

# CS101 Algorithms and Data Structures

Stack

Textbook Ch 10.1



# Outline

- Stack ADT
- Implementation
- Example applications

# Reverse-Polish Notation

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

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

The operator is placed 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

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:

$$\begin{array}{ccccccc} 3 & 4 & + & 5 & \times & 6 & - \\ & 7 & & 5 & \times & 6 & - \\ & & 35 & & 6 & - \\ & & & & 29 & & \end{array}$$

# Reverse-Polish Notation

Other examples:

3 4 5 × + 6 −

3 20 + 6 −

23 6 −

17

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

3 4 5 6 − × +

3 4 −1 × +

3 −4 +

−1

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

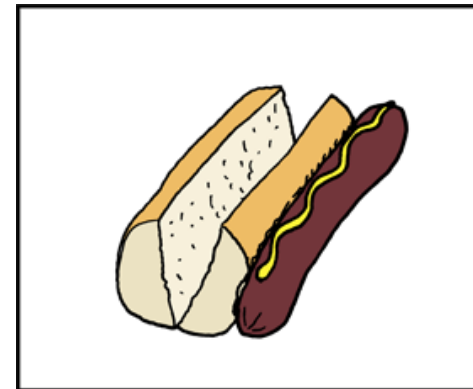
# Reverse-Polish Notation

This is called *reverse-Polish* notation after the mathematician Jan Łukasiewicz



Narodowe Archiwum Cyfrowe, sygn. 1-II-358

<http://www.audiovis.nac.gov.pl/>



REVERSE POLISH SAUSAGE

<http://xkcd.com/645/>

# 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 Notation

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

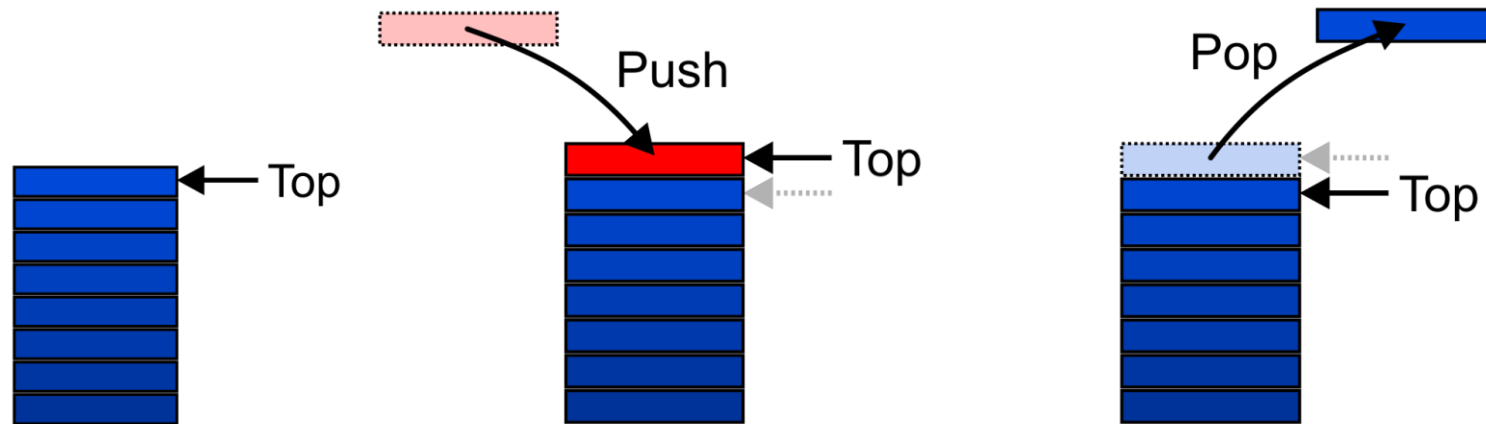
- operands are processed by pushing them onto the stack
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack



# Stack ADT

Also called a *last-in–first-out* (LIFO) behaviour

- Graphically, we may view these operations as follows:



# Applications

Numerous applications:

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

# Reverse-Polish Notation

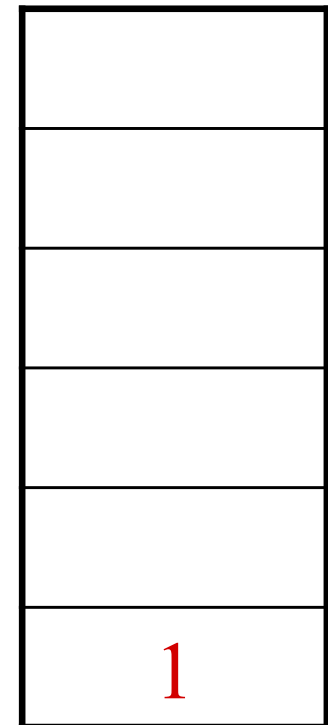
Evaluate the following reverse-Polish expression using a stack:

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 1 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

2
1

# Reverse-Polish Notation

Push 3 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

3
2
1

# Reverse-Polish Notation

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

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

5
1

# Reverse-Polish Notation

Push 4 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

4
5
1



# Reverse-Polish Notation

Push 5 onto the stack

1 2 3 + 4 **5** 6 × − 7 × + − 8 9 × +

<b>5</b>
4
5
1

# Reverse-Polish Notation

Push 6 onto the stack

1 2 3 + 4 5 **6** × − 7 × + − 8 9 × +

<b>6</b>
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  $\times$  +

30
4
5
1

# Reverse-Polish Notation

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

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

−26
5
1

# Reverse-Polish Notation

Push 7 onto the stack

1 2 3 + 4 5 6 × − *7* × + − 8 9 × +

<i>7</i>
−26
5
1

# Reverse-Polish Notation

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

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 × +

$-177$
1

# Reverse-Polish Notation

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

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

178



# Reverse-Polish Notation

Push 8 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

8
178

# Reverse-Polish Notation

Push 1 onto the stack

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

9
8
178

# Reverse-Polish Notation

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

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

72
178

# Reverse-Polish Notation

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

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

250

# Reverse-Polish Notation

Thus

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

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:

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

# Stack ADT

- Uses an explicit linear ordering
- Two principal operations
  - *Push*: insert an object onto the top of the stack
  - *Pop*: erase the object on the top of the stack
  - *CreateStack*: generate an empty stack
  - *IsEmpty*: determine if stack is empty
  - *IsFull*: determine if stack is full

# Outline

- Stack ADT
- **Implementation**
- Example applications

# Implementations

We will look at two implementations of stacks:

- Singly linked lists
- One-ended arrays

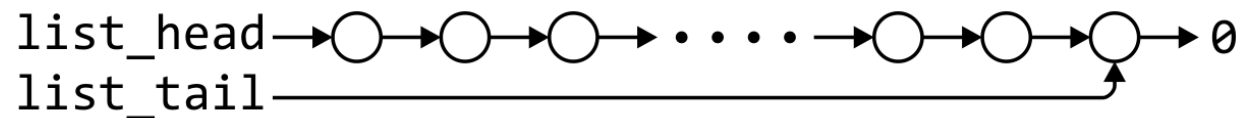
The optimal asymptotic run time of any algorithm is  $\Theta(1)$

- The run time of the algorithm is independent of the number of objects being stored in the container



# Linked-List Implementation

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



	Front/ $1^{\text{st}}$	Back/ $n^{\text{th}}$
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

# Single\_list Definition

The definition of single list class:

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

        int size() const;           /* return the length of the List */
        bool empty() const;         /* return true when List is empty */
        Type front() const;         /* return the data in the first node */
        Type back() const;          /* return the data in the last node */
        Single_node<Type> *head() const; /* return the first node */
        Single_node<Type> *tail() const; /* return the last node */
        int count( Type const & ) const; /* counts the number of instances of data*/

        void push_front( Type const & ); /* insert a node as the first node*/
        void push_back( Type const & ); /* insert a node as the last node*/
        Type pop_front(); /* return the data in the first node and delete the first node*/
        int erase( Type const & ); /* removes the nodes containing that integer*/

};
```

# 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 );
}
```

```
void push_front( int )
```

We could, however, note that when the list is empty, `list_head == 0`, thus we could shorten this to:

```
void List::push_front( int n ) {  
    list_head = new Node( n, list_head );  
}
```

If it is empty, we start with:

`list_head`  $\longrightarrow$  0

and, if we try to add 81, we should end up with:

`list_head`  $\longrightarrow$  (81)  $\longrightarrow$  0

```
void push_front( int )
```

We could, however, note that when the list is empty, `list_head == 0`, thus we could shorten this to:

```
void List::push_front( int n ) {  
    list_head = new Node( n, list_head );  
}
```

If it is not empty, we start with:



and, if we try to add 70, we should end up with:



# 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();
}
```

# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```

```
int front() const  
  
int List::front() const {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    return head()->retrieve();  
}
```



# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();    e = 70  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```



**int front() const**

```
int List::front() const {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    return head()->retrieve();  
}
```

# int pop\_front()

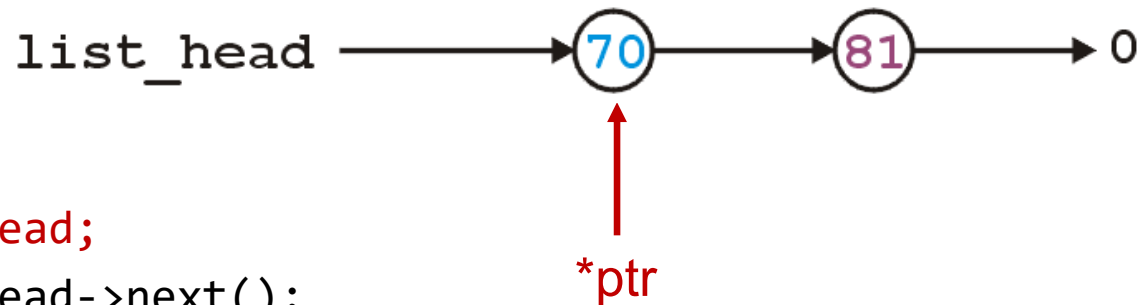
The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```

# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

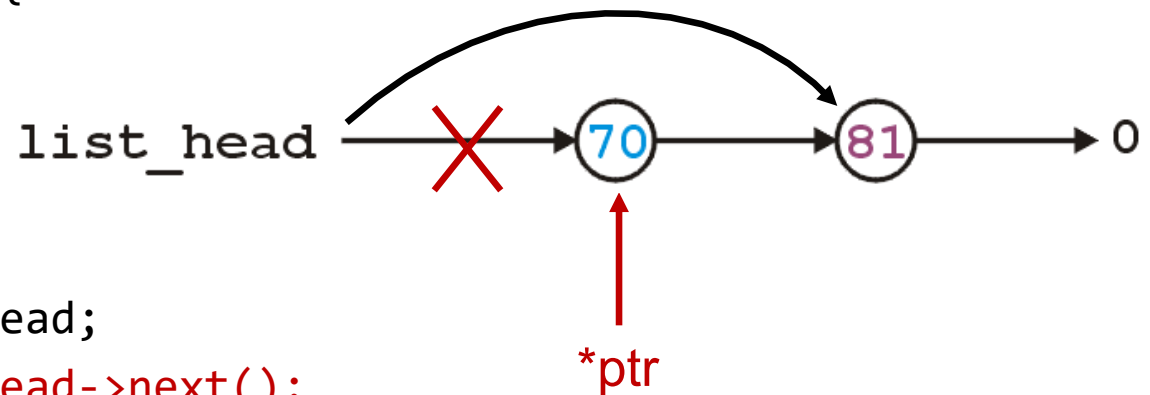
```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```



# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

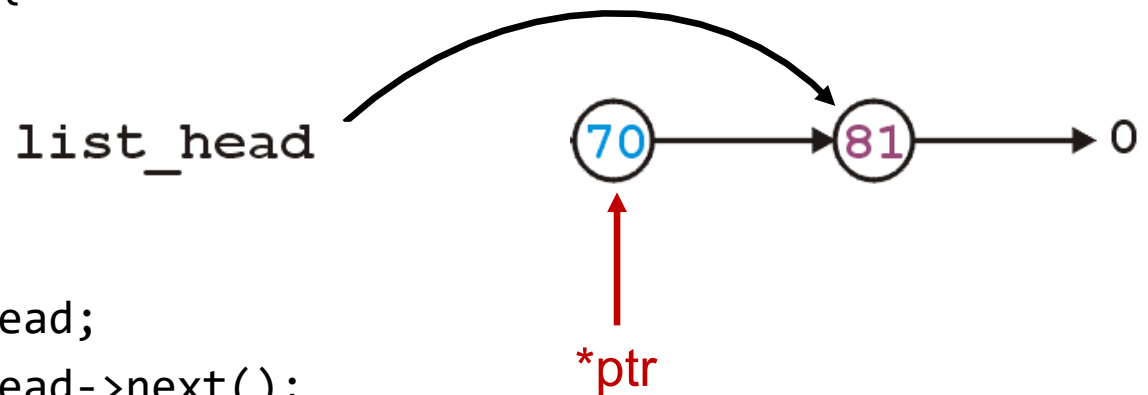
```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```



# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

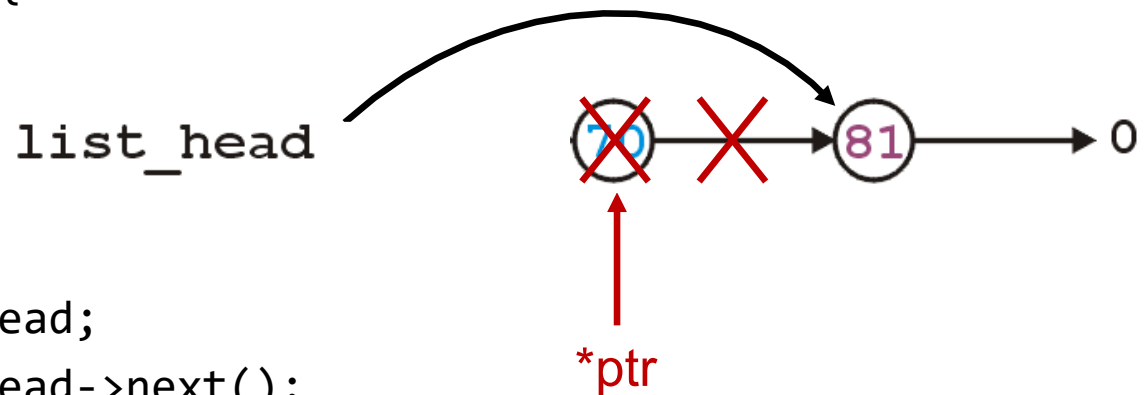
```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```



# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```



# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```



# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```





# 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

# Array Implementation

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



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

# Stack-as-Array Class

```
template <typename Type>
class Stack {
    private:
        int stack_size; //number of objects in the stack
        int array_capacity; //capacity of the array
        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

```
#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 destructor must release the memory for the array

```
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 );
}
```

# 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();
    }

    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

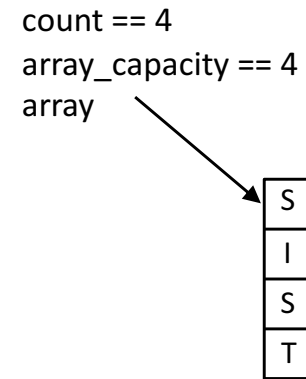
The best option is to increase the array capacity

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

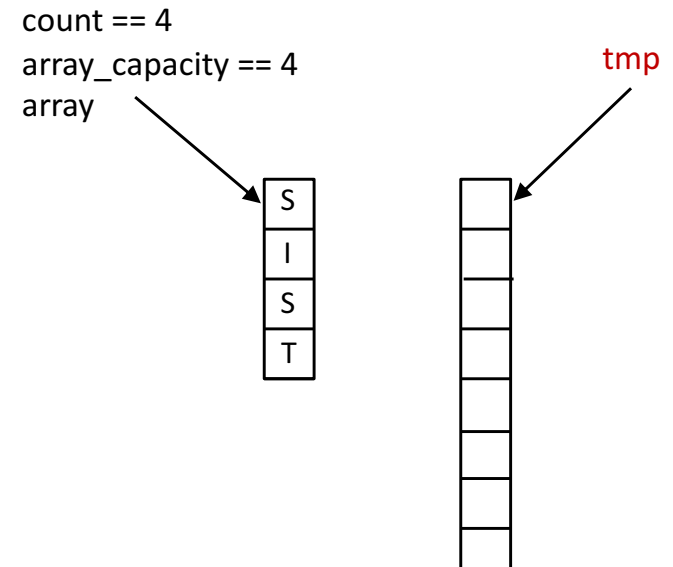
First, let us visualize what must occur to allocate new memory



# Array Capacity

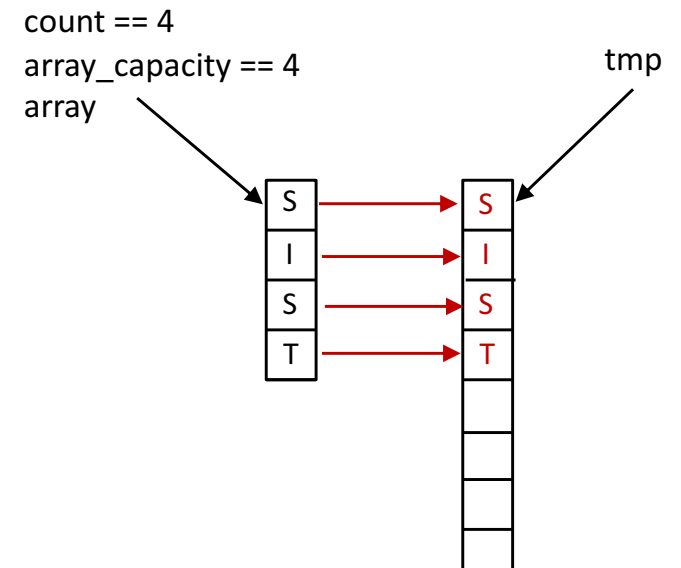
First, this requires a call to `new Type[N]` where  $N$  is the new capacity

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



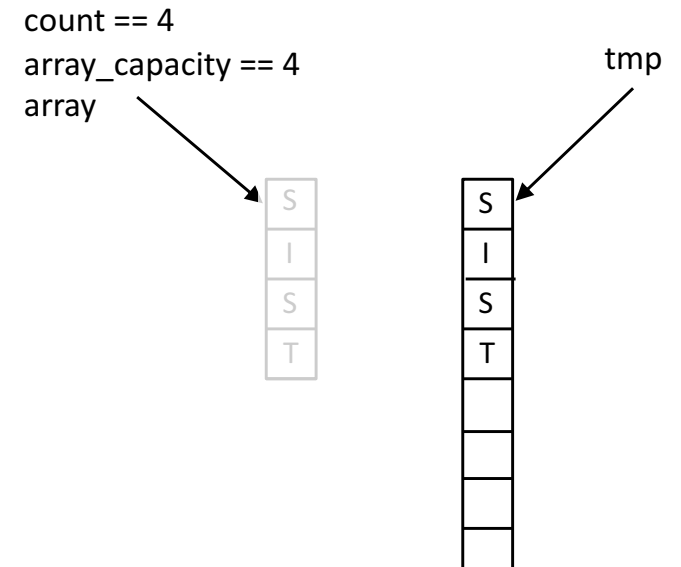
# Array Capacity

Next, the values must be copied over



# Array Capacity

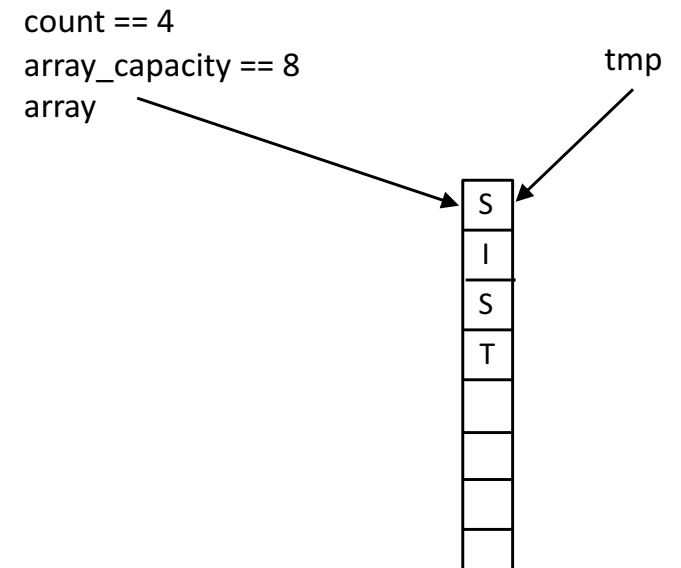
The memory for the original array must be deallocated





# Array Capacity

Finally, the appropriate member variables must be reassigned

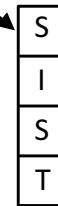


# Array Capacity

The implementation:

```
void double_capacity() {
```

```
count == 4  
array_capacity == 4  
array
```



```
}
```

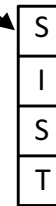
# Array Capacity

The implementation:

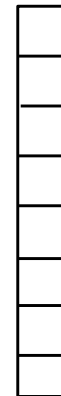
```
void double_capacity() {
```

```
    Type *tmp_array = new Type[2*array_capacity];
```

count == 4  
array\_capacity == 4  
array



tmp\_array

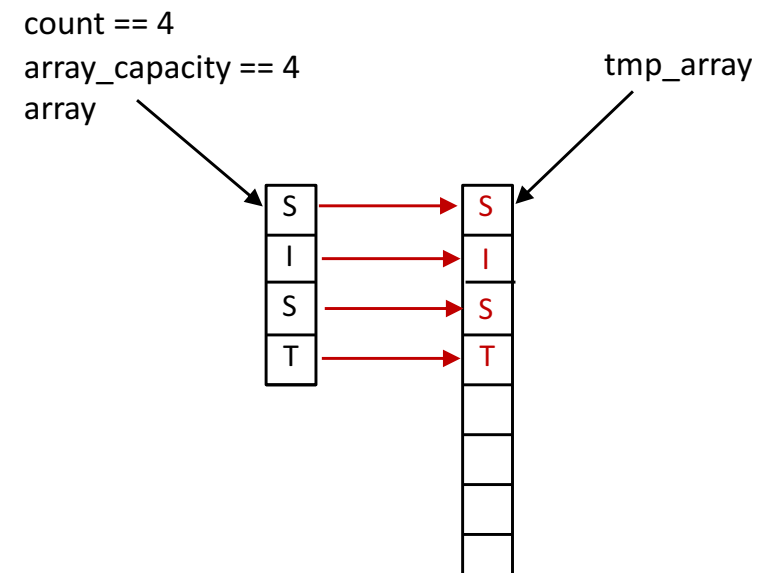


```
}
```

# 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];  
    }  
  
}
```



# 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;  
  
}
```

count == 4  
array\_capacity == 4  
array



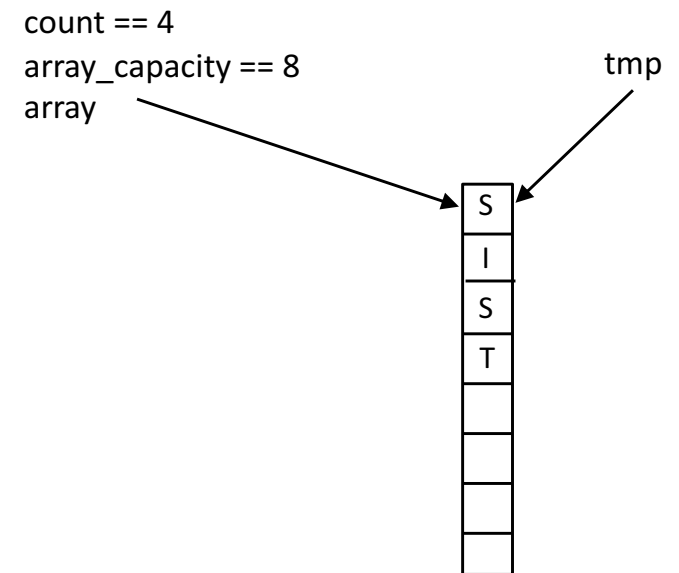
tmp\_array



# Array Capacity

The implementation:

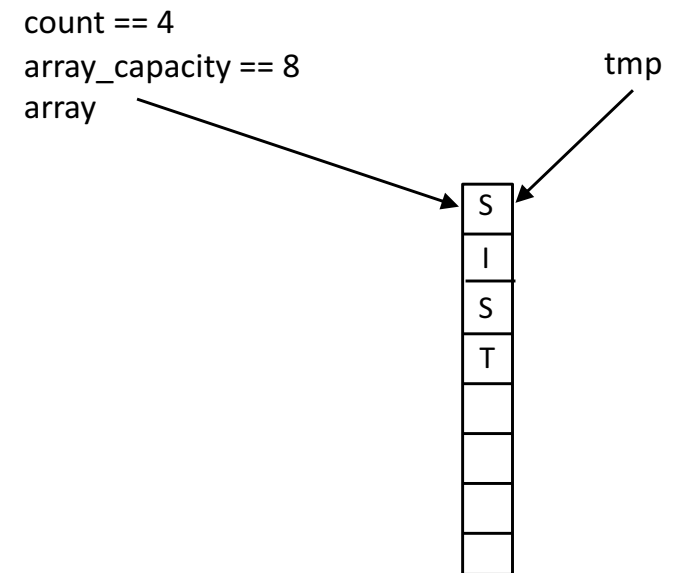
```
void double_capacity() {  
    Type *tmp_array = new Type[2*array_capacity];  
  
    for ( int i = 0; i < array_capacity; ++i ) {  
        tmp_array[i] = array[i];  
    }  
  
    delete [] array;  
    array = tmp_array;  
  
}
```



# Array Capacity

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

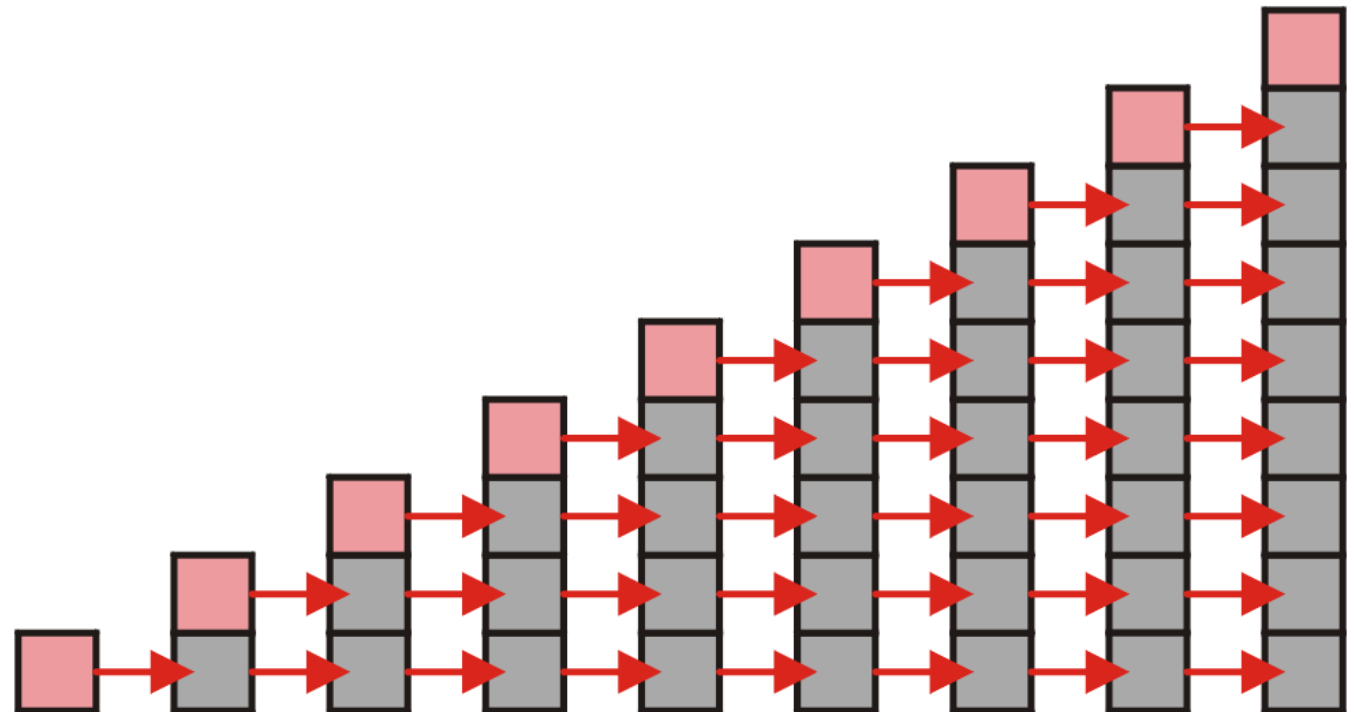
To state the average run time, we will introduce the concept of **amortized time**:

- If  $n$  operations requires  $\Theta(f(n))$ , 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  $\Theta(1)$

# 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 insert  $k$  objects

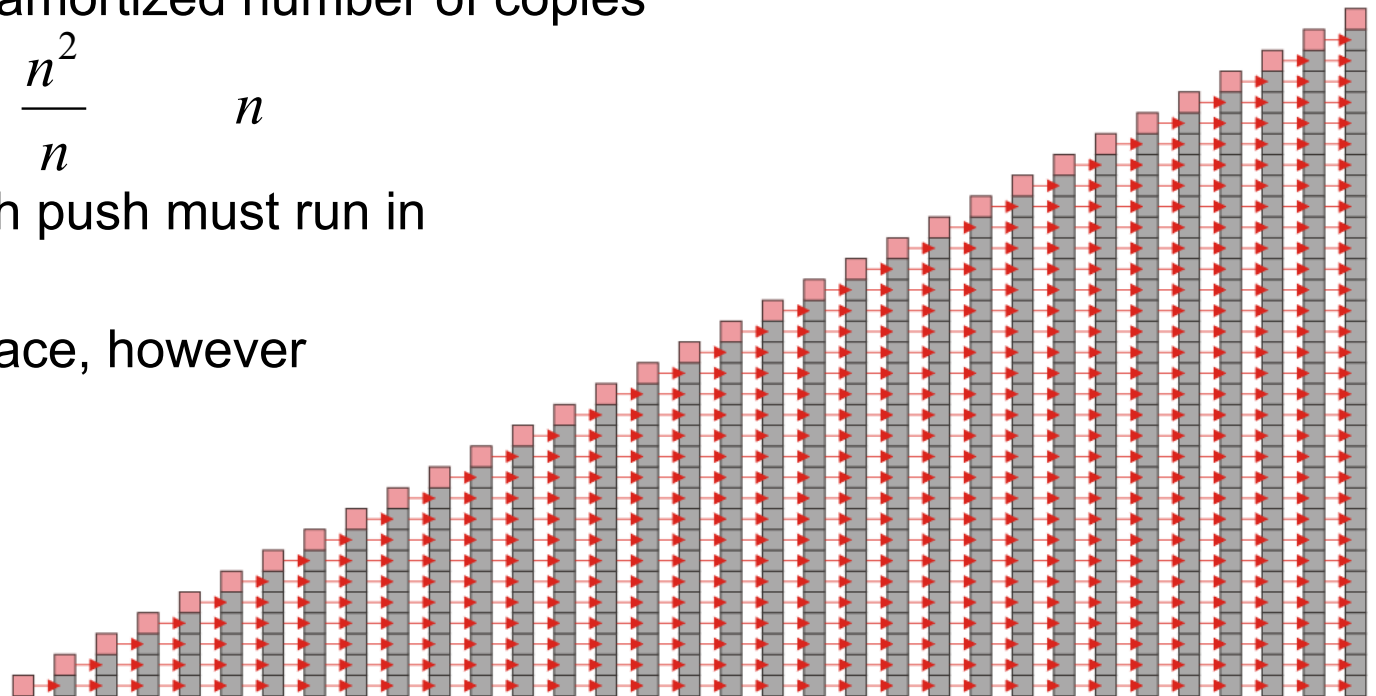
- The pushing of the  $k^{\text{th}}$  object on the stack requires  $k-1$  copies
- The total number of copies is now given by:

$$\sum_{k=1}^n (k-1) = \sum_{k=1}^n k - n = \frac{n(n-1)}{2} = n \cdot \frac{n-1}{2} = \frac{n^2}{2}$$

- Therefore, the amortized number of copies is given by

$$\frac{\frac{n^2}{2}}{n} = \frac{n}{2}$$

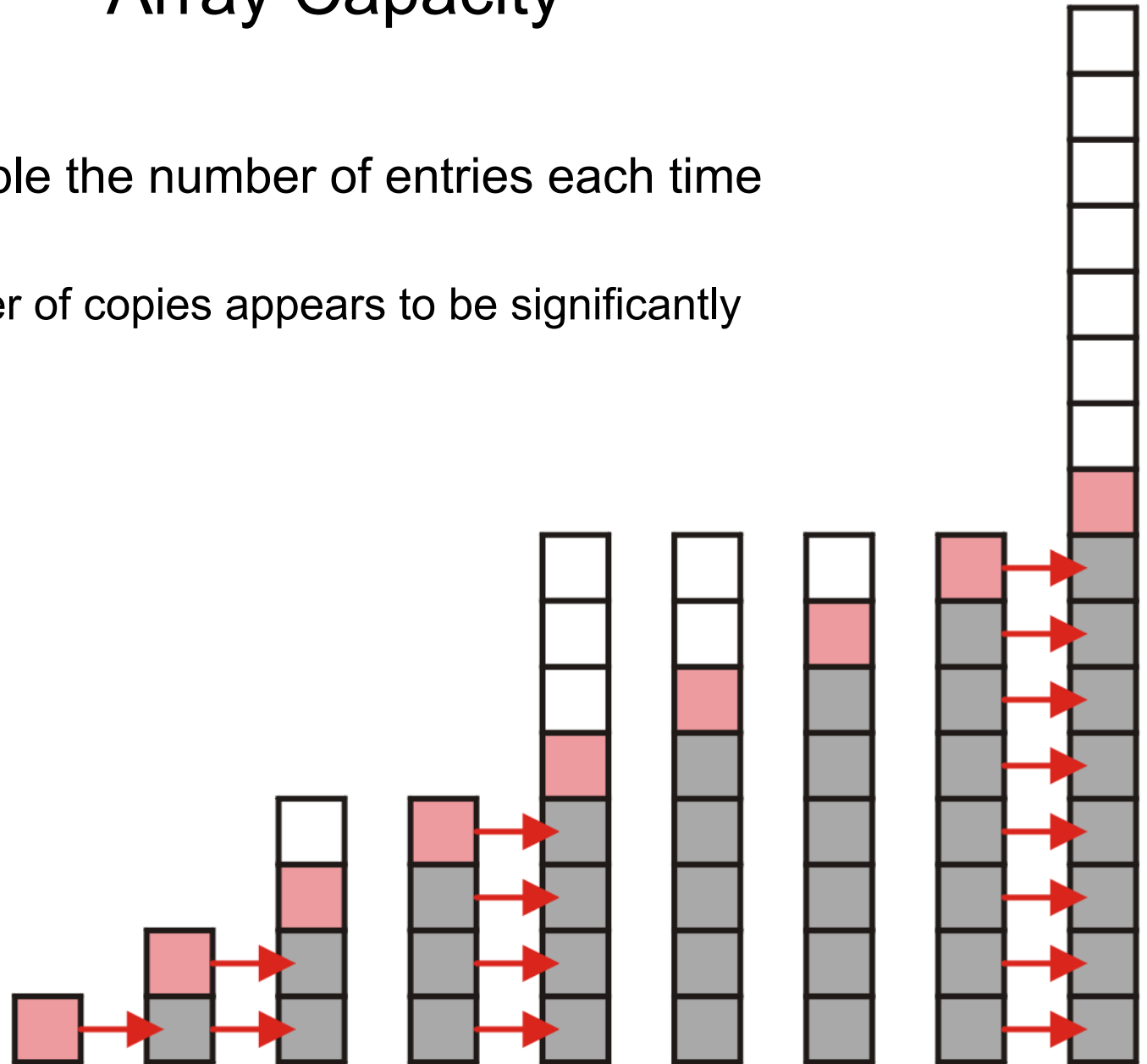
- Therefore, each push must run in  $\Theta(n)$  time
- The wasted space, however is  $\Theta(1)$



# 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

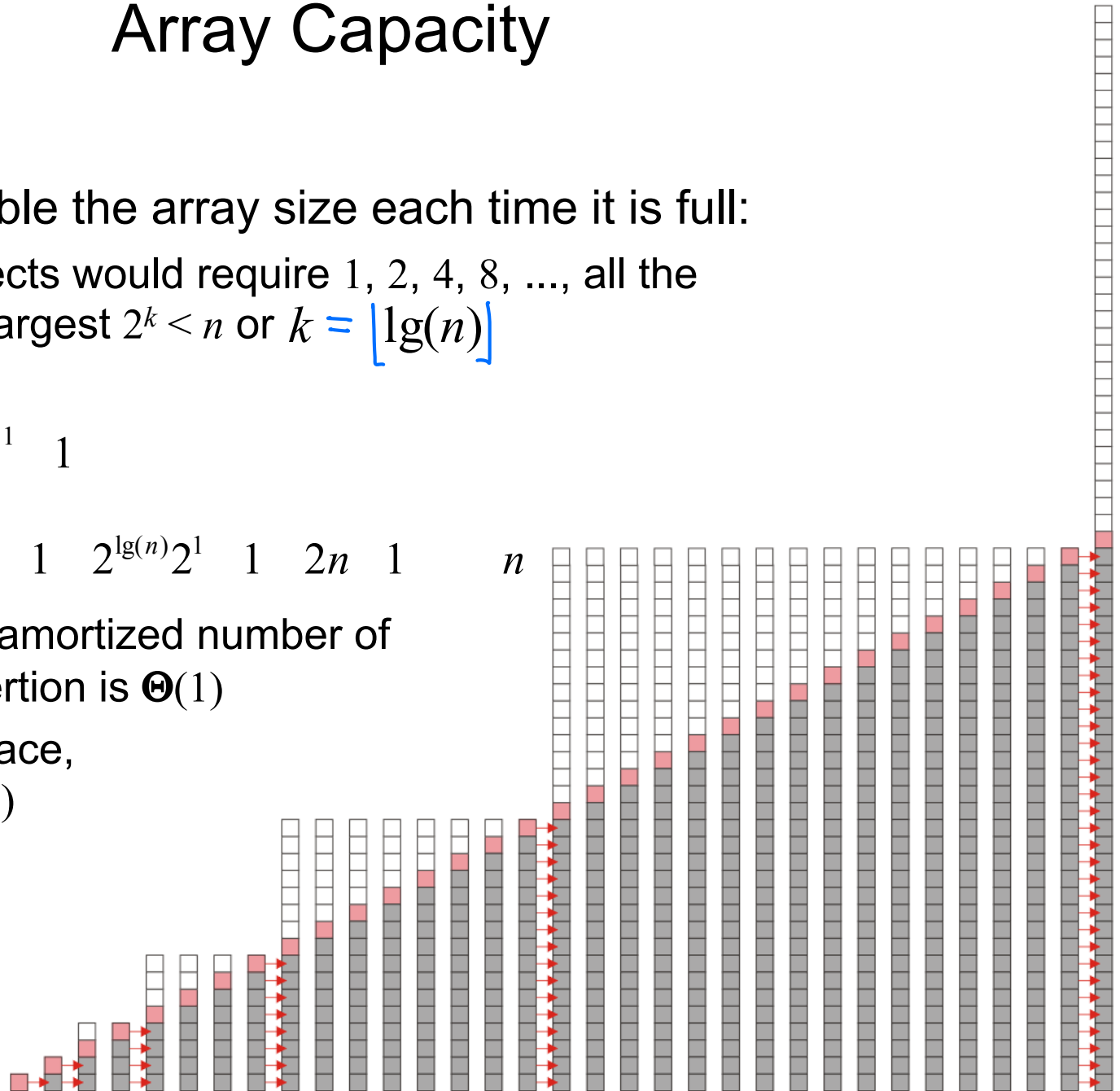
Suppose we double the array size each time it is full:

- Inserting  $n$  objects would require 1, 2, 4, 8, ..., all the way up to the largest  $2^k < n$  or  $k = \lfloor \lg(n) \rfloor$

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

$$2^{\lg(n)-1} + 1 + 2^{\lg(n)-1} + 1 + \dots + 2n + 1 = n$$

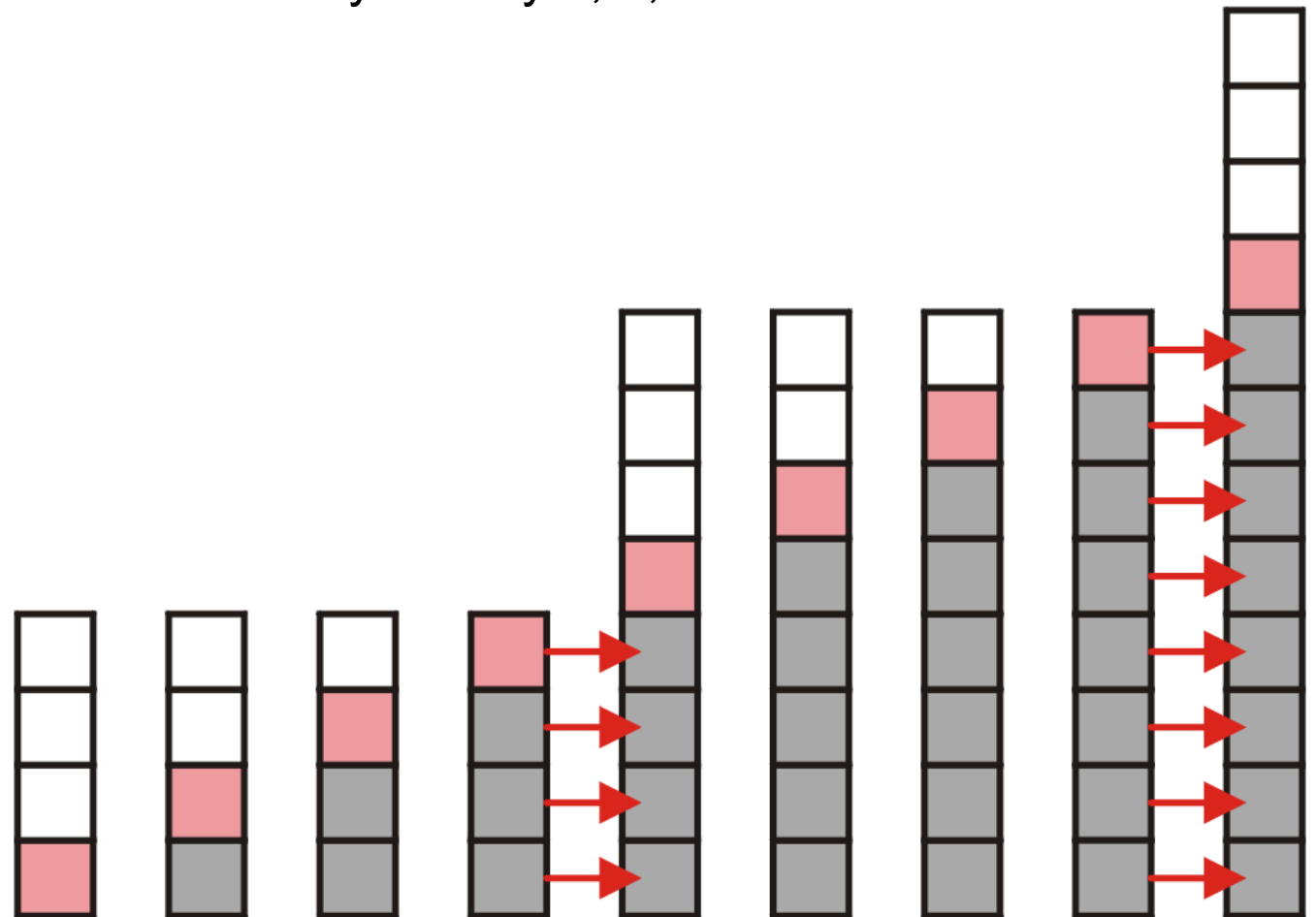
- Therefore, the amortized number of copies per insertion is  $\Theta(1)$
- The wasted space, however is  $O(n)$



# Array Capacity

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

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

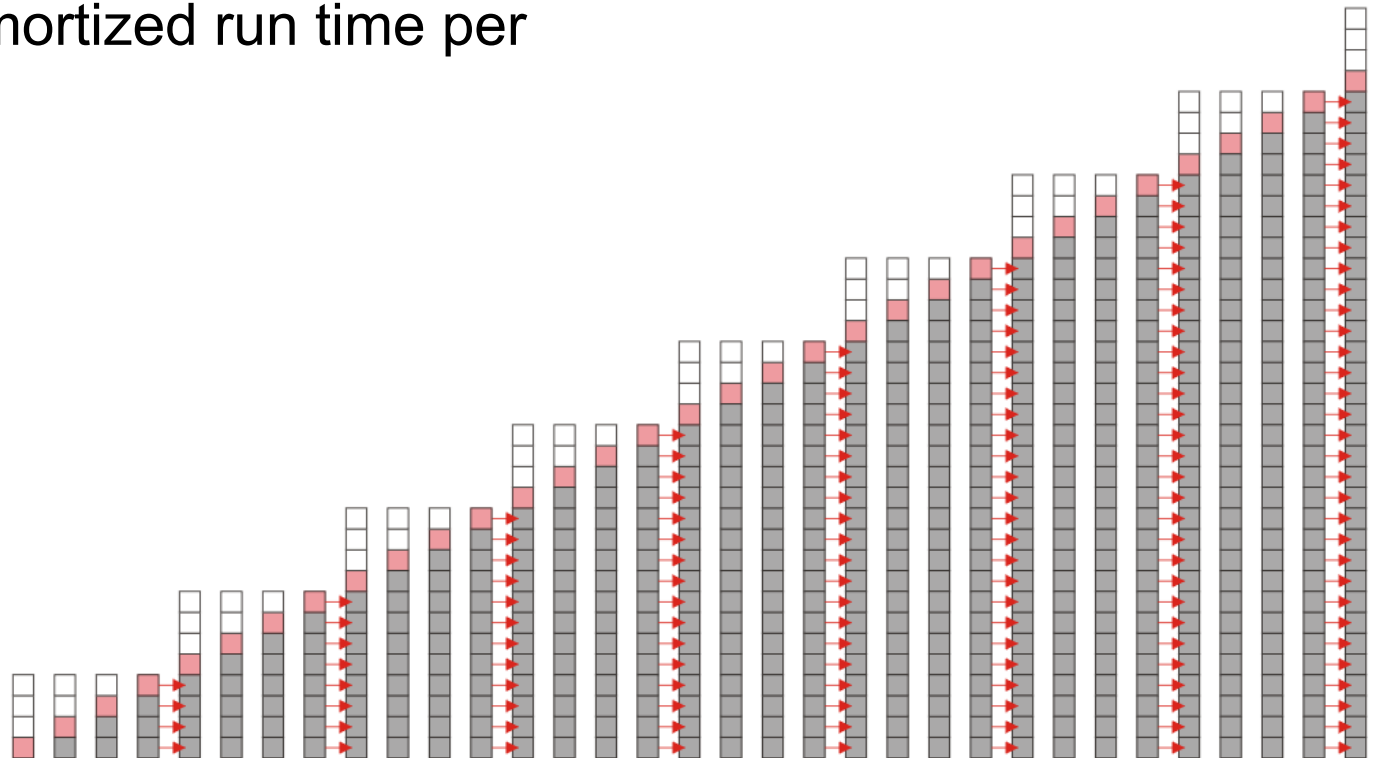


# Array Capacity

Suppose we increase it by a **constant** value  $m$

$$\sum_{k=1}^{n/m} mk + \sum_{k=1}^{n/m} k = \frac{\frac{n}{m} \cdot \frac{n}{m} \cdot 1}{2} = \frac{n^2}{2m} = \frac{n}{2} \cdot \frac{n}{m} = \Theta(n^2)$$

Therefore, the amortized run time per insertion is  $\Theta(n)$



# 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 $r > 1$	$1/(r - 1)$	$(r - 1)n$



# Outline

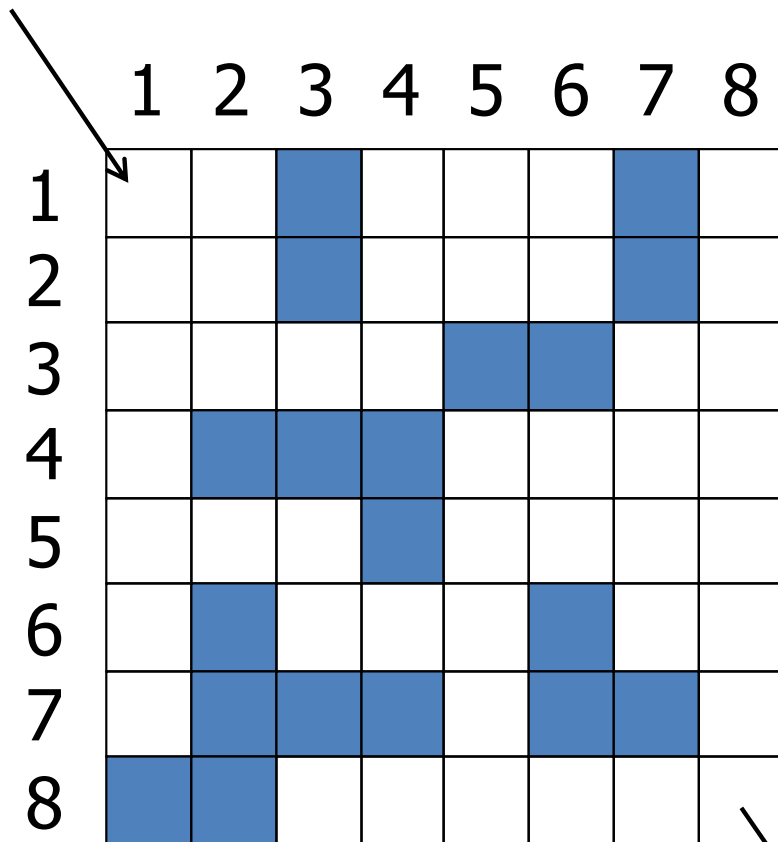
- Stack ADT
- Implementation
- Example applications

# Example: Maze Routing

## Objective:

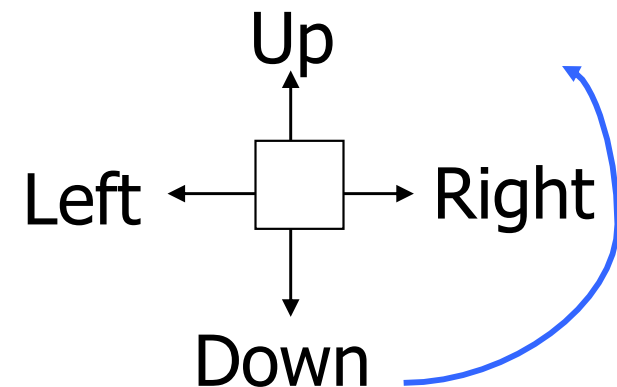
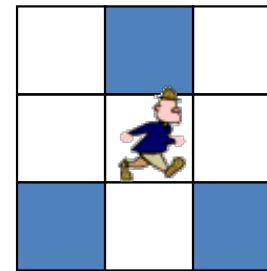
One path from the entrance to the exit

## Entrance



## Directions:

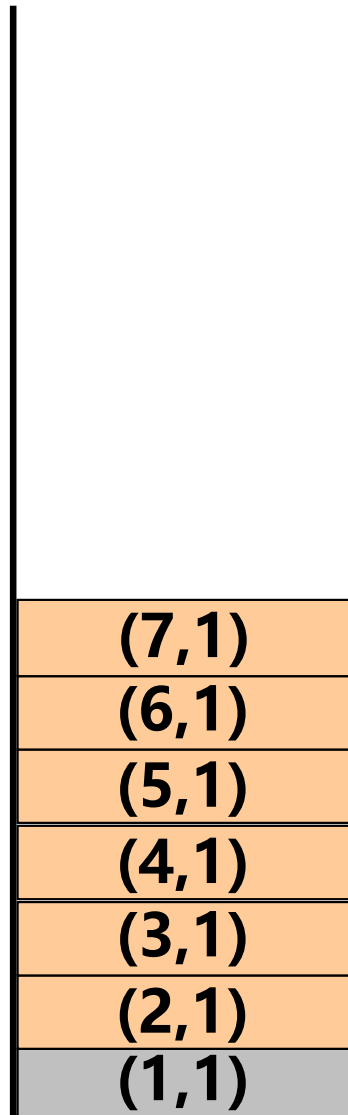
Up(N), Down(S), Left(W), Right(E)




## Rules:

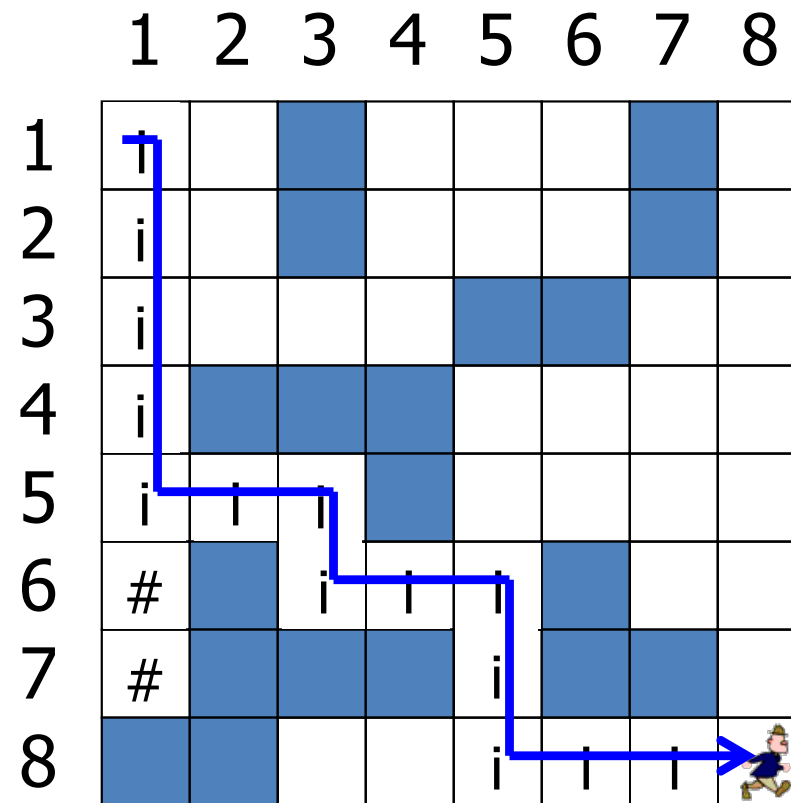
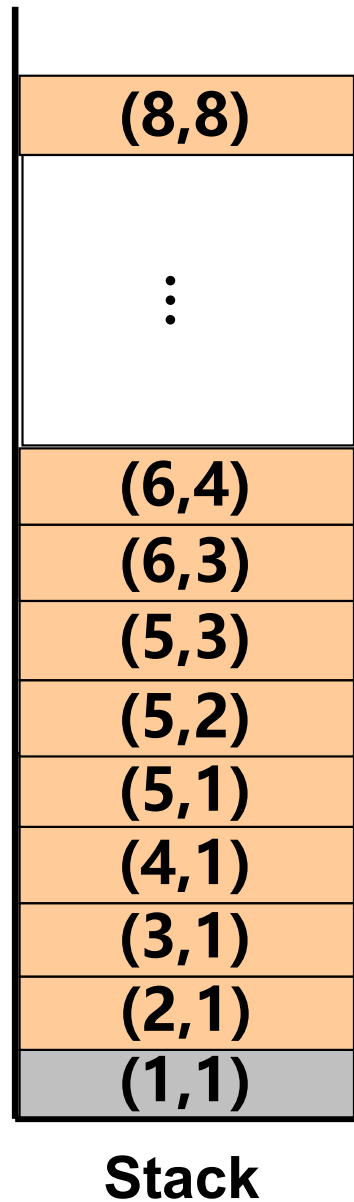
Go down first. If possible, the man can further explore. Otherwise, the man explores the possible next steps in a counterclockwise direction.

# Example: Maze Routing



	1	2	3	4	5	6	7	8
1	i							
2	i							
3	i							
4	i							
5	i							
6	i							
7								
8								

# Example: Maze Routing



# Application: Parsing

Most parsing uses stacks

Examples includes:

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

# Parsing XHTML

*A markup language* is a means of annotating a document to given context to the text

- The annotations give information about the structure or presentation of the text

The best known example is HTML, or HyperText Markup Language

- We will look at XHTML

# 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>
```

*Nesting* indicates that any closing tag must match the most recent opening tag

# Parsing XHTML

Strategy for parsing XHTML:

- read through 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>**

<b>&lt;html&gt;</b>			
---------------------	--	--	--

# 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

# Parsing C++

Like opening and closing tags, C++ parentheses, brackets, and braces must be similarly nested:

```
void initialize( int *array, int n ) {  
    for ( int i = 0; i < n; ++i ) {  
        array[i] = 0;  
    }  
}
```

# Function calls

```
int a() {  
    b();  
    c();  
    return 0;  
}
```

```
int b() { return 0; }
```

```
int c() { return 0; }
```

```
int main() {  
    a();  
    return 0;  
}
```

# Function calls

<b>main</b>			
-------------	--	--	--

main() calls a()

<b>main</b>	<b>a</b>		
-------------	----------	--	--

a() calls b()

<b>main</b>	<b>a</b>	<b>b</b>	
-------------	----------	----------	--

b() returns

<b>main</b>	<b>a</b>		
-------------	----------	--	--

a() calls c()

<b>main</b>	<b>a</b>	<b>c</b>	
-------------	----------	----------	--

c() returns

<b>main</b>	<b>a</b>		
-------------	----------	--	--

a() returns

<b>main</b>			
-------------	--	--	--



# Function calls

```
int a() {  
    return a();  
}
```

# Function calls

calls a()

<b>a</b>			
----------	--	--	--

a() calls a()

<b>a</b>	<b>a</b>		
----------	----------	--	--

a() calls a()

<b>a</b>	<b>a</b>	<b>a</b>	
----------	----------	----------	--

a() calls a()

<b>a</b>	<b>a</b>	<b>a</b>	<b>a</b>
----------	----------	----------	----------

a() calls a()

<b>a</b>	<b>a</b>	<b>a</b>	<b>a</b>
----------	----------	----------	----------



*Stack Overflow!*

# StackOverflow

StackExchange ▾

[sign up](#) [log in](#) [tour](#) [help](#) ▾ [stack overflow careers](#)



Questions

Tags

Users

Badges

Unanswered

Ask Question

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

[Take the 2-minute tour](#)

Here's how it works:



Anybody can ask a question



Anybody can answer



The best answers are voted up and rise to the top

## Top Questions

interesting

404

featured

hot

week

month

0 votes 0 answers 1 view

Values being passed into a function are recognised when I print them out in that function, but don't seem to be recognised if I run an if statement?

c

asked 51 secs ago Kev O'Brien 1

0 votes 0 answers 10 views

SQL Unique Distinct Column 1 and not the other columns

sql sql-server-2008 unique distinct

modified 53 secs ago collapsar 5,394

0 votes 0 answers 2 views

Colon Separated List in SQL Server (Like LISTAGG)

sql-server-2014

asked 1 min ago cscratch 1

0 votes 0 answers 6 views

font-family: "Shojumaru",cursive,Arial,serif; , is not displaying correctly on

## Hot Network Questions

Custom action just after commit deployment phase of publishing transaction

How to cut paper without scissors?

I am a PhD student and hate it here. How can I warn prospective students during admit weekend without ruining my reputation?

How to hold SMD parts in place while soldering?

Calculating range of ArrayLists

Compiling small part of text

Why must resistors be on the respective anode terminals instead

# Summary

- Stack ADT
  - Push, pop, LIFO
- Implementation
  - Linked list
  - Array
    - How to increase the array capacity
- Example Applications
  - Reverse-Polish Notation
  - Maze Routing

# 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`

