Tutorial 3: Lists, Stacks and Queues

September 5, 2022

Gu Zhenhao

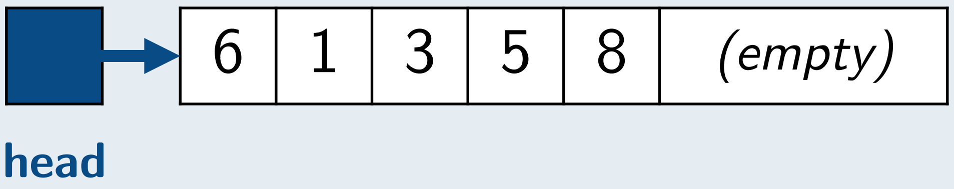
** Partly adopted from tutorial slides by [Wang Zhi Jian](#).*

Arrays & Linked Lists

What is the best way to implement an ADT?

Implementation of List/Queue/Stack ADT

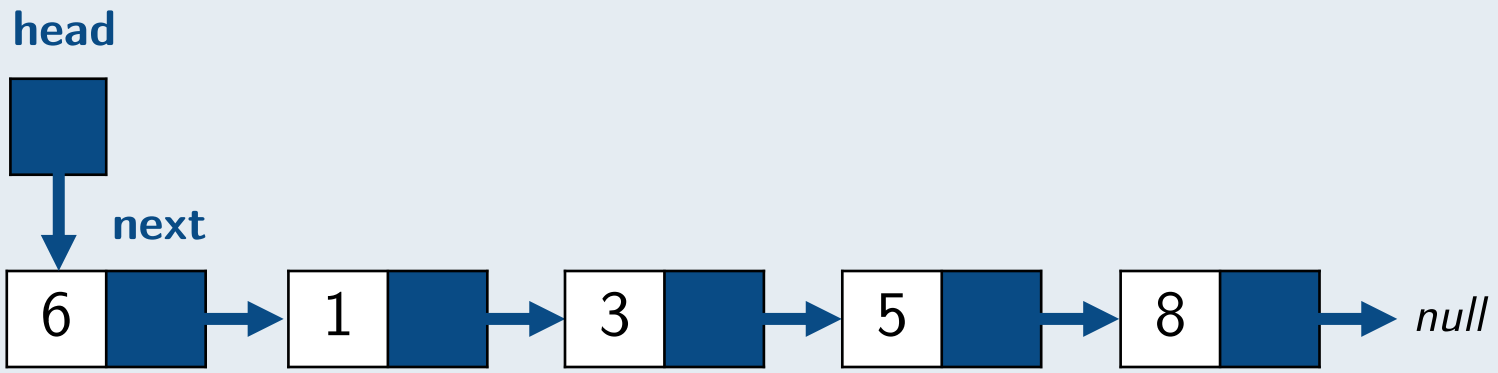
As array:



- **Pros**: Fast random access: $O(1)$ time to access any element; faster to scan the whole list due to *cache locality* (though all take $O(n)$).
- **Cons**: may waste some space; slow insertion/deletion/resizing.

Implementation of List/Queue/Stack ADT

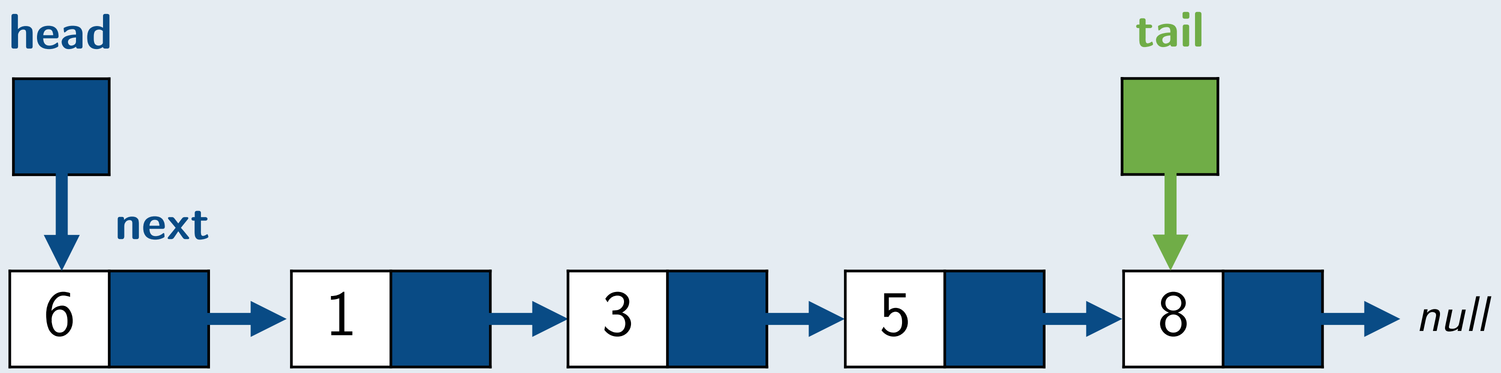
As linked list:



- **Pros**: Dynamic size and flexible memory usage; faster insertion/deletion (especially at head).
- **Cons**: Needs more space to store the pointers, no random access.

Variants of Linked Lists

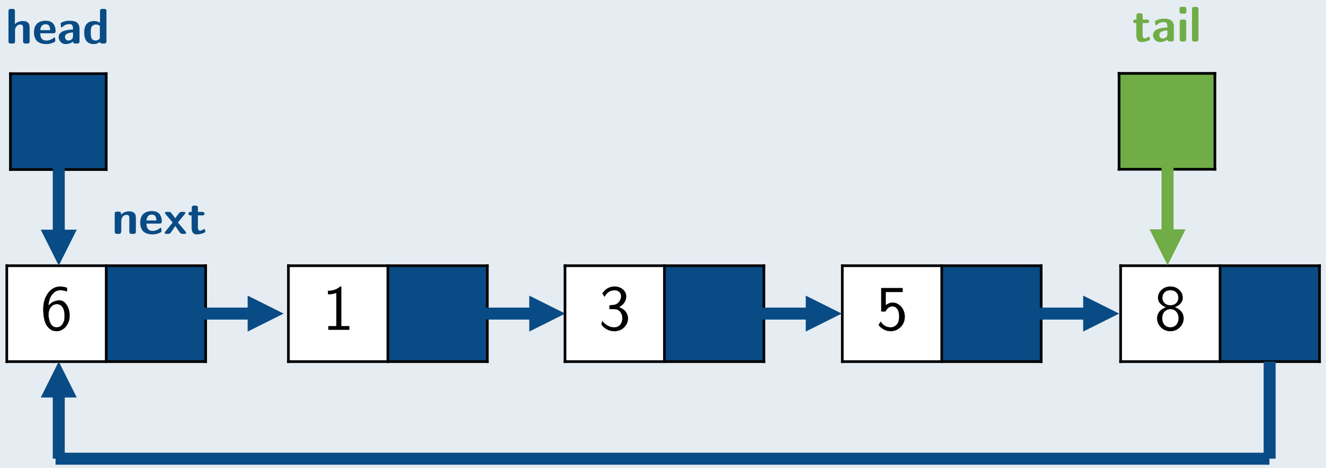
Tailed linked list:



- $O(1)$ access and insert at the tail.

Variants of Linked Lists

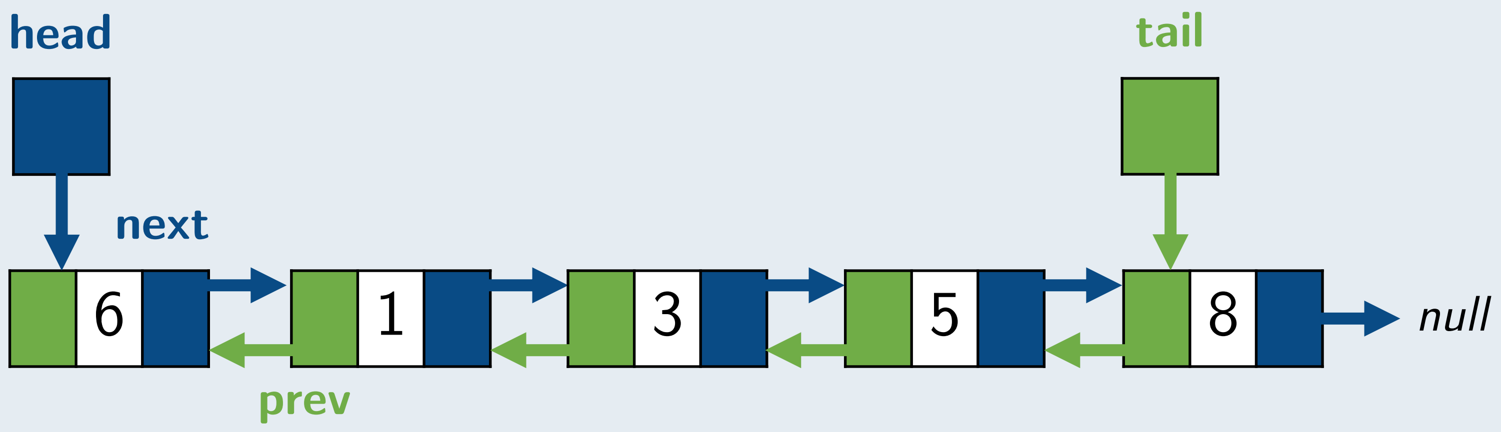
Circular linked list:



- Allow cycling through the list repeatedly.

Variants of Linked Lists

Doubly linked list:



- Allow fast accessing previous items.

Problem 1.a

True or false: *Deletion in any Linked List can always be done in $O(1)$ time.*

False. To delete an element in linked list:

1. Start from head pointer and find the item to be deleted. ($O(n)$ time)
2. Delete the item by modifying pointers. ($O(1)$ time)

Problem 1.b

True or false: *A search operation in a Doubly Linked List will only take $O(\log n)$ time.*

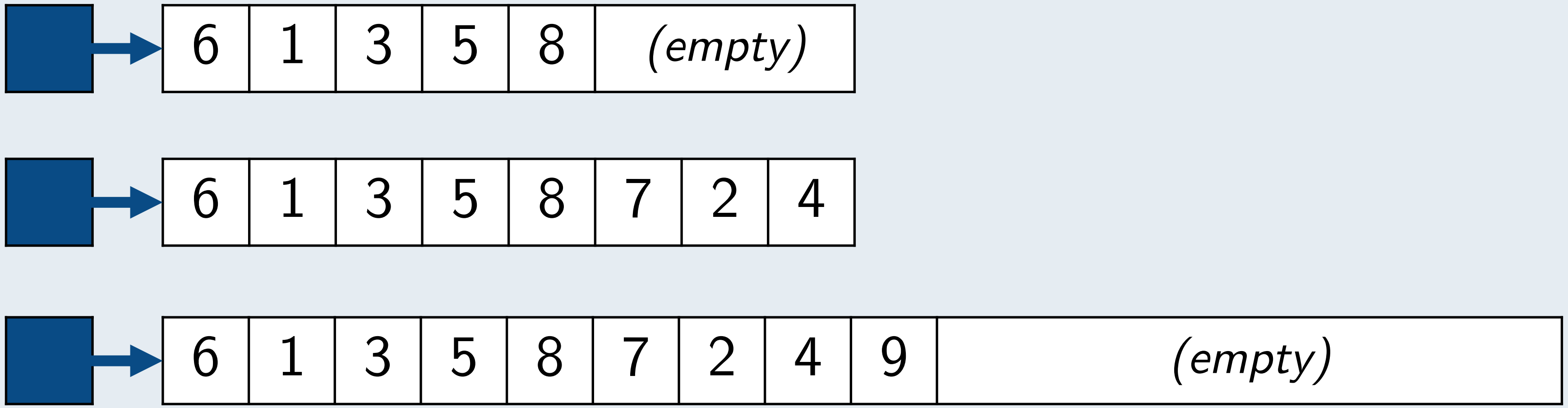
False. To use binary search on a list,

1. List should be sorted,
2. Should allow random access to "jump around" the list,

Doubly linked list has neither.

Problem 1.c

True or false: *All operations in a stack are $O(1)$ time when implemented using an array.*



Inserting needs $O(n)$ in worst case when resizing:

1. Allocate a new space of size $2n$.
2. Copy the array to the new space.

Problem 1.c

True or false: *All operations in a stack are $O(1)$ time when implemented using an array.*

False. The *push* operation needs in worst case $O(n)$.

You can also say true. You will need n $O(1)$ push operations so that the next push operation cost $O(n)$. So the total *amortized cost* is still $O(1)$.

** See more on amortized analysis in the appendix!*

Problem 1.d

True or false: *A stack can be implemented with a Singly Linked List with no tail reference with $O(1)$ time for all operations.*

True. *push* and *pop* only require insert and delete at the head of linked list, which cost $O(1)$.

Problem 1.e

True or false: *All operations in a queue are $O(1)$ time when implemented using a Doubly Linked List with no modification.*

True. Doubly Linked List has a tail reference so both enqueue at front and dequeue at the end costs $O(1)$.

You can also say false by assuming* the doubly linked list may not have a tail reference.

** Reasonable assumptions are accepted as correct answer in CS2040S exams.*

Problem 1.f

True or false: *Three items A, B, C are inserted (in this order) into an unknown data structure X. If the first element removed from X is B, X can be a queue.*

False. If X is a queue (FIFO), then the first element removed is A.

Summary

Operations	Array	Linked List
`getItemAtIndex`	$O(1)$	$O(n)$
`getFirst/getLast`	$O(1)$	$O(1)^*$
`addAtIndex/removeAtIndex`	$O(n)$	$O(n)$
`addFront/removeFront`	$O(n)$	$O(1)$
`addBack/removeBack`	$O(n)$ ($O(1)$ amortized)	$O(1)^*$

Lists: depends on application, usually needs `getItemAtIndex`.

Stack (LIFO): needs `addFront/removeFront`.

Queue (FIFO): needs `addFront/removeBack`.

** Needs tail reference for $O(1)$ operation at back.*

New Operations

How to enable other operations in those ADT?

Problem Solving Strategies

1. **Think of a trivial answer.**

   Brute-force algorithms? Some algorithms we've learned?

2. **Try with some simple examples.**

   Some small or trivial input?

3. **See where we can improve.**

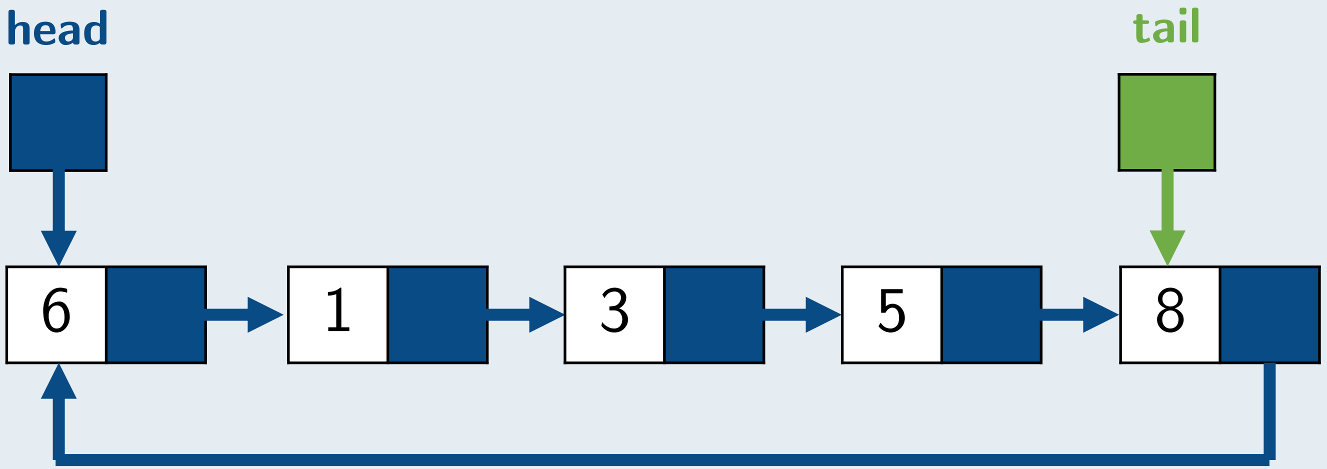
   Use better data structure? Delete some redundant work?

4. **Consider special and boundary cases.**

   What if input size is small, say 1 or 2? What if input is special?

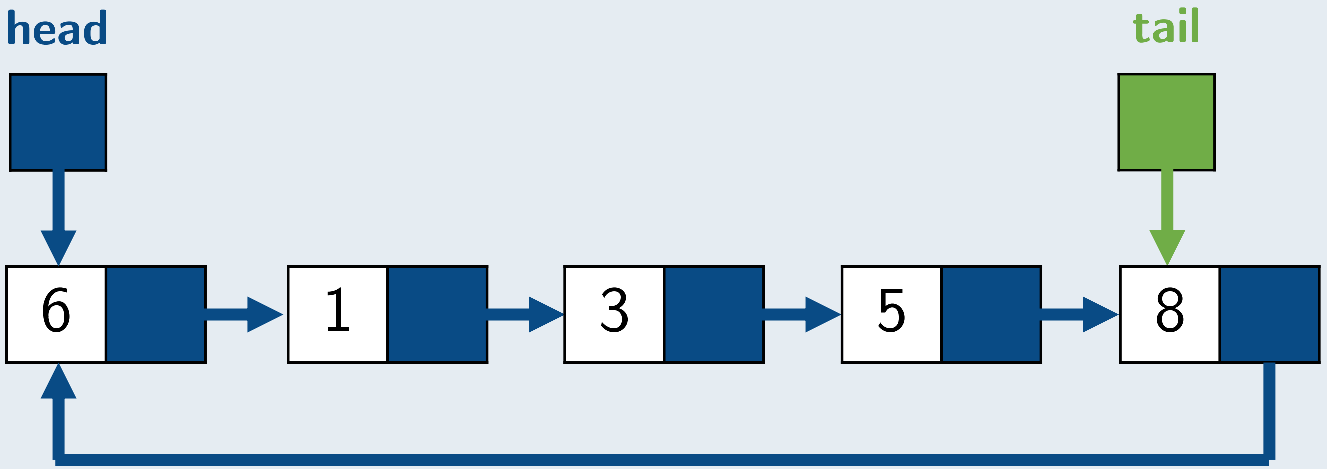
Problem 2

Goal: enable *swap* operation (swapping adjacent elements) in a circular linked list.



Problem 2

Simple Example: Try to swap element 3 and 5.

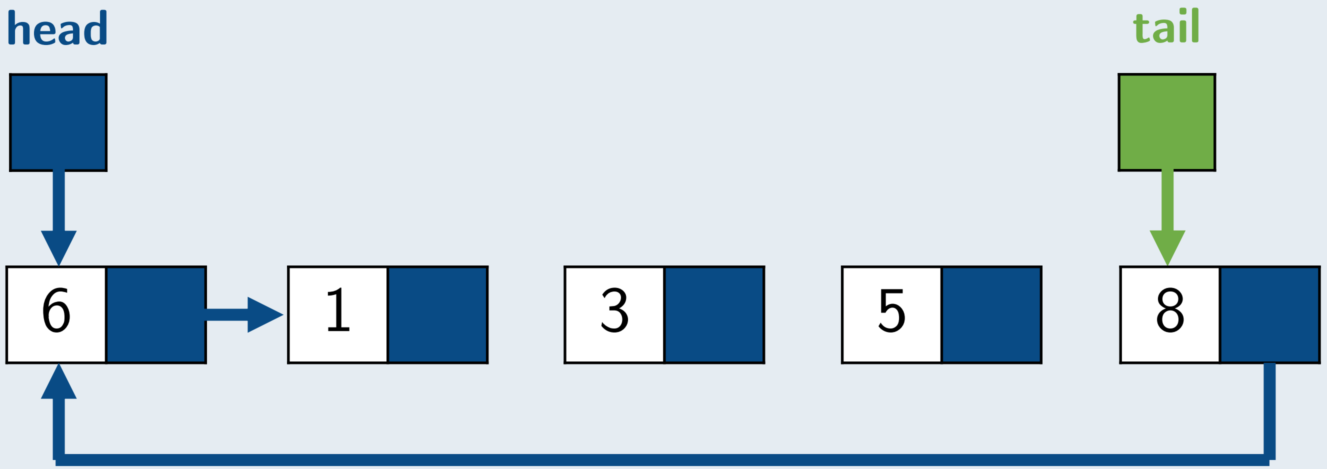


Which pointers do we need to modify?

We basically want $1 \to 5 \to 3 \to 8$.

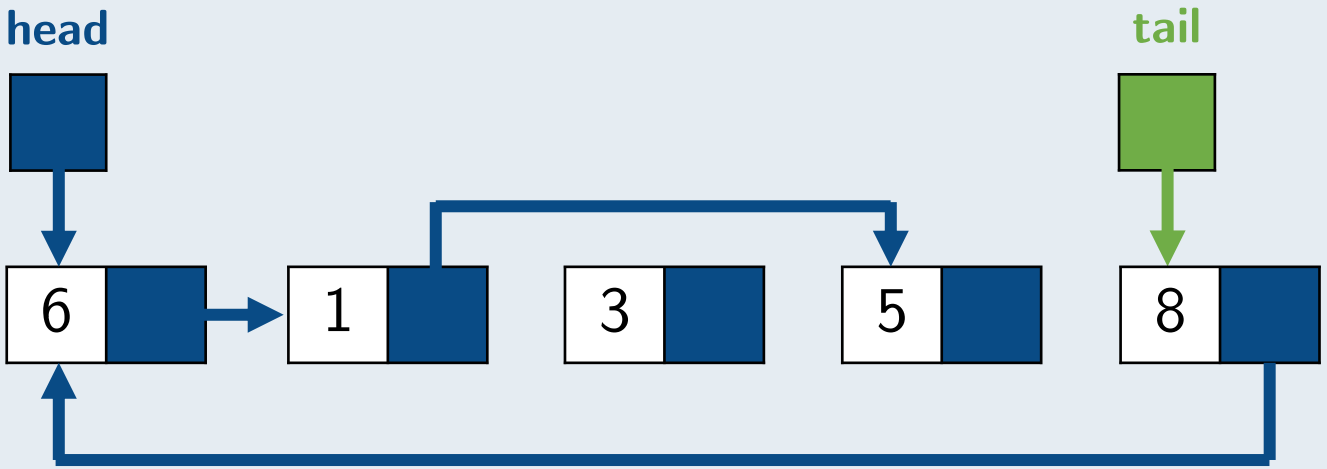
Problem 2

Simple Example: Try to swap element 3 and 5.



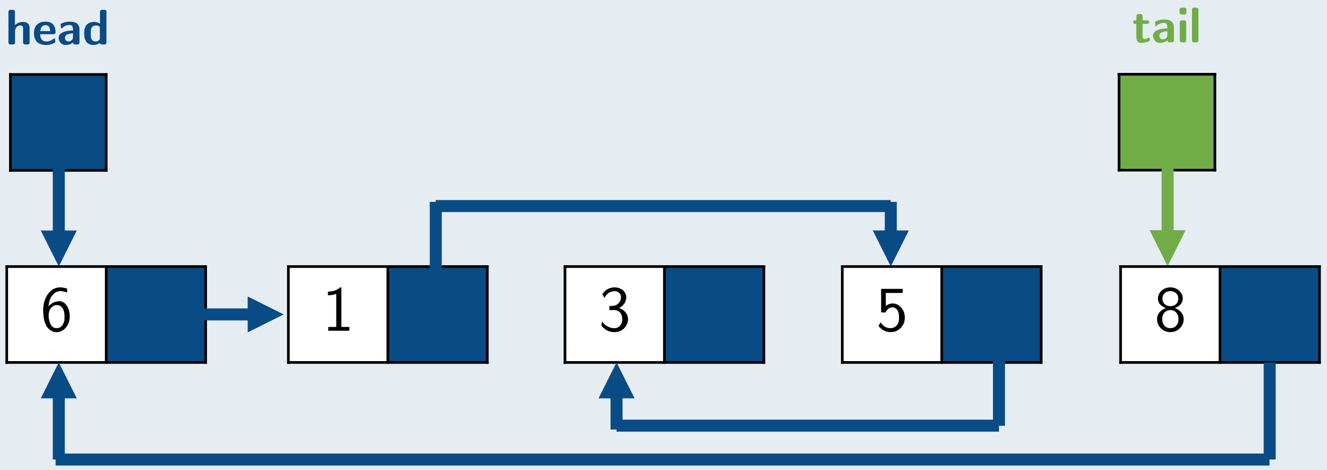
Problem 2

Simple Example: Try to swap element 3 and 5.



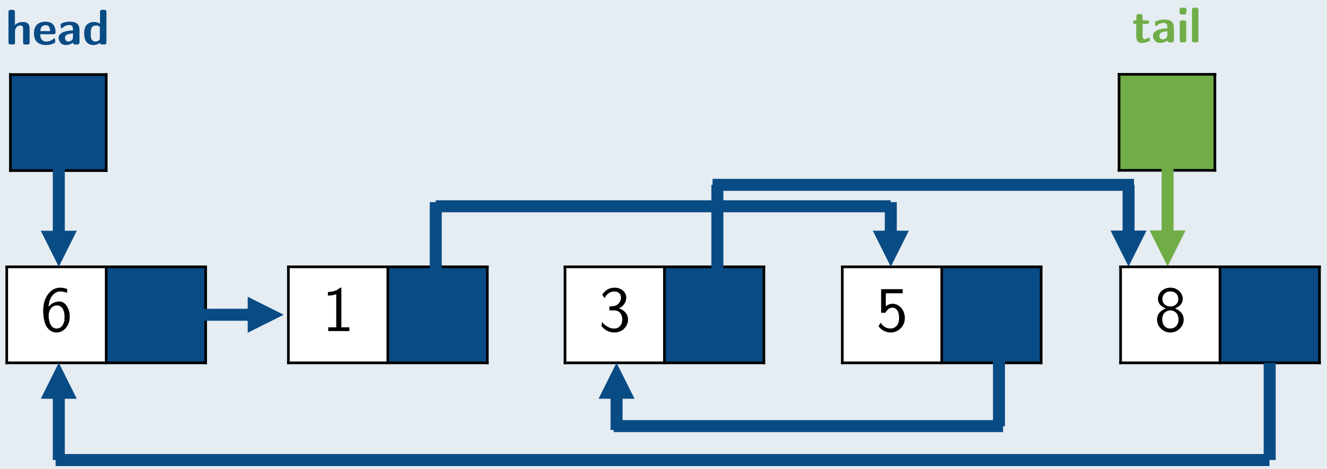
Problem 2

Simple Example: Try to swap element 3 and 5.



Problem 2

Simple Example: Try to swap element 3 and 5.

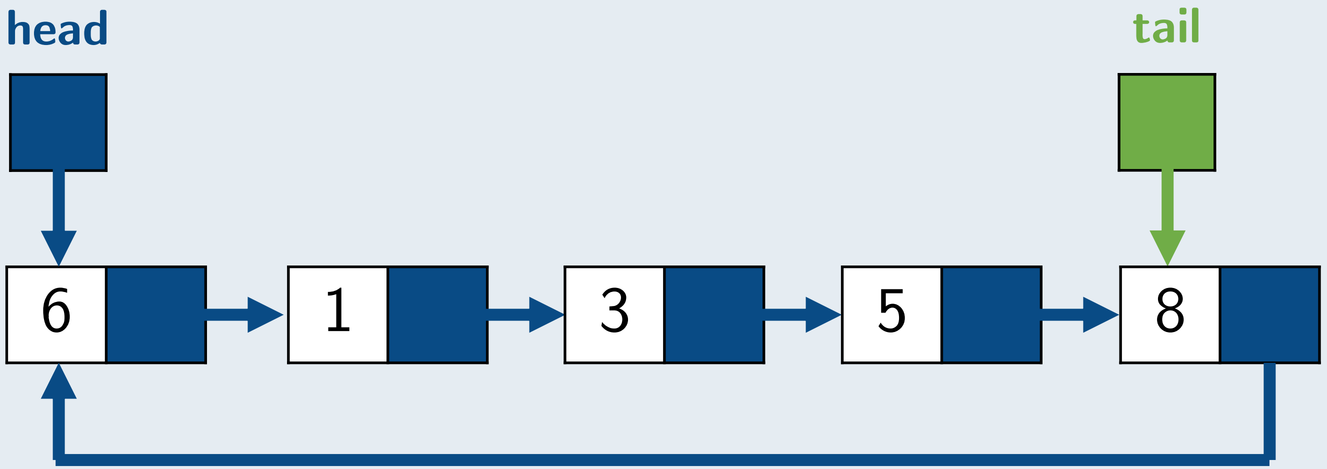


Need to modify 3 pointers. the previous one and the two we are swapping.

Problem 2

Boundary Case: *What if the items are at head and/or tail?*

Try to swap element 5 and 8.



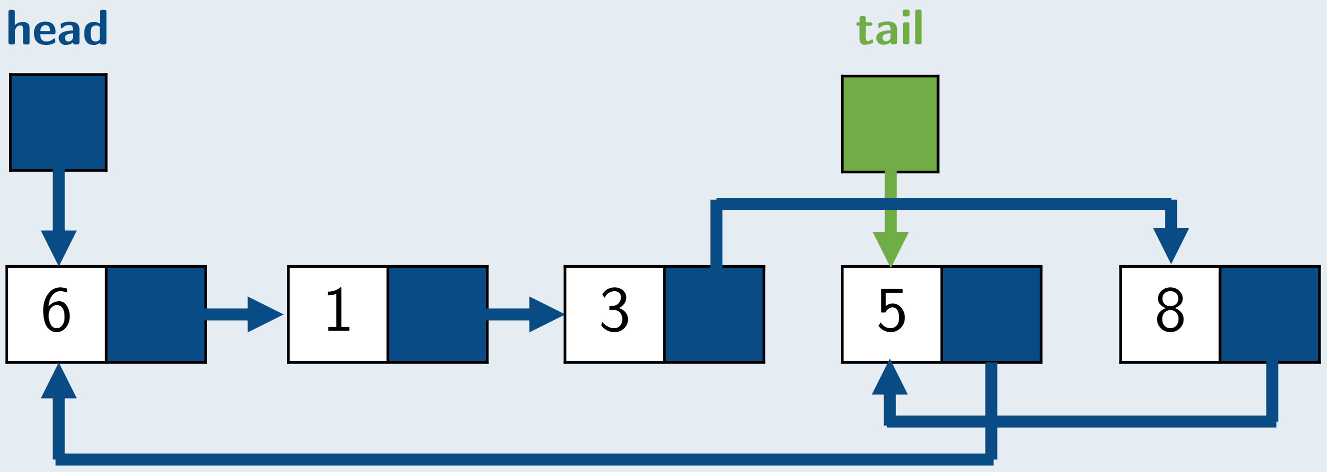
Which pointers do we need to modify?

We basically want $3 \to 8 \to 5 \to 6$.

Problem 2

Boundary Case: *What if the items are at head and/or tail?*

Try to swap element 5 and 8.

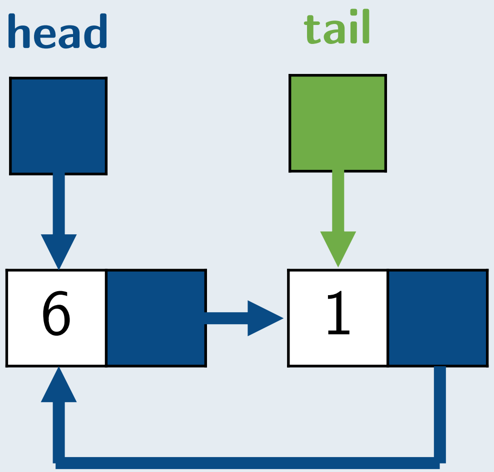


Need to modify the 3 pointers plus head and/or tail pointer.

Problem 2

Special Case: *What if there are less than 3 pointers to modify?*

Say we have only two elements.

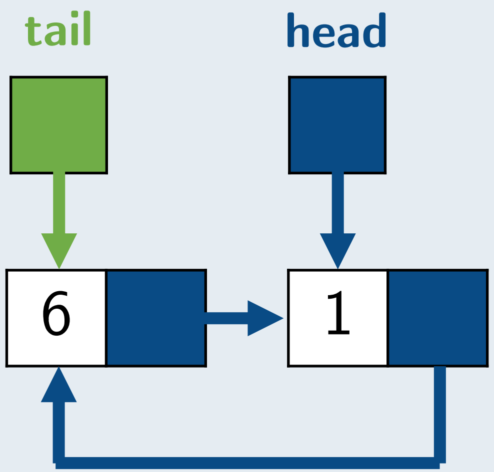


Which pointers do we need to modify?

Problem 2

Special Case: *What if there are less than 3 pointers?*

Say we have only two elements.

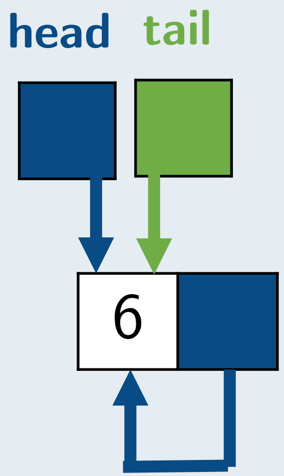


Only the head and tail pointers!

Problem 2

Special Case: *What if there are less than 3 pointers?*

Say we have only one element.

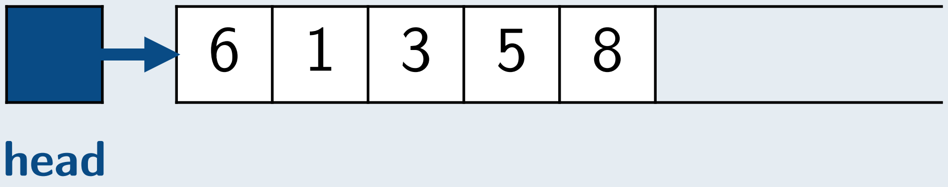


No need to do anything :-D

* *See `CircularLinkedList.java` for a detailed solution.*

Problem 3

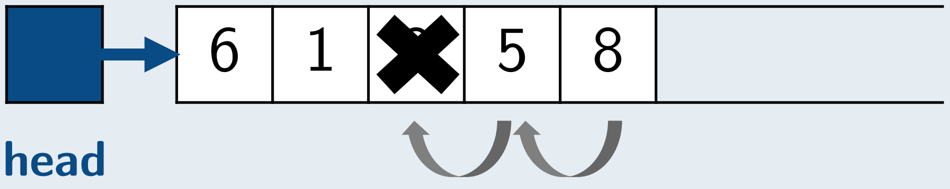
- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).



Simple Example: suppose we want to delete item 3.

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).

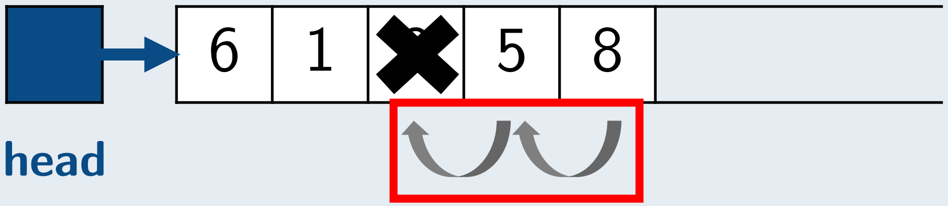


Trivial answer: Simply delete the item from the array.

Time complexity of *leave*: $O(n)$.

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).

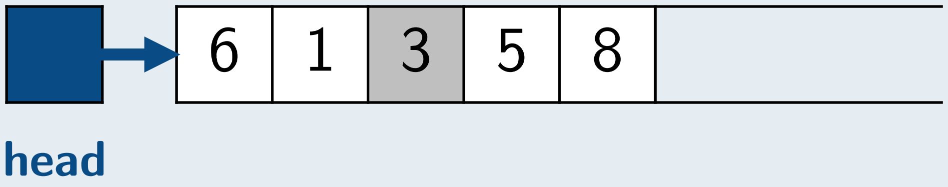


Not necessary if we remember who has left!

Note. Actually we don't need the queue to be correct. We just need to make sure other operations *enqueue* and *dequeue* return the correct result.

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).

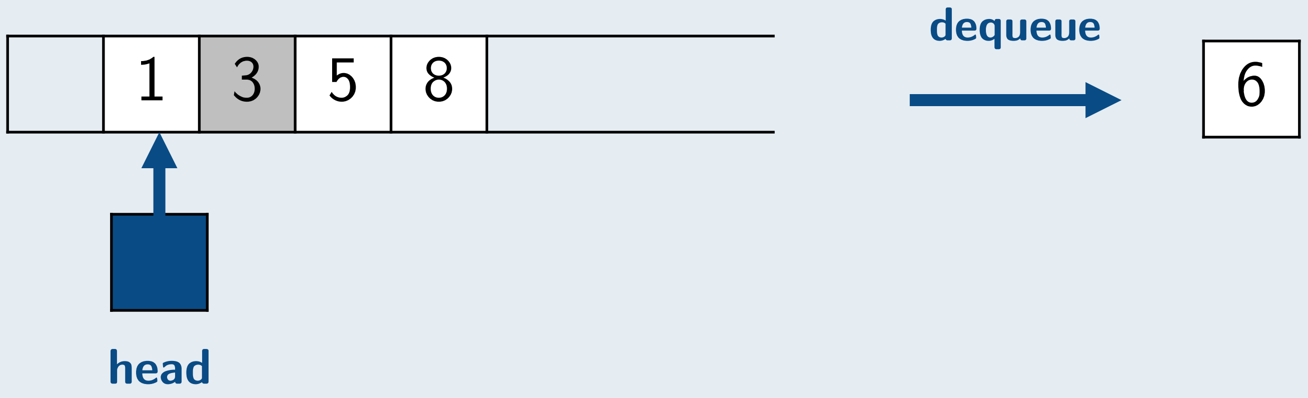


Idea 1 (Lazy deletion):

1. Simply mark the item as *left*.
2. *dequeue* skips the *left* items and continue on to next item.

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).

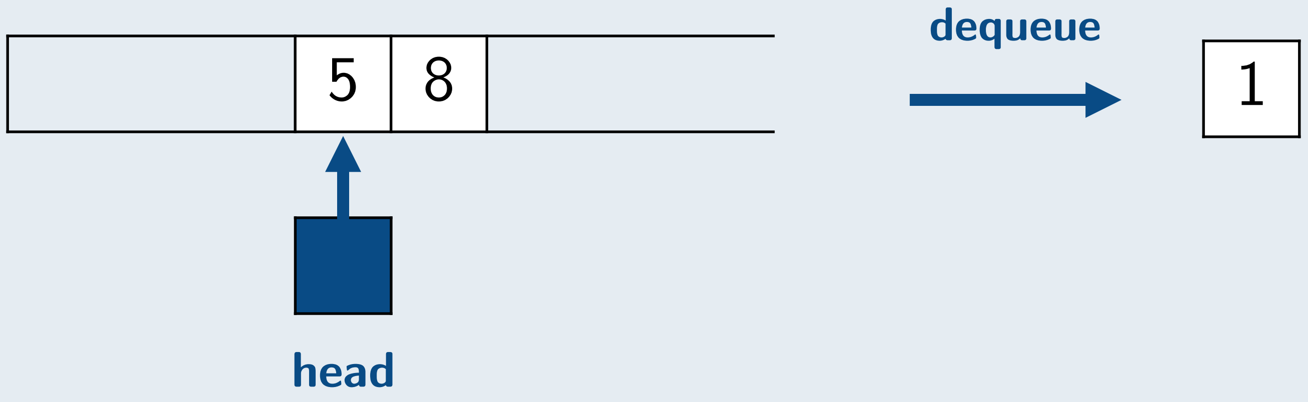


Idea 1 (Lazy deletion):

1. Simply mark the item as *left*.
2. *dequeue* skips the *left* items and continue on to next item.

Problem 3

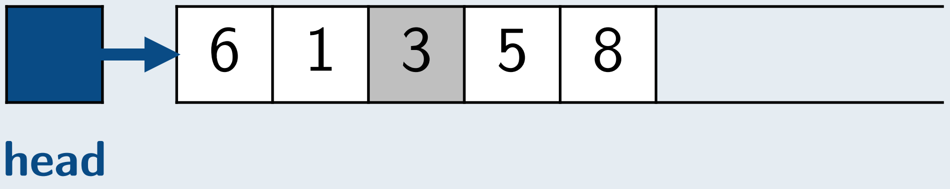
- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).



Note. You may also need a tail pointer for fast enqueue and indicate which is the last item.

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).

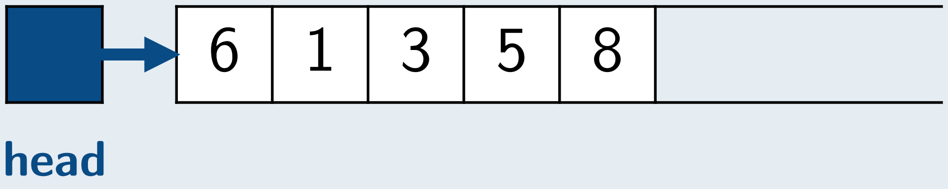


Idea 1 (Lazy deletion):

- *leave* costs only $O(1)$.
- *dequeue* may cost $O(n)$ if $O(n)$ items have left (still $O(1)$ amortized).

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).



Left items
3

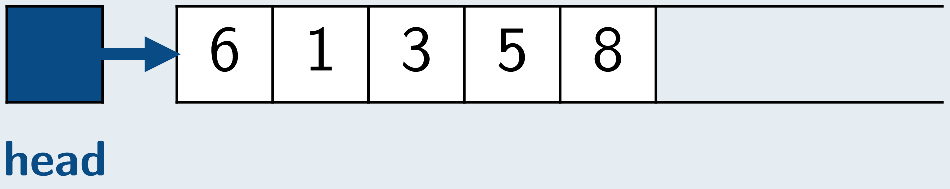
Idea 2:

1. Keep a dictionary (which data structure?) on who has left.
2. *dequeue* skips items in the dictionary and continue onto next ones

** See `WaitingQueue.java` for a detailed solution.*

Problem 3

- **Goal**: enable *leave* operation that allows a certain item to leave the queue (as an array).



Left items
3

Idea 2:

1. If we use hash table, leave costs $O(1)$.
2. dequeue also costs $O(1)$ amortized.
3. Needs $O(n)$ additional space for the hash table.

Applications of ADT

How to efficiently use lists, stacks & queues?

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

Problem 4

- **Goal**: evaluate Lisp expression.

`( + ( - 6 ) ( * 2 3 4 ) )`

We need to ...

1. Match each pair of parenthesis, ← Last-in, first-out
2. Evaluate expression inside each parenthesis **in correct order**.
   e.g. `( / 4 2 1 )` should be `4/2/1=2` instead of `1/2/4=0.125`.

First-in, first-out

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
↑
```

To match parenthesis, let's start with a stack.

(	

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack.

(	+	(	

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack.

(	+	(	-	

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack.

(	+	(	-	6	

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack.

(	+	(	-	6	

A right parenthesis!

Problem 4

- **Goal**: evaluate Lisp expression.

(+ (- 6) (* 2 3 4))

To match parenthesis, let's start with a stack.

(	+	(	-	

$\xrightarrow{\text{pop}}$ 6

pop until the last left parenthesis we've pushed.

Problem 4

- **Goal**: evaluate Lisp expression.

(+ (- 6) (* 2 3 4))

To match parenthesis, let's start with a stack.

(	+	(		$\xrightarrow{\text{pop}}$	6	-

pop until the last left parenthesis we've pushed.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack.

Idea: Use another stack to reverse the order!

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

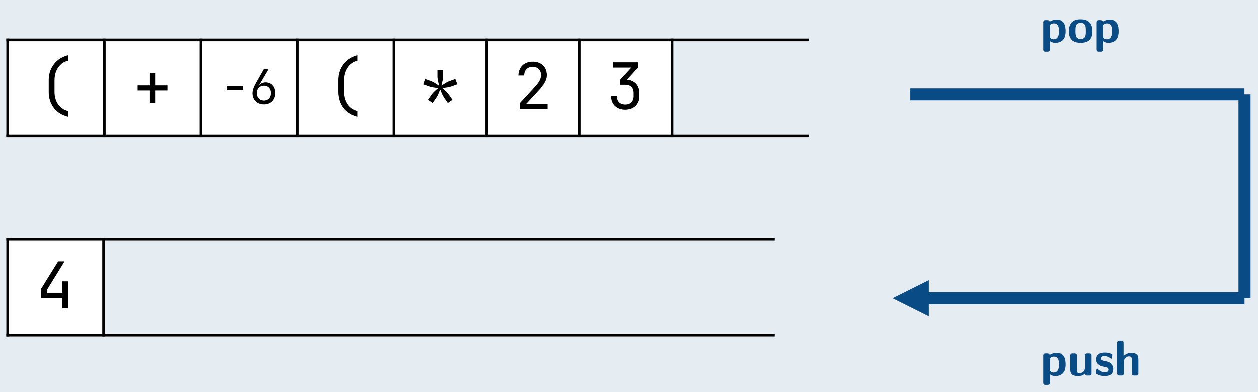
(	+	-6	(	*	2	3	4

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

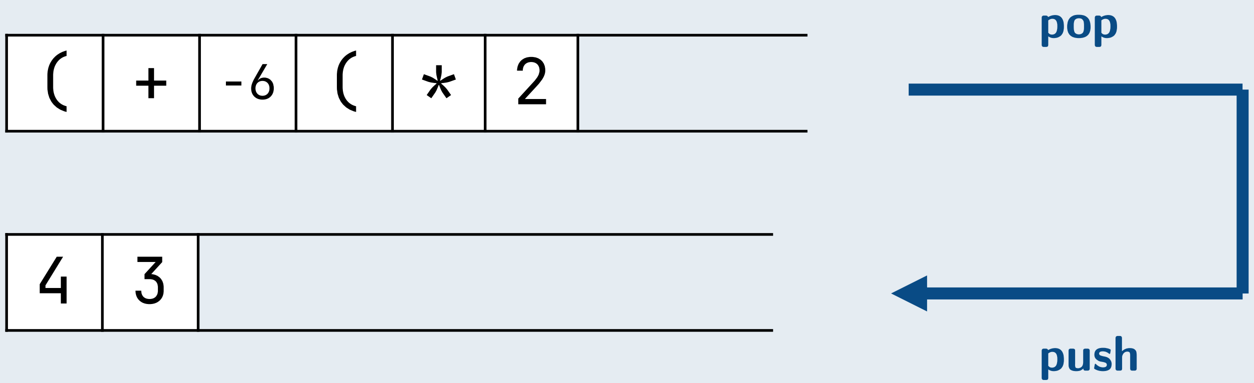


Problem 4

- **Goal**: evaluate Lisp expression.

(+ (- 6) (* 2 3 4))

To match parenthesis, let's start with a stack. Another stack for reversing order.

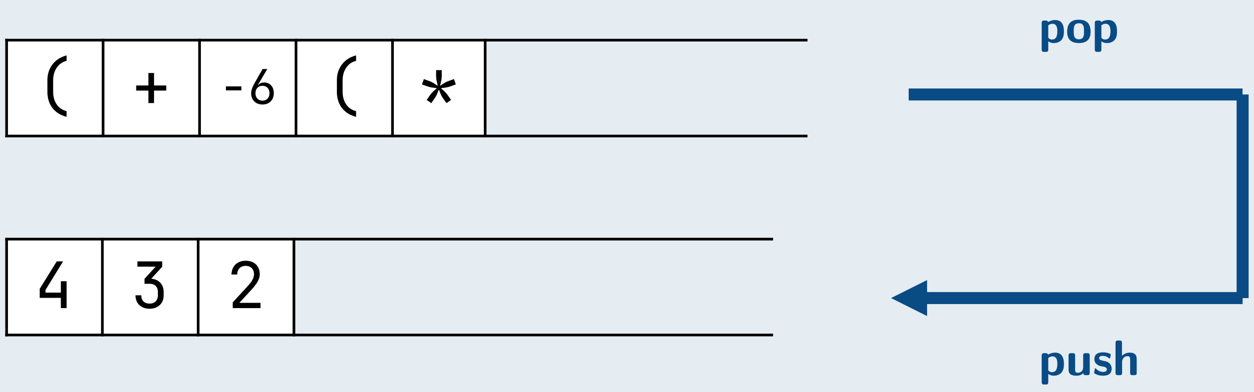


Problem 4

- **Goal**: evaluate Lisp expression.

(+ (- 6) (* 2 3 4))

To match parenthesis, let's start with a stack. Another stack for reversing order.

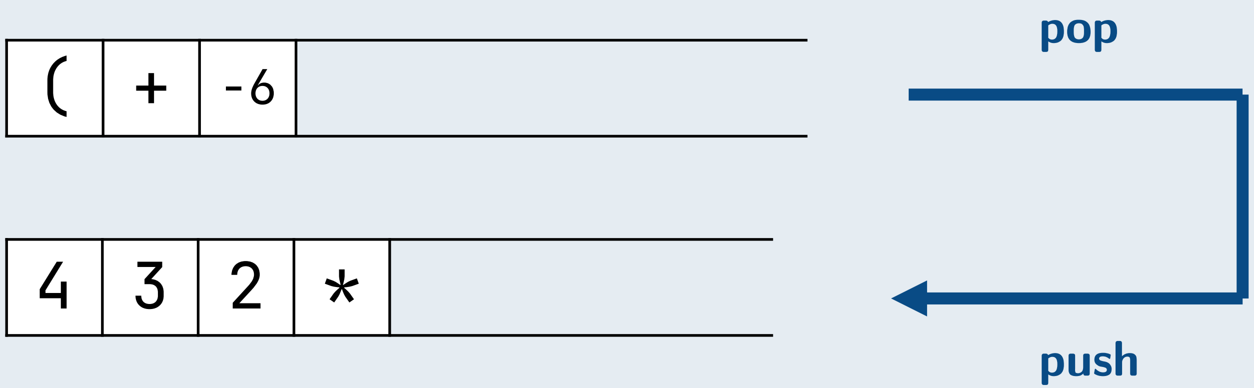


Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.



Problem 4

- **Goal**: evaluate Lisp expression.

`( + ( - 6 ) ( * 2 3 4 ) )`

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

`( + ( - 6 ) ( * 2 3 4 ) )`

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

```
( + ( - 6 ) ( * 2 3 4 ) )
```

To match parenthesis, let's start with a stack. Another stack for reversing order.

Problem 4

- **Goal**: evaluate Lisp expression.

`( + ( - 6 ) ( * 2 3 4 ) )`

Each element pushed and popped twice: $O(n)$ time.

** See* `ExpEval.java` *for a detailed solution.*

Appendix

Amortized Analysis *

- **Purpose**: take the frequency of an operation into account.
- **Example**: I eat at the Deck and spend \$5 for lunch every day. Once every month I decide to reward myself and have a \$100 lunch.
    - **Worst case analysis**: $\leq$ \$100 for lunch every day.
    - **Amortized analysis**: $\leq$ \$8 for lunch every day. (reserve \$3 for each lunch at the Deck, and spend them all on the rewarding lunch)

A more reasonable way to describe expensive operations that rarely happen!

Amortized Analysis *

- **Example**: Inserting at the back of an array usually costs $O(1)$. After $O(n)$ insertions, the next insert will cost $O(n)$ for resizing.
  - **Worst case analysis**: $O(n)$ per insert.
  - **Amortized analysis**: $O(1)$ per insert. (reserve $O(1)$ time for each regular insert, and spend them all on the resizing insert)

End of File

Thank you very much for your attention :-)