CS 10: Problem solving via Object Oriented Programming

Prioritizing

Agenda

1. Priority queues
2. Heaps
3. Implementing a PriorityQueue with a Heap
4. Java’s PriorityQueue implementation
5. Supplemental information

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

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

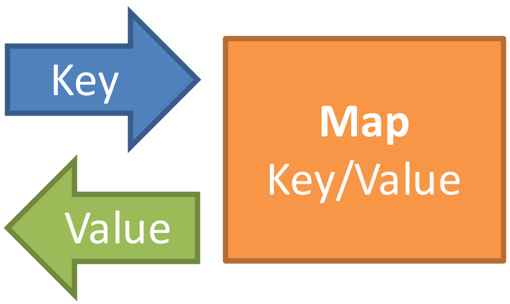
Need a way to go to front of queue

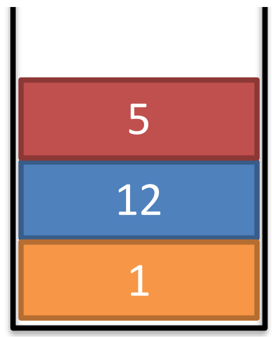
Enter the priority queue

Image: flickr

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

Queue (FIFO)





Stack (LIFO)

Priority Queue

Maps are a Key/Value store

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

Stacks/Queues arrival order

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

Priority Queue order

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

Priority Queues have the ability to extract the highest priority item

Priority Queue Overview

- Min: lowest priority number removed first ("number 1 for landing")
- Max: highest priority number removed first

Analogous methods for max priority queue

- **Min Priority Queue ADT 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)

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

Priority Queue

Key	11	8	9
Value	Machine 1	Machine 2	Machine 3

Machine 2

Start job at time 2
Job takes 6 minutes

Add to Priority Queue that job will finish at time 8

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

Machine 3

Start job at time 4
Job takes 5 minutes

Add to Priority Queue that job will finish at time 9

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

```java
public interface MinPriorityQueue<E extends Comparable<E>> {
    /**
     * Is the priority queue empty?
     * @return true if the priority queue is empty, false if not empty.
     */
    public boolean isEmpty();

    /**
     * Insert an element into the queue.
     * @param element thing to insert
     */
    public void insert(E element);

    /**
     * Return the element with the minimum key, without removing it from the queue.
     * @return the element with the minimum key in the priority queue
     */
    public E minimum();

    /**
     * Return the element with the minimum key, and remove it from the queue.
     * @return the element with the minimum key in the priority queue
     */
    public E extractMin();
}
```

- **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**

Could implement the Priority Queue using a sorted or unsorted List

Unsorted List

15	6	9	27

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

Could implement the Priority Queue using a sorted or unsorted List

Unsorted List

15	6	9	27

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

15	6	9	27

Sorted List

27	15	9	6

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

15	6	9	27

Sorted List

27	15	9	6

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

15	6	9	27

Sorted List

27	15	9	6

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

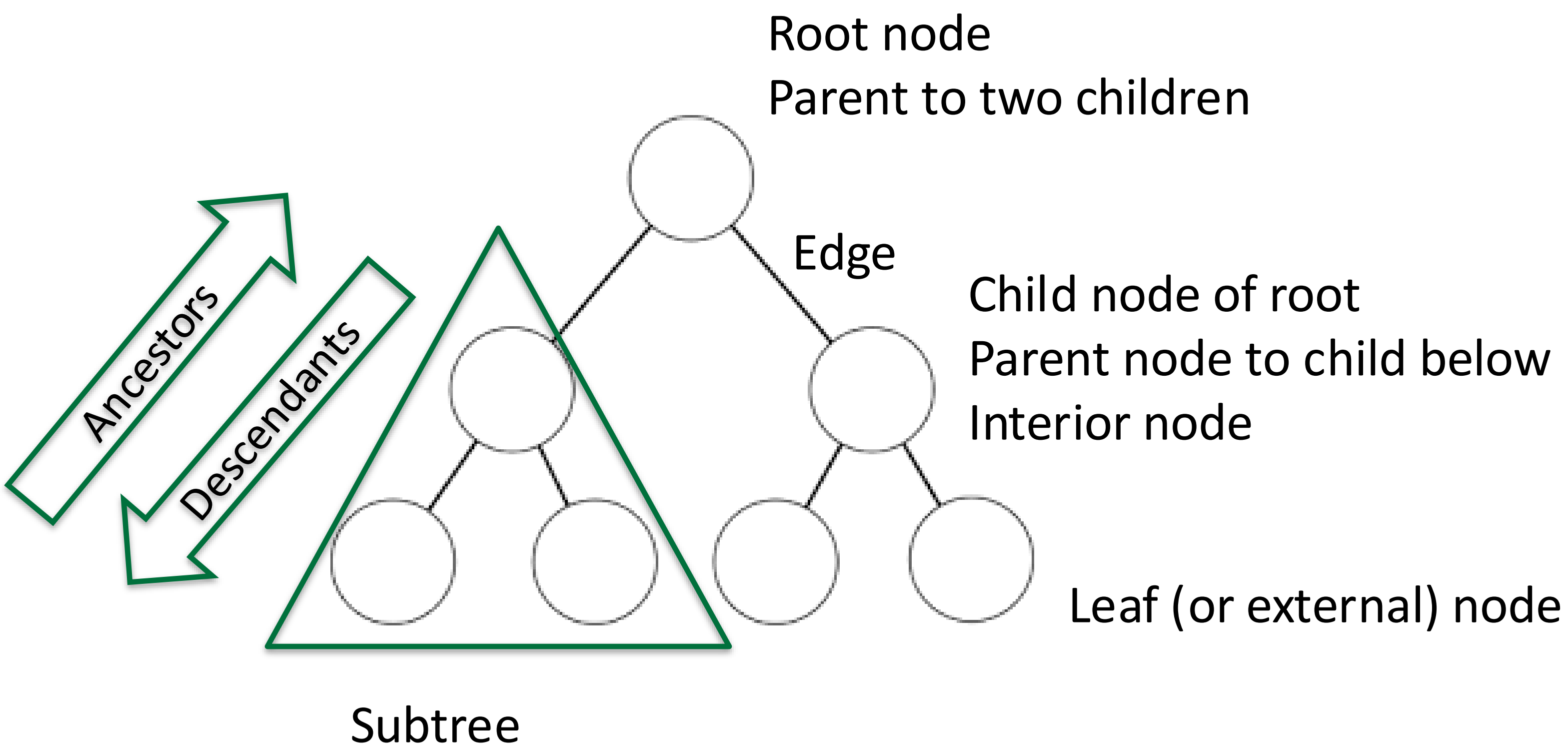
Either way we pay a price, on min/extractMin or on insert
Heaps are a better choice

Agenda

1. Priority queues
2. Heaps
3. Implementing a PriorityQueue with a Heap
4. Java's PriorityQueue implementation
5. Supplemental information

Heaps are conceptually based on Binary Trees

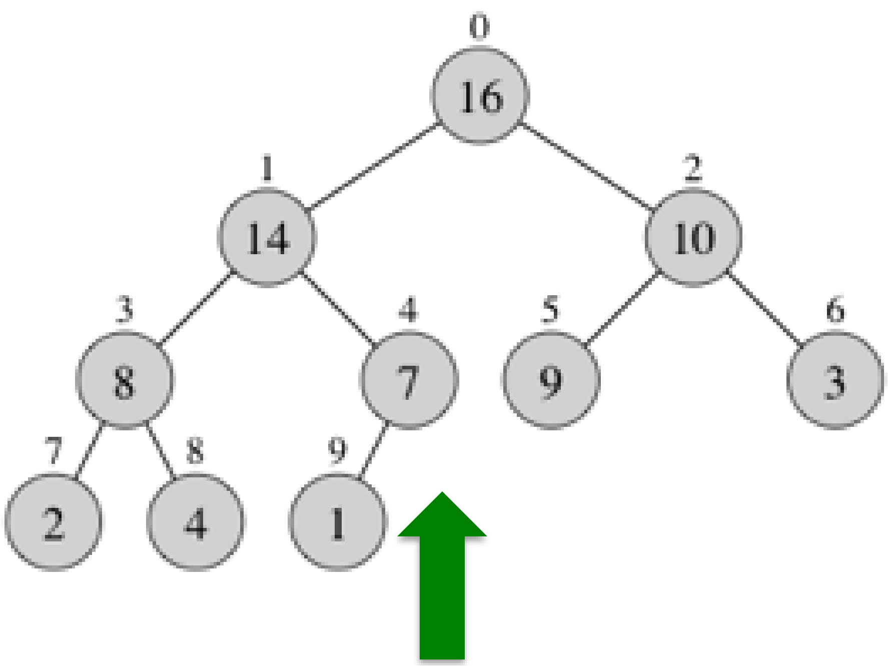
Tree data structure



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
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



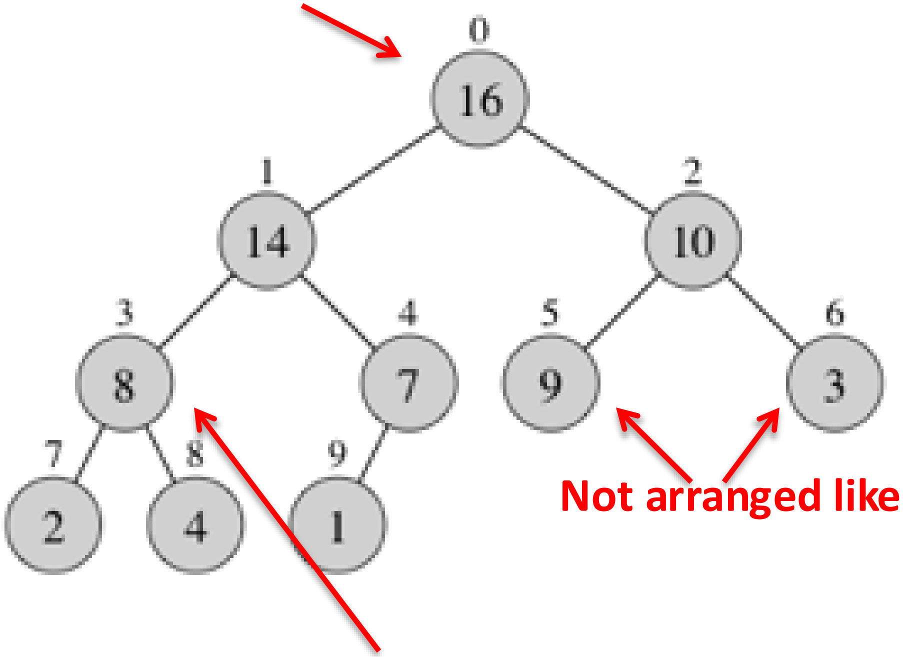
Next node added here

Shape property

- Nodes added starting from root and building downward
- Nodes added left to right
- New level started only once a prior level is filled
- 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



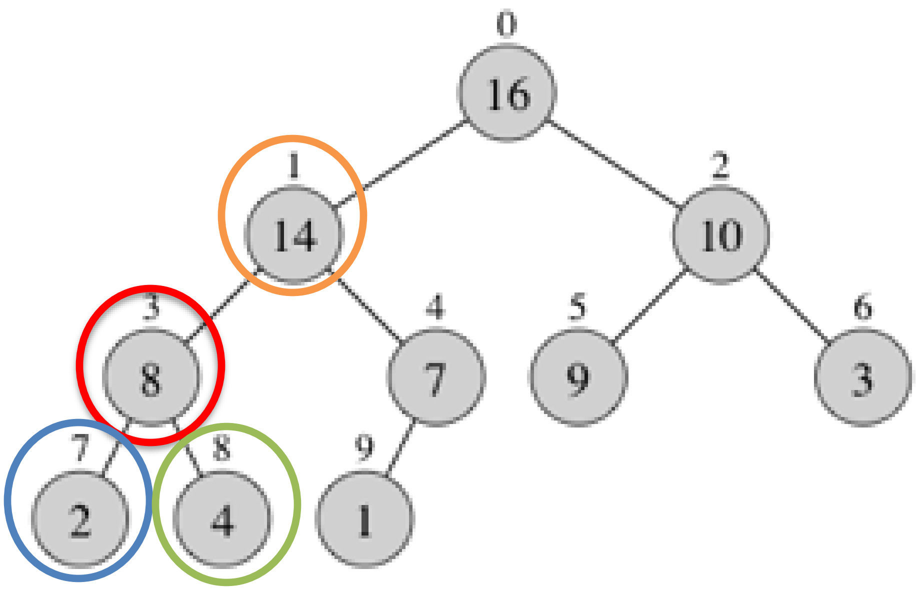
Order property

- ∀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

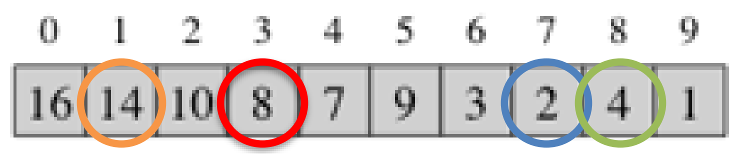
Array implementation

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

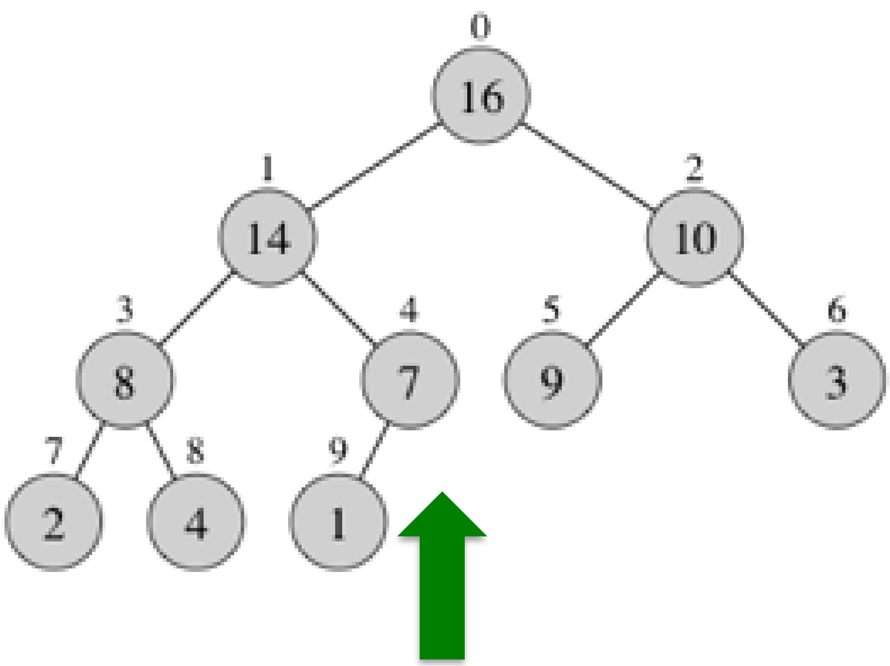
1. Priority queues
2. Heaps
3. Implementing a PriorityQueue with a Heap
4. Java’s PriorityQueue implementation
5. Supplemental information

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)

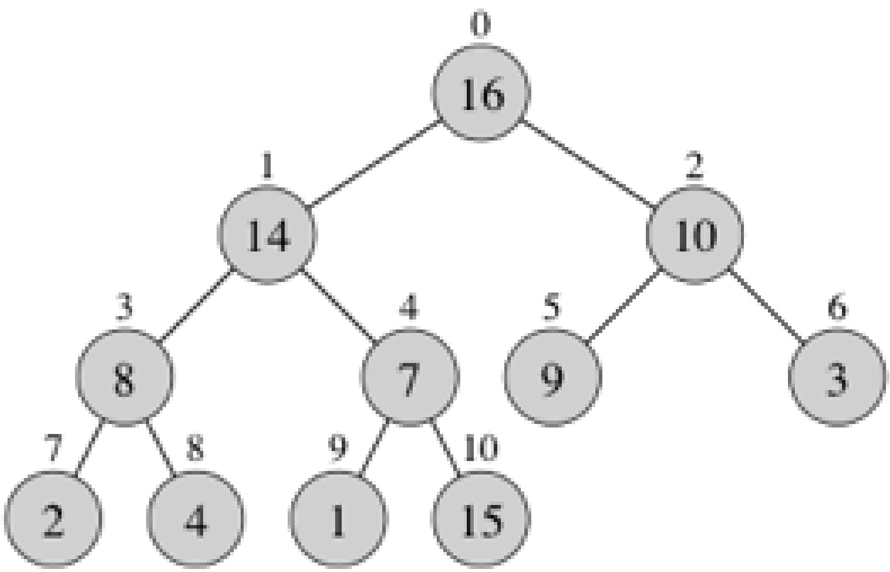


Next node added here

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

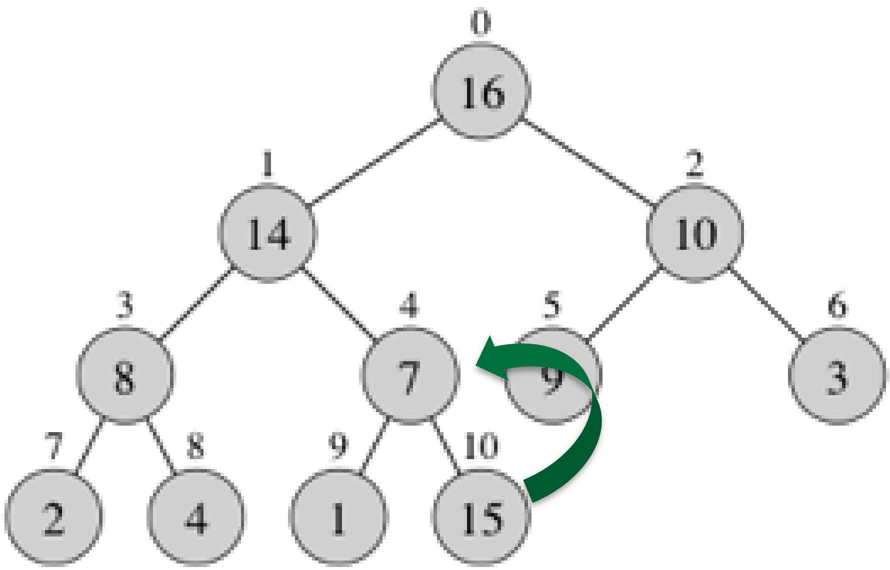
- Shape property: fill in next spot in left to right order (index i=10)

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

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

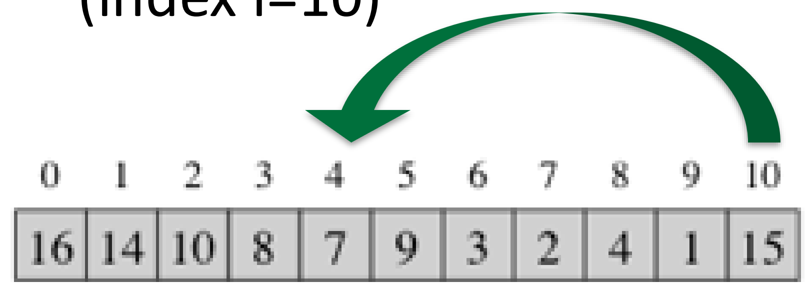
Inserting into max heap must keep both shape and order properties intact

Max heap insert



Insert 15

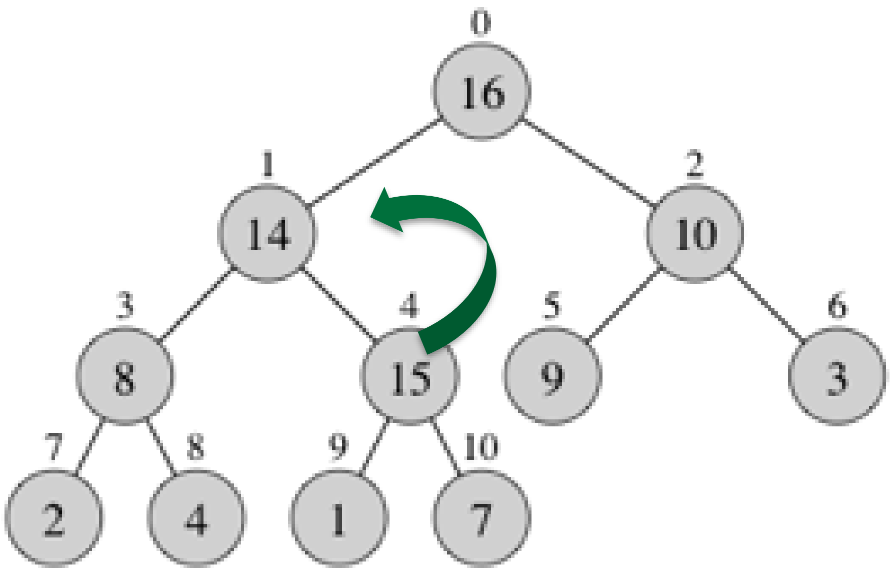
- Shape property: fill in next spot in left to right order (index i=10)



- 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

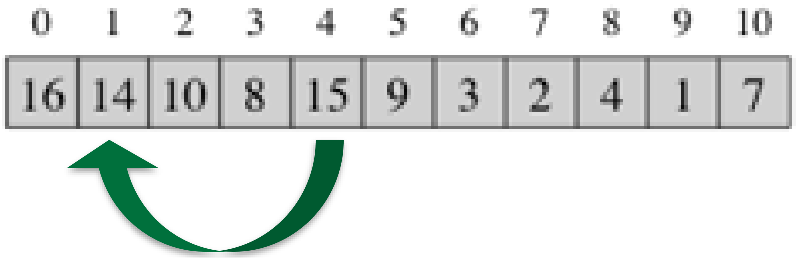
We may have to swap multiple times to get both heap properties

Max heap insert



Insert 15

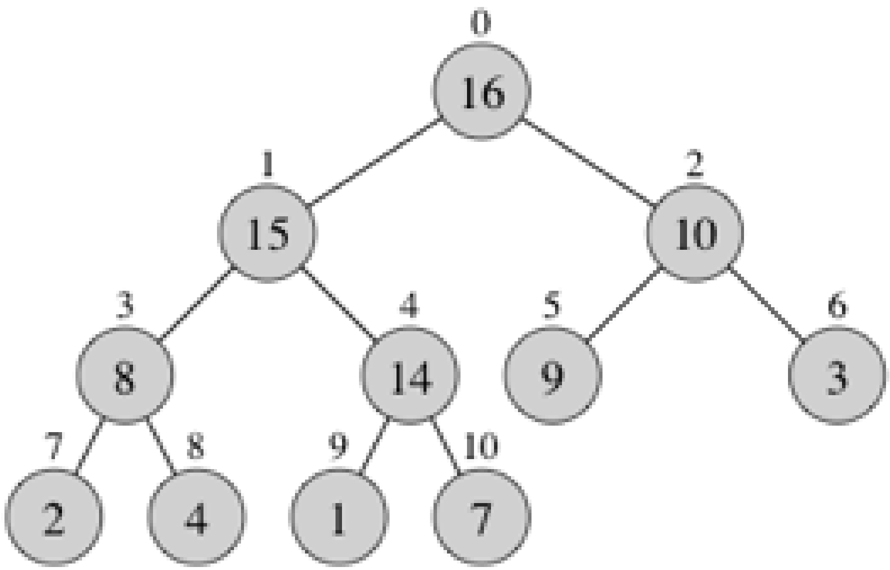
- Shape property: good!
- Order property: parent must be larger than children, not met



- 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 (smallest) node at root**

Insert 15

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

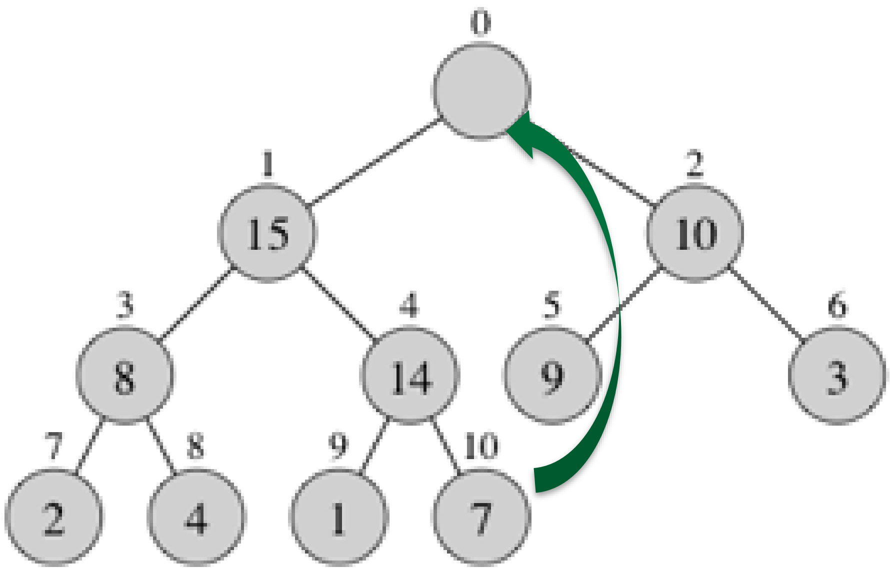
0	1	2	3	4	5	6	7	8	9	10
16	15	10	8	14	9	3	2	4	1	7

General rule

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

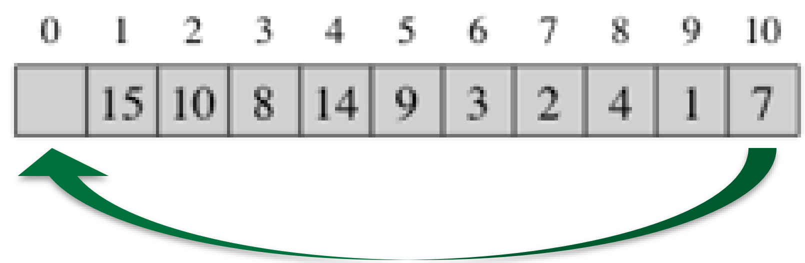
extractMax means removing the root, but that leaves a hole

extractMax



extractMax -> 16

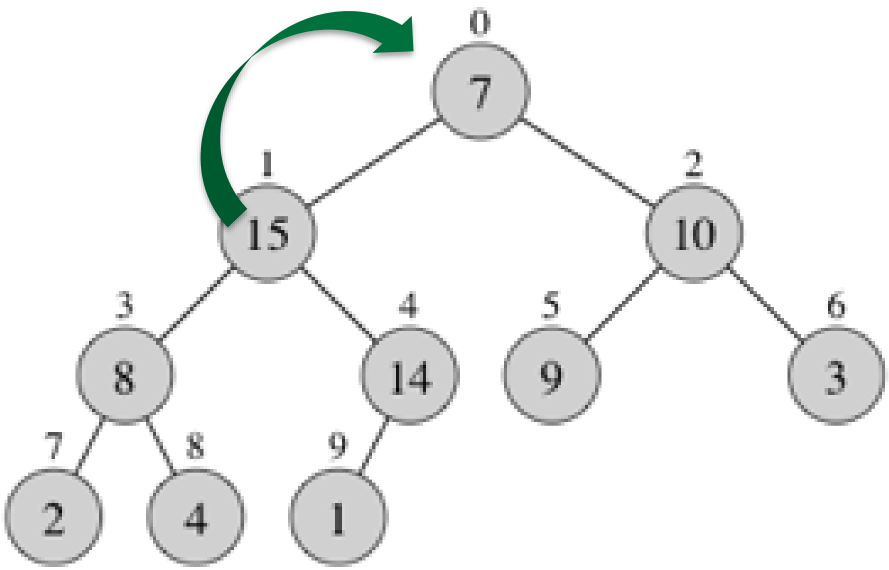
- Max position is at root (index 0)
- Removing it leaves a hole, violating shape property



- 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



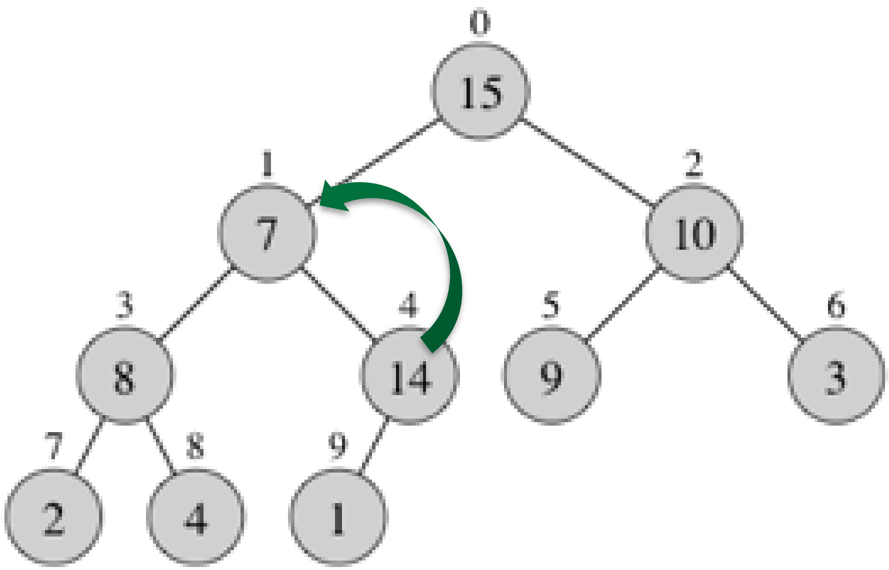
After swap

- 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

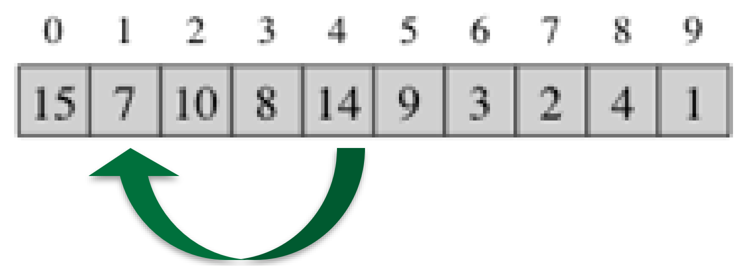
May need multiple swaps to restore order property

extractMax



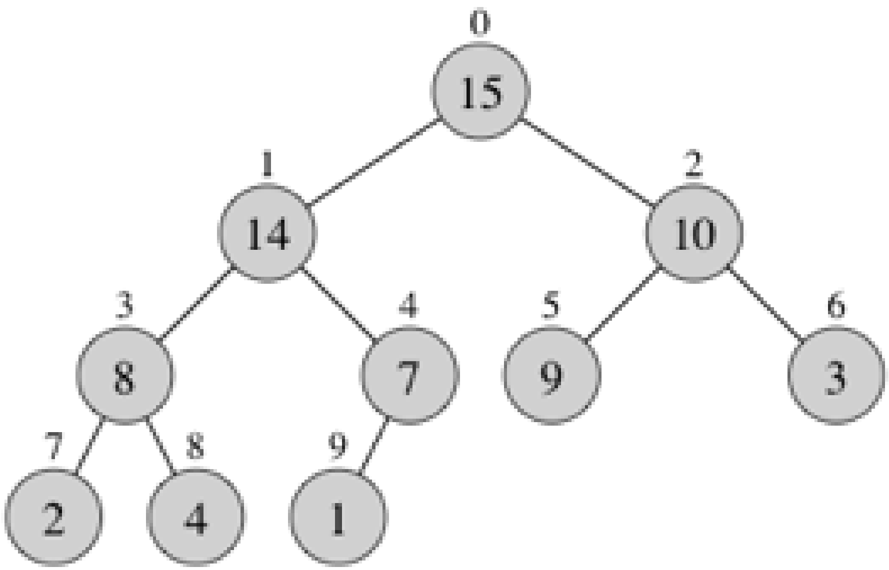
After swap 15 and 7

- 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!

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

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

Heap elements extend Comparable

```
public class HeapMinPriorityQueue<E extends Comparable<E>>
implements MinPriorityQueue<E> {
    private ArrayList<E> heap;

    /**
     *  Constructor
     */
    public HeapMinPriorityQueue() {
        heap = new ArrayList<E>();
    }

```

ArrayList called *heap* will hold the heap

NOTE: example was for a MAX Priority Queue, this code implements a <u>MIN</u> Priority Queue

Easy to change to this code to a MAX Priority Queue **(see slide 32)**

Helper functions make finding parent and children easy

HeapMinPriorityQueue.java

Helper functions

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

```
// Swap two locations i and j in ArrayList a.
private static <E> void swap(ArrayList<E> a, int i, int j) {
    E temp = a.get(i); //temporarily hold item at index i
    a.set(i, a.get(j)); //set item at index i to item at index j
    a.set(j, temp); //set item at index j to temp
}

// Return the index of the left child of node i.
private static int leftChild(int i) {
    return 2*i + 1;
}

// Return the index of the right child of node i.
private static int rightChild(int i) {
    return 2*i + 2;
}

// Return the index of the parent of node i
// (Parent of root will be -1)
private static int parent(int i) {
    return (i-1)/2;
}
```

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**

```java
public void insert(E element) {
    heap.add(element);          // Put new value at end;
    int loc = heap.size()-1;  // and get its location

    // Swap with parent until parent not larger
    while (loc > 0 && heap.get(loc).compareTo(heap.get(parent(loc))) < 0) {
        swap(heap, loc, parent(loc));
        loc = parent(loc);
    }
}
```

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

HeapMinPriorityQueue.java

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

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

```java
    public void insert(E element) {
        heap.add(element);          // Put new value at end;
        int loc = heap.size()-1;    // and get its location

        // Swap with parent until parent not larger
        while (loc > 0 && heap.get(loc).compareTo(heap.get(parent(loc))) < 0) {
            swap(heap, loc, parent(loc));
            loc = parent(loc);
        }
    }
```

- **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

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

```java
    public E extractMin() {
        if (heap.size() <= 0)
            return null;
        else {
            E minVal = heap.get(0); //min will be at node 0
            heap.set(0, heap.get(heap.size()-1));  // Move last to position 0
            heap.remove(heap.size()-1); //remove last item to maintain shape prope
            minHeapify(heap, 0); //recursively swap to maintain order property
            return minVal; //return min value
        }
    }
```

- **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

```java
    private static <E extends Comparable<E>> void
    minHeapify(ArrayList<E> a, int i) {
        int left = leftChild(i);    // index of node i's left child
        int right = rightChild(i);  // index of node i's right child
        int smallest;    // will hold the index of the node with the smallest eleme
        // among node i, left, and right

        // Is there a left child and, if so, does the left child have an
        // element smaller than node i?
        if (left <= a.size()-1 && a.get(left).compareTo(a.get(i)) < 0)
            smallest = left;   // yes, so the left child is the largest so far
        else
            smallest = i;      // no, so node i is the smallest so far

        // Is there a right child and, if so, does the right child have an
        // element smaller than the larger of node i and the left child?
        if (right <= a.size()-1 && a.get(right).compareTo(a.get(smallest)) < 0)
            smallest = right;  // yes, so the right child is the largest

        // If node i holds an element smaller than both the left and right
        // children, then the min-heap property already held, and we need do
        // nothing more.  Otherwise, we need to swap node i with the larger
        // of the two children, and then recurse down the heap from the larger chil
        if (smallest != i) {
            swap(a, i, smallest); //put smallest in spot i, largest in spot smalle
            minHeapify(a, smallest); //maintain heap starting from smallest index (
        }
    }
```

a = heap, i = starting index

Get left and right children indices

- **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

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted List	Sorted List
isEmpty	O(1)	O(1)	O(1)

isEmpty()

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

Run time analysis shows Priority Queue heap implementation better than previous

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

insert()

- **Heap**: insert at end O(1), then may have to bubble up height of tree; $O(\log_2 n)$
- **Unsorted ArrayList:** just add on end of ArrayList; O(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 List	Sorted List
isEmpty	O(1)	O(1)	O(1)
insert	**$O(\log_2 n)$**	O(1)	**O(n)**
minimum	**O(1)**	**Θ(n)**	O(1)

minimum()

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

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted List	Sorted List
isEmpty	O(1)	O(1)	O(1)
insert	**$O(\log_2 n)$**	O(1)	**O(n)**
minimum	**O(1)**	**Θ(n)**	O(1)
extractMin	**$O(\log_2 n)$**	**Θ(n)**	O(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 fill hole with last item; O(n)
- **Sorted ArrayList:** return last item in ArrayList; O(1)

Run time analysis shows Priority Queue heap implementation better than previous

Operation	Heap	Unsorted List	Sorted List
isEmpty	O(1)	O(1)	O(1)
insert	**$O(\log_2 n)$**	O(1)	**O(n)**
minimum	**O(1)**	**Θ(n)**	O(1)
extractMin	**$O(\log_2 n)$**	**Θ(n)**	O(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 queues
2. Heaps
3. Implementing a PriorityQueue with a Heap
4. Java's PriorityQueue implementation
5. Supplemental information

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

SimpleStudent.java

```
public class SimpleStudent implements Comparable<SimpleStudent> {
  private String name;
  private int year;

  public SimpleStudent(String name, int year) {
    this.name = name;
    this.year = year;
  }

  /**
   * Comparable: just use String's version (lexicographic)
   */
  @Override
  public int compareTo(SimpleStudent s2) {
    return name.compareTo(s2.name);
  }

  @Override
  public String toString() {
    return name + " "+year;
  }
```

Stores data about a student's *name* and *year*

Class implements Comparable so PriorityQueue holding *SimpleStudent objects* can compare students

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

Here we use the built in String *compareTo()* method to evaluate *SimpleStudents* 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**

This approach sorts increasing alphabetically by student name

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

SimpleStudent.java

```
public static void main(String[] args) {
   //create List of students and add some
   List<SimpleStudent> students = new ArrayList<SimpleStudent>();
   students.add(new SimpleStudent("charlie", 18));
   students.add(new SimpleStudent("alice", 20));
   students.add(new SimpleStudent("bob", 19));
   students.add(new SimpleStudent("elvis", 21));
   students.add(new SimpleStudent("denise", 20));
   System.out.println("original:" + students);

   // Three methods for using Comparator

   // Method 1:
   // Create Java PriorityQueue and use Student
   // class's compareTo method (lexicographic order)
   // this is used if comparator not passed to PriorityQueue constructor
   PriorityQueue<SimpleStudent> pq = new PriorityQueue<SimpleStudent>();
   pq.addAll(students); //add all Students in ArrayList in one statement

   //remove until empty (this essentially sorting!)
   System.out.println("\nlexicographic:");
   while (!pq.isEmpty()) System.out.println(pq.remove());
```

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

- **Priority Queue created to hold *SimpleStudent* Objects**
- **No Comparator provided in constructor**
- **By default PriorityQueue will use *SimpleStudent* 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 *SimpleStudent* object using *compareTo()***

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

SimpleStudent.java

```
public static void main(String[] args) {
   //create List of students and add some
   List<SimpleStudent> students = new ArrayList<SimpleStudent>();
   students.add(new SimpleStudent("charlie", 18));
   students.add(new SimpleStudent("alice", 20));
   students.add(new SimpleStudent("bob", 19));
   students.add(new SimpleStudent("elvis", 21));
   students.add(new SimpleStudent("denise", 20));
   System.out.println("original:" + students);

   // Three methods for using Comparator

   // Method 1:
   // Create Java PriorityQueue and use Student
   // class's compareTo method (lexicographic order)
   // this is used if comparator not passed to PriorityQueue constructor
   PriorityQueue<SimpleStudent> pq = new PriorityQueue<SimpleStudent>();
   pq.addAll(students); //add all Students in ArrayList in one statement

   //remove until empty (this essentially sorting!)
   System.out.println("\nlexicographic:");
   while (!pq.isEmpty()) System.out.println(pq.remove());
```

Output in alphabetical order

```
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 Compator and pass to Priority Queue constructor

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

SimpleStudent.java

- **Still in *main()***
- **Method 2: 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()***

```
// Method 2:
// Use a custom Comparator.compare (length of name) instead
// of using the element's compareTo function
// Java will use this to compare two Students (here on length of name)
class NameLengthComparator implements Comparator<SimpleStudent> {
   public int compare(SimpleStudent s1, SimpleStudent s2) {
      return s1.name.length() - s2.name.length();
   }
}
Comparator<SimpleStudent> lenCompare = new NameLengthComparator();
pq = new PriorityQueue<SimpleStudent>(lenCompare); //passing Comparator to PriorityQueue
pq.addAll(students); //add all students to PriorityQueue
System.out.println("\nlength:");
//remove until empty (sorting)
while (!pq.isEmpty()) System.out.println(pq.remove());
```

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

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

SimpleStudent.java

- **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()***

```java
// Method 2:
// Use a custom Comparator.compare (length of name) instead
// of using the element's compareTo function
// Java will use this to compare two Students (here on length of name)
class NameLengthComparator implements Comparator<SimpleStudent> {
  public int compare(SimpleStudent s1, SimpleStudent s2) {
    return s1.name.length() - s2.name.length();
  }
}
Comparator<SimpleStudent> lenCompare = new NameLengthComparator();
pq = new PriorityQueue<SimpleStudent>(lenCompare); //passing Comparator to PriorityQueue
pq.addAll(students); //add all students to PriorityQueue
System.out.println("\nlength:");
//remove until empty (sorting)
while (!pq.isEmpty()) System.out.println(pq.remove());
```

- **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()***

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

SimpleStudent.java

```
// Method 2:
// Use a custom Comparator.compare (length of name) instead
// of using the element's compareTo function
// Java will use this to compare two Students (here on length of name)
class NameLengthComparator implements Comparator<SimpleStudent> {
   public int compare(SimpleStudent s1, SimpleStudent s2) {
      return s1.name.length() - s2.name.length();
   }
}
Comparator<SimpleStudent> lenCompare = new NameLengthComparator();
pq = new PriorityQueue<SimpleStudent>(lenCompare); //passing Comparator to PriorityQueue
pq.addAll(students); //add all students to PriorityQueue
System.out.println("\nlength:");
//remove until empty (sorting)
while (!pq.isEmpty()) System.out.println(pq.remove());
```

Output based on name length

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

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

SimpleStudent.java

```
//Method 3:
// Use a custom Comparator via Java 8 anonymous function (here based on year)
// pass Comparator to PriorityQueue constructor
pq = new PriorityQueue<SimpleStudent>((SimpleStudent s1, SimpleStudent s2) -> s2.year - s1.year);
pq.addAll(students); //add all students to Priority Queue
System.out.println("\nyear:");
//remove until empty (sorting)
while (!pq.isEmpty()) System.out.println(pq.remove());
```

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

Method 3: Use anonymous function in Priority Queue declaration

SimpleStudent.java

```
//Method 3:
// Use a custom Comparator via Java 8 anonymous function (here based on year)
// pass Comparator to PriorityQueue constructor
pq = new PriorityQueue<SimpleStudent>((SimpleStudent s1, SimpleStudent s2) -> s2.year - s1.year);
pq.addAll(students); //add all students to Priority Queue
System.out.println("\nyear:");
//remove until empty (sorting)
while (!pq.isEmpty()) System.out.println(pq.remove());
```

Output based on year in descending order (reversed order of compared objects)

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

Created a *Max* Priority Queue by simply reversing compare

Agenda

1. Priority queues
2. Heaps
3. Implementing a PriorityQueue with a Heap
4. Java’s PriorityQueue implementation
5. Supplemental information

Supplemental material

1. Reading from a file

2. Heapsort

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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- **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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- 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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- 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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- 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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- 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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- **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

```java
    public static List<Student> readRoster2(String fileName) throws IOException {
        List<Student> roster = new ArrayList<Student>();
        BufferedReader input;

        // Open the file, if possible
        try {
            input = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.err.println("Cannot open file.\n" + e.getMessage());
            return roster;
        }

        // Read the file
        try {
            // Line by line
            String line;
            int lineNum = 0;
            while ((line = input.readLine()) != null) {
                System.out.println("read @"+lineNum+"`"+line+"'");
                // Comma separated
                String[] pieces = line.split(",");
                if (pieces.length != 2) {
                    //did not get two elements in this line, output an error message
                    System.err.println("bad separation in line "+lineNum+":"+line);
                }
                else {
                    // got two elements for this line
                    try {
                        // Extract year as an integer, if possible
                        Student s = new Student(pieces[0], Integer.parseInt(pieces[1]));
                        System.out.println("=>"+s);
                        roster.add(s); //good student, add to roster
                    }
                    catch (NumberFormatException e) {
                        // couldn't parse second element as integer
                        System.err.println("bad number in line "+lineNum+":"+line);
                    }
                }
                lineNum++;
            }
        }
```

- 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

Supplemental material

1. Reading from a file
2. Heapsort

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

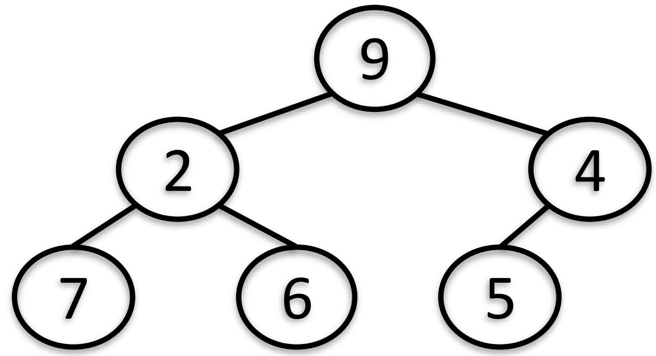
Step 1: build heap in place

Build heap given unsorted array

Array

9	2	4	7	6	5

Conceptual heap tree



Given array in unsorted order

First build a heap in place

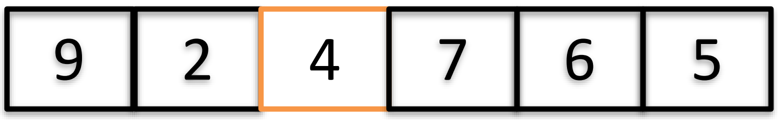
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Non heap!

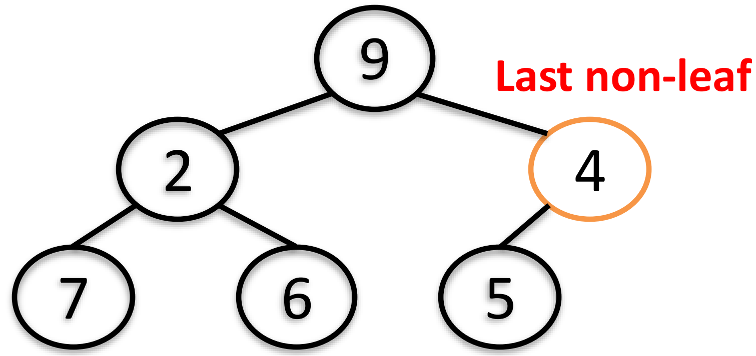
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Last non-leaf will be parent of last leaf

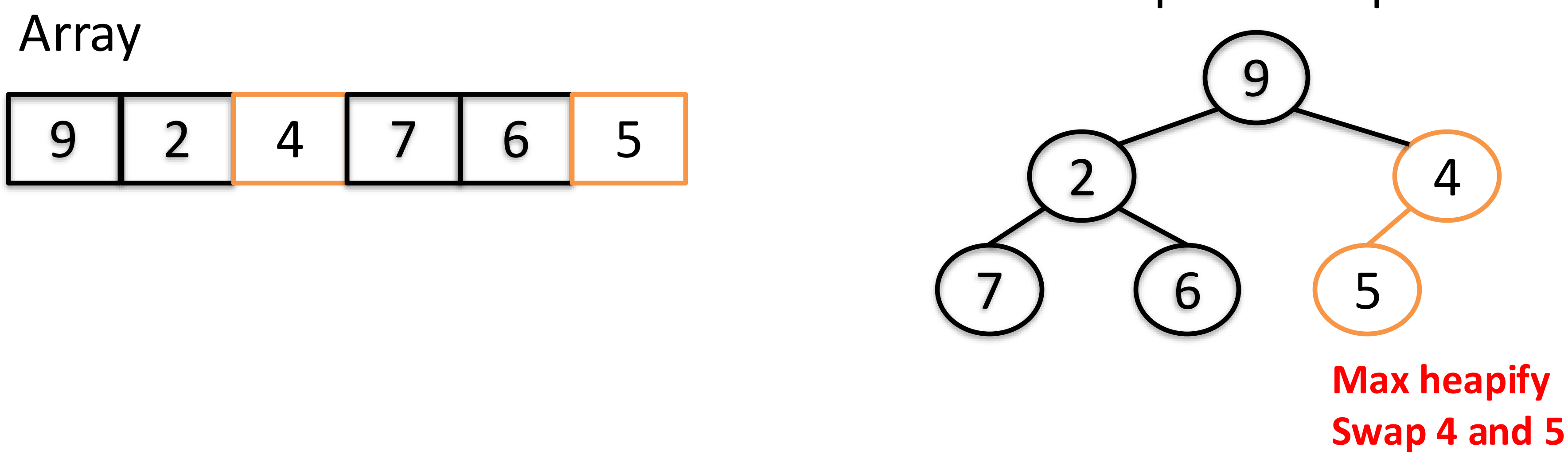
Given array in unsorted order

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



Given array in unsorted order

First build a heap in place

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

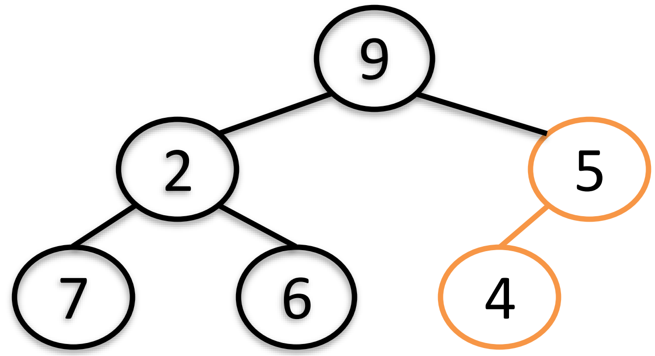
Step 1: build heap in place

Build heap given unsorted array

Array

9	2	5	7	6	4

Conceptual heap tree



Given array in unsorted order

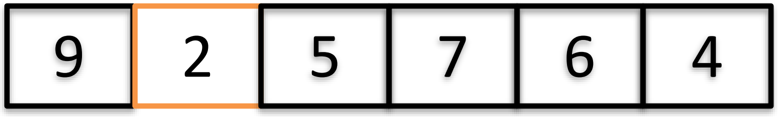
First build a heap in place

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

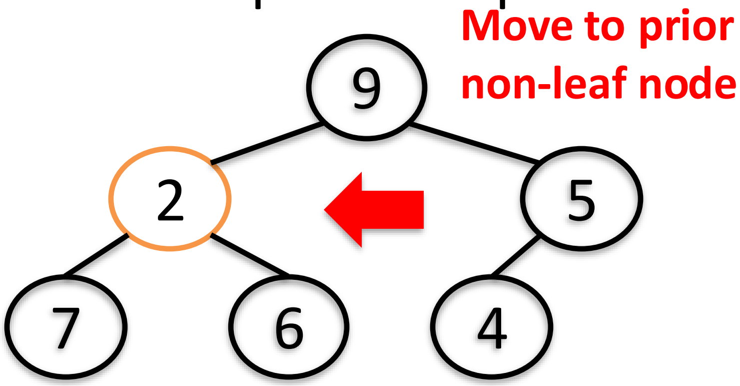
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



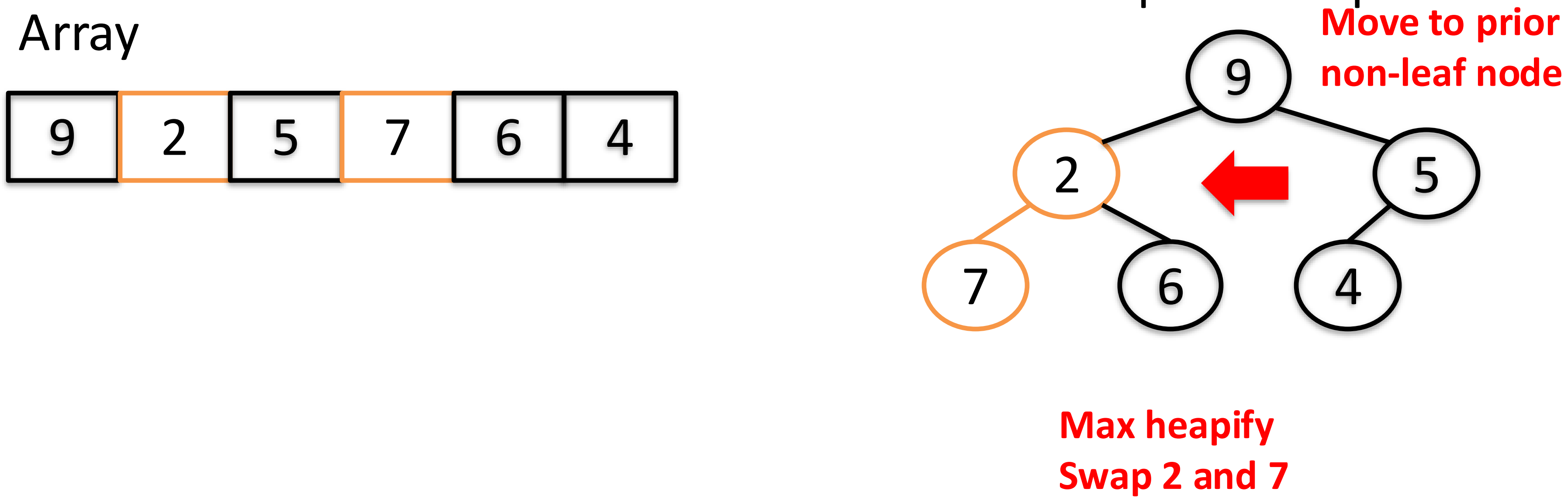
Given array in unsorted order

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



Given array in unsorted order

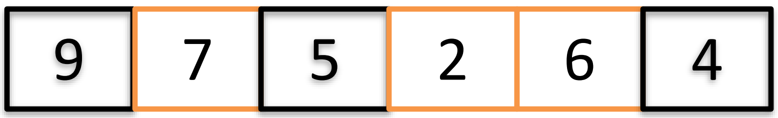
First build a heap in place

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

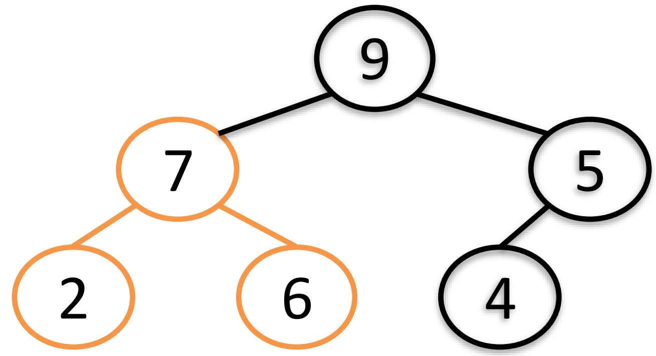
Step 1: build heap in place

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

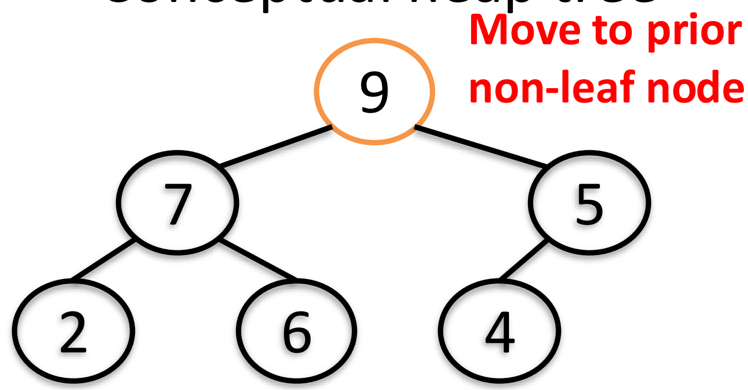
Step 1: build heap in place

Build heap given unsorted array

Array

9	7	5	2	6	4

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

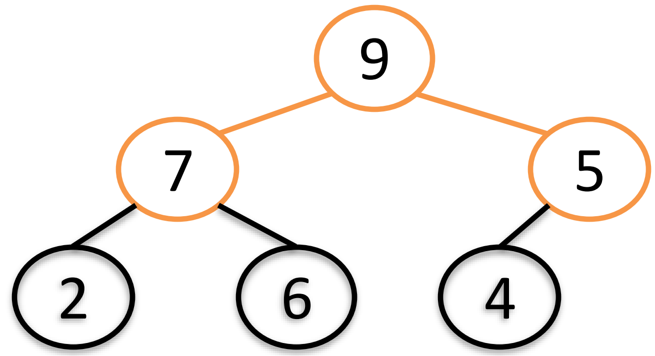
Step 1: build heap in place

Build heap given unsorted array

Array

9	7	5	2	6	4

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

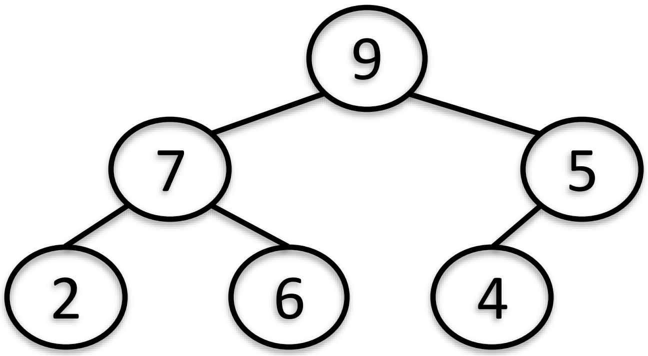
Step 1: build heap in place

Build heap given unsorted array

Array

9	7	5	2	6	4

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

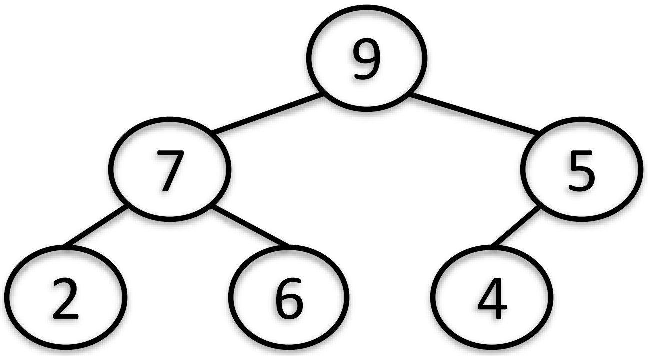
After building the heap, parents are larger than children, but items may not be sorted

Array

9	7	5	2	6	4

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

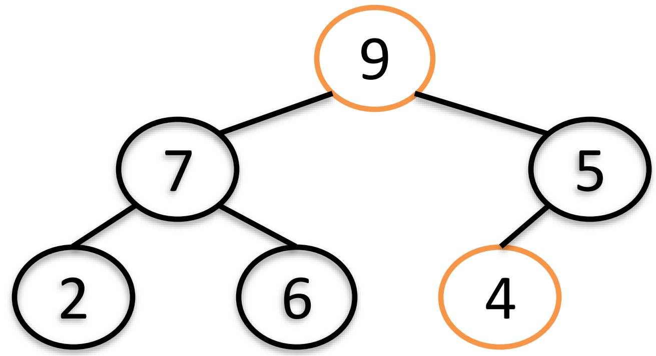
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Array

9	7	5	2	6	4

Conceptual heap tree



extractMax() = 9
Swap with last item in array

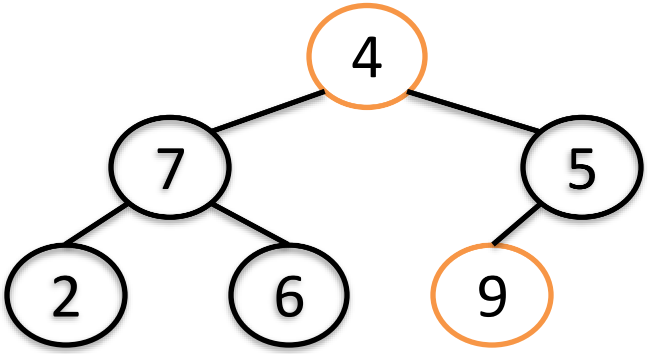
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Array

4	7	5	2	6	9

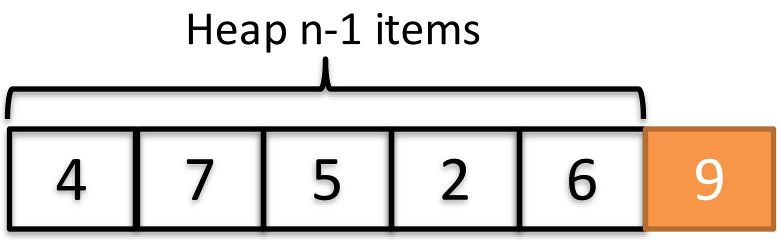
Conceptual heap tree



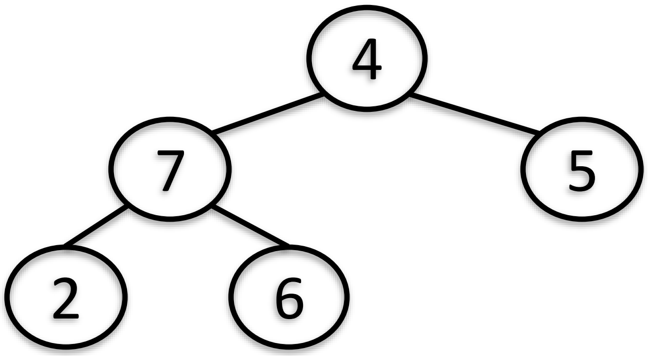
extractMax() = 9
Swap with last item in array

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

Heap on left, sorted on right



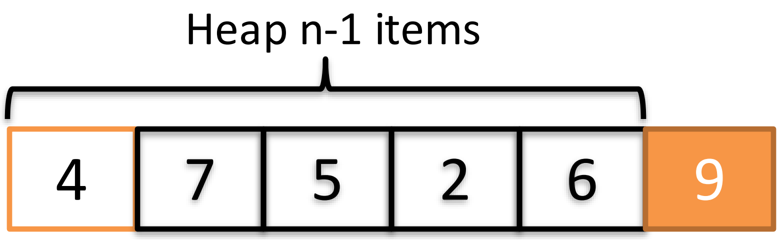
Conceptual heap tree



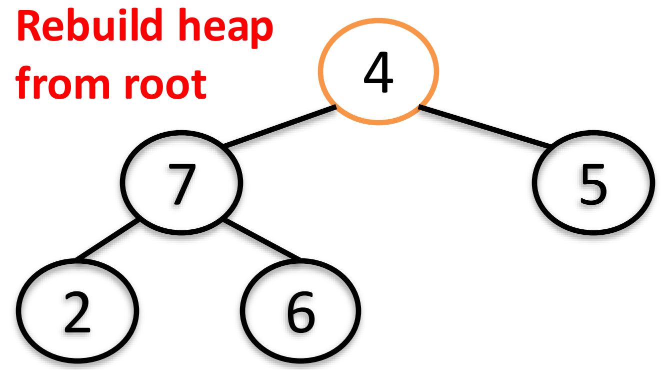
Rebuild heap on n-1 items

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

Heap on left, sorted on right



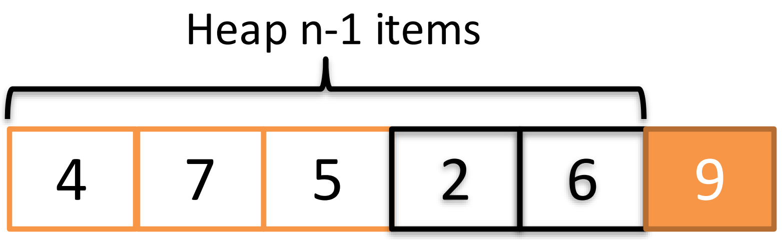
Conceptual heap tree



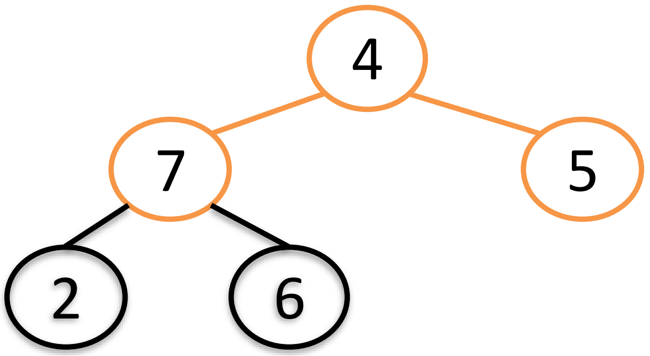
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree



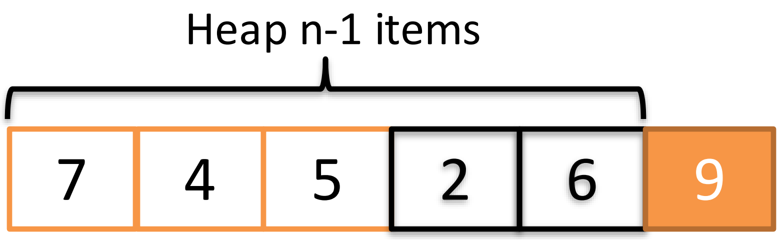
Swap 4 with largest child 7

Max heapify
Swap 7 and 4

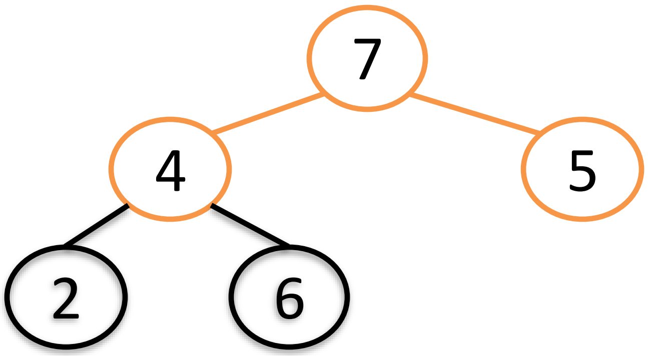
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree



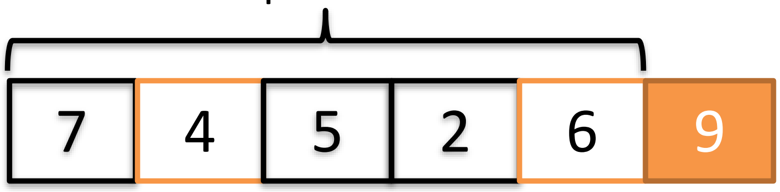
Max heapify
Swap 7 and 4

Rebuild heap on n-1 items

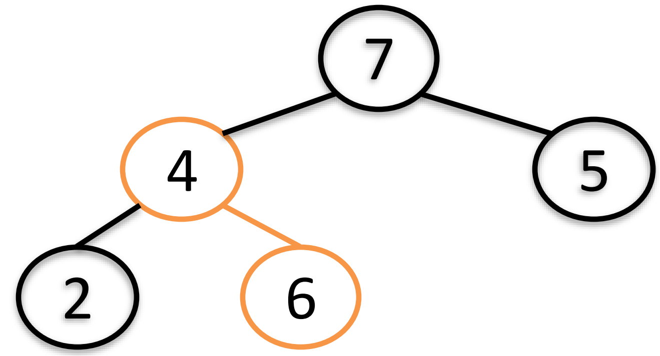
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Heap n-1 items



Conceptual heap tree



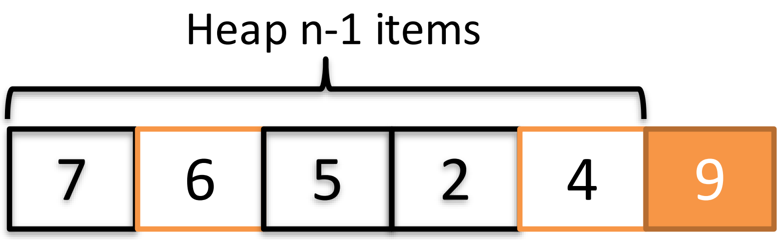
Swap 4 with largest child 6

Max heapify
Swap 4 and 6

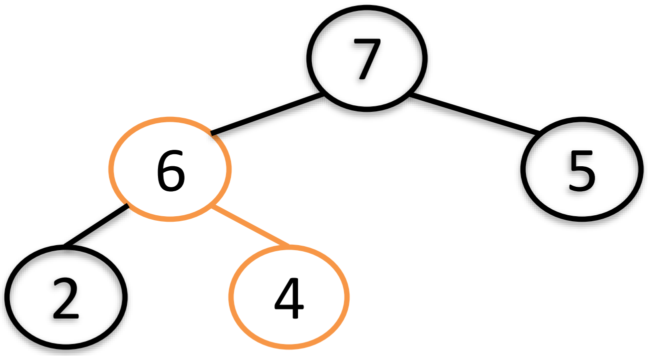
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree

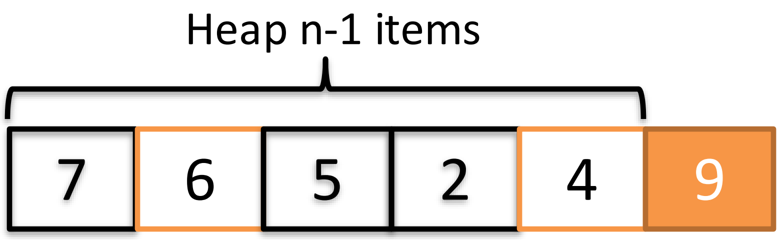


Max heapify
Swap 4 and 6

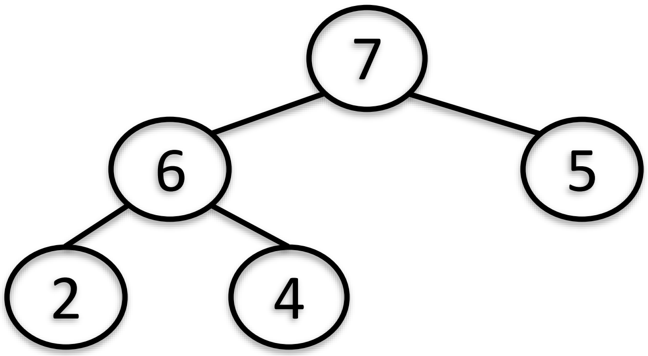
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree

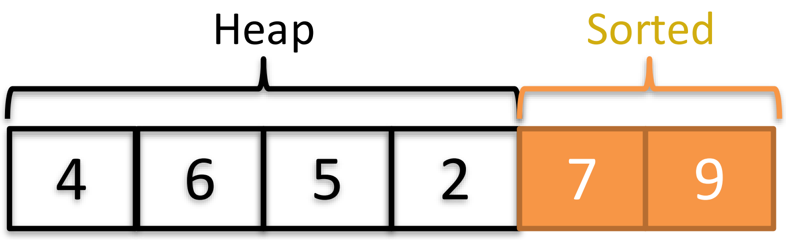


Heap built

Rebuild heap on n-1 items

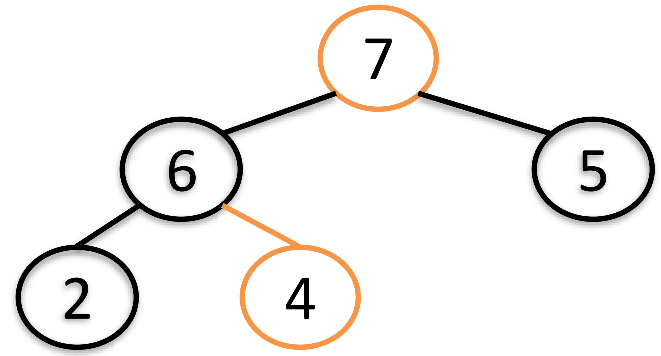
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

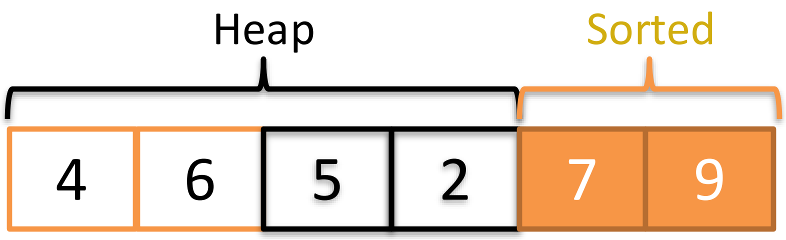


extractMax() = 7

Swap with last item in array

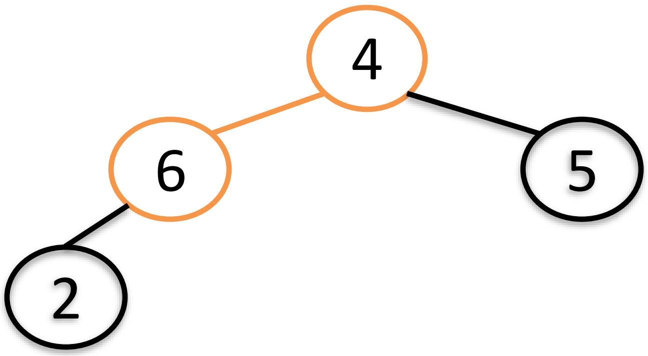
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree



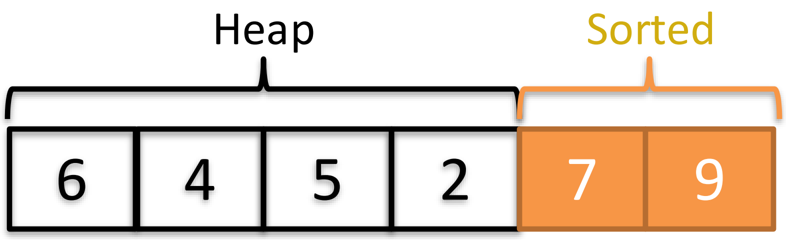
Swap 4 with largest child 6

Max heapify
Swap 4 and 6

Rebuild heap on n-2 items

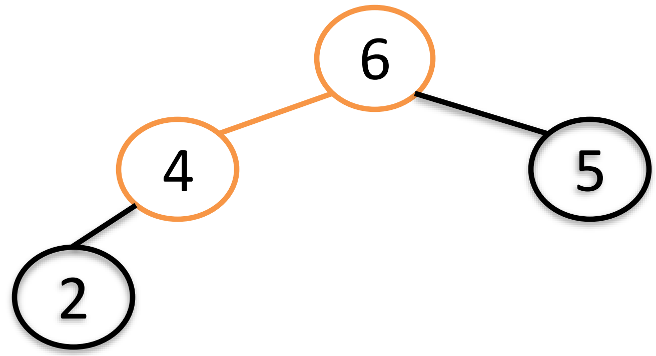
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

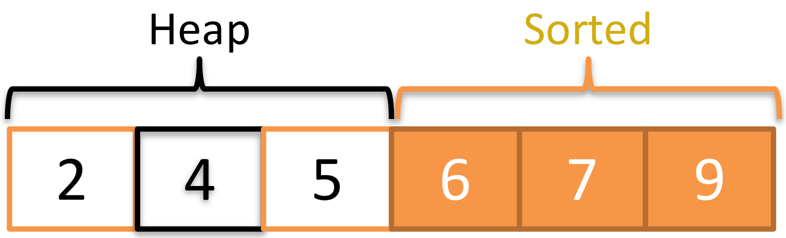


Heap built

Rebuild heap on n-2 items

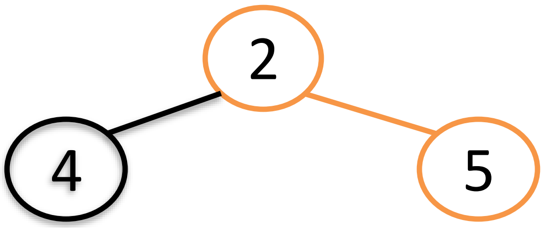
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree



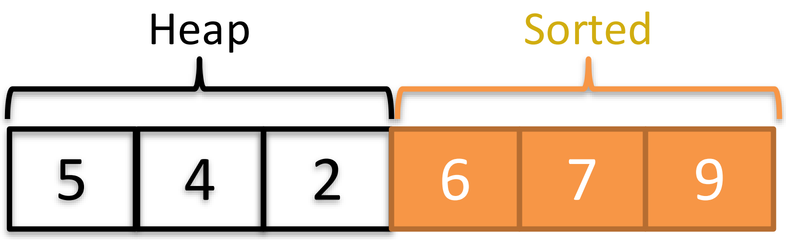
Swap 2 with largest child 5

Max heapify
Swap 5 and 2

extractMax() = 6
Swap with last item in array

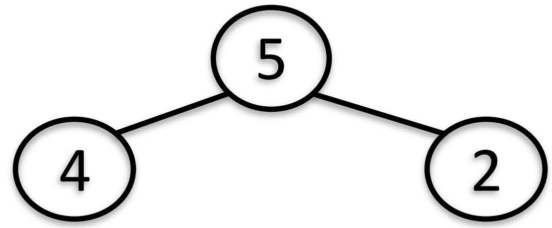
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

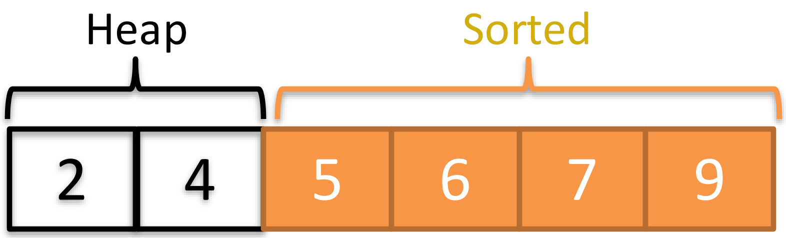


Heap built

Rebuild heap on n-3 items

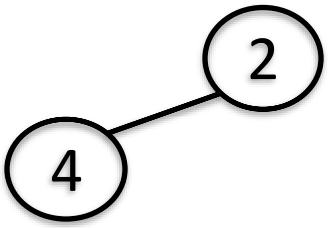
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

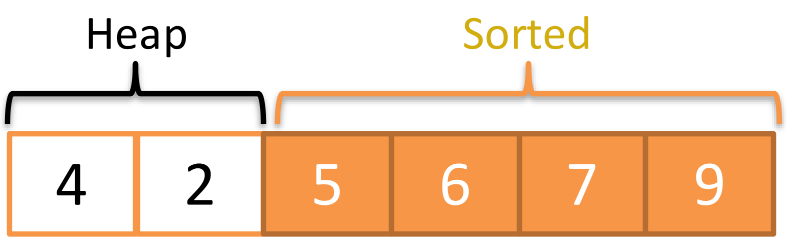


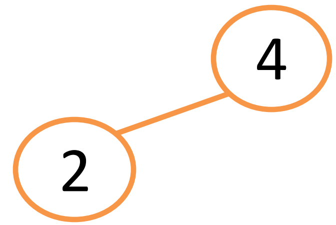
extractMax() = 5
Swap with last item in array

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

Heap on left, sorted on right

Conceptual heap tree





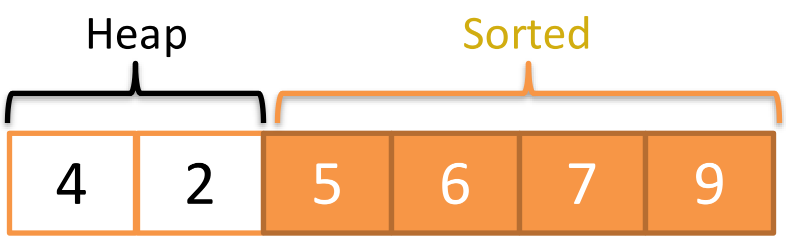
Heap array

Max heapify
Swap 4 and 2

Rebuild heap on n-4 items

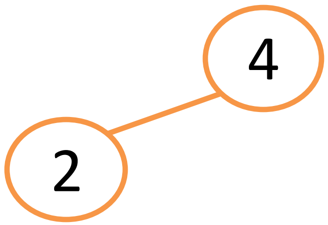
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

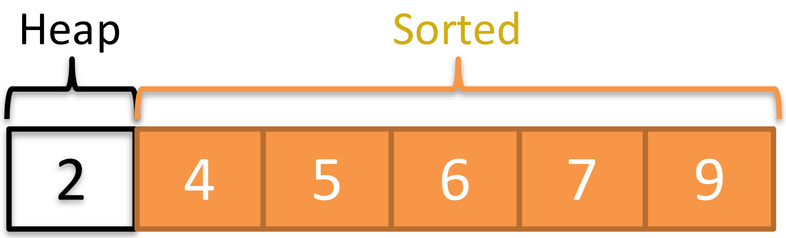


Heap built

Rebuild heap on n-4 items

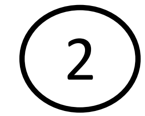
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

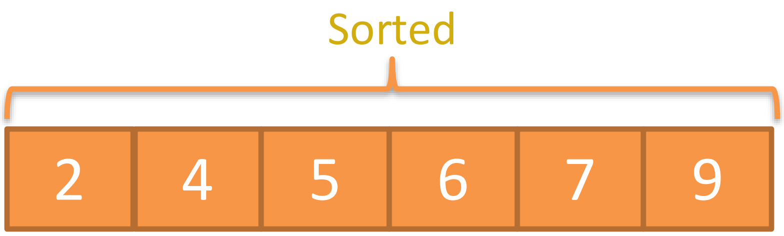


extractMax() = 4
Swap with last item in array

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

Heap on left, sorted on right

Conceptual heap tree



Heap array

Done

Items sorted in place

No extra memory used

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

Code very similar to HeapMinPriorityQueue.java

```java
public class Heapsort<E extends Comparable<E>> {
    //no constructor!  instead we sort arrays in place

    /**
     * Sort the array a[0..n-1] *inplace* using the heapsort algorithm.
     */
    public void sort(E[] a, int n) {
        heapsort(a, n - 1);
    }

    /**
     * Sort the array a[0..lastLeaf] by the heapsort algorithm.
     */
    private void heapsort(E[] a, int lastLeaf) {
        // First, turn the array a[0..lastLeaf] into a max-heap.
        buildMaxHeap(a, lastLeaf);

        // Once the array is a max-heap, repeatedly swap the root
        // with the last leaf, putting the largest remaining element
        // in the last leaf's position, declare this last leaf to no
        // longer be in the heap, and then fix up the heap.
        while (lastLeaf > 0) {
            swap(a, 0, lastLeaf);       // swap the root with the last leaf
            lastLeaf--;                 // the last leaf is no longer in the heap
            maxHeapify(a, 0, lastLeaf); // fix up what's left
        }
    }
```

- **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, rebuild heap…

```java
    private void maxHeapify(E[] a, int i, int lastLeaf) {
        int left = leftChild(i);      // index of node i's left child
        int right = rightChild(i);    // index of node i's right chil
        int largest;                  // will hold the index of the r

        // Is there a left child and, if so, does the left child have
        if (left <= lastLeaf && a[left].compareTo(a[i]) > 0)
            largest = left; // yes, so the left child is the largest
        else
            largest = i;    // no, so node i is the largest so far

        // Is there a right child and, if so, does the right child ha
        // element larger than the larger of node i and the left chil
        if (right <= lastLeaf && a[right].compareTo(a[largest]) > 0)
            largest = right; // yes, so the right child is the larges

        /*
         * If node i holds an element larger than both the left and r
         * children, then the max-heap property already held, and we
         * nothing more. Otherwise, we need to swap node i with the l
         * of the two children, and then recurse down the heap from t
         * child.
         */
        if (largest != i) {
            swap(a, i, largest);
            maxHeapify(a, largest, lastLeaf);
        }
    }

    /**
     * Form array a[0..lastLeaf] into a max-heap.
     */
    private void buildMaxHeap(E[] a, int lastLeaf) {
        int lastNonLeaf = (lastLeaf - 1) / 2; // nodes lastNonLeaf+1
        for (int j = lastNonLeaf; j >= 0; j--)
            maxHeapify(a, j, lastLeaf);
    }
```

- **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**

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)* + $O(n \log_2 n) = O(n \log_2 n)$