COMP_SCI 214: Data Structures and Algorithms

Abstract Data Types

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 1 due today
- Homework 2 to be released later today

Self-evals

- Self-evaluation to be available Monday early morning
  - Self-evaluation is only based on **1st submission**
- You will receive 1st round feedback on Sunday (after late token deadline)
- Do your best for 1st hw submission
  - Including extensive tests
  - You'll be working on next HW during resubmission period → you will have limited time to fix issues
- Totally broken first submissions will result in no grader feedback given to you
  - 2nd submission would end up being equally unproductive
- 2nd submission assignment will open on Sunday

Testing

- Debugging tests is easier if you write **small** tests for specific purposes
- If you have a giant block of test, the error will just show you the test line number
  - But if you have 50 lines inside it, it's hard to narrow down the source of the test

Worklists

Worklists

- Say you need a program that:
  - Keeps track of "items" you need to handle
  - Allows you to fetch a single piece of "item" to handle next
- You may want to fetch:
  - The last item in **Last-in-first-out (LIFO)**
  - The earliest item in **First-in-first-out (FIFO)**
  - Some other item (maybe by priority?)

Examples of worklists

- Is each a FIFO or a LIFO?
  1. Food orders to handle
     - FIFO
  2. Pages clicked on in a browser (with ability to go back)
     - LIFO
  3. Dishes placed in a sink to wash
     - LIFO
  4. Edits in a word processor that allows undoing
     - LIFO
  5. Playing tracks in a playlist
     - FIFO

Stacks and queues

- Last-in-first-out worklist is called “Stack”
- First-in-first-out worklist is called “Queue”

Stack data and operations

- **Data:** set of task or item objects
- **Operations:**
  - Push item onto top of stack
  - Pop the top item from stack
  - Check if the stack is empty

Queue data and operations

- **Data:** set of task or item objects
- **Operations:**
  - Add item to the end of queue (enqueue)
  - Remove item from front of queue (dequeue)
  - Check if the queue is empty

How can we implement stacks and queues?

- Using data structures we know so far:
  - Vector/Array
  - Linked list

- Does this approach seem familiar?

Recall: Dynamic arrays

- What if we want to have a resizable collection of elements?
  - Like Python's `list` or Java's `ArrayList`

- Can we implement this with any of the data structures we've seen so far?
  - Vector/Array
  - Linked list

Recall: Comparing implementations for dynamic arrays

- Both class implementations have the exact same operations (`get_ith`, `append`)
- The way a client would use both implementations is exactly the same

```
let arr = DynamicArray()
arr.append(2)
```

  - But each implementation is doing something else under the hood
- The dynamic array idea here is **abstract**
- The underlying vector or linked lists are **concrete** implementations

Similarly…

- Stacks and queues are described **<u>only</u>** by their operations and expected behavior
  - **Abstract Data Types**

- We are choosing to implement them using vectors or linked lists
  - Since that's all we've learned so far
  - **Data structure**

Abstract Data Types

Abstract Data Types (ADTs)

- Proposed by Barbara Liskov in 1874
  - Turing Award winner 2008 for this work (and more)
- One of the most important advances in programming
- Will be a major guiding notion in this class

What is an ADT?

- An ADT defines:
  - A set of (abstract) objects or values
  - A set of (abstract) operations on those values
- An ADT omits:
  - How the values are concretely represented (data type, layout, etc.)
  - How the operations actually work
- Offers clients and developers freedom:
  - Can choose between many different representations and operation implementations (with tradeoffs)

- **Data:** set of task or item objects
- **Operations:**
  - Push item onto top of stack
  - Pop the top item from stack
  - Check if the stack is empty

ADT: Stack

- Abstract values look like: →
- Abstract operations signature:
  - push(Stack, Element): None
  - pop(Stack): Element
  - empty?(Stack): Bool

top	34
	2
	6
	-9

ADT: Stack

- Abstract values look like: →

top	34
	2
	6
	-9

- Abstract operations signature as DSSL2 interface:

```
interface STACK:
    def push(self, element)
    def pop(self)
    def empty?(self)
```

- DSSL2 interface ≈ C++ abstract class
  - Interfaces specify operations but not how they work
  - Classes implement interfaces to fill in how they work

ADT: Stack

- Abstract values look like: →

top	34
	2
	6
	-9

- Abstract operations signature as DSSL2 interface (with contracts):

```
interface STACK:
    def push(self, element: T ) -> NoneC
    def pop(self) -> T
    def empty?(self ) -> bool?
```

- Contracts check type-like constraints during program execution
  - See docs and supplementary video on Canvas

ADT: Practice with stack operations (LIFO)

Operations

- Abstract stack variable **s**

- `s.empty?()` → return T
- `s.push(6)` → pushes silently
- `s.push(5)` → pushes silently
- `s.push(-2)` → pushes silently
- `s.pop()` → remove and return -2
- `s.pop()` → remove and return 5
- `s.empty?()` → return F

Data

- Possible abstract representation of **s**

ADT: Practice with queue operations (FIFO)

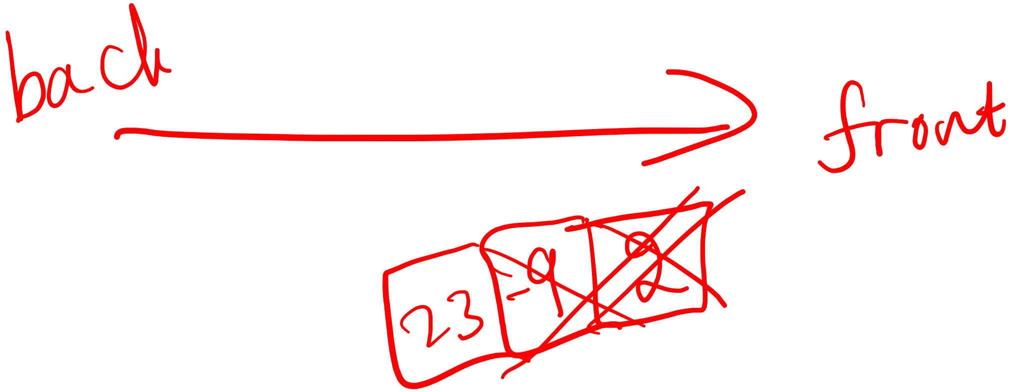
Operations

- Abstract queue variable **q**

- `q.dequeue()` → remove and return error
- `q.enqueue(2)` → pushes silently
- `q.enqueue(-9)` → pushes silently
- `q.empty?()` → return F
- `q.dequeue()` → remove and return 2
- `q.enqueue(23)` → pushes silently
- `q.dequeue()` → remove and return -9

Data

- Possible abstract representation of **q**



Stack vs. queue interfaces: What's the difference?

```
interface STACK[T]:
    def push(self, element: T) -> NoneC
    def pop(self) -> T
    def empty?(self) -> bool?
interface QUEUE[T]:
    def enqueue(self, element: T) -> NoneC
    def dequeue(self) -> T
    def empty?(self) -> bool?
```

- They seem the same except for the function and interface names!
- But queues and stacks should be doing different things under the hood!
  - How can we define these requirements?

ADTs should contain one more thing

- An ADT defines:
  - A set of (abstract) objects or values
  - A set of (abstract) operations on those values
  - A set of laws that specify correct behavior (inputs/outputs/program state change)
    - So an implementer knows how to implement operations correctly
    - So a client using the ADT implementation knows what to expect

Adding laws

$$\{p\}\ f\,(x) \Rightarrow y\ \{q\}$$

means that if precondition p is true when we apply f to x then we will get y as a result, and postcondition q will be true afterward.

- Examples:
  - $\{a = [2, 4, 6, 8]\}\ a[2] \Rightarrow 6\ \{a = [2, 4, 6, 8]\}$
  - $\{a = [2, 4, 6, 8]\}\ a[2] = 19 \Rightarrow \text{None}\ \{a = [2, 4, 19, 8]\}$

Hoare triples

$$\{p\}\ \ f\,(x) \Rightarrow y\ \ \{q\}$$

- This notation is called Hoare triples, after Sir C. A. R. (Tony) Hoare
  - 1980 Turing award, quicksort, concurrency, etc.
- **Note:** this is not code, it's *math* that says what code should do.

Adding laws to stack ADTs

```
def push(self, element)
def pop(self)
def empty?(self)
```

- Abstract values look like: |3, 4, 5 | *(bottom -> top)*
- Laws:
  - { s = | |} *s.empty?()* ⇒ True {}   #Empty postcondition => same as precondition
  - { s = |$e_1, \ldots, e_k, e_{k+1}$|} *s.empty?()* ⇒ False {}
  - { s = |$e_1, \ldots, e_k$|} *s.push(e)* ⇒ None { s = |$e_1, \ldots, e_k$, e|}
  - { s = |$e_1, \ldots, e_k, e_{k+1}$|} *s.pop()* ⇒ e_{k+1} { s = |$e_1, \ldots, e_k$|}
- Anything missing?
  - If there is no law for a case, we say the law is silent

Adding laws to queue ADTs

```
def enqueue(self, element)
def dequeue(self)
def empty?(self)
```

- Abstract values look like: | 3, 4, 5 | *(front -> back)*
- Laws:
  - $\{ q = |\,| \}$ *q.empty?()* ⇒ True $\{\}$
  - $\{ q = | e_1, \ldots, e_k, e_{k+1} | \}$ *q.empty?()* ⇒ False $\{\}$
  - $\{ q = | e_1, \ldots, e_k | \}$ *q.enqueue(e)* ⇒ None $\{ q = | e_1, \ldots, e_k, e | \}$
  - $\{ q = | \cancel{e_1}, \ldots, e_k | \}$ *q.dequeue()* ⇒ e_1 $\{ q = | e_2, \ldots, e_k | \}$

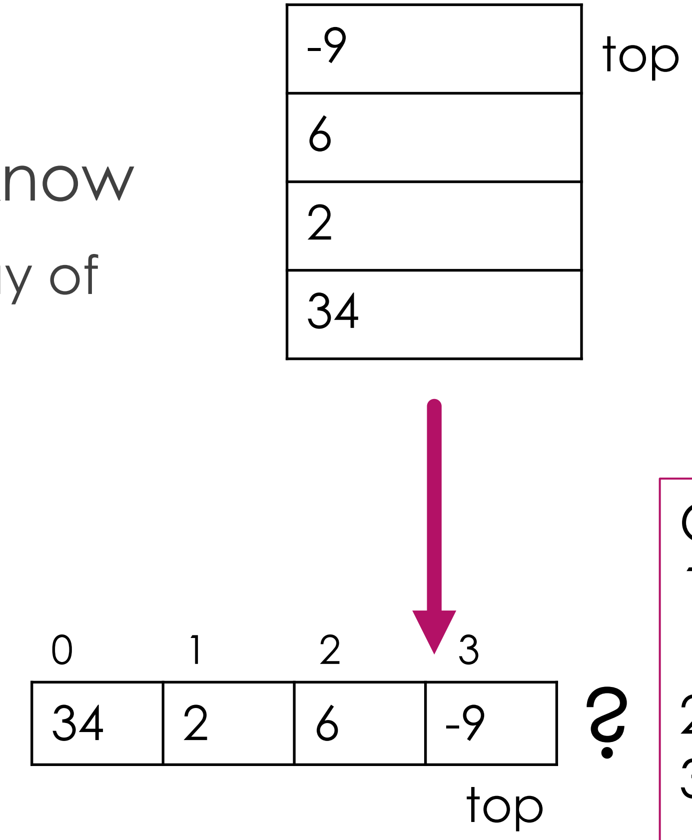
Pause

- Any questions or anything unclear?

Implementing stacks and queues

How can we implement a stack?

- Using a data structure we know
  - A vector/array of fixed size



Questions we need to answer:

1. How we can enable adding of new elements?
2. Where can we add a new element?
3. Where do we remove an element from?

What do we need for an implementation?

1. A concrete data representation of the stack or queue using array
2. Function definitions for interface functions while satisfying laws
3. A representation for each item in the stack/queue

Let's brainstorm these 2

Brainstorm for step 1

- What information (as a variable) do we need to keep track of for a stack array?
  - An array
    - How big?
  - What else?

34	2	6	-9		

Questions we need to answer:
1. How we can enable adding of new elements?
2. Where can we add a new element?
3. Where do we remove an element from?

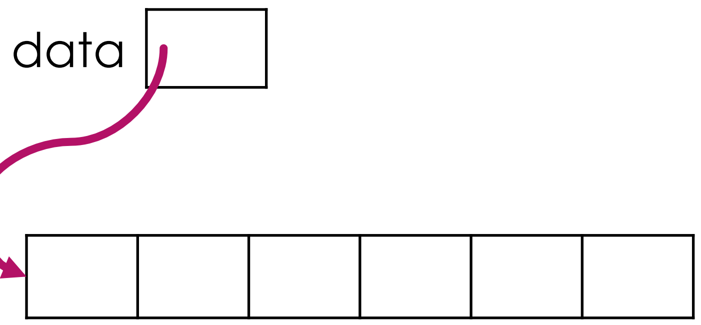
In-class exercise up next

- Reminders:
  - Link is available on Canvas on the homepage (if on mobile: click on "Syllabus" to get to the homepage)
  - This is not an attendance quiz → graded based on engagement and specific criteria (which are specified)

- Questions are NOT to be shared with your classmates not here
- Modifier contains flexibility if you need to miss some classes (if you're sick or any other reason)

In-class exercise (4 minutes)

1. To ensure the array has space for additions down the line, which approach would you choose?
    a. allocate a large array at the start and disallow insertions when it's full
    b. keep making new arrays (like with dynamic arrays) each time an element is added?
2. Explain your answer (1-2 sentences)



- Got us thinking about tradeoffs!

Brainstorm for step 1

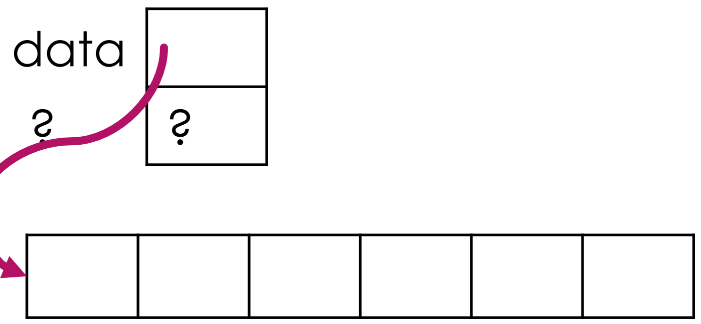
- What information (as a variable) do we need to keep track of for a stack array?
  - An array
    - Sufficiently large with empty spaces
  - What else?

34	2	6	-9		

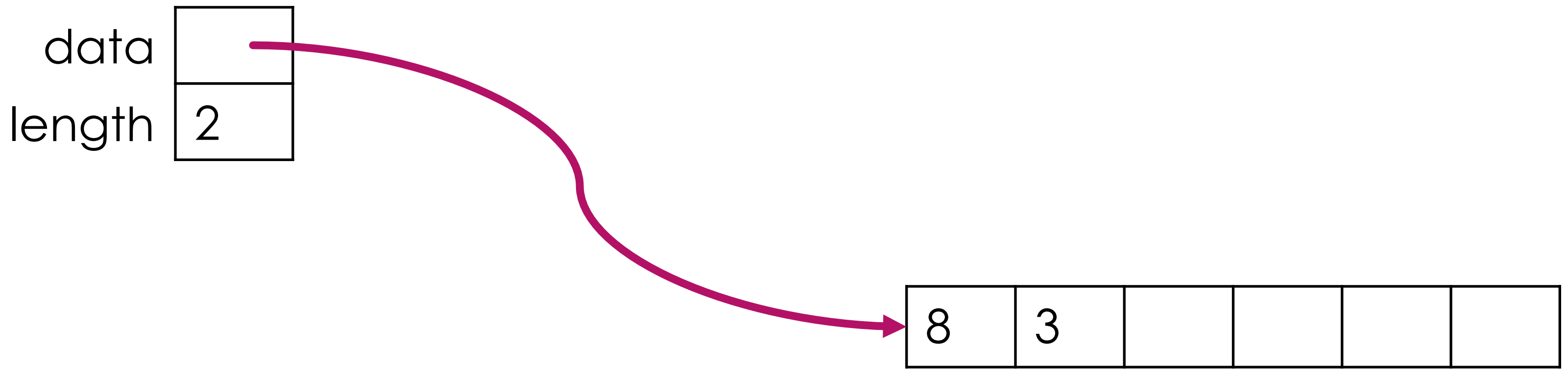
Questions we need to answer:
1. How we can enable adding of new elements?
2. Where can we add a new element?
3. Where do we remove an element from?

What else does the representation need?

- What's another variable that can be added to the representation to help operations know where to add or remove?



Step 1: Representation of data



- `length` tells us how many items are "in" the array
  - Not the same as the size/capacity of the array
  - There may be empty unused spaces in the array

Brainstorm for step 2

- **<u>Given:</u>** array of fixed size, length of stack
- How would we implement `push(element)`?

Before the operation

0	1	2	3	4	
34	2	6	-9		

length = 4

Push 45 onto stack

After the operation

34	2	6	-9	45	

length = 5

Returned from operation None

Brainstorm for step 2 contd.

- **Given:** array of fixed size, length of stack
- How would we implement `pop()` ?

Before the operation

34	2	6	23		

Pop from stack

length = 4

After the operation

34	2	6			

length = 3

Returned from operation

23

Brainstorm for step 2 contd.

- **<u>Given:</u>** array of fixed size, index of current top
- How would we implement `empty?()` ?

Before the operation

34	2	6	23		

Check if empty?

length = 4

After the operation

34	2	6	23		

length = 4

Returned from operation False

What do we need for an implementation?

1. A concrete data representation of the stack or queue using array
2. Function definitions for interface functions while satisfying laws
3. A representation for each item in the stack/queue

Have ideas now

- Let's think about a concrete implementation now

Stacks: Implementation step 1

stack-array.rkt

- Define array implementation to hold elements
  - Define array of fixed (maybe large) capacity
  - Keep track of number of elements

```
class StackArray[T] (STACK): # T can be any type
    let data: VecC[OrC(T, NoneC)]
    let length: int?
```

Stacks: Implementation step 2

stack-array.rkt

- Define stack functions required by interface using arrays
- Define any other functions relevant to implementation

```
class StackArray[T] (STACK):
    # fields from previous slide here

    def __init__(self, cap): #Specify array capacity …
    def push(self, element: T) -> NoneC: …
    def pop(self) -> T: …
    def empty?(self ) -> bool: …
```

Stacks: Implementation step 3

stack-array.rkt

- Define representation for each element (only needed for tests and actual usage of stack class). Could be:
  - Numbers, strings or other basic types
  - A struct object (need to define this first). Example:

```
struct browser_click:
    let url
    let timestamp
```

- `let s = StackArray[int?](5) #int becomes the T type`
- `let s = StackArray[browser_click?](6) #same here`

Pause

- Any questions or anything unclear?

If time: let’s try using our StackArray library

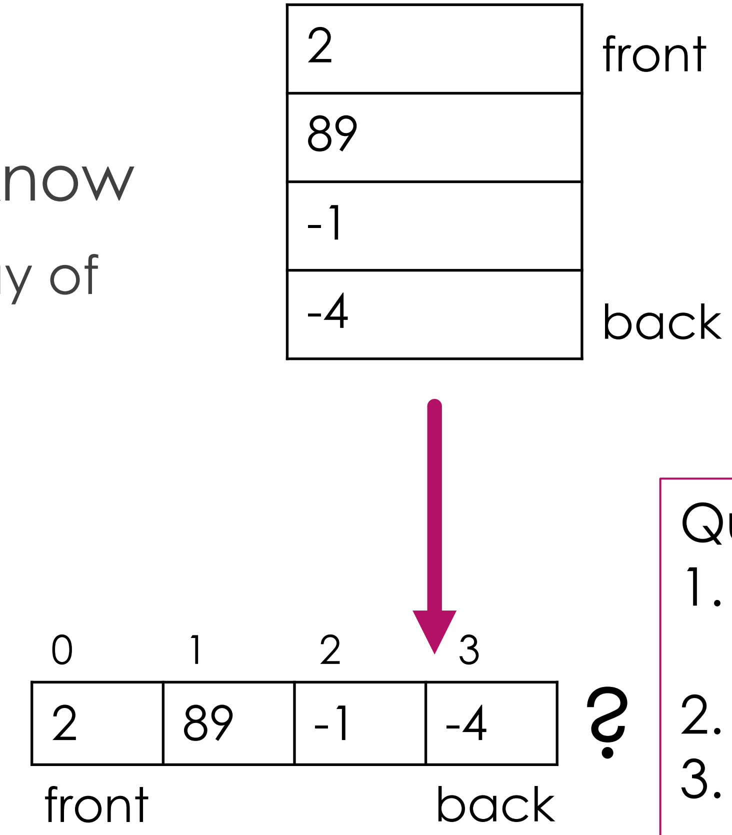
stack-array.rkt

- In DrRacket
- Code on Canvas under “Materials”

- Run the code and inside the console (lower portion of screen):
  - `let sa = StackArray[int](4)`
  - `sa.push(5)`
  - `sa.pop()`
  - …and keep trying other functions

How can we implement a queue?

- Using a data structure we know
  - A vector/array of fixed size



Questions we need to answer:

1. How can we enable adding of new elements?
2. Where can we add a new element?
3. Where should we remove an element from?

Implementation steps

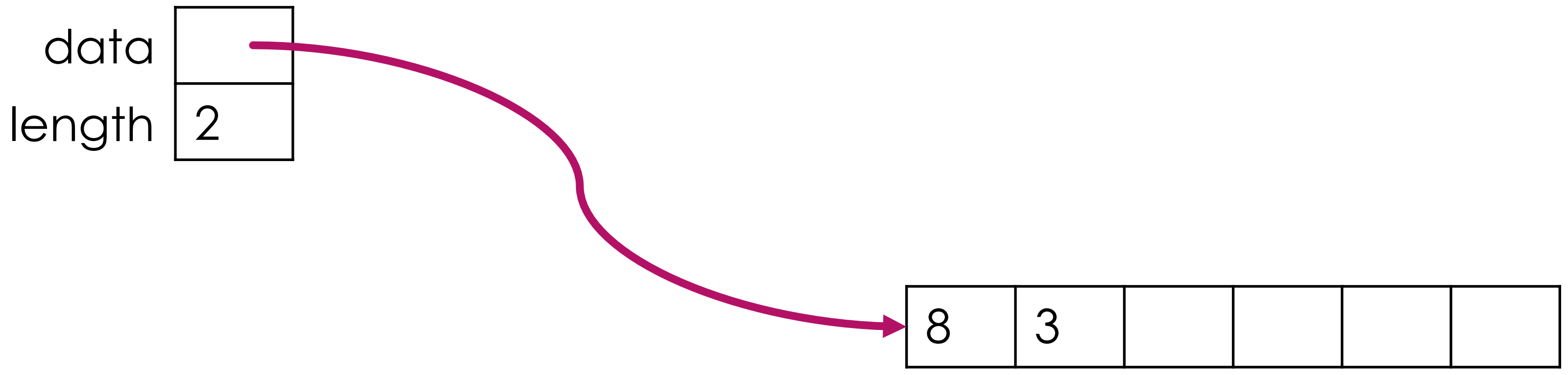
1. A concrete data representation of the stack or queue using array
2. Function definitions for interface functions while satisfying laws
3. A representation for each item in the stack/queue

Step 1: Representation of data

- What information do we need to keep track of a queue array?
  - An array of some sufficiently large capacity
  - Length field

- Attempt: same information as stack

Attempt at Step 1



- `length` tells us how many items are "in" the array
  - Not the size of the array
  - There may be empty unused spaces in the array

Dequeue operation

Before the operation

f b

~~23~~	90	0	1		

length = 4

Dequeue from front

After the operation

	90	0	1		

length = 3

Returned from operation 23

Dequeue again from front

try at idx 0

After the operation

	90	0	1		

length = 3

Dequeue operation

Before the operation

ngth = ______

What are the issue with this approach?

What else is needed?

After the operation

length = ______

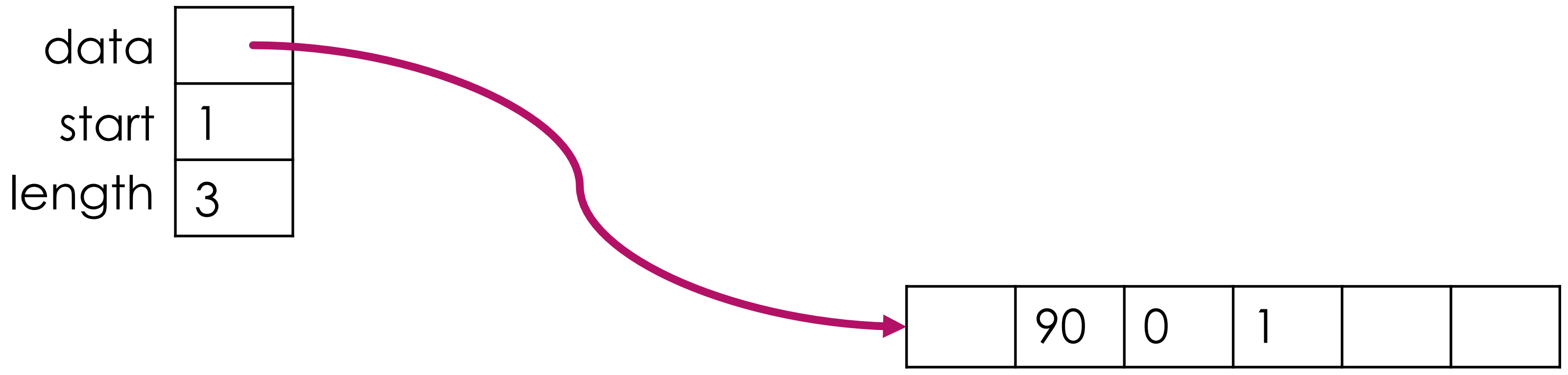
Returned from operation

Dequeue again from front

After the operation

length = ______

Step 1: Representation of data



Dequeue operation

Before the operation

~~23~~	90	0	1		

Dequeue from front

length = 4
start = 0

After the operation

	90	0	1		

Returned from operation 23

length = 3
start = 1

Dequeue again from front

After the operation

		0	1		

Return 90

length = 2
start = 2

Step 2: Operation implementation

- Operations implemented similarly to stack array
  - Enqueue: Add at `start + length`, increment `length`
  - Dequeue: Remove at `start`, increment `start`

- Consider queue: `start = 4, length = 4`

				2	-5	9	10

  - To enqueue, there isn't space at index 4+4=8 (out of bounds)
  - But lots of space in the array being wasted

Ring buffer implementation

- We can avoid wasted space with a ring buffer implementation
  - Data representation stays the same
  - Operations have slightly more complexity in implementation
- Treat array as a circle or ring
  - When space at end if over, circle back to beginning
  - Enqueue in next available spot

Step 2: Ring buffer implementation

- Enqueue into the queue below with `start = 4, length = 4`

8 9 10 11 12

				2	-5	9	10

- Enqueue instead at:

```
(start + length) % <array capacity>
(start + length) % data.len()
```

- After dequeue, set `start` to `(start + 1) % <array capacity>`

Pause

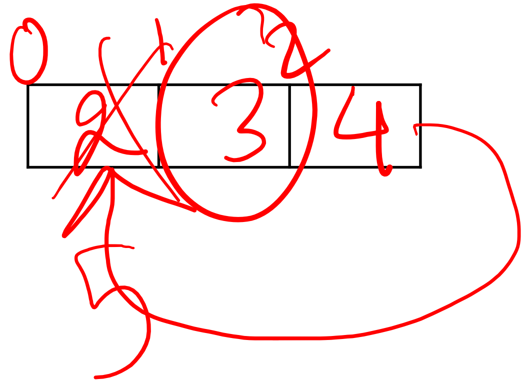
- Any questions or anything unclear?

Exercise: Try it out yourself

- Abstract queue variable **q**

	start	length
beginning	0	0
q.enqueue(2)	0	1
q.enqueue(3)	0	2
q.dequeue()	1	1
q.enqueue(4)	1	2
q.enqueue(5)	1	3
q.dequeue()	2	2

Ring buffer array



Are these the best we can do?

- Stack and queue capacities are limited
- We could create a new array each time we need to expand
  - There is a way to do this efficiently (we may see this later in the quarter)
  - But generally seems inefficient and time-consuming

- What about if we used linked lists instead?

If time: let’s play with a RingBuffer library

ring-buffer.rkt

- In DrRacket
- Code on Canvas under “Materials”