Stacks

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

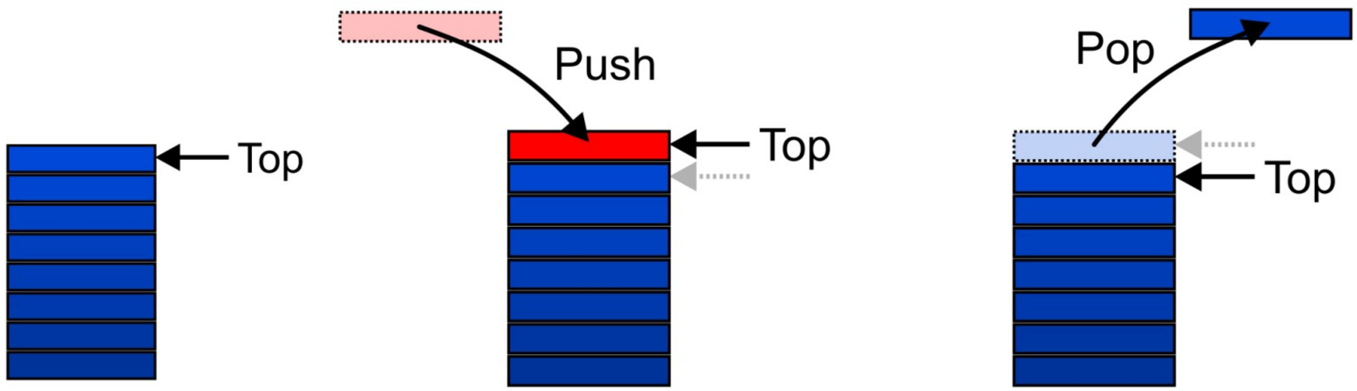
- This topic discusses the concept of a stack:
  - Description of an Abstract Stack
  - List applications
  - Implementation
  - Example applications
    - Parsing: XHTML, C++
    - Function calls
    - Reverse-Polish calculators
    - Robert's Rules
  - Standard Template Library

Abstract Stack

- An Abstract Stack (Stack ADT) is an abstract data type which emphasizes specific operations:
  - Insertions and removals are performed individually
  - Inserted objects are *pushed onto* the stack
  - The *top* of the stack is the most recently object pushed onto the stack
  - When an object is *popped* from the stack, the current *top* is erased

Abstract Stack

- Also called a *last-in–first-out* (LIFO) behaviour
  - Graphically, we may view these operations as follows:



Check more: https://en.wikipedia.org/wiki/Undefined_behavior

- There are two exceptions associated with abstract stacks:
  - It is an undefined operation to call either pop or top on an empty stack

Applications

- Numerous applications:
  - Parsing code:
    - Matching parenthesis
    - XML (e.g., XHTML)
  - Tracking function calls
  - Dealing with undo/redo operations
  - Reverse-Polish calculators

- The stack is a very simple data structure
  - Given any problem, if it is possible to use a stack, this significantly simplifies the solution

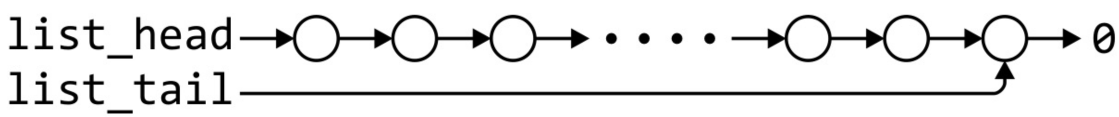
Stack: Implementations

- We will look at two implementations of stacks:
  - Singly linked lists
  - One-ended arrays

- Note: The optimal asymptotic run time of any algorithm is $\Theta(1)$
  - The run time of the algorithm is independent of the number of objects being stored in the container
  - We will always attempt to achieve this lower bound

Implementation: w/ Linked-List

- Operations at the front of a singly linked list are all $\Theta(1)$



	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

- The desired behavior of an Abstract Stack can be performed by all operations at the front of linked-list

Stack-as-List Class

- The stack class using a singly linked list has a single private member variable:

```
template <typename Type>
class Stack {
    private:
        Single_list<Type> list;
    public:
        bool empty() const;
        Type top() const;
        void push( Type const & );
        Type pop();
};
```

Stack-as-List Class

- The empty and push functions just call the appropriate functions of the Single_list class

```
template <typename Type>
bool Stack<Type>::empty() const {
    return list.empty();
}

template <typename Type>
void Stack<Type>::push( Type const &obj ) {
    list.push_front( obj );
}
```

Stack-as-List Class

- The top and pop functions, however, must check the boundary case:

```
template <typename Type>
Type Stack<Type>::top() const {
    if ( empty() ) {
        throw underflow();
    }

    return list.front();
}
```

```
template <typename Type>
Type Stack<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    return list.pop_front();
}
```

Implementation: w/ Array

- For one-ended arrays, all operations at the back are $\mathbf{\Theta}(1)$

	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Stack-as-Array Class

- We need to store an array:
  - In C++, this is done by storing the address of the first entry

```
template <typename Type>
class Stack {
    private:
        int stack_size;
        int array_capacity;
        Type *array;
    public:
        Stack( int = 10 );
        ~Stack();
        bool empty() const;
        Type top() const;
        void push( Type const & );
        Type pop();
};
```

Empty

- The stack is empty if the stack size is zero:

```
template <typename Type>
bool Stack<Type>::empty() const {
    return ( stack_size == 0 );
}
```

Top

- If there are n objects in the stack, the last is located at index $n - 1$

```
template <typename Type>
Type Stack<Type>::top() const {
    if ( empty() ) {
        throw underflow();
    }

    return array[stack_size - 1];
}
```

Pop

- Removing an object simply involves reducing the size
  - By decreasing the size, the previous top of the stack is now at the location `stack_size`

```
template <typename Type>
Type Stack<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    --stack_size;
    return array[stack_size];
}
```

Push

- Pushing an object onto the stack can only be performed if the array is not full

```
template <typename Type>
void Stack<Type>::push( Type const &obj ) {
    if ( stack_size == array_capacity ) {
        throw overflow();  // return ??
    }

    array[stack_size] = obj;
    ++stack_size;
}
```

Exceptions

- The case where the array is full is not defined in the Abstract Stack

- If the array is filled, we have five options:
  - Increase the size of the array
  - Throw an exception
  - Ignore the element being pushed
  - Replace the current top of the stack
  - Put the pushing process to "sleep" until something else removes the top of the stack

Array Capacity

- If dynamic memory is available, you can increase the array capacity

- If we increase the array capacity, the question is:
  - How much?
  - 1) By a constant? `array_capacity += c;`
  - 2) By a multiple? `array_capacity *= c;`

Array Capacity Enlargement and Run times

- First, we recognize that any time that we push onto a full stack, this requires to copy n items and the run time is $\Theta(n)$

- Therefore, push is usually $\Theta(1)$ except when new memory is required

Array Capacity Enlargement and Run times

- To state the average run time, we will introduce the concept of **amortized time**:
  - If n operations requires $\Theta(f(n))$ in total, we will say that an individual operation has an amortized run time of $\Theta(f(n)/n)$
  - Therefore, if inserting n objects requires:
    - $\Theta(n^2)$ items to be copied, the amortized time is $\Theta(n)$
    - $\Theta(n)$ items to be copied, the amortized time is $\Theta(1)$

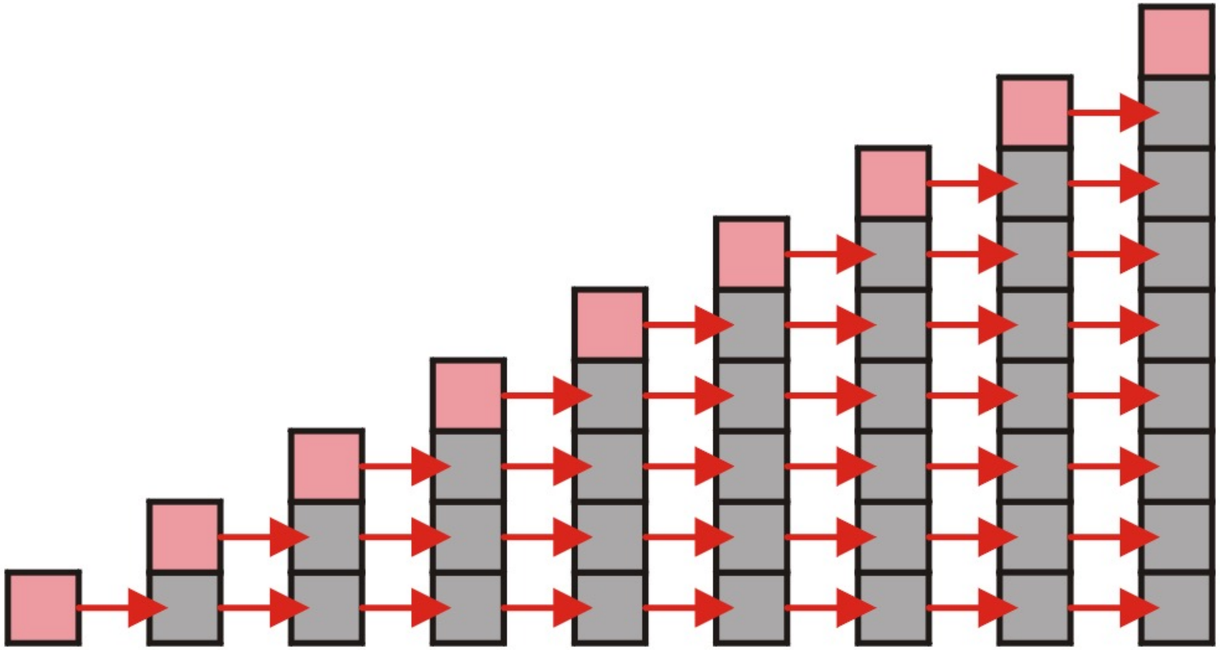
Definition

Amortized cost: Given a sequence of n operations, the amortized cost is:

$$\frac{\text{Cost}(n \text{ operations})}{n}$$

Array Capacity: Increase by 1

- Let us consider the case of increasing the capacity by 1 each time the array is full
  - With each insertion when the array is full, this requires all entries to be copied



Array Capacity: Increase by 1

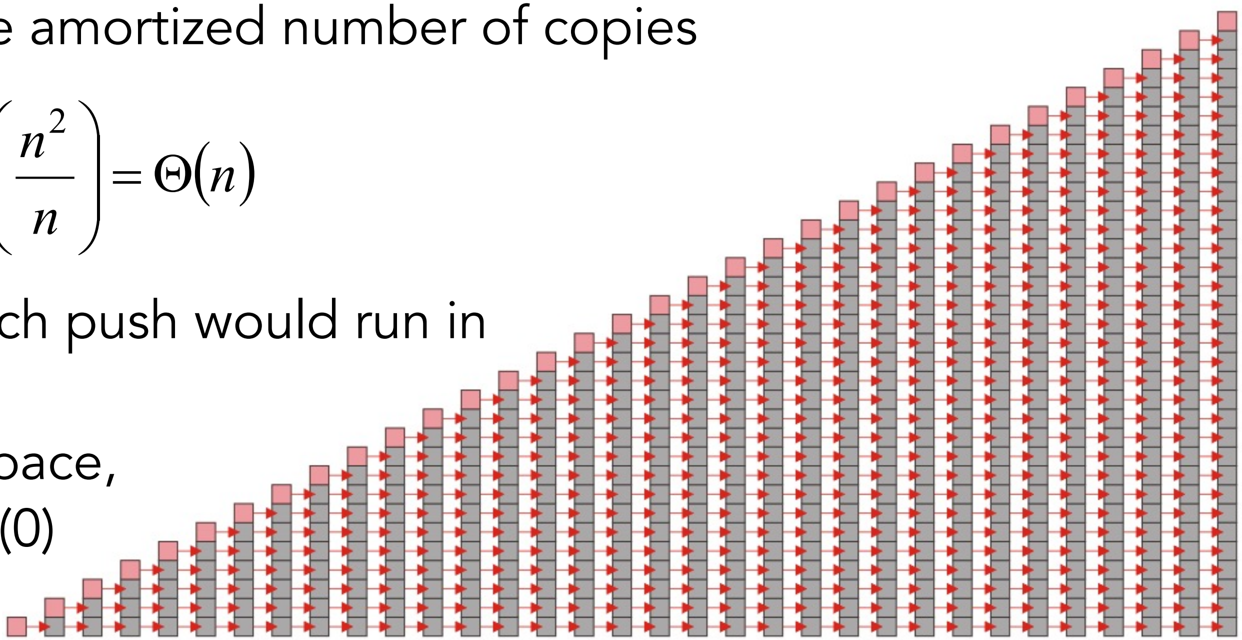
- Suppose we insert n objects
  - The pushing of the k^{th} object on the stack requires $k - 1$ copies
  - The total number of copies is now given by:

$$\sum_{k=1}^{n}(k-1) = \left(\sum_{k=1}^{n} k\right) - n = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} = \Theta\left(n^2\right)$$

  - Therefore, the amortized number of copies is given by

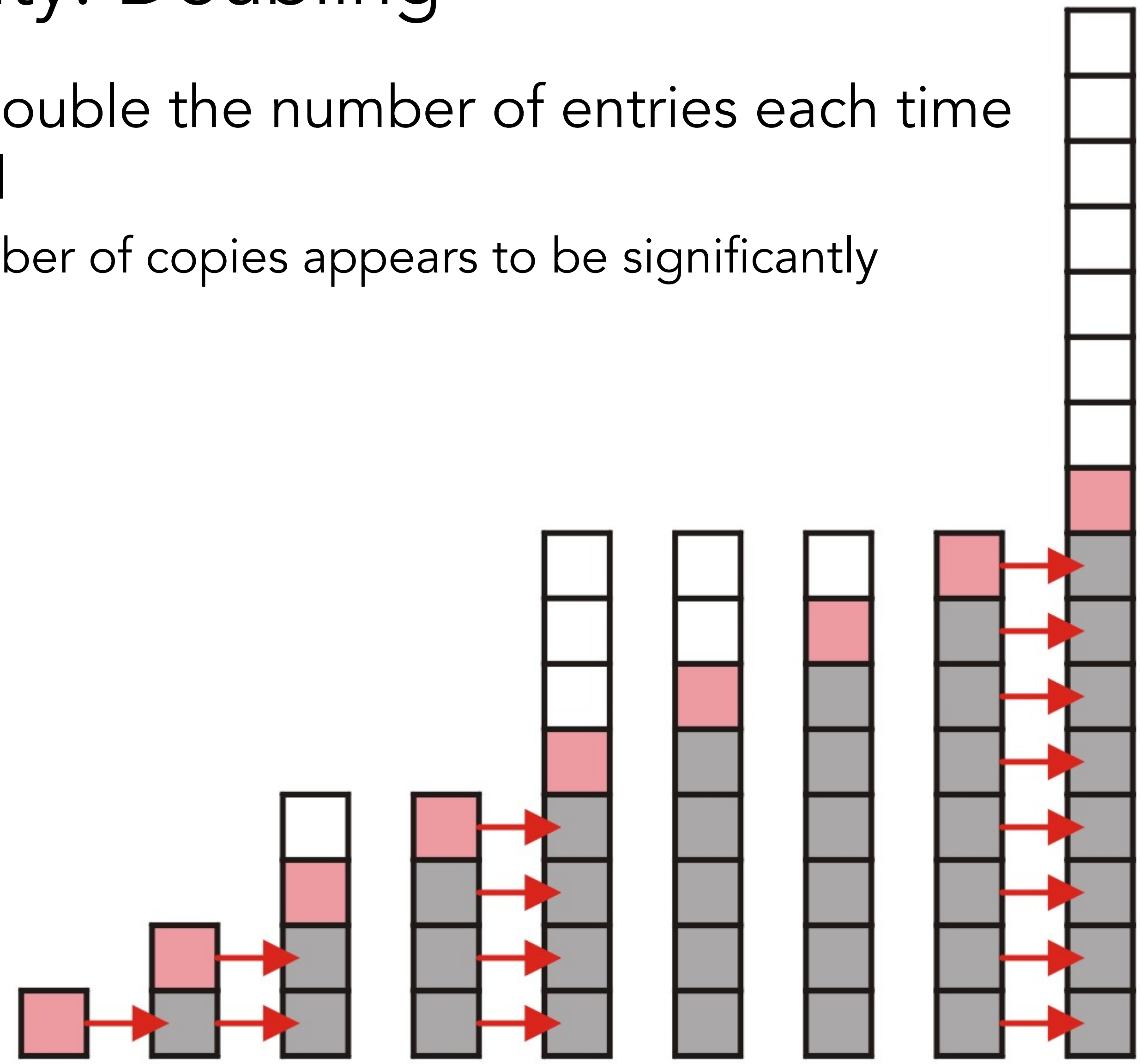
$$\Theta\left(\frac{n^2}{n}\right) = \Theta(n)$$

  - Therefore, each push would run in $\Theta(n)$ time
  - The wasted space, however, is $\Theta(0)$



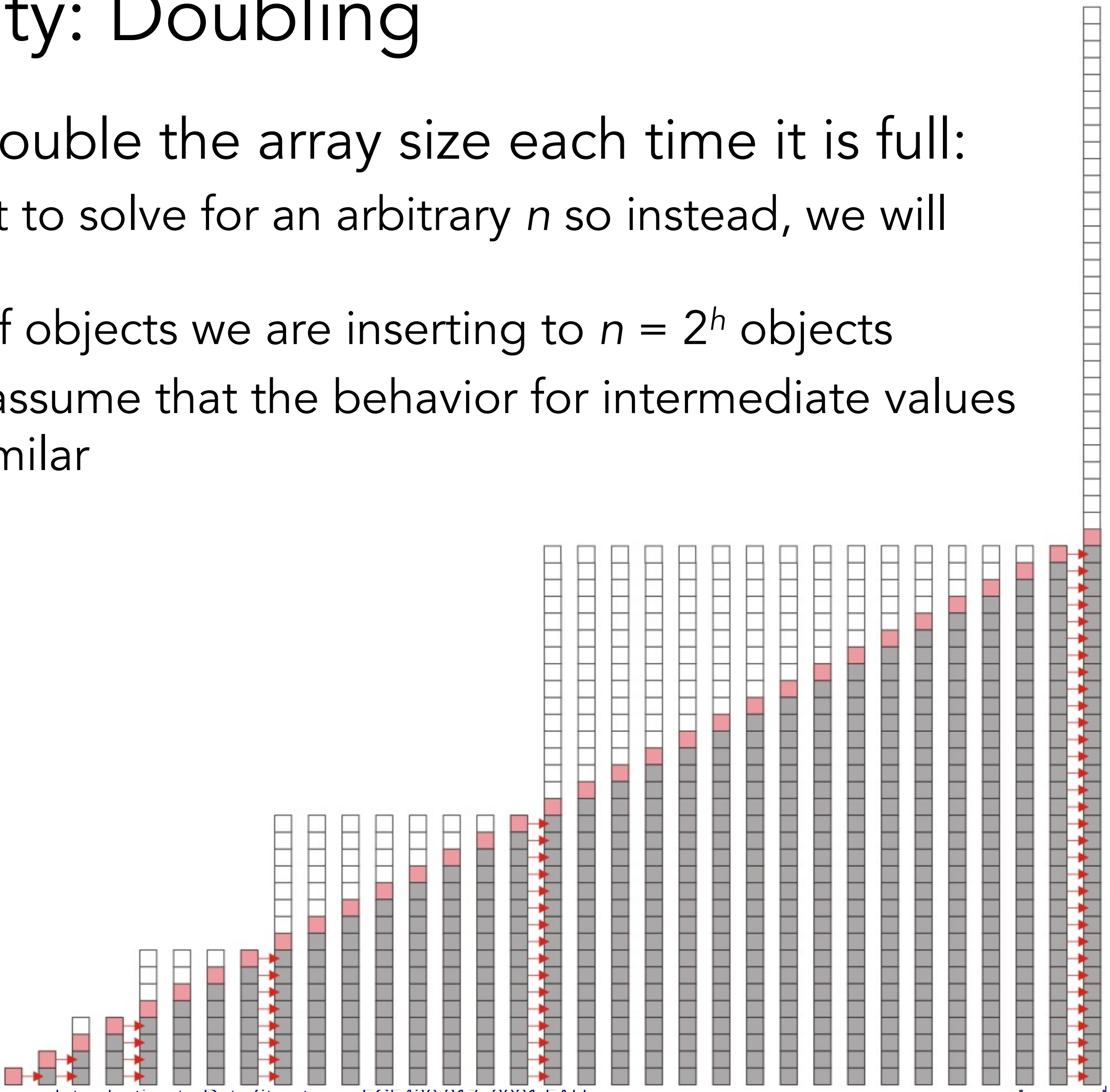
Array Capacity: Doubling

- Suppose we double the number of entries each time the array is full
  - Now the number of copies appears to be significantly fewer



Array Capacity: Doubling

- Suppose we double the array size each time it is full:
  - This is difficult to solve for an arbitrary n so instead, we will restrict
    the number of objects we are inserting to $n = 2^h$ objects
  - We will then assume that the behavior for intermediate values of n will be similar



Array Capacity: Doubling

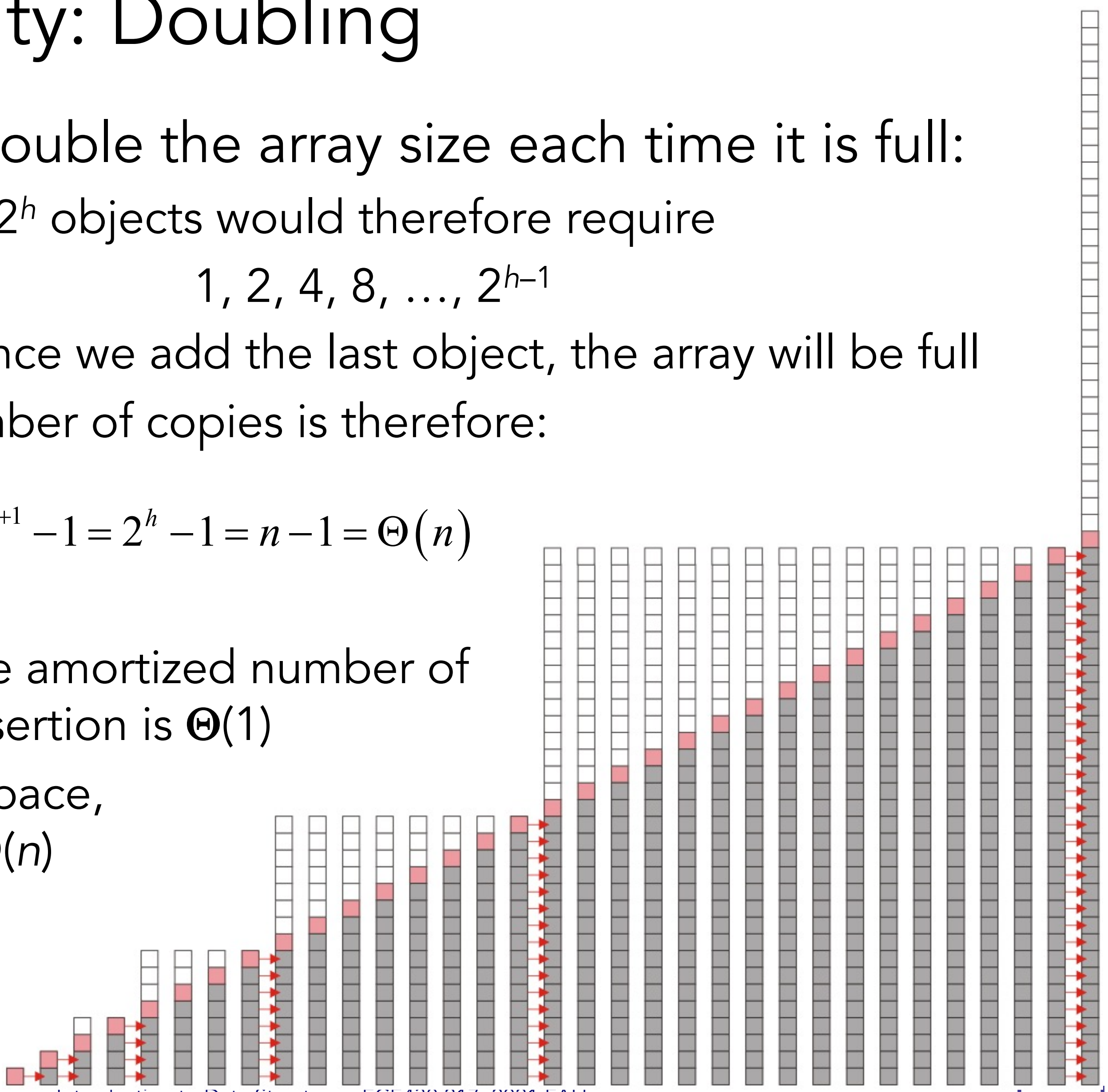
- Suppose we double the array size each time it is full:
  - Inserting $n = 2^h$ objects would therefore require

    $$1, 2, 4, 8, \ldots, 2^{h-1}$$

    copies, for once we add the last object, the array will be full
  - The total number of copies is therefore:

    $$\sum_{k=0}^{h-1} 2^k = 2^{(h-1)+1} - 1 = 2^h - 1 = n - 1 = \Theta(n)$$

  - Therefore, the amortized number of copies per insertion is $\Theta(1)$
  - The wasted space, however, is $O(n)$



NXC LAB

Application: Parsing

- Most parsing uses stacks

- Examples includes:
  - Matching tags in XHTML
  - In C++, matching
    - parentheses (...)
    - brackets, and [...]
    - braces { ... }

Parsing XHTML

- XHTML is made of nested
  - *opening tags*, e.g., `<some_identifier>`, and
  - matching *closing tags*, e.g., `</some_identifier>`

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

Parsing XHTML

- *Nesting* indicates that any closing tag must match the most <u>recent</u> opening tag

- Strategy for parsing XHTML:
  - read though the XHTML linearly
  - place the opening tags in a stack
  - when a closing tag is encountered, check that it matches what is on top of the stack

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

- We are finished parsing, and the stack is empty

- Possible errors:
  - a closing tag which does not match the opening tag on top of the stack
  - a closing tag when the stack is empty
  - the stack is not empty at the end of the document

Reverse-Polish Notation

- Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

- The operator is placed (inserted) between two operands

- One weakness: parentheses are required

$$(3 + 4) \times 5 - 6 = 29$$
$$3 + 4 \times 5 - 6 = 17$$
$$3 + 4 \times (5 - 6) = -1$$
$$(3 + 4) \times (5 - 6) = -7$$

Reverse-Polish Notation

- In Reverse-Polish Notation (RPN), the operations are placed first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$\rightarrow \quad 3 \;\; 4 \;\; + \;\; 5 \;\; \times \;\; 6 \;\; -$$

- Parsing reads left-to-right and performs any operation on the last two operands:

$$3 \;\; 4 \;\; + \;\; 5 \;\; \times \;\; 6 \;\; -$$

$$\rightarrow \quad 7 \;\; 5 \;\; \times \;\; 6 \;\; -$$

$$\rightarrow \quad 35 \;\; 6 \;\; -$$

$$29$$

RPN → https://en.wikipedia.org/wiki/Reverse_Polish_notation
PN or NPN → https://en.wikipedia.org/wiki/Polish_notation

3 4 + vs. + 3 4

Reverse-Polish Notation

- Other examples:

$$3 \; 4 \; 5 \; \times \; + \; 6 \; -$$
$$\rightarrow 3 \; 20 \; + \; 6 \; -$$
$$\rightarrow 23 \; 6 \; -$$
$$\rightarrow 17$$

$$3 \; 4 \; 5 \; 6 \; - \; \times \; +$$
$$\rightarrow 3 \; 4 \; -1 \; \times \; +$$
$$\rightarrow 3 \; -4 \; +$$
$$\rightarrow -1$$

Reverse-Polish Notation

- Benefits:
  - No ambiguity and no brackets are required
  - It is the same process used by a computer to perform computations:
    - operands must be loaded before performing the operation
  - Reverse-Polish can be processed using stacks

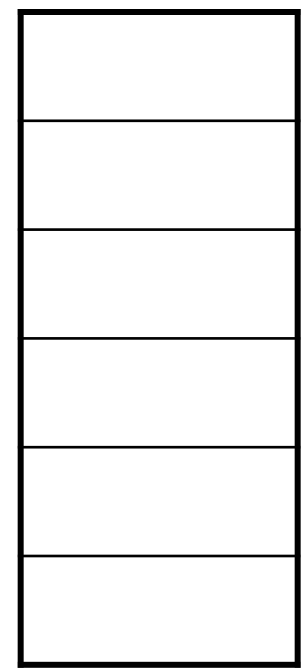
Reverse-Polish Notation

- The easiest way to parse reverse-Polish notation is to use an operand stack:
  - operands are processed by pushing them onto the stack
  - when processing an operator:
    - pop the last two items off the operand stack,
    - perform the operation, and
    - push the result back onto the stack

Reverse-Polish Notation

- Evaluate the following reverse-Polish expression using a stack:

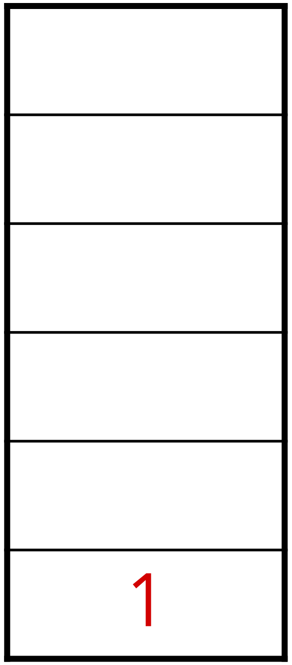
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

- Push 1 onto the stack

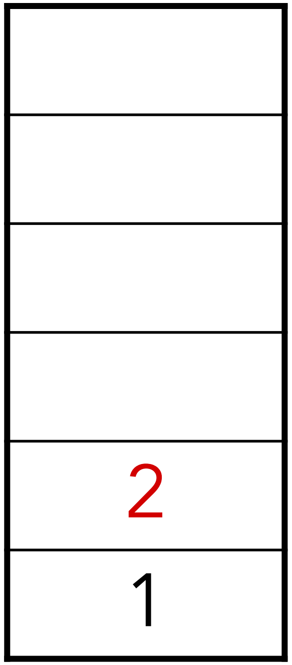
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Push 1 onto the stack

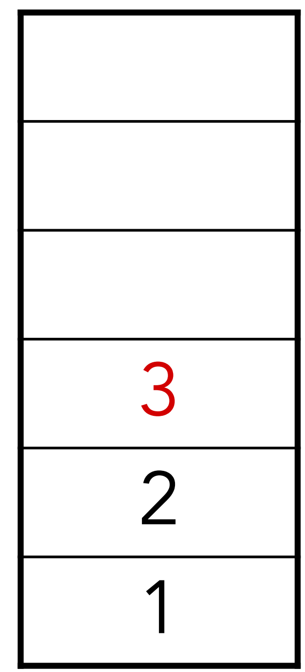
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Push 3 onto the stack

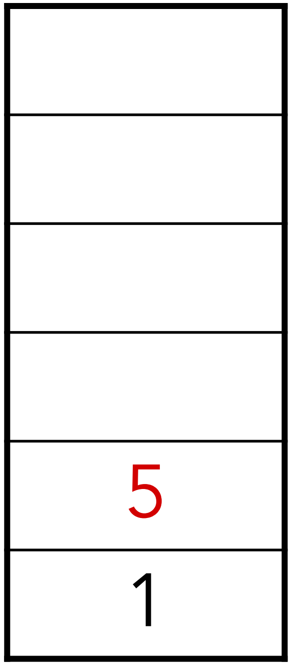
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Pop 3 and 2 and push 2 + 3 = 5

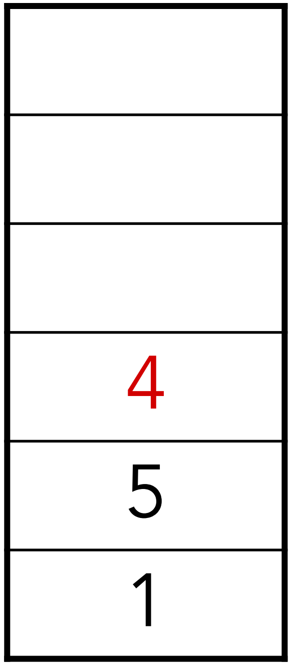
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Push 4 onto the stack

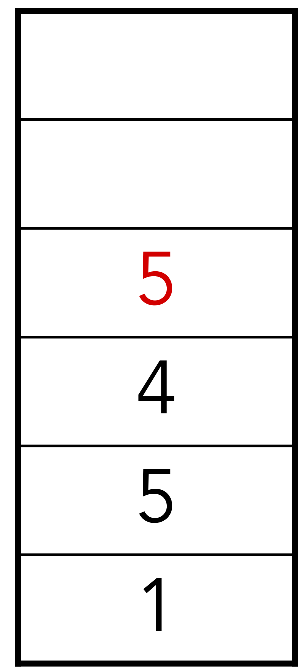
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Push 5 onto the stack

$$1\ \ 2\ \ 3\ +\ 4\ \ \color{red}{5}\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$$



Reverse-Polish Notation

- Push 6 onto the stack

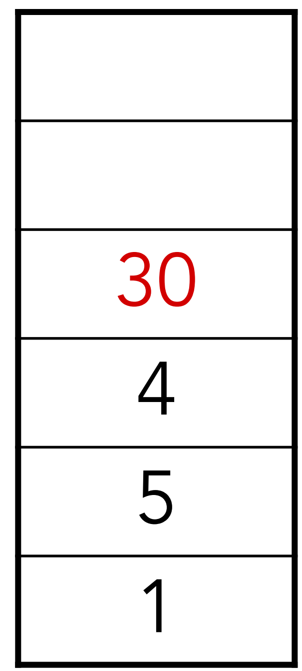
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

6
5
4
5
1

Reverse-Polish Notation

- Pop 6 and 5 and push 5 × 6 = 30

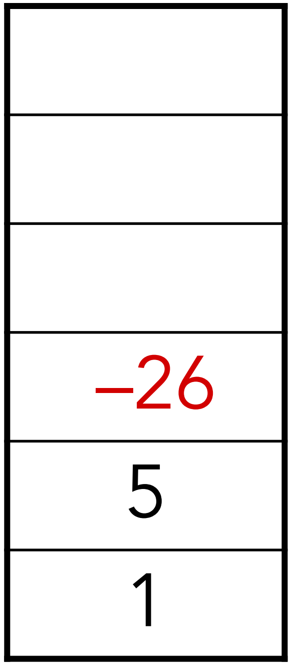
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Pop 30 and 4 and push 4 – 30 = –26

  1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Push 7 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

7
–26
5
1

Reverse-Polish Notation

- Pop 7 and –26 and push –26 × 7 = –182

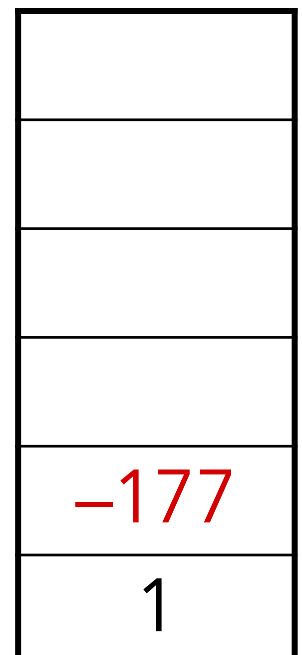
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

–182
5
1

Reverse-Polish Notation

- Pop –182 and 5 and push –182 + 5 = –177

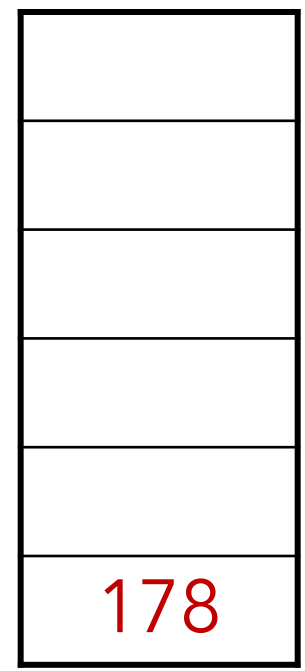
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Pop –177 and 1 and push 1 – (–177) = 178

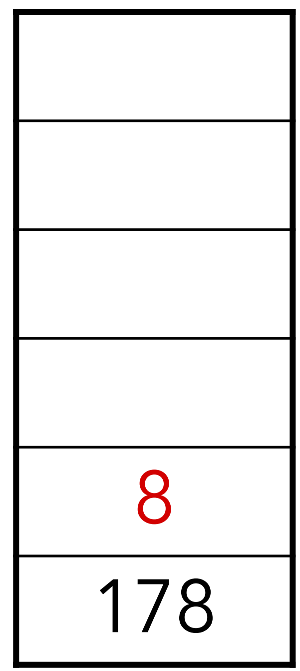
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

- Push 8 onto the stack

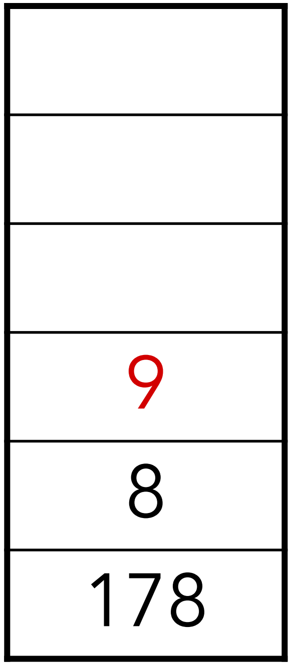
  1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Push 1 onto the stack

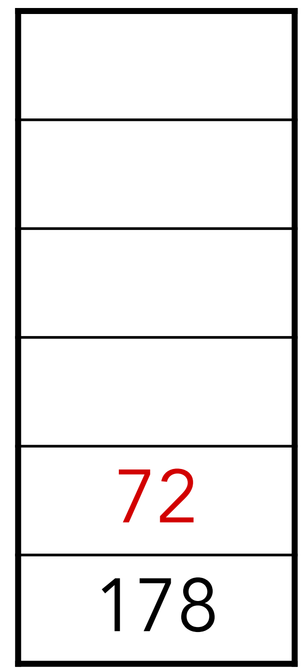
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

- Pop 9 and 8 and push 8 × 9 = 72

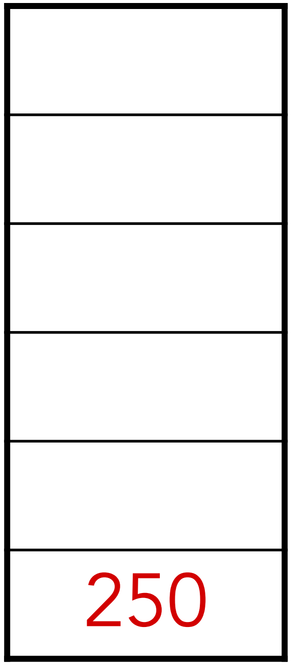
  1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

- Thus

$$1\ \ 2\ \ 3\ \ +\ \ 4\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$

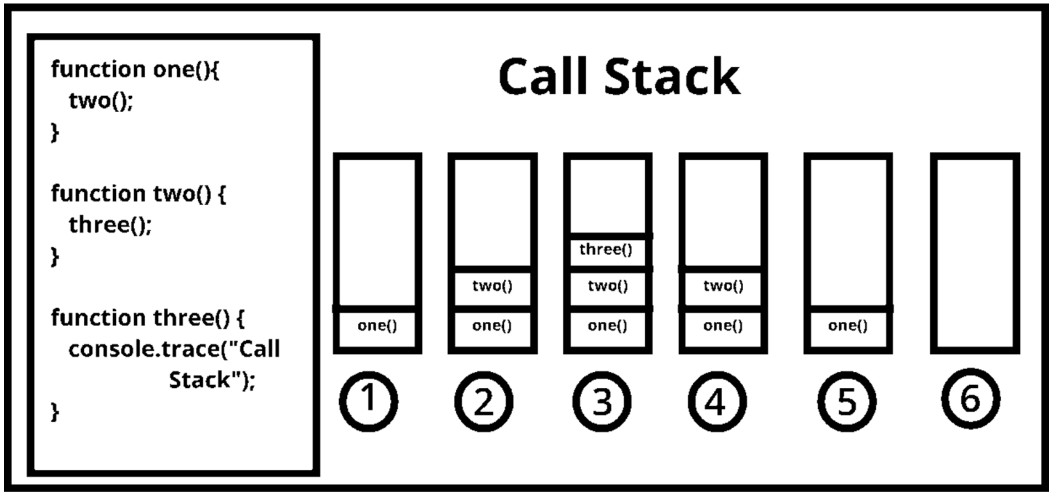
  evaluates to the value on the top: 250

- The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

Function Calls

- In the Computer Architecture class, you will see how stacks are implemented in CPUs to facilitate function calling
- Function calls are similar to problem solving presented earlier:
  - you write a function to solve a problem
  - the function may require sub-problems to be solved, hence, it may call another function
  - once a function is finished, it returns to the function which called it



https://en.wikipedia.org/wiki/Call_stack

Summary: Stacks

- The stack is the simplest of all ADTs
  - Understanding how a stack works may be trivial
  - May be not that simple to understand its applications and meanings

- We looked at:
  - Parsing, function calls, and reverse Polish

References

- Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.
- Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, §11.1, p.200.
- Weiss, *Data Structures and Algorithm Analysis in C++*, 3rd Ed., Addison Wesley, §3.6, p.94.
- Koffman and Wolfgang, "Objects, Abstraction, Data Strucutes and Design using C++", John Wiley & Sons, Inc., Ch. 5.
- Wikipedia, http://en.wikipedia.org/wiki/Stack_(abstract_data_type)