CS101 Data Structure

Heaps and Priority Queues

Textbook Ch 6

Outline

- Priority queue
- Binary heap
- Heapsort

Outline

This topic will:

- Review queues
- Discuss the concept of priority and priority queues
- Look at two simple implementations:
  - Arrays of queues
  - AVL trees
- Introduce heaps, an alternative tree structure which has better run-time characteristics

Background

We have discussed Abstract Lists with explicit linear orders

- Arrays, linked lists, strings

We saw three cases which restricted the operations:

- Stacks, queues, deques

Following this, we looked at search trees for storing implicit linear orders: Abstract Sorted Lists

- Run times were generally $\Theta(\ln(n))$

We will now look at a restriction on an implicit linear ordering:

- Priority queues

Definition

Queues

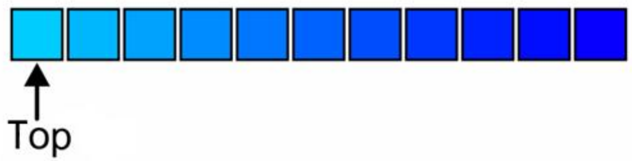
- The order may be summarized by *first in, first out*

Priority queues

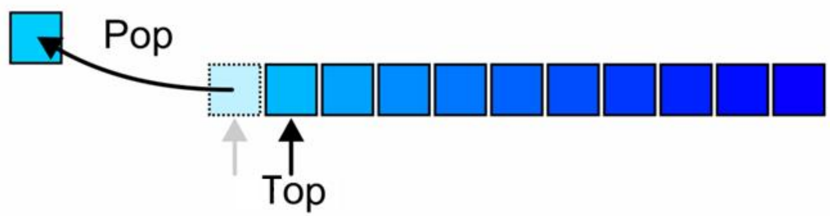
- Each object is associated with a priority
  - The value 0 has the *highest* priority, and
  - The higher the number, the lower the priority
- We pop the object which has the highest priority

Operations

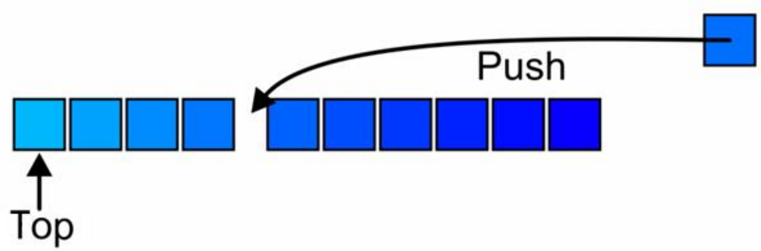
The top of a priority queue is the object with highest priority



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



Push places a new object into the appropriate place



Lexicographical Priority

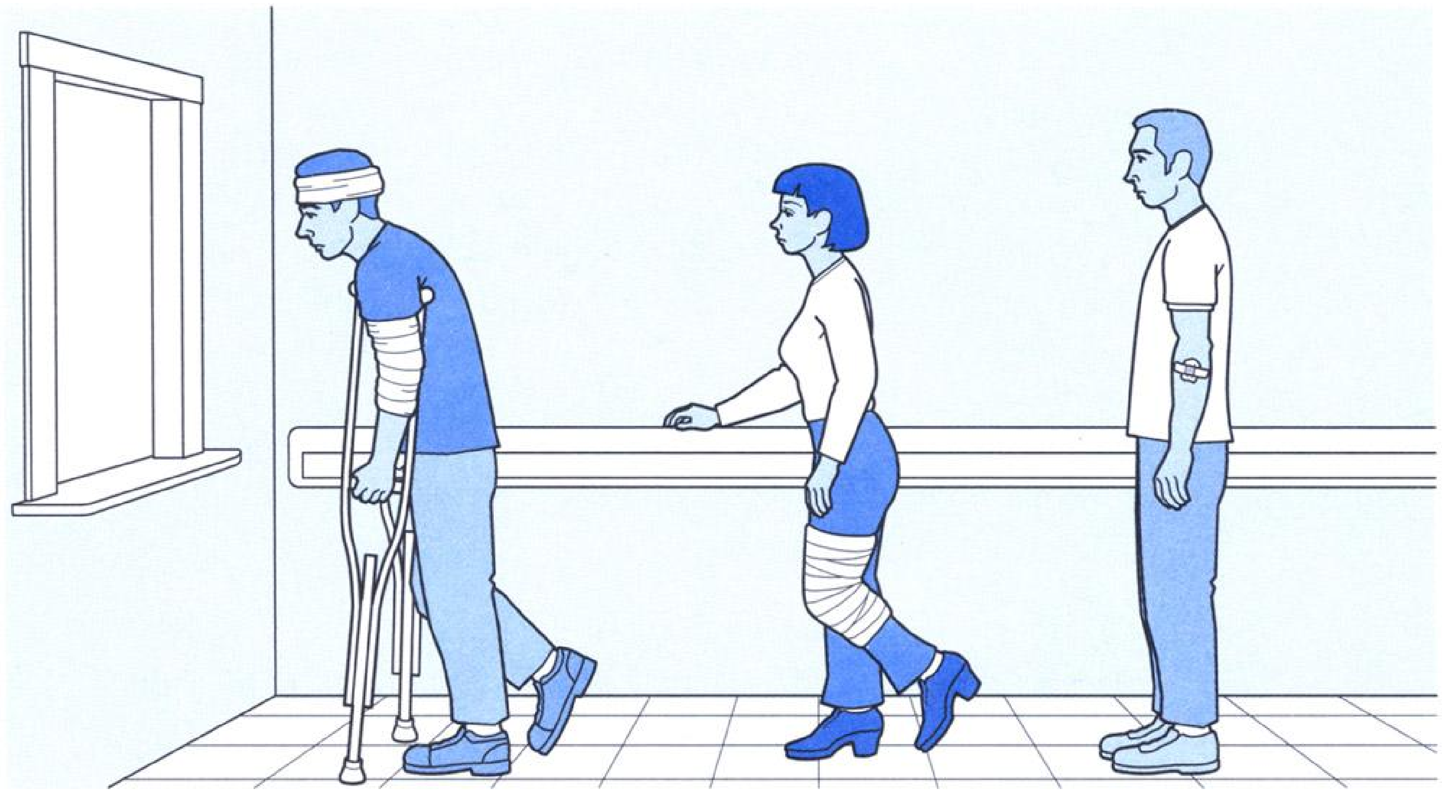
Priority may also depend on multiple variables:

- Two values 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)

Application



Application

Process priority in operation systems

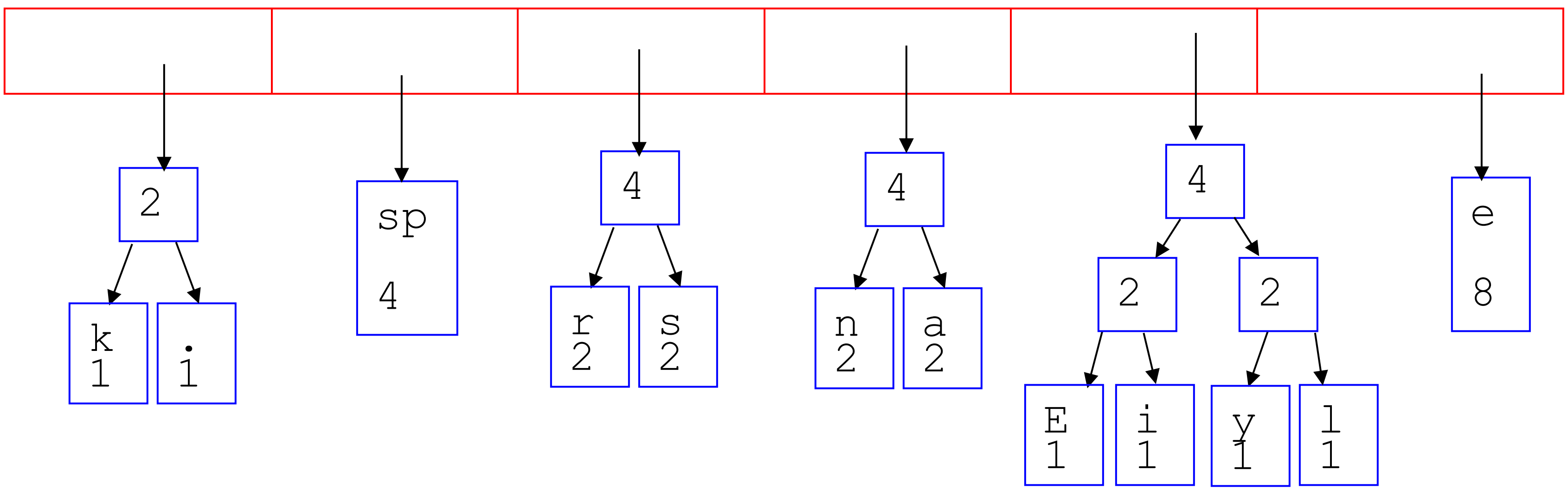
- In Unix, you may set the priority of a process, e.g.,

  % nice +15 ./a.out

  reduces the priority of the execution of the routine a.out by 15

Application

We will see later how priority queue is used in Huffman coding.



Implementations

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

We will look at an implementation using a data structure we already know:

– Multiple queues — one for each priority

Then we will introduce a more appropriate data structure: *heap*

Multiple Queues

Assume there is a fixed number of priorities, say M

- Create an array of M queues
- Push a new object onto the queue corresponding to the priority
- Top and pop find the first non-empty queue with highest priority

Multiple Queues

```
template <typename Type, int M>
class Multiqueue {
    private:
        queue<Type> queue_array[M];
        int queue_size;
    public:
        Multiqueue();
        bool empty() const;
        Type top() const;
        void push( Type const &, int );
        Type pop();
};

template <typename Type, int M>
Multiqueue<Type>::Multiqueue():
queue_size( 0 ) {
    // The compiler allocates memory for the M queues
    // and calls the constructor on each of them
}

template <typename Type, int M>
bool Multiqueue<Type>::empty()  const{
    return ( queue_size == 0 );
}
```

Multiple Queues

```
template <typename Type, int M>
void Multiqueue<Type>::push( Type const &obj, int pri ) {
    if ( pri < 0 || pri >= M ) {
        throw illegal_argument();
    }

    queue_array[pri].push( obj );
    ++queue_size;
}

template <typename Type, int M>
Type Multiqueue<Type>::top() const {
    for ( int pri = 0; pri < M; ++pri ) {
        if ( !queue_array[pri].empty() ) {
            return queue_array[pri].front();
        }
    }

    // The priority queue is empty
    throw underflow();
}
```

```
template <typename Type, int M>
Type Multiqueue<Type>::pop() {
    for ( int pri = 0; pri < M; ++pri ) {
        if ( !queue_array[pri].empty() ) {
            --queue_size;
            return queue_array[pri].pop();
        }
    }

    // The priority queue is empty
    throw underflow();
}
```

Multiple Queues

The run times are reasonable:

- Push is $\Theta(1)$
- Top and pop are both $\mathbf{O}(M)$

Problems:

- It restricts the range of priorities
- The memory requirement is $\Theta(M + n)$

AVL Trees

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

– Insertion is $\Theta(\ln(n))$ `void insert( Type const & );`
– Top is $\Theta(\ln(n))$ `Type front();`
– Remove is $\Theta(\ln(n))$ `bool remove( front() );`

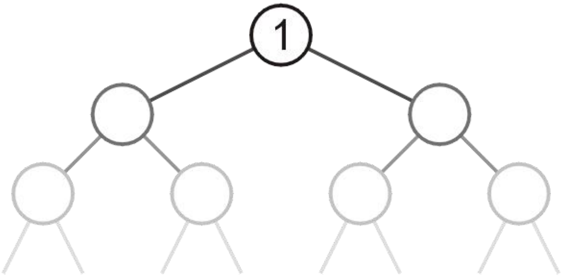
There is significant overhead for maintaining both the tree and the corresponding balance

Heaps

Can we do better?

We need a *heap*

- A tree with the top object at the root
- We will look at binary heaps
- Numerous other heaps exists:
  - d-ary heaps
  - Leftist heaps
  - Skew heaps
  - Binomial heaps
  - Fibonacci heaps
  - Bi-parental heaps



Summary

This topic:

- Introduced priority queues
- Considered two obvious implementations:
  - Arrays of queues
  - AVL trees
- Discussed the run times and claimed that a variation of a tree, a heap, can do better

References

Cormen, Leiserson, Rivest and Stein,
Introduction to Algorithms, The MIT Press, 2001, §6.5, pp.138-44.

Mark A. Weiss,
Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, 2006, Ch.6, p.213.

Joh Kleinberg and Eva Tardos,
Algorithm Design, Pearson, 2006, §2.5, pp.57-65.

Elliot B. Koffman and Paul A.T. Wolfgang,
Objects, Abstractions, Data Structures and Design using C++, Wiley, 2006, §8.5, pp.489-96

Outline

- Priority queue
- Binary heap
- Heapsort

Outline

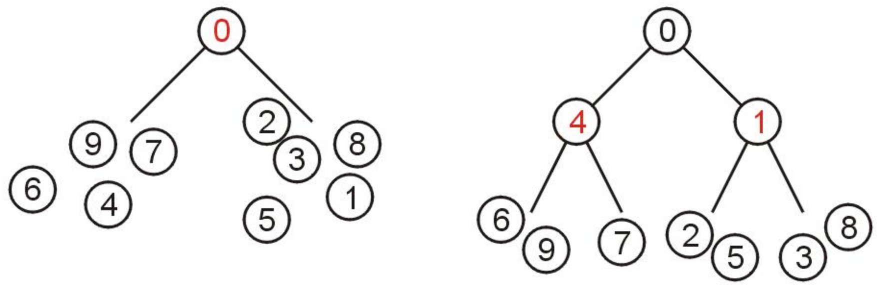
In this topic, we will:

- Define a binary min-heap
- Look at some examples
- Operations on heaps:
  - Top
  - Pop
  - Push
- An array representation of heaps
- Define a binary max-heap
- Using binary heaps as priority queues

Definition

A non-empty tree is a min-heap if

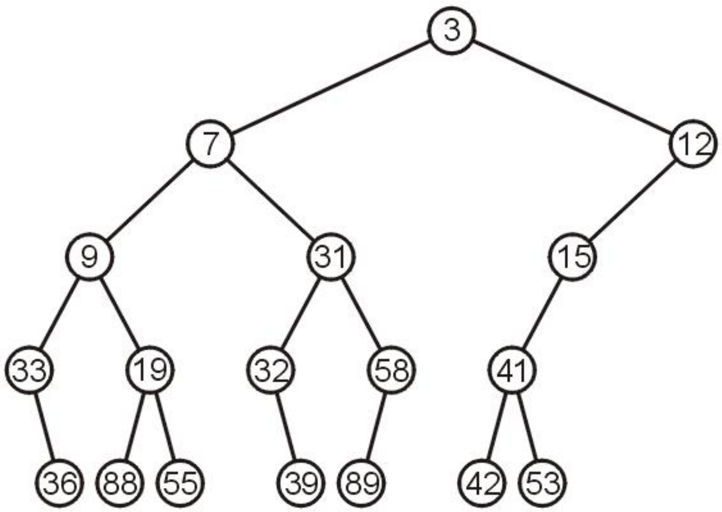
- The key associated with the root is less than or equal to the keys associated with the sub-trees (if any)
- The sub-trees (if any) are also min-heaps



There is no other relationship between the elements in the subtrees!

Example

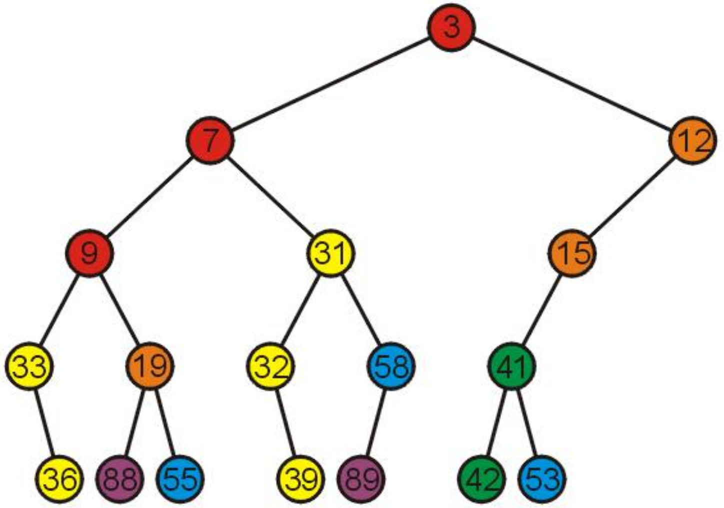
This is a (*naive*) binary min-heap:



Example

Adding colour, we observe

- The left subtree has the smallest (7) and the largest (89) objects
- No relationship between items with similar priority



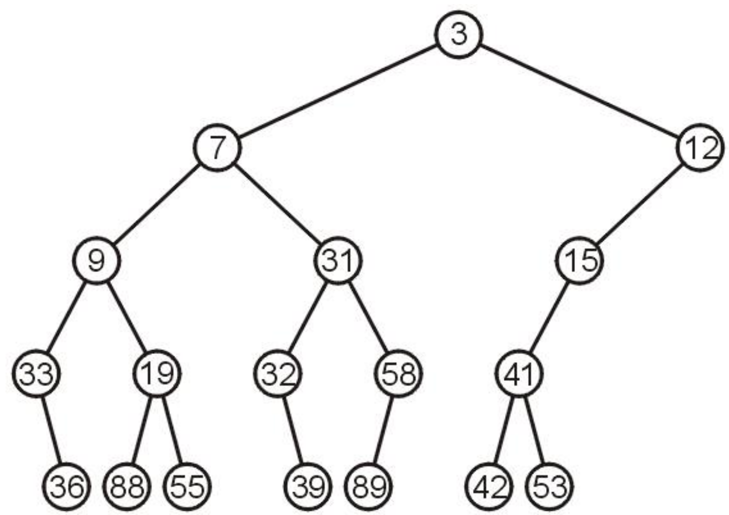
Operations

We will consider three operations:

- Top
- Pop
- Push

Example

We can find the top object in $\Theta(1)$ time: 3



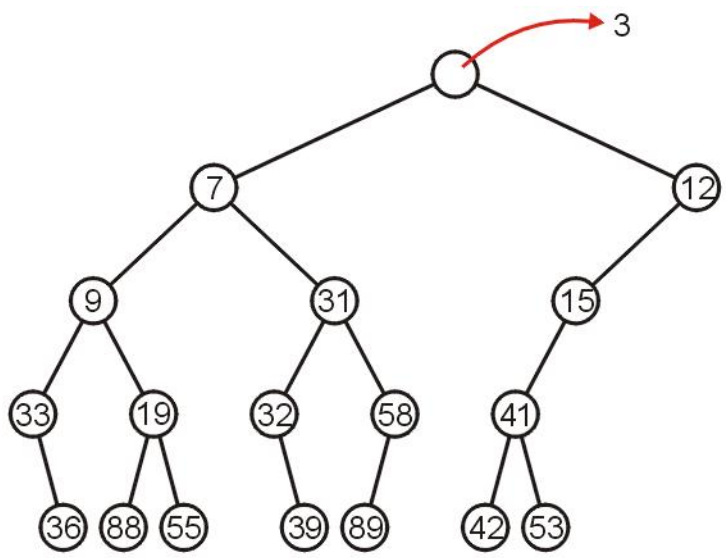
Pop

To remove the minimum object:

- Promote the node of the sub-tree which has the least value
- Recursively process the sub-tree from which we promoted the least value

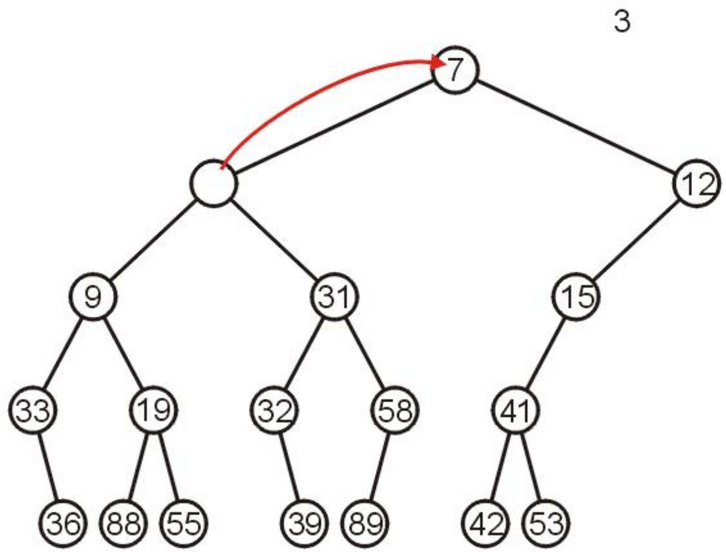
Pop

Using our example, we remove 3:



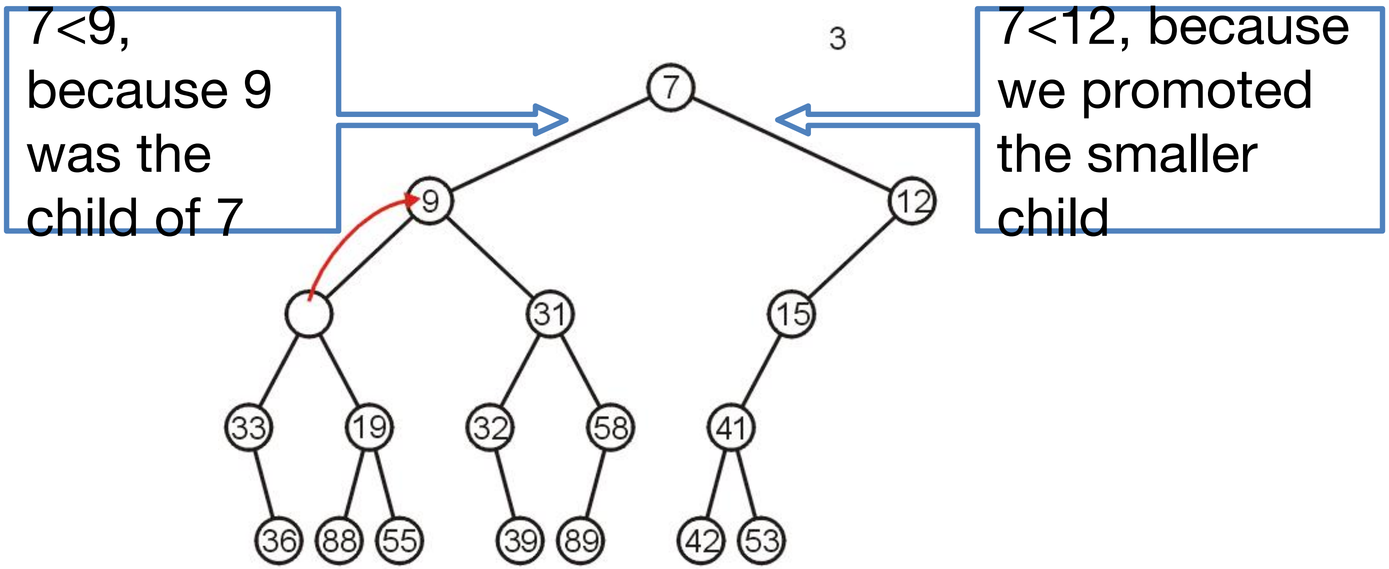
Pop

We promote 7 (the minimum of 7 and 12) to the root:



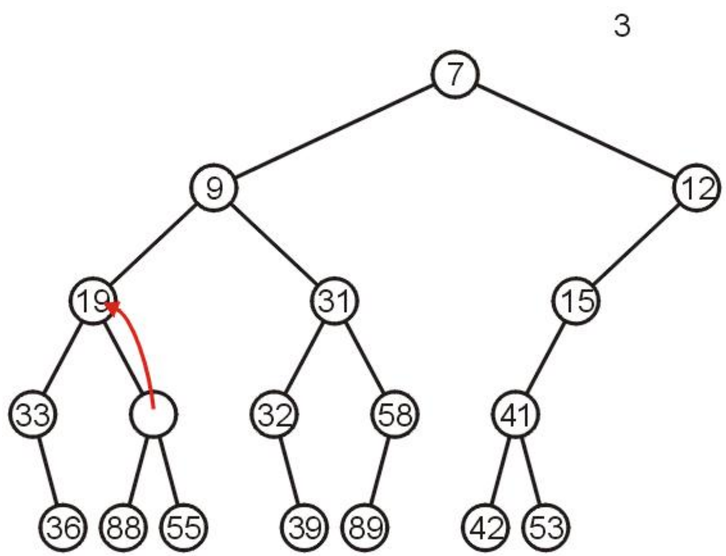
Pop

In the left sub-tree, we promote 9:



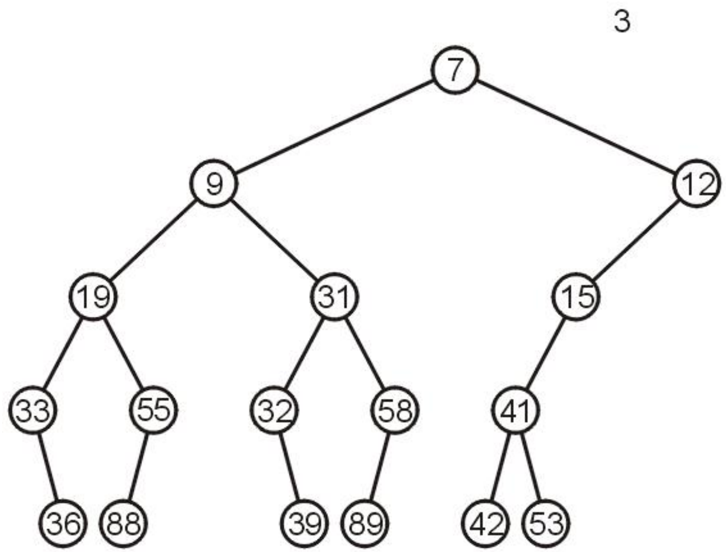
Pop

Recursively, we promote 19:



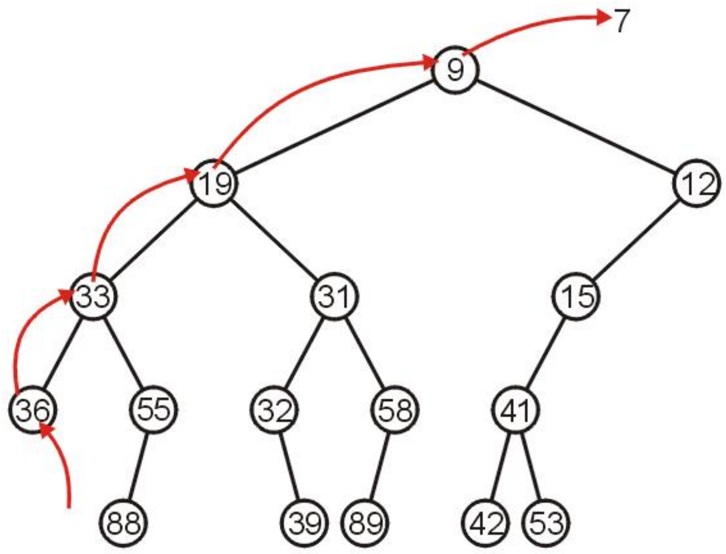
Pop

Finally, 55 is a leaf node, so we promote it and delete the leaf



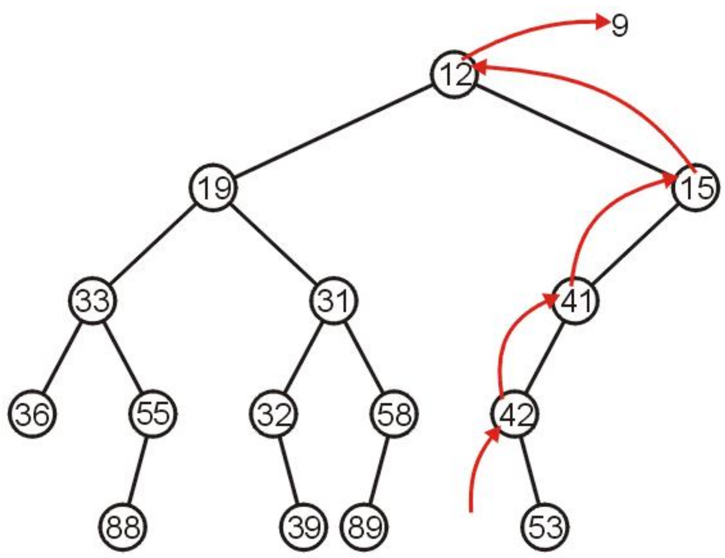
Pop

Repeating this operation again, we can remove 7:



Pop

If we remove 9, we must now promote from the right sub-tree:



Push

Inserting into a heap may be done either:

- At a leaf (move it up if it is smaller than the parent)
- At the root (insert the larger object into one of the subtrees)

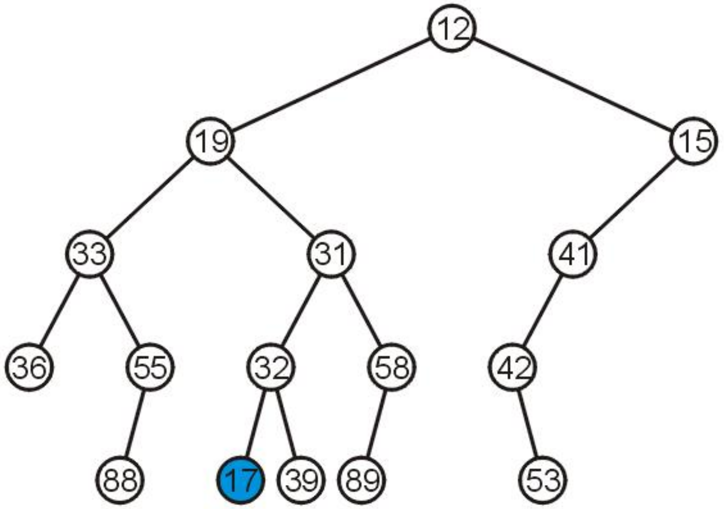
We will use the first approach with binary heaps

- Other heaps use the second

Push

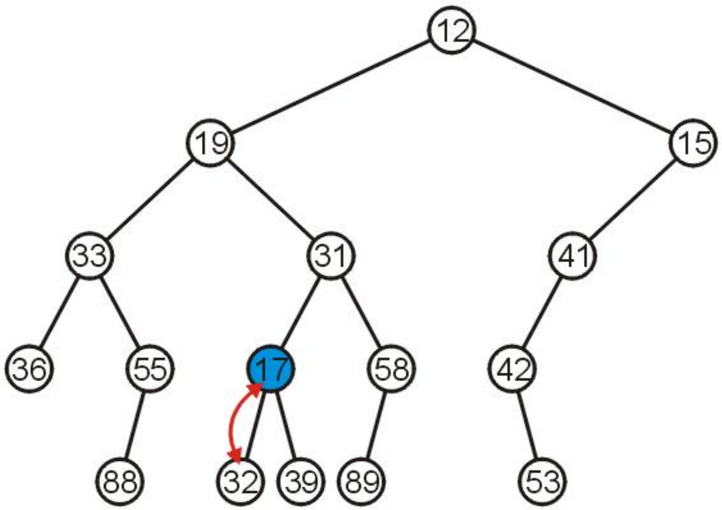
Inserting 17 into the last heap

– Select an arbitrary node to insert a new leaf node:



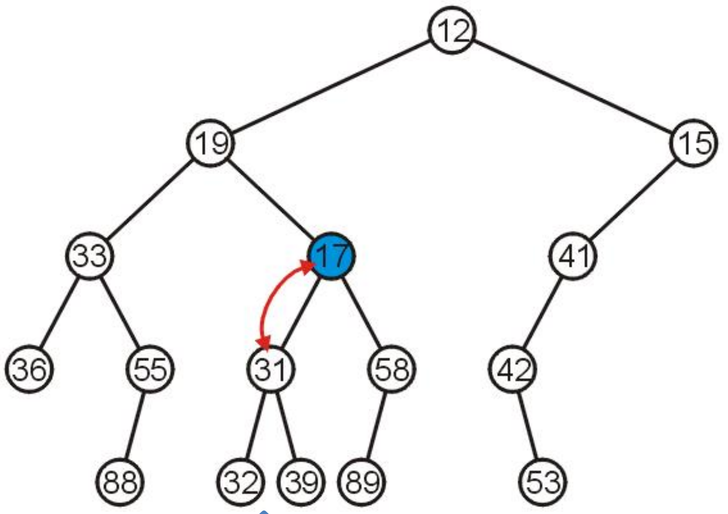
Push

The node 17 is less than the node 32, so we swap them



Push

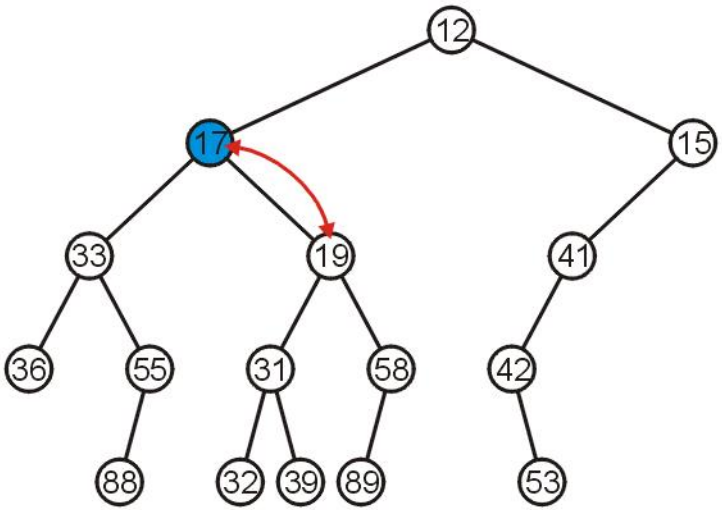
The node 17 is less than the node 31; swap them



31 is smaller than 32 and 39 because 31 was the ancestor of 32 and 39

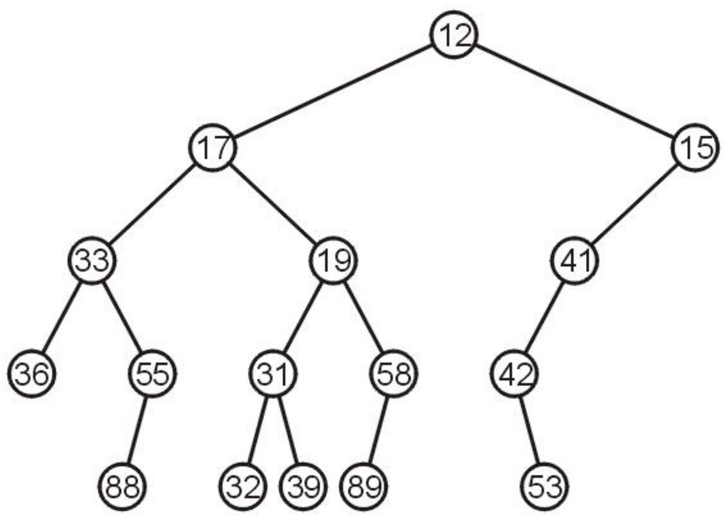
Push

The node 17 is less than the node 19; swap them



Push

The node 17 is greater than 12 so we are finished



Push

This process is called *percolation*, that is, the lighter (smaller) objects move up from the bottom of the min-heap

Implementations

With binary search trees, we introduced the concept of *balance*

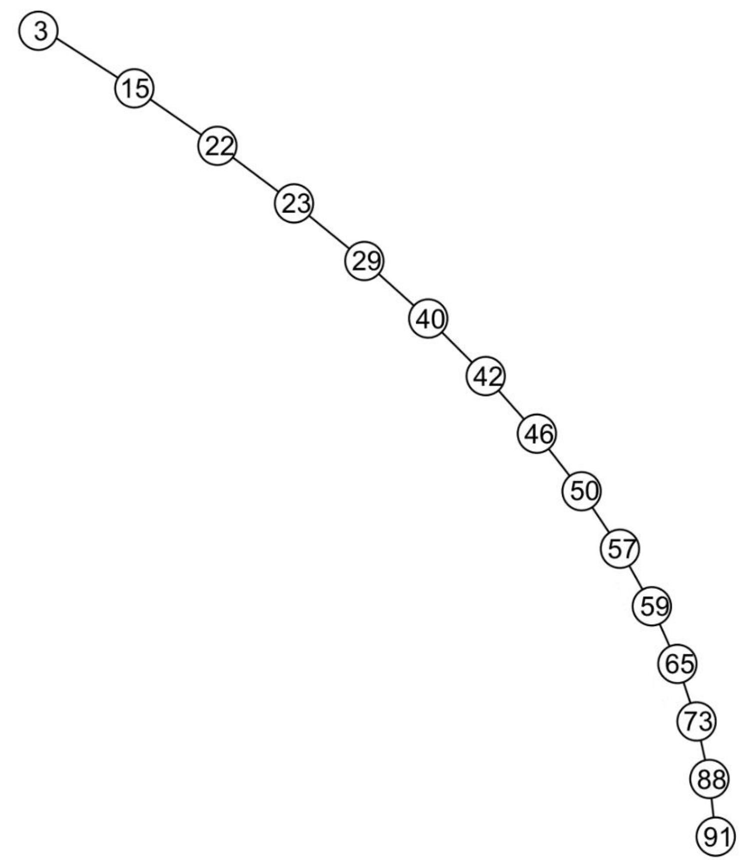
From this, we looked at:

- AVL Trees
- B-Trees
- Red-black Trees (not course material)

How can we determine where to insert so as to keep balance?

Time Complexity

- Time complexity of pop and push?
  - O(*n*)
  - Worst case: the binary tree is highly unbalanced
- Can we do better?
  - Keep balance of the binary tree



Balance

There are multiple means of keeping balance with binary heaps:

- Complete binary trees
- Leftist heaps
- Skew heaps

This defines the actual "binary heap"

We will look at using complete binary trees

- It has optimal memory characteristics but sub-optimal run-time characteristics

Complete Trees

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

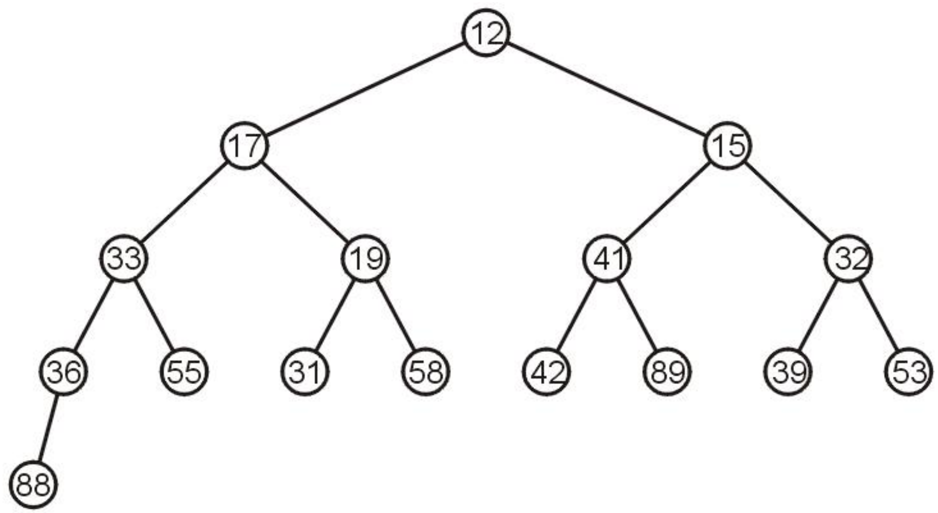
We have already seen

– 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!

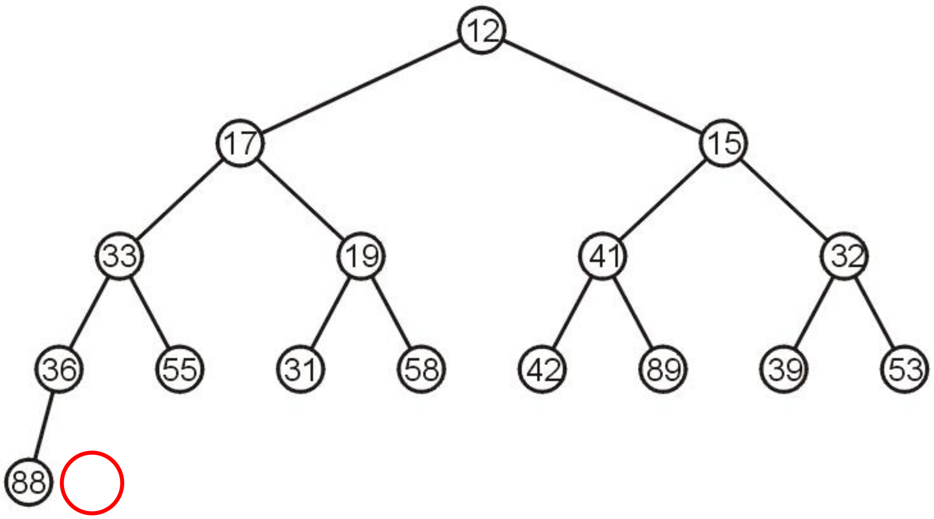
Complete Trees

For example, the previous heap may be represented as the following (non-unique!) complete tree:



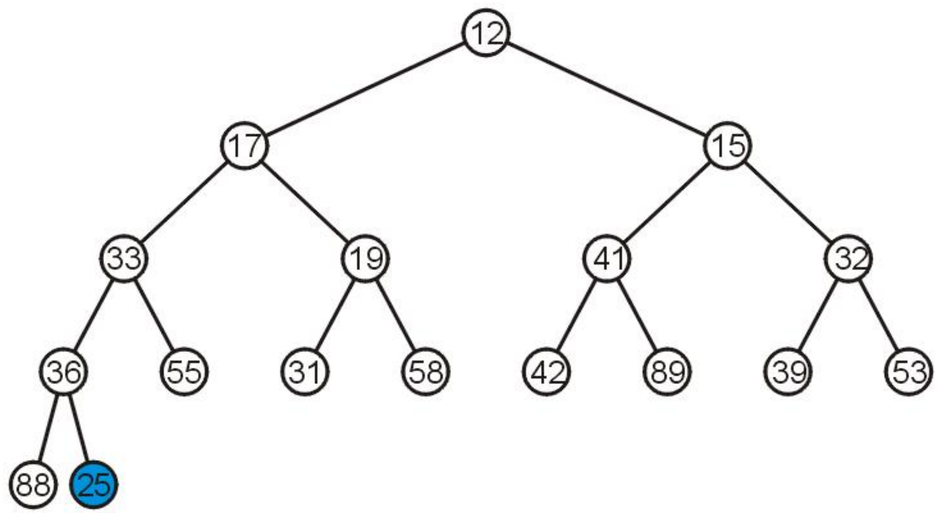
Complete Trees: Push

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



Complete Trees: Push

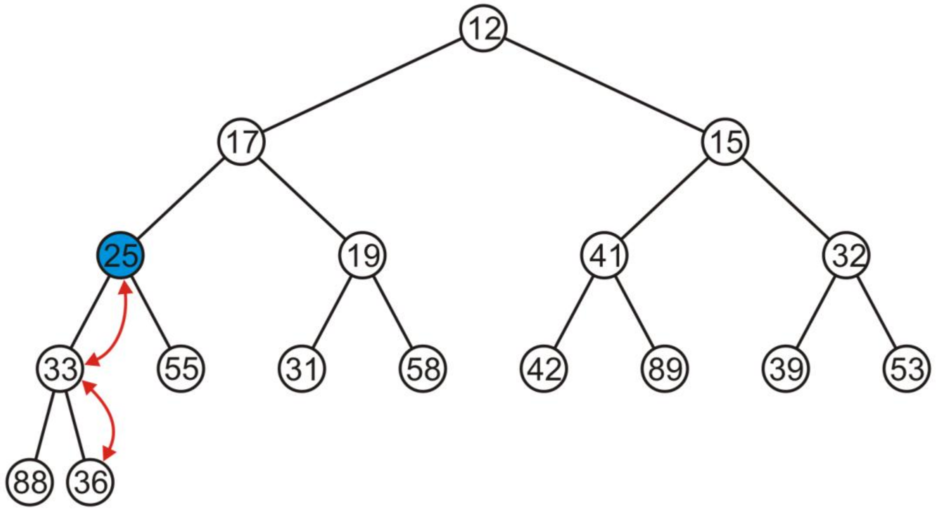
For example, push 25:



Complete Trees: Push

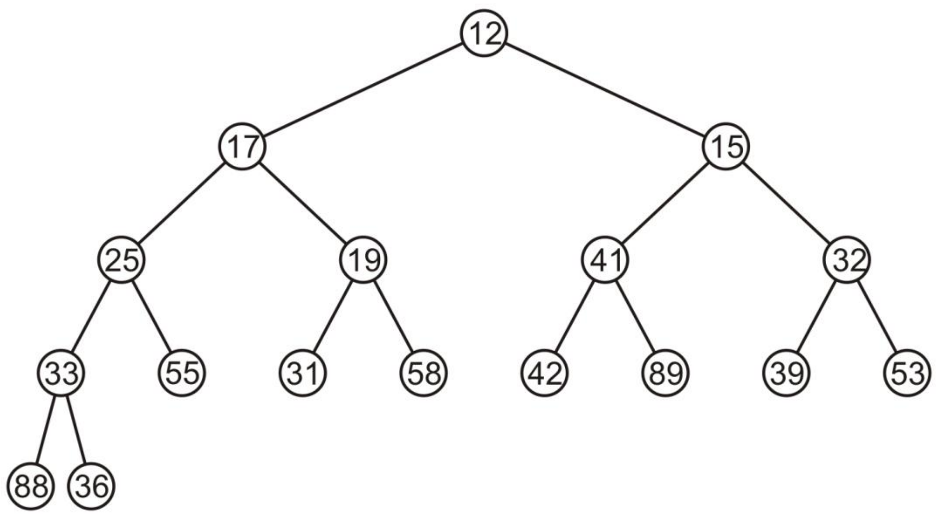
We have to percolate 25 up into its appropriate location

- The resulting heap is still a complete tree



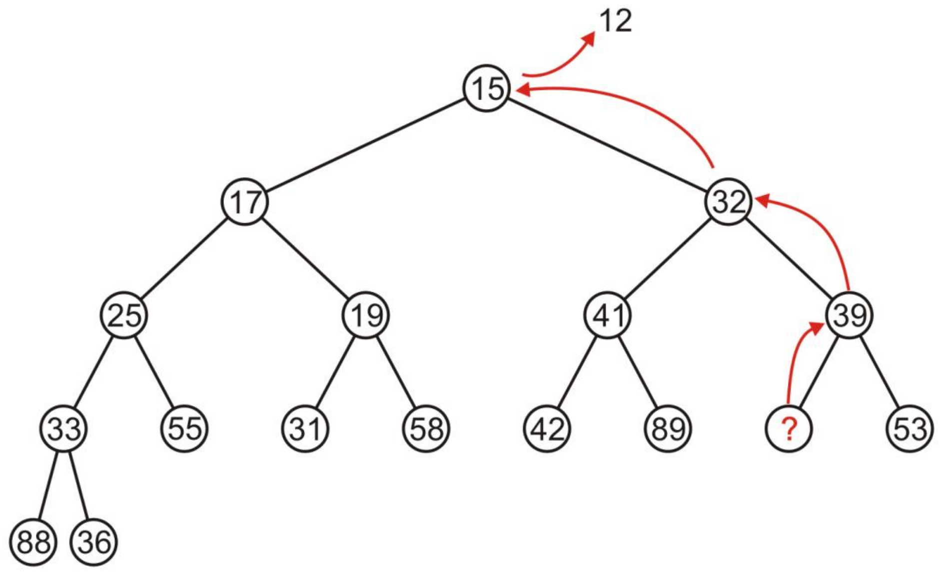
Complete Trees: Pop

Suppose we want to pop the top entry: 12



Complete Trees: Pop

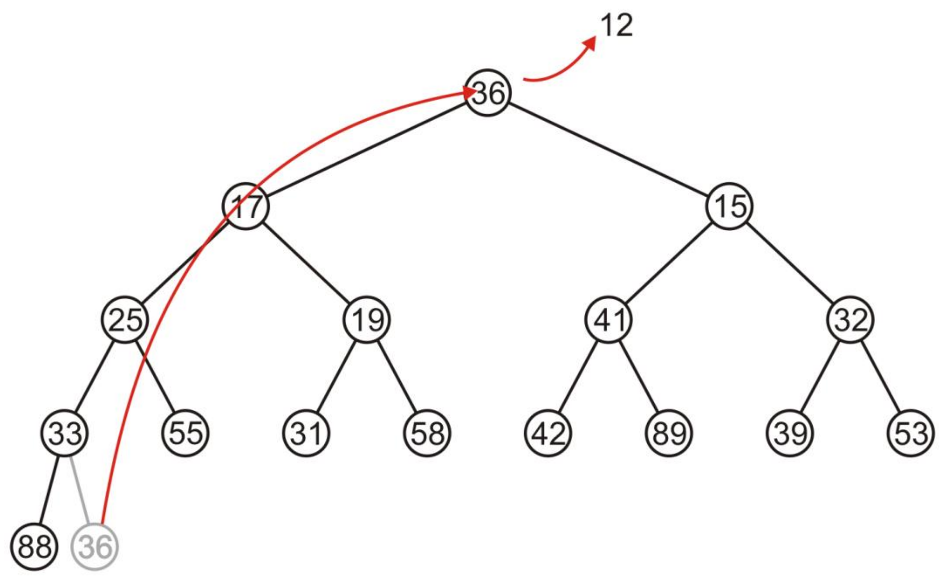
Percolating up creates a hole leading to a non-complete tree



What's wrong?

Complete Trees: Pop

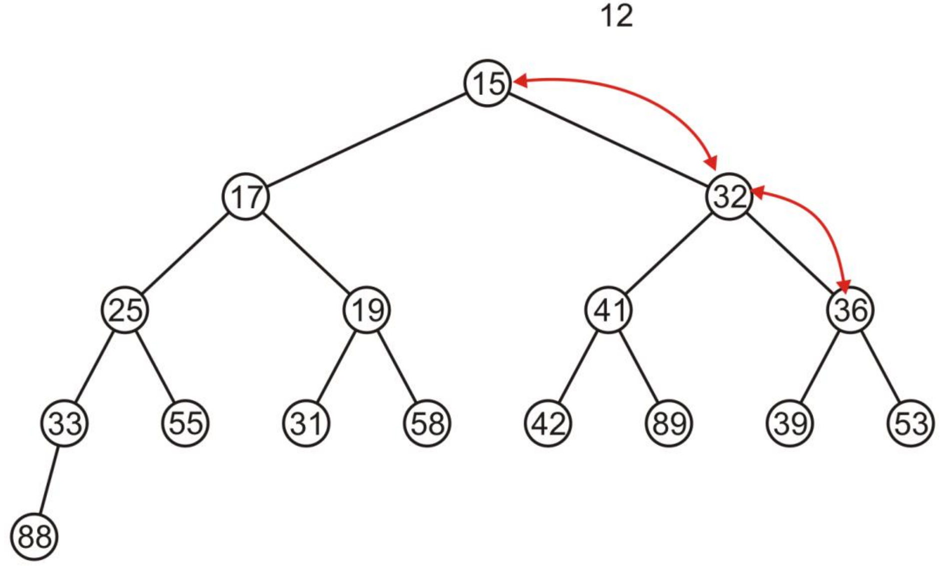
Instead, copy the last entry in the heap to the root



Complete Trees: Pop

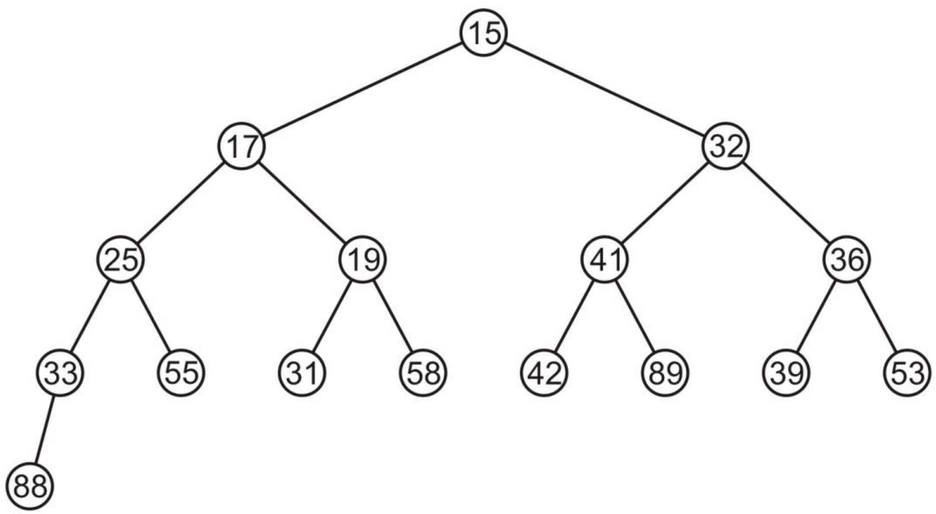
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger



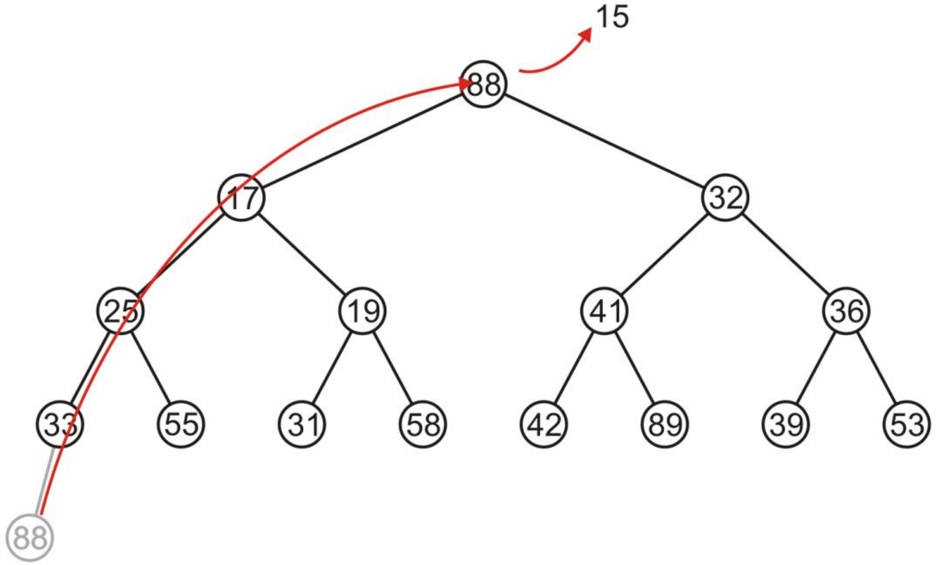
Complete Trees: Pop

The resulting tree is now still a complete tree:



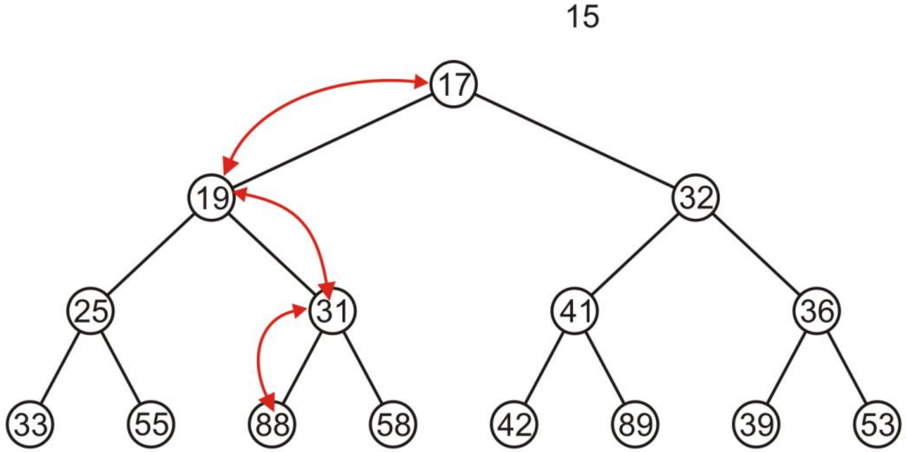
Complete Trees: Pop

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



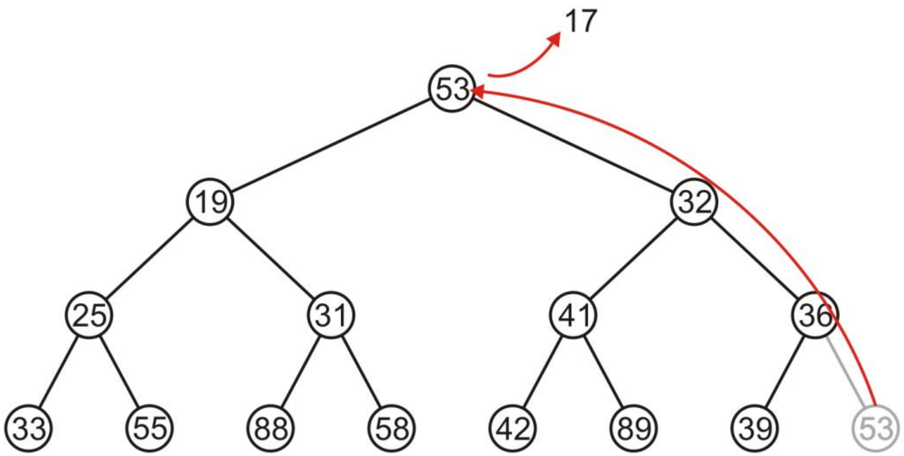
Complete Trees: Pop

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



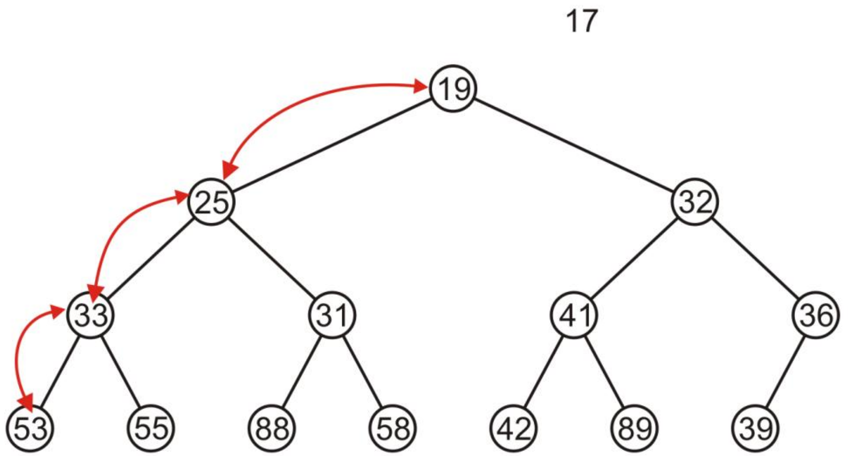
Complete Trees: Pop

In popping 17, 53 is moved to the top



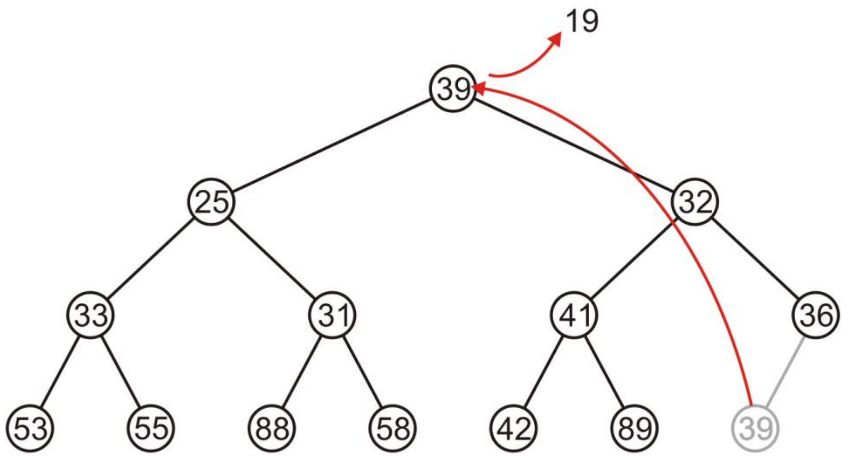
Complete Trees: Pop

And percolated down, again to the deepest level



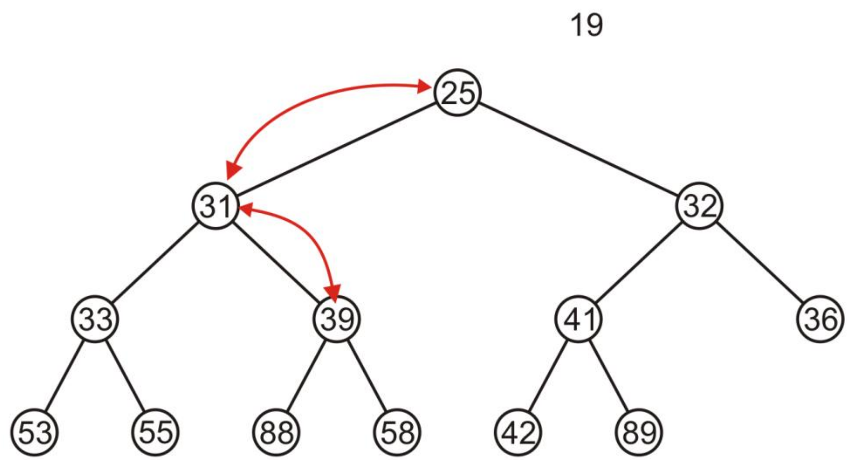
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees: Pop

Which is then percolated down to the second deepest level



Complete Tree

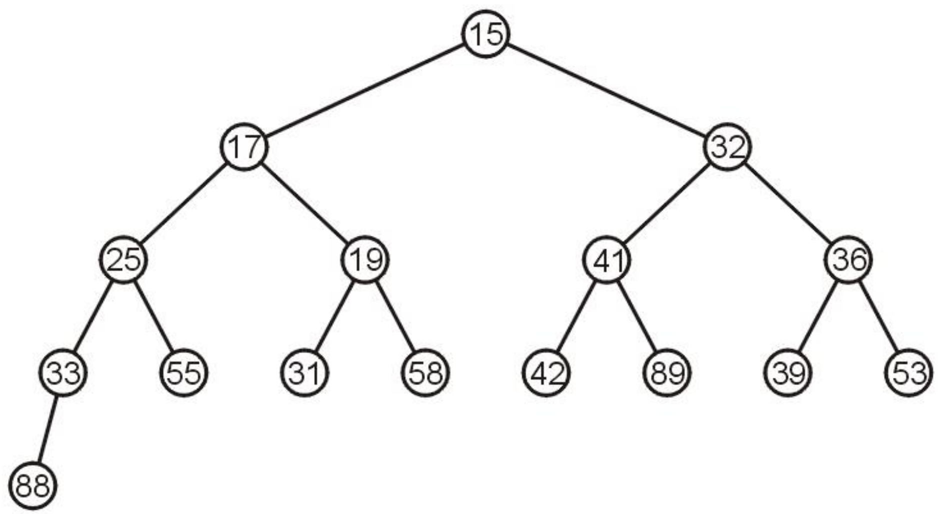
Therefore, we can maintain the complete-tree shape of a heap

We may store a complete tree using an array:

- The array is filled using breadth-first traversal on the tree

Array Implementation

For the heap

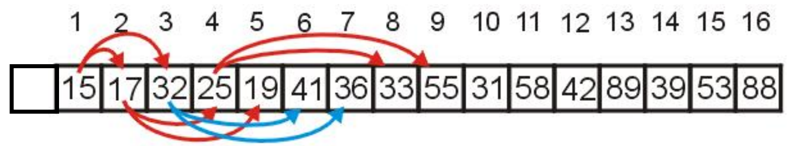


a breadth-first traversal yields:

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

Array Implementation

We start at index 1 when filling the array.

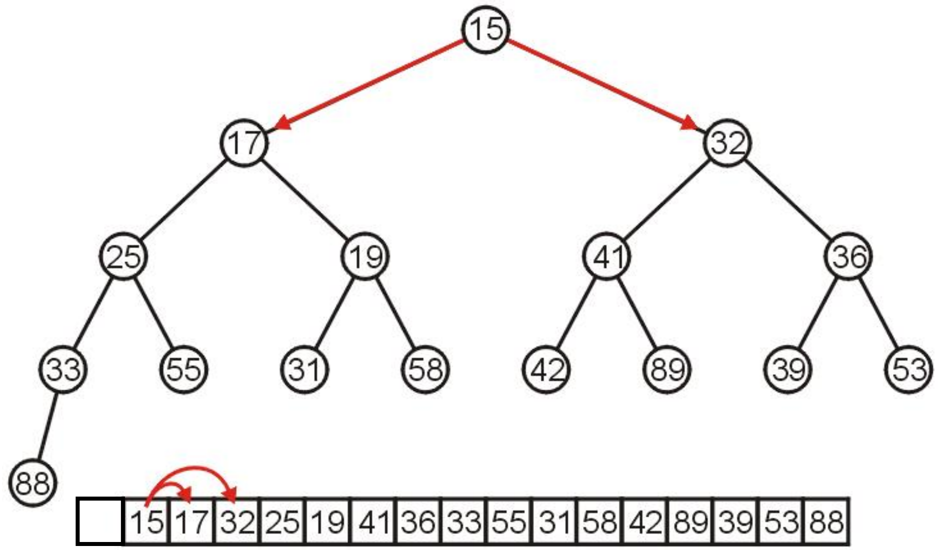


Given the entry at index k, it follows that:

- The parent of node is a $k/2$ — parent = k >> 1;
- the children are at $2k$ and $2k + 1$ — left_child = k << 1; right_child = left_child | 1;

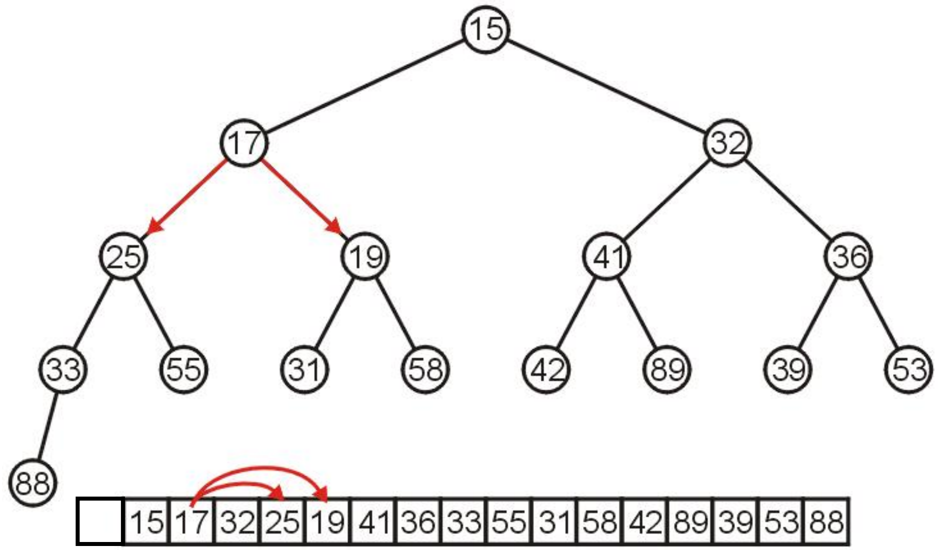
Array Implementation

The children of 15 are 17 and 32:



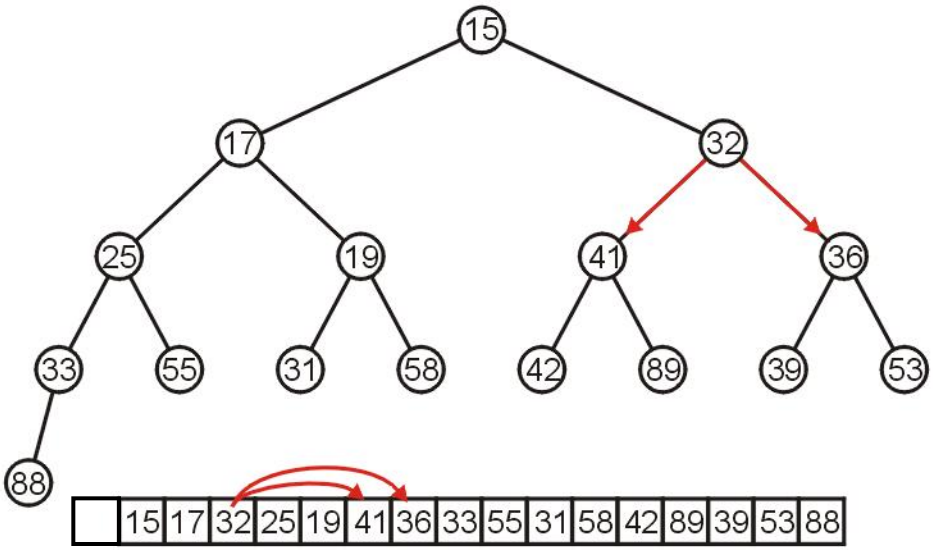
Array Implementation

The children of 17 are 25 and 19:



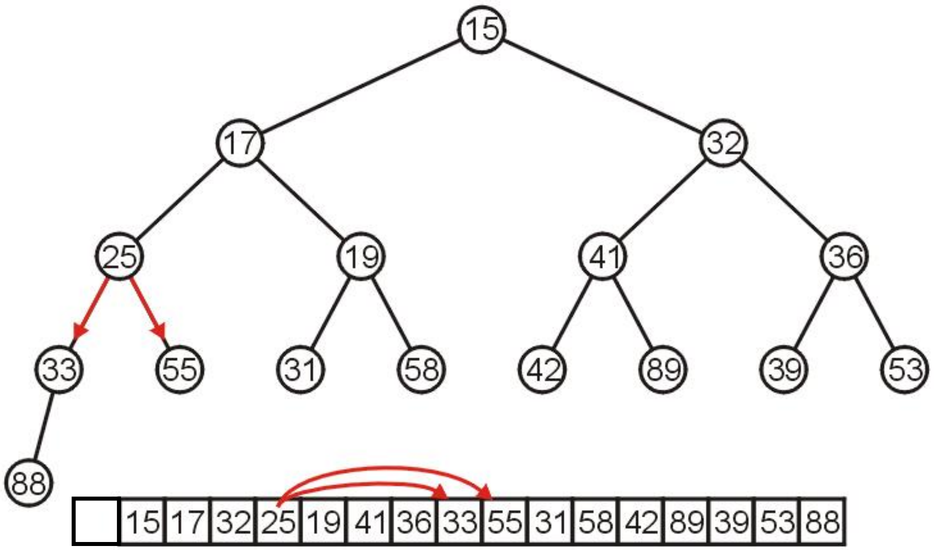
Array Implementation

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 **N** entries, then the next empty node in the corresponding complete tree is at location **L = N + 1**

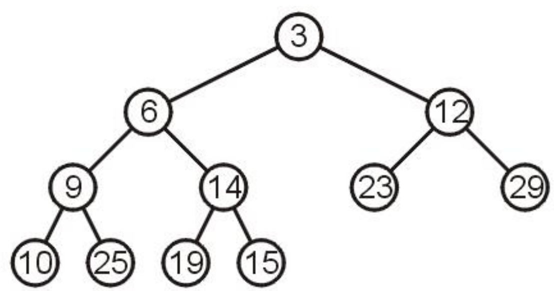
We compare the item at location **L** with the item at **L/2**

If they are out of order

- Swap them
- Set **L/= 2** and repeat

Array Implementation

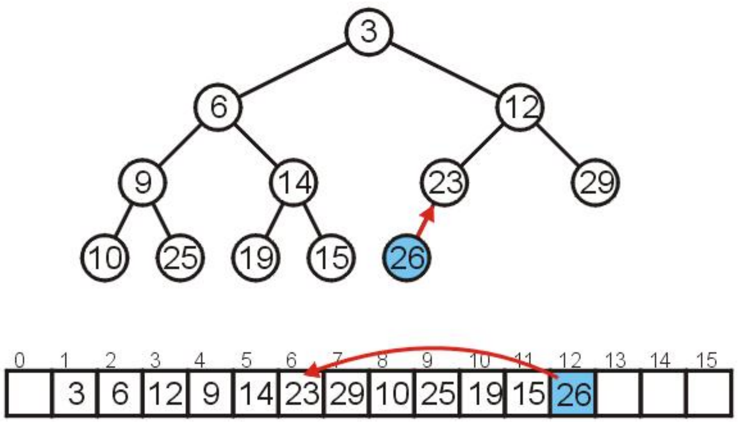
Consider the following heap, both as a tree and in its array representation



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

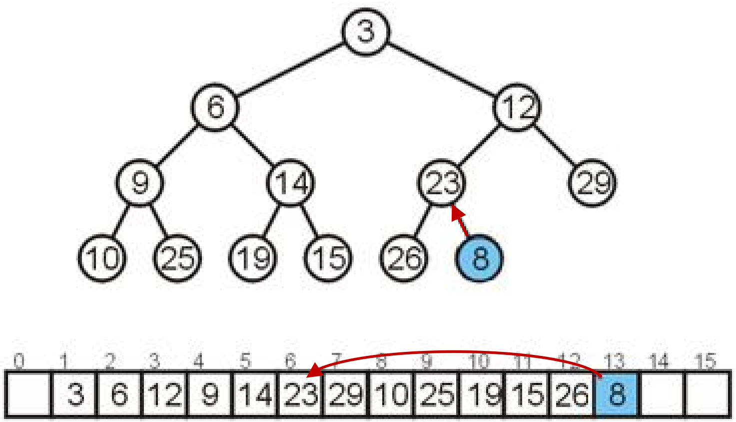
Inserting 26 requires no changes



Array Implementation: Push

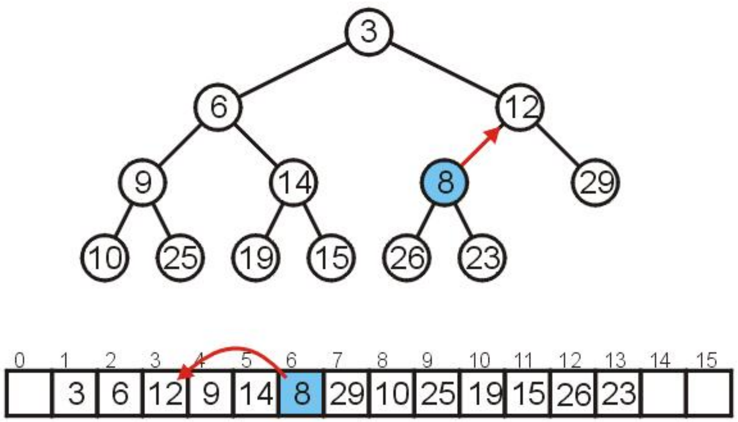
Inserting 8 requires a few percolations:

– Swap 8 and 23



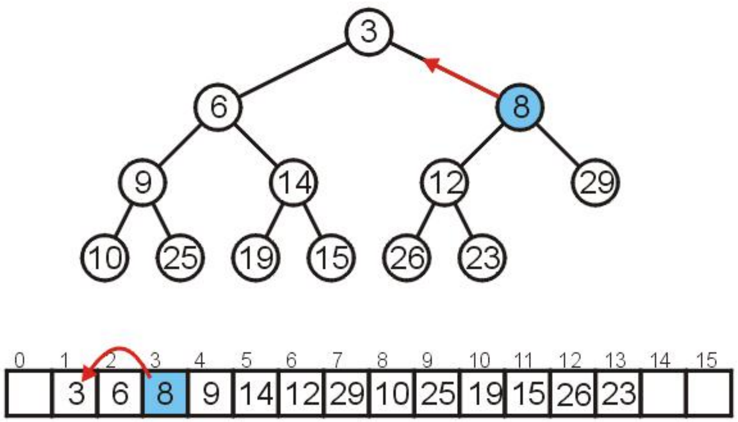
Array Implementation: Push

Swap 8 and 12



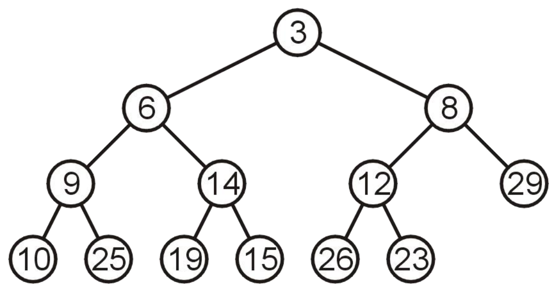
Array Implementation: Push

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



Array Implementation: Pop

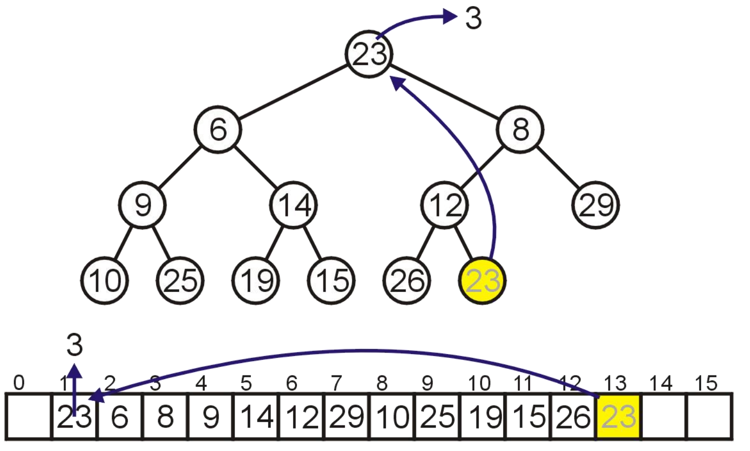
As before, popping the top has us copy the last entry to the top



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

As before, popping the top has us copy the last entry to the top

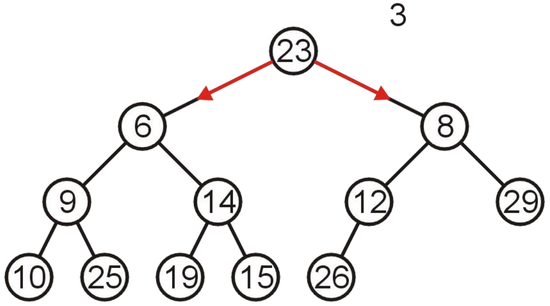


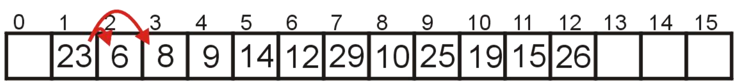
Array Implementation: Pop

Now percolate down

Compare Node 1 with its children: Nodes 2 and 3

– Swap 23 and 6

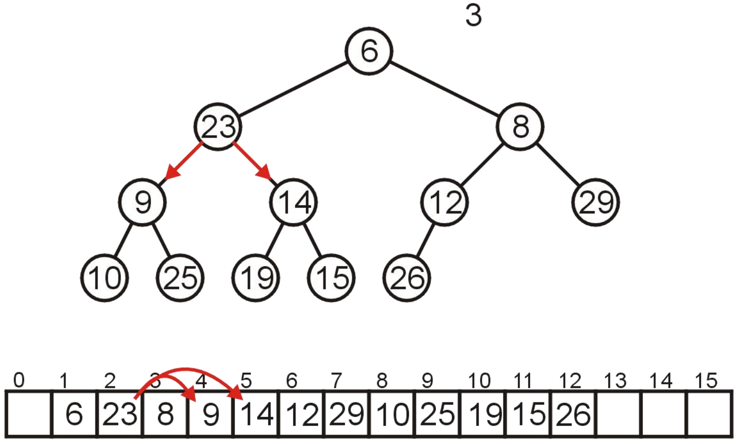




Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

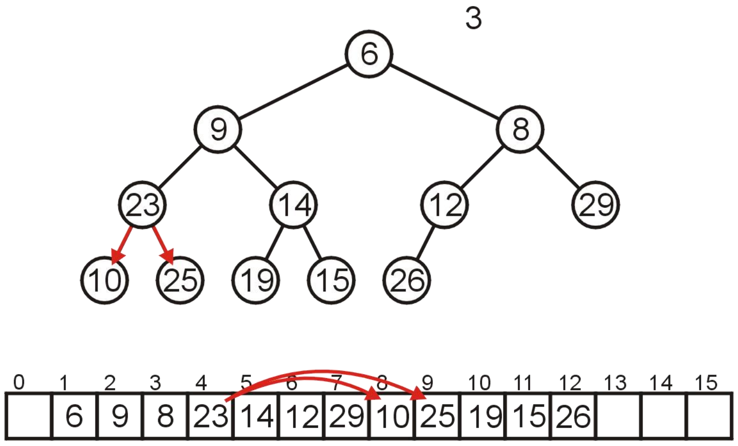
– Swap 23 and 9



Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

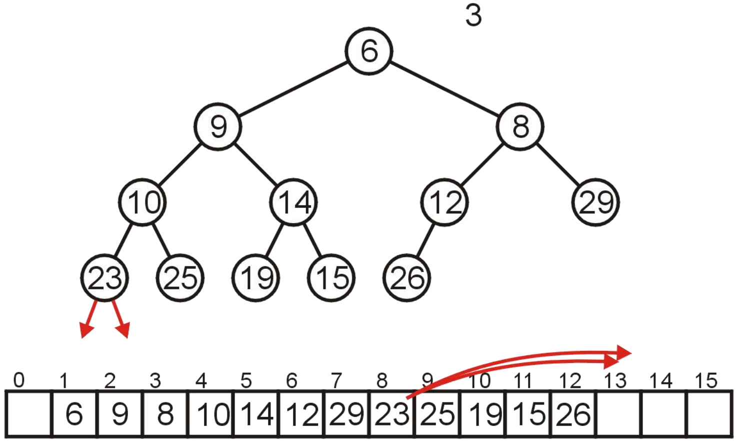
- Swap 23 and 10



Array Implementation: Pop

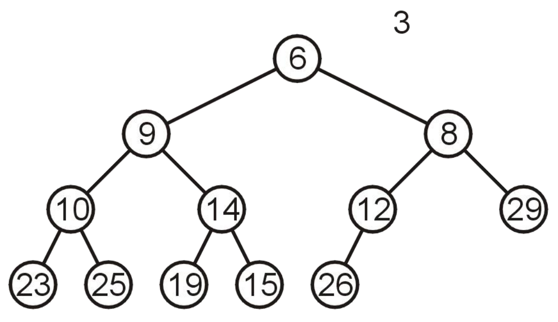
The children of Node 8 are beyond the end of the array:

– Stop



Array Implementation: Pop

The result is a binary min-heap

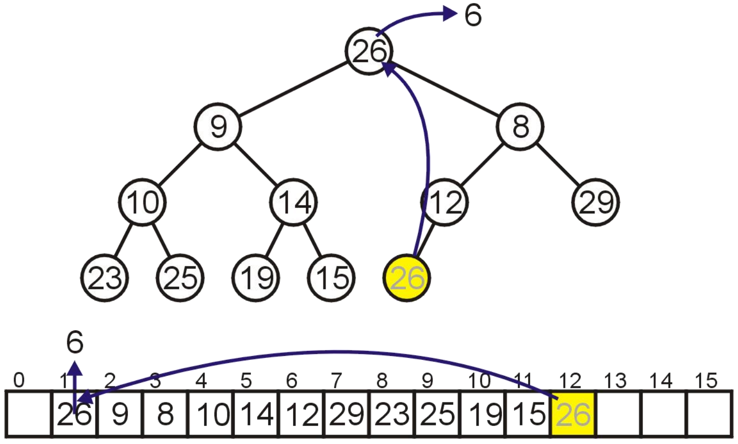


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	6	9	8	10	14	12	29	23	25	19	15	26			

Array Implementation: Pop

Dequeuing the minimum again:

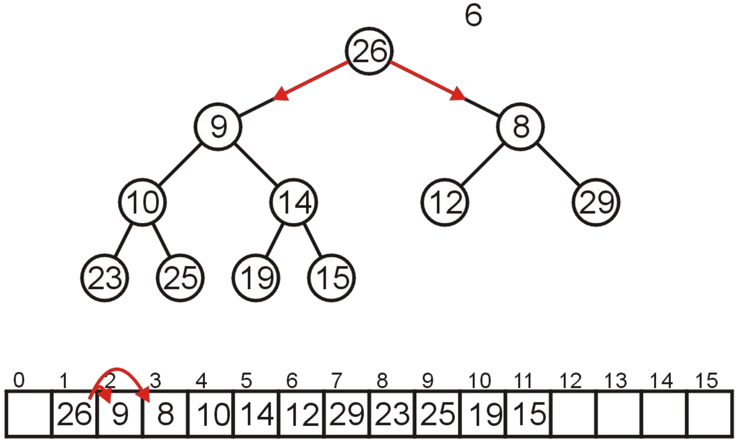
– Copy 26 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

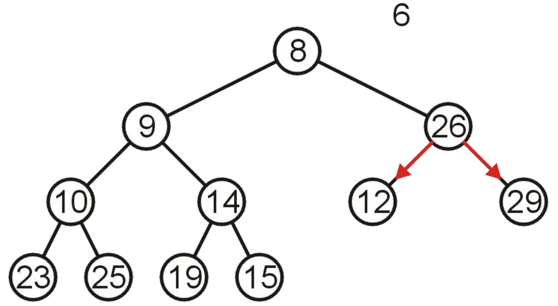
- Swap 26 and 8



Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

- Swap 26 and 12

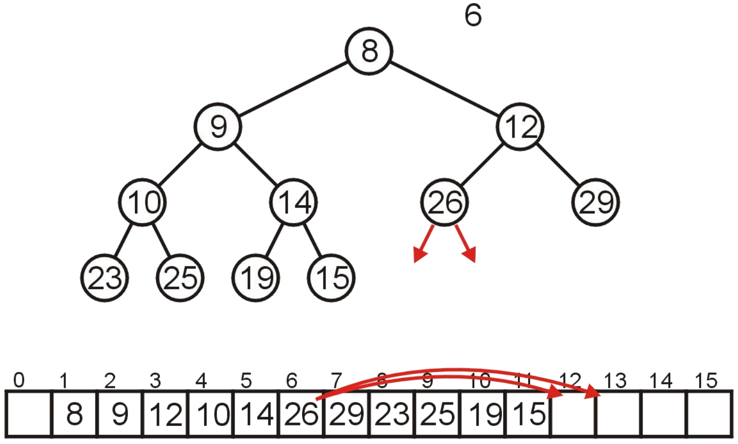


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	8	9	26	10	14	12	29	23	25	19	15				

Array Implementation: Pop

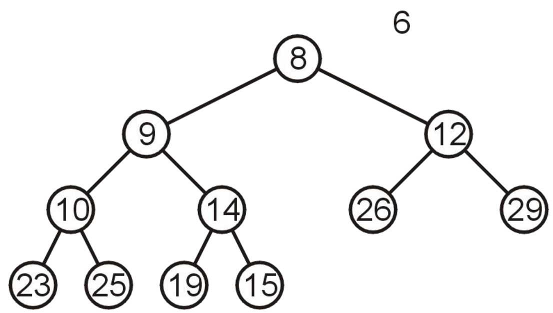
The children of Node 6, Nodes 12 and 13 are unoccupied

- Currently, N == 11



Array Implementation: Pop

The result is a min-heap

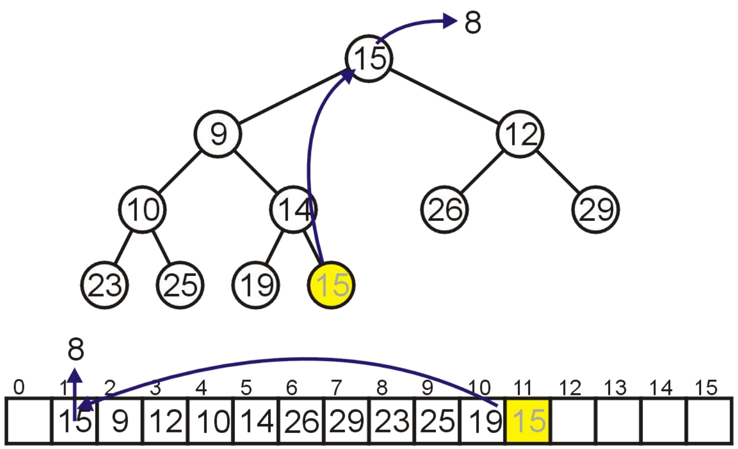


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	8	9	12	10	14	26	29	23	25	19	15				

Array Implementation: Pop

Dequeuing the minimum a third time:

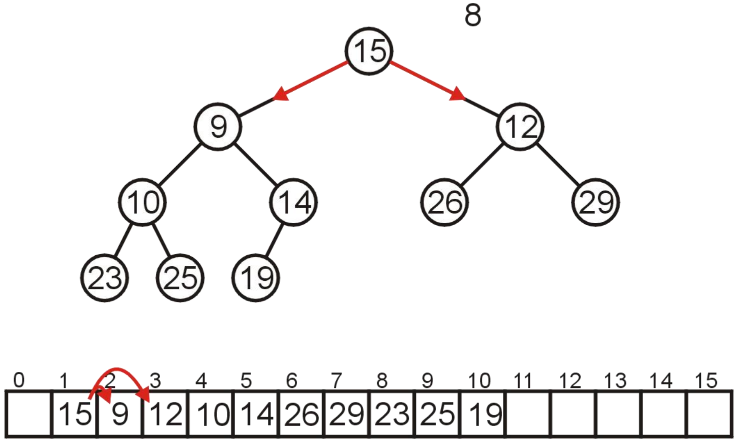
- Copy 15 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

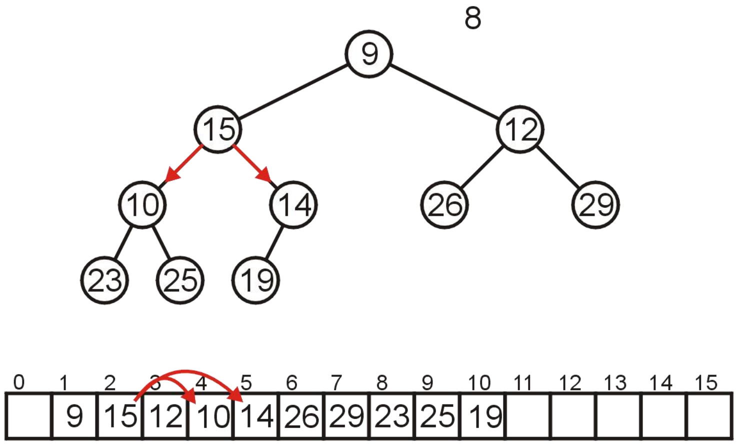
- Swap 15 and 9



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

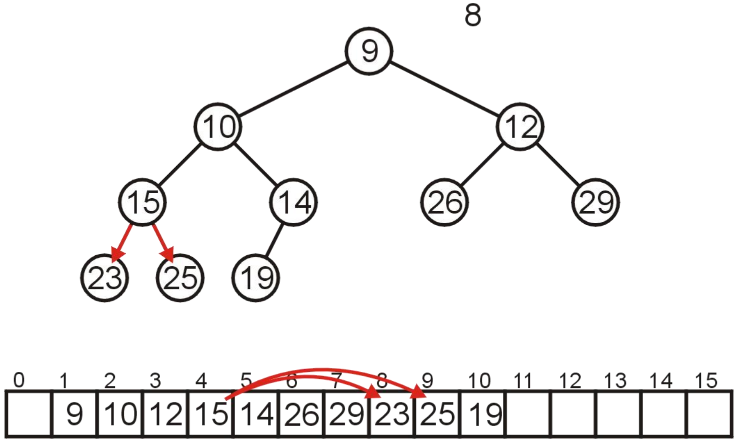
– Swap 15 and 10



Array Implementation: Pop

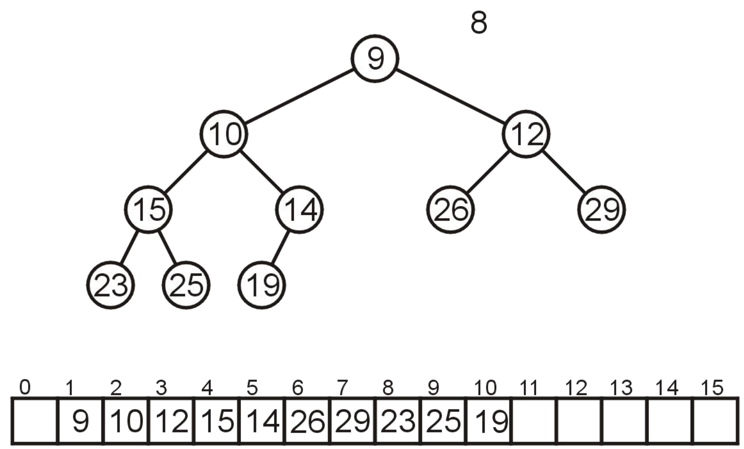
Compare Node 4 with its children: Nodes 8 and 9

- 15 < 23 and 15 < 25 so stop



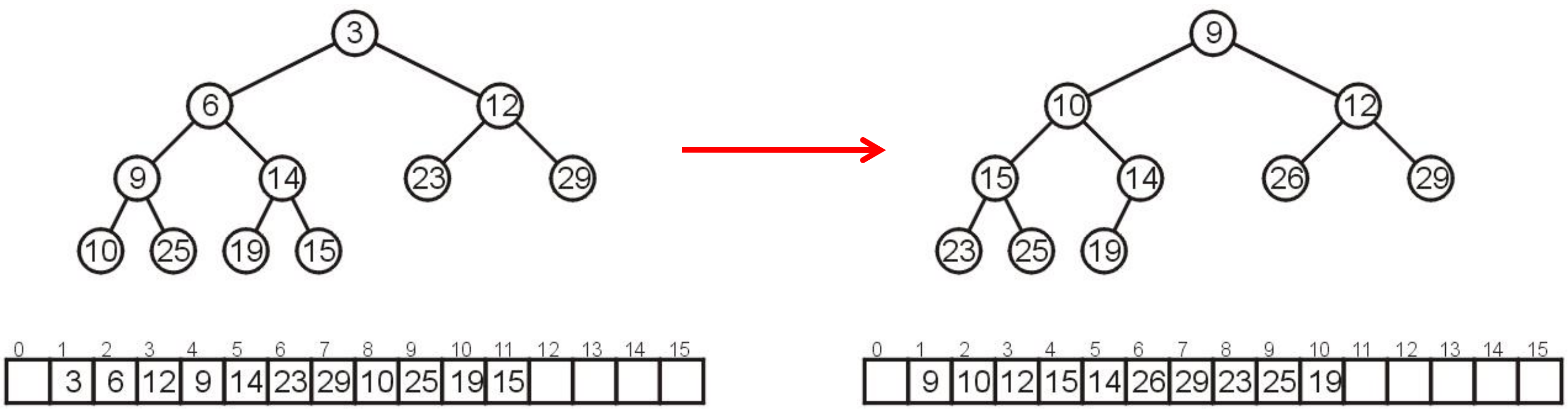
Array Implementation: Pop

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 $\Theta(1)$

Popping the top object is $\mathbf{O}(\ln(n))$

- We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth

Pushing an object is also $\mathbf{O}(\ln(n))$

- If we insert an object less than the root, it will be moved up to the top

Space complexity $\mathbf{O}(n)$

So binary heap is a better implementation of priority queue

Run-time Analysis

If we are inserting an object less than the root (at the front), then the run time will be $\Theta(\ln(n))$

If we insert at the back (greater than any object) then the run time will be $\Theta(1)$

How about an arbitrary insertion?

– It will be $\mathbf{O}(\ln(n))$? Could the average be less?

Run-time Analysis

With each percolation, it will move an object past half of the remaining entries in the tree

- Therefore, after one percolation, it will probably be past half of the entries, and therefore *on average* will require no more percolations

$$\frac{1}{n}\sum_{k=0}^{h}(h-k)2^{k} = \frac{2^{h+1}-h-2}{n}$$

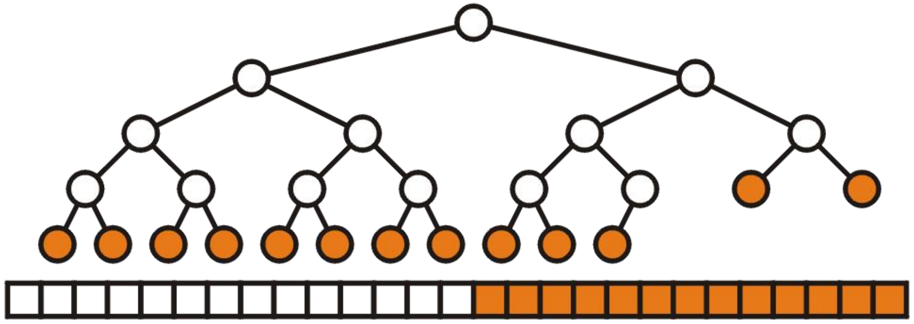
$$= \frac{n-h-1}{n} = \mathbf{\Theta}(1)$$

Therefore, we have an average run time of $\Theta(1)$

Run-time Analysis

An arbitrary removal requires that all entries in the heap be checked: $\mathbf{O}(n)$

A removal of the largest object in the heap still requires all leaf nodes to be checked – there are approximately $n/2$ leaf nodes: $\mathbf{O}(n)$



Run-time Analysis

Thus, our grid of run times is given by:

	front	arbitrary	back
insert	$\mathbf{O}(\ln(n))^*$	$\mathbf{O}(1)$	$\mathbf{O}(1)$
access	$\mathbf{O}(1)$	$\mathbf{O}(n)$	$\mathbf{O}(n)$
delete	$\mathbf{O}(\ln(n))$	$\mathbf{O}(n)$	$\mathbf{O}(n)$

Run-time Analysis

Some observations:

- Continuously inserting at the front of the heap (*i.e.*, the new object being pushed is less than everything in the heap) causes the run-time to drop to $\mathbf{O}(\ln(n))$
- If the objects are coming in order of priority, use a regular queue with swapping
- Merging two binary heaps of size n is a $\Theta(n)$ operation

Run-time Analysis

Other heaps have better run-time characteristics

- Leftist, skew, binomial and Fibonacci heaps all use a node-based implementation requiring $\Theta(n)$ additional memory
- For Fibonacci heaps, the run-time of all operations (including merging two Fibonacci heaps) except pop are $\Theta(1)$

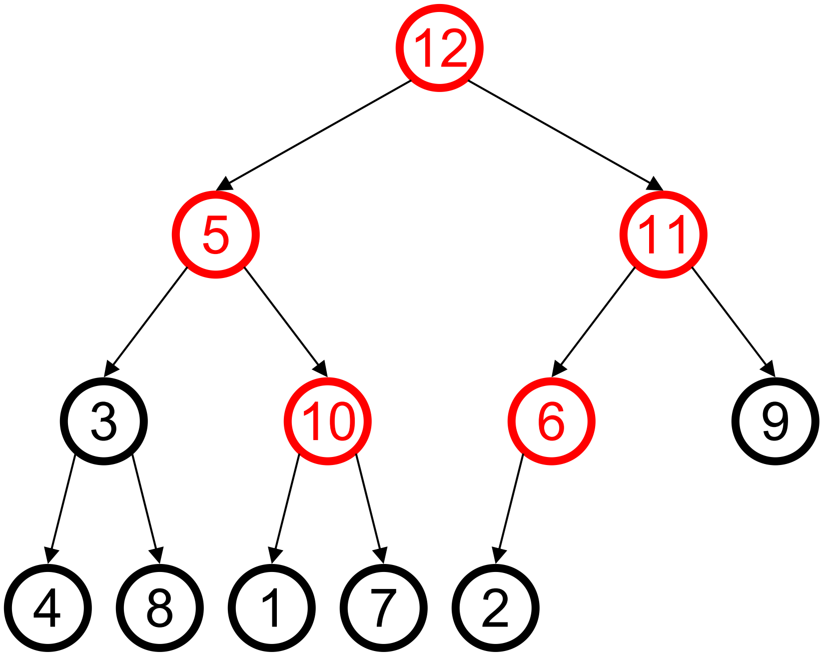
Build Heap

- Task: Given a set of n keys, build a heap all at once
- Approach 1
  - Repeatedly perform push
- Complexity
  - $O(n \ln(n))$

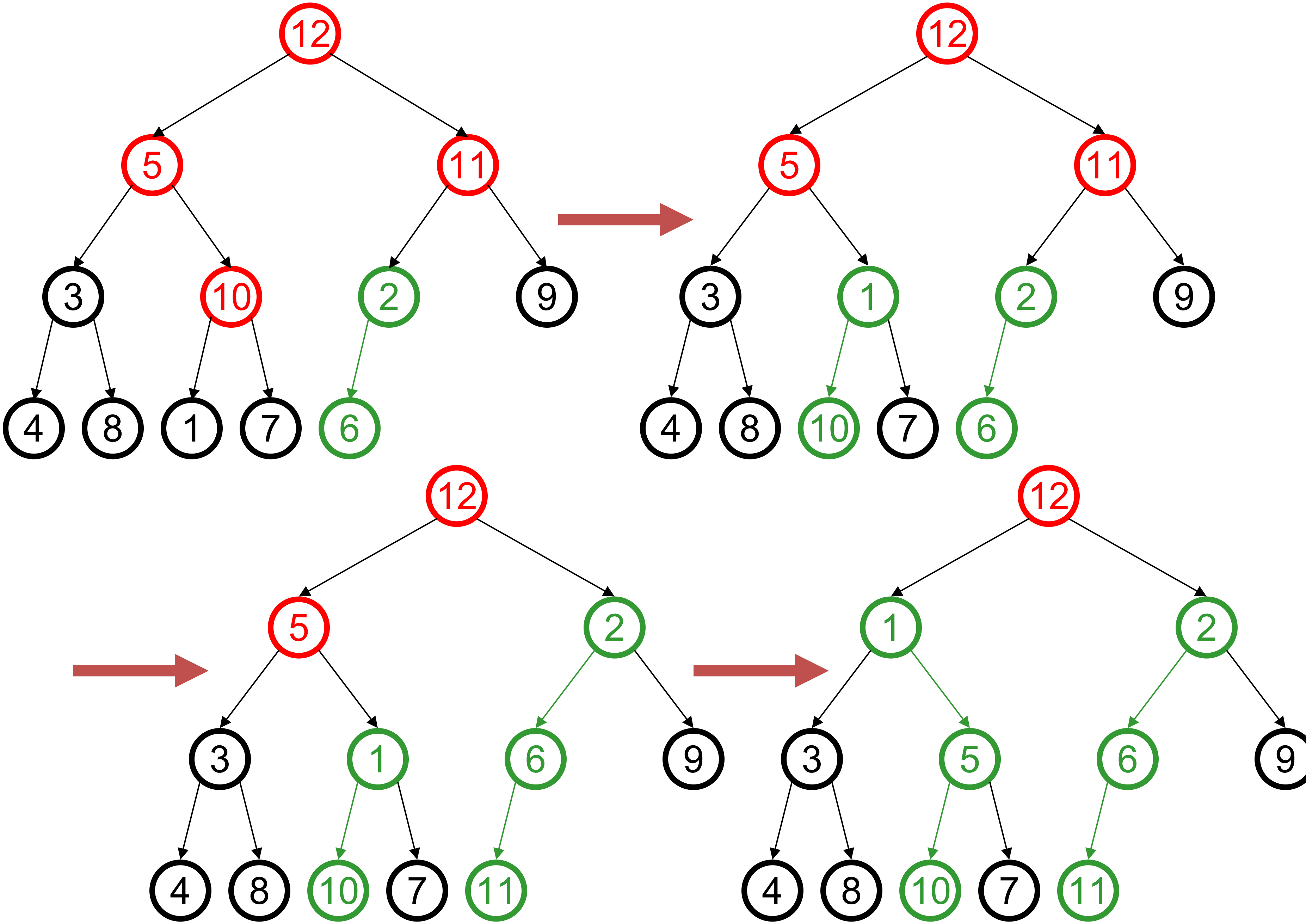
Floyd's Method

Put the keys in a binary tree and fix the heap property!

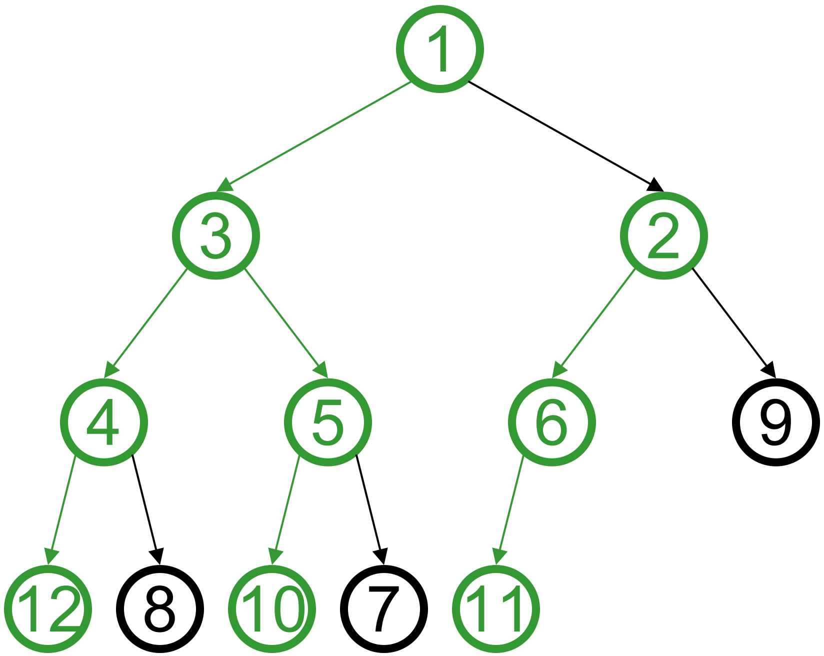
```
buildHeap(){
   for (i=size/2; i>0; i--)
      percolateDown(i);
}
```



12	5	11	3	10	6	9	4	8	1	7	2



Finally…



Complexity of Build Heap

- No percolation for the leaf nodes (n/2 nodes)
- At most n/4 nodes percolate down 1 level
  at most n/8 nodes percolate down 2 levels
  at most n/16 nodes percolate down 3 levels
  …

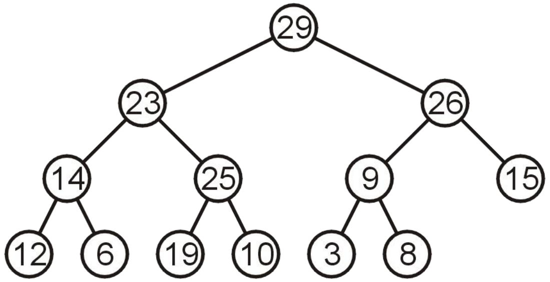
$$1\frac{n}{4}+2\frac{n}{8}+3\frac{n}{16}+\ldots=\sum_{i=1}^{\log n} i\frac{n}{2^{i+2}}=\frac{n}{4}\sum_{i=1}^{\log n}\frac{i}{2^i}=\frac{n}{4}2=\frac{n}{2}$$

$\Theta(n)$

Binary Max Heaps

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		

Outline

- Priority queue
- Binary heap
- Heapsort

Heapsort

- Sorting
  - take a list of objects $(a_0, a_1, ..., a_{n-1})$
  - return a reordering $(a'_0, a'_1, ..., a'_{n-1})$ such that $a'_0 \leq a'_1 \leq \cdot\cdot\cdot \leq a'_{n-1}$

- Heapsort
  - Place the objects into a heap
    - O(*n*) time
  - Repeatedly popping the top object until the heap is empty
    - O(*n* ln(*n*)) time
  - Time complexity: O(*n* ln(*n*))

In-place Implementation

Problem:

- This solution requires additional memory: a min-heap of size n
- This requires $\Theta(n)$ memory

If the unsorted objects are stored in an array, 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

Instead of implementing a min-heap, consider a max-heap:

– The maximum element is at the top of the heap

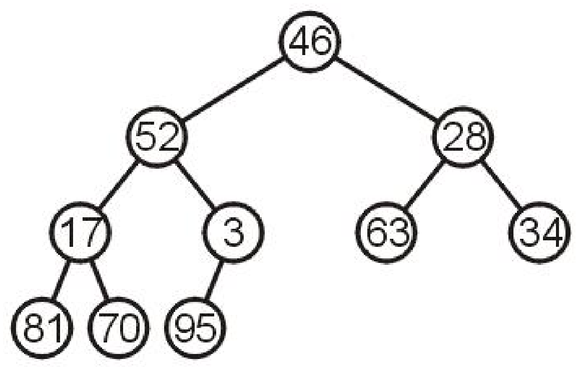
We then repeatedly pop the top object and move it to the end of the array.

In-place Implementation

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

This array represents the following complete tree:

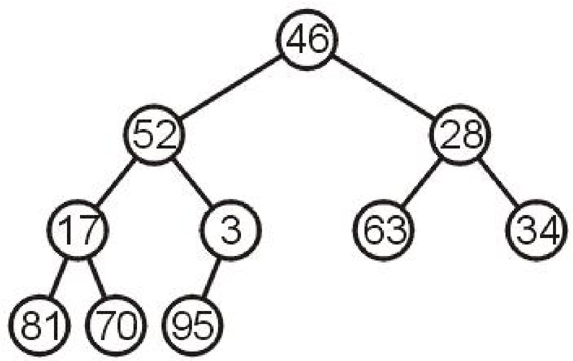


In-place Implementation

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

Because we start at 0 (instead of 1 as in array storage of complete trees), we need different formulas for finding the children and parent



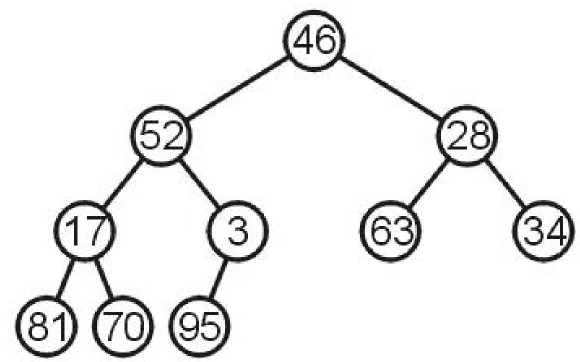
Children	2*k + 1 2*k + 2
Parent	(k + 1)/2 – 1

Example Heap Sort

First, 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 last non-leaf node is in position $n/2-1$



Thus, we start with position $10/2-1 = 4$

Example Heap Sort

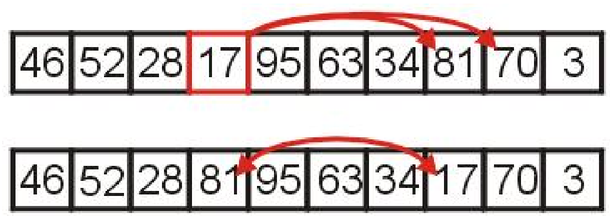
We compare 3 with its child and swap them

46	52	28	17	3	63	34	81	70	95

46	52	28	17	95	63	34	81	70	3

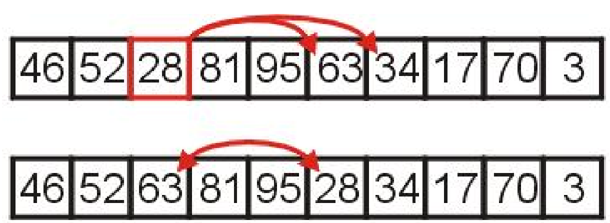
Example Heap Sort

We compare 17 with its two children and swap it with the maximum child (81)



Example Heap Sort

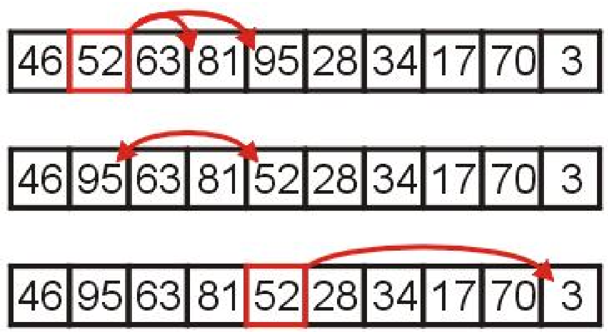
We compare 28 with its two children, 63 and 34, and swap it with the largest child



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

46	95	63	81	52	28	34	17	70	3
95	46	63	81	52	28	34	17	70	3
95	46	63	81	52	28	34	17	70	3
95	81	63	46	52	28	34	17	70	3
95	81	63	46	52	28	34	17	70	3
95	81	63	70	52	28	34	17	46	3

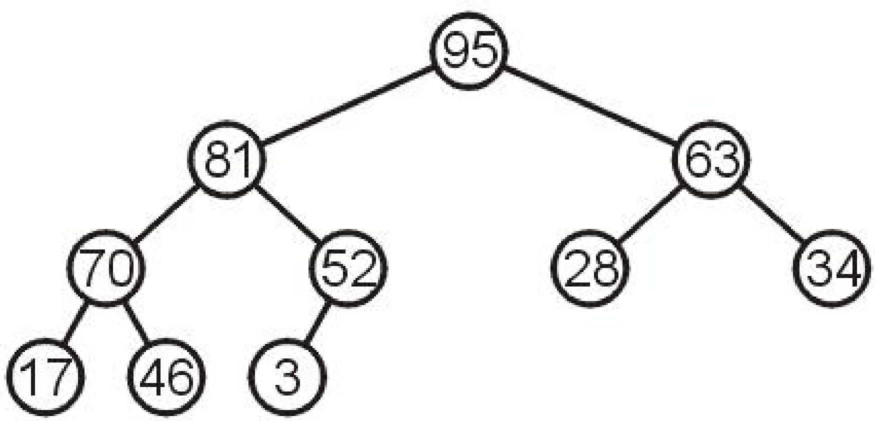
Heap Sort Example

We have now converted the unsorted array

46	52	28	17	3	63	34	81	70	95

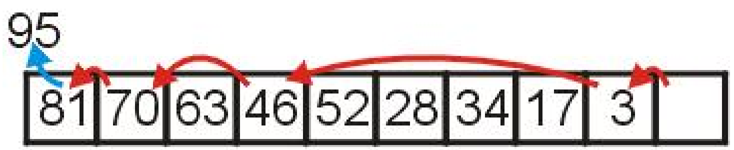
into a max-heap:

95	81	63	70	52	28	34	17	46	3

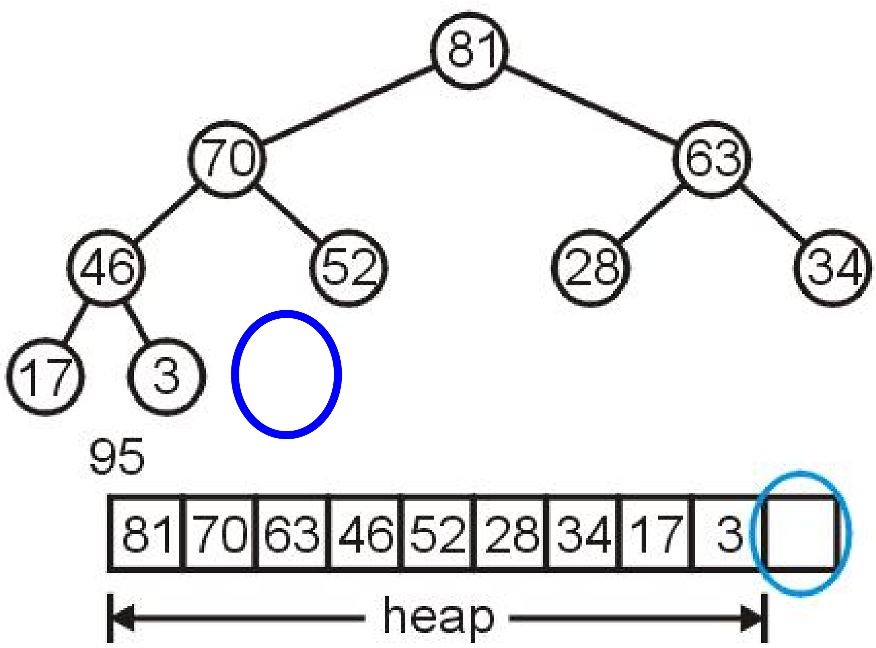


Heap Sort Example

We pop the maximum element of this heap



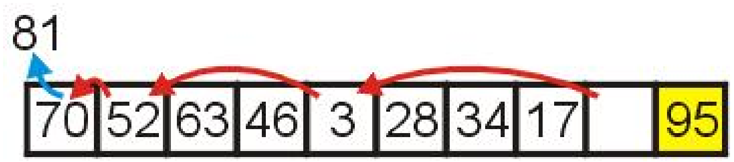
This leaves a gap at the back of the array:

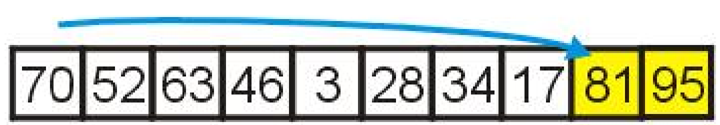


Heap Sort Example

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:

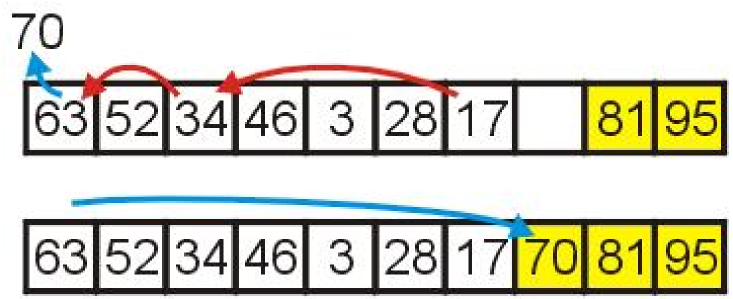




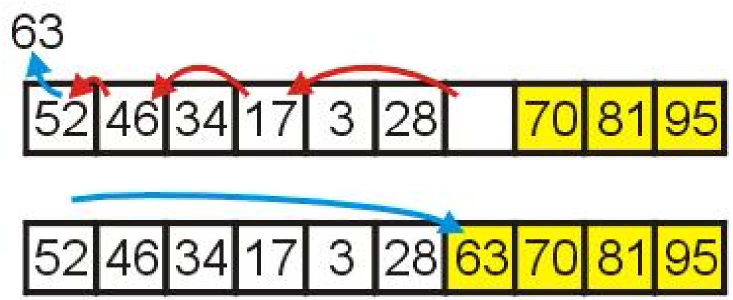
Heap Sort Example

Repeat this process

- Pop and append 70



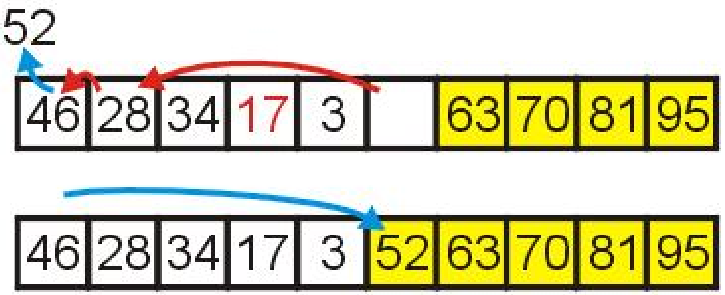
- Pop and append 63



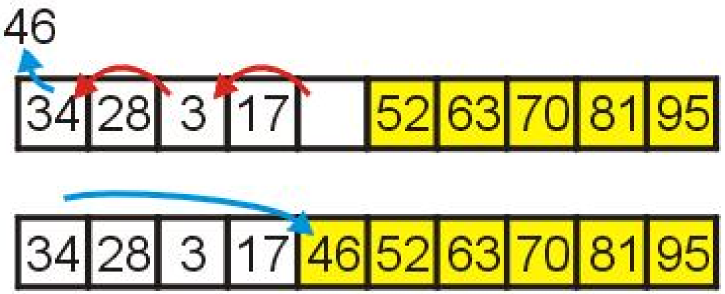
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



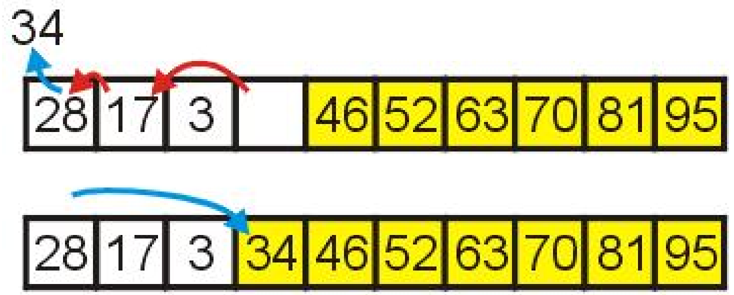
- Pop and append 46



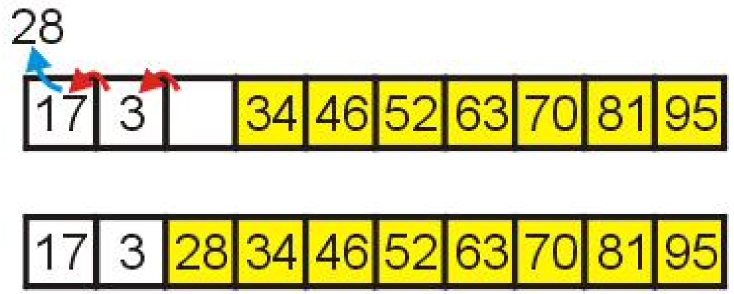
Heap Sort Example

Continuing...

– Pop and append 34

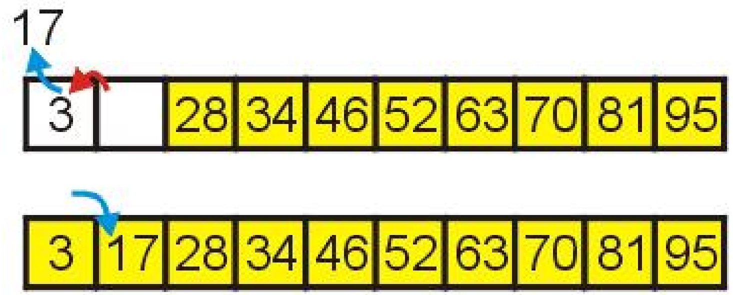


– Pop and append 28



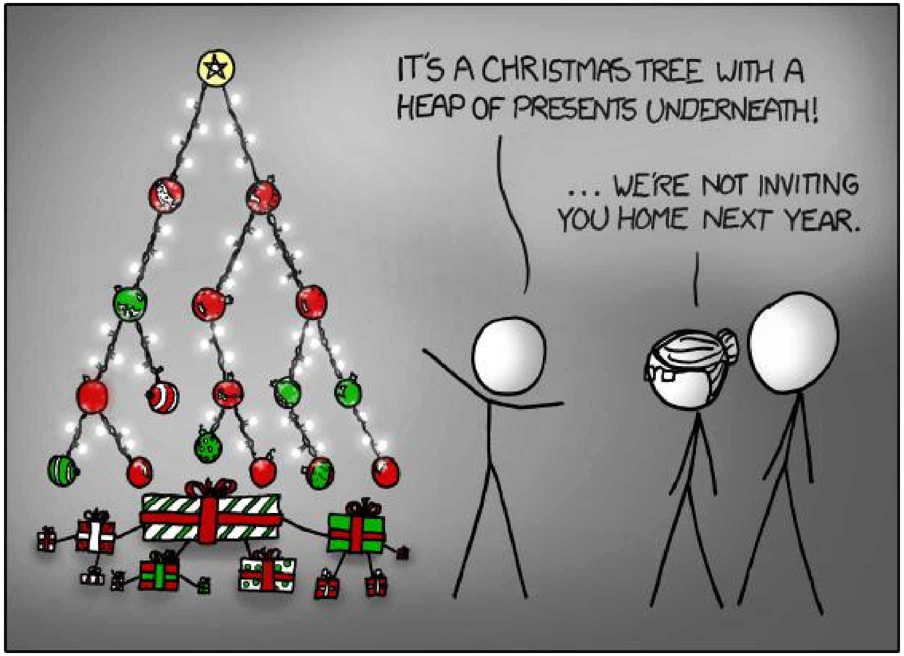
Heap Sort Example

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



Example

Here we have a max-heap of presents under a red-green tree:



http://xkcd.com/835/

Priority Queues

Now, does using a heap ensure that that object in the heap which:

- has the highest priority, and
- of that highest priority, has been in the heap the longest

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

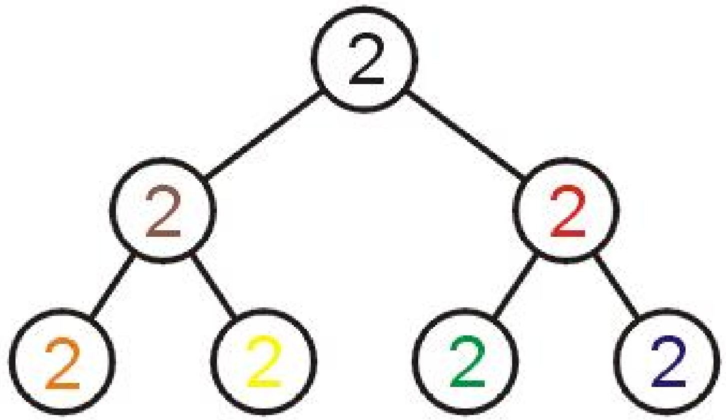
2, 2, 2, 2, 2, 2, 2

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:

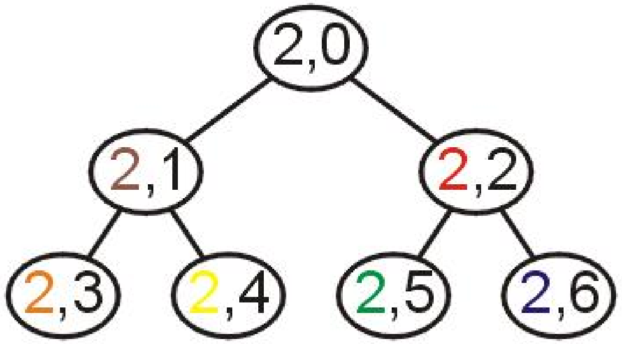
- Come up with an algorithm which removes all seven objects in the original order

Lexicographical Ordering

A better solution is to modify the priority:

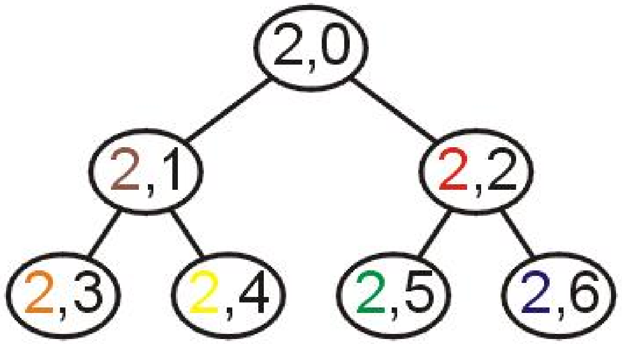
- Track the number of insertions with a counter k (initially 0)
- For each insertion with priority n, create a hybrid priority (n, k) where:

$$(n_1, k_1) < (n_2, k_2) \text{ if } n_1 < n_2 \text{ or } (n_1 = n_2 \text{ and } k_1 < k_2)$$



Priority Queues

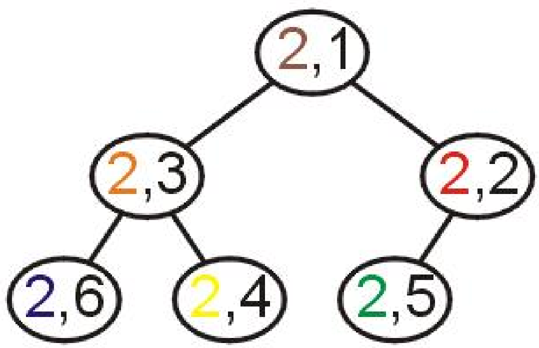
Removing the objects would be in the following order:



Priority Queues

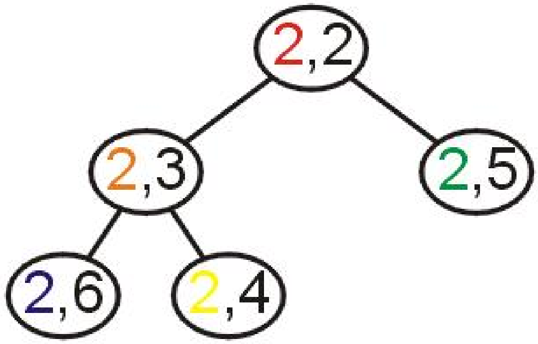
Popped: 2

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



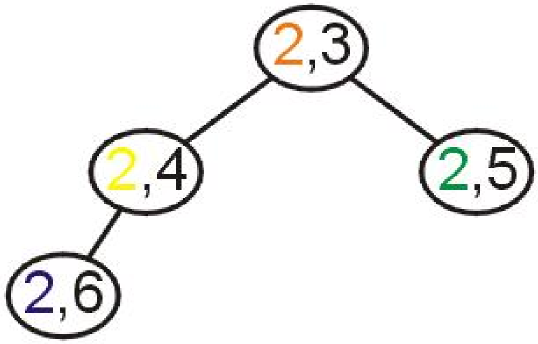
Priority Queues

Removing the objects would be in the following order:



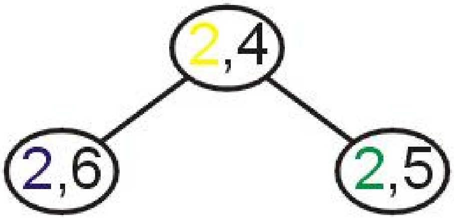
Priority Queues

Removing the objects would be in the following order:



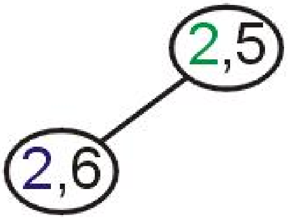
Priority Queues

Removing the objects would be in the following order:



Priority Queues

Removing the objects would be in the following order:



Summary

In this talk, we have:

- Discussed binary heaps
- Looked at an implementation using arrays
- Analyzed the run time:
  - Head $\Theta(1)$
  - Push $\Theta(1)$ average
  - Pop $O(\ln(n))$
- Discussed implementing priority queues using binary heaps
- The use of a lexicographical ordering

Summary

- Priority queue
  - pop the object with the highest priority
- Binary heap
  - Operations
    - Top $\Theta(1)$
    - Push $O(\ln(n))$
    - Pop $O(\ln(n))$
    - Build $\Theta(n)$
  - Implementation using arrays
- Heapsort
  - Time: $O(n \ln(n))$
  - Space: $O(1)$

References

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §7.2.3, p.144.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, §7.1-3, p.140-7.

[3] Weiss, *Data Structures and Algorithm Analysis in C++*, *3rd Ed.*, Addison Wesley, §6.3, p.215-25.

Usage Notes

- These slides are made publicly available on the web for anyone to use
- If you choose to use them, or a part thereof, for a course at another institution, I ask only three things:
  - that you inform me that you are using the slides,
  - that you acknowledge my work, and
  - that you alert me of any mistakes which I made or changes which you make, and allow me the option of incorporating such changes (with an acknowledgment) in my set of slides

Sincerely,

Douglas Wilhelm Harder, MMath

dwharder@alumni.uwaterloo.ca