CS 106B, Lecture 15
Priority Queues and Heaps

reading:

Ch. 11.2, 11.4, 12.1 - 12.3

This document is copyright (C) Stanford Computer Science, Marty Stepp, Julie Zelenski, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Keith Schwarz, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, and others.

Prioritization problems

- **print jobs:** Lab printers accept jobs from all over the building. Faculty jobs print before staff, then grad, ugrad student jobs.

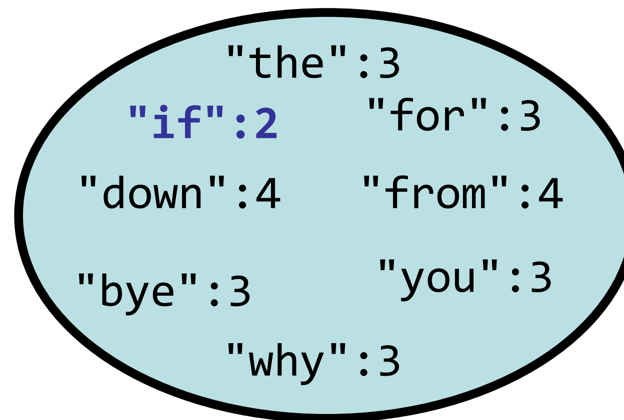
- **ER scheduling:** A gunshot victim should be treated sooner than a person with a cold, regardless of arrival time.

- *We want a "queue" with these operations:*
  - ***add*** *an element (print job, patient, etc.)*
  - ***get/remove*** *the* ***most "important"*** *or "urgent" element

Priority Queue ADT

- **priority queue**: Provides fast access to its highest-priority element.
  - **enqueue**: adds an element at a given priority
  - **peek**: returns highest-priority value
  - **dequeue**: removes/returns highest-priority value

```
pq.enqueue("if", 2);
pq.enqueue("from", 4);
...
```



`pq.dequeue()`

"if"

PriorityQueue members

`#include "priorityqueue.h"`

pq.**enqueue**(*value*, *pri*)	adds value to queue, with the given priority number
pq.**dequeue**()	returns the value in the queue with most urgent (minimum) priority; throws an error if queue is empty
pq.**peek**()	returns most urgent (minimum) priority element without removing it; throws error if queue is empty
pq.**peekPriority**()	returns *priority* of most urgent (minimum) priority element; throws error if queue is empty
pq.**isEmpty**()	returns `true` if queue contains no elements (size 0)
pq.**size**()	returns the number of elements in the queue
out **<<** *pq*	return/print a string such as `"{3, 42, -7, 15}"`
pq.**changePriority**(*value*, *pri*)	alters an existing element's priority to be more urgent
pq.**clear**()	removes all elements of the queue

```
PriorityQueue<string> faculty;
faculty.enqueue("Julie", 3);    // semi urgent priority
```

Exercise: SL scheduling

- Write code to show in what order our SLs choose their LaIR hours.
  - SLs with more seniority (quarters worked) get to choose first.
  - Each line of the input file contains the year the SL began working with us, and the quarter (1=fall, 2=winter, 3=spring, 4=summer).
  - Input file format:

    name year quarter

    name year quarter

    name year quarter

    ...

```
Zack 2014 2
Sara 2012 4
Tyler 2013 1
```

Exercise solution

```
PriorityQueue<string> SLs;   // read the contents of sls.txt
ifstream input;              // into a priority queue
input.open("sls.txt");
string slName;
int year;
int quarter;
while (input >> slName >> year >> quarter) {
    // store with year,quarter as priority so that the SLs
    // come out of the PQ in descending order of seniority
    // (e.g. year=2013, qtr=4 => priority = 20134)
    int priority = year * 10 + quarter;
    SLs.enqueue(slName, priority);
}

// pull the SLs out of the PQ from most to least seniority
while (!SLs.isEmpty()) {
    string sl = SLs.dequeue();
    cout << sl << " picks next." << endl;
}
```

PQ as array

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	x	b	a	m	q	t				
priority	5	4	8	5	5	2				
size	6	*capacity*			10					

unsorted

- PQ implemented using an unsorted **array**:
  - Which operations are slow? **enqueue**? **dequeue**? **peek**?
  - What is good/bad about this implementation?

PQ as sorted array

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	t	b	m	q	x	a				
priority	2	4	5	5	5	8				
size	6		*capacity*		10					

sorted

- PQ implemented using a *sorted* **array**:
  - Which operations are slow? **enqueue**? **dequeue**? **peek**?
  - What is good/bad about this implementation?

Heaps

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a			
priority		2	5	4	5	5	8			
size	6	*capacity*	10							

- **heap**: A special arrangement of elements in an array.
  - The start index is 1. *(index 0 is empty and unused)*
  - Every index *i* has a "parent" index: `i/2`
    and two "child" indexes: `i*2, i*2 + 1`
  - *Ordering:* Parents must have lower priority than their children.
  - called a "**binary min-heap**"

Min-heap add (enqueue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a	**k**		
priority		2	5	4	5	5	8	**1**		
size	**7**		*capacity*		10					

– `pq.`**`enqueue`**`("k", 1);`

- **enqueue**: place new element at first empty index.
  – But now it may be out-of-order.
  – So **swap it upward** with its parent until it is in order.
    - Is this fast or slow?

Min-heap bubble-up

index	0	1	2	3	4	5	6	7	8	9
value		k	m	t	x	q	a	b		
priority		1	5	2	5	5	8	4		

size 7 capacity 10

bubble up "k"

- The bubble-up process for `"k":1` :
  - index 7 (k:1) swaps up with index 3 (b:4)
  - index 3 (k:1) swaps up with index 1 (t:2)
  - Not every added element bubbles all the way to the top!

Implementing peek

index	0	1	2	3	4	5	6	7	8	9
value		k	m	t	x	q	a	b		
priority		1	5	2	5	5	8	4		
size	7	capacity	10							

- `pq.peek()` --> "k"
- `pq.peekPriority()` --> 1

- Finding the min-priority element in a min-heap is trivial.
  - It is always located at index 1!
    - Is this fast or slow?

Heap remove (dequeue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		**k**	c	f	p	e	v	**y**		
priority		**1**	2	4	7	5	8	**6**		
size	7	*capacity*	10							

- `pq.`**`dequeue`**`()      --> "k"`

- When removing the min-priority element from a heap:
  - First move the last element up to the start, index 1.
  - Then swap it downward with its *most-urgent child* until in order.
    - This process is called "**bubbling down**" or "percolating down".
    - Is this fast or slow?

Heap bubble-down

– move index 7 (y:6) up to index 1

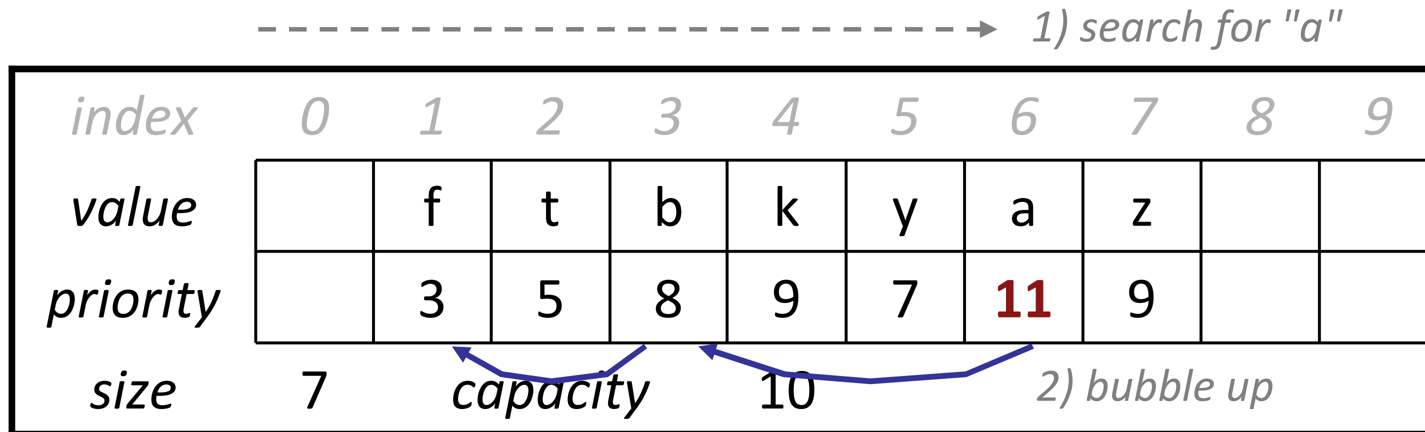
index	0	1	2	3	4	5	6	7	8	9
value		y	c	f	p	e	v			
priority		6	2	4	7	5	8			
size	6	capacity		10						

– swap index 1 with most urgent child at index 2 (c:2)

– swap index 2 with most urgent child at index 5 (e:5)

index	0	1	2	3	4	5	6	7	8	9
value		c	e	f	p	**y**	v			
priority		2	5	4	7	**6**	8			
size	6	capacity		10						

Heap change priority



- `pq.changePriority("a", 2);`

- To implement a **change-priority** operation:
  - Loop sequentially over the array to find the element
  - Set its new priority and "**bubble up**" the element until in order
    - This will restore the heap ordering property.
  - Is this fast or slow?

Max-Heap

index	0	1	2	3	4	5	6	7	8	9
value		a	t	b	x	q	a	m		
priority		8	4	2	5	7	3	9		
size	7	*capacity*		10						

- **max-heap**: Parents must have *higher* priority than their children.
  - All algorithms are the same, but when bubbling, use > for comparison rather than <