CS 151
Priority Queues
Announcements

Prelab 7 is due Monday morning. Please hand in the structural induction worksheet (posted by end of today) separately from the rest of your prelab.

Lab 6 due on Sunday night.

CS department winter term organizational meeting:
  Tuesday Nov 6, 4:30pm in King 237
Motivation

A queue is an ADT (abstract data type) that stores items first-in first-out.

- It assumes that the first item in has the highest priority for removal

When is this not the case?

- The president enters a bank. You’d probably serve him first.
- Planes at an airport. Air Force One would probably jump the queue.
- Immigration lines. Citizens and folks with special cards get priority.

We’d like a data type that allows us to repeatedly remove the highest priority item, where priority doesn’t have to be the same as “age in the queue”.

A queue no longer works because it’s not general enough.

A BST would work, but it has more overhead than we need, because it allows us access to arbitrary items, whereas we’ll only need access to the highest priority one.

Our goal will be to get $O(1)$ access to the max priority element.
A priority queue stores a collection of comparable elements such that the highest priority element is “at the front”. Operations are

- `insert(element)` aka `offer` — add element to the queue
- `removeMin()` aka `poll` — remove the highest priority element
- `findMin()` aka `peek` — return but do not remove the highest priority element
- `size()`
- `isEmpty()`
- `traverse()` / `iterate()`

We want all of these operations to be $O(1)$ except for insert and `removeMin`, which we want in $O(\log n)$ time.
Implementing PQ’s

A priority queue stores a collection of comparable elements such that the highest priority element is “at the front”. Operations are

- `insert(element)` aka offer — add element to the queue
- `removeMin()` aka poll — remove the highest priority element
- `findMin()` aka peek — return but do not remove the highest priority element
- `size()`
- `isEmpty()`
- `traverse()` / `iterate()`

idea #1: implement PQ using an unordered array as backing storage

Then what would the running time of the operations be?

- `insert(element)` — $O(1)$ (just add it to the end)
- `removeMin()` — $O(n)$ (takes $O(n)$ time to find min in unordered list)
- `findMin()` — $O(n)$ (takes $O(n)$ time to find min in unordered list)
- `size()` — $O(1)$
- `isEmpty()` — $O(1)$
- `traverse()` / `iterate()` — $O(n)$ if don’t care about order, longer if you do.
Implementing PQ’s

A priority queue stores a collection of comparable elements such that the highest priority element is “at the front”. Operations are

- `insert( element )` aka offer — add element to the queue
- `removeMin()` aka poll — remove the highest priority element
- `findMin()` aka peek — return but do not remove the highest priority element
- `size()`
- `isEmpty()`
- `traverse()` / `iterate()`

Idea #2: implement PQ using an ordered array / LL as backing storage

Then what would the running time of the operations be?

- `insert( element )` — $O(n)$ (to find the correct location for the element)
- `removeMin()` — $O(n)$ if an array due to shifting, $O(1)$ if LL
- `findMin()` — $O(1)$ (it’s the front element)
- `size()` — $O(1)$
- `isEmpty()` — $O(1)$
- `traverse()` / `iterate()` — $O(n)$
Implementing PQ’s

A priority queue stores a collection of comparable elements such that the highest priority element is “at the front”. Operations are

- `insert( element )` aka offer — add element to the queue
- `removeMin()` aka poll — remove the highest priority element
- `findMin()` aka peek — return but do not remove the highest priority element
- `size()`
- `isEmpty()`
- `traverse()` / `iterate()`

idea #3: implement PQ using a balanced binary search tree as backing storage

Then what would the running time of the operations be?

- `insert( element )` — $O(\log n)$
- `removeMin()` — $O(\log n)$
- `findMin()` — $O(\log n)$
- `size()` — $O(1)$
- `isEmpty()` — $O(1)$
- `traverse()` / `iterate()` — $O(n)$
A binary heap is an array-based data structure that combines a queue and a binary search tree. A binary heap is

- a complete binary tree (a binary tree where each level is completely filled except for possibly the last, which is filled from left-to-right.)
- with the heap order property:
  - for all nodes n, the value/key of n is $\leq$ value/key of its children

Examples:
A **binary heap** is an array-based data structure that combines a queue and a binary search tree. A binary heap is

- a complete binary tree (a binary tree where each level is completely filled except for possibly the last, which is filled from left-to-right.)
- with the **heap order property**:
  - for all nodes n, the value/key of n is $\leq$ value/key of its children

The nodes of a heap are stored in an array in level-order (as produced by bfs.)
Place the root node at index 1 (NOT index 0 this time!)

Examples:
Binary Heap

Since a heap is a complete binary tree stored in an array from index 1...

If a node is at position i in the array,
- its left child (if it has one), is at index $2i$
- its right child (if it has one) is at index $2i+1$
- its parent (if it has one) is at index $\lfloor i/2 \rfloor$

So it is easy to access these relatives in $O(1)$ time.
Priority Queue Insertion with Heap

To insert an element into a PQ’s heap, we add it as a leaf in the next open spot (which happens to be the “size+1” index.) Also add the element at position 0 (this will be explained momentarily.)

Then, we percolate / bubble the node up into its correct position.

Ex:

\[ \text{percolateUp( index )} \] – move the element at index index up into the correct spot

- compare data[index] to data[ parent(index) ]
- if parent is <= current element, we have heap order property, return
- else, parent is greater than element:
  - swap the parent and current element
  - set index = parent(index)
  - percolateUp( index )

Note that sooner or later, parent will be <= current, because put current element at position 0. (mystery solved!)

RT: if percolate up to the top, get \( O(\log n) \). But on average takes 2.6 comparisons to find a spot, i.e. moves up 1.6 levels so it’s usually \( O(1) \)!
To delete the min element from a PQ’s heap, we need to remove the root node then reorder the tree so that the last leaf (but not its data) is removed.

idea: replace the minimum element (the root) with the last child of the tree. Then percolate the new root’s value down the tree into its correct position.

Ex:
percolateDown(index) – move the elt at index index down to the correct spot

• loop
  • if index is a leaf, return (it’s smaller than its non-existent children)
  • else, let minChild = index of smaller of index’s children (or, child)
    • if data[minChild] < data[index] then we’re out of heap order
      • swap the two entries
      • set index = minChild
    • else, return (we’re done.)

RT: if percolate down to the bottom, get $O(\log n)$. Note that percolate(index) is actually $O(\text{height of the tree rooted at index})$. 