CS 10:
Problem solving via Object Oriented Programming

Prioritizing 2

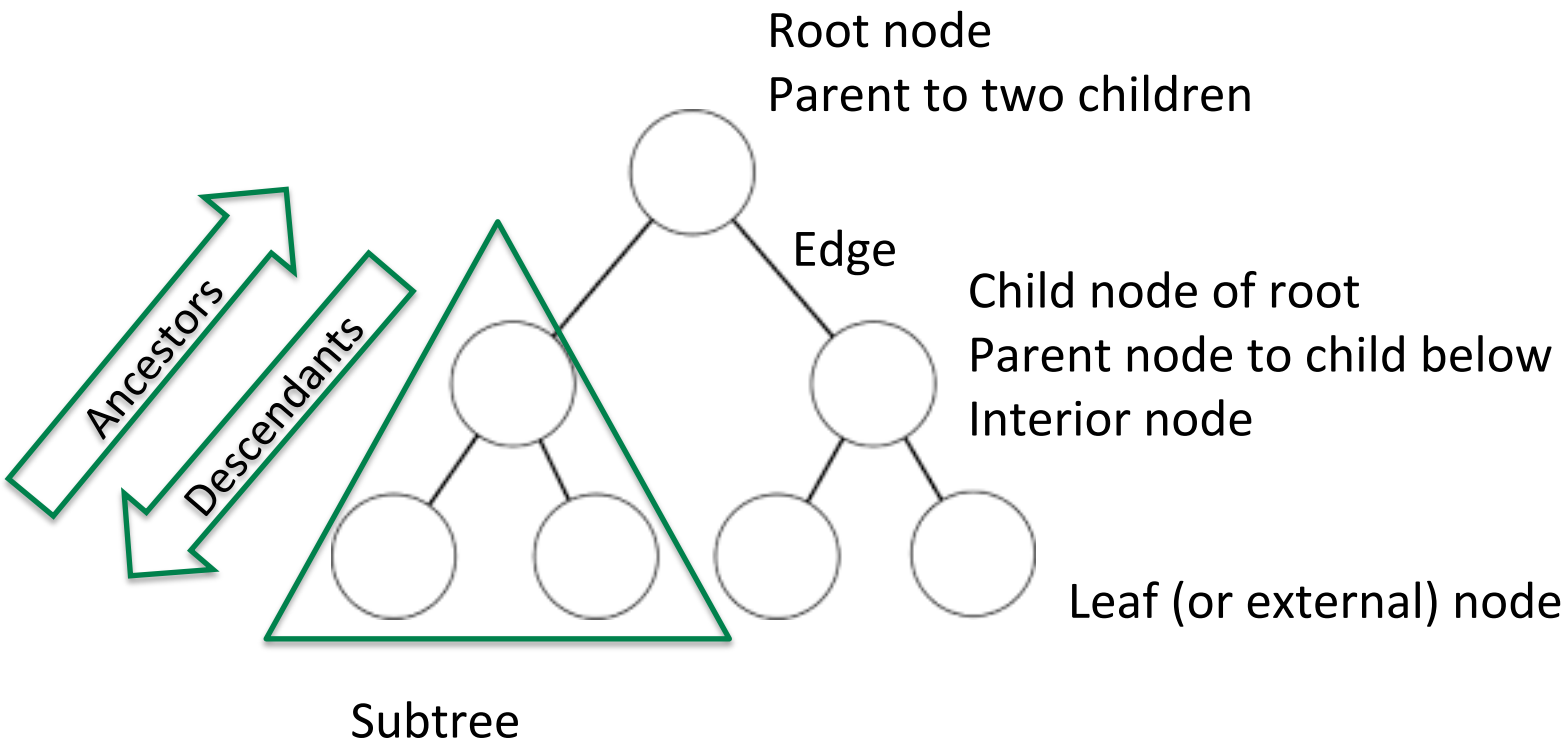
Agenda

1. Heaps

2. Heap sort

Heaps are based on Binary Trees

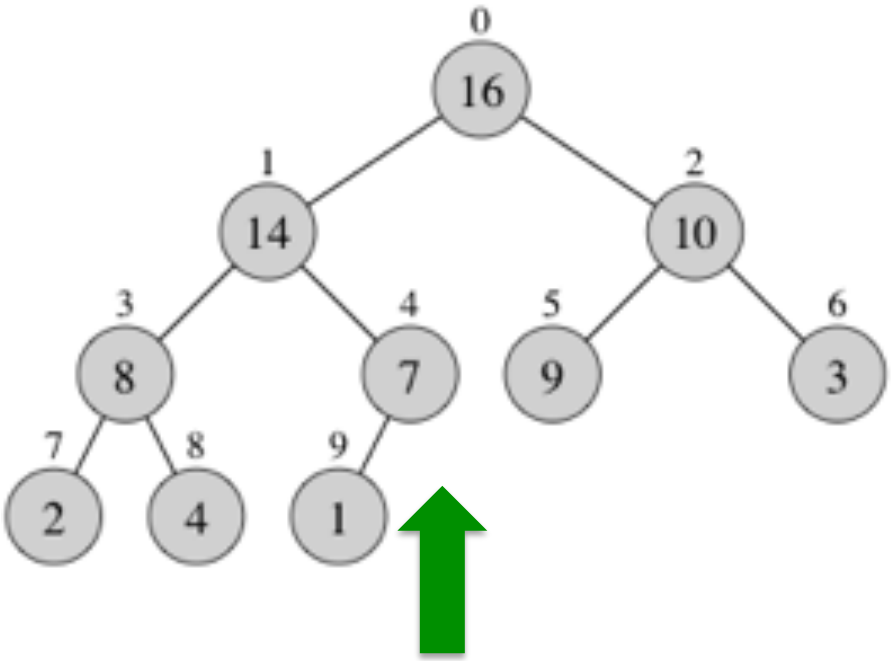
Tree data structure



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



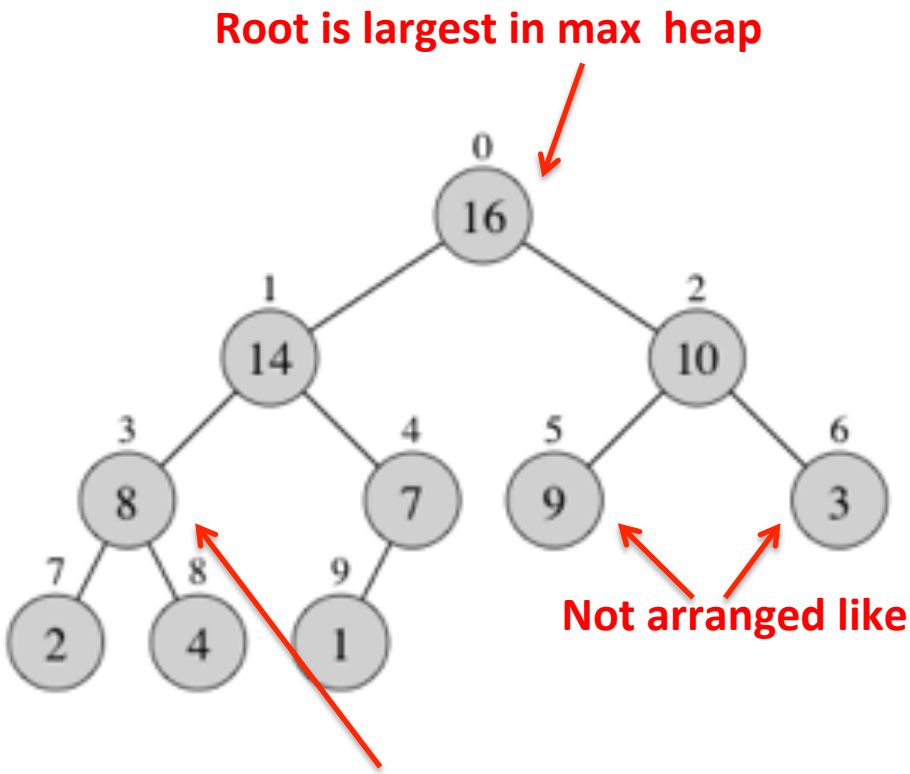
Next node added here

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
- Makes height as small as possible – $\log_2 n$
- Prevents “vines”

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

Order property keeps nodes organized



Order property

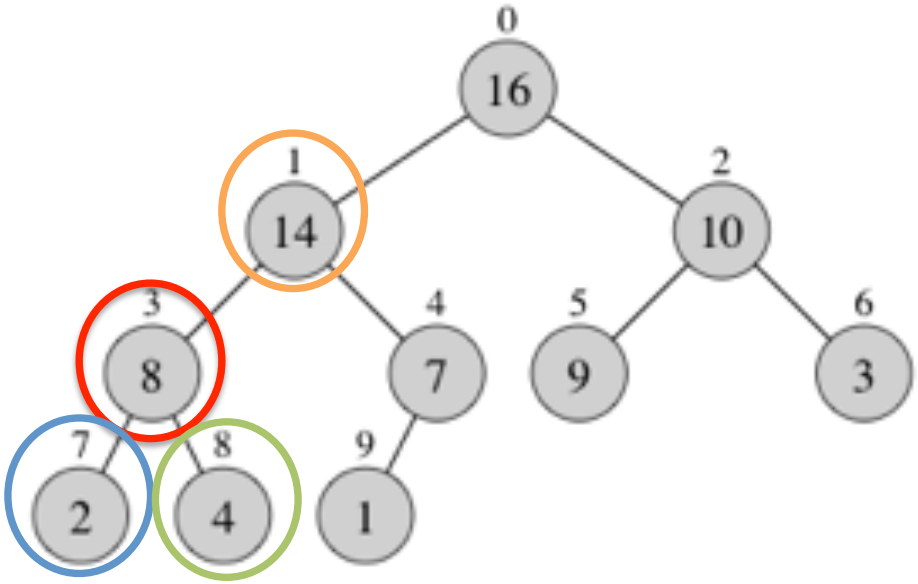
Reverse inequality for min heap

- ∀ nodes $i \neq$ 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

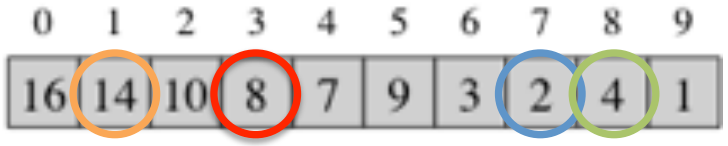
Array implementation

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*

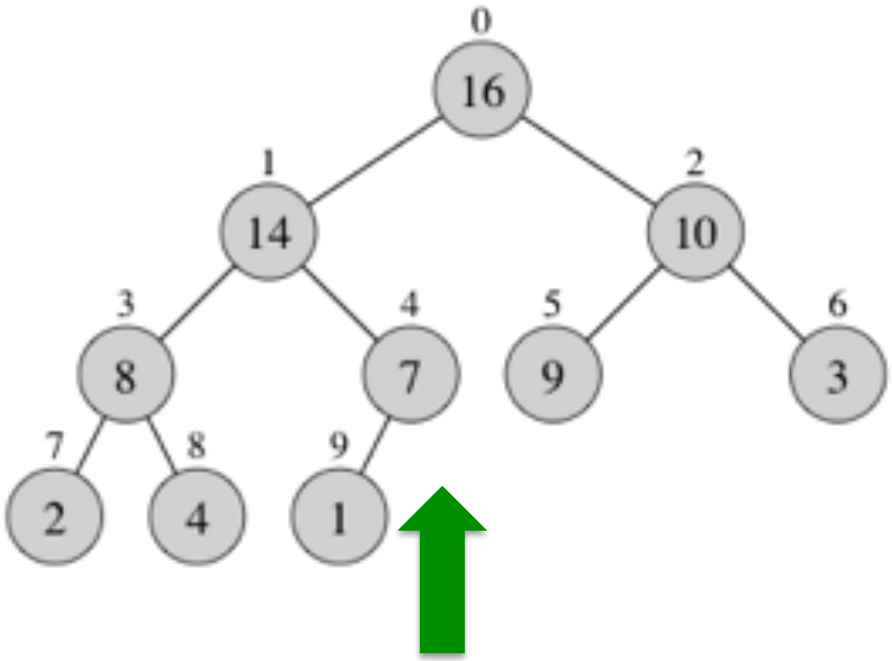


Node 3 containing 8

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

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

Max heap insert



Next node added here

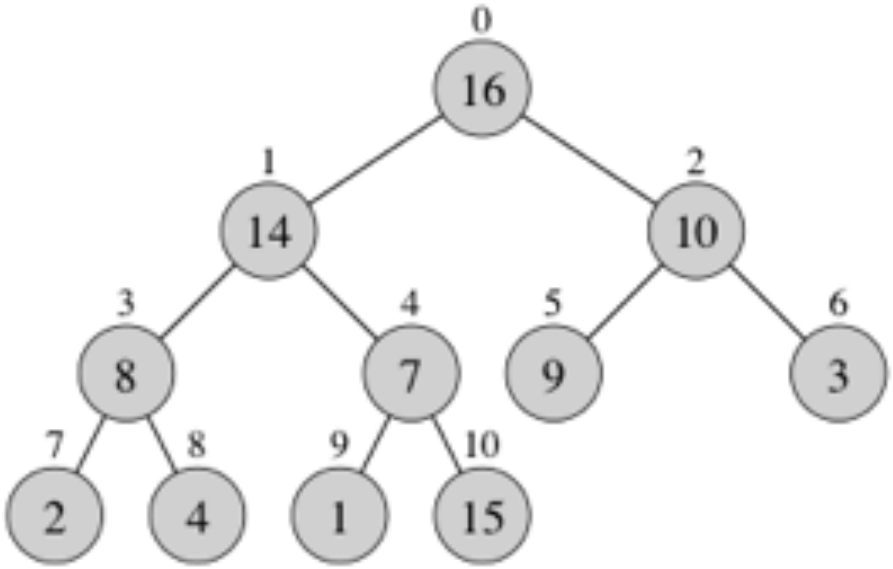
Insert 15

- 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

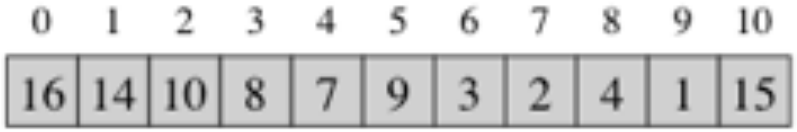
Inserting into max heap must keep both shape and order properties intact

Max heap insert



Insert 15

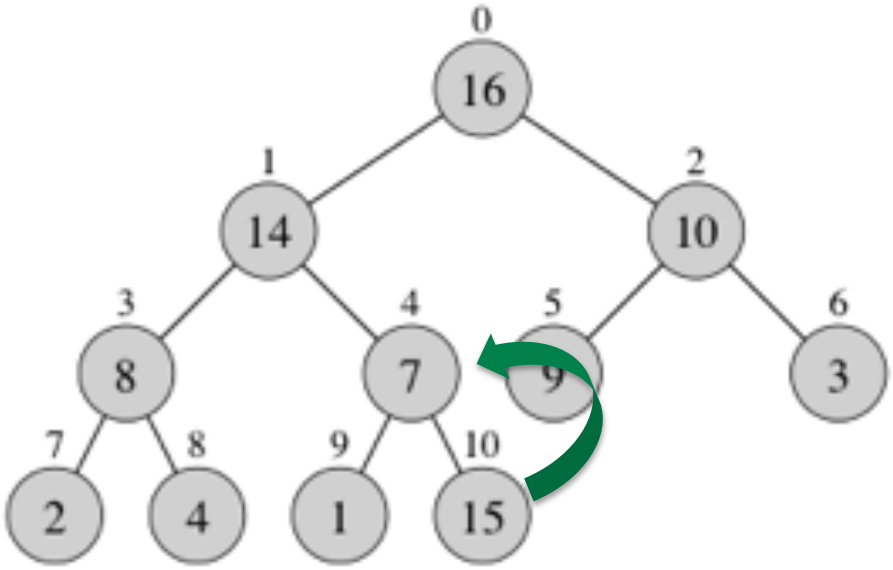
- Shape property: fill in next spot in left to right order (index i=10)



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

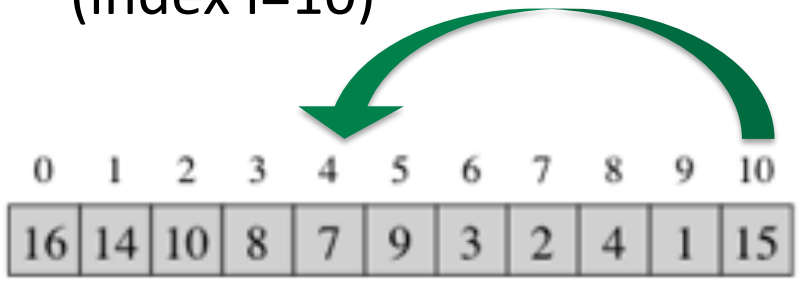
Inserting into max heap must keep both shape and order properties intact

Max heap insert



Insert 15

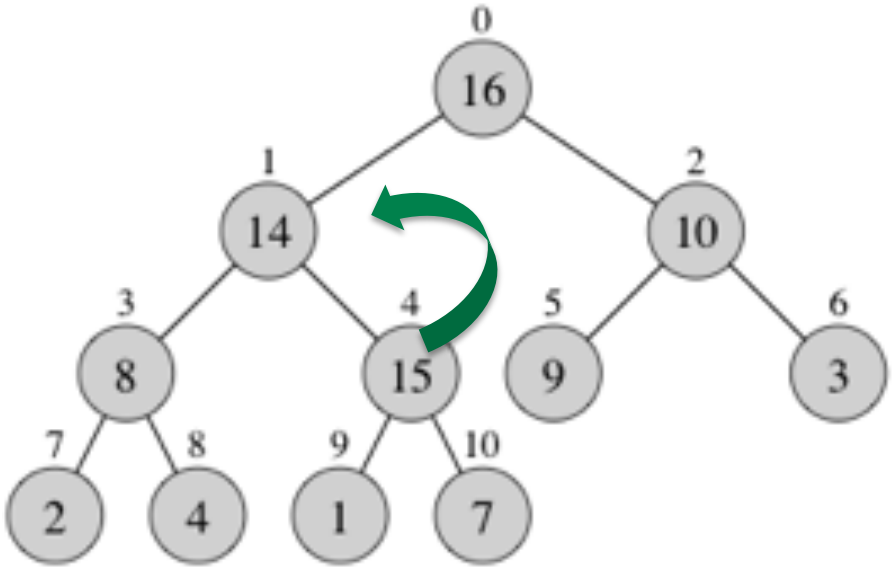
- Shape property: fill in next spot in left to right order (index i=10)



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

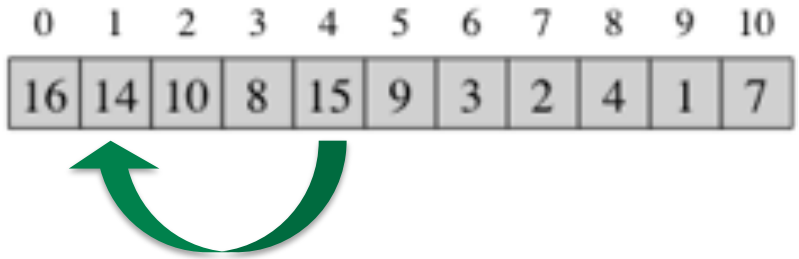
We may have to swap multiple times to get both heap properties

Max heap insert



Insert 15

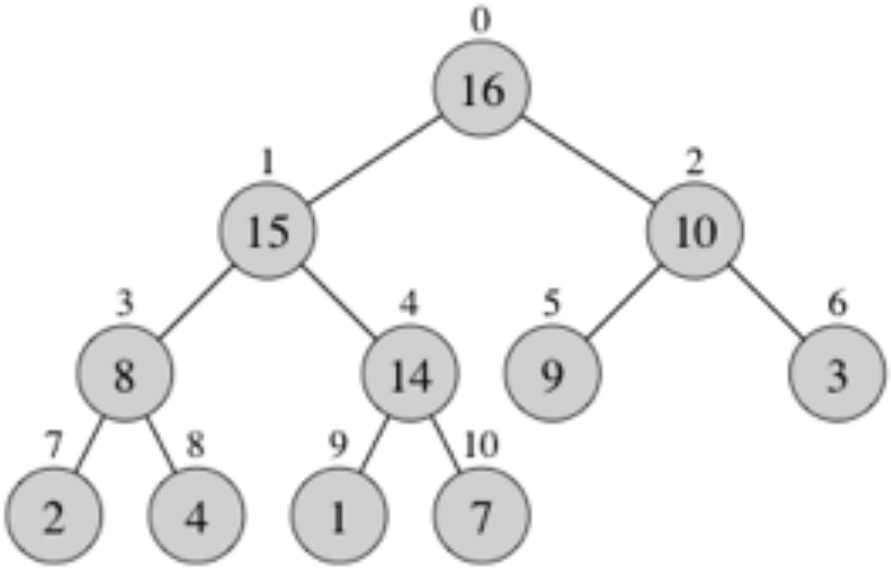
- Shape property: good!
- Order property: parent must be larger than children, not met



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

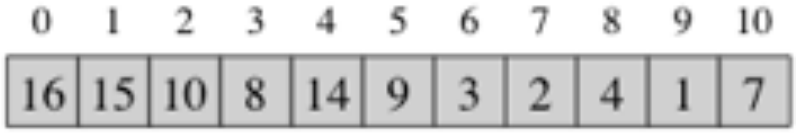
Eventually we will find a spot for the newly inserted item, even if that spot is the root

Max heap insert



Insert 15

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

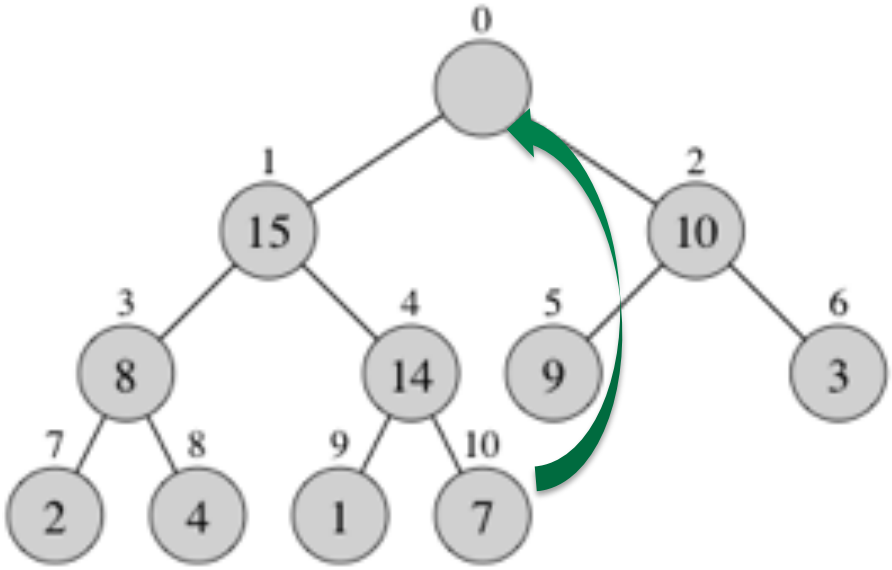


General rule

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

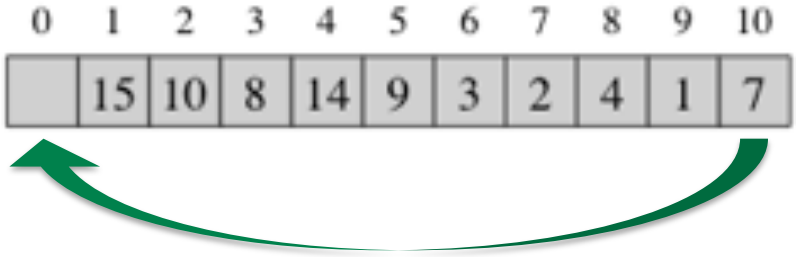
extractMax means removing the root, but that leaves a hole

extractMax



extractMax -> 16

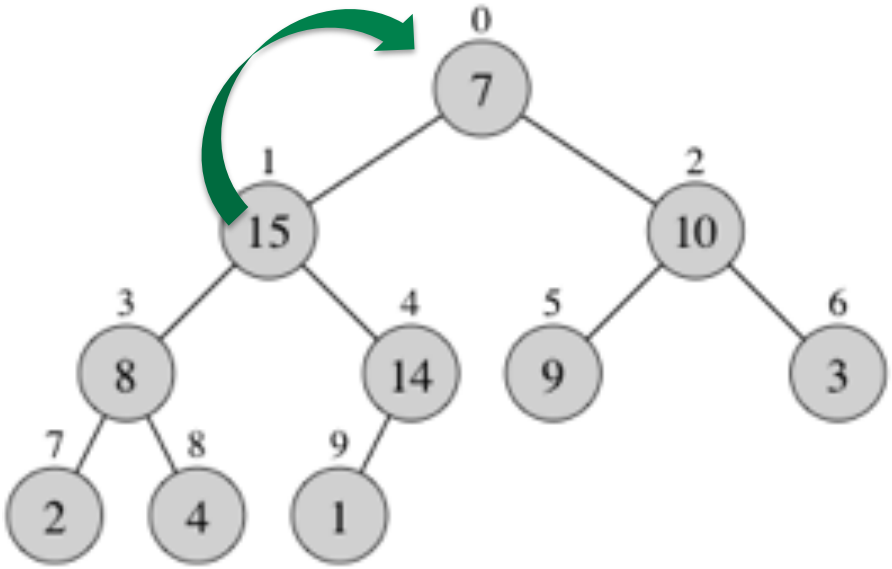
- Max position is at root (index 0)
- Removing it leaves a hole, violating shape property



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

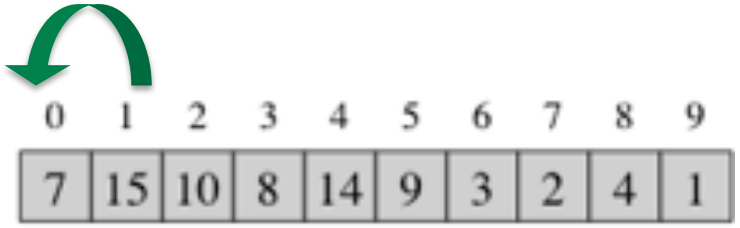
Moving bottom right node to root restores shape, but not order property

extractMax



After swap

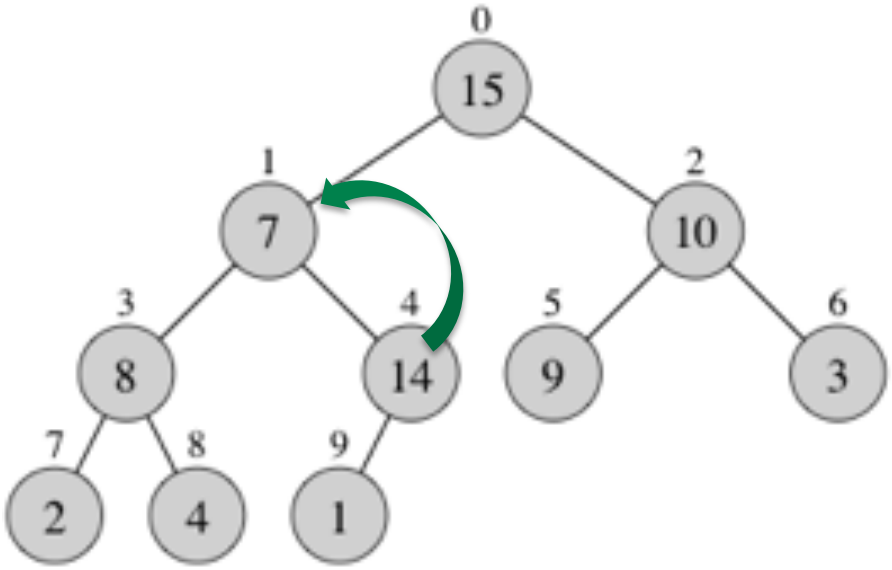
- Shape property: good!
- Order property: want max at root, but do not have that



- Left and right subtrees still valid
- Swap root with larger child
- Will be greater than new root and everything in subtree

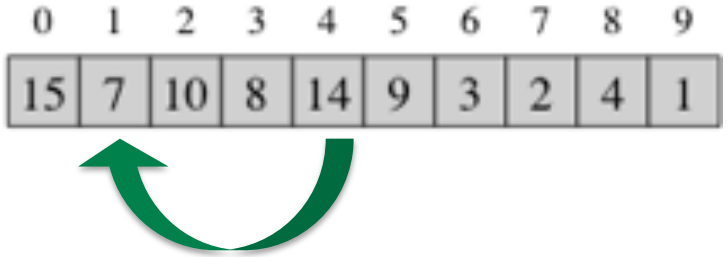
May need multiple swaps to restore order property

extractMax



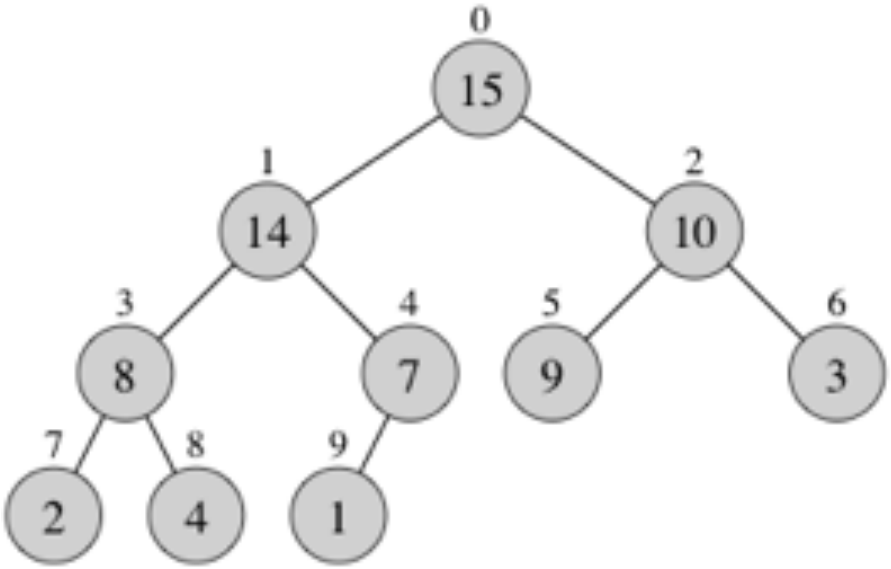
After swap 15 and 7

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



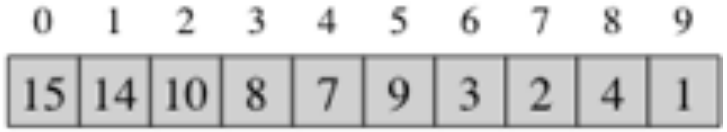
Stop once order property is restored

extractMax



After swap 7 and 14

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



Can implement heap-based Min Priority Queue using an ArrayList

HeapMinPriorityQueue.java

Heap elements extend Comparable

ArrayList called *heap* will hold the heap

```
 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
```

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     }
```

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

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 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    }
```

- **Swap if not root (loc==0) and element < parent**
- **Continue to "bubble up" inserted node until reach root or parent < element**
- **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

Run time analysis shows the heap implementation is better than previous

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

isEmpty()

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

Run time analysis shows the heap implementation is better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
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 the heap implementation is better than previous

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

minimum()

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

Run time analysis shows the heap implementation is better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	O(1)	O(1)	O(1)
insert	$O(\log_2 n)$	O(1)	**O(n)**
minimum	O(1)	**O(n)**	O(1)
extractMin	$O(\log_2 n)$	**O(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, O(n), remove, then move all other items; O(n)
- **Sorted ArrayList:** return last item in ArrayList; O(1)

Run time analysis shows the heap implementation is better than previous

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	O(1)	O(1)	O(1)
insert	$O(\log_2 n)$	O(1)	**O(n)**
minimum	O(1)	**O(n)**	O(1)
extractMin	$O(\log_2 n)$	**O(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. Heaps
2. Heap sort

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

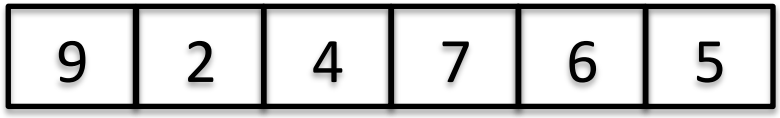
2. Extract max (index 0) and swap with item at end of array, then rebuild heap not considering last item

Does not require additional memory to sort

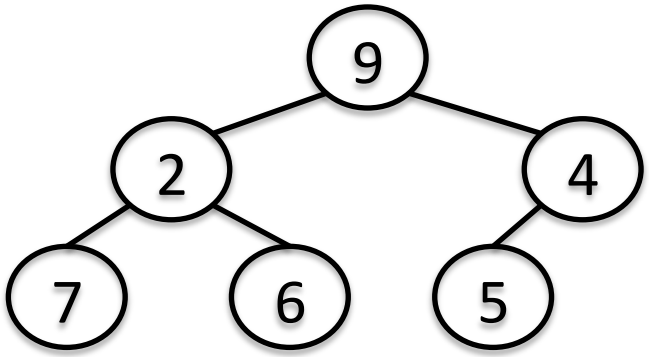
Step 1: build heap in place

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

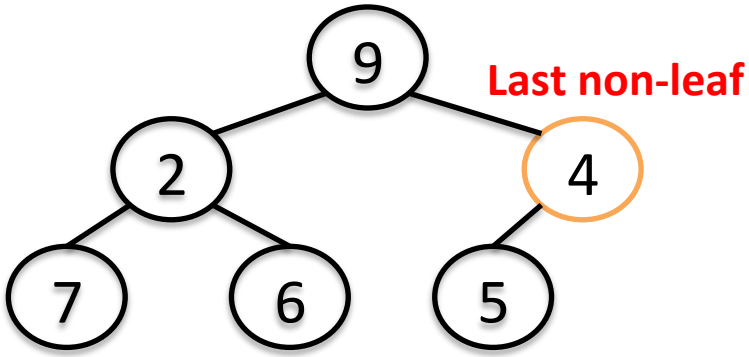
Non heap!

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

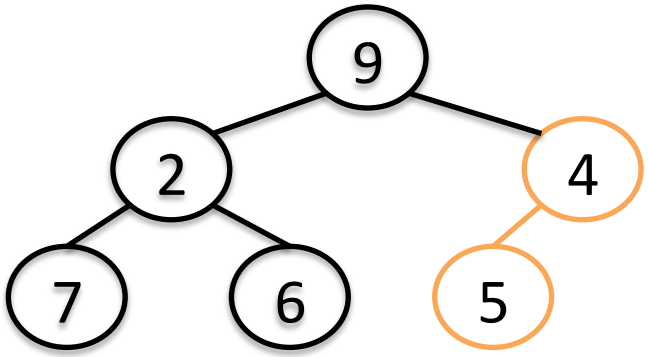
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Max heapify
Swap 4 and 5

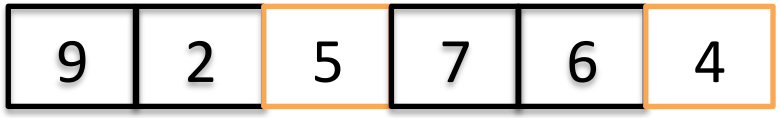
Given array in unsorted order
First build a heap in place

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

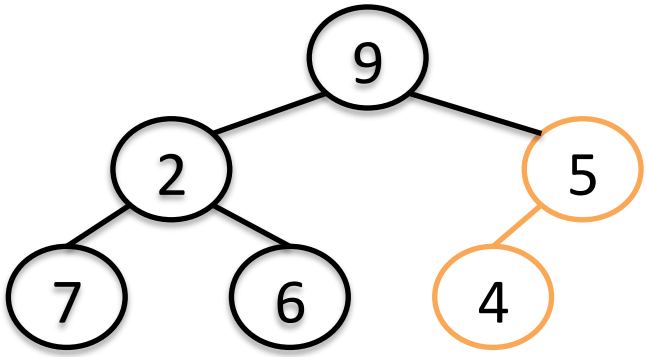
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Given array in unsorted order

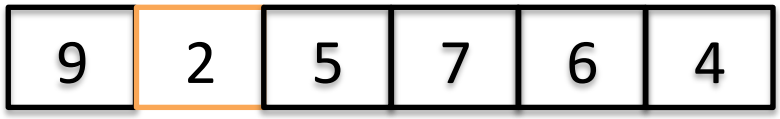
First build a heap in place

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

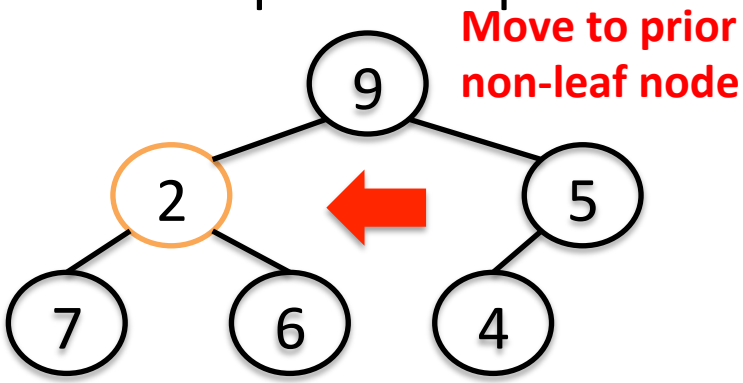
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Given array in unsorted order

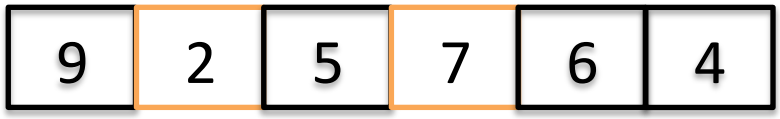
First build a heap in place

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

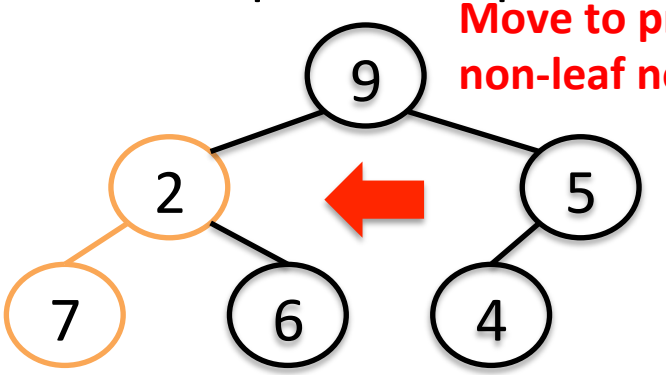
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Move to prior non-leaf node

Max heapify
Swap 2 and 7

Given array in unsorted order
First build a heap in place

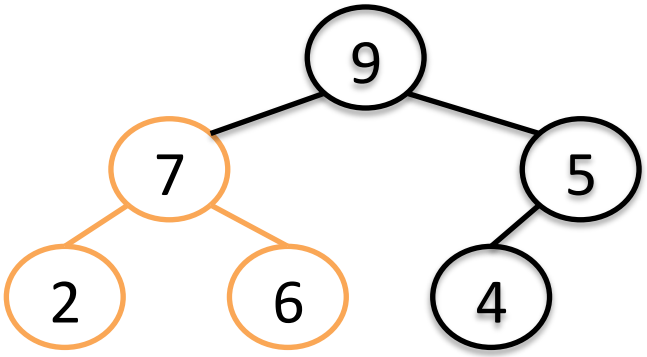
- Start at last non-leaf and heapify
- Repeat for other non-leaf nodes

Step 1: build heap in place

Build heap given unsorted array

Array

Conceptual heap tree



Max heapify
Swap 2 and 7

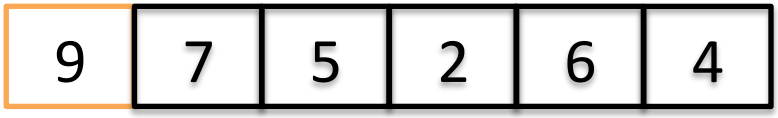
Given array in unsorted order
First build a heap in place

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

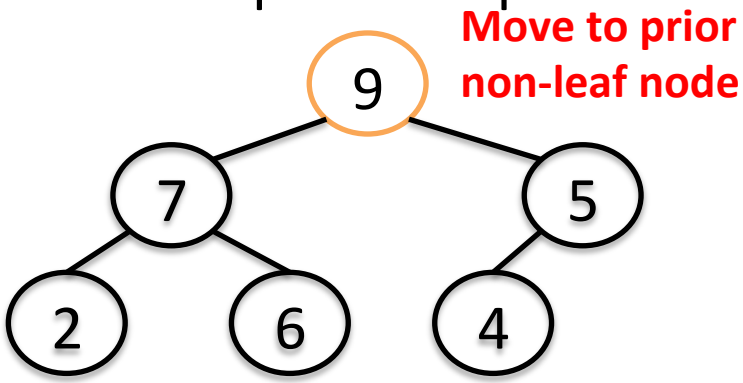
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Given array in unsorted order

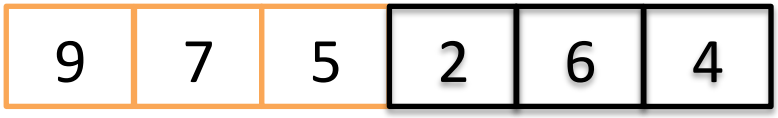
First build a heap in place

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

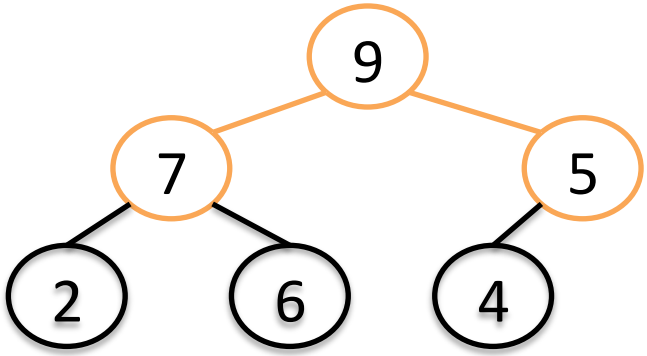
Step 1: build heap in place

Build heap given unsorted array

Array



Conceptual heap tree



Max heapify
In order, no need to swap

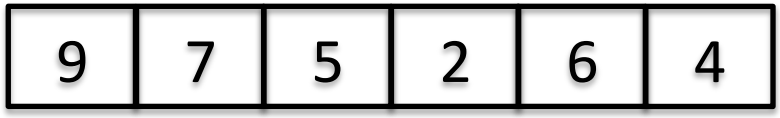
Given array in unsorted order
First build a heap in place

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

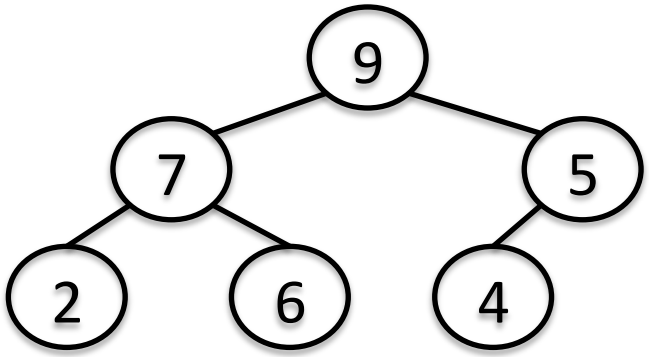
Step 1: build heap in place

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

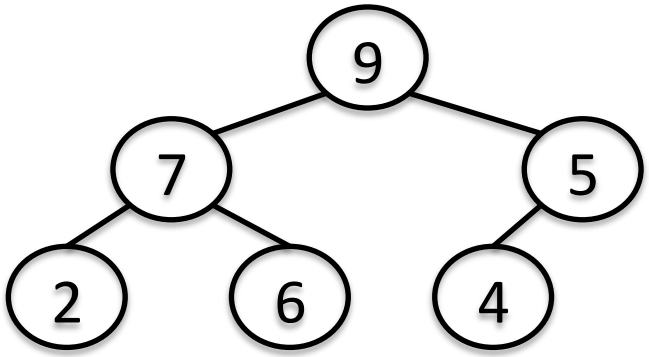
After building the heap, parents are larger than children, but items may not be sorted

Array

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

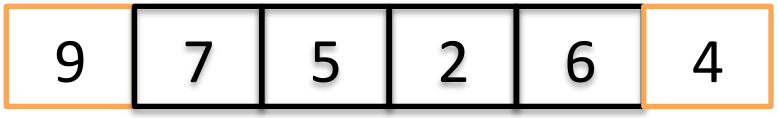
Traversing tree doesn't work either

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

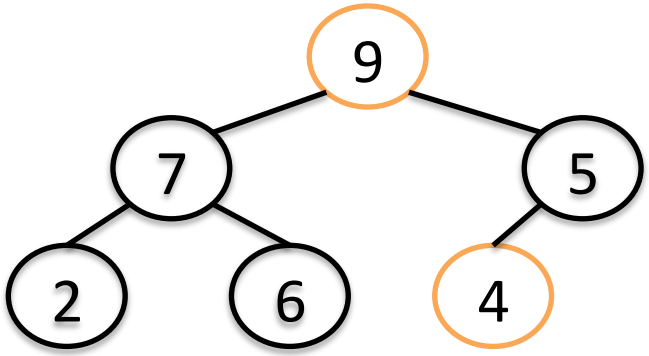
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Array



Conceptual heap tree



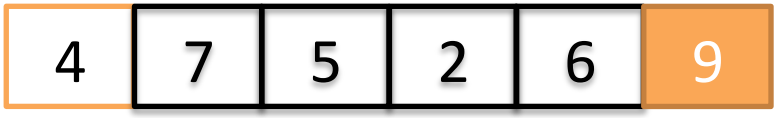
extractMax() = 9

Swap with last item in array

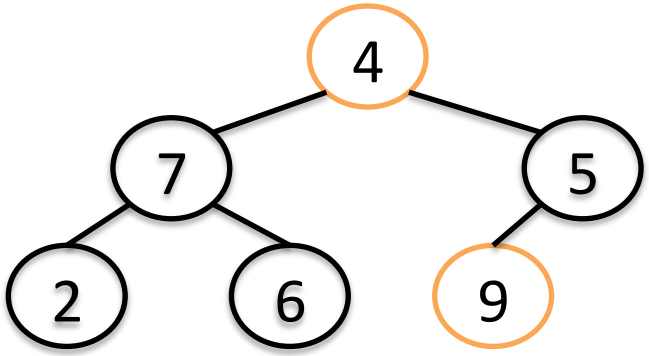
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right

Array



Conceptual heap tree

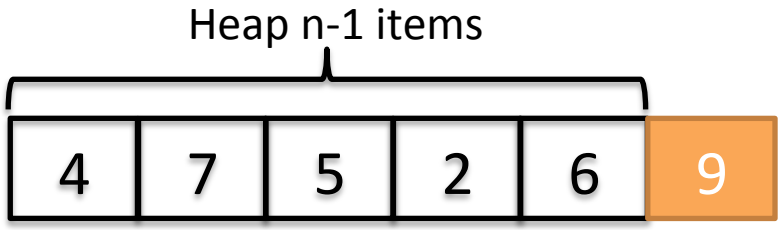


extractMax() = 9

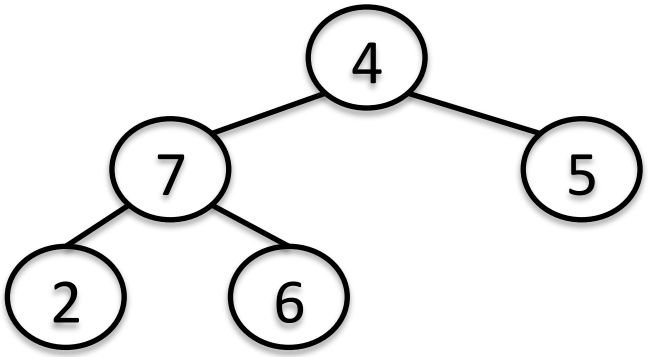
Swap with last item in array

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

Heap on left, sorted on right



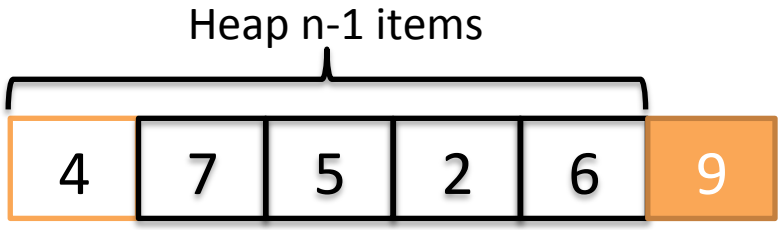
Conceptual heap tree



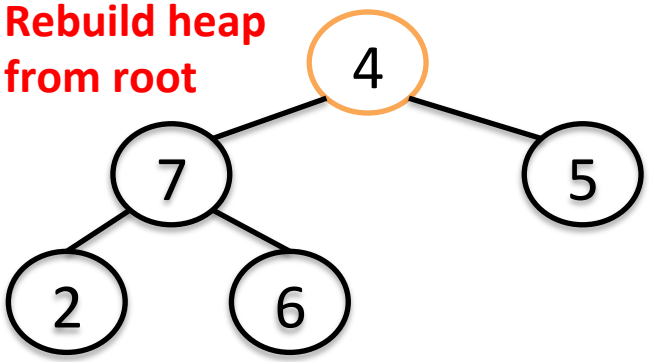
Rebuild heap on n-1 items

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

Heap on left, sorted on right



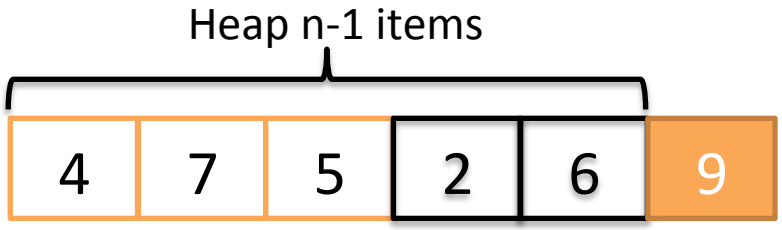
Conceptual heap tree



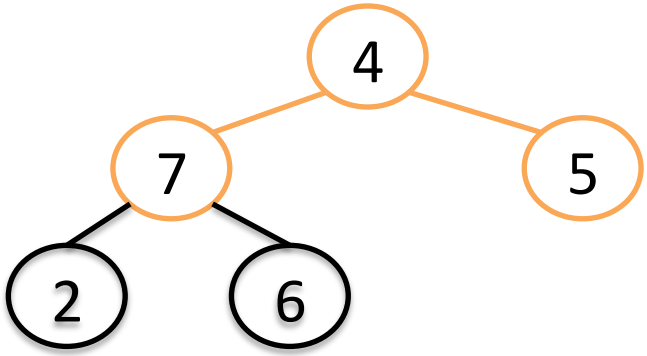
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree



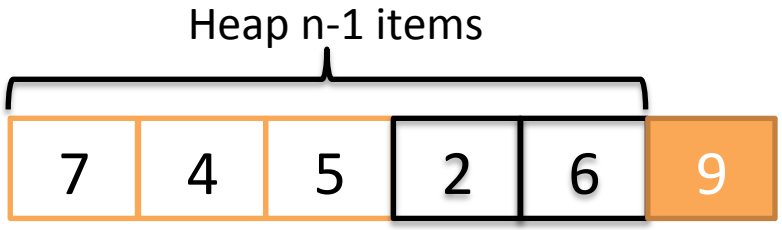
Swap 4 with largest child 7

Max heapify
Swap 7 and 4

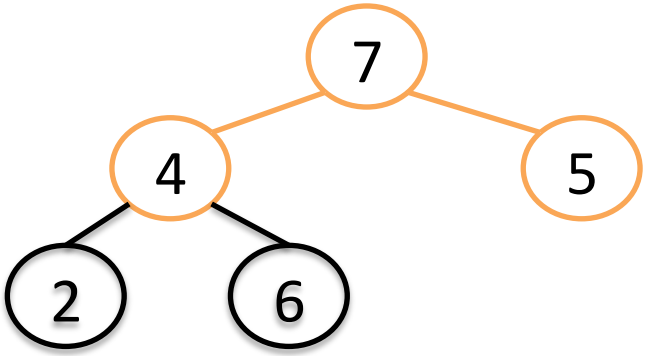
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree

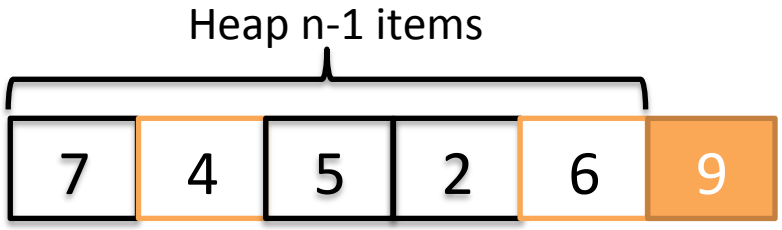


Max heapify
Swap 7 and 4

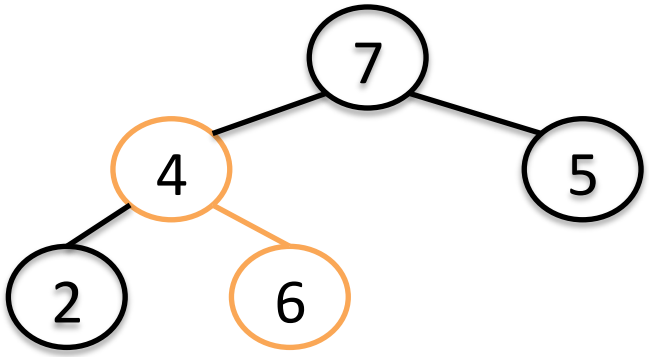
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree



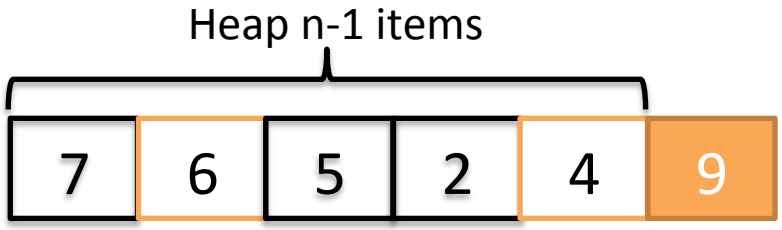
Swap 4 with largest child 6

Max heapify
Swap 4 and 6

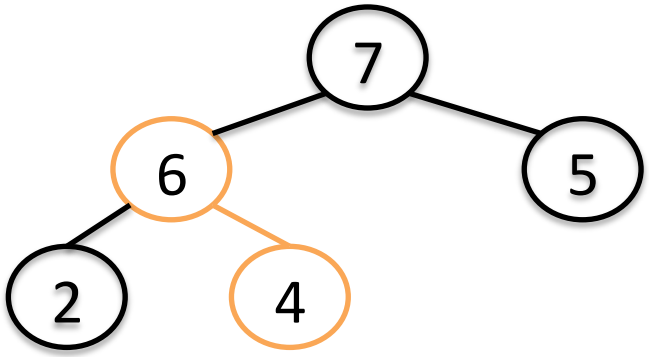
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree

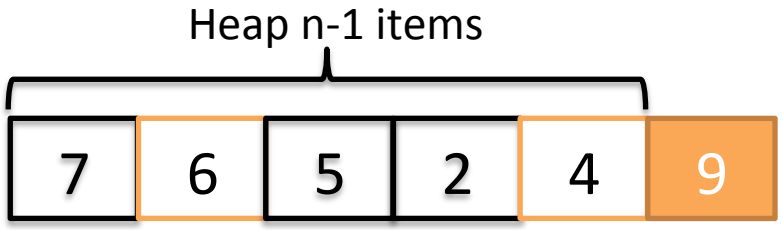


Max heapify
Swap 4 and 6

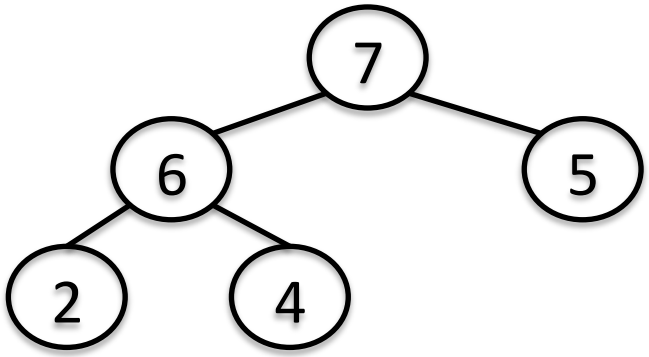
Rebuild heap on n-1 items

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

Heap on left, sorted on right



Conceptual heap tree

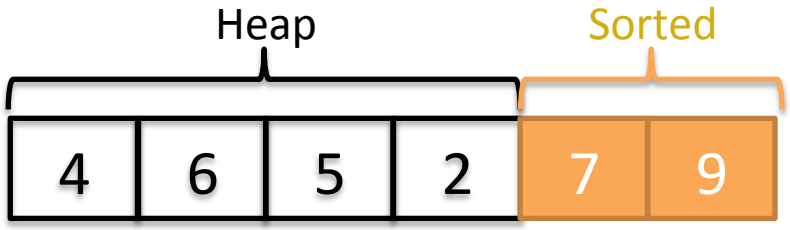


Heap built

Rebuild heap on n-1 items

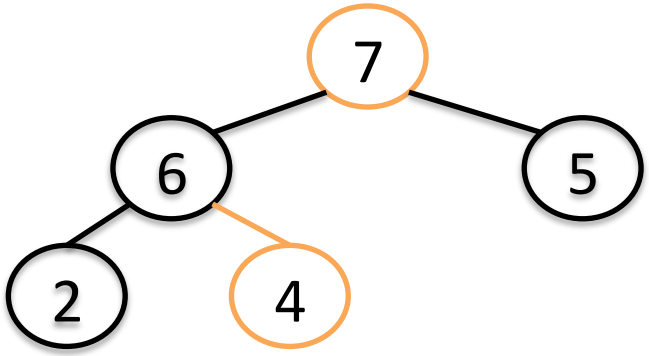
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

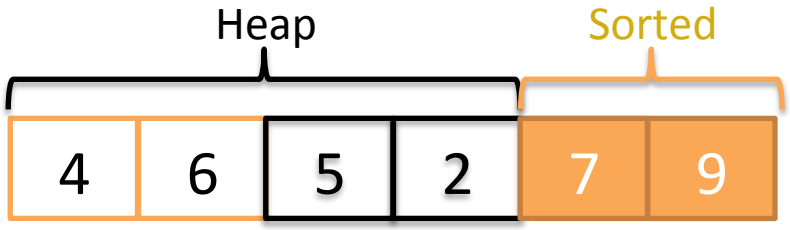


extractMax() = 7

Swap with last item in array

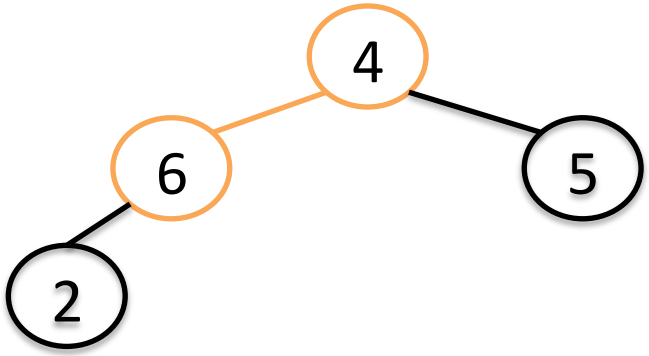
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree



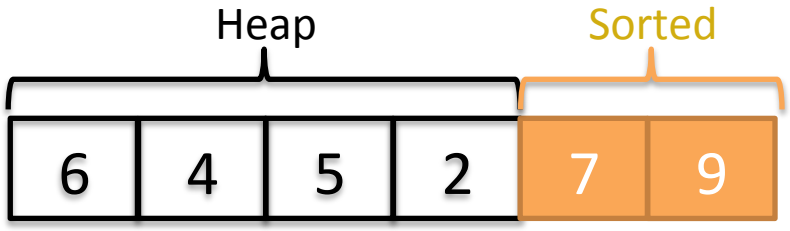
Swap 4 with largest child 6

Max heapify
Swap 4 and 6

Rebuild heap on n-2 items

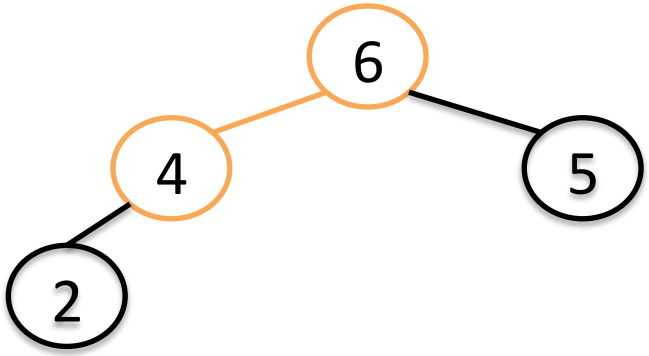
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

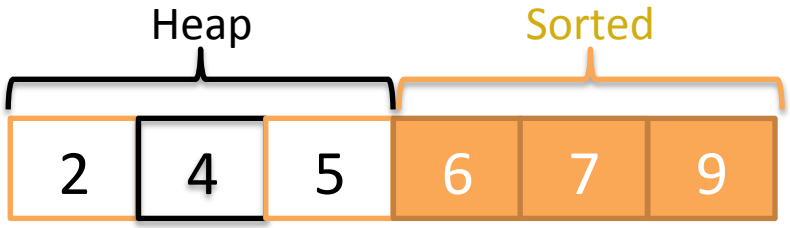


Heap built

Rebuild heap on n-2 items

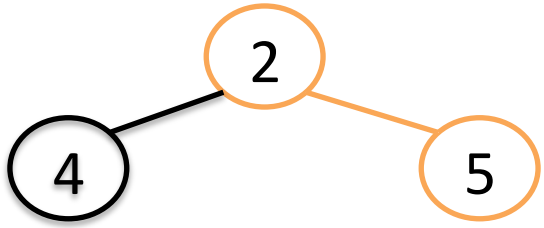
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree



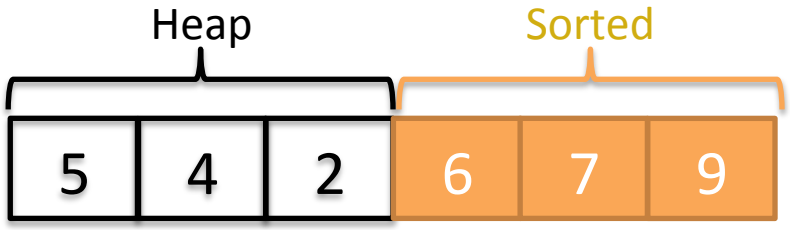
Swap 2 with largest child 5

Max heapify
Swap 5 and 2

extractMax() = 6
Swap with last item in array

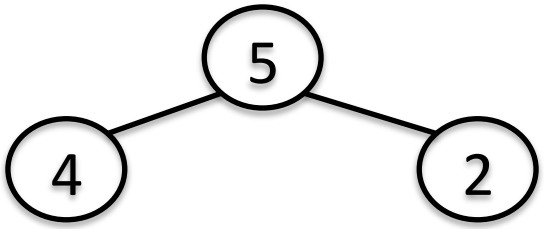
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

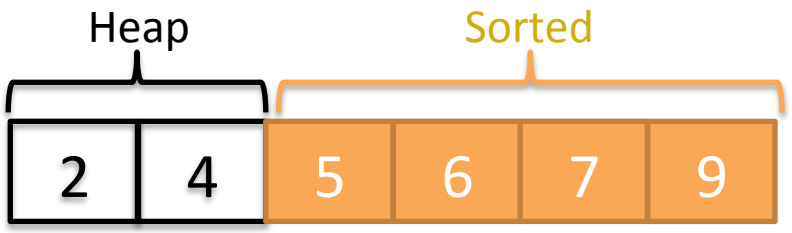


Heap built

Rebuild heap on n-3 items

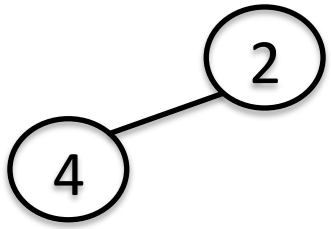
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

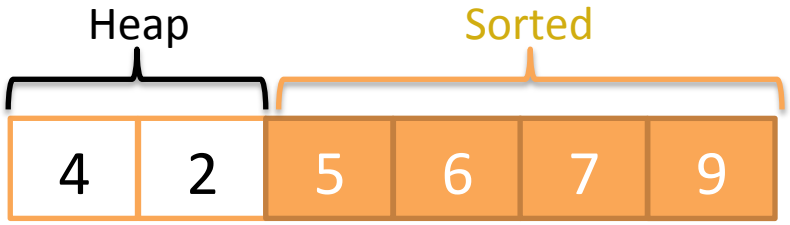


extractMax() = 5

Swap with last item in array

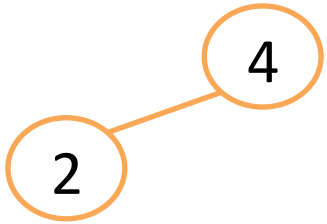
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

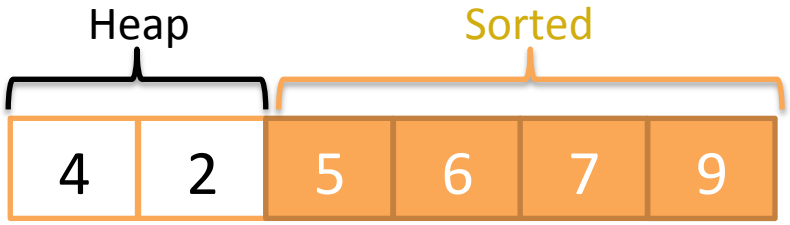


Max heapify
Swap 4 and 2

Rebuild heap on n-4 items

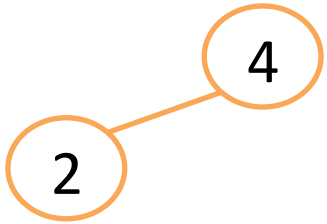
Step 2: Repeatedly *extractMax()* and store at end, rebuild heap on n-1 items

Heap on left, sorted on right



Heap array

Conceptual heap tree

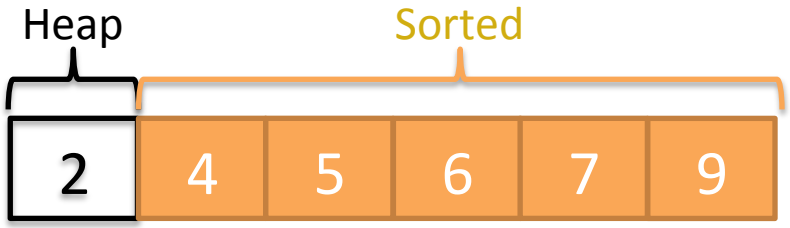


Heap built

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

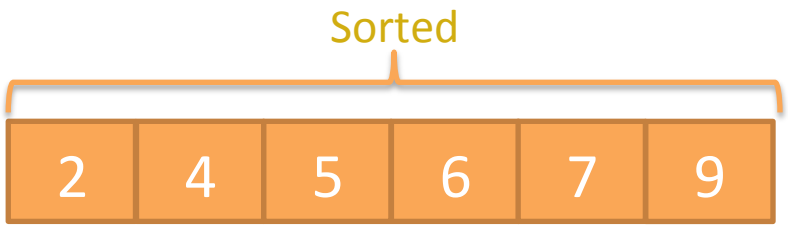
Conceptual heap tree

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     }
```

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 chi
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 parent and two children

If largest not parent, swap parent 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 $O(n \log_2 n)$

Each swap might take $\log_2 n$ operations to restore Heap

Might have to make n swaps

Code in heapsort.java