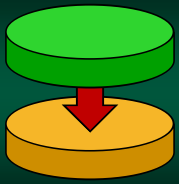
Queues & Stacks in Practice

Section 1.3

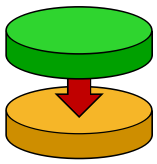


The System Stack and Heap

How it works? …um, I forgot

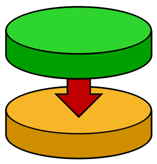
The System Stack and Heap

- Your computer maintains two distinct types of memory for running programs: the stack and heap
- The stack is used to …
  - store function states
  - this includes local variables



The Heap

- *Heap* is used to store *dynamic* allocation
- … not to be confused with the Heap Data Structure (which we will cover later)



The Heap

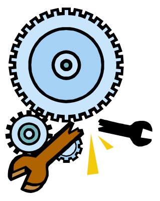
- Anytime you create objects using "new"…
  - the heap is used to allocate storage
  - system performs garbage cleanup after the memory is no longer needed
- Unlike the stack, data persists regardless of function calls

Garbage Collection

- Programming languages use *garbage collection* reclaim unused data from the heap
- Policy is to reclaim the memory used by objects that *can no longer be accessed (i.e. no references)*

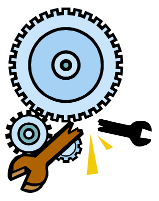
Loitering

- It is possible to "remove" an item from the ADT, but accidently keep a reference (link) to it
- The item is effectively an *orphan* - it will be never be accessed again by the ADT

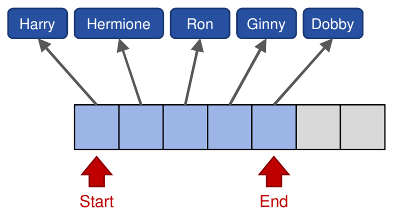


Loitering

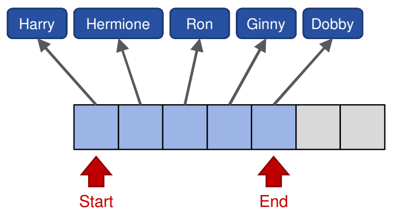
- The garbage collector has no way to know unless it's overwritten
- So, under this condition, the object is said to *loiter* – stay in memory with <u>no</u> purpose



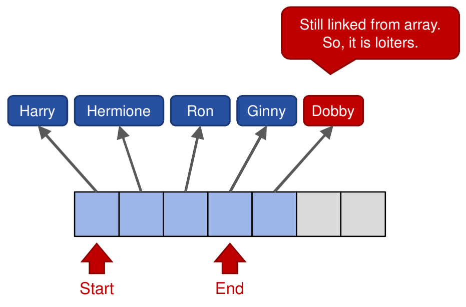
Array Storing a List (partially filled)

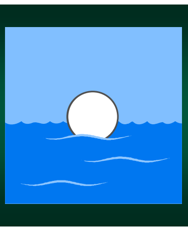


Delete Last – Move End



Dobby is still linked…





Cursors

Okay, now its getting weird

Cursors

- *Cursors* are a melding of the idea of arrays and linked lists
- Cursors want to minimize the constant creation and deletion of new nodes
- So, it maintains an array of unused nodes

Cursors

- Multiple nodes are allocated early - called a *pool*
- If a node is needed, one is removed from the pool
- If a node is removed, and the array has room, it is placed back in the array

The Reason

- Arrays can be wasteful …
  - in space – when there are partially filled arrays
  - in time – created and destroyed frequently
- Linked lists can be wasteful…
  - require memory to be allocated each time a node is created
  - puts a lot of work on the heap

Even more approaches

- You can also use another "pool" linked list
- So, your Linked List class
  - would have a linked list of valid nodes
  - and another list of unused notes
  - the danger here is that you don't limit the size of the pool – and it grows ***forever***
  - so, if you use two linked lists, keep a pool member count

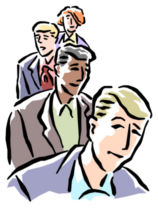
Queues & Stacks in Practice

1001 Uses!

(I meant 1,001 – not 9)

HTML Tag Matching

- HTML is a hierarchical structure
- HTML consists of tags
  - each tag can also embed other tags
  - allows text to be aligned, made bold, etc…



HTML Tag Matching

- Web browsers read the text and apply a tag depending if it is active
- They maintain a stack…
  - push a start tag, pop and end tag
  - if the HTML is correct, they should match
  - *… with the exception of the unary tags*

HTML Tag Matching

```
<html>
<body>
<center>
<h1>Banks of Sacramento</h1>
</center>
A bully ship and a bully crew,<br>
<i>Doo-da! Doo-da!</i><br>
A bully mate and a captain
too,<br>
<i>Doo-da! Doo-da-day!</i><br>
<i>And it's blow, ye winds,
blow,<br>
For Californi-o.<br>
For there's plenty of gold,<br>
So I've been told,<br>
On the banks of the
Sacramento.</i><br>
</body>
</html>
```

Banks of Sacramento

A bully ship and a bully crew,
Doo-da! Doo-da!
A bully mate and a captain too,
Doo-da! Doo-da-day!

Then blow, ye winds, blow,
For Californi-o.
For there's plenty of gold,
So I've been told,
On the banks of the Sacramento.

Balanced Parentheses

- When analyzing arithmetic expressions…
  - it is important to determine whether it is balanced with respect to parentheses
  - otherwise, the expression is incorrect
- A great solution is a stack
  - push each `(` and pop each `)`
  - at the end, the stack should be empty
  - also, if you attempt to pop on an empty stack, the expression is invalid

Balanced Parenthesis Examples

`(a + b)`	Balanced
`(a + b))`	Pop empty stack
`) a + b (`	Pop empty stack
`(a + (b + 1) * c) / e`	Balanced
`(a * (b + ((d + e) * f))`	Stack has 1 left

Balanced Parentheses

- *But wait…*
  - can we just use a "parenthesis level" counter?
  - if it is `>= 1` at the end or it ever is `< 0`, the expression is invalid
- Sorry, it won't work
  - some expressions allow `{ }` and `[ ]`
  - a simple counter is insufficient
  - stack can check if the pop'd item matches

Balanced Parenthesis Examples

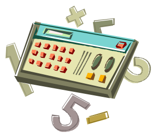
`[a + b]`	Balanced
`(a + b}`	Mismatch
`{[ a + b })`	Mismatch
`(a + (b + 1) * c / e`	Unbalanced
`(a * [b + {c + d} * e])`	Balanced

Evaluating Expressions

A Stack and Queue working together!

Evaluating Expressions

- It is a common task in programs to evaluate mathematical expressions and get a result
- Computers can perform this task using an algorithm *created by Dijkstra*, but we will get into that later



Evaluating Expressions

- First, we need to look at mathematical expressions
- We commonly using infix notation which is not stack or queue "friendly"
  - there are, however, two alternative notations
  - *one of which is stack friendly*

Infix Notation

- Using *infix notation*, we put the operating in between the two operators
- This is the standard format used today

To add the numbers *a* and *b*, we type:	`a + b`
To divide *a* by *b*, we type:	`a / b`

Prefix Notation

- *Prefix notation*, rather than putting the operator between the operands, puts it first
- It is also called *"Polish Notation"*
- Used by the LISP programming language

To add the numbers *a* and *b*, we type:	`+ a b`
To divide *a* by *b*, we type:	`/ a b`

Postfix Notation

- *Postfix notation* puts the operator at the end
- Also called *"Reverse Polish Notation" (RPN)*
- Since the operator is last, we can also use it as a "trigger" to perform math

To add the numbers *a* and *b*, we type:	`a b +`
To divide *a* by *b*, we type:	`a b /`

Where are My Parenthesis?

Infix	Prefix	Postfix
`a + b * c`	`+ a * b c`	`a b c + *`
`(a – b) * c`	`– a b * c`	`a b – c *`
`(a / (b – c) + d)`	`+ / a – b c d`	`a b c – / d +`
`(a + b / (c – d))`	`+ a / b – c d`	`a b c d – / +`

Where are My Parenthesis?

- Infix is the only notation that needs parentheses to change precedence
- The order of operators handles precedence in prefix and postfix

Converting to Prefix or Postfix

- Why are learning this... *be patient!*
- Converting from infix to postfix or prefix notation is easy to do by hand
- Did you notice that the operands did not change order? They were always *a, b, c…*
- We just need to rearrange the operators

Convert Infix to Prefix / Postfix

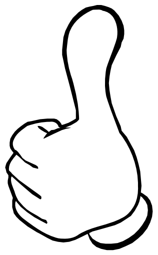
1. Make it a *fully parenthesized expression (FPE)* - one pair of parentheses enclosing each operator and its operands
2. Move the operators to the start (prefix) or end (postfix) of each sub-expression
3. Finally, remove all the parenthesis

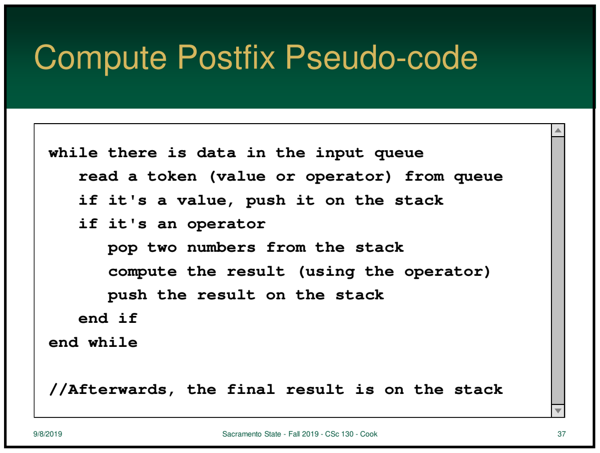
Infix to Postfix

```
   a / (b – c) + d
1. ((a / (b – c)) + d)
2. ((a (b c – ) / ) d + )
3. a b c – / d +
```

Compute Postfix Algorithm

- Computing a postfix expression is unbelievably easy
- All you need is:
  - one queue of values & operators
  - and one stack
- In fact, on classic Hewlett Packard calculators, all operations are stack based



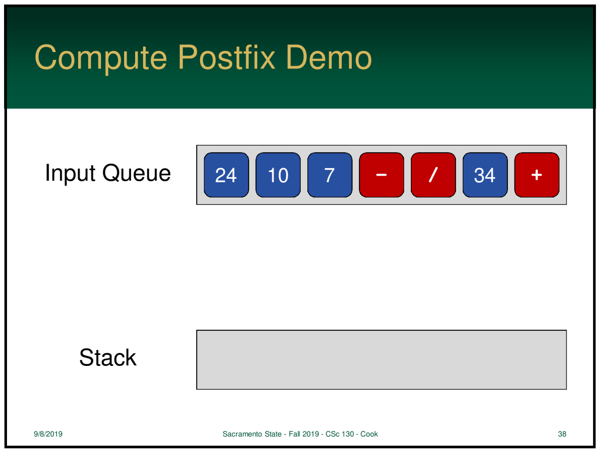


Compute Postfix Pseudo-code

```
while there is data in the input queue
   read a token (value or operator) from queue
   if it's a value, push it on the stack
   if it's an operator
      pop two numbers from the stack
      compute the result (using the operator)
      push the result on the stack
   end if
end while

//Afterwards, the final result is on the stack
```

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 37

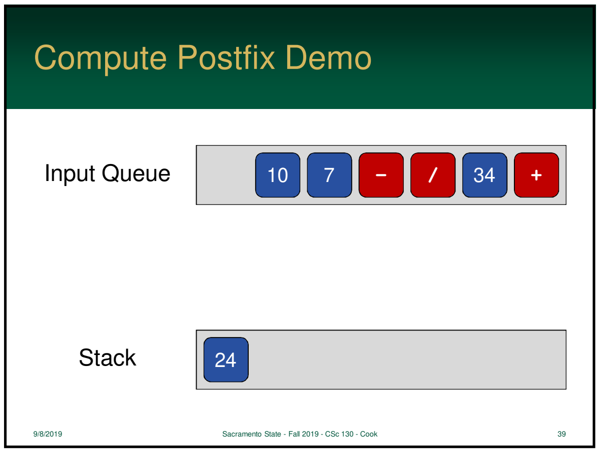


Compute Postfix Demo

Input Queue: 24 10 7 − / 34 +

Stack:

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 38

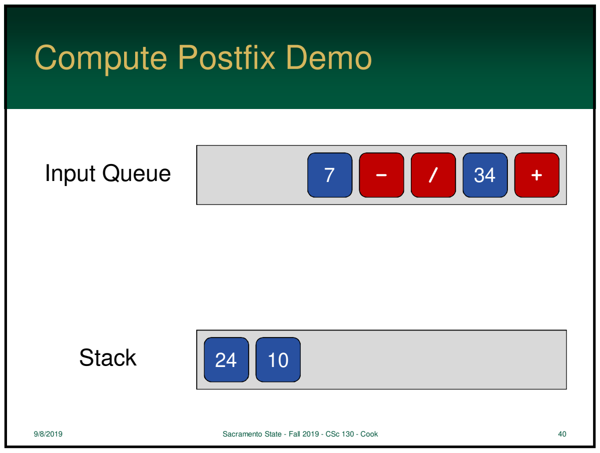


Compute Postfix Demo

Input Queue: 10 7 − / 34 +

Stack: 24

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 39

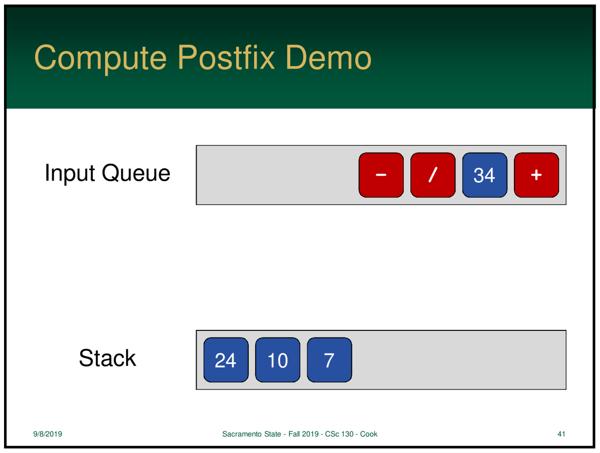


Compute Postfix Demo

Input Queue: 7 − / 34 +

Stack: 24 10

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 40

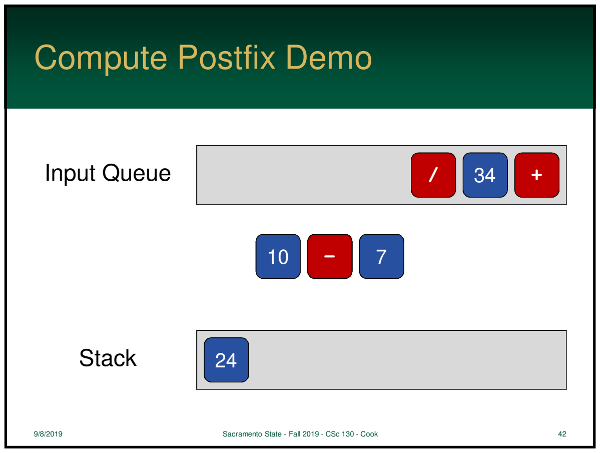


Compute Postfix Demo

Input Queue: − / 34 +

Stack: 24 10 7

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 41



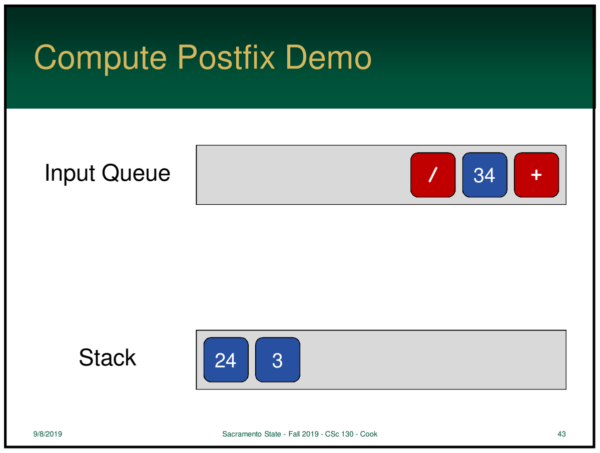
Compute Postfix Demo

Input Queue: / 34 +

10 − 7

Stack: 24

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 42

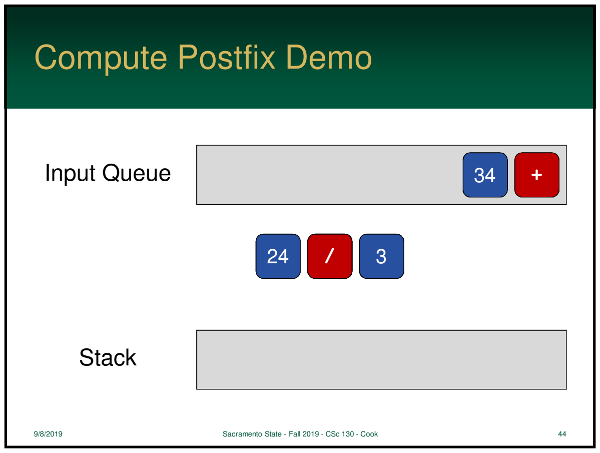


Compute Postfix Demo

Input Queue: / 34 +

Stack: 24 3

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 43



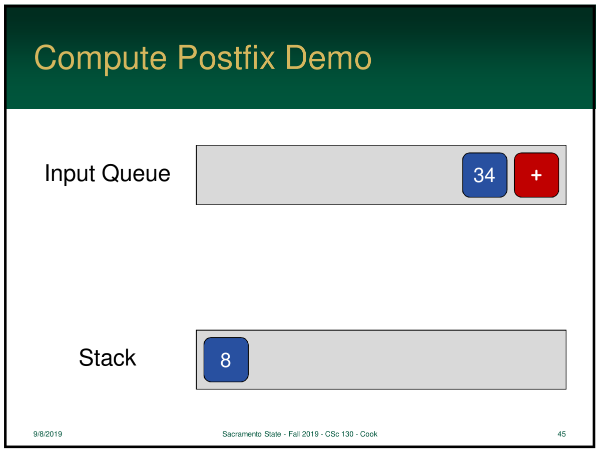
Compute Postfix Demo

Input Queue: 34 +

24 / 3

Stack:

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 44

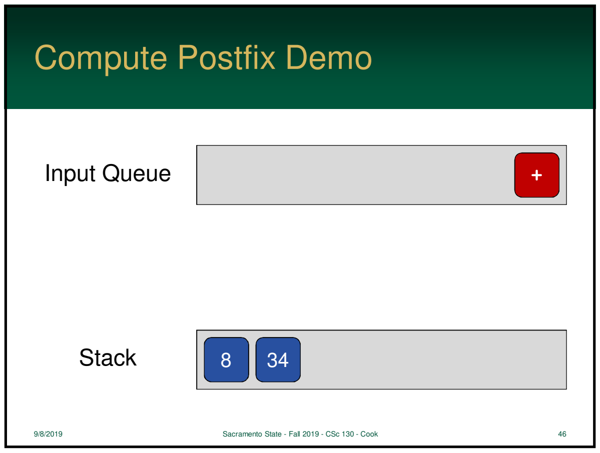


Compute Postfix Demo

Input Queue: 34 +

Stack: 8

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 45

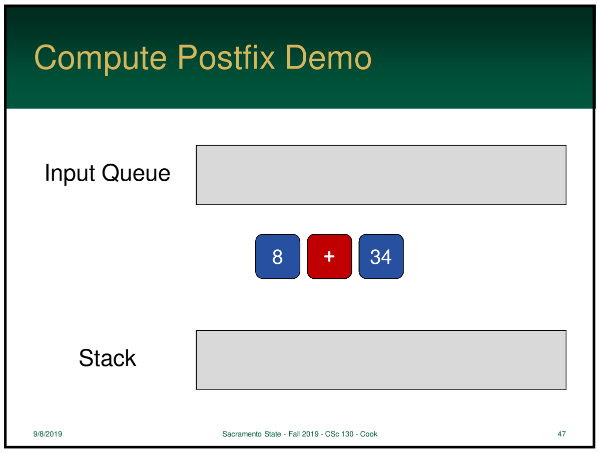


Compute Postfix Demo

Input Queue: +

Stack: 8 34

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 46



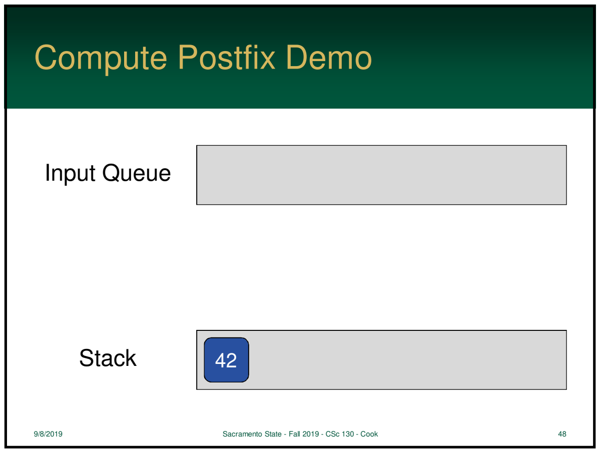
Compute Postfix Demo

Input Queue:

8 + 34

Stack:

9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 47



Compute Postfix Demo

Input Queue:

Stack: 42

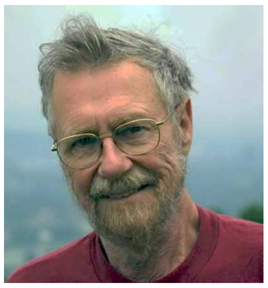
9/8/2019 Sacramento State - Fall 2019 - CSc 130 - Cook 48

Infix to Postfix Algorithm

Let the computer do the work…

Edsger Dijkstra

- Edsger Dijkstra is a World famous computer scientist
- He invented a wealth of algorithms that we use today
- For his contributions, he received the Turing Award



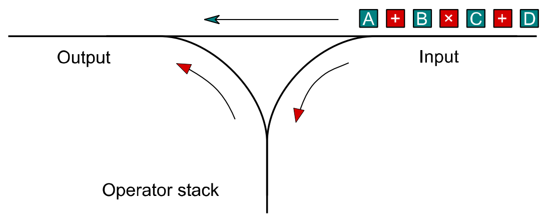
Infix to Postfix Algorithm

- Infix expressions need to be converted to postfix to be evaluated
- Dijkstra's *Shunting-yard algorithm* performs this task

Shunting-yard algorithm

- Named after railroad shunting yards – which move trains onto different tracks
- Dijkstra's solution uses an input queue, operator stack, and output queue

Shunting-yard Algorithm



Shunting-yard Algorithm

- The most basic version of this algorithm requires *Fully-Parenthesized Expression (FPE)*
- This means, there is no precedence and parenthesis are put around every operator

FPE Shunting-yard Algorithm

```
while the input queue has tokens
   read a token from the input queue
   if the token is a…
      operand : add it to output queue
      operator : push it on the stack
      '(' : push it onto the stack
      ')' :
         while the top of stack isn't a '('
            pop an operator
            add it to the output queue
         end while
         pop and discard the extra '('
   end if
end while
```

FPE Shunting-yard Algorithm

Input Queue	((a * (b + c)) / d)
Operator Stack	
Output Queue	

FPE Shunting-yard Algorithm

Input Queue	(a * (b + c)) / d)
Operator Stack	(
Output Queue	

FPE Shunting-yard Algorithm

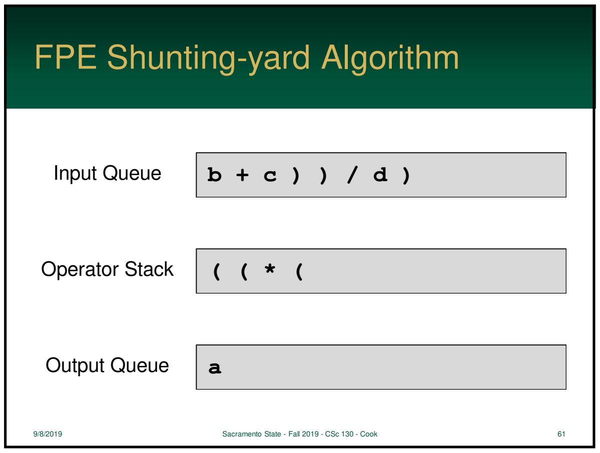
Input Queue	a * (b + c)) / d)
Operator Stack	((
Output Queue	

FPE Shunting-yard Algorithm

Input Queue	* (b + c)) / d)
Operator Stack	((
Output Queue	a

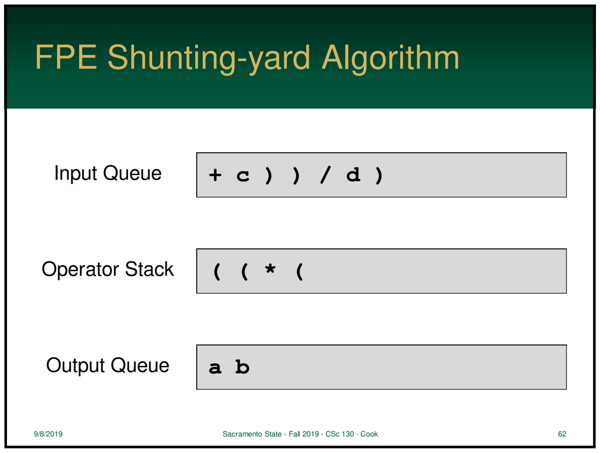
FPE Shunting-yard Algorithm

Input Queue	(b + c)) / d)
Operator Stack	((*
Output Queue	a



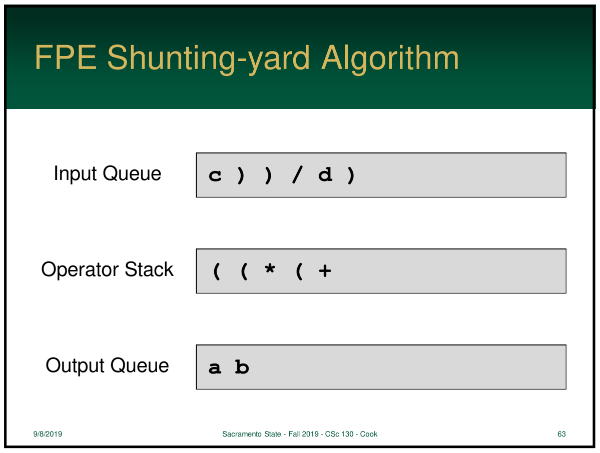
FPE Shunting-yard Algorithm

Input Queue	`b + c ) ) / d )`
Operator Stack	`( ( * (`
Output Queue	`a`



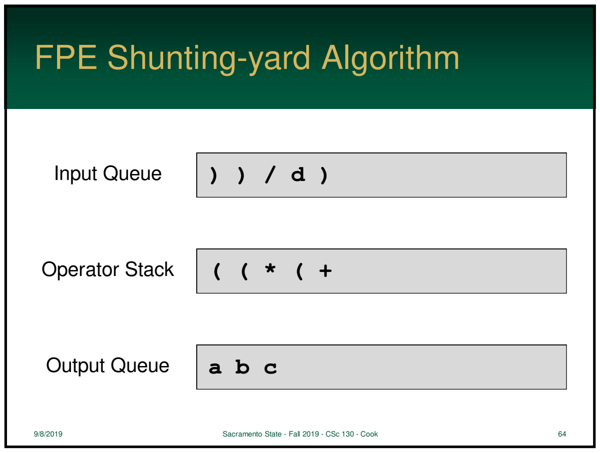
FPE Shunting-yard Algorithm

Input Queue	`+ c ) ) / d )`
Operator Stack	`( ( * (`
Output Queue	`a b`



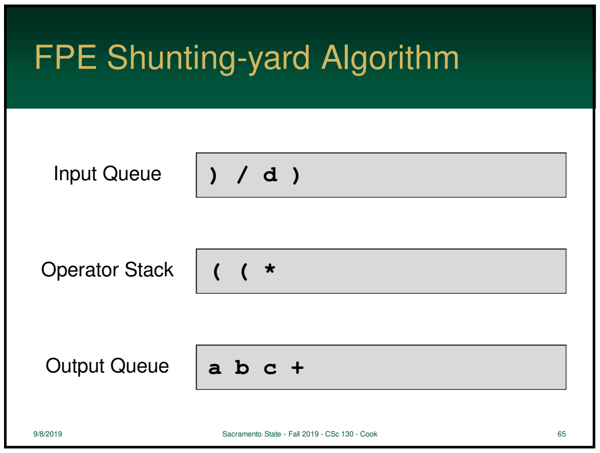
FPE Shunting-yard Algorithm

Input Queue	`c ) ) / d )`
Operator Stack	`( ( * ( +`
Output Queue	`a b`



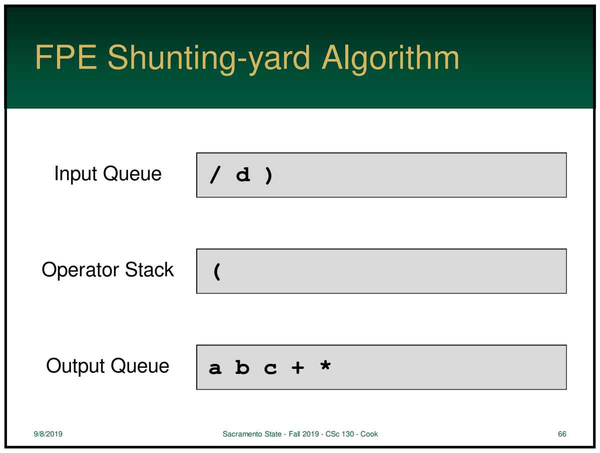
FPE Shunting-yard Algorithm

Input Queue	`) ) / d )`
Operator Stack	`( ( * ( +`
Output Queue	`a b c`



FPE Shunting-yard Algorithm

Input Queue	`) / d )`
Operator Stack	`( ( *`
Output Queue	`a b c +`



FPE Shunting-yard Algorithm

Input Queue	`/ d )`
Operator Stack	`(`
Output Queue	`a b c + *`

FPE Shunting-yard Algorithm

Input Queue	`d )`
Operator Stack	`( /`
Output Queue	`a b c + *`

FPE Shunting-yard Algorithm

Input Queue	`)`
Operator Stack	`( /`
Output Queue	`a b c + * d`

FPE Shunting-yard Algorithm

Input Queue	
Operator Stack	
Output Queue	`a b c + * d /`

Too Many Paranthesis!

- FPE's are rarely used in real-World examples
- In fact, we use precedence rules to simplify expressions
- Fortunately, the algorithm can be modified, *very easily*, to handle precedence!

FPE Shunting-yard Algorithm

```
while the input queue has tokens
   read a token from the input queue
   if the token is a…
      operand : add it to output queue
      operator : new rules – see next slide
      '(' : push it onto the stack
      ')' :
         while the top of stack isn't a '('
            pop an operator
            add it to the output queue
         end while
         pop and discard the '('
   end if
end while
```

Operator: New Rules

- When you read an operator from the input queue….
- … go into a loop that looks at the top of the stack and compares its precedence to the current operator
- If the current operator is …
  - left-associative, pop while the top is **>=**
  - right-associative, pop while the top is **>**

Shunting-yard Algorithm Operators

- Stop if you hit a '('
- Each pop'd operator is put directly on the output queue
- Finally, push the current operator onto the stack

Operator Associatively

Operator	Associatively
`+ – * /`	Left
`^` (exponent)	Right

Shunting-yard Algorithm Example 1

Input Queue	`a – b * c + d`
Operator Stack	
Output Queue	

Shunting-yard Algorithm Example 1

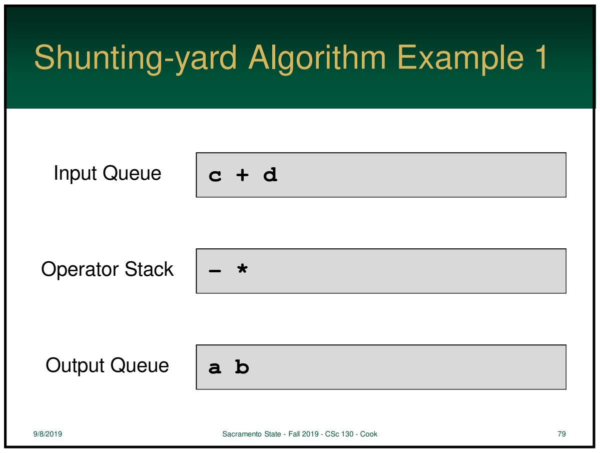
Input Queue	`– b * c + d`
Operator Stack	
Output Queue	`a`

Shunting-yard Algorithm Example 1

Input Queue	`b * c + d`
Operator Stack	`–`
Output Queue	`a`

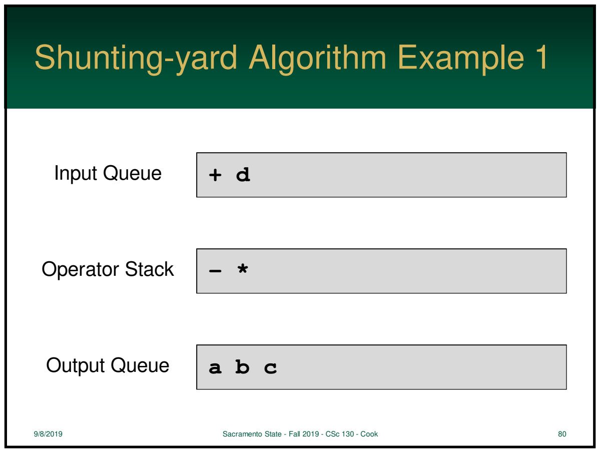
Shunting-yard Algorithm Example 1

Input Queue	`* c + d`
Operator Stack	`–`
Output Queue	`a b`



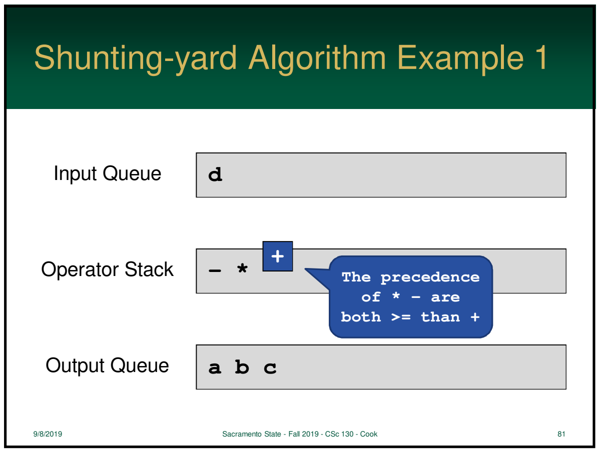
Shunting-yard Algorithm Example 1

Input Queue	c + d
Operator Stack	- *
Output Queue	a b



Shunting-yard Algorithm Example 1

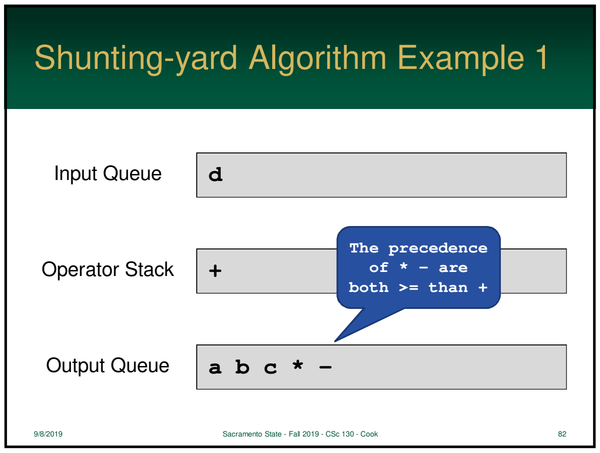
Input Queue	+ d
Operator Stack	- *
Output Queue	a b c



Shunting-yard Algorithm Example 1

Input Queue	d
Operator Stack	- * +
Output Queue	a b c

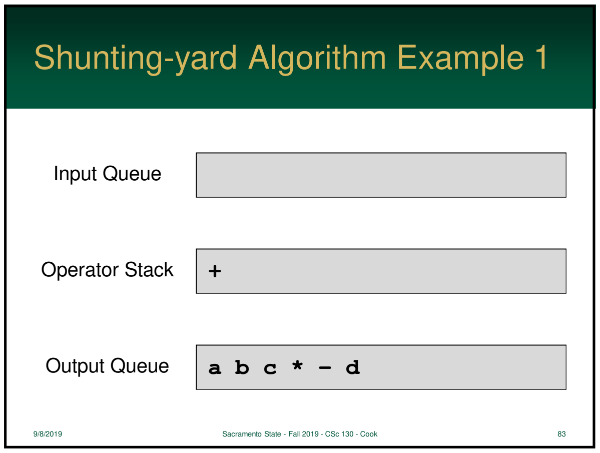
The precedence of * - are both >= than +



Shunting-yard Algorithm Example 1

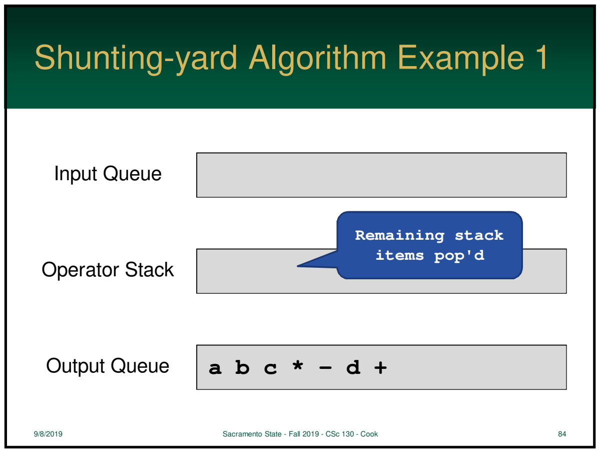
Input Queue	d
Operator Stack	+
Output Queue	a b c * -

The precedence of * - are both >= than +



Shunting-yard Algorithm Example 1

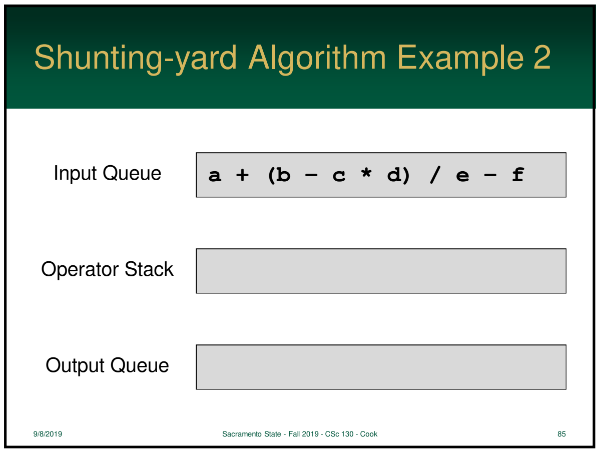
Input Queue	
Operator Stack	+
Output Queue	a b c * - d



Shunting-yard Algorithm Example 1

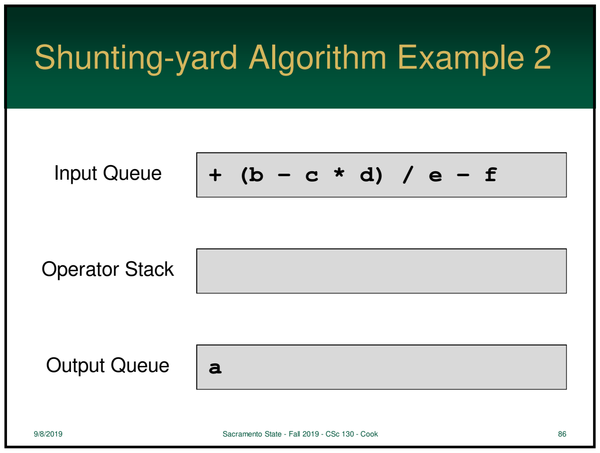
Input Queue	
Operator Stack	
Output Queue	a b c * - d +

Remaining stack items pop'd



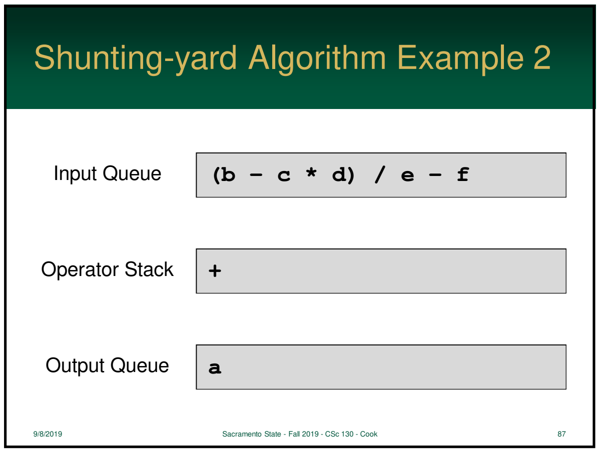
Shunting-yard Algorithm Example 2

Input Queue	`a + (b – c * d) / e – f`
Operator Stack	
Output Queue	



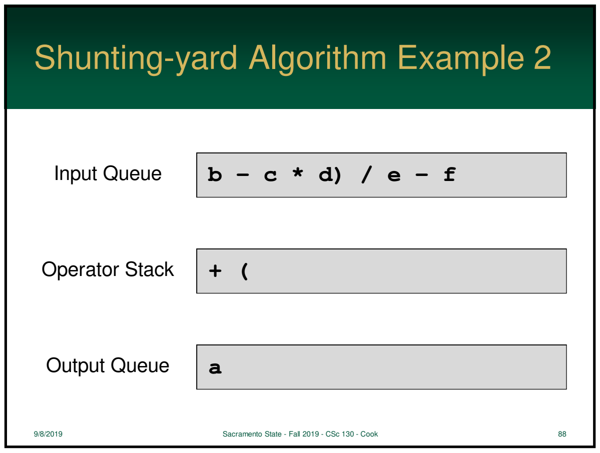
Shunting-yard Algorithm Example 2

Input Queue	`+ (b – c * d) / e – f`
Operator Stack	
Output Queue	`a`



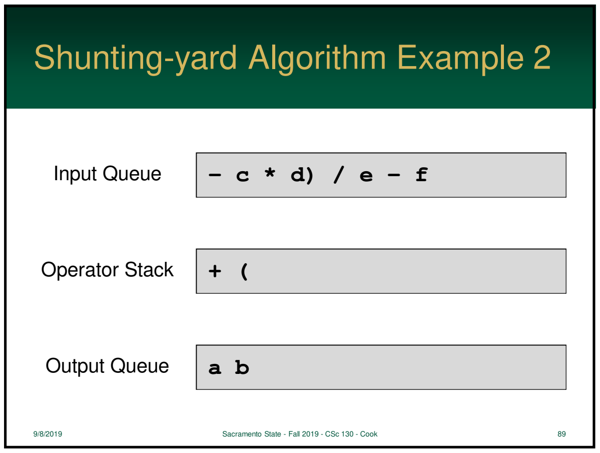
Shunting-yard Algorithm Example 2

Input Queue	`(b – c * d) / e – f`
Operator Stack	`+`
Output Queue	`a`



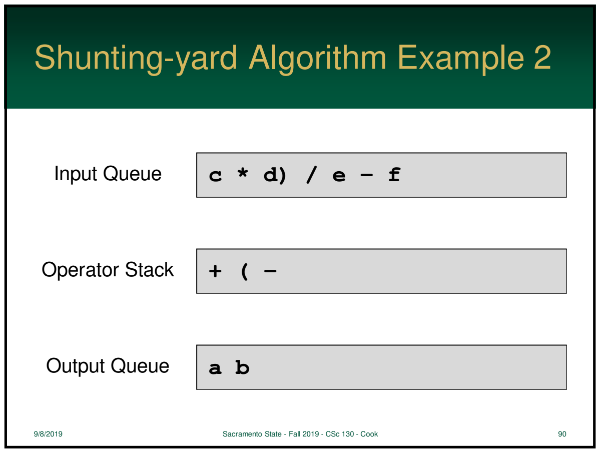
Shunting-yard Algorithm Example 2

Input Queue	`b – c * d) / e – f`
Operator Stack	`+ (`
Output Queue	`a`



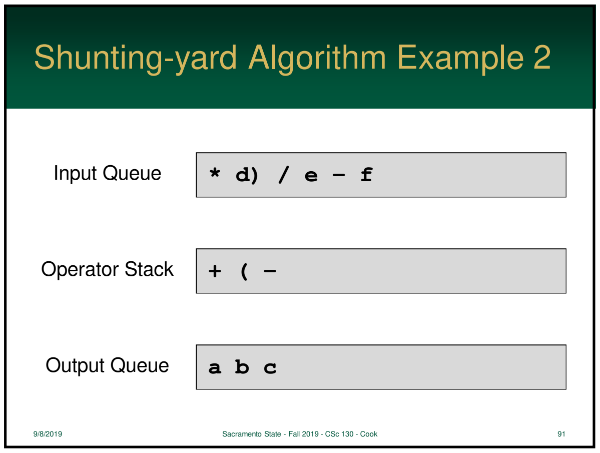
Shunting-yard Algorithm Example 2

Input Queue	`– c * d) / e – f`
Operator Stack	`+ (`
Output Queue	`a b`



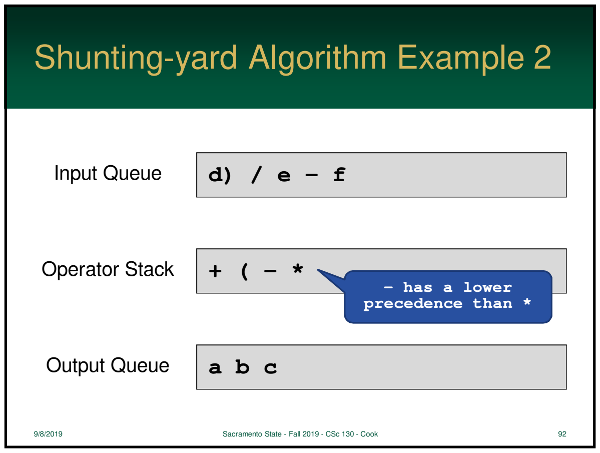
Shunting-yard Algorithm Example 2

Input Queue	`c * d) / e – f`
Operator Stack	`+ ( –`
Output Queue	`a b`



Shunting-yard Algorithm Example 2

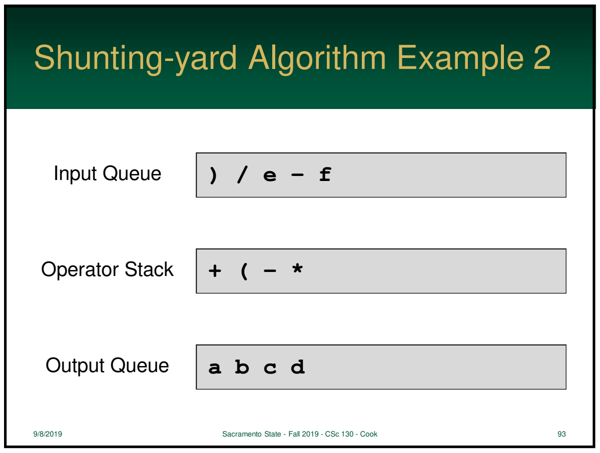
Input Queue	`* d) / e – f`
Operator Stack	`+ ( –`
Output Queue	`a b c`



Shunting-yard Algorithm Example 2

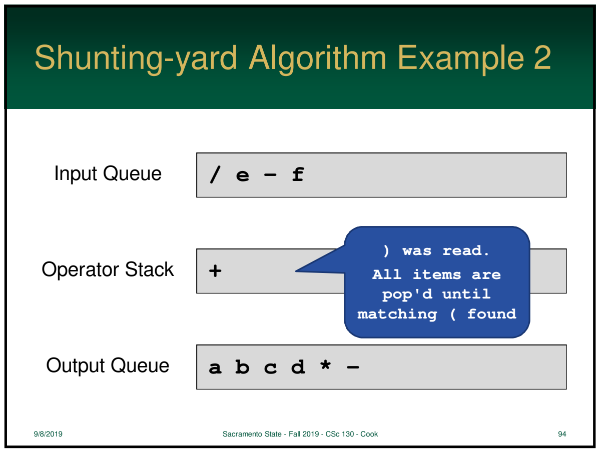
Input Queue	`d) / e – f`
Operator Stack	`+ ( – *`
Output Queue	`a b c`

– has a lower precedence than *



Shunting-yard Algorithm Example 2

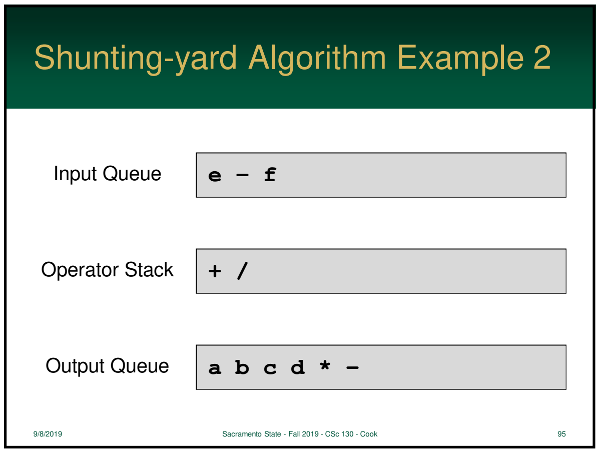
Input Queue	`) / e – f`
Operator Stack	`+ ( – *`
Output Queue	`a b c d`



Shunting-yard Algorithm Example 2

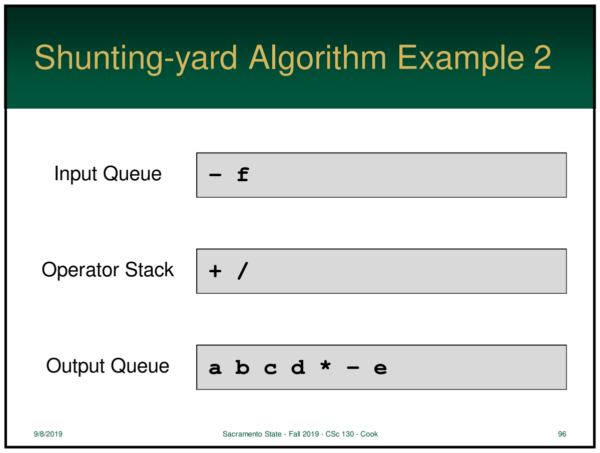
Input Queue	`/ e – f`
Operator Stack	`+`
Output Queue	`a b c d * –`

) was read.
All items are pop'd until matching (found



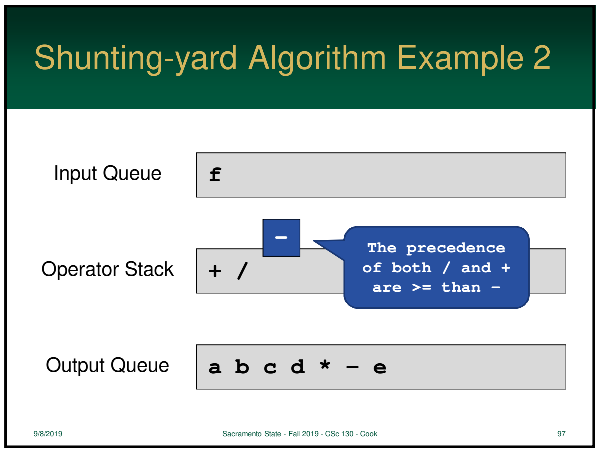
Shunting-yard Algorithm Example 2

Input Queue	`e – f`
Operator Stack	`+ /`
Output Queue	`a b c d * –`



Shunting-yard Algorithm Example 2

Input Queue	`– f`
Operator Stack	`+ /`
Output Queue	`a b c d * – e`

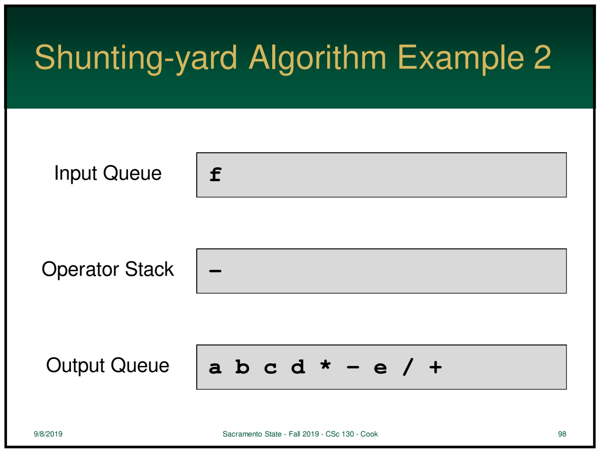


Shunting-yard Algorithm Example 2

Input Queue	`f`
Operator Stack	`+ /`
Output Queue	`a b c d * – e`

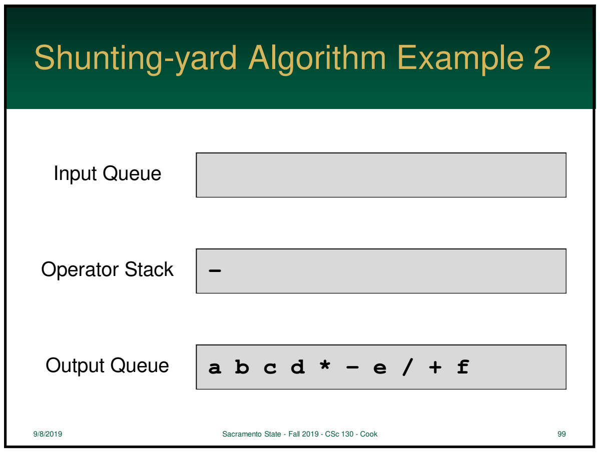
`–`

The precedence of both / and + are >= than –



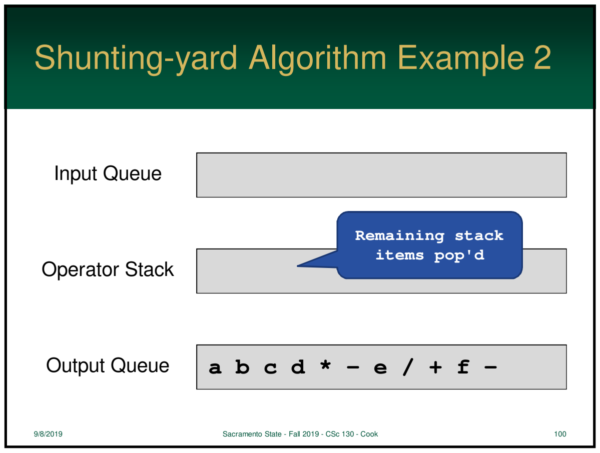
Shunting-yard Algorithm Example 2

Input Queue	`f`
Operator Stack	`–`
Output Queue	`a b c d * – e / +`



Shunting-yard Algorithm Example 2

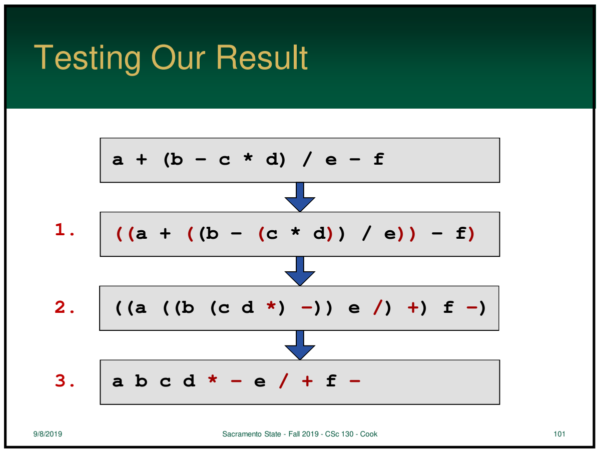
Input Queue	
Operator Stack	`–`
Output Queue	`a b c d * – e / + f`



Shunting-yard Algorithm Example 2

Input Queue	
Operator Stack	
Output Queue	`a b c d * – e / + f –`

Remaining stack items pop'd



Testing Our Result

```
a + (b – c * d) / e – f
```

1. `((a + ((b – (c * d)) / e)) – f)`
2. `((a ((b (c d *) –)) e /) +) f –)`
3. `a b c d * – e / + f –`