CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson

260 (255) Sudikoff

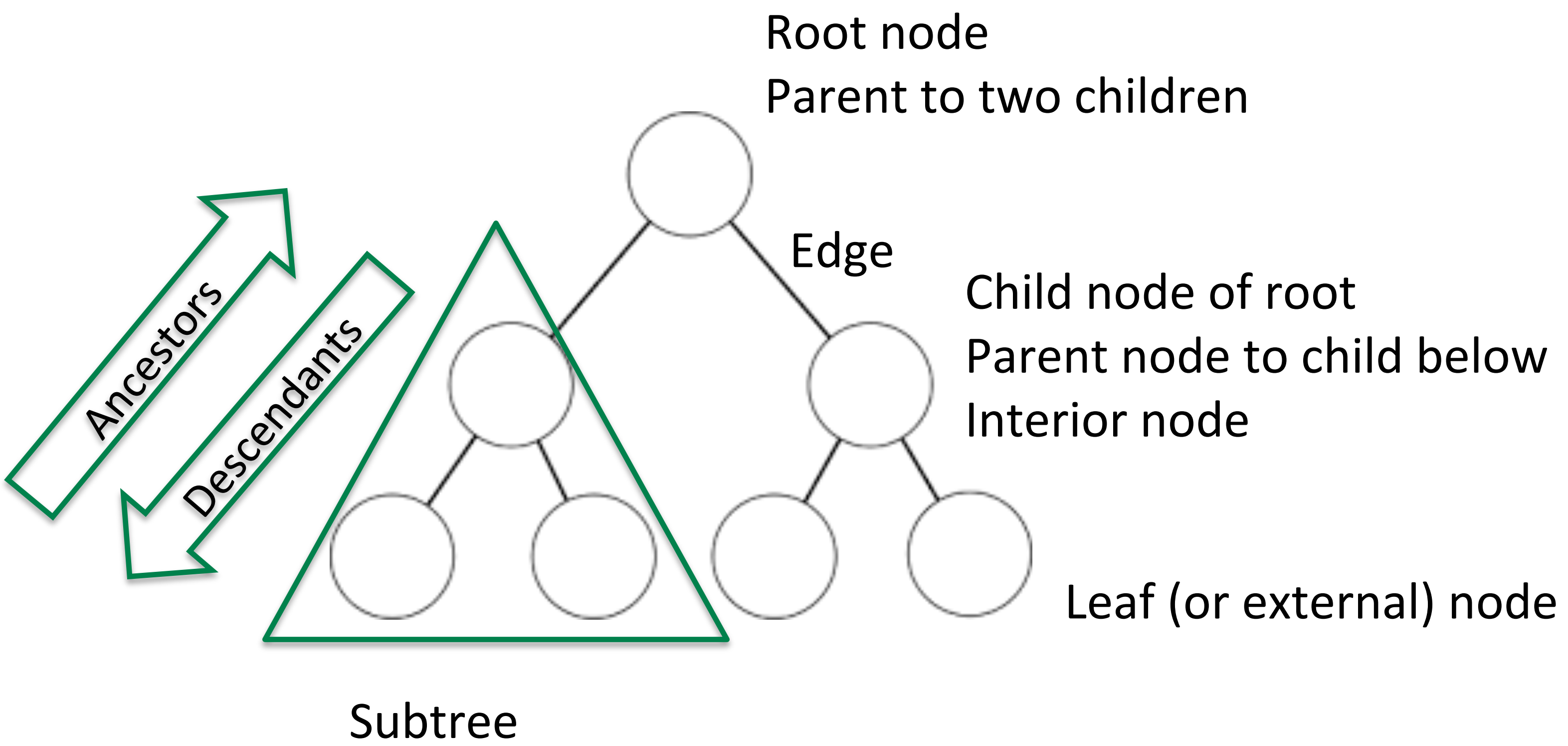
Day 14 – Prioritizing 2

Agenda

1. Heaps
2. Heap sort

Heaps based on Binary Trees

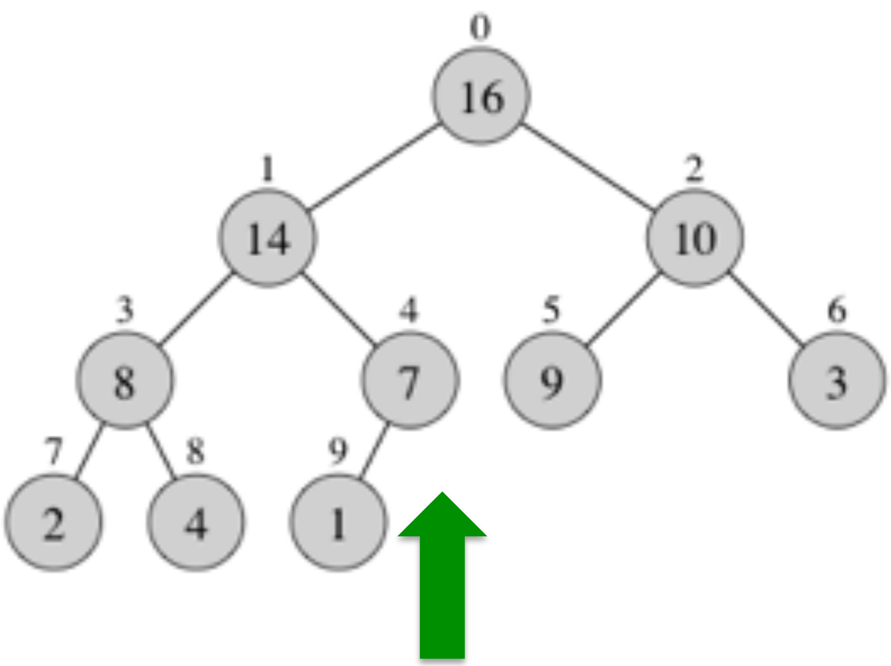
Tree data structure



In a Binary Tree, each node has 0, 1, or 2 children
Height is longest path from root to leaf
Each node has a key and a value

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

Shape property keeps tree compact



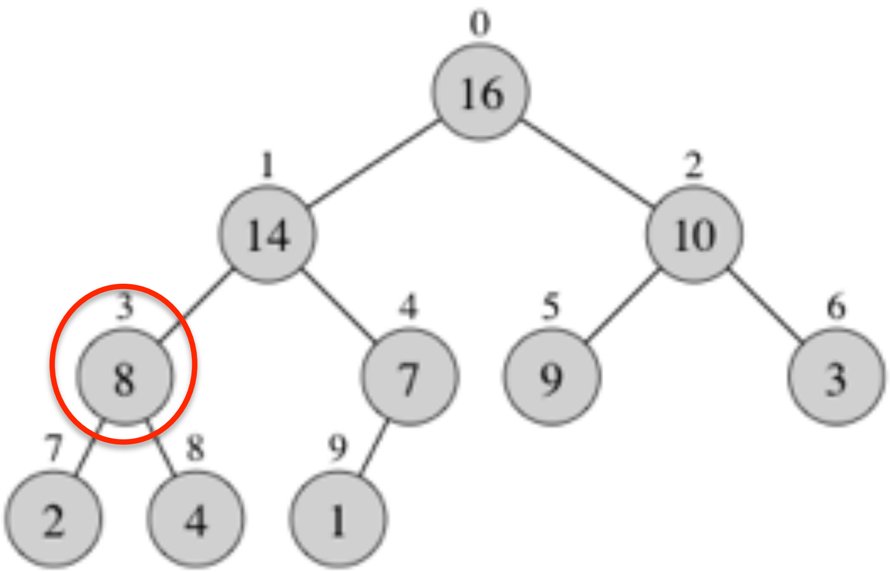
Next node
added here

Adding nodes

- Nodes added from left to right
- New level started only once a prior level is filled
- "Complete" tree
- Makes height as small as possible – $\log_2 n$
- Prevents "vines"

The shape property makes an array a natural implementation choice

Array implementation



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*

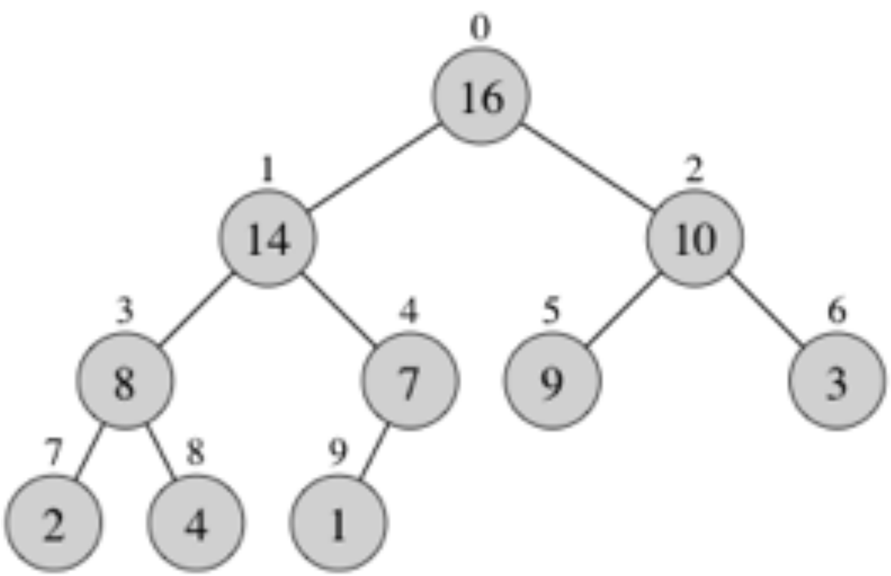
0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Node 3 containing 8

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

Heap-Order property specifies the relationship between nodes

Heap-Order property



Max heap

$\forall$ nodes $i \neq$ root,
value(parent(i)) $\geq$ value(i)

Min heap

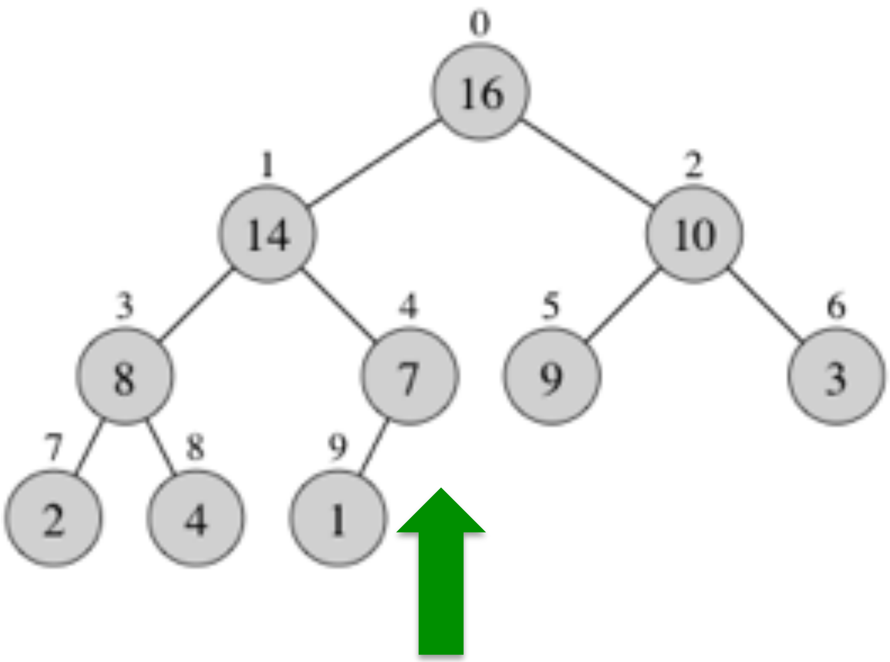
$\forall$ nodes $i \neq$ root,
value(parent(i)) $\leq$ value(i)

Root is max (or min) of entire tree

Any node is max (or min) of subtree below that node

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

Max heap insert



Next node
added here

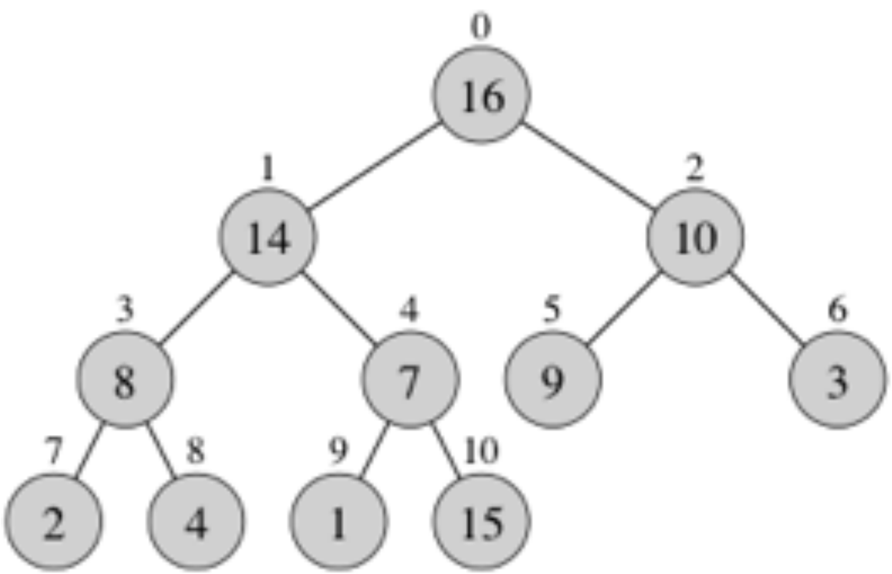
Insert 15

- 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

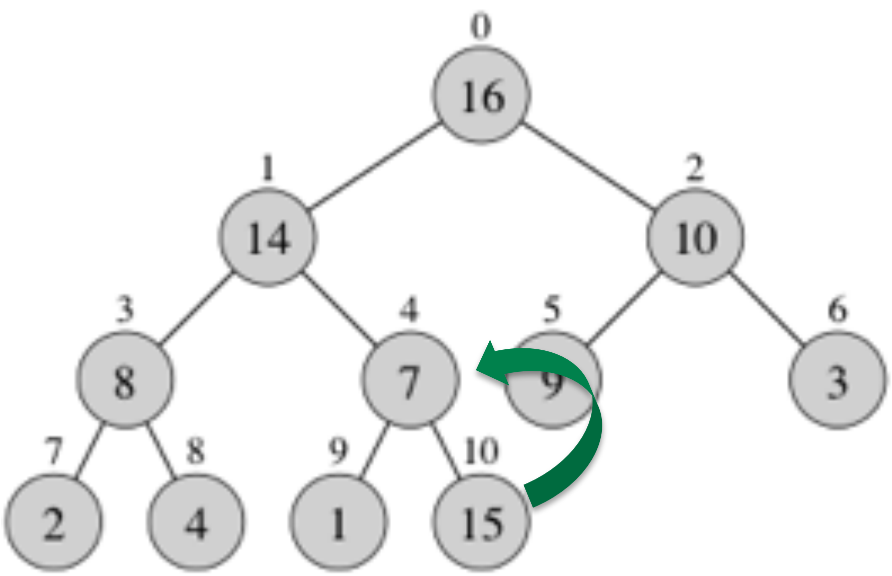
- Shape property: fill in next spot in left to right order (index i=10)

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

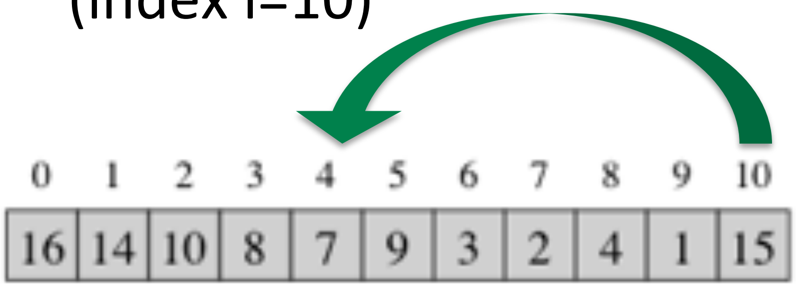
Inserting into max heap must keep both shape and order properties intact

Max heap insert



Insert 15

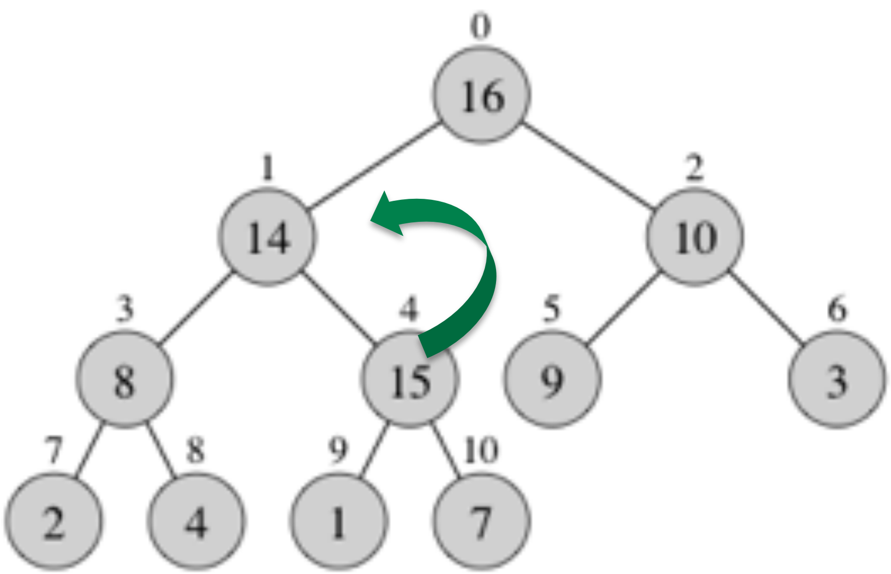
- Shape property: fill in next spot in left to right order (index i=10)



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

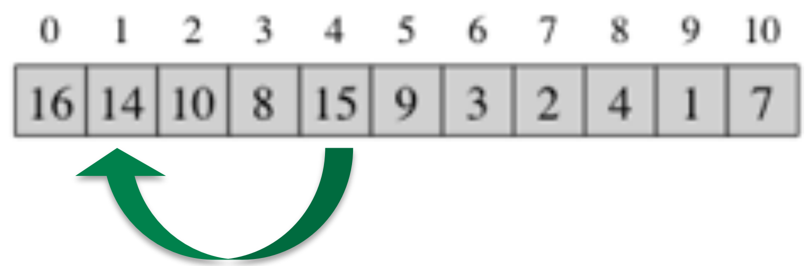
We may have to swap multiple times to get both heap properties

Max heap insert



Insert 15

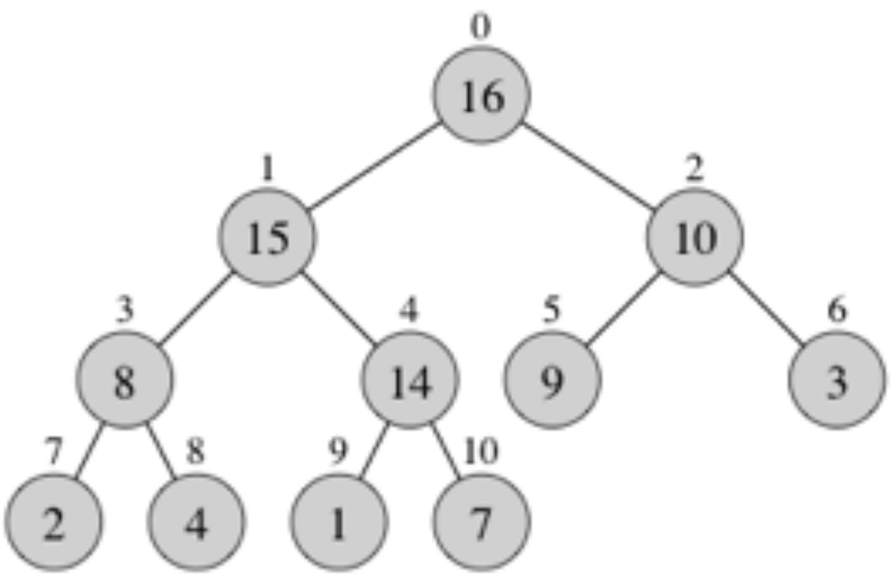
- Shape property: good!
- Order property: parent must be larger than children, not met



- 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

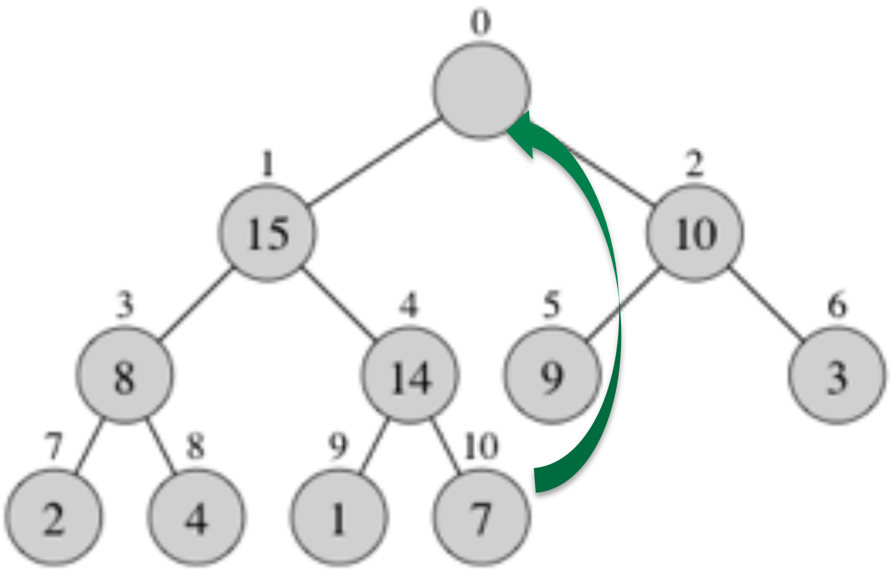
0	1	2	3	4	5	6	7	8	9	10
16	15	10	8	14	9	3	2	4	1	7

General rule

- Keep swapping until order property holds again

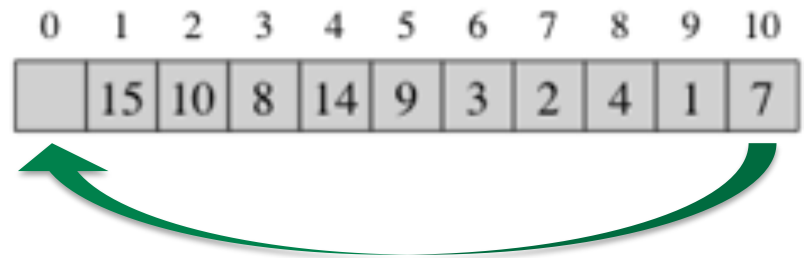
extractMax means removing the root, but that leaves a hole

extractMax



extractMax

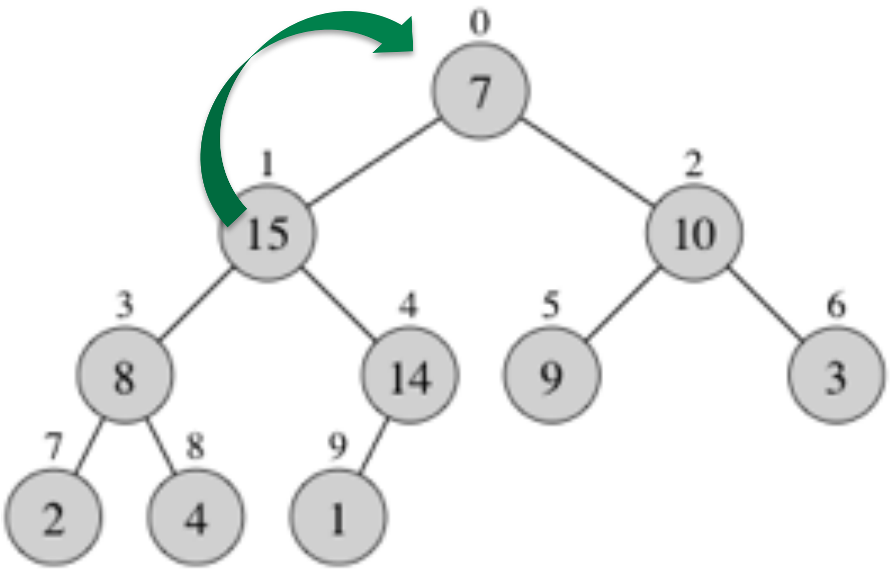
- Max position is at root (index 0)
- Removing it leaves a hole, violating shape property



- 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



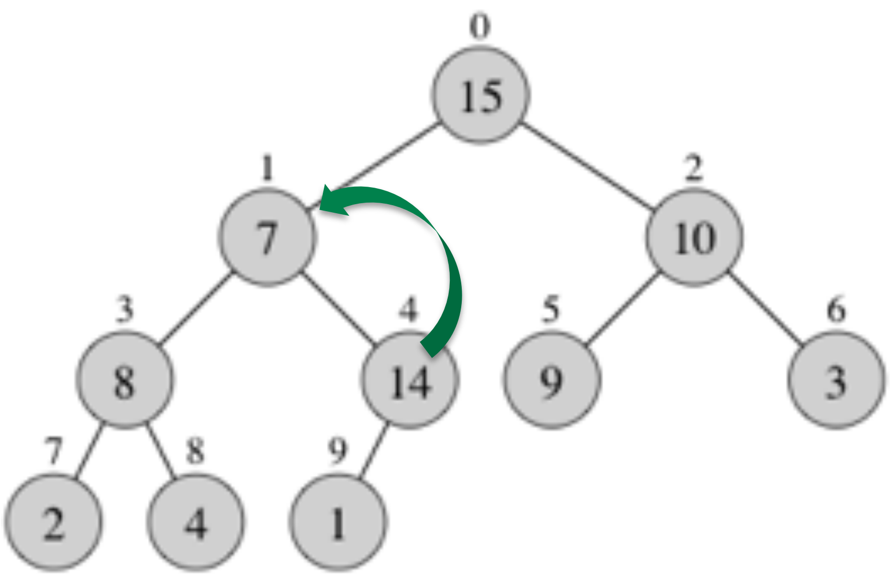
After swap

- 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

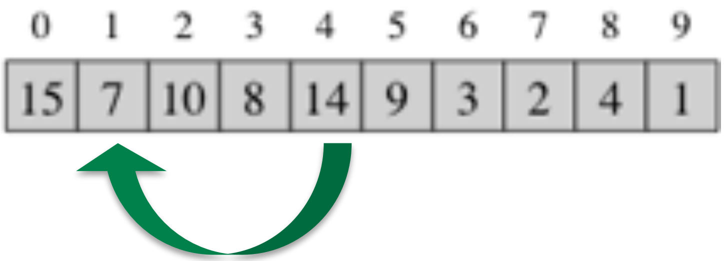
May need multiple swaps to restore order property

extractMax



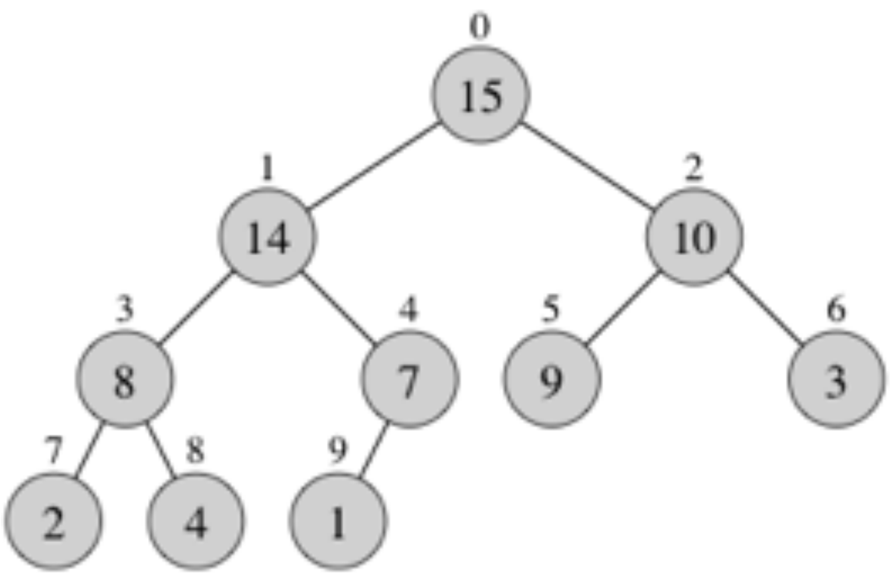
After swap 15 and 7

- 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!

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

HeapMinPriorityQueue implements a heap- based Min Priority Queue

HeapMinPriorityQueue.java

- Store items in an ArrayList called *heap*
- Helper functions *parent(), left(), right()* calculate indexes of these locations given node index
- *swap()* exchanges places of two nodes
- extractMin()
  - Remove item at index 0
  - Copy last item to index 0
  - Remove last item
  - Restore heap property by repeated swapping in *minHeapify()*
- *insert()*
  - Add item to end of heap
  - Repeated swap with parent if element smaller
- Run

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)

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: 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

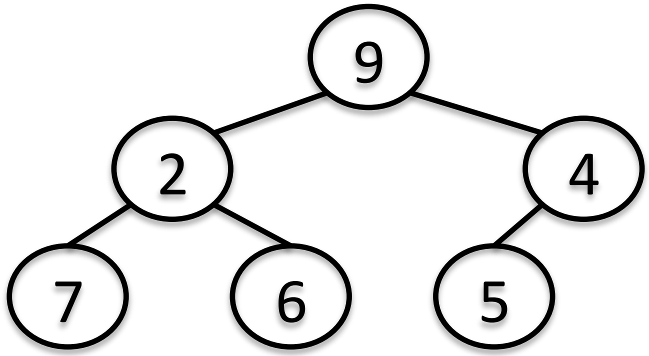
Does not require additional memory to sort

Step 1: build heap in place

Build heap given unsorted array

Given

9	2	4	7	6	5

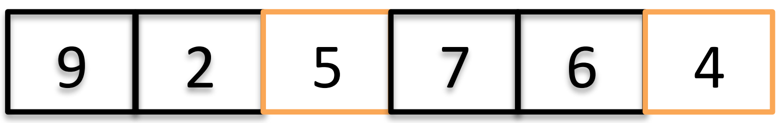


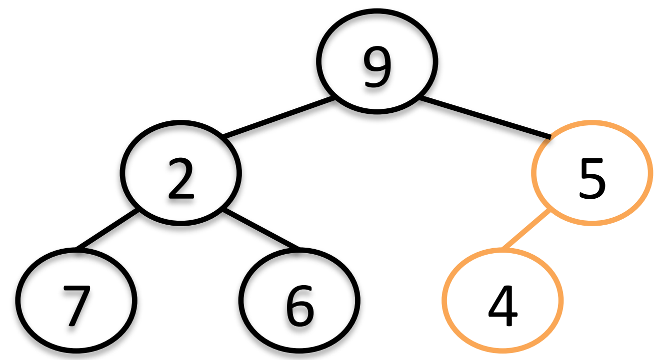
Non heap!

Given array in unsorted
First build a heap in place
Start at last leaf and heapify last leaf's parent and children (4 and 5)
Repeat for other non-leaf nodes (2 and 9)

Step 1: build heap in place

Build heap given unsorted array



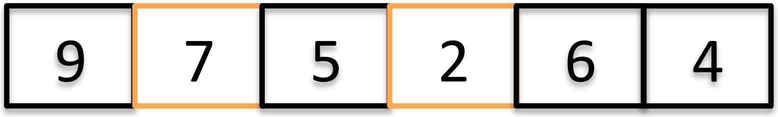


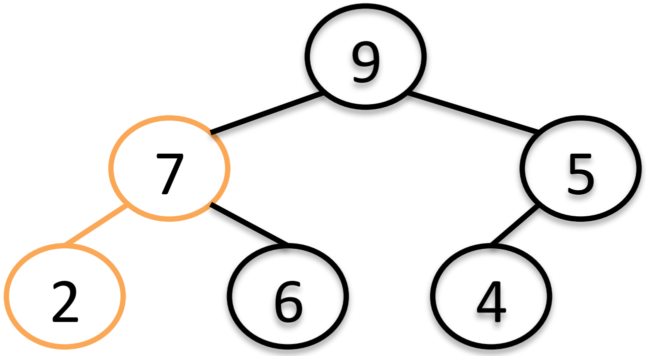
Non heap!

Given array in unsorted
First build a heap in place
Start at last leaf and heapify last leaf's parent and children (4 and 5)
Repeat for other non-leaf nodes (2 and 9)

Step 1: build heap in place

Build heap given unsorted array





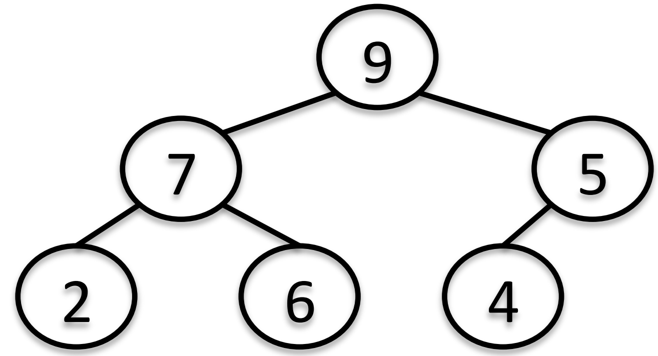
Non heap!

Given array in unsorted
First build a heap in place
Start at last leaf and heapify last leaf's parent and children (4 and 5)
Repeat for other non-leaf nodes (2 and 9)

Step 1: build heap in place

Build heap given unsorted array

9	7	5	2	6	4



Now it's a heap!

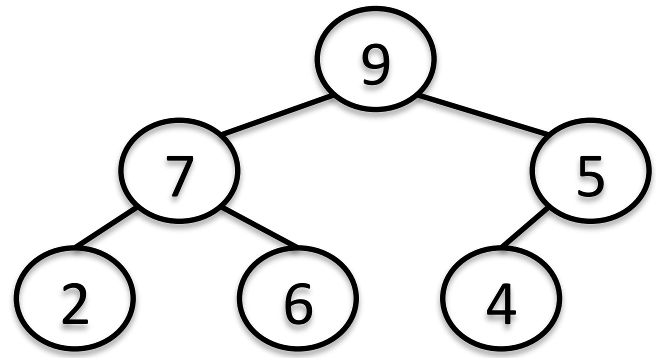
Given array in unsorted
First build a heap in place
Start at last leaf and heapify last leaf's parent and children (4 and 5)
Repeat for other non-leaf nodes (2 and 9)

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

Step 1: Build heap given unsorted 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

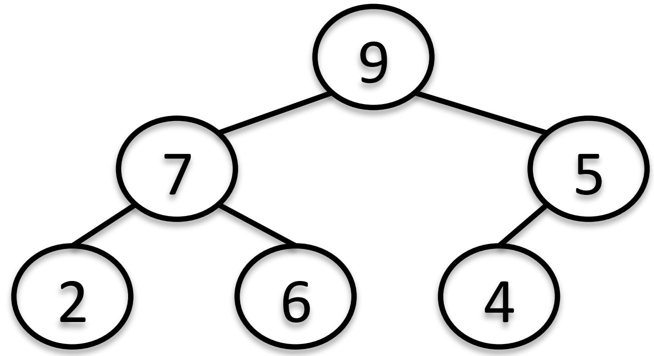
- Preorder = 9,7,2,6,5,4
- Inorder = 2,7,6,9,4,5
- Post order = 2,6,7,4,5,9

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right

9	7	5	2	6	4

Heap array

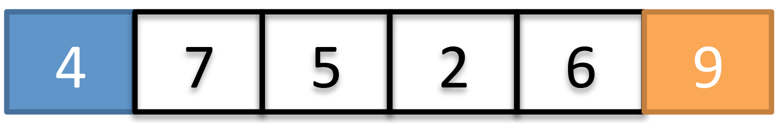


Conceptual heap tree

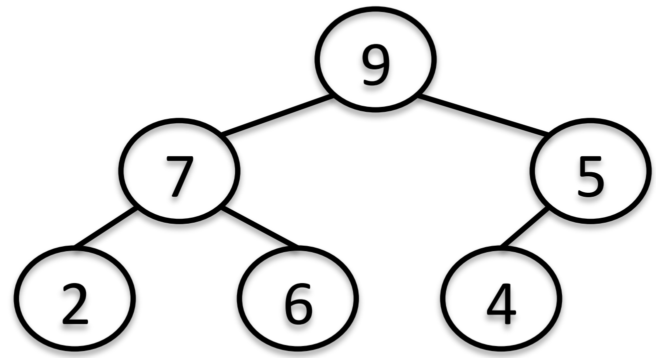
extractMin = 9
Swap with last item in array

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

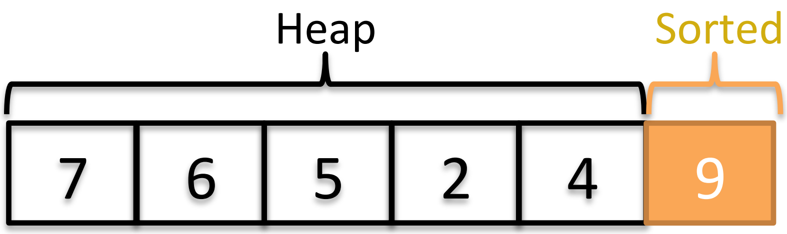


Conceptual heap tree

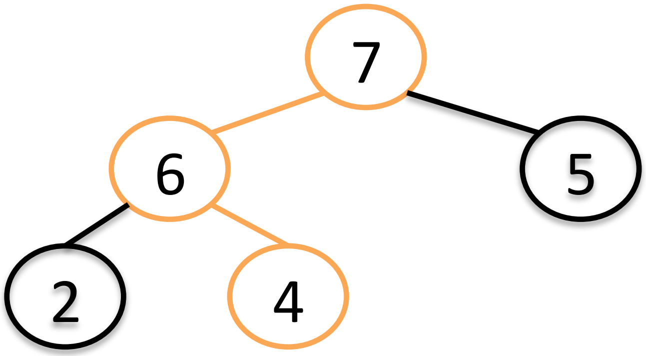
extractMin = 9
Swap with last item in array

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

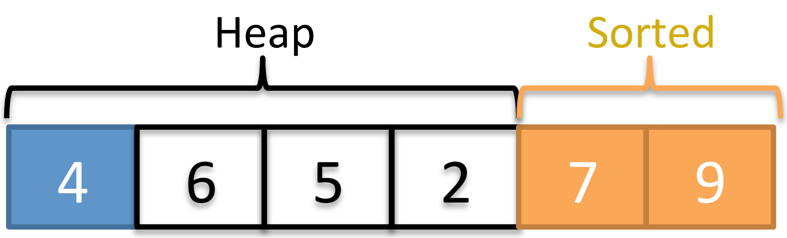


Conceptual heap tree

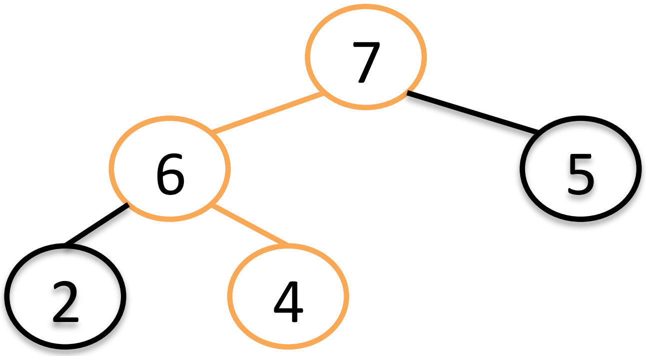
Rebuild heap on n-1 items

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

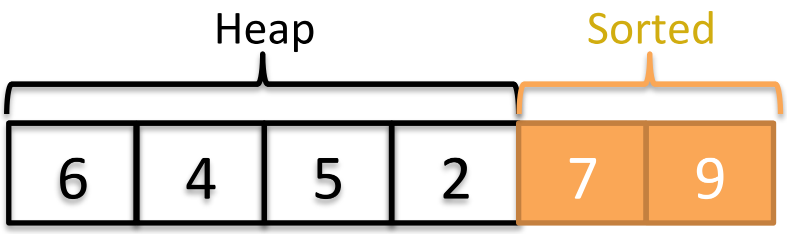


Conceptual heap tree

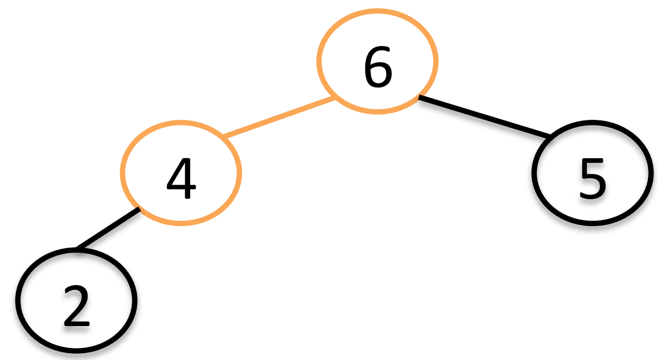
extractMax = 7
Swap with last item in array

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

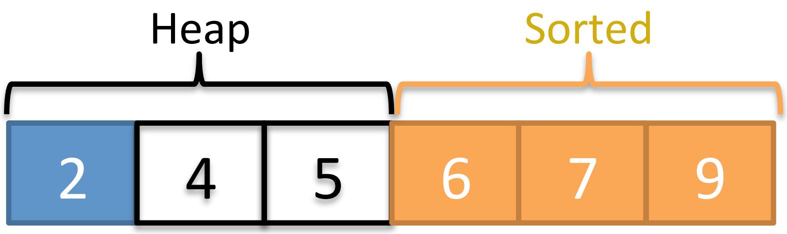


Conceptual heap tree

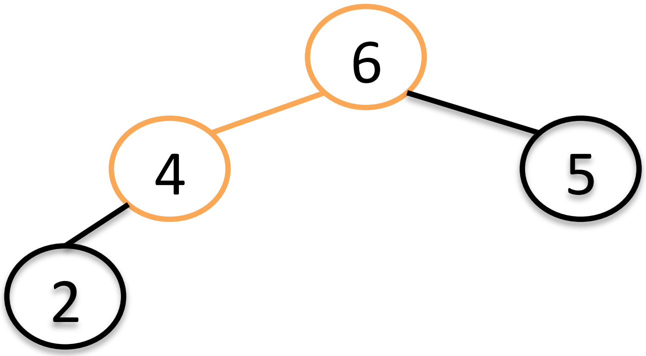
Rebuild heap on n-2 items

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

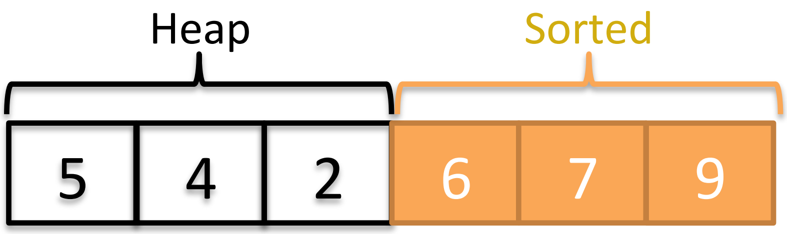


Conceptual heap tree

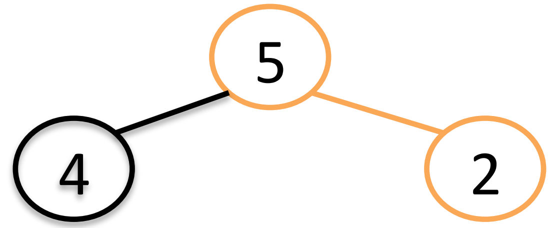
extractMax = 6
Swap with last item in array

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

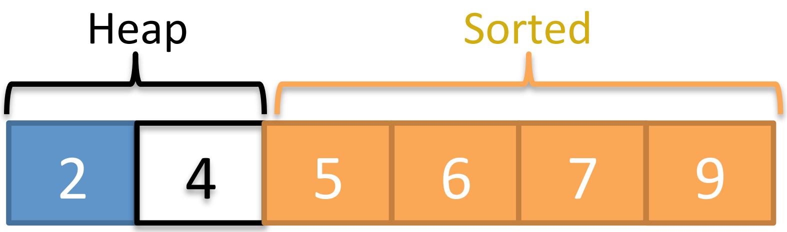


Conceptual heap tree

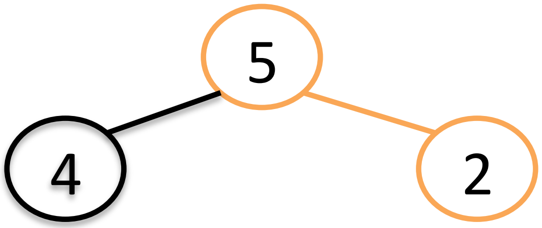
Rebuild heap on n-3 items

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

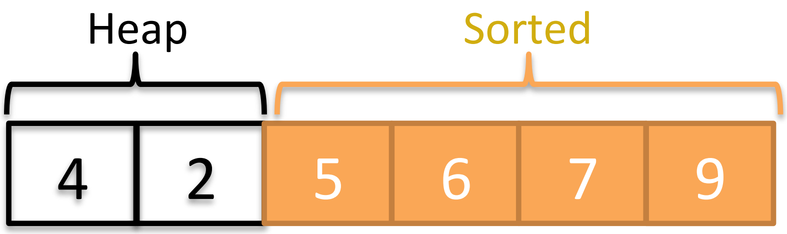


Conceptual heap tree

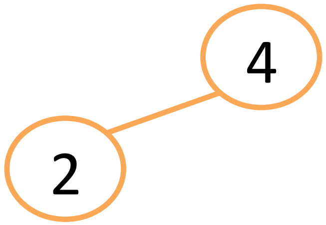
extractMax = 5
Swap with last item in array

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

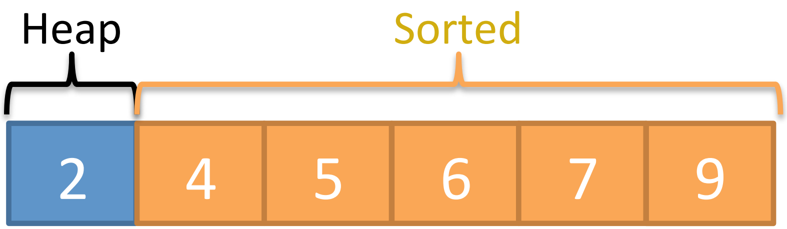


Conceptual heap tree

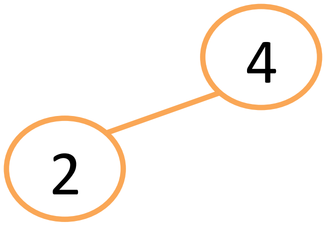
Rebuild heap on n-4 items

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

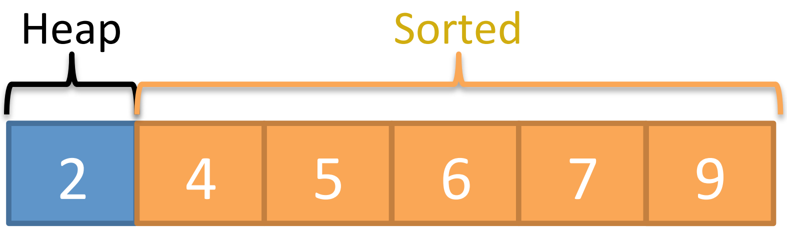


Conceptual heap tree

extractMax = 4
Swap with last item in array

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



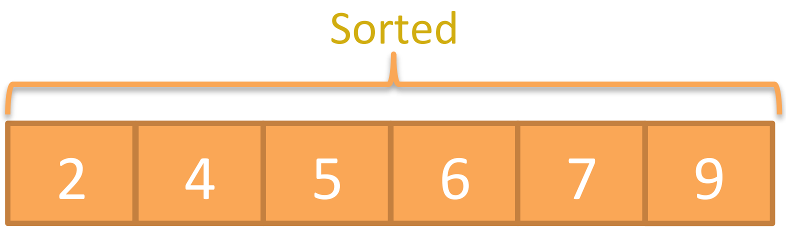
Heap array

Conceptual heap tree

Rebuild heap on n-5 items

Step 2: Repeatedly extractMax and store at end, rebuild heap

Heap on left, sorted on right



Heap array

Conceptual heap tree

Done

Items sorted in place

No extra memory used

Heapsort in two steps

Heapsort.java

Two step process:

1. First build heap
   - Set lastLeaf to last index (n-1)
   - Calculate lastNonLeaf
   - While lastNonLeaf > 0
     - Fix up heap with lastNonLeaf and it's children
     - Move to previous non leaf node
2. After heap built, repeatedly extractMax and store at end

Run time O(n log n)

Each swap might take log n operations to restore Heap

Might have to make n swaps