Priority Queues

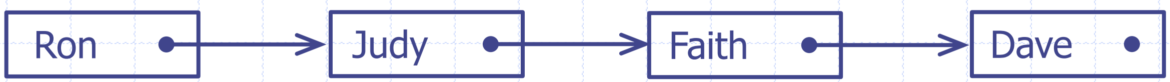
Sections 8.1-8.2

Priority Queue ADT

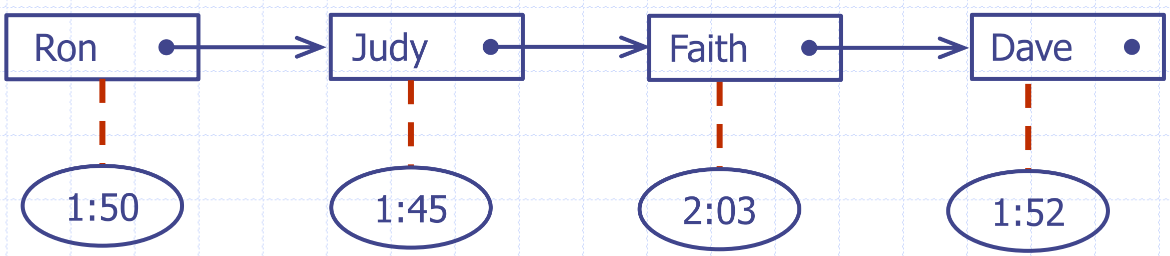
- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e)
    inserts an entry e
  - removeMin()
    removes an entry with smallest key (the one that min would return).
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Ranking

- Often, a program needs to rank, or assign a priority to, some of its objects (elements).
- This priority may be based on something internal to the object (ranking people by alphabetical order of their first name).

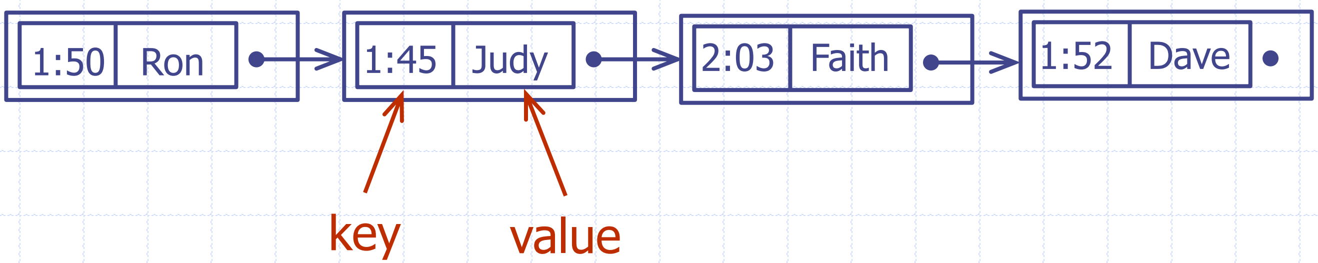


- It may also be based on something external to the object (ranking people by the time they walked into an office).



Ranking

- When the ranking is by something external, we need a way to associate the external data with elements. This is typically done with an entry, which is a pair (key, value) where the key is the external data indicating the priority and the value is the element being assigned a ranking.



- By considering entries rather than elements, we convert all rankings to use something internal to the object.

Comparisons of Entries

- Method 1: override operator< on entries.
  - This is not very flexible; it requires different code for each type of entry and each comparison method.
- Method 2: use a separate object to compare entries.
  - Flexible. Can have different objects for different types of comparisons; can use generic entry code.
  - This leads to the Comparator ADT.

```
template<typename K, typename V>
class Entry {
  private:
    K* key;
    V* value;
  …
}
```

Comparator ADT

- Implements the boolean function isLess(p, q), which tests whether p < q
- Can derive other relations from this:
  - (p == q) is equivalent to (!isLess(p, q) && !isLess(q, p))
- Can implement in C++ by overloading "()"

Two ways to compare 2D points:

```
class LeftRight {       // left-right comparator
public:
    bool operator()(const Point2D& p,
              const Point2D& q) const
    { return p.getX() < q.getX(); }
};
class BottomTop {    // bottom-top
public:
    bool operator()(const Point2D& p, const
    Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Using Comparators

```
template <typename E, typename C>         // element type and comparator type
void printSmaller(const E& p, const E& q, const C& isLess) {
    cout << (isLess(p, q) ? p : q) << endl;
}

…

Point2D p(1.3, 5.7), q(2.5, 0.6);
LeftRight leftRight;
BottomTop bottomTop;
printSmaller(p, q, leftRight);            // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);            // outputs: (2.5, 0.6)
```

An Informal Priority Queue Interface

```
template <typename E, typename C>      // element type and comparator type
class PriorityQueue {
public:
    int size() const;
    bool isEmpty() const;
    void insert(const E& e);
    const E& min() const throw(QueueEmpty);
    void removeMin() throw(QueueEmpty);
};
```

Total Order Relations

- The relationship encoded by a Comparator must be a consistent ordering.
  - Keys in a priority queue can be arbitrary objects on which an order is defined
  - Two distinct entries in a priority queue can have the same key.
  - Entries in a priority queue can be compared in any fashion at any time.
- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of min/removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
    Input sequence ***S***, comparator ***C*** for the elements of ***S***
    Output sequence ***S*** sorted in increasing order according to ***C***
    P ← priority queue with comparator ***C***
    while ¬***S.empty*** ()
        e ← ***S.front***()
        S.eraseFront()
        P.insert (***e***)
    while ¬***P.empty***()
        e ← ***P.min***()
        P.removeMin()
        S.insertBack(***e***)

Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$n + (n - 1) + \ldots + 1$$

- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	..	..
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to

     $$1 + 2 + \ldots + n$$

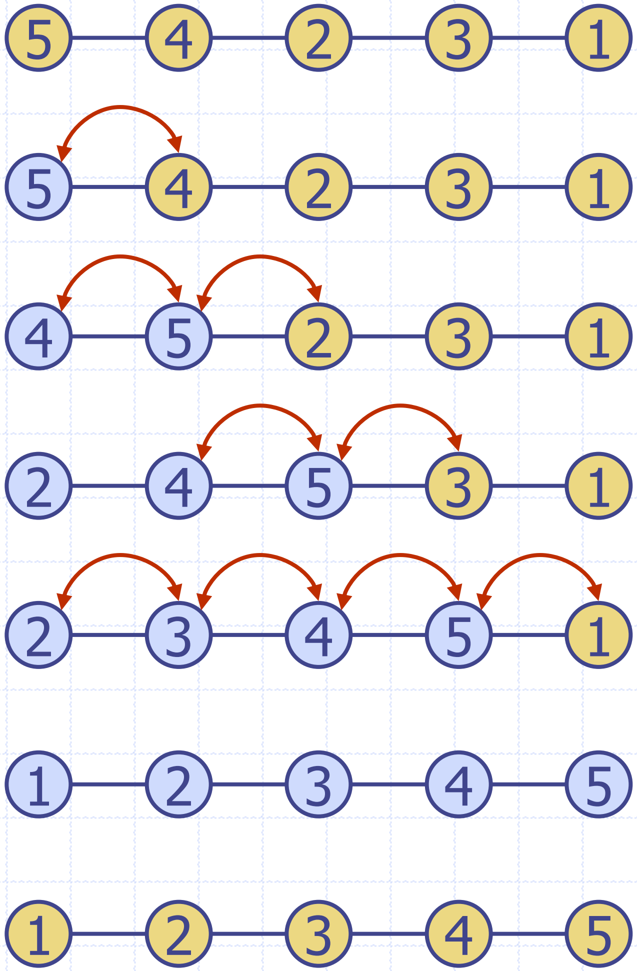
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Code Clarity

- Clarity is perhaps the most important aspect of clean professional code. Correctness can follow from clarity but not vice-versa.
- Code is written once but read and modified several times. It is worth putting extra effort into the writing to make the reading and modification easier.
- Two major contributors to clarity are naming and formatting.

Naming

- The naming of variables and functions is the most difficult and important thing in creating clean code. It determines whether your code communicates or obscures what it does.
- Choose names that are descriptive. Facility with natural language will help: dollarsShoes, dollarsForShoes, dollarsToShoes, dollarsInShoes, and dollarsByShoes all have different meanings.
- Don't squish words: does wrdSc mean word scope or weird science?

Naming

- Don't let a variable name get too long (say, 30 characters) or too short.
- In particular, don't use 1-letter variable names except i, j, k as indexes in for- or while- loops, i and j as integer arguments to a short function, and s and t as string arguments to short functions.
- Some standard abbreviations are okay:

numWidgets	the number of widgets
widgetNum	the number of a particular widget
maxWidgets	the maximum number of widgets
avgWidgets	the average number of widgets

Formatting

- Formatting is the layout of the code on the page/screen. It includes blank lines and indentation. Formatting should be done to increase readability.
- Use a standard formatting style, but be open to other people using other styles. Use whatever style is required at your workplace. If your workplace doesn't have a style guide, it should.
- For instance, one could use two blank lines between functions, one blank line between the function comment and the function itself, and one blank line between code lines to help indicate grouping of operations.

Formatting

- For indentation amount, 4 spaces seems standard, but I've also seen 2-space, 3-space, and tab indentation.
- Standard C++ style formatting is:

```
for (x; x; x) {
    xxxxxx;
    xxxxxx;
}
```

```
if (xxxxxx) {
    xxxxxx;
    xxxxxx;
}
else {
    xxxxxx;
    xxxxxx;
}
```

Formatting

- But I've also seen the following:

```
if (xxxxxx)
{
    xxxxxx;
    xxxxxx;
}
else
{
    xxxxxx;
    xxxxxx;
}
```

```
if (xxxxxx) {
    xxxxxx;
    xxxxxx;
} else {
    xxxxxx;
    xxxxxx;
}
```

Formatting

```
if (xxxxxx)
  {
    xxxxxx;
    xxxxxx;
  }
else
  {
    xxxxxx;
    xxxxxx;
  }
```