• Running time for remove smallest?
• Copy last element into root, remove last element: $O(1)$, given a $O(1)$ time method to find the last element
• Push the root down: $O(\text{height of the tree})$ (Worst case, push element all the way down)
  • As before, Complete Binary Tree with $n$ elements has height $\Theta(\lg n)$
• Total time: $\Theta(\lg n)$ in the worst case

08-21: Representing Heaps

• Represent heaps using pointers, much like BSTs
  • Need to add parent pointers for insert to work correctly
  • Need to maintain a pointer to the location to insert the next element (this could be hard to update & maintain)
  • Space needed to store pointers – 3 per node – could be greater than the space need to store the data in the heap!
  • Memory allocation and deallocation is slow

• There is a better way!

08-22: Representing Heaps

A Complete Binary Tree can be stored in an array:

```
   1
  / \
 2   14
 /   /\n5   3  16 15
 / \   / \\
7  6  8  9
```

08-23: CBTs as Arrays

• The root is stored at index 1
• For the node stored at index $i$:
  • Left child is stored at index $2 \times i$
  • Right child is stored at index $2 \times i + 1$
  • Parent is stored at index $\lfloor i/2 \rfloor$

08-24: CBTs as Arrays

Finding the parent of a node

```cpp
int parent(int n) {
    return (n / 2);
}
```
Finding the left child of a node

```c
int leftchild(int n) {
    return 2 * n;
}
```

Finding the right child of a node

```c
int rightchild(int n) {
    return 2 * n + 1;
}
```

08-25: **Building a Heap**
Build a heap out of \( n \) elements

08-26: **Building a Heap**
Build a heap out of \( n \) elements

- Start with an empty heap
- Do \( n \) insertions into the heap

```java
MinHeap H = new MinHeap();
for(i=0 < i<A.size(); i++)
    H.insert(A[i]);
```

Running time?

08-27: **Building a Heap**
Build a heap out of \( n \) elements

- Start with an empty heap
- Do \( n \) insertions into the heap

```java
MinHeap H = new MinHeap();
for(i=0 < i<A.size(); i++)
    H.insert(A[i]);
```

Running time? \( O(n \log n) \) – is this bound tight?

08-28: **Building a Heap**
Total time: \( c_1 + \sum_{i=1}^{n} c_2 \log i \)

\[
c_1 + \sum_{i=1}^{n} c_2 \log i \geq \sum_{i=n/2}^{n} c_2 \log i \\
\geq \sum_{i=n/2}^{n} c_2 \log(n/2) \\
= (n/2)c_2 \log(n/2) \\
= (n/2)c_2((\log n) - 1) \\
\in \Omega(n \log n)
\]

Running Time: \( \Theta(n \log n) \)

08-29: **Building a Heap**
Build a heap from the bottom up
• Place elements into a heap array
• Each leaf is a legal heap
• First potential problem is at location \([i/2]\)

08-30: **Building a Heap**
Build a heap from the bottom up

• Place elements into a heap array
• Each leaf is a legal heap
• First potential problem is at location \([i/2]\)

\[
\text{for}(i = n/2; i >= 0; i--) \\
\text{pushdown}(i);
\]

08-31: **Building a Heap**
How many swaps, worst case? If every \texttt{pushdown} has to swap all the way to a leaf:
- \(\frac{n}{4}\) elements \(1\) swap
- \(\frac{n}{8}\) elements \(2\) swaps
- \(\frac{n}{16}\) elements \(3\) swaps
- \(\frac{n}{32}\) elements \(4\) swaps

\[
\text{Total # of swaps:} \\
\frac{n}{4} + 2 \frac{n}{8} + 3 \frac{n}{16} + 4 \frac{n}{32} + \ldots + (\log n) \frac{n}{n}
\]

08-32: **Decreasing a Key**
• Given a specific element in a heap, how can we decrease the key of that element, and maintain the heap property?
  • Examples

08-33: **Decreasing a Key**
• Given a specific element in a heap, how can we decrease the key of that element, and maintain the heap property?
  • Examples
  • Push the element up the tree, just like after an insert
  • Examples

08-34: **Decreasing a Key**
• Decrease the key of a specific element in a heap:
  • Decrease the key value
  • Push the element up the tree, just like after an insert
  • Time required?

08-35: **Decreasing a Key**
• Decrease the key of a specific element in a heap:
• Decrease the key value
• Push the element up the tree, just like after an insert
• Time required: $\Theta(lg n)$, in the worst case.

08-36: **Removing an Element**

• Given a specific element in a heap, how can we remove that element, and maintain the heap property?
  • Examples

08-37: **Removing an Element**

• Given a specific element in a heap, how can we remove that element, and maintain the heap property?
  • Examples
  • Decrease key to a value $< root$
  • Remove smallest element

08-38: **Removing an Element**

• Given a specific element in a heap, how can we remove that element, and maintain the heap property?
  • Examples
  • Decrease key to a value $< root$. Time $\Theta(lg n)$ worst case
  • Remove smallest element. Time $\Theta(lg n)$ worst case

08-39: **Java Specifics**

• When inserting an element, push value up until it reaches the root, or it’s $\geq$ its parent
  • Our while statement will have two tests
• We can insert a *sentinel* value at index 0, guaranteed to be $\leq$ any element in the heap
  • Now our while loop only requires a single test