Priority Queues

Stores prioritized elements
- No notion of storing at particular position
Returns elements in priority order
- Order determined by key

What's so different?

Stacks and Queues
- Removal order determined by order of inserting
Sequences
- User chooses exact placement when inserting and explicitly chooses removal order
Priority Queue
- Order determined by key
- Key may be part of element data or separate

What's it good for?

Order of returned elements is not FIFO or LIFO (as in queue or stack)
Random access not necessary (as in sequence) or desirable
Examples
- Plane landings managed by air traffic control
- Processes scheduled by CPU
- College admissions process for students
  — What are some of the criteria?

College Admissions Key

Student submits:
- Personal data (geography, is parent alum?, activities?)
- Transcript
- Essays
- Standardized test scores
- Recommendations
Admissions agent:
- Each datum converted to number
- Formula converts to single numeric key

Student selection process

Simple scheme
- Collect applications until due date
- Sort by keys
- Take top $k$ students
More realistic
- Prioritize applications as they come in
- Accept some top students ASAP
- Maybe even change data/key as you go
Priority Queue ADT
insertItem(k, e): insert element e with key k
extractMin(): return element with minimum key and remove from queue
minElement(): return (look at) min element
minKey(): return minimum key
size(): return number of elements
isEmpty(): size == 0?

Keys, Comparitors and Total Orders
Key type needs comparison operator (returns boolean) with following properties:
• Reflexive: k ≤ k
• Antisymmetric: (k1 ≤ k2) && (k2 ≤ k1) → k1 = k2
• Transitive: ((k1 ≤ k2) && (k2 ≤ k3) → k1 ≤ k3
These properties guarantee consistent, total ordering

Abstracting Comparitors
Allows for different types of comparison
• e.g. Numeric vs. lexicographic (for strings)
Several approaches possible
• Build PQ object to know about specific key type and comparison
• Build key object to know about comparison
• Build separate comparator object for each type of comparison
Book argues for #3, but I also recommend #2

Implementing PQ with Unsorted Sequence
Each call to insertItem(k, e) uses insertLast( ) to store in Sequence
• O(1) time
Each call to extractMin( ) traverses the entire sequence to find the minimum, then removes element
• O(n) worst case

Implementing PQ with Sorted Sequence
Each call to insertItem(k, e) traverses sorted sequence to find correct position, then does insert
• O(n) worst case
Each call to extractMin( ) does removeFirst( )
• O(1) time

Sorting Using a PQ
Elements begin in arbitrary order in a sequence
Move elements from sequence into PQ
Extract elements from PQ and reinsert into sequence in priority order
Analysis depends on implementation choices
Analyzing Queue Efficiency for Sorting

N insertElement( ) operations followed by N extractMin( ) operations

Selection Sort

PQ sorting using unsorted sequence

Insert all n items in input order

Extract by selecting min item n times

Insertion Sort

PQ sorting using sorted sequence

Sequentially insert items into sequence in sorted order

Extract items easily from sorted sequence

Sort Analysis

foreach element, Ei, in S O(n)
PQ.insert(Ei) \( \sum_{i=0}^{n-1} O(i) \)
while !PQ.empty() O(n)
PQ.extractMin() \( \sum_{i=0}^{n-1} O(i) \)

\( O(n) + \sum_{i=0}^{n-1} O(i) + \sum_{i=0}^{n-1} O(i) = O(n) + O(n) + O(n^2) = O(n^2) \)

Heap

Binary tree-based data structure

- Complete in the sense that it fills up levels as completely as possible
- Height of tree is \( O(\log n) \)

Stores elements with keys

All nodes satisfy the heap property:

- The key value at a node is less than or equal to the key value of the node’s children

Allows insertItem( ) and extractMin( ) in \( O(\log n) \) time

Heap Example

- Node values are keys

- The heap property is satisfied
PQ Quiz Show!

Heap, or Not A Heap?

(no paper necessary)

Heap, or Not a Heap?

Inserting into Heap

Create new node as “last” element
Insert key/element into new node
Bubble node upward until heap property is satisfied

while (!isRoot(node) &&
   (node.key < node.parent.key))
   swap(node, parent)

(just pseudocode - can’t do it exactly like this in Java)
Heap Insert Example

10
15
19
17
31
32
22
21
35
20
30
21
19
17
31
32
22
35
30
20
25

Heap Insert Analysis

New node always inserted at lowest level
Node bubbles upward
  * up to root in worst case
  * path length to root is $O(\log n)$
Total time for insert is $O(\log n)$

Extracting from Heap

Copy element from root node
Copy element/key from last node to root node
Delete last node
Bubble root node downward until heap property satisfied

while (!isExternal(node) &&
  (node.key > node.smallestChild.key))
  swap(node, node.smallestChild)
Heap Extract Example

Copy Extracted Element

Move last node to root

Bubble Downward

Bubble Downward

All done
Heap Extract Analysis

Again, each swap takes constant time
Maximum swaps is path length from root to leaf
→ Total work is \( \log n \times O(1) = O(n) \)

Sort Analysis

foreach element, \( E_i \), in \( S \)
\( O(n) \)

while !PQ.empty()
\( O(n) \)

\( O(n) + 2 \sum_{i=1}^{n} O(\log i) < O(n) + 2 \sum_{i=1}^{n} O(\log n) \)
\( = O(n) + 2n \times O(\log n) = O(n \log n) \)

(showing \( \Theta(n \log n) \) is a bit harder)

In-class Exercise

What does the heap look like after the following sequence of insertions:

5 30 2 15 7 45 20 6 18