

```
from(#ccc),to(#ddd))}a.gb1,a.gb2,a.gb3,a.gb4{color:#11c !imp  
0}#gbz{left:0;padding-left:4px}#gbg{right:0;padding-right:5px;  
2d2d;background-image:none;_background-image:none;background-p  
1;filter:alpha(opacity=100);position:absolute;top:0;width:100  
play:none !important}.gbm{position:absolute;z-index:999;top:-  
0 1px 5px #ccc;box-shadow:0 1px 5px #ccc}.gbrtl .gbm{-moz-bo  
0}.gbxms{background-color:#ccc;display:block;position:absolut  
rosoft.Blur(pixelradius=5);*opacity:1;*top:-2px;*left:-5px;*  
ur(pixelradius=5)";opacity:1\0/top:-4px\0/left:-5px;*  
lor:#c0c0c0;display:-moz-inline-block;vertical-align:middle;  
t{zoom:1}gbt
```

03

January 30 - February 5, 2024

Complexity Analysis, Amortization, Strings

Announcements

- Project 1 is due **THIS THURSDAY, FEB 1!!!** 🤯
- Lab 2 autograder (Anyfix) due Monday, **Feb 5**.
- Lab 2 assignment (quiz) due this Monday, **Feb 5**.
- Lab 3 handwritten (Anagram) due this Monday, **Feb 5**.
- Lab 3 AG and quiz due next Monday, Feb 12.
- Project 2 will be released this Tuesday, Feb 6.
- Midterm will be on Tuesday, March 5.
 - No labs on that week!
 - Announcement to come about the exam and SSD accommodations.
 - Keep up the good work!

Agenda

- Recurrence Relations
- Vector Implementation
- Amortization
- Strings
- Handwritten Problem

Recurrence Relations

Recurrence Relations

- A recurrence relation is an equation that is defined in terms of itself.
- Recurrence relations define a sequence of values (think Fibonacci).
- Many algorithms have time complexities which are naturally modeled by recurrence relations
 - i.e. algorithms that create smaller, but similar, sub-problems
 - Binary Search, Fibonacci
- Example:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

The Master Theorem

- The Master Theorem can be used to find the complexities of certain recurrence relations.
- If $T(n)$ is a monotonically increasing function that satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = c_0$$

where $a \geq 1$ and $b > 1$, and $f(n) = \Theta(n^c)$, then:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{if } a > b^c \\ \Theta(n^c \log_2 n), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

The Master Theorem

- Given the Master Theorem:
what is the complexity of
the recurrence relation

$$T(n) = \begin{cases} c_0, & n = 1 \\ 8T(n/2) + 3n, & n > 1. \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{if } a > b^c \\ \Theta(n^c \log_2 n), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

The Master Theorem

- Given the Master Theorem:
what is the complexity of
the recurrence relation

$$T(n) = \begin{cases} c_0, & n = 1 \\ 8T(n/2) + 3n, & n > 1. \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{if } a > b^c \\ \Theta(n^c \log_2 n), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

$$\begin{aligned} a &= 8 & b^c &= 2^1 = 2 & a &> b^c \\ b &= 2 \\ c &= 1 \end{aligned}$$

$$\begin{aligned} \Theta(n^{\log_b a}) &= \Theta(n^{\log_2 8}) \\ &= \Theta(n^3) \end{aligned}$$

The Substitution Method

- If the conditions do not match exactly, we can't use the Master Theorem.
- Suppose we have the following recurrence relation:

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

- Can we use the Master Theorem here?
 - No, we can't, since the recurrence is of the form $T(n-1)$.
- However, we do know the base case: that $T(1) = 1$.
- How do we convert $2T(n-1) + 1$ into a form that contains the base case $T(1)$? If we can, we can just substitute $T(1)$ with 1.
 - We can substitute $T(n-1)$ with $T(n-2)$, then $T(n-3)$, then $T(n-4)$, ..., until we get to $T(1)$!

The Substitution Method

- The steps of the substitution method are as follows:
 - Substitute the formula for $T(n)$ into the recurrence terms on the RHS of the equation until a pattern is found
 - Find a pattern that describes $T(n)$ at the k^{th} step
 - Solve for k such that the base case is present on the RHS. This makes the recurrence easy to solve for in closed form because you know the value of your base case (in the example below, we can replace $T(1)$ with 1).

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

The Substitution Method

- **Step 1:** Substitute the formula for $T(n)$ into the recurrence terms on the RHS of the equation until a pattern is found.

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n - 1) + 1$	$T(n - 1) = 2 * T(n - 2) + 1$
2		
3		
4		
...		
k	???????	

The Substitution Method

- Step 1:** Substitute the formula for $T(n)$ into the recurrence terms on the RHS of the equation until a pattern is found.

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n-1) + 1$	$T(n-1) = 2 * T(n-2) + 1$
2	$T(n) = 2 * (2 * T(n-2) + 1) + 1$ $= 2 * 2 * T(n-2) + 2 + 1$	$T(n-2) = 2 * T(n-3) + 1$
3		
4		
...		
k	???????	

replace $T(n-1)$ with $2T((n-1)-1) + 1$, or $2T(n-2) + 1$

Tip: Don't simplify terms too much – leave them expanded so you can see a pattern!

The Substitution Method

- Step 1:** Substitute the formula for $T(n)$ into the recurrence terms on the RHS of the equation until a pattern is found.

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n-1) + 1$	$T(n-1) = 2 * T(n-2) + 1$
2	$T(n) = 2 * (2 * T(n-2) + 1) + 1$ $= 2 * 2 * T(n-2) + 2 + 1$	$T(n-2) = 2 * T(n-3) + 1$
3	$T(n) = 2 * 2 * (2 * T(n-3) + 1) + 2 + 1$ $= 2 * 2 * 2 * T(n-3) + 2 * 2 + 2 + 1$	$T(n-3) = 2 * T(n-4) + 1$
4		
...		
k	???????	

The Substitution Method

- Step 1:** Substitute the formula for $T(n)$ into the recurrence terms on the RHS of the equation until a pattern is found.

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n-1) + 1$	$T(n-1) = 2 * T(n-2) + 1$
2	$T(n) = 2 * (2 * T(n-2) + 1) + 1$ $= 2 * 2 * T(n-2) + 2 + 1$	$T(n-2) = 2 * T(n-3) + 1$
3	$T(n) = 2 * 2 * (2 * T(n-3) + 1) + 2 + 1$ $= 2 * 2 * 2 * T(n-3) + 2 * 2 + 2 + 1$	$T(n-3) = 2 * T(n-4) + 1$
4	$T(n) = 2 * 2 * 2 * (2 * T(n-4) + 1) + 2 * 2 + 2 + 1$ $= 2 * 2 * 2 * 2 * T(n-4) + 2 * 2 * 2 + 2 * 2 + 2 + 1$	
...		
k	???????	

The Substitution Method

- **Step 2:** Find a pattern that describes $T(n)$ at the k^{th} step.

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n-1) + 1$	$T(n-1) = 2 * T(n-2) + 1$
2	$T(n) = 2 * (2 * T(n-2) + 1) + 1$ $= 2 * 2 * T(n-2) + 2 + 1$	$T(n-2) = 2 * T(n-3) + 1$
3	$T(n) = 2 * 2 * (2 * T(n-3) + 1) + 2 + 1$ $= 2 * 2 * 2 * T(n-3) + 2 * 2 + 2 + 1$	$T(n-3) = 2 * T(n-4) + 1$
4	$T(n) = 2 * 2 * 2 * (2 * T(n-4) + 1) + 2 * 2 + 2 + 1$ $= 2 * 2 * 2 * 2 * T(n-4) + 2 * 2 * 2 + 2 * 2 + 2 + 1$...
...		
k	???????	

The Substitution Method

- **Step 2:** Find a pattern that describes $T(n)$ at the k^{th} step.

$$T(n) \in \begin{cases} 2T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n-1) + 1$	$T(n-1) = 2 * T(n-2) + 1$
2	$T(n) = 2 * (2 * T(n-2) + 1) + 1$ $= 2 * 2 * T(n-2) + 2 + 1$	$T(n-2) = 2 * T(n-3) + 1$
3	$T(n) = 2 * 2 * (2 * T(n-3) + 1) + 2 + 1$ $= 2 * 2 * 2 * T(n-3) + 2 * 2 + 2 + 1$	$T(n-3) = 2 * T(n-4) + 1$
4	$T(n) = 2 * 2 * 2 * (2 * T(n-4) + 1) + 2 * 2 + 2 + 1$ $= 2 * 2 * 2 * 2 * T(n-4) + 2 * 2 * 2 + 2 * 2 + 2 + 1$...
...		
k	$T(n) = 2^k \cdot T(n-k) + \sum_{m=0}^{k-1} 2^m$	

The Substitution Method

- **Step 3:** Solve for k such that the base case is present on the RHS. This makes the recurrence easy to solve for in closed form because you know the value of your base case.

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

$$T(n) \in \begin{cases} 2T(n - 1) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

- What is our base case? $T(1) = 1$
- What is $T(n)$ at the k^{th} step? $T(n) = 2^k T(n - k) + \sum_{m=0}^{k-1} 2^m$
- How can we replace $T(n - k)$ with our base case $T(1)$?
 - $T(n - k) = T(1)$ when $n - k = 1$, or when $k = n - 1$
- Substitute $n - 1$ for k in our equation for $T(n)$ at the k^{th} step:
 - $T(n) = 2^k T(n - k) + \sum_{m=0}^{k-1} 2^m = 2^{n-1} T(1) + \sum_{i=0}^{n-2} 2^i$
 $= 2^{n-1} + 2^{n-1} - 1 = \mathbf{O(2^n)}$

↖
this is our base
case, $T(1) = 1$

Formula to simplify summation

$$\sum_{i=0}^x 2^i = 2^{x+1} - 1$$

$$\begin{aligned} x &= n - 2 \\ (2^{((n-2)+1)} - 1) / (2 - 1) \\ &= 2^{(n-1)} - 1 \end{aligned}$$

you don't need to know this equation!

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$T(n) =$

```
void foo(int n) {  
    if (n == 1)  
        return;  
  
    foo(n / 4);  
  
    int k = n * n;  
  
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);  
  
    for (int i = 0; i < n; ++i)  
        foo(n / 2);  
  
    bar(k);  
} // foo()
```

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$$T(n) = T(n/4)$$

foo calls itself with an input size of $n/4$, so this adds a recurrence term of $T(n/4)$

```
void foo(int n) {  
    if (n == 1)  
        return;  
    foo(n / 4);  
  
    int k = n * n;  
  
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);  
  
    for (int i = 0; i < n; ++i)  
        foo(n / 2);  
  
    bar(k);  
} // foo()
```

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$$T(n) = T(n/4) + n \log n$$

bar is called n times in this loop, so the complexity of this component is $n \log n$

```
void foo(int n) {  
    if (n == 1)  
        return;  
    foo(n / 4);  
  
    int k = n * n;  
  
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);  
  
    for (int i = 0; i < n; ++i)  
        foo(n / 2);  
  
    bar(k);  
} // foo()
```


Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$$T(n) = T(n/4) + n^2 n \log n$$

this $n \log n$ component is called in a loop that runs n^2 times

```
void foo(int n) {  
    if (n == 1)  
        return;  
    foo(n / 4);  
  
    int k = n * n;  
  
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);  
  
    for (int i = 0; i < n; ++i)  
        foo(n / 2);  
  
    bar(k);  
} // foo()
```

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$$T(n) = T(n/4) + n^3 \log n + nT(n/2)$$

foo is called with an input size of $n/2$, but this is done n times in the for loop

```
void foo(int n) {  
    if (n == 1)  
        return;
```

```
    foo(n / 4);
```

```
    int k = n * n;
```

```
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);
```

```
    for (int i = 0; i < n; ++i)  
        foo(n / 2);
```

```
    bar(k);  
} // foo()
```

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$$T(n) = T(n/4) + n^3 \log n + nT(n/2) + \log(n^2)$$

a $\log(n)$ process is called with an input size of n^2 , so the added term is $\log(n^2)$

```
void foo(int n) {  
    if (n == 1)  
        return;  
  
    foo(n / 4);  
  
    int k = n * n;  
  
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);  
  
    for (int i = 0; i < n; ++i)  
        foo(n / 2);  
  
    bar(k);  
} // foo()
```

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume $\text{bar}(n)$ runs in $\log(n)$ time.

$$T(n) = T(n/4) + n^3 \log n + nT(n/2) + 2 \log n$$

using log rules, this can be simplified to $2 \log n$ (bringing down the exponent)

```
void foo(int n) {  
    if (n == 1)  
        return;  
    foo(n / 4);  
  
    int k = n * n;  
  
    for (int i = 0; i < k; ++i)  
        for (int j = 0; j < n; ++j)  
            bar(n);  
  
    for (int i = 0; i < n; ++i)  
        foo(n / 2);  
  
    bar(k);  
} // foo()
```

More Practice

- Given the function below, calculate the recurrence relation.

$T(n) =$

```
int foo(int n){
    if (n <= 0)
        return 1;
    int x = n - 4;
    int s = 0;
    while (x > 4) {
        s += foo(n/4);
        x /= 2;
    }
    for (int i = 0; i < x; ++i) {
        s += foo(i);
    }
    s += foo(n/3) + foo(n - 2);
    s *= 2;
    return s;
}
```

More Practice

- Given the function below, calculate the recurrence relation.

$$T(n) = \log(n - 4)(1 + T(n/4)) + 4T(1) + T(n/3) + T(n - 2) + 1$$

```
int foo(int n){
    if (n <= 0)
        return 1;
    int x = n - 4;
    int s = 0;
    while (x > 4) {
        s += foo(n/4);
        x /= 2;
    }
    for (int i = 0; i < x; ++i) {
        s += foo(i);
    }
    s += foo(n/3) + foo(n - 2);
    s *= 2;
    return s;
}
```


More Practice

- Given the function below, calculate the recurrence relation.

$$T(n) = \log(n)T(n/4) + T(n/3) + T(n - 2) + \log(n)$$

```
int foo(int n){
    if (n <= 0)
        return 1;
    int x = n - 4;
    int s = 0;
    while (x > 4) {
        s += foo(n/4);
        x /= 2;
    }
    for (int i = 0; i < x; ++i) {
        s += foo(i);
    }
    s += foo(n/3) + foo(n - 2);
    s *= 2;
    return s;
}
```

Vectors: Array Resizing

Resize and Reserve

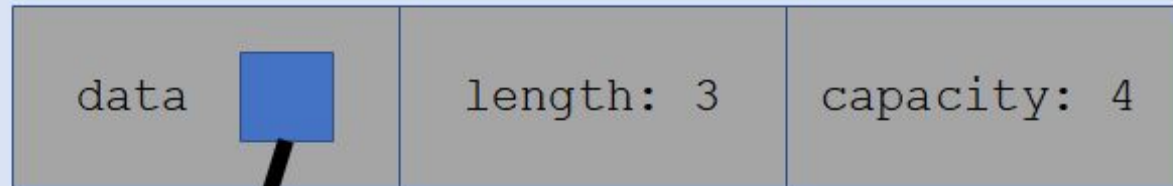
- Vectors have a data pointer, a “size,” and a “capacity.”
- The **capacity** is how much *room* they have.
- The **size** is how many elements they actually contain.
- Recall:
 - `vec.resize(new_size)`
 - changes size (and increases capacity if needed)
 - calling `vec.push_back(x)` after resizing adds x AFTER newly created items
 - `vec.reserve(new_capacity)`
 - does NOT change size, but increases capacity (if needed)
 - calling `vec.push_back(x)` adds x at the same place as before
 - does not do anything if `new_capacity` is smaller than the current capacity

Resize and Reserve

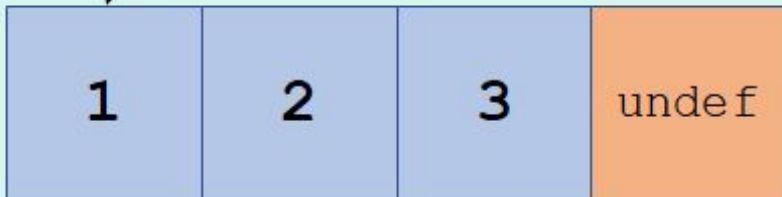
- In lab 1, we introduced the vector ADT, but we told you not to worry about its implementation.
 - We told you to `resize` and `reserve` if you know the size beforehand because “capacity increases are expensive.”
 - But why? What’s the difference between pushing back 100 elements with and without resizing or reserving?
- If you don’t tell the vector what its size should be, the capacity of your vector may be more than what you need - wasted memory!
- Let’s look at how vectors are implemented under the hood.

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```



Stack

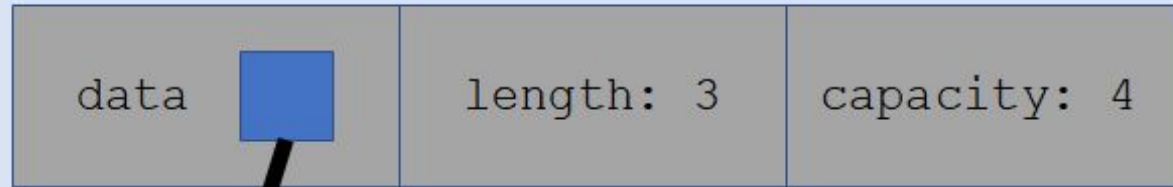


Heap

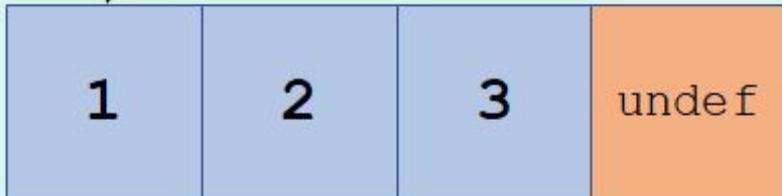
Operations

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```



Stack



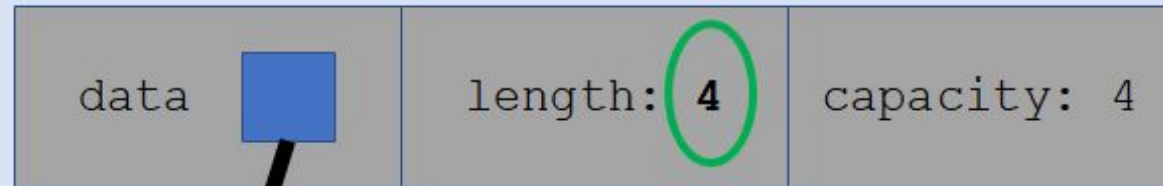
Heap

Operations

```
vec.push_back(6);
```


Resize and Reserve

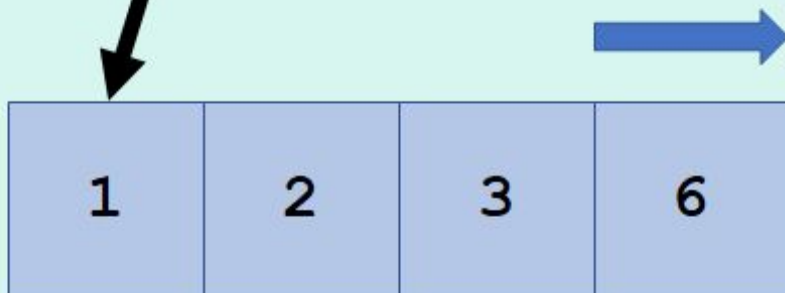
```
vector<int> vec = {1, 2, 3};
```



Stack

Operations

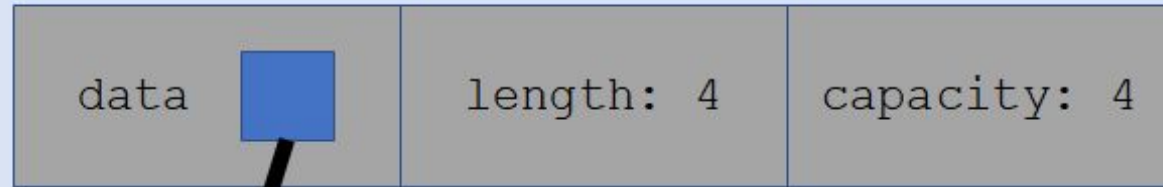
```
vec.push_back(6);
```



Heap

Resize and Reserve

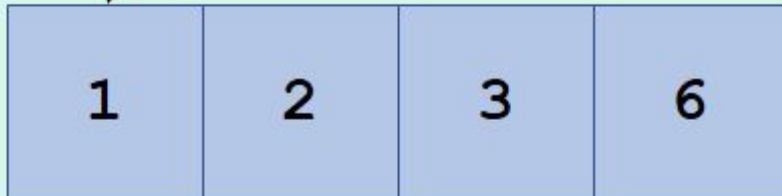
```
vector<int> vec = {1, 2, 3};
```



Stack

Operations

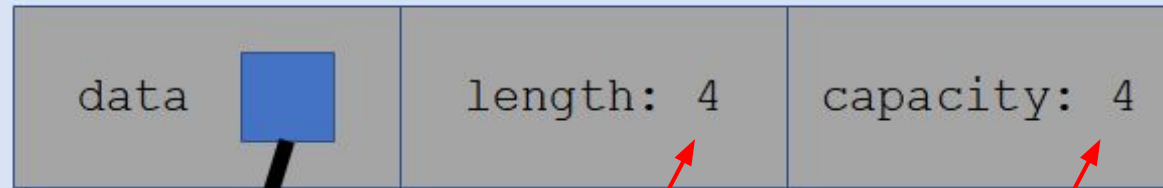
```
vec.push_back(6);  
vec.push_back(5);
```



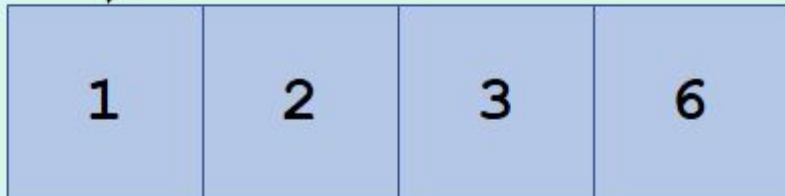
Heap

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```



**length = capacity, so double
in capacity and reallocate!**



Stack

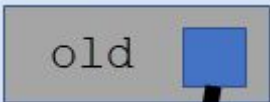
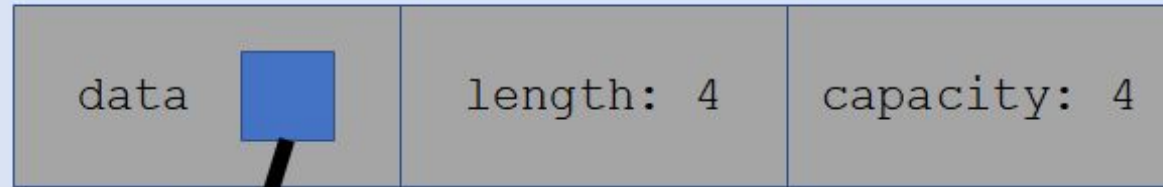
Operations

```
vec.push_back(6);  
vec.push_back(5);
```

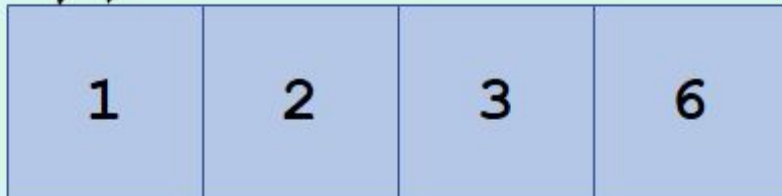
Heap

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```



Stack



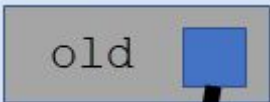
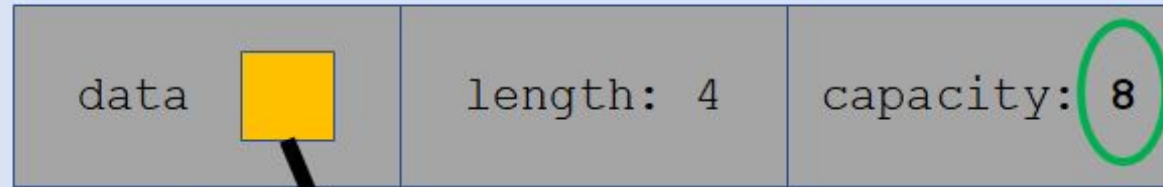
Heap

Operations

```
vec.push_back(6);  
vec.push_back(5);
```

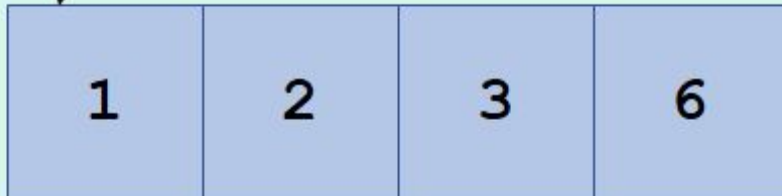
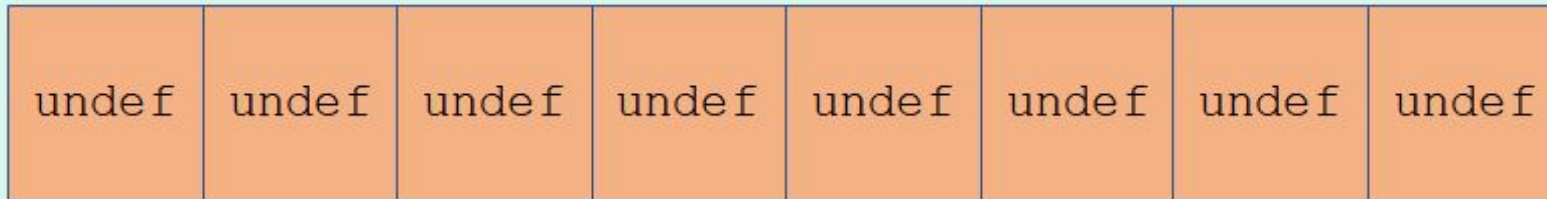
Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```



Stack

New!



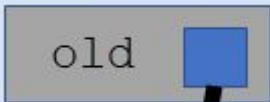
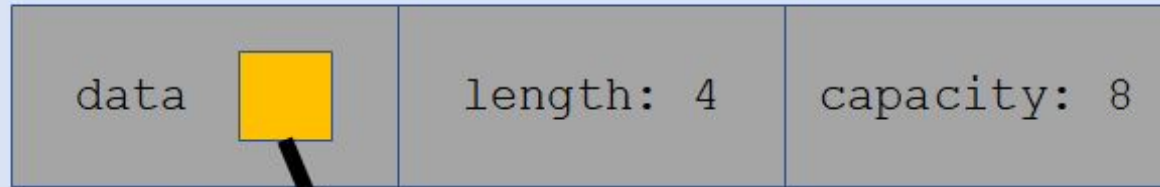
Heap

Operations

```
vec.push_back(6);  
vec.push_back(5);
```

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

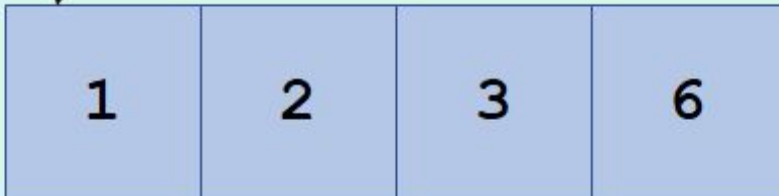
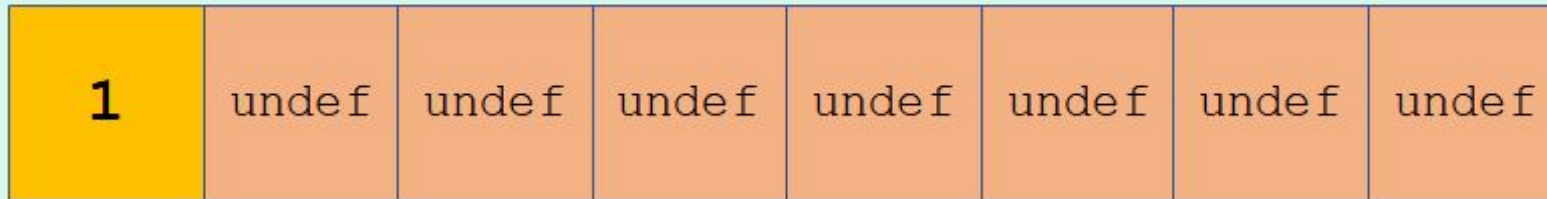


Stack

Operations

```
vec.push_back(6);  
vec.push_back(5);
```

New!



Heap

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

data



length: 4

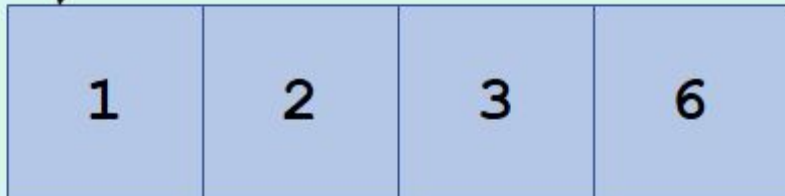
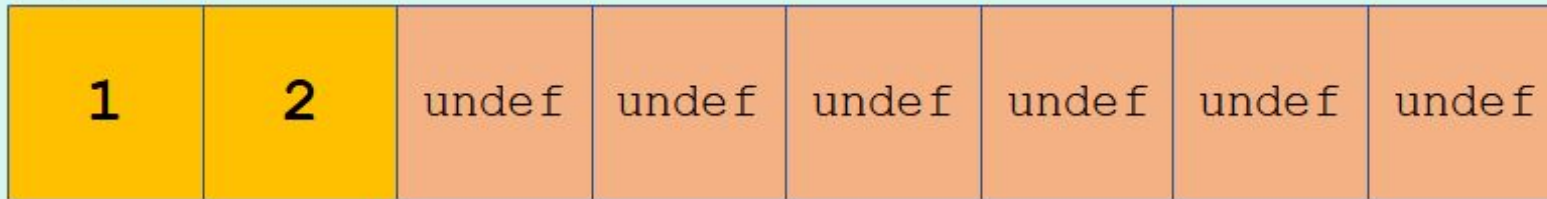
capacity: 8

old



Stack

New!



Operations

```
vec.push_back(6);
```

```
vec.push_back(5);
```

Heap

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

data



length: 4

capacity: 8

old



Stack

New!

1

2

3

undef

undef

undef

undef

undef

1

2

3

6

Heap

Operations

```
vec.push_back(6);
```

```
vec.push_back(5);
```


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

data



length: 4

capacity: 8

old



Stack

New!

1

2

3

6

undef

undef

undef

undef

1

2

3

6

Heap

Operations

```
vec.push_back(6);
```

```
vec.push_back(5);
```

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

data



length: 4

capacity: 8

Stack

Operations

```
vec.push_back(6);
```

```
vec.push_back(5);
```

New!

1

2

3

6

undef

undef

undef

undef

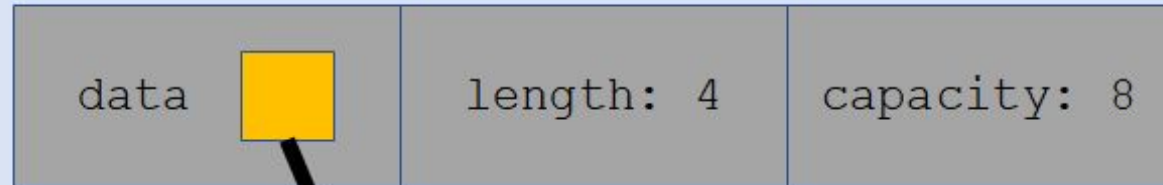
1

6

Heap

Resize and Reserve

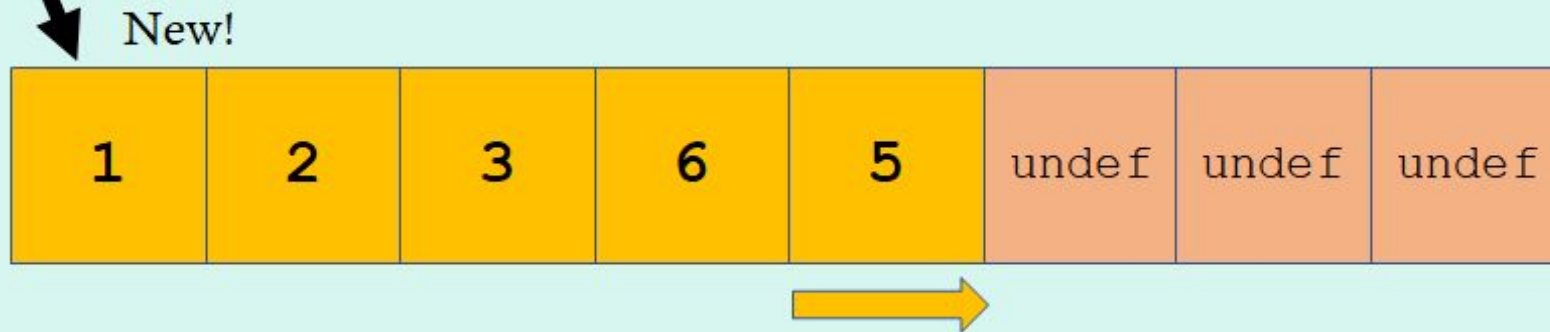
```
vector<int> vec = {1, 2, 3};
```



Stack

Operations

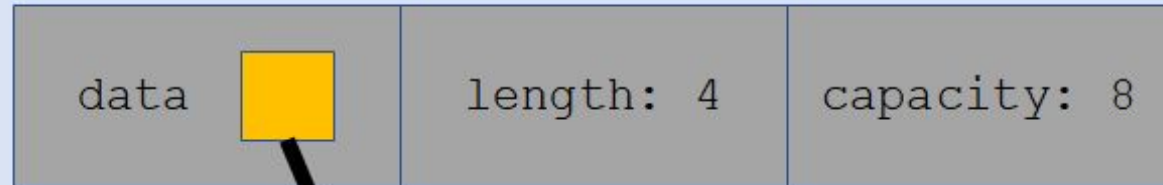
```
vec.push_back(6);  
vec.push_back(5);
```



Heap

Resize and Reserve

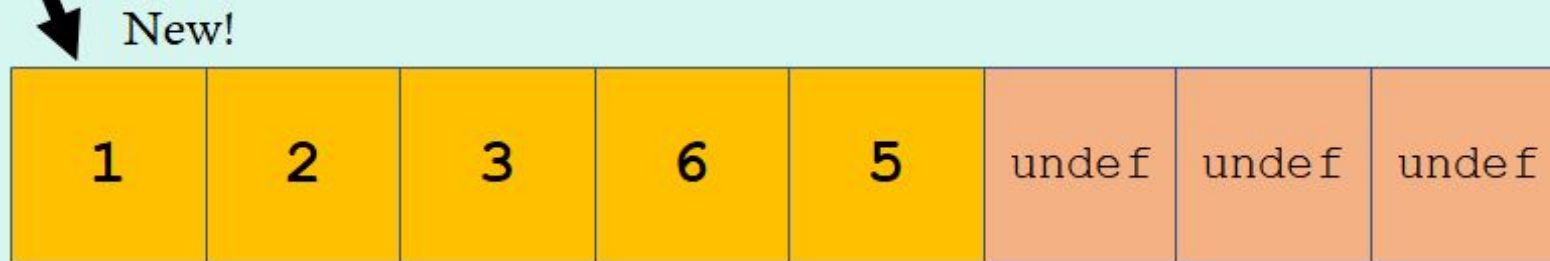
```
vector<int> vec = {1, 2, 3};
```



Stack

Operations

```
vec.push_back(6);  
vec.push_back(5);
```

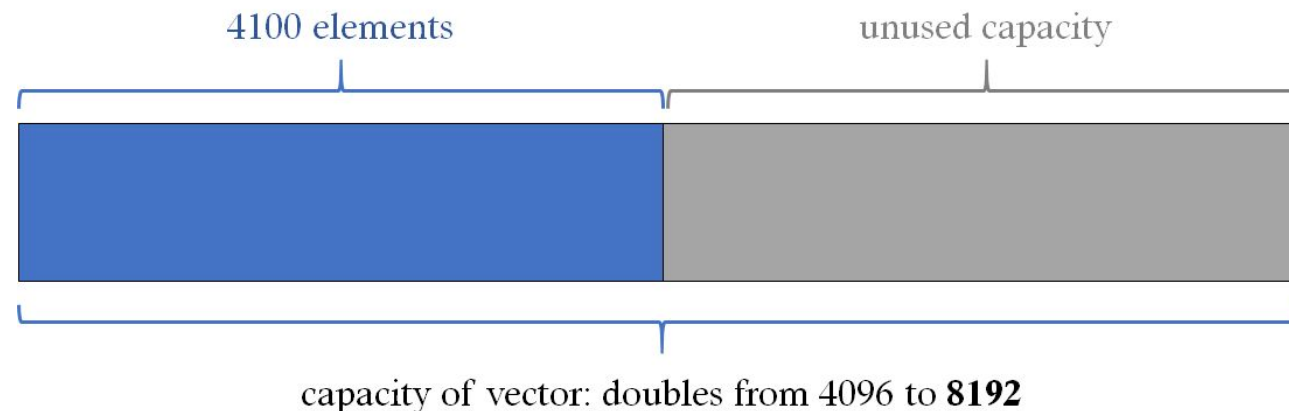


done!

Heap

Resize and Reserve

- Why is this important to know? Suppose you wanted to store 4100 elements in a vector, and you only call `.push_back()` without resizing or reserving. What happens?
 - The vector doubles capacity from 1 to 2 to 4 to 8 to 16 to 32 to 64 to 128 to 256 to 512 to 1024 to 2048 to 4096...
 - When you insert the 4097th element, the vector's capacity doubles to **8192**!
 - You have double the amount of memory that you actually need - wasteful!



Pointer and Iterator Invalidation

- A final comment: when strings, vectors, and other “auto-resizing” containers reallocate their memory, they move their data.
 - When this happens, pointers to their data are **invalidated**!
 - if we store a pointer or an iterator to `vec[10]` and the vector resizes, the pointer is invalidated.
- However, if you do not change a vector’s size or capacity throughout its lifetime, pointers to its elements will not be invalidated.
 - So, if you know the vector’s size in advance, set it using `resize` or `reserve` and then don’t change it!

Amortization

Amortized Complexity

- An alternative way to measure the cost of an operation:
 - We've heard of *worst case* and *average case*
 - **Worst case:** what is the max amount of resources needed for *one* operation?
 - **Average case:** what is the *expected value* of resources needed?
 - Think sorting with random input
 - Amortized analysis is totally different!
 - AMORTIZED ANALYSIS != AVERAGE ANALYSIS
- **Amortized cost:**
 - How much does a *collection* of operations contribute to the cost of the program?
 - Used when the worst case is too pessimistic

Amortized Complexity

3/31

Total Cost: --

- Let's look at an example:
- What is the daily cost of living in an apartment?
 - You pay \$600 rent on the first day of each month
 - There are 30 days in a month

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Amortized Complexity

4/1

Total Cost: \$600

- Let's look at an example:
- What is the daily cost of living in an apartment?
 - You pay \$600 rent on the first day of each month
 - There are 30 days in a month

\$600 rent is paid on the first of the month

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
		2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Amortized Complexity

4/6

Total Cost: \$600

- Let's look at an example:
- What is the daily cost of living in an apartment?
 - You pay \$600 rent on the first day of each month
 - There are 30 days in a month

\$600 rent is paid on the first of the month

For the remaining 29 days, you don't have to pay anything

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Amortized Complexity

4/13

Total Cost: \$600

- Let's look at an example:
- What is the daily cost of living in an apartment?
 - You pay \$600 rent on the first day of each month
 - There are 30 days in a month

\$600 rent is paid on the first of the month

For the remaining 29 days, you don't have to pay anything

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Amortized Complexity

4/20

Total Cost: \$600

- Let's look at an example:
- What is the daily cost of living in an apartment?
 - You pay \$600 rent on the first day of each month
 - There are 30 days in a month

\$600 rent is paid on the first of the month

For the remaining 29 days, you don't have to pay anything

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
21	22	23	24	25	26	27
28	29	30				

Amortized Complexity

4/27

Total Cost: \$600

- Let's look at an example:
- What is the daily cost of living in an apartment?
 - You pay \$600 rent on the first day of each month
 - There are 30 days in a month

\$600 rent is paid on the first of the month

For the remaining 29 days, you don't have to pay anything

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
28	29	30				

Total Cost: **still \$600**

- For the remaining 29 days, you don't have to pay anything**

APRIL 2019						
SUN	MON	TUE	WED	THU	FRI	SAT

Total Cost: **still \$600**

- # APRIL 2019

[illegible]

Amortized Complexity

- Amortized complexity gives us a better approximation of the complexity of an operation if the worst case is too pessimistic.
 - Consider the example of pushing an element into a vector:
 - the worst case time complexity of pushing back an element into a vector is $\Theta(n)$ since we might have to reallocate and copy over the entire vector...
 - However, reallocation doesn't happen often! It would be a bit extreme to treat `.push_back()` as an $\Theta(n)$ process.
 - Let's look at this in more detail.

Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



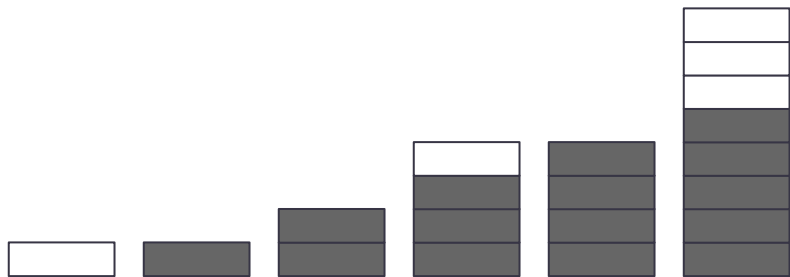
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



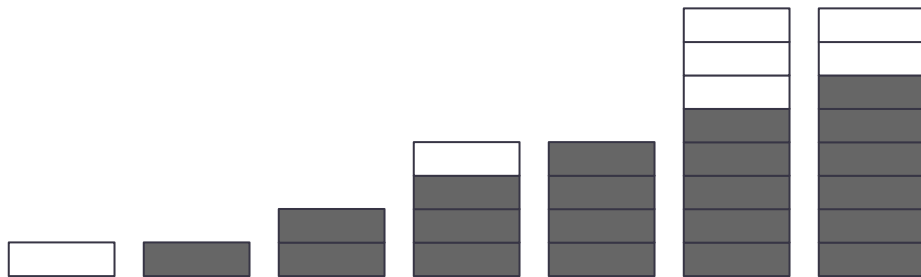
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



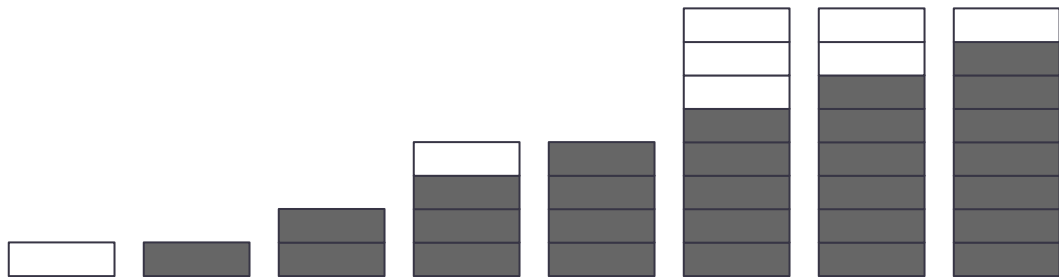
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



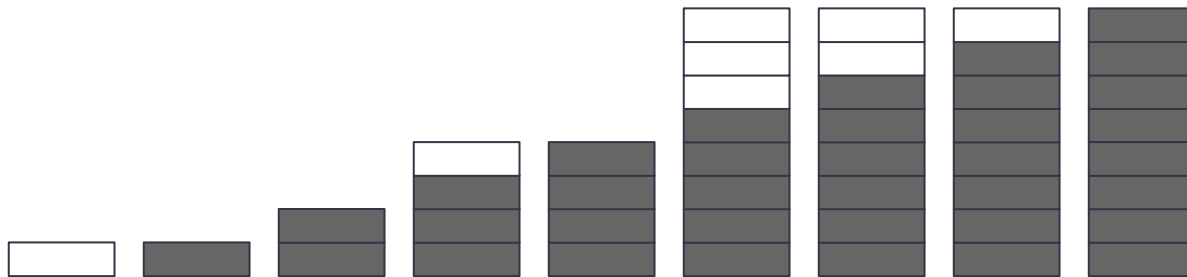
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



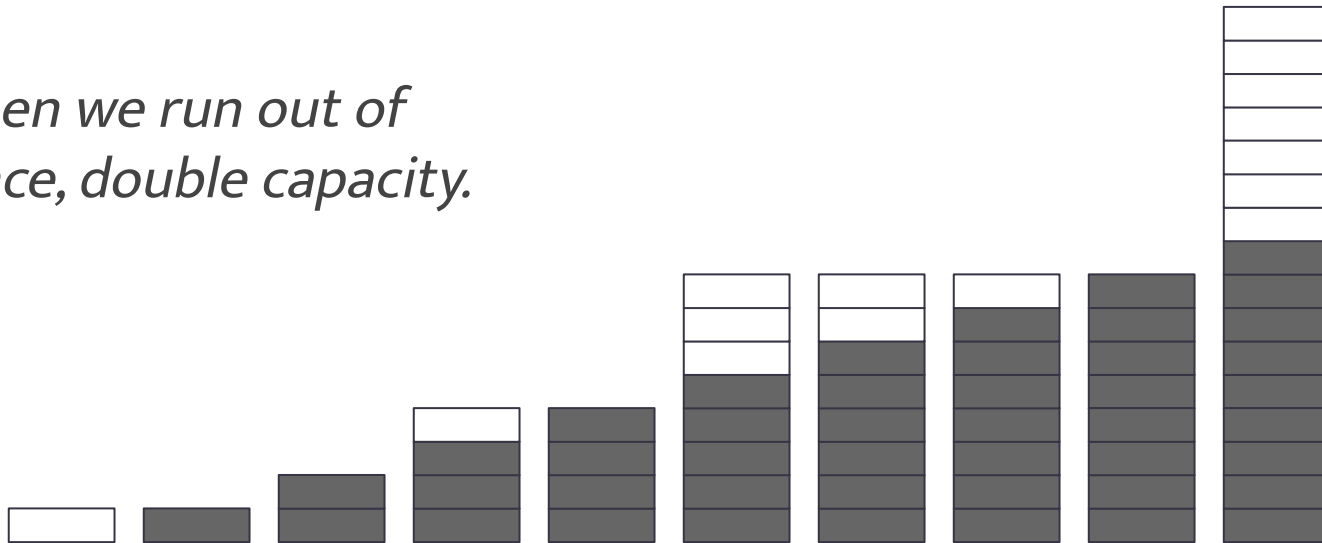
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



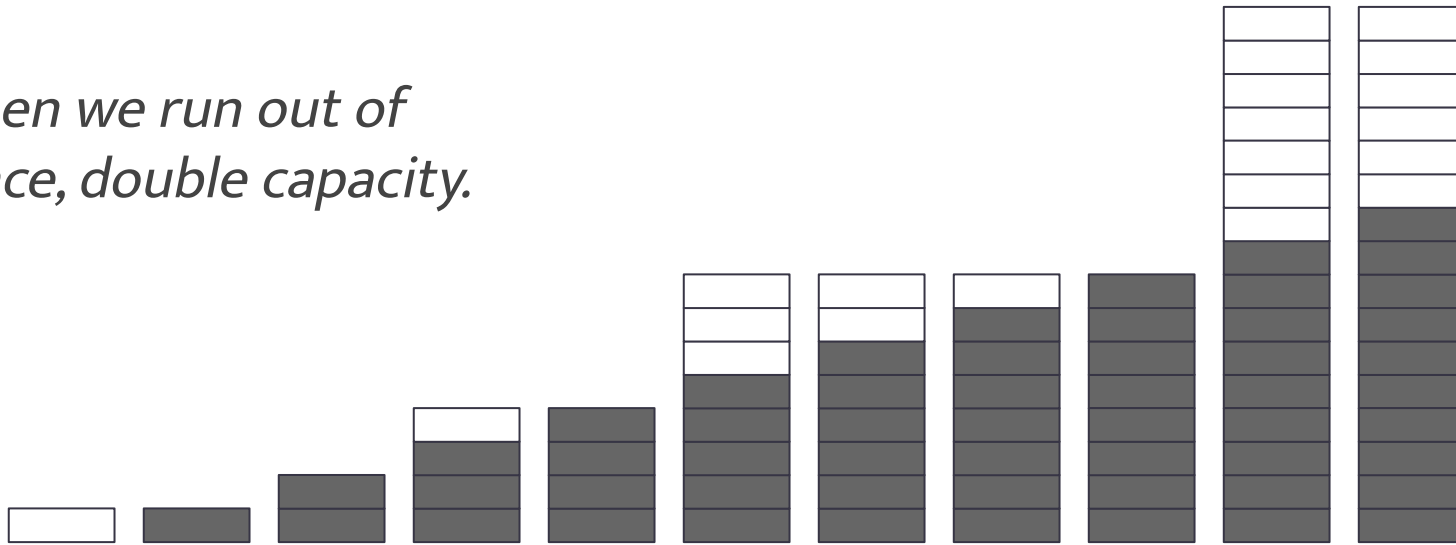
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



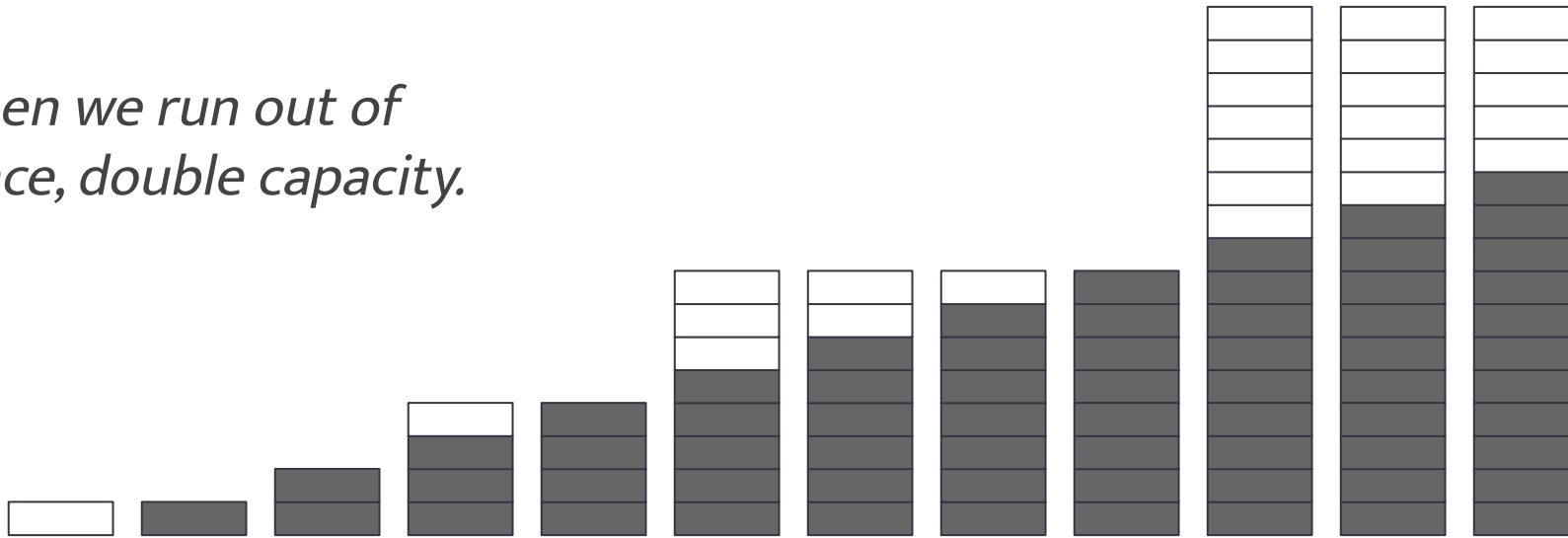
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



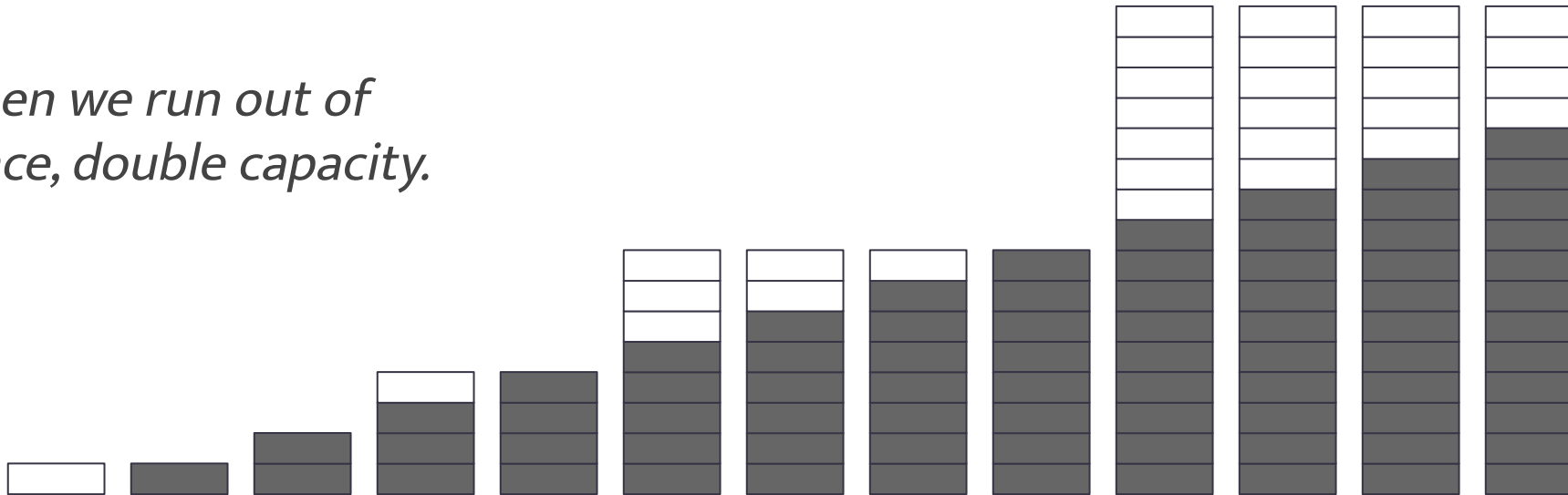
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



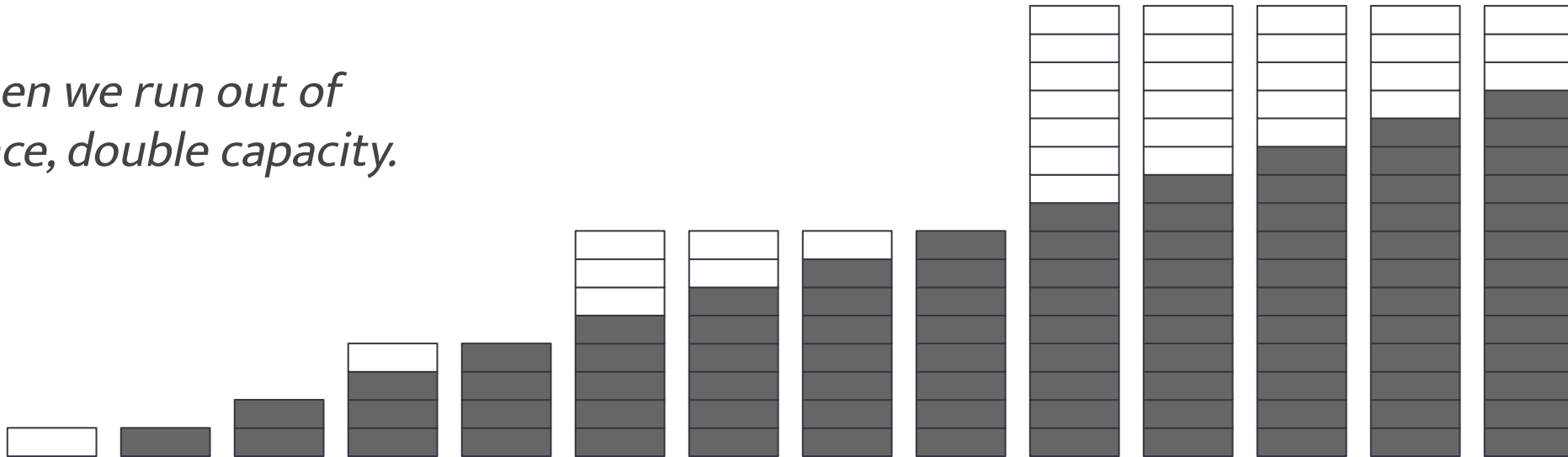
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



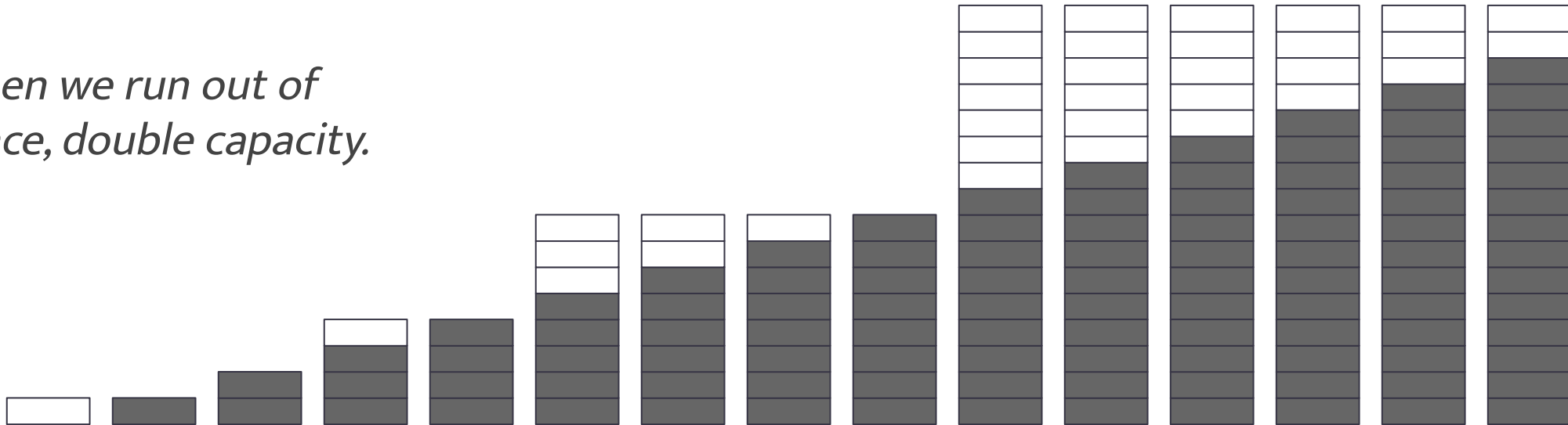
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



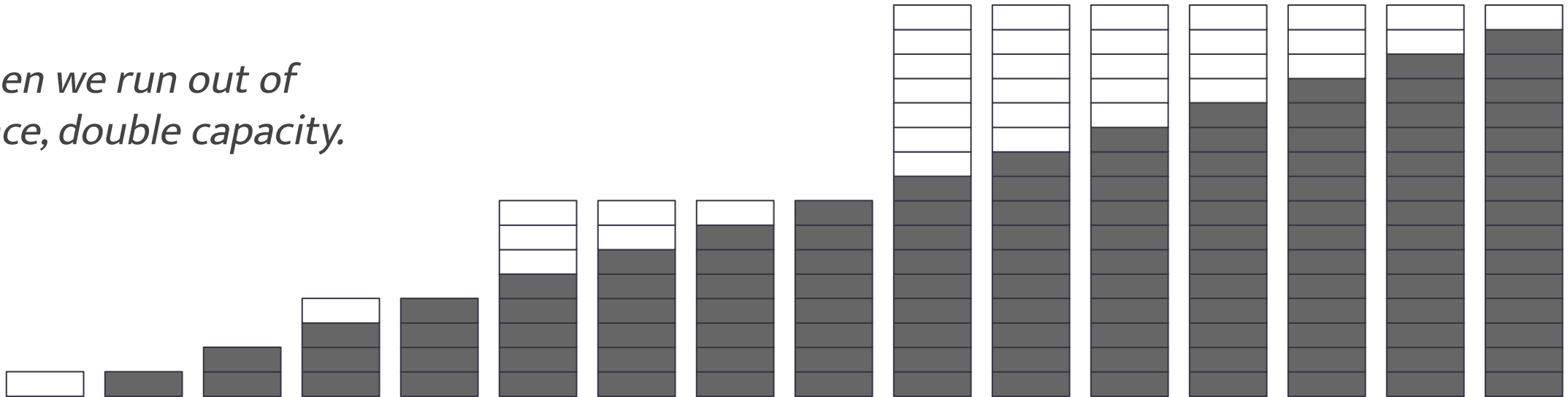
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



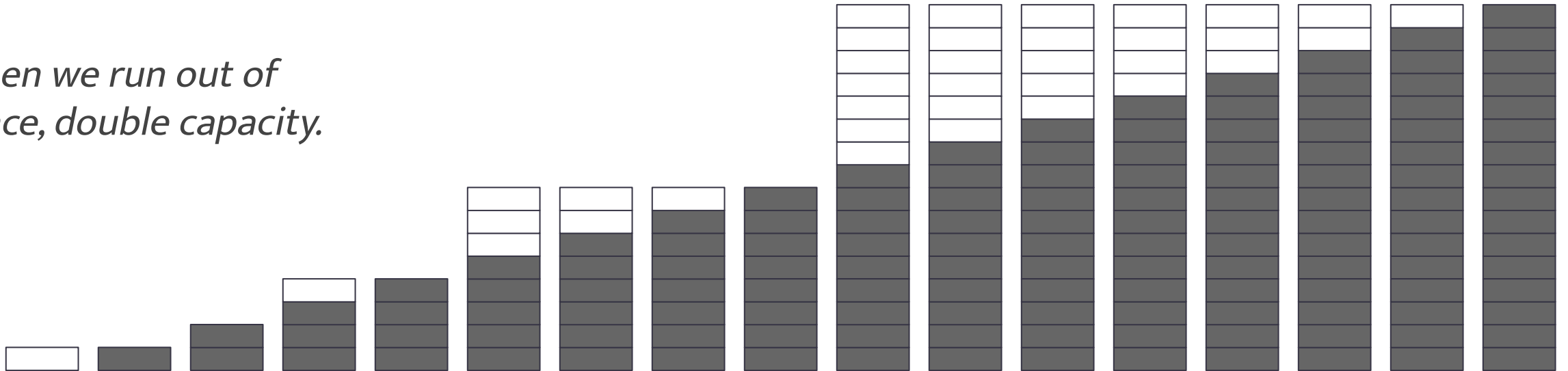
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.



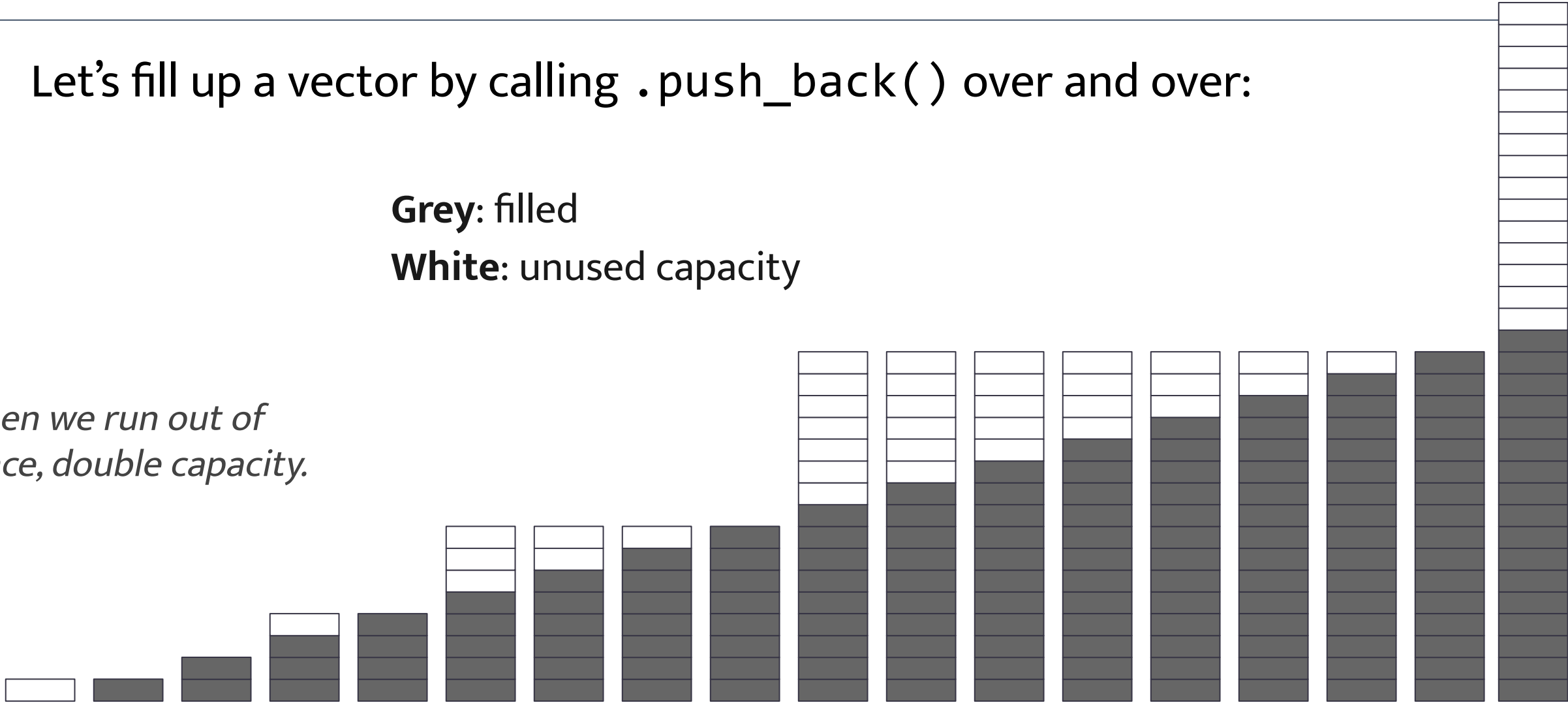
Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

White: unused capacity

When we run out of space, double capacity.

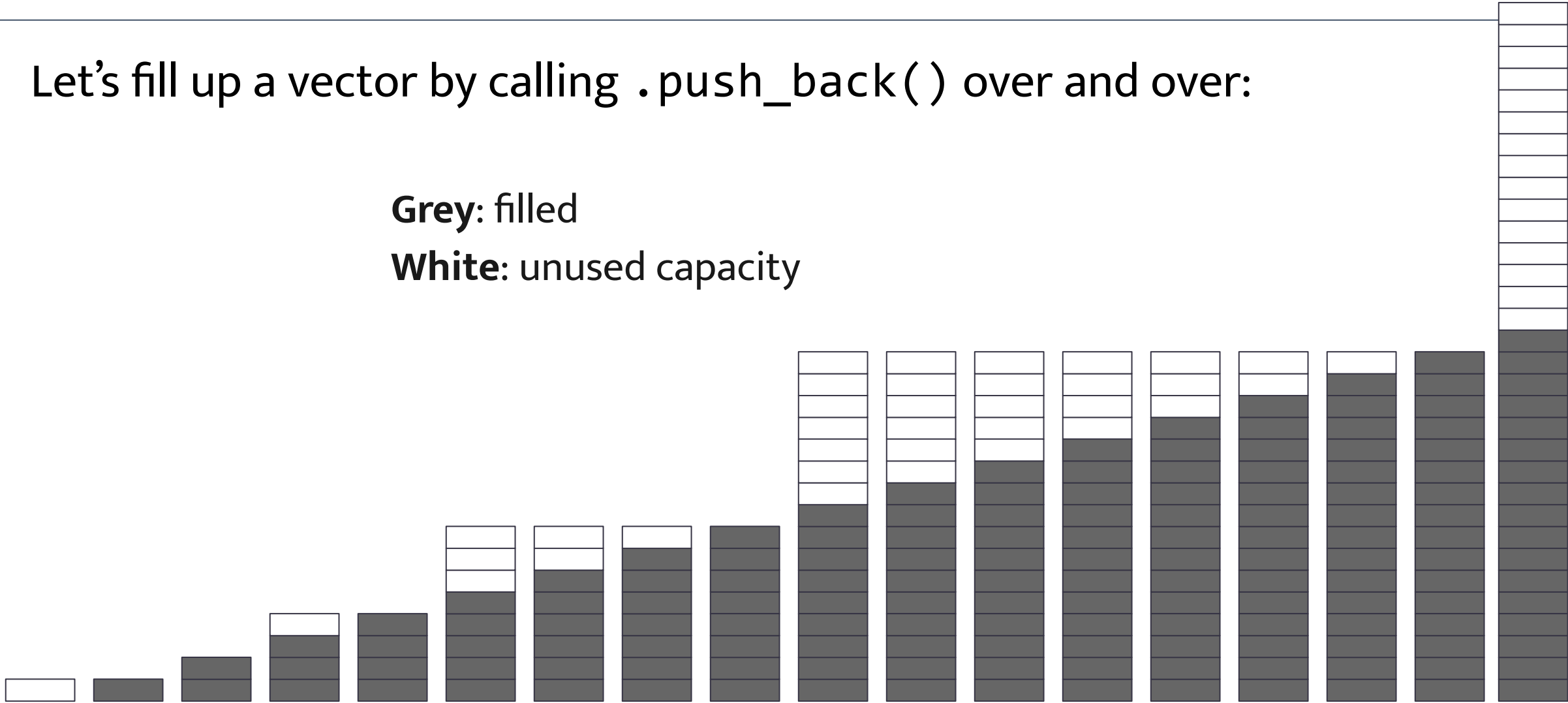


Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Grey: filled

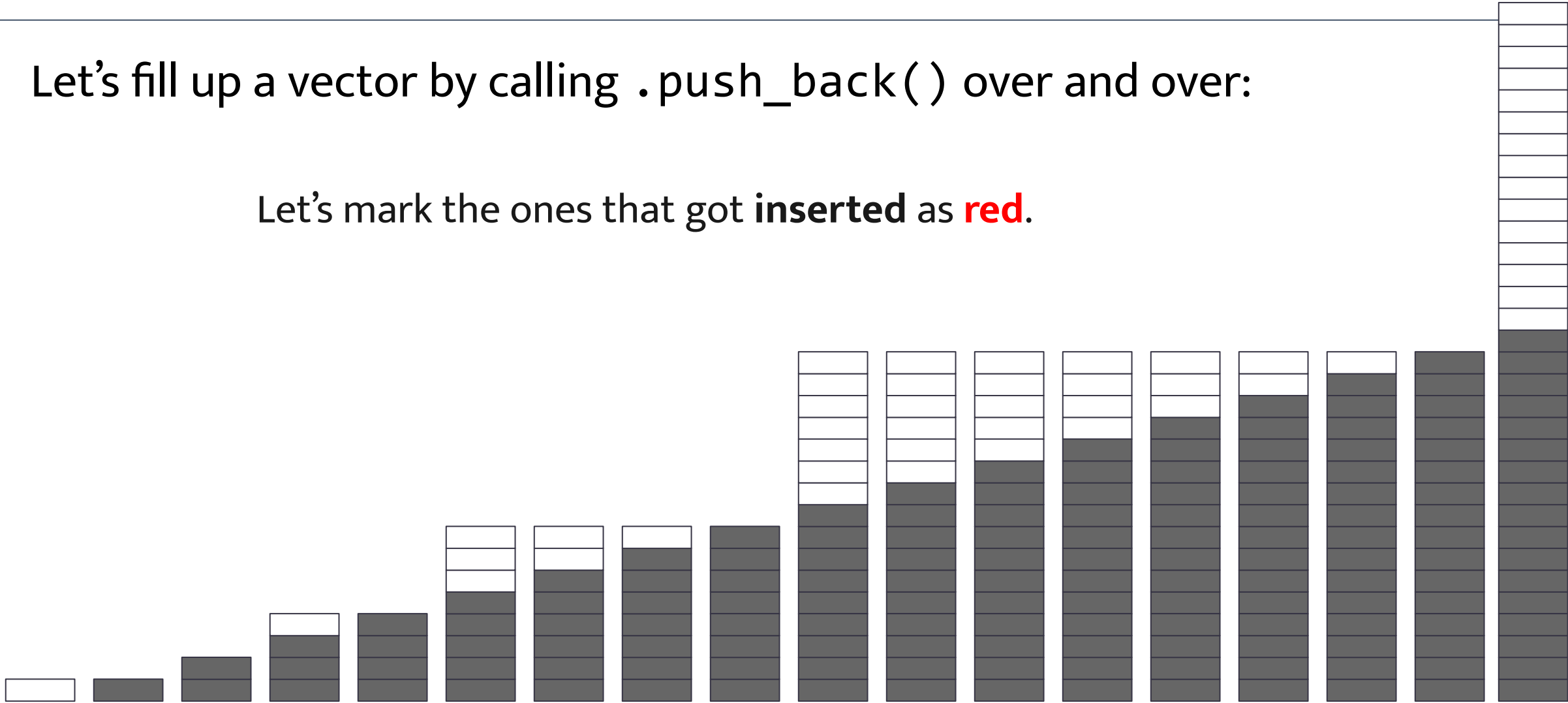
White: unused capacity



Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

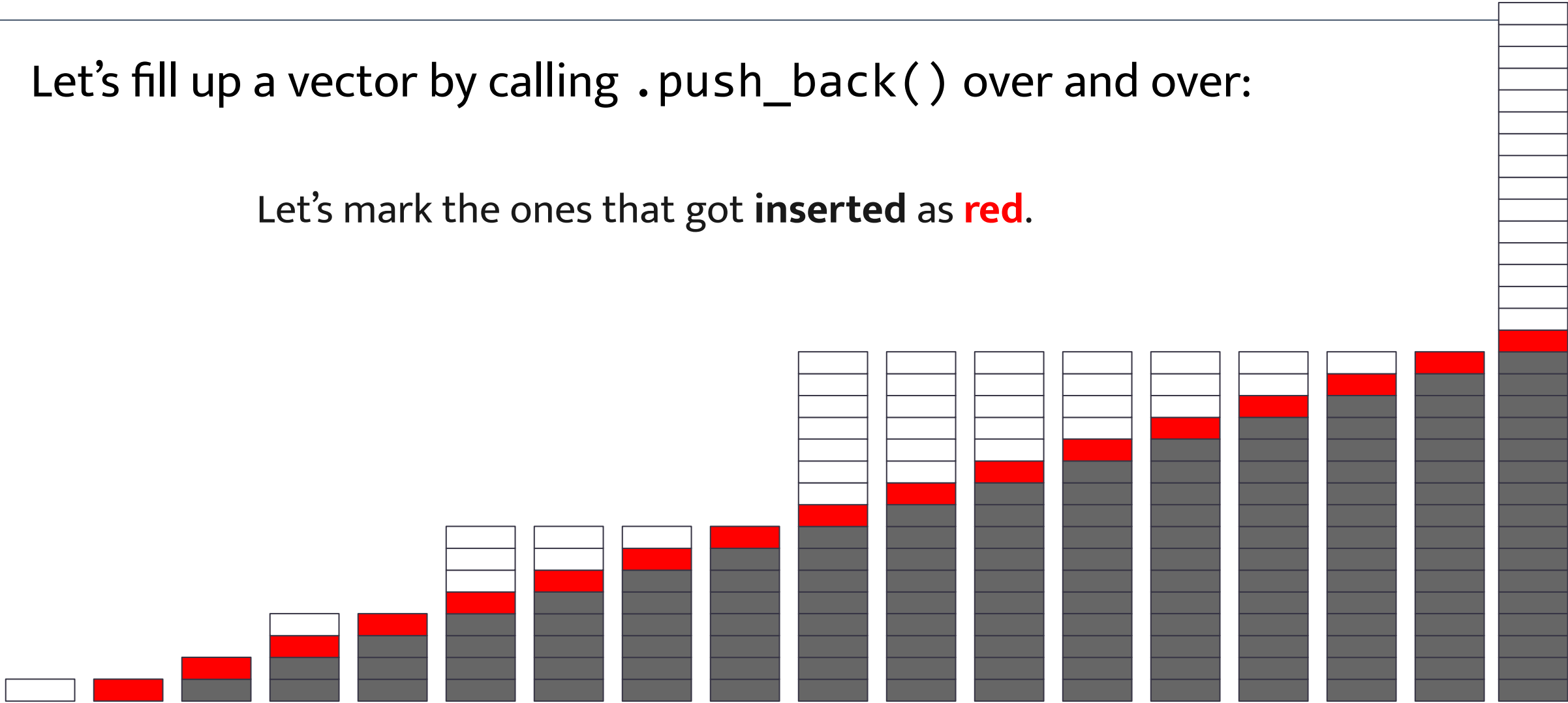
Let's mark the ones that got **inserted** as **red**.



Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

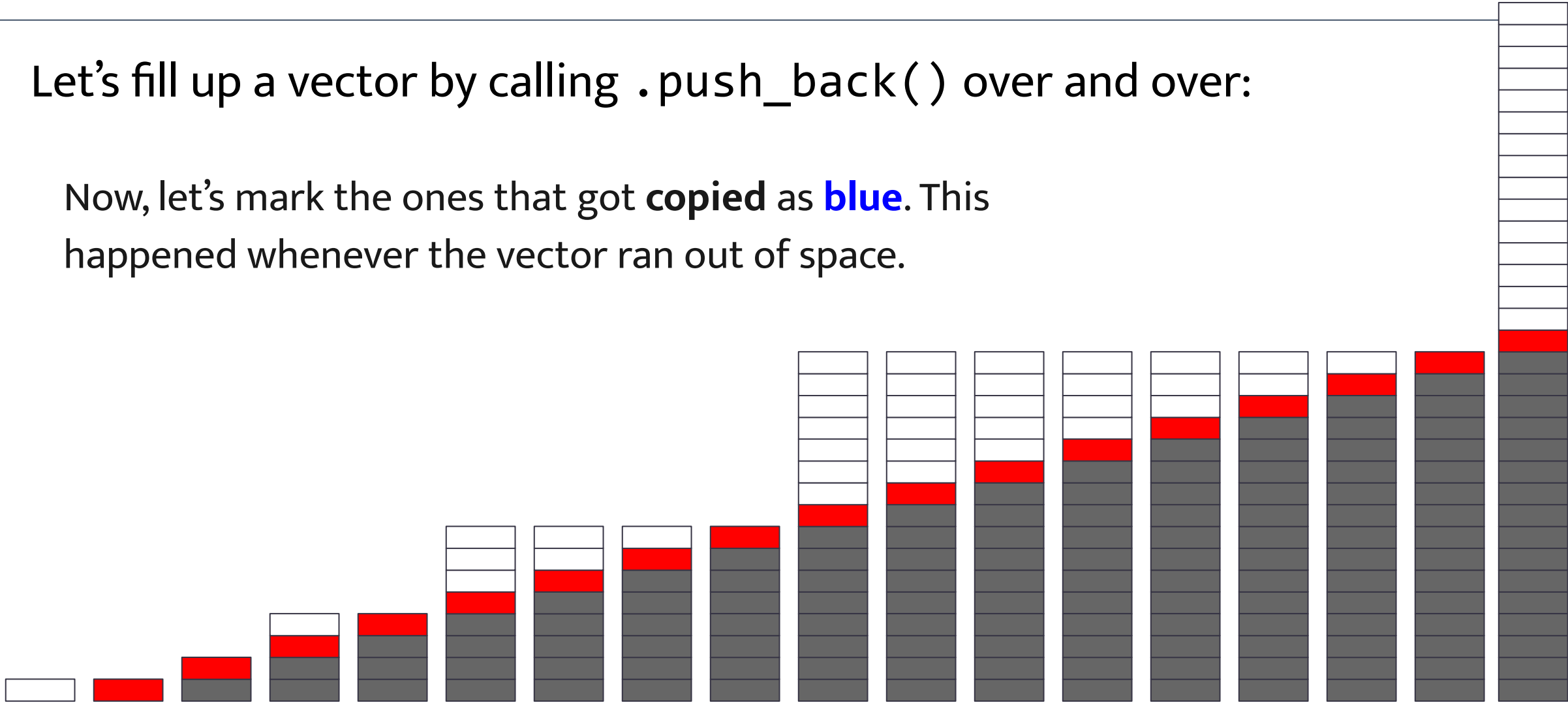
Let's mark the ones that got **inserted** as **red**.



Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

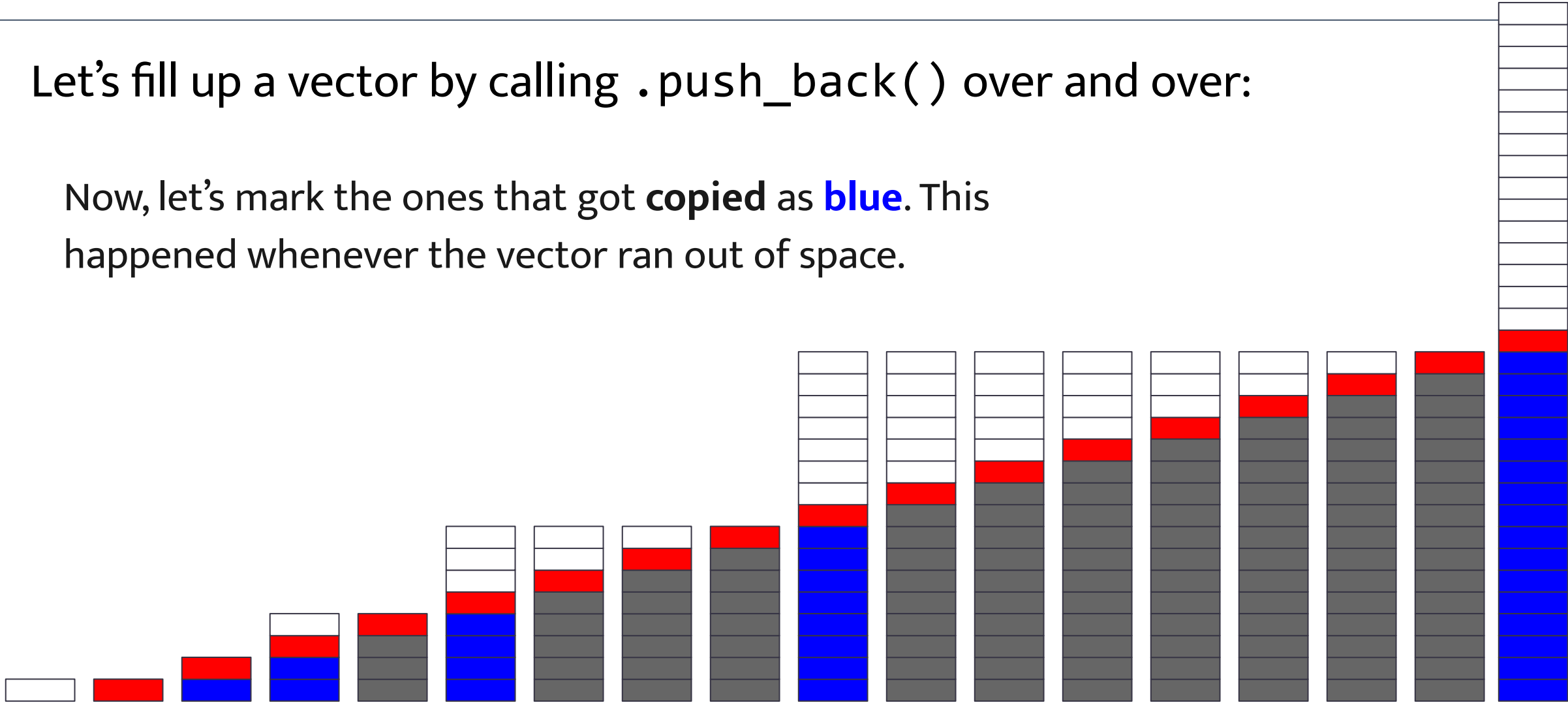
Now, let's mark the ones that got **copied** as **blue**. This happened whenever the vector ran out of space.



Amortized Complexity

- Let's fill up a vector by calling `.push_back()` over and over:

Now, let's mark the ones that got **copied** as **blue**. This happened whenever the vector ran out of space.



Amortized Complexity

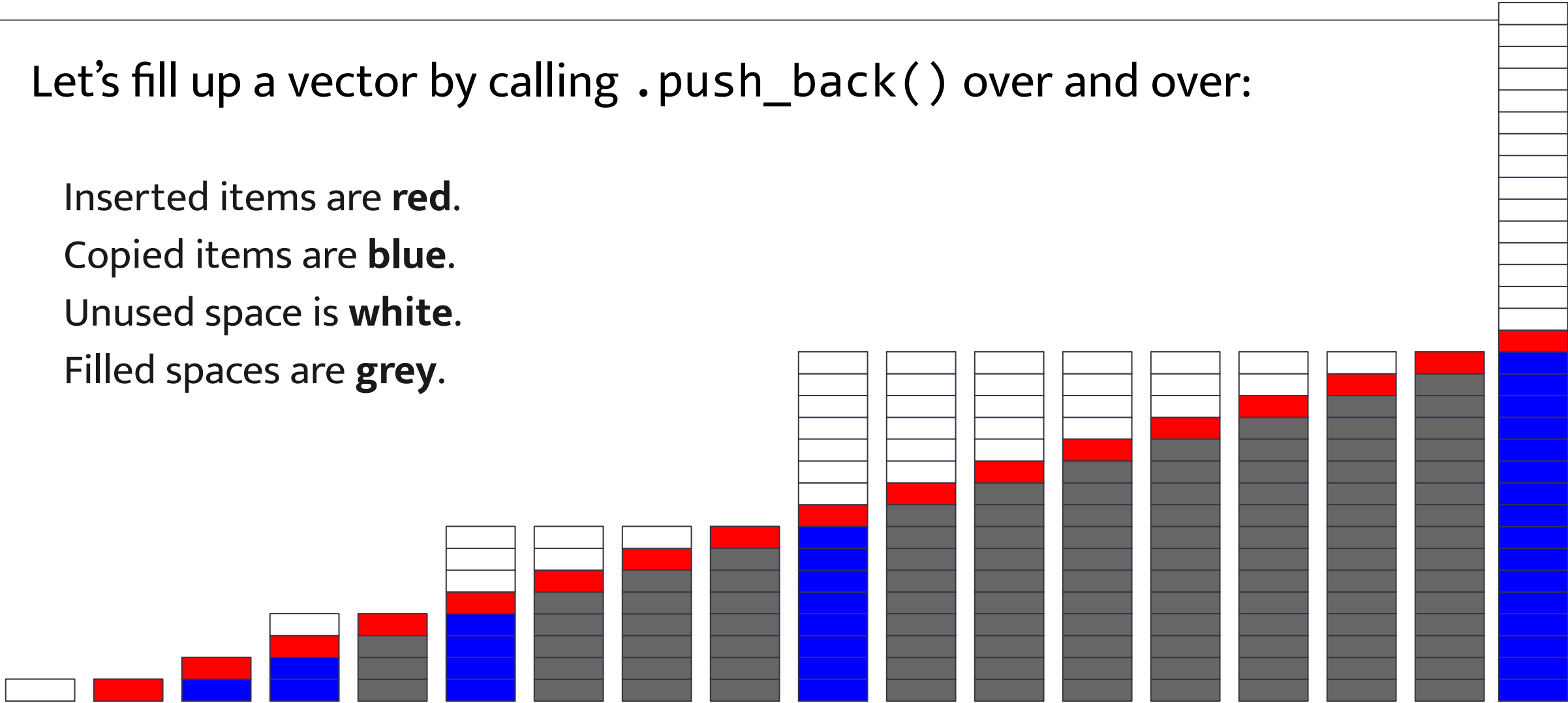
- Let's fill up a vector by calling `.push_back()` over and over:

Inserted items are **red**.

Copied items are **blue**.

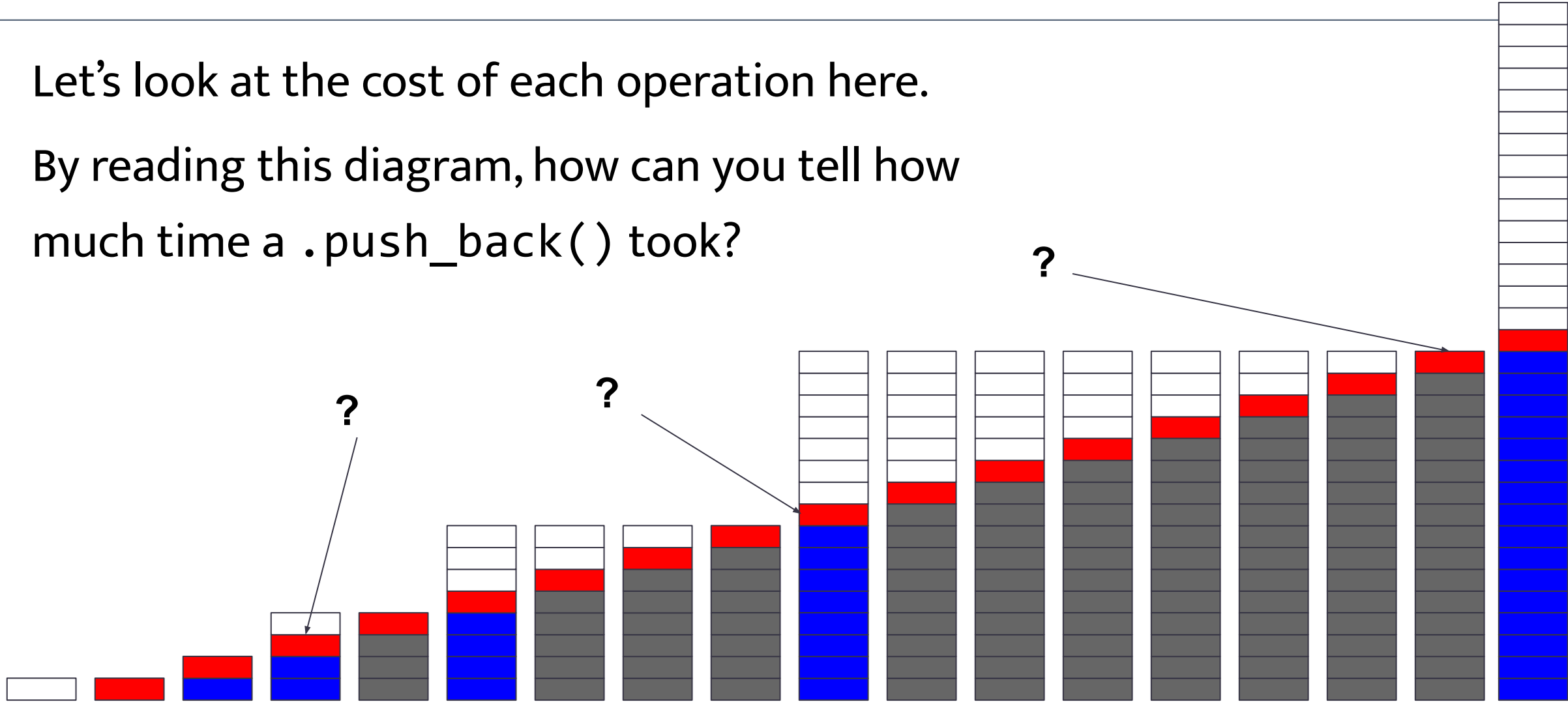
Unused space is **white**.

Filled spaces are **grey**.



Amortized Complexity

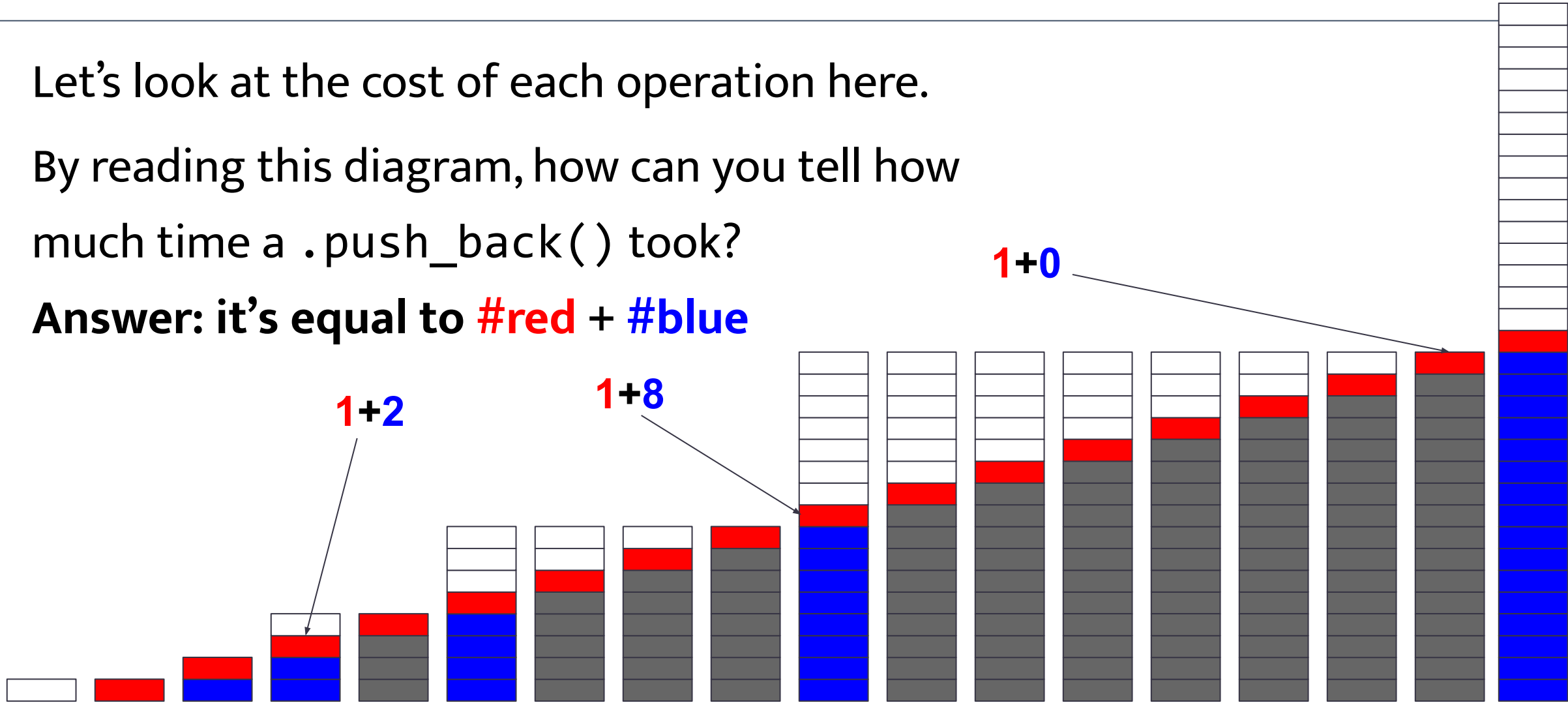
- Let's look at the cost of each operation here.
- By reading this diagram, how can you tell how much time a `.push_back()` took?



Amortized Complexity

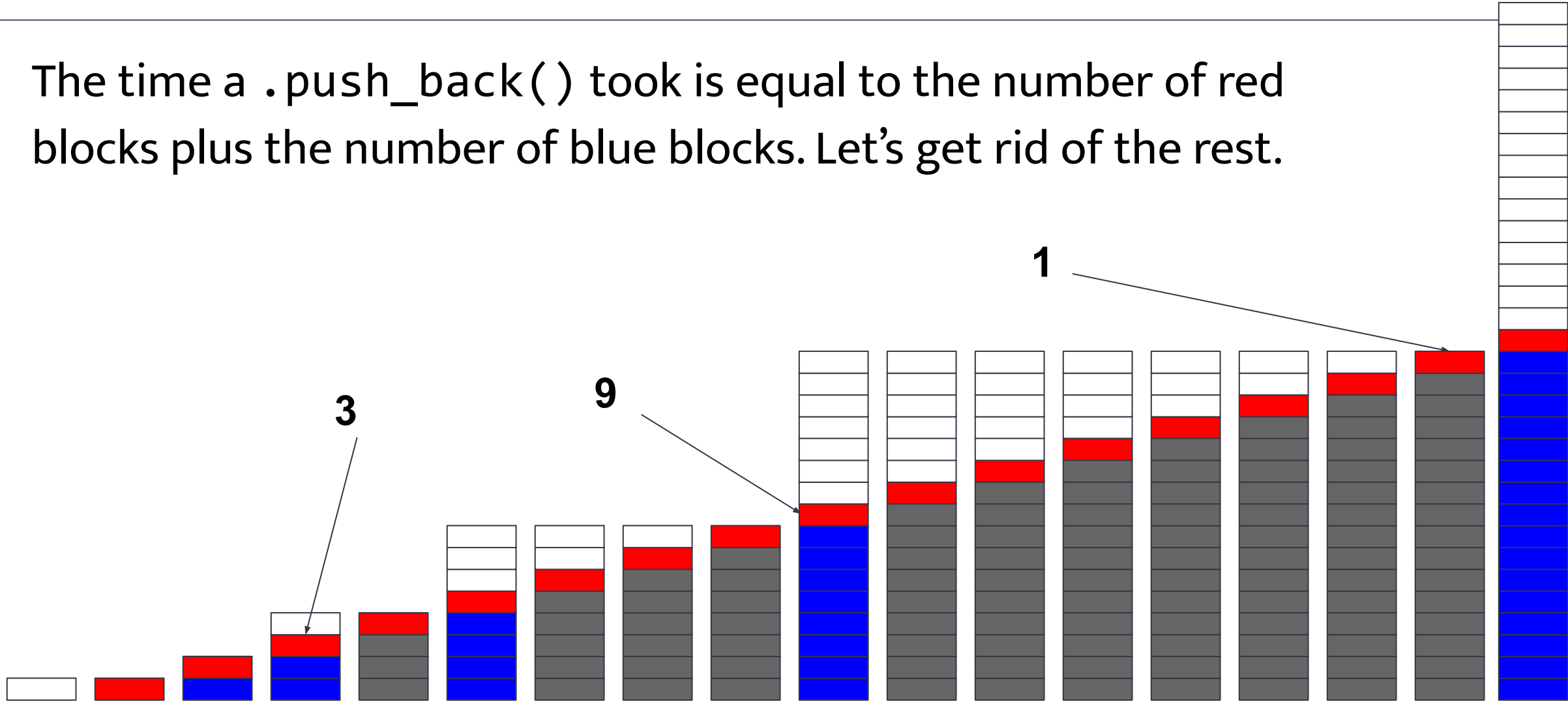
- Let's look at the cost of each operation here.
- By reading this diagram, how can you tell how much time a `.push_back()` took?

Answer: it's equal to **#red** + **#blue**



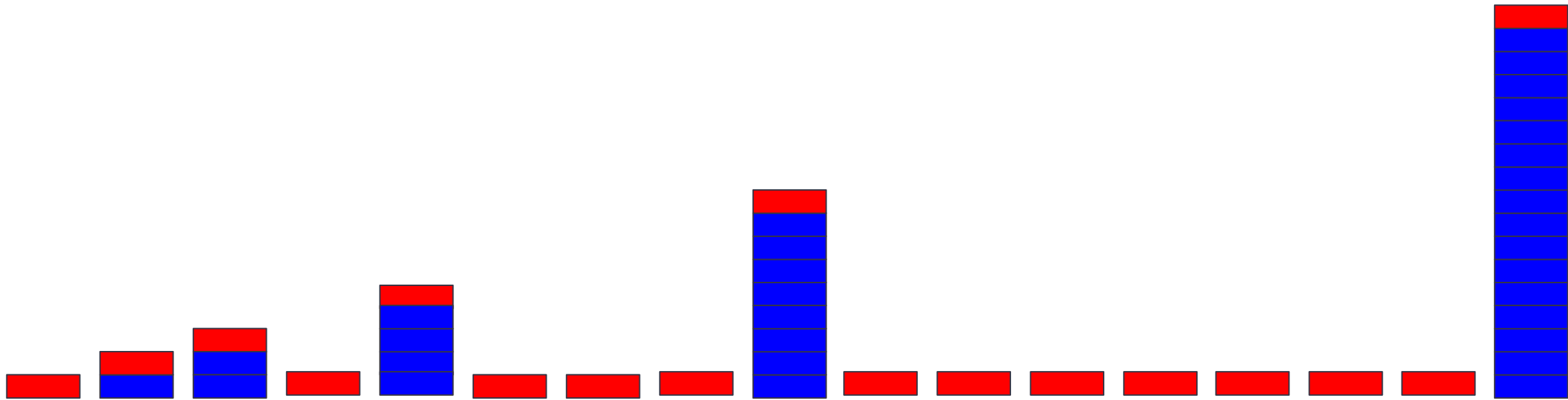
Amortized Complexity

- The time a `.push_back()` took is equal to the number of red blocks plus the number of blue blocks. Let's get rid of the rest.



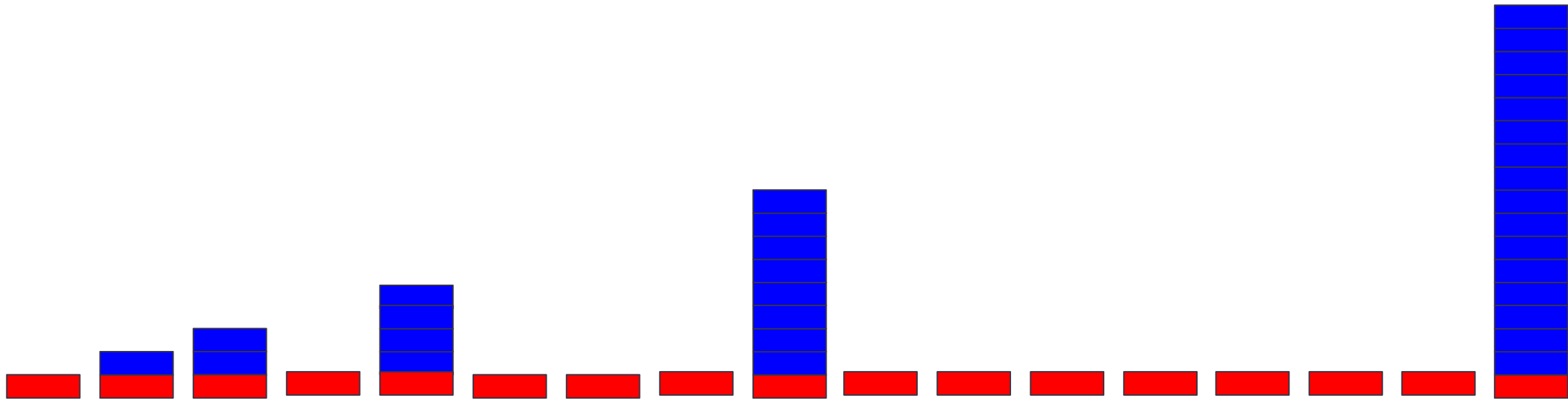
Amortized Complexity

- The *height* of the tower is now the *cost* of each `.push_back()`!



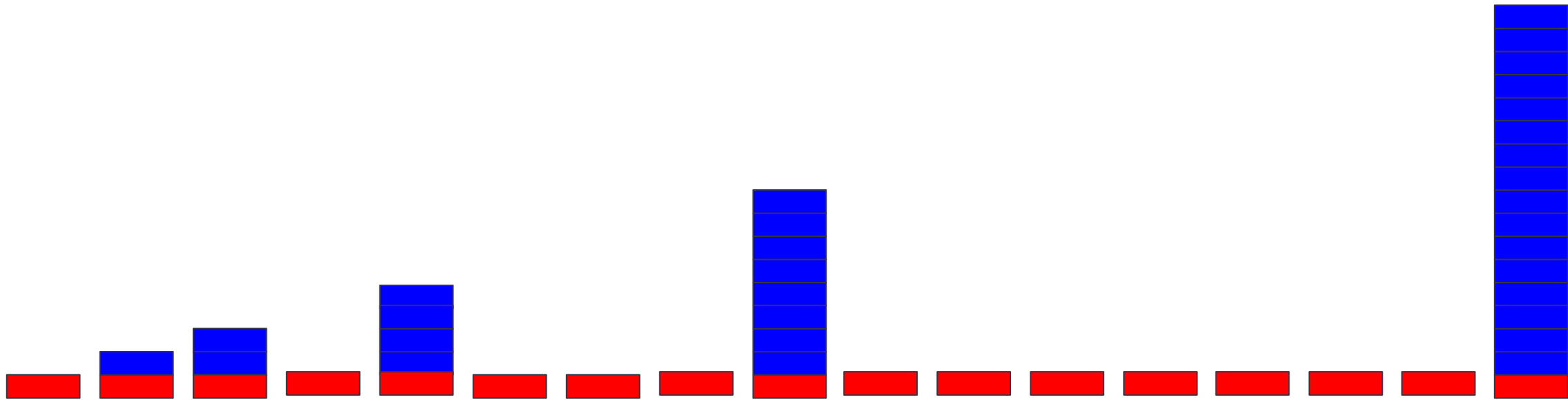
Amortized Complexity

- To make things nicer, we'll move the reds to the bottom.
 - This doesn't change their height, so the total cost is the same.



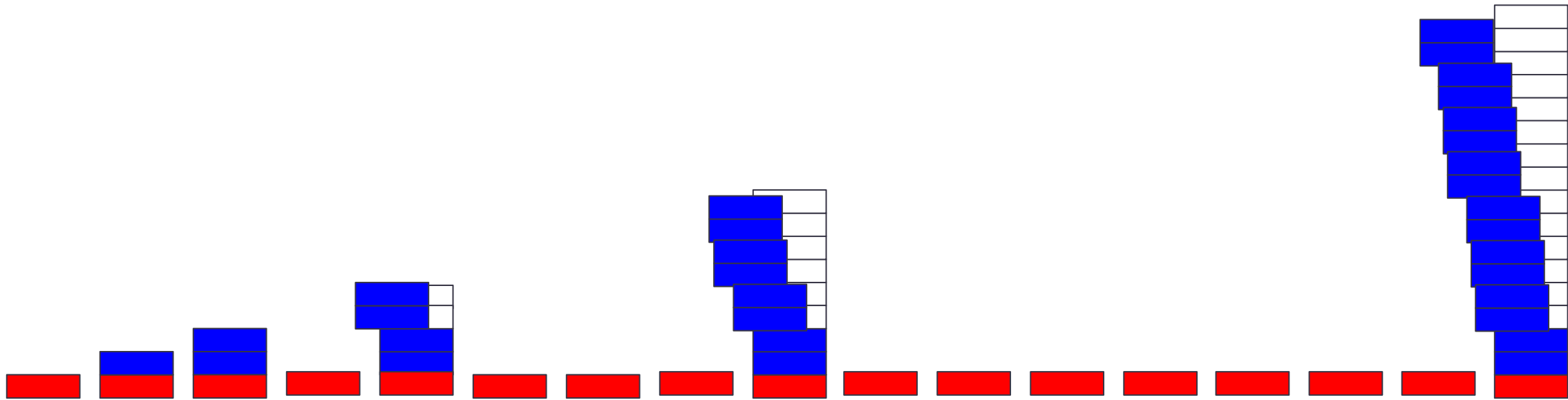
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



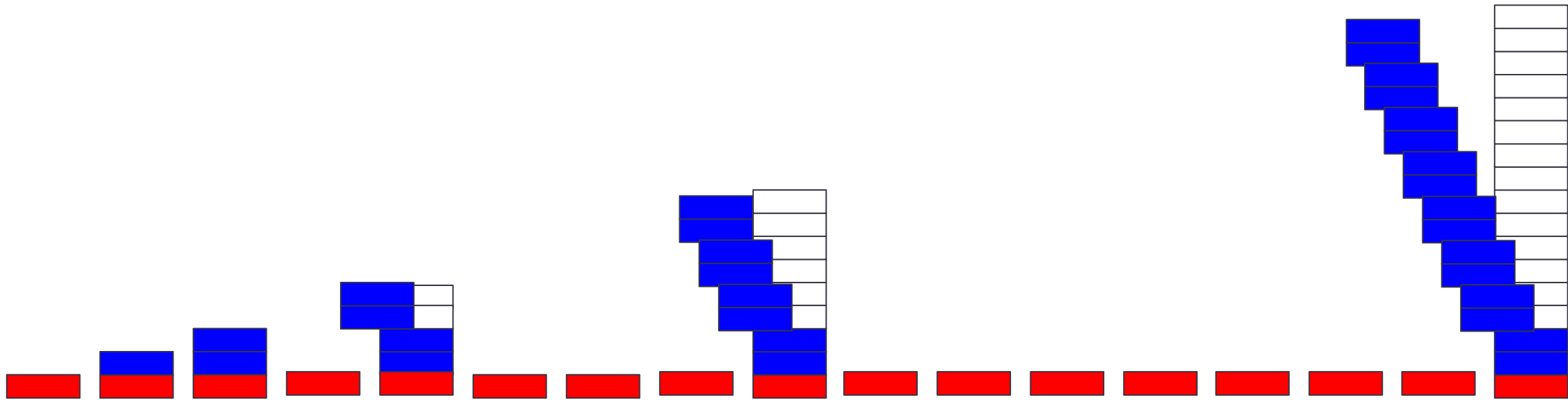
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



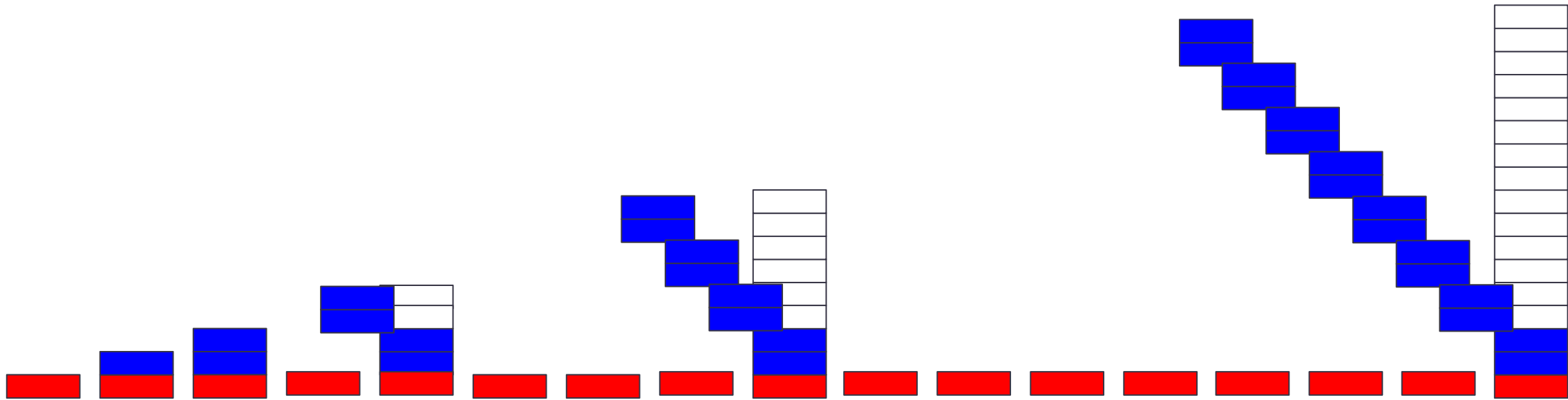
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



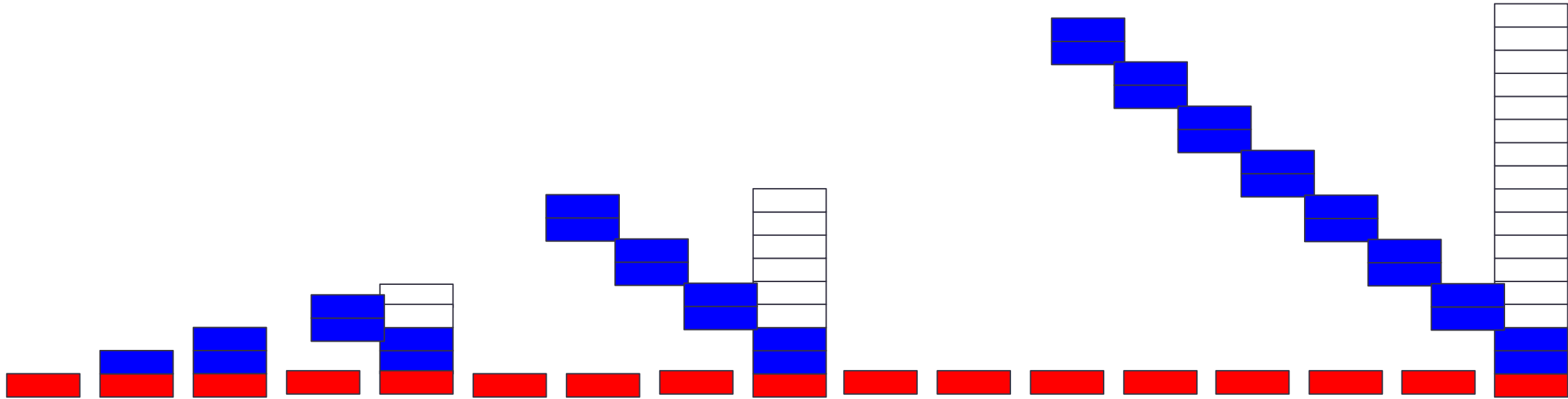
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



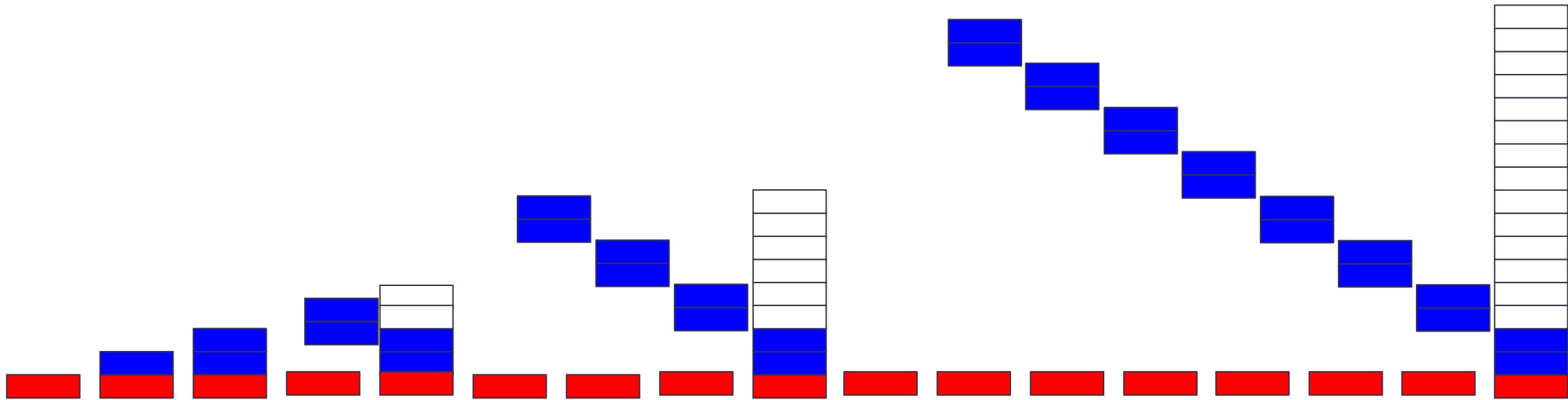
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



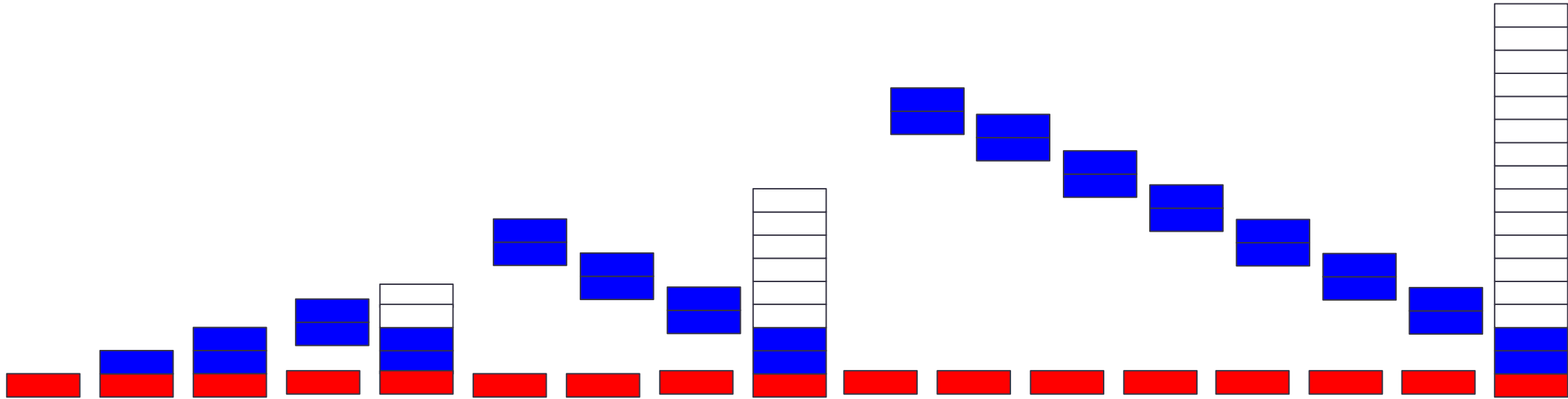
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



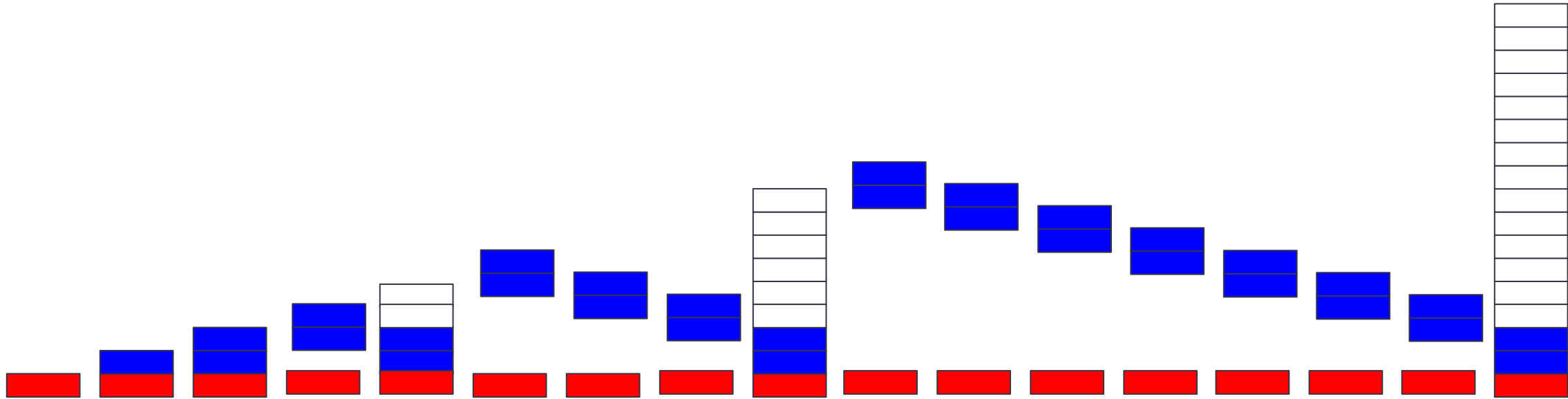
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



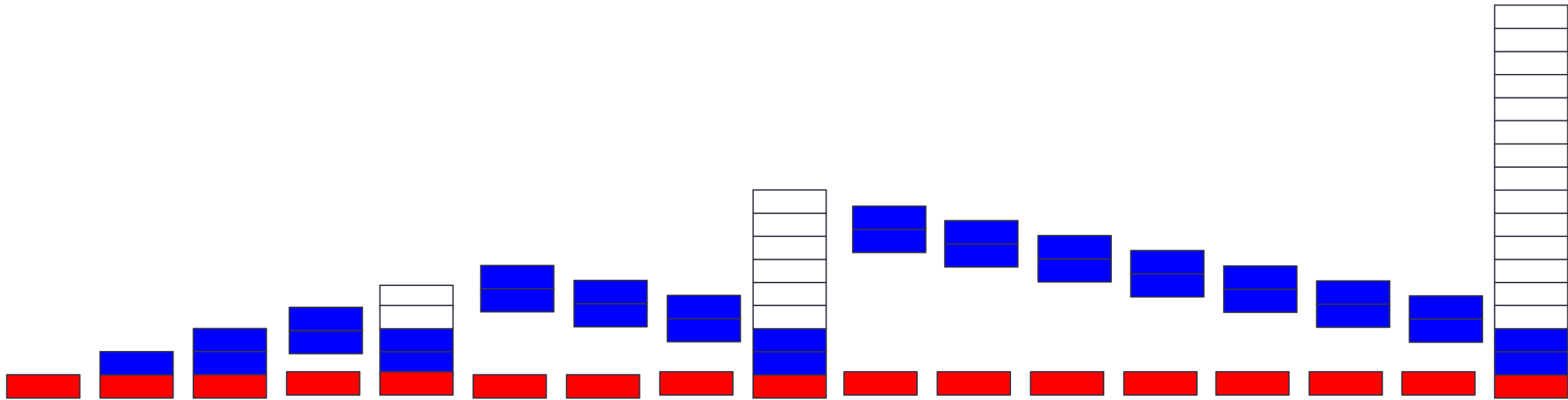
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



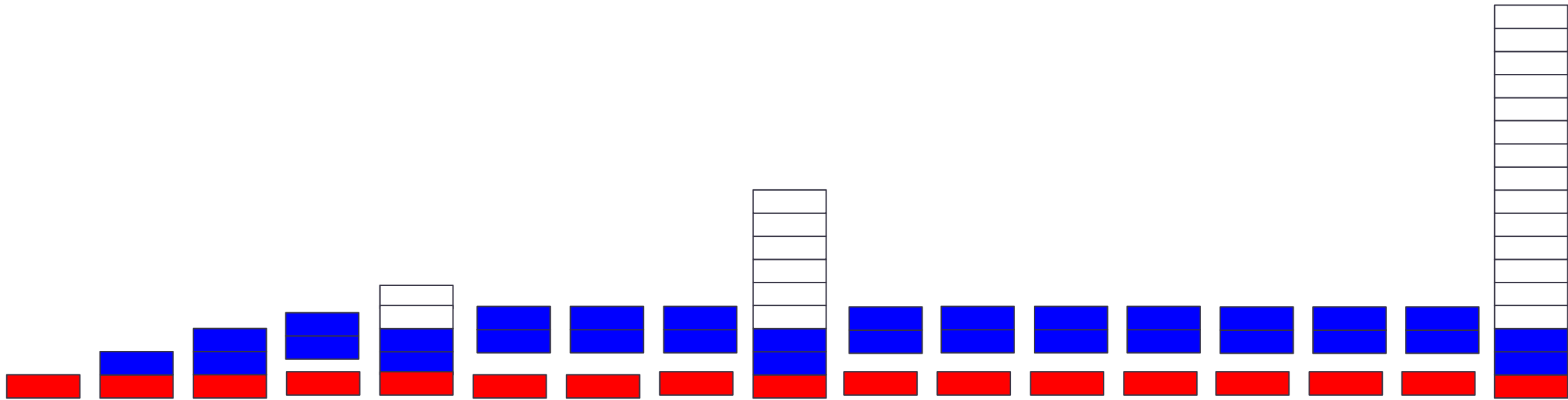
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



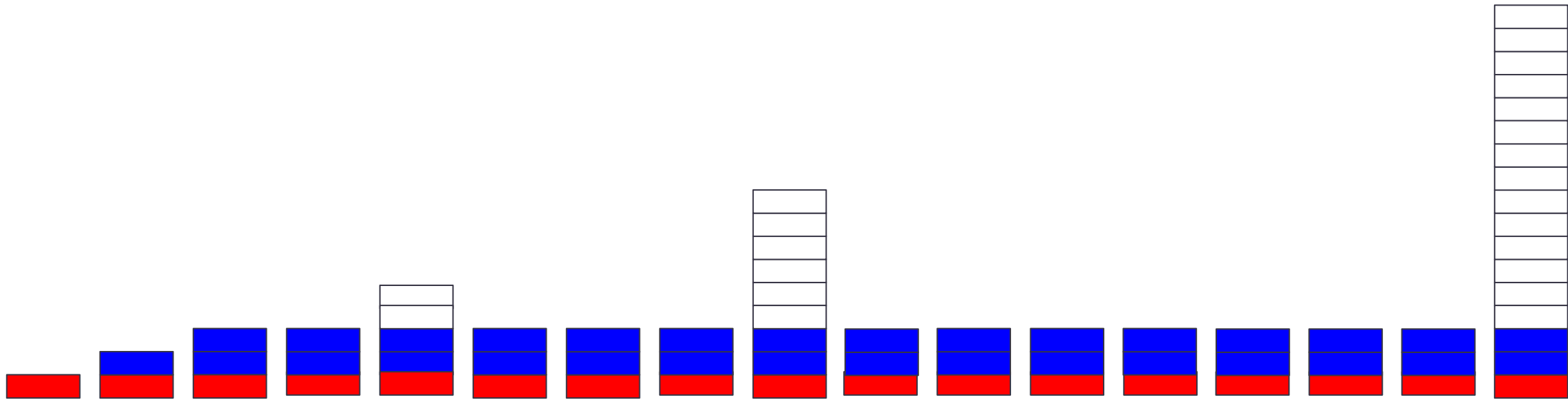
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



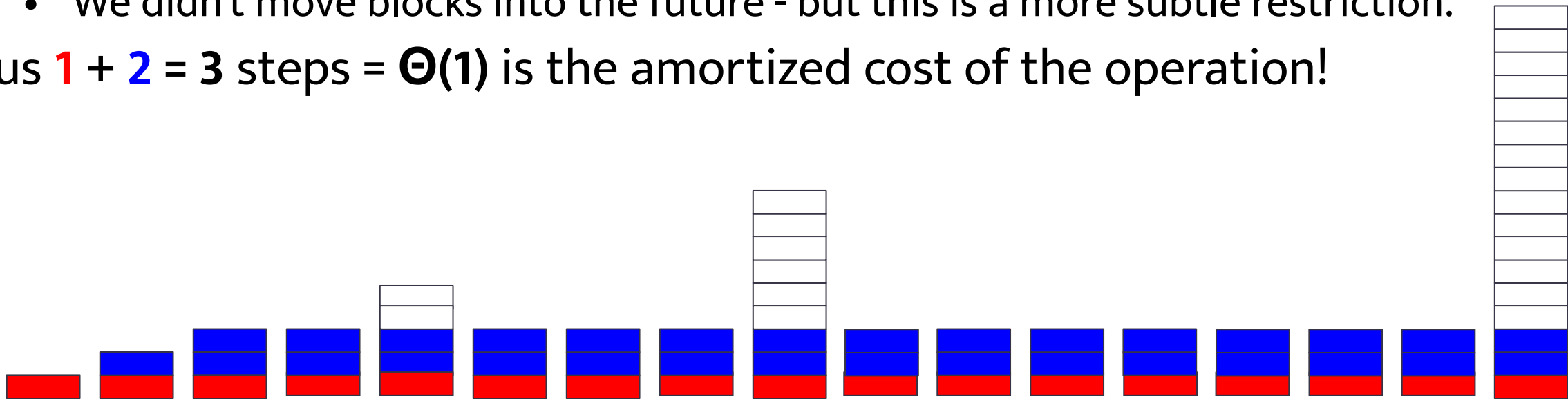
Amortized Complexity

- Now, we will **amortize the costs!** Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



Amortized Complexity

- Hey look, it's constant!
- All we did was shuffle around the costs.
 - The **total** is the same. *Very important.*
 - The **number** of operations is the same.
 - We didn't move blocks into the future - but this is a more subtle restriction.
- Thus **1 + 2 = 3** steps = $\Theta(1)$ is the amortized cost of the operation!



More Amortized Complexity

- When a vector reaches capacity, it reallocates and the capacity *doubles*...
 - thus, the capacity jumps from n to $2n$... for an expensive $\Theta(n)$ operation, we get n free pushes \rightarrow the amortized complexity is $\Theta(n)/n = \Theta(1)$.
- But what if the vector reallocates with a capacity of $n + 100$ instead of $2n$? What is the amortized complexity now? Is it still $\Theta(1)$?

More Amortized Complexity

- When a vector reaches capacity, it reallocates and the capacity *doubles*...
 - thus, the capacity jumps from n to $2n$... for an expensive $\Theta(n)$ operation, we get n free pushes \rightarrow the amortized complexity is $\Theta(n)/n = \Theta(1)$.
- But what if the vector reallocates with a capacity of $n + 100$ instead of $2n$? What is the amortized complexity now? Is it still $\Theta(1)$?
 - **No!** - instead of getting n free pushes for each $\Theta(n)$ reallocation, we only get 100. Thus, if we spread the $\Theta(n)$ work across the 100 free constant-time pushes, we get an amortized complexity of $\Theta(n)/100$, which is still $\Theta(n)$!

Amortized Complexity: A Wrap-Up

- Amortization:
 - I'm measuring the total cost of a sequence of operations.
 - Some of my operations are **expensive**, but the majority of my operations are **cheap**. So multiplying the largest cost by the number of operations gives a cost that is **too high**.
 - e.g. the worst case time complexity of pushing an element into a vector is $\Theta(n)$, but is the worst case time complexity of pushing n elements into a vector $\Theta(n^2)$, since you are doing a worst case $\Theta(n)$ operation n times? No!
 - When the excess work from the **expensive** is averaged out over the **cheap** operations, I find a more accurate upper bound for the complexity of that operation.

C-Strings and C++ Strings

C-Strings

- C-strings are arrays of characters terminated by a null-character.
- How to declare a c-string:

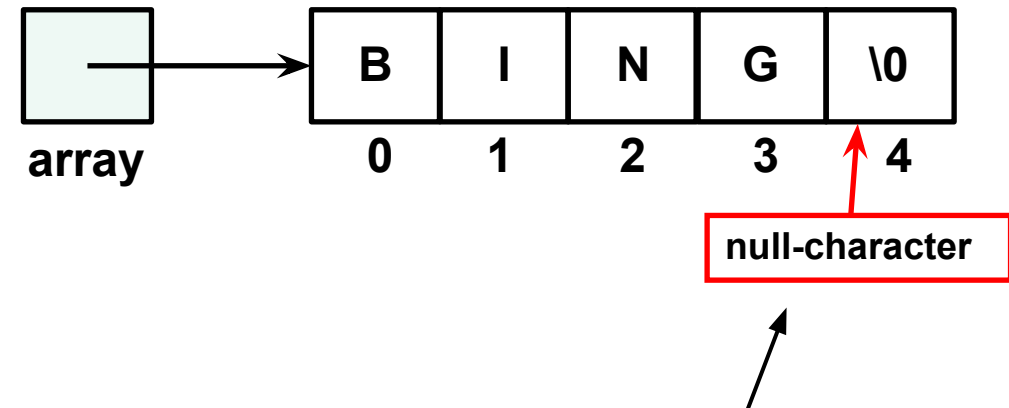
- `const char* array = "BING";`

OR

- `char array[] = "BING";`

OR

- `char* array = new char[5];`
`array[0] = 'B';`
`array[1] = 'I';`
`array[2] = 'N';`
`array[3] = 'G';`
`array[4] = '\0';`



The null character (sentinel) is important! It signals to the program that the string ends here.

C++ Strings

- Full-fledged object, defined in `<string>` of STL:

```
string str1 = "BING";           // str1 contains "BING"  
string str2("BING");           // str2 contains "BING"  
string str3("BING", 2, 1);      // str3 contains "N"
```

- STL has many member functions for C++ strings:

- Iterators: `begin, end, etc.`
- Capacity: `size, length, resize, reserve, clear, etc.`
- Element Access: `operator[], at, back, front`
- Modifiers: `operator+=, append, push_back, etc.`
- Operations: `c_str, get_allocator, find, etc.`

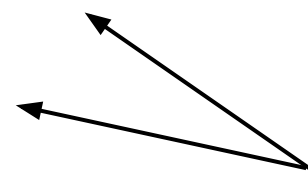
C-Strings vs. C++ Strings

- Use of functions on strings:
 - C-string (function call with string passed as function argument)

```
char str[] = "BING";  
cout << strlen(str); // prints 4
```

- C++ string (dot operator for member function)

```
string str("BING");  
cout << str.length(); // prints 4  
cout << str.size(); // prints 4
```



size and length are
aliases... they do the
same thing!

C-Strings vs. C++ Strings

- **In general, use C++ strings over C-strings:**
 - encapsulation
 - size is stored - no need to keep track of null terminator
 - more fully featured: iterators, easy growth syntax, safety with `getline`, `to_string`
 - some C++ string functions are faster than their C-string counterparts
 - e.g. `.length()` is much faster than `strlen()`

Handwritten Problem

Handwritten Problem

- Write a function that takes in two strings and returns whether they are anagrams of each other (words that contain the same letters). The only characters will be spaces and lowercase letters. Do this in $\Theta(n)$ time.
 - **Example 1:** Given `s1 = "anagram"` and `s2 = "nagaram"`, return true.
 - **Example 2:** Given `s1 = "i love eecs"` and `s2 = "i scole ve e"`, return true.
 - **Example 3:** Given `s1 = "anagrams"` and `s2 = "anagrams anagrams"`, return false.
 - **Example 4:** Given `s1 = "cats"` and `s2 = "cat"`, return false.
- Completion of this problem is worth 5 points.

```
// check if two strings are anagrams  
bool isAnagram(const string &s1, const string &s2);
```

Lab 2 Handwritten Review

Handwritten Problem

- Implement a queue with a singly linked list:

```
template <typename T>
class LinkedQueue {
private:
    Node<T>* head = nullptr;
    Node<T>* tail = nullptr;
    size_t count = 0;
public:
    T front() const { /* Implement */ }
    void pop() { /* Implement */ }
    void push(T x) { /* Implement */ }
    size_t size() const { return count; }
    bool empty() const { return count == 0; }
    ~LinkedQueue() { /* Implement */ }
};
```

```
template <typename T>
struct Node {
    T value;
    Node* next;
};
```

Lab 2 Written Problem: Linked List Queue

- Implement several functions of a queue using a linked list.
 - implement the front() operation:

```
template<typename T>
class LinkedList {
private:
    Node<T> *head;
    Node<T> *tail;
    size_t count;
public:
    T front() {
        assert(!empty());
        return head->value;
    }
};
```

Lab 2 Written Problem: Linked List Queue

- Implement several functions of a queue using a linked list.
 - implement the pop() operation:

```
template<typename T>
class LinkedQueue {
private:
    Node<T> *head;
    Node<T> *tail;
    size_t count;
public:
    void pop() {
        assert(!empty());
        Node<T>* temp = head;
        head = head->next; //move head forward
        delete temp; //delete node
        if (!head) tail = nullptr; //if deleted the only node then set tail to null
        --count;
    }
};
```

Lab 2 Written Problem: Linked List Queue

- Implement several functions of a queue using a linked list.
 - implement the push() operation:

```
template<typename T>
class LinkedQueue {
private:
    Node<T> *head;
    Node<T> *tail;
    size_t count;
public:
    void push(const T& x) {
        Node<T>* new_tail = new Node<T>{x, nullptr}; //allocate memory
        if (tail) tail->next = new_tail; //add to end of LL if it exists
        else head = new_tail; //otherwise set head to only node in LL
        tail = new_tail; //set tail pointer to new node
        ++count;
    }
};
```

Lab 2 Written Problem: Linked List Queue

- Implement several functions of a queue using a linked list.
 - implement the destructor operation:

```
template<typename T>
class LinkedQueue {
private:
    Node<T> *head;
    Node<T> *tail;
    size_t count;
public:
    ~LinkedQueue() {
        while (head){
            Node<T> *temp = head;
            head = head->next;
            delete temp;
        }
    }
};
```


Practice Problems

Strings

Leetcode Problem 3: Longest Substring without repeating characters

Leetcode Problem 28: Implement strStr()

Leetcode Problem 344: Reverse String

Leetcode Problem 151: Reverse Words in a String

Leetcode Problem 186: Reverse Words in a String 2

Leetcode Problem 443: String Compression

Leetcode Problem 412: Fizz Buzz

Leetcode Problem 884: Uncommon words from two sentences

Leetcode Problem 13: Roman to Integer

Leetcode Problem 14: Longest Common Prefix

Leetcode Problem 539: Minimum Time Difference

Leetcode Problem 680: Valid Palindrome 2

Leetcode Problem 792: Number of Matching Subsequences

Leetcode Problem 953: Verifying an Alien Dictionary

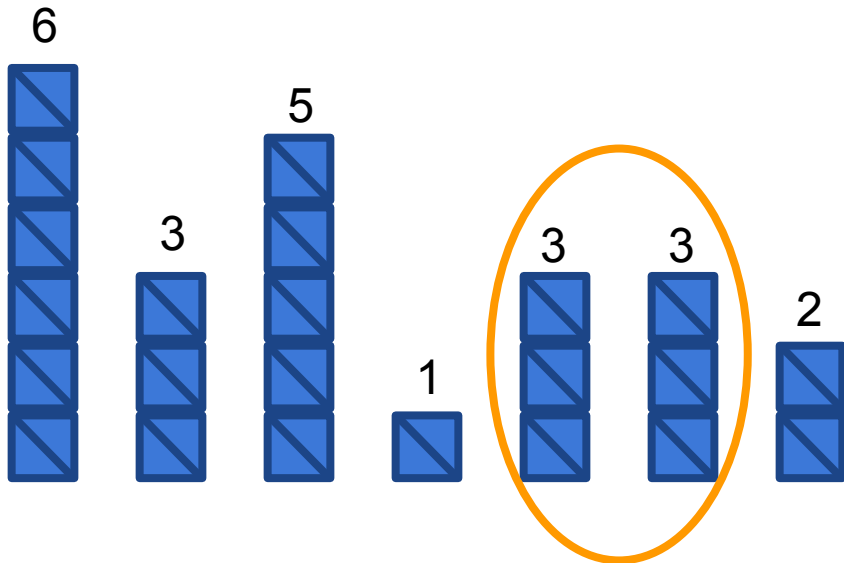
Leetcode Problem 1041: Robot Bounded in a Circle

Leetcode Problem 1047: Remove All Adjacent Duplicates In String

Leetcode Problem 1249: Minimum Remove to Make Valid
Parenthesis

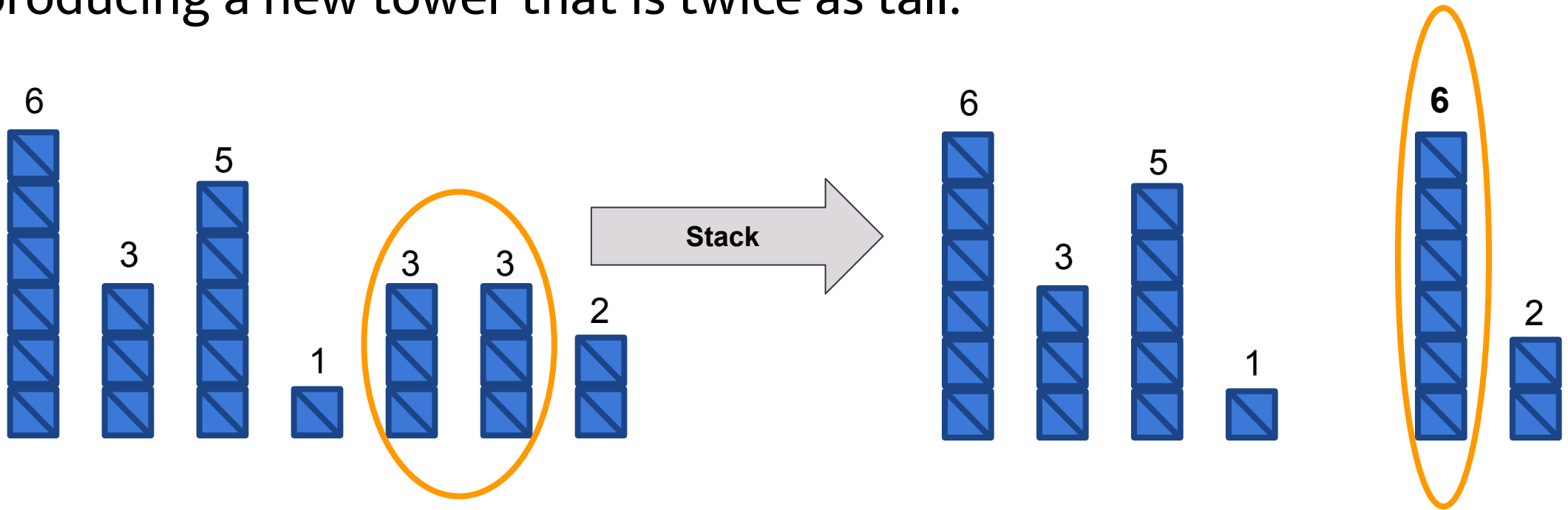
More Amortized Complexity

- Your warehouse gets into the *box stacking* business.
- You have various towers of boxes; each has a *height* (how many boxes it contains). Two towers can be ***stacked*** if they have the *same height*, producing a new tower that is twice as tall.



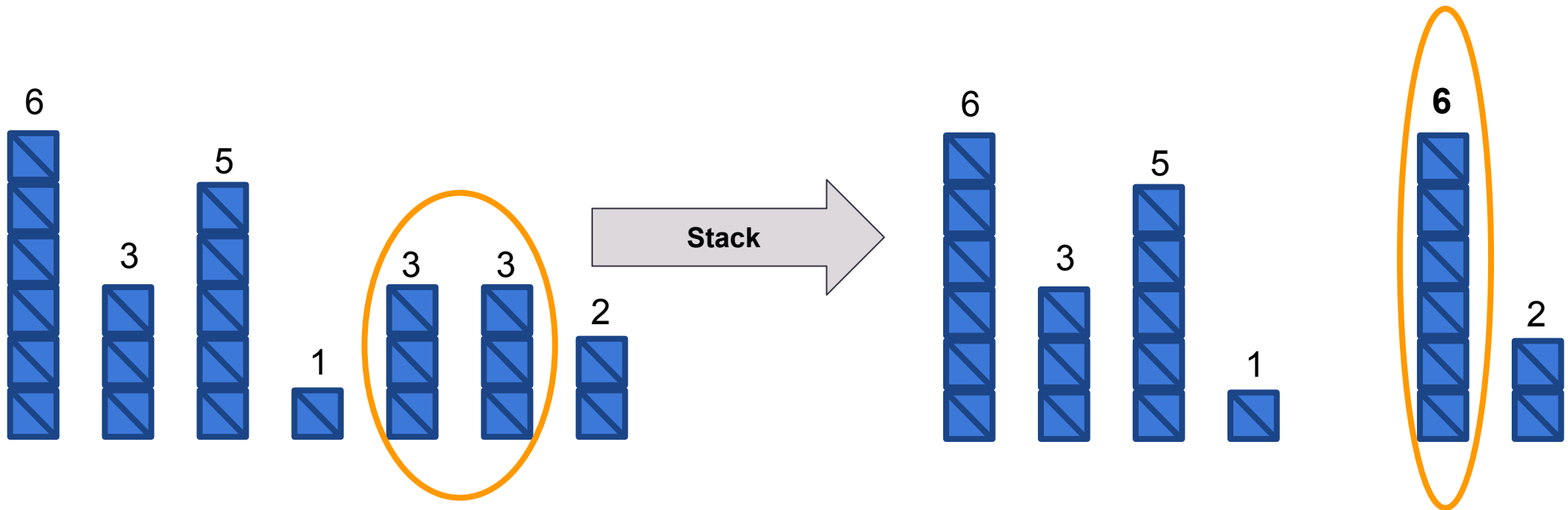
More Amortized Complexity









- Your warehouse gets into the *box stacking* business.
- You have various towers of boxes; each has a *height* (how many boxes it contains). Two towers can be **stacked** if they have the *same height*, producing a new tower that is twice as tall.






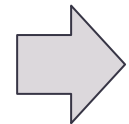





More Amortized Complexity

- Your warehouse gets into the *box stacking* business.
- You have various towers of boxes; each has a *height* (how many boxes it contains). Two towers can be **stacked** if they have the *same height*, producing a new tower that is twice as tall.
- We can also **import** single-crate towers and **export** towers of any height.






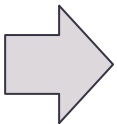





Order:	Import $+1$ 	Stack <i>(must be the same height)</i> <div style="display: flex; align-items: center;"> <div style="text-align: center; margin-right: 10px;"> h  </div> <div style="text-align: center; margin-right: 10px;"> h  </div> <div style="font-size: 2em; margin: 0 10px;">→</div> <div style="text-align: center;"> $2h$  </div> </div>	Export h 
Actual Cost:	1 	h 	h 

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

Order:	<div data-bbox="318 114 496 171">Import</div> <div data-bbox="356 342 458 399">+1</div> <div data-bbox="356 414 458 514"></div>	<div data-bbox="687 57 942 207">Stack (must be the same height)</div> <div data-bbox="624 271 700 314">h</div> <div data-bbox="624 321 700 528"></div> <div data-bbox="751 271 828 314">h</div> <div data-bbox="751 321 828 528"></div> <div data-bbox="879 342 1006 464"></div> <div data-bbox="1044 64 1121 114">$2h$</div> <div data-bbox="1044 114 1121 528"></div>	<div data-bbox="1223 99 1401 157">Export</div> <div data-bbox="1286 178 1337 221">h</div> <div data-bbox="1261 221 1363 521"></div>
Actual Cost:	<div data-bbox="343 664 394 721">1</div> <div data-bbox="394 642 471 742"></div>	<div data-bbox="802 664 853 721">h</div> <div data-bbox="853 642 930 742"></div>	<div data-bbox="1286 664 1337 721">h</div> <div data-bbox="1337 642 1414 742"></div>

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?

Order:	Import $+1$ 	Stack <i>(must be the same height)</i> <div style="display: flex; align-items: center;"> <div style="text-align: center;"> h  </div> <div style="text-align: center;"> h  </div> <div style="margin: 0 20px;">  </div> <div style="text-align: center;"> $2h$  </div> </div>	Export h 
Actual Cost:	1 	h 	h 

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?

hk  ? How does h relate to k ?

More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations



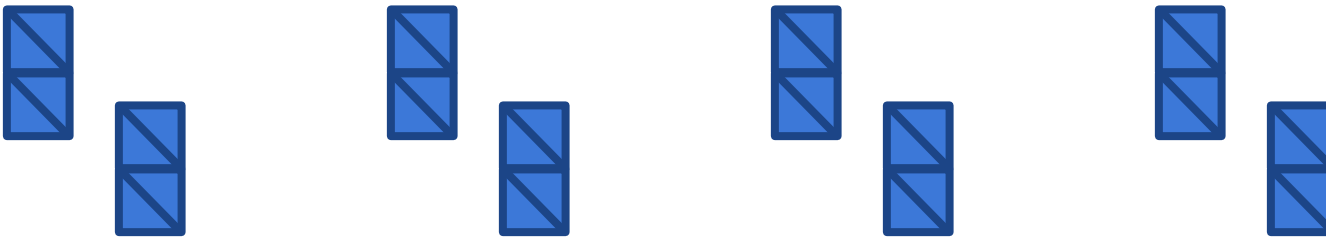
More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations
 - ...



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations
 - ...
 - perform one **stack($k/2$)** operation



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations
 - ...
 - perform one **stack($k/2$)** operation



More Amortized Complexity


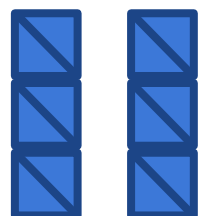





- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations
 - ...
 - perform one **stack($k/2$)** operation
- Height: $k/2$
- Total operation count:
$$k/2 + k/4 + k/8 + \dots + 1 = k(1/2 + 1/4 + \dots) = k$$



More Amortized Complexity

- How does h relate to k ?
- h can get pretty big:
 - perform $k/2$ **import** operations
 - perform $k/4$ **stack(1)** operations
 - perform $k/8$ **stack(2)** operations
 - perform $k/16$ **stack(4)** operations
 - ...
 - perform one **stack($k/2$)** operation
- Height: $k/2$
- Total operation count:
$$k/2 + k/4 + k/8 + \dots + 1 = k(1/2 + 1/4 + \dots) = k$$
Thus, $h \leq k/2$.


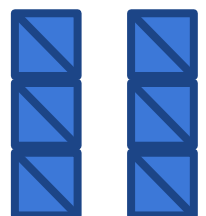







	<p>Import</p> <p>+1</p> 	<p>Stack (must be the same height)</p> <p>h h</p>  <p>→</p> <p>$2h$</p> 	<p>Export</p> <p>h</p> 
<p>Actual Cost:</p>	<p>1</p> 	<p>h</p> 	<p>h</p> 

$$h \leq k/2$$

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?


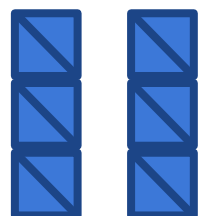





	<p>Import</p> <p>+1</p> 	<p>Stack (must be the same height)</p> <p>h h</p>  <p>→</p> <p>$2h$</p> 	<p>Export</p> <p>h</p> 
<p>Order:</p>	<p>Actual Cost:</p> <p>1</p> 	<p>h</p> 	<p>h</p> 

$$h \leq k/2$$

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?


If we spend the maximum cost on every box: $k^2/2$ 


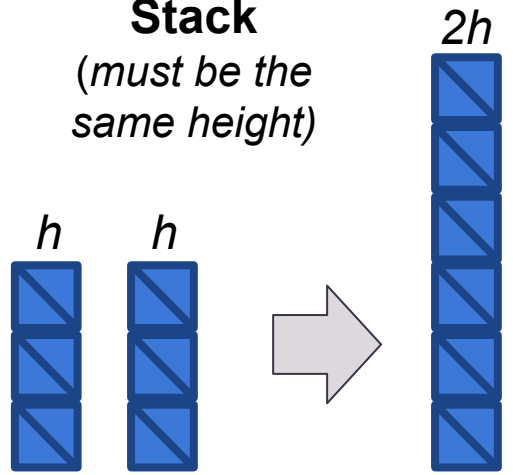




	<p>Import</p> <p>+1</p> 	<p>Stack (must be the same height)</p> <p>h h</p>  <p>→</p> <p>$2h$</p> 	<p>Export</p> <p>h</p> 
<p>Actual Cost:</p>	<p>1</p> 	<p>h</p> 	<p>h</p> 

$$h \leq k/2$$

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?

If we spend the maximum cost on every box: $k^2/2$  **Can we do better???**


Order:	Import 	Stack <i>(must be the same height)</i> 	Export 
Actual Cost:	1 	h 	h 


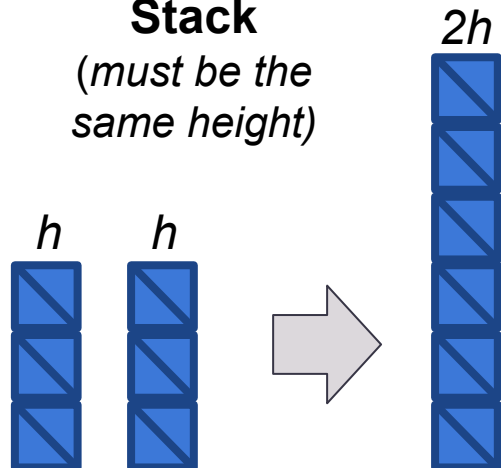




$$h \leq k/2$$

Your turn!

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?

If we spend the maximum cost on every box: $k^2/2$  **Can we do better???**

Order:	Import 	Stack <i>(must be the same height)</i> 	Export 
Actual Cost:	1 	h 	h 


$$h \leq k/2$$


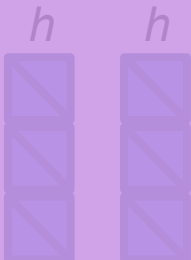





Your turn!

HINT: follow one box... how much cost can you attribute to one crate over its lifetime?

Every morning, the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

In terms of k , what is the maximum amount of fuel that you can spend in one day?

If we spend the maximum cost on every box: $k^2/2$  **Can we do better???**

	Import	Stack (must be the same height)	Export
Order:	+1 	 	
Actual Cost:	1 	h 	h 

$$h \leq k/2$$

THE

your turn!

HINT: follow one box... how much cost can you attribute to one crate over its lifetime?

ANSWER

Every morning the warehouse begins empty. You then receive k orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height to make one tower.

In terms of k , what is the maximum amount of fuel that you can spend in one day?

If we spend the maximum cost on every box: $k^2/2$ 

Can we do better???

More Amortized Complexity

- Every box is involved in at most:
 - ??? import operation
 - ??? stack operations
 - ??? export operation



$$h \leq k/2$$

More Amortized Complexity

- Every box is involved in at most:

- 1 **import operation**



- $\log_2(k/2)$ **stack operations** (but twice as many crates as fuel)


- 1 **export operation**

$$h \leq k/2$$

More Amortized Complexity

- Every box is involved in at most:

$$h \leq k/2$$

- 1 import operation
-  $\log_2(k/2)$ stack operations (but twice as many crates as fuel)
- 1 export operation

- If we get the *import* operation to pay for all of these, it costs:

$$= 1 + \log_2(k/2)/2 + 1$$

$$= 1.5 + \log_2(k)/2$$

the other two operations (stack and export) are free

Thus, at most $k \log_2(k)/2 + 1.5k$ fuel can be spent in one day!