CPS 305

Data Structures

Prof. Alex Ufkes

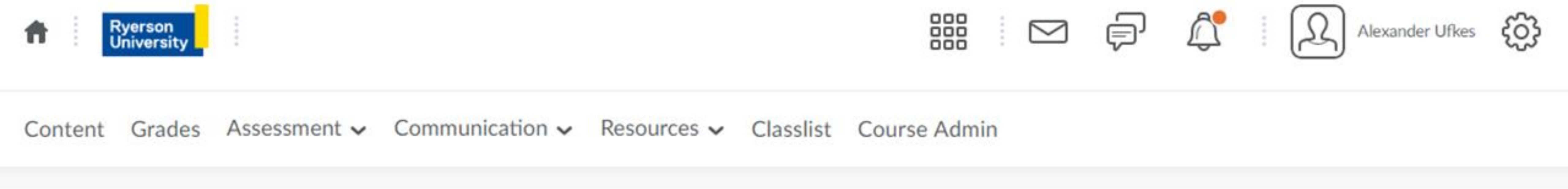
Topic 6: More ADTs – Stacks, Queues, Sets, & Variants

Toronto Metropolitan University

Notice!

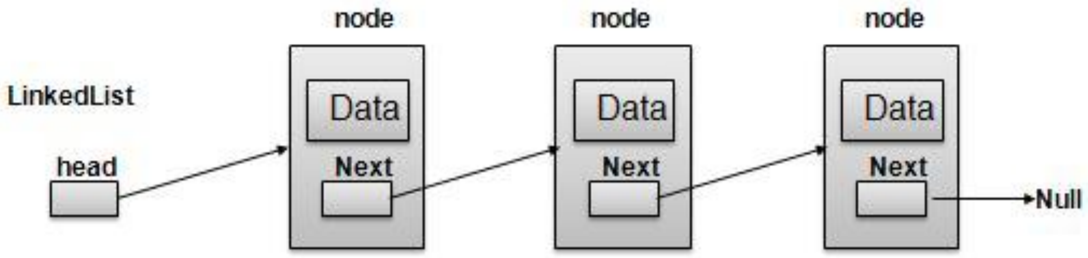
Obligatory copyright notice in the age of digital delivery and online classrooms:

Course Administration

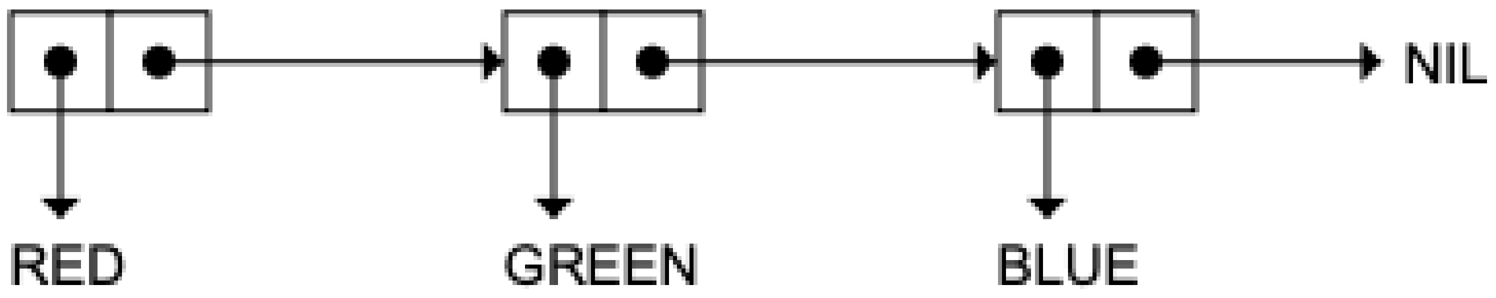


- Midterm next week! See D2L announcement.

Previously: Linked Lists



`(RED GREEN BLUE)`



Previously: Linked Lists in LISP

Use the length form to count elements:

```
* (length '(RED GREEN BLUE))
3
* (length '((BLUE SKY) (GREEN GRASS) (BROWN EARTH)))
3
*
* (length ())
0
* (length NIL)
0
```

Empty List?
- Use `()` or `NIL`

Previously: Accessing

`FIRST`, `SECOND`, `THIRD`, `REST`

Lisp's primitive functions for extracting elements from a list:

```
(first '(a b c d))    => a
(second '(a b c d))   => b
(third '(a b c d))    => c
```

REST returns a list containing everything ***but*** the first element:

```
(rest '(a b c d))             => (b c d)
(rest (rest '(a b c d)))      => (c d)
```

Previously: The CONStructor

CONS creates a cons cell:

- takes two arguments – an element, and a list
- Returns a pointer to a new cons cell whose CAR points to the first parameter and whose CDR points to the second

```
* (cons 'sink '(or swim))
(SINK OR SWIM)
* (cons 'sink ())
(SINK)
* (cons '(or swim) ())
((OR SWIM))
* (cons 'bond '(james bond))
(BOND JAMES BOND)
```

quicksort.lisp | make-list.lisp ✕

make-list.lisp > ...

```lisp
(defun mymake-list-rec (n element &optional (acc nil))
    (if (= n 0) acc ; If n is zero, we're done. Return acc.
        (mymake-list-rec (1- n) element (cons element acc))
    )
)

(defun mymake-list-it (n elem)
    (let ((acc nil))
    (dotimes (i n acc)
        (setf acc (cons elem acc)))
    )
)
```

```
* (load "make-list.lisp")
T
* (mymake-list-rec 3 'a)
(A A A)
* (mymake-list-it 7 'b)
(B B B B B B B)
```

Previously: Built-in Constructors

QUOTE, MAKE-LIST, LIST

```
> '("hello" world 111)
("hello" WORLD 111)
> (make-list 3)
(NIL NIL NIL)
> (make-list 3 :initial-element 'a)
(A A A)
> (make-list 3 :initial-contents '(a b c))
(A B C)
> (list "hello" 'world 111)
("hello" WORLD 111)
```

Previously: Traversal

```
* (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
* (dolist (x '(1 2 3)) (print x) (if (evenp x) (return "HI")))
1
2
"HI"
```

Can exit DOLIST early with return, just like other loops

Moving On…

Recall: LISP Structures

We can define a sequence of elements (structure) in LISP using **`defstruct`**:

Name of struct

Struct elements

```
(defstruct movie title director year type)
```

When you create a struct, LISP creates the following for you automatically:

- Constructor: **`MAKE-structureName`** to create instances
- Accessors: **`structureName-fieldname`** to set/get fields

Linked Lists: From Scratch in LISP

- Not every language has a built-in linked list type
- And if it does, perhaps we want more freedom to modify the implementation.

Basic structure:

- A *node* tuple stores reference to data item and reference to next item in the list.

```
(defstruct node data next)
```

Linked Lists: From Scratch in LISP

Basic structure:

- A *node* tuple stores reference to data item and reference to next item in the list.

```
(defstruct node data next)
```

- The list itself is also a tuple, containing a reference to head, and the size.

```
(defstruct my-list
  (head nil :type (or node null))
  (size 0 :type (integer 0)))
```

Linked Lists: From Scratch in LISP

```
(defstruct my-list
  (head nil :type (or node null))
  (size 0 :type (integer 0)))
```

Field Name: `head`
Initial Value: `nil`
Type Spec: `(or node null)`

- The *head* field can be of type *node* or *null*.
- null indicates that the field can also be nil
- This signifies the end of the list (or an empty list).

Linked Lists: From Scratch in LISP

```
(defstruct my-list
  (head nil :type (or node null))
  (size 0 :type (integer 0)))
```

Field Name: `size`
Initial Value: `0`
Type Spec: `(integer 0)`

- The size field must be a non-negative integer

Linked Lists: From Scratch in LISP

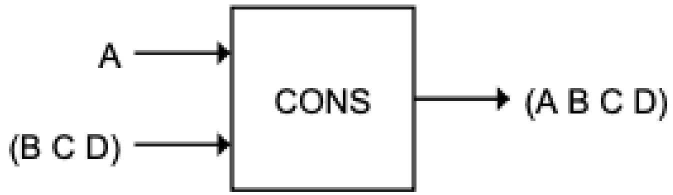
Keeping track of size:

- Makes length function O(1)
- Adds extra work every time a node is added/removed, must update size field
- Worth it? Maybe, maybe not.

Linked Lists: From Scratch in LISP

My CONStructor

(MY-CONS ITEM LIST)



- Should work like LISP's built-in cons
- ITEM is a **node**, LIST is a **`my-list`**

```
(defstruct node data next)

(defstruct my-list
  (head nil :type (or node null))
  (size 0 :type (integer 0)))
```

Linked Lists: From Scratch in LISP

```
(defun my-cons (data list)

  (let ((new-head (make-node :data data
                    :next (my-list-head list))))

    (make-my-list :head new-head
                  :size (1+ (my-list-size list)))
  )
)
```

Create new **node**, pointing to previous head of the list

Create (and return) new **`my-list`**, with head pointing to new-head, and size+1

```lisp
(defun my-cons (data list)
  (let ((new-head (make-node :data data
                  :next (my-list-head list))))
    (make-my-list :head new-head
                  :size (1+ (my-list-size list)))
) )
```

```
* (load "my-list.lisp")
T
* (defvar ll (my-cons 'A (make-my-list)))
LL
* LL
#S(MY-LIST :HEAD #S(NODE :DATA A :NEXT NIL) :SIZE 1)
* (setf ll (my-cons 'B LL))
#S(MY-LIST :HEAD #S(NODE :DATA B :NEXT #S(NODE :DATA A :NEXT NIL)) :SIZE 2)
```

Linked Lists: From Scratch in LISP

Accessors? **`my-first`** and **`my-rest`**

First a helper:

```
(defun is-empty (list)
    (equalp list (make-my-list)))
```

- Use EQUALP to compare structures. Can't use EQUAL

Linked Lists: From Scratch in LISP

Accessors? **`my-first`** and **`my-rest`**

```
(defun is-empty (list) (equalp list (make-my-list)))

(defun my-first (alist) (node-data (my-list-head alist)))
```

Access data field from that node

Access head field of my-list (returns a node)

Linked Lists: From Scratch in LISP

```
(defun is-empty (list) (equalp list (make-my-list)))

(defun my-first (alist) (node-data (my-list-head alist)))


(defun my-rest (alist)
  (if (is-empty alist) alist
    (make-my-list :head (node-next (my-list-head alist))
                  :size (1- (my-list-size alist))
    )
  )
)
```

If empty, return argument list

Else, return new list:

- Head is next node of previous head.
- Size is one less than old size.

Linked Lists: From Scratch in LISP

```
(defun my-first (alist) (node-data (my-list-head alist)))

(defun my-rest (alist)
  (if (is-empty alist) alist
    (make-my-list :head (node-next (my-list-head alist))
                  :size (1- (my-list-size alist))
) ) )
```

```
* ll
#S(MY-LIST :HEAD #S(NODE :DATA B :NEXT #S(NODE :DATA A :NEXT NIL)) :SIZE 2)
* (my-first ll)
B
* (my-rest ll)
#S(MY-LIST :HEAD #S(NODE :DATA A :NEXT NIL) :SIZE 1)
```

Linked Lists: From Scratch in LISP

Access n^{th} element?
Use **ELT**:

How about **`my-elt`**?

```
* (ELT '(A B C D E F) 0)
A
* (ELT '(A B C D E F) 1)
B
* (ELT '(A B C D E F) 2)
C
* (ELT '(A B C D E F) 3)
D
* (ELT '(A B C D E F) 4)
E
* (ELT '(A B C D E F) 5)
F
```

Linked Lists: From Scratch in LISP

```lisp
(defun my-elt (list index)
  (dotimes (i (my-list-size list))
    (if (= i index) (return (node-data (my-list-head list))))
      (setf list (my-rest list))
    )
  )
)
```

```
* ll
  #S(MY-LIST :HEAD #S(NODE :DATA B :NEXT
  #S(NODE :DATA A :NEXT NIL)) :SIZE 2)
* (my-elt ll 0)
B
* (my-elt ll 1)
A
*
```

Linked Lists: From Scratch in LISP

Searching? Can do both iterative and recursive:

```
(defun my-search-iter (item alist)
  (dotimes (i (my-list-size alist))
    (if (equalp item (my-first alist))
      (return item)
    (setf alist (my-rest alist)))
  )
)
```

Returns the item if found, returns NIL if not

Linked Lists: From Scratch in LISP

Searching? Can do both iterative and recursive:

```
(defun my-search-rec (item alist)
  (cond ((is-empty alist) NIL)
        ((equalp item (my-first alist)) item)
        (T (my-search-rec item (my-rest alist)))
  )
)
```

```
(defun my-search-iter (item alist)
    (dotimes (i (my-list-size alist))
        (if (equalp item (my-first alist)) (return item))
    (setf alist (my-rest alist))))
)

(defun my-search-rec (item alist)
    (cond ((is-empty alist) NIL)
          ((equalp item (my-first alist)) item)
          (T (my-search-rec item (my-rest alist)))
    )
)
```

```
* (my-search-iter 'A ll)
A
* (my-search-iter 'F ll)
NIL
* (my-search-iter 'B ll)
B
* (my-search-rec 'B ll)
B
* (my-search-rec 'A ll)
A
* (my-search-rec 'H ll)
NIL
```

Practice problem:

- Modify both of these to return the index, rather than the item.
- Similar to indexOf() in Python.

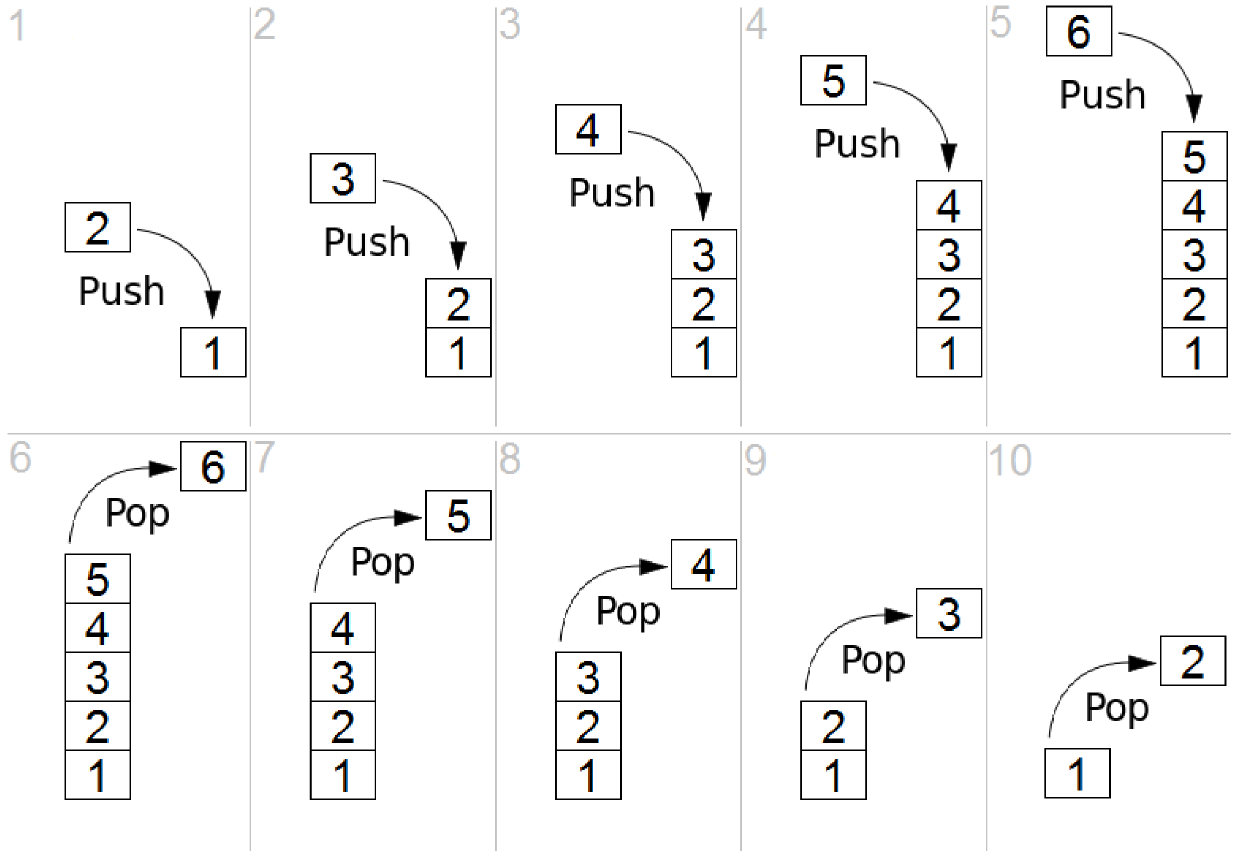
Array & Linked ADTs

- We've got LISP's list type
- We even built our own!
- We've got contiguous arrays
- Now, we can build some ADTs using both arrays and linked lists.

Stacks Queues Deques Sets

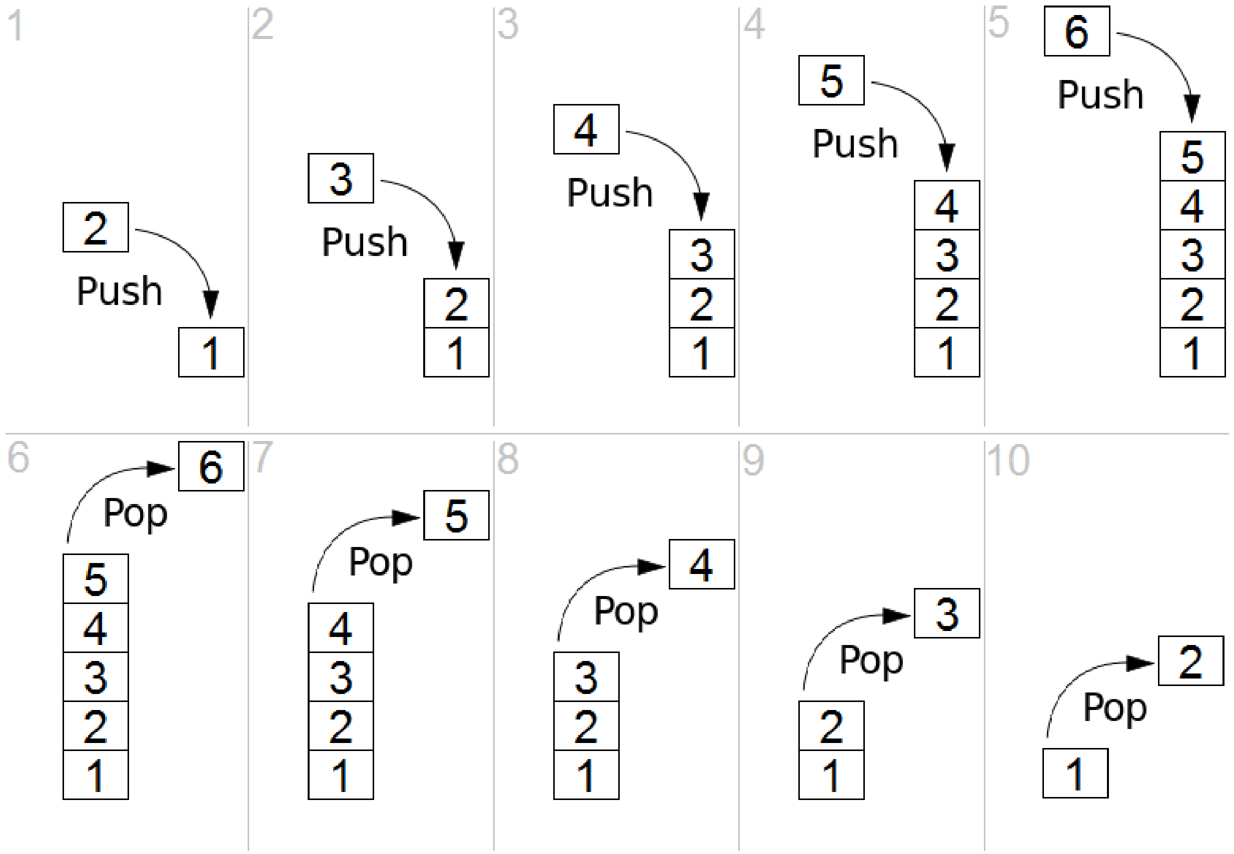
Stacks

Stack



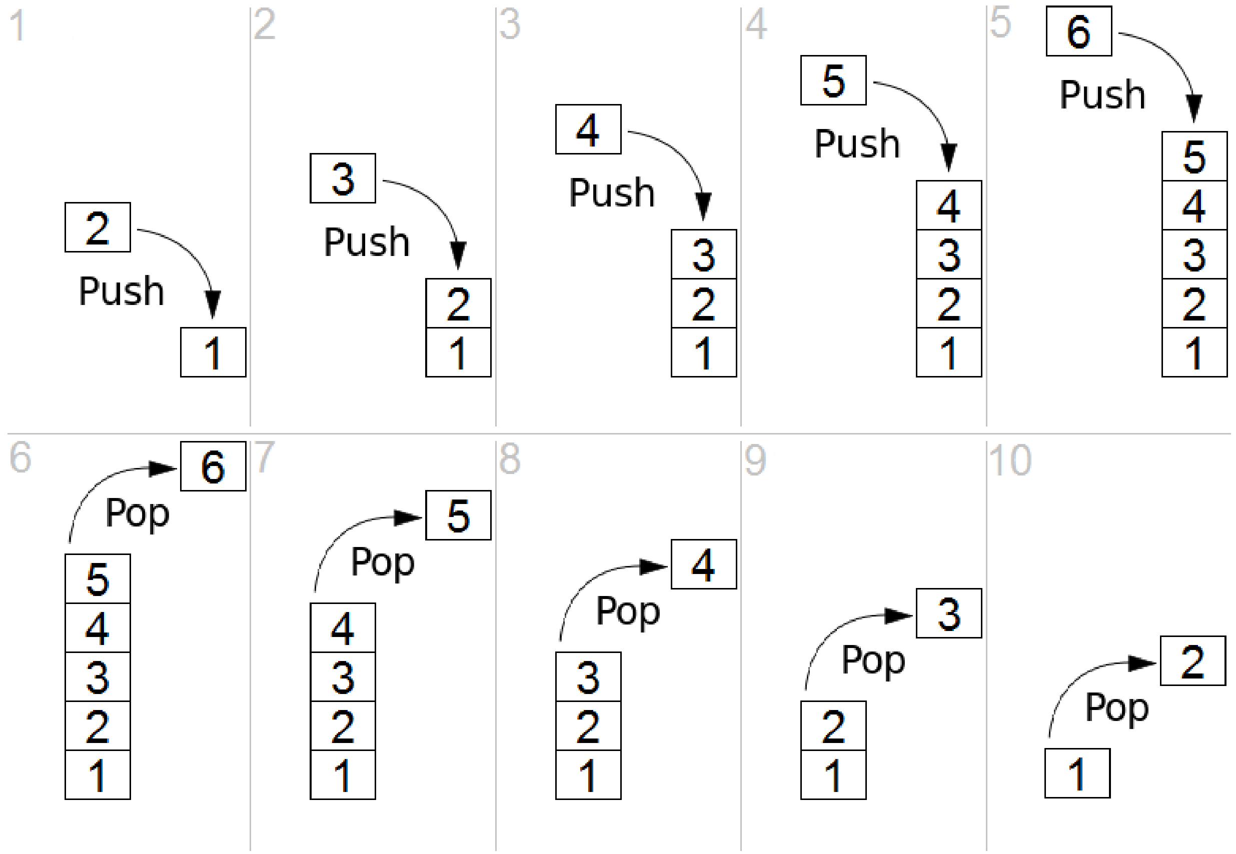
- Two operations, push() and pop().
- Optional 3rd, peek(), doesn't modify the stack
- LIFO – Last In, First Out

Stack



- Easy to implement with linked list or array.
- **Push:** add to the "top" of the list.
- **Pop:** remove from the "top" of the list
- Both are O(1)

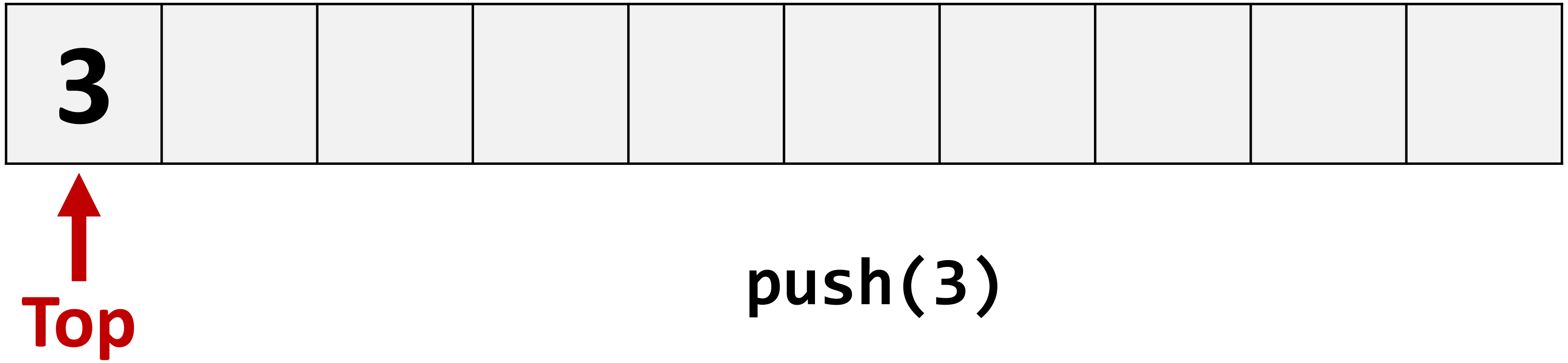
Stack: Applications?



- *Undo* button in a text editor
- Navigating backwards in a web browser (*back* button)
- Function call stack for an executing program
- (This is where the term “stack overflow” comes from)

Stack: Using an Array

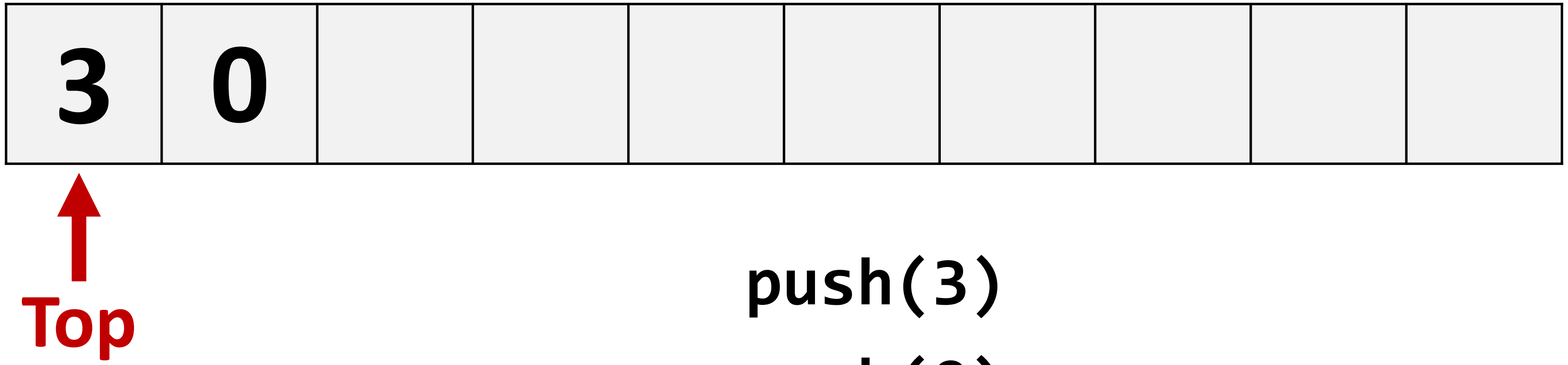
Maintain index at top of stack (back)



`push(3)`

Stack: Using an Array

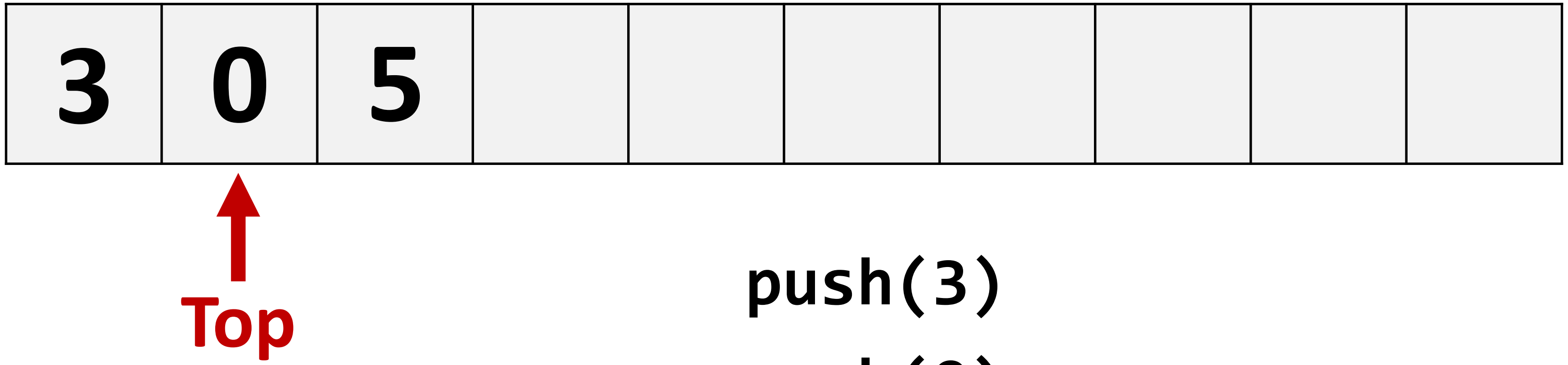
Maintain index at top of stack (back)




```
push(3)
push(0)
```

Stack: Using an Array

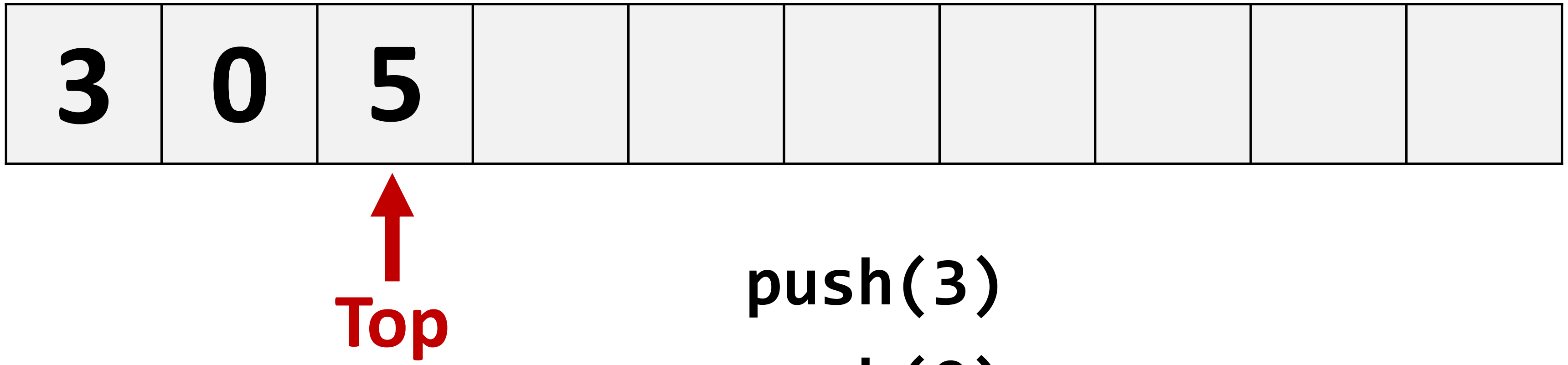
Maintain index at top of stack (back)




```
push(3)
push(0)
push(5)
```

Stack: Using an Array

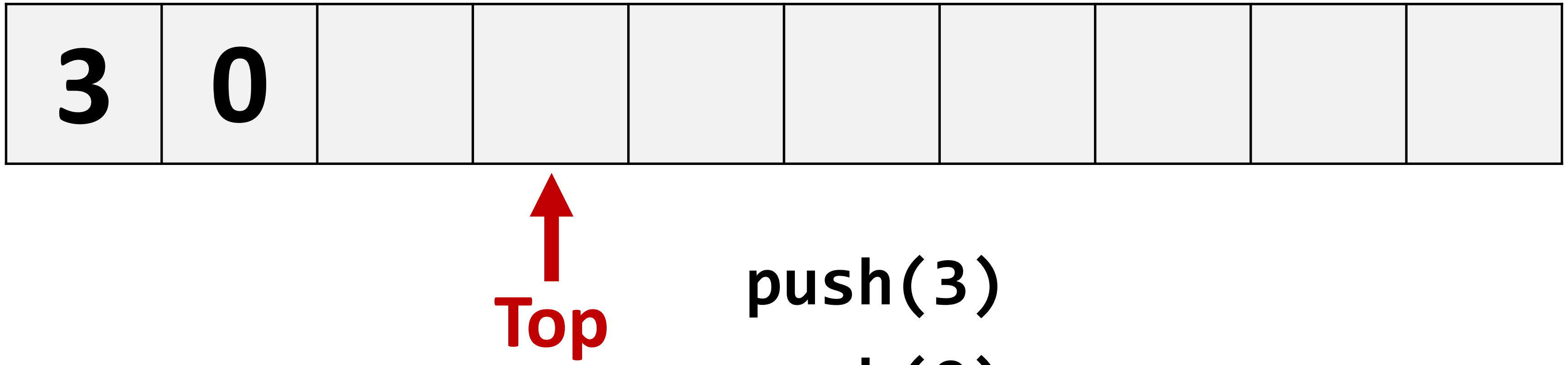
Maintain index at top of stack (back)




```
push(3)
push(0)
push(5)
pop()
```

Stack: Using an Array

Maintain index at top of stack (back)




```
push(3)
push(0)
push(5)
pop()
```

Stack: Using an Array

Maintain index at top of stack (back)

3	0	1	2	3	4	5	6	7	8

Top

```
push(3)    push(1)
push(0)       ⋮
push(5)
pop()      push(9)
```

?

Stack: Using an Array

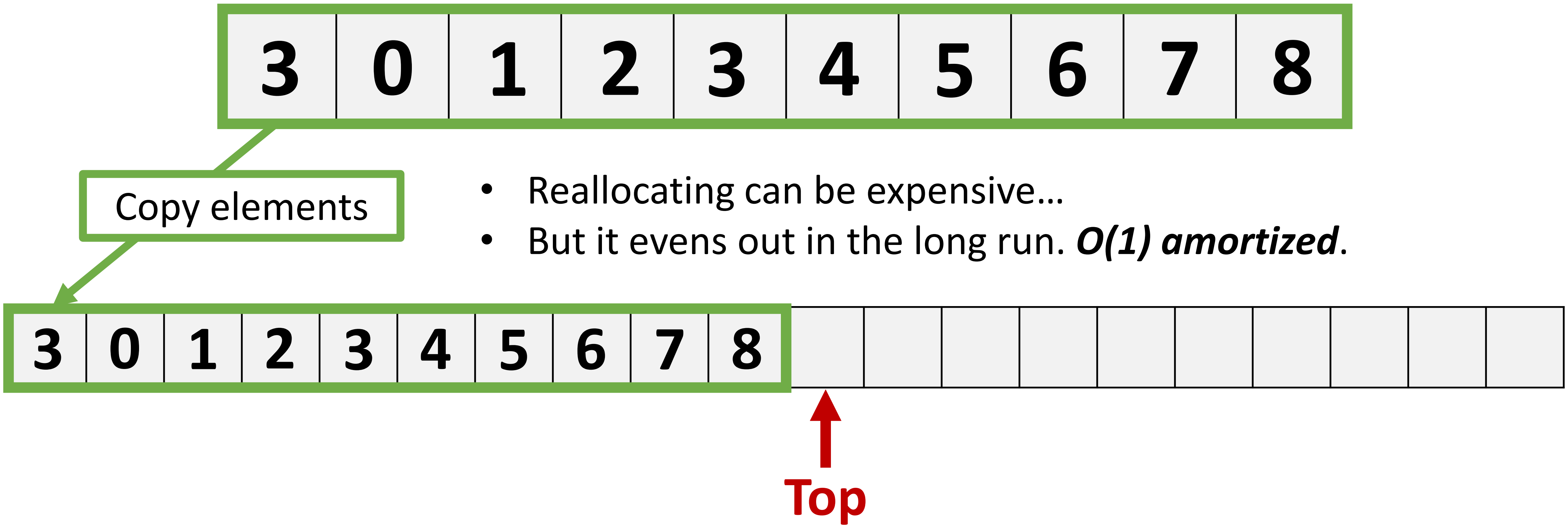
Maintain index at top of stack (back)

3	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

When using an array implementation, common practice is to double the array size upon reaching capacity.

Stack: Using an Array

Maintain index at top of stack (back)



- Reallocating can be expensive…
- But it evens out in the long run. ***O(1) amortized***.

Stack: Using an Array

O(1) Amortized?

- Reallocating is expensive. But…
- In practice, if we double the size each time, it'll be twice as long until we must resize again.
- Mitigate this by allocating a large(r) array initially.
- Appending is *amortized* O(1)
    - Time for n insertions is O(n)
    - Earlier insertions might be slower, if we're resizing.
- LISP does this! Remember the dynamic vector!

Set initial size to zero

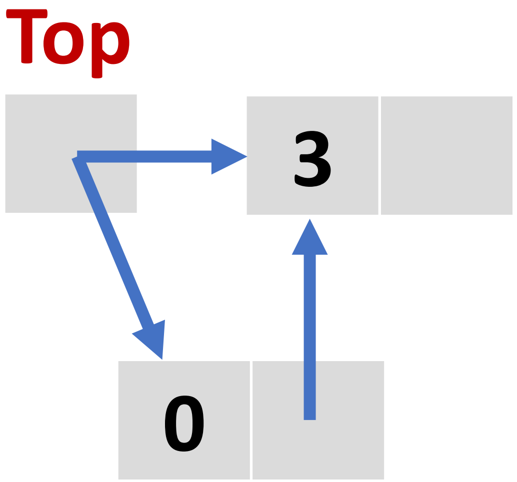
```lisp
(let ((vec (make-array 0 :fill-pointer t :adjustable t)))
    (dotimes (i 10)
        (vector-push-extend i vec)
        (describe vec)
    )
)
```

```
PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp e
--script vector.lisp
#(0)
  [vector]

Element-type: T
Fill-pointer: 1
Size: 1
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 1) {2355BFB7}>
#(0 1)
  [vector]

Element-type: T
Fill-pointer: 2
Size: 2
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 2) {235F2FDF}>
#(0 1 2)
  [vector]

Element-type: T
Fill-pointer: 3
Size: 4
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 4) {235F6987}>
#(0 1 2 3)
  [vector]

Element-type: T
Fill-pointer: 4
Size: 4
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 4) {235F6987}>
#(0 1 2 3 4)
```

```
Element-type: T
Fill-pointer: 6
Size: 8
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 8) {235FAC37}>
#(0 1 2 3 4 5 6)
  [vector]

Element-type: T
Fill-pointer: 7
Size: 8
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 8) {235FAC37}>
#(0 1 2 3 4 5 6 7)
  [vector]

Element-type: T
Fill-pointer: 8
Size: 8
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 8) {235FAC37}>
#(0 1 2 3 4 5 6 7 8)
  [vector]

Element-type: T
Fill-pointer: 9
Size: 16
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 16) {23603087}>
#(0 1 2 3 4 5 6 7 8 9)
  [vector]

Element-type: T
Fill-pointer: 10
Size: 16
Adjustable: yes
Displaced: no
Storage vector: #<(SIMPLE-VECTOR 16) {23603087}>
```

Stack: Using a Linked List

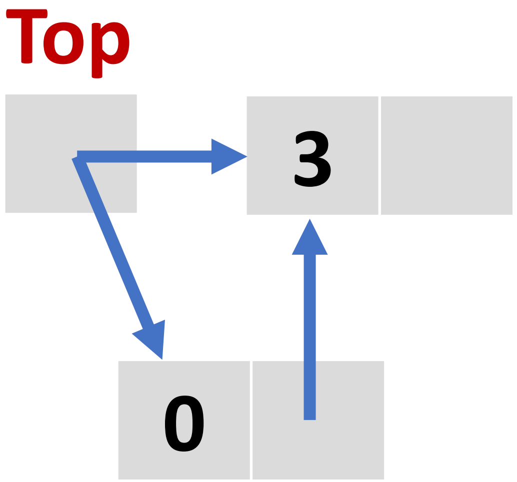
Let the front of the list represent the top of the stack:




```
push(3)
push(0)
```

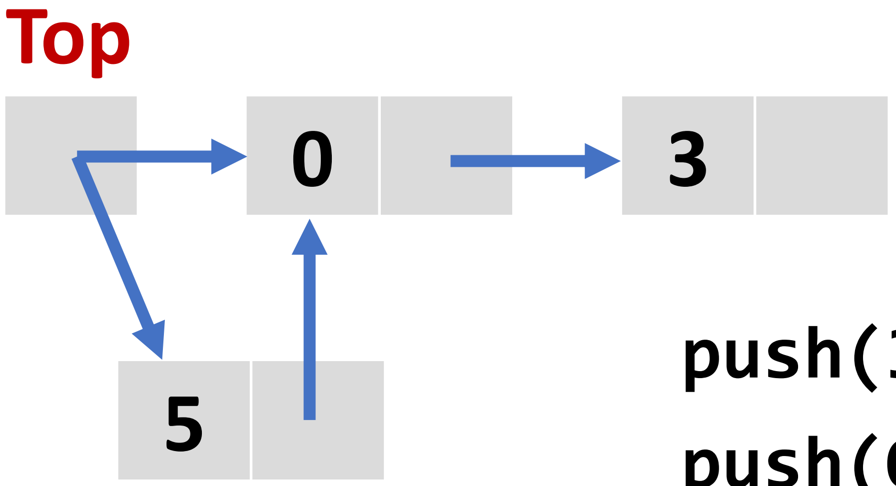
Stack: Using a Linked List

Let the front of the list represent the top of the stack:



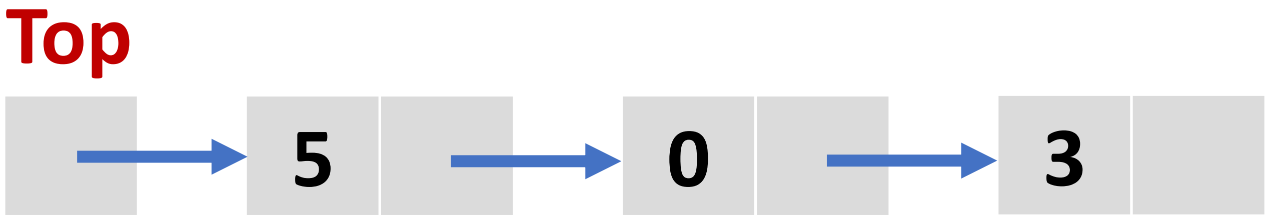

```
push(3)
push(0)
```

Stack: Using a Linked List



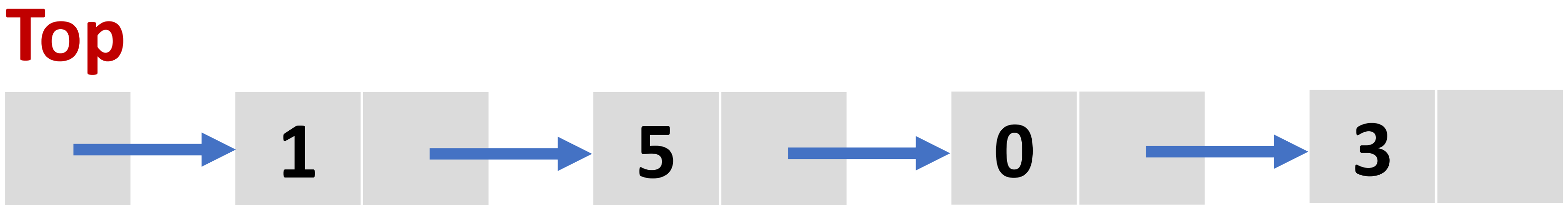

```
push(3)
push(0)
push(5)
```

Stack: Using a Linked List



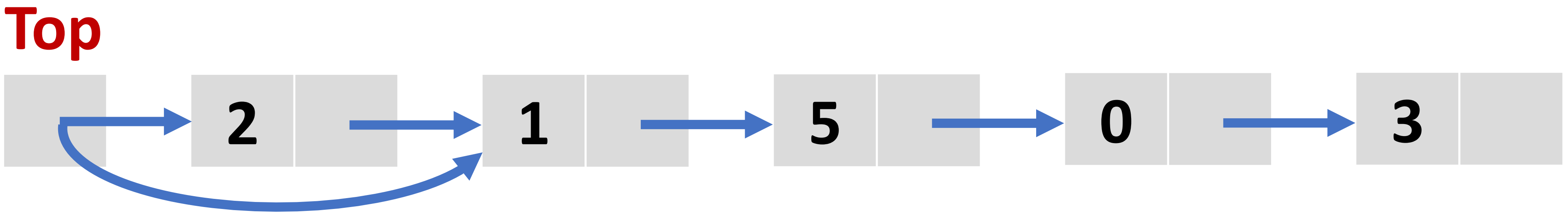

```
push(3)    push(1)
push(0)
push(5)
```

Stack: Using a Linked List




```
push(3)     push(1)
push(0)     push(2)
push(5)
```

Stack: Using a Linked List




```
push(3)     push(1)
push(0)     push(2)
push(5)     pop()
```

… And so on, and so forth.

Stack: Array VS Linked List

Both achieve O(1) for all operations!
- push(), pop(), and peek()
- What about memory use?

Array:
- Less space per element, but…
- Growing (and shrinking) can cause spikes
- Good cache locality

Linked List:
- Elements are allocated and freed independent of others
- No need to "move" any cells once they're placed

Stack: Implementation?

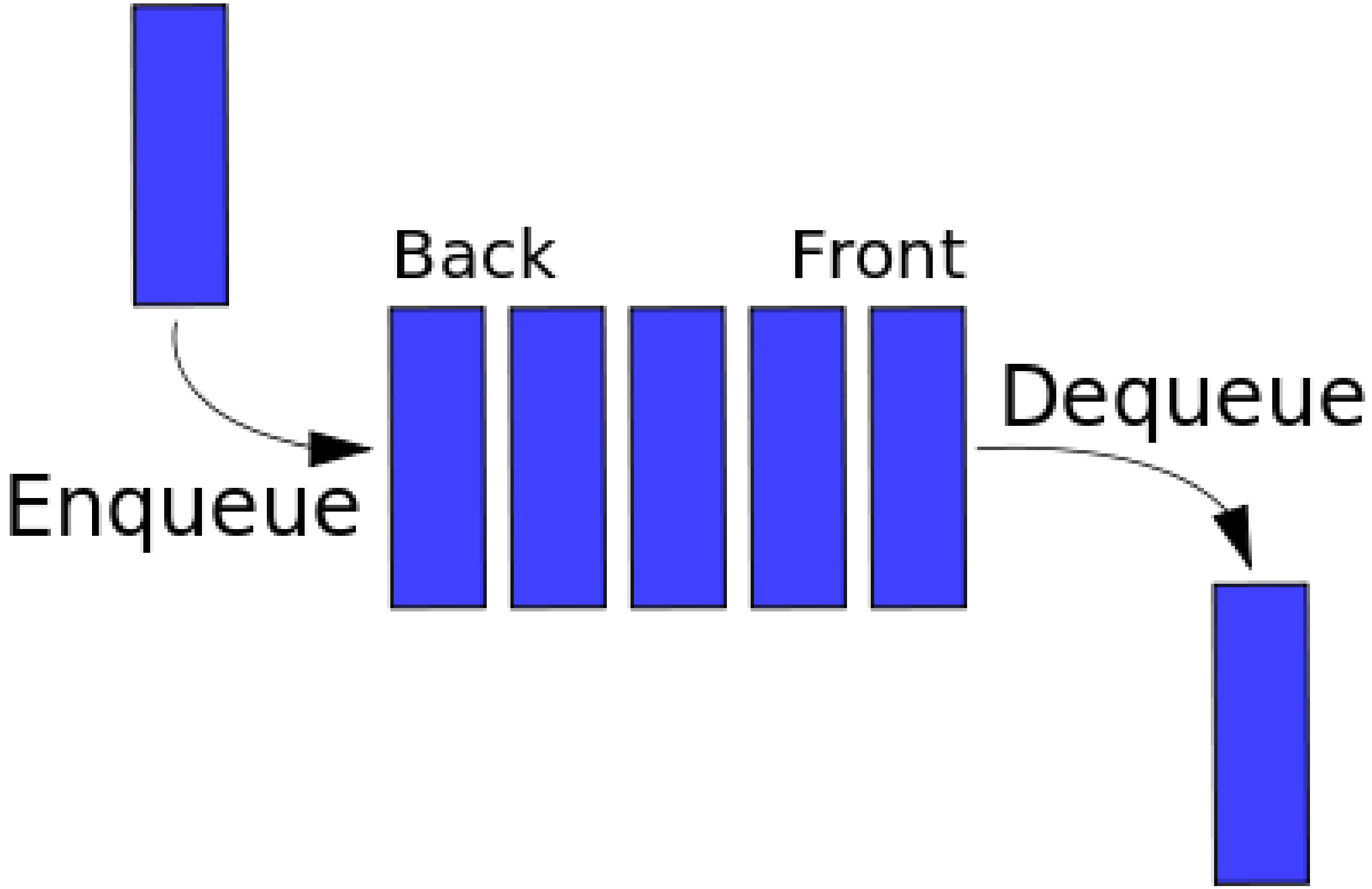
LISP has **PUSH** and **POP** functions built in:

Notice: PUSH and POP both mutate the list.

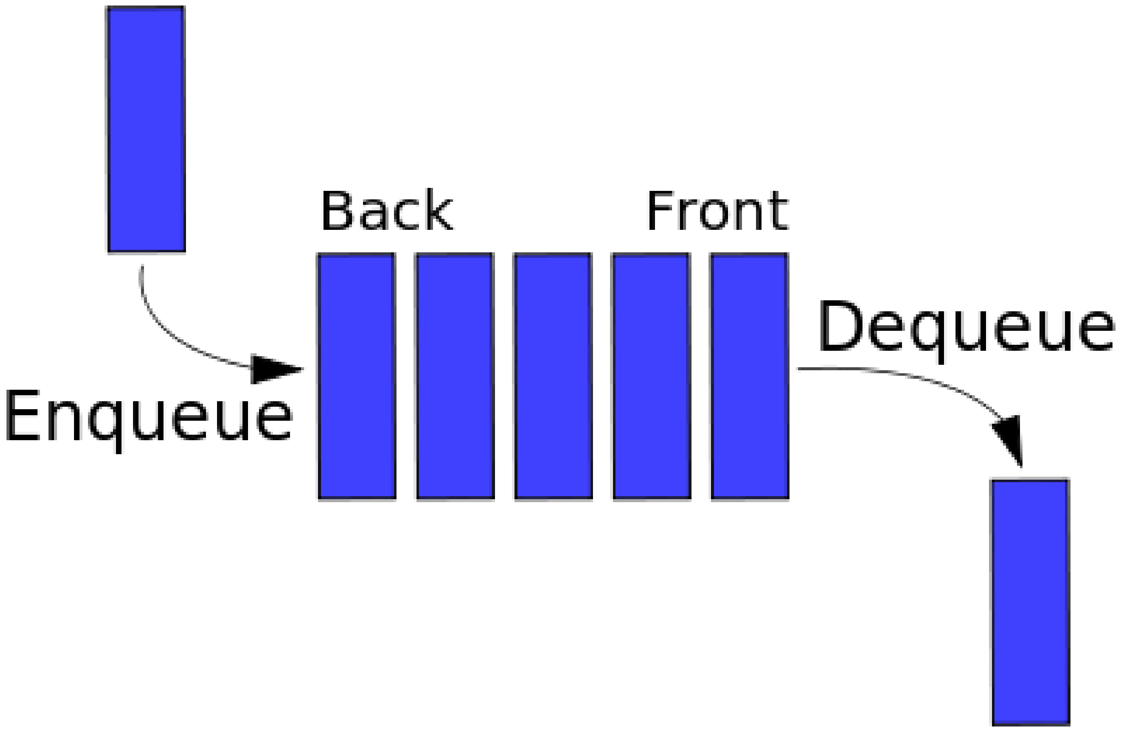
```
* (defvar *stack* '())
*STACK*
* (push 21 *stack*)
(21)
* (push 42 *stack*)
(42 21)
* (push 99 *stack*)
(99 42 21)
* (pop *stack*)
99
* *stack*
(42 21)
*
```

Queues

Queue



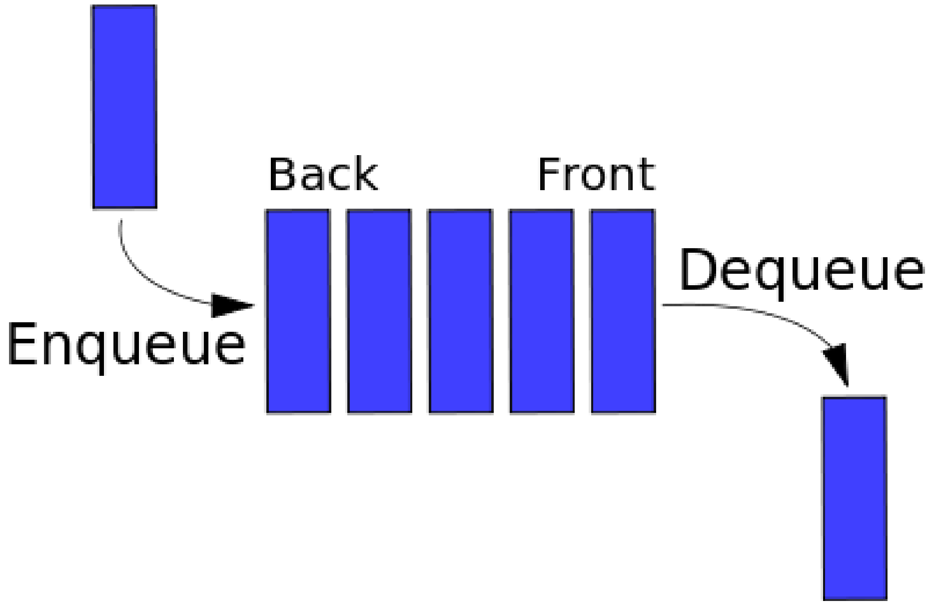
Queue



- Two operations, **enqueue** and **dequeue**.
- Optional 3rd operation, **peek**, doesn't modify the queue
- FIFO – First In, First Out
- Linked list provides efficient implementation
- O(1) enqueue and dequeue

Queue

Linked list provides efficient implementation

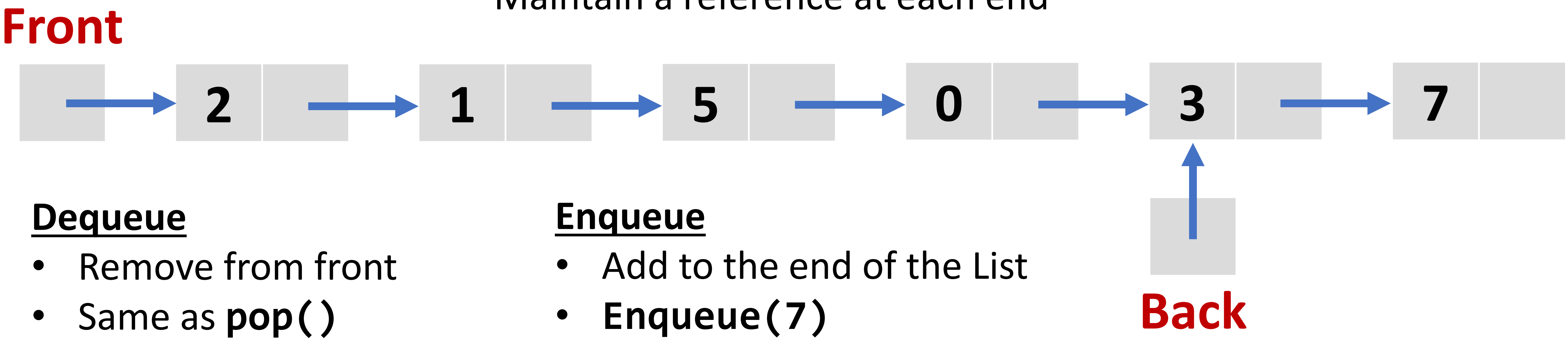


<u>We understand Linked Lists:</u>

- Enqueue is add to back
- Dequeue is remove from front
- Both O(1) – provided we maintain a references to front and back.

Queue: Linked List

Maintain a reference at each end



Dequeue

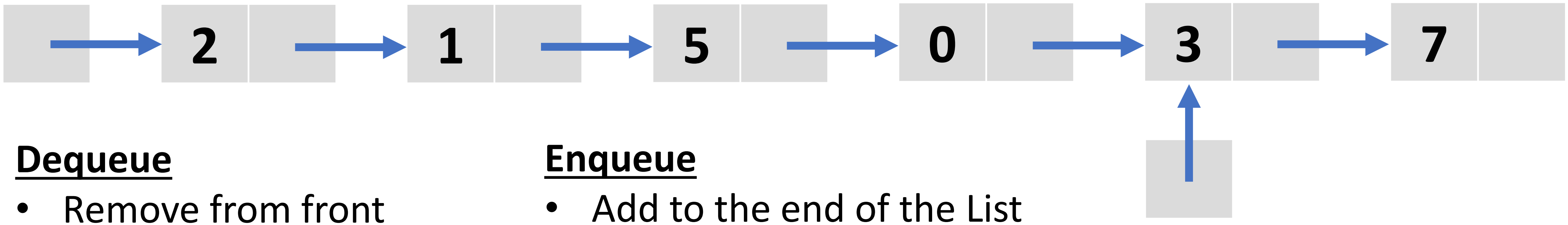
- Remove from front
- Same as **`pop()`**
- No need to see it again.

Enqueue

- Add to the end of the List
- **`Enqueue(7)`**
  - Update back.next
  - Update back

Queue: Linked List

Maintain a reference at each end



Dequeue

- Remove from front
- Same as **`pop()`**
- No need to see it again.

Enqueue

- Add to the end of the List
- **`Enqueue(7)`**
  - Update back.next
  - Update back

Queue: Implementation

```
(defstruct node data next)

(defstruct my-list
  (front nil :type (or node null))
  (back  nil :type (or node null))
  (size 0 :type (integer 0)))
```

Same implementation as our linked list, but now we have a reference for front and back

Queue: Implementation

```
(defstruct node data next)

(defstruct my-list
  (front nil :type (or node null))
  (back  nil :type (or node null))
  (size 0 :type (integer 0)))
```

Dequeue?

- Combination of first and rest.
- Remove first and return it.
- Update front reference.

Enqueue?

- Create new node, append it to the back of the list.
- Update back reference.

<u>You'll implement these in Lab 3</u>

Queue: Using Arrays?

Not terribly effective:

i	0	1	2	3	4	5	6	7	8	9	
a[i]	9	4	7	12	14	19	0	1	2	3	4?

- Like a stack, we have to grow array when full
- We have the additional problem of having to shift elements upon removal.

Queue: Using Arrays?

Shift elements upon removal?

i	0	1	2	3	4	5	6	7	8	9
a[i]				12	14	19	0	1	2	3

Dequeue()

Dequeue()

Dequeue()

- We would have to periodically shift elements back down.
- Leaving leading empty space while growing the array is wasteful.

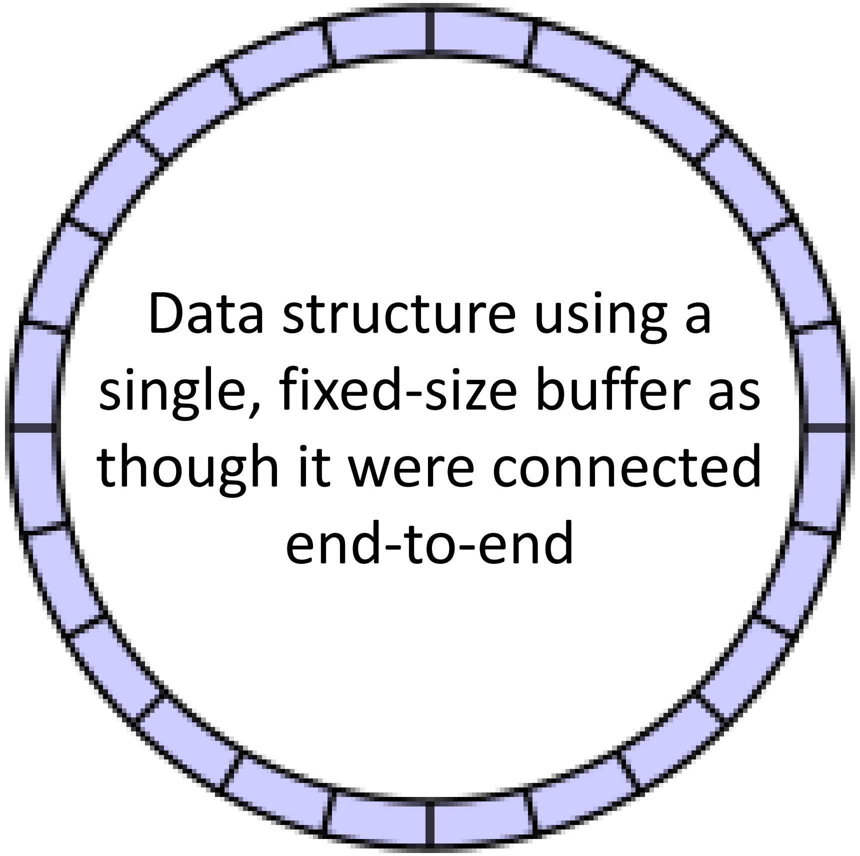
Queue: Using Arrays?

An array, when considered as a *linear* sequence of elements, is not a great option for a queue

However! With clever indexing, we can treat an array as a *circular buffer*, and avoid the trouble of shifting elements.

Circular Buffer

Or circular queue, cyclic buffer, ring buffer…



Data structure using a single, fixed-size buffer as though it were connected end-to-end

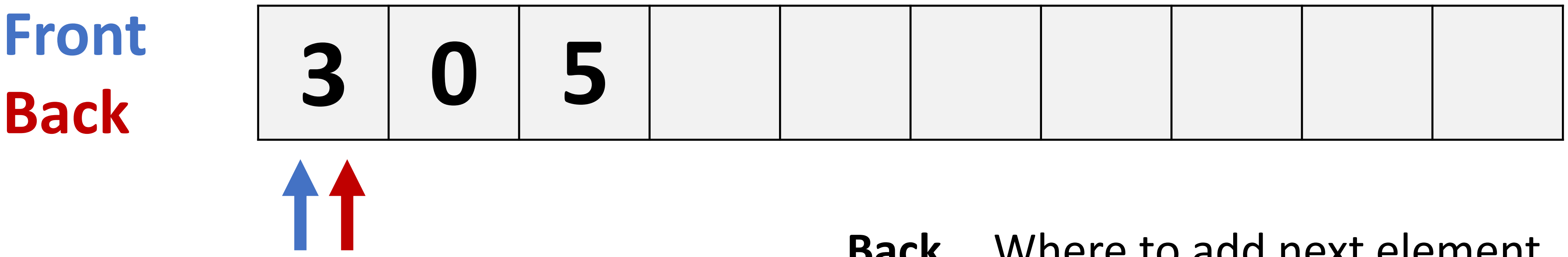
Circular Buffer

Very efficient data structure for implementing Queues that have a max size:

- Allocate an array with **`max_size`** elements
- Maintain two indexes – **`front`** and **`back`**
  - Insert at back, remove at front.
- When back reaches end of buffer, it wraps around to the start.
- Eliminates the need for shifting elements!

Circular Buffer: Using a Queue

Max size is 10 elements



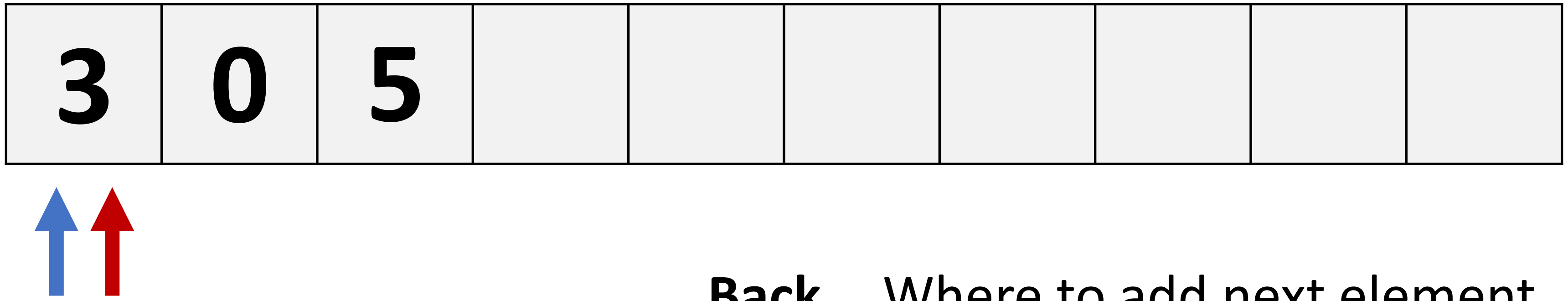
Back Where to add next element.
Front Next element to be removed.

Enqueue 3, 0, 5

Circular Buffer: Using a Queue

Max size is 10 elements

Front
Back



Back Where to add next element.
Front Next element to be removed.

Enqueue 3, 0, 5

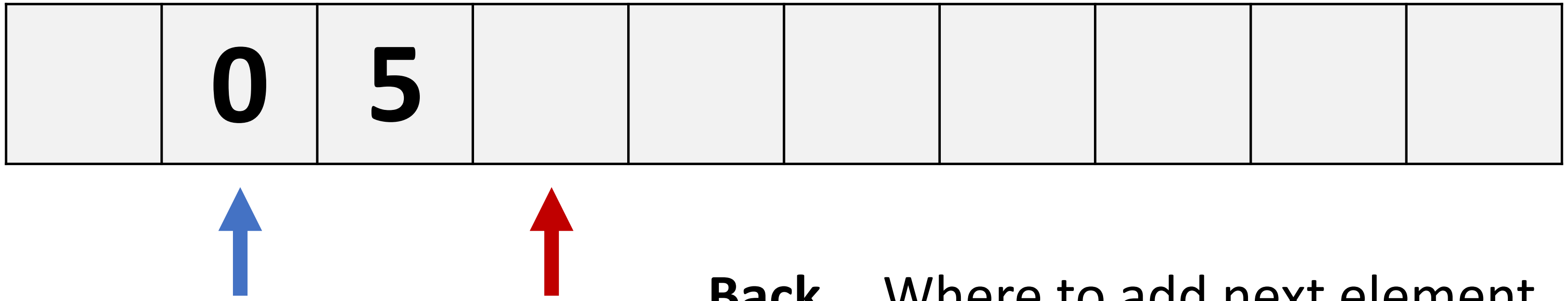
Dequeue()

Dequeue()

Circular Buffer: Using a Queue

Max size is 10 elements

Front
Back



Back Where to add next element.
Front Next element to be removed.

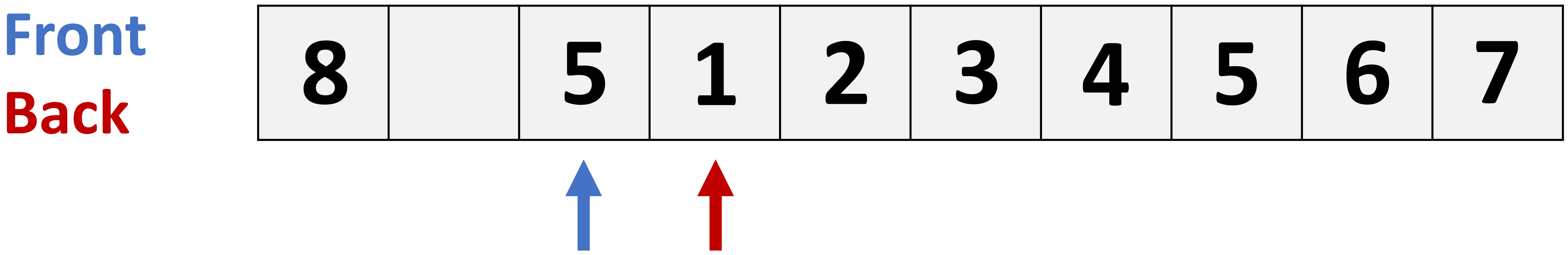
Enqueue 3, 0, 5

Dequeue()

Dequeue()

Circular Buffer: Using a Queue

Max size is 10 elements



Enqueue 1, 2, 3, 4, 5, 6, 7, 8

Circular Buffer: Using a Queue

Max size is 10 elements

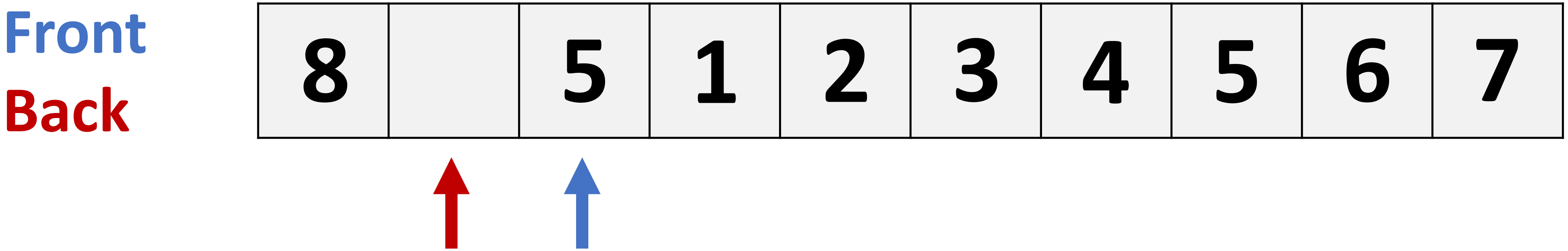
Front

Back

8		5	1	2	3	4	5	6	7

Enqueue 1, 2, 3, 4, 5, 6, 7, 8

Circular Buffer: Using a Queue



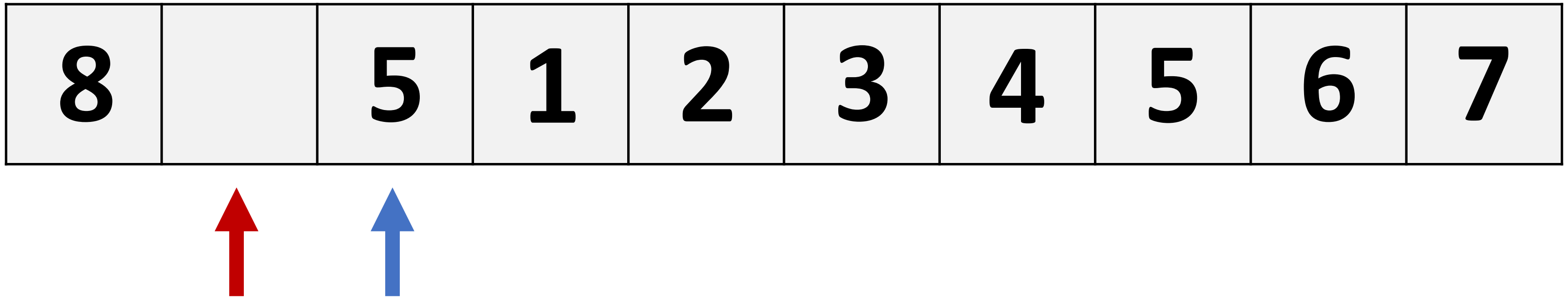
Memory addressing is linear! How do we update front and back efficiently?

Upon enqueue: `back = (back + 1) % max_size`
Upon dequeue: `front = (front + 1) % max_size`

If there's an element at **`back`**, buffer is full.
If there's no element at **`front`**, buffer is empty.

Circular Buffer: Using a Queue

Front
Back



We can determine the number of elements:

```
count = (back – front) % max_size
```

Circular Buffer: Using a Queue

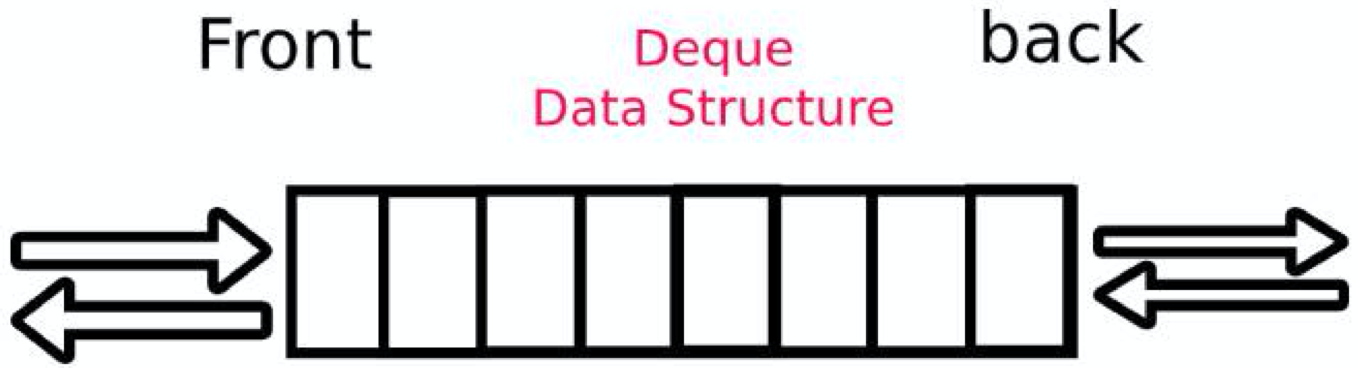
8		5	1	2	3	4	5	6	7

- Every queue operation is constant time.
- Array means good cache locality, less memory per node
- However, if size is *not* fixed, a linked list might be preferable
- Of course, we can also grow the circular buffer if we want.

Deque: Double-ended Queue

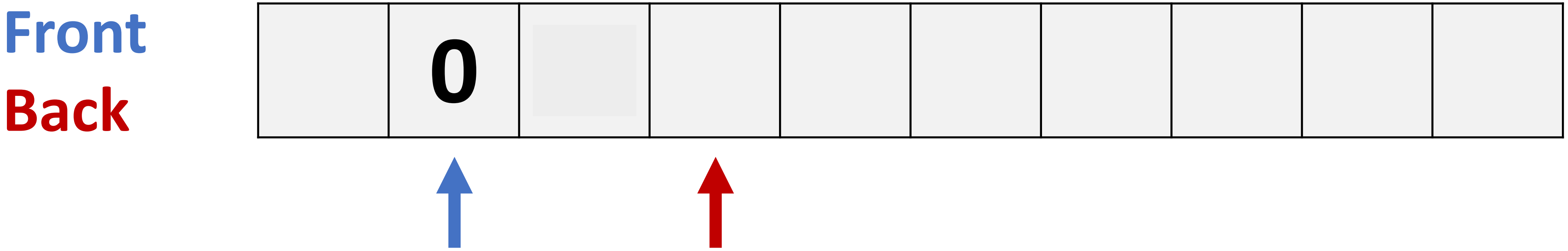
Implements all previously seen operations:

- Add to front
- Remove from front
- Add to back
- Remove from back



Deque: Using Circular Buffer

- Works much the same way as a standard Queue.
- Need to be a bit more creative with indexing

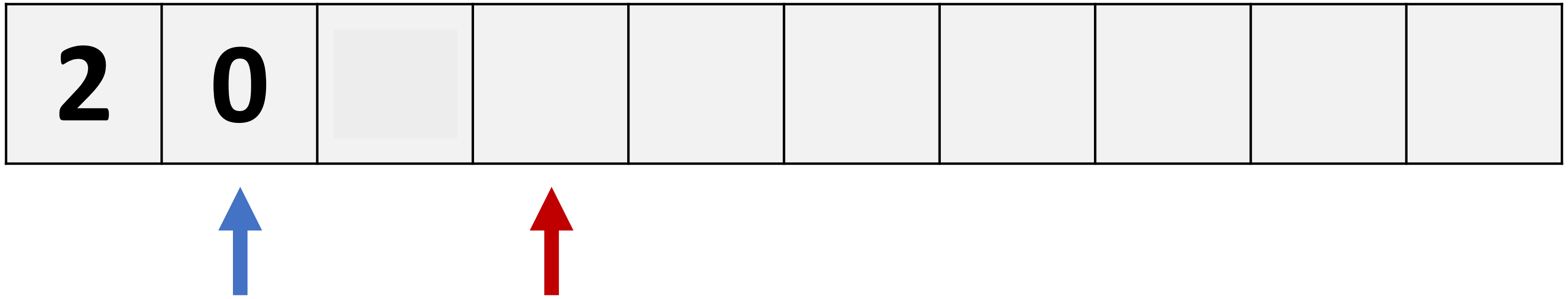


Upon removeBack: `back = (back - 1) % max_size`

Deque: Using Circular Buffer

- Works much the same way as a standard Queue.
- Need to be a bit more creative with indexing

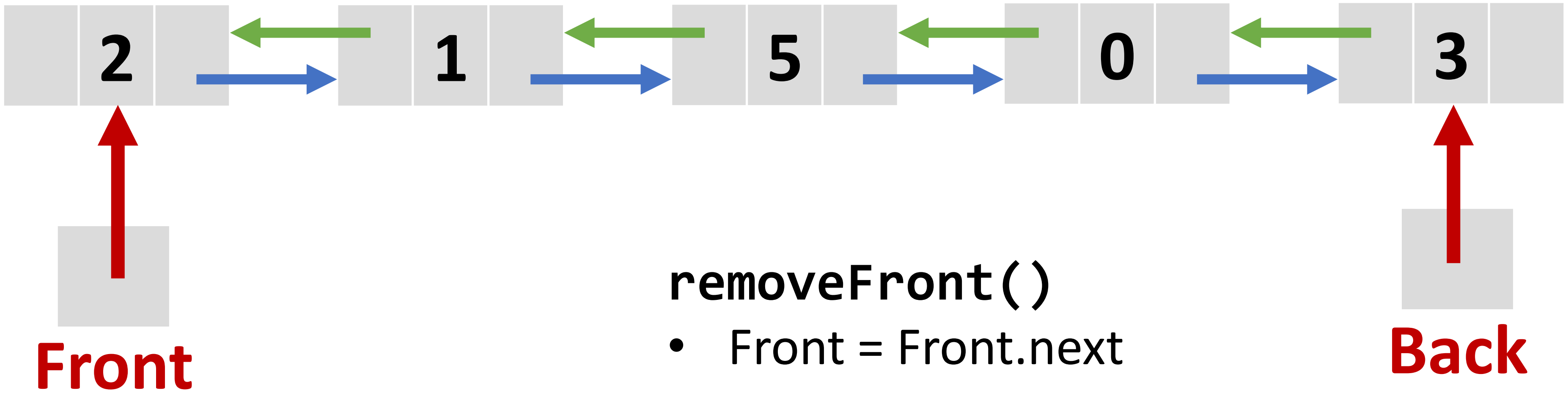
Front
Back



Upon removeBack: `back = (back - 1) % max_size`
Upon addFront: `front = (front - 1) % max_size`

Deque: Linked List

Use a ***doubly*** linked list!

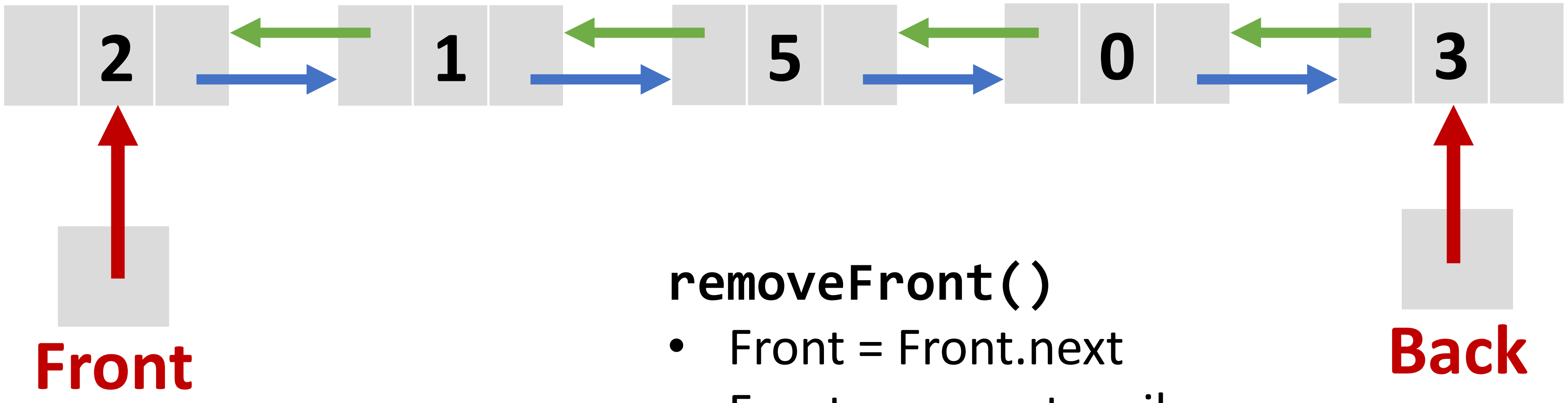


`removeFront()`

- Front = Front.next

Deque: Linked List

Use a ***doubly*** linked list!



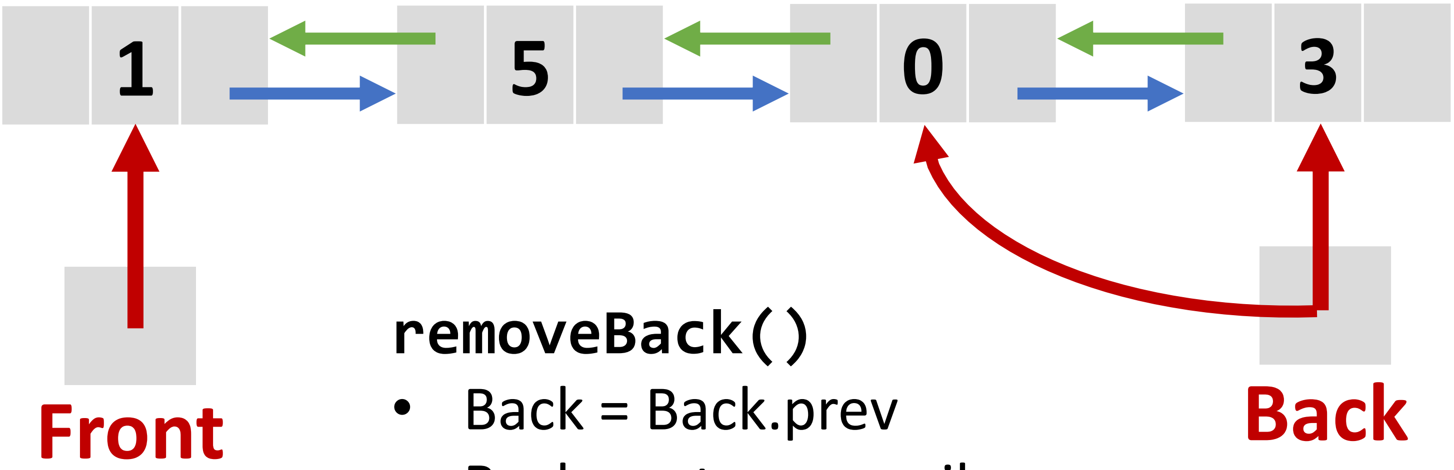
`removeFront()`

- Front = Front.next
- Front.prev.next = nil
- Front.prev = nil

`removeBack()` ?

Deque: Linked List

Use a ***doubly*** linked list!



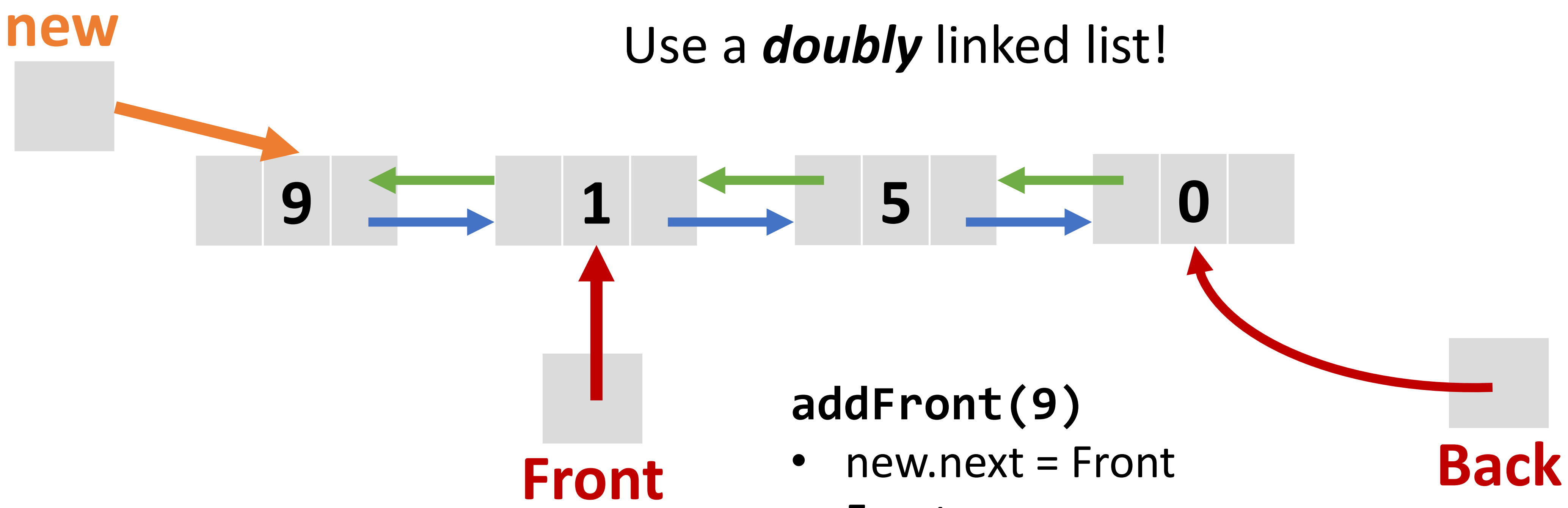
`removeBack()`

- Back = Back.prev
- Back.next.prev = nil
- Back.next = nil

This is where we need the `prev` reference!

Deque: Linked List

Use a ***doubly*** linked list!



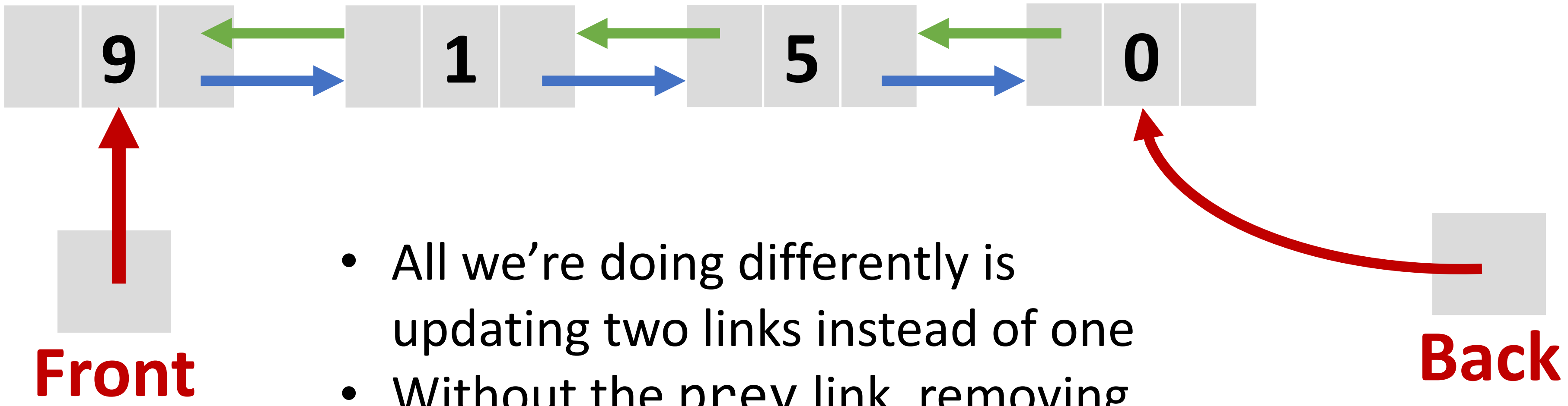
`addFront(9)`

- new.next = Front
- Front.prev = new
- Front = new

`addBack()` functions similarly.

Deque: Linked List

Use a ***doubly*** linked list!



- All we're doing differently is updating two links instead of one
- Without the `prev` link, removing from back would require iterating to the 2nd last element.

Array VS Linked List?

As with most choices in programming and design, there is no one size fits all answer. There are always tradeoffs.

		Linked list	Dynamic Array
	Indexing	O(n)	O(1)
Insert &	Beginning	O(1)	O(n)
delete:	End	O(1) with reference	O(1) amortized
	Middle	Search time + O(1)	O(n)

Sets

Sets

- Sets are an ADT with special properties that differ from stacks and queues.
- You've seen sets in Python, so these properties should be familiar:

1. A set cannot contain duplicate elements:
   - Each unique value can only exist once
2. Sets are *unordered*:
   - Set elements are not stored in any meaningful order.
   - Asking for the third element in a set makes no sense.
3. A consequence of 2), direct indexing is not supported.

So what operations *are* supported?

Sets: Operations

Test membership:	Checks whether an item is in the set
Add and remove:	Insert function must ensure no duplication.
Subset test:	Checks whether one set is a subset of another.
Union:	Combine two sets, ensuring no duplicates.
Intersection:	Find elements that two sets have in common.
Difference:	Remove one set’s elements from another.

- The best way to implement a set is using a Hash Table.
- We haven’t learned these yet, so let’s use a List instead.

Sets: Operations

Test Membership

We could write this ourselves using a loop, but LISP has a built-in form:

```
* (member 4 '(3 5 7 6 4 2 1))
(4 2 1)
* (member 9 '(3 5 7 6 4 2 1))
NIL
```

- If found, return the rest of the list with the found element at the front.
- If not found, return NIL.

Sets: Insert & Remove

```lisp
(defun insert-set (x set)
  (if (member x set)
    set
  (cons x set))
)
```

- If element is already in set, return set as-is
- Otherwise return set with item added.

```
* (insert-set 99 '(1 2 3 4))
(99 1 2 3 4)
* (insert-set 3 '(1 2 3 4))
(1 2 3 4)
```

Sets: Insert & Remove

```
(defun remove-set (x set &optional (acc ()))
  (cond ((null set) acc)
        ((equal x (first set)) (append acc (rest set)))
        (t (remove-set x (rest set) (cons (first set) acc)))
) )
```

- Traverse list, building up visited elements in a new list as we go.
- If we don't find the element, return original list as-is (first cond clause)
- Found element? Concatenate tail of list with list containing previous elements.
- Not found, not at end of list? Make recursive call on rest of list.

Sets: Insert & Remove

```
(defun remove-set (x set &optional (acc ()))
  (cond ((null set) acc)
        ((equal x (first set)) (append acc (rest set)))
        (t (remove-set x (rest set) (cons (first set) acc)))
) )
```

```
* (remove-set 3 '(4 6 8 3 5 9))
(8 6 4 5 9)
* (remove-set 12 '(4 6 8 3 5 9))
(9 5 3 8 6 4)
```

Sets: IS-SUBSET

In mathematics, **A** is a subset of **B** if all elements in **A** are also in **B**

```
(defun subset-set (a b)
  (dolist (item a t)
    (unless (member item b)
    (return))
  )
)
```

Iterative:

- Traverse using DOLIST form, return T.
- If we find an item in A that isn't in B, return (exit dolist, return NIL)

```
* (subset-set '(1 2 3) '(1 2 3 4 5 6))
T
* (subset-set '(1 2 3) '(0 2 3 4 5 6))
NIL
```

Sets: IS-SUBSET

In mathematics, **A** is a subset of **B** if all elements in **A** are also in **B**

```
(defun subset-set-rec (a b)
  (if (null a) T
    (and (member (first a) b)
         (subset-set-rec (rest a) b))
  )
)
```

Recursive:

- If A is empty, return T
- Else, test membership of first, recurse on rest

```
* (subset-set-rec '(1 2 3) '(0 2 3 4 5 6))
NIL
* (subset-set-rec '(1 2 3) '(1 2 3 4 5 6))
T
```

Sets: Union? Intersection?

You’ll do these in Lab #5 this week

In Summary

Fundamental ADTs

- Implementing Linked Lists
- Stacks, Queues, and their variants
- Array VS Linked implementations
- Set ADT using a list.