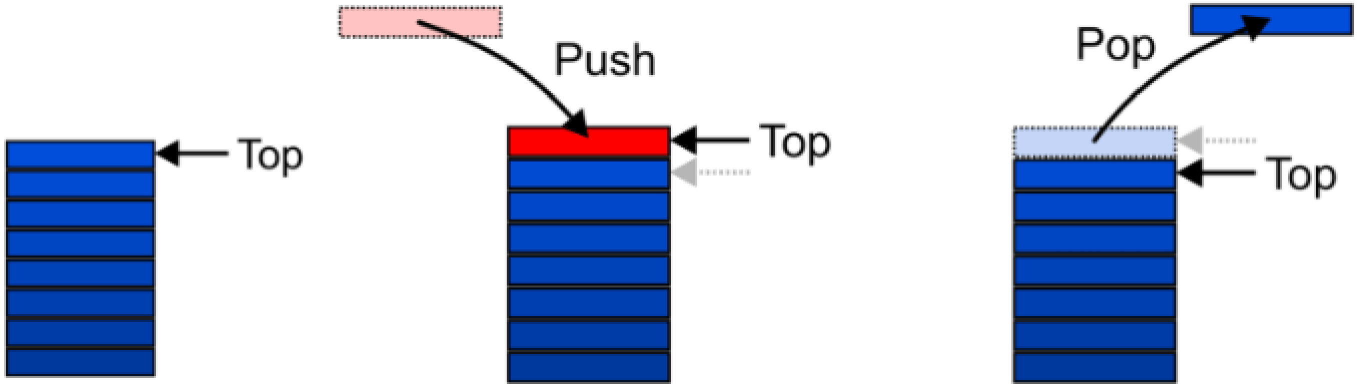
Stacks

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

- Also called a last-in-first-out (LIFO) behavior
- 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

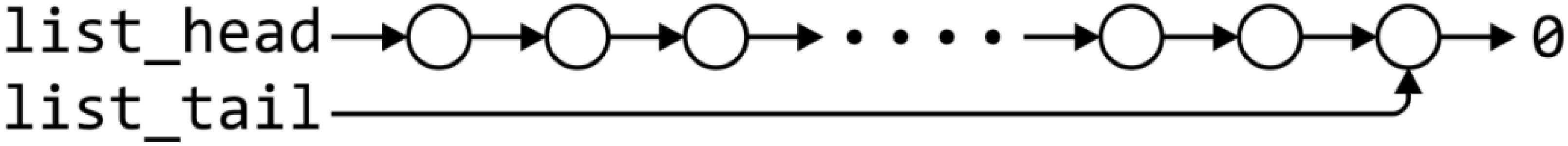
- Problem solving
    - Solving one problem may lead to subsequent problems
    - These problems may result in further problems
    - As problems are solved, your focus shifts back to the problem which lead to the solved problem
- Notice that function calls behave similarly
    - A function is a collection 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 $\Theta(1)$



- With asymptotic analysis of linked lists, we can now make the following statements:

	front / 1st node	back / nth node
find	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(1)$	$\Theta(1)$
erase	$\Theta(1)$	$\Theta(n)$

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

`LinkedList` Definition

`LinkedList` Definition

In [2]:

```
template <typename Type>
class LinkedList {
private:
    SinglyLinkedNode<Type> *list_head;

public:
    LinkedList();
    ~LinkedList();

    // Accessors
    bool empty() const;
    Type front() const;
    SinglyLinkedNode<Type>* begin() const;

    int size() const;
    int count( const Type & ) const;
    SinglyLinkedNode<Type>* find( const Type & ) const;

    // Mutators
    void push_front( const Type & );
    Type pop_front();
};
```

Stack-as-List Class

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

Stack-as-List Class

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

In [3]:

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

Stack-as-List Class

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

In [3]:

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

- A constructor and destructor is not needed
    - Because **list** is declared, the compiler will call the constructor of the `LinkedList` class when the `Stack` is constructed

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

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

In [4]:

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

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

In [4]:

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

In [5]:

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

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

In [4]:

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

In [5]:

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

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

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

In [4]:

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

In [5]:

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

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

In [6]:

```
template <typename Type>
Type Stack<Type>::top() const {
    if ( empty() ) {
        throw std::underflow_error("Stack is empty");
    }
    return list.front();
}
```

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

In [4]:

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

In [5]:

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

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

In [6]:

```
template <typename Type>
Type Stack<Type>::top() const {
    if ( empty() ) {
        throw std::underflow_error("Stack is empty");
    }
    return list.front();
}
```

In [7]:

```
template <typename Type>
Type Stack<Type>::pop() {
    if ( empty() ) {
        throw std::underflow_error("Stack is empty");
    }
    return list.pop_front();
}
```

Array Implementation

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

- With asymptotic analysis of array lists, we can now make the following statements:

	front / 1st node	**back / nth node**
find	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(n)$	$\Theta(1)$
erase	$\Theta(n)$	$\Theta(1)$

- The desired behavior of an Abstract Stack may be reproduced by performing all operations at the back

Design

- 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

Stack-as-Array Class

In [8]:

```
template <typename Type>
class ArrayStack {
    private:
        int stack_size;
        int array_capacity;
        Type *array;
    public:
        ArrayStack( int = 10 );
        ~ArrayStack();
        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

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

In [9]:

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

Destructor

- The call to `new` in the constructor requested memory from the operating system
    - The destructor must return that memory to the operating system

Destructor

- The call to `new` in the constructor requested memory from the operating system
    - The destructor must return that memory to the operating system

In [10]:

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

`empty`

- The stack is empty if the stack size is zero

`empty`

- The stack is empty if the stack size is zero

In [11]:

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

`top`

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

`top`

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

In [12]:

```
template <typename Type>
Type ArrayStack<Type>::top() const {
    if ( empty() ) {
        throw std::underflow_error("Stack is empty");
    }
    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`

`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`

In [13]:

```
template <typename Type>
Type ArrayStack<Type>::pop() {
    if ( empty() ) {
        throw std::underflow_error("Stack is empty");
    }
    --stack_size;
    return array[stack_size];
}
```

`push`

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

`push`

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

In [14]:

```
template <typename Type>
void ArrayStack<Type>::push( Type const &obj ) {
    if ( stack_size == array_capacity ) {
        throw overflow_error("Stack is empty");  // 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;`

1. 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`
2. Next, the values must be copied over
3. The memory for the original array must be deallocated
4. Finally, the appropriate member variables must be reassigned

1. 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`
2. Next, the values must be copied over
3. The memory for the original array must be deallocated
4. Finally, the appropriate member variables must be reassigned

In [ ]:

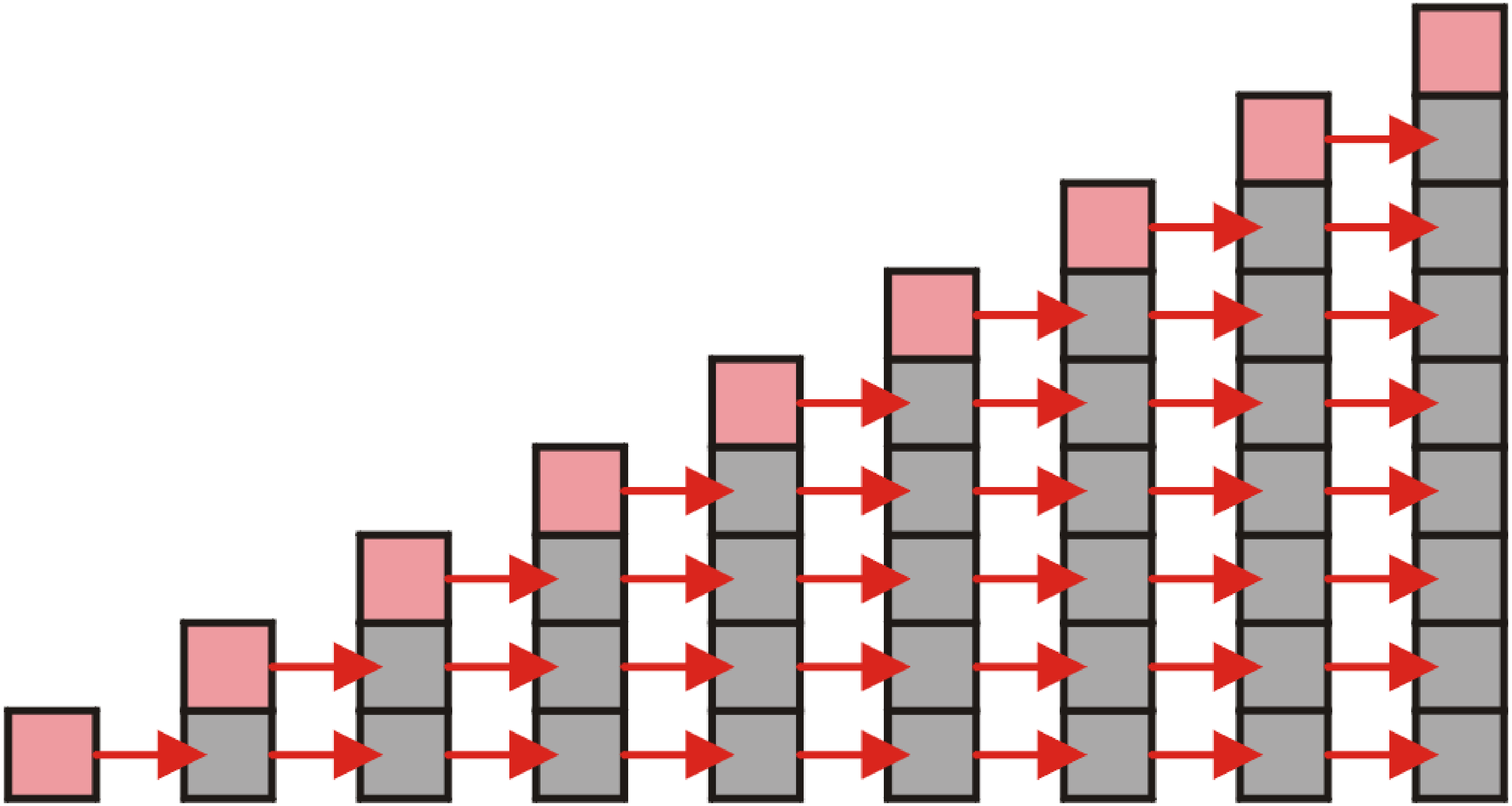
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity]; //| Step 1
                                                  //|--------
    for ( int i = 0; i < array_capacity; ++i ) {  //|
        tmp_array[i] = array[i];                  //| Step 2
    }                                             //|--------
                                                  //|
    delete [] array;                              //| Step 3
                                                  //|--------
    array = tmp_array;                            //|
    array_capacity *= 2;                          //| Step 4
}
```

- 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

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

- 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

- 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



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

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

- Therefore, the amortized number of copies is given by

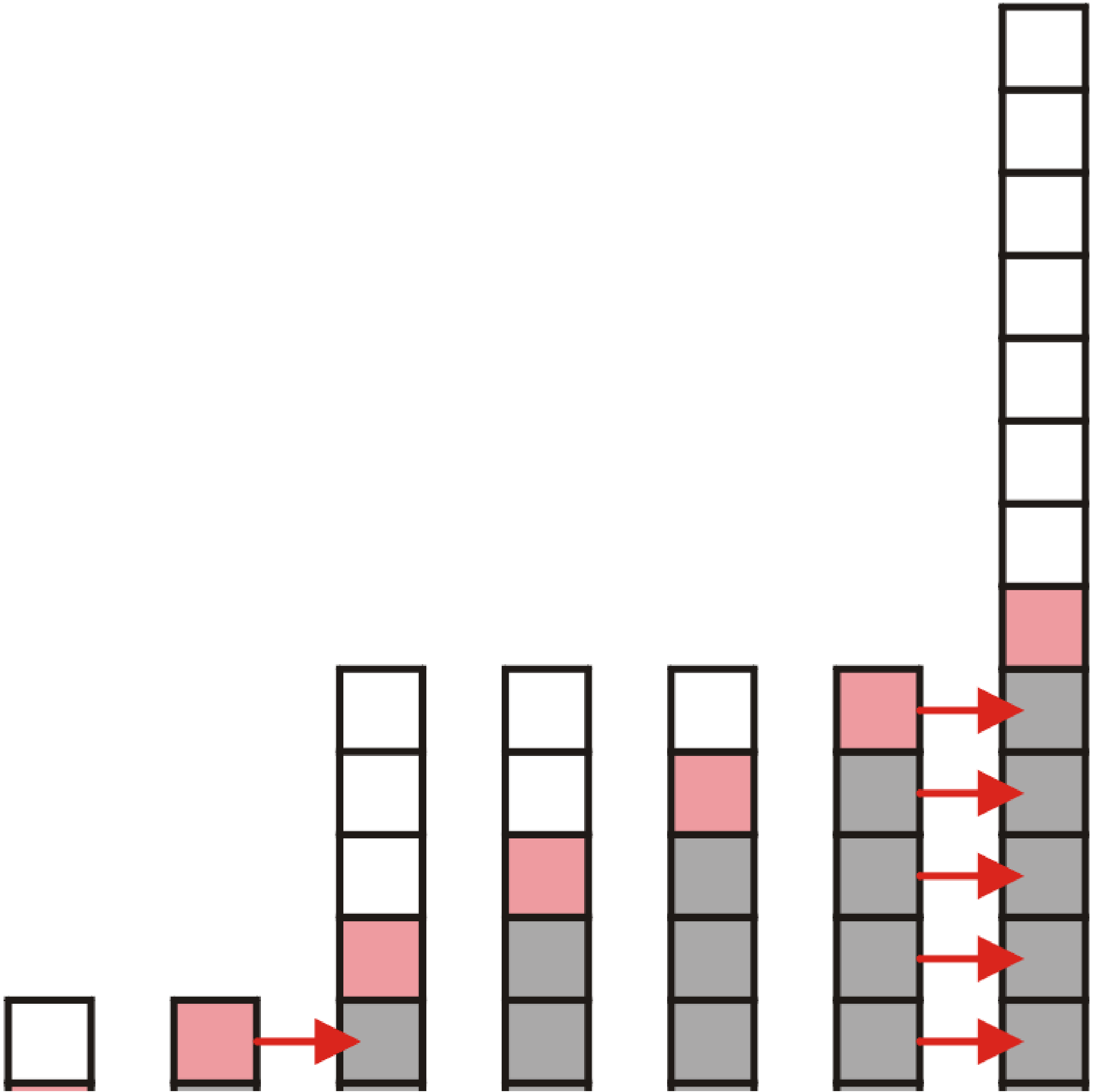
$$\Theta\left(\frac{n^2}{n}\right) = \Theta(n)$$

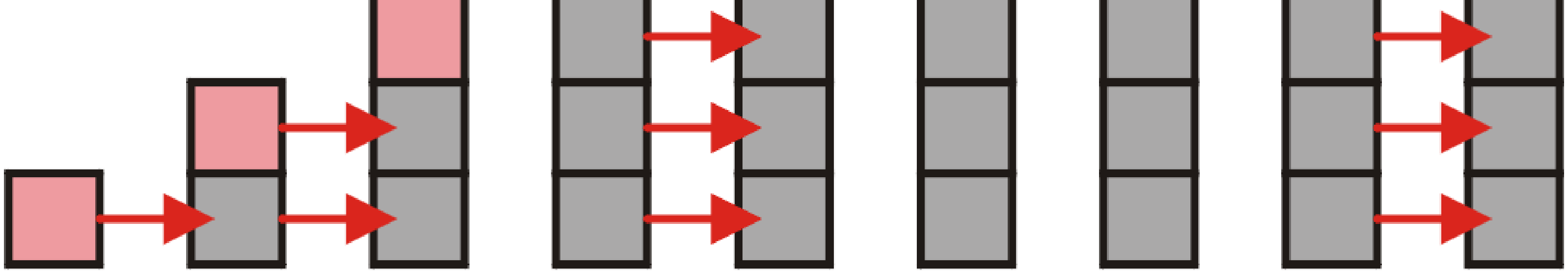
- Therefore each push must run in $\Theta(n)$ time

- The wasted space, however is $\Theta(1)$

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

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





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

- 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

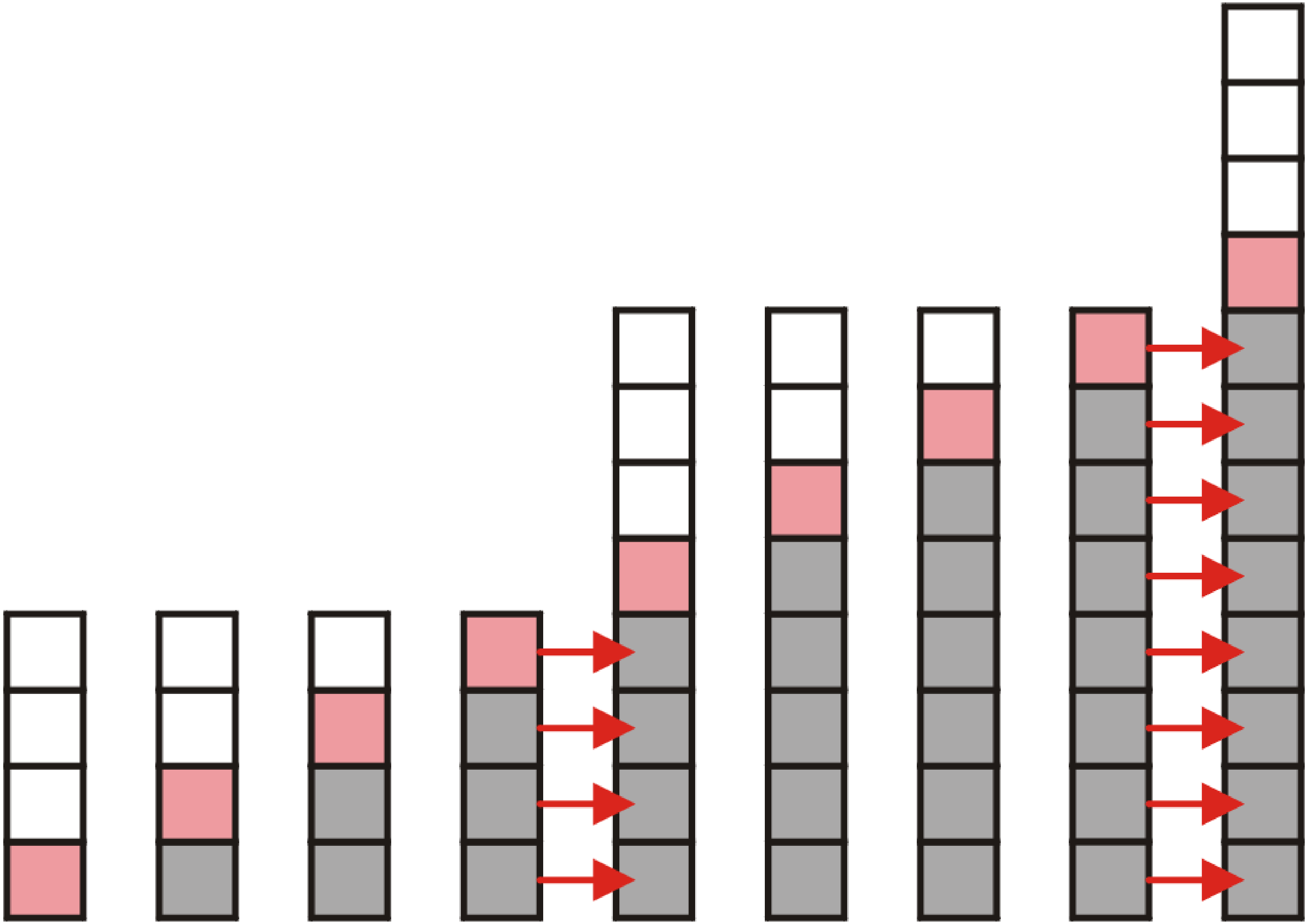
- The total number of copies is therefore

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

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

- What if we increase the array size by a larger constant?
    - For example, increase the array size by 4, 8, 100?

- What if we increase the array size by a larger constant?
  - For example, increase the array size by 4, 8, 100?



- Suppose we increase it by a constant value m and we add n = ℓm objects
  - To add n items, we will have to make

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

copies in total, or

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

- The amortized number of copies is

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

as m is fixed

- Note the difference in worst-case amortized scenarios

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

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

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

$$3+4\times 5-6=17$$

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

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

- 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:

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

$$7\ 5\times 6-$$

$$35\ 6-$$

$$29$$

- 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 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
```

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

Example

Example

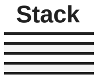
- Evaluate the following reverse-Polish expression using a stack:

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

Example

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

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

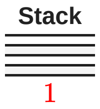


- Push 1 onto the stack

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

- Push 1 onto the stack

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



Stack

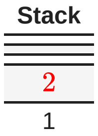
1

- Push 2 onto the stack

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

- Push 2 onto the stack

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



Stack

2

1

- Push 3 onto the stack

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

- Push 3 onto the stack

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

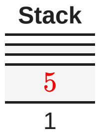
Stack
3
2
1

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

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

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

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



- Push 4 onto the stack

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

- Push 4 onto the stack

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

Stack
4
5
1

- Push 5 onto the stack

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

- Push 5 onto the stack

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

Stack
5
4
5
1

- Push 6 onto the stack

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

- Push 6 onto the stack

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

Stack
6
5
4
5
1

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

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

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

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

Stack
30
4
5
1

- Pop 30 and 4 and push $4 - 30 = -\mathbf{26}$

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

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

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

Stack
-26
5
1

- Push 7 onto the stack

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

- Push 7 onto the stack

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

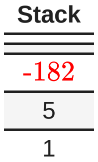
Stack
7
-26
5
1

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

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

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

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



Stack

-182

5

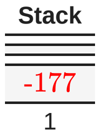
1

- Pop -182 and 5 and push $-182 \textcolor{red}{+} 5 = -177$

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

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

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



Stack

-177

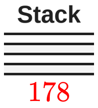
1

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

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

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

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



Stack

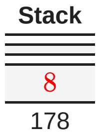
178

- Push 8 onto the stack

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

- Push 8 onto the stack

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



Stack

8

178

- Push 9 onto the stack

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

- Push 9 onto the stack

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

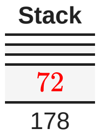
Stack
9
8
178

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

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

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

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



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

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

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

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

Stack

250

- 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:

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

- Incidentally,

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

  which has the reverse-Polish notation of
- The equivalent in-fix notation is

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

- 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

Standard Template Library

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

In [ ]:

```
template <typename T>
class stack {
public:
    stack();
    bool empty() const;
    int size() const;
    const T& top() const;
    void push( const T& );
    void pop();
};
```

In [16]:

```
#include <stack>
{
    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;
}
```

```
Top: 42
Top: 13
Size: 1
```

In [16]:

```
#include <stack>
{
    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;
}
```

```
Top: 42
Top: 13
Size: 1
```

- 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