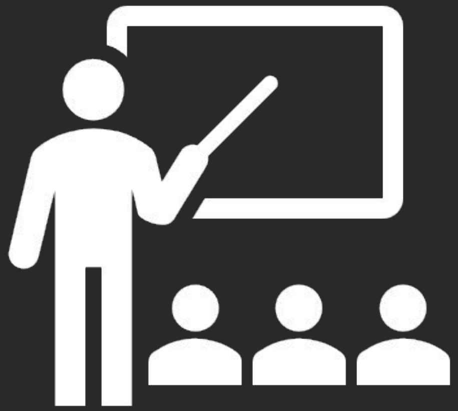
Containers

CS 106L Spring 2020 – Avery Wang and Anna Zeng
Stanford University

20 April 2020

Game Plan



- sequence containers
- container adaptors
- associative containers

Supplemental Material

- Regular slides: stuff we will go through lecture, absolutely necessary to understand future lectures.
- Supplemental material: important to know if you want to be considered "proficient" in C++ by the end of the class.
- Highly recommend you review the supplemental material.

Key questions we will answer today

- when should each container be used?
- how is the STL different from the Stanford libraries?
- what are some best practices and common pitfalls?

- homework: practice using the containers to solve problems.
- if time permits at the end, I'll do one problem

We'll assume you're familiar with

Stanford Library

- Vector<T>
- Stack<T>, Queue<T>
- Map<T>, Set<T>

C++ details we will cover

Library

- std::vector
- std::deque/list
- std::map/unordered_map
- std::set/unordered_set
- std::stack/queue

Overview of STL

Adaptors

Containers

Iterators

Functions

Algorithms

Sequence Containers

1. std::vector - Stanford vs. STL
2. std::vector - safety and efficiency
3. std::deque vs. std::vector

Stanford vs. STL vector: very similar!

```
// Stanford
Vector<char> vec{'a', 'b', 'c'};

vec[0] = 'A';
cout << vec[vec.size()-1];

for (int i = 0; i < vec.size(); ++i) {
  ++vec[i];
}

for (auto& elem : vec) {
  --elem;
}
```

```
// STL
std::vector<char> vec{'a', 'b', 'c'};

vec[0] = 'A';
cout << vec[vec.size()-1]; // or vec.back()

for (size_t i = 0; i < vec.size(); ++i) {
  ++vec[i];
}

for (auto& elem : vec) {
  --elem;
}
```

Answer on chat: what is different?

Stanford vs. STL vector: fairly close!

```cpp
// Stanford
Vector<char> vec(3, 'c');
if (vec.isEmpty() || vec.size() == 3) {
  vec.clear();
}

vec[-1] = 1; // BAD, error thrown!
Vector<char> vec2(3, 'c');
if (vec == vec2) {
  vec2.remove(vec2.size()-1);
}
```

```cpp
// STL
std::vector<char> vec(3, 'c');
if (vec.empty() || vec.size() == 3) {
  vec.clear();
}

vec[-1] = 1; // BAD, but no error thrown!
std::vector<char> vec2(3, 'c');
if (vec == vec2) {
  vec2.pop_back();
}
```

Answer on chat: what is different?

Stanford vs. STL vector: a summary

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 (* does not bounds check!)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);` `int k = v[i]; (*)`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (* does not bounds check!)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;` `v[i] = k; (*)`

Stanford vs. STL vector: a summary

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 (* does not bounds check!)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);` `int k = v[i]; (*)`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (* does not bounds check!)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;` `v[i] = k; (*)`

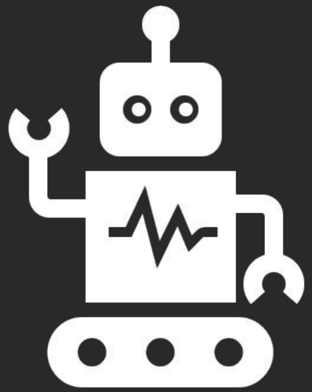
What is missing on this chart?

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 (* does not bounds check!)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);` `int k = v[i]; (*)`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (* does not bounds check!)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;` `v[i] = k; (*)`

Stanford vs. STL vector: a summary

What you want to do	Stanford Vector<int>	std::vector<int>
Add j to the front of the vector	`v.insert(0, k);`	`v.insert(v.begin(), k);`
Insert k at some index i	`v.insert(i, k);`	`v.insert(v.begin()+i, k);`
Remove the element at index i	`v.remove(i);`	`v.erase(v.begin()+i);`
Get the sublist in indices [i, j)	`v.subList(i, j);`	`vector<int> c (v.begin()+i,v.begin()+j);`
Create a vector that is two vectors appended together.	`vector<int> v = v1 + v2;`	`// kinda complicated`

This'll require understanding iterators.
Next lecture!



Questions

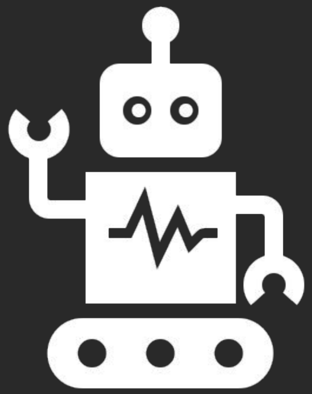
Answer 2 questions.

operator[] does not perform bounds checking

You'll see operator[] used, but rarely will you see at().

This follows the C++ philosophy to never waste time.

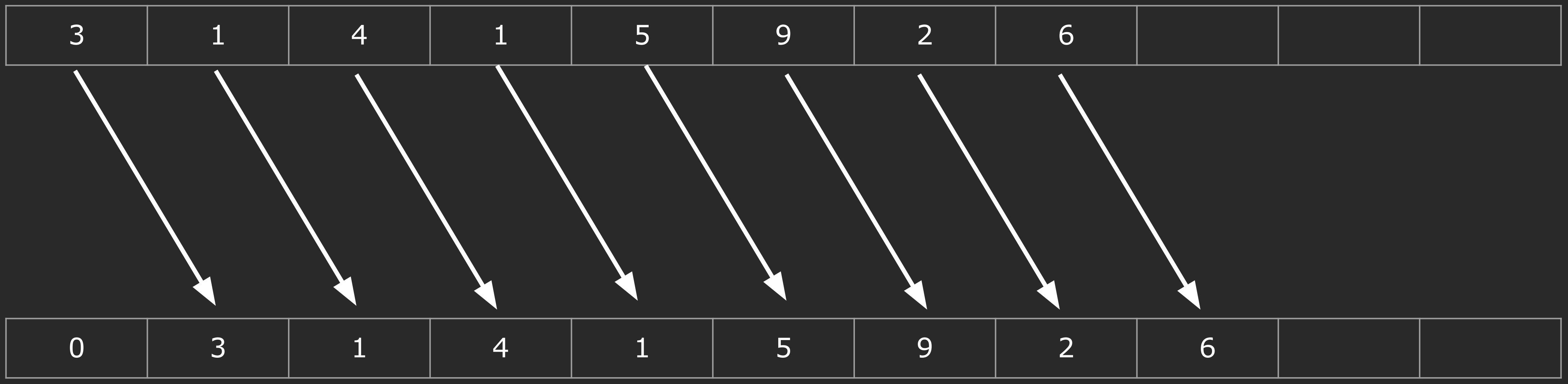
```
std::vector<int> vec{5, 6};   // {5, 6}
vec[1] = 3;                   // {5, 3}
vec[2] = 4;                   // undefined behavior
```



Example

Front insertion speed.

vector does not have a push_front function



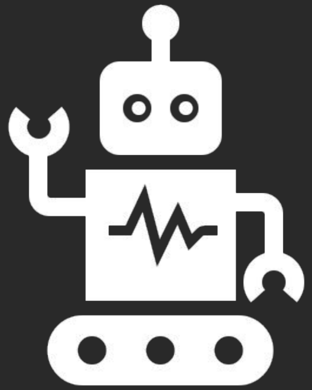
vector does not have a push_front function

vector does not have a push_front function

```
v.push_front(0);                    // not a real function
```

Recurring C++ pattern: don't provide functions which might be mistaken to be efficient when it's not.

sidenote: more specifically, linear time complexity.



Questions

Answer 2 questions.

What if you really wanted fast insertion to the front?

std::deque provides fast insertion anywhere

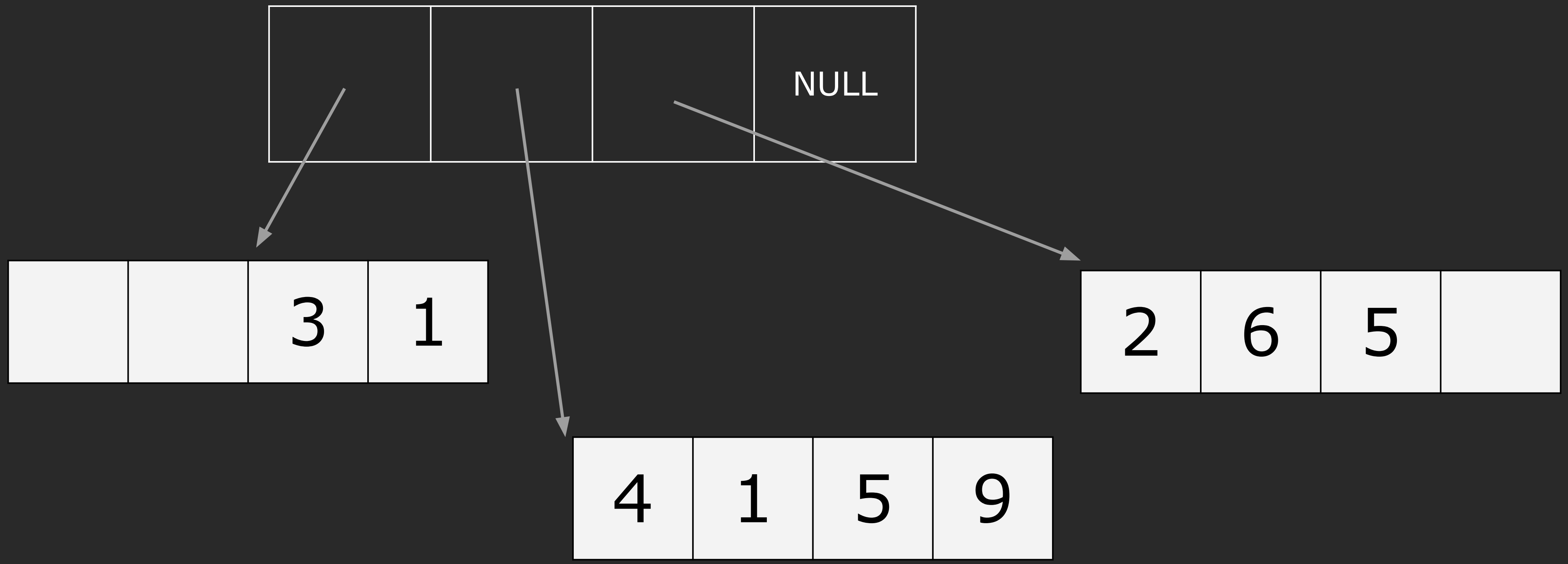
std::deque has the exact same functions as std::vector
but also has push_front and pop_front.

```
std::deque<int> deq{5, 6};     // {5, 6}
deq.push_front(3);             // {3, 5, 6}
deq.pop_back(4);               // {3, 5}
deq[1] = -2;                   // {3, -2}
```

How is a deque implemented?

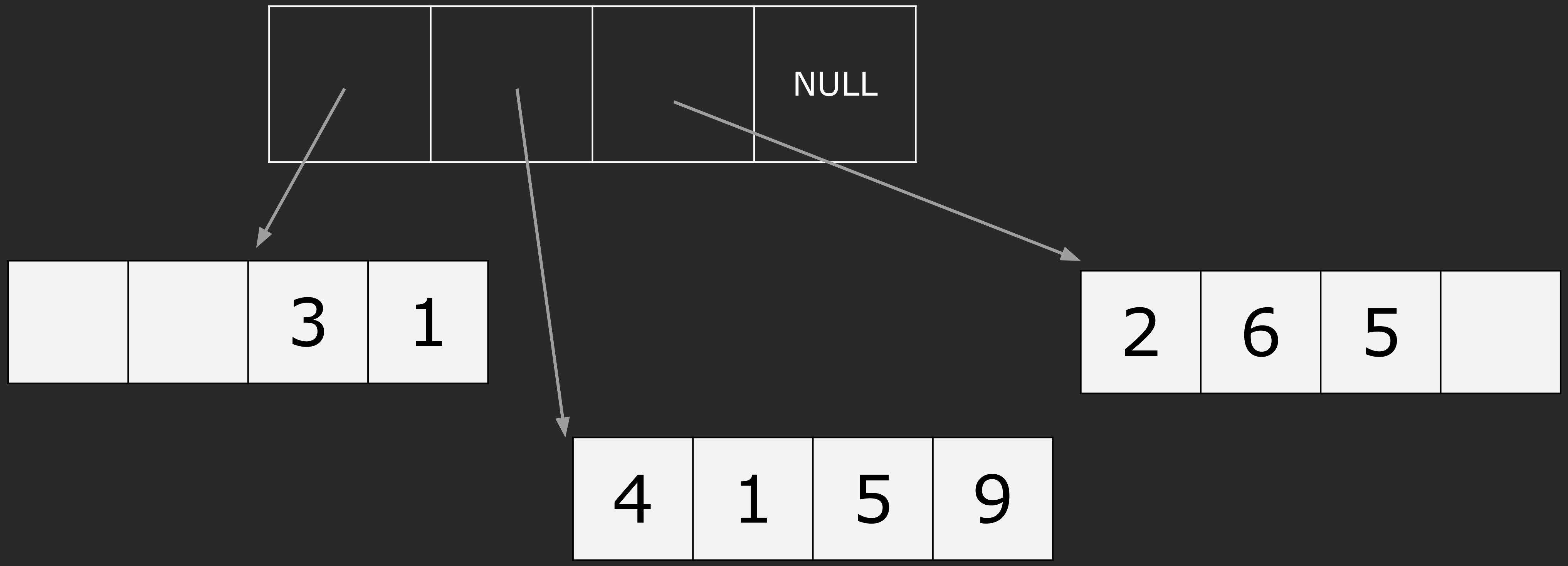
How does std::deque<T> work?

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



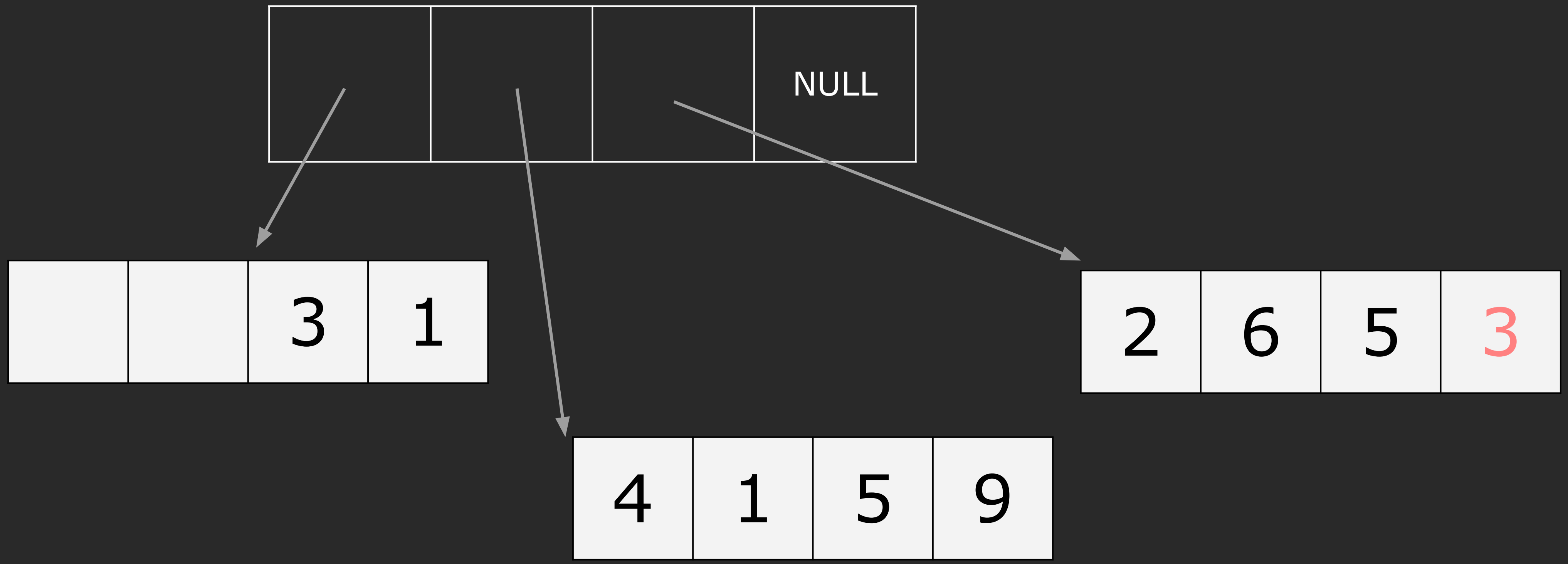
How does std::deque<T> work?

How would you do push_back(3)?



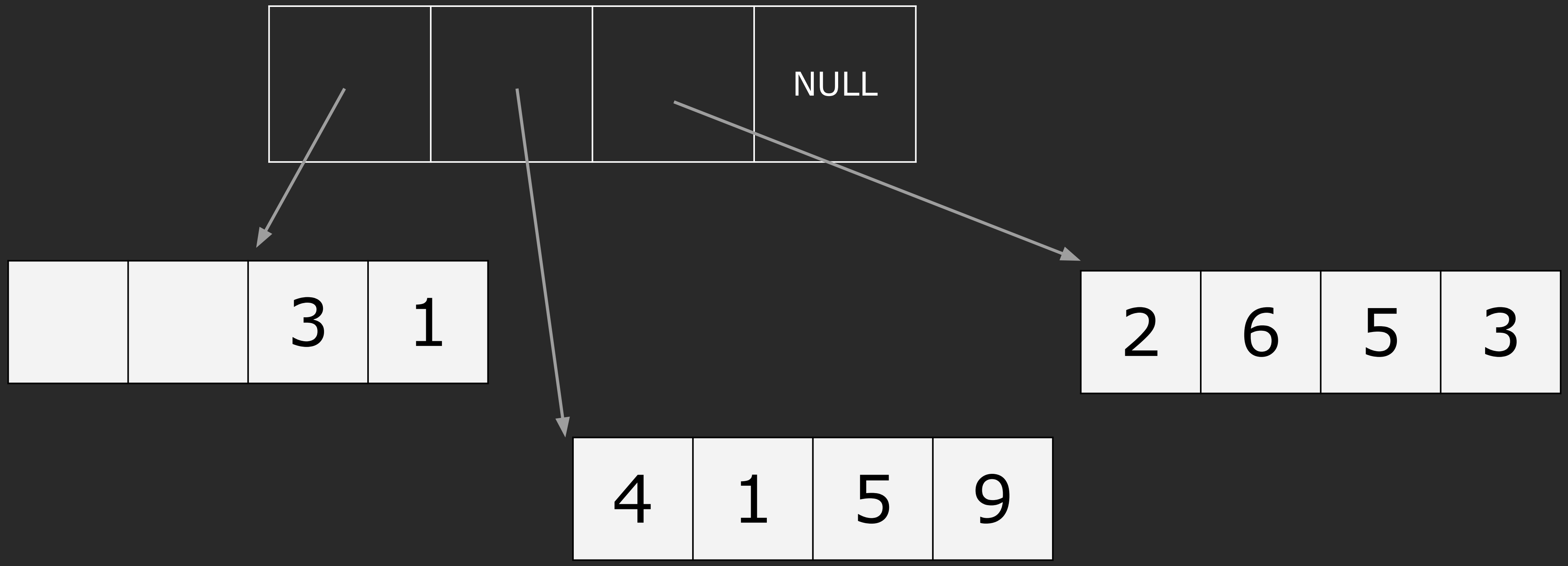
How does std::deque<T> work?

How would you do push_back(3)?



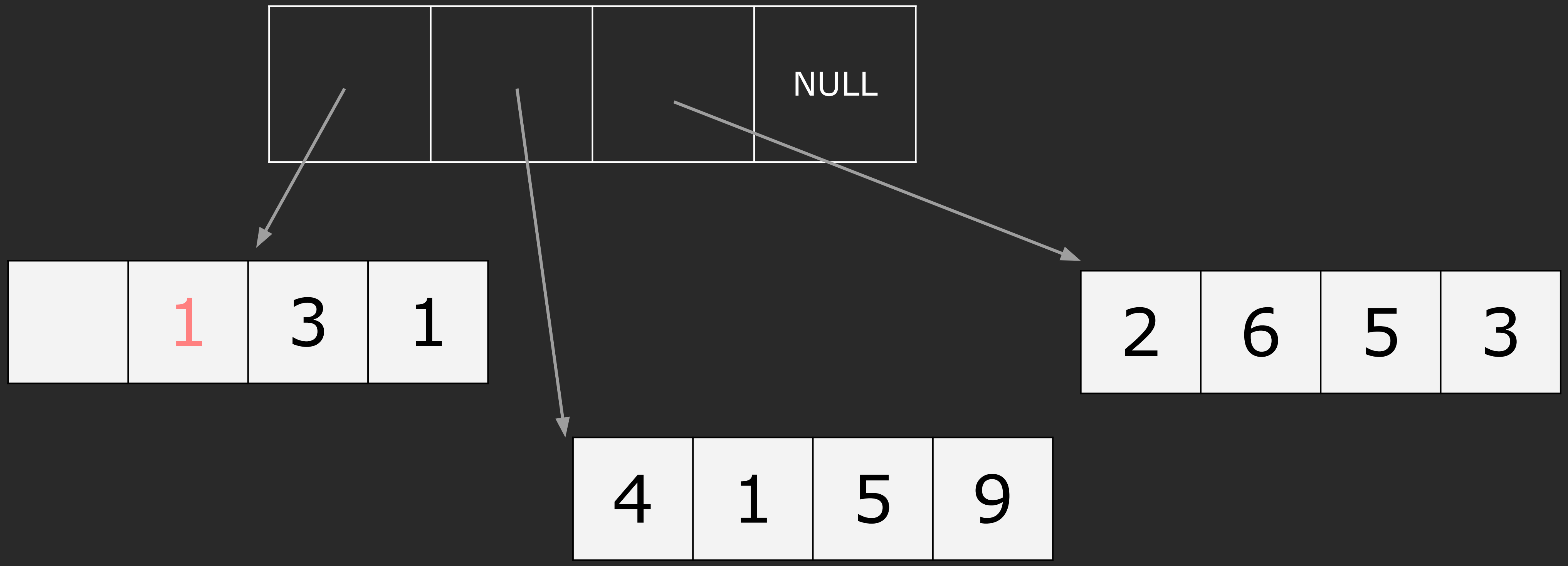
How does std::deque<T> work?

How would you do push_front(1)?



How does std::deque<T> work?

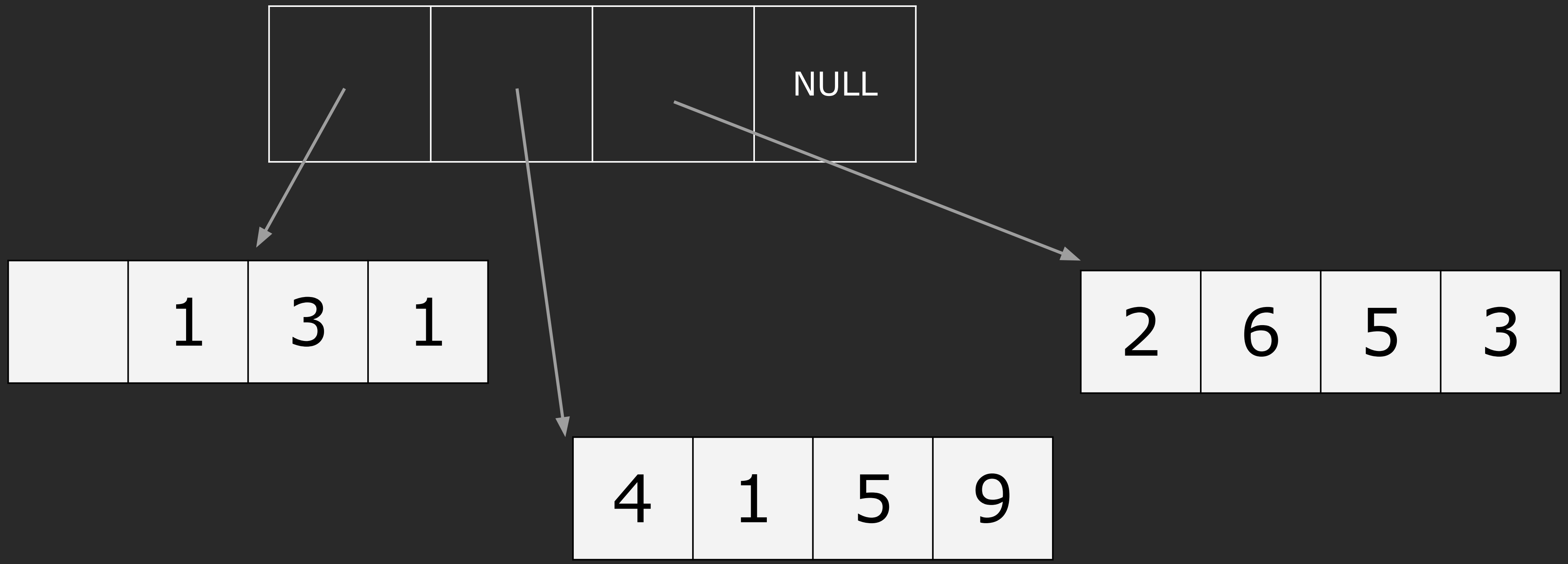
How would you do push_front(1)?



Supplemental Material

How does std::deque<T> work?

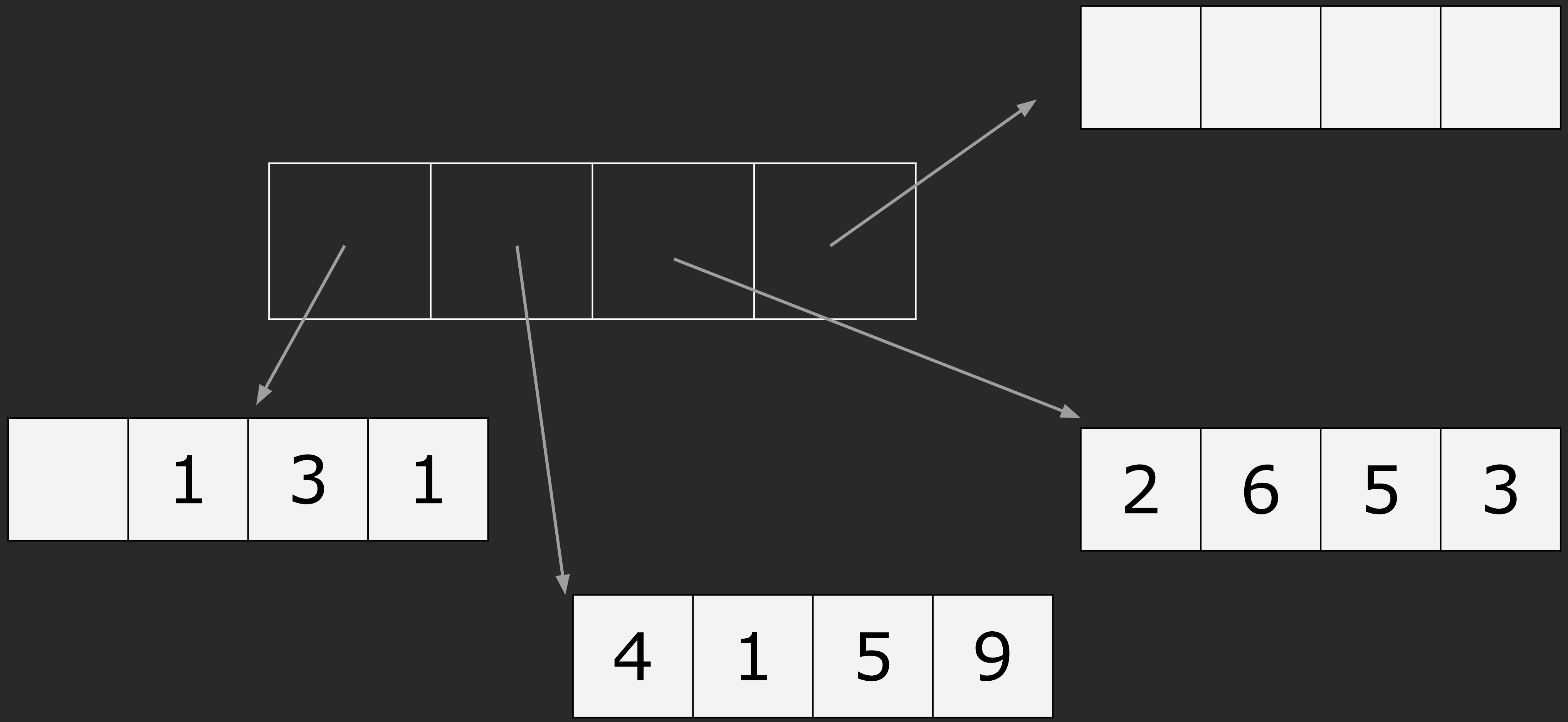
How would you do push_back(7)?



Supplemental Material

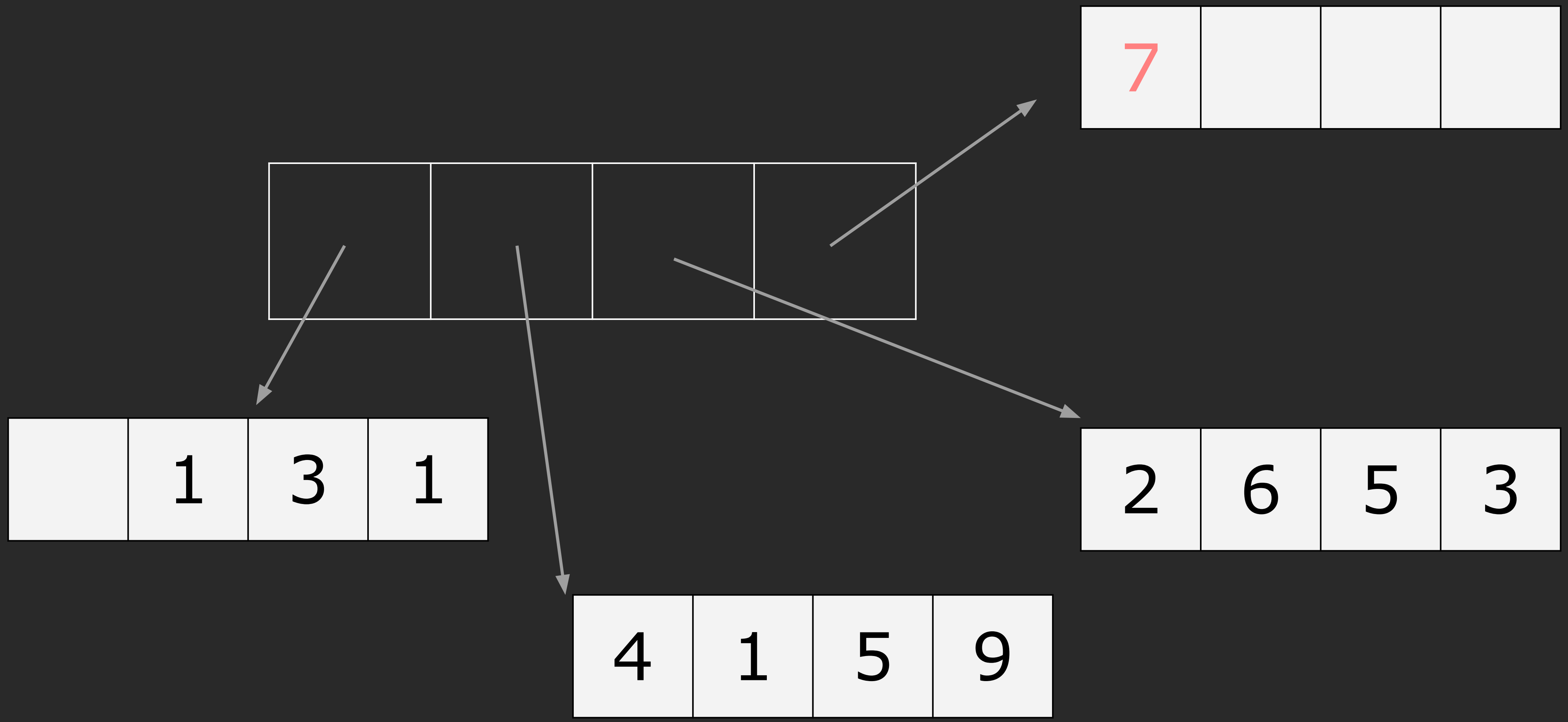
How does std::deque<T> work?

How would you do push_back(7)?



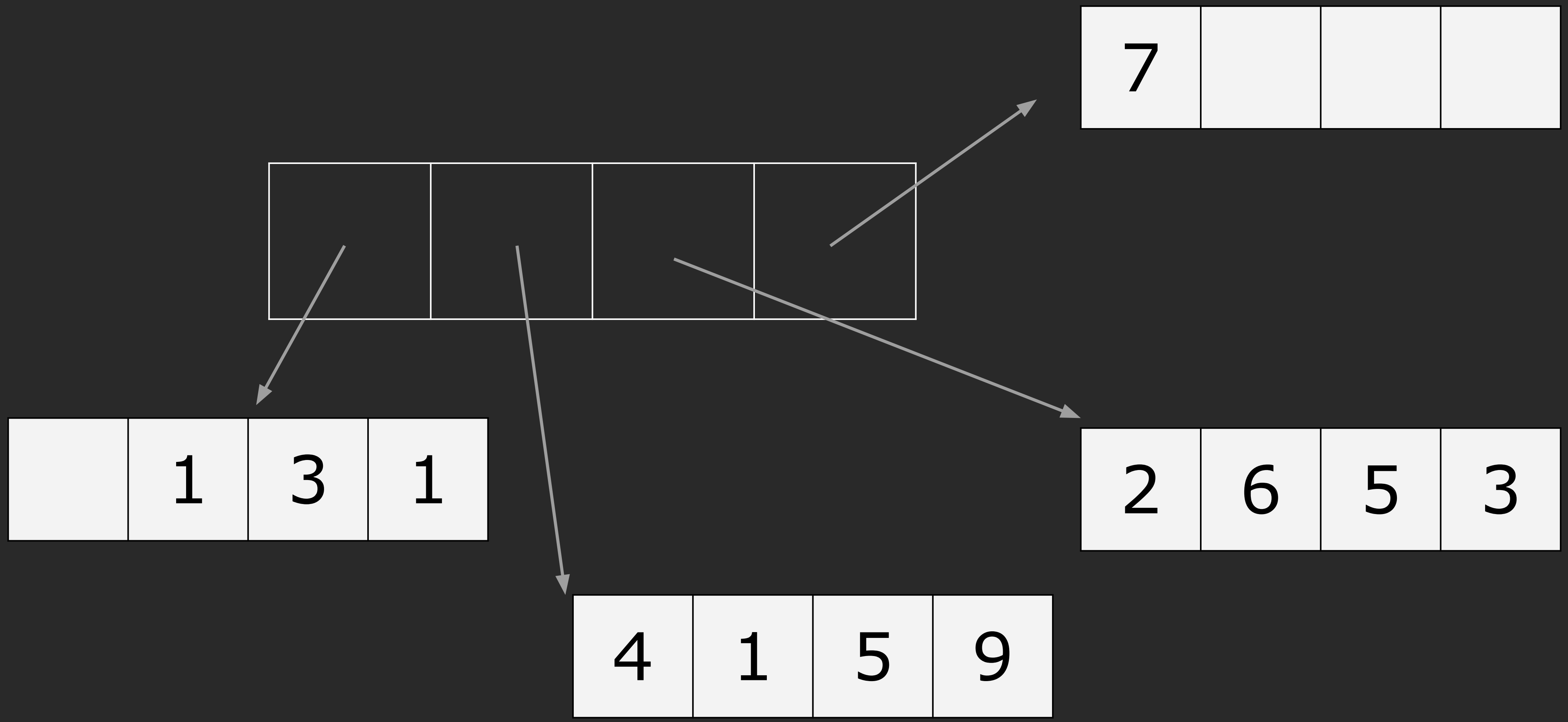
How does std::deque<T> work?

How would you do push_back(7)?



How does std::deque<T> work?

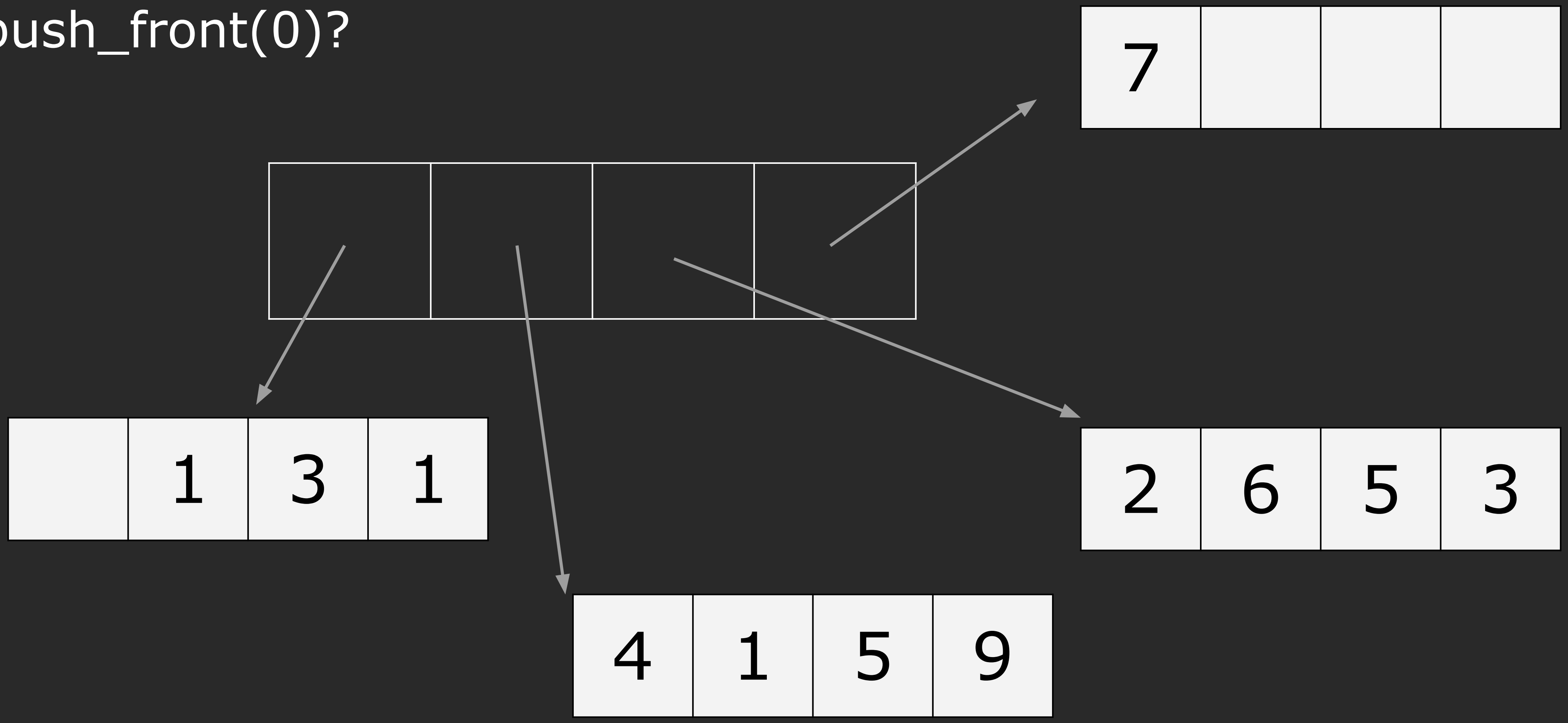
How would you do push_back(7)?



How does std::deque<T> work?

How would you do push_front(8)

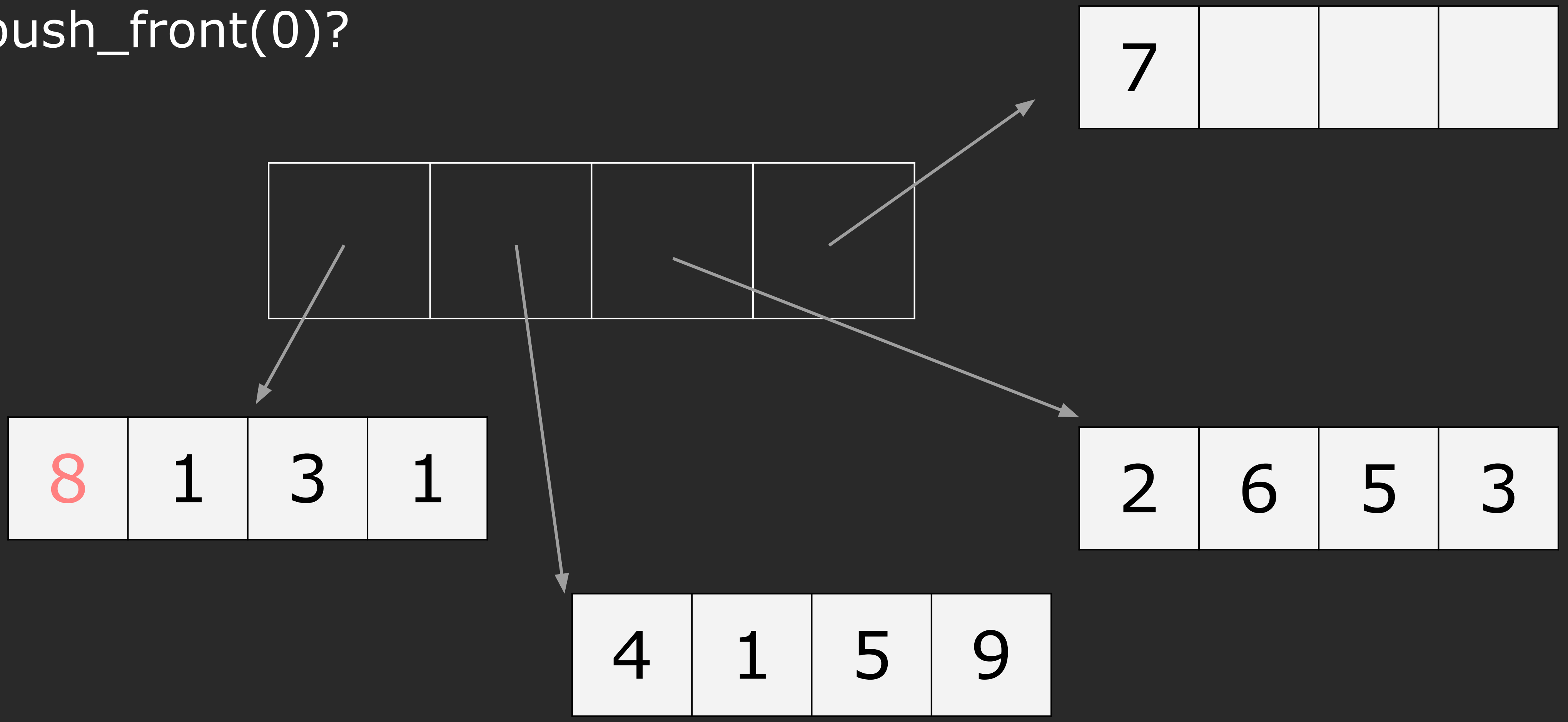
then push_front(0)?



Supplemental Material

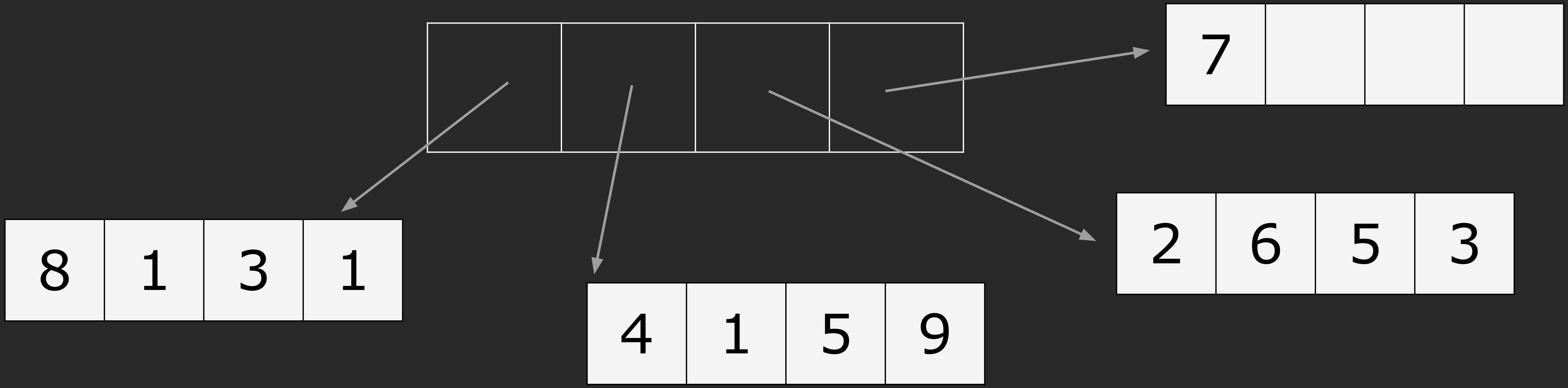
How does std::deque<T> work?

How would you do push_front(8)
then push_front(0)?

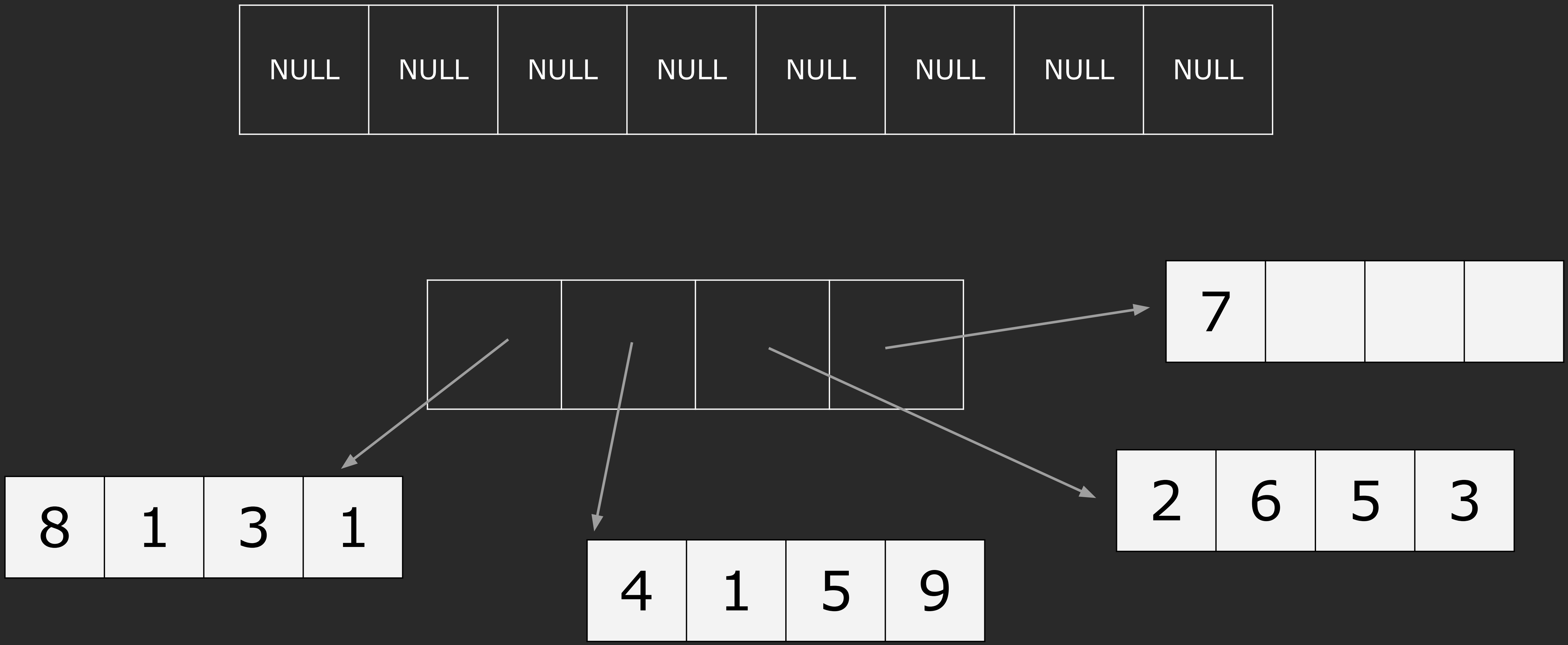


Supplemental Material

How does std::deque<T> work?

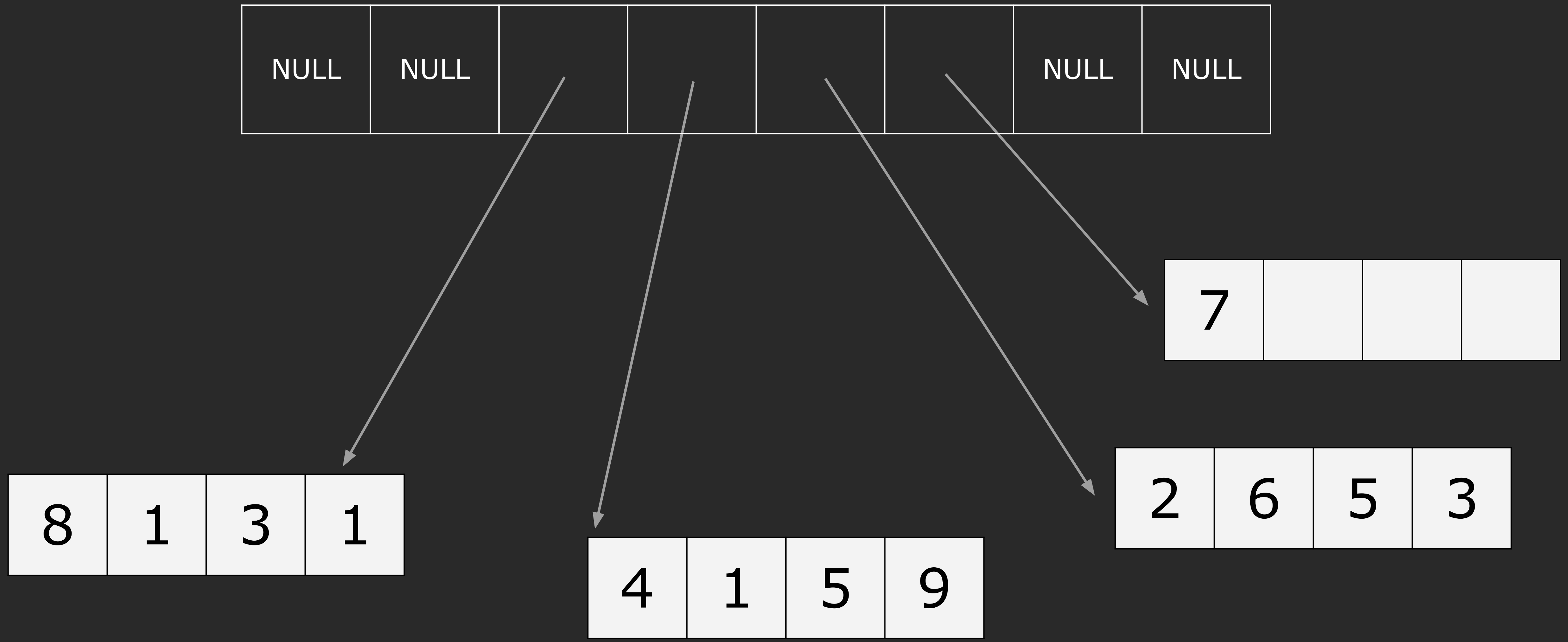


How does std::deque<T> work?

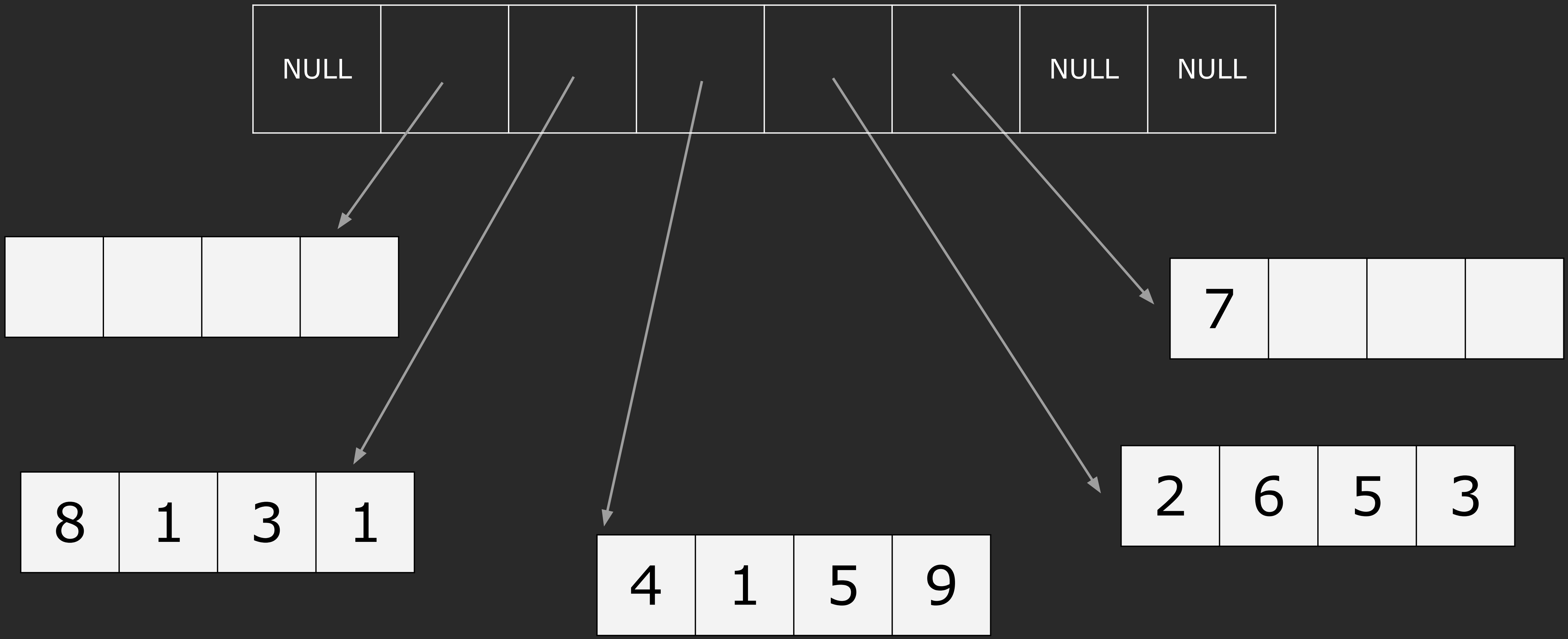


Supplemental Material

How does std::deque<T> work?

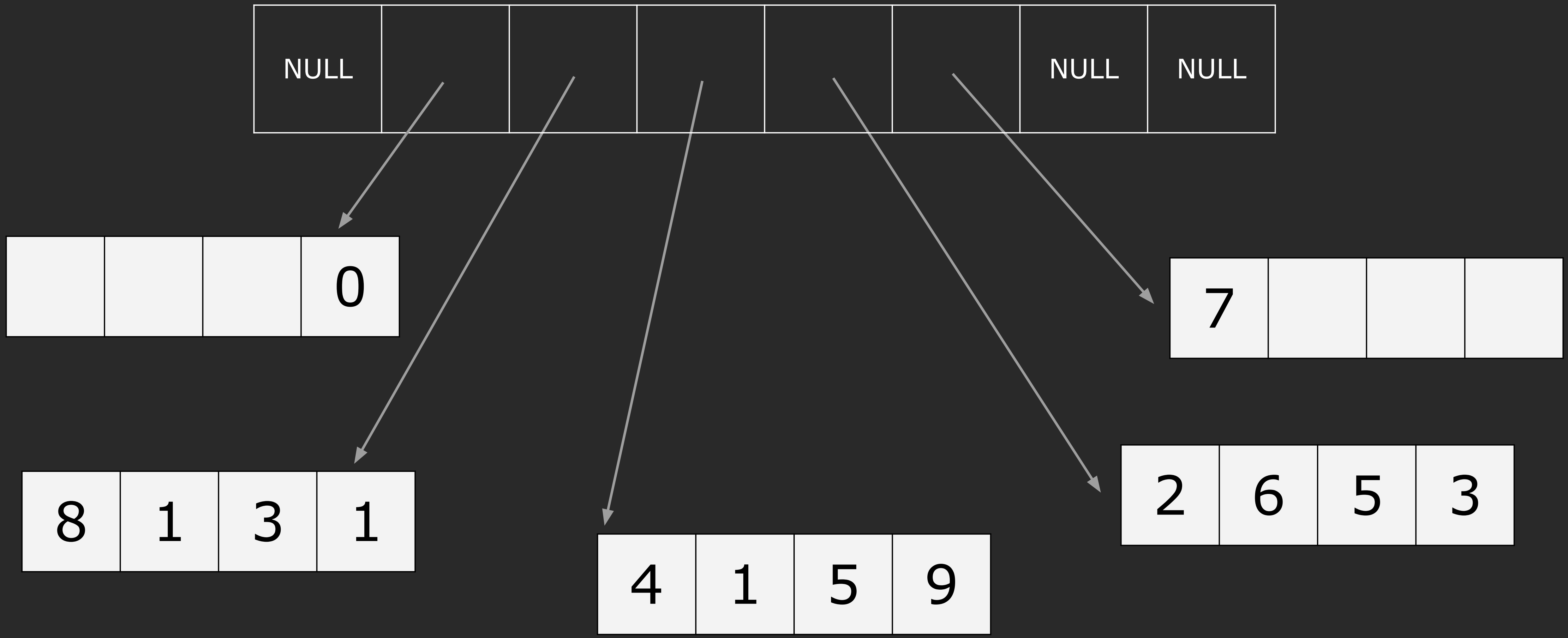


How does std::deque<T> work?



Supplemental Material

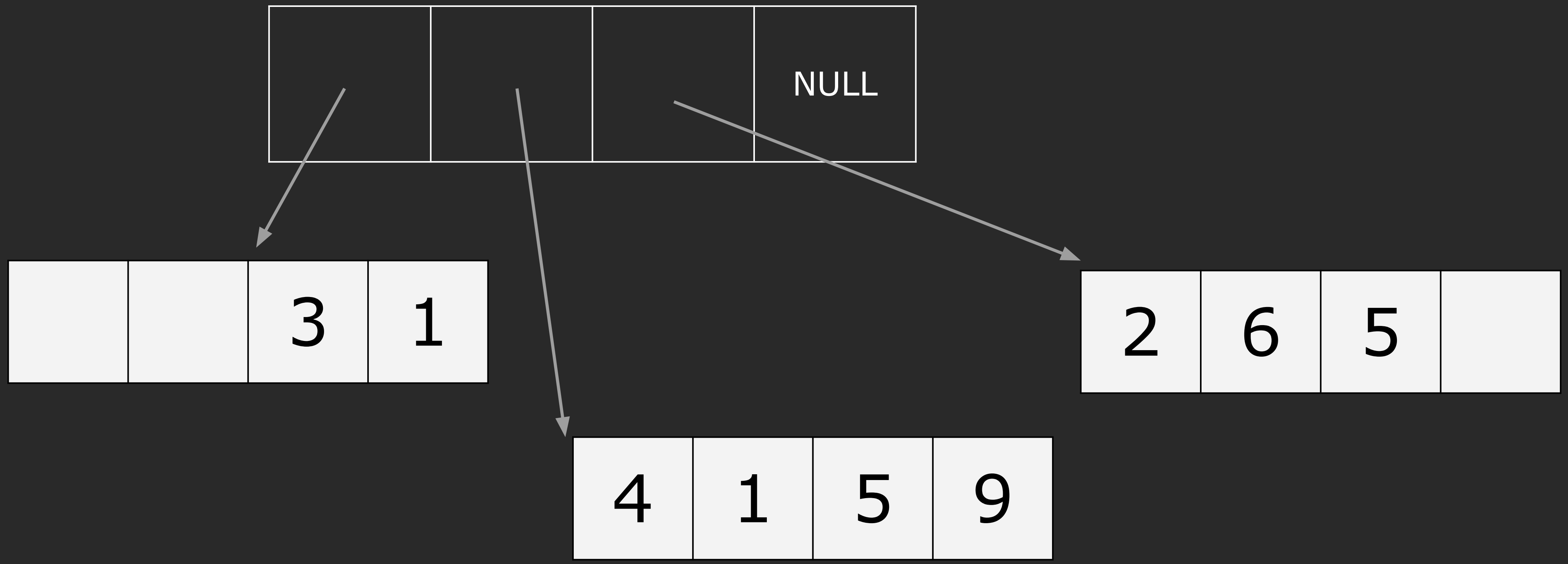
How does std::deque<T> work?



How does std::deque<T> work?

Difficult thought question that we won't answer in class:

How fast is inserting? Is it better of worse than a std::vector<T>?



std::list is kinda like std::stack + std::queue

std::list provides fast removal from the front and end

but you can't access any elements in the middle.

```
std::list<int> list{5, 6};    // {5, 6}
list.push_front(3);
list.pop_back(4);
```

Sidenote: usually a doubly-linked list. There's also a forward_list that's a singly-linked list.

when to use which sequence container?

	std::vector	std::deque	std::list
Indexed Access	Super Fast	Fast	Impossible
Insert/remove front	Slow	Fast	Fast
Insert/remove back	Super Fast	Very Fast	Fast
Ins/rem elsewhere	Slow	Fast	Very Fast
Memory	Low	High	High
Splicing/Joining	Slow	Very Slow	Fast
Stability (Iterators, concurrency)	Poor	Very Poor	Good

Sidenote: color-wise vector might not look great, but remember that indexed access and inserting to the back are the most common uses of sequence containers.

Sidenote: don't take what I say for granted. Run the sample code to test it out yourself!

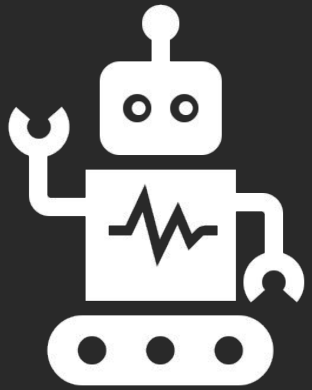
Example

std::vector vs. std::deque speed

Summary from the ISO Standard

"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

Answer 2 questions.

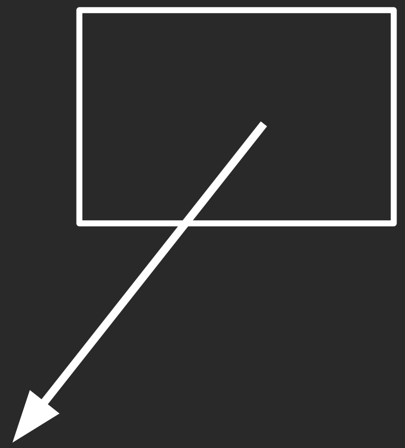
Summary of Sequence Containers

vector: use for most purposes

deque: frequent insert/remove at front

list: very rarely - if need splitting/joining

how is a vector implemented?



internally, a vector consists of an fixed-size array.
the array is automatically resized when necessary.

size = number of elements in the vector
capacity = amount of space saved for the vector

0	3	1	4	1	5	9	2	6		

best practices: if possible, reserve before insert

What's the best way to create a vector of the first 1,000,000 integers?

```
std::vector<int> vec;

for (size_t i = 0; i < 1000000; ++i) {
  vec.push_back(i);
}
```

Problem: internally the array is resized and copied many times.

best practices: if possible, reserve before insert

What's the best way to create a vector of the first 1,000,000 integers?

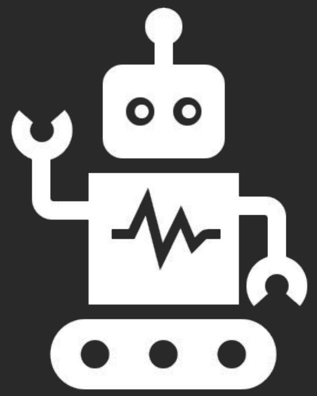
```
std::vector<int> vec;
vec.reserve(1000000);
for (size_t i = 0; i < 1000000; ++i) {
  vec.push_back(i);
}
```

Other best practices that we won't go over.

1. Consider using shrink_to_fit if you don't need the memory.
2. Call empty(), rather than check if size() == 0.
3. Don't use vector<bool> ("noble failed experiment")
4. A ton of other stuff after we talk about iterators!

If curious, ask us after class!

recommended reading: *Effective STL*



Example

CS 106B vector examples, using the STL vector/deque/list
Some performance analysis

Summary of Supplemental Material

It's easy to write inefficient code.

Know about the common pitfalls - prevent as much resizing as much as possible.

Container Adaptors

1. What is a container adaptor?
2. std::stack and std::queue

What is a wrapper (in general)?

A wrapper for an object changes how external users can interact with that object.

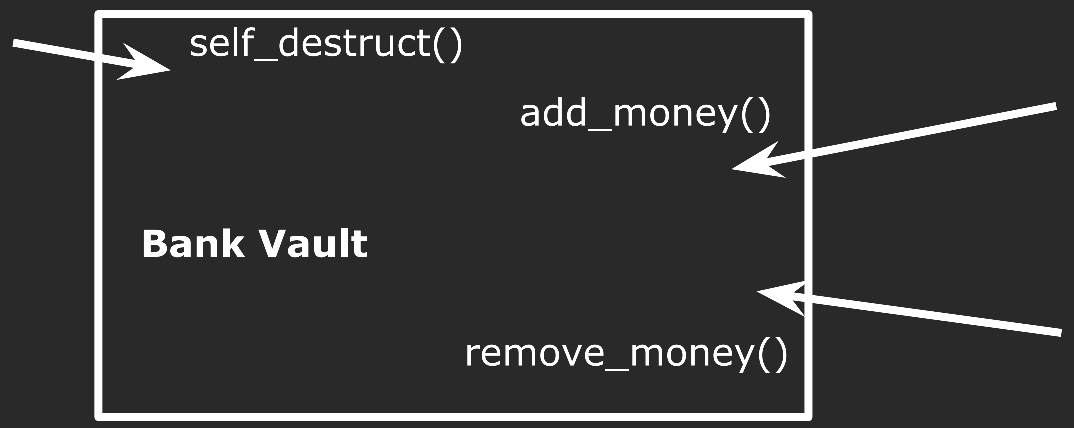
What is a wrapper (in general)?

Here is a bank vault.

Bank Vault

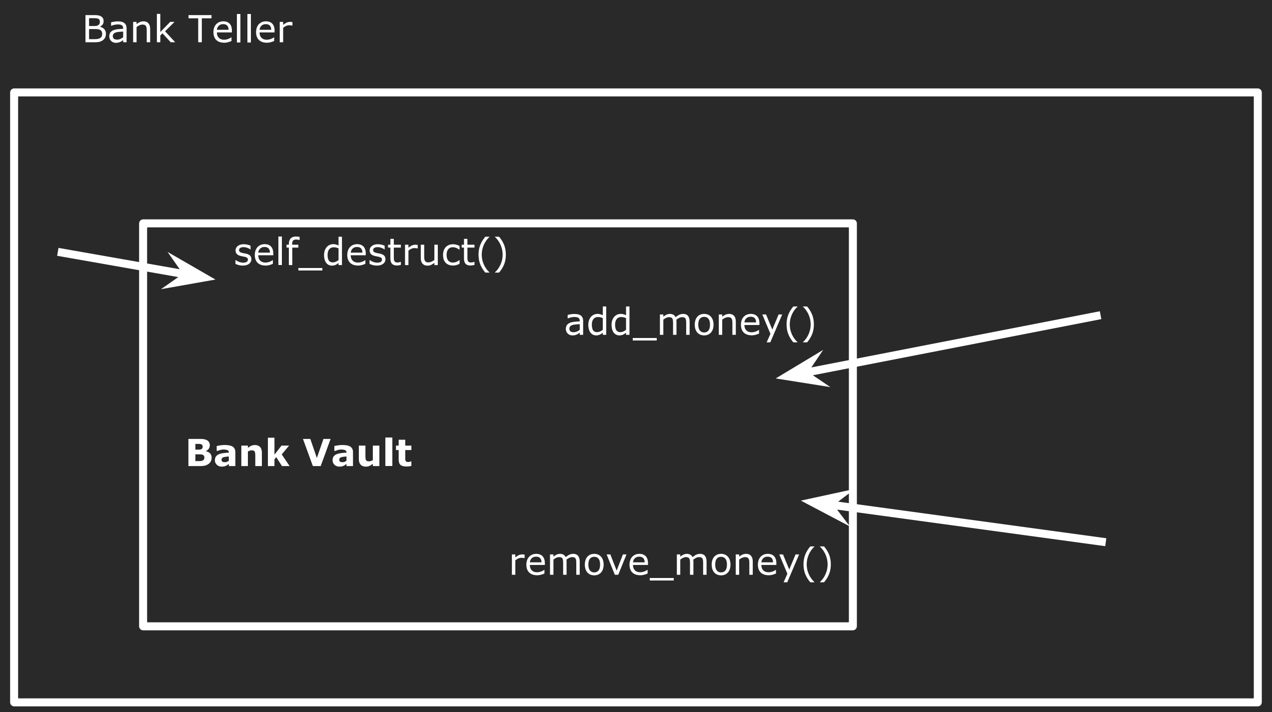
What is a wrapper (in general)?

There are many ways you can interact with a bank vault. It would be bad if people outside the bank could interact in these manners freely.



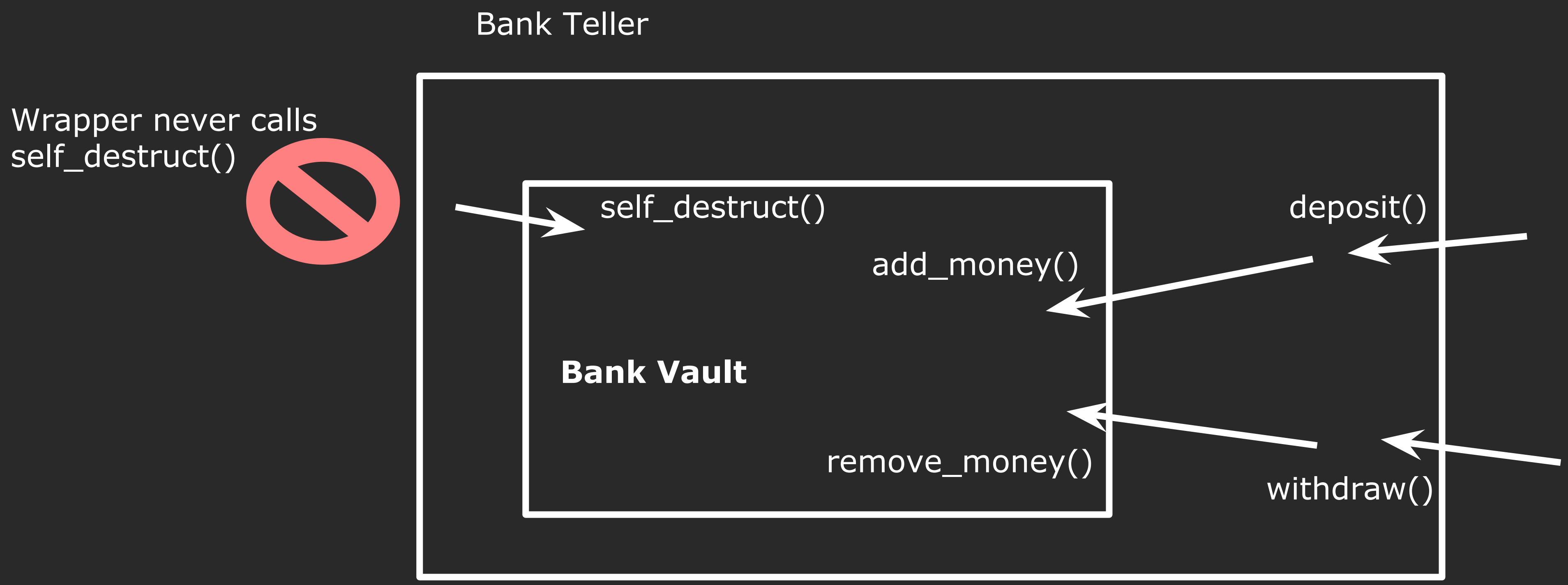
What is a wrapper (in general)?

The bank teller limits your access to the bank value.



What is a wrapper (in general)?

The bank teller is in charge of forwarding your request to the actual bank vault itself.



How do you design a stack?

Container adaptors provide a different interface for sequence containers. You can choose what the underlying container is!

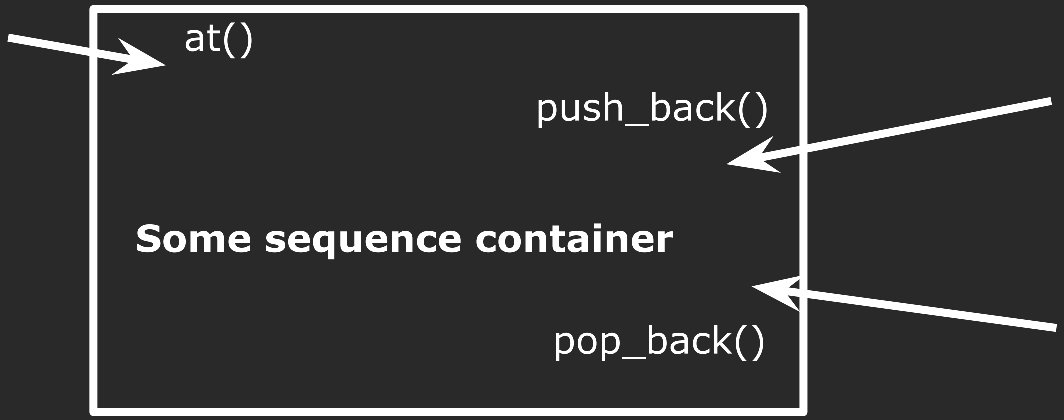
How do you design a stack?

Container adaptors provide a different interface for sequence containers. You can choose what the underlying container is!

Some sequence container

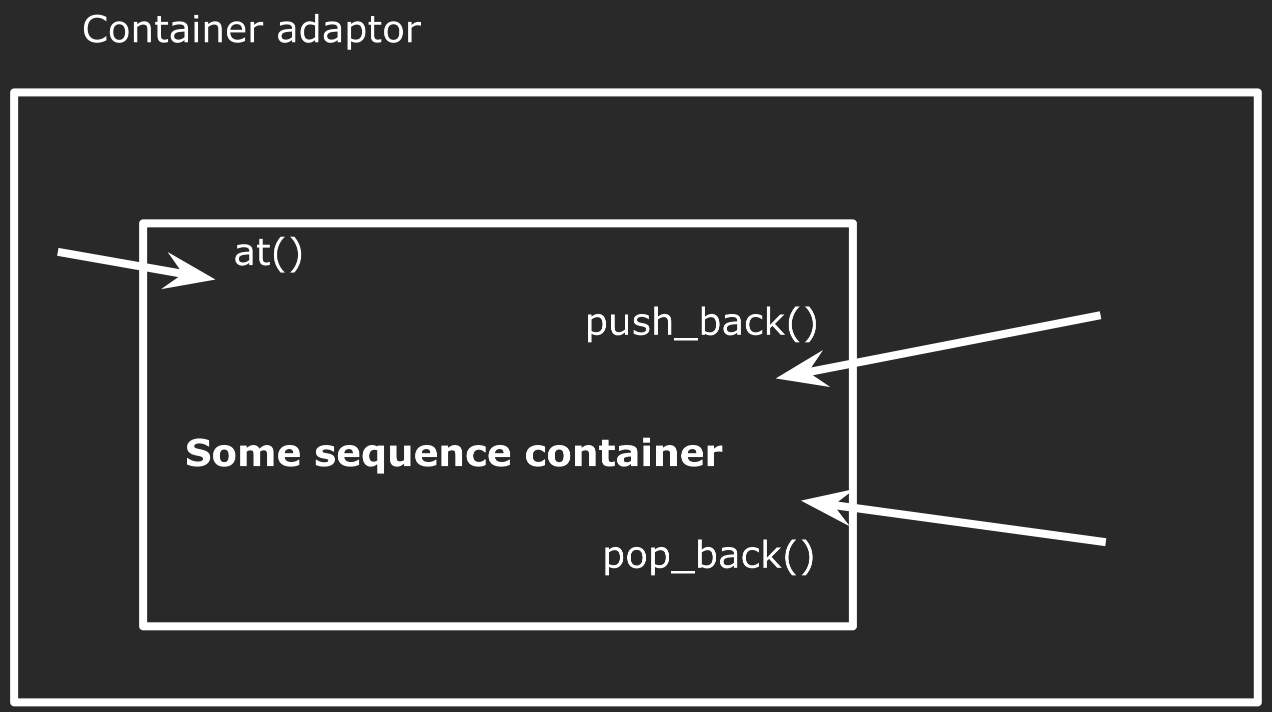
How do you design a stack?

Container adaptors provide a different interface for sequence containers. You can choose what the underlying container is!



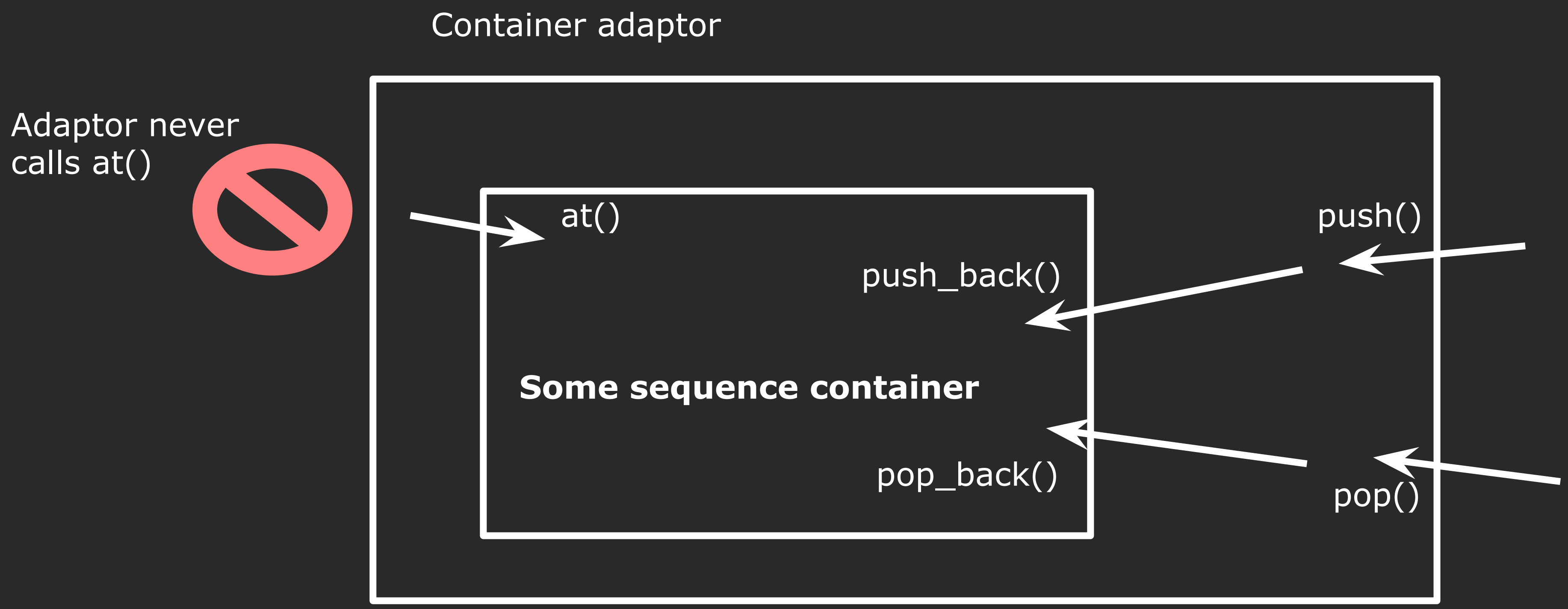
How do you design a stack?

Container adaptors provide a different interface for sequence containers. You can choose what the underlying container is!



How do you design a stack?

Container adaptors provide a different interface for sequence containers. You can choose what the underlying container is!



Quiz Question

Which data structure should we use to implement a stack? How about a queue?

A. std::vector<T>

B. std::deque<T>

C. Both are equally good

D. Both are equally bad

Answer on Poll:

Which data structure should we use to implement a stack? How about a queue?

A. std::vector<T>

B. std::deque<T>

C. Both are equally good

D. Both are equally bad

Answer on chat:

Why?

Concrete Example

std::stack

Defined in header <stack>

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

Why deque as opposed to vector or list?

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a LIFO (last-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 stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

std::queue

Defined in header <queue>

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

No surprise

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.

Concrete Example

std::stack

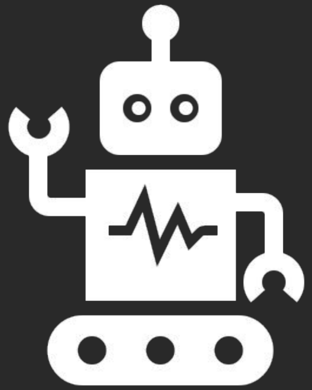
Defined in header <stack>

```
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 LIFO (last-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 stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

```
std::stack<int> stack_d;                   // Container = deque
std::stack<int, std::vector<int>> stack_v; // Container = vector
std::stack<int, std::list<int>> stack_l;   // Container = list
```



Questions

Answer 2 questions.

You'll be using std::priority_queue for A1!

std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user-provided `Compare` can be supplied to change the ordering, e.g. using `std::greater<T>` would cause the smallest element to appear as the `top()`.

Working with a `priority_queue` is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

CS 106B A5 is basically to write this container adaptor.

Associative Containers

1. std::set functions
2. std::map functions and auto-insertion
3. type requirements

Stanford vs. STL set: a summary

What you want to do	**Stanford Set<int>**	**std::set<int>**
Create an empty set	Set<int> s;	set<int> s;
Add k to the set	s.add(k);	s.insert(k);
Remove k from the set	s.remove(k);	s.erase(k);
Check if k is in the set (* C++20)	if (s.contains(k)) ...	if (s.count(k)) ... if (s.contains(k)) (*)
Check if the set is empty	if (s.isEmpty()) ...	if (s.empty()) ...

Answer on chat: what is different?

Stanford vs. STL set: a summary

What you want to do	Stanford Set<int>	std::set<int>
Create an empty set	Set<int> s;	set<int> s;
Add k to the set	s.add(k);	s.insert(k);
Remove k from the set	s.remove(k);	s.erase(k);
Check if k is in the set (* C++20)	if (s.contains(k)) ...	if (s.count(k)) ... if (s.contains(k)) (*)
Check if the set is empty	if (s.isEmpty()) ...	if (s.empty()) ...

There are functions for size, ==, !=, clear, etc.

STL does not have member functions for subset, difference, union, intersection, etc,

but there are STL algorithms!

Stanford vs. STL map: a summary

What you want to do	**Stanford Map<int, char>**	**std::map<int, char>**
Create an empty map	`Map<int, char> m;`	`map<int, char> m;`
Add key k with value v into the map	`m.put(k, v);`	`m.insert({k, v});`
Remove key k from the map	`m.remove(k);`	`m.erase(k);`
Check if k is in the map (* C++20)	`if (m.containsKey(k)) ...`	`if (m.count(k)) ...` `if (m.contains(k))` (*)
Check if the map is empty	`if (m.isEmpty()) ...`	`if (m.empty()) ...`
Retrieve or overwrite value associated with key k (error if does not exist)	Impossible (put does auto-insert)	`char c = m.at(k);` `m.at(k) = v;`
Retrieve or overwrite value associated with key k (auto-insert if DNE)	`char c = m[k];` `m[k] = v;`	`char c = m[k];` `m[k] = v;`

Stanford vs. STL map: a summary

What you want to do	**Stanford Map<int, char>**	**std::map<int, char>**
Create an empty map	`Map<int, char> m;`	`map<int, char> m;`
Add key k with value v into the map	`m.put(k, v);`	`m.insert({k, v});`
Remove key k from the map	`m.remove(k);`	`m.erase(k);`
Check if k is in the map (* C++20)	`if (m.containsKey(k)) ...`	`if (m.count(k)) ...` `if (m.contains(k))` (*)
Check if the map is empty	`if (m.isEmpty()) ...`	`if (m.empty()) ...`
Retrieve or overwrite value associated with key k (error if does not exist)	Impossible (put does auto-insert)	`char c = m.at(k);` `m.at(k) = v;`
Retrieve or overwrite value associated with key k (auto-insert if DNE)	`char c = m[k];` `m[k] = v;`	`char c = m[k];` `m[k] = v;`

STL maps stores std::pairs.

The underlying type stored in a

`std::map<K, V>`

is a

`std::pair<const K, V>`.

sidenote: why is the key const? You should think about this, and you'll need a good answer when A2 rolls around.

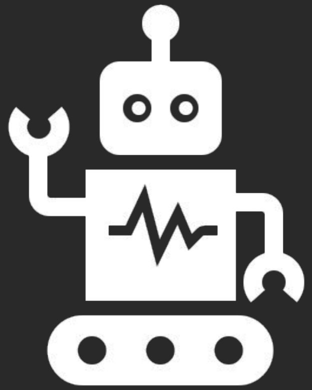
Stanford/STL sets + maps require comparison operator

By default, the type (for sets) or key-type (for maps) must have a comparison (<) operator defined for them.

(There is an alternative that we will cover in ~2 lectures)

```
std::set<std::pair<int, int>> set1;  // OKAY - comparable
std::set<std::ifstream> set2;        // ERROR - not comparable

std::map<std::set<int>, int> map1;   // OKAY - comparable
std::map<std::function, int> map2;   // ERROR - not comparable
```



Questions

Answer 4 questions.

Iterating over the elements of a STL set/map.

- Exactly the same as in CS 106B - no modifying the container in the loop!
- The elements are ordered based on the operator< for element/key.
- Because maps store pairs, each element m is an std::pair that you can use structured binding on.

```
for (const auto& element : s) {
  // do stuff with key
}

for (const auto& [key, value] : m) {
  // do stuff with key, value
}
```

unordered_map and unordered_set

Each STL set/map comes with an unordered sibling. Their usage is the same, with two differences:

- Instead of a comparison operator, the element (set) or key (map) must have a hash function defined for it (you'll learn more in A2).
- The unordered_map/unordered_set is generally faster than map/set.

Don't worry too much about these just yet - you'll implement unordered_map in assignment 2!

multimap, multiset (+ unordered siblings)

Each STL set/map (+ unordered_set/map) comes with an multi- cousin. Their usage is the same, with two differences:

- You can have multiple of the same element (set) or key (map).
- insert, erase, and retrieving behave differently (since they may potentially have to retrieve multiple elements/values).

I've actually never used these before. Let us know if you actually ever use one. Otherwise, let's not spend too much time on this.

Summary of Associative Containers

set: membership

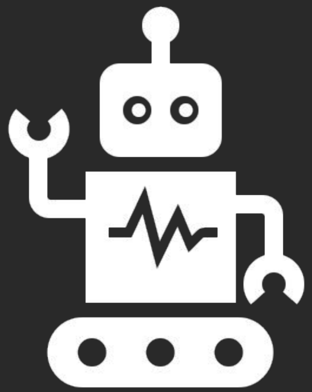
map: key/value pairs

set/map require comparison operator

map stores std::pairs

auto-insertion = useful but be careful

unordered/multi stuff - know they exist



Deep Dive into Documentation

map and set

We've run into a few problems with STL containers

Concrete things we haven't been able to do yet...

- how to insert an element at a certain position in a vector?
- how do we create sublists of a vector? how about for a set?
- how do we iterate over all the elements in an associative container?

Ideas we want to move toward...

- can I write a function that works for any container?
- can I operate over a container while completing ignoring the container?

Why does CS 106B not teach the Standard Libraries?

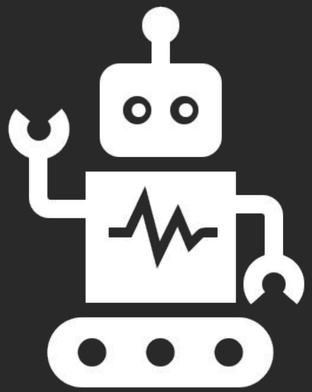
(it's not a fluke. there's a genius pedagogical reason behind it)

Today's Lecture in Bullet Points

- std::vector methods are very similar to Stanford Vector
- std::vector indexing has [ ] and at() functions. only at() does error-checking
- std::deque is like vector, inserting to front is fast, other operations slower
- std::stack and std::queue adapt a vector/deque to their unique interface
- std::set is like Stanford Set. Main difference: count() instead of contains()
- std::map kinda like Stanford Map, but internally stores pair<K, V>
- std::map's operator[] has auto-insertion, but at() does not
- set/map require a comparison operator (since they are internally sorted)
- you should practice using these containers yourself
- you should read the STL documentation for more details

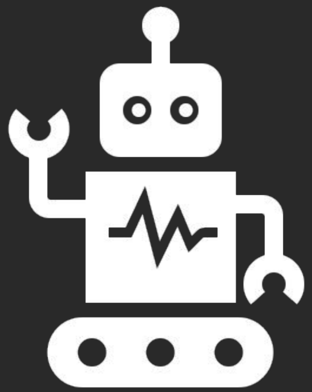
Next time

Iterators



Homework

Finish GraphViz
Start "Cracking the Coding Interview in C++"



After Class Optional Example

CS 106B map/set examples, using the STL map/set