More Data Structures

for PA#3 and Project

Beyond Arrays

- In C++, there is a `vector` class as part of the `std` namespace.
  - Likewise, this class internally uses an array and resizes it when necessary as new items are added to the conceptual underlying list.
  - This resizing is also handled internally and automatically by the class.

Lists

- Note that we now have two different ways of storing data, each of which has its own pros and cons.
  - Arrays
  - Good for adding items to the end of lists and for random access to items within the list.
  - Bad for cases with many additions and removals at various places within the list.

Lists

- Note that we now have two different ways of storing data, each of which has its own pros and cons.
  - Linked Lists
  - Better for adding and removing items at random locations within the list.
  - Bad at randomly accessing items from the list.
    - Note that to use a random item within the list, we must travel the chain to find it.

Lists

- Note that both of these objects fulfill the same end goal – to represent a group of objects with some implied ordering upon them.
- While they meet this goal differently, their primary purpose is identical.

Beyond Lists

- We have this notion of a “list” structure, which maps its stored objects to indices.
  - What if we don’t actually need to have a lookup position for our stored objects?
  - But wait! How could we possibly iterate over the objects in a for loop?

The Iterator

- Many programming languages provide objects called *iterators* for enumerating objects contained within data structures.
  - C++ and Java are no exceptions.
  - C++'s versions are defined in the <iterator> header file.

The Iterator

- This iterator may be used to get each contained object in order, one at a time, in a controllable manner.
  - It's especially designed to work well with `for` loops.

The Iterator

- Example code:

```
vector<int> numbers;

// omitted code initializing numbers.

iterator<int> iter;
for(iter = numbers.begin();
   iter != numbers.end(); iter++)
{
   cout << *iter << ' ';
}
```

The Iterator

- In C++, iterators are designed to look like and act something like pointers.
  - The * and -> operators are overloaded to give pointer-like semantics, allowing users of the iterator object to "dereference" the object currently "referenced" by the iterator.

The Iterator

- In C++, iterators are designed to look like and act something like pointers.
  - Furthermore, note the use of `operator ++` to increment the iterator onto the next item.
  - This is another way we can interact with pointers; it's useful for iterating across an array while using pointer semantics.

The Iterator

```
vector<int> numbers;

// omitted code initializing numbers.

iterator<int> iter;
for(iter = numbers.begin();
    iter != numbers.end(); iter++)
{
    cout << *iter << ' ';
}
```

The Iterator

- C++11 also provides an alternate version of the for-loop which is designed to work with iterable structures and iterators.

```
vector<Person> structure;
for(Person &p:structure)
{
   //Code.
}
```

The Iterator

- Both the `std::vector` and `std::list` classes of C++ implement iterators.
  - begin() returns an iterator to the list's first element.
  - end() is a special iterator "just after" the final element of the list, useful for checking when we're done with iteration.

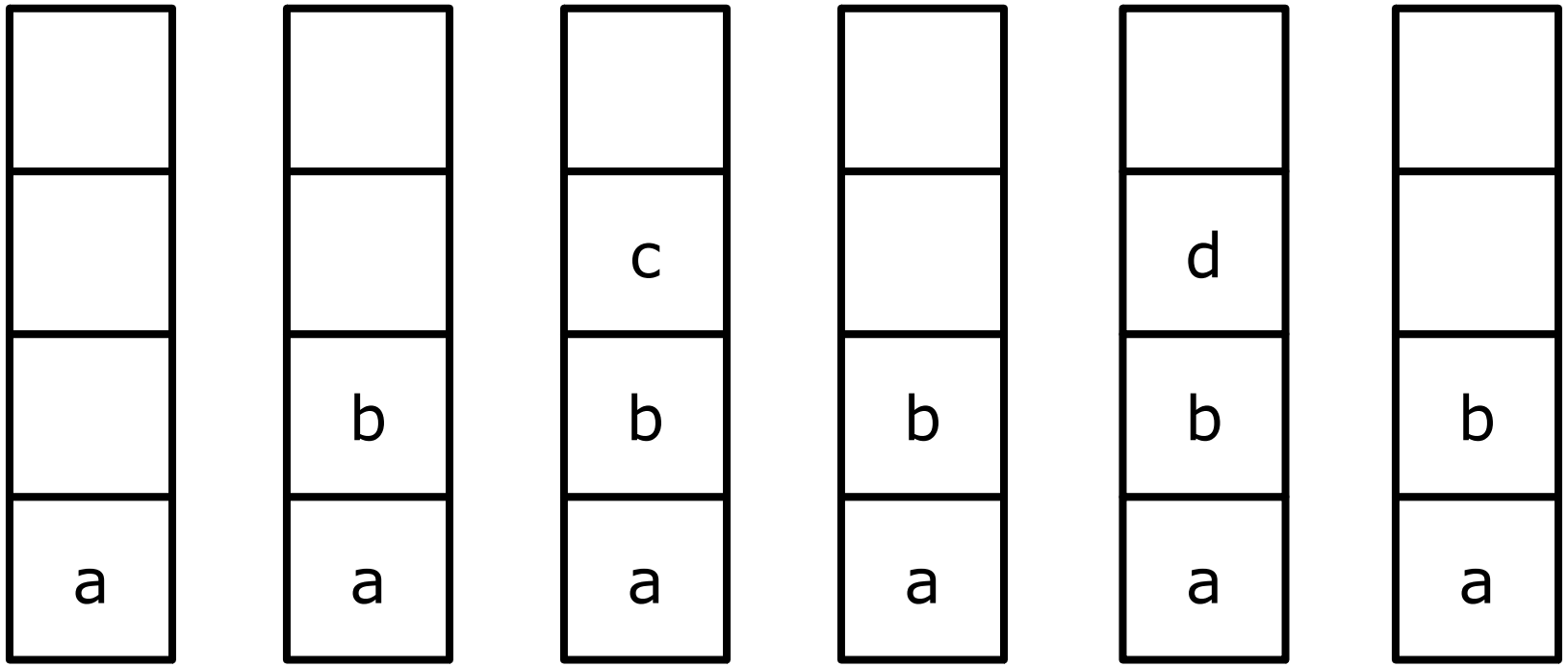
Input/Output Modeling

- Certain data structures exist to model specialized, restricted input and output behavior.
  - Consider the usual interaction someone might have with a *stack* of papers.
  - Another possibility: the usual behavior of a group of people waiting in line… in a *queue* waiting to be served.

Stacks

- The data structure known as a *stack* is a "Last In, First Out" (LIFO) structure.
  - That is, the last input to the structure is the first output obtained from it.
  - Consider a stack of papers – when searching through it, one typically starts at the top and searches downward, from newest to oldest.

Stacks



Stacks

- Stacks are a **very good** model for function calls.
  - Stacks are *the* model of how recursion mechanically works.
  - In turn, recursion is necessary for operating upon many data structures.

Stacks

- When debugging, the *stack trace* (or *call stack*) of a program at a given point of execution is exactly this – a description of the order of active method calls within the program.
- The area of memory where function data lives is literally called the *stack* space.

Stacks + Math

- Let’s consider the following mathematical expression:

$$2 + 5 * 7 – 6 / 3$$

- What order do we perform the operations in?
  - Consider trying to code something that would be able to interpret this!

Stacks + Math

- Using the standard order of operations, this becomes:

$$2 + (5 * 7) - (6 / 3)$$

- The postfix notation for this:

$$2\ 5\ 7\ *\ +\ 6\ 3\ /\ -$$

Stacks + Math

2 + (5 * 7) – (6 / 3)

2 + (35) – (2)

37 – 2

35

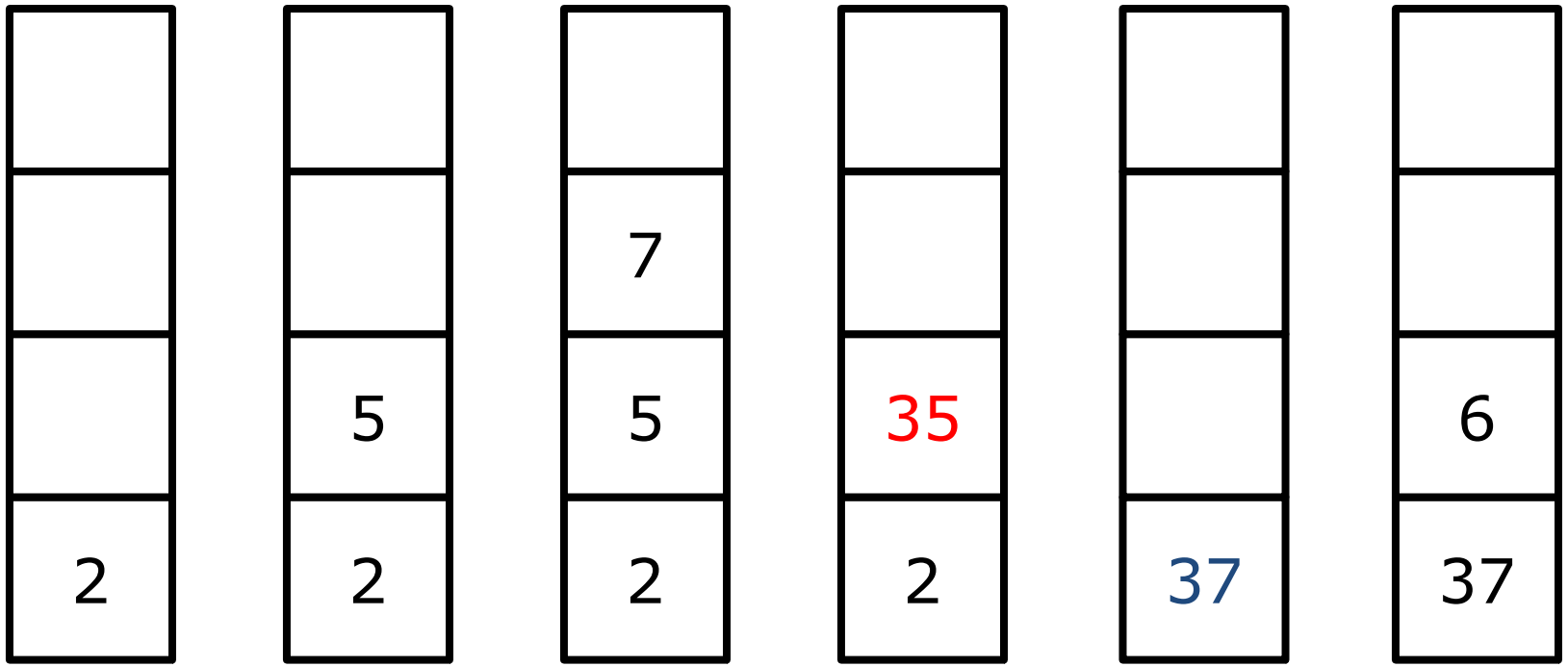
Stacks + Math

2 5 7 * + 6 3 / -

- Let’s see how this facilitates getting the right answer.

Stacks + Math

2 5 7 * + 6 3 / -



Stacks + Math

2 + (5 * 7) – (6 / 3)

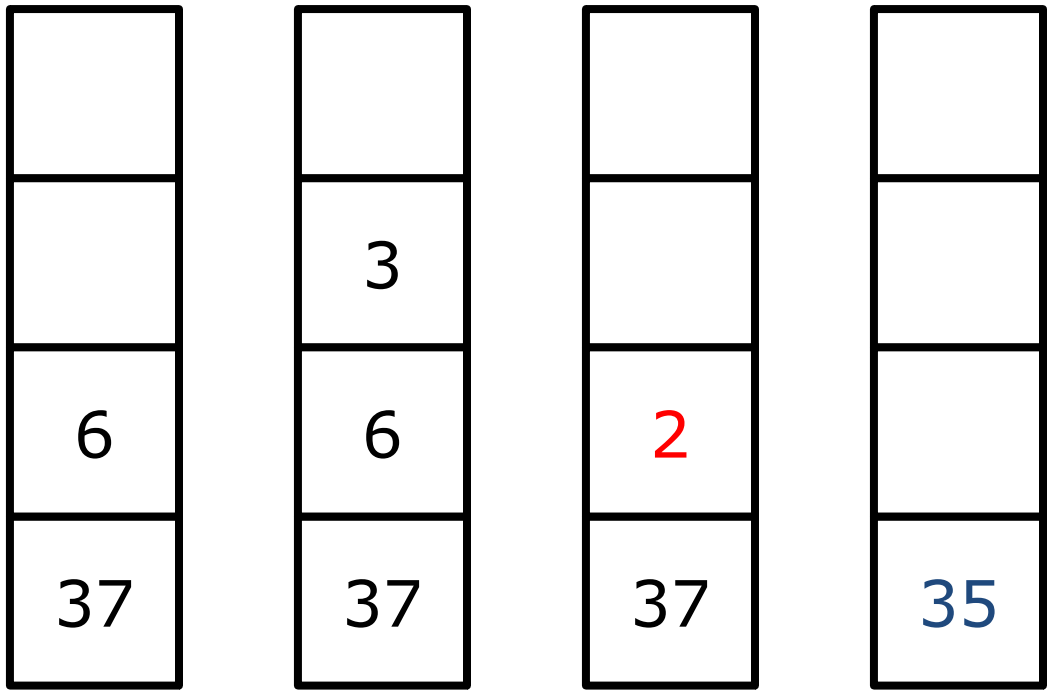
2 + (35) – (2)

37 – 2

35

Stacks + Math

2 5 7 * + 6 3 / -



Stacks + Math

2 + (5 * 7) – (6 / 3)

2 + (35) – (2)

37 – 2

35

Stacks + Math

- Math done in "standard" (i.e, *infix* notation) is typically first converted to postfix notation for actual computation.
  - This "conversion" is known as the Shunting-yard algorithm. It's up on Wikipedia, so feel free to take a look.

Stacks

- C++ provides the `std::stack` class.
  - This implementation is something of a "wrapper class" that uses a `vector`, `list`, or deque internally, limiting it to stack-like behavior.
  - We'll see deques in a moment.
  - The methods `push_back()`, `pop_back()`, and `back()` are designed from a stack perspective.

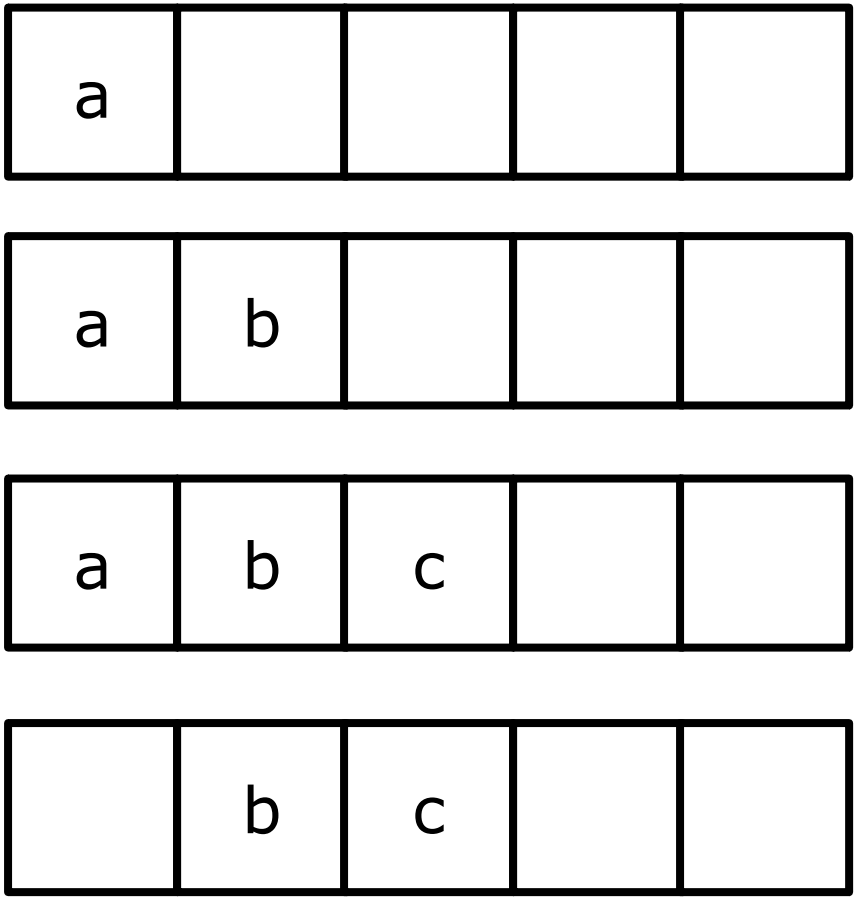
Queues

- The data structure known as a *queue* is a "First In, First Out" (FIFO) structure.
  - That is, the first input to the structure is the first output obtained from it.
  - Consider a line of people – the person in front has priority to whatever the line is waiting on… like buying tickets at the movies or gaining access to a sports event.

Queues

- Queues are significantly like lists, except that we have additional restrictions placed on them.
  - Additions may *only* happen at the list's end.
  - Removals may *only* happen at the list's beginning.
- As a result, *standard* array-based behavior may not be optimal.

Queues



Queues

- In C++, the queue class is provided.
  - This implementation is also something of a "wrapper class" that uses a `list`, or deque internally, limiting it to queue-like behavior.
  - `list` works well as a queue, as linked-lists can easily be altered from both ends.

Stacks + Queues

- The "deque", or *double-ended queue*, combines the behaviors of stacks and queues into a single structure.
  - Items may be added or removed at either end of the structure.
  - This allows for either LIFO or FIFO behavior – it's all in how you use the structure.
  - Mixed behavior is also possible, so beware!

Deques

- C++ thus defines the deque class for such uses.
  - This is a full-fledged object in its own right, and is array-based.
  - It may use multiple arrays and modular arithmetic, to allow efficient additions at the front for example.
  - It is the default object used internally by both `stack` and `queue`.