Containers

CS 106L, Fall ‘21

How do we handle data in the STL?

Today's agenda

- What makes a container?
- The Standard Template Library (STL)
- Containers (Stanford vs STL)
  - Sequence
  - Associative
  - How do they actually work?

How do we store data in programs?

What do we need in order to do so?

Python and Stanford Library

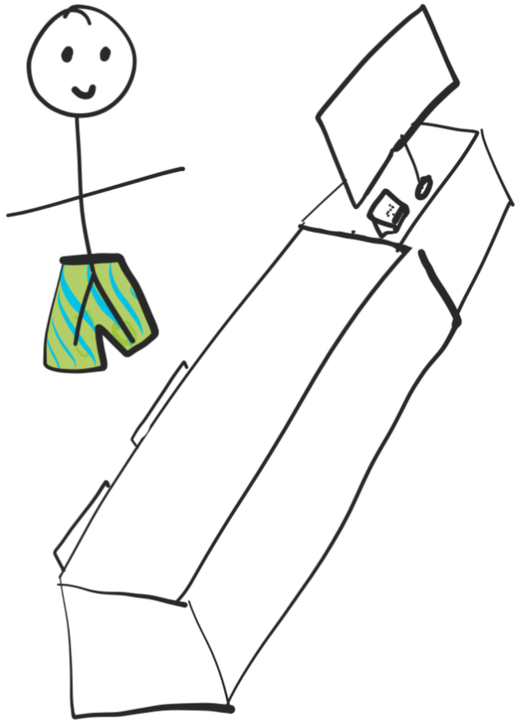
Classic examples of data structures:

```
lst = []
dictionary = {}
hash_table = set()
string = "adc"
```

```
Vector<int> lst;
Map<int, int> dictionary;
Set<int> hash_table;
std::string str = "adc";
```

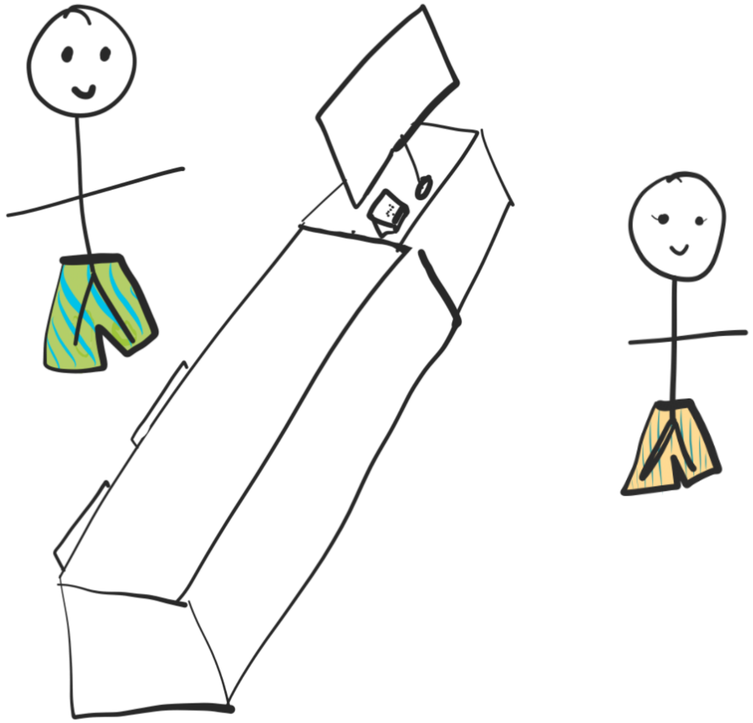
What makes a container?

- Let’s design one! We’ll be making a queue.



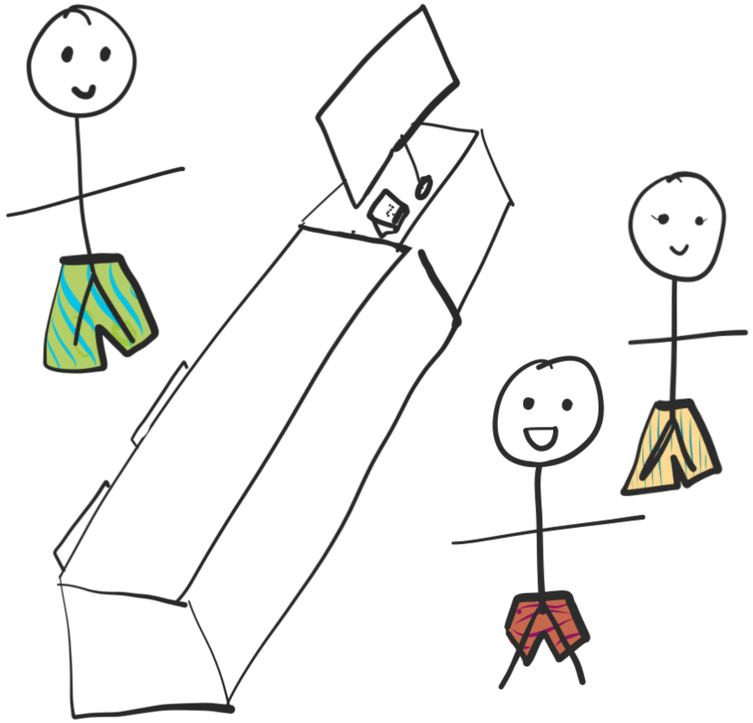
What makes a container?

- Let’s design one! We’ll be making a queue.



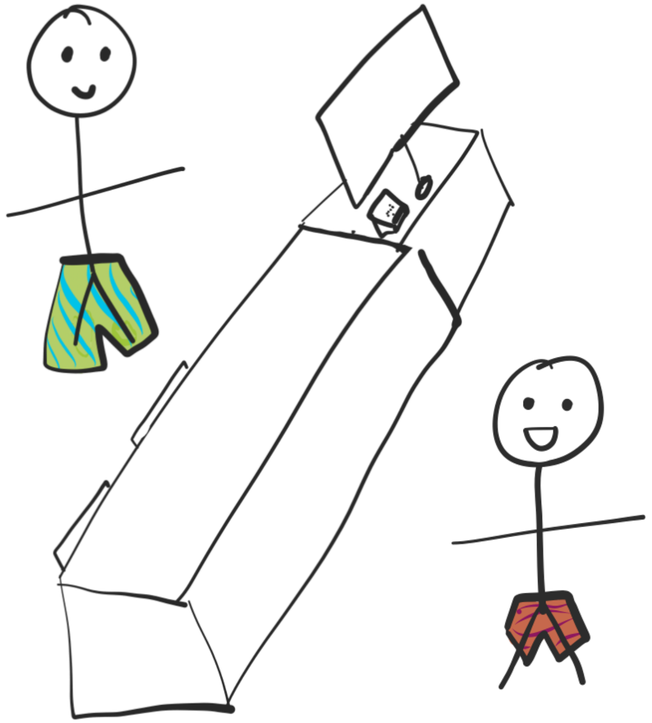
What makes a container?

- Let’s design one! We’ll be making a queue.



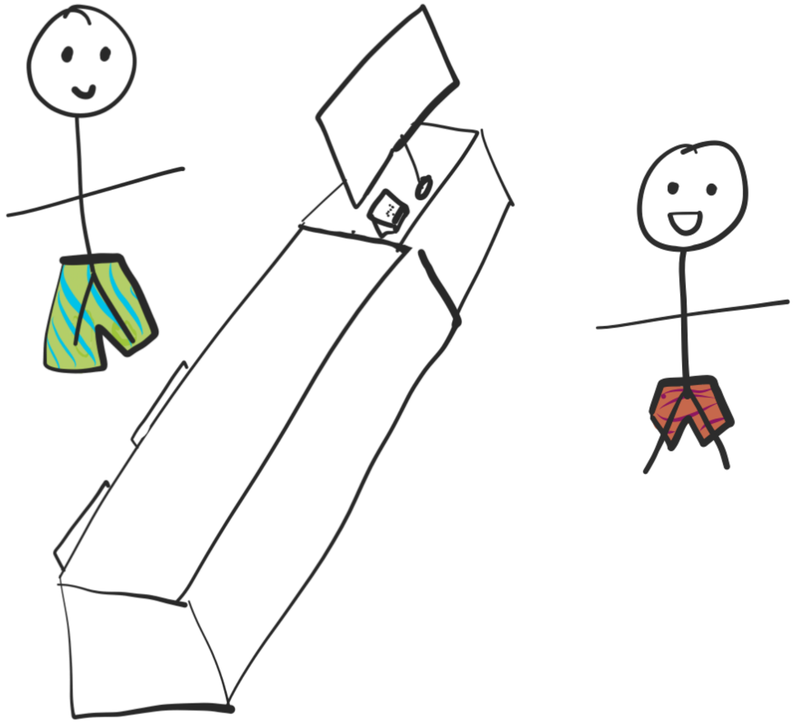
What makes a container?

- Let’s design one! We’ll be making a queue.



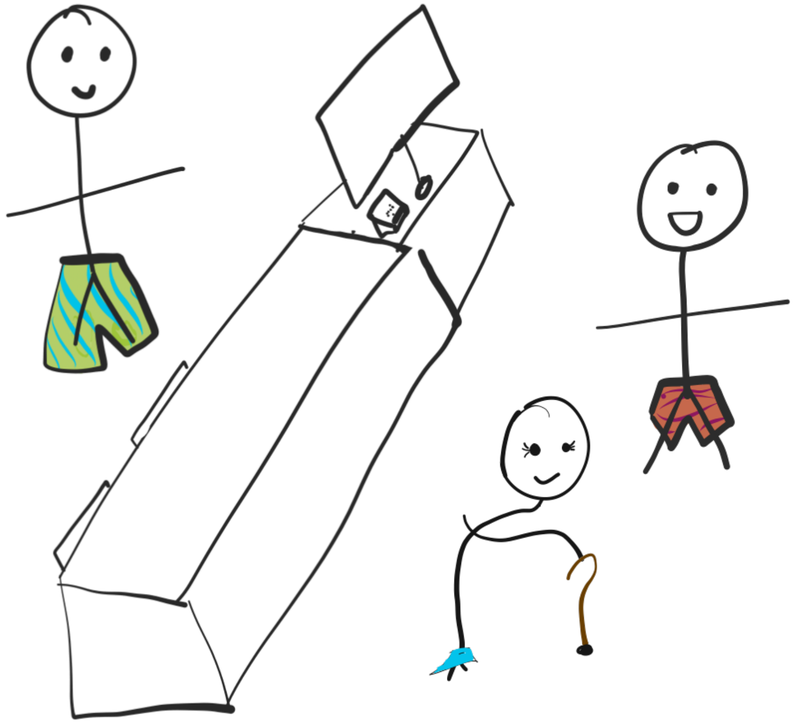
What makes a container?

- Let’s design one! We’ll be making a queue.



What makes a container?

- Let’s design one! We’ll be making a queue.



This seems like a pretty common problem, right?

- What if I have to make a queue for a movie ticket line?
  - for a car wash?
  - for a burst of API requests?
- We need a place to store common definitions for data structures.

Standard Template Library

The core of modern C++!

What's in the STL?

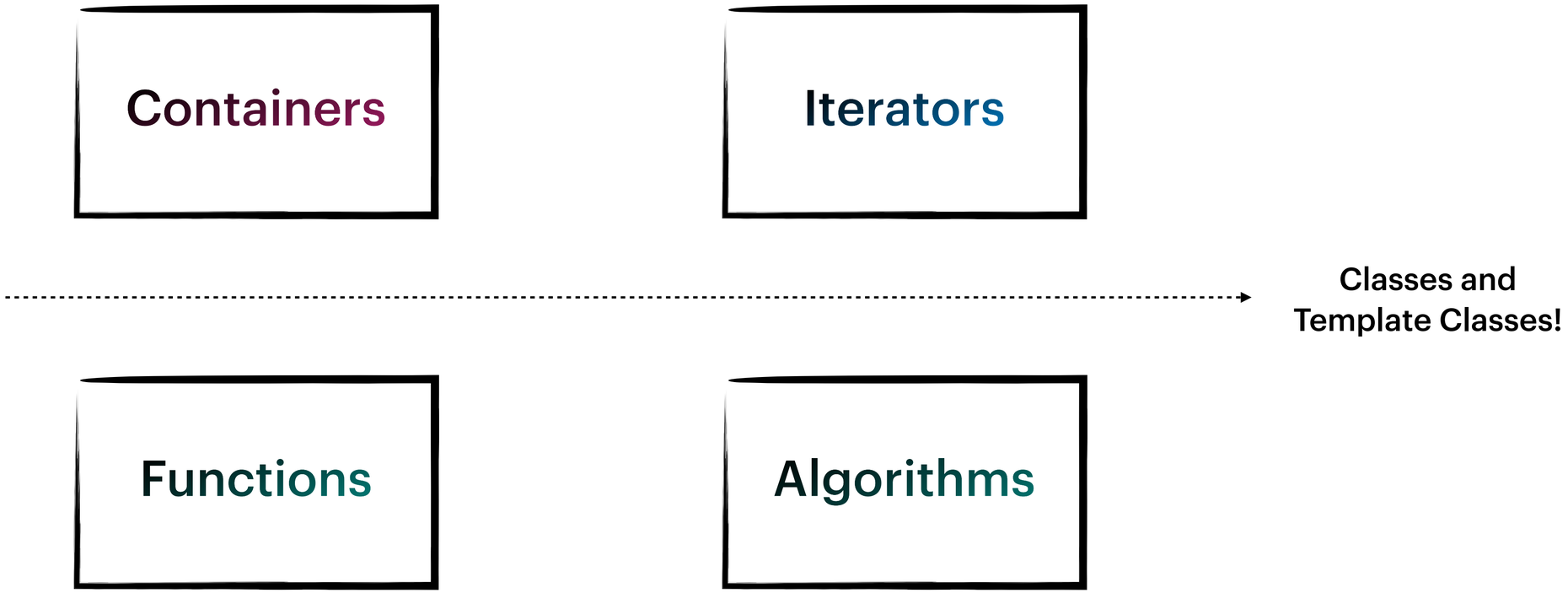
Containers

Iterators

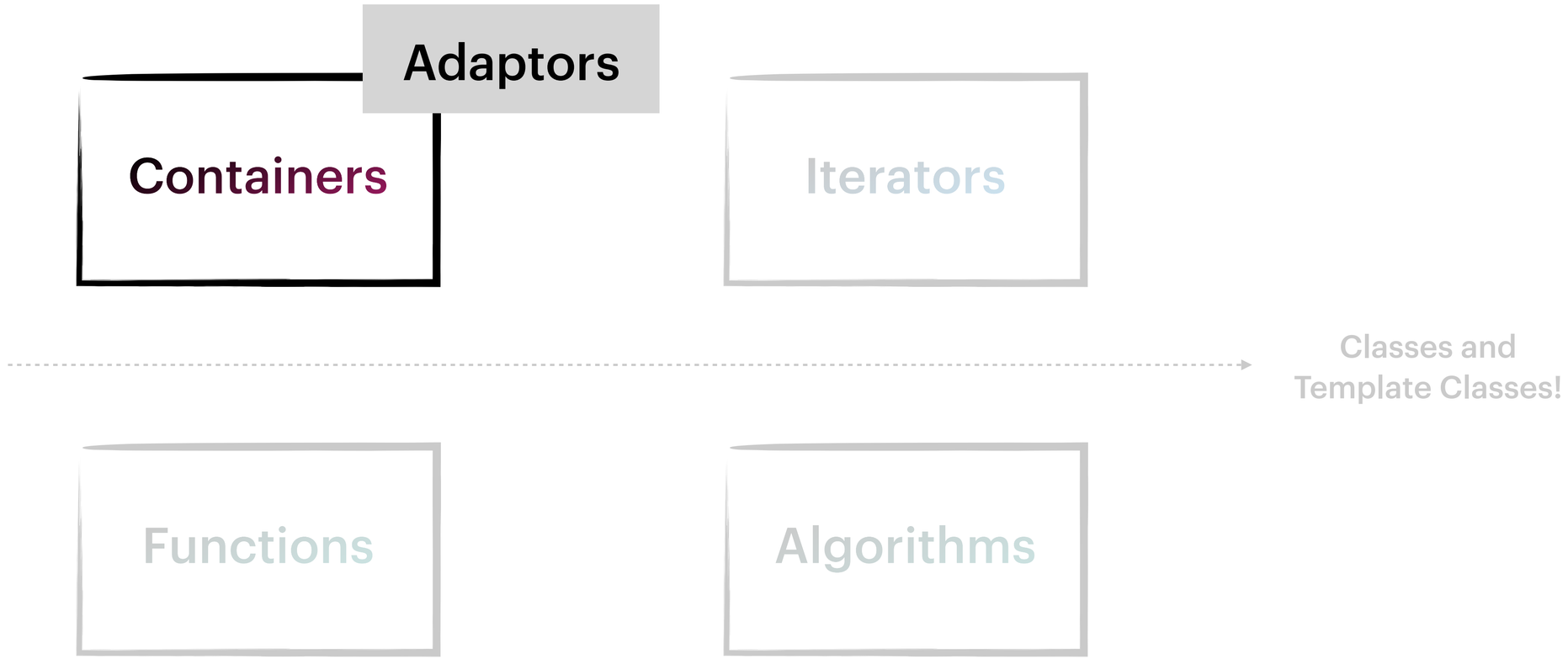
Functions

Algorithms

What's in the STL?



What's in the STL?

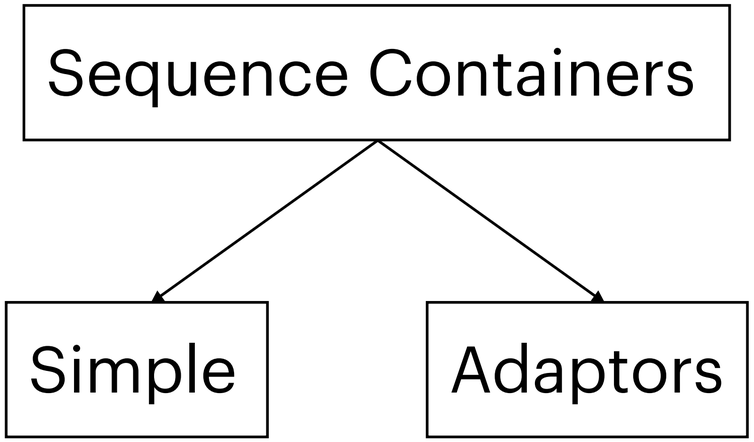


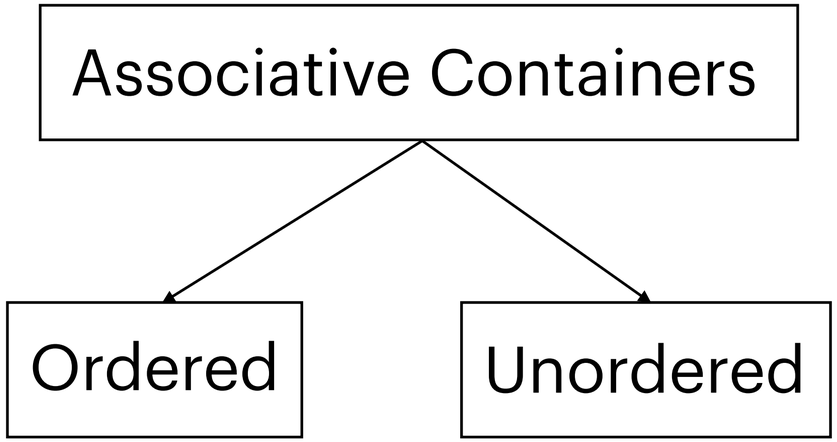
Types of containers

- All containers can hold almost all elements.

Types of containers

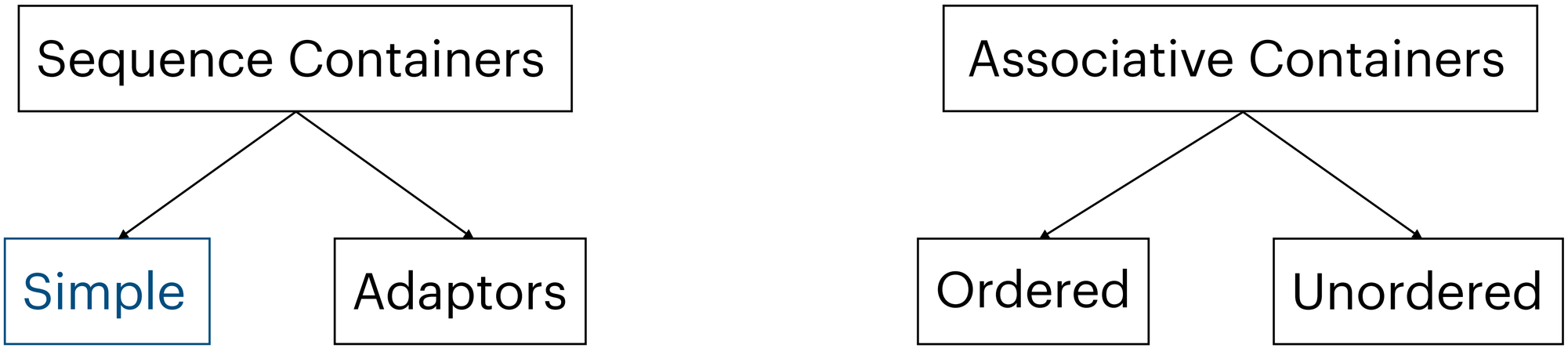
- All containers can hold almost all elements.





Types of containers

- All containers can hold almost all elements.



<> vector (adding + removing elements at end)

⇅ deque (adding + removing elements anywhere but end)

↓ list (adding + removing elements anywhere, but no random access)

() tuple (different data types, but immutable)

How do we use the STL? (and an aside on “::”)

```
#import <vector>

int main () {
  std::vector<int> vec;
  ...
}
```

Just two steps!

1. Import the relevant STL feature
2. Use it with “std::<STL feature name here>”

“::” -> Scope Resolution Operator

For heavily used items, we can use certain classes and datatypes in the std namespace, e.g. “using std::string;”

```
using namespace std;
```

Live Demo!

Let’s compare the Stanford’s Vector and STL’s vector:
(QT Creator Project)

Stanford "Vector" vs STL "vector"

What you want to do	Stanford `Vector<int>`	`std::vector<int>`
Create a new, empty vector	`Vector<int> vec;`	`std::vector<int> vec;`
Create a vector with **`n`** copies of 0	`Vector<int> vec(n);`	`std::vector<int> vec(n);`
Create a vector with **`n`** copies of a value **`k`**	`Vector<int> vec(n, k);`	`std::vector<int> vec(n, k);`
Add a value **`k`** to the end of a vector	`vec.add(k);`	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`	`vec.clear();`
Get the element at index **`i`**	`int k = vec[i];`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`	`vec.size();`
Loop through vector by index **`i`**	`for (int i = 0; i < vec.size(); ++i) ...`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index **`i`**	`vec[i] = k;`	`vec[i] = k;` (does **not** bounds check)

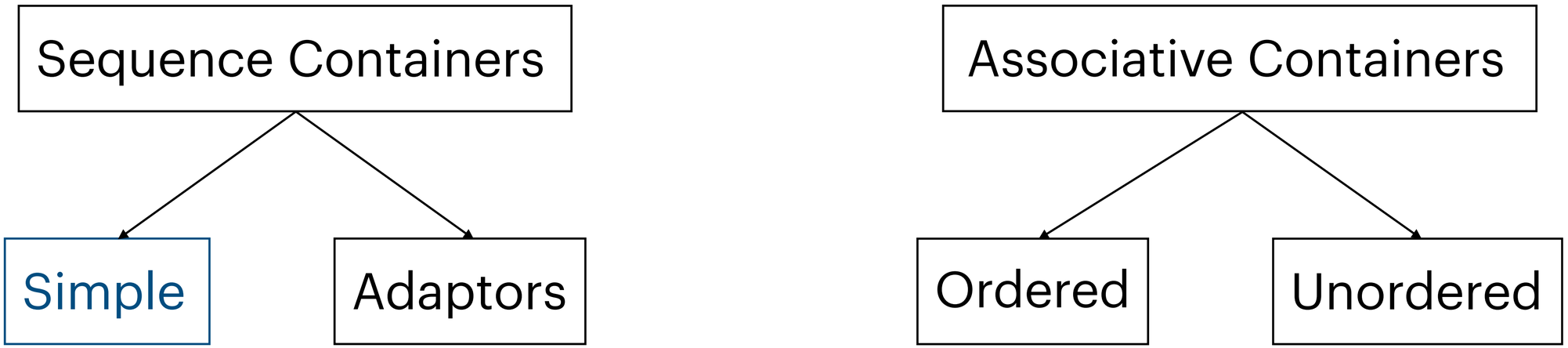
Stanford “Vector” vs STL “vector”

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

We need ***iterators*** to understand these. Next Lecture!

Types of containers

- In general, how do we pick between containers?



<> vector (adding + removing elements at end)

⇅ deque (adding + removing elements anywhere but end)

↓ list (adding + removing elements anywhere, but no random access)

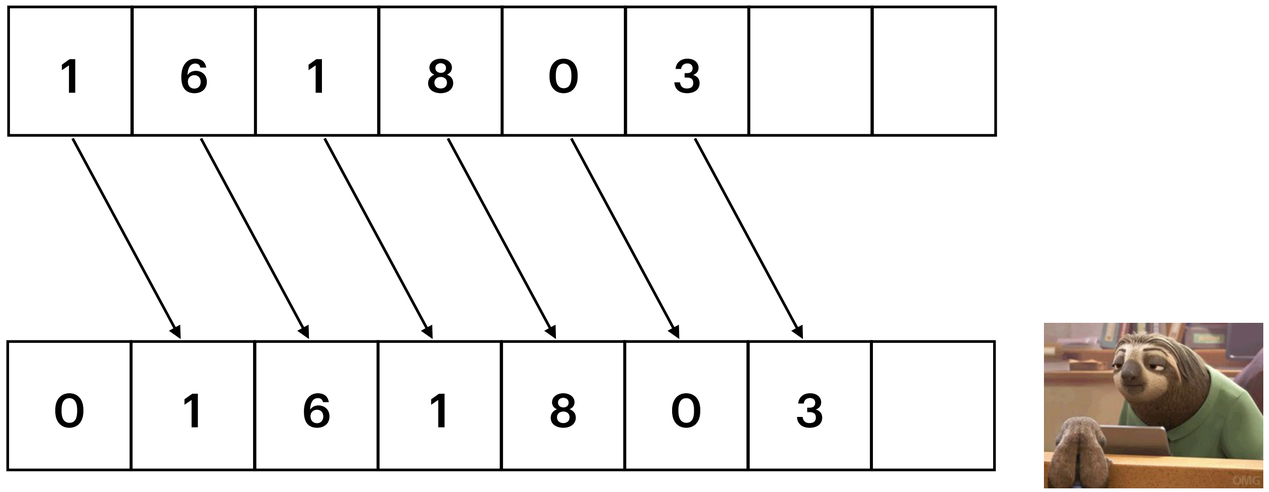
() tuple (different data types, but immutable)

Live Demo!

Why can’t we just use a vector for everything?
vector_time_trials.cpp

Why is there no std::vector::push_front()?

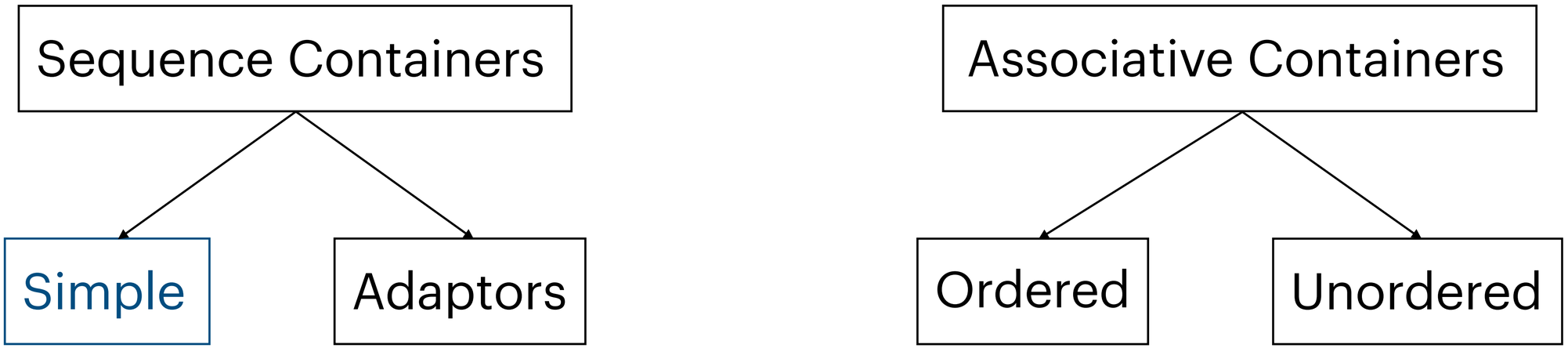
- Because it’s super slow!
- This is what is happening to the elements inside the vector:



(in general, if something is extremely inefficient, C++ does not expose a method for it!)

Types of containers

- In general, how do we pick between containers?



<> vector (adding + removing elements at end)

⇅ deque (adding + removing elements anywhere but end)

↓ list (adding + removing elements anywhere, but no random access)

() tuple (different data types, but immutable)

Live Demo!

Let’s compare speeds of several operations!
vector_list_deque.cpp

How is a vector actually implemented?

- Internally, a **`std::vector`** consists of a fixed-sized array
  - It automatically resizes for you! (Arrays will be discussed in CS 106B after week 5)

How is a vector actually implemented?

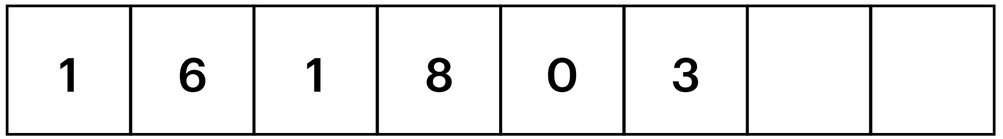
- Internally, a **`std::vector`** consists of a fixed-sized array
  - It automatically resizes for you! (Arrays will be discussed in CS 106B after week 5)

`size` = number of elements in a vector

`capacity` = amount of space saved for a vector

`size` = 6

`capacity` = 8

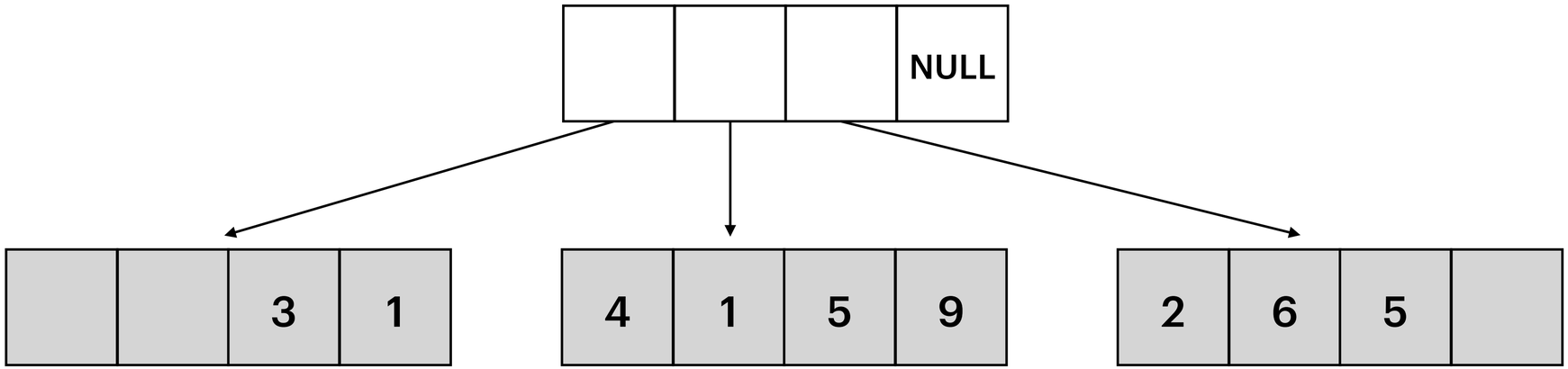


How is a deque actually implemented?

- There’s no single, common implementation of a deque, but a common one looks like this:

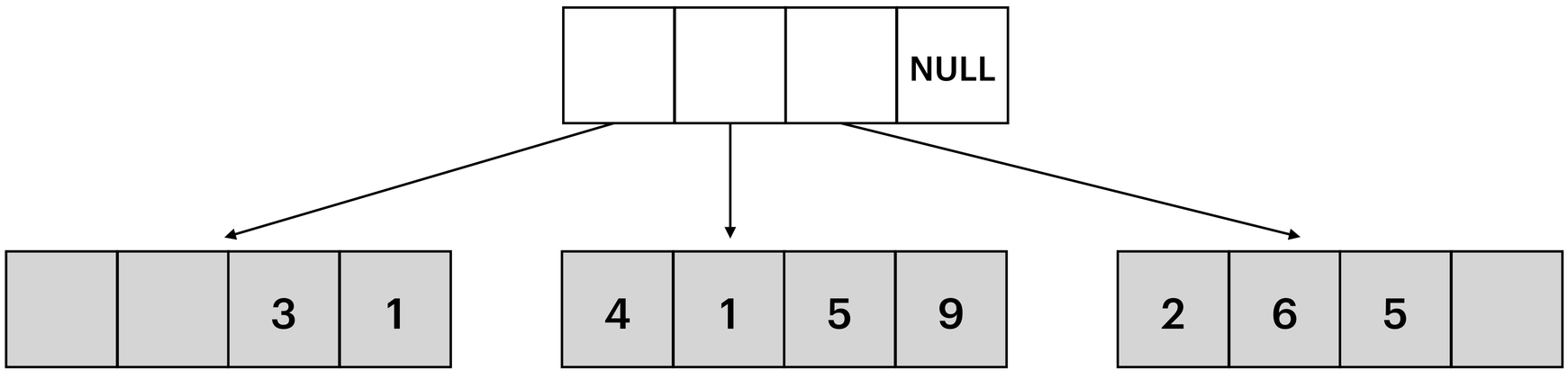
How is a deque actually implemented?

- There's no single, common implementation of a deque, but a common one looks like this:



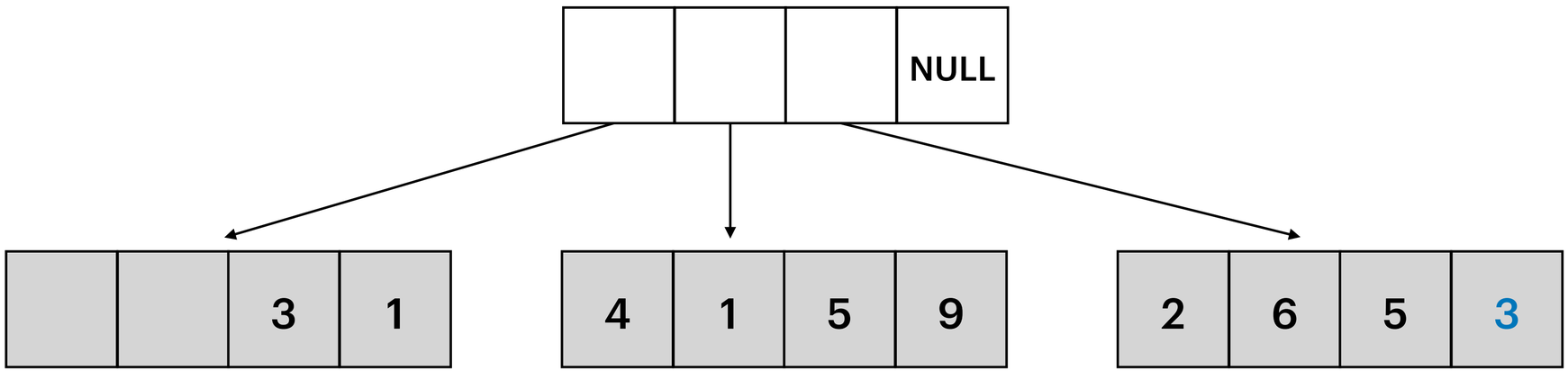
How is a deque actually implemented?

- So how do we **`push_back(3)`**?



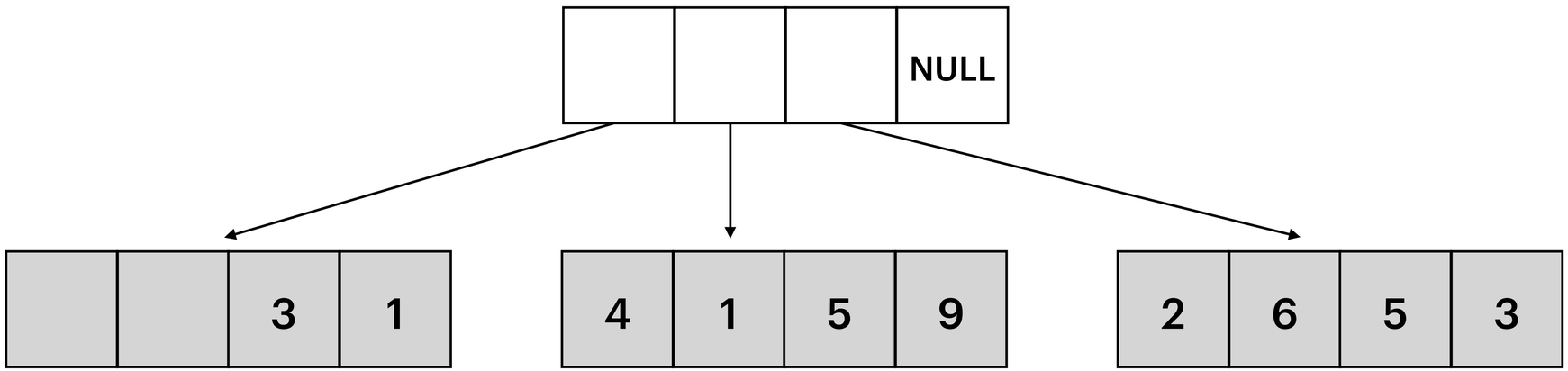
How is a deque actually implemented?

- So how do we **`push_back(3)`**?



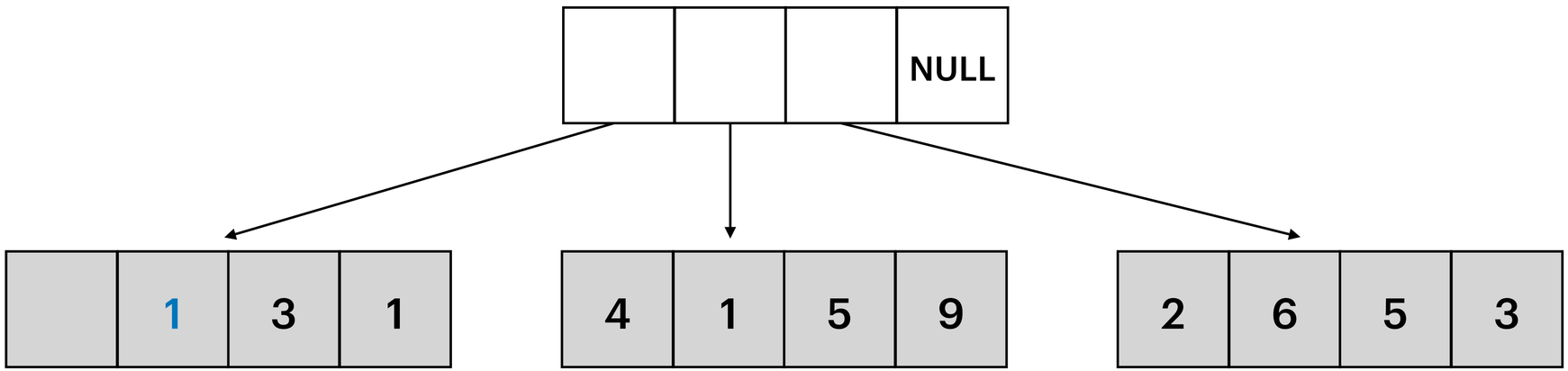
How is a deque actually implemented?

- So how do we **`push_front(1)`**?



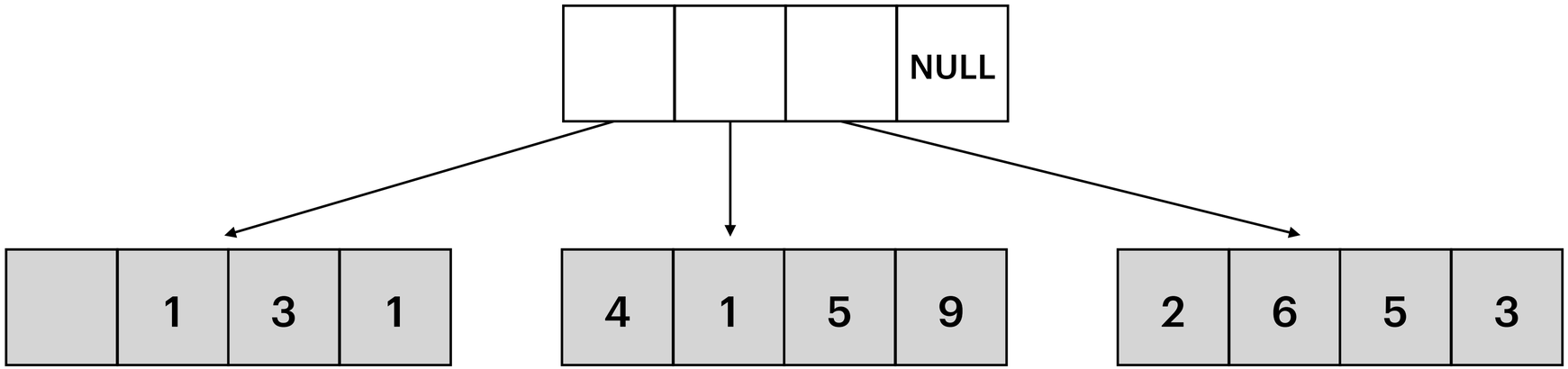
How is a deque actually implemented?

- So how do we **`push_front(1)`**?



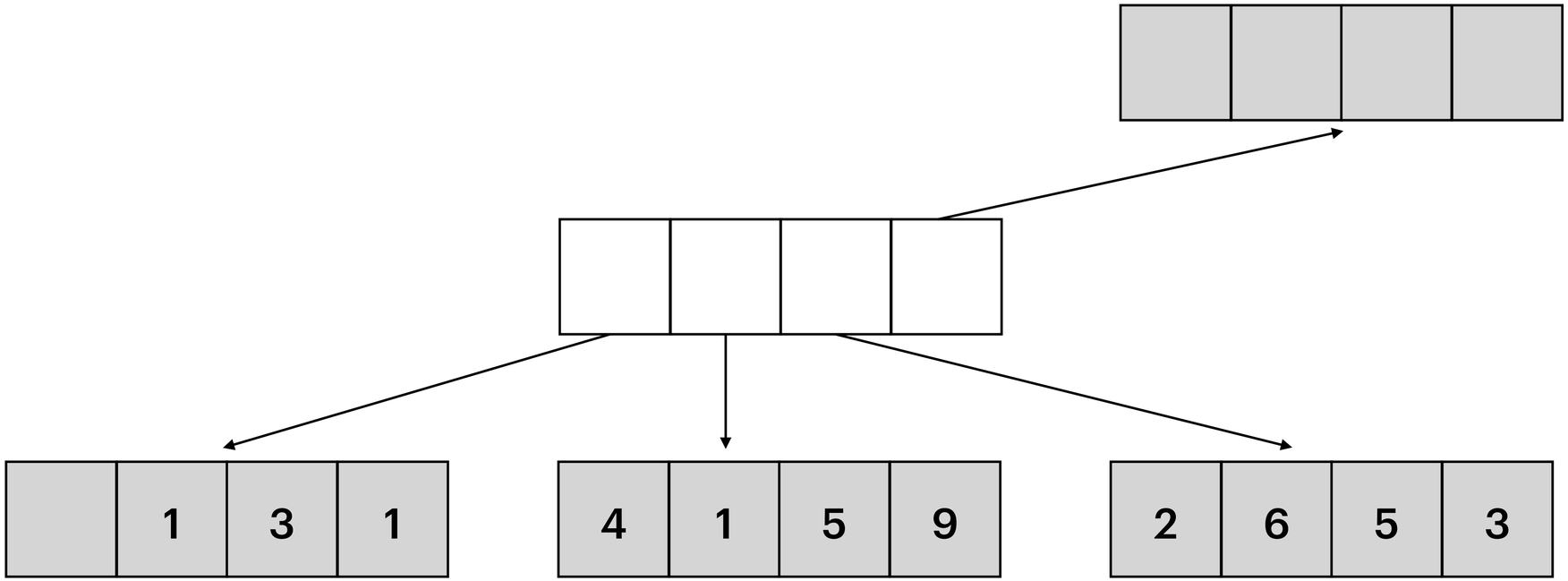
How is a deque actually implemented?

- Now, how do we **`push_back(7)`**? Haven't we run out of space in the last array?



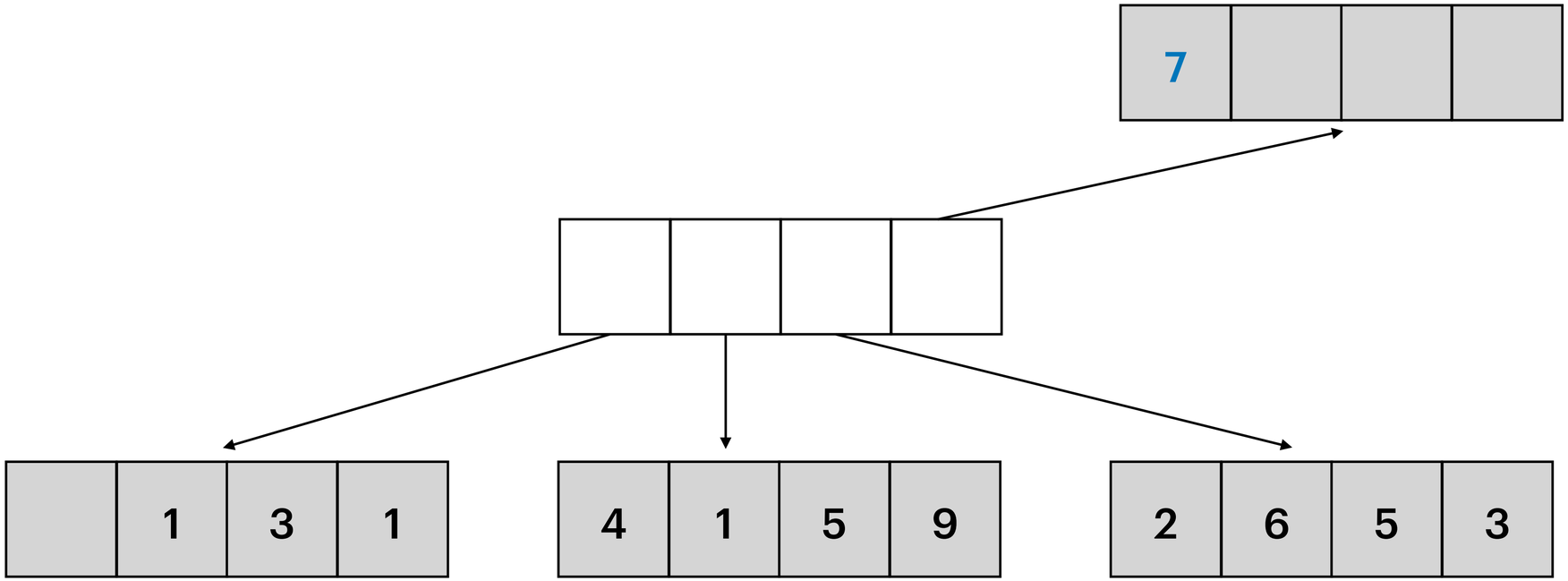
How is a deque actually implemented?

- Now, how do we **`push_back(7)`**? Haven't we run out of space in the last array?



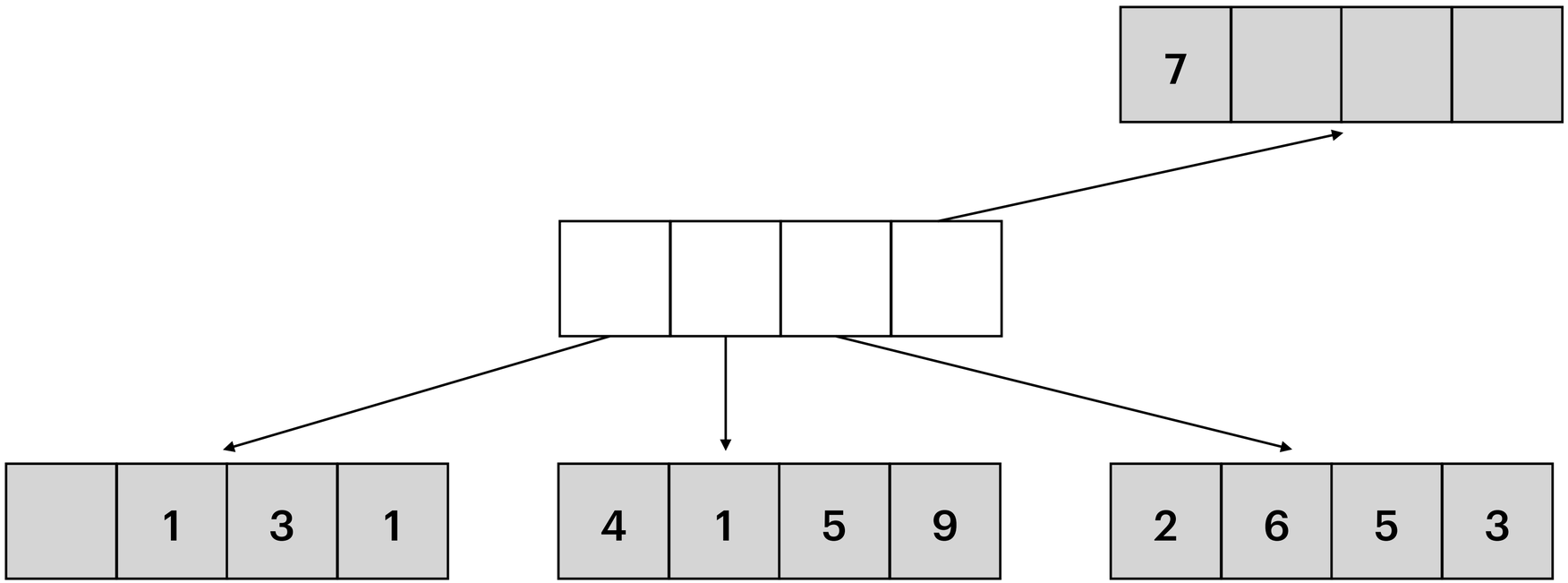
How is a deque actually implemented?

- Now, how do we **`push_back(7)`**? Haven't we run out of space in the last array?



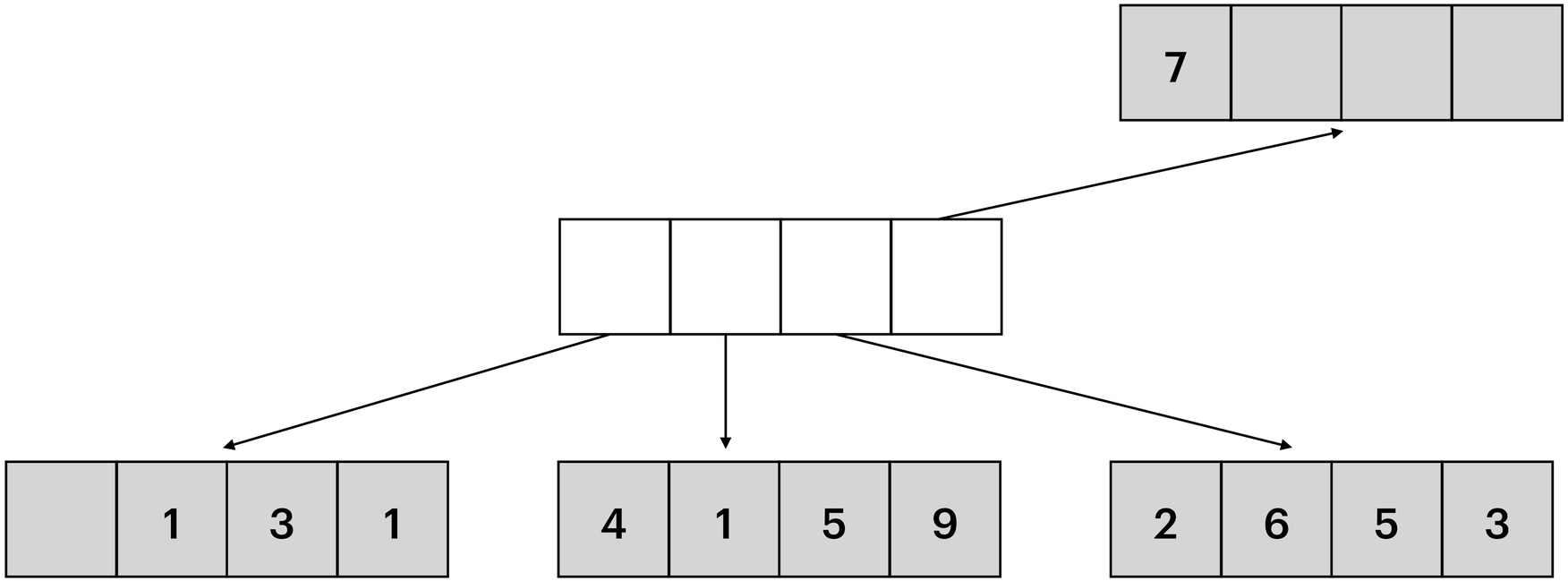
How is a deque actually implemented?

- Lastly, how can we **`push_front(8)`** and then **`push_front(0)`**?



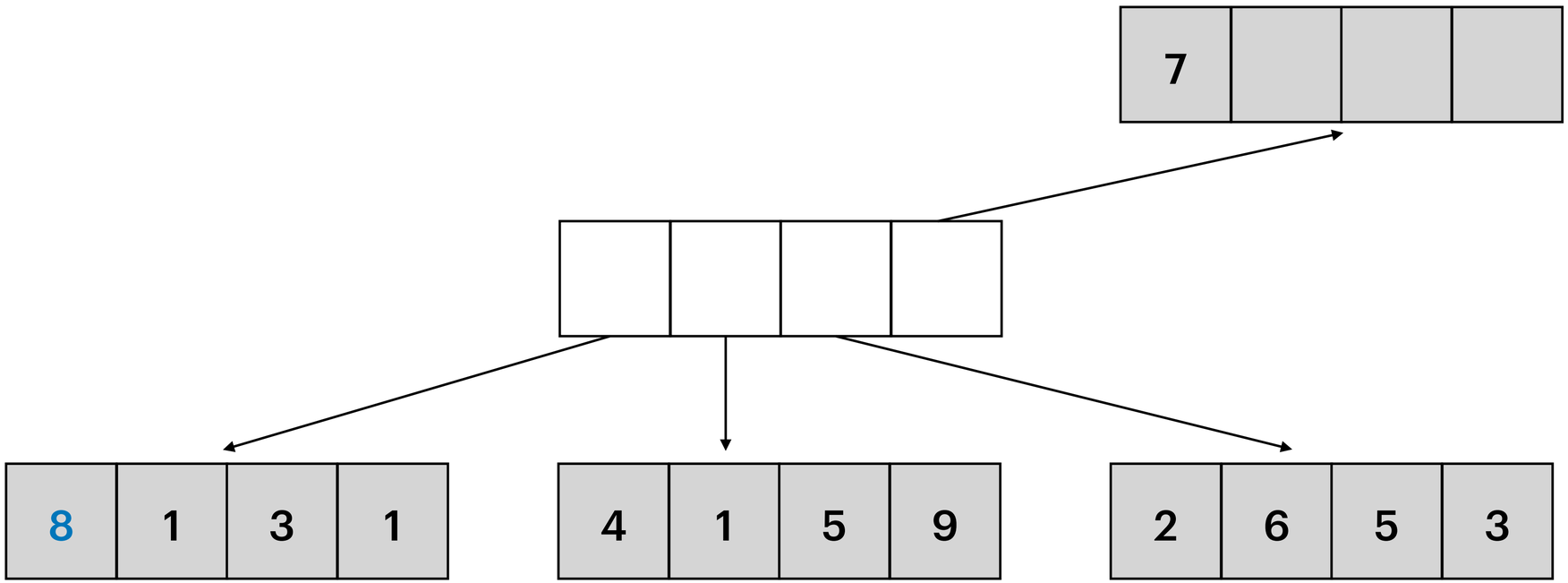
How is a deque actually implemented?

- Lastly, how can we **`push_front(8)`** and then **`push_front(0)`**?

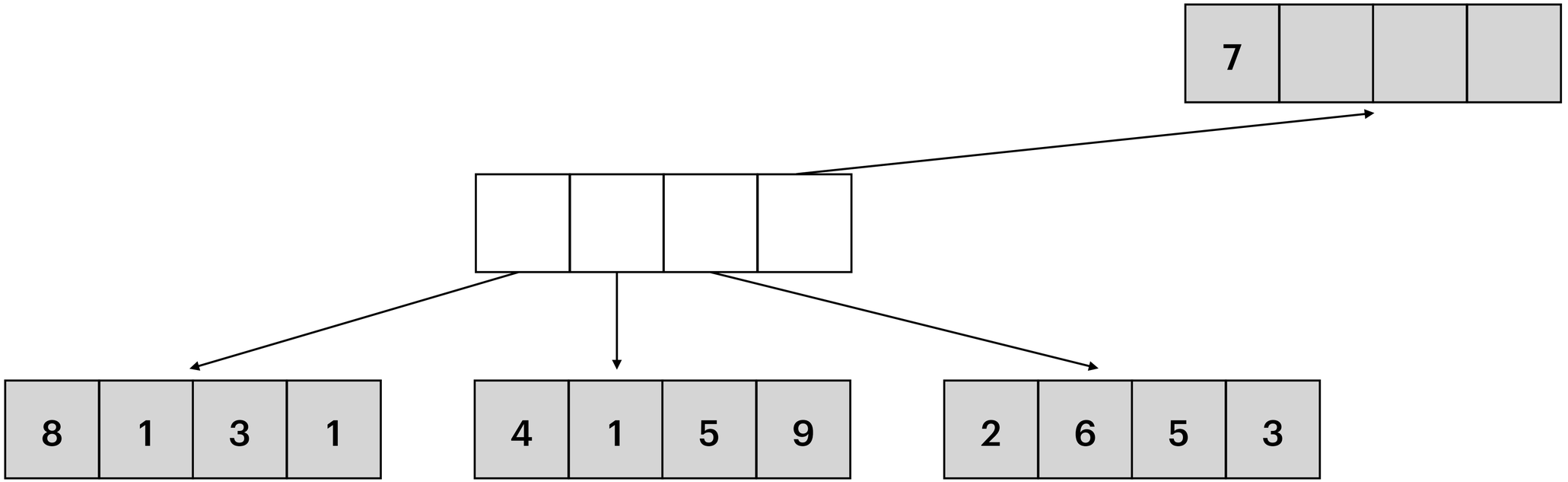


How is a deque actually implemented?

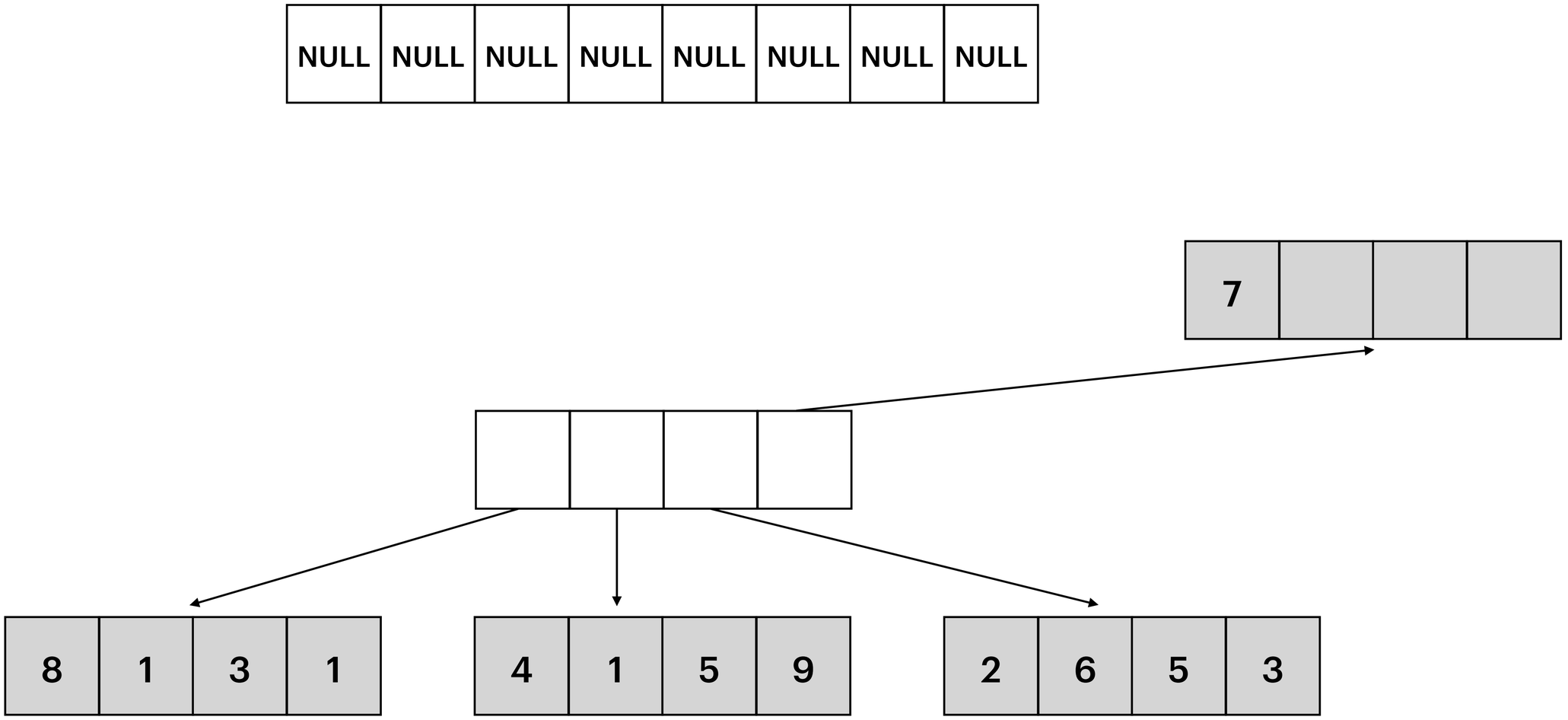
- Lastly, how can we **`push_front(8)`** and then **`push_front(0)`**?



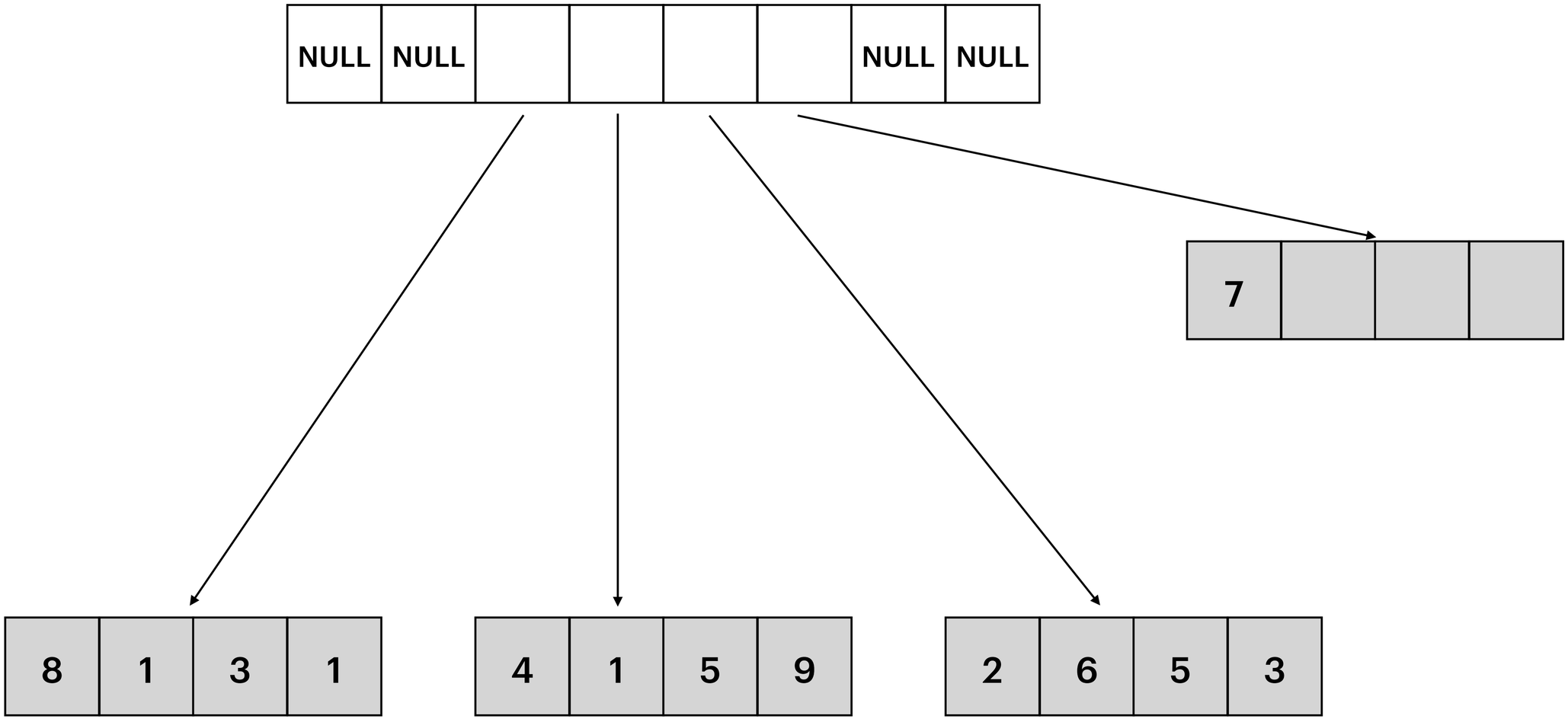
How is a deque actually implemented?



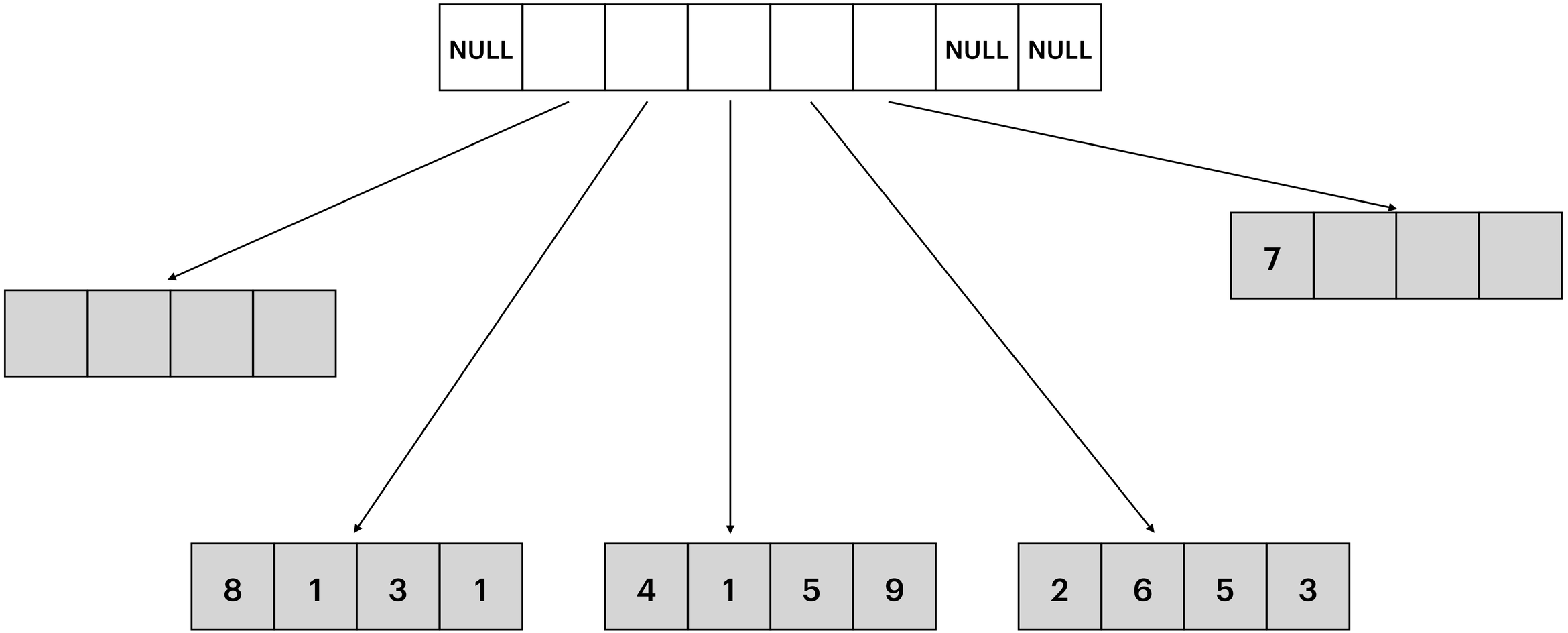
How is a deque actually implemented?



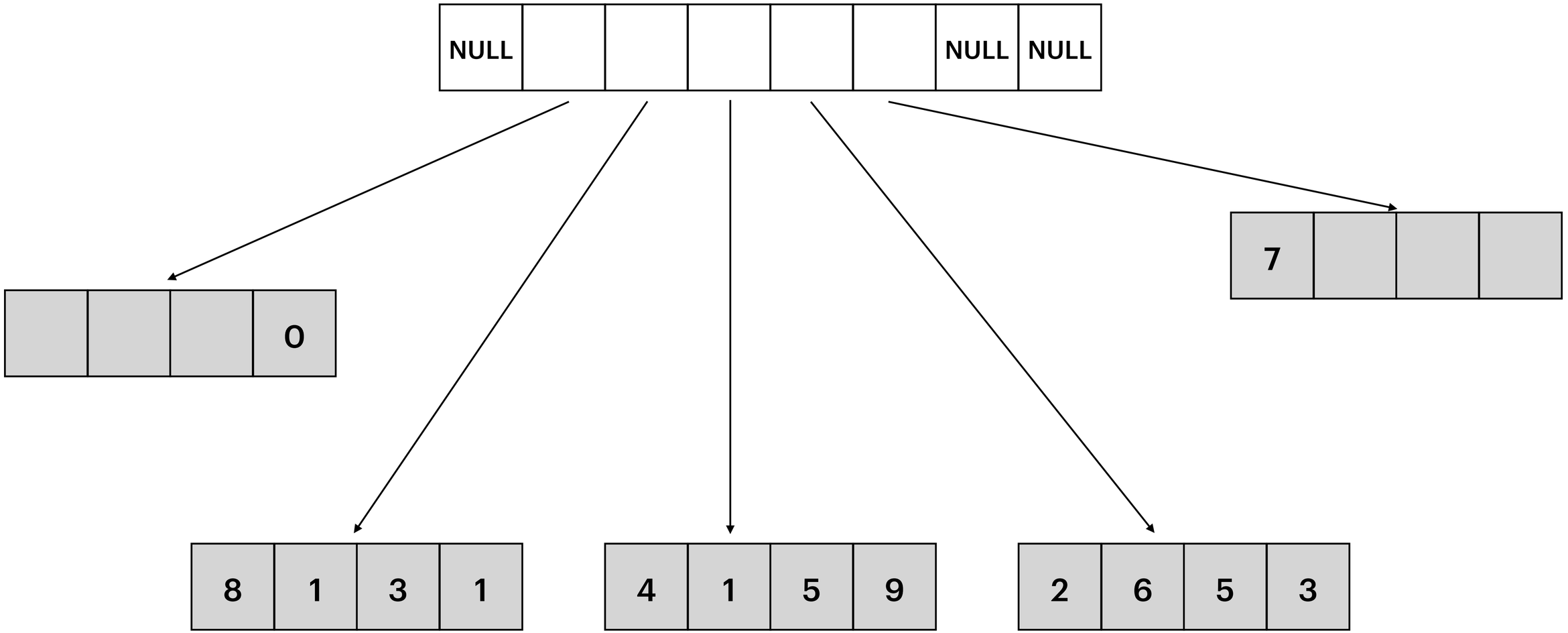
How is a deque actually implemented?



How is a deque actually implemented?



How is a deque actually implemented?



How is a list actually implemented?

- Recap: a list provides fast insertion anywhere, but no random (indexed) access

How is a list actually implemented?

- Recap: a list provides fast insertion anywhere, but no random (indexed) access

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

Usually a doubly linked list. There's also a forward_list that's singly linked. Linked lists will be covered at the end of 106B, so don't fret if this footnote is unfamiliar to you!

When to use which sequence container?

What you want to do	`std::vector`	`std::deque`	`std::list`
Insert/remove in the front	Slow	Fast	Fast
Insert/remove in the back	Super Fast	Very Fast	Fast
Indexed Access	Super Fast	Fast	Impossible
Insert/remove in the middle	Slow	Fast	Very Fast
Memory usage	Low	High	High
Combining (splicing/joining)	Slow	Very Slow	Fast
Stability* (iterators/concurrency)	Bad	Very Bad	Good

These two are the most common!

Don’t worry if you don’t know what stability means!
It’s a fairly advanced concept that you don’t need to understand in order to grasp the core of this slide.

Summary of Sequence Containers

`std::vector`: use for almost everything

`std::deque`: use if you are frequently inserting/removing at front

`std::list`: use very rarely, and only if you need to split/join multiple lists

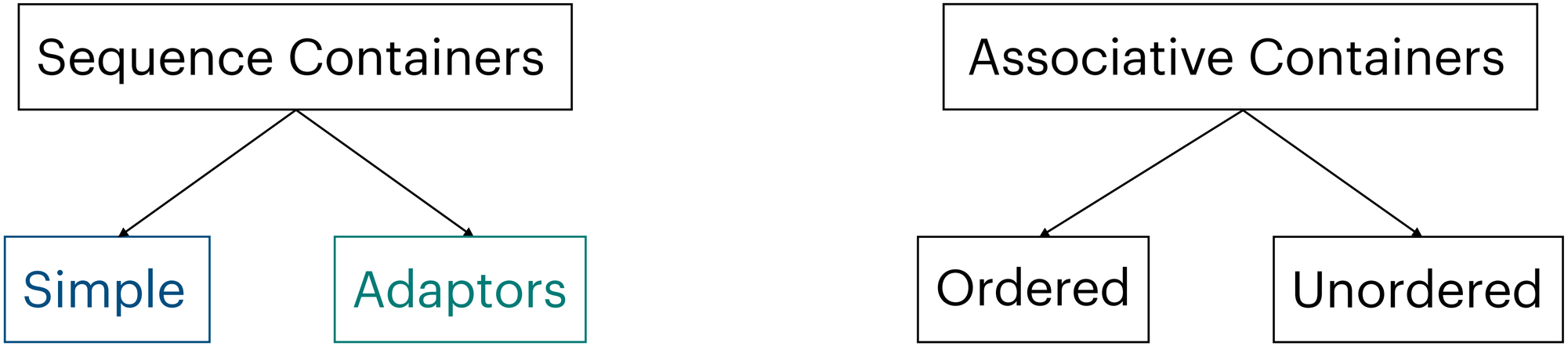
Container Adaptors

What is a container adaptor?

`std::stack` and `std::queue`

Types of containers

- All containers can hold almost all elements.



<> vector

⇅ deque

↓ list

() tuple

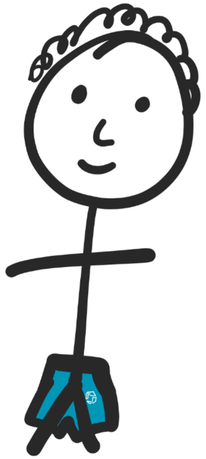
📚 stack (adding/removing elements from the front)

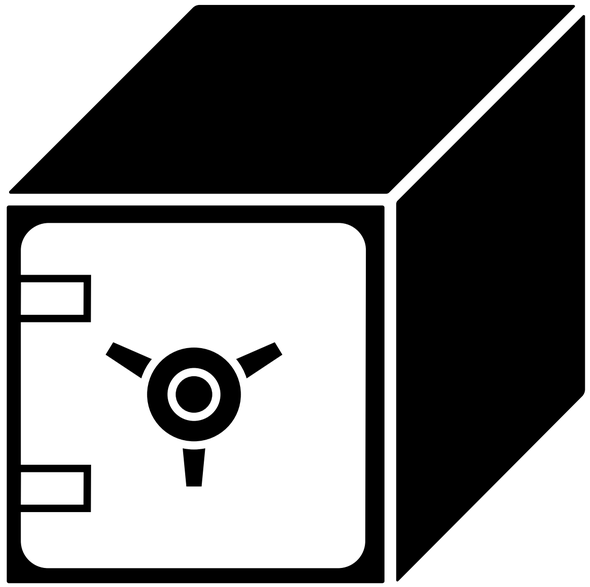
queue (adding elements from the front, removing from the back)

priority_queue (adding elements with a priority, always removing the highest priority-element)

What is a wrapper?

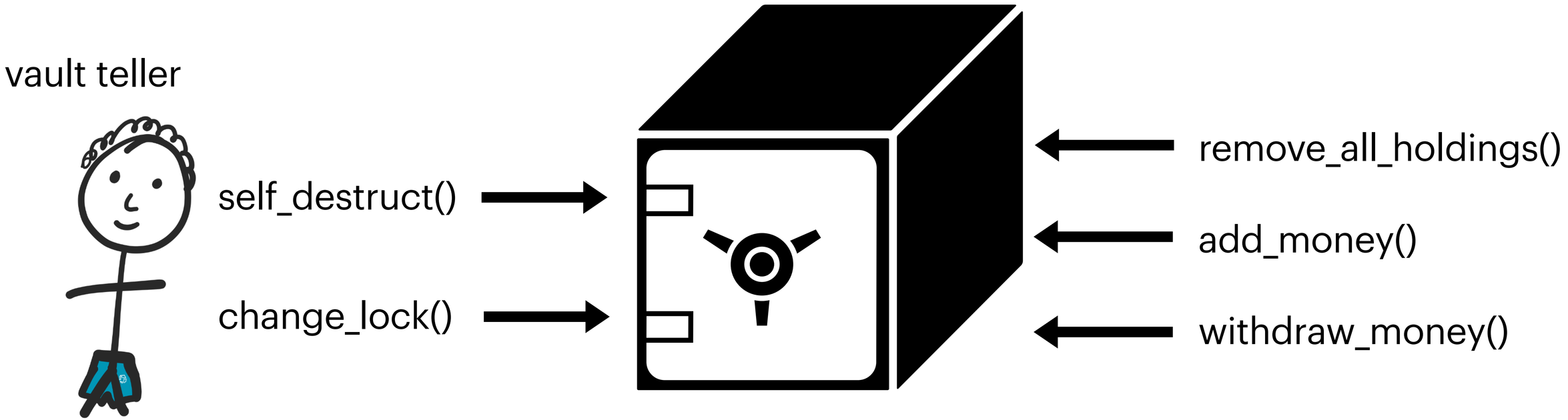
- A wrapper on an object changes how external users can interact with that object.





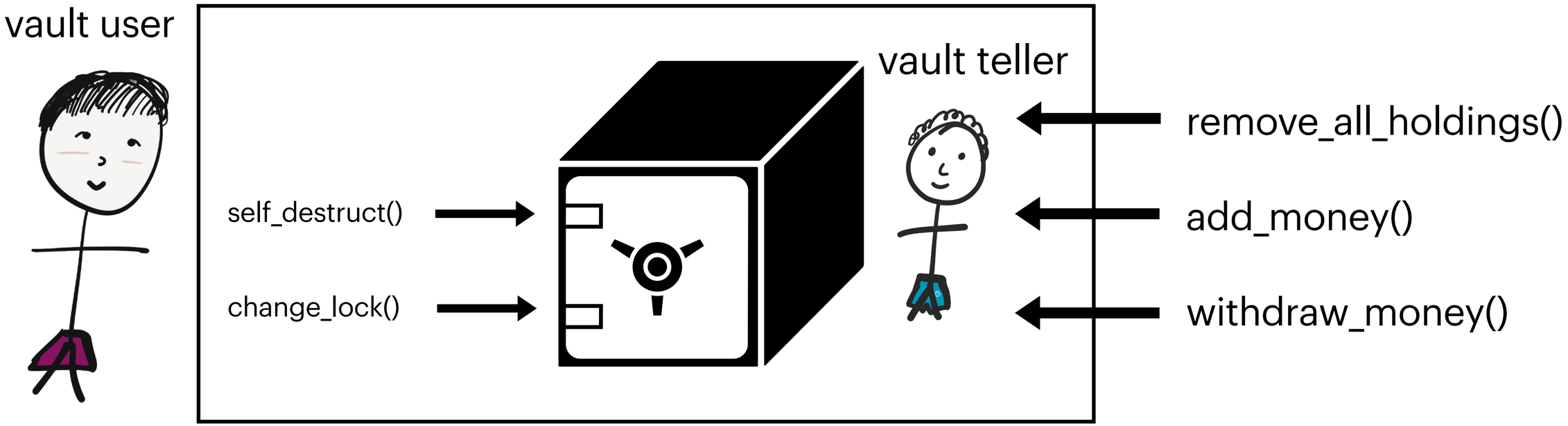
What is a wrapper?

- The vault's owner has access to all possible ways to use the vault!
- Should a vault customer be able to do the same actions as the vault owner?



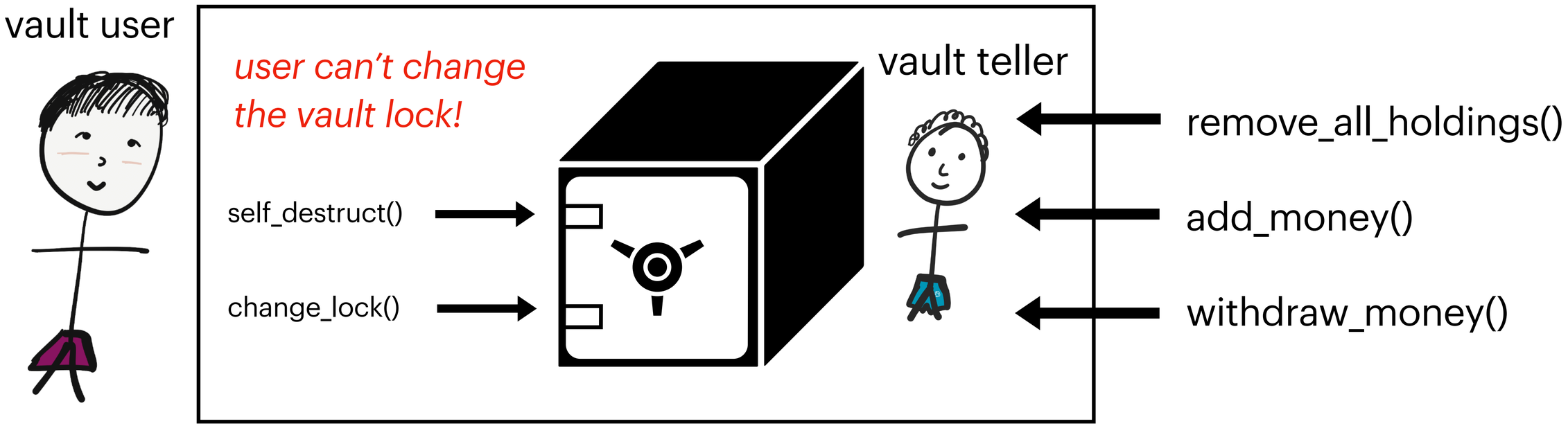
What is a wrapper?

- Of course not! The vault teller limits your access to the vault.
- The teller is in charge of forwarding your requests to the actual vault.



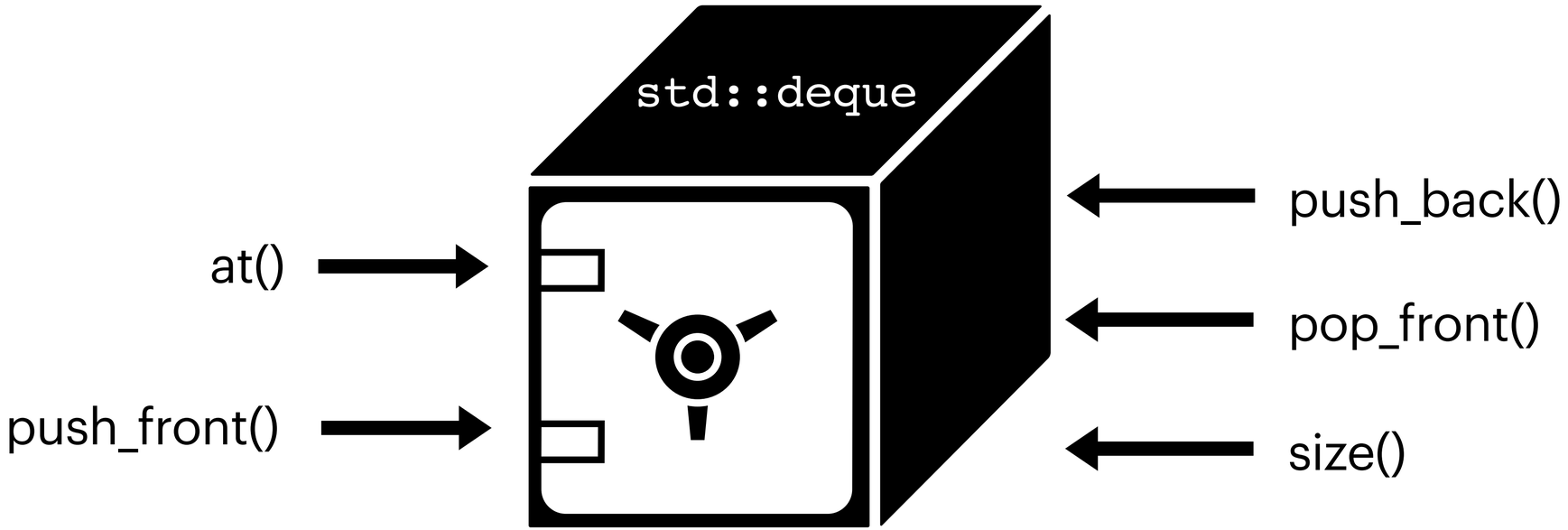
What is a wrapper?

- Of course not! The vault teller limits your access to the vault.
- The teller is in charge of forwarding your requests to the actual vault.



Container adaptors are wrappers in C++!

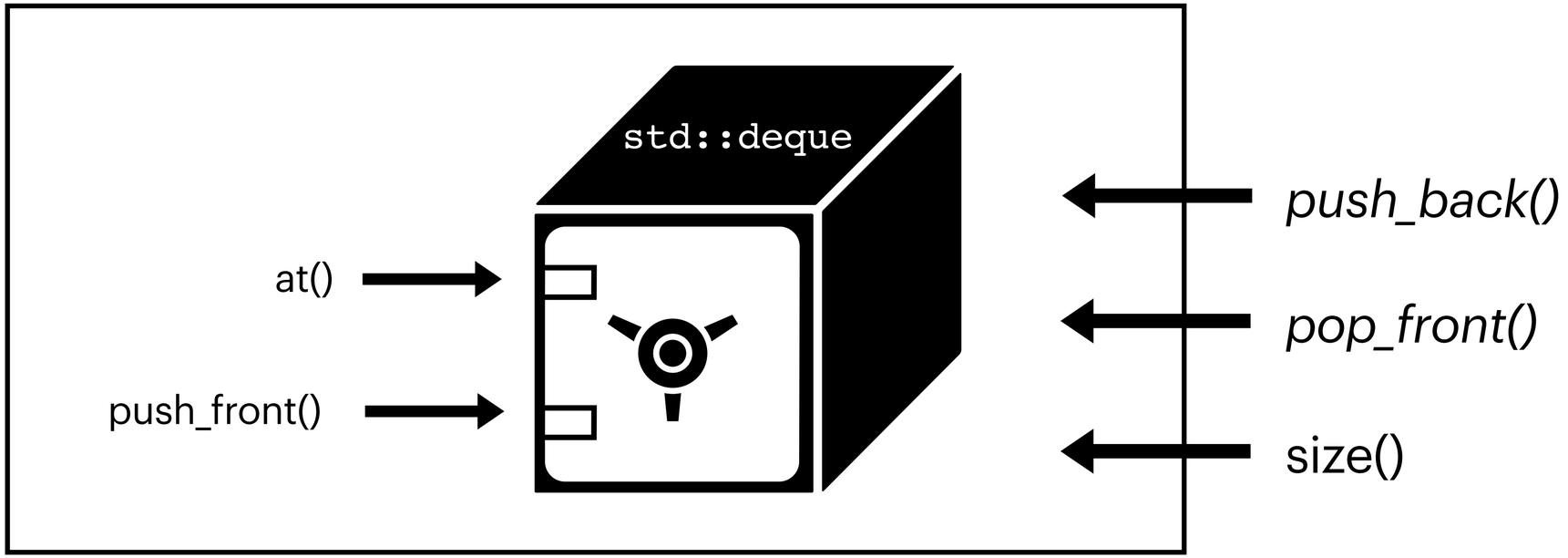
- Container adaptors provide a different interface for sequence containers.
- You can choose what the underlying container is!
- For instance, let's choose a deque as our underlying container, and let's implement a queue!



Container adaptors are wrappers in C++!

- Container adaptors provide a different interface for sequence containers.
- You can choose what the underlying container is!
- For instance, let's choose a deque as our underlying container, and let's implement a queue!

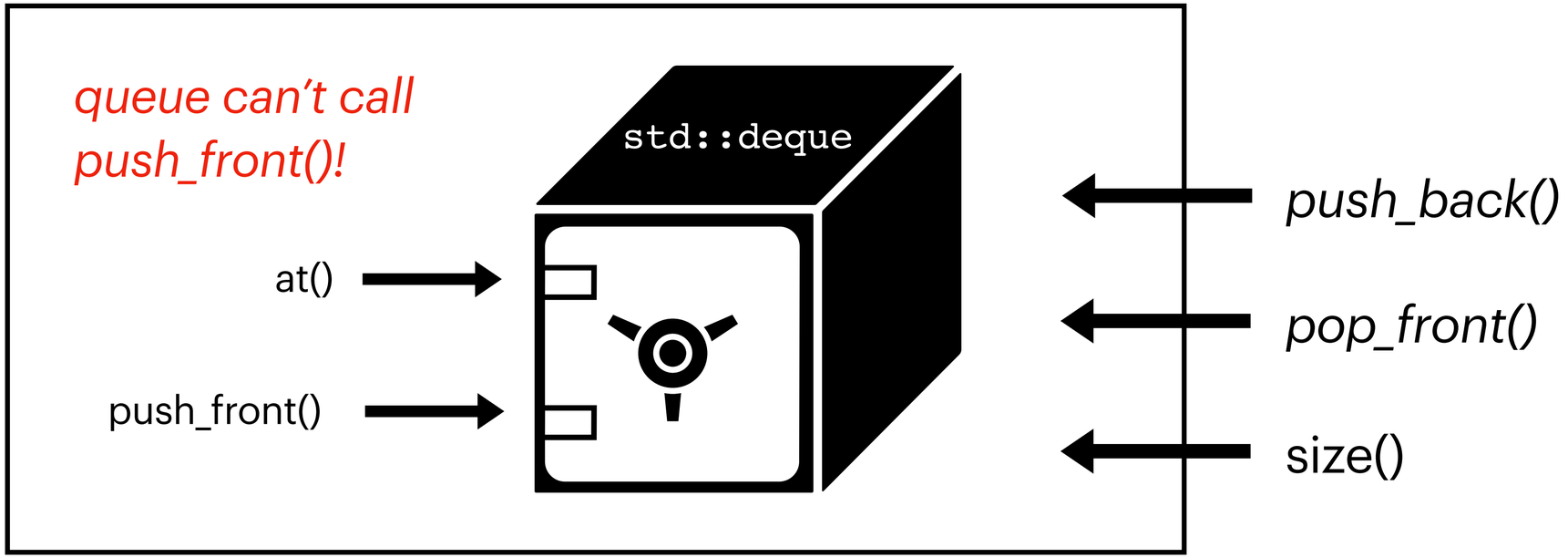
`std::queue`



Container adaptors are wrappers in C++!

- Container adaptors provide a different interface for sequence containers.
- You can choose what the underlying container is!
- For instance, let's choose a deque as our underlying container, and let's implement a queue!

`std::queue`



std::stack and std::queue

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.

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.

Concrete examples with `std::queue`

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.

```
std::queue<int> stack_deque;                    // Container = std::deque

std::queue<int, std::list<int>> stack_list;     // Container = std::list

std::queue<int, std::vector<int>> stack_vector; // Container = std::vector?
```

Concrete examples with `std::queue`

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.

```
std::queue<int> stack_deque;                    // Container = std::deque

std::queue<int, std::list<int>> stack_list;     // Container = std::list

std::queue<int, std::vector<int>> stack_vector; // Container = std::vector
```

removing from the front of a vector is slow!

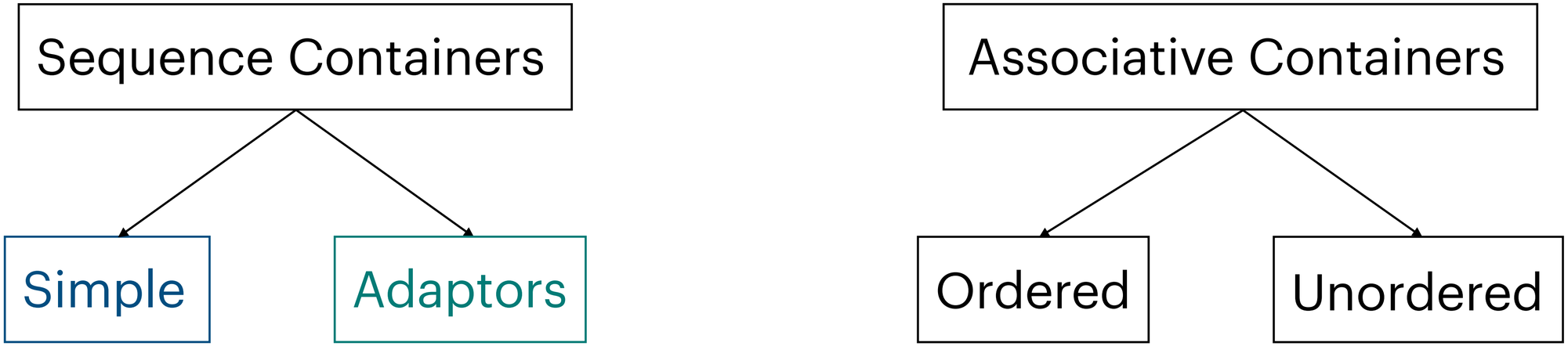
Some member functions of `std::queue`

Member functions

(constructor)	constructs the queue (public member function)
(destructor)	destructs the queue (public member function)
`operator=`	assigns values to the container adaptor (public member function)
Element access	
`front`	access the first element (public member function)
`back`	access the last element (public member function)
Capacity	
`empty`	checks whether the underlying container is empty (public member function)
`size`	returns the number of elements (public member function)
Modifiers	
`push`	inserts element at the end (public member function)
`emplace` (C++11)	constructs element in-place at the end (public member function)
`pop`	removes the first element (public member function)
`swap` (C++11)	swaps the contents (public member function)

Types of containers

- All containers can hold almost all elements.



<> vector

⇅ deque

↓ list

() tuple

📚 stack (adding/removing elements from the front)

queue (adding elements from the front, removing from the back)

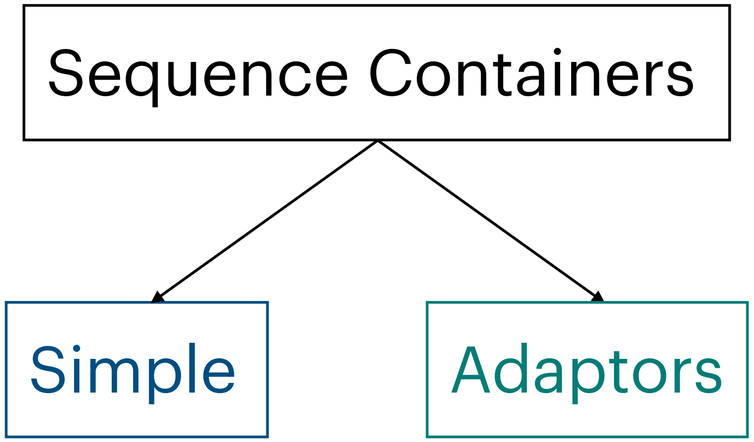
priority_queue (adding elements with a priority, always removing the highest priority-element)

Associative Containers

`std::set` functions
`std::map` functions and auto-insertion
type requirements

Types of containers

- All containers can hold almost all elements.



<> vector

⇅ deque

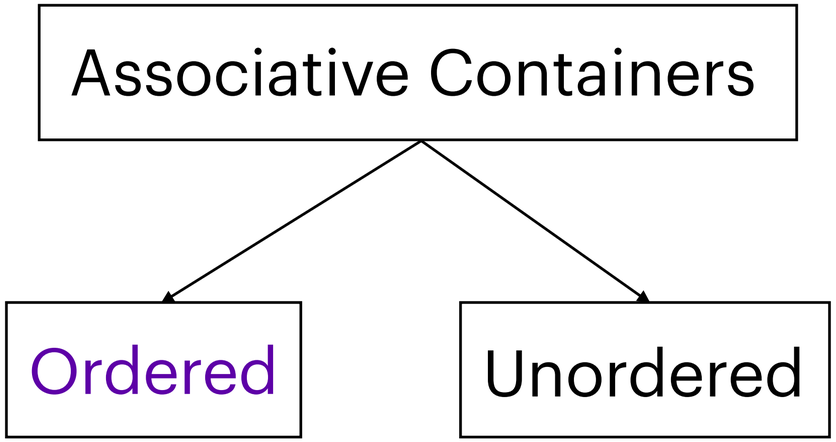
↓ list

() tuple

📚 stack

queue

priority_queue



{} set (unique elements)

{:} map (key value pairs)

Live Demo!

Let’s compare the Stanford’s Map/Set and STL’s map/set:
(QT Creator Project)

Stanford "Set" vs STL "set"

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

Stanford "Map" vs STL "map"

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

STL maps actually store pairs!

Every **`std::map<K, V>`** is actually backed by:

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

- Why do pairs make sense here?
  - Why not just tuples?
- Why is it **`const K`** instead of just **`K`**?

hint: **`std::pair`**'s are just two-element tuples!

Iterating through maps and sets

- Exactly the same as CS106B!
- Because maps are implemented with **`std::pair`**, you can use structured binding on them!

```
std::set<...> s;
std::map<..., ...> m;

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

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

Both Stanford and STL sets+maps require comparison operator!

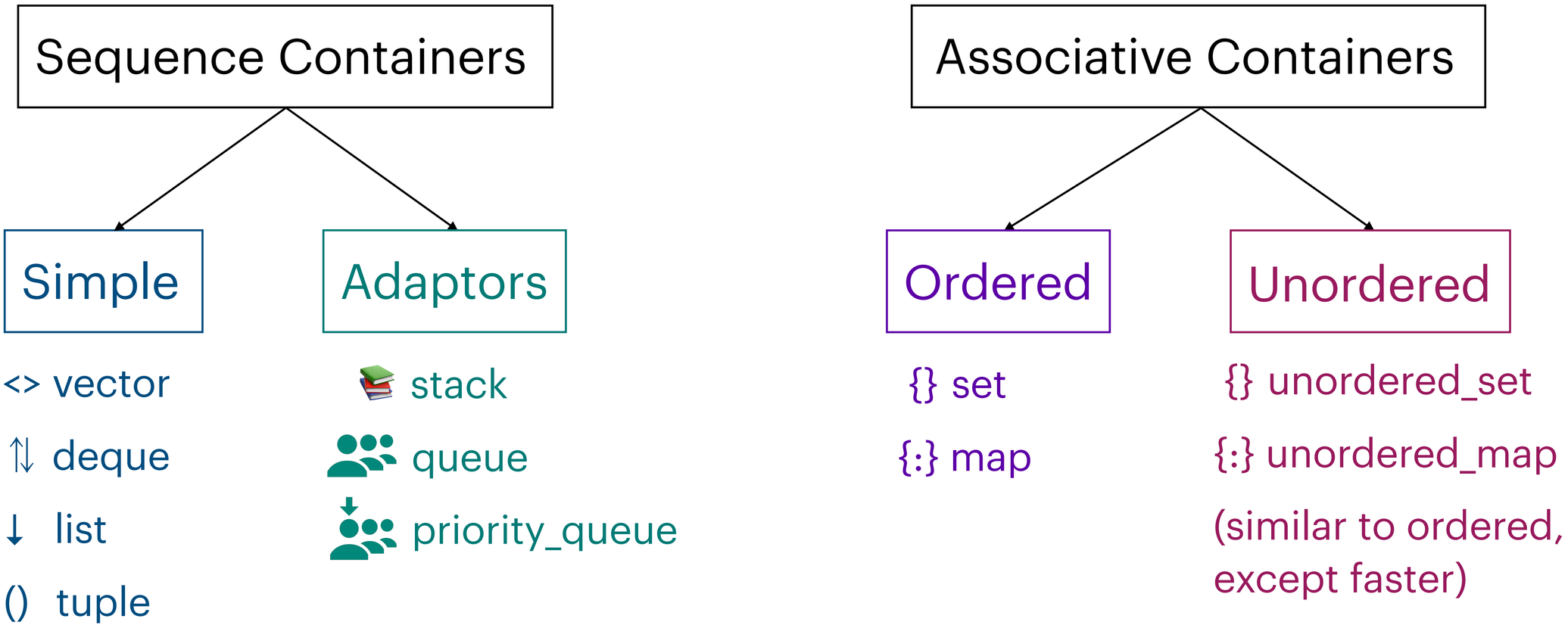
- By default, the type (for sets) or key's type (for maps) must have a comparison operator (<) defined.

```
std::set<int> set1; // A OK! ints are comparable with <
std::set<std::ifstream> set2; // not ok. how do we compare ifstreams with < ?

std::map<int, int> map1; // A OK! ints are comparable with <
std::set<std::ifstream, int> map2; // not ok. how do we compare ifstreams with < ?
```

Types of containers

- All containers can hold almost all elements.



unordered_map and unordered_set

- Each STL set/map comes with an unordered sibling. They're almost the same, except:
  - Instead of a comparison operator, the set/map type must have a hash function defined for it.
    - Simple types, like i**`nt, char, bool, double,`** and even **`std::string`** are already supported!
    - Any containers/collections need you to provide a hash function to use them.
  - unordered_map/unordered_set are generally faster than map/set.

That's a lot! Any broad tips for choosing one?

- What do they all have in common?
  - You can copy all of them!
  - You can check if two have the same elements in the same order.
  - You can get their size.
  - You can use iterators to access them (next lecture).

That's a lot! Any broad tips for choosing one?

- How are they different?
  - Most containers can hold any data type.
  - Unordered associative containers (sets and maps) are tricky to get working if the element or key is another collection.

Recap of STL Containers!

- Sequence Containers
  - **`std::vector`** - use for almost everything
  - **`std::deque`** - use when you need fast insertion to front AND back
- Container Adaptors
  - **`std::stack`** and **`std::queue`**
- Associative Containers
  - **`std::map`** and **`std::set`**
  - if using simple data types/you're familiar with hash functions, use **`std::unordered_map`** and **`std::unordered_set`**