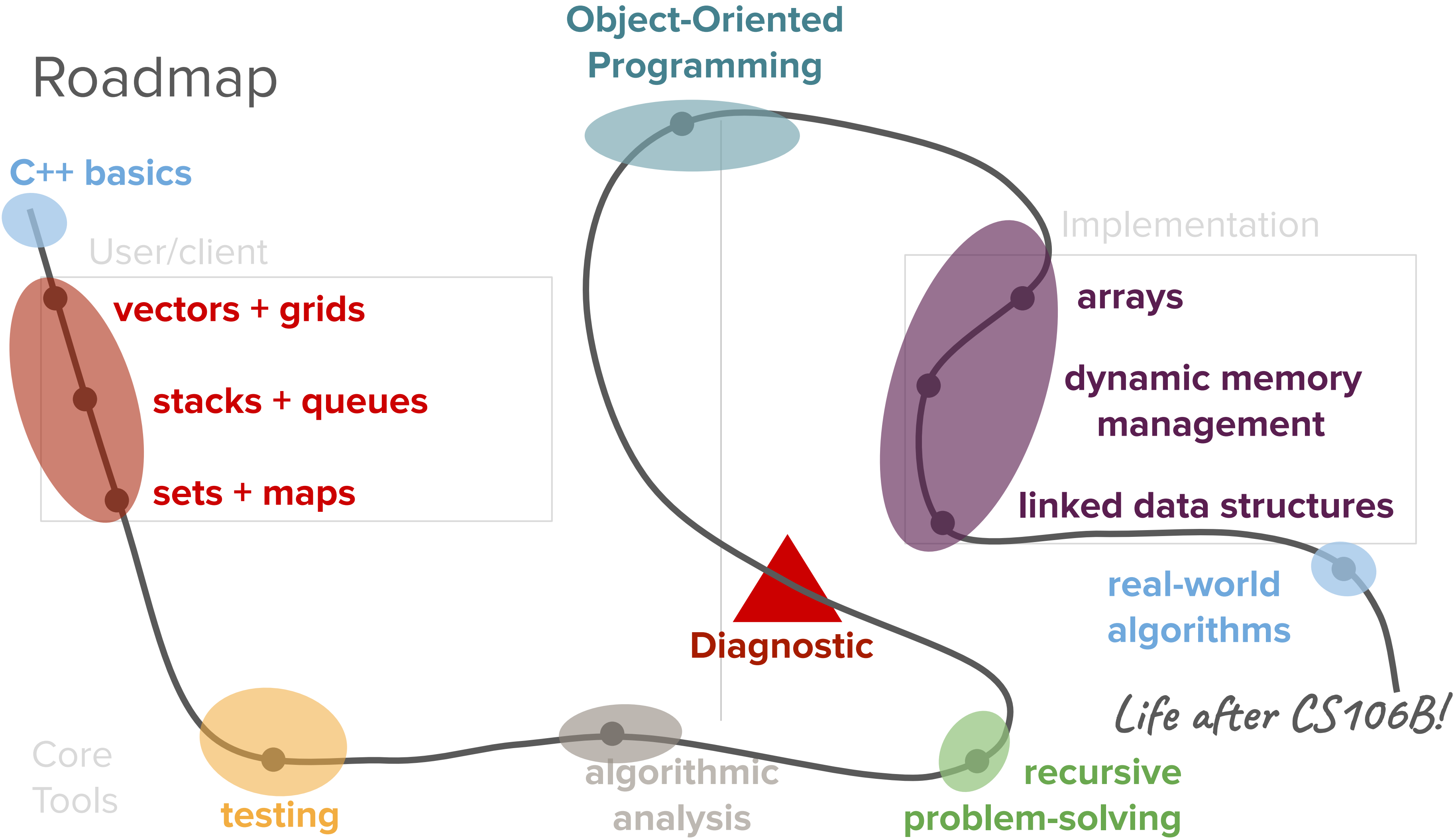
Priority Queues and Heaps

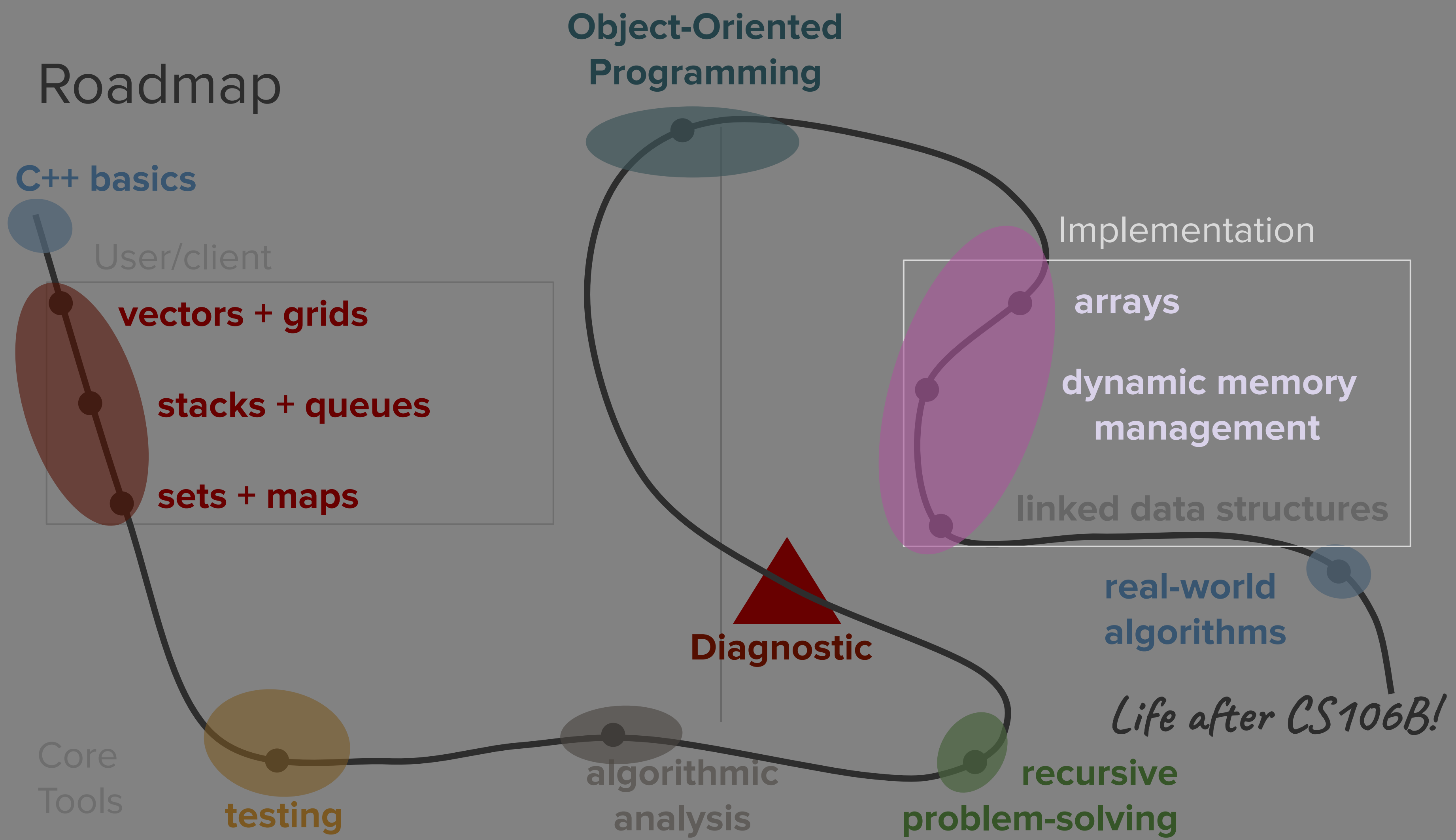
What pointer (aka piece of advice) would you give a prospective CS106B student about how to be successful in the class?

(put your answers in the chat)

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
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

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
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's question

How can we make use of multiple levels of abstraction in our data storage techniques to build better ADTs?

Today's topics

1. Review (`OurVector`)
2. Priority Queues
3. Binary Heaps

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
  - When we left off on Tuesday, `OurVector` was limited to **storing a fixed number of elements**. For now, if we run out space we just throw an error.

`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;
};
```

`OurVector` Member Variables

- `int* elements;`
    - A pointer to an array of integers, which will act as our underlying data storage mechanism.

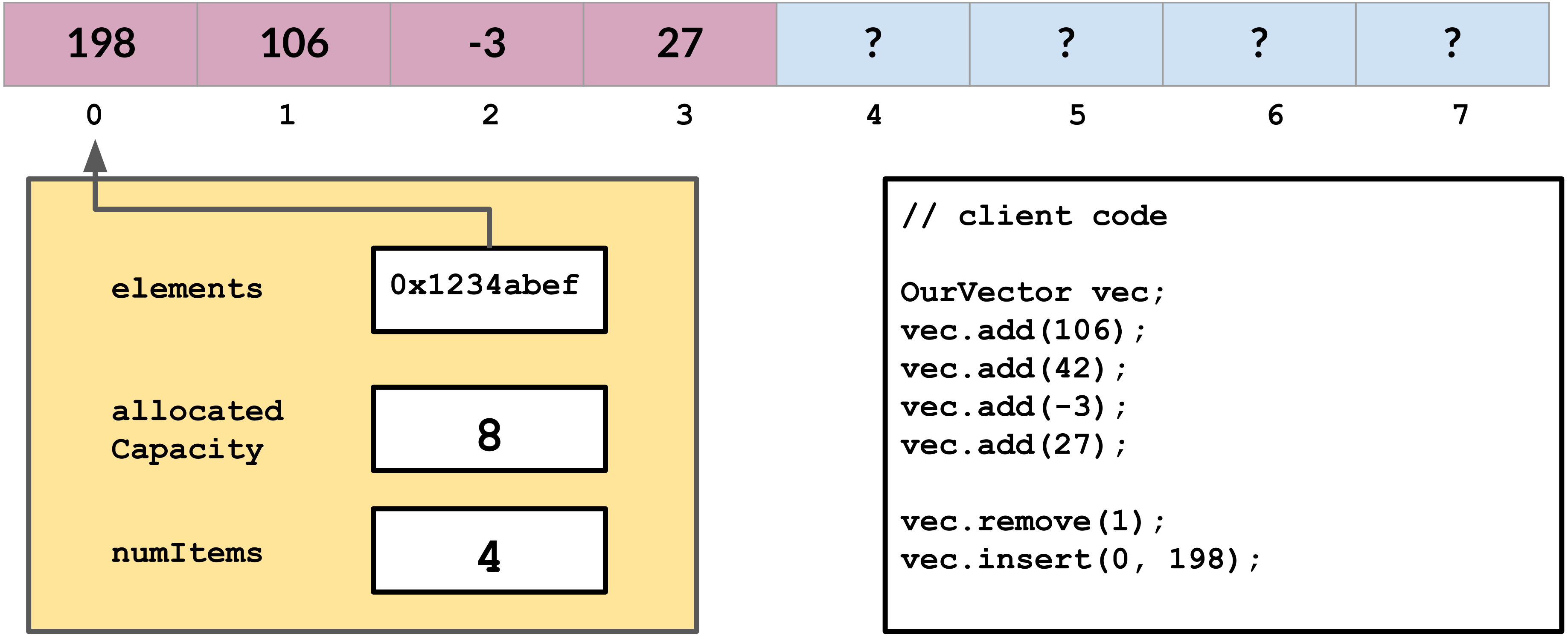
- `int allocatedCapacity;`
    - An integer that stores the size of the allocated elements array. Remember, arrays don't have any conception/knowledge of their own size, so we must manually track this!

- `int numItems;`
    - An integer that stores the number of elements currently stored in the vector.

Review: Constructors and Destructors

- The **constructor** is the special method that gets called when a new instance of a class is declared. In this method, we initialize all of our member variables to the appropriate values, including allocating any necessary memory.
    - The constructor for a class named `ClassName` has signature `ClassName(args);`
- The **destructor** is the special method that gets called when an instance of a class goes out of scope and thus is destroyed. In this method, we most often are responsible for freeing any dynamically allocated memory used by the instance.
    - The destructor for a class named `ClassName` has signature `~ClassName();`

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);
```

Running Out of Space

- Our current implementation very quickly runs out of space to store elements.

- What should we do when this happens?
  - Currently, we just throw an error. That doesn't seem quite right. What if all data structures we used were limited to hold only 8 items?
  - Instead, we need a way to **dynamically resize (grow)** our internal data storage mechanism.

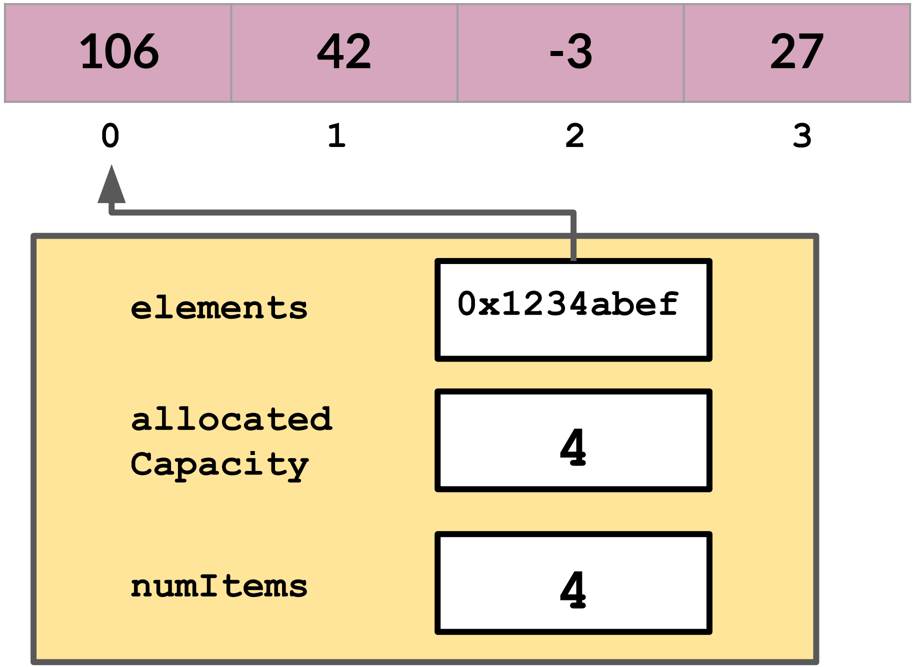
Dynamic Array Growth

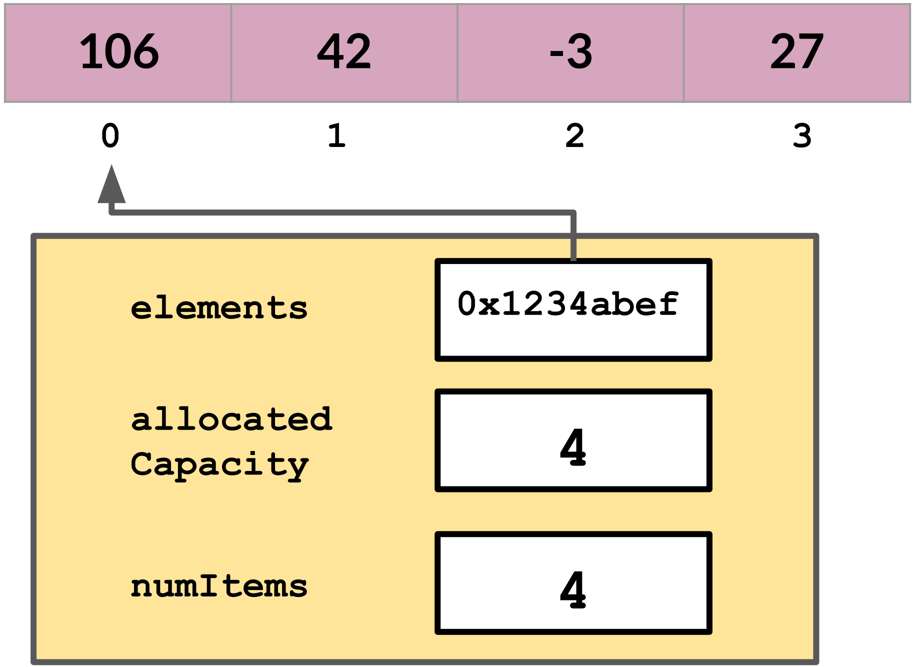
A Day in the Life of a Hermit Crab

- Hermit crabs are interesting animals. They live in scavenged shells that they find on the seafloor. Once in a shell, this is their lifestyle (with a bit of poetic license):
    - Grow until they have outgrown their current shell. Then, follow these 5 steps.
        - Find another, larger shell.
        - Move all their stuff into the new shell.
        - Leave the old shell on the seafloor.
        - Update their address with the Hermit Crab Postal Service.
        - Make note of their new shell's spacious capacity by posting on Hermit Crab Instagram.
- While this is purposefully a bit of a silly analogy, this process models almost exactly what we need to do in order to dynamically resize our internal data storage mechanism.

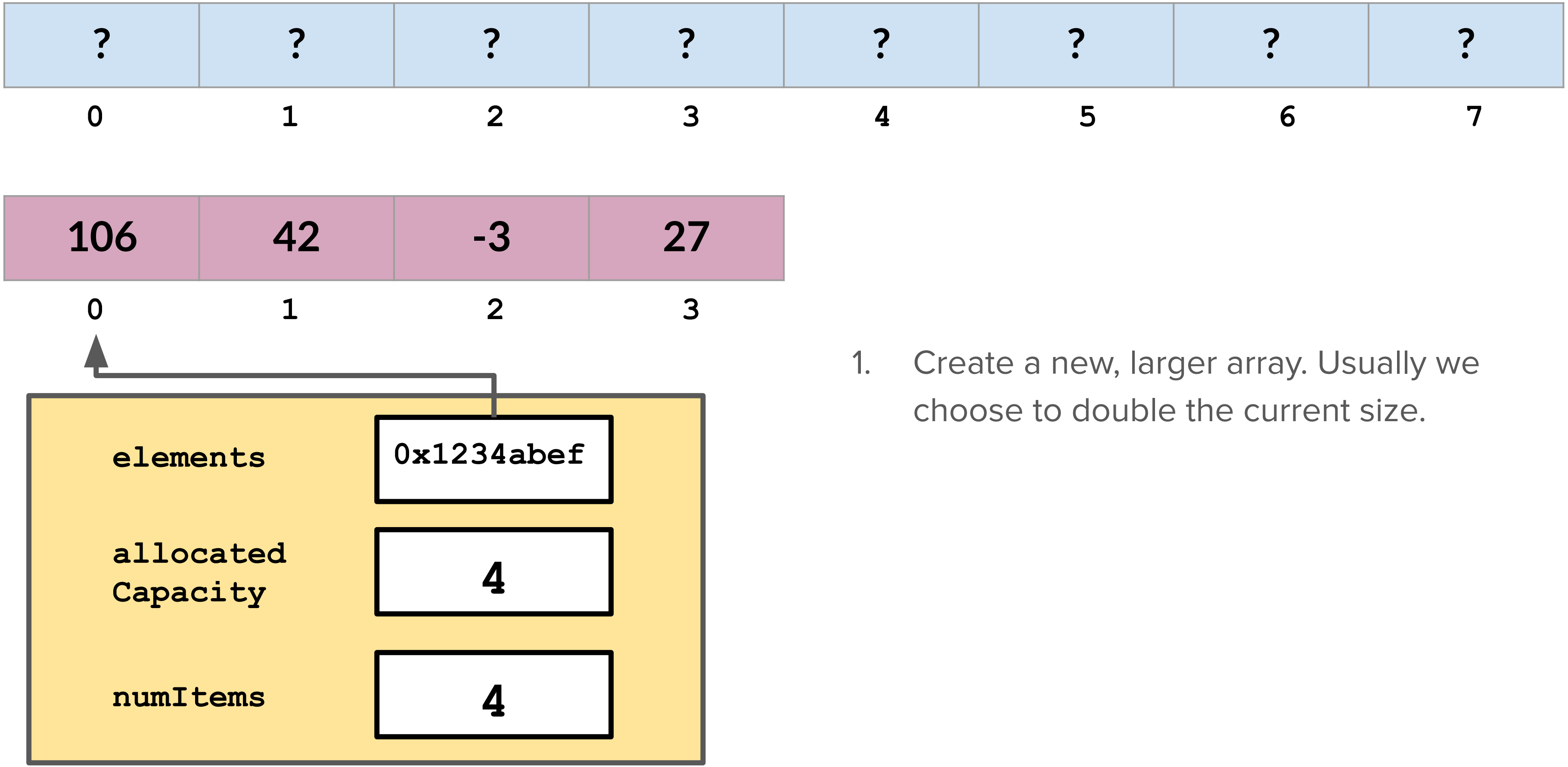
A Day in the Life of a Growable Array

- In essence, when we run out of space in our array, we want to allocate a new array that is bigger than our old array so we can store the new data and keep growing. These "growable arrays" follow a five-step expansion that mirrors the hermit crab model (with poetic license).
  - Grow the array until we run out of space (how can we tell if we've run out of space?)
    - Create a new, larger array. Usually we choose to **double** the current size.
    - Copy the old array elements to the new array.
    - Delete (free) the old array.
    - Point the old array variable to the new array.
    - Update the associated capacity variable for the array.

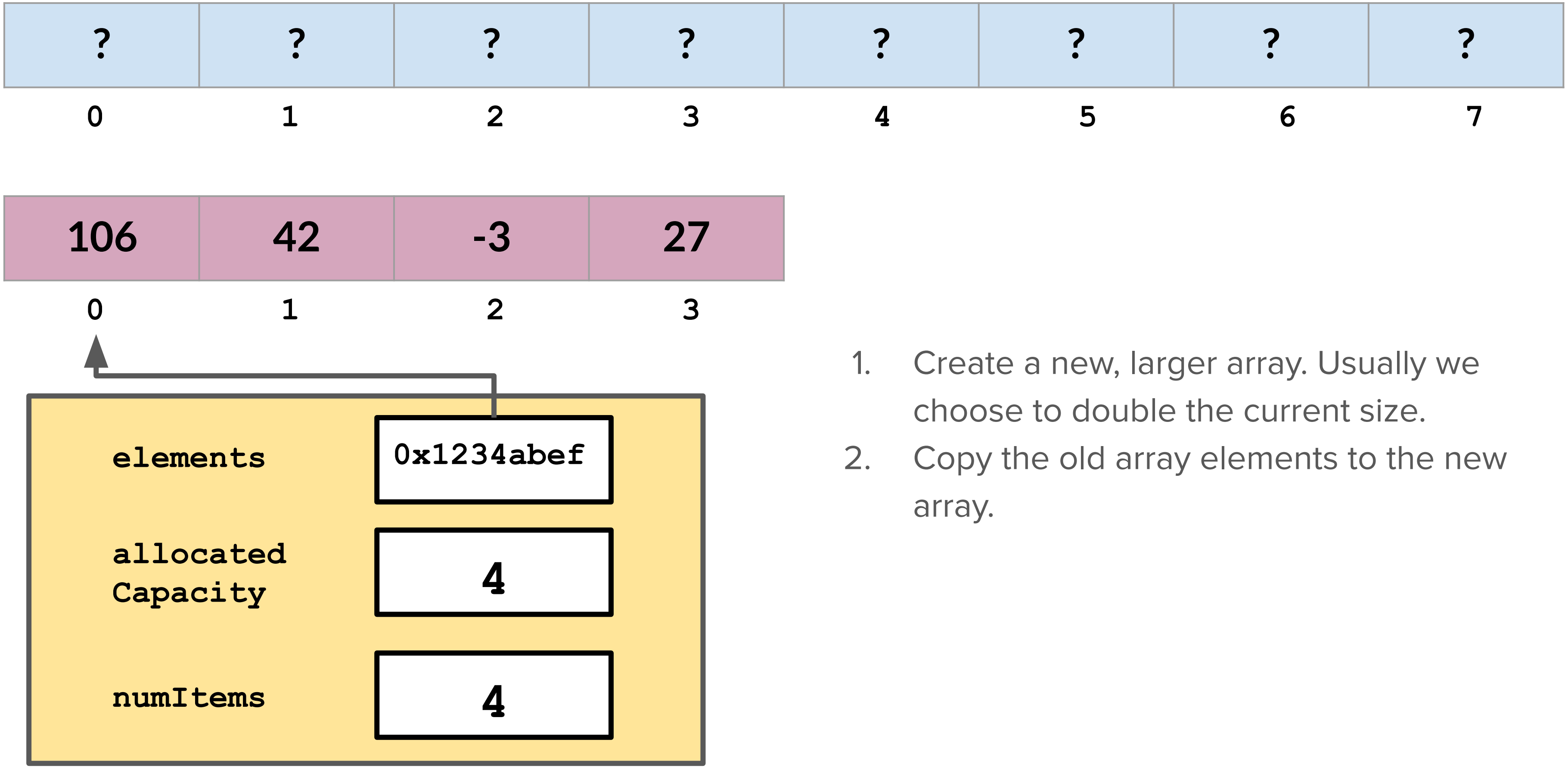

106
42
-3
27
0
1
2
3
elements
0x1234abef
allocated
Capacity
4
numItems
4



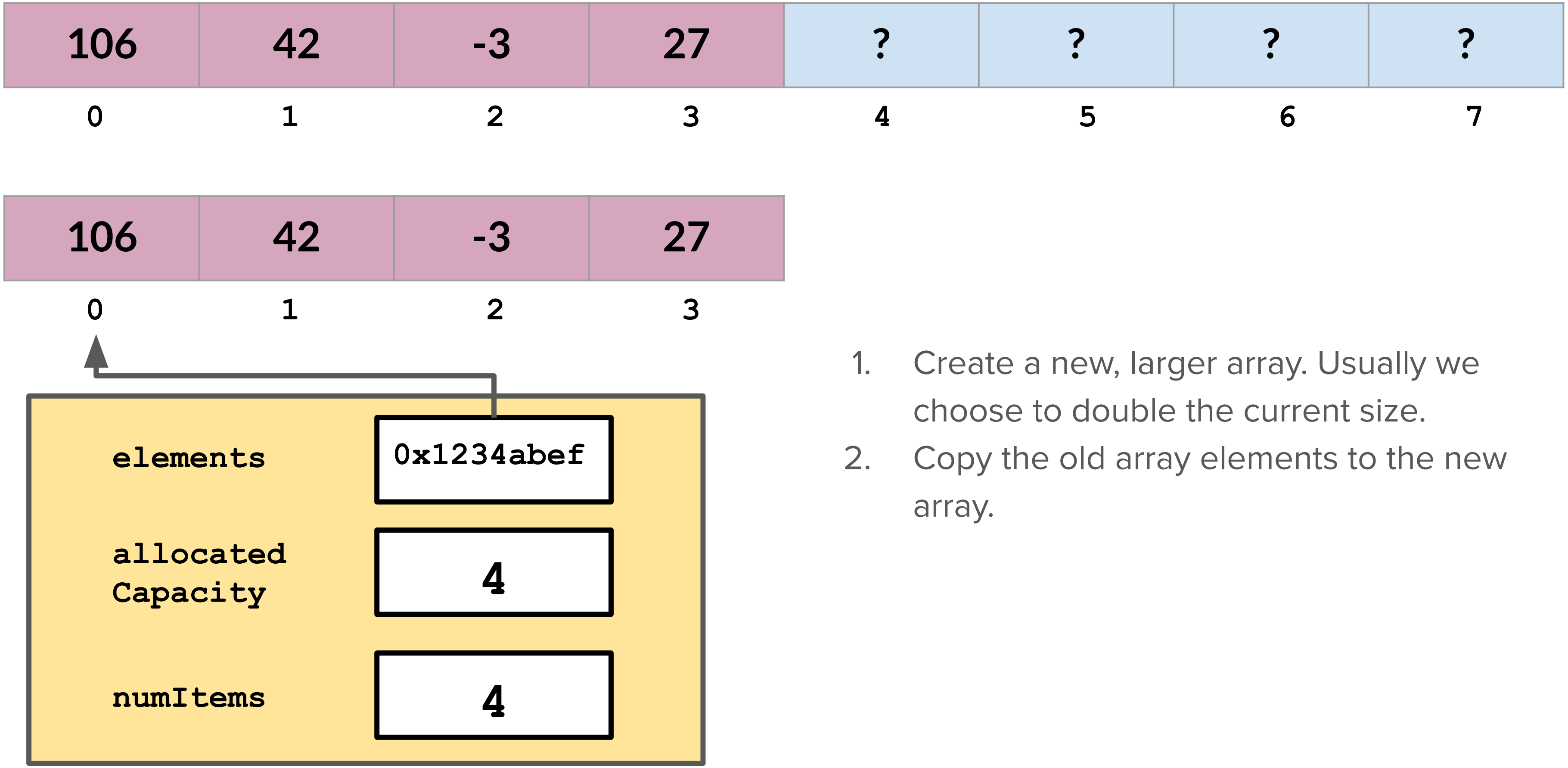
1. Create a new, larger array. Usually we choose to double the current size.



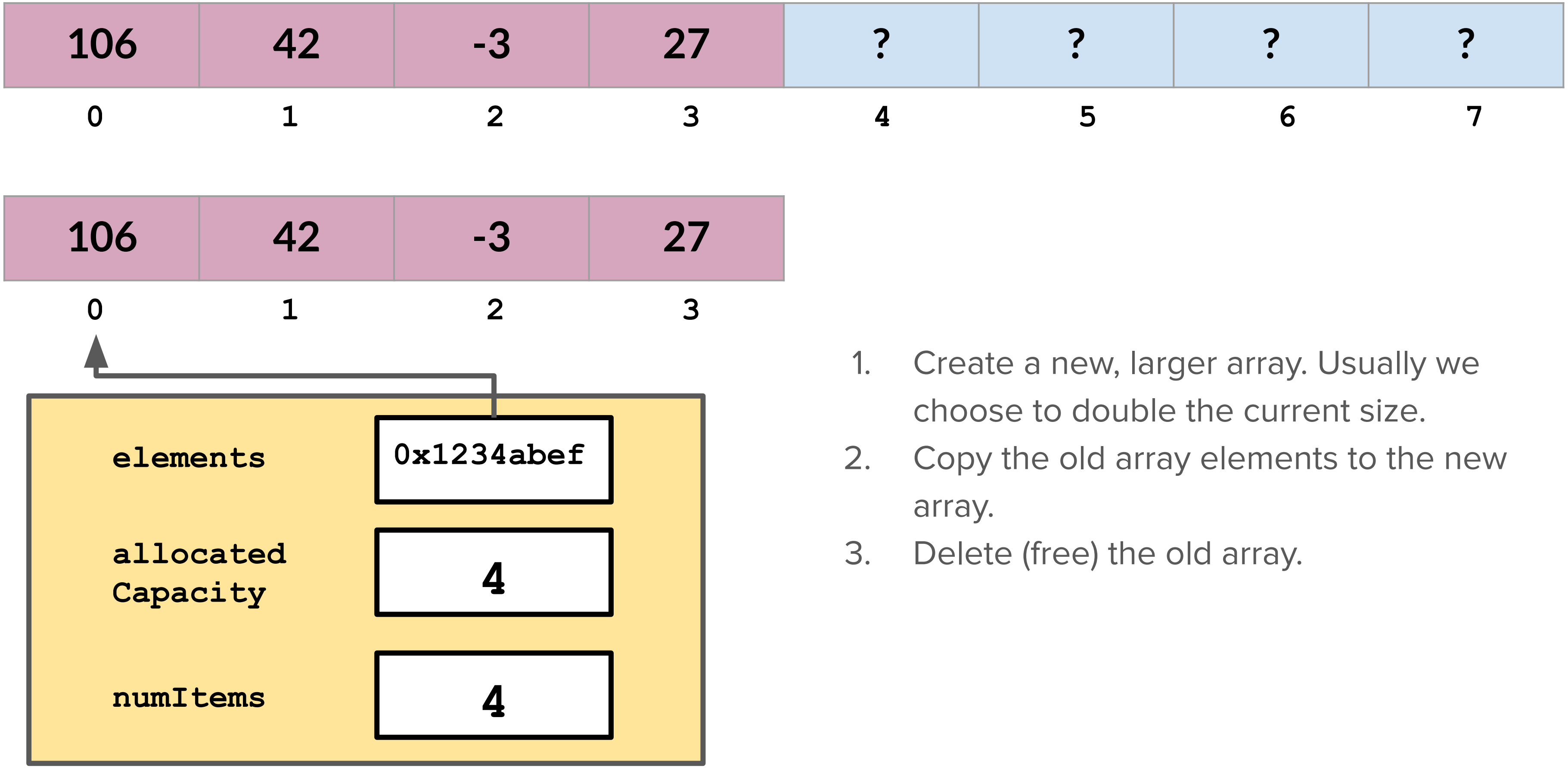
1. Create a new, larger array. Usually we choose to double the current size.



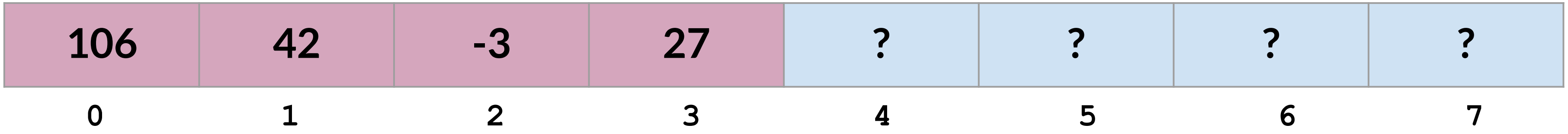
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.

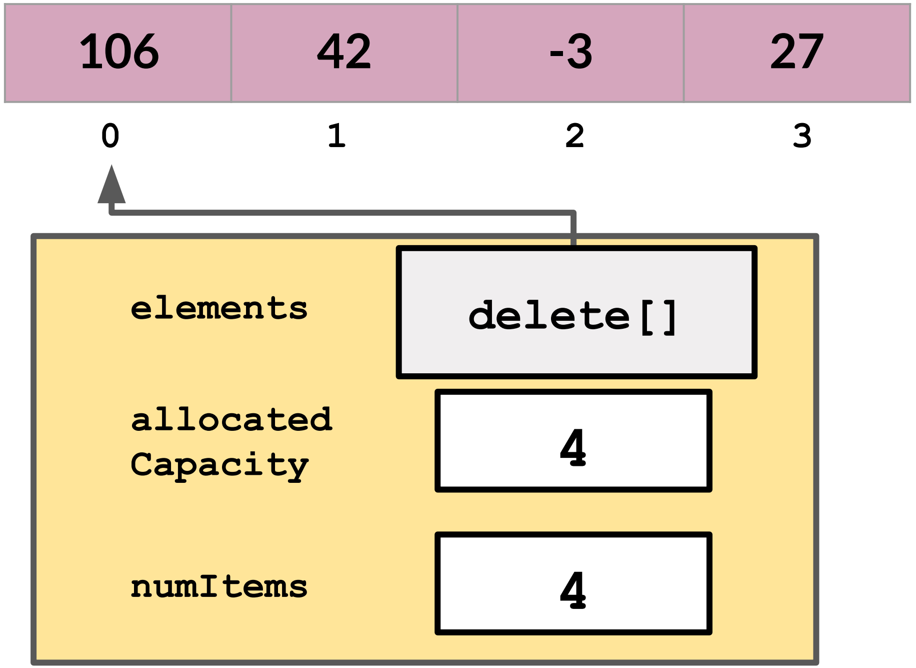


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.

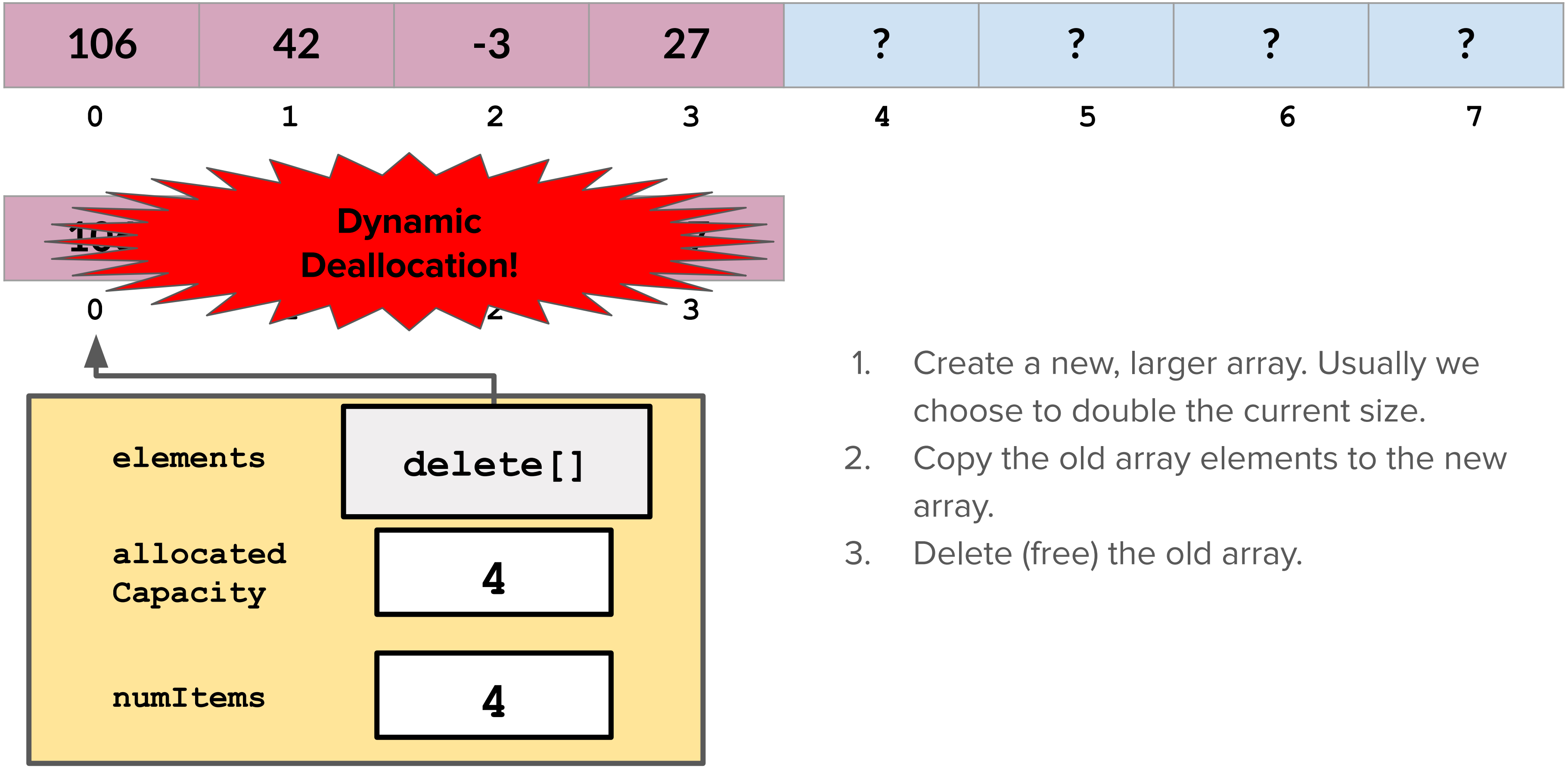


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.

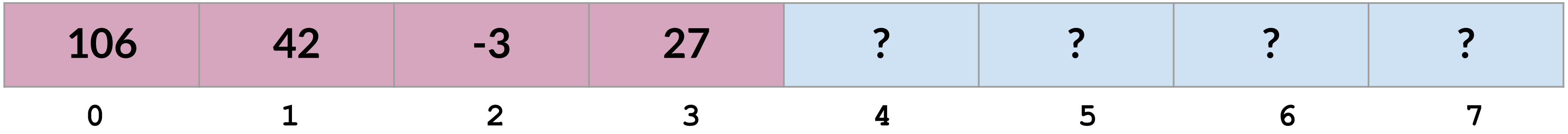


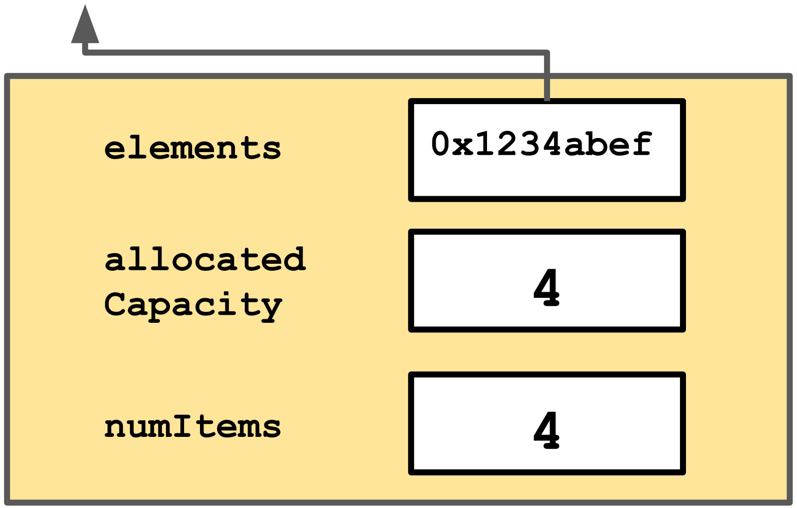


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.

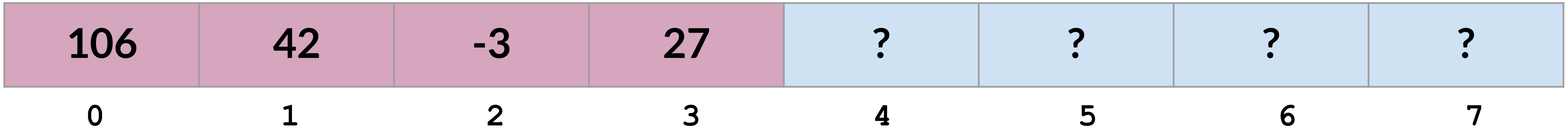


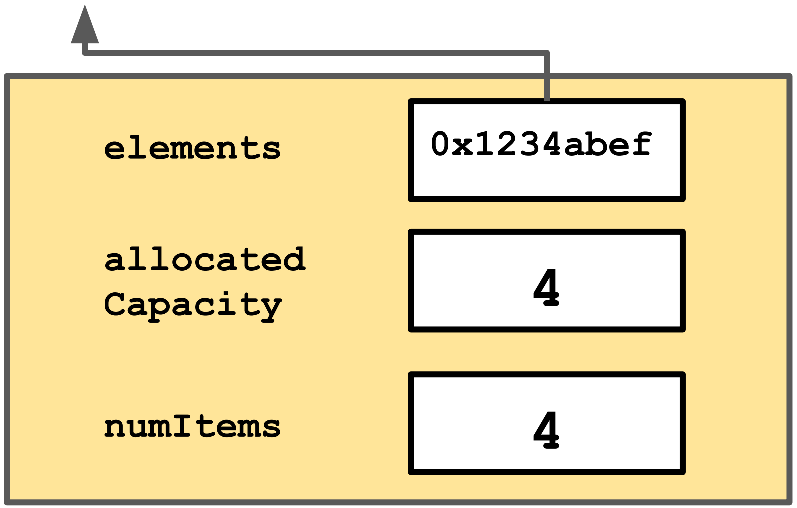
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.



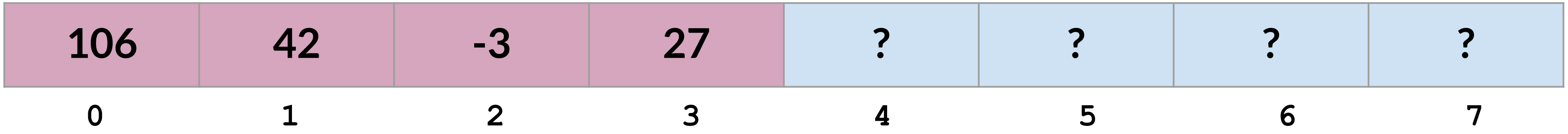


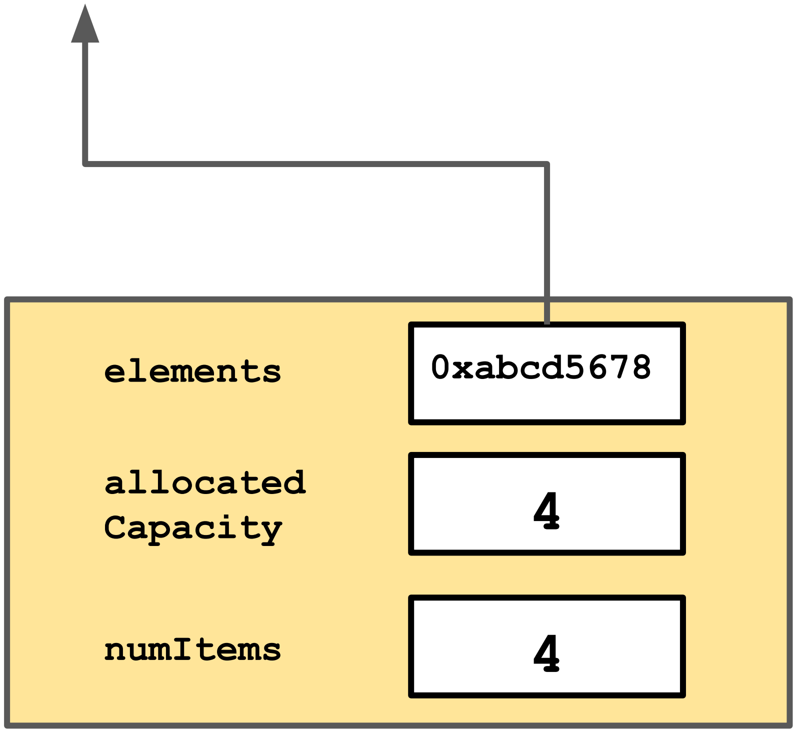
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.



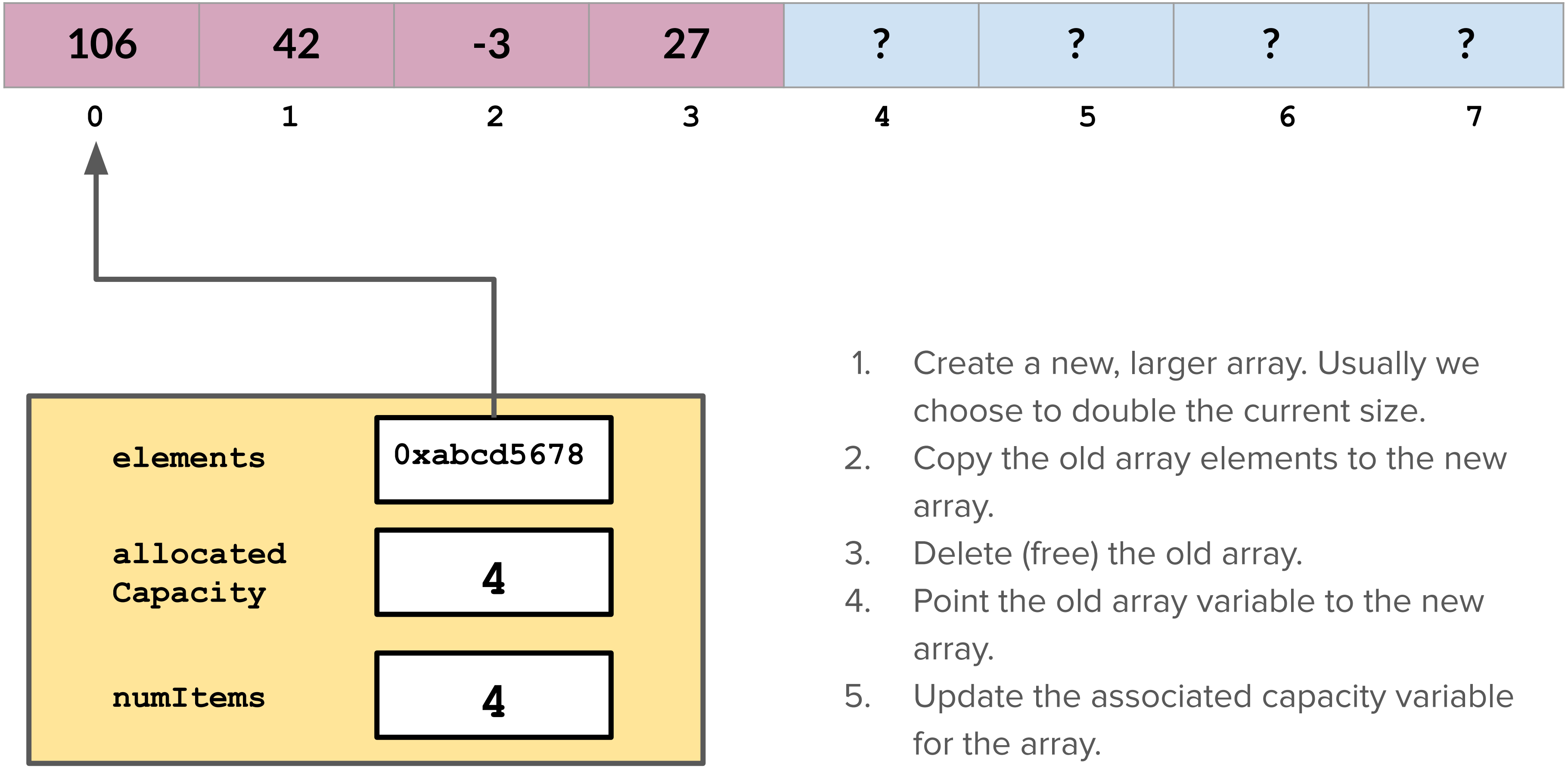


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.
4. Point the old array variable to the new array.

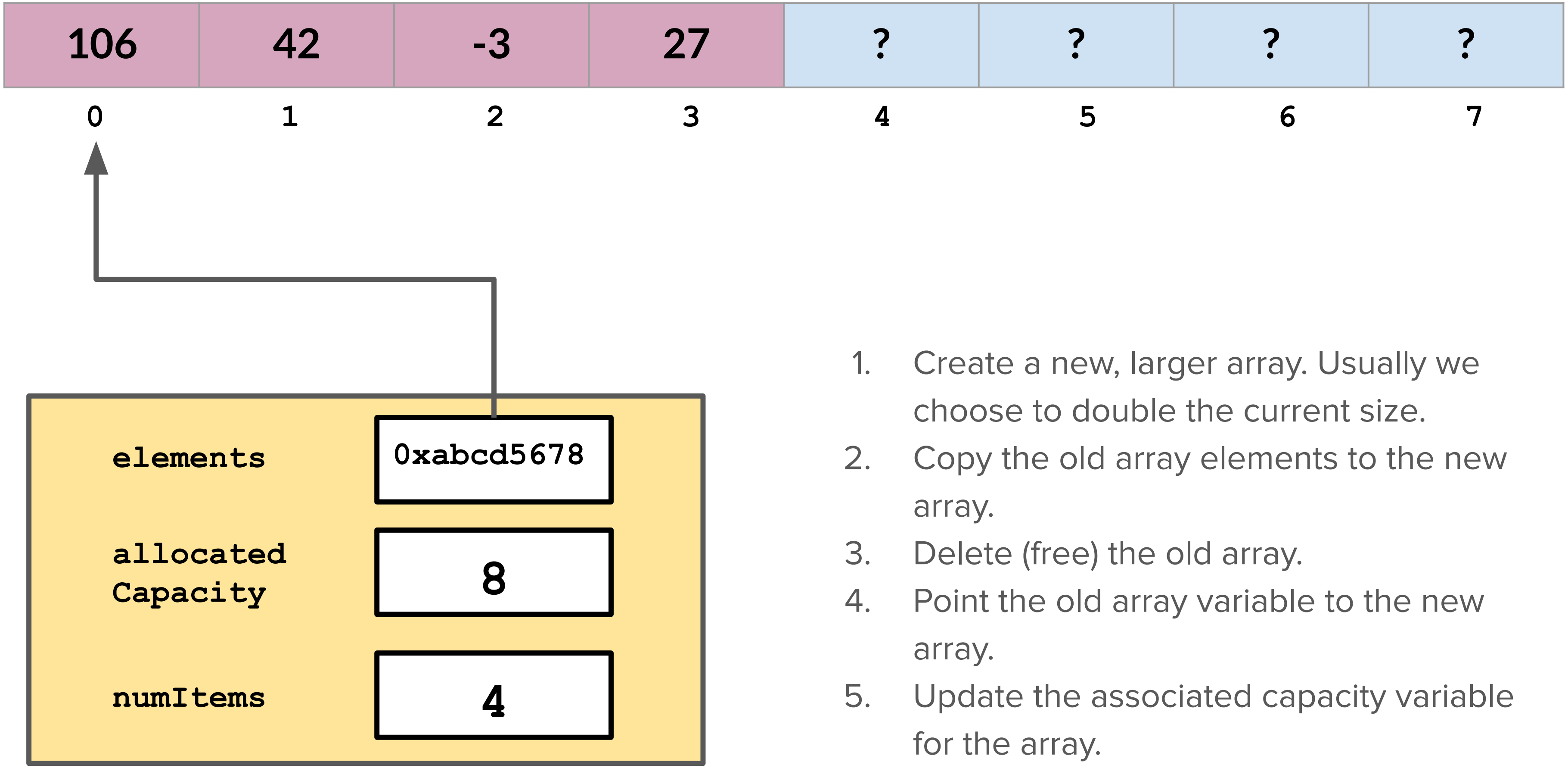




1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.
4. Point the old array variable to the new array.



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.



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.

Let's Code It! (Monday wrap-up)

`expand()` private helper function

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.

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

Abstract Data Structures

Abstraction boundary for the user

Data Organization Strategies

Fundamental C++ Data Storage

Levels of abstraction

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

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Levels of abstraction

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

What you'll focus on for Assignment 4

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Levels of abstraction

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

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

Abstract Data Structures

Data Organization Strategies

Fundamental C++ 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, linked lists, etc.)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ 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)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Levels of abstraction

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

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

What we'll focus on today!

tores our data?
(arrays)

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.

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.

- Useful for maintaining data sorted based on priorities
  - Emergency room waiting rooms

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.

- Useful for maintaining data sorted based on priorities
  - Emergency room waiting rooms
  - Different airline boarding groups (families and first class passengers, frequent flyers, boarding group A, boarding group B, etc.)

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.

- Useful for maintaining data sorted based on priorities
  - Emergency room waiting rooms
  - 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!

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.

- Useful for maintaining data sorted based on priorities
  - Emergency room waiting rooms
  - 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)

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)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Announcements

Announcements

- Assignment 4 is due next **Tuesday, July 27 at 11:59pm PDT.**
- The recording of the Assignment 4 YEAH session is on Ed.
- Make sure you're reading and applying your assignment feedback!
- LaIR is open today and tomorrow for A4 debugging help! Would strongly recommend!

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)

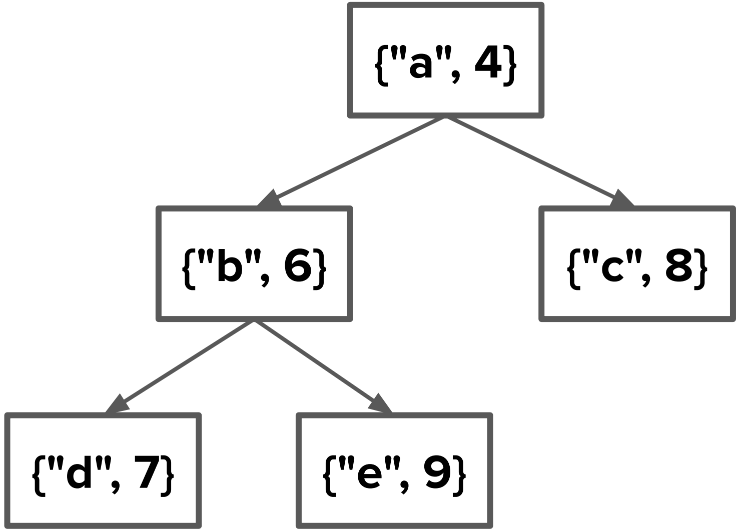
What is a binary heap?

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

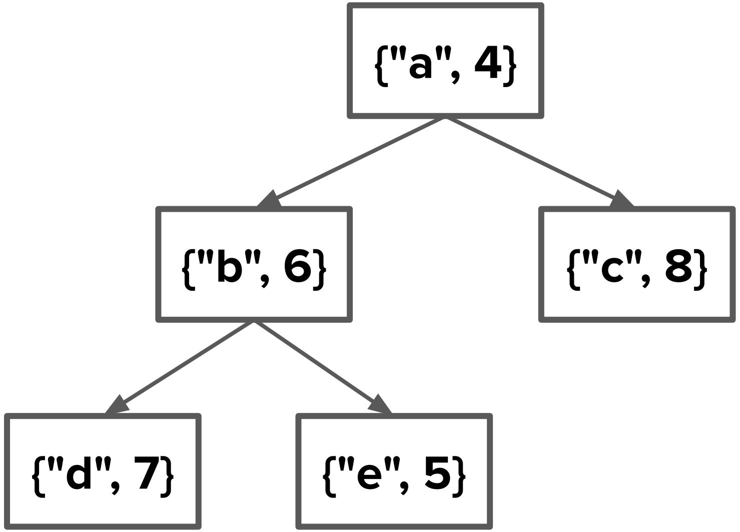
- Additional properties
  - **Binary**: Two children per parent (but no implied orderings between siblings)
  - **Completely filled** (each parents must have 2 children) except for the bottom level, which gets populated from **left to right**

- 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

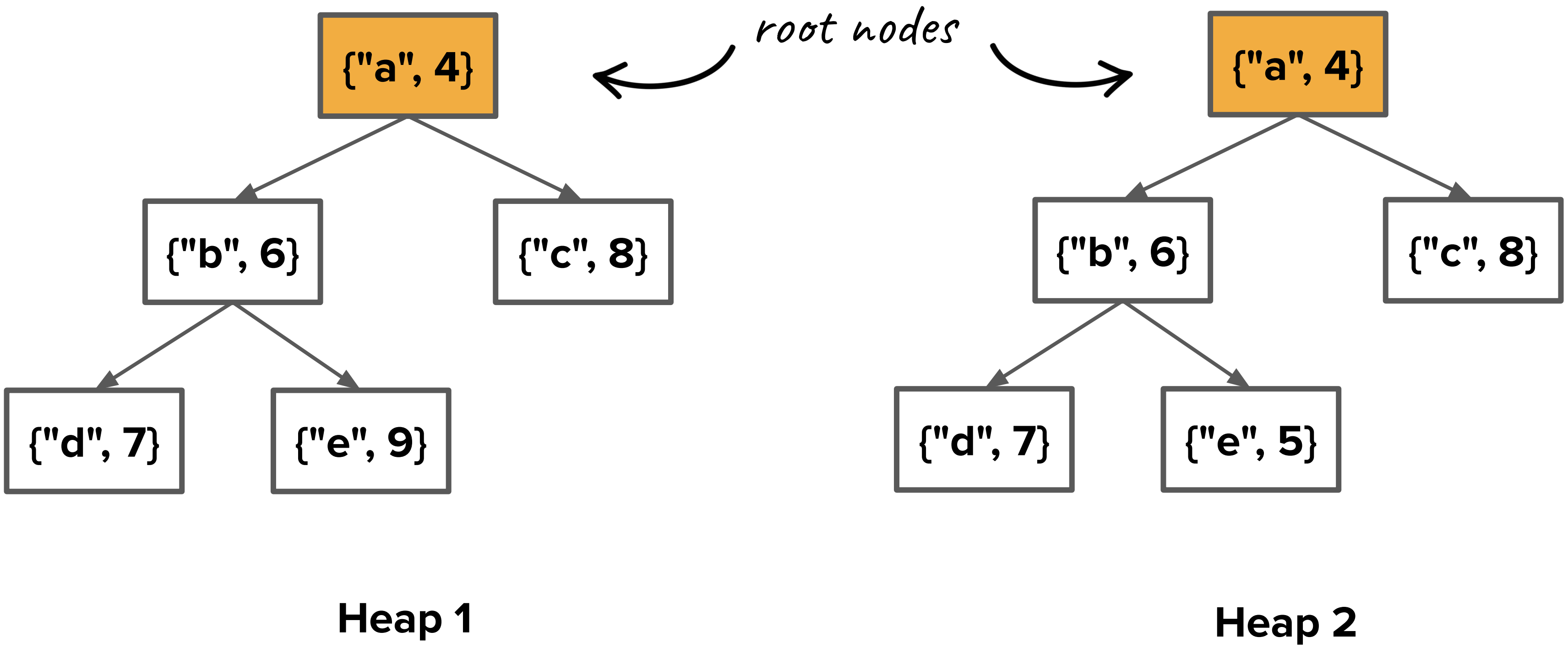


Heap 1

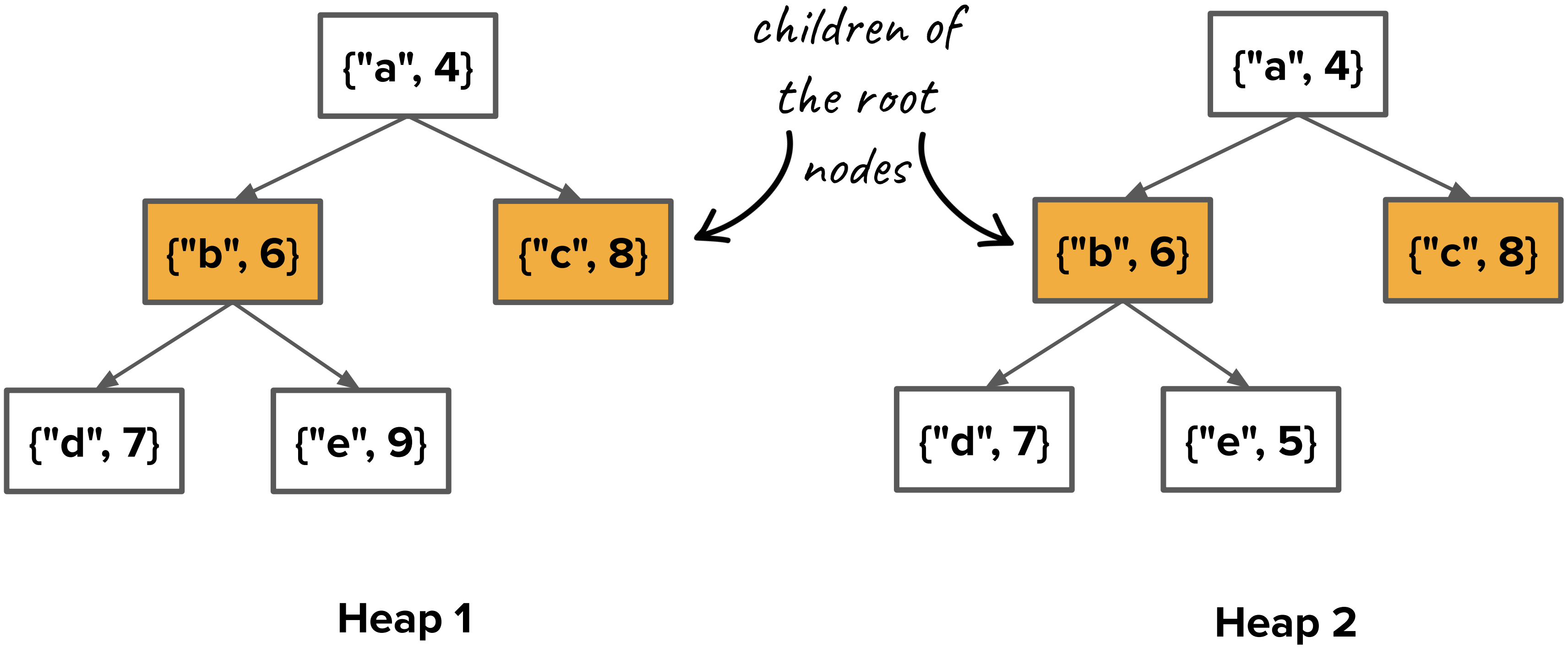


Heap 2

Spot the Valid Min-Heap

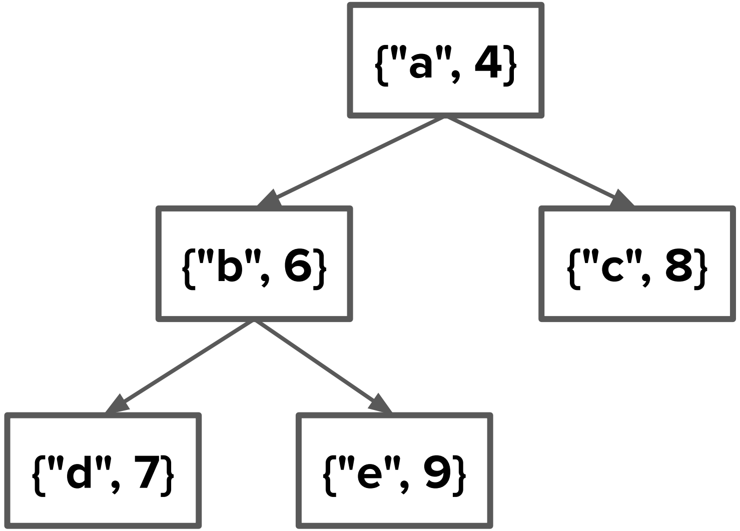


Spot the Valid Min-Heap

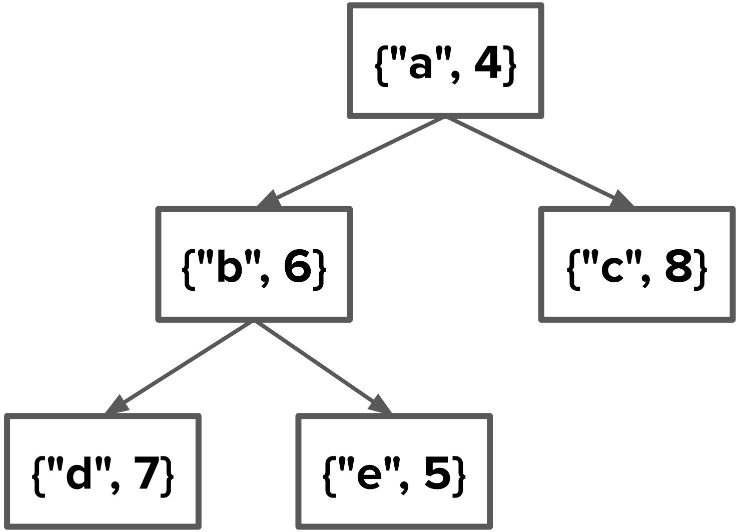


Spot the Valid Min-Heap

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

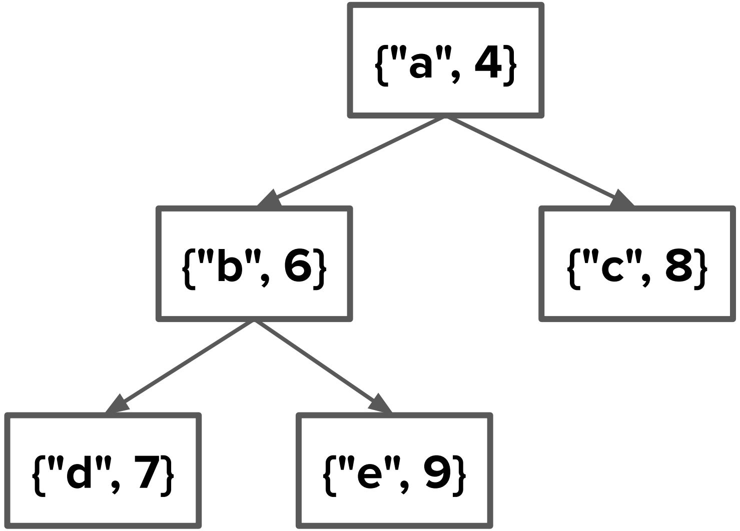


Heap 1

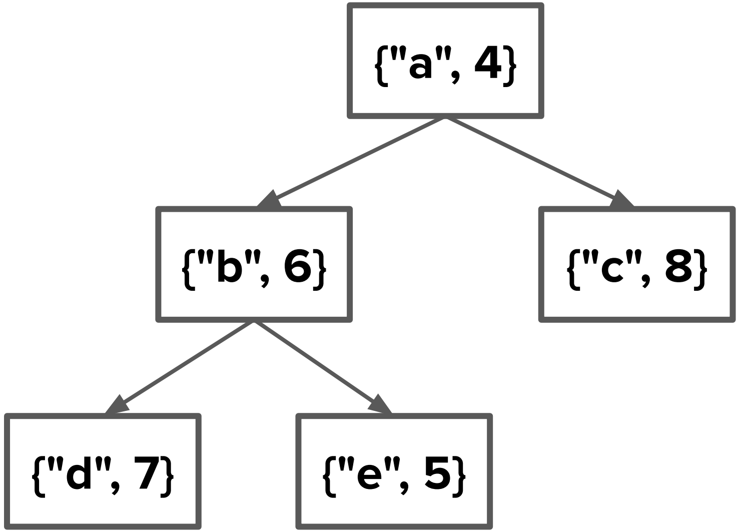


Heap 2

Spot the Valid Min-Heap

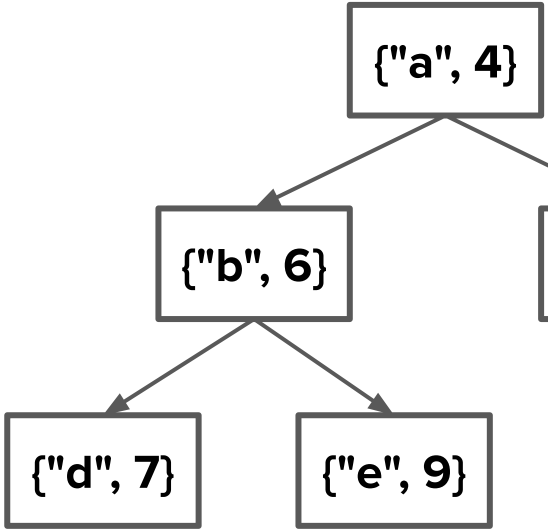


Heap 1



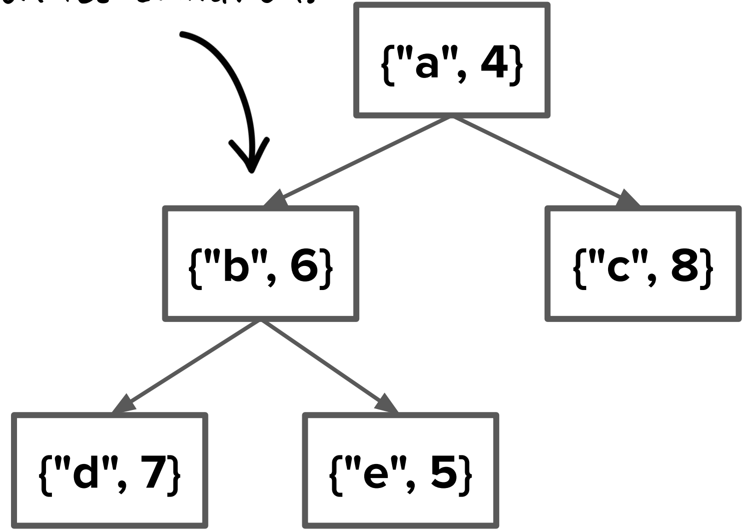
Heap 2

Spot the Valid Min-Heap



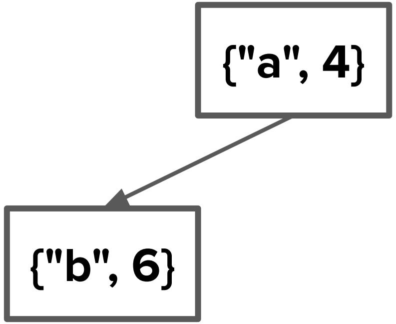
Heap 1

This element is not smaller than both its children!

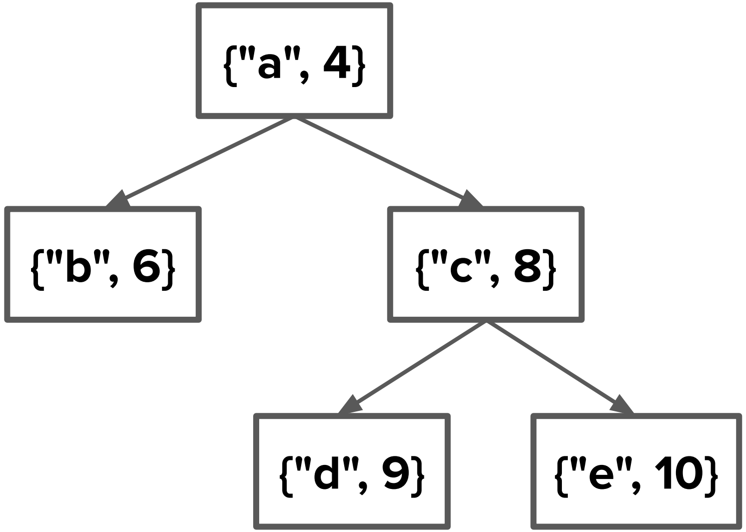


Heap 2

Spot the Valid Min-Heap (Round 2)

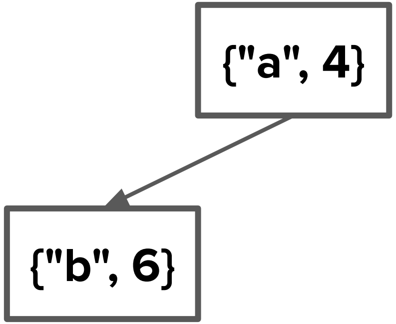


Heap 1



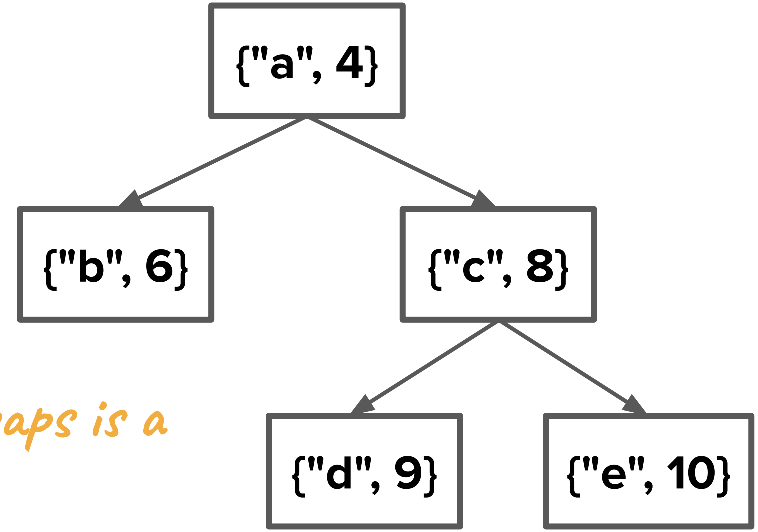
Heap 2

Spot the Valid Min-Heap (Round 2)



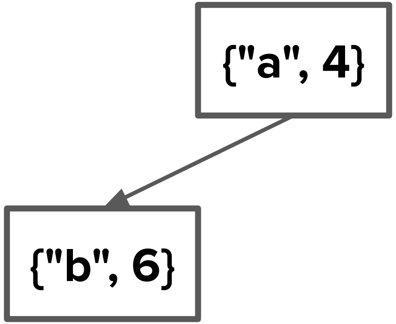
Heap 1

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

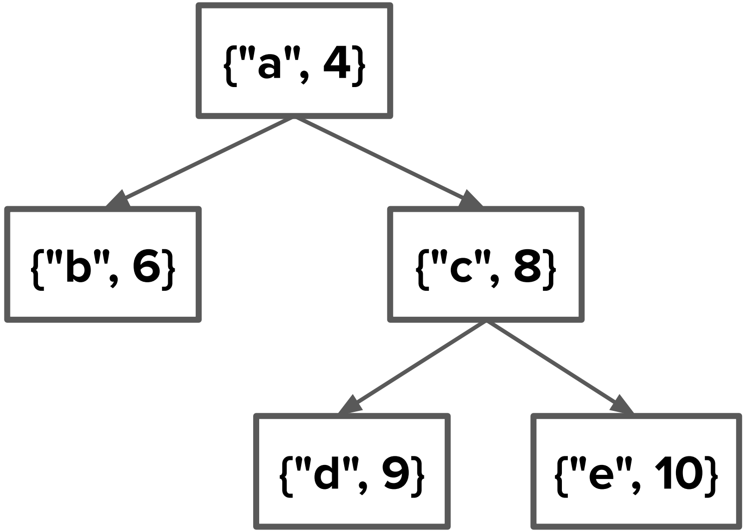


Heap 2

Spot the Valid Min-Heap (Round 2)

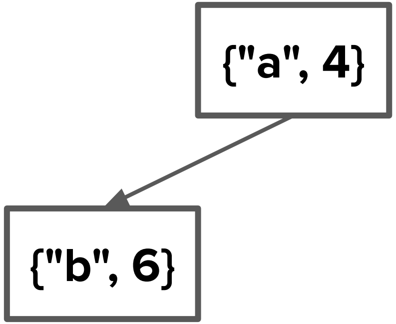


Heap 1

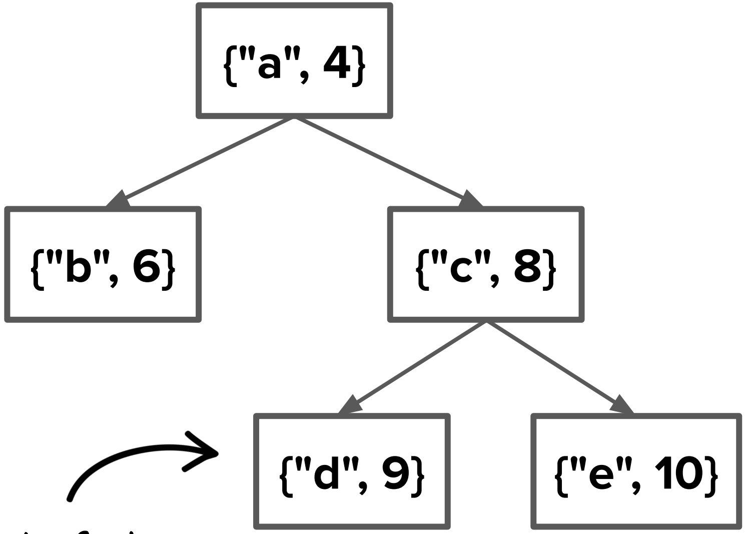


Heap 2

Spot the Valid Min-Heap (Round 2)



Heap 1



This level of the heap is not complete

Heap 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)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

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**)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

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.

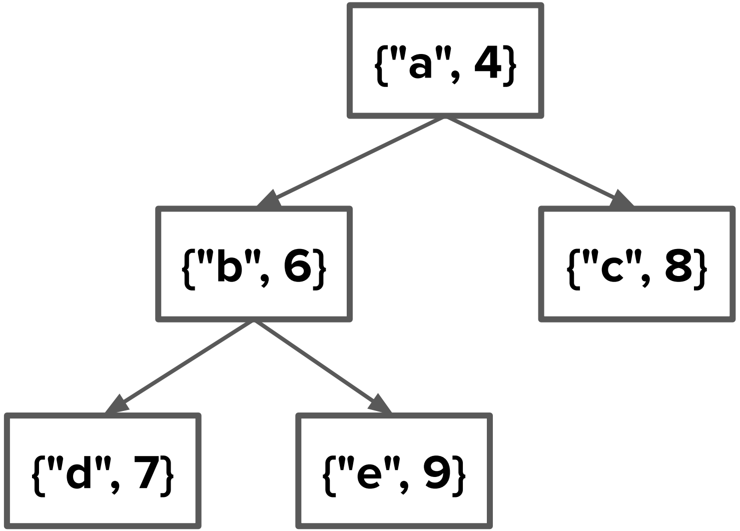
Binary heaps + implementation

- 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.

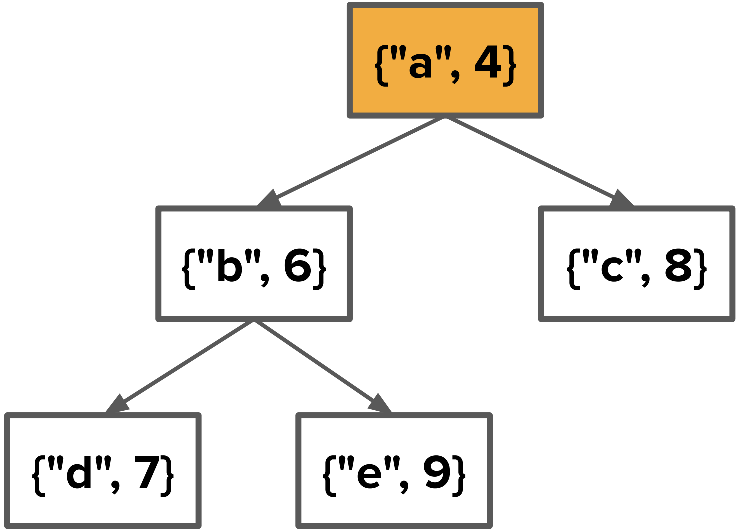
How are parents and children in the tree related in the array?

Binary heaps + implementation



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

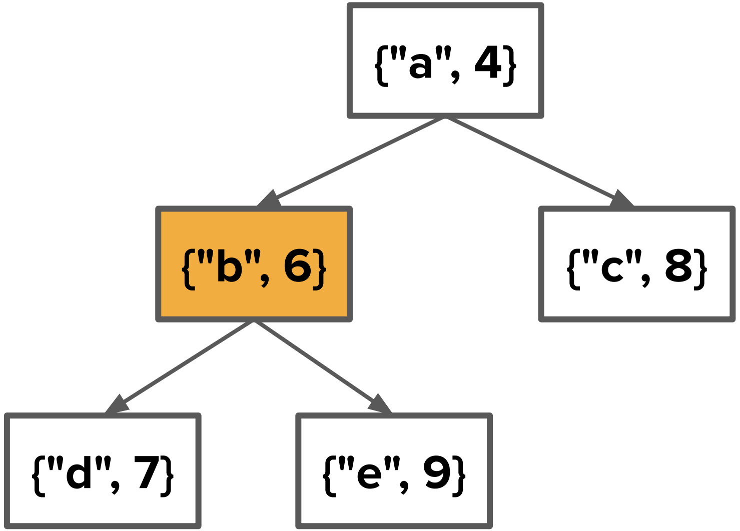
Binary heaps + implementation



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

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

Binary heaps + implementation

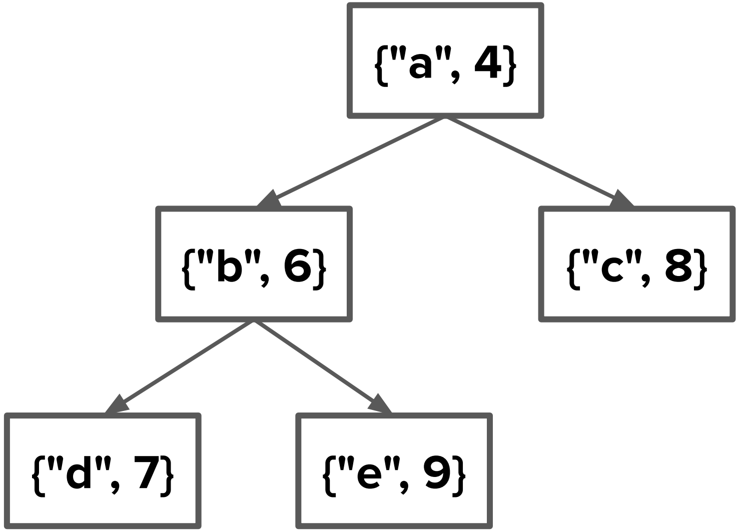


{“a”, 4}	{“b”, 6}	{“c”, 8}	{“d”, 7}	{“e”, 9}
0	1	2	3	4

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

Binary heaps + implementation

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



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

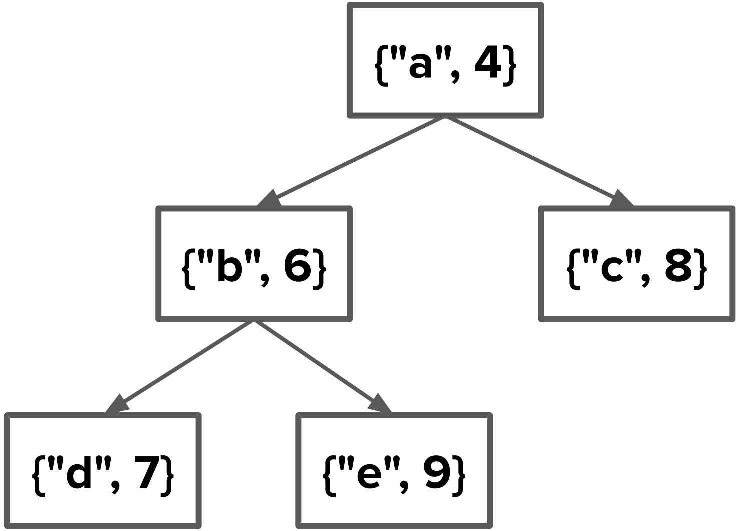
Parent index: 1
Left child: 3
Right child: 4

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

Manipulating heap contents

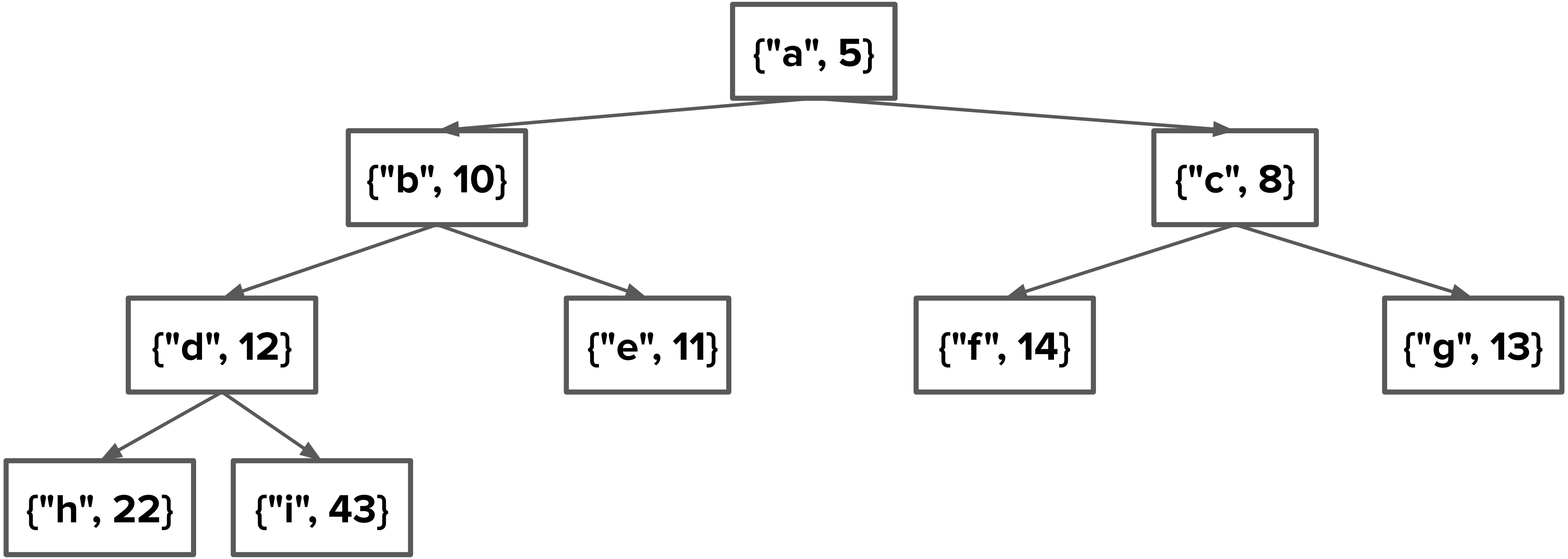
Heap operations

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.

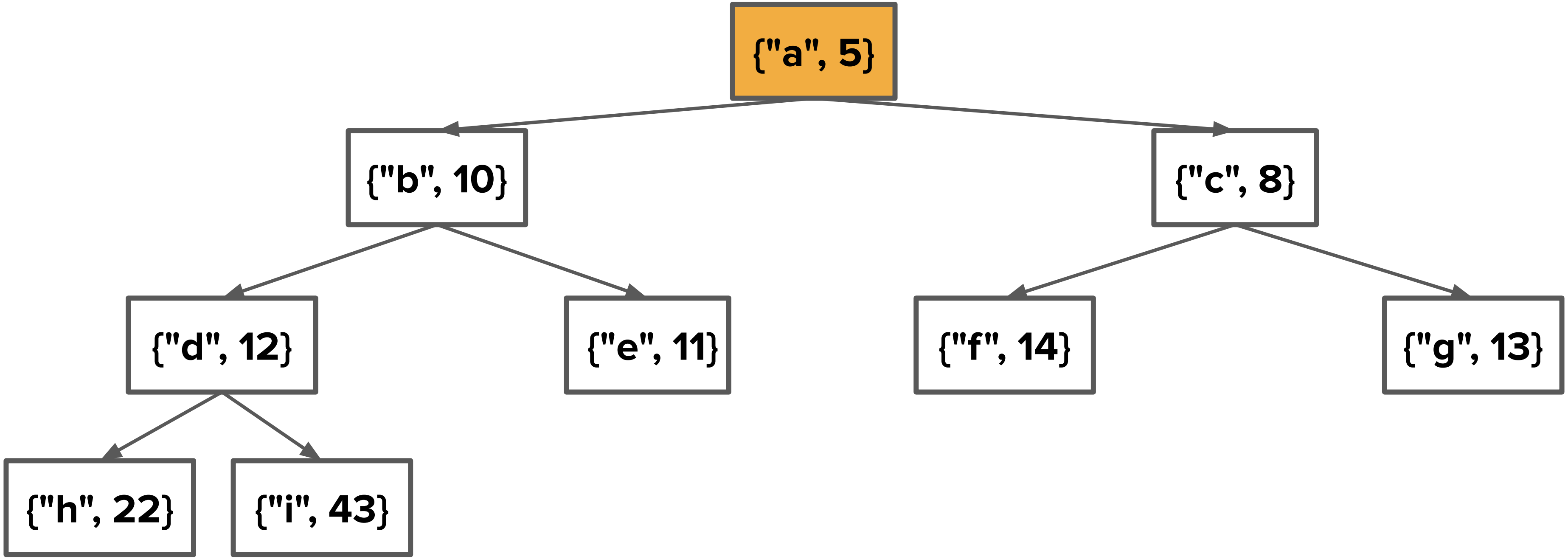
- **`dequeue()`**: remove the highest priority (smallest element for a min-heap) from the heap. This changes the state of the heap, and thus we have to do work to restore the heap property.



{"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?

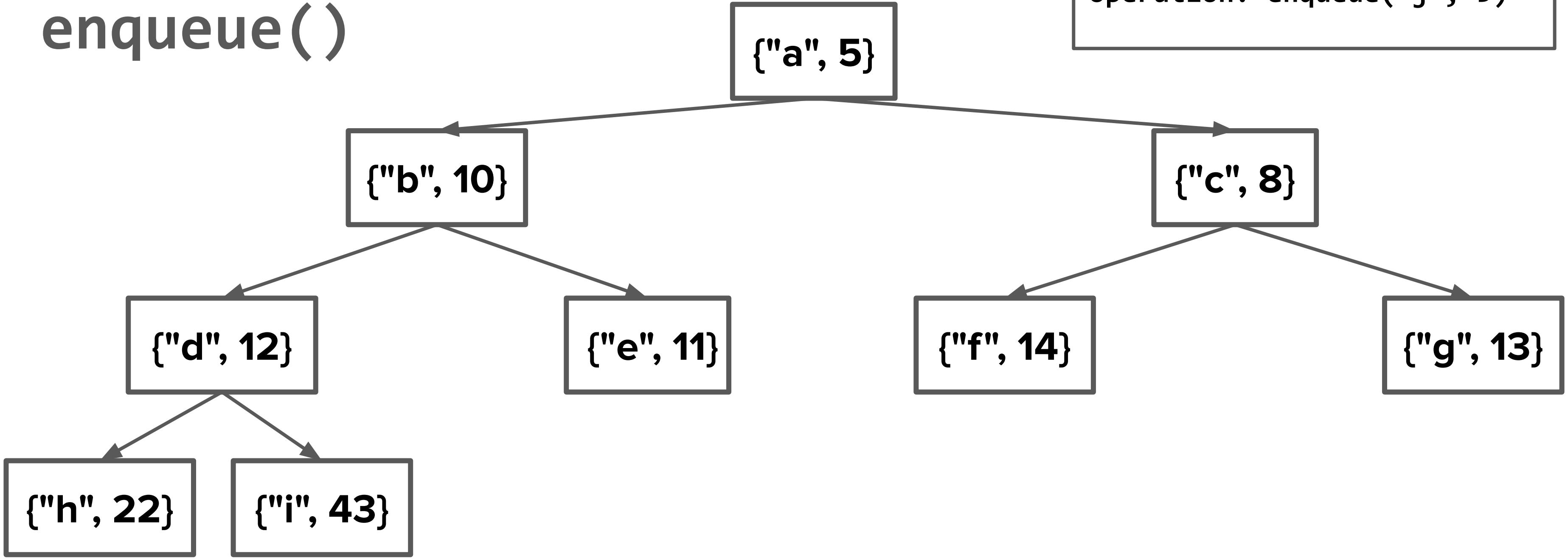
- Example: What if we called **`enqueue({"j", 9})`** into the heap from before?

- The key is to understand how heaps are built: it is critical that we fill each level from left to right.

- So, we 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;`**

enqueue()

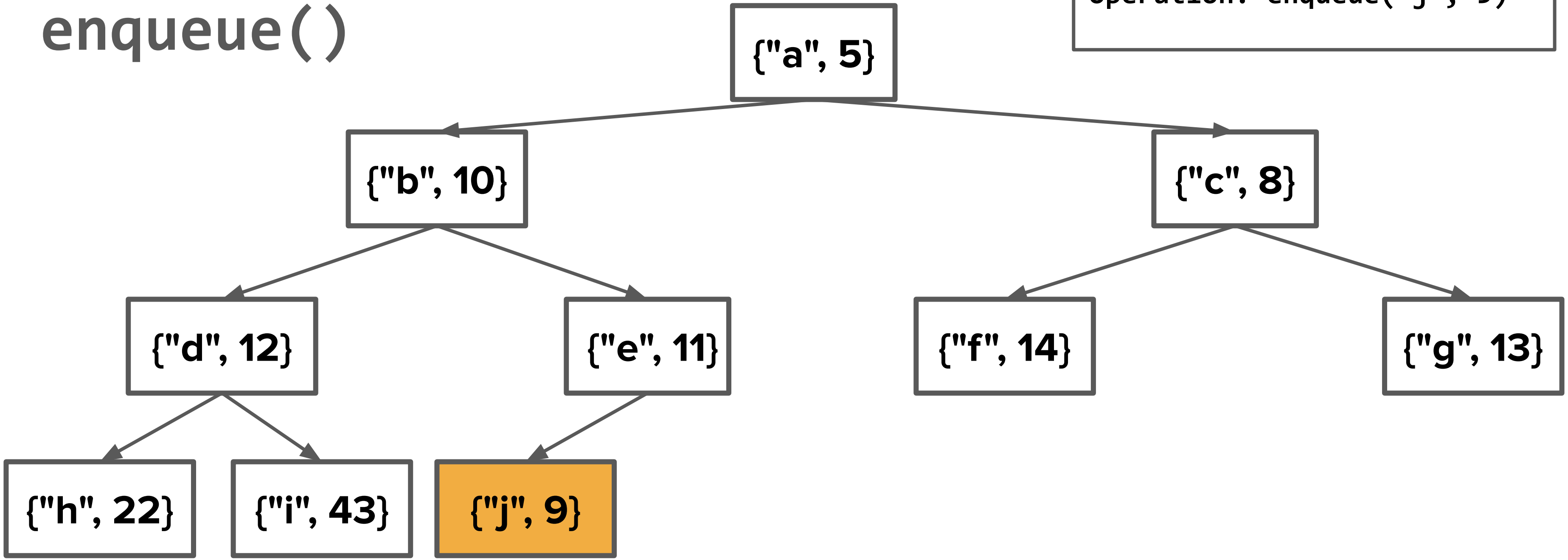
Operation: enqueue("j", 9)



{"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()

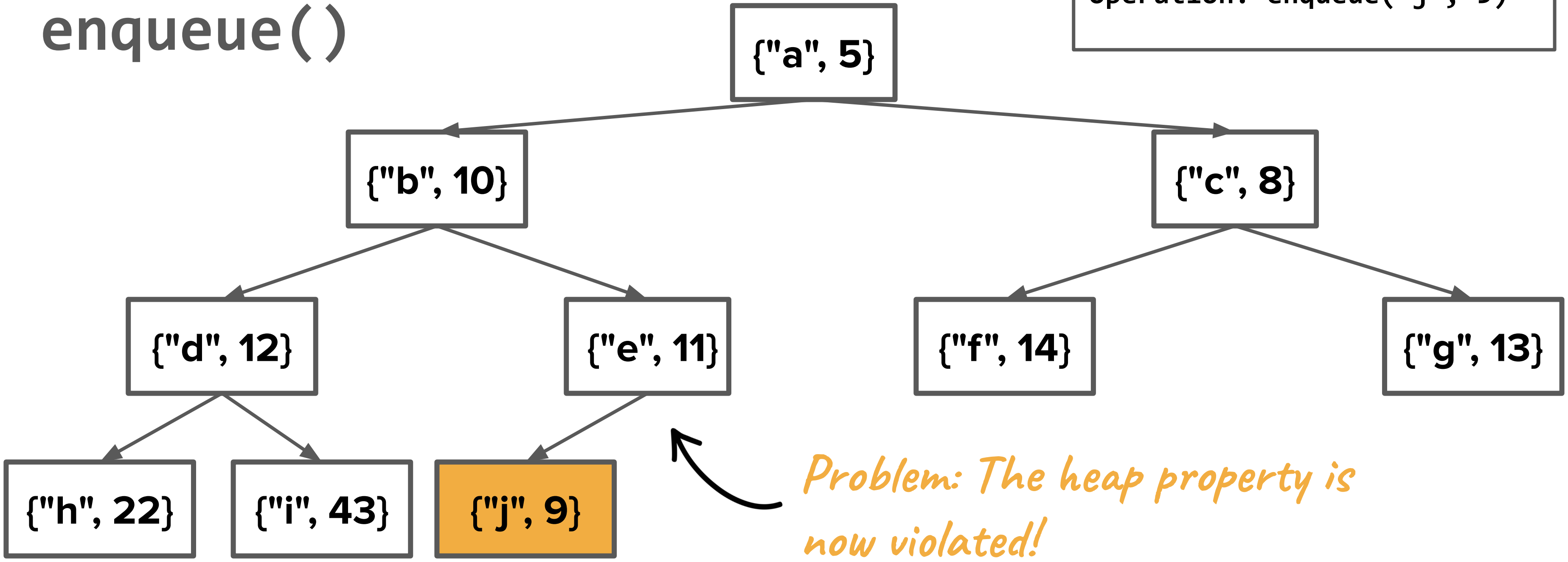
```
Operation: enqueue("j", 9)
```



{"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

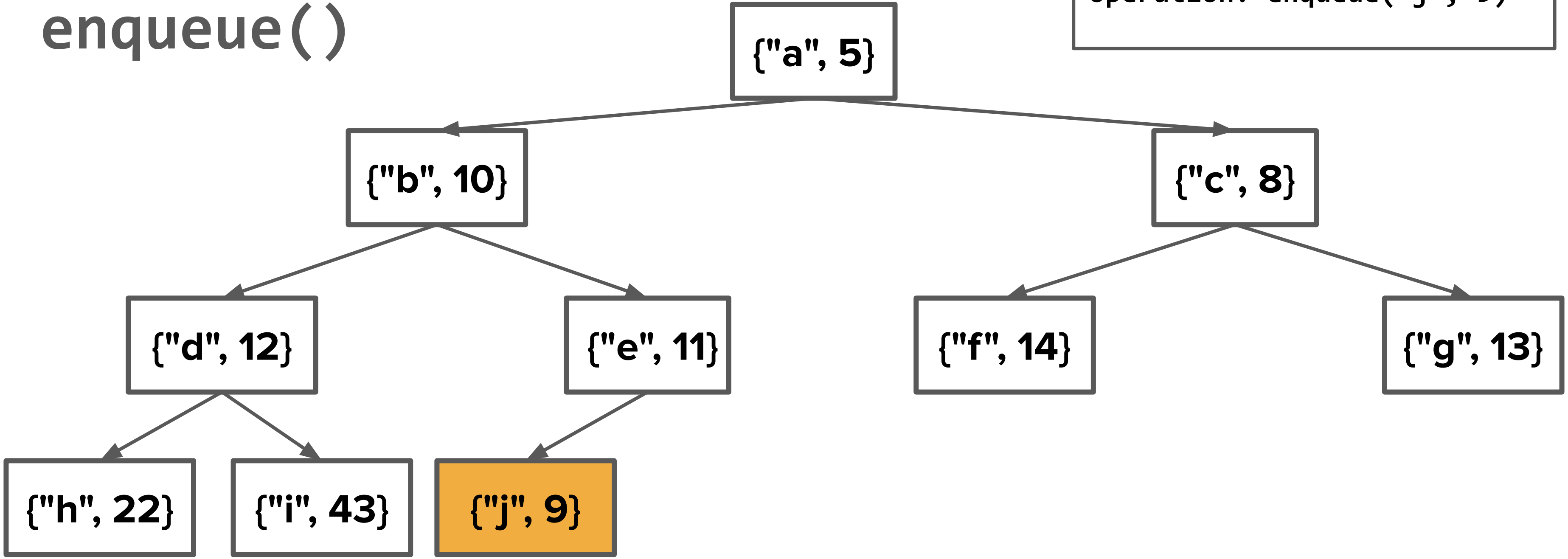
enqueue()

- Inserting our new element into the first empty slot destroyed the heap property – it is now our job to "fix things up" and restore the heap property.

- 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()

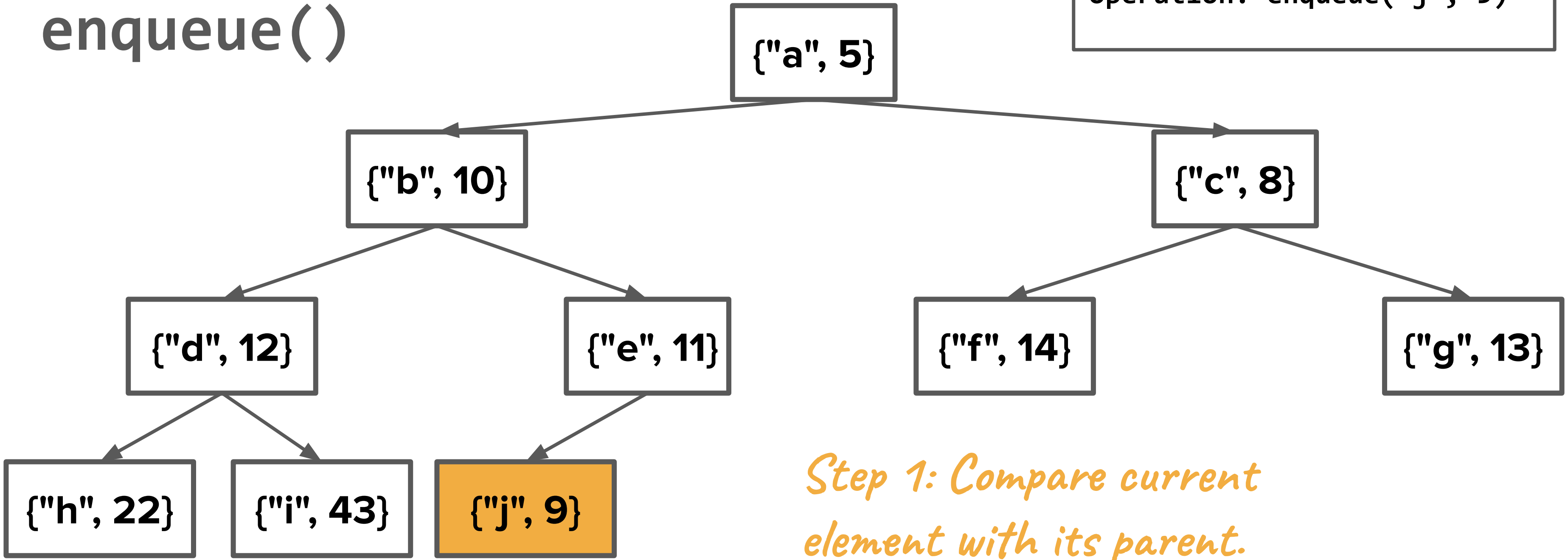
`Operation: enqueue("j", 9)`



{"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)

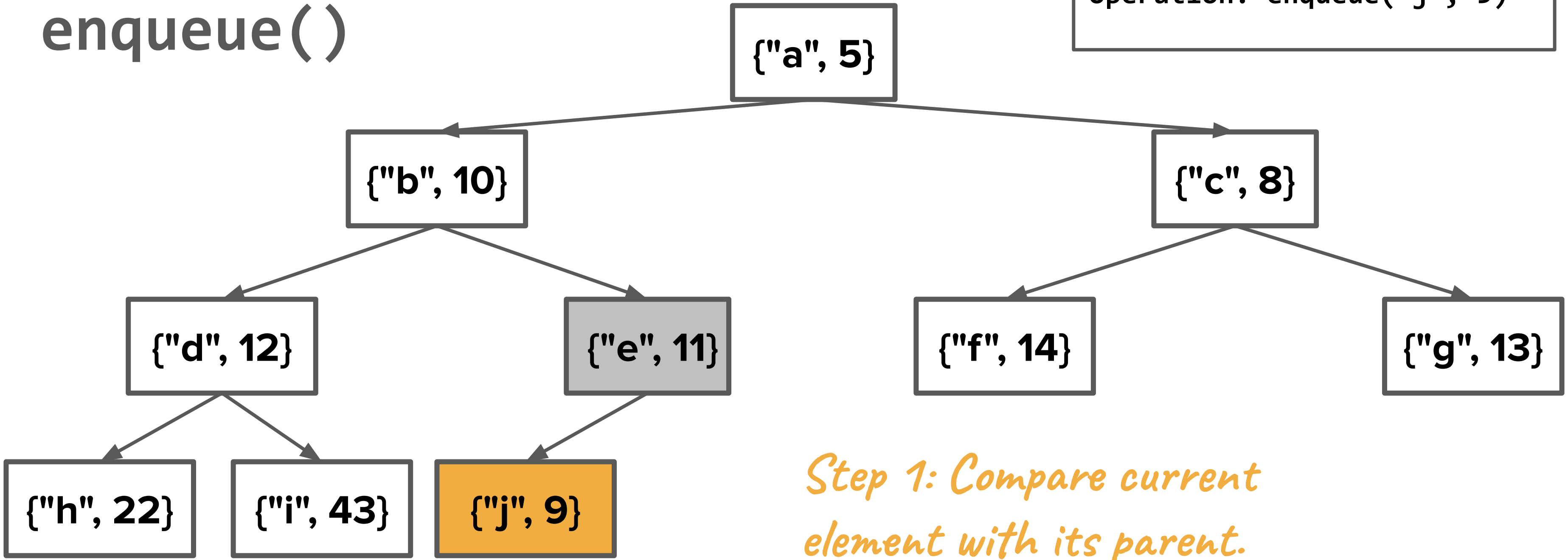


Step 1: Compare current element with its parent.

{"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)`

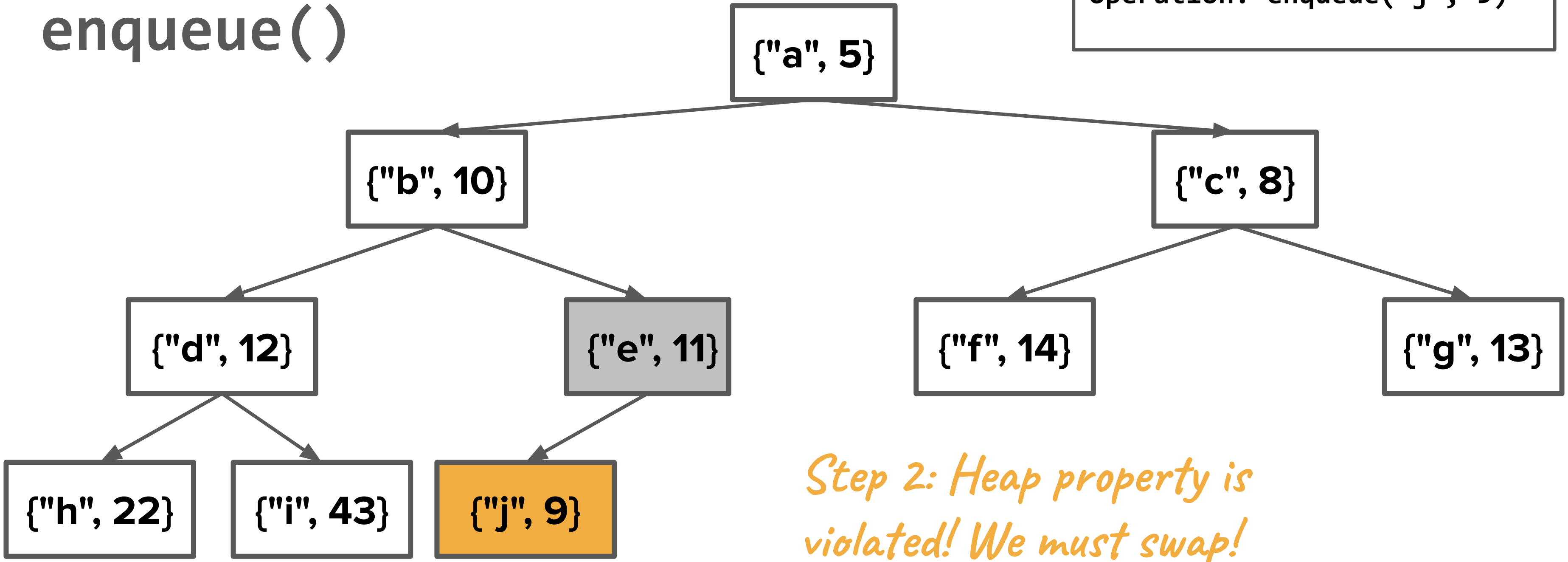


Step 1: Compare current element with its parent.

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

enqueue()

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

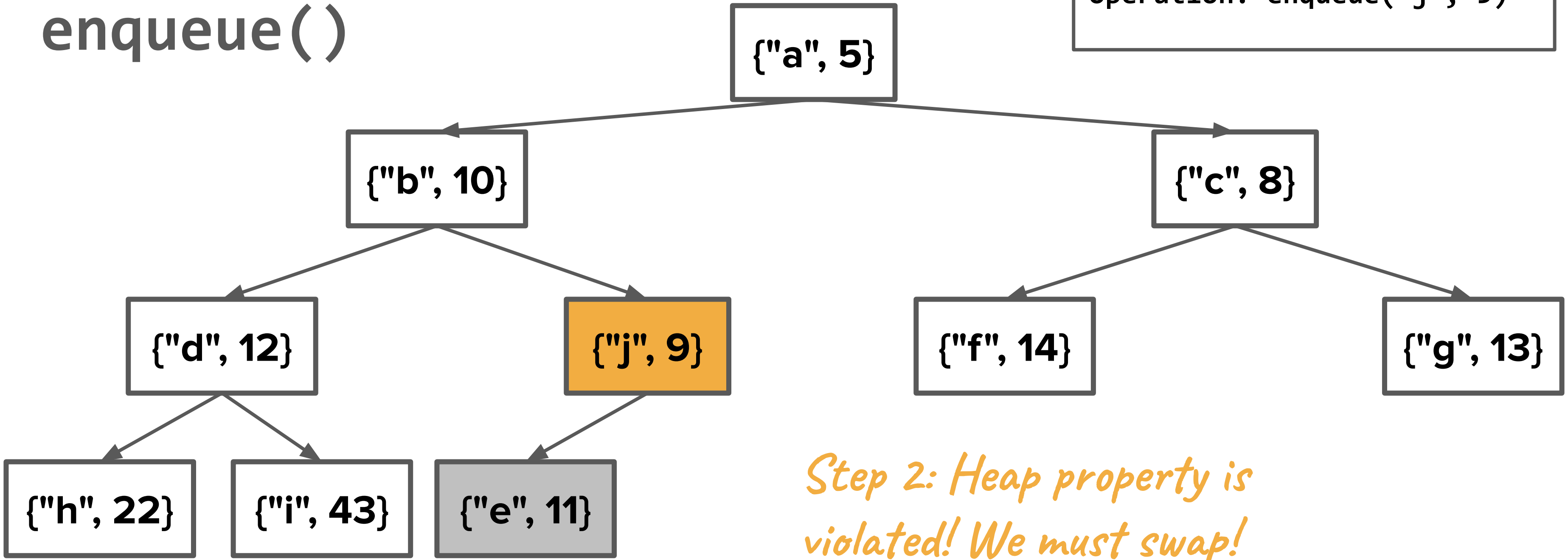


Step 2: Heap property is violated! We must swap!

{"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)

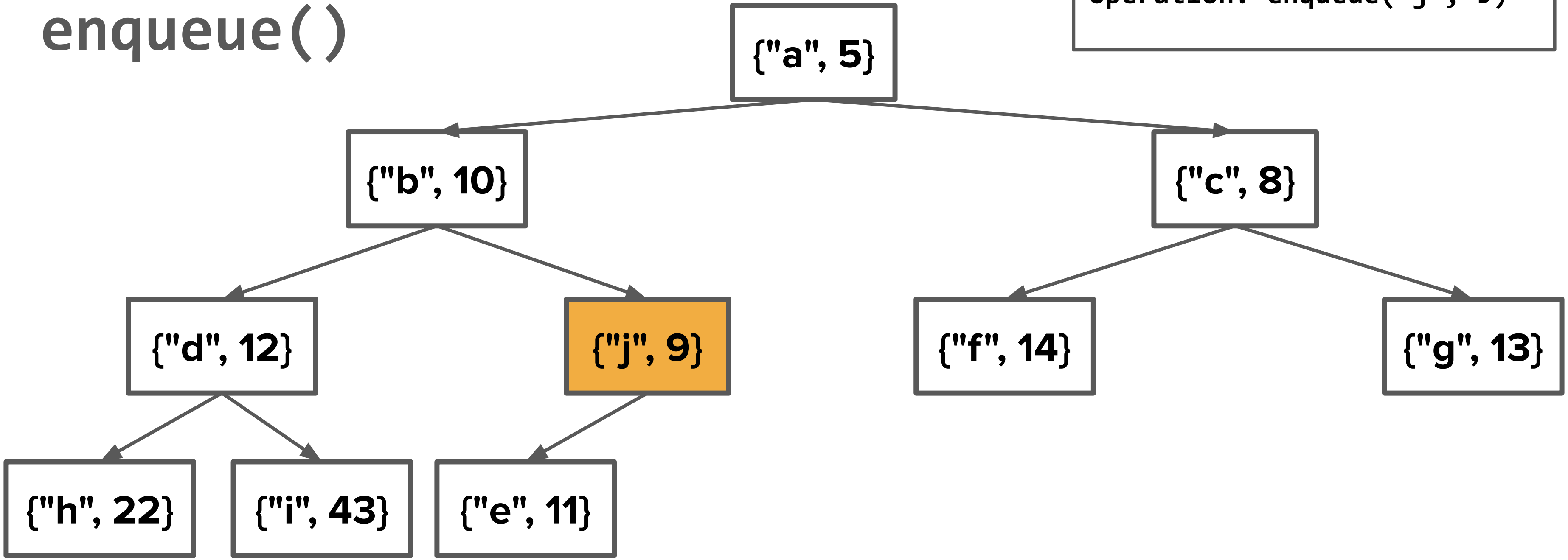


Step 2: Heap property is violated! We must swap!

{"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()

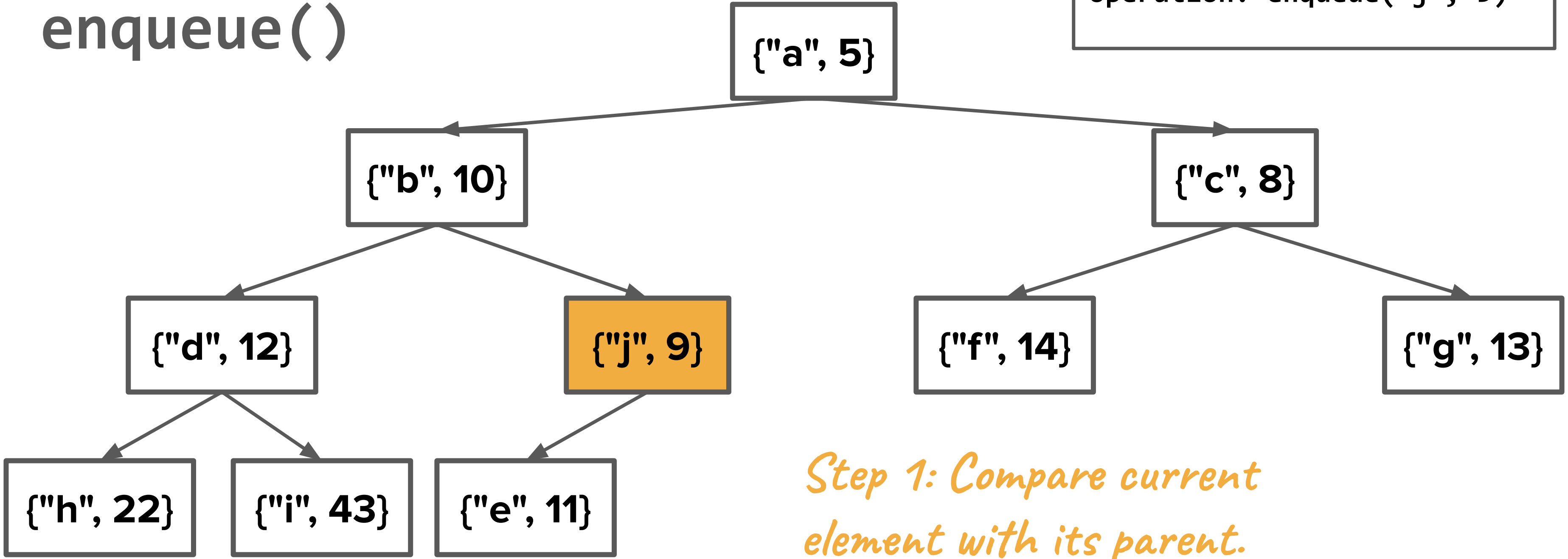
```
Operation: enqueue("j", 9)
```



{"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)`

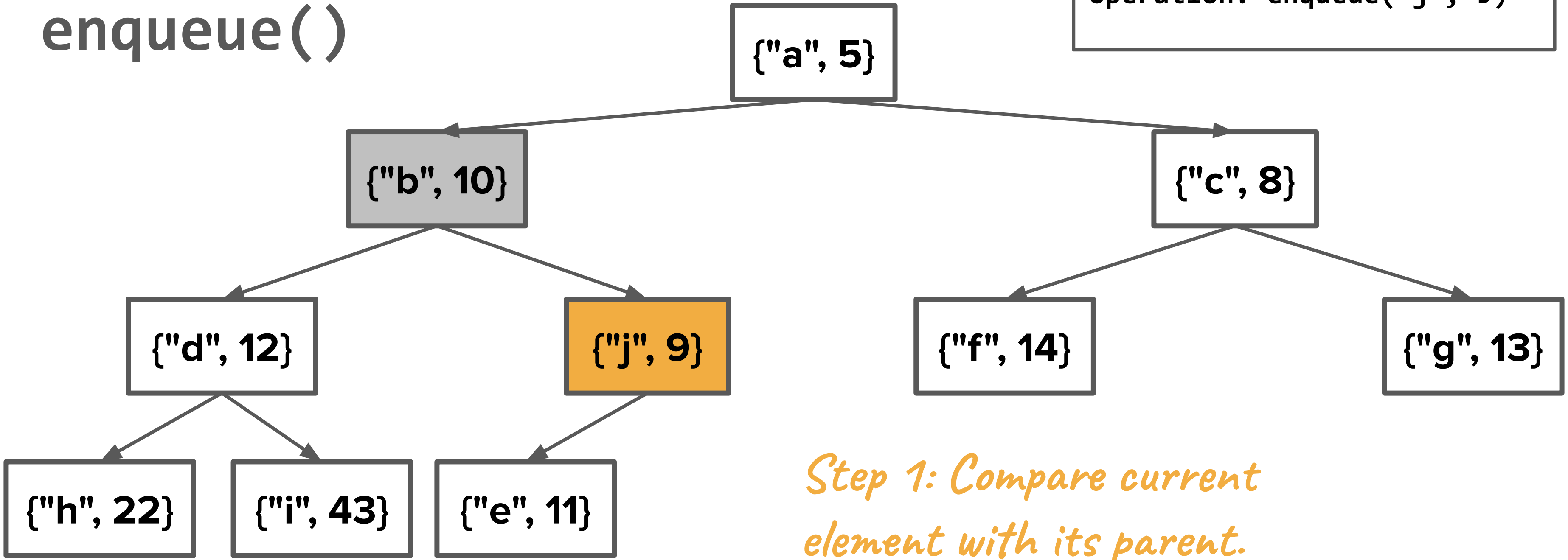


Step 1: Compare current element with its parent.

{"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)`

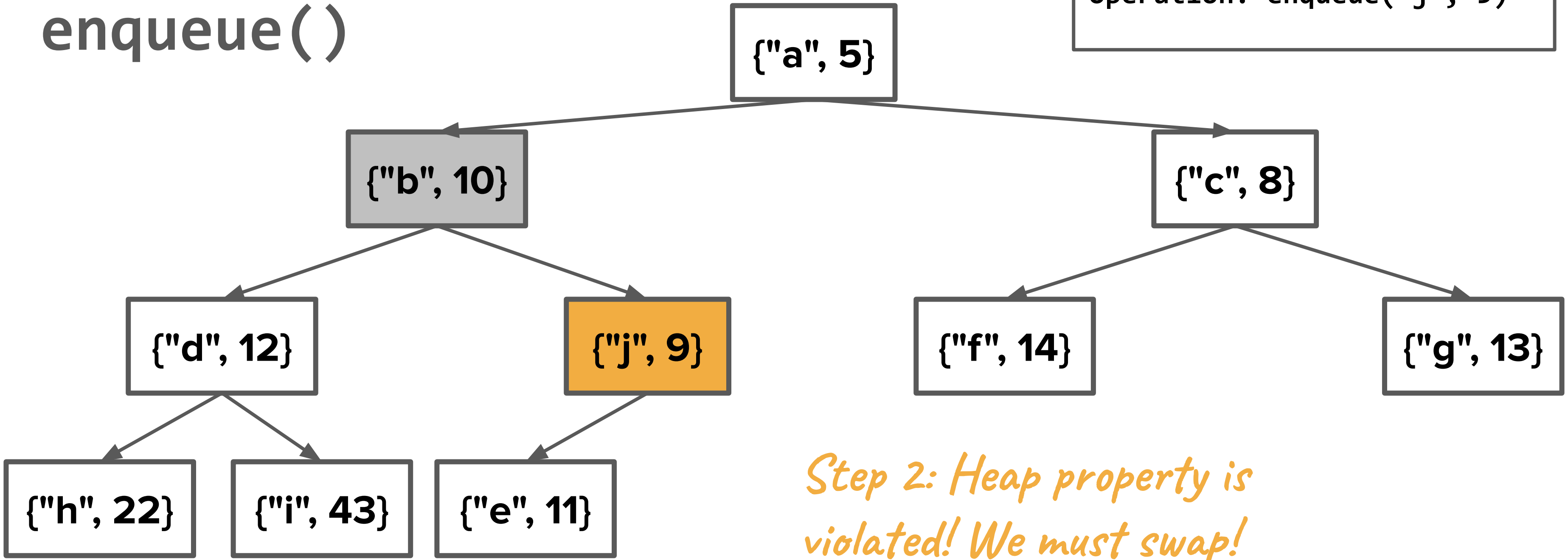


Step 1: Compare current element with its parent.

{"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)

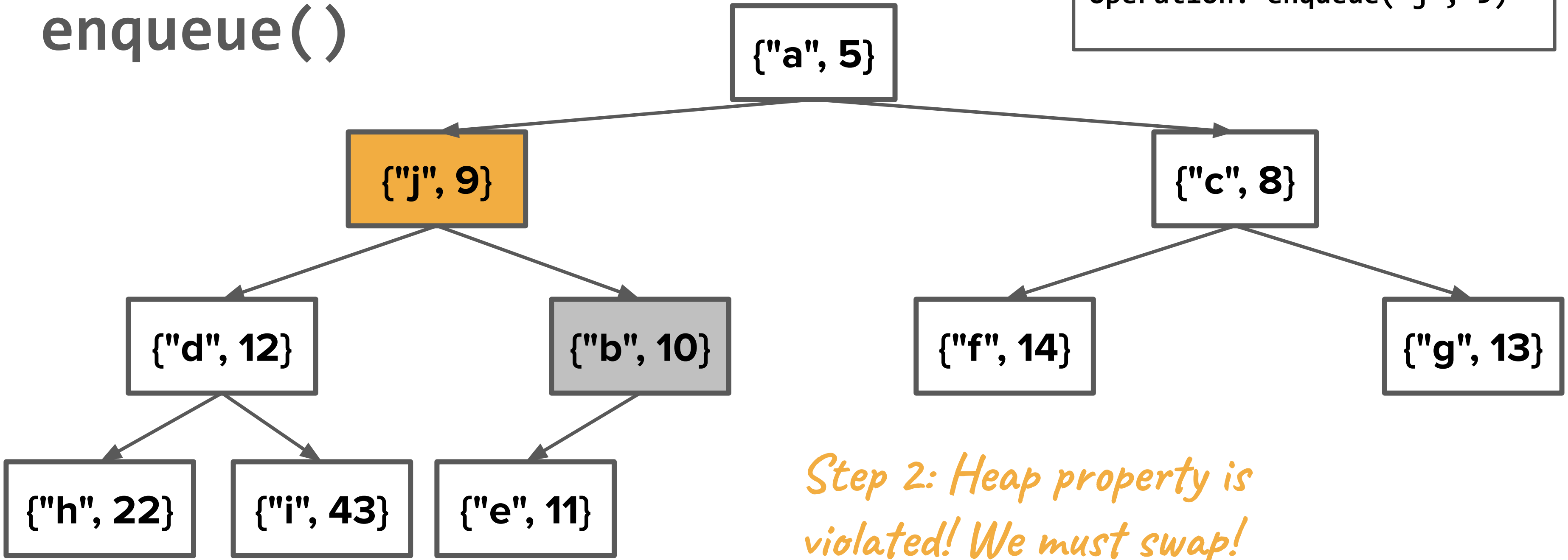


Step 2: Heap property is violated! We must swap!

{"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)`

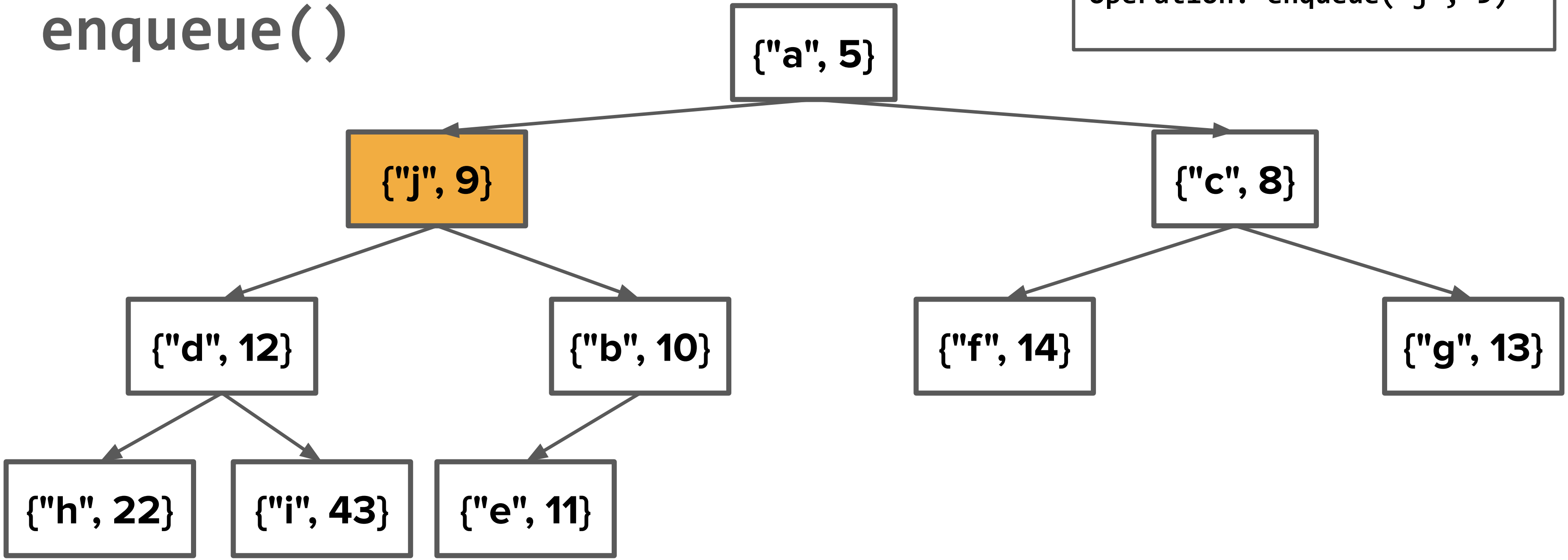


Step 2: Heap property is violated! We must swap!

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

enqueue()

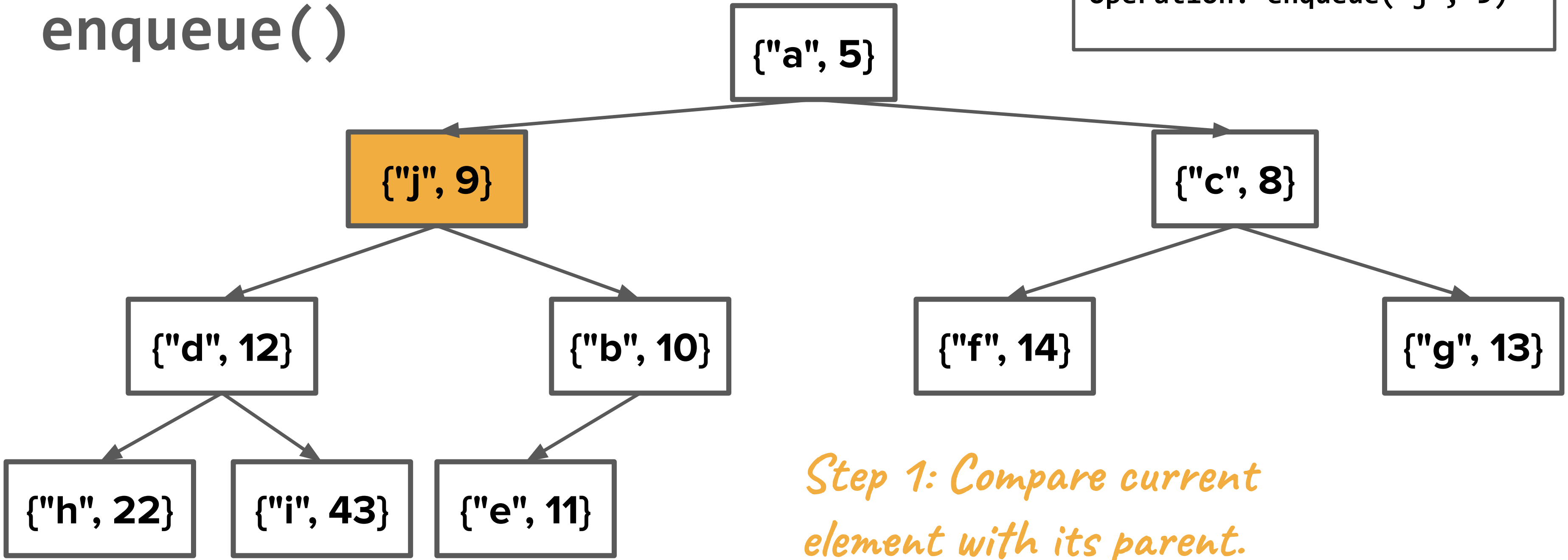
`Operation: enqueue("j", 9)`



{"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)`

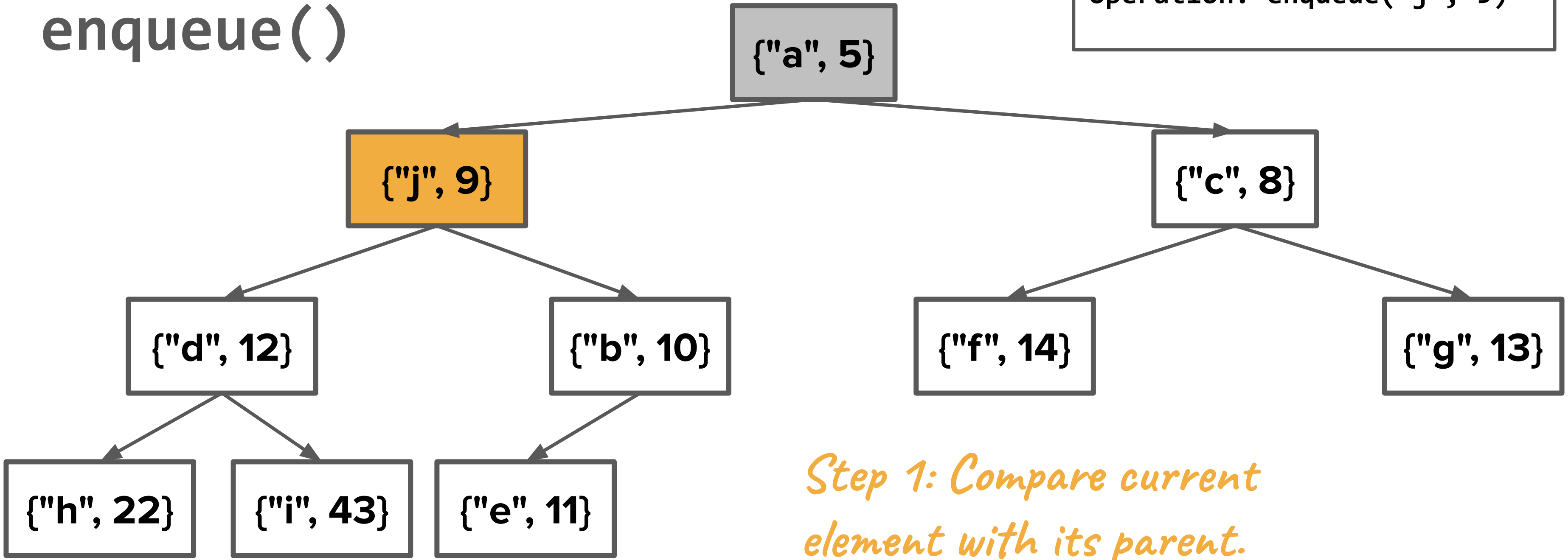


Step 1: Compare current element with its parent.

{"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)`

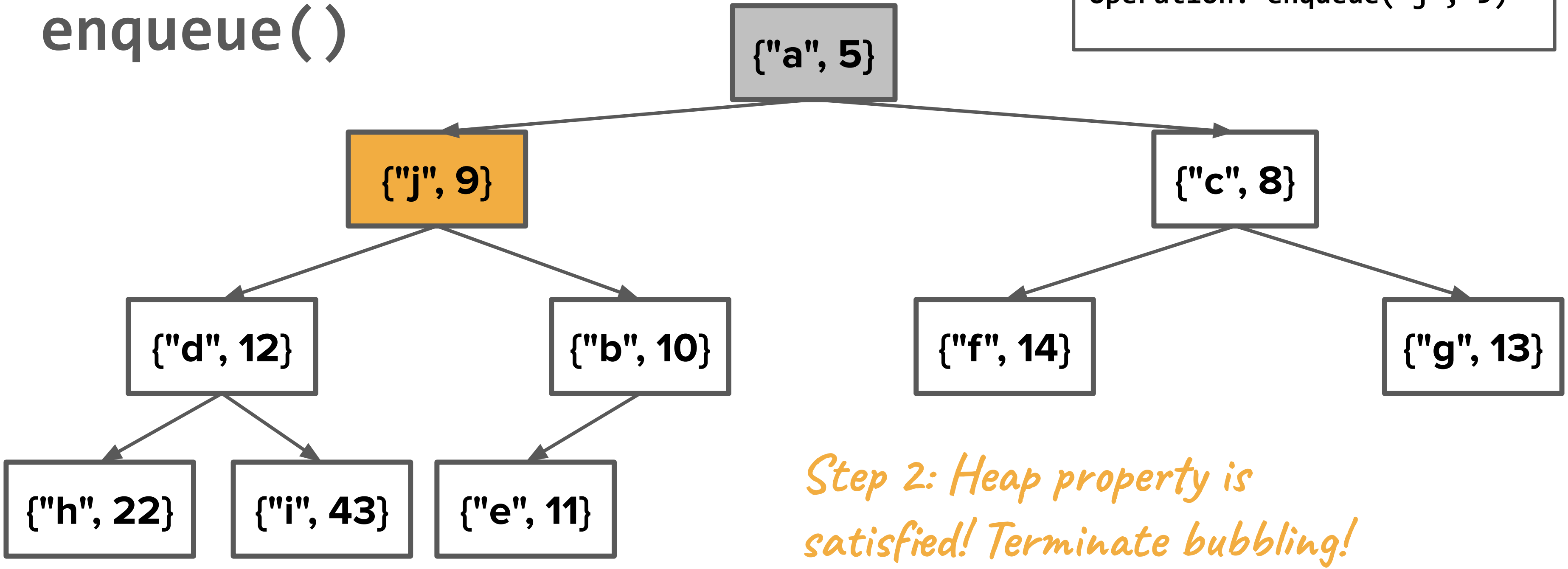


Step 1: Compare current element with its parent.

{"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)

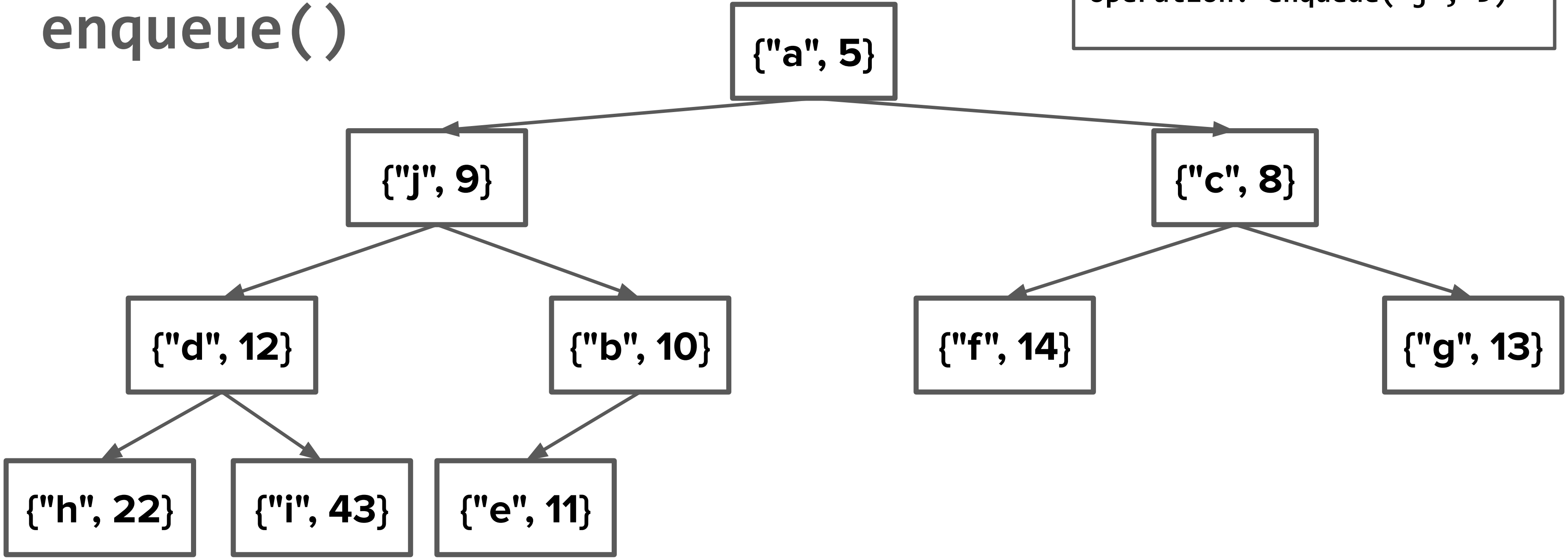


Step 2: Heap property is satisfied! Terminate bubbling!

0	1	2	3	4	5	6	7	8	9	10	11
{"a", 5}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	{"e", 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.
    - Guiding question: *what is the maximum depth of a binary heap from root to bottom in terms of n total elements?*

`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.

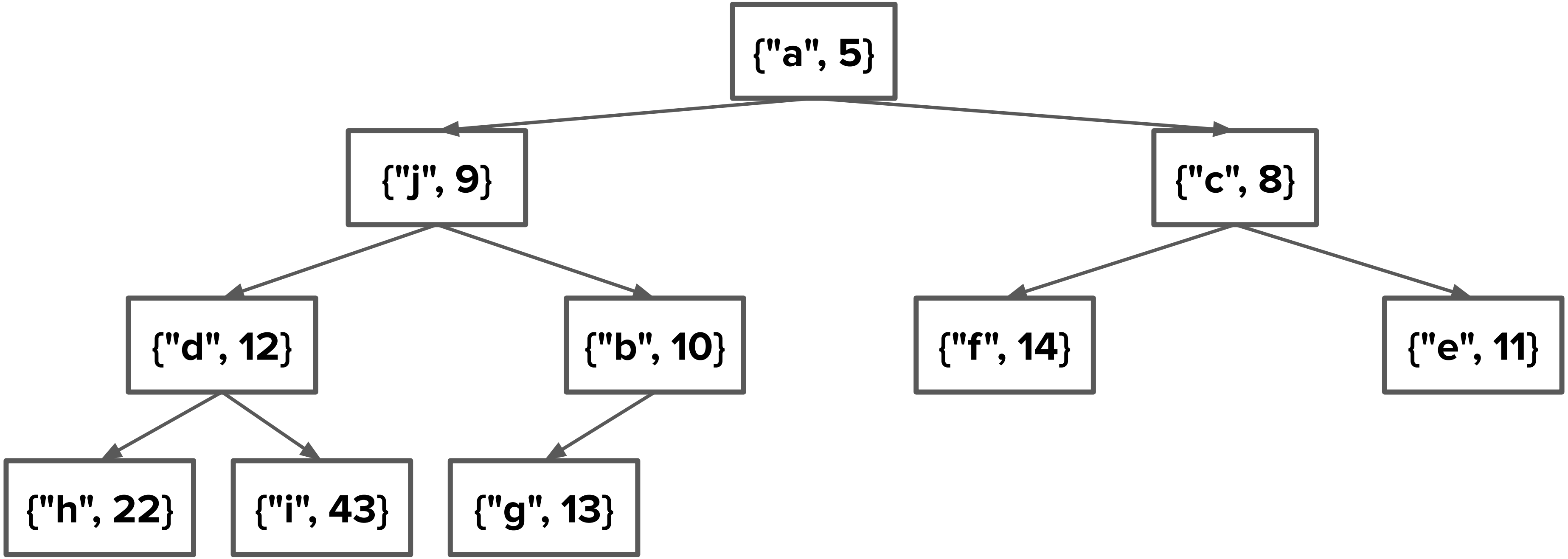
- What is the runtime complexity of the **enqueue** operation?
    - In the worst case scenario, we have to bubble up the new element all the way up to the root position.
    - Since there are **`n`** total elements, the tree will have **`log n`** levels, which means we would do **`log n`** comparisons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

dequeue()

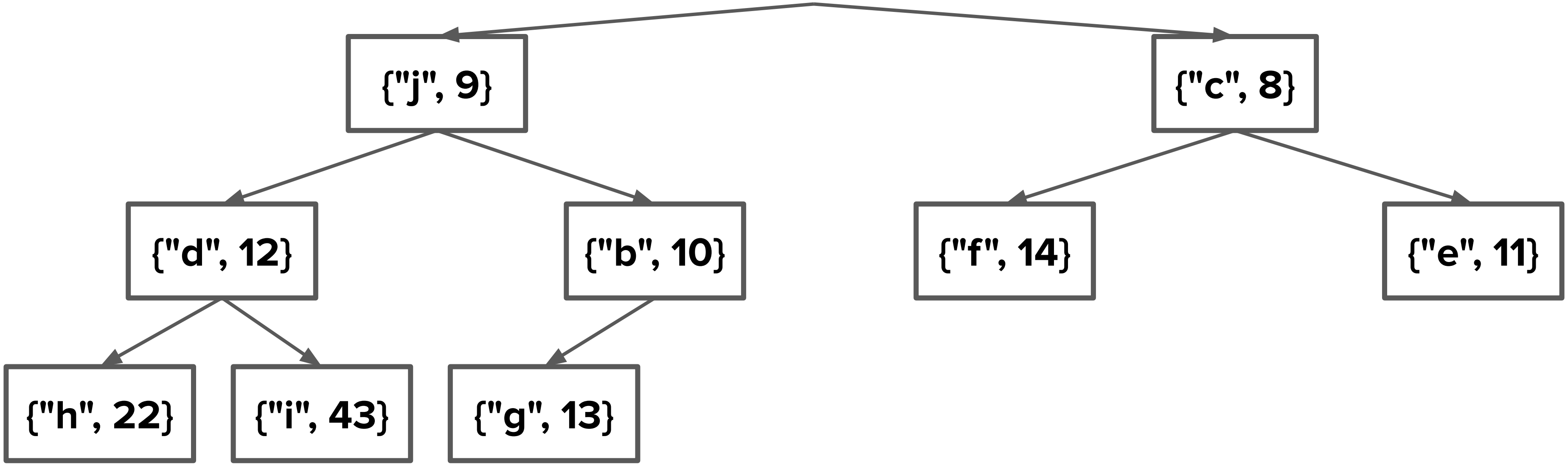
- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!

`dequeue()`

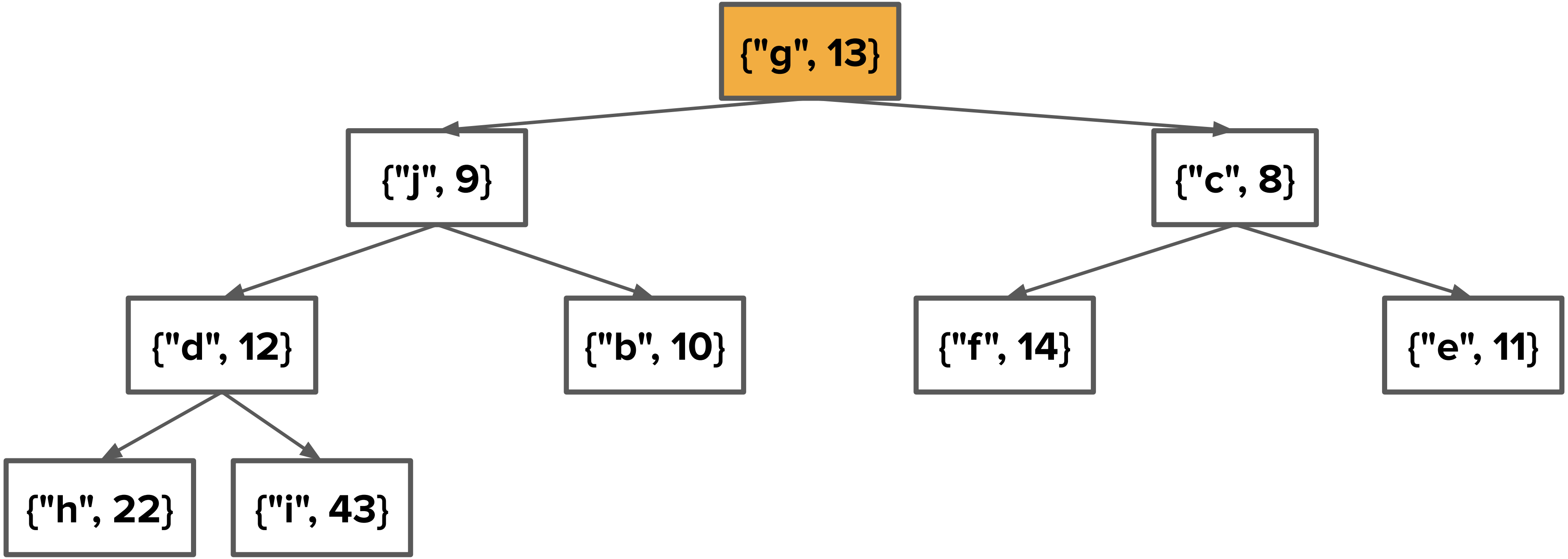
- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!



{"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



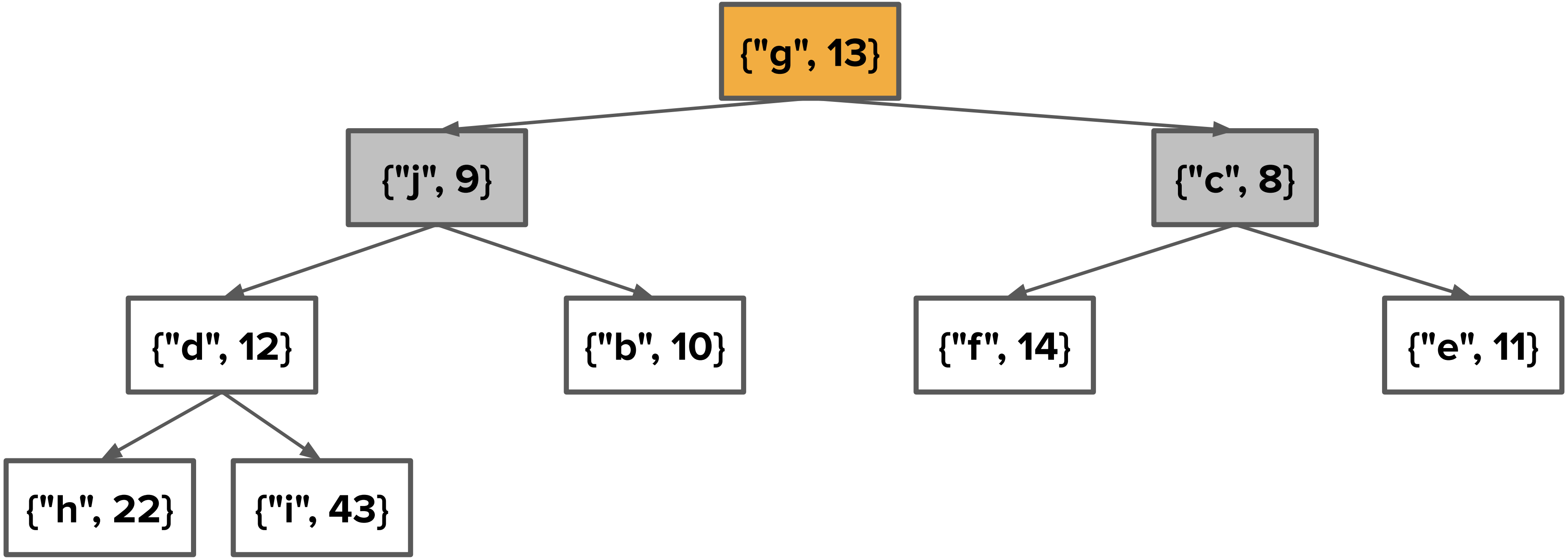
{"g", 13}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

dequeue()

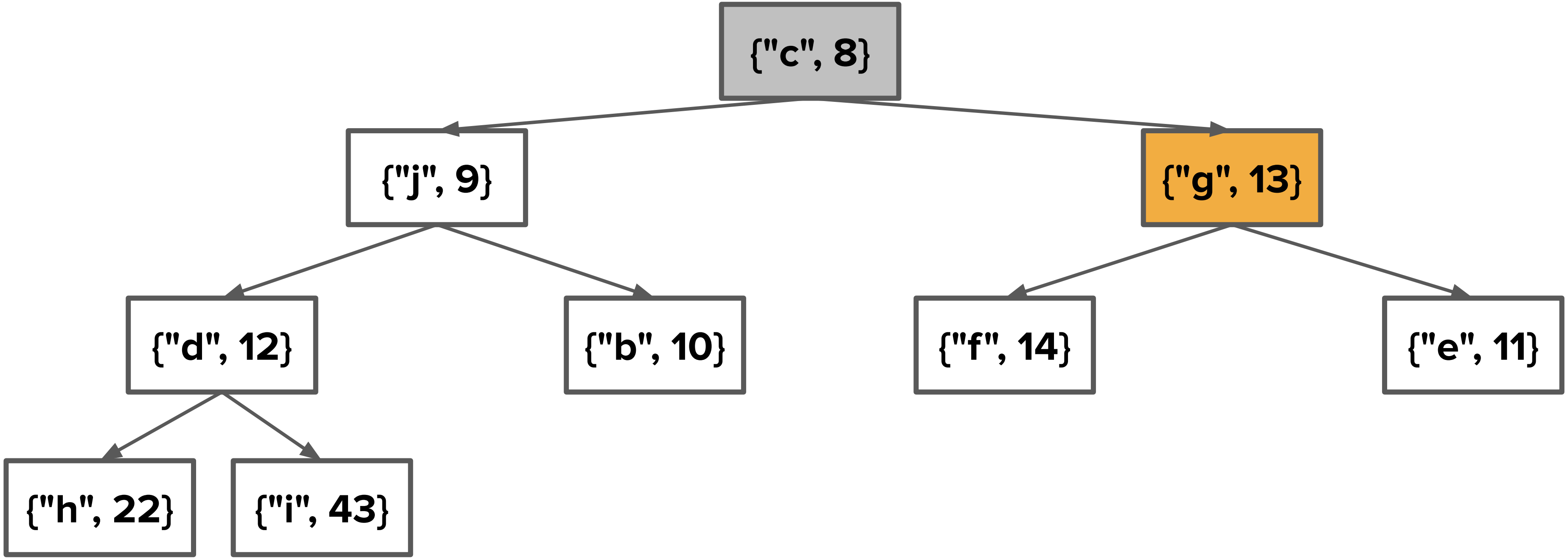
- Remove the minimum element: the root of the tree.

- Replace the root with the "last" element in our tree (last level, farthest right) since we know that location will end up empty.

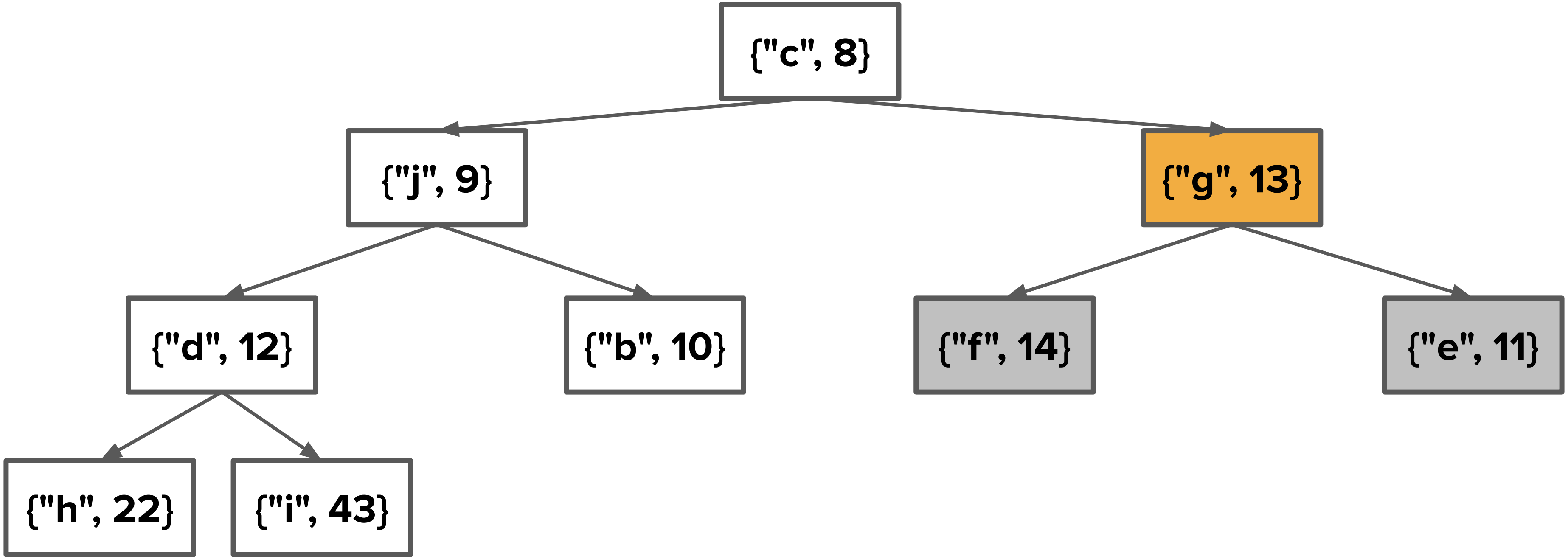
- **Bubble down** to regain the *heap property*!
  - Compare the moved element to its new children.
    - If one of the two children is smaller, swap with that child.
    - If both of the children are smaller, swap with the one that's smaller.
  - Repeat until you no longer bubble down or there are no more children to compare against.



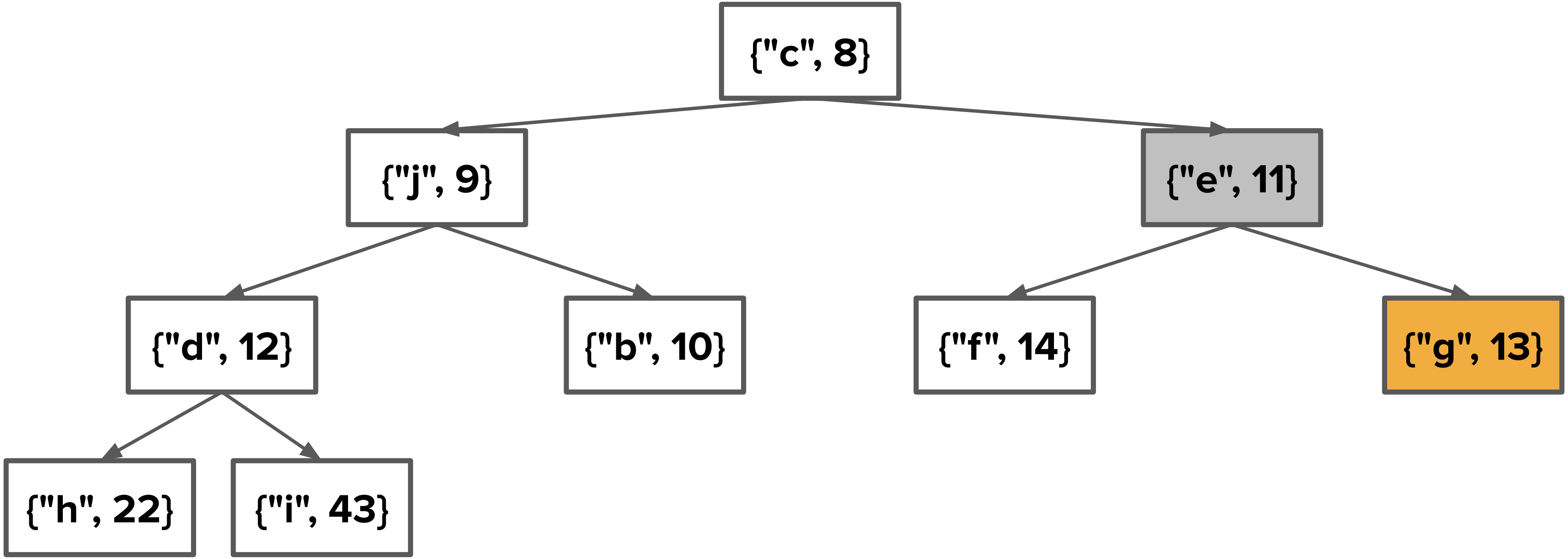
{"g", 13}	{"j", 9}	{"c", 8}	{"d", 12}	{"b", 10}	{"f", 14}	{"e", 11}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11



{"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

`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

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)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Summary

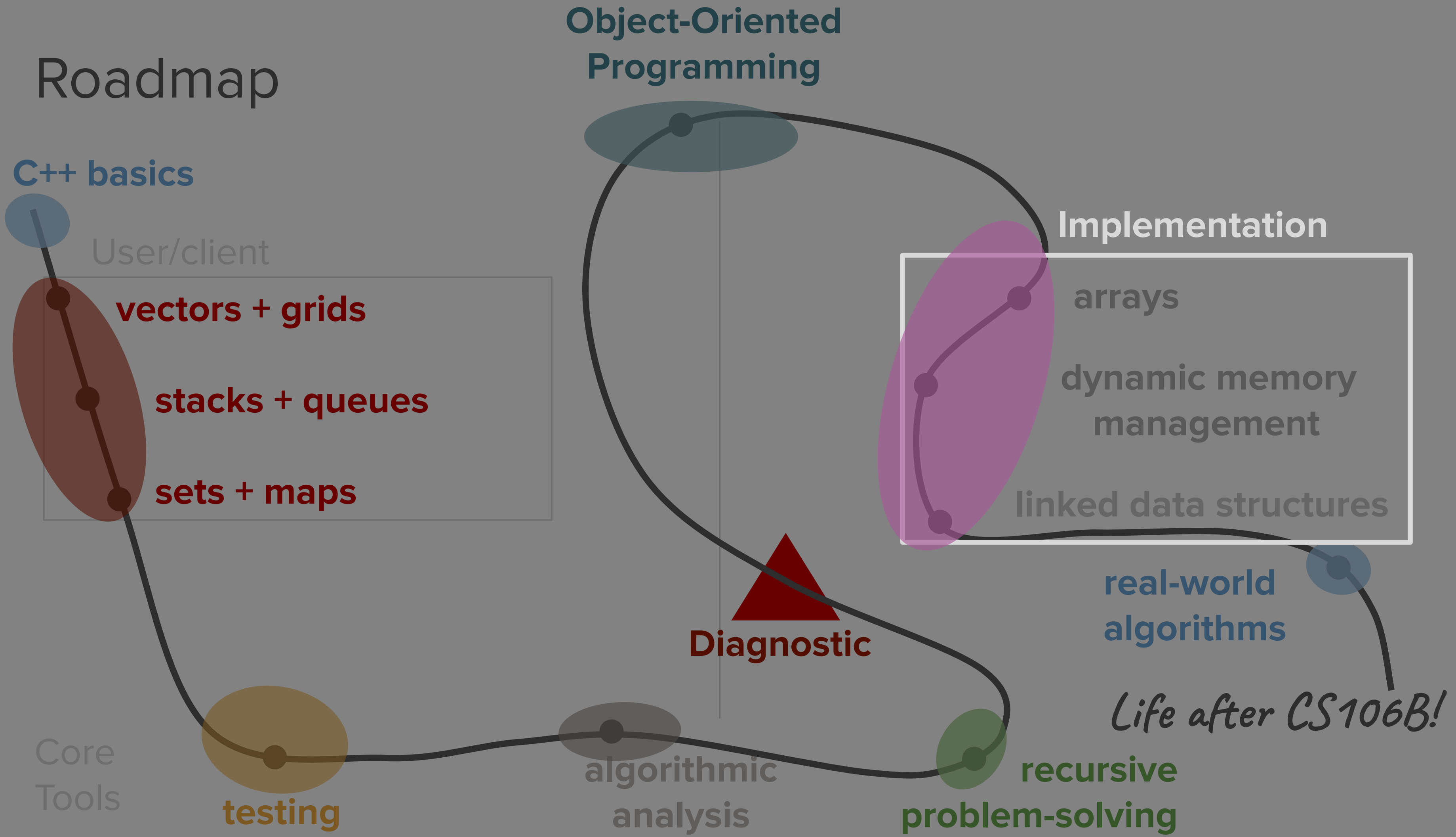
- **Priority queues** are queues ordered by **priority** of their elements, where the **highest priority** elements get dequeued first.

- **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.

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
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Memory and Pointers