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:

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

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

$$35 \;\; 6 \;\; -$$

$$29$$

Reverse-Polish Notation

Other examples:

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

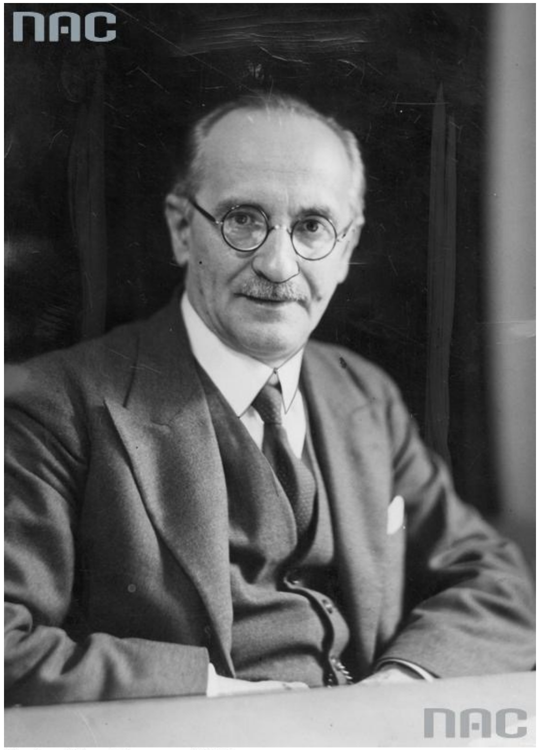
$$3 + 4 \times 5 - 6 = 17$$

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

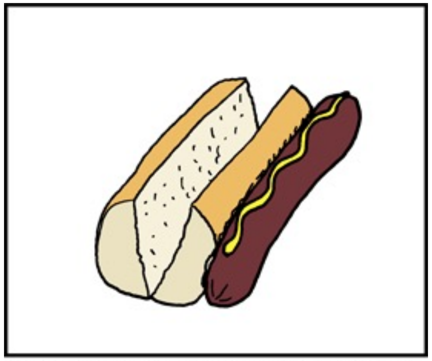
$$3 + 4 \times (5 - 6) = -1$$

Reverse-Polish Notation

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



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

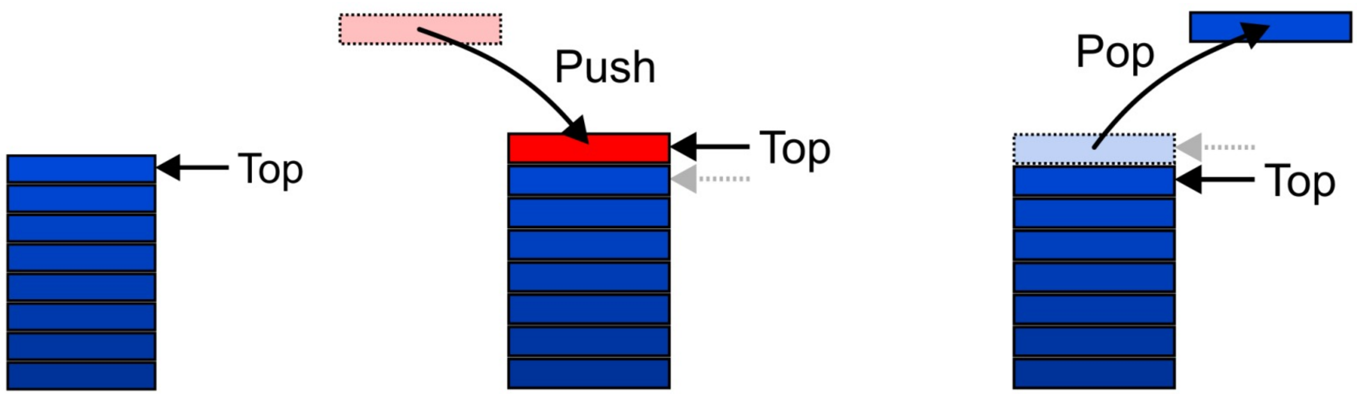
The easiest way to parse reverse-Polish notation is to use an operand stack:

- 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

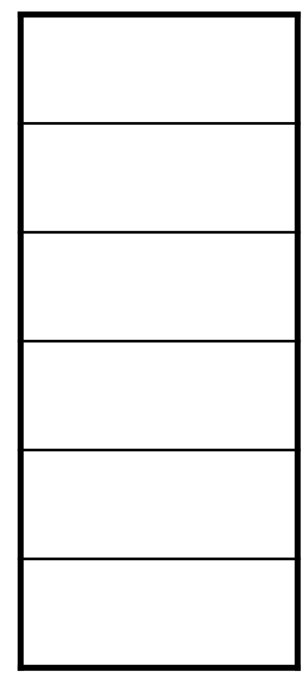
Numerous applications:

- 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:

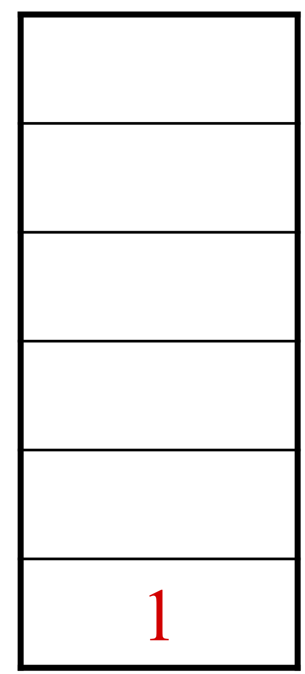
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Push 1 onto the stack

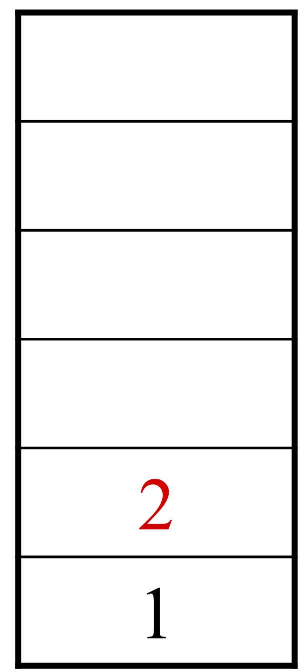
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 1 onto the stack

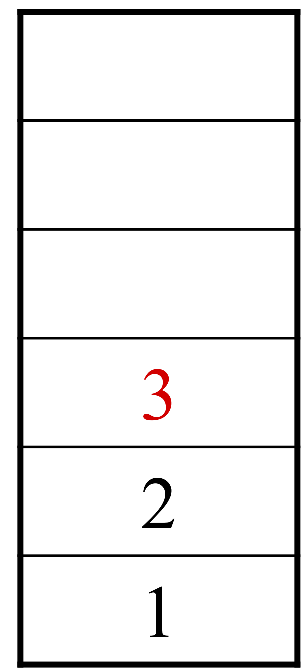
$1\ \ 2\ \ 3\ \ +\ \ 4\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$



Reverse-Polish Notation

Push 3 onto the stack

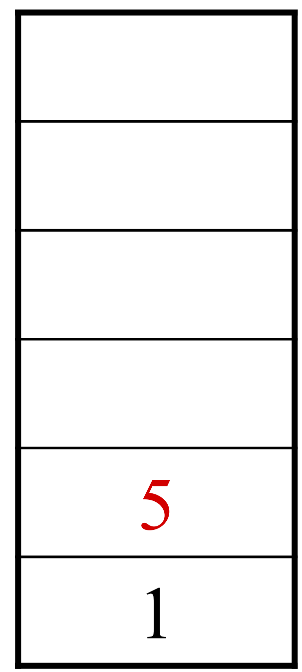
$$1 \quad 2 \quad \color{red}{3} \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

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

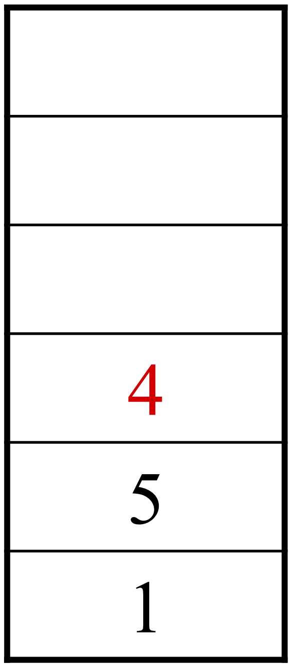
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Push 4 onto the stack

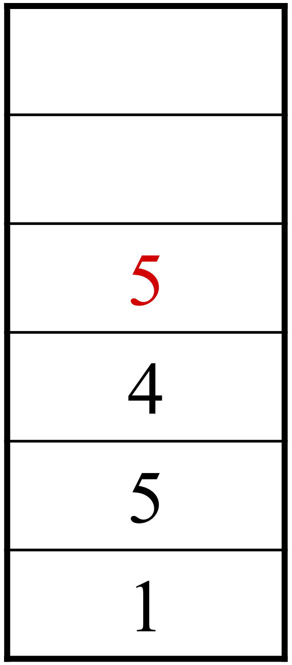
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 5 onto the stack

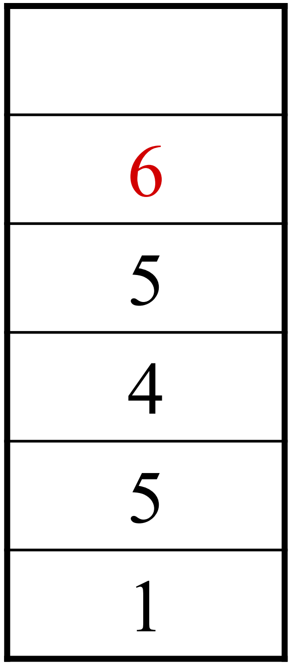
$$1\ \ 2\ \ 3\ \ +\ \ 4\ \ {\color{red}5}\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$



Reverse-Polish Notation

Push 6 onto the stack

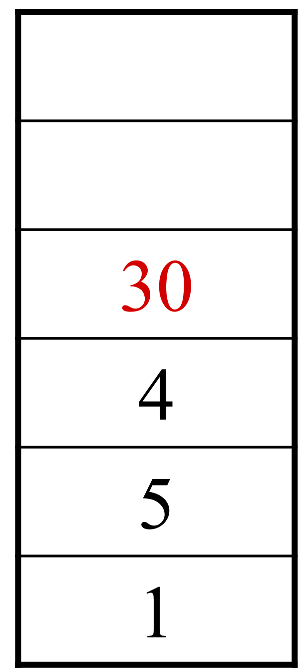
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

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

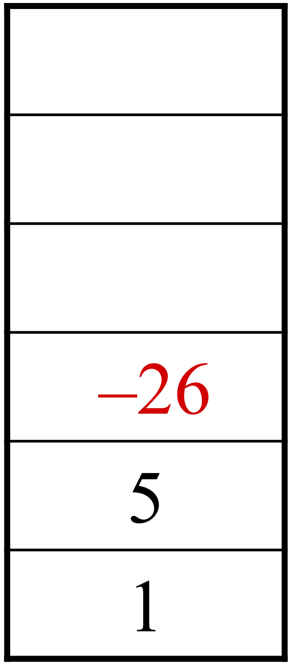
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

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

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



Reverse-Polish Notation

Push 7 onto the stack

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

7
–26
5
1

Reverse-Polish Notation

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

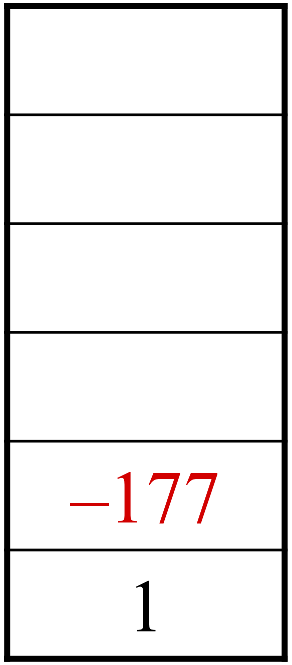
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

-182
5
1

Reverse-Polish Notation

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

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



Reverse-Polish Notation

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

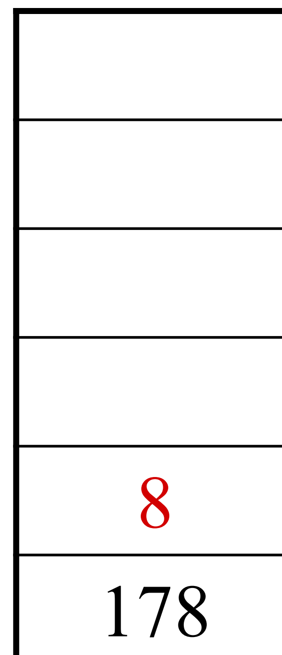
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

178

Reverse-Polish Notation

Push 8 onto the stack

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



Reverse-Polish Notation

Push 1 onto the stack

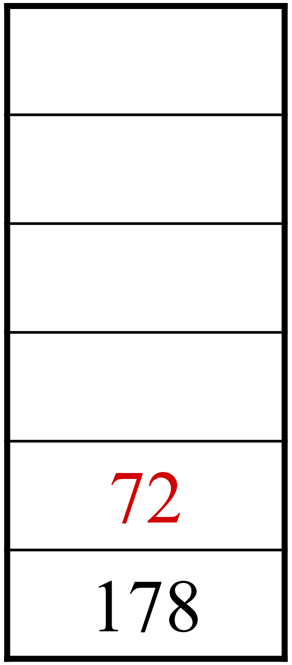
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

9
8
178

Reverse-Polish Notation

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

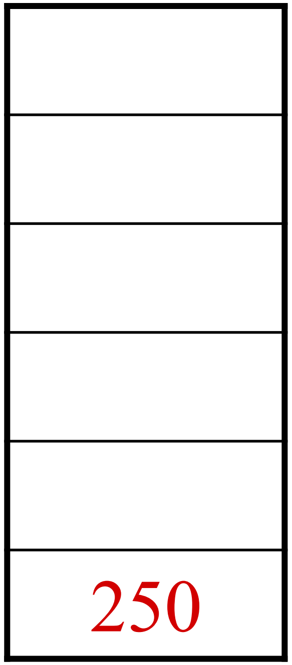
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



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:

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

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:

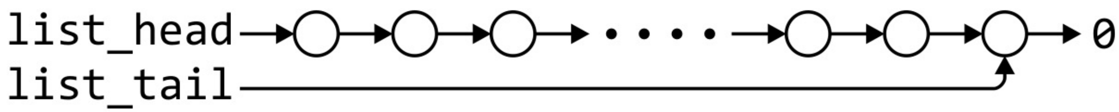
- Singly linked lists
- One-ended arrays

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^{st}	**Back/n^{th}**
Find		
Insert		
Erase		

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

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:

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

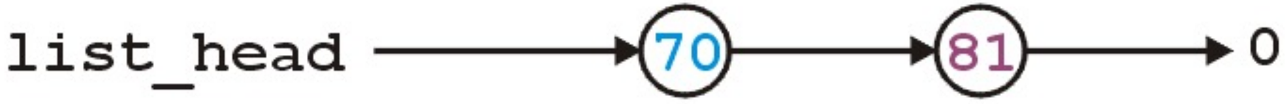
void push_front(int)

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:



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;
}
```

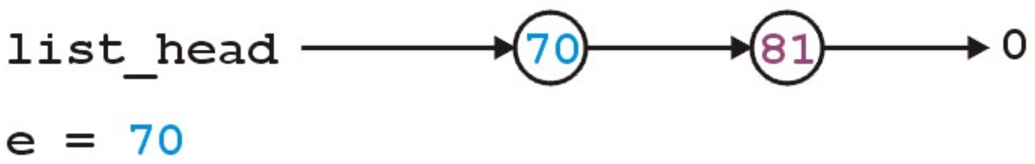
int front() const

```
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();
    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()

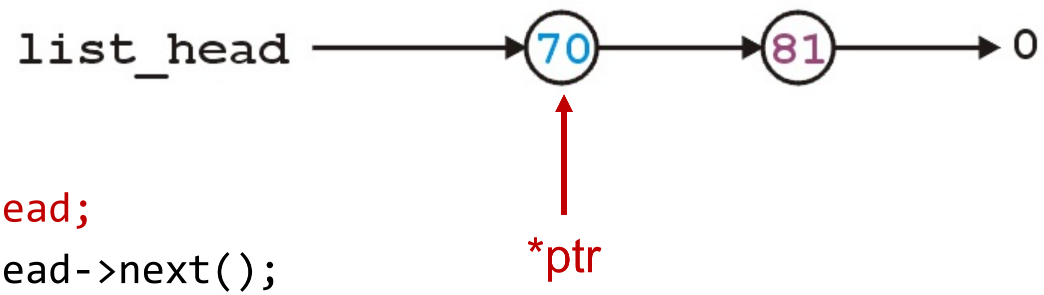
The correct implementation assigns a temporary pointer to point to the node being deleted:

```
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:

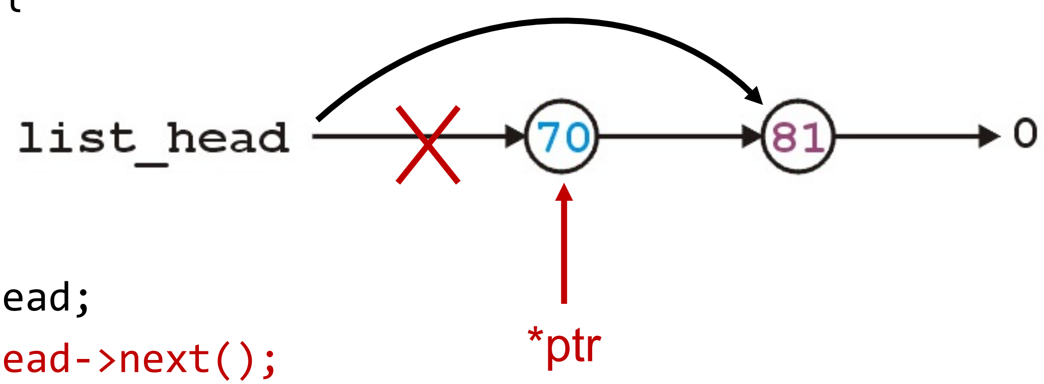
```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    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:

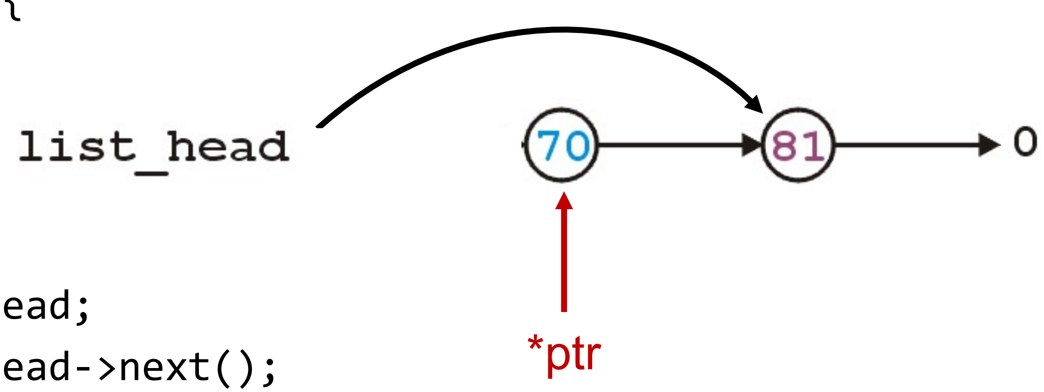
```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    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:

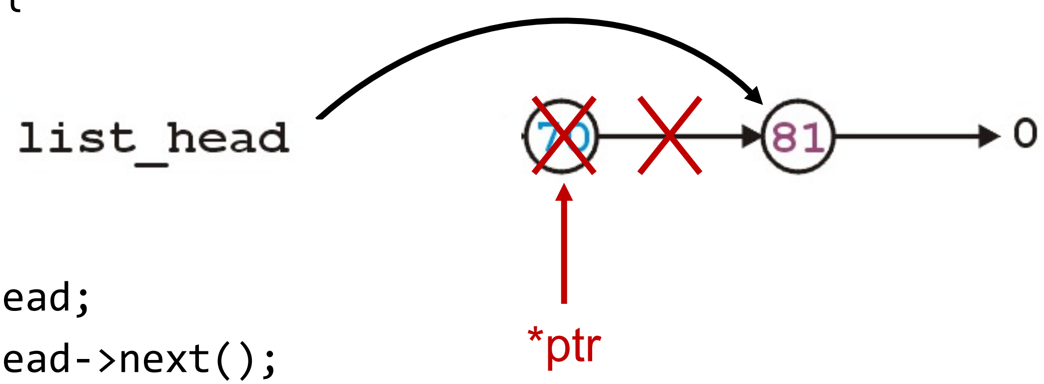
```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    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;
}
```

Array Implementation

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

	Front/1^{st}	**Back/n^{th}**
Find		
Insert		
Erase		

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;
}
```

Array Capacity

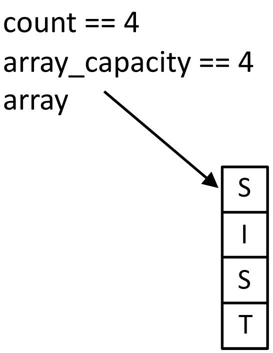
The best option is to increase the array capacity

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

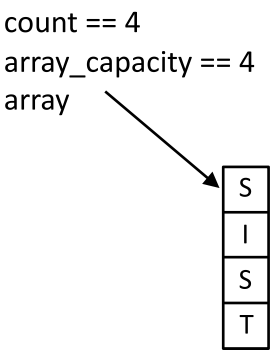
First, let us visualize what must occur to allocate new memory



Array Capacity

The implementation:

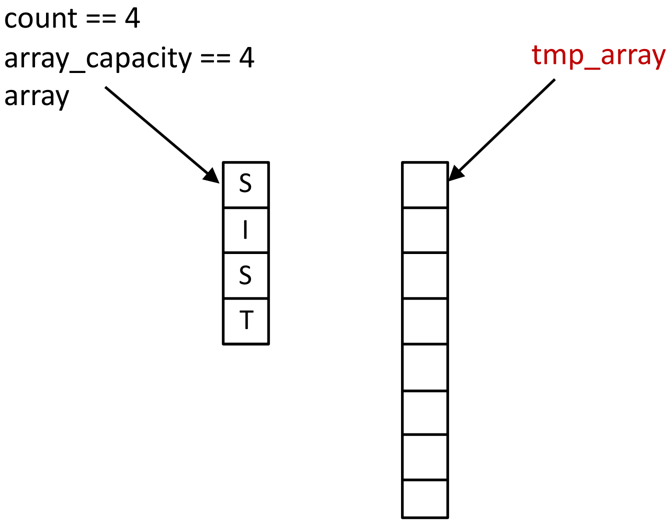
```
void double_capacity() {

}
```



Array Capacity

The implementation:

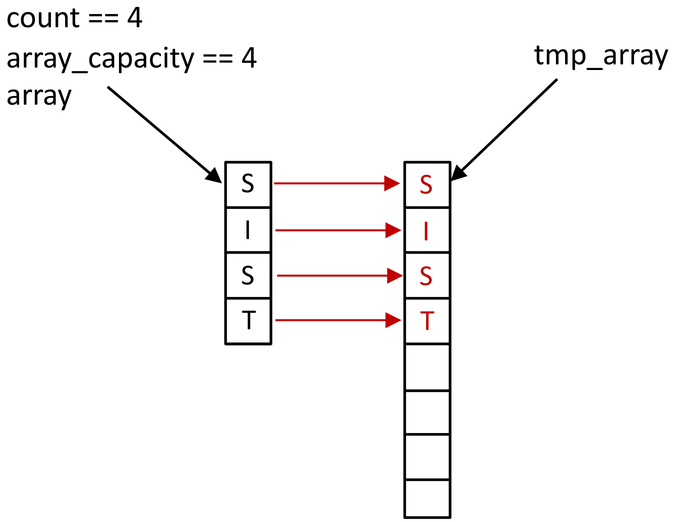
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];



}
```



Array Capacity

The implementation:

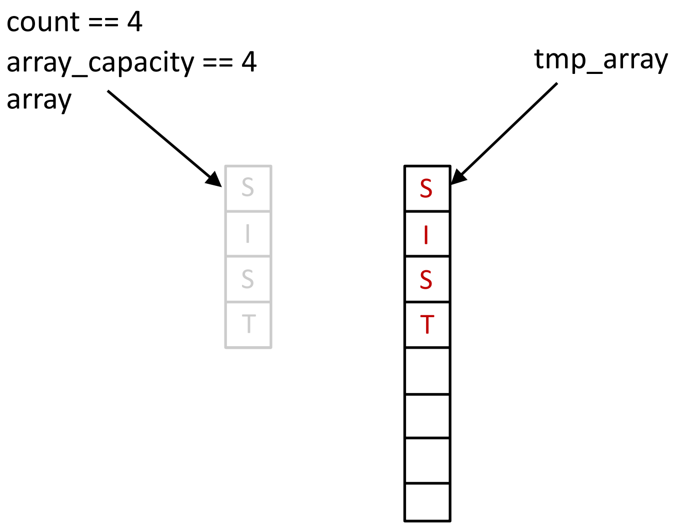
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

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




}
```



Array Capacity

The implementation:

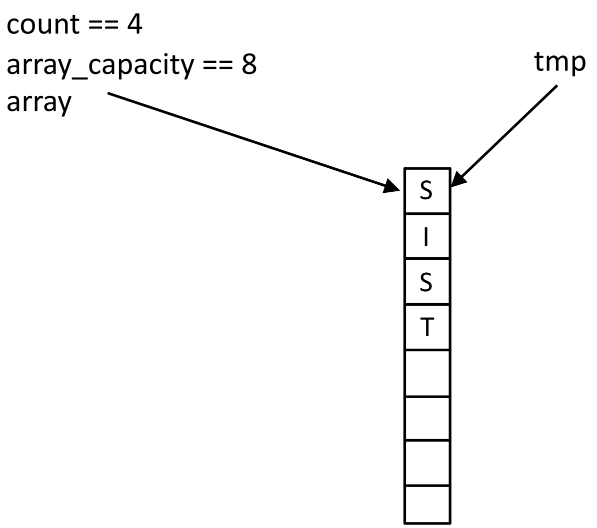
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

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

    delete [] array;



}
```



Array Capacity

The implementation:

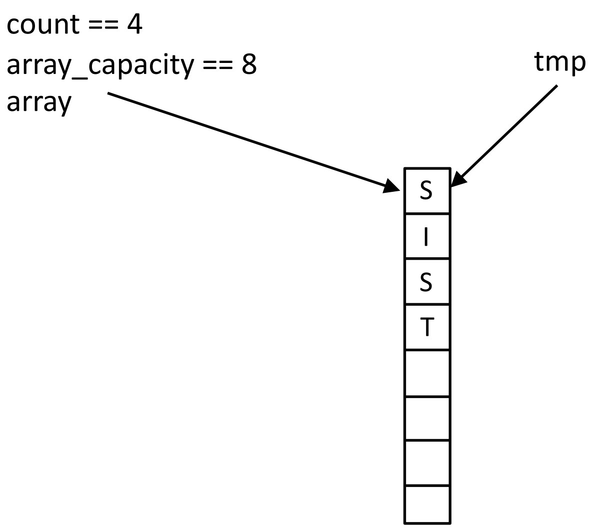
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

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

    delete [] array;
    array = tmp_array;


}
```



Array Capacity

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

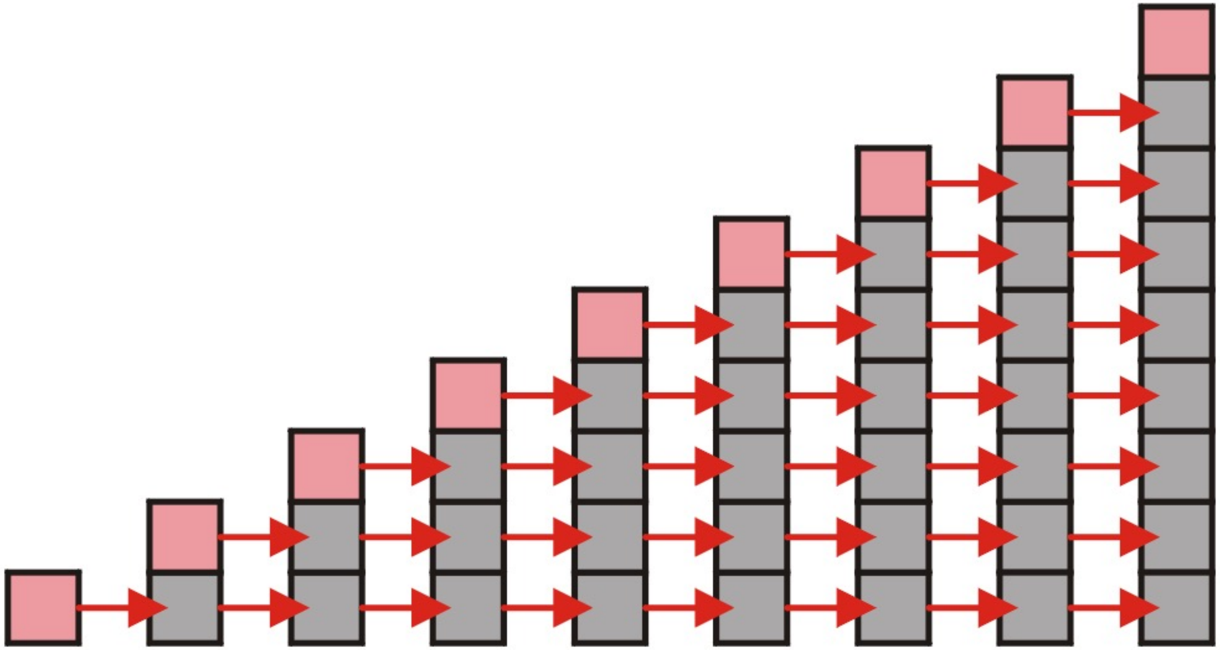
To state the average run time, we will introduce the concept of **amortized time:**

- 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



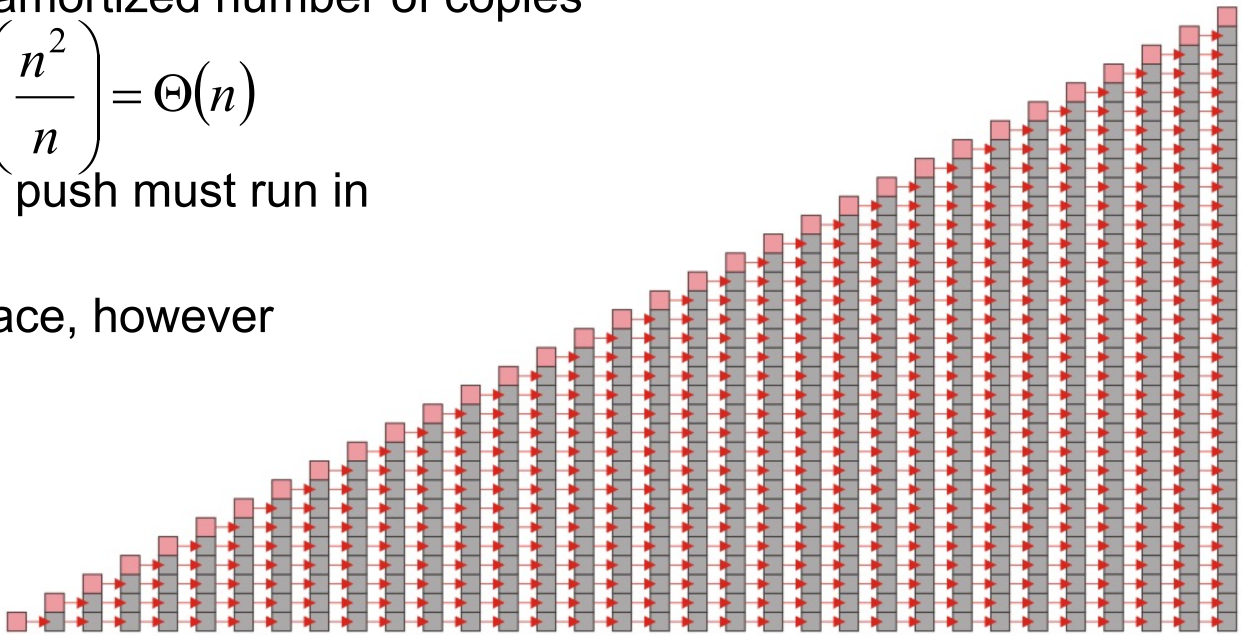
Array Capacity

Suppose we insert k objects

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

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

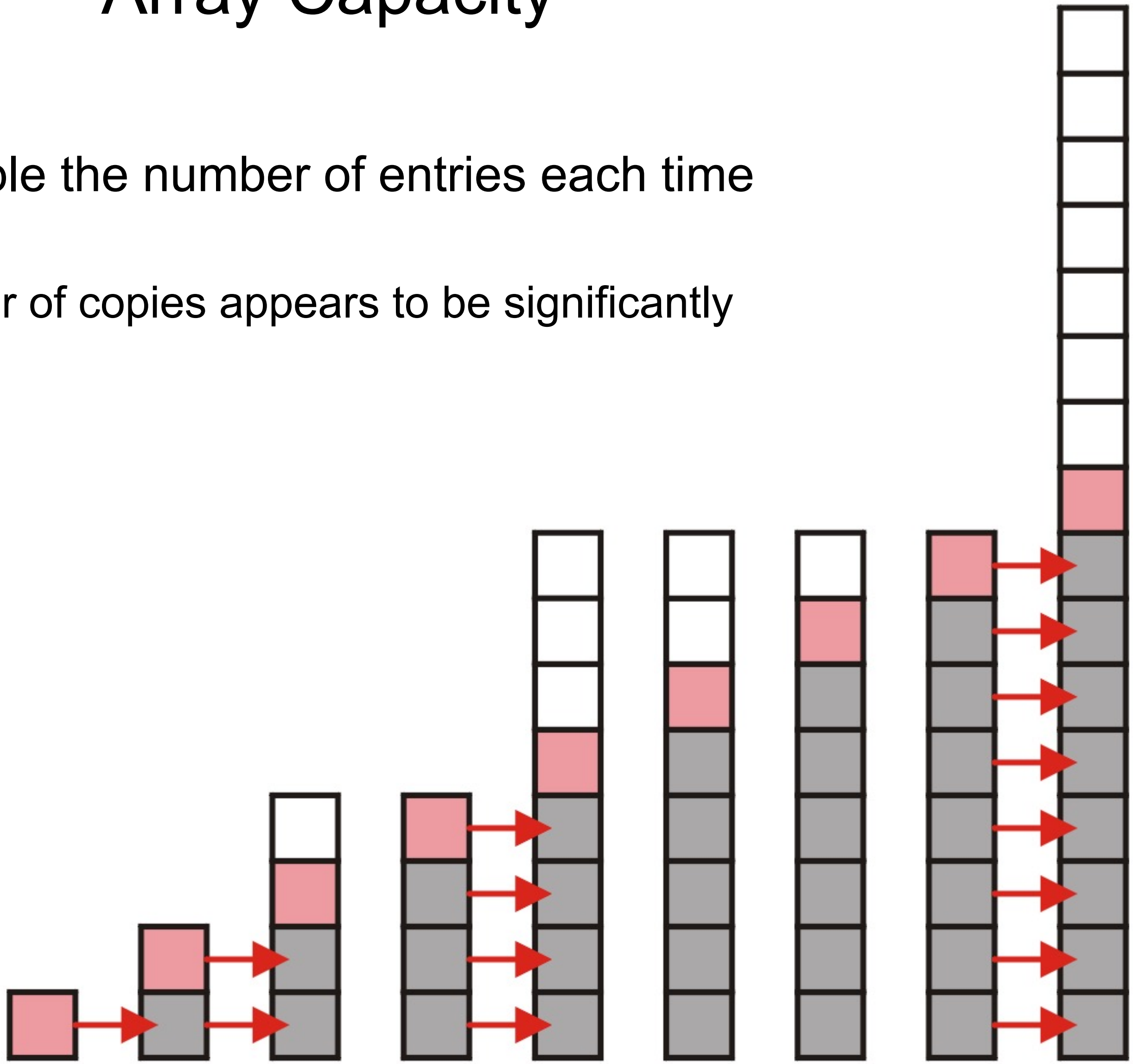
- Therefore, the amortized number of copies is given by $$\Theta\left(\frac{n^2}{n}\right)=\Theta(n)$$
- Therefore each push must run in $\Theta(n)$ time
- The wasted space, however is $\Theta(1)$



Array Capacity

Suppose we double the number of entries each time the array is full

- Now the number of copies appears to be significantly fewer



Array Capacity

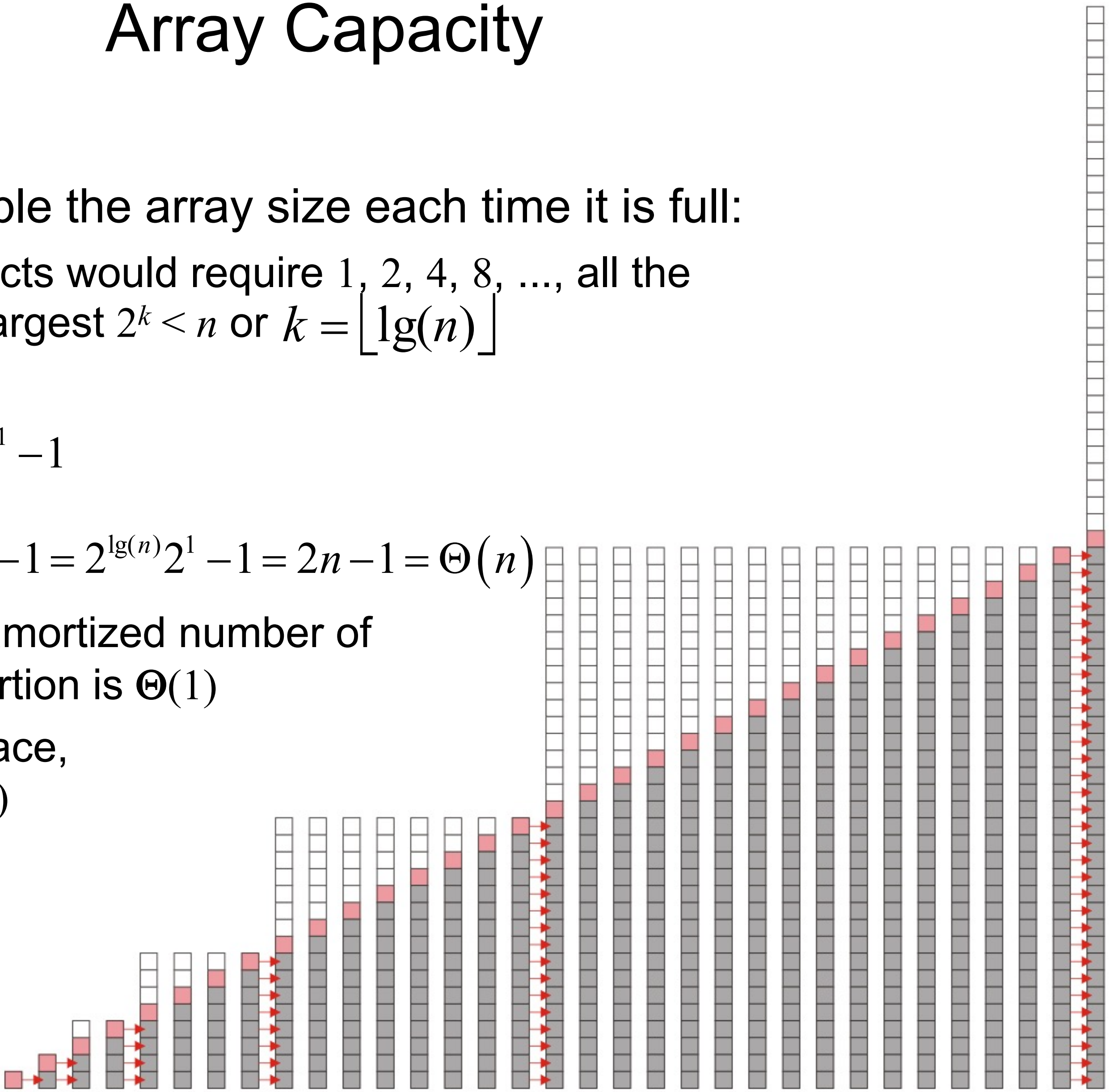
Suppose we double the array size each time it is full:

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

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

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

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



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

Summary

- Stack ADT
  - Push, pop, LIFO
- Implementation
  - Linked list
  - Array
    - How to increase the array capacity
- Applications
  - Parsing XHTML
  - Function calls
  - Reverse-Polish Notation