A Data structure that is organized based on what is important “now”

Data Structures so far:
- Arrays and ArrayLists (Java only)
- Linked Lists – singly, doubly, circular, multi
- Stacks and Queues
- Binary Search Trees
- Hash tables
- Now to priority queues...

Grocery Store Example
- Suppose there are 5 people in line and each one requires a service time (in mins) 10, 4, 5, 6, 12
- What is the average service time per customer?

- Suppose we decided to service smaller times first.
- What is the average time now?

Hence we can make things efficient by dynamically reorganizing things

What if we keep things in a random array and always serve the next min/max?

We need a data structure that can support that

What is we use a sorted array?
Priority Queue
(The Binary Heap)

Implementing a PQ:
• How do we implement a PQ?
  • Need a data structure that can support the following operations well
    • insertMin/Max, findMin/max, removeMin/Max
  • Possible data structures
    • Unordered list?
      • What is the insertion complexity? Removal complexity?
    • Sorted List?
      • What about a binary search tree?

Complete Binary Trees
(implementing PQ’s)

Recall - Complete binary trees

Representing complete binary trees
• Can be represented using
  • Linked structures? (hard)
  • Arrays! (easy)
Representing complete binary trees

- Arrays
  - Parent at position \( i \)
  - Children at \( 2i \) and \( 2i + 1 \).

A heap can be represented using a complete binary tree

Binary Heap properties

Must satisfy two properties

1. Structure property
   - Complete binary tree
   - Hence: efficient compact representation

2. Heap order property
   - Parent keys less than children keys
   - Hence: rapid insert, findMin, and deleteMin
   - \( O(\log N) \) for insert and deleteMin
   - \( O(1) \) for findMin

Priority Queue operations using a binary heap

- How to code a PQ operations using a Heap?
  - findMin() –
    - The code
      ```java
      public boolean isEmpty() {
          return size == 0;
      }
      public Comparable findMin() {
          if(isEmpty()) return null;
          return heap[1];
      }
      ```
    - FindMin() does not change the tree
    - Trivially preserves the invariant
**Insert Operation**
- Insert(x) –
  - Put the new element into next leaf position (to maintain complete tree property) and then swap it up as long as it’s <= its parent
- More formally...

**Insert (Comparable x)**
- Process
  1. Create a “hole” at the next tree cell for x.
     ```java
     heap[size+1]
     ```
     This preserves the completeness of the tree assuming it was complete to begin with.
  2. Percolate the hole up the tree until the heap order property is satisfied.
     This assures the heap order property is satisfied assuming it held at the outset.

**Percolation up**
- Bubble the hole up the tree until the heap order property (HOP) is satisfied.

```
public void insert(Comparable x) throws Overflow {
    if(isFull()) throw new Overflow();
    for(int i = ++size;
        i>1 && x.compareTo(heap[i/2])<0;
        i/=2)
        (heap[i] = heap[i/2];)
    heap[i] = x;
}
```
Examples