Adaptable Priority Queues

CS2515

Many real world queues are not FIFO ...

Hospital waiting lists

- patients with life-threatening illness will be moved up the queue

Air traffic control

- airplanes with low fuel will be landed first

Access nodes forwarding packets in (e.g.) LTE networks

- packets from voice calls preferred over buffered video

Manufacturing scheduling

- jobs with closest due dates are preferred

CS2515

The Element

Items will now be stored with two pieces of data:

- the *value*, representing the original item
- the *key*, representing its priority value

Any data type will do for the keys, as long as we can compare them.

By convention, lower keys represent higher priority elements.

```
class Element:
    def __init__(self, key, value):
        self._key = key
        self._value = value

    def __eq__(self, other):
        return self._key == other._key

    def __lt__(self, other):
        return self._key < other._key
```

The Priority Queue ADT

add(key, value)	add a new element into the priority queue
min()	return the value with the minimum key
remove_min()	remove and return the value with the minimum key
is_empty()	return True if no items in the priority queue
length()	return the number of items in the priority queue

Priority queue: implementation complexity:

CS2515

	add(k,v)	min()	remove_min()	is_empty() length()	build full PQ
unsorted list	O(1)* append(E(k,v))	O(n)	O(n)	O(1)	O(n)
unsorted DLL	O(1) add at end	O(n)	O(n)	O(1)	O(n)
sorted list	O(n)	O(1)	O(1) min at end	O(1)	$O(n^2)$
sorted DLL	O(n)	O(1)	O(1)	O(1)	$O(n^2)$
AVL tree	O(log n)	O(log n)	O(log n)	O(1)	O(n log n)
Binary heap	O(log n)*	O(1)	O(log n)	O(1)	O(n log n) or O(n)

Hospital waiting lists
- patients with life-threatening illness will be moved up the queue

In practice, the priority of patients will change, as their illnesses improve or deteriorate. Some patients may leave the waiting list ...

Can we do this in our existing PriorityQueue implementations?

We need to enable:
- updating the key of an item
- reading the current key of an item
- removing an item

with reasonable time-complexity.

The PriorityQueue ADT does not give us the flexibility we need.

- It only give us access to the item with minimum key.
- we have no way of changing the key of an item.
- If we could change the key, or remove an item, some of our implementations of the PriorityQueue would then be inconsistent
    - e.g. the heap property might be violated

We need a new ADT for *Adaptable* Priority Queues.

We could implement an internal dictionary behind the ADT which gives fast access, but it requires the external code to provide hashable identifiers for the items as well as the key (i.e. priority) and the item itself.

To be able to change the priority of an item, or remove an item from the APQ, we need some way of locating its Element in the APQ.

We will return a reference to the Element when we add an item, and then the calling program can store it and get access later

Getting a reference to the Element is not enough

- in the list implementations, we need to find the list cell pointing to the element when we want to remove it
- in linked data structures, we need references to the 'Node' in the structure

We will modify the Element class to represent some location information, relevant to the implementation of the APQ.

We need to make sure we maintain that information consistently when we do any operations in the APQ

The *Adaptable* Priority Queue ADT

add(key, item)	add a new item into the priority queue with priority key, and return its Element in the APQ
min()	return the value with the minimum key
remove_min()	remove and return the value with the minimum key
is_empty()	return True if no items in the priority queue
length()	return the number of items in the priority queue
update_key(element, newkey)	update the key in element to be newkey, and rebalance the APQ
get_key(element)	return the current key for element
remove(element)	remove the element from the APQ, and rebalance APQ

```python
class Element:
    """ A key, value and index. """

    def __init__(self, k, v, i):
        self._key = k
        self._value = v
        self._index = i

    def __eq__(self, other):
        return self._key == other._key

    def __lt__(self, other):
        return self._key < other._key

    def _wipe(self):
        self._key = None
        self._value = None
        self._index = None
```

For the implementations using an array-based heap or an unsorted list

APQ as an unsorted list

		Complexity?
add(key, item)	add a new item into the priority queue with priority key, and return its Element in the APQ	
- create the Element, append to the list, return the Element		O(1)*
min()	return the value with the minimum key	
- linear search of list for the Element with minimum key, return key,value		O(n)
remove_min()	remove and return the value with the minimum key	
- linear search of list for Element with min key, swap into last place, pop the element, return key,value		O(n)

APQ as an unsorted list (cont)

Complexity?

update_key(element, newkey)
update the key in element to be newkey, and rebalance the APQ

-update the element's key — O(1)

get_key(element) return the current key for element

-return the element's key — O(1)

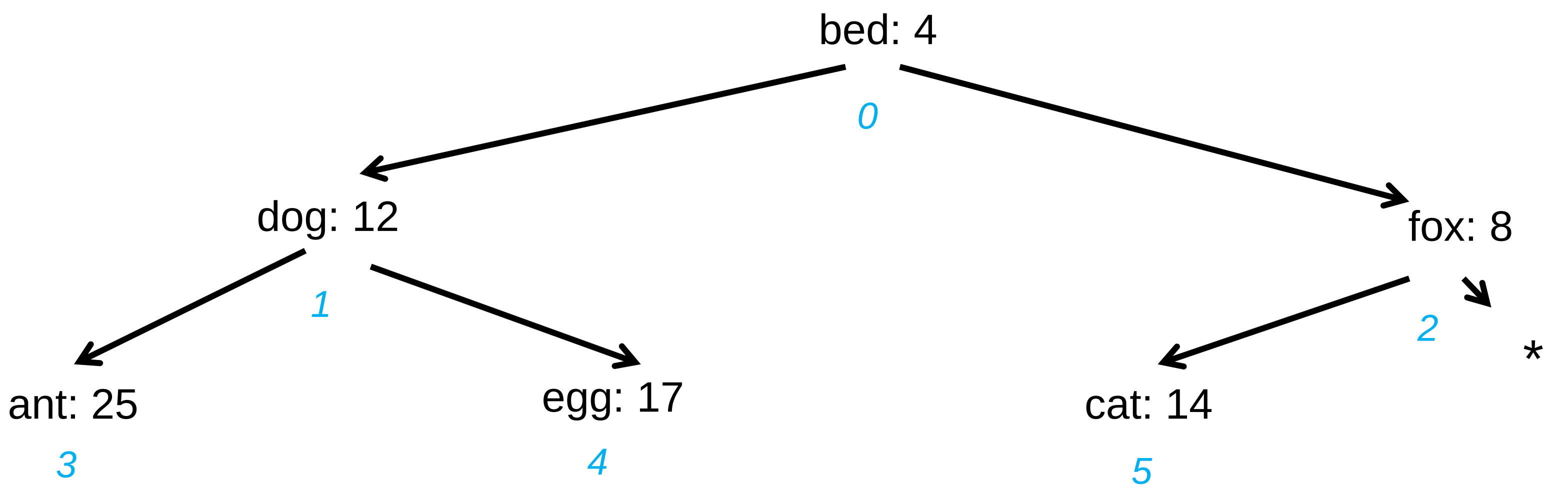
remove(element) remove the element from the APQ, and rebalance APQ

- get the element by its index, swap into last place, pop the last element, return key,value of popped element — O(1)*

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update ant to have key = 3?

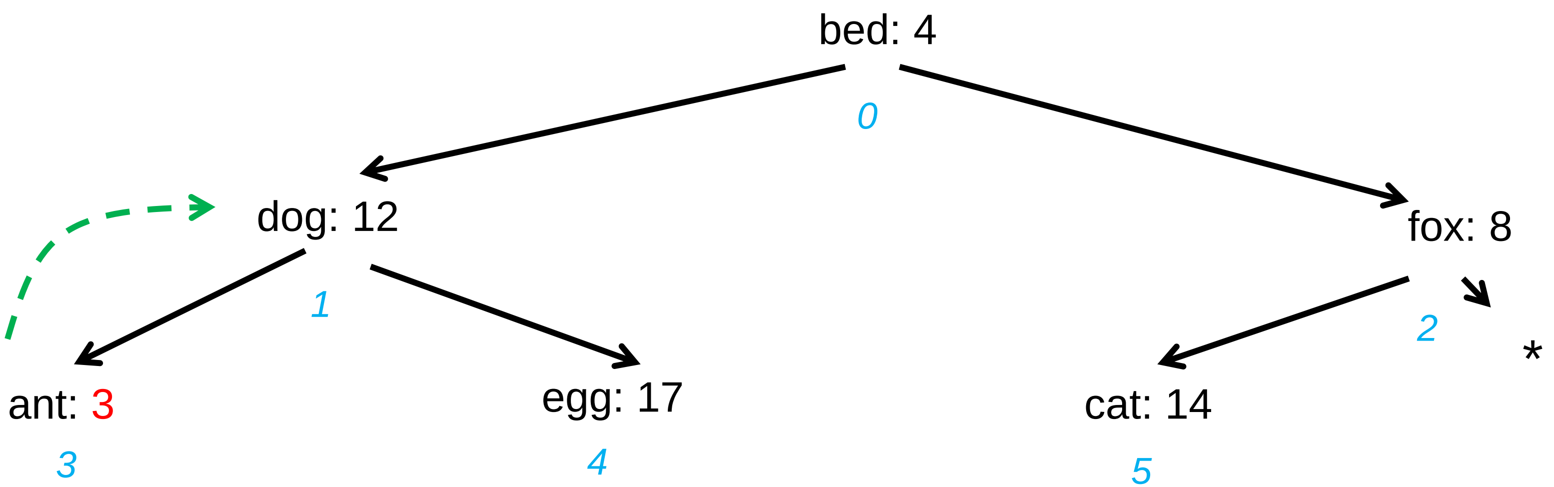


bed: 4 (0)	dog: 12 (1)	fox: 8 (2)	ant: 25 (3)	egg: 17 (4)	cat: 14 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update ant to have key = 3?

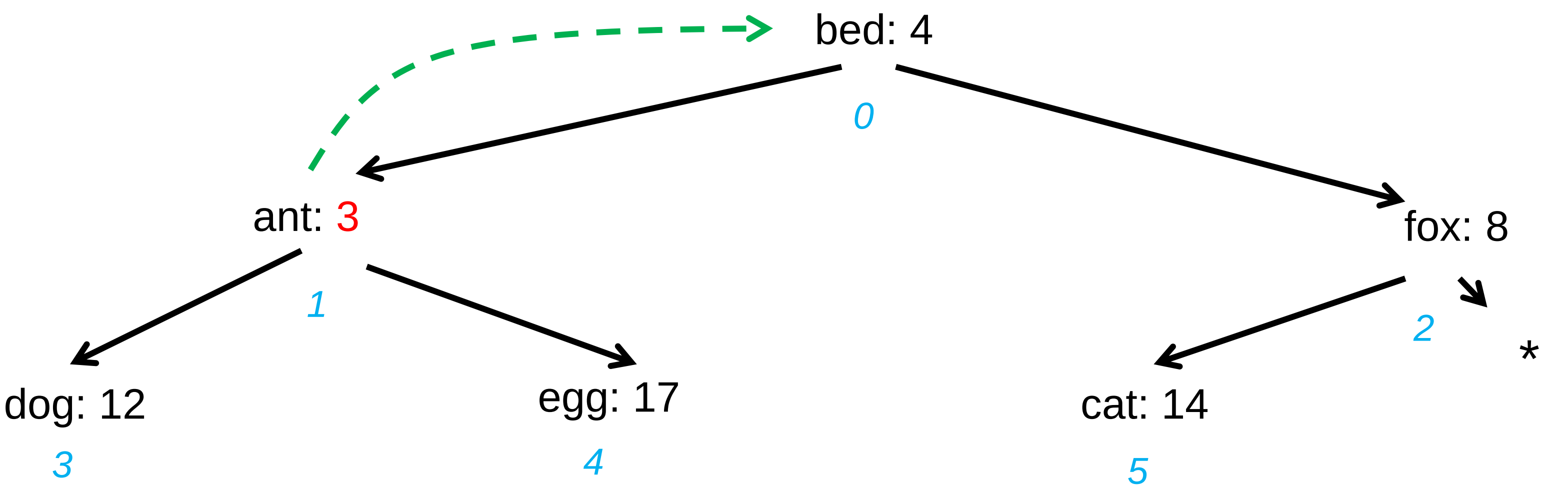


bed: 4 (0)	dog: 12 (1)	fox: 8 (2)	ant: 3 (3)	egg: 17 (4)	cat: 14 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update ant to have key = 3?

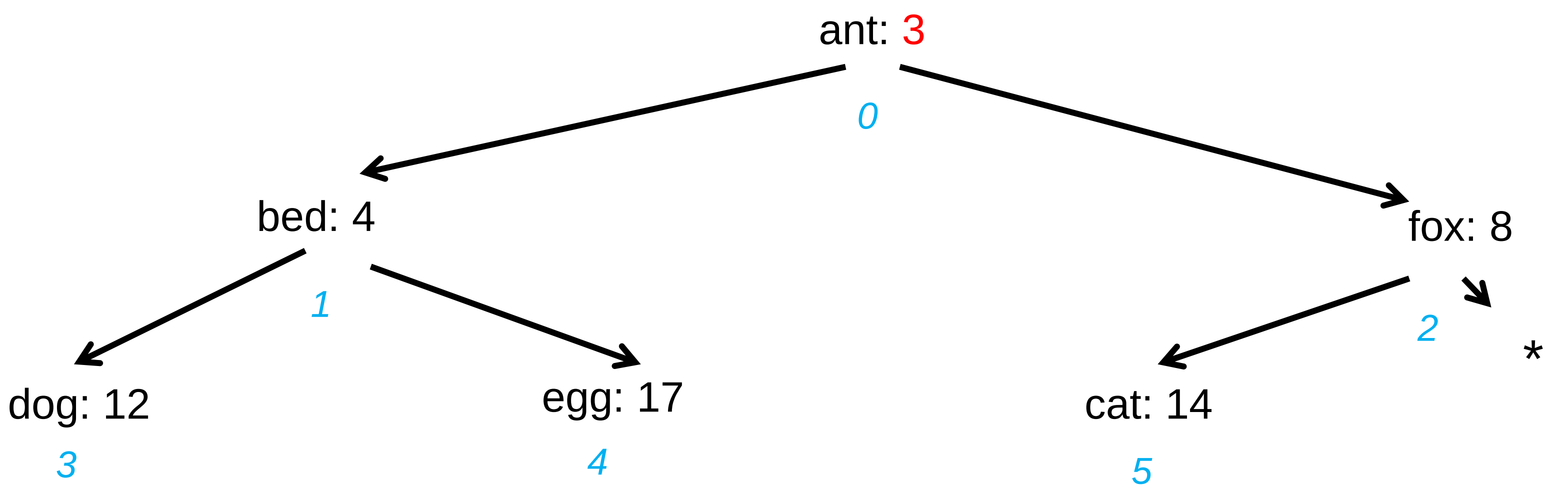


bed: 4 (0)	ant: 3 (1)	fox: 8 (2)	dog: 12 (3)	egg: 17 (4)	cat: 14 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update ant to have key = 3?

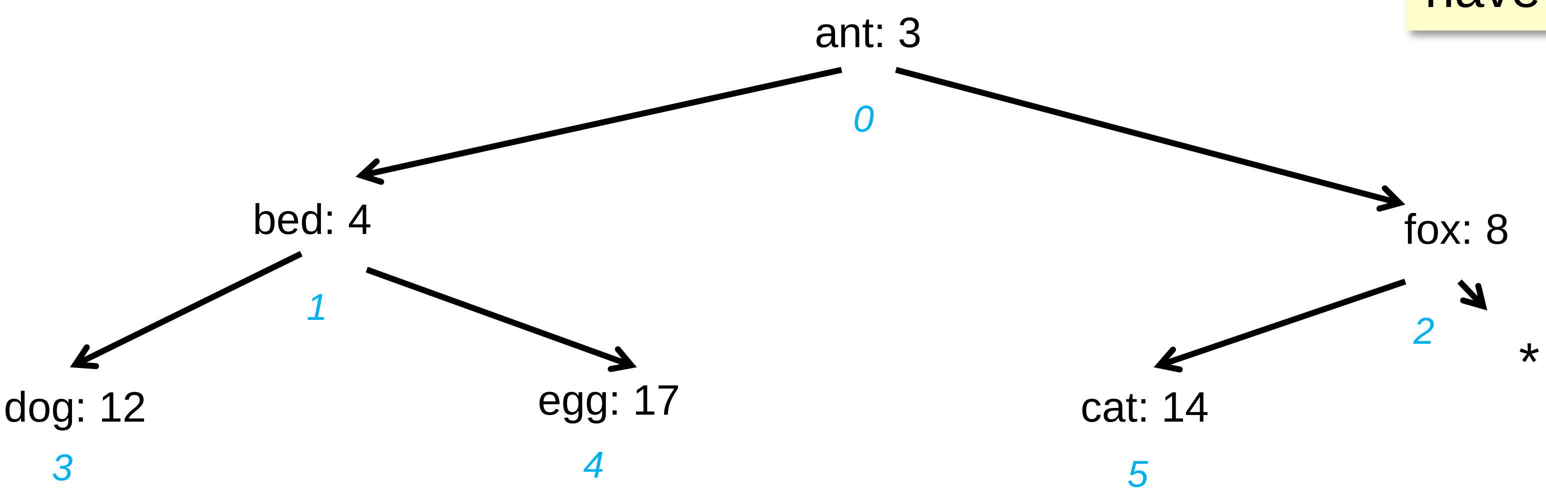


ant: 3 (0)	bed: 4 (1)	fox: 8 (2)	dog: 12 (3)	egg: 17 (4)	cat: 14 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update fox to have key = 15?

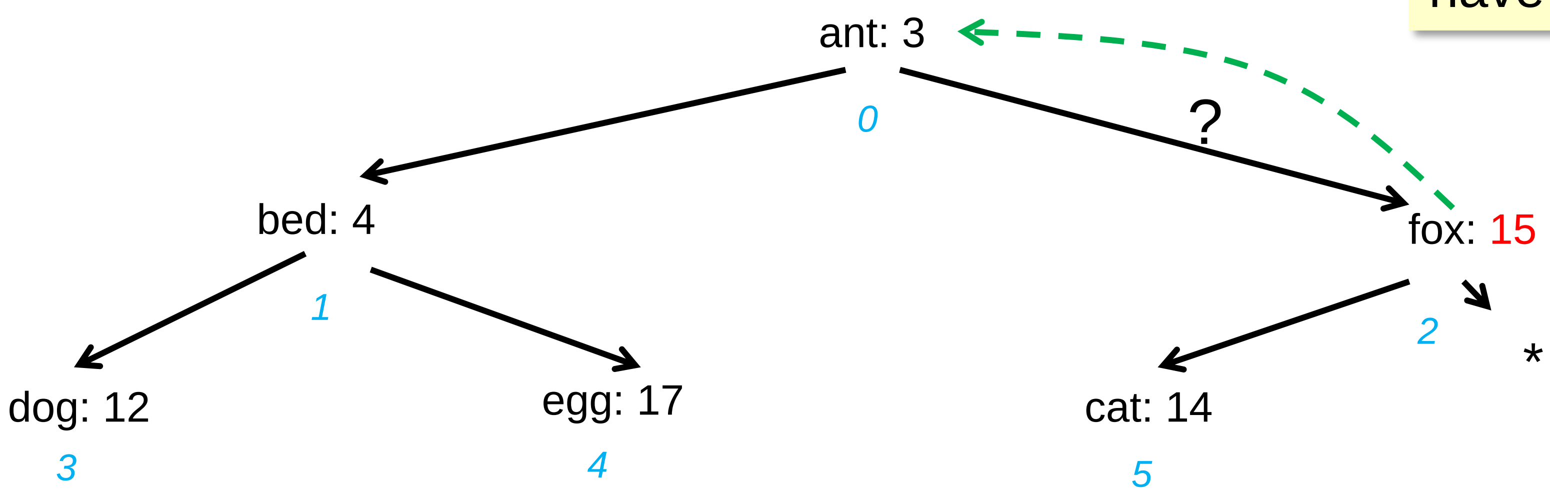


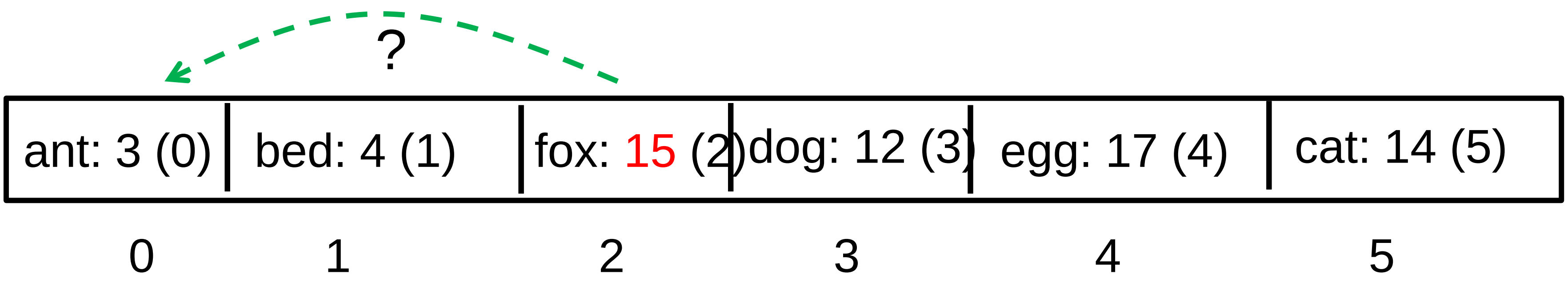
ant: 3 (0)	bed: 4 (1)	fox: 8 (2)	dog: 12 (3)	egg: 17 (4)	cat: 14 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update fox to have key = 15?

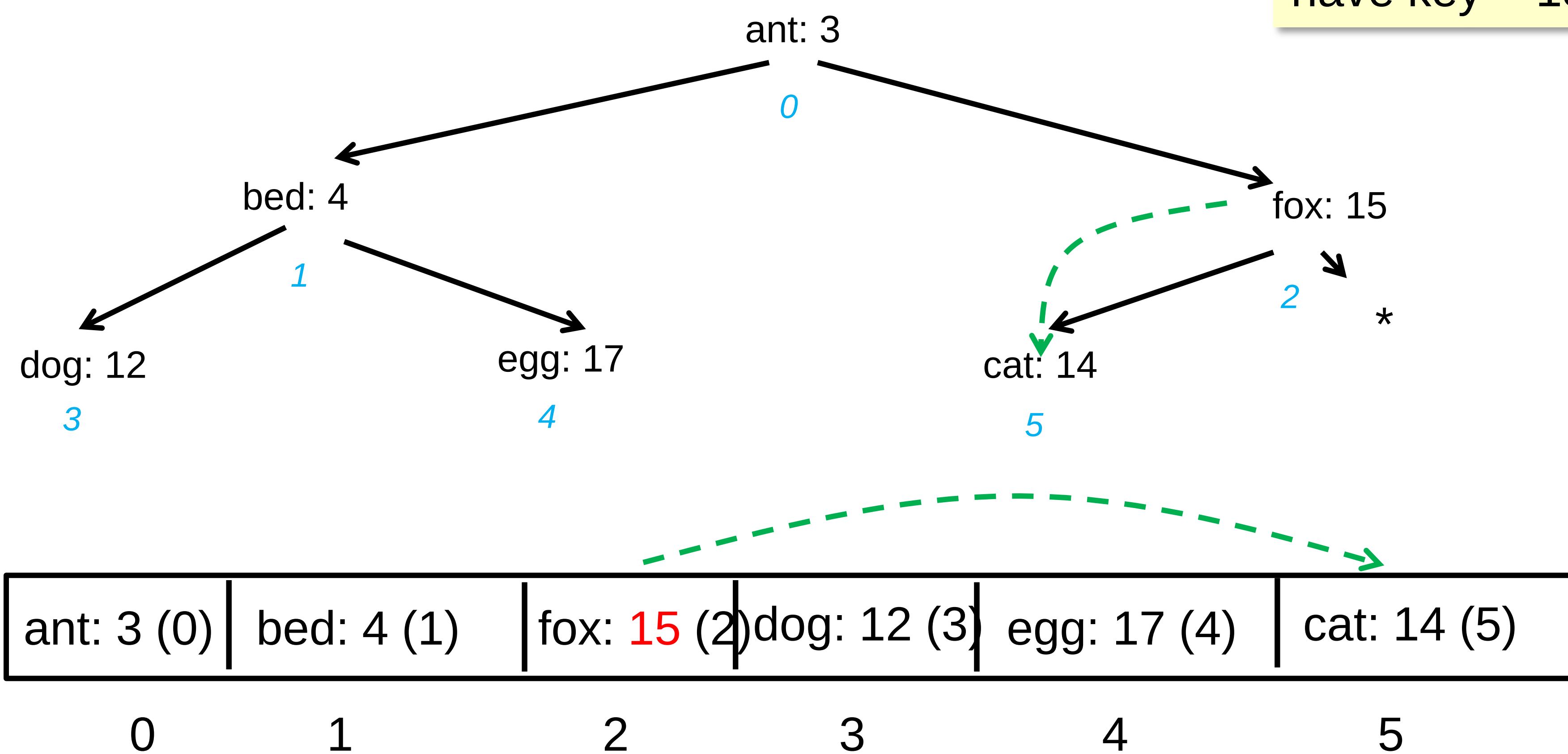




Implement as an array-based heap?

Complete binary tree
key of parent < key of child

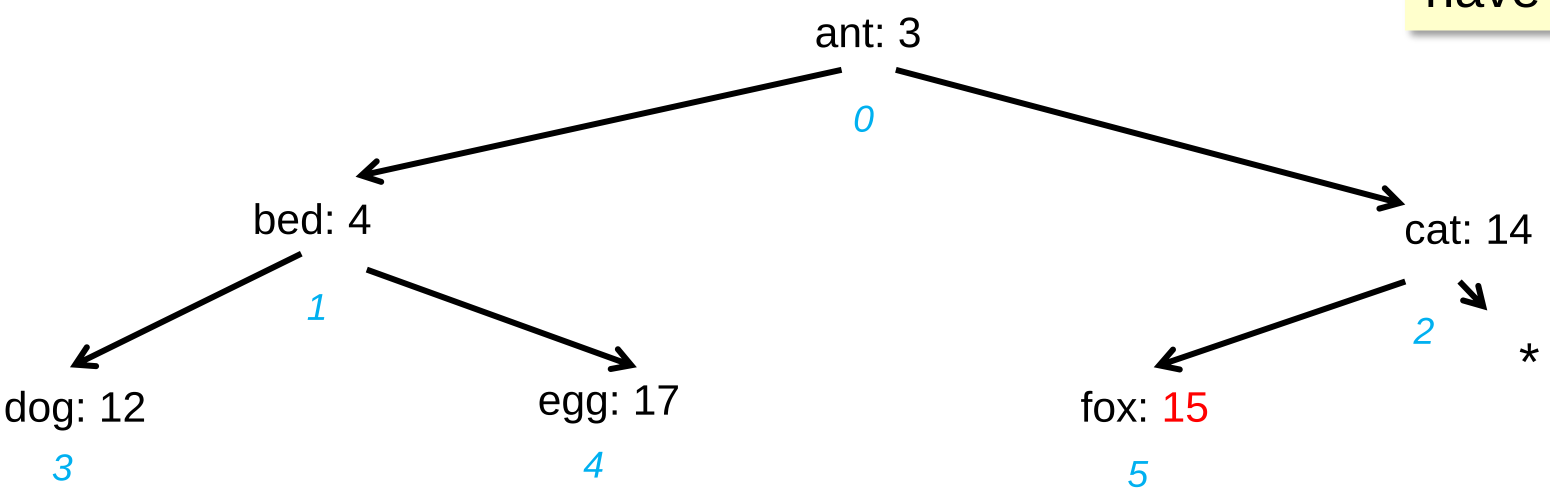
Update fox to
have key = 15?



Implement as an array-based heap?

Complete binary tree
key of parent < key of child

Update fox to have key = 15?

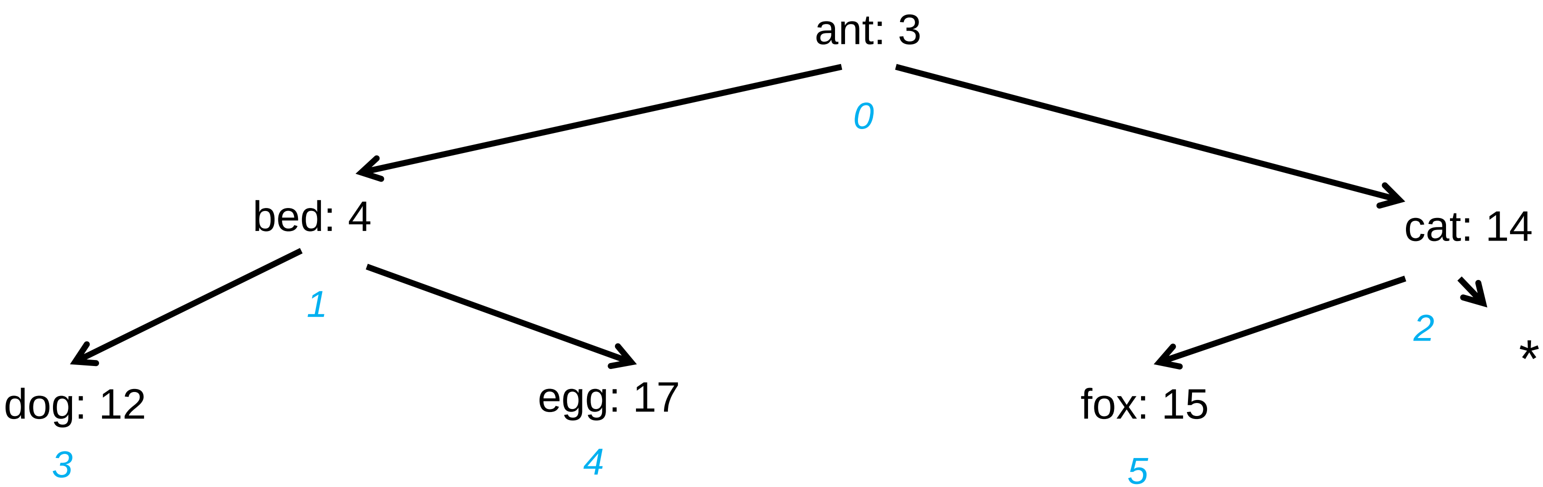


ant: 3 (0)	bed: 4 (1)	cat: 14 (2)	dog: 12 (3)	egg: 17 (4)	fox: 15 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

remove bed?

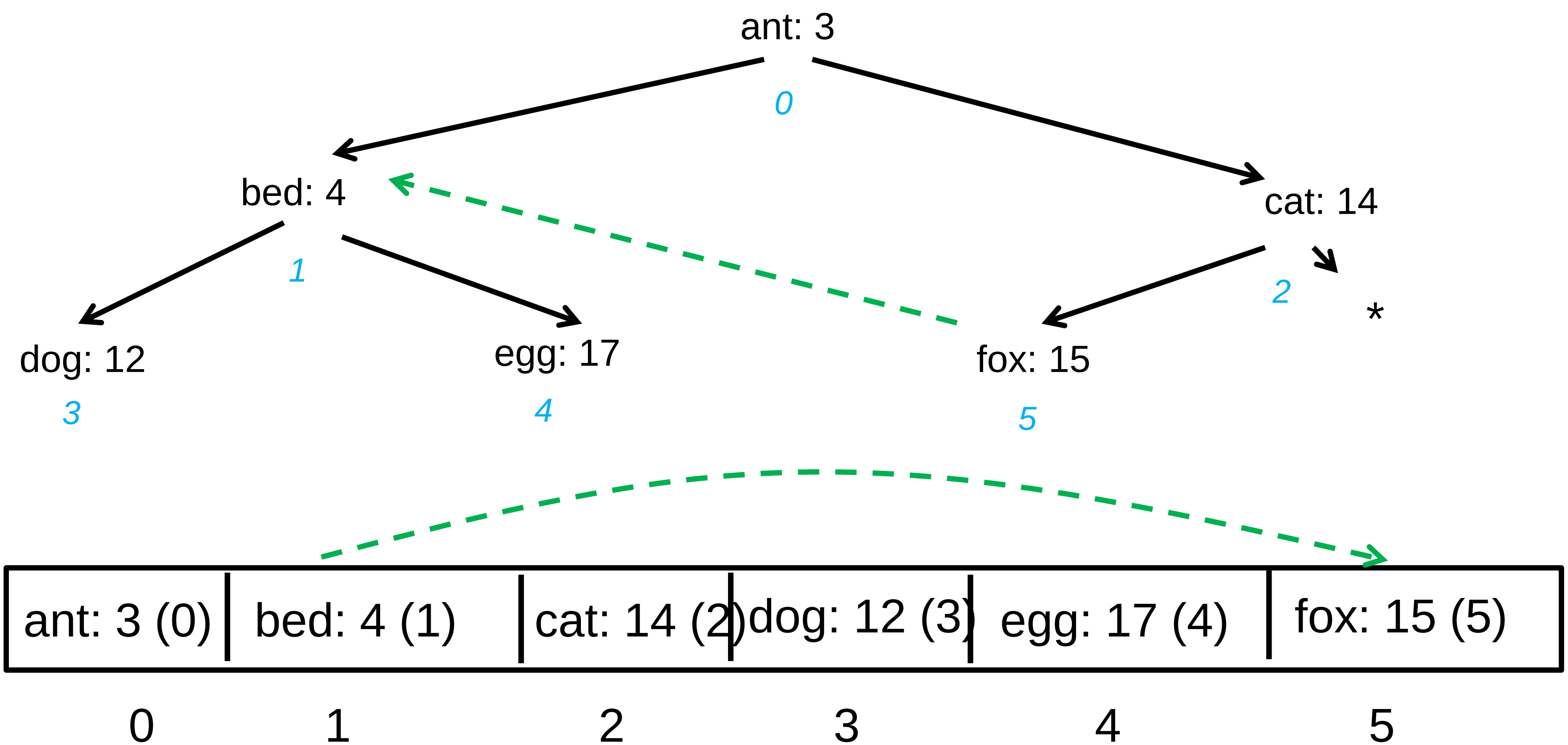


ant: 3 (0)	bed: 4 (1)	cat: 14 (2)	dog: 12 (3)	egg: 17 (4)	fox: 15 (5)
0	1	2	3	4	5

Implement as an array-based heap?

Complete binary tree
key of parent < key of child

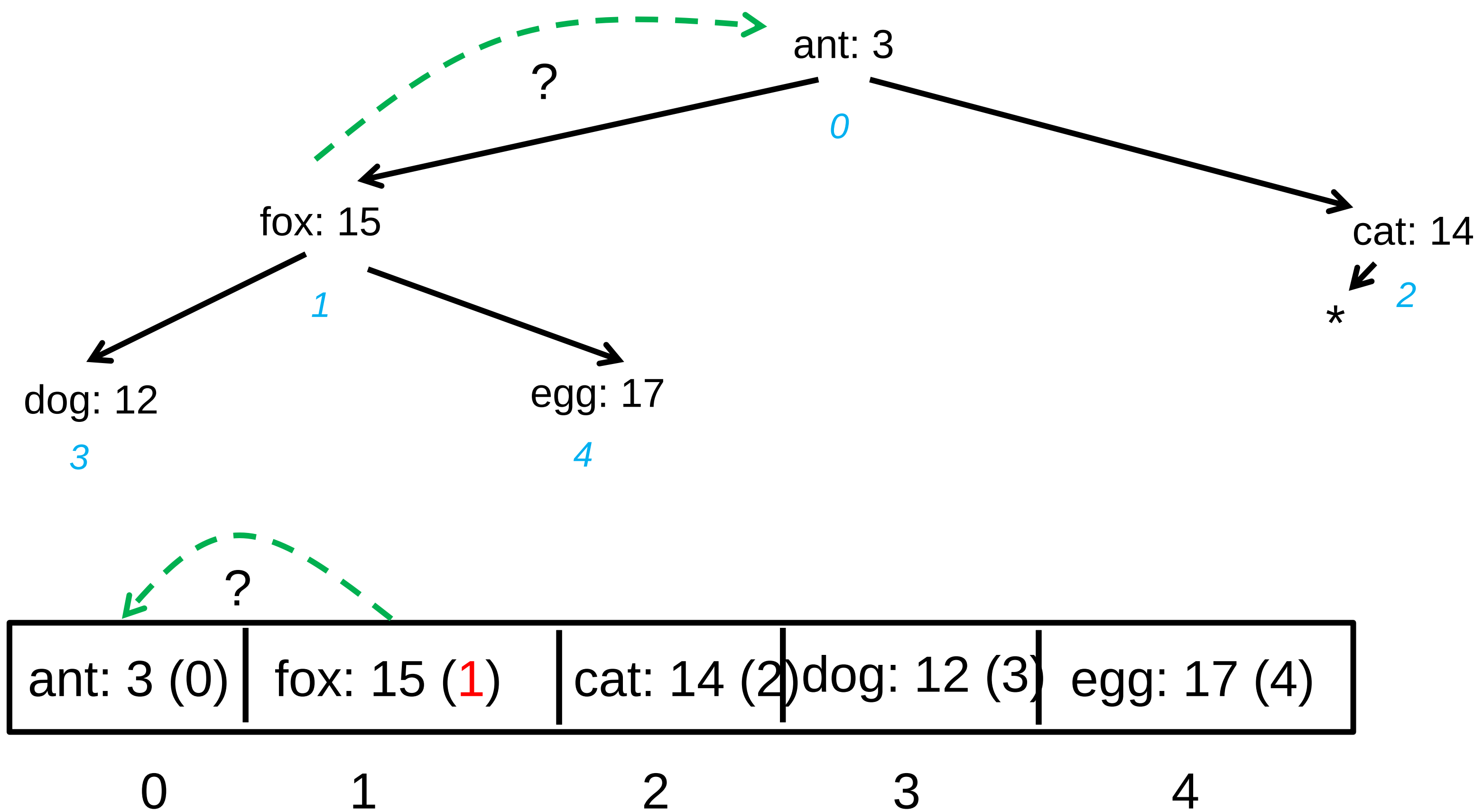
remove bed?



Implement as an array-based heap?

Complete binary tree
key of parent < key of child

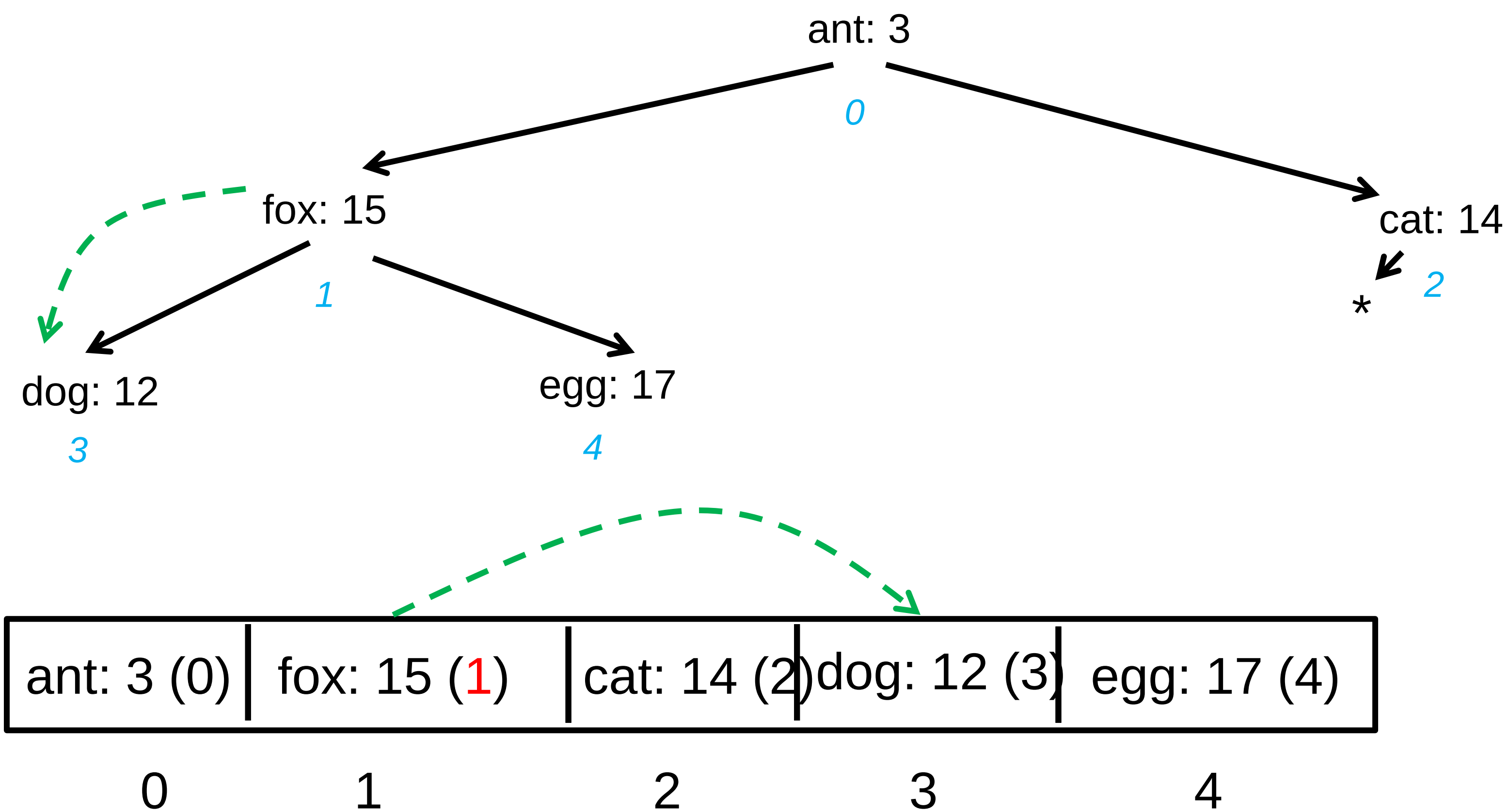
remove bed?



Implement as an array-based heap?

Complete binary tree
key of parent < key of child

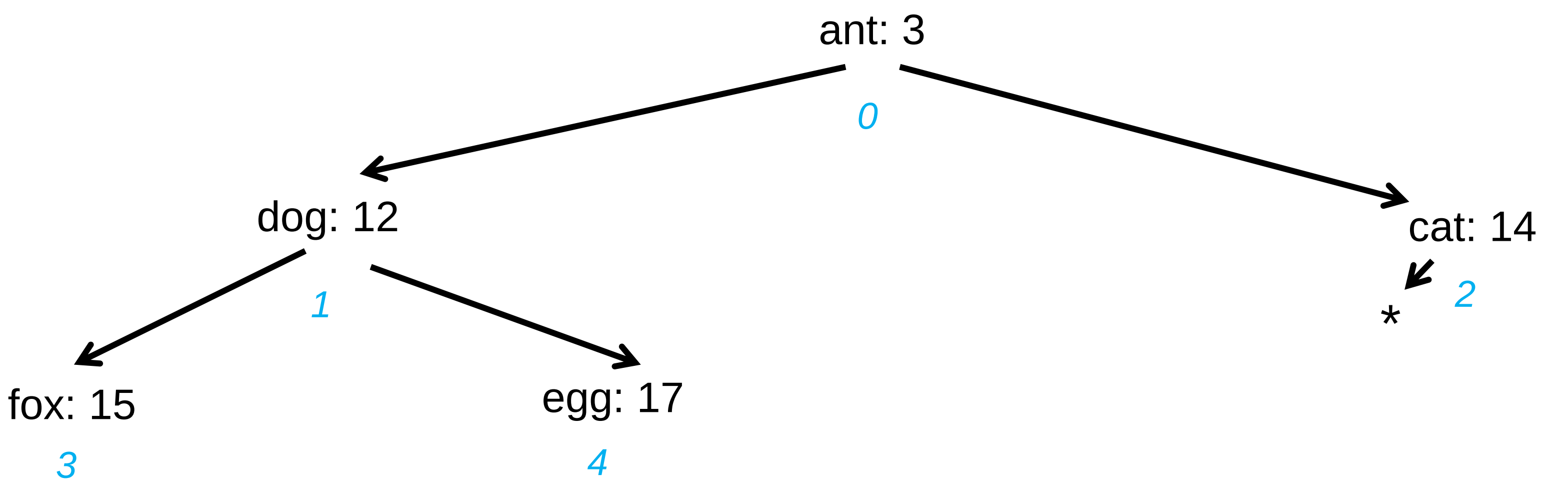
remove bed?



Implement as an array-based heap?

Complete binary tree
key of parent < key of child

remove bed?



0	1	2	3	4
ant: 3 (0)	dog: 12 (1)	cat: 14 (2)	fox: 15 (3)	egg: 17 (4)

APQ as an array-based binary heap

		Complexity?
add(key, item)	add a new item into the priority queue with priority key, and return its Element in the APQ	
- create the Element with index of last place, add to heap, return the Element		O(log n)*
min()	return the value with the minimum key	
- read first cell in array and return		O(1)
remove_min()	remove and return the value with the minimum key	
- swap first element into last place, pop the element, bubble top element down, return popped key,value		O(log n)*

APQ as an unsorted list (cont)

Complexity?

update_key(element, newkey)
update the key in element to be newkey, and rebalance the APQ

-update the element's key, if key less than parent's, bubble up; else bubble down

O(log n)

get_key(element) return the current key for element

-return the element's key

O(1)

remove(element) remove the element from the APQ, and rebalance APQ

O(log n)*

-swap last element with one in element's index,
if swap key < parent, bubble up; else bubble down.
Pop the last element, return key,value

BUT:

whenever we swap two elements in the heap (during a swap or a bubble up or bubble down)

we must update the _index attributes in the Elements.

Next lecture

Shortest paths in weighted graphs