STACKS

The Stack ADT

- A stack is a essentially a list in which you can only add and remove from the front only.
  - You cannot directly access elements in the middle or at the end- only at the front.
- Stacks are a first-in, last-out (FILO) data structure.
  - This also known as a last-in, first-out (LIFO) data structure.

Stack Methods

- push
  - adds to the stack
  - similar to insertHead(obj) or add(0, obj)
- pop
  - removes from the stack
  - similar to removeHead() or remove(0)
- peek
  - looks at the top of the stack but does not change the stack
  - similar to getEntry(1) or get(0)

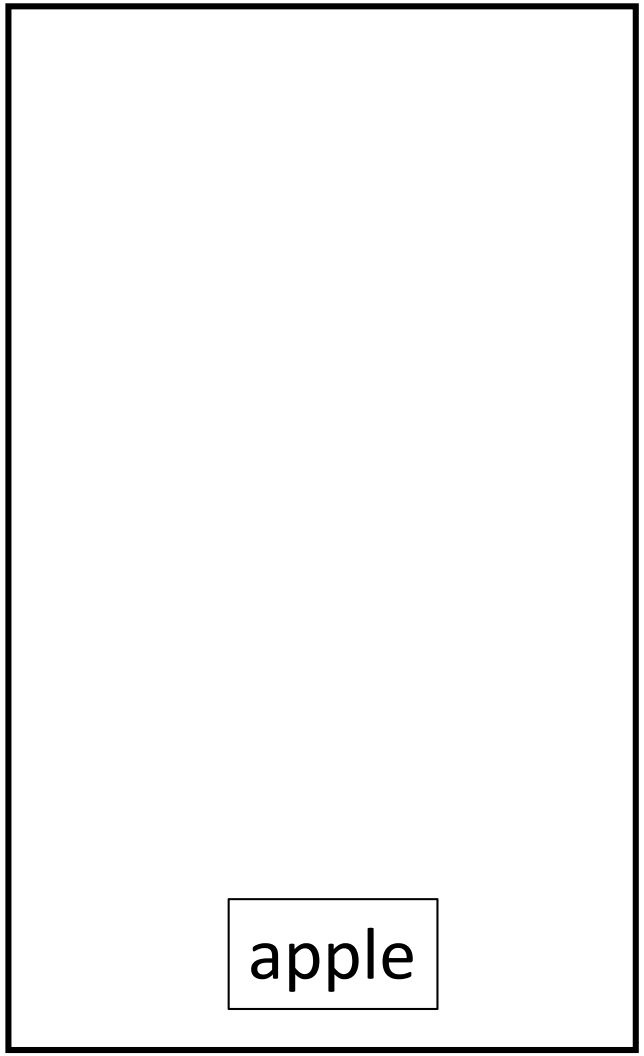
Example

```
wordStack.push("apple");
wordStack.push("banana");
wordStack.push("cantelope");
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```

wordStack

Example

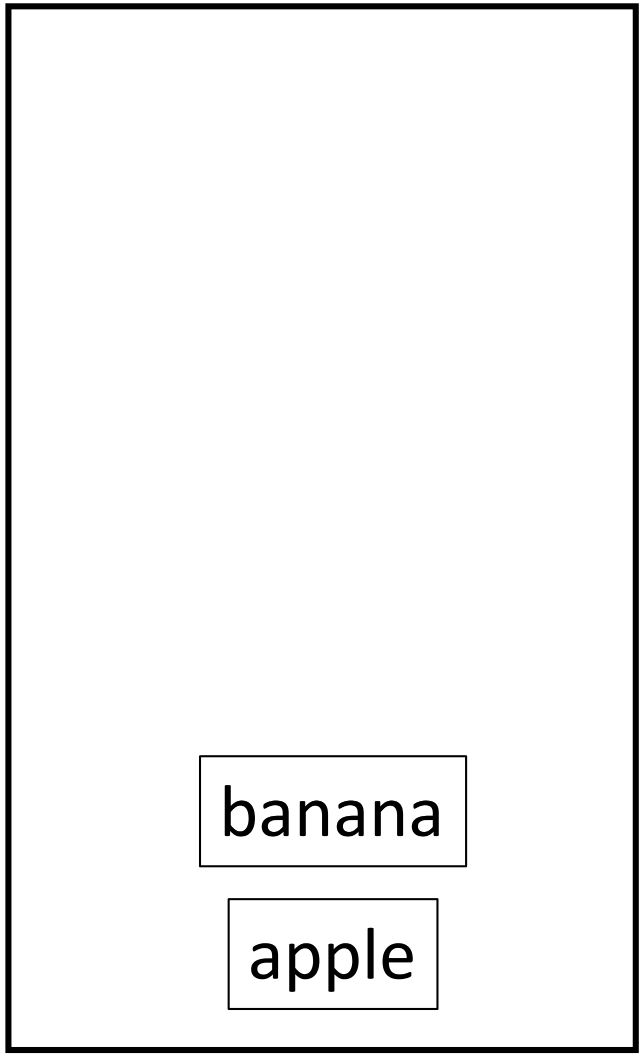
```
wordStack.push(“apple”);
wordStack.push(“banana”);
wordStack.push(“cantaloupe”);
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```



wordStack

Example

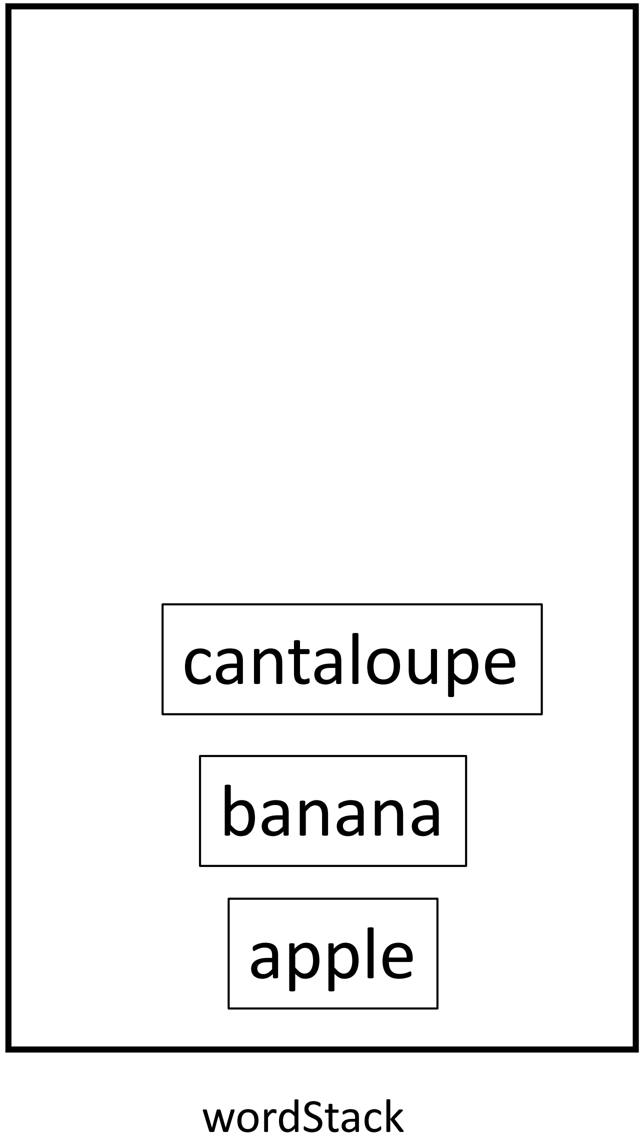
```
wordStack.push(“apple”);
wordStack.push(“banana”);
wordStack.push(“cantaloupe”);
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```



wordStack

Example

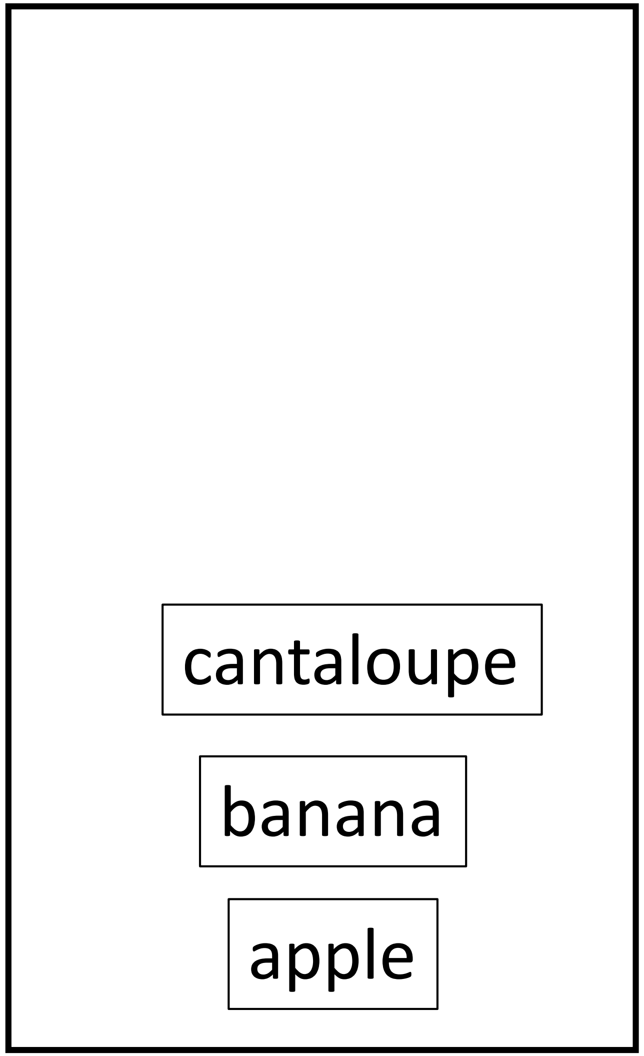
```
wordStack.push("apple");
wordStack.push("banana");
wordStack.push("cantaloupe");
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```



Example

```
wordStack.push(“apple”);
wordStack.push(“banana”);
wordStack.push(“cantaloupe”);
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```

prints: cantaloupe

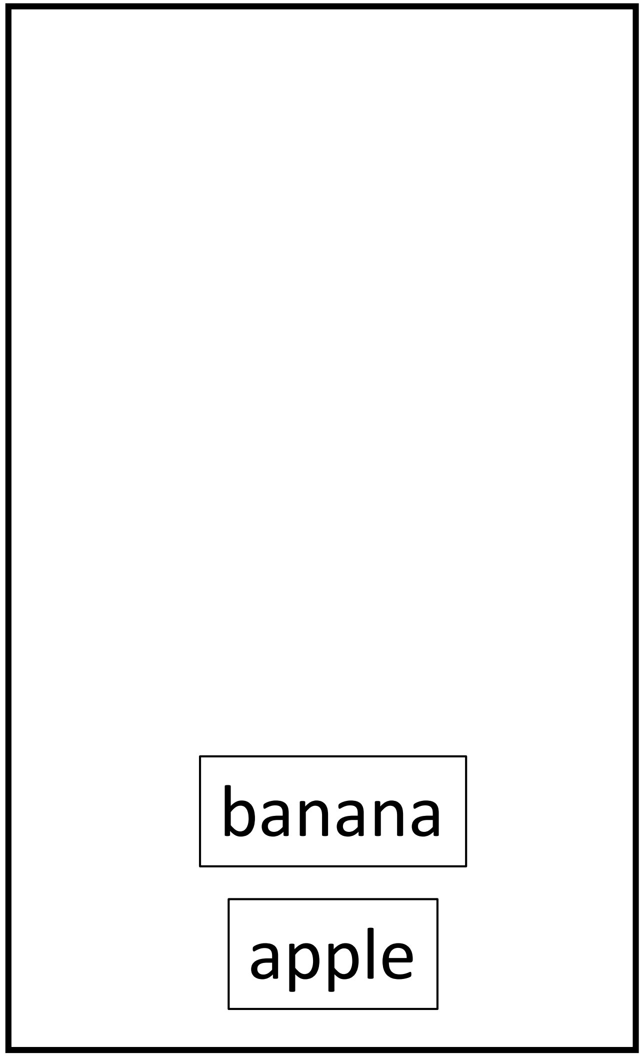


wordStack

Example

```
wordStack.push(“apple”);
wordStack.push(“banana”);
wordStack.push(“cantaloupe”);
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```

prints: cantaloupe

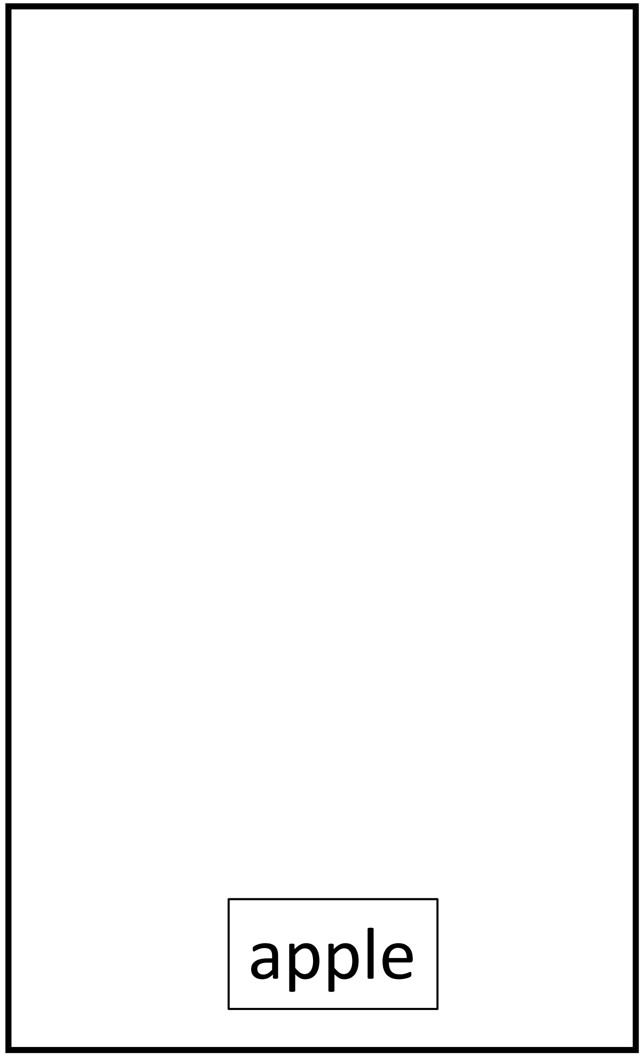


wordStack

Example

```
wordStack.push("apple");
wordStack.push("banana");
wordStack.push("cantaloupe");
System.out.println(wordStack.peek());
System.out.println(wordStack.pop());
System.out.println(wordStack.pop());
```

prints: banana



wordStack

Stacks

- Insertion order is chronological.
- Removal order is reverse chronological.

Common Uses of Stacks

- The Java runtime stack (keeps track of method calls)
- Checking balanced parentheses in infix expressions
- Converting infix expressions to postfix expressions
- Evaluating postfix expressions
- Evaluating infix expressions (by converting to postfix and then evaluating)
- Replacing recursion
- Reversing data

The Runtime Stack

- Stacks keep track of program execution.
- A *program counter* references the current instruction.
- An *activation record* (also called a *frame*) object is created and pushed when a method is called.
  - This record contains the actual parameters (arguments), local variables, and the current instruction (a copy of the program counter).
- When the method completes, the activation record is popped.
  - The new top's program counter is used to resume the next instruction.

- Recursion is a special case of this process where it's the same method that is being used to create activation records that are pushed onto the stack.

Balancing Parentheses

- Compilers use stacks frequently.
- One use is to determine whether parentheses or brackets are properly matched up.
- To check this, we can ignore all other values in the expression and just look at the parentheses.
- There are four cases:
  - balanced — example: { [( )] }
  - unbalanced- extra open parenthesis — example: { ( )
  - unbalanced- extra closed parenthesis — example: ( ) }
  - unbalanced- mismatched parentheses — example: { (})

Algorithm to Check Balanced Parentheses

- The basic idea is to gather up open parentheses on the stack.
- When you find a close parenthesis, pop an element off the stack.
  - The current close parenthesis and the popped open parenthesis should match.
  - This is because the popped parenthesis is the most recent one we saw!
- When we are done, the stack should be empty.
  - All parentheses are matched.

```
while there are more tokens, read in a token
    if the token is an open parenthesis
        push the token onto the stack
    else (the token is a closed parenthesis)
        if the stack is empty
            the expression is unbalanced (we're done- return false because of extra closed parentheses)
        else (the stack is not empty)
            pop a token
            if the closed and open parenthesis don't match
                the expression is unbalanced (we're done- return false because of mismatched parentheses)
// there are no more tokens left- the while loop is done
if the stack is empty
    the expression is balanced (we're done- return true)
else (the stack has tokens remaining in it)
    the expression is unbalanced (we're done- return false because of extra open parentheses)
```

Stack Example

- Review the example trace and code for balanced parentheses.

Infix and Postfix Notations

- binary operators take two operands
  - a+b or b*a
- There are three formats for expressions that use binary operators:
  - infix: the operator is between the operands
    - this is what humans use!
    - a+b or b*a
  - postfix: the operator follows the operands
    - ab+
    - this is what computers prefer!
  - prefix: the operator precedes the operands
    - +ab

Postfix Notation

- Computers prefer postfix notation.
- Postfix is the easiest to evaluate because you can essentially ignore precendence rules and just evaluate operators in the order you find them.
- Behind the scenes, infix expressions written by humans are converted to postfix expressions that are evaluated.

Algorithm- Converting from Infix to Postfix

- Use two stacks: PostFix and OpStack
- To simplify, we'll limit to only four operations:
  - addition, subtraction, multiplication, and division
- We'll use this hierarchy:
  1. multiplication and division (**same precedence as each other)**
  2. addition and subtraction (**same precedence as each other)**

- token: any symbol in an expression
  - could be a an operator, a operand or value, or a parenthesis

while there are tokens to process

case 1: the token is a value

push the value onto the PostFix stack

case 2: the token is an operator

while the precedence of the operator currently on the OpStack >= the precedence of the current operator

Important Note 1: This is >= (not just >)- so if the operator on the top of OpStack has the same precedence as the current one, you should enter this loop!

Important Note 2: Parentheses are not considered in this part of evaluating the contents of OpStack. In this loop, you are just comparing to other operators. If you reach a parenthesis, stop.

topOp = pop the operator from the OpStack

push topOp onto the PostFix stack

push the current operator onto OpStack

case 3: the token is a left parenthesis

push the parenthesis onto the OpStack

case 4: the token is a right parenthesis

while the top element of the OpStack is not a left parenthesis

topOp = pop the operator from the OpStack

push topOp onto the PostFix stack

pop the left parenthesis off the OpStack

// there are no more tokens to process- the while loop is done

while there are still tokens on the OpStack

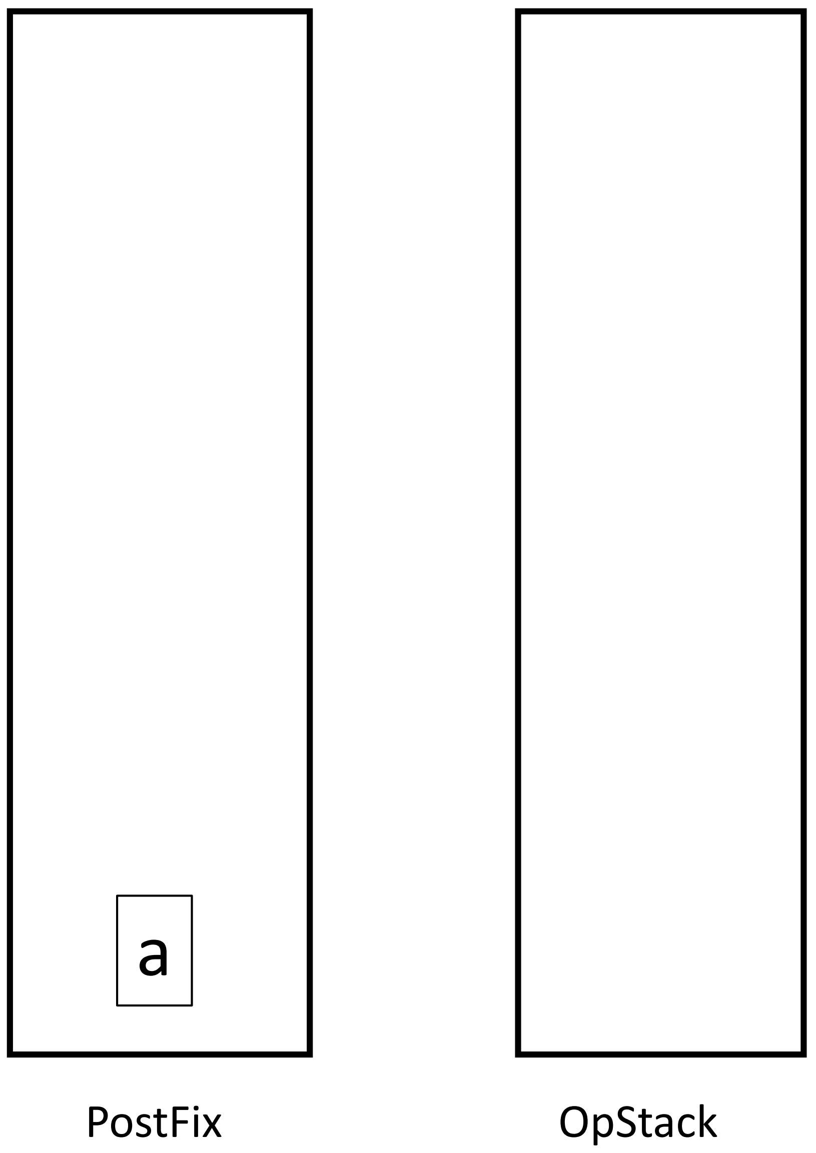
topOp = pop the operator from the OpStack

push topOp onto the PostFix stack

Example

- infix notation: a + b * c / d

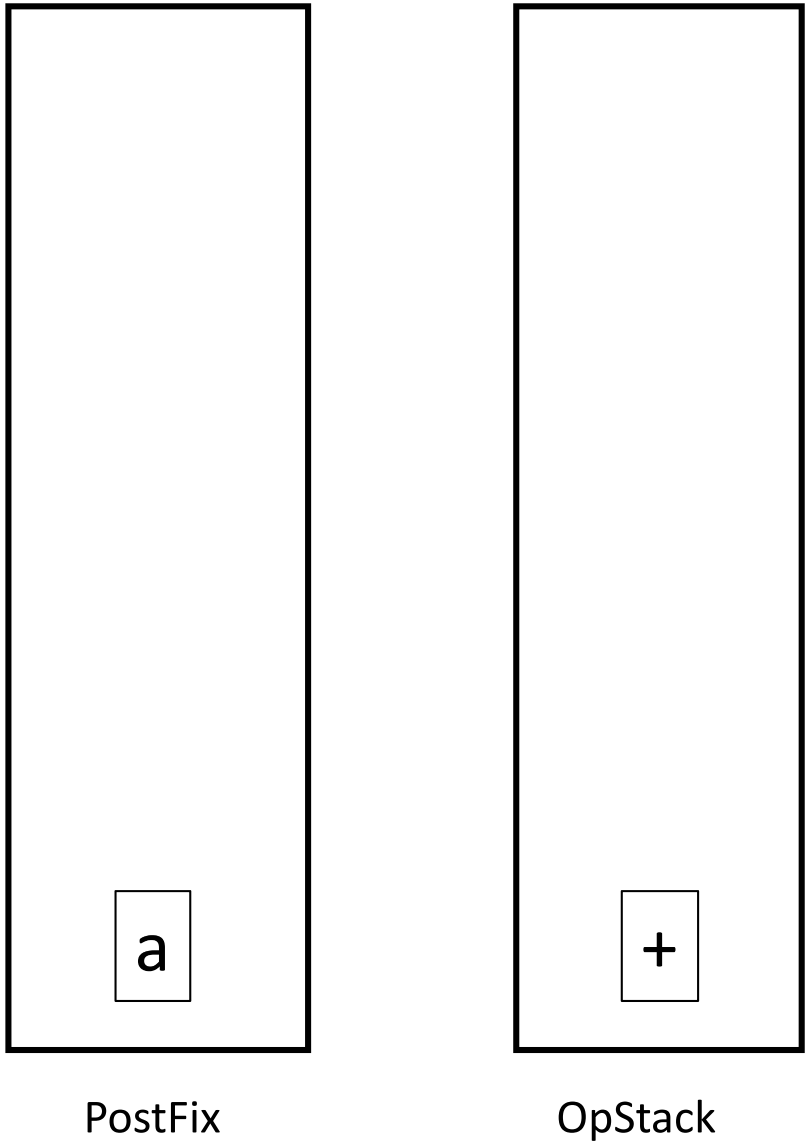
- token: a
- action: push a onto PostFix



Example

- infix notation: a + b * c / d

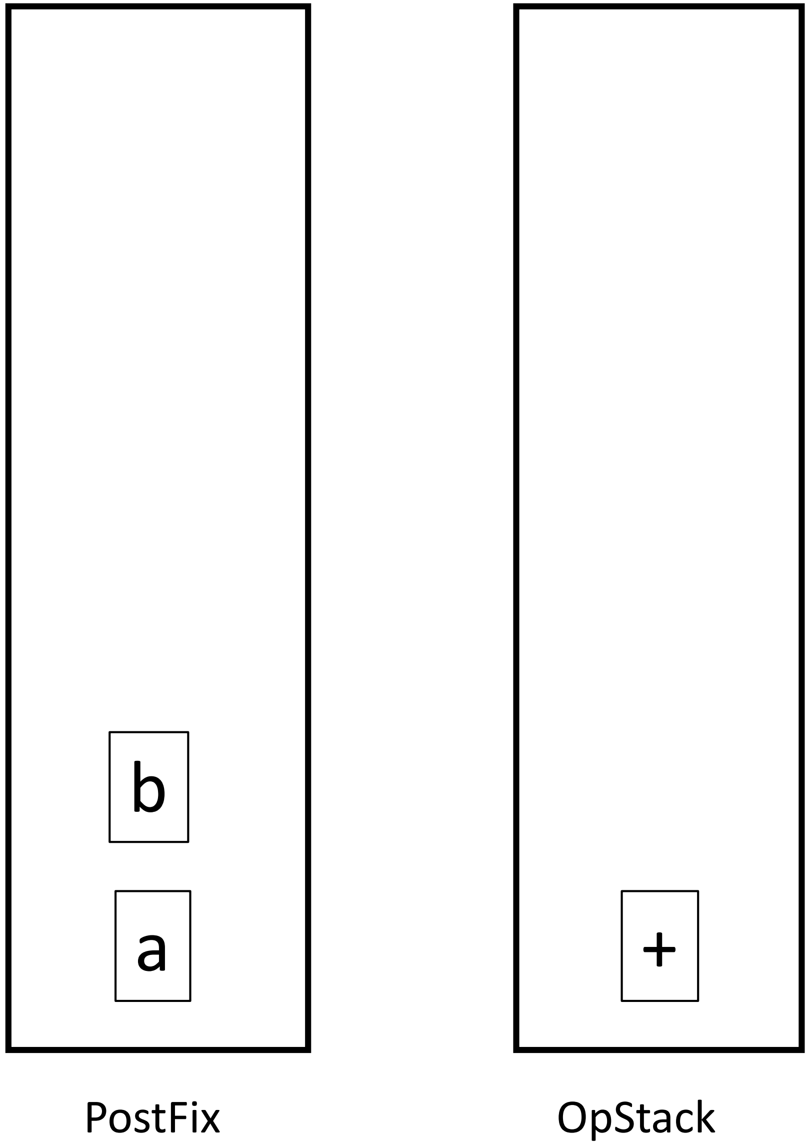
- token: +
- action: push + onto OpStack



Example

- infix notation: a + b * c / d

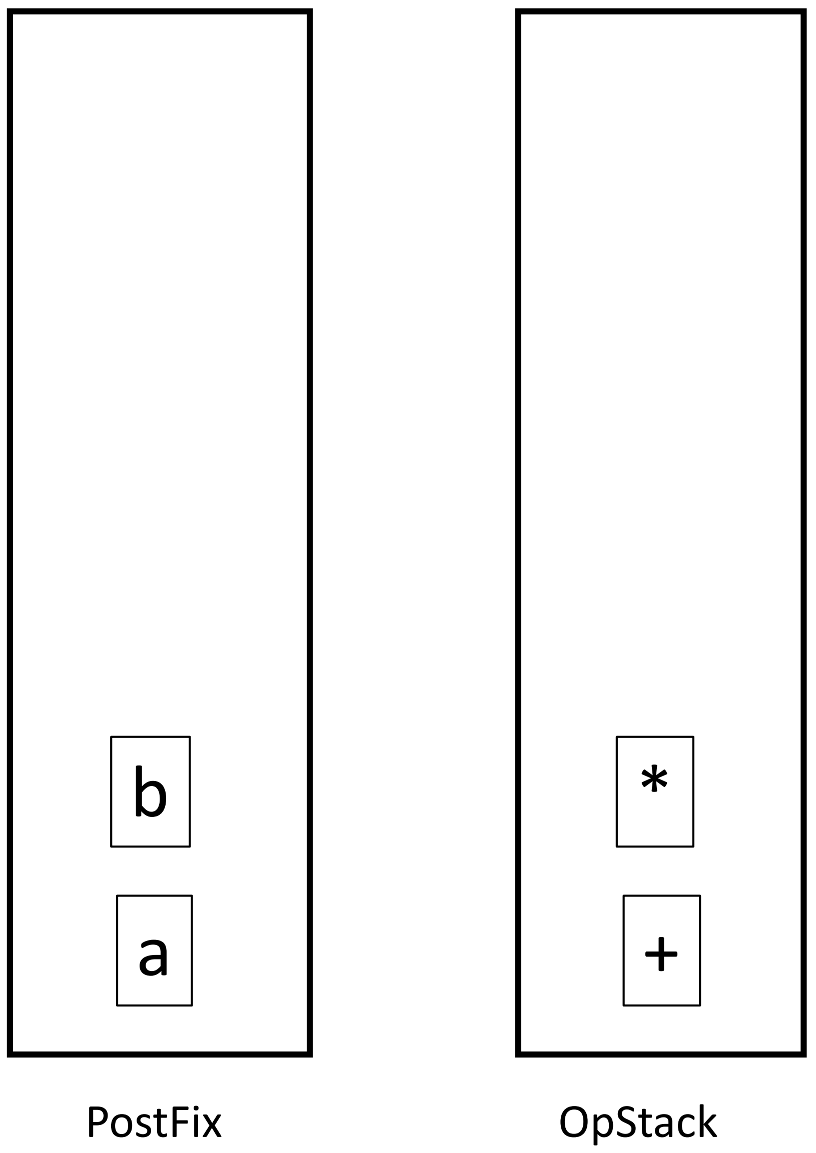
- token: b
- action: push b onto PostFix



Example

- infix notation: a + b * c / d

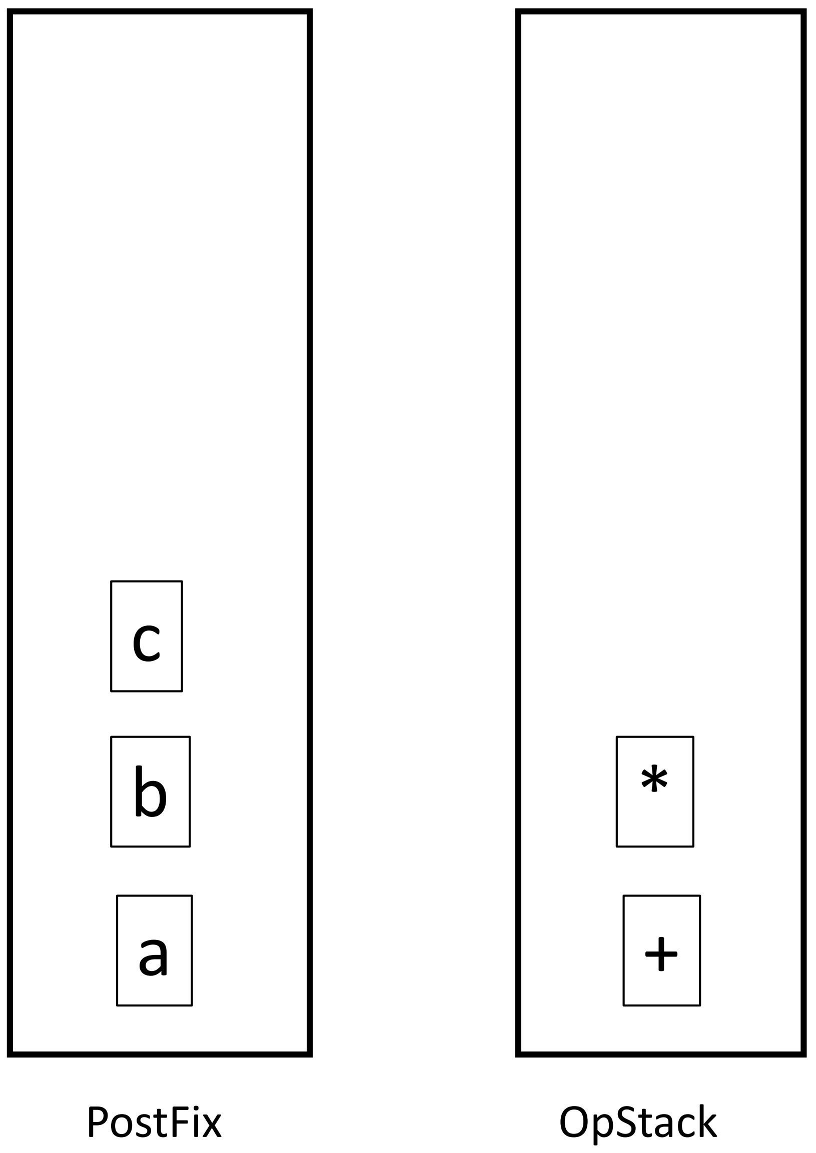
- token: *
- action: the top value of the OpStack (topOp is +) has lower precedence than * so we push * only OpStack



Example

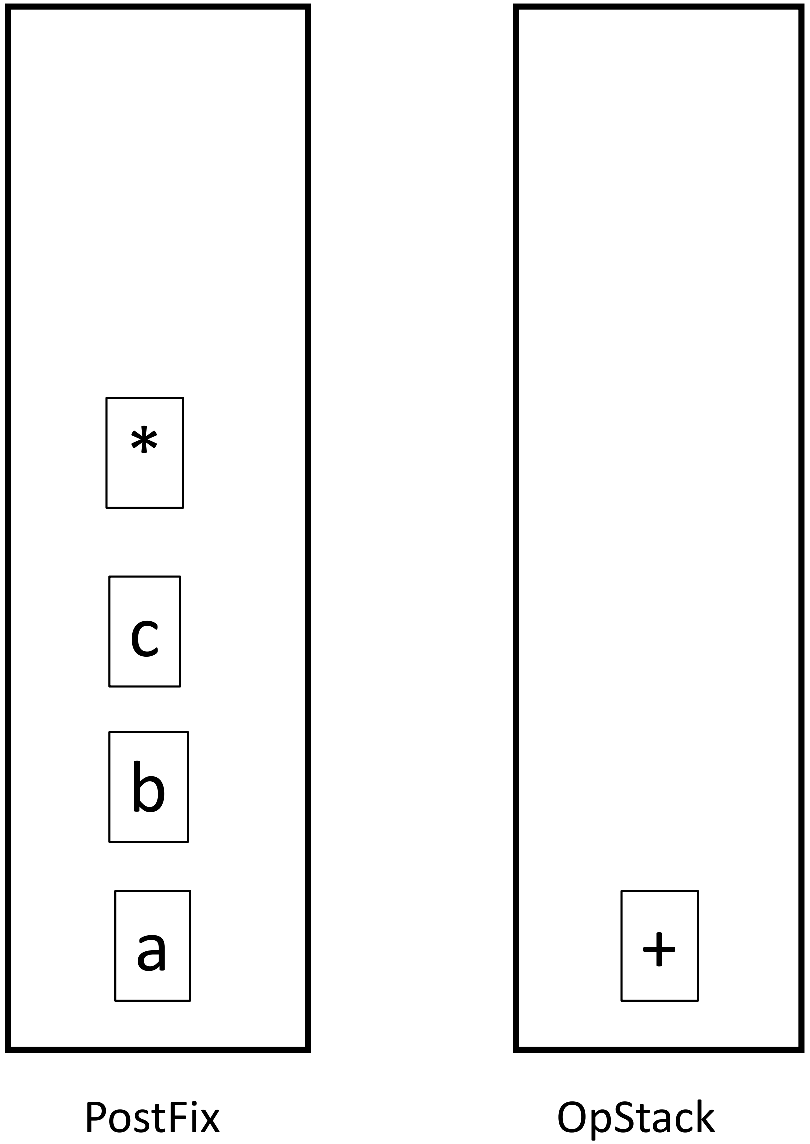
- infix notation: a + b * c / d

- token: c
- action: push c onto PostFix



Example

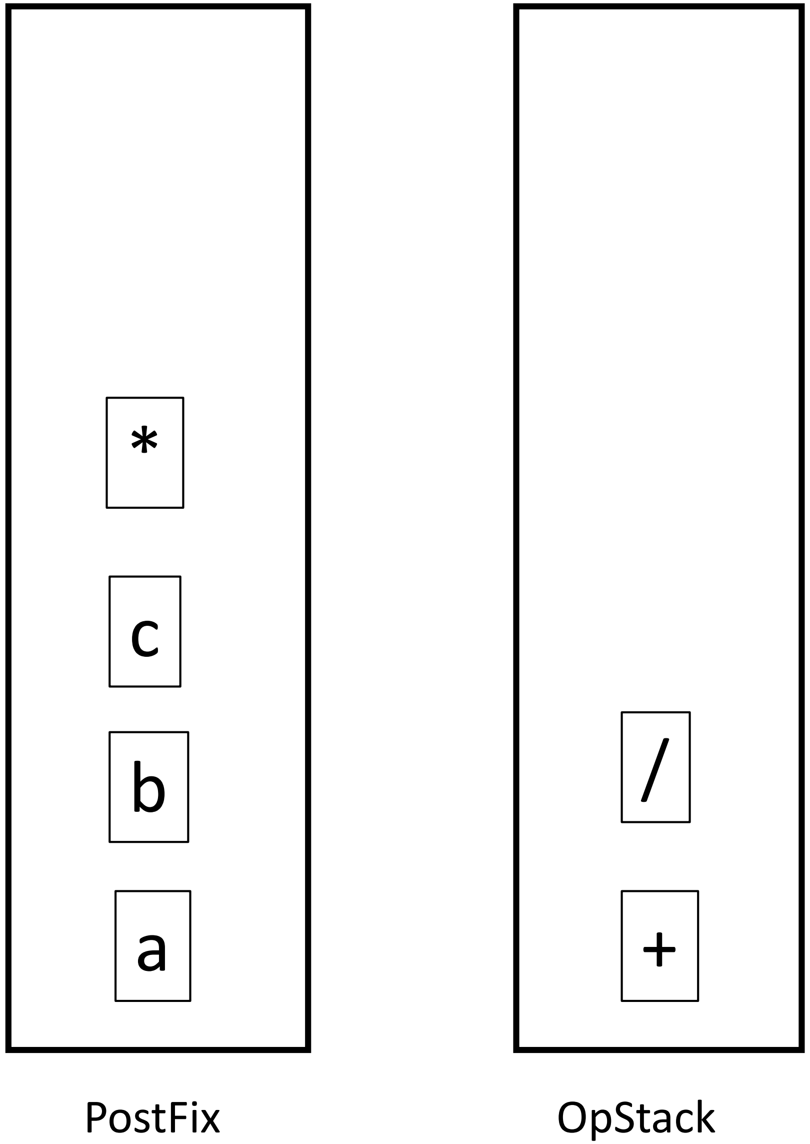
- infix notation: a + b * c / d
- token: /
- action: the top value of the OpStack (topOp is *) has **the same precedence** as /, so we pop topOp and push it onto the PostFix stack



Example

- infix notation: a + b * c / d

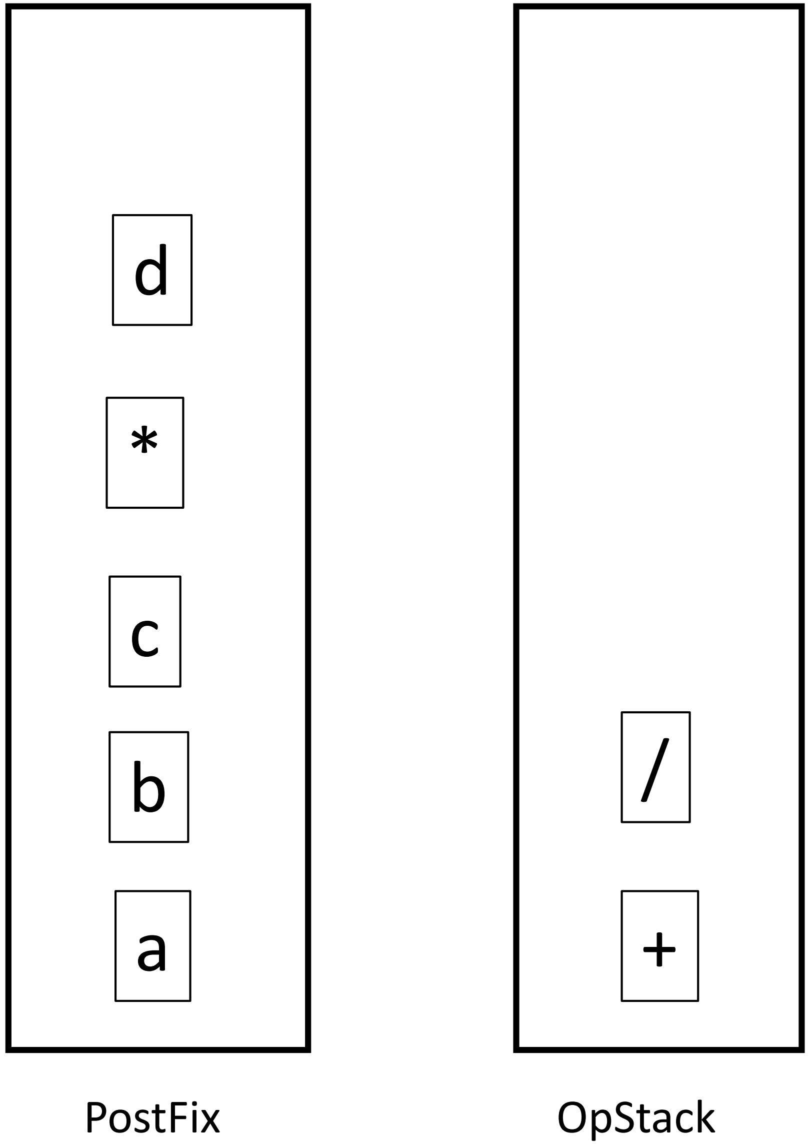
- token: / (continued!)
- action: the top value of the OpStack (topOp is +) has lower precedence than /, so we push / onto the OpStack



Example

- infix notation: a + b * c / d

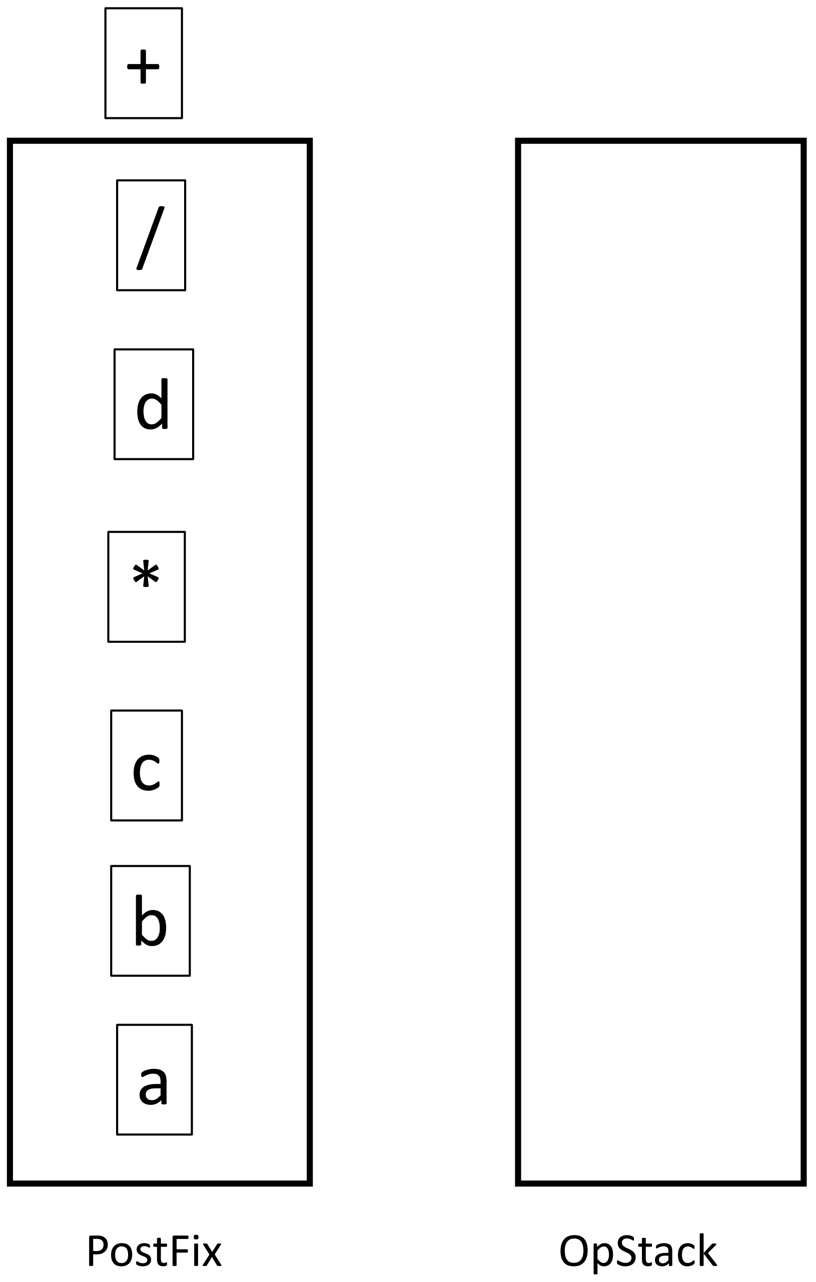
- token: d
- action: push d onto PostFix stack



Example

- infix notation: a + b * c / d

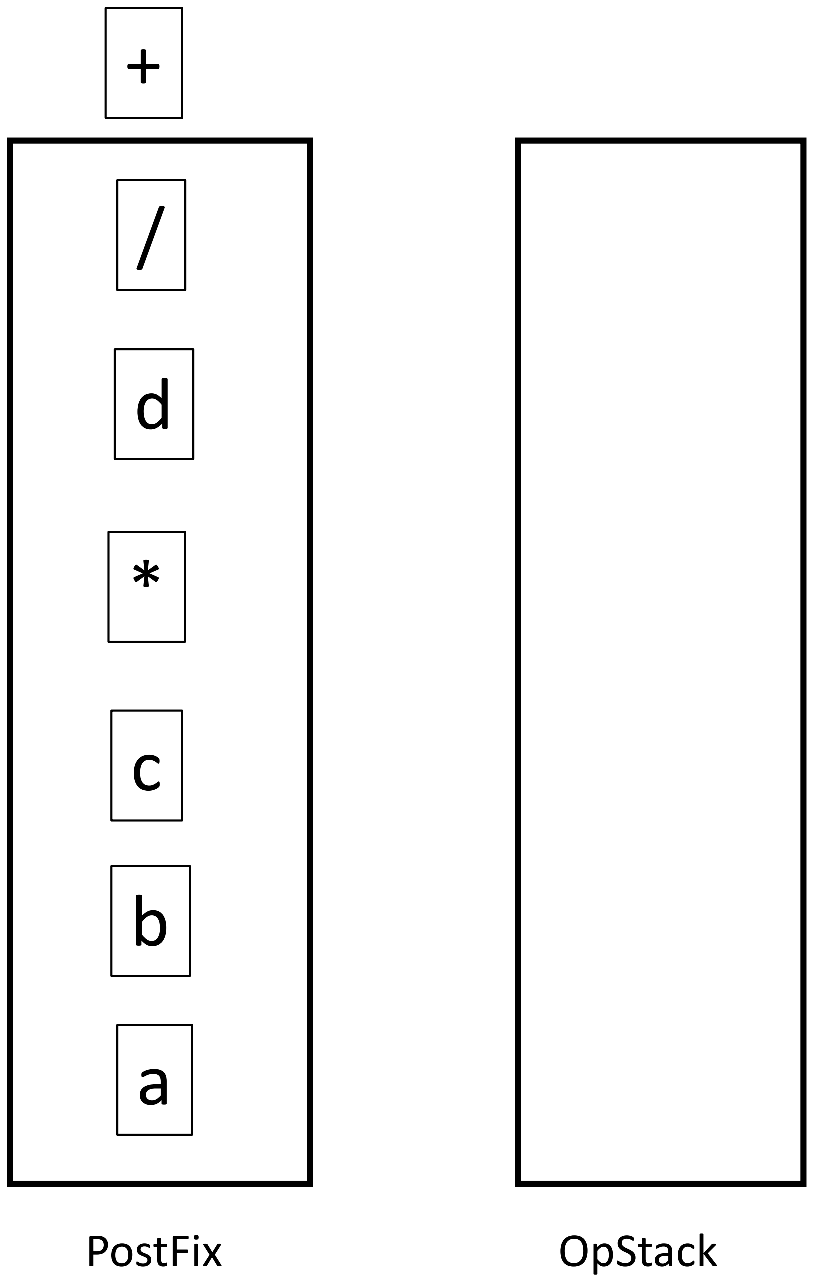
- token: none remain
- action: pop everything from OpStack and push it to PostFix



Example

- infix notation: a + b * c / d

- postfix notation: read from bottom up
- a b c * d / +



Evaluating Postfix Expressions

- We can evaluate a postfix expression with a stack.

- Note: to evaluate the expression generated from the previous algorithm, we'd need to reverse the order (because we read from bottom up).
  - How can we do that?

while there are tokens in the postfix expression

if the token is a value

push the value onto ValueStack

else (the token is an operator)

pop one or two operands (**pop operand2 first and then operand1**) from ValueStack (depending on whether the operator is unary or binary)

Important Note: The first popped number is the *second* operand. Example: 3 6 / evaluates to 0 (3/6), **not** 2 (6/3).

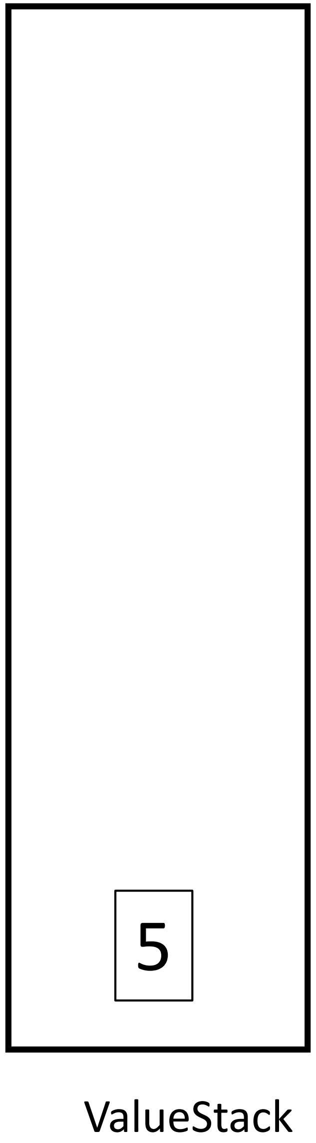
perform the operation

push the result back onto ValueStack

Example

- postfix notation: 5 1 2 + *

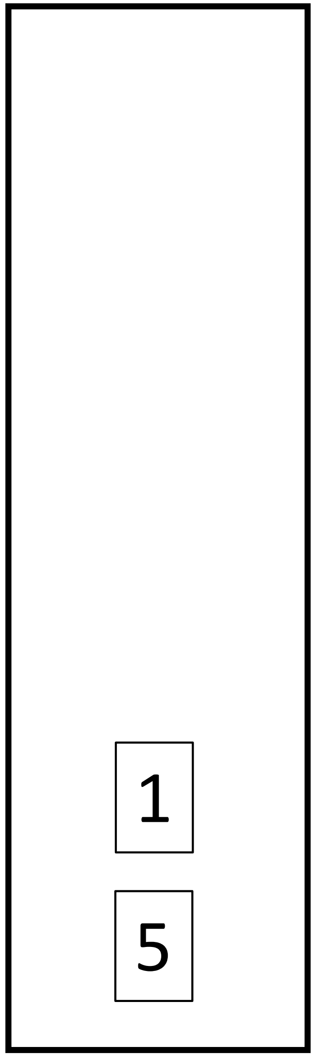
- token: 5
- action: push 5 onto ValueStack



Example

- postfix notation: 5 1 2 + *

- token: 1
- action: push 1 onto ValueStack

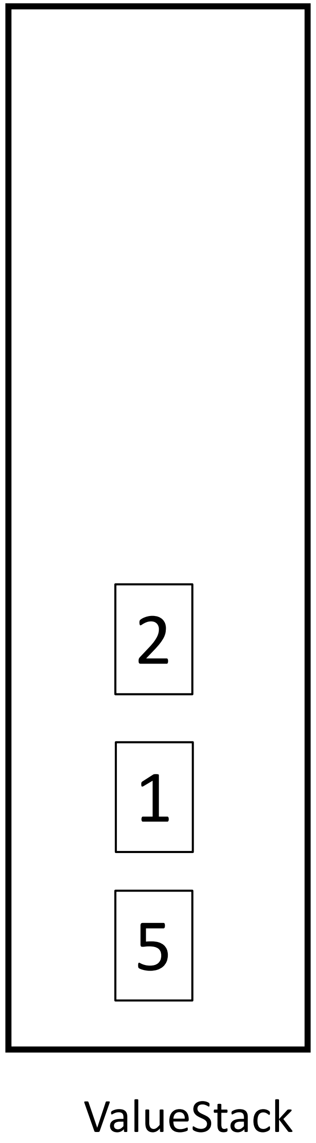


ValueStack

Example

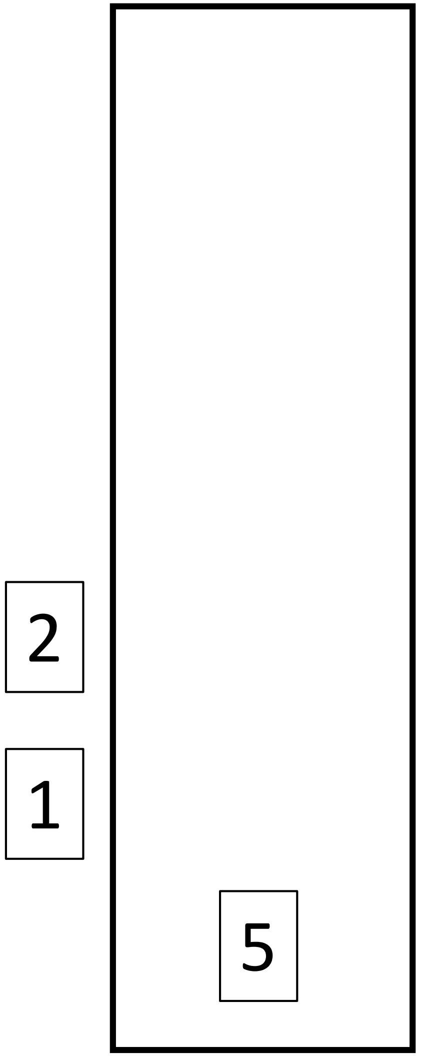
- postfix notation: 5 1 2 + *

- token: 2
- action: push 2 onto ValueStack



Example

- postfix notation: 5 1 2 + *
- token: +
- action: pop two values from the stack- **second operand pops off first!**
  - operand2: 2
  - operand1: 1

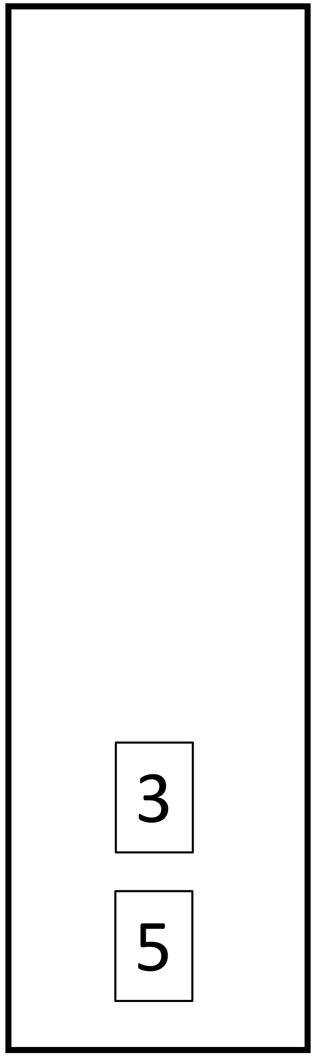


ValueStack

Example

- postfix notation: 5 1 2 + *

- token: + (continued)
- action: perform the calculation and push the result onto ValueStack
  - operand2: 2
  - operand1: 1
  - 1 + 2 = 3

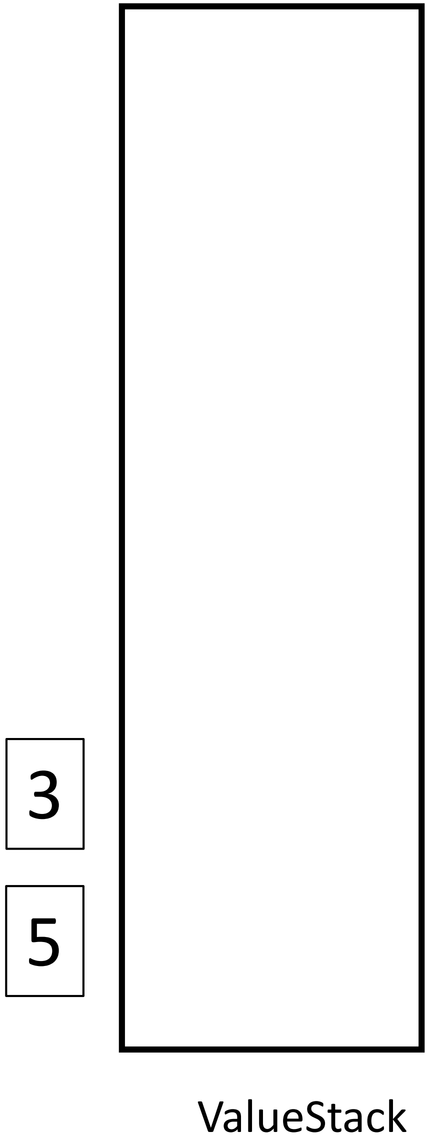


ValueStack

Example

- postfix notation: 5 1 2 + *

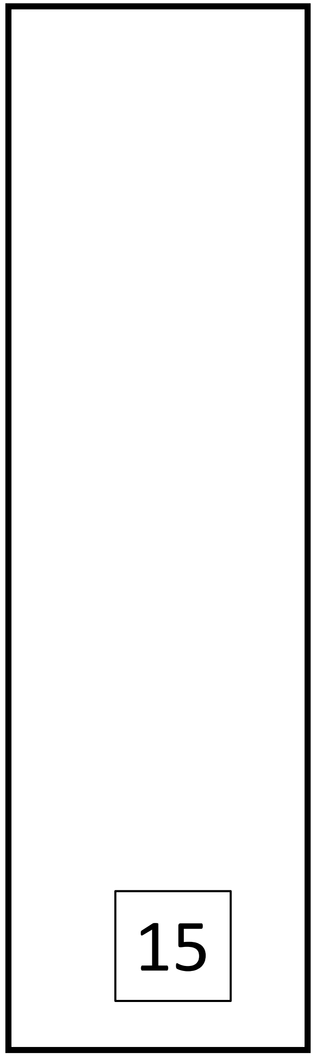
- token: *
- action: pop two values from the stack- **second operand pops off first!**
  - operand2: 3
  - operand1: 5



Example

- postfix notation: 5 1 2 + *

- token: *
- action: (continued)
- action: perform the calculation and push the result onto ValueStack
  - operand2: 3
  - operand1: 5
  - 5 * 3 = 15



ValueStack

Implementing Stacks with Nodes

- Implementing stacks with linked nodes is straightforward.
- firstNode (or head) is the top of the stack.
- Pushes are easy and O(1).
- Pops are easy and O(1).
- Very efficient!

Implementing with an Array

- This is a little tricker.
- There are four ways we could consider:
    1. the top of the stack is always at position 0
    2. the bottom of the stack is always at position 0
    3. the top of the stack is always at position length-1
    4. the bottom of the stack is always at position length-1

Implementing with an Array

1. the top of the stack is always at position 0
2. the bottom of the stack is always at position 0
3. the top of the stack is always at position length-1
4. the bottom of the stack is always at position length-1

	array[0]	array[1]	array[2]	array[3]	array[4]
version 1	topEntry	middleEntry	bottomEntry		
version 2	bottomEntry	middleEntry	topEntry		
version 3			bottomEntry	middleEntry	topEntry
version 4			topEntry	middleEntry	bottomEntry

Implementing with an Array

1. the top of the stack is always at position 0
   - top of stack stored at 0
   - when we push, we have to shift down
   - when we pop, we have to shift up
   - inefficient!

	array[0]	array[1]	array[2]	array[3]	array[4]
version 1	topEntry	middleEntry	bottomEntry		
version 2	bottomEntry	middleEntry	topEntry		
version 3			bottomEntry	middleEntry	topEntry
version 4			topEntry	middleEntry	bottomEntry

Implementing with an Array

2. the bottom of the stack is always at position 0
    - when we add, just put the new element in the next available place
    - no shifting
    - more efficient! (O(1) pushes and pops)
    - need to keep track of the position associated with the topEntry

	array[0]	array[1]	array[2]	array[3]	array[4]
version 1	topEntry	middleEntry	bottomEntry		
version 2	bottomEntry	middleEntry	topEntry		
version 3			bottomEntry	middleEntry	topEntry
version 4			topEntry	middleEntry	bottomEntry

Stacks in Java

- Java provides a [Stack](#) class.
- This is one of the *legacy classes* that was designed with the very first Java version 1.0.