Binary Heap and Heap Sort

Waqar Ahmad
Department of Computer Science
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences (LUMS)

Outline

- Concept of priority and priority queues
- Define a binary min-heap
  - Examples
- Operations on heaps:
  - Top
  - Pop
  - Push
- An array representation of heaps
- Define a binary max-heap
- Using binary heaps as priority queues
- Heap sort

Priority Queue

With queues

- The order may be summarized by *first in, first out*

If each object is associated with a **priority**, we may want to pop the object which has highest priority

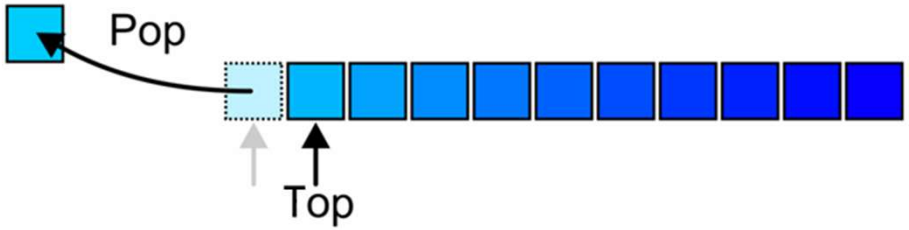
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

- The value 0 has the *highest* priority, and
- The higher the number, the lower the priority

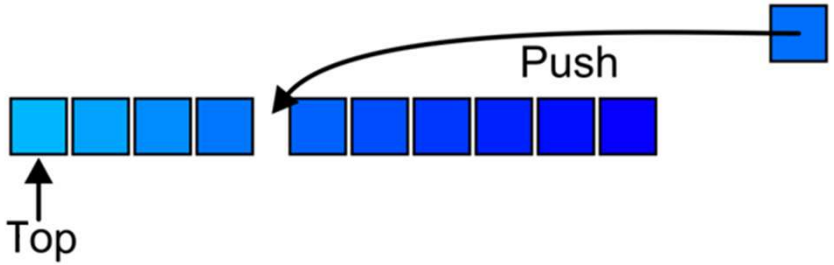
Priority Queue: Operations

The top of a priority queue is the object with highest priority

Popping from a priority queue removes the current highest priority object:



Push places a new object into the queue according to the priority



Lexicographical Priority

Priority may also depend on multiple variables:

- Two values may also specify a priority: (a, b)
- A pair (a, b) has higher priority than (c, d) if:
  - $a < c$, or
  - $a = c$ and $b < d$

For example,

- (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)

Implementations

Our goal is to make the run time of each operation as close to $\Theta(1)$ as possible

We may implement a priority queue using an AVL tree, but a more appropriate data structure is **heap**

Priority Queue using AVL Trees

We could simply insert the objects into an AVL tree where the order is given by the stated priority:

- Insertion is $\Theta(\log(n))$
- Top is $\Theta(\log(n))$
- Remove is $\Theta(\log(n))$

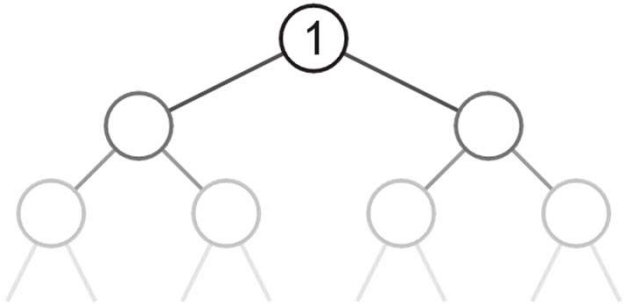
There is significant overhead for maintaining both the tree and the corresponding balance

Heaps

Can we do better in terms of performance?

The next topic defines a *heap*

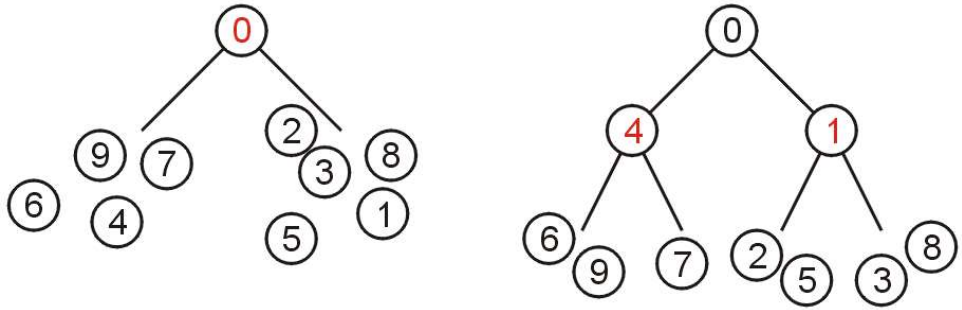
- A tree with the top object at the root
- We will cover binary heaps.
- Numerous other heaps exists.



min-Heap: Definition

A non-empty binary tree is a **min-heap** if

- The key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps
- The tree is a complete binary tree



From this definition:

- A single node is a min-heap
- All keys in either sub-tree are greater than the root key
- There is no other relationship between the elements in the two subtrees

Complete Trees

By using complete binary trees, we will be able to maintain, with minimal effort, the complete tree structure for a heap

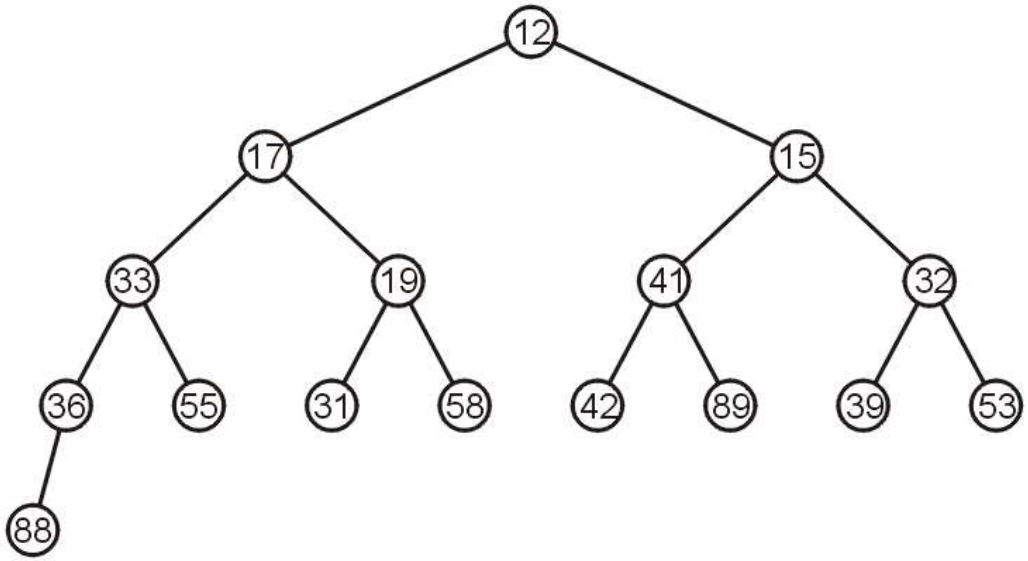
We have already seen

- It is easy to store a complete tree as an array

If we can store a heap of size n as an array of size $\Theta(n)$, this would be great!

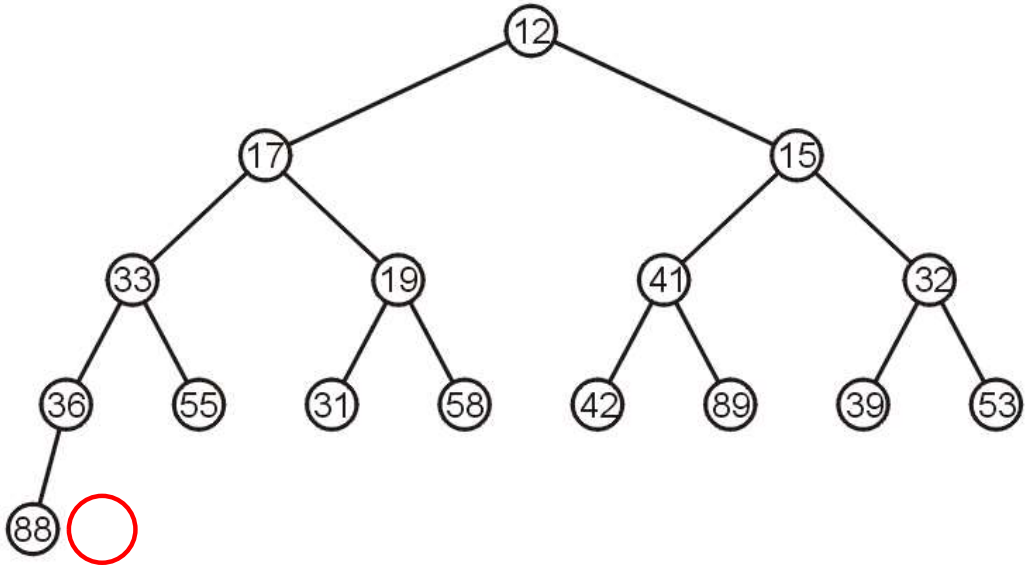
Heap as a Complete Tree

For example, here is a min-heap represented as a complete binary tree.



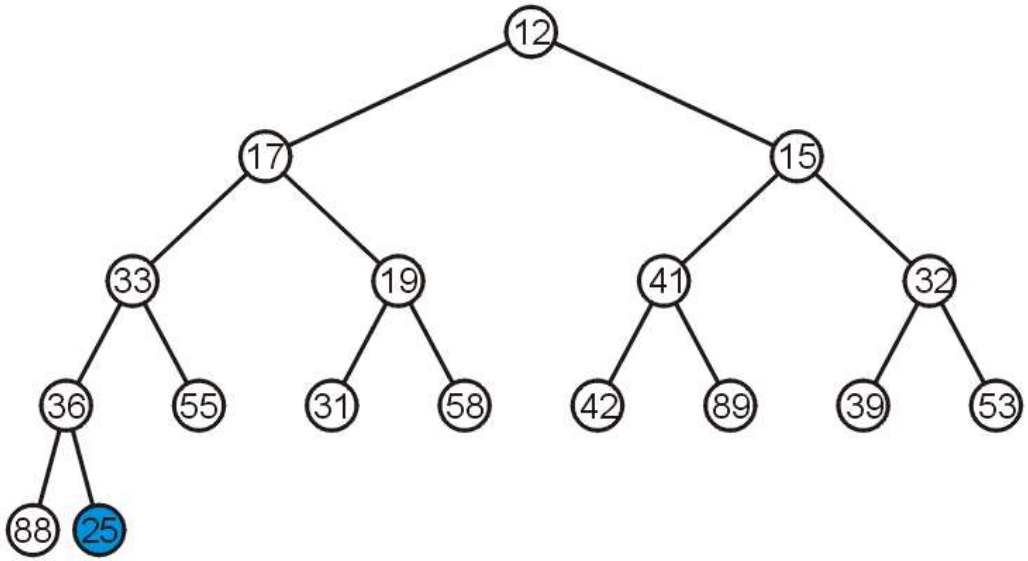
Heap: Push

If we insert into a heap represented as a complete tree, we need to place the new node as a leaf node in the appropriate location and percolate up



Heap: Push

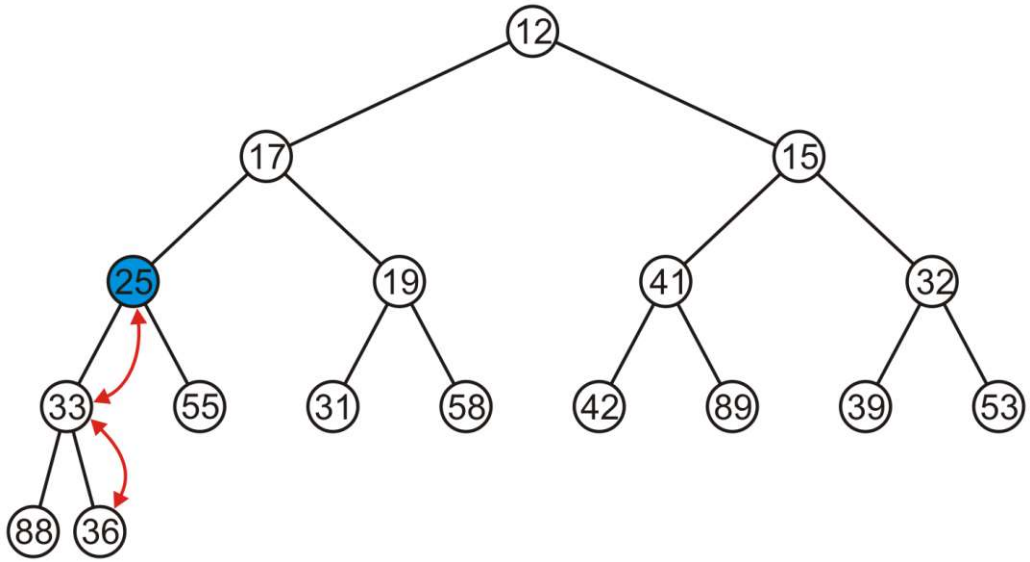
For example, push 25:



Heap: Push

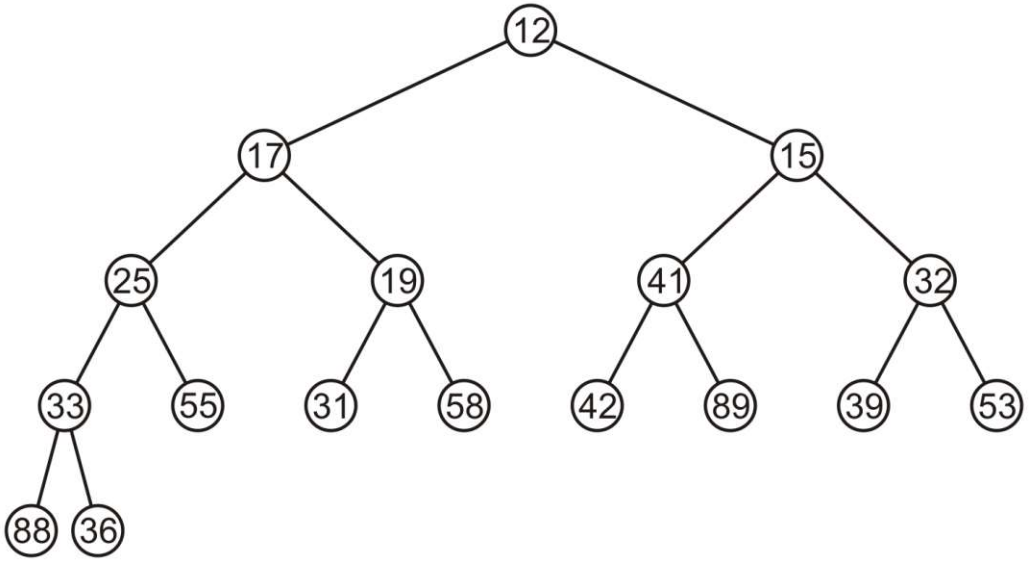
We have to percolate 25 up into its appropriate location

– The resulting heap is still a complete tree



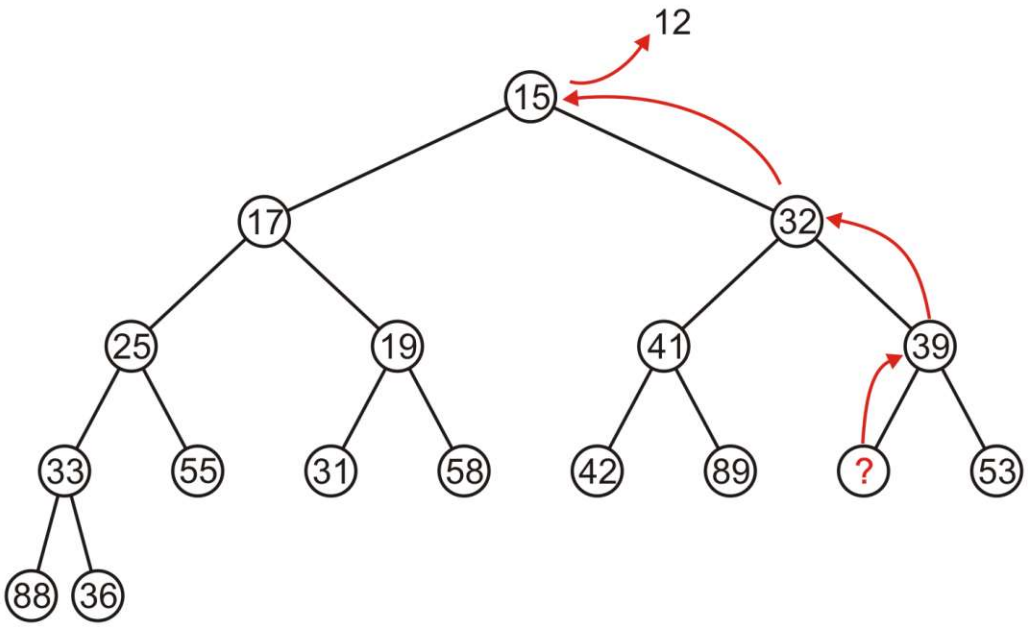
Heap: Pop

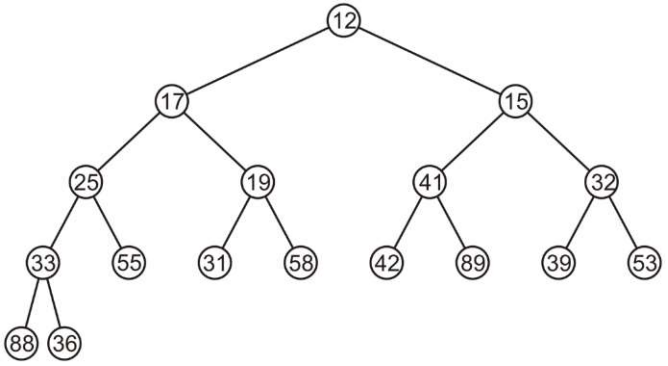
Suppose we want to pop the top entry: 12



Heap: Pop

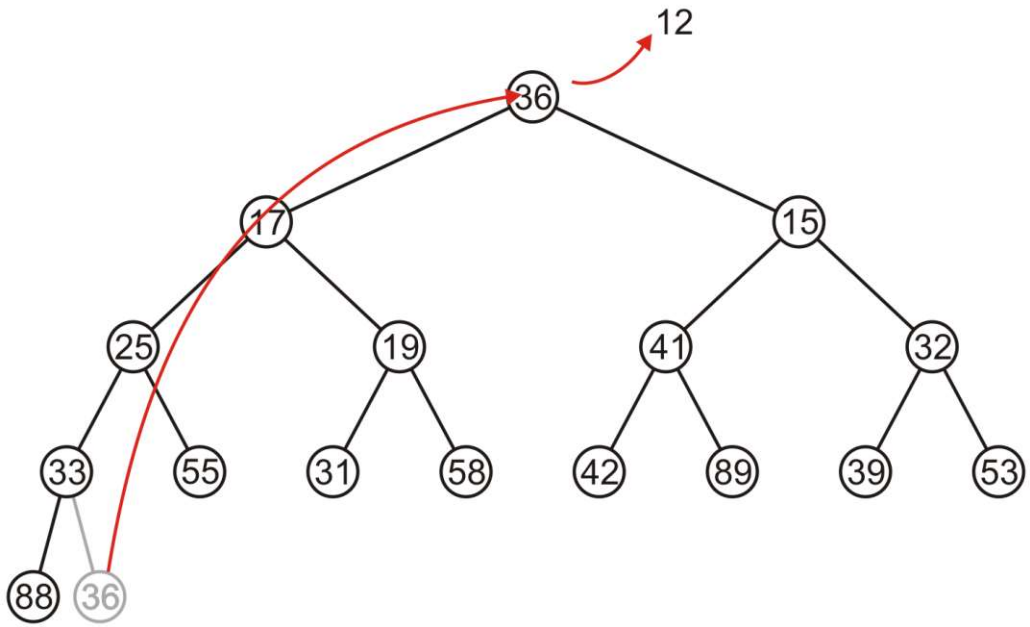
Percolating up creates a hole leading to a **non-complete tree**





Heap: Pop

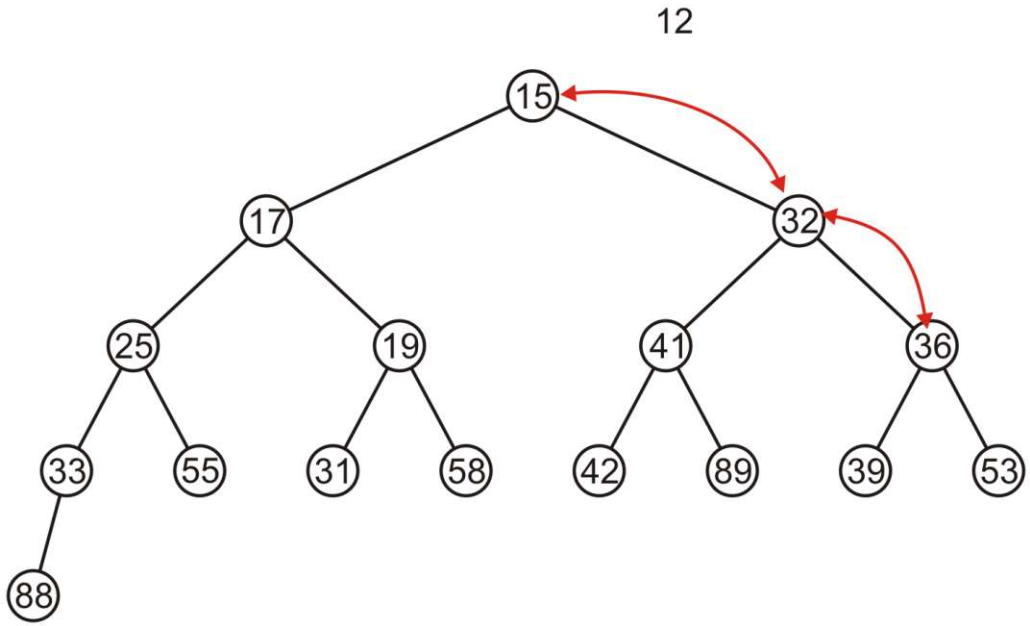
Alternatively, copy the last entry of the heap to the root



Heap: Pop

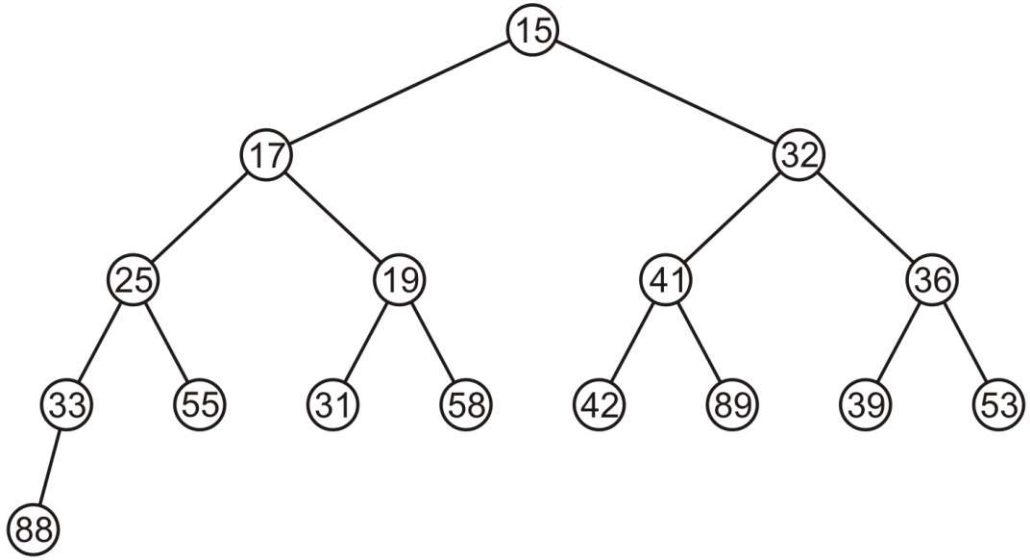
Now, percolate 36 down swapping it with the smallest of its children

– We stop when both children are larger



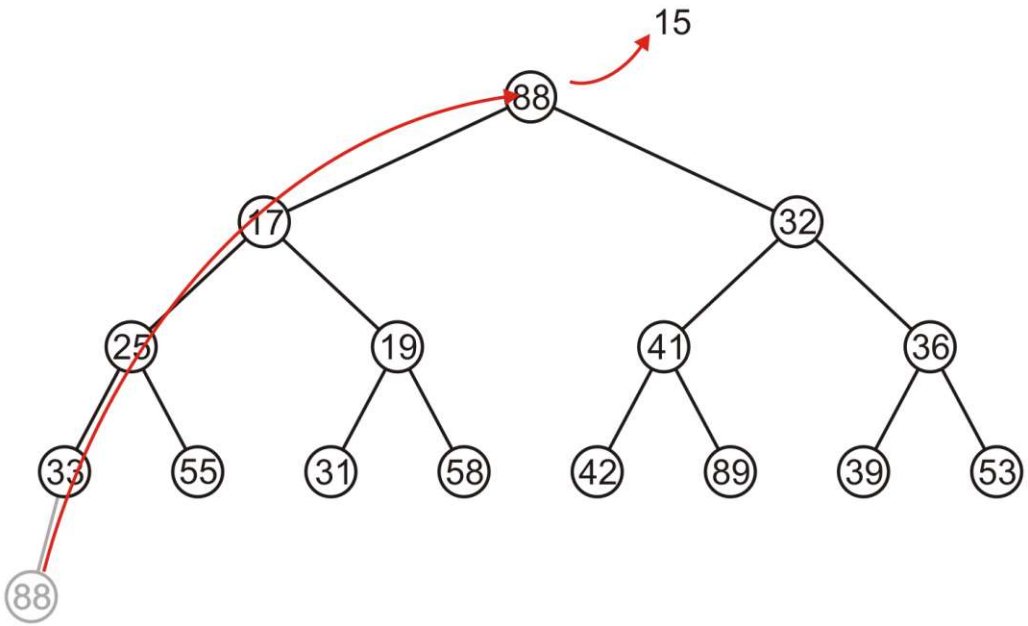
Heap: Pop

The resulting tree is now still a complete tree:



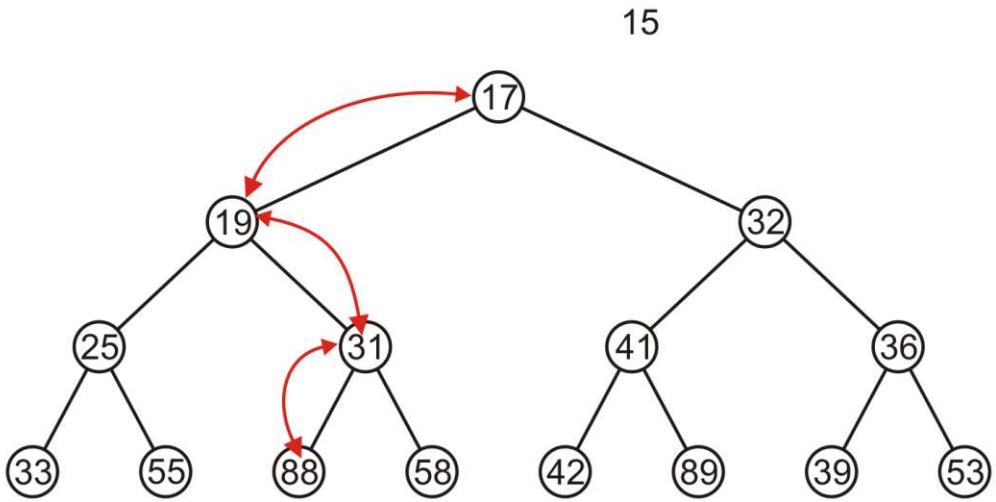
Heap: Pop

Again, popping 15, copy up the last entry: 88



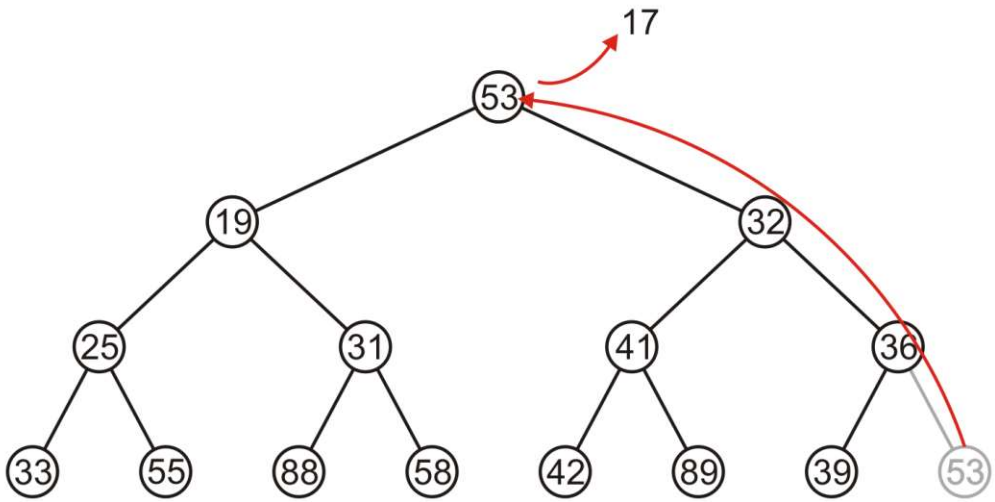
Heap: Pop

This time, it gets percolated down to the point where it has no children



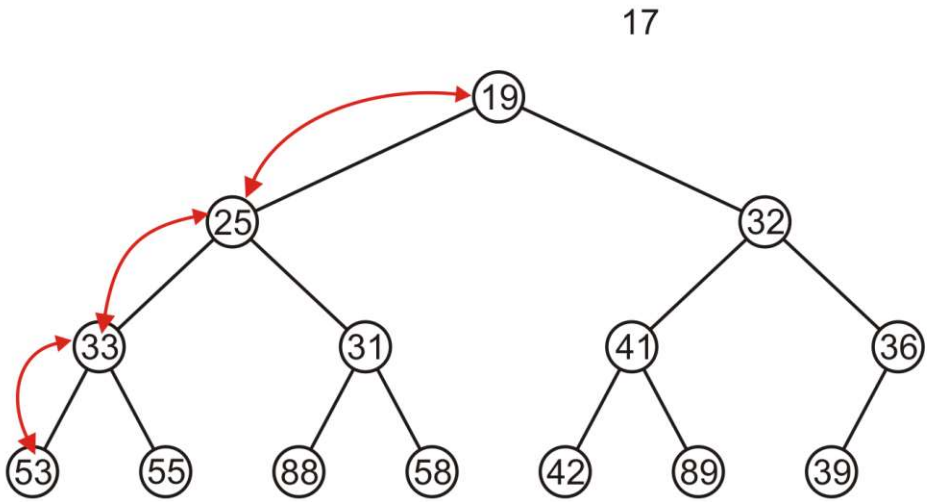
Heap: Pop

In popping 17, 53 is moved to the top



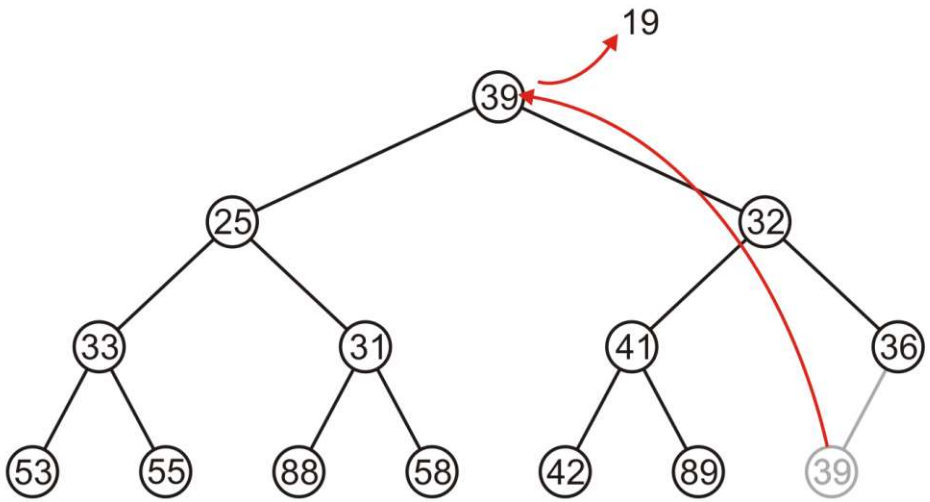
Heap: Pop

And percolated down, again to the deepest level



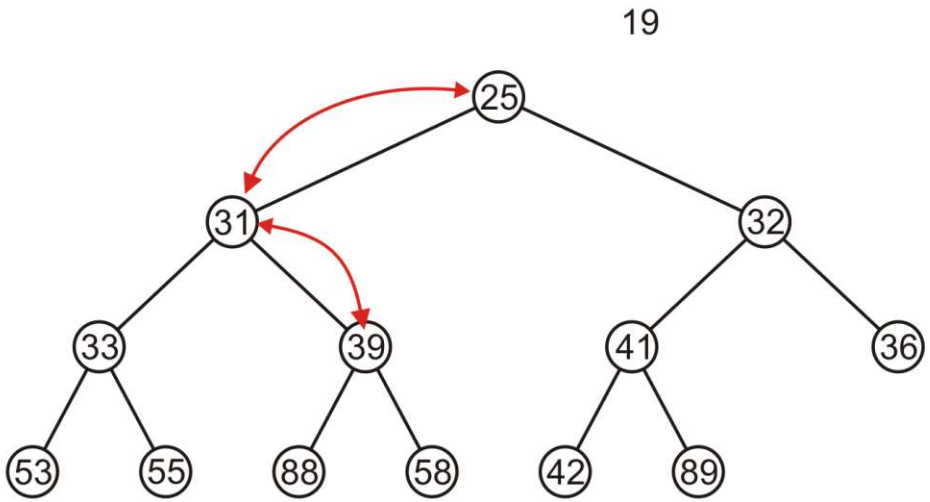
Heap: Pop

Popping 19 results in copying 39 up



Heap: Pop

39 is then percolated down to the second deepest level



Heap as a Complete Tree

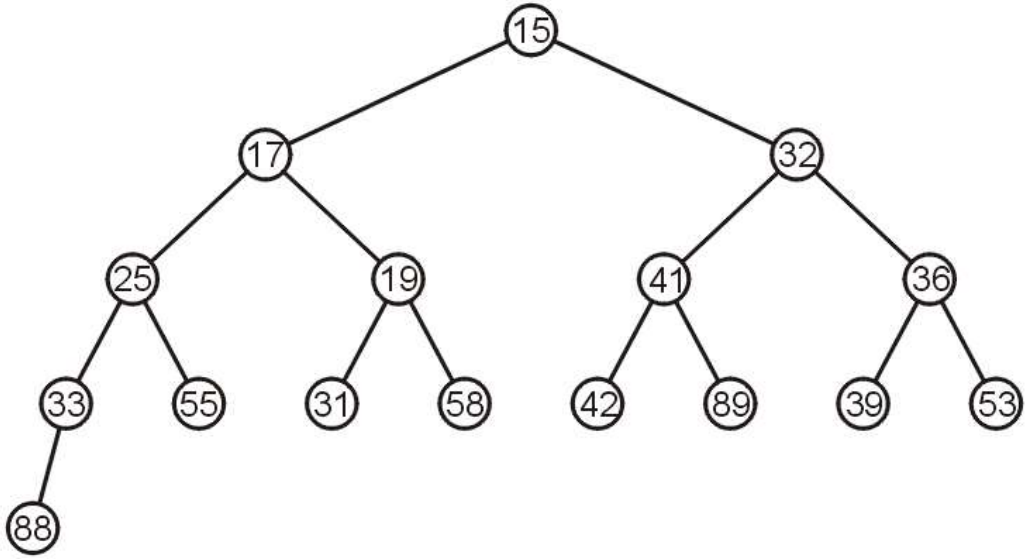
Therefore, we can maintain the complete-tree shape of a heap

We may store a complete tree using an array:

- A complete tree is filled in breadth-first traversal order
- The array is filled using breadth-first traversal

Array Implementation

For the heap

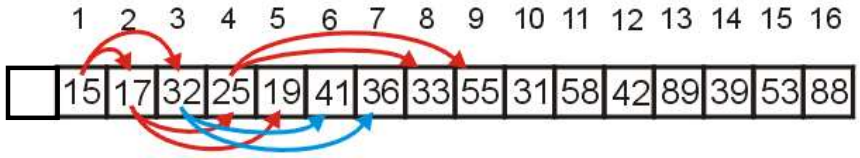


a breadth-first traversal yields:

	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

Array Implementation

If we associate an index–starting at 1–with each entry in the breadth-first traversal, we get:



Given the entry at index k, it follows that:

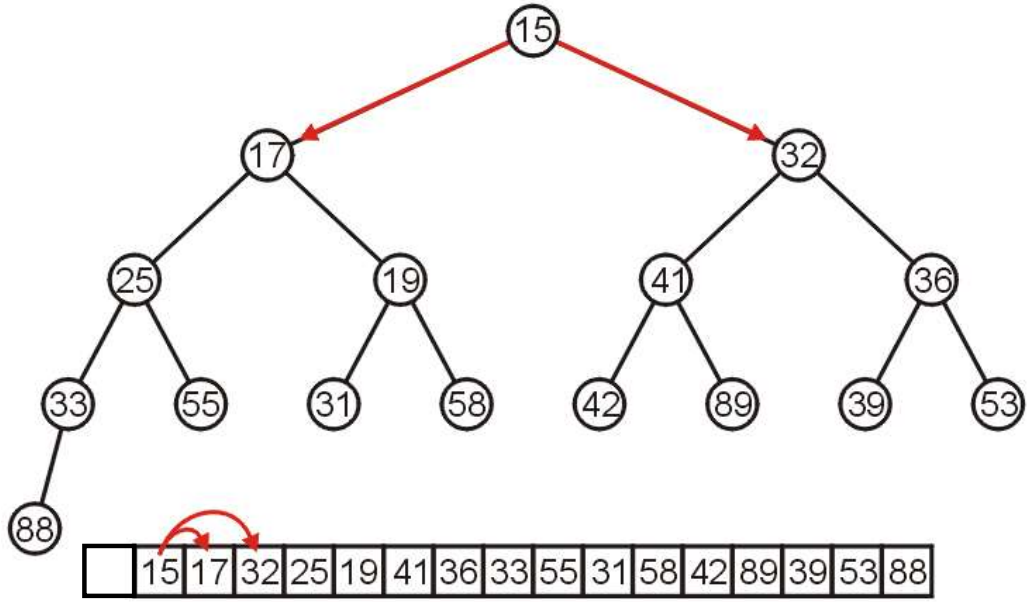
- The parent of node is at $k/2$
- the children are at $2k$ and $2k+1$

```
parent = k >> 1;
```

```
left_child = k << 1;
right_child = left_child | 1;
```

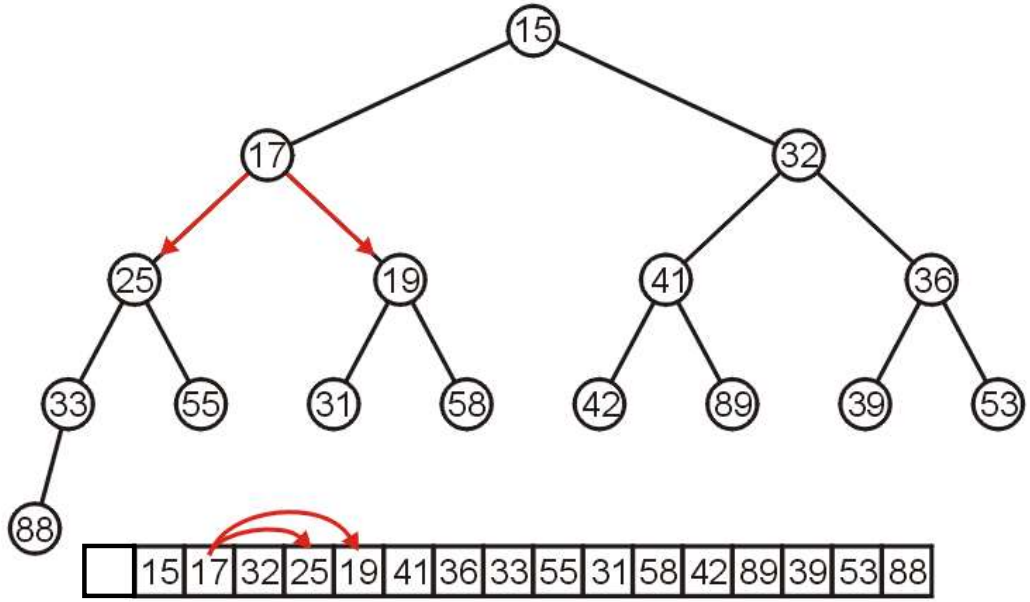
Array Implementation

The children of 15 are 17 and 32:



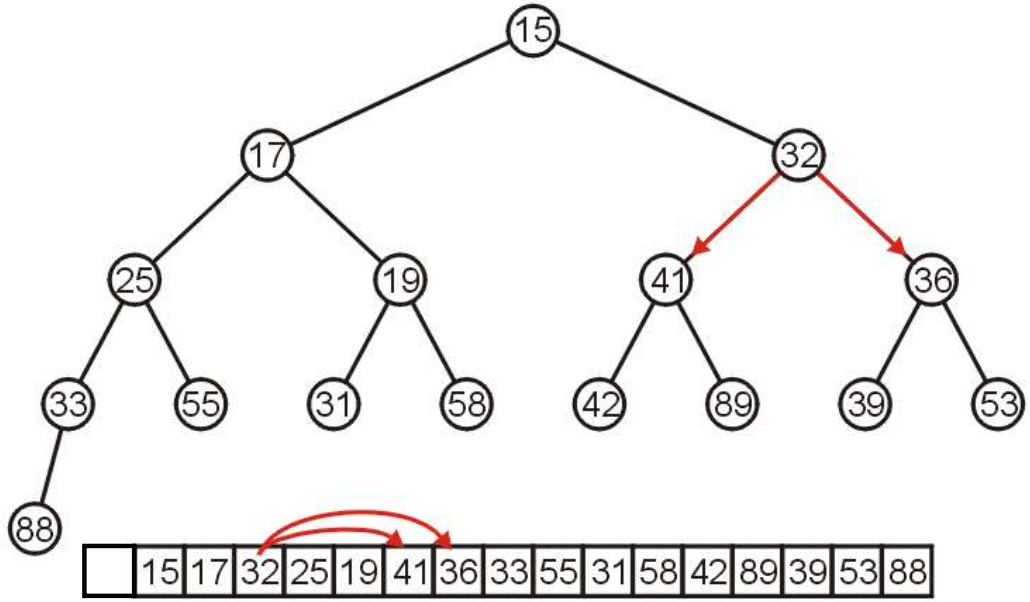
Array Implementation

The children of 17 are 25 and 19:



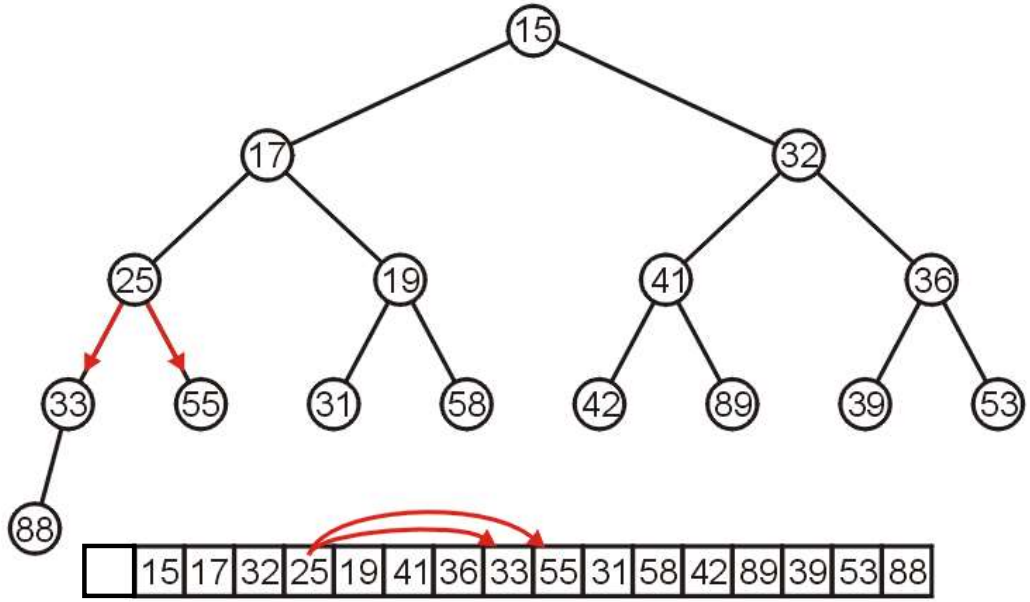
Array Implementation

The children of 32 are 41 and 36:



Array Implementation

The children of 25 are 33 and 55:



Array Implementation

If the heap-as-array has **`count`** entries, then the next empty node in the corresponding complete tree is at location

```
position = count + 1
```

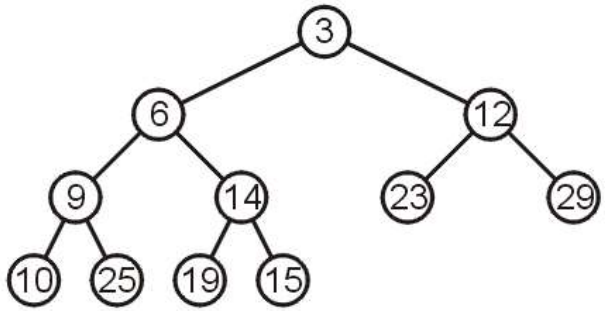
We compare the item at location **`position`** with the item at **`position/2`**

If they are out of order

- Swap them, and set **`position /= 2`** and repeat.

Array Implementation

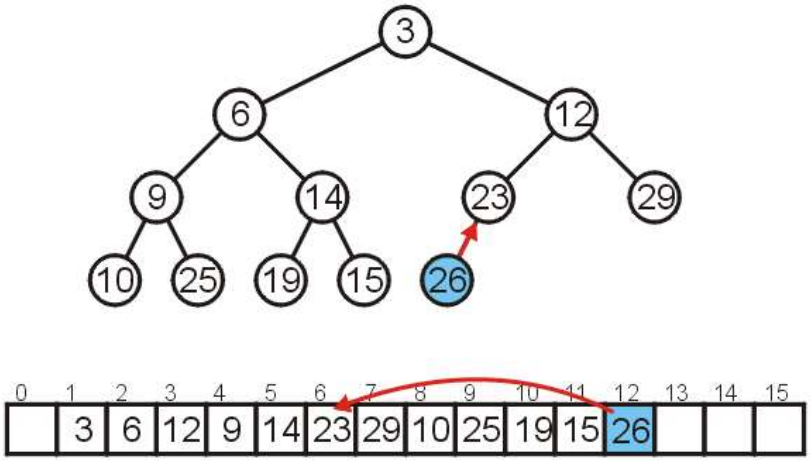
Consider the following heap, both as a tree and in its array representation



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	12	9	14	23	29	10	25	19	15				

Array Implementation: Push

Inserting 26 requires no changes



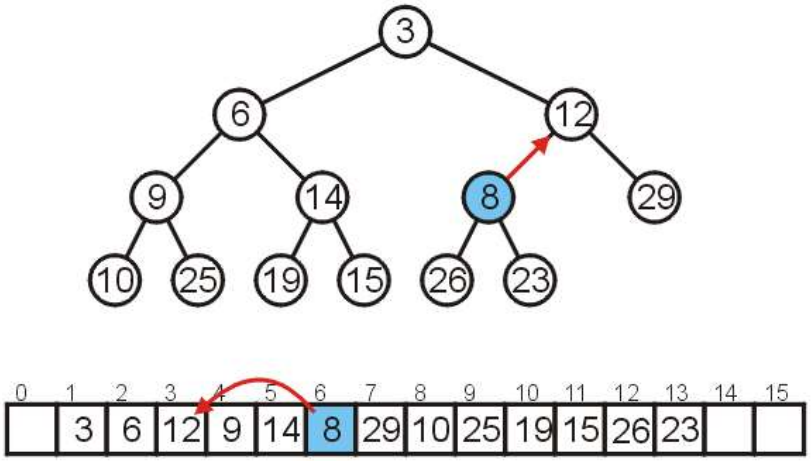
Array Implementation: Push

Inserting 8 requires a few percolations:

- Swap 8 and 23

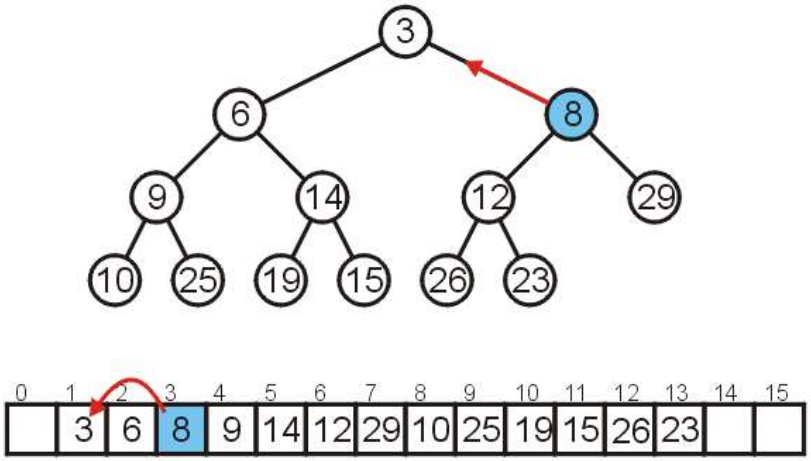
Array Implementation: Push

Swap 8 and 12



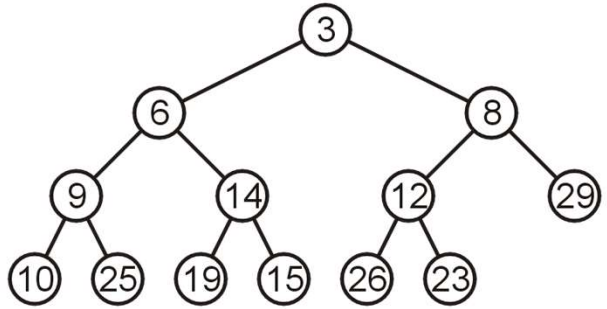
Array Implementation: Push

At this point, it is greater than its parent, so we are done



Array Implementation: Pop

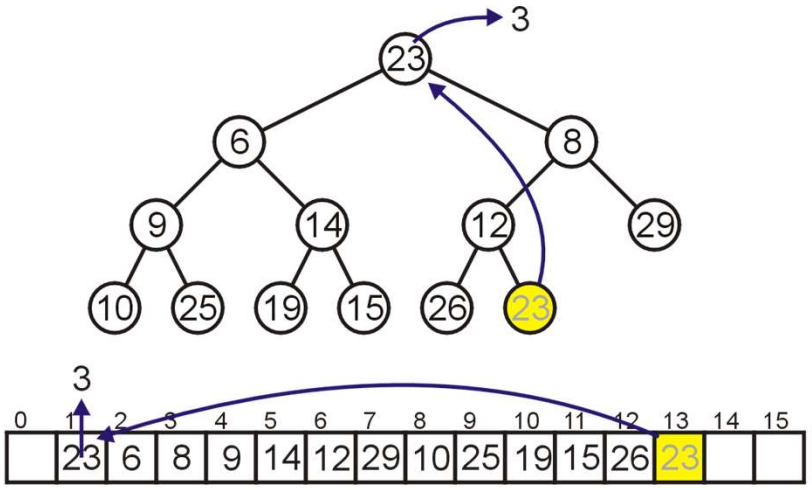
Popping the top requires copying the last entry to the top



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

Array Implementation: Pop

Copy the last object, 23, to the root

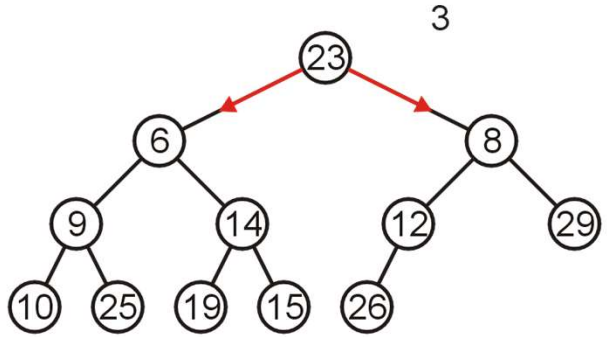


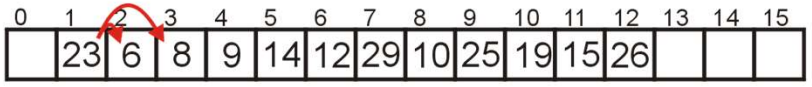
Array Implementation: Pop

Now percolate down

Compare Node 1 with its children: Nodes 2 and 3

– Swap 23 and 6

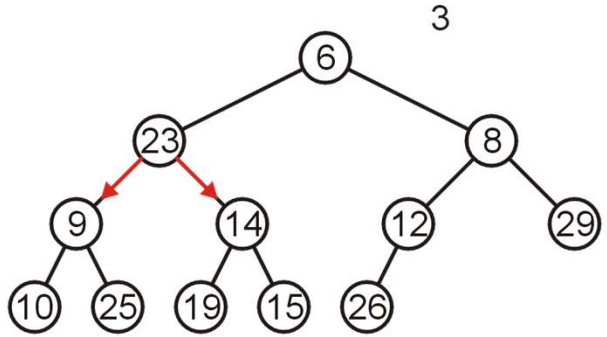




Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

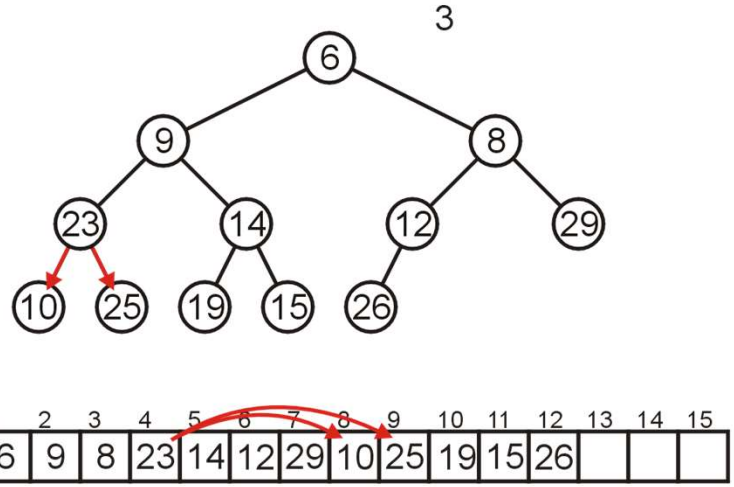
– Swap 23 and 9



Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

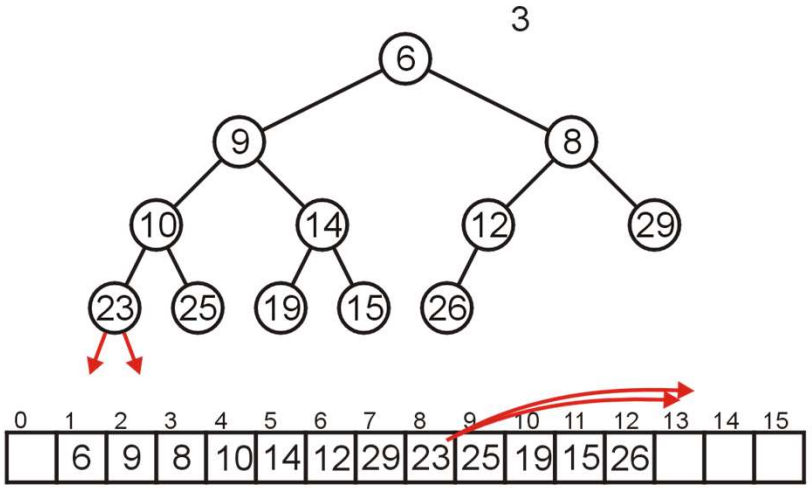
– Swap 23 and 10



Array Implementation: Pop

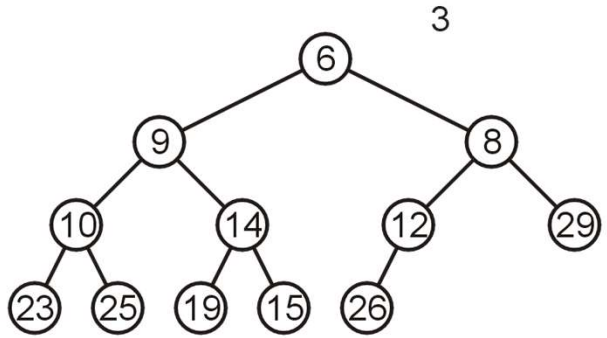
The children of Node 8 are beyond the end of the array:

– Stop



Array Implementation: Pop

The result is a binary min-heap

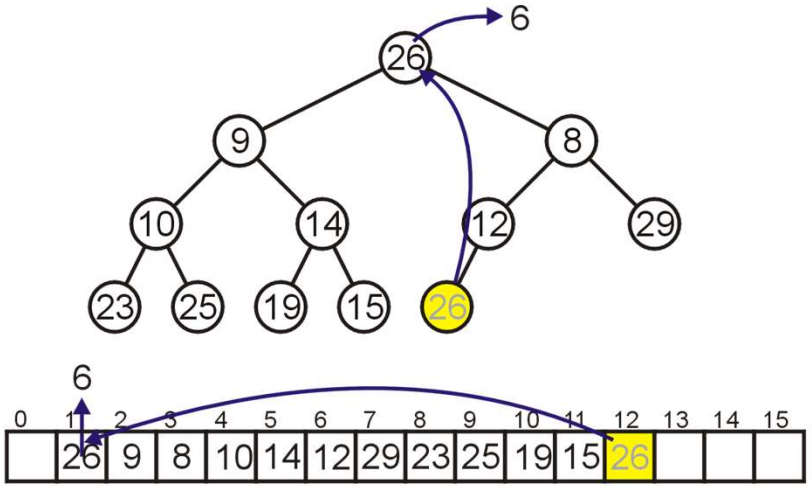


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	6	9	8	10	14	12	29	23	25	19	15	26			

Array Implementation: Pop

Dequeuing the minimum again:

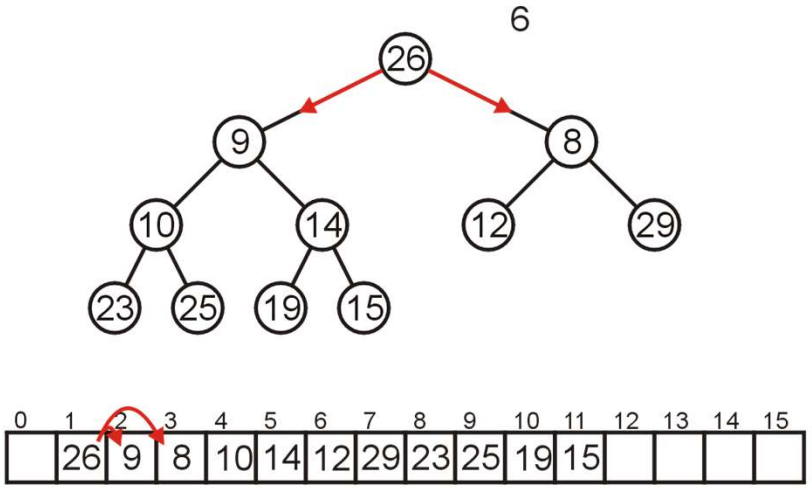
– Copy 26 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

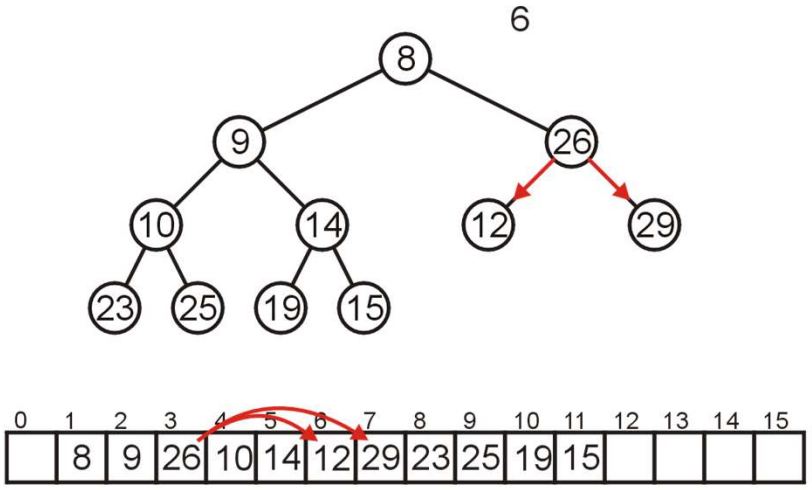
– Swap 26 and 8



Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

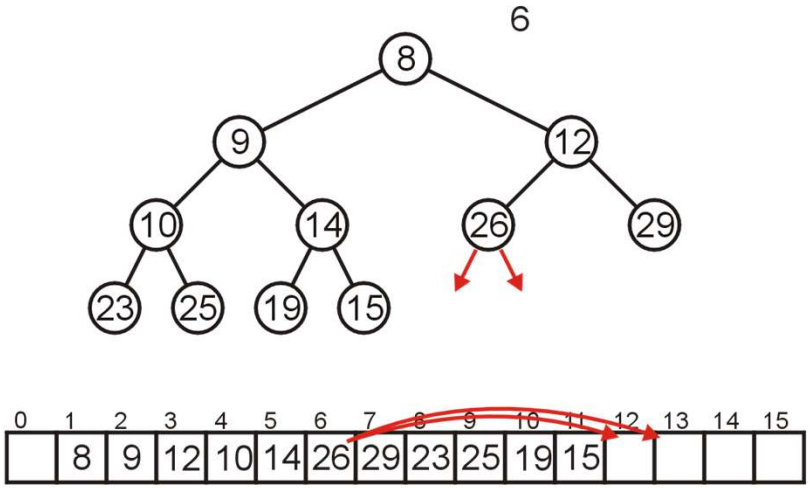
– Swap 26 and 12



Array Implementation: Pop

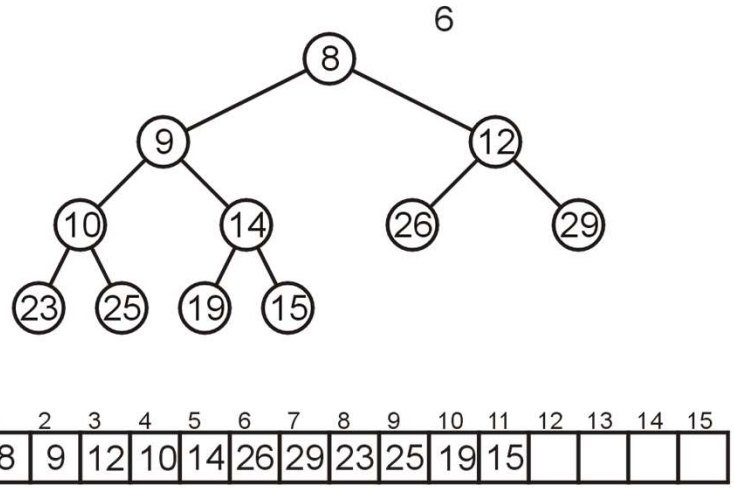
The children of Node 6, Nodes 12 and 13 are unoccupied

- Currently, `count == 11`



Array Implementation: Pop

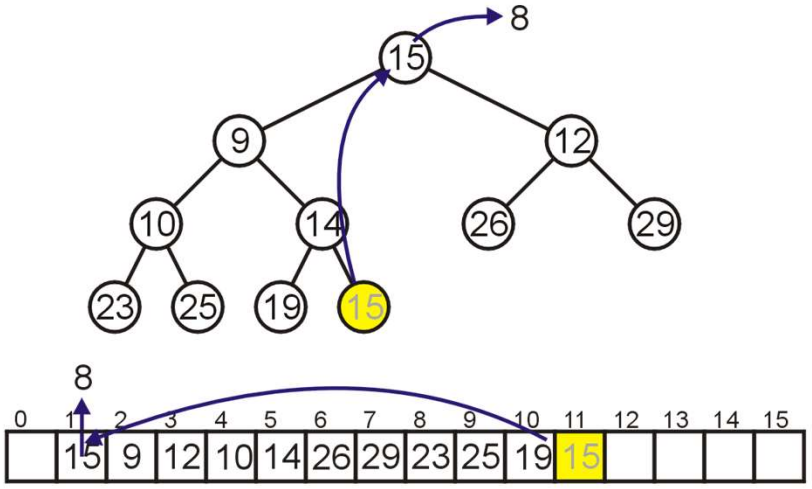
The result is a min-heap



Array Implementation: Pop

Dequeuing the minimum a third time:

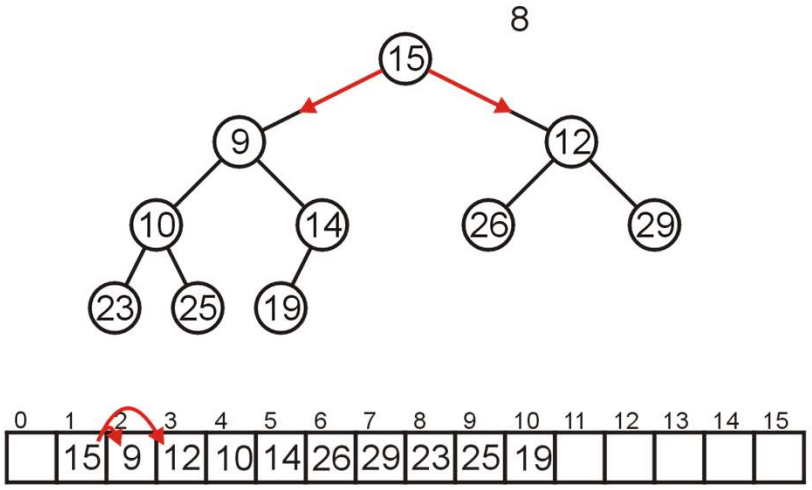
– Copy 15 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

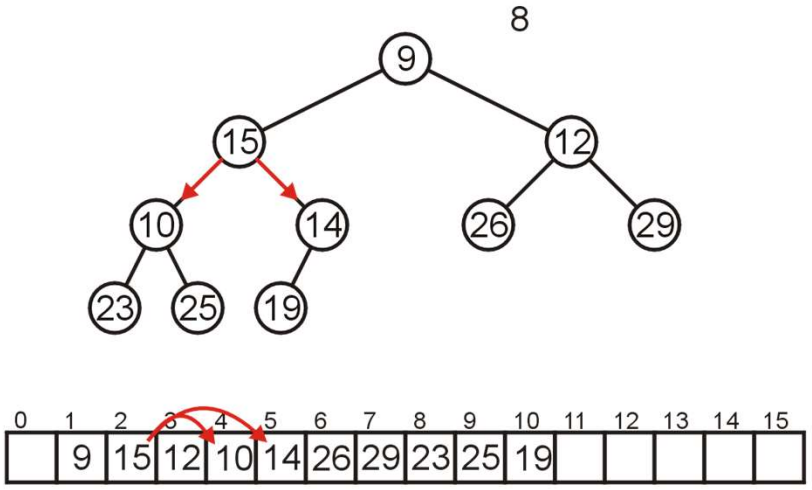
– Swap 15 and 9



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

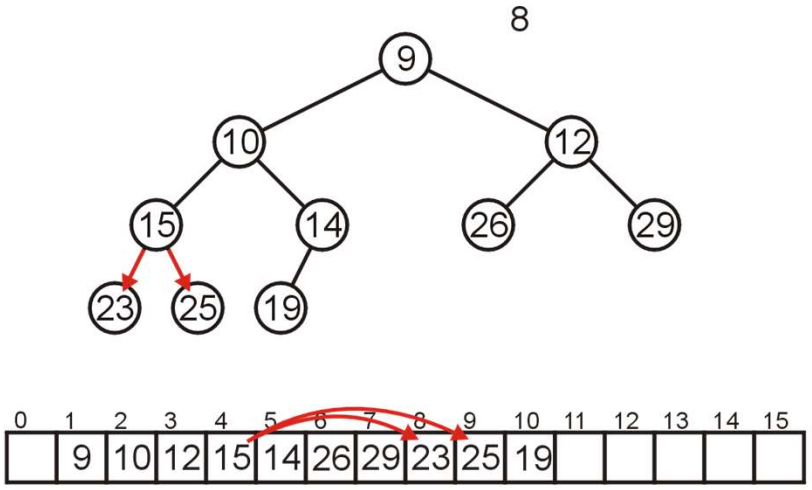
– Swap 15 and 10



Array Implementation: Pop

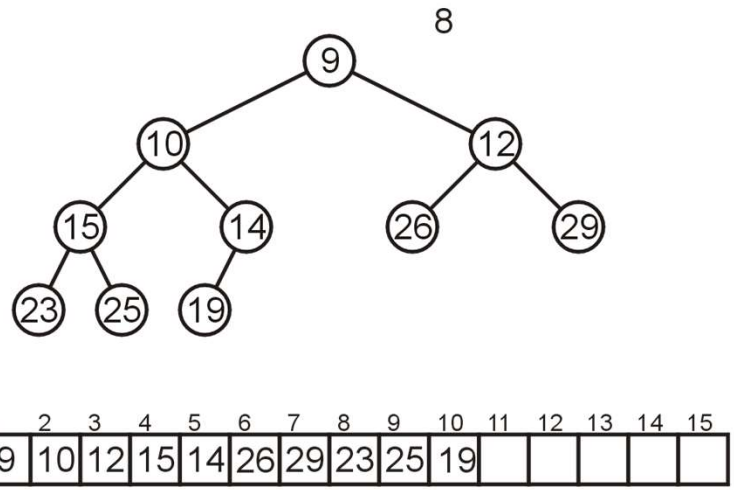
Compare Node 4 with its children: Nodes 8 and 9

– 15 < 23 and 15 < 25 so stop



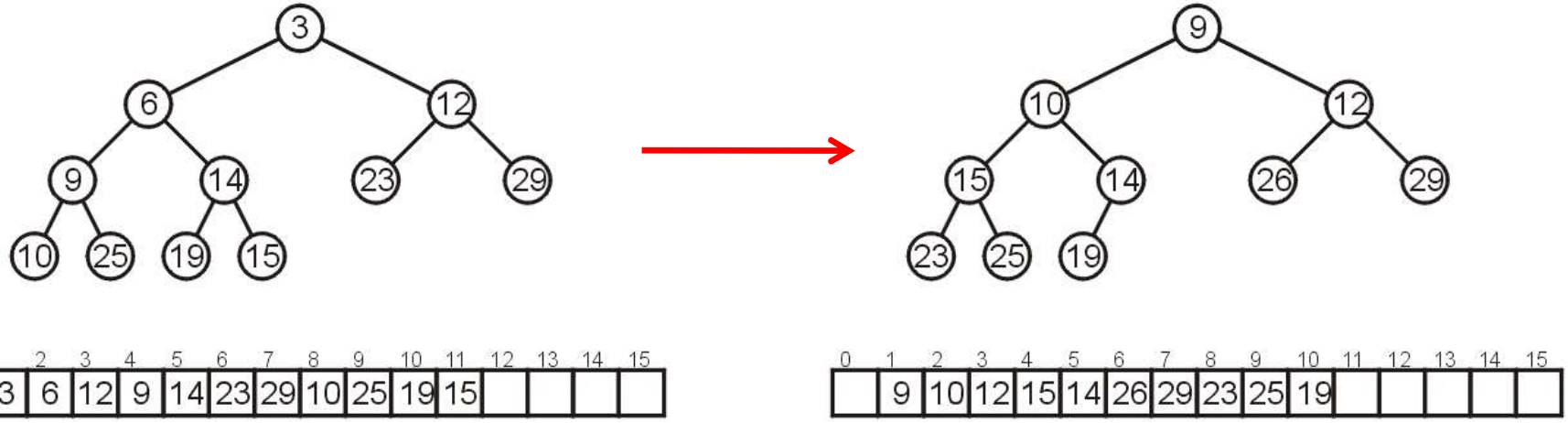
Array Implementation: Pop

The result is a properly formed binary min-heap



Array Implementation: Pop

After all our modifications, the final heap is



Run-time Analysis

Accessing the top object is $\Theta(1)$

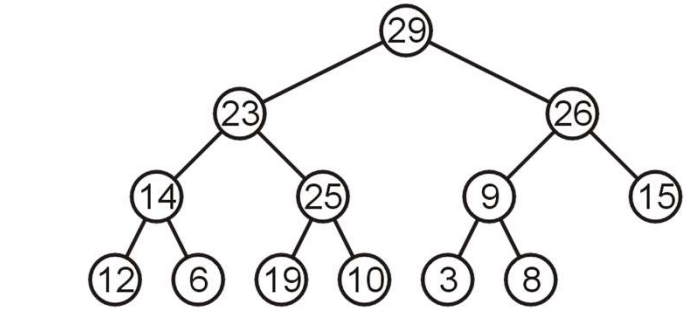
- **Pop**
  - Popping the top object is $\mathbf{O}(\log(n))$
    - We copy something that is already in the lowest depth—it may be moved back to the lowest depth

- **Push**
  - If we are inserting an object less than the root, the run time will be $\Theta(\log(n))$
  - If we insert an object greater than any of the existing objects, the run time will be $\Theta(1)$

Binary Max Heap

A binary max-heap is identical to a binary min-heap except that the parent is always larger than either of the children

For example, the same data as before stored as a max-heap yields



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	29	23	26	14	25	9	15	12	6	19	10	3	8		

Priority Queues

Consider inserting seven objects, all of the same priority (colour indicates order):

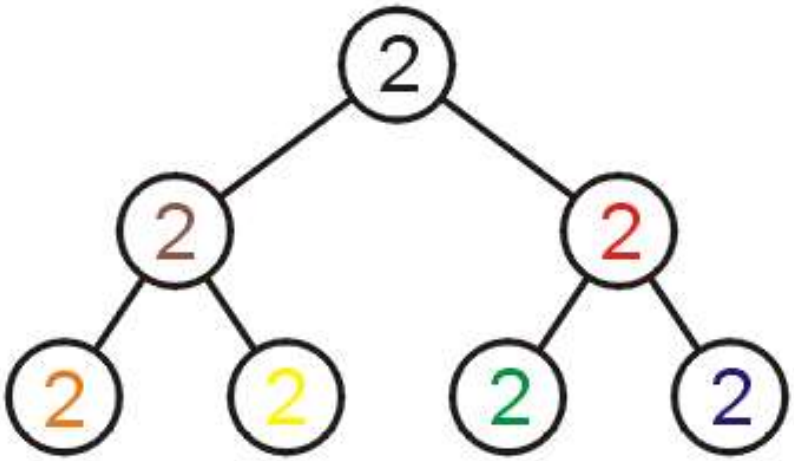
2, 2, 2, 2, 2, 2, 2

Priority Queues

Whatever algorithm we use for promoting must ensure that the first object remains in the root position

- Thus, we must use an insertion technique where we only percolate up if the priority is lower

The result:



Challenge:

- How can you possibly remove all seven objects in the order they were inserted?

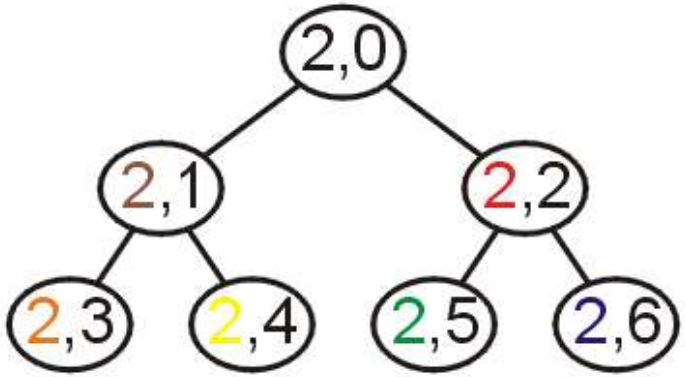
2, 2, 2, 2, 2, 2, 2

Lexicographical Ordering

A solution is to modify the priority:

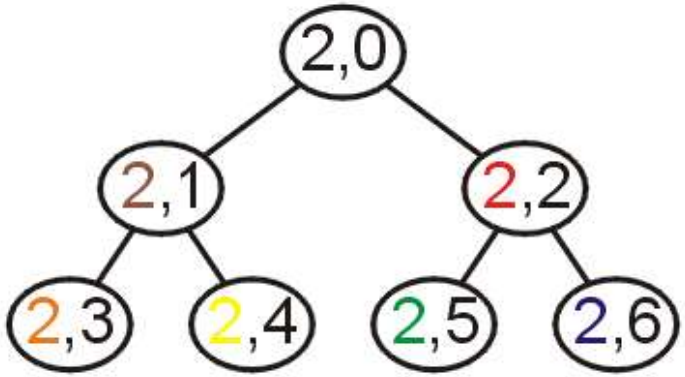
- Track the number of insertions with a counter k (initially 0)
- For each insertion with priority n, create a hybrid priority (n, k) where:

  $(n_1, k_1) < (n_2, k_2)$ if $n_1 < n_2$ or ($n_1 = n_2$ and $k_1 < k_2$)



Priority Queues

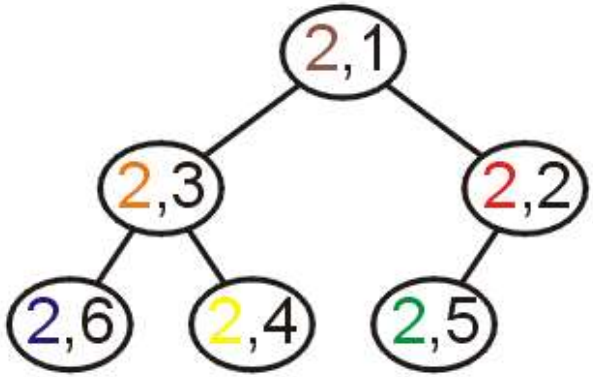
Removing the objects would be in the following order:



Priority Queues

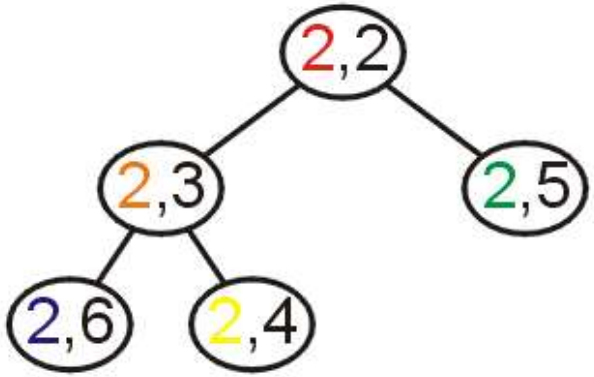
Popped: 2

- First, (2,1) < (2, 2) and (2, 3) < (2, 4)



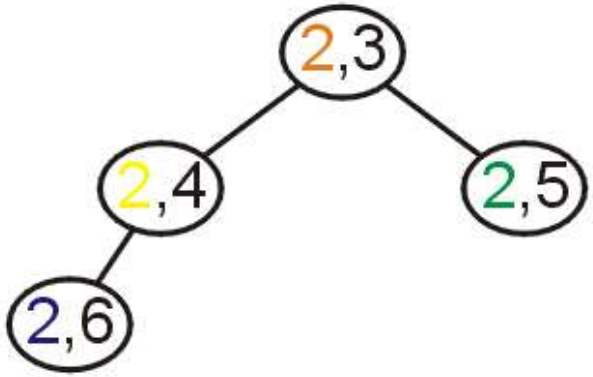
Priority Queues

Removing the objects would be in the following order:



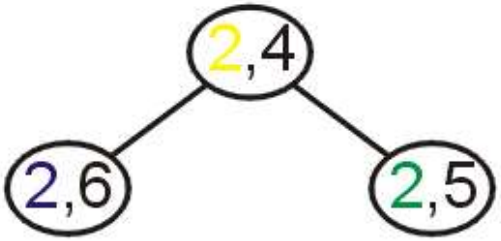
Priority Queues

Removing the objects would be in the following order:



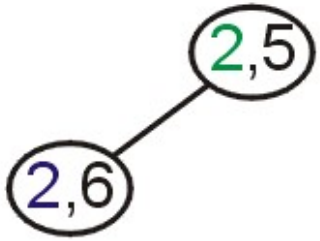
Priority Queues

Removing the objects would be in the following order:



Priority Queues

Removing the objects would be in the following order:



Heap Sort

Heap Sort

- Inserting n objects into a min-heap and then taking n objects will result in them coming out in order

- Strategy: given an unsorted list with n objects,
  - place them into a heap (heapification) and then
  - take them out

- This solution requires additional memory, i.e., a min-heap of size n
  - This requires $\Theta(n)$ memory and is therefore not in place

Time complexity: $\Theta(\mathrm{n}\ \log(n))$

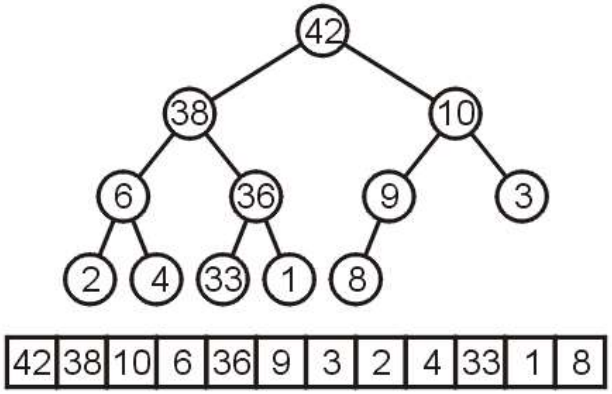
In-place Implementation

Is it possible to perform a heap sort in place, that is, require at most $\Theta(1)$ memory (a few extra variables)?

In-place Implementation

Consider a max-heap:

– A heap where the maximum element is at the top of the heap and the next to be popped

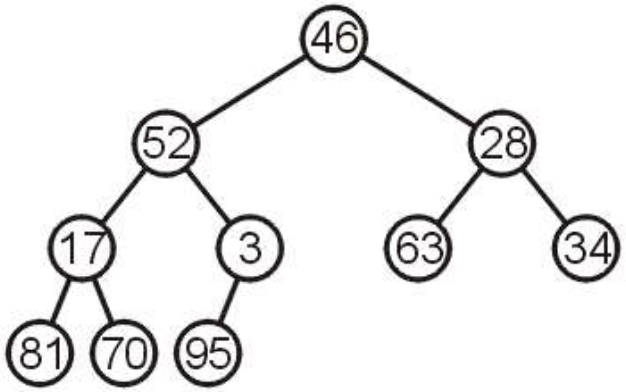


In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

This array represents the following complete tree:

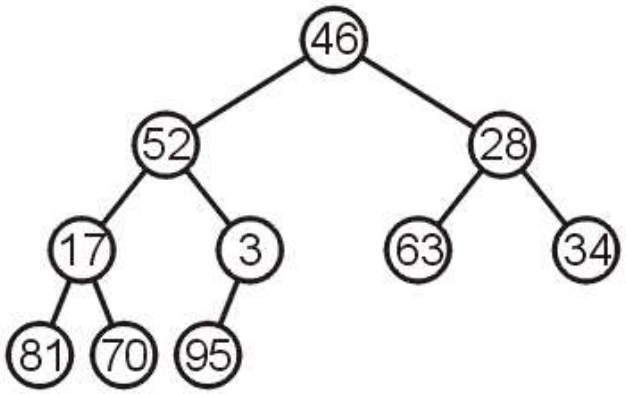


This is neither a min-heap, max-heap, or binary search tree

In-place Heapification

46	52	28	17	3	63	34	81	70	95

If arrays start at 0 (we started at entry 1 for binary heaps) , we need different formulas for the children and parent



The formulas are now:

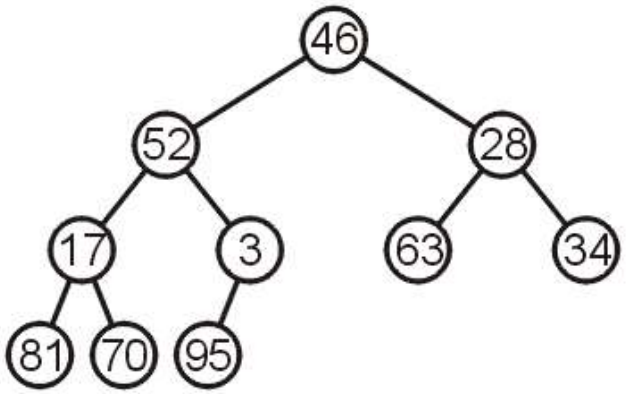
Children	`2*k + 1`	`2*k + 2`
Parent	`(k + 1)/2 - 1`	

In-place Heapification

Can we convert this complete tree into a max heap?

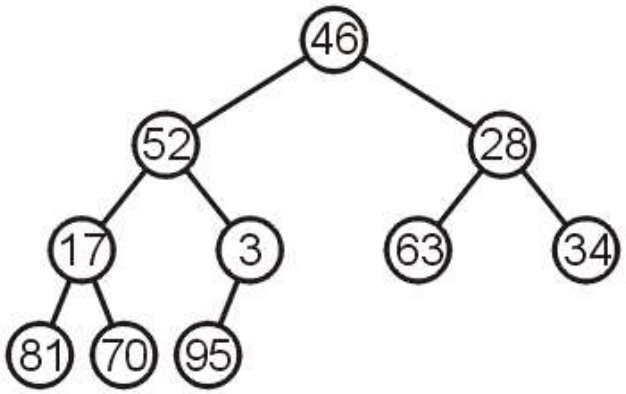
Restriction:

- The operation must be done in-place



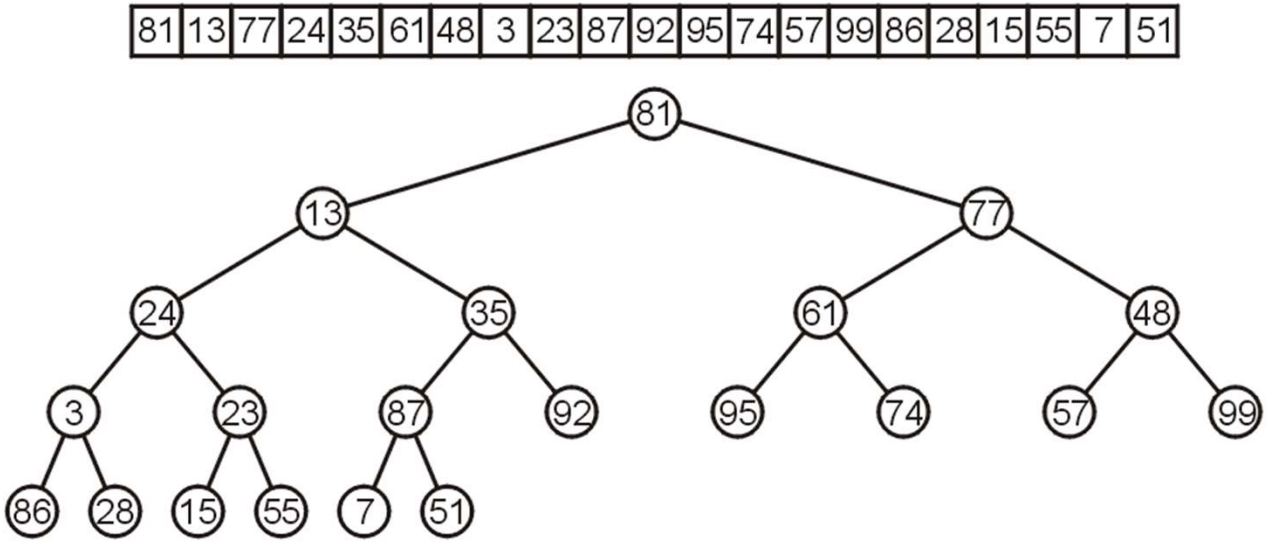
In-place Heapification

- Strategy
  - Start from the leaf nodes, all leaf nodes are already max heaps, and then make corrections so that previous nodes also form max heaps



In-place Heapification

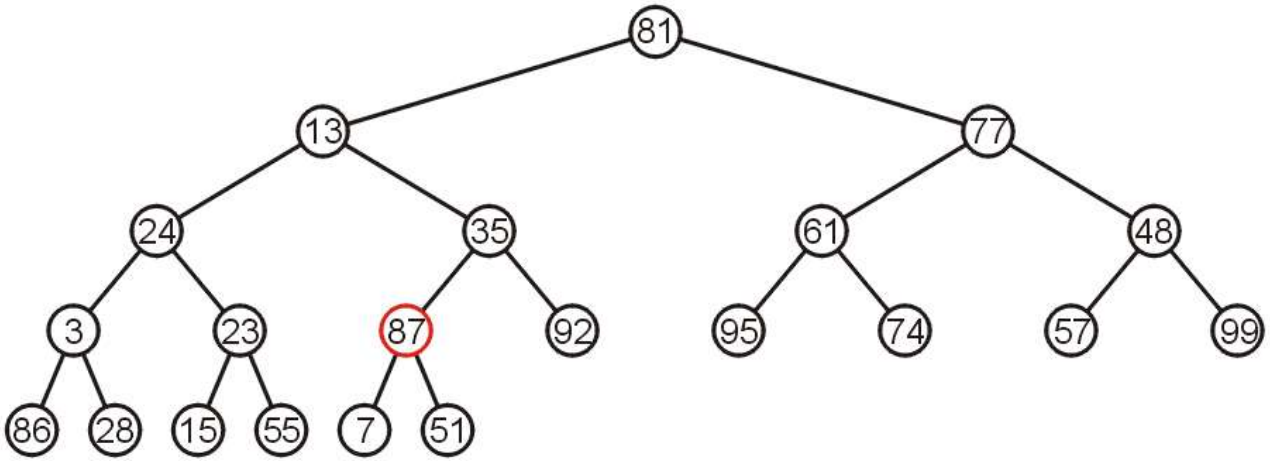
Each leaf node is a max heap on its own



In-place Heapification

All leaf nodes are heaps

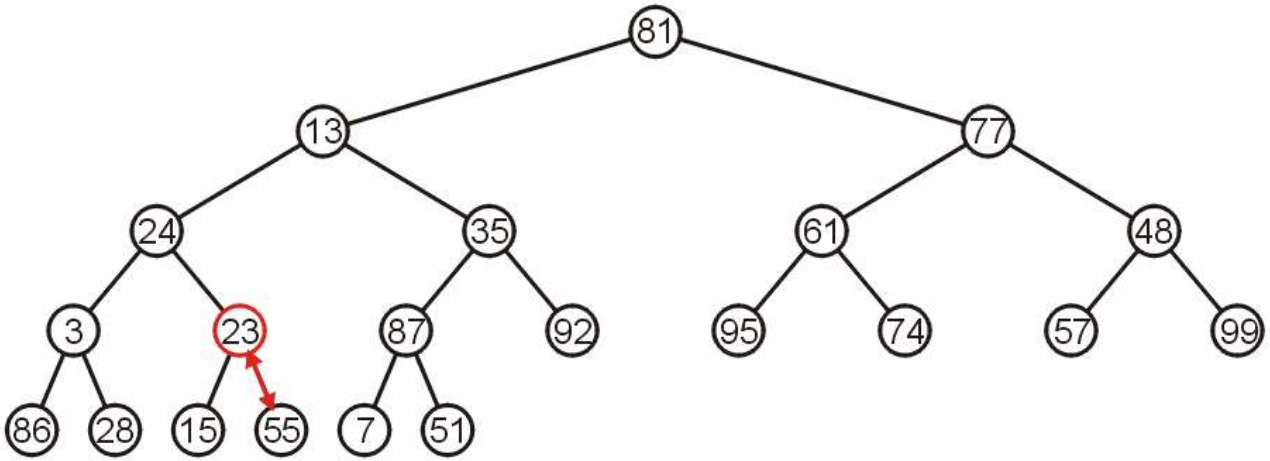
Also, the subtree with 87 as the root is a max-heap



In-place Heapification

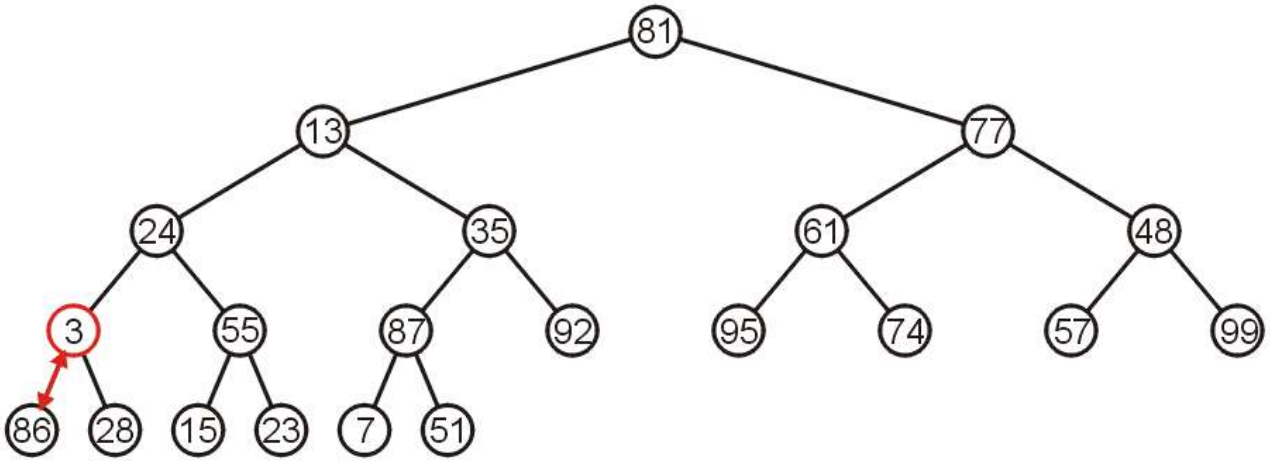
The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap

This process is termed *percolating down*



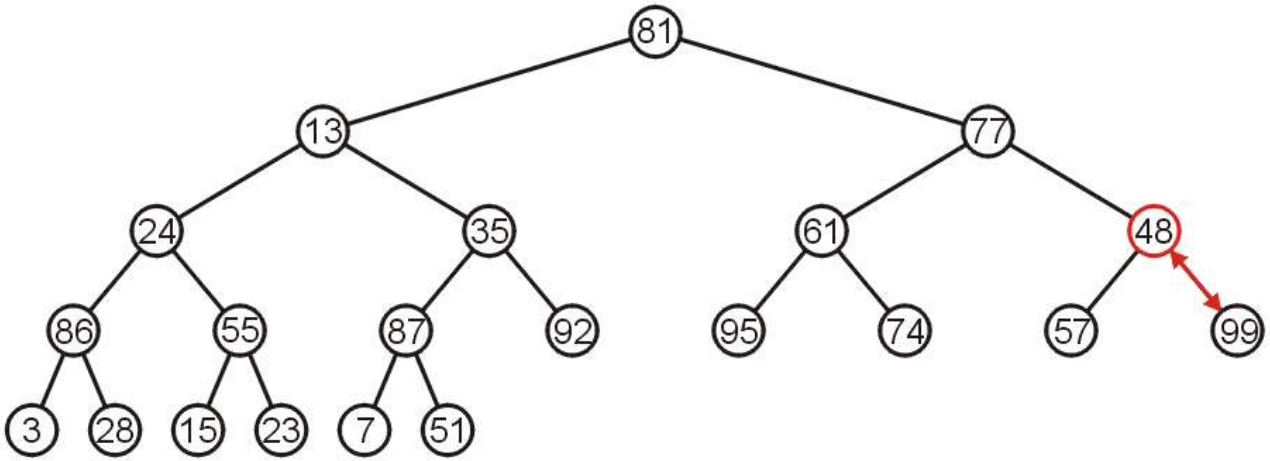
In-place Heapification

The subtree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86



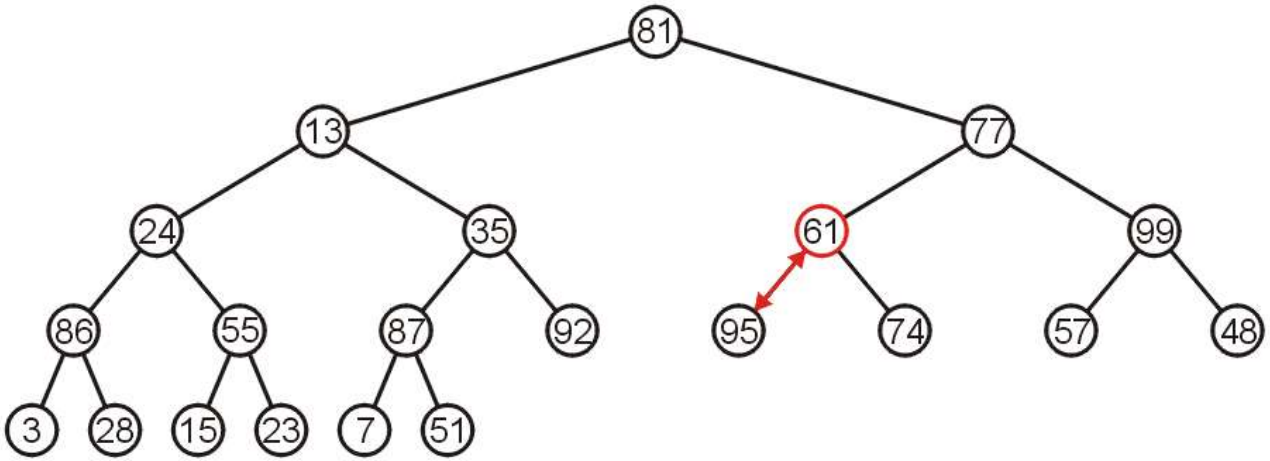
In-place Heapification

Starting with the next higher level, the subtree with root 48 can be turned into a max-heap by swapping 48 and 99



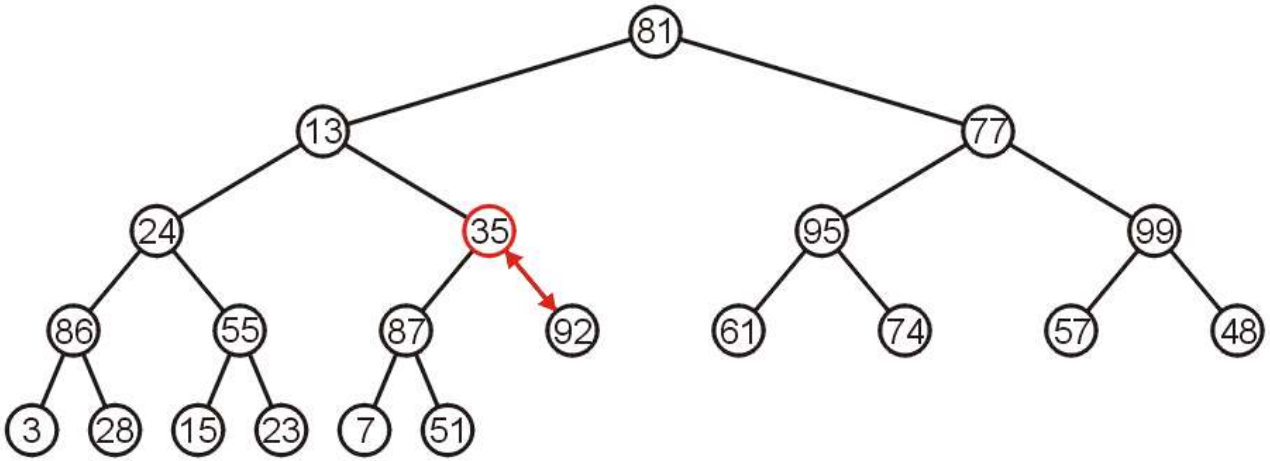
In-place Heapification

Similarly, swapping 61 and 95 creates a max-heap of the next subtree



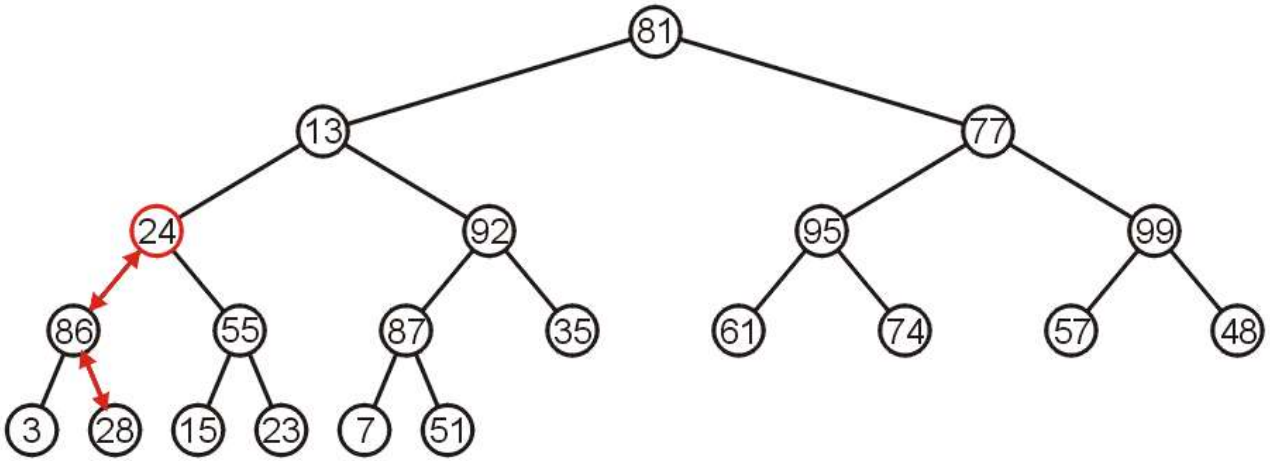
In-place Heapification

As does swapping 35 and 92



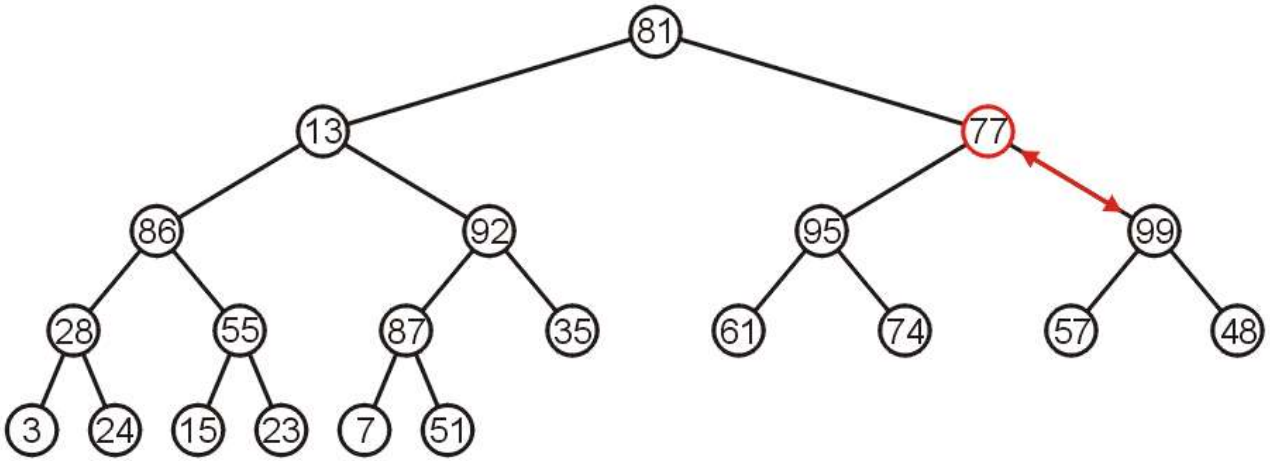
In-place Heapification

The subtree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28



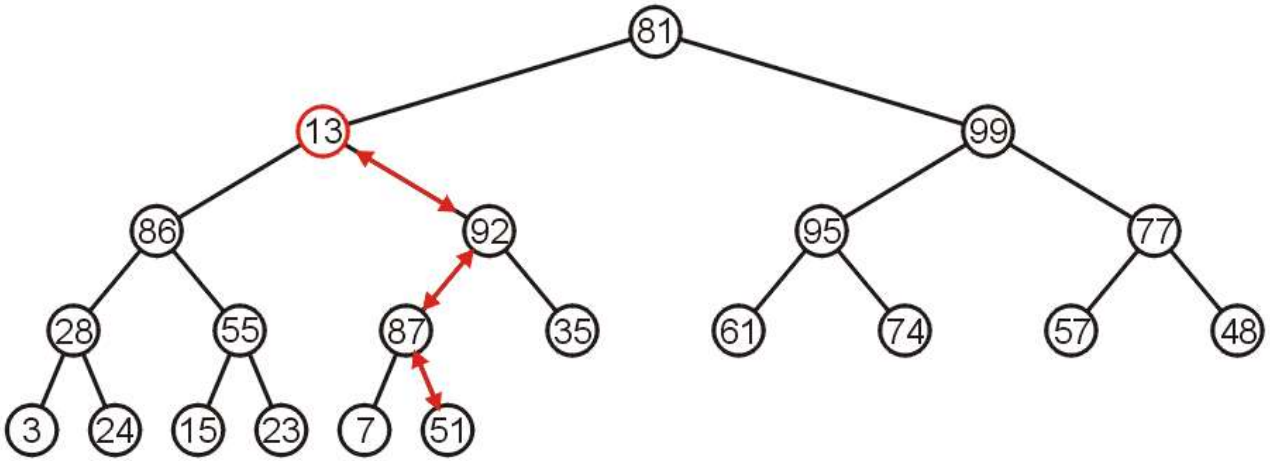
In-place Heapification

The right-most subtree of the next higher level may be turned into a max-heap by swapping 77 and 99



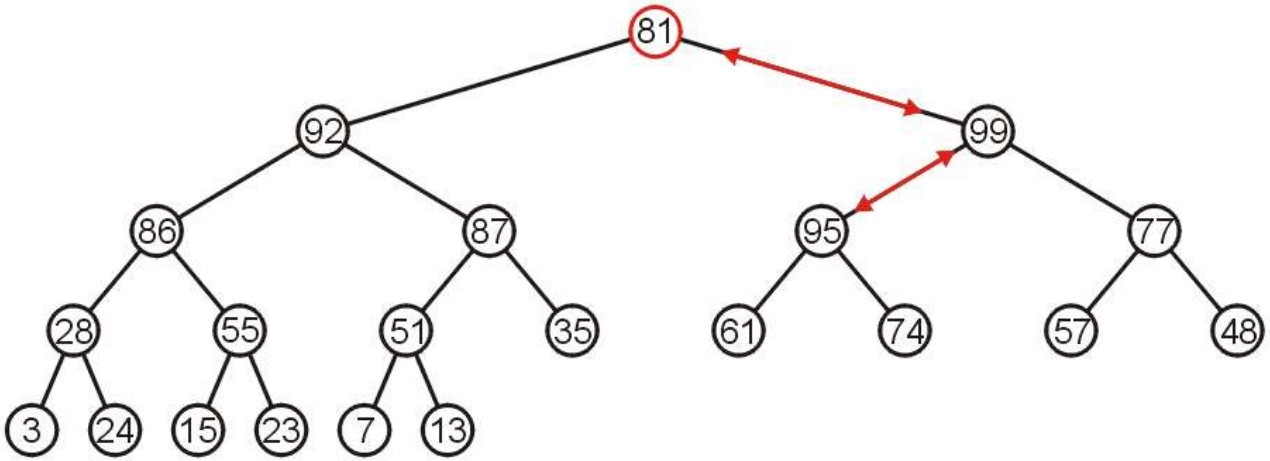
In-place Heapification

However, to turn the next subtree into a max-heap requires that 13 be percolated down to a leaf node



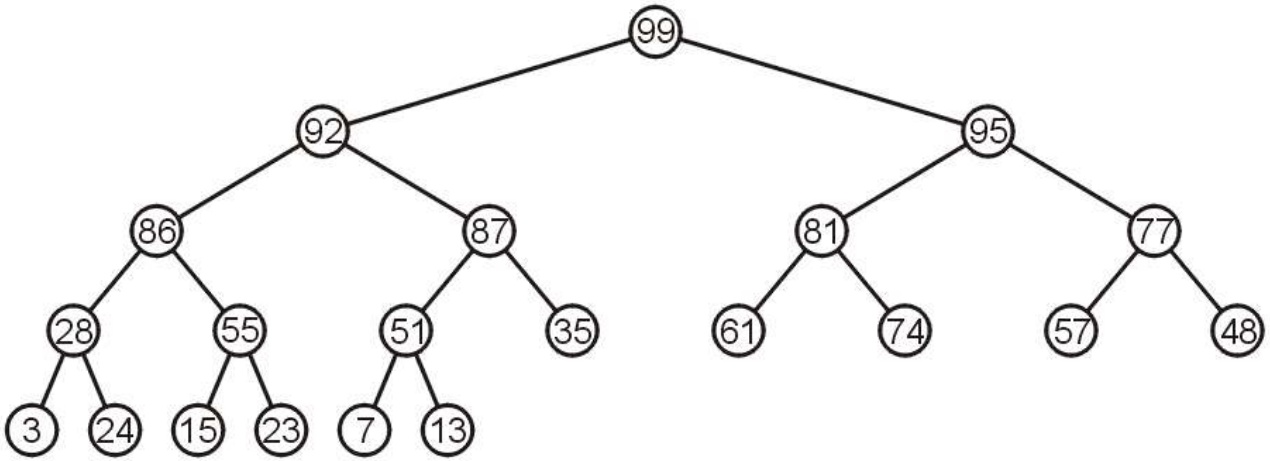
In-place Heapification

The root need only be percolated down by two levels



In-place Heapification

The final product is a max-heap



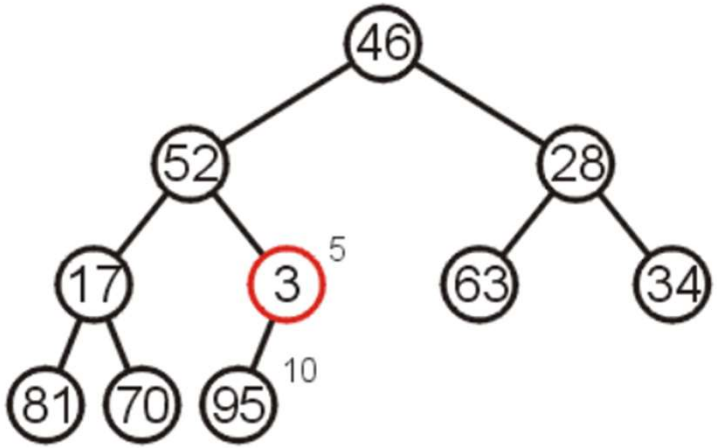
Example Heap Sort

Let us look at this example: we must convert the unordered array with $n = 10$ elements into a max-heap

46	52	28	17	3	63	34	81	70	95

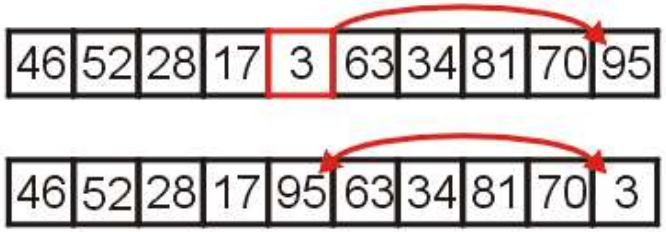
None of the leaf nodes need to be percolated down, and the first non-leaf node is in position $n/2$

Thus we start with position $10/2 = 5$



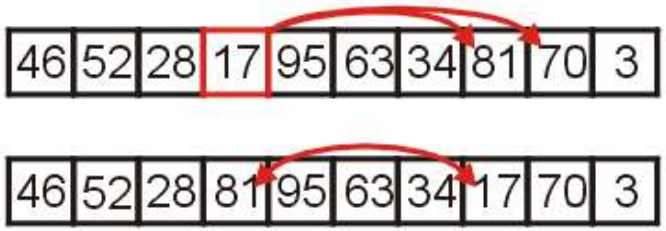
Example Heap Sort

We compare 3 with its child and swap them



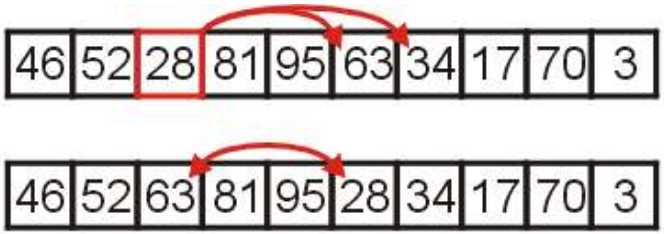
Example Heap Sort

We compare 17 with its two children and swap it with the maximum child (70)



Example Heap Sort

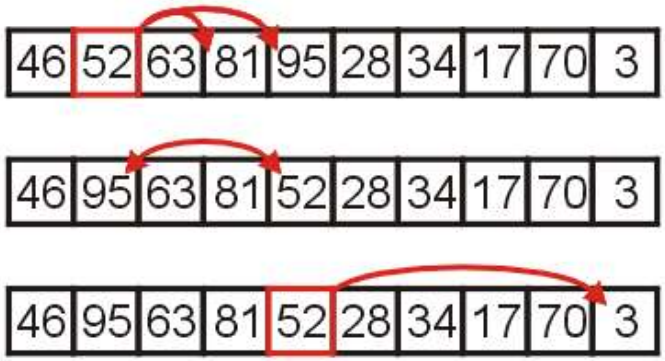
We compare 28 with its two children, 63 and 34, and swap it with the largest child



Example Heap Sort

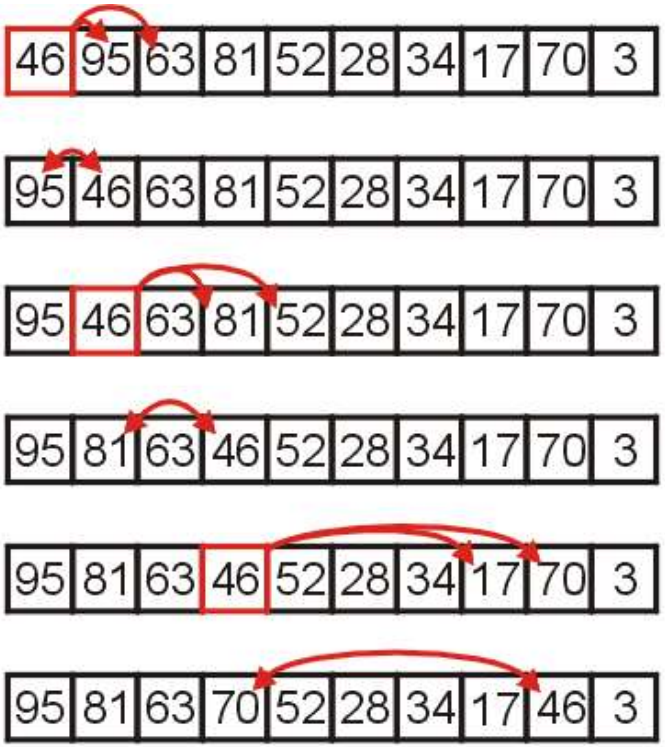
We compare 52 with its children, swap it with the largest

– Recursing, no further swaps are needed



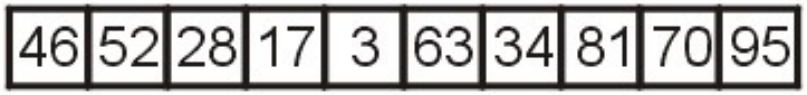
Example Heap Sort

Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70

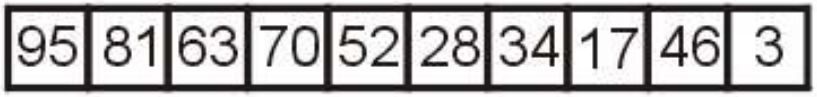


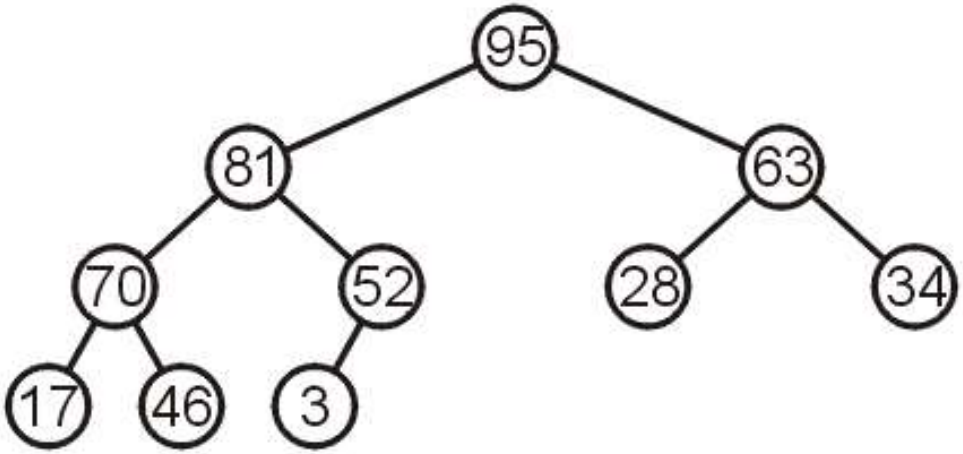
Heap Sort Example

We have now converted the unsorted array



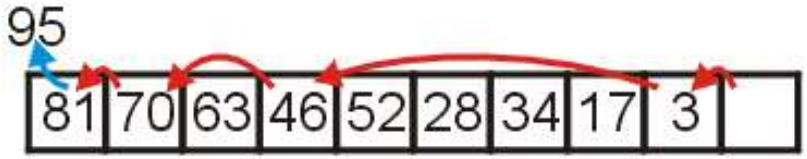
into a max-heap:



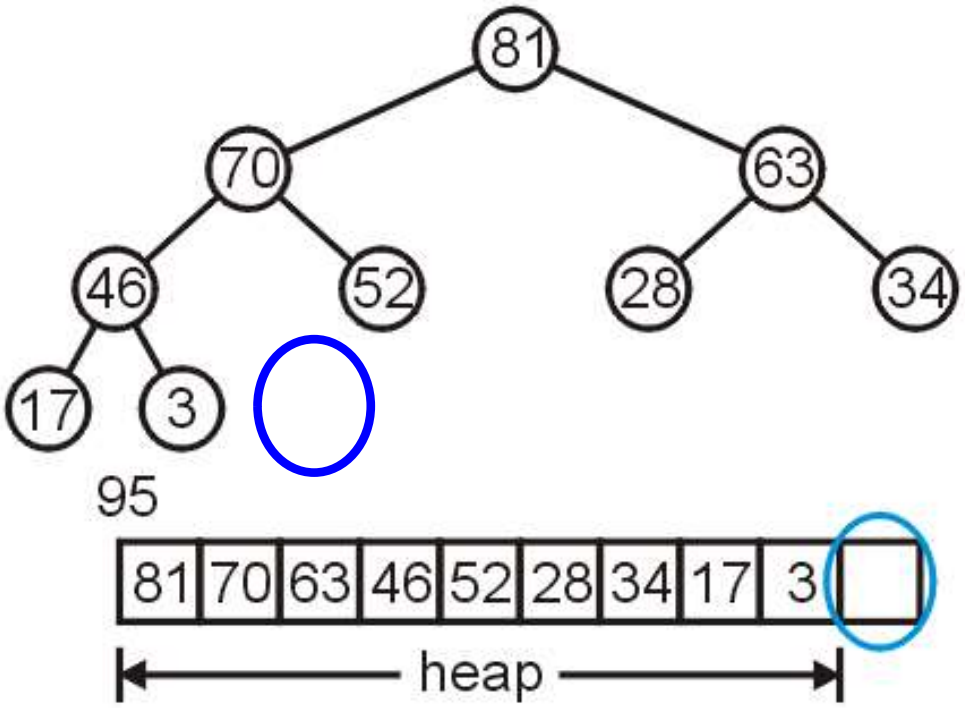


Heap Sort Example

Suppose we pop the maximum element of this heap

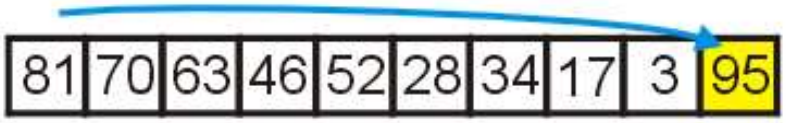


This leaves a gap at the back of the array:

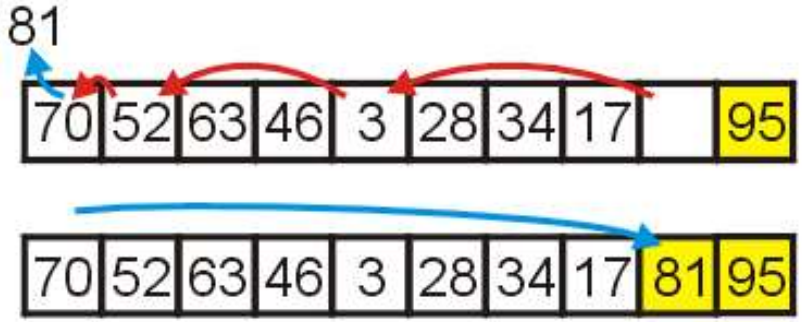


Heap Sort Example

This is the last entry in the array, so why not fill it with the largest element?



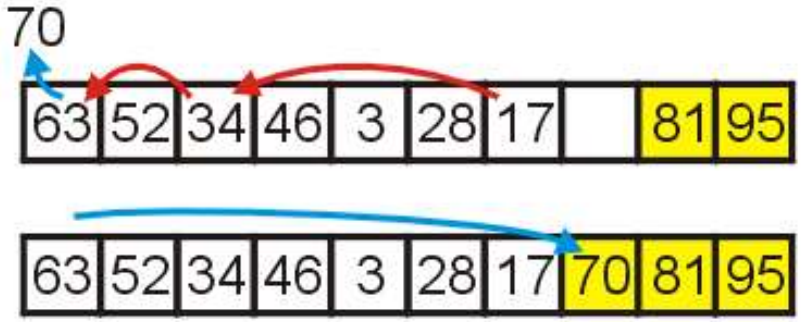
Repeat this process: pop the maximum element, and then insert it at the end of the array:



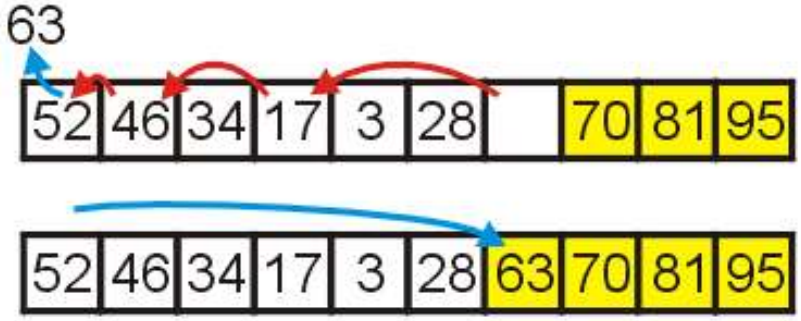
Heap Sort Example

Repeat this process

– Pop and append 70



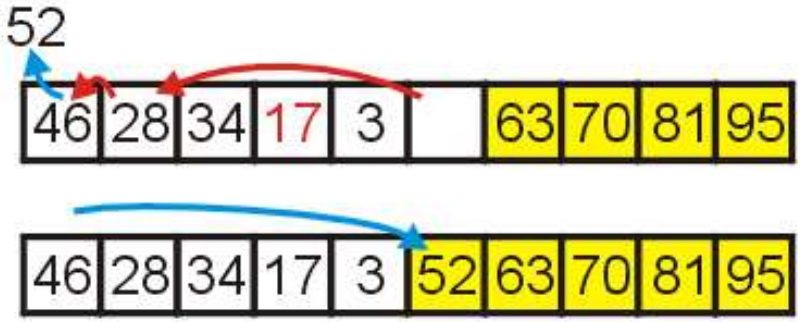
– Pop and append 63



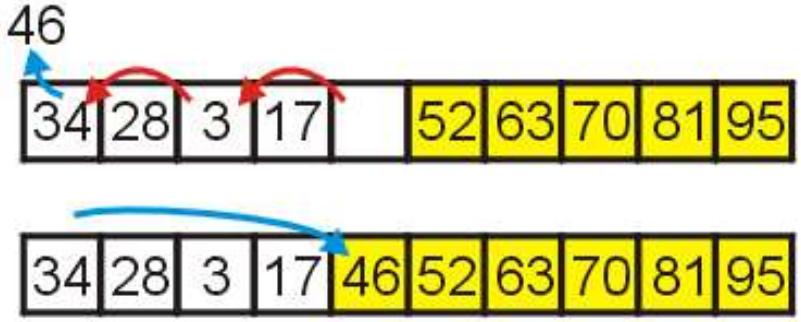
Heap Sort Example

We have the 4 largest elements in order

– Pop and append 52



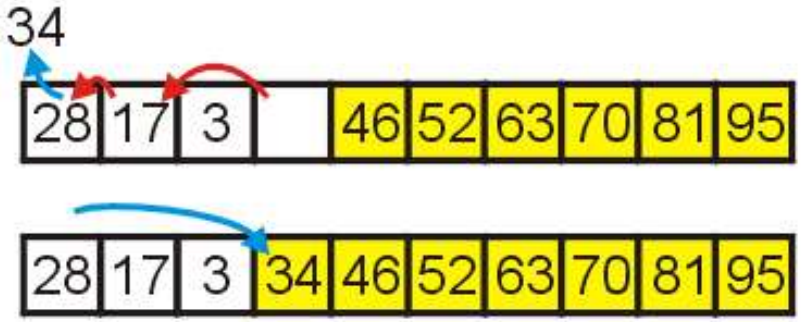
– Pop and append 46



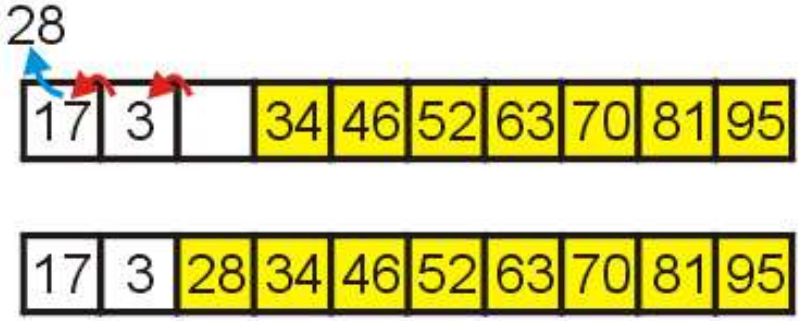
Heap Sort Example

Continuing...

– Pop and append 34

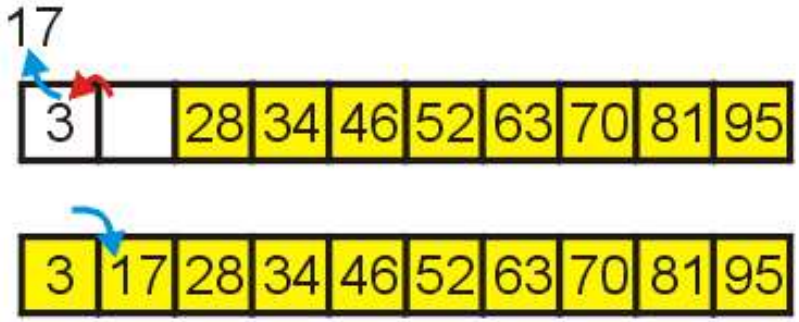


– Pop and append 28



Heap Sort Example

Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted



Heap Sort

Heapification runs in $\Theta(n)$

Popping n items from a heap of size n, runs in $\Theta(n \log(n))$ time

Therefore, the total algorithm will run in $\Theta(n \log(n))$ time

References and Acknowledgements

- The content provided in the slides are borrowed from different sources including Goodrich's book on Data Structures and Algorithms in C++, Cormen's book on Introduction to Algorithms, Weiss's book , Data Structures and Algorithm Analysis in C++, 3rd Ed., Algorithms and Data Structures at University of Waterloo (https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/) and https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html.
- The primary source of slides is https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials, courtesy of Douglas Wilhelm Harder.