CSE373: Data Structures and Algorithms

Lecture 1: Introduction; ADTs; Stacks/Queues

Lauren Milne

Summer 2015

Welcome!

We will learn *fundamental data structures* and *algorithms* for organizing and processing information

- "Classic" data structures / algorithms
- Analyze efficiency
- When to use them

Today in class:

- Introductions and course mechanics
- What this course is about
- Start *abstract data types* (ADTs), *stacks*, and *queues*

Course staff

Lauren Milne
3rd Year CSE Ph.D. Grad Student
Works with Richard Ladner in Accessibility
Does triathlons and skijoring in free time

milnel2@cs.washington.edu
cse373-staff@cs.washington.edu

Course staff

Jiechen Chen
2^{nd} Year CSE Ph.D. Grad Student
Works in Theory. She moved 7 times
Before graduating from college!

`cse373-staff@cs.washington.edu`

Course staff

Yanling He
5^{th} Year CSE Masters Student,
Who loves climbing and caving.

`cse373-staff@cs.washington.edu`

Course staff

Naozumi Hiranuma

`cse373-staff@cs.washington.edu`

Course staff

Mert Saglam
2^{nd} Year CSE Ph.D. Grad Student
Works in Complexity Theory

cse373-staff@cs.washington.edu

Course staff

Mauricio Hernandez
Junior majoring in Electrical Engineering
With a concentration in Embedded Systems
Originally from Tabasco, Mexico!

`cse373-staff@cs.washington.edu`

Course Information

- http://courses.cs.washington.edu/courses/cse373/15su/
- Textbook: Weiss 3^{rd} Edition in Java
- Course email list: **cse373a_15sp@u.washington.edu**
- College of Arts & Sciences Instructional Computing Lab
  - http://depts.washington.edu/aslab/
  - Or your own machine
- Will use Java for the programming assignments
- Eclipse is recommended programming environment

Office Hours

- M 1:30-2:30 (Lauren)
- Tu 12-1 pm (Jiechen)
- W 12-1 pm (Nao)
- W 4-5 pm (Mert)
- Th 9:30-10:30 am (Yanling)
- F 12-1 pm (Mauricio)

Do these work for everyone?

Collaboration and Academic Integrity

DON'T CHEAT!

Seriously, read the policy online, follow Gilligan's Island rule.

What this course will cover

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Trees, Hashing, Dictionaries
- Heaps, Priority Queues
- Sorting
- Disjoint Sets
- Graph Algorithms
- Introduction to Parallelism and Concurrency

Goals

- Be able to make good design choices as a developer
- Be able to justify and communicate your design decisions
- This is not a course about Java!

To-do list

In next 24-48 hours:

- Read the web page
- Read all course policies
- Read Chapters 3.1 (lists), 3.6 (stacks) and 3.7 (queues) of the Weiss book
  - Relevant to Homework 1, due next week!

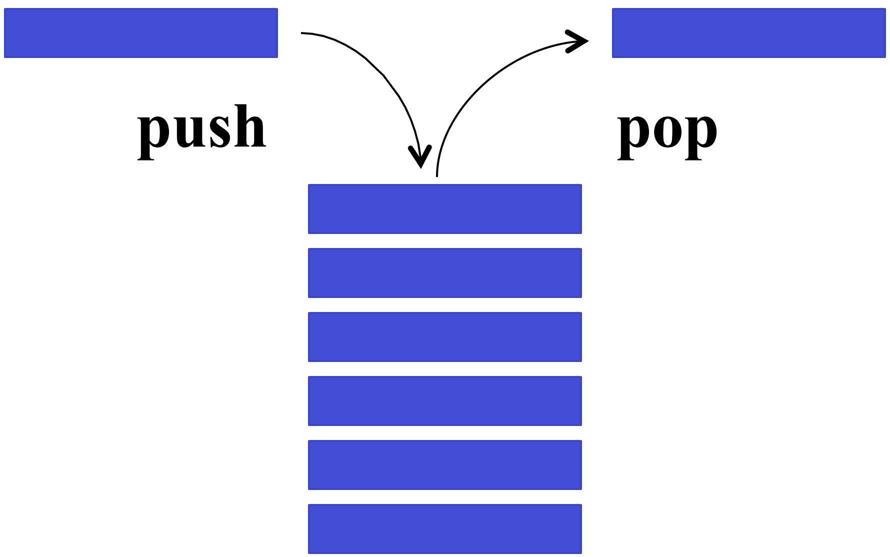
- Set up your Java environment for Homework 1

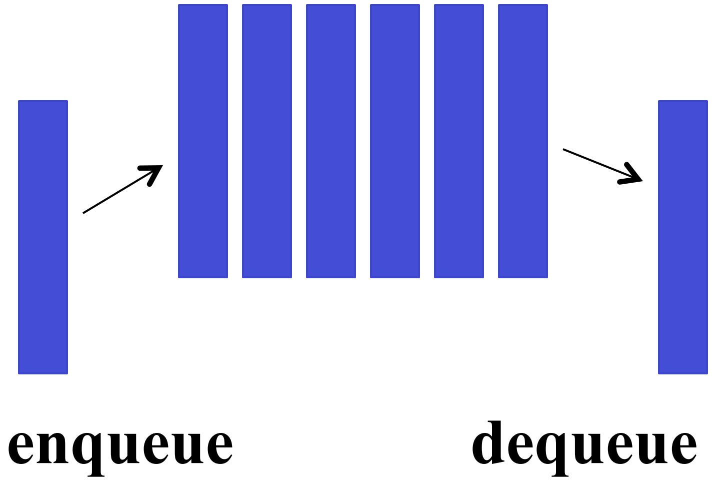
http://courses.cs.washington.edu/courses/cse373/15su/

Data structures

A data structure is a way to organize information to enable *efficient* computation over that information

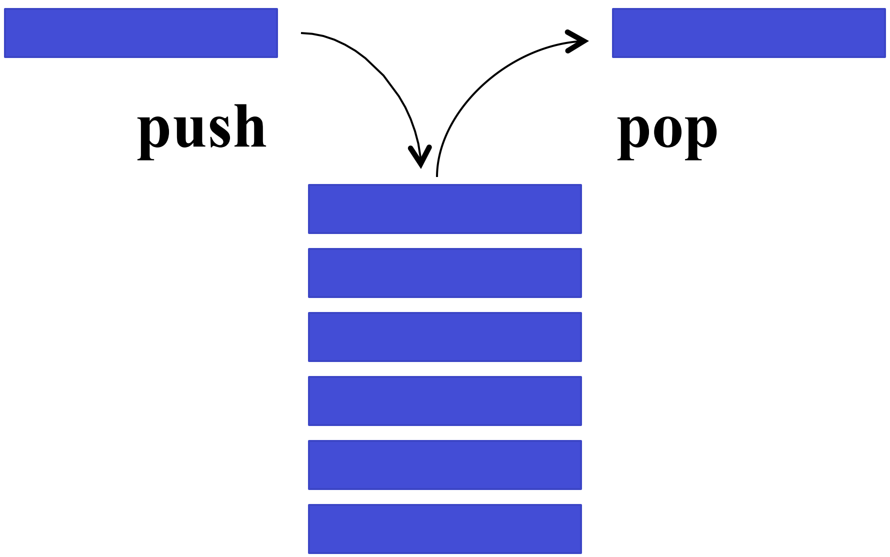
What are some examples?





Data structures

A data structure is a way to organize information to enable *efficient* computation over that information



It supports certain operations, each with a:

- Meaning
- Performance

Trade-offs

A data structure strives to provide many useful, efficient operations.

But there are unavoidable trade-offs:

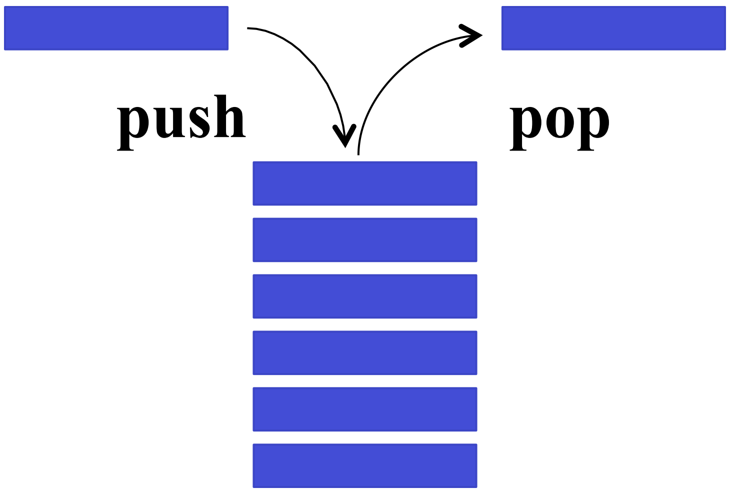
- Time vs space
- One operation's efficiency vs another
- Generality vs simplicity vs performance

So, when should I use different data structures?

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with set of operations
  - Not concerned with implementation details
- Data structure
  - A specific organization of data and family of algorithms for implementing an ADT
- Algorithm
  - A high level, language-independent description of a step-by-step process
- Implementation of a data structure
  - A specific implementation in a specific language

Example: Stacks



- The ***Stack*** ADT supports operations:
  - **`isEmpty`**
  - **`push`**
  - **`pop`**
  - What else?

- A Stack data structure could use a linked-list or an array and associated algorithms for the operations

- One implementation is in the library **`java.util.Stack`**

Why Useful

Stack ADT arises all the time in programming

- Recursive function calls
- Balancing symbols in programming (parenthesis)
- Evaluating postfix notation: 3 4 + 5 *
- Conversion from infix ((3+4)*5) to postfix notation

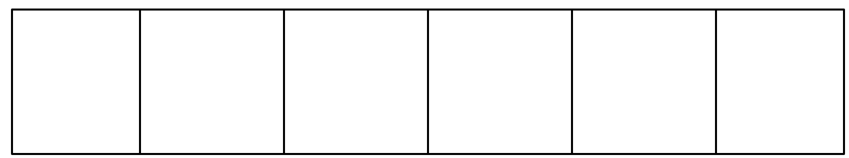
We can code up a reusable library

We can communicate in high-level terms

Stack Implementations

- stack as an array

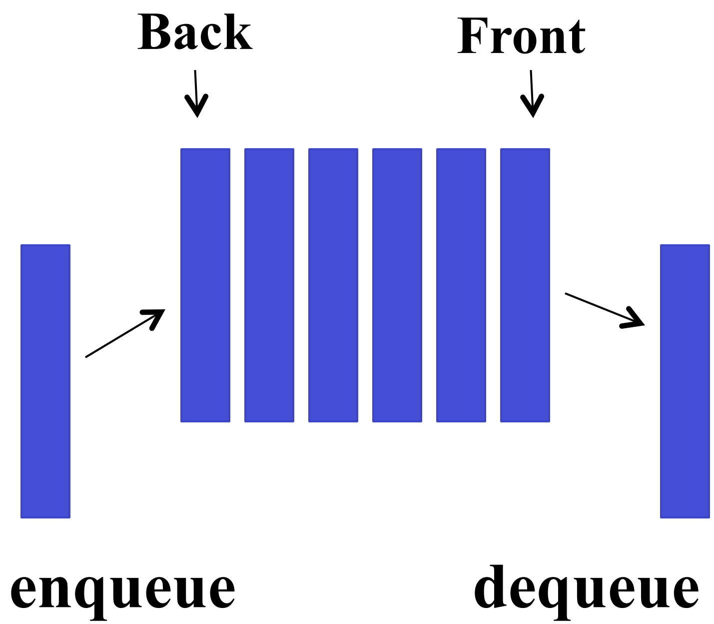
1. Initially empty
2. Push(‘a’)
3. Push(‘b’)
4. Pop()



- stack as a linked list

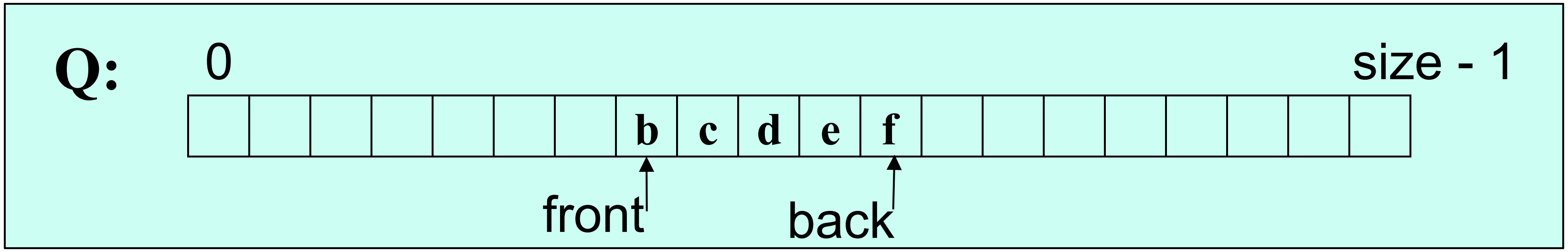
The Queue ADT

- Operations
  - **`enqueue`**
  - **`dequeue`**
  - **`is_empty`**
  - **`What else?`**



- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)

Circular Array Queue Data Structure

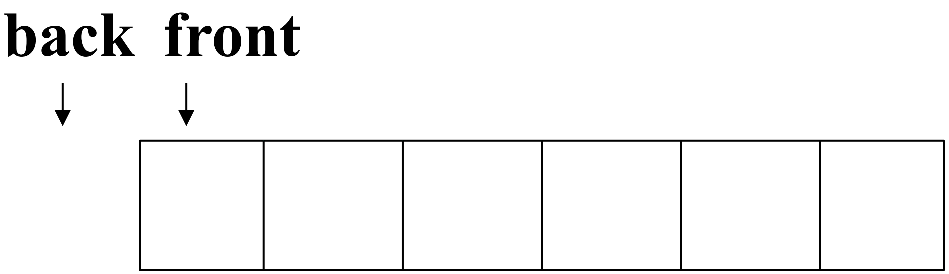



```
// Basic idea only!
enqueue(x) {
  next = (back + 1) % size
  Q[next] = x;
  back = next
}
```

```
// Basic idea only!
dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}
```

- What if ***queue*** is empty?
- What if ***array*** is full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?
- Why do we use a circular array for a queue vs a standard array for a stack?

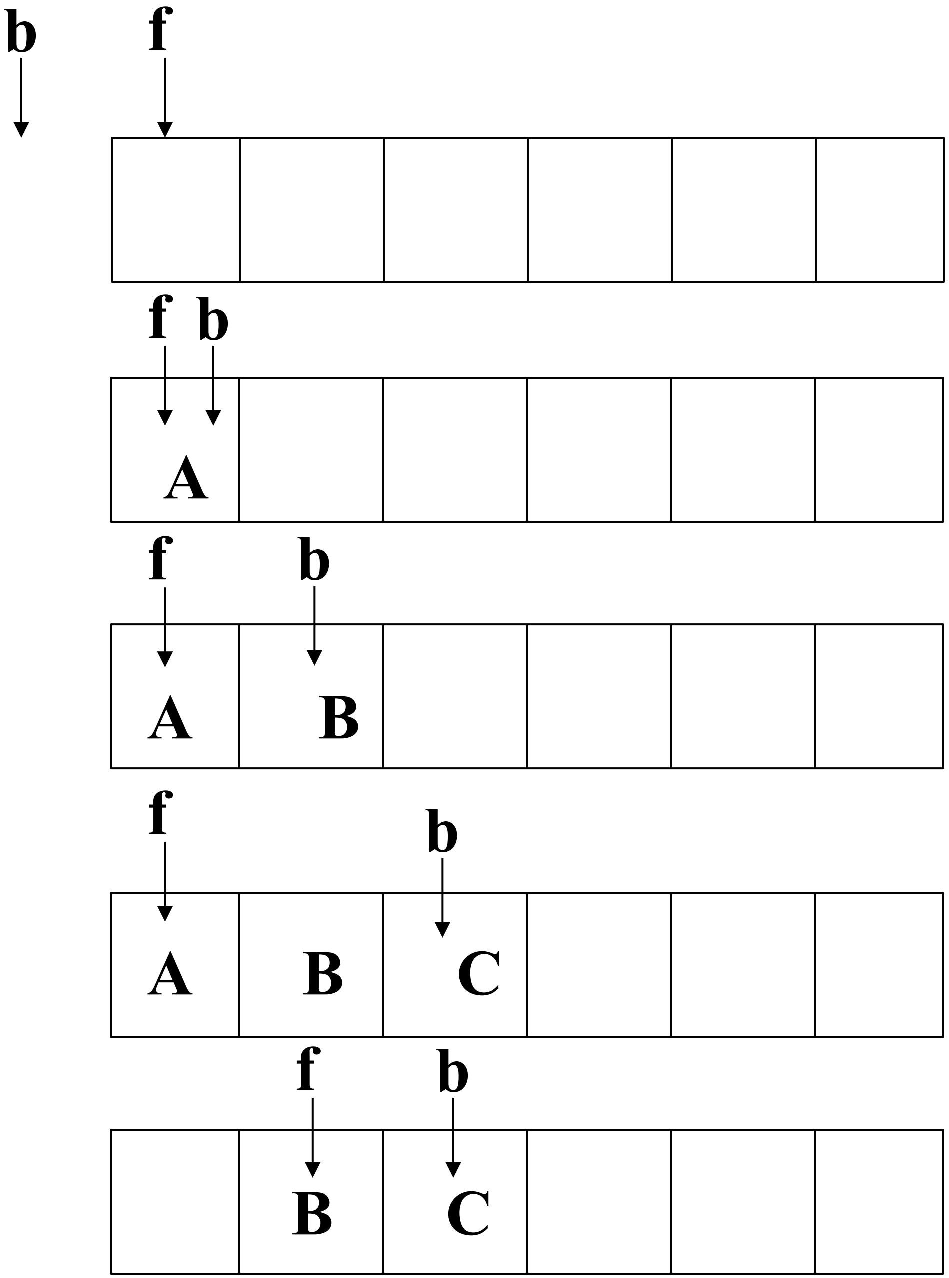
In Class Practice




```
// Basic idea only!
enqueue(x) {
  next = (back + 1) % size
  Q[next] = x;
  back = next
}
```

```
// Basic idea only!
dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}
```

enqueue('A')
enqueue('B')
enqueue('C')
o = dequeue()
o = dequeue()
enqueue('D')
enqueue('E')
enqueue('F')
enqueue('G')
enqueue('H')
enqueue('I')

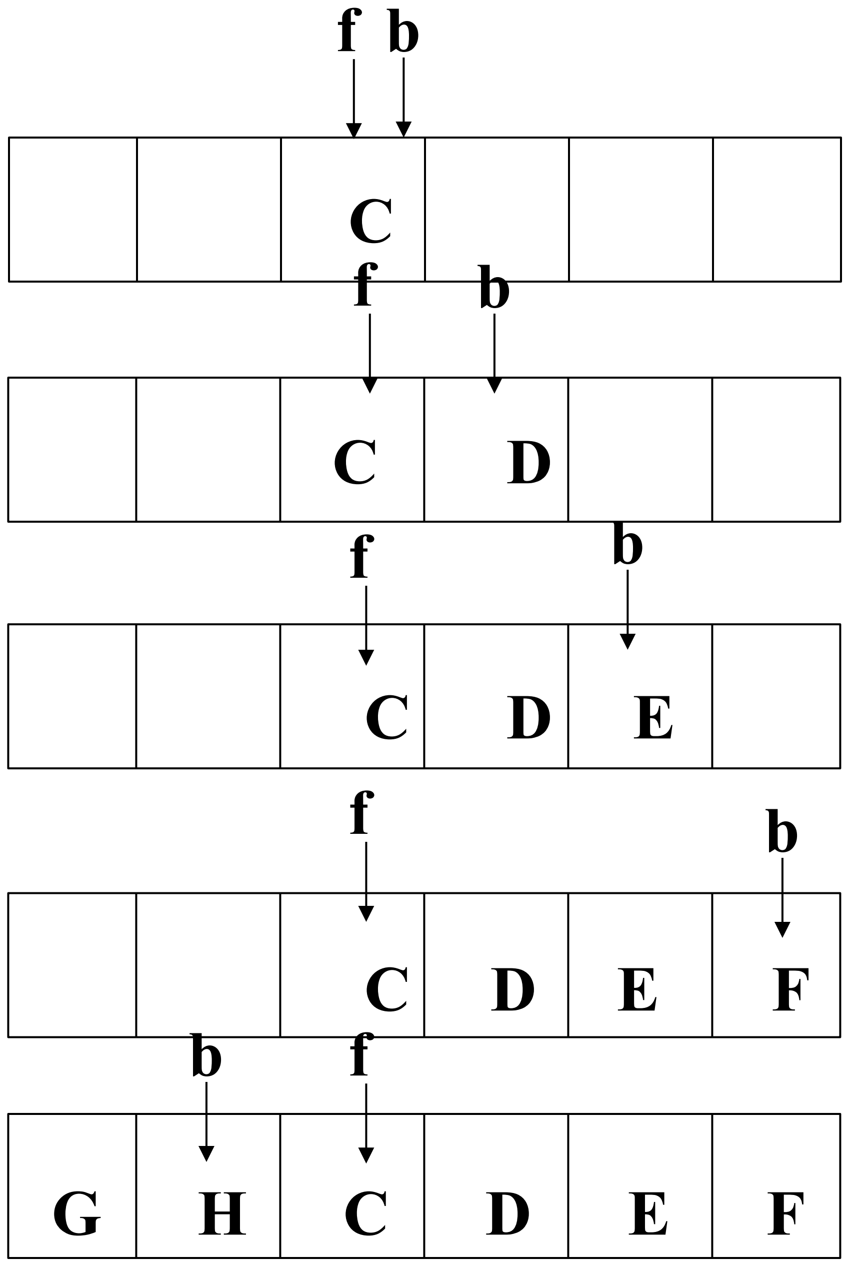


enqueue('A')

enqueue('B')

enqueue('C')

o = dequeue()



o = dequeue

enqueue('D')

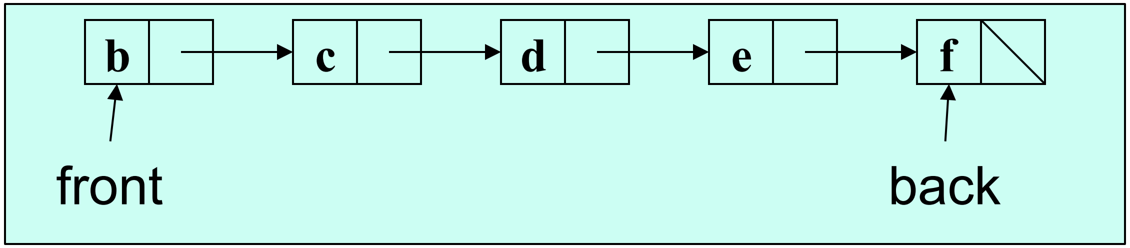
enqueue('E')

enqueue('F')

enqueue('G'), enqueue('H')

enqueue('i')

Linked List Queue Data Structure




```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

```
// Basic idea only!
dequeue() {
  x = front.item;
  front = front.next;
  return x;
}
```

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- Can ***list*** be full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?

Circular Array vs. Linked List

Array:

- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast
- Constant-time access to k^{th} element
- For operation insertAtPosition, must shift all later elements
  - Not in Queue ADT

List:

- Always just enough space
- But more space per element
- Operations very simple / fast
- No constant-time access to k^{th} element
- For operation insertAtPosition must traverse all earlier elements
  - Not in Queue ADT

Conclusion

- Abstract data structures allow us to define a new data type and its operations.
- Each abstraction will have one or more implementations.
- Which implementation to use depends on the application, the expected operations, the memory and time requirements.
- Both stacks and queues have array and linked implementations.
- We'll look at other ordered-queue implementations later.