

# Algorithms

## Priority queue (Heap) (Ch. 6)

Morgan Ericsson

# Today

- » Priority queues
- » Heaps
  - » Binary
  - » d-Heaps
- » Applications
  - » Sorting

# Priority Queues

# Collections

- » We have seen several examples of collections with insert and delete operations
  - » Which element should be deleted?
- » So far
  - » Stack, remove last added
  - » Queue, remove first added
  - » Randomized queue, remove random

# Priority queue

- » Remove the largest or smallest item
- » Can be used to find the “thing” with the highest (or lowest) priority
  - » Compared to FIFO ordering
- » Imaging a print queue
  - » FIFO, all jobs have the same priority
  - » PQ, some jobs are more important than others
- » Several other applications

# Priority queue

- » The operations are similar to what we have seen in other collections
- » But, we add *max* and *remove-max* (or *min*, depending on our goal)

# Implementation

- » Assume we use a list to implement a priority queue
- » We need ordered elements
- » This means either insert or remove must be  $O(n)$ 
  - » The other can be  $O(1)$
- » For example, if we insert first (or last)
  - » Insert is  $O(1)$
  - » Max is  $O(n)$

# Binary search tree

- » We know where to find the max (or min) in a BST
- » On average  $O(\log_2 n)$ , so better than list
- » Will a non-balanced tree to ok?



# Implementation (re-used from Week 4)

```
1  from dataclasses import dataclass
2
3  @dataclass
4  class BTreeNode:
5      key: int
6      left: 'BTreeNode|None' = None
7      right: 'BTreeNode|None' = None
8
9  class BST:
10     def __init__(self) -> None:
11         self.root = None
```

# Implementation (re-used from Week 4)

```
1  from fastcore.basics import patch
2
3  @patch
4  def add(self:BST, key:int) -> None:
5      self.root = self._add(self.root, key)
6
7  @patch
8  def _add(self:BST, n:BTNode|None, key:int) -> BTNode|None:
9      if n is None:
10         return BTNode(key)
11
12     if n.key > key:
13         n.left = self._add(n.left, key)
14     elif n.key < key:
15         n.right = self._add(n.right, key)
16
17     return n
```

# Implementation (re-used from Week 4)

```
1 @patch
2 def delete(self:BST, key:int) -> None:
3     self.root = self._delete(self.root, key)
4
5 @patch
6 def _delete(self:BST, n:BTNode|None, key:int) -> BTNode|None:
7     if n is None:
8         return None
9     if n.key > key:
10         n.left = self._delete(n.left, key)
11     elif n.key < key:
12         n.right = self._delete(n.right, key)
13     else:
14         if n.right is None:
15             return n.left
16         if n.left is None:
17             return n.right
18         n.key = self._min(n.right)
19         n.right = self._delete(n.right, n.key)
20     return n
```

# Implementation (re-used from Week 4)

```
1 @patch
2 def _min(self:BST, n:BTNode) -> int:
3     if n.left is None:
4         return n.key
5     else:
6         return self._min(n.left)
```

# Max and remove max

```
1  @patch
2  def max(self:BST) -> int:
3      return self._max(self.root)
4
5  @patch
6  def _max(self:BST, n:BTNode) -> int:
7      if n.right is None:
8          return n.key
9      else:
10         return self._max(n.right)
```

# Max and remove max

```
1 @patch
2 def delMax(self:BST) -> int:
3     mx = self._max(self.root)
4     self.root = self._delete(self.root, mx)
5
6     return mx
```

# Testing it

```
1 import random
2
3 sq = random.sample(range(1, 10_001), k=500)
4
5 t = BST()
6 for v in sq:
7     t.add(v)
8
9 r1 = [t.delMax() for _ in range(5)]
10 r2 = sorted(sq, reverse=True)[:5]
11
12 assert r1 == r2
```

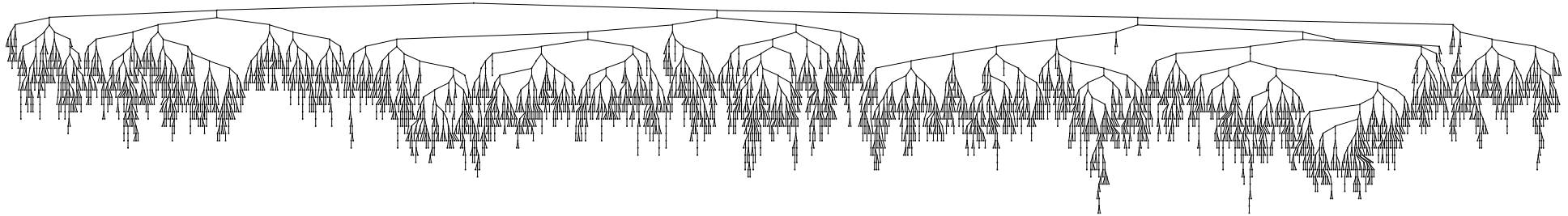
# Solved?

- » Yes and no
- » We can use BSTs to implement priority queues
- » Do we know how balanced the BSTs are?
  - » We could always use AVL or Splay
- » There is a simpler way



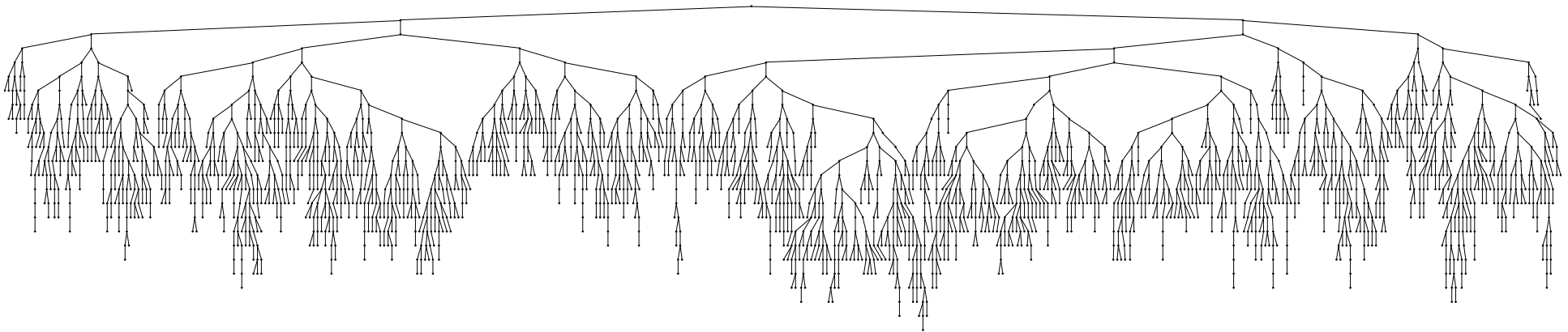
# Trees (before)

Inserting 5 000 random values (no dups)



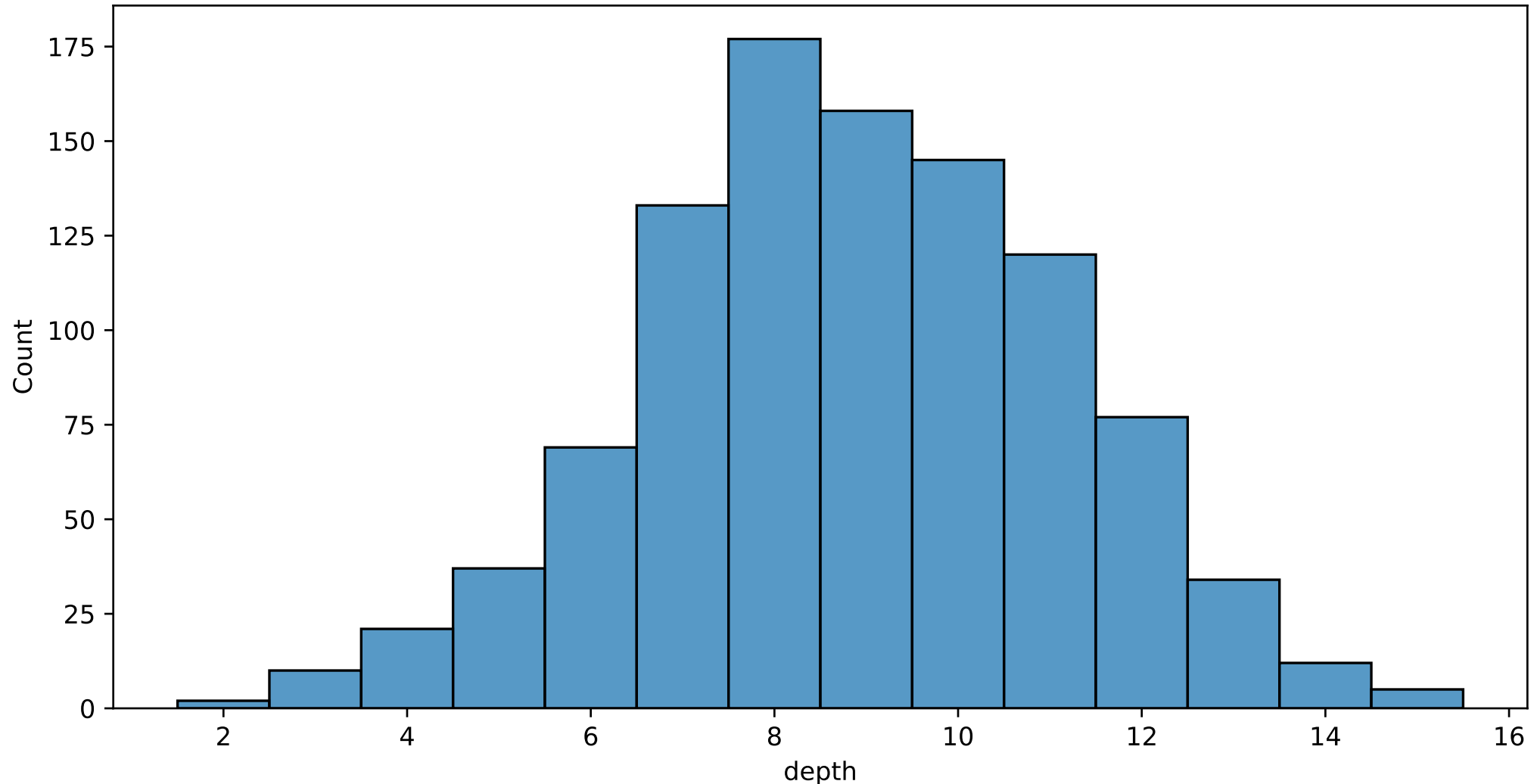
# Trees (after)

After 2 500 calls to **delMax**



# Average depth

**10 000 trees, each with 5 000 nodes (height  $\geq 12$ )**



# Heaps

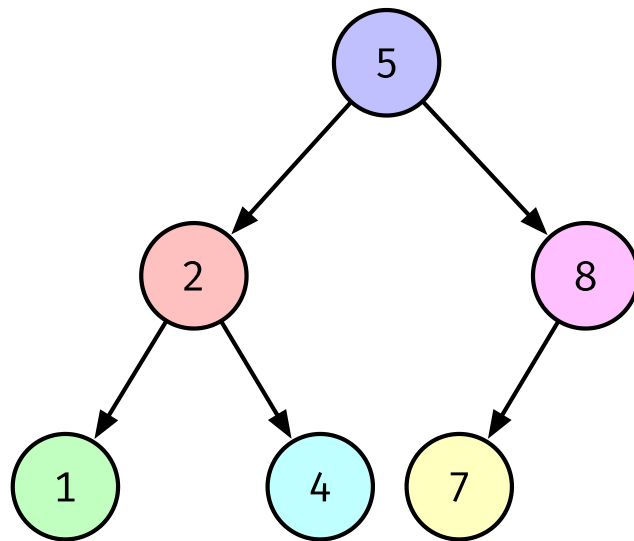
# Binary Heaps

- » A binary heap is an almost complete binary tree
- » That satisfies the heap property

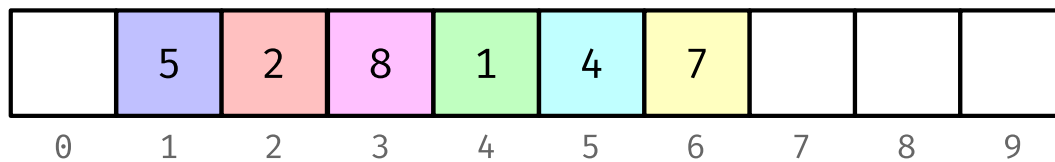
# Binary trees and arrays

- » We can store a binary tree in an array
  - » Removes the need for links
- » The tree is stored level by level
  - » Starting with the root
- » We use 1-based indexing

# Example



Children to Node  $k$   
is at  $2k$  and  $2k+1$



# Implementation

```
1 class BTAarray:  
2     def __init__(self, cap:int = 15) -> None:  
3         self.cap = cap + 1  
4         self.bt = [None] * self.cap
```



# Implementation

```
1 @patch
2 def add(self:BTArray, key:int) -> None:
3     self._add(1, key)
4
5 @patch
6 def _add(self:BTArray, n:int, key:int) -> None:
7     if self.bt[n] == None:
8         self.bt[n] = key
9         return
10
11     if self.bt[n] > key:
12         return self._add(2 * n, key)
13     elif self.bt[n] < key:
14         return self._add(2 * n + 1, key)
```

# Testing it

```
1 t = BTAarray(8)
2 t.add(5)
3 t.add(2)
4 t.add(8)
5 t.add(1)
6 t.add(4)
7 t.add(7)
8
9 print(t.bt)
```

[None, 5, 2, 8, 1, 4, 7, None, None]

# Binary heap

- » Almost complete binary tree stored in an array
- » Heap condition
  - » Depends on what kind of priority queue we implement
- » Parent's key is no smaller (larger) than the children's keys
- » So, largest (smallest) key is at the root / index 1

# A priority queue (max)

```
1 class PQMx:
2     def __init__(self, cap:int = 16) -> None:
3         self.cap = cap + 1
4         self.h = [None] * self.cap
5         self.sz = 0
```

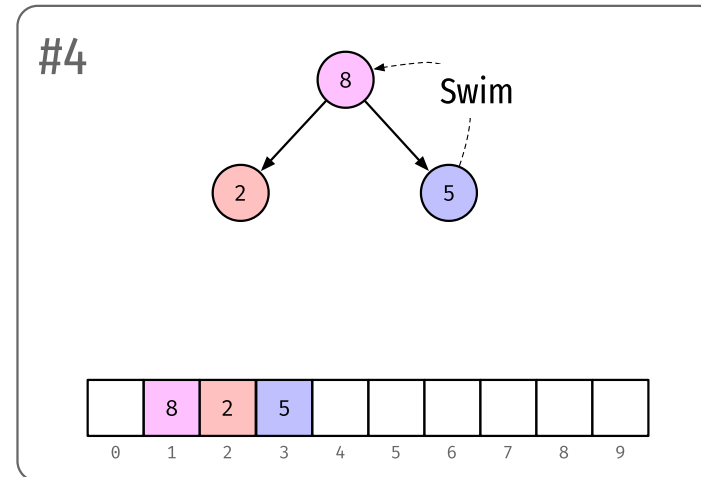
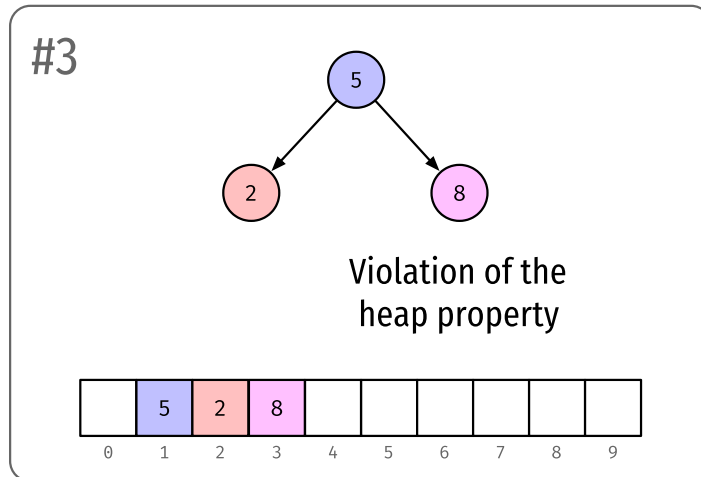
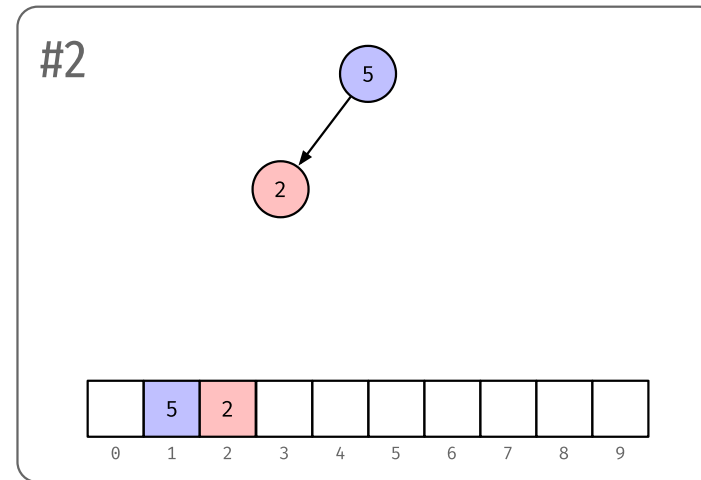
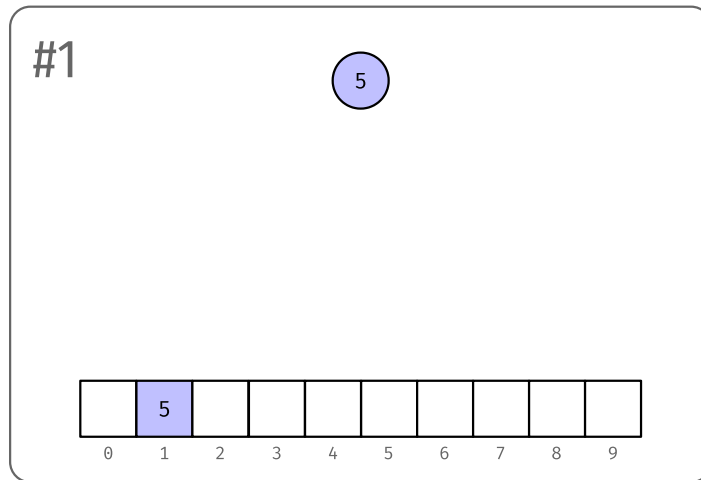
# A priority queue (max)

```
1 @patch
2 def insert(self:PQMx, key:int) -> None:
3     self.sz += 1
4     self.h[self.sz] = key
5     self._swim(self.sz)
```

# Swim?

- » If a child's key becomes larger than a parent's key
  - » It needs to swim / bubble up
- » Exchange key in child with key in parent
  - » Until the heap property is restored

# Swim



# Swim

```
1 @patch
2 def _swim(self:PQMx, k:int) -> None:
3     while k > 1 and self.h[k // 2] < self.h[k]:
4         self.h[k], self.h[k // 2] = \
5             self.h[k // 2], self.h[k]
6         k = k // 2
```



# Getting the max

```
1  @patch
2  def delMax(self:PQMx) -> int|None:
3      if self.sz > 0:
4          mx = self.h[1]
5          self.h[1], self.h[self.sz] = \
6              self.h[self.sz], self.h[1]
7          self.sz -= 1
8          self._sink(1)
9          self.h[self.sz + 1] = None
10
11      return mx
```

# Sink?

- » Opposite of swim
- » If a parent's key becomes smaller than one or both of the children's
  - » It should sink / bubble down
- » Exchange key in parent with key in larger child
  - » Until heap property is restored

# Sink

```
1  @patch
2  def _sink(self:PQMx, k:int) -> None:
3      while 2 * k <= self.sz:
4          j = 2 * k
5          if j < self.sz and self.h[j] < self.h[j + 1]:
6              j += 1
7
8          if not self.h[k] < self.h[j]:
9              break
10
11         self.h[k], self.h[j] = \
12             self.h[j], self.h[k]
13         k = j
```

# Adding a print

```
1 @patch
2 def printQ(self:PQMx) -> None:
3     for i in range(1, self.sz+1):
4         print(self.h[i])
```

# Testing simple function

```
1 pq = PQMx(8)
2 pq.insert(5)
3 pq.insert(2)
4 pq.insert(8)
5 pq.insert(1)
6 pq.insert(4)
7 pq.insert(7)
8
9 pq.printQ()
```

8  
4  
7  
1  
2  
5

# And with the previous example

```
1 pq = PQMx(500)
2 for v in sq:
3     pq.insert(v)
4
5 r1 = [pq.delMax() for _ in range(5)]
6 r2 = sorted(sq, reverse=True)[:5]
7
8 assert r1 == r2
```

# Analysis

- » The “expensive” operations are *sink* and *swim*
- » *swim* requires at most  $1 + \log_2 n$  compares
- » And *sink*, at most  $2 \cdot \log_2 n$  compares
- » So, *insert* and *delMax* are  $O(\log_2 n)$

# Open issues

- » Min heap is easy, just swap from  $>$  to  $<$
- » We should use immutable types for the keys in the heap
  - » Keys should not change while in the heap
- » We should consider growing the heap if it becomes full
  - » Just like we did the array-based list



# d-Heaps

- » The tree does not have to be binary
- » We can have 3, 4 or more children per node and still use it as a heap
- » The more children, the shallower the tree is
- » So, find becomes cheaper
  - » still  $O(\log n)$ , just that  $\log_3 1000$  is smaller than  $\log_2 1000$
- » And delete becomes more expensive ( $O(d \log_d n)$ )

# Applications

# Sorting

- » We can, unsurprisingly, use heaps to sort
- » Simplest case, we simply fill the heap and use `delMax` to populate the list

# Idea

```
1 rr = range(1000)
2 sq = random.sample(rr, k=16)
3 assert sq != sorted(sq)
4 pq = PQMx(16)
5 for v in sq:
6     pq.insert(v)
7
8 l = [0]*16
9 for i in range(15,-1,-1):
10     l[i] = pq.delMax()
11
12 assert l == sorted(sq)
```

# Sorting

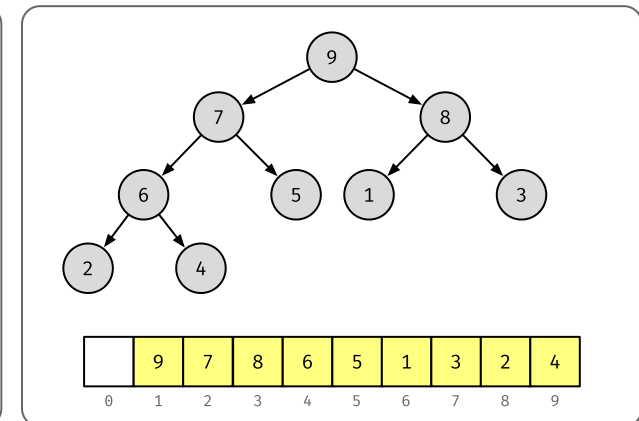
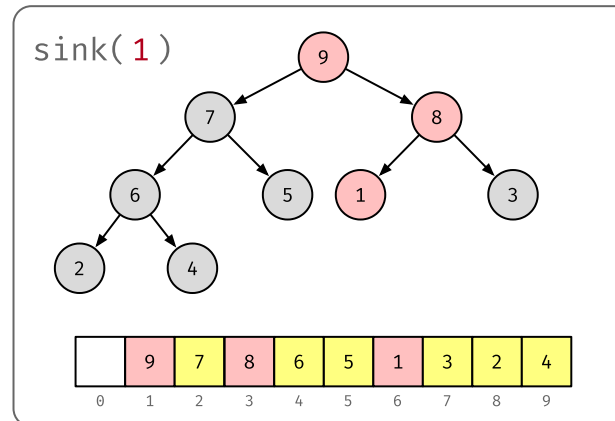
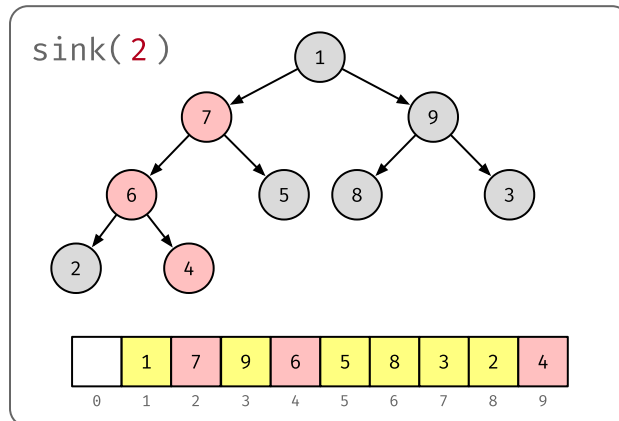
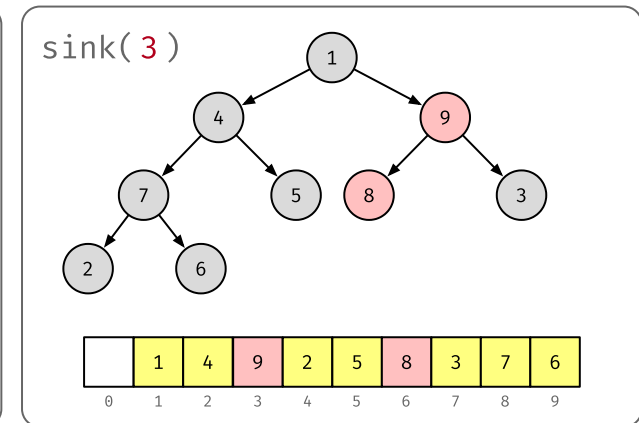
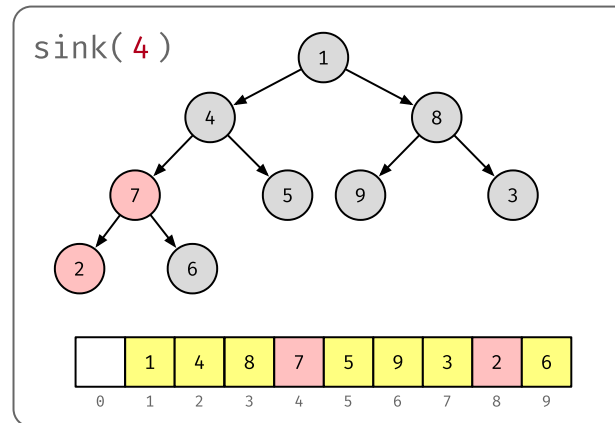
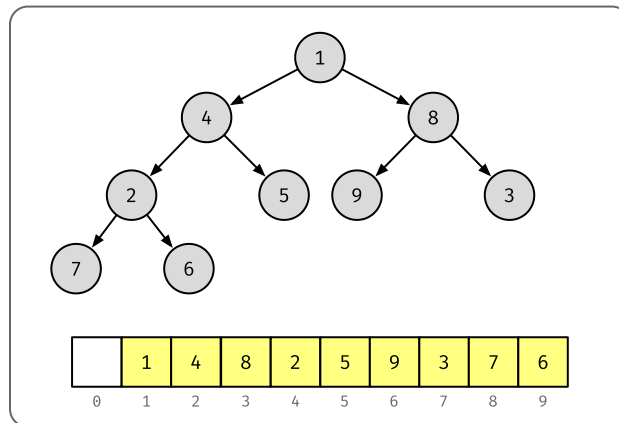
- » Works, but we can do better
- » Current method required  $2n$  space
- » We can do it in-place

# Heap sort

- » Build a max heap
- » Remove the max  $n$  times, but leave it in the array

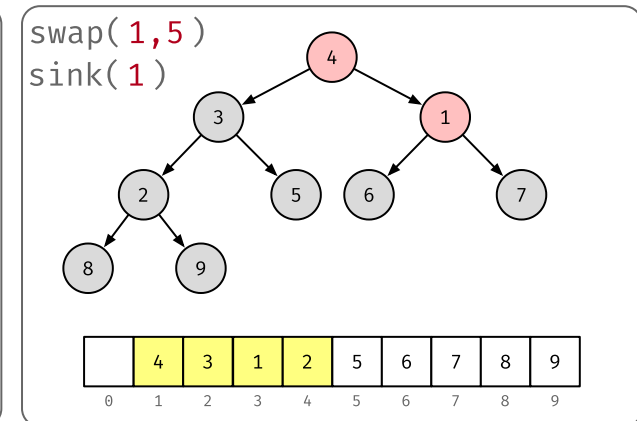
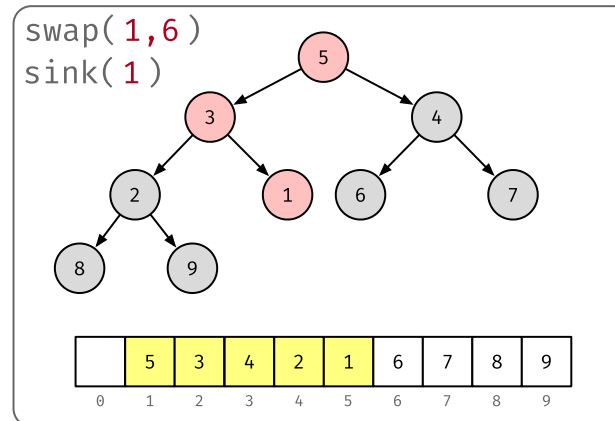
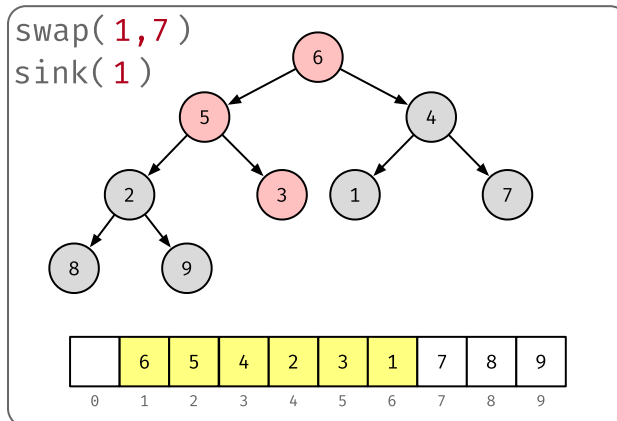
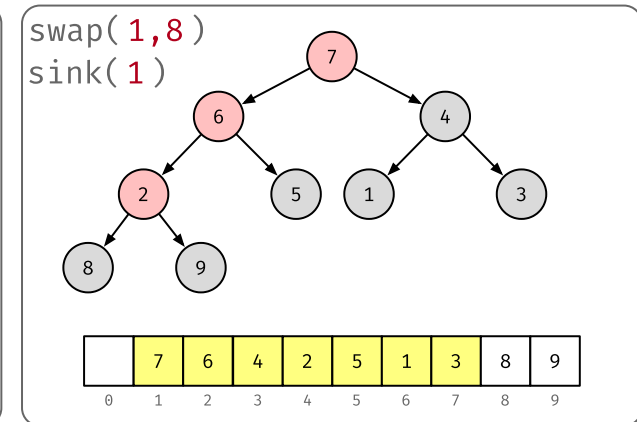
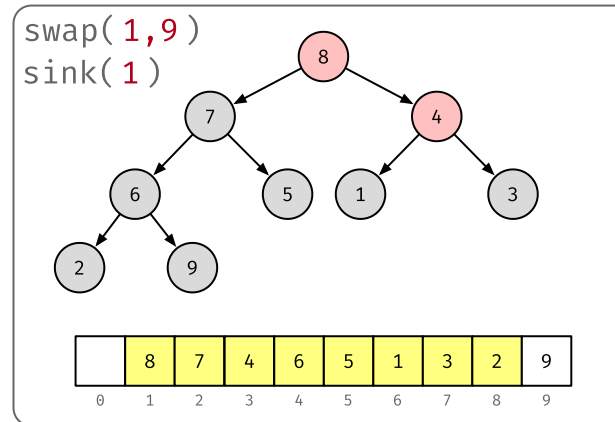
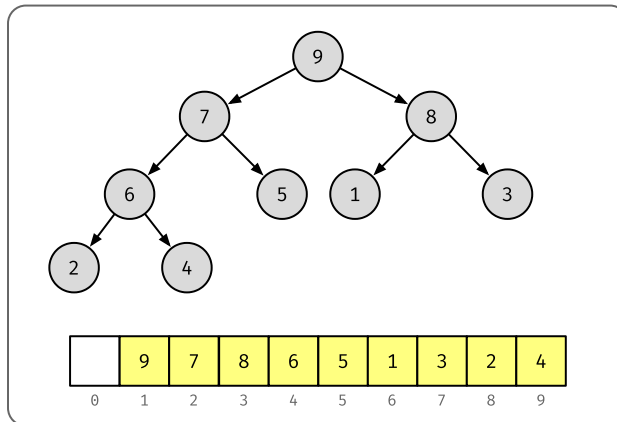
# The idea

## Build the heap bottom-up



# The idea

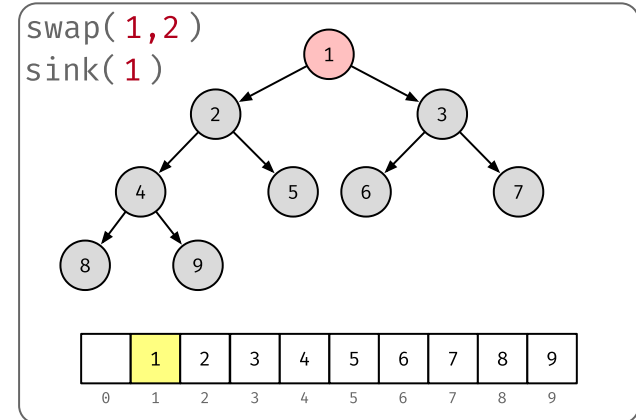
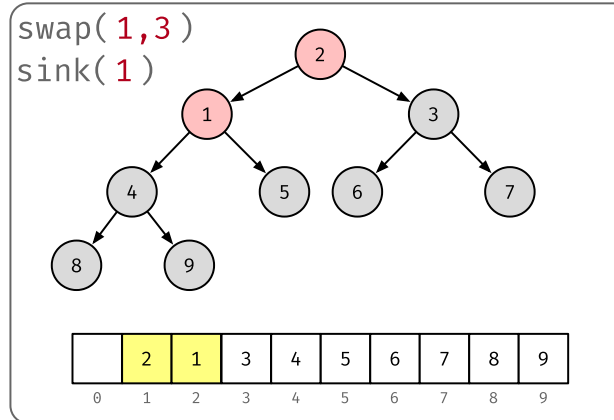
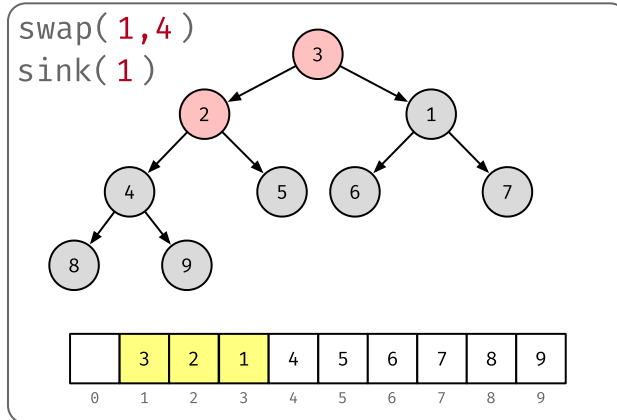
## Sort down (remove max)





# The idea

## Sort down (remove max)



# Implementation

```
1 class HeapSort:
2     def __init__(self, l) -> None:
3         self.h = [None] + l
4         self.sz = len(self.h) - 1
```

# Implementation

```
1  @patch
2  def _sink(self:HeapSort, k:int) -> None:
3      while 2 * k <= self.sz:
4          j = 2 * k
5          if j < self.sz and self.h[j] < self.h[j + 1]:
6              j += 1
7
8          if not self.h[k] < self.h[j]:
9              break
10
11         self.h[k], self.h[j] = \
12             self.h[j], self.h[k]
13         k = j
```

# Implementation

```
1  @patch
2  def sort(self:HeapSort) -> None:
3      k = self.sz // 2
4      while k >= 1:
5          self._sink(k)
6          k -= 1
7
8      while self.sz > 1:
9          self.h[1], self.h[self.sz] = \
10             self.h[self.sz], self.h[1]
11         self.sz -= 1
12         self._sink(1)
```

# Testing it

```
1 r1 = [1, 4, 8, 2, 5, 9, 3, 7, 6]
2 hs = HeapSort(r1)
3 print('orig:',hs.h[1:])
4 hs.sort()
5 print('sort:',hs.h[1:])
```

orig: [1, 4, 8, 2, 5, 9, 3, 7, 6]

sort: [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Analysis

- »  $2n \log n$  best and average case
- »  $O(n \log n)$
- » Good sorting algorithm, but a few practical drawbacks
  - » (More about that next time)

# Reading instructions

# Reading instructions

- » Ch. 6.1 - 6.5
- » Ch. 6.9 (for priority queues in Java)
- » Ch. 7.5 (heapsort)