CS 10:
Problem solving via Object Oriented Programming

Prioritizing

Agenda

1. Priority Queue ADT
2. Implementation choices
3. Java's built-in PriorityQueue
4. Reading from a file

We can model airplanes landing as a queue

Airplanes queued to land

Each airplane assigned a priority to land in order of arrival

First in traffic pattern, first to land

Image: flickr

Sometimes higher priority issues arise and we need to change order

Airplanes queued to land

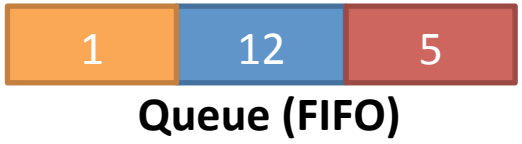
Suddenly one aircraft has an in flight emergency, and needs to land now!

Need a way to go to front of queue

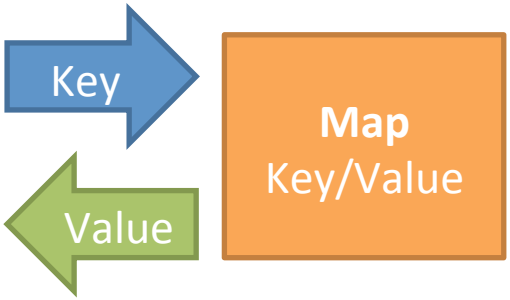
Enter the priority queue

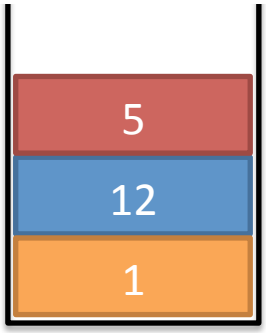
Image: flickr

Priority Queues store/retrieve objects based on priority, not identity



Queue (FIFO)





Stack (LIFO)

Priority Queue

Maps are a Key/Value store

- *put(Key, Value)* stores a *Value* associated with a *Key* (e.g., Key: Student ID and Value: Student Record)
- *get(Key)* return *Value* associated with *Key*
- Keys unique; identify object
- No ordering among Keys

Stacks/Queues arrival order

- Item order depends on when item arrived
- Only one item accessible at any time (top or front)

Priority Queue order

- Items stored/ retrieved by *priority*
- Priority does not represent identity as with a Map Key
- Not dependent on arrival order like Stack/Queue

Priority Queues have the ability to extract the highest priority item

Min Priority Queue Overview

- Lowest priority number removed first ("number 1 for landing")
- Can be used for sorting (put everything in, then repeatedly extract lowest priority number, one at a time, until queue empty)
- **Operations**
  - `insert(element)` – insert `element` into Priority Queue
    - Like BST, elements need a way to compare with each other to see which is the smallest, so `element` should implement *compareTo()*
    - We will say whatever *compareTo()* uses to compare elements is the *Key*
    - Many elements can have the same Key in a Priority Queue
  - `extractMin()` – remove and return element with smallest Key
  - `minimum()` – return element with smallest Key, but leaves the element in Priority Queue (like *peek()* or *front()* in Stack or Queue)
  - `isEmpty()` – true if no items stored, false otherwise
  - `decreaseKey()` – reduces an element's priority number (take CS 31 for more details on this)

Max Priority Queue works similarly, but extracts the largest priority item with extractMax()

Priority Queues are extensively used in simulations and scheduling

Job scheduling example

Machine 1

Start job at time 0
Job takes 11 minutes

Add to Priority Queue that job will finish at time 11

Machine 2

Start job at time 2
Job takes 6 minutes

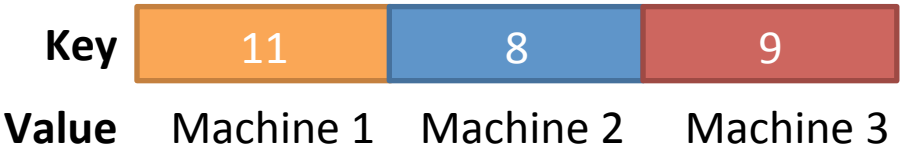
Add to Priority Queue that job will finish at time 8

Machine 3

Start job at time 4
Job takes 5 minutes

Add to Priority Queue that job will finish at time 9

Priority Queue



Which machine will finish first? When will it be?
***extractMin()* to find out**

No need to run simulation and check each minute to see if any machine finishes at times 0 through 7; can jump to time 8

Which machine will finish next?
***extractMin()* again and get time 9**

MinPriorityQueue.java specifies interface

MinPriorityQueue.java

```
 6 public interface MinPriorityQueue<E extends Comparable<E>> {
 7     /**
 8      * Is the priority queue empty?
 9      * @return true if the priority queue is empty, false if not empty.
10      */
11     public boolean isEmpty();
12
13     /**
14      * Insert an element into the queue.
15      * @param element thing to insert
16      */
17     public void insert(E element);
18
19     /**
20      * Return the element with the minimum key, without removing it from the queue.
21      * @return the element with the minimum key in the priority queue
22      */
23     public E minimum();
24
25     /**
26      * Return the element with the minimum key, and remove it from the queue.
27      * @return the element with the minimum key in the priority queue
28      */
29     public E extractMin();
30 }
```

- As with BST, elements must extend Comparable
- Allows Java to compare elements and determine which one is smaller
- Uses *compareTo()* method on element objects
- Can make a Max Priority Queue by reversing the *compareTo()* method
- Note: no ability to get items by index!
- Can only extract smallest (or largest) item

Agenda

1. Priority Queue ADT
2. Implementation choices
3. Java's built-in PriorityQueue
4. Reading from a file

There are a number of implementation choices, but some are not a good fit

Choice	Fit	Notes
Stack/Queue		• Elements ordered by arrival time • Can only access one element (top or front) • Element with higher priority that arrives out of sequence can not be reached

There are a number of implementation choices, but some are not a good fit

Choice	Fit	Notes
Stack/Queue		• Elements ordered by arrival time • Can only access one element (top or front) • Element with higher priority that arrives out of sequence can not be reached
Map		• Have to know the Key in order to find item • In scheduling example, would have to check each minute 0 through 7

There are a number of implementation choices, but some are not a good fit

Choice	Fit	Notes
Stack/Queue		• Elements ordered by arrival time • Can only access one element (top or front) • Element with higher priority that arrives out of sequence can not be reached
Map		• Have to know the Key in order to find item • In scheduling example, would have to check each minute 0 through 7
Unsorted List		• *insert()* fast, O(1) • *extractMin()* slow – search entire List for min Key, O(n)

There are a number of implementation choices, but some are not a good fit

Choice	Fit	Notes
Stack/Queue		• Elements ordered by arrival time • Can only access one element (top or front) • Element with higher priority that arrives out of sequence can not be reached
Map		• Have to know the Key in order to find item • In scheduling example, would have to check each minute 0 through 7
Unsorted List		• *insert()* fast, O(1) • *extractMin()* slow – search entire List for min Key, O(n)
Sorted List		• *extractMin()* fast, O(1) • *insert()* slow – find right place, make hole, O(n)

There are a number of implementation choices, but some are not a good fit

Choice	Fit	Notes
Stack/Queue		• Elements ordered by arrival time • Can only access one element (top or front) • Element with higher priority that arrives out of sequence can not be reached
Map		• Have to know the Key in order to find item • In scheduling example, would have to check each minute 0 through 7
Unsorted List		• *insert()* fast, O(1) • *extractMin()* slow – search entire List for min Key, O(n)
Sorted List		• *extractMin()* fast, O(1) • *insert()* slow – find right place, make hole, O(n)
Binary Search Tree		• Not bad, but we do not enforce balance on BST • *extractMin() O(h)* (could be better than O(n), but not necessarily) • We will do better next class using a Heap

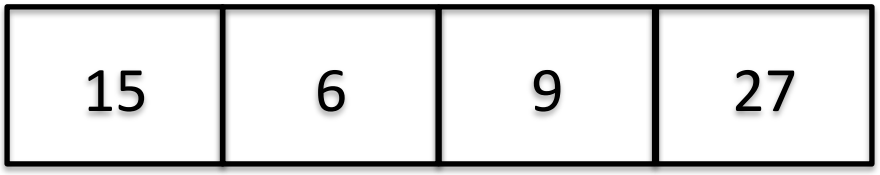
There are several ways to implement a PriorityQueue, today we look at Lists

1. **Unsorted List**

2. **Sorted List**

We can implement a PriorityQueue with an unsorted ArrayList

Unsorted ArrayList implementation

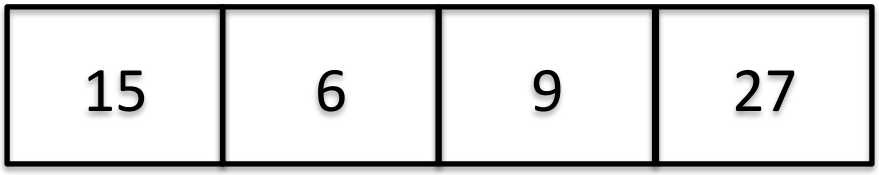


Keep elements unsorted in ArrayList

isEmpty() is O(1) with an unsorted ArrayList

Unsorted ArrayList implementation

isEmpty()



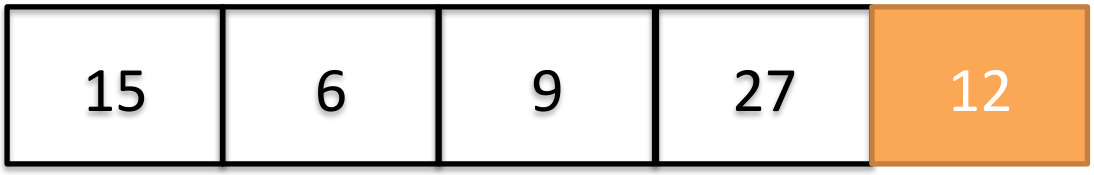
isEmpty – just check ArrayList *size()* method

Operation	Run time	Notes
`isEmpty`	**O(1)**	**Checks size == 0**

insert() is also O(1) with an unsorted ArrayList

Unsorted ArrayList implementation

insert(12)



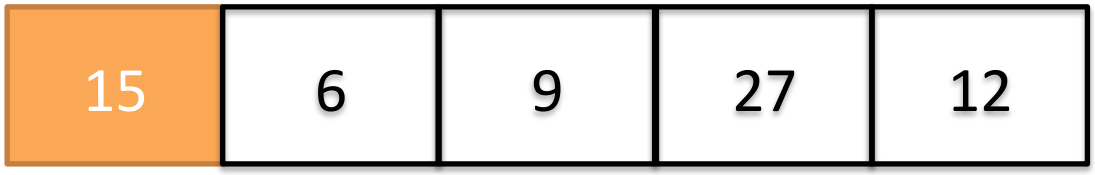
insert – just add element to end of ArrayList

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	**O(1)**	**Add on to end (amortized)**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()
Check 15



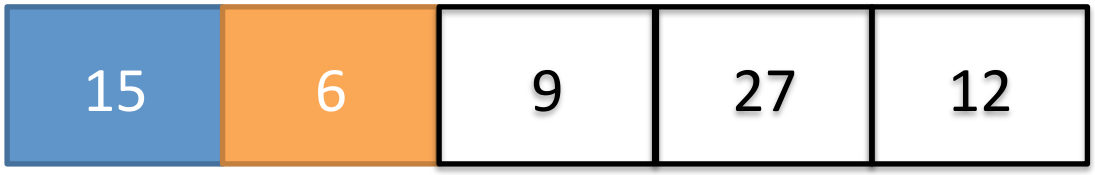
extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()
Check 6
Smallest 15



extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()
Check 9
Smallest 6

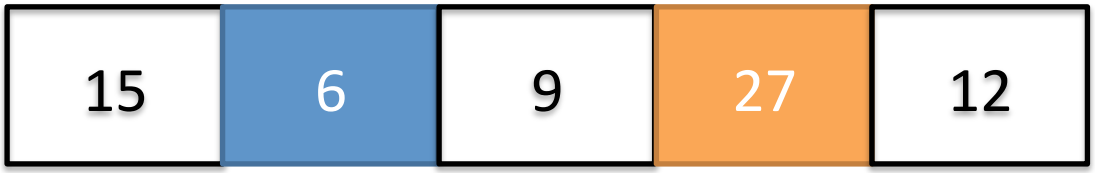
extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()
Check 27
Smallest 6



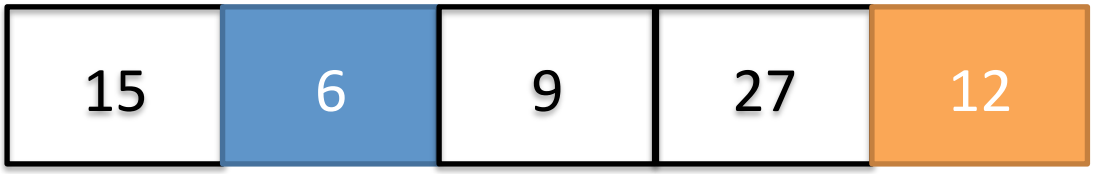
extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()
Check 12
Smallest 6



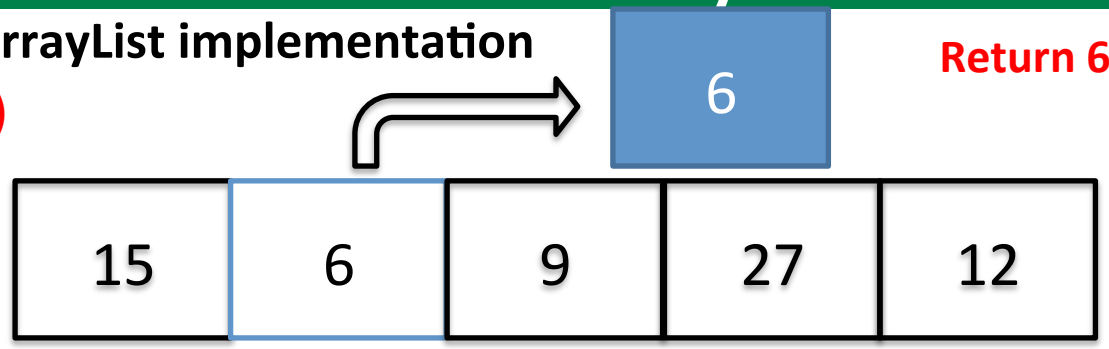
extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()



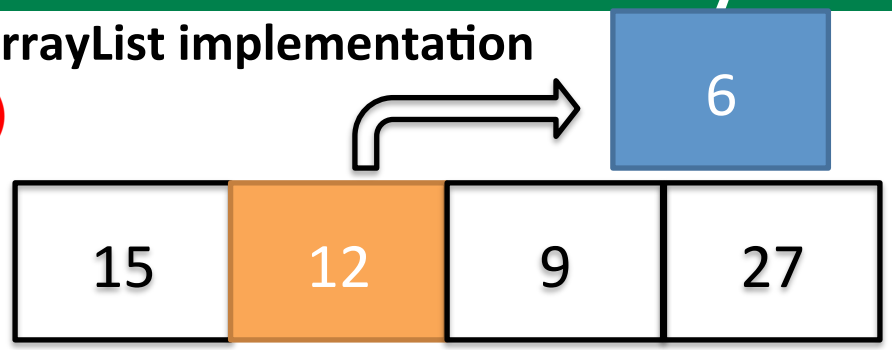
extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

minimum() and *extractMin()* are both O(n) with an unsorted ArrayList

Unsorted ArrayList implementation

extractMin()



Return 6

Fill hole with last item

No need to slide items left

Nice

We will use this trick again with Heaps

extractMin – loop to find smallest and move last item to smallest index to fill hole

Operation	Run time	Notes
`isEmpty`	O(1)	Checks size == 0
`insert`	O(1)	Add on to end (amortized)
`minimum`	**O(n)**	**Must loop through all elements to find smallest**
`extractMin`	**O(n)**	**Loop through all elements and move to fill hole**

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

```
 8 public class ArrayListMinPriorityQueue<E extends Comparable<E>>
 9 implements MinPriorityQueue<E> {
10     private ArrayList<E> list;  // list of elements
11
12     /**
13      * Constructor
14      */
15     public ArrayListMinPriorityQueue() {
16         list = new ArrayList<E>();
17     }
18
19     /**
20      * Is the priority queue empty?
21      * @return true if the queue is empty, false if not empty.
22      */
23     public boolean isEmpty() {
24         return list.size() == 0;
25     }
26
27     /**
28      * Insert an element into the priority queue.
29      * Keep in decreasing order
30      * @param element the element to insert
31      */
32     public void insert(E element) {
33         list.add(element);
34     }
```

- **Implements MinPriorityQueue interface using ArrayList**
- **Store elements in ArrayList called *list***
- **Elements must provide *compareTo()* because we say E extends *Comparable***
- ***isEmpty()* just checks ArrayList *size()* method**
- **Inserting is easy, just tack new element on to end of ArrayList**

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

```
40    private int indexOfMinimum() {
41        // Search through the entire array for the smallest element.
42        int smallestIndex = 0;
43
44        for (int i = 1; i < list.size(); i++) {
45            // If the current smallest is greater than the element at index i,
46            // then make the element at index i the new smallest.
47            if (list.get(smallestIndex).compareTo(list.get(i)) > 0)
48                smallestIndex = i;
49        }
50
51        return smallestIndex;
52    }
53
54
55    /**
56     * Return the element with the minimum key, and remove it from the queue.
57     * @return the element with the minimum key, or null if queue empty.
58     */
59    public E extractMin() {
60        if (list.size() <= 0)
61            return null;
62        else {
63            int smallest = indexOfMinimum();   // index of the element with the
64            E minElement = list.get(smallest);  // the actual element
65
66            // Move the element in the last position to this position.
67            // Faster than removing from the middle.
68            list.set(smallest, list.get(list.size()-1));
69
70            // We no longer have an element in that last position.
71            list.remove(list.size()-1);
72
73            // Return the element with the minimum key.
74            return minElement;
75        }
76    }
```

- **extractMin() finds smallest index with call to *indexOfMin()***

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

```
40    private int indexOfMinimum() {
41        // Search through the entire array for the smallest element.
42        int smallestIndex = 0;
43
44        for (int i = 1; i < list.size(); i++) {
45            // If the current smallest is greater than the element at index i,
46            // then make the element at index i the new smallest.
47            if (list.get(smallestIndex).compareTo(list.get(i)) > 0)
48                smallestIndex = i;
49        }
50
51        return smallestIndex;
52    }
53
54
55    /**
56     * Return the element with the minimum key, and remove it from the queue.
57     * @return the element with the minimum key, or null if queue empty.
58     */
59    public E extractMin() {
60        if (list.size() <= 0)
61            return null;
62        else {
63            int smallest = indexOfMinimum();   // index of the element with the
64            E minElement = list.get(smallest);  // the actual element
65
66            // Move the element in the last position to this position.
67            // Faster than removing from the middle.
68            list.set(smallest, list.get(list.size()-1));
69
70            // We no longer have an element in that last position.
71            list.remove(list.size()-1);
72
73            // Return the element with the minimum key.
74            return minElement;
75        }
76    }
```

Loop through all elements, compare (using *compareTo()*)with smallest so far, return index of smallest element O(n)

- **extractMin() finds smallest index with call to *indexOfMin()***

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

```
40    private int indexOfMinimum() {
41        // Search through the entire array for the smallest element.
42        int smallestIndex = 0;
43
44        for (int i = 1; i < list.size(); i++) {
45            // If the current smallest is greater than the element at index i,
46            // then make the element at index i the new smallest.
47            if (list.get(smallestIndex).compareTo(list.get(i)) > 0)
48                smallestIndex = i;
49        }
50
51        return smallestIndex;
52    }
53
54
55    /**
56     * Return the element with the minimum key, and remove it from the queue.
57     * @return the element with the minimum key, or null if queue empty.
58     */
59    public E extractMin() {
60        if (list.size() <= 0)
61            return null;
62        else {
63            int smallest = indexOfMinimum();   // index of the element with the
64            E minElement = list.get(smallest); // the actual element
65
66            // Move the element in the last position to this position.
67            // Faster than removing from the middle.
68            list.set(smallest, list.get(list.size()-1));
69
70            // We no longer have an element in that last position.
71            list.remove(list.size()-1);
72
73            // Return the element with the minimum key.
74            return minElement;
75        }
76    }
```

Loop through all elements, compare (using *compareTo()*)with smallest so far, return index of smallest element O(n)

- **extractMin() finds smallest index with call to *indexOfMin()***
- **Store smallest element**

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

```
40    private int indexOfMinimum() {
41        // Search through the entire array for the smallest element.
42        int smallestIndex = 0;
43
44        for (int i = 1; i < list.size(); i++) {
45            // If the current smallest is greater than the element at index i,
46            // then make the element at index i the new smallest.
47            if (list.get(smallestIndex).compareTo(list.get(i)) > 0)
48                smallestIndex = i;
49        }
50
51        return smallestIndex;
52    }
53
54
55    /**
56     * Return the element with the minimum key, and remove it from the queue.
57     * @return the element with the minimum key, or null if queue empty.
58     */
59    public E extractMin() {
60        if (list.size() <= 0)
61            return null;
62        else {
63            int smallest = indexOfMinimum();   // index of the element with the
64            E minElement = list.get(smallest); // the actual element
65
66            // Move the element in the last position to this position.
67            // Faster than removing from the middle.
68            list.set(smallest, list.get(list.size()-1));
69
70            // We no longer have an element in that last position.
71            list.remove(list.size()-1);
72
73            // Return the element with the minimum key.
74            return minElement;
75        }
76    }
```

Loop through all elements, compare (using *compareTo()*)with smallest so far, return index of smallest element O(n)

- **extractMin() finds smallest index with call to *indexOfMin()***
- **Store smallest element**
- **Move last element into index of smallest to avoid creating a hole**

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

```
40    private int indexOfMinimum() {
41        // Search through the entire array for the smallest element.
42        int smallestIndex = 0;
43
44        for (int i = 1; i < list.size(); i++) {
45            // If the current smallest is greater than the element at index i,
46            // then make the element at index i the new smallest.
47            if (list.get(smallestIndex).compareTo(list.get(i)) > 0)
48                smallestIndex = i;
49        }
50
51        return smallestIndex;
52    }
53
54
55    /**
56     * Return the element with the minimum key, and remove it from the queue.
57     * @return the element with the minimum key, or null if queue empty.
58     */
59    public E extractMin() {
60        if (list.size() <= 0)
61            return null;
62        else {
63            int smallest = indexOfMinimum();   // index of the element with the
64            E minElement = list.get(smallest); // the actual element
65
66            // Move the element in the last position to this position.
67            // Faster than removing from the middle.
68            list.set(smallest, list.get(list.size()-1));
69
70            // We no longer have an element in that last position.
71            list.remove(list.size()-1);
72
73            // Return the element with the minimum key.
74            return minElement;
75        }
76    }
```

Loop through all elements, compare (using *compareTo()*)with smallest so far, return index of smallest element O(n)

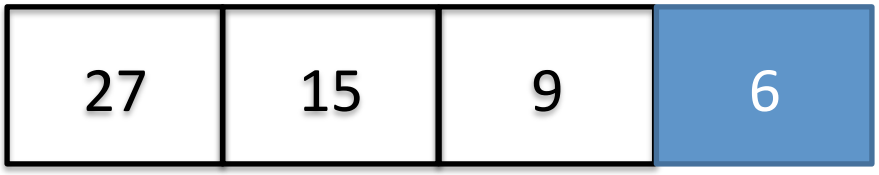
- **extractMin() finds smallest index with call to *indexOfMin()***
- **Store smallest element**
- **Move last element into index of smallest to avoid creating a hole**
- **Remove last item and then return smallest element**

There are several ways to implement a PriorityQueue, today we look at two

1. **Unsorted List**
2. **Sorted List**

We can improve *extractMin()* by using a sorted List, but inserts take more time

Sorted ArrayList implementation

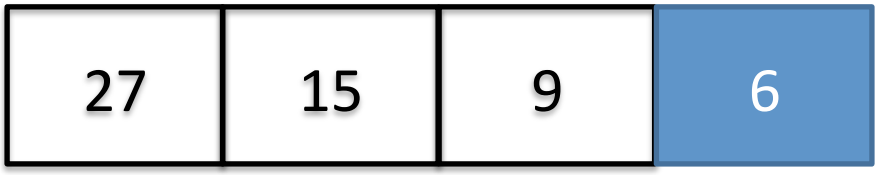


Keep elements sorted in ArrayList with smallest always at end

isEmpty() is O(1) with a sorted ArrayList

Sorted ArrayList implementation

isEmpty()



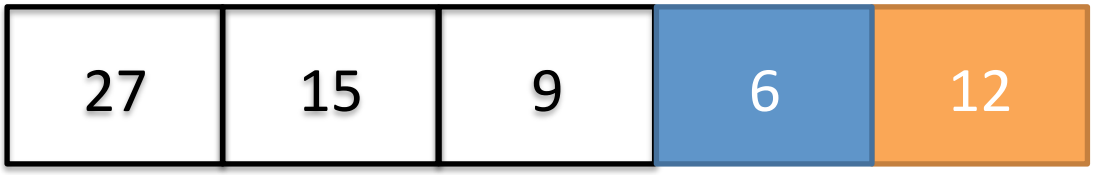
isEmpty() – just check ArrayList *size()* method

Operation	Run time	Notes
`isEmpty`	**O(1)**	**Return size, same as unsorted**

insert() is O(n) with a sorted ArrayList

Sorted ArrayList implementation

insert(12)



insert() – need to loop backward to find slot for new element, then move other elements right

Operation	Run time	Notes
`isEmpty`	O(1)	Return size, same as unsorted
`insert`	**O(n)**	**Insert in place and move other items right**

insert() is O(n) with a sorted ArrayList

Sorted ArrayList implementation

insert(12)

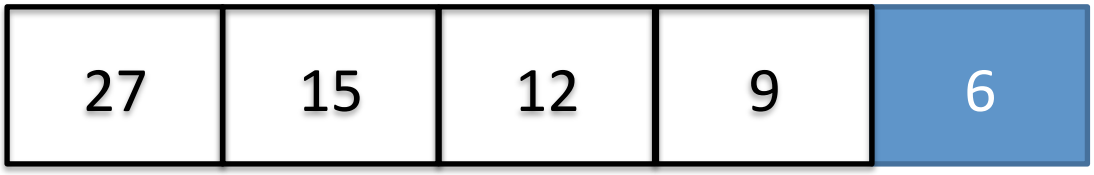
insert() – need to loop backward to find slot for new element, then move other elements right

Operation	Run time	Notes
`isEmpty`	O(1)	Return size, same as unsorted
`insert`	**O(n)**	**Insert in place and move other items right**

minimum() and *extractMin()* improve to O(1) with a sorted ArrayList

Sorted ArrayList implementation

extractMin()



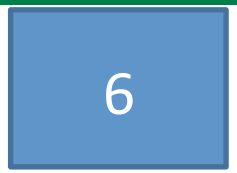
extractMin() – just remove the last element

Operation	Run time	Notes
`isEmpty`	O(1)	Return size, same as unsorted
`insert`	O(n)	Insert in place and move other items right
`minimum`	**O(1)**	**Get last element**
`extractMin`	**O(1)**	**Get last element, no need to move items**

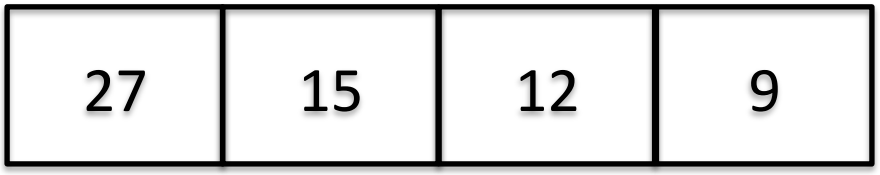
minimum() and *extractMin()* improve to O(1) with a sorted ArrayList

Sorted ArrayList implementation

extractMin()



Return 6



extractMin() – just remove the last element

Operation	Run time	Notes
`isEmpty`	O(1)	Return size, same as unsorted
`insert`	O(n)	Insert in place and move other items right
`minimum`	**O(1)**	**Get last element**
`extractMin`	**O(1)**	**Get last element, no need to move items**

SortedArrayList implementation improves *extractMin()*, but at expense of *insert()*

SortedArrayListMinPriorityQueue.java

```
33      public void insert(E element) {
34          int p;  // Current position in the list.
35
36          //loop backward from end toward front
37          //continue if new element larger than current element
38          for (p = list.size(); p > 0 && list.get(p-1).compareTo(element) < 0; p--)
39              ;
40
41          //add new element at index found
42          list.add(p, element);
43      }
44
45
46      /**
47       * Return the element with the minimum key, without removing it from the queue.
48       * @return the element with the minimum key, or null if queue empty.
49       */
50      public E minimum() {
51          if (list.size() == 0)
52              return null;
53          else
54              return list.get(list.size() - 1);  // Last item is smallest
55      }
56
57      /**
58       * Return the element with the minimum key, and remove it from the queue.
59       * @return the element with the minimum key, or null if queue empty.
60       */
61      public E extractMin() {
62          if (list.size() == 0)
63              return null;
64          else {                              // Shrink the size
65              return list.remove(list.size()-1);  // and return the smallest element
66          }
67      }
68
```

Store elements in ArrayList called *list*

***insert()* is O(n)**
Loop backward to find appropriate slot *p*, O(n)
Insert element at that slot
***add(p,element)* moves other elements right which is also O(n), plus O(1) for actual insert into array**
total = O(n) + O(n) +O(1) = O(2n+1) = O(n)

minimum()* and *extractMin()* are easy, just get/remove last element in *list

Implementations have different strengths, but no practical difference

Operation	Unsorted	Sorted
`isEmpty`	O(1)	O(1)
`insert`	O(1)	**O(n)**
`minimum`	**O(n)**	O(1)
`extractMin`	**O(n)**	O(1)

- Generally have the same number of inserts as extracts, so no real difference, unless just looking for min without extracting
- We will do better next class when we look at heaps!

Agenda

1. Priority Queue ADT
2. Implementation choices
3. Java's built-in PriorityQueue
4. Reading from a file

Java implements a *PriorityQueue*, but with non-standard names

Java's *PriorityQueue* Operations

- *isEmpty == isEmpty*
- *insert == add*
- *minimum == peek*
- *extractMin == remove*

Why *remove()* instead of *extractMin()?*
We will control if the min or max gets removed (next slides show how)

If we use our own Objects in *PriorityQueue*, need to provide way to compare objects

Student.java

Three ways to compare objects in Java's Priority Queue:

- Method 1: Objects stored in Priority Queue provide a *compareTo()* method

- Method 2: Instantiate a custom Comparator and pass to Priority Queue constructor

- Method 3: Use anonymous function in Priority Queue declaration

Use Student object to demonstrate the three Priority Queue methods

Student.java

```
11 public class Student implements Comparable<Student> {
12     private String name;
13     private int year;
14
15     public Student(String name, int year) {
16         this.name = name;
17         this.year = year;
18     }
19
20     /**
21      * Comparable: just use String's version (lexicographic)
22      */
23     @Override
24     public int compareTo(Student s2) {
25         return name.compareTo(s2.name);
26     }
27
28     @Override
29     public String toString() {
30         return name + " '"+year;
31     }
```

Student class implements Comparable so PriorityQueue holding *Student objects* can compare students

Student stores data about a student's *name* and *year*

If we are going to use *Student* in a PriorityQueue, need a way to tell which ones are bigger, the same, or smaller than other Students

This approach sorts increasing alphabetically by student name

Here we use the built in String *compareTo()* method to evaluate *Students* based on *name* (could reverse *compareTo()* for descending order)

- If this *name < s2.name* return negative
- If this *name equals s2.name* return 0
- If this *name > s2.name* return positive

Method 1: Objects in Priority Queue provide *compareTo()* method

Student.java

```
33    public static void main(String[] args) {
34        //create ArrayList of students and add some
35        ArrayList<Student> students = new ArrayList<Student>();
36        students.add(new Student("charlie", 18));
37        students.add(new Student("alice", 20));
38        students.add(new Student("bob", 19));
39        students.add(new Student("elvis", 21));
40        students.add(new Student("denise", 20));
41        System.out.println("original:" + students);
42
43        // Three methods for using Comparator
44
45        // Method 1:
46        // Create Java PriorityQueue and use Student
47        // class's compareTo method (lexicographic order)
48        // this is used if comparator not passed to PriorityQueue constructor
49        PriorityQueue<Student> pq = new PriorityQueue<Student>();
50        pq.addAll(students); //add all Students in ArrayList in one statement
51
52        //remove until empty (this essentially sorting!)
53        System.out.println("\nlexicographic:");
54        while (!pq.isEmpty()) System.out.println(pq.remove());
55
56
```

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> Student [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 16, 2018, 11:02:00 AM)

```
original:[charlie '18, alice '20, bob '19, elvis '21, denise '20]

lexicographic:
alice '20
bob '19
charlie '18
denise '20
elvis '21
```

Output in alphabetical order

- ***Student* Objects added to ArrayList in undefined order**
- ***Student* objects have *name* and *year* instance variables**

- **Priority Queue created to hold *Student* Objects**
- **No Comparator provided in constructor**
- **By default PriorityQueue will use *Student* object's *compareTo()* to find min Key**
- **ArrayList of students is added to PriorityQueue with *addAll()* method**

- **Output in sorted order**
- **Each time while loop executes, removes smallest *Student* object using *compareTo()***

If we use our own PriorityQueue, we need to provide way to compare objects

Student.java

Three ways to compare objects in Java's Priority Queue:

- Method 1: Objects stored in PriorityQueue provide a *compareTo()* method

- Method 2: Instantiate a custom Comparator and pass to Priority Queue constructor

- Method 3: Use anonymous function in Priority Queue declaration

Method 2: Define custom Compator and pass to Priority Queue constructor

Student.java

What if Object has *compareTo()* but you want a different order?

```
57          // Method 2:
58          // Use a custom Comparator.compare (length of name) instead
59          // of using the element's compareTo function
60          // Java will use this to compare two Students (here on length of name)
61          class NameLengthComparator implements Comparator<Student> {
62              public int compare(Student s1, Student s2) {
63                  return s1.name.length() - s2.name.length();
64              }
65          }
66          Comparator<Student> lenCompare = new NameLengthComparator();
67          pq = new PriorityQueue<Student>(lenCompare); //passing Comparator to PriorityQueue
68          pq.addAll(students); //add all students to PriorityQueue
69          System.out.println("\nlength:");
70          //remove until empty (sorting)
71          while (!pq.isEmpty()) System.out.println(pq.remove());
72
73          //Method 3:
74          // Use a custom Comparator via Java 8 anonymous function (here based on year)
75          // pass Comparator to PriorityQueue constructor
76          pq = new PriorityQueue<Student>((Student s1, Student s2) -> s2.year - s1.year);
77          pq.addAll(students); //add all students to Priority Queue
78          System.out.println("\nyear:");
79          //remove until empty (sorting)
80          while (!pq.isEmpty()) System.out.println(pq.remove());
81      }
82  }
83
```

- Still in *main()*
- Define Comparator class that requires *compare()* method
- *compare()* has two *Student* params
- Here we use length of *name* to compare two *Student* Objects
- *compare()* returns negative, equal, or positive same as *compareTo()*

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> Student [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 10, 2018, 11:21:36 AM)

```
length:
bob '19
elvis '21
alice '20
denise '20
charlie '18
```

Method 2: Define custom Compator and pass to Priority Queue constructor

Student.java

What if Object has *compareTo()* but you want a different order?

```
57          // Method 2:
58          // Use a custom Comparator.compare (length of name) instead
59          // of using the element's compareTo function
60          // Java will use this to compare two Students (here on length of name)
61          class NameLengthComparator implements Comparator<Student> {
62              public int compare(Student s1, Student s2) {
63                  return s1.name.length() - s2.name.length();
64              }
65          }
66          Comparator<Student> lenCompare = new NameLengthComparator();
67          pq = new PriorityQueue<Student>(lenCompare); //passing Comparator to PriorityQueue
68          pq.addAll(students); //add all students to PriorityQueue
69          System.out.println("\nlength:");
70          //remove until empty (sorting)
71          while (!pq.isEmpty()) System.out.println(pq.remove());
72
73          //Method 3:
74          // Use a custom Comparator via Java 8 anonymous function (here based on year)
75          // pass Comparator to PriorityQueue constructor
76          pq = new PriorityQueue<Student>((Student s1, Student s2) -> s2.year - s1.year);
77          pq.addAll(students); //add all students to Priority Queue
78          System.out.println("\nyear:");
79          //remove until empty (sorting)
80          while (!pq.isEmpty()) System.out.println(pq.remove());
81      }
82  }
83
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

<terminated> Student [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 10, 2018, 11:21:36 AM)

```
length:
bob '19
elvis '21
alice '20
denise '20
charlie '18
```

Output sorted by length of name

- **Still in *main()***
- **Define Comparator class that requires *compare()* method**
- ***compare()* has two *Student* params**
- **Here we use length of *name* to compare two *Student* Objects**
- ***compare()* returns negative, equal, or positive same as *compareTo()***
- **Instantiate new Comparator**
- **Create new Priority Queue and pass Comparator in constructor**
- **Then fill Priority Queue with students**
- **Sort by looping until Priority Queue empty**
- **Each time remove *Student* with smallest Key as determined by Comparator *instead of* *Student*'s *compareTo()***

If we use our own PriorityQueue, we need to provide way to compare objects

Student.java

Three ways to compare objects in Java's Priority Queue:

- Method 1: Objects stored in Priority Queue provide a *compareTo()* method

- Method 2: Instantiate a custom Comparator and pass to Priority Queue constructor

- Method 3: Use anonymous function in Priority Queue declaration

Method 3: Use anonymous function in Priority Queue declaration

Student.java

```
73          //Method 3:
74          // Use a custom Comparator via Java 8 anonymous function (here based on year)
75          // pass Comparator to PriorityQueue constructor
76          pq = new PriorityQueue<Student>((Student s1, Student s2) -> s2.year - s1.year);
77          pq.addAll(students); //add all students to Priority Queue
78          System.out.println("\nyear:");
79          //remove until empty (sorting)
80          while (!pq.isEmpty()) System.out.println(pq.remove());
81      }
82  }
83
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

<terminated> Student [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 10, 2018, 11:21:36 AM)

```
year:
elvis '21
denise '20
alice '20
bob '19
charlie '18
```

- **Anonymous functions don't have a name**
- **Declared "inline"**
- **Sometimes called "lambda function"**
- **Here compare Students based on *year***
- **Passed to Priority Queue constructor**
- **Students removed by anonymous function order (*year* in this case), not *compareTo()* order**

Method 3: Use anonymous function in Priority Queue declaration

Student.java

```
73          //Method 3:
74          // Use a custom Comparator via Java 8 anonymous function (here based on year)
75          // pass Comparator to PriorityQueue constructor
76          pq = new PriorityQueue<Student>((Student s1, Student s2) -> s2.year - s1.year);
77          pq.addAll(students); //add all students to Priority Queue
78          System.out.println("\nyear:");
79          //remove until empty (sorting)
80          while (!pq.isEmpty()) System.out.println(pq.remove());
81      }
82  }
83
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

<terminated> Student [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 10, 2018, 11:21:36 AM)

```
year:
elvis '21
denise '20
alice '20
bob '19
charlie '18
```

Created a *Max* Priority Queue by simply reversing compare

- **Anonymous functions don't have a name**
- **Declared "inline"**
- **Sometimes called "lambda function"**
- **Here compare Students based on *year***
- **Passed to Priority Queue constructor**
- **Students removed by anonymous function order (*year* in this case), not *compareTo()* order**

Output sorted by student year in *descending* order (reversed normal order of compared objects)

Agenda

1. Priority Queue ADT
2. Implementation choices
3. Java's built-in PriorityQueue
4. Reading from a file

Use a BufferedReader to read a file line by line until reaching the end of file

Roster.java

```
BufferedReader input = new BufferedReader(new FileReader(fileName));
String line;
int lineNum = 0;
while ((line = input.readLine()) != null) {
  System.out.println("read @"+lineNum+"`"+line+"'");
  lineNum++;
}
```

- *BufferedReader* opens file with name *filename*
- Reading will start at beginning of file
- Each line from file stored in *line* in while loop
- *input.readLine* will return null at end of file
- Here we are just printing each line

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- **Many possible exceptions reading data from a file:**
  - **File may not be found**
  - **Some data might be missing (e.g., name without a year)**
  - **Some data might be invalid (e.g., year is not a valid Integer)**

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- This method reads a comma separated variable (csv) file
- Each line should have student name and year
- Creates a Student Object from each line of the file
- Returns a List of Student Objects with one entry for each valid line
- File name to read is passed as String parameter

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- Create new BufferedReader
- Catch error if file not found

- This method reads a comma separated variable (csv) file
- Each line should have student name and year
- Creates a Student Object from each line of the file
- Returns a List of Student Objects with one entry for each valid line
- File name to read is passed as String parameter

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- This method reads a comma separated variable (csv) file
- Each line should have student name and year
- Creates a Student Object from each line of the file
- Returns a List of Student Objects with one entry for each valid line
- File name to read is passed as String parameter

- Create new BufferedReader
- Catch error if file not found

- Read each line of file, store in *line* String
- Split() on comma, make sure we got two parts (input could be invalid)

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- **Got two elements after *split()***
- **Try to parse as *name* as String and *year* as Integer**
- **Add to *roster* if valid student**

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- Got two elements after *split()*
- Try to parse as *name* as String and *year* as Integer
- Add to *roster* if valid student

- If second element not Integer:
  - Catch error
  - Print error message
  - Keep reading

When reading files, we need to be ready to handle many different exceptions

Roster.java

```
76    public static List<Student> readRoster2(String fileName) throws IOException {
77        List<Student> roster = new ArrayList<Student>();
78        BufferedReader input;
79
80        // Open the file, if possible
81        try {
82            input = new BufferedReader(new FileReader(fileName));
83        }
84        catch (FileNotFoundException e) {
85            System.err.println("Cannot open file.\n" + e.getMessage());
86            return roster;
87        }
88
89        // Read the file
90        try {
91            // Line by line
92            String line;
93            int lineNum = 0;
94            while ((line = input.readLine()) != null) {
95                System.out.println("read @"+lineNum+"`"+line+"'");
96                // Comma separated
97                String[] pieces = line.split(",");
98                if (pieces.length != 2) {
99                    //did not get two elements in this line, output an error message
100                   System.err.println("bad separation in line "+lineNum+":"+line);
101               }
102               else {
103                   // got two elements for this line
104                   try {
105                       // Extract year as an integer, if possible
106                       Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
107                       System.out.println("=>"+s);
108                       roster.add(s); //good student, add to roster
109                   }
110                   catch (NumberFormatException e) {
111                       // couldn't parse second element as integer
112                       System.err.println("bad number in line "+lineNum+":"+line);
113                   }
114               }
115               lineNum++;
116           }
117       }
```

- Got two elements after *split()*
- Try to parse as *name* as String and *year* as Integer
- Add to *roster* if valid student

- If second element not Integer:
  - Catch error
  - Print error message
  - Keep reading

Close file in *finally* block (not shown) – always runs