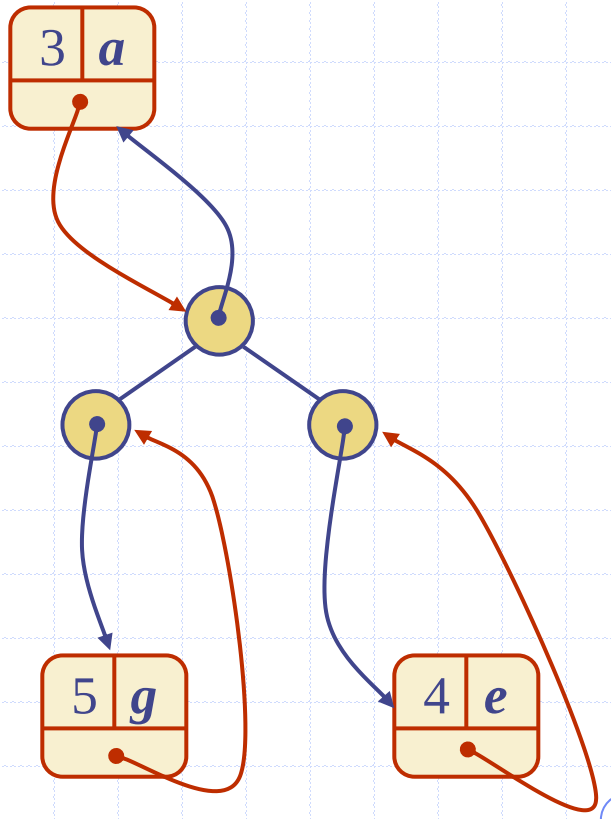
Adaptable Priority Queues and Maps

Sections 8.4 and 9.1



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, which 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

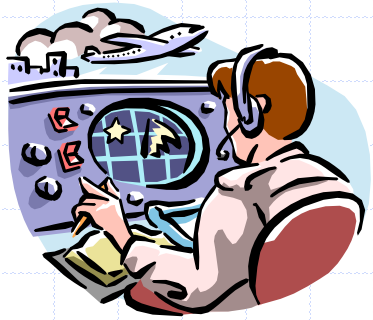
Example

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

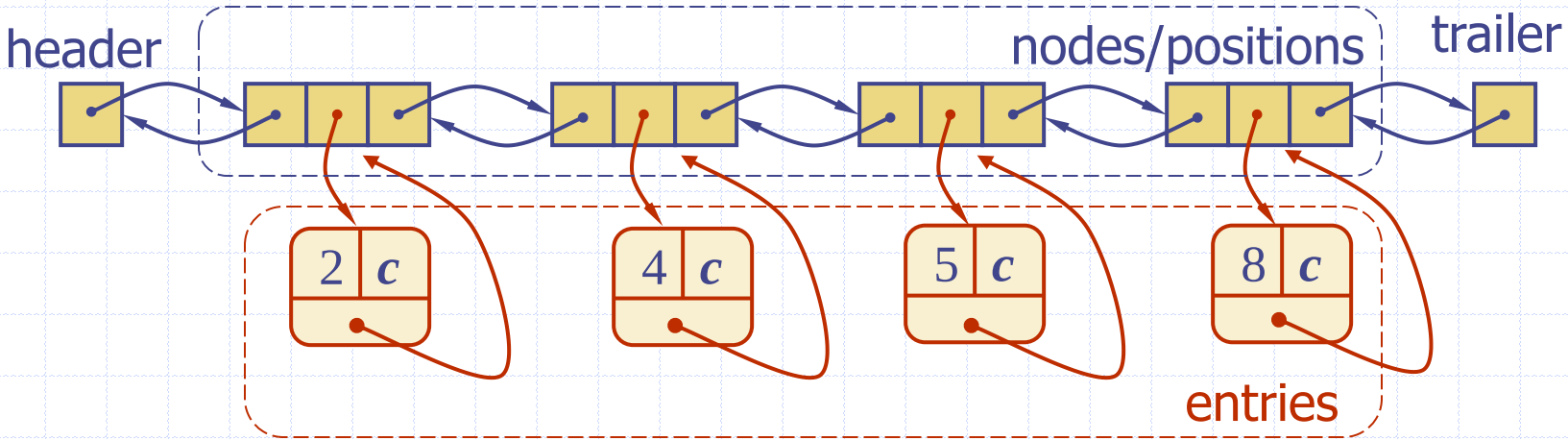
Location-Aware Entries



- 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

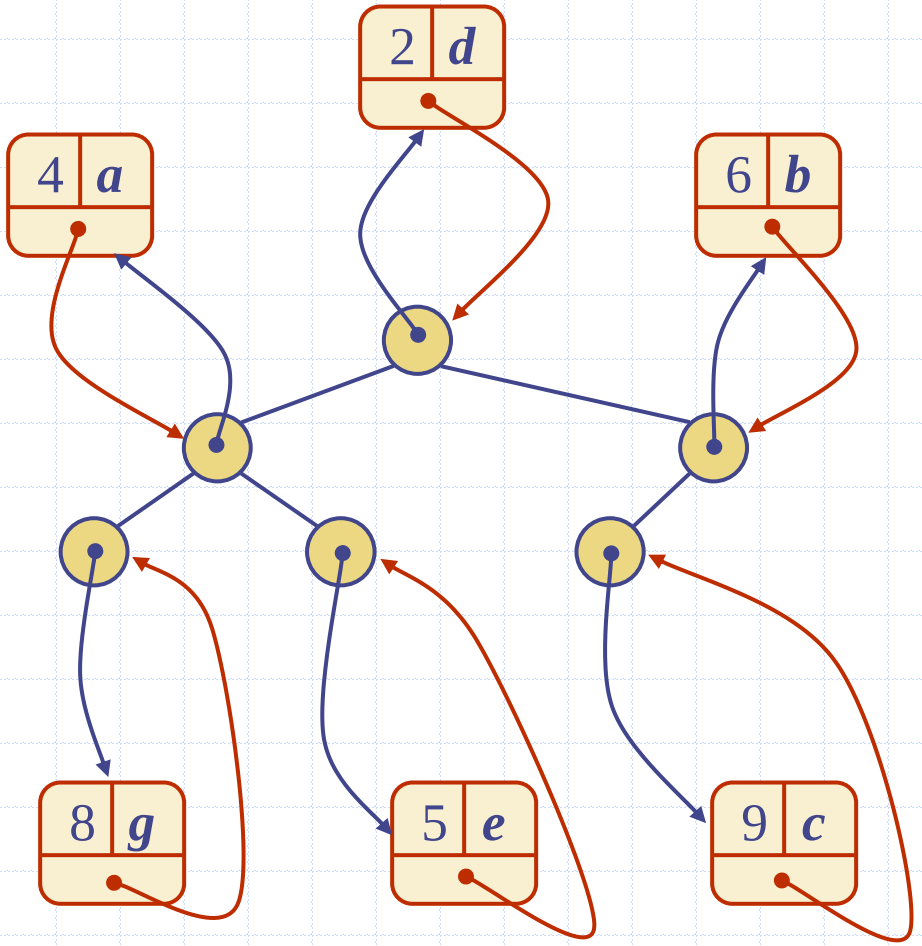
List Implementation

- 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	$\boldsymbol{O(1)}$	$O(n)$	$\boldsymbol{O(\log n)}$

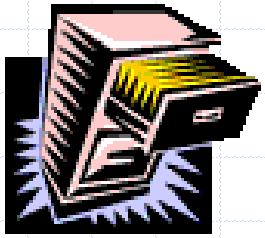
Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

Entry ADT

- An entry stores a key-value pair (k,v)
- Methods:
  - key(): return the associated key
  - value(): return the associated value
  - setKey(k): set the key to k
  - setValue(v): set the value to v

The Map ADT



- find(k): if the map M has an entry with key k, return an iterator to it; else, return special iterator end
- put(k, v): if there is no entry with key k, insert entry (k, v), and otherwise set the entry's value to v. Return an iterator to the new/modified entry
- erase(k): if the map M has an entry with key k, remove it from M. Otherwise give an error.
- erase(p): Remove from M the entry referenced by the iterator p. If p is end, or is not in the map, give an error.
- size(), empty()
- begin(), end(): return iterators to beginning and end of M

Example

Operation	*Output*	*Map*
empty()	**true**	Ø
put(5,A)	[(5,A)]	(5,A)
put(7,B)	[(7,B)]	(5,A),(7,B)
put(2,C)	[(2,C)]	(5,A),(7,B),(2,C)
put(8,D)	[(8,D)]	(5,A),(7,B),(2,C),(8,D)
put(2,E)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
find(7)	[(7,B)]	(5,A),(7,B),(2,E),(8,D)
find(4)	end	(5,A),(7,B),(2,E),(8,D)
find(2)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
erase(5)	—	(7,B),(2,E),(8,D)
erase(2)	—	(7,B),(8,D)
find(2)	end	(7,B),(8,D)
empty()	**false**	(7,B),(8,D)

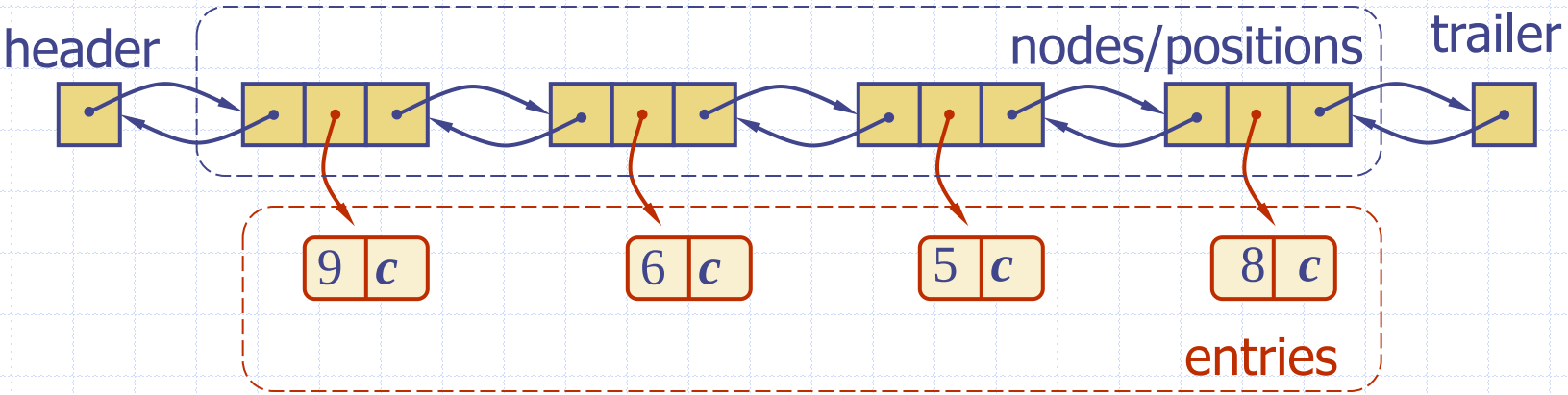
Informal Map Interface

```
template <typename K, typename V>
class Map {
public:
    class Entry;
    class Iterator;

    int size() const;
    bool empty() const;
    Iterator find(const K& k) const;
    Iterator put(const K& k, const V& v);
    void erase(const K& k)
        throw(NonexistentElement);
    void erase(const Iterator& p)
        throw(NonexistentElement);
    Iterator begin();
    Iterator end();
};
```

A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The find Algorithm

```
Algorithm find(k):
    for each p in [S.begin(), S.end()) do
        if p→key() = k then
            return p
    return S.end() {there is no entry with key equal to k}
```

We use p→key() as a shortcut for (*p).key()

The put Algorithm

```
Algorithm put(k,v):
    for each p in [S.begin(), S.end()) do
        if p→key() = k then
            p→setValue(v)
            return p
    p = S.insertBack((k,v)) {there is no entry with key k}
    n = n + 1 {increment number of entries}
    return p
```

The erase Algorithm

```
Algorithm erase(k):
    for each p in [S.begin(), S.end()) do
        if p.key() = k  then
            S.erase(p)
            n = n – 1      {decrement number of entries}
```

Performance of a List-Based Map

- Performance:
  - put takes $O(n)$ time since we need to determine whether it is already in the sequence
  - find and erase take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)