Primary operations they needed:
- Insert
- Find item with highest priority
  - e.g., findMin() or findMax()
- Remove an item with highest priority
  - e.g., removeMin() or removeMax()

How do we implement these operations?
- Simplest approach: arrays
Unsorted array PQ

- **Insert:**
  - Add new item to the end of the array
  - $\Theta(1)$
- **Find:**
  - Search for the highest priority item (e.g., min or max)
  - $\Theta(n)$
- **Remove:**
  - Search for the highest priority item and delete
  - $\Theta(n)$

- Runtime for use in Huffman tree generation?
Sorted array PQ

- **Insert:**
  - Add new item in appropriate sorted order
  - $\Theta(n)$
- **Find:**
  - Return the item at the end of the array
  - $\Theta(1)$
- **Remove:**
  - Return and delete the item at the end of the array
  - $\Theta(1)$

- Runtime for use in Huffman tree generation?
So what other options do we have?

- What about a binary search tree?
  - Insert
    - Average case of $\Theta(lg\ n)$, but worst case of $\Theta(n)$
  - Find
    - Average case of $\Theta(lg\ n)$, but worst case of $\Theta(n)$
  - Remove
    - Average case of $\Theta(lg\ n)$, but worst case of $\Theta(n)$

- OK, so in the average case, all operations are $\Theta(lg\ n)$
  - No constant time operations
  - Worst case is $\Theta(n)$ for all operations
Is a BST overkill?

- Our find and remove operations only need the highest priority item, not to find/remove any item
  - Can we take advantage of this to improve our runtime?
    - Yes!
A heap is complete binary tree such that for each node $T$ in the tree:

- $T$.item is of a higher priority than $T$.right_child.item
- $T$.item is of a higher priority than $T$.left_child.item

It does not matter how $T$.left_child.item relates to $T$.right_child.item

- This is a relaxation of the approach needed by a BST
Heap PQ runtimes

- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained
Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property
Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3
Tricky to delete root…
  ○ So let’s simply overwrite the root with the item from the last leaf and delete the last leaf
    ■ But then the root is violating the heap property…
      ● So we push the root down the tree until it is supporting the heap property
Min heap removal

NO!
Heap runtimes

- Find
  - $\Theta(1)$

- Insert and remove
  - Height of a complete binary tree is $\lg n$
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are $\Theta(\lg n)$
Heap implementation

- Simply implement tree nodes like for BST
  - This requires overhead for dynamic node allocation
  - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
  - We can easily represent a complete binary tree using an array
Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index $i$
  - $\text{parent}(i) = \lfloor (i - 1) / 2 \rfloor$
  - $\text{left}_\text{child}(i) = 2i + 1$
  - $\text{right}_\text{child}(i) = 2i + 2$

For arrays indexed from 0
Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- “Remove” the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat
Heap sort analysis

- **Runtime:**
  - Worst case:
    - $n \log n$
- **In-place?**
  - Yes
- **Stable?**
  - No
What if we want to update an Object?

- What is the runtime to find an arbitrary item in a heap?
  - $\Theta(n)$
  - Hence, updating an item in the heap is $\Theta(n)$

- Can we improve of this?
  - Back the PQ with something other than a heap?
  - Develop a clever workaround?
Indirection

- Maintain a second data structure that maps item IDs to each item’s current position in the heap
- This creates an *indexable* PQ
Let’s say I’m shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.

Keep objects of the following type in the heap:

```java
class CardPrice implements Comparable<CardPrice>{
    public String store;
    public double price;
    public CardPrice(String s, double p) { … }
    public int compareTo(CardPrice o) {
        if (price < o.price) { return -1; }
        else if (price > o.price) { return 1; }
        else { return 0; }
    }
}
```
Indirection example

- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

- Update price for NE: 340.00
- Update price for NCIX: 345.00
- Update price for BB: 200.00