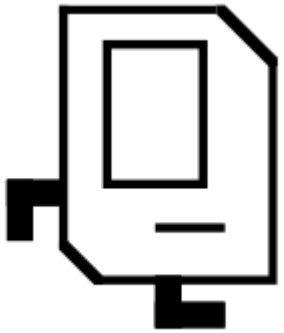
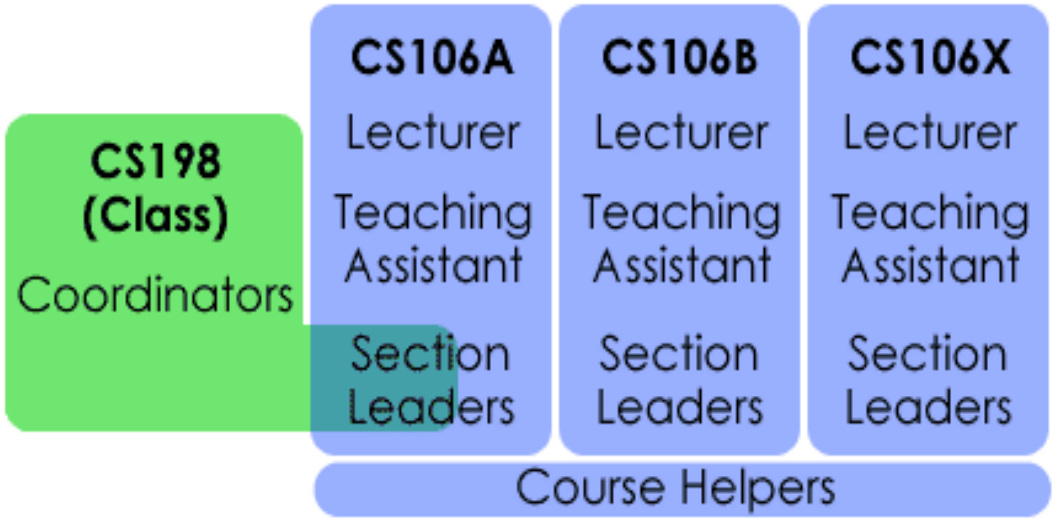
Before We Get Started

CS198 Section Leading



cs198@cs.stanford.edu

What is Section Leading?



Section leaders help with the teaching and grading of CS106A/B/X

What do Section Leaders do?

- Teach a weekly 50 minute section
- Help students in the LaIR
- Grade CS106 assignments and exams
- Hold IGs with students
- Have fun!

Prerequisites

For this round of applications, we are looking for applicants who are currently in CS106B/X... and that's you!

We are looking for section leaders from **any major** that can relate to students and clearly explain concepts.

Times and Requirements

You'll need to:

- Section lead for **two quarters!**
- Take CS198 for 3-4 units (1st quarter only)
- Attend staff meetings (Monday, 4:30-6:00PM)
- Attend workshops (Mon/Wed evenings)
- Fulfill all teaching, LaIR, and grading responsibilities

Why Section Lead?

- Teaching is an incredible experience!
- Work directly with students
- Participate in fun events
- Join an amazing group of people
- Leave your mark on campus

Participate in fun events

SLs keeping it classy for “LaIR Formal”

Welcome breakfast for new SLs

Participate in fun events

SLs at an “Iron Chef” event with Google

SLs on a SF Bay boat cruise with Microsoft

Join an amazing group of people

Mehran Sahami
Professor
Stanford

Marissa Mayer
CEO
Yahoo

Mike Schroepfer
CTO
Facebook

Leave your mark on campus

Section Leading is operated by current undergraduate and MS students.

Section Leading is a great way to make a substantive contribution to a meaningful program on campus.

Apply Now

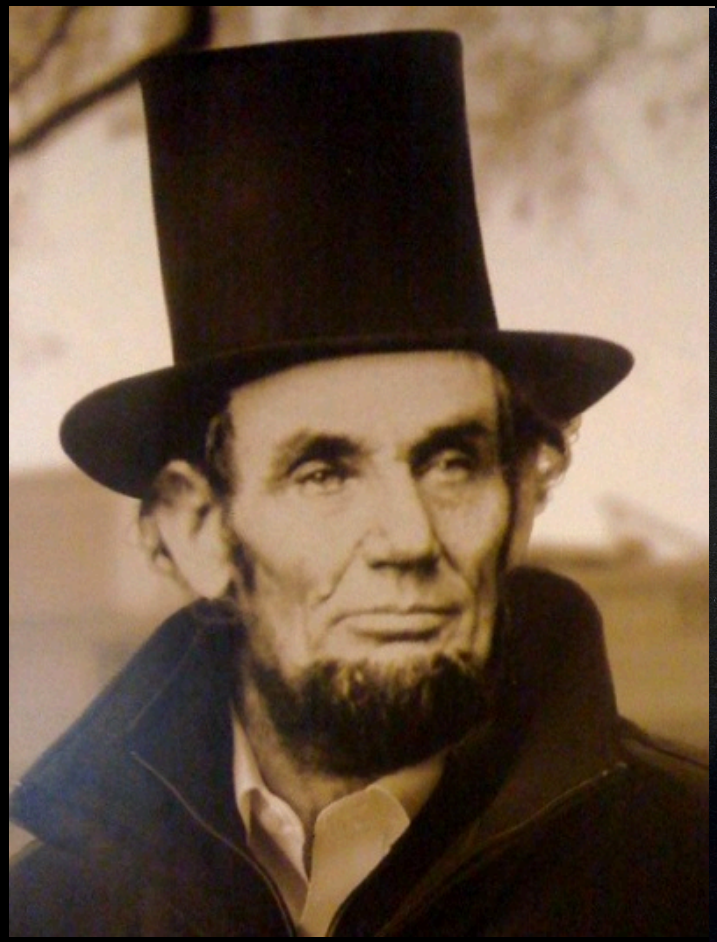
Applications are open now!
Round 2 deadline: Friday, February 19th

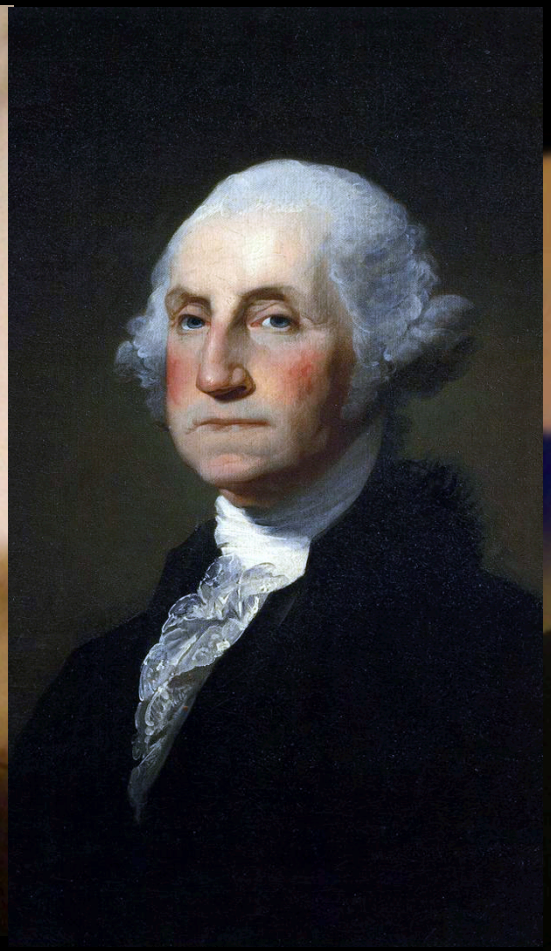
Online application: cs198.stanford.edu
Contact us: cs198@cs.stanford.edu

Thanks!

Chris Announcements

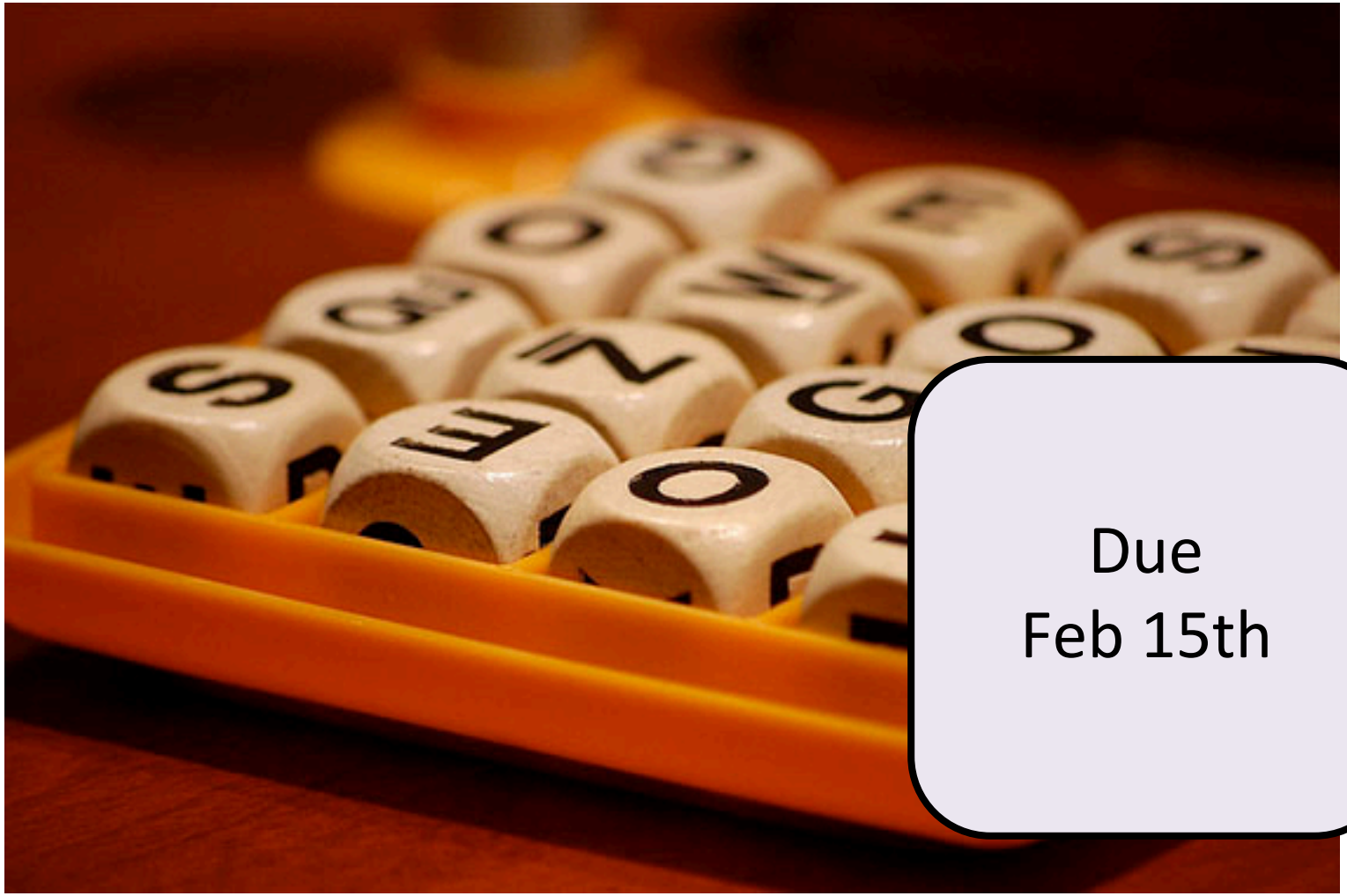
President's Day





Gravitational Waves Observed

Boggle



Due
Feb 15th

Priority Queue

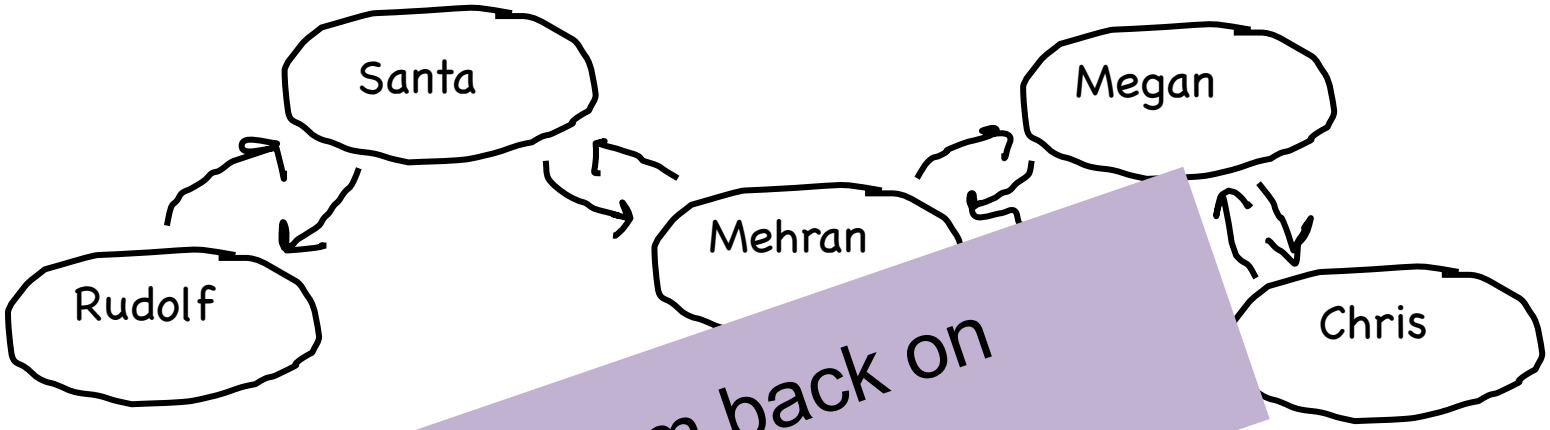
Priority Queue

Concepts:
Classes
Dynamic Alloc.
Linked Lists
Heaps

Start
Feb 15nd

Due
Feb 24th

Midterm



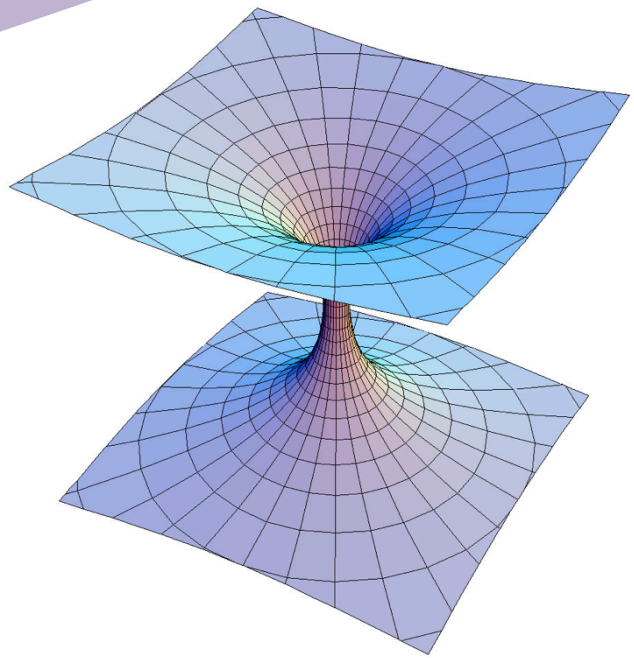
Get them back on Wednesday

PANDORA
internet radio

Tell us your favorite song and we'll create a station that explores that part of the music universe.

Enter Song

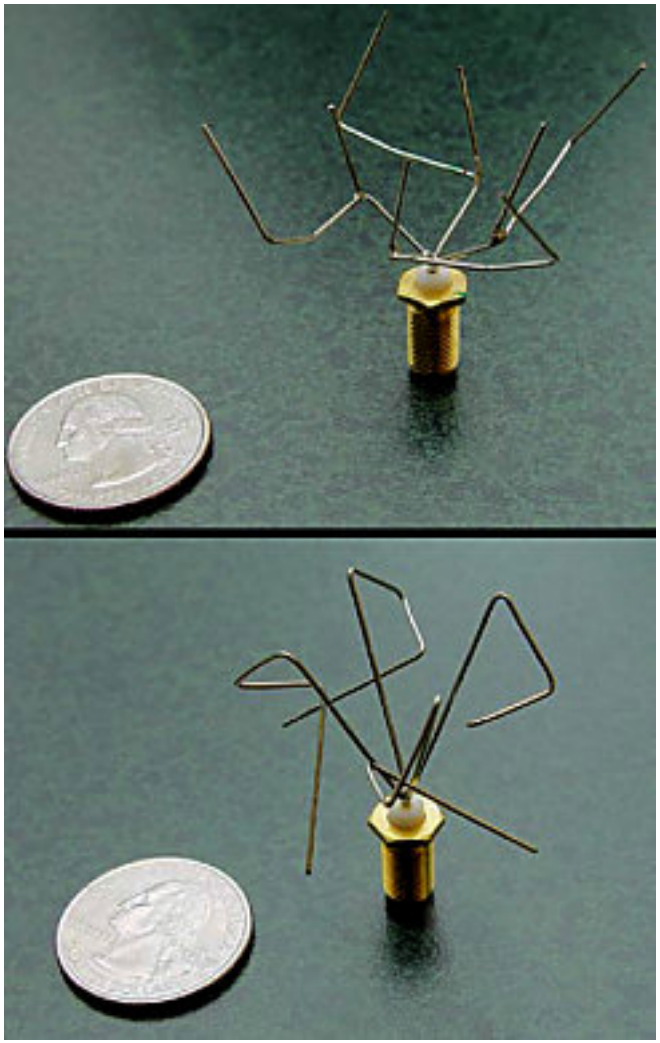
Create



Something cool.

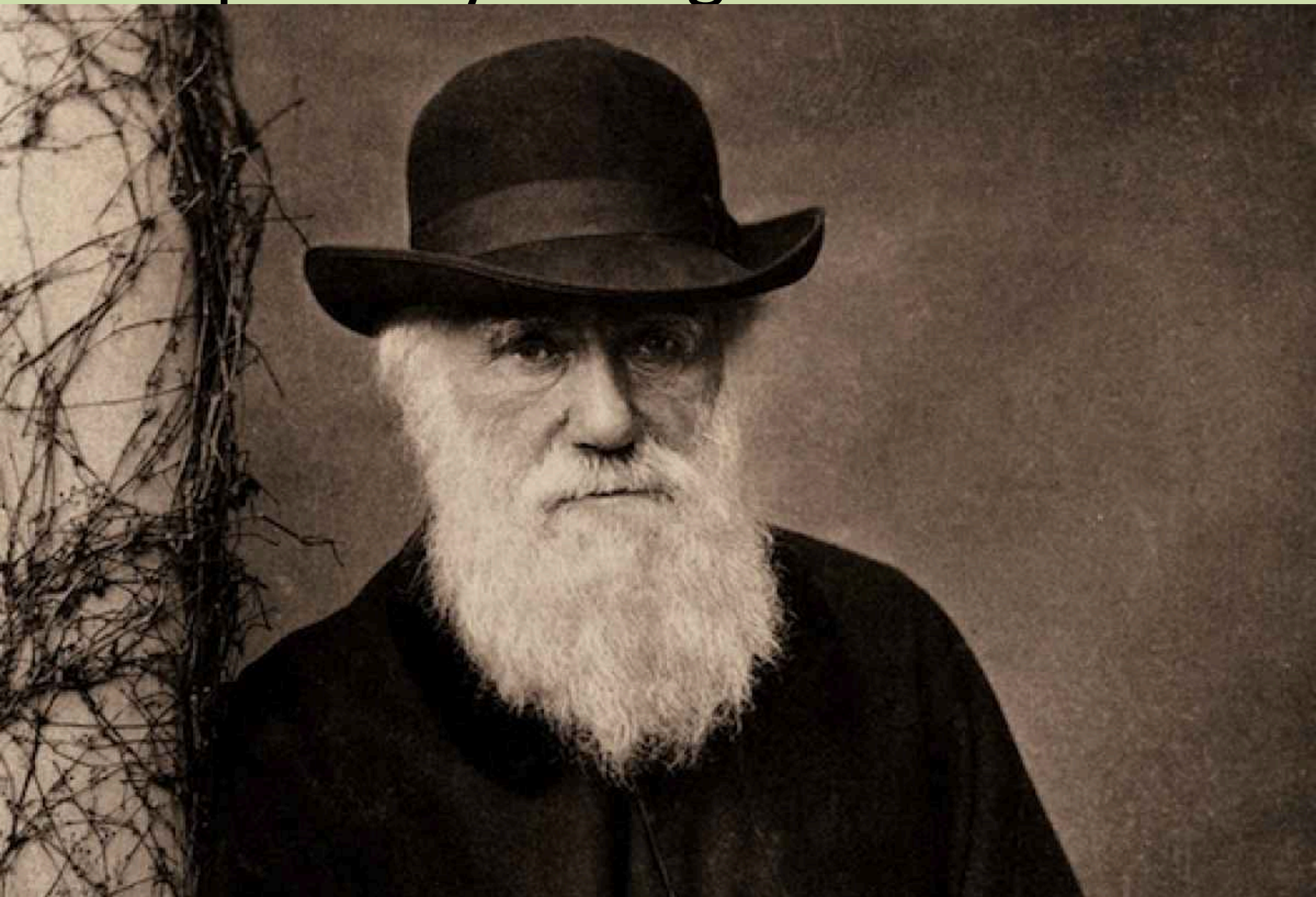
Computer Invented This

Antennae for microsatellites (~25kg)



2006

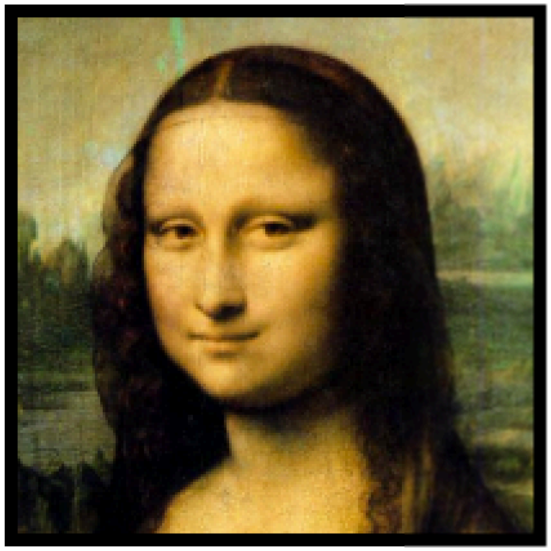
Inspired by Biological Evolution



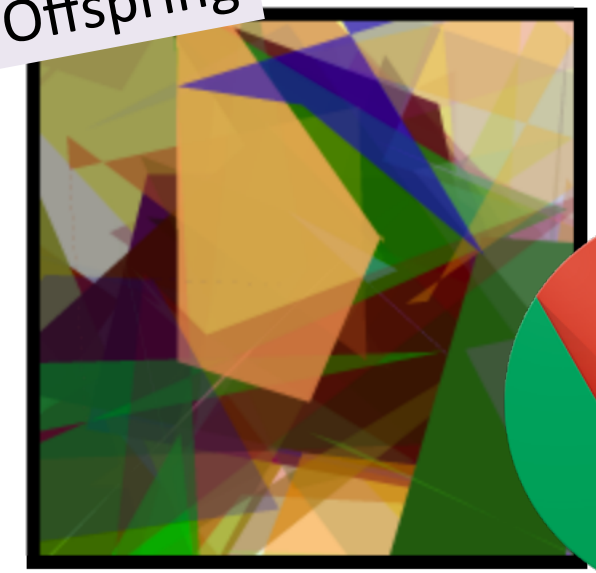
Evolving Mona Lisa

Best

Goal



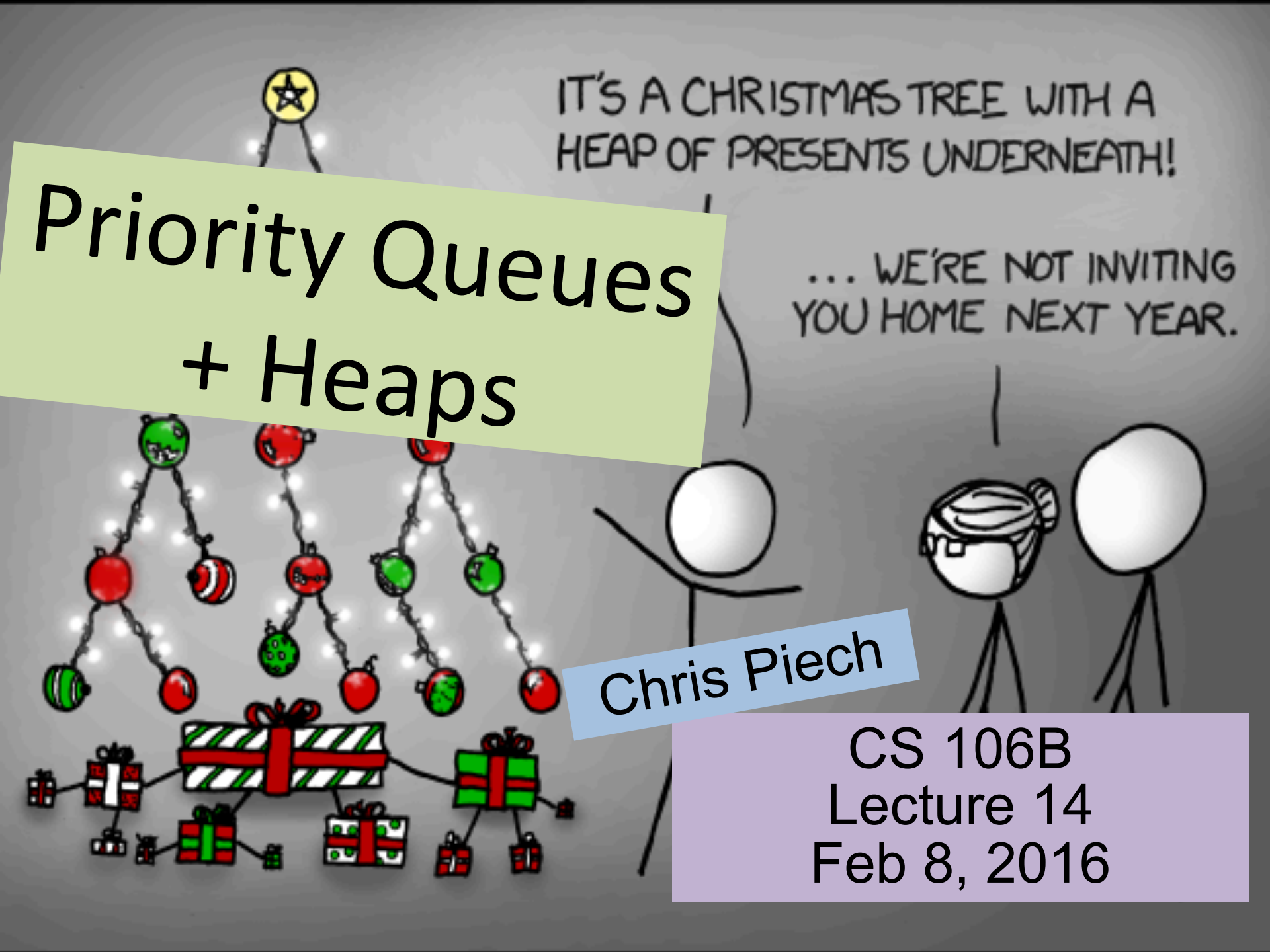
Last Offspring



How do Evolutionary Algorithms work?

Short answer: PQueues

Priority Queues + Heaps



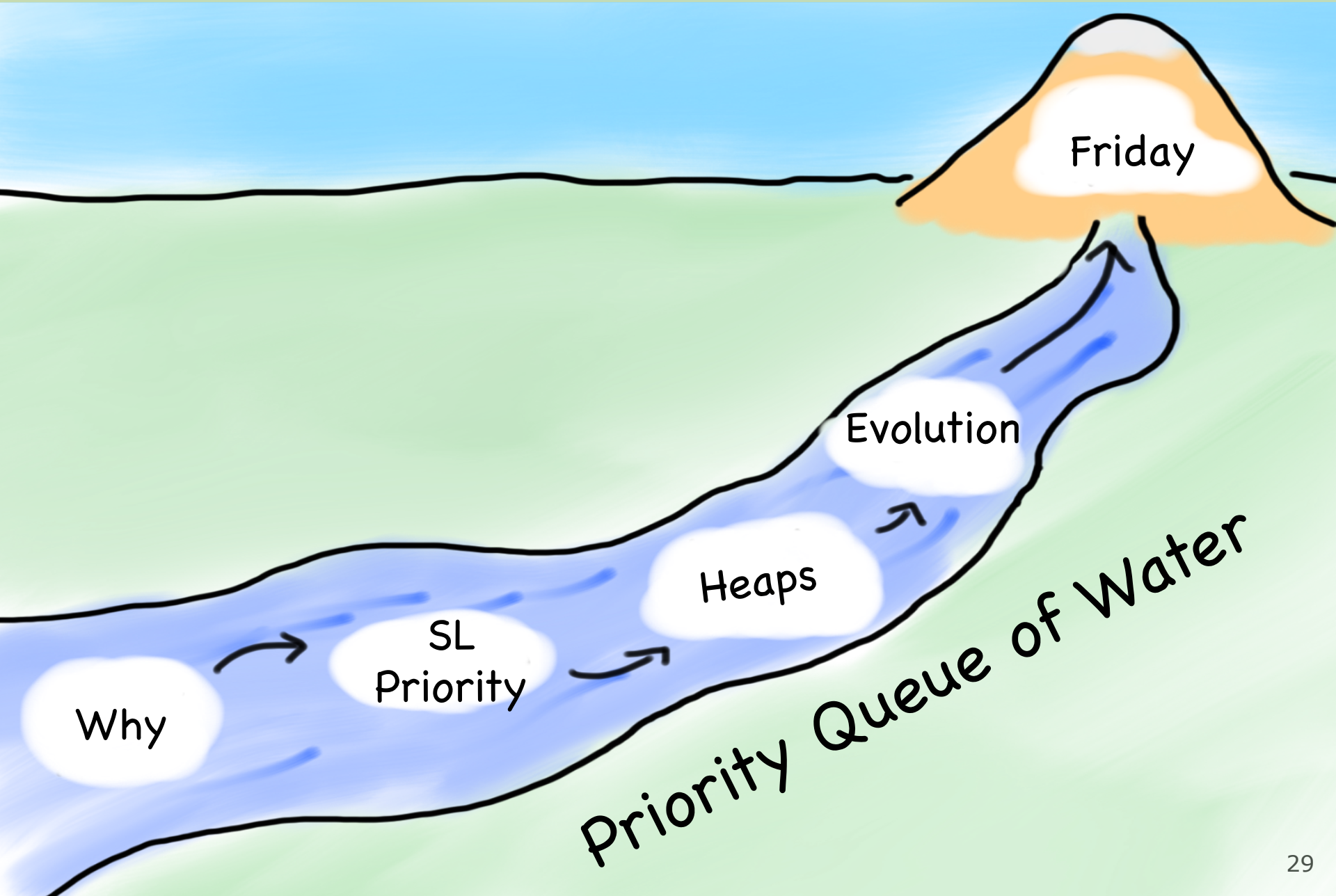
Chris Piech

CS 106B
Lecture 14
Feb 8, 2016

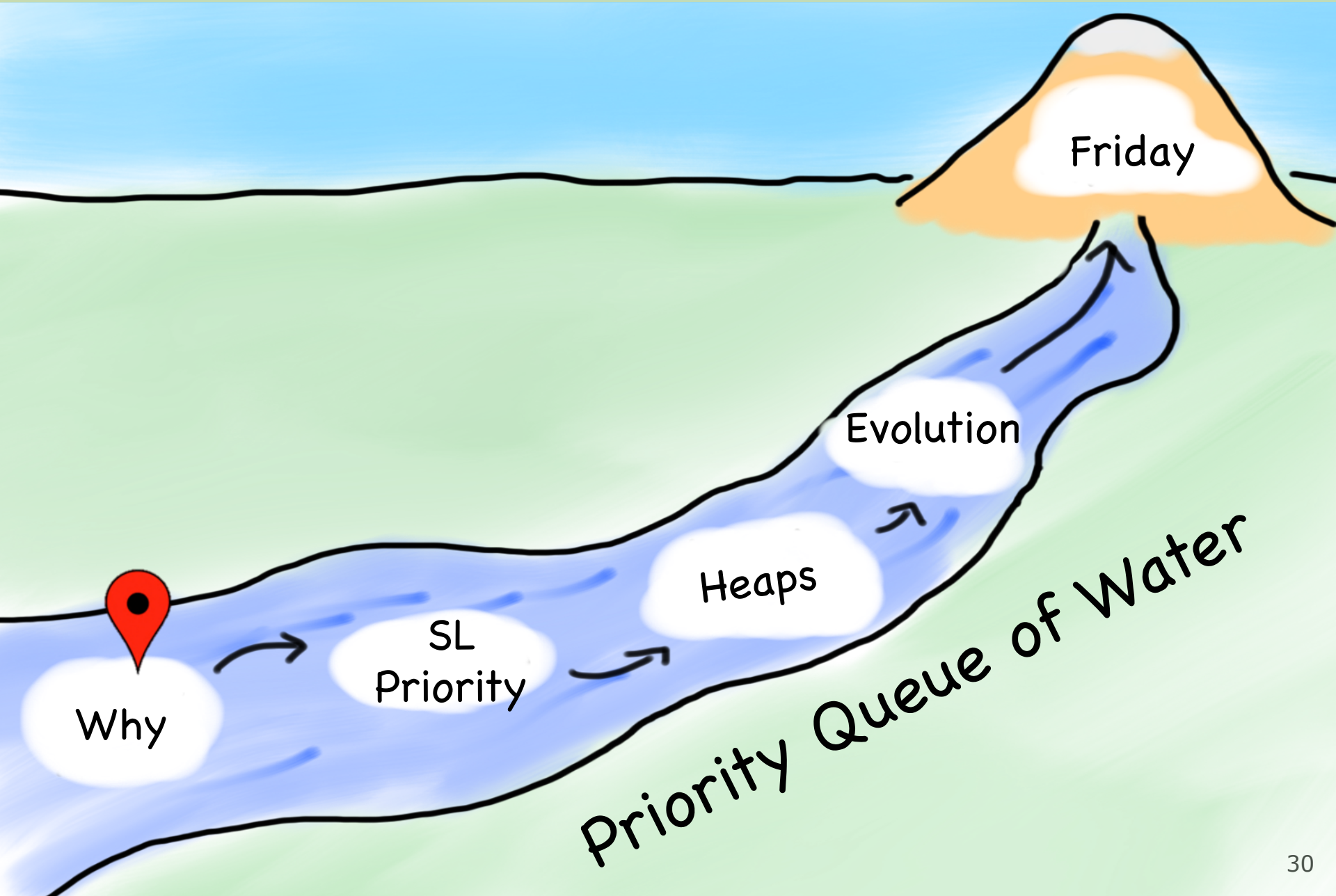
Today's Goals

1. Care About Priority Queues
2. Learn about Heaps

Today's Goals



Today's Goals



Imagine you worked in the ER

Would look something like this…

Prioritization problems

ER scheduling: Scheduling patients for treatment in the ER. A gunshot victim should be treated sooner than a guy with a cold, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?

key operations we want:

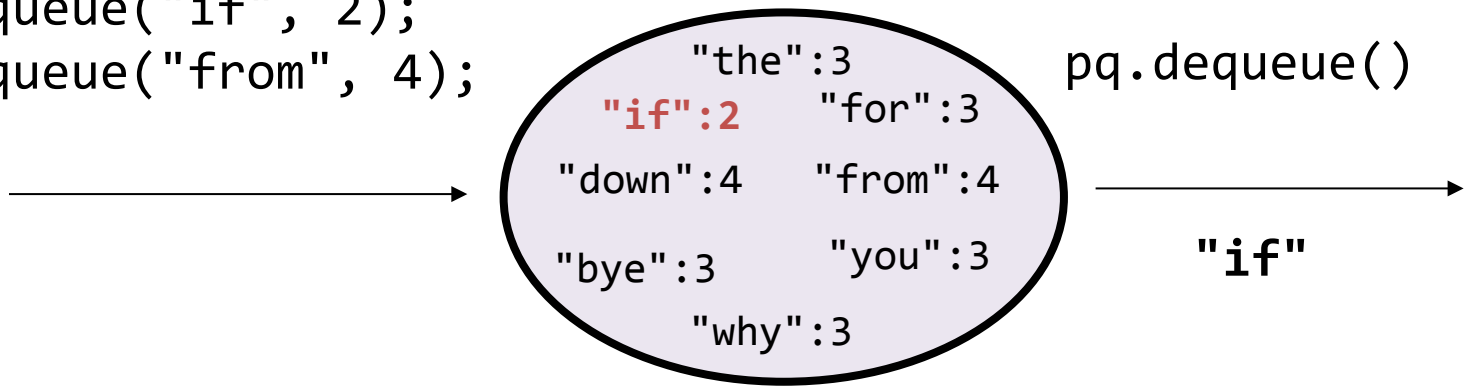
- ***add*** *an element (print job, patient, etc.)*
- ***Get/remove*** *the* ***most "important"*** *or "urgent" element*

Priority Queue ADT

priority queue: A collection of ordered elements that provides fast access to the highest-priority element.

- enqueue: adds an element with a given priority
- peek: returns highest-priority value
- dequeue: removes/returns highest-priority value

```
pq.enqueue("if", 2);
pq.enqueue("from", 4);
...
```



priority queue (using lengths as priorities, in this case)

PriorityQueue members

```
#include "pqueue.h"
```

`pq.enqueue(value, pri);`	adds value to queue, with the given priority number
`pq.dequeue()`	returns the value in the queue with highest (minimum) priority; throws an error if queue is empty
`pq.peek()`	returns highest (minimum) priority element without removing it; throws error if queue is empty
`pq.peekPriority()`	returns *priority* of highest (minimum) priority element; throws error if queue is empty
`pq.isEmpty()`	returns `true` if queue contains no elements (size 0)
`pq.size()`	returns the number of elements in the queue
`pq.toString()`	returns a string such as `"{3, 42, -7, 15}"`
`pq.changePriority(value, pri);`	alters an existing element's priority
`pq.clear();`	removes all elements of the set

```
PriorityQueue<string> faculty;
faculty.enqueue(”Chris", 5);    // low priority
faculty.enqueue("Eric", 2);     // high priority
...
```

Seen PQueues Before

Exercise: SL scheduling

- Every quarter the SLs choose their LaIR hours.
  - SLs with more seniority (quarters worked) get to choose first.
  - Each line of the input file contains the year the SL began working with us, and the quarter (1=fall, 2=winter, 3=spring, 4=summer).
  - Write code to read the list of SLs and output the order of choosing.

  - Input file format:

```
name year quarter
name year quarter
name year quarter
...
```

```
Zack 2014 2
Sara 2012 4
Tyler 2013 1
```

And… Here it is!

Exercise solution

```
PriorityQueue<string> SLs;   // read the contents of sls.txt
ifstream input;              // into a priority queue
input.open("sls.txt");
string slName;
int year;
int quarter;
while (input >> slName >> year >> quarter) {
    // store with year,quarter as priority so that the SLs
    // come out of the PQ in descending order of seniority
    // (e.g. year=2013, qtr=4 => priority = 1973)
    int priority = year - (10 * quarter);
    SLs.enqueue(slName, priority);
}

// pull the SLs out of the PQ from most to least seniority
while (!SLs.isEmpty()) {
    string sl = SLs.dequeue();
    cout << sl << " picks next." << endl;
}
```

Exercise solution

```
PriorityQueue<string> SLs;   // read the contents of sls.txt
ifstream input;              // into a priority queue
input.open("sls.txt");
string slName;
int year;
int quarter;
while (input >> slName >> year >> quarter) {
    // store with year,quarter as priority so that the SLs
    // come out of the PQ in descending order of seniori
    // (e.g. year=2013, qtr=4 => priority = 1973)
    int priority = year - (10 * quarter);
    SLs.enqueue(slName, priority);
}

// pull the SLs out of the PQ from most to lea    eniority
while (!SLs.isEmpty()) {
    string sl = SLs.dequeue();
    cout << sl << " picks next." << endl;
}
```

Remember that low priorities come out first

What a beautiful ADT!

How does it work?

Implementing PQ

There are several ways one might implement a priority queue.

PQ as Vector

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	x	b	a	m	q	t				
priority	5	4	8	5	5	2				
size	6		*capacity*		10					

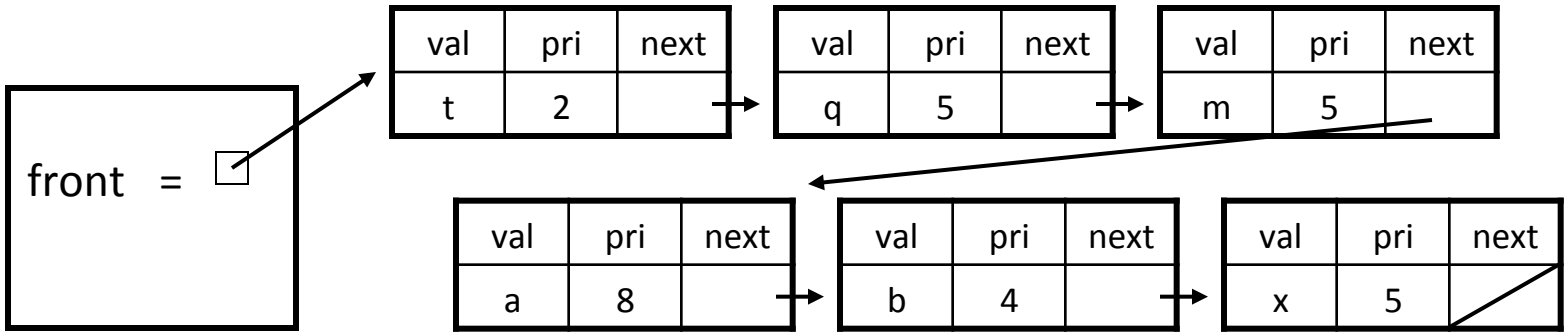
- enqueue by adding to *end*
- dequeue by searching for / removing highest-priority element
  - What is the big-Oh of enqueue? dequeue? peek?

PQ as Sorted Vector

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	t	b	m	q	x	a				
priority	2	4	5	5	5	8				
size	6		*capacity*		10					

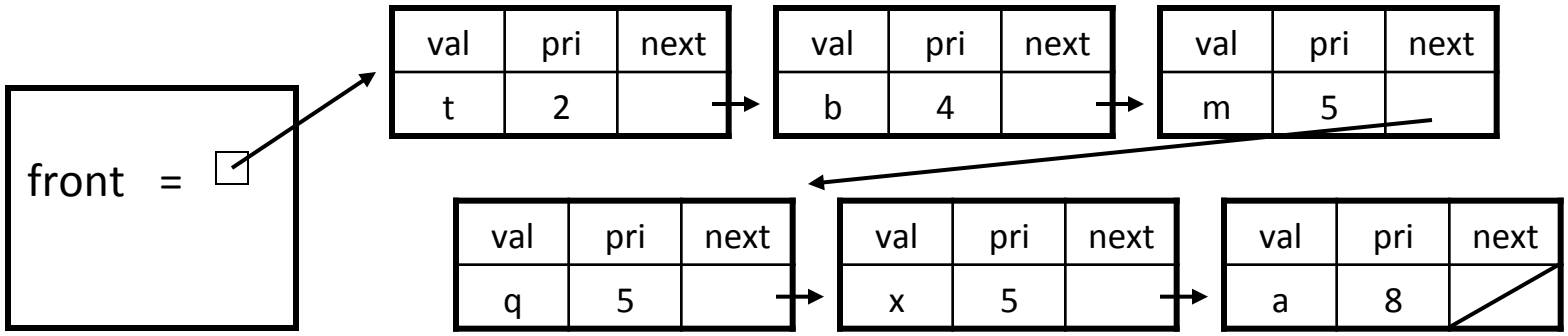
- enqueue by adding to proper place to preserve sorted order
- dequeue the *first* element
  - What is the big-Oh of enqueue? dequeue? Peek?

PQ as Linked List



- enqueue by adding to *front*
- dequeue by searching for / removing highest-priority element
  - What is the big-Oh of enqueue? dequeue? peek?

PQ as Sorted Linked List



- enqueue by adding to proper place to preserve sorted order
- dequeue the *first* element
  - What is the big-Oh of enqueue? dequeue? peek?

Sorted = fast dequeue / peek

Unsorted = fast enqueue

Best of both worlds?

Max Heaps

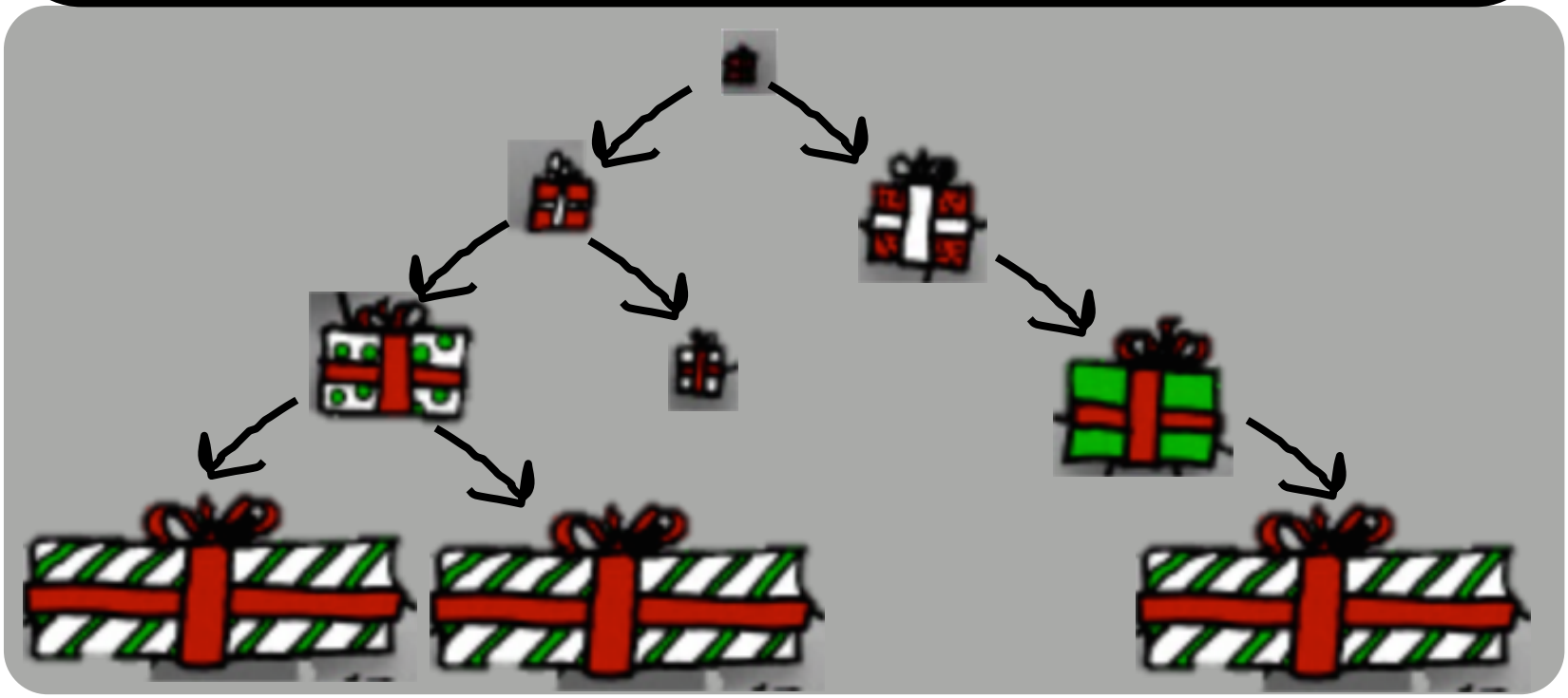
Tree

Every branch takes you to a “smaller” node

Min Heaps

Tree

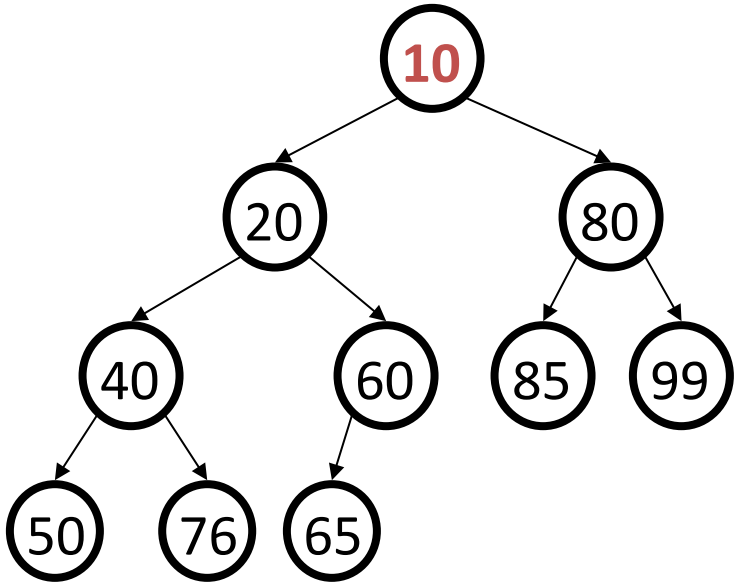
Every branch takes you to a “larger” node



Priority Heaps

Tree

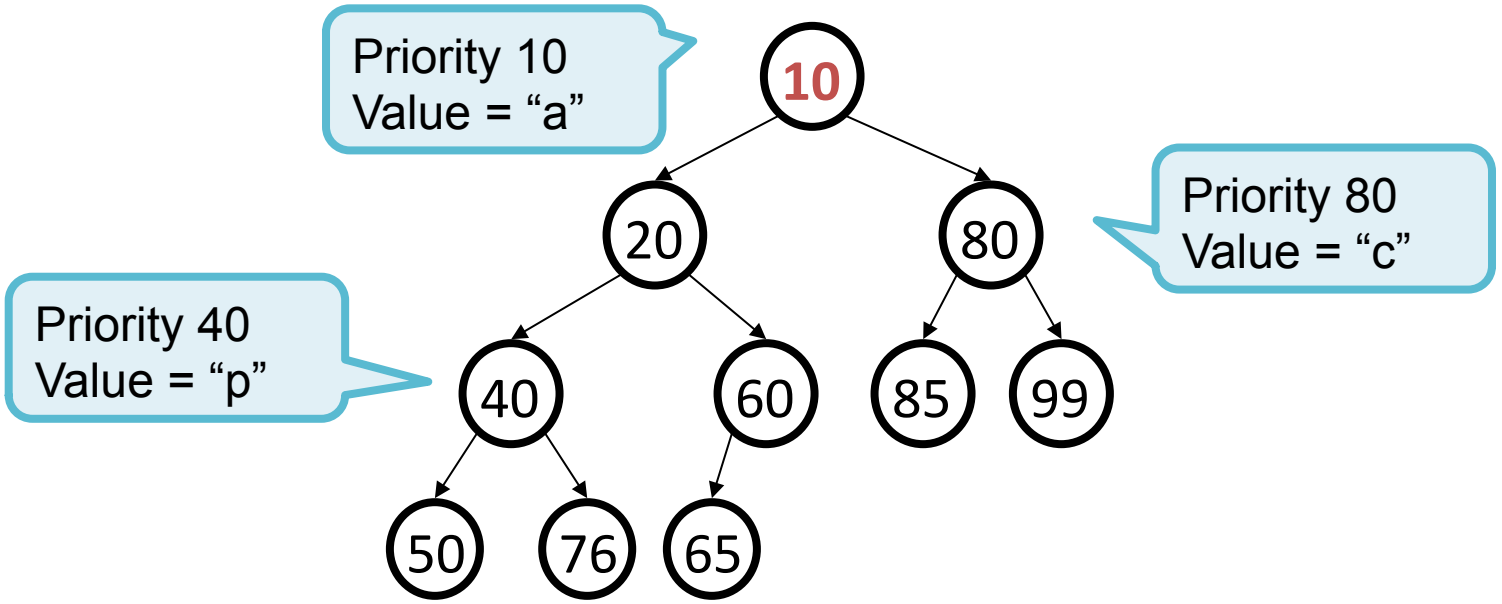
Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Priority Heaps

Tree

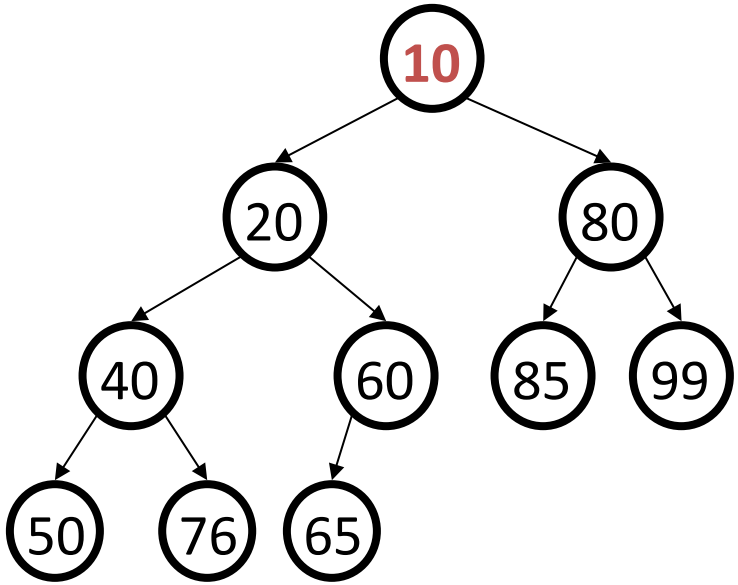
Heap Ordered: Every branch takes you to a "greater" node.
Binary: Every node has at most two children.
Complete: There are no "gaps" in the tree.



Priority Heaps

Tree

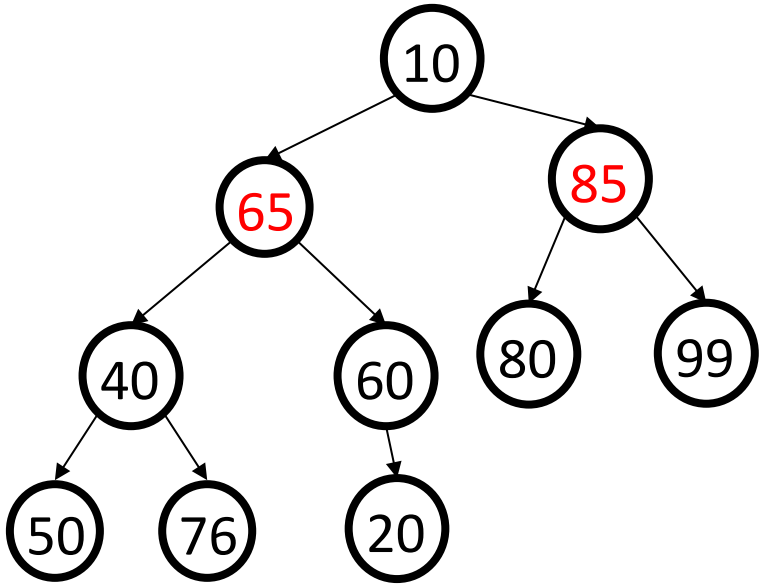
Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Not Priority Heaps

Tree

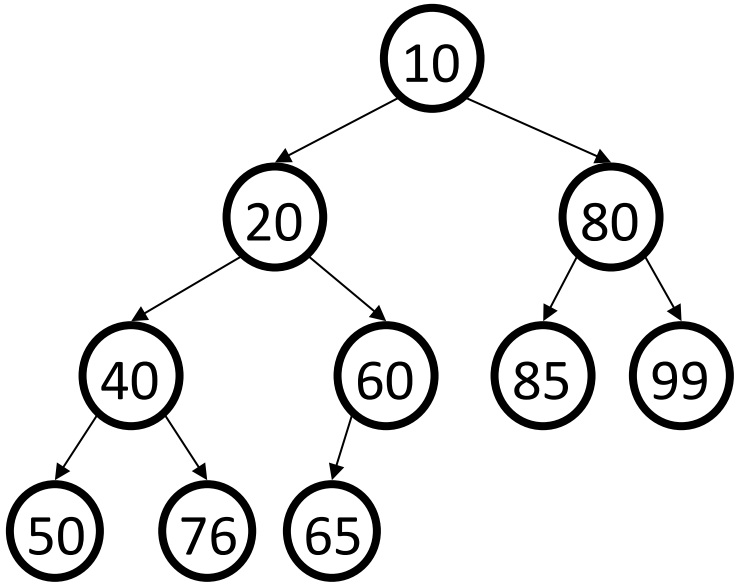
Not Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Priority Heaps

Tree

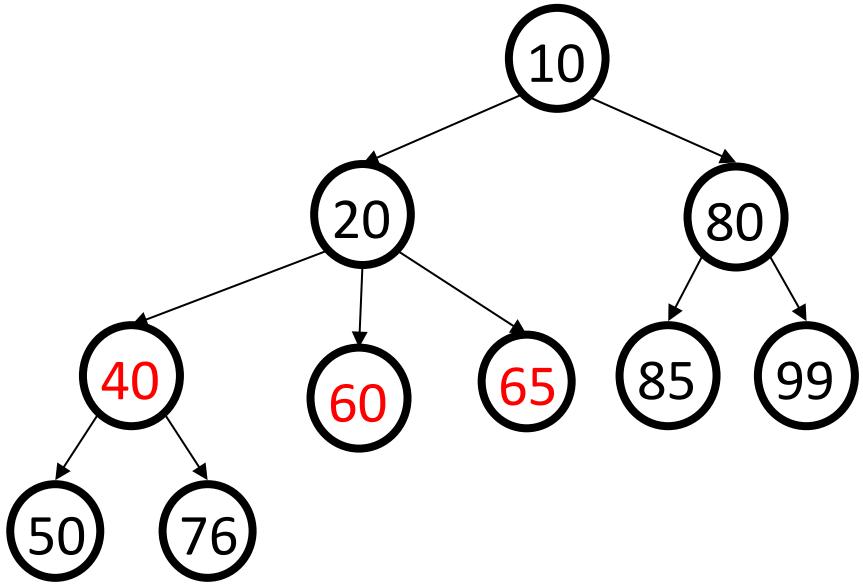
Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Not Priority Heaps

Tree

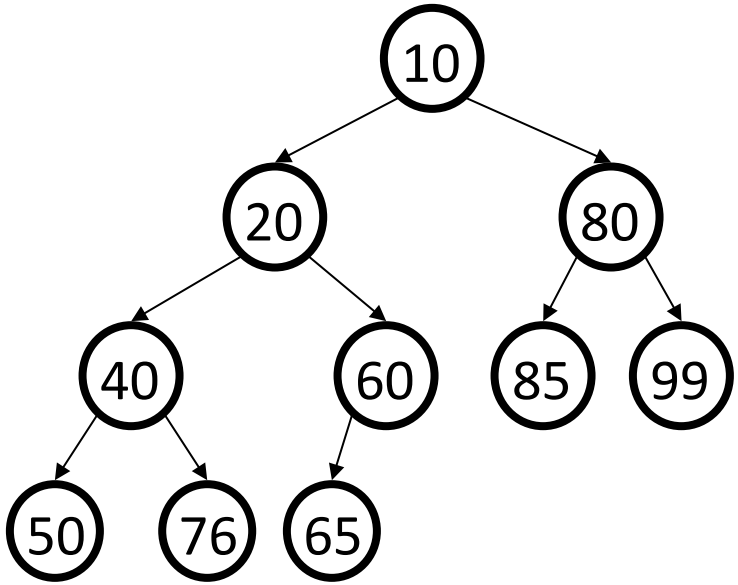
Heap Ordered: Every branch takes you to a “greater” node.
Not Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Priority Heaps

Tree

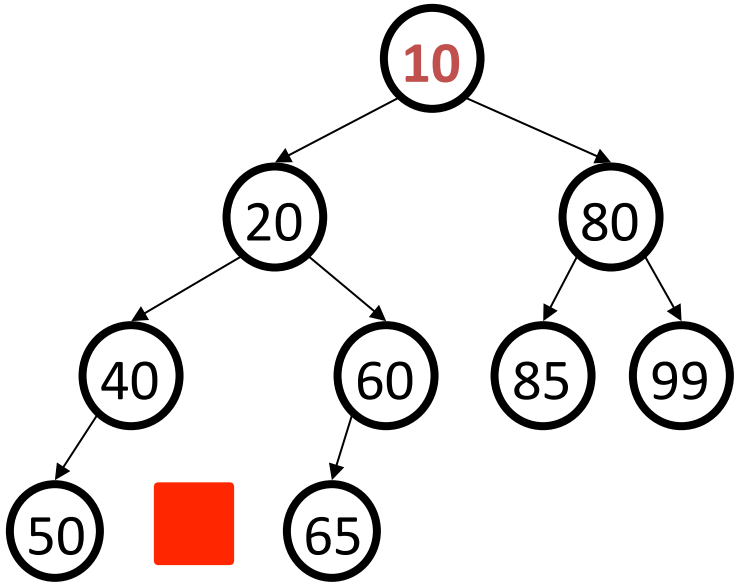
Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Priority Heaps

Tree

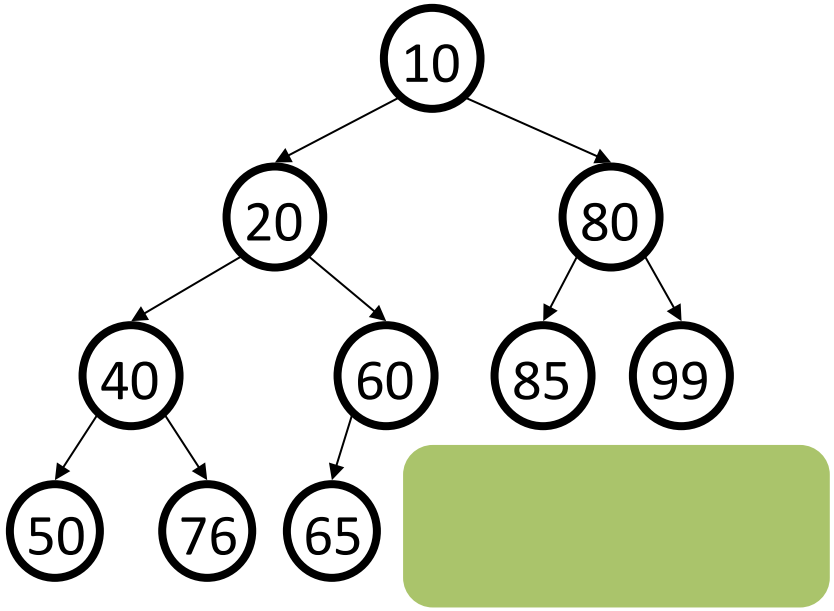
Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Not Complete: There are no “gaps” in the tree.



Priority Heaps

Tree

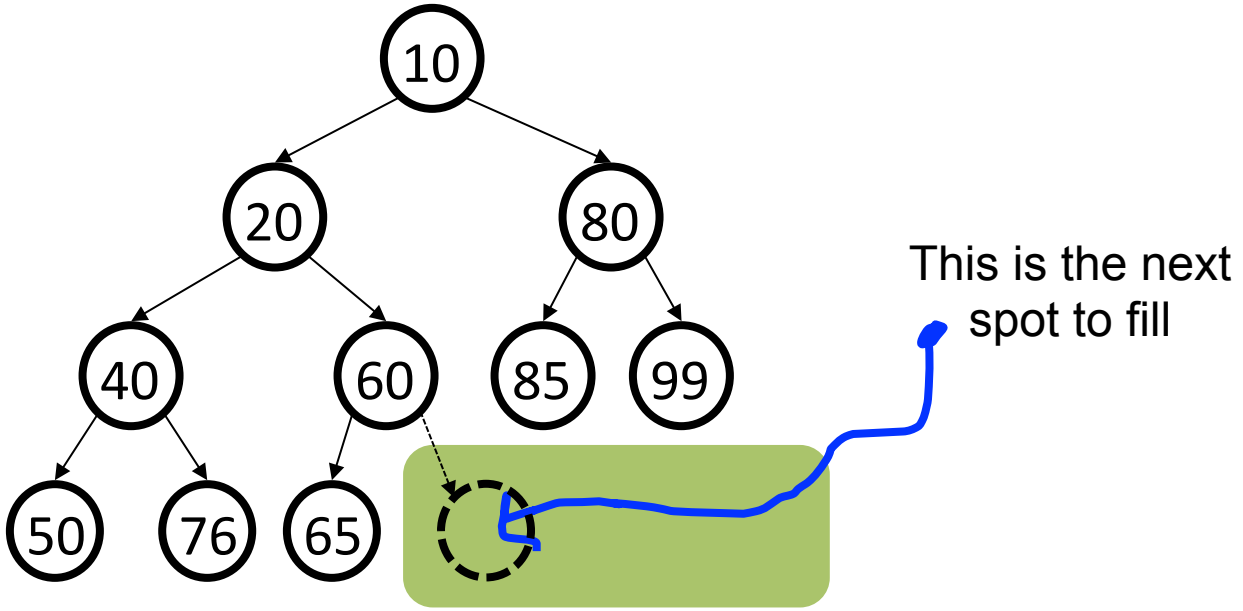
Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



Priority Heaps

Tree

Heap Ordered: Every branch takes you to a “greater” node.
Binary: Every node has at most two children.
Complete: There are no “gaps” in the tree.



How do we represent that?

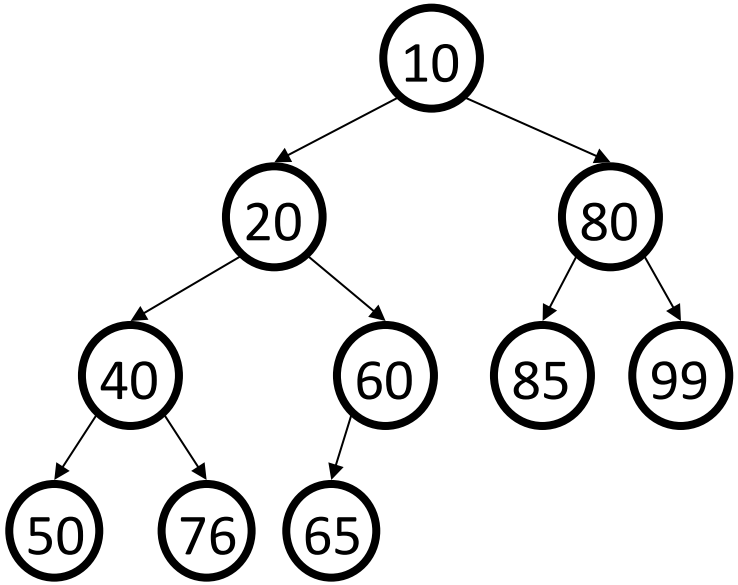
Heaps

Since it's binary and complete, it turns out we can use an array!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a			
priority		2	5	4	5	5	8			
size	6		*capacity*		10					

- The "start" or "root" index is 1.
- Every index *i* has a "parent" index: i/2
- Every index has two "child" indexes: i*2, i*2 + 1

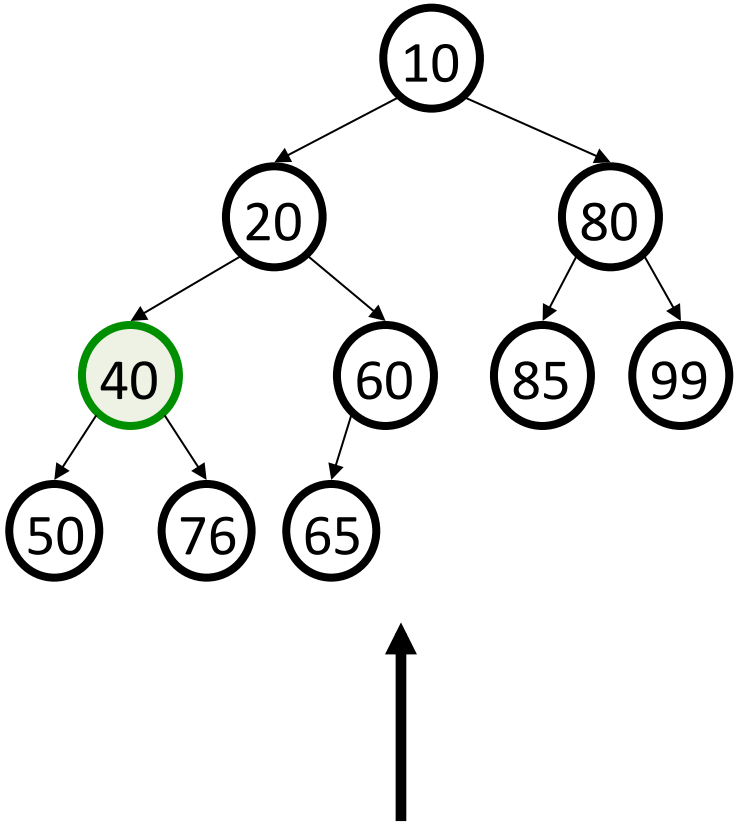
Heap as Tree



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

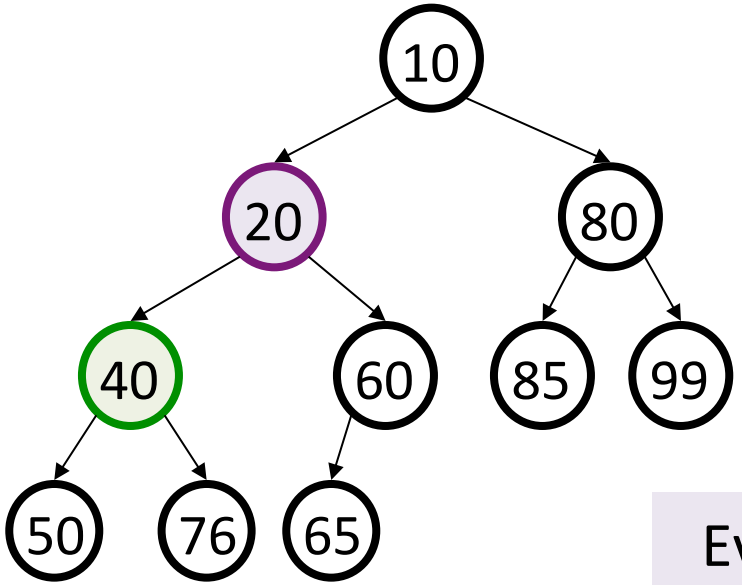
Heap as Tree



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Heap as Tree

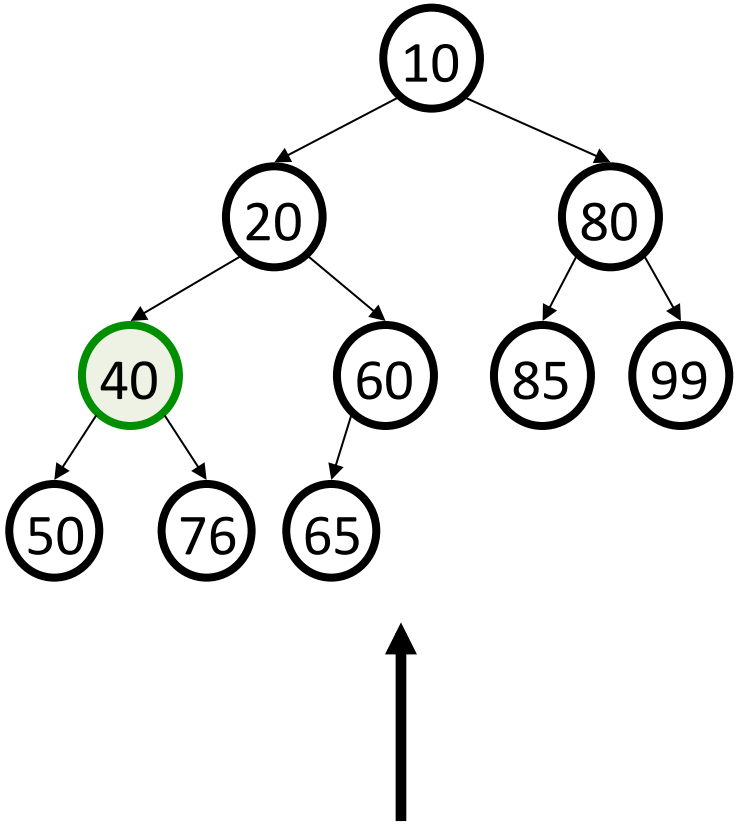


Every index *i* has a "parent" index: *i*/2

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

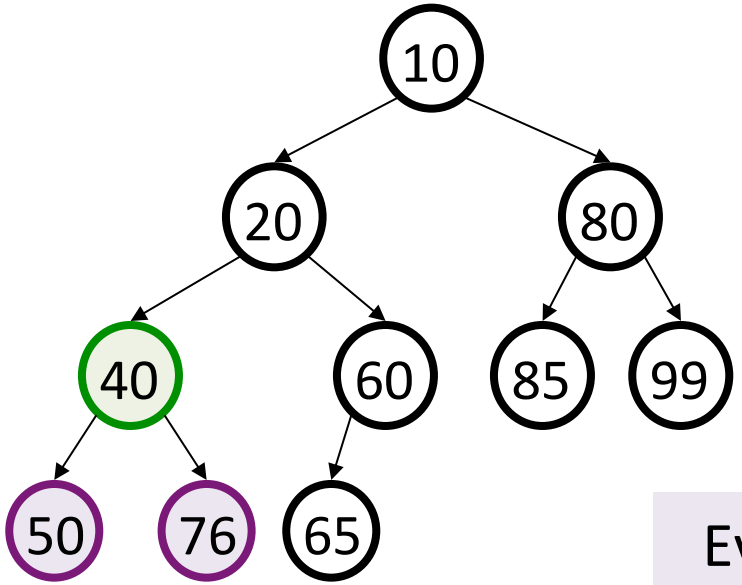
Heap as Tree



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Heap as Tree



Every index *i* has two children at index 2 **i* and 2 * *i* + 1

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Heap add (enqueue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a			
priority		2	5	4	5	5	8			

size 6 *capacity* 10

```
pq.enqueue("k", 1);
```

Heap add (enqueue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a	**k**		
priority		2	5	4	5	5	8	**1**		
size	**7**	*capacity*		10						

```
pq.enqueue("k", 1);
```

1. Place it in the first empty index

Heap add (enqueue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a	**k**		
priority		2	5	4	5	5	8	**1**		
size	**7**	*capacity*		10						

```
pq.enqueue("k", 1);
```

1. Place it in the first empty index
2. Bubble up to conserve "heap" property

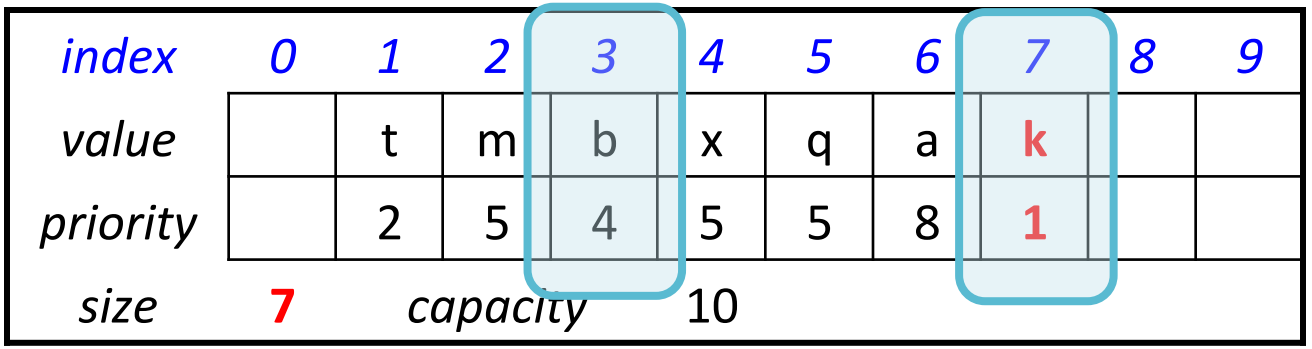
Bubble Up!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	b	x	q	a	**k**		
priority		2	5	4	5	5	8	**1**		
size	**7**	*capacity*			10					

```
pq.enqueue("k", 1);
```

1. Check priority against its parent.
2. If priority is smaller: swap the two.
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

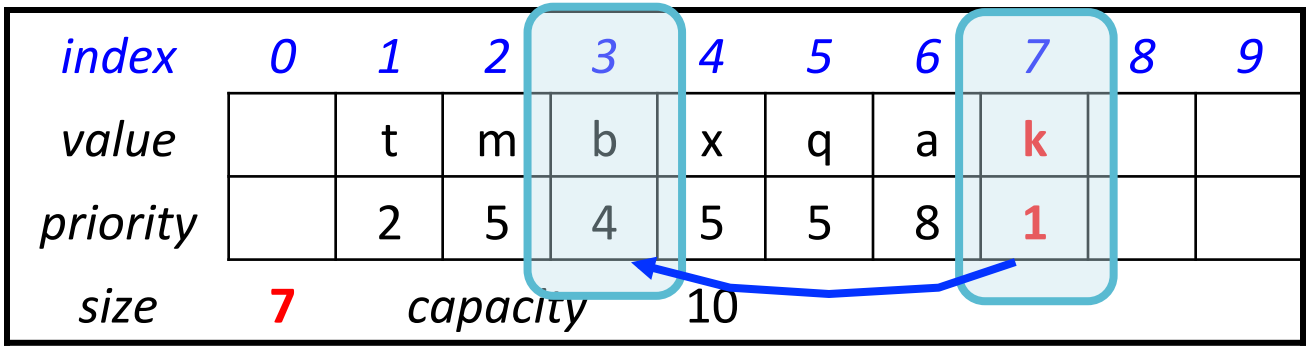
Bubble Up!




```
pq.enqueue("k", 1);
```

1. **Check priority against its parent.**
2. If priority is smaller: swap the two.
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

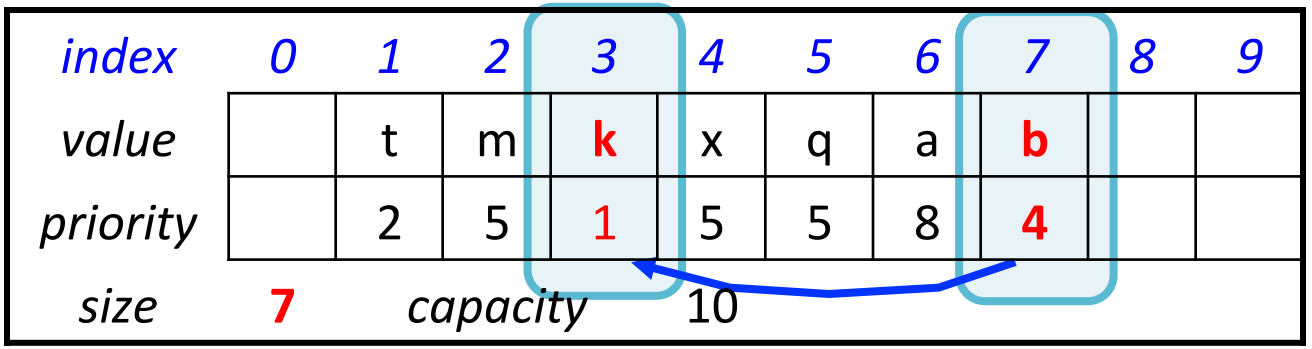
Bubble Up!




```
pq.enqueue("k", 1);
```

1. Check priority against its parent.
2. **If priority is smaller: swap the two.**
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

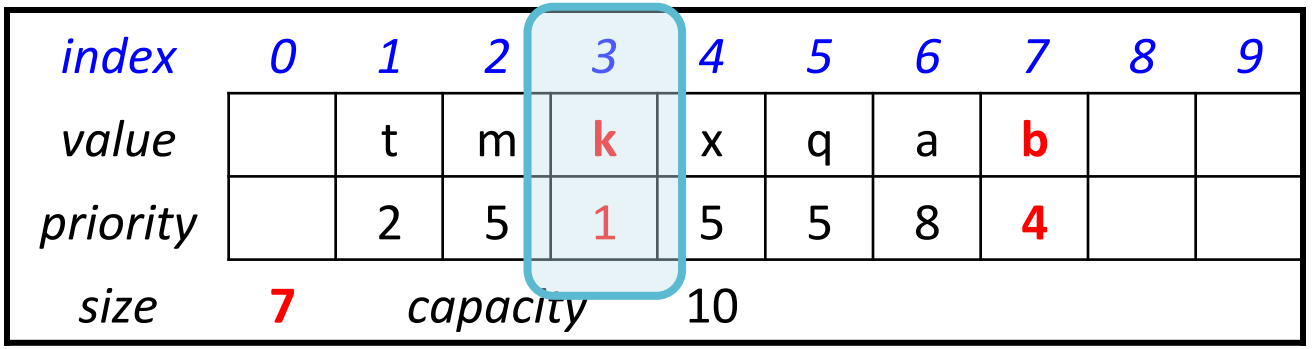
Bubble Up!




```
pq.enqueue("k", 1);
```

1. Check priority against its parent.
2. **If priority is smaller: swap the two.**
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

Bubble Up!



1. Check priority against its parent.
2. If priority is smaller: swap the two.
3. **Repeat until the priority of the parent is smaller or you reach the top of the tree**

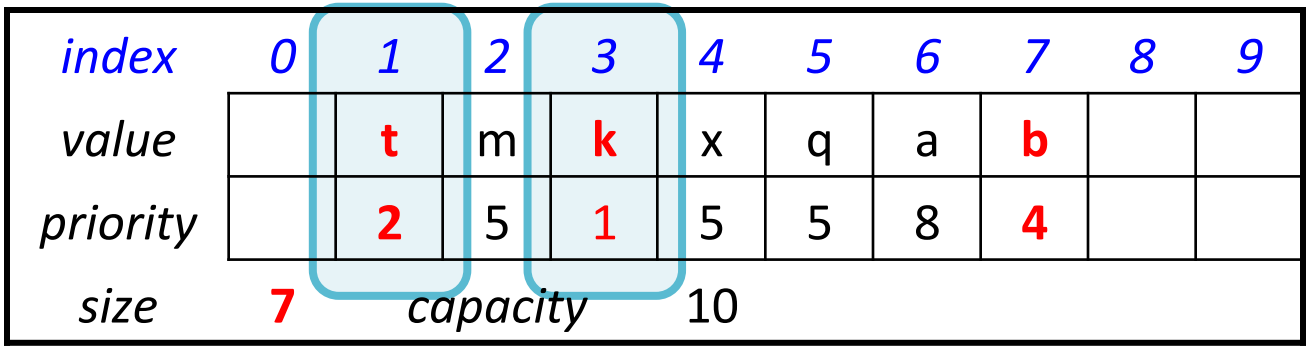
Bubble Up!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		t	m	**k**	x	q	a	**b**		
priority		2	5	1	5	5	8	**4**		

size **7** *capacity* 10

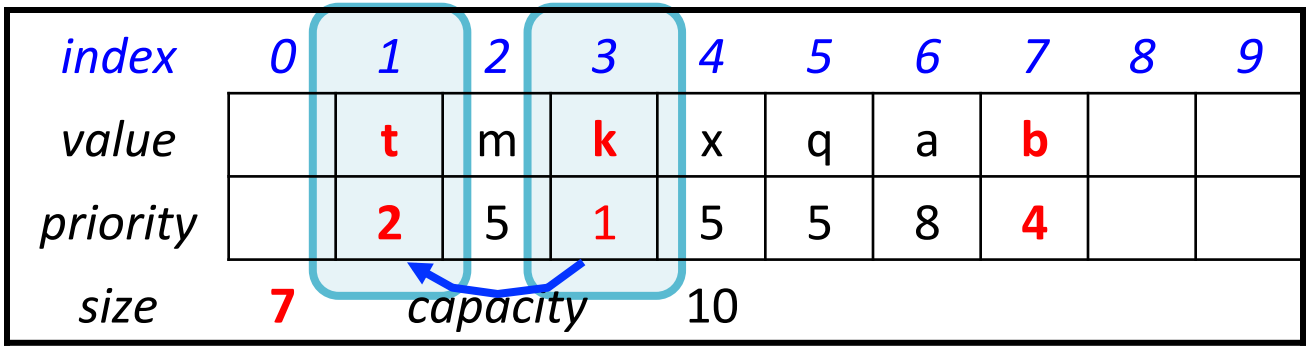
1. **Check priority against its parent.**
2. If priority is smaller: swap the two.
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

Bubble Up!



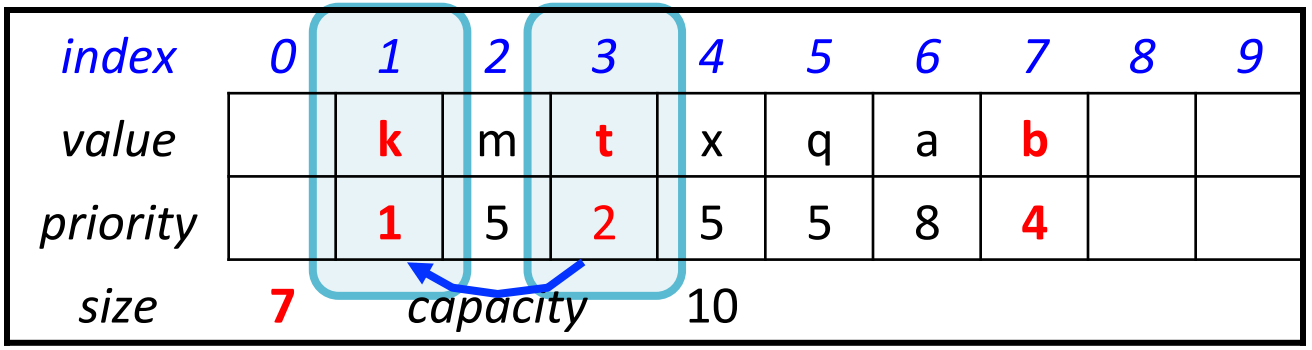
1. **Check priority against its parent.**
2. If priority is smaller: swap the two.
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

Bubble Up!



1. Check priority against its parent.
2. **If priority is smaller: swap the two.**
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

Bubble Up!



1. Check priority against its parent.
2. **If priority is smaller: swap the two.**
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

Bubble Up!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		**k**	m	**t**	x	q	a	**b**		
priority		**1**	5	**2**	5	5	8	**4**		
size	**7**	*capacity*			10					

1. Check priority against its parent.
2. **If priority is smaller: swap the two.**
3. Repeat until the priority of the parent is smaller or you reach the top of the tree

Bubble Up!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		**k**	m	**t**	x	q	a	**b**		
priority		**1**	5	**2**	5	5	8	**4**		

size **7** *capacity* 10

1. Check priority against its parent.
2. If priority is smaller: swap the two.
3. **Repeat until the priority of the parent is smaller or you reach the top of the tree**

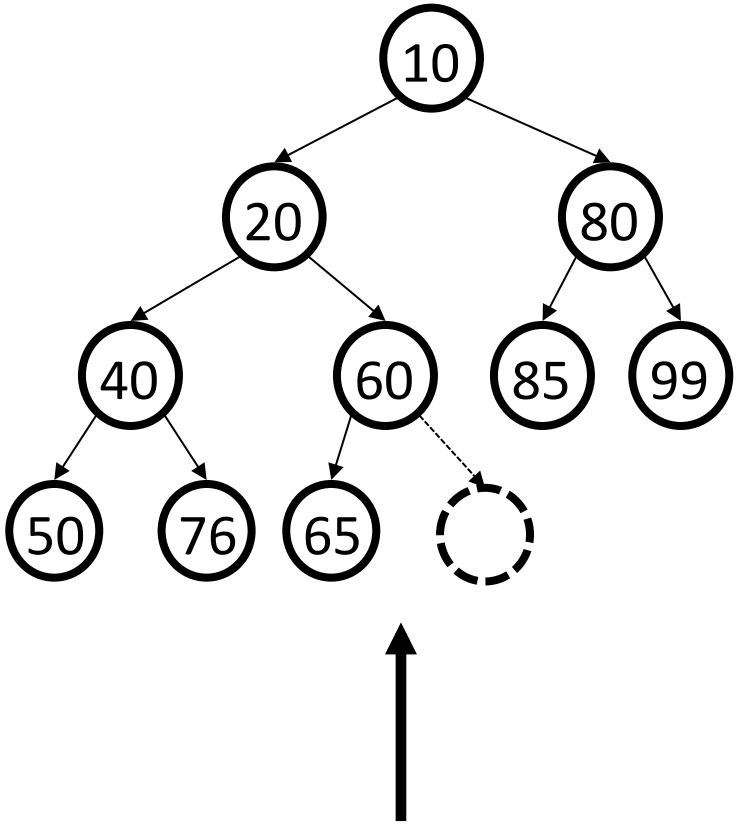
Bubble Up!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		**k**	m	**t**	x	q	a	**b**		
priority		**1**	5	2	5	5	8	**4**		
size	**7**	*capacity*		10						

1. Check priority against its parent.
2. If priority is smaller: swap the two.
3. **Repeat until the priority of the parent is smaller or you reach the top of the tree**

Big O?

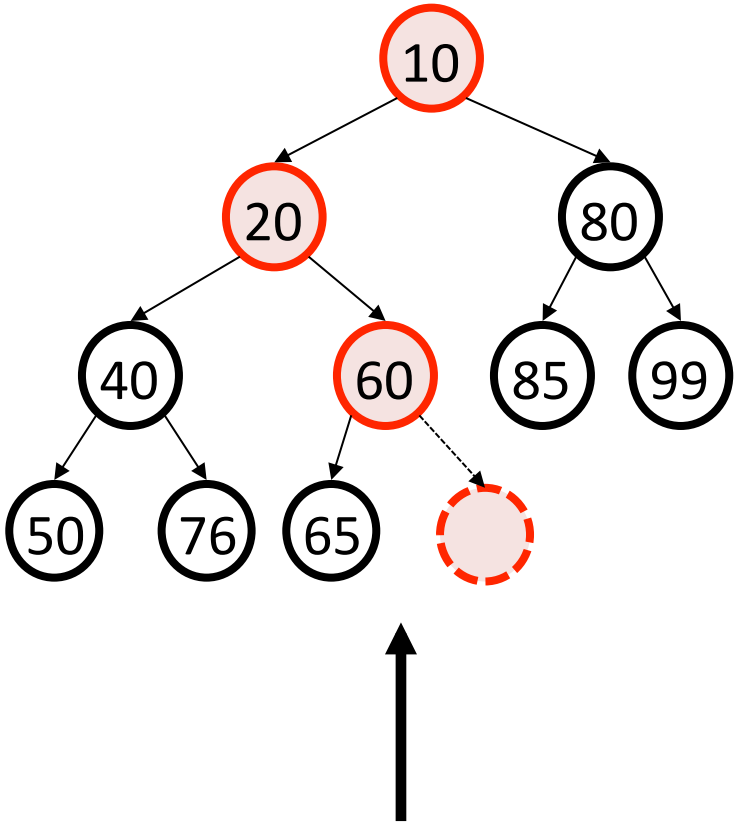
Worst Case



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

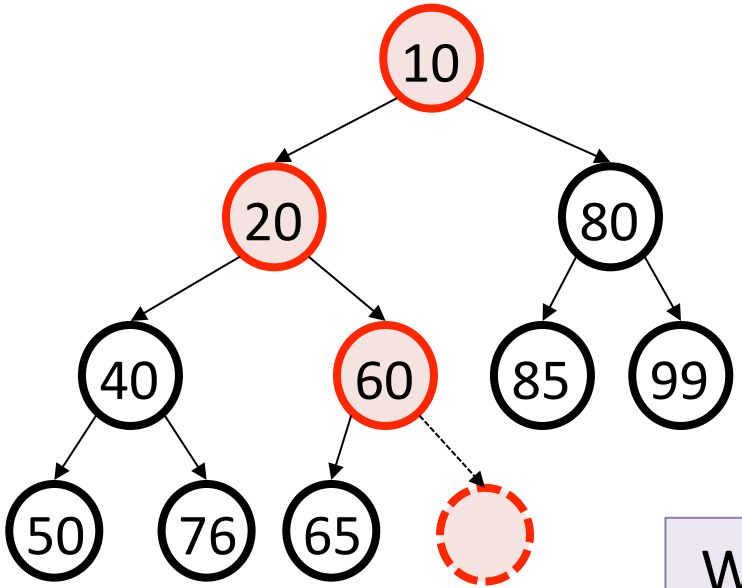
Worst Case



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Worst Case

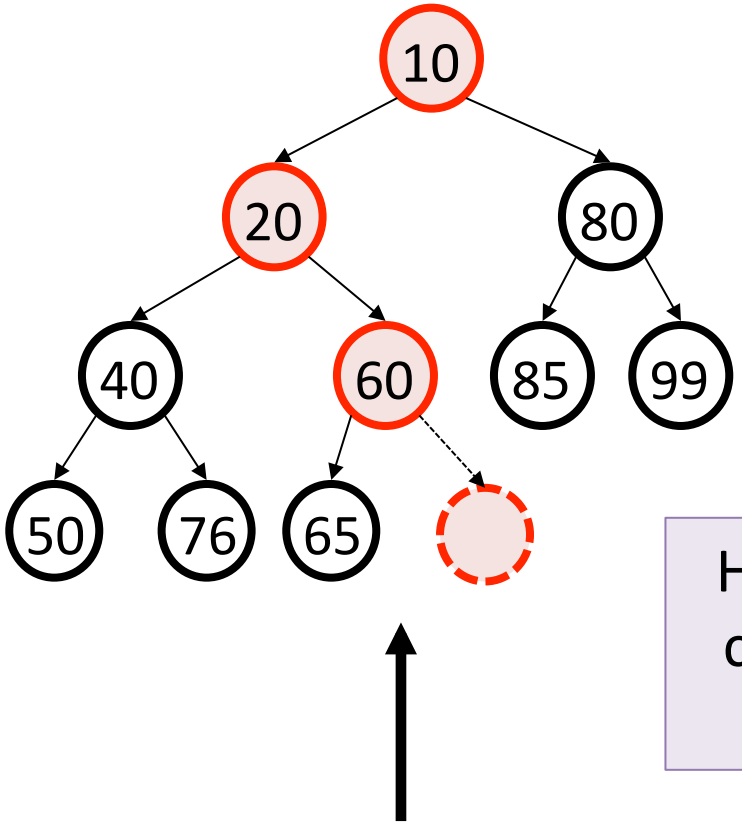


What is the depth of the tree?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Worst Case

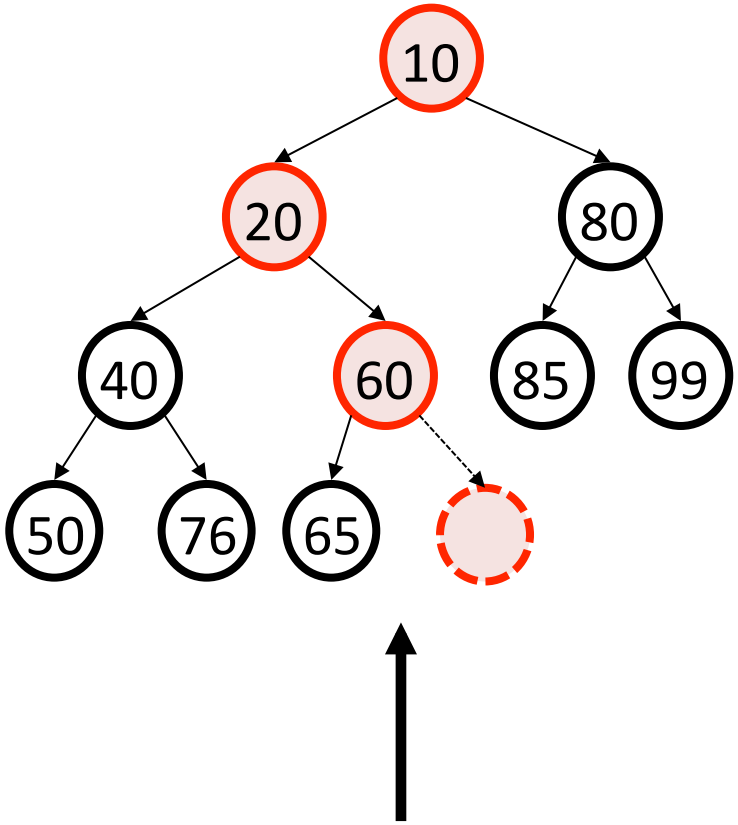


How many times can you divide n by 2?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Worst Case



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	65	0	0

size 10 capacity 13

Enqueue Big O?

$$\mathcal{O}(\log n)$$

Peek

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		**k**	m	t	x	q	a	b		
priority		**1**	5	2	5	5	8	4		

size 7 *capacity* 10

```
pq.peek() --> "k"
pq.peekPriority() --> 1
```

It is trivial to peek at the min-priority element in a heap. It is always located in index 1! What is the Big-Oh?

Peek Big O?

$$\mathcal{O}(1)$$

Heap remove (dequeue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		k	c	f	p	e	v	y		
priority		1	2	4	7	5	8	6		

size 7 *capacity* 10

```
pq.dequeue() --> "k"
```

1. Save the first value
2. Move the last element to the first
3. Bubble up the smaller of the children.
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		k	c	f	p	e	v	y		
priority		1	2	4	7	5	8	6		

size 7 *capacity* 10

```
pq.dequeue() --> "k"
```

1. **Save the first value**
2. Move the last element to the first
3. Bubble up the smaller of the children.
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

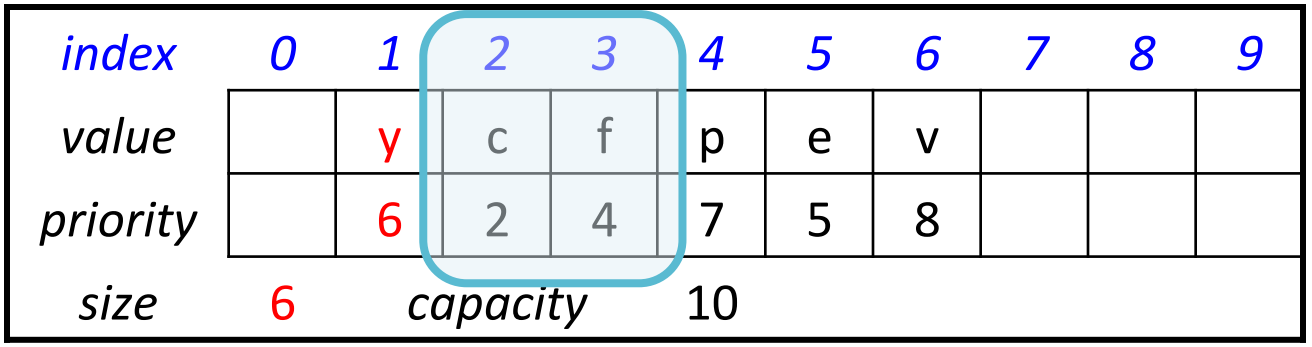
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		y	c	f	p	e	v	y		
priority		6	2	4	7	5	8	6		

size 6 *capacity* 10

```
pq.dequeue() --> "k"
```

1. Save the first value
2. **Move the last element to the first**
3. Bubble up the smaller of the children.
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)




```
pq.dequeue() --> "k”
```

1. Save the first value
2. Move the last element to the first
3. **Bubble up the smaller of the children.**
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		y	c	f	p	e	v			
priority		6	2	4	7	5	8			
size	6		*capacity*		10					

`pq.dequeue() --> "k”`

1. Save the first value
2. Move the last element to the first
3. **Bubble up the smaller of the children.**
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

index	0	1	2	3	4	5	6	7	8	9
value		c	y	f	p	e	v			
priority		2	6	4	7	5	8			

size 6 capacity 10

```
pq.dequeue() --> "k"
```

1. Save the first value
2. Move the last element to the first
3. **Bubble up the smaller of the children.**
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		c	y	f	p	e	v			
priority		2	6	4	7	5	8			
size	6		*capacity*		10					

```
pq.dequeue() --> "k”
```

1. Save the first value
2. Move the last element to the first
3. Bubble up the smaller of the children.
4. **Repeat step 3 until you get to the bottom of the tree.**

Heap remove (dequeue)

index	0	1	2	3	4	5	6	7	8	9
value		c	y	f	p	e	v			
priority		2	6	4	7	5	8			

size 6 capacity 10

```
pq.dequeue() --> "k"
```

1. Save the first value
2. Move the last element to the first
3. **Bubble up the smaller of the children.**
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

index	0	1	2	3	4	5	6	7	8	9
value		c	y	f	p	e	v			
priority		2	6	4	7	5	8			

size 6 capacity 10

```
pq.dequeue() --> "k"
```

1. Save the first value
2. Move the last element to the first
3. **Bubble up the smaller of the children.**
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

index	0	1	2	3	4	5	6	7	8	9
value		c	e	f	p	y	v			
priority		2	5	4	7	6	8			

size 6 capacity 10

`pq.dequeue() --> "k"`

1. Save the first value
2. Move the last element to the first
3. **Bubble up the smaller of the children.**
4. Repeat step 3 until you get to the bottom of the tree.

Heap remove (dequeue)

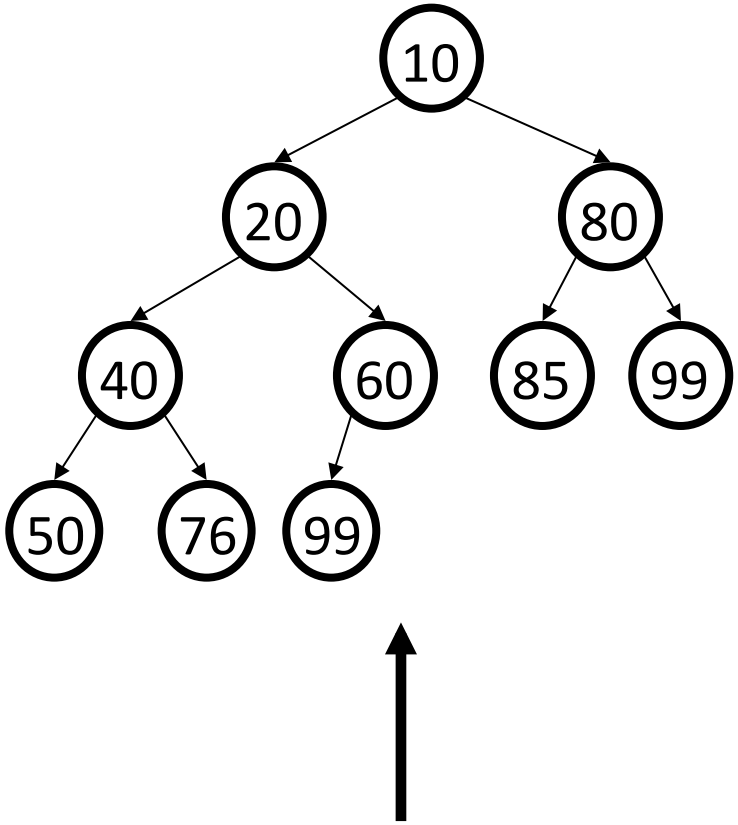
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		c	e	f	p	y	v			
priority		2	5	4	7	6	8			
size	6	*capacity*		10						

```
pq.dequeue() --> "k"
```

1. Save the first value
2. Move the last element to the first
3. Bubble up the smaller of the children.
4. **Repeat step 3 until you get to the bottom of the tree.**

Big O?

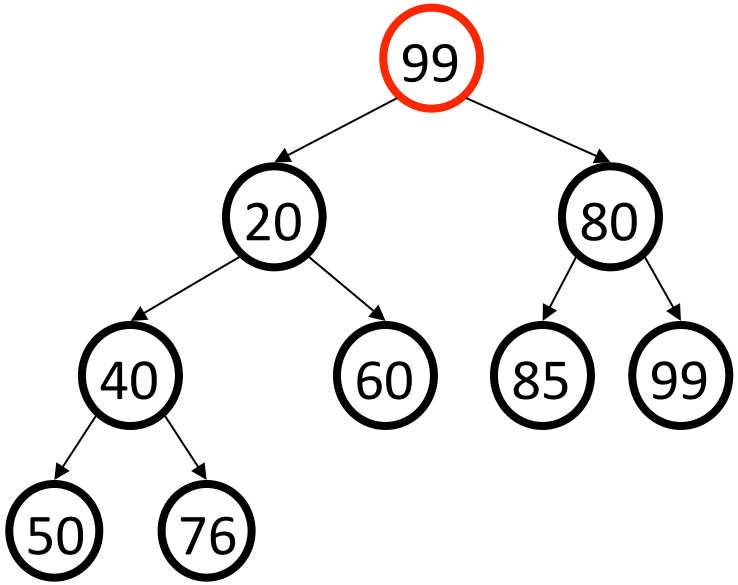
Worst Case



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	99	0	0

size 10 capacity 13

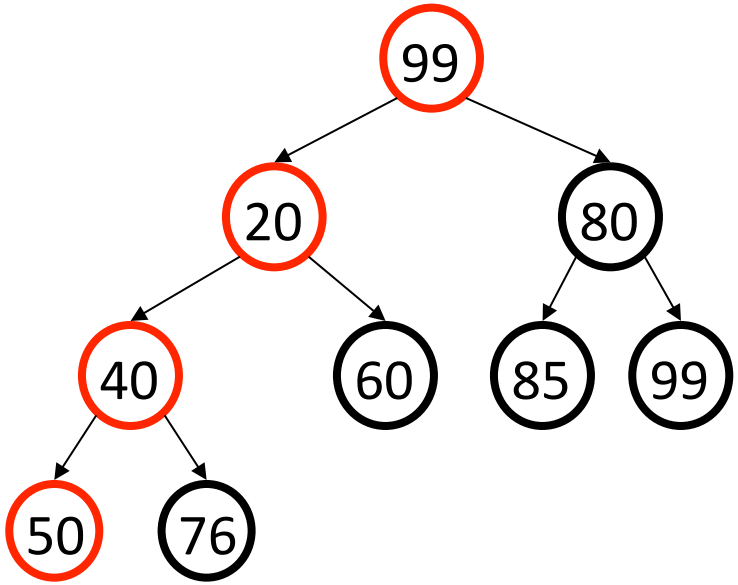
Worst Case



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	99	0	0

size 10 capacity 13

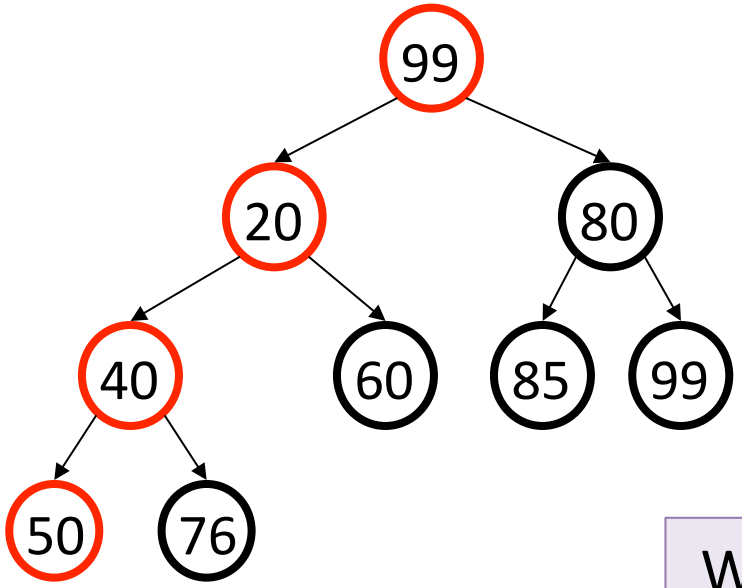
Worst Case



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	99	0	0

size 10 capacity 13

Worst Case

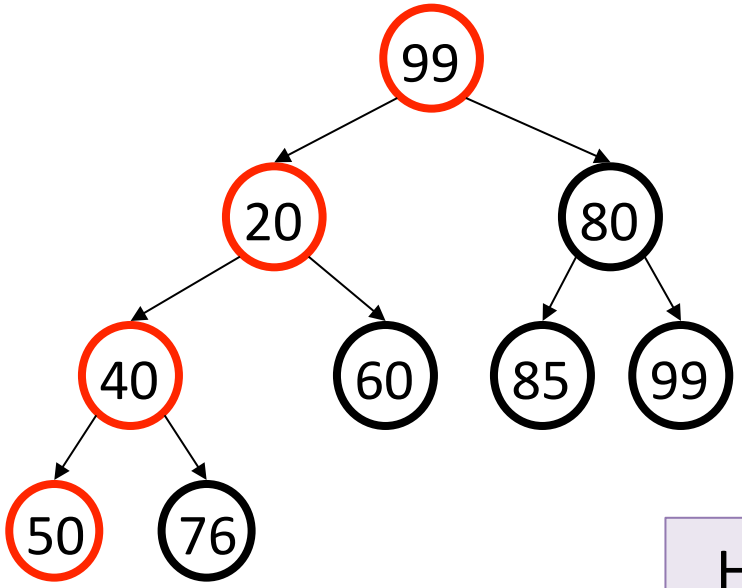


What is the depth of the tree?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	99	0	0

size 10 capacity 13

Worst Case



How many times can you divide n by 2?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	0	10	20	80	40	60	85	99	50	76	99	0	0

size 10 capacity 13

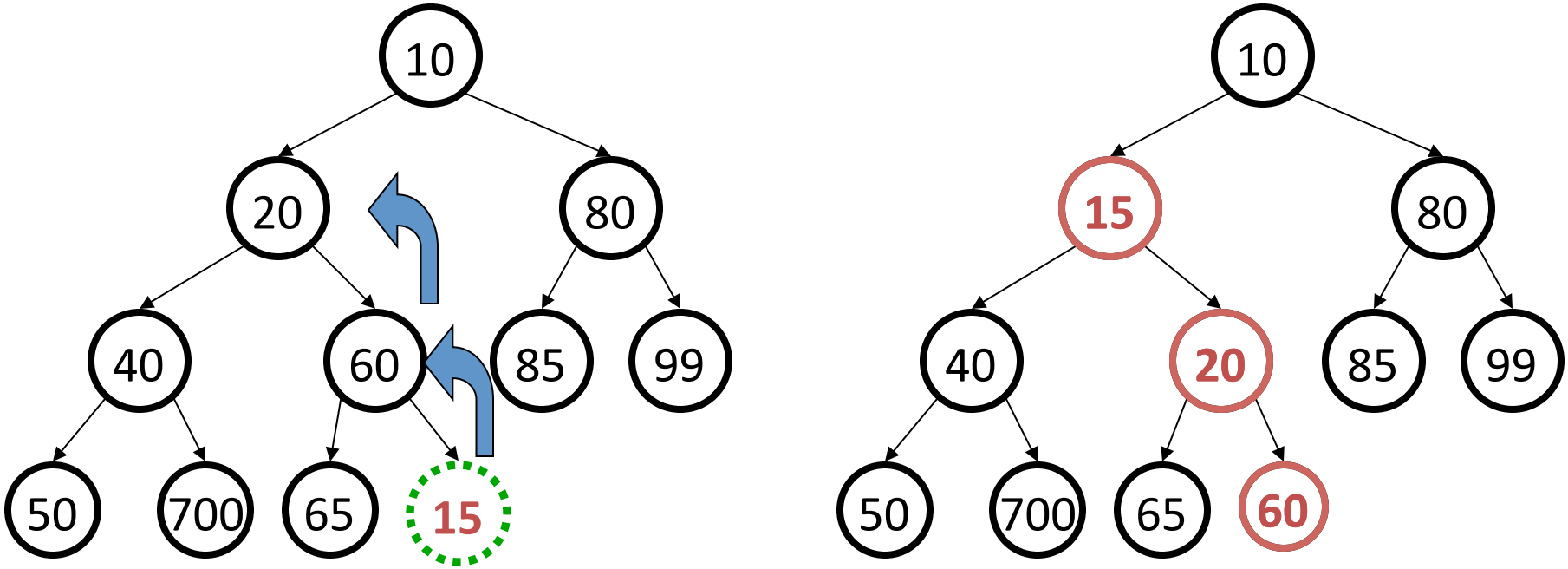
Enqueue Big O?

$$\mathcal{O}(\log n)$$

See Enqueue / Dequeue
as Trees

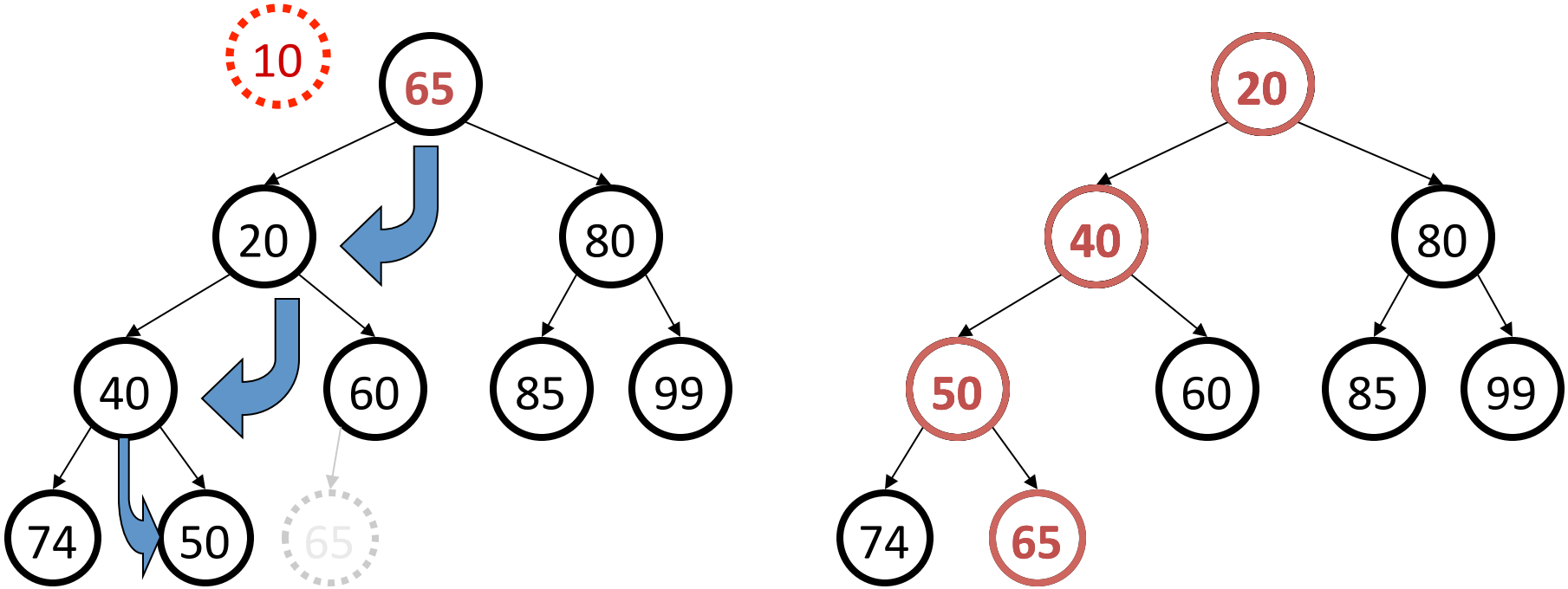
Enqueue

- When adding to a heap, the value is first placed at bottom-right.
  - To restore heap ordering, the newly added element is shifted ("**bubbled**") up the tree until it reaches its proper place.
  - Enqueue 15 at bottom-right; bubble up until in order.



Dequeue

- To restore heap order, the improper root is shifted ("bubbled") down the tree by swapping with its smaller (higher-pri) child.
  - dequeue min of 10; swap up bottom-right leaf of 65; bubble down.

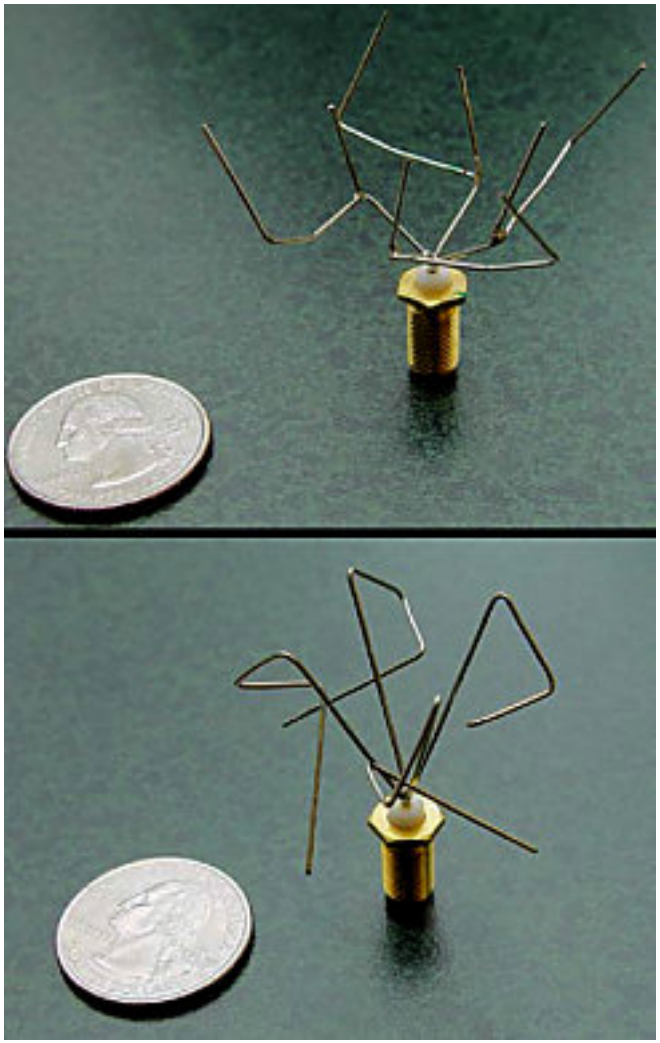


Today's Goals

Today's Goals

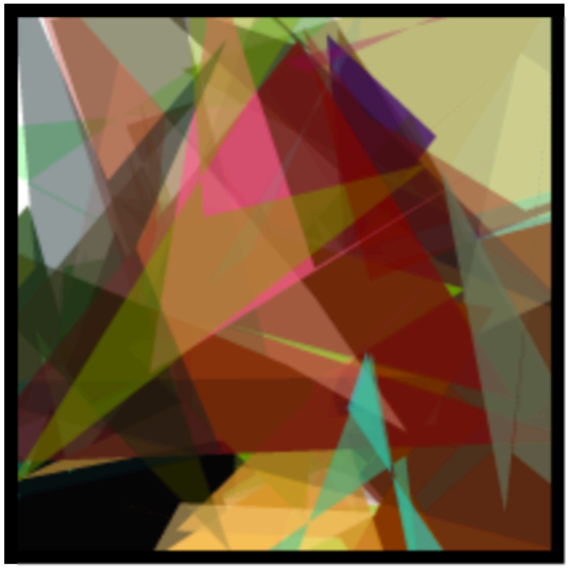
Evolutionary Algorithms

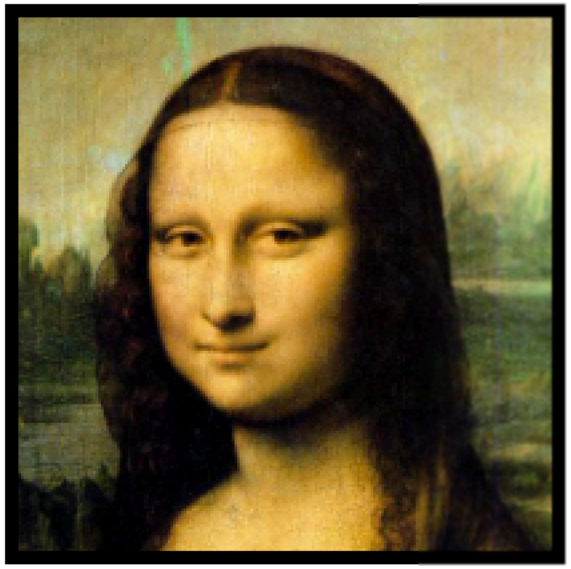
Antennae for microsatellites (~25kg)

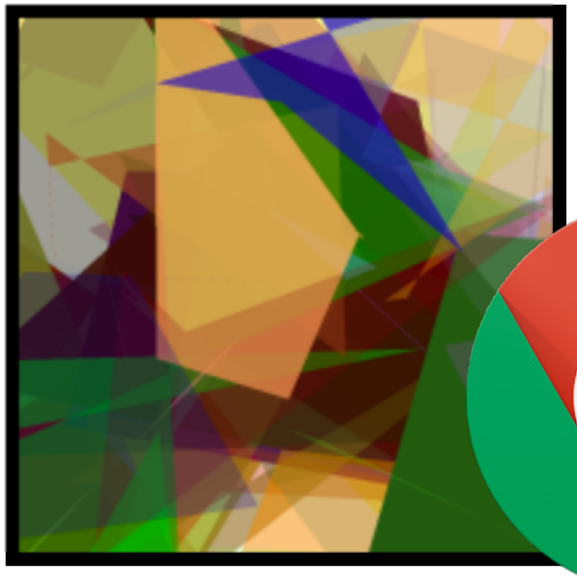


2006

Evolving Mona Lisa

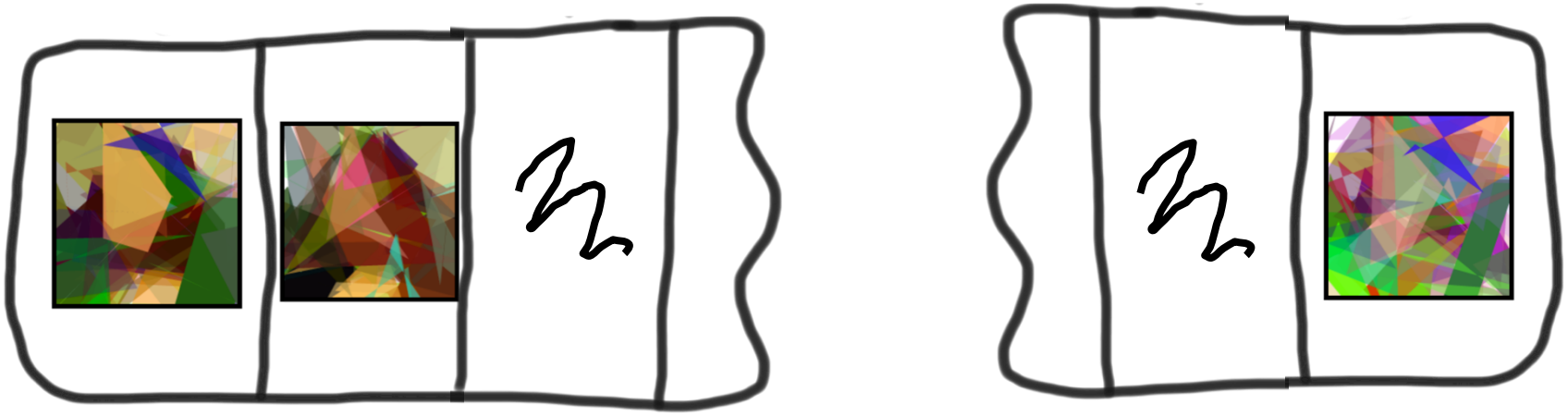






Evolution PQueue

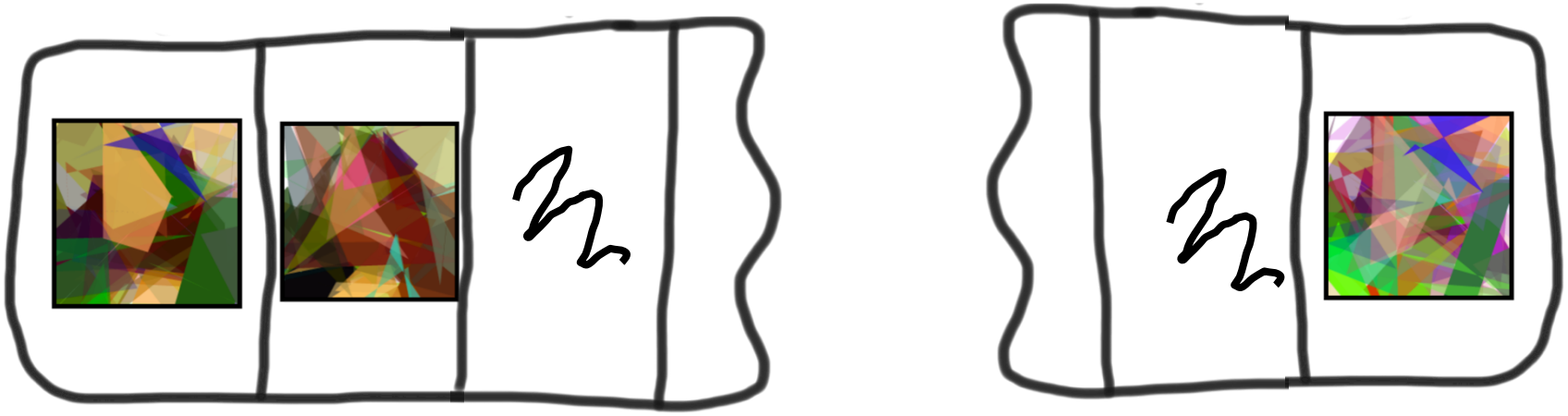
```
PQueue<Species *> generation
```



Priority is how different the image is from the target

Natural Selection

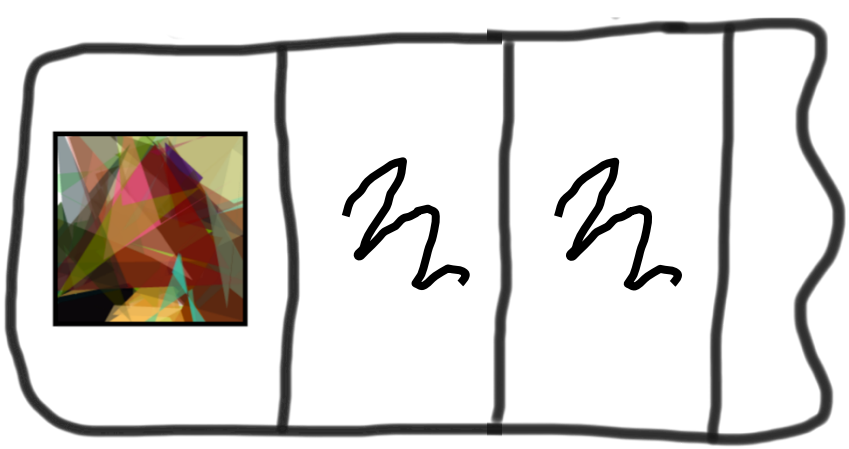
```
PQueue<Species *> generation
```

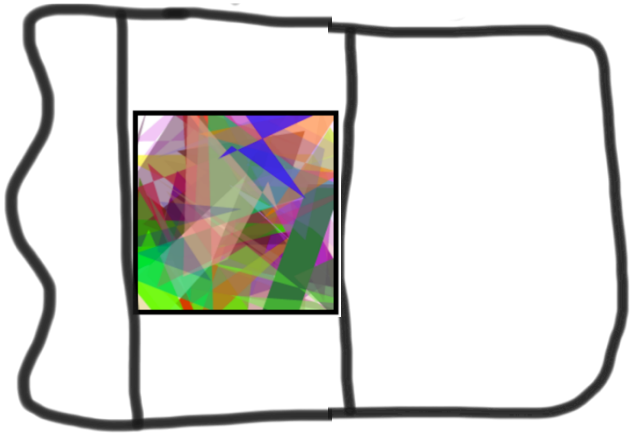


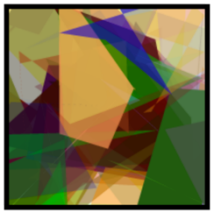
Dequeue (and save) the top
M species

Natural Selection

```
PQueue<Species *> generation
```



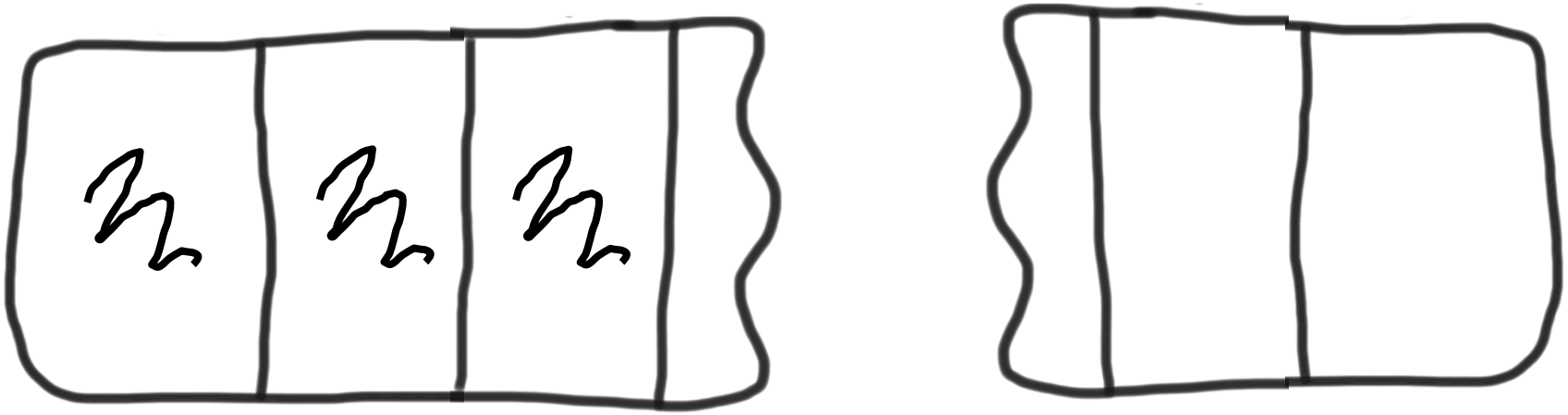


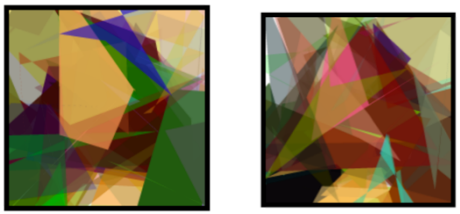


Dequeue (and save) the top
M species

Natural Selection

```
PQueue<Species *> generation
```

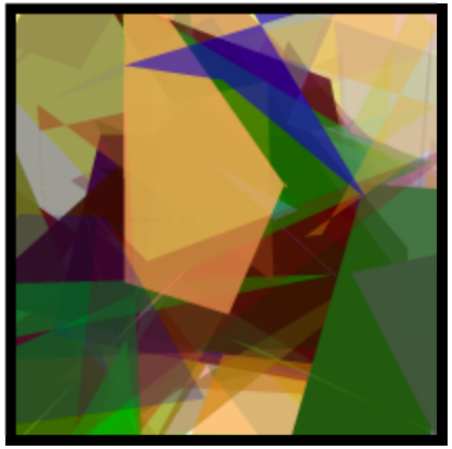




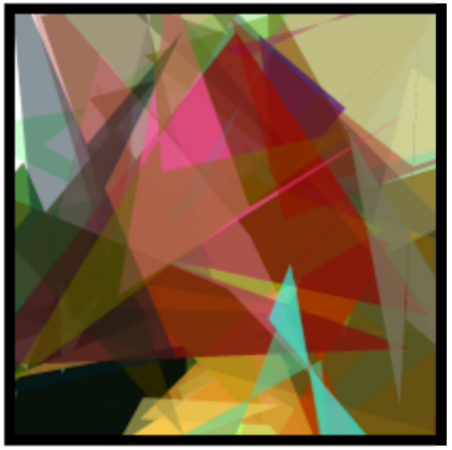
Dequeue (and save) the top
M species

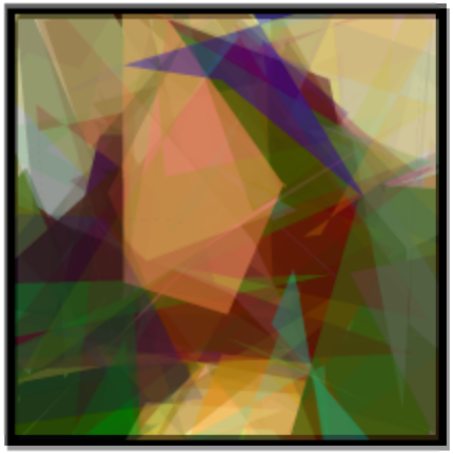
Mate

Species * parent1



Species * parent2

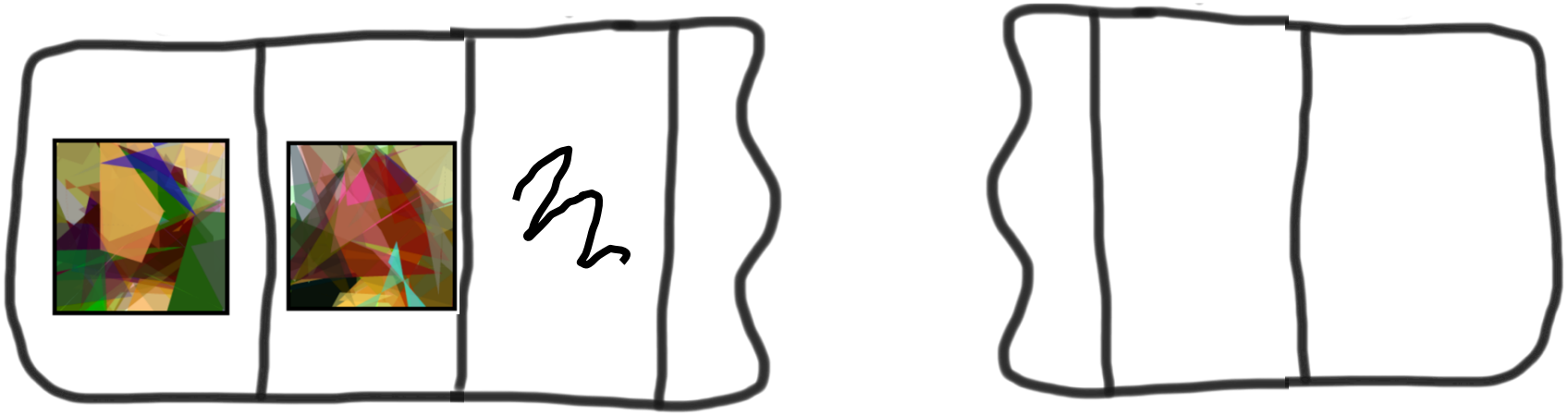




Species * child

Repopulate

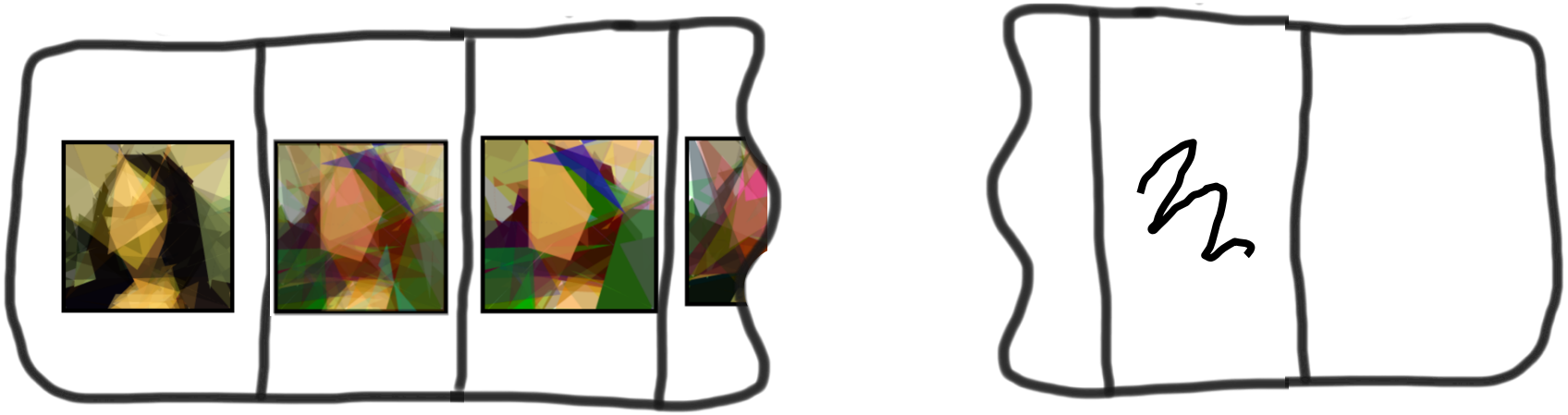
```
PQueue<Species *> generation
```



Enqueue all parents

Repopulate

```
PQueue<Species *> generation
```



Enqueue all children

Repeat

Evolutionary Algorithms

```
Pqueue<Species *> alive;
populate pqueue with random species

while(true){
  dequeue the best species as parents
  mate the parents to produce new children
  start a new generation
  enqueue the parents and the children
}
```

Evolutionary Algorithms

```
Pqueue<Species *> alive;
for(int i = 0; i < N_POP; i++) {
  Species * starter = new Species();
  alive->enqueue(starter, starter->getFitness());
}

while(true){
  Vector<Species *> parents;
  for(int i = 0; i < N_SURVIVORS; i++) {
    parents.add(alive->dequeue());
  }
  Vector<Species *> children = mate(parents, N_CHILDREN);
  alive.clear();
  for(Species * p : parents) {
    alive.enqueue(p, p->getFitness());
  }
  for(Species * c : children) {
    alive.enqueue(c, c->getFitness());
  }
}
```

Evolutionary Algorithms

```
Pqueue<Species *> alive;
for(int i = 0; i < N_POP; i++) {
  Species * starter = new Species();
  alive->enqueue(starter, starter->getFitness());
}
```

```
while(true){
  Vector<Species *> parents;
  for(int i = 0; i < N_SURVIVORS; i++) {
    parents.add(alive->dequeue());
  }
  Vector<Species *> children = mate(parents, N_CHILDREN);
  alive.clear();
  for(Species * p : parents) {
    alive.enqueue(p, p->getFitness());
  }
  for(Species * c : children) {
    alive.enqueue(c, c->getFitness());
  }
}
```

Evolutionary Algorithms

```
Pqueue<Species *> alive;
for(int i = 0; i < N_POP; i++) {
  Species * starter = new Species();
  alive->enqueue(starter, starter->getFitness());
}
```

```
while(true){
  Vector<Species *> parents;
  for(int i = 0; i < N_SURVIVORS; i++) {
    parents.add(alive->dequeue());
  }
  Vector<Species *> children = mate(parents, N_CHILDREN);
  alive.clear();
  for(Species * p : parents) {
    alive.enqueue(p, p->getFitness());
  }
  for(Species * c : children) {
    alive.enqueue(c, c->getFitness());
  }
}
```

Evolutionary Algorithms

```
Pqueue<Species *> alive;
for(int i = 0; i < N_POP; i++) {
  Species * starter = new Species();
  alive->enqueue(starter, starter->getFitness());
}

while(true){
  Vector<Species *> parents;
  for(int i = 0; i < N_SURVIVORS; i++) {
    parents.add(alive->dequeue());
  }
  Vector<Species *> children = mate(parents, N_CHILDREN);
  alive.clear();
  for(Species * p : parents) {
    alive.enqueue(p, p->getFitness());
  }
  for(Species * c : children) {
    alive.enqueue(c, c->getFitness());
  }
}
```

Evolutionary Algorithms

```
Pqueue<Species *> alive;
for(int i = 0; i < N_POP; i++) {
  Species * starter = new Species();
  alive->enqueue(starter, starter->getFitness());
}

while(true){
  Vector<Species *> parents;
  for(int i = 0; i < N_SURVIVORS; i++) {
    parents.add(alive->dequeue());
  }
  Vector<Species *> children = mate(parents, N_CHILDREN);
  alive.clear();
  for(Species * p : parents) {
    alive.enqueue(p, p->getFitness());
  }
  for(Species * c : children) {
    alive.enqueue(c, c->getFitness());
  }
}
```

Evolutionary Algorithms

```
Pqueue<Species *> alive;
for(int i = 0; i < N_POP; i++) {
  Species * starter = new Species();
  alive->enqueue(starter, starter->getFitness());
}

while(true){
  Vector<Species *> parents;
  for(int i = 0; i < N_SURVIVORS; i++) {
    parents.add(alive->dequeue());
  }
  Vector<Species *> children = mate(parents, N_CHILDREN);
  alive.clear();
  for(Species * p : parents) {
    alive.enqueue(p, p->getFitness());
  }
  for(Species * c : children) {
    alive.enqueue(c, c->getFitness());
  }
}
```

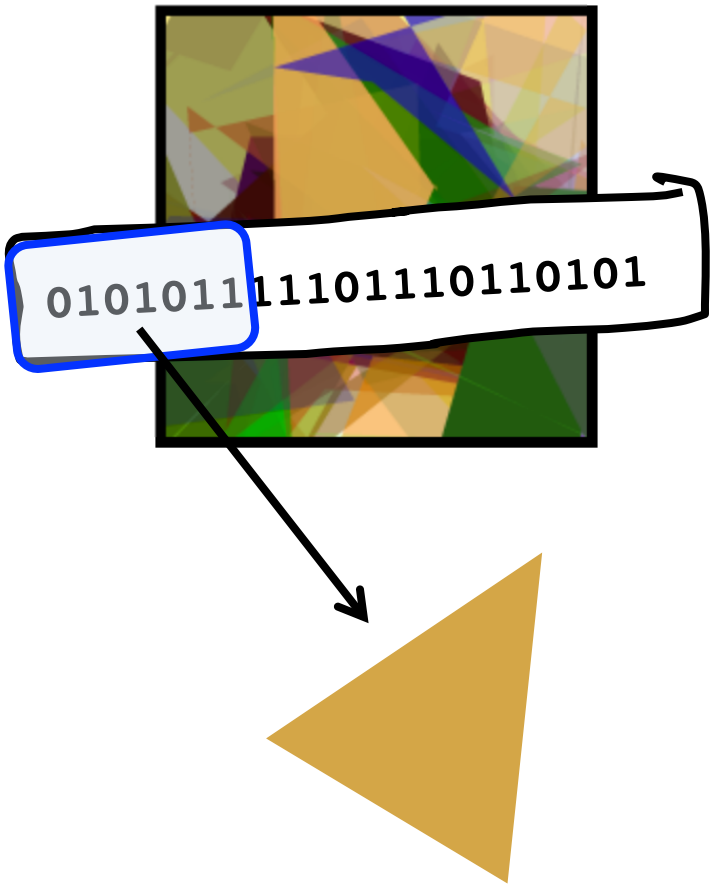
Mate

Species * parent1

Species * parent2

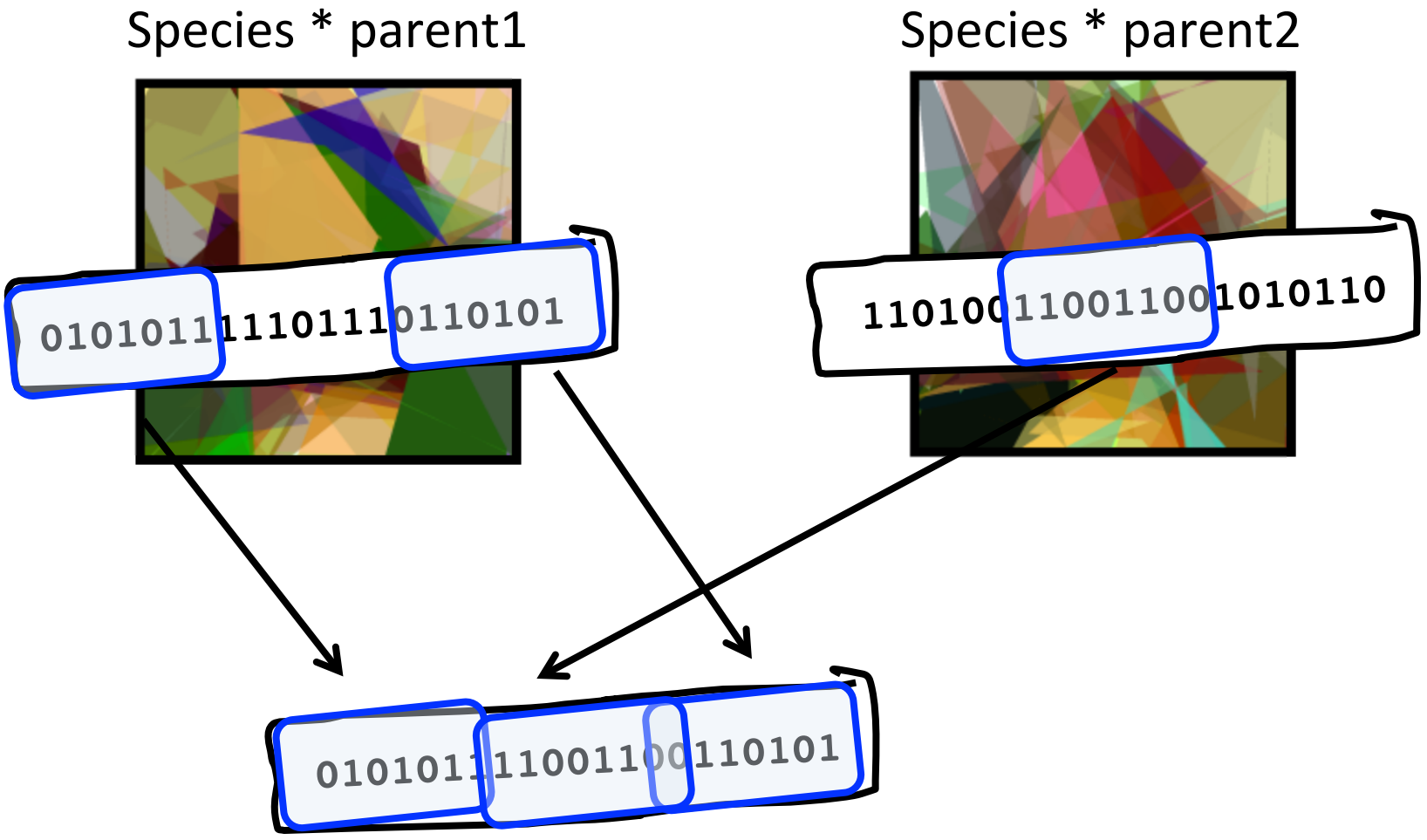
DNA

Species * parent1

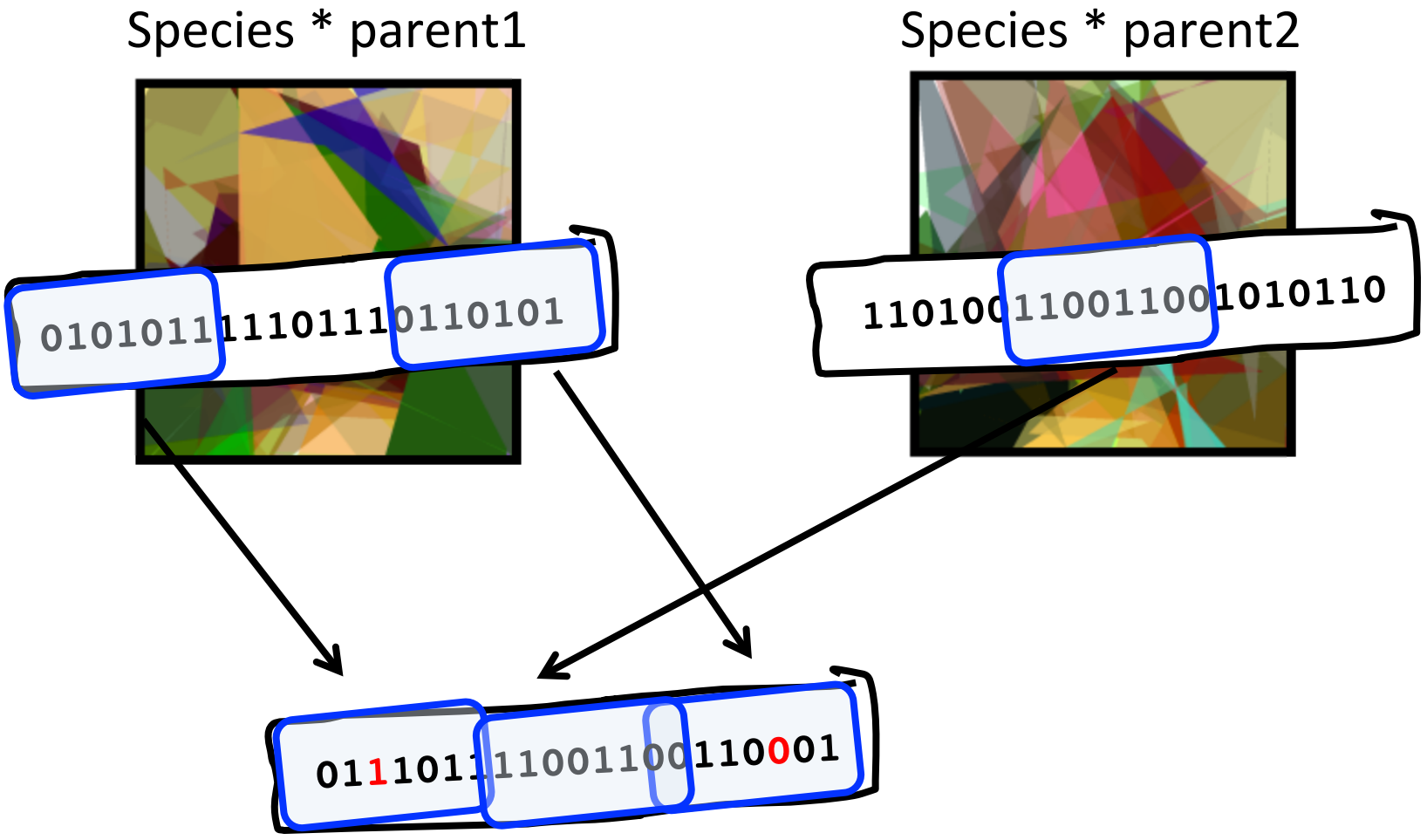


Species * parent2

Crossover


Species * parent1
Species * parent2
0101011 1110111 0110101
1101001 1001100 1010110
0101011 1100110 0110101

Mutation


Species * parent1
Species * parent2
01010111110111 0110101
1101001100110010101 10
01110111001100110001

Mate

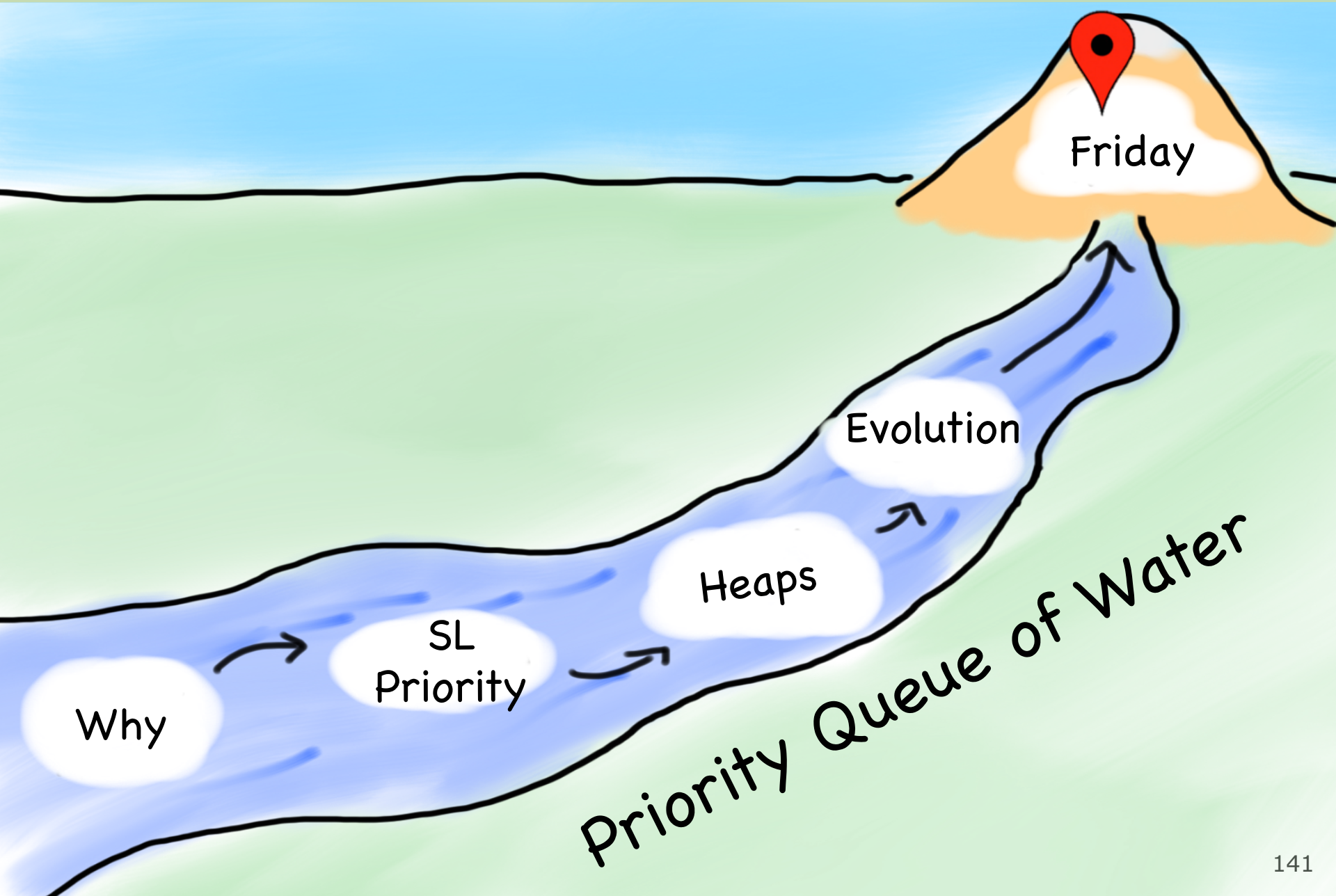
Species * parent1

Species * parent2

Species * child

Today's Goals

Today's Goals



Today's Goals

1. Care About Priority Queues
2. Learn about Heaps