Sets and Maps

Amrita Kaur

July 5, 2023

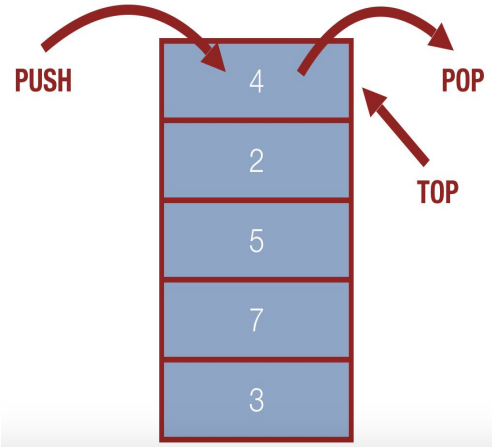
Contributions made from previous CS106B Instructors

Announcements and Reminders

- Assignment 1 due Friday at 11:59pm
- Midterm conflicts or OAE accommodations emailed to us by 7/10
- Participation grades for Week 1 Section have been posted
- Week 2 Section starts today!

Review

Stacks



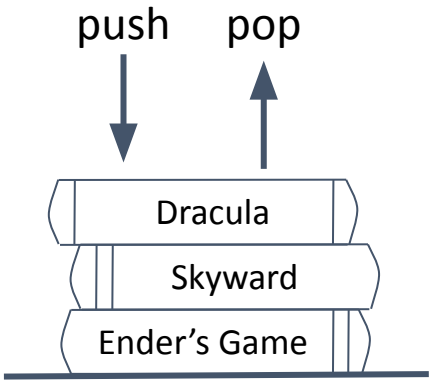
- Ordered
- Last In, First Out (LIFO)
- Only the top element of the stack is accessible
- Important operations:
  - **`stack.push(value)`**: Add an element onto the top of the stack
  - **`stack.pop()`**: Remove an element from the top of the stack and return it
  - **`stack.peek()`**: Look at the element from the top of the stack, but don't remove it

Queues

- Ordered
- First In, First Out (FIFO)
- Add to back, remove from front
- Important operations:
  - **`queue.enqueue(value)`**: Add an element to the back of the queue
  - **`queue.dequeue()`**: Remove an element from the front of the queue and return it
  - **`queue.peek()`**: Look at the element from the front of the queue, but don't remove it

Stack

Last **I**n, **F**irst **O**ut (LIFO)



Queue

First **I**n, **F**irst **O**ut (FIFO)

Tradeoffs with Stacks and Queues

What are some downsides?

- No random access of elements
- Difficult to traverse - requires removal of elements
- No easy way to search

What are some benefits?

- Useful for many real world problems
- Easy to build such that access is guaranteed to be fast

Reversing Words in a Sentence

Let’s build a program from scratch that reverses the words in a sentence.

Example input: “the cat in the hat”

Example output: “hat the in cat the”

Let’s make a plan! Some things to think about:

- Which ADT should we use?
- What steps will we need to do?

Reversing Words in a Sentence

ADT: Stack

Steps:

1. Read a word from the string (done reading when we reach a space)
2. Push word onto stack
3. Repeat steps 1 and 2 until we've pushed all the words to the stack
4. Pop the words from the stack and print with spaces

Recap of ADTs So Far

ADTs with indices

Types

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

Properties

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

ADTs without indices

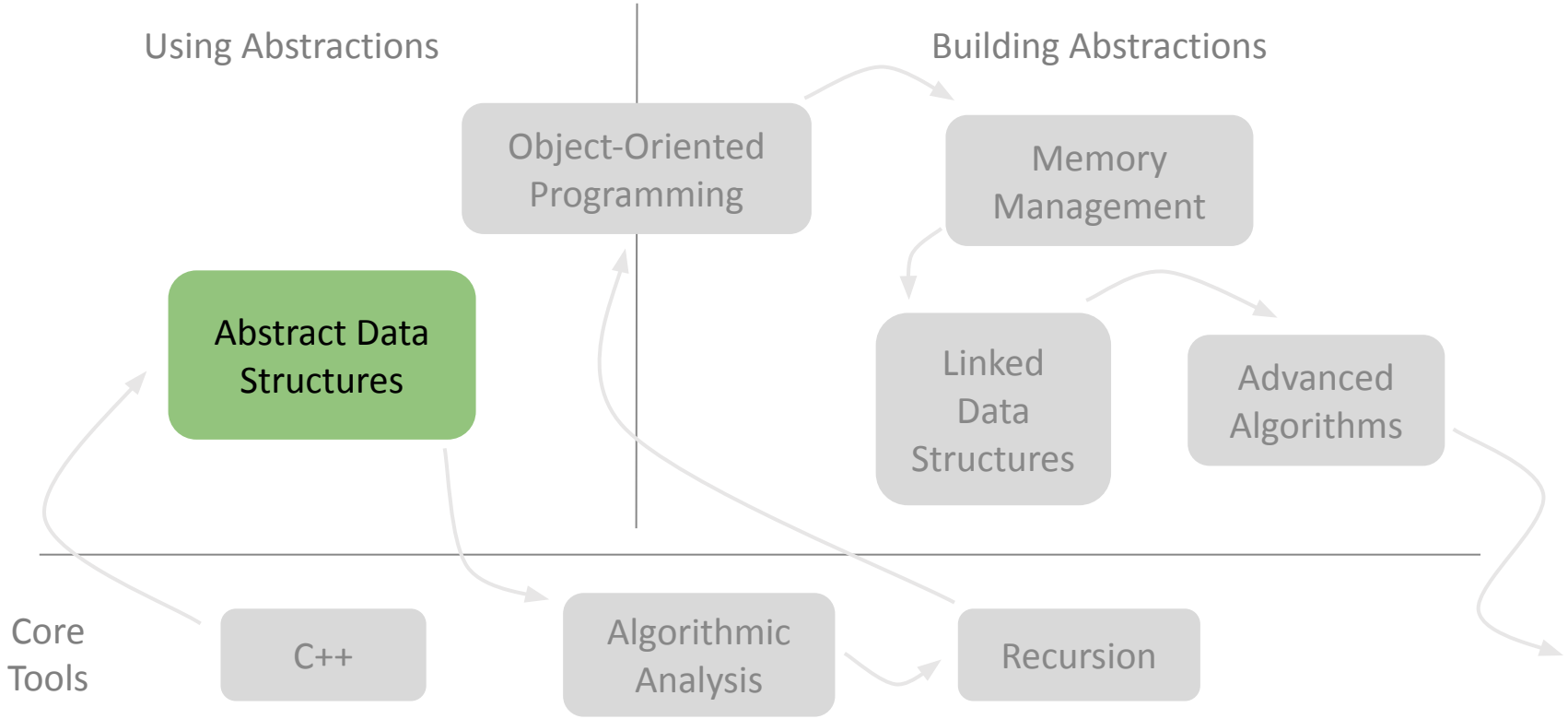
Types

- Stacks (LIFO)
- Queues (FIFO)

Properties

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

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Unordered Data

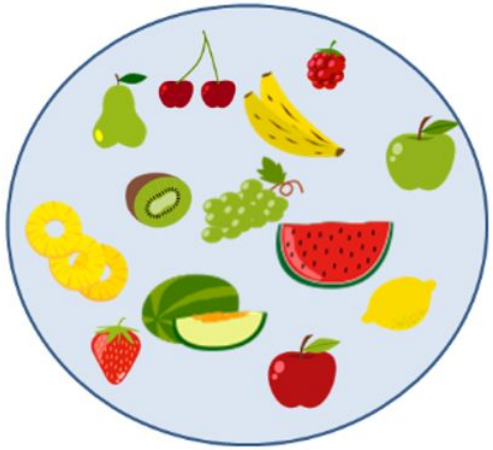
What are some examples of unordered data that you've encountered?

- Grocery list
- Unique visitors to a website
- Shuffled playlist of songs
- List of people who liked a post

Sets

What is a Set?

- An abstract data type (ADT)
  - Unordered collection of elements
- Stanford C++ library (here)
  - `#include "set.h"`
- No duplicate elements in a set
  - All unique elements
- Elements are not indexed
- Faster at finding elements than ordered data structures



The Stanford Set Library

```
#include “set.h”
```

- **`set.add(value)`**: Adds the value to the set, ignores if the set already contains the value
- **`set.remove(value)`**: Removes the value from the set, ignores if the value is not in the set
- **`set.contains(value)`**: Returns a boolean value, true if the set contains the value, false otherwise
- **`set.isEmpty()`**: Returns a boolean value, true if the set is empty, false otherwise
- **`set.size()`**: Returns the number of elements in the set

For more information, check out the Stanford Set class documentation!

Set Operations: Creating

```
Set<string> flagSet;
```

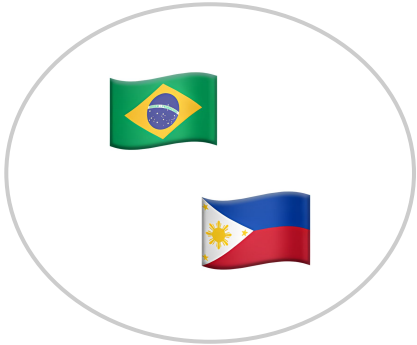

flagSet

Set Operations: Adding Elements

```
Set<string> flagSet;
flagSet.add("brazil");
```

Set Operations: Adding Elements

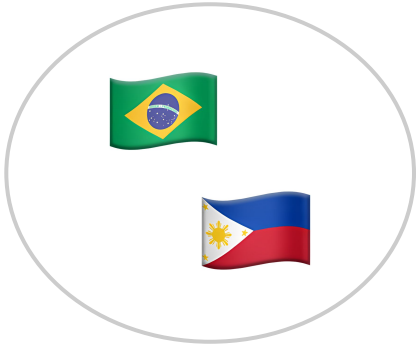
```
Set<string> flagSet;
flagSet.add(“brazil”);
flagSet.add(“philippines”);
```



flagSet

Set Operations: Adding Elements

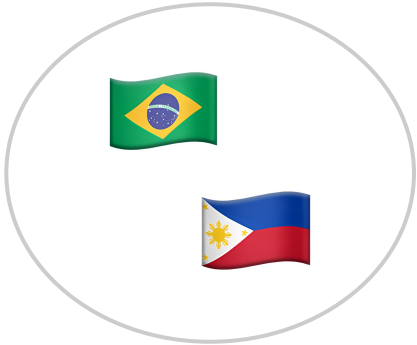
```
Set<string> flagSet;
flagSet.add(“brazil”);
flagSet.add(“philippines”);
flagSet.add(“brazil”);
```



flagSet

Set Operations: Removing Elements

```
Set<string> flagSet;
flagSet.add(“brazil”);
flagSet.add(“philippines”);
flagSet.add(“brazil”);
flagSet.remove(“brazil”);
```



flagSet

Set Operations: Removing Elements

```
Set<string> flagSet;
flagSet.add(“brazil”);
flagSet.add(“philippines”);
flagSet.add(“brazil”);
flagSet.remove(“brazil”);
cout << flagSet.remove(“philippines”) << endl;
```

flagSet

Set Operations: Removing Elements

```
Set<string> flagSet;
flagSet.add(“brazil”);
flagSet.add(“philippines”);
flagSet.add(“brazil”);
flagSet.remove(“brazil”);
```

flagSet

Set Operations: Checking for Elements

```
Set<string> flagSet;
flagSet.add(“brazil”);
flagSet.add(“philippines”);
flagSet.add(“brazil”);
flagSet.remove(“brazil”);
cout << flagSet.contains(“canada”) << endl;
```

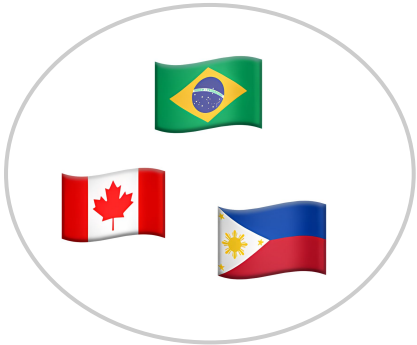
flagSet

Console:

```
false
```

Set Operations: Creating with Elements

```
Set<string> flagSet = {“brazil”, “philippines”,
                       “canada”};
```



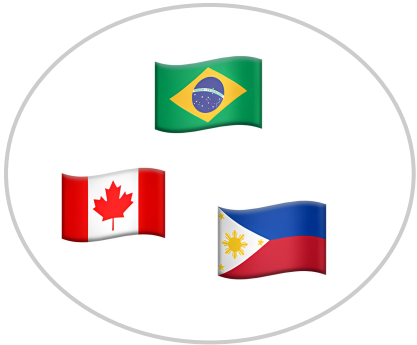
flagSet

Set Operations: Printing

```
Set<string> flagSet = {“brazil”, “philippines”,
                       “canada”};
cout << flagSet << endl;
```

Console:

```
{“brazil”, “canada”,
“philippines”}
```



flagSet

Set Patterns and Pitfalls

- Use for each loops to iterate over a set

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

- Cannot use anything that attempts to index into a set

```
for(int i=0; i < set.size(); i++) {
   // does not work, no index!
   cout << set[i];
}
```

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

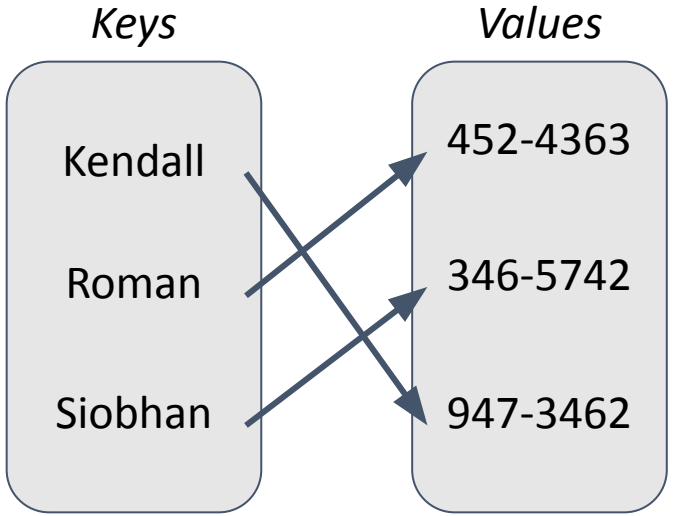
Sets can be compared, combined, etc

- **`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`**)

Unique Words Coding Example

Maps

What is a Map?



- An abstract data type (ADT)
  - Unordered collection of elements
- Stanford C++ library (here)
  - `#include "map.h"`
- Collection of pairs
  - Sometimes called key/value pairs
  - Use the key to quickly find the value
- Generalization of ordered data structure, where "indices" are not integers

The Stanford Map Library

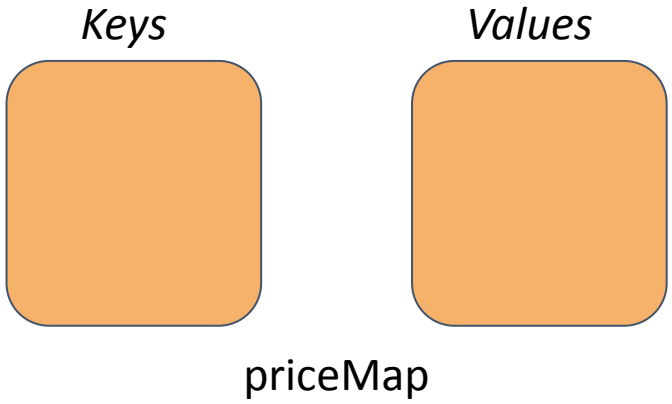
`#include "map.h"`

- **`map.clear()`**: Removes all key/value pairs from the map
- **`map.containsKey(key)`**: Returns **`true`** if the map contains a value for the given key
- **`map[key]`:** Returns the value mapped to the given key
  - If **`key`** is not in the map, adds it with the default value (e.g., **`0`** or **`""`**)
- **`map.get(key)`**: Returns the value mapped to the given key
  - If **`key`** is not in the map, returns the default value for the value type, but does not add it to the map.
- **`map.isEmpty()`**: Returns **`true`** if the map contains no key/value pairs (size 0)
- **`map.keys()`**: Returns a **`Vector`** copy of all keys in the map
- **`map[key] = value`** and **`map.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
- **`map.remove(key)`**: Removes any existing mapping for the given key (ignored if the key doesn't exist in the map)
- **`map.size()`**: Returns the number of key/value pairs in the map
- **`map.toString()`**: Returns a string such as "**`{a:90, d:60, c:70}`**"
- **`map.values()`**: Returns a **`Vector`** copy of all the values in the map

For more information, check out the Stanford Map class documentation!

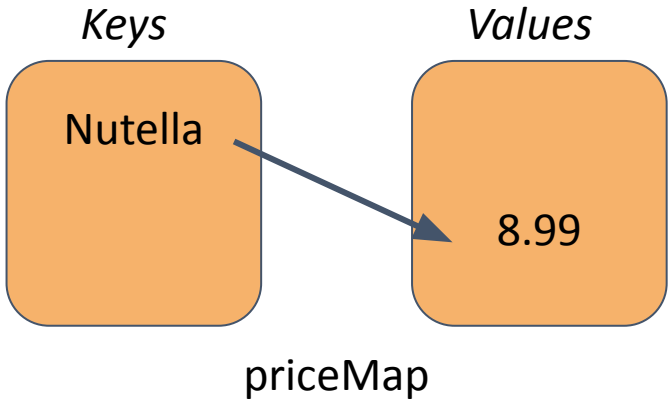
Map Operations: Creating

```
Map<string, double> priceMap;
```



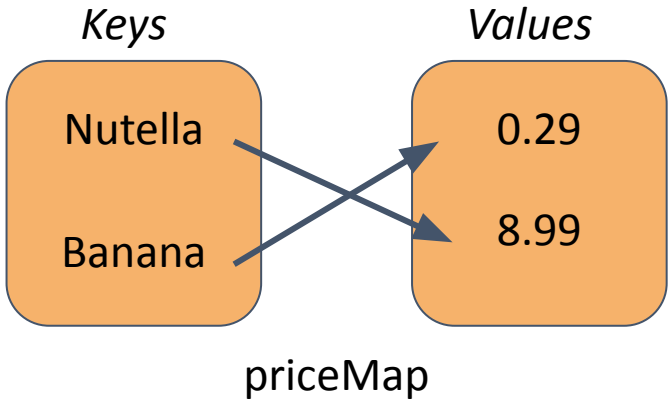
Map Operations: Adding Elements

```
Map<string, double> priceMap;
priceMap[“Nutella”] = 8.99;
```



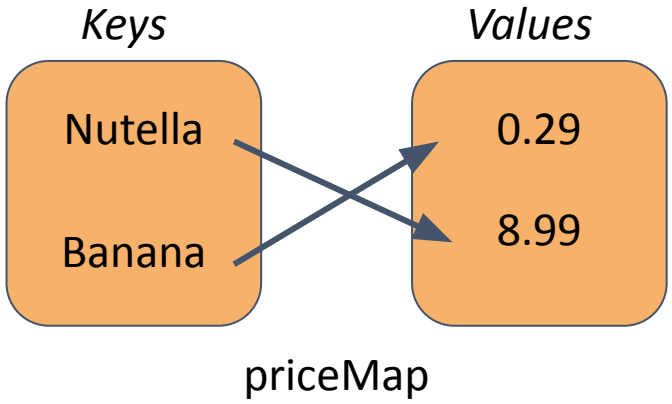
Map Operations: Adding Elements

```
Map<string, double> priceMap;
priceMap[“Nutella”] = 8.99;
priceMap.put(“Banana”, 0.29);
```



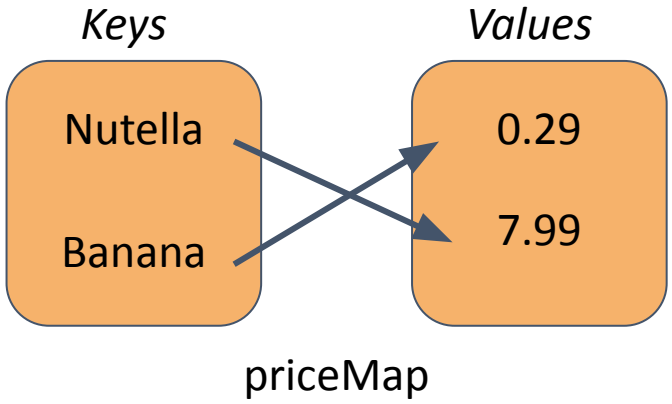
Map Operations: Adding Elements

```
Map<string, double> priceMap;
priceMap[“Nutella”] = 8.99;
priceMap.put(“Banana”, 0.29);
priceMap.put(“Nutella”, 7.99);
```



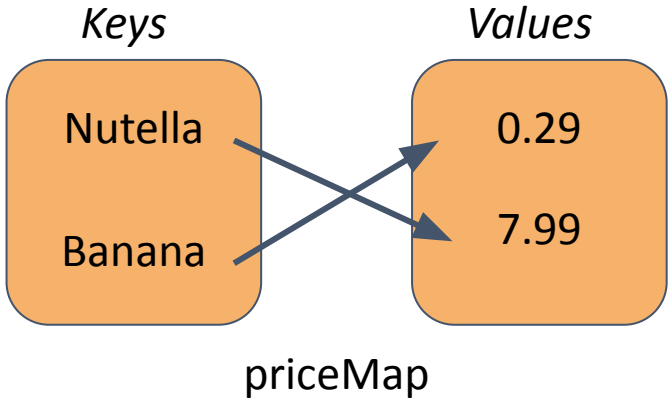
Map Operations: Adding Elements

```
Map<string, double> priceMap;
priceMap[“Nutella”] = 8.99;
priceMap.put(“Banana”, 0.29);
priceMap.put(“Nutella”, 7.99);
```



Map Operations: Accessing Elements

```
Map<string, double> priceMap;
priceMap[“Nutella”] = 8.99;
priceMap.put(“Banana”, 0.29);
priceMap.put(“Nutella”, 7.99);
cout << priceMap[“Banana”] << endl;
```

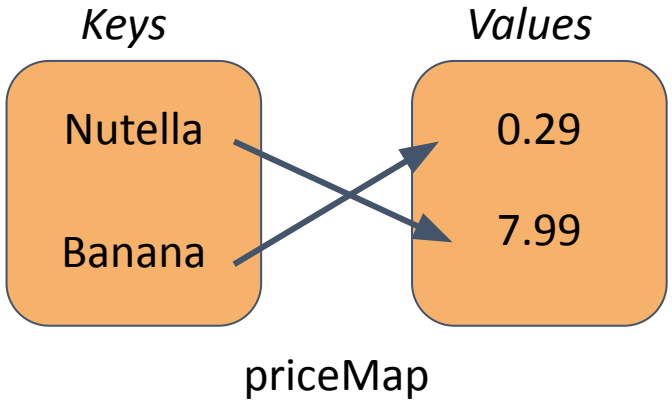


Console:

```
0.29
```

Map Operations: Accessing Elements

```
Map<string, double> priceMap;
priceMap[“Nutella”] = 8.99;
priceMap.put(“Banana”, 0.29);
priceMap.put(“Nutella”, 7.99);
cout << priceMap[“Banana”] << endl;
cout << priceMap.get(“Banana”) << endl;
```

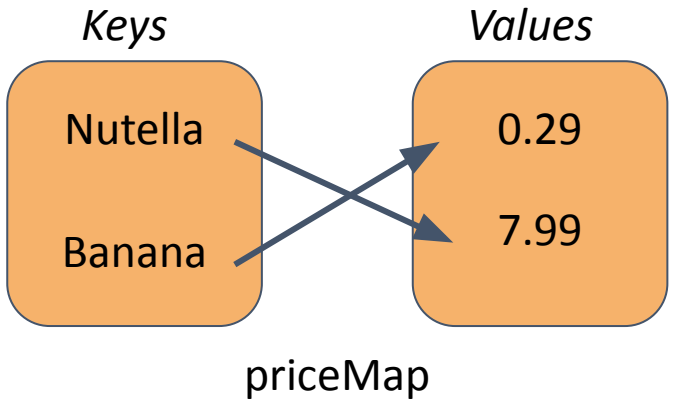


Console:

```
0.29
0.29
```

Map Operations: Creating with Elements

```
Map<string, double> priceMap =
            {{“Nutella”,7.99},{“Banana”,0.29}};
```

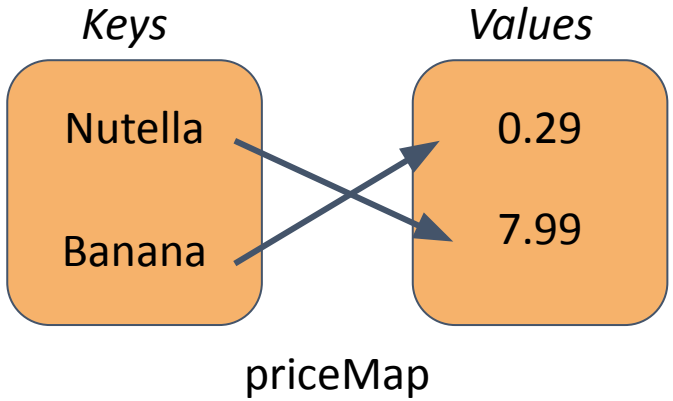


Map Operations: Printing

```
Map<string, double> priceMap =
                {{“Nutella”,7.99},{“Banana”,0.29}};
cout << priceMap << endl;
```

Console:

```
{"Banana":0.29,
"Nutella":8.99}
```



Map Patterns and Pitfalls

- Use for each loops to iterate over a map

```
for(type currKey : map) {
    // see map values using map[currKey]
    // don't edit the map
}

for(type currKey : map.keys()) {
    // see map values using map[currKey]
    // can now edit the map!
}
```

Map Patterns and Pitfalls

- Auto-insert: a 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]++;
}
```

Map Patterns and Pitfalls

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

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

Map Patterns and Pitfalls

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

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

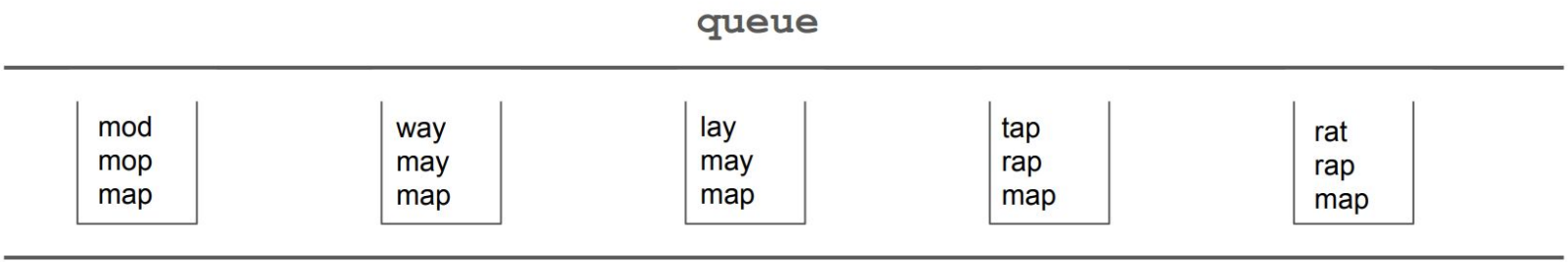
Unique Words Coding Example (Extended Version)

Nested Data Structures

Nested Data Structures

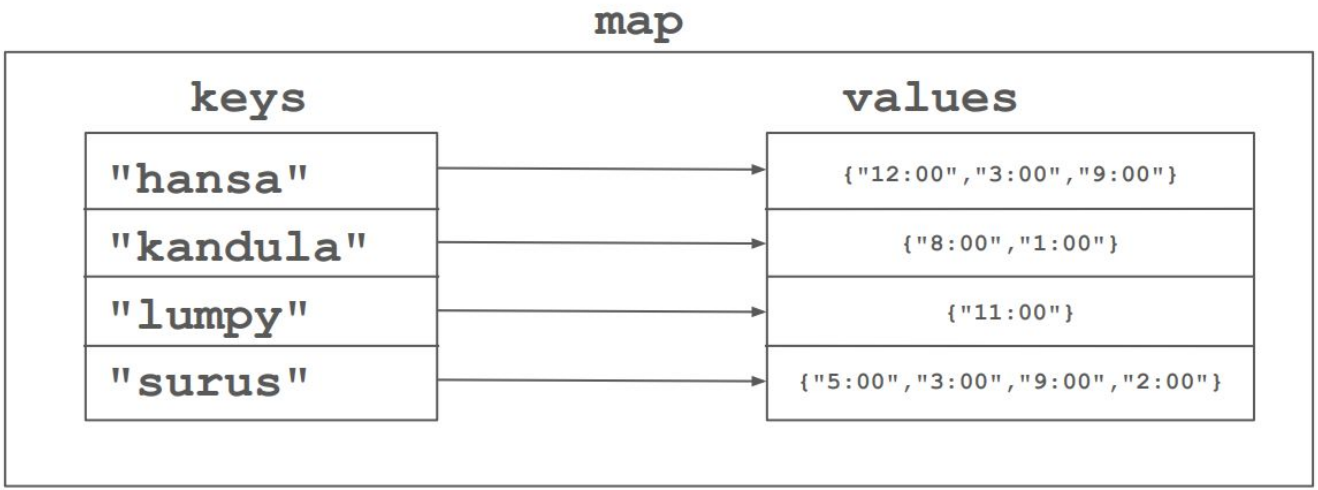
- Use one ADT as the data type inside of another ADT
- A great way of organizing data with complex structure
- Explore more in Assignment 2!

Nested Data Structures



Queue<_____>

Nested Data Structures

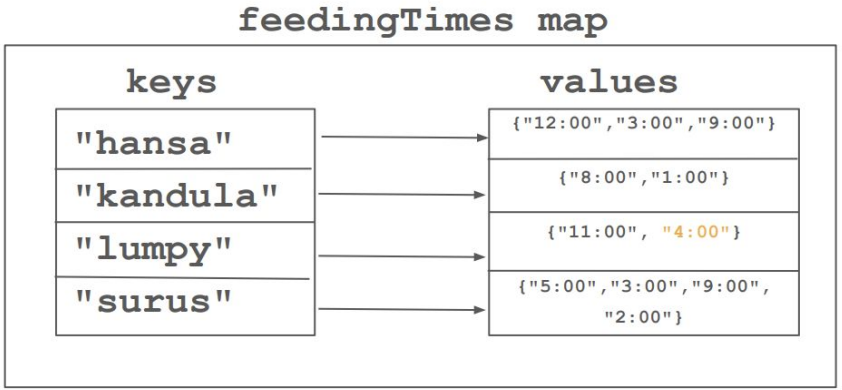


Map< _____, _____ >

Modifying Nested Data Structures

We want to add a second feeding time for “lumpy” at 4:00.

Which snippets of code will correctly update the map?



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

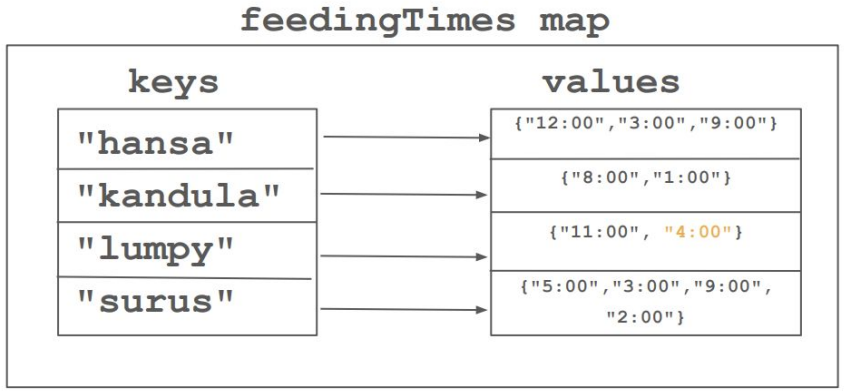
2.
```
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
```

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

Modifying Nested Data Structures

We want to add a second feeding time for “lumpy” at 4:00.

Which snippets of code will correctly update 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;
```

[] 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");
```

[] 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.

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

[] 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.

```
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 opportunities for bugs and mistakes at each level of nesting
  - Specifically in C++, working with nested data structures can be tricky due the use of references and copies.

Recap of ADTs

Ordered ADTs

Elements with indices

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

Elements without indices

- Stacks (LIFO)
- Queues (FIFO)

Unordered ADTs

- Sets (unique elements)
- Maps (key, value pairs)