Tutorial 03 - Linked List, Stack, Queue, Deque

CS2040S Semester 1 2023/2024

By Wu Biao, adapted from previous slides

Set real display name

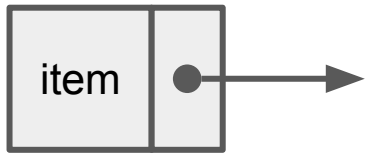
https://pollev.com/rezwanarefin430

Content review

Singly vs Doubly Linked List

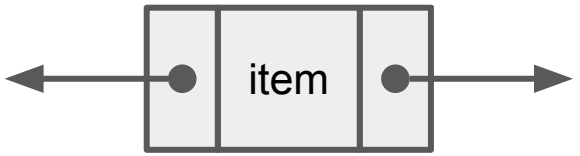
Singly Linked List (SLL) only has *next* pointers.

- Can only iterate *forward.*



Doubly Linked List (DLL) has both *next* and *prev* pointers.

- Can iterate both *forward* and *backward.*



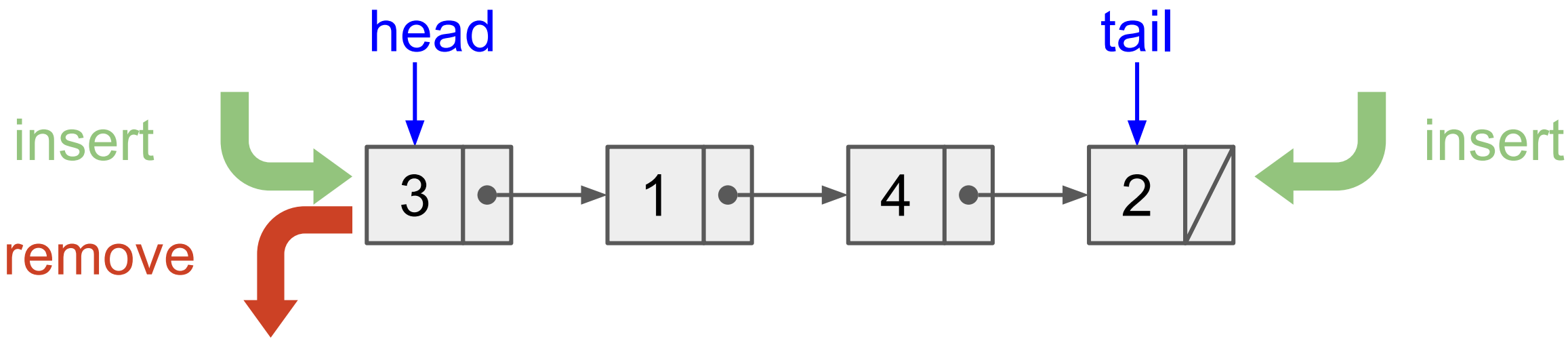
Singly/Doubly Linked List are **data structure implementations**, not **ADT!**

List ADT ‘variants’

- Realize that Stack, Queue and Deque ADTs are similar to List ADT (subset of operations).
- Singly Linked List can be used to implement:
  - Stack ADT
  - Queue ADT
- Doubly Linked List can be used to implement:
  - Deque ADT

Singly Linked List (SLL)

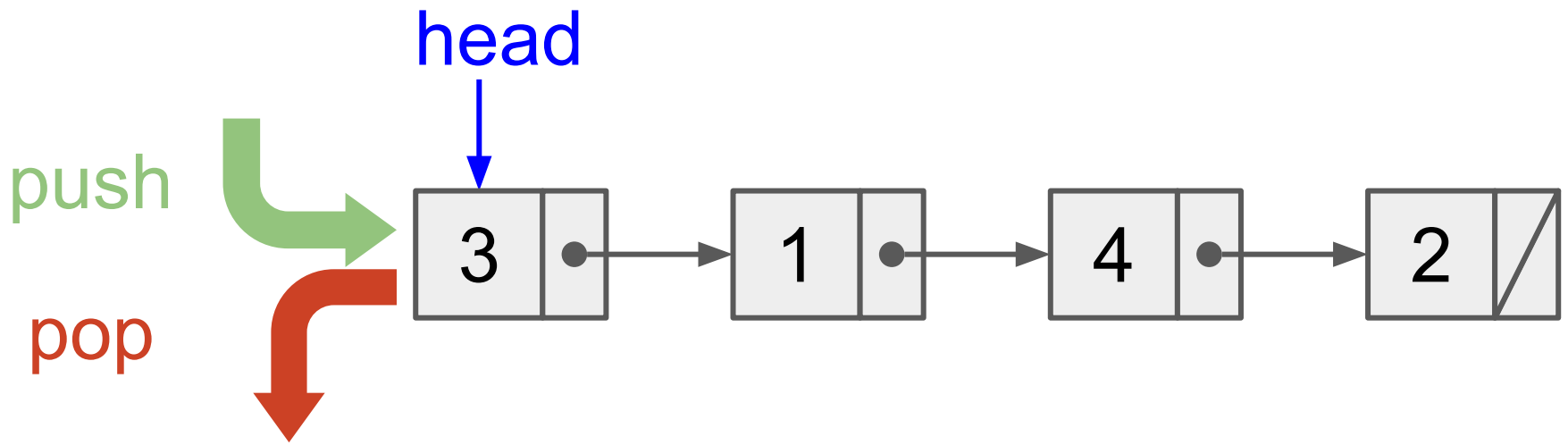
Following operations in $O(1)$



Stack (implemented via SLL)

Subset of List ADT operations

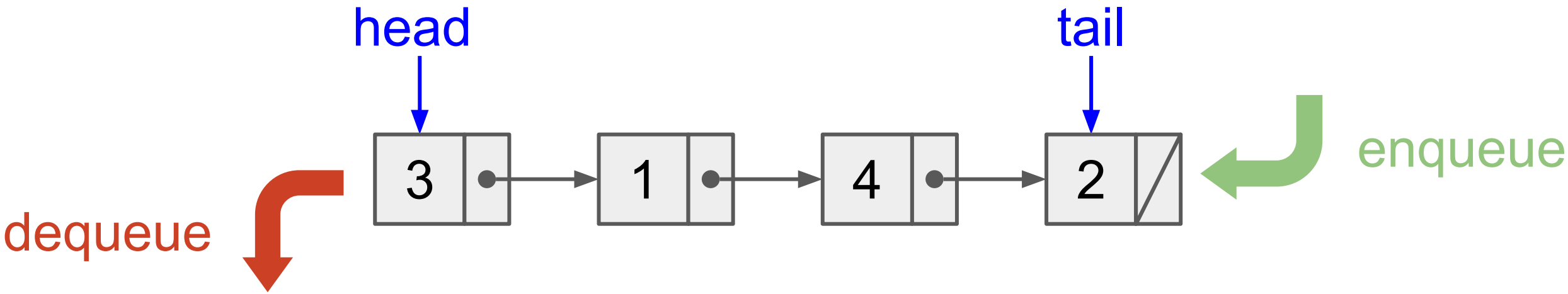
Following operations in $O(1)$



Queue (implemented via SLL)

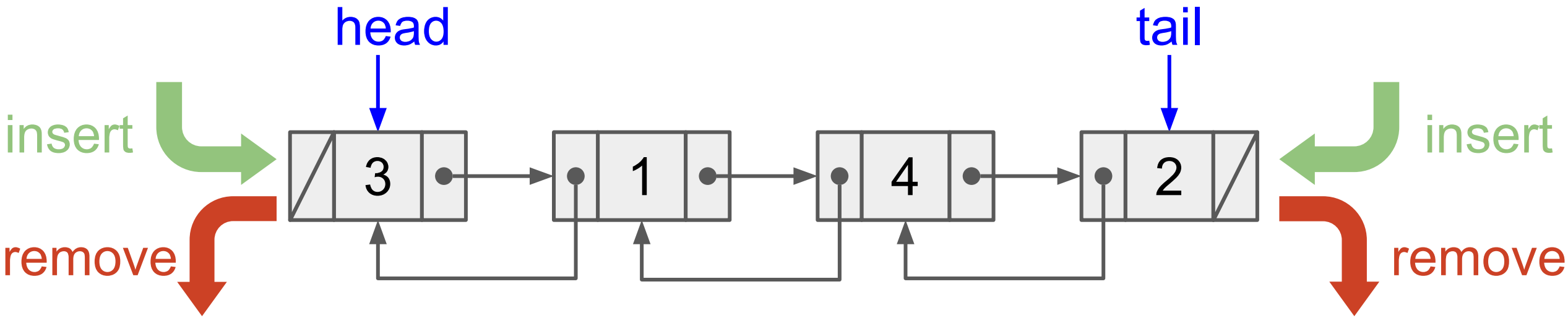
Subset of List ADT

Following operations in $O(1)$



Doubly Linked List (DLL)

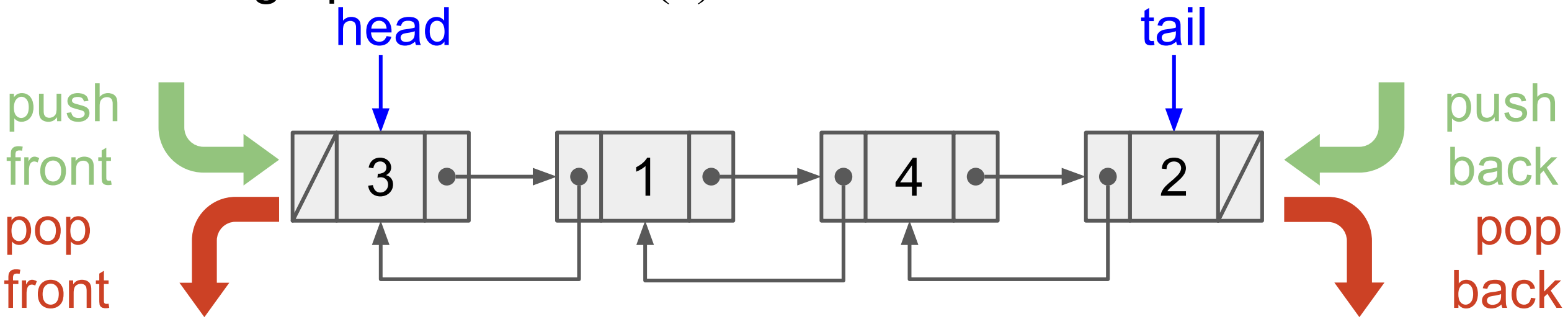
Following operations in $O(1)$



Deque (implemented via DLL)

Just doubly linked list without *search* and *operations in the middle*.

Following operations in $O(1)$



Q1: Linked List, Mini Experiment

Mode → Action ↓	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
peek-front()	$O(1)$				
peek-back()					$O(1)$
insert(0, new-v)				$O(1)$	
insert(N, new-v)					$O(1)$
insert(i, new-v), $i \in [1..N-1]$		Not allowed			
remove(0)					
remove($N-1$)		Not allowed			
remove(i), $i \in [1..N-2]$				$O(N)$	

Mode → Action ↓	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
peek-front()	$O(1)$				
peek-back()	$O(1)$				$O(1)$
insert(0, new-v)	$O(1)$			$O(1)$	
insert(N, new-v)	$O(1)$				$O(1)$
insert(i, new-v), $i \in [1..N-1]$	$O(N)$	Not allowed			
remove(0)	$O(1)$				
remove($N-1$)	$O(N)$	Not allowed			
remove(i), $i \in [1..N-2]$	$O(N)$			$O(N)$	

Mode → Action ↓	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
peek-front()	$O(1)$	$O(1)$			
peek-back()	$O(1)$	Not allowed			$O(1)$
insert(0, new-v)	$O(1)$	$O(1)$		$O(1)$	
insert(N, new-v)	$O(1)$	Not allowed			$O(1)$
insert(i, new-v), $i \in [1..N-1]$	$O(N)$	Not allowed			
remove(0)	$O(1)$	$O(1)$			
remove($N-1$)	$O(N)$	Not allowed			
remove(i), $i \in [1..N-2]$	$O(N)$	Not allowed		$O(N)$	

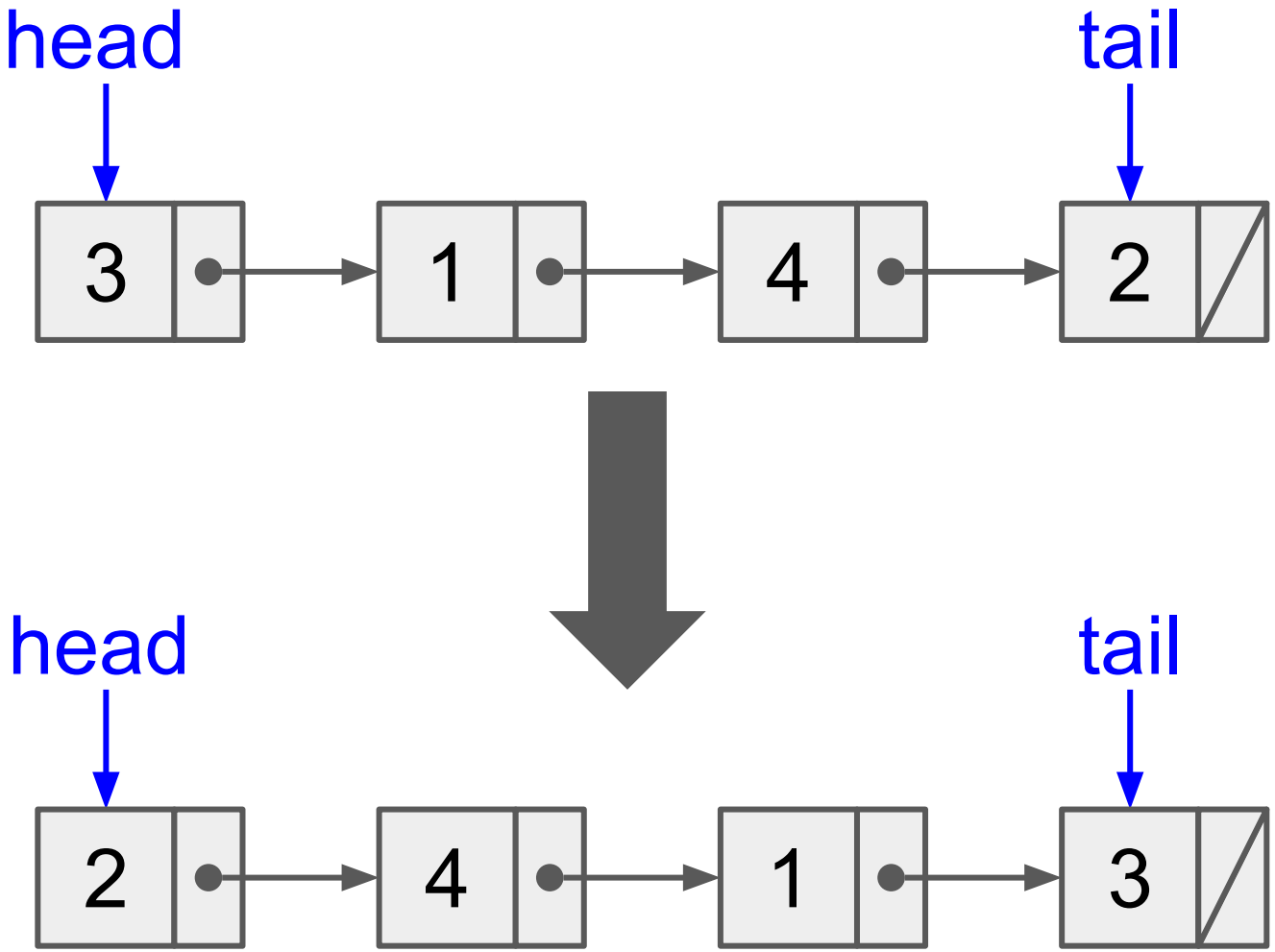
Mode → Action ↓	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
peek-front()	$O(1)$	$O(1)$	$O(1)$		
peek-back()	$O(1)$	Not allowed	Not allowed		$O(1)$
insert(0, new-v)	$O(1)$	$O(1)$	Not allowed	$O(1)$	
insert(N, new-v)	$O(1)$	Not allowed	$O(1)$		$O(1)$
insert(i, new-v), $i \in [1..N-1]$	$O(N)$	Not allowed	Not allowed		
remove(0)	$O(1)$	$O(1)$	$O(1)$		
remove($N-1$)	$O(N)$	Not allowed	Not allowed		
remove(i), $i \in [1..N-2]$	$O(N)$	Not allowed	Not allowed	$O(N)$	

Mode → Action ↓	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
peek-front()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	
peek-back()	$O(1)$	Not allowed	Not allowed	$O(1)$	$O(1)$
insert(0, new-v)	$O(1)$	$O(1)$	Not allowed	$O(1)$	
insert(N, new-v)	$O(1)$	Not allowed	$O(1)$	$O(1)$	$O(1)$
insert(i, new-v), $i \in [1..N-1]$	$O(N)$	Not allowed	Not allowed	$O(N)$	
remove(0)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	
remove($N-1$)	$O(N)$	Not allowed	Not allowed	$O(1)$	
remove(i), $i \in [1..N-2]$	$O(N)$	Not allowed	Not allowed	$O(N)$	

Mode → Action ↓	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
peek-front()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
peek-back()	$O(1)$	Not allowed	Not allowed	$O(1)$	$O(1)$
insert(0, new-v)	$O(1)$	$O(1)$	Not allowed	$O(1)$	$O(1)$
insert(N, new-v)	$O(1)$	Not allowed	$O(1)$	$O(1)$	$O(1)$
insert(i, new-v), $i \in [1..N-1]$	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed
remove(0)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
remove($N-1$)	$O(N)$	Not allowed	Not allowed	$O(1)$	$O(1)$
remove(i), $i \in [1..N-2]$	$O(N)$	Not allowed	Not allowed	$O(N)$	Not allowed

Q2: `reverseList()` & `sortList()`

Reversing a SLL



Reversing a SLL

Would anyone like to share?

Describe your solution.

Reversing a SLL (Array method)

1. Loop through **A**, store pointers to every element in an array
2. Optional: Deconstruct **A**, if memory is tight
3. Loop through the array in reverse order, construct the reversed linked list **B**

Complexity: $O(N)$ time and space

Reversing a SLL (Stack reverse method)

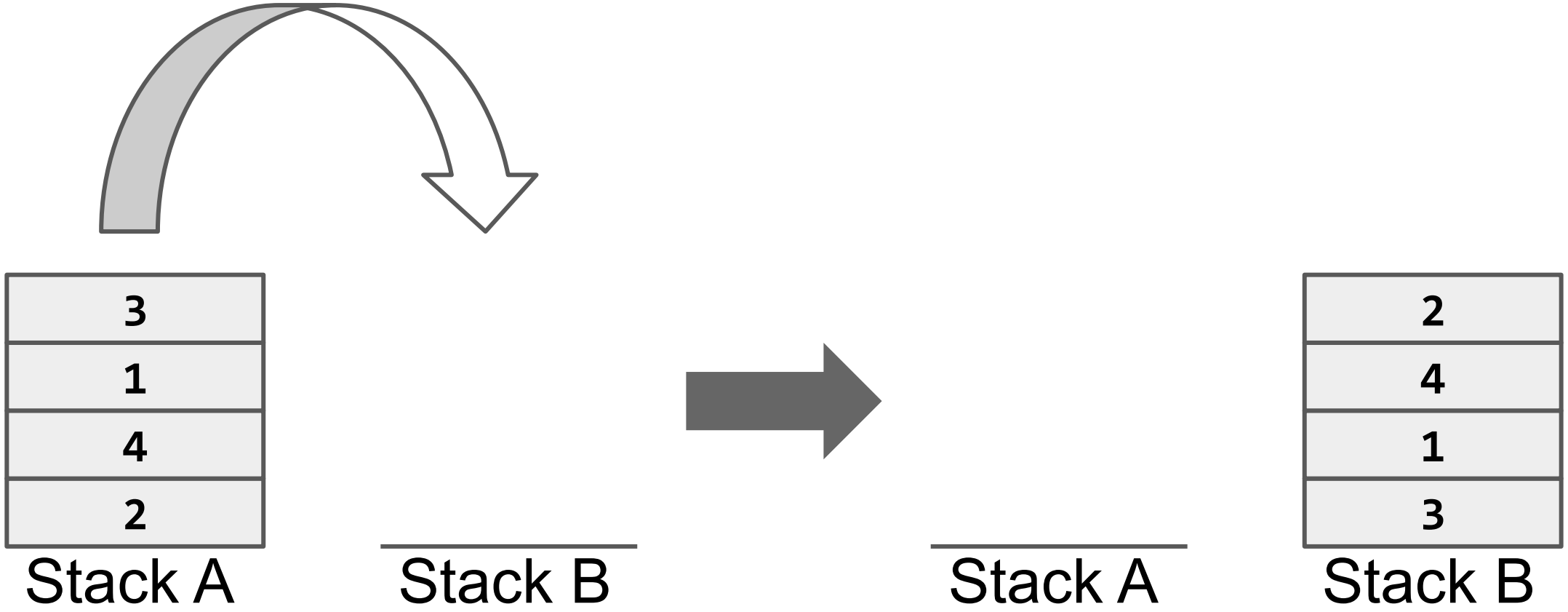
Let the original list be **A** and another empty list be **B**

1. Iterate through **A** (forward),
2. Push elements successively at the head of **B**
3. Once complete, **B** will have the all the items of **A** but in reverse ordering

Complexity: $O(N)$ time and space

Reversing a SLL (Stack reverse method)

Analogous to reversing a stack:



Reversing a SLL (Recursion method)

Property: Every vertex in a SLL is a sublist starting with that vertex

1. We can also use recursion to go deep from the head element to the tail element in O(N).
2. Then, as the recursion unwinds, we can reverse the link from u -> v into v -> u.
3. Lastly, we swap the head and tail pointers.

Complexity: $O(N)$ time and space (May stack overflow if N large)

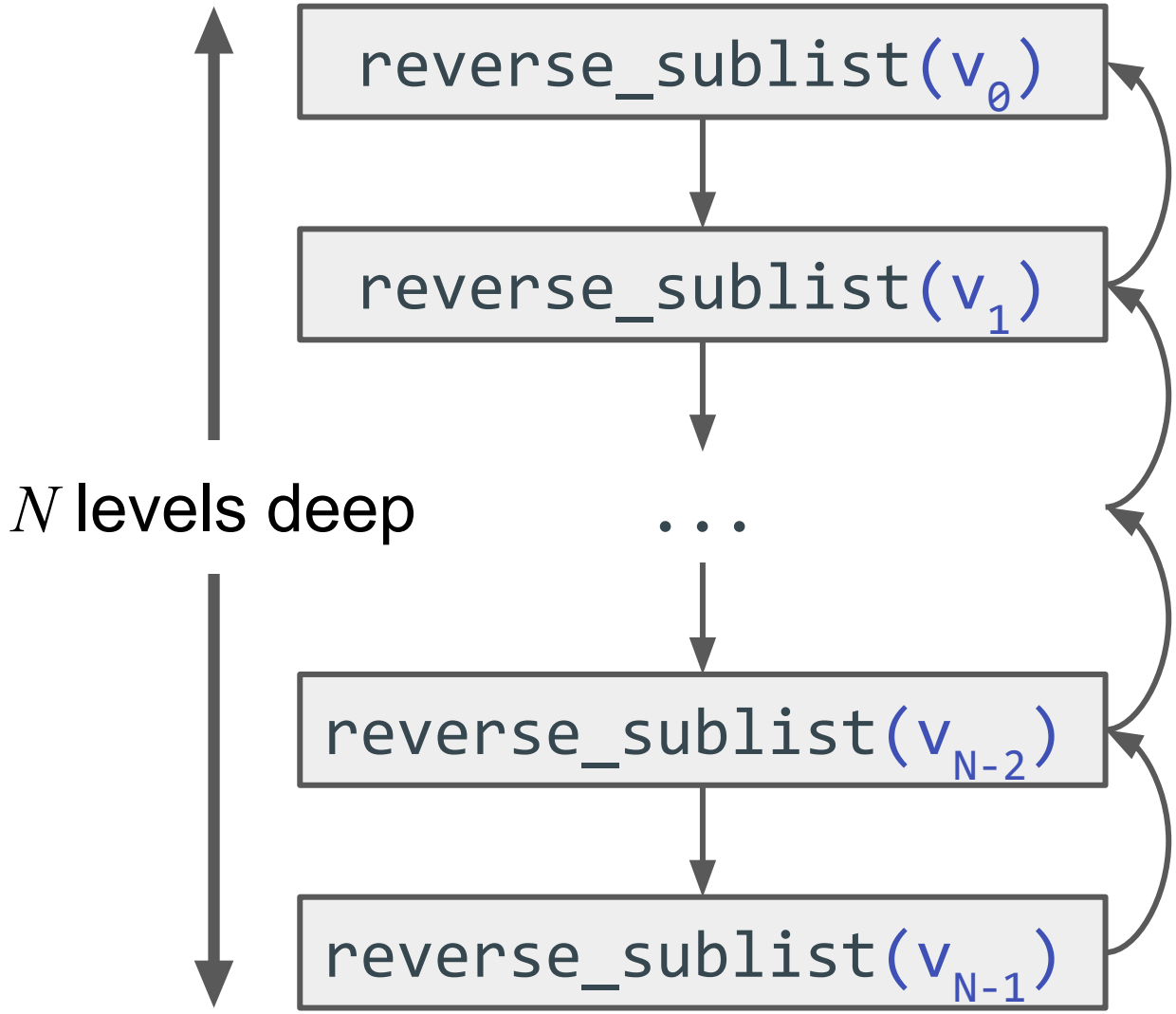
Reversing a SLL (recursion method)

```
/* Given sublist beginning at v, returns tail of the reversed sublist */
Vertex ReverseSubList(Vertex v) {
    /* Base case: sublist of 1 item is already reversed */
    if (v.next == null) return v;
    /* Recursive step */
    Vertex reversedEnd = ReverseSubList(v.next);
    /* Deferred operations */
    reversedEnd.next = v;     // Push v to rear of reversed sublist
    v.next = null;            // Make v the tail of sublist
    return v;                 // Return tail of the reversed sublist
}
void ReverseList() {
    ReverseSubList(head);
    swap(head, tail);
}
```

Reversing a SLL (recursion method)

Complexity analysis:

- **N** stack frames are maintained so space complexity is **O(N).**
- Each stack frame incurs constant time and is visited twice so time complexity is **O(2N)** which is **O(N).**



Reversing a SLL

Can we do this faster than $\mathbf{O(N)}$?

Reversing a SLL

Can we do this faster than $\mathbf{O(N)}$?

No! Why?

At least $\mathbf{N}$ items that need to change their next pointers.

Hence, time complexity is $\mathbf{\Omega(N)}$.

(Omega: at least this time complexity regardless of input)

Sorting a SLL

Would anyone like to share?

Describe your solution.

Sorting a SLL

Surprisingly can implement almost all we have learnt

- MergeSort(): almost the same as usual, and use create new SLL during merging process which takes **O(N).**

Sorting a SLL

Surprisingly can implement almost all we have learnt

- MergeSort(): almost the same as usual, and use create new SLL during merging process which takes **O(N).**
- (Randomized) Quick sort: Instead of pivoting on the original SLL, create a new SLL with the pivot initially. For each element in the original SLL, add to the head of the new SLL if it’s smaller than pivot, or to the tail otherwise. This still takes **O(N).**

Sorting a SLL

Can we do this faster than **O(N log N)**?

Sorting a SLL

Can we do this faster than $O(N \log N)$?

- No. It's O(N log N) lower bound for comparison based sorting algorithm.

Q3: Lisp Arithmetic Evaluator

Solving by hand

```
   ( + ( - 6 ) ( * 2 3 4 ) ( / 120 1 2 5) )
= ( + ( - 6 ) ( * 2 3 4 ) ( / 120 1 2 5) )
= ( + ( - 6 ) ( * 2 3 4 ) ( 12 ) )
= ( + ( - 6 ) ( * 2 3 4 ) ( 12 ) )
= ( + ( - 6 ) ( 24 ) ( 12 ) )
= ( 30 )
```

Modular Programming

Let's tackle the problem incrementally. We shall start with evaluating a single expression without any nested sub-expressions. For example:

```
<operator> 2.0 3.0 4.0 4.9 ...
```

We perform the operations on the list of operands using the operator.

Modular Programming

Now that our program can solve for simple expressions, how can we modify it to handle operands that are nested sub-expressions? For example:

```
<operator> 2.0 3.0 (...) 4.9 ...
```

Example

```
   ( + ( - 6 ) ( * 2 3 4 ( / 120 1 2 5) ) ) )
= ( + -6 ( * 2 3 4 ( / 120 1 2 5) ) )
= ( + -6 ( * 2 3 4 12 ))
= ( + -6 228 )
= 282
```

How do we evaluate these nested parentheses?

Algorithm: using 2 Stacks

Idea: Read left to right. Whenever) is seen, evaluate and replace the expression between last ( ).

Algorithm:

- Read input and push into stack A.
- Extract expressions by popping them into another stack B.
- Pop elements from B and evaluate.
- Push the result in A. Equivalent to replacing the last ( ) expression.

Example

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

Stack A

Stack B

Example

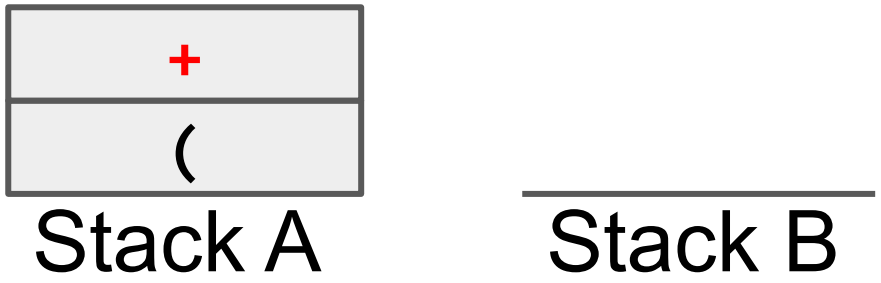
(+ (- 6) (* 2 3 4))

Token (is not closing parenthesis, so we push it into stack A.

Example

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

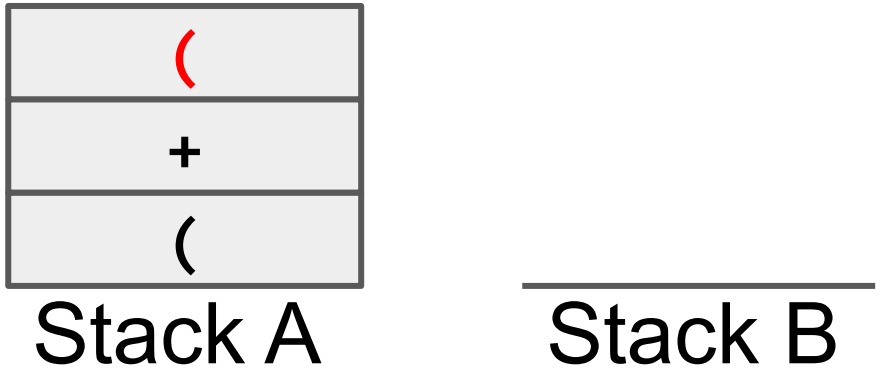
Token + is not closing parenthesis, so we push it into stack A.



Example

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

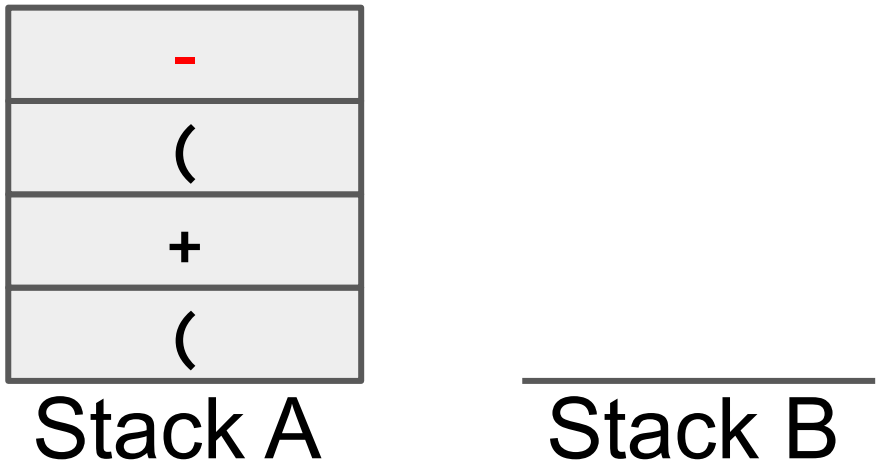
Token (is not closing parenthesis, so we push it into stack A.



Example

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

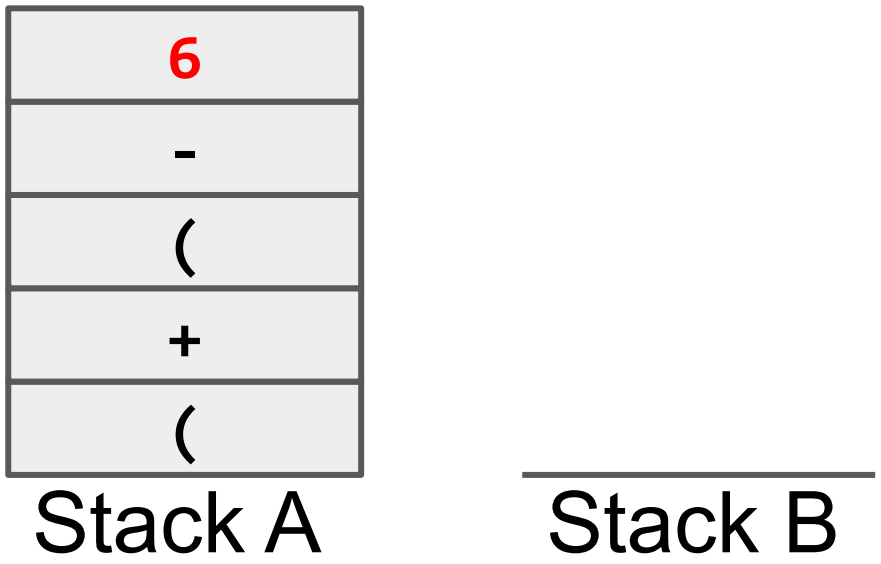
Token - is not closing parenthesis, so we push it into stack A.



Example

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

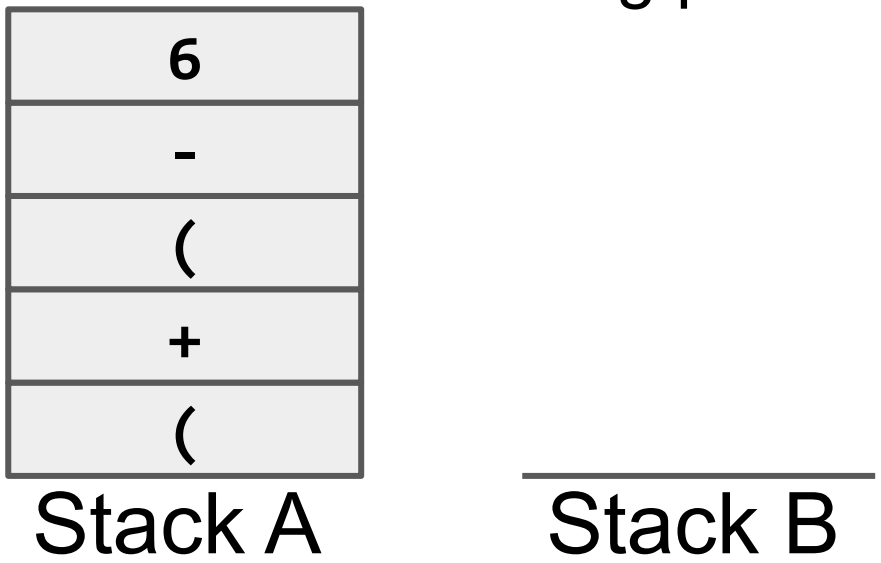
Token **6** is not closing parenthesis, so we push it into stack A.



Example

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

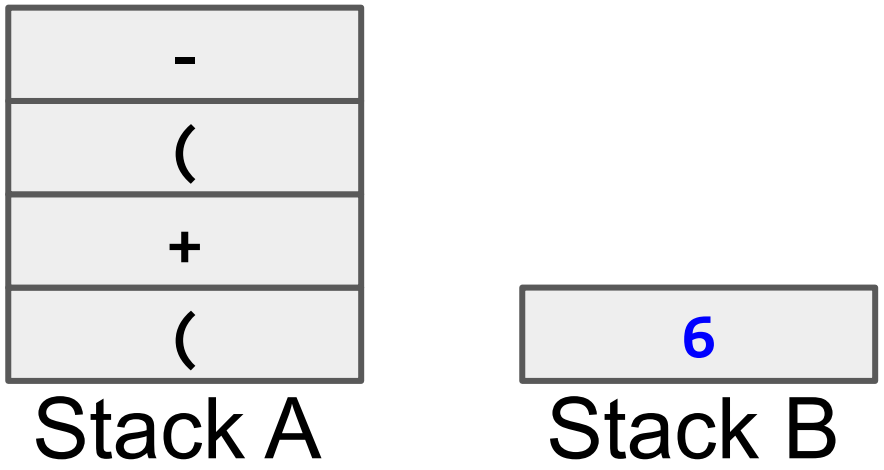
Encountered closing parenthesis! We are ready to evaluate a sub-expression so we will pop items from stack A into B until we encounter an opening parenthesis. You should convince yourself that it will be the matching parenthesis encapsulating the sub-expression!



Example

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

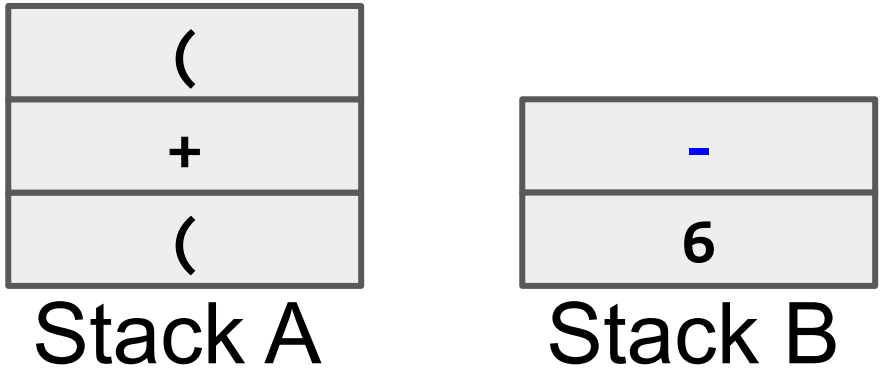
Pop from stack A. Token **6** is not opening parenthesis, so we push it to B.



Example

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

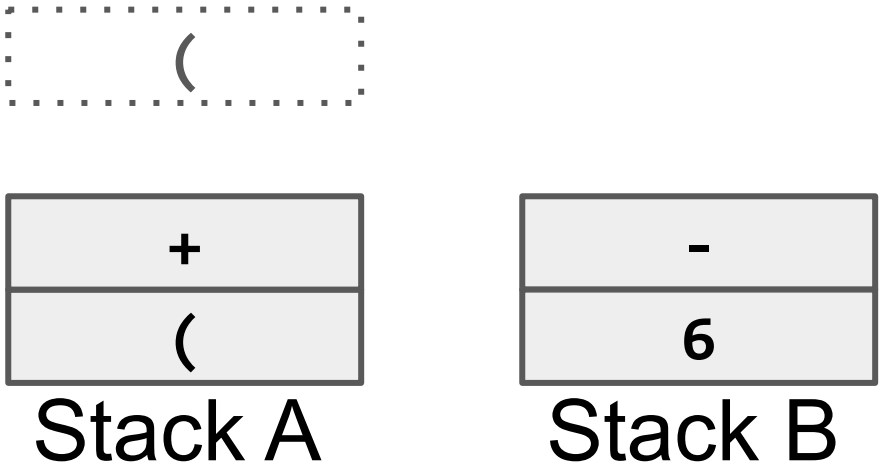
Pop from stack A. Token `-` is not opening parenthesis, so we push it to B.



Example

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

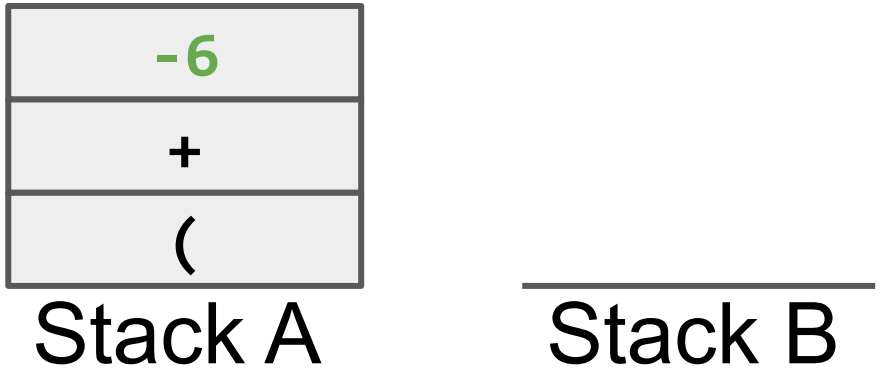
Pop from stack A. Encountered opening parenthesis! Stack B now contains a complete sub-expression ready for evaluation!



Example

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

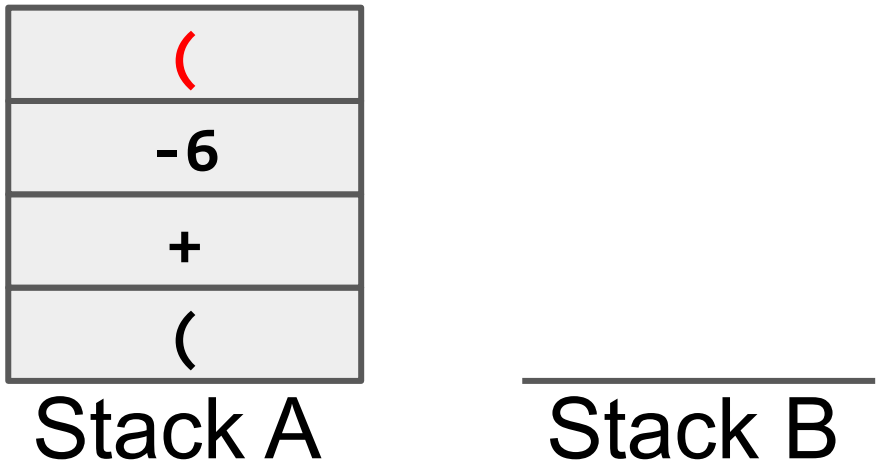
Sequentially pop everything from stack B, we recover the full sub-expression from left to right: `- 6` which is evaluated to be `-6`. There are still tokens left in the expression so we push this evaluated value back into stack A.



Example

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

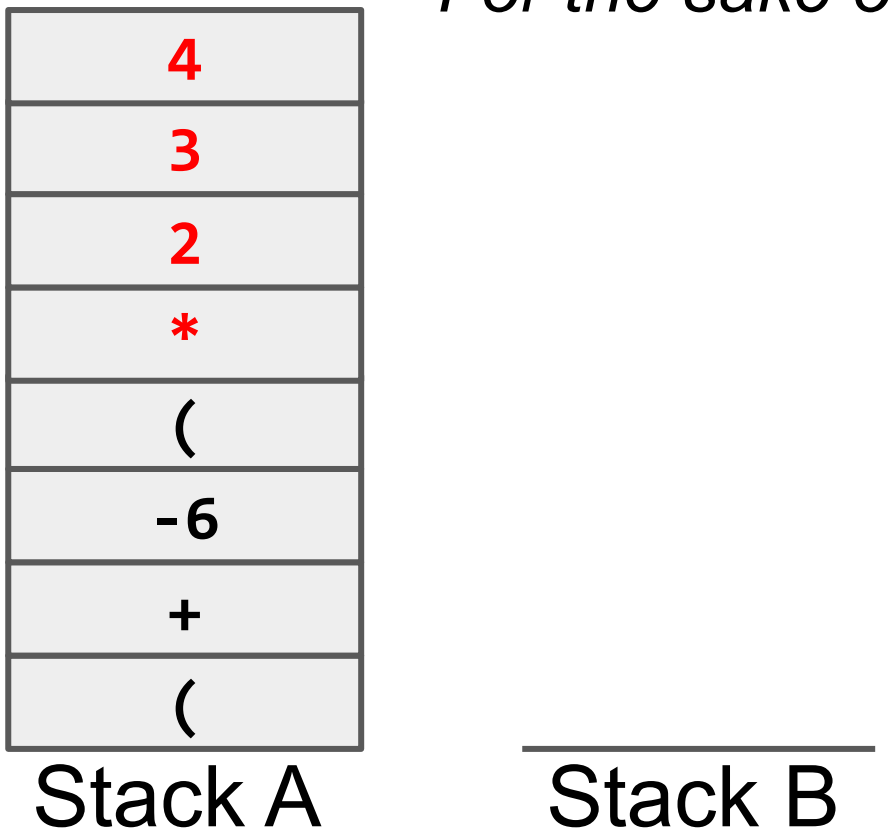
We continue from where we last left off in the expression. Token `(` is not closing parenthesis, so we push it into stack A.



Example

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

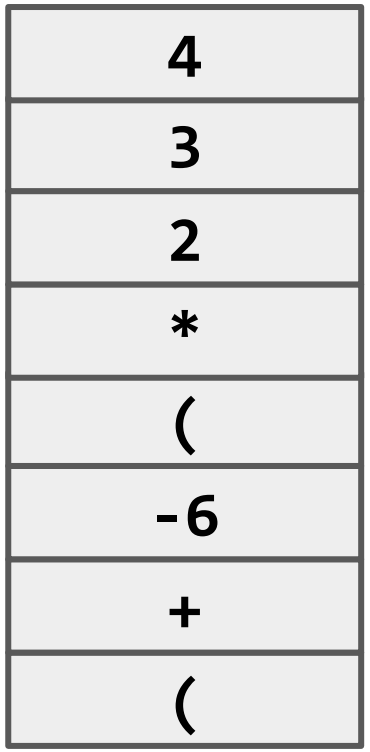
For the sake of brevity, we shall fast forward



Example

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

Encountered closing parenthesis in expression!
We will pop everything from stack A into B until opening parenthesis encountered



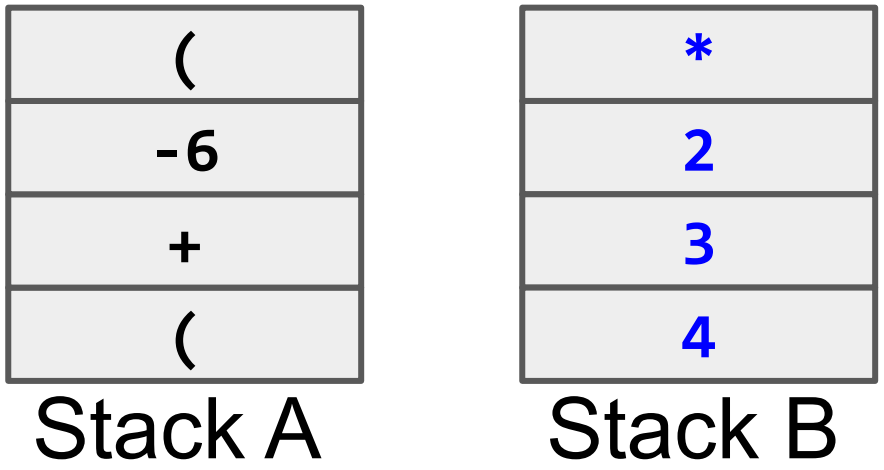
Stack A

Stack B

Example

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

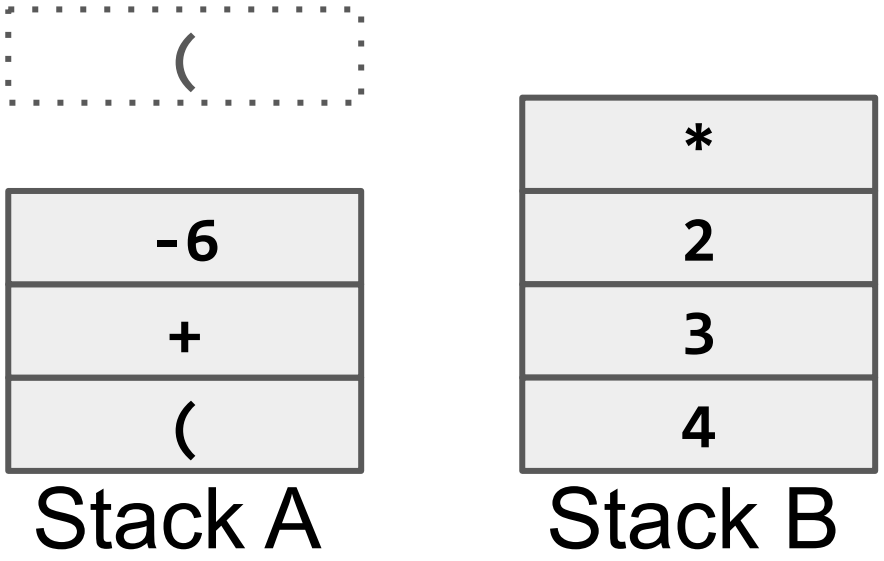
Fast forwarded



Example

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

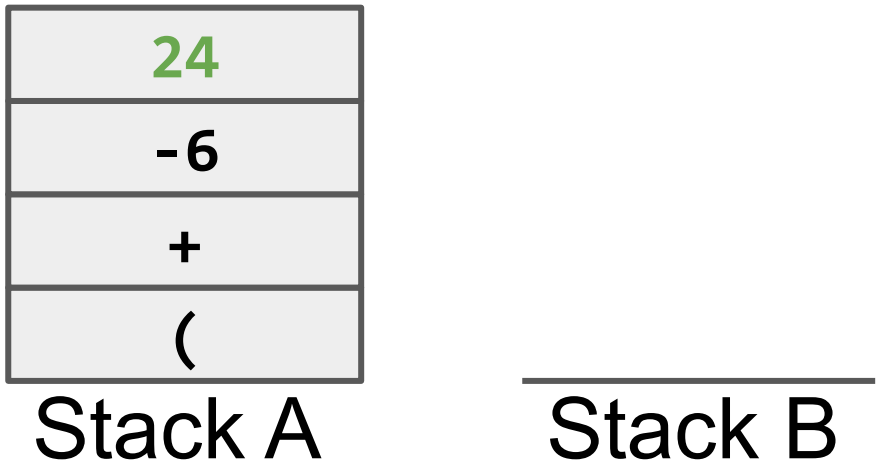
Encountered opening parenthesis in stack A, so we halt popping into B.
We are ready to evaluate sub-expression in stack B



Example

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

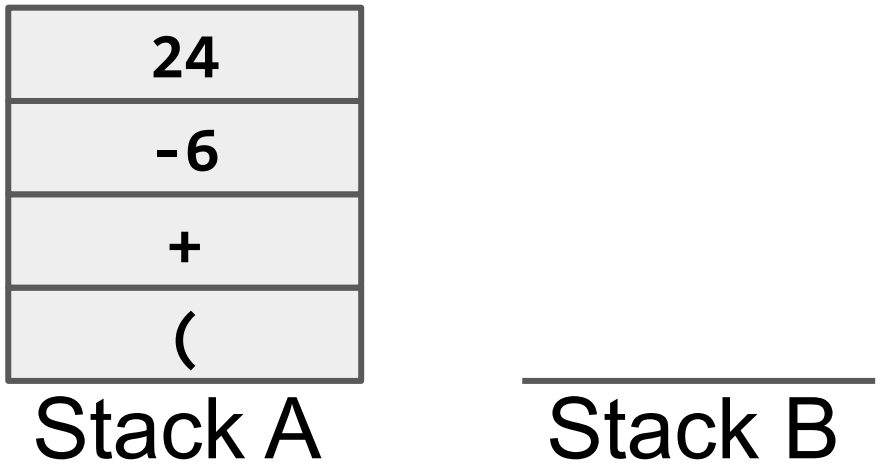
Sequentially pop everything in stack B, we recover sub-expression `* 2 3 4` which is evaluated to be `24`. There are still tokens left in the expression so we push this evaluated value back into stack A.



Example

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

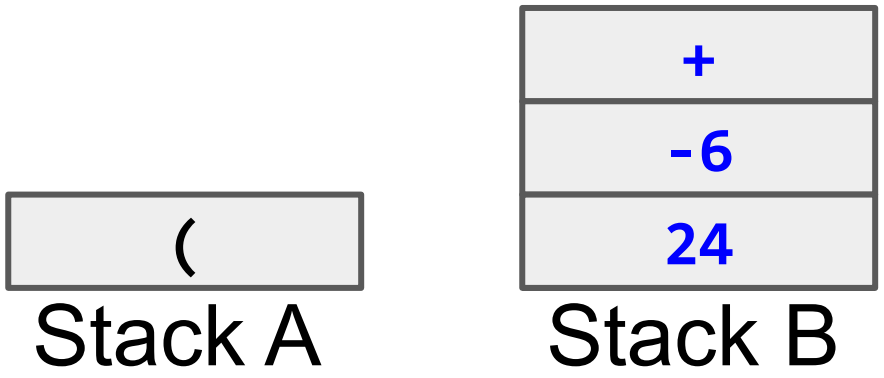
We continue from where we left off in the expression. Encountered closing parenthesis! We will pop everything from stack A into B until opening parenthesis encountered



Example

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

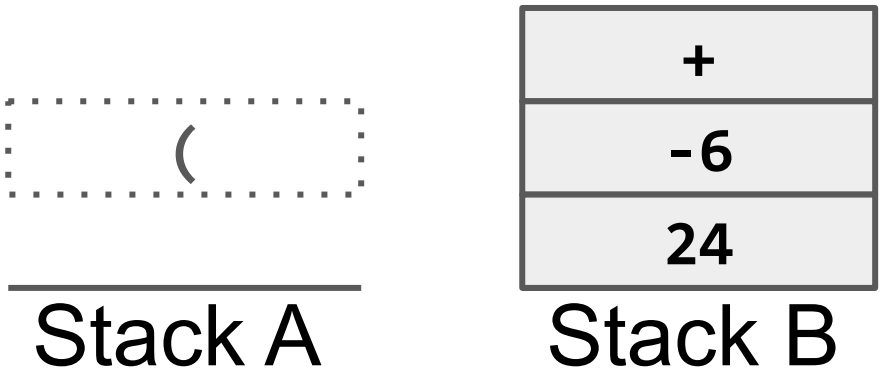
Fast forwarded



Example

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

Encountered opening parenthesis in stack A, so we halt pushing into B. We are ready to evaluate sub-expression in stack B **`+ -6 24`** which is evaluated to be **`18`**. However, since we have no more tokens left in the expression and stack A is now empty, this is the final expression to evaluate and so we return this result.



Attendance
Break
Questions

PS3 Discussion

PS3: /sim

Need a data structure that can accommodate:

- Go to the beginning
- Go to to the end
- Delete in the middle
- Move to the next element

PS3: /sim

Need a data structure that can accommodate:

- Go to the beginning
- Go to to the end
- Delete in the middle
- Move to the next element

Linked List!

PS3: /janeeyre

You are not expected to know the required technique to solve this problem yet.

This week try to get parsing the input right.

How to parse a line like this:

```
“A Book Name” 1234
```

PS3: /janeeyre

You are not expected to know the required technique to solve this problem yet.

This week try to get parsing the input right.

How to parse a line like this:

```
“A Book Name” 1234
```

Split line at “ character, then take second and last tokens.

Java LinkedList, Stack, Queue, Deque

Search to get the documentation

https://visualgo.net/training?diff=Medium&n=5&tl=5&module=list

If have time

Hands on Session: /joinstrings

Hands On: /joinstrings

- Given some strings. Concatenate them in a given order.
  - Issue: Java Strings are immutable.

Hands On: /joinstrings

- Given some strings. Concatenate them in a given order.
  - Issue: Java Strings are immutable.
- Hint 5 min: Store each partial string in a linked list fashion.
  - String i →String j → …

Hands On: /joinstrings

- Given some strings. Concatenate them in a given order.
  - Issue: Java Strings are immutable.
- Hint 5 min: Store each partial string in a linked list fashion.
  - String i →String j → …
- Hint 10 min: Each node will need to store:
  - The word.
  - The next Node.
  - The last word in this chain (for efficient merging).

Thank You!

Anonymous Feedback:
https://forms.gle/MkETeXdUT53Vhh896