Adaptable Priority Queues

Entry and Priority Queue ADTs

- An entry stores a (key, value) pair
- Entry ADT methods:
  - getKey(): returns the key associated with this entry
  - getValue(): returns the value paired with the key associated with this entry

- Priority Queue ADT:
  - insert(k, x) inserts an entry with key k and value x
  - removeMin() removes and returns the entry with smallest key
  - min() returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

Example

- Online trading system where orders to purchase and sell a stock are stored in two priority queues (one for sell orders and one for buy orders) as (p,s) entries:
  - The key, p, of an order is the price
  - The value, s, for an entry is the number of shares
  - A buy order (p,s) is executed when a sell order (p′,s′) with price p′≤p is added (the execution is complete if s′≥s)
  - A sell order (p,s) is executed when a buy order (p′,s′) with price p′≥p is added (the execution is complete if s′≥s)
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Methods of the Adaptable Priority Queue ADT

- remove(e): Remove from P and return entry e.

- replaceKey(e,k): Replace with k and return the key of entry e of P; an error condition occurs if k is invalid (that is, k cannot be compared with other keys).

- replaceValue(e,x): Replace with x and return the value of entry e of P.

Example

Operation	Output	P
insert(5,*A*)	e_1	(5,*A*)
insert(3,*B*)	e_2	(3,*B*),(5,*A*)
insert(7,*C*)	e_3	(3,*B*),(5,*A*),(7,*C*)
min()	e_2	(3,*B*),(5,*A*),(7,*C*)
key(e_2)	3	(3,*B*),(5,*A*),(7,*C*)
remove(e_1)	e_1	(3,*B*),(7,*C*)
replaceKey(e_2,9)	3	(7,*C*),(9,*B*)
replaceValue(e_3,*D*)	*C*	(7,*D*),(9,*B*)
remove(e_2)	e_2	(7,*D*)

Locating Entries

❑ In order to implement the operations remove(k), replaceKey(e), and replaceValue(k), we need fast ways of locating an entry e in a priority queue.

❑ We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.

Location-Aware Entries

❑ A locator-aware entry identifies and tracks the location of its (key, value) object within a data structure

❑ Intuitive notion:
- Coat claim check
- Valet claim ticket
- Reservation number

❑ Main idea:
- Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

List Implementation

❑ A location-aware list entry is an object storing
- key
- value
- position (or rank) of the item in the list

❑ In turn, the position (or array cell) stores the entry

❑ Back pointers (or ranks) are updated during swaps
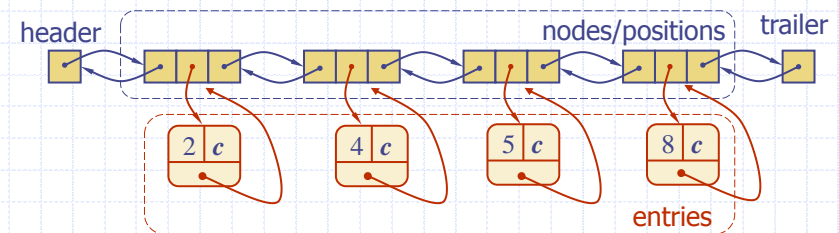
Heap Implementation

- A location-aware heap entry is an object storing
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps
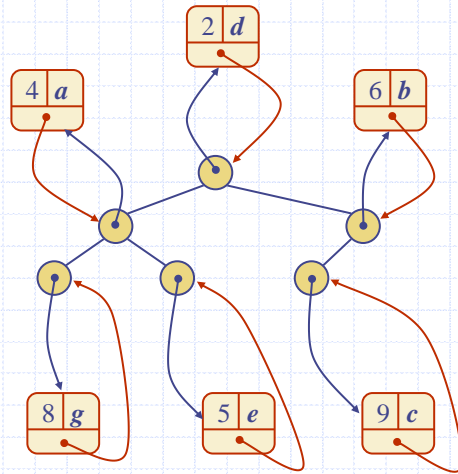
Performance

- Improved times thanks to location-aware entries are highlighted in red

Method	Unsorted List	Sorted List	Heap
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$
replaceValue	$O(1)$	$O(1)$	$O(1)$