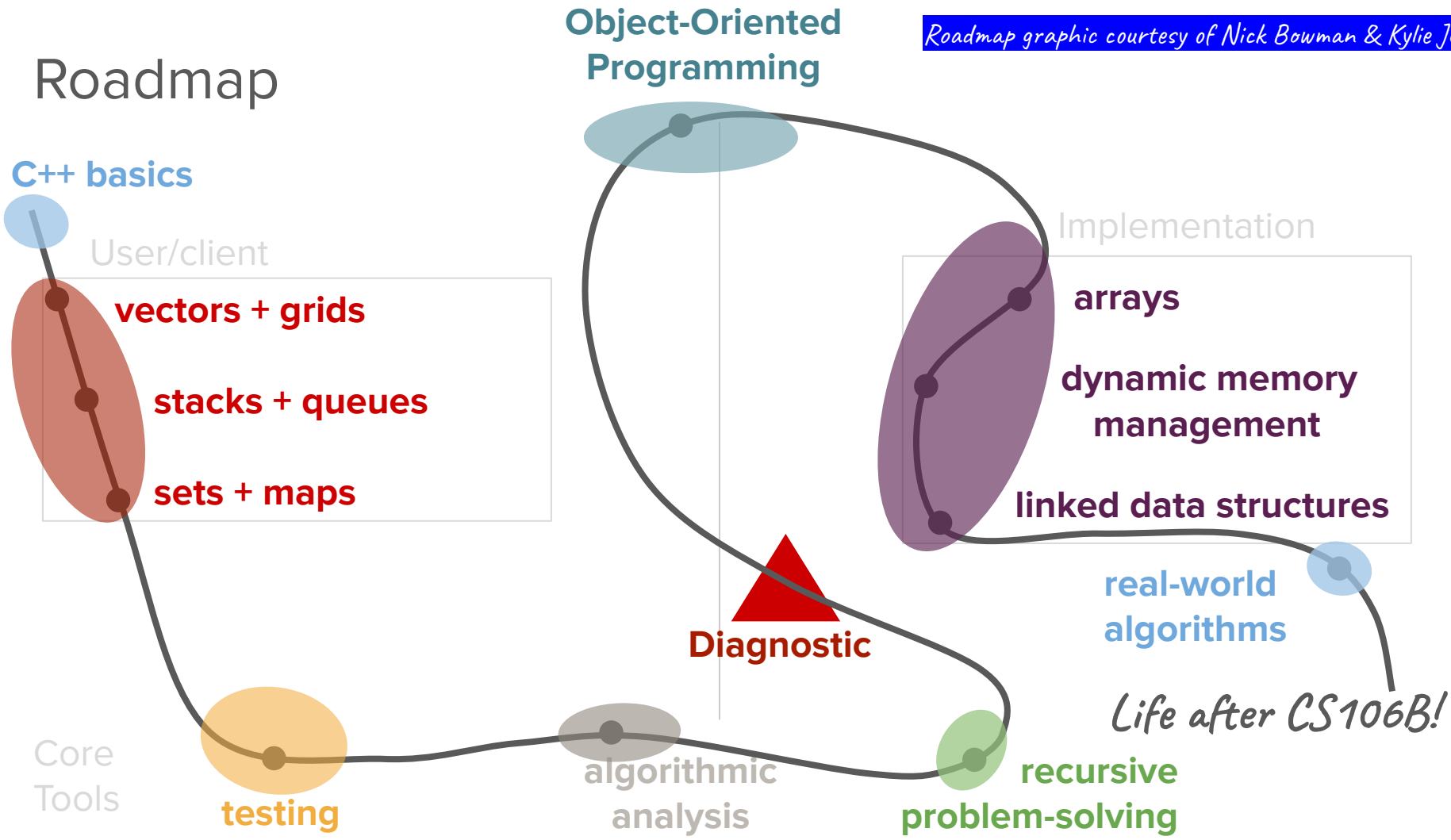
Unordered Data Structures: Sets and Maps

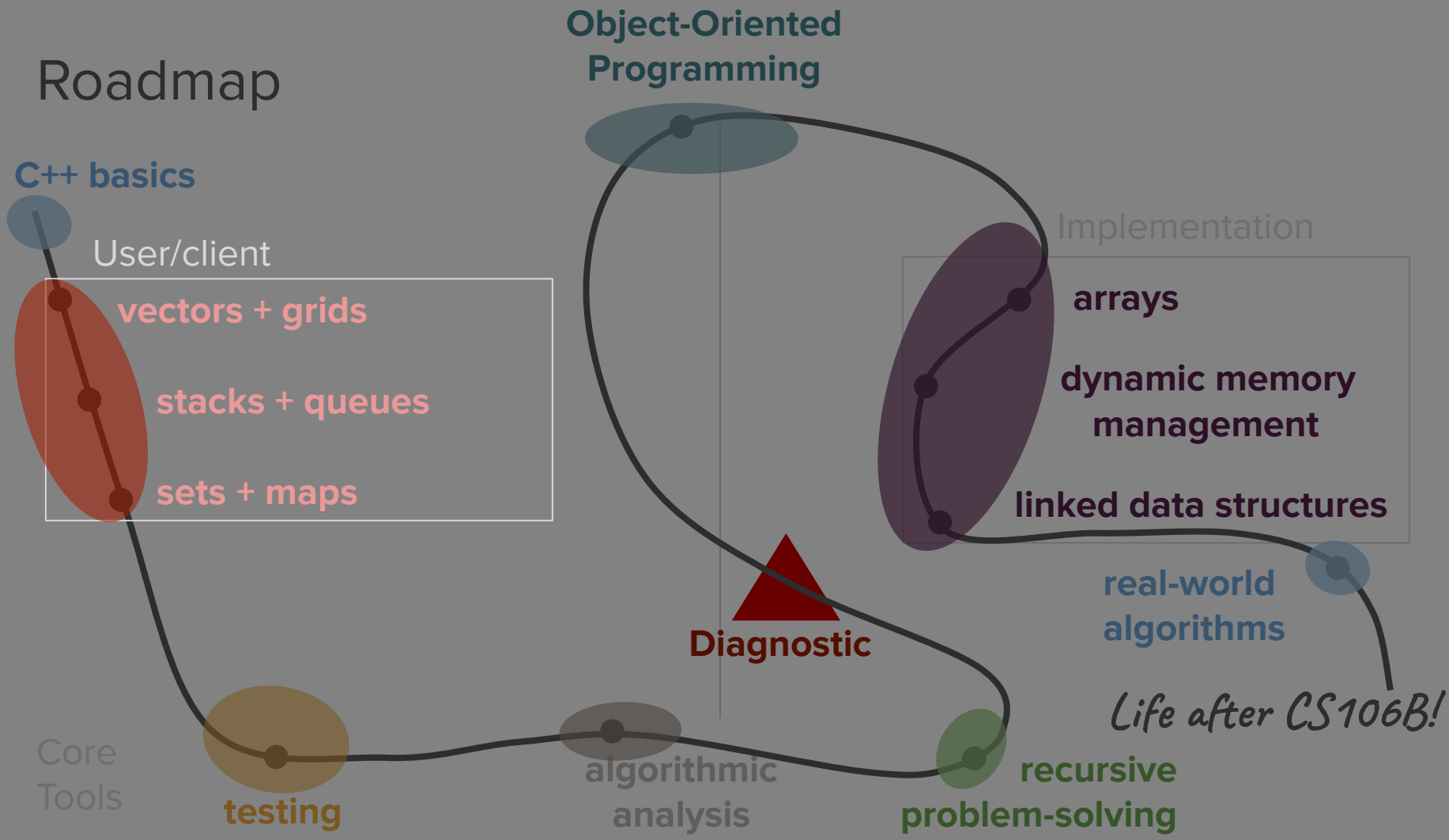
What's an example of "unordered data" that you've encountered in your life?

(put your answers in the chat)

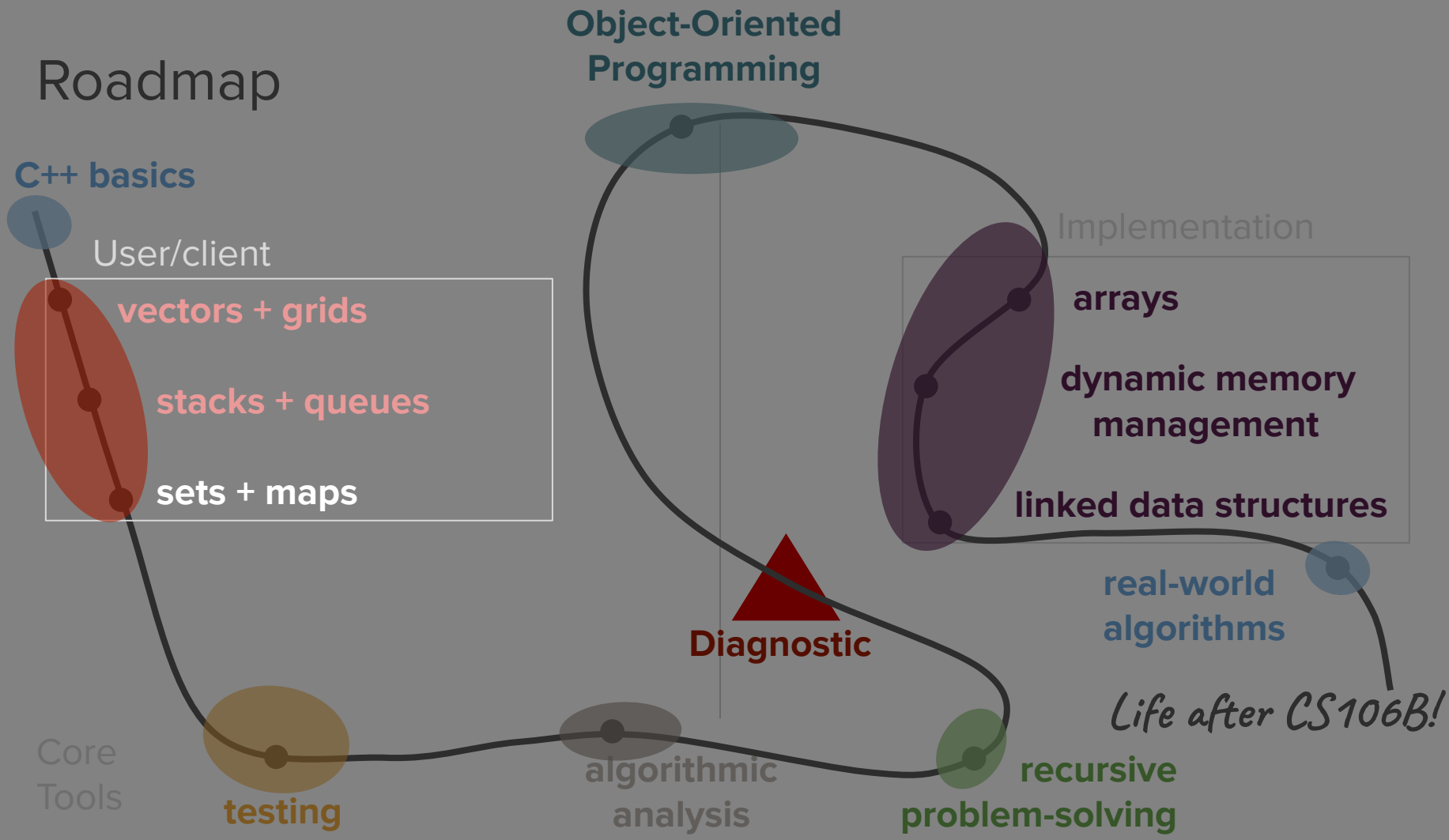
Roadmap


Object-Oriented Programming
Roadmap graphic courtesy of Nick Bowman & Kylie Jue
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

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

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

Today's topics

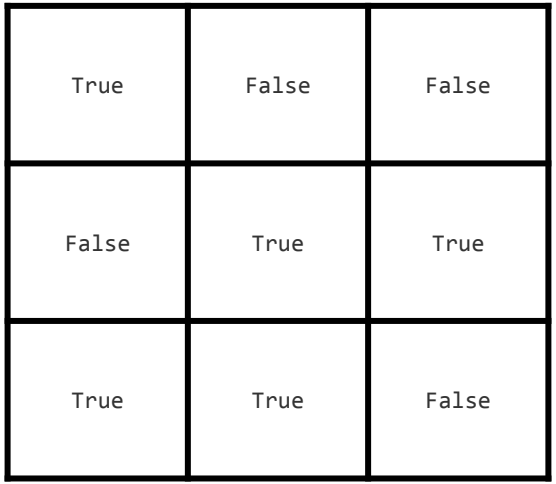
1. Review
2. Sets
3. Maps

Review

(grids and queues and stacks, oh my!)

What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, and more
- Three ways to declare a grid
  - Grid<type> gridName;
  - Grid<type> gridName(numRows, numCols);
  - Grid<type> gridName = {{r0c0, r0c1, r0c2},
    {r1c0, r1c1, r1c2},...};



Definition

struct

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

The **GridLocation** struct

- 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;
```

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

What is a queue?

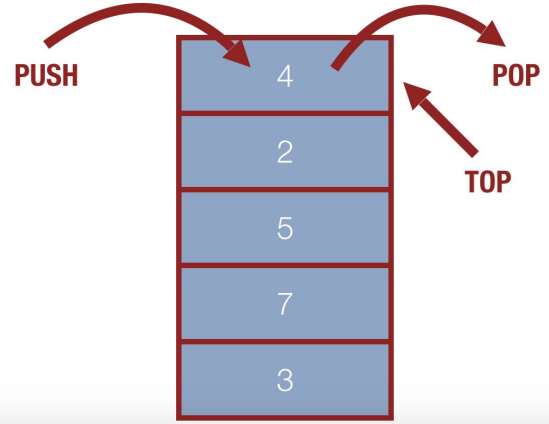
- Like a real queue/line!

- **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)

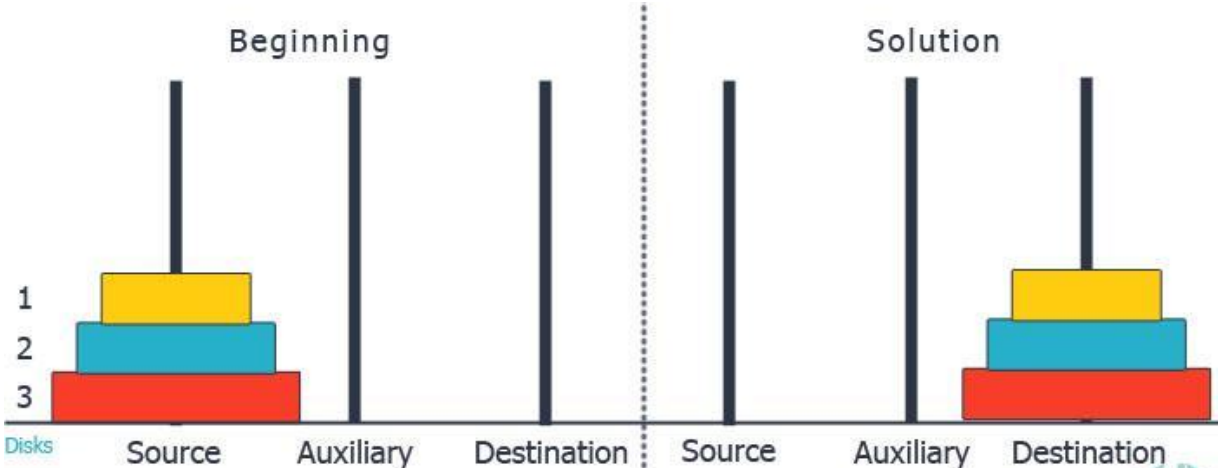
Traits:

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

Activity:
`towersOfHanoi()`

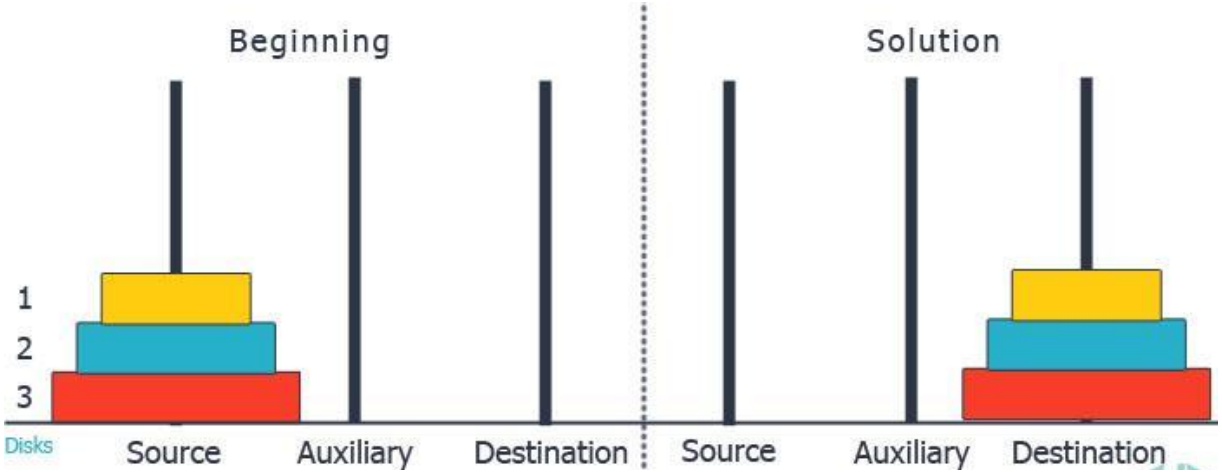
Towers of Hanoi

- 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

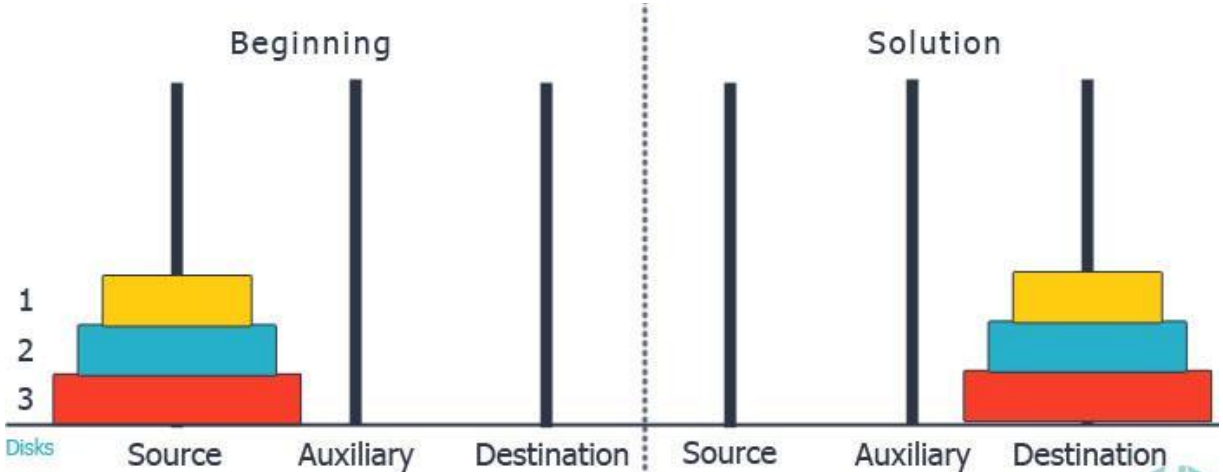
- Rules:
  - Can only move one disk at a time
  - You cannot place a larger disk on top of a smaller disk



Discuss: How would you solve this problem?

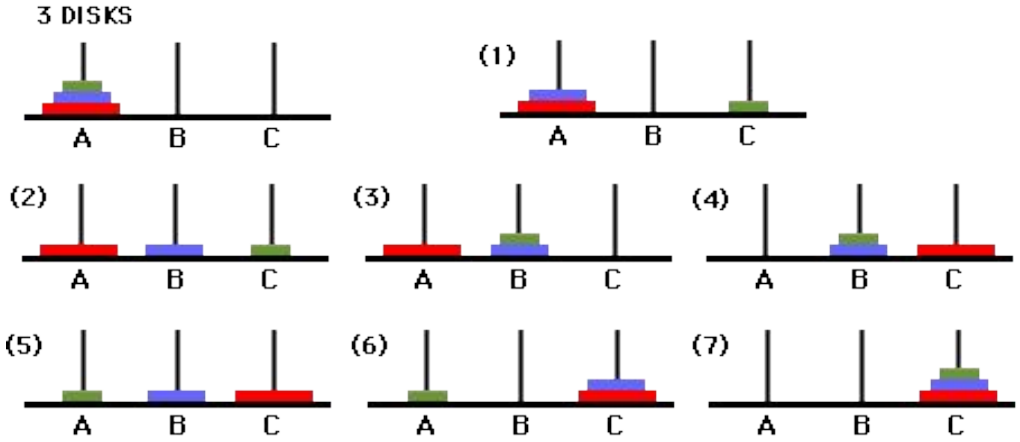
[breakout rooms]

Towers of Hanoi



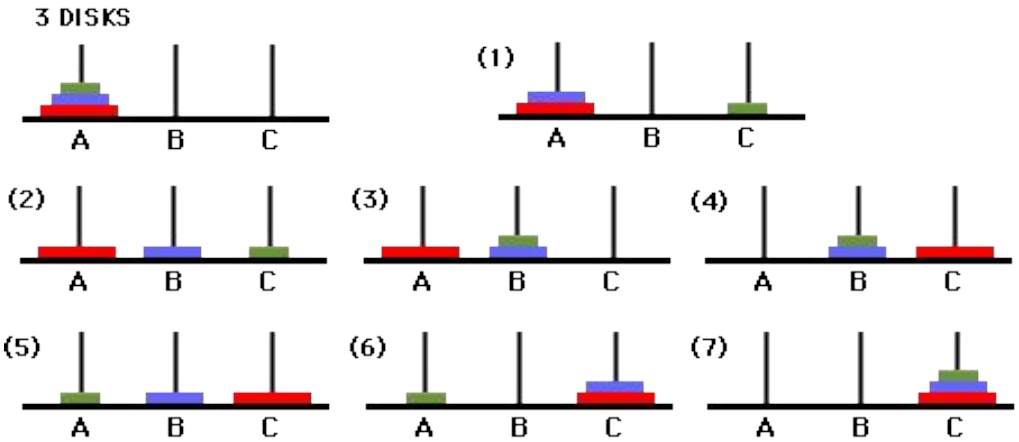

```
Stack <int> source = {3, 2, 1};
Stack <int> auxiliary;
Stack <int> destination;  // this should become {3, 2, 1}
```

Pseudocode

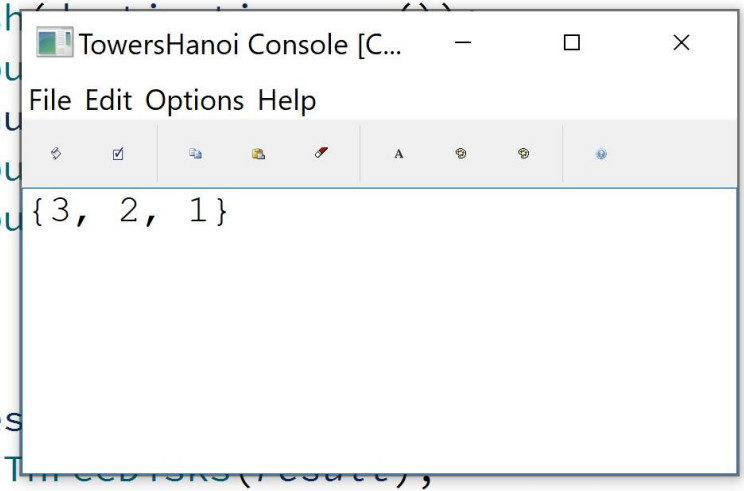


(1) Move disk 1 to destination
(2) Move disk 2 to auxiliary
(3) Move disk 1 to auxiliary
(4) Move disk 3 to destination
(5) Move disk 1 to source
(6) Move disk 2 to destination
(7) Move disk 1 to destination

```cpp
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

void towersOfHanoiThreeDisks(Stack<int>& destination) {
    Stack<int> source = {3, 2, 1};
    Stack<int> auxiliary;
    destination.push(source.pop());
    auxiliary.push(source.pop());
    auxiliary.push(destination.pop());
    destination.push(source.pop());
    source.push(auxiliary.pop());
    destination.push(auxiliary.pop());
    destination.push(source.pop());
}

int main() {
    Stack<int> result; // The final peg
    towersOfHanoiThreeDisks(result);
    cout << result << endl; // Show the final peg
    return 0;
}
```



```
void towersOfHanoiThreeDisks(Stack<int>& destination) {
    Stack<int> source = {3, 2, 1};
    Stack<int> auxiliary;
    destination.push(source.pop());
    auxiliary.push(source.pop());
    auxiliary.push
    destination.pu
    source.push(au
    destination.pu
    destination.pu
}

int main() {
    Stack<int> res
    towersOfHanoiT
    cout << result << endl; // Show the final peg
    return 0;
}
```



Code the solution for more than 3 disks!

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

Today's question

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

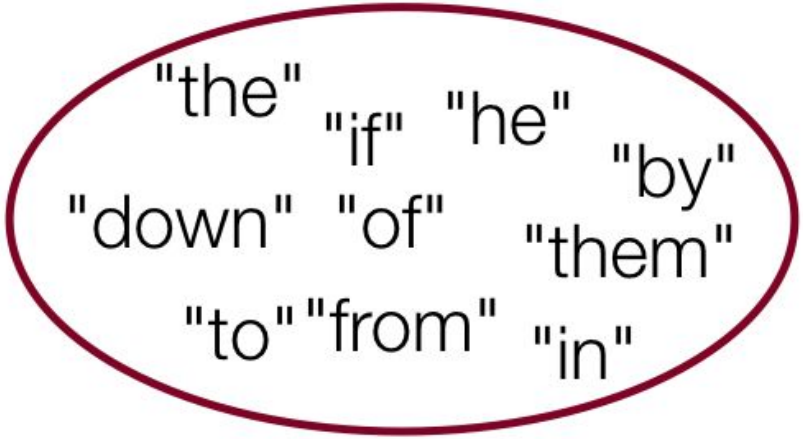
Why do we use unordered ADTs?

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

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" on Thursday.

- 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(“Harry");
friends.add(“Hermione");
friends.add(“Ron");
// can also use: Set<string> friends = {“Harry”, “Hermione”,
“Ron”};

cout << friends.contains(“Voldemort") << endl;

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

Set example

```
Set<string> friends;
friends.add(“Harry");
friends.add(“Hermione");
friends.add(“Ron");
// can also use: Set<string> friends = {“Harry”, “Hermione”,
“Ron”};

cout << boolalpha << friends.conta                    ort") << noboolalpha
    << endl;

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

```
false
Harry
Hermione
Ron
```

Set operands

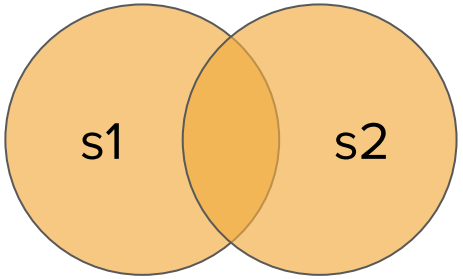
Sets can be compared, combined, etc.

- `s1 == s2`

  `true` if the sets contain exactly the same elements
- `s1 != s2`

  `true` if the sets don't contain the exact same elements

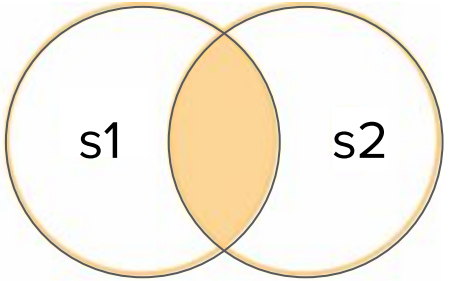
Set operands



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)

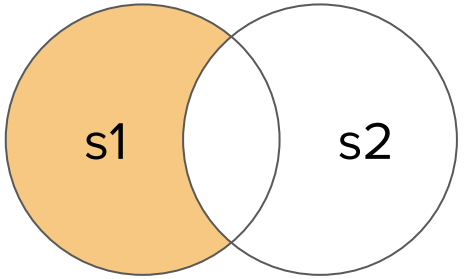
Set operands



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

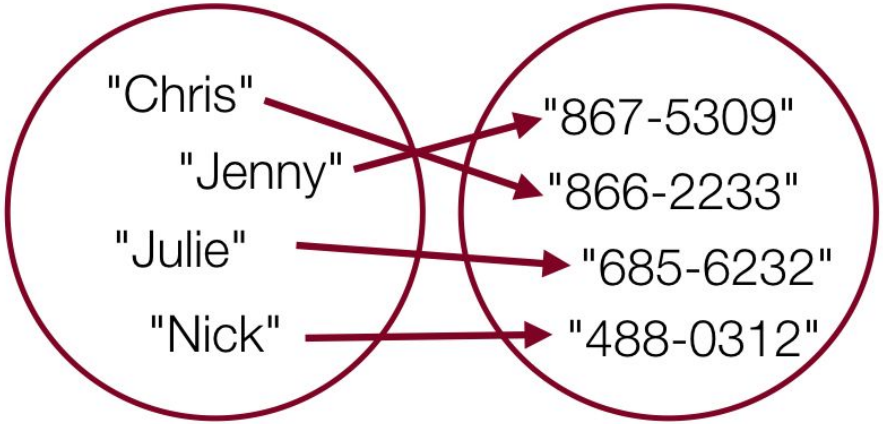
Announcements

- Assignment 1 is due tonight at 11:59pm PT. Recall that we have a 48-hour grace period for submissions, so you'll have until 7/1 at 11:59PM PT

- Assignment 2 will be out by the end-of-the-day tomorrow.
  - YEAH hours: Hosted by Jin-Hee and Grant, **Thursday** 7/1 at 7PM PDT

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");

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;
```

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;
```

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;

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

Accessing values

Map example

```
// declare and initialize map with all keys and values

Map<string, string> phoneBook =  {   {"Chris", "867-5309"},
                                {"Jenny", "867-5309"},
                                    {"Julie", "867-5309"},
                                {"Nick", "867-5309"} };
```

Common Map patterns and pitfalls

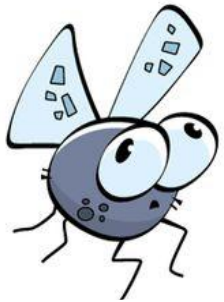
- Use for each loops to iterate over maps

```
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

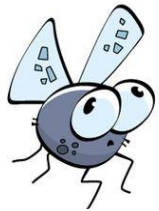
```
Map<string, int> freqMap;
...
// get key to test if it's in the map
if (freqMap[key] == 0) {
    cout << key << " is in the map" << endl;
}
```

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> freqMap;
...
// get key to test if it's in the map
if (freqMap[key] == 0) { // this will always be true!
    cout << key << " is in the map" << 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> 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() method

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;
...
// use containsKey method, no auto-insert
if (freqMap.containsKey(key)) { // correct way to do it
    cout << key << " is in the map" << 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)
- Keys (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)
- Keys (keys unique)

Useful when numerical ordering of data isn't optimal

Ordered ADTs

Elements accessible by indices:

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

Elements not accessible by indices:

- Queues (FIFO)
- Stacks (LIFO)

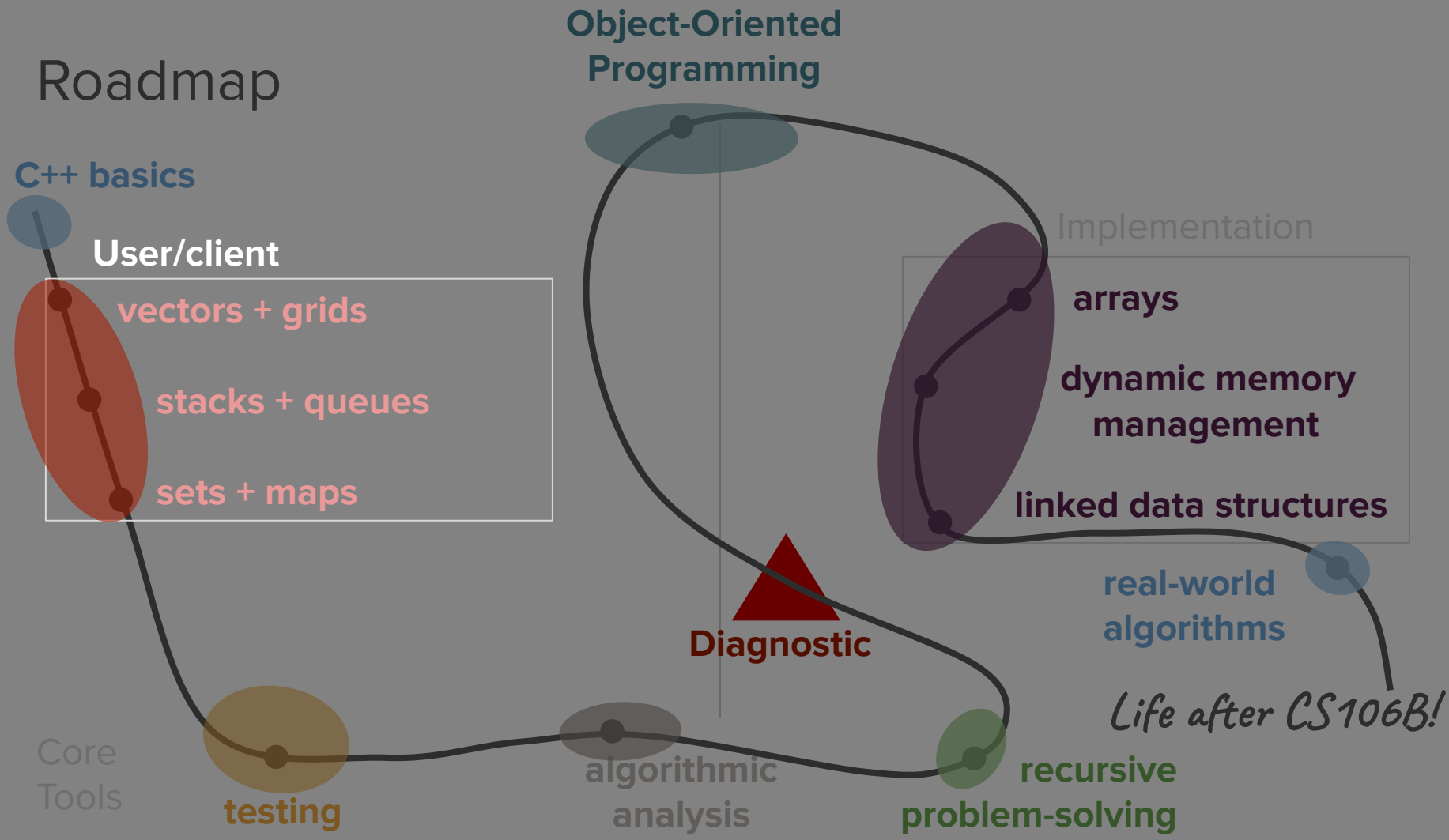
Unordered ADTs

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

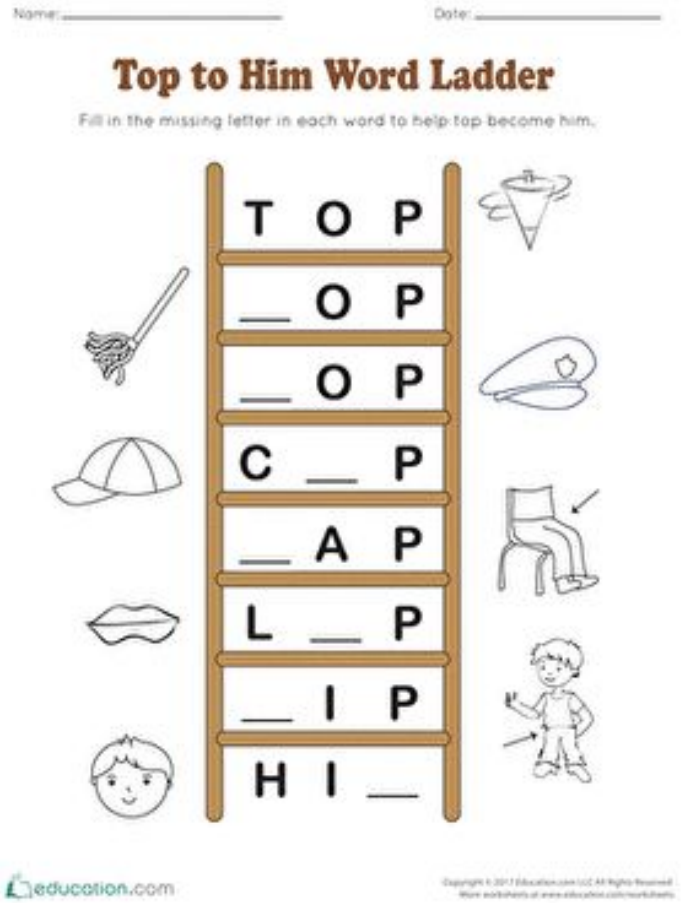
ADTs Takeaway: Matching structure with purpose results in better efficiency!

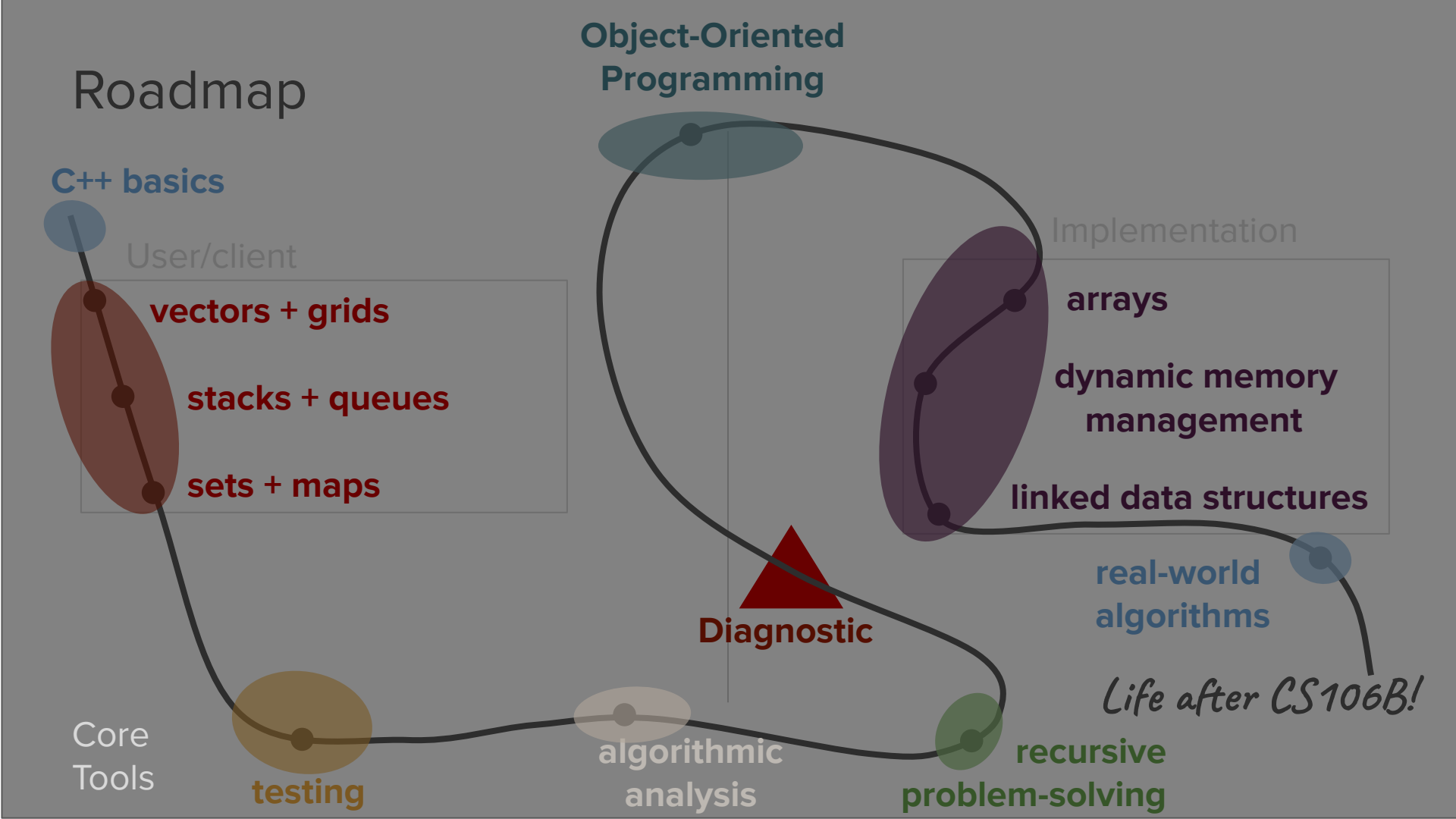
What’s next?

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

Tomorrow:


Name: _____ Date: _____
Top to Him Word Ladder
Fill in the missing letter in each word to help top become him.
T O P
_ O P
_ O P
C _ P
_ A P
L _ P
_ I P
H I _
education.com


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