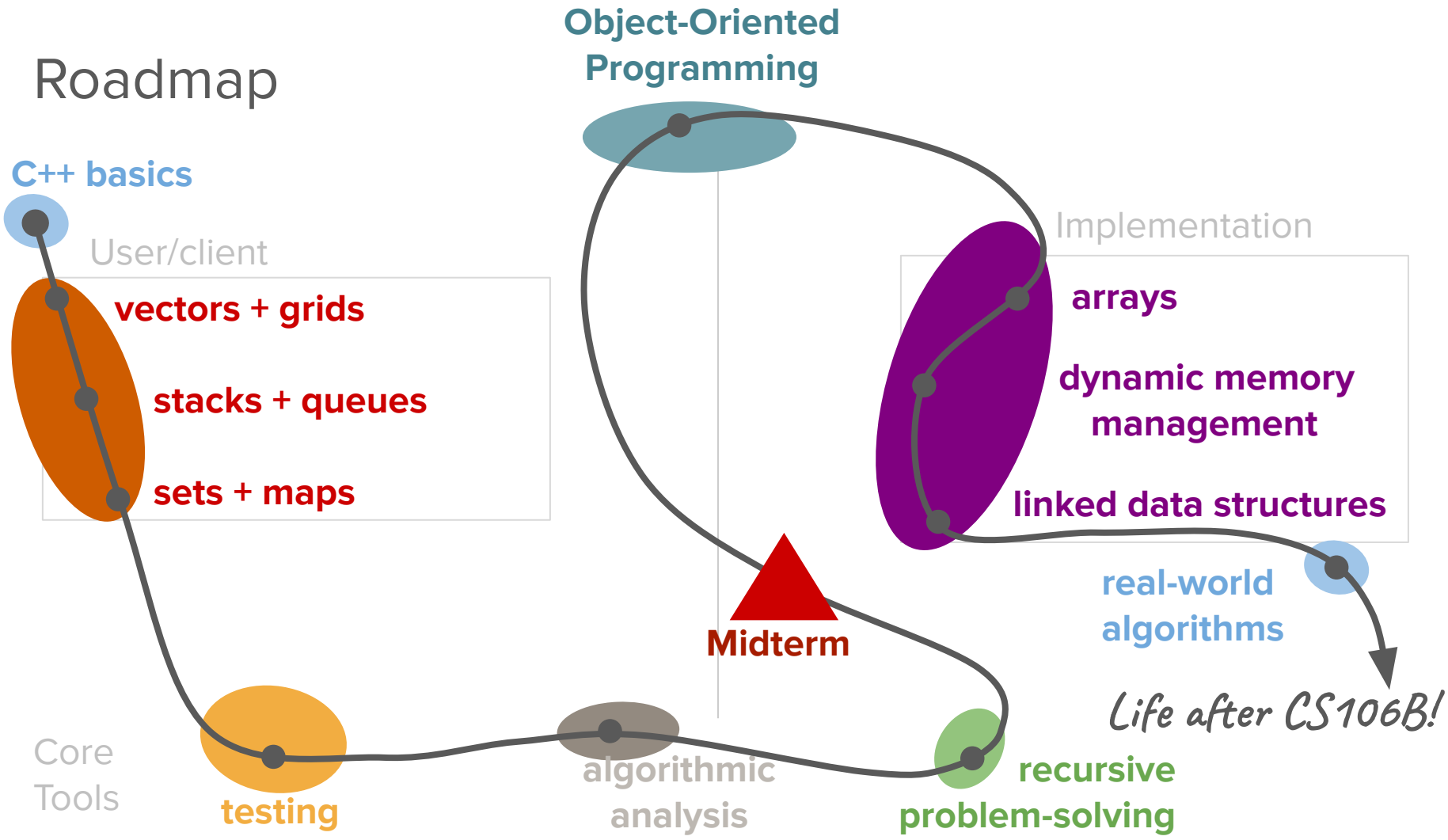
Priority Queues and Heaps

What is an example of a real-world system where you need to put people in a ranked or prioritized order?

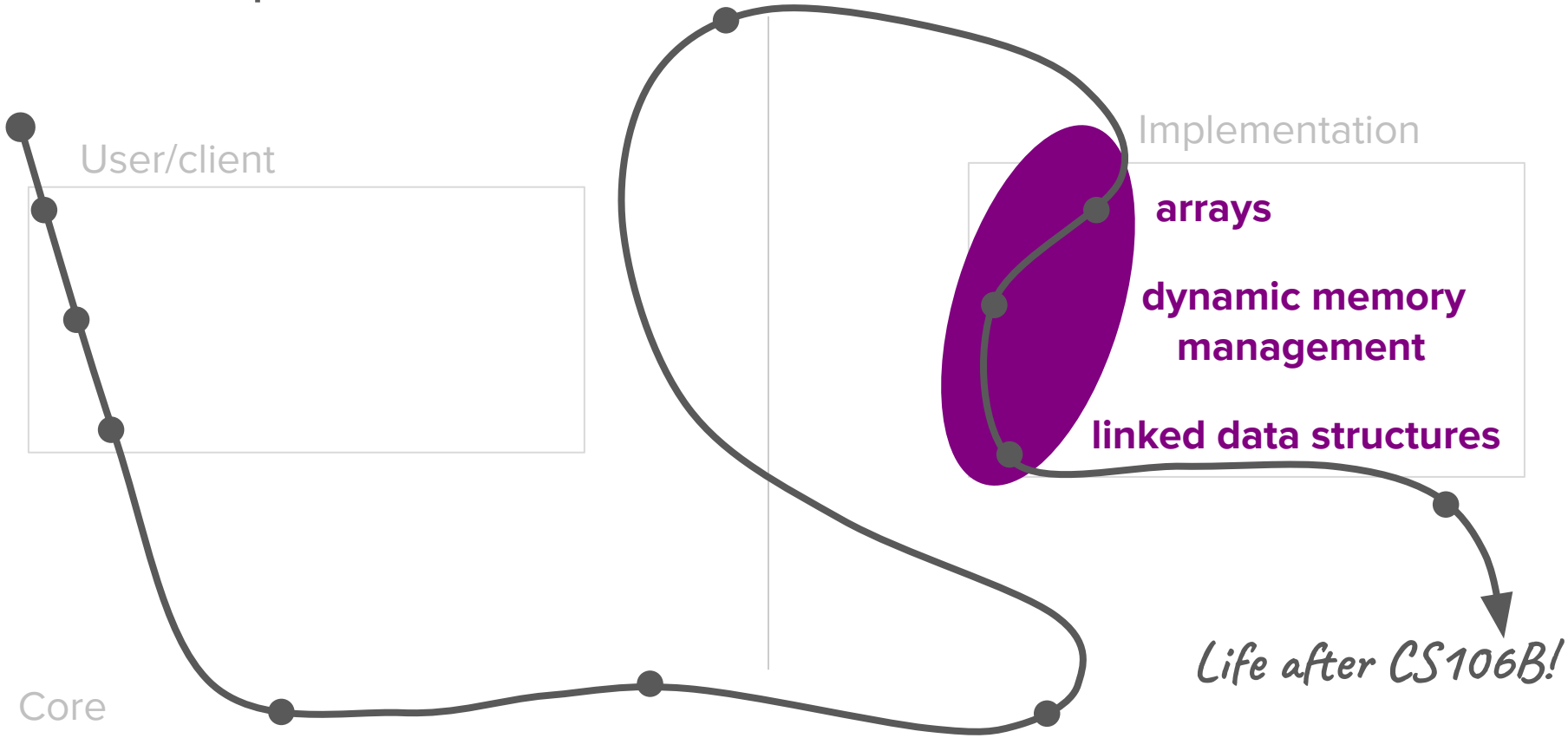
pollev.com/cs106bpolls

What is an example of a real-world system where you need to put people in a ranked or prioritized order?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Midterm
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


User/client
Implementation
arrays
dynamic memory management
linked data structures
Life after CS106B!
Core
Tools

Today's questions

How can we make use of multiple levels of abstraction to build better ADTs?

How do we implement algorithms for prioritizing data?

Today's topics

1. Review (`OurVector`)
2. Priority Queues
3. Binary Heaps
4. Human Prioritization Algorithms

Review

[implementing `OurVector`]

What is `OurVector`?

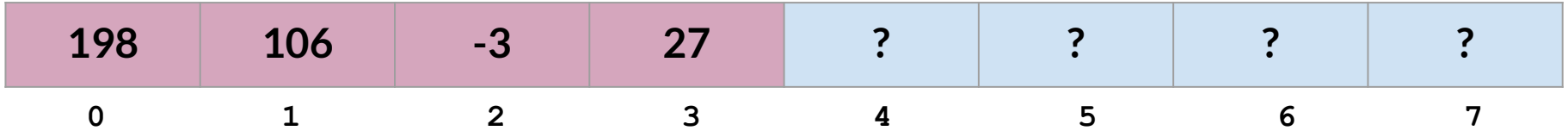
- Goal: Implement own version of the Stanford C++ Vector

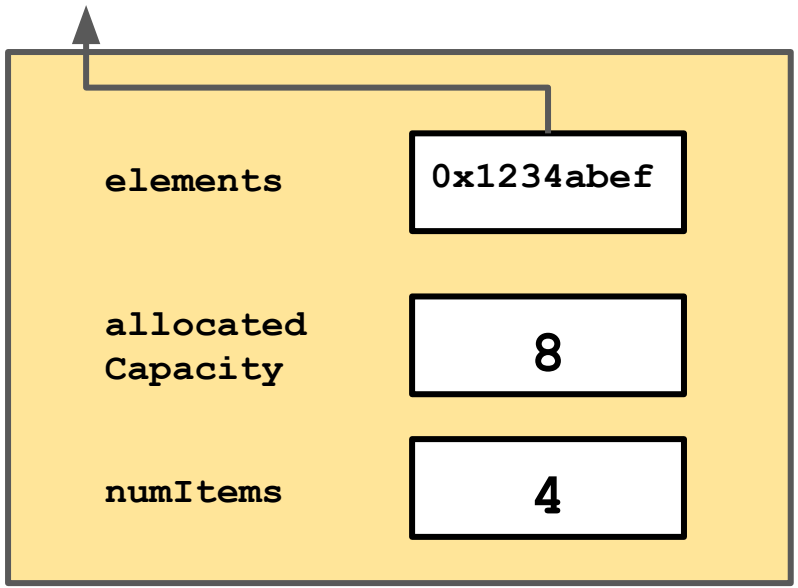
- Scope Constraints:
  - We will only implement a subset of the functionality that the Stanford Vector provides.
  - `OurVector` can **only store integers** and is not be configurable to store other types.

`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;
};
```

Review: `OurVector` internal state






```
// client code

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

vec.remove(1);
vec.insert(0, 198);
```

Dynamic Array Growth

1. Find another, larger shell.
2. Move all their stuff into the new shell.
3. Leave the old shell on the seafloor.
4. Update their address with the Hermit Crab Postal Service.
5. Make note of their new shell's spacious capacity by posting on Hermit Crab Instagram.

Dynamic Array Growth

```
77  void OurVector::expand() {
78      // 1. Create a new, larger array. Usually we choose to double the current size.
79      int* newElements = new int[2 * allocatedCapacity];
80      // 2. Copy the old array elements to the new array.
81      for (int i = 0; i < numItems; i++) {
82          newElements[i] = elements[i];
83      }
84      // 3. Delete (free) the old array.
85      delete[] elements;
86      // 4. Point the old array variable to the new array.
87      elements = newElements;
88      // 5. Update the associated capacity variable for the array.
89      allocatedCapacity *= 2;
90  }
```

Implementing ADT Classes

- The first step of implementing an ADT class (as with any class) is answering the three important questions regarding its public interface, private member variables, and initialization procedures.

- Most ADT classes will need to store their data in an underlying array. The organizational patterns of data in that array may vary, so it is important to illustrate and visualize the contents and any operations that may be done.

- The paradigm of "growable" arrays allows for fast and flexible containers with dynamic resizing capabilities that enable storage of large amounts of data.

Implementing ADT Classes

What about more complex ADTs?

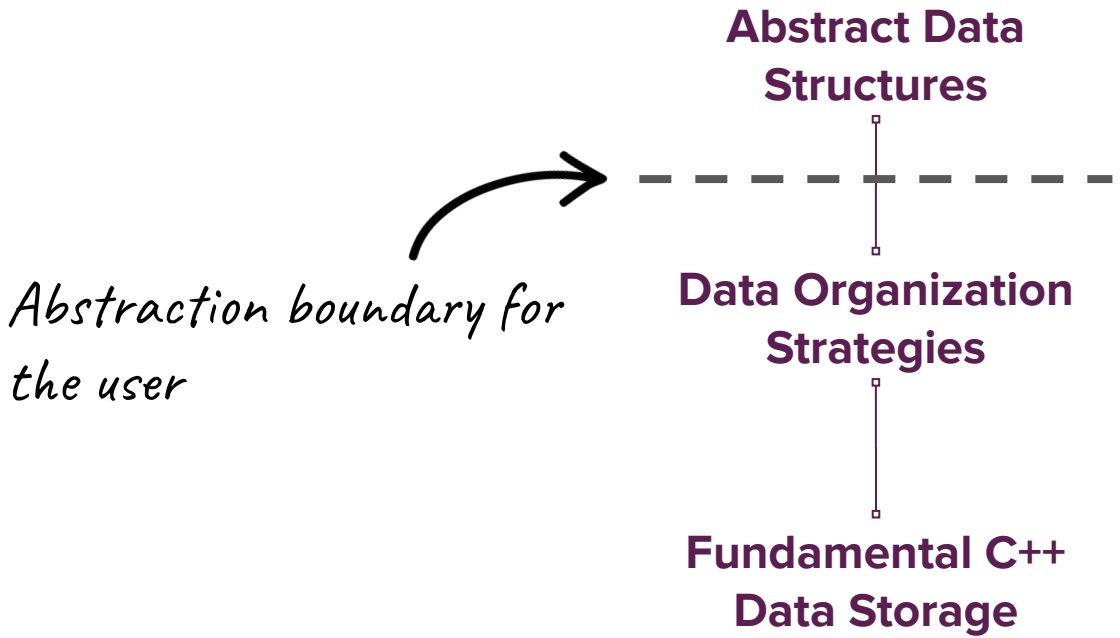
- The first step of implementing an ADT class (as with any class) is answering the three important questions regarding its public interface, private member variables, and initialization procedures.

- Most ADT classes will need to store their data in an underlying array. The organizational patterns of data in that array may vary, so it is important to illustrate and visualize the contents and any operations that may be done.

- The paradigm of "growable" arrays allows for fast and flexible containers with dynamic resizing capabilities that enable storage of large amounts of data.

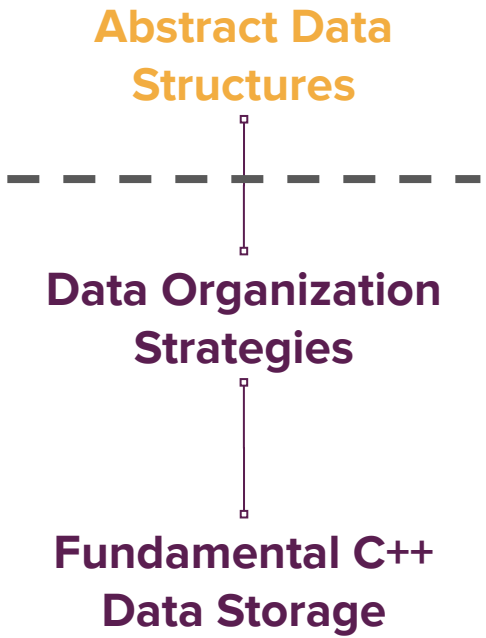
Multiple Levels of Abstraction

Levels of abstraction



Levels of abstraction

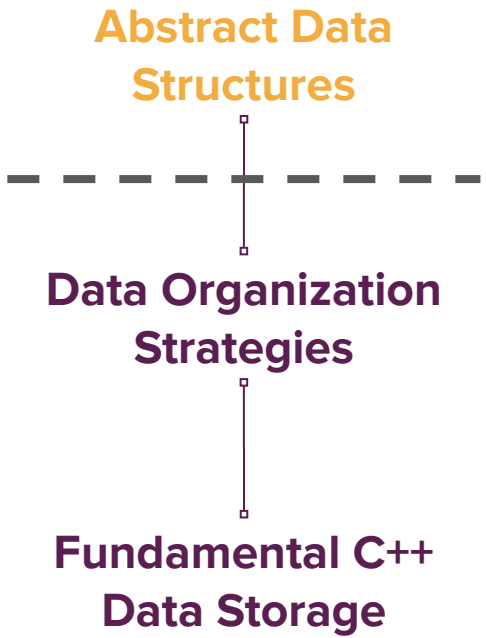
What is the interface for the user?
(Vectors, Sets, Queues, Grids, etc.)



Levels of abstraction

What is the interface for the user?
(Priority Queue)

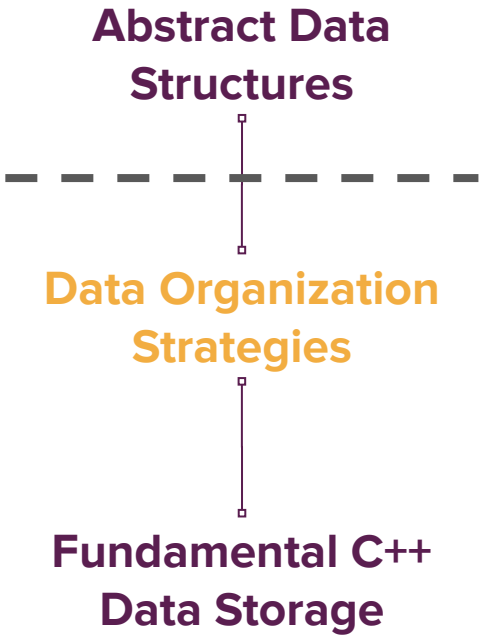
What you'll focus on for Assignment 4



Levels of abstraction

What is the interface for the user?
(Priority Queue)

How is our data organized?
(sorted array, binary heap)

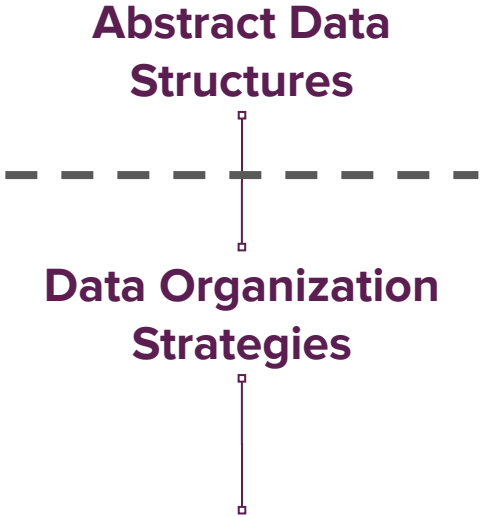


Levels of abstraction

What is the interface for the user?
(Priority Queue)

How is our data organized?
(sorted array, binary heap)

What stores our data?
(arrays, linked lists, etc.)



Levels of abstraction

What is the interface for the user?
(Priority Queue)

How is our data organized?
(sorted array, binary heap)

What stores our data?
(arrays)

Levels of abstraction

What is the interface for the user?
(**Priority Queue**)

How is our data organized?
(sorted array, **binary heap**)

tores our data?
(arrays)

What we'll focus on today!

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Priority Queues

What is a priority queue?

- A queue that orders its elements based on a provided “priority”

What is a priority queue?

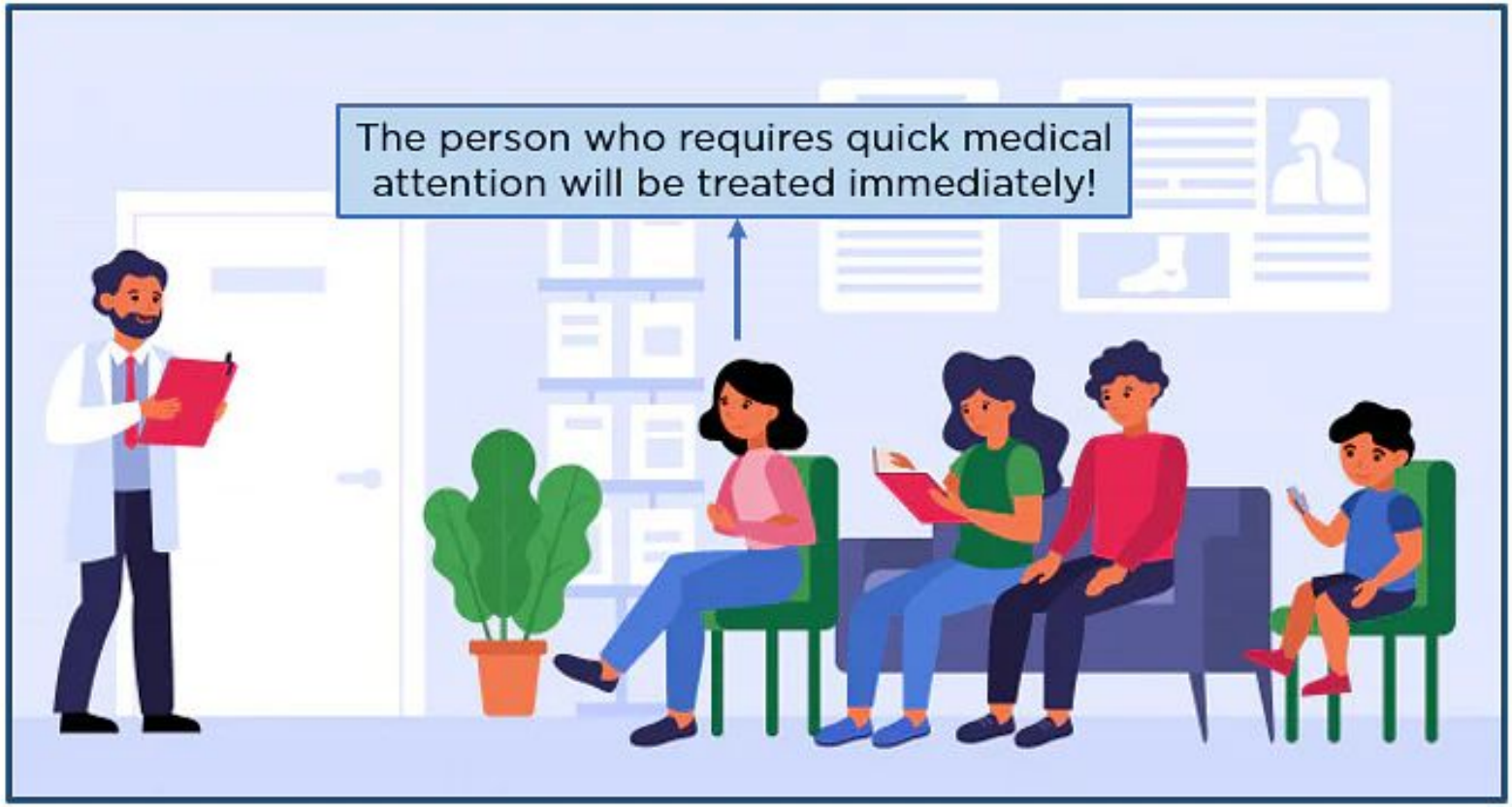
- A queue that orders its elements based on a provided “priority”

- Like regular queues, you cannot index into them to get an item at a particular position.

Where are they used?

- Medical queues: ER waiting rooms, organ matches, vaccine availability

Hospital Emergency Queue



Where are they used?

- Medical queues: ER waiting rooms, organ matches, vaccine availability

- Different airline boarding groups (families and first class passengers, frequent flyers, boarding group A, boarding group B, etc.)

Where are they used?

- Medical queues: ER waiting rooms, organ matches, vaccine availability
- Different airline boarding groups (**families and first class passengers**, frequent flyers, boarding group A, boarding group B, etc.)

Individual data points can have the same priority!

Where are they used?

- Medical queues: ER waiting rooms, organ matches, vaccine availability
- Different airline boarding groups (families and first class passengers, frequent flyers, boarding group A, boarding group B, etc.)
- Filtering data to get the top X results (e.g. most popular Google searches or fastest times for the Women’s 800m freestyle swimming event)

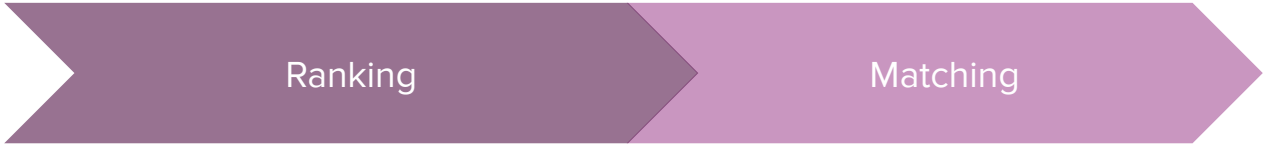
Where are they used?

- Medical queues: ER waiting rooms, organ matches, vaccine availability
- Different airline boarding groups (families and first class passengers, frequent flyers, boarding group A, boarding group B, etc.)
- Filtering data to get the top X results (e.g. most popular Google searches or fastest times for the Women’s 800m freestyle swimming event)
- College admissions
- Social assistance programs

Los Angeles County Coordinated Entry System (CES)

An electronic registry of unhoused persons who are applying or have applied to housing support programs offered by Los Angeles County.

How does it work?



Algorithm uses personal data to assign a number from 1-17, least vulnerable tomost vulnerable.

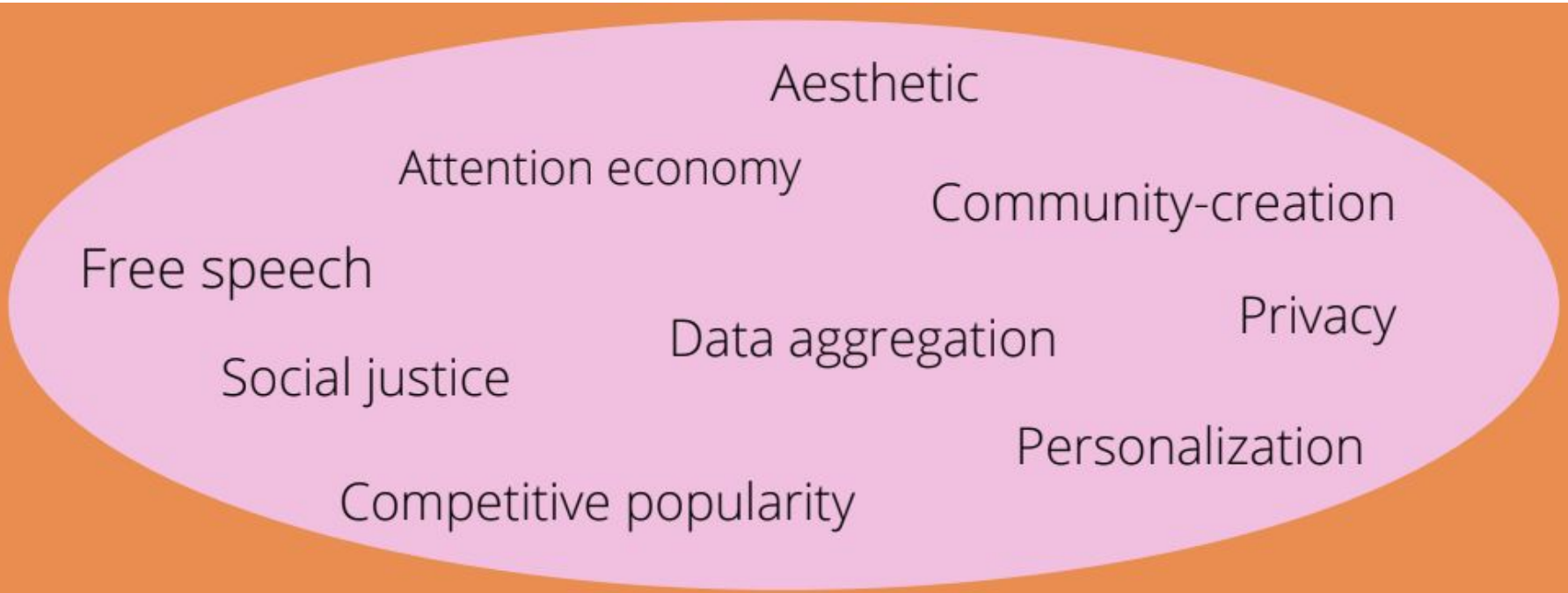
Risk score is used to prioritize and assign housing and housing related services.

How does it work?

???	Ranking	Matching
	Algorithm uses personal data to **assign a number from 1-17**, least vulnerable tomost vulnerable.	Risk score is used to prioritize and assign housing and housing related services.

Values in technology

- Design decisions encode values that express what we care about.



Values in technology

- Design decisions encode values that express what we care about.

- These values can reveal our assumptions about the world and the people who will be interacting with our design and benefiting from it.

Values in technology

- Design decisions encode values that express what we care about.
- These values can reveal our assumptions about the world and the people who will be interacting with our design and benefiting from it.
- Despite the best intentions, sometimes design decisions have unintended consequences that evoke different values than those of the original creators.

Values in technology

- Design decisions encode values that express what we care about.
- These values can reveal our assumptions about the world and the people who will be interacting with our design and benefiting from it.
- Despite the best intentions, sometimes design decisions have unintended consequences that evoke different values than those of the original creators.

In the case of priority queues, the "priorities" themselves represent a very explicit value system!

For now, we'll assume we have the priorities...

Three fundamental operations

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

Less fundamental 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

How do we design **`PriorityQueue`**?

1. Member functions: *What public interface should* **`PriorityQueue`** *support? What functions might a client want to call?*

2. Member variables: *What private information will we need to store in order to keep track of the data stored in* **`PriorityQueue`***?*

3. Constructor: *How are the member variables initialized when a new instance of* **`PriorityQueue`** *is created?*

How do we design **`PriorityQueue`**?

1. **Member functions**: *What public interface should* **`PriorityQueue`** *support? What functions might a client want to call?*

2. Member variables: *What private information will we need to store in order to keep track of the data stored in* **`PriorityQueue`***?*

3. Constructor: *How are the member variables initialized when a new instance of* **`PriorityQueue`** *is created?*

We'll provide the public interface...

How do we design **`PriorityQueue`**?

1. Member functions: *What public interface should* `PriorityQueue` *support? What functions might a client want to call?*

2. **Member variables**: *What private information will we need to store in order to keep track of the data stored in* **`PriorityQueue`***?*

3. **Constructor**: *How are the member variables initialized when a new instance of* **`PriorityQueue`** *is created?*

You get to decide on the implementation details!

How do we implement **`PriorityQueue`**?

- We want to be able to access the element that has the highest priority in constant-time (i.e. **`peek()`**).

How do we implement **`PriorityQueue`**?

- We want to be able to access the element that has the highest priority in constant-time (i.e. **`peek()`**).

- **Idea**: We can keep a sorted array where the elements are in order of their priority (highest priority is at the end of the array)!
  - Dequeue will be fast – just get the last element in the array.
  - But every time we enqueue something, we have to adjust the entire array...

How do we implement **`PriorityQueue`**?

- We want to be able to access the element that has the highest priority in constant-time (i.e. **`peek()`**).

- **Idea**: We can keep a sorted array where the elements are in order of their priority (highest priority is at the end of the array)!
  - Dequeue will be fast – just get the last element in the array.
  - But every time we **enqueue** something, we have to adjust the entire array...

You'll get to implement this on the assignment!

How do we implement **`PriorityQueue`**?

- We want to be able to access the element that has the highest priority in constant-time (i.e. **`peek()`**).

- **Idea**: We can keep a sorted array where the elements are in order of their priority (highest priority is at the end of the array)!
  - Dequeue will be fast – just get the last element in the array.
  - But every time we enqueue something, we have to adjust the entire array...

- Can we do better?

How do we implement **`PriorityQueue`**?

- We want to be able to access the element that has the highest priority in constant-time (i.e. **`peek()`**).

- **Idea**: We can keep a sorted array where the elements are in order of their priority (highest priority is at the end of the array)!
  - Dequeue will be fast – just get the last element in the array.
  - But every time we enqueue something, we have to adjust the entire array...

- Can we do better? (yes!)

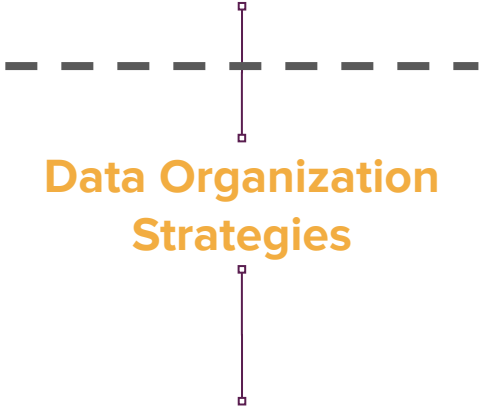
There are multiple possible implementations for the same ADT!

Levels of abstraction

What is the interface for the user?
(Priority Queue)

How is our data organized?
(sorted array, **binary heap**)

What stores our data?
(arrays)



Announcements

Announcements

- Midterm regrade requests are due **Friday by 11:59pm**.
  - If you opt into mid-quarter check-ins, you must also sign up to meet with your section leader via the IG scheduling feature on Paperless. Reach out to your section leader if there aren't times available yet!

- Assignment 2 revisions are due **Friday, July 22 at 11:59pm**.

- The final project proposals are due **Sunday, July 24 at 11:59pm**. Please read the full guidelines on the course website.

- Assignment 4 was released on last night and is due next **Tuesday, July 26 at 11:59pm**.
  - YEAH hours are **today at 5pm in Hewlett 103**. (Note the not-typical room!)

Binary Heaps

What is a binary heap?

- A heap is a tree-based structure that satisfies the *heap property* that **parents have a higher priority than any of their children**.

What is a binary heap?

- A heap is a tree-based structure that satisfies the *heap property* that **parents have a higher priority than any of their children**.

- Additional properties
  - **Binary**: Two children per parent (but no implied orderings between siblings)

What is a binary heap?

- A heap is a tree-based structure that satisfies the *heap property* that **parents have a higher priority than any of their children**.

- Additional properties
  - **Binary**: Two children per parent (but no implied orderings between siblings)
  - **Completely filled** (each parents must have 2 children) except for the bottom level, which gets populated from **left to right**

What is a binary heap?

- A heap is a tree-based structure that satisfies the *heap property* that **parents have a higher priority than any of their children**.

- Additional properties
  - **Binary**: Two children per parent (but no implied orderings between siblings)
  - **Completely filled** (each parents must have 2 children) except for the bottom level, which gets populated from **left to right**

- Two types ➜ which we use depends on what we define as a “higher” priority
  - Min-heap: smaller numbers = higher priority (closer to the root)
  - Max-heap: larger numbers = higher priority (closer to the root)

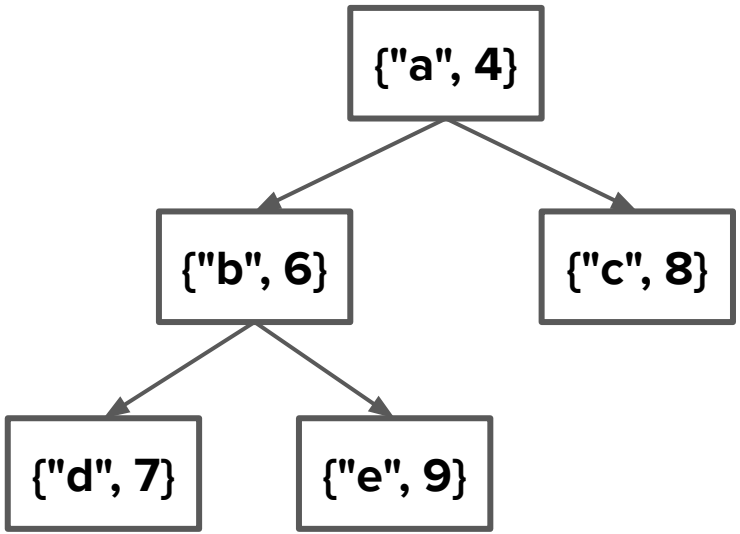
What is a binary heap?

- A heap is a tree-based structure that satisfies the *heap property* that **parents have a higher priority than any of their children**.

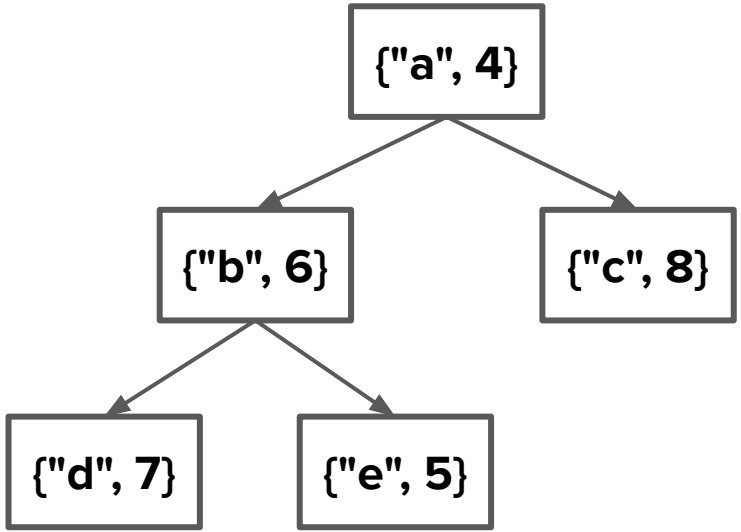
- Additional properties
  - **Binary**: Two children per parent (but no implied orderings between siblings)
  - **Completely filled** (each parents must have 2 children) except for the bottom level, which gets populated from **left to right**

- Two types ➜ which we use depends on what we define as a “higher” priority
  - **Min-heap**: smaller numbers = higher priority (closer to the root)
  - Max-heap: larger numbers = higher priority (closer to the root)

Spot the Valid Min-Heap

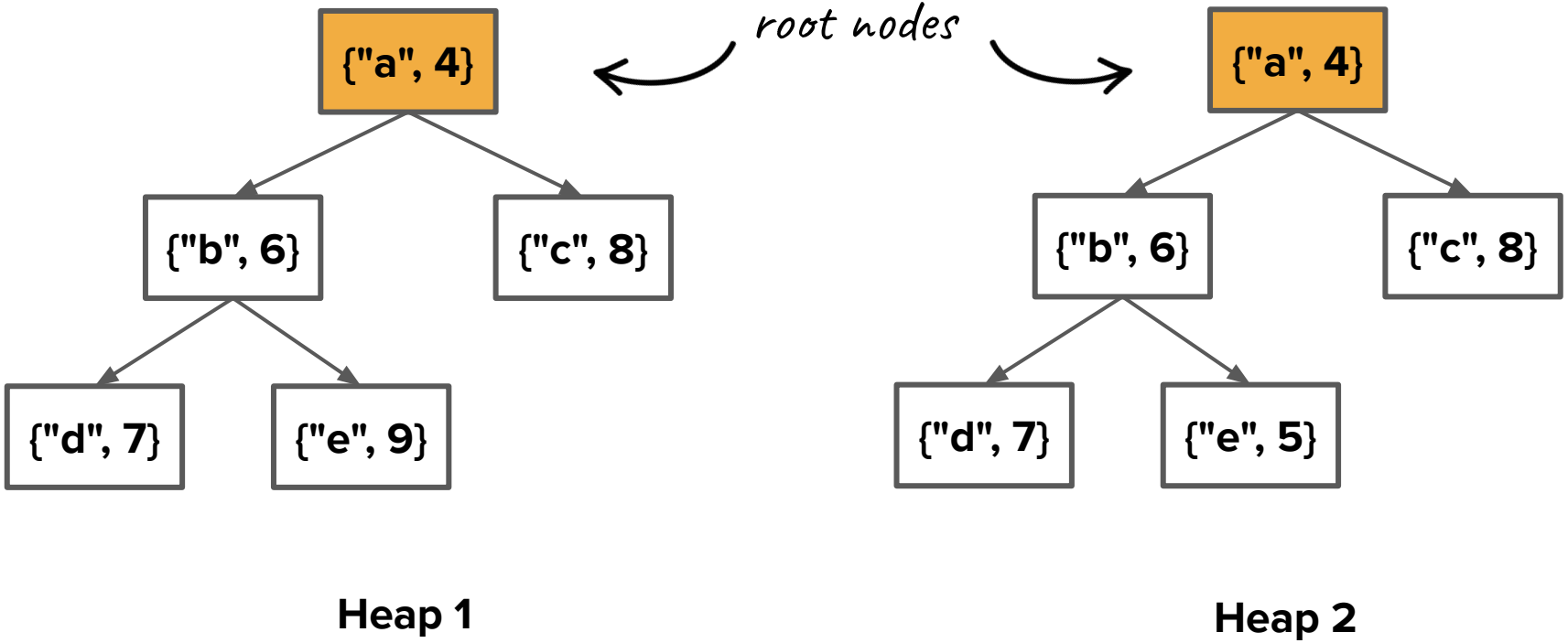


Heap 1



Heap 2

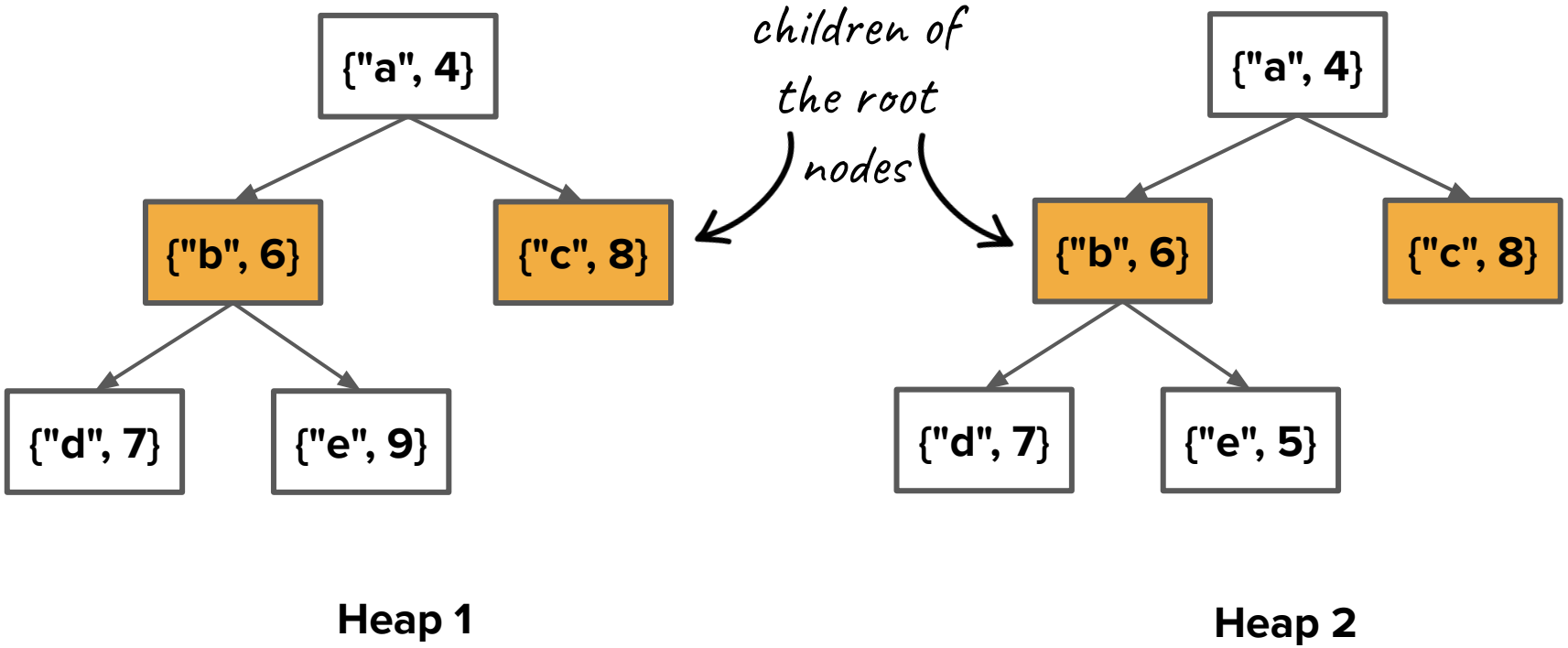
Spot the Valid Min-Heap



Heap 1

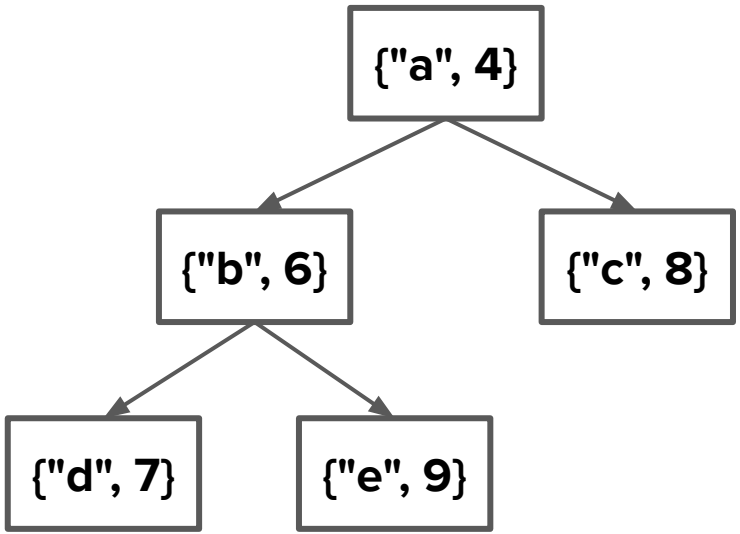
Heap 2

Spot the Valid Min-Heap

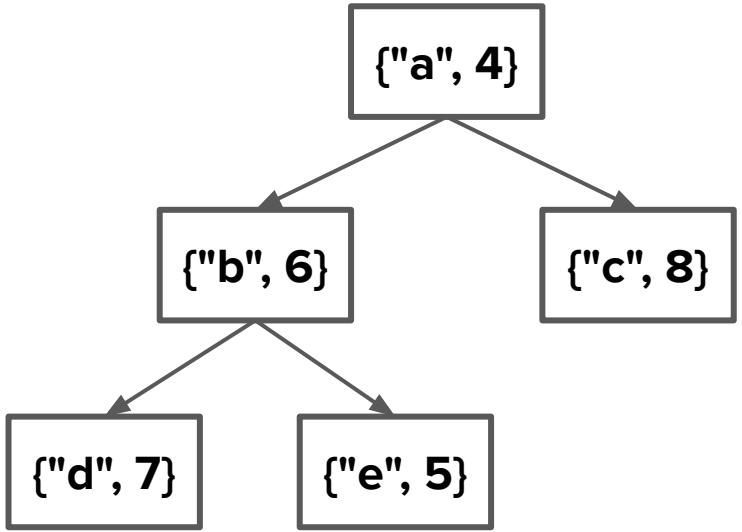


Spot the Valid Min-Heap

Poll: Which of these heaps is a valid min-heap?



Heap 1



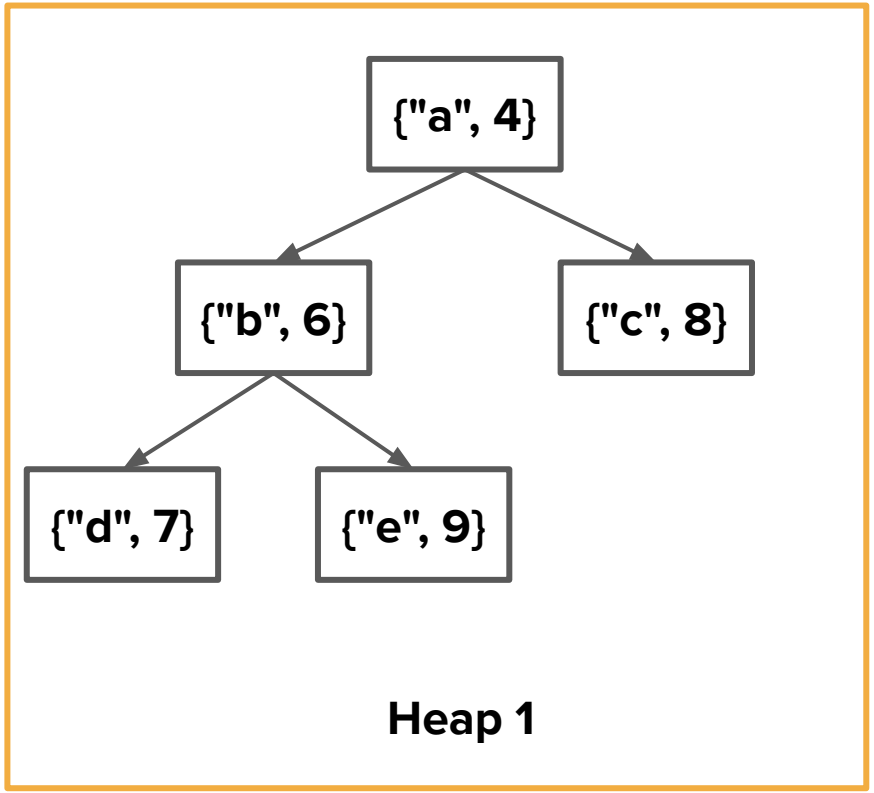
Heap 2

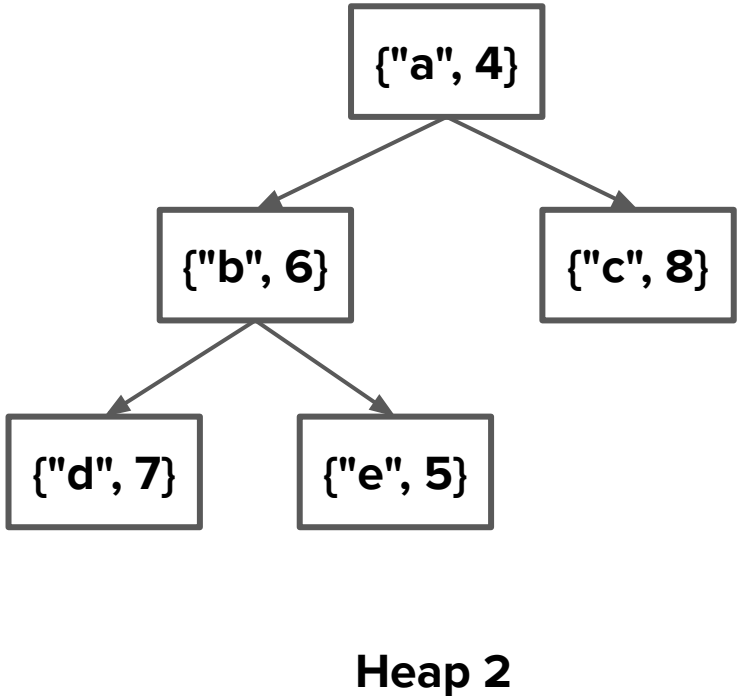
Which of these heaps is a valid min-heap?

Heap 1

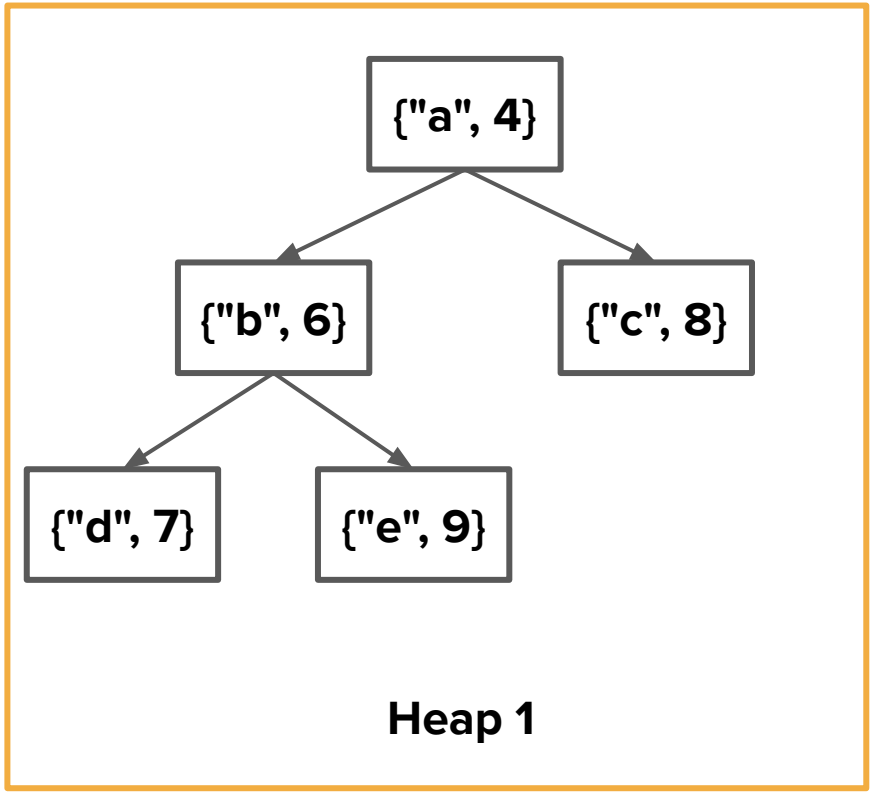
Heap 2

Spot the Valid Min-Heap



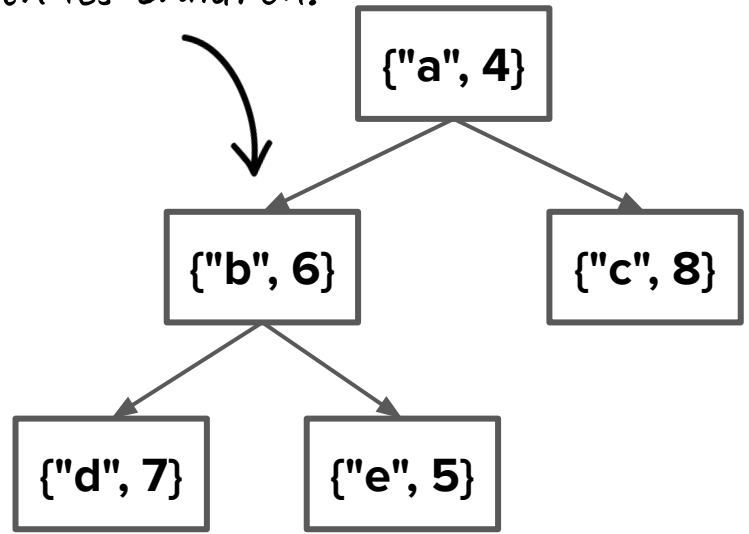


Spot the Valid Min-Heap



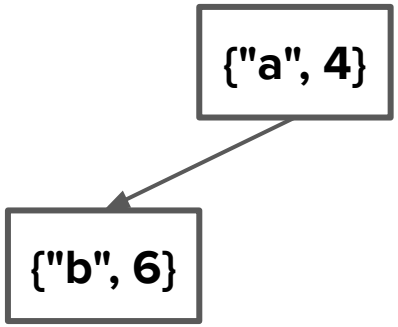
Heap 1

This element is not smaller than both its children!

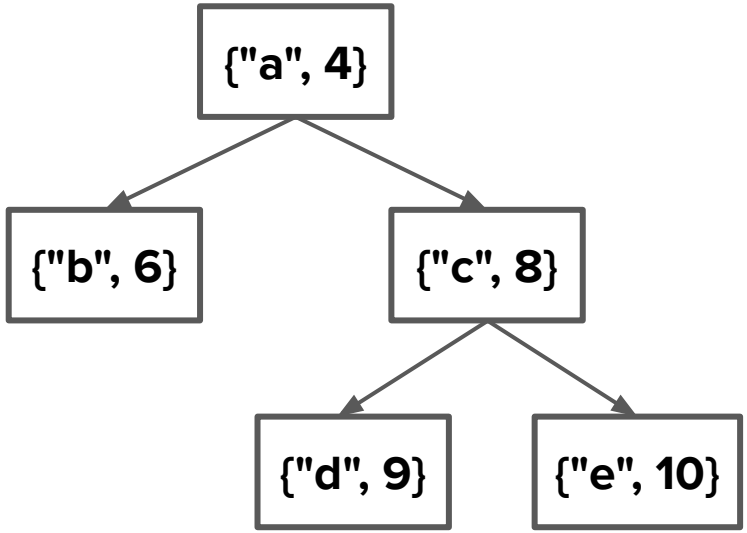


Heap 2

Spot the Valid Min-Heap (Round 2)

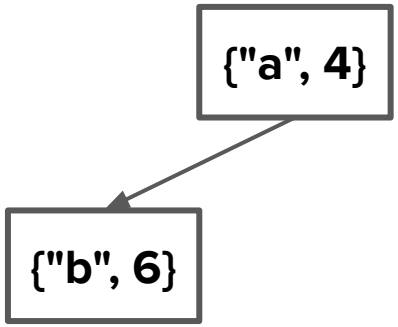


Heap 1

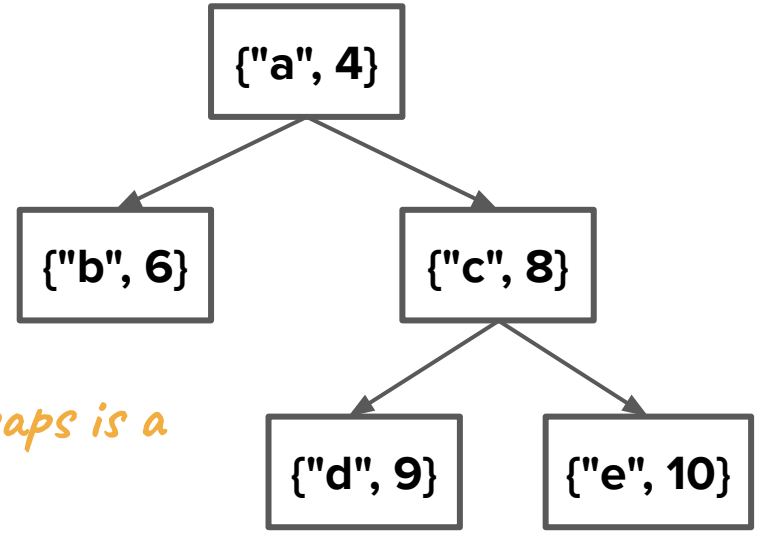


Heap 2

Spot the Valid Min-Heap (Round 2)



Heap 1



Heap 2

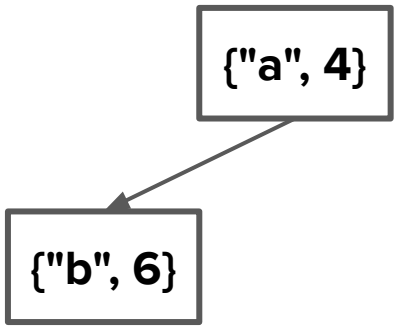
Poll: Which of these heaps is a valid min-heap?

Which of these heaps is a valid min-heap?

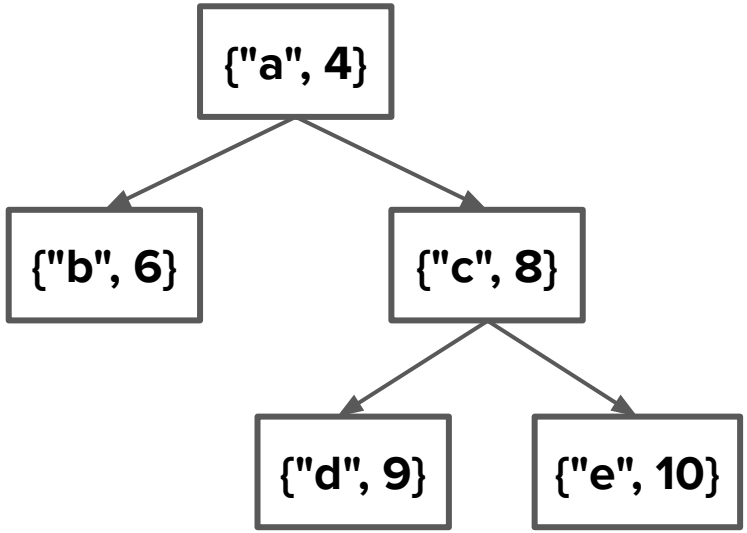
Heap 1

Heap 2

Spot the Valid Min-Heap (Round 2)

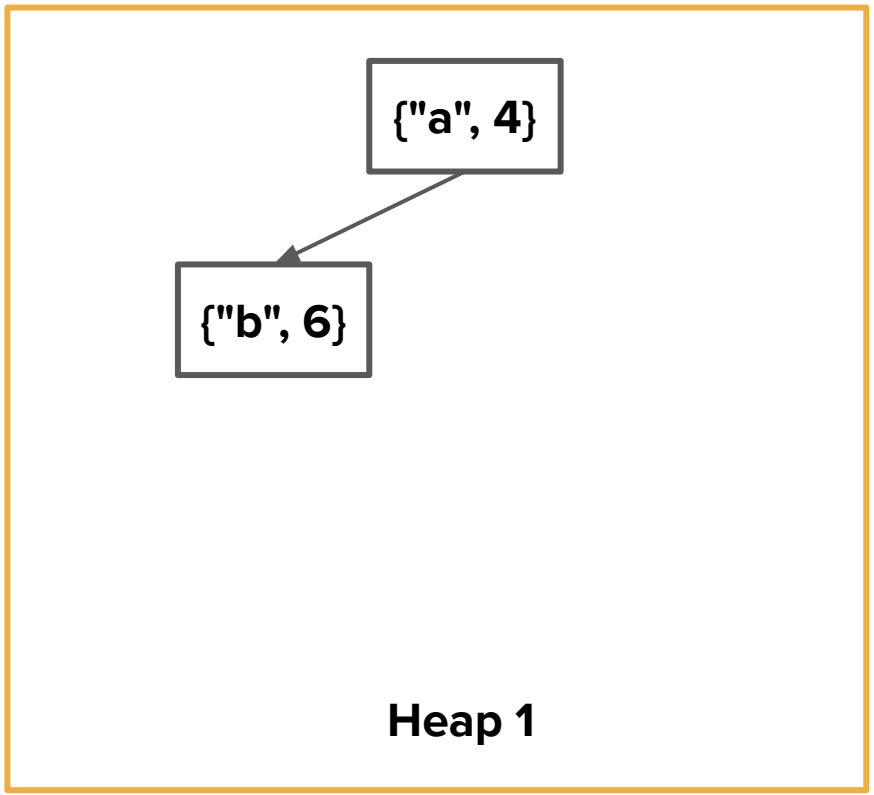


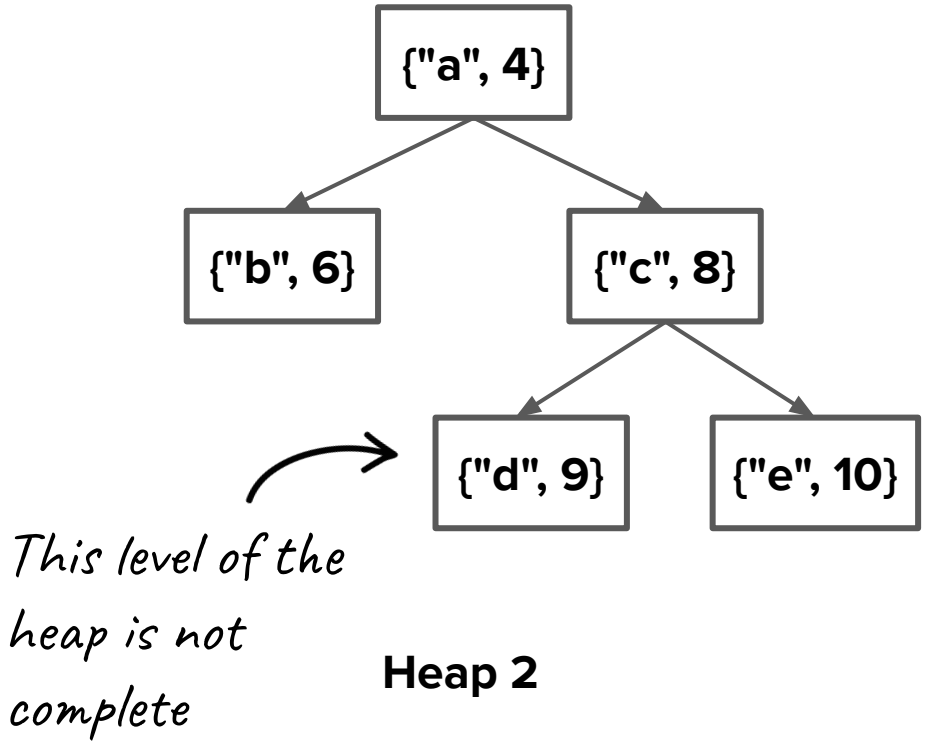
Heap 1



Heap 2

Spot the Valid Min-Heap (Round 2)





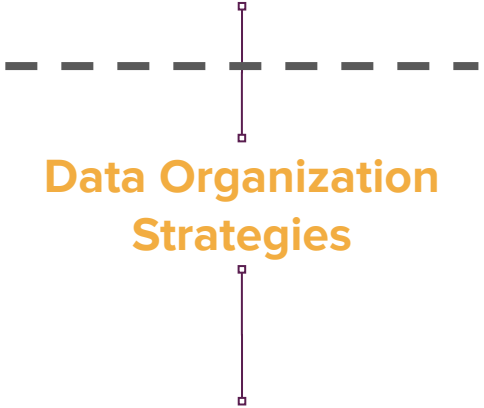
Binary heaps and implementation

Binary heaps and implementation

What is the interface for the user?
(Priority Queue)

How is our data organized?
(sorted array, **binary heap**)

What stores our data?
(arrays)

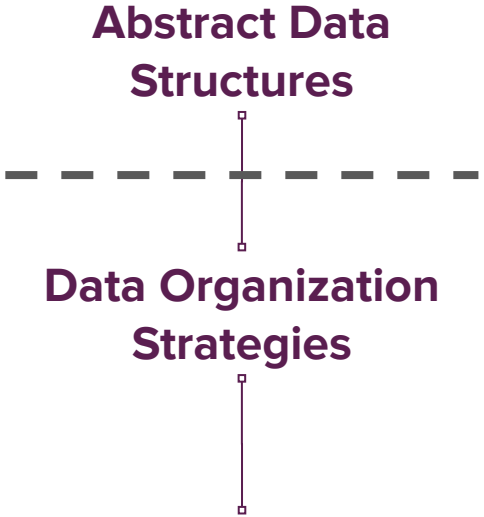


Binary heaps and implementation

What is the interface for the user?
(Priority Queue)

How is our data organized?
(**binary heap**)

What stores our data?
(**arrays**)



Binary heaps + implementation

- Binary heaps are both another way to implement **`PriorityQueue`** and also an abstraction on top of arrays!

Binary heaps + implementation

- Binary heaps are both another way to implement **`PriorityQueue`** and also an abstraction on top of arrays!

- Later, we will see a different approach to storing tree structures, but for heaps (which look like trees), the best solution is actually a simple array.
  - The reason for this is because of the **complete** nature of the structure, with all levels filled from left to right.

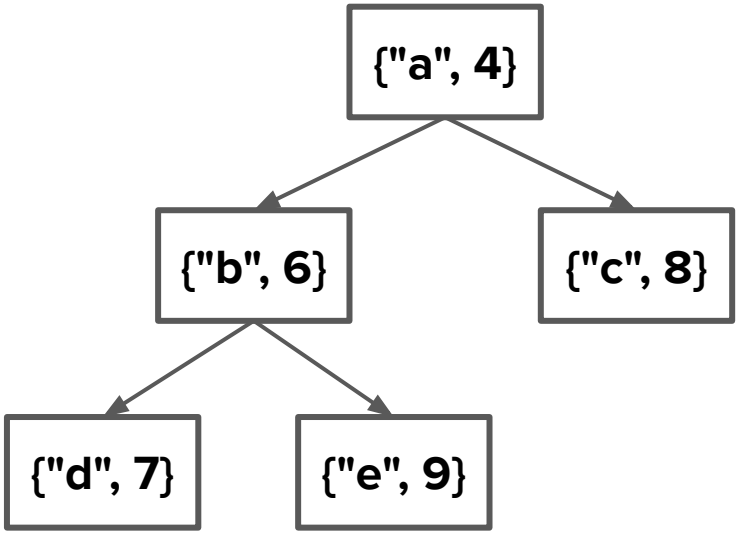
Binary heaps + implementation

- Binary heaps are both another way to implement **`PriorityQueue`** and also an abstraction on top of arrays!

- Later, we will see a different approach to storing tree structures, but for heaps (which look like trees), the best solution is actually a simple array.
  - The reason for this is because of the **complete** nature of the structure, with all levels filled from left to right.

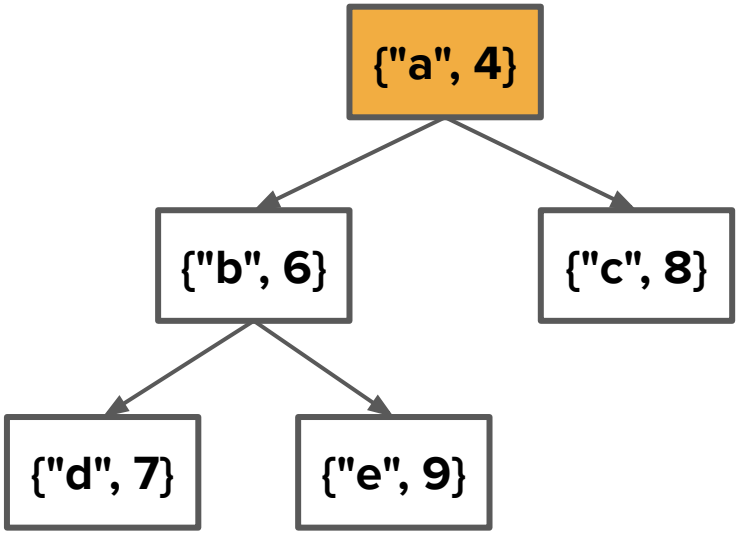
How are parents and children in the tree related in the array?

Binary heaps + implementation



{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

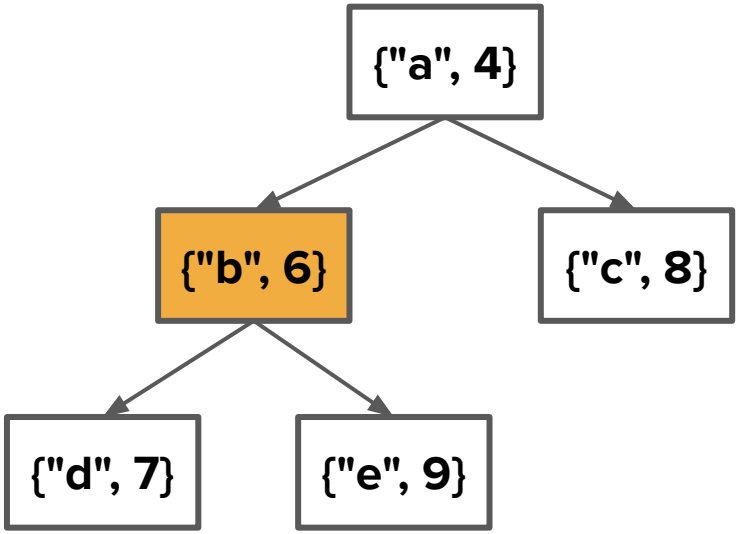
Binary heaps + implementation



{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

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

Binary heaps + implementation

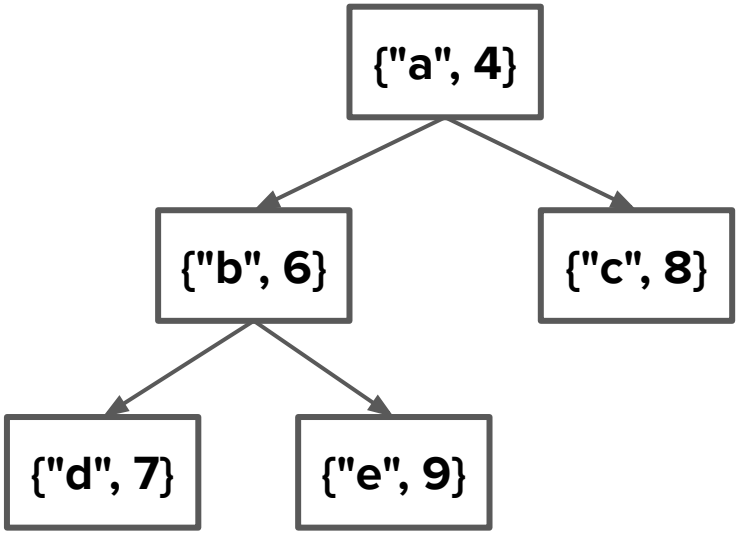


{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

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

Binary heaps + implementation

Parent: `i`
Left child: **`2*i + 1`**
Right child: **`2*i + 2`**



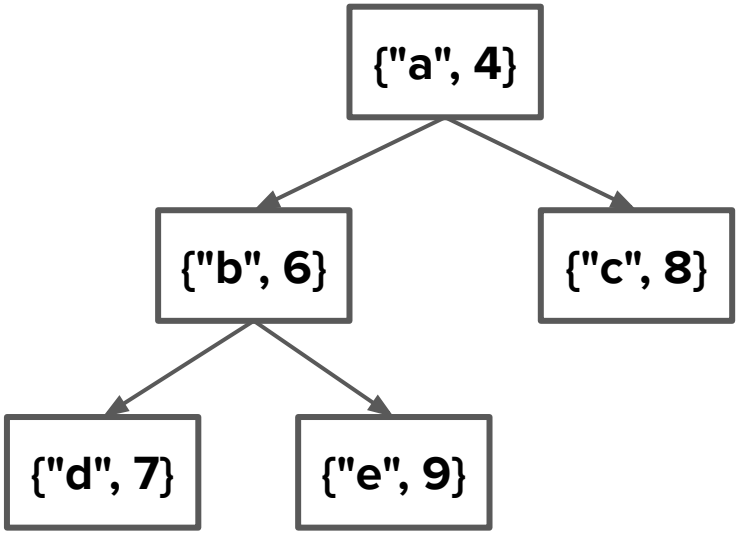
{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

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

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

Binary heaps + implementation

Parent: **`(i-1) / 2`**
Child: `i`

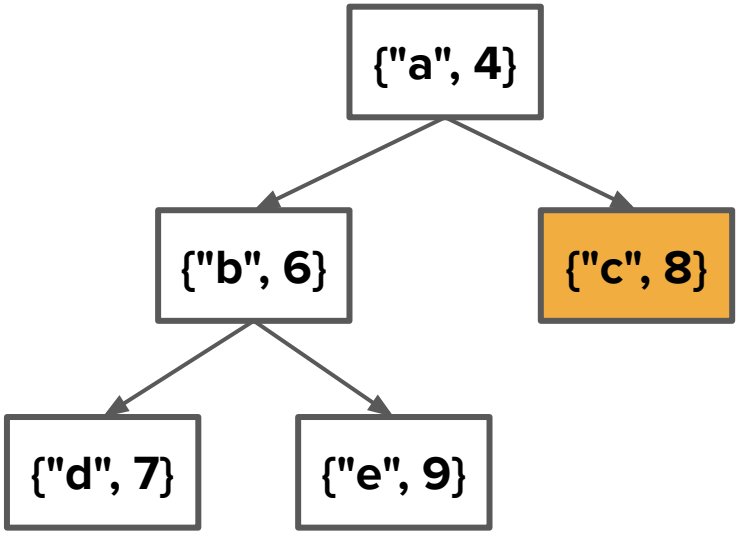


{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

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

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

Binary heaps + implementation



{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

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

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

Parent index: 2
Left child: 5
Right child: 6

Manipulating heap contents

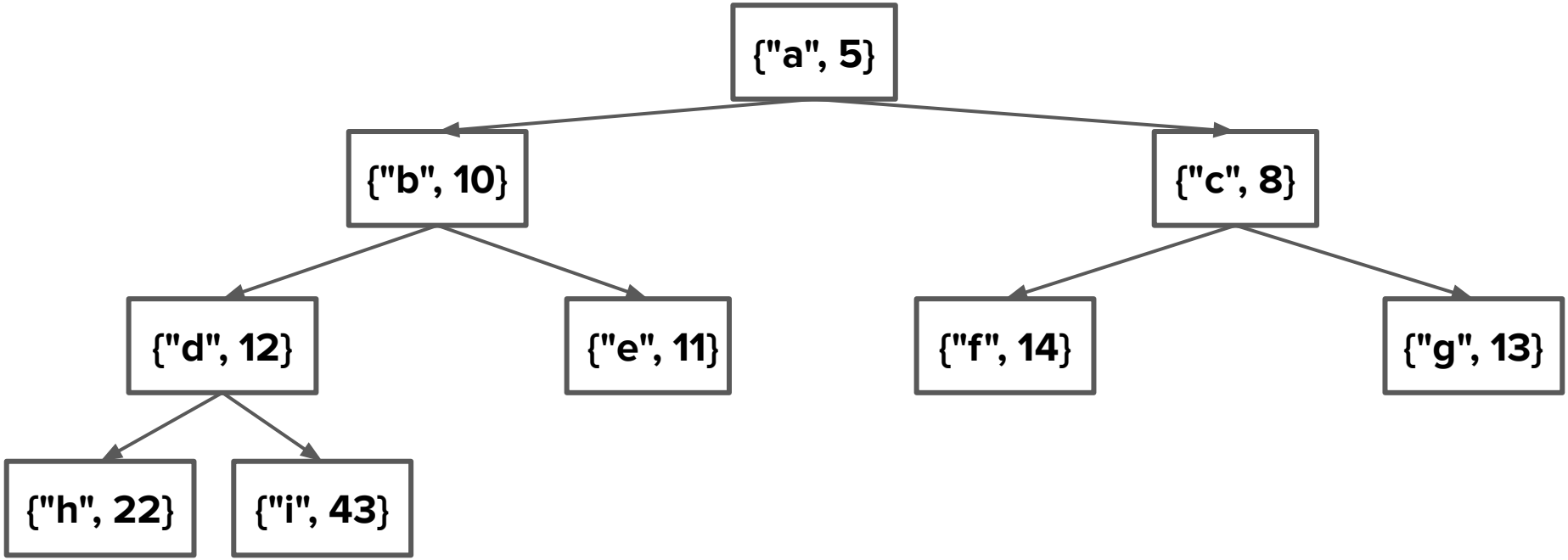
Heap operations

There are three important operations in a heap:

- **`peek()`:** return the element with the highest priority (lowest number for a min-heap). This operation does not change the state of the heap at all.

- **`enqueue(e)`**: insert an element **`e`** into the heap. Insertion of this element must result in a heap that still retains the heap property! Accomplishing this will require some clever manipulation.

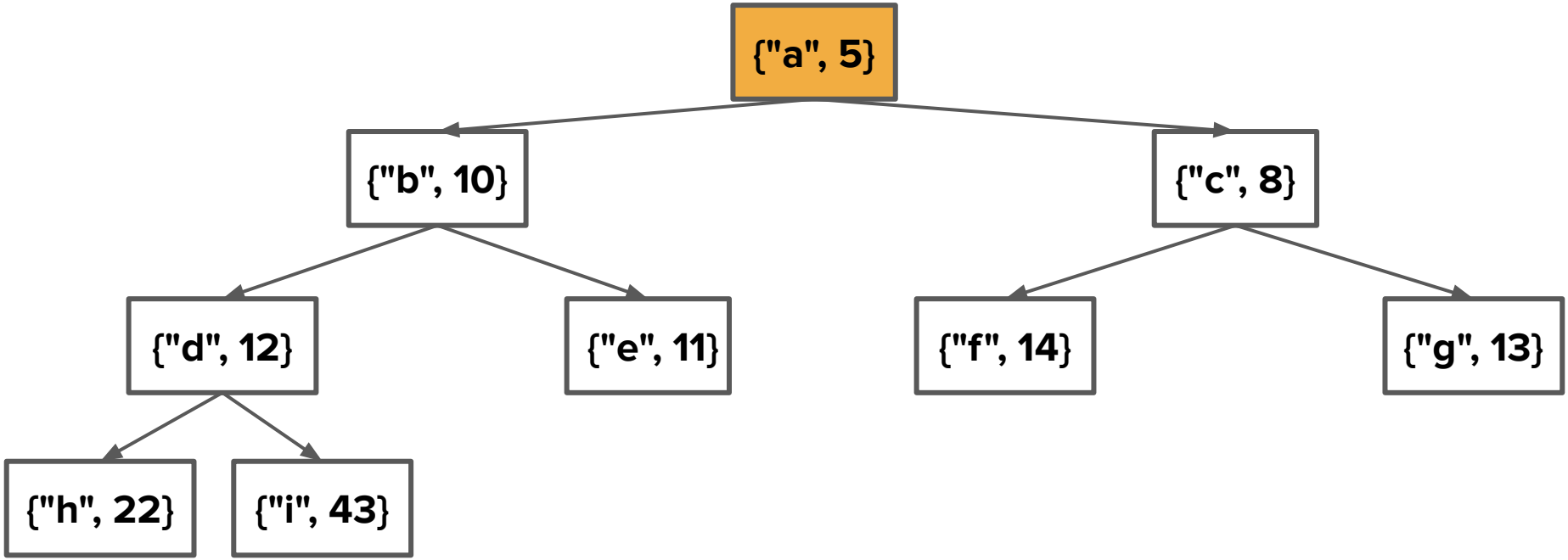
- **`dequeue()`:** remove the highest priority (smallest element for a min-heap) from the heap. This changes the state of the heap, and thus we have to do work to restore the heap property.



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

`peek()`

- Look at the root of the tree (position 0 in your array)
- `O(1)`



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

`enqueue()`

- How might we go about inserting into a binary heap?
- Example: What if we called **`enqueue({"j", 9})`** into the heap from before?
- The key is to understand how heaps are built: it is critical that we fill each level from left to right.

`enqueue()`

- Start by putting the element into the first empty slot at the bottom level. Similar to how we did with the **`OurVector`** class, we can say something along the lines of **`heap[heapSize] = newElement;`**

- Inserting our new element into the first empty slot may have destroyed the heap property so now we have to fix it.

- To do so, we **"bubble up"** the new element into its correct spot.

`enqueue()`

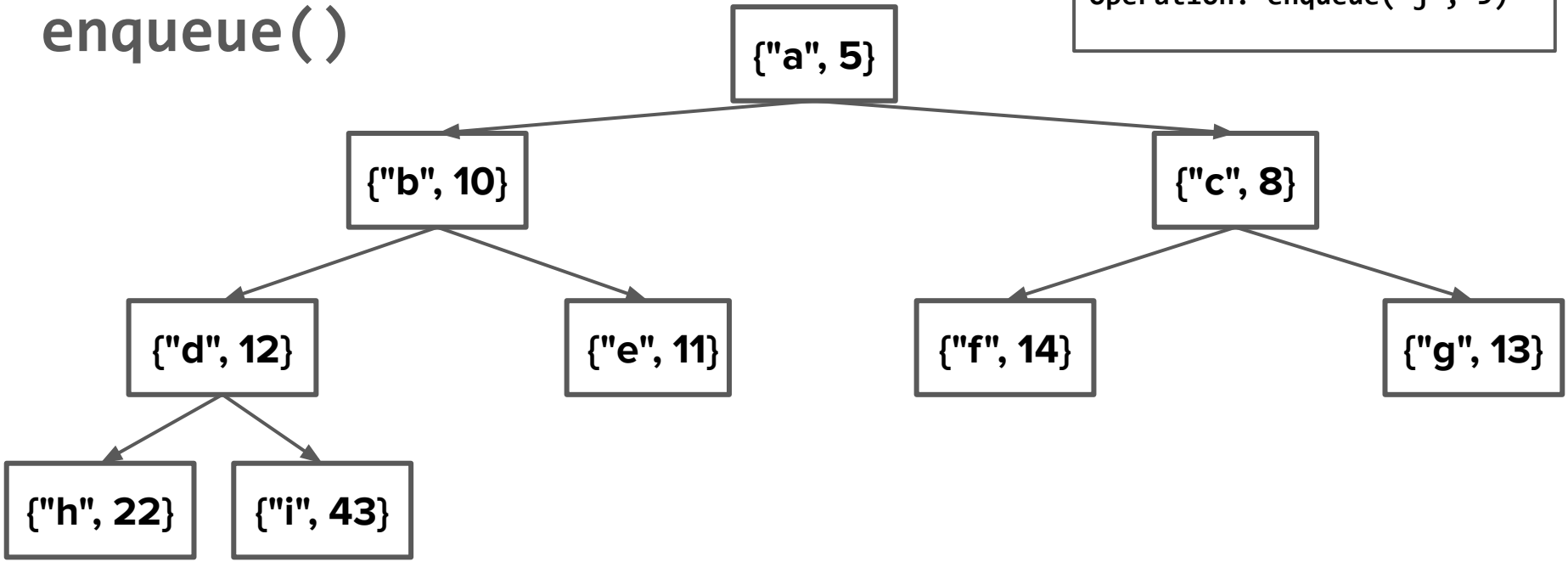
- Start by putting the element into the first empty slot at the bottom level. Similar to how we did with the **`OurVector`** class, we can say something along the lines of **`heap[heapSize] = newElement;`**

- Inserting our new element into the first empty slot may have destroyed the heap property so now we have to fix it.

- To do so, we **"bubble up"** the new element into its correct spot.

enqueue()

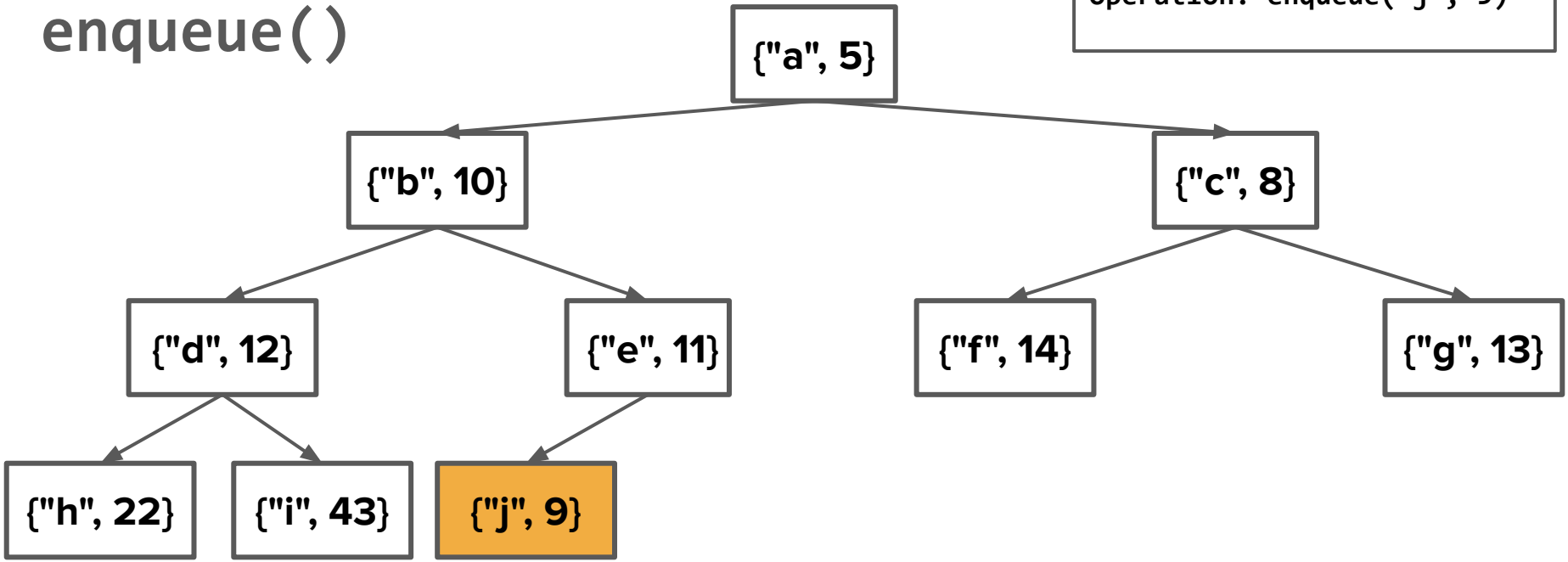
```
Operation: enqueue("j", 9)
```



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

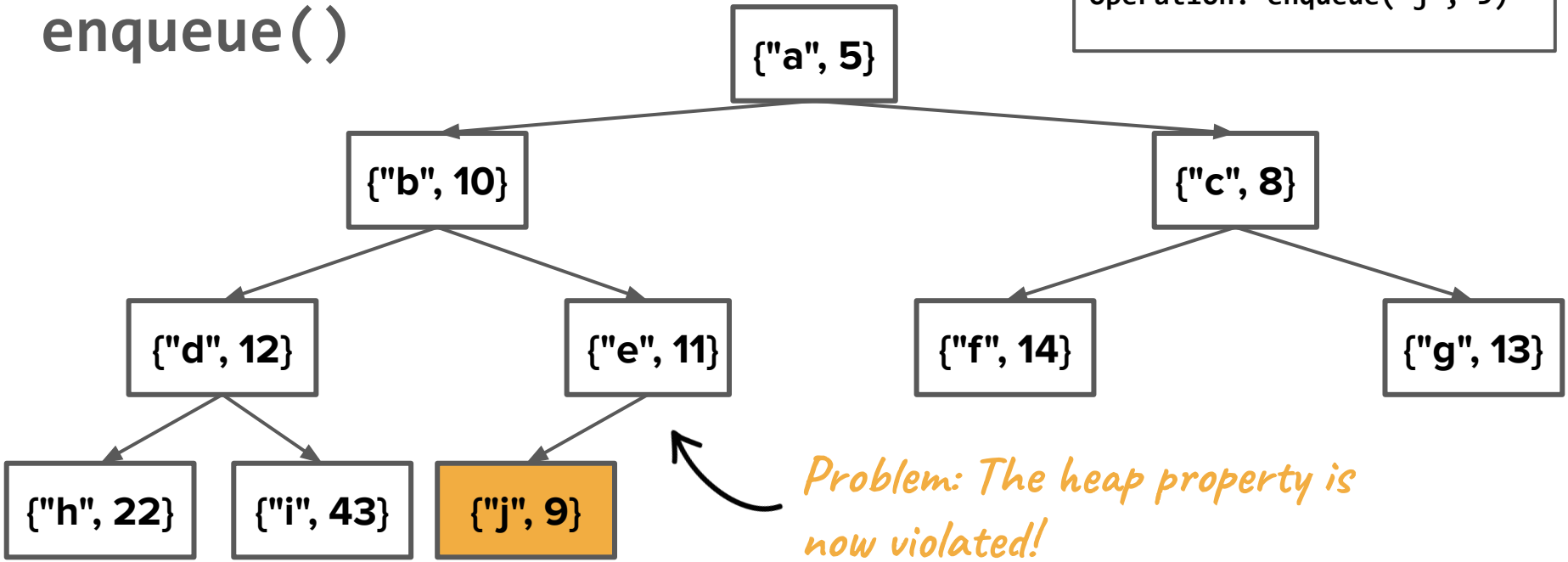
Operation: enqueue("j", 9)



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"j", 9}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"j", 9}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

`enqueue()`

- Start by putting the element into the first empty slot at the bottom level. Similar to how we did with the `OurVector` class, we can say something along the lines of `heap[heapSize] = newElement;`

- Inserting our new element into the first empty slot may have destroyed the heap property so now we have to fix it.

- To do so, we **"bubble up"** the new element into its correct spot.

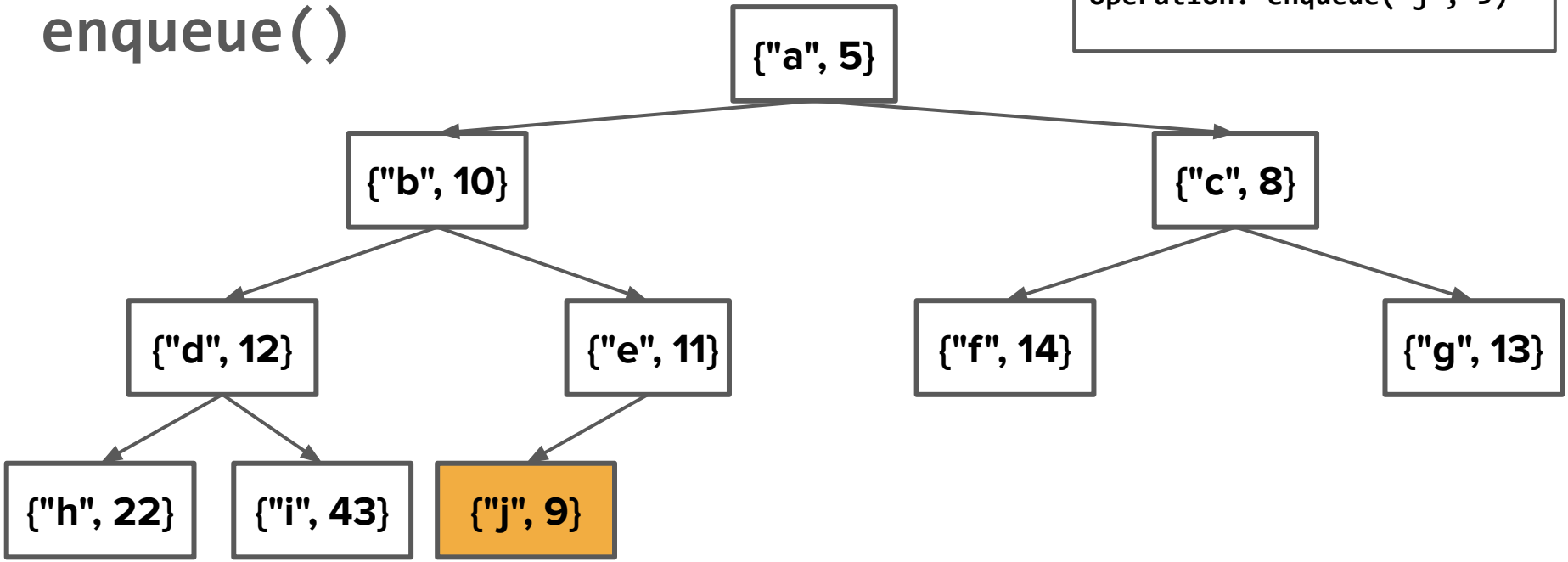
enqueue()

- Inserting our new element into the first empty slot may have destroyed the heap property so now we have to fix it.

- To do so, we **"bubble up"** the new element into a spot in the heap that is more fitting of its priority.
  - Look at the newly added element and its parent. Do they have a proper min-heap relationship (that is, is the parent smaller than the child element)?
    - If yes, then we're done, terminate the bubble up process.
    - If not, **swap the newly added element with its parent.**
  - Repeat the above steps until the process terminates or until the newly added element becomes the root of the heap.

enqueue()

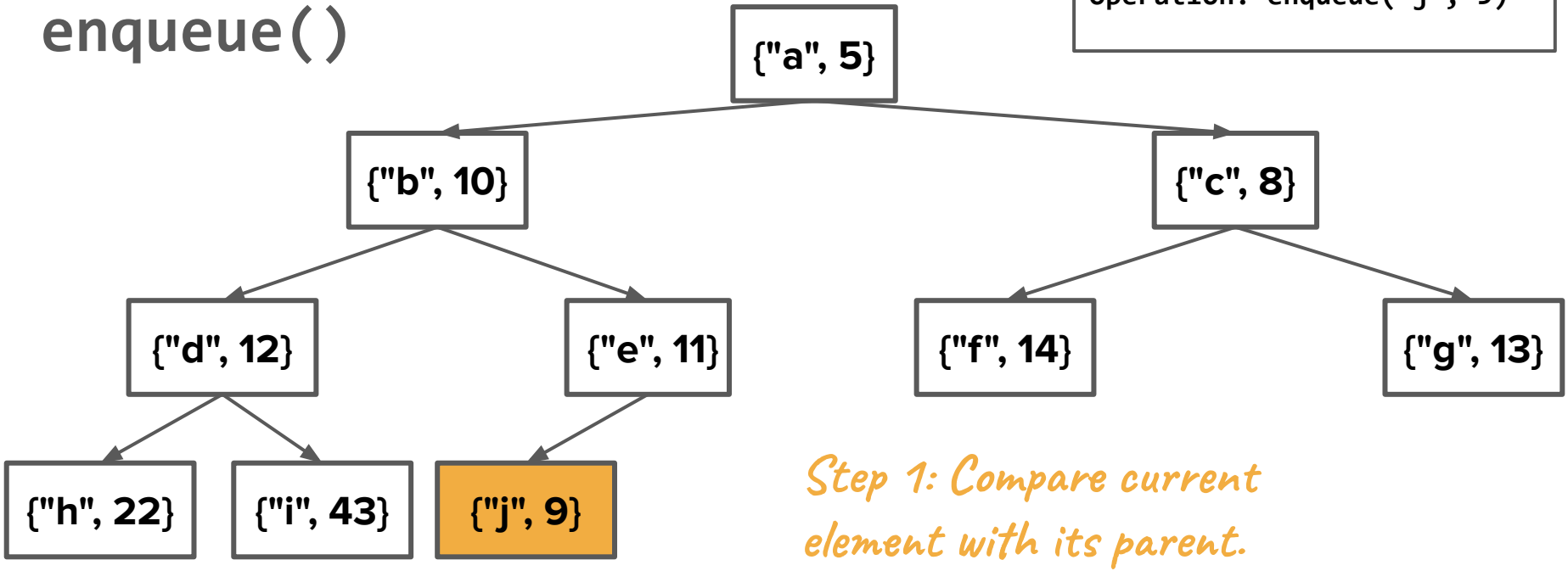
Operation: enqueue("j", 9)



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"j", 9}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

```
Operation: enqueue("j", 9)
```

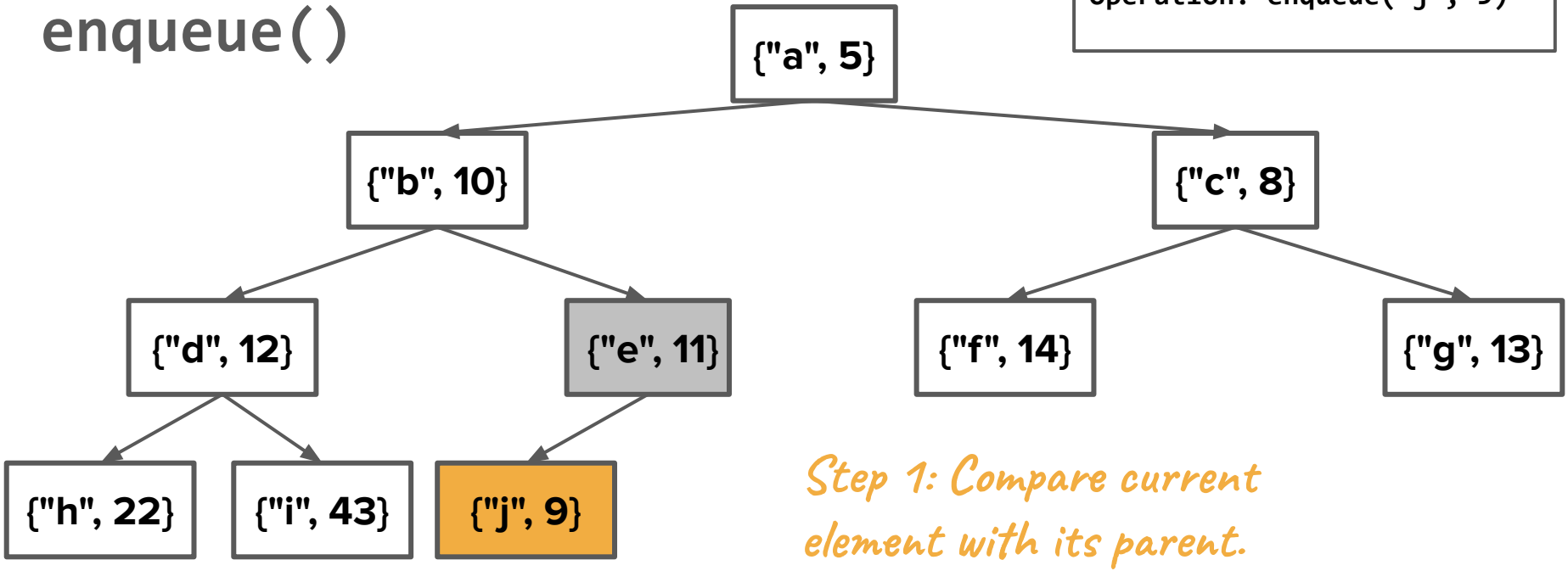


Step 1: Compare current element with its parent.

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"j", 9}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

```
Operation: enqueue("j", 9)
```

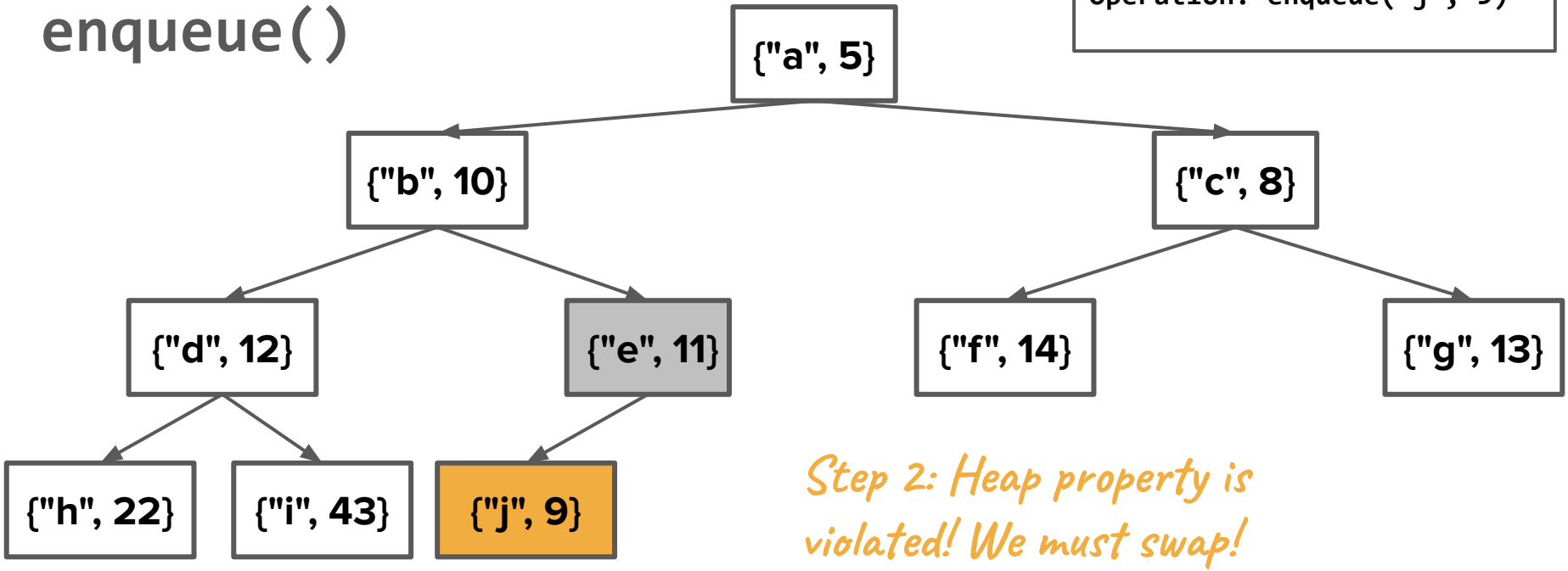


Step 1: Compare current element with its parent.

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"j", 9}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

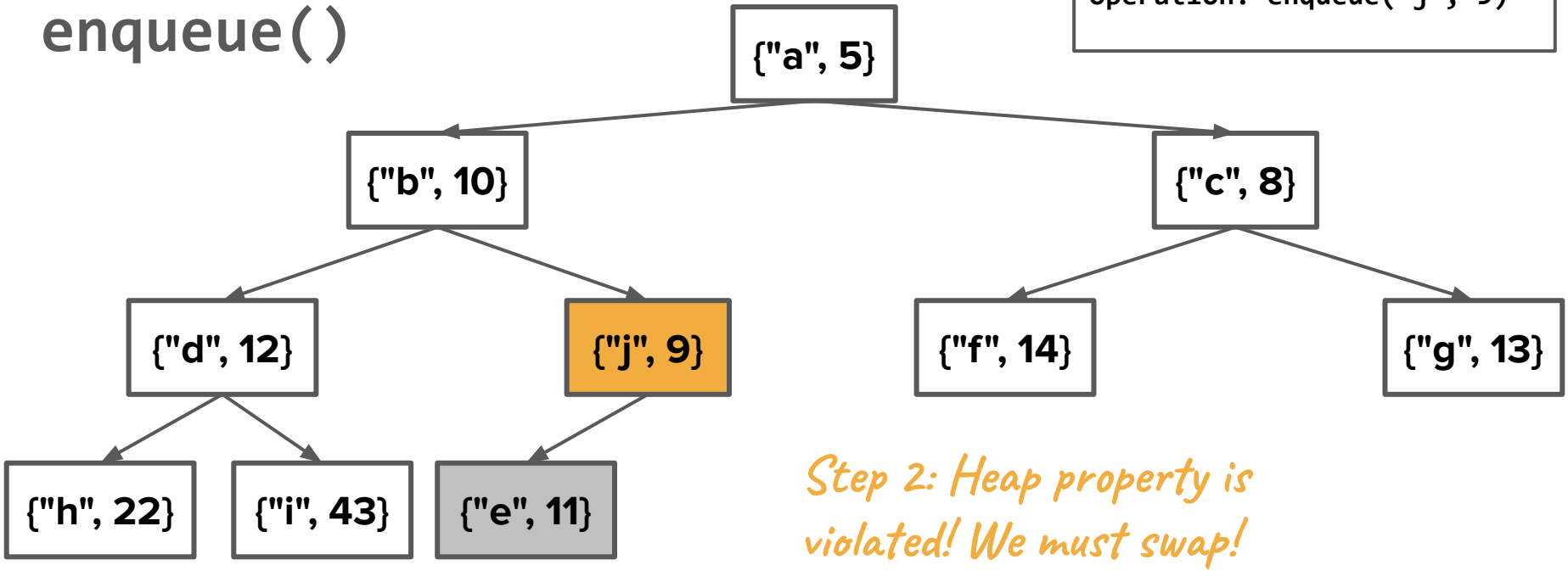


Step 2: Heap property is violated! We must swap!

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"j", 9}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

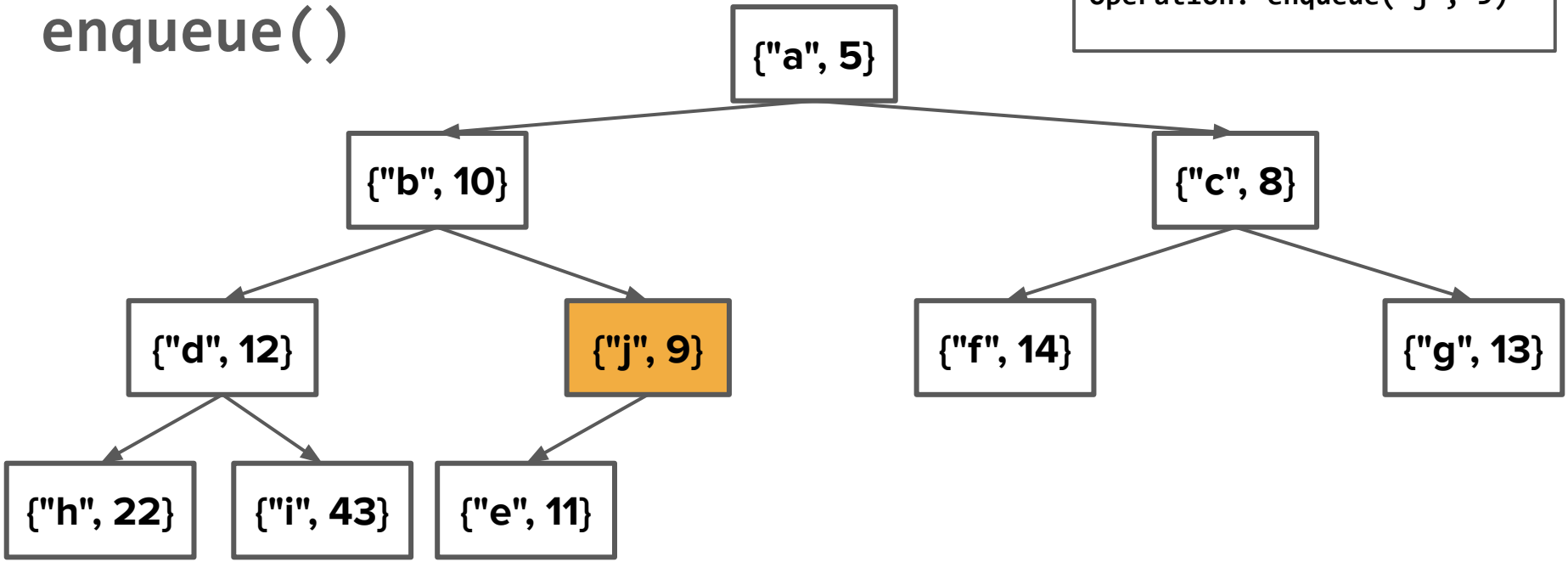


Step 2: Heap property is violated! We must swap!

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"j", 9}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

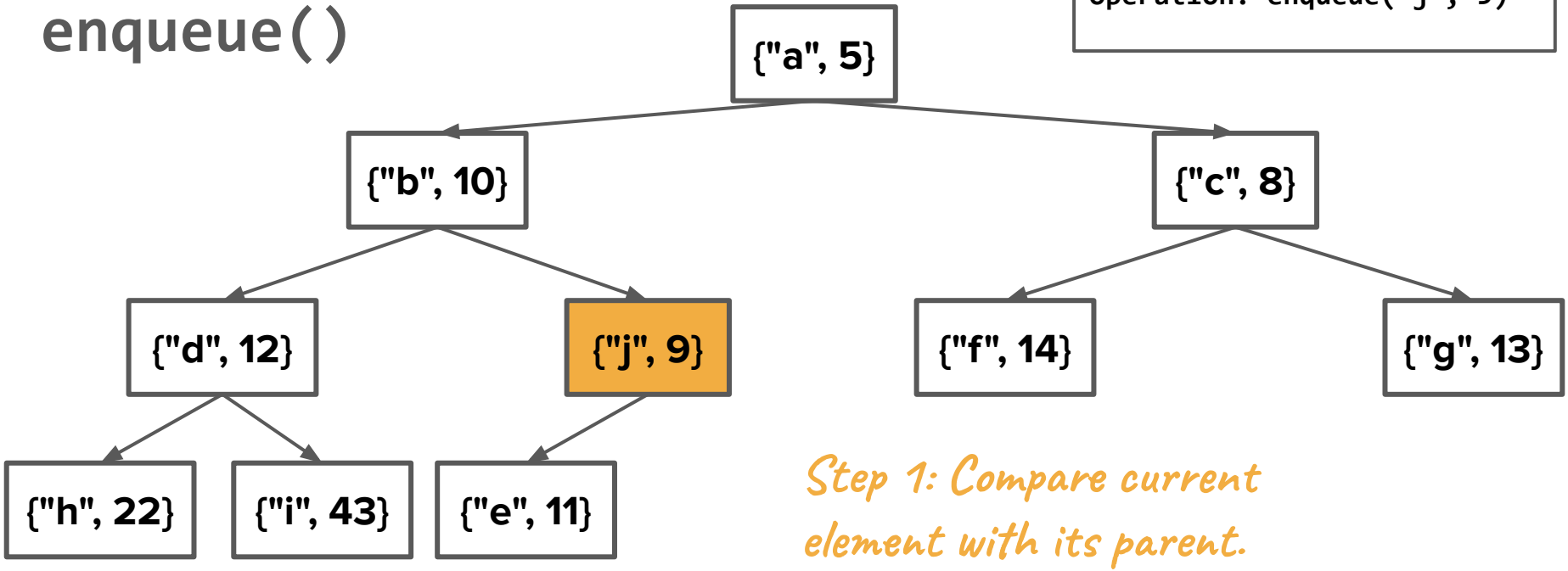
Operation: enqueue("j", 9)



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"j", 9}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

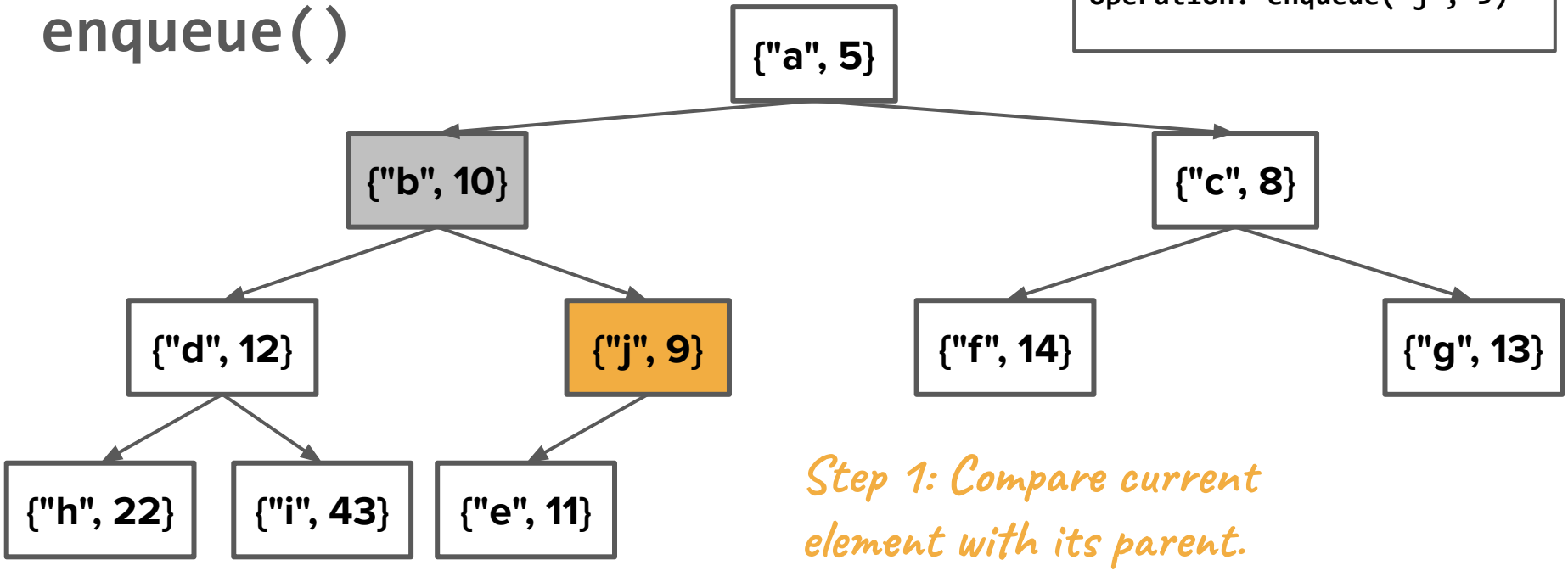


Step 1: Compare current element with its parent.

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"j", 9}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

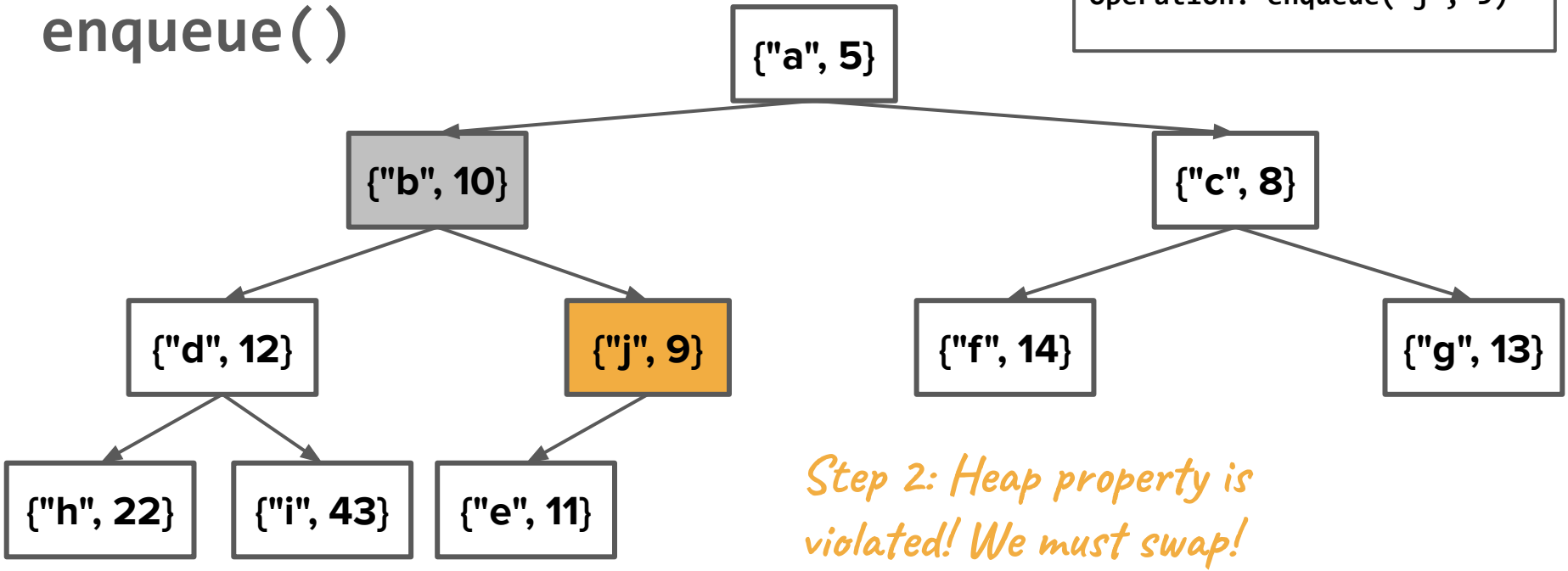


Step 1: Compare current element with its parent.

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"j", 9}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

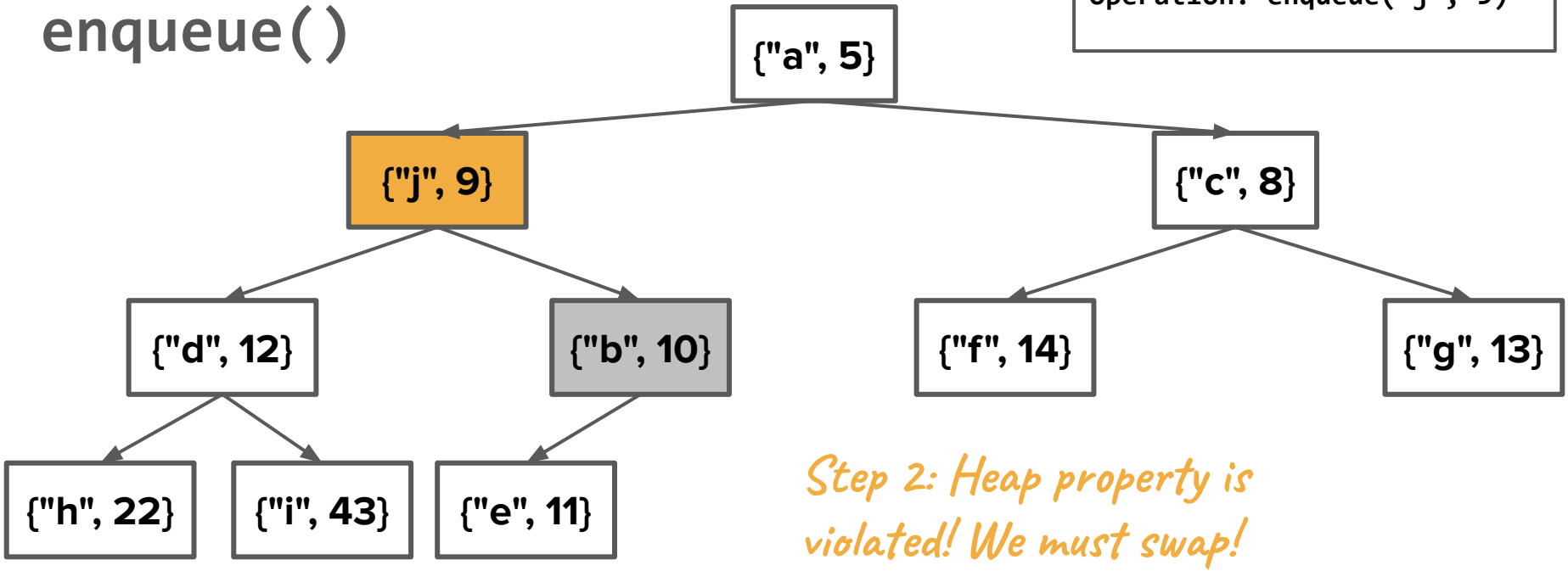


Step 2: Heap property is violated! We must swap!

{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"j", 9}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

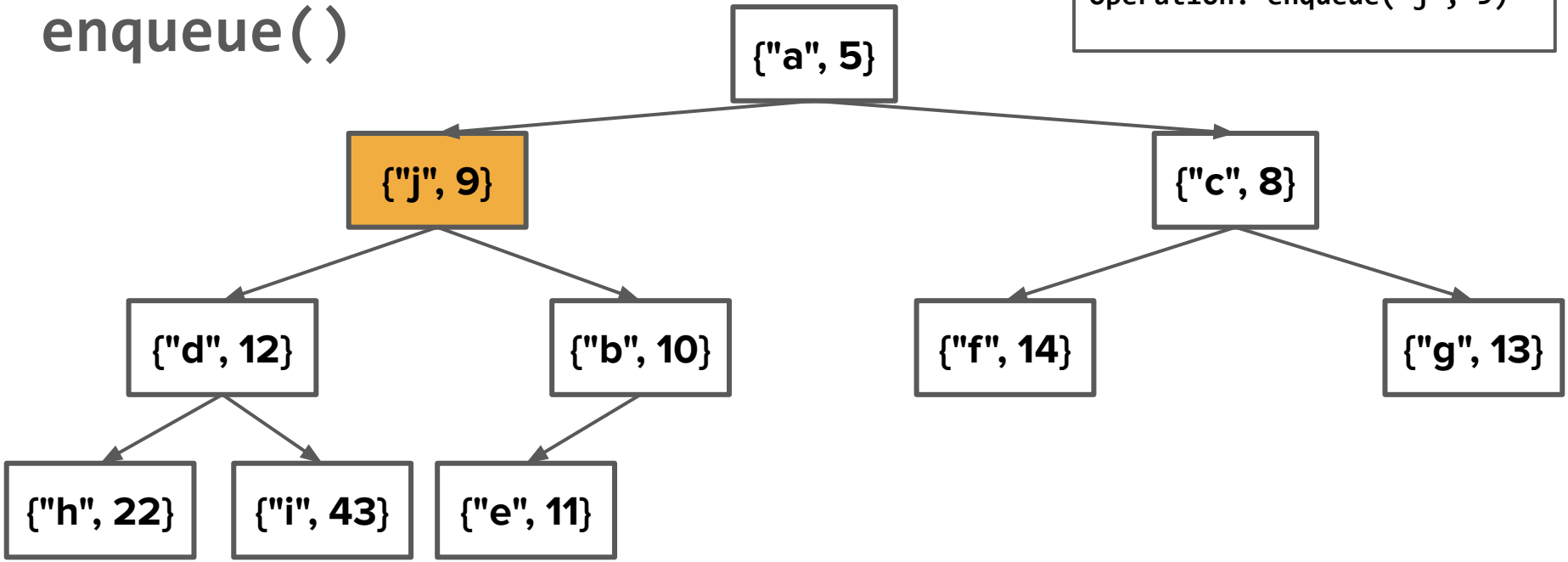


Step 2: Heap property is violated! We must swap!

{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

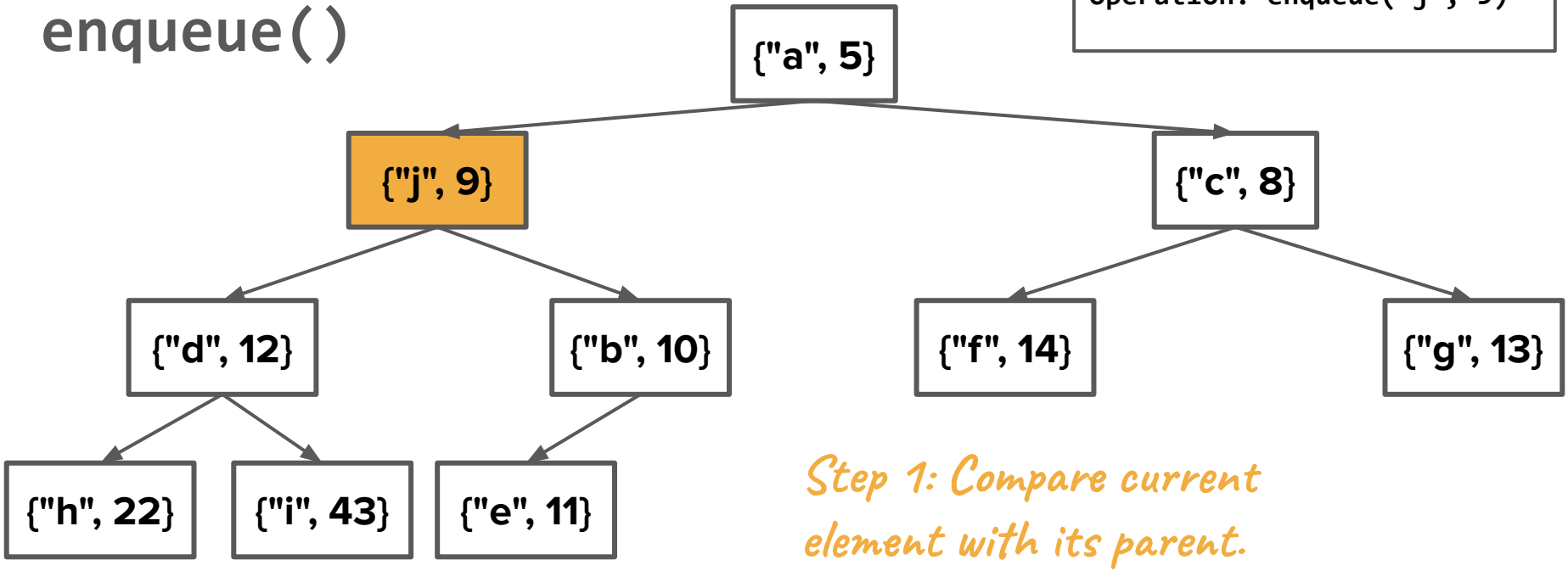
Operation: enqueue("j", 9)



{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

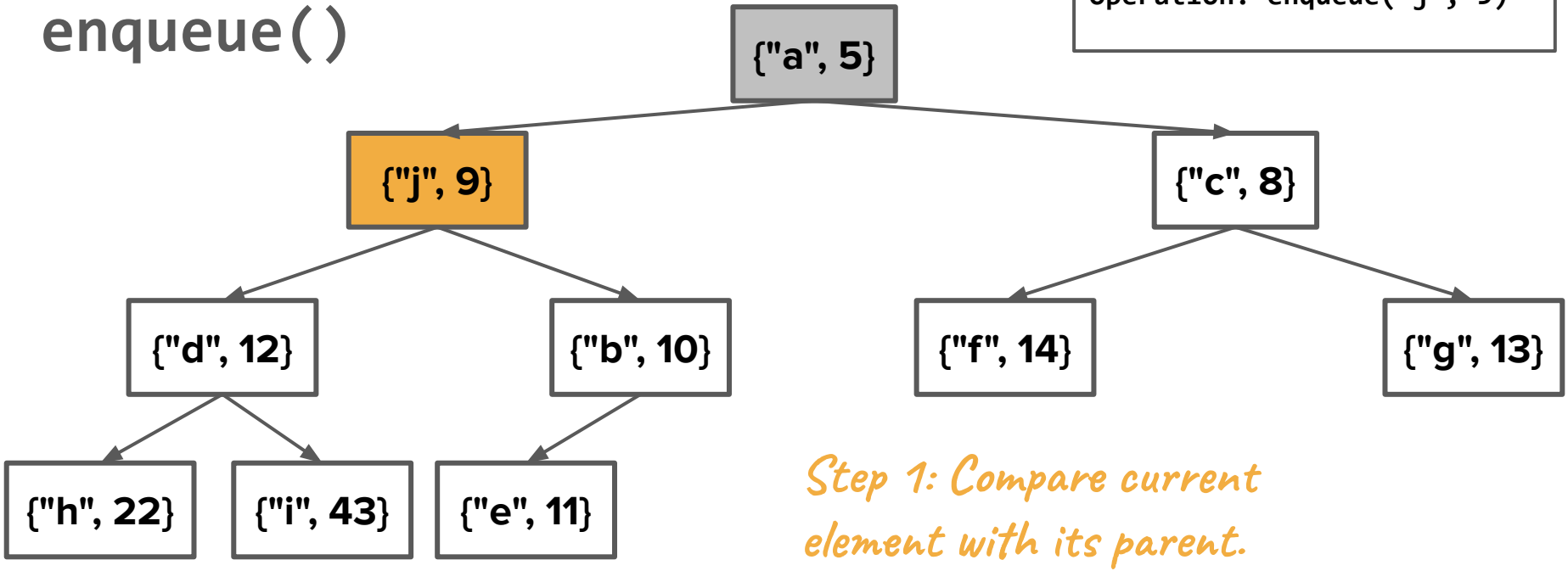


Step 1: Compare current element with its parent.

{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

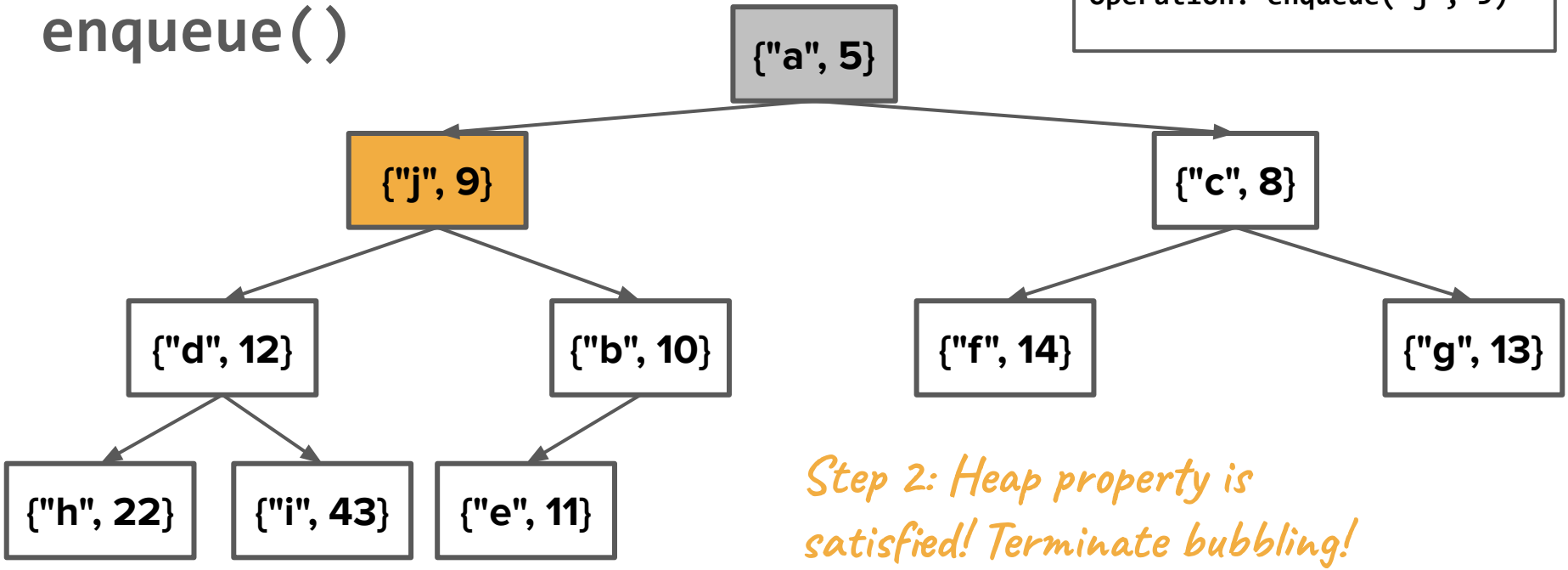


Step 1: Compare current element with its parent.

{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

Operation: enqueue("j", 9)

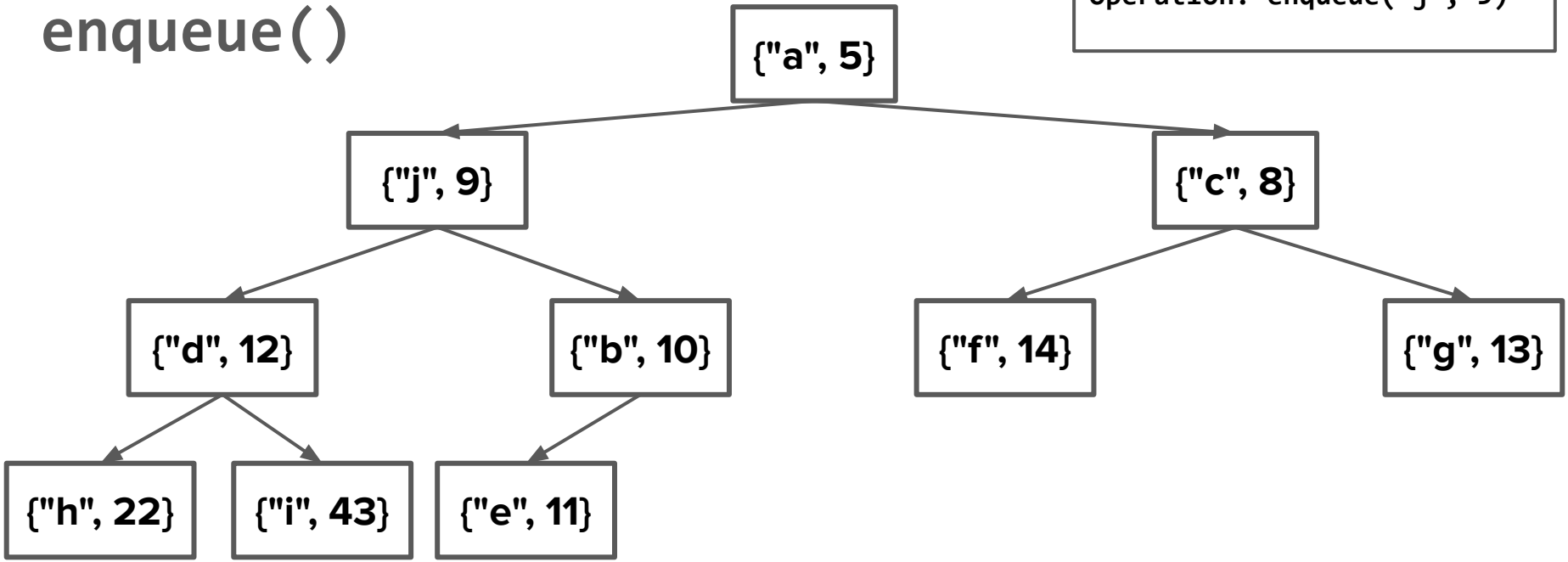


Step 2: Heap property is satisfied! Terminate bubbling!

{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

```
Operation: enqueue("j", 9)
```



{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 11}	?	?
0	1	2	3	4	5	6	7	8	9	10	11

`enqueue()`

- After our "bubble up" process completes, the heap is in a proper state again. Yay! We have now successfully inserted a new element into the heap.

- For a cool animation of this process across many **enqueue** operations, check out this cool online heap animation.

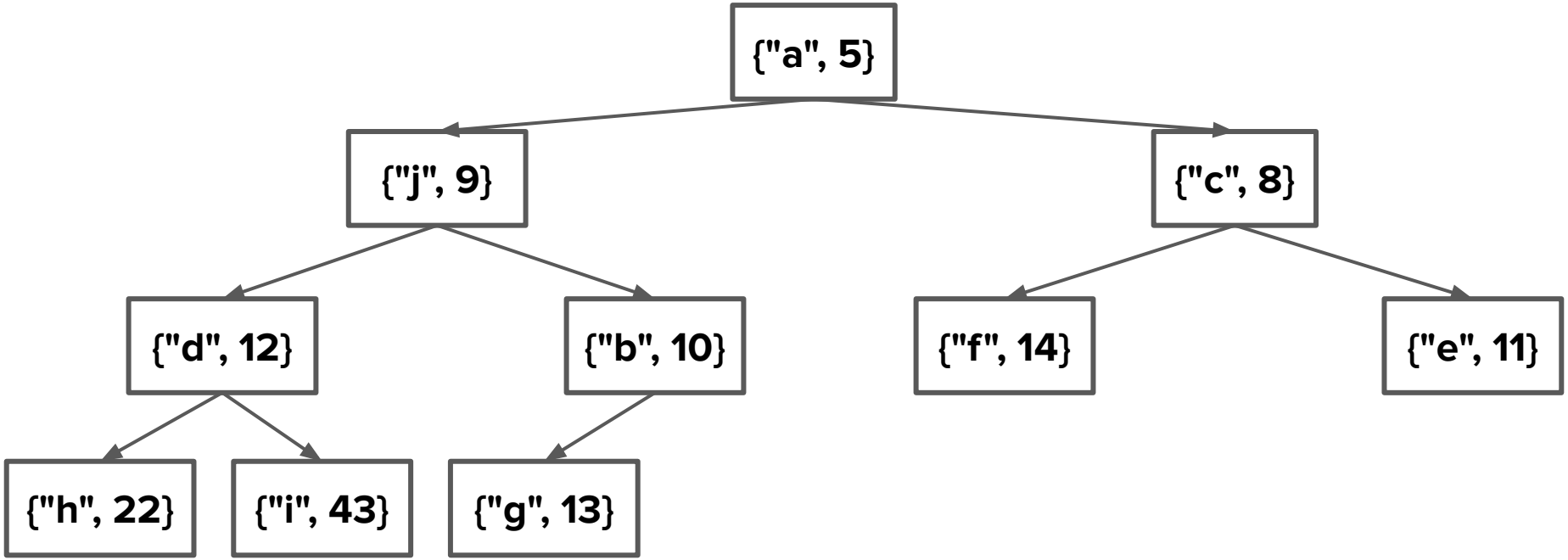
- What is the runtime complexity of the **enqueue** operation?
  - In the worst case scenario, we have to bubble up the new element all the way up to the root position.
  - Since there are **n** total elements, the tree will have **`log n`** levels, which means we would do **`log n`** comparisons and **`log n`** swaps along the way.
  - The overall complexity is **`O(log n)`**, which we know is blazingly fast! How cool!

`dequeue()`

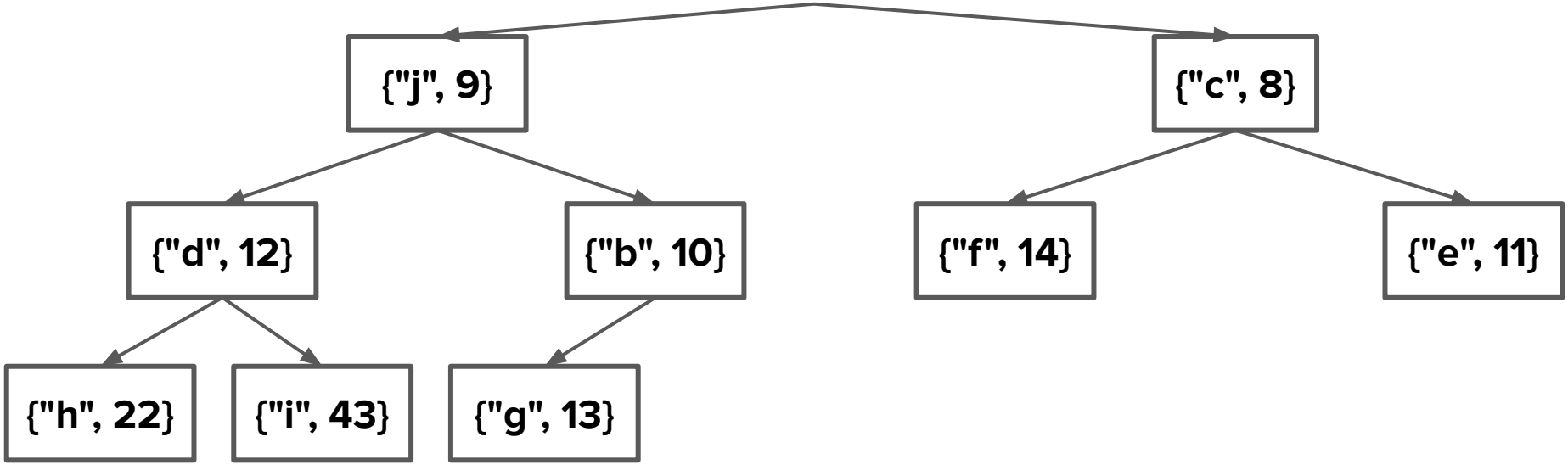
- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!

`dequeue()`

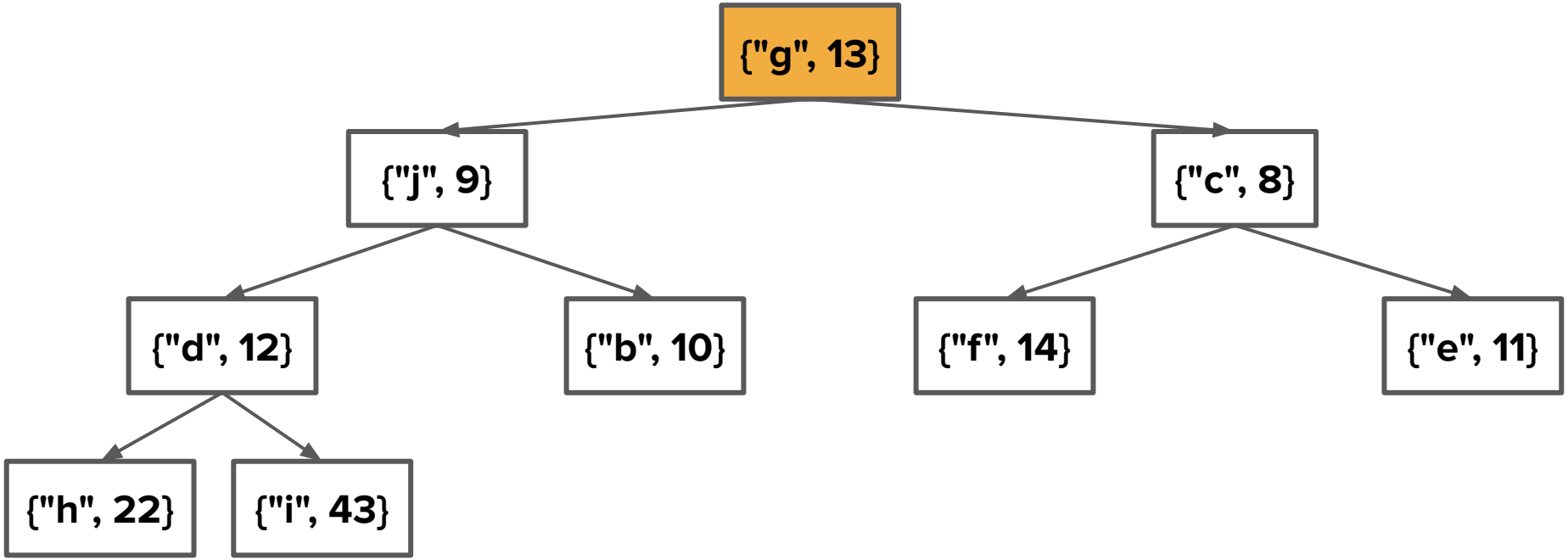
- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!



{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	{"g", 13}	?	?
0	1	2	3	4	5	6	7	8	9	10	11



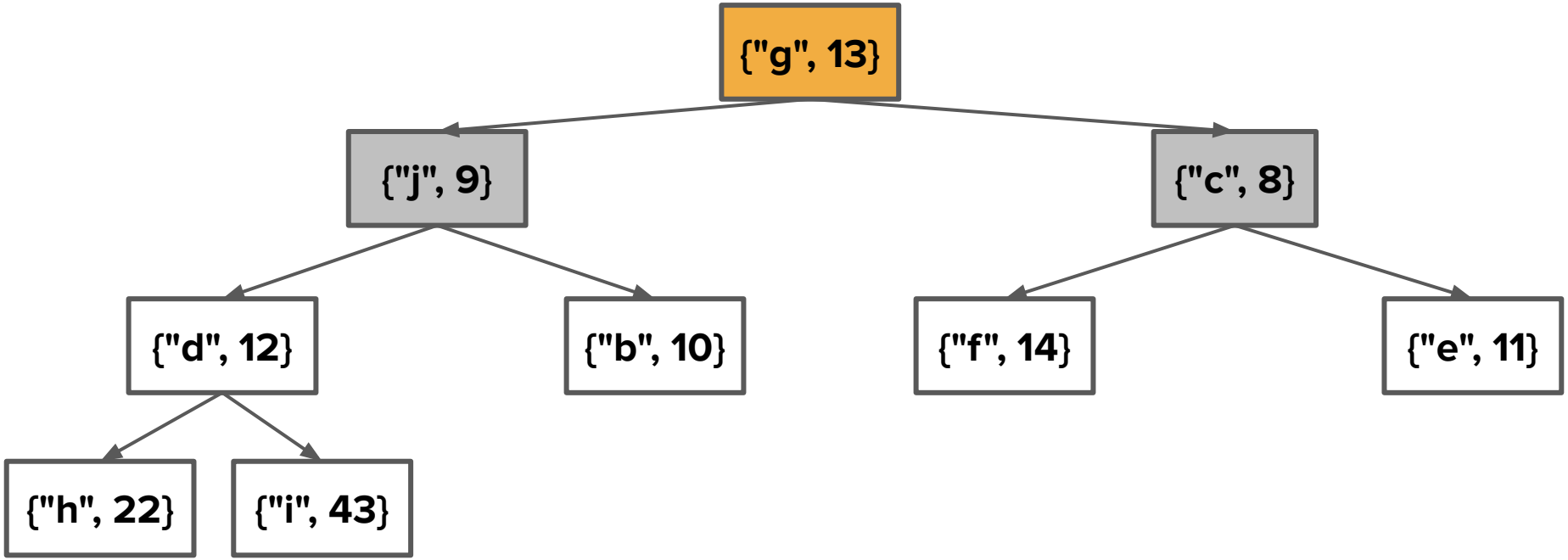
	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	{"g", 13}	?	?
0	1	2	3	4	5	6	7	8	9	10	11



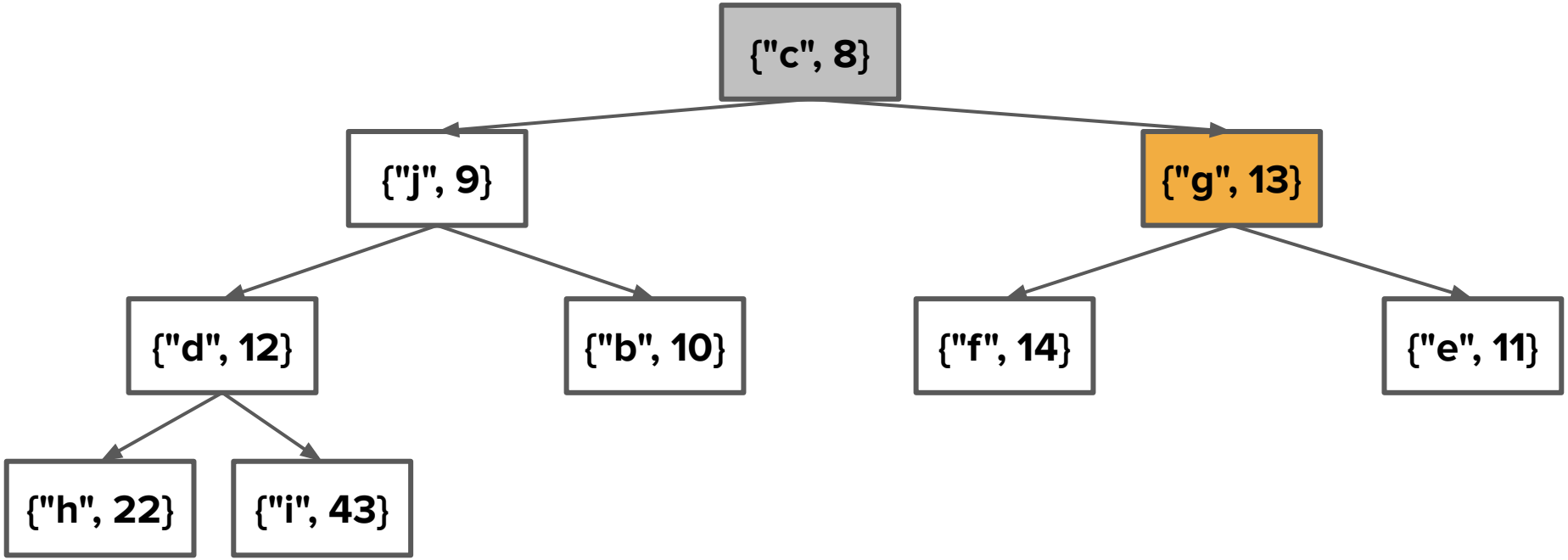
{"g", 13}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

dequeue()

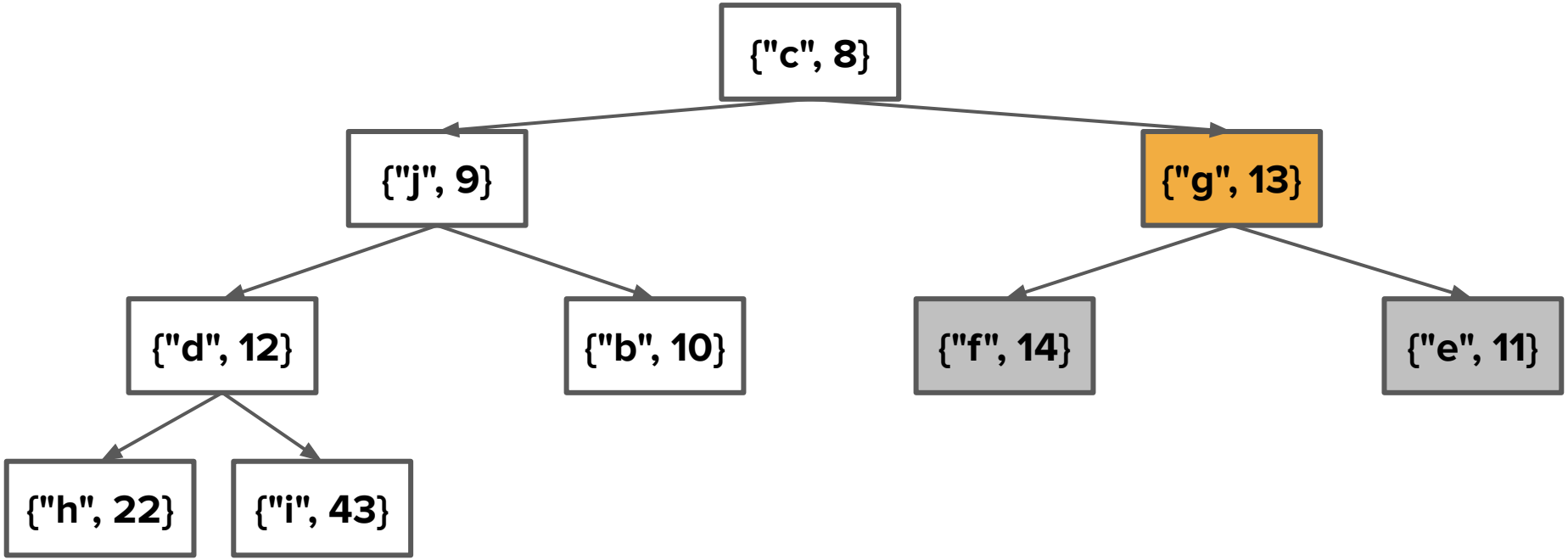
- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- **Bubble down** to regain the *heap property*!
  - Compare the moved element to its new children.
    - If one of the two children is smaller, swap with that child.
    - If both of the children are smaller, swap with the one that’s smaller.
  - Repeat until you no longer bubble down or there are no more children to compare against.



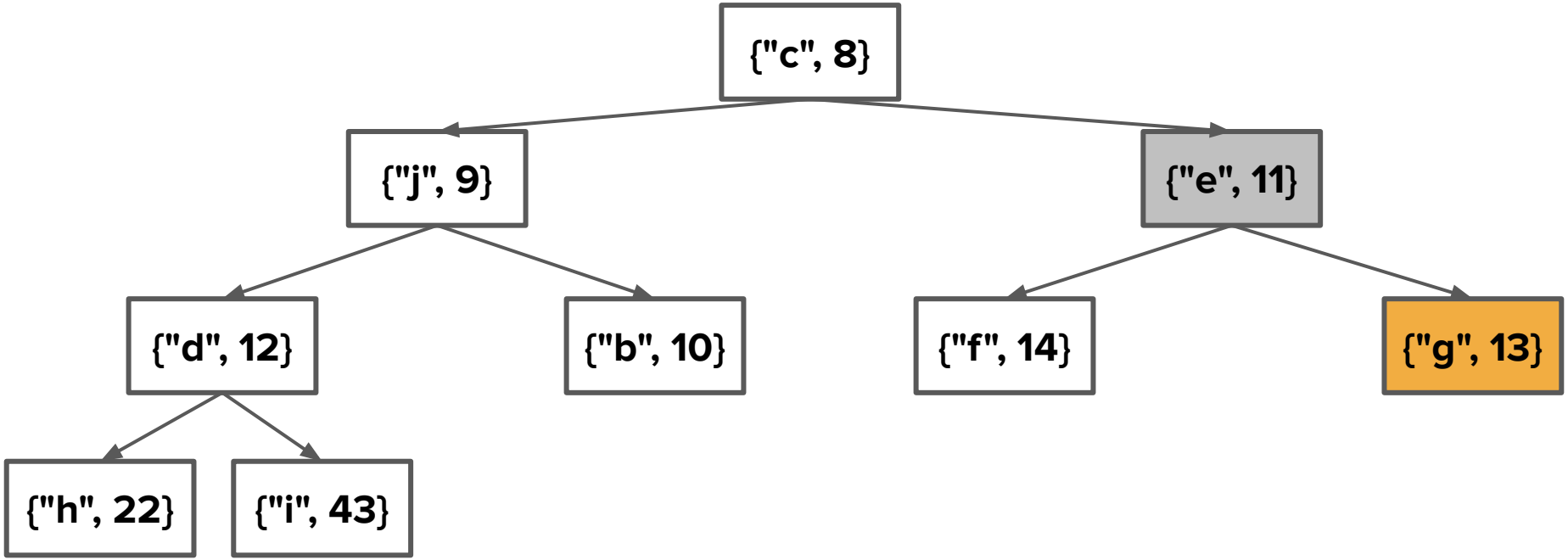
{"g", 13}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11



{"c", 8}	{"j", 9}	{"g", 13}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

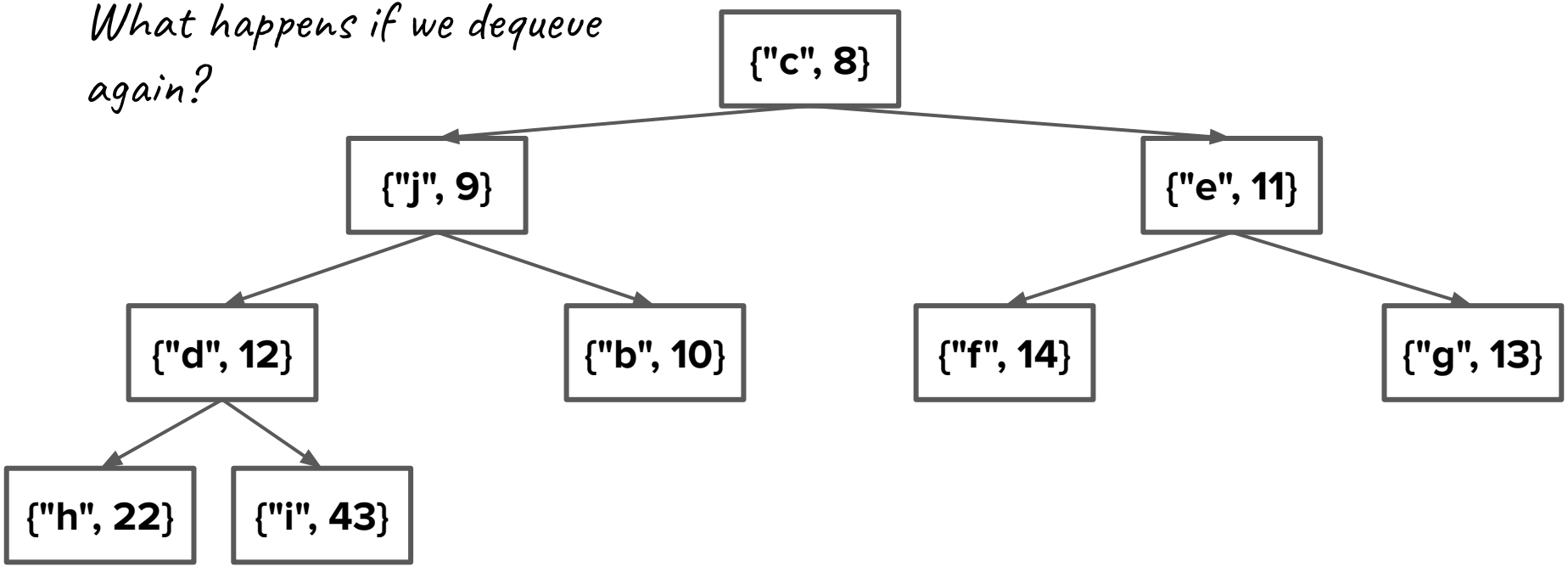


{"c", 8}	{"j", 9}	{"g", 13}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11



{"c", 8}	{"j", 9}	{"e", 11}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

What happens if we dequeue again?



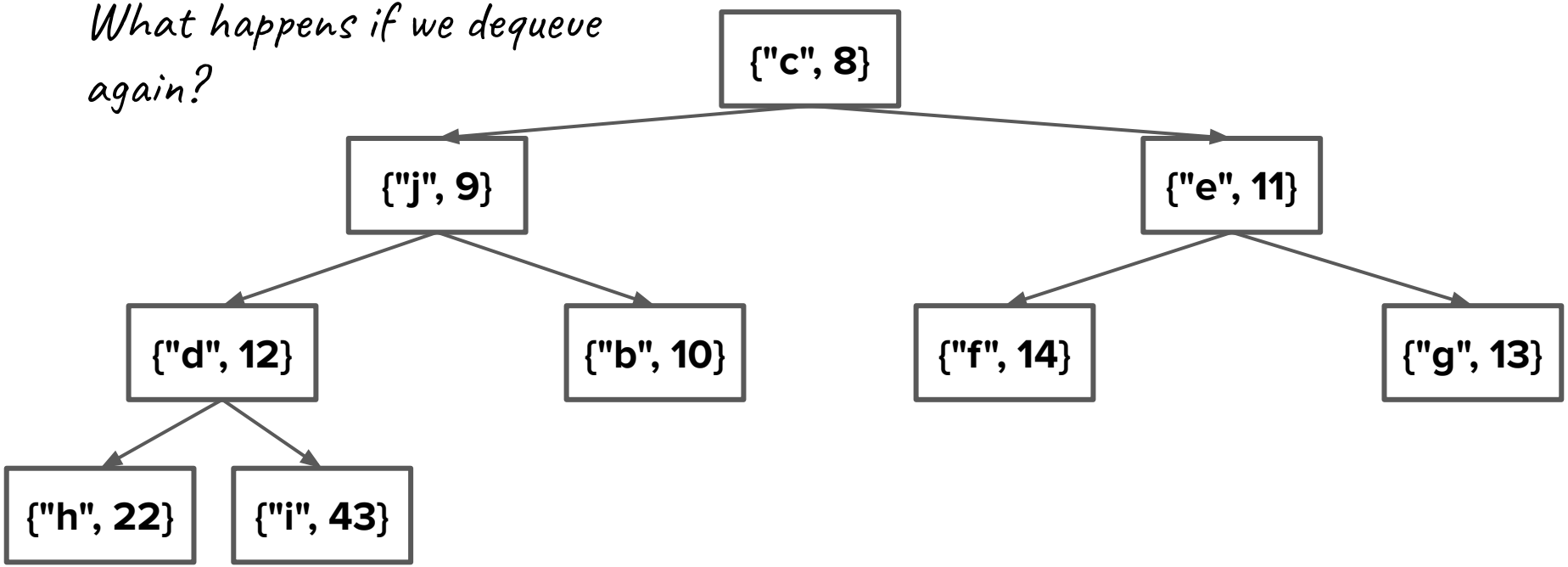
{"c", 8}	{"j", 9}	{"e", 11}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

Attendance ticket:

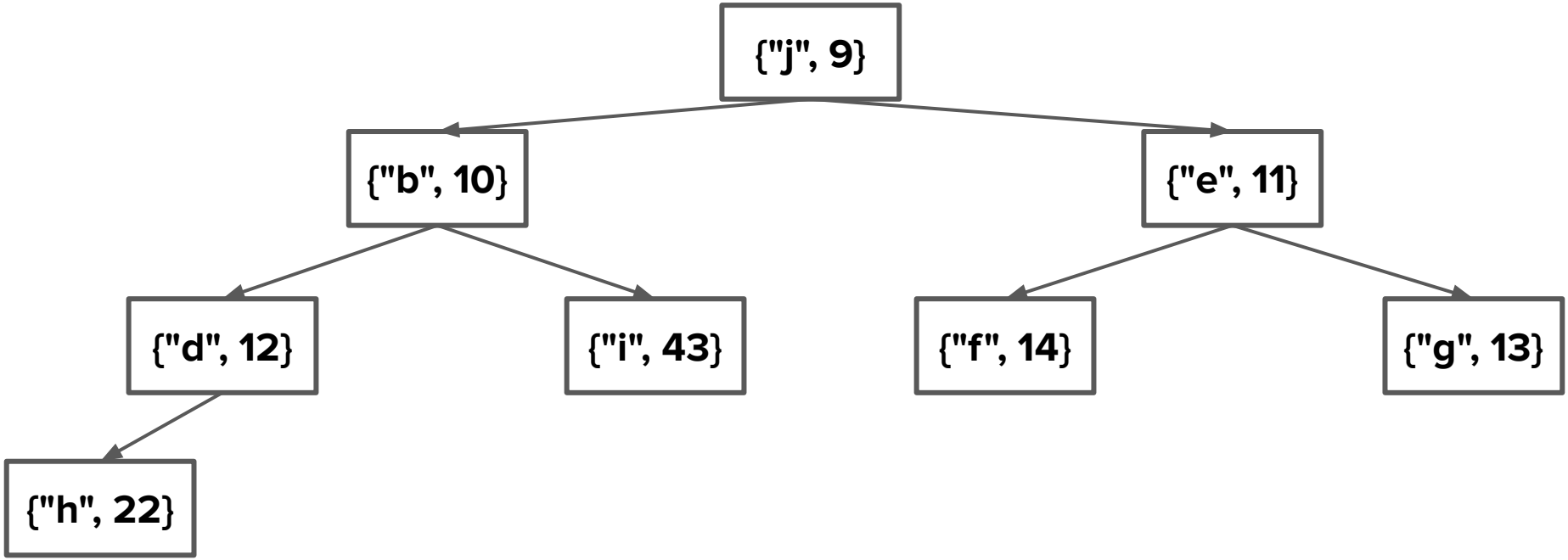
https://tinyurl.com/dequeueC

Please don’t send this link to students who are not here. It’s on your honor!

What happens if we dequeue again?



{"c", 8}	{"j", 9}	{"e", 11}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11



{"j", 9}	{"b", 10}	{"e", 11}	{"d", 12}	{"i", 43}	{"f", 14}	{"g", 13}	{"h", 22}	?	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

dequeue()

- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!
- **O(log n)**: At worst, you do one comparison at each level of the tree.

dequeue()

- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!
- **O(log n)**: At worst, you do one comparison at each level of the tree.

We have a data structure with only `O(log n)` *and* `O(1)` *operations!*

Summary

- **Priority queues** are queues ordered by **priority** of their elements, where the **highest priority** elements get dequeued first.

- **Binary heaps** are a good way of organizing data when creating a priority queue.
    - Use a min-heap when a smaller number = higher priority (what you'll use on the assignment) and a max-heap when a larger number = higher priority.

- There can be multiple ways to implement the same abstraction! For both ways of implementing our priority queues, we'll use **arrays** for data storage.

Levels of abstraction

What is the interface for the user?
(Priority Queue)

How is our data organized?
(sorted array, binary heap)

What stores our data?
(arrays)

Human Prioritization Algorithms

How do we calculate priorities?

Values in technology

- Design decisions encode values that express what we care about.
- These values can reveal our assumptions about the world and the people who will be interacting with our design and benefiting from it.
- Despite the best intentions, sometimes design decisions have unintended consequences that evoke different values than those of the original creators.

In the case of priority queues, the "priorities" themselves represent a very explicit value system!

A Case Study

Los Angeles County Coordinated Entry System (CES)

An electronic registry of unhoused persons who are applying or have applied to housing support programs offered by Los Angeles County.

How does it work?

???	Ranking	Matching
	Algorithm uses personal data to **assign a number from 1-17**, least vulnerable tomost vulnerable.	Risk score is used to prioritize and assign housing and housing related services.

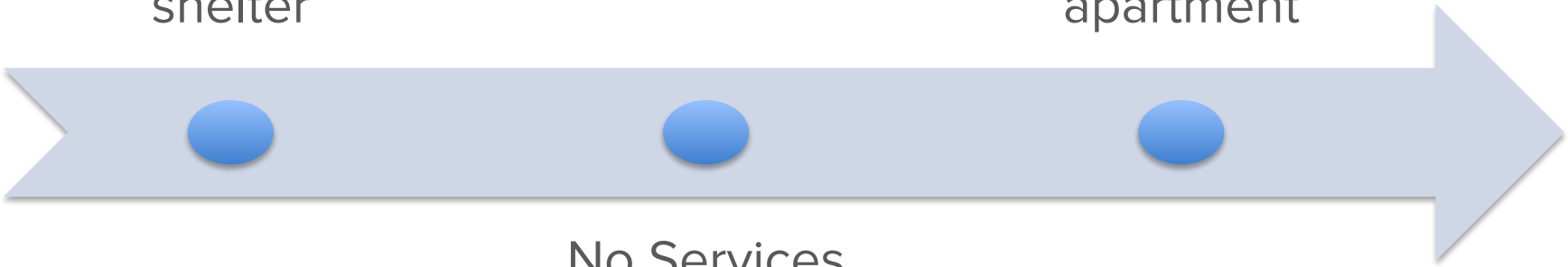
How does it work?

Data gathering	Ranking	Matching
Unhoused person provides (very) personal information including name, DOB, immigration status, current & past mental health, sexual activity, substance usage.	Algorithm uses personal data to assign a number from 1-17, least vulnerable tomost vulnerable.	Risk score is used to prioritize and assign housing and housing related services.

[Simplified] CES Binning System

1-4: Least vulnerable → short-term shelter

14-17: most vulnerable → long term apartment



No Services Provided

Vulnerability assessment

- If no one intervenes
  - Death
  - Chronic homelessness
  - Use of costly social services

- Some criteria
  - Physical or mental illness
  - Disability
  - Addiction
  - Length of time unhoused

Values intended by the designers

Evidence-driven policy design

Efficient use of resources

Neutrality

Priority of the worst-off

Promoting autonomy

Positive outcomes

“I’m doing the matching and it’s very unbiased as far as our work because the computer tells me, based on a scoring system, which families are higher need than other families.”
- Worker Interviewed According to Need Podcast

The CES did improve matching between people and services!

Unintended consequences and values

- Data collection demands and risks
- Certain groups of people benefited more / less than others
- Resource allocation

Unintended consequences and values

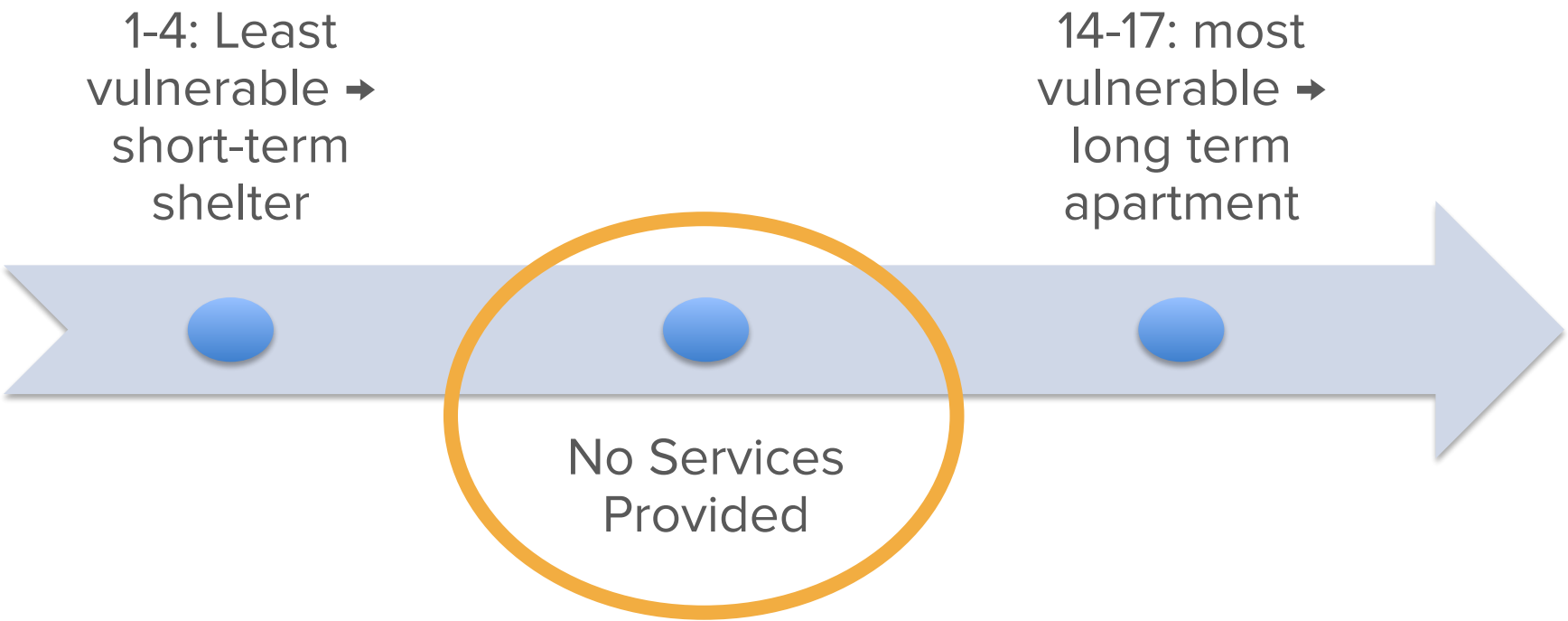
- Data collection demands and risks
  - The collection of data itself is demanding for the data subjects.
    - In addition to name, DOB, and demographic information, they are asked for very personal information, like immigration status, history of mental health and substance use.

  - Generating and storing a large pool of data about a vulnerable population itself incurs risks.
    - What if the data leaks or is hacked?
    - What if other agencies like ICE attempt to access collected data, such as immigration status + where the person is during the day?

Unintended consequences and values

- Data collection demands and risks
  - Survey data “expires” every six months (although they are not deleted from the system!)
  - This means the people in the middle have to provide the information every six months, re-taking the survey or interview.
  - People who are asked to fill out the survey multiple times with without being offered any services may become cynical and disengaged.

[Simplified] CES Binning System



Unintended consequences and values

- Certain groups of people benefited more / less than others
  - Because of economic disparities, it is easier for people of color to become unhoused.
  - But a higher proportion of white unhoused people met some of the criteria of high vulnerability. As a result, they scored higher on the ranking and were more likely to receive benefits.
  - The binning also creates a middle group of unhoused folks who never qualify for housing and have no other paths to services. The algorithm unintentionally created a new category of people seen as persistently unhouseable.

Unintended consequences and values

- Resource allocation
  - The CES has cost LA County $11 million so far, which is about $1,140 per person who has gotten housing through it.
  - An algorithmic prioritization system requires engineers to build it, social workers to collect data for it, and tech people to maintain it.
  - The CES did improve matching between people and services, but didn’t increase the total number of apartments, vouchers, or shelter beds available, and therefore didn’t decrease the number of people unhoused.

Unintended consequences and values

- Data collection demands and risks
- Certain groups of people benefited more / less than others
- Resource allocation

How would you evaluate this system?
What are the tradeoffs between the intended encoded values and the unintended encoded values?

Frameworks for evaluating automated systems

Virginia Eubanks' Two Questions for Automated Decision-Systems

1. "Does the tool [or algorithm] increase the self-determination and agency of the decision subjects?"

2. "Would the tool be tolerated if it was targeted at non-poor [housed, etc.] people?" (Eubanks 2018)

If the answer is no, Eubanks argues that we should reconsider the design or consider not building the system at all.

Autonomy

“Individual autonomy is … the capacity to be one’s own person, to live one’s life according to reasons and motives that are taken as one’s own and not the product of manipulative or distorting external forces, to be in this way independent …

to govern oneself, to be directed by considerations, desires, conditions, and characteristics that are not simply imposed externally upon one, but are part of what can somehow be considered one’s authentic self” (Christman 2020).

Autonomy

“Individual autonomy is … the capacity to be one’s own person, to live one’s life according to reasons and motives that are taken as one’s own and not the product of manipulative or distorting external forces, to be in this way independent …

to govern oneself, to be directed by considerations, desires, conditions, and characteristics that are not simply imposed externally upon one, but are part of what can somehow be considered one’s authentic self” (Christman 2020).

How might a prioritization algorithm diminish or enhance autonomy?

- Who gains access to services and who doesn’t
- Whose data is collected and what are the effects of that data surveillance
- Who gets to make the decisions about who/what is considered “high priority”

Embedded Ethics Case Study

Ethics case study written by Katie Creel, Nick Bowman, and Neel Kishnani.

Design analysis

You're working as a software engineer and have been contracted to build a priority queue for a client. The client's specification requests that the design support exactly three levels of priority: "high," "medium," and "low." Elements enqueued at the same priority level are to be processed in FIFO (first-in-first-out) order. Your manager proposes a design for a new **PQueue** class that has an internal implementation consisting of three **ordinary `Queue`s**: one for high priority elements, one for medium priority elements, and one for low priority elements. To dequeue elements, it would first drain the high priority queue, then medium, and then low.

> **Q15**. Consider the differences between this three-bin priority queue and the priority queue you implemented on your assignment. Which would be more efficient to insert elements into and why? More generally, what are the benefits and disadvantages of using the three-bin priority queue vs. a regular priority queue?

Your manager tasks you with implementing the proposed priority queue. Now, you have to decide the threshold between a low priority element, a medium priority element, and a high priority element.

> **Q16**. Describe a real-world system where a three-bin priority queue could be used. What factors would you use to distinguish between a low vs. medium vs. high priority element? What limitations might you need to consider when using a three-bin priority queue to represent this system?

Validity of rank-based systems

Say a college admissions department used a priority queue to rank their applicants. The admissions team decides on an algorithm to assign each applicant a weighted score, claiming that it takes into account the applicant's GPA, course load, extracurriculars, and personal statements. Each applicant's score would be a **`double`**, and a higher score would make them higher priority in the queue (more likely to be admitted). Once the admissions team builds up the priority queue, they take the top 500 applicants (based on their weighted score) and admit them.

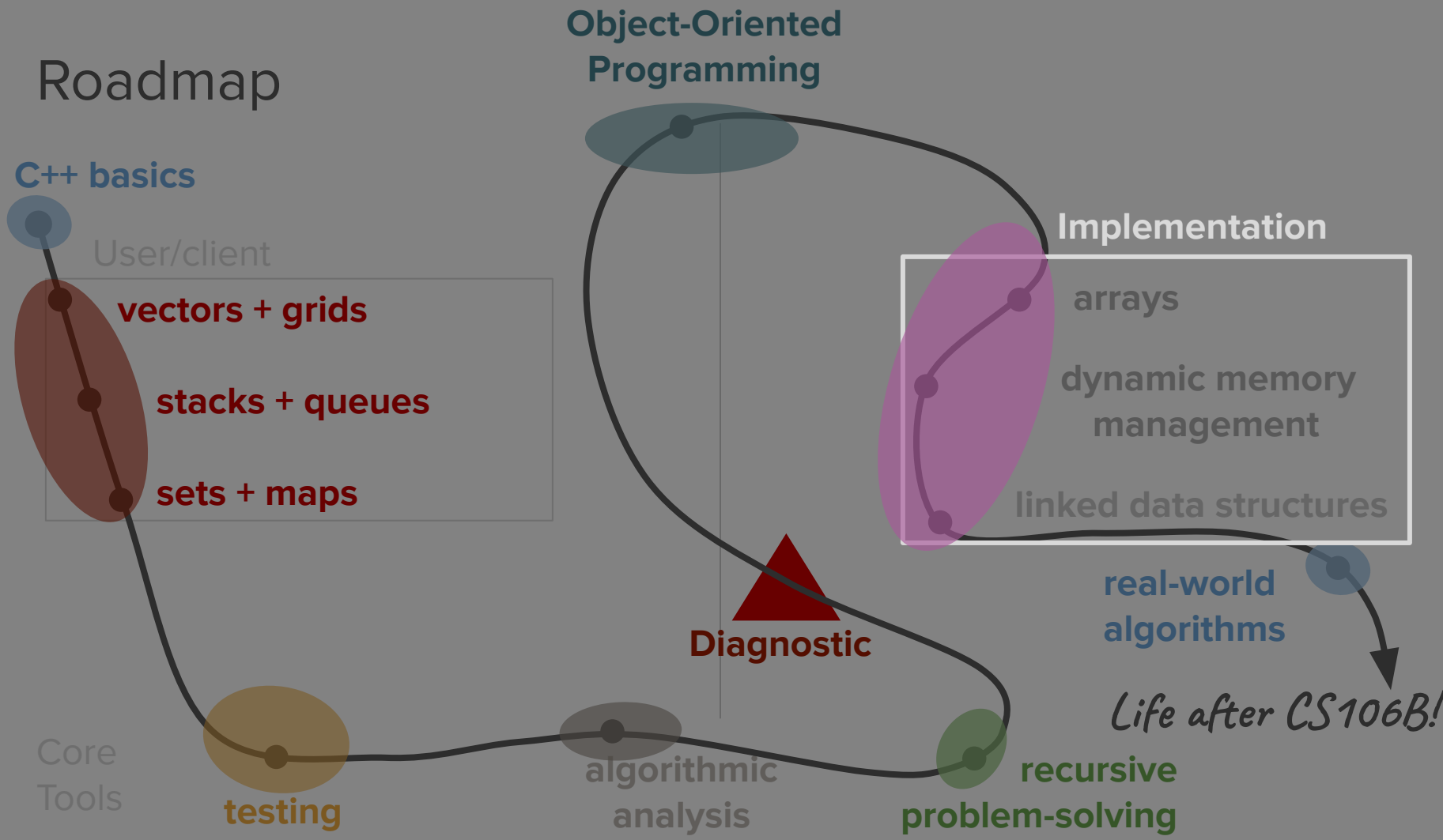
Values in technology

- Design decisions encode values that express what we care about.
- These values can reveal our assumptions about the world and the people who will be interacting with our design and benefiting from it.
- Despite the best intentions, sometimes design decisions have unintended consequences that evoke different values than those of the original creators.

In the case of priority queues, the "priorities" themselves represent a very explicit value system!

What’s next?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Life after CS106B!
Diagnostic
Core Tools
testing
algorithmic analysis
recursive problem-solving

Memory and Pointers