Algorithms and Data Structures

Linear Data Structures

dr Szymon Murawski

Comarch SA

March 28, 2019

Table of contents I

Course plan

Course plan

Primitive data types

- In programming types are used to represent data
- Primitive data types are the basis of constructing more advanced structures. They are also expected to be the fastest constructs the language has to offer.
- Each language offers different set of primitive data types, but usually they include:
  - **Integers** with different precision (`int`, `long`, `uint`)
  - **Floating-point numbers** with different precision and scale (`float`, `double`, `real`)
  - **Bools** with true/false value (`bool`)
  - **Characters** representing a single or number of symbols, digits, letters or control codes (`char`, `string`)
  - **References**, values referring to another object. Also known as pointers.
- Other built-in types offered in some languages include: complex numbers, tuples, associative arrays (hashes), first-class functions

Compound data types

- Compound/composite data types are types that can be constructed using primitive data types or other compound data types.
- Basic compound data types:
  - Arrays
  - Structures
  - Classes
  - Sets
  - Tuples
  - Strings
  - Enumerations
- Structures and classes behave VERY differently in each C-based language

Value and reference types

- Data types can either by a value or reference type
- Value of a value type is the actual value
- Value of a reference type is a reference to another value
- A reference allows program to fetch specific data
- A reference can be implemented as a memory address, pointer, file handle, URL etc.
- There are no global rules which data types are value or reference types, it all depends on programming language

Value and reference types

Language	Value types	Reference types
C++	booleans, characters, integer numbers, floating-point numbers, arrays, classes (including strings, lists, dictionaries, sets, stacks, queues), enumerations	pointers
C#	structures (including booleans, characters, integer numbers, floating-point numbers, fixed-point numbers, lists, dictionaries, sets, stacks, queues, optionals), enumerations	classes (including immutable strings, arrays, tuples, lists, dictionaries, sets, stacks, queues), interfaces, pointers
Java	booleans, characters, integer numbers, floating-point numbers	arrays, classes (including immutable strings, lists, dictionaries, sets, stacks, queues, enumerations), interfaces, null pointer
JavaScript		immutable booleans, immutable floating-point numbers, immutable symbols, immutable strings, undefined, prototypes (including lists, null pointer)
Python		classes (including immutable booleans, immutable integer numbers, immutable floating-point numbers, immutable complex numbers, immutable strings, byte strings, immutable byte strings, immutable tuples, immutable ranges, immutable memory views, lists, dictionaries, sets, immutable sets, null pointer)

Allocating memory

There are two ways that memory gets allocated for data storage:

- Compile Time (static) Allocation
  - Memory for named variables is allocated by the compiler
  - Exact size and type of storage must be known at compile time
  - For standard array declarations, this is why the size has to be constant
- Dynamic Memory Allocation
  - Memory allocated "on the fly" during run time
  - Dynamically allocated space usually placed in a program segment known as the heap or the free store
  - Exact amount of space or number of items does not have to be known by the compiler in advance.
  - For dynamic memory allocation, pointers are crucial

Dynamic memory allocation

- We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
- For this reason, dynamic allocation requires two steps:
  - Creating the dynamic space.
  - Storing its address in a pointer (so that the space can be accesed)
- To dynamically allocate memory in C++, we use the new operator.
- Deallocation is the "clean-up" of space being used for variables or other data storage
- De-allocation can be:
  - Manually done by a programmer via operations `free, delete`
  - Automatic, by a garbage collector (Java, C#)
  - Combination of above methods - smart pointers (C++),
- Improper handling of memory allocation can result in hard to detect errors - memory leaks!

Course plan

Abstract data structures

- Data types described before are static - they do not grow/shrink with the data
- It is also hard to represent complex data relations with them
- That's why we introduce abstract data structures (data structures from now on)
- Data structures can modify size and relationship between data dynamically during runtime
- Some algorithms require specific data structures to work
- Some problems are easily solvable just by using appropriate data structure
- Choosing a proper data structure for a problem has significant impact on effectiveness of implementation

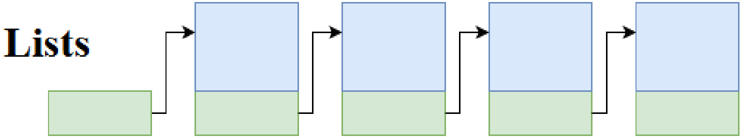
Complex relation - I'm my own grandpa by Ray Stevens

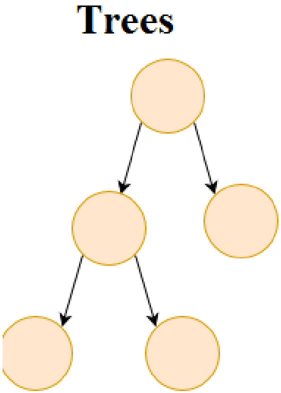
https://www.youtube.com/watch?v=qu_Y1wQ923g

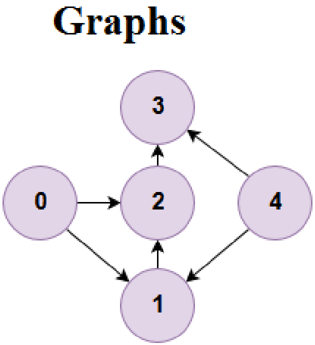
Many, many years ago when I was 23
I was married to a widow who was pretty as can be
This widow had a grown-up daughter who had hair of red
My father fell in love with her and soon they too were wed
This made my dad my son-in-law and really changed my life
For now my daughter was my mother 'cause she was my father's wife
And to complicate the matter even though it brought me joy
I soon became the father of a bouncing baby boy
My little baby then became a brother-in-law to dad
And so he became my uncle though it mad me very sad
For if he were my uncle that would also made him brother
Of the widows grown-up daughter who was of course my stepmother
Father's wife then had a son who kept them on the run
And he became my grandchild for he was my daughter's son
My wife is now my mother's mother and it makes me blue
Because although she is my wife she's my grandmother two
Now if my wife is my grandmother then I'm her grandchild
And every time I think of it, nearly drives me wild
'Cause now I have become the strangest case you ever saw

As husband of my grandmother I am my own grandpa

Example of data structures







Data structures requirements

Data structures to work properly require the following:

- Dynamic memory allocation
- Ability to make cyclic/infinite references
- Mechanism for creating references
- Ability to make many references to the same element
- Recurrent references

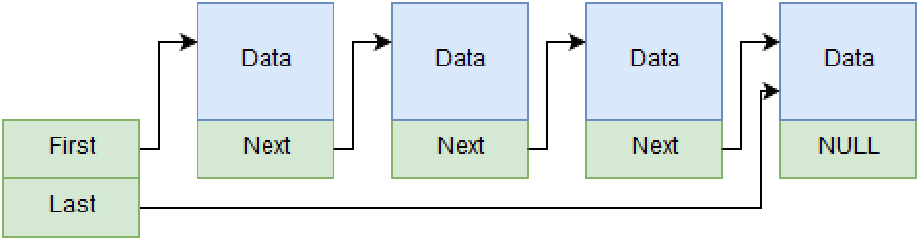
Operations on data structures

- Operations on data structures can be grouped in two categories
- **Queries**, which return information about data in structure. Examples:
    - Search - looks up a specific element
    - Minimum - fetches minimum value
    - Successor - returns reference to next element
    - Predecessor - returns reference to previous element
- **Modifying operations**, which modify the data. Examples:
    - Insert - insert new element into structure
    - Delete - deletes an element
    - Empty - deletes all data
    - Pop - deletes an element and returns its value

Course plan

Linked list

- Linked lists are the most basic dynamic data structures
- Linked lists are a set of nodes, each containing some data and a reference to the next node
- Contrary to arrays, order of the elements is not given by their physical placement in memory
- They are basis of other, more complicated data structures, mainly stacks and queues



Course plan

Implementation

```
class Node{
  Node next;
  Object data;
}

class LinkedList{
  Node First
  Node Last
  int Count
  void AddFirst(Object data)
  void AddLast(Object data)
  void RemoveFirst()
  void RemoveLast()
  Node SearchNode(Object data)
  void DeleteNode(Node node)
  void AddAfter(Object data, Node node)
}
```

Implementation part two

```
AddFirst(Object data){
  Node newNode = new Node(First, data);
  newNode.next = First;
  First = newNode;
  Count++;
}

AddLast(Object data){
  Node newNode = new Node(null, data);
  Last.next = newNode;
  Last = newNode;
  Count++;
}

Search(Object data){
  Node current = First;
  while current.data != data
    current = current.next;
  return current;
}
```

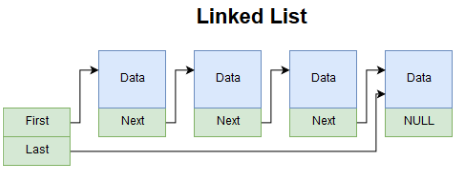
Implementation part three

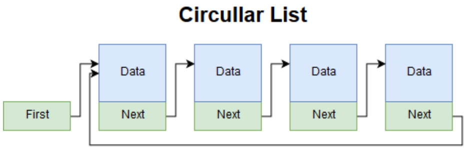
```
RemoveFirst(){
  return unless First != null
  First = First.Next;
  Count--;
}

RemoveLast(){
  return unless First != null
  Node secondToLastNode = First;
  while secondToLastNode.next != Last
    secondToLastNode = secondToLastNode.next;
  secondToLastNode.next = null;
  Last = secondToLastNode;
  Count--;
}
```

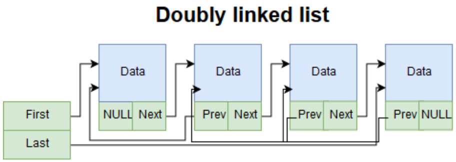
Implementation part four

```
DeleteNode (Node node){
  return unless First != null
  if (First == node)
    RemoveFirst()
  else if (Last == node)
    RemoveLast()
  else
    Node previousNode = First;
    while previousNode.next != node
      previousNode = previousNode.next;
    return unless previousNode != null;
    previousNode.next = node.next;
    Count--;
}
```

Linked lists types







Linked lists vs arrays

Complexity

- Accessing first or last object is $O(1)$
- Insertions are $O(1)$
- Removing first element is $O(1)$, last is $O(n)$ ($O(1)$ for doubly-linked list)
- Accessing any other element is $O(n)$
- Searching linked list is $O(n)$

- Linked lists allow for dynamic memory allocation, they resize during runtime. Arrays have fixed size.
- Removing or inserting an element in the middle of linked list is easy, for arrays it requires moving elements.
- Arrays require contiguous block of memory, linked list can be placed into many unconnected blocks.
- Linked lists do not allow for random access, only sequential access is possible.
- Linked lists require addition space for header info.

Course plan

Linked lists in C#

- Linked lists are available through System.Collections.Generic namespace
- `LinkedList<T>` is a generic class, meaning that `T` can be any kind of object
- Each element of `LinkedList<T>` is of class `LinkedListNode<T>`
- C# implements linked lists as doubly-linked lists

LinkedList<T>

Properties

- Count
- First
- Last

Methods

- AddAfter(LinkedListNode<T>, T), AddBefore(LinkedListNode<T>, T)
- AddFirst(T), AddLast(T)
- Clear() - deletes all nodes from list
- Contains(T) - true/false, depending on whether a node containing value exists
- CopyTo(T[], int) - copies whole list into array starting from specified index
- Find(T), FindLast(T)
- Remove(LinkedListNode<T>), Remove(T), RemoveFirst(), RemoveLast()

LinkedListNode<T>

Properties

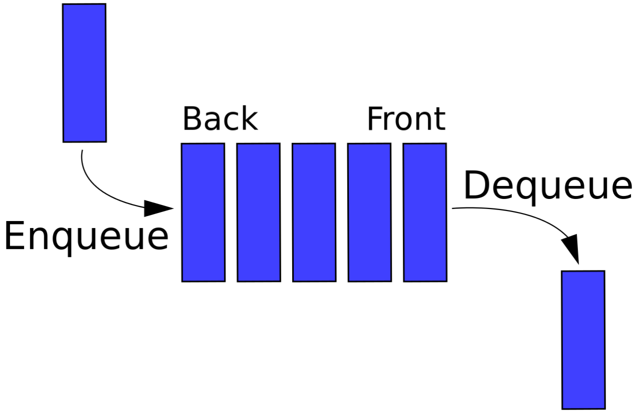
- List - get list, that this node belongs to
- Next - get next node in list
- Previous - get previous node in list
- Value - get value of node

Course plan

Queues

- Data structures that behaves just like real world queues
- Queues are a list structures with two access points called the **front** and **rear**.
- Insert operation is called **enqueue** it adds the element at the rear
- Delete operation is called **dequeue** it removes and returns the element from the front
- Queues have First-In, First-Out characteristics (FIFO)
- Only the oldest item is accessible
- Supportive functions
    - `peek()` - returns first element without removing it
    - `isEmpty()` - checks if queue is empty
    - `isFull()` - checks if queue is full, implementation will differ depending on specific problem

Queue visualization



Queues applications

Queues are used whenever a group of objects uses single shared resource. Because of that queues are widely.

- Job scheduling
- Serving http requests
- Printer queue
- Message passing between processes uses buffer - often implemented as a queue
- Breadth-first search on a graph

Queue<T>

Properties

- Count - number of elements in queue

Methods

- Clear() - clears queue of all data
- Contains(T) - checks whether queue contains specified element
- CopyTo(T[], int) - copies contents of queue into array T starting from specified index
- Dequeue() - removes and returns first element into the queue
- Enqueue(T) - inserts new element T into the queue
- Peek() - returns first element without removing it
- ToArray() - copies queue to new array of type T
- TrimExcess() - Sets the capacity of queue to the actual number of elements in queue

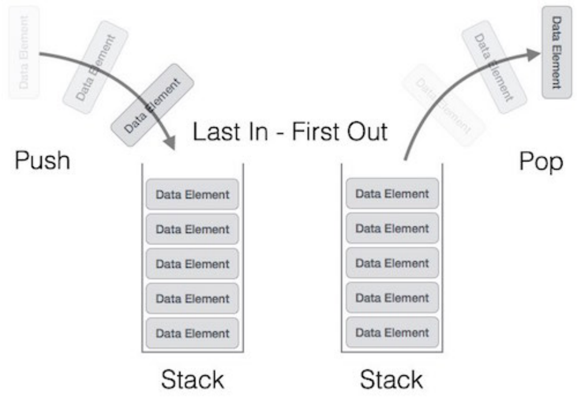
Priority Queue

- Priority queues are data structure in which every element has priority assigned to it
- Operation Enqueue takes one more argument - priority
- Dequeue operations always returns an element with the highest priority
- Simple implementation of priority queue would use an unordered linked-list as the backbone, resulting in $O(1)$ enqueue time, but $O(n)$ dequeue time
- In reality priority queues are often implemented as heaps with $O(lg\ n)$ complexity for both enqueue and dequeue
- We will return to the topic of priority queues in future lectures

Course plan

Stack

- Stack is a data structure, that has only one access point - top
- Stacks have Last-in, First-Out characteristics (LIFO)
- Sometimes also called FILO (First-in, Last-out) structures, but industry standard is to call them LIFO structures
- Behaves like stack of plates or cards - you can only access the top plate/card
- Insert operation is called `push` - it adds element at the top of the stack
- Delete operation is called pop - it removes element from the top of the stack
- Supportive functions
  - `peek()` - returns top element without removing it
  - `isEmpty()` - checks whether stack is empty
  - `isFull()` - checks if stack is full


Push
Data Element
Data Element
Data Element
Last In - First Out
Data Element
Data Element
Data Element
Data Element
Data Element
Stack
Data Element
Data Element
Data Element
Data Element
Data Element
Stack
Data Element
Data Element
Data Element
Pop

Stack<T>

Properties

- Count - number of elements in stack

Methods

- Clear() - clears stack of all data
- Contains(T) - checks whether stack contains specified element
- CopyTo(T[], int) - copies contents of stack into array T starting from specified index
- Peek() - returns first element without removing it
- Pop() - removes and returns an element from the top of the stack
- Push(T) - insert an element at the top of the stack
- ToArray() - copies stack to new array of type T
- TrimExcess() - Sets the capacity of stack to the actual number of elements in stack

Stack applications

- Reversing a word
- All "undo" mechanisms use stack to keep track of actions performed
- Traversing a maze - at each junction push it onto the stack, then pop the stack if you reached a dead end
- Function calls are kept on stack - if there are more calls than the stack could handle we got the infamous StackOverflowException

Traversing a graph

Searching

- We search graph using either Breadth or Depth-First Search algorithms
- In reality those algorithms are the same, they just use different data structure under the hood!
- BFS uses queue, DFS - stack

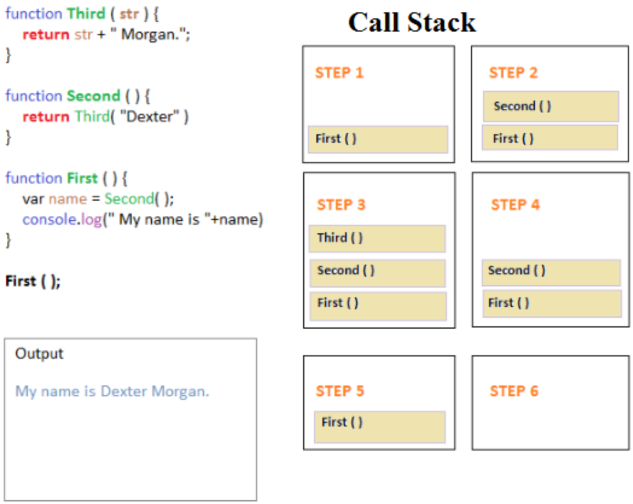
```
create [stack/queue]
push first node into [stack/queue]
while [stack/queue]
  pop/dequeue
  if found return
  else push all neighbours into [stack/queue]
```

Course plan

JavaScript

- JavaScript is a language used in web browsers to display interactive web pages
- Web browsers have dedicated engine to execute JavaScript
- JavaScript is single-threaded, meaning only one statement is executed at a time
- As JS engine processes script line by line, it pushes and pops function calls into and out of the stack
- For example: if engine steps into `foo()` instruction if pushes it into the stack and when engine reaches `return;` of that function id pops the `foo()` out of the stack

JavaScript Call Stack



```
function Third ( str ) {
   return str + " Morgan.";
}

function Second ( ) {
   return Third( "Dexter" )
}

function First ( ) {
   var name = Second( );
   console.log(" My name is "+name)
}

First ( );
```

```
Output

My name is Dexter Morgan.
```

Figure: Source: Siddhartha Chowdhury @medium.com

Asynchronous JavaScript

- If JavaScript could work only in synchronous mode, pages would take a long time to load
- During the load time it would also be impossible to interact with page
- The answer to that problem are asynchronous calls
- For example - when you open a facebook page only some part of the page is loaded initially, rest is downloaded asynchronously, so you can click on links or check messages
- For JavaScript to asynchronous callbacks to work three more pieces are required - WebApis, Event Loop and Message Queue
- JS asynchronous calls are redirected to specific Web API
- When asynchronous call is finished Web API enqueues new message into the message queue
- Event Loop checks if call stack is empty. If so, it pushes first message from message queue.

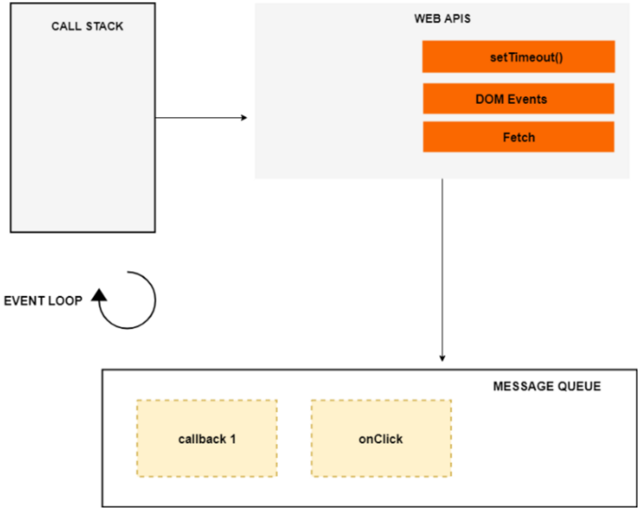


Figure: Source: blog.bitsrc.io

Course plan

Evaluating expressions

- Operators: $+ - /*$, operands (values): $2, 1, 5, 12$
- There are two main ways of how to write mathematical expression:
  - **Infix**, where the operators are in between operands: $2 + 3 * (7 + 5)$
  - **Postfix**, where the operators follow operands: $2375 + *+$
- Postfix, or Reverse Polish notation is used in calculators, as expressions written using postfix notation are evaluated faster
- No parenthesis are required in postfix notation!

Evaluating postfix expressions

Pseudocode

Input

-

```
postfixEvaluator(inputs)
  create stack
  for input in inputs
    if input is a number
      push input onto stack
    else
      op2 = stack.pop()
      op1 = stack.pop()
      output = evaluate(op1 input op2)
      push output onto stack
  return stack.pop
```

Evaluating postfix expressions - example

- Expression infix: $2 + 3 * (7 + 5)$
- Expression postfix: $2375 + * +$

expression	operation	stack
2375+*+	push 2	2
2375+*+	push 3	3 2
2375+*+	push 7	7 3 2
2375+*+	push 5	5 7 3 2
2375+*+	push $7 + 5$	12 3 2
2375+*+	push $3 * 12$	36 2
2375+*+	push $2 + 36$	38
2375+*+	pop	

Infix to postfix conversion

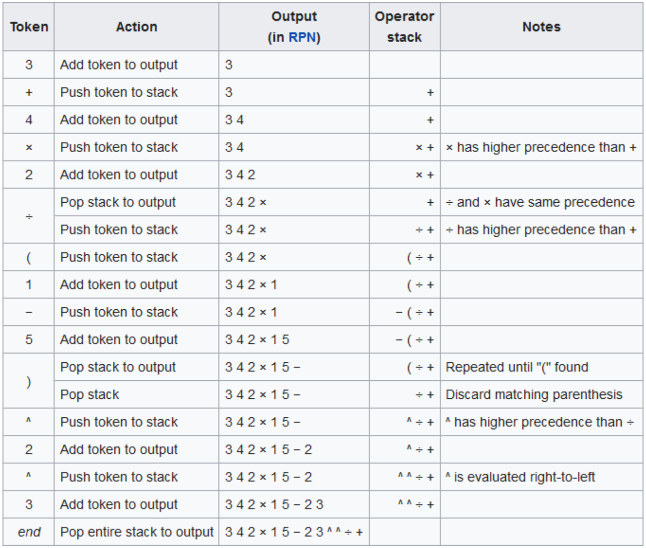
- To transform infix expression into postfix one we use Shunting-yard algorithm
- Algorithm was developed by Edsger Dijkstra
- In this algorithm we output operands as soon as we see them, but we keep operators on the stack
- Precedence of operators is very important in this algorithm, quick review: $\wedge, *, /, +, -$

Operator	Precedence	Associativity
$\wedge$	4	Right
$*$	3	Left
$/$	3	Left
$+$	2	Left
$-$	2	Left

Shunting-yard algorithm

```
InfixToPostfix(expression)
  create stack
  foreach token in expression
    if token is a number
      print token
    else if token is an operator
      while Precedence(stack.peek()) > Precedence(token)
          OR (Precedence(stack.peek()) == Precedence(token)
            AND Associativity(token) = 'left')
        print stack.pop()
      stack.push(token)
    else if token == '('
      stack.push(token)
    else if token == ')'
      while stack.peek() != '('
        print stack.pop()
      stack.pop() //To remove ( from top of the stack
  while stack.Count > 0
    print stack.pop()
```

Input: $3 + 4 * 2/(1 - 5) \wedge 2 \wedge 3$



Token	Action	Output (in RPN)	Operator stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
×	Push token to stack	3 4	× +	× has higher precedence than +
2	Add token to output	3 4 2	× +	
÷	Pop stack to output	3 4 2 ×	+	÷ and × have same precedence
	Push token to stack	3 4 2 ×	÷ +	÷ has higher precedence than +
(	Push token to stack	3 4 2 ×	(÷ +	
1	Add token to output	3 4 2 × 1	(÷ +	
−	Push token to stack	3 4 2 × 1	− (÷ +	
5	Add token to output	3 4 2 × 1 5	− (÷ +	
)	Pop stack to output	3 4 2 × 1 5 −	(÷ +	Repeated until "(" found
	Pop stack	3 4 2 × 1 5 −	÷ +	Discard matching parenthesis
^	Push token to stack	3 4 2 × 1 5 −	^ ÷ +	^ has higher precedence than ÷
2	Add token to output	3 4 2 × 1 5 − 2	^ ÷ +	
^	Push token to stack	3 4 2 × 1 5 − 2	^ ^ ÷ +	^ is evaluated right-to-left
3	Add token to output	3 4 2 × 1 5 − 2 3	^ ^ ÷ +	
end	Pop entire stack to output	3 4 2 × 1 5 − 2 3 ^ ^ ÷ +		

Figure: Source: Wikipedia