We will be developing data structures that support queries and updates.

Example where a data structure will help:
- Event driven system
  - Event: (start of call i, time when i starts)
  - Simulator: pick next event, process it, maybe update event queue.
- How to maintain events?
  - Need support for fast:
    - insert new event
    - pick “next event”, i.e. event with smallest time key.

Several possible approaches

- Keep all events in a list.
  - Easy to insert - O(1)
  - Hard to extract - \( \Omega(n) \)
- Sorted list:
  - Easy to extract - O(1)
  - Hard to insert - \( \Omega(n) \)
  - We would like something like:
    - insert \( O(\log n) \)
    - extract \( O(\log n) \)

Tradeoff

Heaps

- Nearly complete binary tree with:
  - Max heap property: \( A[parent(i)] \geq A[i] \)
  - Claim: max is at the root (by induction on the size of the heap)

Pointers are not the most efficient solution.
Instead, \( parent(i) \) is stored in each element.
Example: parent of the 5th element is at 2.

Fixing a broken heap

- Assume problem is only at the root:
  - Now the problem “moved” down, into right tree.
  - Recurse in this tree, exchanging 5 and 8, its largest child.

Correctness of fixing the heap

- \( b \) is larger than \( c \), and \( a \), thus the only problem can be between \( a \) and one of its children.
- Formal proof - by induction on the height of \( a \).
- This procedure will be called \( \text{Heapify}(A, i, n) \).
- Makes subtree rooted at \( A(i) \) into a heap.
- Time: \( O(\log n) \). (Why?)
Inserting new element

- Similar to Heapify:
  - Insert new element
  - Place new element in last
  - While parent(i) != null
    - If A(i) < A(parent(i)) return
    - Else exchange A(i), A(parent(i))
    - i = parent(i)
  - end
- Example:
  - Propagate up, O(lg n). Correctness ??