Algorithms and Data Structures

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)

```python
from dataclasses import dataclass

@dataclass
class BTNode:
  key: int
  left: 'BTNode|None' = None
  right: 'BTNode|None' = None

class BST:
  def __init__(self) -> None:
    self.root = None
```

Implementation (re-used from Week 4)

```python
from fastcore.basics import patch

@patch
def add(self:BST, key:int) -> None:
  self.root = self._add(self.root, key)

@patch
def _add(self:BST, n:BTNode|None, key:int) -> BTNode|None:
  if n is None:
    return BTNode(key)

  if n.key > key:
    n.left = self._add(n.left, key)
  elif n.key < key:
    n.right = self._add(n.right, key)

  return n
```

Implementation (re-used from Week 4)

```
@patch
def delete(self:BST, key:int) -> None:
  self.root = self._delete(self.root, key)

@patch
def _delete(self:BST, n:BTNode|None, key:int) -> BTNode|None:
  if n is None:
    return None
  if n.key > key:
    n.left = self._delete(n.left, key)
  elif n.key < key:
    n.right = self._delete(n.right, key)
  else:
    if n.right is None:
      return n.left
    if n.left is None:
      return n.right
    n.key = self._min(n.right)
    n.right = self._delete(n.right, n.key)
  return n
```

Implementation (re-used from Week 4)

```python
@patch
def _min(self:BST, n:BTNode) -> int:
  if n.left is None:
    return n.key
  else:
    return self._min(n.left)
```

Max and remove max

```
@patch
def max(self:BST) -> int:
  return self._max(self.root)

@patch
def _max(self:BST, n:BTNode) -> int:
  if n.right is None:
    return n.key
  else:
    return self._max(n.right)
```

Max and remove max

```
@patch
def delMax(self:BST) -> int:
  mx = self._max(self.root)
  self.root = self._delete(self.root, mx)

  return mx
```

Testing it

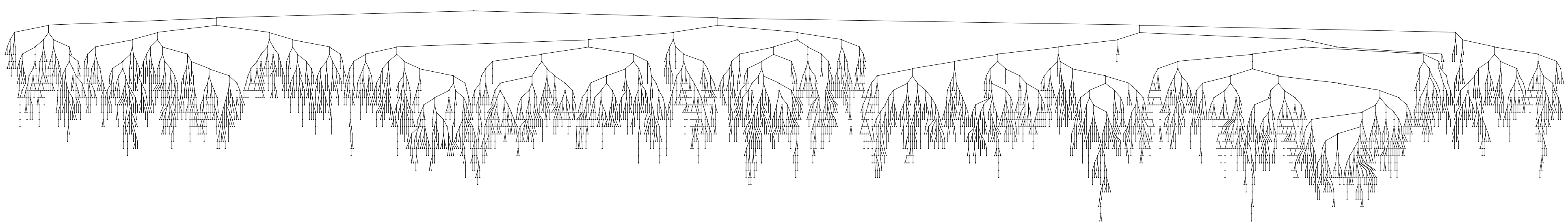
```
import random

sq = random.sample(range(1, 10_001), k=500)

t = BST()
for v in sq:
  t.add(v)

r1 = [t.delMax() for _ in range(5)]
r2 = sorted(sq, reverse=True)[:5]

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

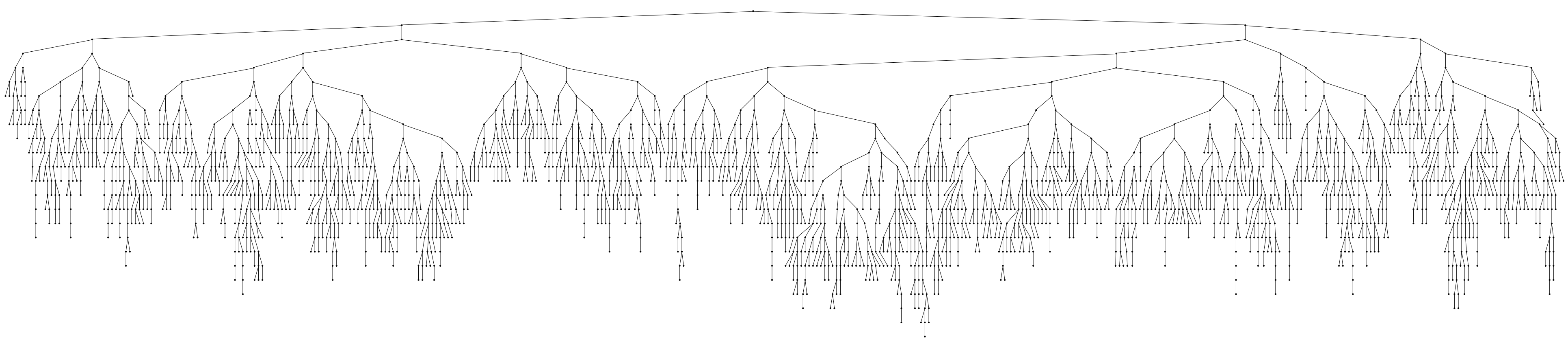
Trees (before)

Inserting 5 000 random values (no dups)



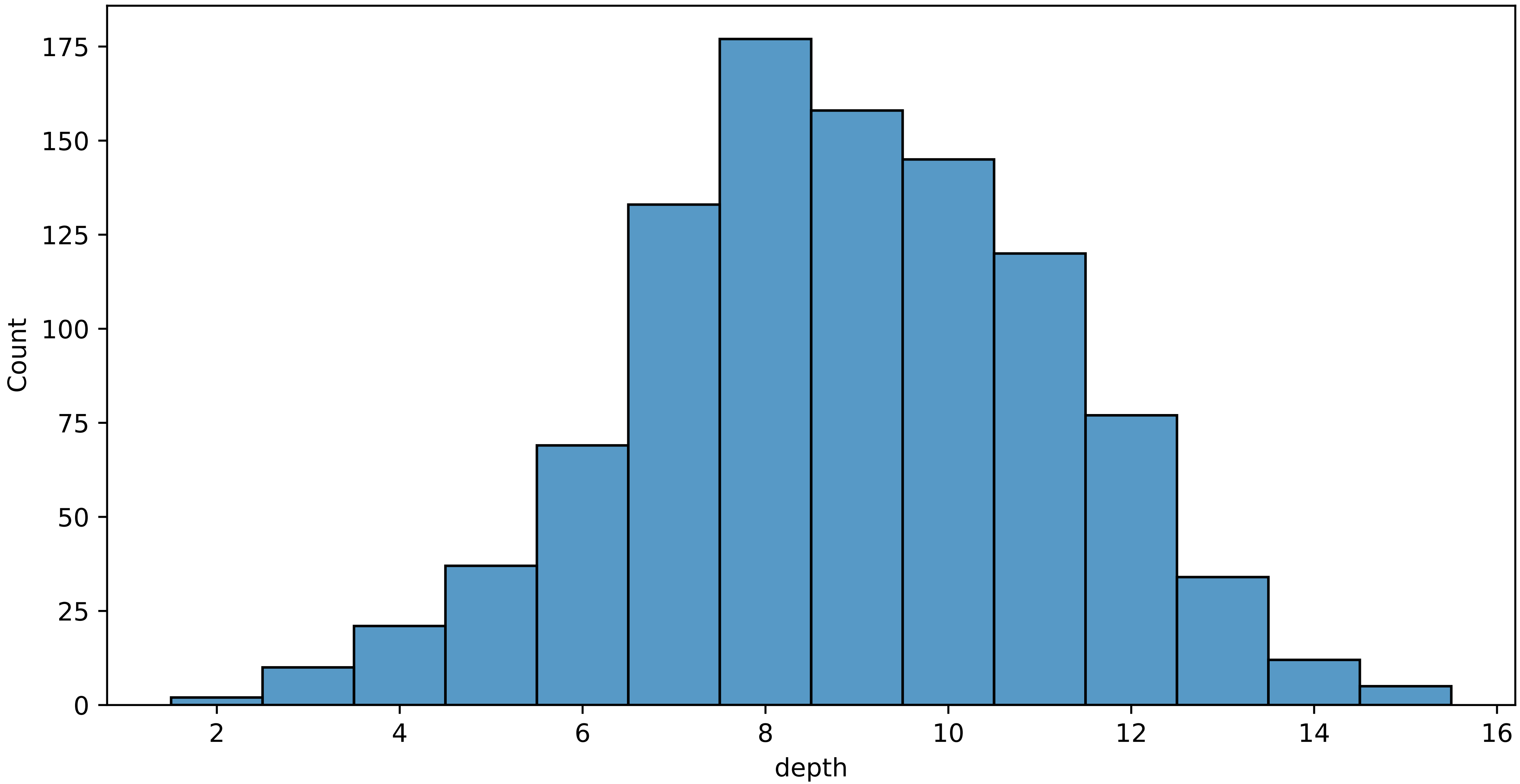
Trees (after)

After 2 500 calls to `delMax`



Average depth

10 000 trees, each with 5 000 nodes (height ≥ 12)



Heaps

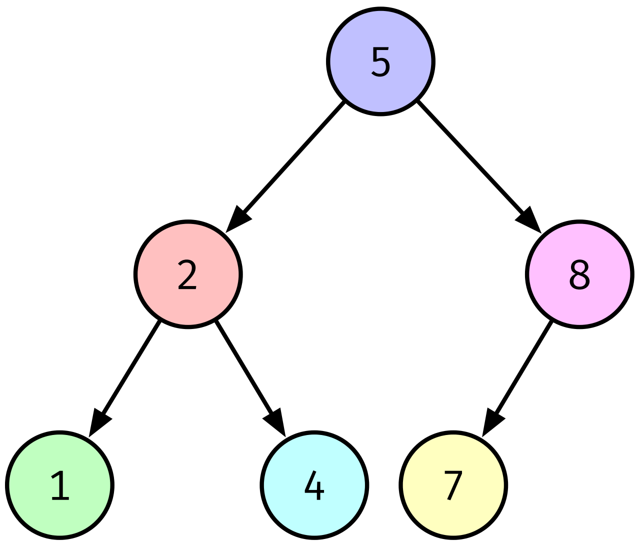
Binary Heaps

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

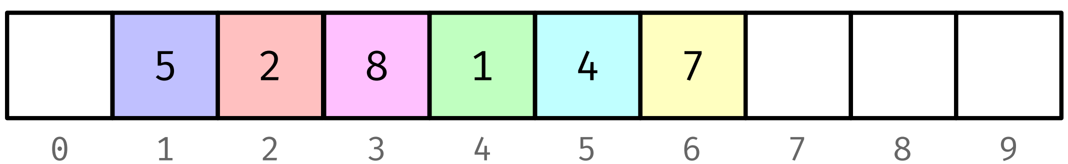
Binary trees and arrays

- 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
are at 2k and 2k+1



Implementation

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

Implementation

```python
@patch
def add(self:BTArray, key:int) -> None:
  self._add(1, key)

@patch
def _add(self:BTArray, n:int, key:int) -> None:
  if self.bt[n] == None:
    self.bt[n] = key
    return

  if self.bt[n] > key:
    return self._add(2 * n, key)
  elif self.bt[n] < key:
    return self._add(2 * n + 1, key)
```

Testing it

```
t = BTArray(8)
t.add(5)
t.add(2)
t.add(8)
t.add(1)
t.add(4)
t.add(7)

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)

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

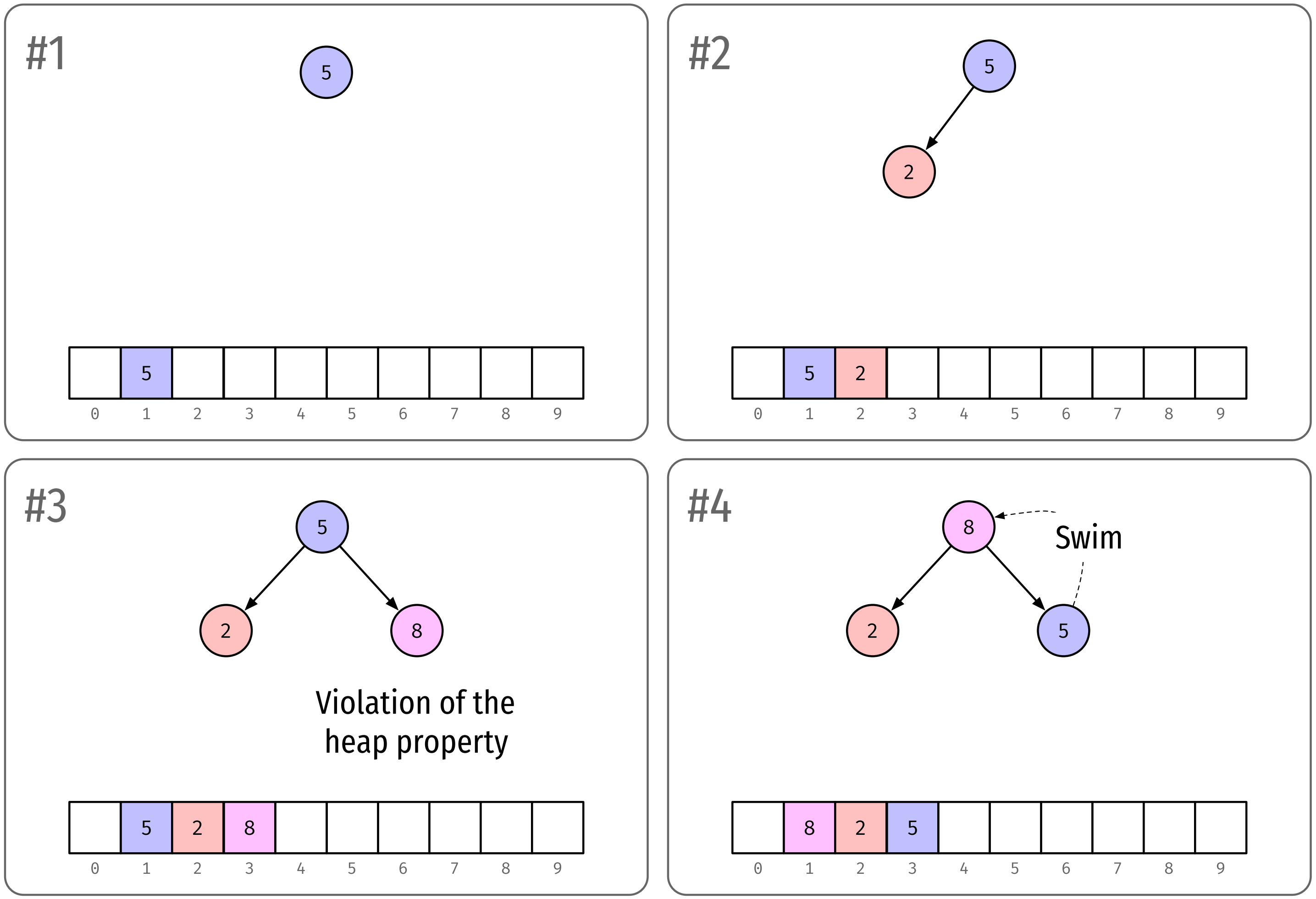
A priority queue (max)

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


#1
5
5
0 1 2 3 4 5 6 7 8 9
#2
5
2
5 2
0 1 2 3 4 5 6 7 8 9
#3
5
2
8
Violation of the heap property
5 2 8
0 1 2 3 4 5 6 7 8 9
#4
8
Swim
2
5
8 2 5
0 1 2 3 4 5 6 7 8 9

Swim

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

Getting the max

```python
@patch
def delMax(self:PQMx) -> int|None:
  if self.sz > 0:
    mx = self.h[1]
    self.h[1], self.h[self.sz] = \
                  self.h[self.sz], self.h[1]
    self.sz -= 1
    self._sink(1)
    self.h[self.sz + 1] = None

    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

```python
@patch
def _sink(self:PQMx, k:int) -> None:
  while 2 * k <= self.sz:
    j = 2 * k
    if j < self.sz and self.h[j] < self.h[j + 1]:
      j += 1

    if not self.h[k] < self.h[j]:
      break

    self.h[k], self.h[j] = \
                    self.h[j], self.h[k]
    k = j
```

Adding a print

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

Testing simple function

```
pq = PQMx(8)
pq.insert(5)
pq.insert(2)
pq.insert(8)
pq.insert(1)
pq.insert(4)
pq.insert(7)

pq.printQ()
```

```
8
4
7
1
2
5
```

And with the previous example

```
pq = PQMx(500)
for v in sq:
  pq.insert(v)

r1 = [pq.delMax() for _ in range(5)]
r2 = sorted(sq, reverse=True)[:5]

assert r1 == r2
```

Analysis

- The "expensive" operations are *sink* and *swim*
- *swim* requires at most $1 + \log_2 \mathrm{N}$ compares
- And *sink*, at most $2 \cdot \log_2 \mathrm{N}$ compares
- So, *insert* and *delMax* are $\mathrm{O}(\log_2 \mathrm{N})$

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

```python
rr = range(1000)
sq = random.sample(rr, k=16)
assert sq != sorted(sq)
pq = PQMx(16)
for v in sq:
  pq.insert(v)

l = [0]*16
for i in range(15,-1,-1):
  l[i] = pq.delMax()

assert l == sorted(sq)
```

Sorting

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

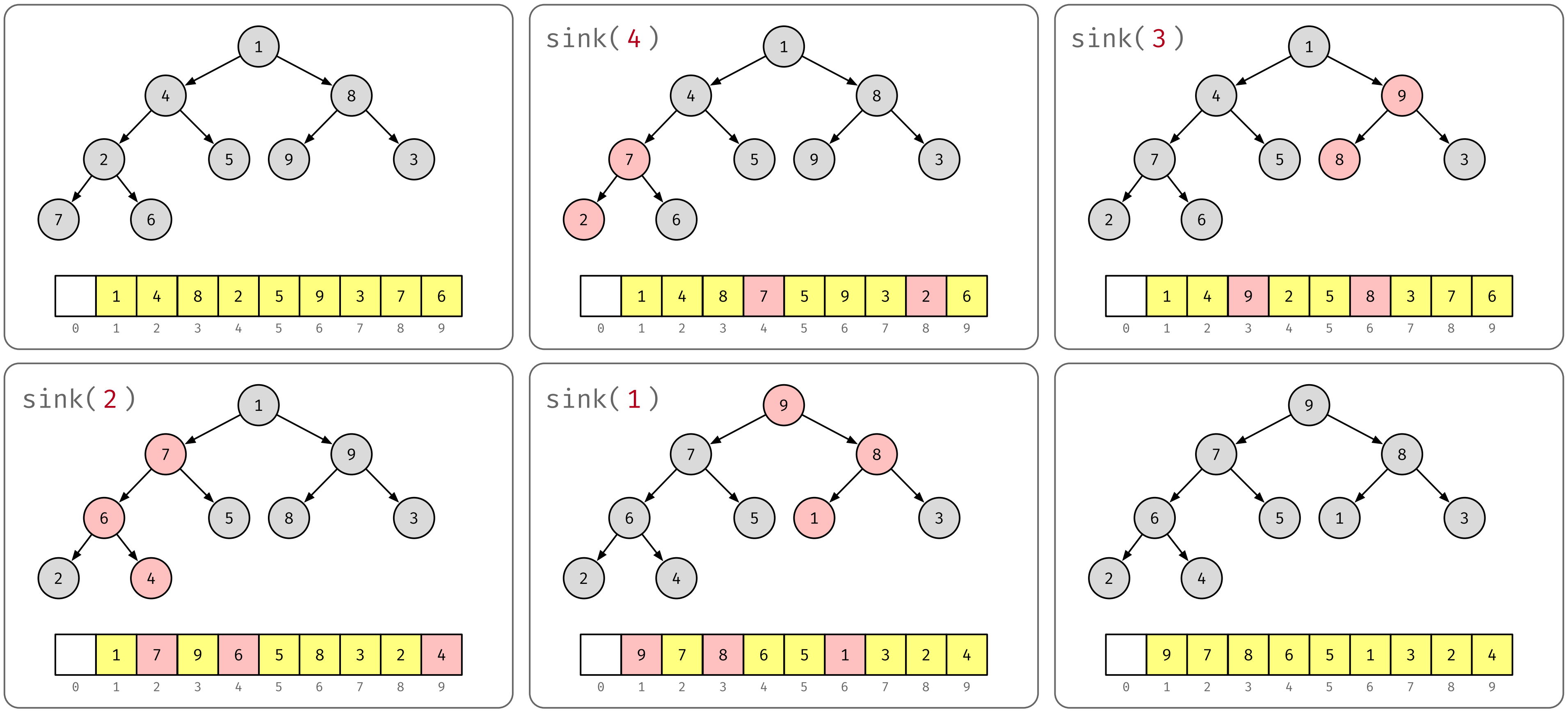
Heap sort

» Build a max heap

» Remove the max n times, but leave it in the array

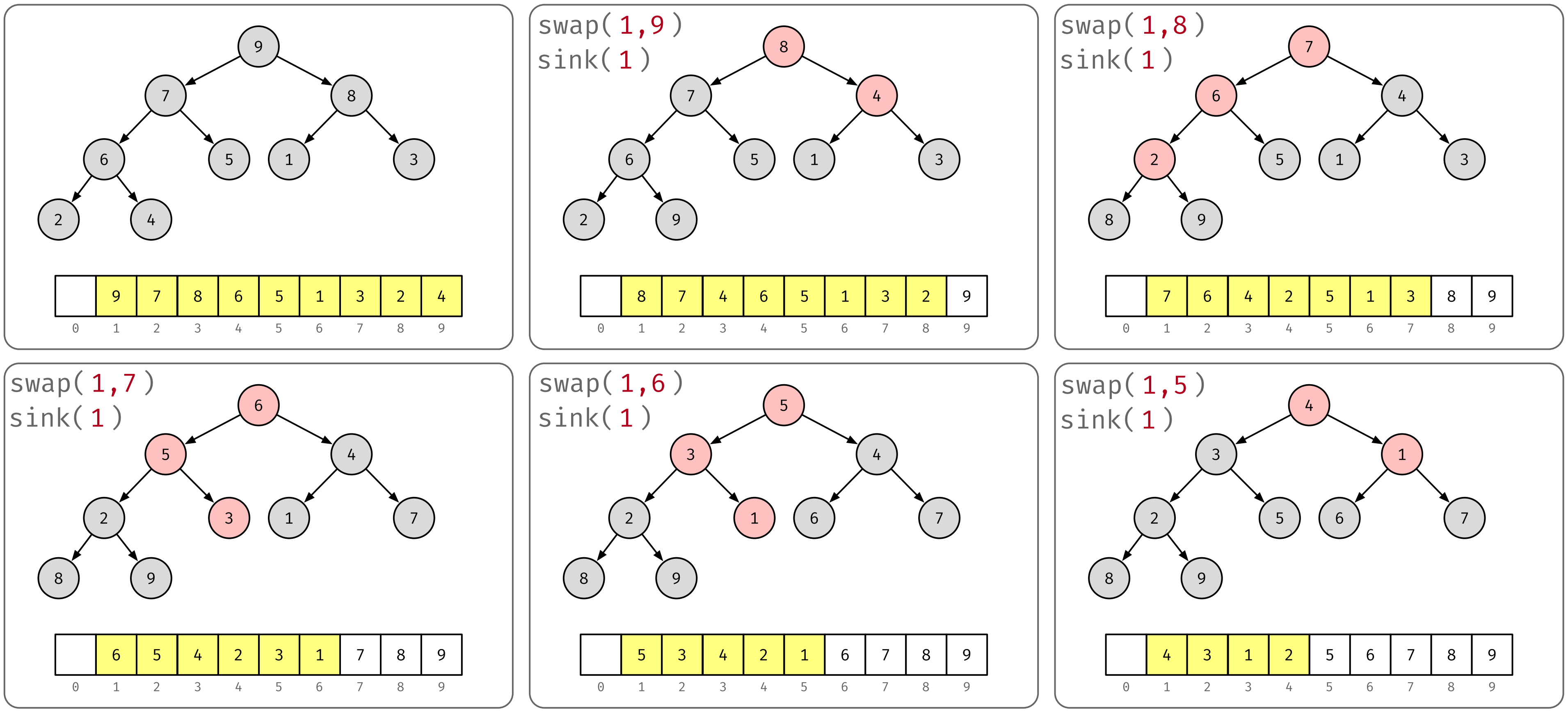
The idea

Build the heap bottom-up



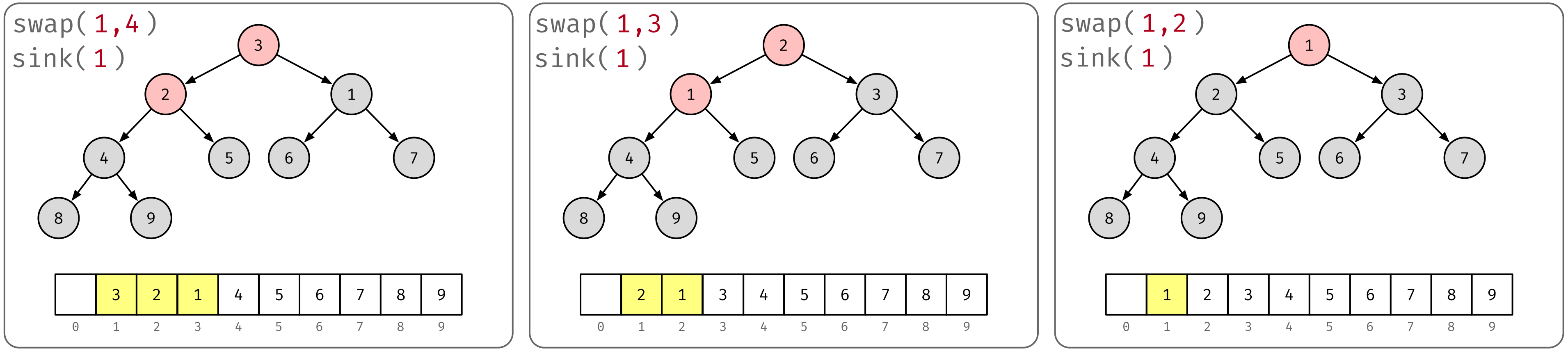
The idea

Sort down (remove max)



The idea

Sort down (remove max)



Implementation

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

Implementation

```python
@patch
def _sink(self:HeapSort, k:int) -> None:
  while 2 * k <= self.sz:
    j = 2 * k
    if j < self.sz and self.h[j] < self.h[j + 1]:
      j += 1

    if not self.h[k] < self.h[j]:
      break

    self.h[k], self.h[j] = \
                    self.h[j], self.h[k]
    k = j
```

Implementation

```
@patch
def sort(self:HeapSort) -> None:
  k = self.sz // 2
  while k >= 1:
    self._sink(k)
    k -= 1

  while self.sz > 1:
    self.h[1], self.h[self.sz] = \
            self.h[self.sz], self.h[1]
    self.sz -= 1
    self._sink(1)
```

Testing it

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