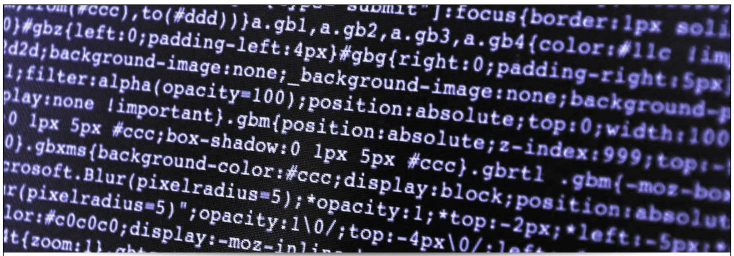


03

September 17 - 23, 2024

Complexity Analysis, Amortization, Strings

Announcements

- Project 1 is due **TUESDAY, SEPT 17!!!** 🤯
- Lab 2 autograder (Anyfix) due Monday, **Sept 23**.
- Lab 2 assignment (quiz) due Monday, **Sept 23**.
- Lab 3 handwritten (Anagram) due Monday, **Sept 23** (in-lab).
- Lab 3 AG and quiz due Monday, **Sept 30**.
- Project 2 will be released Thursday, Sept 19 (due **Oct 10**).
- Midterm will be on Thursday, October 17.
  - 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 modelled 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

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$
$$T(1) = c_0$$

where a ≥ 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

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

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

a = 8 $b^c = 2^1 = 2$ a > b^c

b = 2

c = 1

$$\Theta(n^{log_b a}) = \Theta(n^{log_2 8})$$
$$= \Theta(n^3)$$

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

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.

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

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

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

    $= 2^{n-1} + 2^{n-1} - 1 =$ **$O(2^n)$**

this is our base case, T(1) = 1

Formula to simplify summation

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

x = n – 2

$(2^{((n-2)+1)} - 1) / (2 - 1)$

$= 2^{(n-1)} - 1$

you don't need to know this equation!

Identifying Recurrence Relations

- Given the function below, calculate the recurrence relation. Assume `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

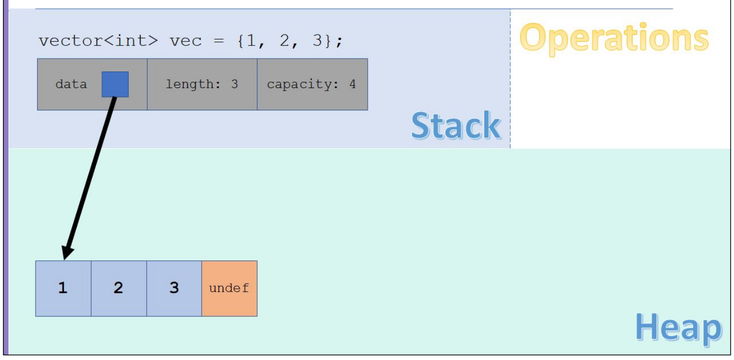
- Vectors have a data pointer, a "size," and a "capacity."
- The **capacity** is how much *room* they have.
- The **size** is how many elements they actually contain.

- 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

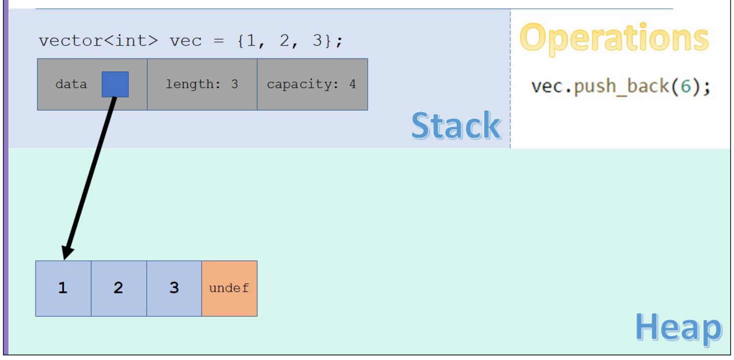
Resize and Reserve

- 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.

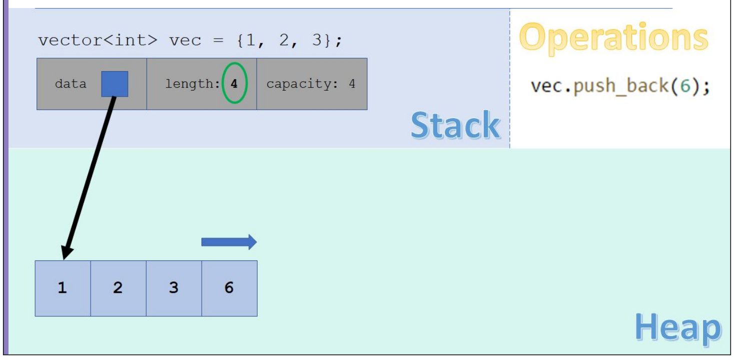
Resize and Reserve



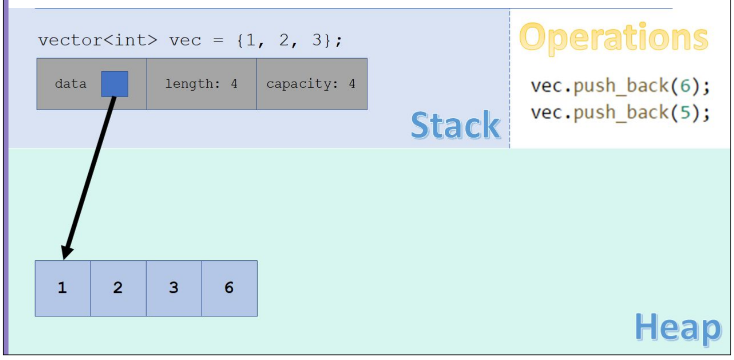
Resize and Reserve



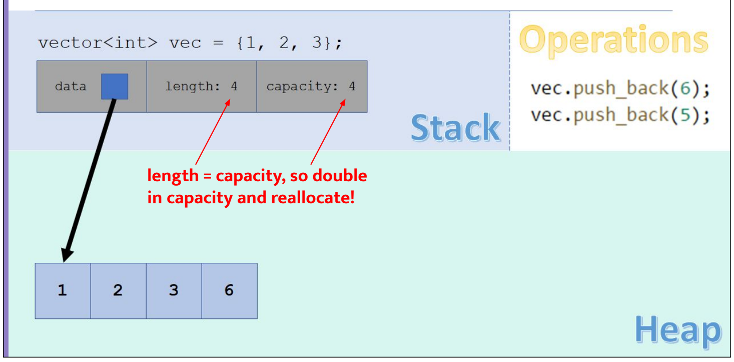
Resize and Reserve



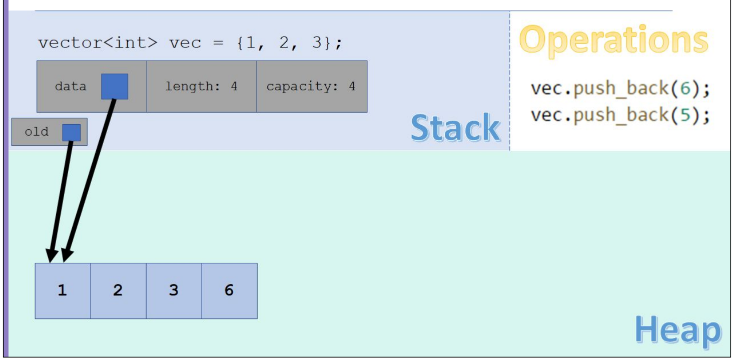
Resize and Reserve

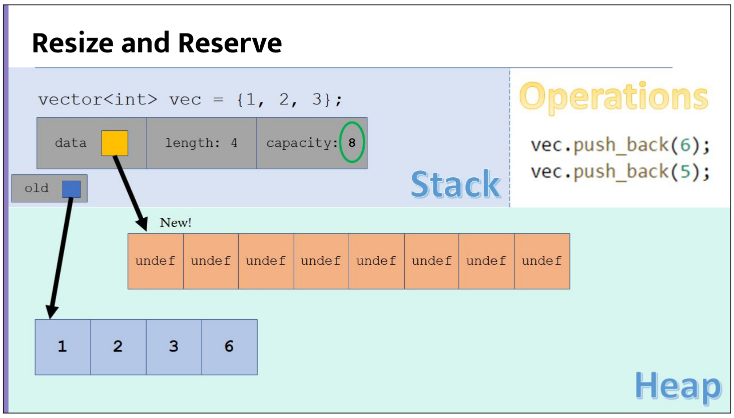


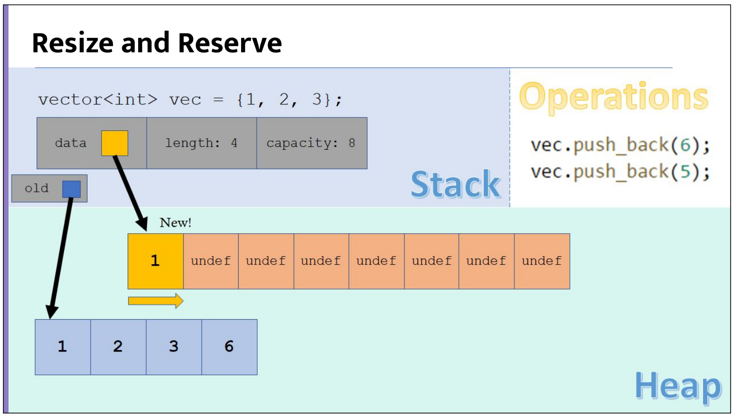
Resize and Reserve

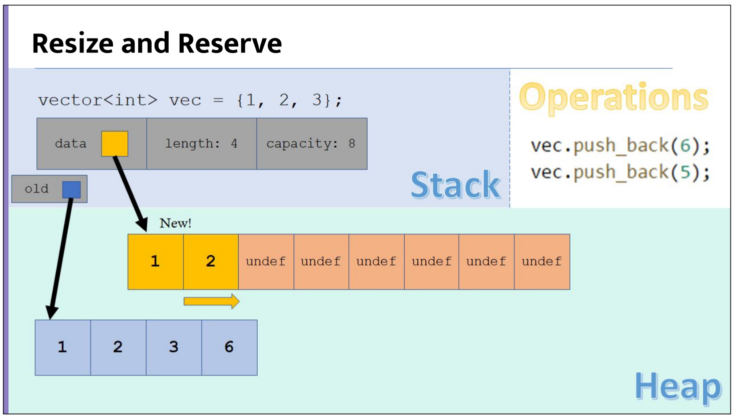


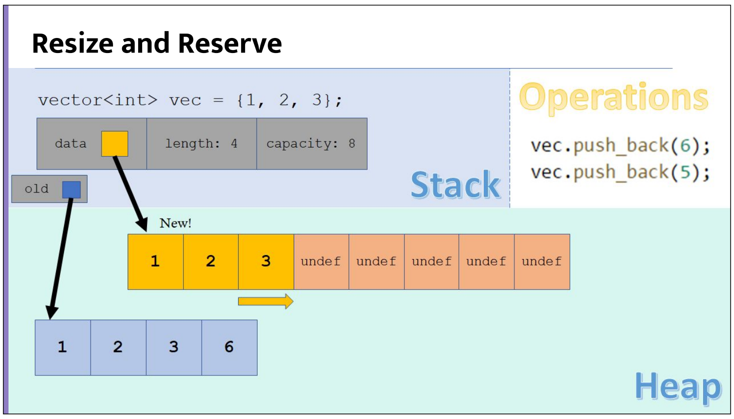
Resize and Reserve

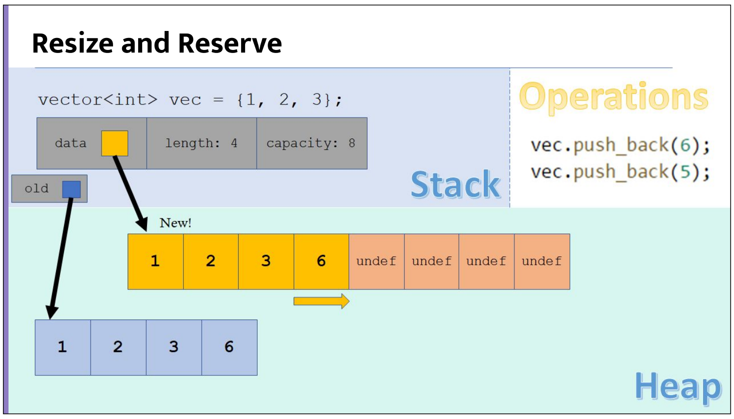


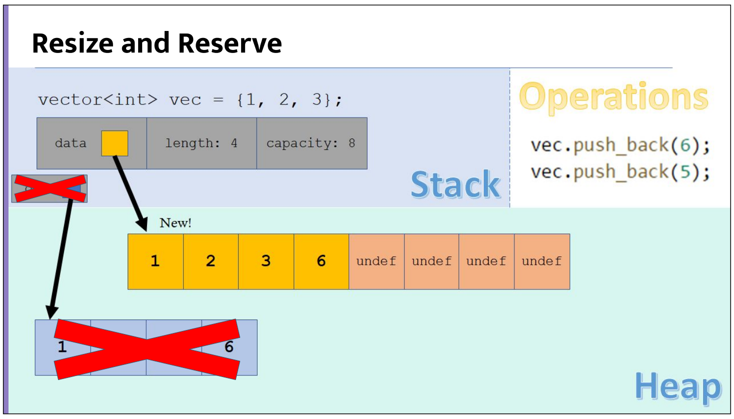

Resize and Reserve
vector<int> vec = {1, 2, 3};
data
length: 4
capacity: 8
old
Stack
New!
undef undef undef undef undef undef undef undef
1 2 3 6
Heap
Operations
vec.push_back(6);
vec.push_back(5);

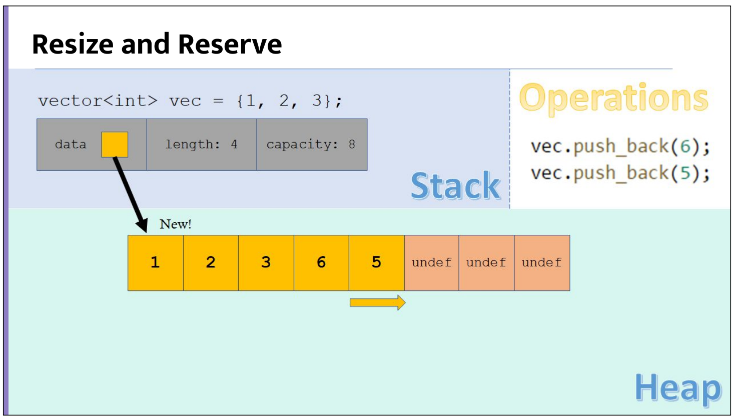

Resize and Reserve
vector<int> vec = {1, 2, 3};
data
length: 4
capacity: 8
old
Stack
New!
1 undef undef undef undef undef undef undef
1 2 3 6
Heap
Operations
vec.push_back(6);
vec.push_back(5);

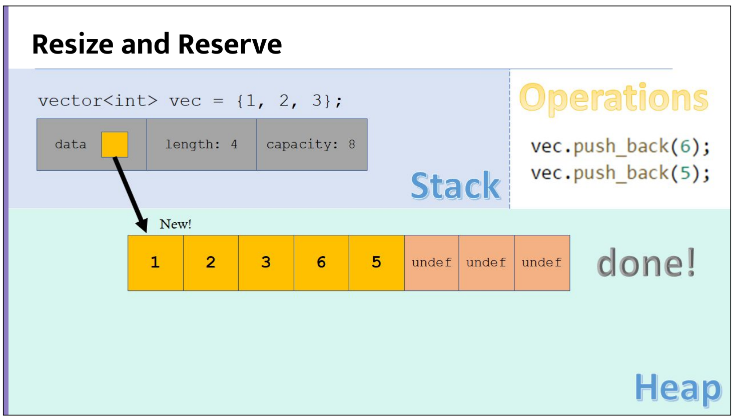

Resize and Reserve
vector<int> vec = {1, 2, 3};
data
length: 4
capacity: 8
old
Stack
New!
1 2 undef undef undef undef undef undef
1 2 3 6
Heap
Operations
vec.push_back(6);
vec.push_back(5);


Resize and Reserve
vector<int> vec = {1, 2, 3};
data
length: 4
capacity: 8
old
Stack
New!
1 2 3 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);

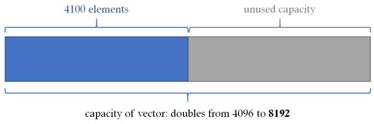

Resize and Reserve
vector<int> vec = {1, 2, 3};
data
length: 4
capacity: 8
Stack
New!
1 2 3 6 undef undef undef undef
1 6
Heap
Operations
vec.push_back(6);
vec.push_back(5);


Resize and Reserve
vector<int> vec = {1, 2, 3};
data
length: 4
capacity: 8
Stack
New!
1 2 3 6 5 undef undef undef
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
New!
1 2 3 6 5 undef undef undef
done!
Heap
Operations
vec.push_back(6);
vec.push_back(5);

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!



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				

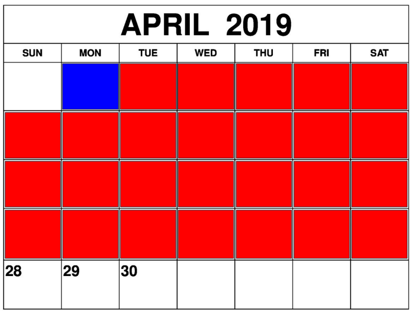
Amortized Complexity

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



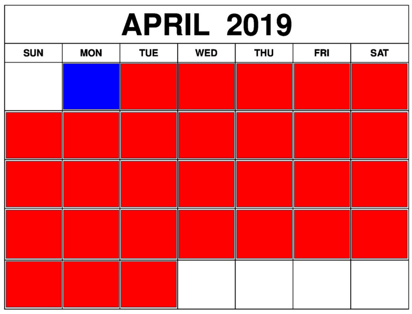
Amortized Complexity

4/30 Total Cost: **still $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

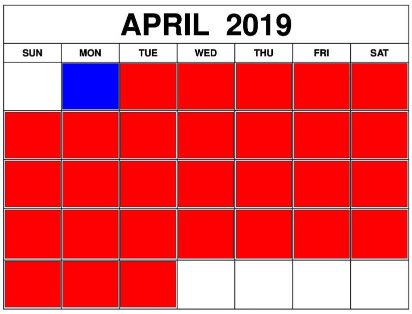
For the remaining 29 days, you don't have to pay anything



Amortized Complexity

4/30 Total Cost: **still $600**

- You paid $600 rent on 4/1
  - But is the daily cost of living in the apt $600? No!
  - With a $600 payment, you can live in the apt for ~29 days without paying!
- After amortization, you are actually spending $20 per day, despite it seeming like much more earlier in the month.



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 Θ(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 Θ(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

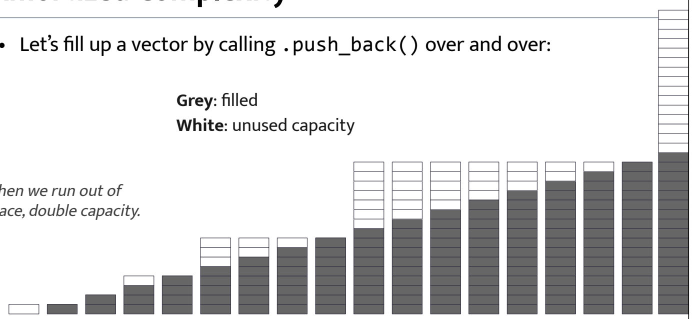
When we run out of space, double capacity.

Amortized Complexity

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

Grey: filled
White: unused capacity

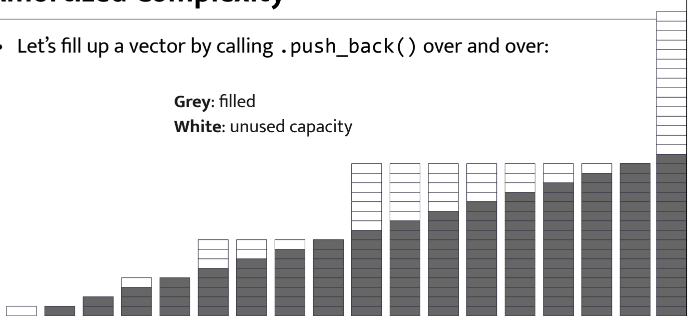
When we run out of space, double capacity.



Amortized Complexity

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

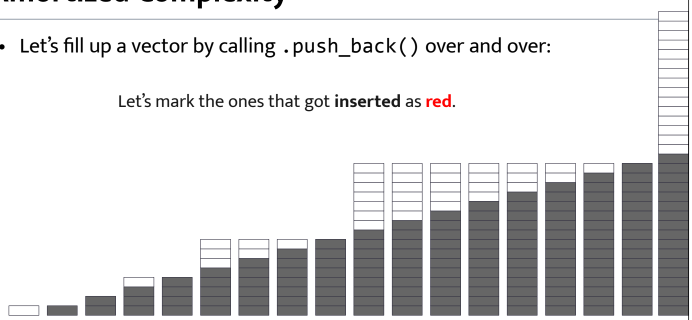
Grey: filled
White: unused capacity



Amortized Complexity

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

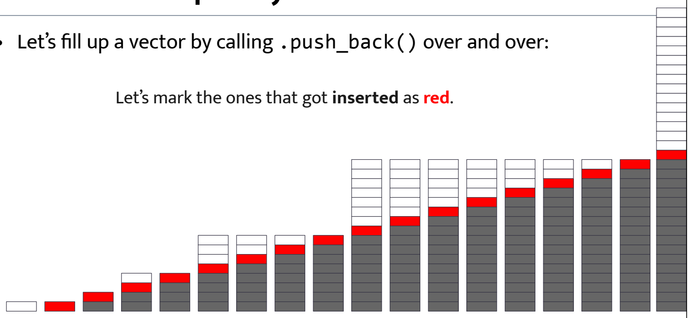
Let's mark the ones that got **inserted** as **red**.



Amortized Complexity

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

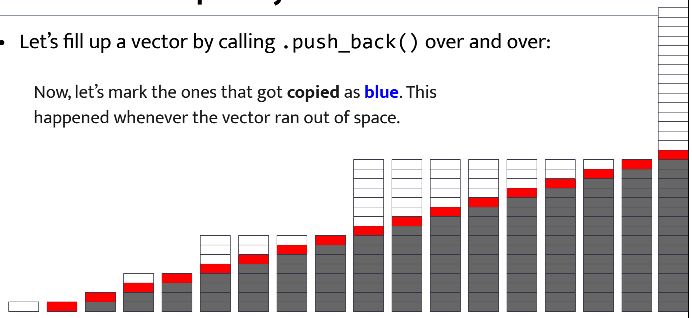
Let's mark the ones that got **inserted** as **red**.



Amortized Complexity

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

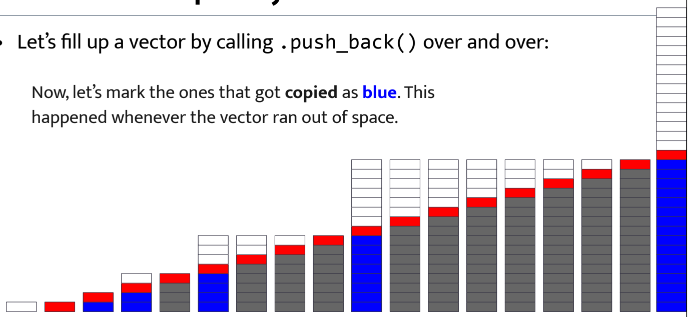
Now, let's mark the ones that got **copied** as **blue**. This happened whenever the vector ran out of space.



Amortized Complexity

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

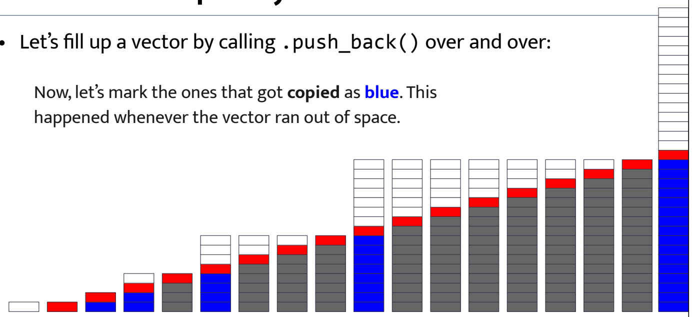
Now, let's mark the ones that got **copied** as **blue**. This happened whenever the vector ran out of space.



Amortized Complexity

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

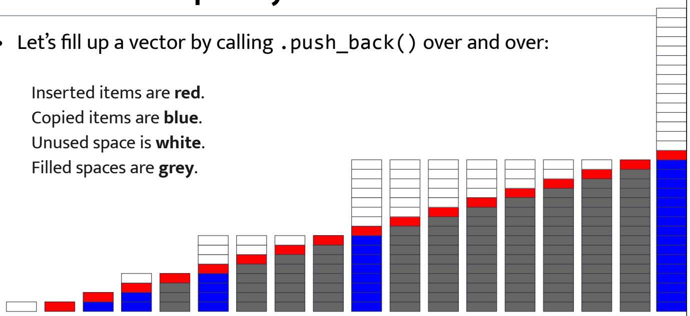
Now, let's mark the ones that got **copied** as **blue**. This happened whenever the vector ran out of space.



Amortized Complexity

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

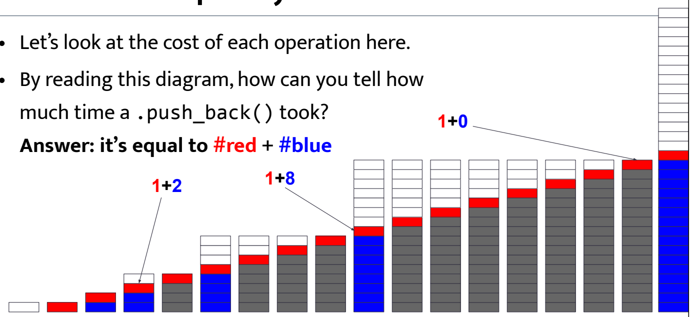
Inserted items are **red**.
Copied items are **blue**.
Unused space is **white**.
Filled spaces are **grey**.



Amortized Complexity

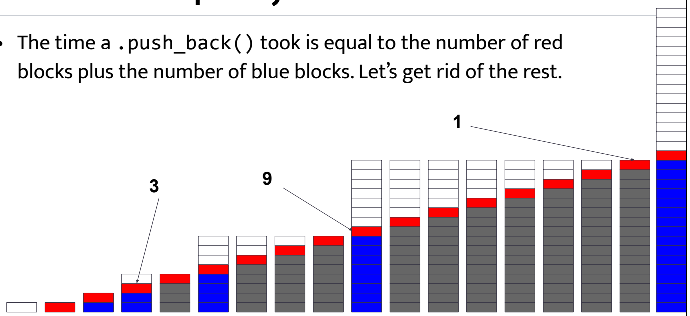
- Let's look at the cost of each operation here.
- By reading this diagram, how can you tell how much time a `.push_back()` took?

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



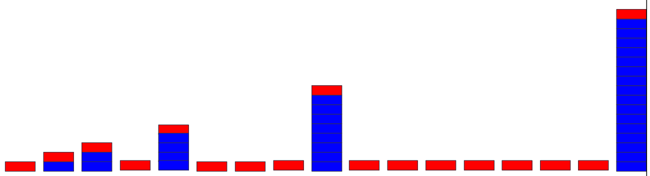
Amortized Complexity

- 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.



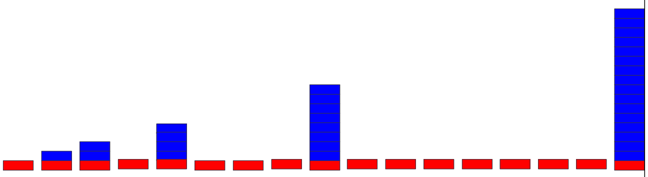
Amortized Complexity

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



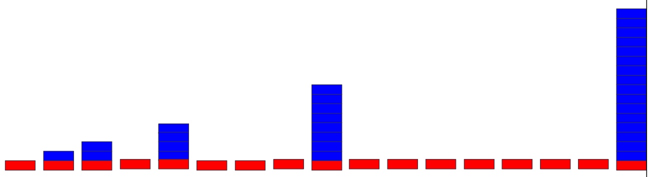
Amortized Complexity

- 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.



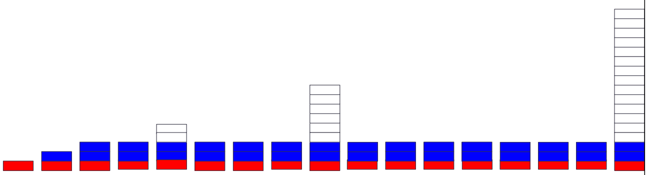
Amortized Complexity

- 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.



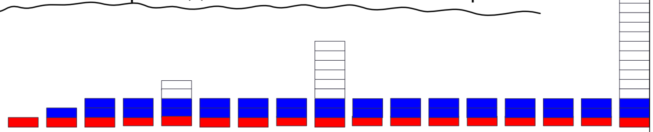
Amortized Complexity

- 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

- 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 ⟶ 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 ⟶ 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 $\Theta(n)$!

Pointer and Iterator Invalidation

- A final comment: strings, vectors, and other "auto-resizing" containers may reallocate their memory and 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!

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.

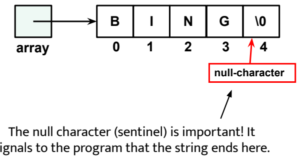
C-Strings and C++ Strings

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';
    ```



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 vs 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);
```
```