Data Structures and Algorithms

Lesson 7: *Use Heaps!*

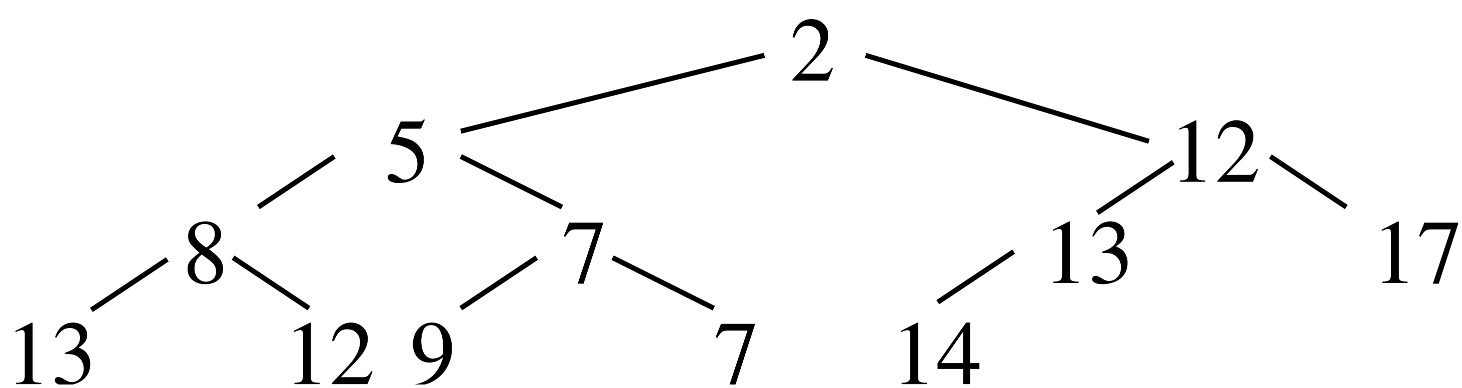
Heaps, Priority queues, Heap Sort

Outline

1. **Heaps**
2. Priority Queues
   - Array Implementation
3. Heap Sort

Heaps

- A (min-)heap is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree: every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left
  - ⋆ Each child has a value 'greater than or equal to' its parent



Adding to the Heap

- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering

Adding to the Heap

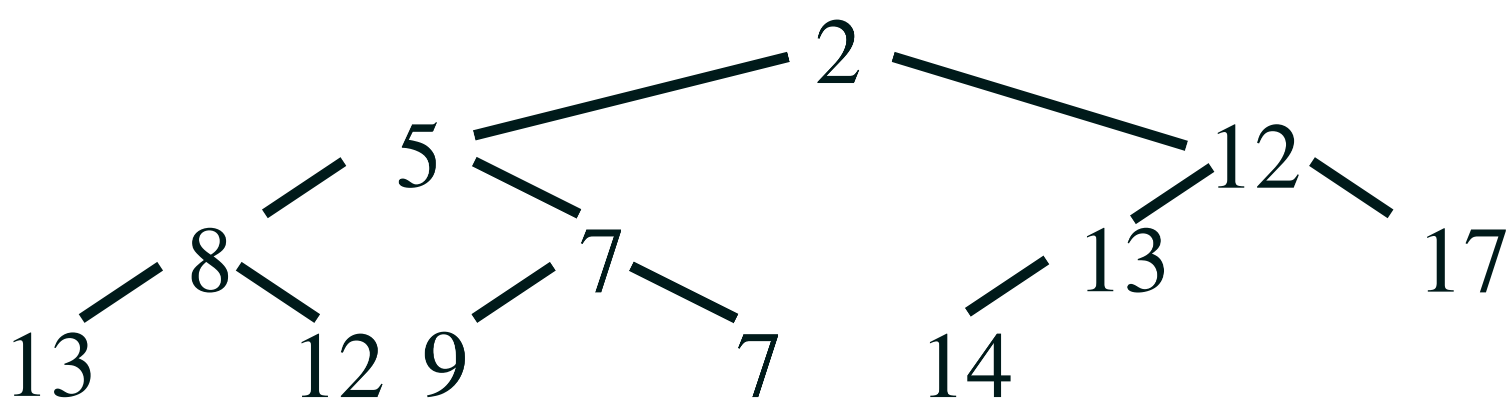
- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering

Adding to the Heap

- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering

Adding to the Heap

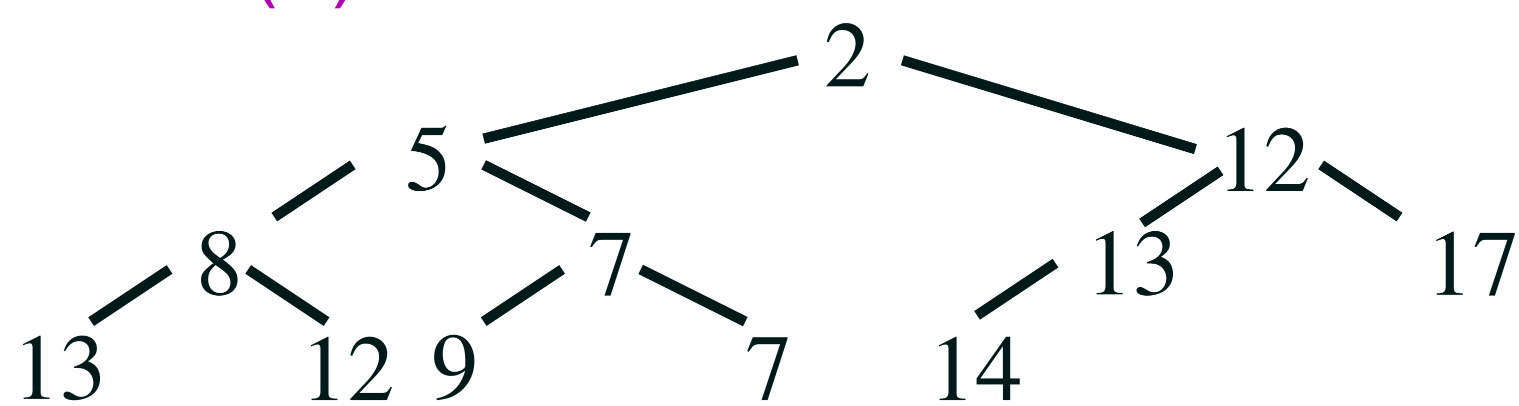
- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering



Adding to the Heap

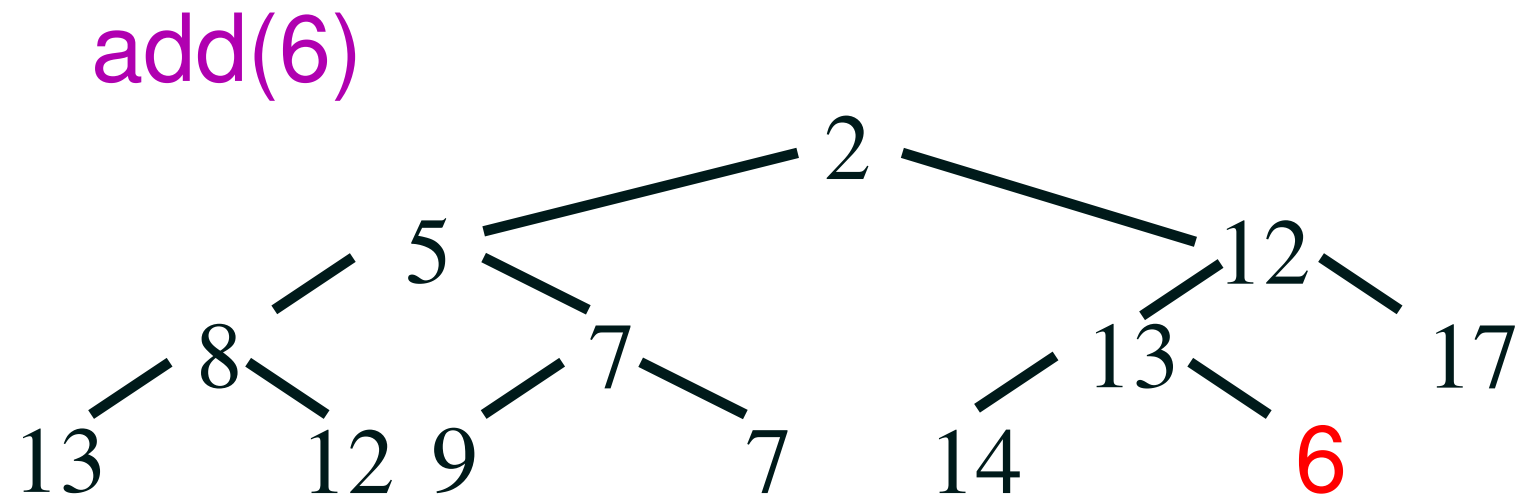
- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering

add(6)



Adding to the Heap

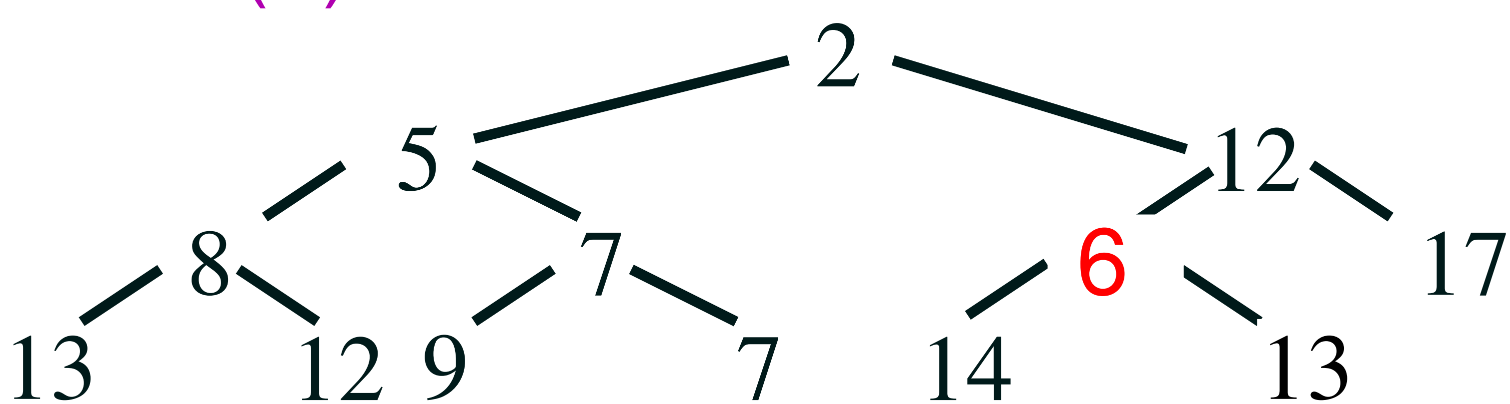
- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering



Adding to the Heap

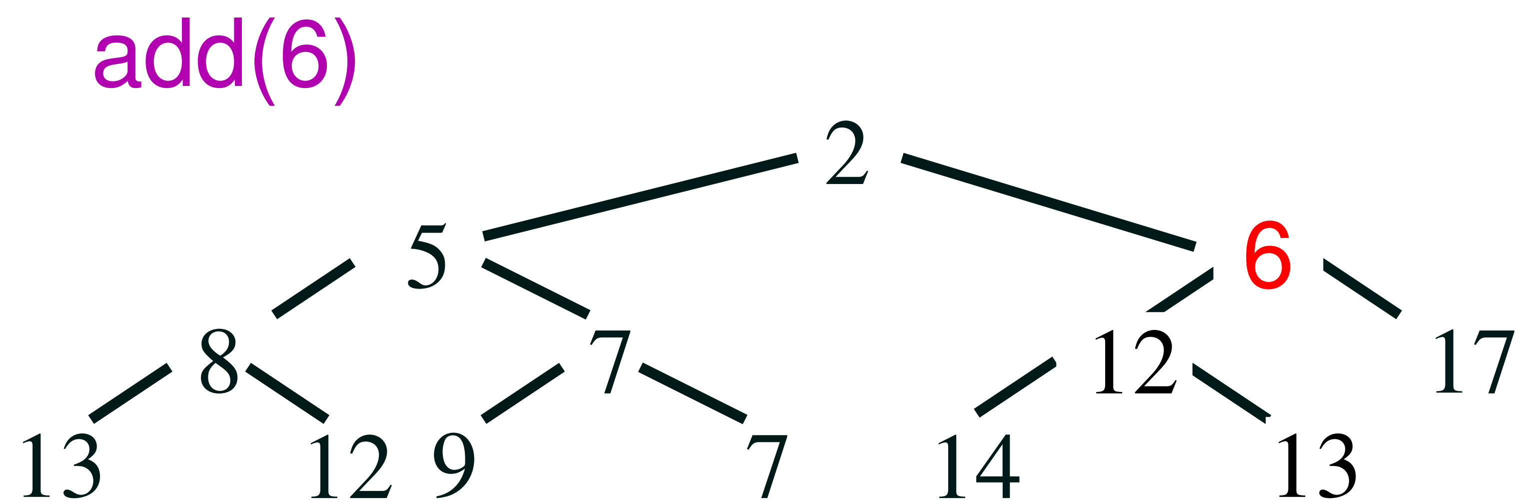
- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering

add(6)



Adding to the Heap

- Heaps are easy to maintain
- To add an element to a heap:
  - ⋆ Add the element to the next available space in the tree
  - ⋆ Percolate the node up the tree to maintain the correct ordering



Outline

1. Heaps
2. **Priority Queues**
   - Array Implementation
3. Heap Sort

Priority Queues

- One of the prime uses of heaps is to implement a Priority Queue
  - ⋆ A Priority Queue is a queue with priorities
  - ⋆ That is, we assign a priority to each element we add
  - ⋆ The head of the queue is the element with highest priority (smallest number)

- Used, for example, to implement “greedy algorithms”

Priority Queue Interface

- A simple Priority Queue interface might include

```
interface PQ<T>
{
    int size();                              // return number of elements
    boolean isEmpty();                       // true if queue is empty,
    void add(T element, int priority);       // add element to queue
    T getMin();                              // return head of queue
    T removeMin();                           // dequeue head of queue
}
```

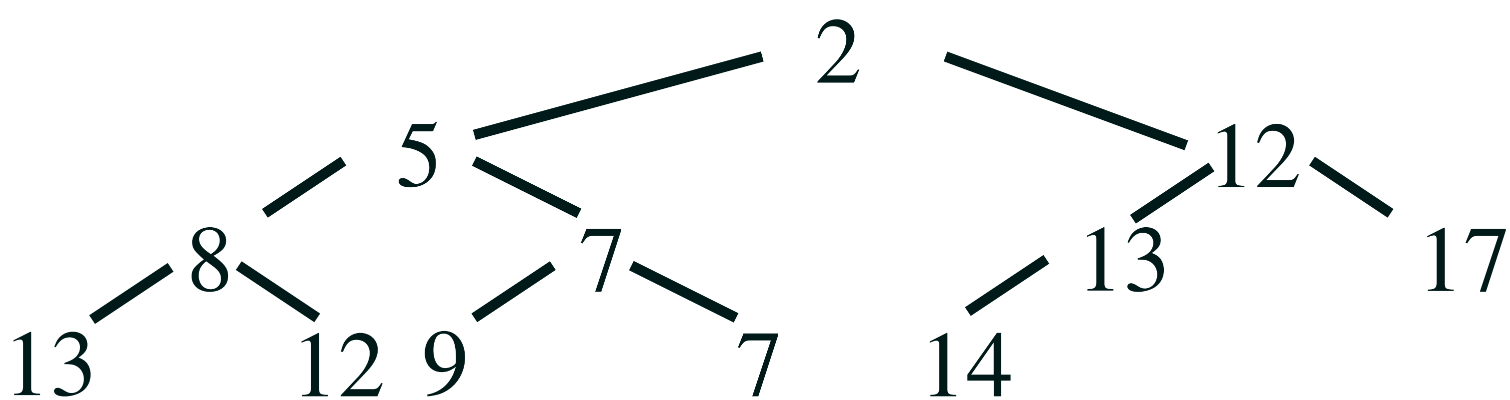
- Java has a PriorityQueue class which extends AbstractQueue and is part of the Java Collection framework

removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child

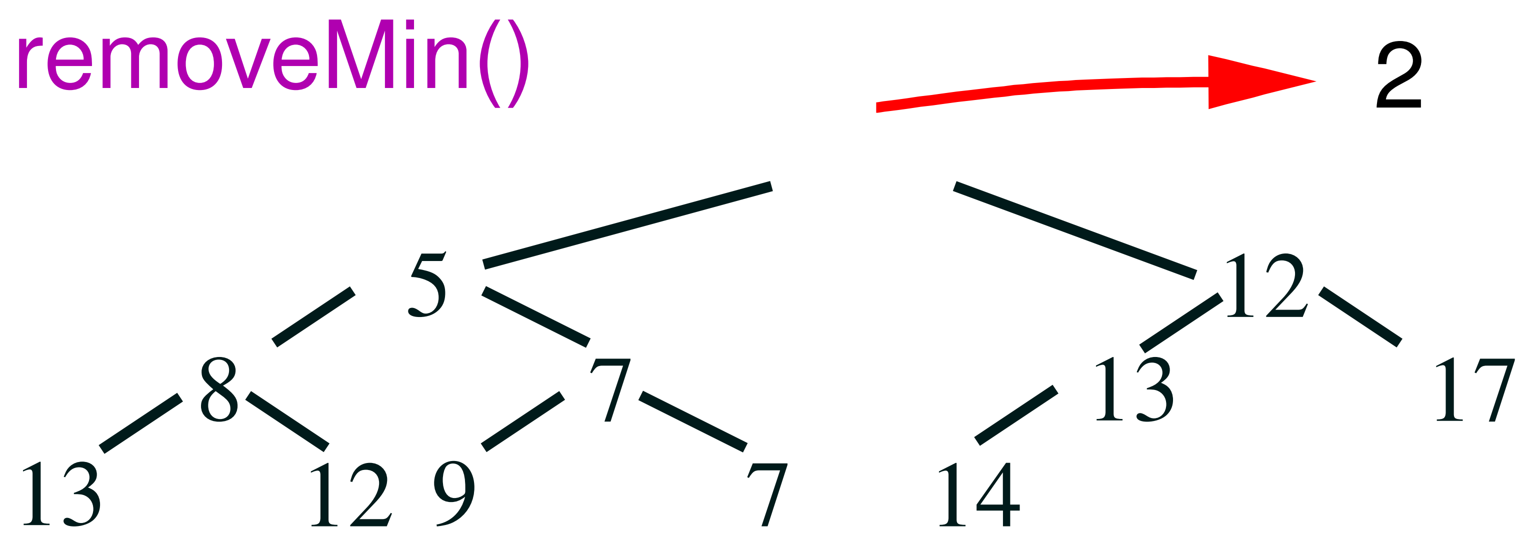
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



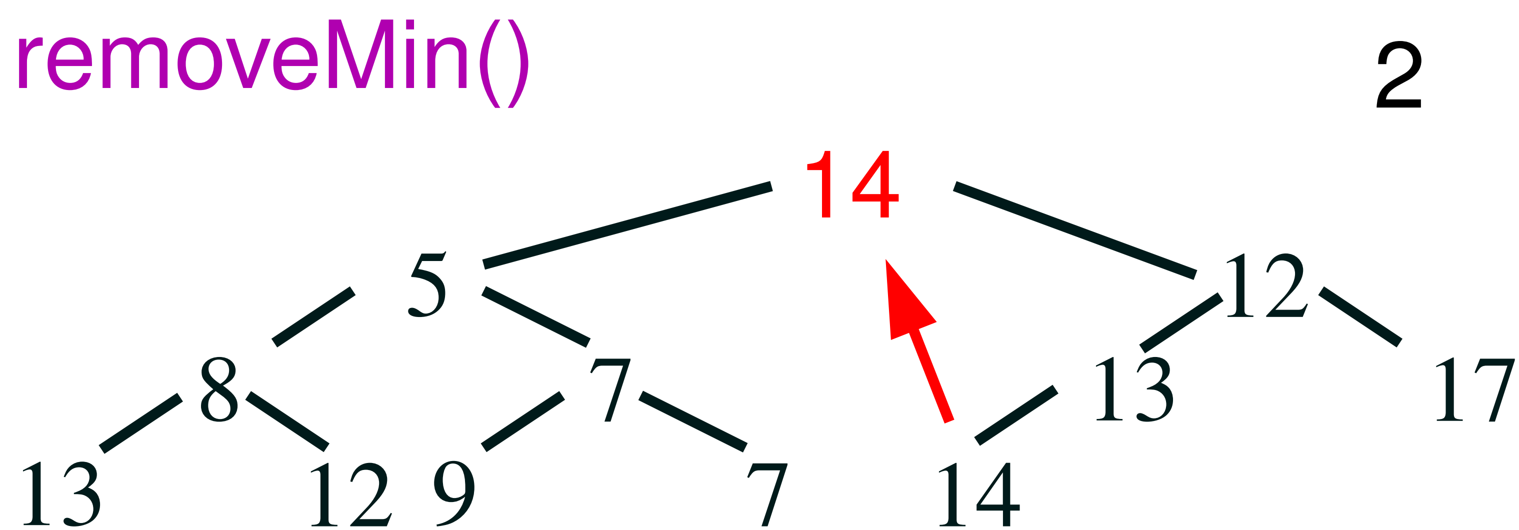
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



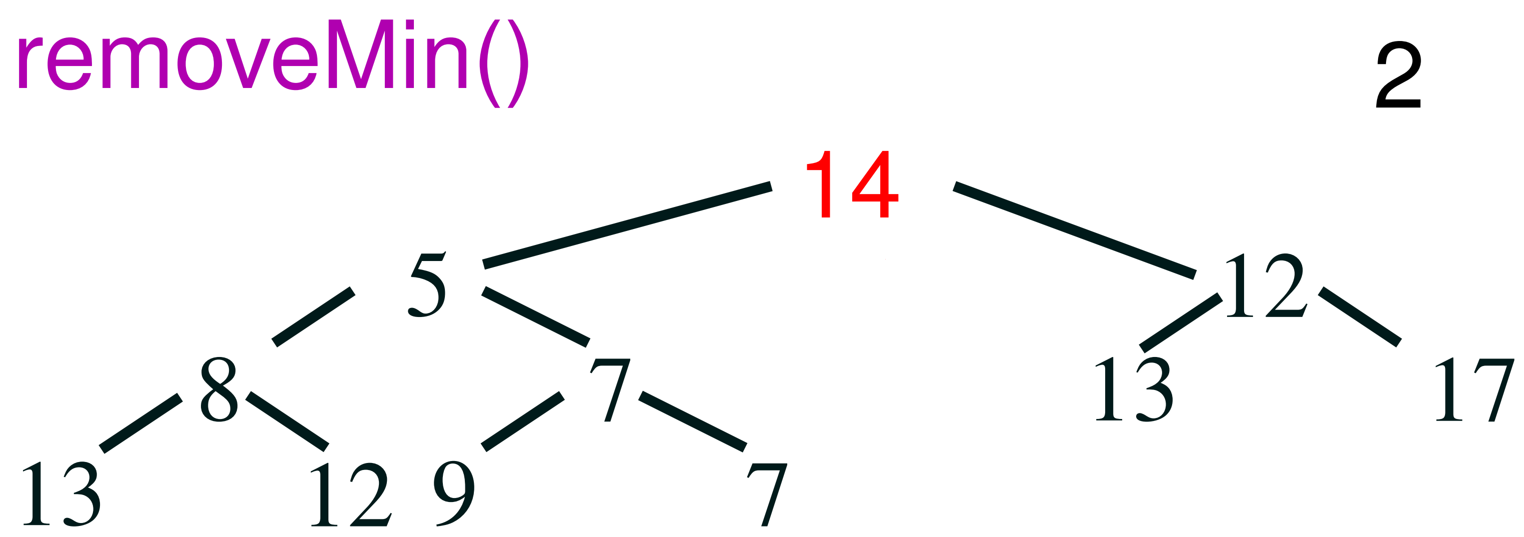
`removeMin`

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



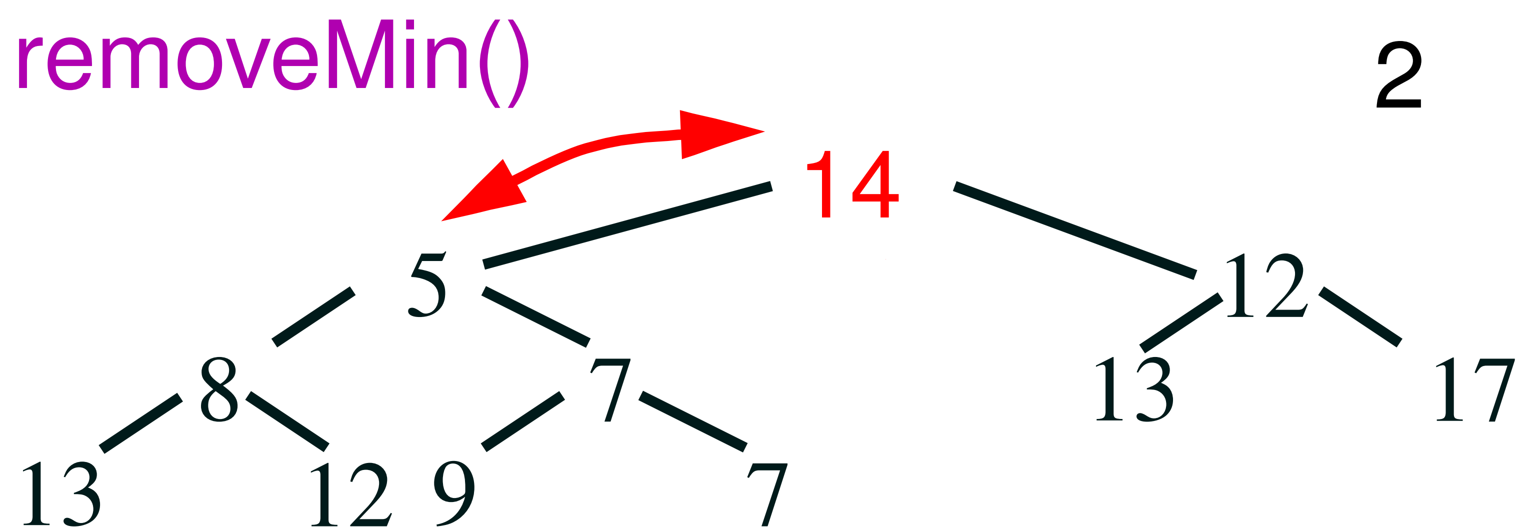
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



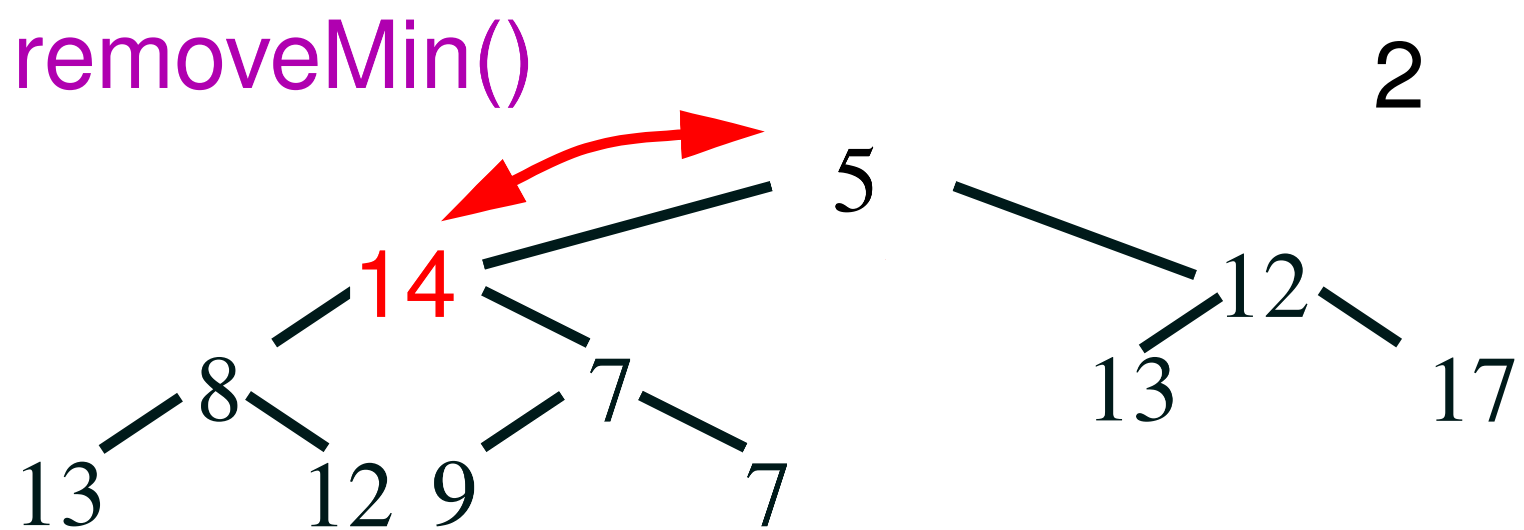
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - Pop the root
  - Replace it with the last element in the heap
  - Percolate this element down to the bottom of the heap choosing the minimum child



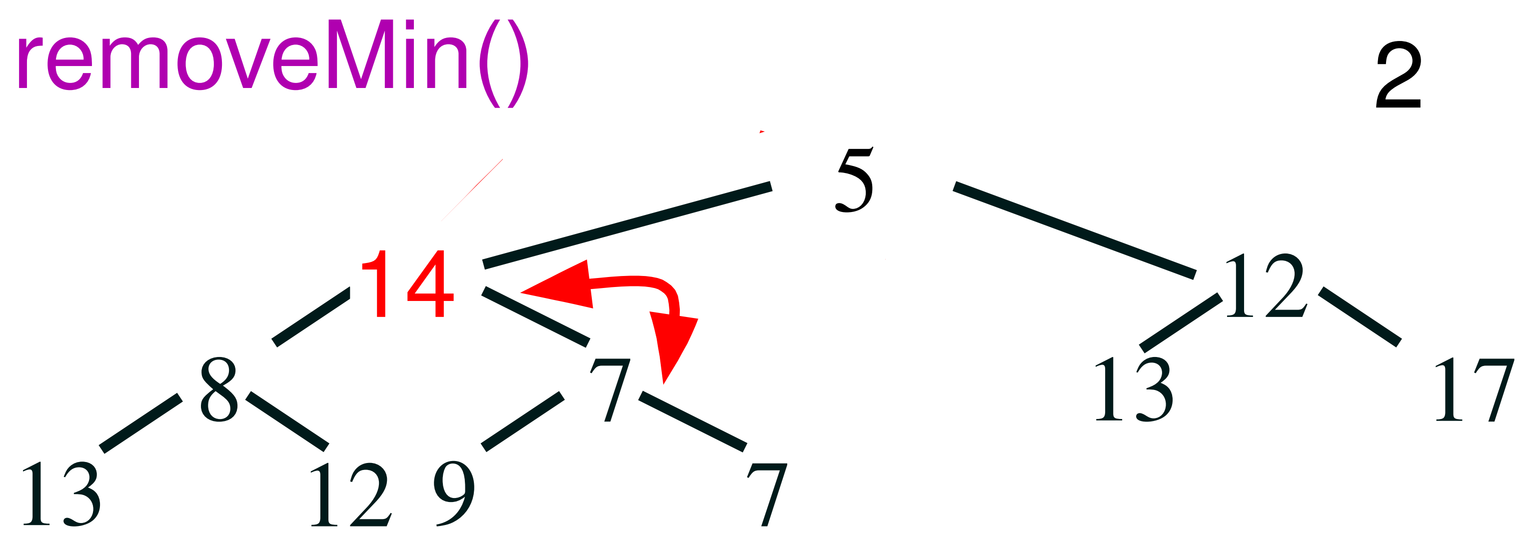
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



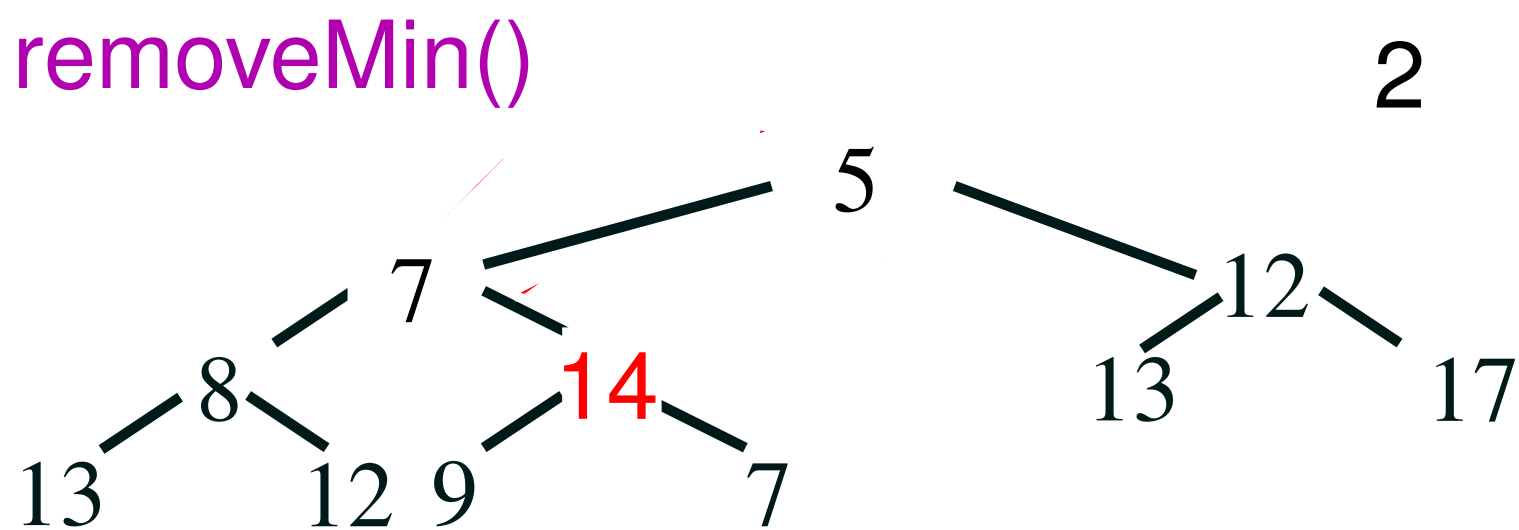
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



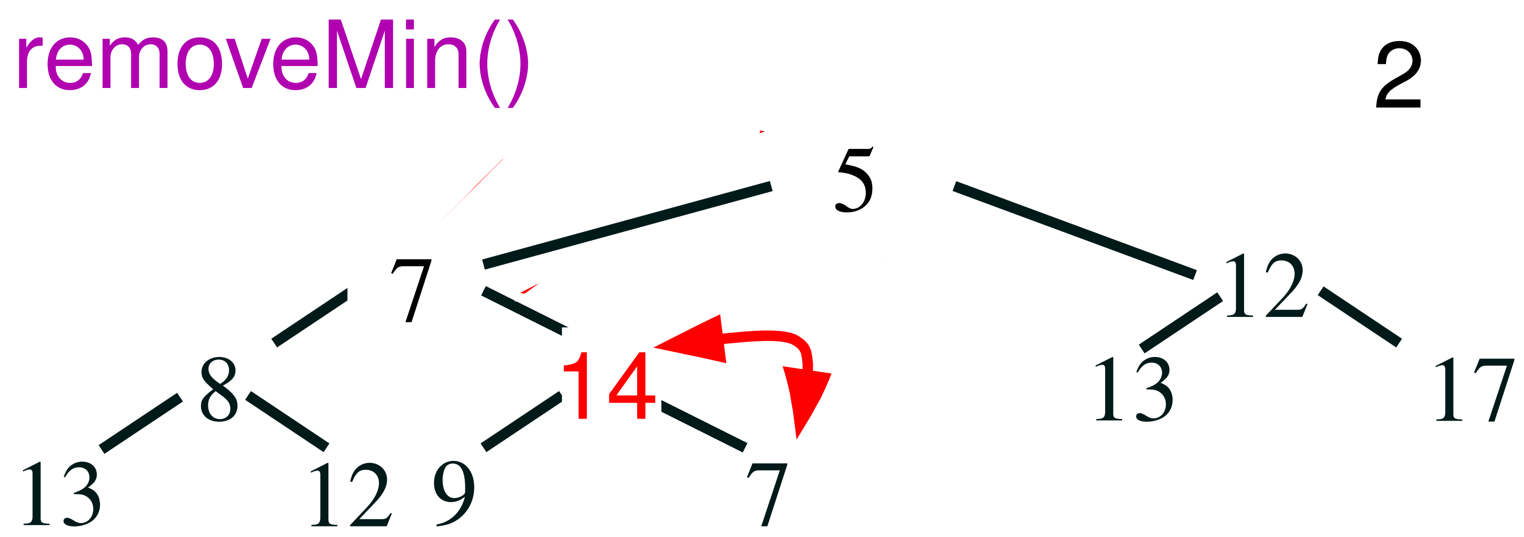
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



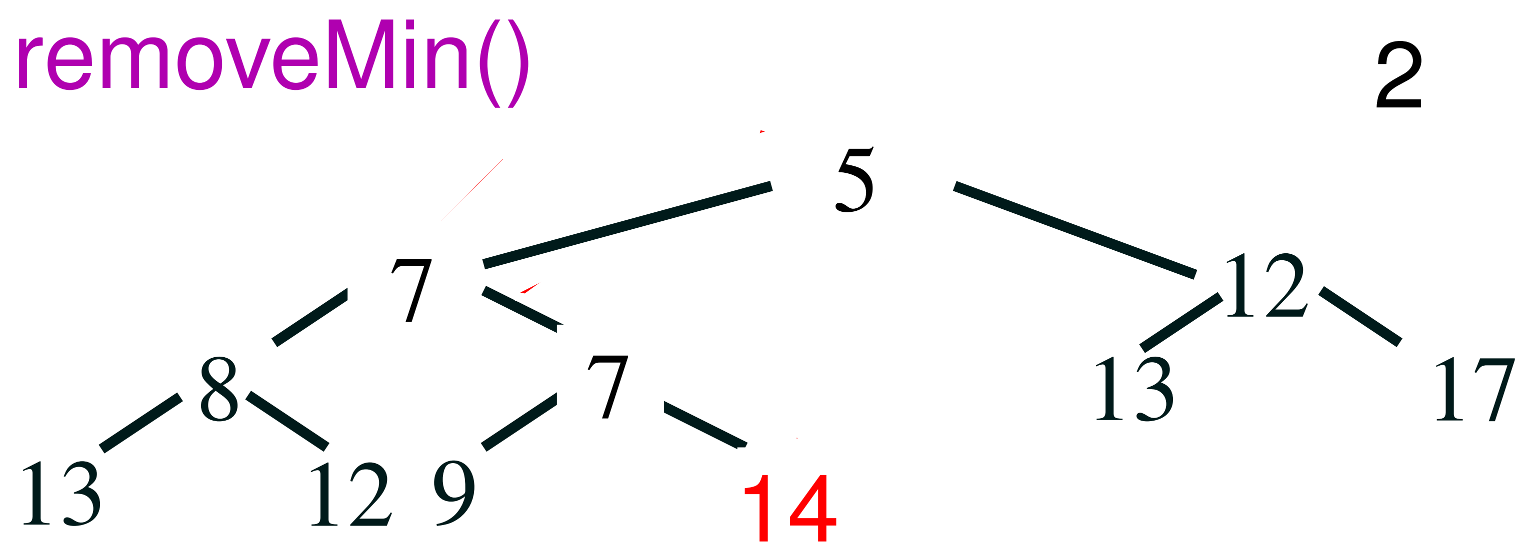
removeMin

- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



removeMin

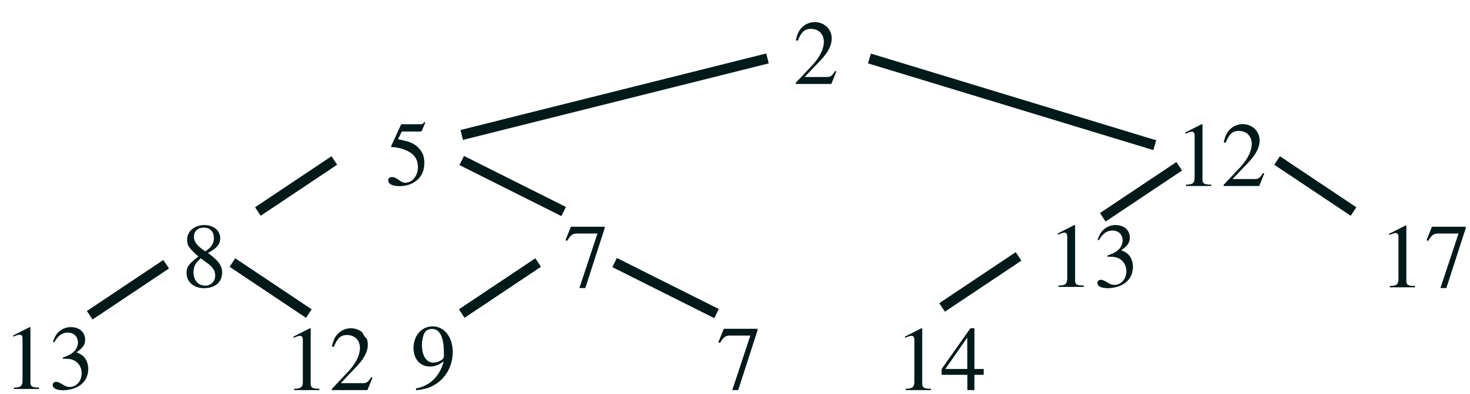
- The minimum element is the root of the tree
- To remove this element:
  - ⋆ Pop the root
  - ⋆ Replace it with the last element in the heap
  - ⋆ Percolate this element down to the bottom of the heap choosing the minimum child



Array Implementation of Heaps

- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!

2	5	12	8	7	13	17	13	12	9	7	14

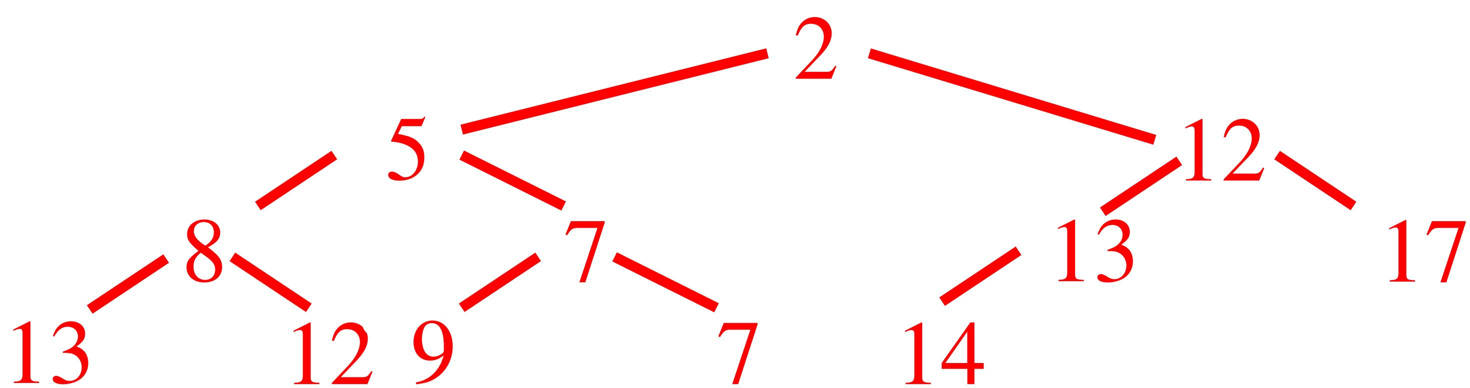


Array Implementation of Heaps

- The surprising thing about heaps is that they can be implemented efficiently using arrays

- This is because the tree is complete!

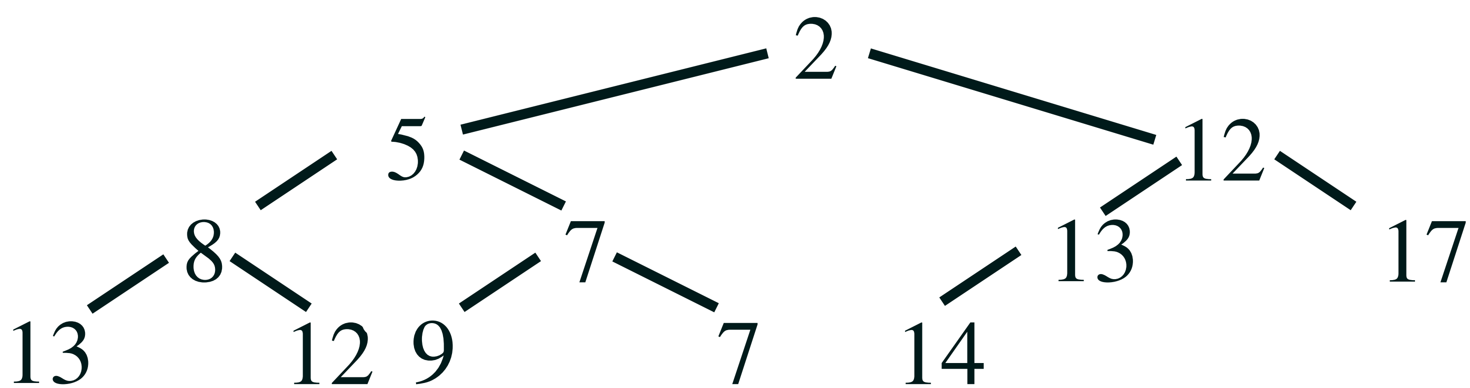
2	5	12	8	7	13	17	13	12	9	7	14



Array Implementation of Heaps

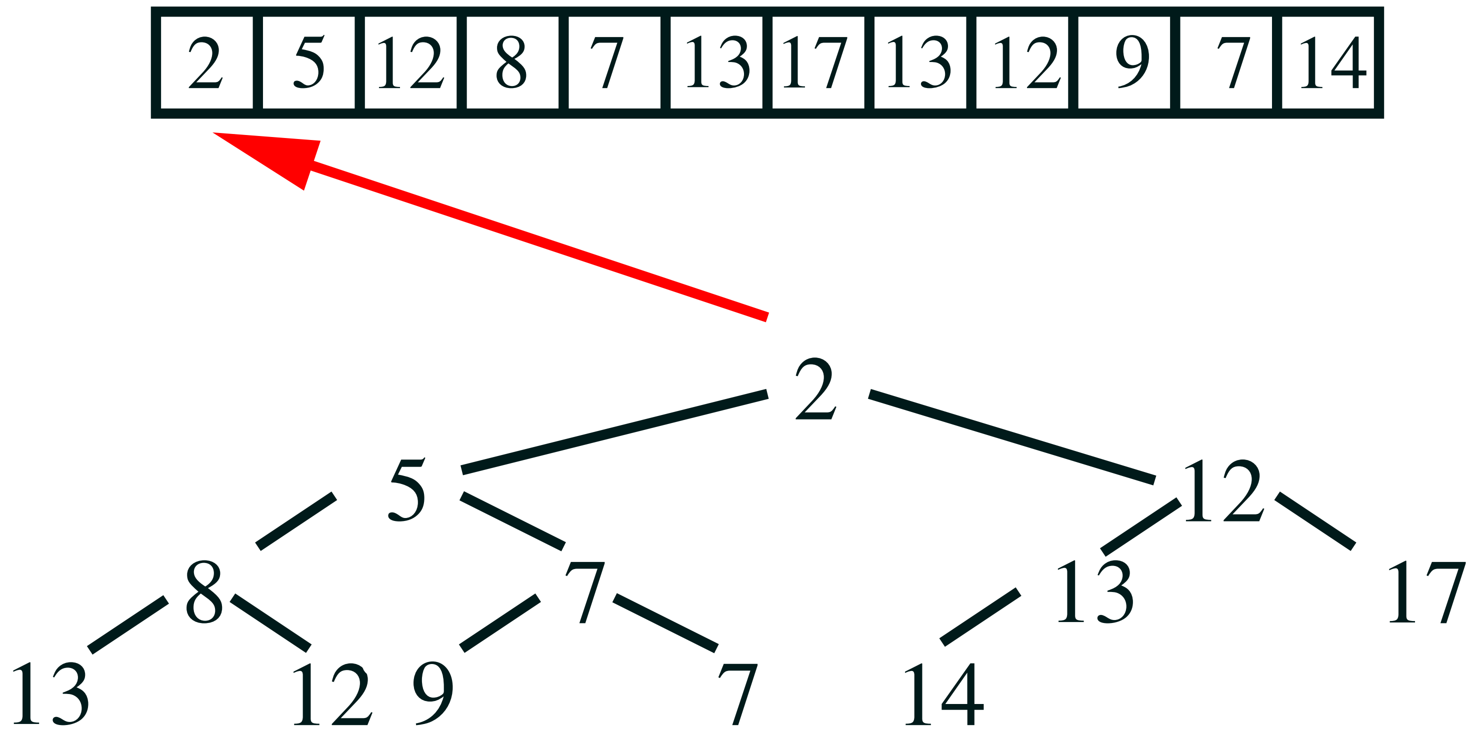
- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!

2	5	12	8	7	13	17	13	12	9	7	14



Array Implementation of Heaps

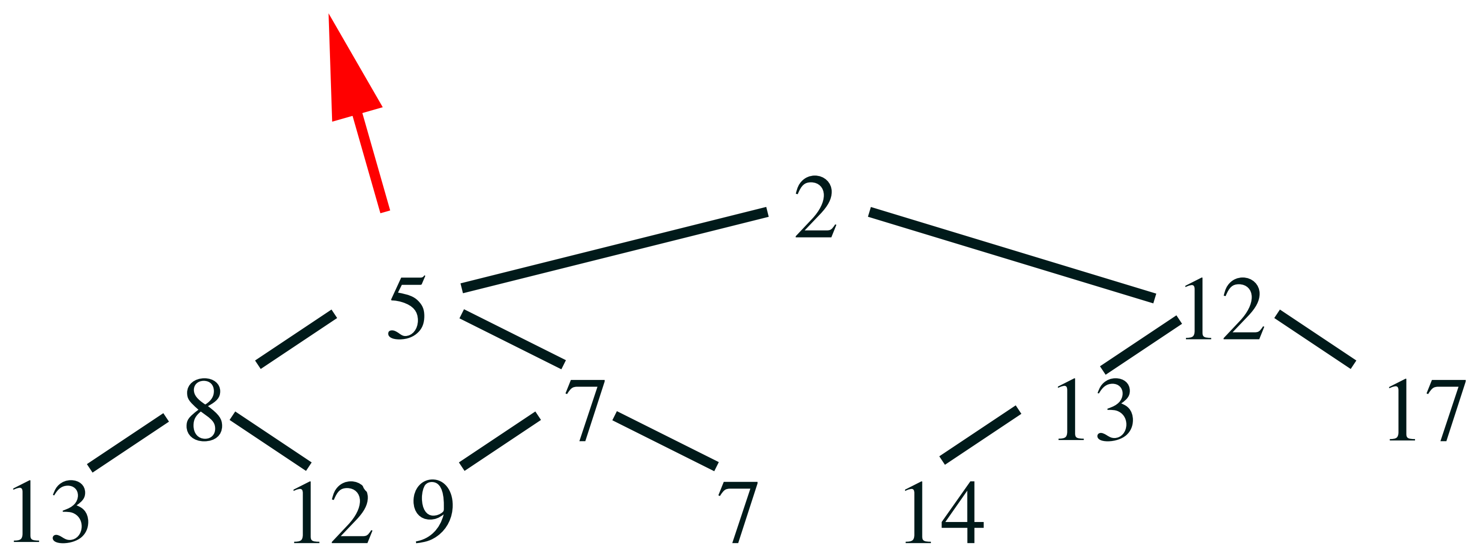
- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!



Array Implementation of Heaps

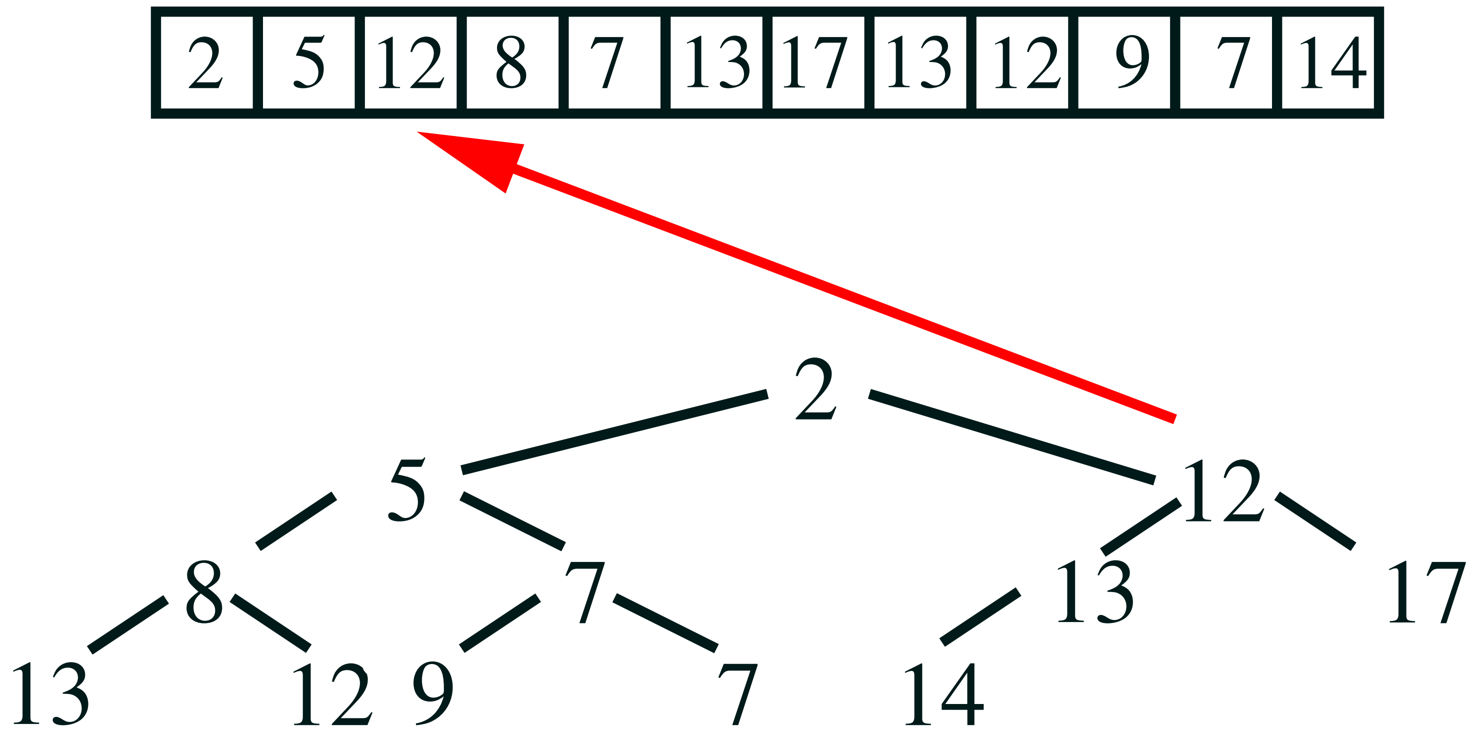
- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!

2	5	12	8	7	13	17	13	12	9	7	14



Array Implementation of Heaps

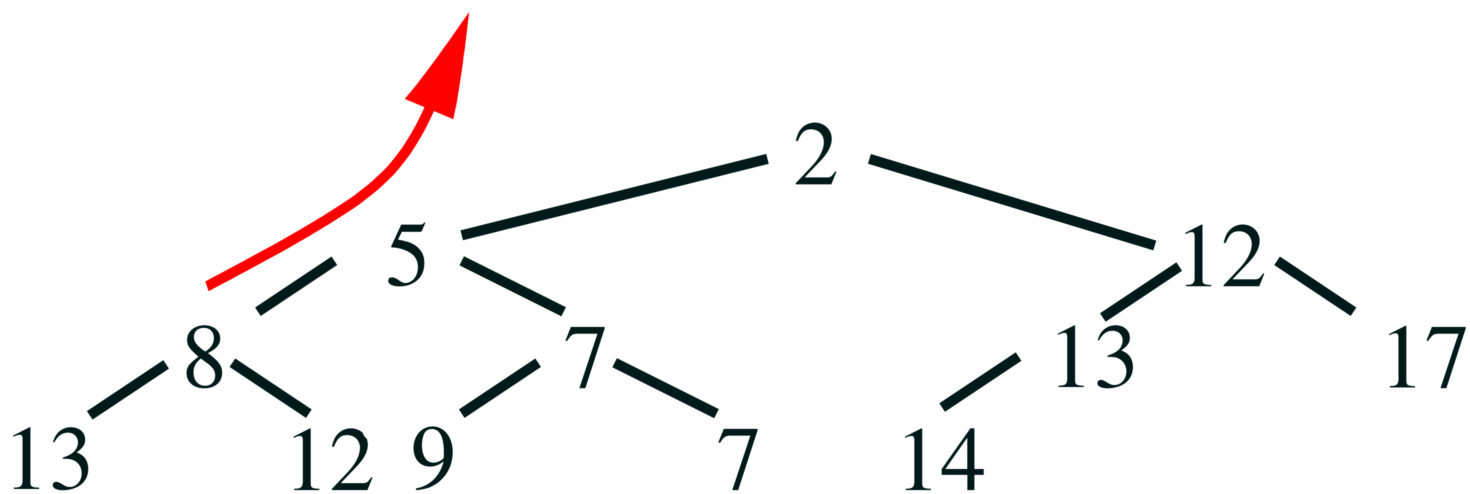
- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!



Array Implementation of Heaps

- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!

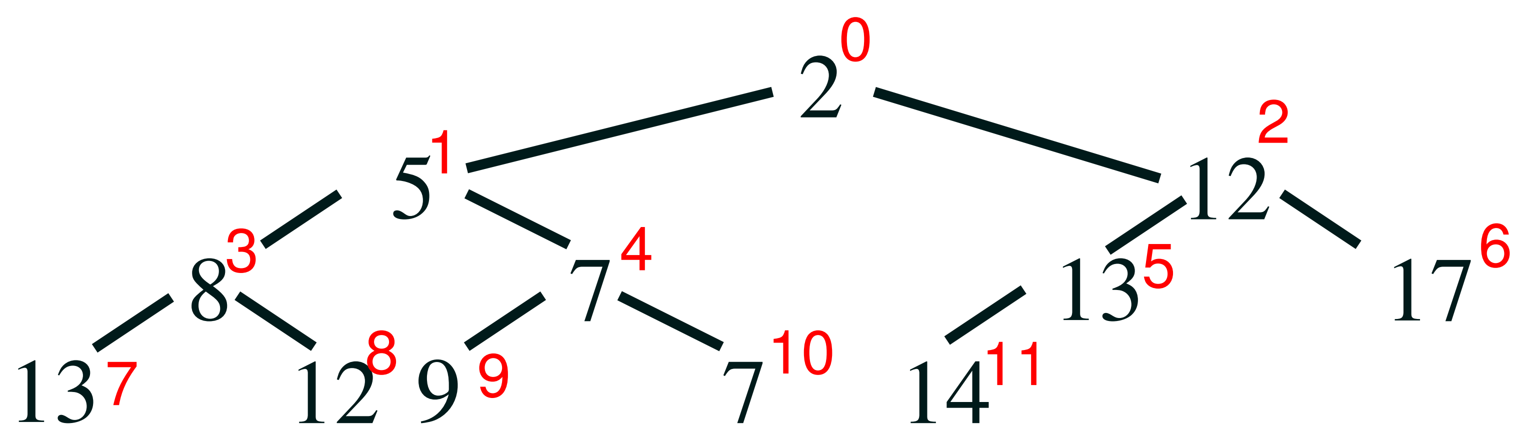
2	5	12	8	7	13	17	13	12	9	7	14



Array Implementation of Heaps

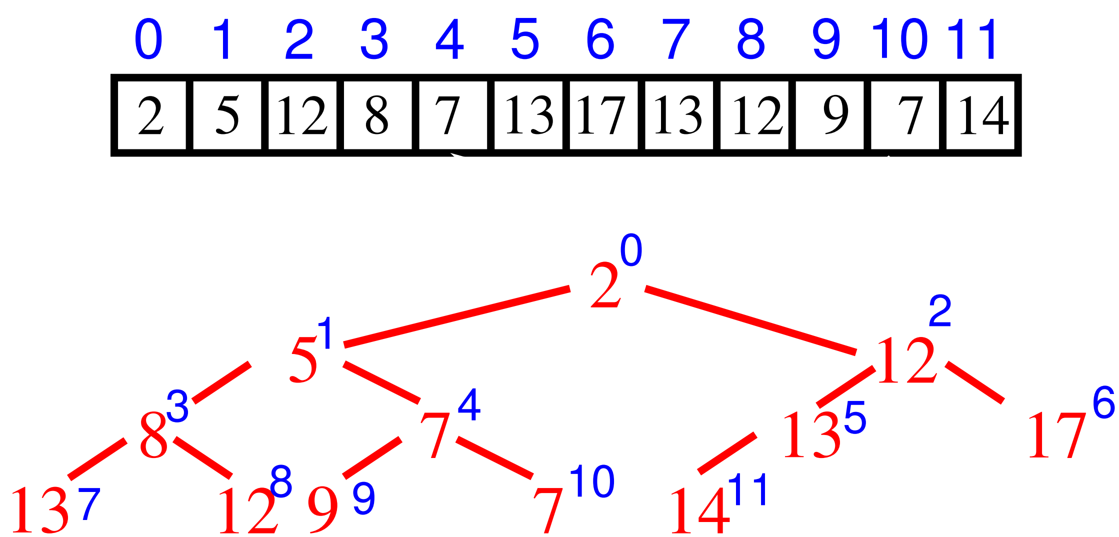
- The surprising thing about heaps is that they can be implemented efficiently using arrays
- This is because the tree is complete!

0	1	2	3	4	5	6	7	8	9	10	11
2	5	12	8	7	13	17	13	12	9	7	14



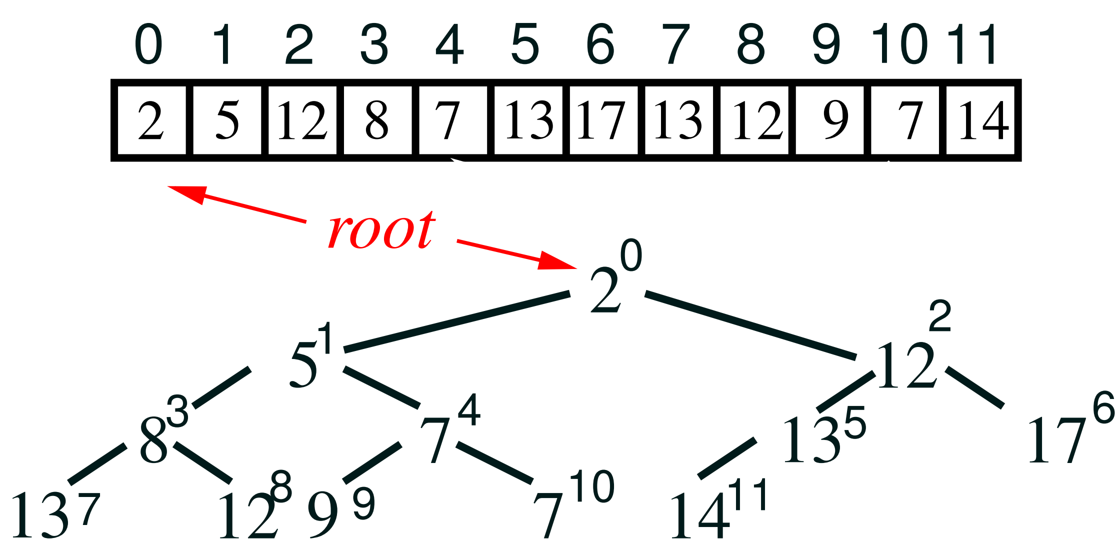
Navigating a Heap

- To navigate a heap we note that
  - ⋆ The root of the tree is at array location 0
  - ⋆ The last element in the heap is at array location `size()-1`
  - ⋆ The parent of a node k is at array location $\lfloor (k-1)/2 \rfloor$
  - ⋆ The children of node k are at array locations $2k+1$ and $2k+2$



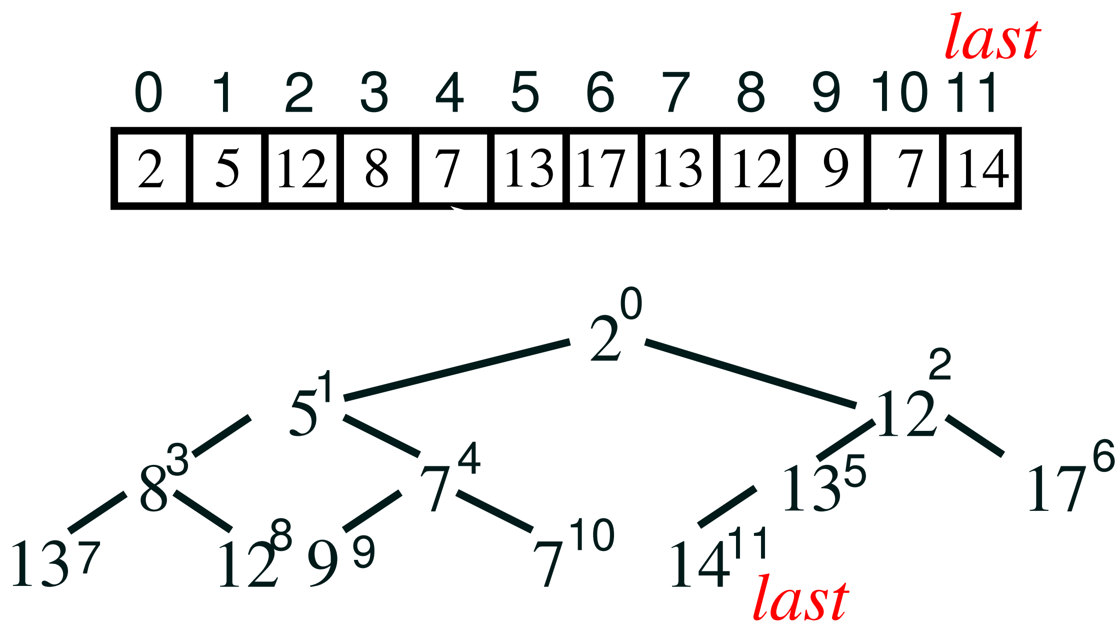
Navigating a Heap

- To navigate a heap we note that
  - ⋆ The root of the tree is at array location 0
  - ⋆ The last element in the heap is at array location `size()-1`
  - ⋆ The parent of a node k is at array location $\lfloor (k-1)/2 \rfloor$
  - ⋆ The children of node k are at array locations $2k+1$ and $2k+2$



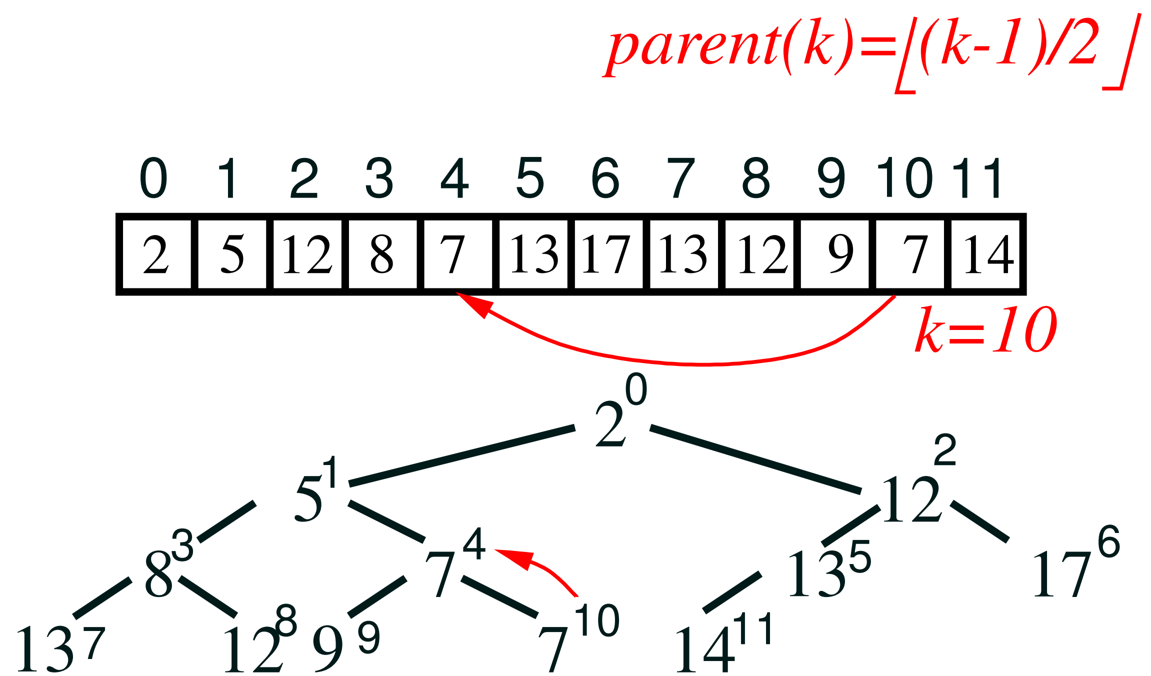
Navigating a Heap

- To navigate a heap we note that
  - ⋆ The root of the tree is at array location 0
  - ⋆ The last element in the heap is at array location `size()-1`
  - ⋆ The parent of a node k is at array location $\lfloor (k-1)/2 \rfloor$
  - ⋆ The children of node k are at array locations $2k+1$ and $2k+2$



Navigating a Heap

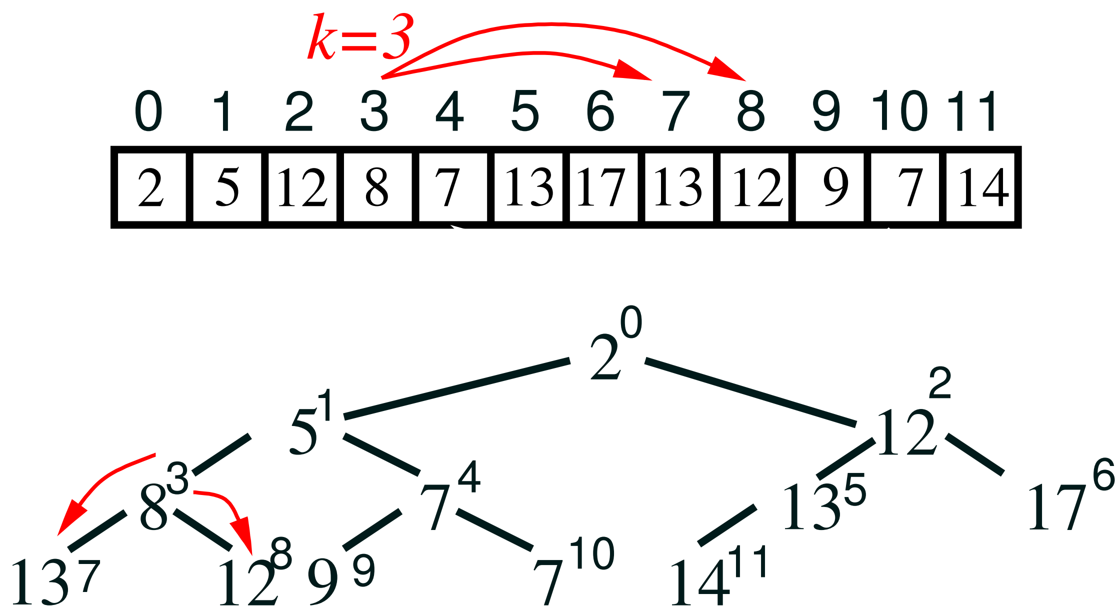
- To navigate a heap we note that
  - ⋆ The root of the tree is at array location 0
  - ⋆ The last element in the heap is at array location `size()-1`
  - ⋆ The parent of a node k is at array location $\lfloor (k-1)/2 \rfloor$
  - ⋆ The children of node k are at array locations $2k+1$ and $2k+2$



Navigating a Heap

- To navigate a heap we note that
  - The root of the tree is at array location 0
  - The last element in the heap is at array location `size()-1`
  - The parent of a node k is at array location $\lfloor (k-1)/2 \rfloor$
  - The children of node k are at array locations $2k+1$ and $2k+2$

children(k) =2k+1,2k+2



Code for a Priority Queue

```
import java.util.*;

public class HeapPQ<T> implements PQ<T>
{
    private List<T> list;

    public HeapPQ(int initialCapacity)
    {
        list = new ArrayList<T>(initialCapacity);
    }

    public int size() { return list.size(); }

    public boolean isEmpty() { return list.size()==0; }

    public T getMin() { return list.get(0); }
```

Adding an Element

```java
public void add(T element)
{
    list.add(element);
    percolateUp();
}

private void percolateUp()
{
    int child = list.size()-1;

    while (child>0) {
        int parent = (child-1)>>1;   // floor((child-1)/2)
        if (compare(child, parent) >= 0)
            break;
        swap(parent,child);
        child = parent;
    }
}
```

- `compare` and `swap` are trivial helper function

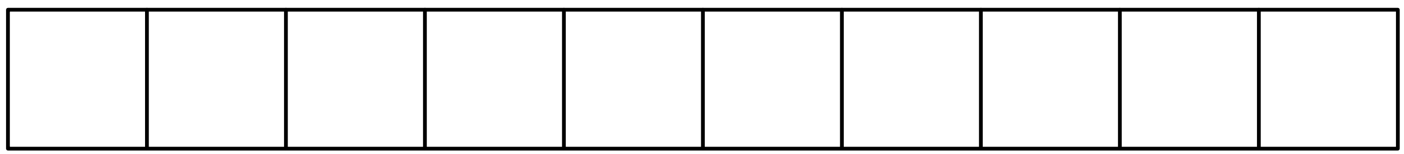
Popping the Top

```
public T removeMin() {
    T minElem = list.get(0);
    list.set(0, list.get(list.size()-1));
    list.remove(list.size()-1);
    percolateDown(0);
    return minElem;
}

private void percolateDown(int parent) {
    int child = (parent<<1) + 1;  // 2*parent+1
    while (child < list.size()) {
        if (child+1 < list.size() && compare(child,child+1) > 0)
            child++;
        if (compare(child, parent)>=0)
            break;
        swap(parent, child);
        parent = child;
        child = (parent<<1) + 1;
    }
}
```

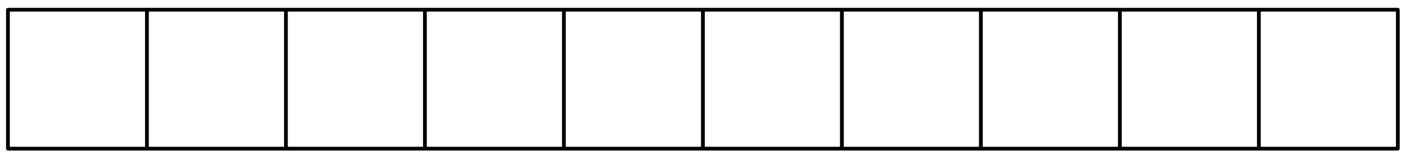
Heaps in Action

```
Heap heap = new Heap;
```



Heaps in Action

`heap.add(5)`

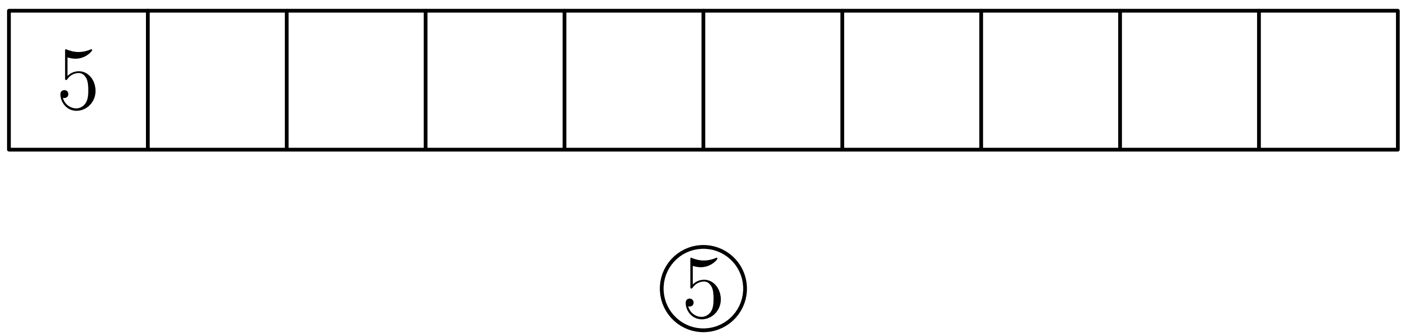


Heaps in Action

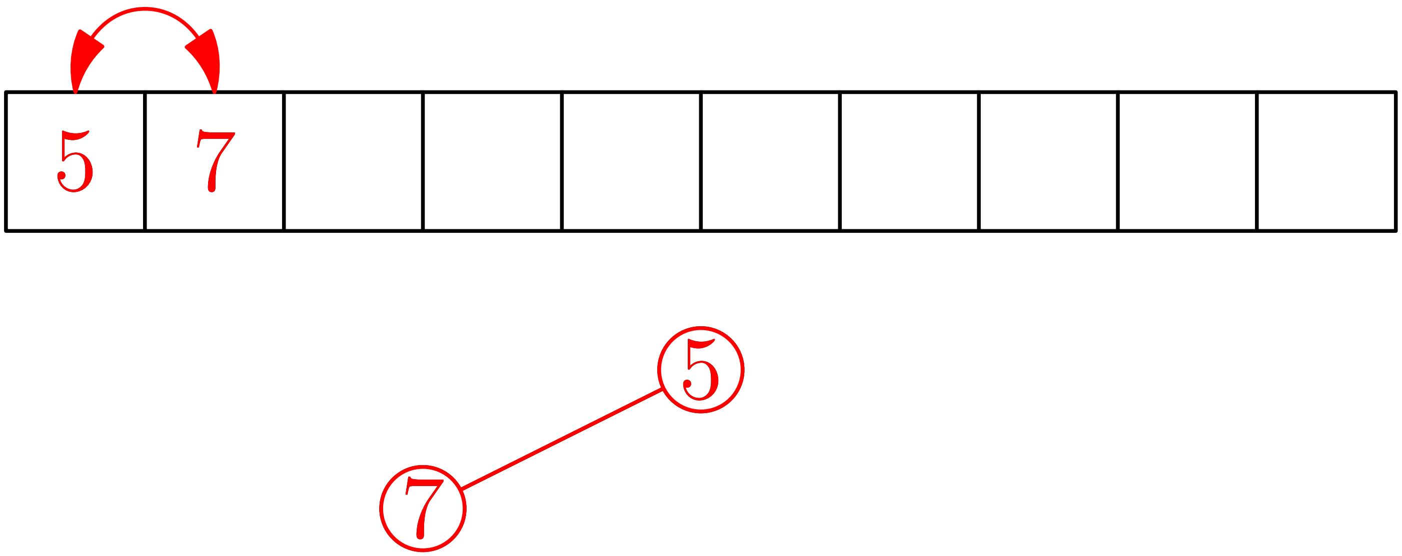
5									

Heaps in Action

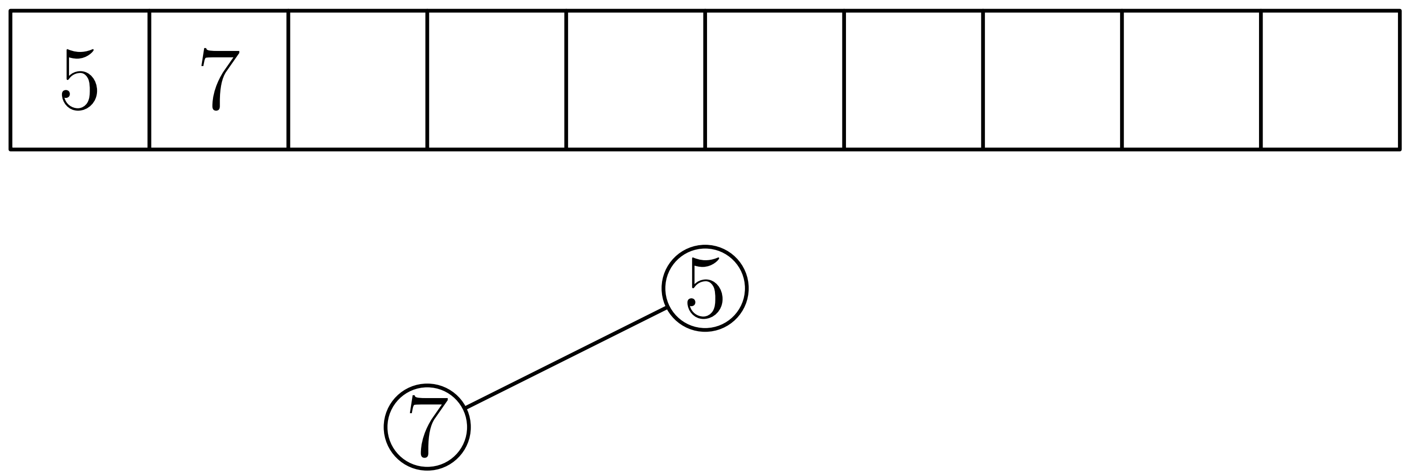
`heap.add(7)`



Heaps in Action



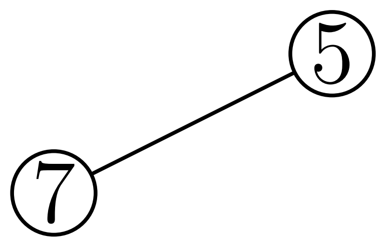
Heaps in Action



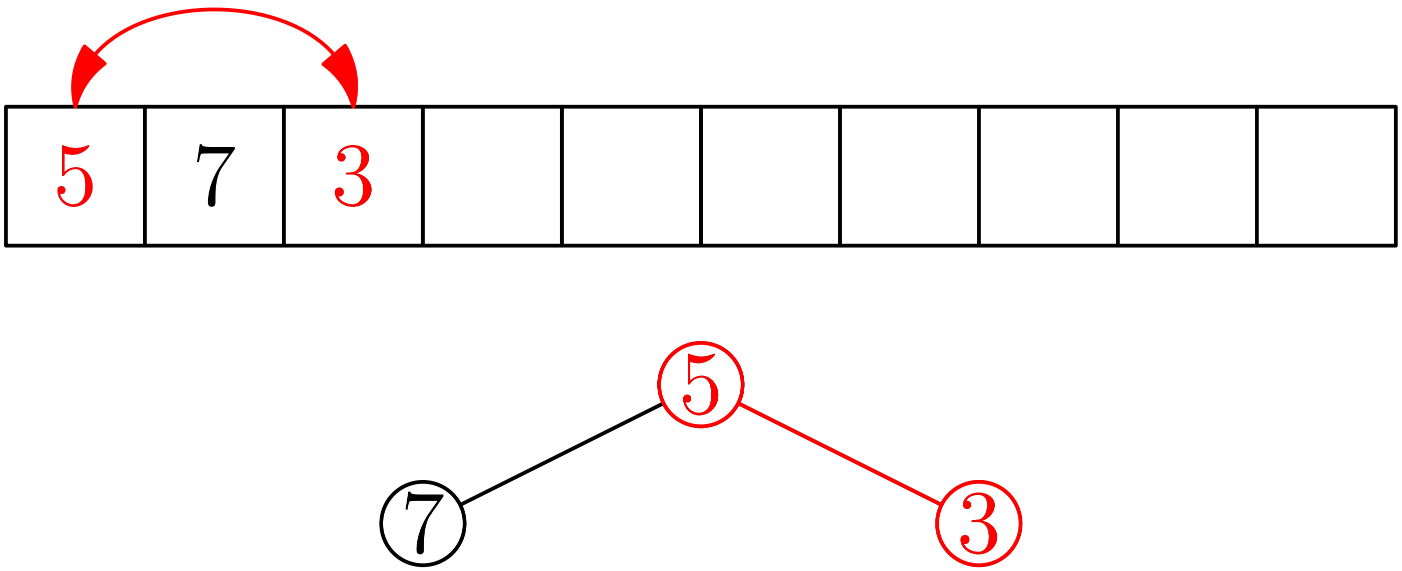
Heaps in Action

`heap.add(3)`

5	7								

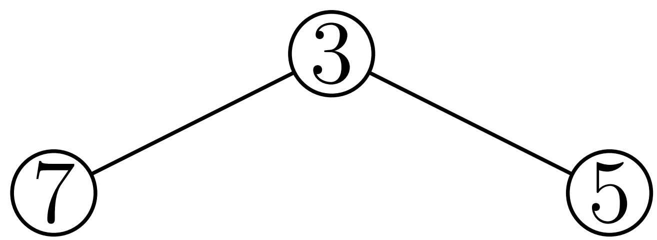


Heaps in Action



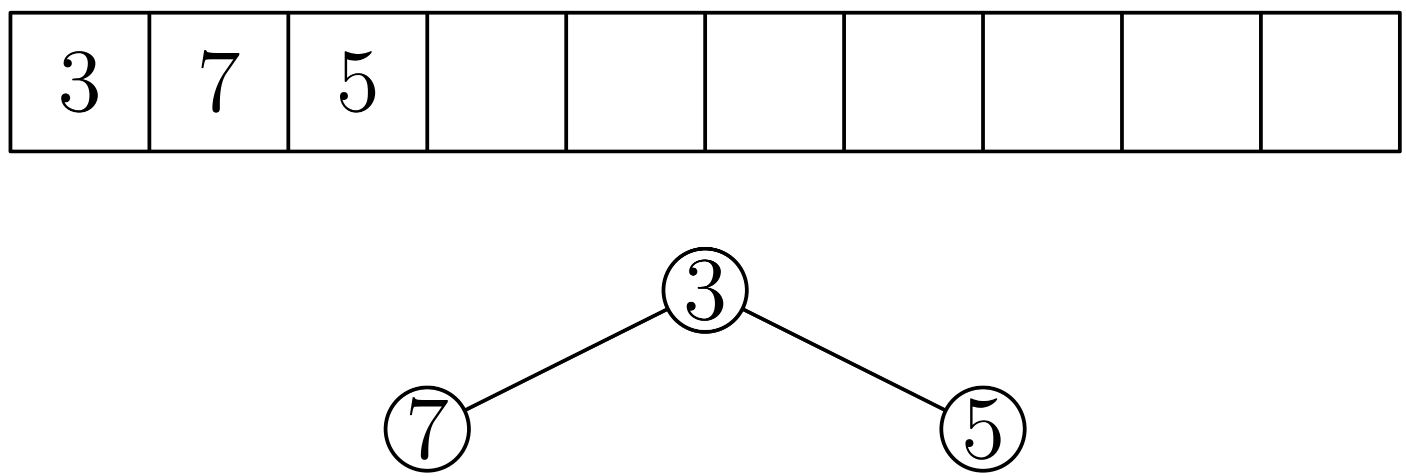
Heaps in Action

3	7	5							

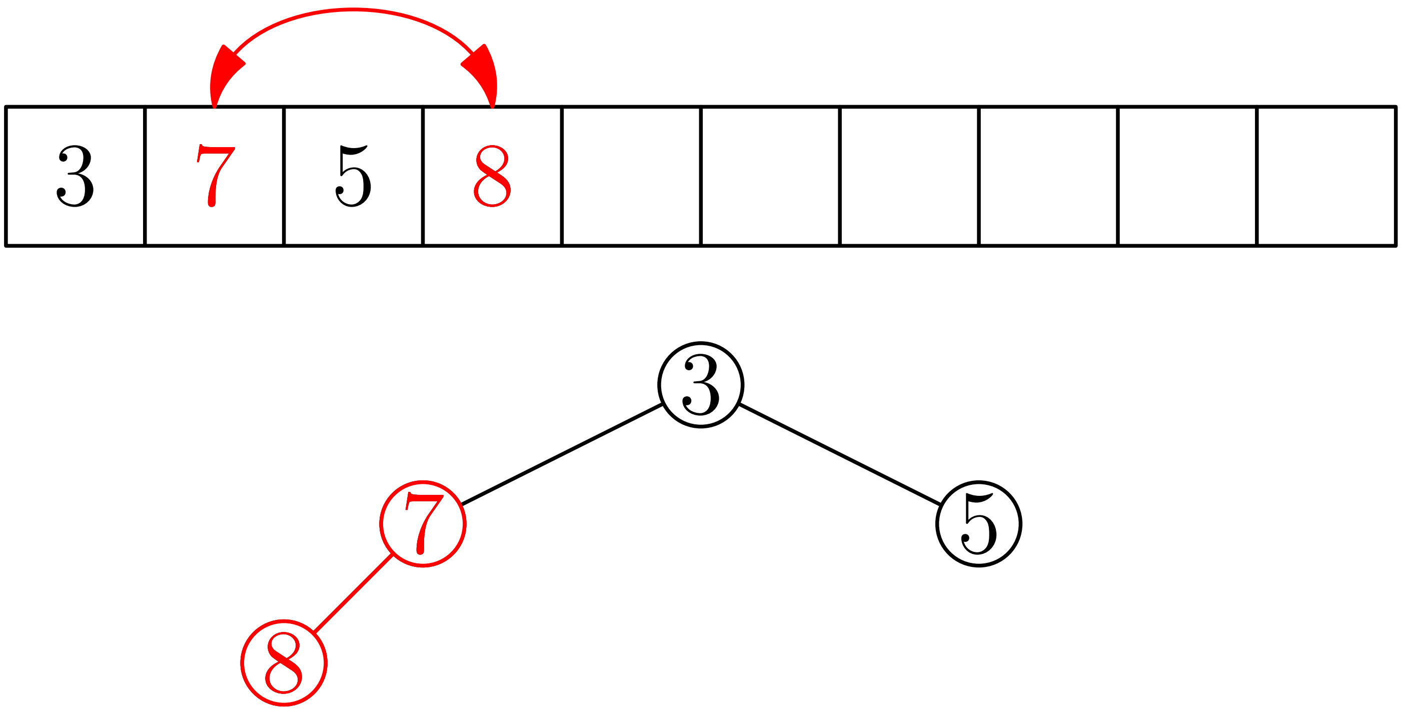


Heaps in Action

`heap.add(8)`

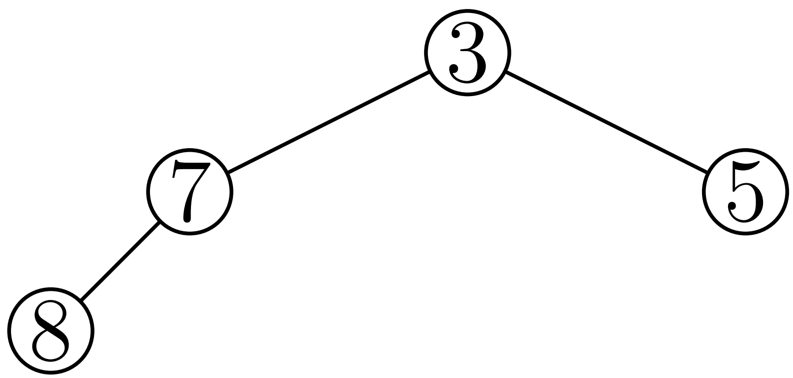


Heaps in Action



Heaps in Action

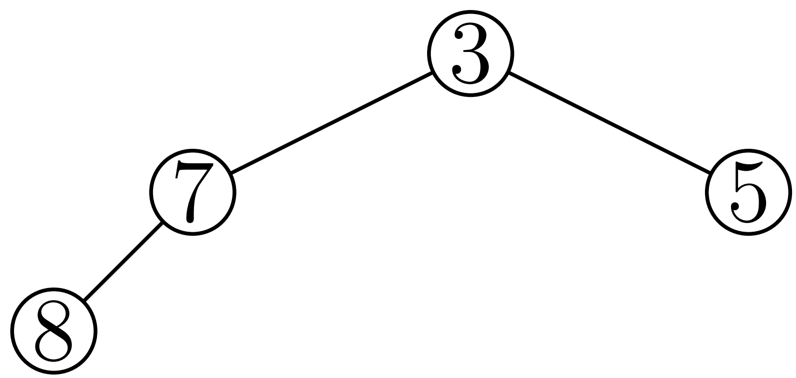
3	7	5	8						



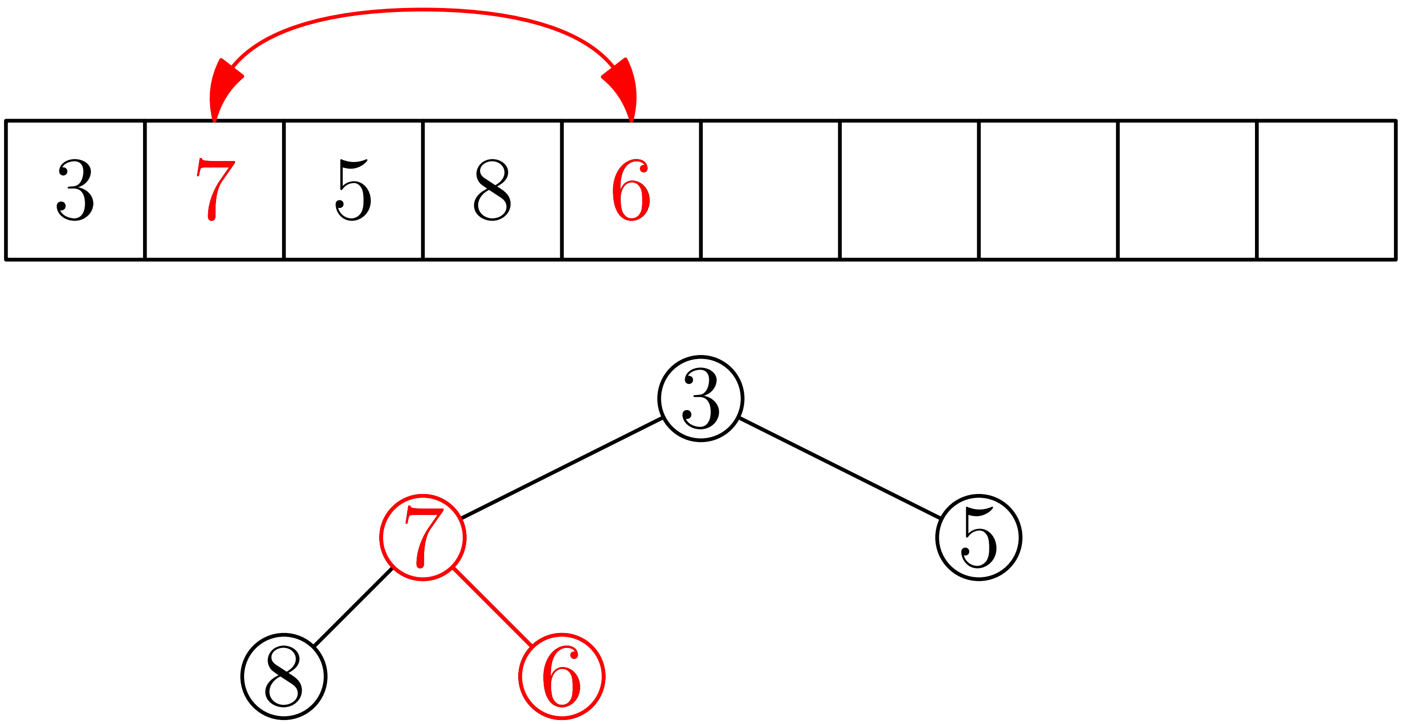
Heaps in Action

`heap.add(6)`

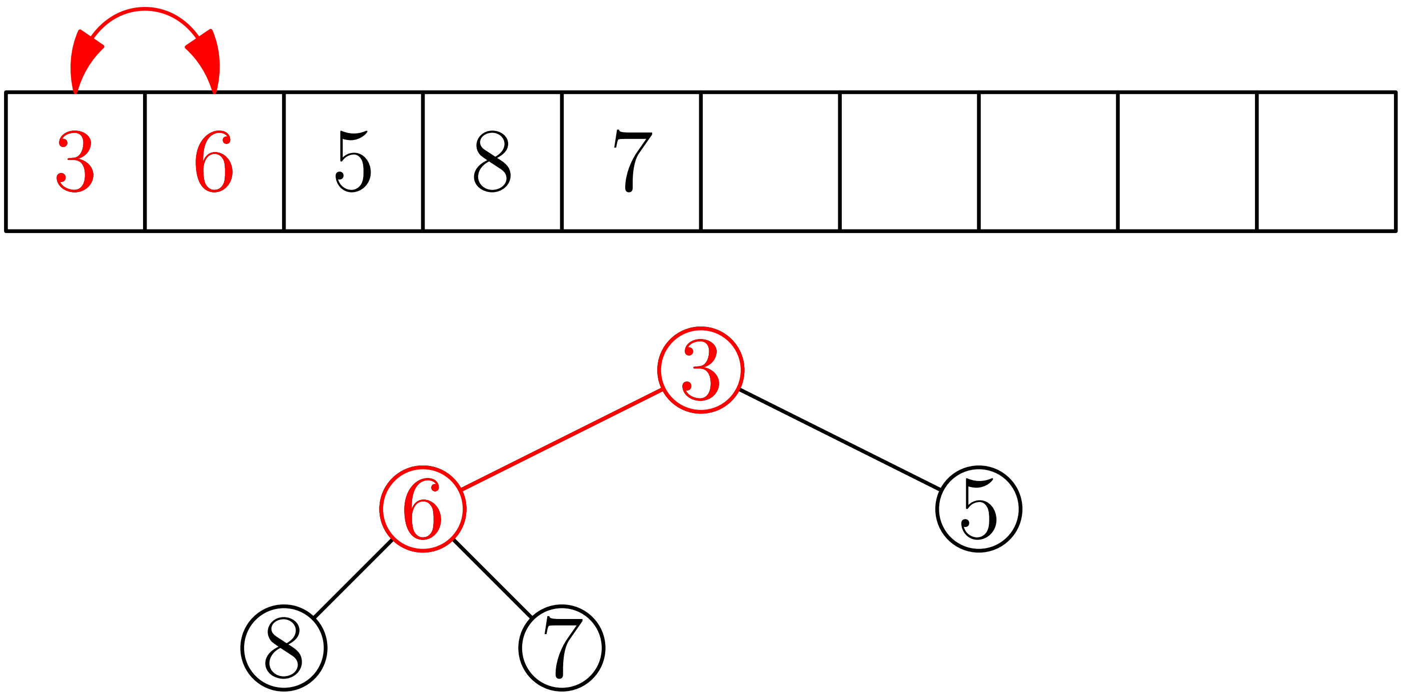
3	7	5	8						



Heaps in Action

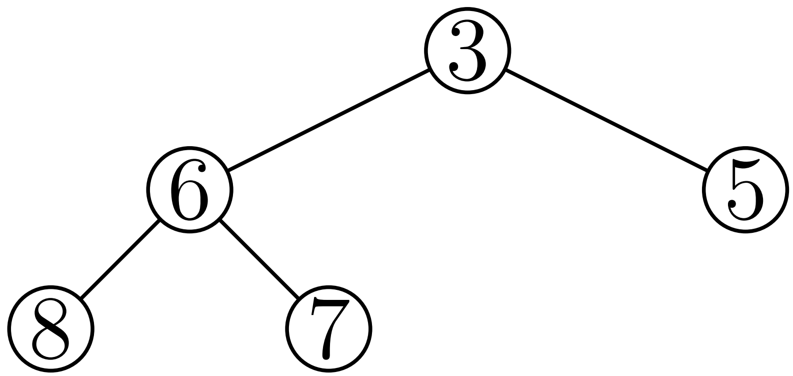


Heaps in Action



Heaps in Action

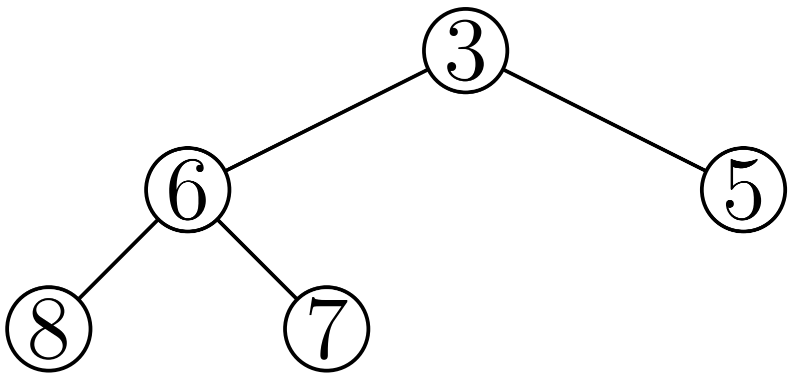
3	6	5	8	7					



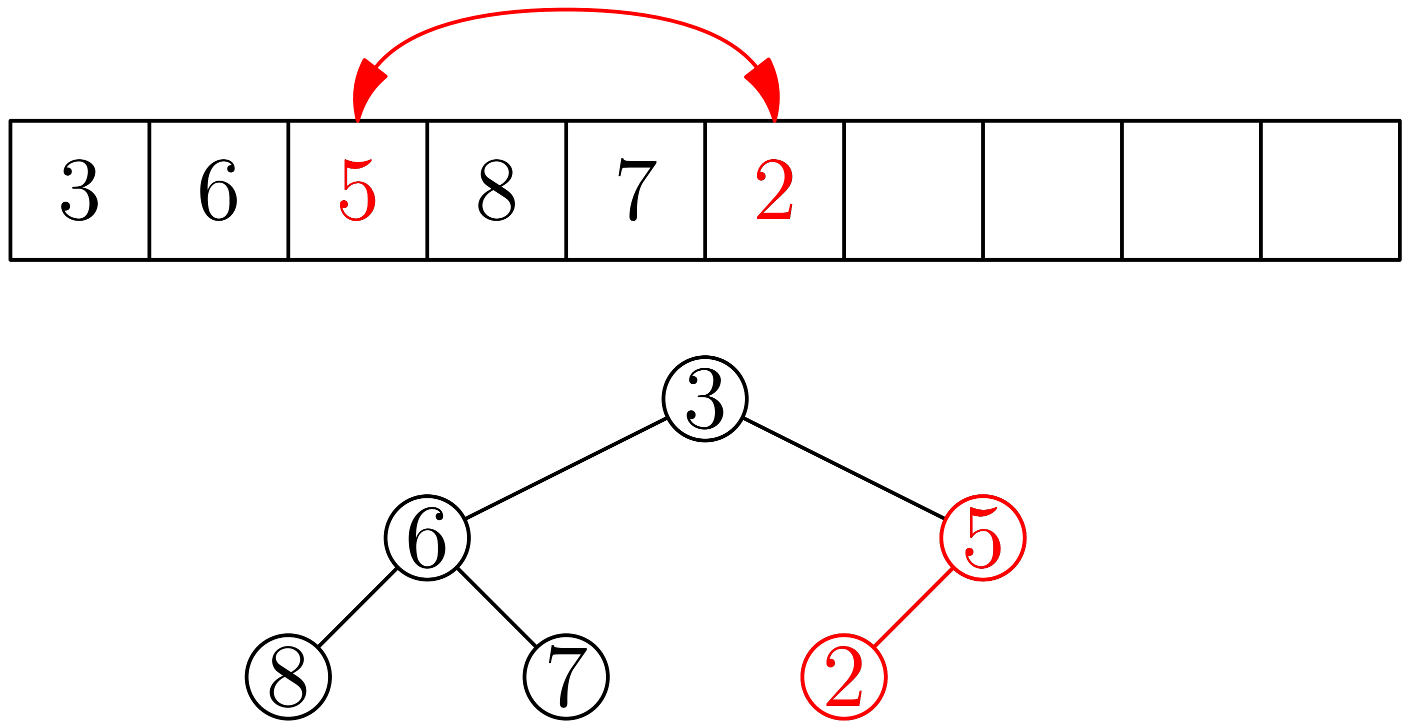
Heaps in Action

`heap.add(2)`

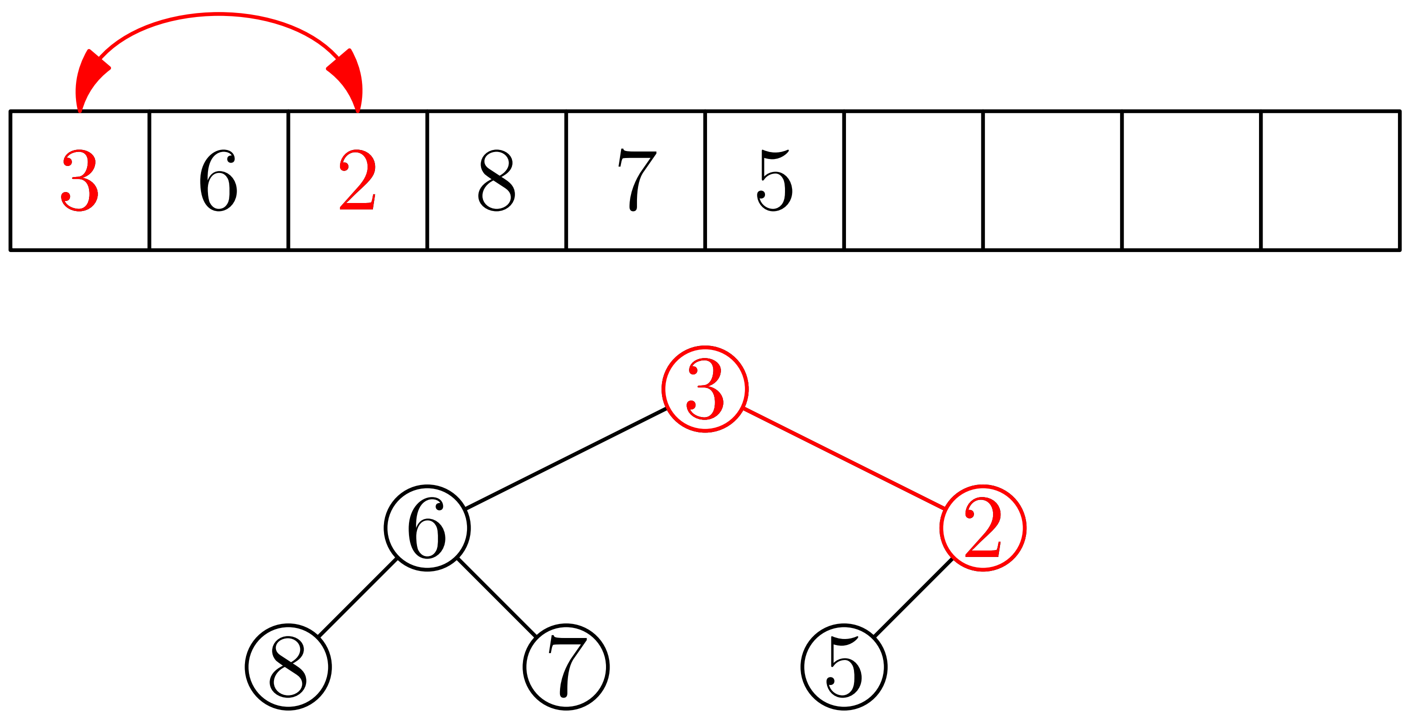
3	6	5	8	7					



Heaps in Action

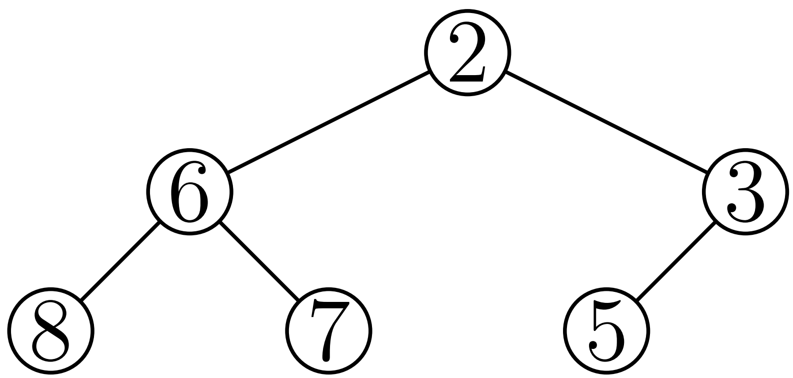


Heaps in Action



Heaps in Action

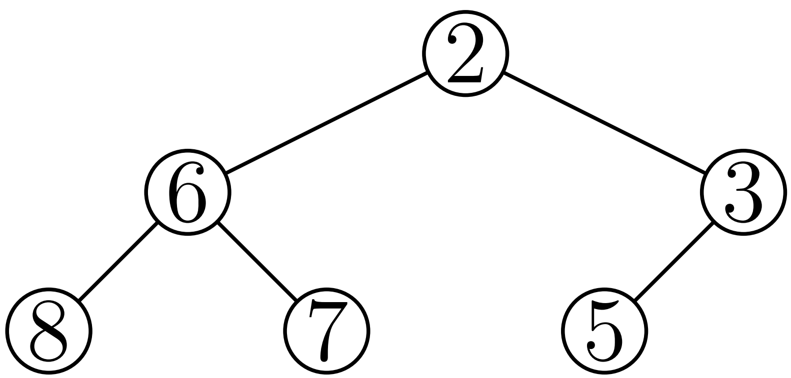
2	6	3	8	7	5				



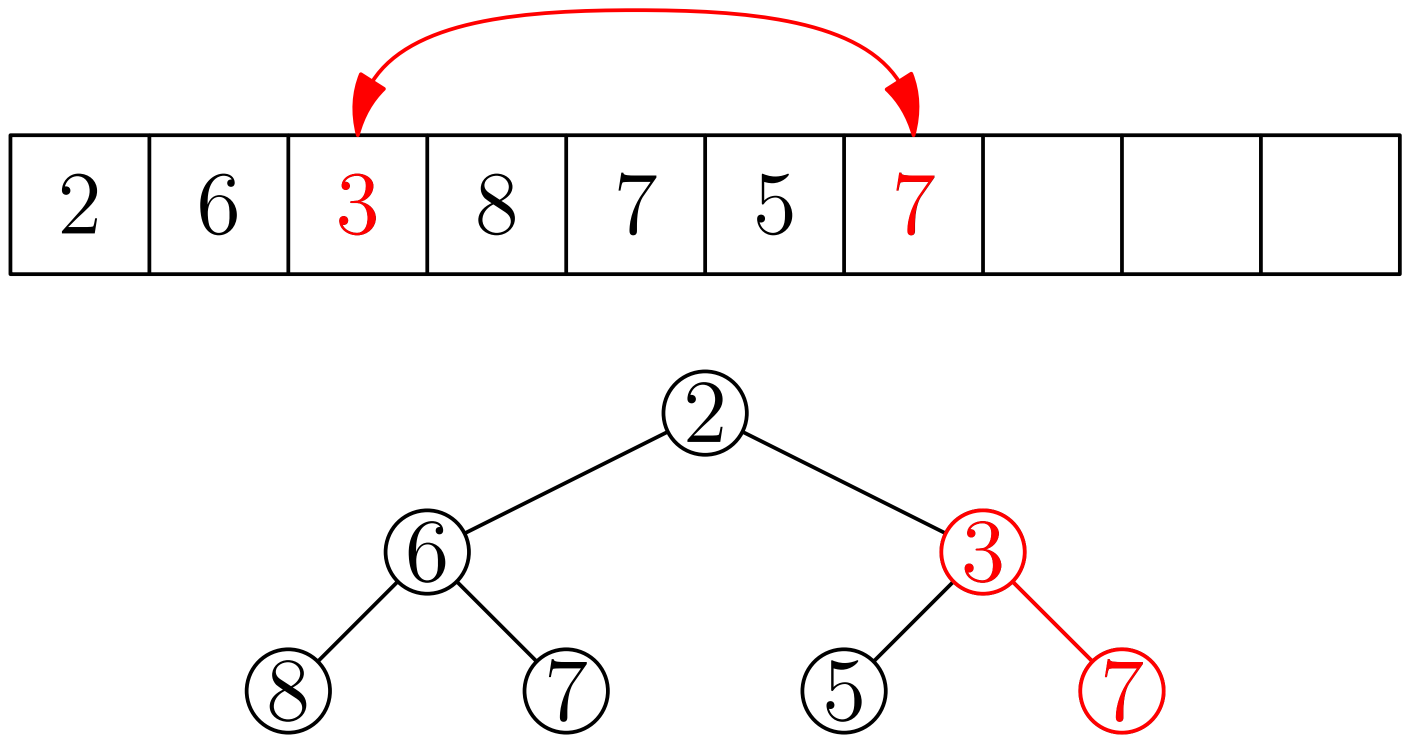
Heaps in Action

`heap.add(7)`

2	6	3	8	7	5				

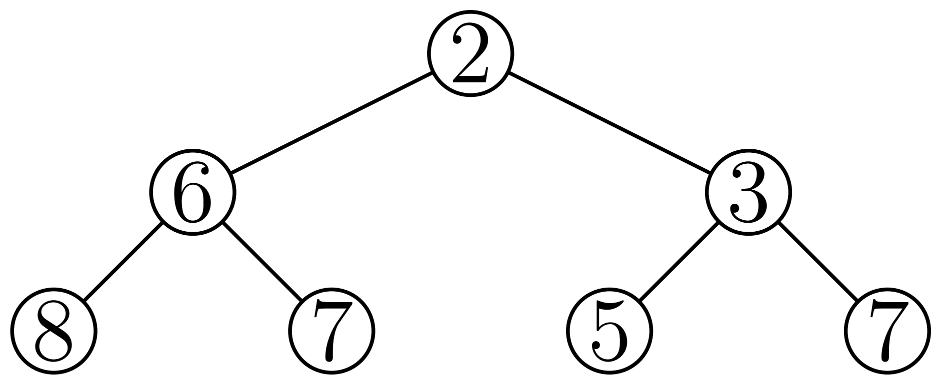


Heaps in Action



Heaps in Action

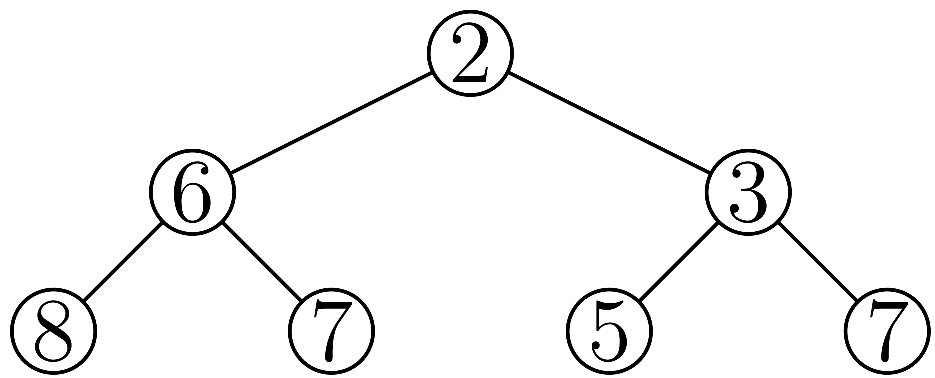
2	6	3	8	7	5	7			



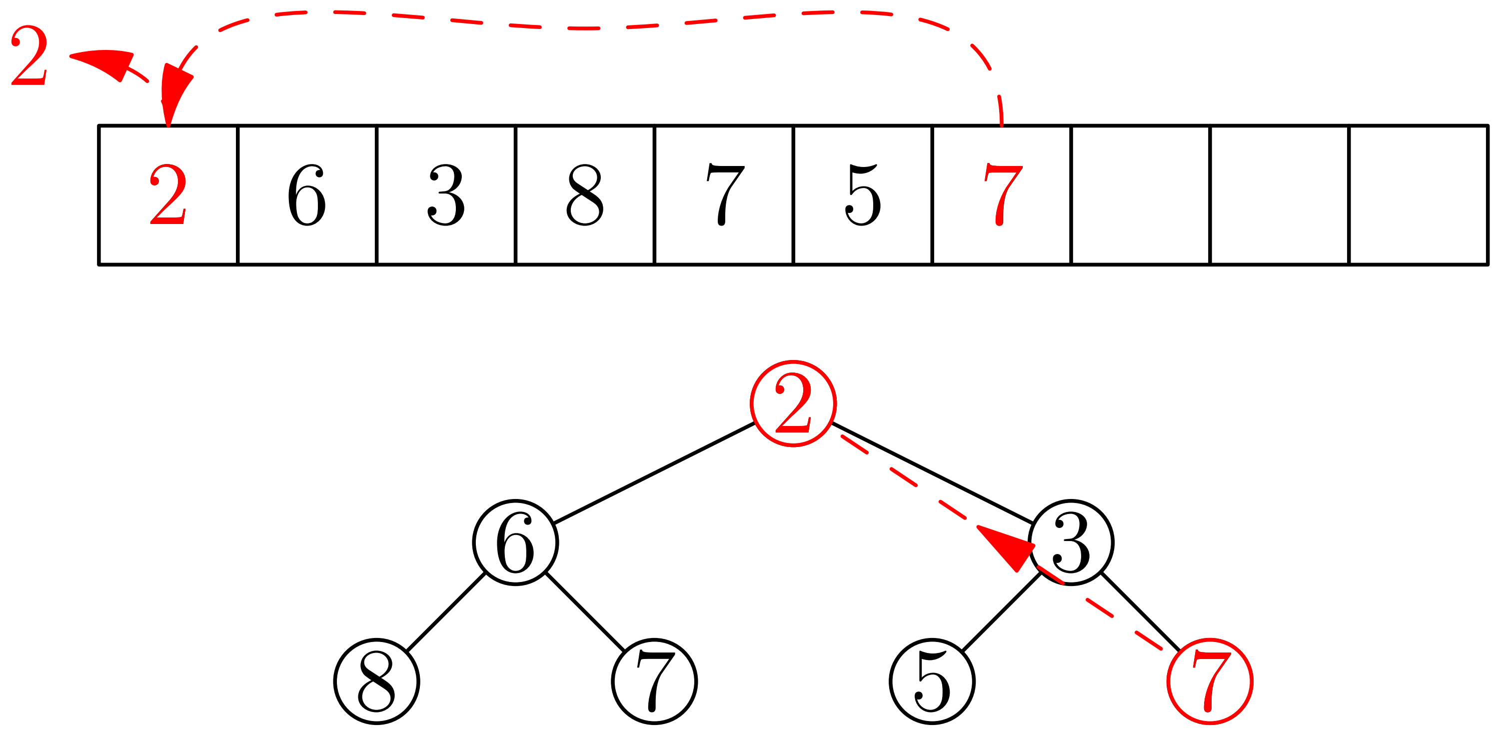
Heaps in Action

`heap.removeMin()`

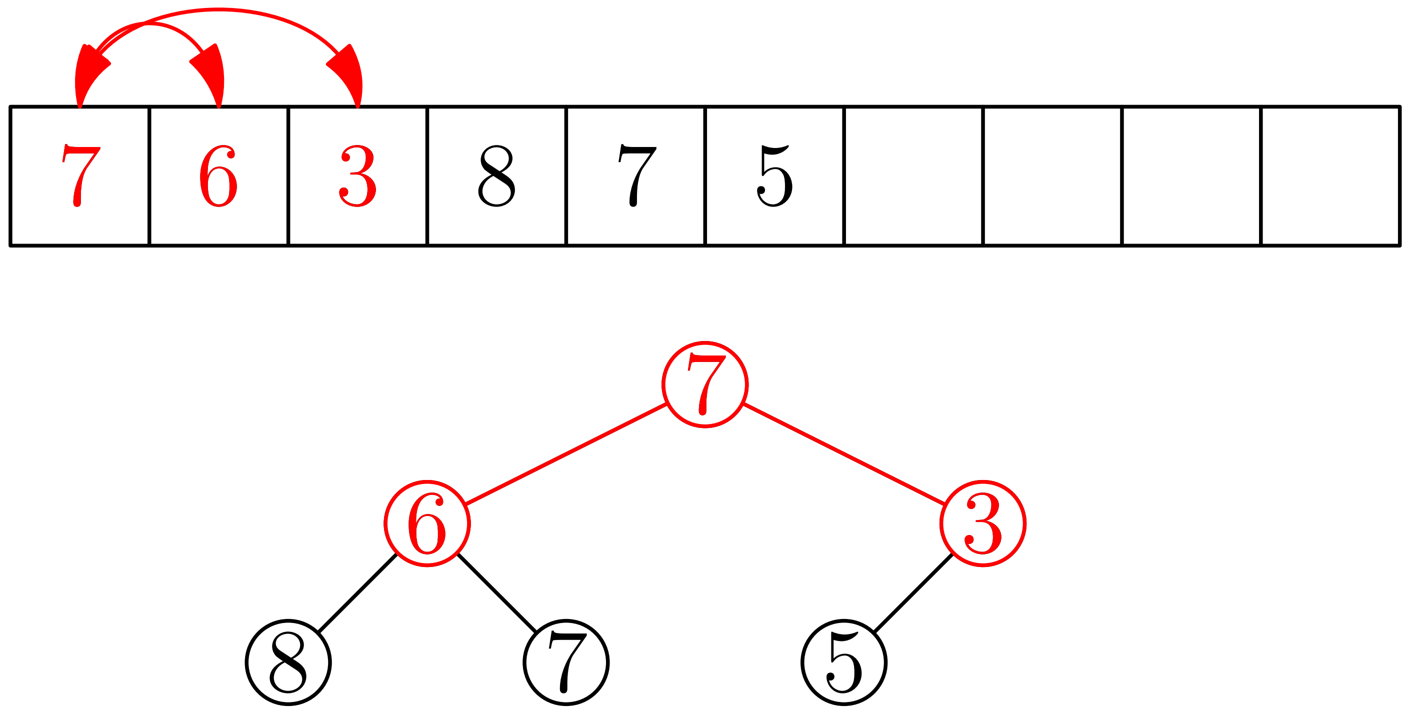
2	6	3	8	7	5	7			



Heaps in Action

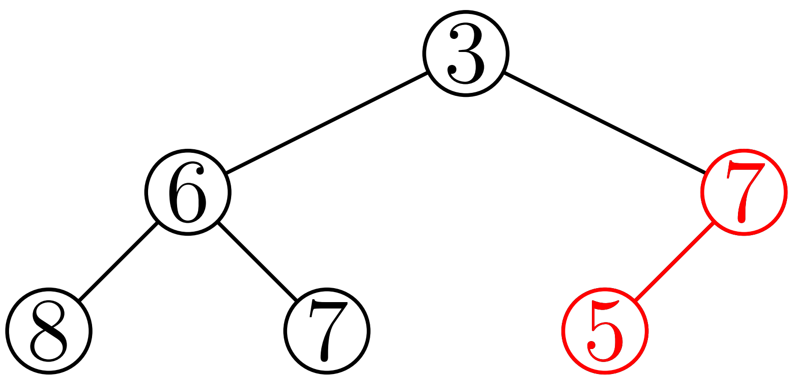


Heaps in Action

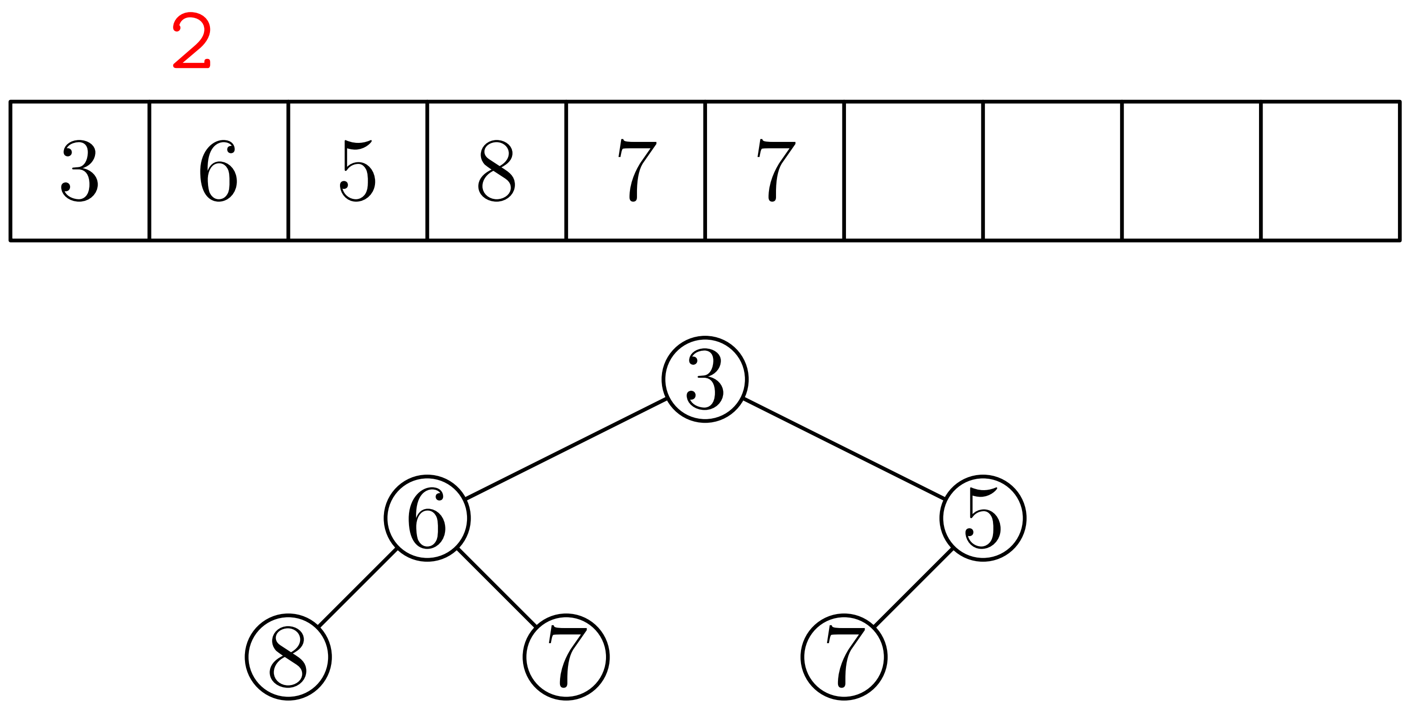


Heaps in Action

3	6	7	8	7	5				



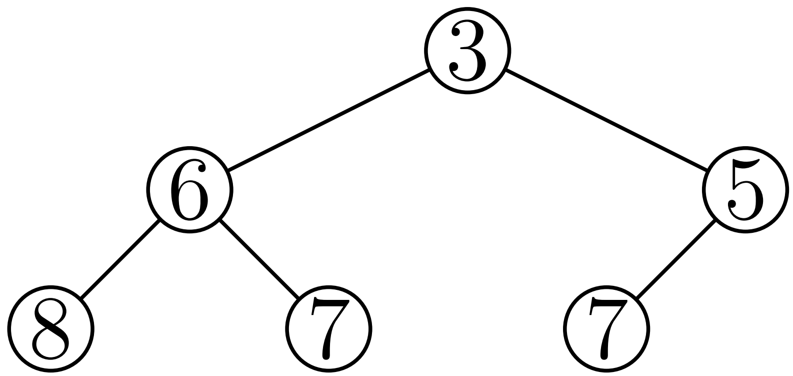
Heaps in Action



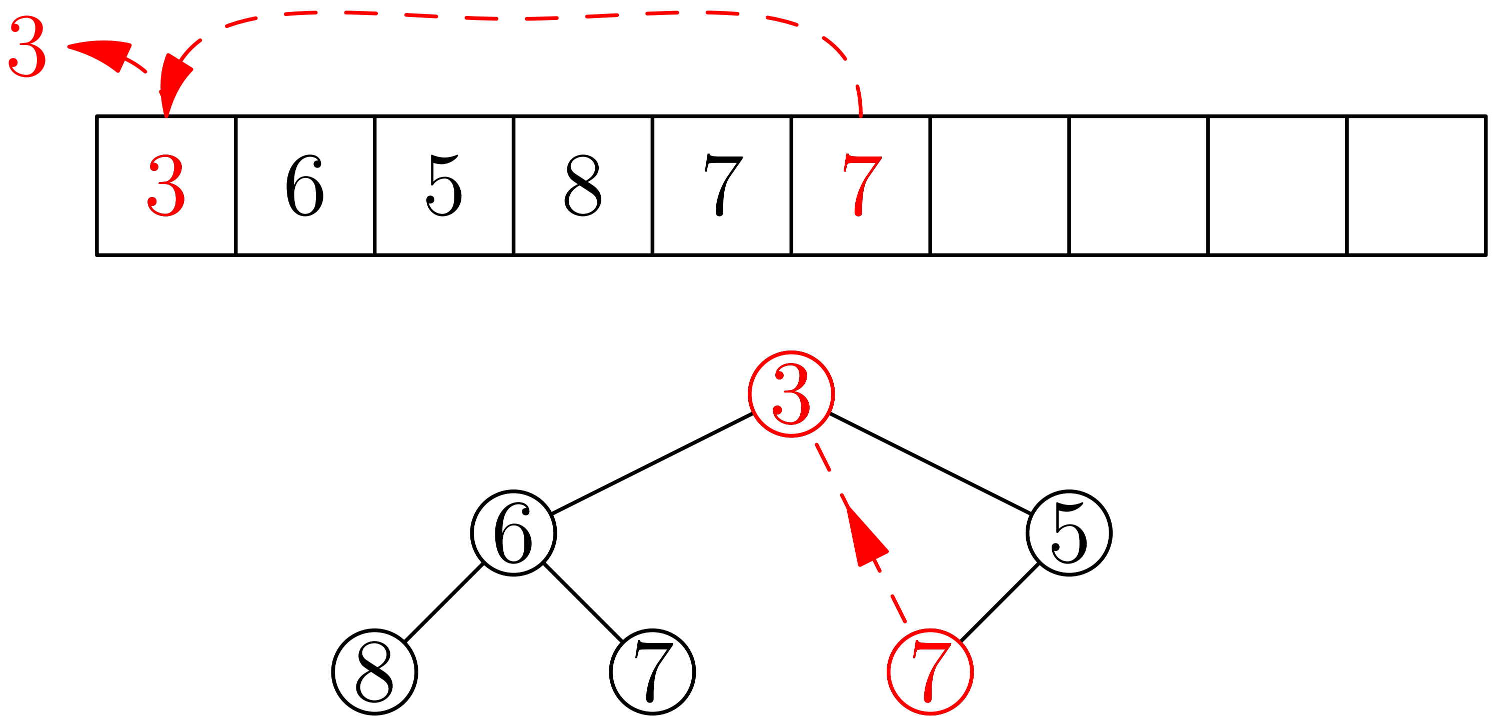
Heaps in Action

`heap.removeMin()`

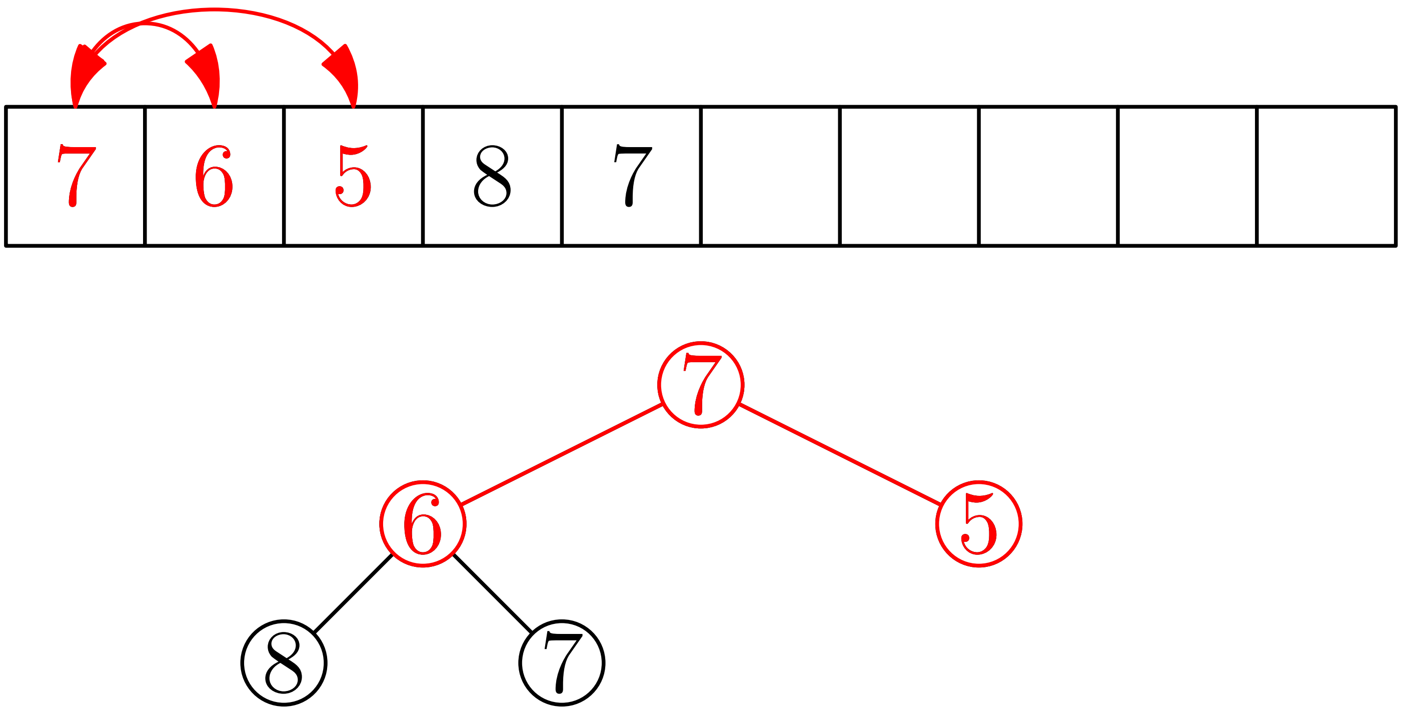
3	6	5	8	7	7				



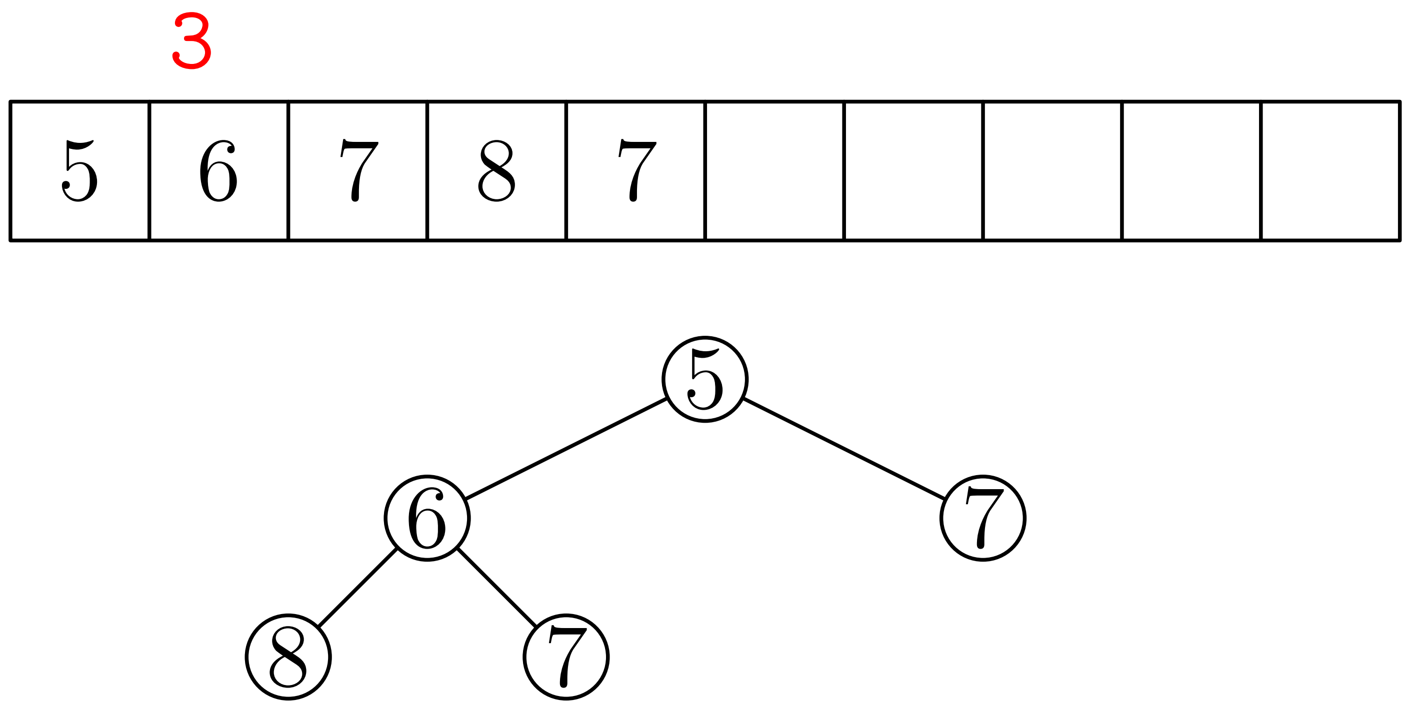
Heaps in Action



Heaps in Action



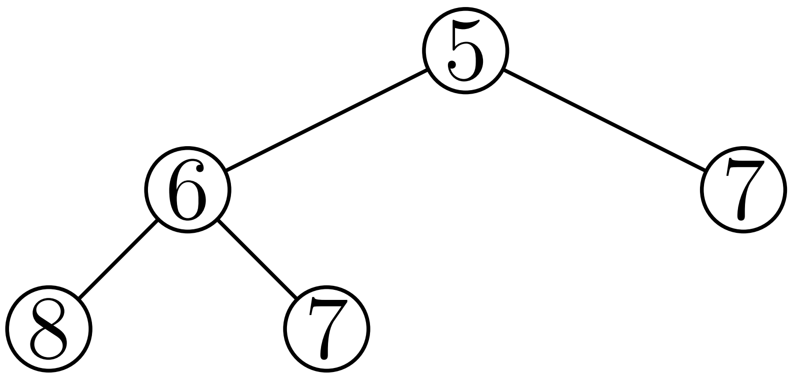
Heaps in Action



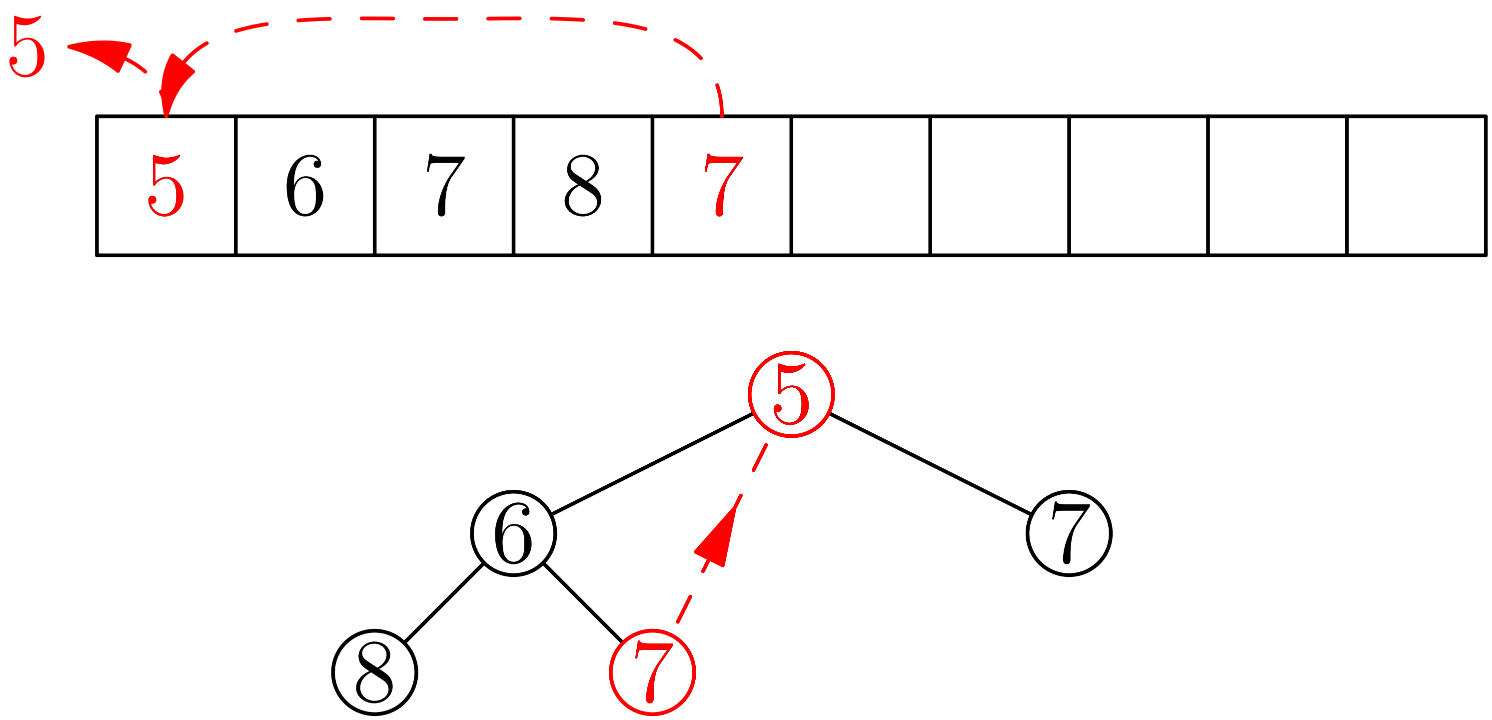
Heaps in Action

`heap.removeMin()`

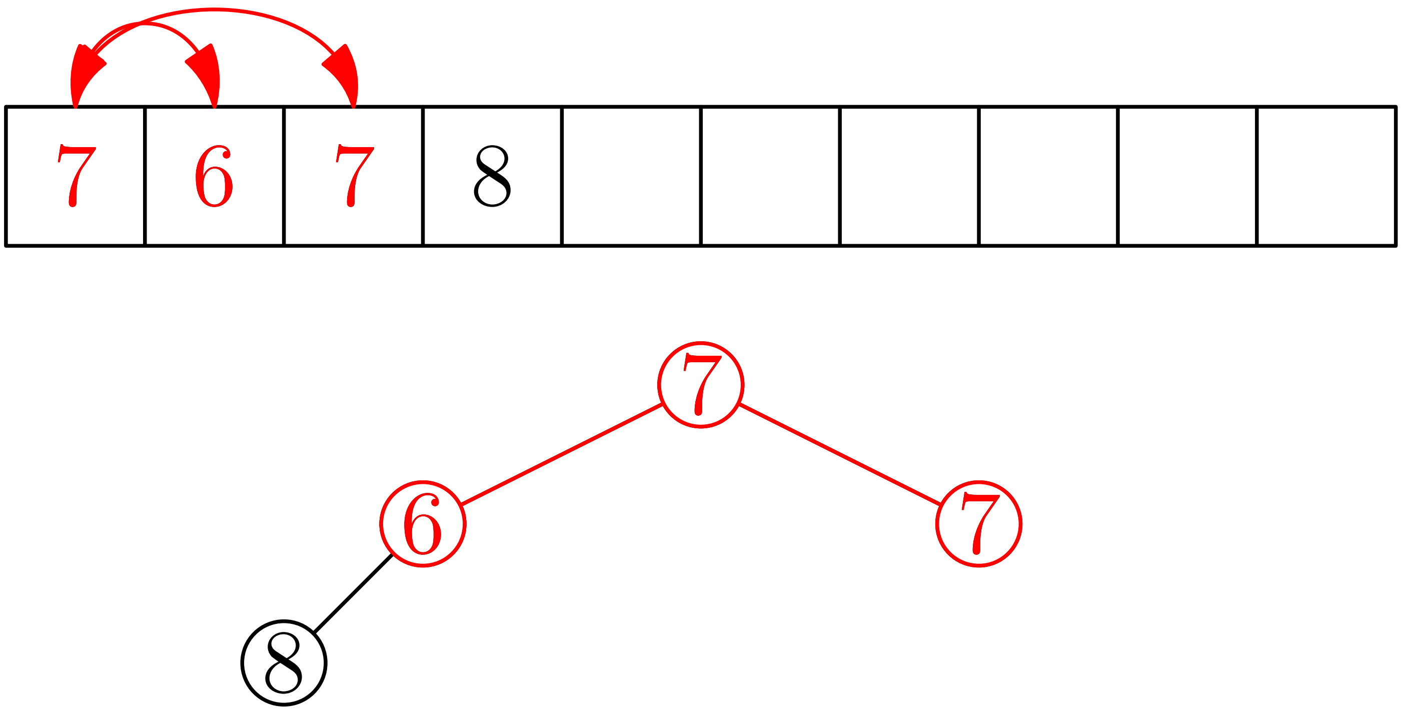
5	6	7	8	7					



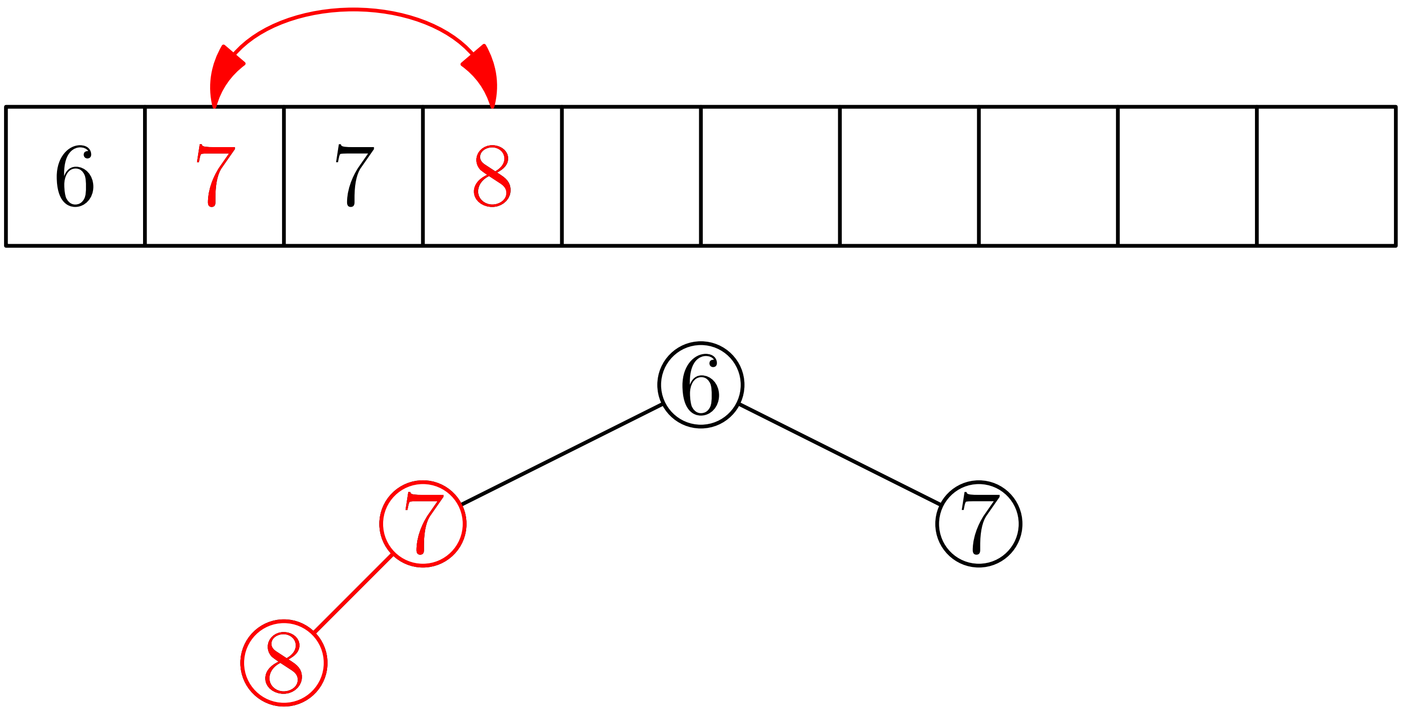
Heaps in Action



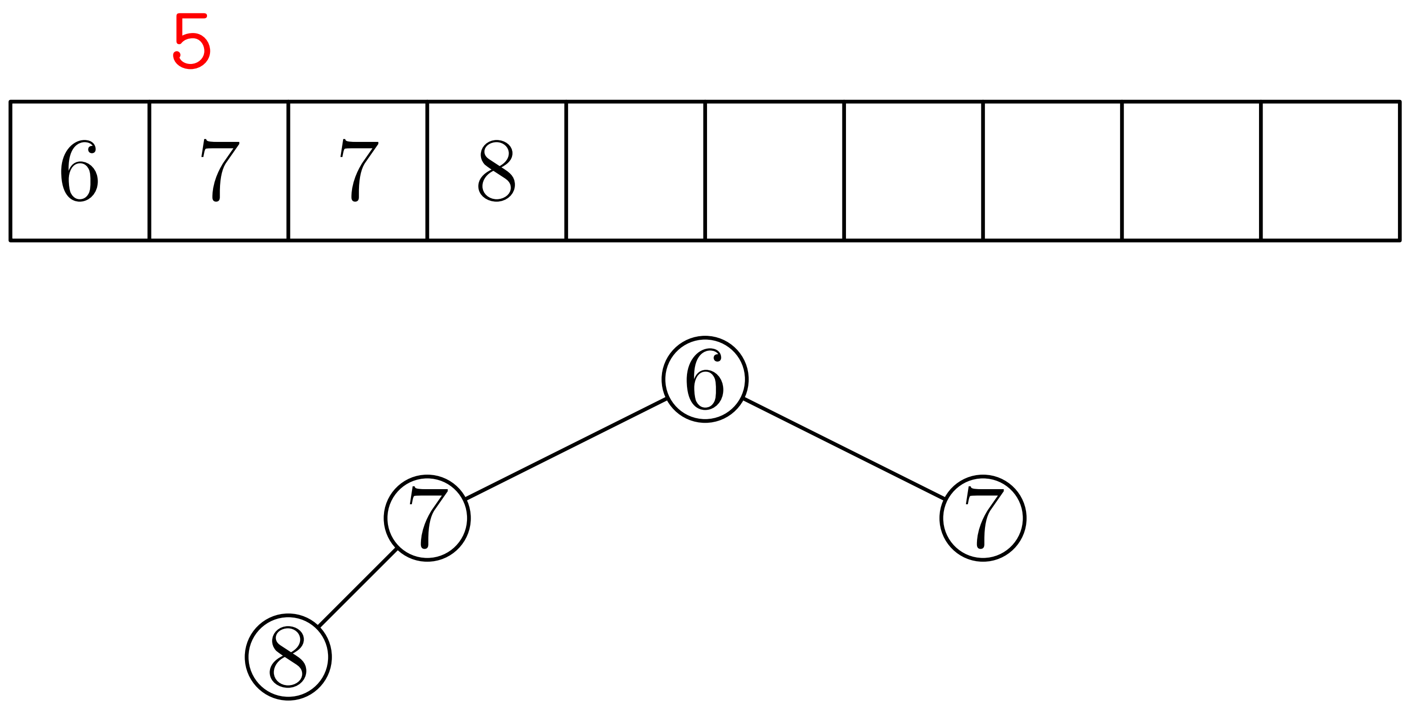
Heaps in Action



Heaps in Action



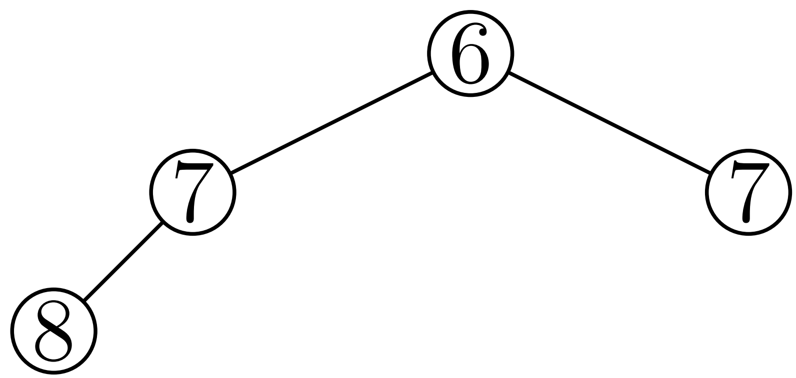
Heaps in Action



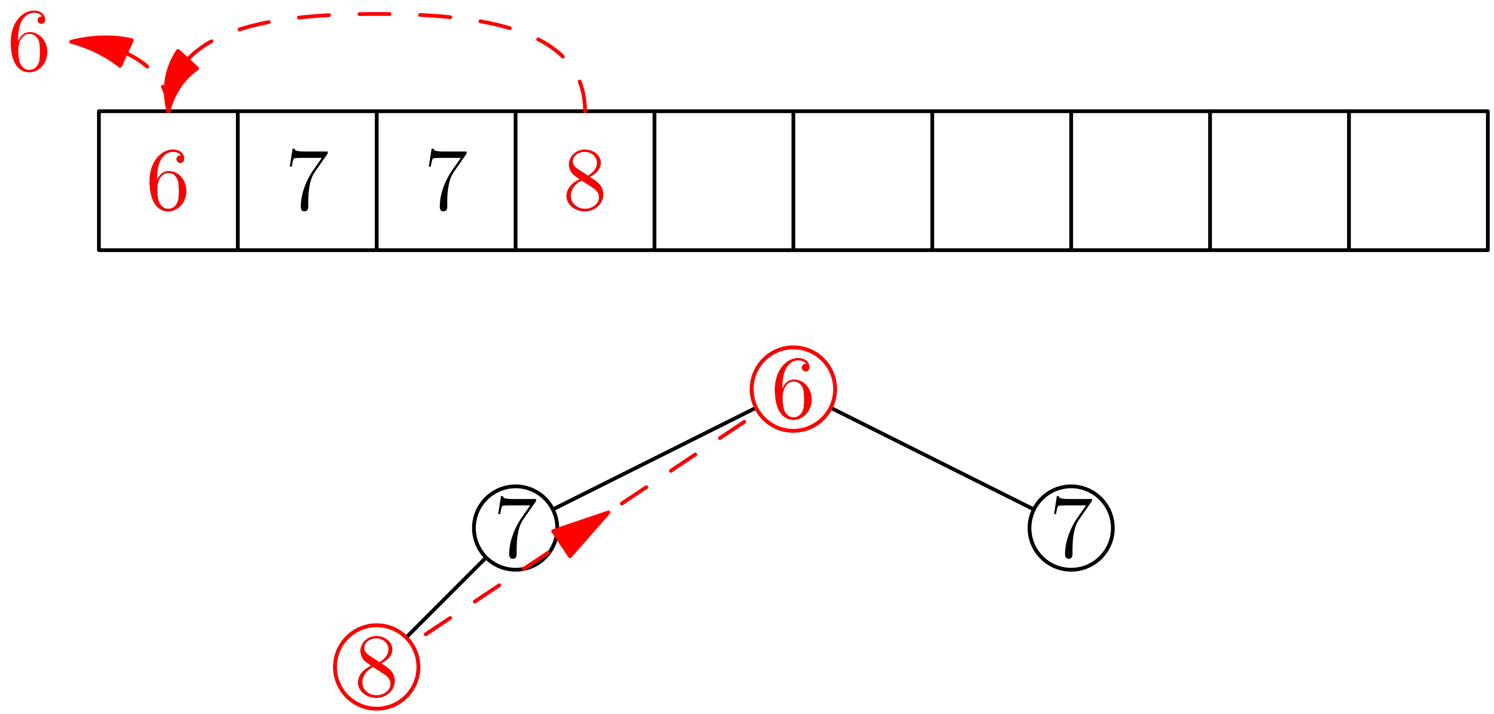
Heaps in Action

`heap.removeMin()`

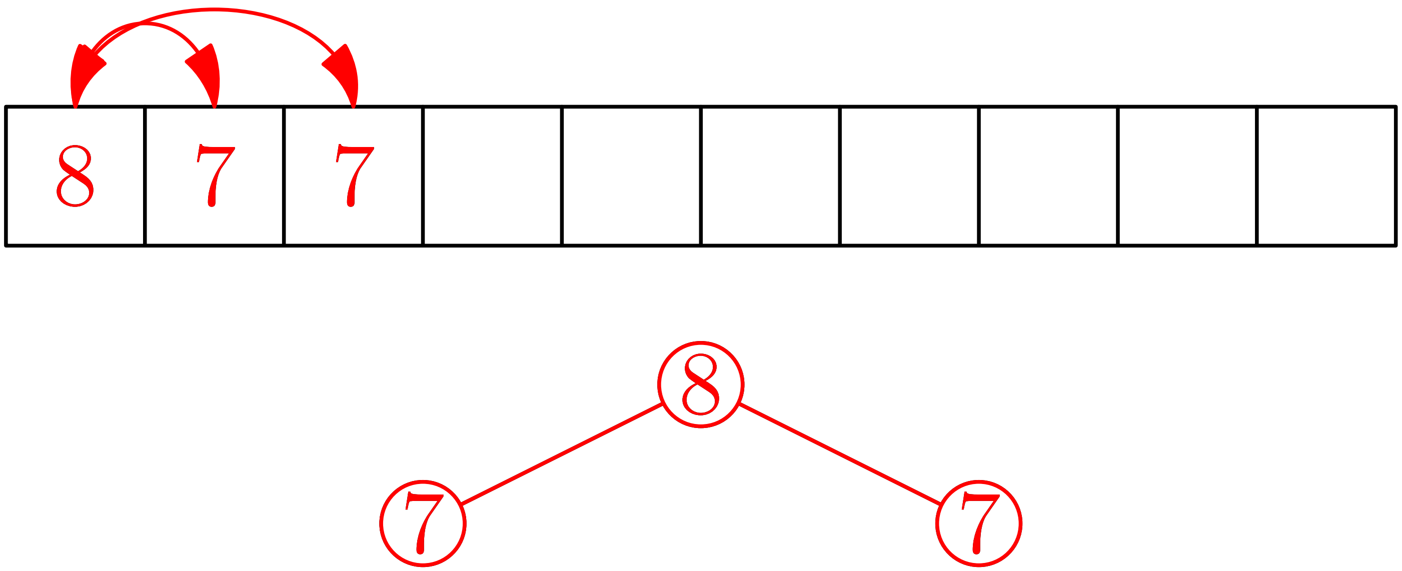
6	7	7	8						



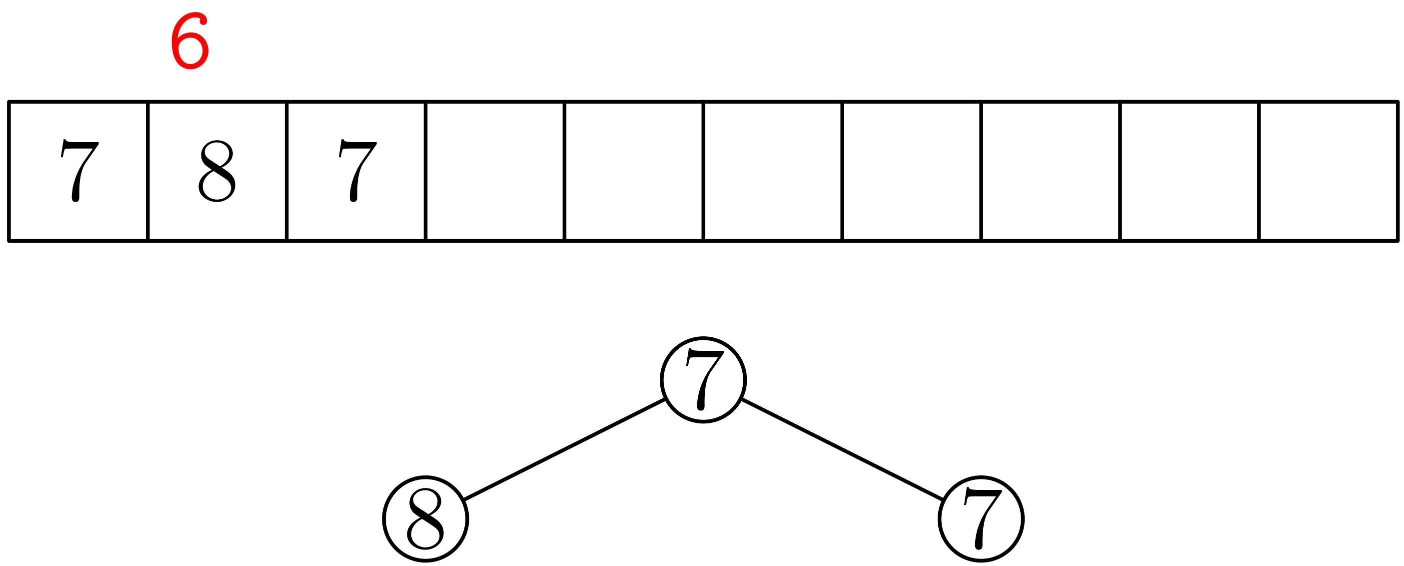
Heaps in Action



Heaps in Action



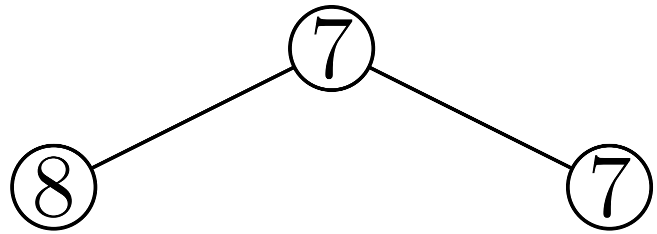
Heaps in Action



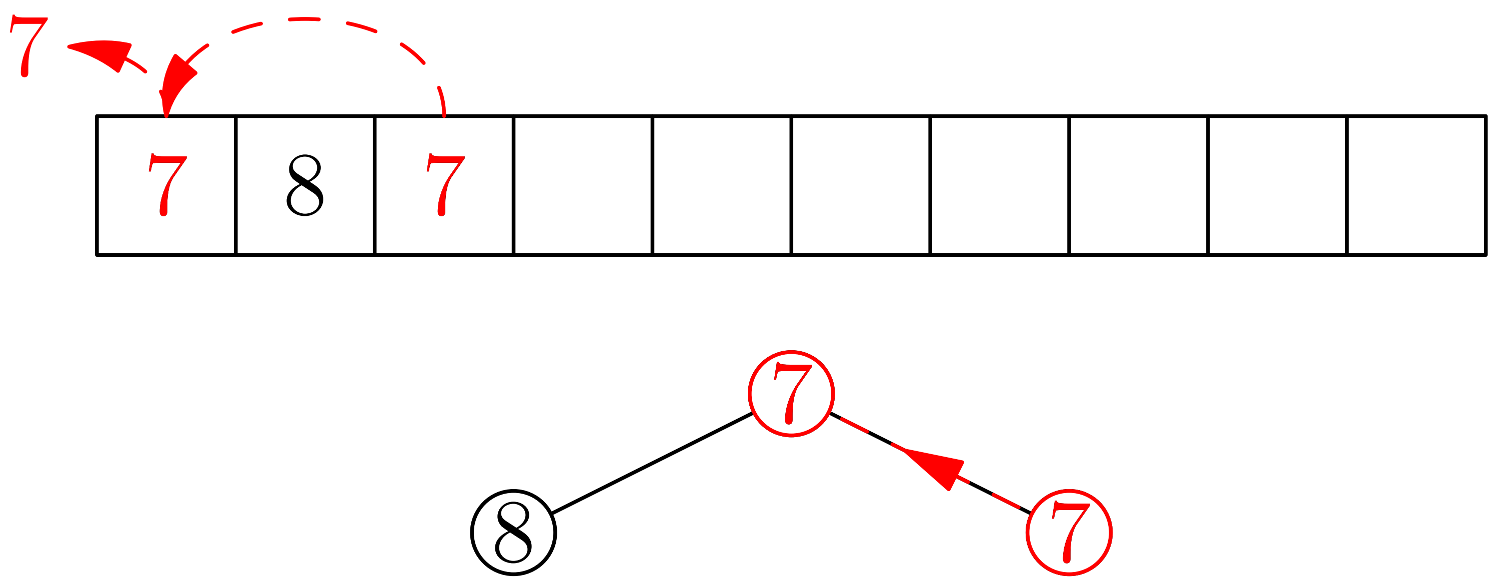
Heaps in Action

`heap.removeMin()`

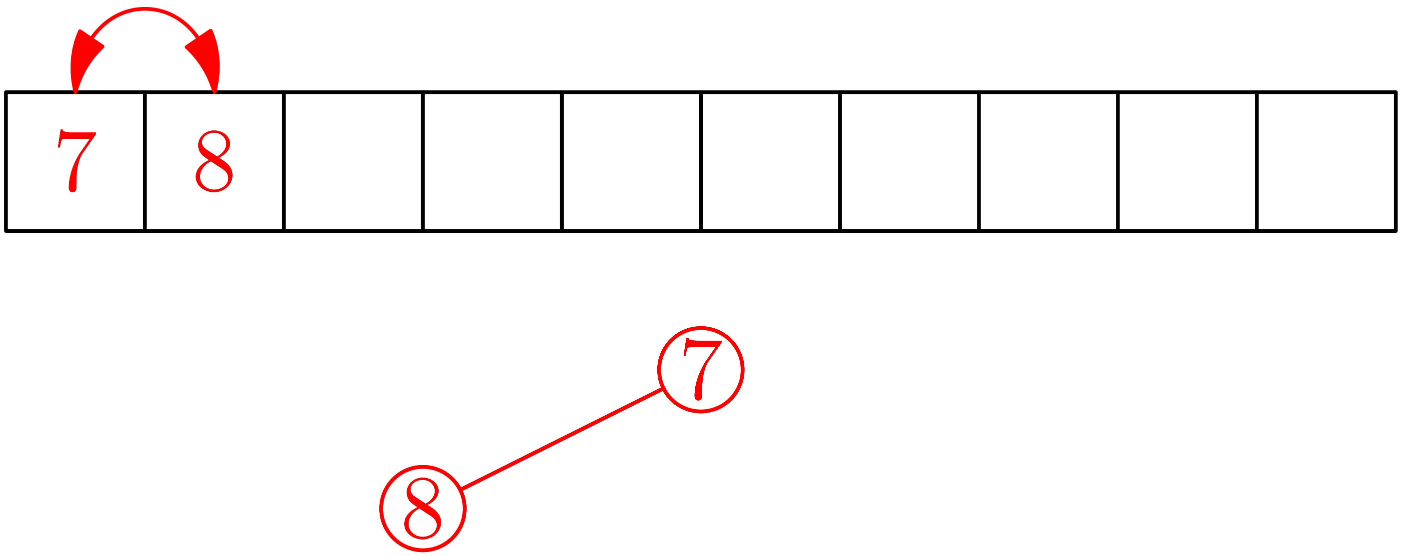
7	8	7							



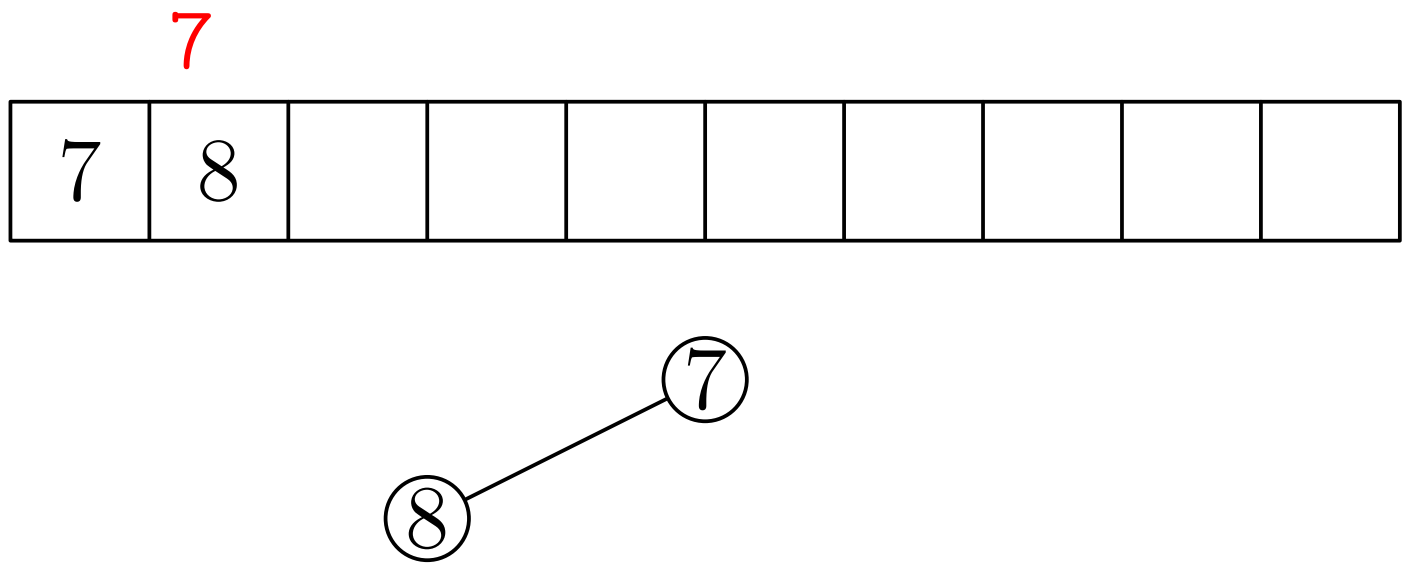
Heaps in Action



Heaps in Action



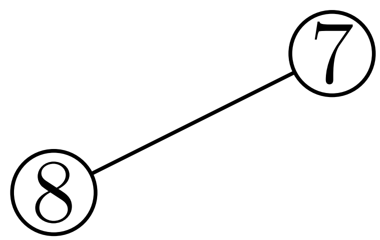
Heaps in Action



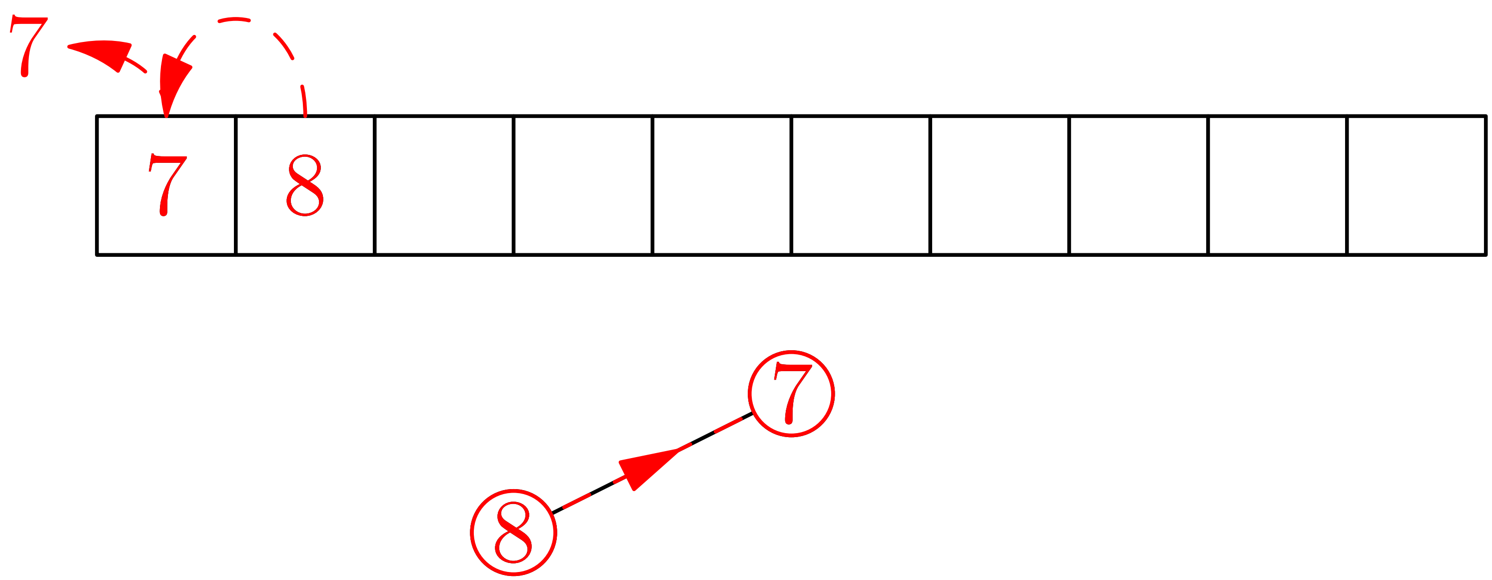
Heaps in Action

`heap.removeMin()`

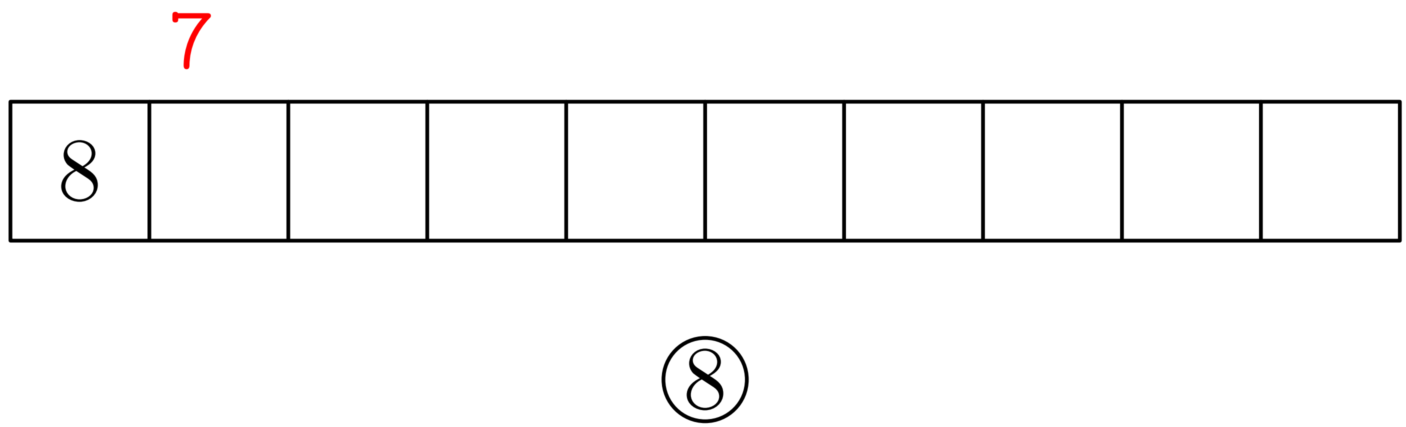
7	8								



Heaps in Action

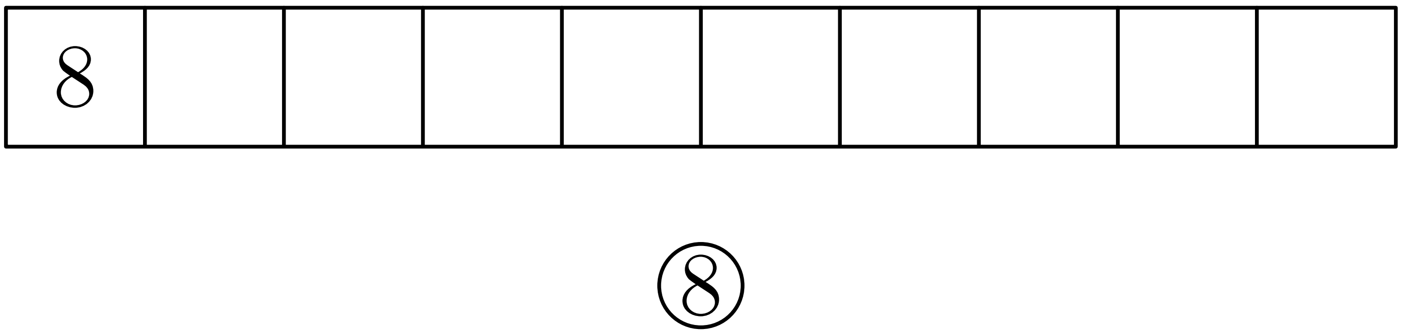


Heaps in Action

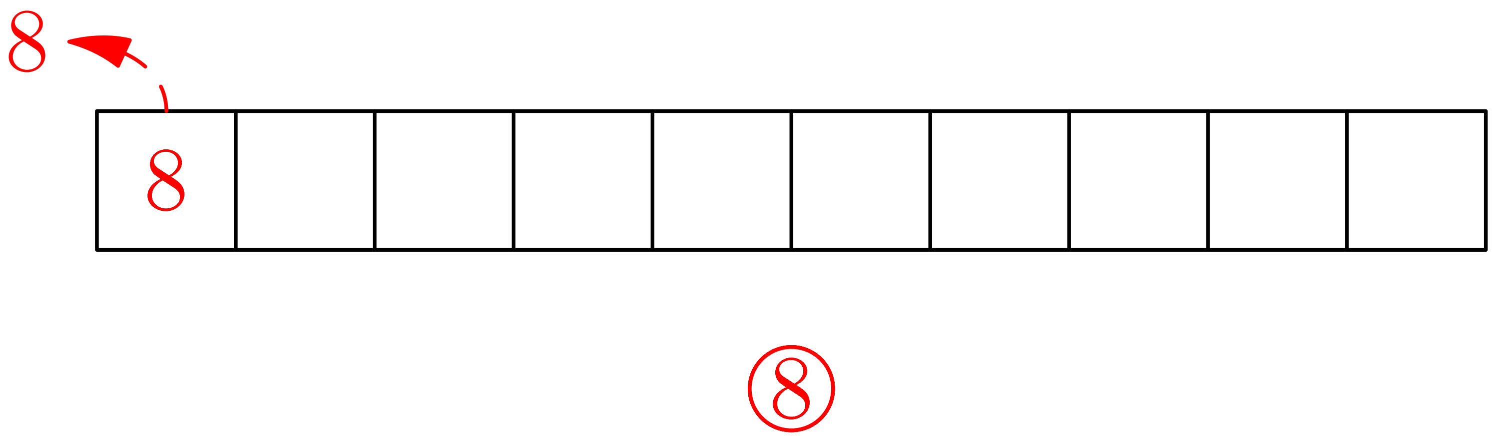


Heaps in Action

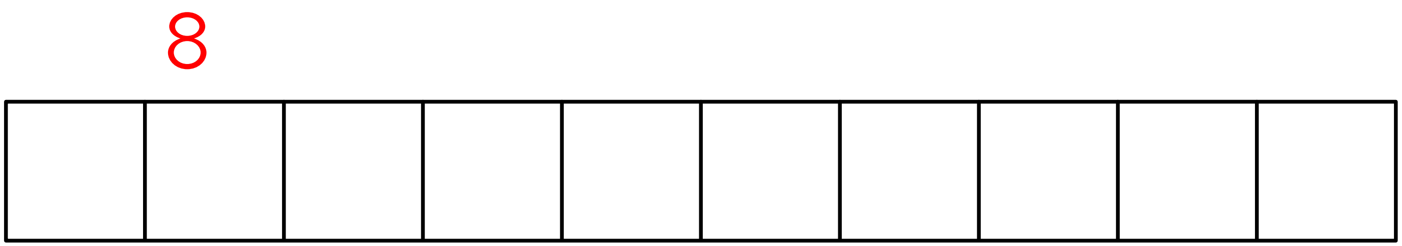
heap.removeMin()



Heaps in Action



Heaps in Action



Time Complexity of Heaps

- The two important operations are `add` and `removeMin`
- These work by percolating an element up the tree, respectively by percolating an element down the tree
- The number of elementary operations in `add`/`removeMin` depends on the depth of the tree, which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $\Theta(\log(n))$ in the worst case
- Except `add` could also require resizing the array, but the amortised cost of this is low

Back to Priority Queues

- We implemented a priority queue using a heap earlier (`HeapPQ<T>`)
- To make a priority queue we use a `PriorityTask` class for the queue elements:

```
Queue<PriorityTask> pq = new HeapPQ<PriorityTask>();

pq.add(new PriorityTask(stuff, priority));
```

- where

```
class public PriorityTask implements Comparable<PriorityTask> {
  private Stuff stuff;
  private int priority;

  public int compareTo(PriorityTask rhs) {
    return priority-rhs.priority;
  }
⋮
```

Outline

1. Heaps
2. Priority Queues
   - Array Implementation
3. **Heap Sort**

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```java
public static <T> void sort(List<T> aList)
{
    PQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
        aHeap.add(element);

    aList.clear();
    while(aHeap.size() > 0)
        aList.add(aHeap.removeMin());
}
```

- Note that this is not an in-place sort algorithm – it uses $\Theta(n)$ additional memory!
- The standard Heap Sort algorithm sorts in place.

Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**

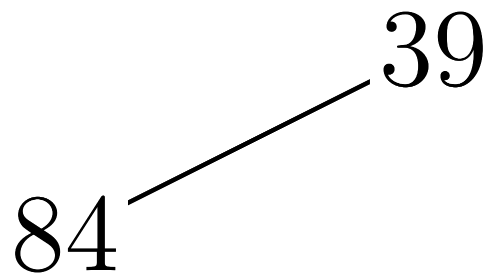
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**

84

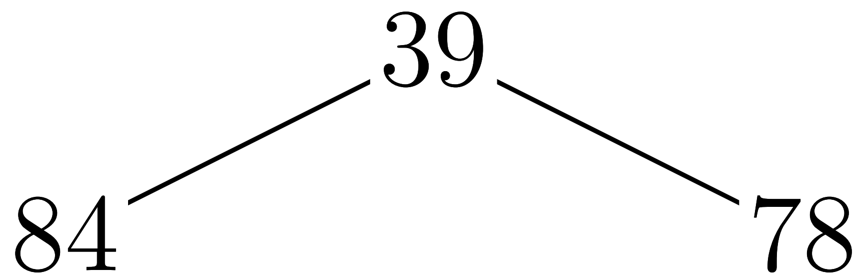
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



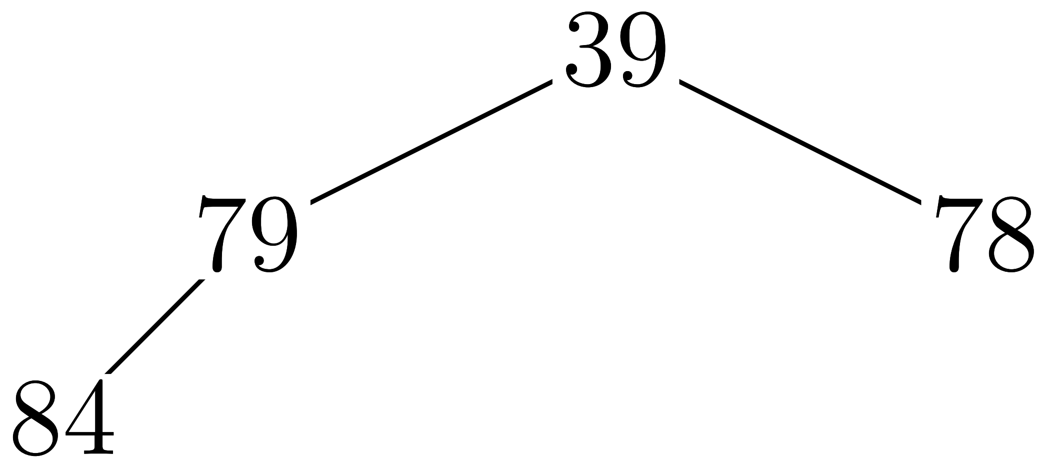
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



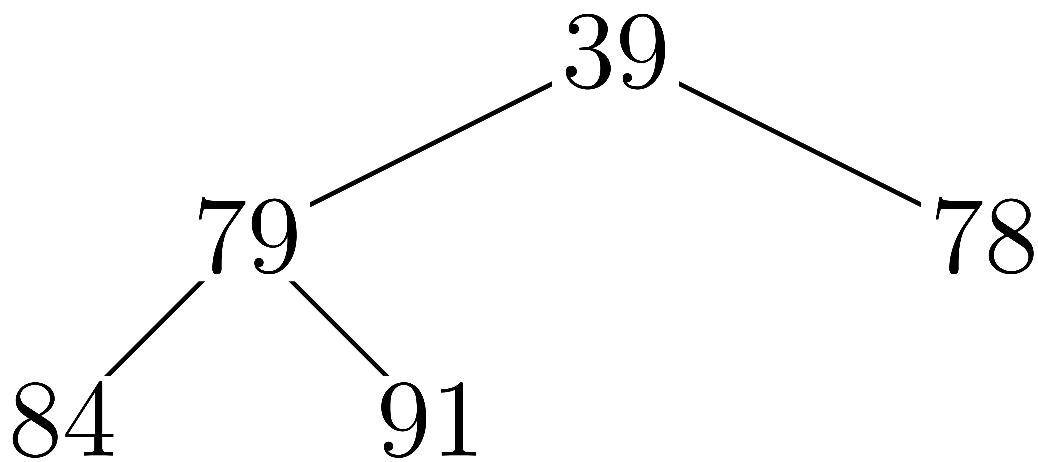
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



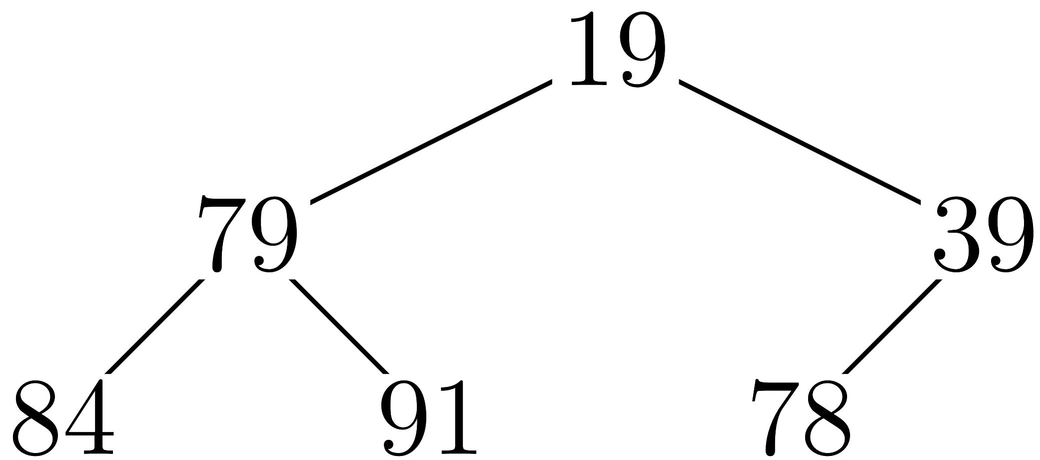
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



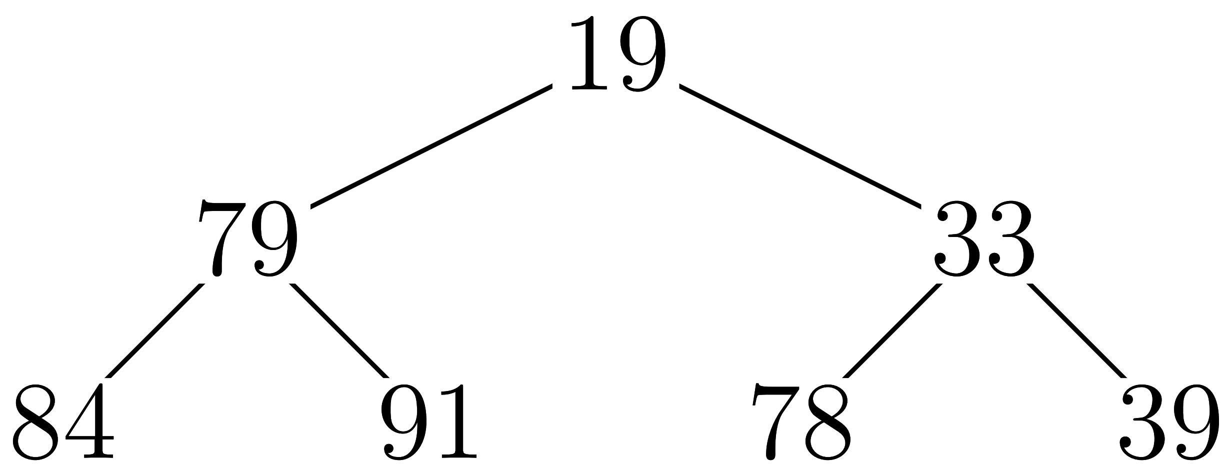
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



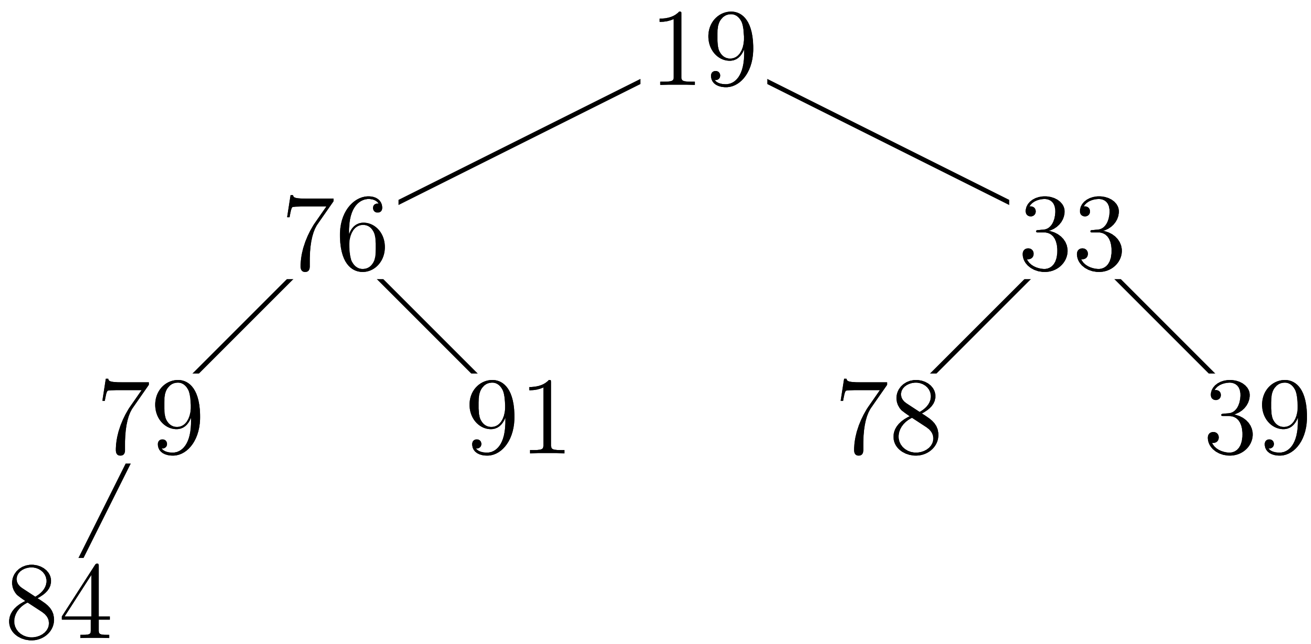
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



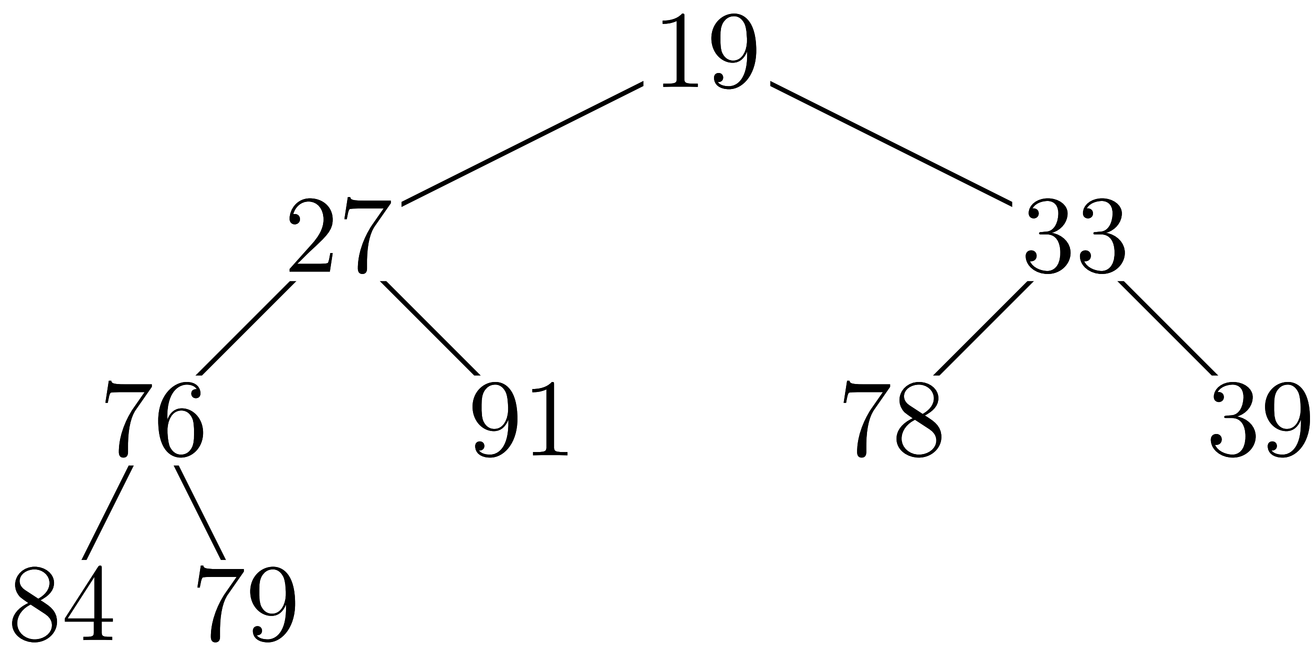
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



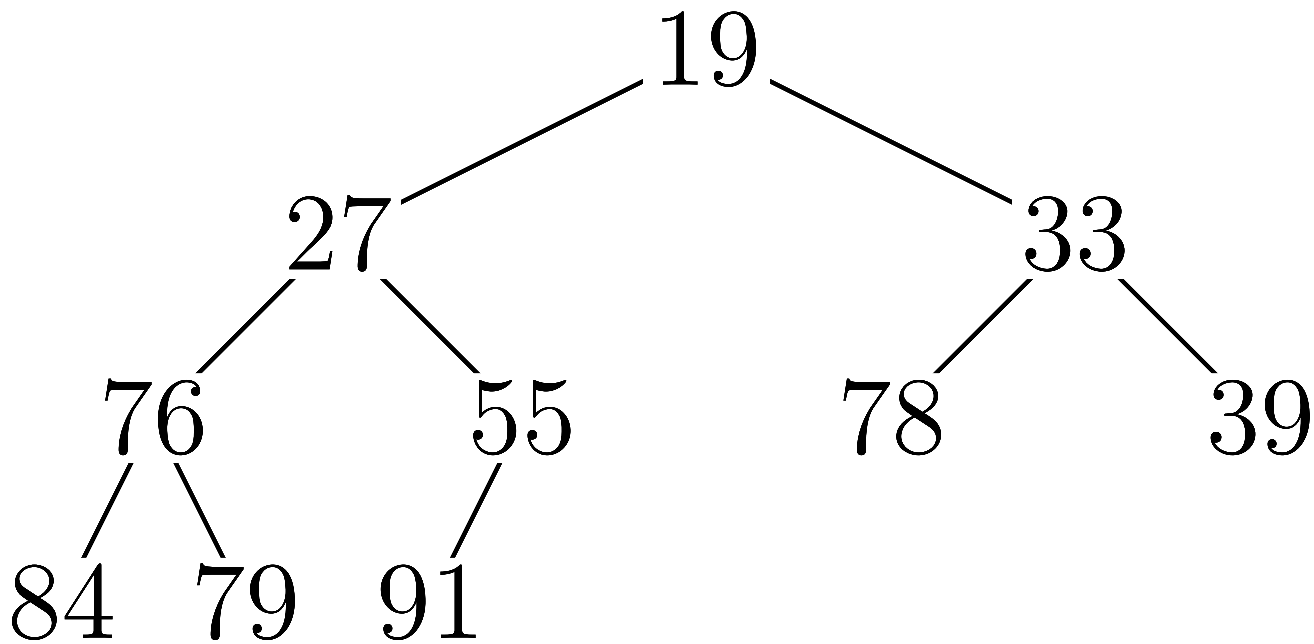
Example of Heap Sort

84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**

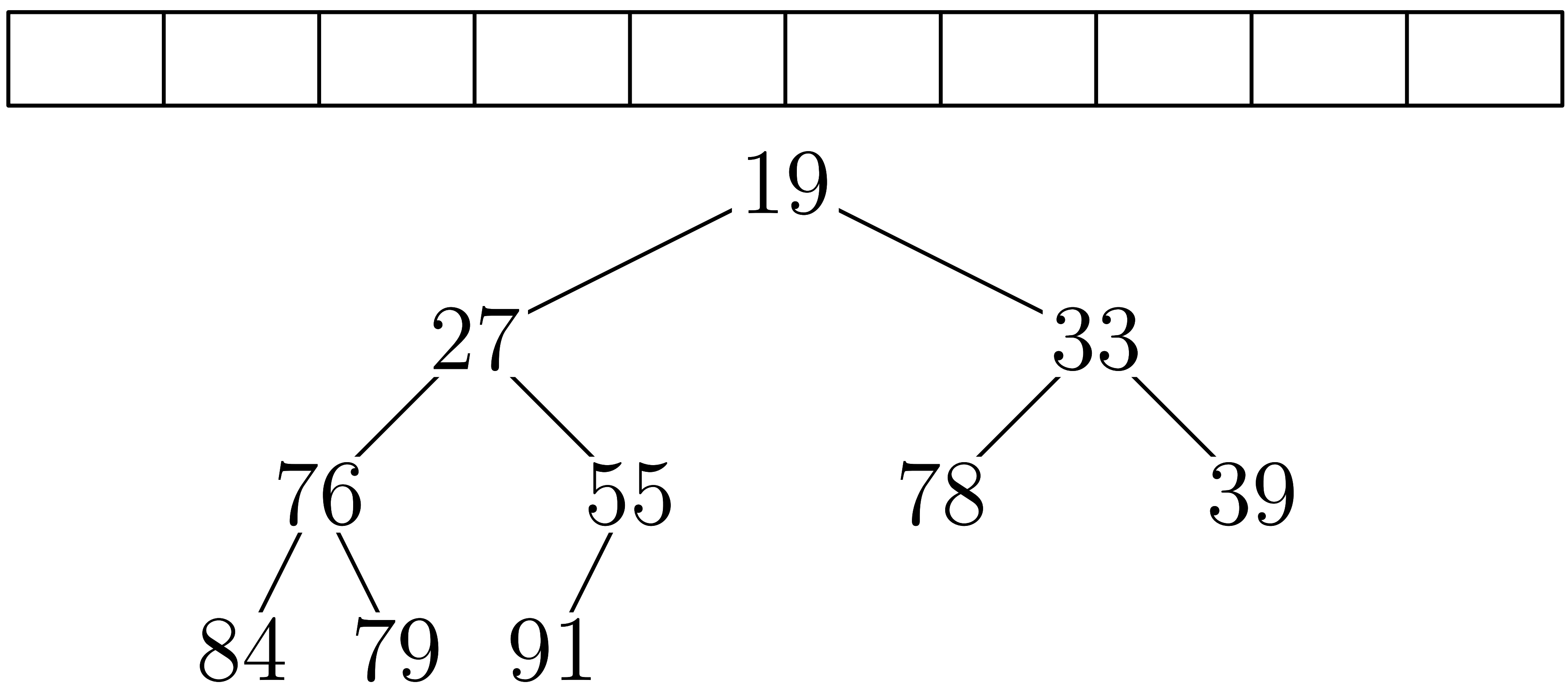


Example of Heap Sort

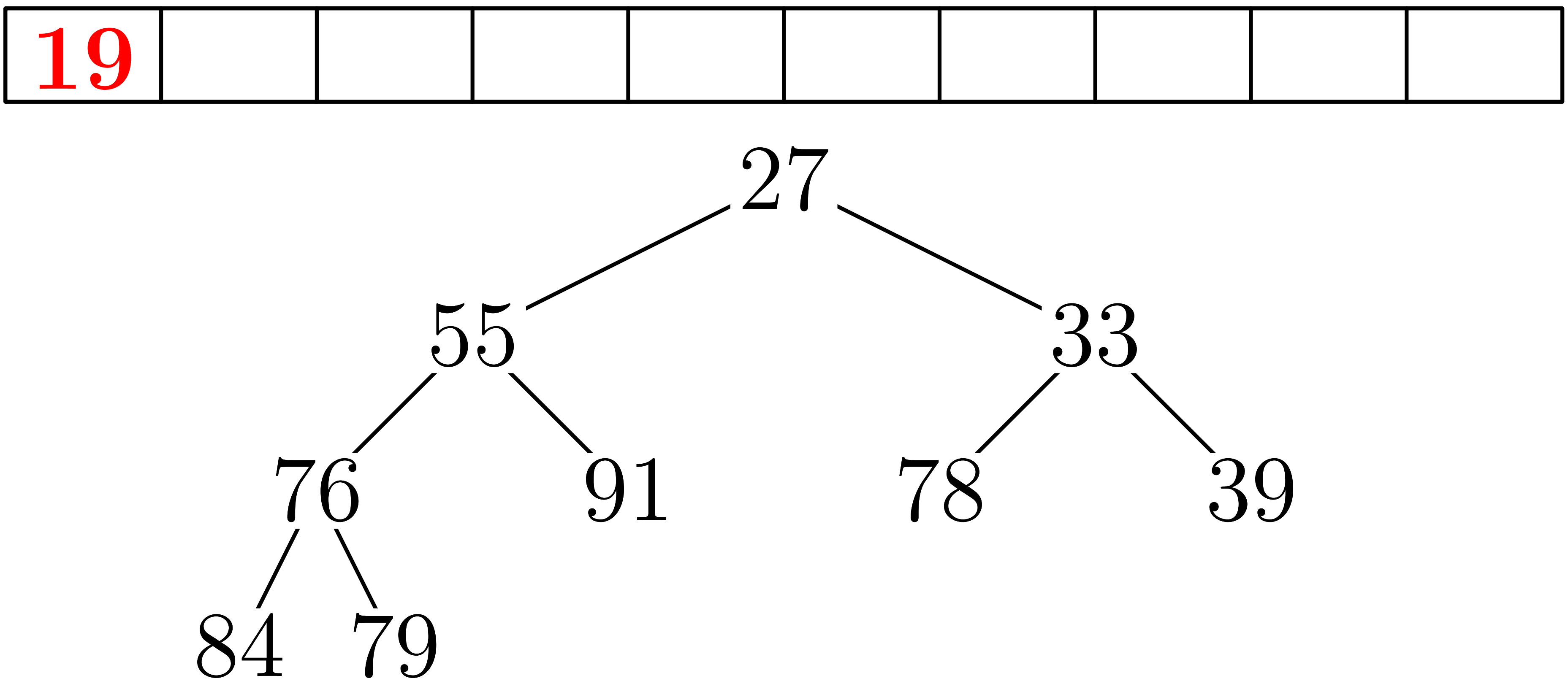
84	**39**	**78**	**79**	**91**	**19**	**33**	**76**	**27**	**55**



Example of Heap Sort

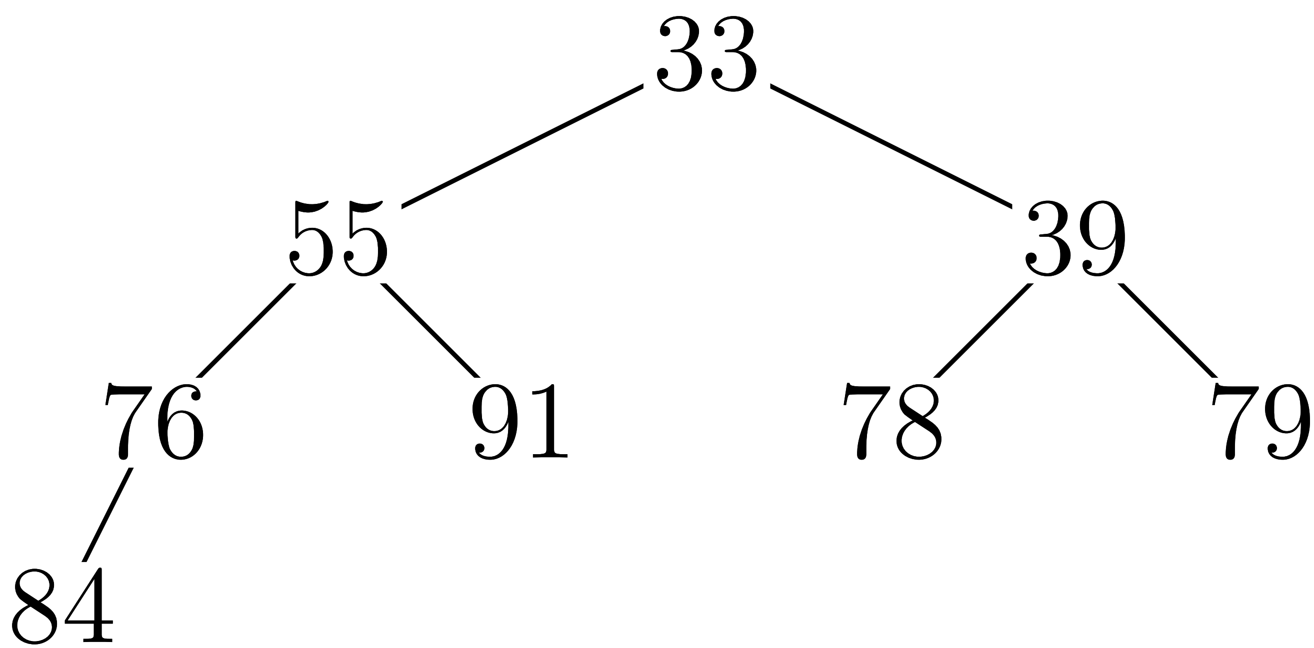


Example of Heap Sort



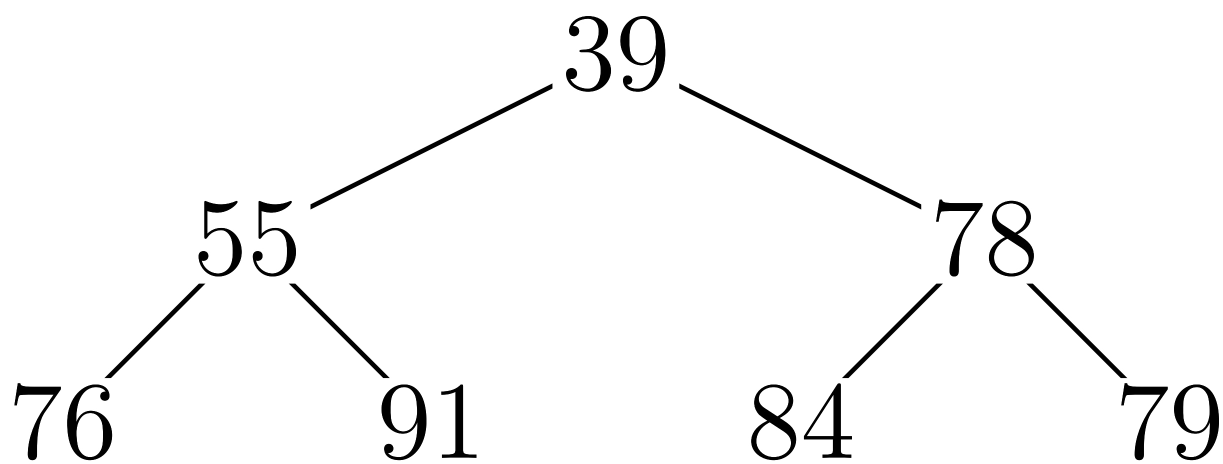
Example of Heap Sort

19	**27**								



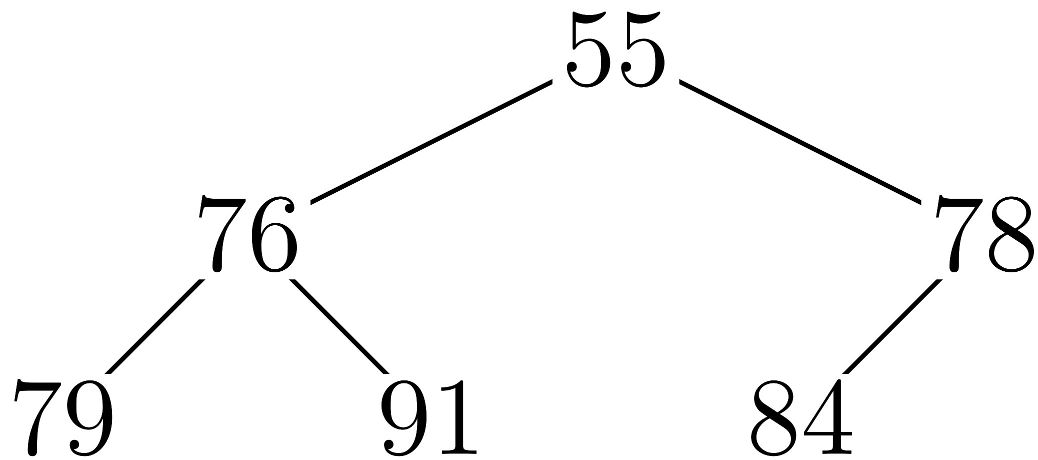
Example of Heap Sort

19	**27**	**33**							



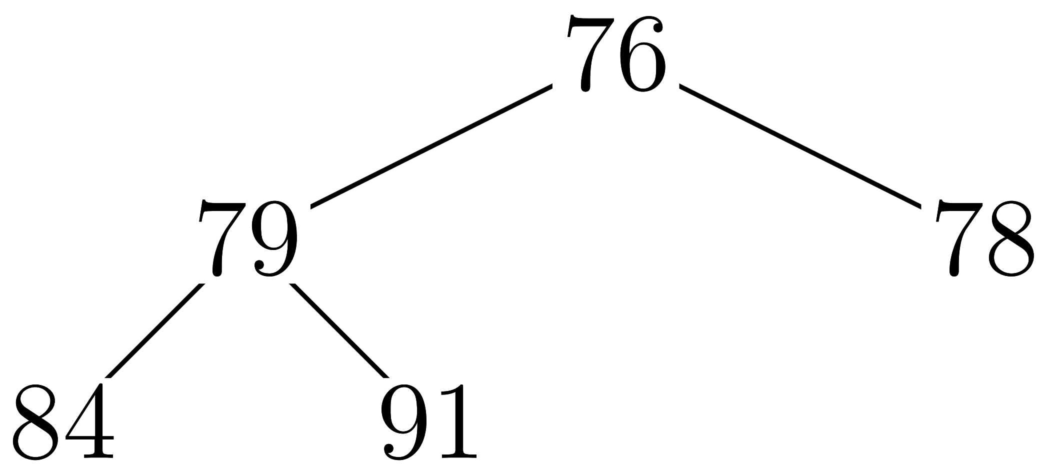
Example of Heap Sort

19	**27**	**33**	**39**						



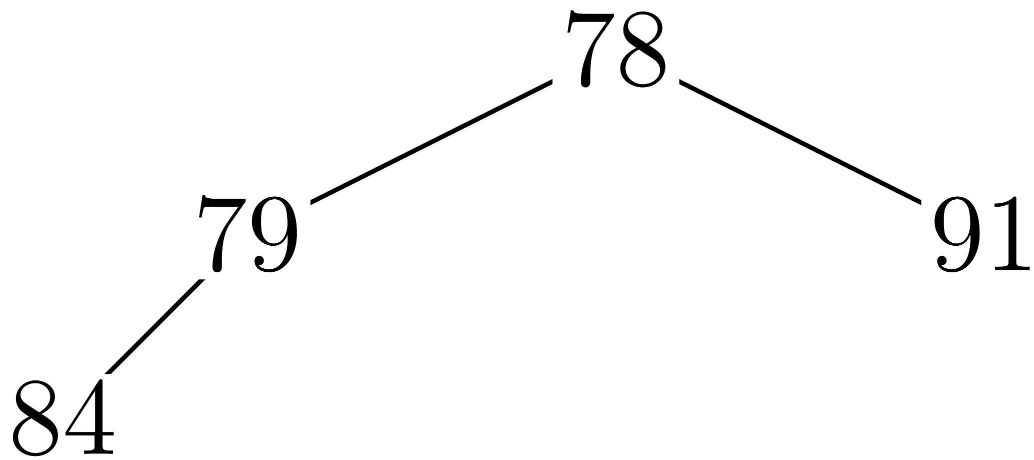
Example of Heap Sort

19	**27**	**33**	**39**	**55**					



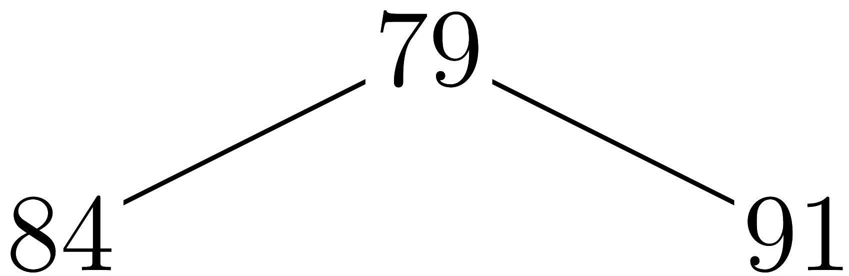
Example of Heap Sort

19	**27**	**33**	**39**	**55**	**76**				



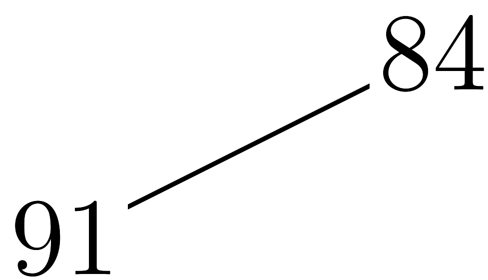
Example of Heap Sort

19	**27**	**33**	**39**	**55**	**76**	**78**			



Example of Heap Sort

19	**27**	**33**	**39**	**55**	**76**	**78**	**79**		



Example of Heap Sort

19	**27**	**33**	**39**	**55**	**76**	**78**	**79**	**84**	

$$91$$

Example of Heap Sort

19	**27**	**33**	**39**	**55**	**76**	**78**	**79**	**84**	**91**

Complexity of Heap Sort

- As we have to add n elements and then remove n elements, the worst-case time complexity is **log-linear**, i.e. $O(n\log(n))$
- This is actually a very efficient algorithm

A word on the standard Heap Sort algorithm (not examinable!)

- Standard Heap Sort (invented by John Williams in 1964) works as follows:

  1. start with a non-sorted array
  2. transform this into a max-heap without using any additional storage
     - ⋆ How can you do this?
  3. order the resulting array by repeatedly removing the maximum from the current heap
     - ⋆ How can you do this?

Standard Heap Sort (not examinable!)

The following implementation of the standard Heap Sort algorithm uses variants of the methods `percolateUp()` and `percolateDown()` that take an additional argument giving the heap size. (This is, in general, different from `list.size()`, both when repeatedly adding to the heap and when repeatedly removing the maximum element from the current heap.)

```
public void Heap Sort(){
   // starts with an unsorted list and produces a sorted list
   for (i =1; i<list.size(); i++)
       percolateUp(i+1);
   for (i=list.size()-1; i>0; i--){
      swap(0,i);
      percolateDown(0,i);
   }
}
```

Lessons

- Heaps are a powerful data structure – they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queues have many uses
  - They are used in operating systems
  - They can be used to perform pretty efficient sort
  - They are often used for implementing greedy-type algorithms