

COMP_SCI 214: Data Structures and Algorithms

The Dictionary ADT

PROF. SRUTI BHAGAVATULA

Announcements

- ▶ HW2 self-eval is live
 - ▶ If you aren't happy with how HW1 self-evaluation went, don't worry that's one of the harder ones to give you an idea of how they go; you still have 4 left!
- ▶ HW2 feedback is out and resubmission is live
- ▶ Worksheets due Thursday
- ▶ HW3 to be released Thursday
- ▶ Exam 1 is next Thursday: topics include Lecture 1 until Lecture 7 (this Thursday)
 - ▶ More details and practice exam later this week

Another ADT

Motivating scenario

- ▶ I have an address book with all my friends' information (address, phone number, birthday, etc.). I'd like to be able to store all my friends' information quickly and also lookup a friend's information easily.
- ▶ How can I do this?

What do we need to implement this?

- ▶ A way to keep track of friends' names and store their information "next" to their names
- ▶ A way to just look for a friend's name and get all their information
- ▶ A way to remove a friend's information from our address book if they're no longer our friend
- ▶ A way to check if we already have a friend's information
- ▶ The friend and their information can be represented as some kind of mapping

More generally

- ▶ We need a program to store a mapping that allows us the following operations:
 - ▶ Insert data (value) against a unique ID (key)
 - ▶ Look up a key and get the value
 - ▶ Delete a key and its value
 - ▶ Check if we have a key
- ▶ Not sure what data structure to use yet, but we know what we need from the program
 - ▶ Seems reminiscent of an Abstract Data Type (ADT)?

Recall: What is an Abstract Data Type?

- ▶ An ADT defines:
 - ▶ A set of (abstract) objects or values
 - ▶ A set of (abstract) operations on those values
- ▶ An ADT omits:
 - ▶ How the values are concretely represented (data type, layout, etc.)
 - ▶ How the operations actually work

ADTs so far

► Stack

- Push item onto top of stack
- Pop the top item from stack
- Check if the stack is empty



► Queue

- Add item to the end of queue (enqueue)
- Remove item from front of queue (dequeue)
- Check if the queue is empty



The next ADT: Dictionary

- ▶ Dictionaries map *keys* to *values*
 - ▶ Like in an actual dictionary (words → definitions)
 - ▶ Like what python `dict()` does
- ▶ Operations:
 - ▶ Insert data (value) against a unique ID (key) in a dictionary
 - ▶ Look up a key in the dictionary and get its associated value
 - ▶ Delete a key and its value from the dictionary
 - ▶ Check if a key exists in the dictionary
- ▶ Other requirements we may decide to set:
 - ▶ Keys must be distinct
 - ▶ There is no ordering among keys



ADT: Dictionary

- ▶ **Abstract values** look like:
 - ▶ {'Aparna': 'SF', 'Maria': 'Chicago', 'Aditi': 'Bengaluru'}
- ▶ How do we specify an ADT in DSSL2?
 - ▶ Interface with contracts:

```
interface DICT[K, V]: # key and value types up to client
  def mem?(self, key: K) -> bool?
  def get(self, key: K) -> V
  def put(self, key: K, value: V) -> NoneC
  def del(self, key: K) -> NoneC
```

- ▶ Next: we need to specify some laws for these ADT operations

Dictionary laws: mem?

- **Abstract values** look like this:

- $\{ \text{'Aparna'}: \text{'SF'}, \text{'Maria'}: \text{'Chicago'}, \text{'Aditi'}: \text{'Bengaluru'} \}$

- Laws for `mem? ()`:

- $\{d = \{ \dots, k_i: v_i, \dots \} \} \quad d.\text{mem?}(k_i) \Rightarrow \text{True} \quad \{d = \{ \dots, k_i: v_i, \dots \} \}$

- $\{d = \{ \dots, k_i: v_i, \dots \} \wedge \forall i, k_i \neq k \}$

$d.\text{mem?}(k) \Rightarrow \text{False}$

$\{d = \{ \dots, k_i: v_i, \dots \} \wedge \forall i, k_i \neq k \}$

Dictionary laws: get

- ▶ **Abstract values** look like this:

- ▶ $\{ \text{'Aparna'}: \text{'SF'}, \text{'Maria'}: \text{'Chicago'}, \text{'Aditi'}: \text{'Bengaluru'} \}$

- ▶ Laws for `get ()`:

- ▶ $\{d = \{ \dots, k_i: v_i, \dots \} \} \quad d.\text{get}(k_i) \Rightarrow v_i \quad \{d = \{ \dots, k_i: v_i, \dots \} \}$

Dictionary laws: put

- ▶ **Abstract values** look like this:

- ▶ $\{ \text{'Aparna'}: \text{'SF'}, \text{'Maria'}: \text{'Chicago'}, \text{'Aditi'}: \text{'Bengaluru'} \}$

- ▶ Laws for `put ()`:

- ▶ $\{d = \{ \dots, k_i: v_i, \dots \} \wedge \forall i, k_i \neq k\}$

$$d.\text{put}(k, v) \Rightarrow \text{None}$$
$$\{d = \{ \dots, k_i: v_i, \dots, k: v \} \}$$

Dictionary laws: del

- ▶ **Abstract values** look like this:

- ▶ `{'Aparna': 'SF', 'Maria': 'Chicago', 'Aditi': 'Bengaluru'}`

- ▶ Laws for `del()`:

- ▶ $\{d = \{ \dots, k_j:v_j, k_i:v_i, k_m:v_m, \dots \} \}$

$$d.del(k_i) \Rightarrow None$$

$$\{d = \{ \dots, k_j:v_j, k_m:v_m, \dots \} \}$$

What next?

ADT vs. Concrete data structure

- ▶ We know how we want the dictionaries to behave
 - ▶ Abstract Data Type (ADT)
 - ▶ E.g., How we use `dict()` in Python
- ▶ We don't yet know how to build one yet
 - ▶ Concrete data structure
 - ▶ E.g., how `dict()` may be actually implemented under the hood
- ▶ Interface vs. Implementation

Variants to a dictionary ADT

- ▶ Many possible behaviors of dictionary ADTs
 - ▶ All with the same basic idea: a collection of key-value pairs where you can insert or delete
 - ▶ Functions have many possible names for the same operations
- ▶ Some variants (can be many more):
 - ▶ ADTs without a delete operation
 - ▶ Duplicate keys are allowed
 - ▶ Deleting or getting an absent key doesn't trigger an error; fails silently or returns a "default" value
 - ▶ "Putting" an entry with an existing key isn't allowed

Bidirectional mappings

- ▶ Very common in programming and life; dictionaries can help!
- ▶ What if we want to look things up in two directions?
 - ▶ Convert letters (A-Z) to their respective numbers (1-26) and vice versa
 - ▶ Get the cities of my friends but also see which friend lives in which city (if I want to visit)
 - ▶ In a competition, find out the rank of a player but also find the player at a rank
- ▶ How can we do this?
 - ▶ Maintain two dictionaries! One for each direction, for example,
 - ▶ dict1 named "friend_to_cities" (key = friend, value = city)
 - ▶ dict2 named "cities_to_friends" (key = city, value = friend)

Pause

- ▶ Any questions?
- ▶ Anything unclear or that I missed?

Implementing dictionaries

How can we implement a dictionary?

- ▶ What data structures have we we seen so far?
 - ▶ Arrays
 - ▶ Linked lists


Array as a simple dictionary

- ▶ An array already implements a very basic dictionary
 - ▶ Key: array index (type: always integer ≥ 0)
 - ▶ Value: element value at index (type: whatever type the array holds)

	0	1	2	3	4	5
arr	'sr'	'ut'	'i'	'oi'	'n'	'k'

- ▶ Called "Direct Addressing" when the index is the key

Array as a simple dictionary

- ▶ Operations:
 - ▶ Lookup key and get a value
 - ▶ `arr[4]` 

	0	1	2	3	4	5
arr	'sr'	'ut'	'i'	'oi'	'n'	'k'

Array as a simple dictionary

- ▶ Operations:
 - ▶ Lookup key and get a value
 - ▶ `arr[4]`
 - ▶ Update value against a key:
 - ▶ `arr[4] = 'oink'`

	0	1	2	3	4	5
arr	'sr'	'ut'	'i'	'oi'	'oink'	'k'

Array as a simple dictionary

- ▶ Operations:

- ▶ Lookup key and get a value

- ▶ `arr[4]`

- ▶ Update value against a key:

- ▶ `arr[4] = 'oink'`

	0	1	2	3	4	5
arr	'sr'	'ut'	'i'	'oi'	'oink'	'k'

- ▶ How might we implement put, and del?

- ▶ All array indices (and therefore, keys) already exist

- ▶ Maybe we don't allow these operations with this implementation (a variant)

Limitations of direct addressing

- ▶ Approach only suitable for specific use-cases
- ▶ Limited by a fixed number of key values
- ▶ Dictionaries often need more complex keys
- ▶ Arrays could still be used but in a different way!

What about the other concrete DSes we know about?

- ▶ How can we represent collections of key-value pairs using arrays or linked lists?

What other implementations could we use?

- ▶ An array of (key, value) pairs → structs or tuples
 - ▶ Unsorted, or sorted according to key
- ▶ Linked list of (key, value) pairs
- ▶ Hash table (next time)
- ▶ Other data structures we'll see later in the quarter!
- ▶ We'll also call key-value pairs **entries** (singular: **entry**)


Dictionaries so far

	Unsorted array of (key, value) pairs	Sorted array of (key, value) pairs; sorted by key	Linked list of (key, value) pairs
Lookup	$O(n)$		
Insert	$O(n)$ since insertion may sometimes require making new array		
Delete	$O(n)$		
Member?	$O(n)$		

If we use efficient dynamic arrays; we can get $O(1)$ amortized → later in the question

What do we need for an implementation?

1. A concrete data representation of the dictionary using a specific data structure
2. Function definitions for interface functions while satisfying laws
3. A representation for each key-value pair in the dictionary

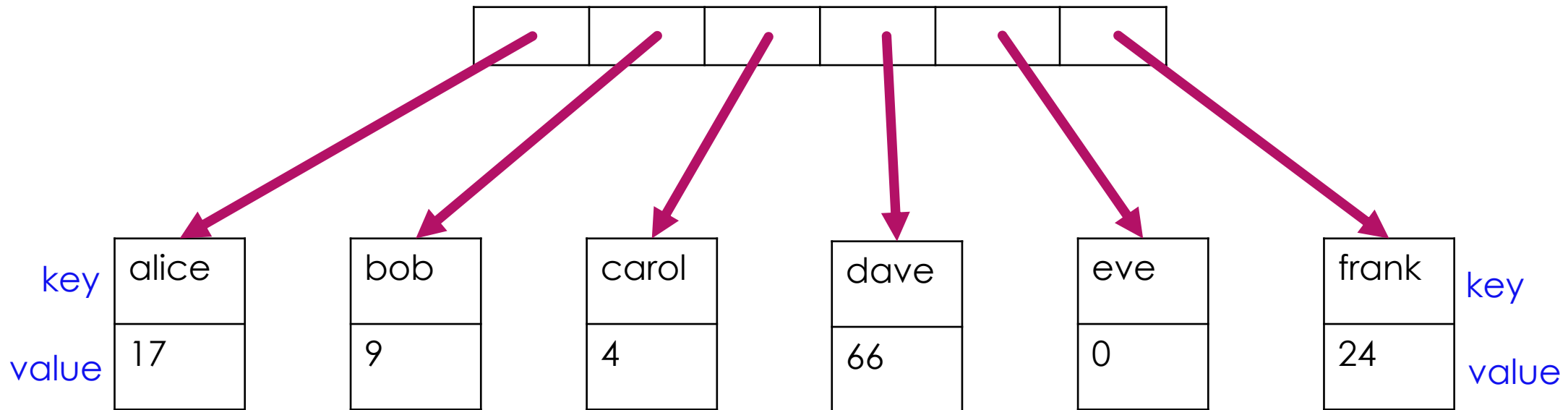


We'll discuss these two conceptually

Implementation #1

SORTED ARRAY OF
KEY-VALUE PAIRS

Sorted array dictionary



- We look things up by key, therefore, we'll keep the entries in the array sorted by key

Lookup operation

- ▶ Keys are sorted; use binary search!
 - ▶ **Possible because we have $O(1)$ access to any element at a given index and so we can find new midpoints in $O(1)$!**
- ▶ Need to look up key 75?
 - ▶ Lookup key in at most $\log_2 n$ steps
 - ▶ Return value for key
- ▶ Lookup complexity: $O(\log n)$

2	17	19	56	75	77	90
---	----	----	----	----	----	----

Here, an array with just keys visible

You can assume the values are there too, just not shown

Member? operation

- ▶ Same approach as lookup
- ▶ Lookup a key and just return true/false instead of value
- ▶ Member? complexity: $O(\log n)$

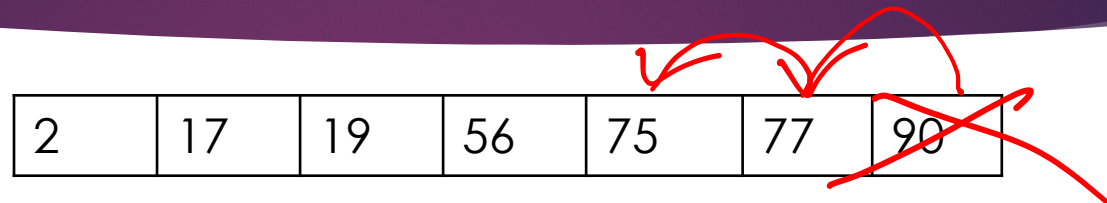
2	17	19	56	75	77	90
---	----	----	----	----	----	----

Insert operation

2	17	19	56	75	77	90
---	----	----	----	----	---------------	----

- ▶ Insert a new entry with key = 76?
 - ▶ If the array is full: Make a new array and copy over elements, shift elements after desired spot to the right
 - ▶ If the array has space: Shift elements after desired spot to make space for new one
- ▶ There may be as many as n elements to copy or shift over
- ▶ Insert complexity: $O(n)$

Delete operation



The diagram shows an array with seven elements: 2, 17, 19, 56, 75, 77, and 90. Red arrows indicate the deletion of the element 75. One arrow points from 75 to the space before it, and another points from 77 to the space before it. A red 'X' is drawn over the last two elements, 77 and 90, indicating they are shifted one position to the left.

2	17	19	56	75	77	90
---	----	----	----	----	----	----

- ▶ Delete an entry with key = 75?
 - ▶ Find the index of entry with key 75
 - ▶ Shift over elements after this location to the left, overwriting the entry to delete
- ▶ Lookup will be $O(\log n)$ but there may be as many as n elements to copy or shift over
- ▶ Delete complexity: $O(n)$

Dictionaries so far

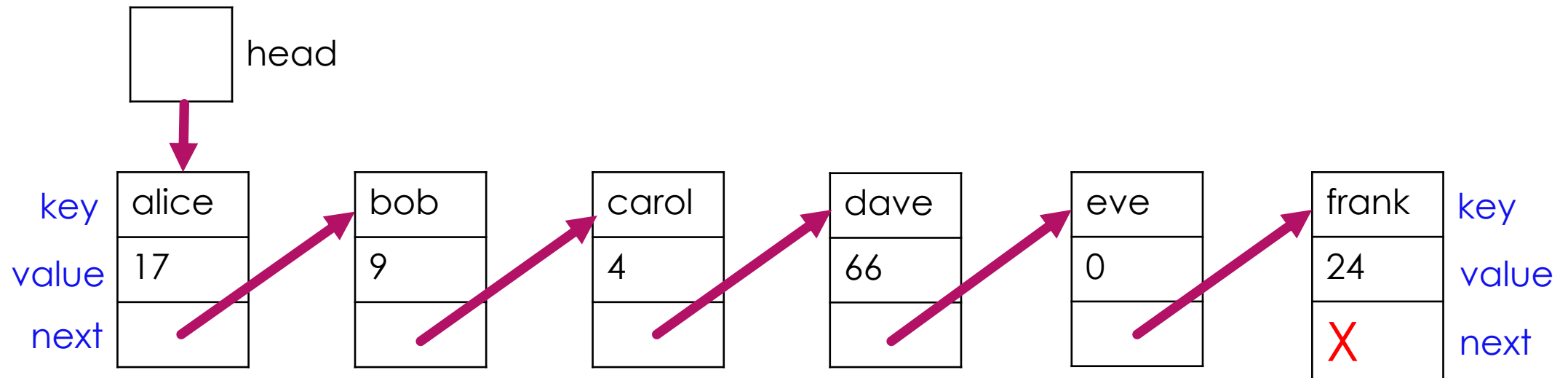
	Unsorted array of (key, value) pairs	Sorted array of (key, value) pairs; sorted by key	Linked list of (key, value) pairs
Lookup	$O(n)$	$O(\log n)$	
Insert	$O(n)$	$O(n)$	
Delete	$O(n)$	$O(n)$	
Member?	$O(n)$	$O(\log n)$	

Implementation #2

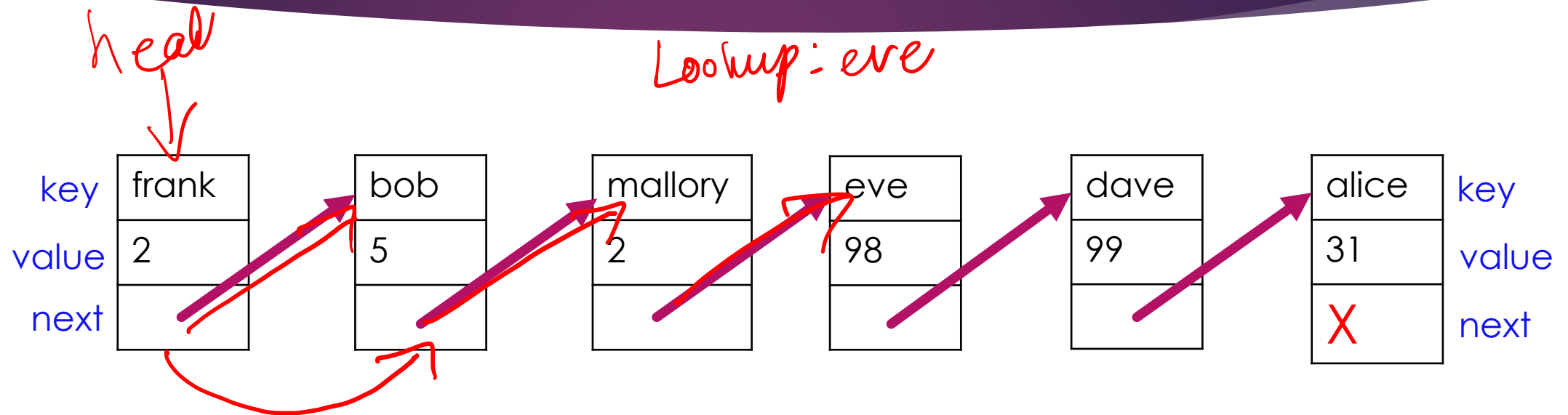
LINKED LIST OF
KEY-VALUE PAIRS

Linked list as an association list

- ▶ An association list is a collection of (key, value) pairs or entries
- ▶ We can store an association list using a linked list data structure where each node has 3 fields now

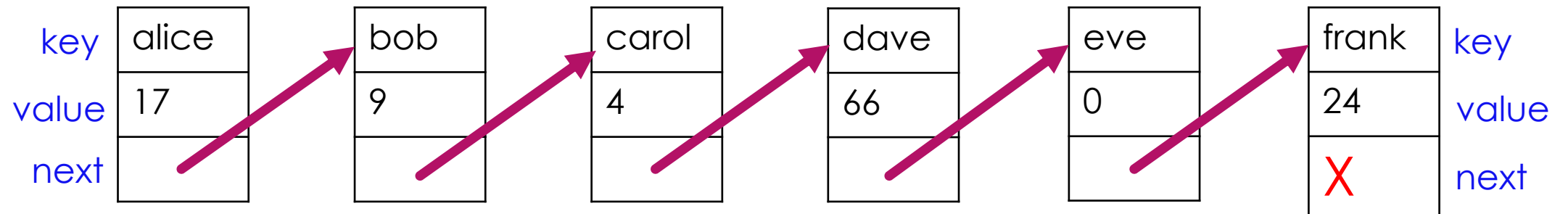


Lookup operation



- Need to do linear search
 - Aside: What if the association list was sorted by key?
- Lookup complexity: $O(n)$

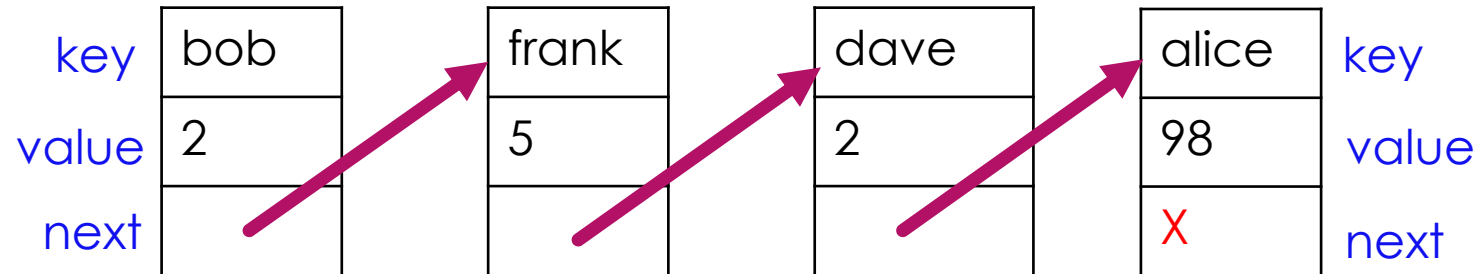
In-class exercise (5 minutes)



Assume that the association list is sorted (e.g., above) and you only have access to the head of the list. Binary search would still not be possible and the complexity cannot be reduced below $O(n)$.

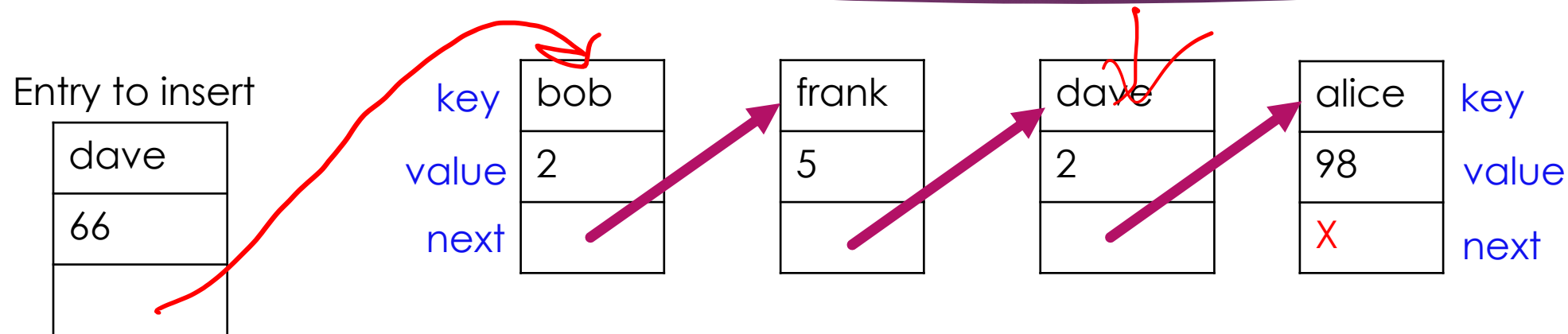
1. Justify why this is the case in a few sentences (be specific).

Insert operation



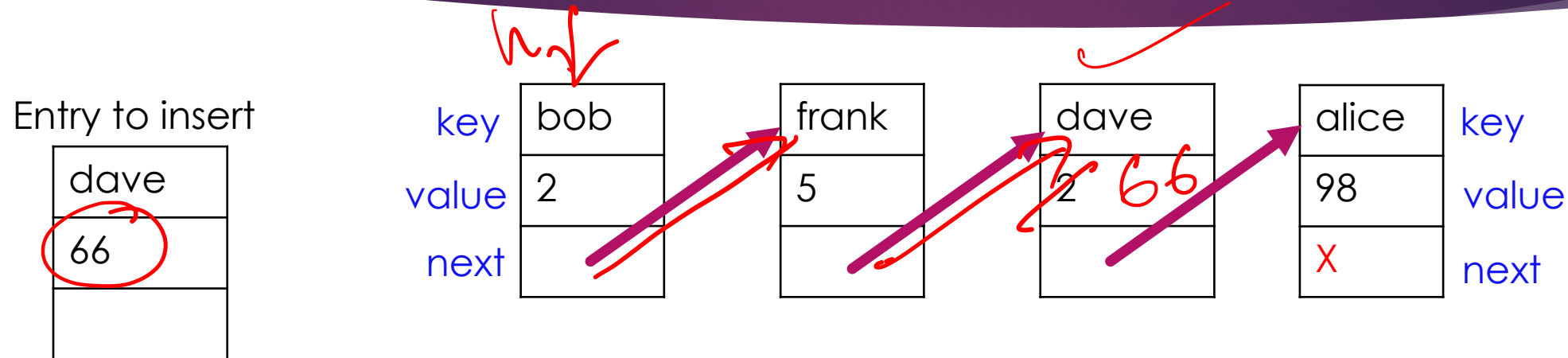
- Where do we insert to get an efficient operation?

Insert operation



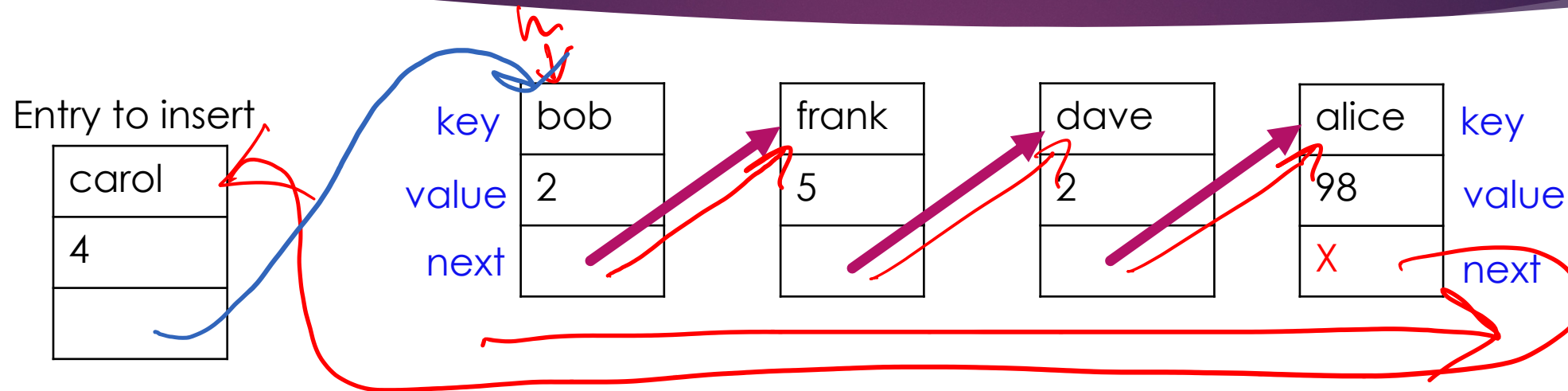
- ▶ Inserting at the front would give us $O(1)$ complexity
- ▶ Any issues with that though?
 - ▶ Our constraints don't allow duplicate keys which "dave" is
 - ▶ Need to go through the entire list to see if "dave" already exists, **then** insert

Insert operation



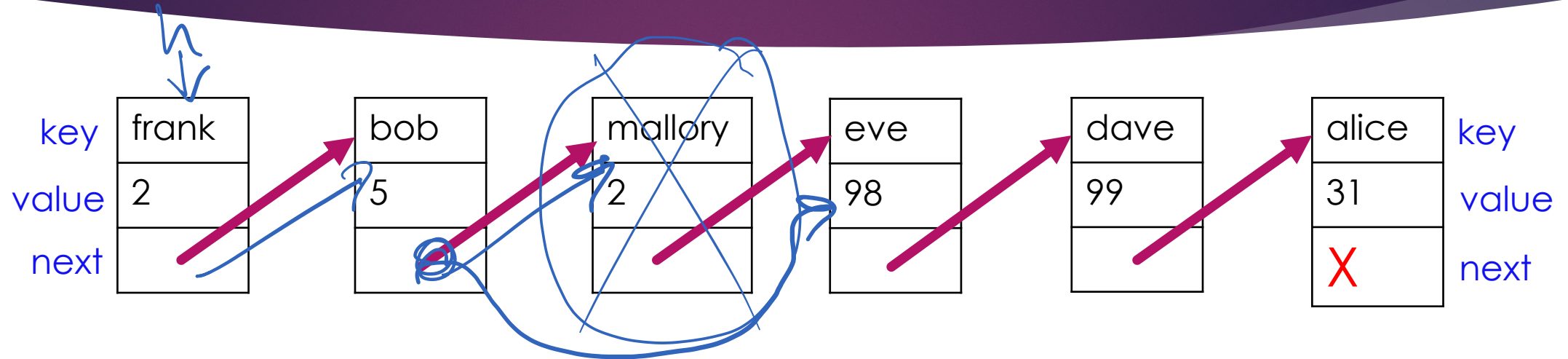
- ▶ Let's go through the whole list first to see if "dave" is there
 - ▶ "dave" is there so we can't insert this new node
 - ▶ We could error out or update "dave"'s value with the new value
- ▶ Lookup complexity: $O(n)$

Insert operation



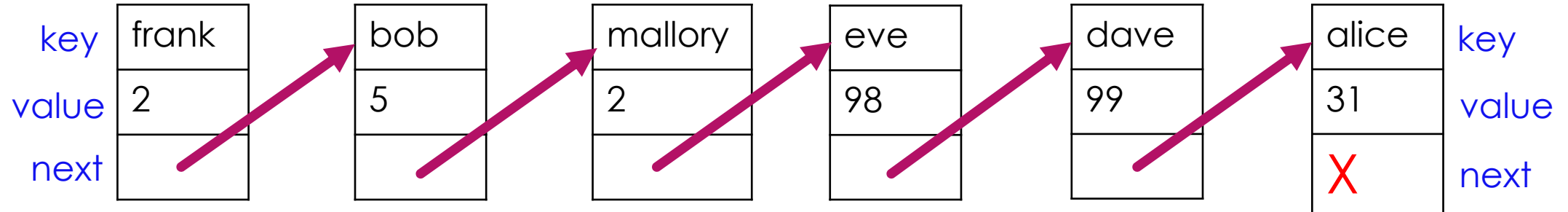
- ▶ Let's try inserting "carol"
 - ▶ Look through list and didn't find "carol" so we can add "carol"'s entry
 - ▶ Where? Either front or back is fine!
- ▶ Lookup complexity: $O(n)$

Delete operation



- ▶ Traverse the list to find the entry to remove and then change the arrows surrounding this entry
- ▶ Delete complexity: $O(n)$

Member? operation



- ▶ Same approach as lookup
- ▶ Lookup a key and just return true/false instead of value
- ▶ Member? complexity: $O(n)$

Dictionaries so far

	Unsorted array of (key, value) pairs	Sorted array of (key, value) pairs; sorted by key	Linked list of (key, value) pairs
Lookup	$O(n)$	$O(\log n)$	$O(n)$
Insert	$O(n)$	$O(n)$	$O(n)$
Delete	$O(n)$	$O(n)$	$O(n)$
Member?	$O(n)$	$O(\log n)$	$O(n)$

Dictionaries so far

Can we do better across the board?

Maybe! (Next time and future)

Insert	$O(n)$	$O(n)$	$O(n)$
Delete	$O(n)$	$O(n)$	$O(n)$
Member?	$O(n)$	$O(\log n)$	$O(n)$

On your own: Comparing dictionaries

- ▶ Multiple data structures following the same ADT interface means we can swap implementations in and out!
 - ▶ Like how you tested your playlist with a `ListQueue` and a `RingBuffer`
- ▶ Main different is efficiency → We can see this empirically!
- ▶ See `benchmarks.rkt` on Canvas and run the benchmark tests at the bottom for the dictionaries you want to analyze