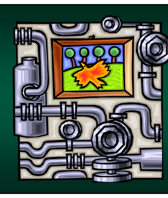
Stacks & Queues

Part 3

1



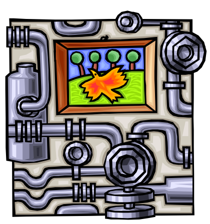
Data Abstraction

Abstraction is power

2

Abstract Data Types

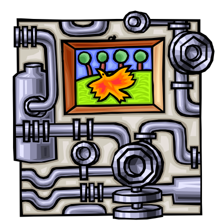
- *Data types* are used in practically all programming languages
- The core data types found in language is known as a *primitive data type*



3

Data Types Specify 2 Things

1. Set of possible values
2. *Operations* on the data
   - these are alternatively called *functions* or *methods*
   - data types often define the errors can occur during each operation



4

Integer Example

- *int* is a type (found in most languages)
- The 32-bit version can contain values from -2^{31} to $2^{31}-1$

```
int n;
```


5

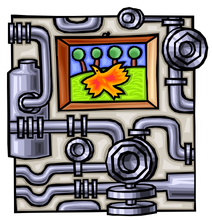
Integer Example

- Operations include: `+`, `*`, `-`, `/`, `%`, and many more (e.g. comparisons)

```
int n;
```


6

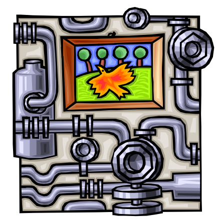
Abstract Data Types



- An *abstract data type (ADT)* hides how it is implemented from the *client* (programmer)
- The client only interacts with the defined operations

7

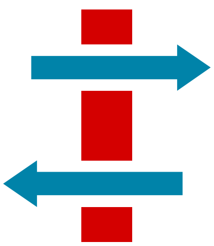
Abstract Data Types



- This layer of abstraction separates implementation from behavior
- And, it allows you to change the data structure – without breaking the ADT

8

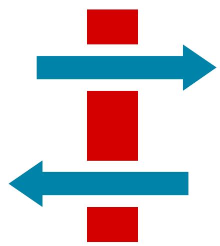
ADTs vs Data Structures



- An ADT is implementation independent
- Can, internally, use any data structure
  - array, linked list, etc…
  - depending how the ADT works, some are better than others

9

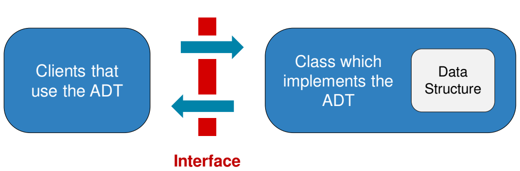
ADTs vs Data Structures



- ADT defines an *interface*
- It defines:
  - operations (public methods)
  - properties (public fields)

10

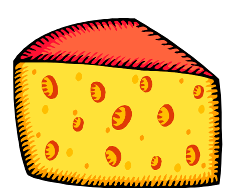
Data Structures



11

Example ADT: Cheese Trader

- Data stores orders of cheese
- The operations supported are
  - buy (cheese, count)
  - sell (cheese, count)
  - cancel (Order)
  - balance – current funds



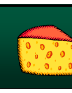
12

Example ADT: Cheese Trader

- Error conditions:
  - nonexistent cheese
  - sell a cheese we don't have
  - count is not greater than 0

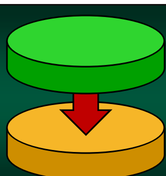
13

Cheese Trader Interface



public class CheeseTrader		
int	buy(String name, int count)	*Returns order #*
int	sell(String name, int count)	*Returns order #*
void	cancel(int order)	
double	balance()	

14



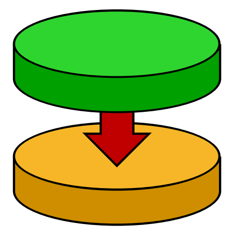
Stacks

Piles of… Data

15

Stack

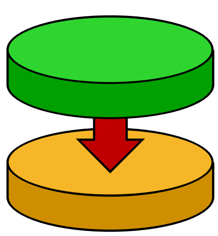
- The *Stack ADT* stores objects based on the concept of a stack of items – like a stack of dishes
- Data can only be added to or removed from the top of the stack



16

Stack

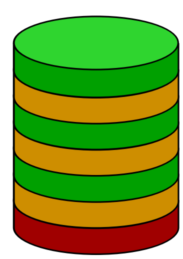
- This gives a first-in-last-out logic (aka FILO)
- Same concept is also called last-in-first-out (LIFO)



17

Stack Operation: Push

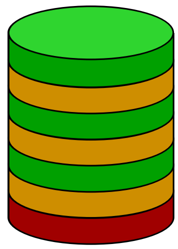
- A value is added to the stack
- It is placed on the top location
- Rest of the items are "covered"



18

Stack Operation: Pop

- Removes an item from the stack
- Last item added is removed
- 2nd item becomes the top



19

Stack Interface

public class Stack		
	Stack()	*Create empty stack*
void	push(Object item)	
Object	pop()	
Object	top()	*Return top. Sometimes called Peek()*
bool	isEmpty()	

20

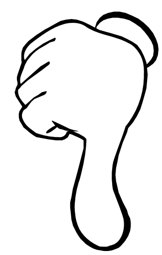
Stacks: Error Conditions

- The execution of an operation may sometimes cause an error condition, called an *exception*
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty

21

Resizing an Array-Based Stack

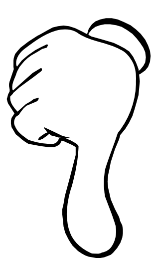
- For stacks, if a dynamically allocated array is used, each pop/push will require the entire array to be resized
- It will require O(n)
- So, a dynamic array is a poor choice



22

One Solution… Not a Great One

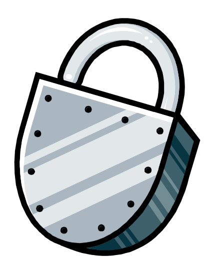
- The array *could* grow/shrink by a specific # of elements
- So, the array will resize only when a new "block" of elements is needed
- Like a fixed-capacity array, we need to keep an end index



23

Fixed-Capacity Stacks

- A fixed-capacity array can be used instead
- For a *fixed-capacity stack*, an array is an excellent choice – in specific situations…



24

Array-Based Fixed-Capacity Stack

- The stack would behave as normal until the capacity is reached
- In this case, one of two things will happen…

Spring 2024 Sacramento State - Cook - CSc 130 25

25

When the Stack is filled…

1. Stack throws an *Overflow Error*
2. Stack discards an object
   - the bottom of the stack is typically removed
   - this gives the space needed for the newly pushed object
   - e.g. the history feature of your web browser

Spring 2024 Sacramento State - Cook - CSc 130 26

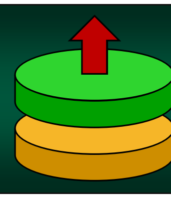
26

Stack Summary

Operation	Fixed-Capacity Array	Resizable Array	Linked List
Pop()	O(1)	**O(n)**	O(1)
Push()	O(1)	**O(n)**	O(1)
Top()	O(1)	O(1)	O(1)

Spring 2024 Sacramento State - Cook - CSc 130 27

27



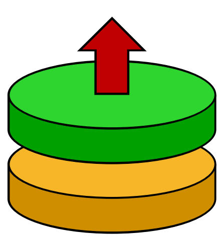
Queues

Conga-line of Data!

28

Queues

- *Queue ADT* stores list of arbitrary objects
- Based on the concept of a line – e.g. when you buy groceries
- Objects enter the back of the line, and must wait for prior items to leave before they do

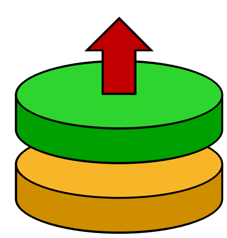


Spring 2024 Sacramento State - Cook - CSc 130 29

29

Queues

- In most parts of the World, they call a "line" a "queue"
- Main queue operations:
  - enqueue (object): place on item on the queue
  - dequeue: removes and returns the first inserted object

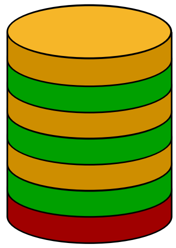


Spring 2024 Sacramento State - Cook - CSc 130 30

30

Queue Operation: Enqueue

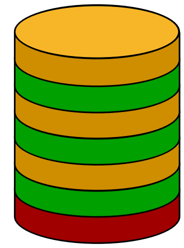
- When an object is "enqueued", it is put on to the **end** of the queue
- The items on the top of the queue are not covered



31

Queue Operation: Dequeue

- Dequeue removes the item from the front of the queue
- Second item becomes the new first item
- This gives a first-in-first-out logic (aka FIFO)



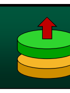
32

Auxiliary Queue Operations

- Queues also tend to have some operations defined
- These are not necessary, but they are useful
- Auxiliary operations:
  - peek: return the next object without removing it. This is also sometimes called "front"
  - size: returns the number of objects on the queue
  - isEmpty: indicates whether the queue contains no objects. This is an alterative to size()

33

Queue Interface



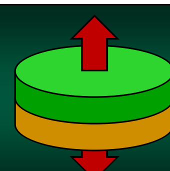
public class Queue		
	Queue()	*Create empty queue*
void	enqueue(Object item)	
Object	dequeue()	
int	size()	
Object	peek()	*Return first item, without dequeue*

34

Queue Summary

Operation	Fixed-Capacity Array	Resizable Array	Linked List
Enqueue()	O(1)	**O(n)**	O(1)
Dequeue()	O(1)	**O(n)**	O(1)
Peek()	O(1)	O(1)	O(1)

35



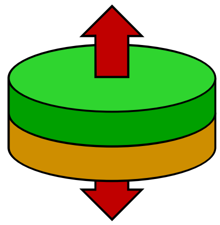
The Deque ADT

Time to shuffle the "deck"

36

Deque ADT

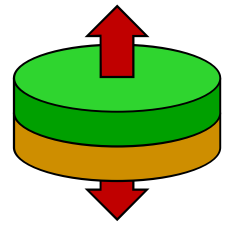
- There is a variant of the queue called a *deque* (pronounced "deck")
- The name is derived from **d**ouble-**e**nded **que**ue (sometimes it is shorted more to DQ)



37

Deque ADT

- As the name implies, it's a queue allows insertions and removals from both ends
- It is a merging of a stack and queue ADT and the operations are union of the two
- Be warned: name of each operation varies greatly between programming languages



38

Deque ADT

- addFront
  - place an object on the front of the deque
  - this is same as stack "push"
  - also called: offerFirst, pushFirst
- addBack
  - place an object on the end of the deque
  - this is the same as queue "enqueue"
  - also called: offerLast, pushLast

39

Deque ADT

- removeFront
  - remove an object from the front of the deque
  - same as: queue "dequeue" or stack "pop"
  - also called: pollFirst, popFront
- removeBack
  - this is unique – and not found in either a stack or queue ADT
  - also called pollLast, popBack

40

Deque Interface

public class Deque		
	`Deque()`	*Create empty deque*
`void`	`addFront(Object item)`	
`void`	`addBack(Object item)`	
`Object`	`removeFront()`	
`Object`	`removeBack()`	
`Object`	`peekFront()`	
`Object`	`peekBack()`	
`bool`	`isEmpty()`	

41

Deque Example

1. `addFront('N')` N
2. `addBack('E')` E
3. `addFront('W')` W
4. `addBack('D')` D
5. `addFront('P')` P

42

Deque Advantages

- A deque can function as either a stack or queue
- "Add Front" operation can be used to "redo" or "undo" a queue removal – remove then put it back in line
- There are some scenarios where this logic is needed

43

Deque Disadvantages

- While, Stacks/Queues can be created with a single-linked-list, *a Deque requires a double-linked-list*
- …otherwise, removing items from the end would require O(n) – *even with a tail node*
- Also, the link overhead (memory requirements) is doubled

44

Deque Summary

Operation	Fixed Array	Resizable Array	Single Linked List	Double Linked List
addFront()	O(1)	**O(n)**	O(1)	O(1)
addBack()	O(1)	**O(n)**	O(1)	O(1)
removeFront()	O(1)	**O(n)**	O(1)	O(1)
removeBack()	O(1)	**O(n)**	**O(n)**	O(1)

45

Queues & Stacks in Practice

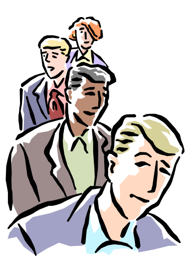
1001 Uses!

(I meant 1,001 – not 9)

46

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…



47

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*

48

HTML Tag Matching

```
<html>
<body>
<center>
<h1>Banks of Sacramento</h1>
</center>
<i>A bully ship and a bully crew.<br>
Hoo-da! Hoo-da!<br>
A bully mate and a captain too.<br>
Hoo-da! Hoo-da-day!<br>
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.
Hoo-da! Hoo-da!
A bully mate and a captain too.
Hoo-da! Hoo-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.

49

Balanced Parentheses

- When analyzing arithmetic expressions often the structure of the expression needs to be checked
- For example:
  - are operators in the correct place?
  - are the parenthesis balanced?

50

Balanced Parentheses

- Let's look at parenthesis
- One approach…
  - can we just use a "parenthesis count"
  - if it isn't 0 at the end then the expression is invalid
- Sorry, it won't work…
  - some expressions allow { } and [ ]
  - …and they may be in the wrong place

51

Balanced Parentheses

- A great solution is a stack
- Approach…
  - 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
- It can also catch mismatched symbols

52

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

53

Balanced Parenthesis Examples

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

54

Evaluating Expressions

A Stack and Queue working together!

55

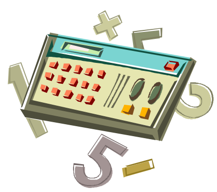
Evaluating Expressions

- 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

56

Evaluating Expressions

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



57

Infix Notation

- Using *infix notation*, we put the operator in between the two operands
- 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`

58

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`

59

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 /`

60

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 – / +`

61

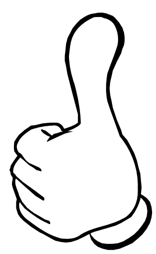
Where are My Parenthesis?

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

62

Compute Postfix Algorithm

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



63

Compute Postfix Pseudo-code

```
while there is data in the input queue
   dequeue a token (value or operator)
   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
```


64

Compute Postfix Demo

Input Queue: 24 10 7 – / 34 +

`24 / (10 – 7) + 34`

Stack:

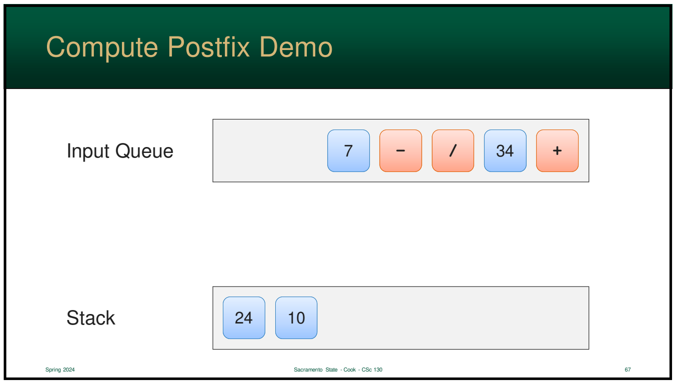
65

Compute Postfix Demo

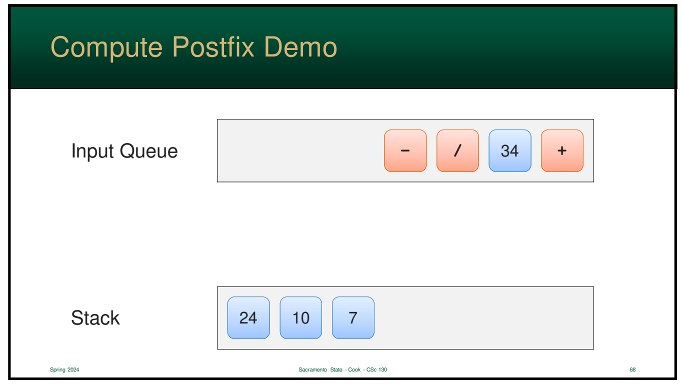
Input Queue: 10 7 – / 34 +

Stack: 24

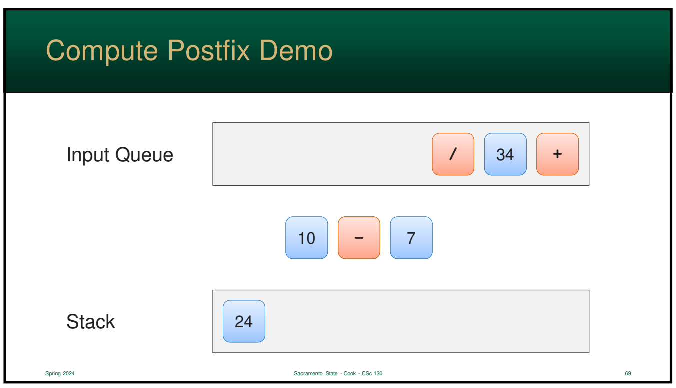
66



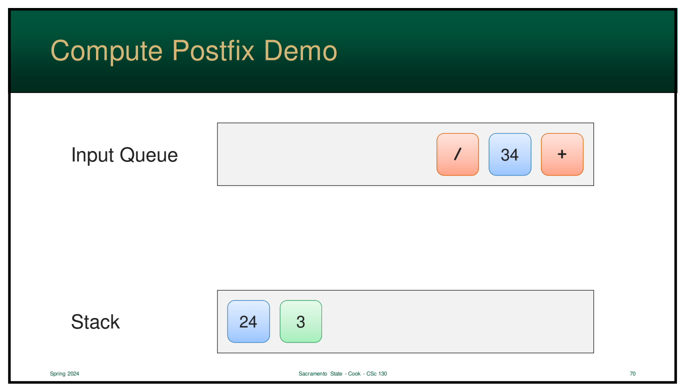
67



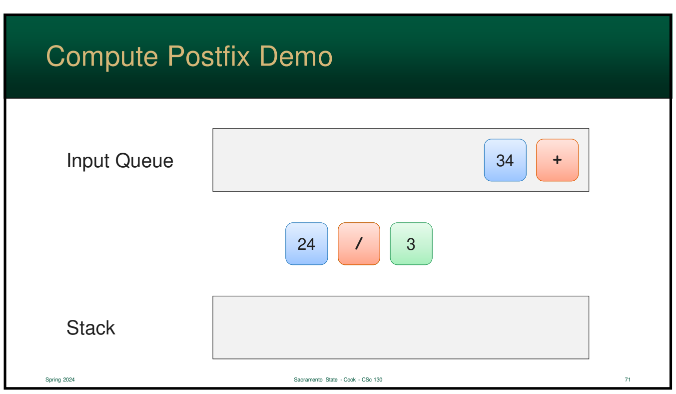
68



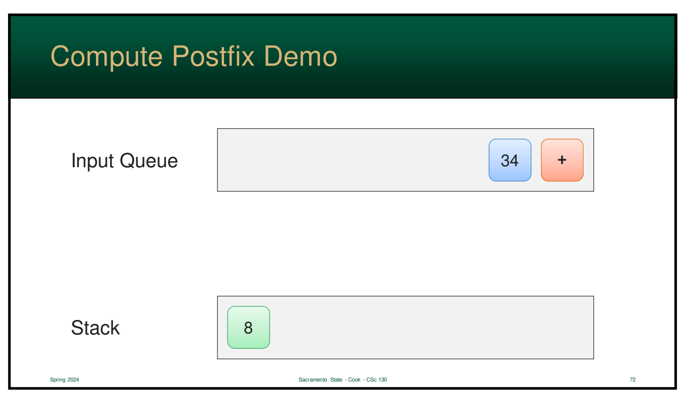
69



70



71



72

Compute Postfix Demo

Input Queue: +

Stack: 8 34

73

Compute Postfix Demo

Input Queue

8 + 34

Stack

74

Compute Postfix Demo

Input Queue

Stack: 42

75

Converting to Prefix or Postfix

- Why are learning this... *be patient!*
- Converting infix to either 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

76

Convert Infix to Prefix / Postfix

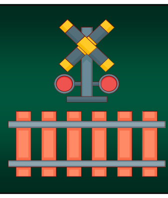
1. Make it a *Fully Parenthesized Expression (FPE)* - one pair of parentheses enclosing each operator and its operands
2. Move the operators to the start (prefix) or end (postfix) of each sub-expression
3. Finally, remove all the parenthesis

77

Infix to Postfix

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


78



Infix to Postfix Algorithm

Let the computer do the work…

79

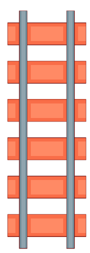
Edsger Dijkstra

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

80

Infix to Postfix Algorithm

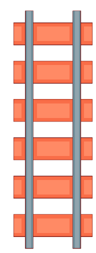
- Infix expressions need to be converted to postfix to be evaluated
- Dijkstra's *Shunting-yard algorithm* performs this task



81

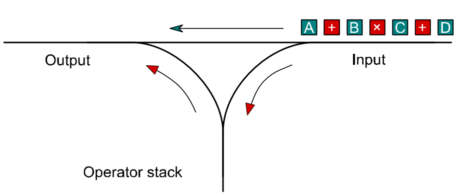
Shunting-yard algorithm

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



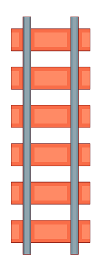
82

Shunting-yard Algorithm



83

Shunting-yard Algorithm



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

84

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
```

Spring 2024 Sacramento State - Cook - CSc 130 85

85

FPE Shunting-yard Algorithm

Input Queue: ((a * (b + c)) / d)

Operator Stack:

Output Queue:

Spring 2024 Sacramento State - Cook - CSc 130 86

86

FPE Shunting-yard Algorithm

Input Queue: ((a * (b + c)) / d)

Operator Stack:

Output Queue:

Spring 2024 Sacramento State - Cook - CSc 130 87

87

FPE Shunting-yard Algorithm

Input Queue: (a * (b + c)) / d)

Operator Stack: (

Output Queue:

Spring 2024 Sacramento State - Cook - CSc 130 88

88

FPE Shunting-yard Algorithm

Input Queue: a * (b + c)) / d)

Operator Stack: ((

Output Queue:

Spring 2024 Sacramento State - Cook - CSc 130 89

89

FPE Shunting-yard Algorithm

Input Queue: * (b + c)) / d)

Operator Stack: ((

Output Queue: a

Spring 2024 Sacramento State - Cook - CSc 130 90

90

FPE Shunting-yard Algorithm

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

Spring 2024 Sacramento State - Cook - CSc 130 91

91

FPE Shunting-yard Algorithm

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

Spring 2024 Sacramento State - Cook - CSc 130 92

92

FPE Shunting-yard Algorithm

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

Spring 2024 Sacramento State - Cook - CSc 130 93

93

FPE Shunting-yard Algorithm

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

Spring 2024 Sacramento State - Cook - CSc 130 94

94

FPE Shunting-yard Algorithm

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

Spring 2024 Sacramento State - Cook - CSc 130 95

95

FPE Shunting-yard Algorithm

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

Spring 2024 Sacramento State - Cook - CSc 130 96

96

FPE Shunting-yard Algorithm

Input Queue: `) / d )`

Operator Stack: `( ( *`

Output Queue: `a b c +`

97

FPE Shunting-yard Algorithm

Input Queue: `/ d )`

Operator Stack: `( ( * )`

Output Queue: `a b c +`

98

FPE Shunting-yard Algorithm

Input Queue: `/ d )`

Operator Stack: `(`

Output Queue: `a b c + *`

99

FPE Shunting-yard Algorithm

Input Queue: `d )`

Operator Stack: `( /`

Output Queue: `a b c + *`

100

FPE Shunting-yard Algorithm

Input Queue: `)`

Operator Stack: `( /`

Output Queue: `a b c + * d`

101

FPE Shunting-yard Algorithm

Input Queue:

Operator Stack: `( / )`

Output Queue: `a b c + * d`

102

FPE Shunting-yard Algorithm

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

103

FPE Shunting-yard Algorithm

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

104

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!

105

Non-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
```


106

Operator: New Rules

```
if operator is left-associative
    while top of stack is ≥ operator and not a '('
       pop the stack
       add it to the output queue
    end while
if operator is right-associative
    while top of stack is > operator and not a '('
       pop the stack
       add it to the output queue
    end while
push the operator onto the stack
```


107

Operator Associatively

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

108

Shunting-yard Algorithm Example 1

Input Queue: a − b * c + d

Operator Stack:

Output Queue:

109

Shunting-yard Algorithm Example 1

Input Queue: a − b * c + d

Operator Stack:

Output Queue:

110

Shunting-yard Algorithm Example 1

Input Queue: − b * c + d

Operator Stack:

Output Queue: a

111

Shunting-yard Algorithm Example 1

Input Queue: b * c + d

Operator Stack: −

Output Queue: a

112

Shunting-yard Algorithm Example 1

Input Queue: * c + d

Operator Stack: −

Output Queue: a b

113

Shunting-yard Algorithm Example 1

Input Queue: c + d

Operator Stack: − *

Output Queue: a b

114

Shunting-yard Algorithm Example 1

Input Queue: + d

Operator Stack: – *

Output Queue: a b c

115

Shunting-yard Algorithm Example 1

Input Queue: d

Operator Stack: – * +

The precedence of * – are both ≥ than +

Output Queue: a b c

116

Shunting-yard Algorithm Example 1

Input Queue: d

Operator Stack: +

Output Queue: a b c * –

117

Shunting-yard Algorithm Example 1

Input Queue:

Operator Stack: +

Remaining stack items pop'd

Output Queue: a b c * – d

118

Shunting-yard Algorithm Example 1

Input Queue:

Operator Stack:

Output Queue: a b c * – d +

119

Shunting-yard Algorithm Example 1

Input Queue:

Operator Stack:

Output Queue: a b c * – d +

120

Shunting-yard Algorithm Example 2

Input Queue: a + (b − c * d) / e − f

Operator Stack:

Output Queue:

121

Shunting-yard Algorithm Example 2

Input Queue: a + (b − c * d) / e − f

Operator Stack:

Output Queue:

122

Shunting-yard Algorithm Example 2

Input Queue: + (b − c * d) / e − f

Operator Stack:

Output Queue: a

123

Shunting-yard Algorithm Example 2

Input Queue: (b − c * d) / e − f

Operator Stack: +

Output Queue: a

124

Shunting-yard Algorithm Example 2

Input Queue: b − c * d) / e − f

Operator Stack: + (

Output Queue: a

125

Shunting-yard Algorithm Example 2

Input Queue: − c * d) / e − f

Operator Stack: + (

Output Queue: a b

126

Shunting-yard Algorithm Example 2

Input Queue: c * d) / e – f

Operator Stack: + (–

Output Queue: a b

127

Shunting-yard Algorithm Example 2

Input Queue: * d) / e – f

Operator Stack: + (–

Output Queue: a b c

128

Shunting-yard Algorithm Example 2

Input Queue: d) / e – f

Operator Stack: + (– *

Output Queue: a b c

129

Shunting-yard Algorithm Example 2

Input Queue:) / e – f

Operator Stack: + (– *

Output Queue: a b c d

130

Shunting-yard Algorithm Example 2

Input Queue: / e – f

Operator Stack: + (– *)

Output Queue: a b c d

131

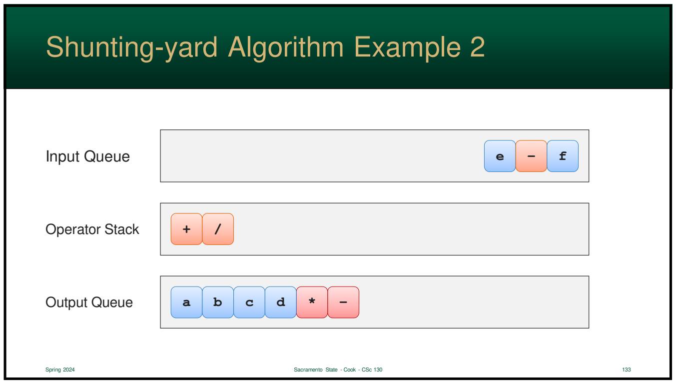
Shunting-yard Algorithm Example 2

Input Queue: / e – f

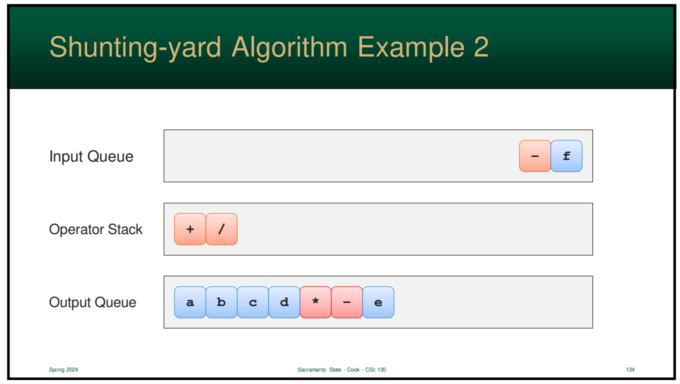
Operator Stack: +

Output Queue: a b c d * –

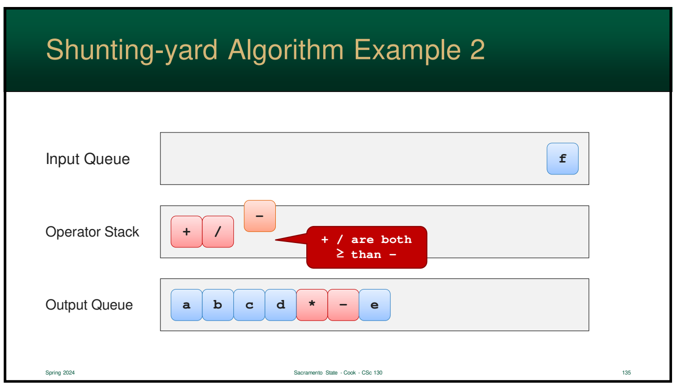
132



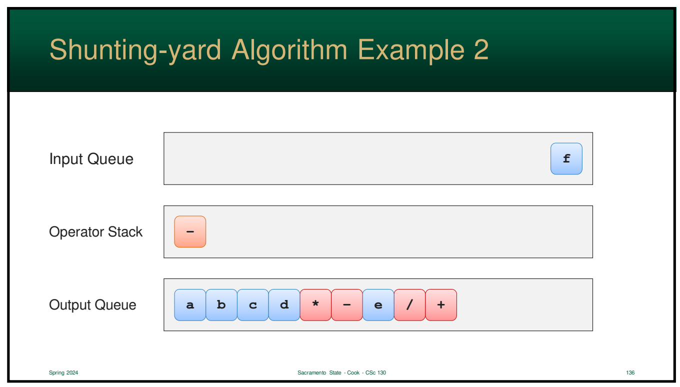
133



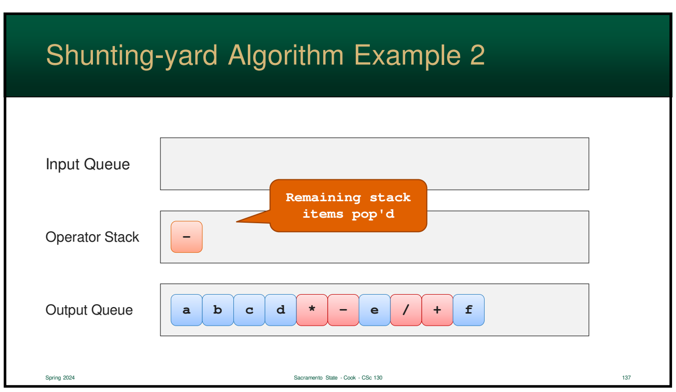
134



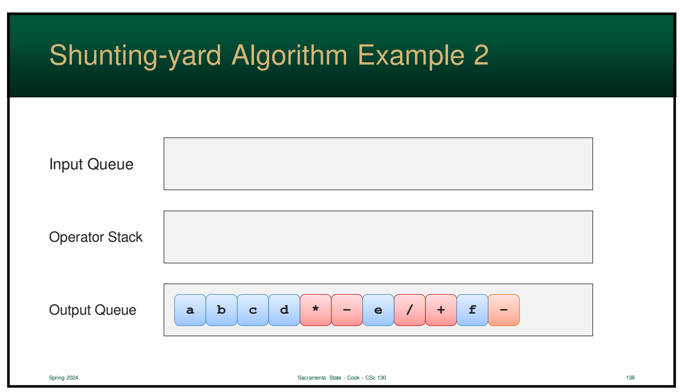
135



136



137



138

Shunting-yard Algorithm Example 2

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

Spring 2024 Sacramento State - Cook - CSc 130 139

139

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 −

Spring 2024 Sacramento State - Cook - CSc 130 140

140