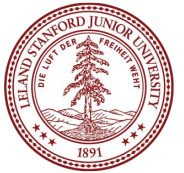
Unordered Data Structures: Sets and Maps

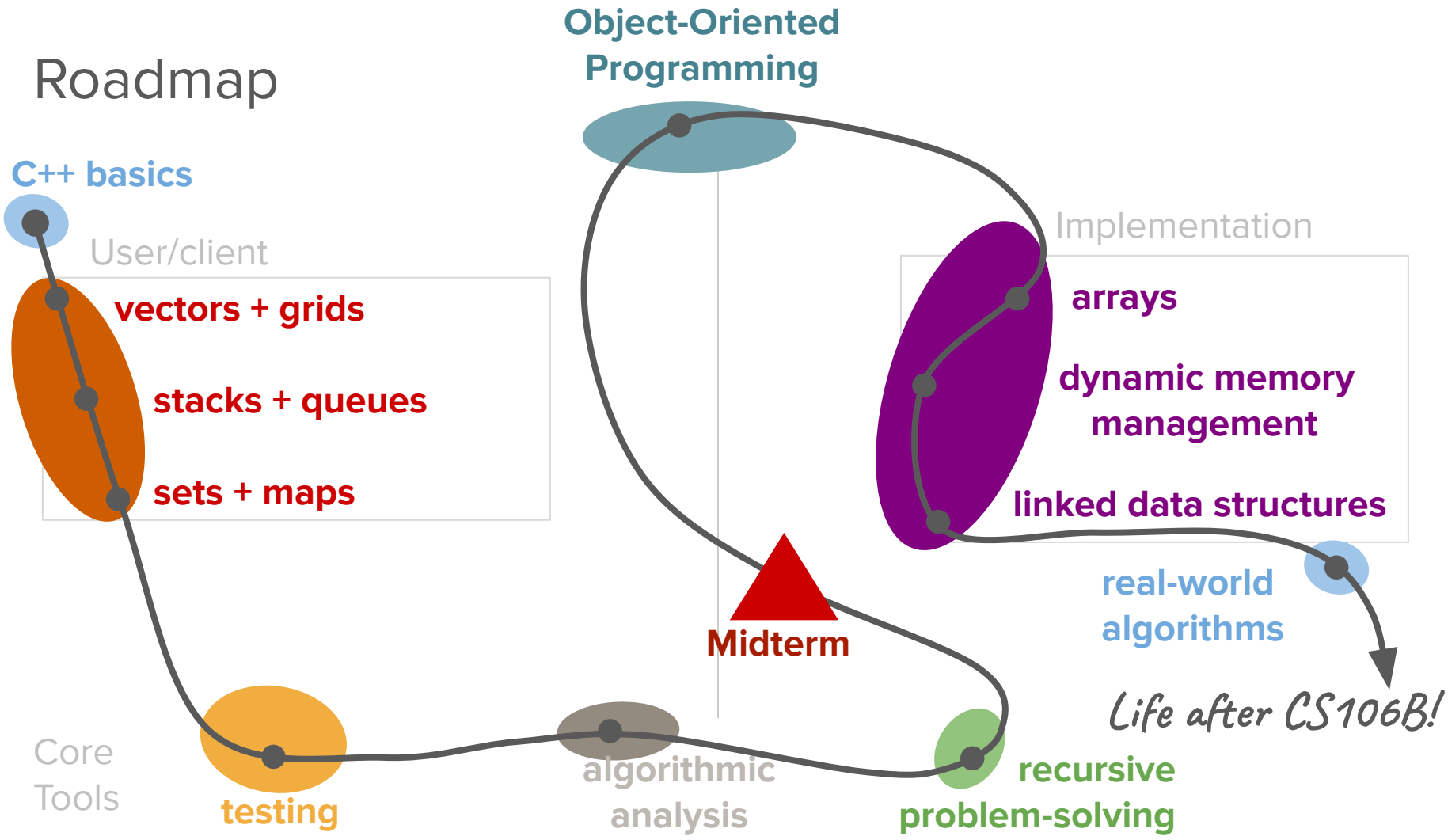
What's an example of "unordered data" that you've encountered in your life?

PollEv.com/cs106bpolls

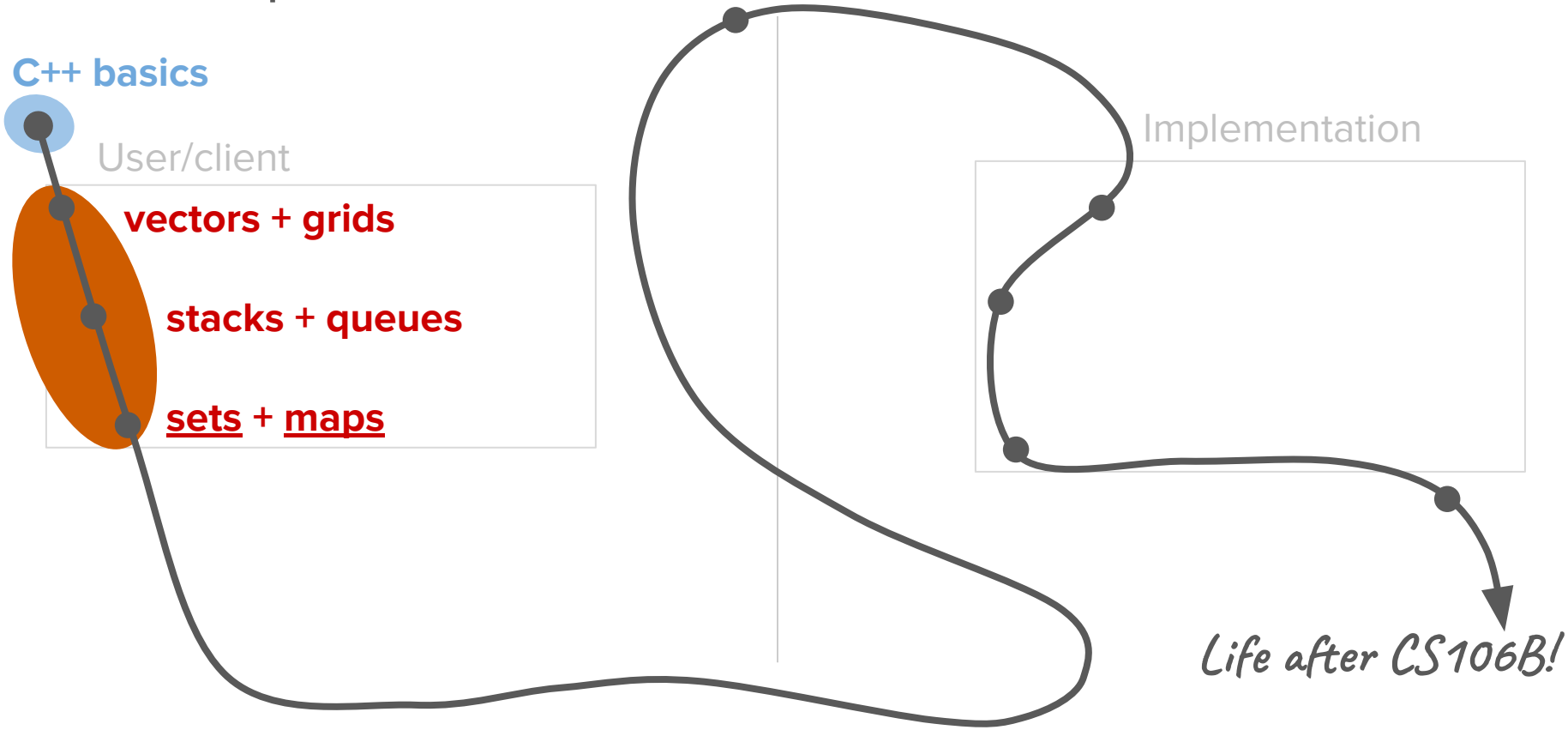


Unordered Data

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
Midterm
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
Life after CS106B!


The Container Store

Today's question

When is it appropriate to use different types of **unordered** data structures?

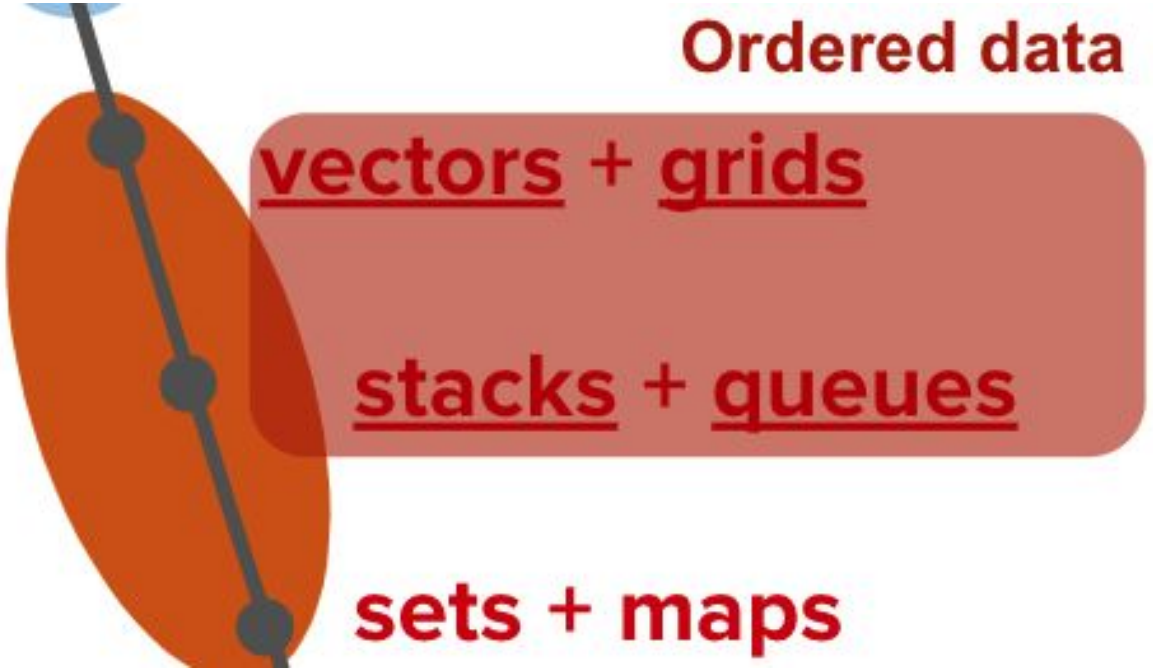
Today's topics

1. Review
2. Sets
3. Maps
4. (if time) Nested ADTs

Review

(grids and queues and stacks, oh my!)

The “container store”



What is a grid?

a0	a1	a2
b0	b1	b2
c0	c1	c2

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - Grid<type> gridName;
  - Grid<type> gridName(numRows, numCols);
  - Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};
- We could use a combination of Vectors to simulate a 2D matrix, but a Grid is easier!

Definition

struct

A way to bundle different types of information in C++ – like creating a custom data structure.

The **GridLocation** struct

```
struct GridLocation {
   int row;
   int col;
}
```

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

```
GridLocation origin = {0, 0};
```

```
GridLocation origin;
origin.row = 0;
origin.col = 0;
```

What is a queue?

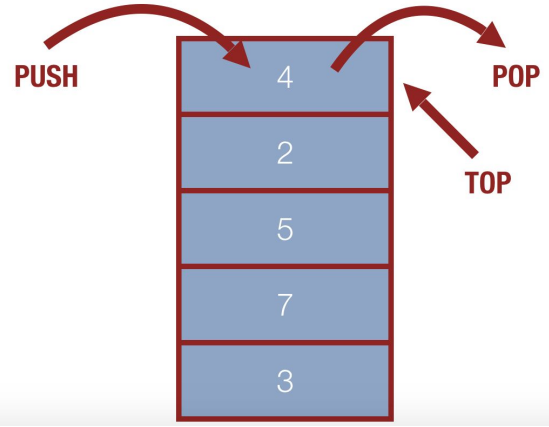
- Like a real queue/line!

- **F**irst person **I**n is the **F**irst person **O**ut (FIFO)
  - When you remove (dequeue) people from the queue, you remove them from the front of the line.

- Last person in is the last person served
  - When you insert (enqueue) people into a queue, you insert them at the back (the end of the line).

What is a stack?

- Modeled like an actual stack (of pancakes)
- Only the top element in a stack is accessible.
  - The **L**ast item **I**n is the **F**irst one **O**ut. (LIFO)
- The push, pop, and top operations are the only operations allowed by the stack ADT.



Ordered ADTs with accessible indices

Types:

- Vectors (1D)
- Grids (2D)

Traits:

- Easily able to search through all elements
- Can use the indices as a way of structuring the data

Ordered ADTs where you can't access elements by index

Types:

- Queues (FIFO)
- Stacks (LIFO)

Traits:

- Constrains the way you can insert and access data
- More efficient for solving specific LIFO/FIFO problems

What ADT should we use?

PollEv.com/cs106bpolls

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

Which ADT(s) would you use?

Vectors

Grids

Queues

Stacks

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- **The undo button in a text editor**
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

Which ADT(s) would you use?

Vectors

Grids

Queues

Stacks

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

(Queues)

Stacks

- The undo button in a text editor
- **Jobs submitted to a printer that can also be cancelled**
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

Which ADT(s) would you use?

Vectors

Grids

Queues

Stacks

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- **LaIR requests**
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

Which ADT(s) would you use?

Vectors

Grids

Queues

Stacks

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- **Your browsing history**
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

Which ADT(s) would you use?

Vectors

Grids

Queues

Stacks

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- **Google spreadsheets**
- Call centers (“your call will be handled by the next available agent”)

Which ADT(s) would you use?

Vectors

Grids

Queues

Stacks

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

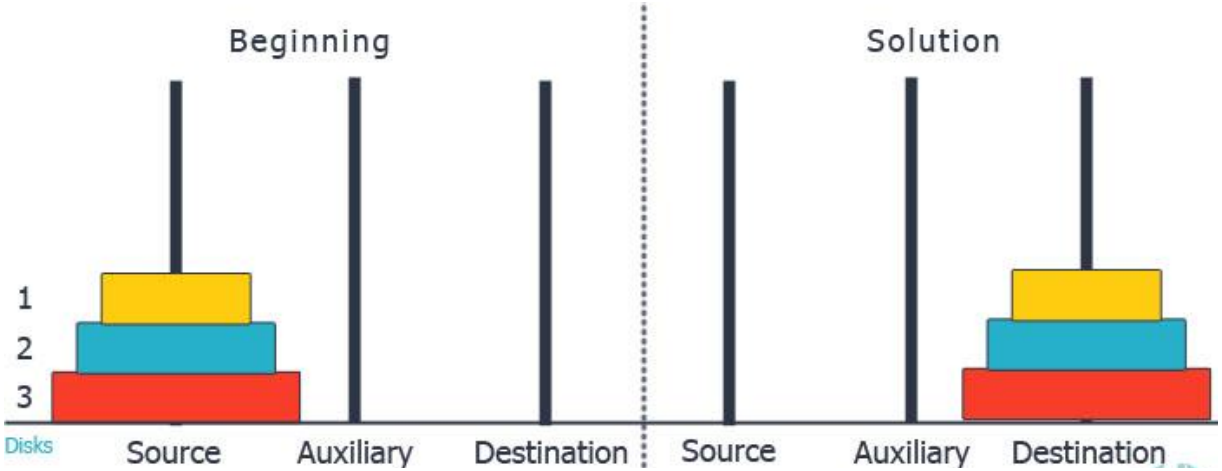
Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- **Call centers (“your call will be handled by the next available agent”)**

Using data structures:
Towers of Hanoi

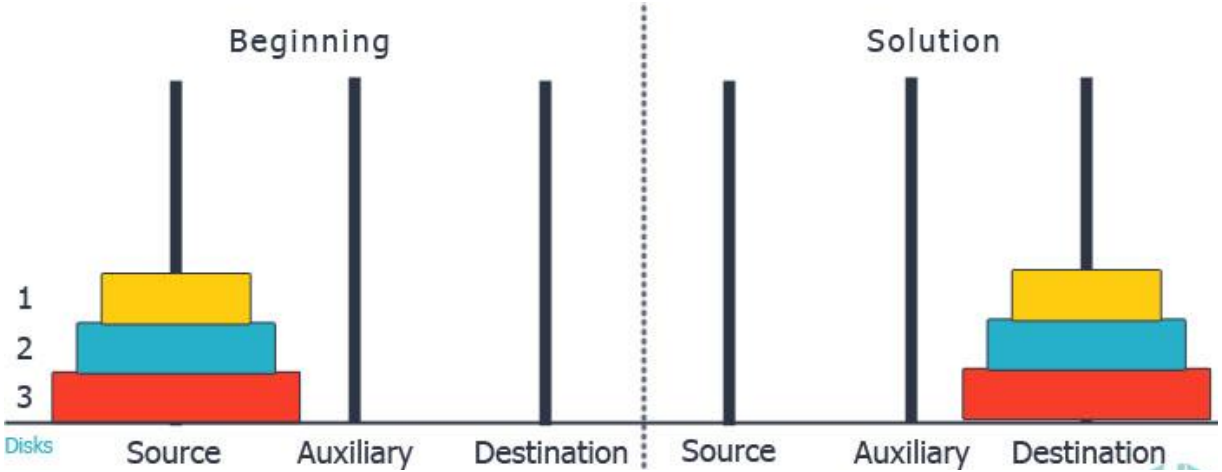
Towers of Hanoi

- Setup:
  - Three “towers”
  - N disks of decreasing sizes (below: N = 3)
- Goal: Move the disk stack from the first peg to the last peg



Towers of Hanoi

- Rules:
  - Can only move one disk at a time
  - You cannot place a larger disk on top of a smaller disk

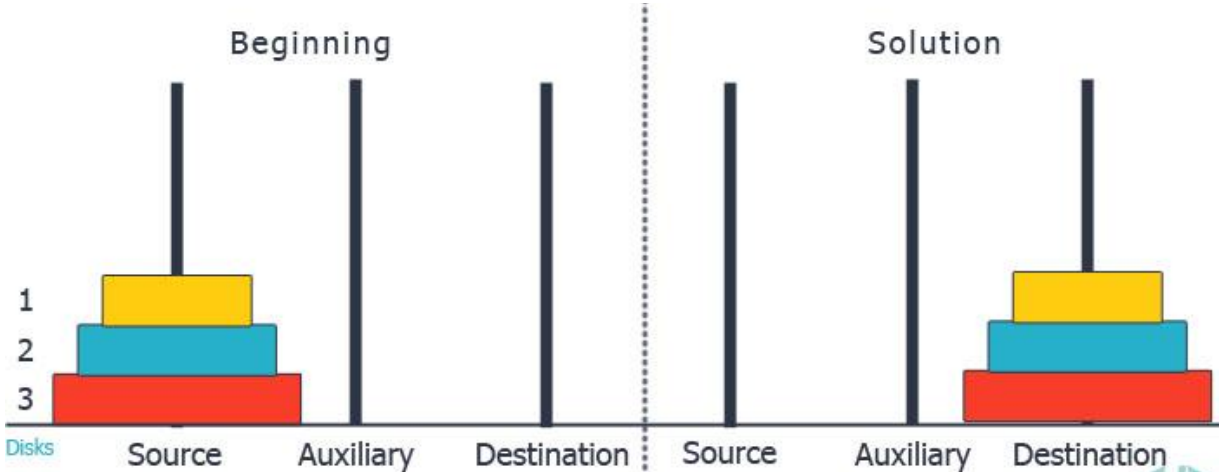


Attendance ticket:

https://tinyurl.com/cs106btowers

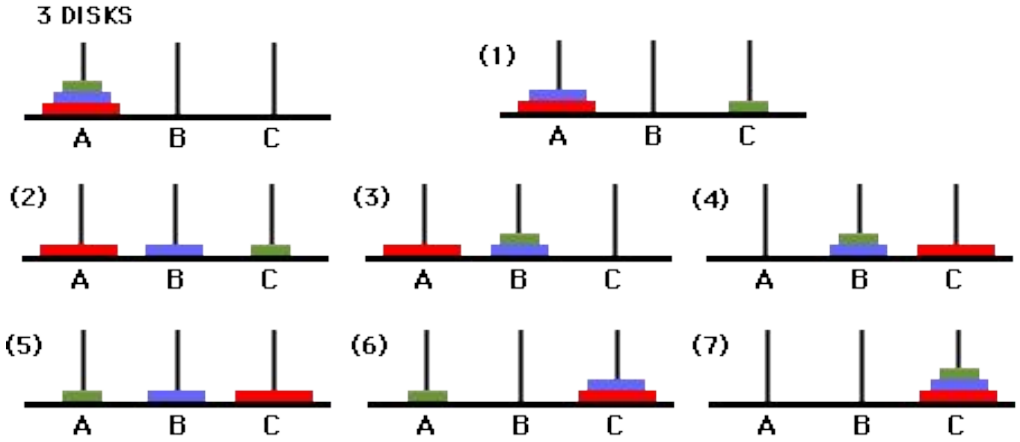
Please don’t send this link to students who are not here. It’s on your honor!

Towers of Hanoi




```
Stack <int> source = {3, 2, 1};
Stack <int> auxiliary;
Stack <int> destination;  // this should become {3, 2, 1}
```

Pseudocode



(1) Move disk 1 to destination
(2) Move disk 2 to auxiliary
(3) Move disk 1 to auxiliary
(4) Move disk 3 to destination
(5) Move disk 1 to source
(6) Move disk 2 to destination
(7) Move disk 1 to destination

Let’s look at the code solution for three disks!

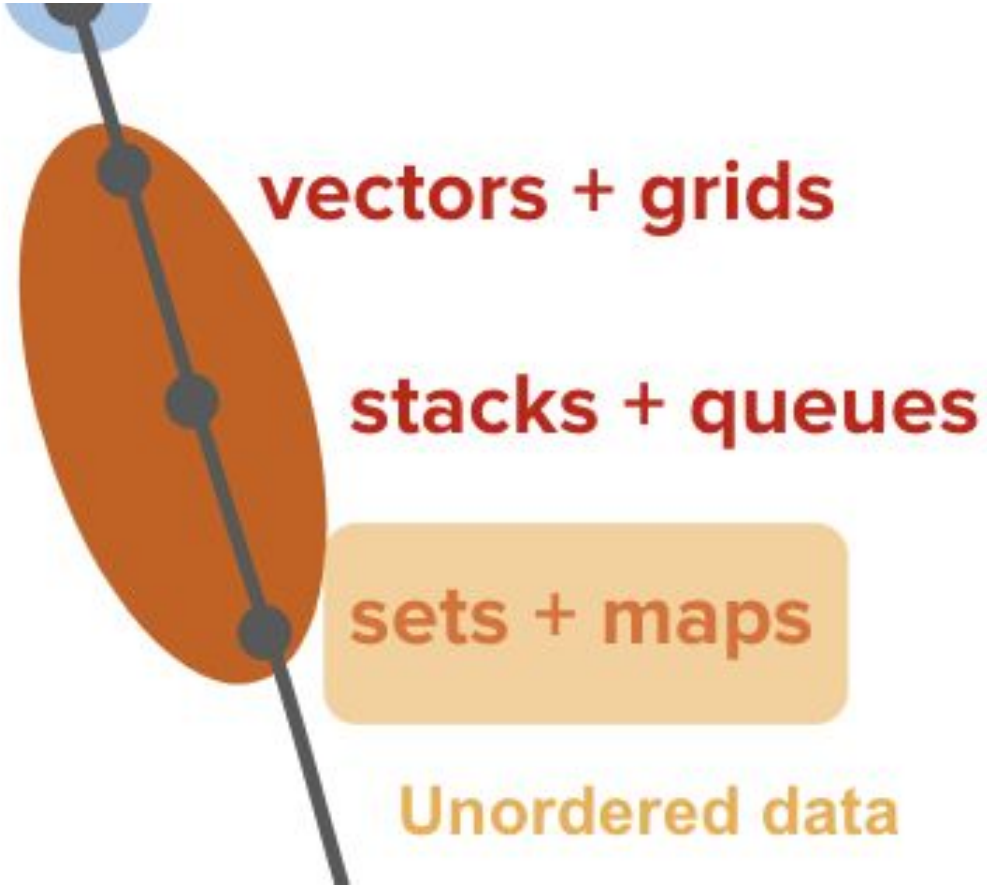
[Ed workspace]

Let’s look at the code solution for three disks!

Challenge for home: How would you generalize your solution to N disks instead of just 3?

Why do we use unordered ADTs?

The “container store”



Examples of unordered data

- Unique visitors to a website
- Shuffled playlist with no duplicate songs
- People and their passport numbers on a particular flight
- A recipe with ingredients and their quantities
- Products placed into categories in an online storefront

Examples of unordered data

When we say "unordered" vs. "ordered," we're referring specifically to numerical orderings.

- Unique visitors to a website
- Shuffled playlist with no duplicate songs
- People and their passport numbers on a particular flight
- A recipe with ingredients and their quantities
- Products placed into categories in an online storefront

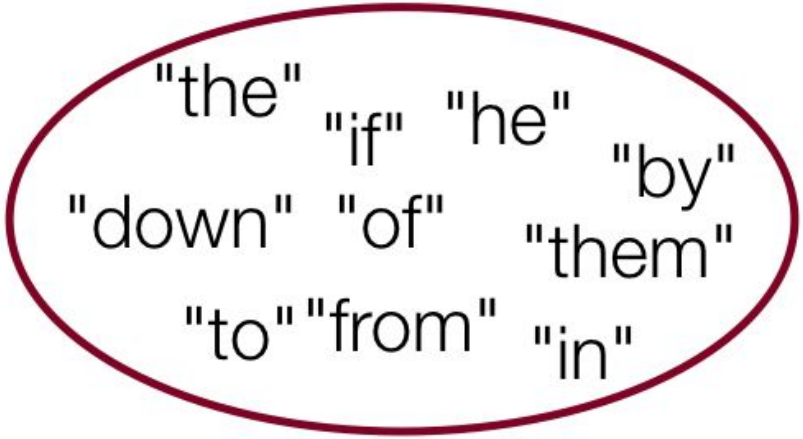
Examples of unordered data

- Unique visitors to a website
- Shuffled playlist with no duplicate songs
- People and their passport numbers on a particular flight
- A recipe with ingredients and their quantities
- Products placed into categories in an online storefront

Sometimes numerical indices/ordering is not the most efficient way to store information!

Sets

What is a set?



- A set is a collection of elements with no duplicates.

- Sets are faster than ordered data structures like vectors – since there are no duplicates, it's faster for them to find things.
  - (Later in the quarter we'll learn about the details of the underlying implementation that makes this abstraction efficient.)
  - We'll formally define "faster" tomorrow.

- Sets don't have indices!

Set methods

- Sets have (at least) the following operations (and they are *fast*):
  - **`add(value)`**: adds a value to a set, and ignores if the set already contains the value
  - **`contains(value)`**: returns **`true`** if the set contains the value, **`false`** otherwise.
  - **`remove(value)`**: removes the value from the set. Does nothing if the value is not in the set.
  - **`size()`**: returns the number of elements in the set
  - **`isEmpty()`**: returns **`true`** if the set is empty, **`false`** otherwise

- For the exhaustive list, check out the Stanford libraries documentation.

Set example

```
Set<string> friends;
friends.add("jenny");
friends.add("kylie");
friends.add("trip");
// can also use: Set<string> friends = {“jenny”, “kylie”, “trip”};
```

Set example

```
Set<string> friends;
friends.add("jenny");
friends.add("kylie");
friends.add("trip");
// can also use: Set<string> friends = {“jenny”, “kylie”, “trip”};

cout << boolalpha << friends.contains("voldemort") << noboolalpha
    << endl;
```

Set example

```
Set<string> friends;
friends.add("jenny");
friends.add("kylie");
friends.add("trip");
// can also use: Set<string> friends = {“jenny”, “kylie”, “trip”};

cout << boolalpha << friends.contains("voldemort") << noboolalpha
    << endl;

for(string person : friends) {
    cout << person << endl;
}
```

Set example

```
Set<string> friends;
friends.add("jenny");
friends.add("kylie");
friends.add("trip");
// can also use: Set<string> friends = {“jenny”, “kylie”, “trip”};

cout << boolalpha << friends.contains("voldemort") << noboolalpha
    << endl;

for(string person : friends) {
    cout << person << endl;
}
```

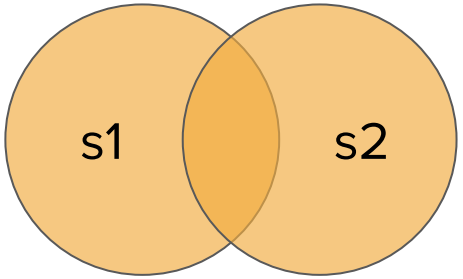
```
false
jenny
kylie
trip
```

Set operands

Sets can be compared, combined, etc.

- `s1 == s2`
  `true` if the sets contain exactly the same elements
- `s1 != s2`
  `true` if the sets don't contain the exact same elements

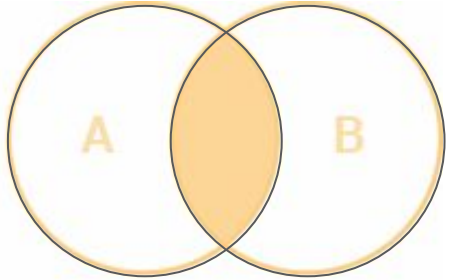
Set operands



Sets can be compared, combined, etc.

- **`s1 == s2`**
  `true` if the sets contain exactly the same elements
- **`s1 != s2`**
  `true` if the sets don't contain the exact same elements
- **`s1 + s2`**
  returns the *union* of **`s1`** and **`s2`** (i.e., all elements in both)

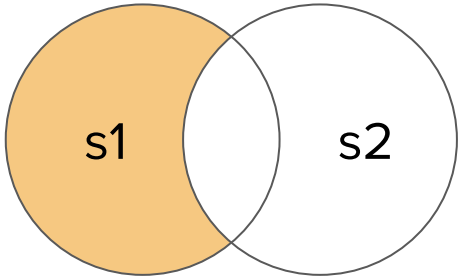
Set operands



Sets can be compared, combined, etc.

- **`s1 == s2`**
  `true` if the sets contain exactly the same elements
- **`s1 != s2`**
  `true` if the sets don't contain the exact same elements
- **`s1 + s2`**
  returns the *union* of **`s1`** and **`s2`** (i.e., all elements in both)
- **`s1 * s2`**
  returns the *intersection* of **`s1`** and **`s2`** (i.e., only the elements in both sets)

Set operands



Sets can be compared, combined, etc.

- **`s1 == s2`**
  `true` if the sets contain exactly the same elements
- **`s1 != s2`**
  `true` if the sets don't contain the exact same elements
- **`s1 + s2`**
  returns the *union* of **`s1`** and **`s2`** (i.e., all elements in both)
- **`s1 * s2`**
  returns the *intersection* of **`s1`** and **`s2`** (i.e., only the elements in both sets)
- **`s1 - s2`**
  returns the *difference* of **`s1`** and **`s2`** (the elements in **`s1`** but not in **`s2`**)

Common Set patterns and pitfalls

- Use for each loops to iterate over sets

```
for (type currElem : set) {
    // process elements one at a time
}
```

- You cannot use anything that attempts to index into the set (e.g. `for (int i = 0;..)` or `set[i]`)

Unique words program

[live coding]

Announcements

Announcements

- Assignment 1 is due Friday at 11:59pm PDT.

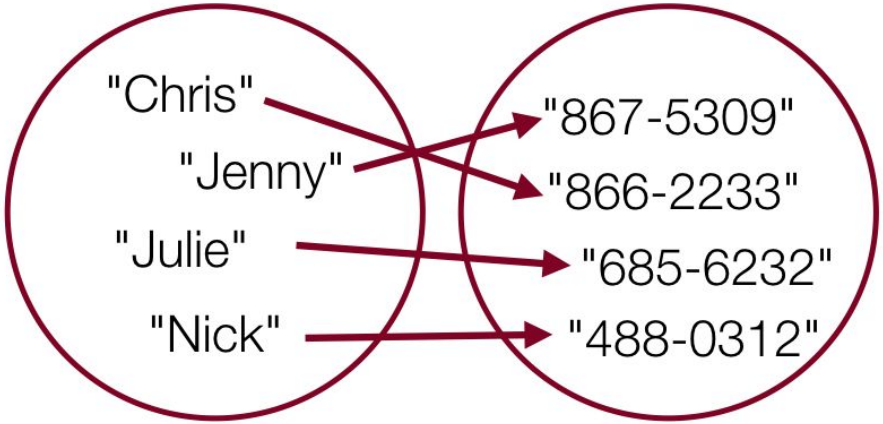
- Assignment 2 will be out by the end-of-the-day on Thursday and will be due next Thursday at 11:59pm PDT.
    - We're still finalizing YEAH timing and will announce that tomorrow.

- No lecture on Monday! But we will be releasing a shorter required video over the weekend to help you get started with nested ADTs for A2 instead.

- Jenny will be covering my group OH today since I'm not in person.

Maps

What is a map?



- A map is a collection of key/value pairs, and the key is used to quickly find the value.

- Other terms you may hear for a map are dictionary (Python) or associative array.

- A map is an alternative to an ordered data structure, where the “indices” no longer need to be integers.

Map methods

- The following functions are part of the **`Map`** class:
  - **`m.clear()`** : removes all key/value pairs from the map
  - **`m.containsKey(key)`** : returns **`true`** if the map contains a value for the given key
  - **`m[key]`**
    `m.get(key)` : returns the value associated with key in this map. If **`key`** is not found, returns the default value for ValueType.
  - **`m.isEmpty()`** : returns **`true`** if the map contains no key/value pairs (size 0)
  - **`m.keys()`** : returns a **`Vector`** copy of all keys in the map
  - **`m[key] = value`**
    `m.put(key, value)` : adds a mapping from the given key to the given value; if the key already exists, **replaces** its value with the given one
  - **`m.remove(key)`** : removes any existing mapping for the given **`key`** (ignored if the **`key`** doesn't exist in the map)
  - **`m.size()`** : returns the number of key/value pairs in the map
  - **`m.values()`** : returns a **`Vector`** copy of all the values in the map
- For the exhaustive list, check out the Stanford library documentation.

Map example

```
// maps from string keys to string values
Map<string, string> phoneBook;

//  key                 value
phoneBook["Jenny"] = "867-5309"; // or
phoneBook.put("Jenny", "867-5309");
```

Map example

```
// maps from string keys to string values
Map<string, string> phoneBook;

//  key                 value
phoneBook["Jenny"] = "867-5309"; // or
phoneBook.put("Jenny", "867-5309");
```

Inserting new values

Map example

```
// maps from string keys to string values
Map<string, string> phoneBook;

//  key                 value
phoneBook["Jenny"] = "867-5309"; // or
phoneBook.put("Jenny", "867-5309");

string jennyNumber = phoneBook["Jenny"]; // or
string jennyNumber = phoneBook.get("Jenny");
cout << jennyNumber << endl;
```

Accessing values

Map example

```
// maps from string keys to string values
Map<string, string> phoneBook;

//  key                  value
phoneBook["Jenny"] = "867-5309"; // or
phoneBook.put("Jenny", "867-5309");

string jennyNumber = phoneBook["Jenny"]; // or
string jennyNumber = phoneBook.get("Jenny");
cout << jennyNumber << endl;

// maps from string keys to Vector<double> values
Map<string, Vector<double>> accounts;
```

Common Map patterns and pitfalls

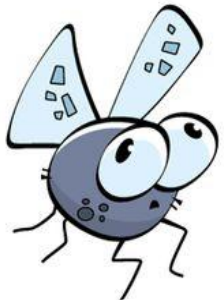
- Use for each loops to iterate over maps

```
for (type curKey : map) {
    // see map values using map[curKey]
}
```

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

```
for (type curKey : map) {
    // see map values using map[curKey]
}
```



But don't remove keys within the loop as you're iterating!

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

```
for (type curKey : map.keys()) {
    // see map values using map[curKey]
}
```

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

```
for (type curKey : map.keys()) {
    // see map values using map[curKey]
}
```

Okay to edit map within this loop because `.values()/.keys()` *makes a Vector copy of the values/keys.*

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

- Auto-insert: a map feature that can also cause bugs

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Common Map patterns and pitfalls

- Use for each loops to iterate over maps
- Auto-insert: a map feature that can also cause bugs

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

This auto-inserts the key `text` *into the map if it doesn't already exist!*

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

- Auto-insert: a map feature that can also cause bugs

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Note: auto-insertion only happens with the [] operator, not the .get() function

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

- Auto-insert: a map feature that can also cause bugs

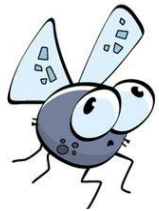
```
Map<string, int> playerPointsMap;
...
// get key to test if it's in the map
if (playerPointsMap[key] == 0) {
    cout << key << " already exists" << endl;
}
```

Common Map patterns and pitfalls

- Use for each loops to iterate over maps

- Auto-insert: a map feature that can also cause bugs

```
Map<string, int> playerPointsMap;
...
// get key to test if it's in the map
if (playerPointsMap[key] == 0) { // will always be true!
    cout << key << " already exists" << endl;
}
```



Common Map patterns and pitfalls

- Use for each loops to iterate over maps

- Auto-insert: a map feature that can also cause bugs

```
Map<string, int> playerPointsMap;
...
// use containsKey function, no auto-insert
if (playerPointsMap.containsKey(key)) { // correct way
    cout << key << " already exists" << endl;
}
```

Unique words program (extended)

[live coding]

ADT summary...

Ordered ADTs

Elements accessible by indices:

- Vectors (1D)
- Grids (2D)

Elements not accessible by indices:

- Queues (FIFO)
- Stacks (LIFO)

Unordered ADTs

- Sets (elements unique)
- Maps (keys unique)

Ordered ADTs

Elements accessible by indices:

- Vectors (1D)
- Grids (2D)

Elements not accessible by indices:

- Queues (FIFO)
- Stacks (LIFO)

Unordered ADTs

- Sets (elements unique)
- Maps (keys unique)

Useful when numerical ordering of data isn't optimal

Ordered ADTs

Elements accessible by indices:

- Vectors (1D)
- Grids (2D)

Elements not acces

- Queues (FIFO)
- Stacks (LIFO)

Unordered ADTs

- Sets (elements unique)
- Keys (keys unique)

Just like numerical ordering of

ADTs Takeaway: Matching structure with purpose results in better efficiency!

Nested Data Structures

Nested Data Structures

- Nesting data structures (using one ADTs as the data type inside of another ADT) is a great way of organizing data with complex structure.

Nested Data Structures

- Nesting data structures (using one ADTs as the data type inside of another ADT) is a great way of organizing data with complex structure.

- You will thoroughly explore nested data structures (specifically nested Sets and Maps) in Assignment 2!

Nested Data Structures

- Nesting data structures (using one ADTs as the data type inside of another ADT) is a great way of organizing data with complex structure.

- You will thoroughly explore nested data structures (specifically nested Sets and Maps) in Assignment 2!

- This weekend's recorded video will go into an in-depth example of using nested data structures.

Nested Data Structures Example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo.

Nested Data Structures Example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo.

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

Nested Data Structures Example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

- Data Structure Declaration
  - `Map<string, Vector<string>>`

Nested Data Structures Example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

- Data Structure Declaration
  - `Map<string, Vector<string>>`

Quick lookup by animal name

Nested Data Structures Example

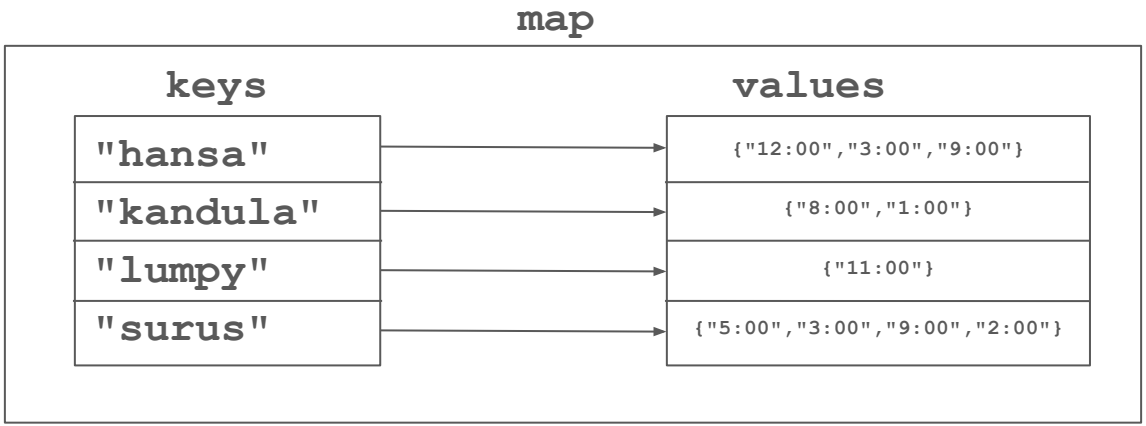
- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

- Data Structure Declaration
  - `Map<string, Vector<string>>`

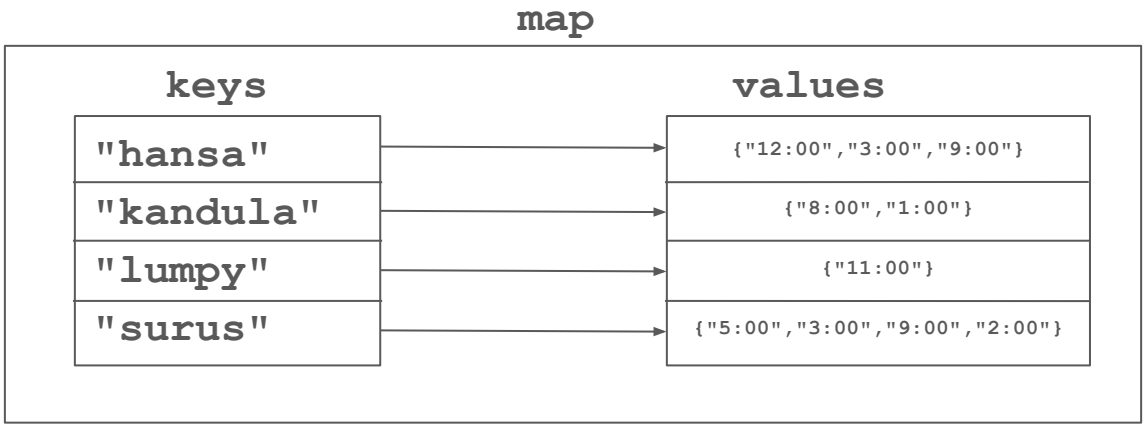
Store multiple, ordered feeding times per animal

Nested Data Structures Example



Wonderful diagram and animal naming borrowed from Sonja Johnson-Yu

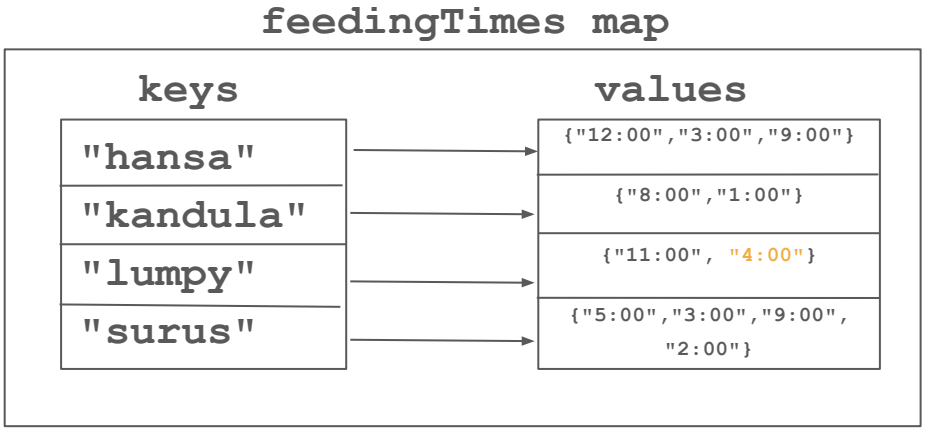
Nested Data Structures Example



How do we use modify the internal values of this map?

Nested Data Structures Example

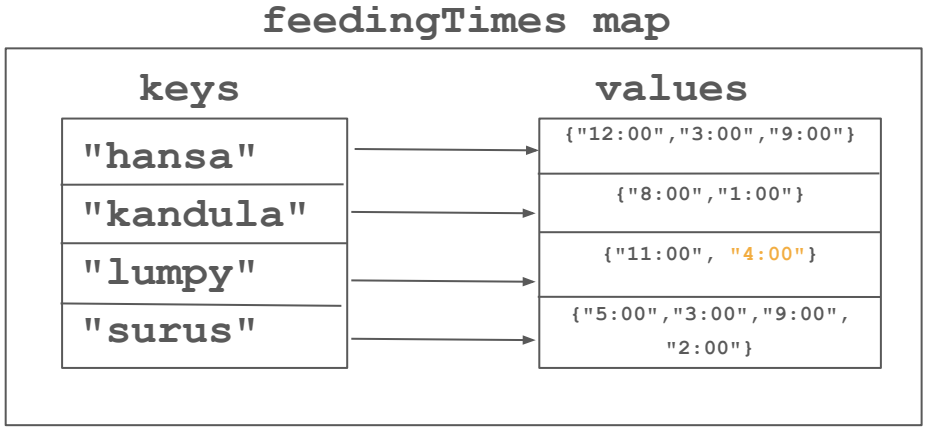
Goal: We want to add a second feeding time of 4:00 for "lumpy".



Nested Data Structures Example

Goal: We want to add a second feeding time of 4:00 for "lumpy".

Which of the following three snippets of code will correctly update the state of the map?



1.
```
feedingTimes["lumpy"].add("4:00");
```

2.
```
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
```

3.
```
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
feedingTimes["lumpy"] = times;
```

PollEv.com/cs106bpolls

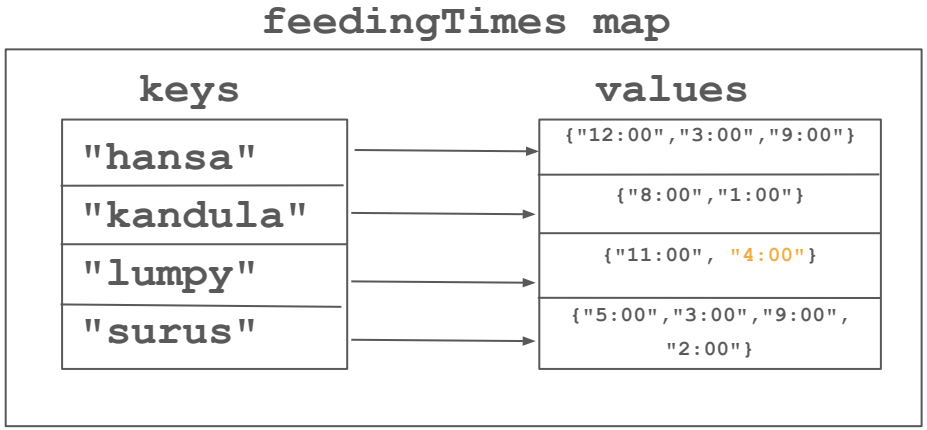
Nested Data Structures: Which snippet of code will correctly update the state of the map?

1.
2.
3.

Nested Data Structures Example

Goal: We want to add a second feeding time of 4:00 for "lumpy".

Which of the following three snippets of code will correctly update the state of the map?




```
1.  feedingTimes["lumpy"].add("4:00");
```

```
2.  Vector<string> times = feedingTimes["lumpy"];
    times.add("4:00");
```

```
3.  Vector<string> times = feedingTimes["lumpy"];
    times.add("4:00");
    feedingTimes["lumpy"] = times;
```

[] Operator and = Operator Nuances

- When you use the [] operator to access an element from a map, you get a reference to the map, which means that any changes you make to the reference will be persistent in the map.

```
feedingTimes["lumpy"].add("4:00");
```

[] Operator and = Operator Nuances

- When you use the [] operator to access an element from a map, you get a reference to the map, which means that any changes you make to the reference will be persistent in the map.

- However, when you use the = operator to assign the result of the [] operator to a variable, you get a copy of the internal data structure.

```
// makes and modifies a copy, not the actual map value:
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
```

[] Operator and = Operator Nuances

- When you use the [] operator to access an element from a map, you get a reference to the map, which means that any changes you make to the reference will be persistent in the map.

- However, when you use the = operator to assign the result of the [] operator to a variable, you get a copy of the internal data structure.

- If you choose to store the internal data structure in an intermediate variable, you must do an explicit reassignment to get your changes to persist.

```
// would store the modified `times` copy in the map
feedingTimes["lumpy"] = times;
```

Nested ADTs Summary

- Powerful
  - Can express highly structured and complex data
  - Used in many real-world systems

- Tricky
  - With increased complexity comes increased cognitive load in differentiating the information stored at each level of the nesting.
  - Specifically in C++, working with nested data structures can be tricky due the use of references and copies. Follow the correct paradigms to stay on track!

One final note... **`const`** reference

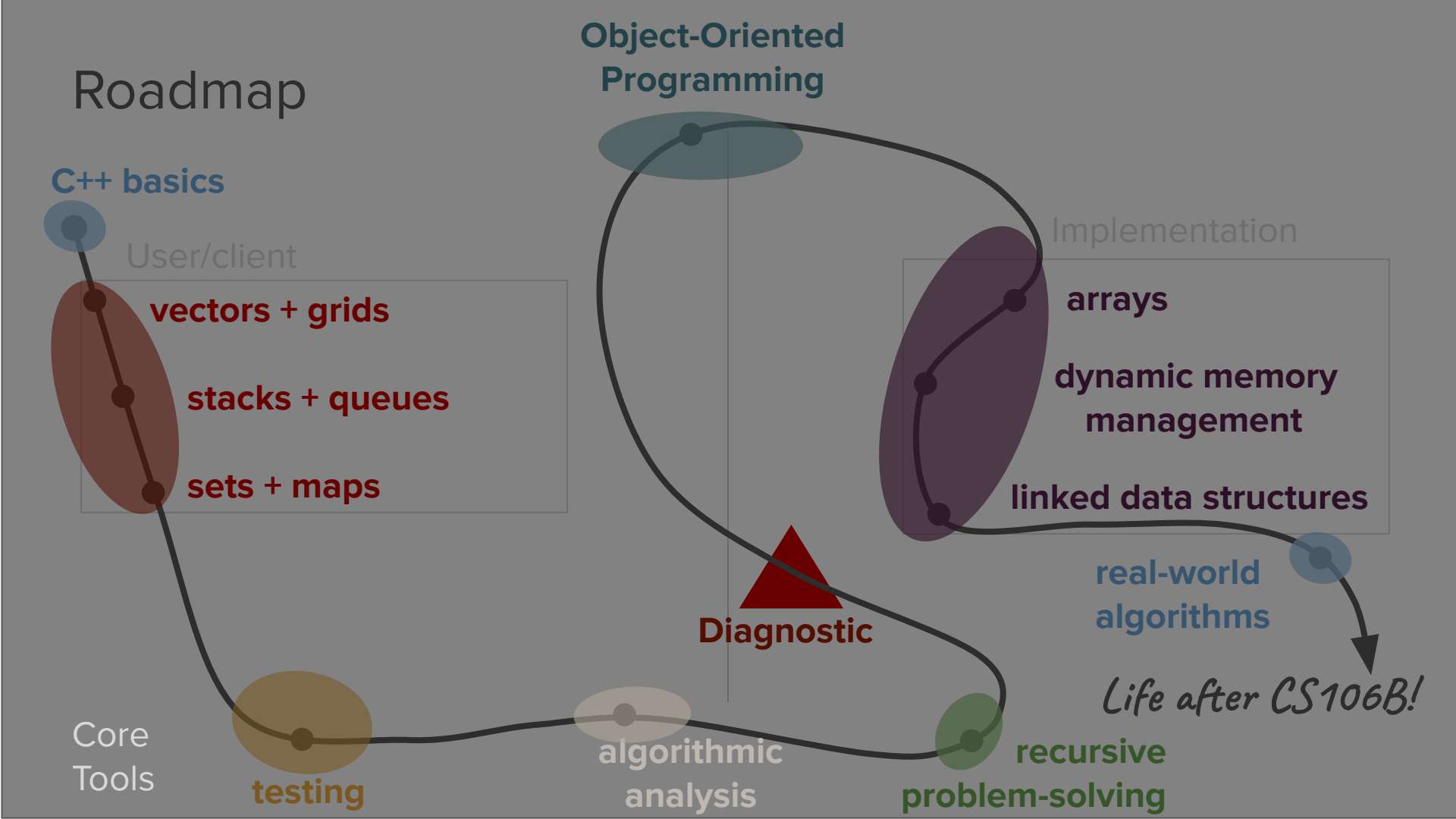
- Passing a large object (e.g. a million-element Vector) by value makes a copy, which is inefficient in time and space.
- Passing parameters by reference avoids making a copy, but creates risk that a function may modify a piece of data that you don't want it to edit.

- Solution: **`const`** reference!
  - The "by reference" part avoids a copy.
  - The "const" (constant) part means that the function can't change that argument.

```
void proofreadLongEssay(const string& essay) {
    /* can read, but not change, the essay. */
}
```


The Container Store

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
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Big O and Algorithmic Analysis