Introduction to Algorithms

Stacks

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

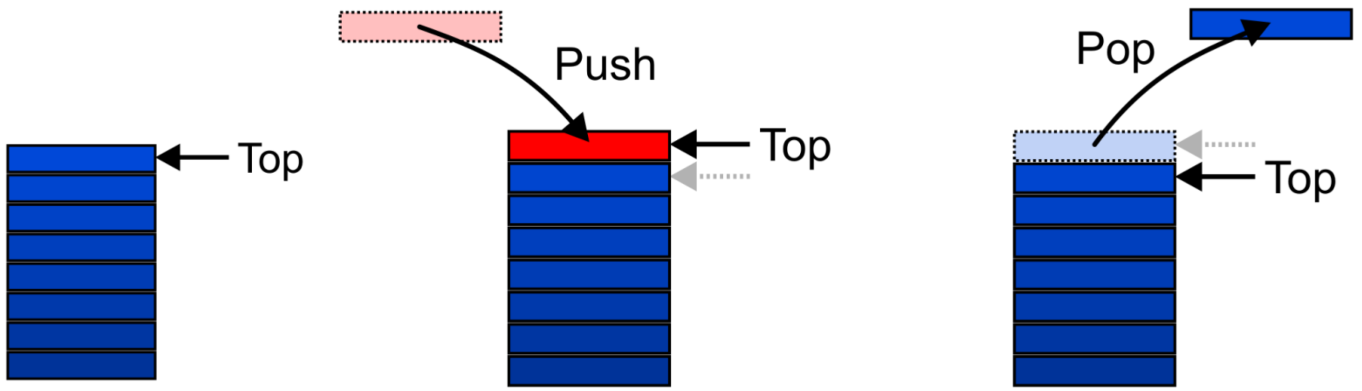
An Abstract Stack (Stack ADT) is an abstract data type which emphasizes specific operations:

- Uses a explicit linear ordering
- 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:



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
- Assembly language

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: Applications

Problem solving:

- Solving one problem may lead to subsequent problems
- These problems may result in further problems
- As problems are solved, your focus shifts back to the problem which lead to the solved problem

Notice that function calls behave similarly:

- A function is a collection of code which solves a problem

Reference: Donald Knuth

Implementations

We will look at two implementations of stacks:

The optimal asymptotic run time of any algorithm is $\Theta(1)$

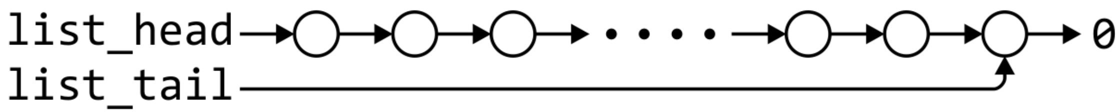
- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

We will look at

- Singly linked lists
- One-ended arrays

Linked-List Implementation

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



	Front/1^{st}	**Back/n^{th}**
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behaviour of an Abstract Stack may be reproduced by performing all operations at the front

`Single_list` Definition

The definition of single list class can be:

```
template <typename Type>
class Single_list {
    public:
       Single_list();
        ~Single_list();

        int size() const;
        bool empty() const;
        Type front() const;
        Type back() const;
        Single_node<Type> *head() const;
        Single_node<Type> *tail() const;
        int count( Type const & ) const;

        void push_front( Type const & );
        void push_back( Type const & );
        Type pop_front();
        int erase( Type const & );
};
```

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

A constructor and destructor is not needed

- Because `list` is declared, the compiler will call the constructor of the `Single_list` class when the `Stack` is constructed

```
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();
}
```

Array Implementation

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

	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Destructor

We need to store an array:

- In C++, this is done by storing the address of the first entry

```
Type *array;
```

We need additional information, including:

- The number of objects currently in the stack

```
int stack_size;
```

- The capacity of the array

```
int array_capacity;
```

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();
};
```

Constructor

The class is only storing the address of the array

- We must allocate memory for the array and initialize the member variables
- The call to `new Type[array_capacity]` makes a request to the operating system for `array_capacity` objects

```
#include <algorithm>
// ...

template <typename Type>
Stack<Type>::Stack( int n ):
stack_size( 0 ),
array_capacity( std::max( 1, n ) ),
array( new Type[array_capacity] ) {
    // Empty constructor
}
```

Constructor

Warning: in C++, the variables are initialized in the order in which they are defined:

```
template <typename Type>
Stack<Type>::Stack( int n ):
stack_size( 0 ),
array_capacity( std::max( 1, n ) ),
array( new Type[array_capacity] ) {
    // Empty constructor
}
```

```
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();
};
```

Destructor

The call to new in the constructor requested memory from the operating system

- The destructor must return that memory to the operating system:

```
template <typename Type>
Stack<Type>::~Stack() {
    delete [] array;
}
```

Empty

The stack is empty if the stack size is zero:

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

The following is unnecessarily tedious:

- The == operator evaluates to either `true` or `false`

```
if ( stack_size == 0 ) {
    return true;
} else {
    return false;
}
```

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

- It is invalid to assign the last entry to “0”
- 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();  // Best solution?????
    }

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

Exceptions

The case where the array is full is not an exception 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

Include a member function `bool full() const;`

Array Capacity

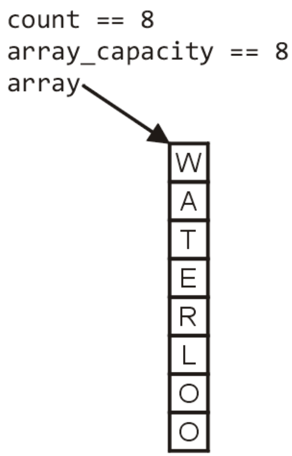
If dynamic memory is available, the best option is to increase the array capacity

If we increase the array capacity, the question is:

- How much?
- By a constant? `array_capacity += c;`
- By a multiple? `array_capacity *= c;`

Array Capacity

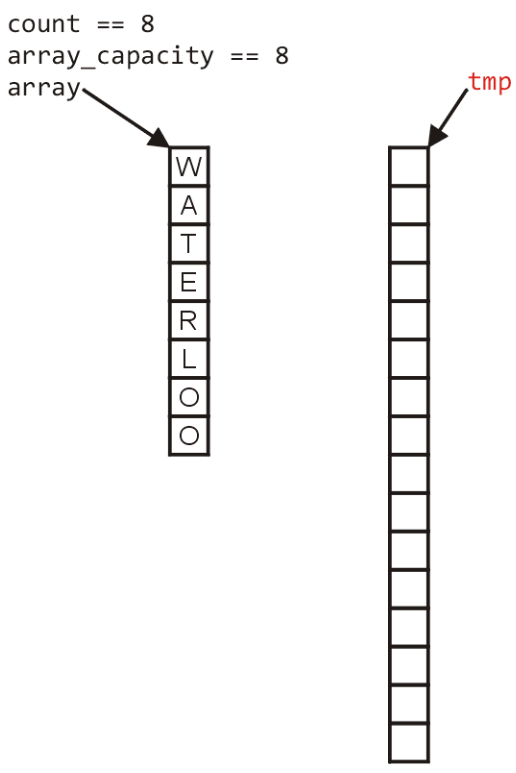
First, let us visualize what must occur to allocate new memory



Array Capacity

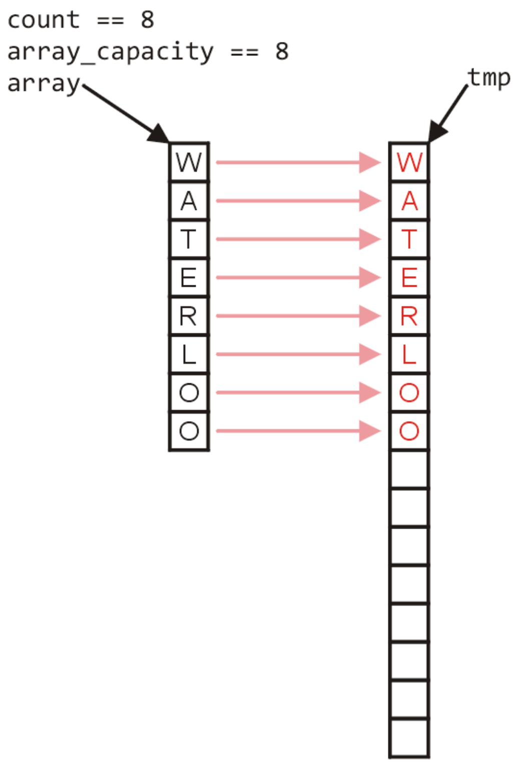
First, this requires a call to `new Type[`*N*`]` where *N* is the new capacity

- We must have access to this so we must store the address returned by new in a local variable, say `tmp`



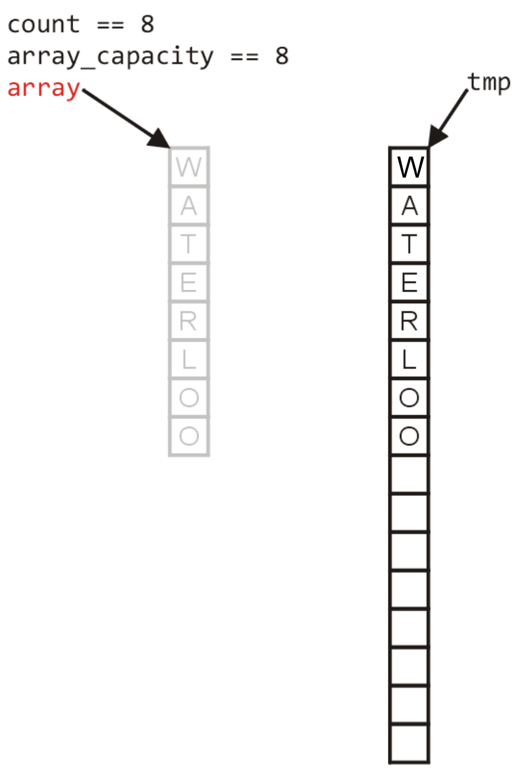
Array Capacity

Next, the values must be copied over



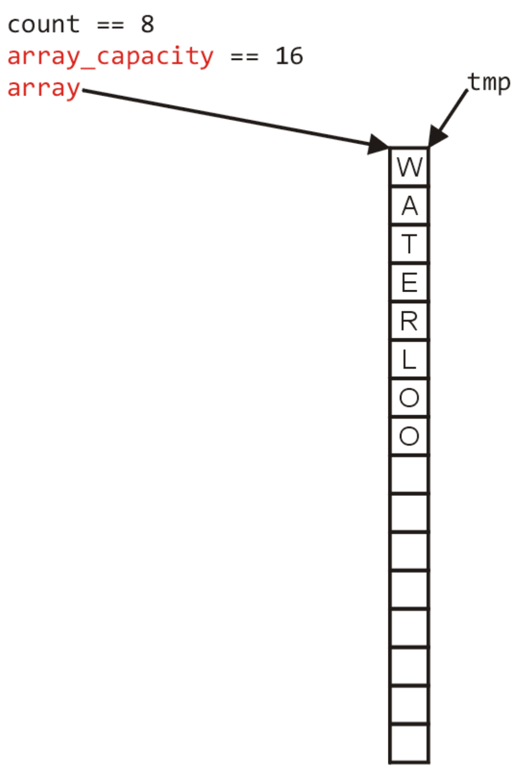
Array Capacity

The memory for the original array must be deallocated



Array Capacity

Finally, the appropriate member variables must be reassigned



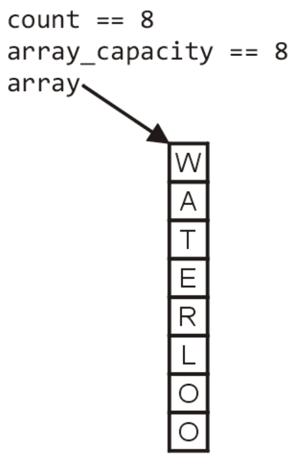
Array Capacity

The implementation:

```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];



}
```



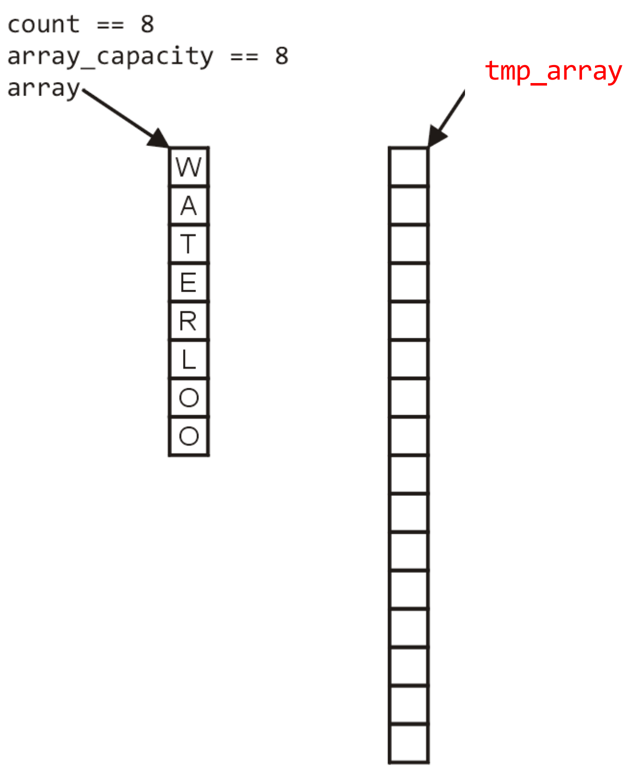
Array Capacity

The implementation:

```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];




}
```



Array Capacity

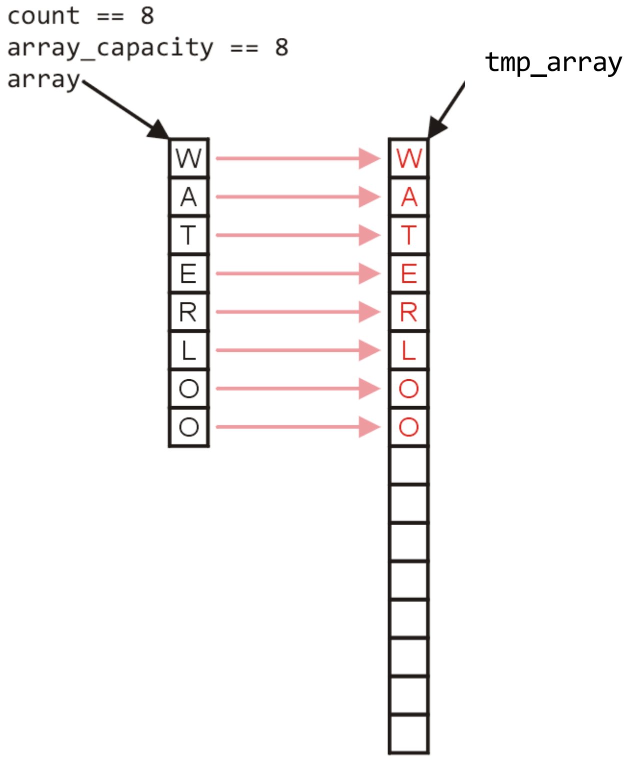
The implementation:

```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }




}
```



Array Capacity

The implementation:

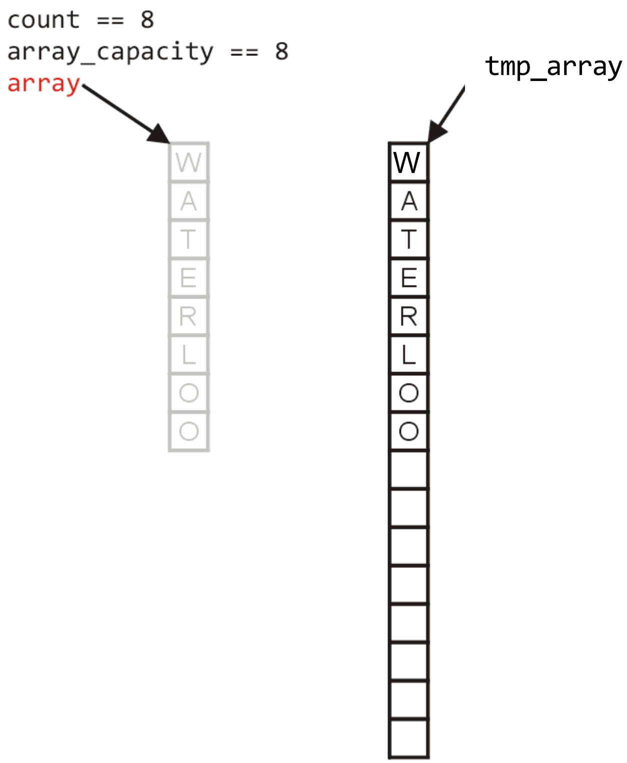
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }

    delete [] array;



}
```



Array Capacity

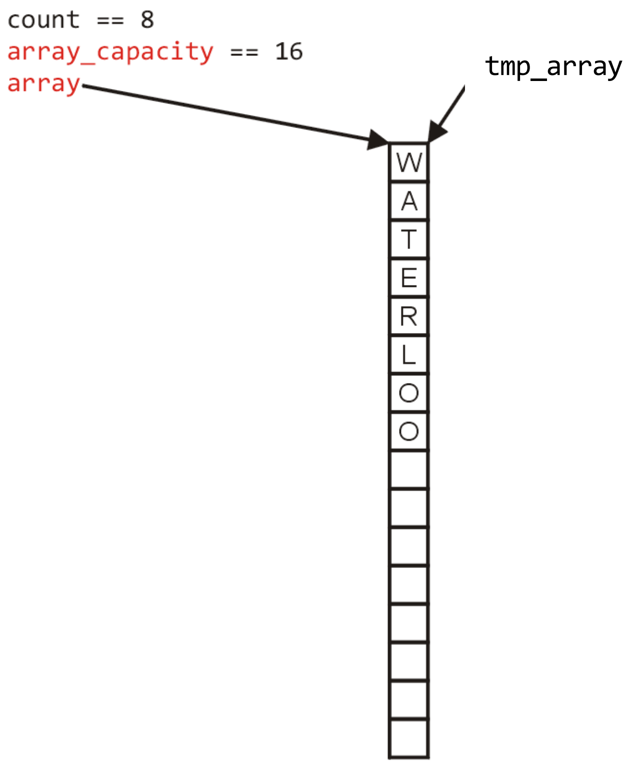
The implementation:

```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }

    delete [] array;
    array = tmp_array;

    array_capacity *= 2;
}
```



Array Capacity

Back to the original question:

- How much do we change the capacity?
- Add a constant?
- Multiply by a constant?

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

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

Array Capacity

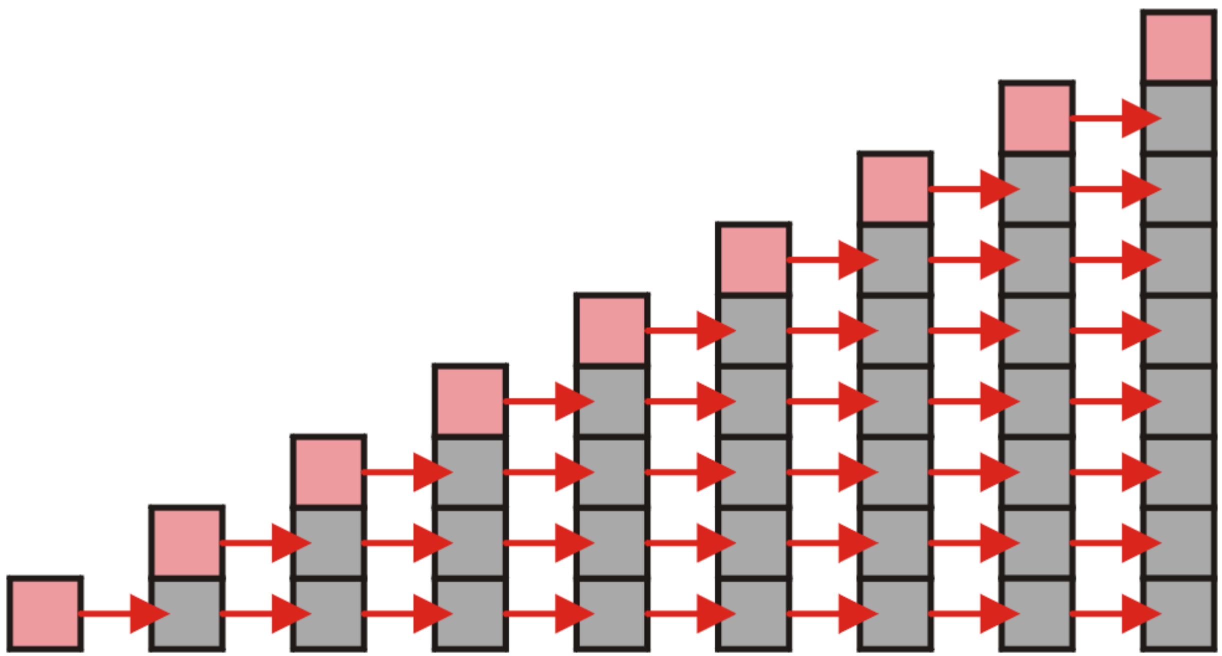
To state the average run time, we will introduce the concept of amortized time:

- If n operations requires $\Theta(\mathrm{f}(n))$, we will say that an individual operation has an amortized run time of $\Theta(\mathrm{f}(n)/n)$
- Therefore, if inserting n objects requires:
  - $\Theta(n^2)$ copies, the amortized time is $\Theta(n)$
  - $\Theta(n)$ copies, the amortized time is $\Theta(1)$

Array Capacity

Let us consider the case of increasing the capacity by 1 each time the array is full

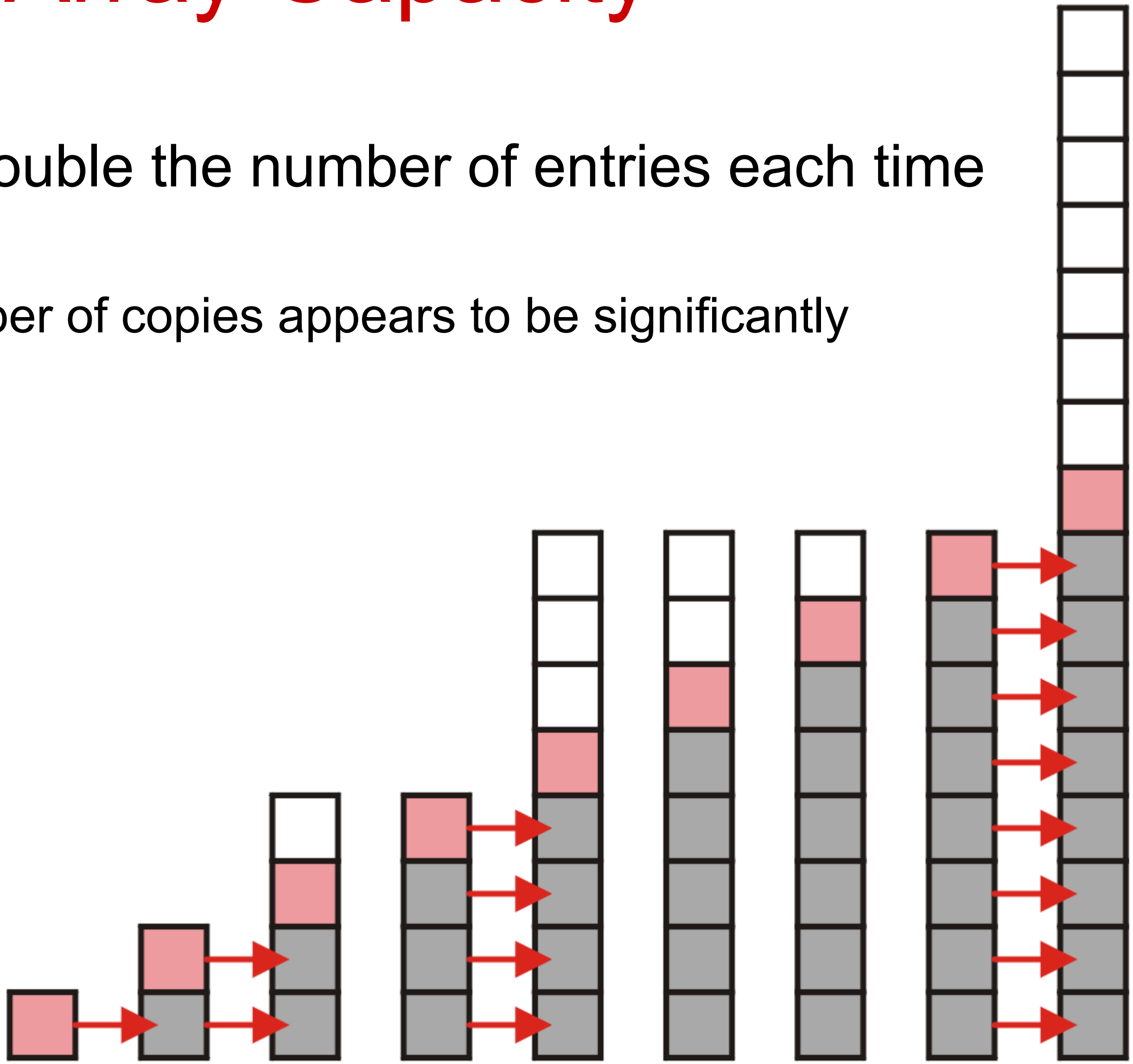
- With each insertion when the array is full, this requires all entries to be copied



Array Capacity

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

Suppose we insert n objects

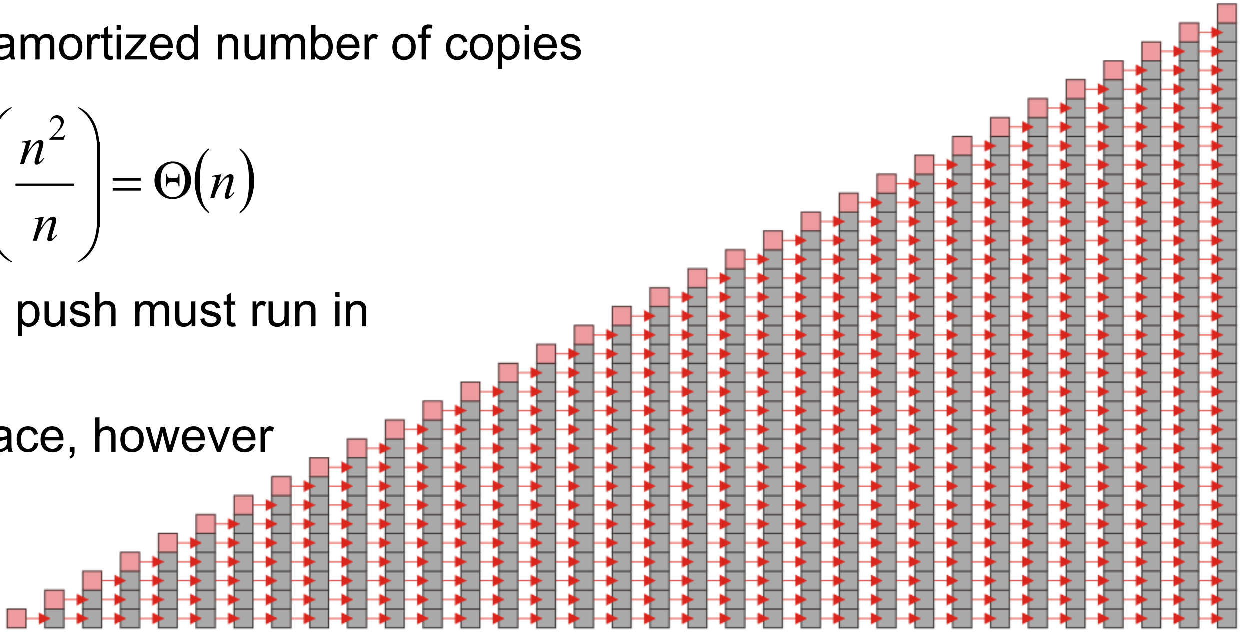
- The pushing of the k^{th} object on the stack requires $k-1$ copies
- The total number of copies is now given by:

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

- Therefore, the amortized number of copies is given by

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

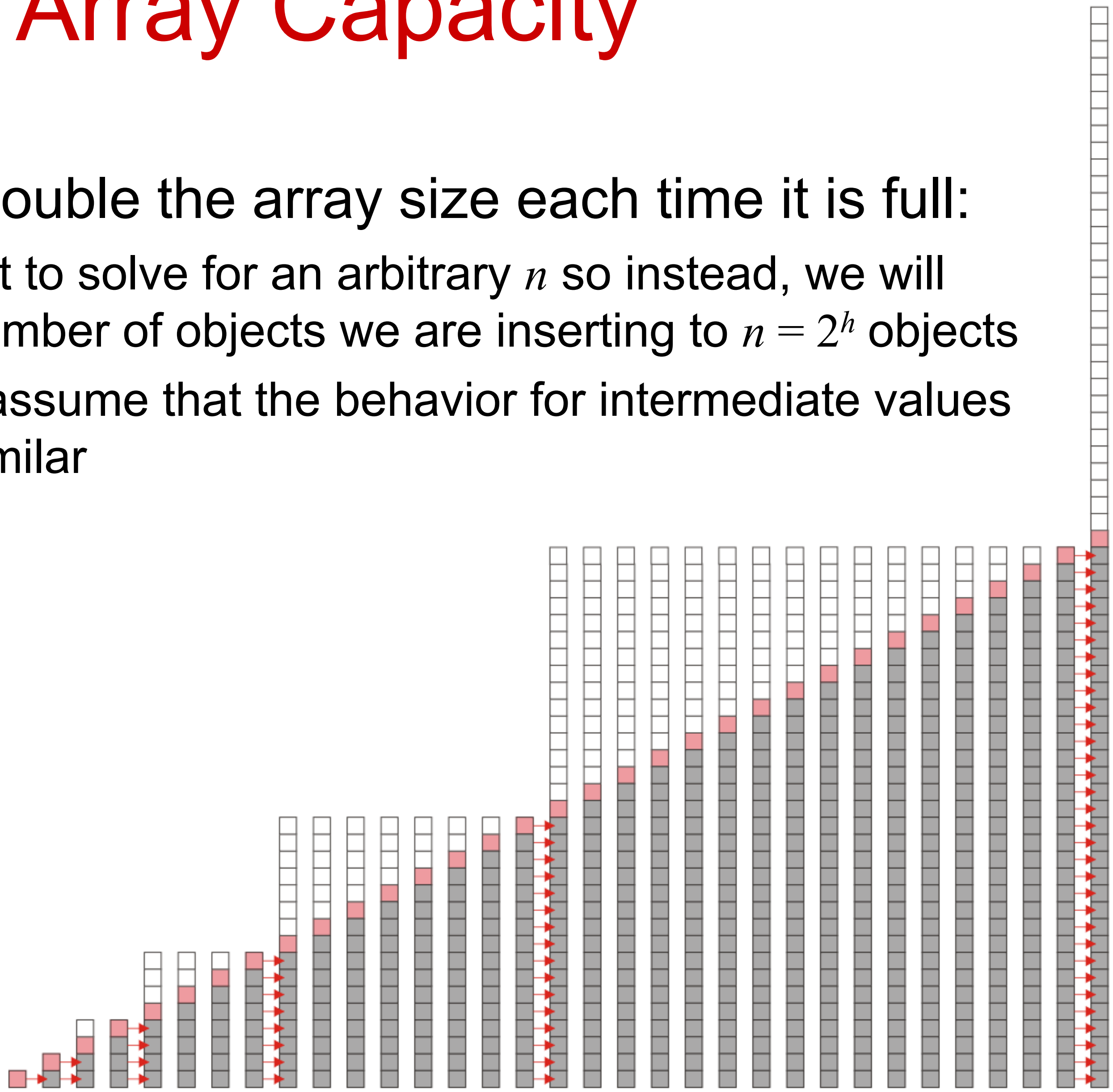
- Therefore each push must run in $\Theta(n)$ time
- The wasted space, however is $\Theta(1)$



Array Capacity

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

Suppose we double the array size each time it is full:

- Inserting $n = 2^h$ objects would therefore require

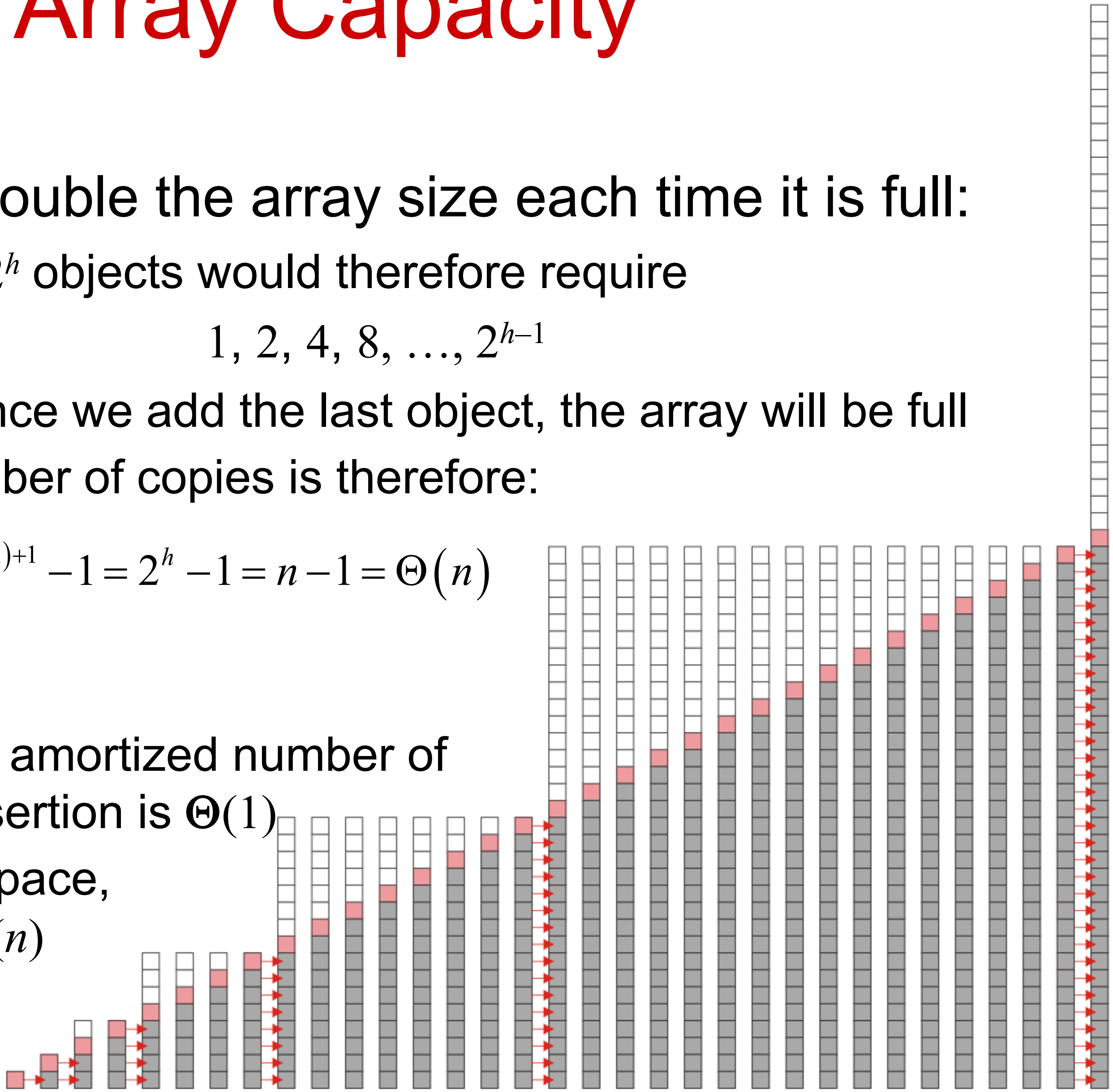
$$1, 2, 4, 8, \ldots, 2^{h-1}$$

  copies, for once we add the last object, the array will be full

- The total number of copies is therefore:

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

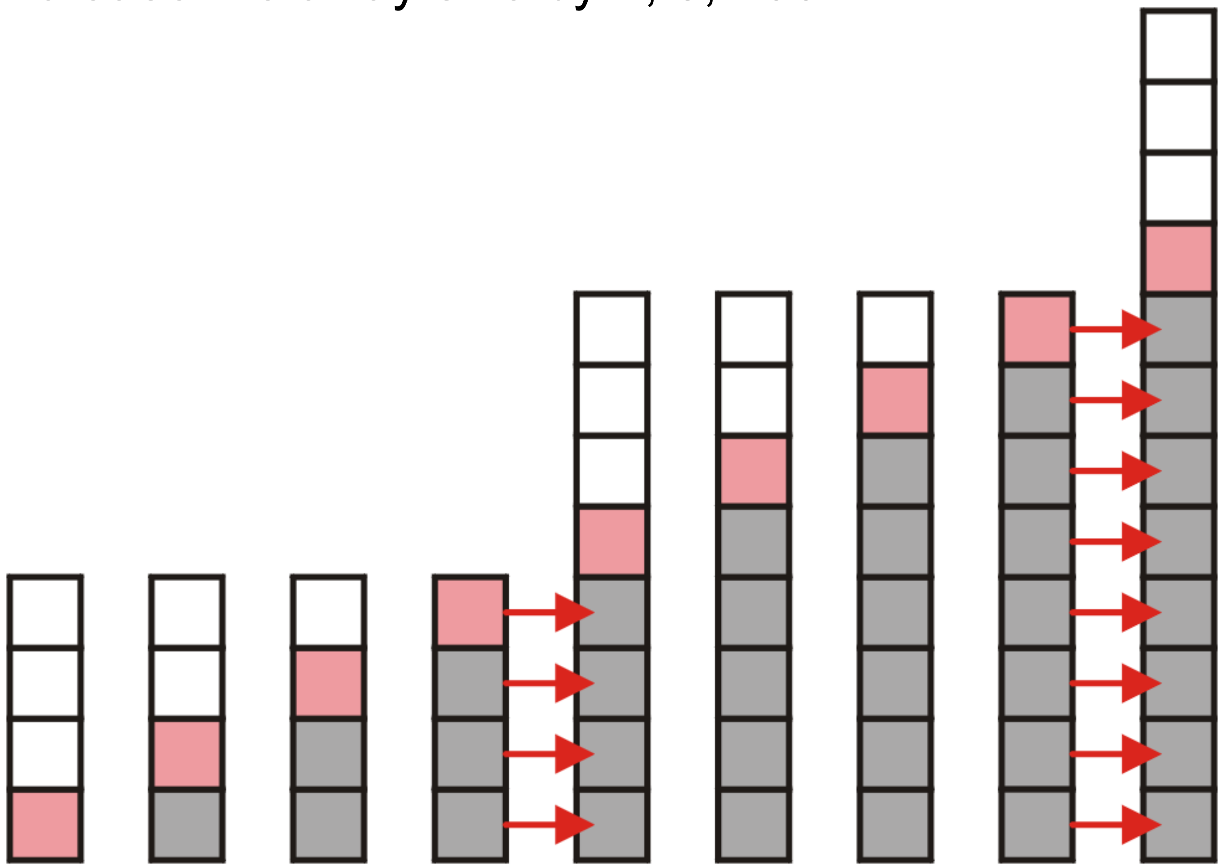
- Therefore the amortized number of copies per insertion is $\Theta(1)$
- The wasted space, however is $\mathbf{O}(n)$



Array Capacity

What if we increase the array size by a larger constant?

- For example, increase the array size by 4, 8, 100?



Array Capacity

Suppose we increase it by a constant value m and we add $n = \ell m$ objects

- To add n items, we will have to make

$$m, 2m, 3m, \ldots, (\ell - 1)m$$

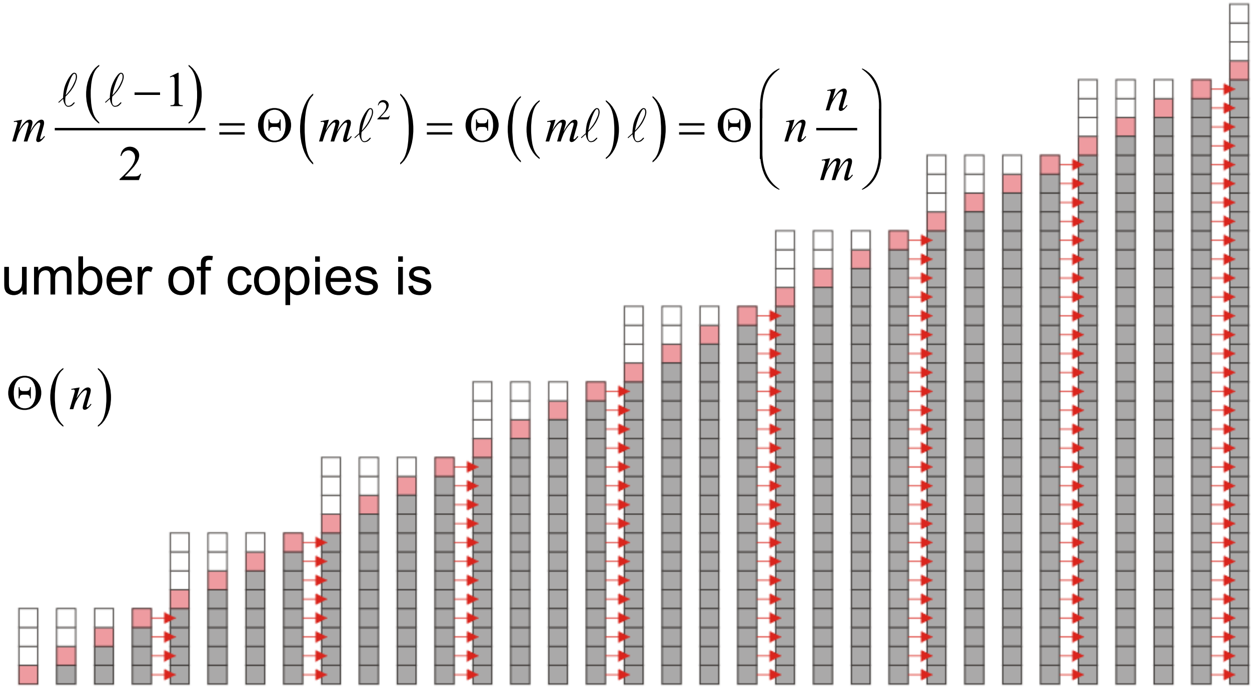
copies in total, or

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

The amoritized number of copies is

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

as m is fixed



Array Capacity

Note the difference in worst-case amortized scenarios:

	Copies per Insertion	**Unused Memory**
Increase by 1	$n - 1$	0
Increase by m	n/m	$m - 1$
Increase by a factor of 2	1	n
Increase by a factor of $r > 1$	$1/(r - 1)$	$(r - 1)n$

Function Calls

This next example discusses function calls

The simple features of a stack indicate why almost all programming languages are based on function calls

Function Calls

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

Function Calls

You will notice that the when a function returns, execution and the return value is passed back to the last function which was called

This is again, the last-in—first-out property

Today’s CPUs have hardware specifically designed to facilitate function calling

Reverse-Polish Notation

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

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

The operator is placed between to 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

Alternatively, we can place the operands first, followed by the operator:

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

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

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

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

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

$$35 \quad 6 \ \ -$$

$$29$$

Reverse-Polish Notation

Other examples:

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

$$3 \quad 20 \quad + \quad 6 \quad -$$

$$23 \quad 6 \quad -$$

$$17$$

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

$$3 \quad 4 \quad -1 \quad \times \quad +$$

$$3 \quad -4 \quad +$$

$$-1$$

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 into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

Reverse-Polish Notation

Reverse-Polish notation is used with some programming languages

- *e.g.*, postscript, pdf, and HP calculators

Similar to the thought process required for writing assembly language code

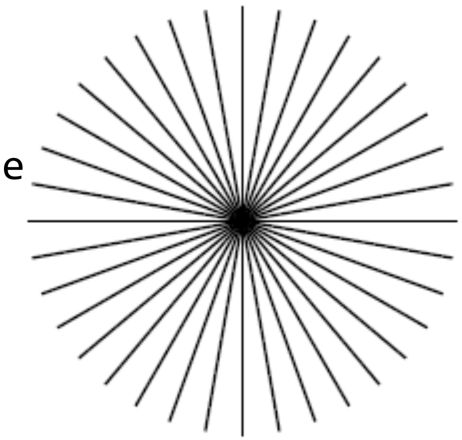
- you cannot perform an operation until you have all of the operands loaded into registers

```
MOVE.L #$2A, D1      ; Load  42 into Register D1
MOVE.L #$100, D2     ; Load 256 into Register D2
ADD D2, D1           ; Add D2 into D1
```

Reverse-Polish Notation

A quick example of postscript:

```
0 10 360 {                   % Go from 0 to 360 degrees in 10-degree steps
    newpath                  % Start a new path
    gsave                    % Keep rotations temporary
        144 144 moveto
        rotate               % Rotate by degrees on stack from 'for'
        72 0 rlineto
        stroke
    grestore                 % Get back the unrotated state
} for % Iterate over angles
```



http://www.tailrecursive.org/postscript/examples/rotate.html

Reverse-Polish Notation

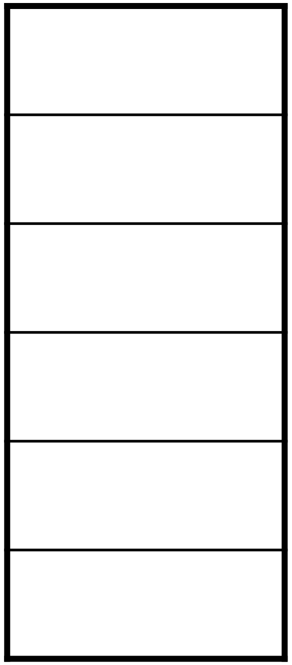
The easiest way to parse reverse-Polish notation is to use an operand stack:

- 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:

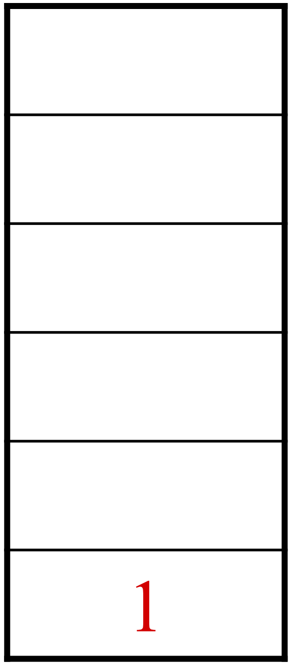
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 1 onto the stack

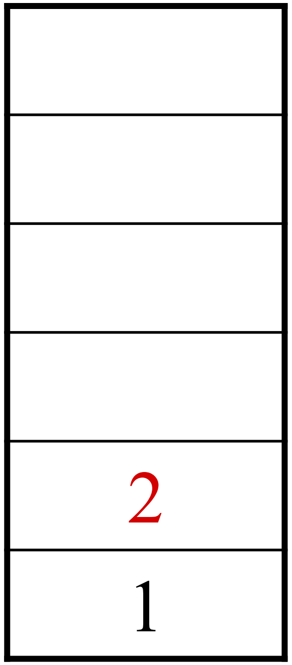
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

Push 1 onto the stack

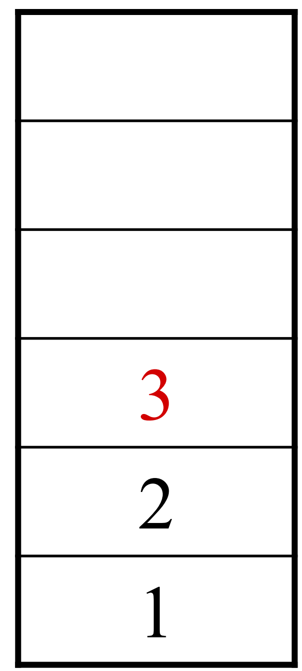
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 3 onto the stack

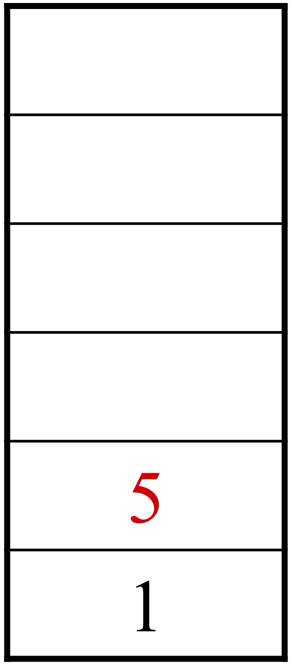
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Reverse-Polish Notation

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

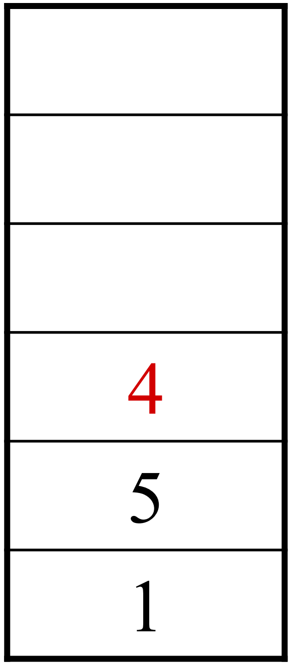
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 4 onto the stack

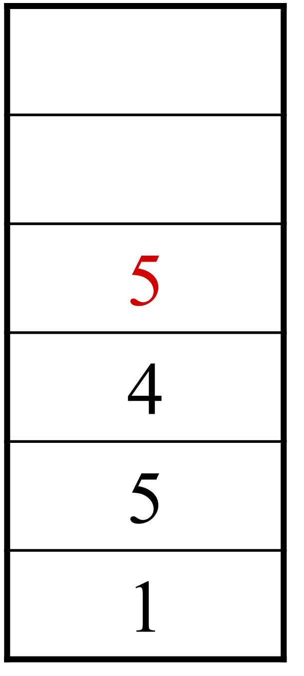
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

Push 5 onto the stack

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



Reverse-Polish Notation

Push 6 onto the stack

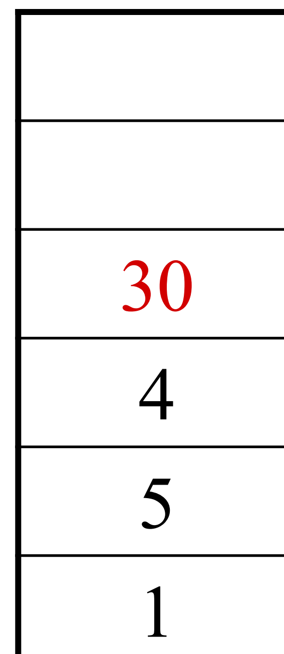
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; {\color{red}6} \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

6
5
4
5
1

Reverse-Polish Notation

Pop 6 and 5 and push $5 \times 6 = 30$

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



Reverse-Polish Notation

Pop 30 and 4 and push $4 - 30 = -26$

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

-26
5
1

Reverse-Polish Notation

Push 7 onto the stack

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

7
–26
5
1

Reverse-Polish Notation

Pop 7 and -26 and push $-26 \times 7 = -182$

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

-182
5
1

Reverse-Polish Notation

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

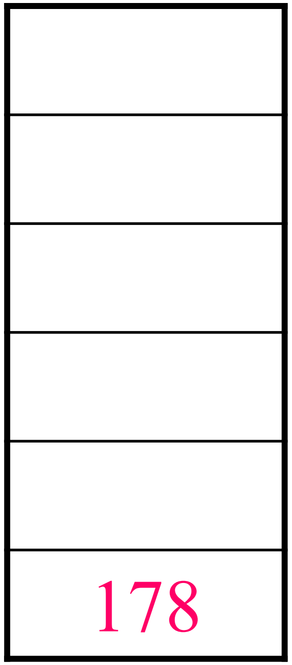
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

-177
1

Reverse-Polish Notation

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

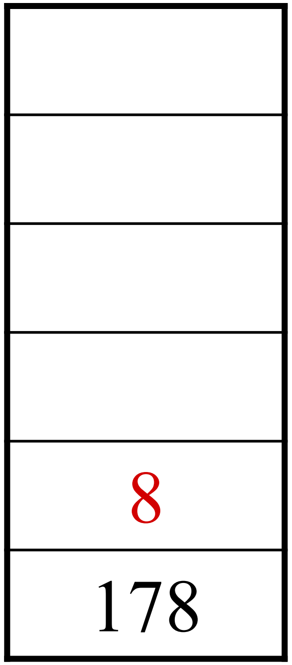
$$1\ \ 2\ \ 3\ \ +\ \ 4\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$



Reverse-Polish Notation

Push 8 onto the stack

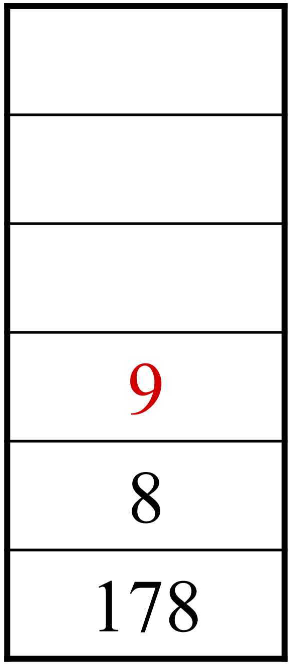
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad {\color{red}8} \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 1 onto the 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

Pop 9 and 8 and push $8 \times 9 = 72$

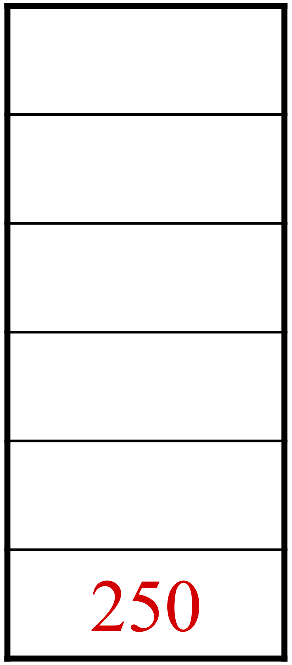
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

72
178

Reverse-Polish Notation

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

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



Reverse-Polish Notation

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

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

We reduce the parentheses using order-of-operations:

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

Reverse-Polish Notation

Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

which has the reverse-Polish notation of

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

For comparison, the calculated expression was

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

Standard Template Library

The Standard Template Library (STL) has a *wrapper* class stack with the following declaration:

```
template <typename T>
class stack {
    public:
        stack();                    // not quite true...
        bool empty() const;
        int size() const;
        const T & top() const;
        void push( const T & );
        void pop();
};
```

Standard Template Library

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> istack;

    istack.push( 13 );
    istack.push( 42 );
    cout << "Top: " << istack.top() << endl;
    istack.pop();                  // no return value
    cout << "Top: " << istack.top() << endl;
    cout << "Size: " << istack.size() << endl;

    return 0;
}
```

Standard Template Library

The reason that the `stack` class is termed a wrapper is because it uses a different container class to actually store the elements

The `stack` class simply presents the *stack interface* with appropriately named member functions:

- `push`, `pop` , and `top`

Stacks

The stack is the simplest of all ADTs

- Understanding how a stack works is trivial

The application of a stack, however, is not in the implementation, but rather:

- Where possible, create a design which allows the use of a stack

We looked at:

- 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)