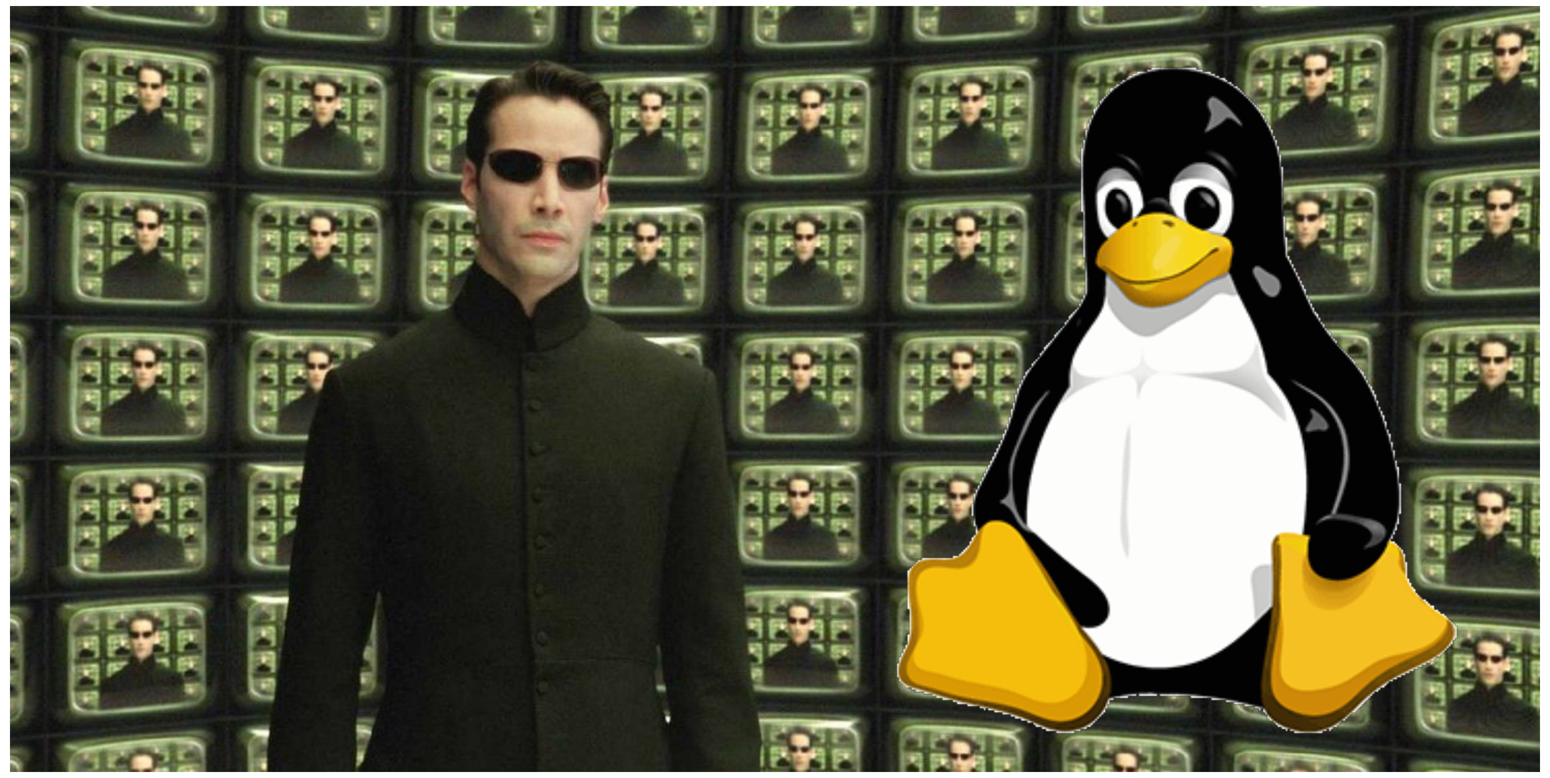
27.07.20 10:47

CSCI 125
Introduction to Computer Science and Programming II
Lecture 7: Data Structure II



Jetic Gū

2020 Summer Semester (S2)

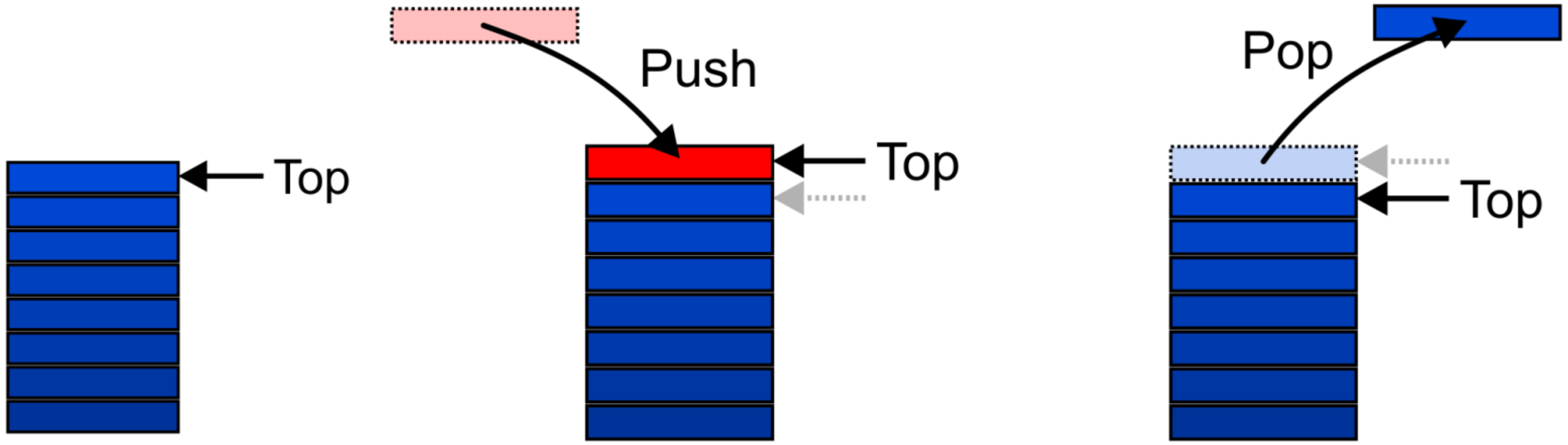
Overview

- Focus: Data Structures
- Architecture: Linux/Unix OS
- Core Ideas:
    1. Stack, Analysis of Stack

Stack

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

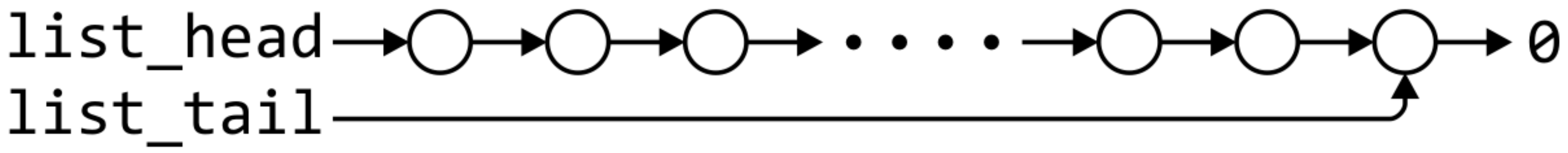
Implementations

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

Concept

Linked-List Implementation

- Operations at the front of a singly linked list are all very efficient: constant steps



	Front/1st	Back/nth
Find	constant	constant
Insert	constant	constant
Erase	constant	up to n

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

Implementation Using MyList

- Recall the definition of `MyList`
    - Class practice 5 (p026)
- We can implement a stack using `MyList`

```
class MyList {
    int value;
    MyList* next;
    MyList* last;
public:
    int length;
    MyList();
    ~MyList();
    void prepend(int val);
    void append(int val);
    int get(int ind);
    int give(int ind, int val);
    int delete(int ind);
};
```

Concept

Stack-as-List Class

- The `Stack` class using a singly linked list has a single private member variable
- 5: `empty`
  returns whether the stack is empty
- 6: `top`
  returns the value of the top element
- 7: `push`
  insert new element at the top
- 8: `pop`
  remote top element

```
1. class Stack {
2.   private:
3.     MyList list;
4.   public:
5.     bool empty();
6.     int top();
7.     void push(int val);
8.     int pop();
9. };
```

Concept

Stack-as-List Class

- Do we need another constructor for `Stack` here?
- Why?
  - Because `list` is declared, the compiler will call the constructor of the `MyList` class when the Stack is constructed

```
class Stack {
  private:
    MyList list;
  public:
    bool empty();
    int top();
    void push(int val);
    int pop();
};
```

Stack-as-List Class

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

```
bool Stack::empty() {
  return list.length==0;
}

void Stack::push(int val) {
  list.prepend(val);
}
```

Stack-as-List Class

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

```
int Stack::top() {
    if ( empty() )
        return -1;
    return list.get(0);
}

int Stack::pop() {
    if ( empty() )
        return -1;
    return list.delete(0);
}
```

Implementation Using MyList

For one-ended arrays, all operations at the back are constant

	Front/1st	Back/nth
Find	constant	constant
Insert	up to n	constant
Erase	up to n	constant

P1
Stack

Stack-as-Array Class

- Implementation using array
  dynamic allocation so more extensible
- 5: `empty`
  returns whether the stack is empty
- 6: `top`
  returns the value of the top element
- 7: `push`
  insert new element at the top
- 8: `pop`
  remote top element

```
1. class Stack {
2.   private:
3.     int *array;
4.   public:
5.     bool empty();
6.     int top();
7.     void push(int val);
8.     int pop();
9. };
```

Concept

P1
Stack

Stack-as-Array Class

- We need additional information, including:
  - The number of objects currently in the stack

    ```
    int stackSize;
    ```

  - The capacity of the array

    ```
    int arrayCapacity;
    ```

Concept

Stack-as-Array Class

- Implementation using array
  dynamic allocation so more extensible
- 5: `empty`
  returns whether the stack is empty
- 6: `top`
  returns the value of the value of the top element
- 7: `push`
  insert new element at the top
- 8: `pop`
  remote top element

```
1. class Stack {
2. private:
3.   int stackSize;
4.   int arrayCapacity;
5.   int *array;
6. public:
7.   Stack( int = 10 );
8.   ~Stack();
9.   bool empty();
10.   int top();
11.   void push( int val );
12.   int pop();
13.};
```

Constructor

- The class is only storing the address of the array
- We must allocate memory for the array and initialise the member variables
- The call to `new int[arrayCapacity]` makes a request to the operating system for `arrayCapacity` member

```
Stack::Stack( int n ) {
    stackSize = 0;
    arrayCapacity =max(1,n);
    array =
     new int[arrayCapacity];
}
```

Destructor

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

```
Stack::~Stack() {
   delete array;
}
```

Concept

Empty

- The stack is empty if the stack size is zero

```
bool Stack::empty() {
  return (stackSize == 0);
}
```

Concept

Top

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

```
int Stack::top() {
  if ( empty() ) {
    return -1;
  }
  return array[stackSize - 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 `stackSize`

```
int Stack::pop() {
  if ( empty() ) {
    return -1;
  }
  --stackSize;
  return array[stackSize];
}
```

Push

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

```
void Stack::push(int val){
 if (stackSize==arrayCapacity)
   return;
 array[stackSize] = val;
 ++stackSize;
}
```

Concept

Others*

- 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();`

* Not required

Array Capacity

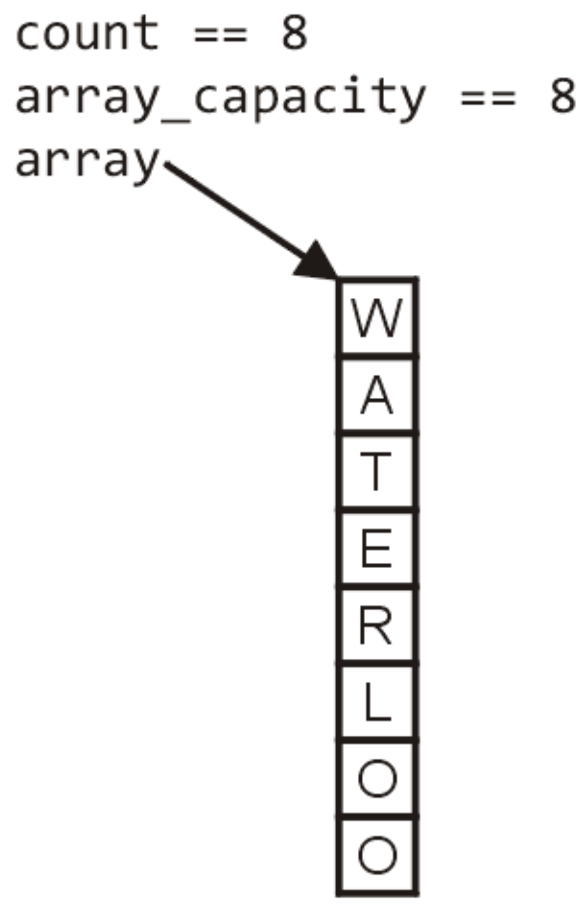
If dynamic memory is available, the best option is to increase the array capacity

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

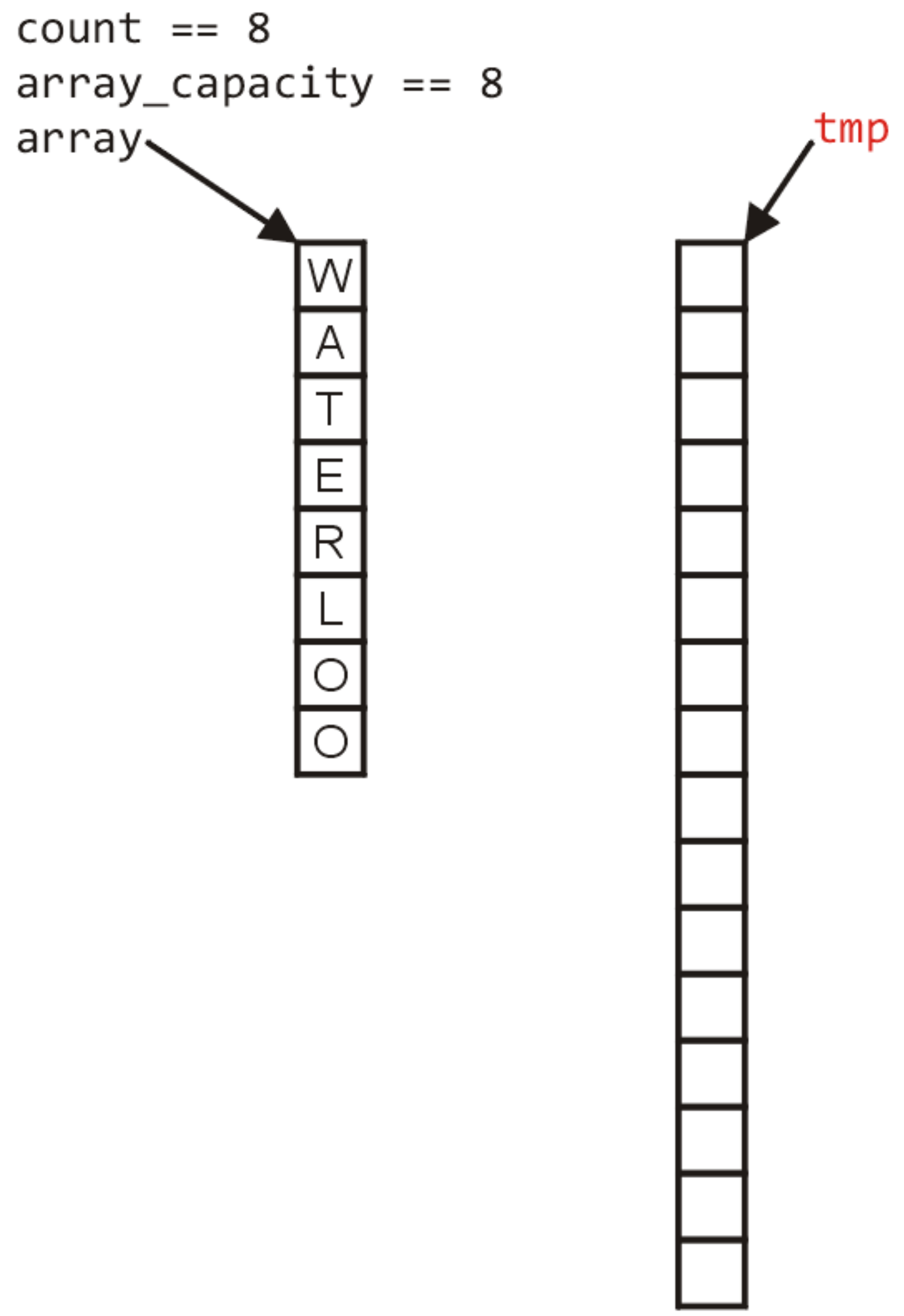
- First, let us visualise what must occur to allocate new memory



Concept

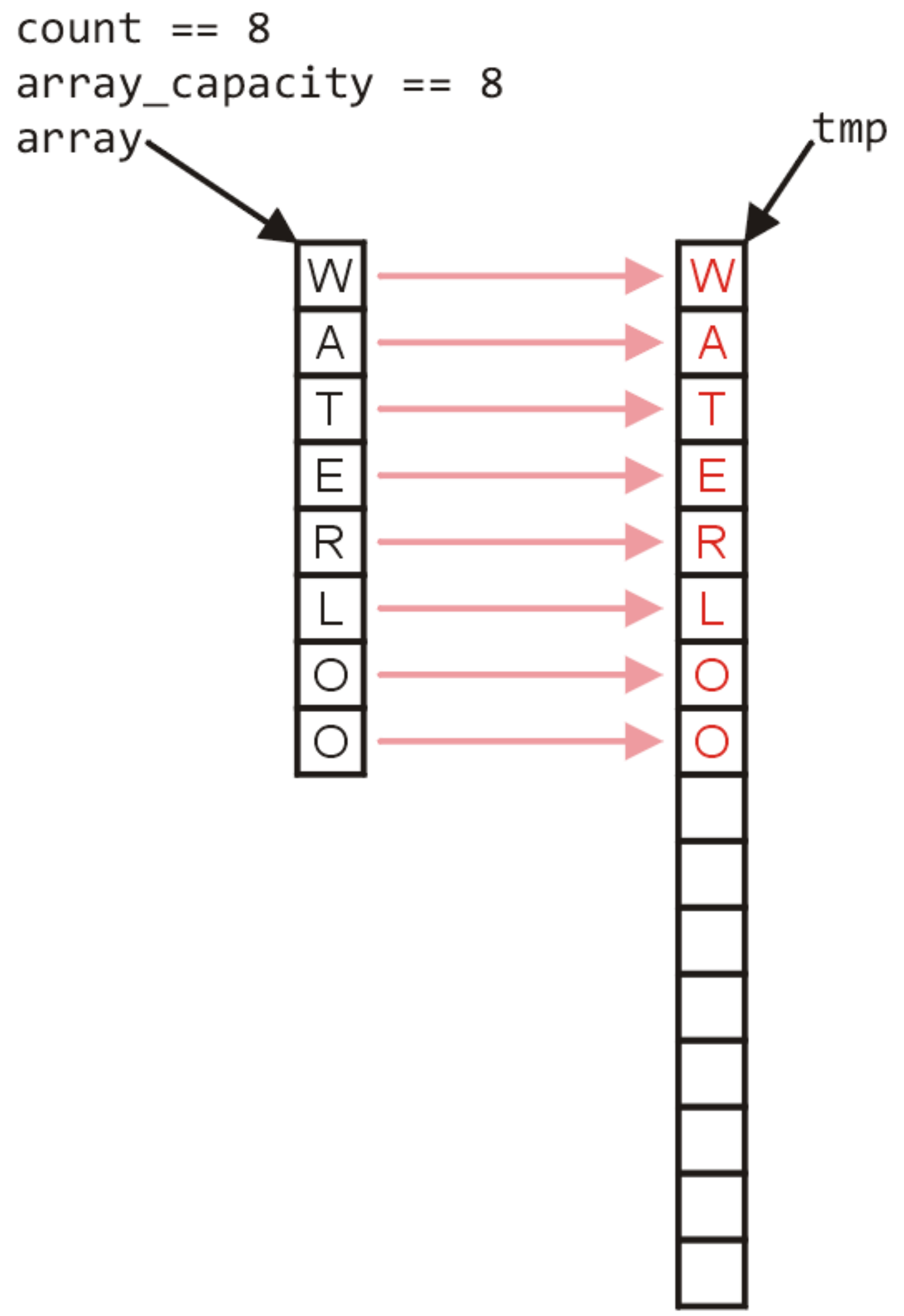
Array Capacity

- First, this requires a call to `new int[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

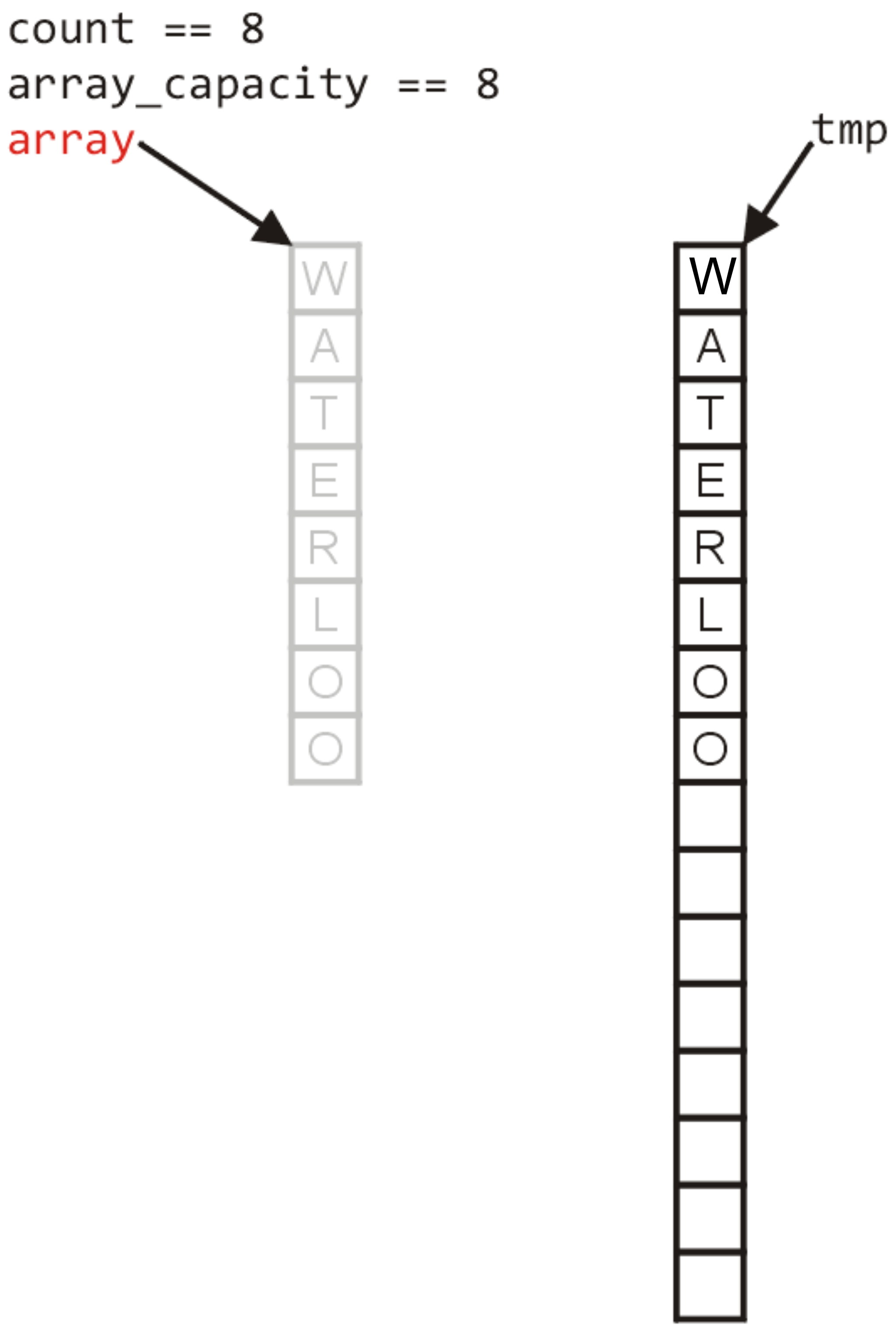
- Next, the values must be copied over



P1
Stack

Array Capacity

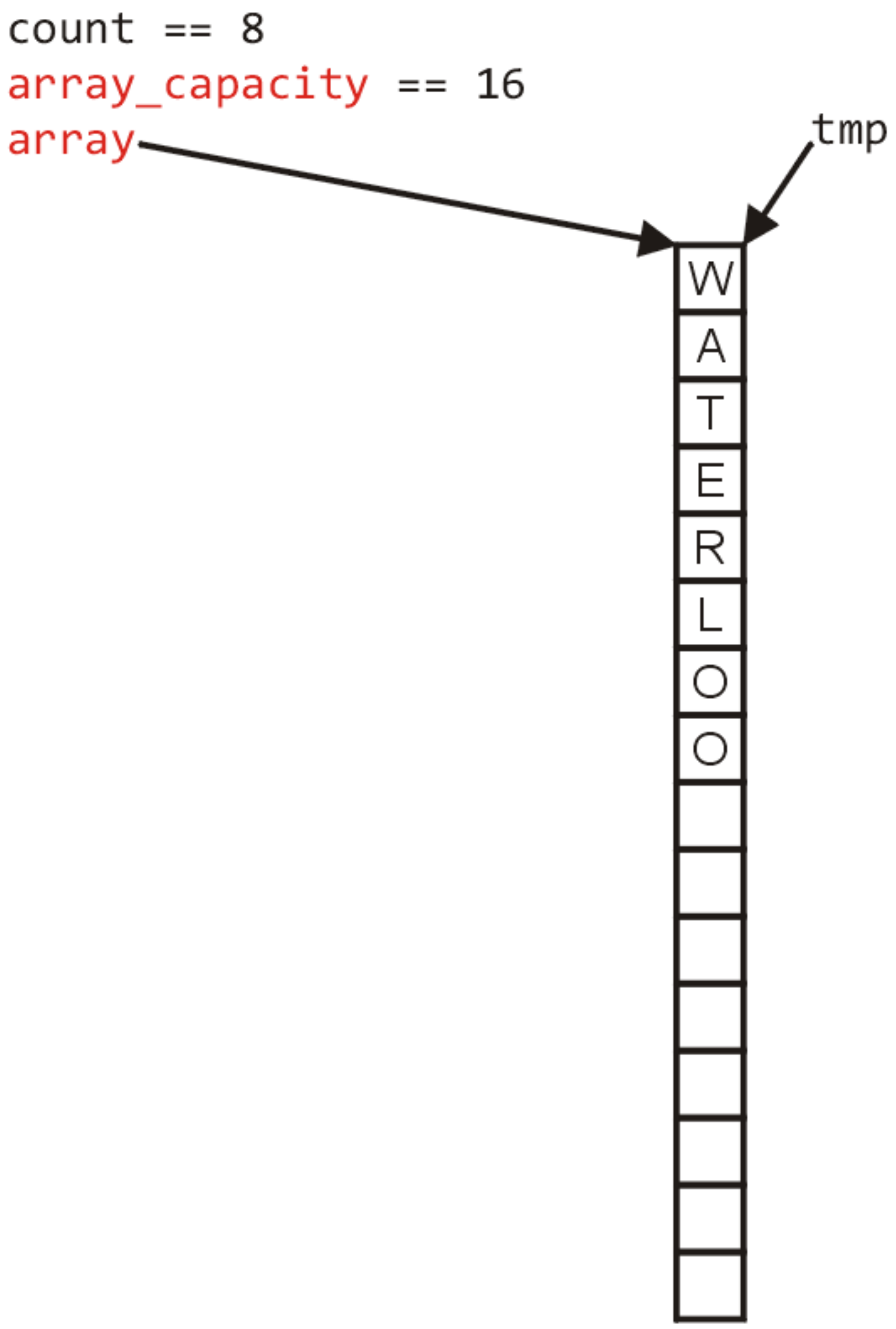
- Deallocate original memory



Concept

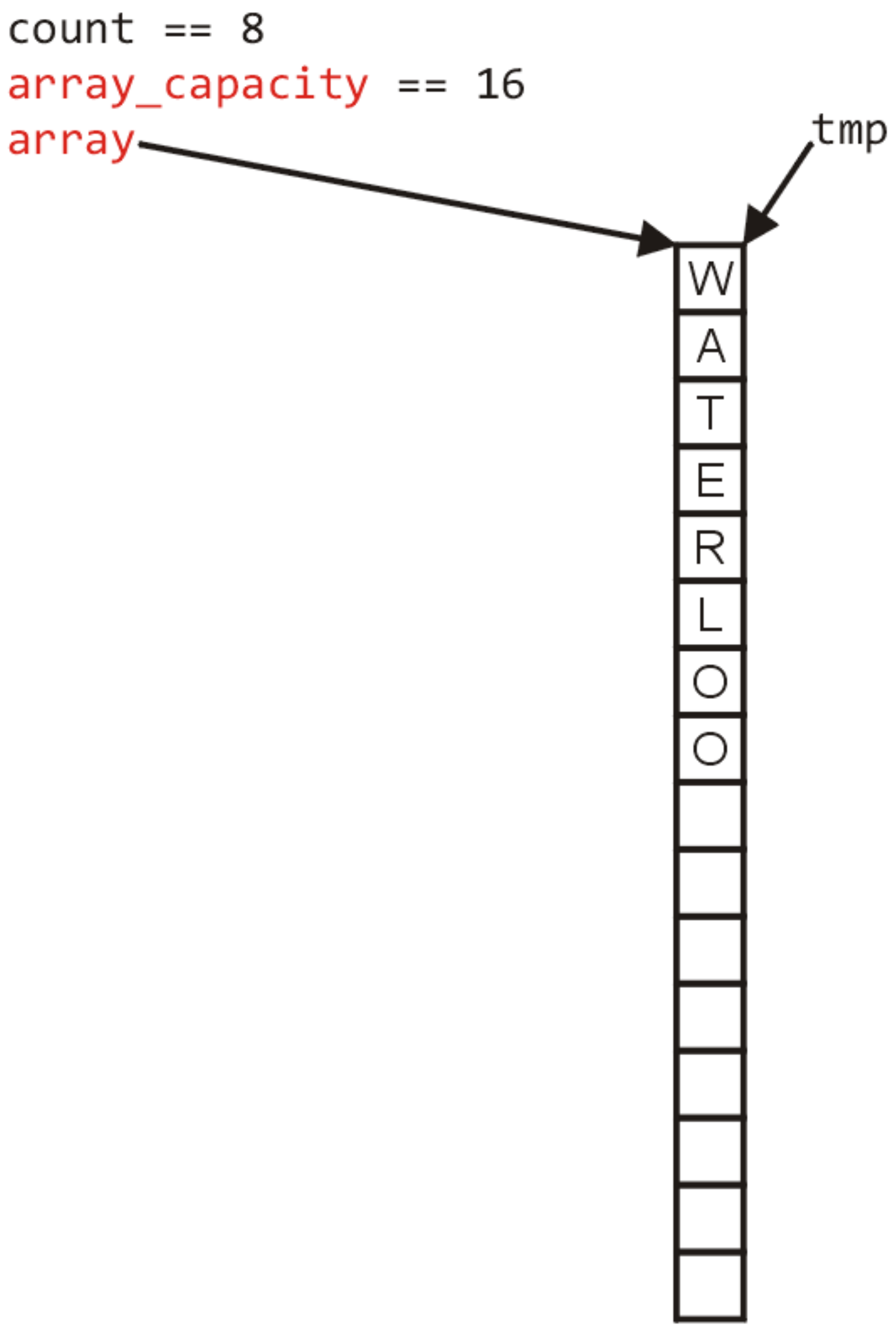
Array Capacity

- Finally, the appropriate member variables must be reassigned



Array Capacity

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



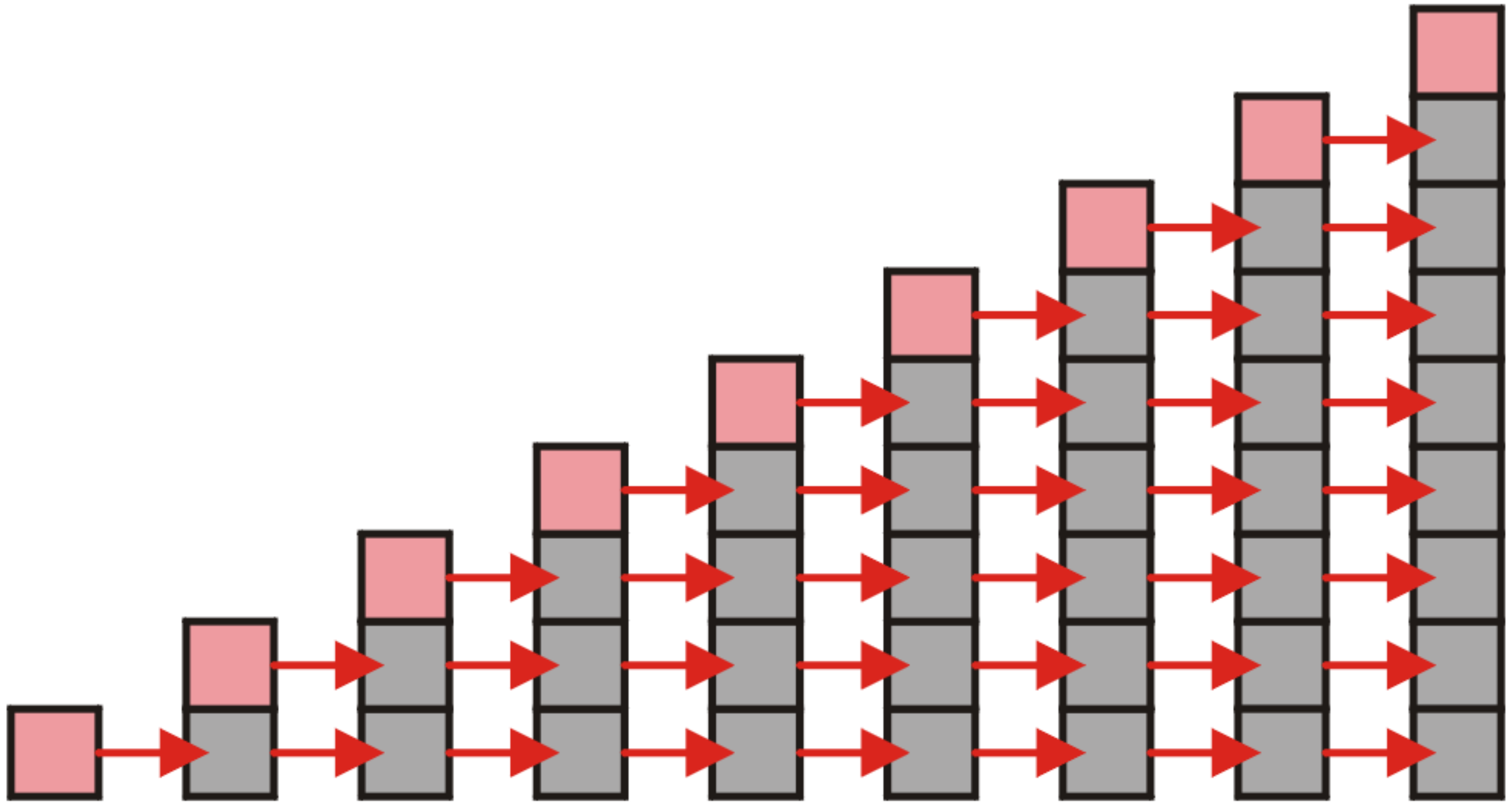
Concept

Array Capacity

- Back to the original question:
  - How much do we change the capacity?
  - Add a constant?
  - Multiply by a constant?
- First, we recognise that any time that we `push` onto a full stack, this requires n copies and the run time is up to N steps
- Therefore, `push` is usually `constant` except when new memory is required

Array Capacity

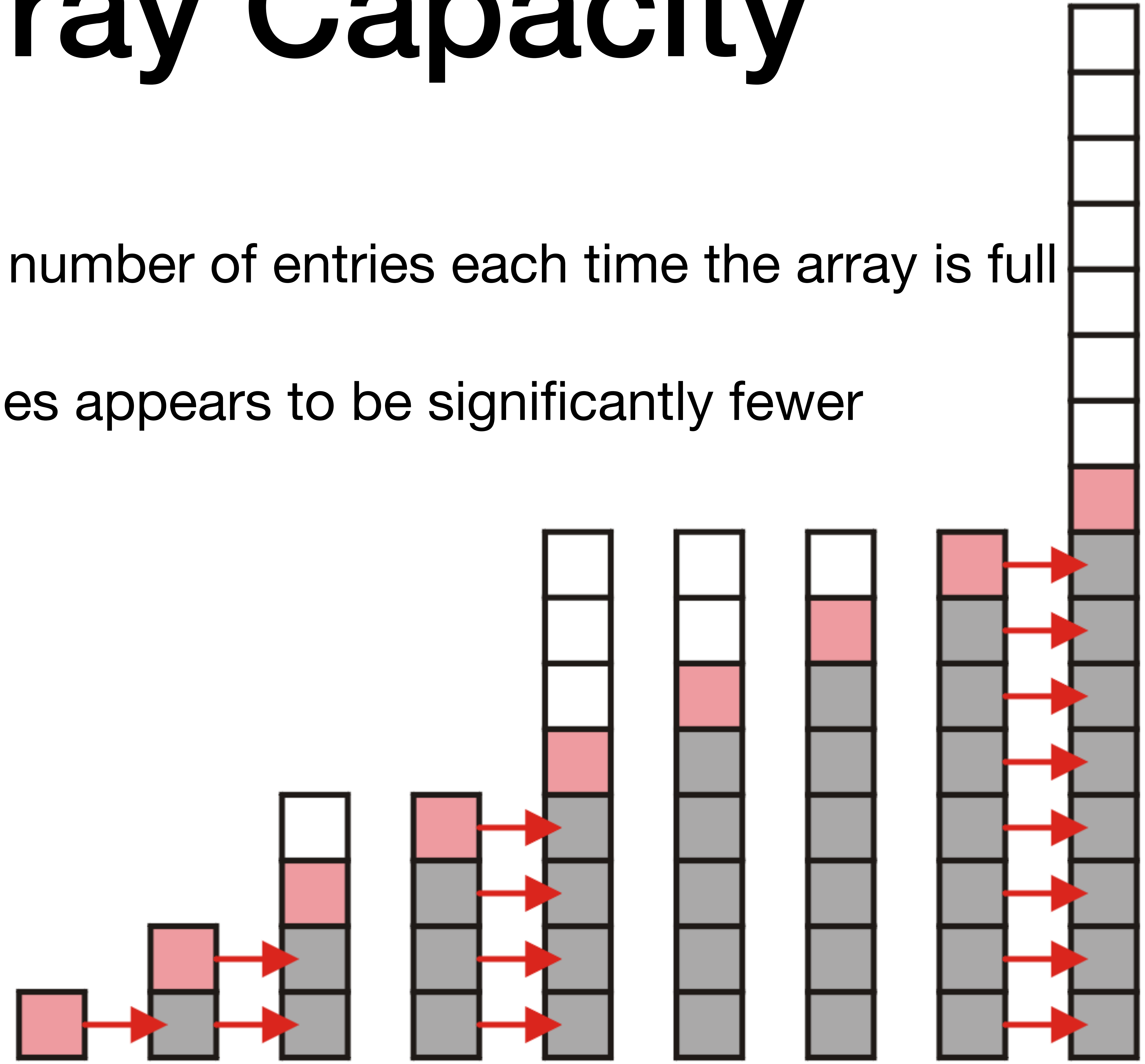
- Consider the case of increasing the capacity by 1 each time the array is full
    - With each insertion when the array is full, this requires all entries to be copied



P1
Stack

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

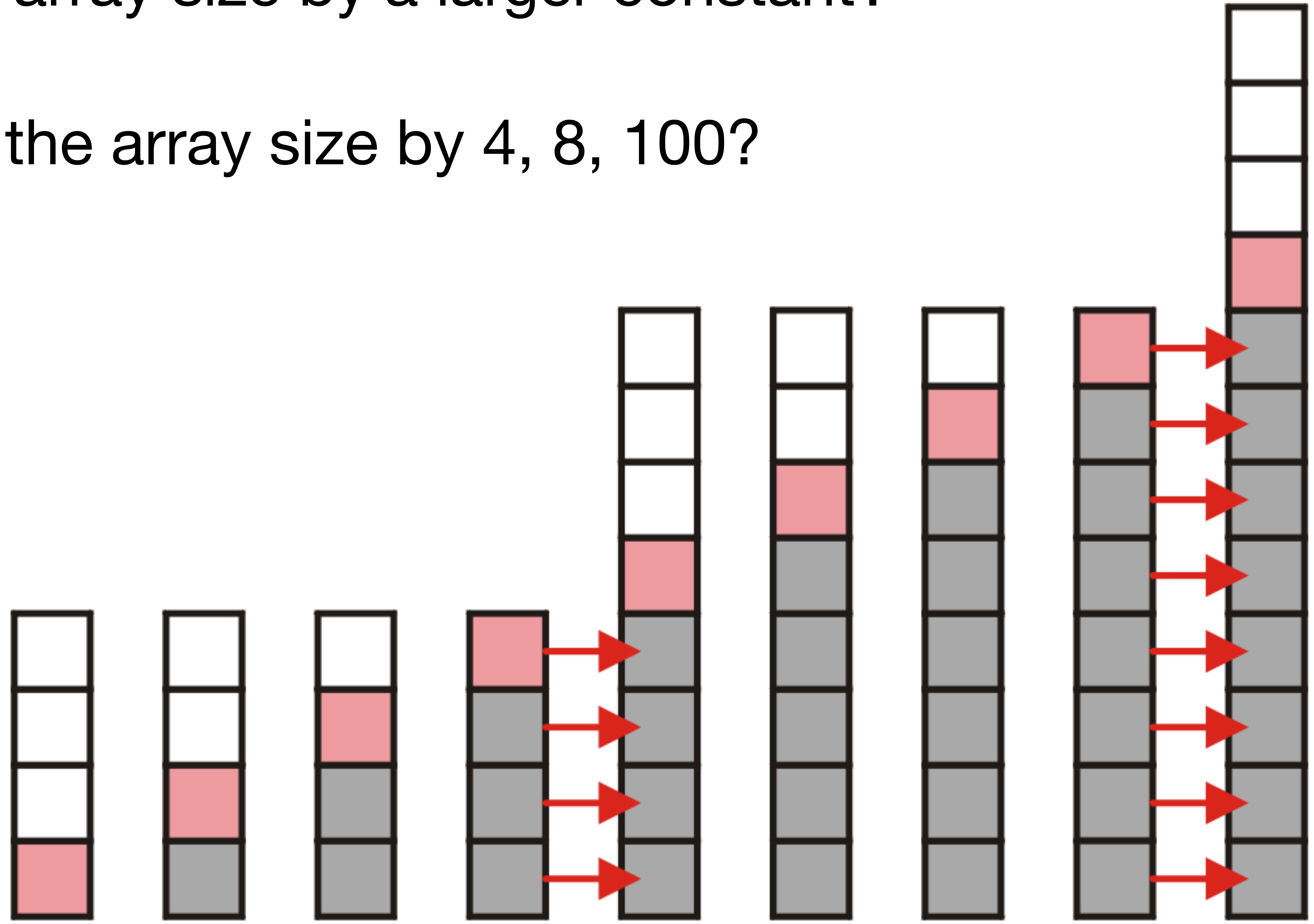


Concept

P1
Stack

Array Capacity

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



Concept

Bracket Matching

Tutorial: Bracket Matching