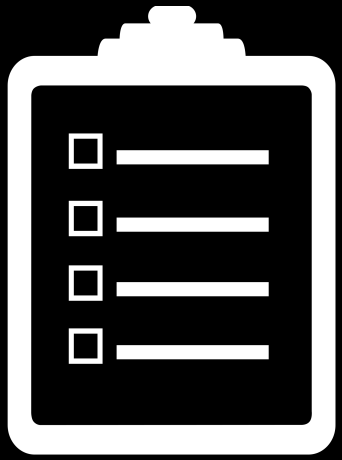
Sequential Containers

Ali Malik
malikali@stanford.edu

Game Plan



Recap

Overview of STL

Sequence Containers

`std::vector`

`std::deque`

Container Adapters

Announcements

Recap

Structs

You can define your own mini-types that bundle multiple variables together:

```
struct point {
    int x;
    int y;
};
```


Useful for Assignment 1

Structs

```
struct point {
    int x;
    int y;
};

point p;
p.x = 12;
p.y = 15;
```

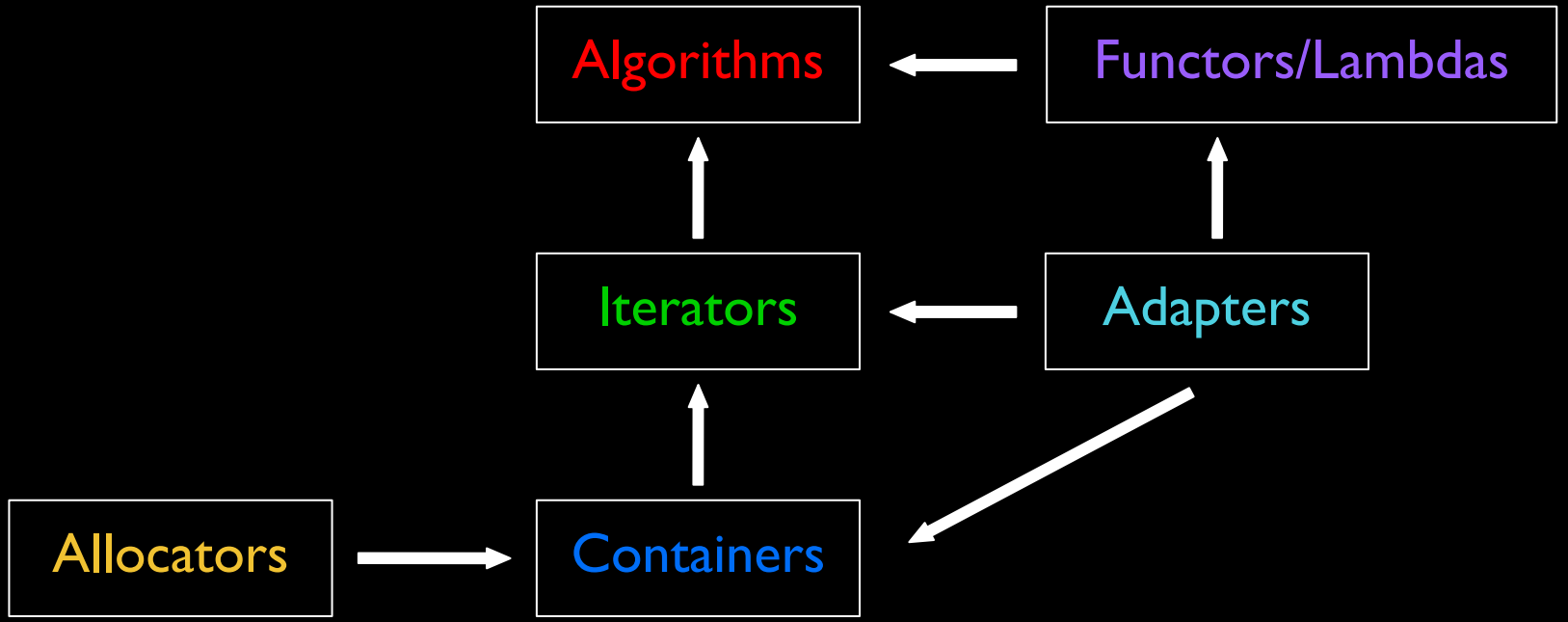
Overview of STL

Overview of STL

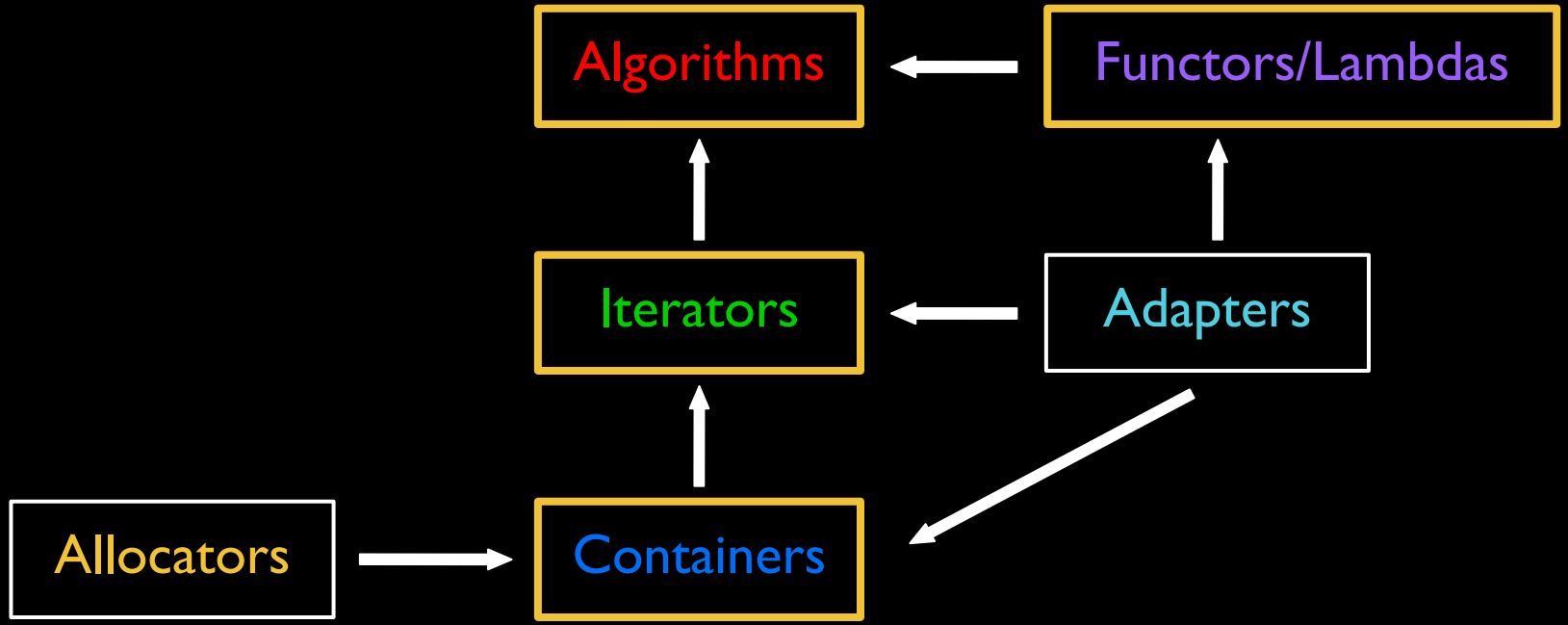
“As mathematicians learned to lift theorems into their most general setting, so I wanted to lift algorithms and data structures”

— *Alex Stepanov, inventor of the STL*

Overview of STL



Overview of STL



Where we are going...

Here is a program that generates a vector with random entries, sorts it, and prints it, all in one go!

```
const int kNumInts = 200;
std::vector<int> vec(kNumInts);
std::generate(vec.begin(), vec.end(), rand);
std::sort(vec.begin(), vec.end());
std::copy(vec.begin(), vec.end(),
                std::ostream_iterator<int>(cout, "\n"));
```

Sequence Containers

Sequence Containers

Provides access to sequences of elements.

Examples:

- `std::vector<T>`
- `std::list<T>`
- `std::deque<T>`

```
std::vector<T>
```

std::vector<T>

A vector represents a sequence of elements of any type.

You specify the type when using the vector:

```
std::vector<int> vecInt;           // vector of ints
std::vector<string> vecStr;        // vector of string
std::vector<myStruct> vecStruct; // vector of myStructs
std::vector<std::vector<string>> vecOfVec; // vector of
                                           vector<string>
```

Summary of `std::vector<T>` vs Stanford `Vector<T>`

What you want to do	Stanford Vector<int>	std::vector<int>
Create an empty vector	`Vector<int> v;`	`vector<int> v;`
Create a vector with n copies of zero	`Vector<int> v(n);`	`vector<int> v(n);`
Create a vector with n copies of a value k	`Vector<int> v(n, k);`	`vector<int> v(n, k);`
Add k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Get the element at index i (verify that i is in bounds)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (verify that i is in bounds)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;`

Summary of `std::vector<T>` vs Stanford `Vector<T>`

What you want to do	Stanford Vector<int>	std::vector<int>
Create an empty vector	`Vector<int> v;`	`vector<int> v;`
Create a vector with n copies of zero	`Vector<int> v(n);`	`vector<int> v(n);`
Create a vector with n copies of a value k	`Vector<int> v(n, k);`	`vector<int> v(n, k);`
Add k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Get the element at index i (verify that i is in bounds)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (verify that i is in bounds)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;`

Summary of `std::vector<T>` vs Stanford `Vector<T>`

What you want to do	Stanford Vector<int>	std::vector<int>
Create an empty vector	`Vector<int> v;`	`vector<int> v;`
Create a vector with n copies of zero	`Vector<int> v(n);`	`vector<int> v(n);`
Create a vector with n copies of a value k	`Vector<int> v(n, k);`	`vector<int> v(n, k);`
Add k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Get the element at index i (verify that i is in bounds)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (verify that i is in bounds)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;`

Summary of `std::vector<T>` vs Stanford `Vector<T>`

What you want to do	Stanford Vector<int>	std::vector<int>
Create an empty vector	`Vector<int> v;`	`vector<int> v;`
Create a vector with n copies of zero	`Vector<int> v(n);`	`vector<int> v(n);`
Create a vector with n copies of a value k	`Vector<int> v(n, k);`	`vector<int> v(n, k);`
Add k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Get the element at index i (verify that i is in bounds)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (verify that i is in bounds)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;`

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

std::vector<T>

Problem:

Write a program that reads a list of integers and finds the median.

Vector Median

(VecMedian.pro)

std::vector<T>

Some new stuff there:

```
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());
```

std::vector<T>

Some new stuff there:

```
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());
```

This is a promise to the compiler that this variable won't change.

std::vector<T>

Some new stuff there:

```
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());
```

This let's us use `vecsz_t` as an alias/synonym for the type `std::vector<int>::size_type;`

std::vector<T>

Some new stuff there:

```
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());
```


This takes a range of the vector and sorts it

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

Why these differences?

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

Why these differences?

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

Why these differences?

Why the Differences?

Why doesn't `std::vector` bounds check by default?

Hint: Remember our discussion of the philosophy of C++

If you write your program correctly, bounds checking will just slow your code down.

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

Why these differences?

Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

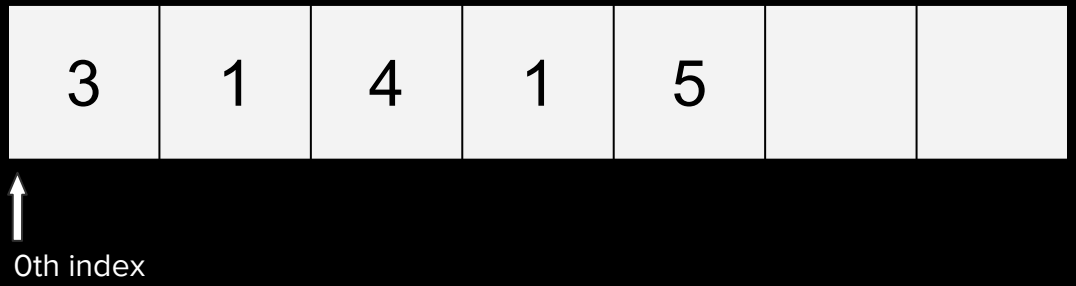
Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`
Apply a function to each element in x	`x.mapAll(fn)`	`// We'll talk about this in another lecture...`
Concatenate vectors v1 and v2	`v1 += v2;`	`// We'll talk about this in another lecture...`

Why these differences?

Why is `push_front` slow?

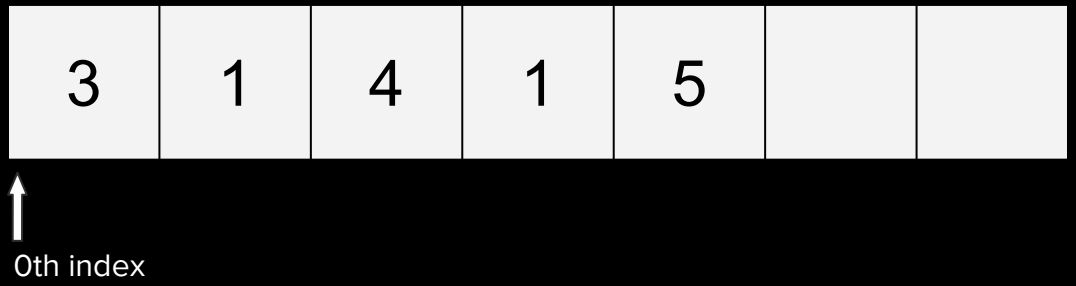
Requires shifting over of the other elements in the vector down one by one (bad).

Illustration: Say we have a small vector



Why is `push_front` slow?

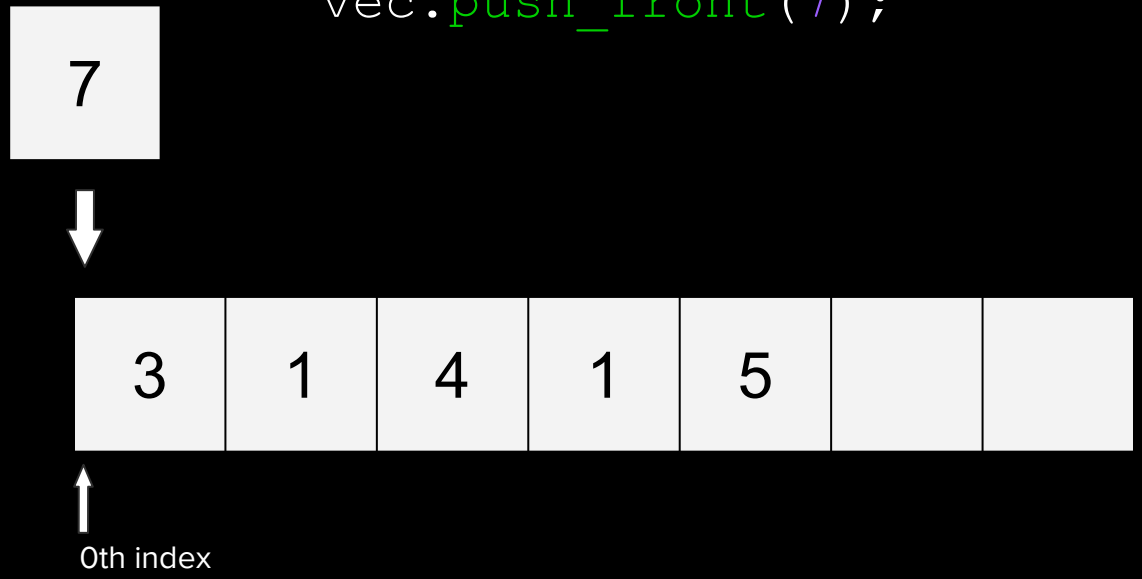
Suppose `push_front` existed and we used it.



Why is `push_front` slow?

Suppose `push_front` existed and we used it.

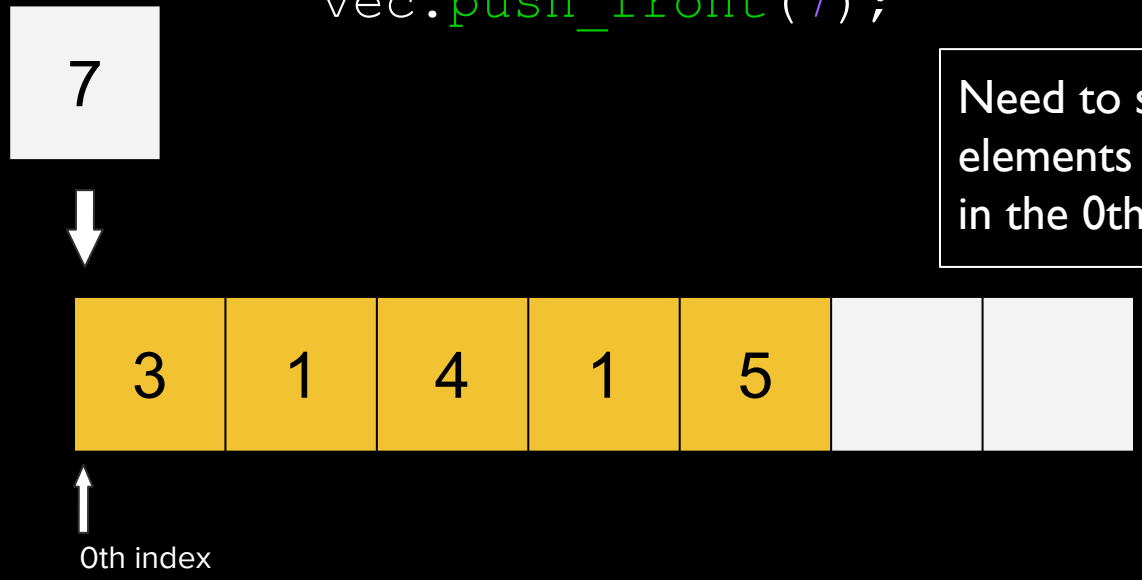
```
vec.push_front(7);
```



Why is `push_front` slow?

Suppose `push_front` existed and we used it.

```
vec.push_front(7);
```

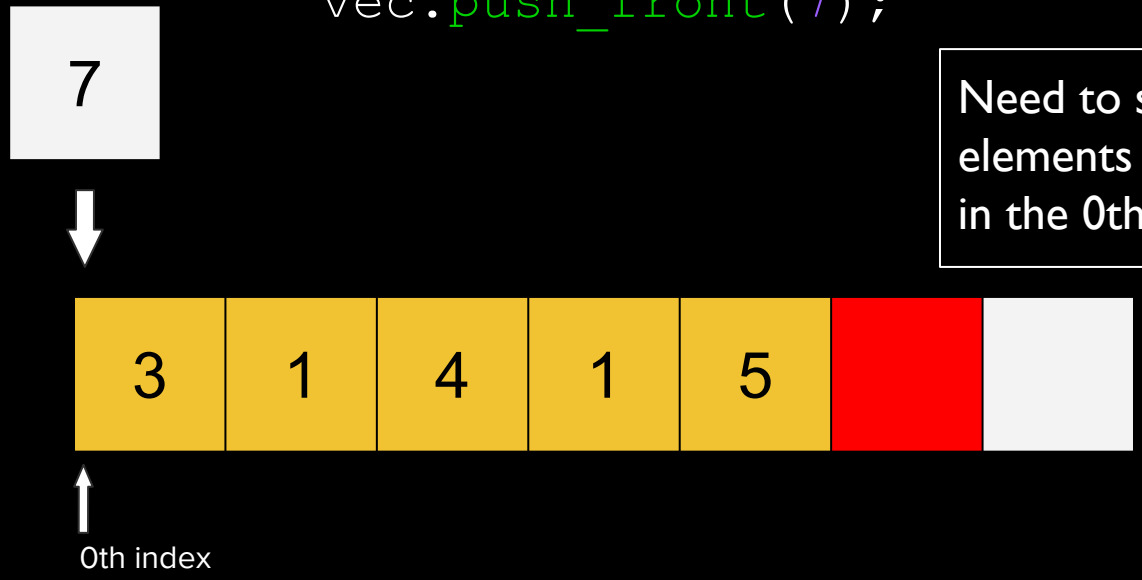


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

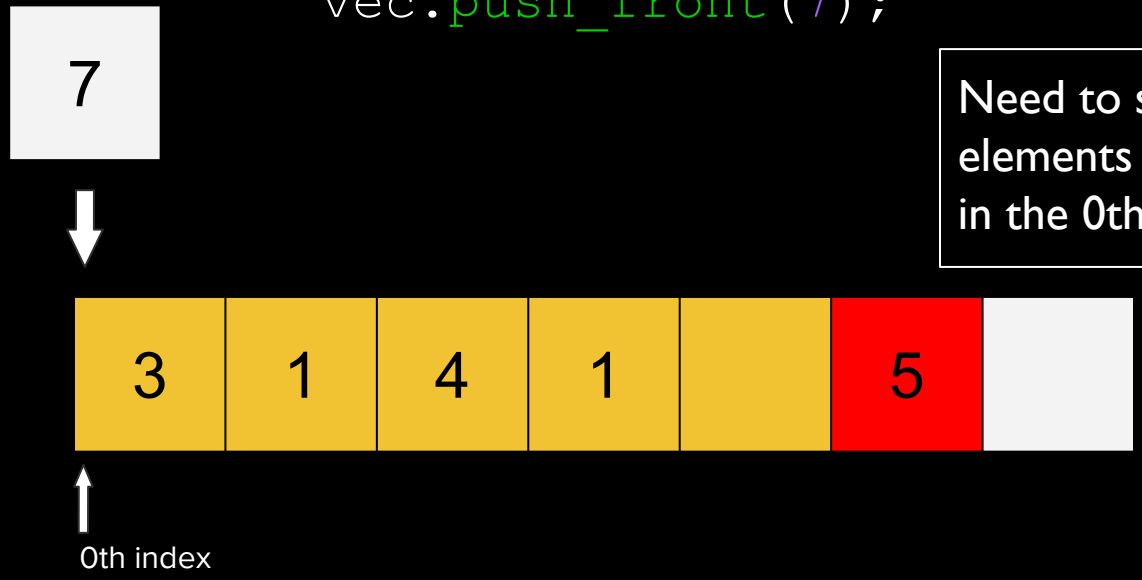


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

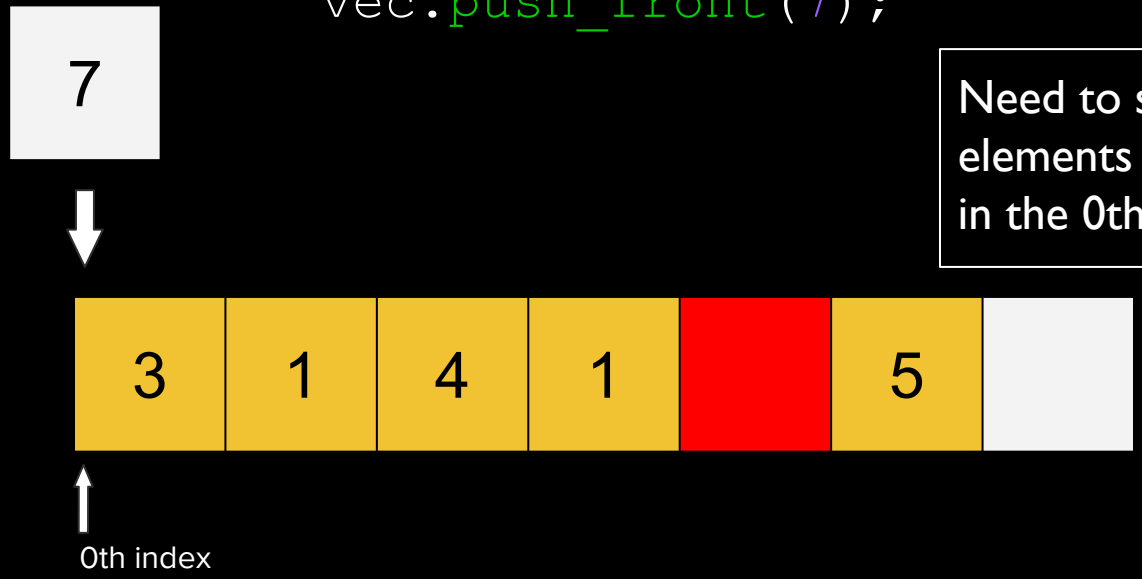


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

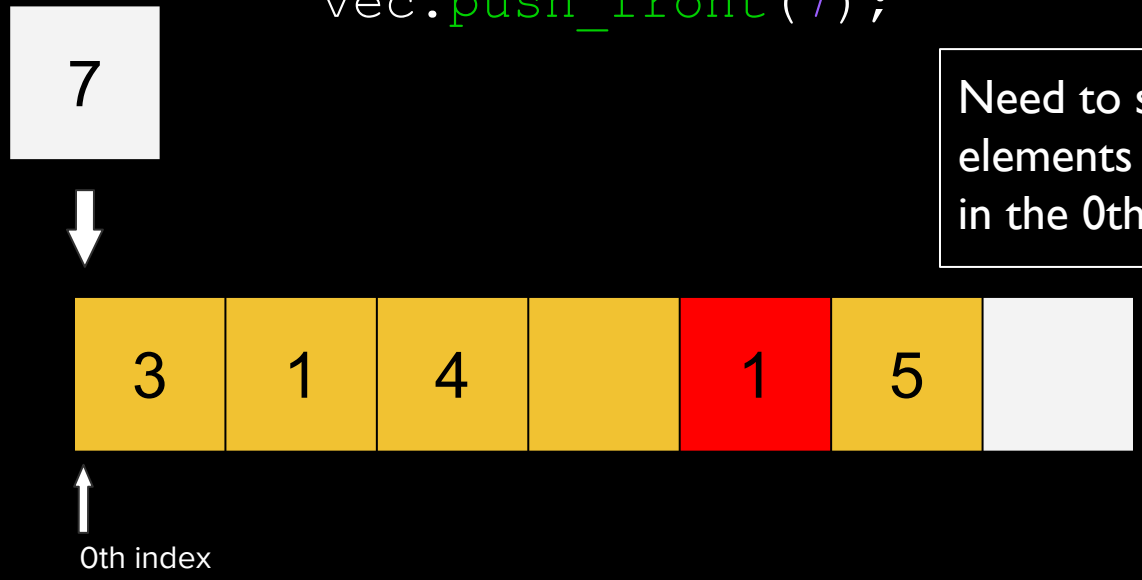


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

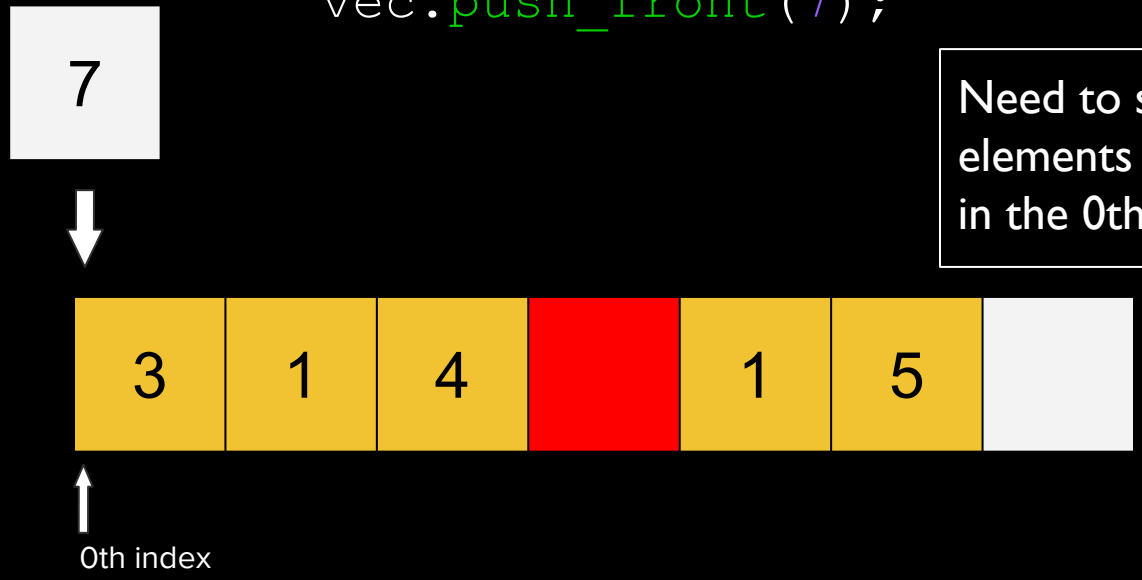


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

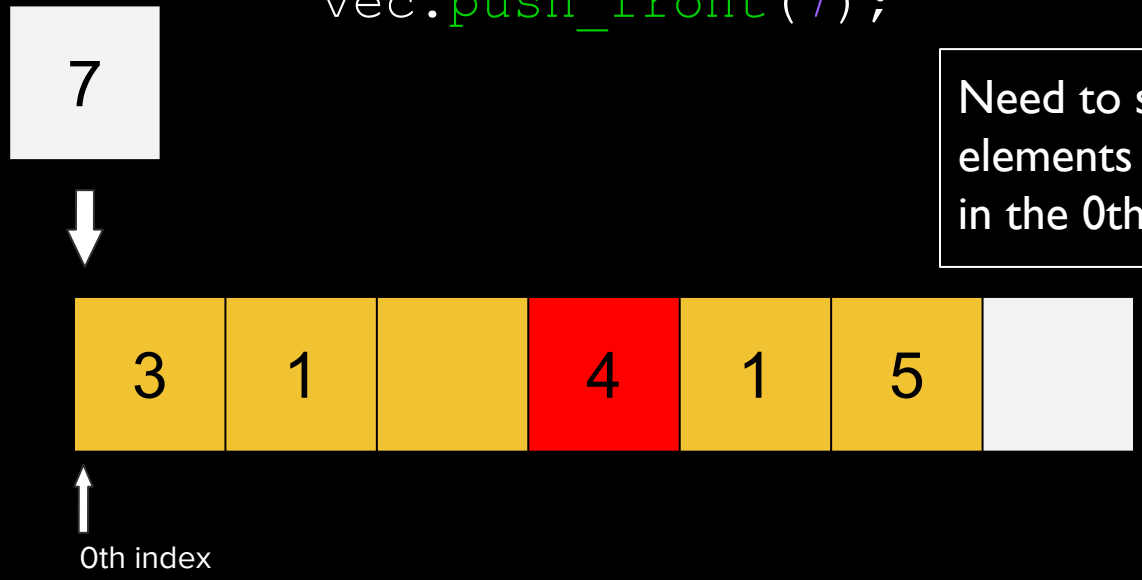


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

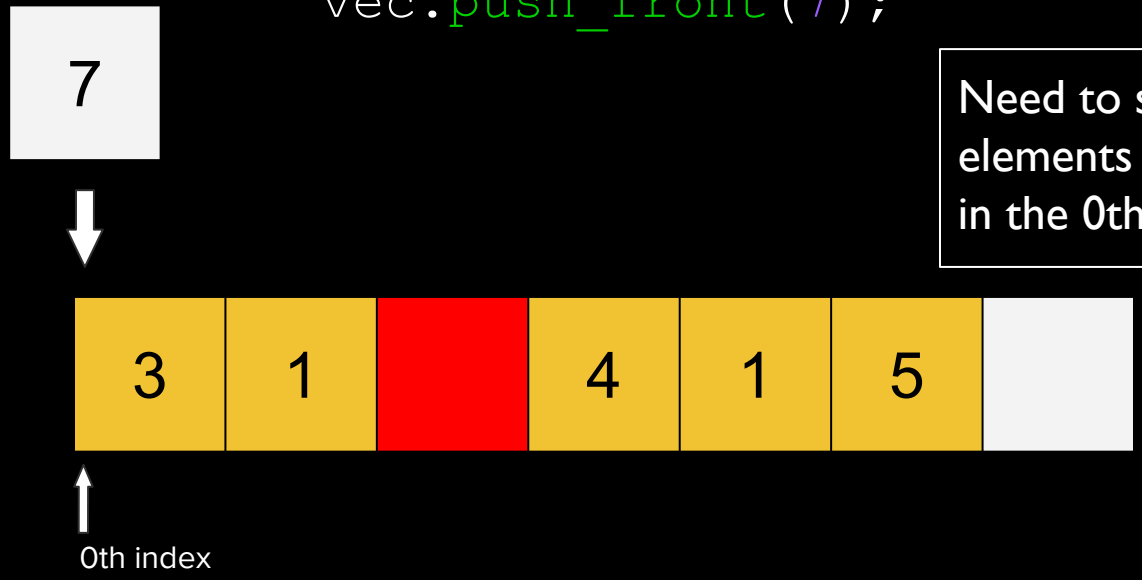


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

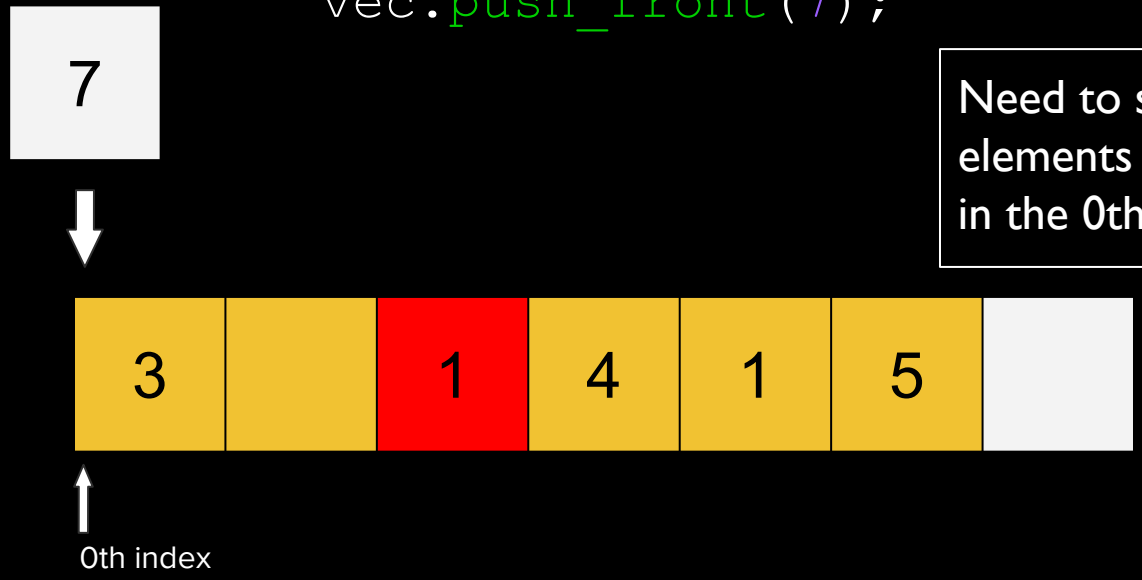


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

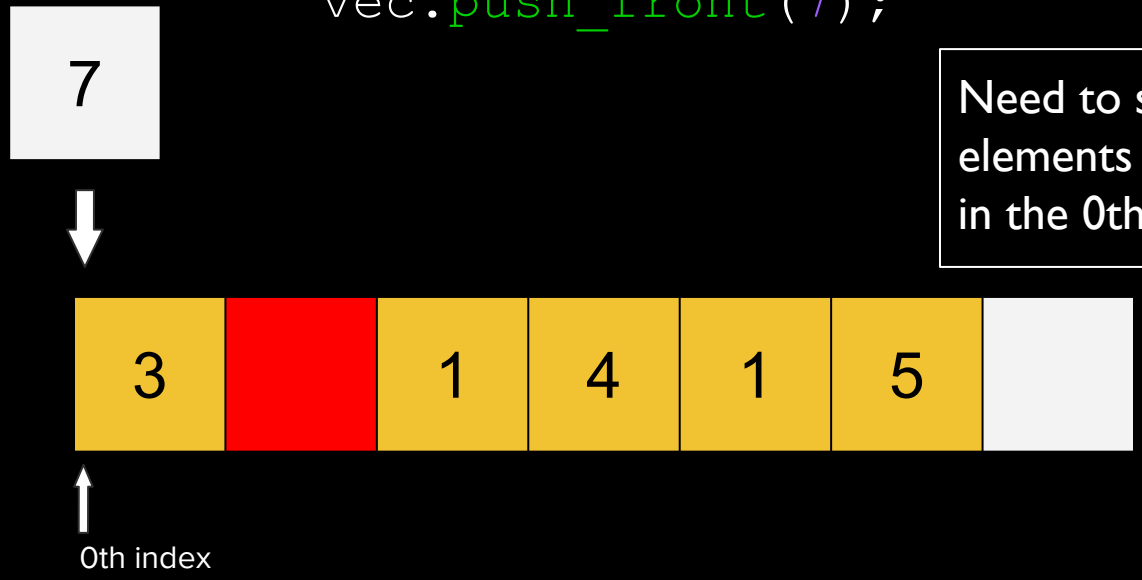


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

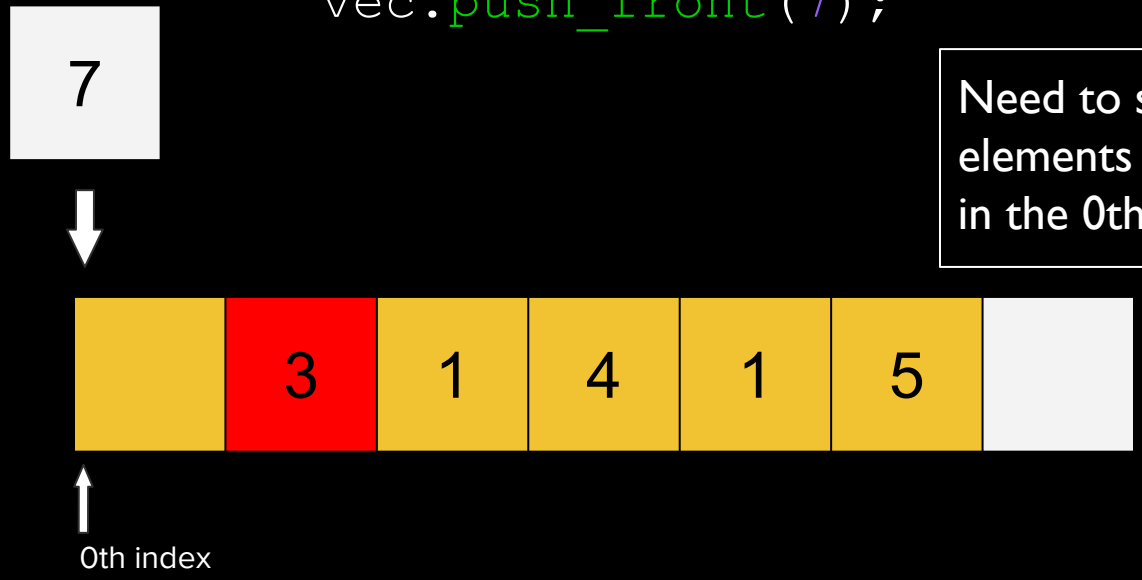


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

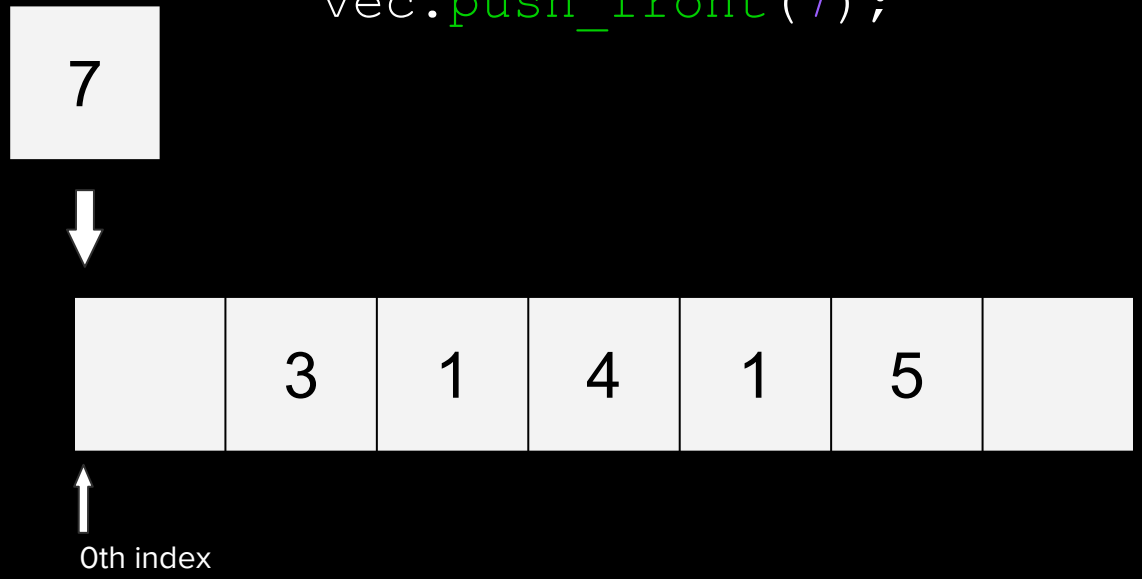


Need to shift these elements up to make space in the 0th position.

Why is `push_front` slow?

Suppose `push_front` existed and we used it

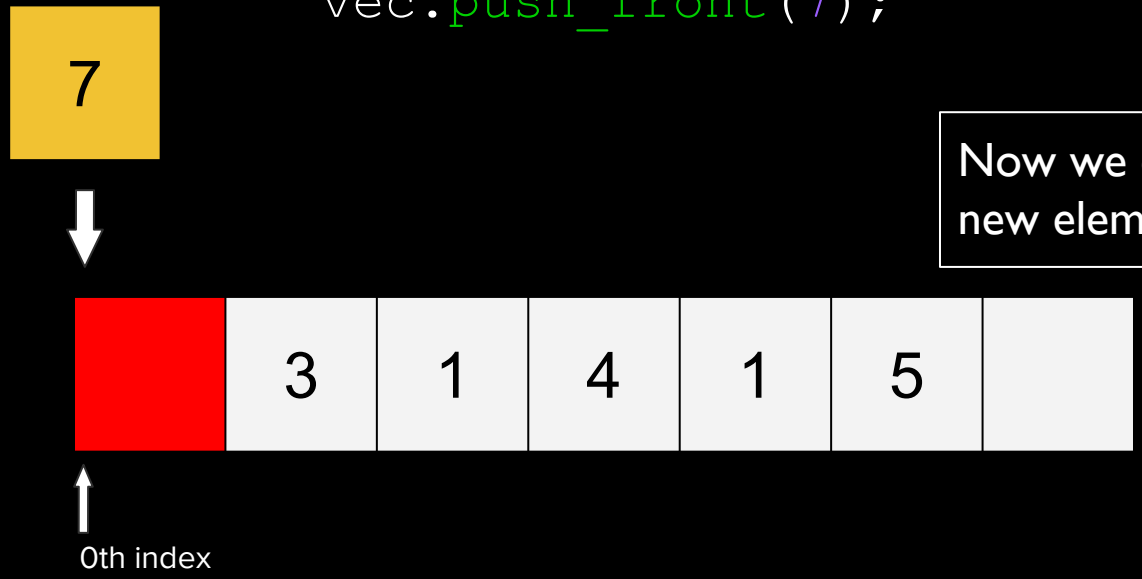
```
vec.push_front(7);
```



Why is `push_front` slow?

Suppose `push_front` existed and we used it

```
vec.push_front(7);
```

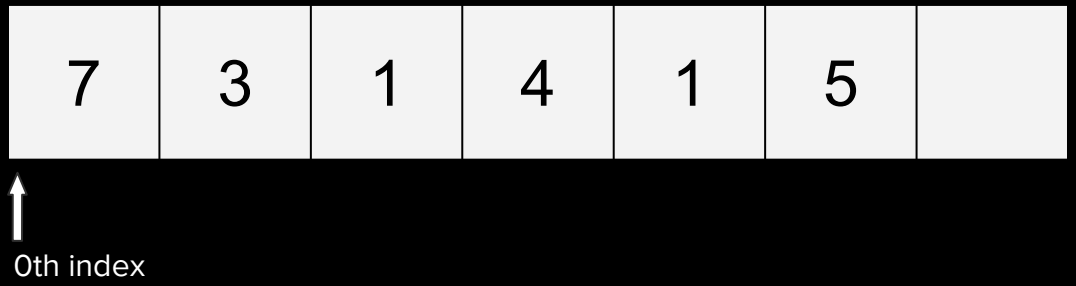


Now we can insert the new element.

Why is `push_front` slow?

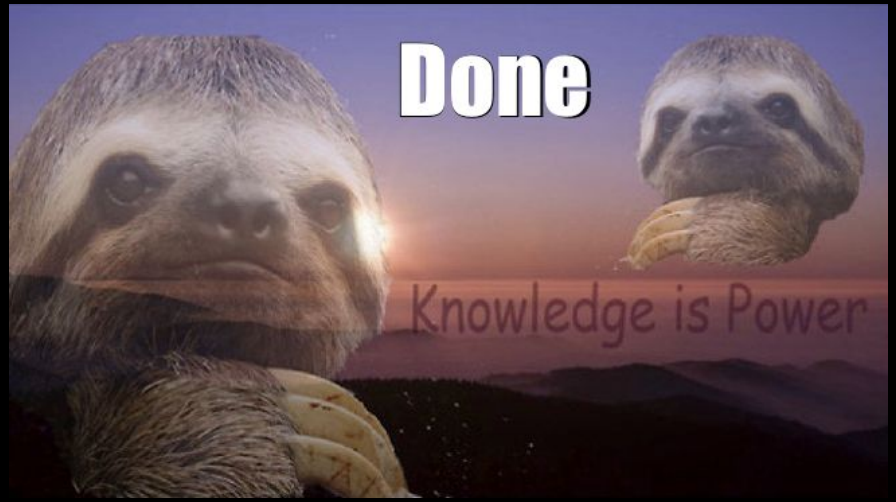
Suppose `push_front` existed and we used it

```
vec.push_front(7);
```



Why is `push_front` slow?

...



7	3	1	4	1	5	

0th index

Why is `push_front` slow?

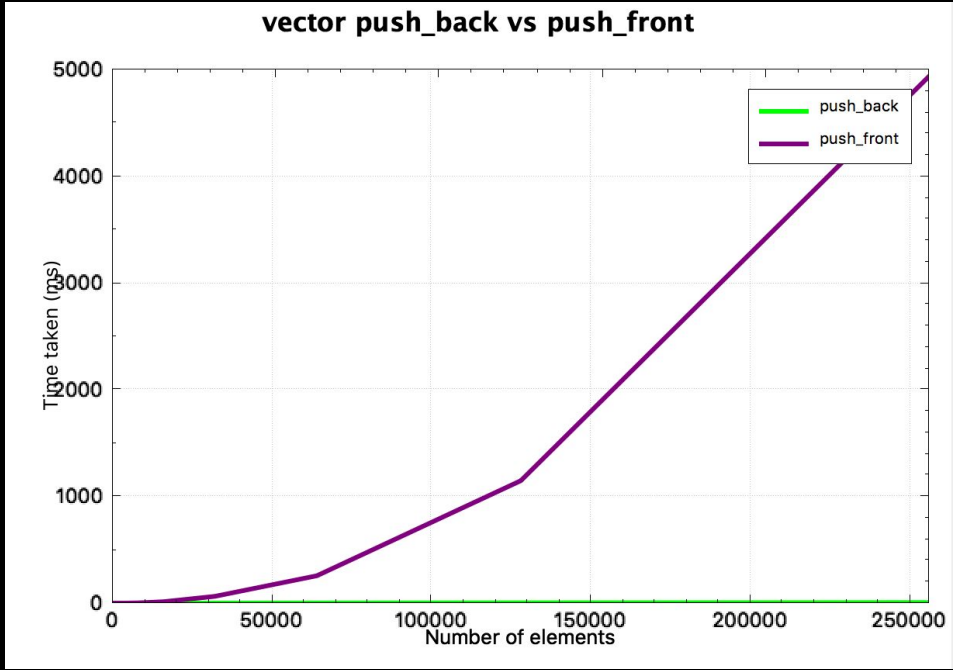
Let’s get a sense of the difference:

Insertion Speed

(`InsertionSpeed.pro`)

Why is `push_front` slow?

The results:



Why is `push_front` slow?

A vector is the prime tool of choice in most applications!

- Fast
- Lightweight
- Intuitive

However, we just saw vectors only grow efficiently in one direction.

Sometimes it is useful to be able to `push_front` quickly!

C++ has a solution!

```
std::deque<T>
```

`std::deque<T>`

A deque (pronounced “deck”) is a double ended queue.

Can do everything a vector can do

and also...

Unlike a vector, it is possible (and fast) to `push_front` and `pop_front`.

std::deque<T>

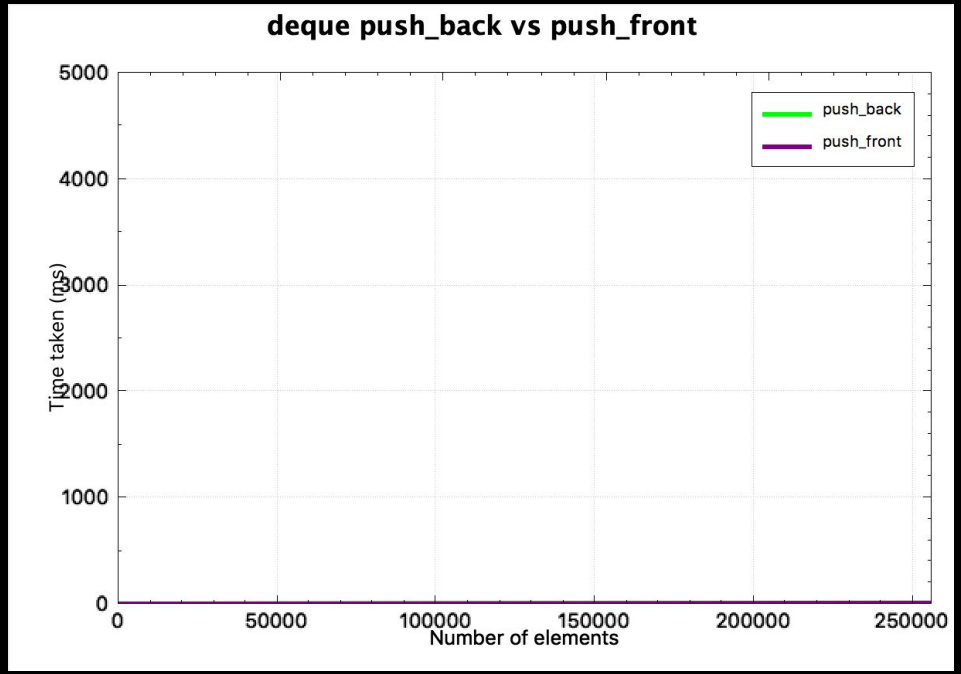
We can see the efficiency of `push_front` with a `std::deque`

Deque Speed

(`DequeSpeed.pro`)

std::deque<T>

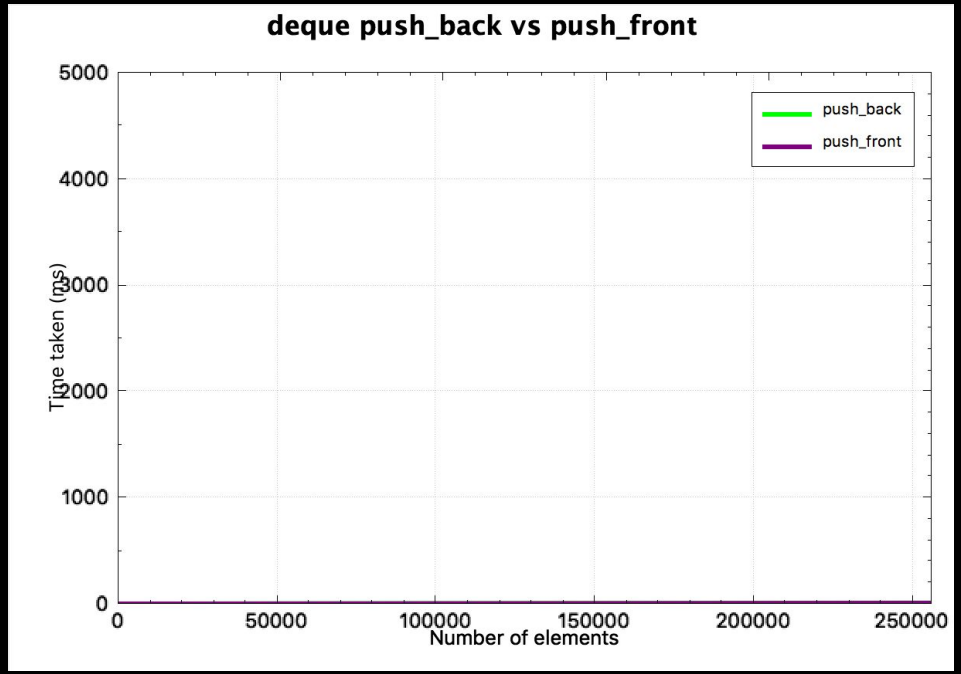
The results:



std::deque<T>

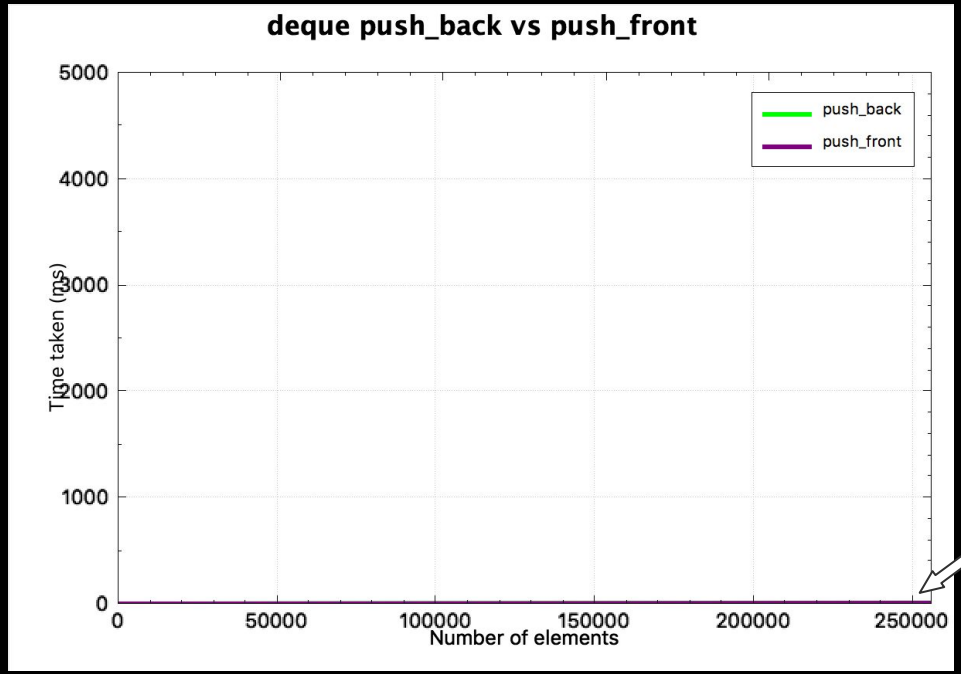
The results:

Same scale as previous graph



std::deque<T>

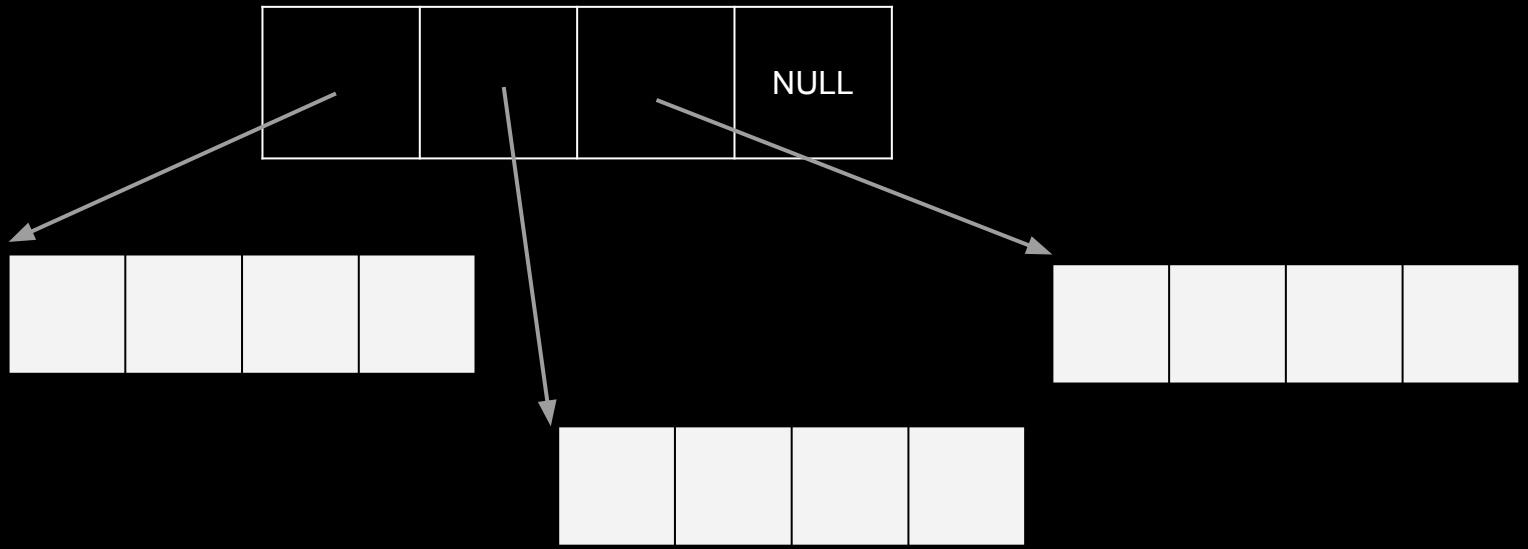
The results:



There are the lines!

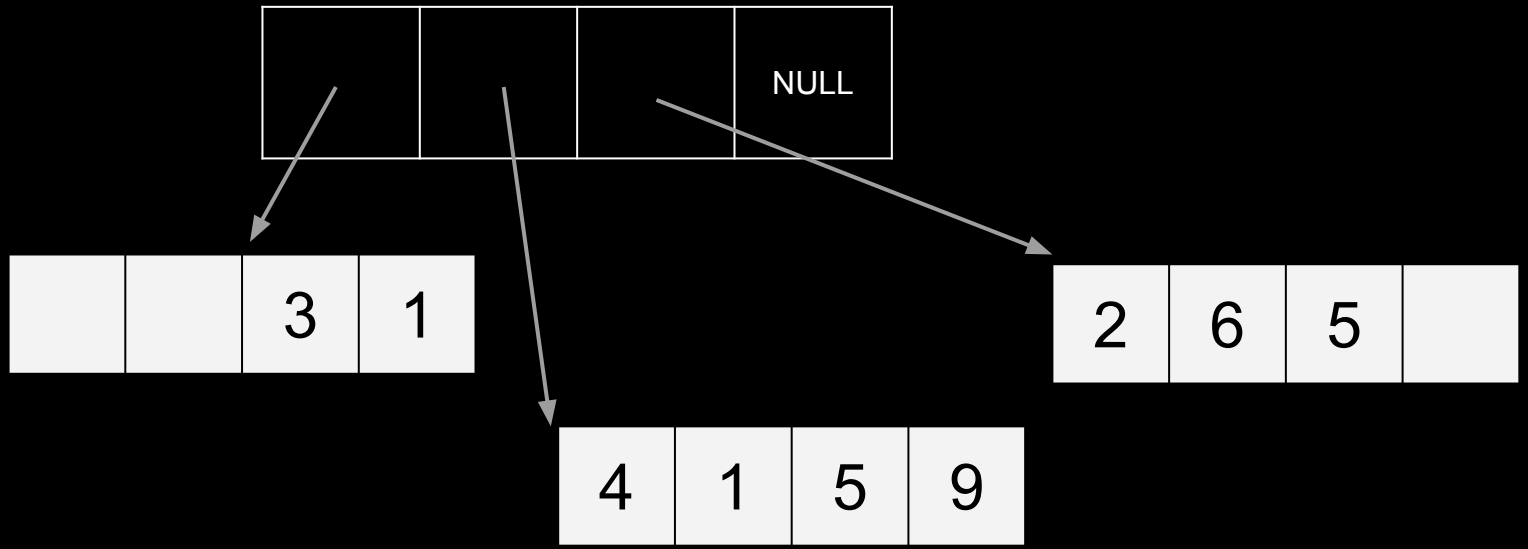
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:



How does `std::deque<T>` work?

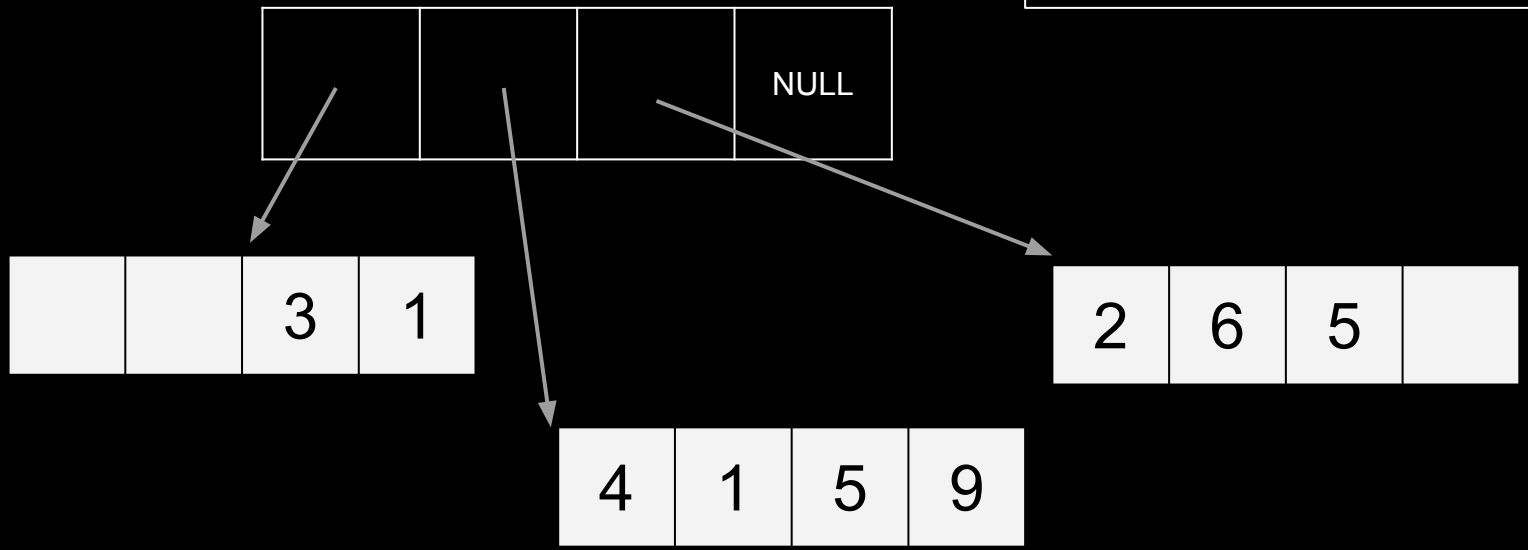
There is no single specific implementation of a deque, but one common one might look like this:



How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

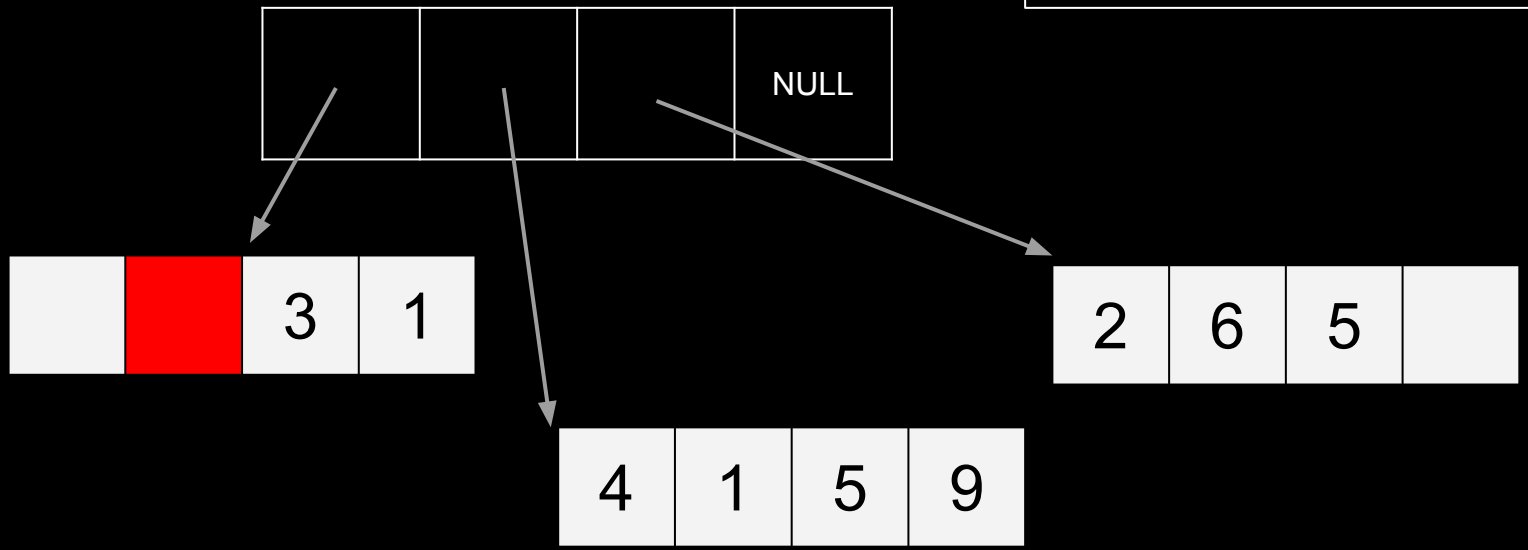
```
deq.push_front(7);
```



How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

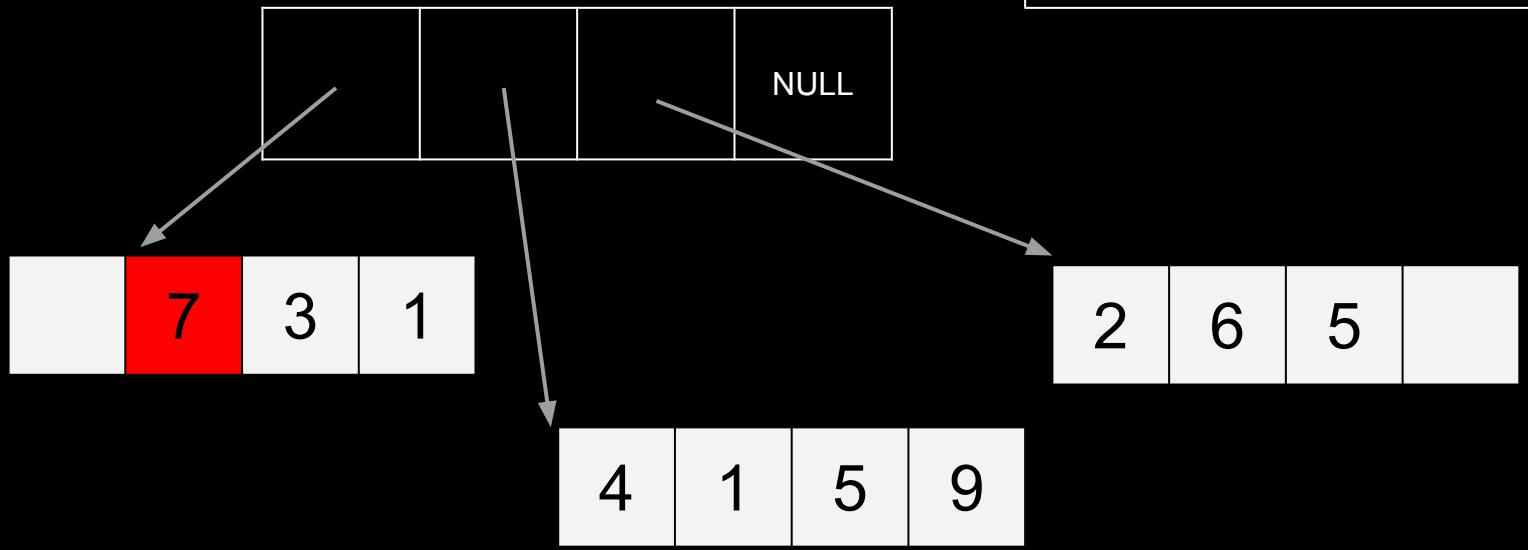
```
deq.push_front(7);
```



How does `std::deque<T>` work?

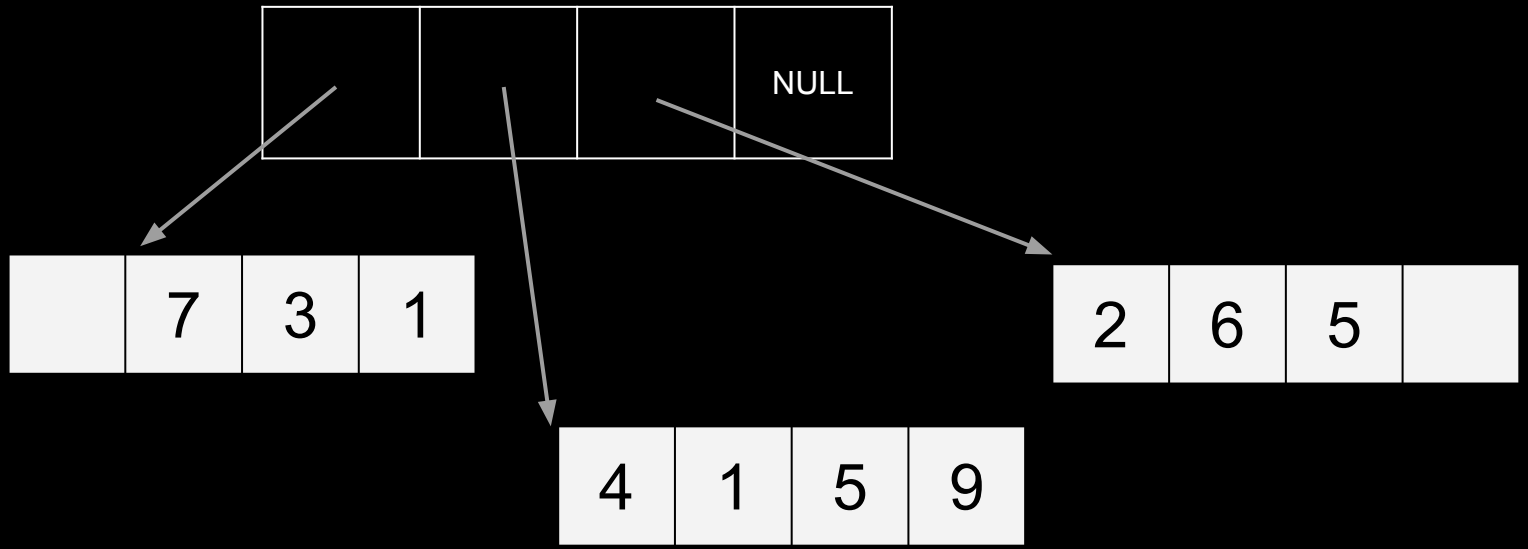
There is no single specific implementation of a deque, but one common one might look like this:

```
deq.push_front(7);
```



How does `std::deque<T>` work?

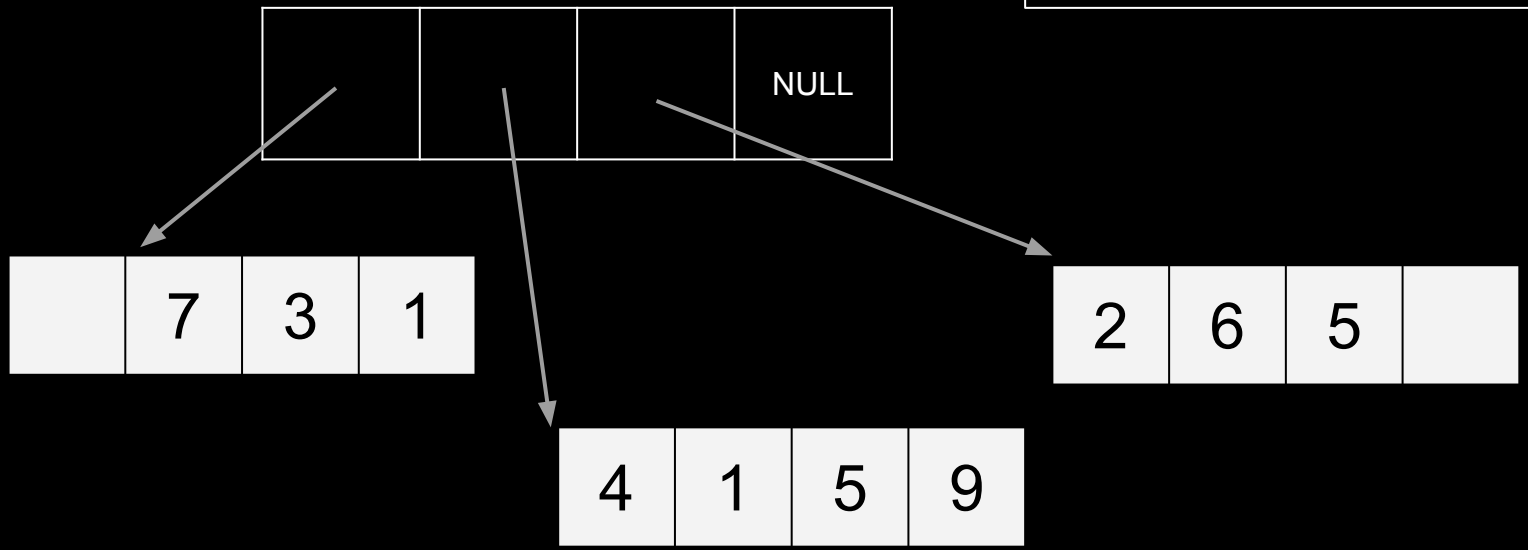
There is no single specific implementation of a deque, but one common one might look like this:



How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

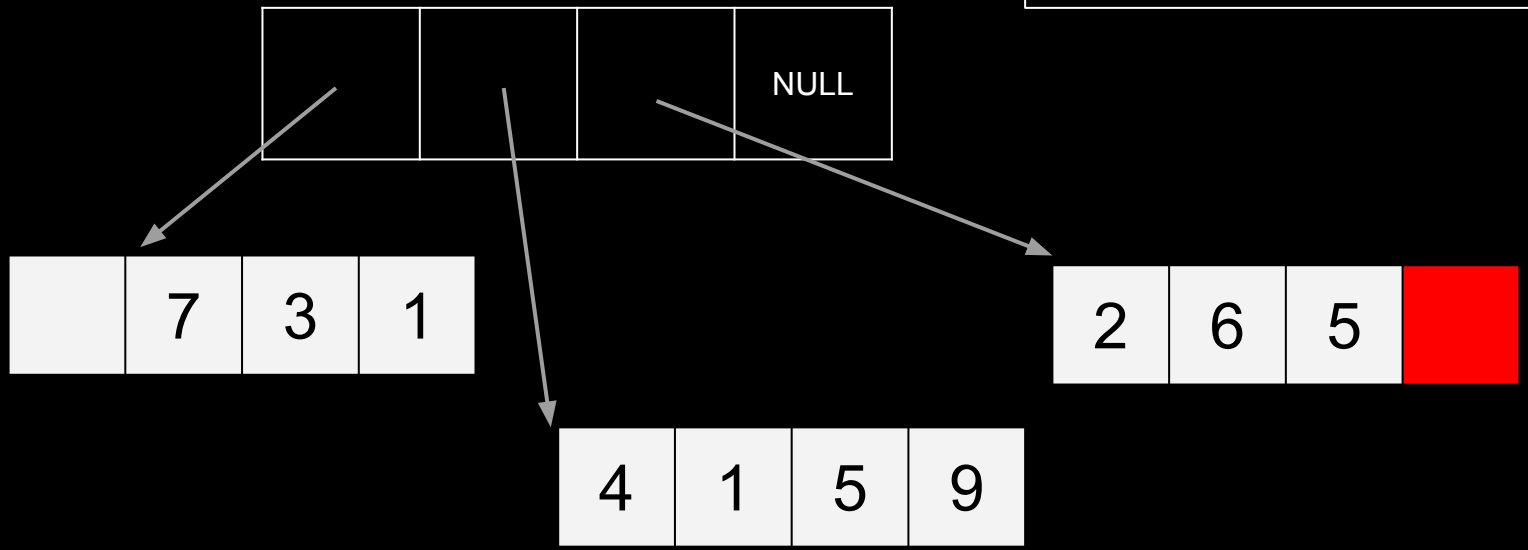
```
deq.push_back(3);
```



How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

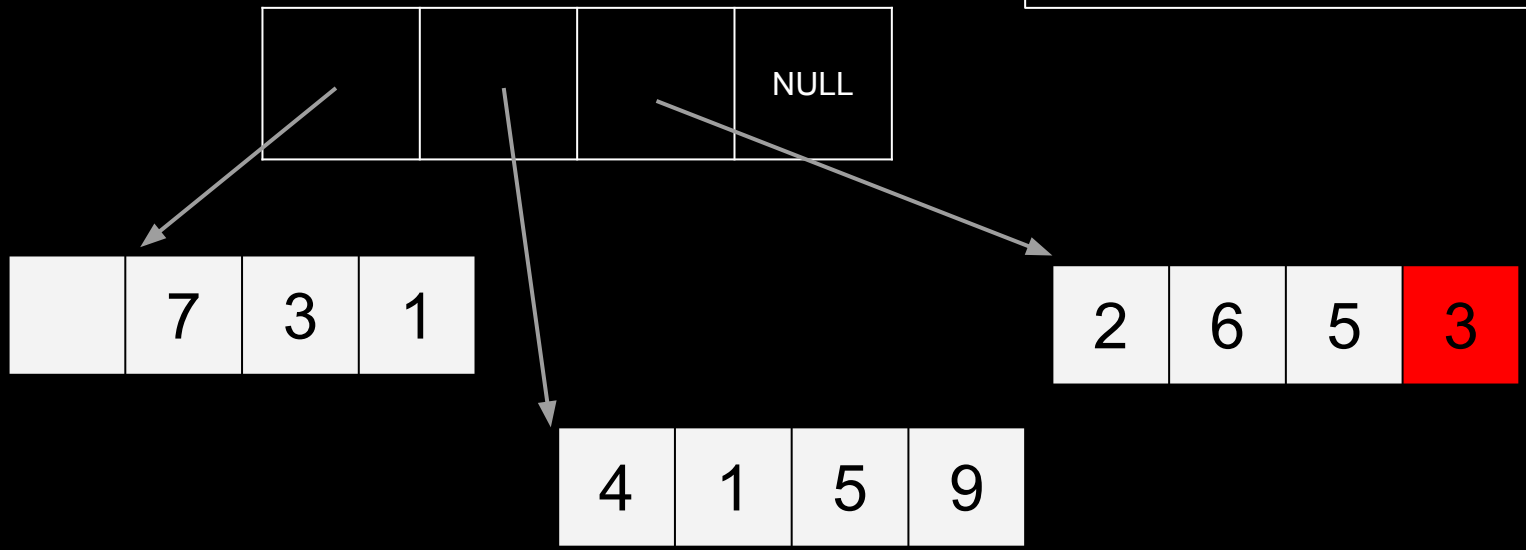
```
deq.push_back(3);
```



How does `std::deque<T>` work?

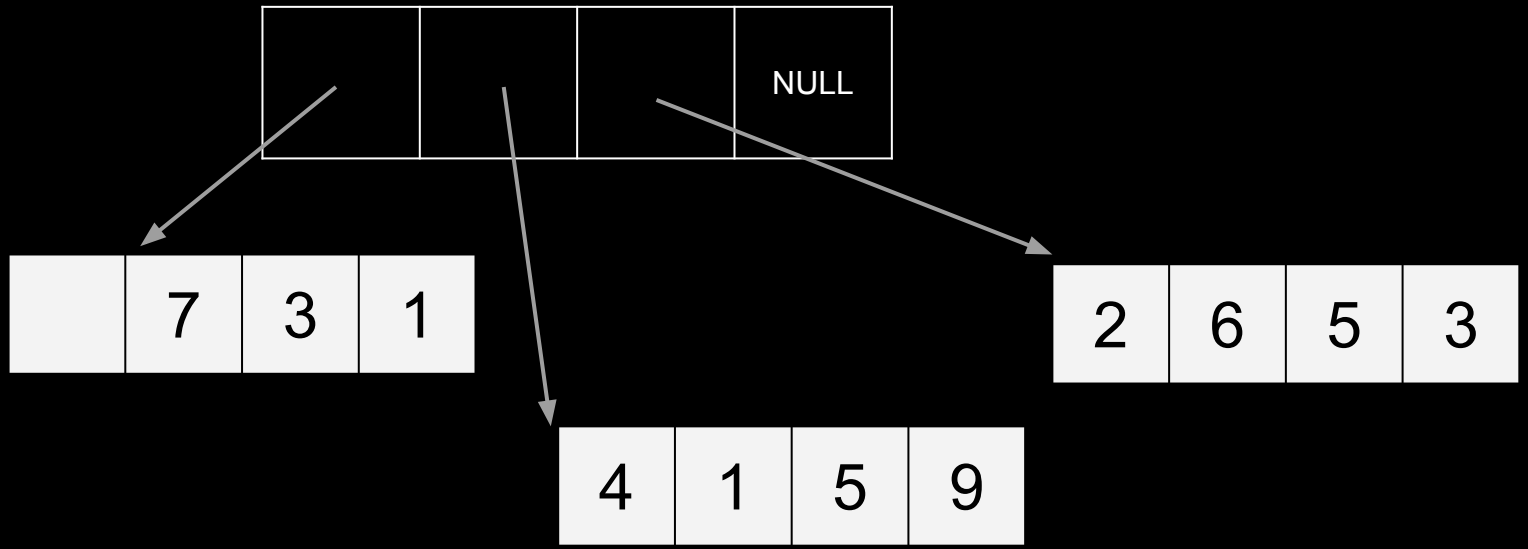
There is no single specific implementation of a deque, but one common one might look like this:

```
deq.push_back(3);
```



How does `std::deque<T>` work?

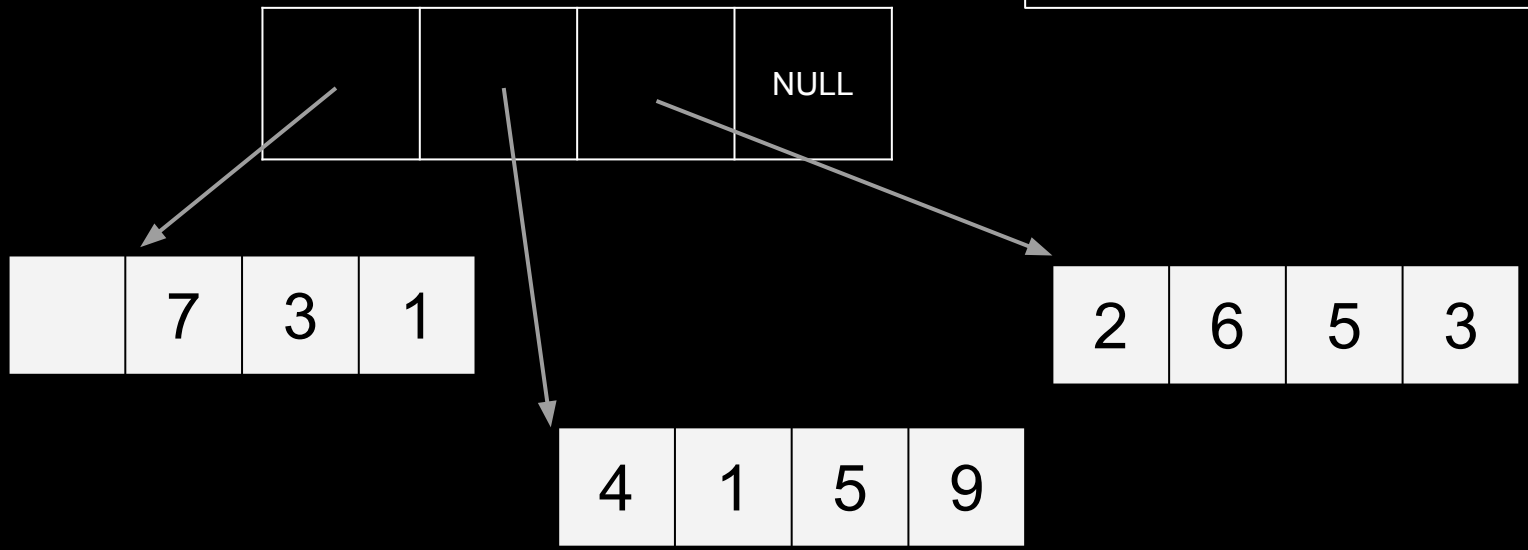
There is no single specific implementation of a deque, but one common one might look like this:



How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

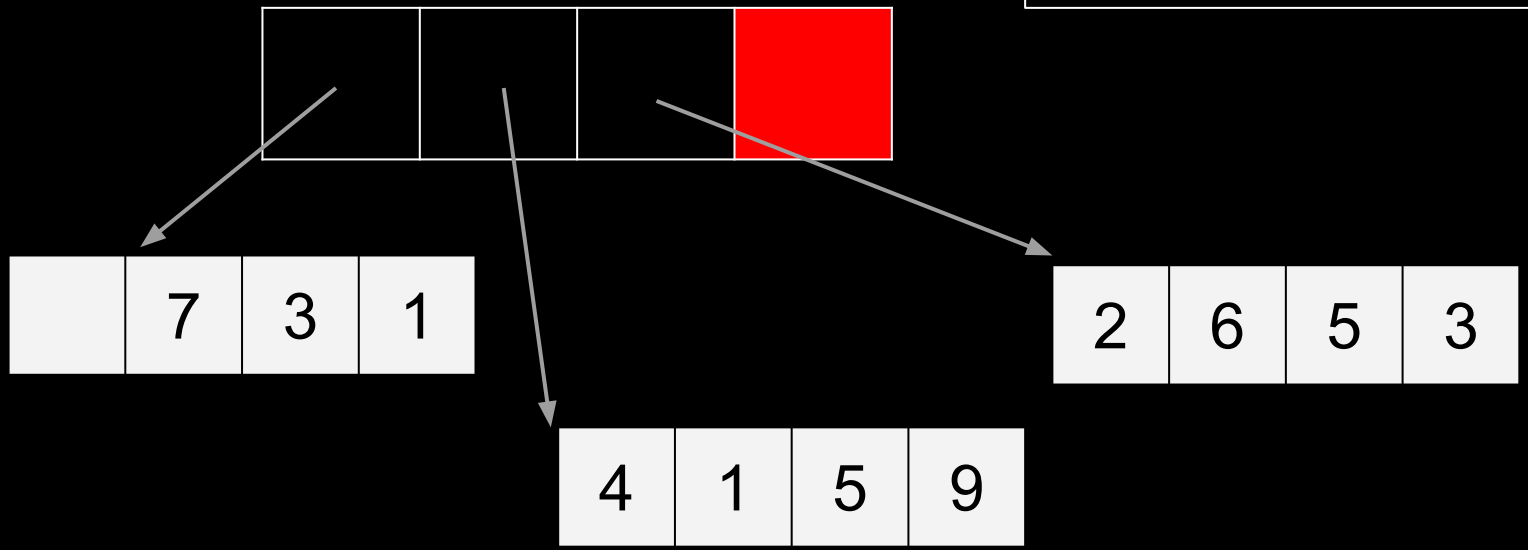
```
deq.push_back(5);
```



How does `std::deque<T>` work?

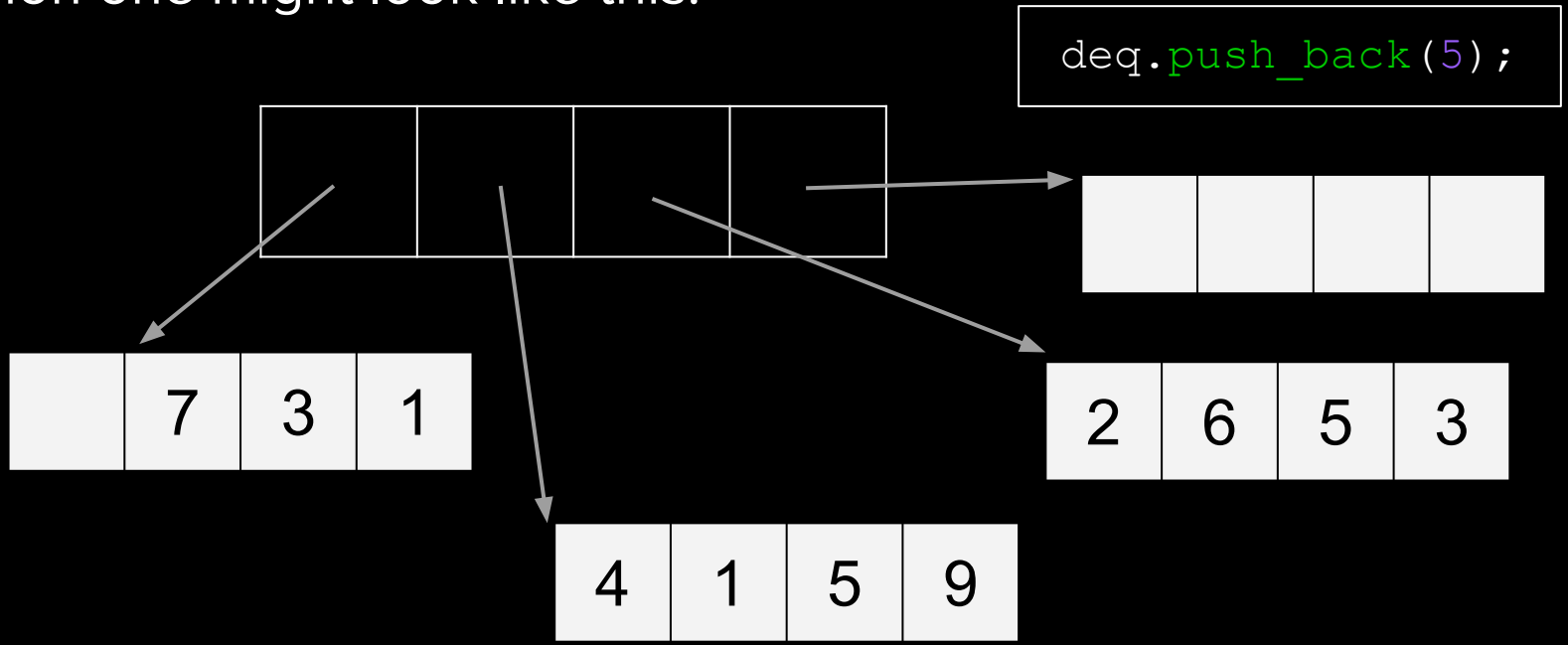
There is no single specific implementation of a deque, but one common one might look like this:

```
deq.push_back(5);
```



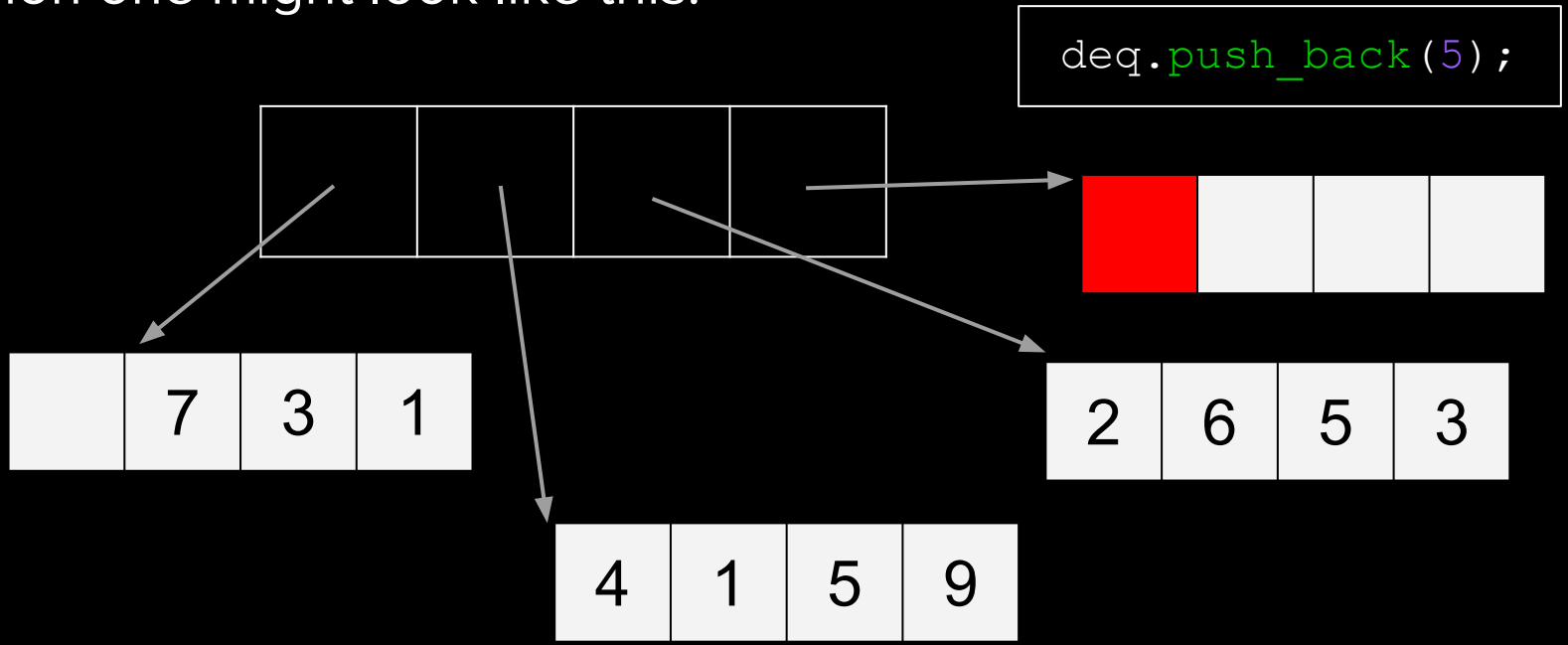
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:



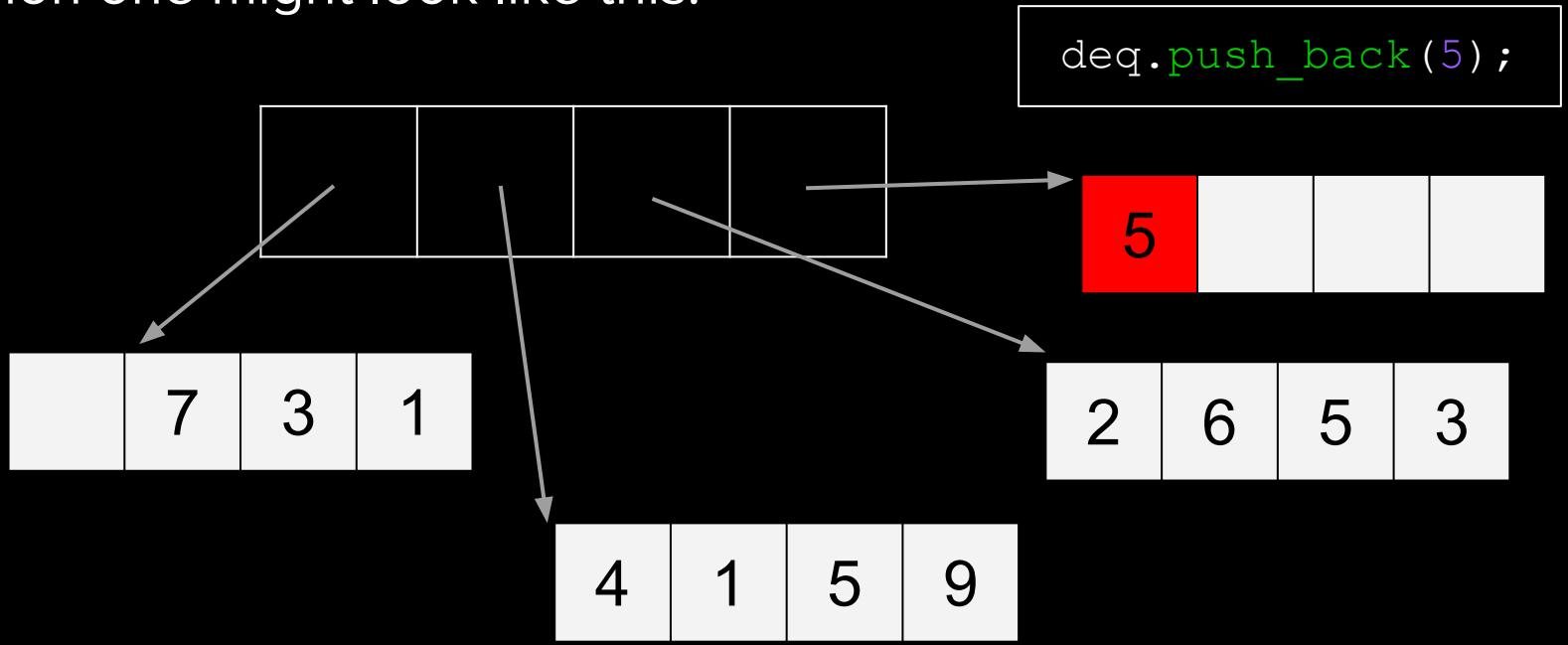
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:



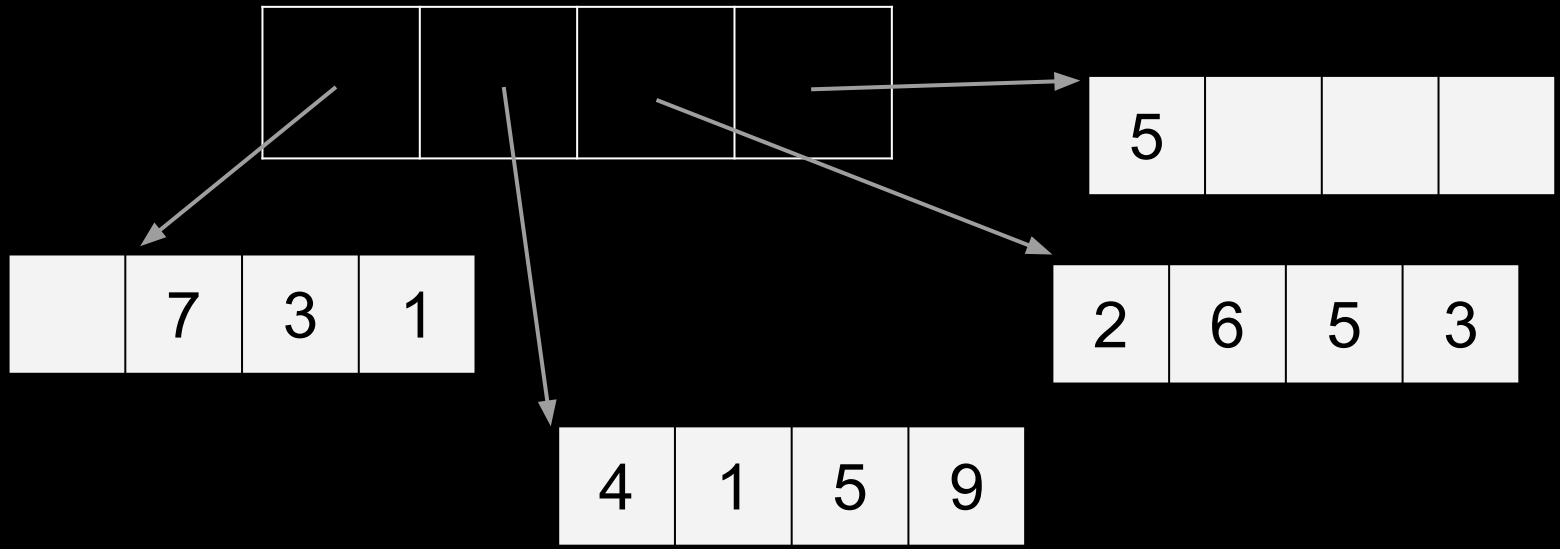
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:



How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:



Wait a minute...

Question

If deque can do everything a vector can do and also has a fast `push_front`...

Why use a vector at all?

Downsides of `std::deque<T>`

Deques support fast `push_front` operations.

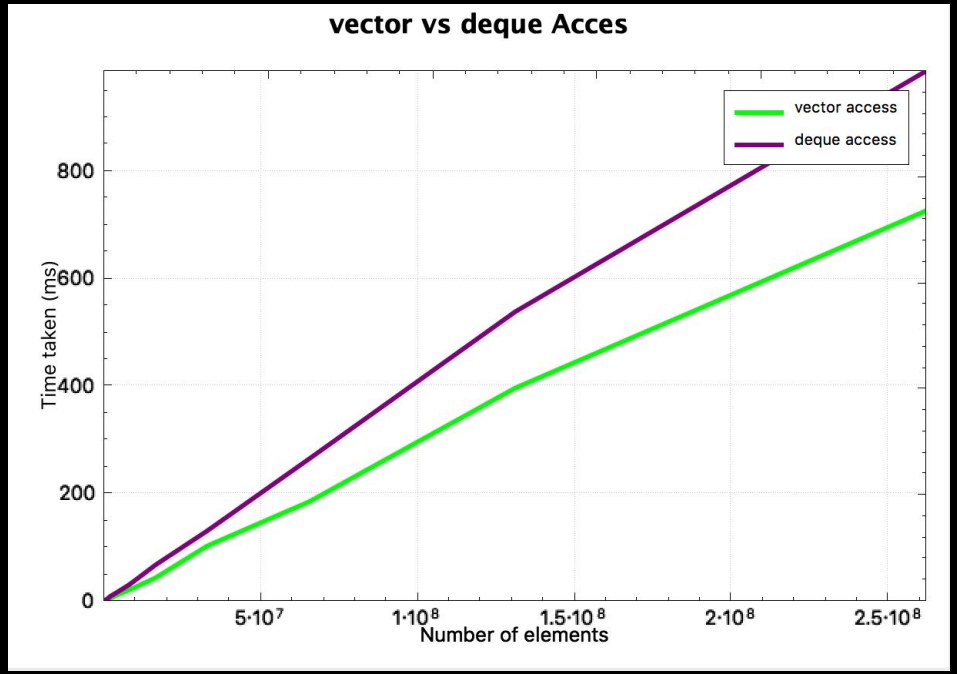
However, for other common operations like element access, vector will always outperform a deque.

Vector vs Deque

(`VecDeqSpeed.pro`)

Downsides of `std::deque<T>`

The results:



Which to Use?

“vector is the type of sequence that should be used by default... deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.”

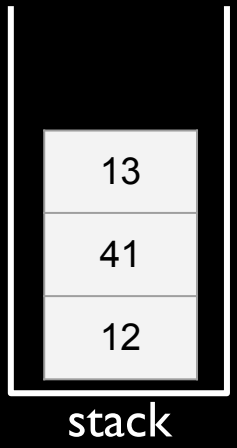
— C++ ISO Standard (section 23.1.1.2):

Questions

Container Adapters

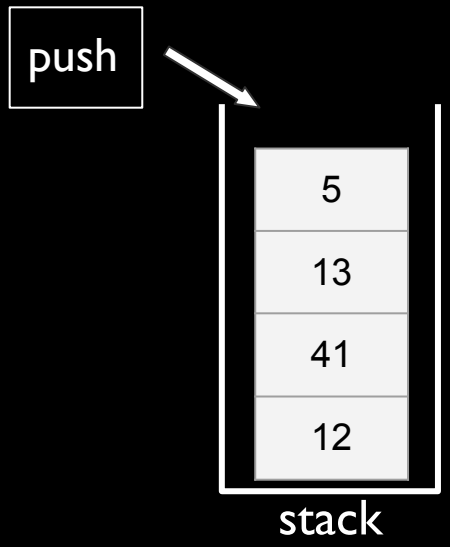
Container Adapters

Recall stacks and queues:



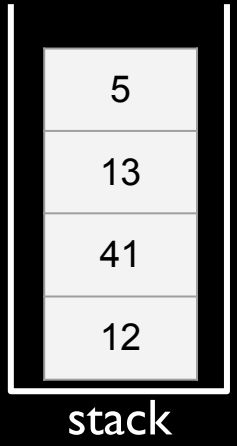
Container Adapters

Recall stacks and queues:



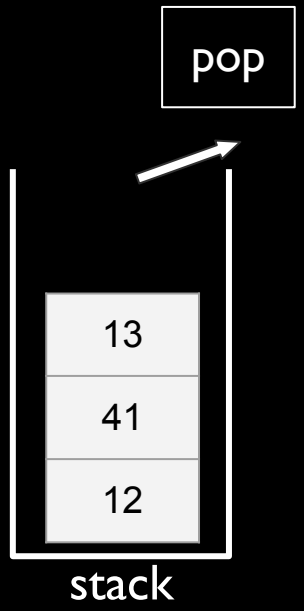
Container Adapters

Recall stacks and queues:



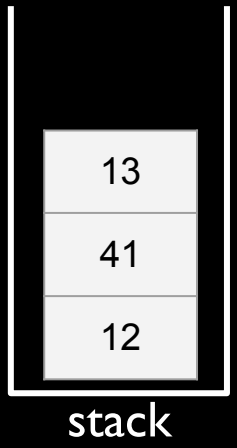
Container Adapters

Recall stacks and queues:



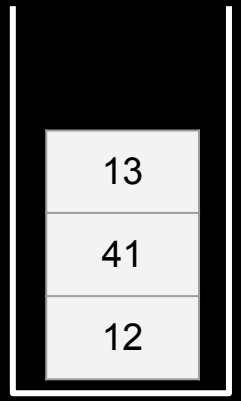
Container Adapters

Recall stacks and queues:

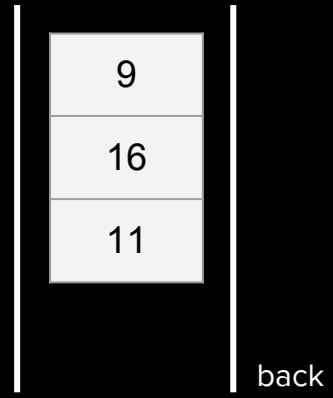


Container Adapters

Recall stacks and queues:



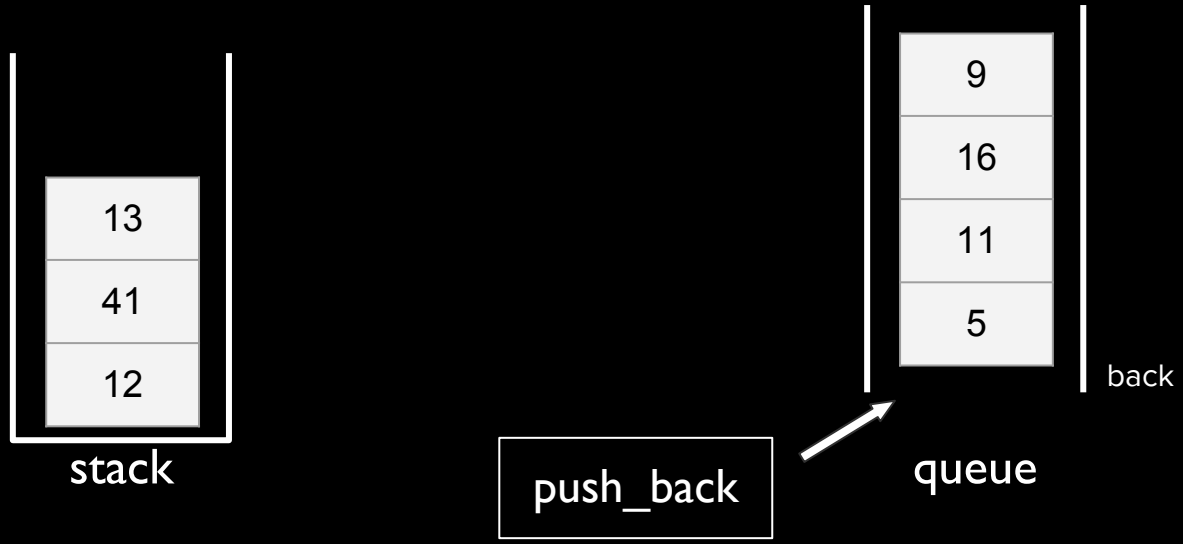
stack



queue

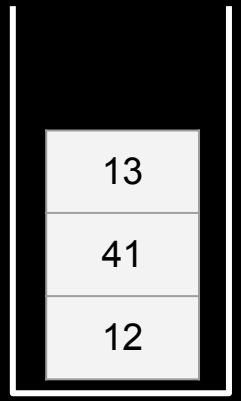
Container Adapters

Recall stacks and queues:

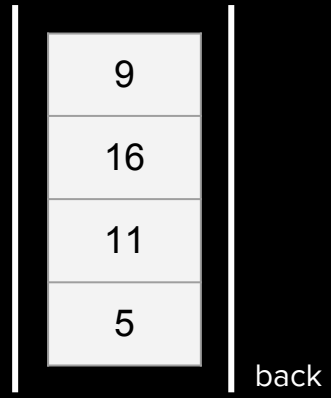


Container Adapters

Recall stacks and queues:



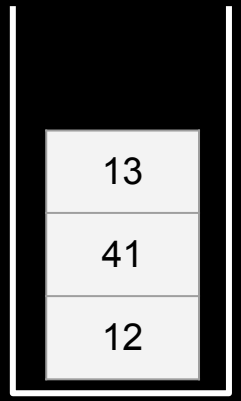
stack



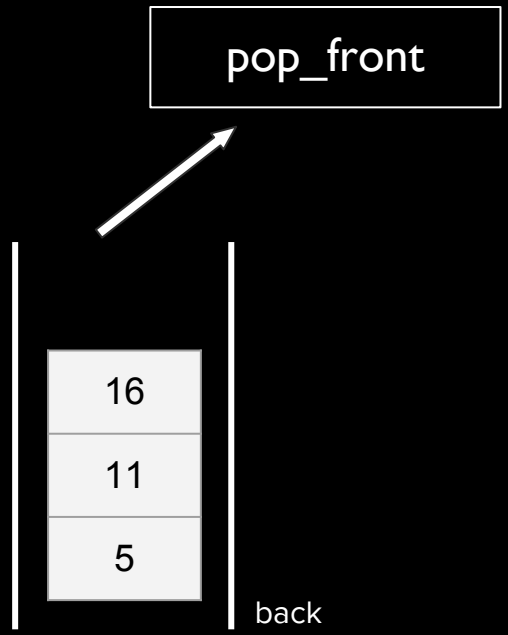
queue

Container Adapters

Recall stacks and queues:



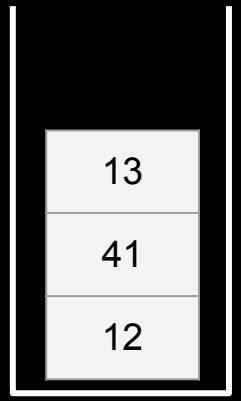
stack



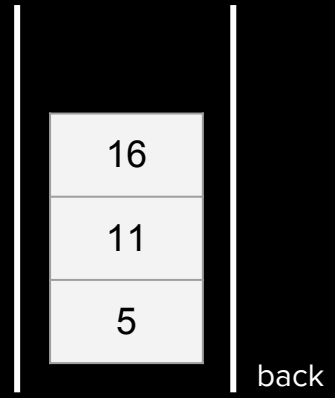
queue

Container Adapters

Recall stacks and queues:



stack



queue

Container Adapters

How can we implement stack and queue using the containers we have?

Stack:

Just limit the functionality of a vector/deque to only allow `push_back` and `pop_back`.

Queue:

Just limit the functionality of a deque to only allow `push_back` and `pop_front`.

Plus only allow access to top element

Container Adapters

For this reason, stacks and queues are knows as container adapters.

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as `Container::value_type`. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:

- `back()`
- `push_back()`
- `pop_back()`

The standard containers `std::vector`, `std::deque` and `std::list` satisfy these requirements.

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The `std::queue` class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as `Container::value_type`. (since C++17)

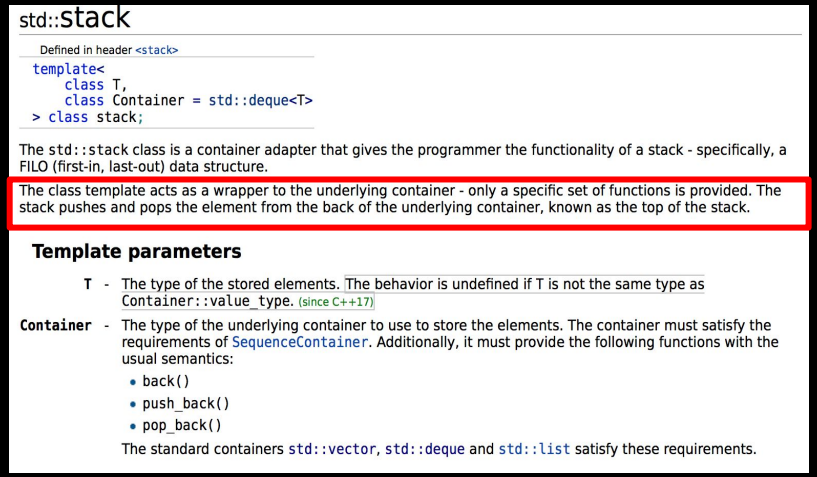
Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:

- `back()`
- `front()`
- `push_back()`
- `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.

Container Adapters

For this reason, stacks and queues are knows as container adapters.



std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The std::stack class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

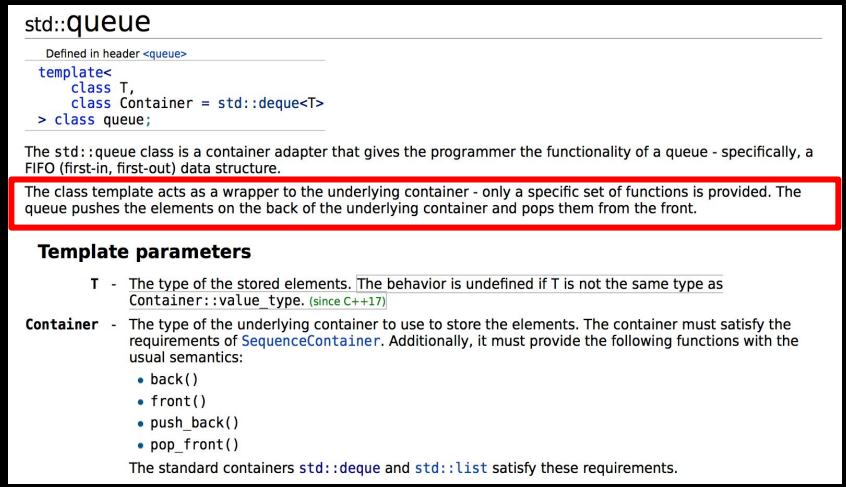
Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:

- back()
- push_back()
- pop_back()

The standard containers std::vector, std::deque and std::list satisfy these requirements.



std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The std::queue class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:

- back()
- front()
- push_back()
- pop_front()

The standard containers std::deque and std::list satisfy these requirements.

Container Adapters

For this reason, stacks and queues are knows as container adapters.

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The std::stack class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:

- back()
- push_back()
- pop_back()

The standard containers std::vector, std::deque and std::list satisfy these requirements.

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The std::queue class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:

- back()
- front()
- push_back()
- pop_front()

The standard containers std::deque and std::list satisfy these requirements.

Next Time

Associative Containers and Iterators