Stack Implementations

Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan

Announcements

Recap

Stack Implementations:
- Array
- Vector
- Linked Chain

Stack ADT

```
#ifndef STACK_H_
#define STACK_H_

template<class T>
class Stack
{

public:
    Stack();
    void push(const T& new_entry); // adds an element to top of stack
    void pop(); // removes element from top of stack
    T top() const; // returns a copy of element at top of stack
    int size() const; // returns the number of elements in the stack
    bool isEmpty() const; // returns true if no elements on stack false otherwise

private:
          //implementation details here


};     //end Stack

#include "Stack.cpp"
#endif // STACK_H_`
```

ADT vs Data Structure

ADT is the logical/abstract description of the organization and operations on the data

Data Structure is the representation/implementation of the ADT

We may have multiple implementations of the same ADT
- 1 ADT
- Multiple Data Structures

To complicate matters, a data structure may be implemented using other data structures
- stack implemented using queue
- priority queue implemented using heap (more on this later)

Choose a Data Structure

Array?

Vector?

Linked chain?

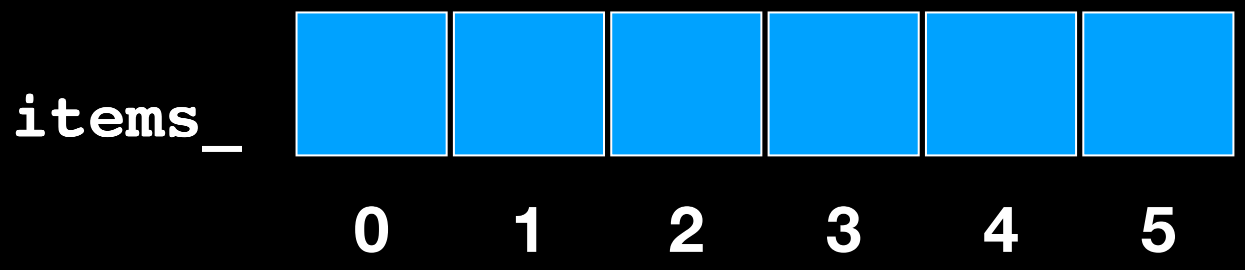
Choose a Data Structure

Inserting and removing from same end (**LIFO**)

Goal: minimize work - Ideally $O(1)$

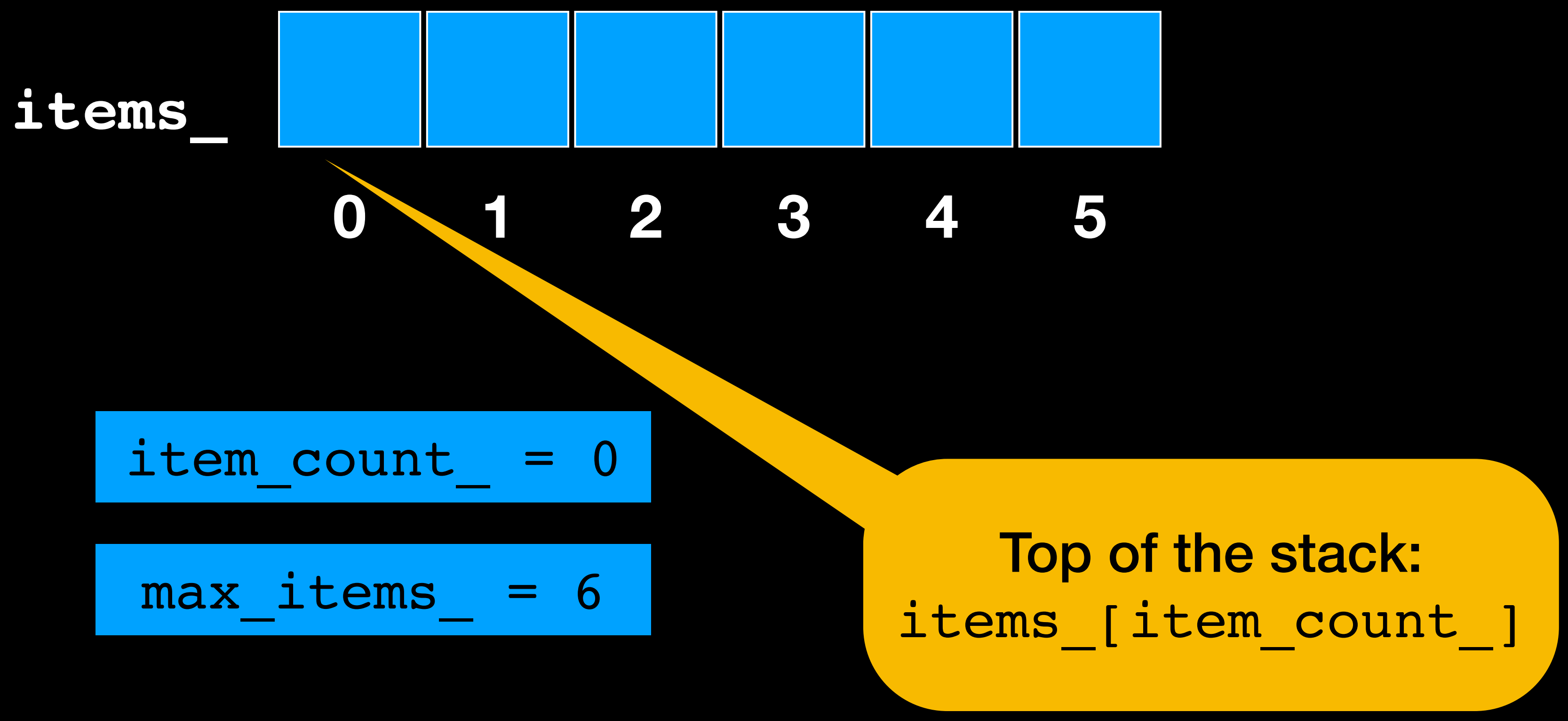
What would you suggest?

Array

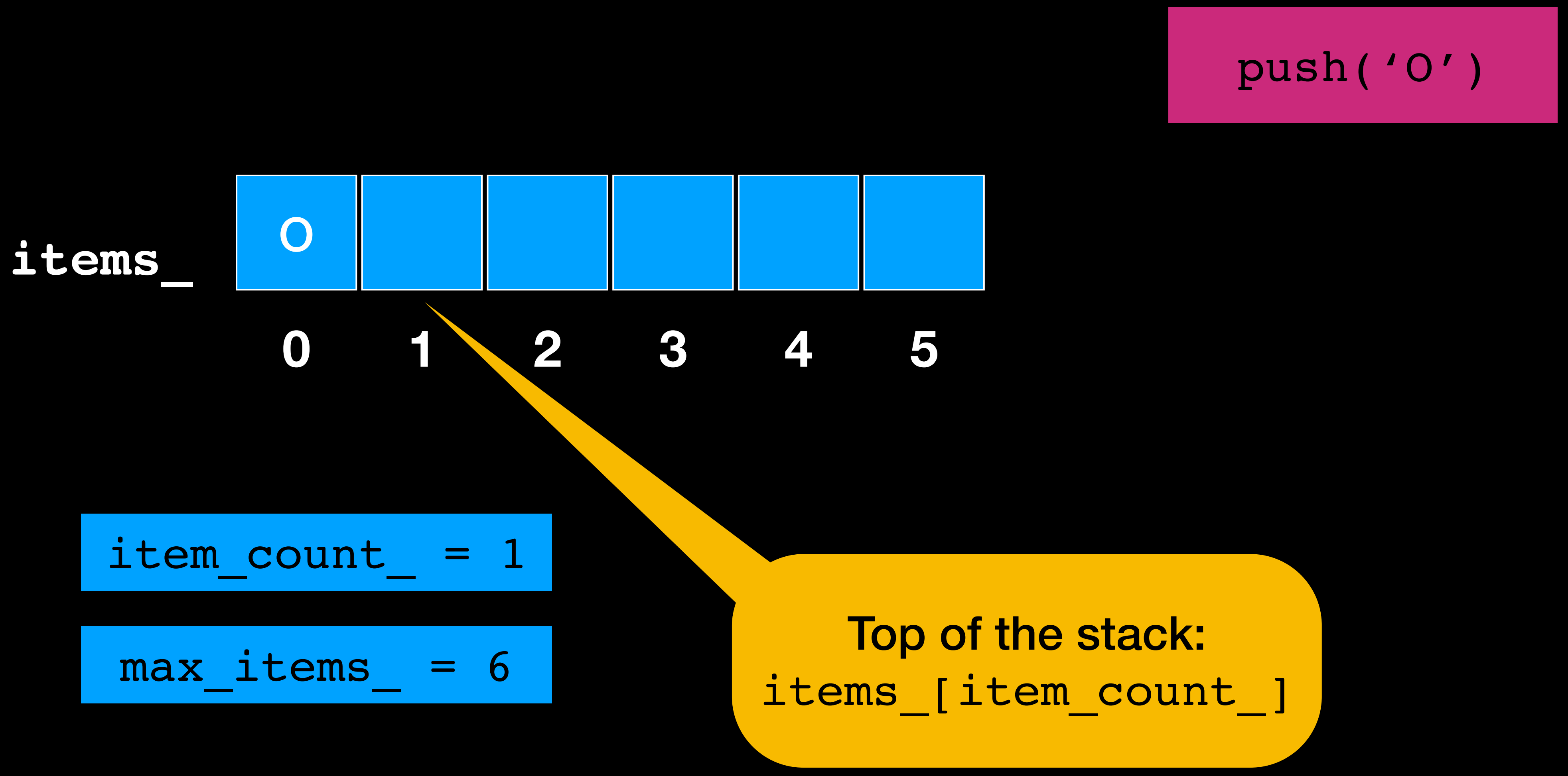


Where is the top of the stack?

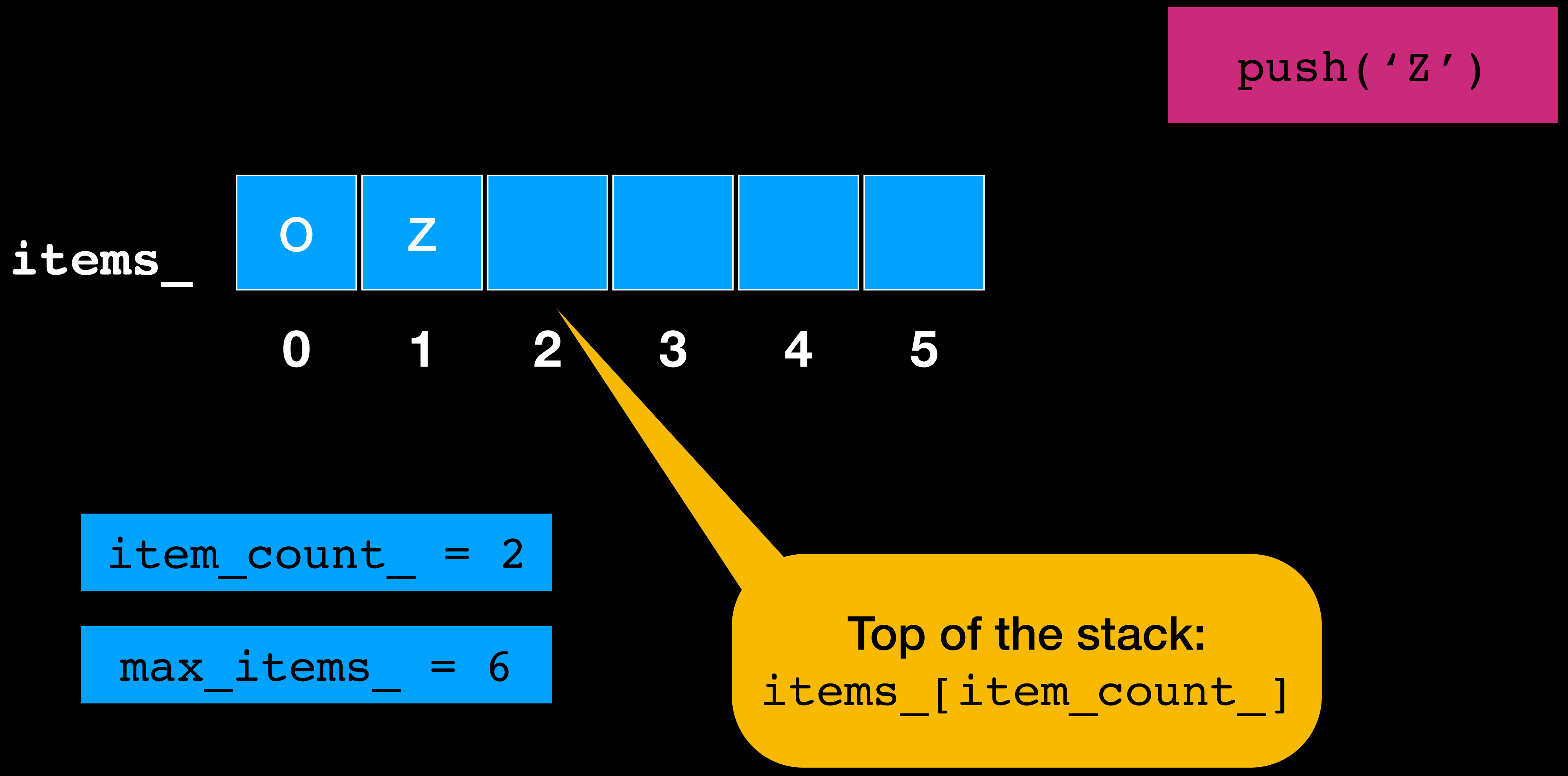
Array


items_
0
1
2
3
4
5
item_count_ = 0
max_items_ = 6
Top of the stack:
items_[item_count_]

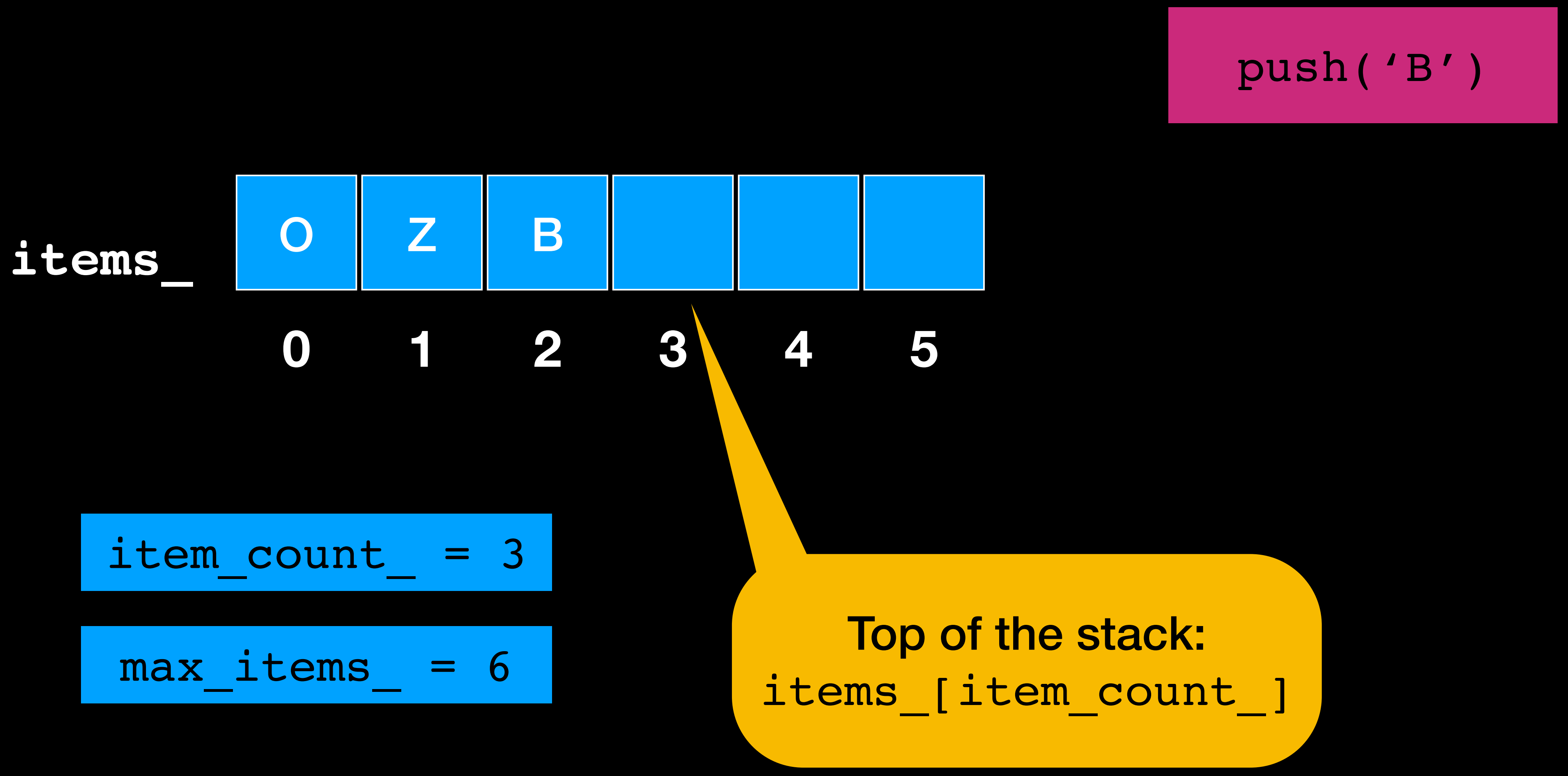
Array


push('O')
items_
O
0
1
2
3
4
5
item_count_ = 1
max_items_ = 6
Top of the stack:
items_[item_count_]

Array


push('Z')
items_
O
Z
0
1
2
3
4
5
item_count_ = 2
max_items_ = 6
Top of the stack:
items_[item_count_]

Array


push('B')
items_
O
Z
B
0
1
2
3
4
5
item_count_ = 3
max_items_ = 6
Top of the stack:
items_[item_count_]

Array

pop()

items_	O	Z				
	0	1	2	3	4	5

item_count_ = 2

max_items_ = 6

Top of the stack:
items_[item_count_]

Pops
items_[item_count_ -1]

Array Analysis

1 assignment + 1 increment/decrement = O(1)

`size` : O(1)

`isEmpty`: O(1)

`push`: O(1)

`pop` : O(1)

`top` : O(1)

GREAT!!!!

Array Analysis

1 assignment + 1 increment/decrement = O(1)

`size` : O(1)

`isEmpty`: O(1)

`push`: O(1)

`pop` : O(1)

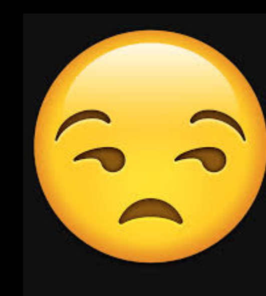
`top` : O(1)

GREAT???

Array

push('T')

items_	O	Z	B	Y	L	P
	0	1	2	3	4	5



Sorry Stack is Full!!!

`item_count_ = 6`

`max_items_ = 6`

Top of the stack:
items_[item_count_]

Vector

```
std::vector<T> some_vector;
```

So what is a vector really?

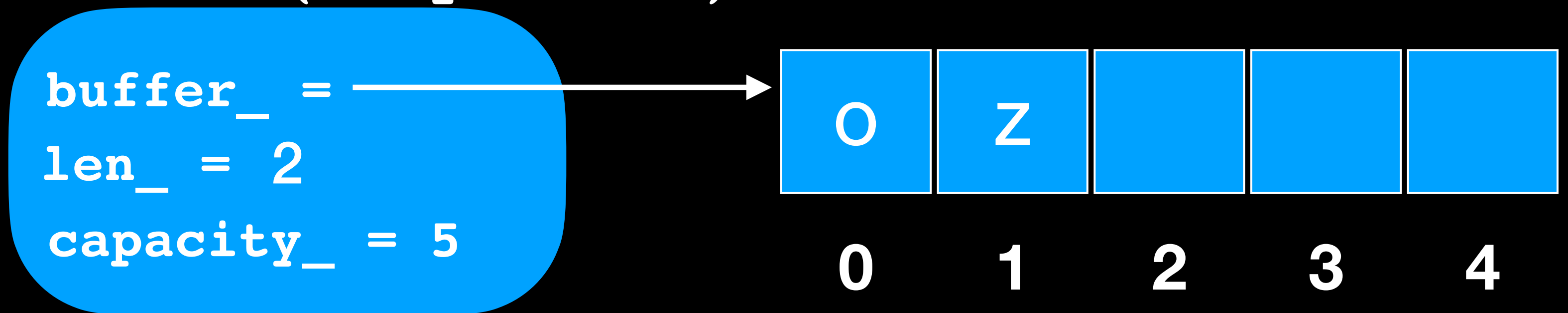
Vector

```
std::vector<T> some_vector;
```

So what is a vector really?

Push and pop same as with arrays

Vector (simplified)



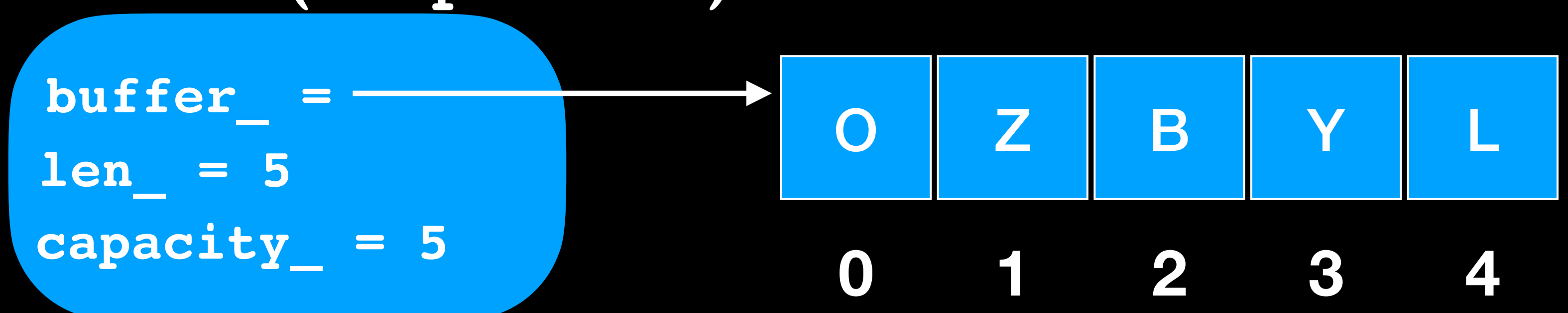
Vector

```
std::vector<T> some_vector;
```

So what is a vector really?

Stack is Full?

Vector (simplified)

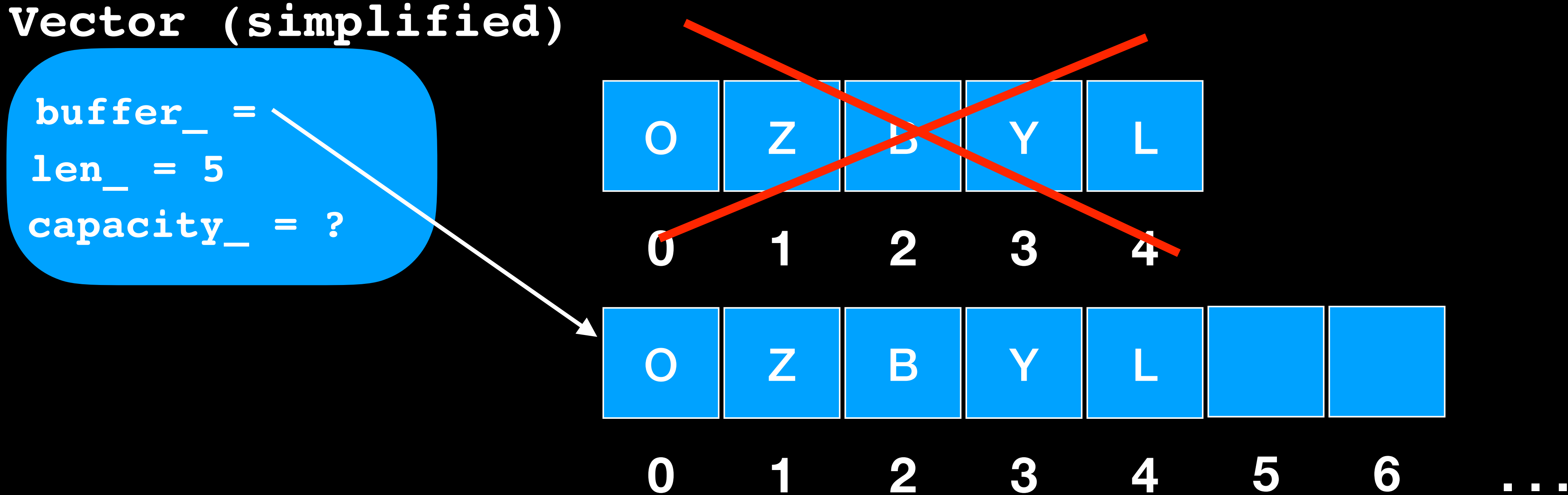


Vector

```
std::vector<T> some_vector;
```

So what is a vector really?

No, I'll Grow!!!



Lecture Activity

How much should it grow?

Write an explanation arguing the pros and cons of growing by the amount you propose

Vector Analysis

1 assignment + 1 increment/decrement = O(1)

`size` : O(1)

`isEmpty`: O(1)

`push`: O(1)

`pop` : O(1)

`top` : O(1)

GREAT!!!!

Vector Analysis

1 assignment + 1 increment/decrement = O(1)

`size` : O(1)

`isEmpty`: O(1)

`push`: O(1)

`pop` : O(1)

`top` : O(1)

GREAT??? → Except when stack is full must:

- allocate new array
- copy elements in new array
- delete old array

Vector Analysis

1 assignment + 1 increment/decrement = O(1)

`size` : O(1)

`isEmpty`: O(1)

`push`: O(1)

`pop` : O(1)

`top` : O(1)

GREAT??? →

Except when stack is full must:

- allocate new array O(1)
- copy elements in new array O(n)
- delete old array O(1)

How should Vector grow?

Sometimes $O(1)$

Sometimes $O(n)$

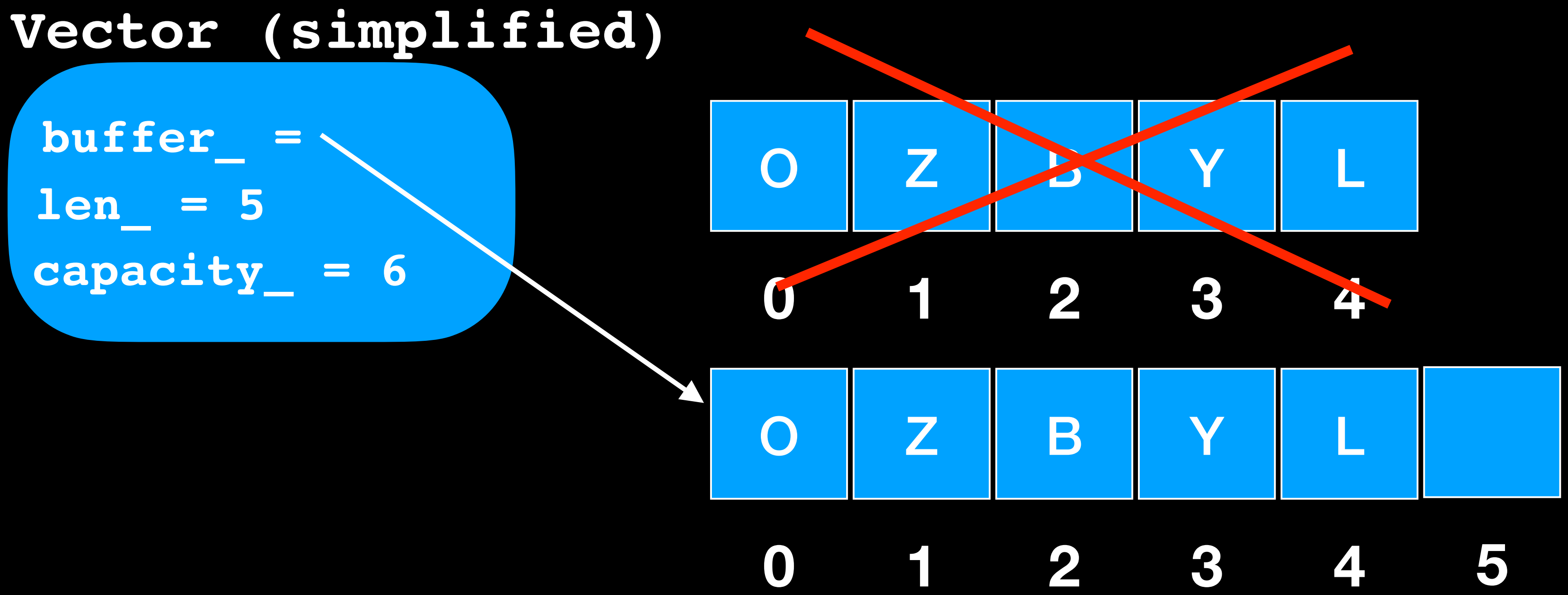
Consider behavior over several pushes

Vector Growth: a naive approach

```
std::vector<T> some_vector;
```

So what is a vector really?

I'll Grow!!!
I will add space for the item to be added



Vector Growth: a naive approach

If vector **grows by 1 each time**, every push costs n "*steps*"

Cost of pushes:

$1 + 2 + 3 + 4 + 5 + . . . + n$

$= n (n+1)/2$

Vector Growth: a naive approach

If vector **grows by 1 each time**, every push costs n "*steps*"

Cost of n pushes:

$1 + 2 + 3 + 4 + 5 + . . . + n$

$= n (n+1)/2$

$= n^2 /2 + n / 2 \quad O(n^2)$

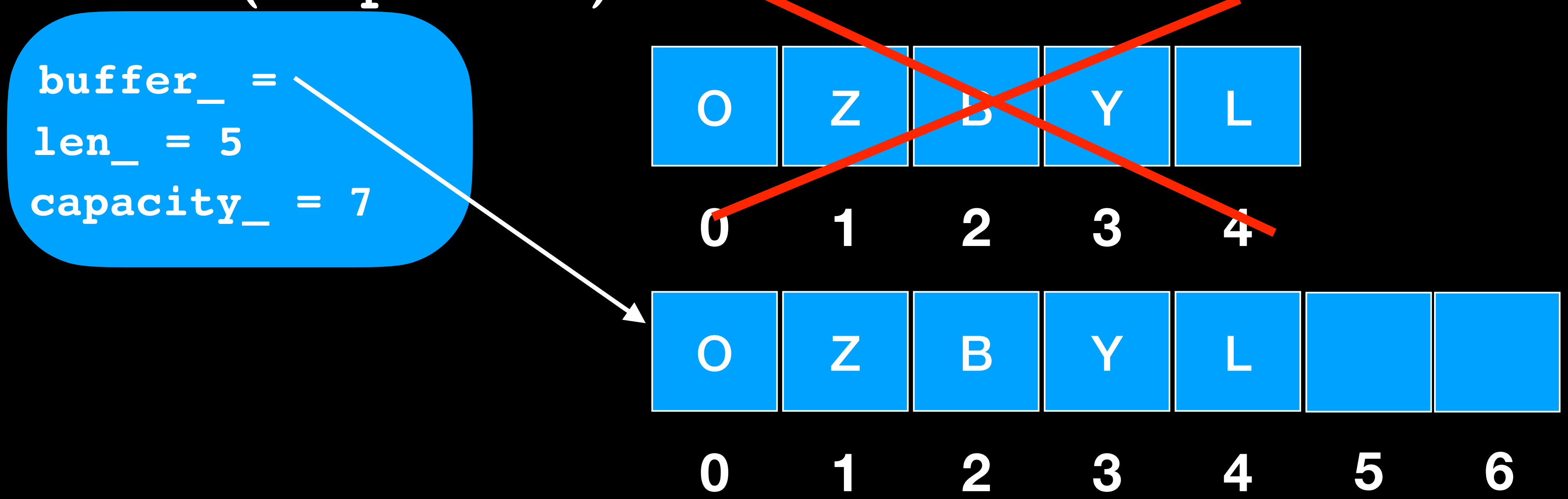
Vector Growth: a better approach

```
std::vector<T> some_vector;
```

So what is a vector really?

I'll Grow!!!
I will add two more slots!

Vector (simplified)



Vector Growth: a better approach

If vector **grows by 2 each time**,

Let a "hard push" be one where the whole vector needs to be copied

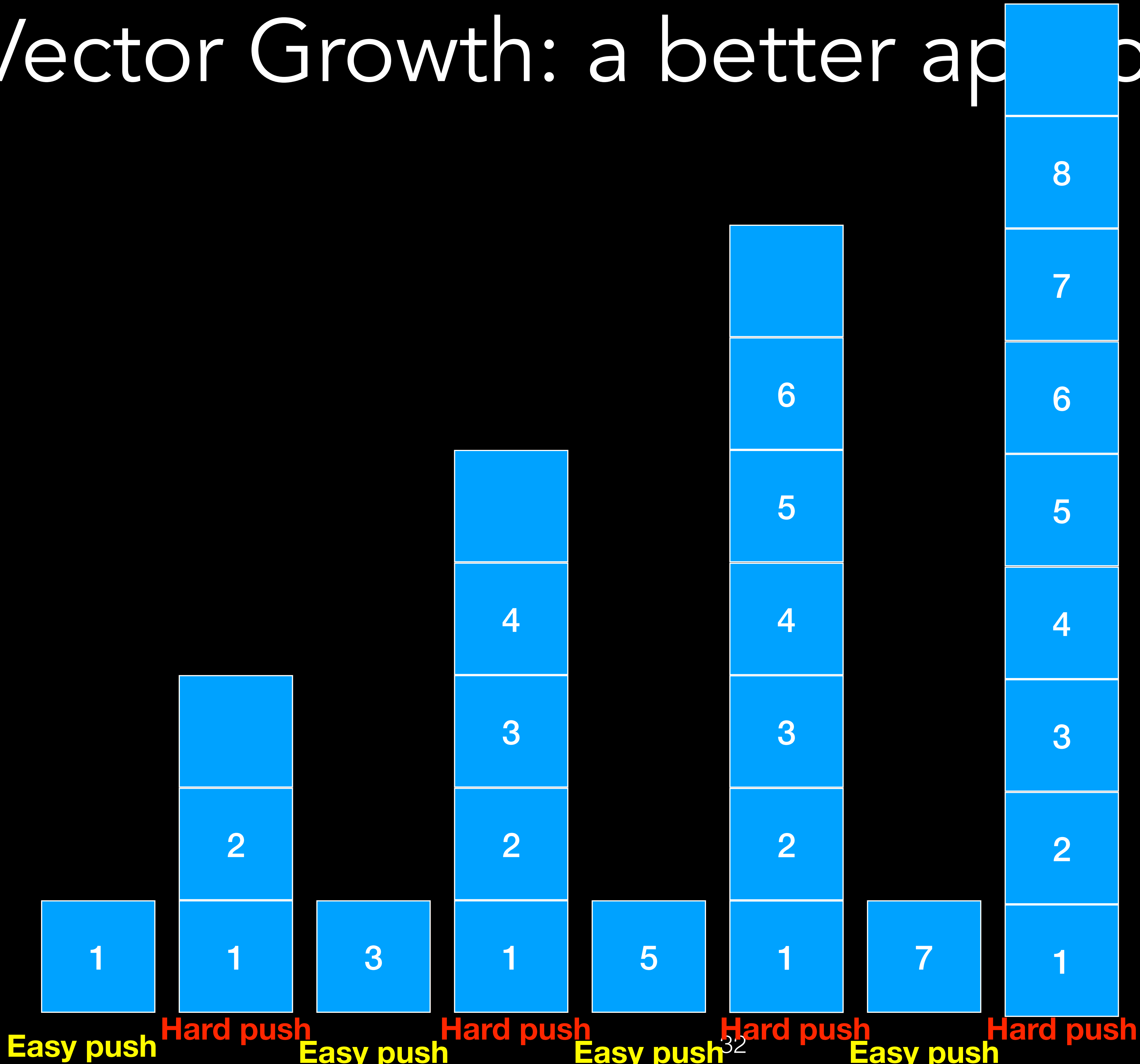
When vector is not copied we have an "easy push"

Now half our pushes will be easy (1 step) and half will be hard (n steps)

So if reconsider the work over several pushes?

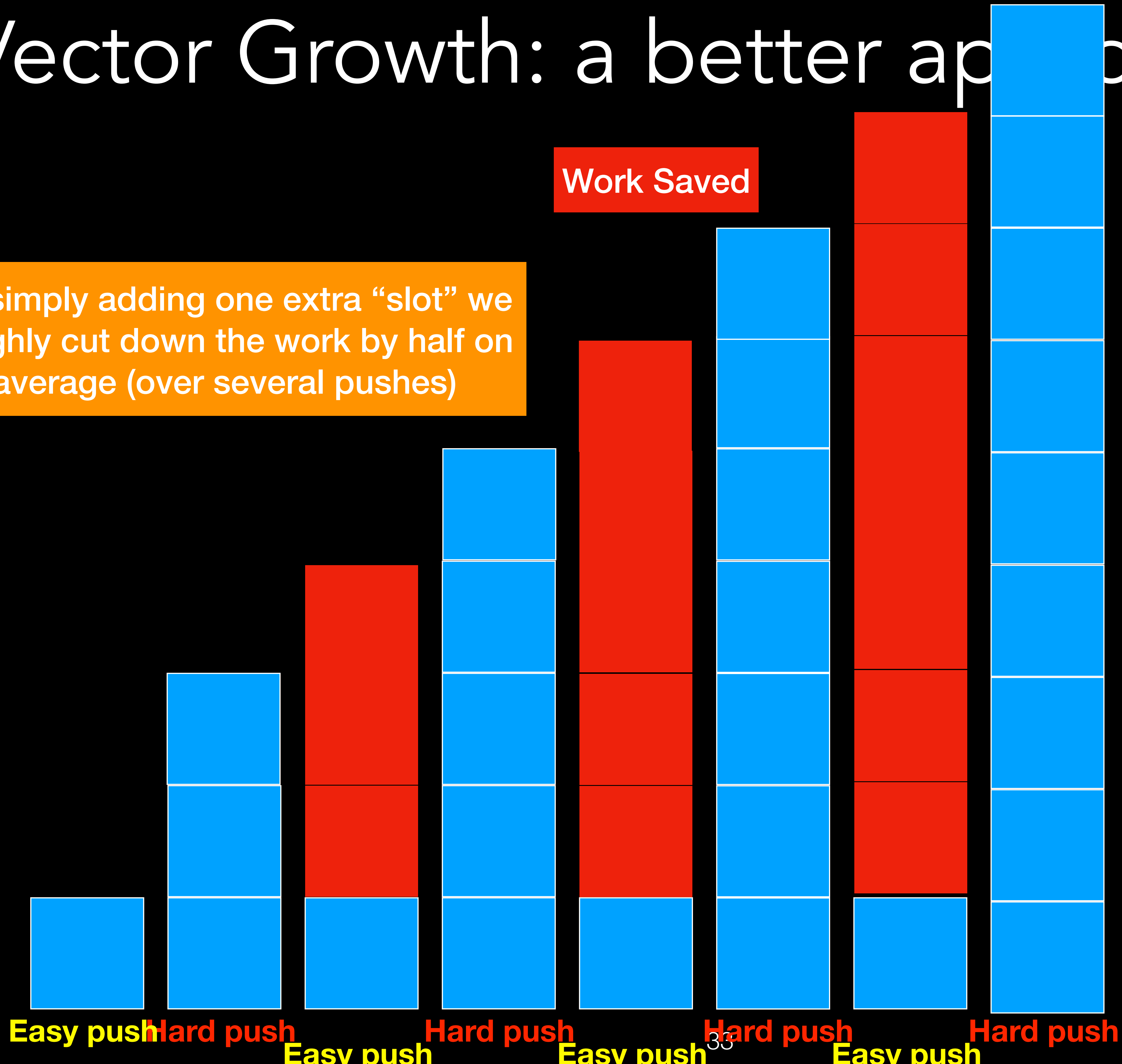
Analysis visualization adapted from Keith Schwarz

Vector Growth: a better approach



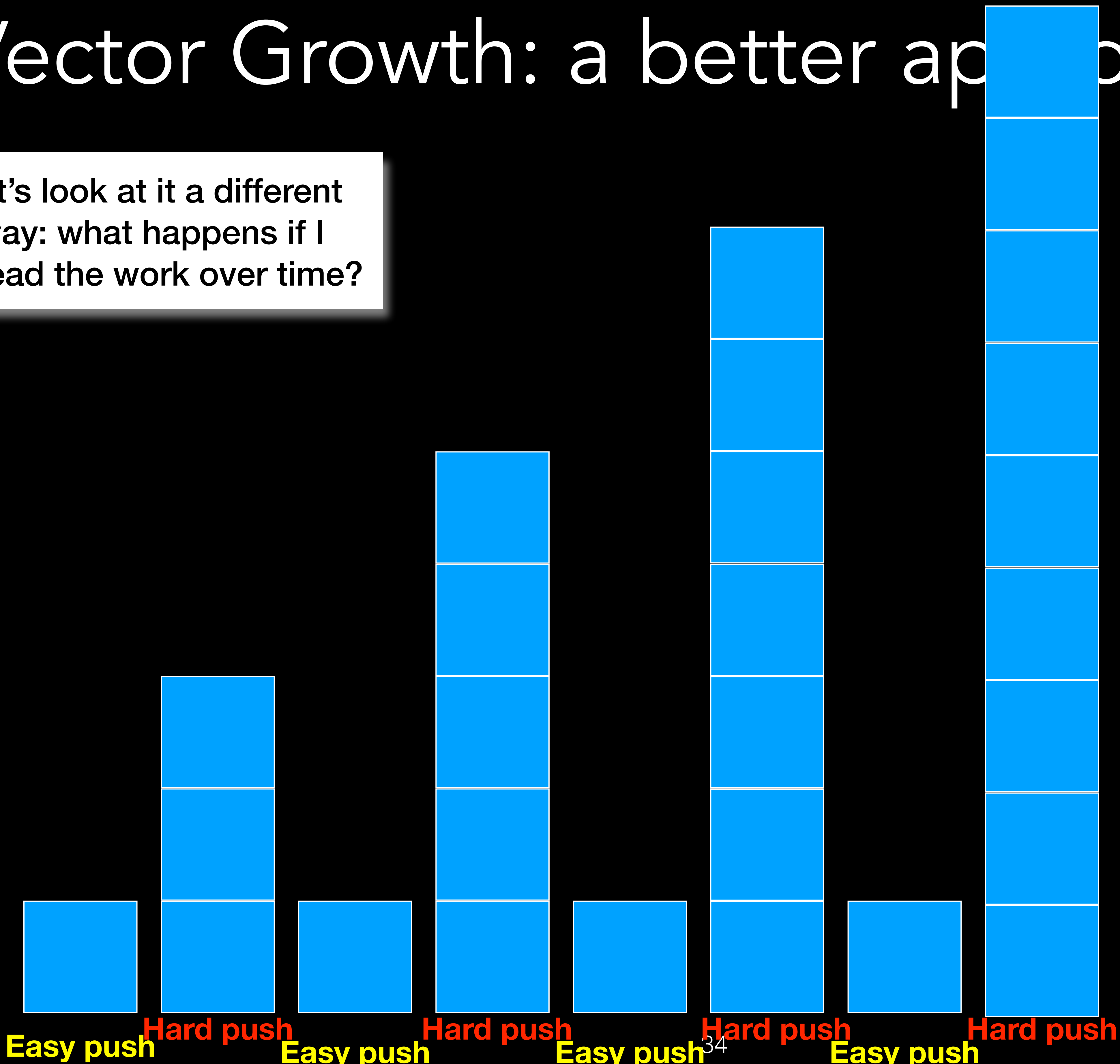
Vector Growth: a better approach

By simply adding one extra “slot” we roughly cut down the work by half on average (over several pushes)



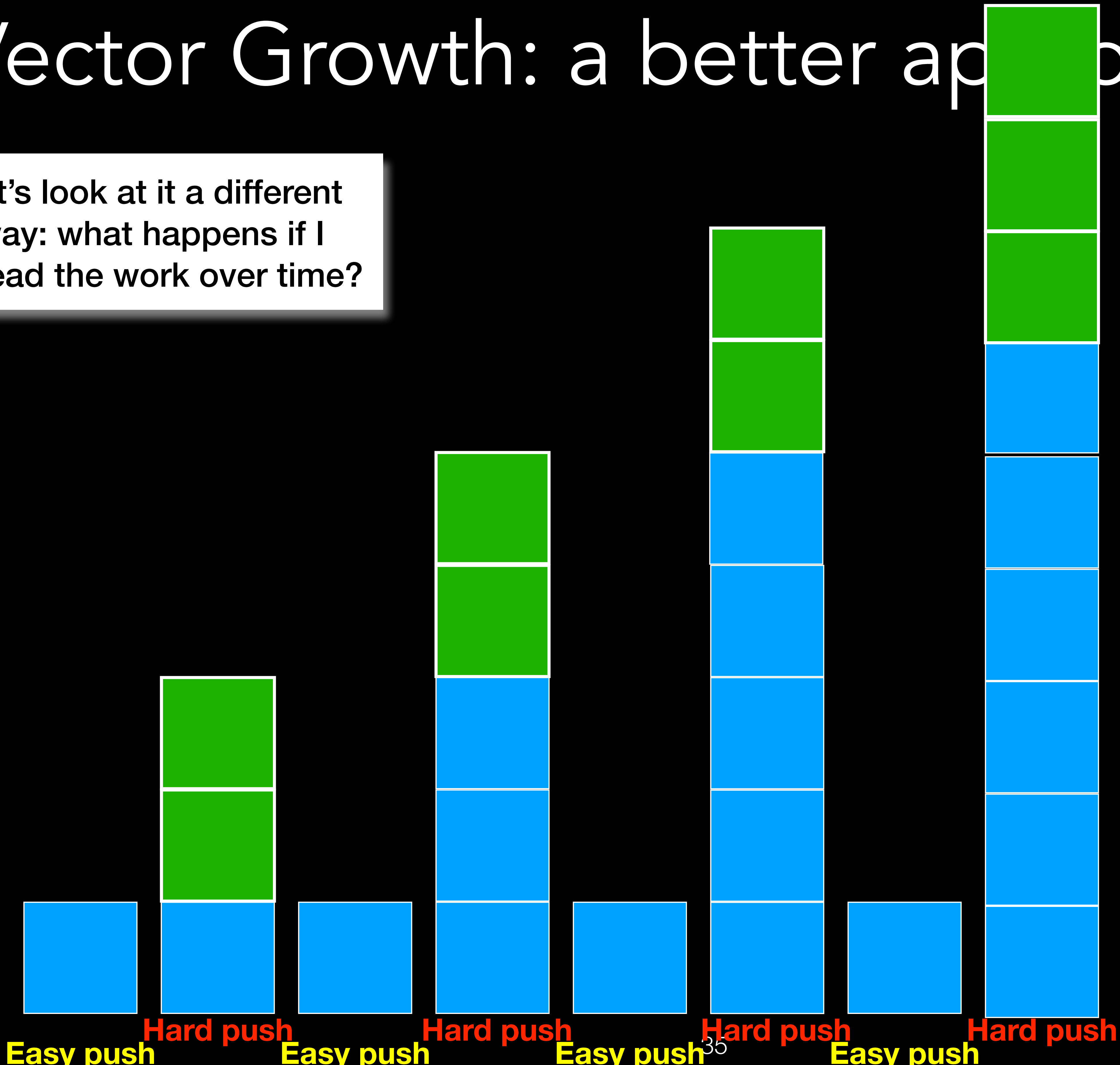
Vector Growth: a better approach

Let's look at it a different way: what happens if I spread the work over time?



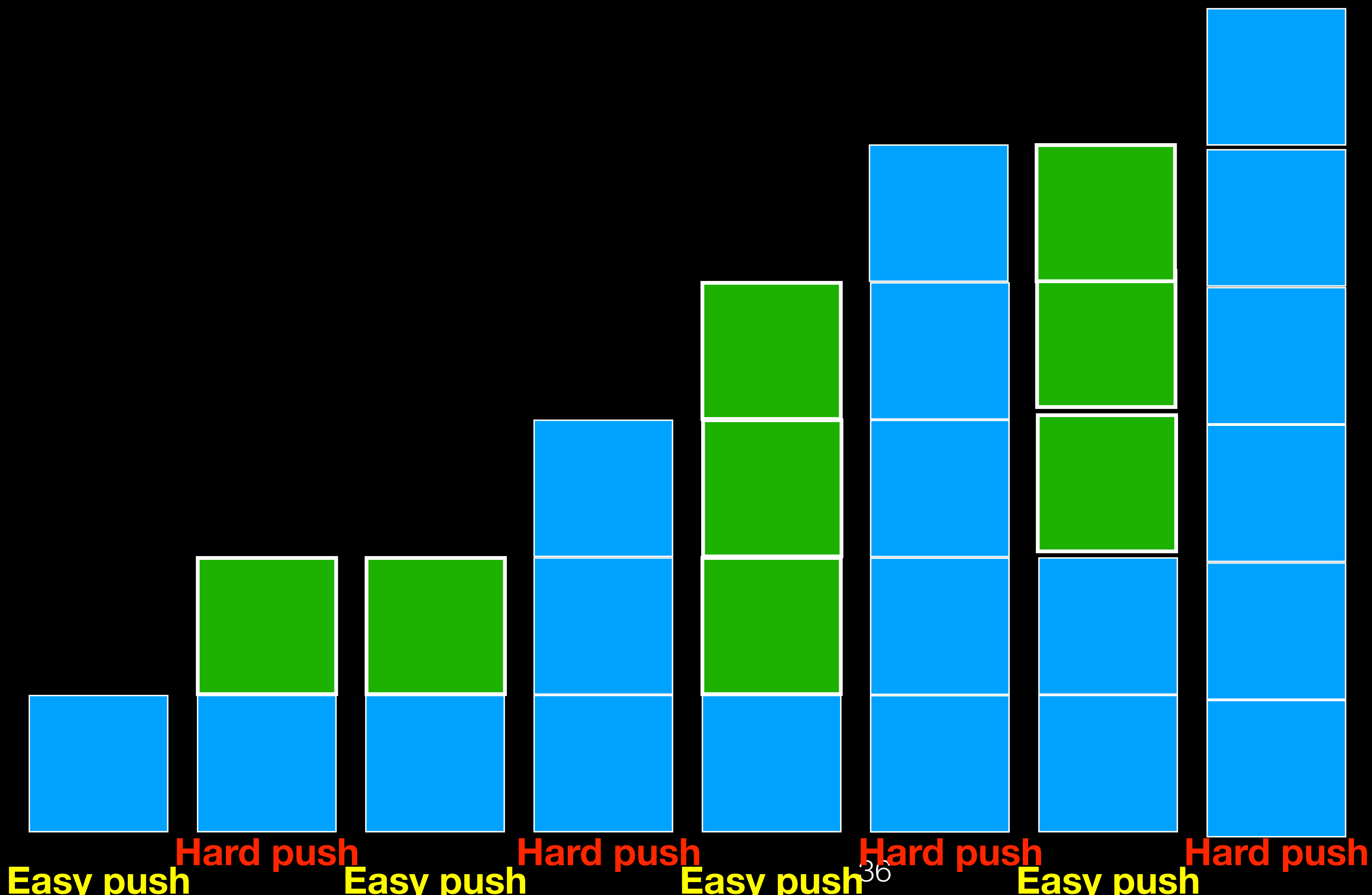
Vector Growth: a better approach

Let's look at it a different way: what happens if I spread the work over time?



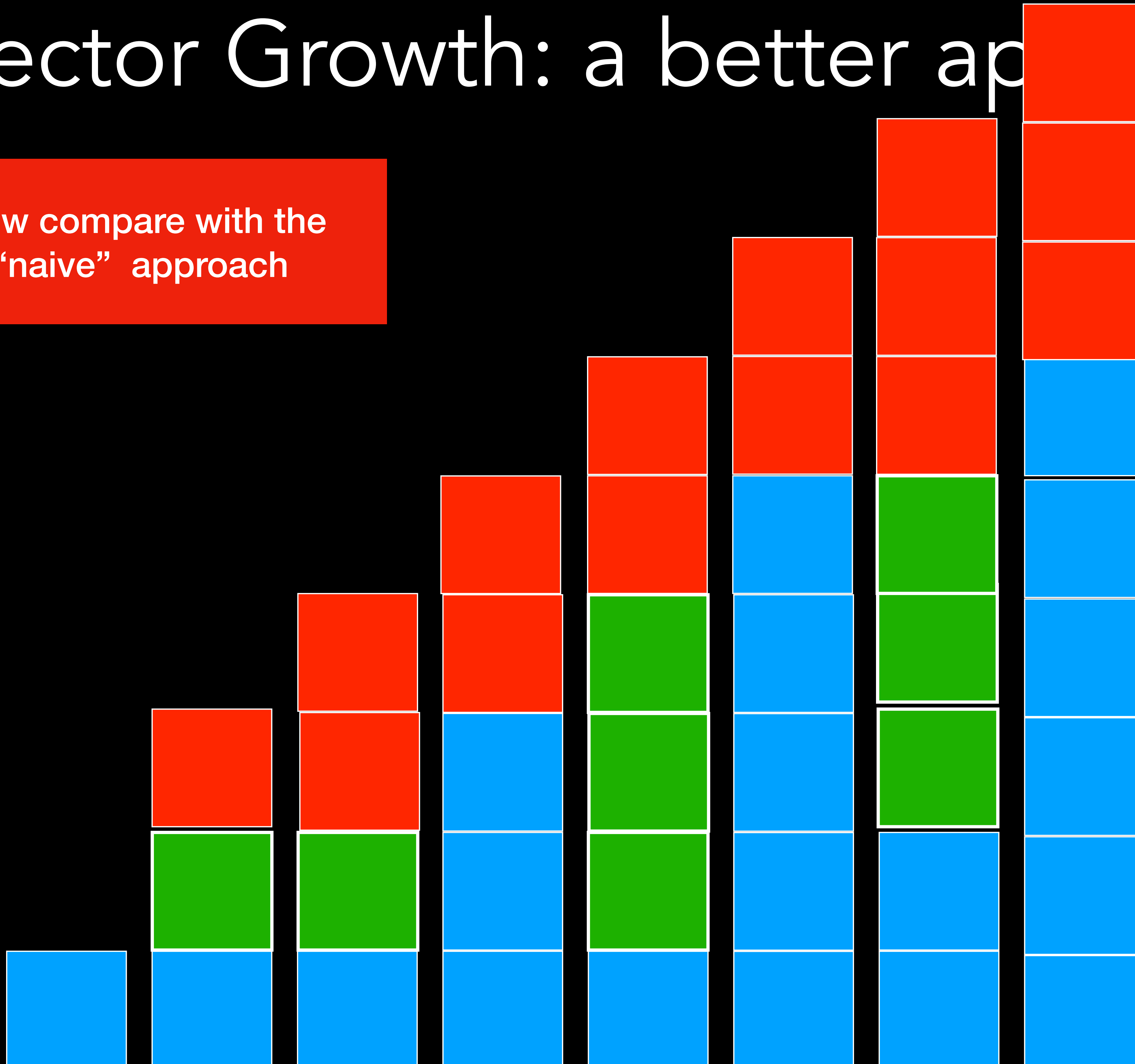
Vector Growth: a better approach

Let's look at it a different way: what happens if I spread the work over time?



Vector Growth: a better approach

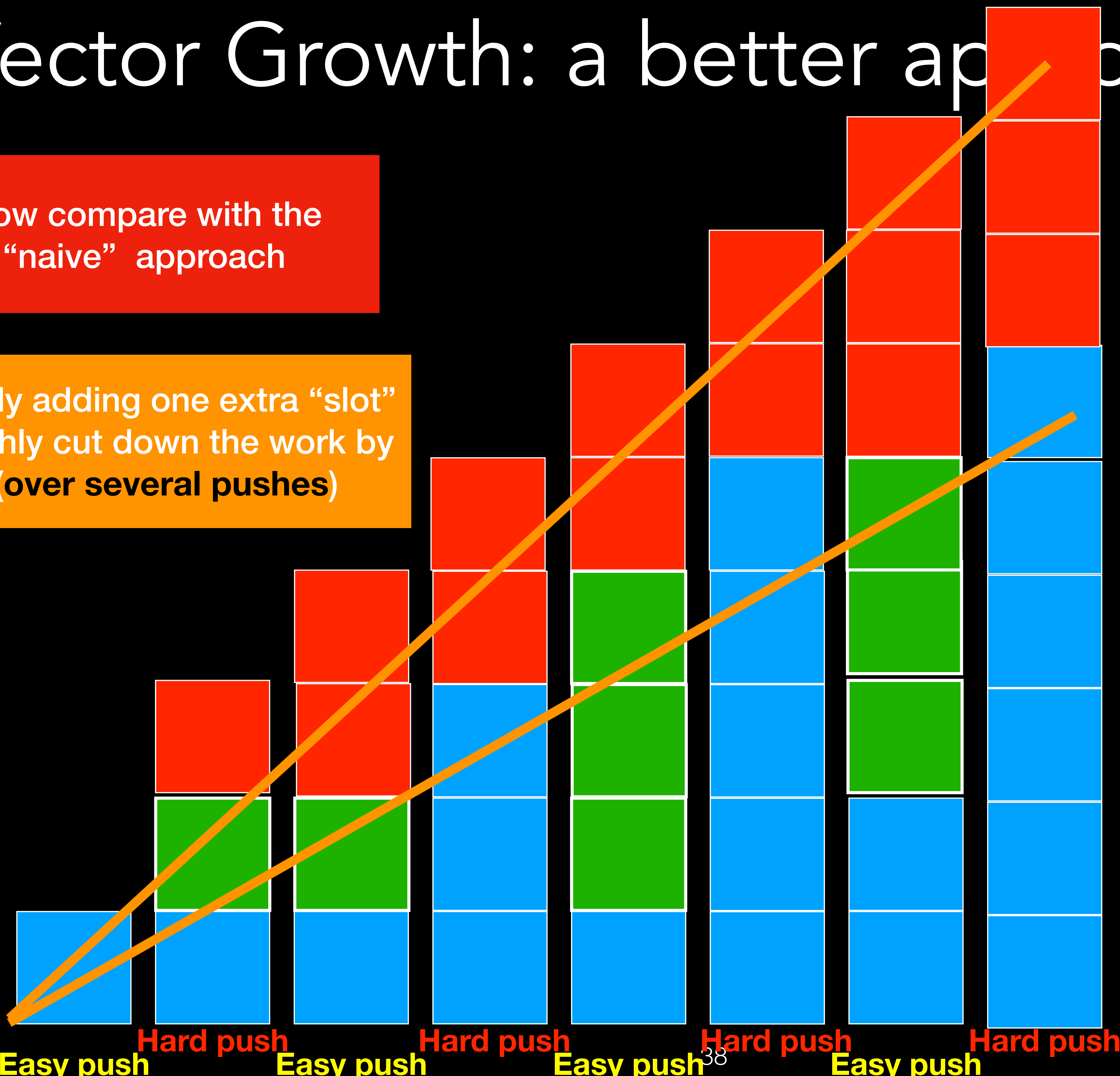
Now compare with the "naive" approach



Vector Growth: a better approach

Now compare with the "naive" approach

By simply adding one extra "slot" we roughly cut down the work by half (**over several pushes**)



Can we do better?

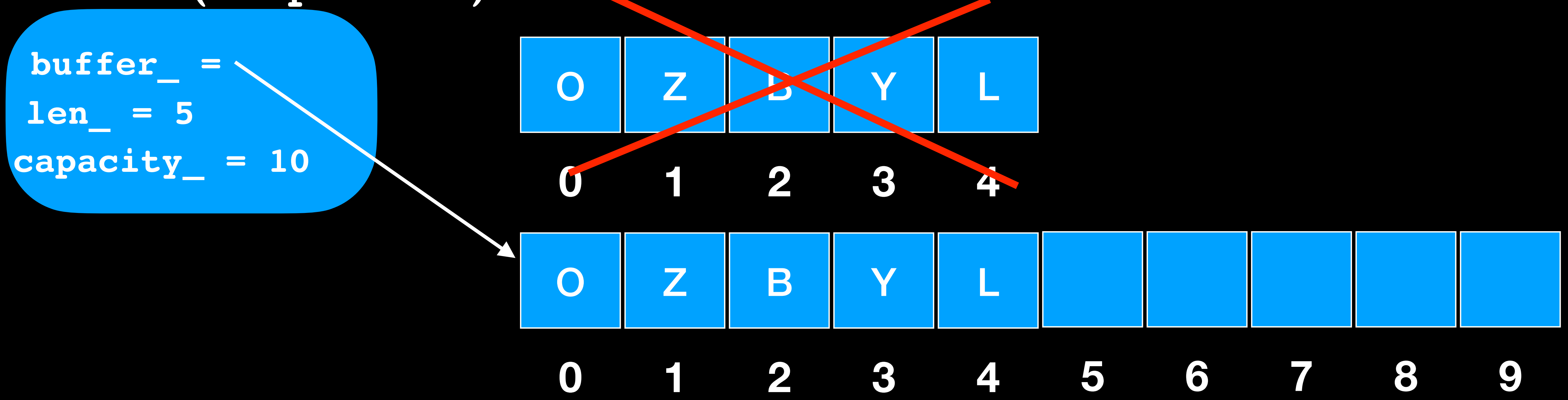
Vector Growth: a much better approach

```
std::vector<T> some_vector;
```

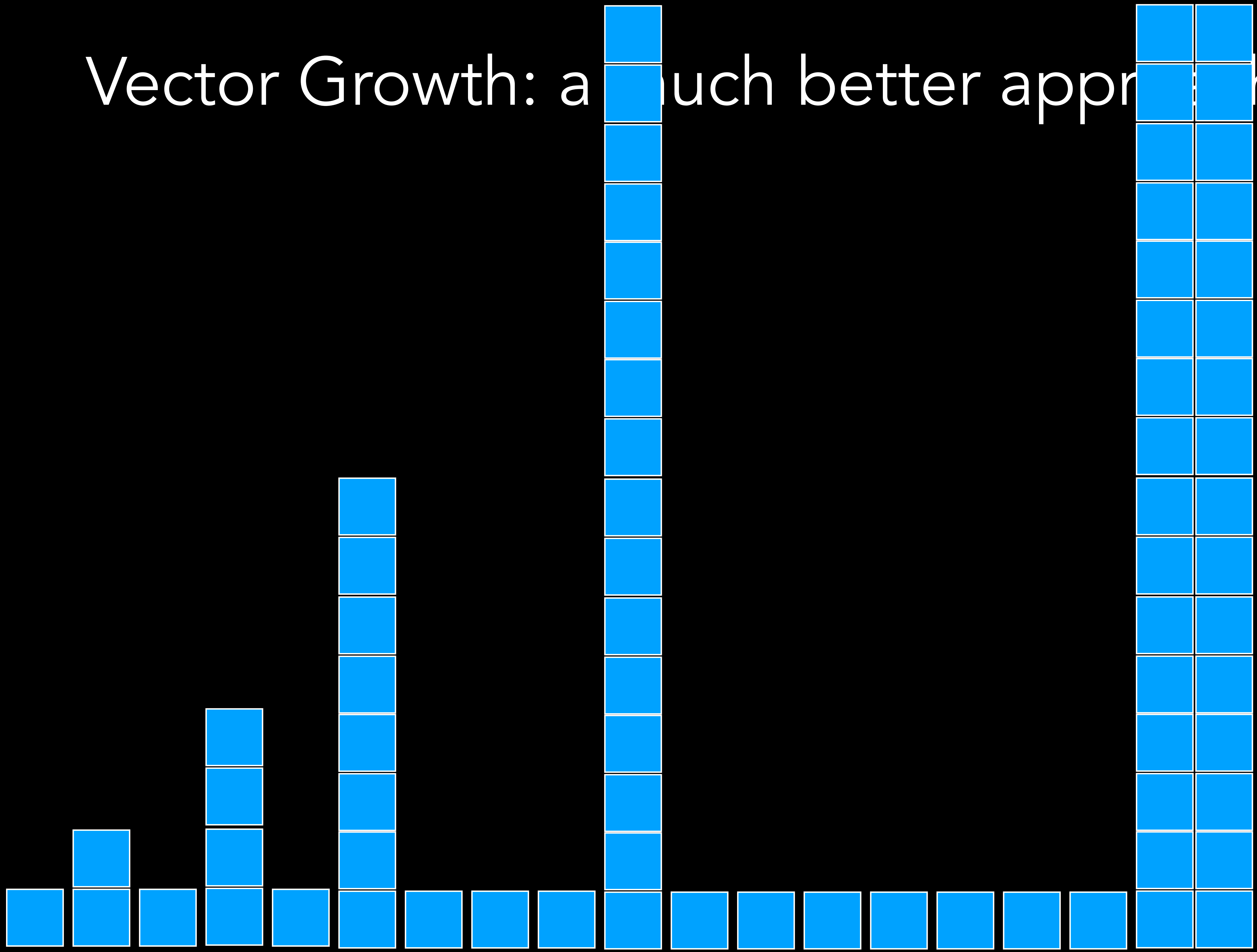
So what is a vector really?

I'll Grow!!!
I'll double my size!

Vector (simplified)

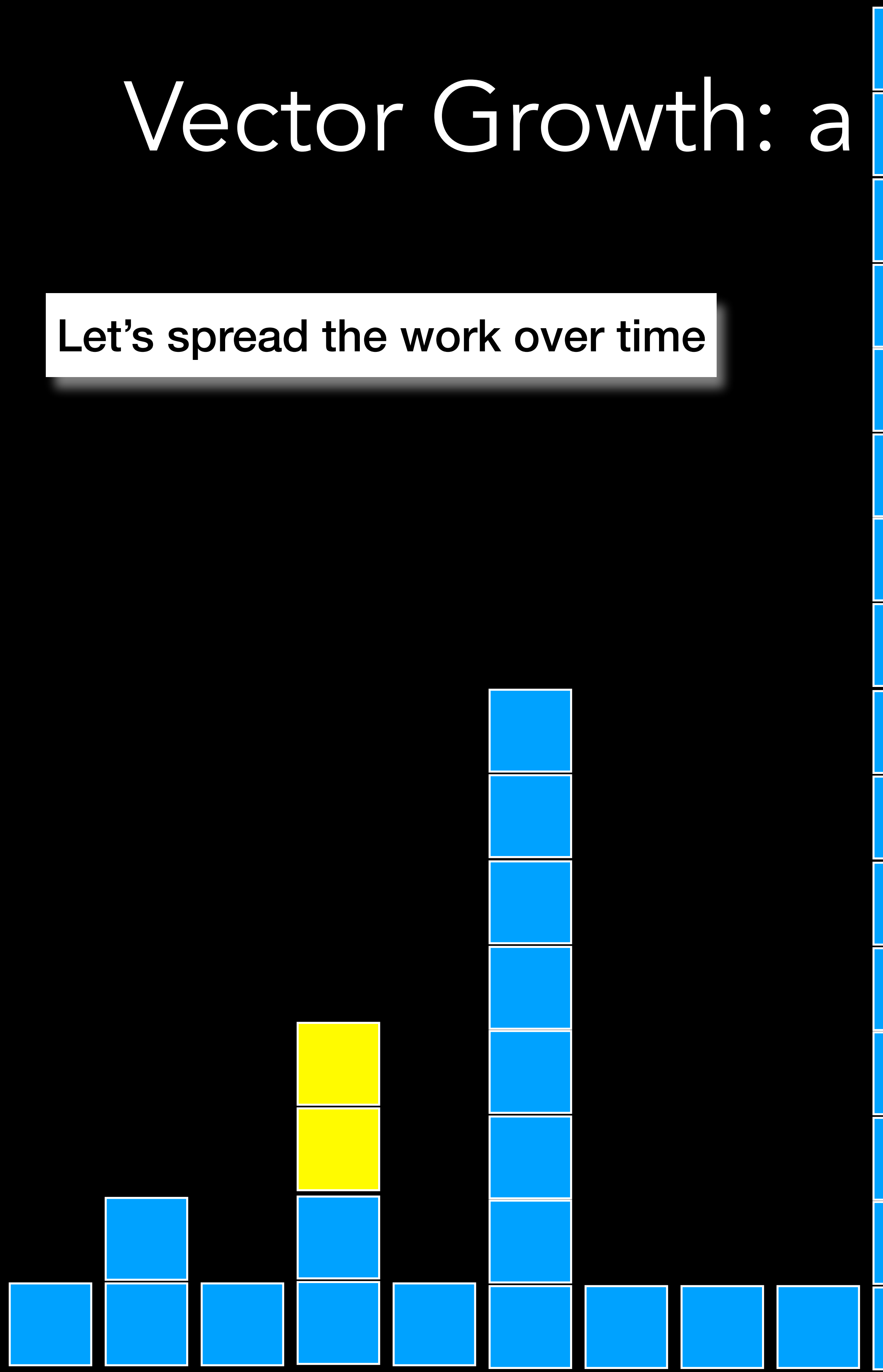


Vector Growth: a much better approach



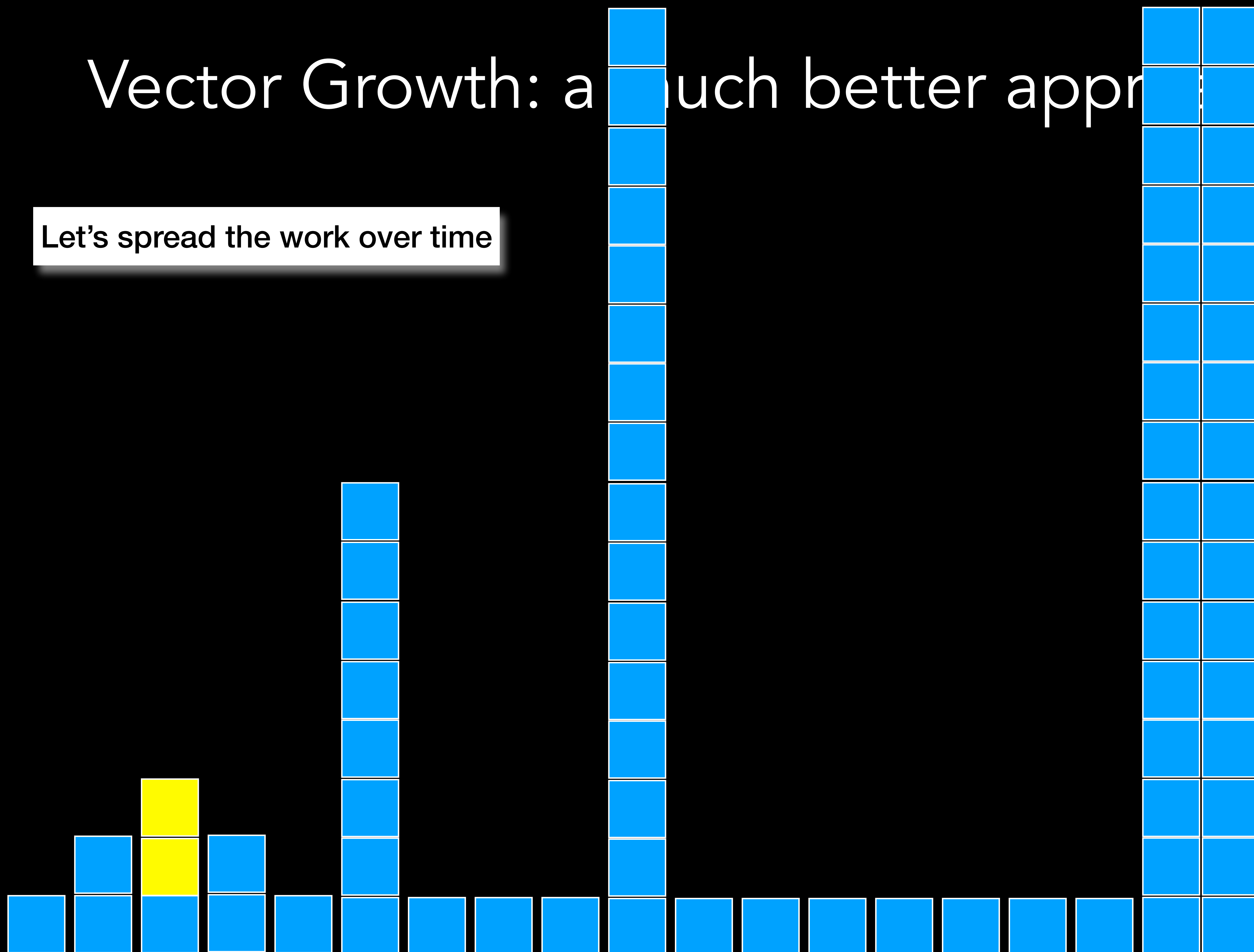
Vector Growth: a much better approach

Let’s spread the work over time



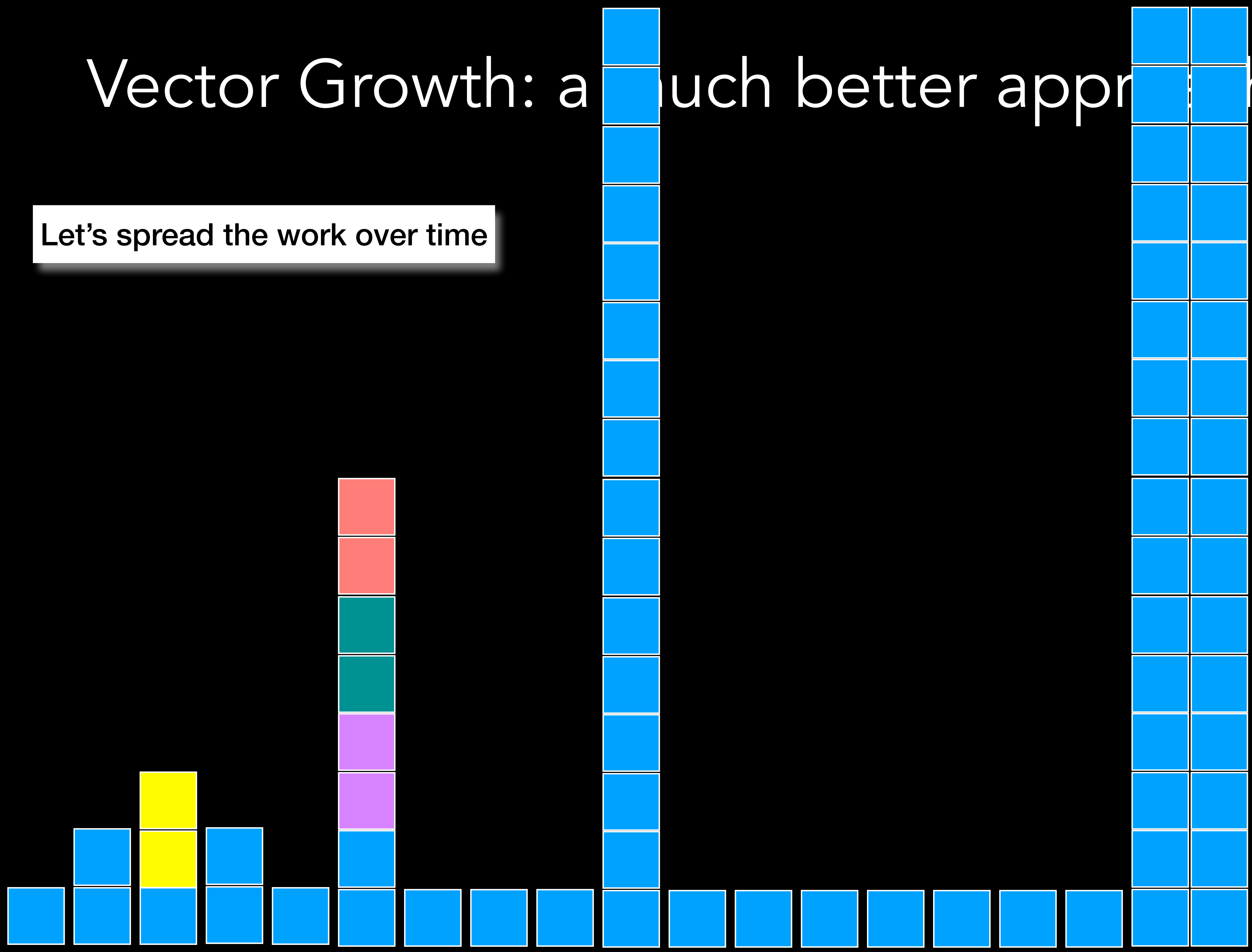
Vector Growth: a much better approach

Let's spread the work over time



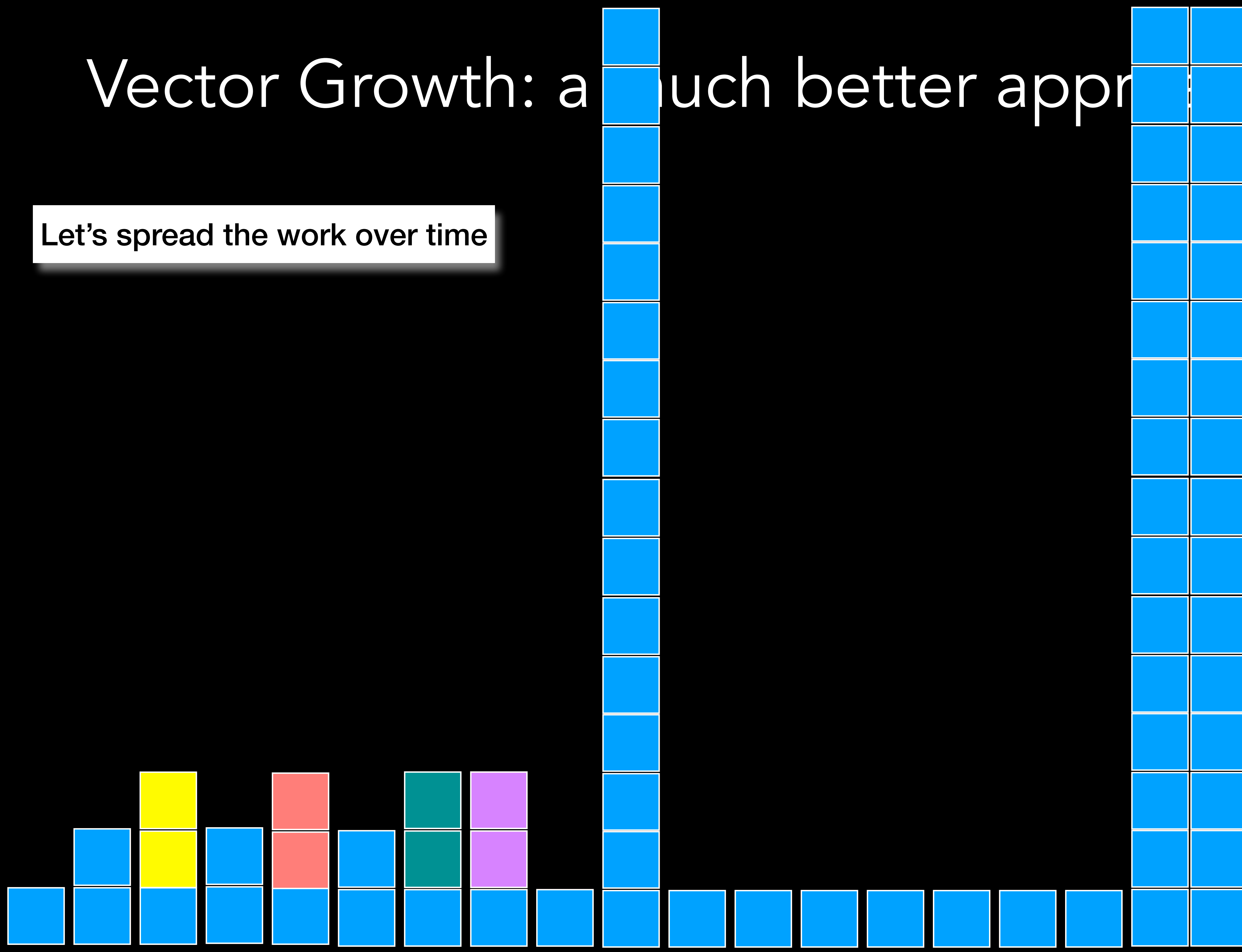
Vector Growth: a much better approach

Let's spread the work over time



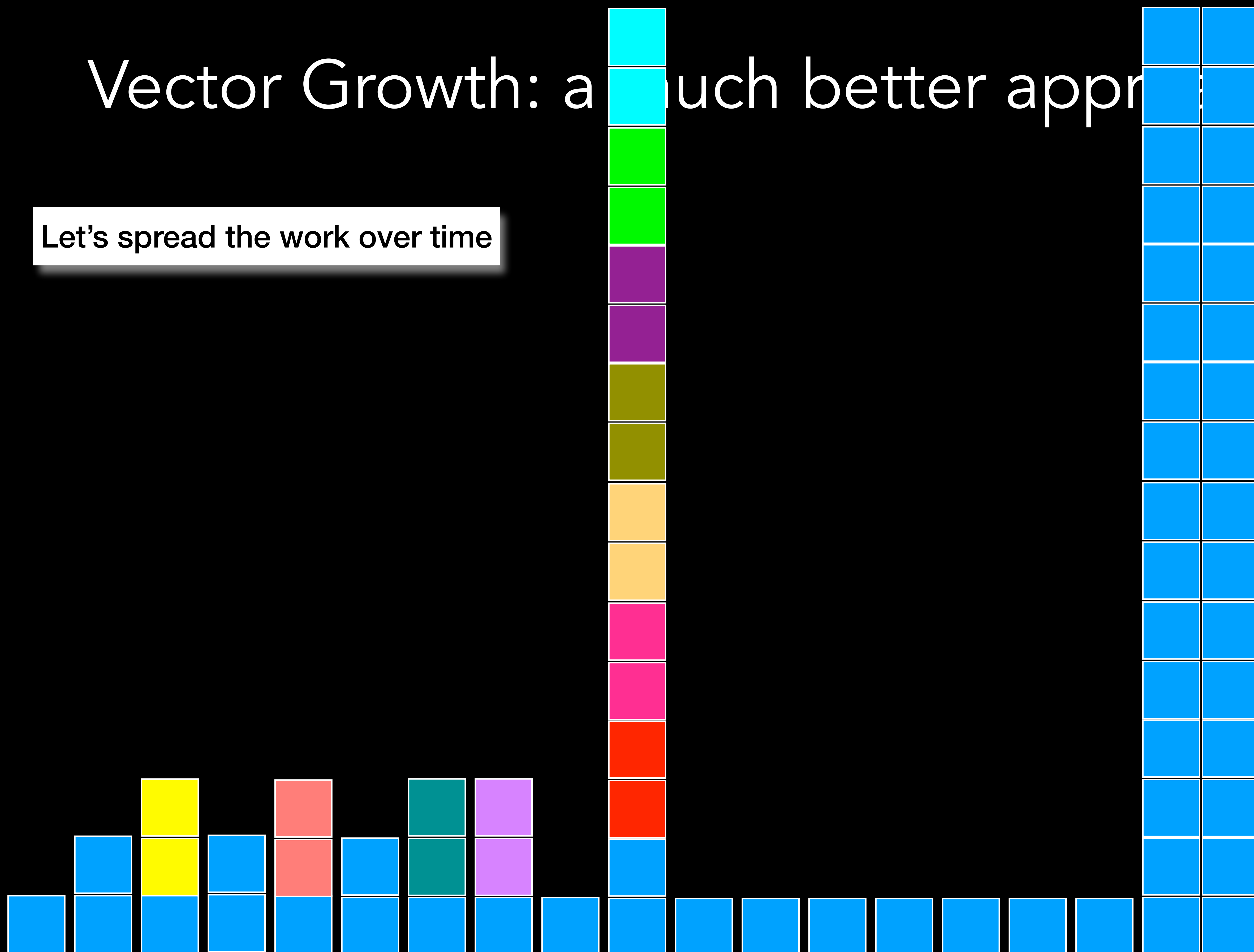
Vector Growth: a much better approach

Let's spread the work over time



Vector Growth: a much better approach

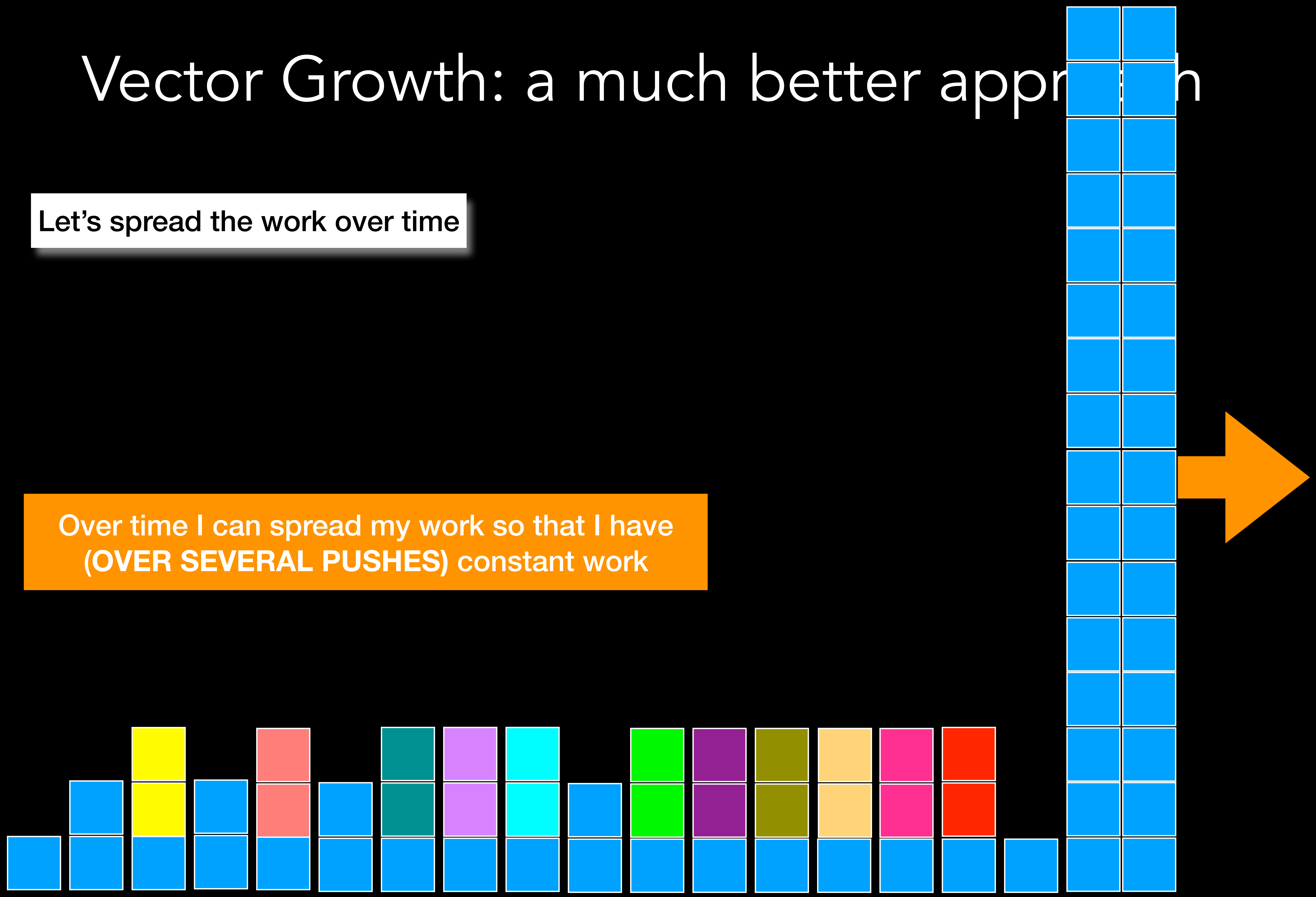
Let's spread the work over time



Vector Growth: a much better appr h

Let’s spread the work over time

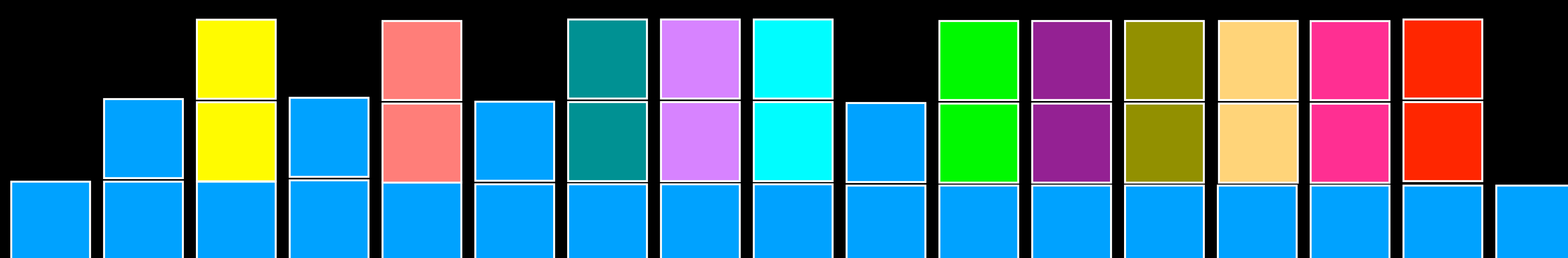
Over time I can spread my work so that I have **(OVER SEVERAL PUSHES)** constant work

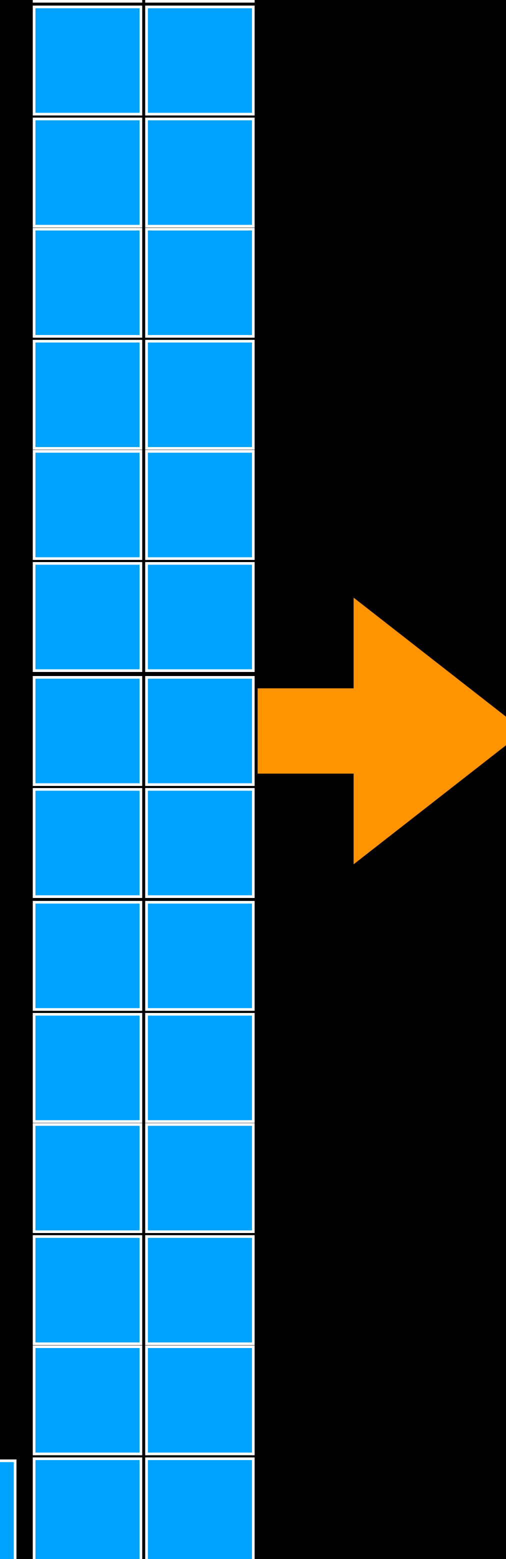


Vector Growth: a much better appr[illegible]h

Let's spread the work over time

Over time I can spread my work so that I have
(**OVER SEVERAL PUSHES**) constant work





Vector Growth summarized

If it grows by 1, $O(n^2)$ over time (**n pushes** - AMORTIZED ANALYSIS)

If it grows by 2, push takes roughly half the "*steps*" but still $O(n^2)$ over time (**n pushes** - AMORTIZED ANALYSIS)

If it doubles its size, push takes $O(1)$ over time (**n pushes** - AMORTIZED ANALYSIS)

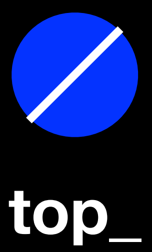
A steadily shrinking Stack

If a large vector suddenly goes almost empty it will still occupy the same space.

Useless memory waste

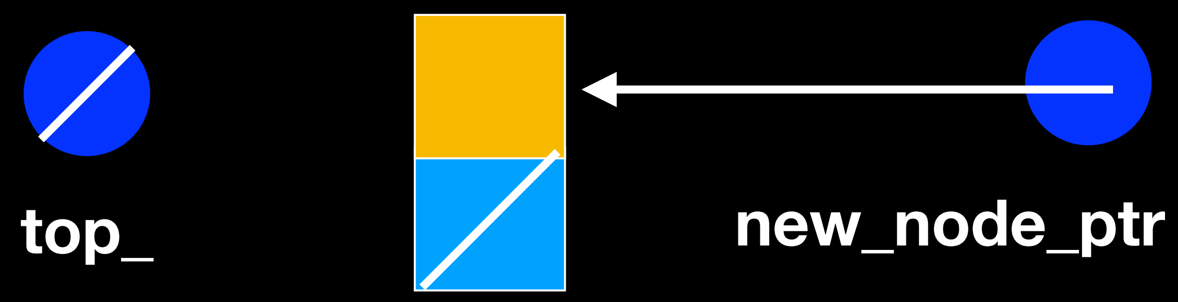
I will not shrink!

Linked Chain



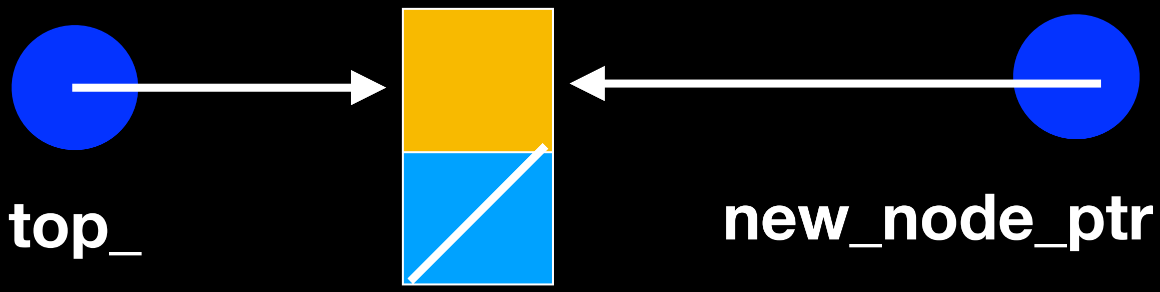
Linked Chain

push



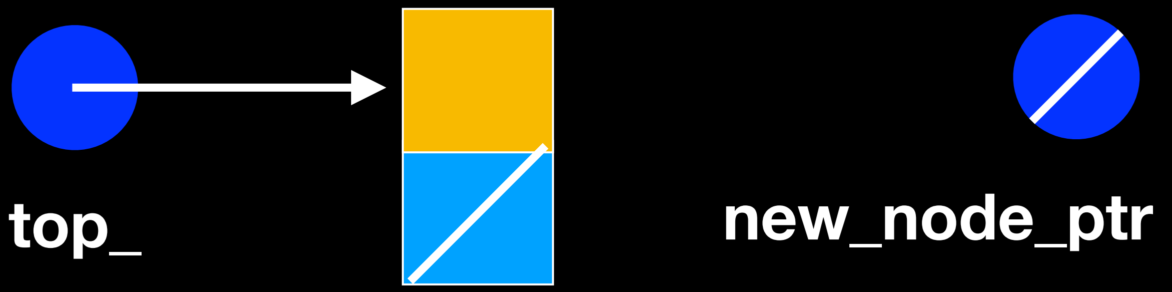
Linked Chain

push

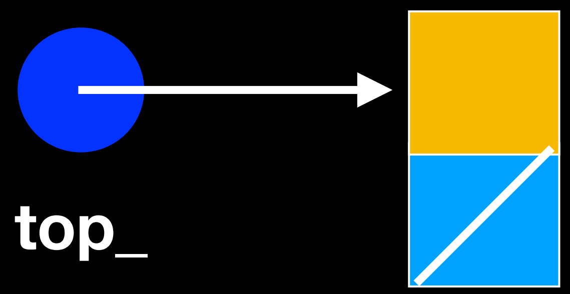


Linked Chain

push

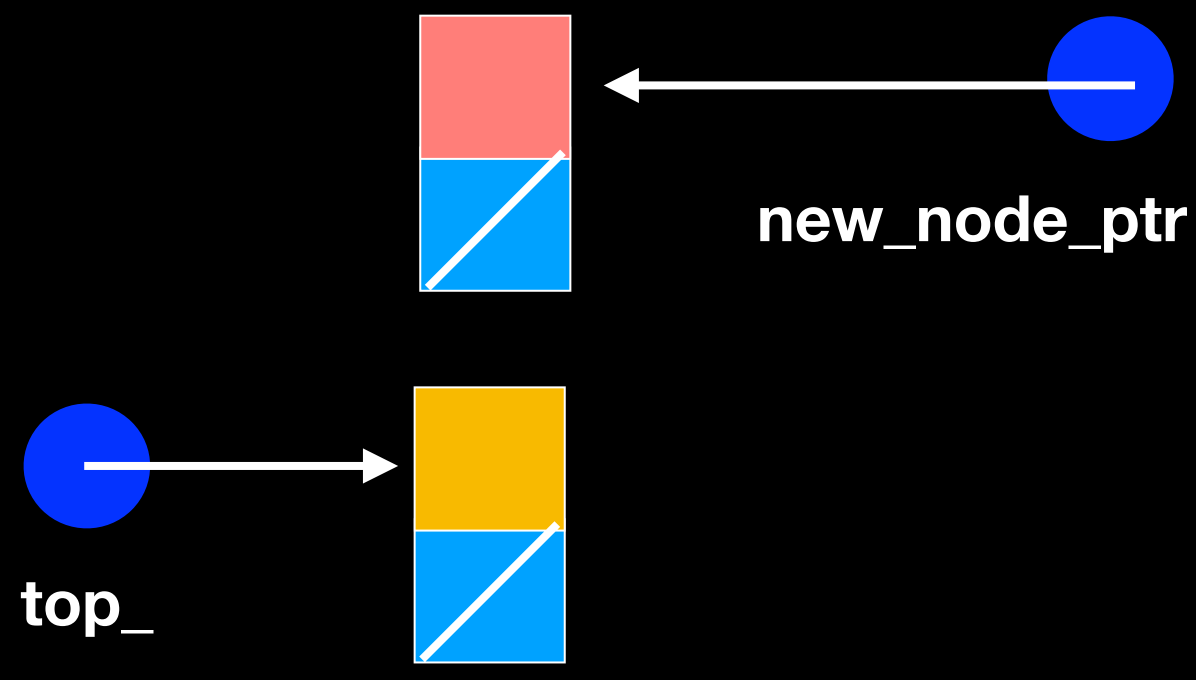


Linked Chain



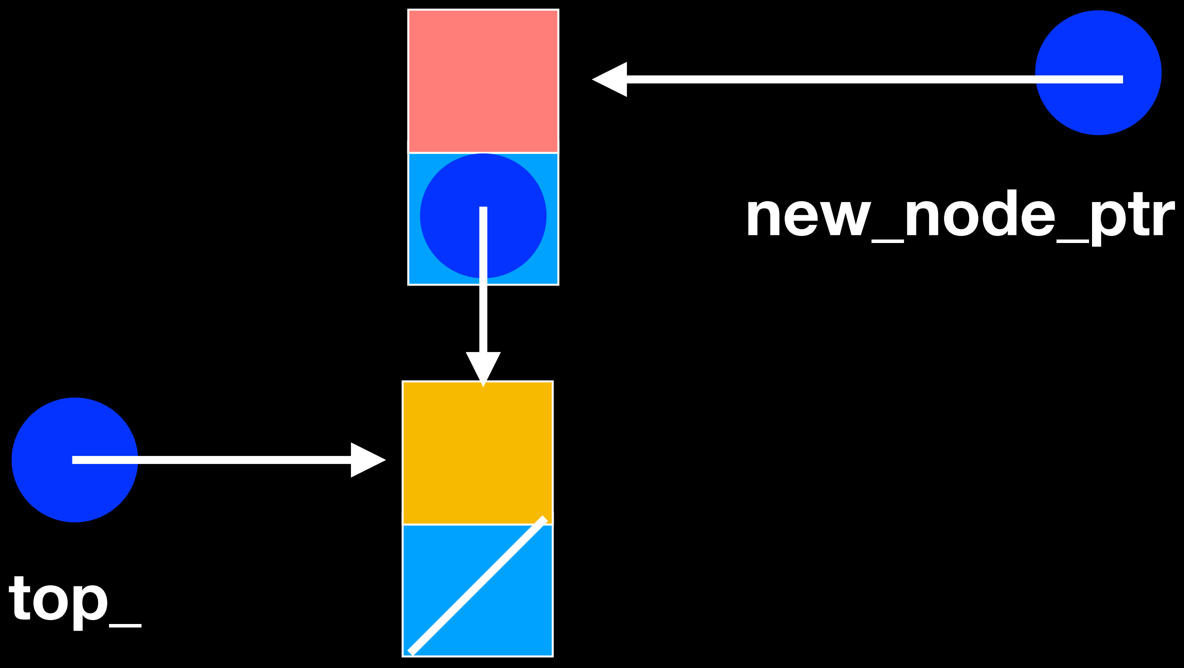
Linked Chain

push



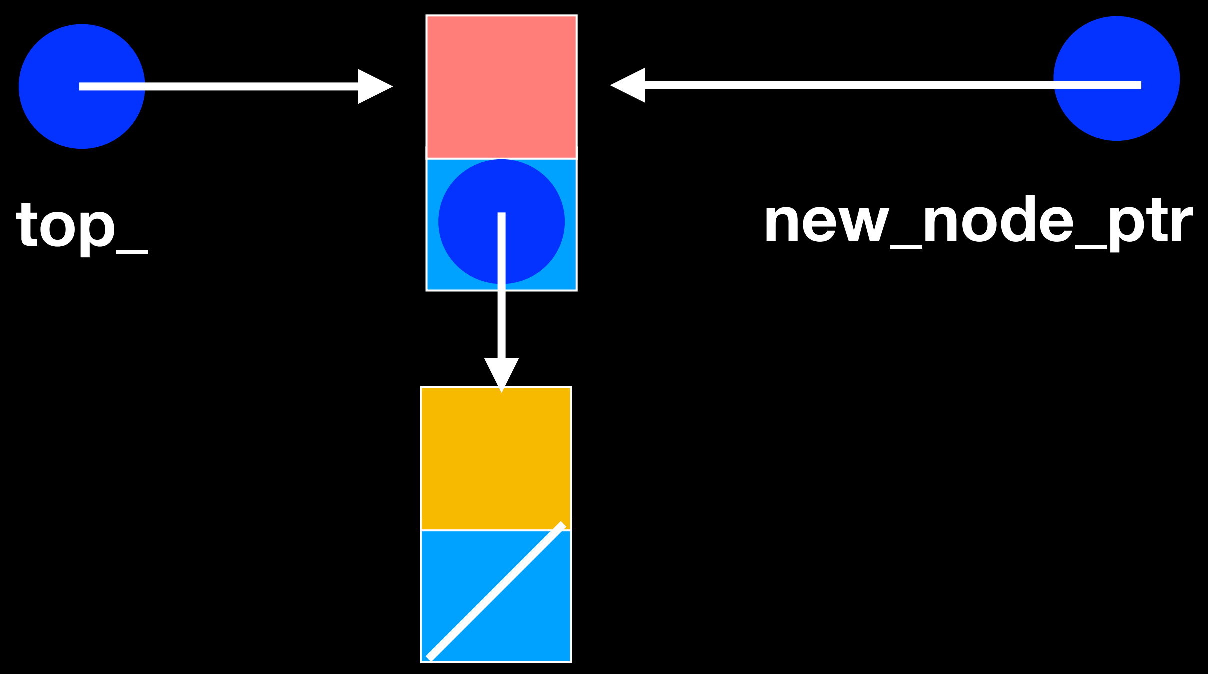
Linked Chain

push



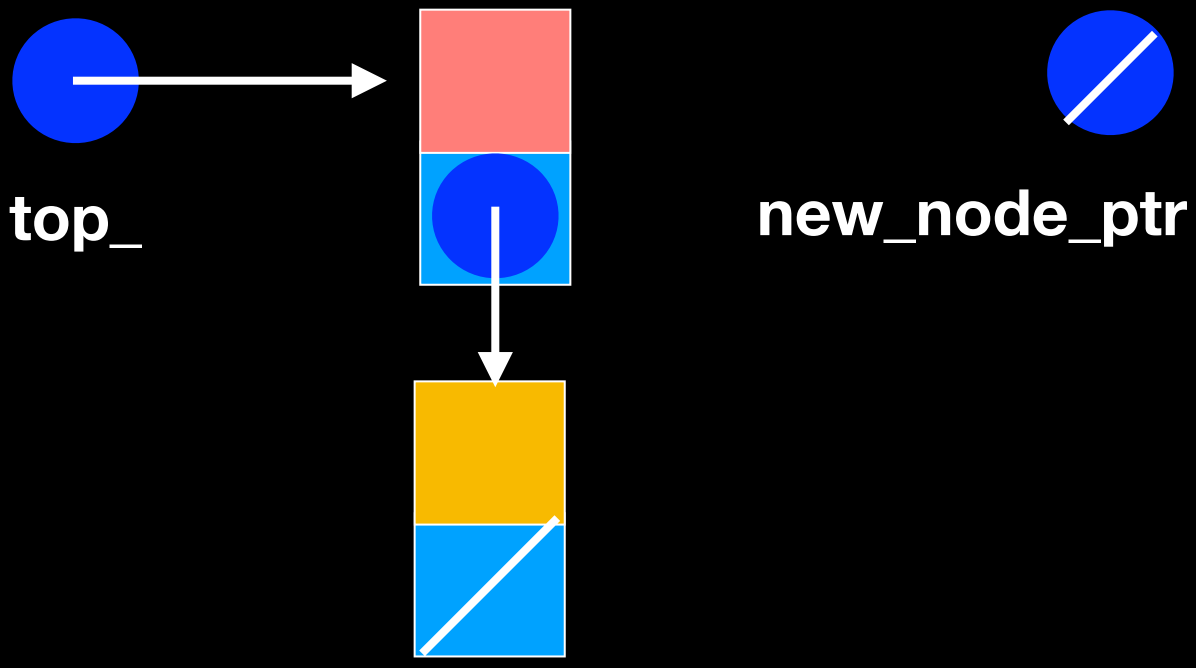
Linked Chain

push

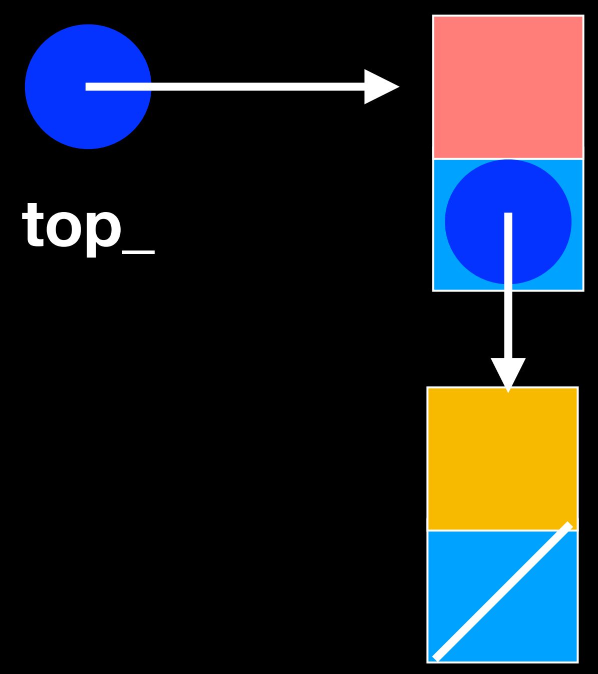


Linked Chain

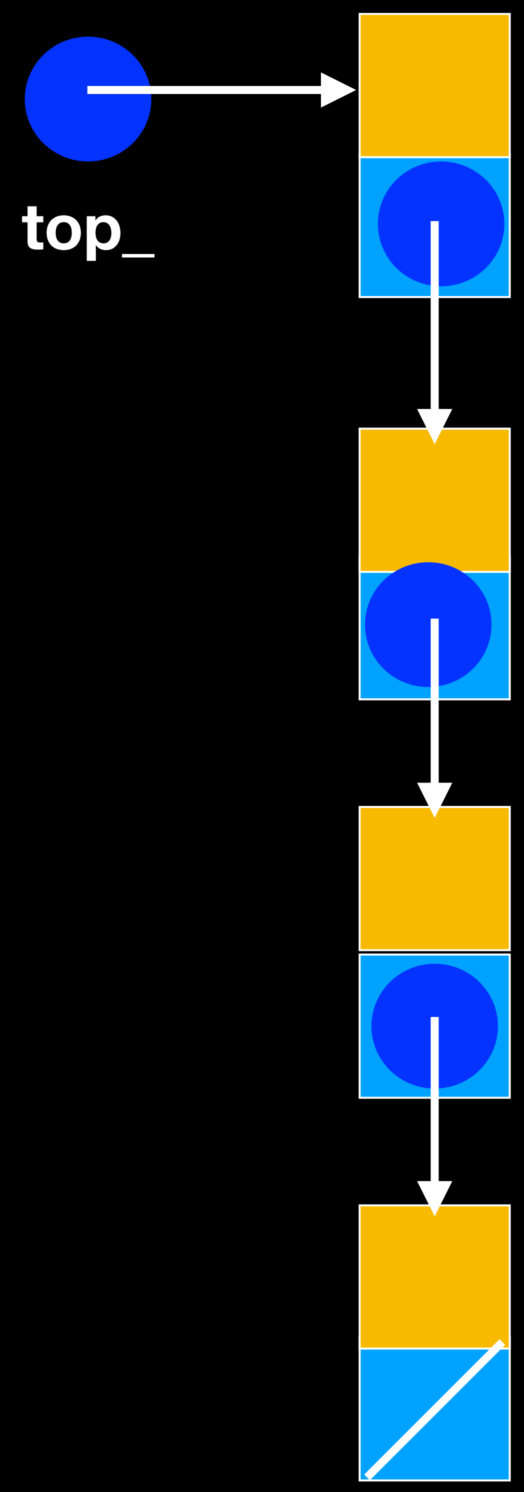
push



Linked Chain

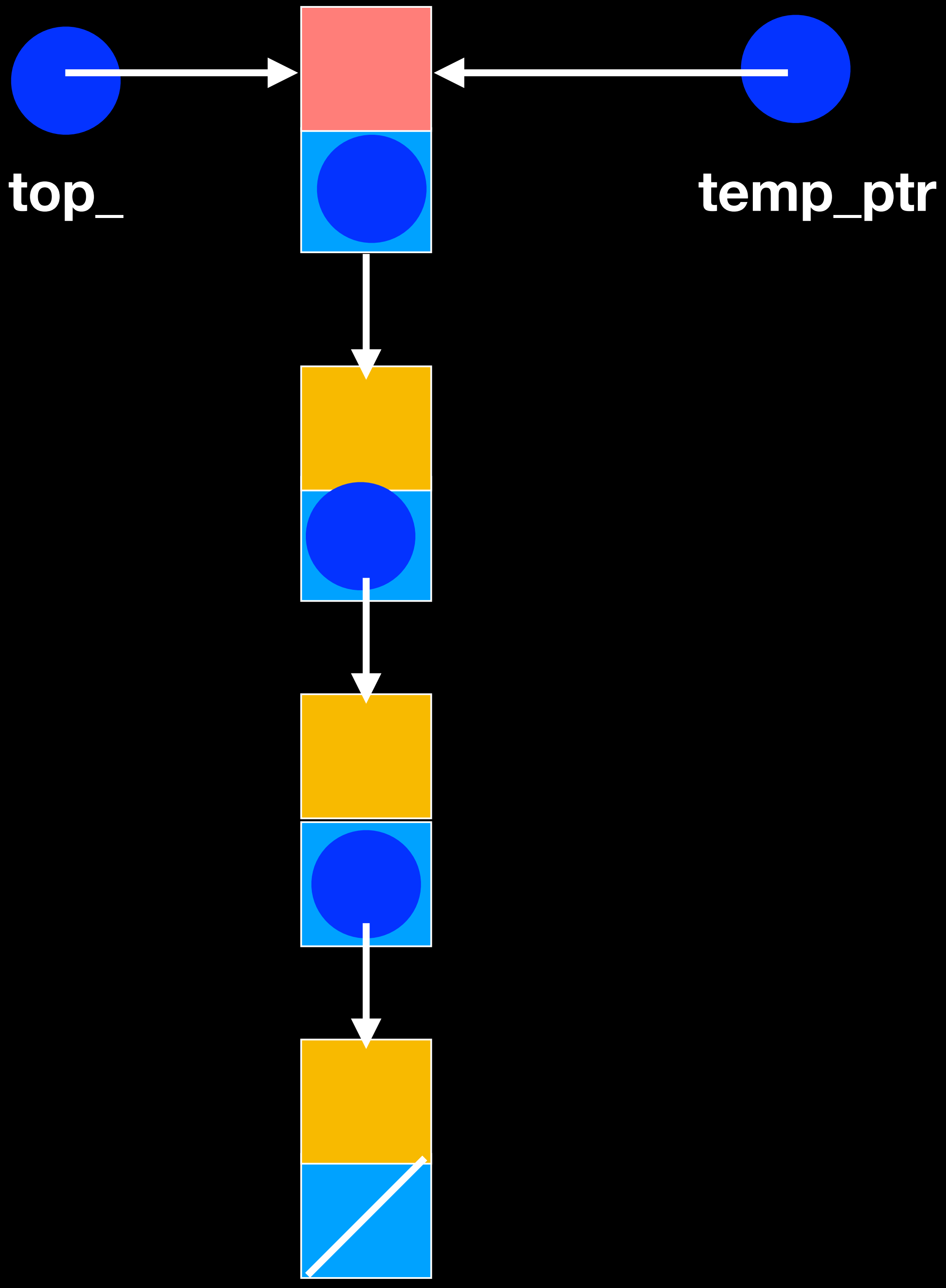


Linked Chain



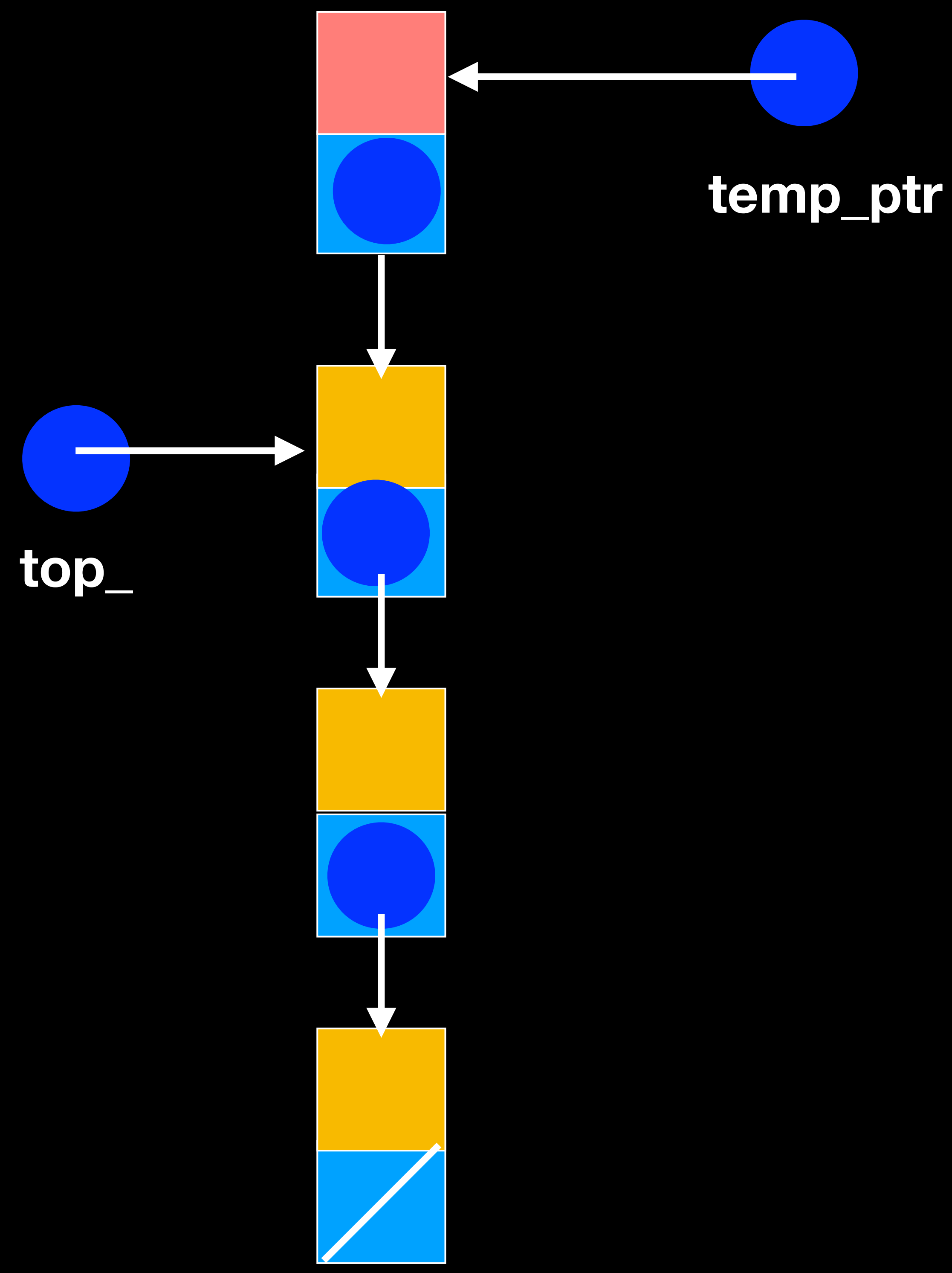
Linked Chain

pop



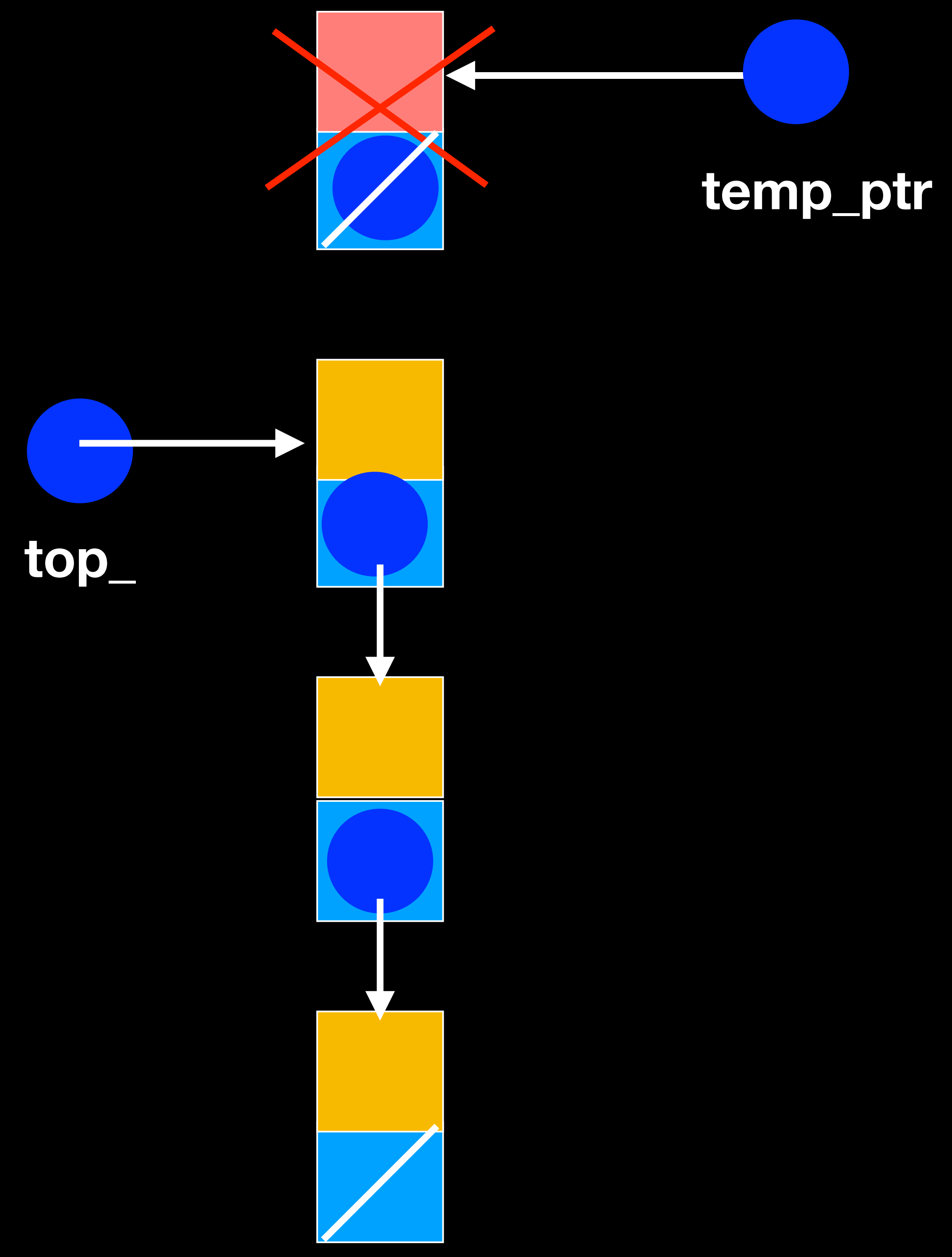
Linked Chain

pop



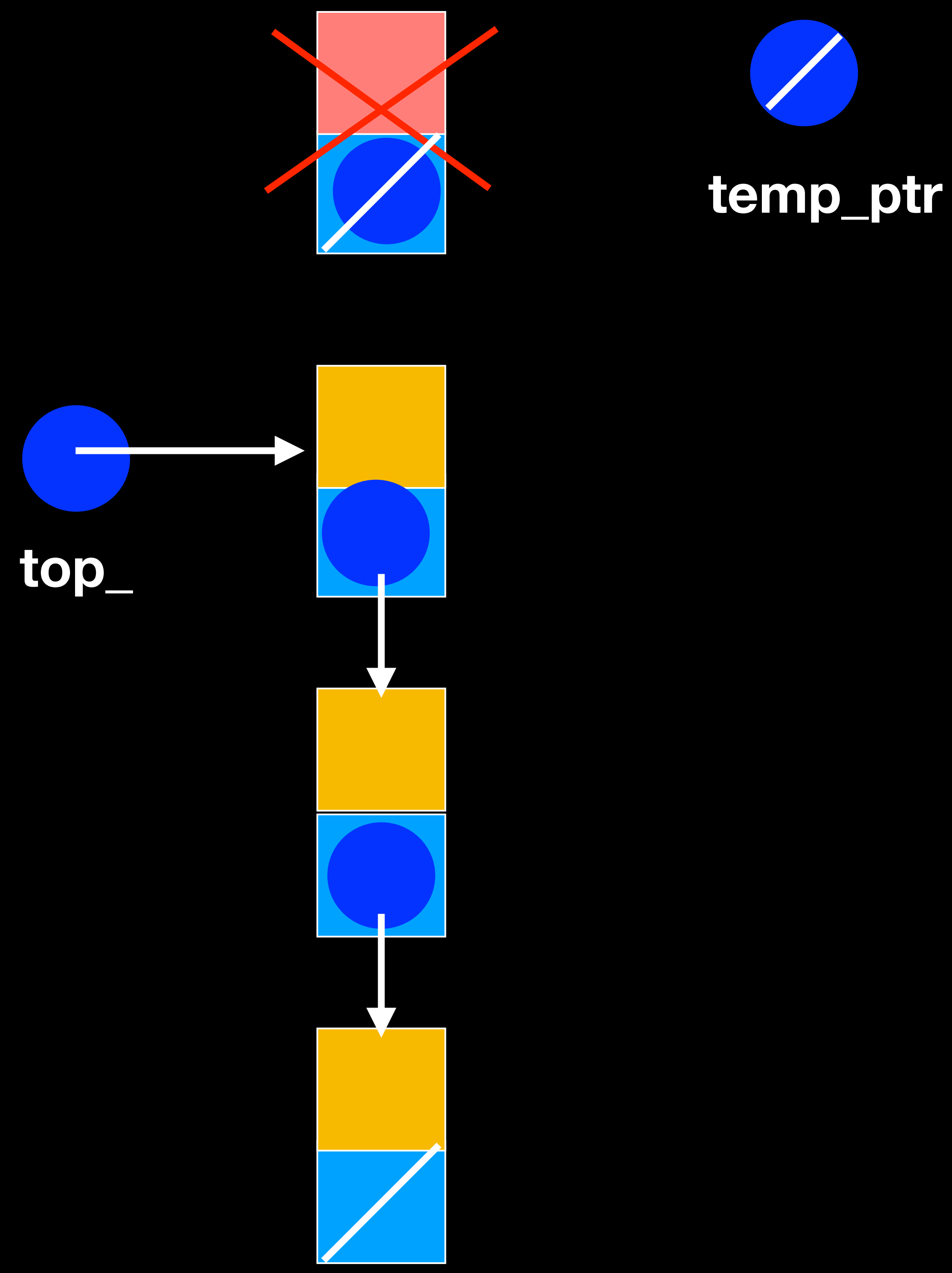
Linked Chain

pop

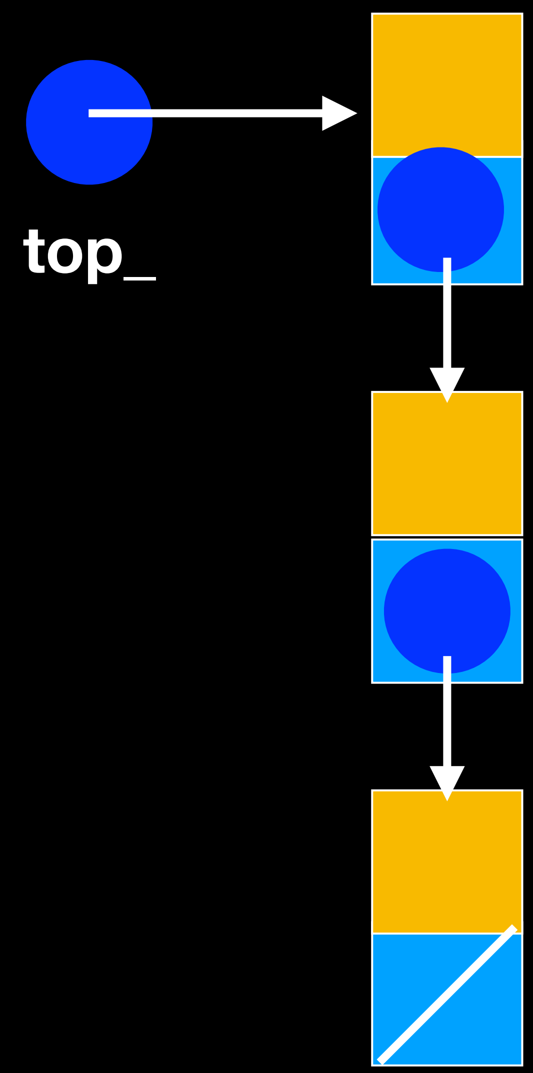


Linked Chain

pop



Linked Chain



Linked-Chain Analysis

1 assignment + 1 increment/decrement = O(1)

`size` : O(1)

`isEmpty`: O(1)

`push`: O(1)

`pop` : O(1)

`top` : O(1)

GREAT!!!! And there is no "Except" case here, every operation is O(1)!

To summarize

Array: O(1) for push and pop, but size is bounded

Vector: size is unbounded but

-Some push operations take O(1), others take O(n) -> O(1) over time (AMORTIZED ANALYSIS)

Linked-Chain: O(1) for push and pop and size is unbounded

Implement Stack ADT

```
#ifndef STACK_H_
#define STACK_H_

template<class T>
class Stack
{

public:
    Stack();
    void push(const T& newEntry); // adds an element to top of stack
    void pop(); // removes element from top of stack
    T top() const; // returns a copy of element at top of stack
    int size() const; // returns the number of elements in the stack
    bool isEmpty() const; // returns true if no elements on stack false otherwise

private:

    // implementation-specific members here


};    //end Stack

#include "Stack.cpp"
#endif // STACK_H_
```

What should we add here to implement it as a linked chain?

Implement Stack ADT

```
#ifndef STACK_H_
#define STACK_H_

template<class T>
class Stack
{

public:
     Stack();
     ~Stack(); // destructor
     Stack(const Stack<T>& a_stack); //copy constructor
     void push(const T& newEntry); // adds an element to top of stack
     void pop(); // removes element from top of stack
     T top() const; // returns a copy of element at top of stack
     int size() const; // returns the number of elements in the stack
    bool isEmpty() const; // returns true if no elements on stack false otherwise

private:
    Node<T>* top_; // Pointer to top of stack
     int item_count;        // number of items currently on the stack

};    //end Stack

#include "Stack.cpp"
#endif // STACK_H_
```