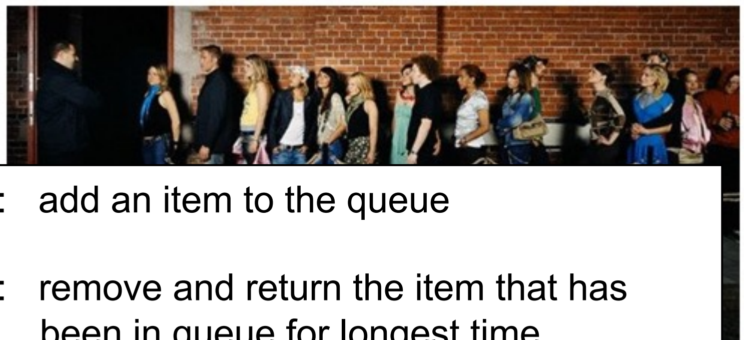
The Priority Queue ADT

Modelling 'queues' where items are selected in order of priority

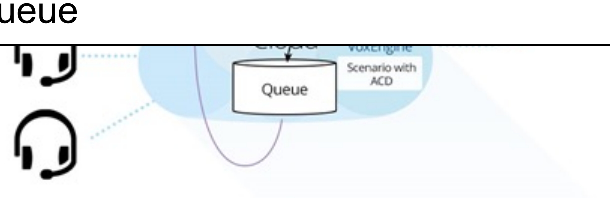
From Lecture 6 ...

A *queue* is a collection of objects, where :
- if we want to take an item, we take it from the *front*
- if we want to add an item, we add it onto the *back*

A queue is *first-in, first-out (FIFO)*



enqueue:	add an item to the queue
dequeue:	remove and return the item that has been in queue for longest time
front:	report the item that has been in queue for longest time
length:	report how many elements are in the queue



Many real world queues are not FIFO ...

Hospital waiting lists
- patients with critical illness will be placed towards the front of the list

Priority Queue: element with highest priority is removed next

Air traffic control
- airplanes with low fuel will be landed first

Access nodes forwarding packets in (e.g.) 4G networks
- packets from voice calls are prioritised over buffered video

Manufacturing scheduling
- jobs with closest due dates are preferred

The Element

Items will now be stored with two pieces of data:

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

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

By convention, lower keys represent higher priority elements.

when are two elements 'the same' in terms of priority ...

when is this element more 'important' ...

```
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
length()	return the number of items in the priority queue

i.e. the top priority item

Challenge: design a PQ implementation

Design an implementation for the Priority Queue ADT, using data structures and techniques we have seen already in the module.

You must support the 4 specified methods of 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
- length() return the number of items in the priority queue

Your solution should be efficient – pay attention to each of the methods, and consider the cost of maintaining the structure over long sequences of operations.

(In an exam question, you would be told: ‘marks will be given for correctness, clarity of description, and efficiency of the implementation’)

Implementations

There are many different ways to implement Priority Queues using the structures and ADTs we have seen already:

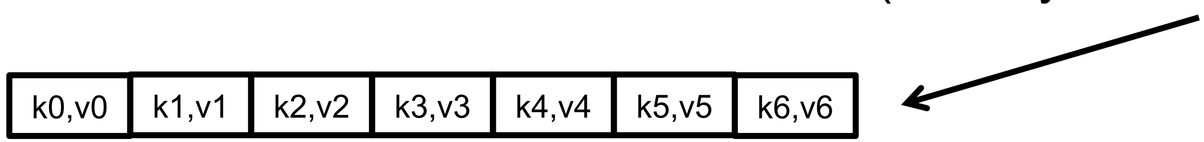
- unsorted Python list
- unsorted doubly linked list (or SLL)
- sorted Python list
- sorted doubly linked list
- binary search tree
  - assuming we can extend to allow duplicate items
- avl trees

how would we do this?

Using an unsorted Python list

Using an unsorted Python list

add(k,v): append here
without worrying about key
(with Python list resizing)



min(): search the unordered list to find the minimum key

remove_min(): search, then pop(i) (with the cost of Python repairing the list)

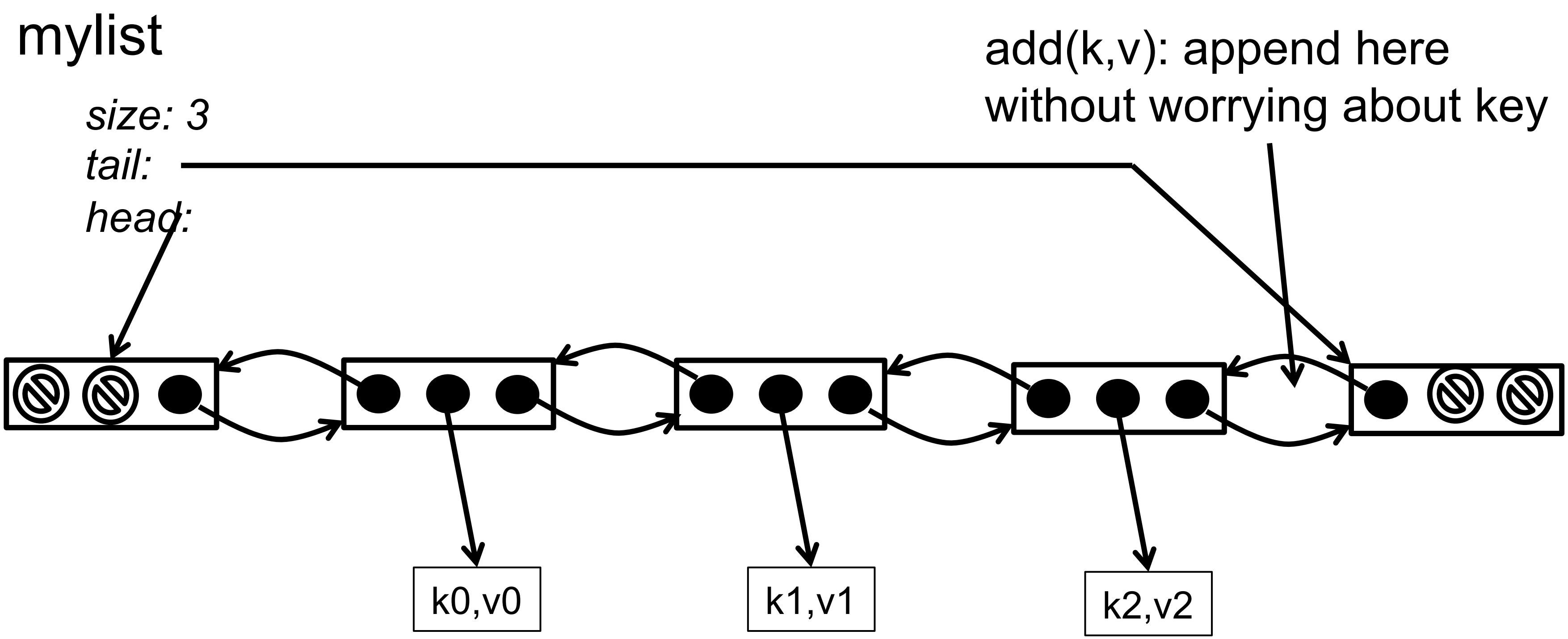
Do you need to pop(i)?

Complexity

	add(k,v)	min()	remove_min()	length()	build full PQ
unsorted list	O(1)* append(E(k,v))	O(n)	O(n)	O(1)	O(n)

Using an unsorted DLL

Using an unsorted DLL



min(): search the unordered list to find the minimum key
remove_min(): search, then remove

Complexity

	add(k,v)	min()	remove_min()	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)

Using a sorted Python list

Using a sorted Python list

add(k,v): find place and
insert (with Python list repair)
highest priority at end

k0,v0	k1,v1	k2,v2	k3,v3	k4,v4	k5,v5	k6,v6

min(): return the last item

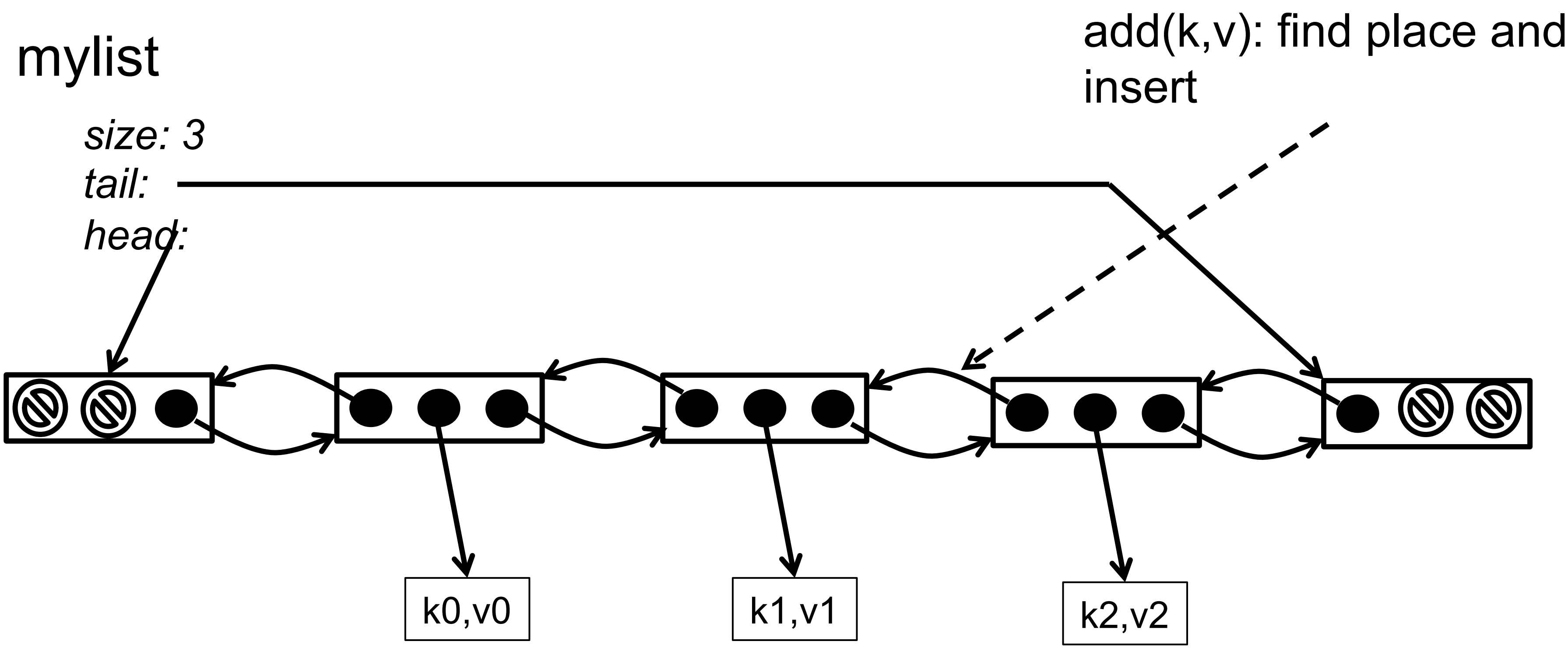
remove_min(): remove and return the last item (no list shuffling)

Complexity

	add(k,v)	min()	remove_min()	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)

Using a sorted DLL

Using a sorted DLL



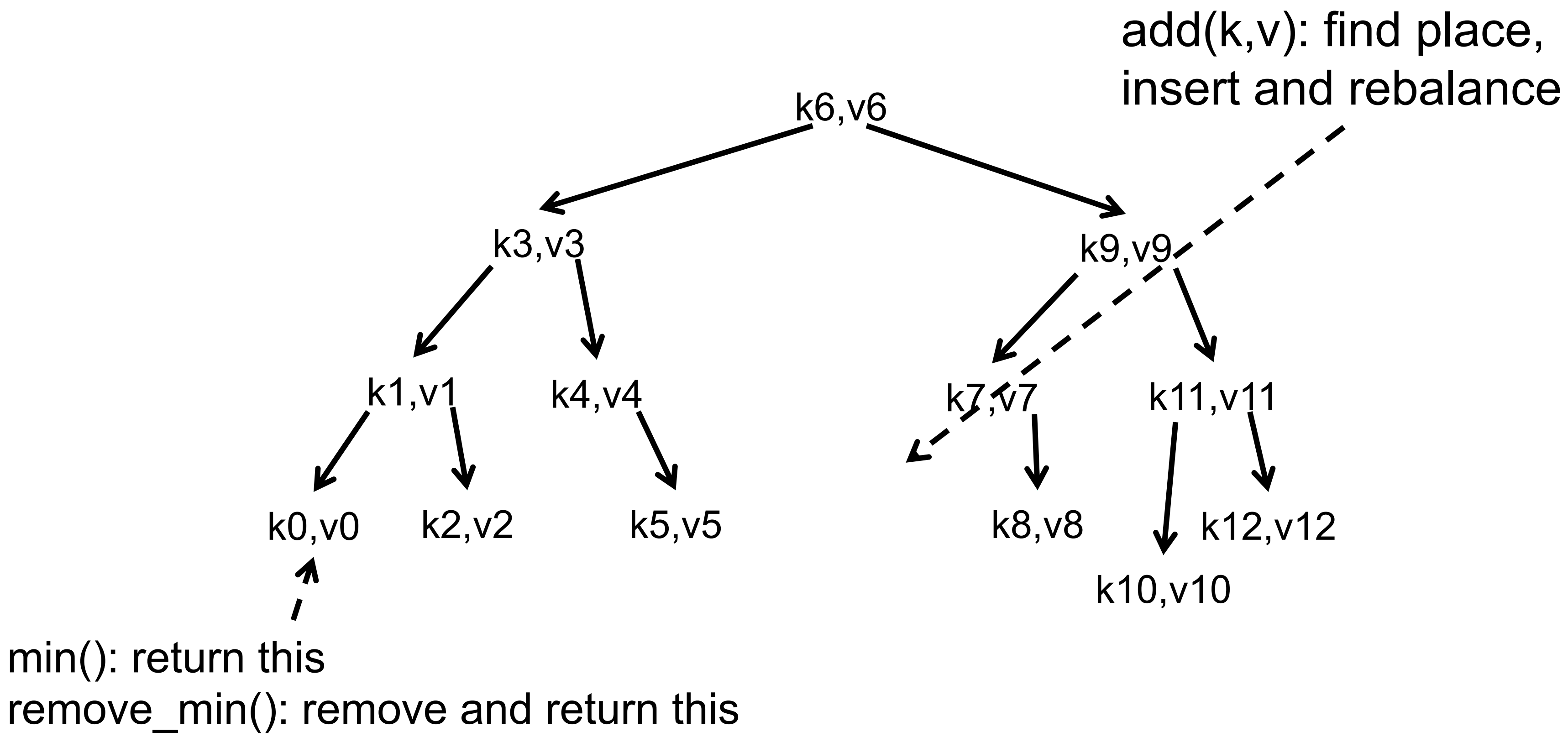
min(): return the first item
remove_min(): remove and return the first item

Complexity

	add(k,v)	min()	remove_min()	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)

Using an AVL tree

Using an AVL tree



Complexity

	add(k,v)	min()	remove_min()	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)

can we do any better?

Next Lecture

The binary heap:
an efficient implementation of a priority queue