Lecture 13
Heaps and priority queues

EECS-214

Queues

- Simplified **sequence** data structure
  - **Insertions** only at the end ("**tail**")
  - **Deletions** only from the beginning ("**head**")

- **First-in, first out**
  - Objects are dequeued in the order they were enqueued

- Simple API
  - **Enqueue**: add item to the end of the queue
  - **Dequeue**: remove item from the front

Priority queues

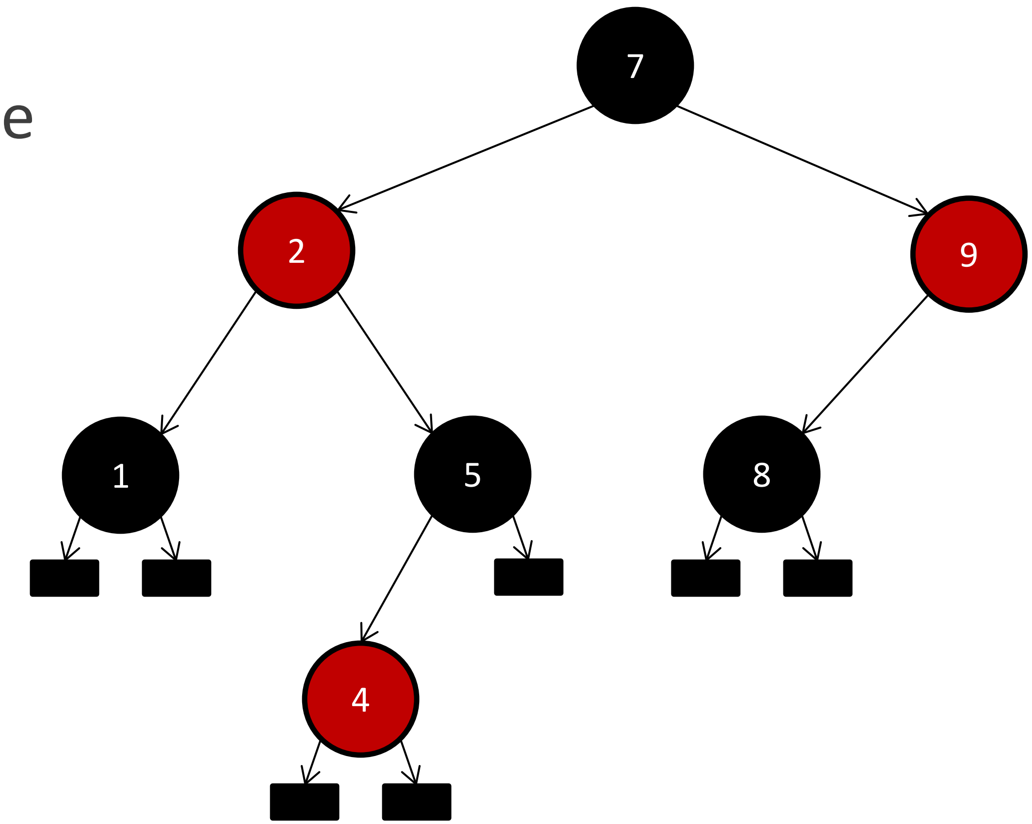
- Like normal queues
  - Objects **wait in line** to be processed
- However, items have an associated **numeric priority**
  - Priority specified when added to queue
  - Objects removed from queue in order of priority
- Slightly different API
  - **Insert**(object, priority)
    - Adds object with specified priority
  - **ExtractMax**()
    - Returns highest priority object

Priority queues

- Like normal queues
  - Objects wait in line to be processed
- However, items have an associated numeric priority
  - Priority specified when added to queue
  - Objects removed from queue in order of priority
- Slightly different API
  - Insert(object, priority)
    - Adds object with specified priority
  - **ExtractMin**()
    - Returns **lowest** priority object

Implementing priority queues

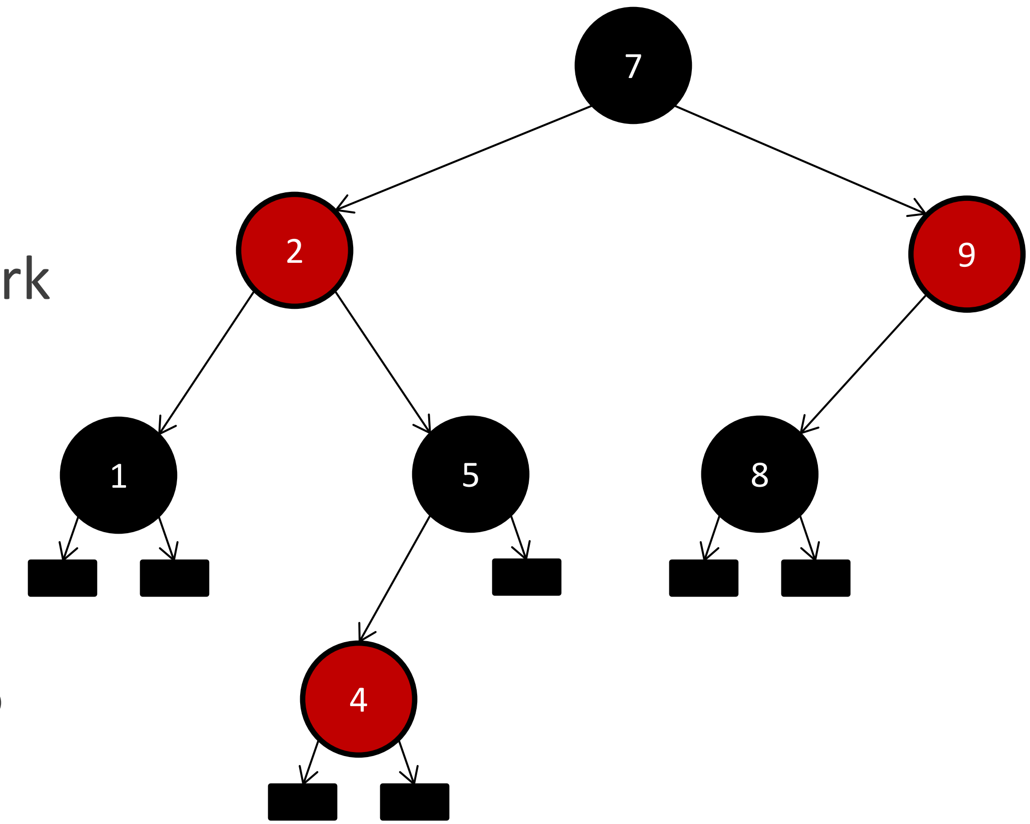
- We can use a **balanced tree** (e.g. a red/black tree) as a priority queue
  - Insert using the normal RBT insert
    - $O(\log n)$ time
  - Extract max is
    - Get the maximum element
      - $O(\log n)$
    - Delete it
      - Also $O(\log n)$
- **$O(\log n)$** time



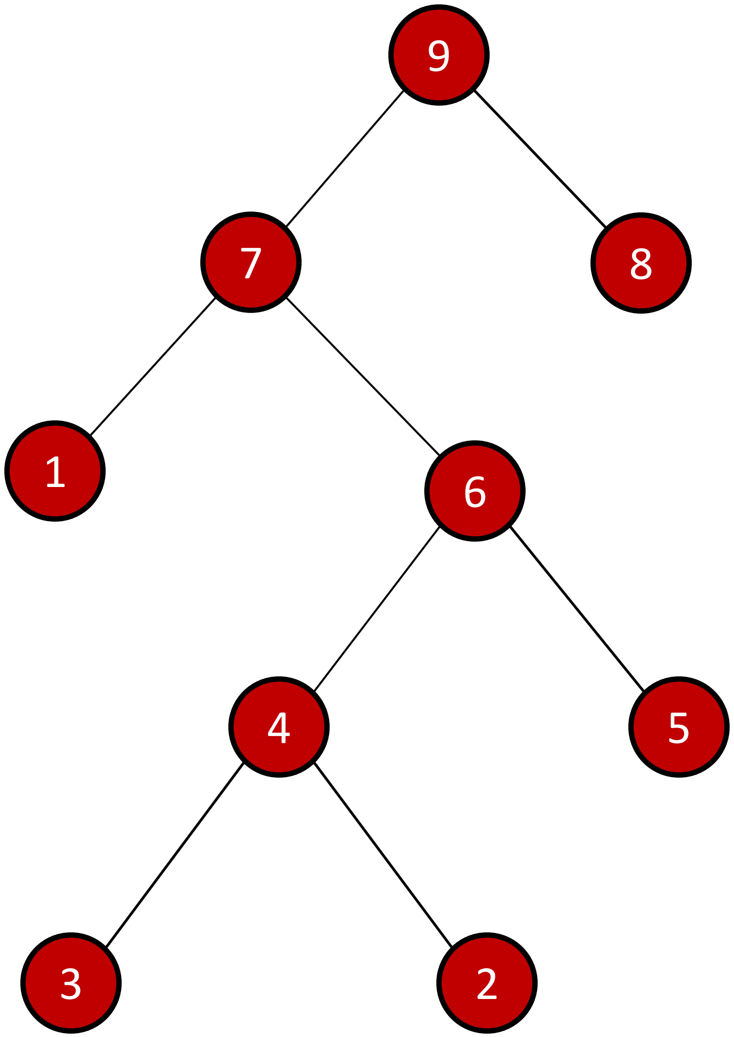
Implementing priority queues

- Unfortunately, red/black trees are **pretty complicated**
  - They go to a lot of work to keep all the items perfectly sorted

- Is there something **simpler** we could do?

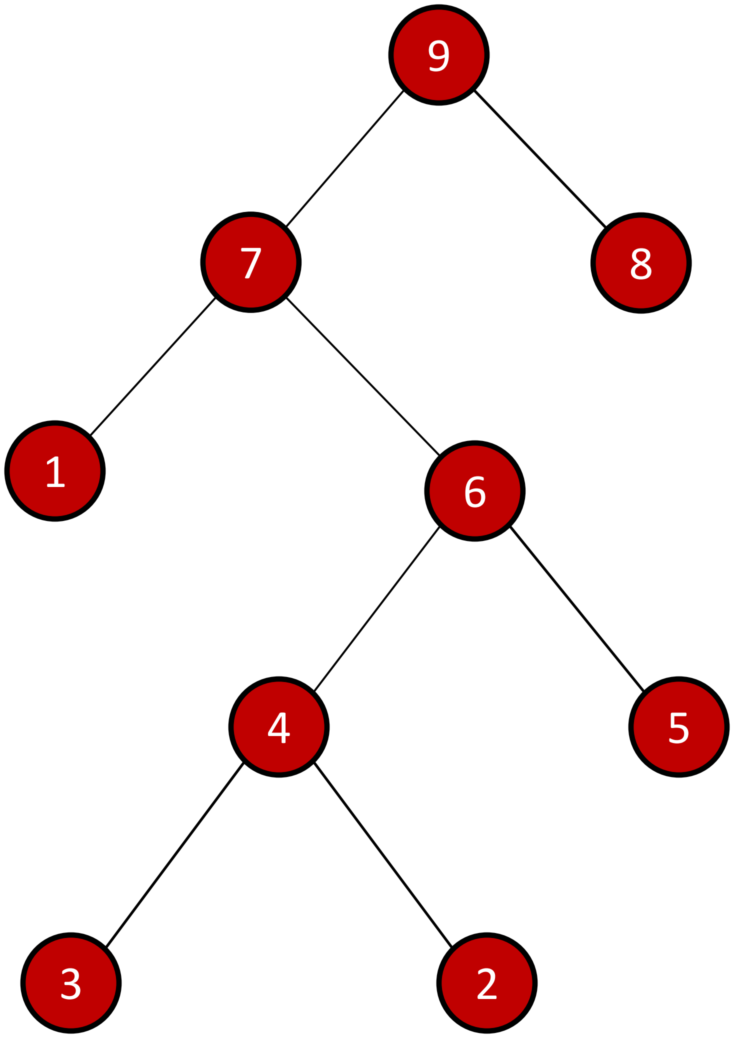


Heaps



- Heaps are a simple **tree structure** for implementing **priority queues**
- Rather than requiring their in-order traversal to be sorted
  - We just require that **parent nodes be larger than** their child nodes
  - Or **smaller**, if it's a **min heap**
- There are lots of exotic types of heaps
  - We'll focus on binary heaps
  - Which are **complete binary trees** with the heap property
  - We'll get to the completeness thing in a minute…

Heaps



Proposition: the **largest element** of a heap is always its **root**

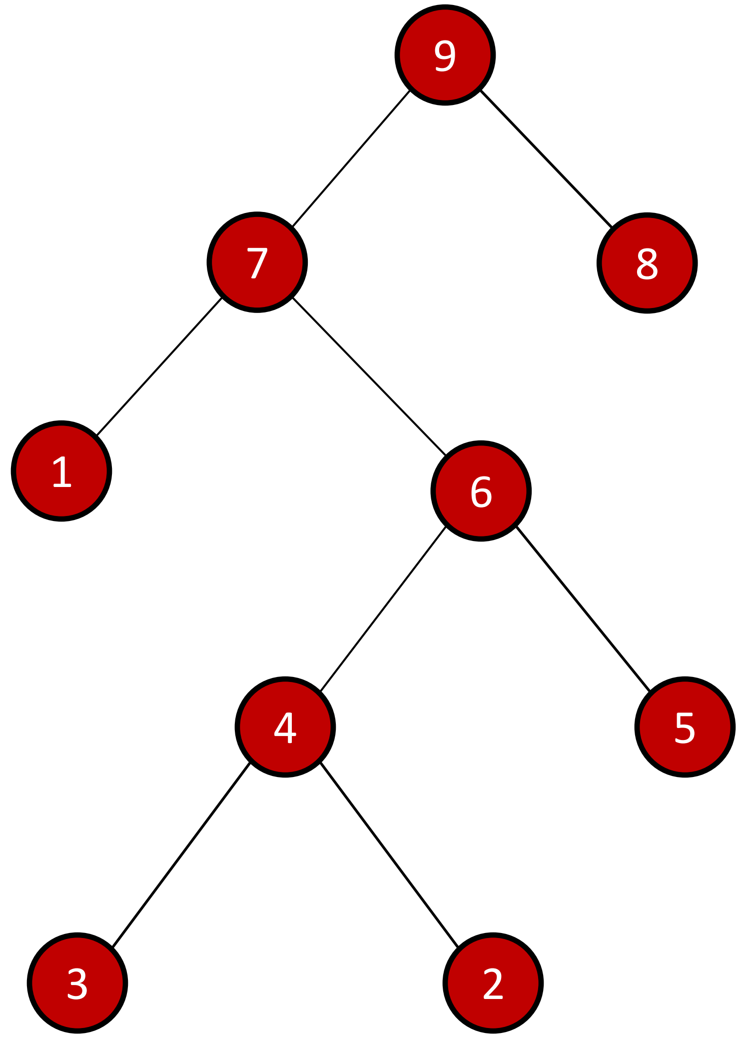
Proof:

- Suppose some **other element** is the largest element
- Since it isn't the root, it **must have a parent**
- Since it's the largest element, it **must be larger than its parent**
- But that **contradicts** the definition of a heap
- So the largest element must be the root

Sketch of heap algorithms

ExtractMax (version 1)

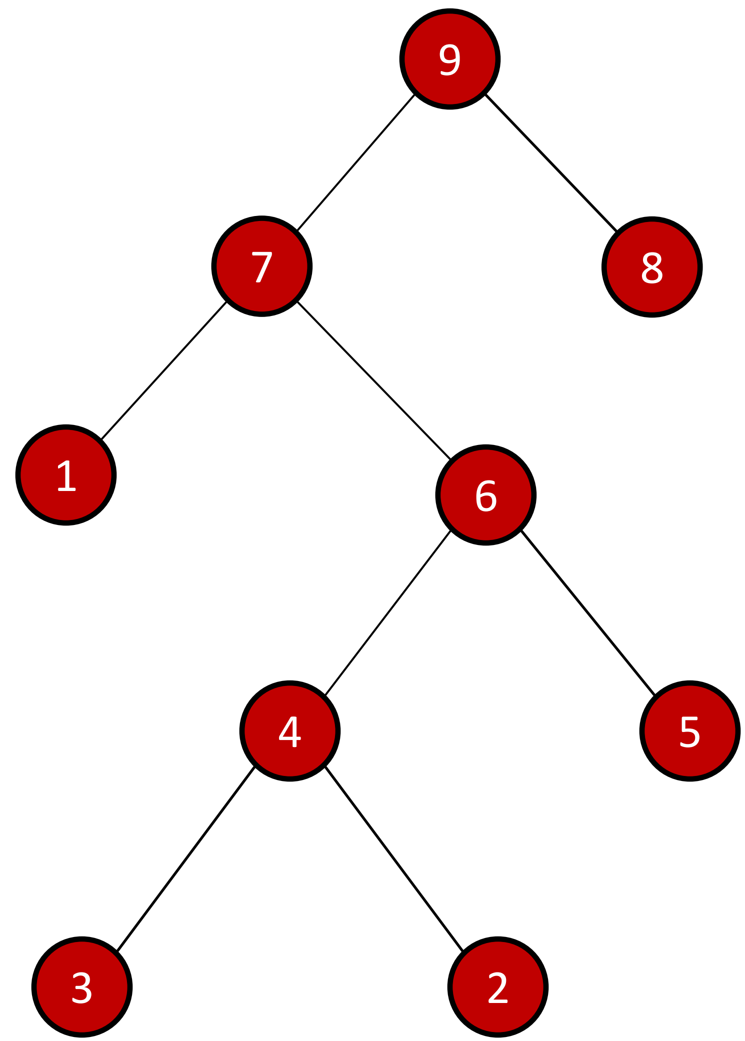
- We know the **root** is the **maximal** element
- So we want to **delete it** and **return it**
- But we need to **replace it** with its **largest child**
  - So we find the largest child
  - And **recursively delete it** from its subtree



Sketch of heap algorithms

ExtractMax (version 1)

- We know the root is the maximal element
- So we want to delete it and return it
- But we need to replace it with its largest child
  - So we find the largest child
  - And recursively delete it from its subtree

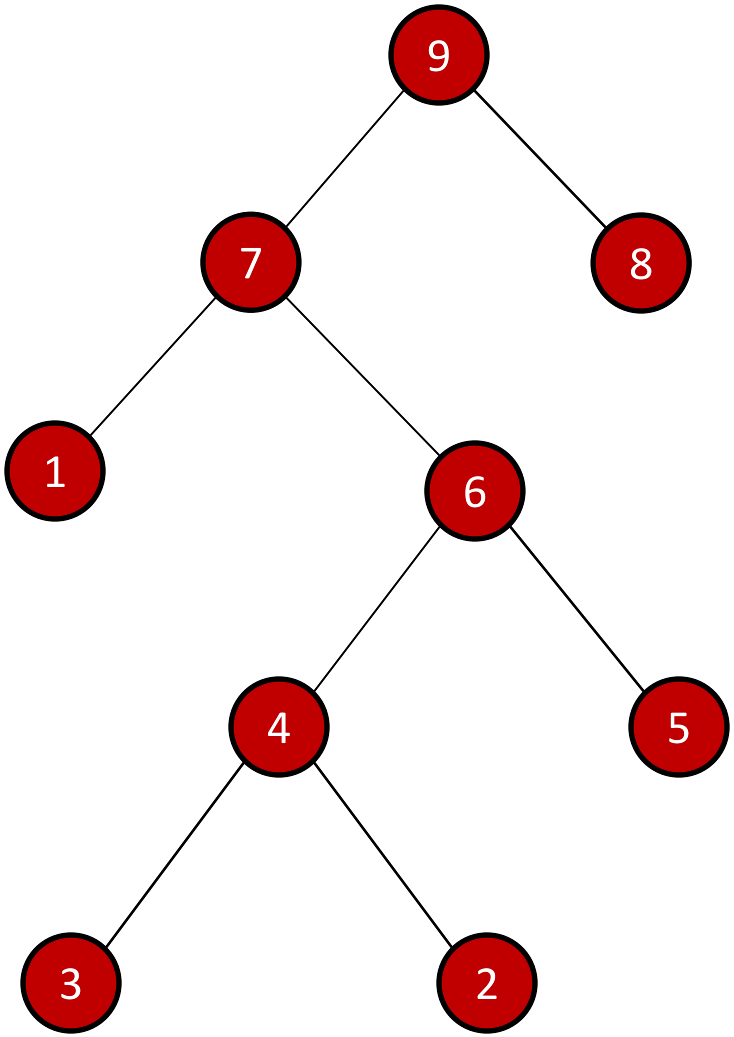


However, this is going to turn out to not to be the most convenient algorithm

Sketch of heap algorithms

ExtractMax (verison 2)

- Replace the root with a **leaf node**
  - Getting to a leaf will **turn out to be easy**
- The leaf node is **probably too small**
- So **move it downward** in the tree to make it be a proper heap again



Sketch of heap algorithms

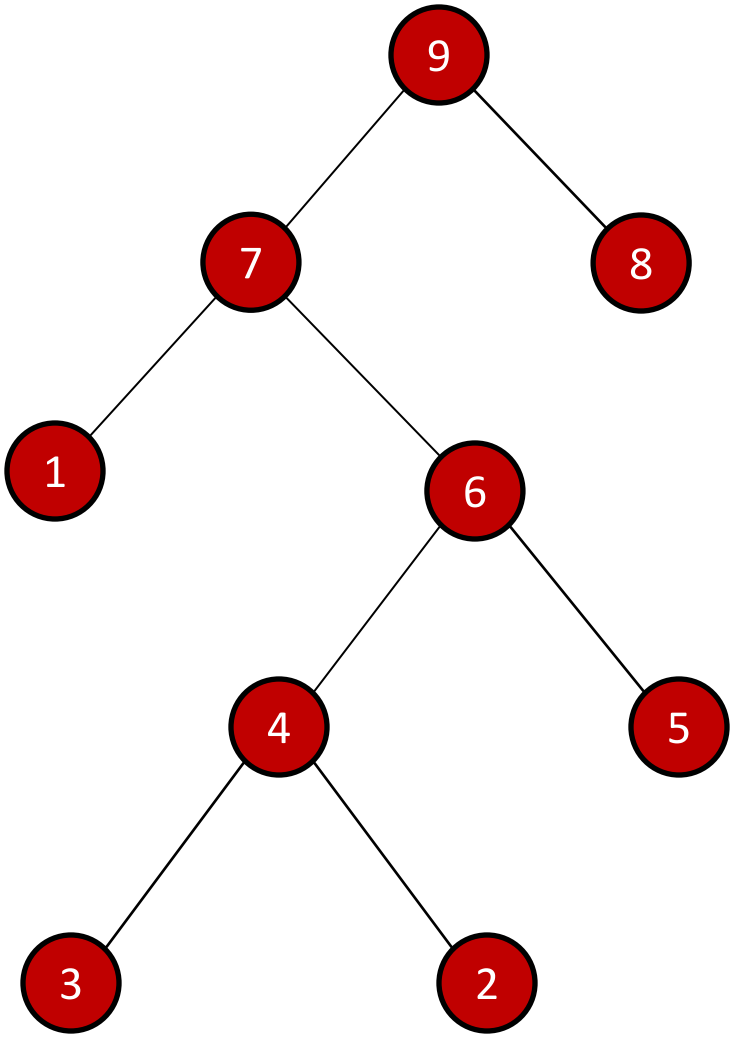
ExtractMax(heap)

result = root of heap

replace root with leaf

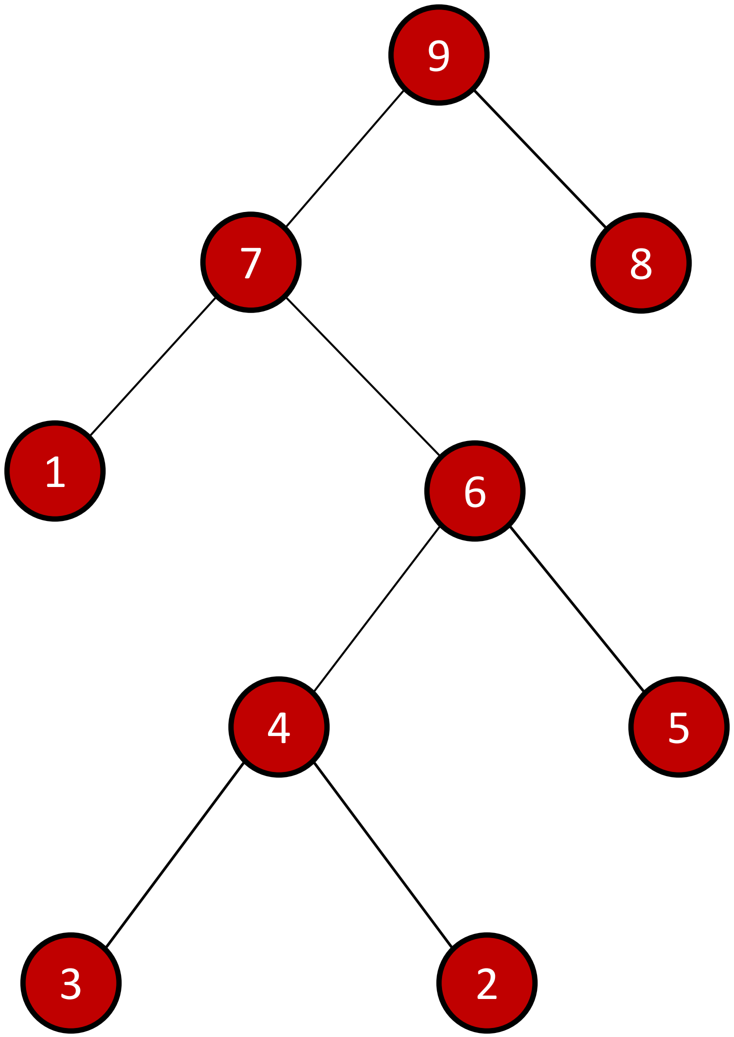
Heapify(heap)

return result



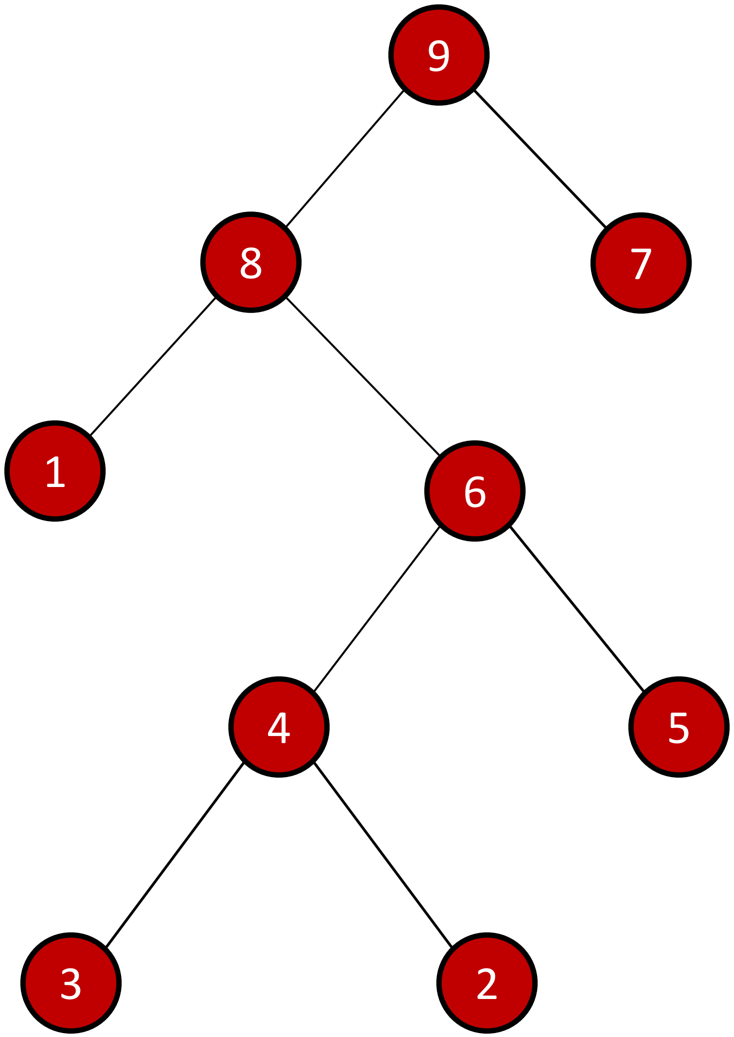
Sketch of heap algorithms

Heapify(root)
    if root > children
        done
    else if left child > root and
            left child > right child
        swap root and left child
        Heapify(left child)
    else
        swap root and right child
        Heapify(right child)



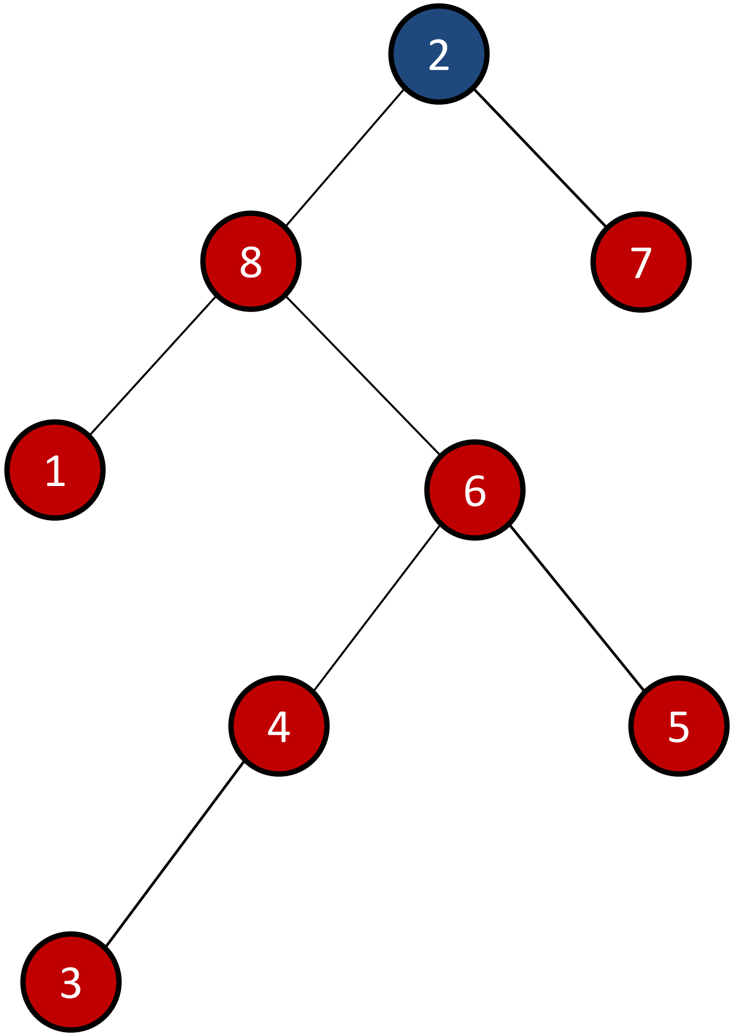
Example

- Starting configuration



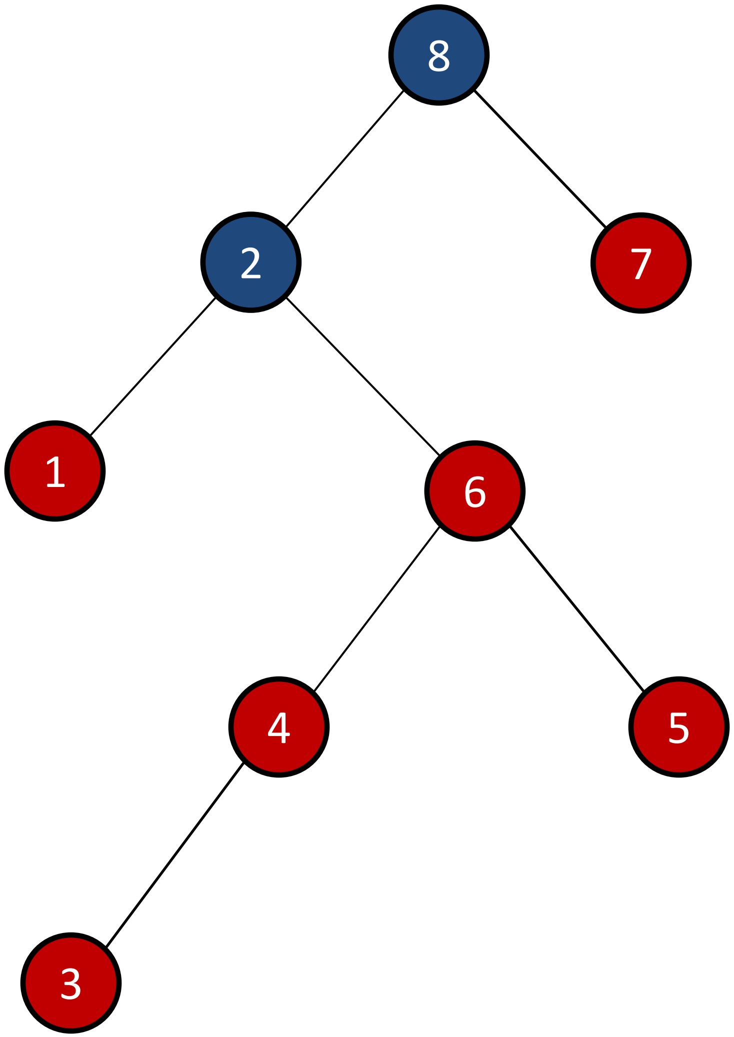
Example

- **Replace root** with **leaf**
  - **Violates** heap property



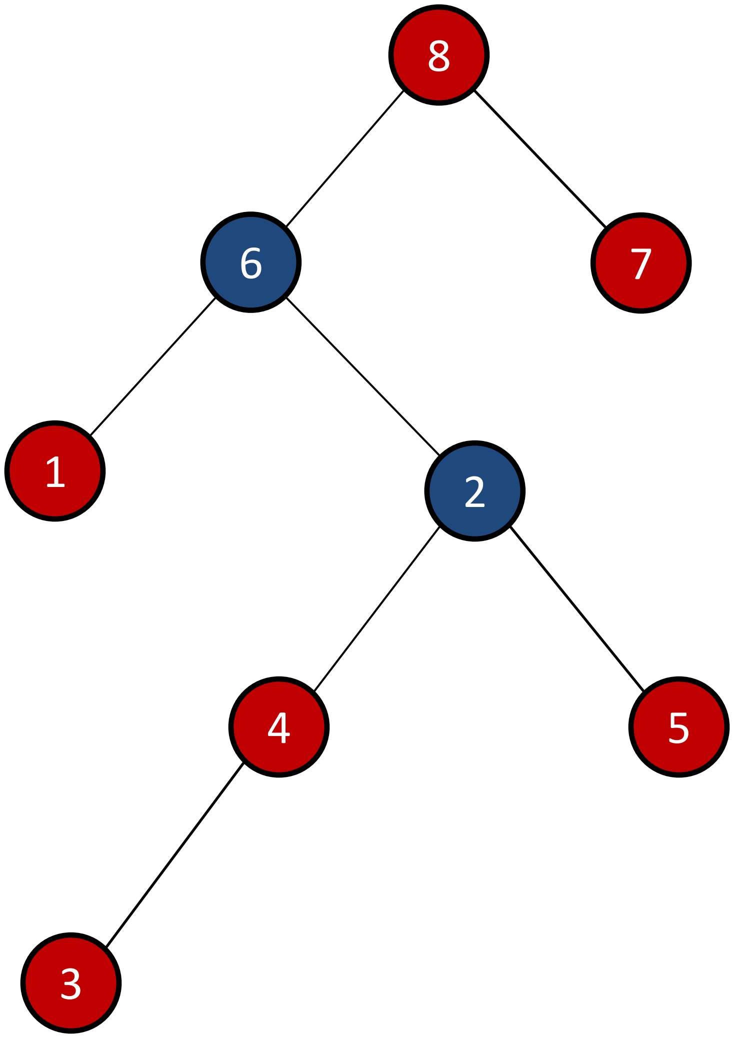
Example

- Replace root with leaf
- **Swap** with **largest** child
  - Still violates heap property



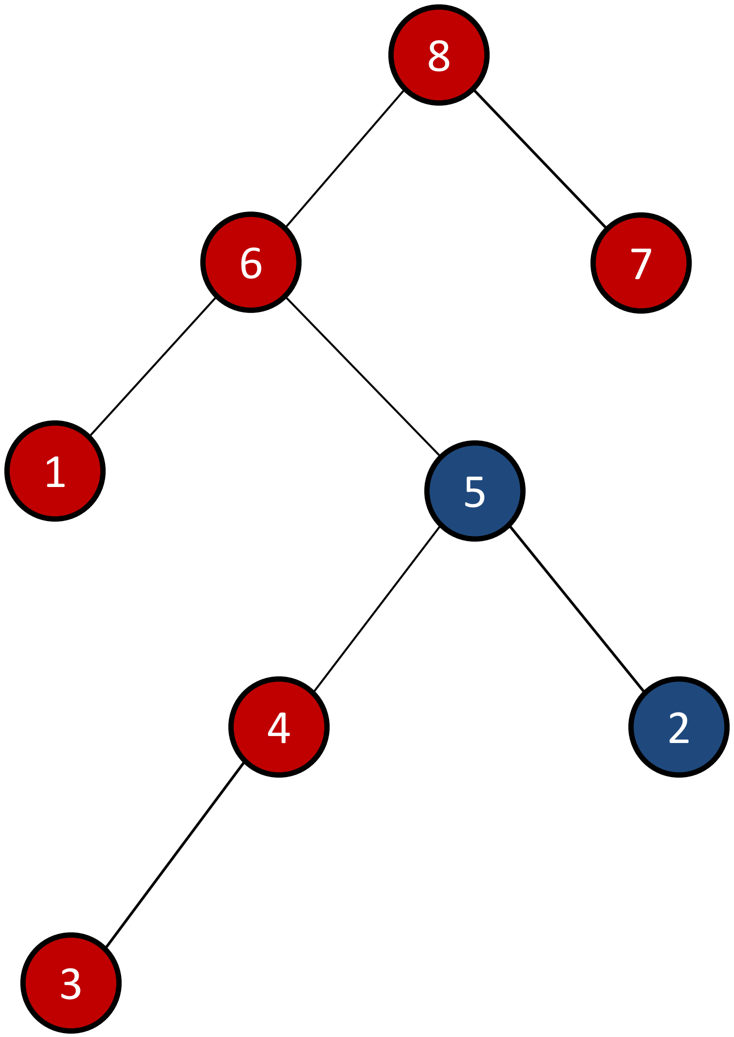
Example

- Replace root with leaf
- Swap with largest child
- **Swap** with **largest** child again
  - Still violates heap property



Example

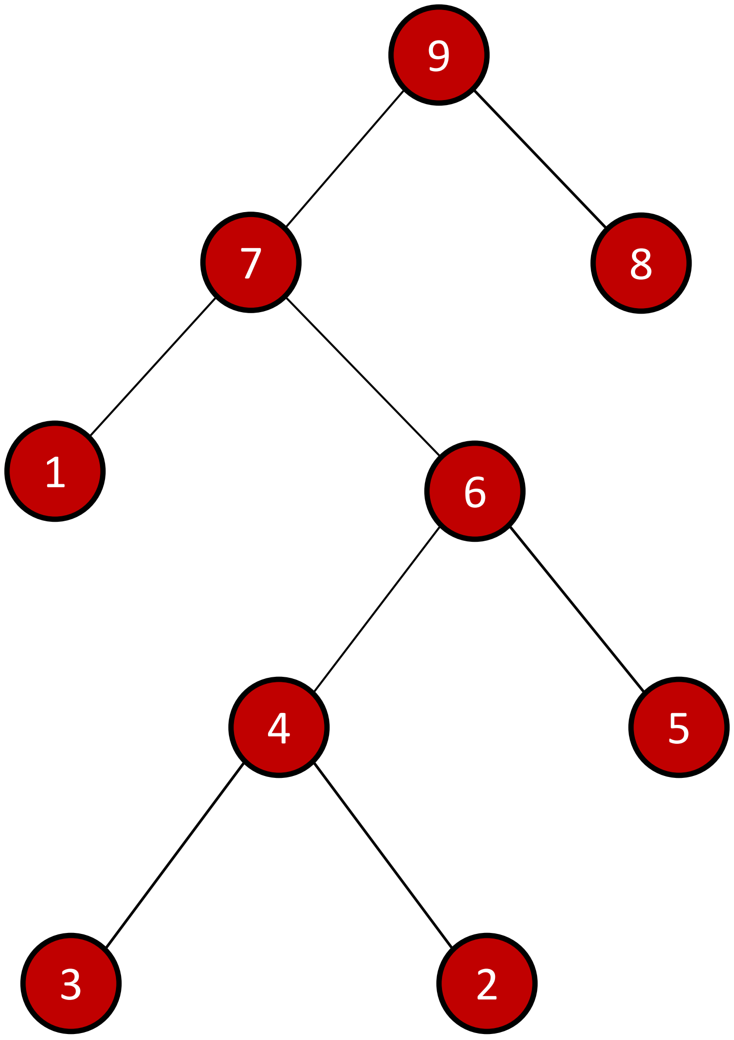
- Replace root with leaf
- Swap with largest child
- Swap with largest child again
- **Swap** with **largest** child again
  - Now a **proper heap**



Sketch of heap algorithms

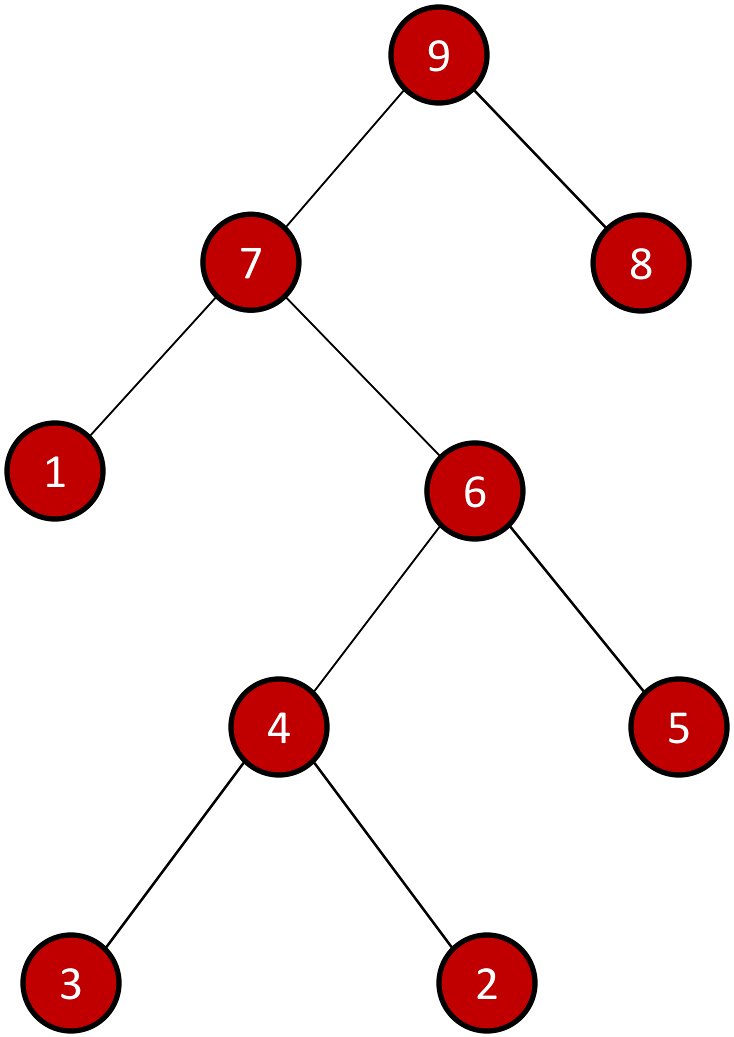
Insert(item)

- **Add item as a leaf**
  - Again, trust us, this will turn out to be easy
- While item > its parent
  - **Swap with parent**
  - Compare it to the next level up



Analysis

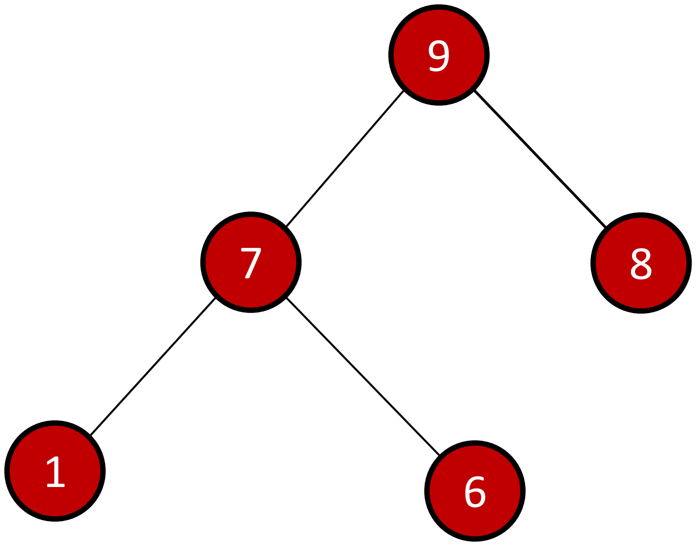
- Both algorithms
  - **Move nodes up/down** tree
  - Perform a **constant** amount of work **at each level**
- So their execution time is $\boldsymbol{O(h)}$
  - Where h is the tree's height
- Again, this is **good, if the tree is balanced**, bad otherwise



Complete binary trees

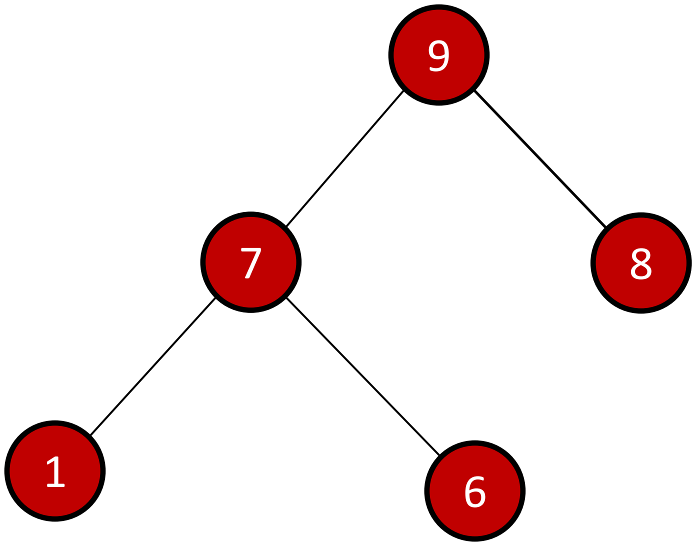
- A **complete** binary tree is a binary tree in which
  - **Every level** of the tree is **full**, **except** possibly the **last**
    - Can't add anymore nodes
  - Every node is **shifted** as far to the **left** as possible

- Complete trees are **optimally balanced**



Binary heaps

- A **binary heap** is a
  - Complete binary tree
  - That satisfies the heap property

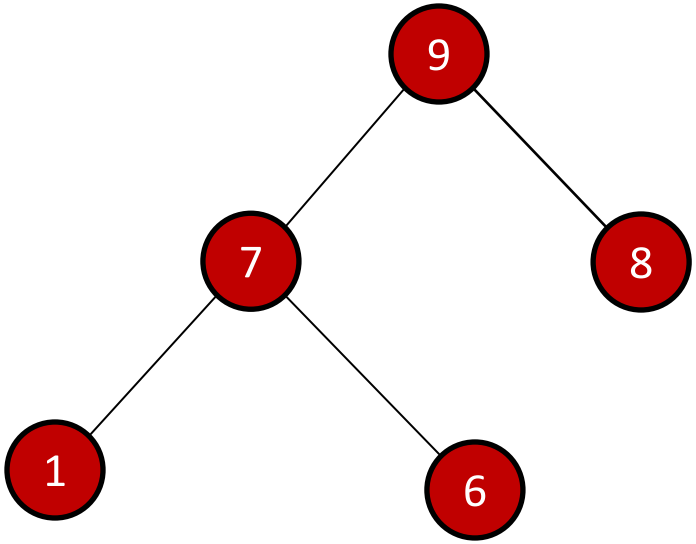


- Great!
- How do we ensure that the heap is a complete binary tree?

Embedding in an array

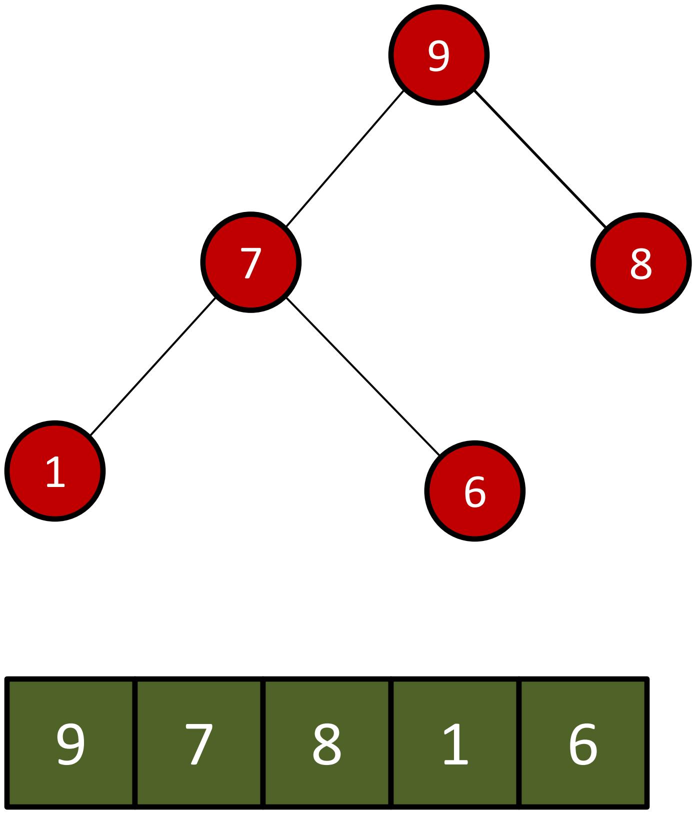
- It turns out that any complete binary tree can be **embedded** an array in a particularly cleaver way

- We can **compute**
  - The position of its **parent** in the array,
  - and the positions of its **children**,
  - directly from its own **position**



Embedding in an array

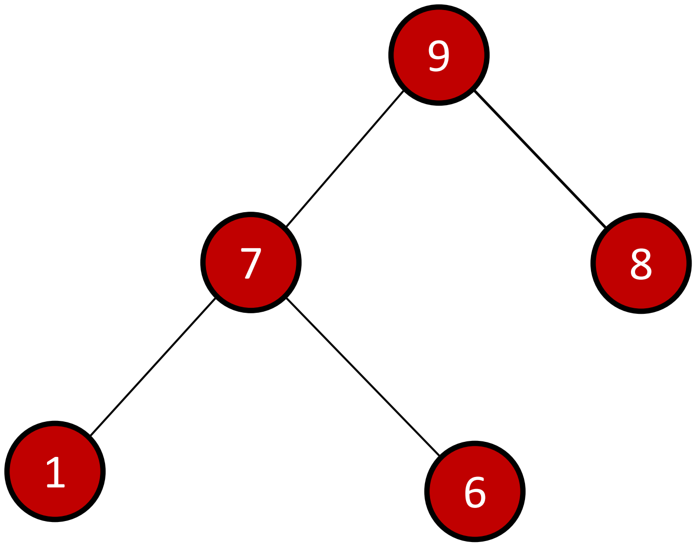
- Store the **root** in the first element (**element 0**)
- For any node
  - Let i be its position in the array (for the root, $i = 0$)
  - Store its **left child** at position $2i + 1$
  - Store its **right child** at position $2i + 2$
  - Its parent can be found at position $\lfloor (i - 1)/2 \rfloor$



- Trust me that this works :-)

Why is this a good representation?

- Very **fast**
  - Can just **allocate a big array** and then never have to call new again
- **Last element** is always a **leaf**
  - Remember our algorithms needed to **add/remove leaves**?



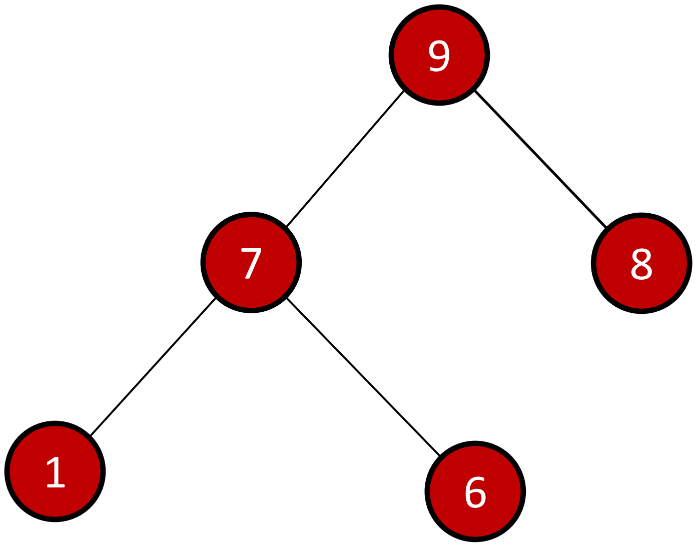
9	7	8	1	6

Representing a heap using an array

- Assume we have an extra field for the array to keep track of the size of the heap

- Define the following utility procedures:

```
Parent(int i)
  return (i-1)/2
Left(int i)
  return 2*i+1
Right(int i)
  return 2*i+2
```



9	7	8	1	6

Heap insertion using the array representation

HeapInsert(A, key)

A.size = A.size + 1

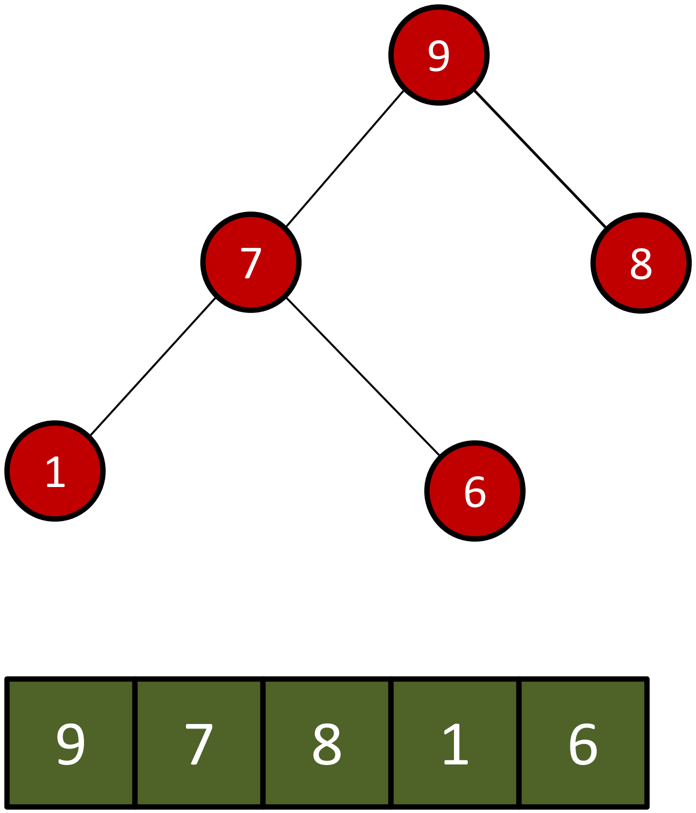
i = A.size

while i>0 and
A[Parent(i)] < key

A[i] = A[Parent(i)]

i = Parent(i)

A[i] = key



Inserting 10

HeapInsert(A, key)

A.size = A.size + 1

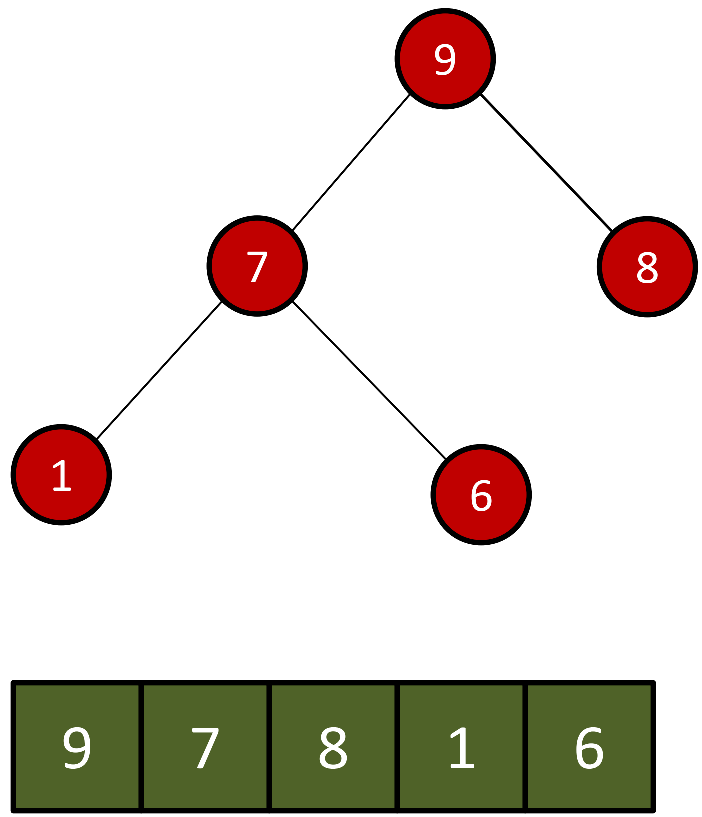
i = A.size

while i>0 and
A[Parent(i)] < key

A[i] = A[Parent(i)]

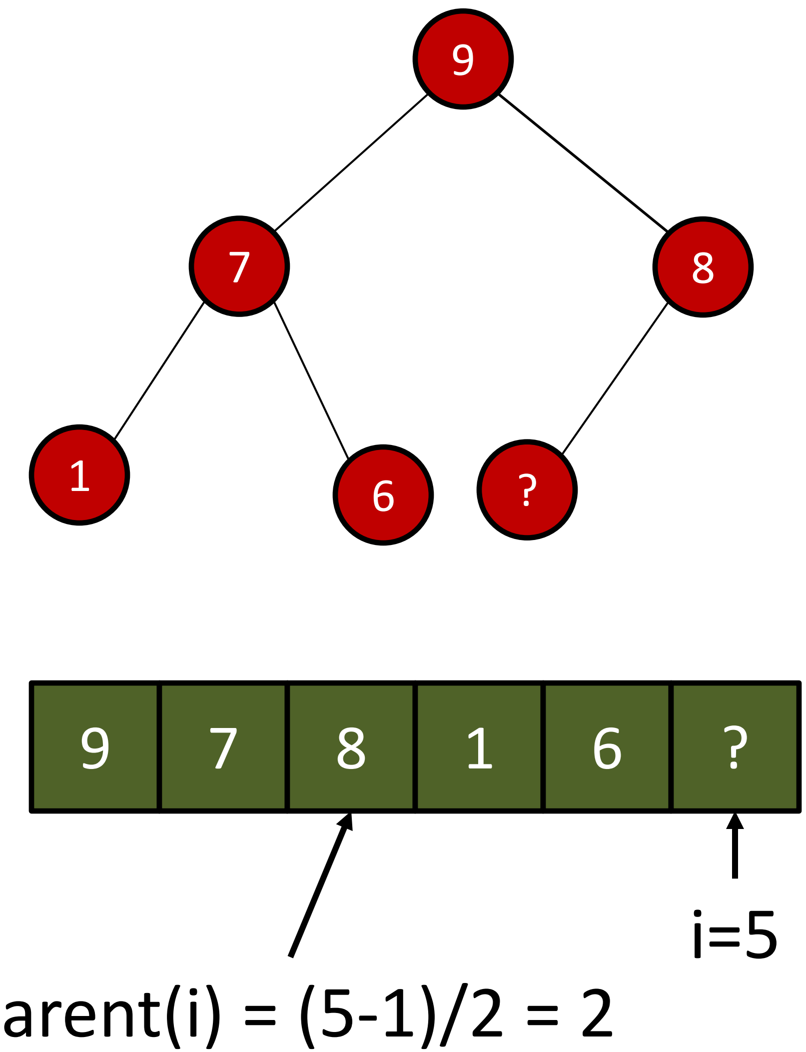
i = Parent(i)

A[i] = key



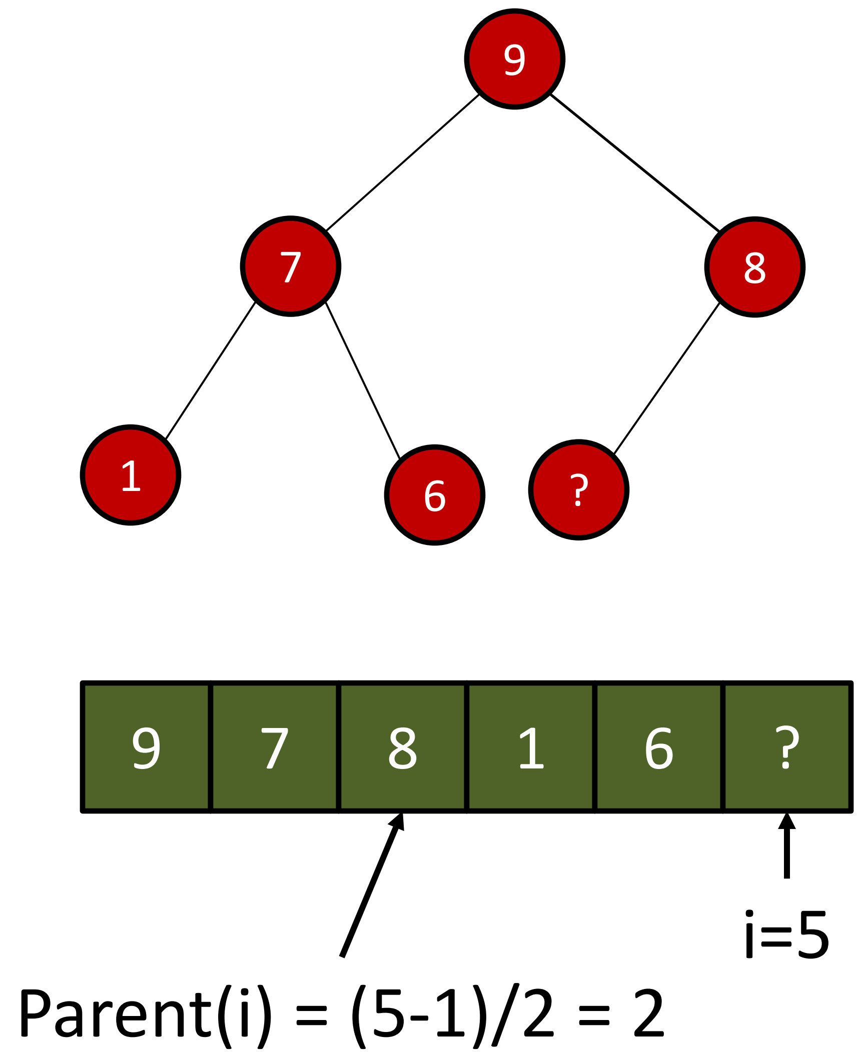
Check parent

```
HeapInsert(A, key)
    A.size = A.size + 1
    i = A.size
    while i>0 and
            A[Parent(i)] < key
        A[i] = A[Parent(i)]
        i = Parent(i)
    A[i] = key
```



8 < 10, so A[Parent(i)] < key

```
HeapInsert(A, key)
  A.size = A.size + 1
  i = A.size
  while i>0 and
        A[Parent(i)] < key
    A[i] = A[Parent(i)]
    i = Parent(i)
  A[i] = key
```



Copy parent down

HeapInsert(A, key)

  A.size = A.size + 1

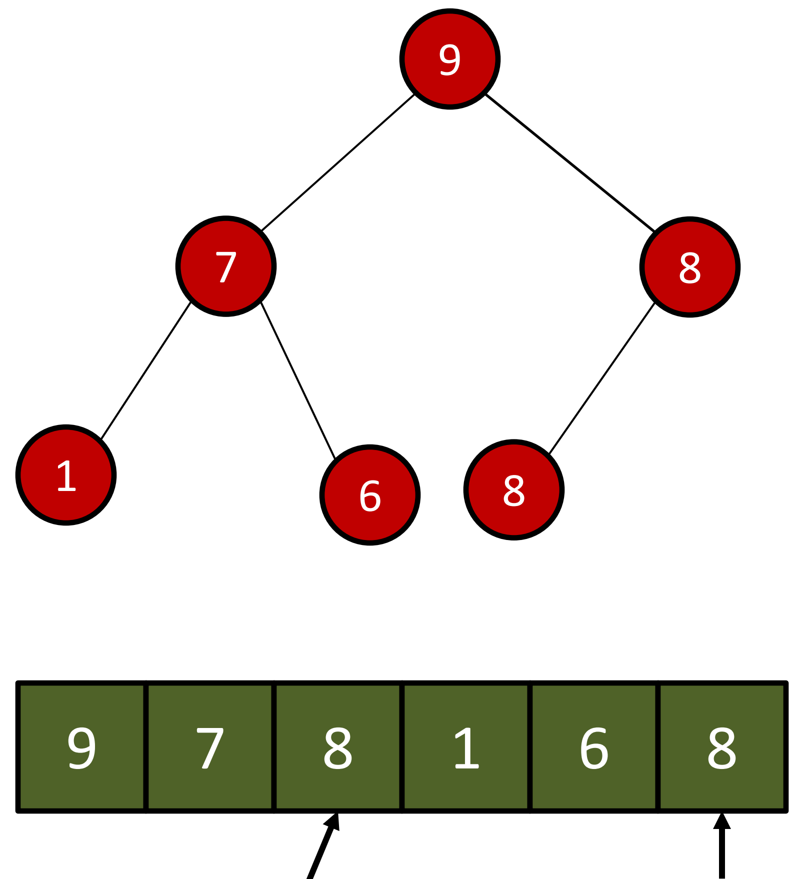
  i = A.size

  while i>0 and
      A[Parent(i)] < key

    A[i] = A[Parent(i)]

    i = Parent(i)

  A[i] = key



i=5

Parent(i) = (5-1)/2 = 2

And move up tree

HeapInsert(A, key)

    A.size = A.size + 1

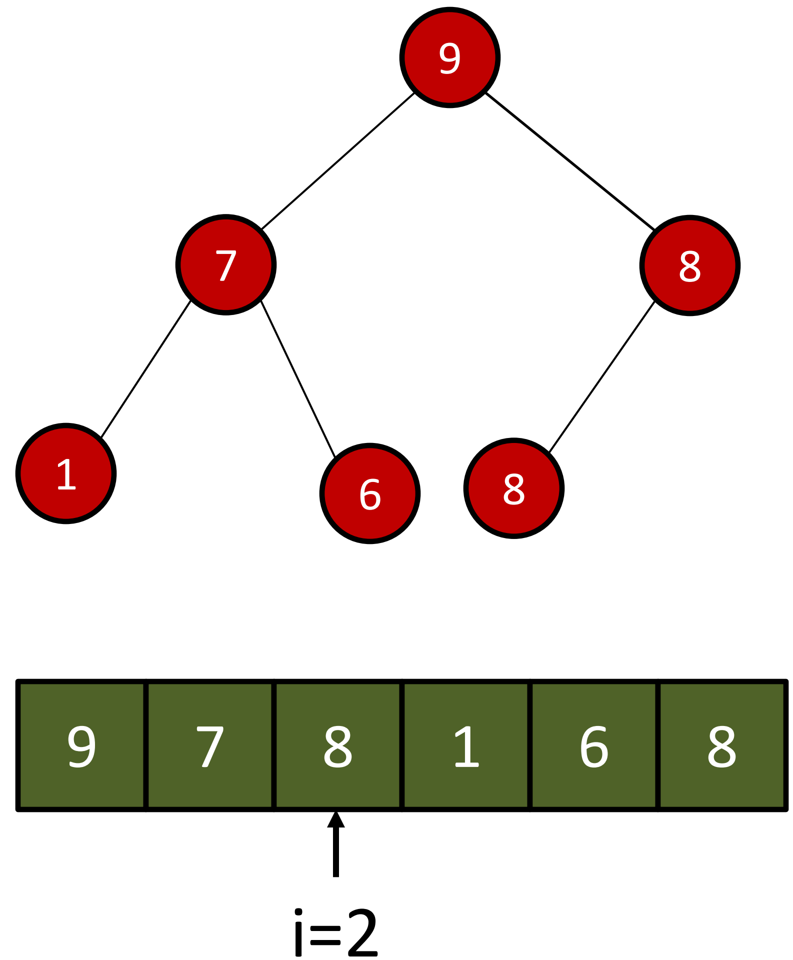
    i = A.size

    while i>0 and
            A[Parent(i)] < key

        A[i] = A[Parent(i)]

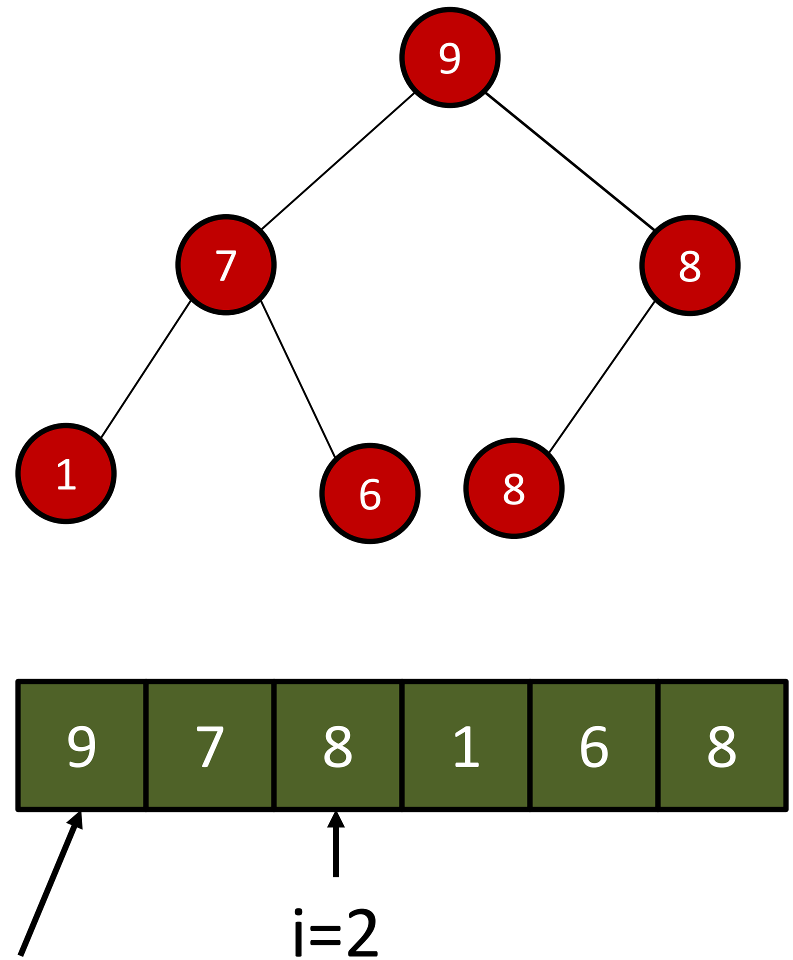
        i = Parent(i)

    A[i] = key



Check parent

```
HeapInsert(A, key)
    A.size = A.size + 1
    i = A.size
    while i>0 and
            A[Parent(i)] < key
        A[i] = A[Parent(i)]
        i = Parent(i)
    A[i] = key
```

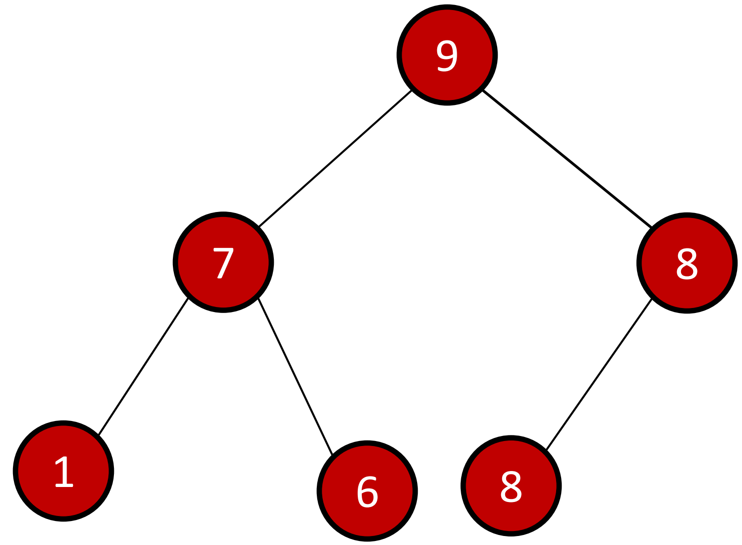


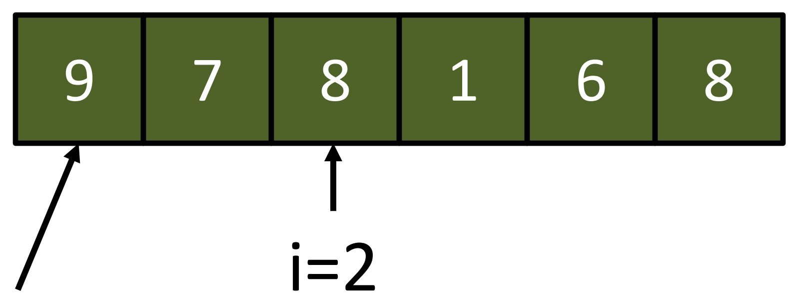
Parent(i) = (2-1)/2 = 0

(remember int arithmetic rounds down)

9 < 10, so A[Parent(i)] < key

```
HeapInsert(A, key)
    A.size = A.size + 1
    i = A.size
    while i>0 and
            A[Parent(i)] < key
        A[i] = A[Parent(i)]
        i = Parent(i)
    A[i] = key
```



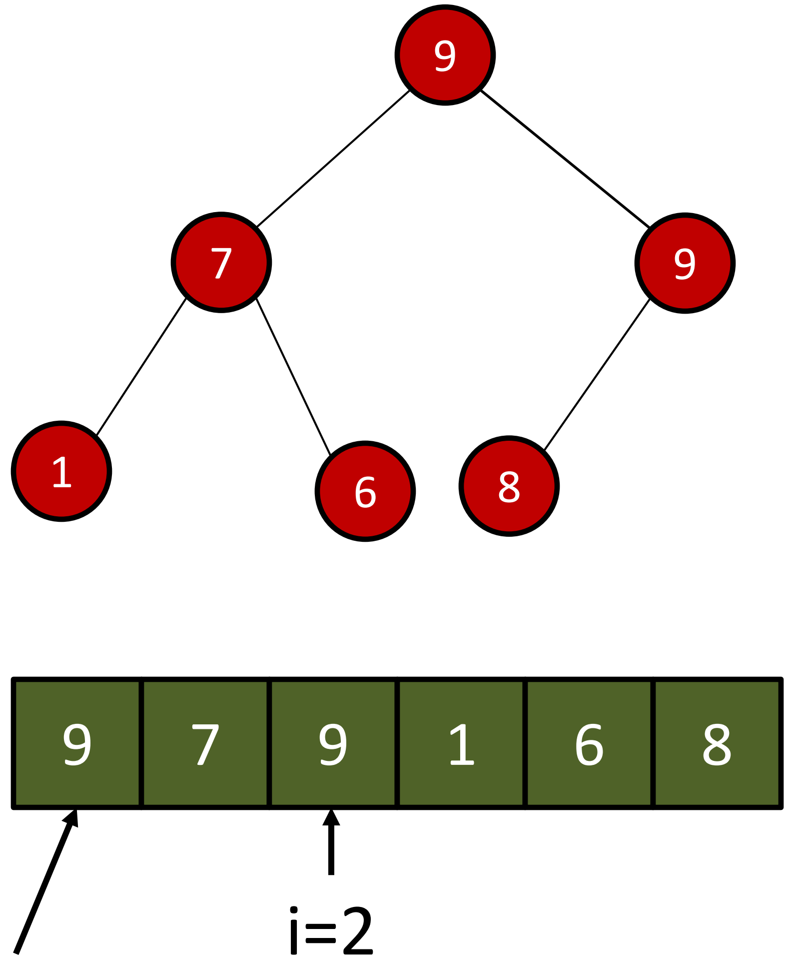


Parent(i) = (2-1)/2 = 0

(remember int arithmetic rounds down)

Copy parent down

```
HeapInsert(A, key)
    A.size = A.size + 1
    i = A.size
    while i>0 and
            A[Parent(i)] < key
        A[i] = A[Parent(i)]
        i = Parent(i)
    A[i] = key
```



Parent(i) = (2-1)/2 = 0

(remember int arithmetic rounds down)

Move up

HeapInsert(A, key)

    A.size = A.size + 1

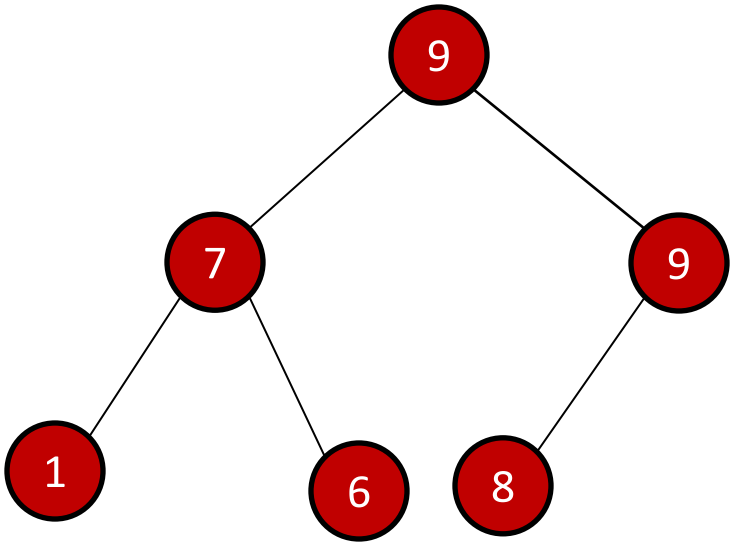
    i = A.size

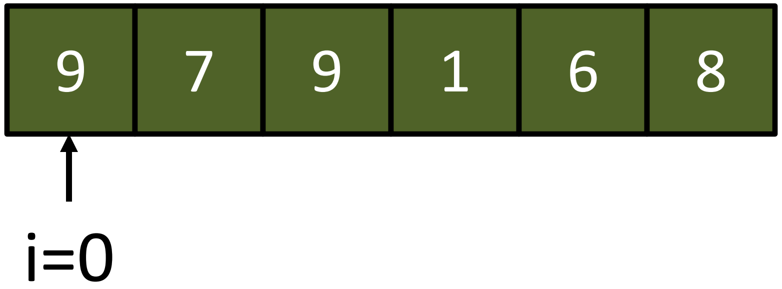
    while i>0 and
            A[Parent(i)] < key

        A[i] = A[Parent(i)]

        i = Parent(i)

    A[i] = key





Can't move farther

HeapInsert(A, key)

  A.size = A.size + 1

  i = A.size

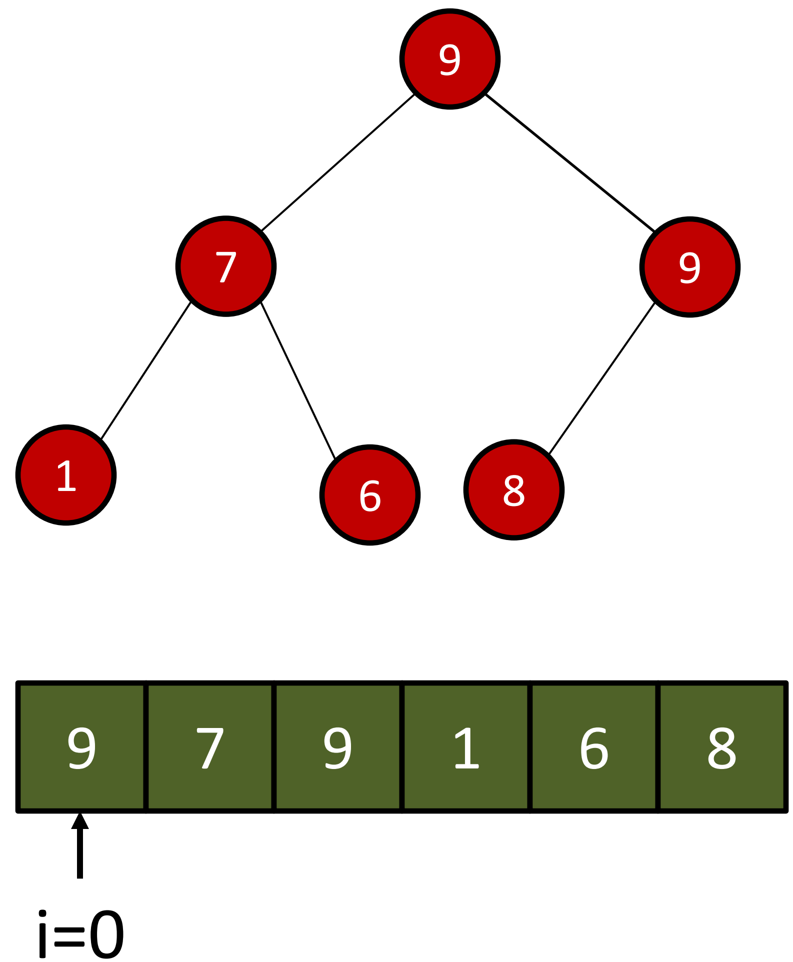
  while **i>0** and

      A[Parent(i)] < key

    A[i] = A[Parent(i)]

    i = Parent(i)

  A[i] = key



Store the new key

HeapInsert(A, key)

A.size = A.size + 1

i = A.size

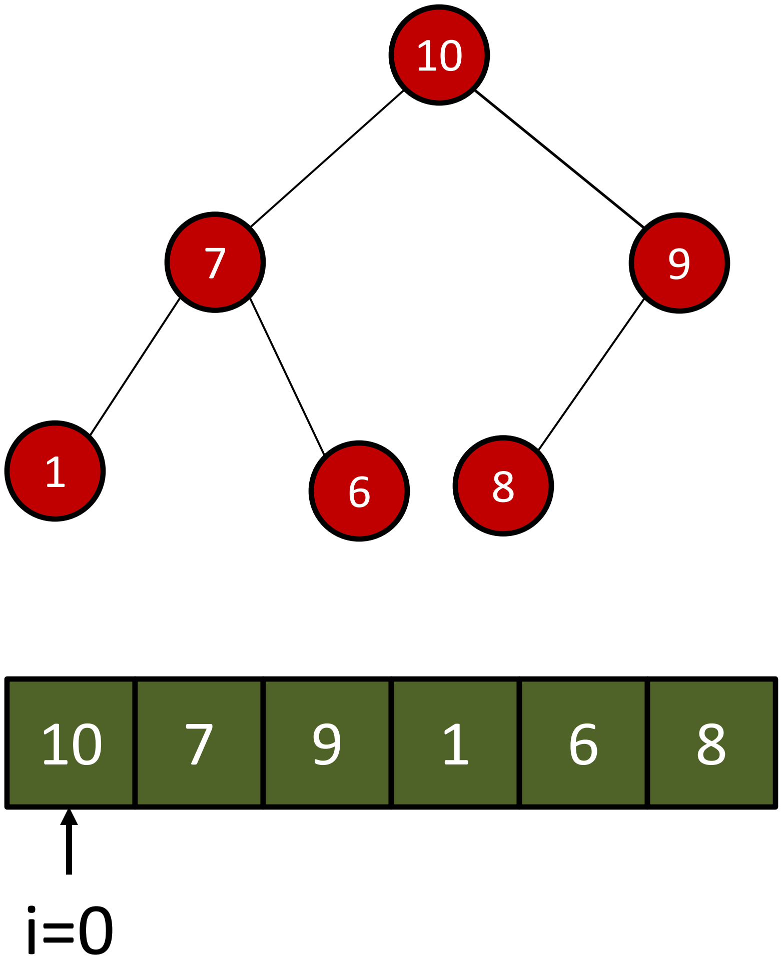
while i>0 and

A[Parent(i)] < key

A[i] = A[Parent(i)]

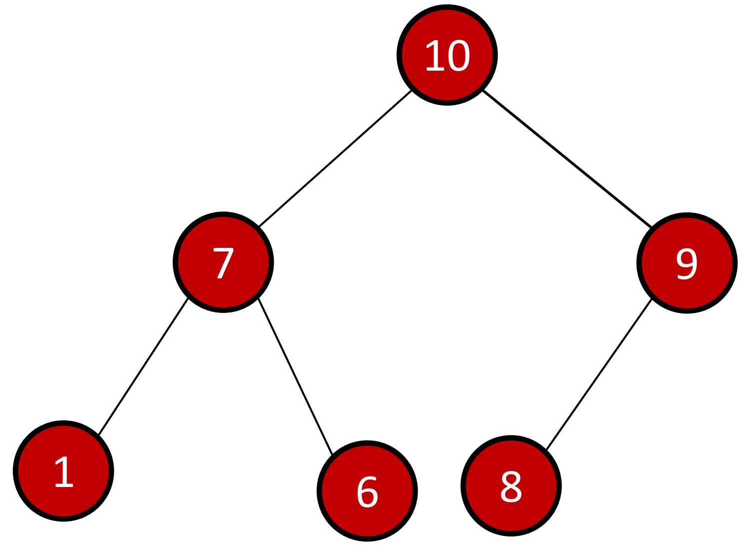
i = Parent(i)

A[i] = key



Done!

```
HeapInsert(A, key)
   A.size = A.size + 1
   i = A.size
   while i>0 and
            A[Parent(i)] < key
      A[i] = A[Parent(i)]
      i = Parent(i)
   A[i] = key
```

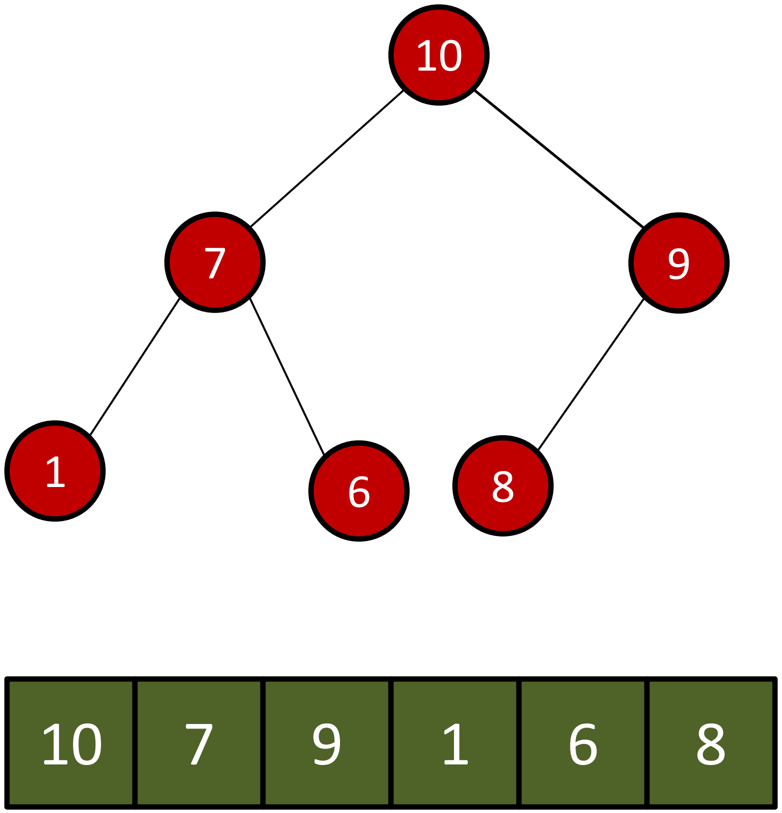


10	7	9	1	6	8

Notice that this is once again a valid heap

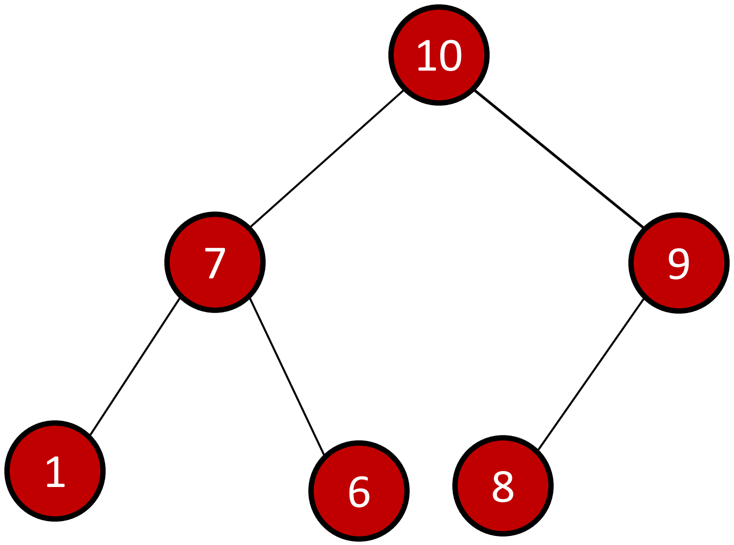
Extracting an element

```
HeapExtractMax(A)
   max = A[0]
   A[0] = A[A.size]
   A.size--
   Heapify(A,0)
   return max
```



Extracting an element

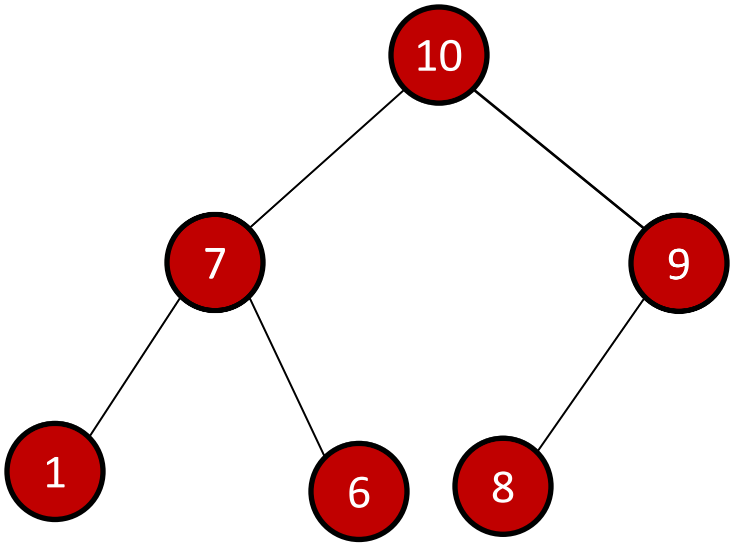
```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



10	7	9	1	6	8

Here we go!

HeapExtractMax(A)
max = A[0]
A[0] = A[A.size]
A.size--
Heapify(A,0)
return max



10	7	9	1	6	8

Remember the max (the root)

HeapExtractMax(A)

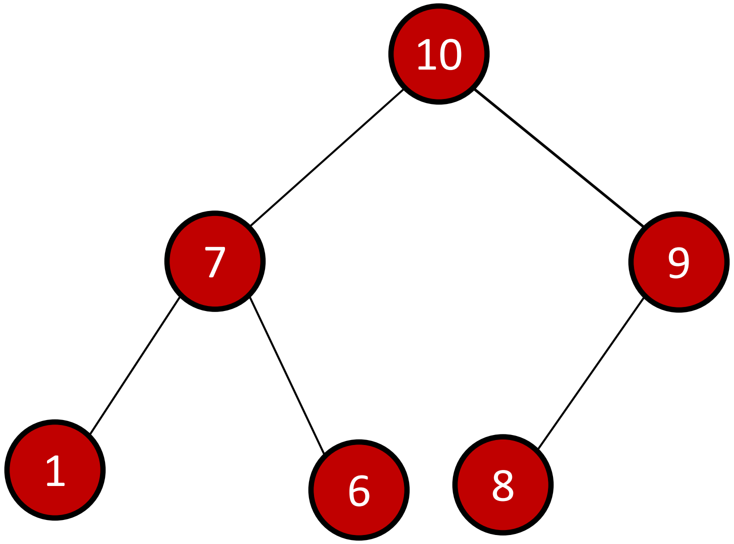
  max = A[0]

  A[0] = A[A.size]

  A.size--

  Heapify(A,0)

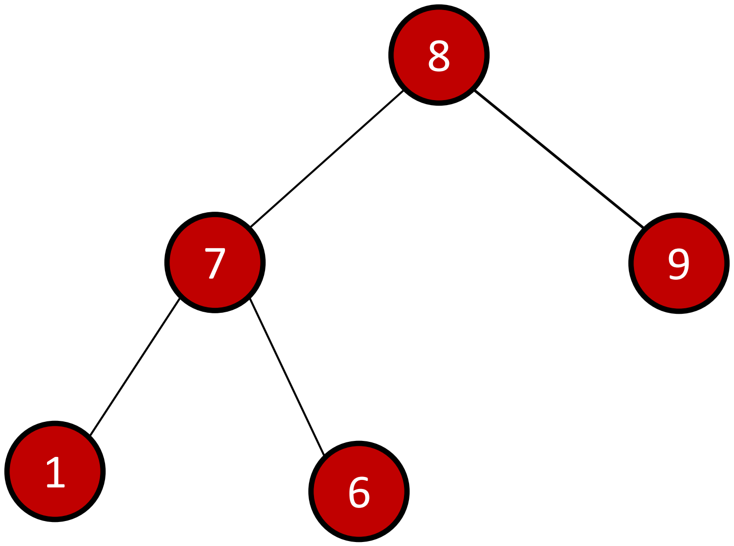
  return max



10	7	9	1	6	8

Move the last leaf to the root

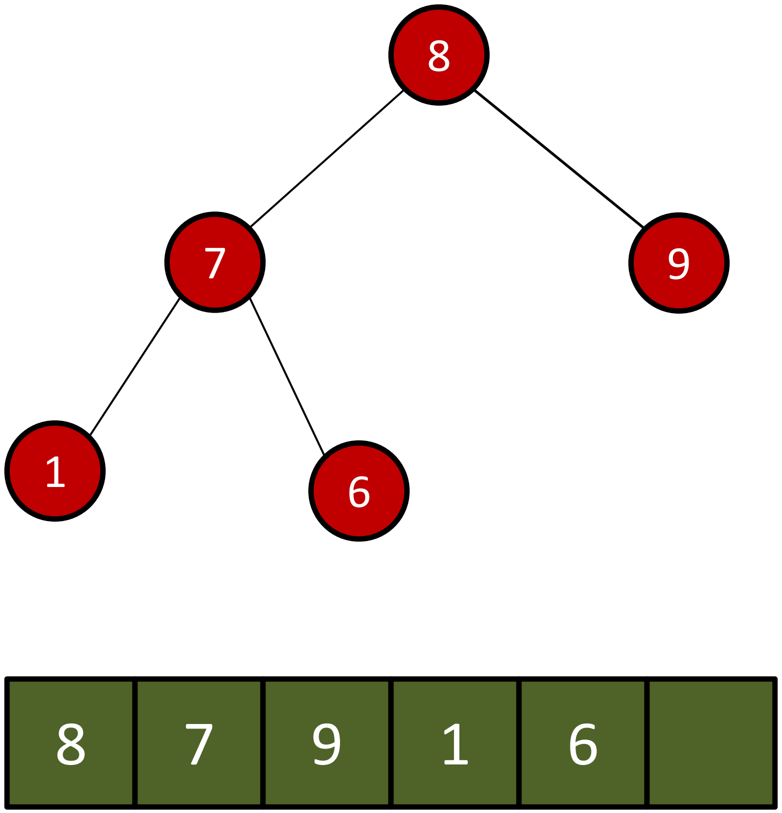
```
HeapExtractMax(A)
   max = A[0]
   A[0] = A[A.size]
   A.size--
   Heapify(A,0)
   return max
```



8	7	9	1	6	

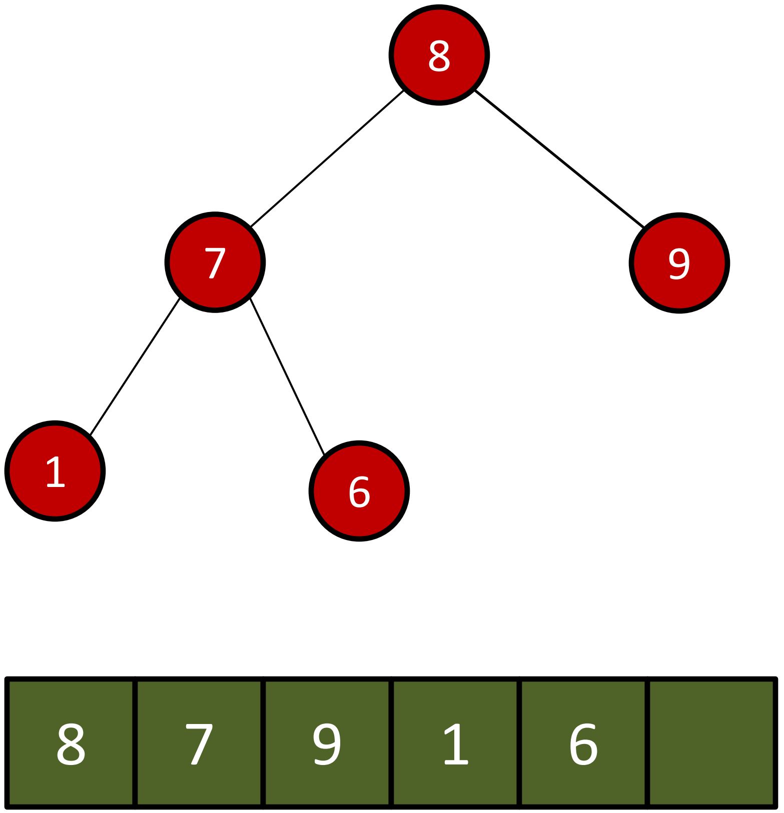
Move the last leaf to the root

```
HeapExtractMax(A)
   max = A[0]
   A[0] = A[A.size]
   A.size--
   Heapify(A,0)
   return max
```



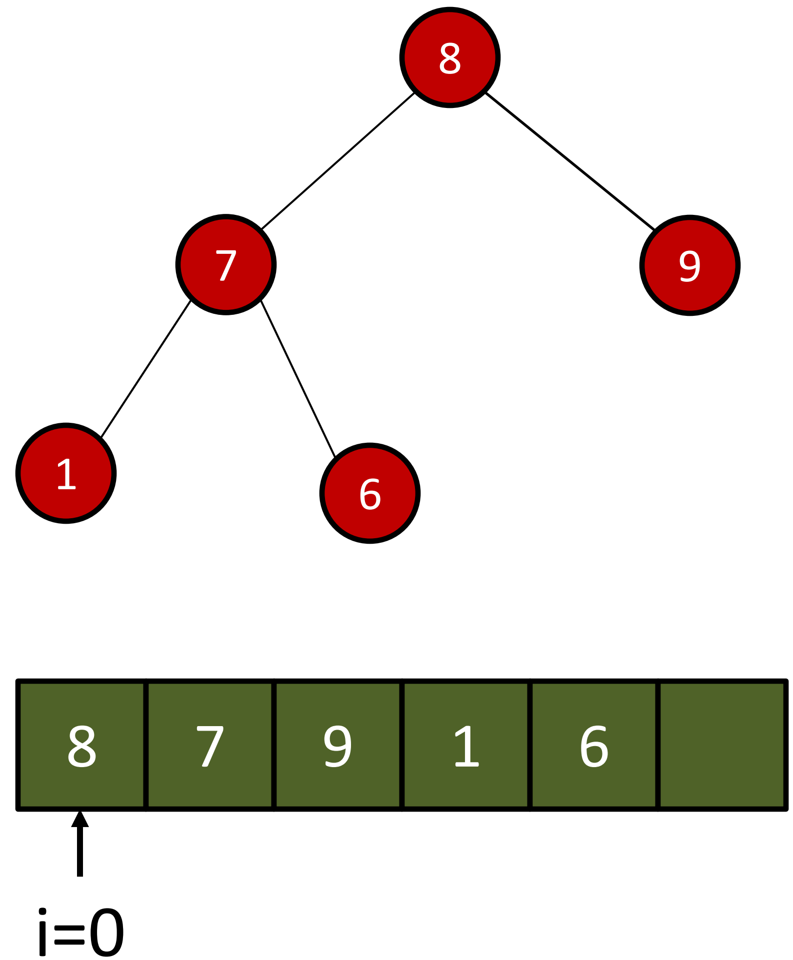
Re-heapify

```
HeapExtractMax(A)
   max = A[0]
   A[0] = A[A.size]
   A.size--
   Heapify(A,0)
   return max
```



Re-heapify

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



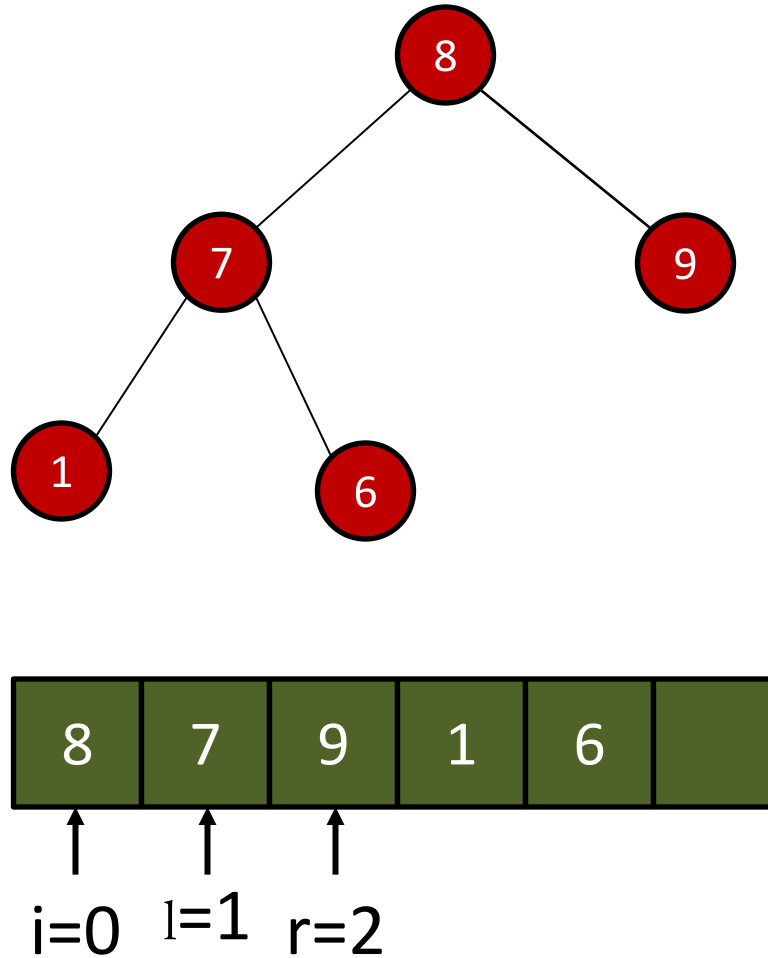
Find the left- and right-children

Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

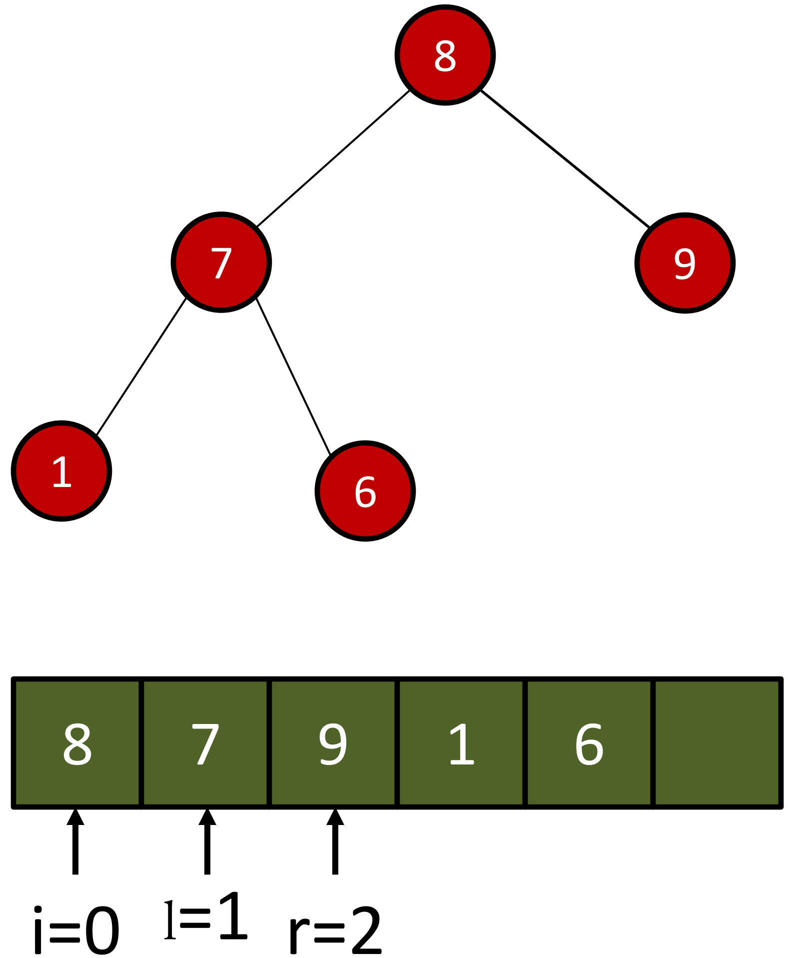
  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)



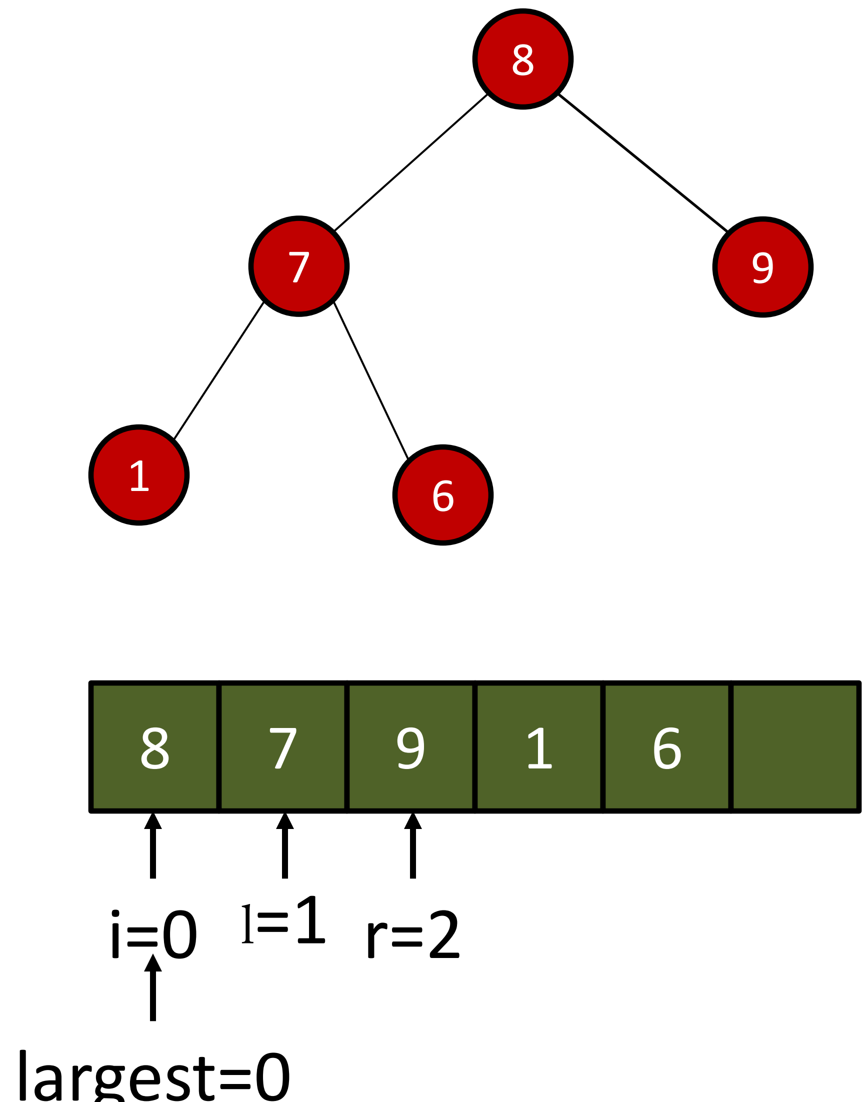
A[l] not > A[i]

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



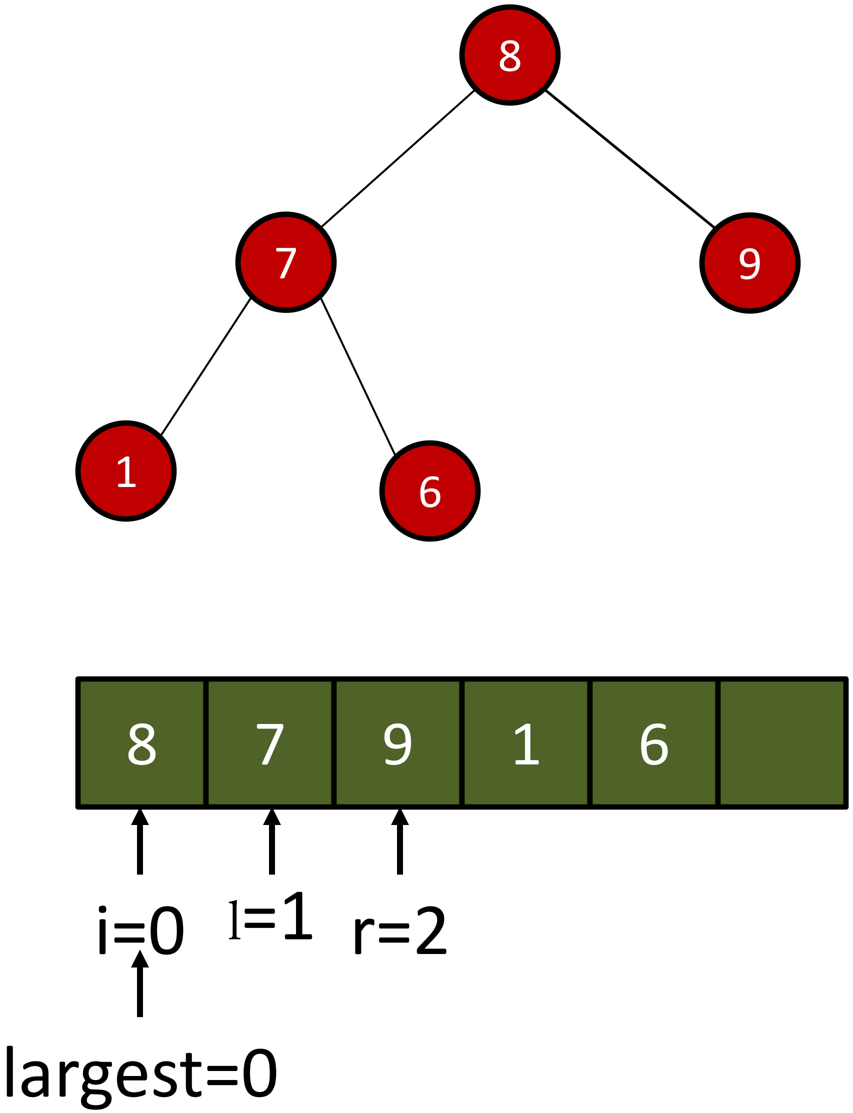
So largest = i

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



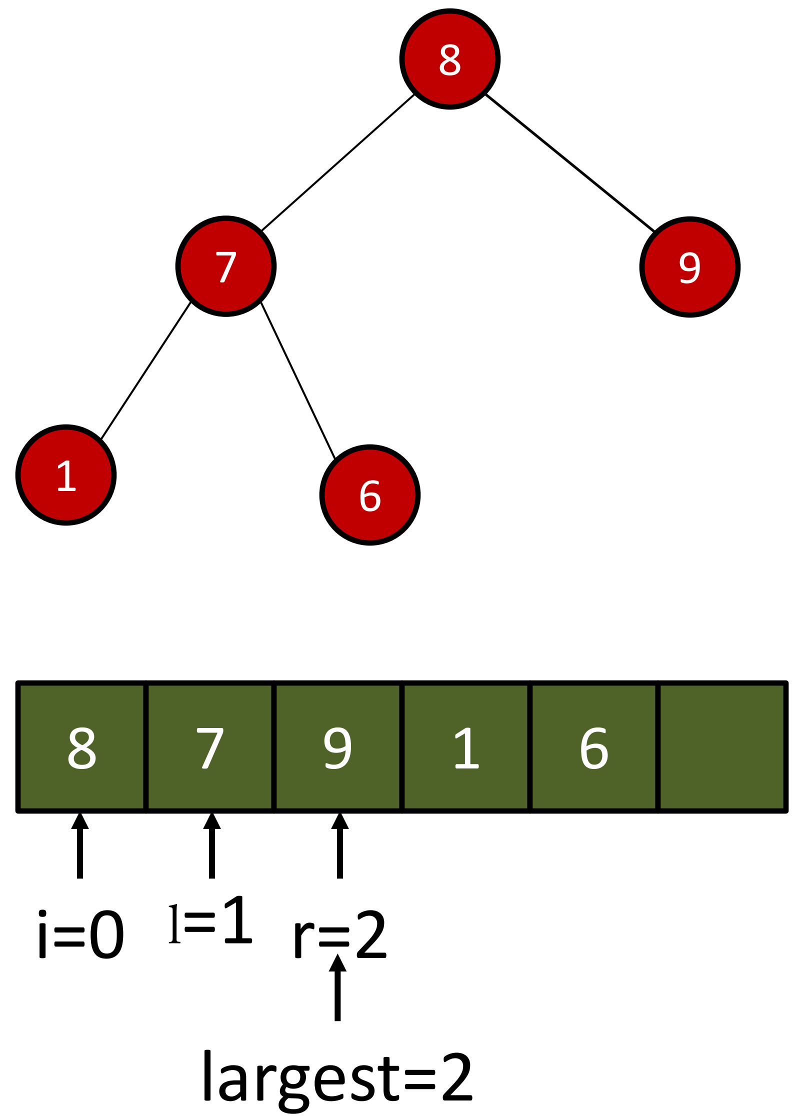
A[r] > A[largest]

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



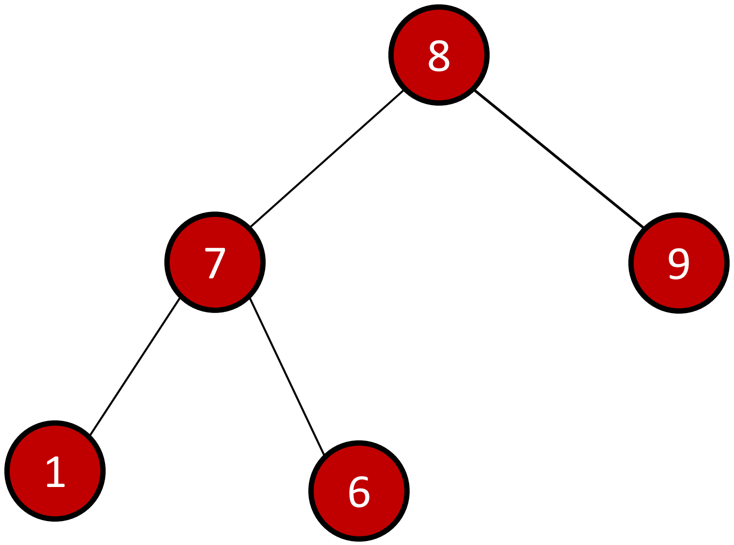
So update largest

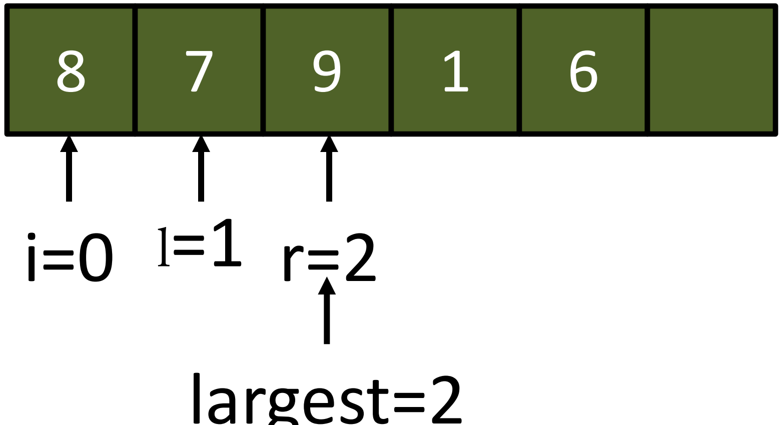
```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



largest isn't i

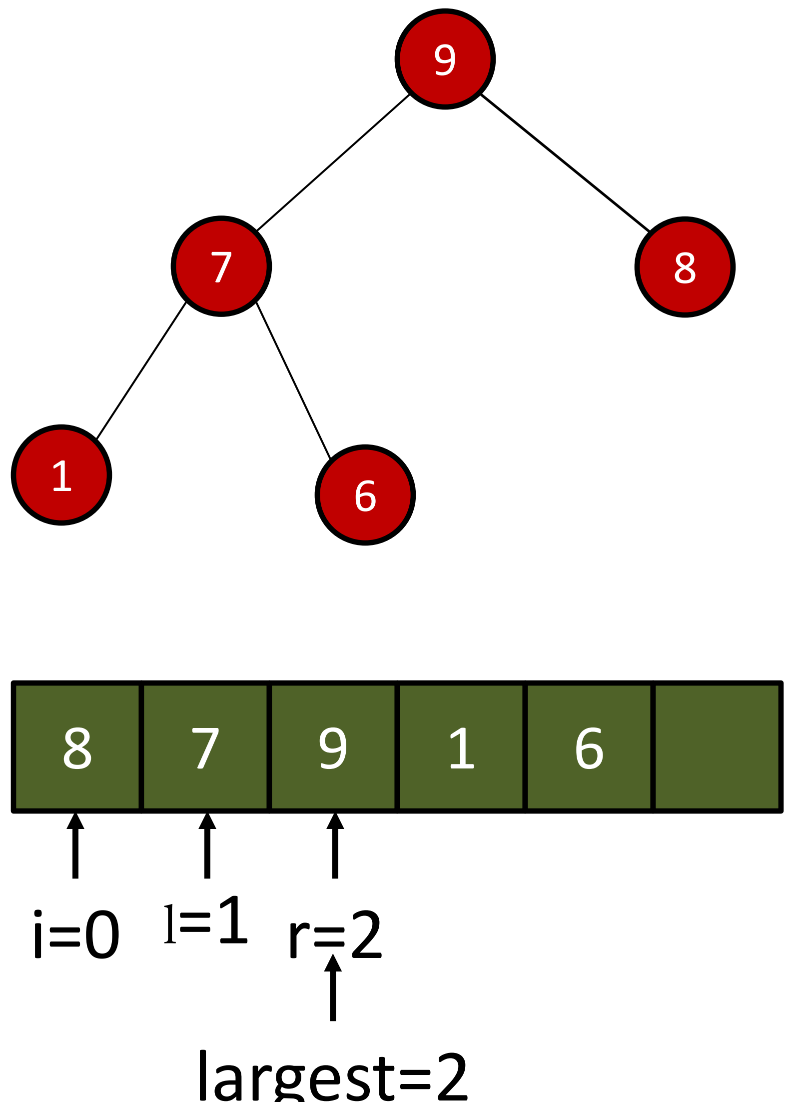
```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```





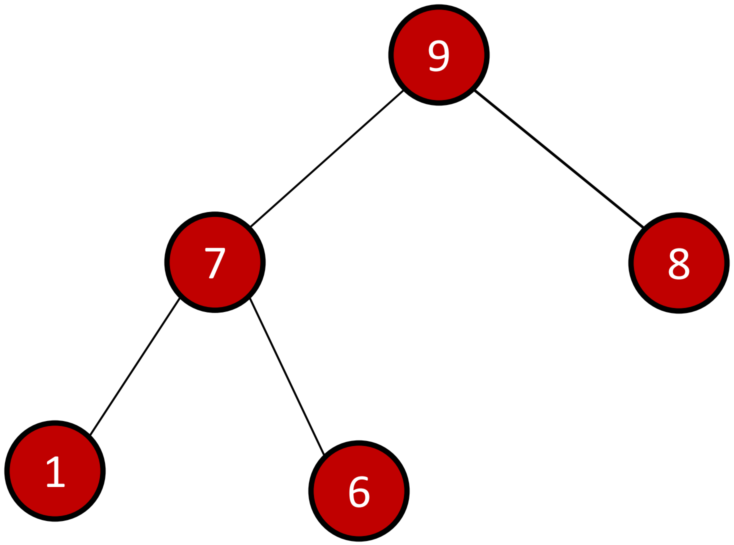
So swap with i

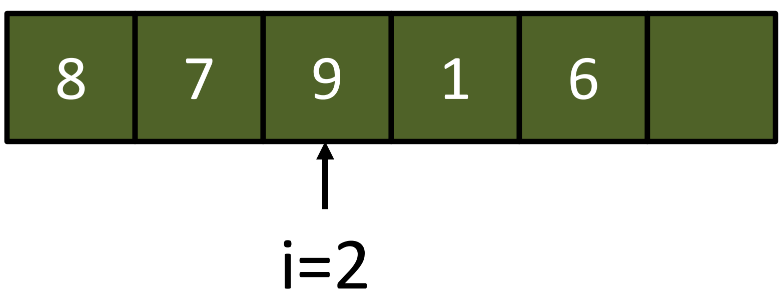
```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



And recurse

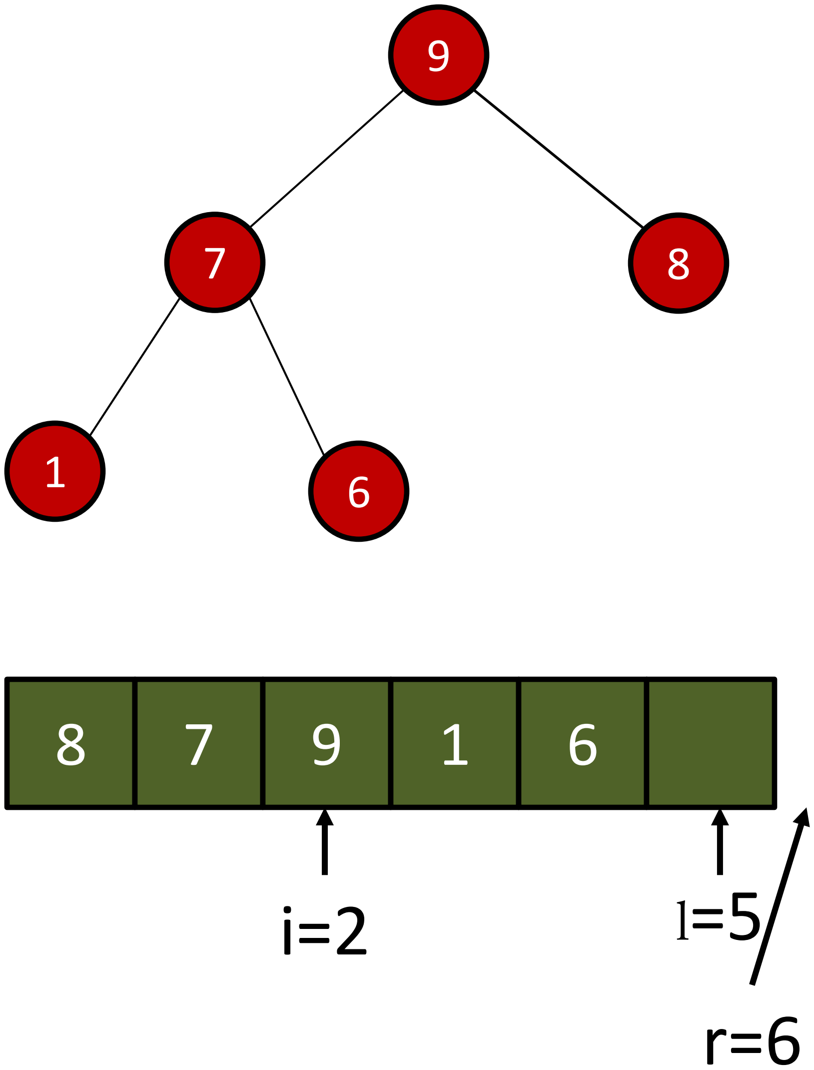
```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```





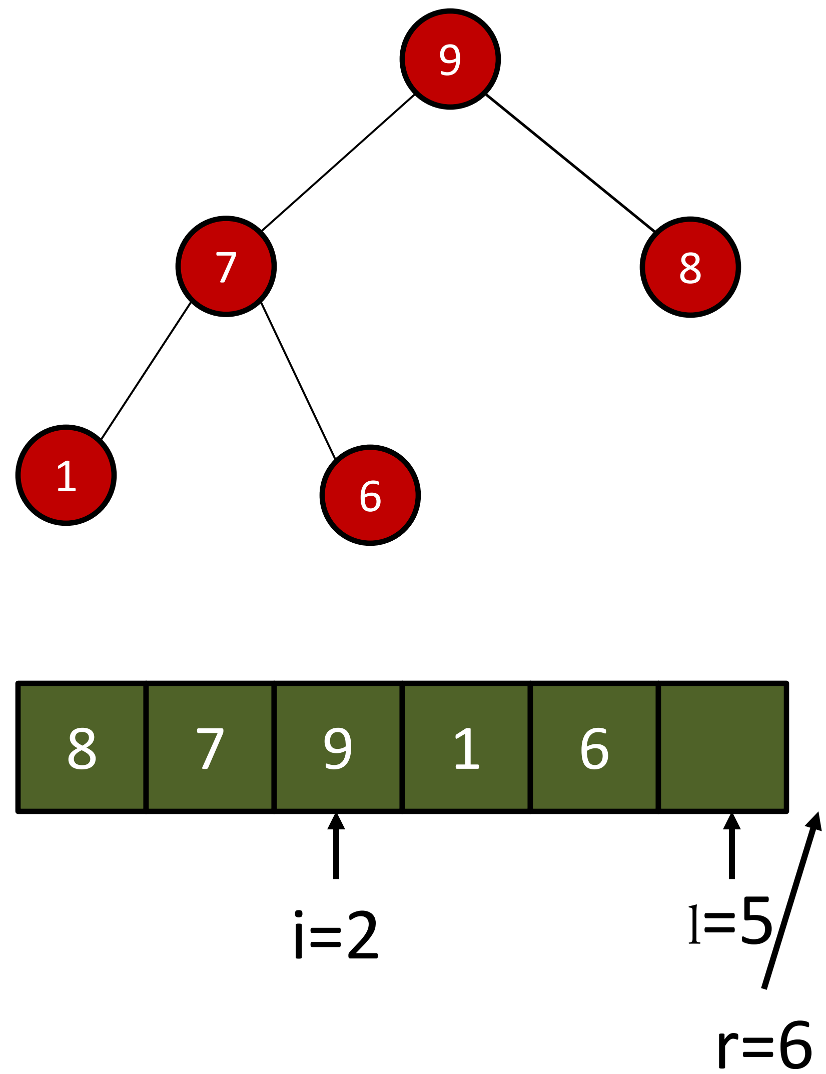
Find children

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



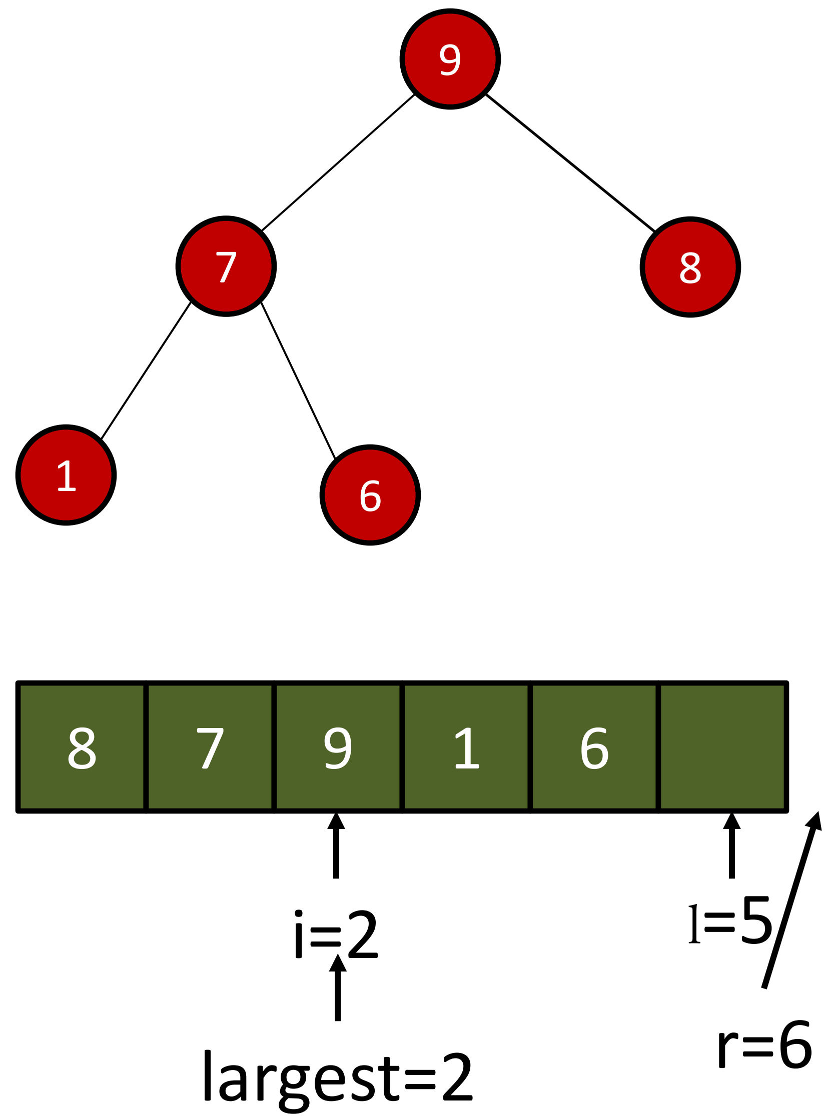
l is off the end of the heap

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



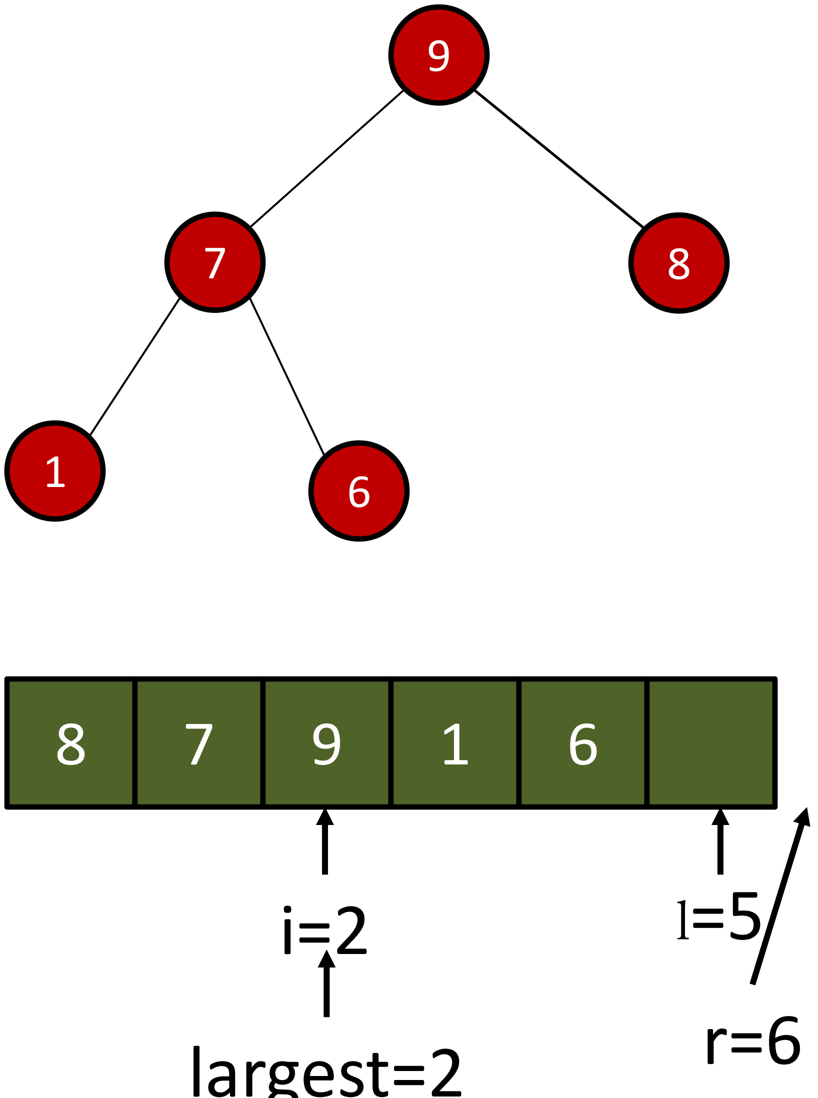
So largest is i

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



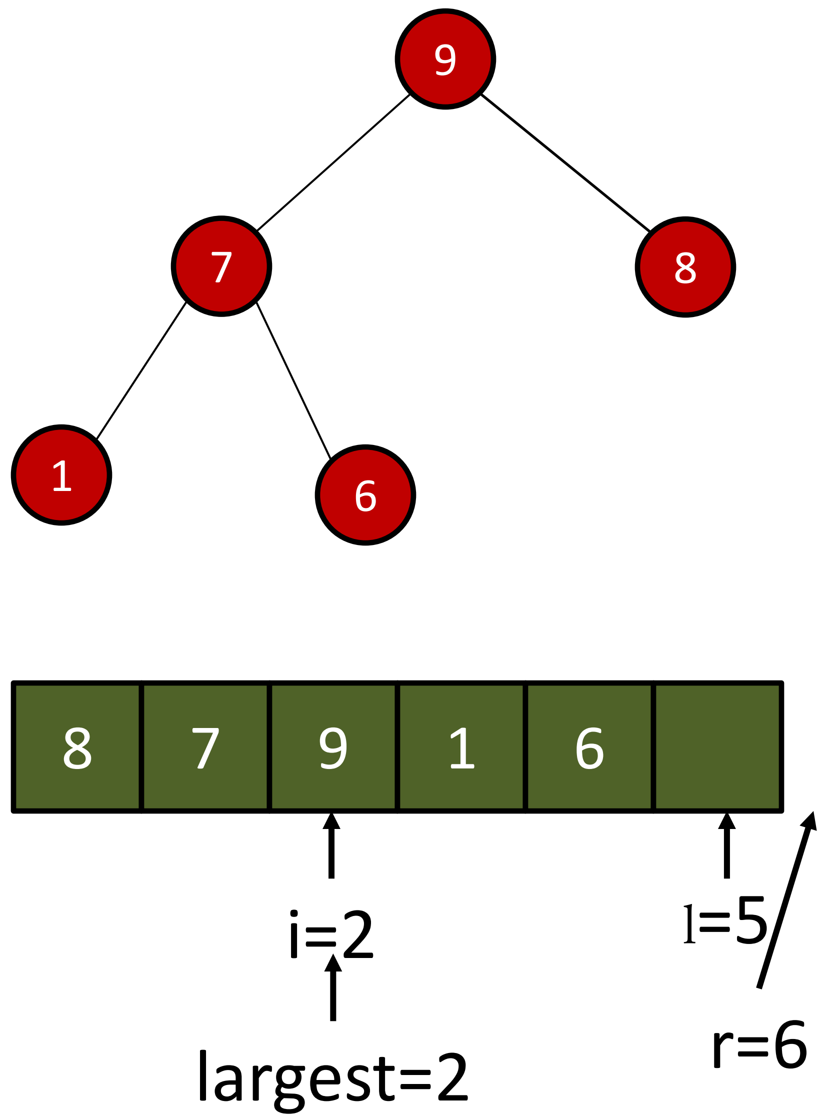
r is also off the end of the heap

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



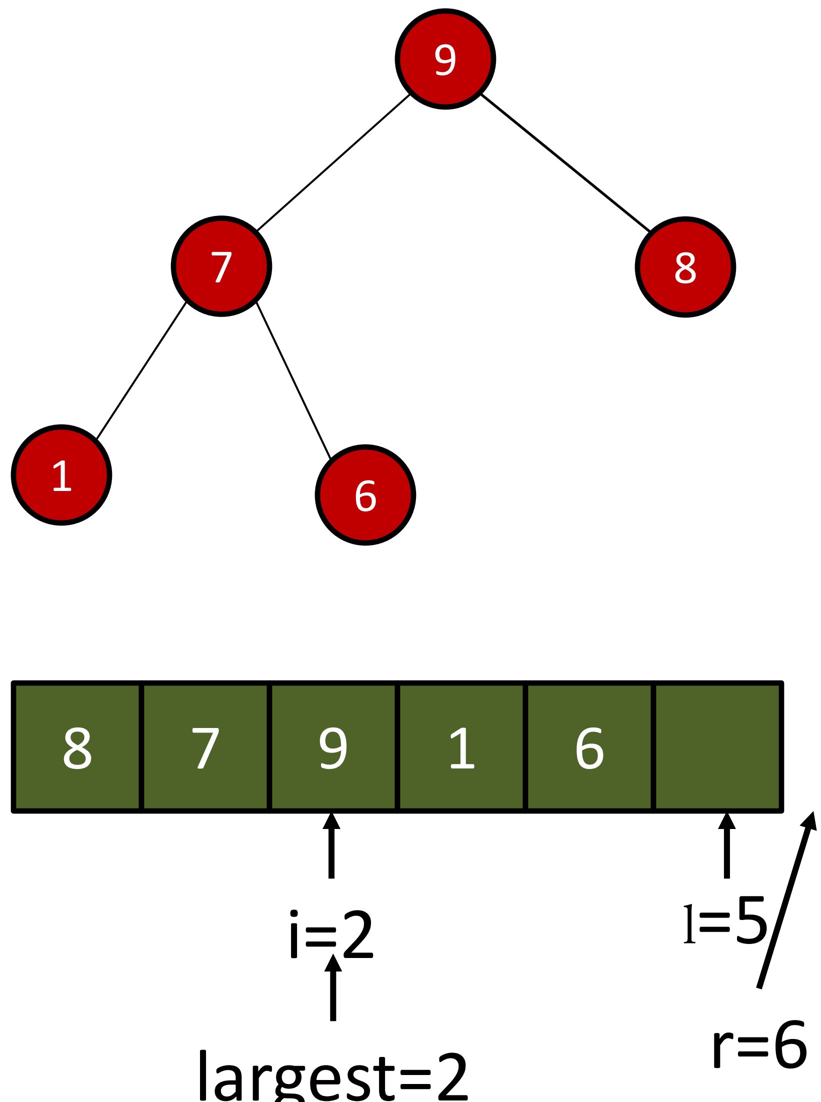
largest=i

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



So we're done

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



next time:
applications of binary heaps