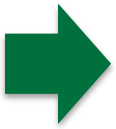
CS 10:
Problem solving via Object Oriented Programming

Prioritizing

Main goals

- Implement priority queue
- Implement priority queue more efficiently with heap

Agenda



1. Priority Queue ADT
2. Implementation choices
3. Java's built-in PriorityQueue

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 the traffic pattern is the first to land (FIFO)

Image: flickr

Sometimes higher priority issues arise and we need a different order

Airplanes queued to land

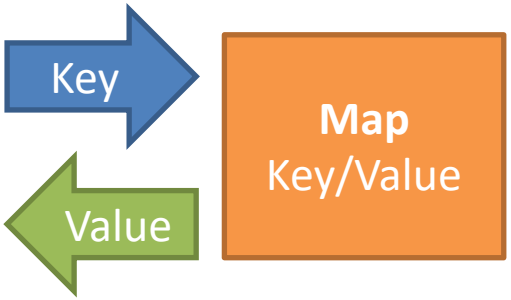
Image: flickr

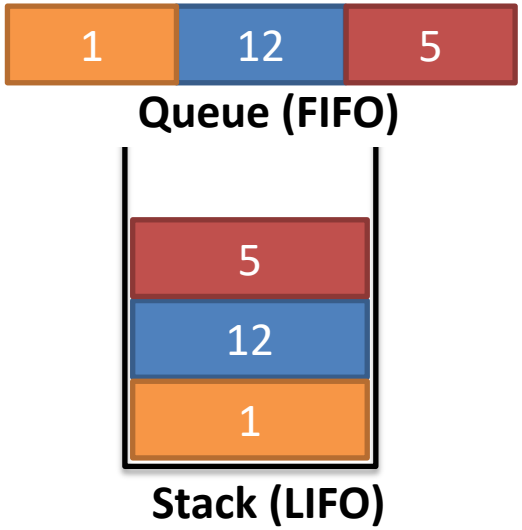
Suddenly one aircraft has an in-flight emergency and needs to land now!

Need a way to go to front of queue

Enter the priority queue

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





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 that 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

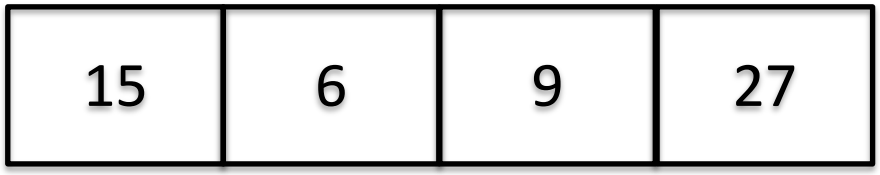
```
 6 public interface MinPriorityQueue<E extends Comparable<E>> {
 7     /**
 8      * Is the priority queue empty?
 9      * @return true if the priority queue is empty, false if not empty.
10      */
11     public boolean isEmpty();
12
13     /**
14      * Insert an element into the queue.
15      * @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 }
```

Agenda

1. Priority Queue ADT
2. Implementation choices
3. Java's built-in PriorityQueue

Could implement the Priority Queue using a sorted or unsorted List

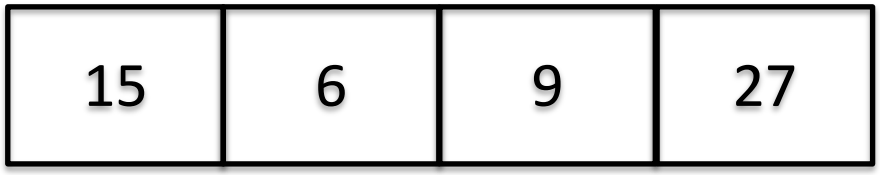
Unsorted List



Operation	**Run time**
`isEmpty`	
`insert`	
`minimum`	
`extractMin`	

Could implement the Priority Queue using a sorted or unsorted List

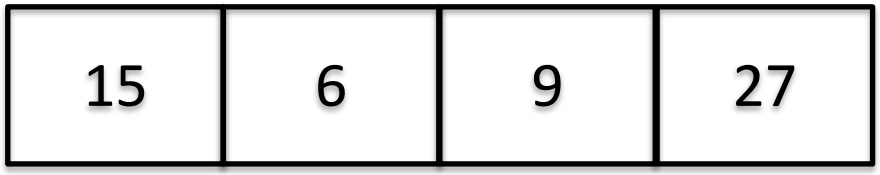
Unsorted List



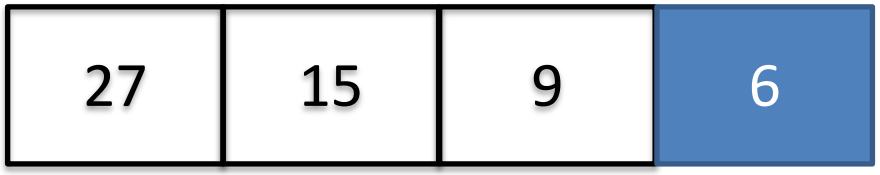
Operation	Run time	Notes
`isEmpty`	O(1)	Check size == 0
`insert`	O(1)	Add on to end (amortized growth)
`minimum`	Θ(n)	Must loop through all elements to find smallest
`extractMin`	Θ(n)	**Loop through all elements and move last item to fill hole**

Could implement the Priority Queue using a sorted or unsorted List

Unsorted List



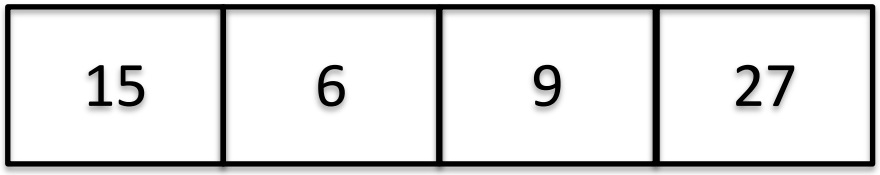
Sorted List



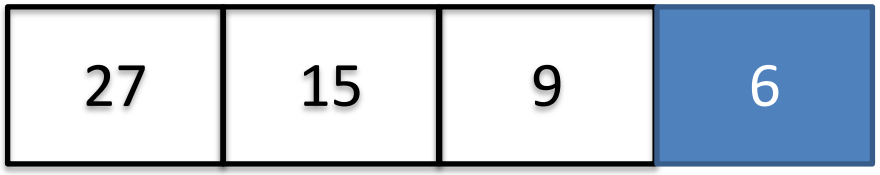
Operation	Unsorted	Sorted	Notes
`isEmpty`	O(1)		
`insert`	O(1)		
`minimum`	Θ(n)		
`extractMin`	Θ(n)		

Could implement the Priority Queue using a sorted or unsorted List

Unsorted List



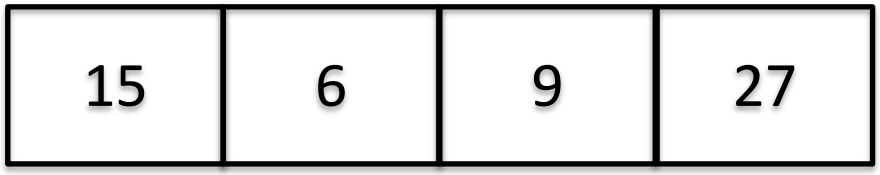
Sorted List



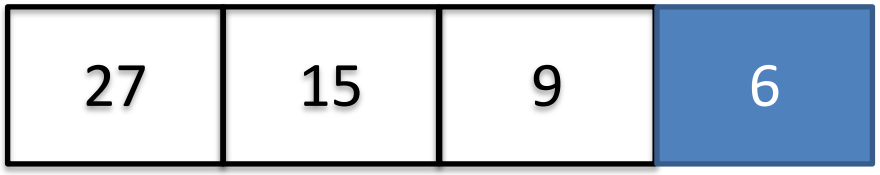
Operation	Unsorted	Sorted	Notes
`isEmpty`	O(1)	O(1)	Check size == 0
`insert`	O(1)	O(2n+1) = O(n)	Insert in order, move
`minimum`	Θ(n)	O(1)	Return last element
`extractMin`	Θ(n)	O(1)	Remove last element

Could implement the Priority Queue using a sorted or unsorted List

Unsorted List



Sorted List



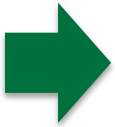
Operation	Unsorted	Sorted	Notes
`isEmpty`	O(1)	O(1)	Check size == 0
`insert`	O(1)	**O(n)**	Insert in order, move
`minimum`	**Θ(n)**	O(1)	Return last element
`extractMin`	**Θ(n)**	O(1)	Remove last element

Either way we pay a price, on min/extractMin or on insert
Heaps are a better choice

Agenda

1. Priority Queue ADT

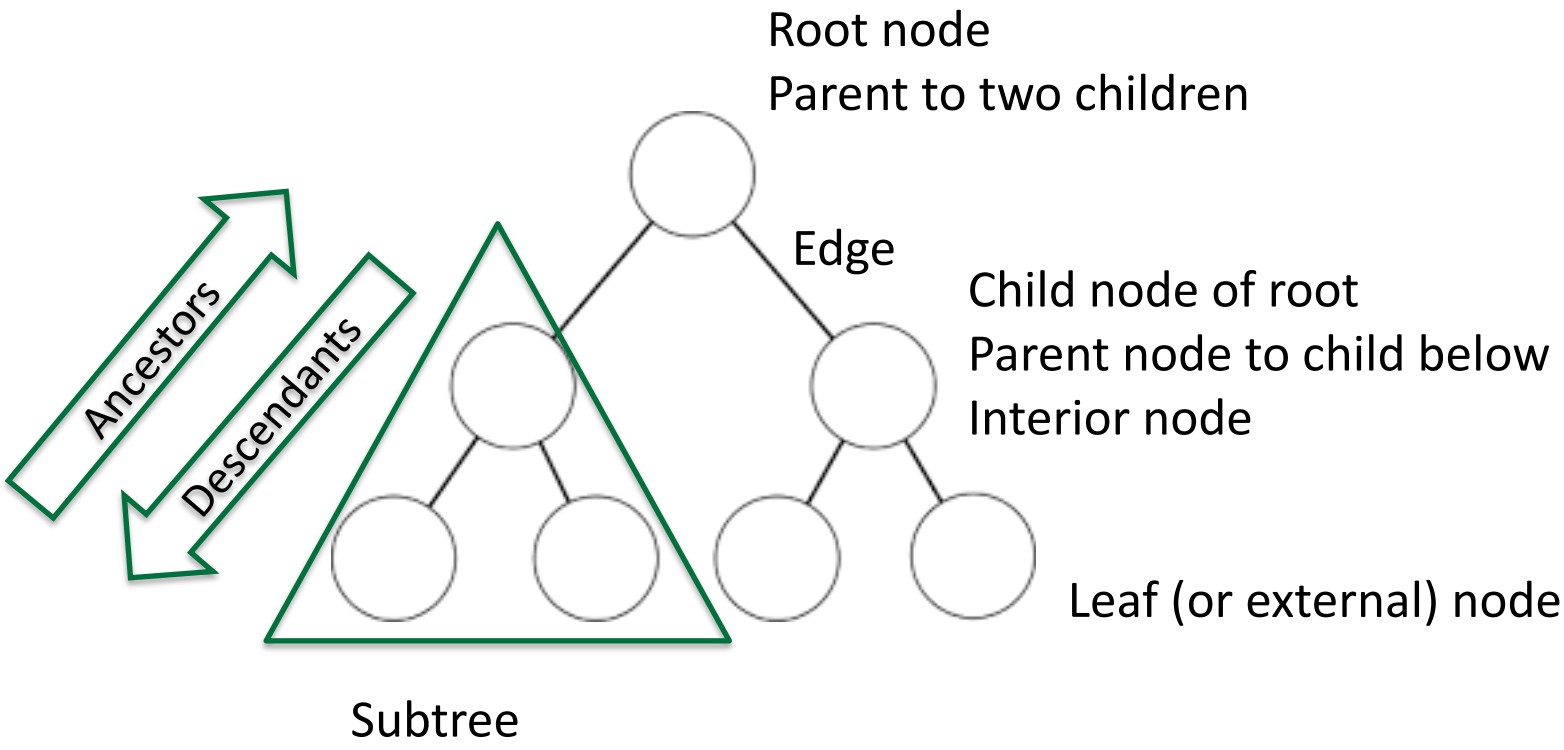
2. Implementation choices



3. Java's built-in PriorityQueue

Heaps are based on Binary Trees

Tree data structure



In a Binary Tree, each node has 0, 1, or 2 children

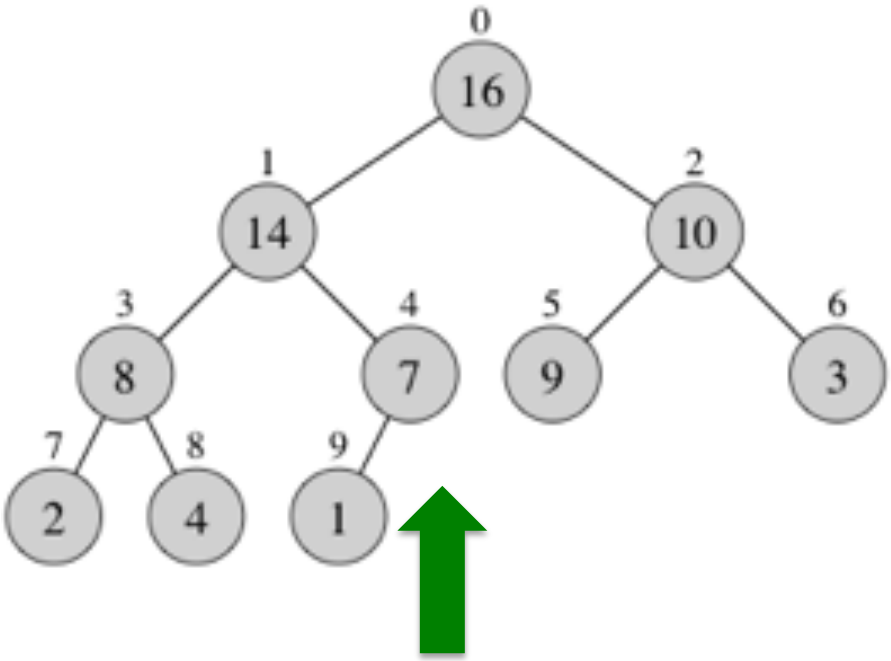
Height is the number of edges on the longest path from root to leaf

Each node has a Key and a Value

No guarantee of balance in Tree, could have Vine

Heaps have two additional properties beyond Binary Trees: Shape and Order

Shape property keeps tree compact



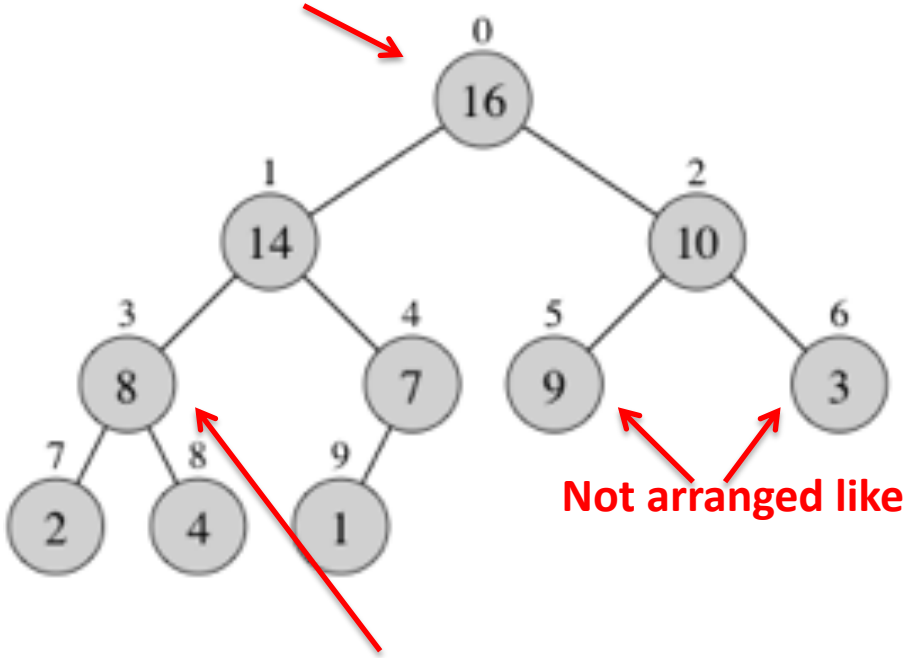
Next node added here

Shape property

- Nodes added starting from root and building downward
- New level started only once a prior level is filled
- Nodes added left to right
- Called a “complete” tree
- Prevents “vines”
- Makes height as small as possible: $h = \lfloor \log_2 n \rfloor$

Heaps have two additional properties beyond Binary Trees: Shape and Order

Order property keeps nodes organized



Order property

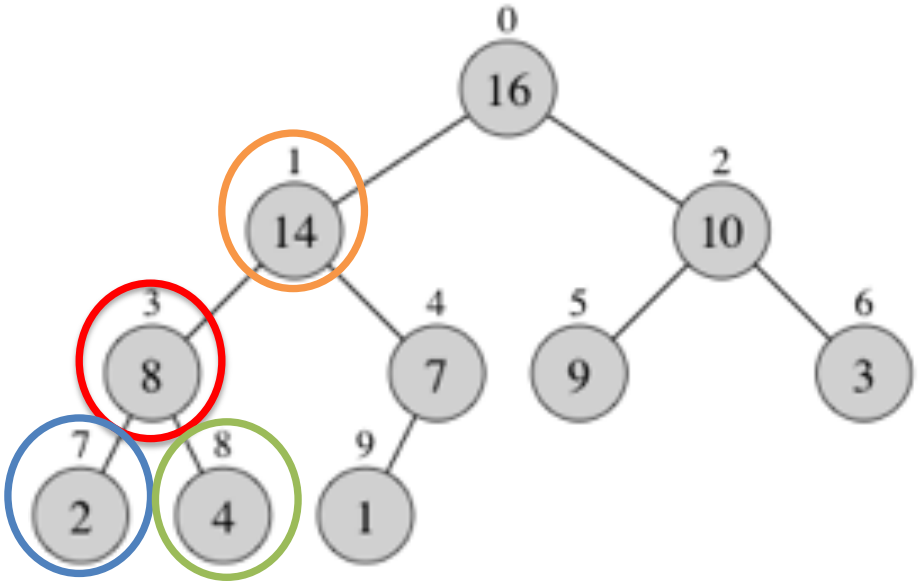
Reverse inequality for min heap

- ∀nodes *i* ≠ root, value(parent(i)) ≥ value(i)
- Root is the largest value in a max heap (or min value in a min heap)
- Largest value at any subtree is at the root of the subtree
- Unlike BST, no relationship between two sibling nodes, other than they are less than parent

The shape property makes an array a natural implementation choice

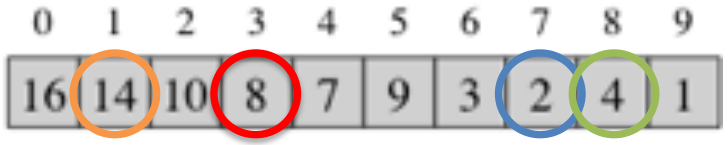
Array implementation

Heap is conceptually a tree, data actually stored in an array



Nodes stored in array

- Node *i* stored at index *i*
- Parent at index *(i-1)/2*
- Left child at index *i*2 +1*
- Right child at index *i*2+2*



Node 3 containing 8

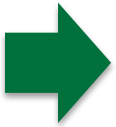
- *i=3*
- Parent = (3-1)/2= 1
- Left child = 3*2+1 = 7
- Right child = 3*2+2=8

Drop any decimal component

Agenda

1. Priority Queue ADT

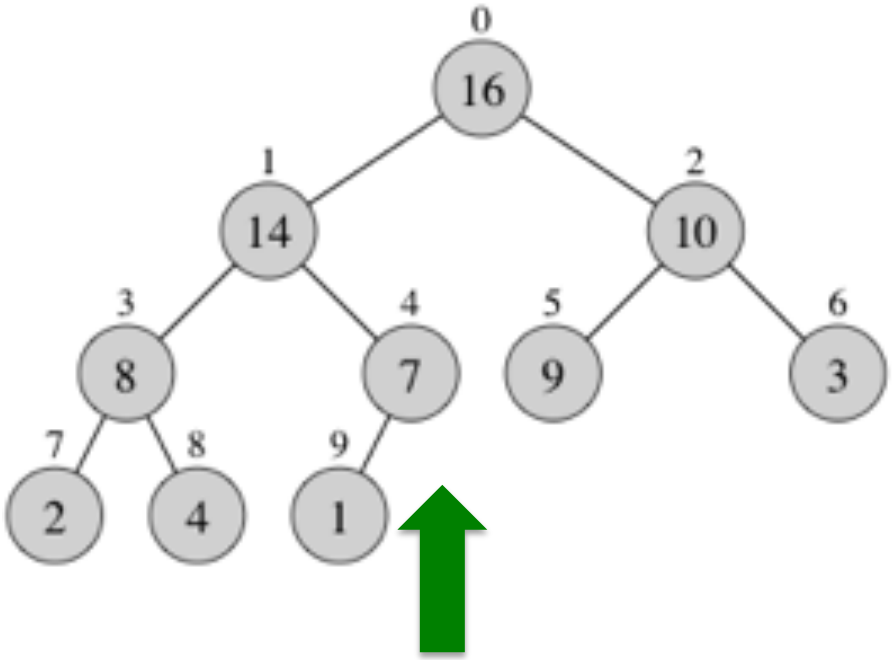
2. Implementation choices



3. Java's built-in PriorityQueue

Inserting into max heap must keep both shape and order properties intact

Max heap insert



Next node added here

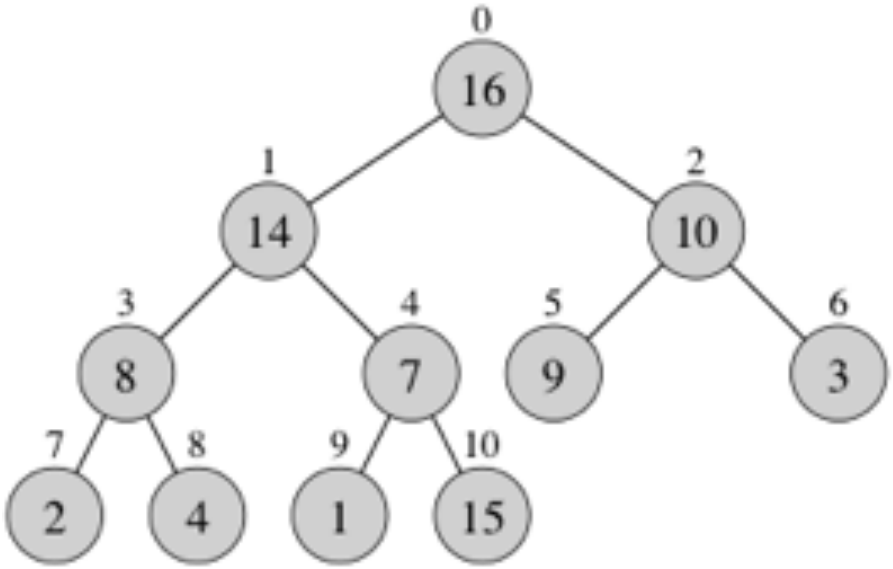
Insert 15

- Shape property: fill in next spot in left to right order (index i=10)

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

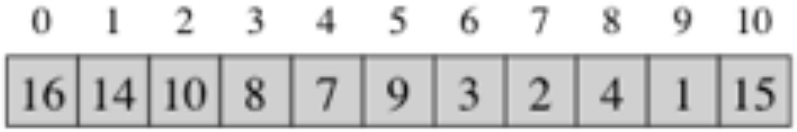
Inserting into max heap must keep both shape and order properties intact

Max heap insert



Insert 15

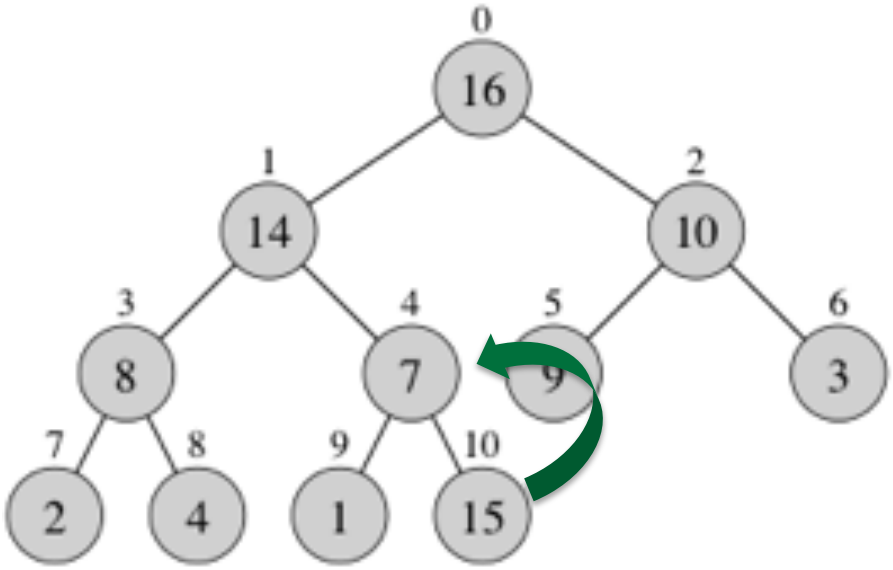
- Shape property: fill in next spot in left to right order (index i=10)



- Order property: parent must be larger than children
- Can’t keep 15 below 7
- Swap parent and child

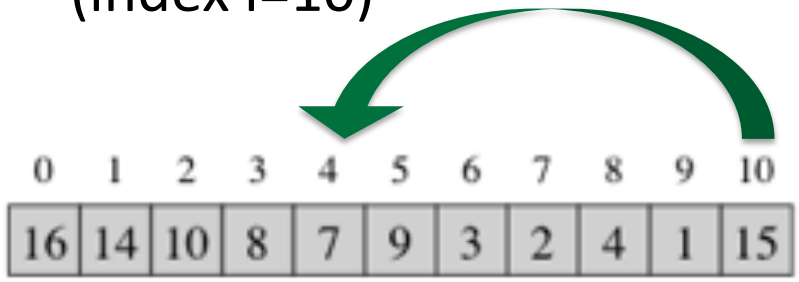
Inserting into max heap must keep both shape and order properties intact

Max heap insert



Insert 15

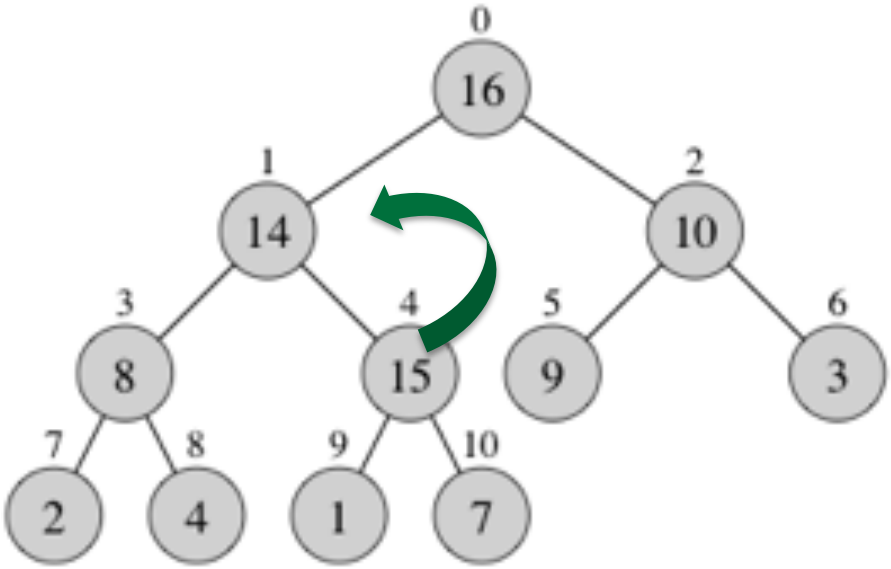
- Shape property: fill in next spot in left to right order (index i=10)



- Order property: parent must be larger than children
- Can't keep 15 below 7
- Swap parent and child
- Parent is at index (i-1)/2 = 4

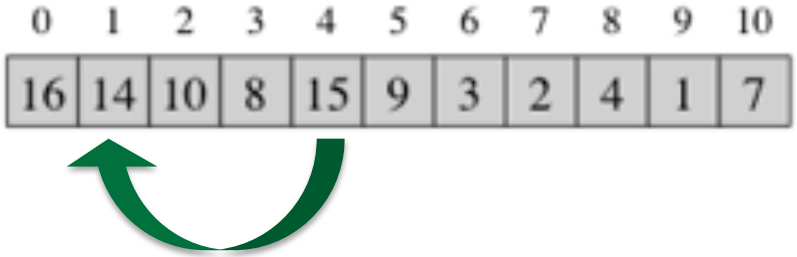
We may have to swap multiple times to get both heap properties

Max heap insert



Insert 15

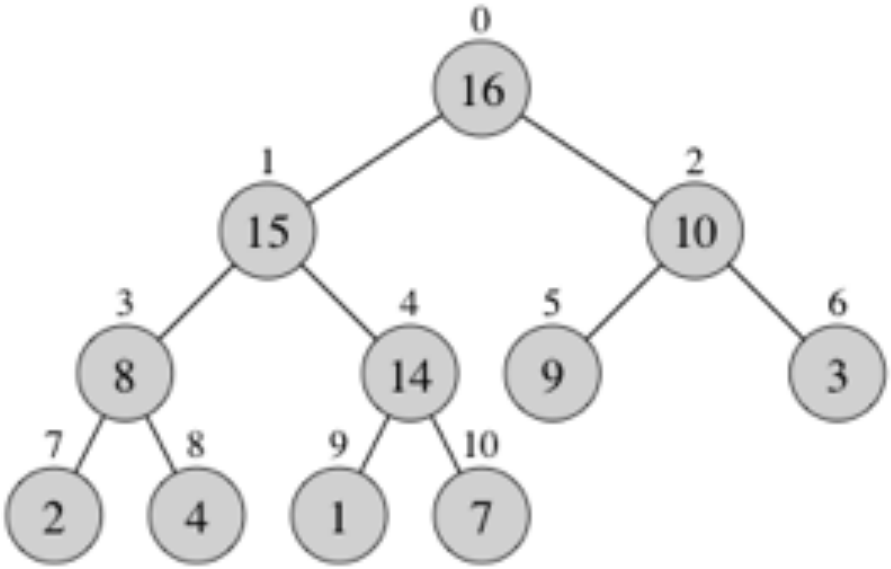
- Shape property: good!
- Order property: parent must be larger than children, not met



- Swap parent and child
- Child is at index *i=4*
- Parent at (i-1)/2=1

Eventually we will find a spot for the newly inserted item, even if that spot is the root

Max heap insert



Insert summary:

- **Add new node at bottom left of tree**
- **Bubble new node up (possibly to root) until order restored**
- **Tree will be as compact as possible**
- **Largest node at root**

Insert 15

- Shape property: good!
- Order property: good!
- Done

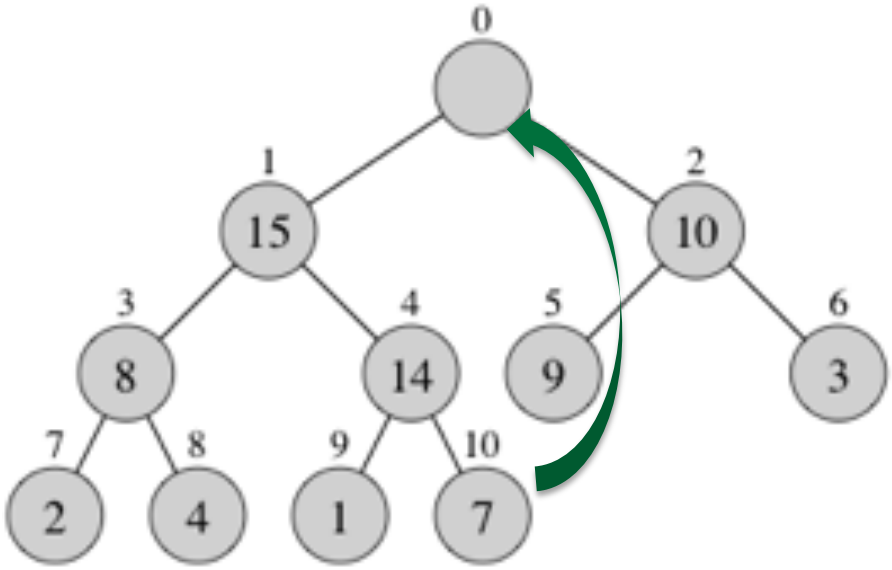
0	1	2	3	4	5	6	7	8	9	10
16	15	10	8	14	9	3	2	4	1	7

General rule

- Keep swapping until order property holds again
- Here done after swapping 14 and 15

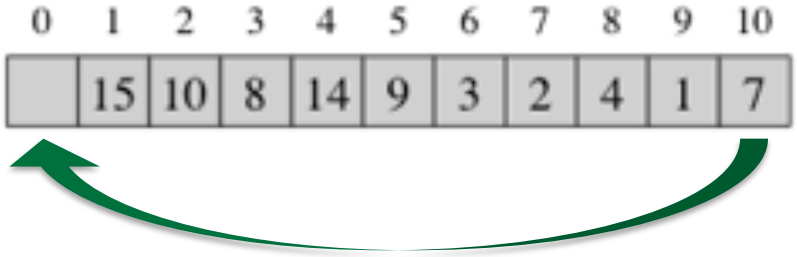
extractMax means removing the root, but that leaves a hole

extractMax



extractMax -> 16

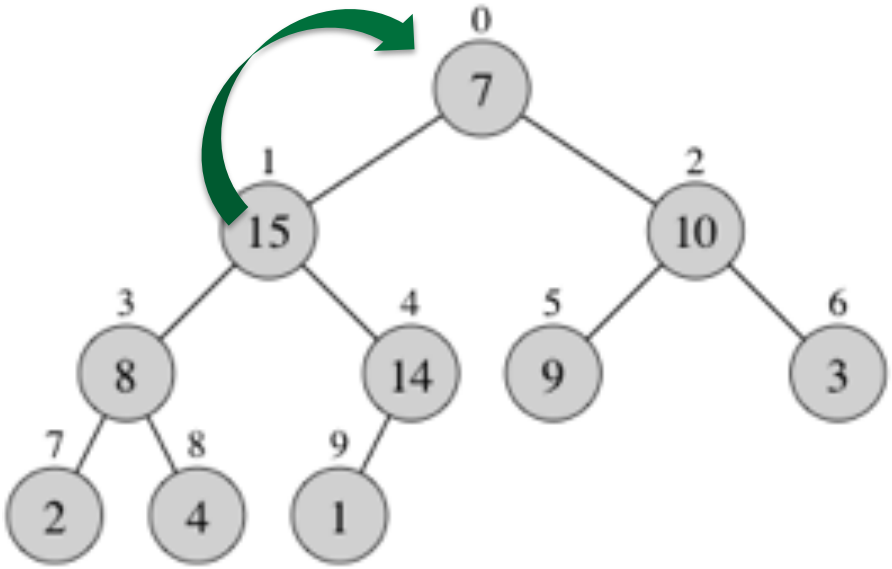
- Max position is at root (index 0)
- Removing it leaves a hole, violating shape property



- Also, bottom right most node must be removed to maintain shape property
- Solution: move bottom right node to root (like unsorted)

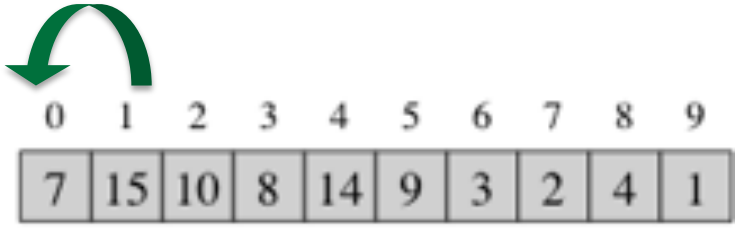
Moving bottom right node to root restores shape, but not order property

extractMax



After swap

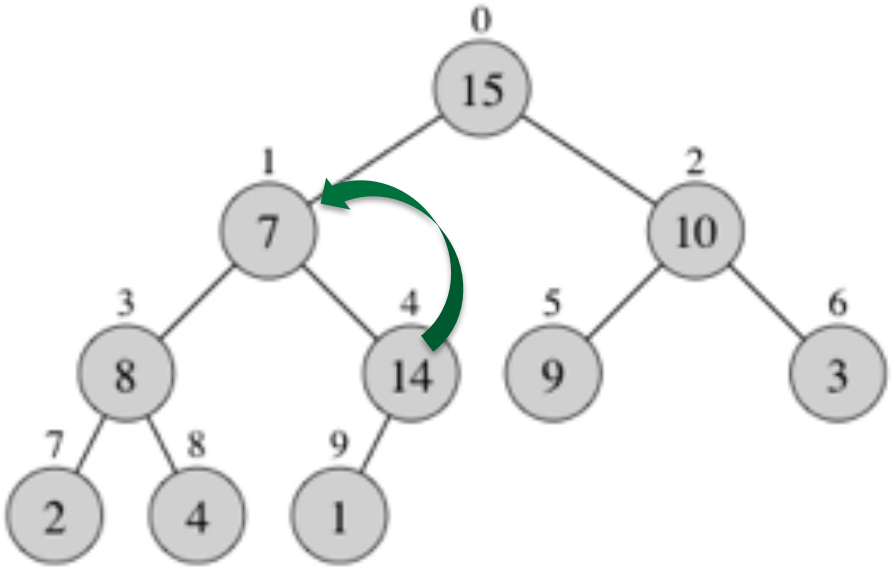
- Shape property: good!
- Order property: want max at root, but do not have that



- Left and right subtrees are still valid
- Swap root with larger child
- New root will be greater than everything in each subtree

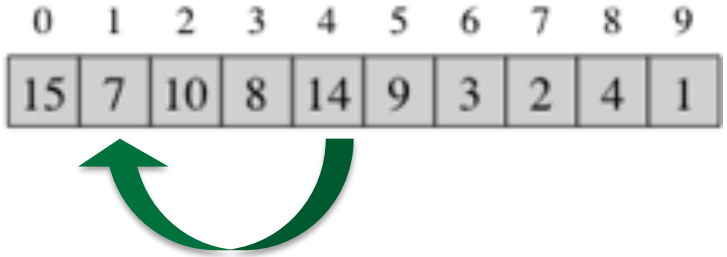
May need multiple swaps to restore order property

extractMax



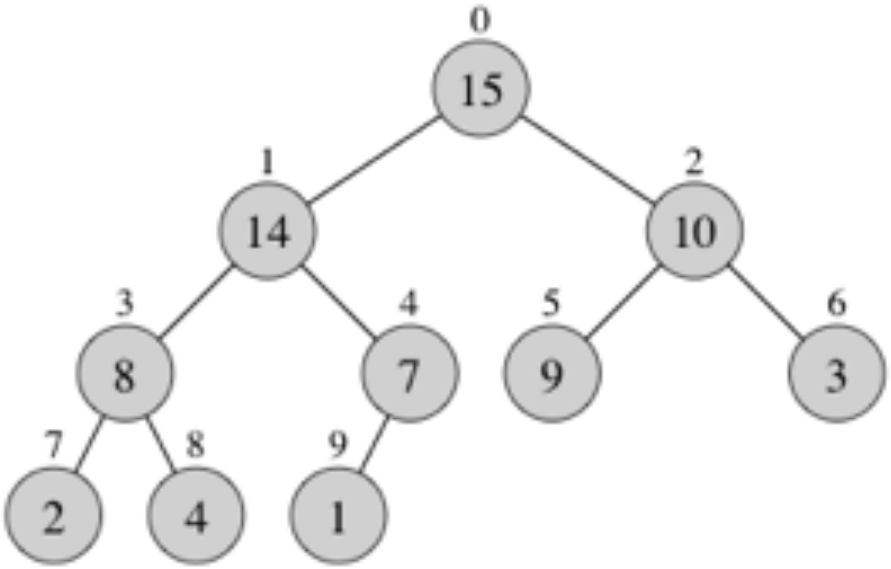
After swap 15 and 7

- Shape property: good!
- Order property: invalid
- Swap node with largest child



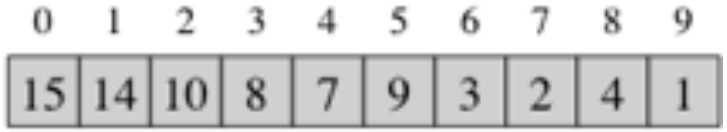
Stop once order property is restored

extractMax



After swap 7 and 14

- Shape property: good!
- Order property: good!



extractMax summary:

- **Remove root**
- **Move last node to root**
- **Bubble new root down by repeatedly swapping with largest child until order is restored**

Can implement heap-based Min Priority Queue using an ArrayList

HeapMinPriorityQueue.java

```
 9 public class HeapMinPriorityQueue<E extends Comparable<E>>
10 implements MinPriorityQueue<E> {
11     private ArrayList<E> heap;
12
13     /**
14      *  Constructor
15      */
16     public HeapMinPriorityQueue() {
17         heap = new ArrayList<E>();
18     }
19
```

NOTE: example was for a MAX Priority Queue, this code implements a MIN Priority Queue

Easy to change to this code to a MAX Priority Queue

Helper functions make finding parent and children easy

HeapMinPriorityQueue.java

```
108     // Swap two locations i and j in ArrayList a.
109     private static <E> void swap(ArrayList<E> a, int i, int j) {
110         E temp = a.get(i); //temporarily hold item at index i
111         a.set(i, a.get(j)); //set item at index i to item at index j
112         a.set(j, temp); //set item at index j to temp
113     }
114
115     // Return the index of the left child of node i.
116     private static int leftChild(int i) {
117         return 2*i + 1;
118     }
119
120     // Return the index of the right child of node i.
121     private static int rightChild(int i) {
122         return 2*i + 2;
123     }
124
125     // Return the index of the parent of node i
126     // (Parent of root will be -1)
127     private static int parent(int i) {
128         return (i-1)/2;
129     }
```

insert() adds a new item to the end and swaps with parent if needed

HeapMinPriorityQueue.java

```
41    public void insert(E element) {
42        heap.add(element);         // Put new value at end;
43        int loc = heap.size()-1;   // and get its location
44
45        // Swap with parent until parent not larger
46        while (loc > 0 && heap.get(loc).compareTo(heap.get(parent(loc))) < 0) {
47            swap(heap, loc, parent(loc));
48            loc = parent(loc);
49        }
50    }
```

extractMin() gets the root at index 0, moves last to root, and "re-heapifies"

HeapMinPriorityQueue.java

```
24⊖     public E extractMin() {
25          if (heap.size() <= 0)
26              return null;
27          else {
28              E minVal = heap.get(0); //min will be at node 0
29              heap.set(0, heap.get(heap.size()-1));  // Move last to position 0
30              heap.remove(heap.size()-1); //remove last item to maintain shape prope
31              minHeapify(heap, 0); //recursively swap to maintain order property
32              return minVal; //return min value
33          }
34      }
```

minHeapify() recursively enforces Shape and Order Properties

HeapMinPriorityQueue.java

```
79      private static <E extends Comparable<E>> void
80      minHeapify(ArrayList<E> a, int i) {
81          int left = leftChild(i);    // index of node i's left child
82          int right = rightChild(i);  // index of node i's right child
83          int smallest;    // will hold the index of the node with the smallest eleme
84          // among node i, left, and right
85
86          // Is there a left child and, if so, does the left child have an
87          // element smaller than node i?
88          if (left <= a.size()-1 && a.get(left).compareTo(a.get(i)) < 0)
89              smallest = left;   // yes, so the left child is the largest so far
90          else
91              smallest = i;      // no, so node i is the smallest so far
92
93          // Is there a right child and, if so, does the right child have an
94          // element smaller than the larger of node i and the left child?
95          if (right <= a.size()-1 && a.get(right).compareTo(a.get(smallest)) < 0)
96              smallest = right;  // yes, so the right child is the largest
97
98          // If node i holds an element smaller than both the left and right
99          // children, then the min-heap property already held, and we need do
100         // nothing more.  Otherwise, we need to swap node i with the larger
101         // of the two children, and then recurse down the heap from the larger chi
102         if (smallest != i) {
103             swap(a, i, smallest); //put smallest in spot i, largest in spot smalle
104             minHeapify(a, smallest); //maintain heap starting from smallest index
105         }
106     }
```

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	Θ(1)	Θ(1)	Θ(1)

isEmpty()

- Each implement just checks size of ArrayList; Θ(1)

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	Θ(1)	Θ(1)	Θ(1)
insert	**$O(\log_2 n)$**	Θ(1)	**O(n)**

insert()

- **Heap**: insert at end Θ(1), then may have to bubble up height of tree; $O(\log_2 n)$
- **Unsorted ArrayList:** just add on end of ArrayList; Θ(1)
- **Sorted ArrayList:** have to find place to insert O(n), then do insert, moving all other items; O(n)

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	Θ(1)	Θ(1)	Θ(1)
insert	**$O(\log_2 n)$**	Θ(1)	**O(n)**
minimum	**Θ(1)**	**Θ(n)**	Θ(1)

minimum()

- **Heap**: return item at index 0 in ArrayList; Θ(1)
- **Unsorted ArrayList:** search Arraylist; Θ(n)
- **Sorted ArrayList:** return last item in ArrayList; Θ(1)

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	Θ(1)	Θ(1)	Θ(1)
insert	**$O(\log_2 n)$**	Θ(1)	**O(n)**
minimum	**Θ(1)**	**Θ(n)**	Θ(1)
extractMin	**$O(\log_2 n)$**	**Θ(n)**	Θ(1)

extractMin()

- **Heap**: return item at index 0, then replace with last item, then bubble down height of tree; $O(\log_2 n)$
- **Unsorted ArrayList:** search Arraylist, Θ(n), remove, then move all other items; O(n)
- **Sorted ArrayList:** return last item in ArrayList; Θ(1)

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	Θ(1)	Θ(1)	Θ(1)
insert	**$O(\log_2 n)$**	Θ(1)	**O(n)**
minimum	**Θ(1)**	**Θ(n)**	Θ(1)
extractMin	**$O(\log_2 n)$**	**Θ(n)**	Θ(1)

With Unsorted ArrayList or Sorted ArrayList, can't escape paying O(n) (either insert or extractMin)

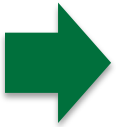
Heap must pay $O(\log_2 n)$, but that is much better than O(n) when n is large

Remember $O(\log_2 n)$ where n = 1 million is 20 (one billion is 30)

Agenda

1. Priority Queue ADT

2. Implementation choices



3. Java's built-in PriorityQueue

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     }
```

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 10, 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
```

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 Comparator 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
74
75
76
77
78
79
80
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

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

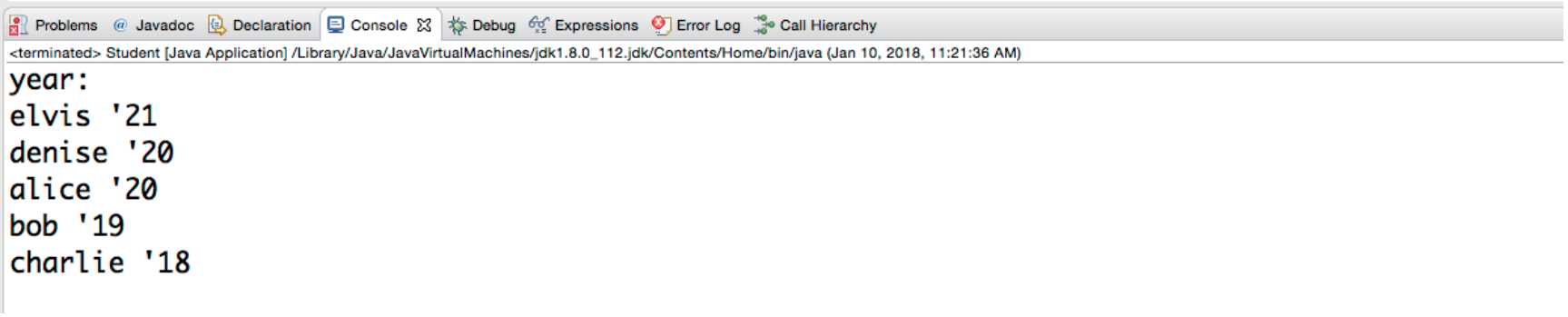
- Method 2: Instantiate a custom Comparator and pass to Priority Queue constructor

- Method 3: Use anonymous function in Priority Queue declaration

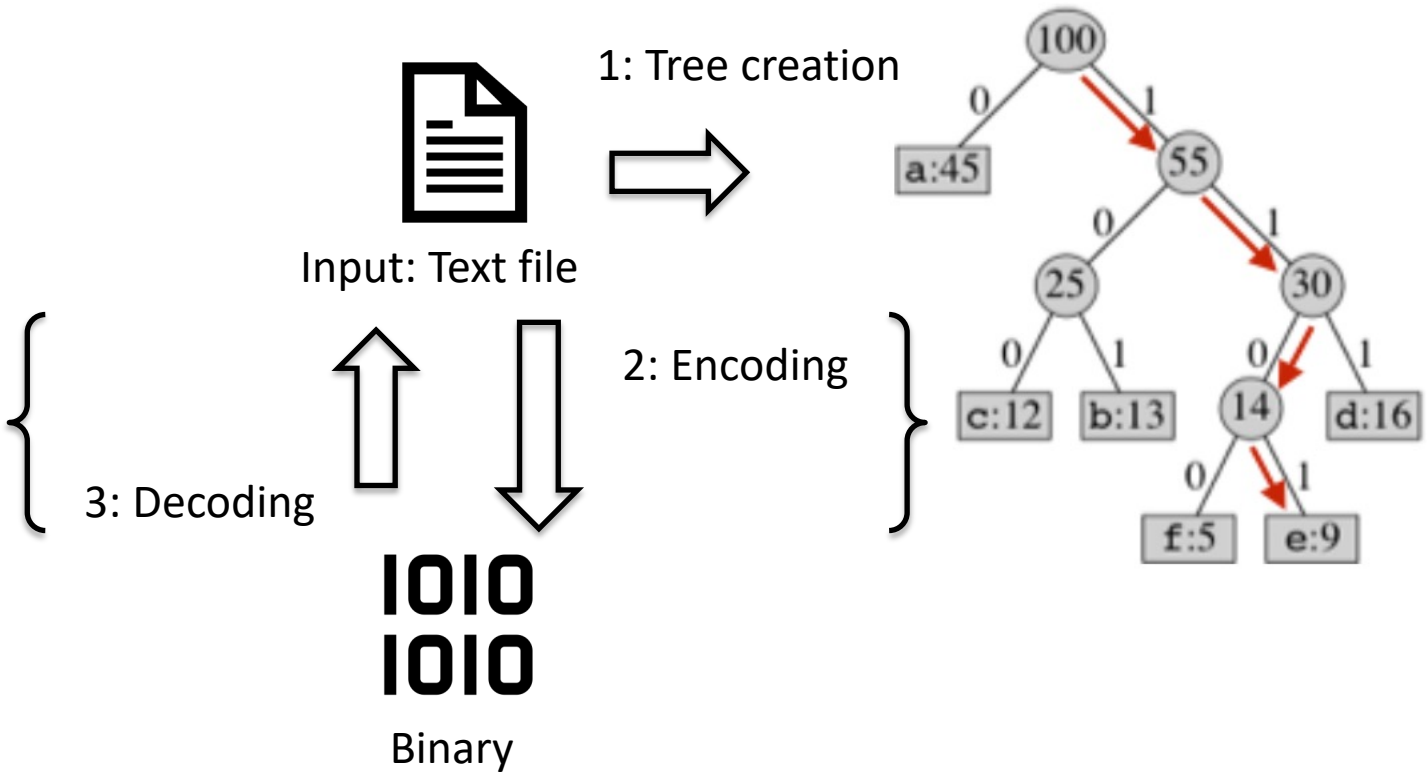
Method 3: Use anonymous function in Priority Queue declaration

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  }
```



PS-3



https://www.cs.dartmouth.edu/cs10/PS-3.html

Summary

- Priority queue have elements returned according to value (min or max)
  - Can implement with unsorted array and sorted array, each with different complexities
- Heaps are based on binary trees and have two main properties
  - Shape
  - Order

- Priority queue implemented with heap can be very efficient

- To use priority queues, objects need to have way to compare with each other
  - Three methods possible

Additional Resources

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, Θ(1) • *extractMin()* slow – search entire List for min Key, Θ(n)

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

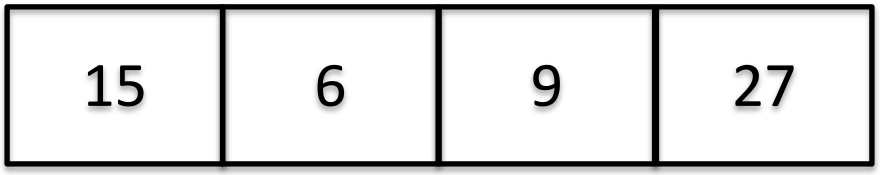
Choice	Fit	Notes
Stack/Queue		• Elements ordered by arrival time • Can only access one element (top or front) • Element with higher priority that arrives out of sequence can not be reached
Map		• Have to know the Key in order to find item • In scheduling example, would have to check each minute 0 through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

We can implement a PriorityQueue with an unsorted ArrayList

Unsorted ArrayList implementation

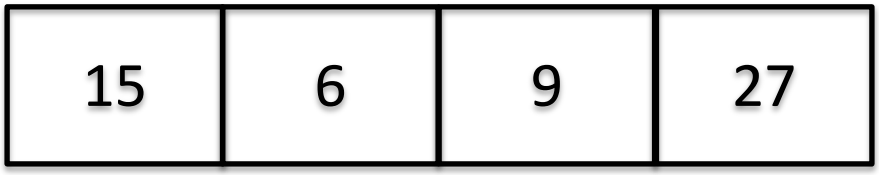


Keep elements unsorted in ArrayList

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

Unsorted ArrayList implementation

isEmpty()



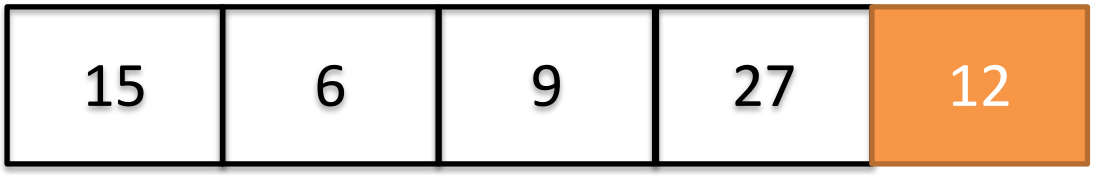
isEmpty – just check ArrayList *size()* method

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

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

Unsorted ArrayList implementation

insert(12)



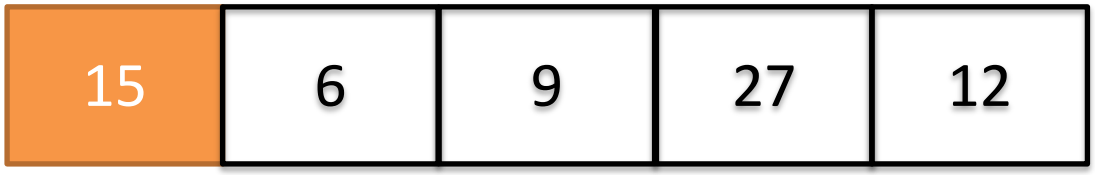
insert – just add element to end of ArrayList

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

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

Unsorted ArrayList implementation

extractMin()
Check 15



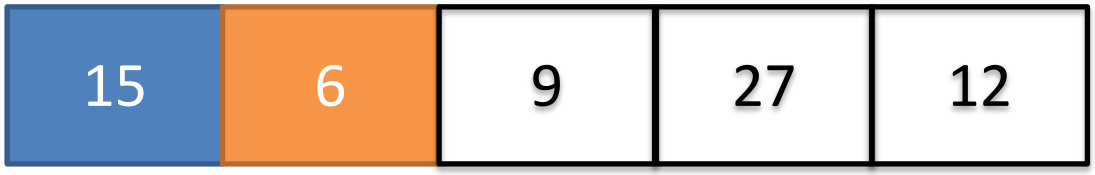
extractMin – loop to find smallest and move last item to smallest index to fill hole

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

minimum() and *extractMin()* are both Θ(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`	Θ(1)	Checks size == 0
`insert`	Θ(1)	Add on to end (amortized)
`minimum`	**Θ(n)**	**Must loop through all elements to find smallest**
`extractMin`	**Θ(n)**	**Loop through all elements and move to fill hole**

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

Unsorted ArrayList implementation

extractMin()
Check 9
Smallest 6

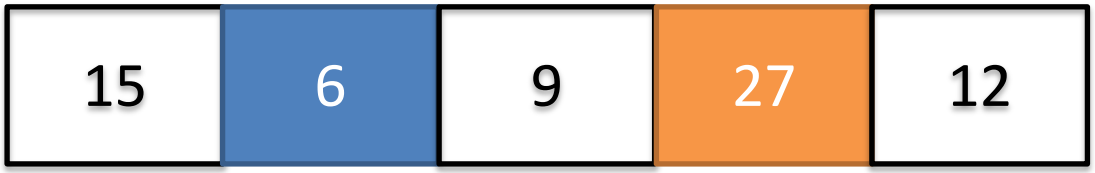
extractMin – loop to find smallest and move last item to smallest index to fill hole

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

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

Unsorted ArrayList implementation

extractMin()
Check 27
Smallest 6



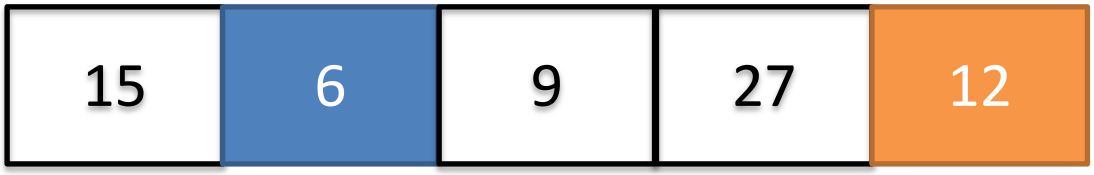
extractMin – loop to find smallest and move last item to smallest index to fill hole

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

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

Unsorted ArrayList implementation

extractMin()
Check 12
Smallest 6



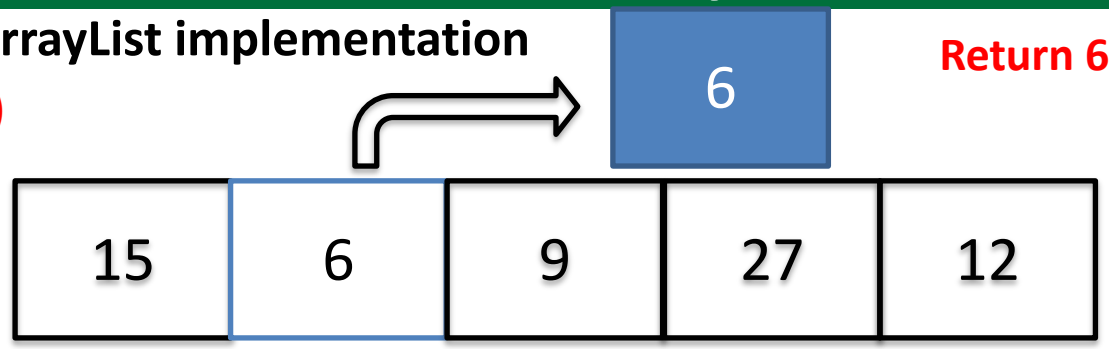
extractMin – loop to find smallest and move last item to smallest index to fill hole

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

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

Unsorted ArrayList implementation

extractMin()



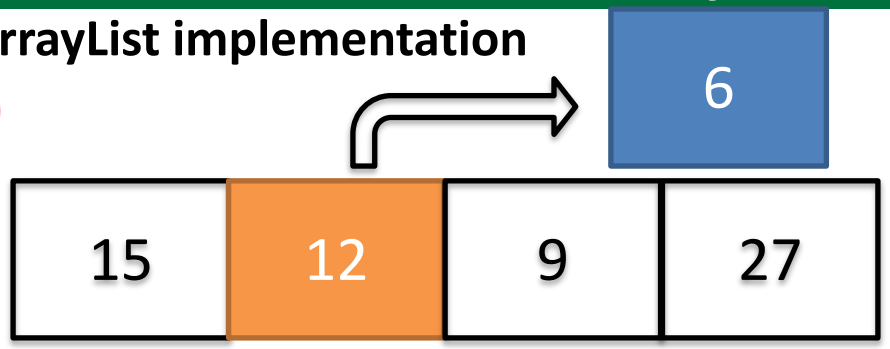
extractMin – loop to find smallest and move last item to smallest index to fill hole

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

minimum() and *extractMin()* are both Θ(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`	Θ(1)	Checks size == 0
`insert`	Θ(1)	Add on to end (amortized)
`minimum`	**Θ(n)**	**Must loop through all elements to find smallest**
`extractMin`	**Θ(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     }
```

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

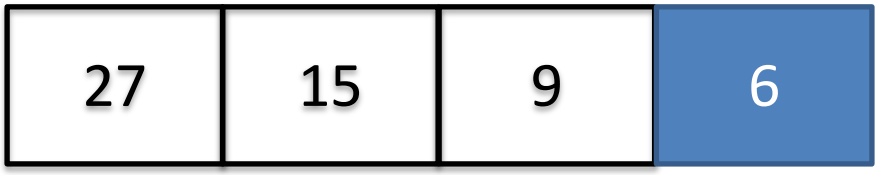
```
40    private int indexOfMinimum() {
41        // Search through the entire array for the smallest element.
42        int smallestIndex = 0;
43
44        for (int i = 1; i < list.size(); i++) {
45            // If the current smallest is greater than the element at index i,
46            // then 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    }
```

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

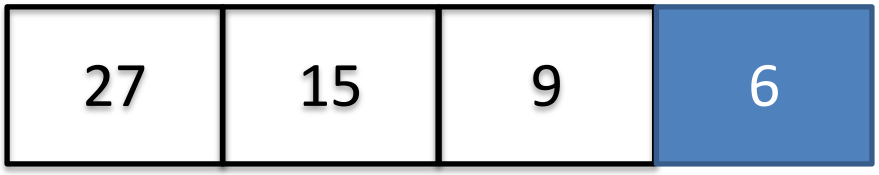


Keep elements sorted in ArrayList with smallest always at end

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

Sorted ArrayList implementation

isEmpty()



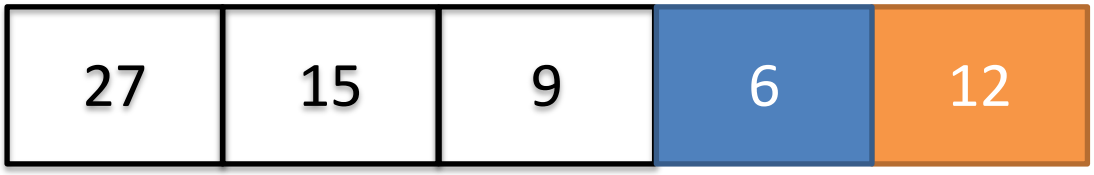
isEmpty() – just check ArrayList *size()* method

Operation	Run time	Notes
`isEmpty`	**Θ(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`	Θ(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)

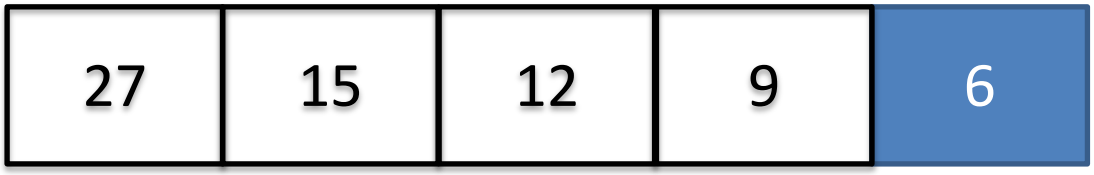
insert() – need to loop backward to find slot for new element, then move other elements right

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

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

Sorted ArrayList implementation

extractMin()



extractMin() – just remove the last element

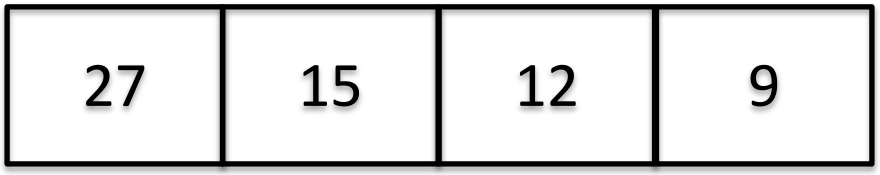
Operation	Run time	Notes
`isEmpty`	Θ(1)	Return size, same as unsorted
`insert`	O(n)	Insert in place and move other items right
`minimum`	**Θ(1)**	**Get last element**
`extractMin`	**Θ(1)**	**Get last element, no need to move items**

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

Sorted ArrayList implementation

extractMin()

Return 6



extractMin() – just remove the last element

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

Implementations have different strengths, but limited practical difference

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

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

MinPriorityQueue.java

ANNOTATED SLIDES

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

ArrayListMinPriorityQueue.java

ANNOTATED SLIDES

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    }
```

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

SortedArrayListMinPriorityQueue.java

ANNOTATED SLIDES

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

HeapMinPriorityQueue.java

ANNOTATED SLIDES

Can implement heap-based Min Priority Queue using an ArrayList

HeapMinPriorityQueue.java

Heap elements extend Comparable

```
 9 public class HeapMinPriorityQueue<E extends Comparable<E>>
10 implements MinPriorityQueue<E> {
11     private ArrayList<E> heap;
12
13     /**
14      *  Constructor
15      */
16     public HeapMinPriorityQueue() {
17         heap = new ArrayList<E>();
18     }
19
```

ArrayList called *heap* will hold the heap

NOTE: example was for a MAX Priority Queue, this code implements a MIN Priority Queue

Easy to change to this code to a MAX Priority Queue

Helper functions make finding parent and children easy

HeapMinPriorityQueue.java

Helper functions
swap()* trades node at index *i* for node at index *j

```
108     // Swap two locations i and j in ArrayList a.
109     private static <E> void swap(ArrayList<E> a, int i, int j) {
110         E temp = a.get(i); //temporarily hold item at index i
111         a.set(i, a.get(j)); //set item at index i to item at index j
112         a.set(j, temp); //set item at index j to temp
113     }
114
115     // Return the index of the left child of node i.
116     private static int leftChild(int i) {
117         return 2*i + 1;
118     }
119
120     // Return the index of the right child of node i.
121     private static int rightChild(int i) {
122         return 2*i + 2;
123     }
124
125     // Return the index of the parent of node i
126     // (Parent of root will be -1)
127     private static int parent(int i) {
128         return (i-1)/2;
129     }
```

leftChild(), rightChild()* and *parent()
calculate positions of nodes relative to *i*

insert() adds a new item to the end and swaps with parent if needed

HeapMinPriorityQueue.java

- **Add element to end of *heap***
- **Start at newly added item's index**

```
41    public void insert(E element) {
42        heap.add(element);          // Put new value at end;
43        int loc = heap.size()-1;    // and get its location
44
45        // Swap with parent until parent not larger
46        while (loc > 0 && heap.get(loc).compareTo(heap.get(parent(loc))) < 0) {
47            swap(heap, loc, parent(loc));
48            loc = parent(loc);
49        }
50    }
```

insert() adds a new item to the end and repeatedly swaps with parent if needed

HeapMinPriorityQueue.java

```
41    public void insert(E element) {
42        heap.add(element);          // Put new value at end;
43        int loc = heap.size()-1;    // and get its location
44
45        // Swap with parent until parent not larger
46        while (loc > 0 && heap.get(loc).compareTo(heap.get(parent(loc))) < 0) {
47            swap(heap, loc, parent(loc));
48            loc = parent(loc);
49        }
50    }
```

- **Add element to end of heap**
- **Start at newly added item's index**

NOTE: reverse *compareTo* inequality to implement a MAX Priority Queue

- **Swap if not root (loc==0) and element < parent**
- **Continue to "bubble up" inserted node until reach root or element > parent**
- **At most *O(h)* swaps (if new node goes all the way up to root)**
- **Due to Shape Property, max *h* is $log_2 n$, so $O(log_2 n)$**

extractMin() gets the root at index 0, moves last to root, and "re-heapifies"

HeapMinPriorityQueue.java

```
24    public E extractMin() {
25        if (heap.size() <= 0)
26            return null;
27        else {
28            E minVal = heap.get(0); //min will be at node 0
29            heap.set(0, heap.get(heap.size()-1));  // Move last to position 0
30            heap.remove(heap.size()-1); //remove last item to maintain shape prope
31            minHeapify(heap, 0); //recursively swap to maintain order property
32            return minVal; //return min value
33        }
34    }
```

- **Where will smallest element be?**
- **Always at the root (index 0)**

- **Move last item into root node to satisfy Shape Property**

- **Update heap so that it satisfies Order Property**
- **May have to "bubble down" the new root down to leaf level**
- **At most $O(h) = O(\log_2 n)$ operations**

minHeapify() recursively enforces Shape and Order Properties

HeapMinPriorityQueue.java

```
79      private static <E extends Comparable<E>> void
80      minHeapify(ArrayList<E> a, int i) {
81          int left = leftChild(i);     // index of node i's left child
82          int right = rightChild(i);  // index of node i's right child
83          int smallest;     // will hold the index of the node with the smallest eleme
84          // among node i, left, and right
85
86          // Is there a left child and, if so, does the left child have an
87          // element smaller than node i?
88          if (left <= a.size()-1 && a.get(left).compareTo(a.get(i)) < 0)
89              smallest = left;   // yes, so the left child is the largest so far
90          else
91              smallest = i;      // no, so node i is the smallest so far
92
93          // Is there a right child and, if so, does the right child have an
94          // element smaller than the larger of node i and the left child?
95          if (right <= a.size()-1 && a.get(right).compareTo(a.get(smallest)) < 0)
96              smallest = right;  // yes, so the right child is the largest
97
98          // If node i holds an element smaller than both the left and right
99          // children, then the min-heap property already held, and we need do
100         // nothing more.  Otherwise, we need to swap node i with the larger
101         // of the two children, and then recurse down the heap from the larger chi
102         if (smallest != i) {
103             swap(a, i, smallest); //put smallest in spot i, largest in spot smalle
104             minHeapify(a, smallest); //maintain heap starting from smallest index
105         }
106     }
```

a = heap, i = starting index

Get left and right children

- Find the smallest node between the current node, and the (possibly) two children
- Track smallest index in *smallest* variable

- If starting index is not the smallest, then swap node at starting index with smallest node
- Bubble down node from *smallest* index

At most $O(h) = O(\log_2 n)$ operations

Heap Sort

SUPPLEMENTAL MATERIAL

Using a heap, we can sort items “in place” in a two-stage process

Heap sort

Given array in unknown order

1. Build max heap in place using array given
   - Start with last non-leaf node, max heapify node and children
   - Move to next to last non-leaf node, max heapify again
   - Repeat until at root
   - NOTE: heap is not necessarily sorted, only know for sure that parent > children and max is at root

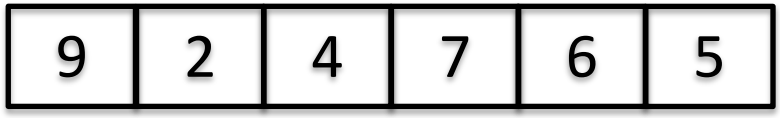
2. Extract max (index 0) and swap with item at end of array, then rebuild heap not considering last item

Does not require additional memory to sort

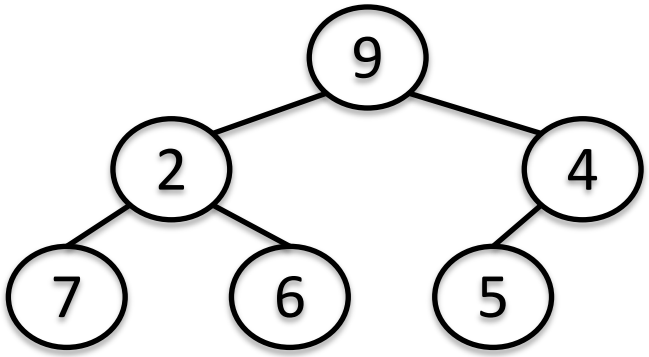
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Given array in unsorted order

First build a heap in place

- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

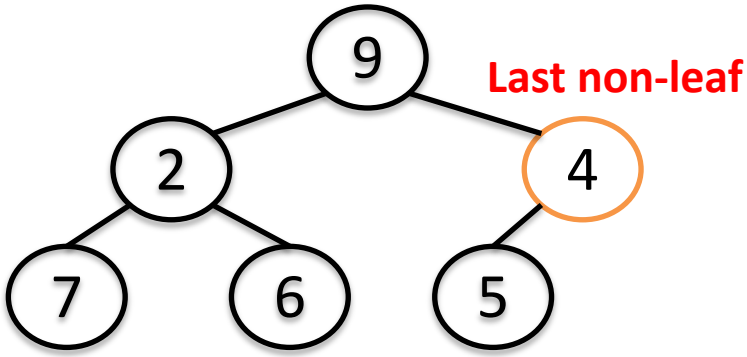
Non heap!

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Last non-leaf will be parent of last leaf

Given array in unsorted order

First build a heap in place

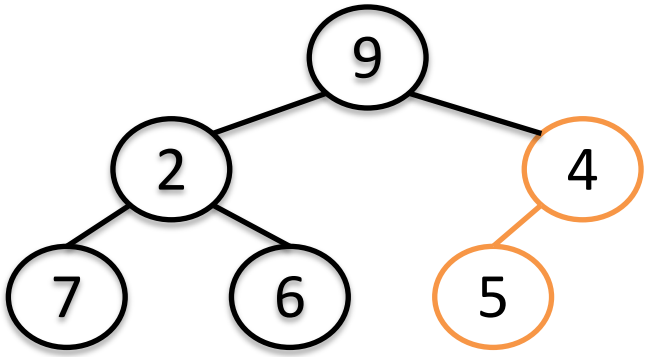
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Max heapify
Swap 4 and 5

Given array in unsorted order

First build a heap in place

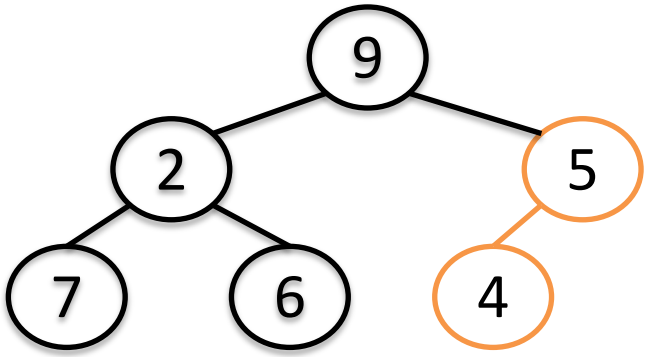
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Given array in unsorted order

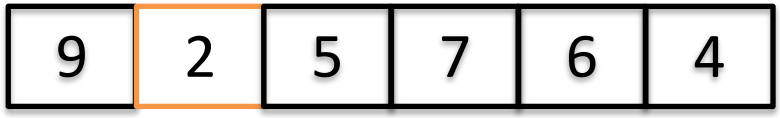
First build a heap in place

- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

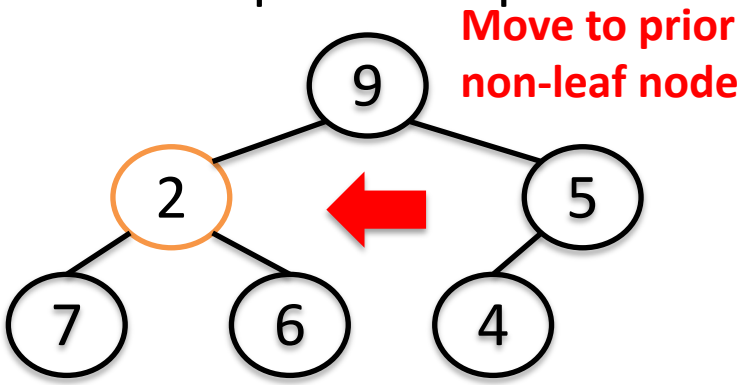
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Given array in unsorted order

First build a heap in place

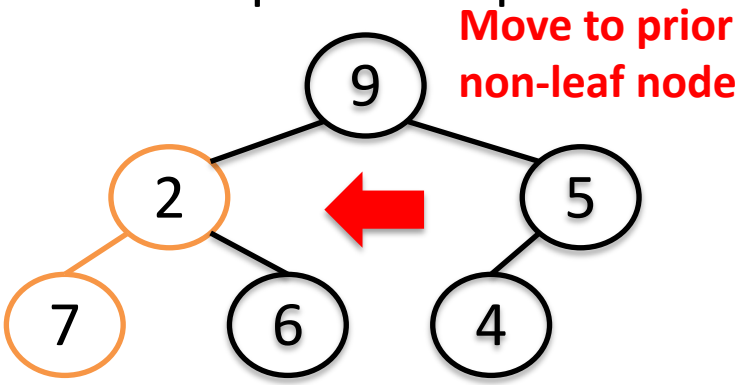
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Max heapify
Swap 2 and 7

Given array in unsorted order

First build a heap in place

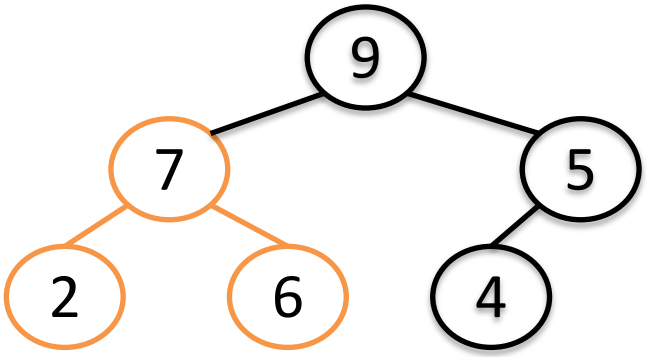
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Max heapify
Swap 2 and 7

Given array in unsorted order

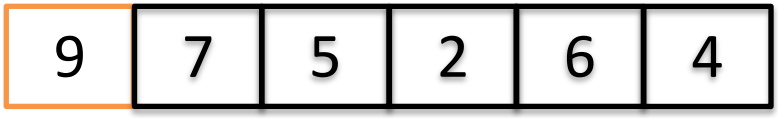
First build a heap in place

- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

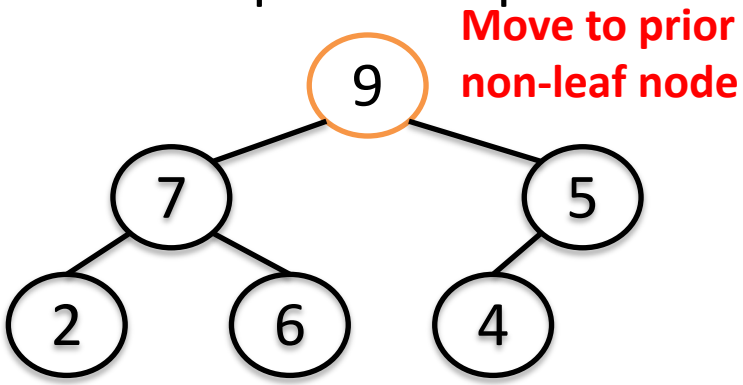
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Given array in unsorted order

First build a heap in place

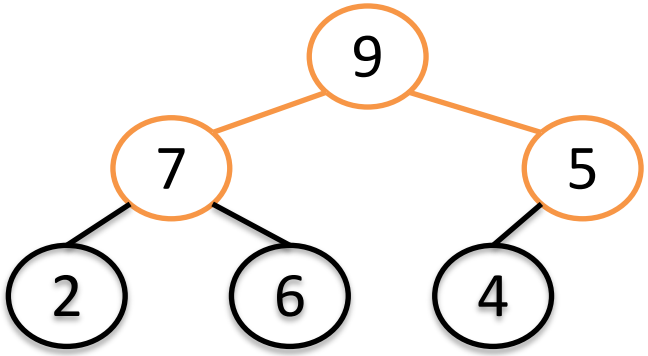
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Max heapify
In order, no need to swap

Given array in unsorted order

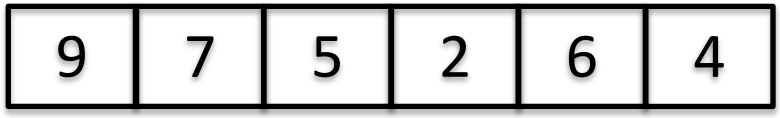
First build a heap in place

- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

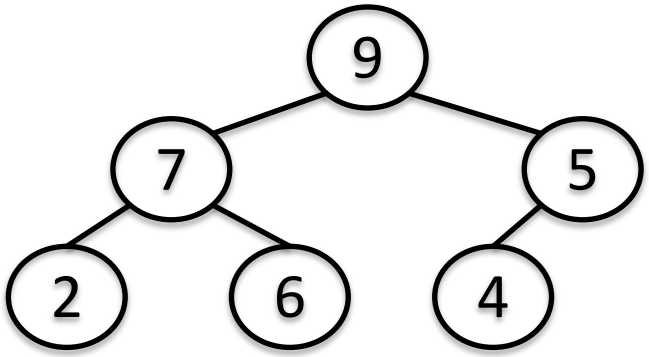
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



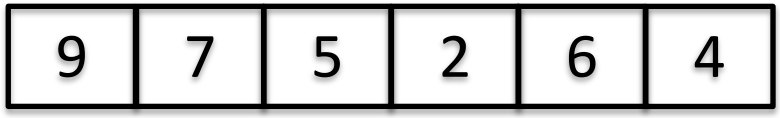
Now it's a max heap!
Satisfies Shape and Order Properties

Given array in unsorted order
First build a heap in place

- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

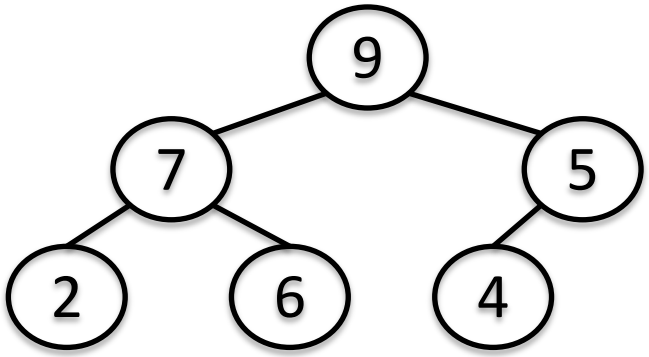
After building the heap, parents are larger than children, but items may not be sorted

Array



Heap array after construction

Conceptual heap tree



Heap order is maintained here

Looping over array does not give elements in sorted order

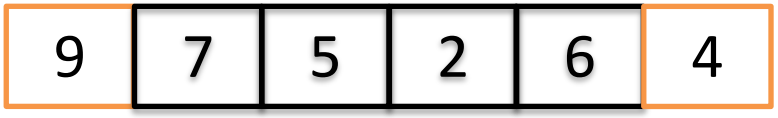
Traversing tree doesn't work either

- Preorder = 9,7,2,6,5,4
- Inorder = 2,7,6,9,4,5
- Post order = 2,6,7,4,5,9

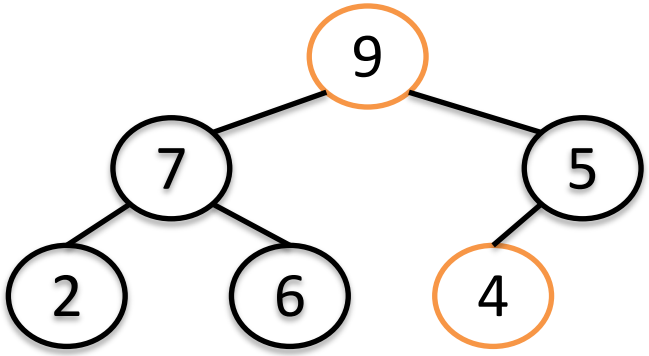
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Array



Conceptual heap tree



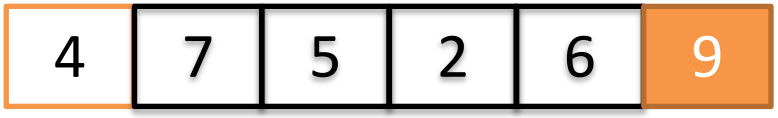
extractMax() = 9

Swap with last item in array

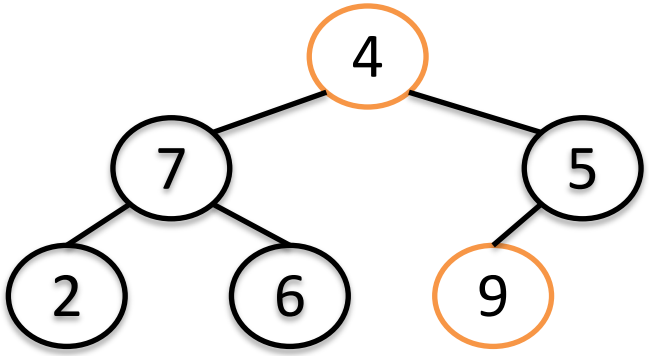
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Array



Conceptual heap tree

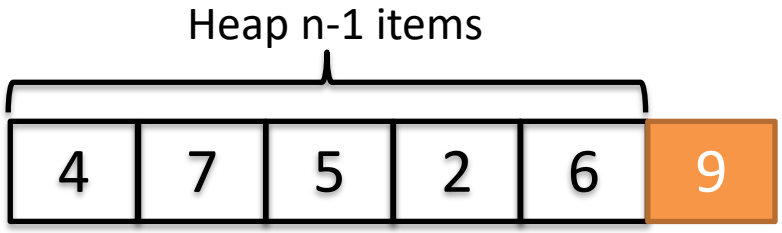


extractMax() = 9

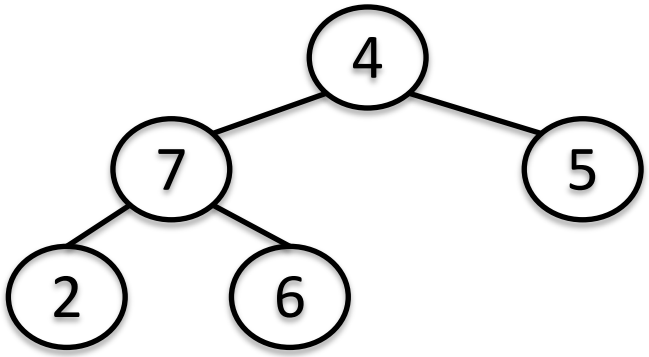
Swap with last item in array

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



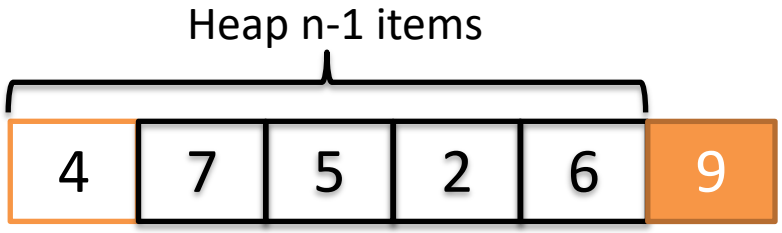
Conceptual heap tree



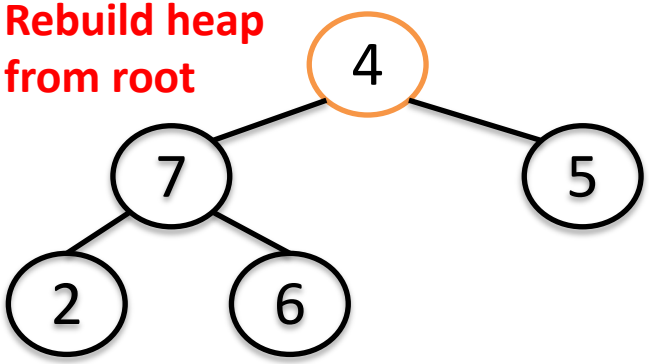
Rebuild heap on n-1 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



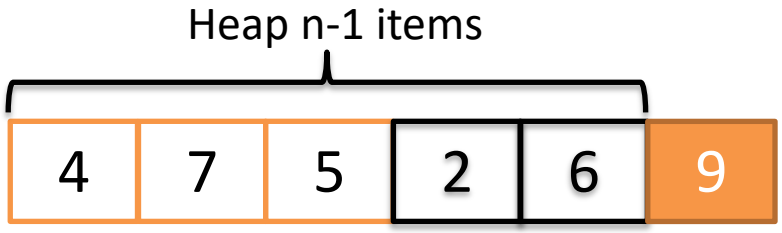
Conceptual heap tree



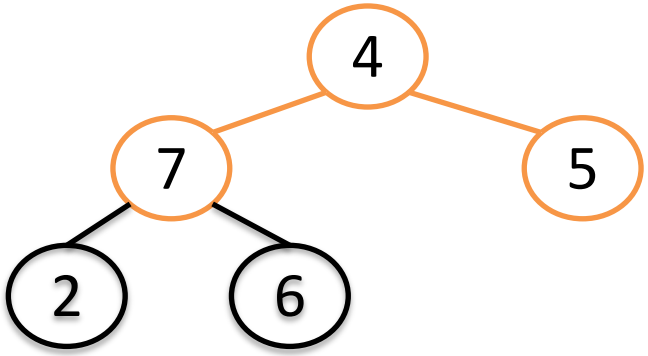
Rebuild heap on n-1 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Conceptual heap tree



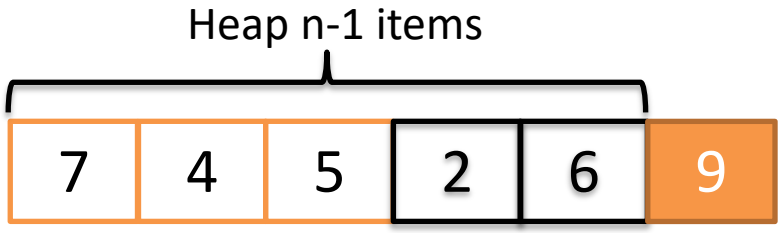
Swap 4 with largest child 7

Max heapify
Swap 7 and 4

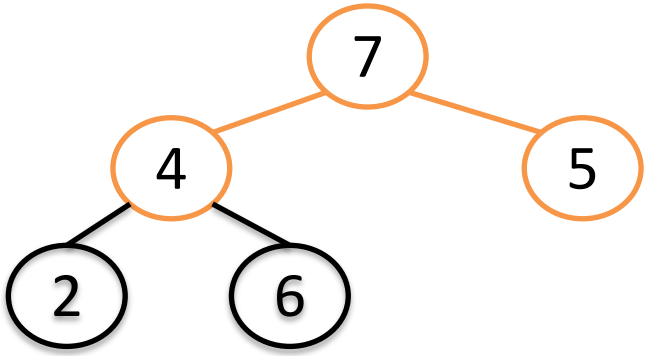
Rebuild heap on n-1 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Conceptual heap tree

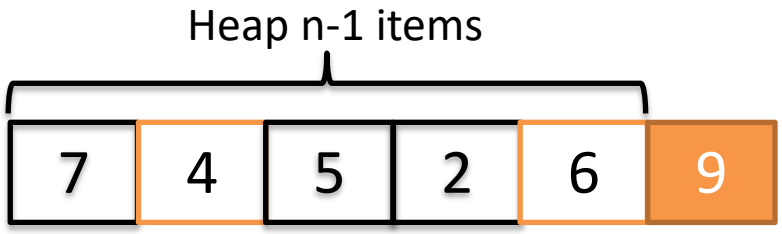


Max heapify
Swap 7 and 4

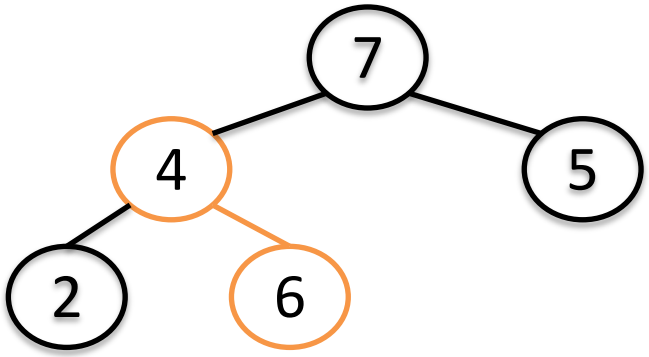
Rebuild heap on n-1 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Conceptual heap tree



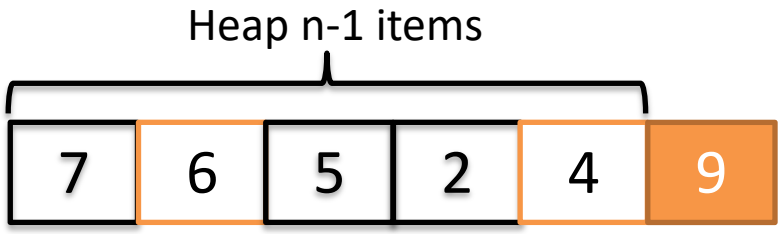
Swap 4 with largest child 6

Max heapify
Swap 4 and 6

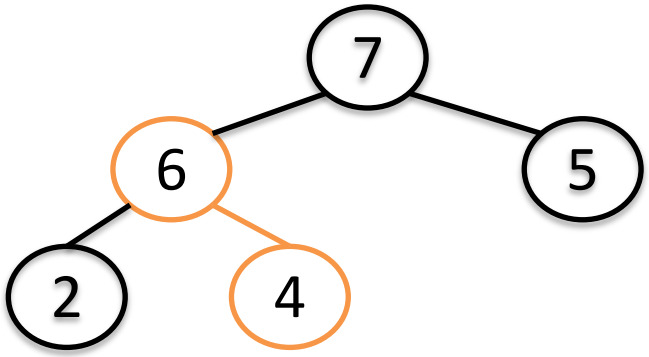
Rebuild heap on n-1 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Conceptual heap tree

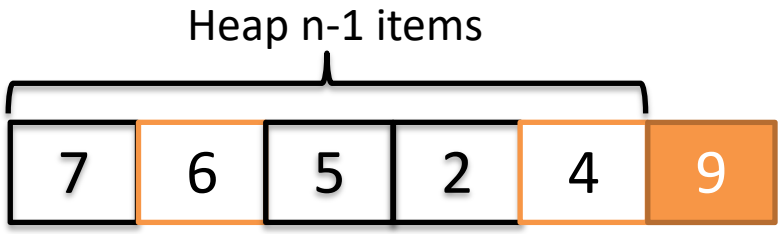


Max heapify
Swap 4 and 6

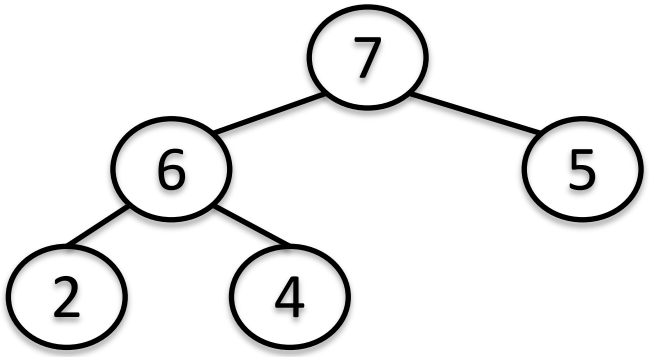
Rebuild heap on n-1 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Conceptual heap tree

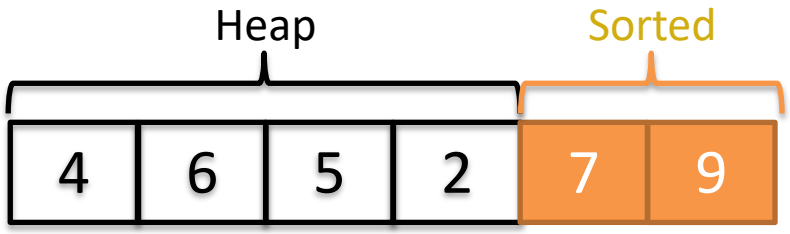


Heap built

Rebuild heap on n-1 items

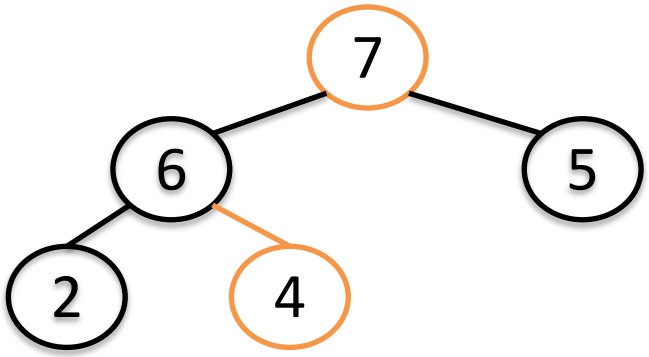
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

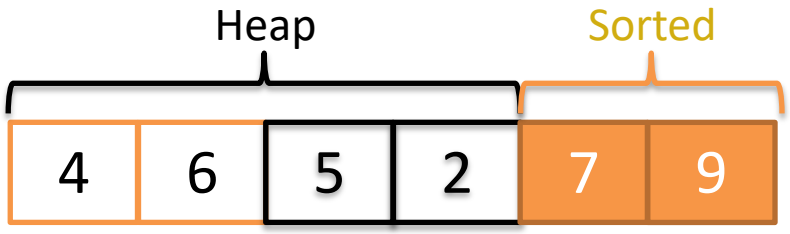


extractMax() = 7

Swap with last item in array

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

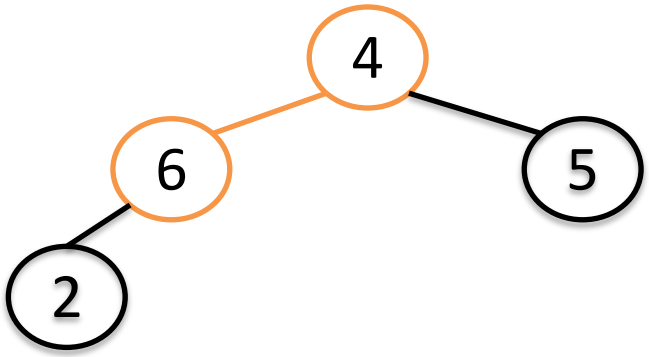
Heap on left, sorted on right



Heap array

Swap 4 with largest child 6

Conceptual heap tree

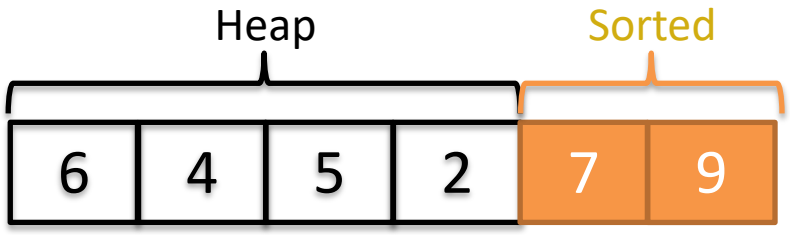


Max heapify
Swap 4 and 6

Rebuild heap on n-2 items

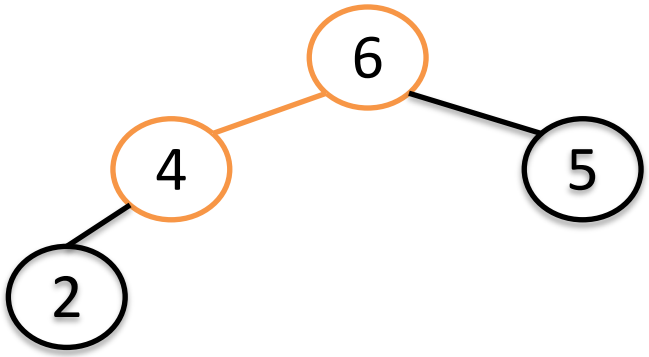
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

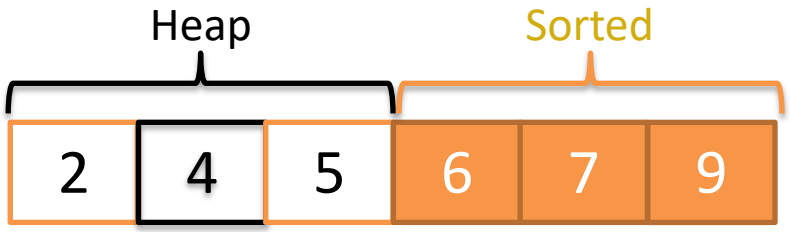


Heap built

Rebuild heap on n-2 items

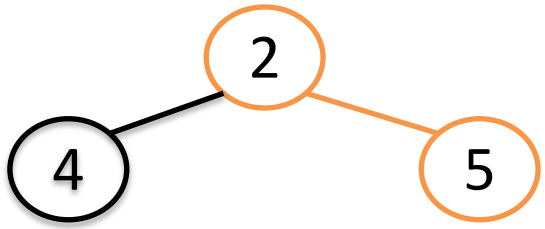
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree



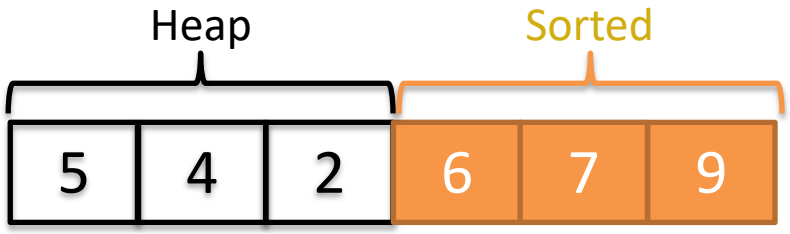
Swap 2 with largest child 5

Max heapify
Swap 5 and 2

extractMax() = 6
Swap with last item in array

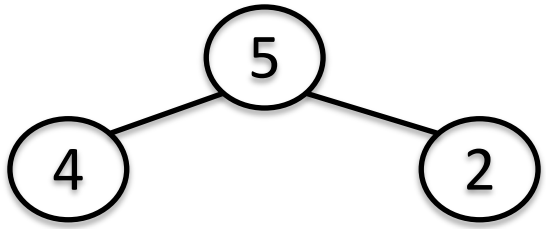
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

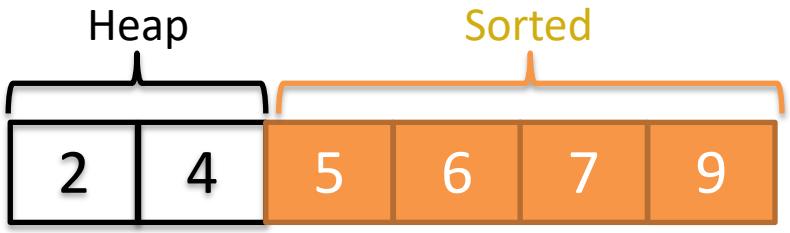


Heap built

Rebuild heap on n-3 items

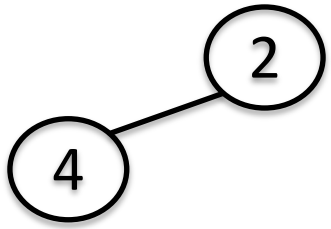
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

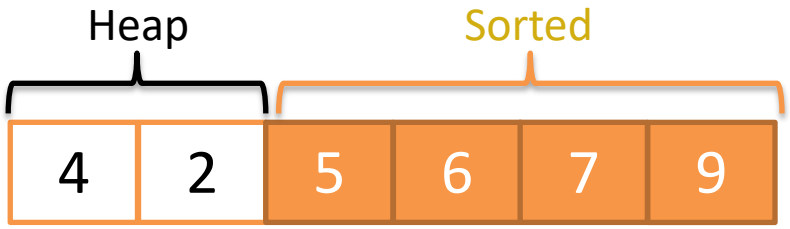


extractMax() = 5

Swap with last item in array

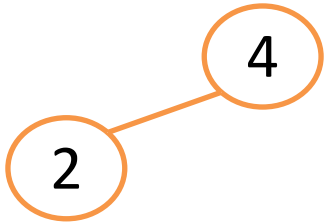
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

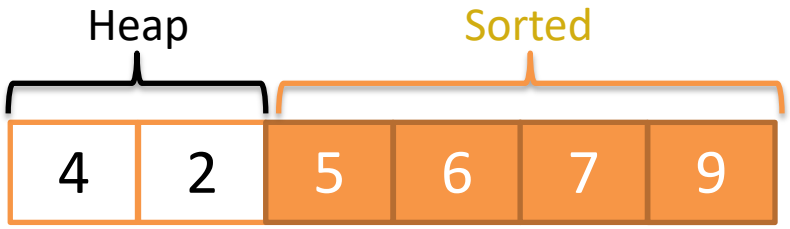


Max heapify
Swap 4 and 2

Rebuild heap on n-4 items

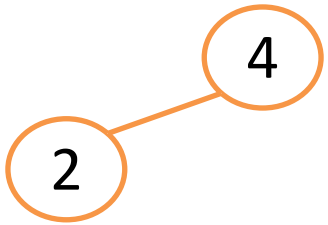
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree



Heap built

Rebuild heap on n-4 items

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Heap array

Conceptual heap tree

extractMax() = 4

Swap with last item in array

Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Heap array

Conceptual heap tree

Done

Items sorted in place

No extra memory used

Heapsort.java: First build heap, then extractMin, rebuilt heap…

```
 9 public class Heapsort<E extends Comparable<E>> {
10     //no constructor!  instead we sort arrays in place
11
12     /**
13      * Sort the array a[0..n-1] *inplace* using the heapsort algorithm.
14      */
15     public void sort(E[] a, int n) {
16         heapsort(a, n - 1);
17     }
18
19     /**
20      * Sort the array a[0..lastLeaf] by the heapsort algorithm.
21      */
22     private void heapsort(E[] a, int lastLeaf) {
23         // First, turn the array a[0..lastLeaf] into a max-heap.
24         buildMaxHeap(a, lastLeaf);
25
26         // Once the array is a max-heap, repeatedly swap the root
27         // with the last leaf, putting the largest remaining element
28         // in the last leaf's position, declare this last leaf to no
29         // longer be in the heap, and then fix up the heap.
30         while (lastLeaf > 0) {
31             swap(a, 0, lastLeaf);       // swap the root with the last leaf
32             lastLeaf--;                 // the last leaf is no longer in the heap
33             maxHeapify(a, 0, lastLeaf); // fix up what's left
34         }
35     }
```

Heapsort.java: First build heap, then extractMin, rebuilt heap…

```
42    private void maxHeapify(E[] a, int i, int lastLeaf) {
43        int left = leftChild(i);      // index of node i's left child
44        int right = rightChild(i);    // index of node i's right chil
45        int largest;                  // will hold the index of the n
46
47        // Is there a left child and, if so, does the left child have
48        if (left <= lastLeaf && a[left].compareTo(a[i]) > 0)
49            largest = left; // yes, so the left child is the largest
50        else
51            largest = i;    // no, so node i is the largest so far
52
53        // Is there a right child and, if so, does the right child ha
54        // element larger than the larger of node i and the left chil
55        if (right <= lastLeaf && a[right].compareTo(a[largest]) > 0)
56            largest = right; // yes, so the right child is the larges
57
58        /*
59         * If node i holds an element larger than both the left and r
60         * children, then the max-heap property already held, and we
61         * nothing more. Otherwise, we need to swap node i with the l
62         * of the two children, and then recurse down the heap from t
63         * child.
64         */
65        if (largest != i) {
66            swap(a, i, largest);
67            maxHeapify(a, largest, lastLeaf);
68        }
69    }
70
71    /**
72     * Form array a[0..lastLeaf] into a max-heap.
73     */
74    private void buildMaxHeap(E[] a, int lastLeaf) {
75        int lastNonLeaf = (lastLeaf - 1) / 2; // nodes lastNonLeaf+1
76        for (int j = lastNonLeaf; j >= 0; j--)
77            maxHeapify(a, j, lastLeaf);
78    }
```

Heapsort in two steps

Given array in unknown order

1. Build max heap in place using array given
   - Start with last non-leaf node, max heapify node and children
   - Move to next to last non-leaf node, max heapify again
   - Repeat until at root
   - NOTE: heap is not necessarily sorted, only know parent > children and max is at root

2. Extract max (index 0) and swap with item at end of array, then rebuild heap not considering last item

Does not require additional memory to sort

Run time:

Building heap is *O(n)* – see course web page (most nodes are leaves)

Each extractMax/swap might need O(log_2 *n)* operations to restore Heap

Make *n-1 = O(n)* extractMax/swaps to get array in sorted order

Total run time is *O(n) + O(n* log_2 *n) = O(n* log_2 *n)*

Heapsort.java

ANNOTATED SLIDES

Heapsort.java: First build heap, then extractMin, rebuilt heap…

```
 9 public class Heapsort<E extends Comparable<E>> {
10     //no constructor!  instead we sort arrays in place
11
12     /**
13      * Sort the array a[0..n-1] *inplace* using the heapsort algorithm.
14      */
15     public void sort(E[] a, int n) {
16         heapsort(a, n - 1);
17     }
18
19     /**
20      * Sort the array a[0..lastLeaf] by the heapsort algorithm.
21      */
22     private void heapsort(E[] a, int lastLeaf) {
23         // First, turn the array a[0..lastLeaf] into a max-heap.
24         buildMaxHeap(a, lastLeaf);
25
26         // Once the array is a max-heap, repeatedly swap the root
27         // with the last leaf, putting the largest remaining element
28         // in the last leaf's position, declare this last leaf to no
29         // longer be in the heap, and then fix up the heap.
30         while (lastLeaf > 0) {
31             swap(a, 0, lastLeaf);       // swap the root with the last leaf
32             lastLeaf--;                 // the last leaf is no longer in the heap
33             maxHeapify(a, 0, lastLeaf); // fix up what's left
34         }
35     }
```

Code very similar to HeapMinPriorityQueue.java

- **Sort() method calls helper with size of heap to consider**
- **Initially consider each element**

First build heap from root to last element to be considered (initially last element, then n-2, then n-3,…)

While not at root, (lastLeaf > 0)
Swap root and last element

Reduce size of heap to consider
Rebuild smaller heap
Done when at root

Heapsort.java: First build heap, then extractMin, rebuilt heap…

```
42    private void maxHeapify(E[] a, int i, int lastLeaf) {
43        int left = leftChild(i);      // index of node i's left child
44        int right = rightChild(i);    // index of node i's right chil
45        int largest;                  // will hold the index of the n
46
47        // Is there a left child and, if so, does the left child have
48        if (left <= lastLeaf && a[left].compareTo(a[i]) > 0)
49            largest = left; // yes, so the left child is the largest
50        else
51            largest = i;    // no, so node i is the largest so far
52
53        // Is there a right child and, if so, does the right child ha
54        // element larger than the larger of node i and the left chil
55        if (right <= lastLeaf && a[right].compareTo(a[largest]) > 0)
56            largest = right; // yes, so the right child is the larges
57
58        /*
59         * If node i holds an element larger than both the left and r
60         * children, then the max-heap property already held, and we
61         * nothing more. Otherwise, we need to swap node i with the l
62         * of the two children, and then recurse down the heap from t
63         * child.
64         */
65        if (largest != i) {
66            swap(a, i, largest);
67            maxHeapify(a, largest, lastLeaf);
68        }
69    }
70
71    /**
72     * Form array a[0..lastLeaf] into a max-heap.
73     */
74    private void buildMaxHeap(E[] a, int lastLeaf) {
75        int lastNonLeaf = (lastLeaf - 1) / 2; // nodes lastNonLeaf+1
76        for (int j = lastNonLeaf; j >= 0; j--)
77            maxHeapify(a, j, lastLeaf);
78    }
```

Finds largest between *i* and two children
If *largest* not *i*, swap *i* and *largest*
Recursively call *maxHeapify()* to bubble down *i* to right place

- ***buildHeap()* builds heap from last non-leaf node (parent of last leaf)**
- **Calls *maxHeapify()* on each non-leaf node until hit root**

Student.java – Method 1 with compareTo

ANNOTATED SLIDES

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()***

Student.java – Method 2 with comparator

ANNOTATED SLIDES

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()***

Student.java – Method 3 with anonymous function

ANNOTATED SLIDES

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)

EXAMPLE OF READING 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       }
```

- 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

- 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