Code Fellows

DSA 05 + Class 26 HashTables and Application State

seattle-javascript-401n14

Lab 25 Review

Career Coaching Workshop #1

Vocab Review!

What is a dispatch?

What is a reducer?

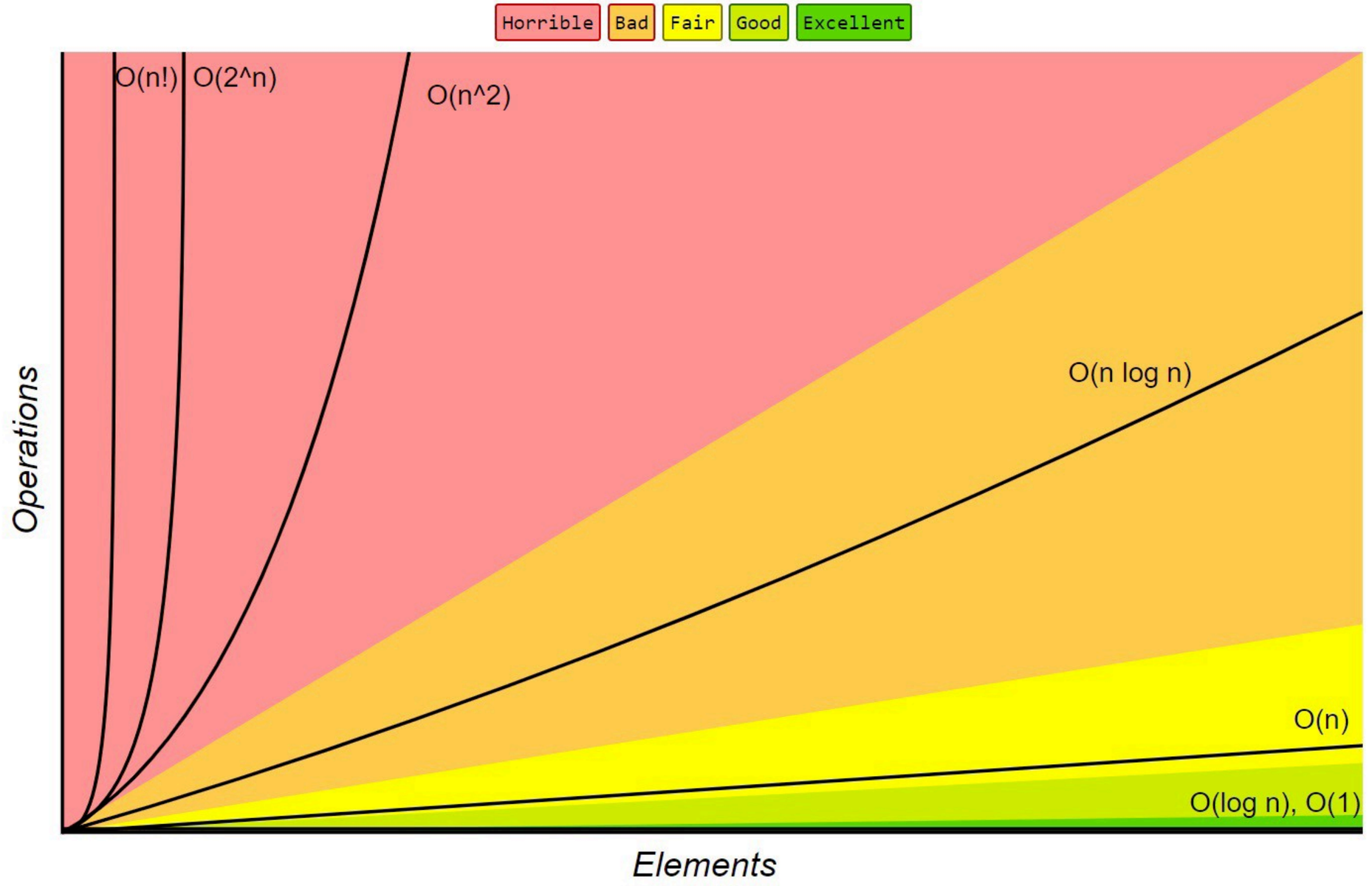
What is context?

What is a hash function?

What is a map?

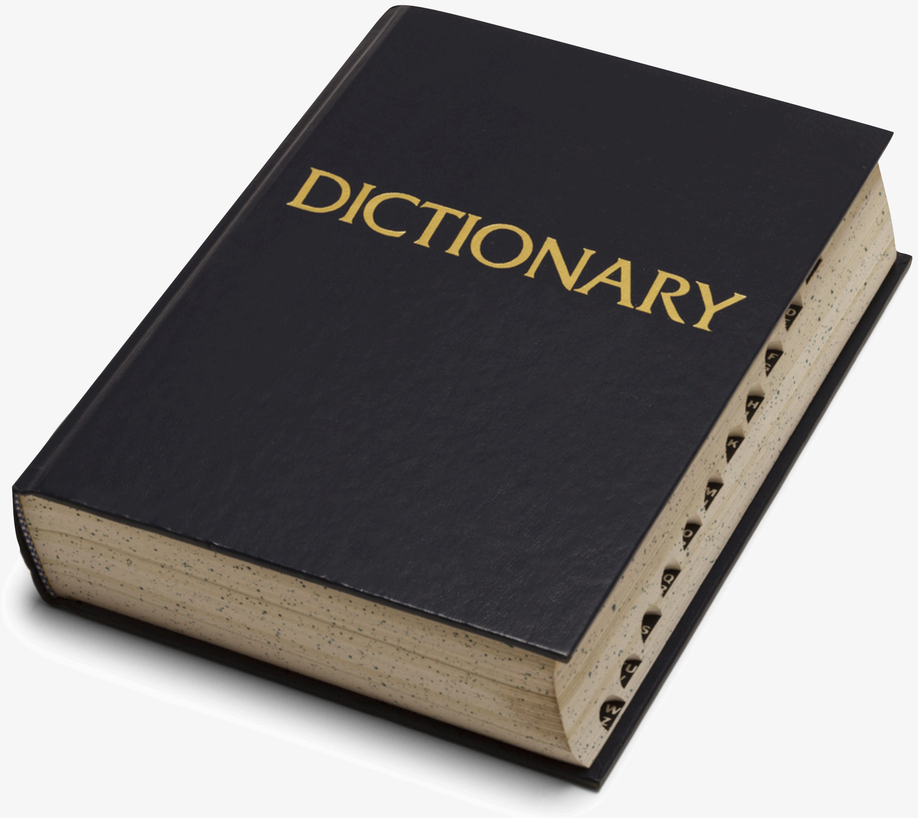
What is what is Big-O?

Big-O Complexity Chart



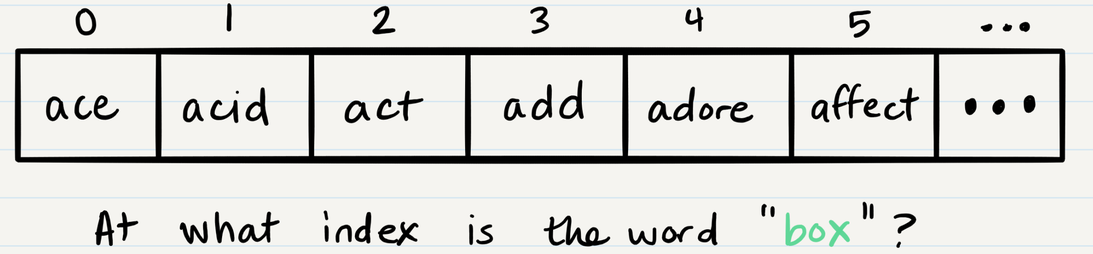
Storing a Dictionary

- I have a dictionary with definitions for each word
- I want to be able to *quickly* look up any word's definition
- What data structure should I use?



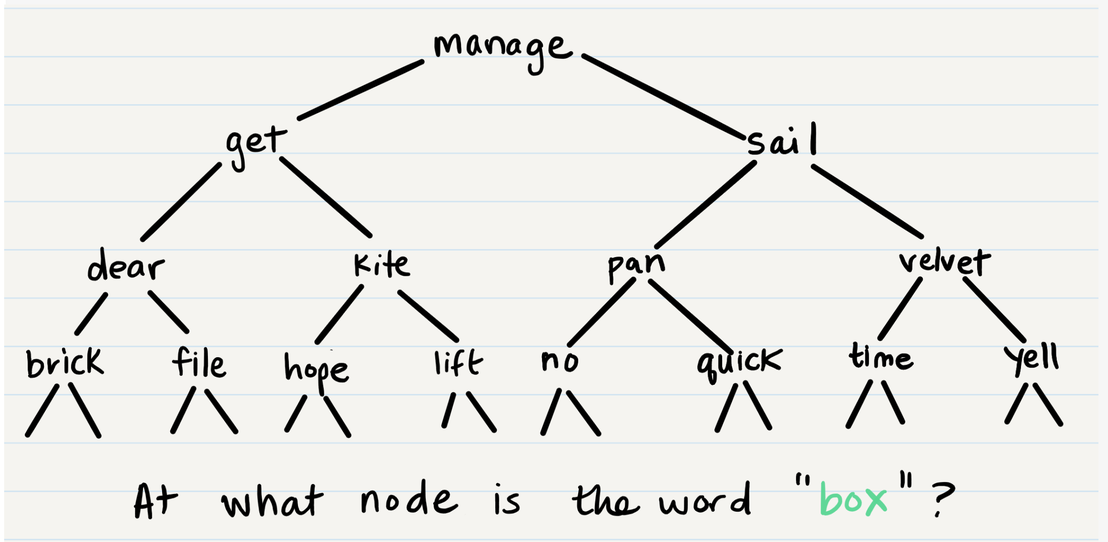
Array or Linked List

- Each time I see a new word in the dictionary, I push it to Array
- Sorting is possible (alphabetically)
- Searching is possible
  - But I still need to *search*
  - Can’t directly jump to the word I want
  - Worst case search is at least `O(logn)`



Stack, Queue, Tree

- Probably either the same or worse than an Array
- We can get a binary search tree at `O(logn)`
- Stacks and Queues are not really an option
    - You have to remove items to get to what you want
    - Otherwise, `O(n)`



Object

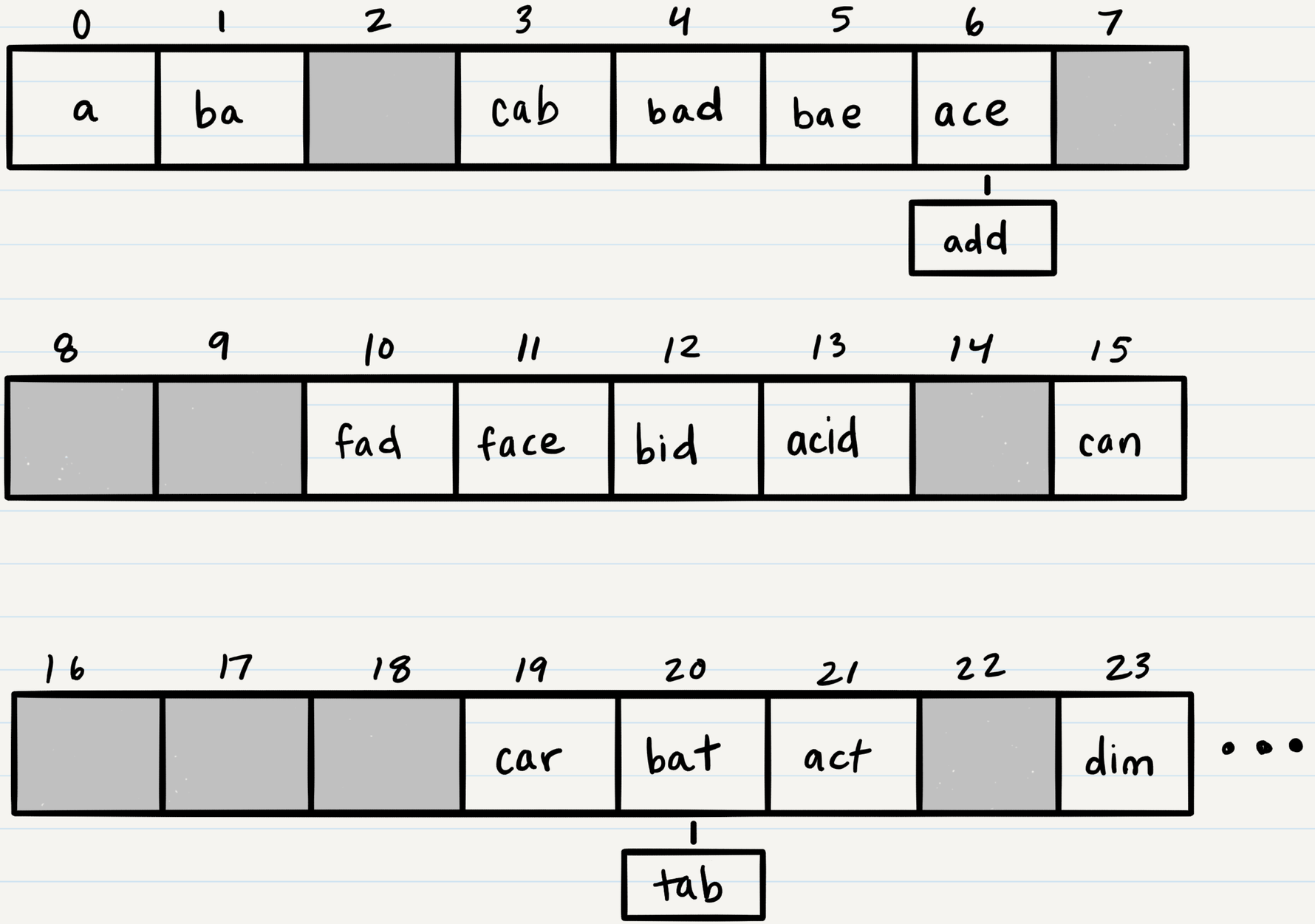
- Seems like a better solution
- We can have each word and definition stored as a key-value
- If we have a word with multiple definitions, store them as an array
- What if our key was something more complicated / an invalid key name?

```
const words = {
    ace: ['The best at', 'A playing card'],
    acid: 'A dangerous chemical',
    act: 'Pretend to be someone else',
    add: 'Combine two numbers into a sum',
    adore: 'Love something',
    ...
};

const box = words['box'];
```

Something Smarter?

- What if I had a function that converted my words to the index of an array?
  - Set a number value for each letter: `a = 0, b = 1, c = 2,` …
  - For every word, sum up the number values:
    - `'ace' >> 0 + 2 + 4 = 6`
    - `'act' >> 0 + 2 + 19 = 21`
    - `'box' >> 1 + 14 + 23 = 38`
  - If there is a *collision*, at that index in the array create a linked list
    - `'bat'` and `'tab'` are both equal to the index `20`

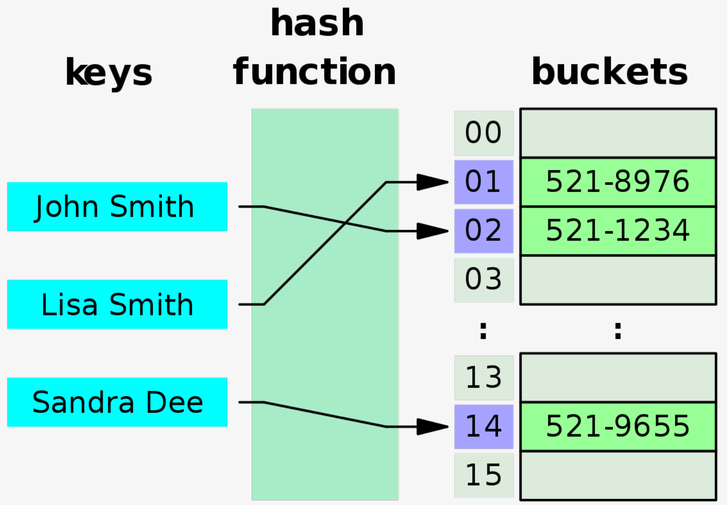

0
1
2
3
4
5
6
7
a
ba
cab
bad
bae
ace
add
8
9
10
11
12
13
14
15
fad
face
bid
acid
can
16
17
18
19
20
21
22
23
car
bat
act
dim
tab

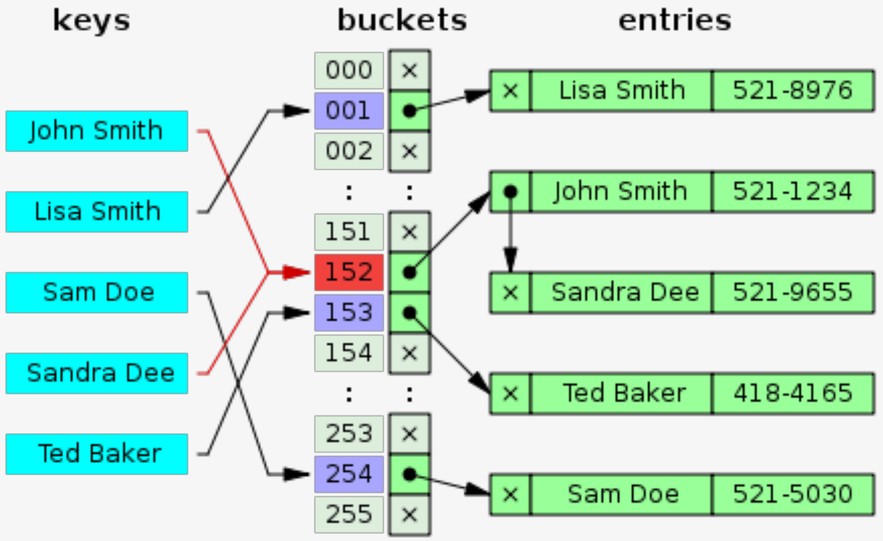
Hash Tables

- **Hash Tables** are large array-like structures
  - Essentially arrays, but indexes are called "**buckets**", and each bucket is initialized to null
  - Why is this distinction important?
    - In some languages arrays take up more space than buckets
- Hash Tables have `insert` and `search` time complexity of `O(1)`!
  - Perfect **hash functions** never create a **collision**
  - If you have collisions, technically, `O(max collision length)`
  - Bad hash table can have `O(n)`, but most have `O(2)` or `O(5)`, which is about `O(1)`

Hash Tables

- Very efficient way to store and search lots of data!
- Hash functions can be whatever we want
    - But the same key should always return the same bucket number
- Think of our `bcrypt` hash algorithm
    - Same password always results in the same hash
        - Salt is used to prevent collisions
- Our hash function should return a number that is less than the number of buckets in the table (mod operator)





Break!

State: The Good Stuff

- Lets us create variables that will re-render our component when changed
    - Great for user input variables
    - Great for variables that affect the display
    - Great for printing dynamic content

State: The Bad Stuff

- Very hard to share state between siblings / independent components
- We can pass from parent to child easily
- We can use context to create provider-consumer relationship
- But still these are structural limitations

Application State

- You know how contexts provide state variables to their consumers?
- What if the entire application provided a state to all of its components
- A single shared global state, saved in a “**store**” file
- Instead of a simple setter, we use reducers and dispatch

```
state.isFetching = true
```

```
state = updateFetching(
      state, true)
```

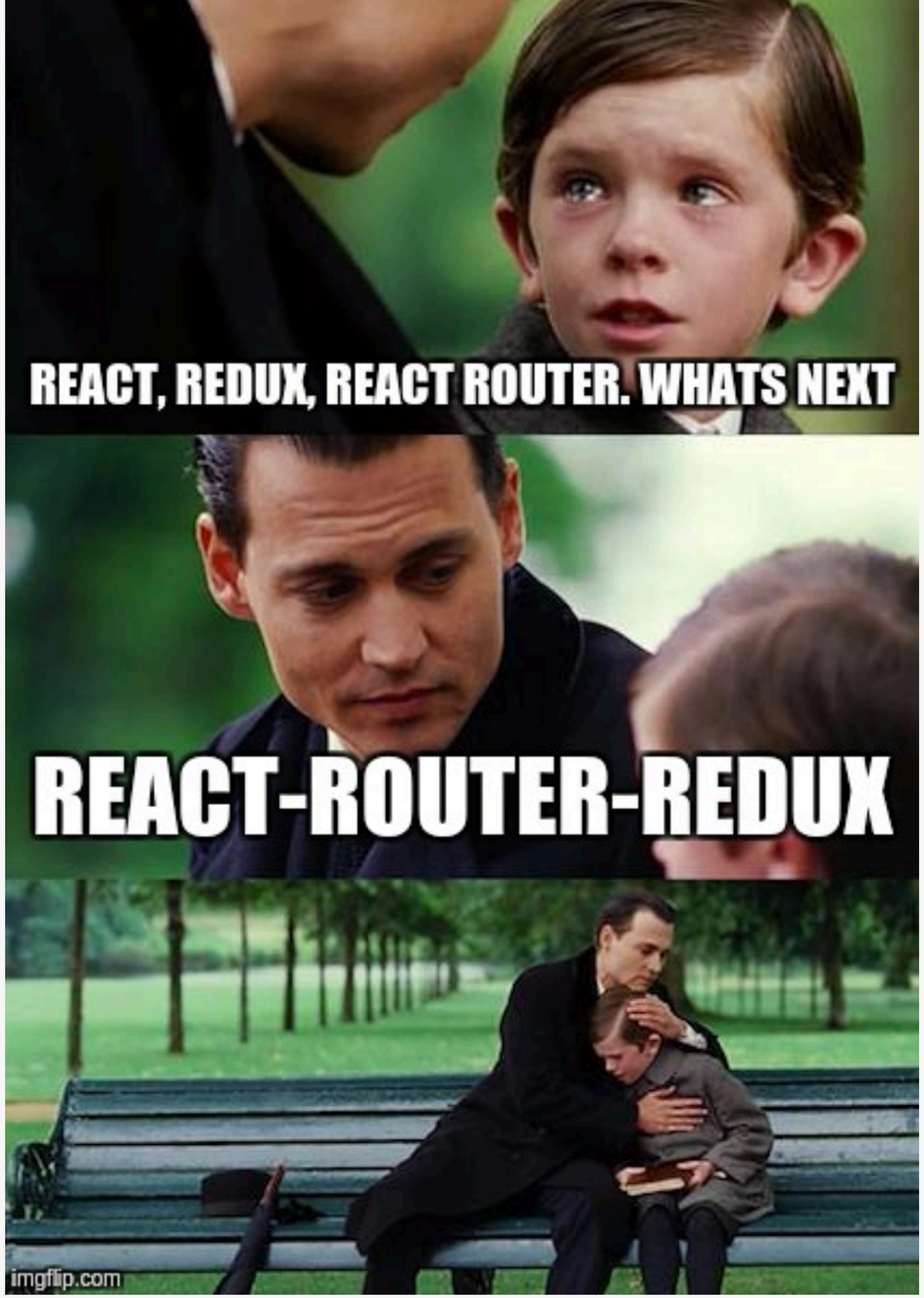
```
action = {
  type: 'UPDATE_FETCHING',
  value: true
}
state = update(state,action)
```

```
store = createStore(state)

store.dispatch(
  updateFetching(true)
);
store.getState();
```

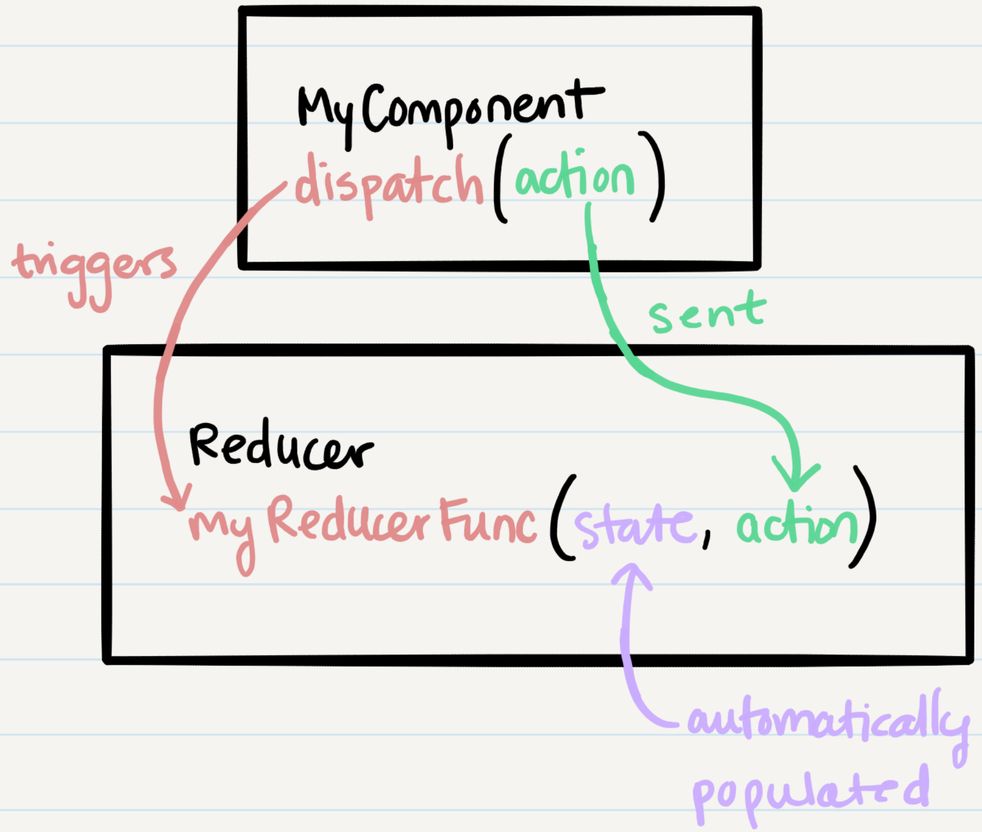
Redux

- A package addition to React
- Exposes a '**createStore**' function, which allows us to share a state (*getter*) + reducers (*setter*) to any component
- It's like `React.createContext()`!
    - **Contexts** are meant to provide to any descendants
    - **Stores** are meant to provide to the full application



Dispatch and Actions

- Redux creates a global state (*getter*) and a global reducer (*setter*)
- **Reducers** take in the current state and an **action** object, and return the new state
- A **dispatch** calls the reducer function, providing the action to the reducer



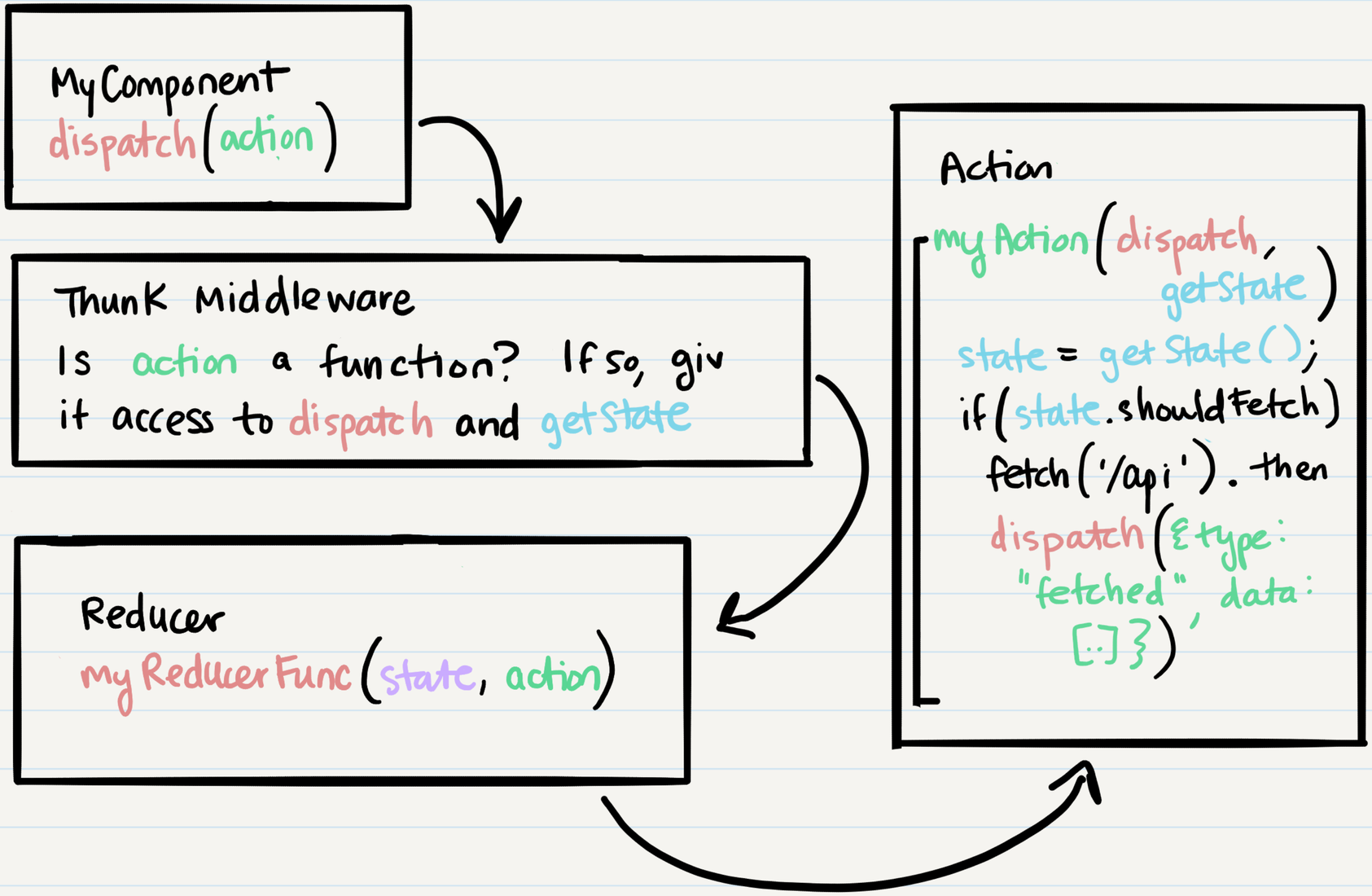
Demo

Let's set up our first application state, using Redux

Thunk

- A **middleware** for Redux
- This allows your reducer to do asynchronous changes to the state
    - Fetch from an API
- Why is this important? Our actions can now be functions!
- `redux-thunk` will automatically pass action functions the dispatch and getState functions as parameters
- “**Thunk**” means a function returned from another function
    - ```
      foo() {
          return fooThunk = () =>
            {console.log(“Thunk!”} }
      ```


      ```


MyComponent
dispatch(action)
Thunk Middleware
Is action a function? If so, giv
it access to dispatch and getState
Reducer
myReducerFunc(state, action)
Action
myAction(dispatch,
getState)
state = getState();
if(state.shouldFetch)
fetch('/api').then
dispatch({type:
"fetched", data:
[..]})

Demo

Let’s make complex action functions using Redux-Thunk

Lab 26 Preview

What's Next:

- Due by Midnight Tonight:
    - **Learning Journal 26**
- Due by Midnight Sunday:
    - **Instructor Sync 02**
    - **Career Accelerator: Program Overview**
    - **Feedback Week 17**
- Due by Midnight Monday:
    - **Code Challenge 26**
- Due by Tuesday 6:30pm:
    - **Lab 26**
    - **Reading Class 27**
- Next Class on Tuesday: **Class 27 - Dynamic Forms**

Questions?