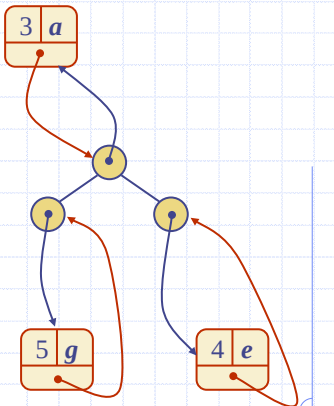
Adaptable Priority Queues



Entry and Priority Queue ADTs

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- The priority queue is associated with a comparator C, that compares two entries
- Priority Queue ADT:
  - insert(e) inserts entry e
  - removeMin() removes the entry with smallest key
  - min() returns, but does not remove, an entry with smallest key
  - size(), empty()

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'$\leq$p is added (the execution is complete if s'$\geq$s)
  - A sell order (p,s) is executed when a buy order (p',s') with price p'$\geq$p is added (the execution is complete if s'$\geq$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

- insert(e): Insert the entry e into P and return a position referring to this entry
- remove(p): Remove from P the entry referenced by position p
- replace(p, e): Replace with e the element associated with the entry referenced by p and return the position of the altered entry

Example

Operation	*Output*	*P*
insert(5,A)	p_1	(5,A)
insert(3,B)	p_2	(3,B), (5,A)
insert(7,C)	p_3	(3,B), (5,A), (7,C)
min()	p_2	(3,B), (5,A), (7,C)
p_2.key()	3	(3,B), (5,A), (7,C)
remove(p_1)	–	(3,B), (7,C)
replace(p_2,(9,D))	p_4	(7,C), (9,D)
replace(p_3,(7,E))	p_5	(7,E), (9,D)
remove(p_4)	–	(7,D)

Locating Entries

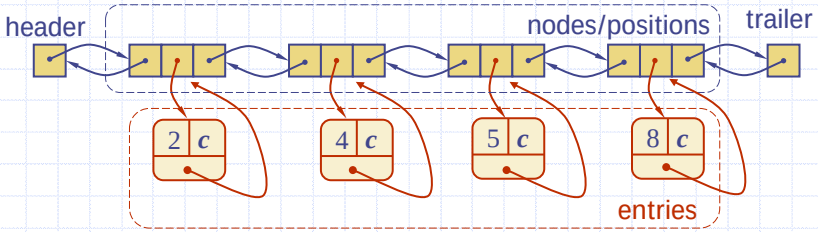
- In order to implement the operations remove(p) and replace(p), and we need fast ways of locating an entry p in a priority queue

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

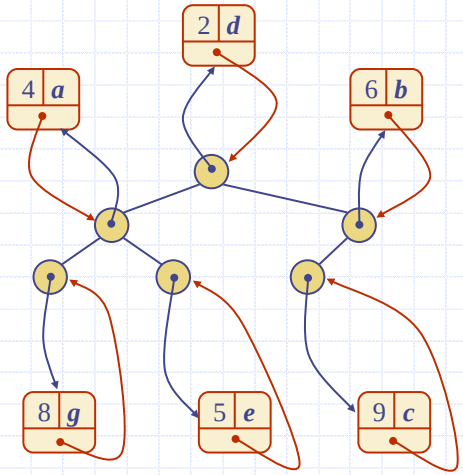
List Implementation

- 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, empty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(\log n)}$
replace	$\boldsymbol{O(1)}$	$O(n)$	$\boldsymbol{O(\log n)}$