Priority Queues and Heaps

Elyse Cornwall

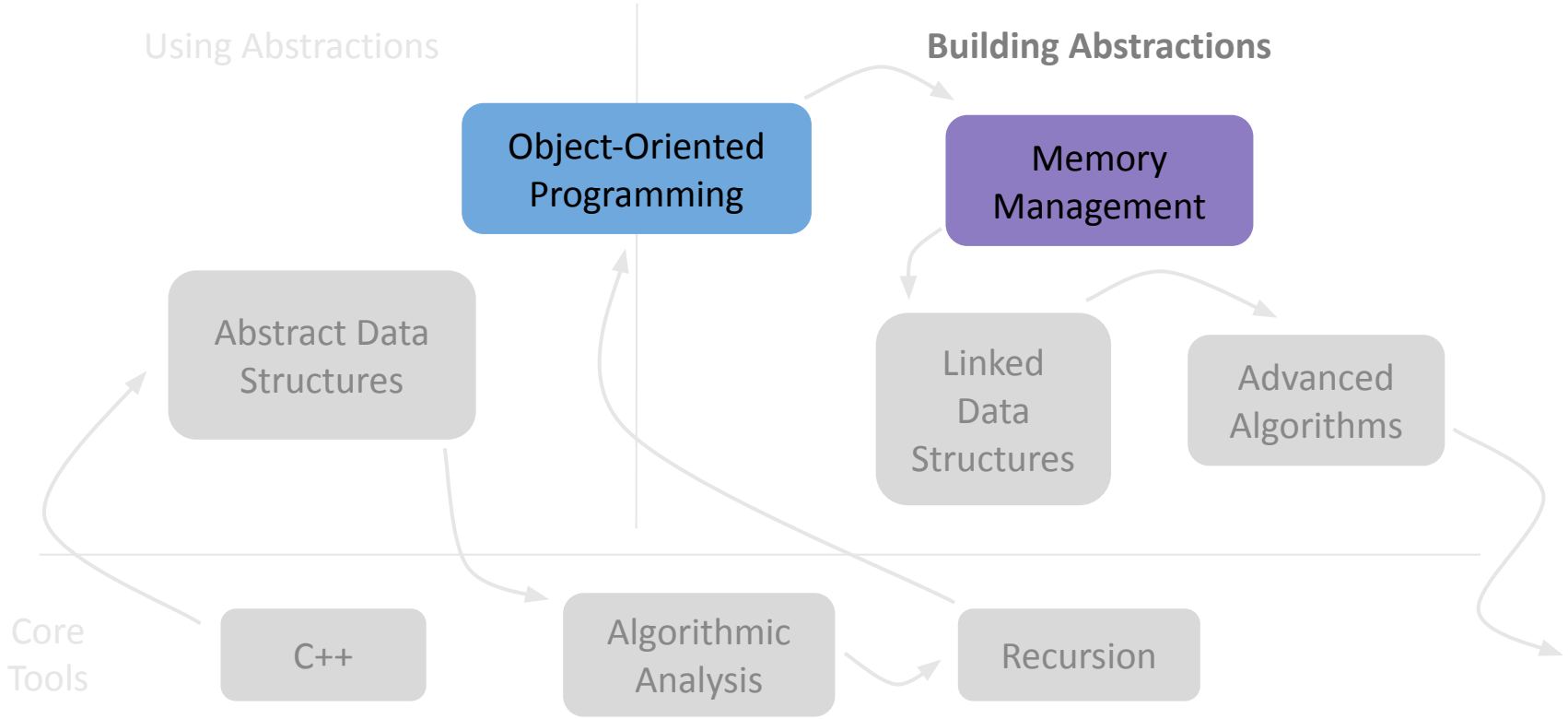
July 26, 2023

Contributions made from previous CS106B Instructors

Announcements

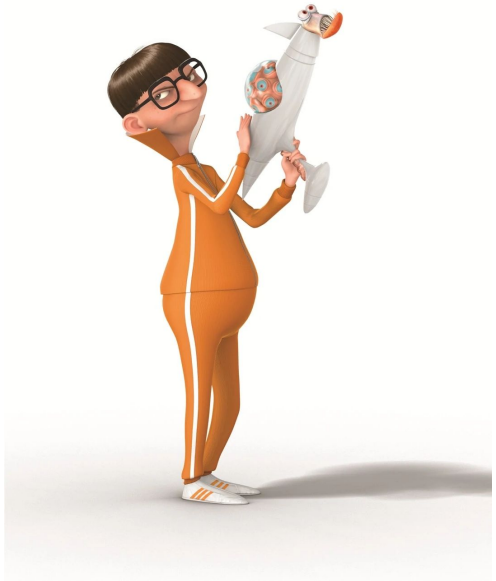
- Assignment 3 is due tonight at 11:59pm
- Assignment 4 will be released this afternoon
  - YEAH Hours today from 3-4pm at this zoom link
  - For parts of the assignment you'll need tomorrow's lecture on pointers
- Midterm regrades must be submitted by tonight at 11:59pm
  - All regrade requests will be handled by the weekend

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Recap: Implementing Classes

Designing `OurVector`



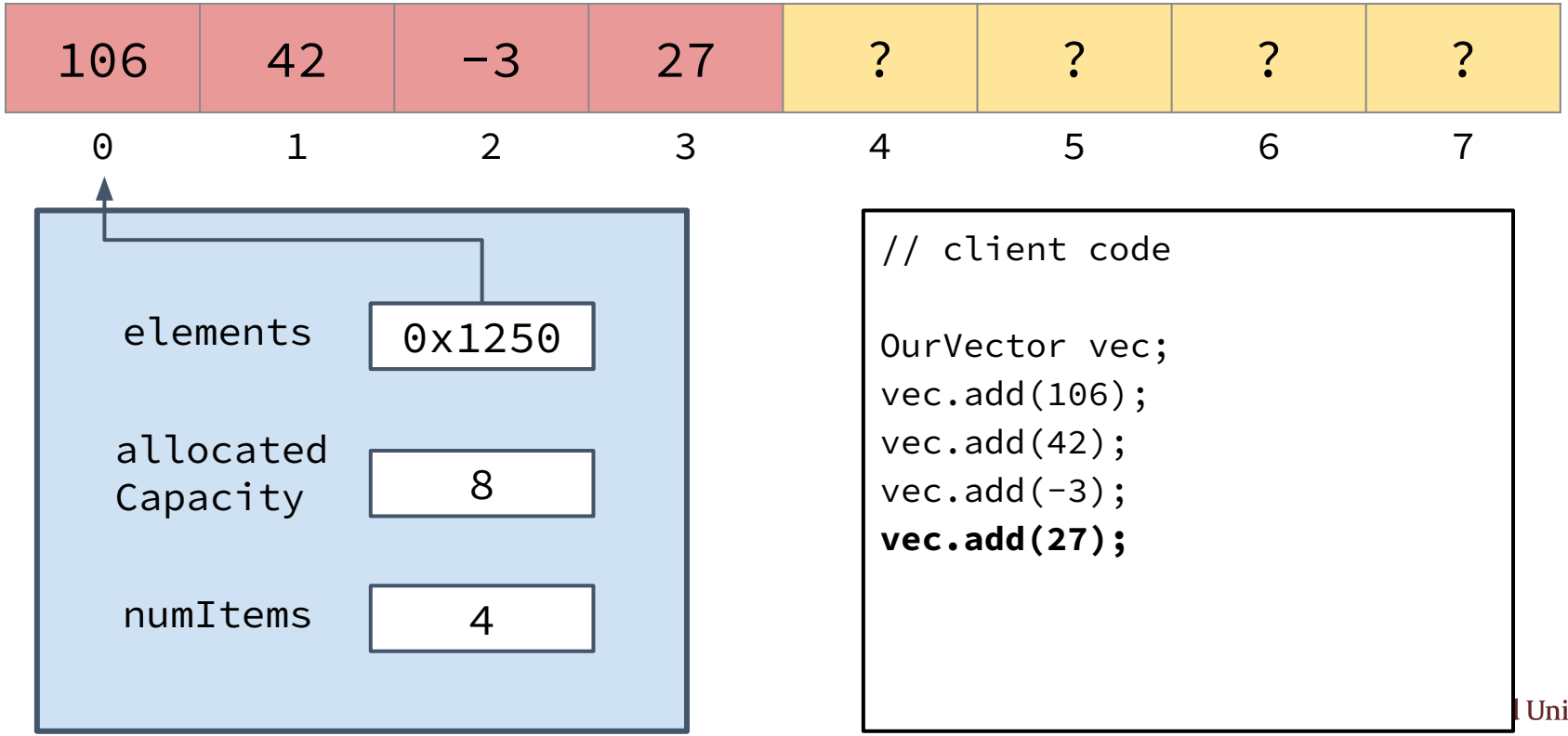
OurVector Header File

```
class OurVector {
public:
    OurVector();
    ~OurVector();
    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();
private:
    int* elements;
    int allocatedCapacity;
    int numItems;
};
```

Arrays

- Lower-level and more limited than Vectors (no member functions!)
- Chunk of space in the computer's memory, split into slots, each of which can contain one piece of information
  - Have a specific type which dictates what information can be held in each slot
  - Each slot has an "index" by which we can refer to it
  - Don’t dynamically resize like the Vector ADT we’re used to
  - We’ll create arrays on the heap, so they persist until we delete them

Adding Elements



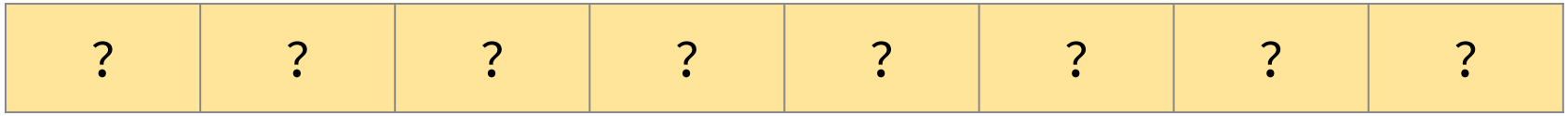

```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

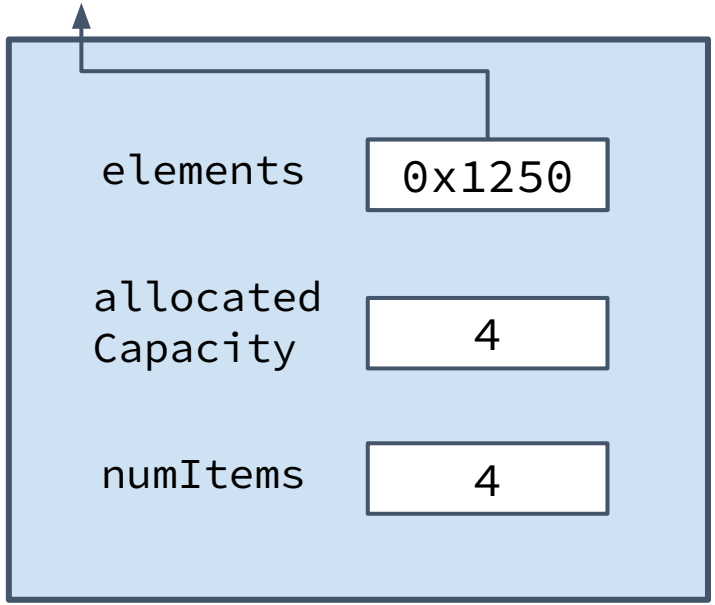
Adding Elements

```
void OurVector::add(int value){
   if (numItems == allocatedCapacity) {
      expand();
   }
   elements[numItems] = value;
   numItems++;
}
```

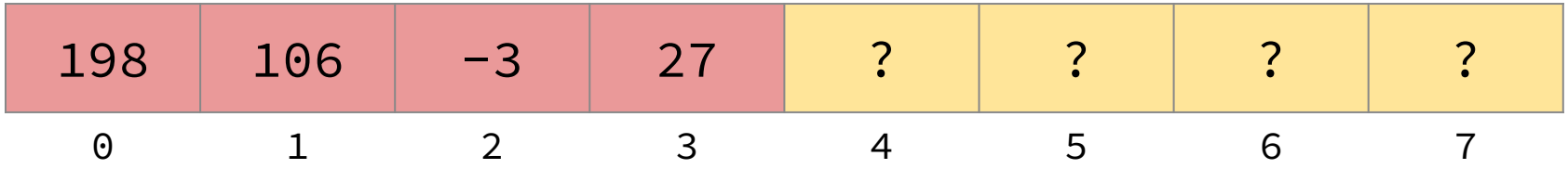
Adding Elements

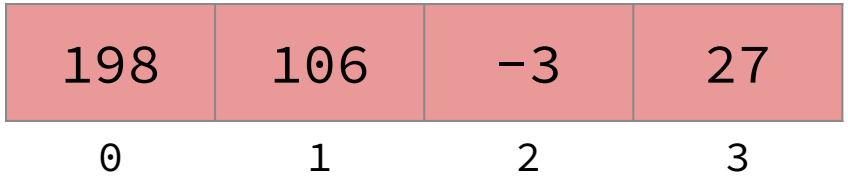
```
void OurVector::add(int value){
   if (numItems == allocatedCapacity) {
      expand();
   }
   elements[numItems] = value;
   numItems++;
}
```

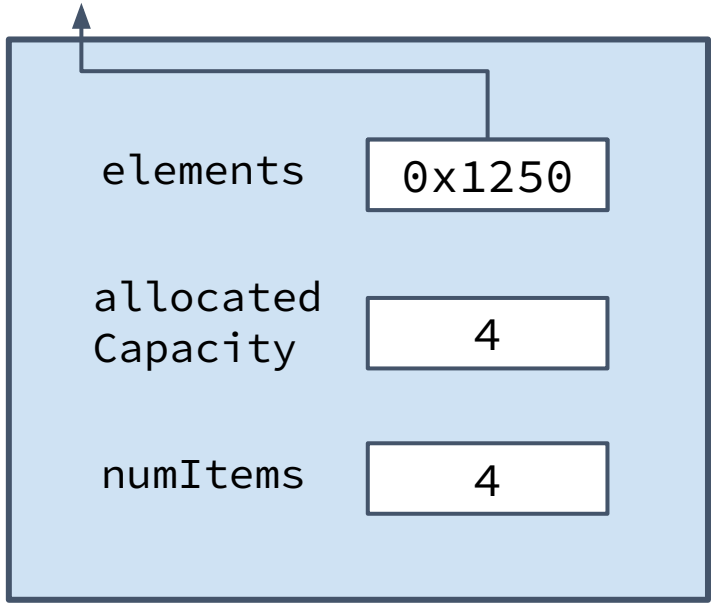




1. Create a new, larger array (usually we choose to double the current size)

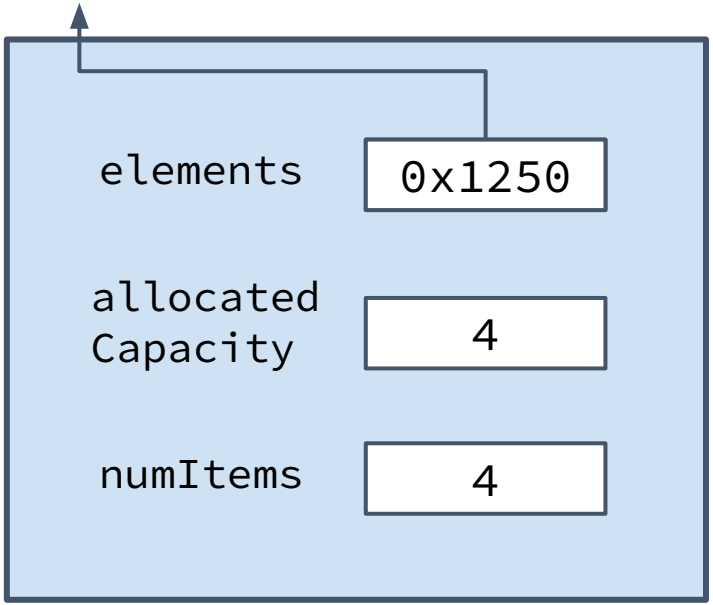




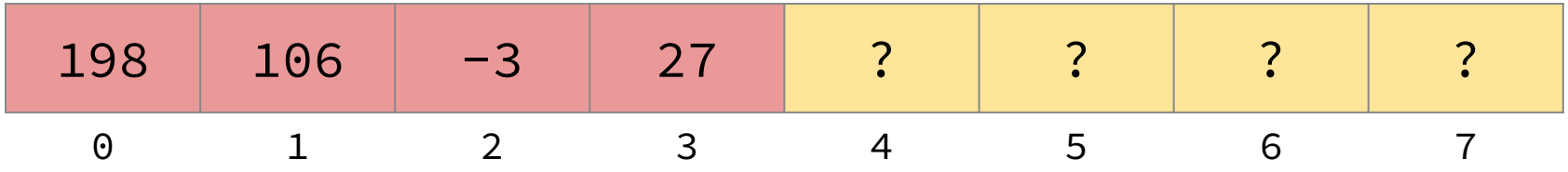


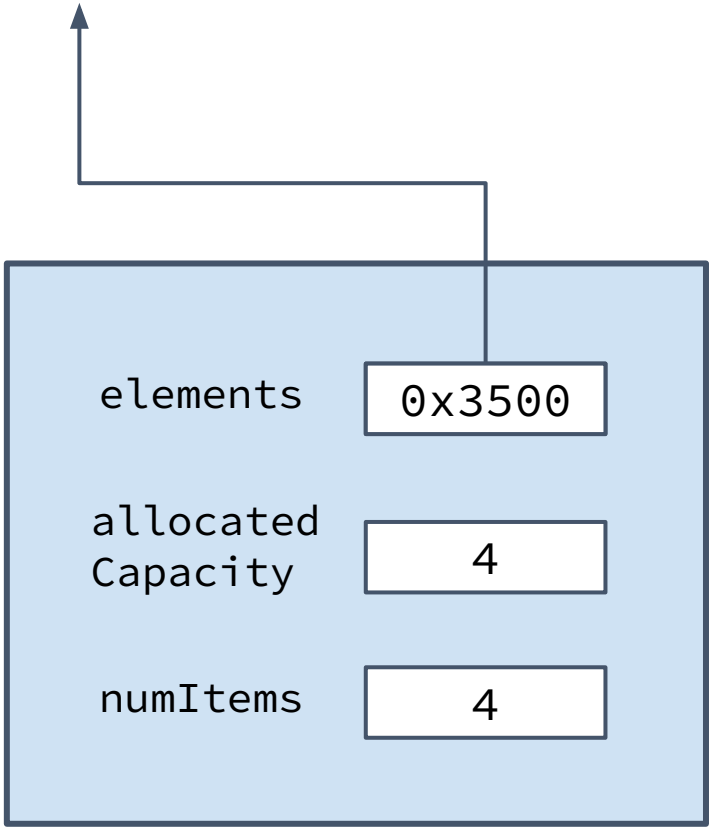
1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array

198	106	-3	27	?	?	?	?
0	1	2	3	4	5	6	7

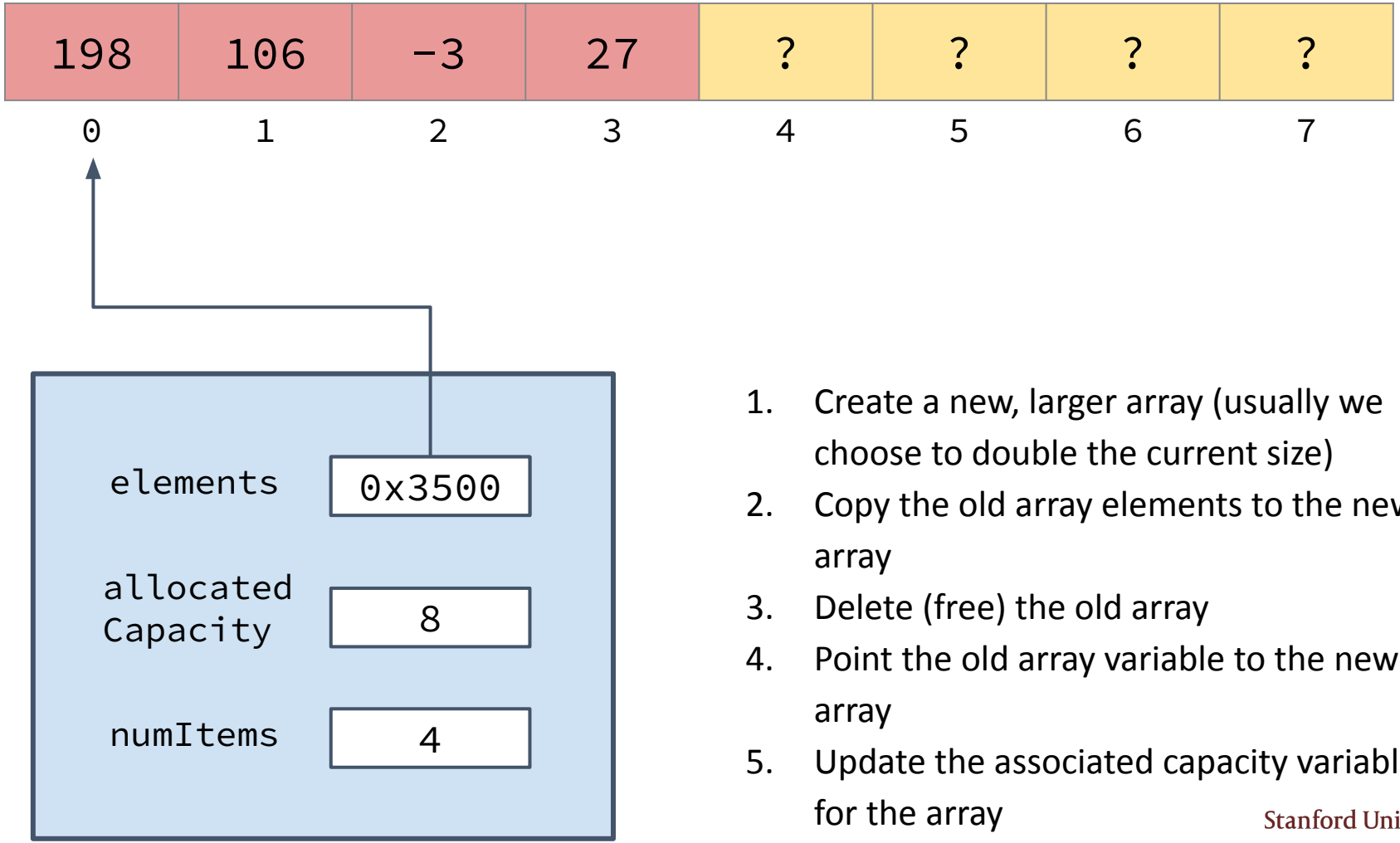


1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array





1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array
4. Point the old array variable to the new array



1. Create a new, larger array (usually we choose to double the current size)
2. Copy the old array elements to the new array
3. Delete (free) the old array
4. Point the old array variable to the new array
5. Update the associated capacity variable for the array

Priority Queues

What's a Priority Queue?

- A queue that sorts its elements based on their priority
- Like regular queues, you can only access the element at the front
  - No indices
- Good way to model ER waiting rooms, organ matches, vaccine availability

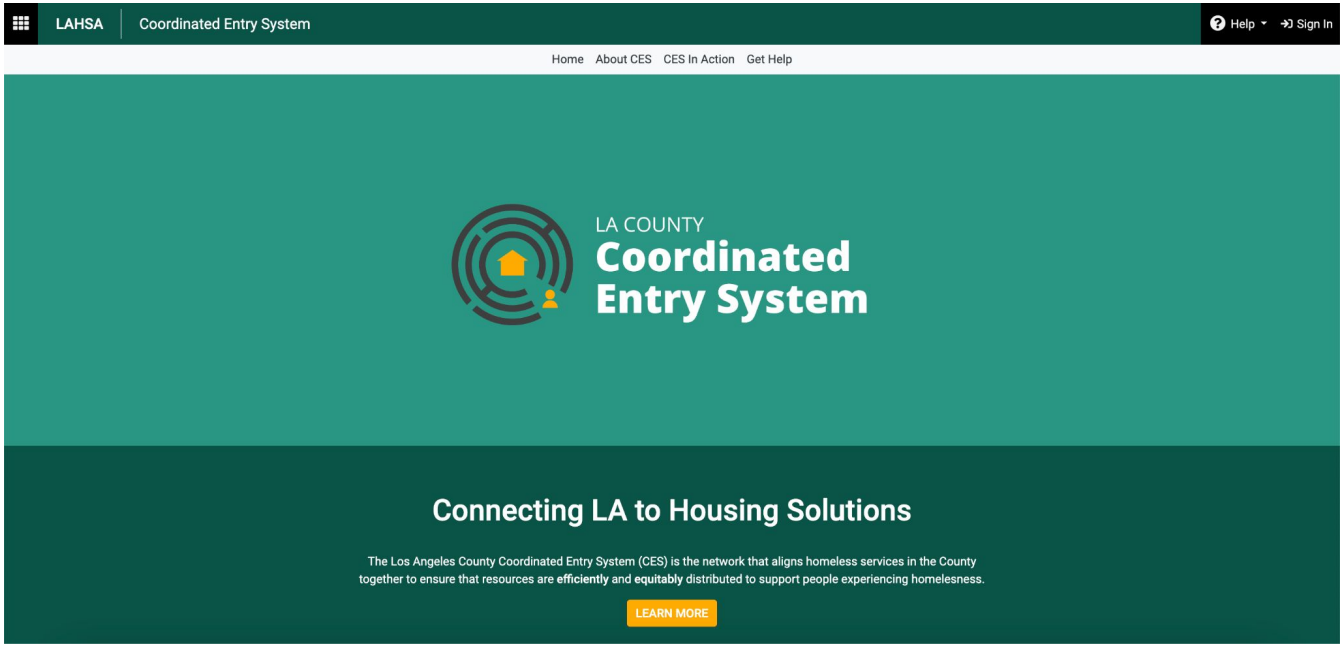
Priority Queue Uses

- ER waiting rooms, organ matches, vaccine availability
- Airplane boarding groups
- Social media feed
- College admissions
- Welfare allocation

What do we prioritize?

Based on slides by Katie Creel and Diana Acosta-Navas

Priority Queues in Action: LA CES



An electronic registry of people experiencing homelessness who are applying or have applied to housing support programs offered by Los Angeles County

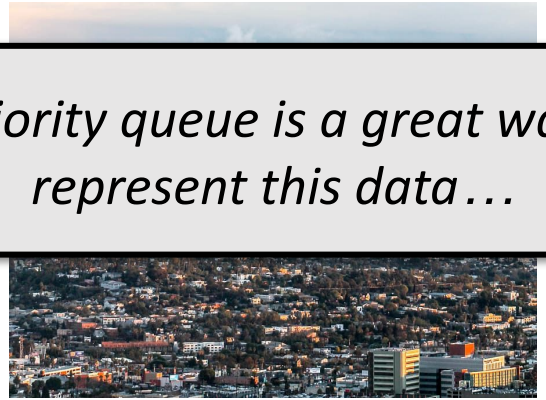
Priority Queues in Action: LA CES

- Algorithm uses personal data to assign a number from 1 to 17, least to most vulnerable
- This risk score is used to prioritize certain individuals when assigning housing and housing-related services

Priority Queues in Action: LA CES

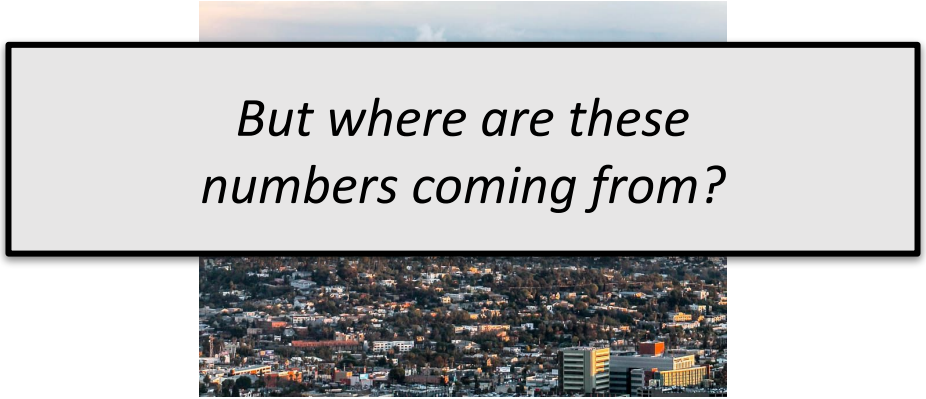
- Algorithm uses personal data to assign a number from 1 to 17, least to most vulnerable
- This risk score is used to prioritize certain individuals when assigning housing and housing-related services

A priority queue is a great way to represent this data…



Priority Queues in Action: LA CES

- Algorithm uses personal data to **assign a number from 1 to 17,** least to most vulnerable
- This risk score is used to prioritize certain individuals when assigning housing and housing-related services



What should we prioritize?

- Fairness and equality?
- Justice?
- Speed and efficiency?
- How do we use demographic information?

What should we prioritize?

- Fairness and equality?
- Justice?
- Speed and efficiency?
- How do we use demographic information?

As we study priority queues, think about what values are being represented when we makes decisions about priority.

Priority Queue Operations

Three basic operations:

- `peek()` - returns the element with the highest priority in the queue without removing it
- `enqueue(elem, priority)` - inserts elem with given priority
- `dequeue()` - removes and returns the element with the highest priority from the queue

Priority Queue Operations

Three more handy operations:

- `size()` - returns the number of elements in the queue
- `isEmpty()` - returns `true` if there are no elements in the queue, `false` otherwise
- `clear()` - empties the queue

Implementing a Priority Queue

Client side

We need these basic operations:

- `peek()`
- `enqueue(elem, priority)`
- `dequeue()`

Implementation side

How should we design our PQ?

Implementing a Priority Queue

Member functions: What functions might a client want to call? ✅

Member variables: What private information will we need to store in order to keep track of the data stored in a Priority Queue?

Constructor: How are the member variables initialized when a new Priority Queue is created?

We already told you what important member functions to implement (peek, enqueue, dequeue)… you decide the rest!

Implementing a Priority Queue

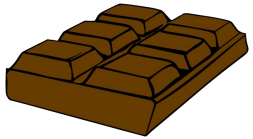
- We want our basic operations to be pretty fast (< `O(n)` if possible!)
- There are many ways we could implement a PQ, but they will have tradeoffs in terms of efficiency

Our PQ Data

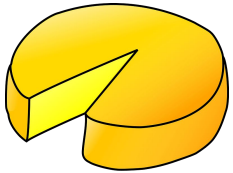
- Each data point has a value and a priority

Our PQ Data

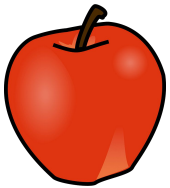
- Each data point has a value and a priority
- Let's say we're prioritizing *health* by giving each food a health ranking between 1-10 (1 is high, 10 is low)



Chocolate, 8



Cheese, 7



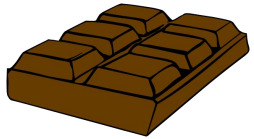
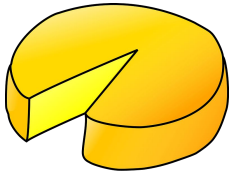
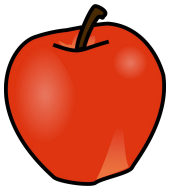
Apple, 3

Kale, 1

First Attempt: Sorted Array

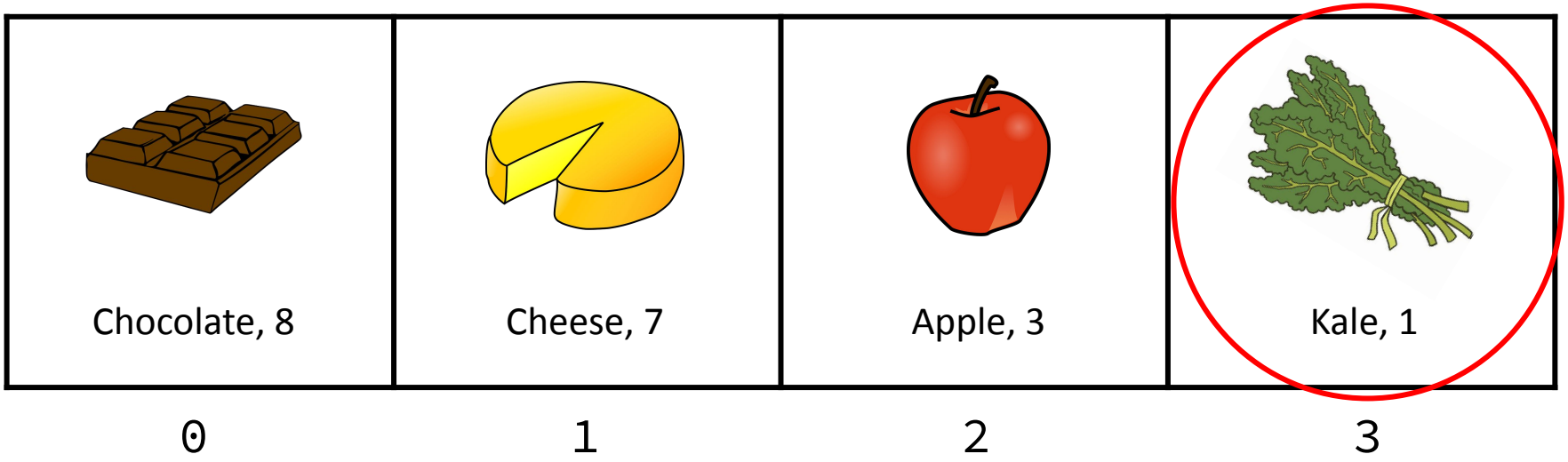
PQ Array

👥 *What's the Big-O runtime of our three basic operations? Peek, enqueue, and dequeue?*

			
Chocolate, 8	Cheese, 7	Apple, 3	Kale, 1
0	1	2	3

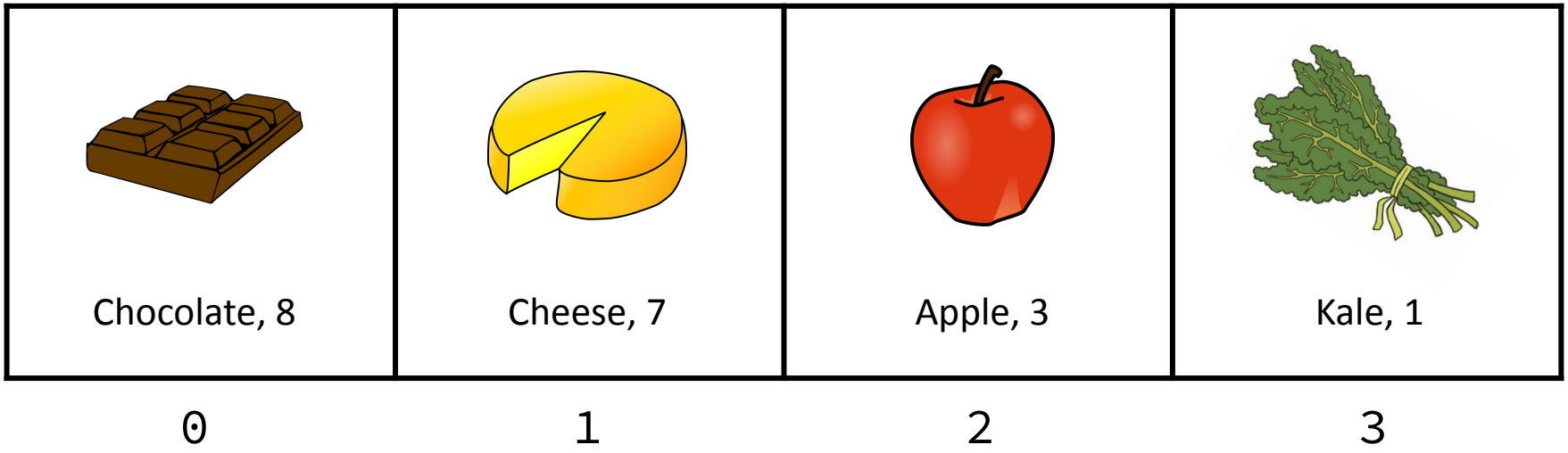
PQ Array - `peek()`

- Return the last (highest priority) element of the array
- This is `O(1)`, we just check what's at the last index of our array

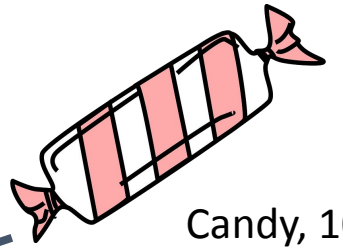


PQ Array - enqueue()

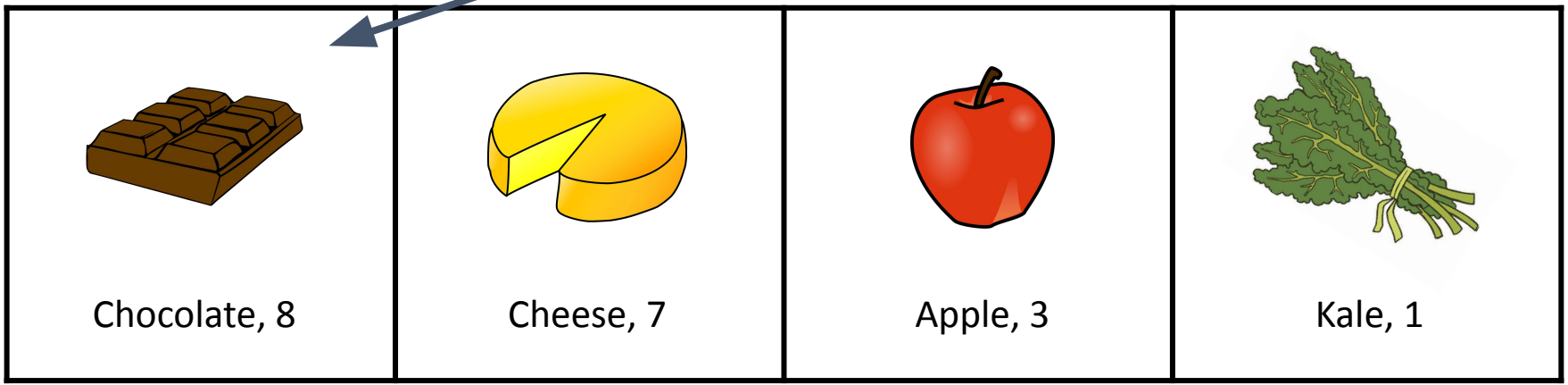
- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over



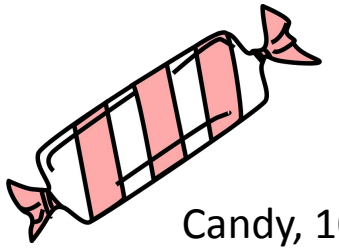
PQ Array - enqueue()



- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over

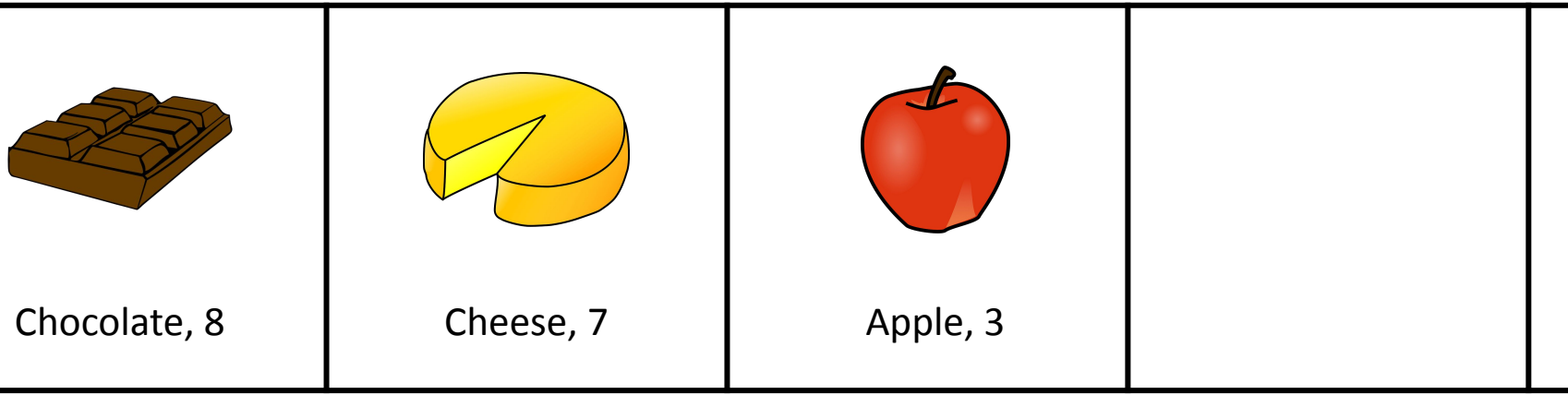


PQ Array - enqueue()

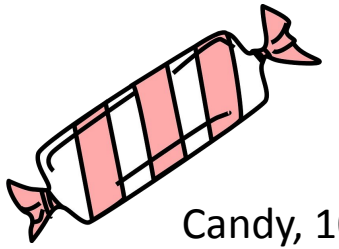


Candy, 10

- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over

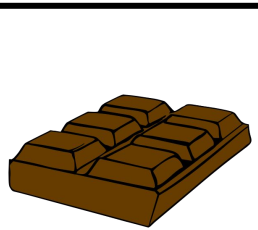


PQ Array - enqueue()

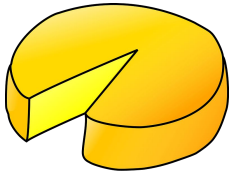


Candy, 10

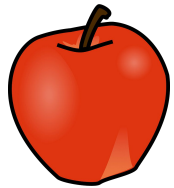
- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over



Chocolate, 8



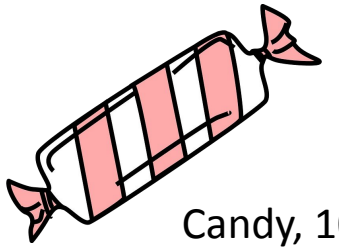
Cheese, 7



Apple, 3

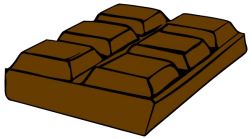
0	1	2	3

PQ Array - enqueue()

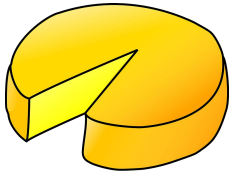


Candy, 10

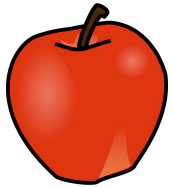
- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over



Chocolate, 8



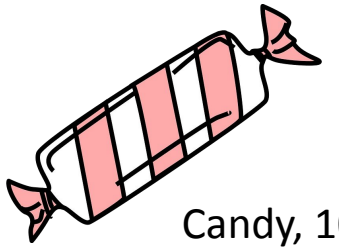
Cheese, 7



Apple, 3

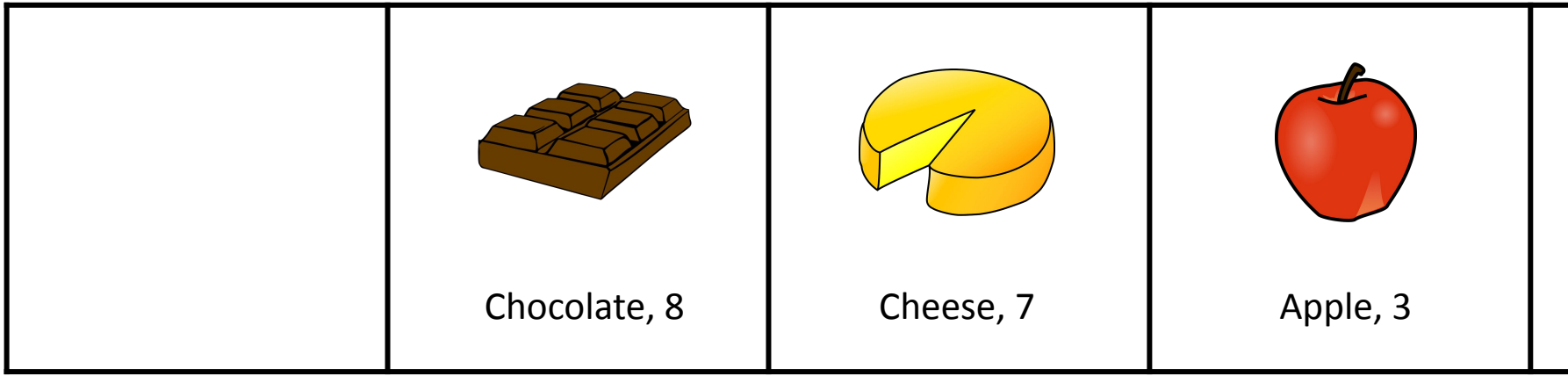
0 1 2 3

PQ Array - enqueue()



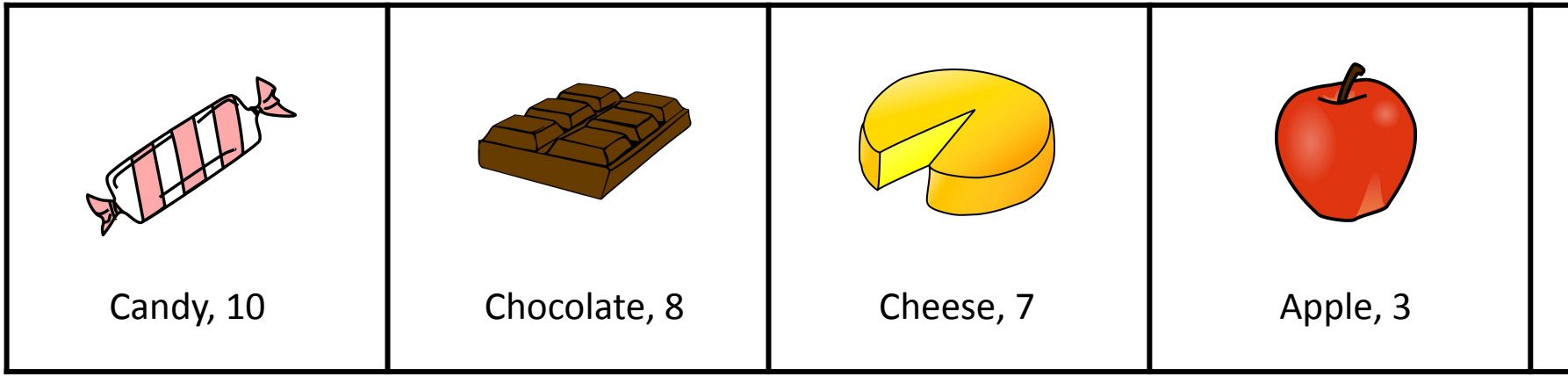
Candy, 10

- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over



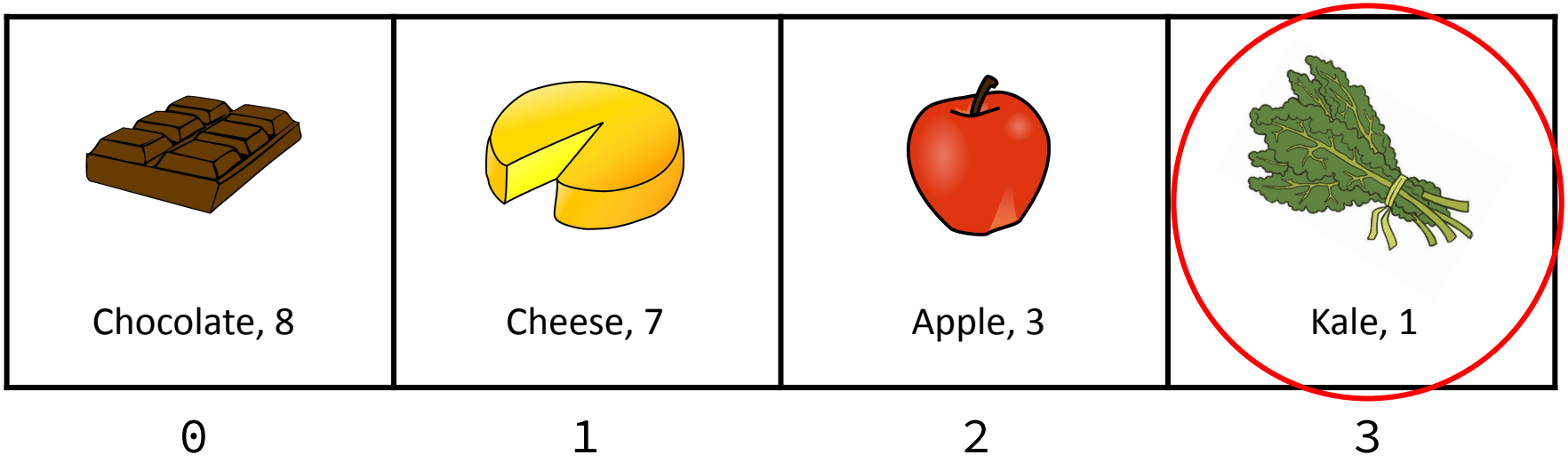
PQ Array - enqueue()

- Add the element into the array in the correct position
- This is O(n), worst case we have to shift n other elements over



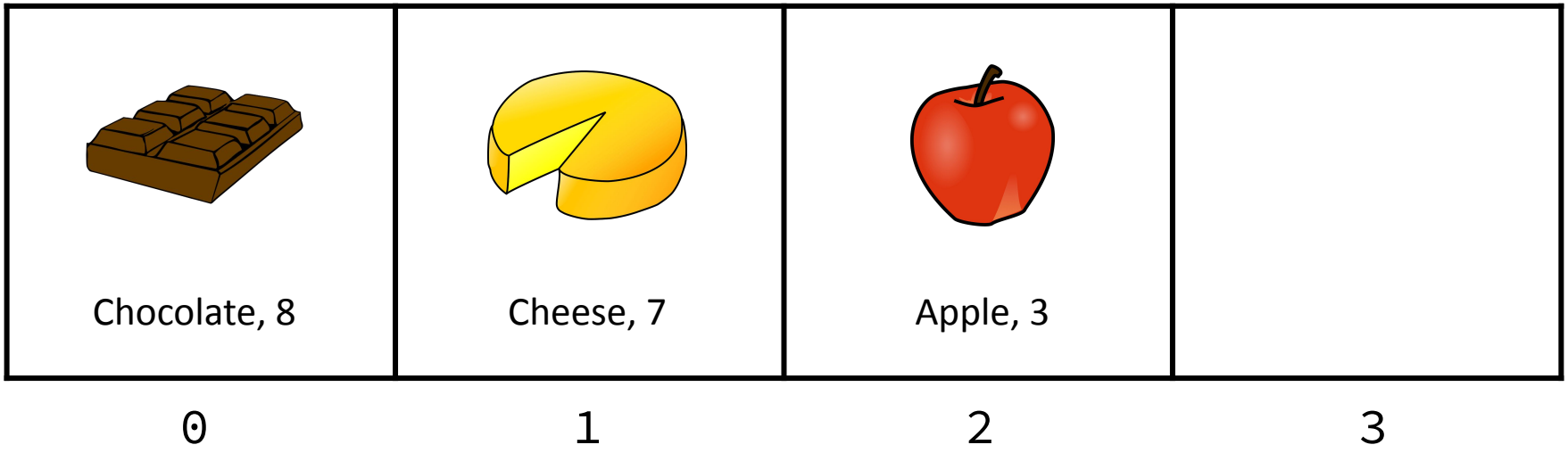
PQ Array - dequeue()

- Remove and return last (highest priority) element of the array
- This is O(1), don't need to move any other elements



PQ Array - dequeue()

- Remove and return last (highest priority) element of the array
- This is O(1), don't need to move any other elements



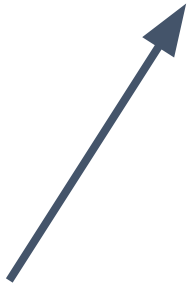
PQ Array Runtimes

- `peek() - O(1)`
- `enqueue(elem, priority) - O(n)`
- `dequeue() - O(1)`

🤔 *How would these runtimes be different if we stored the highest priority element at the beginning of our array?*

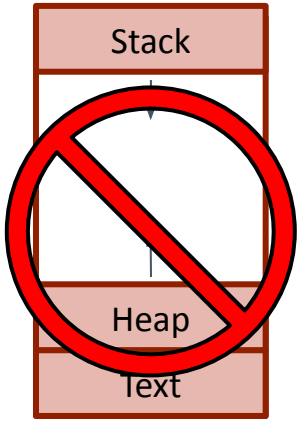
PQ Array Runtimes

- `peek() - O(1)`
- `enqueue(elem, priority) - O(n)`
- `dequeue() - O(1)`



We can do better than this…
Ideally, none of our basic operations are `O(n)`.

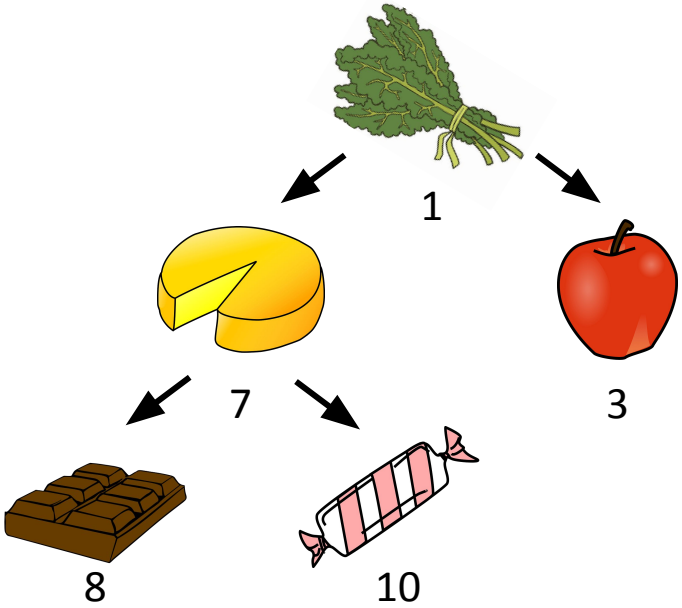
Not related to heap memory



Second Attempt: Binary Heap

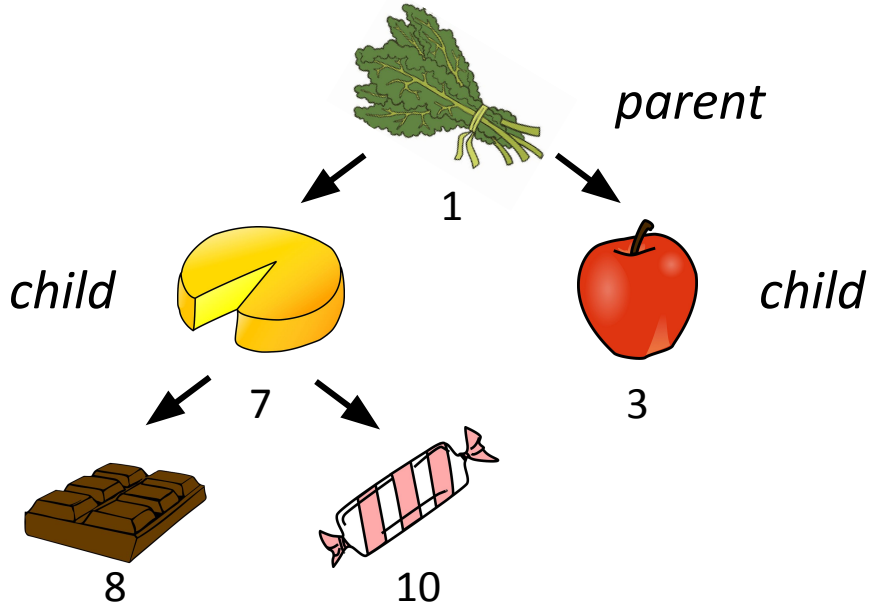
What's a Binary Heap?

- A heap is a tree-based data structure that satisfies the "heap property": parents have a higher priority than their children



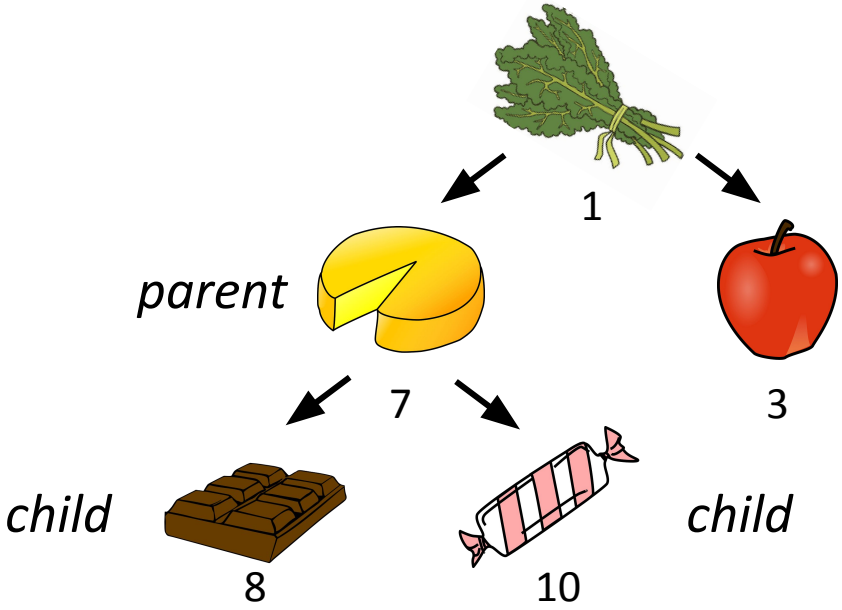
What's a Binary Heap?

- A heap is a tree-based data structure that satisfies the "heap property": parents have a higher priority than their children



What's a Binary Heap?

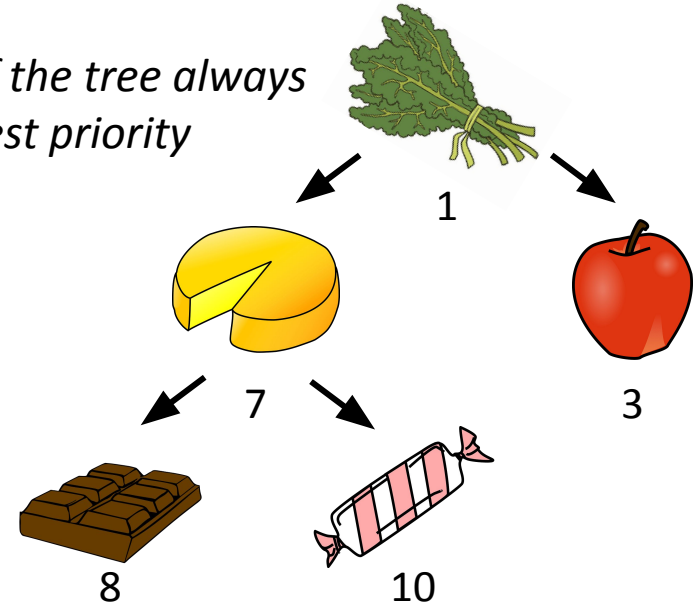
- A heap is a tree-based data structure that satisfies the "heap property": parents have a higher priority than their children



What's a Binary Heap?

- A heap is a tree-based data structure that satisfies the "heap property": parents have a higher priority than their children

The "root" of the tree always has the highest priority

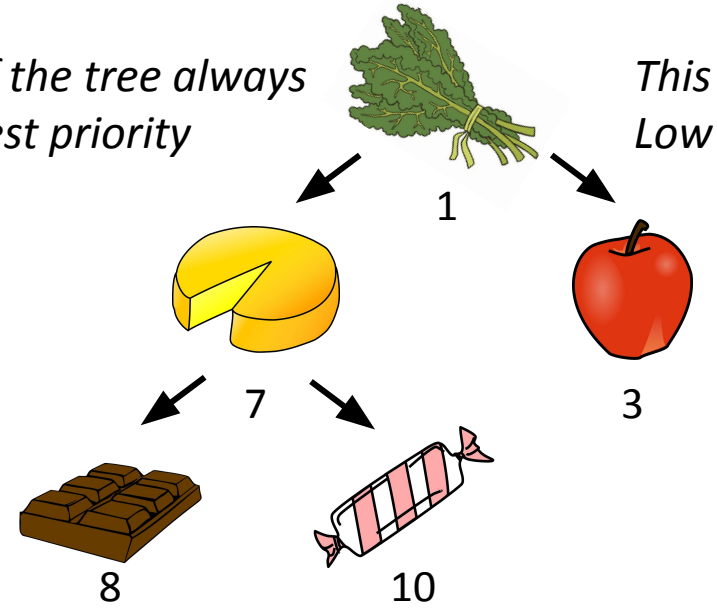


What's a Binary Heap?

- A heap is a tree-based data structure that satisfies the "heap property": parents have a higher priority than their children

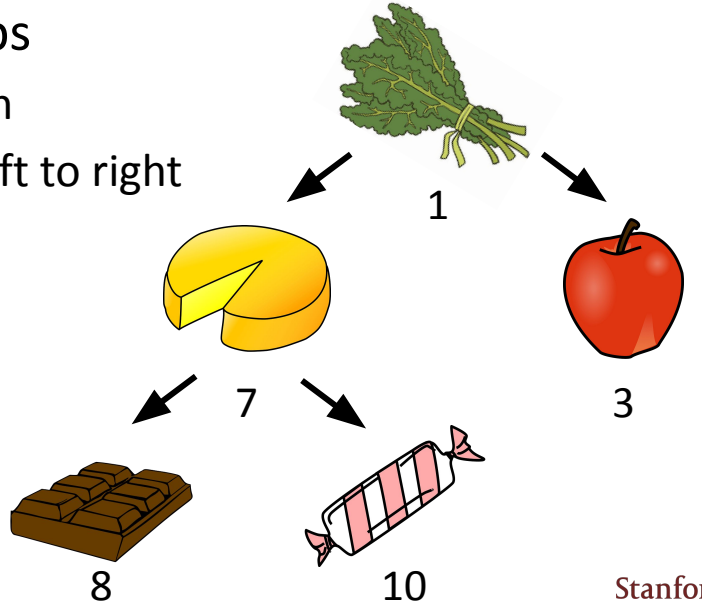
The "root" of the tree always has the highest priority

This is a min-heap: Low value = high priority



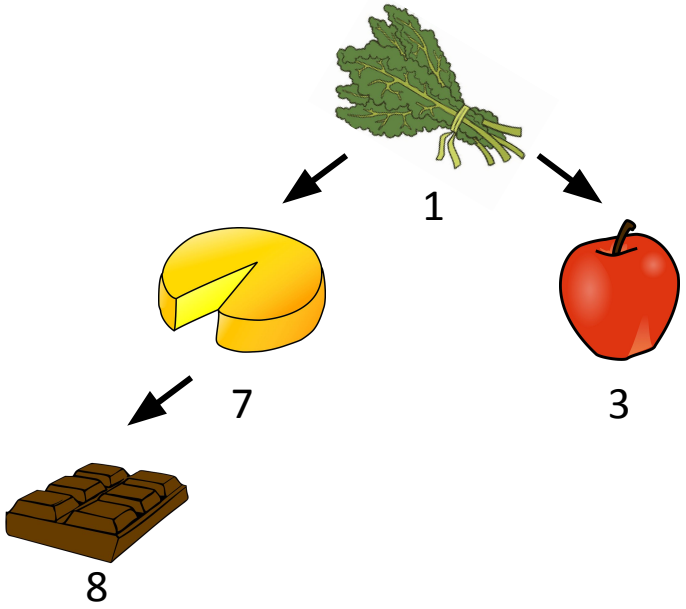
What's a Binary Heap?

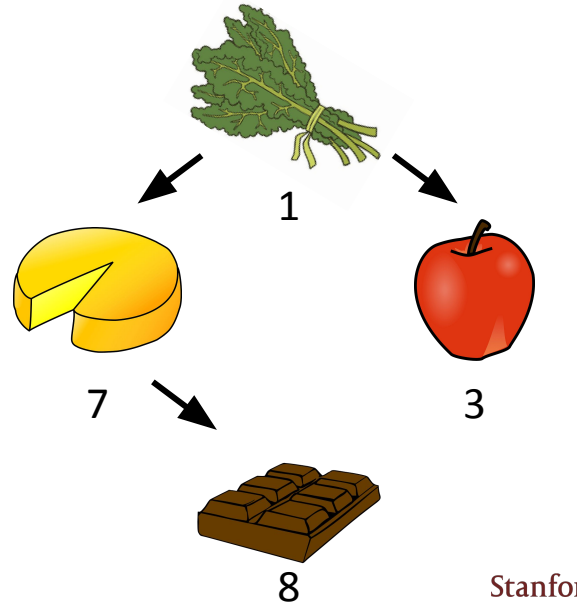
- A heap is a tree-based data structure that satisfies the "heap property": parents have a higher priority than their children
- For now, we'll focus on *binary* heaps
  - Each parent has exactly two children
  - Exception: last level, which we fill left to right



Which is a valid binary heap?

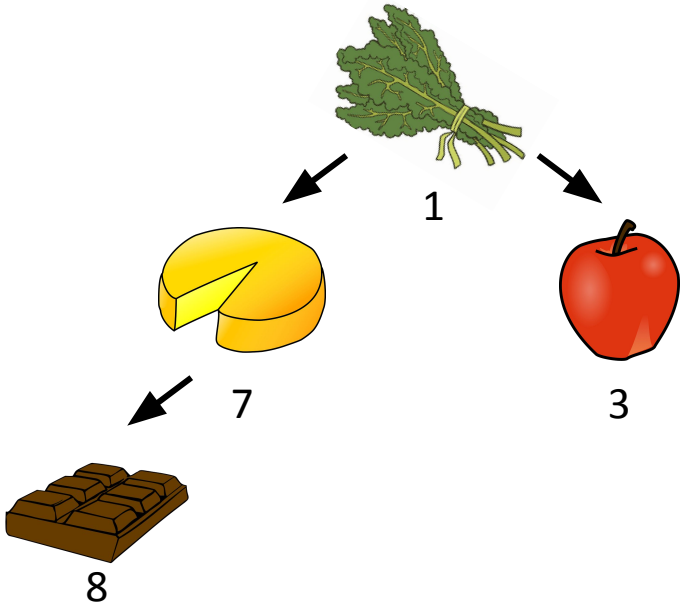
- Parent priority higher than children
- 2 children per parent except in last row, which is filled left to right



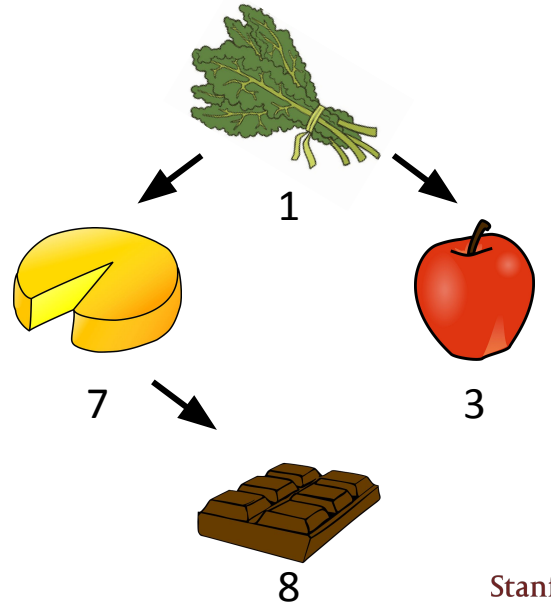


Which is a valid binary heap?

YOU'RE VALID 😎

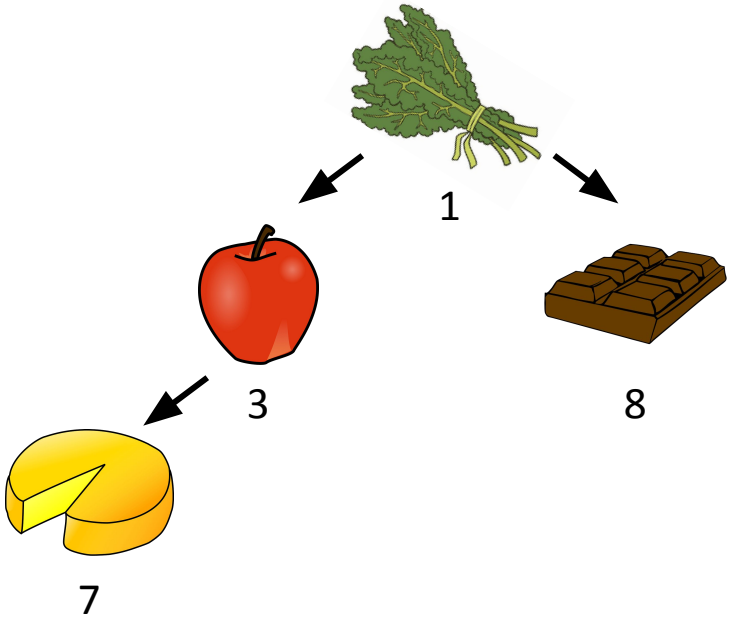


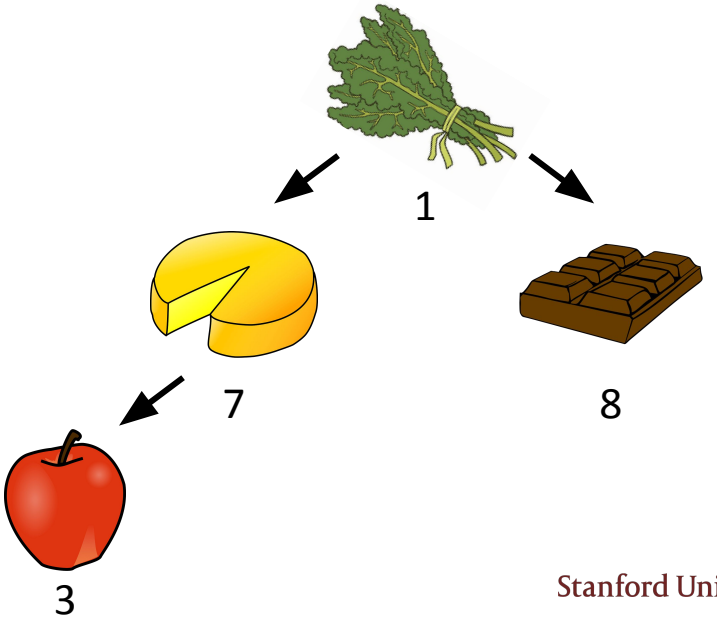
This heap's bottom row isn't filled left to right



Which is a valid binary heap?

- Parent priority higher than children
- 2 children per parent except in last row, which is filled left to right

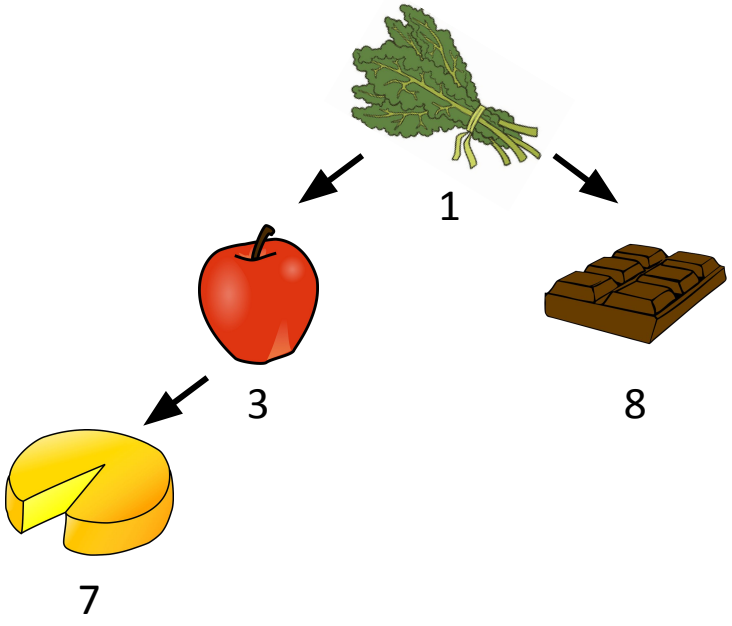


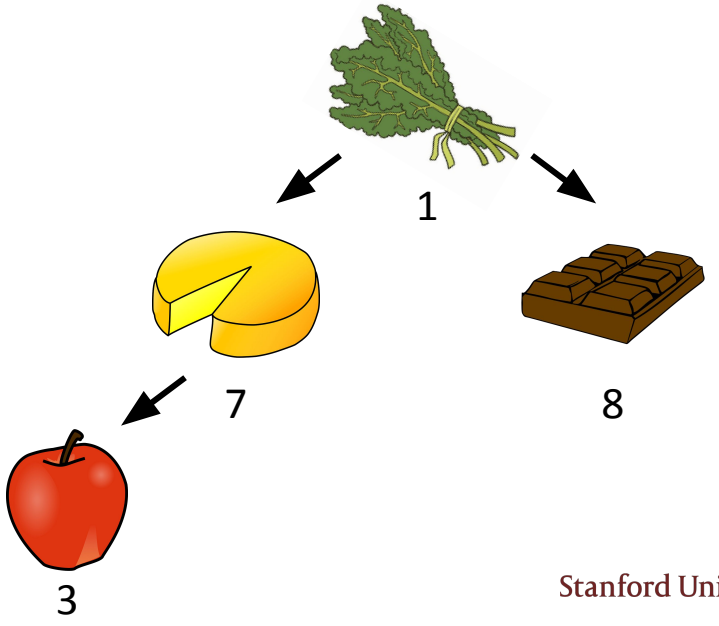


Which is a valid binary heap?

YOU'RE VALID 😎

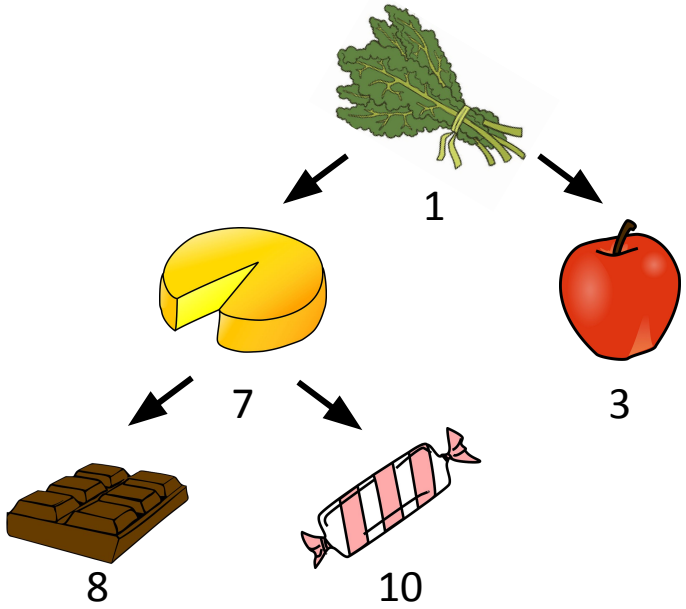
A parent (cheese) has a lower priority than its child (apple)





Implementing a Heap

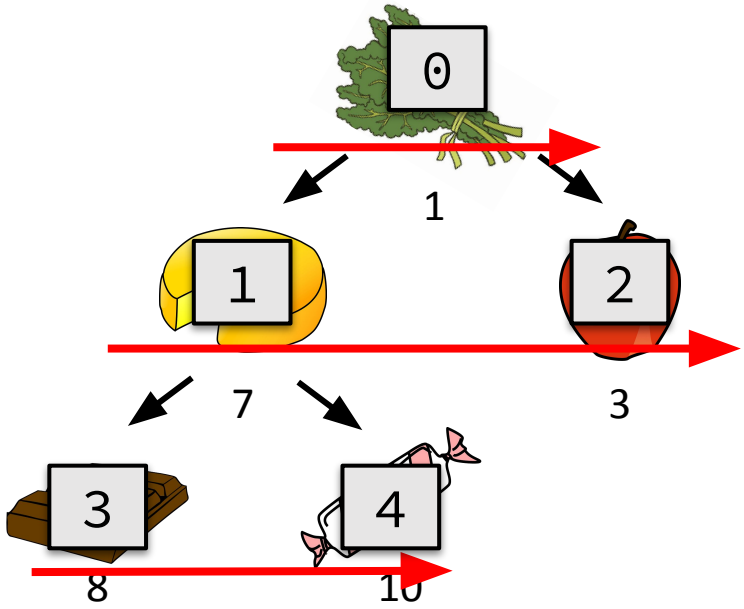
- We could store this tree's data in an array, filling in each element from top to bottom, left to right



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

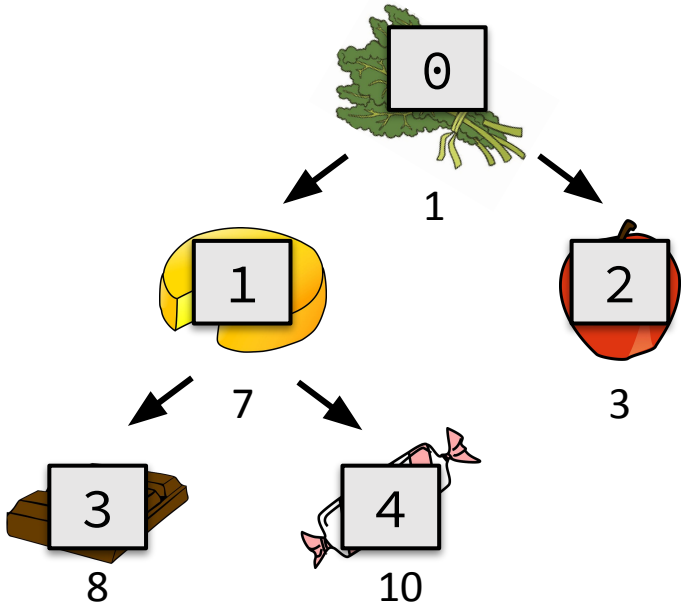
- We could store this tree's data in an array, filling in each element from top to bottom, left to right



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right



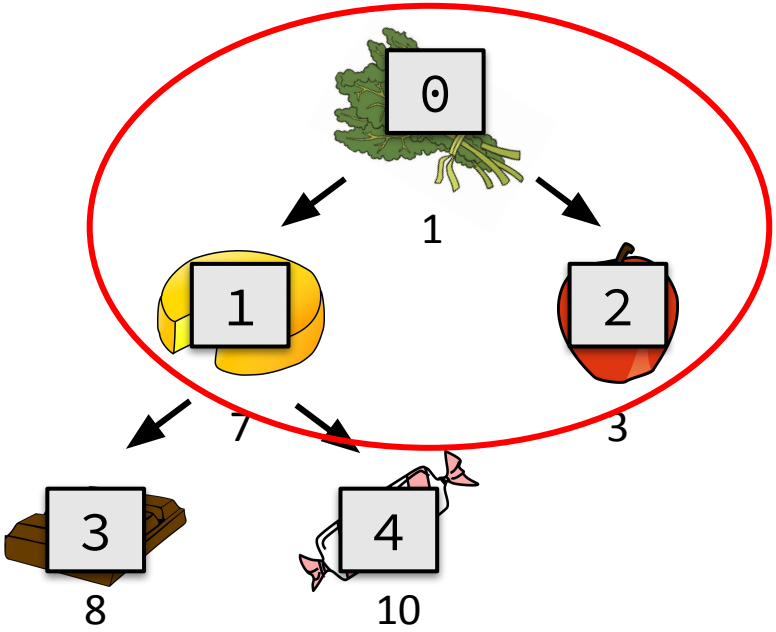
👥 *How are parents and children related in the array?*
Hint: take a look at their indices…

{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right

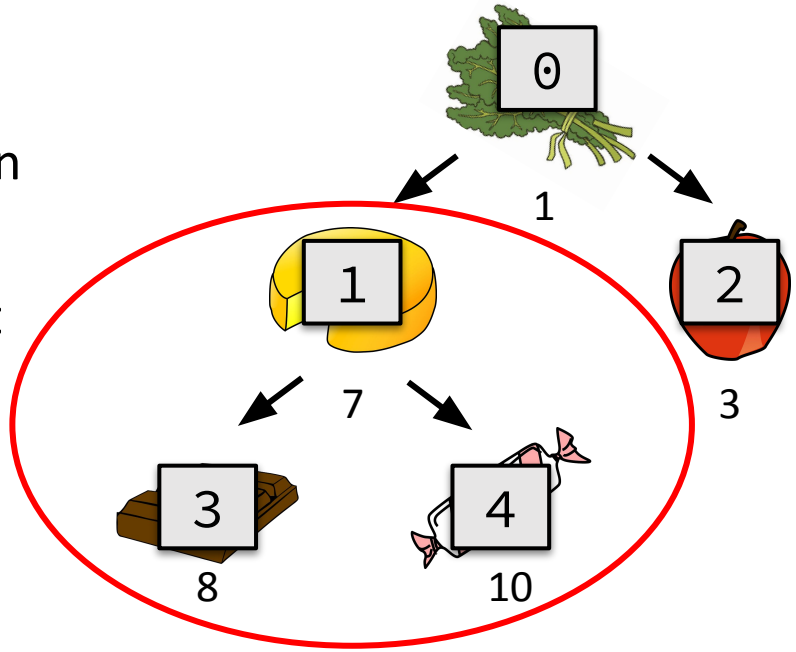
Parent index: 0
Left child index: 1
Right child index: 2



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right



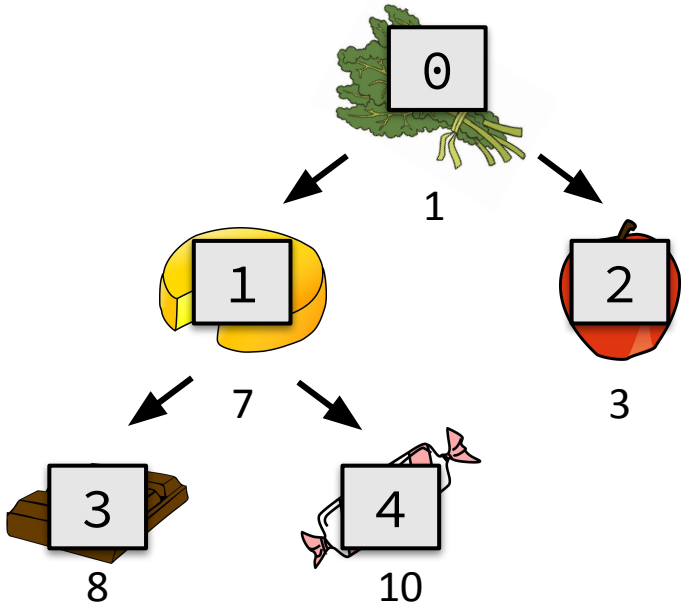
Parent index: 0
Left child index: 1
Right child index: 2

Parent index: 1
Left child index: 3
Right child index: 4

{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right



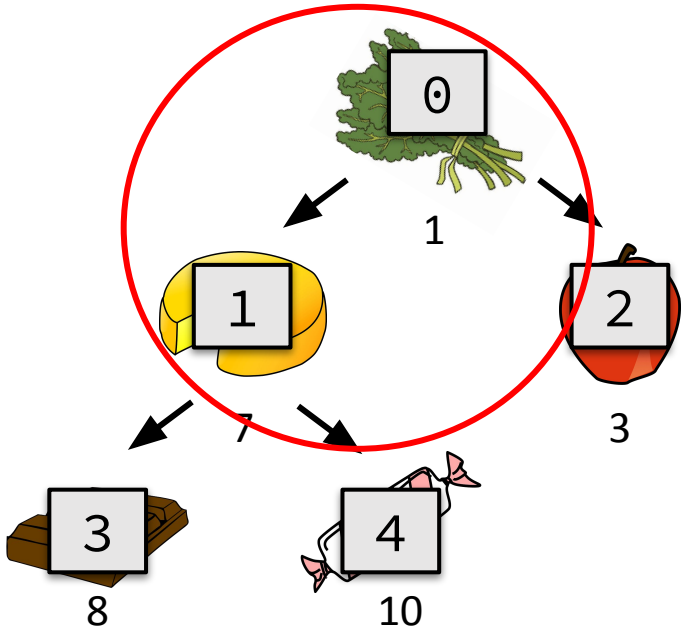
Formula: if parent is at index `i`:
Left child is at `2 * i + 1`
Right child is at `2 * i + 2`

{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right

Child index: 1
Parent index: 0



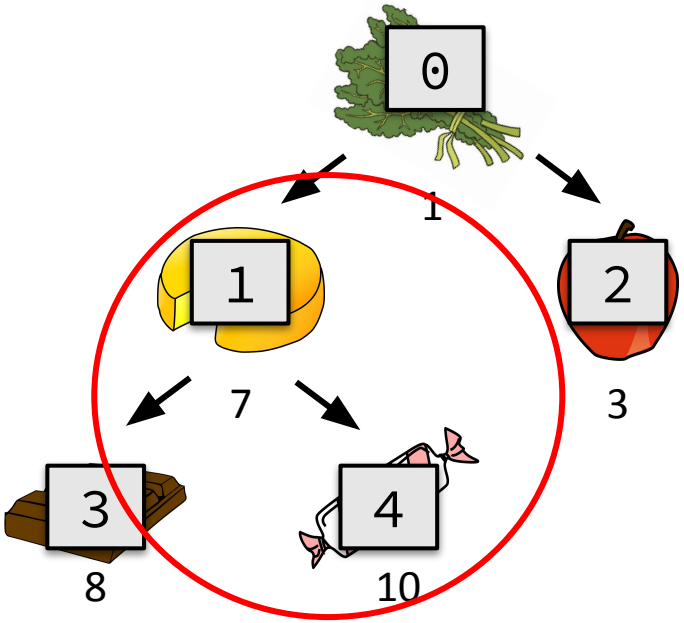
{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right

Child index: 1
Parent index: 0

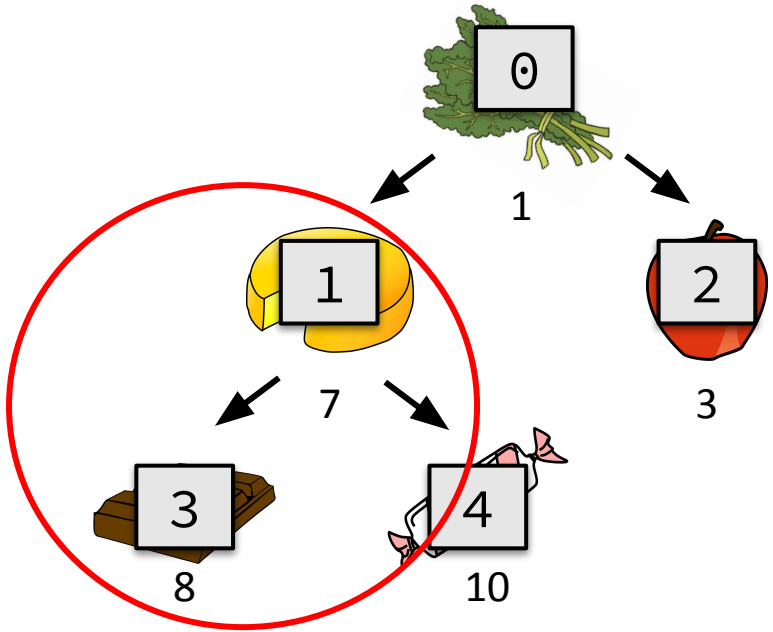
Child index: 4
Parent index: 1



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right



Child index: 1
Parent index: 0

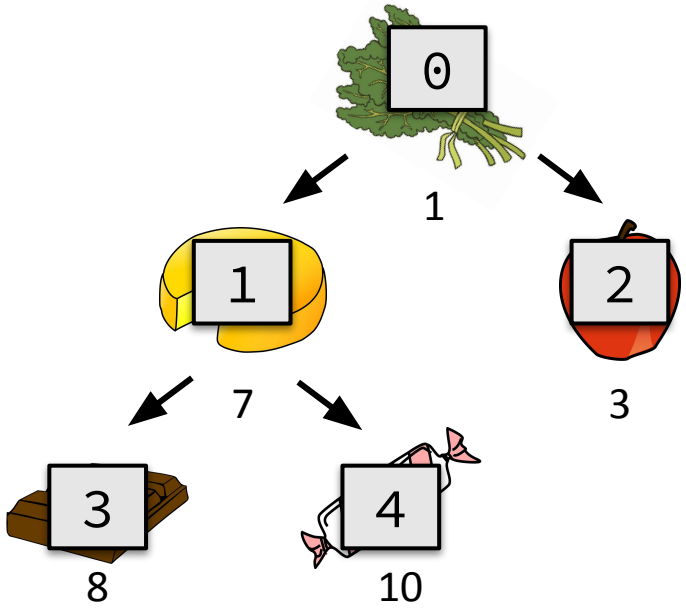
Child index: 4
Parent index: 1

Child index: 3
Parent index: 1

{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right

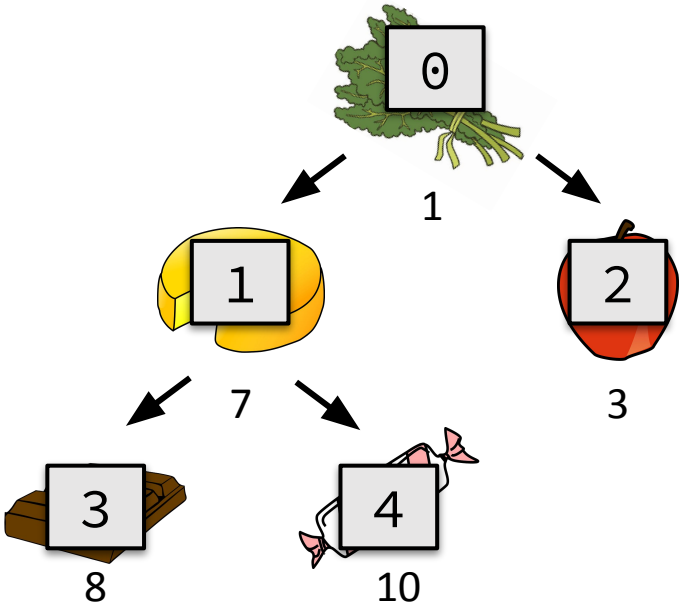


Formula: if child is at index `i`:
Parent is at `(i - 1) / 2`

{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Implementing a Heap

- We could store this tree's data in an array, filling in each element from top to bottom, left to right



Formula: if child is at index `i`:
Parent is at `(i - 1) / 2`

Int division rounds down!

{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

Your Turn!

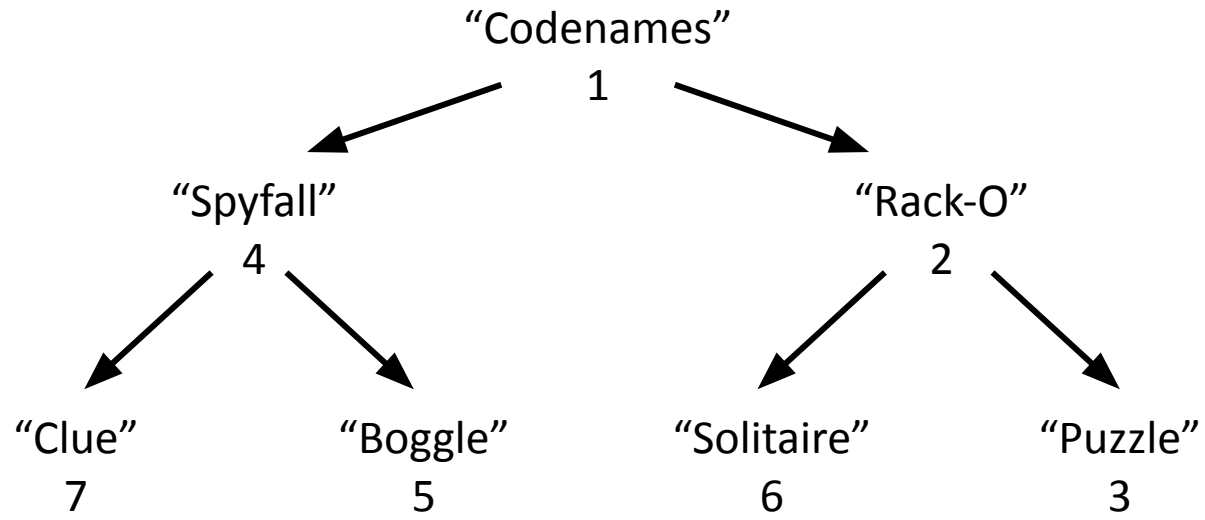
Formula: if parent is at index `i`:
Left child is at `2 * i + 1`
Right child is at `2 * i + 2`

Formula: if child is at index `i`:
Parent is at `(i - 1) / 2`

Given this array (representing a heap):

1. What is the parent of “Clue”?
2. What is the left child of “Rack-O”?
3. Draw the tree corresponding to this heap to confirm your answers!

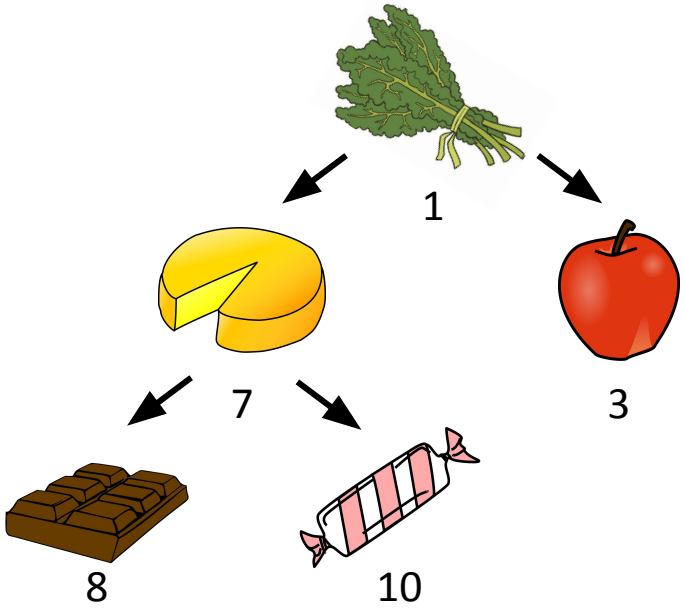
{“Codenames”, 1}	{“Spyfall”, 4}	{“Rack-O”, 2}	{“Clue”, 7}	{“Boggle”, 5}	{“Solitaire”, 6}	{“Puzzle”, 3}
0	1	2	3	4	5	6



{"Codenames", 1}	{"Spyfall", 4}	{"Rack-O", 2}	{"Clue", 7}	{"Boggle", 5}	{"Solitaire", 6}	{"Puzzle", 3}
0	1	2	3	4	5	6

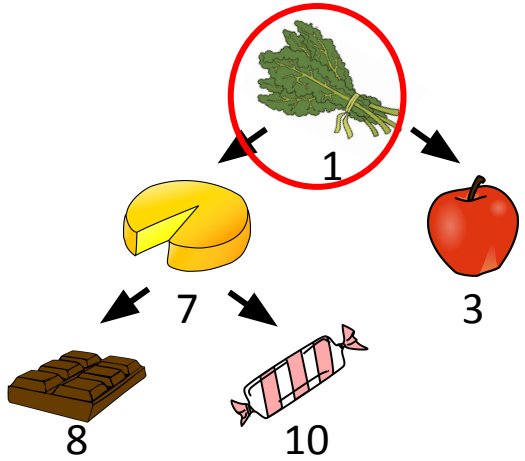
PQ Heap

How might a binary heap help us implement a PQ?
How would peek, enqueue, and dequeue work?



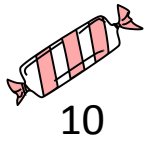
PQ Heap - peek()

- Return the highest priority element, without removing it
- This is O(1), we just check what's at the first index of our array

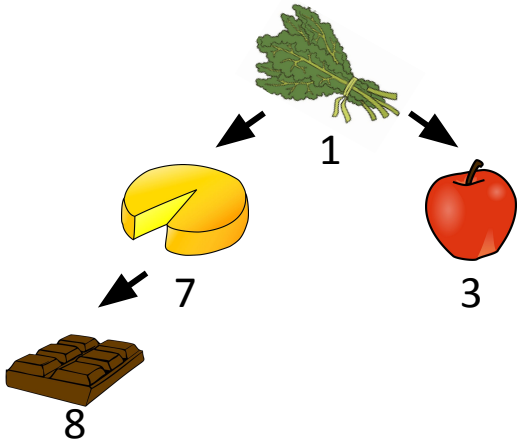


{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}
0	1	2	3

PQ Heap - enqueue()



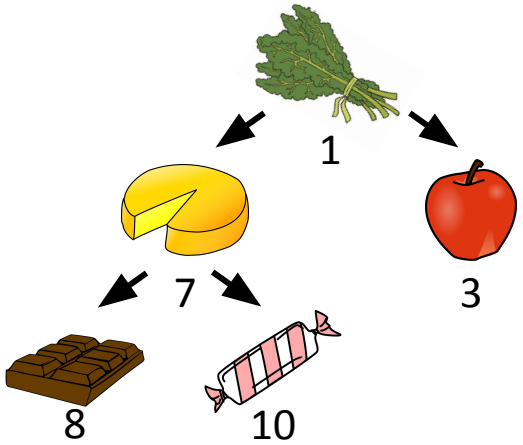
- Add the element into the array in the correct position
- Here's an easy case…



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}
0	1	2	3

PQ Heap - enqueue()

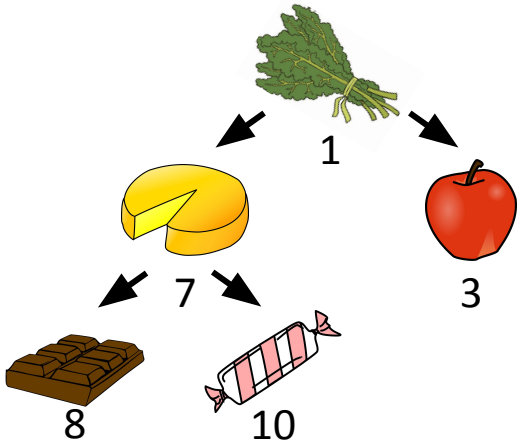
- Add the element into the array in the correct position
- Here's an easy case…



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}	{"candy", 10}
0	1	2	3	4

PQ Heap - enqueue()

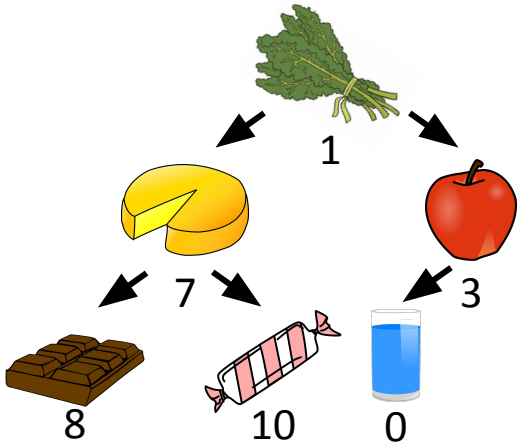
- Add the element into the array in the correct position
- What about this?



{“kale”, 1}	{“cheese”, 7}	{“apple”, 3}	{“cocoa”, 8}	{“candy”, 10}
0	1	2	3	4

PQ Heap - enqueue()

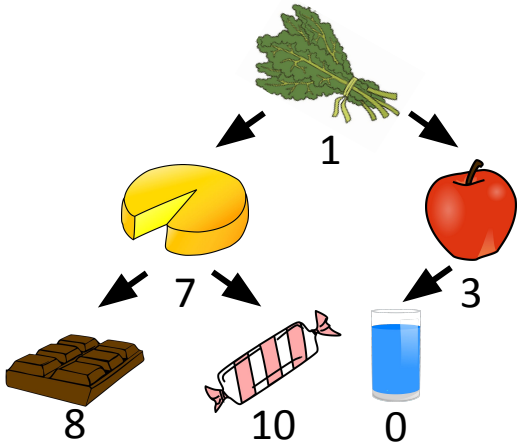
- Add the element into the array in the correct position
- What about this?



{“kale”, 1}	{“cheese”, 7}	{“apple”, 3}	{“cocoa”, 8}	{“candy”, 10}	{“water”, 0}
0	1	2	3	4	5

PQ Heap - enqueue()

- Add the element into the array in the correct position
- What about this?



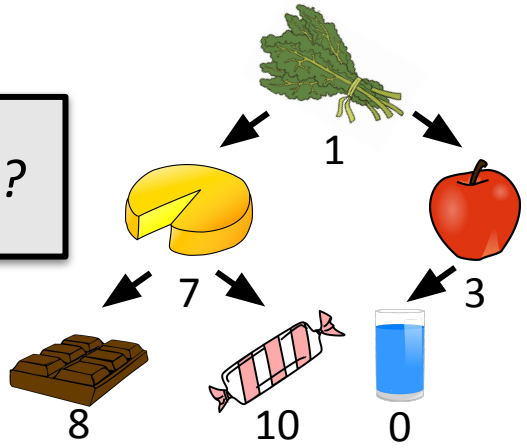
Now the heap property is violated!

{“kale”, 1}	{“cheese”, 7}	{“apple”, 3}	{“cocoa”, 8}	{“candy”, 10}	{“water”, 0}
0	1	2	3	4	5

PQ Heap - enqueue()

- Add the element into the array in the correct position
- What about this?

🤔 *How might we fix this?*

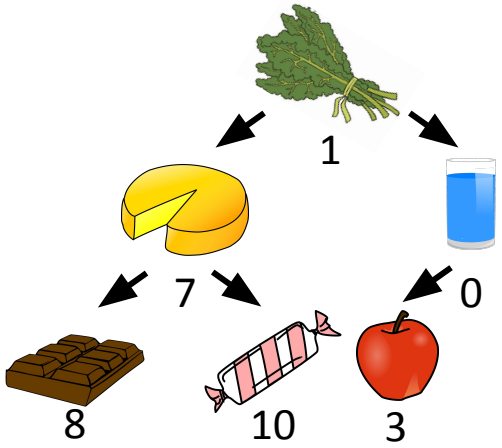


Now the heap property is violated!

{“kale”, 1}	{“cheese”, 7}	{“apple”, 3}	{“cocoa”, 8}	{“candy”, 10}	{“water”, 0}
0	1	2	3	4	5

PQ Heap - enqueue()

- Add the element into the array in the correct position
- What about this?

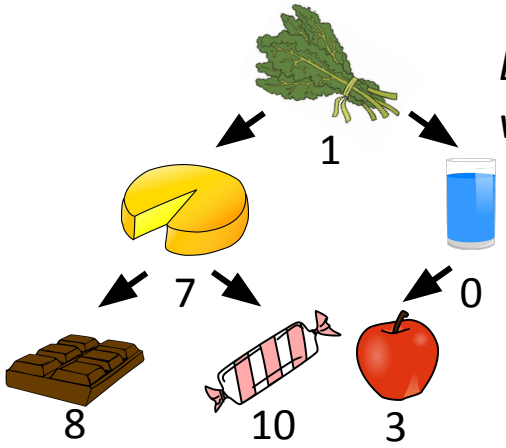


We could swap "apple" and "water"...

{"kale", 1}	{"cheese", 7}	{"water", 0}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

PQ Heap - enqueue()

- Add the element into the array in the correct position
- What about this?

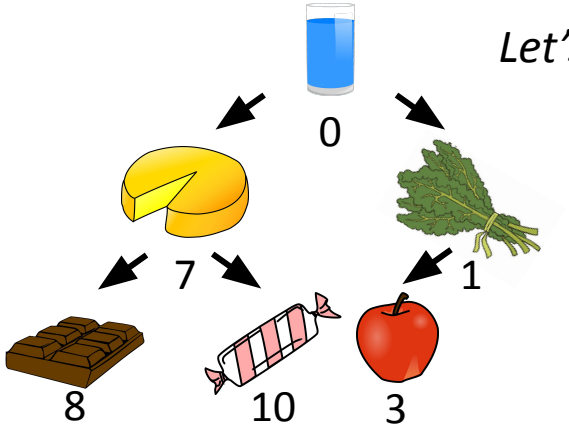


But we run into another violation.

{“kale”, 1}	{“cheese”, 7}	{“water”, 0}	{“cocoa”, 8}	{“candy”, 10}	{“apple”, 3}
0	1	2	3	4	5

PQ Heap - enqueue()

- Add the element into the array in the correct position
- What about this?

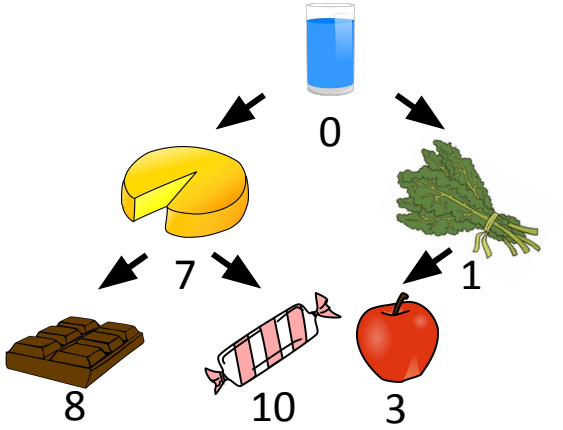


Let's swap again!

{"water", 0}	{"cheese", 7}	{"kale", 1}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

PQ Heap - enqueue()

- Add the element into the array in the correct position
- What about this?



Much better :)
This process was called "bubbling up"!

{"water", 0}	{"cheese", 7}	{"kale", 1}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

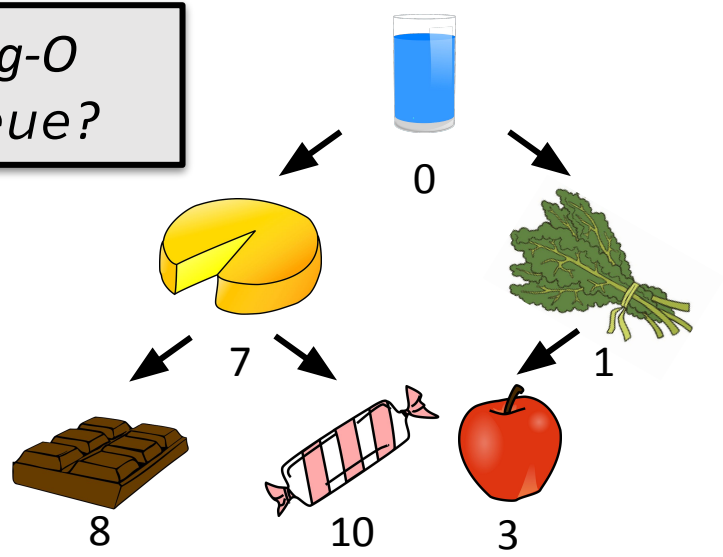
PQ Heap - enqueue()

To enqueue a new element into our PQ Heap, we “bubble up”:

1. Insert element at the end of array
2. If this element has a greater priority than its parent, swap parent and child element
3. Repeat 2 until heap property is satisfied or we reach the root!

PQ Heap - enqueue()

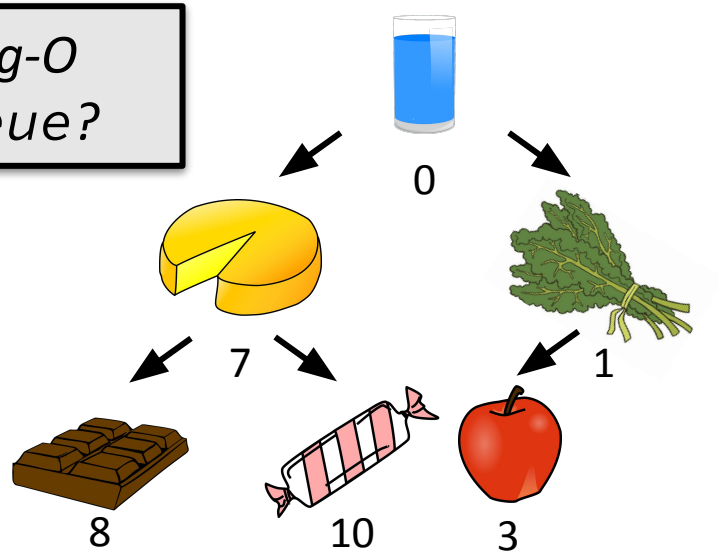
What's the Big-O runtime of enqueue?



{"water", 0}	{"cheese", 7}	{"kale", 1}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

PQ Heap - enqueue()

What's the Big-O runtime of enqueue?

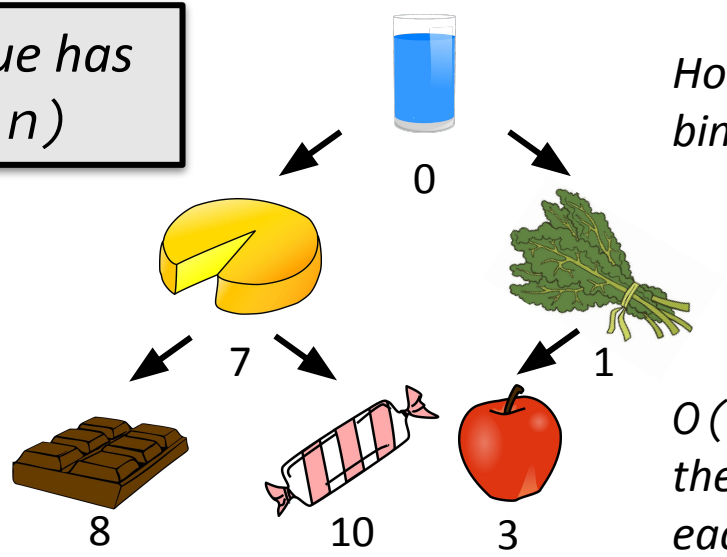


Worst case, we bubble up from the bottom to the top of the tree

{"water", 0}	{"cheese", 7}	{"kale", 1}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

PQ Heap - enqueue()

PQ Heap enqueue has runtime `O(log n)`



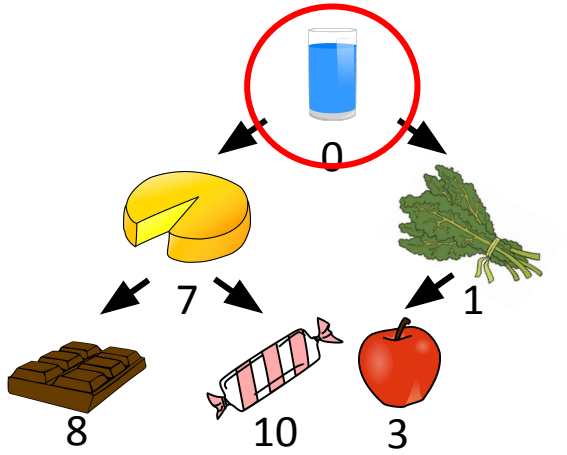
How many levels are in a binary heap with n elements?

`O(log n)` - we're doubling the number of elements at each level

{"water", 0}	{"cheese", 7}	{"kale", 1}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

PQ Heap - dequeue()

- Remove and return first (highest priority) element of the array

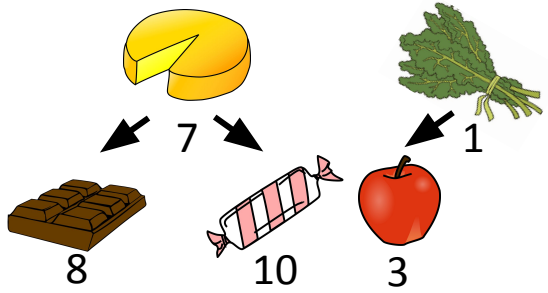


{“water”, 0}	{“cheese”, 7}	{“kale”, 1}	{“cocoa”, 8}	{“candy”, 10}	{“apple”, 3}
0	1	2	3	4	5

PQ Heap - dequeue()

- Remove and return first (highest priority) element of the array

Now what?

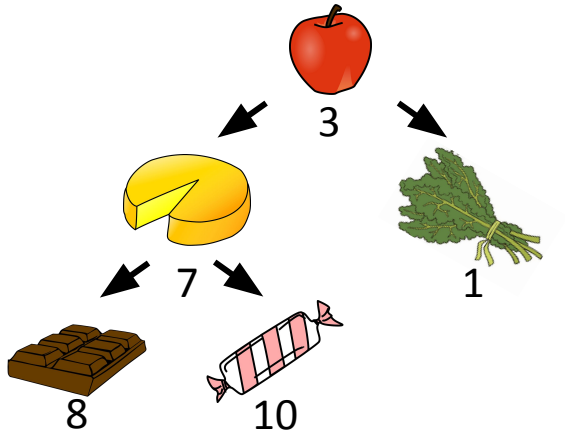


	{“cheese”, 7}	{“kale”, 1}	{“cocoa”, 8}	{“candy”, 10}	{“apple”, 3}
0	1	2	3	4	5

PQ Heap - dequeue()

- Remove and return first (highest priority) element of the array

Move last element to first position in the array

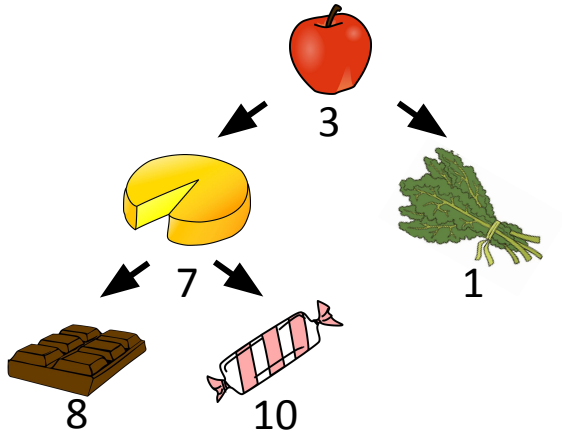


{“apple”, 3}	{“cheese”, 7}	{“kale”, 1}	{“cocoa”, 8}	{“candy”, 10}	
0	1	2	3	4	5

PQ Heap - dequeue()

- Remove and return first (highest priority) element of the array

Bubble down! Swap with higher priority child until heap property is satisfied.

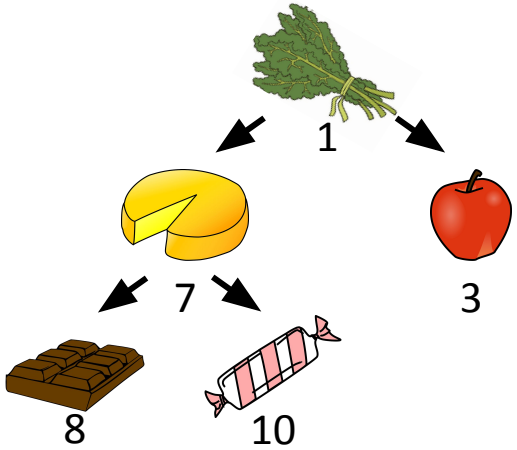


{“apple”, 3}	{“cheese”, 7}	{“kale”, 1}	{“cocoa”, 8}	{“candy”, 10}	
0	1	2	3	4	5

PQ Heap - dequeue()

- Remove and return first (highest priority) element of the array

Bubble down! Swap with smaller child until heap property is satisfied.



{“kale”, 1}	{“cheese”, 7}	{“apple”, 3}	{“cocoa”, 8}	{“candy”, 10}	
0	1	2	3	4	5

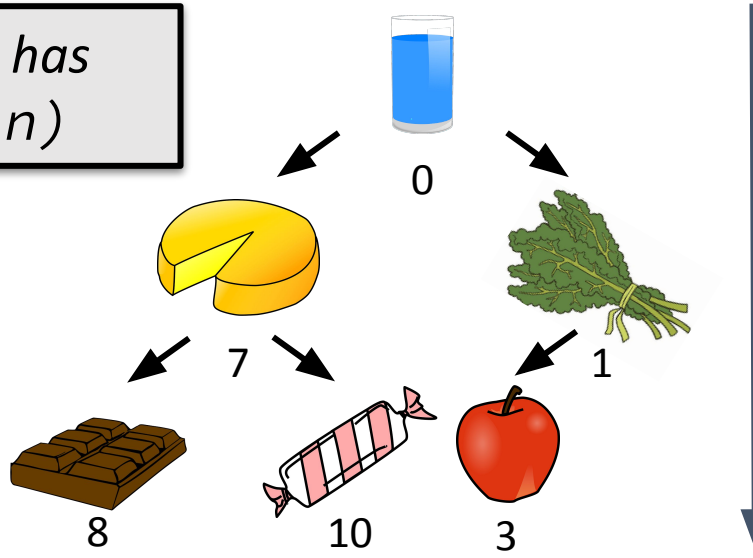
PQ Heap - dequeue()

To dequeue the highest priority element in our PQ Heap:

1. Remove element from the beginning (index 0) of our array
2. Move last element in array to index 0
3. Swap with higher priority child until heap property is satisfied

PQ Heap - dequeue()

PQ Heap dequeue has runtime O(log n)



Worst case, we bubble down from the top to the bottom of the tree

{“water”, 0}	{“cheese”, 7}	{“kale”, 1}	{“cocoa”, 8}	{“candy”, 10}	{“apple”, 3}
0	1	2	3	4	5

PQ Heap Runtimes

- `peek() - O(1)`
- `enqueue(elem, priority) - O(log n)`
- `dequeue() - O(log n)`

Notice how implementing the same data structure with a heap versus sorted array leads to different runtimes.

Stay tuned for Assignment 4!

Recap

- What do priority queues (PQs) represent?
- Implementing PQs with a sorted array
- Implementing PQs with a binary heap
  - Enqueue and bubble up
  - Dequeue and bubble down

Check out this visualization of min heap operations!

Thank you!