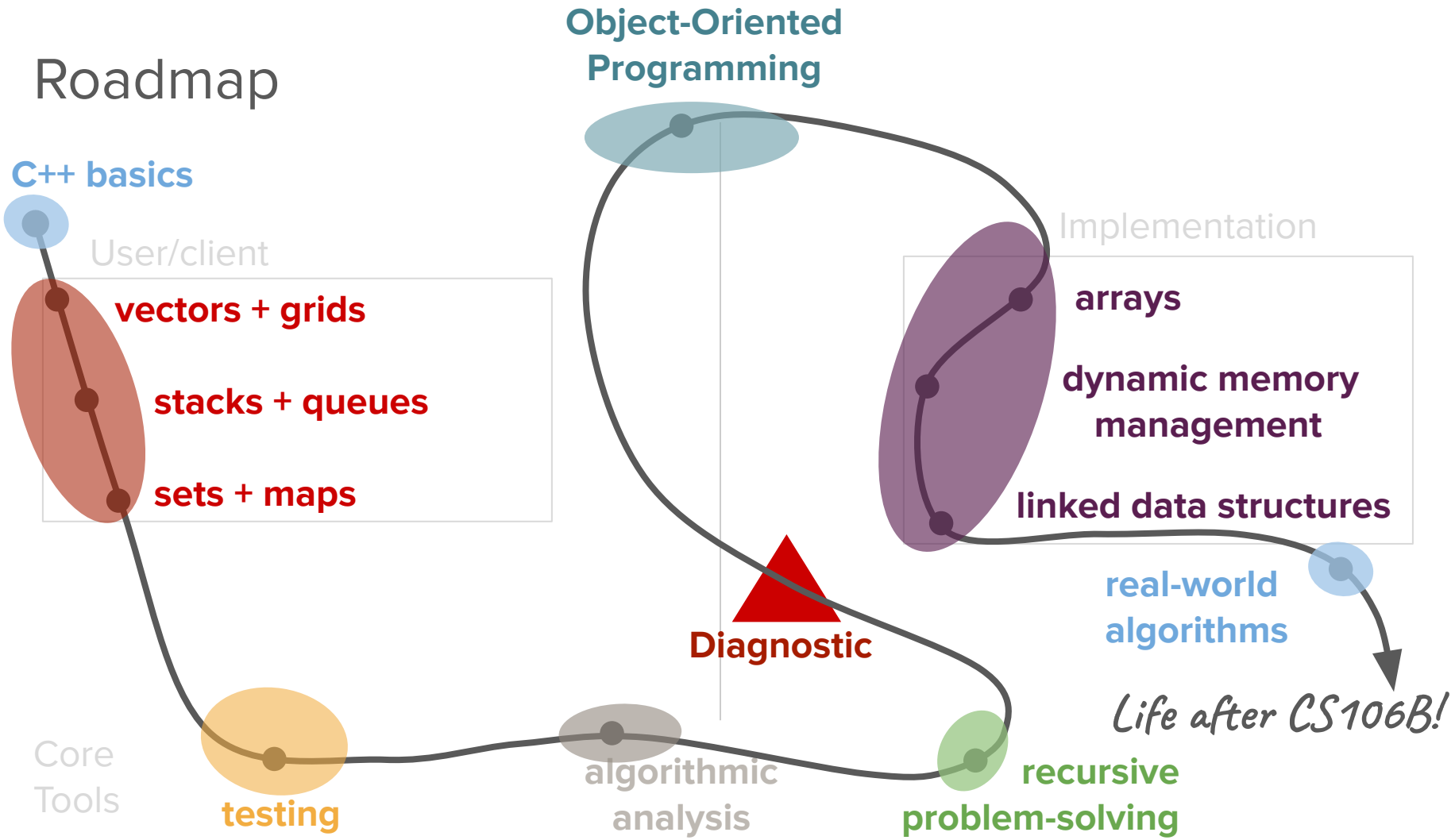
Hashing

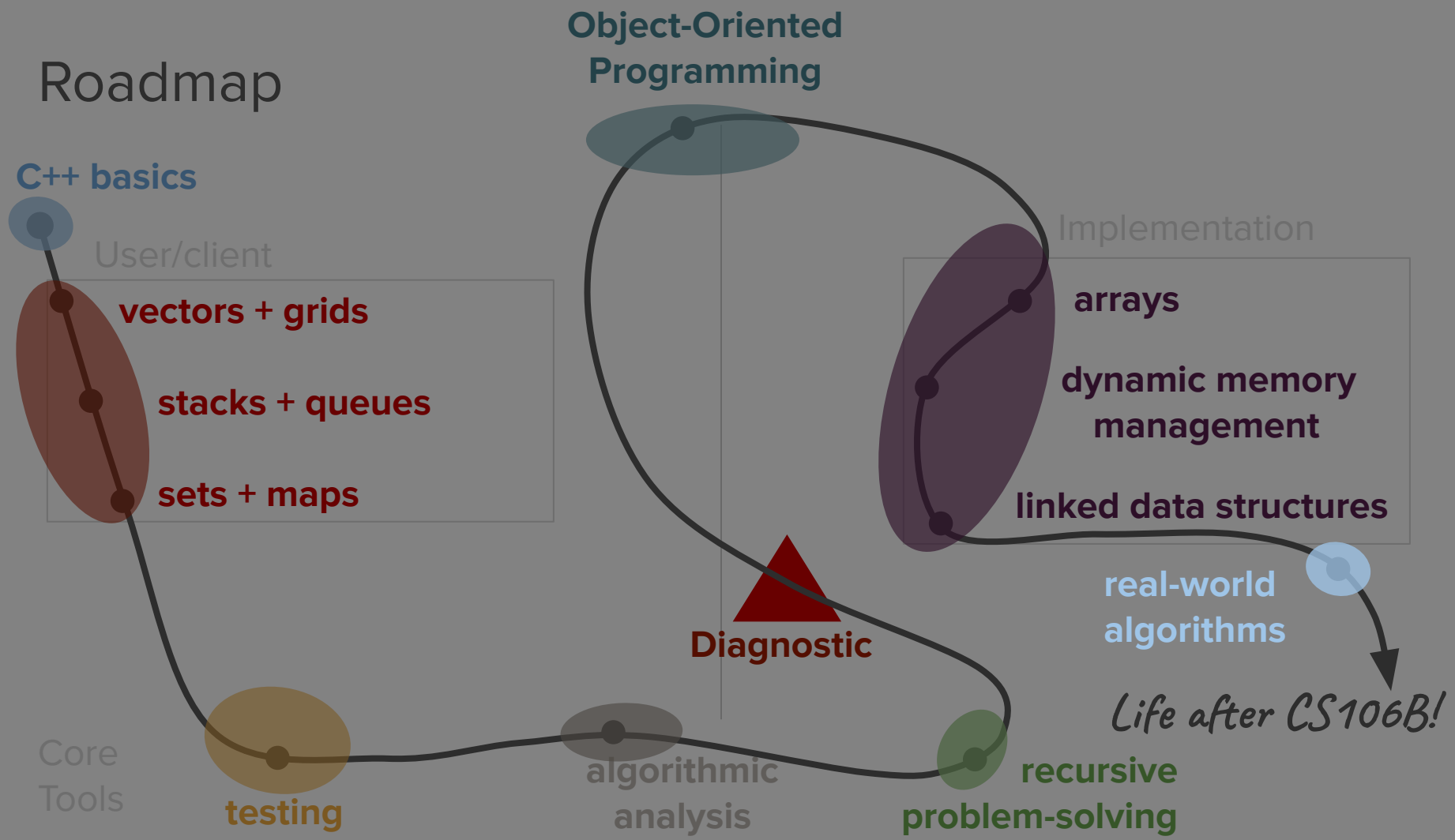
What's an example of compression being used that you've seen when using technology?

(put your answers 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
Life after CS106B!
Diagnostic
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
Life after CS106B!
Diagnostic
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today’s question

How does hashing apply to a variety of computational tasks and real-world problems?

Today's topics

1. What is a hash function?
2. Hashing in ADTs
3. Real-world applications of hashing (Hashzam!)

Review

[Huffman coding]

Why we use compression

- Storing data using the ASCII encoding is portable across systems, but is not ideal in terms of space usage.

- Building custom codes for specific strings might let us save space.

- **Idea:** Use this approach to build a **compression algorithm** to reduce the amount of space needed to store text.
  - In particular, we are interested in algorithms that provide **lossless compression**.
  - Compression algorithms **identify patterns in data** and take advantage of those patterns to come up with more efficient representations of that data!

Taking advantage of redundancy

- Not all letters have the same frequency in **`KIRK'S DIKDIK`**.
- The frequencies of each letter are shown to the right.
- So far, we've given each letter a code of the same length.
- **Key Question:** Can we give shorter encodings to more common characters?

character	*frequency*
K	4
I	3
D	2
R	1
'	1
S	1
␣	1

Prefix codes

- A **prefix code** is an encoding system in which no code is a prefix of another code.

- Here's a sample prefix code for the letters in **`KIRK'S DIKDIK`**.

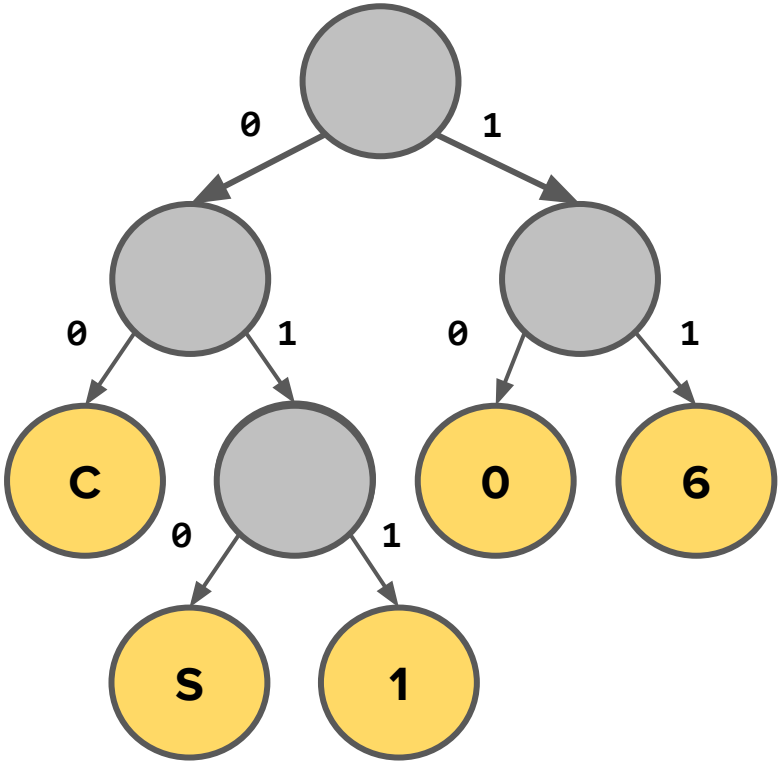
character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
␣	1100

Coding trees

- **Main Insight:** We can represent a prefix coding scheme with a binary tree! This special type of binary tree is called a **coding tree**.

- A coding tree is valid if all the letters are stored at the **leaves**, with internal nodes just doing the routing.

- **Goal:** Find the best coding tree for a string.



Huffman coding

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a **provably minimal encoding** for a given pattern of letter frequencies.

- Different data (different text, different images, etc.) will each have their own personalized Huffman coding tree.

- The Huffman coding algorithm is a flexible, powerful, adaptive algorithm for data compression. And you will implement it on assignment 7!

Huffman coding pseudocode

- To generate the optimal encoding tree for a given piece of text:
  - Build a **frequency table** that tallies the number of times each character appears in the text.
  - Initialize an empty **priority queue** that will hold partial trees (represented as `TreeNode*`)
  - Create **one leaf node per distinct character in the input string**. Add each new leaf node to the priority queue. The weight of that leaf is the frequency of the character.
  - While there are two or more trees in the priority queue:
    - Dequeue the two trees with the smallest weight from the priority queue.
    - **Combine them together to form a new tree** whose weight is the sum of the weights of the two trees.
    - Add that tree back to the priority queue.

Try it yourself!

Build a Huffman tree

1) Build the frequency table

Input Text: **MEMES!**

1) Build the frequency table

Input Text: **MEMES!**

char	frequency
M	2
E	2
S	1
!	1

2) Initialize the priority queue

higher priority ← → lower priority

{ }

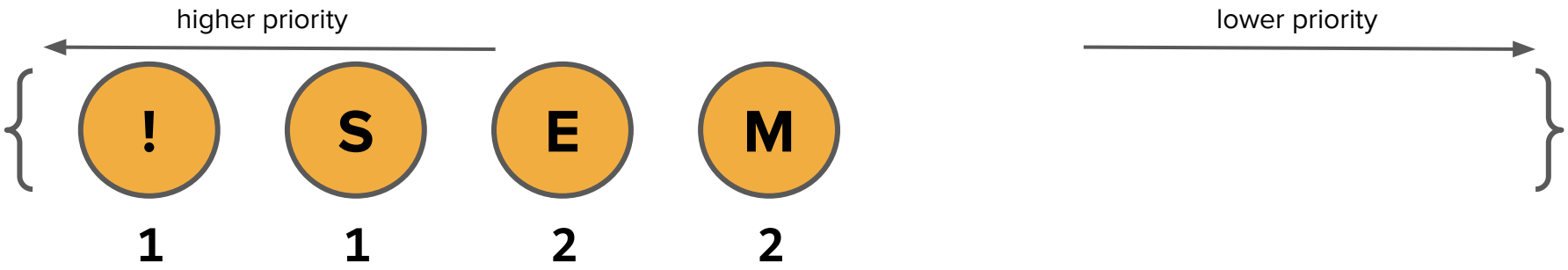
3) Add all unique characters as leaf nodes to queue

higher priority ← → lower priority

{ }

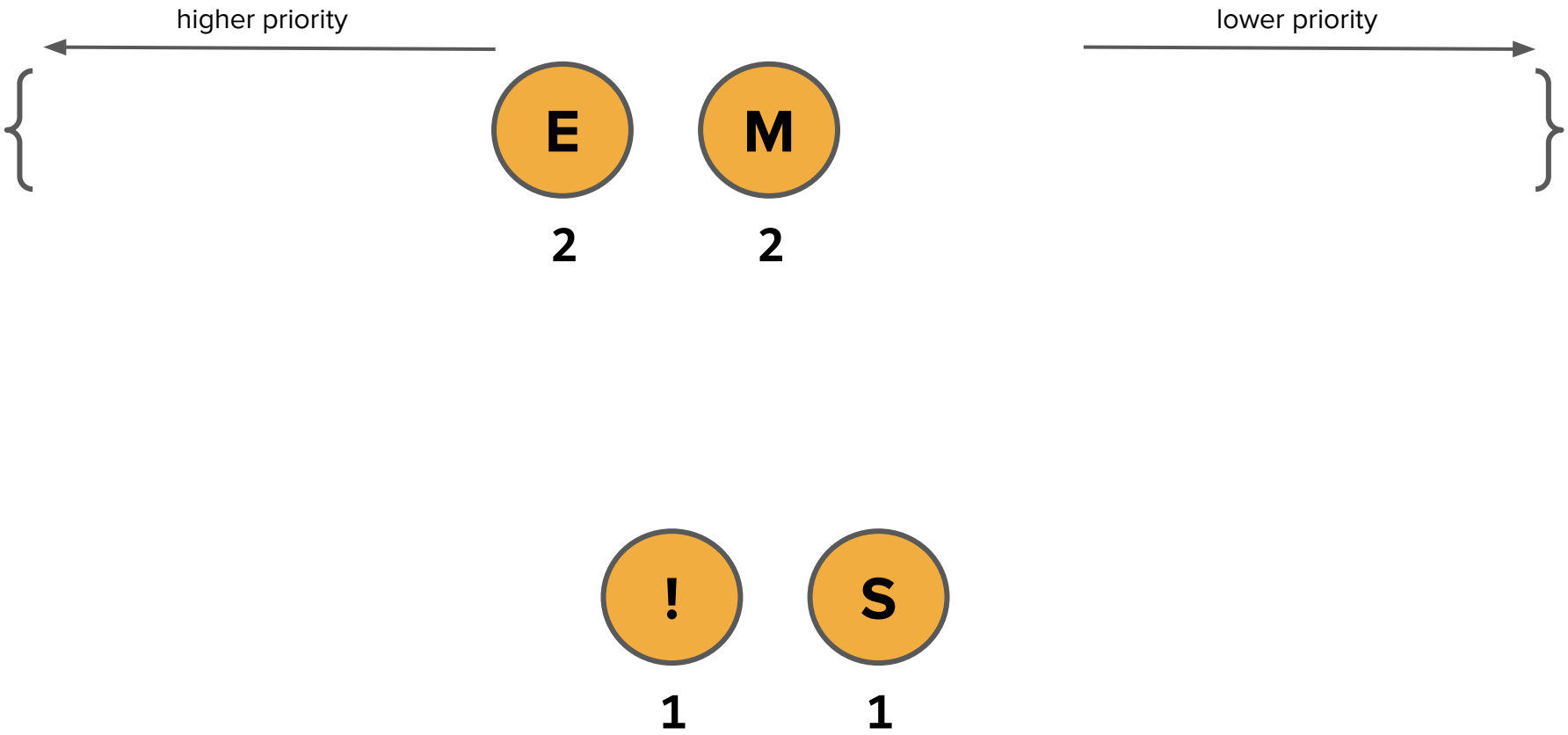
char	frequency
M	2
E	2
S	1
!	1

3) Add all unique characters as leaf nodes to queue

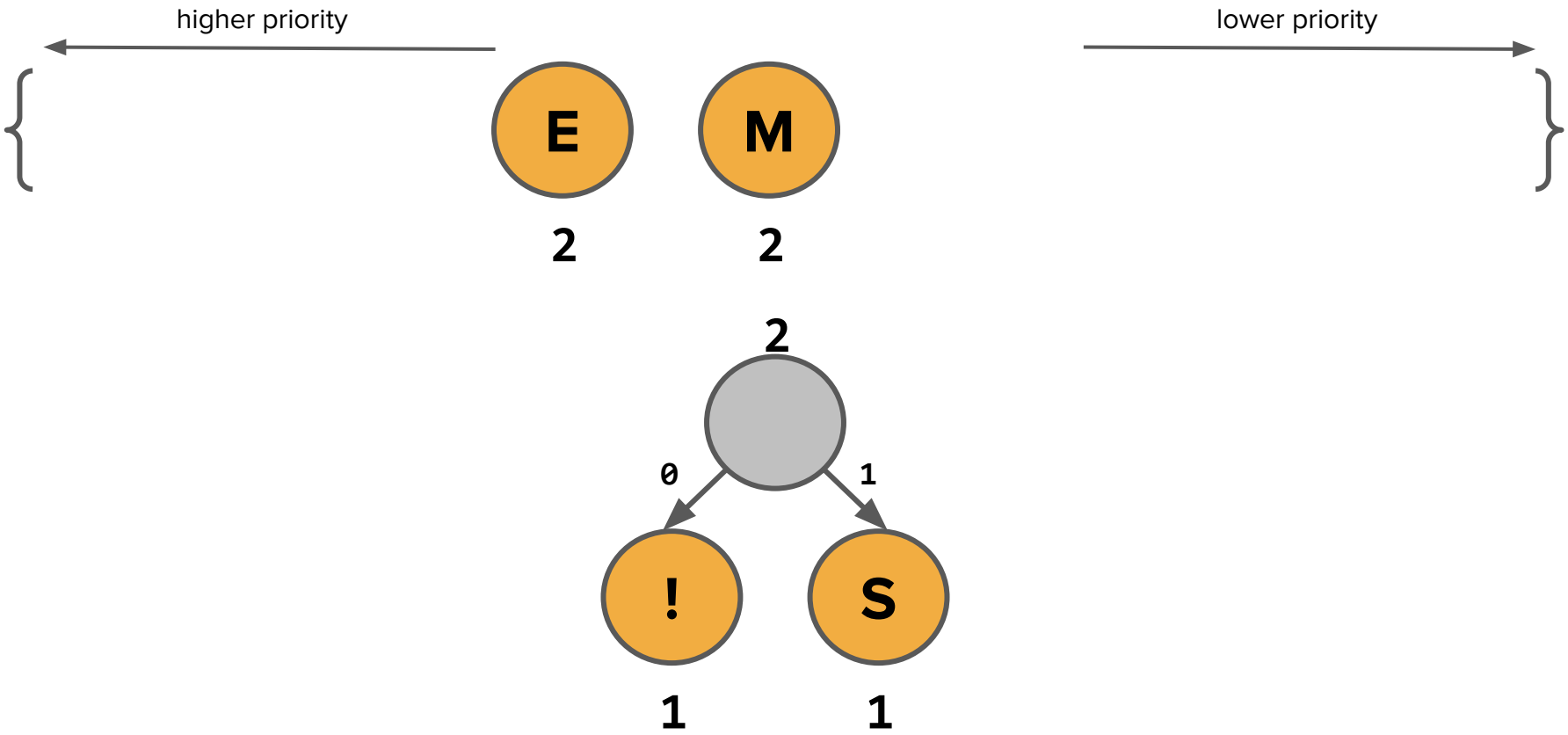


char	frequency
M	2
E	2
S	1
!	1

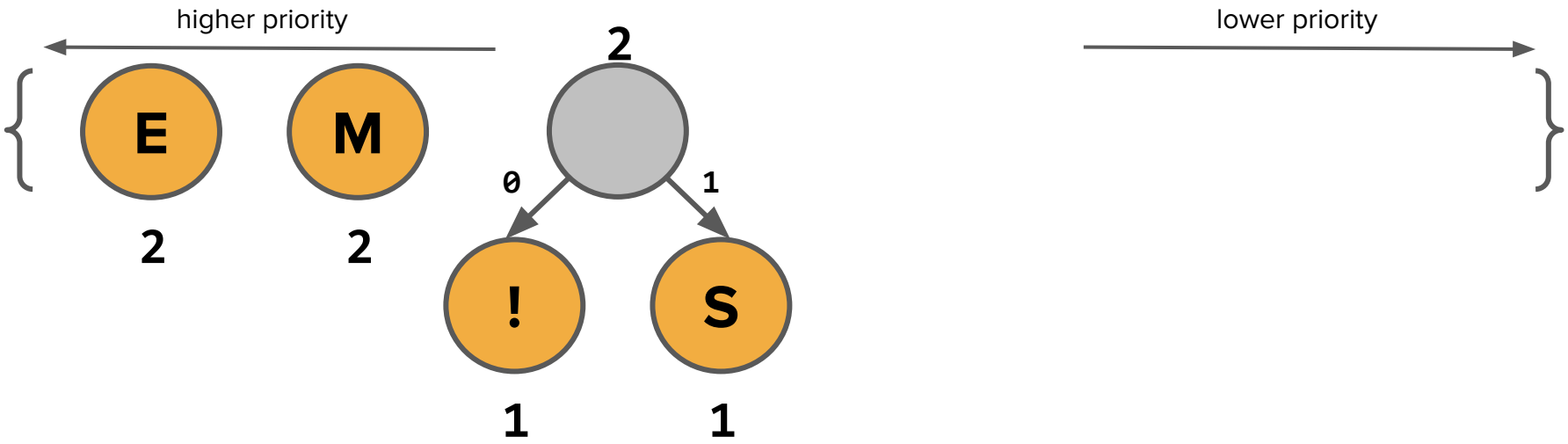
4) Build the Huffman tree by joining adjacent nodes



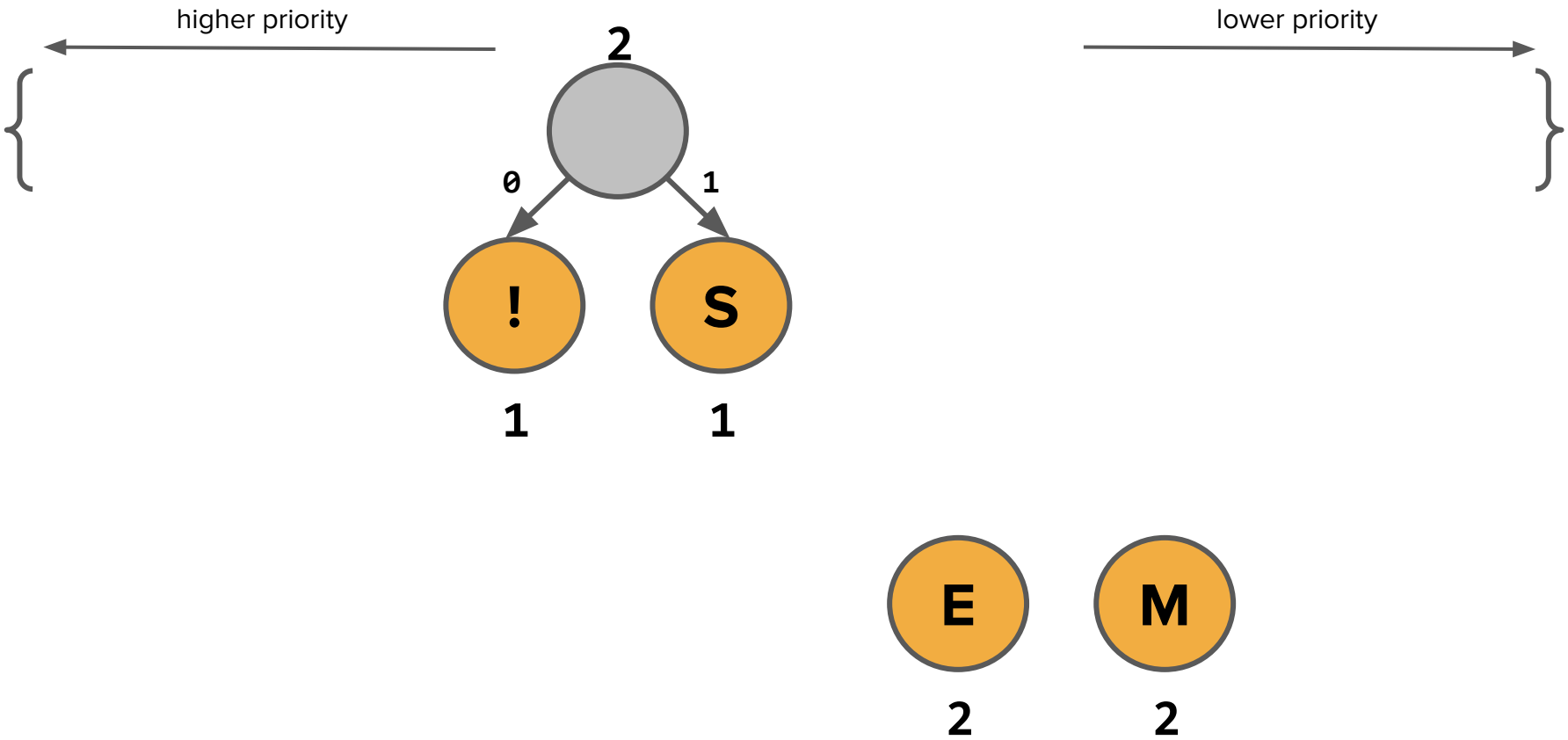
4) Build the Huffman tree by joining adjacent nodes



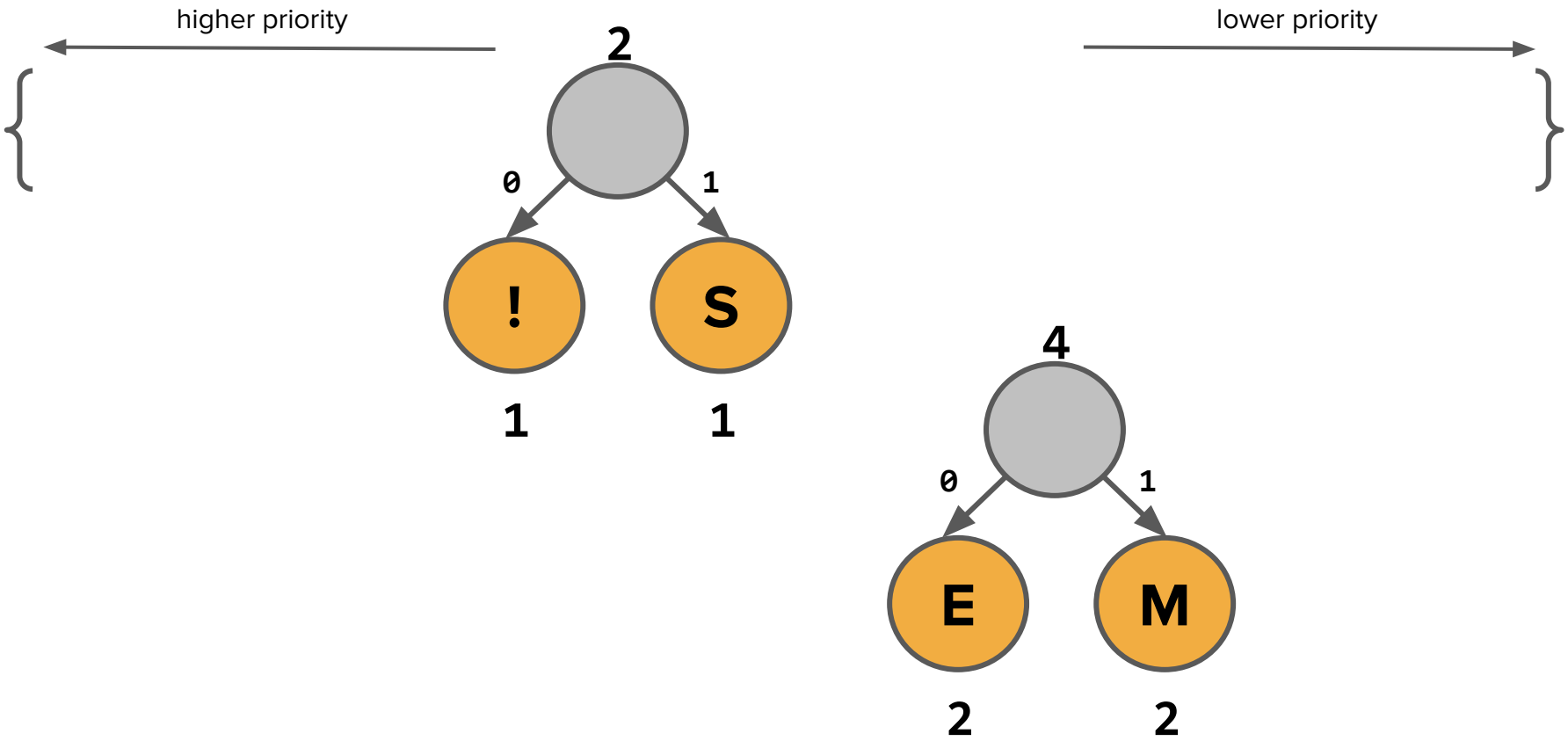
4) Build the Huffman tree by joining adjacent nodes



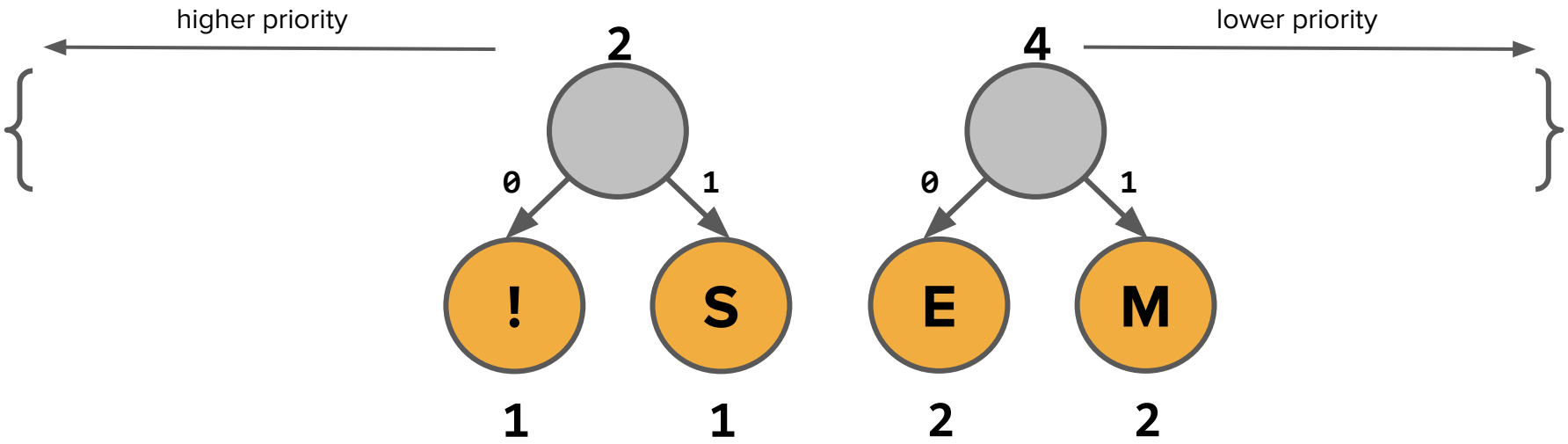
4) Build the Huffman tree by joining adjacent nodes

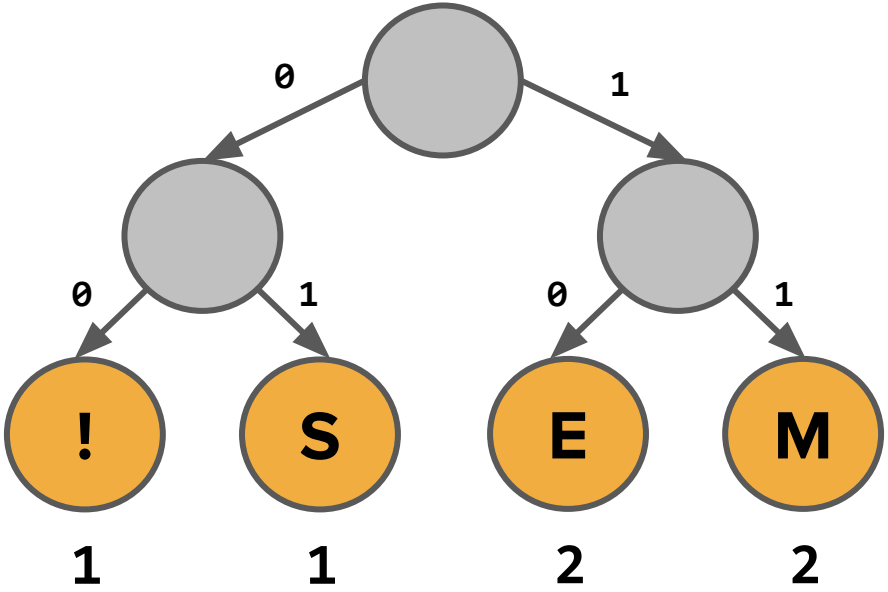


4) Build the Huffman tree by joining adjacent nodes

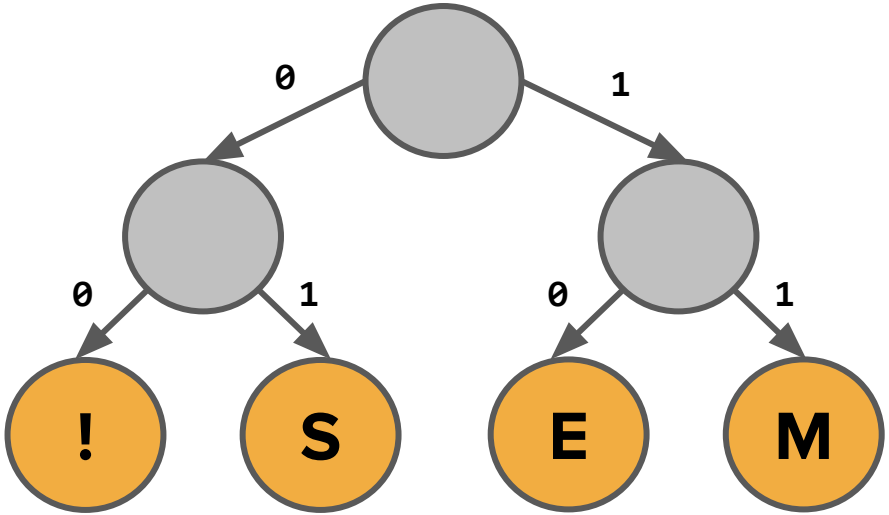


4) Build the Huffman tree by joining adjacent nodes

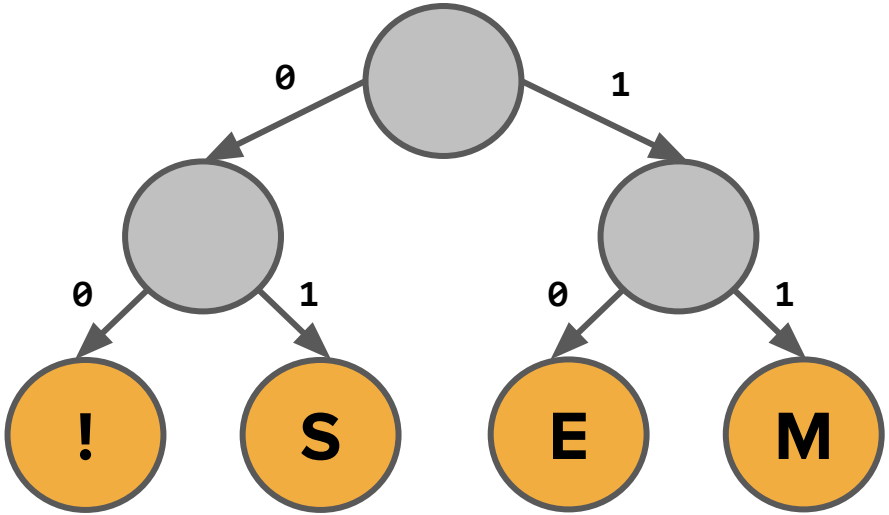



0
1
0
1
0
1
!
S
E
M
1
1
2
2

character	code
M	11
E	10
S	01
!	00

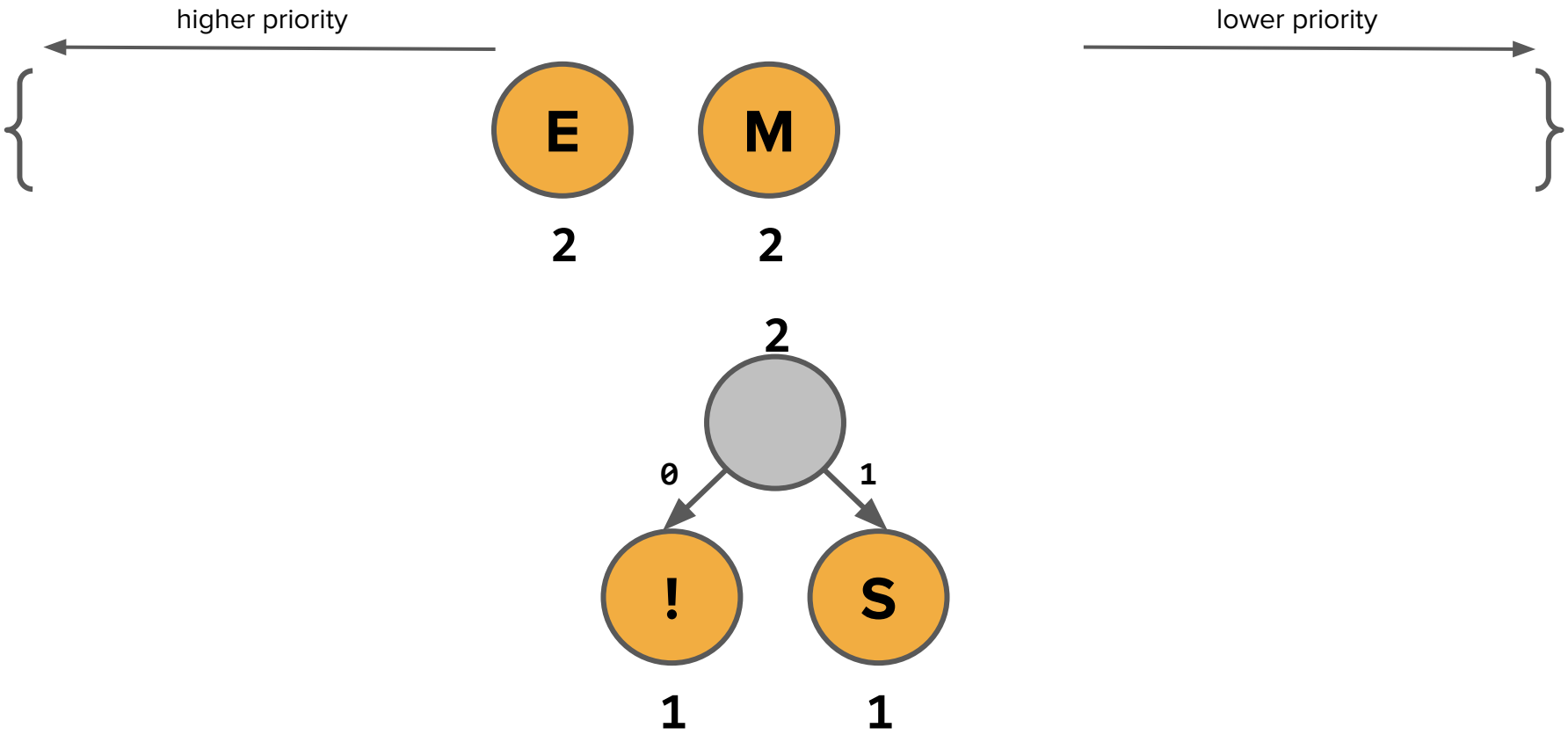


character	code
M	11
E	10
S	01
!	00

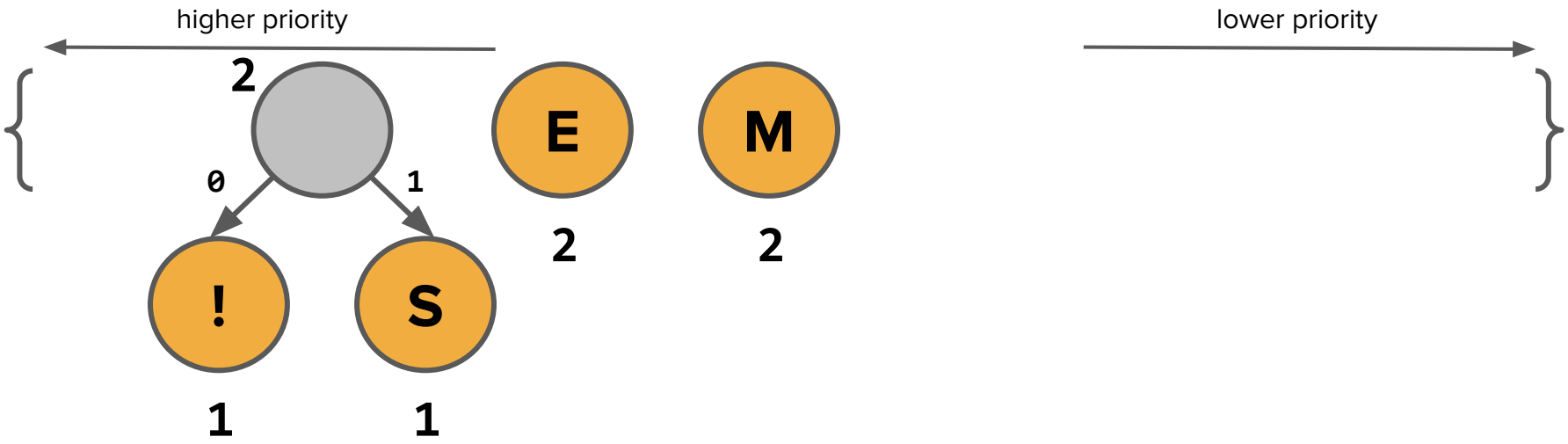


Other valid trees are possible!

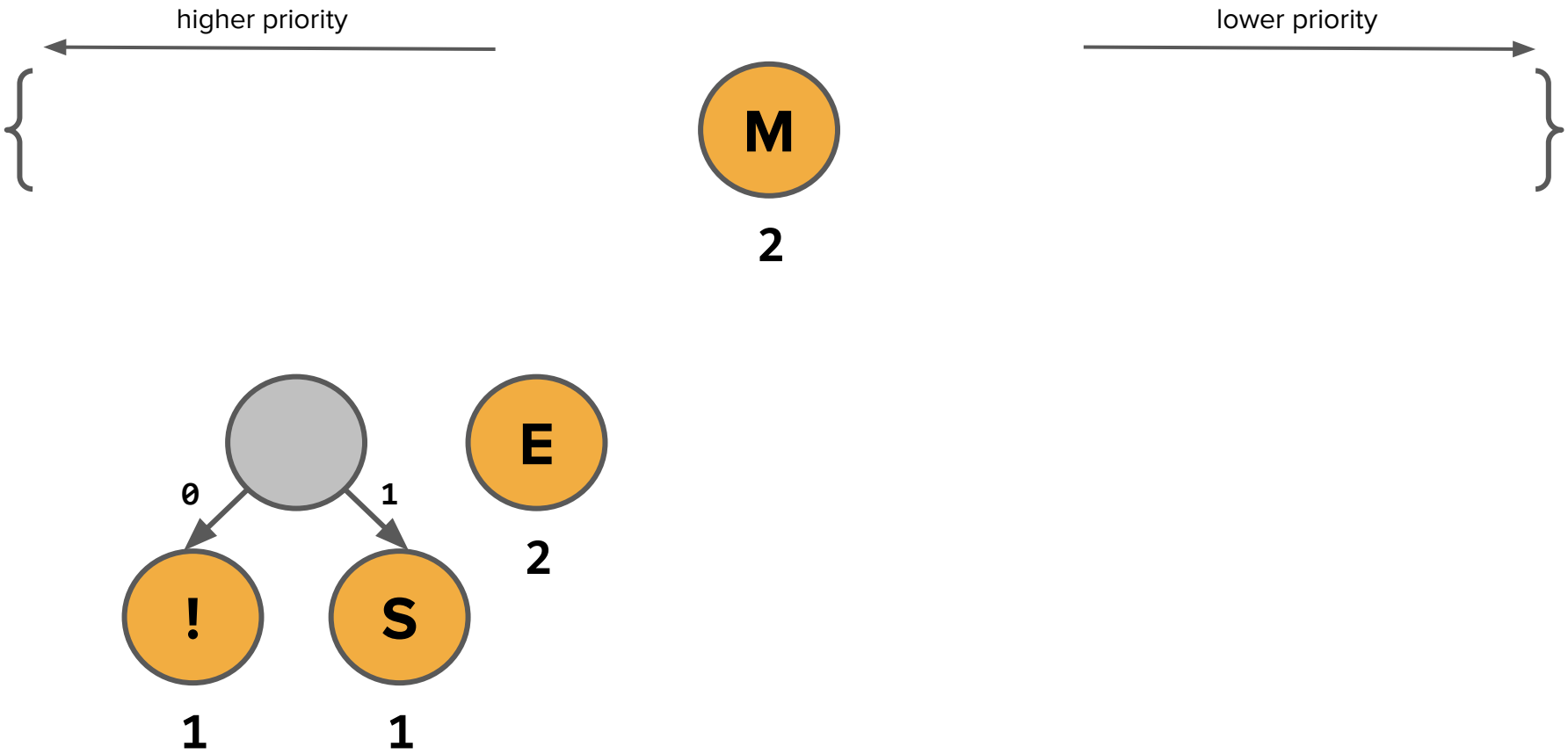
4) Build the Huffman tree by joining adjacent nodes



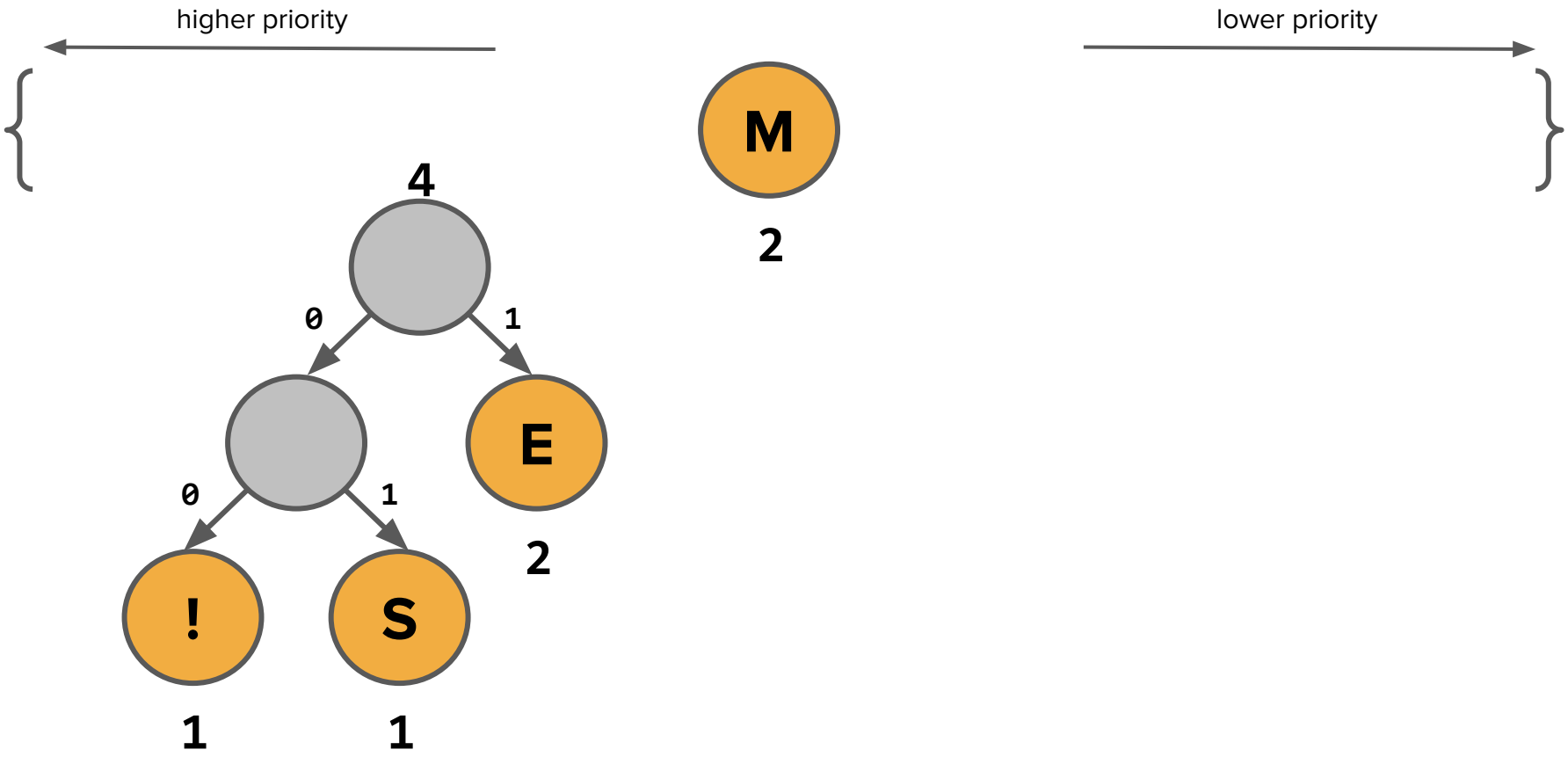
4) Build the Huffman tree by joining adjacent nodes



4) Build the Huffman tree by joining adjacent nodes



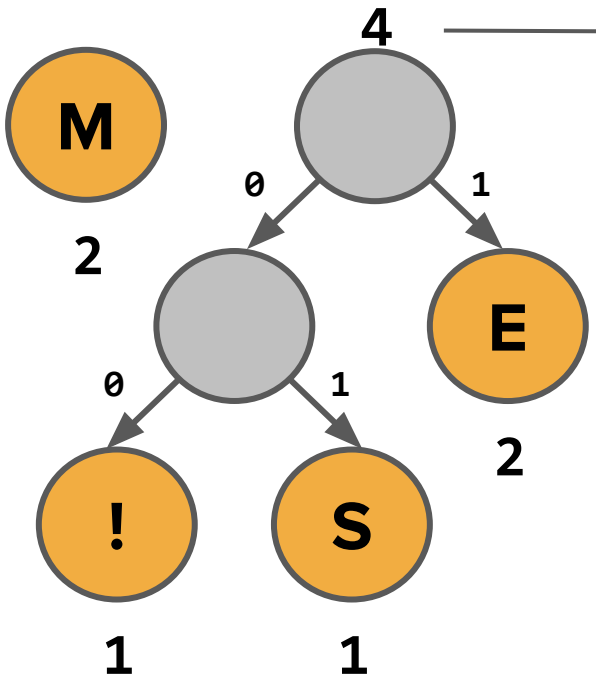
4) Build the Huffman tree by joining adjacent nodes

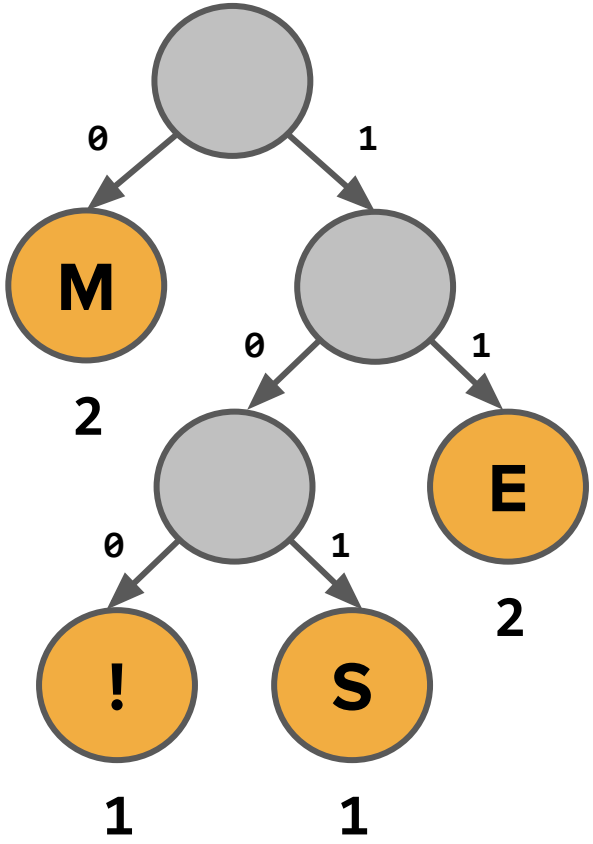


4) Build the Huffman tree by joining adjacent nodes

higher priority

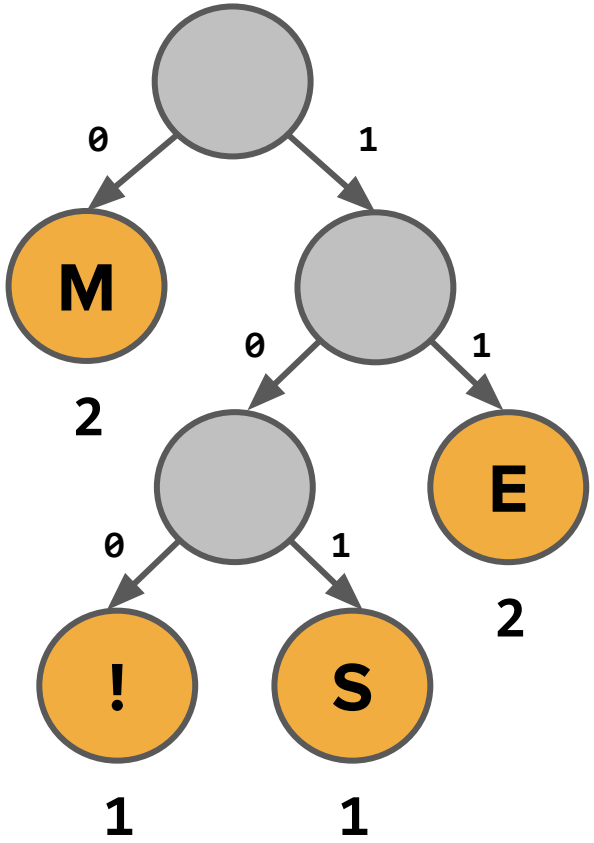
lower priority




0
1
M
2
0
1
E
2
0
1
!
1
S
1

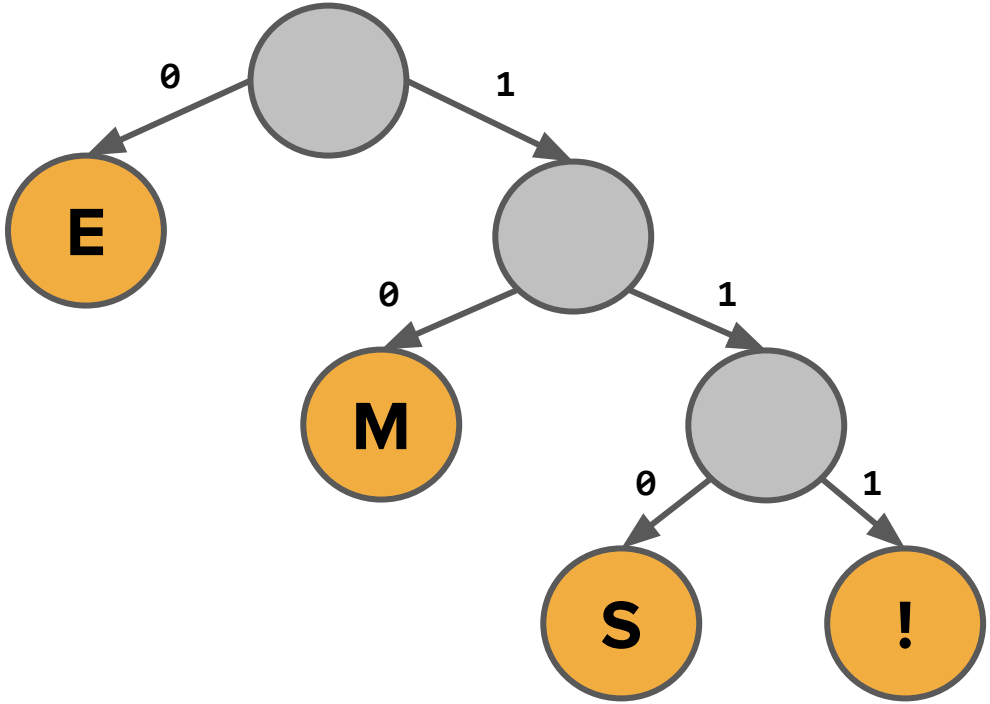
A second tree!

character	code
M	0
E	11
S	101
!	100



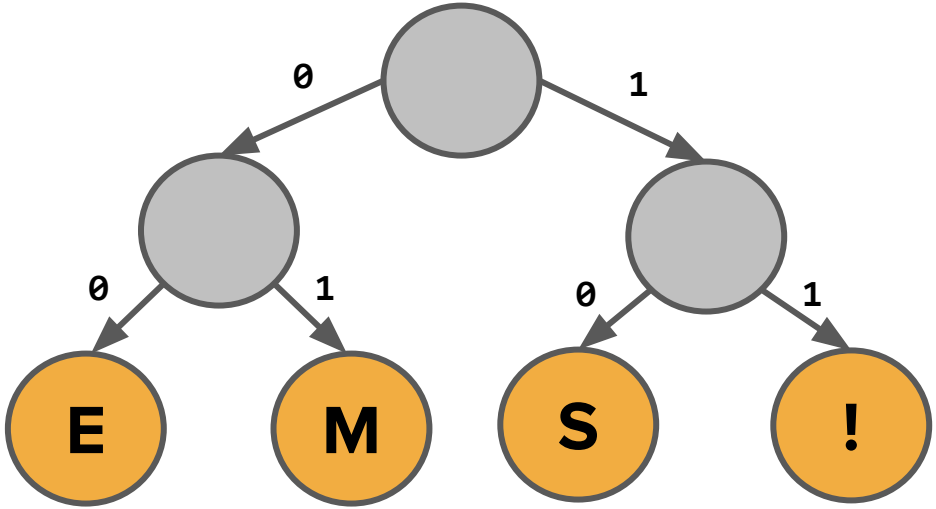
More options!

character	code
M	10
E	0
S	110
!	111



More options!

character	code
M	01
E	00
S	10
!	11



Huffman encoding summary

- Data compression is a very important real-world problem that relies on patterns in data to find efficient, compact data representations schemes.

- In order to support variable-length encodings for data, we must use prefix coding schemes. Prefix coding schemes can be modeled as binary trees.

- Huffman encoding uses a greedy algorithm to construct encodings by building a tree from the bottom up, putting the most frequent characters higher up in the coding tree.

- We need to send the encoding table with the compressed message.

What is hashing?

ADT Big-O Matrix

- Vectors
  - `.size() - O(1)`
  - `.add() - O(1)`
  - `v[i] - O(1)`
  - `.insert() - O(n)`
  - `.remove() - O(n)`
  - `.clear() - O(n)`
  - `traversal - O(n)`

- Grids
  - `.numRows()/.numCols() - O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() - O(1)`
  - `traversal - O(n^2)`

- Queues
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.enqueue() - O(1)`
  - `.dequeue() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`

- Stacks
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.push() - O(1)`
  - `.pop() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`

- Sets
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `.add() - O(log(n))`
  - `.remove() - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

- Maps
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `m[key] - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

ADT Big-O Matrix

- Vectors
  - `.size()` – O(1)
  - `.add()` – O(1)
  - `v[i]` – O(1)
  - `.insert()` – O(n)
  - `.remove()` – O(n)
  - `.clear()` - O(n)
  - `traversal` – O(n)

- Grids
  - `.numRows()/.numCols()` – O(1)
  - `g[i][j]` – O(1)
  - `.inBounds()` – O(1)
  - `traversal` – $O(n^2)$

- Queues
  - `.size()` – O(1)
  - `.peek()` – O(1)
  - `.enqueue()` – O(1)
  - `.dequeue()` – O(1)
  - `.isEmpty()` – O(1)
  - `traversal` – O(n)

- Stacks
  - `.size()` – O(1)
  - `.peek()` – O(1)
  - `.push()` – O(1)
  - `.pop()` – O(1)
  - `.isEmpty()` – O(1)
  - `traversal` – O(n)

- Sets
  - `.size()` – O(1)
  - `.isEmpty()` – O(1)
  - `.add()` – O(log(n))
  - `.remove()` – O(log(n))
  - `.contains()` – O(log(n))
  - `traversal` – O(n)

- Maps
  - `.size()` – O(1)
  - `.isEmpty()` – O(1)
  - `m[key]` – O(log(n))
  - `.contains()` – O(log(n))
  - `traversal` – O(n)

ADT Big-O Matrix

- Vectors
  - `.size() - O(1)`
  - `.add() - O(1)`
  - `v[i] - O(1)`
  - `.insert() - O(n)`
  - `.remove() - O(n)`
  - `.clear() - O(n)`
  - `traversal - O(n)`

- Grids
  - `.numRows()/.numCols() - O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() - O(1)`
  - `traversal - O(n²)`

- Queues
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.enqueue() - O(1)`
  - `.dequeue() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`

- Stacks
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.push() - O(1)`
  - `.pop() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`

- Sets
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `.add() - O(log(n))`
  - `.remove() - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

- Maps
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `m[key] - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

Can we do better???

Can we get constant runtime? (**O(1)**)

Can we get constant runtime? (**O(1)**)

An idea…

Can we get constant runtime? (**O(1)**)

An idea...

- Use an array implementation for the set, and when the user adds a value **i** to your set, store it at index **i** in the array.

```
set.add(7);
set.add(1);
set.add(9);
...
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	1	0	0	0	0	0	7	0	9
size	3	*capacity*			10					

Can we get constant runtime? (**`O(1)`**)

An idea...

- Use an array implementation for the set, and when the user adds a value **`i`** to your set, store it at index **`i`** in the array.

- This would give us constant time lookup!

Can we get constant runtime? (**`O(1)`**)

An idea...

- Use an array implementation for the set, and when the user adds a value **`i`** to your set, store it at index **`i`** in the array.

- This would give us constant time lookup!

What are the problems with this approach...?

Can we get constant runtime? (**O(1)**)

An idea...

- Use an array implementation for the set, and when the user adds a value **`i`** to your set, store it at index **`i`** in the array.

- This would give us constant time lookup!

What are the problems with this approach...?

```
set.add(1000);
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	1	0	0	0	0	0	7	0	9
size	3	*capacity*			10					

Key Idea: Can we get $O(1)$ runtime for lookup operations while only using a fixed amount of space?

Hash functions

Definition

hash function

A function that takes in arbitrary inputs and maps them to a fixed set of outputs.

Remember nameHash?

```
int nameHash(string first, string last){
    static const int kLargePrime = 16908799;
    static const int kSmallPrime = 127;
    int hashVal = 0;

    /* Iterate across all the characters in the first name, then the last name */
    for (char ch: first + last) {
        ch = tolower(ch);
        hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
    }
    return hashVal;
}
```

This is a hash function!

What is a hash function?

- Given an input of a particular type (e.g. string), returns a corresponding **hash value** (usually a number).
  - The values returned by a hash function are called “hash values,” “hash codes,” or “hashes.”

What is a hash function?

- Given an input of a particular type (e.g. string), returns a corresponding **hash value** (usually a number).
  - The values returned by a hash function are called “hash values,” “hash codes,” or “hashes.”

- Two important properties
  1. If given the same input, the hash function must return the same output. (This is also called a **deterministic** function.)

What is a hash function?

- Given an input of a particular type (e.g. string), returns a corresponding **hash value** (usually a number).
  - The values returned by a hash function are called “hash values,” “hash codes,” or “hashes.”

- Two important properties
  1. If given the same input, the hash function must return the same output. (This is also called a **deterministic** function.)
  2. Two different inputs will (usually) produce different outputs, even if the inputs are very similar.

What is a hash function?

- Given an input of a particular type (e.g. string), returns a corresponding **hash value** (usually a number).
  - The values returned by a hash function are called “hash values,” “hash codes,” or “hashes.”

- Two important properties
  1. If given the same input, the hash function must return the same output. (This is also called a **deterministic** function.)
  2. Two different inputs will (usually) produce different outputs, even if the inputs are very similar.

- Designing hash functions is beyond the scope of CS106B! But in the second half of this lecture, we’ll discuss how to use them.

Announcements

Announcements

- Please note that there are **no more grace periods or revisions** given the end-of-quarter timeline!

- Final project reports are due on **Sunday, August 22 at 11:59pm PDT**. You will have the opportunity to schedule your final presentation time with your SL after submitting. Reports should be submitted to Paperless, and time slot sign-ups will also happen through Paperless.

- Assignment 7 is due on **Monday, August 23 at 11:59pm PDT**.

- Friday's lecture is open final project OH. There are no group OH next week.

Hashing and ADTs

ADT Big-O Matrix

- Vectors
  - `.size() - O(1)`
  - `.add() - O(1)`
  - `v[i] - O(1)`
  - `.insert() - O(n)`
  - `.remove() - O(n)`
  - `.clear() - O(n)`
  - `traversal - O(n)`

- Grids
  - `.numRows()/.numCols() - O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() - O(1)`
  - `traversal - O(n²)`

- Queues
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.enqueue() - O(1)`
  - `.dequeue() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`

- Stacks
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.push() - O(1)`
  - `.pop() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`

- Sets
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `.add() - O(log(n))`
  - `.remove() - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

- Maps
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `m[key] - O(log(n))`
  - `.contains() - O(log(n))`
  - `traversal - O(n)`

Can we do better???

Levels of abstraction

What is the interface for the user?
(**Hash**Sets, **Hash**Maps)

Abstract Data Structures

How is our data organized?
(**hash table**)

Data Organization Strategies

What stores our data?
(arrays, linked lists)

Fundamental C++ Data Storage

How is data represented electronically?
(RAM)

Computer Hardware

Creating a **hash table** for data organization

Creating a **hash table** for data organization

- Maintain a large number of small collections called **buckets** (think drawers).
  - Put together, the buckets form a **hash table**!

- Find a **rule** that lets us tell where each object should go (think knowing which drawer is which).

- To find something, only look in the bucket assigned to it (think looking for socks).

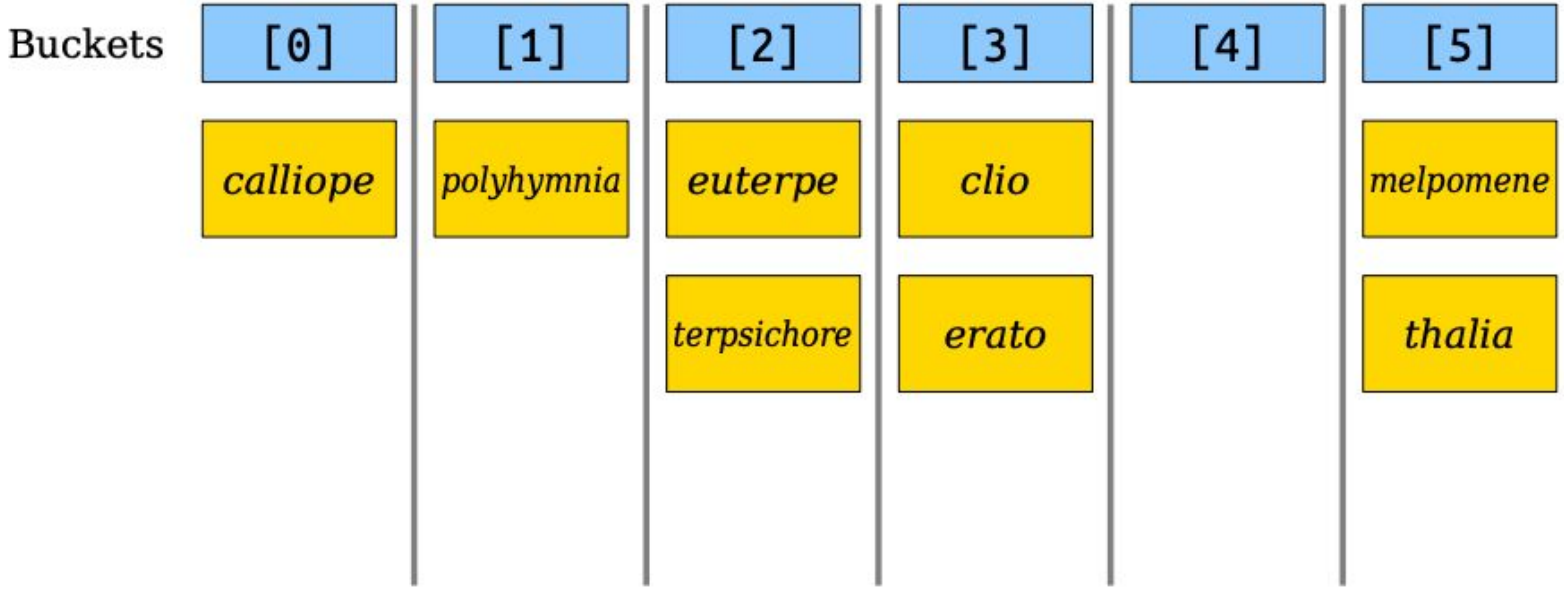
Creating a **hash table** for data organization

- Maintain a large number of small collections called **buckets** (think drawers).
  - Put together, the buckets form a **hash table**!

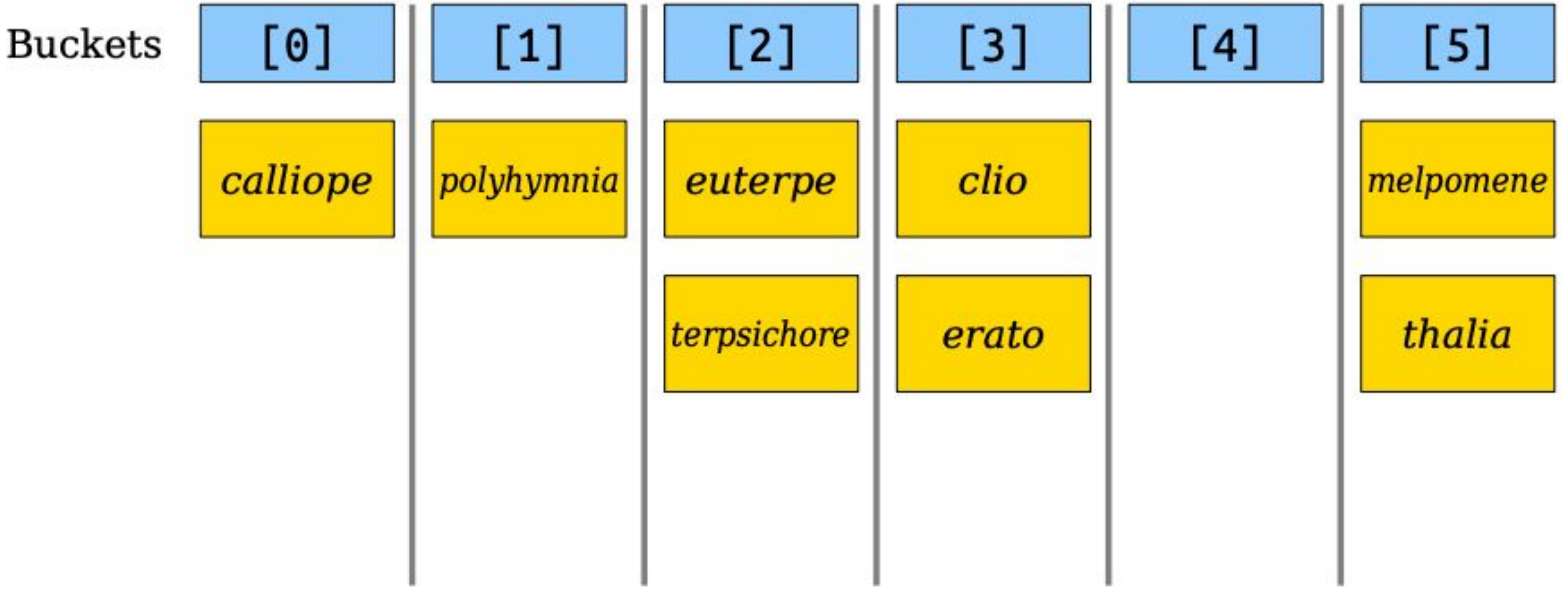
- Find a **rule** that lets us tell where each object should go (think knowing which drawer is which).

- To find something, only look in the bucket assigned to it (think looking for socks).

Use a hash function!

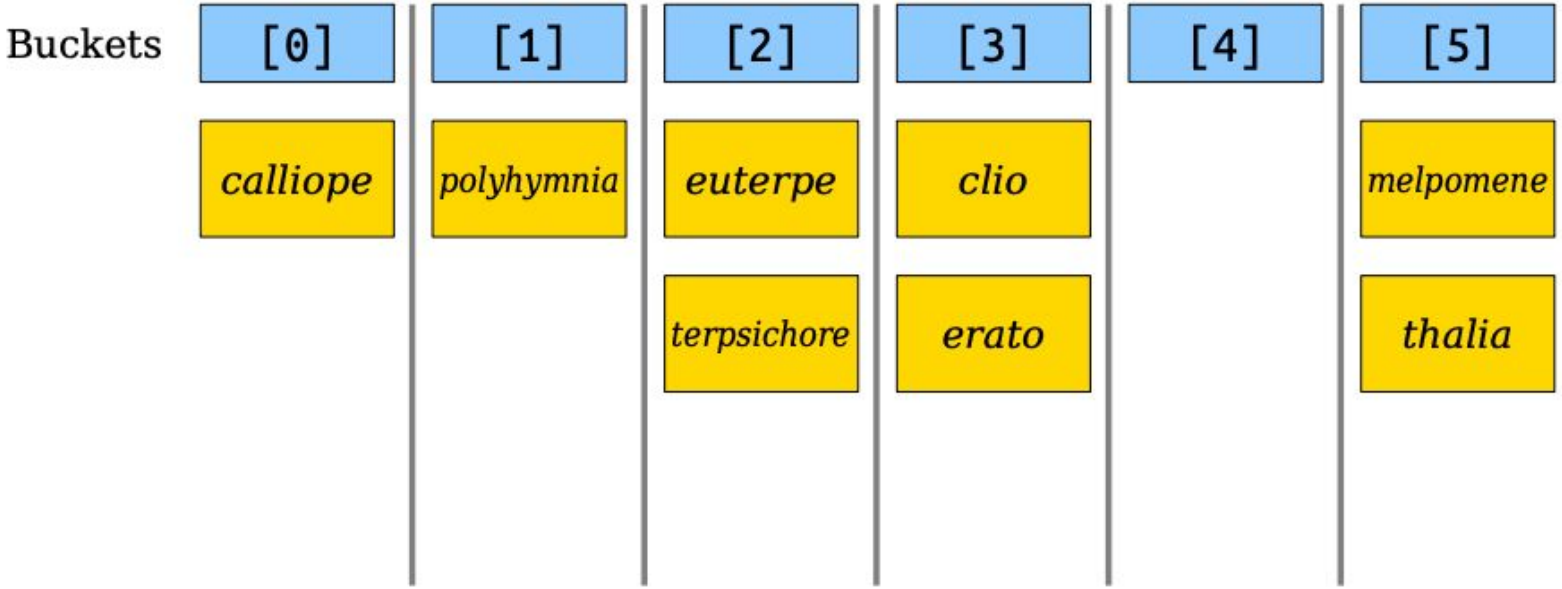


set.add(*urania*)



Our bucket rule:

```
bucket = hash(input) % numBuckets;
```

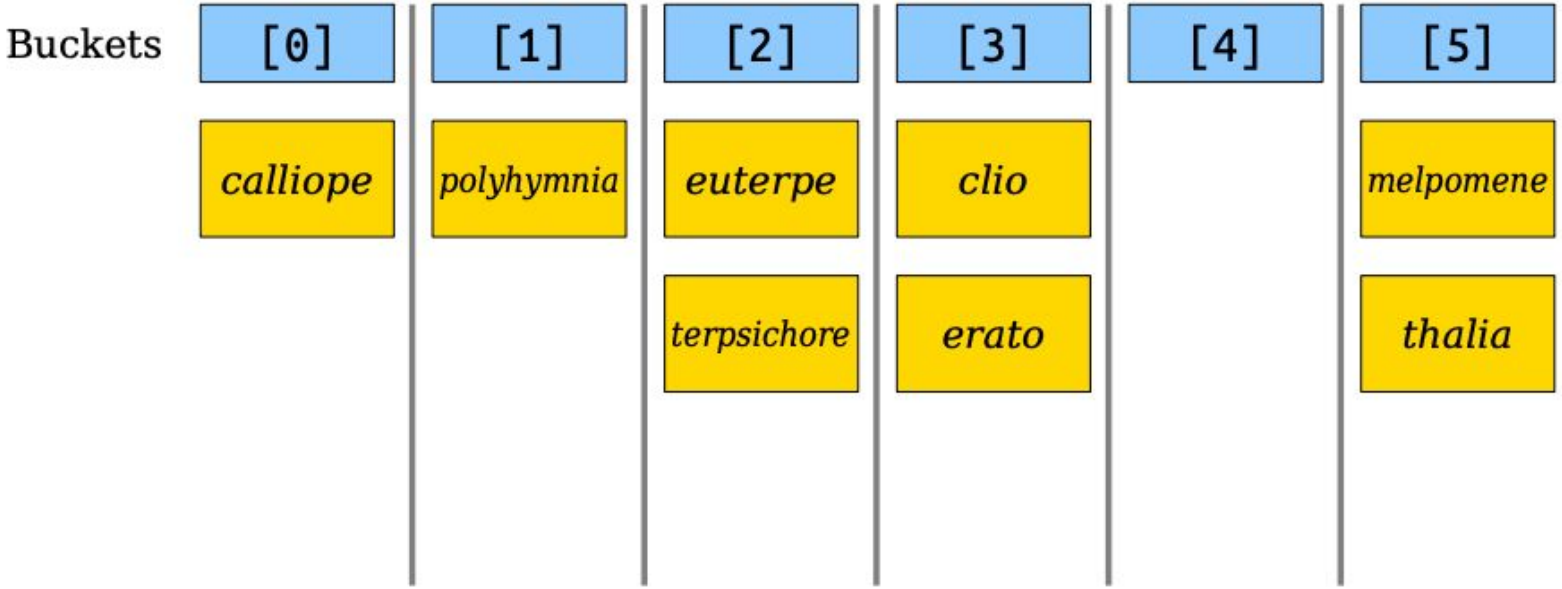


Our bucket rule:

```
bucket = hash(urania) % numBuckets;
```

12206 6

set.add(*urania*)

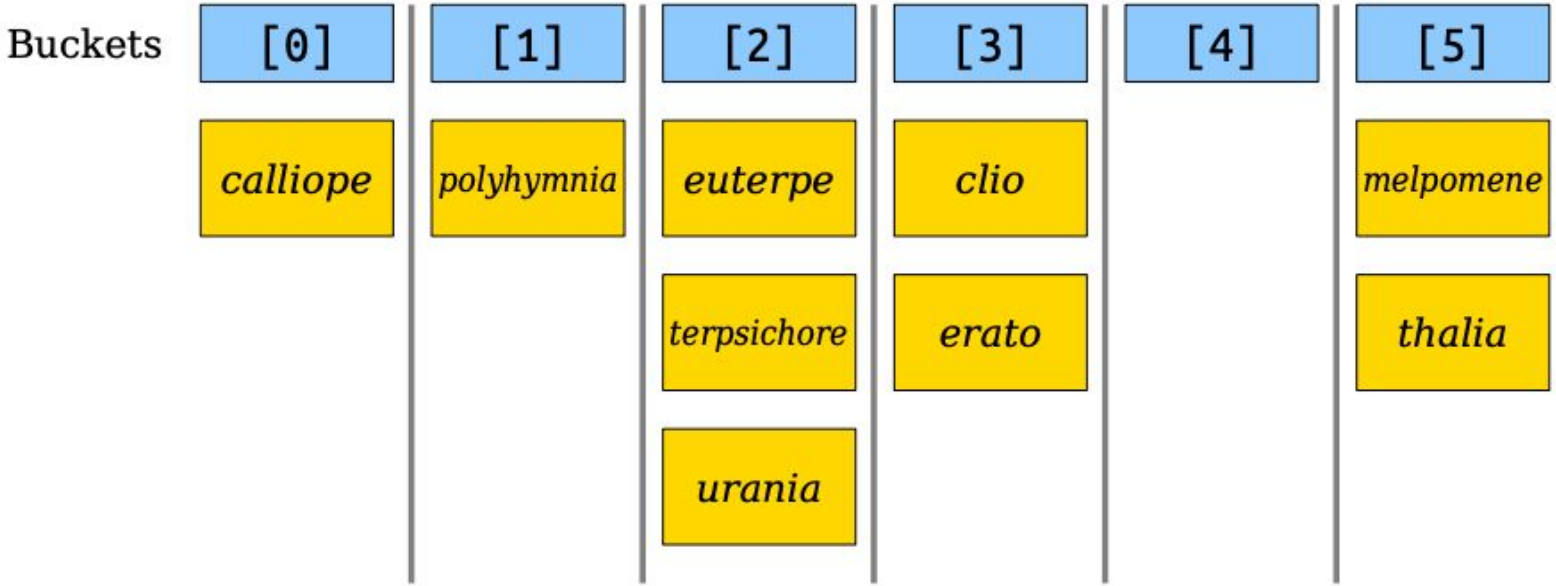


Our bucket rule:

```
bucket = hash(urania) % numBuckets;
```

2 12206 6

set.add(*urania*)



Our bucket rule:

```
bucket = hash(input) % numBuckets;
```

Creating a **hash table** for data organization

- Maintain a large number of small collections called **buckets** (think drawers).
  - Put together, the buckets form a **hash table**!

- Find a **rule** that lets us tell where each object should go (think knowing which drawer is which).

- To find something, only look in the bucket assigned to it (think looking for socks).

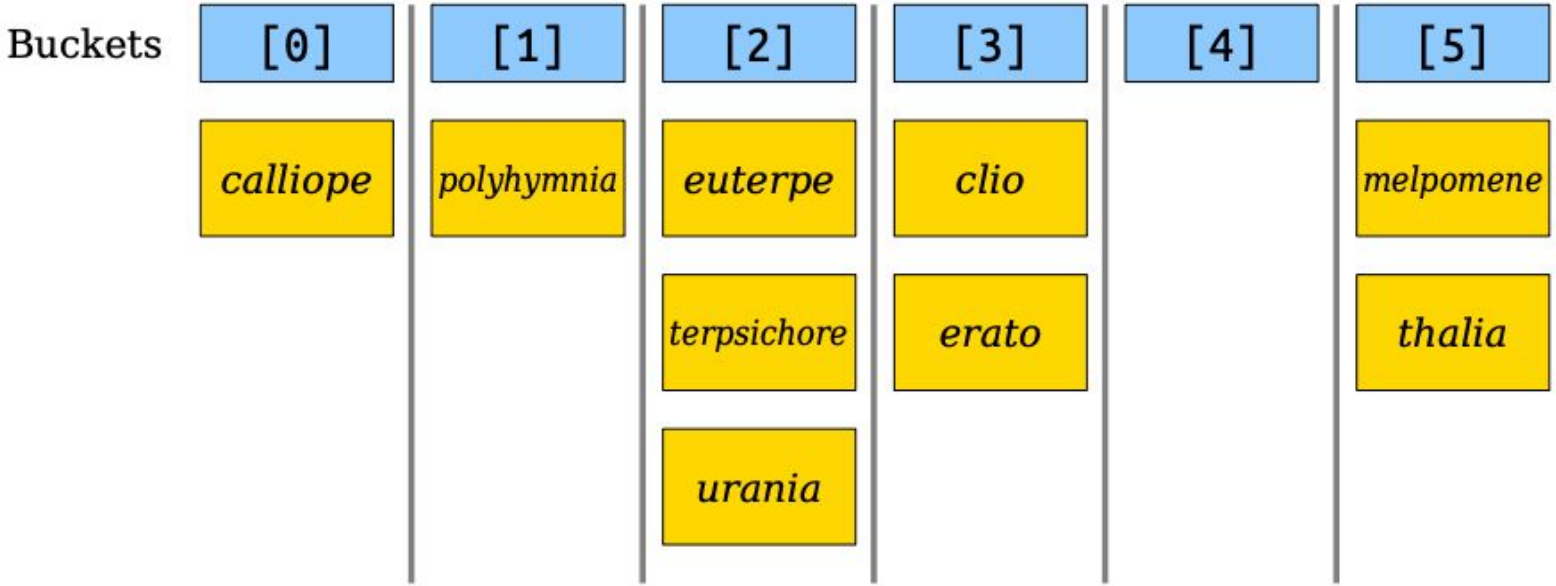
Creating a **hash table** for data organization

- Maintain a large number of small collections called **buckets** (think drawers).
  - Put together, the buckets form a **hash table**!

- Find a **rule** that lets us tell where each object should go (think knowing which drawer is which).

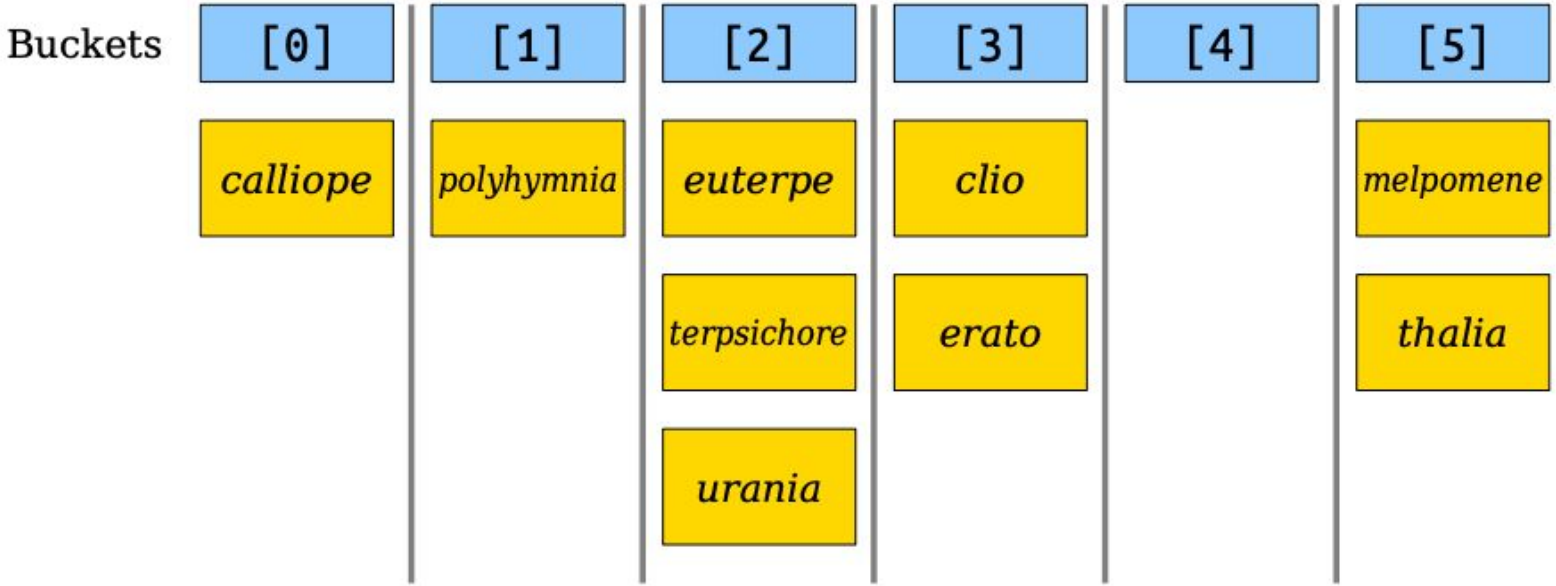
- To find something, only look in the bucket assigned to it (think looking for socks).

We can use the same rule for lookup!



Our bucket rule:

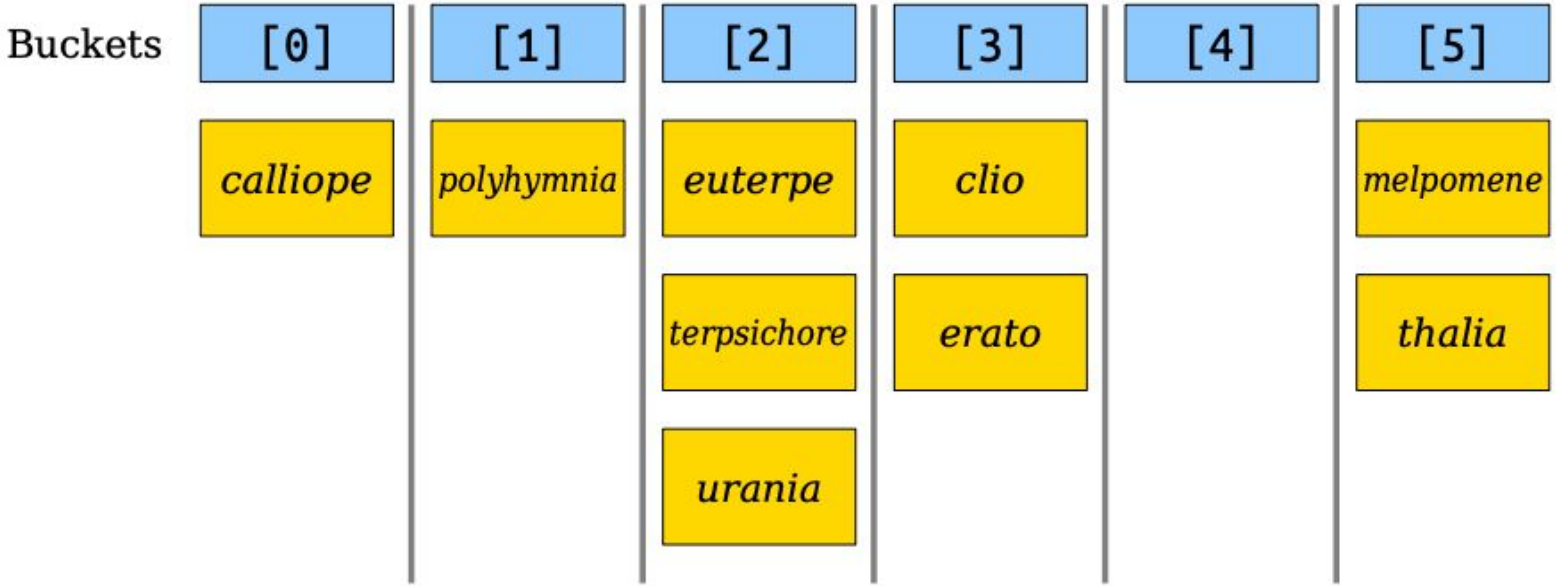
```
bucket = hash(input) % numBuckets;
```



Our bucket rule:

```
bucket = hash(urania) % numBuckets;
```

```
set.contains(urania)
```

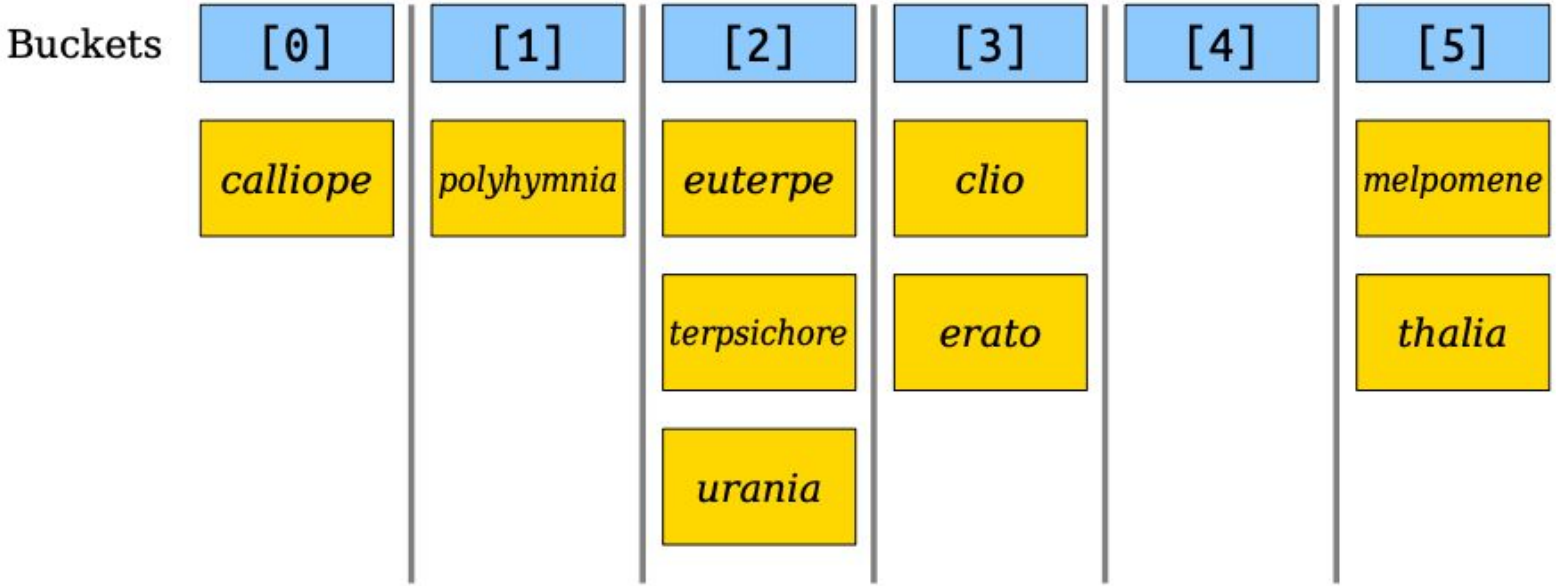


Our bucket rule:

```
bucket = hash(urania) % numBuckets;
```

2

```
set.contains(urania)
```



Look in bucket 2 and traverse until you find **urania** or run out of elements.

Our bucket rule:

```
bucket = hash(urania) % numBuckets;
```

2

```
set.contains(urania)
```

Creating a **hash table** for data organization

- Maintain a large number of small collections called **buckets** (think drawers).
  - Put together, the buckets form a **hash table**!

- Find a **rule** that lets us tell where each object should go (think knowing which drawer is which).

- To find something, only look in the bucket assigned to it (think looking for socks).

How efficient is this?

- Each hash table operation:
  - Chooses a bucket and jumps there
  - Potentially scans everything in the bucket

- **Claim**: The efficiency of our hash table depends on how well-spread-out the elements are.
  - If we want O(1) lookup operations, we want our buckets to have a size of ~1 element on average.

How efficient is this?

- Let’s suppose we have a “strong” hash function that distributes elements fairly evenly.

How efficient is this?

- Let's suppose we have a "strong" hash function that distributes elements fairly evenly.
  - Recall our two hash function properties:
    1. If given the same input, the hash function must return the same output. (This is also called a **deterministic** function.)
    2. Two different inputs will (usually) produce different outputs, even if the inputs are very similar.

How efficient is this?

- Let's suppose we have a "strong" hash function that distributes elements fairly evenly.
  - Recall our two hash function properties:
    1. If given the same input, the hash function must return the same output. (This is also called a **deterministic** function.)
    2. Any given input value will give a "random" output → this creates a relatively equal distribution across buckets.

How efficient is this?

- Let's suppose we have a "strong" hash function that distributes elements fairly evenly.

- Imagine we have **b** buckets and **n** elements in our table.

How efficient is this?

- Let's suppose we have a "strong" hash function that distributes elements fairly evenly.
- Imagine we have **b** buckets and **n** elements in our table.
- On average, how many elements will be in a bucket?

How efficient is this?

- Let's suppose we have a "strong" hash function that distributes elements fairly evenly.
- Imagine we have **b** buckets and **n** elements in our table.
- On average, how many elements will be in a bucket?

$$n / b$$

How efficient is this?

- Let's suppose we have a "strong" hash function that distributes elements fairly evenly.

- Imagine we have **b** buckets and **n** elements in our table.

- On average, how many elements will be in a bucket?

$$n / b$$

- The expected cost of an insertion, deletion, or lookup is therefore:

$$O(1 + n / b)$$

Load factor

- We call α `= n / b` our **load factor**.
- If α gets too big, the hash table will be too slow.
- If α gets too low, the hash table will waste too much space.

Load factor

- We call α `= n / b` our **load factor**.
- If α gets too big, the hash table will be too slow.
- If α gets too low, the hash table will waste too much space.
- **Idea**: If α gets too big, we need to resize our underlying buckets array and rehash the values to new buckets in the larger array.
  - We double our number of buckets when we hit a particular threshold for our load factor. (We'll use the threshold of α `>= 2`.)
  - Very similar to resizing our priority queue!

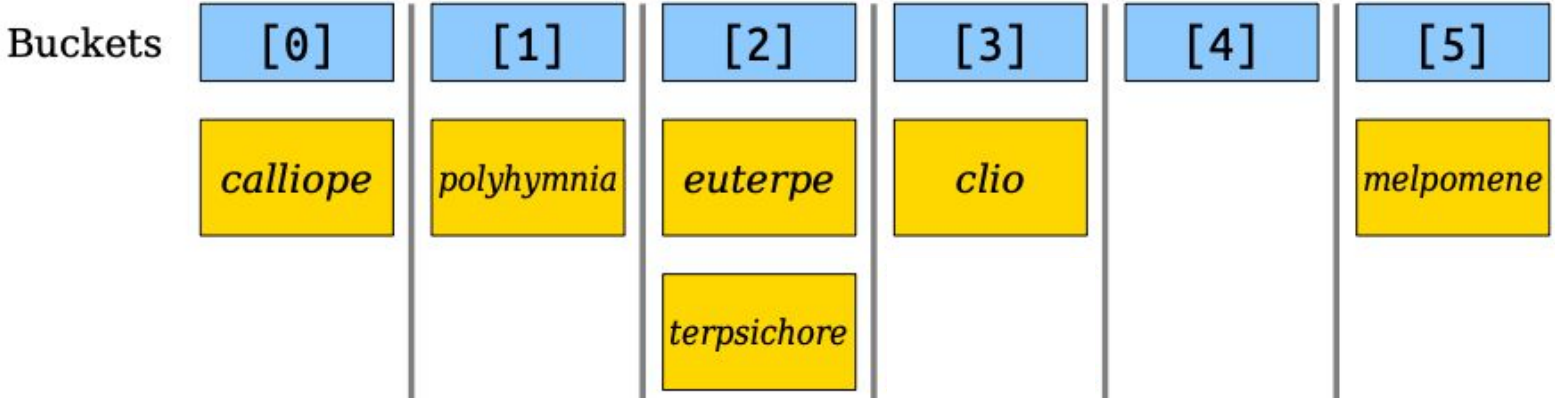
A note about collisions...

- In reality, our hash function will not distribute inputted elements exactly evenly across all buckets.
  - This gives us “collisions”!

A note about collisions...

- In reality, our hash function will not distribute inputted elements exactly evenly across all buckets.
  - This gives us “collisions”!

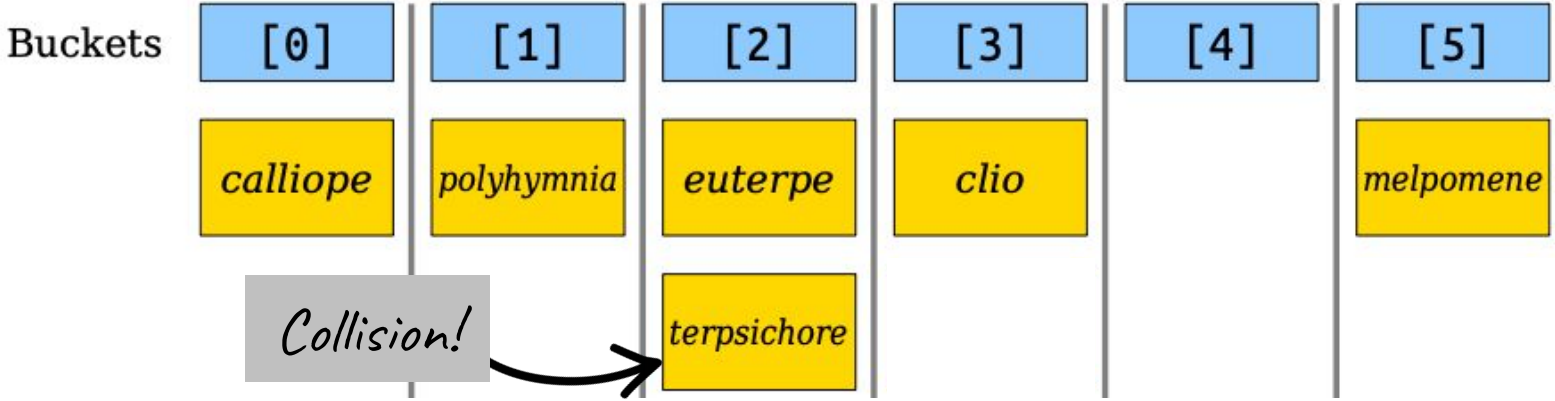
- A **collision** occurs when two or more elements map to the same bucket.



A note about collisions...

- In reality, our hash function will not distribute inputted elements exactly evenly across all buckets.
  - This gives us “collisions”!

- A **collision** occurs when two or more elements map to the same bucket.



A note about collisions...

- In reality, our hash function will not distribute inputted elements exactly evenly across all buckets.
  - This gives us “collisions”!

- A **collision** occurs when two or more elements map to the same bucket.

- To handle collisions, we use a strategy called **chaining**, in which each bucket stores a linked list of elements that point to one another.

Implementing a HashSet

HashSet.h

```
class HashSet {
public:
    HashSet();
    ~HashSet();
    void add(int value);
    void clear();
    bool contains(int value) const;

private:
    HashNode** elements;     // an array of HashNode* (an array of pointers!)
    int mysize;
    int capacity;
    int getIndexOf(int value) const;
    void rehash();
};
```

```
struct HashNode {
    int data;
    HashNode* next;
};
```

HashSet.cpp

```
#include "HashSet.h"

// Initialize our member variables in the constructor
HashSet::HashSet() {
    capacity = 10;
    mysize = 0;
    elements = new HashNode*[capacity](); // all are initialized to nullptr using ()
}

// Private helper for calculating the bucket of a given a value
int HashSet::getIndexOf(int value) const {
    return hash(value) % capacity;
}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {





}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {
    if (!contains(value)) {





    }
}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {
    if (!contains(value)) {
        int bucket = getIndexOf(value);




    }
}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {
    if (!contains(value)) {
        int bucket = getIndexOf(value);
        // insert at the front of the list in that bucket
        elements[bucket] = new HashNode(value, elements[bucket]);

    }
}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {
    if (!contains(value)) {
        int bucket = getIndexOf(value);
        // insert at the front of the list in that bucket
        elements[bucket] = new HashNode(value, elements[bucket]);
        mysize++;
    }
}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {
    if (!contains(value)) {
        int bucket = getIndexOf(value);
        // insert at the front of the list in that bucket
        elements[bucket] = new HashNode(value, elements[bucket]);
        mysize++;
    }
    // We'll add rehashing here later...
}
```

HashSet.cpp

```
// Check if a value is inside our set
bool HashSet::contains(int value) const {





}
```

HashSet.cpp

```
// Check if a value is inside our set
bool HashSet::contains(int value) const {
    HashNode* curr = elements[getIndexOf(value)];





}
```

HashSet.cpp

```
// Check if a value is inside our set
bool HashSet::contains(int value) const {
    HashNode* curr = elements[getIndexOf(value)];
    while (curr != nullptr) {





    }
    return false;
}
```

HashSet.cpp

```
// Check if a value is inside our set
bool HashSet::contains(int value) const {
    HashNode* curr = elements[getIndexOf(value)];
    while (curr != nullptr) {
        if (curr->data == value) {
            return true;
        }
        curr = curr->next;
    }
    return false;
}
```

HashSet.cpp

```
HashSet::~HashSet() {
    clear();                // Remove all elements
    delete[] elements;      // Also delete the array itself
}

// Remove all elements in our set so all buckets in our array are nullptr
void HashSet::clear() {









}
```

HashSet.cpp

```
HashSet::~HashSet() {
    clear();                // Remove all elements
    delete[] elements;      // Also delete the array itself
}

// Remove all elements in our set so all buckets in our array are nullptr
void HashSet::clear() {
    for (int i = 0; i < capacity; i++) {
        // free list in bucket i





    }
    mysize = 0;
}
```

HashSet.cpp

```
HashSet::~HashSet() {
    clear();                // Remove all elements
    delete[] elements;      // Also delete the array itself
}

// Remove all elements in our set so all buckets in our array are nullptr
void HashSet::clear() {
    for (int i = 0; i < capacity; i++) {
        // free list in bucket i
        while (elements[i] != nullptr) {
            HashNode* curListNode = elements[i];
            elements[i] = elements[i]->next;
            delete curListNode;
        }
    }
    mysize = 0;
}
```

HashSet.cpp

```
void HashSet::rehash() {
    HashNode** oldElements = elements;
    int oldCapacity = capacity;
    capacity *= 2;
    elements = new HashNode*[capacity]();
    for (int i = 0; i < oldCapacity; i++) {
        HashNode* curr = oldElements[i];
        while (curr != nullptr) {     // iterate over old bucket
            HashNode* prev = curr;
            curr = curr->next;        // don’t lose access to rest of old bucket
            int newBucket = getIndexOf(prev->data);
            prev->next = elements[newBucket]; // put prev node at front of new bucket
            elements[newBucket] = prev;       // update new bucket pointer
        }
    }
    delete[] oldElements;
}
```

HashSet.cpp

```
// Add a given value to our set
void HashSet::add(int value) {
    if (!contains(value)) {
        int bucket = getIndexOf(value);
        // insert at the front of the list in that bucket
        elements[bucket] = new HashNode(value, elements[bucket]);
        mysize++;
    }
    if (mysize / capacity >= 2) {
        rehash();
    }
}
```

`HashSet` takeaways

- When implementing **`HashSet`**s or **`HashMap`**s, we use an array to store pointers.
  - Each bucket in the array stores a pointer to a linked list in case of collisions.
  - To create a HashMap instead of a HashSet, your node struct would just include both a key and a value (instead of just one field for data).

- Our hash function tells us what bucket in the array our elements should go in.

- Because we can just add new nodes to the **front** of the linked list at the bucket indicated by its hash value, adding to a **`HashSet`** is **`O(1)`**.

- Functions that require lookup are also constant time (i.e. **`O(1)`**) if the **load factor** for our hash table is small!

Applications of Hashing

Hashing is used everywhere!

- In addition to creating hash tables, hash functions themselves are used in a variety of different applications.

- Applications hash your passwords before storing them to obscure the actual contents.

- Hashing is used in cryptography for secure (encrypted) communication and maintaining data integrity.
  - For example, when you communicate over a WiFi network: Is this website secure? Is the this document actually from the person it says it's from? Did your message get tampered with between when you sent it and when the recipient got it?

Hashzam!

(demo courtesy of Chris Piech)

How does it work?

- Attempt #1: Compare at all notes at every time stamp.

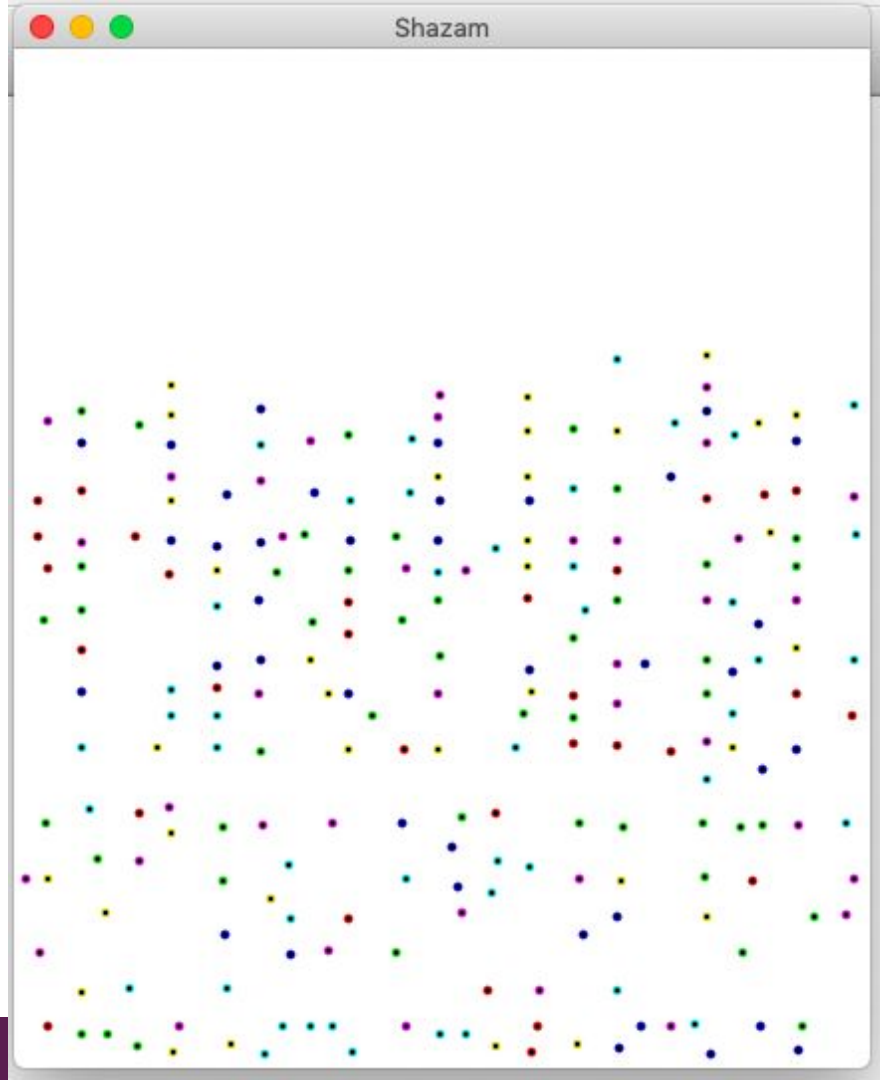
How does it work?

- Attempt #1: Compare at all notes at every time stamp.

- This would require storing all notes at every timestep, i.e. storing entire song files!

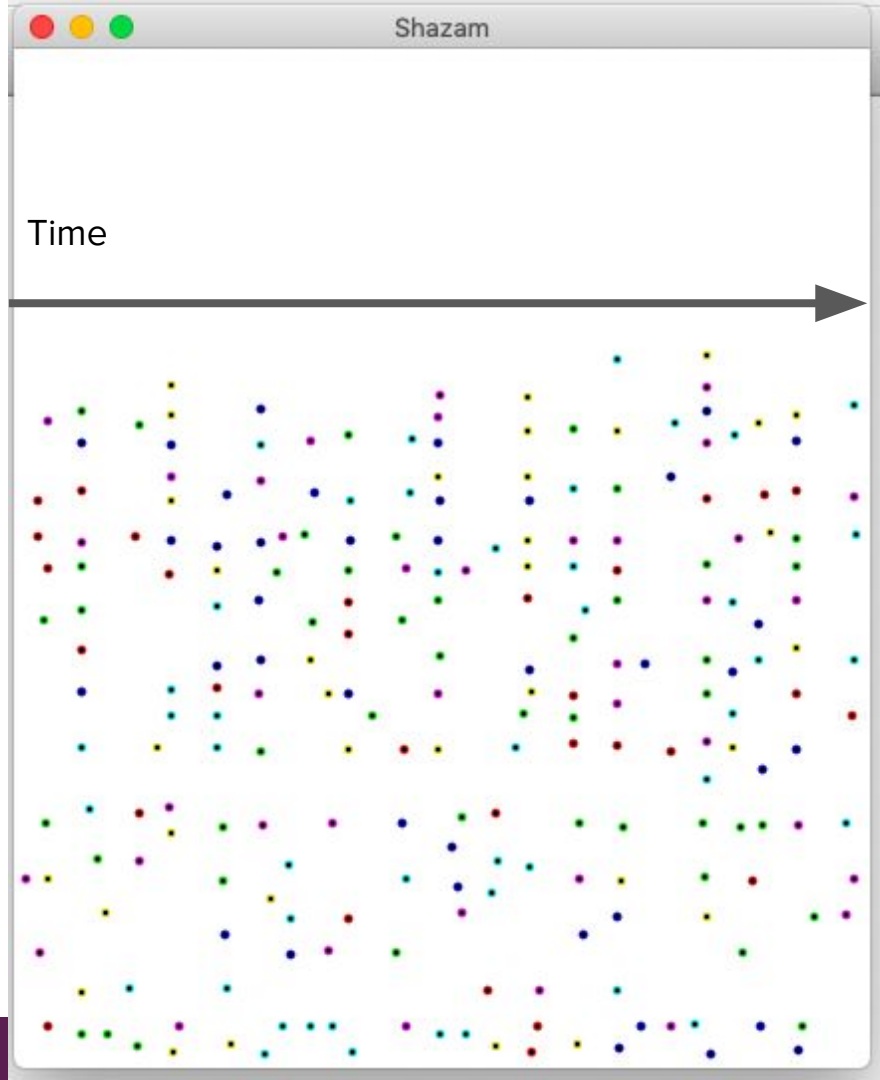
How does it work?

- Instead, look at notes that appear close to one another in time.



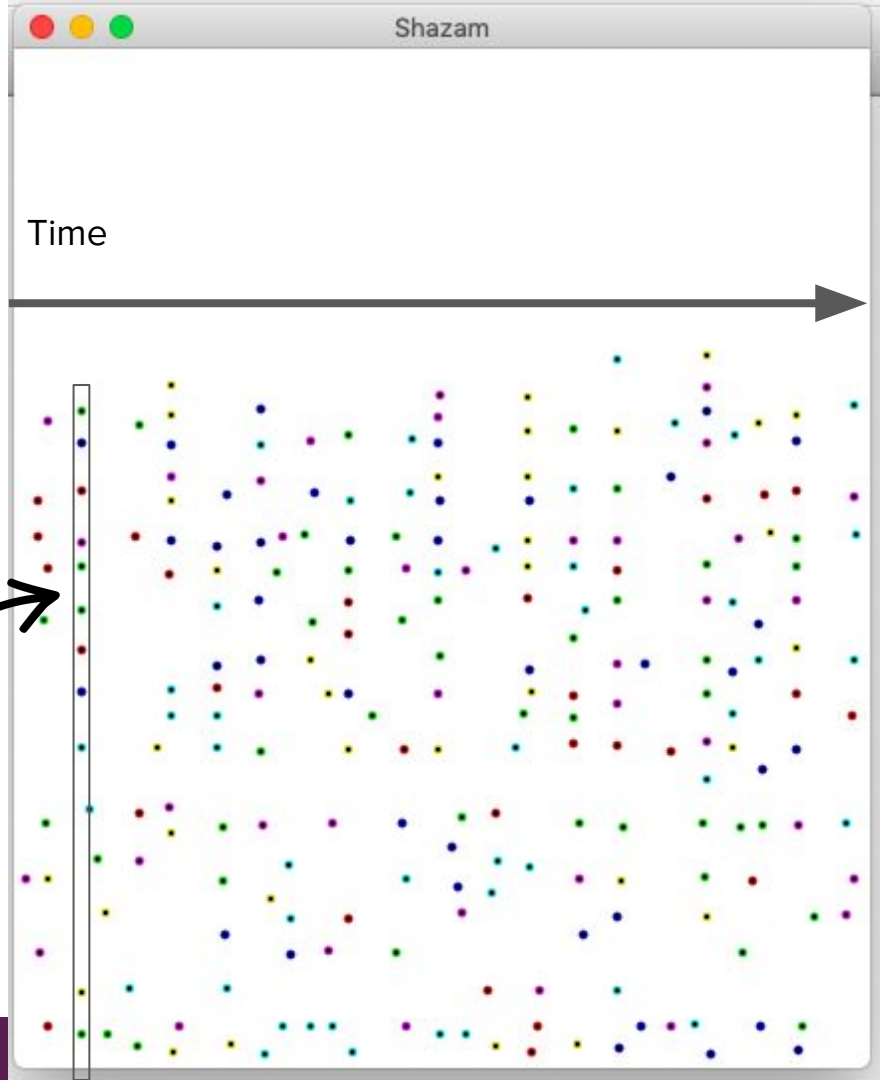
How does it work?

- Instead, look at notes that appear close to one another in time.



How does it work?

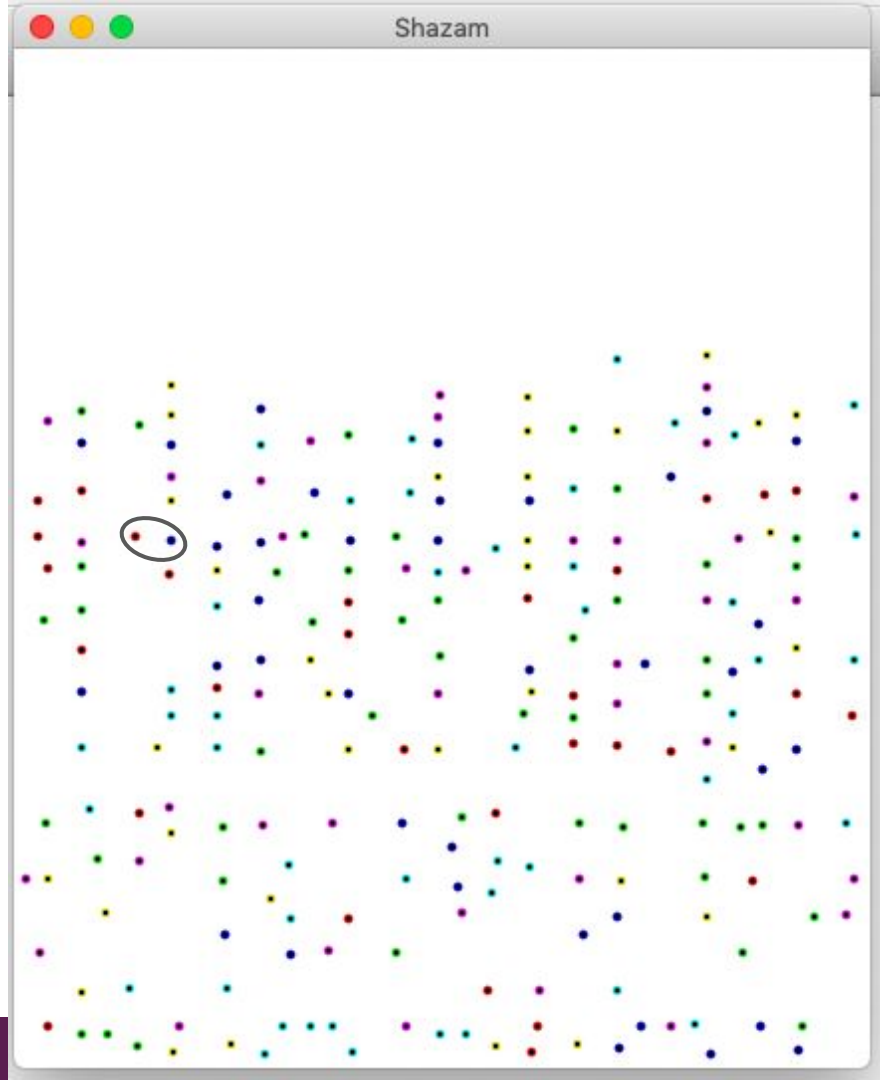
- Instead, look at notes that appear close to one another in time.



All notes at a given timestamp (stacked by pitch).

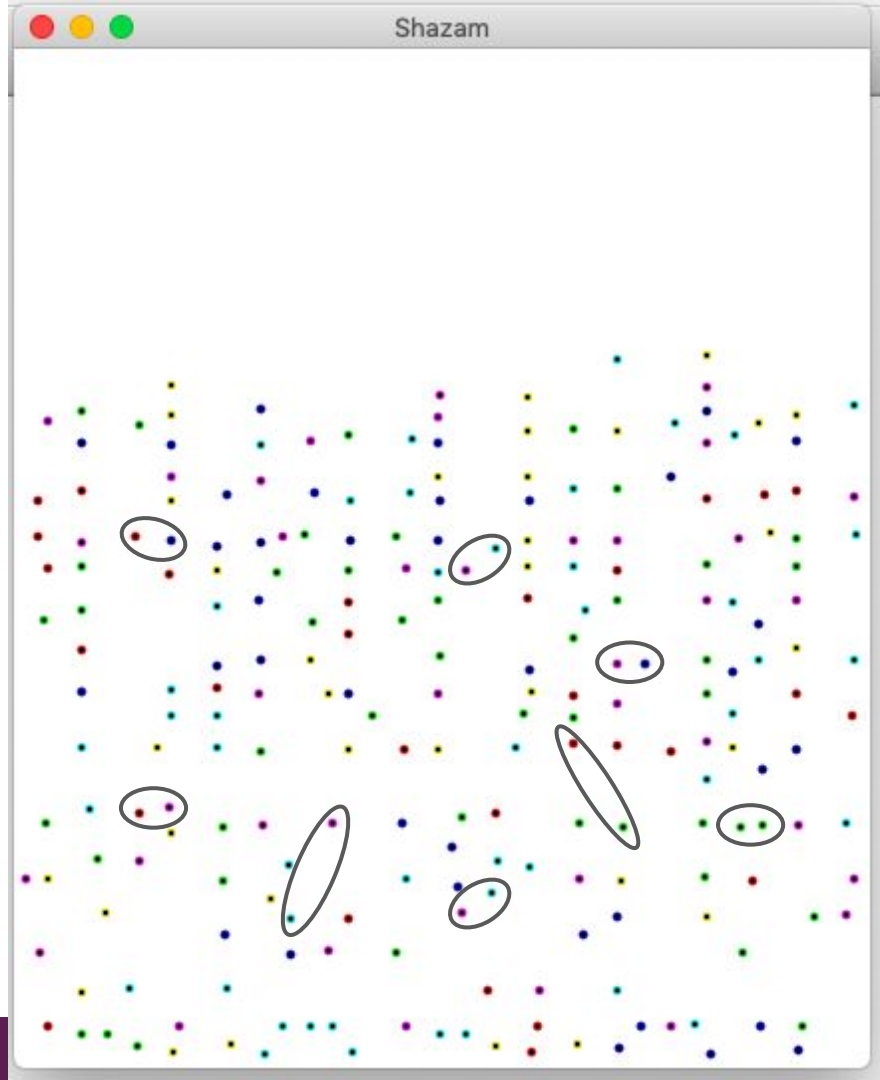
How does it work?

- Instead, look at notes that appear close to one another in time.



How does it work?

- Instead, look at notes that appear close to one another in time.



How does it work?

- Instead, look at notes that appear close to one another in time.

- Attempt #2: Store a frequency map of hashed values for each song.
  - For all notes within a certain timestep of one another throughout a given song, store the **`hash(noteA, noteB, timeDelta)`**.

How does it work?

- Instead, look at notes that appear close to one another in time.

- Attempt #2: Store a frequency map of hashed values for each song.
  - For all notes within a certain timestep of one another throughout a given song, store the **`hash(noteA, noteB, timeDelta)`**.
  - This gives you a frequency map of the hashed values for the song (i.e. hash values and their counts).

How does it work?

- Instead, look at notes that appear close to one another in time.

- Attempt #2: Store a frequency map of hashed values for each song.
  - For all notes within a certain timestep of one another throughout a given song, store the **`hash(noteA, noteB, timeDelta)`**.
  - This gives you a frequency map of the hashed values for the song (i.e. hash values and their counts).
  - Compare the song's frequency map to stored maps to find the closest match.

How does it work?

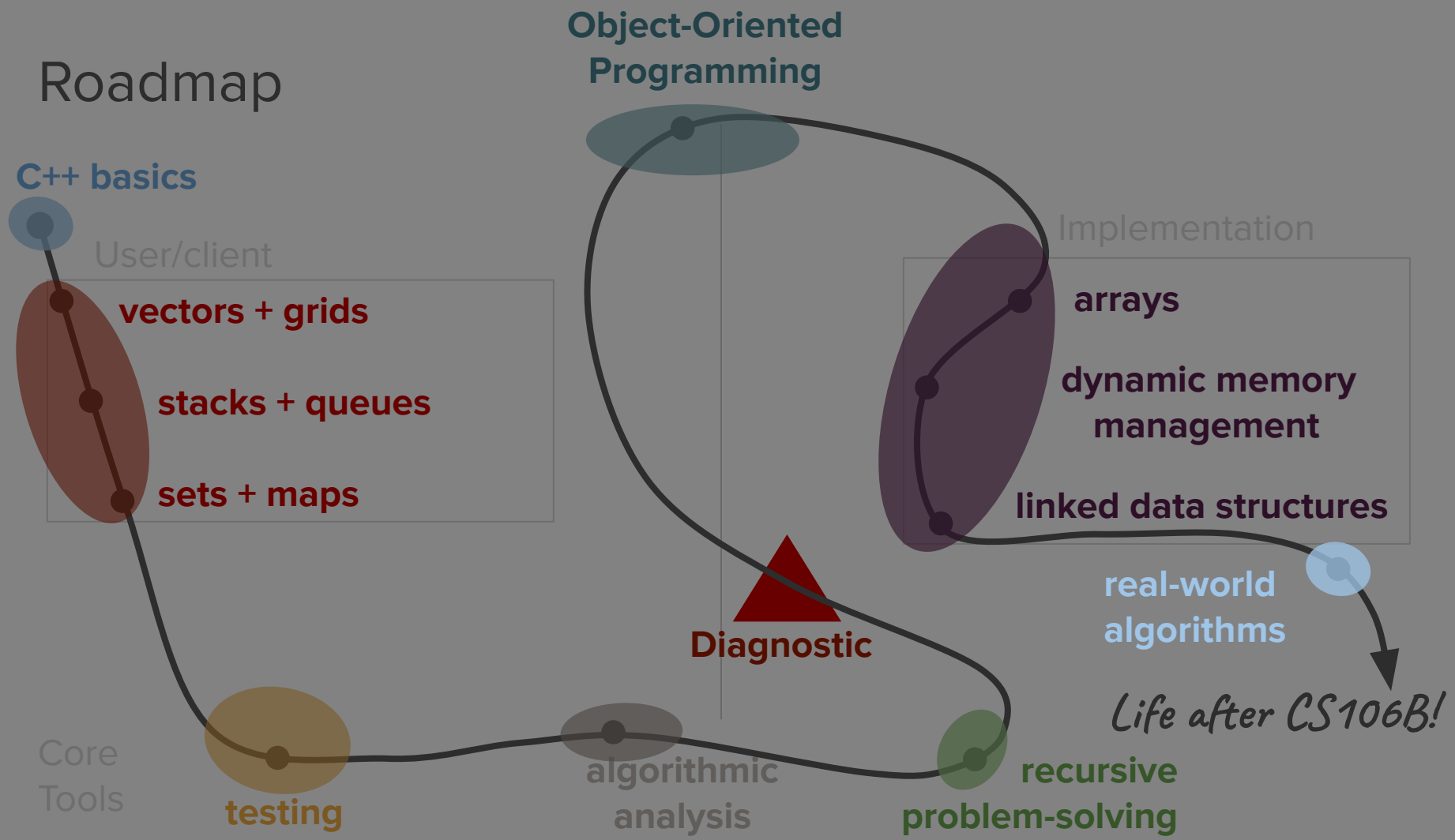
- Instead, look at notes that appear close to one another in time.

- Attempt #2: Store a frequency map of hashed values for each song.
  - For all notes within a certain timestep of one another throughout a given song, store the **`hash(noteA, noteB, timeDelta)`**.
  - This gives you a frequency map of the hashed values for the song (i.e. hash values and their counts).
  - Compare the song's frequency map to stored maps to find the closest match.

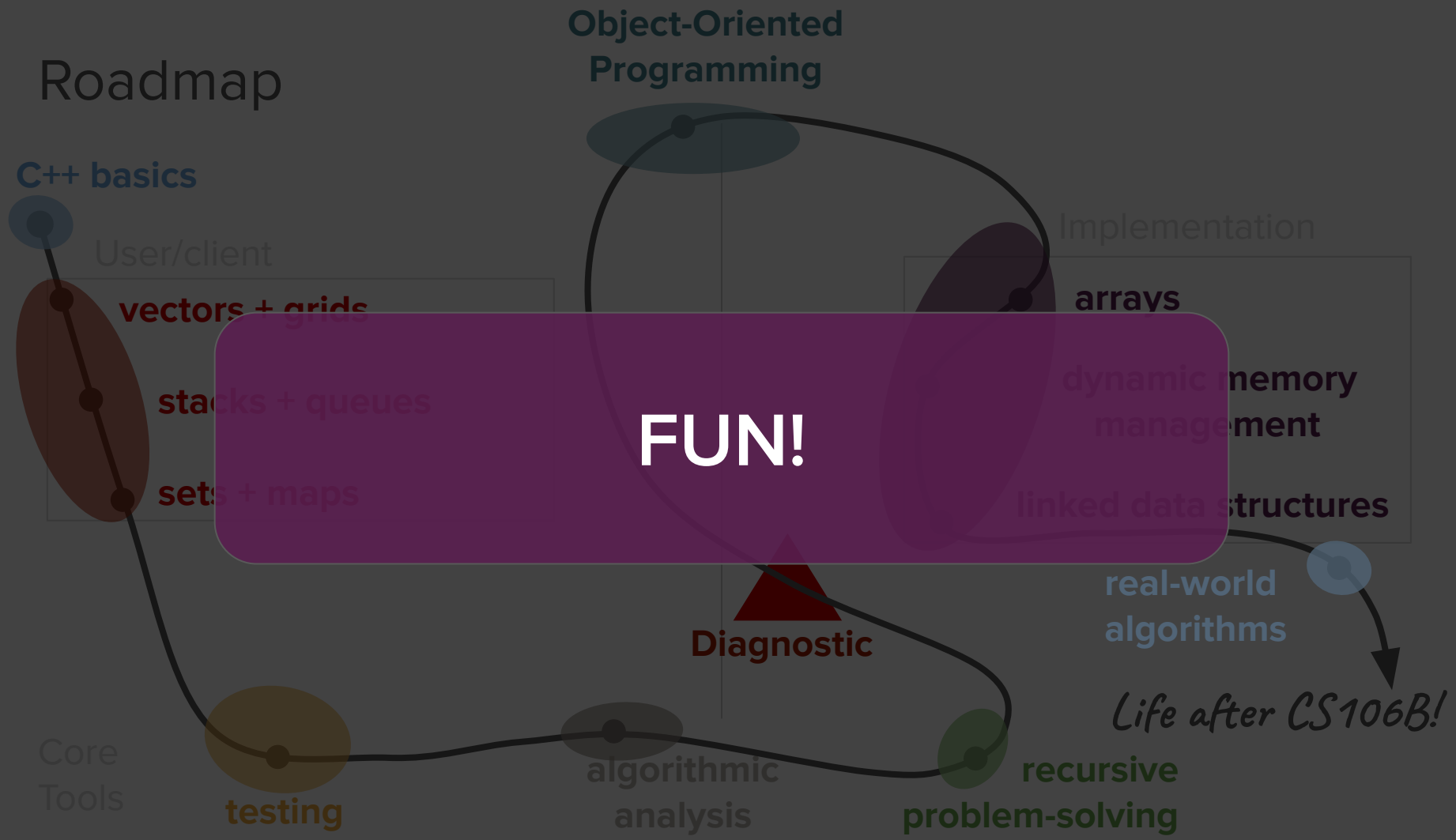
- You only have to store a database of frequency maps, which is more space efficient and enables easier comparison between songs!

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
Life after CS106B!
Diagnostic
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
FUN!
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving