Data Structures Toolbox

Three categories of fundamental data structures and algorithms –

- collections – List, Stack, Queue
  - implementations: array, linked list; sorted vs unsorted
- searching and lookup
  - sequential search, binary search
  - Map/Dictionary
    - implementations: array, linked list; balanced BST (AVL, 2-4)
  - Set
- sorting
  - insertion sort, selection sort, mergesort, quicksort, ...
  - PriorityQueue

Sorting for Algorithm Design

option	applications
sorting algorithm	• when all elements to sort are known at one time
PriorityQueue	• for sorting in a dynamic environment, where the elements are not all known at once • e.g. greedy algorithms such as Dijkstra's algorithm and Prim's algorithm

Pragmatics –

- can handle increasing vs decreasing order, keys vs records, and non-numerical data by abstracting a *comparator* from the sorting algorithm

ADTs – PriorityQueue

PriorityQueue	maintain removal order when there are out-of-order additions	• insert(x,p) – insert elt x with priority p • findMin() or findMax() – find elt with min/max priority • deleteMin() or deleteMax() – remove (and return) elt with min/max key note: a PQ is typically either a min-PQ or a max-PQ – it does not support both min and max operations simultaneously

Priority Queue Implementation

We need some kind of collection to hold the keys/elements in the PQ.

There are two basic collections

- array
- linked list

and two basic ways elements can be ordered within those collections

- not sorted
- sorted

Priority Queue Implementation

operation	array - unsorted	array - sorted	linked list - unsorted	linked list - sorted
find min				
insert				
remove min				

Priority Queue Implementation

operation	array - unsorted	array - sorted	linked list - unsorted	linked list - sorted
find min	**O(1)** – store index of min	**O(1)** – in slot 0	**O(1)** – store node with min	**O(1)** – at head
insert	**O(1)** – add at end	**O(n)** – binary search + shift	**O(1)** – add at head	**O(n)** – sequential search
remove min	**O(n)** – shift + update min index	**O(1)** – using circular array	**O(n)** – update min node	**O(1)** – at head

Tradeoff: fast insert or fast remove, but not both.

Priority Queue Implementation

Can we do better?

Observations.

- either insert or remove takes O(n) time
  - would be nice to reduce this!
- there is an ordering of the elements (by priority)
  - sorted order is exploited in remove min but isn't helpful for insert (binary search in array is offset by having to shift to make room)

Recall: balanced search trees

- insert/remove is O(log n)

Priority Queue Implementation

operation	array - unsorted	array - sorted	balanced BST
find min	**O(1)** – store index of min	**O(1)** – in slot 0	
insert	**O(1)** – add at end	**O(n)** – binary search + shift	
remove min	**O(n)** – shift + update min index	**O(1)** – using circular array	

Priority Queue Implementation

operation	array - unsorted	array - sorted	balanced BST
find min	**O(1)** – store index of min	**O(1)** – in slot 0	**O(1)** – store min node
insert	**O(1)** – add at end	**O(n)** – binary search + shift	**O(log n)** – update tree structure
remove min	**O(n)** – shift + update min index	**O(1)** – using circular array	**O(log n)** – update tree structure + update min node

Tradeoff: worst-case time reduced from O(n) to O(log n), but have lost O(1) insert or remove.

Priority Queue Implementation

operation	array - unsorted	array - sorted	balanced BST	hashtable
find min	**O(1)** – store index of min	**O(1)** – in slot 0	**O(1)** – store min node	
insert	**O(1)** – add at end	**O(n)** – binary search + shift	**O(log n)** – update tree structure	
remove min	**O(n)** – shift + update min index	**O(1)** – using circular array	**O(log n)** – update tree structure + update min node	

Priority Queue Implementation

operation	array - unsorted	array - sorted	balanced BST	hashtable
find min	**O(1)** – store index of min	**O(1)** – in slot 0	**O(1)** – store min node	**O(1)** – store min elt
insert	**O(1)** – add at end	**O(n)** – binary search + shift	**O(log n)** – update tree structure	**O(1)** – hashtable insert + update min
remove min	**O(n)** – shift + update min index	**O(1)** – using circular array	**O(log n)** – update tree structure + update min node	**O(N)** – hashtable remove + update min

- hashtables are good for lookup, but not for ordering

Priority Queue Implementation

Can we do better?

Observation.

- O(log n) for insert, remove min is due to updating the tree structure

operation	balanced BST
find min	**O(1)** – store min node
insert	**O(log n)** – update tree structure
remove min	**O(log n)** – update tree structure + update min node

Priority Queue Implementation

Can we do better?

Consider the essence of the problem –
- don't necessarily need a full sorted order at any one point, just the ability to get the min
- adding an element is a small incremental change
- only a specific element is removed (the min)

Strategy –
- maintain a *partial order* of elements
  - stronger than unsorted (to improve on updating min) but not as strong as sorted (to improve on insert performance)
  -

Priority Queue Implementation

How to implement?

Observations.
- balanced BST = binary tree +
  ordering constraint to aid in search +
  structural constraint to aid in efficiency

Can we do something along these lines for PQs?
- but with a weaker ordering constraint since search only needs to find the min