Tutorial 03 - Linked List, Stack, Queue, Deque

CS2040C Semester 2 2018/2019

Adapted from slides by Ranald, AY1718 S1/S2 Tutor

Content review

Abstract Data Type (ADT)

- ADT is a type whose behavior is described by a set of value and operations
- Called “abstract” because operations are defined **independently of implementation**. i.e does not specify how data will be organized in memory, what algorithms to be used etc.
- The process of providing only the high level schematics and hiding the details is known as *abstraction*!

Data structure

- Data structures are implementation ADTs
- Various ADTs can be implemented using the same data structures

List ADT revisited

Common List ADT operations

`get(i)`	Gets the `i`-th element from the front. (0-indexed)
`search(v)`	Return the first index which contains `v`, or returns -1/`NULL` (to indicate failure)
`insert(i, v)`	Insert element `v` at index `i`.
`remove(i)`	Remove the element at index `i`.

Recall from tutorial 2

Stack ADT

Common Stack ADT operations

`push(v)`	Insert an element `v` at the top of stack
`pop()`	Remove and return the topmost item on stack. If stack is empty, return `NULL`
`peek()`	Return the topmost item on stack without removing it. If stack is empty, return `NULL`

Recall that stack is LIFO/FILO

Queue ADT

Common Queue ADT operations

`enqueue(v)`	Insert an element `v` at the rear of queue
`dequeue()`	Remove and return the frontmost item in queue. If queue is empty, return `NULL`
`peek()`	Return the frontmost item in queue without removing it. If queue is empty, return `NULL`

Recall that queue is FIFO/LILO

Double-ended Queue (Deque) ADT

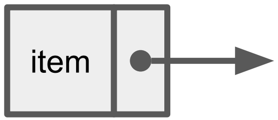
Common Deque ADT operations

`push_front(v)`	Insert an element `v` at front of deque
`push_back(v)`	Insert an element `v` at the rear of deque
`pop_front()`	Remove and return the frontmost item in deque. If deque is empty, return `NULL`
`pop_back()`	Remove and return rearmost item in deque. If deque is empty, return `NULL`
`peek_front()`	Return the frontmost item of deque without removing it. If deque is empty, return `NULL`
`peek_back()`	Return the rearmost item of deque without removing it. If deque is empty, return `NULL`

Singly vs Doubly Linked List

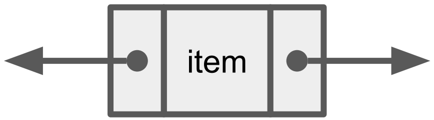
Singly Linked List (SLL) only has *next* pointers.

- Can only iterate *forward*



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

- Can iterate both *forward* and *backward*



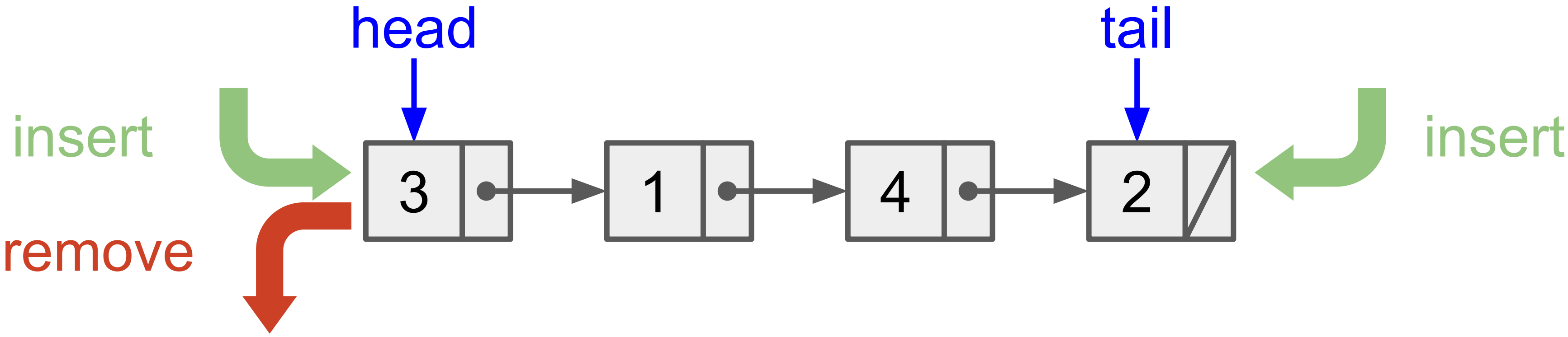
Singly/Doubly Linked List are **data structure implementations**, not **ADT!**

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 (*C++ STL implementation varies*)

Singly Linked List (SLL)

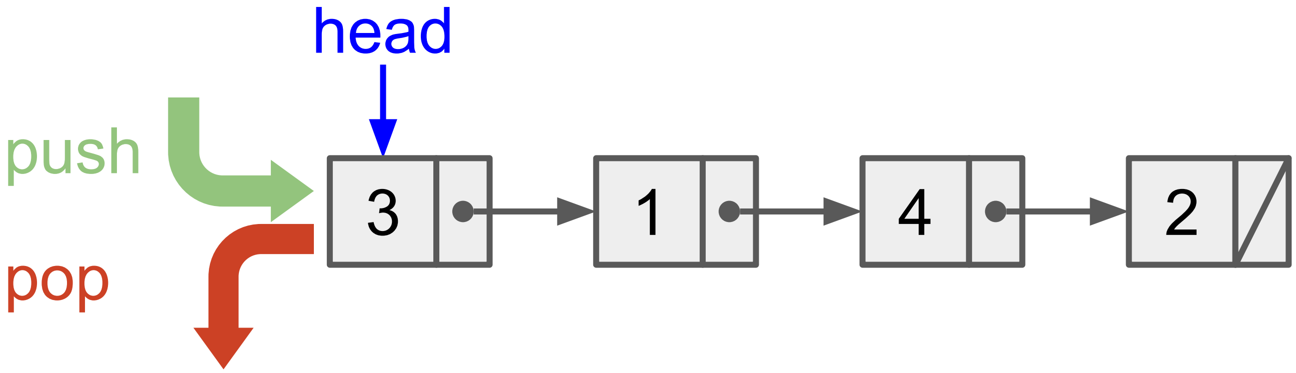
Below operations in *O(1)*



Stack (implemented via SLL)

Subset of List ADT operations

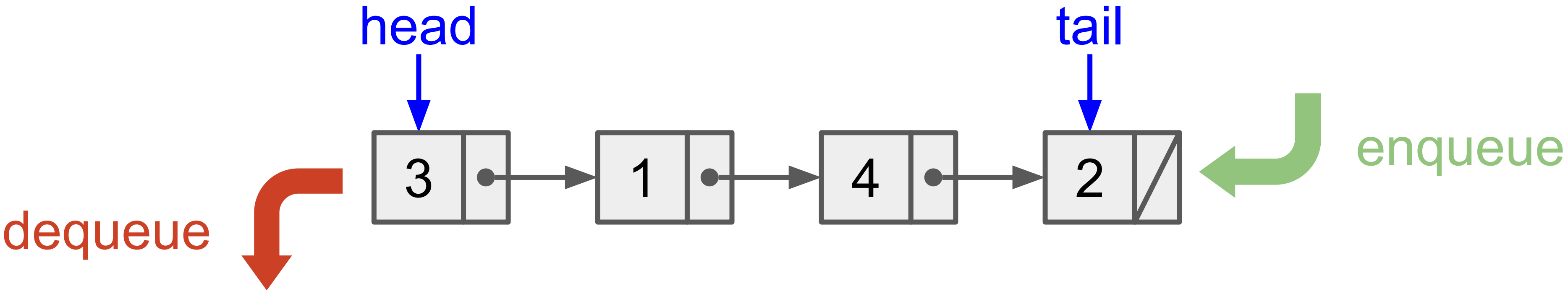
Below operations in $O(1)$



Queue (implemented via SLL)

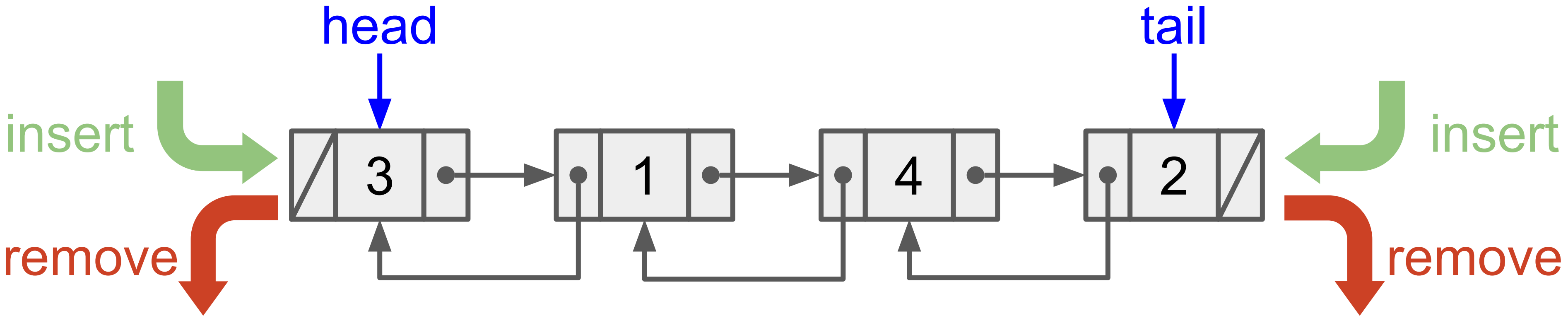
Subset of List ADT

Below operations in $O(1)$



Doubly Linked List (DLL)

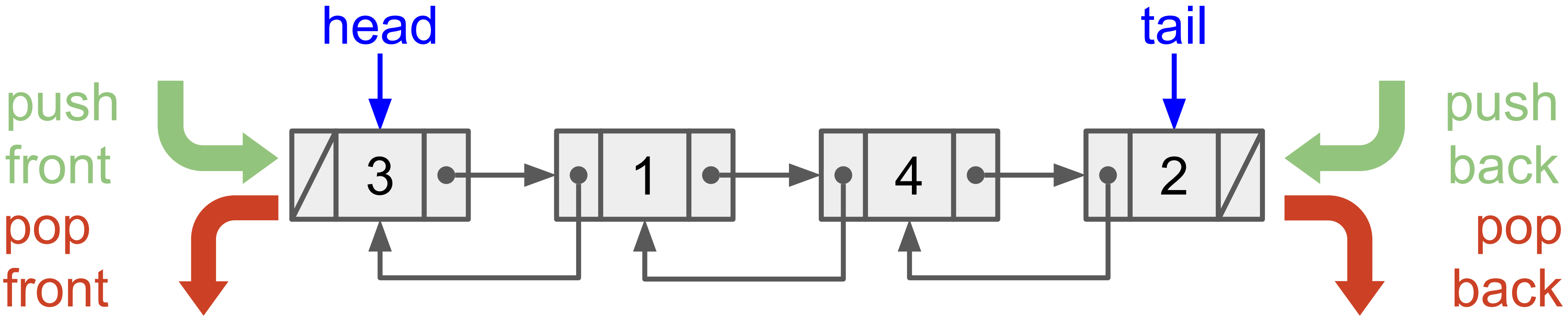
Below operations in $O(1)$



Deque (implemented via DLL)

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

Below 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	$O(N)$	

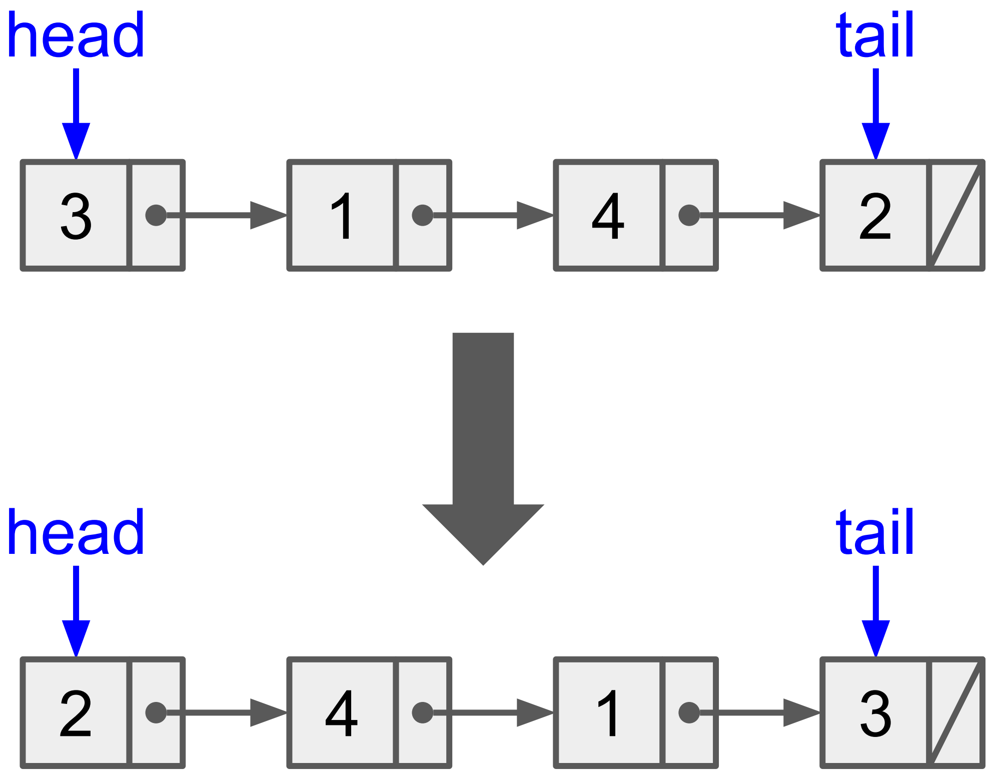
*: However this is allowed in C++ STL library and is $O(1)$

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()`

Reversing a SLL



Reversing a SLL

Would anyone like to share?

Describe your solution.

The rest: figure out the time and space complexities

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)

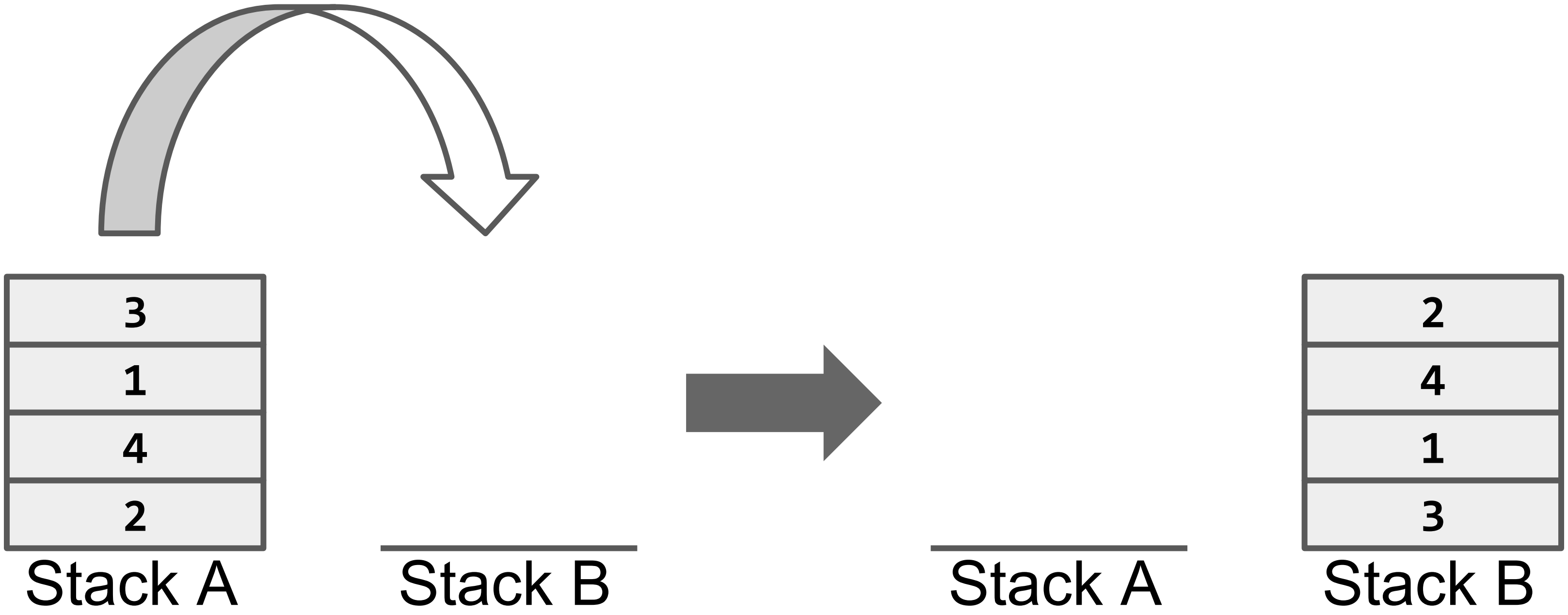
Let the original list be **A** and another empty list be **B**

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

Recurrence relation: Reversing a sublist starting at vertex `v` is the same as reversing a sub-list starting at vertex `v->next` then pushing `v` to the rear of that.

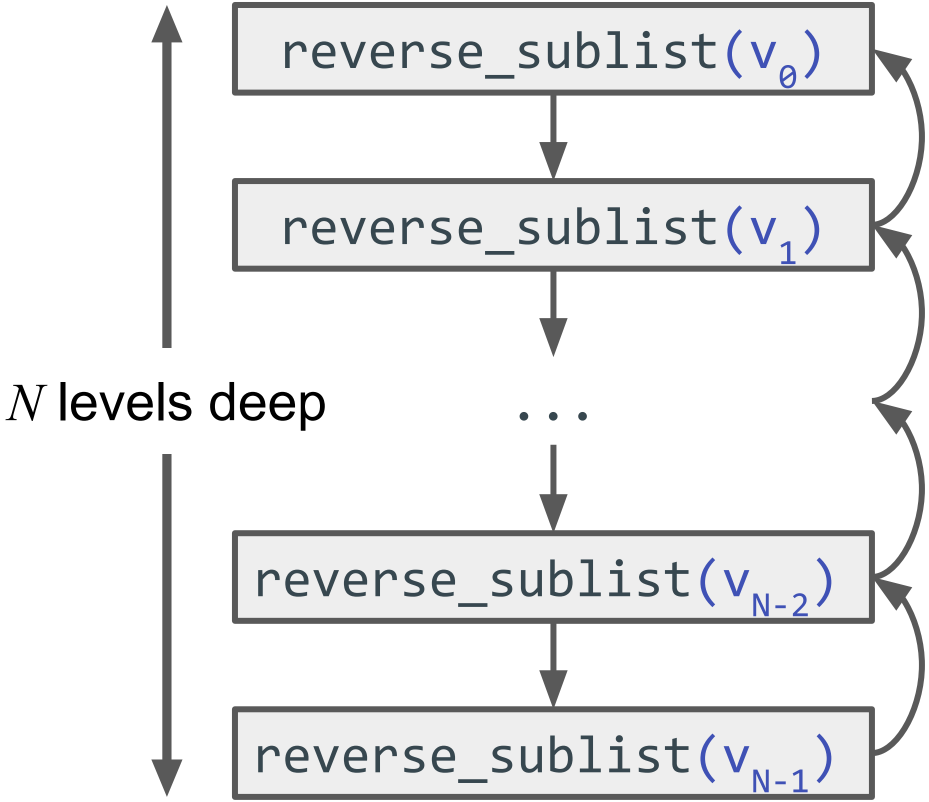
Reversing a SLL (recursion method)

```
/* Given sublist beginning at v, returns tail of the reversed sublist */
Vertex* reverse_sublist(Vertex* v) {
    /* Base case: sublist of 1 item is already reversed */
    if (v->next == NULL) return v;
    /* Recursive step */
    Vertex* reversed_last = reverse(v->next);
    /* Deferred operations */
    reversed_last->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 reverse_list() {
    reverse_sublist(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 (3 pointers method)

1. Declare 3 pointers: `curr` initialized at `head`, `bef` initialized at `NULL`, `aft` initialized at `NULL`
2. While `curr` is not `NULL`
   a. Use `aft` to save `curr->next`
   b. Set `curr->next` as `bef`
   c. Update `bef` as `curr`
   d. Update `curr` as `aft`
3. Swap `head` and `tail`

Complexity: $O(N)$ time, $O(1)$ space

Reversing a SLL (3 pointers method)

```
void reverse_list() {
    Vertex* curr = head, bef, aft;
    while(curr != NULL) {
        aft = curr->next;
        curr->next = bef;
        bef = curr;
        curr = aft;
    }
    swap(head, tail);
}
```

Reversing a SLL

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

Reversing a SLL

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

No! Why?

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

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

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

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 ( 12 ) ) )
= ( + ( - 6 ) ( 288 ) )
= 282
```

How do we evaluate these nested parentheses?

Algorithm: using 2 Stacks *only*

Process only the items between the last pair of parenthesis.

Input is from *left to right*: push into stack **A**.

Popping it would give us *right to left* order.

So we push into another stack **B**, only the items that are between the parenthesis.

Left to right order when we pop it out.

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.

(

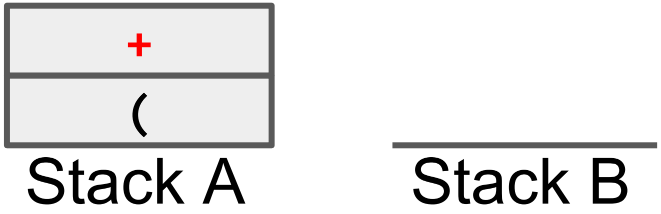
Stack A

Stack B

Example

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

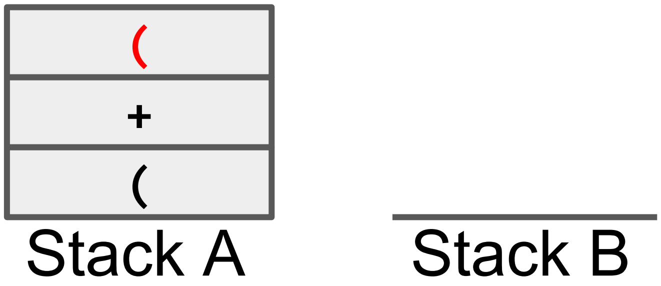
Token `+` is not closing parenthesis, so we push it into stack A.



Example

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

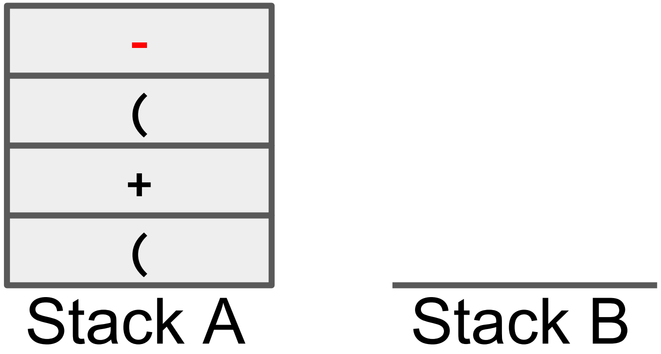
Token `(` is not closing parenthesis, so we push it into stack A.



Example

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

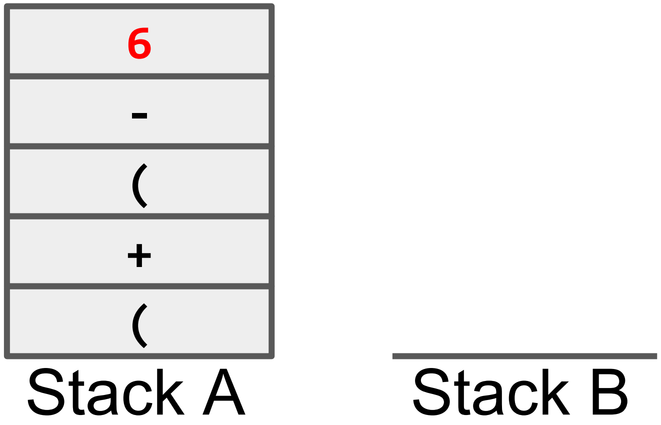
Token `-` is not closing parenthesis, so we push it into stack A.



Example

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

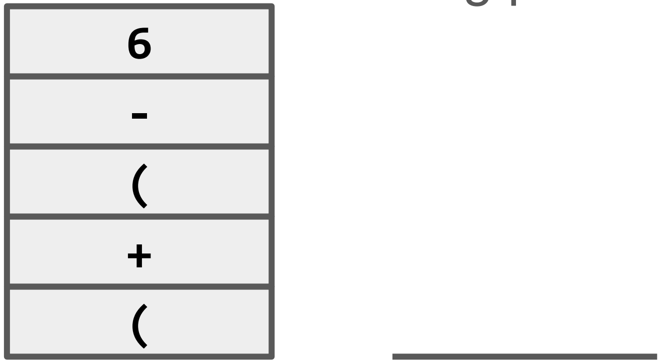
Token **6** is not closing parenthesis, so we push it into stack A.



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!

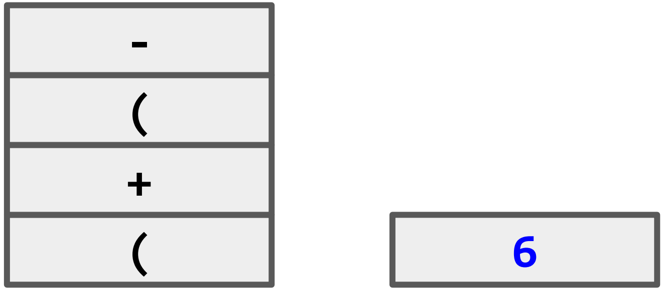


Stack A Stack B

Example

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

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

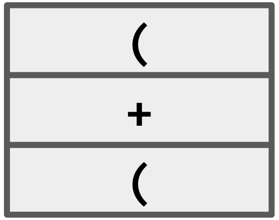


Stack A   Stack B

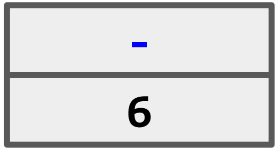
Example

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

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



Stack A

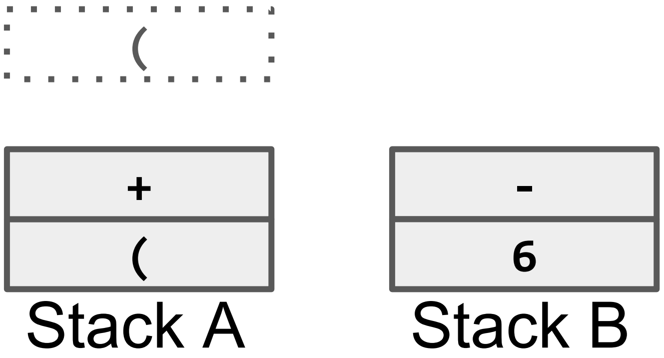


Stack B

Example

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

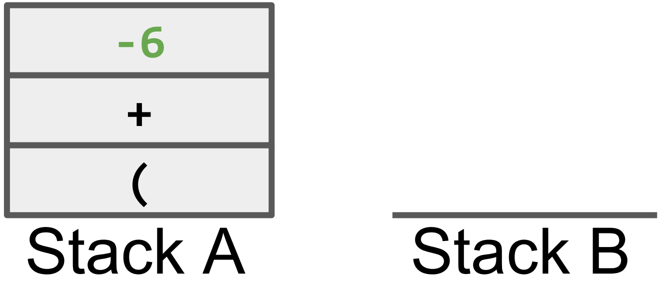
Pop from stack A. Encountered opening parenthesis! Stack B now contains a complete sub-expression ready for evaluation!



Example

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

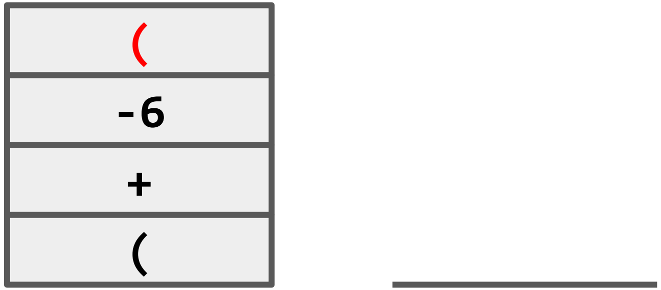
Sequentially pop everything from stack B, we recover the full sub-expression from left to right: `- 6` which is evaluated to be `-6`. There are still tokens left in the expression so we push this evaluated value back into stack A.



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.

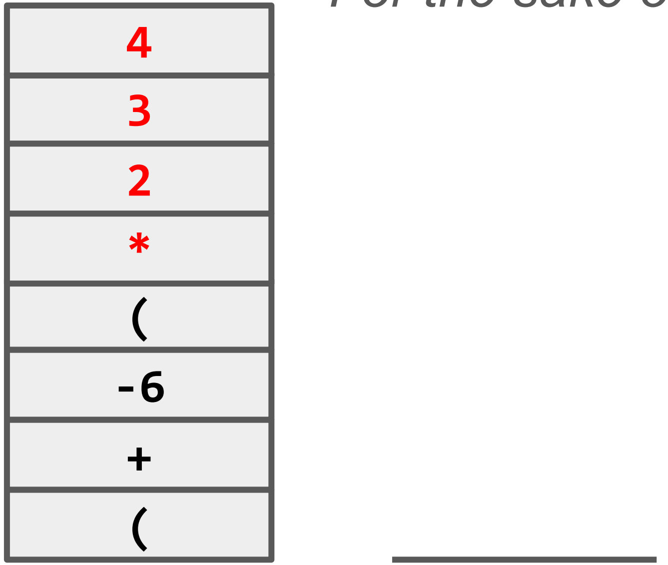


Stack A Stack B

Example

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

For the sake of brevity, we shall fast forward



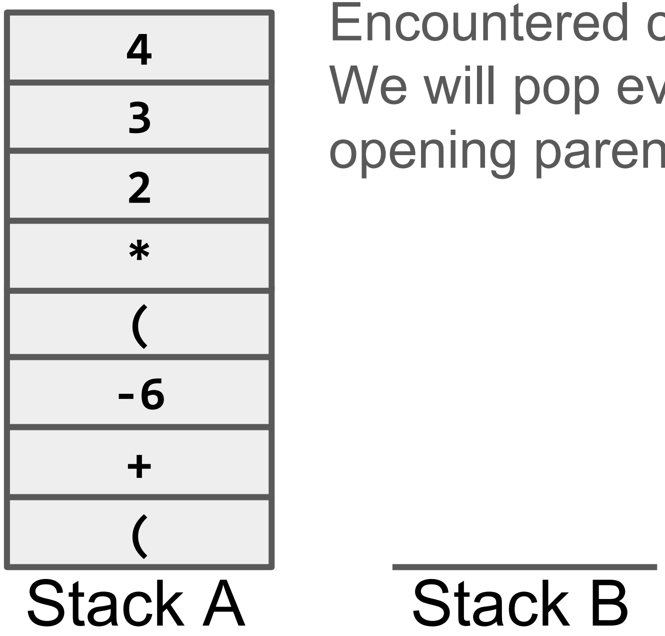
Stack A

Stack B

Example

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

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



Example

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

Fast forwarded

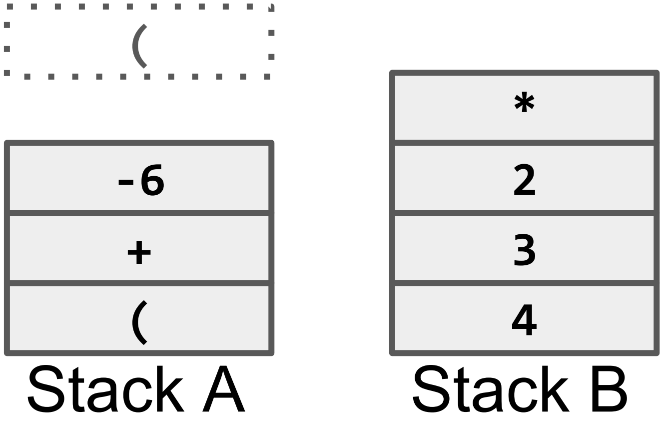
Stack A
(
-6
+
(

Stack B
*
2
3
4

Example

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

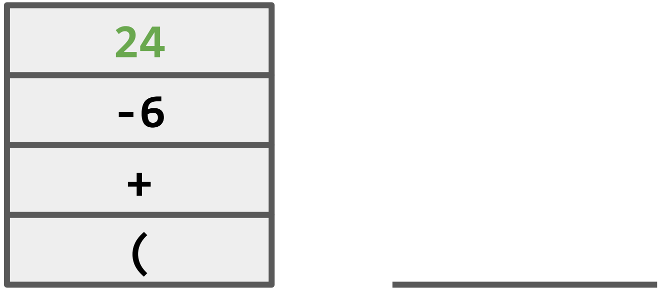
Encountered opening parenthesis in stack A, so we halt popping into B.
We are ready to evaluate sub-expression in stack B



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.



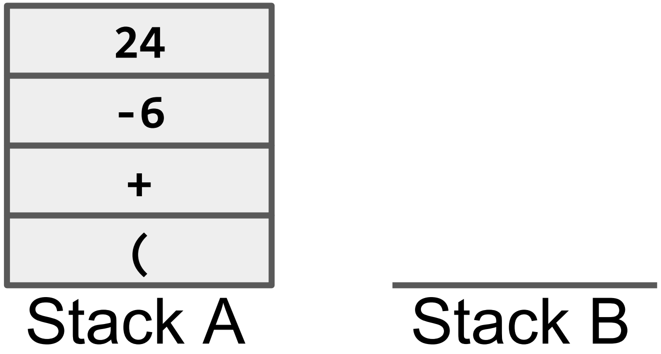
Stack A

Stack B

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

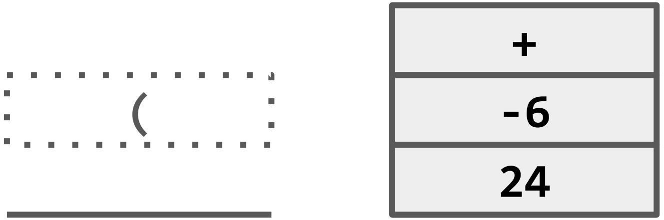
Stack A
(

Stack B
+
-6
24

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.



Stack A Stack B

Stack application

One important use of stacks is to process recursive problems such as linearly nested objects/patterns, as you have just seen.

Eg: Bracket matching

Is `[()([]{})({})]` a valid matched bracket?

What about `[()([]{)(})]`?

Q4: Exercises

Backspace

Sample Input 1

```
a<bc<
```

Sample Output 1

```
b
```

Sample Input 2

```
foss<<rritun
```

Sample Output 2

```
forritun
```

Backspace

What *operations* do we need?

- Insert? (at where?)
- Delete? (at where?)
- Access (iterate through? Random access?)

What data structures can we use?

Backspace

What data structures can we use?

Many!

Which is easier to implement? :D

Broken Keyboard

Sample Input

```
This_is_a_[Beiju]_text
[[]][][]Happy_Birthday_to_Tsinghua_University
```

Sample Output

```
BeijuThis_is_a__text
Happy_Birthday_to_Tsinghua_University
```

Broken Keyboard

What *operations* do we need?

- Insert? (at where?)
- Delete? (at where?)
- Access (iterate through? Random access?)

What data structures can we use?

Broken Keyboard

What data structures can we use?

List

Can we use other data structures?

Why / why not?