Sequence Containers

Reid Watson
(rawatson@stanford.edu)

Administrivia

- Assignment one is out today!
- Quick Demo
- Due: Wednesday October 30th, 11:59 PM

Administrivia

- LaIR help available:
  - Sunday 8-12
  - Monday 8-10
  - More times to be announced, check the website for details
- Email help available
  - cs106l-staff-spr2013@lists.stanford.edu
  - rawatson@stanford.edu
  - Email cs106l-staff for best response time with assignment help

Review: Sequence Containers

- A container class allows you to store an arbitrary number of things
- A sequence container is a container whose elements can be accessed sequentially.
- Sequence containers include vectors, stacks, queues, lists, and priority queues (among others).

What I Want To Show You

- Why the Stanford library exists
- How to use STL sequence containers instead of the Stanford Library
  - We'll look at the differences between STL/Stanford using stack and vector, and we'll also examine a new STL class, deque
- The performance characteristics of various sequence containers, and why you might choose one over another

Why the Stanford Library Exists

Students often ask:

"Why do we need to use the Stanford libraries in CS106B/X?"

Why the Stanford Library Exists

- The Stanford libraries include things not found in the STL (`Grid`, `getInteger` and friends, graphics).
- Many parts of the Stanford library give up performance for simplicity
- Debugging Stanford library code can be much easier than debugging STL code (see HugeError.pro)

Container #1: Stack

First, let's talk about how stacks are represented in the STL.

STL <stack>: What's Similar

What you want to do	Stanford Stack<int>	STL stack<int>
Create a stack	`Stack<int> x;`	`stack<int> x;`
Get the size of a stack	`int size = x.size();`	`int size = x.size();`
Check if a stack is empty	`if (x.isEmpty()) ...`	`if (x.empty()) ...`
Push a value on the stack	`x.push(42);`	`x.push(42);`
Peek at the top element without popping it	`int top = x.peek();`	`int top = x.top();`
Pop off the top element and ignore its value	`x.pop();`	`x.pop();`

STL <stack>: What's Different

What you want to do	Stanford Stack<int>	STL stack<int>
Clear the stack	`x.clear();`	`while(!x.empty())` `  x.pop();`

Convert the stack to a string	`string s = x.`
	`toString();`
		`string s;`
		`while(!x.empty() {`
		`  s += x.top();`
		`  s += " ";`
		`  x.pop();`
		`}`

Pop and save the value	`int top = x.pop();`	`int top = x.top();` `x.pop();`

STL <stack>: Usage

Let's look at a quick demo in `STLStack.pro`

STL <stack>: Why the differences?

Looking at the differences between the STL and the Stanford libraries can help you understand the guiding principles behind how each of these libraries were designed.

STL <stack>: Why the differences?

Why is there no .clear() function for stacks?

STL <stack>: Why the differences?

Why is there no .clear() function for stacks?

- Conceptually, clearing isn't part of the interface to a stack
- It's very easy to write your own clear function:

```
// stack<int> s = ...;
while (!s.empty()) {
  s.pop();
}
```

STL <stack>: Why the differences?

Why doesn't pop return the value it removed?

STL <stack>: Why the differences?

Why doesn't pop return the value it removed?

- The caller might not need the value, in which case returning the value would be wasteful.
- It's easy to write code which pops and saves the value.

```
// stack<int> s = ...;
int value = s.top();
s.pop();
```

STL <stack>: Why the differences?

Why isn't there a `toString` function?

STL <stack>: Why the differences?

Why isn't there a `toString` function?

- Implementing toString would require that the type stored in the stack could be converted to a string
  - For example, you can convert a `stack<int>` to a `string` because you can convert an `int` to a `string`.
- It's tough to say what the "proper" way to convert a stack to a string is

Container #2: Vector

First, let's talk about how vectors are represented in the STL.

STL <vector>: What's Similar

What you want to do	Stanford Vector<int>	STL 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 a value k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear a 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;`

STL <vector>: What's Different

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;`
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...`
Add an element to the beginning of a vector	`// Impossible!`	`// Impossible!`

STL <vector>: Usage

Let's look at a quick demo in `STLVector.pro`

STL <vector>: Why the differences?

Why doesn't vector have bounds checking?

STL <vector>: Why the differences?

Why doesn't vector have bounds checking?

- If you write your program correctly, bounds checking will do nothing but make your code run slower

STL <vector>: Why the differences?

Why is there no push_front method?

STL <vector>: Why the differences?

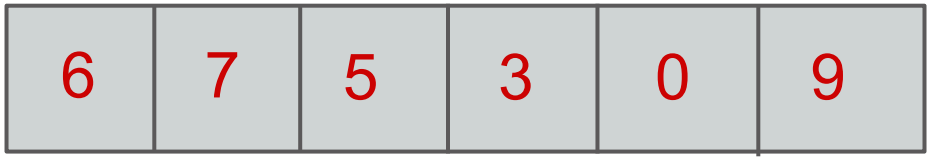
Why is there no push_front method?

- This is a bit more complicated

The Mystery of `push_front`

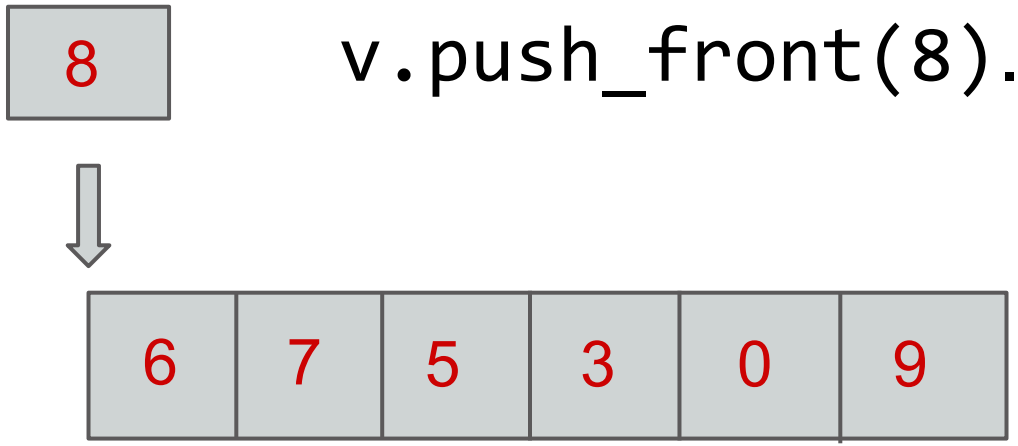
Pushing an element to the front of the vector requires shifting all other elements in the vector down by one, which can be **very** slow

To demonstrate this, let's say we had this nice little vector:



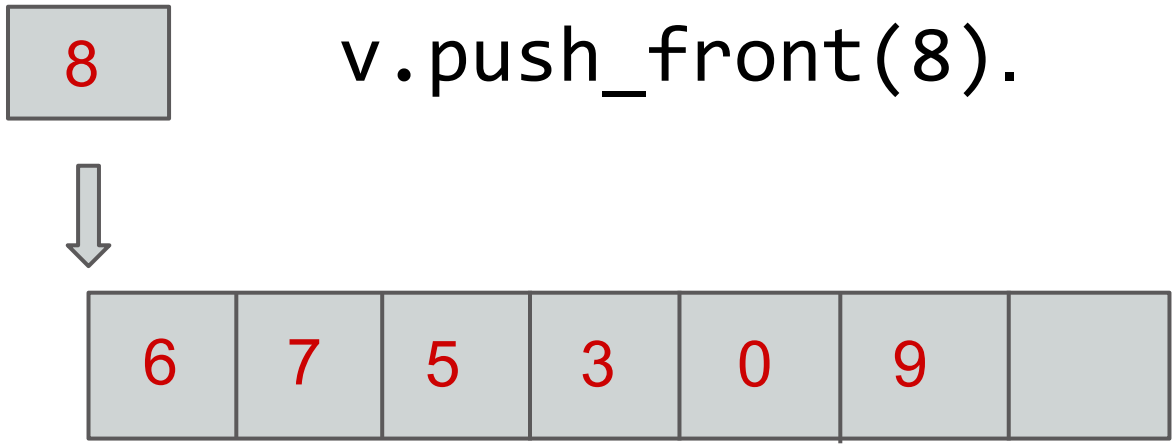
The Mystery of `push_front`

Now, let's say that `push_front` existed, and that you wanted to insert an 8 at the beginning of this vector.



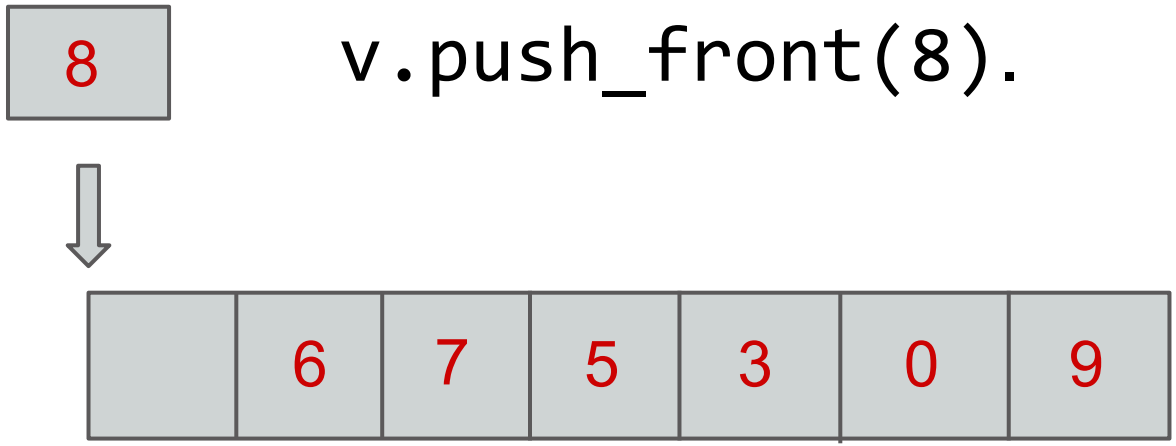
The Mystery of `push_front`

First, we may have to expand the capacity of the vector



The Mystery of `push_front`

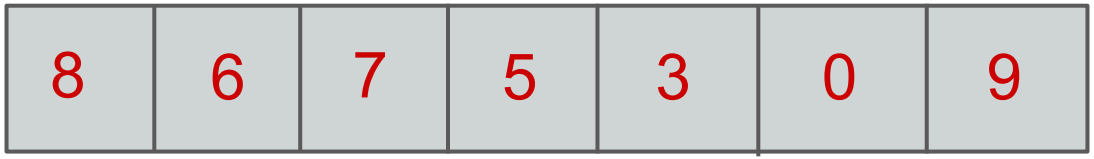
Then, we'll need to shift every single element down one position



The Mystery of `push_front`

Finally, we can actually insert the element we wanted to insert.

```
v.push_front(8).
```



Just how bad is push_front?

```
// Adding to the back
for (int i = 0; i < N; i++)
   v.push_back(i);

// Or: Adding to the front
for (int i = 0; i < N; i++)
   v.insert(v.begin(), i);

// How big can the difference be?
```

Just how bad is push_front?

	`push_front`	`push_back`
N = 1000	0.01	0
N = 10000	0.89	0.01
N = 100000	117.98	0.04
N = 1000000	Hours	0.31
N = 10000000	Years	3.16

You can see the difference between an $O(n^2)$ algorithm and an O(n) algorithm!

STL <deque>: What's a deque?

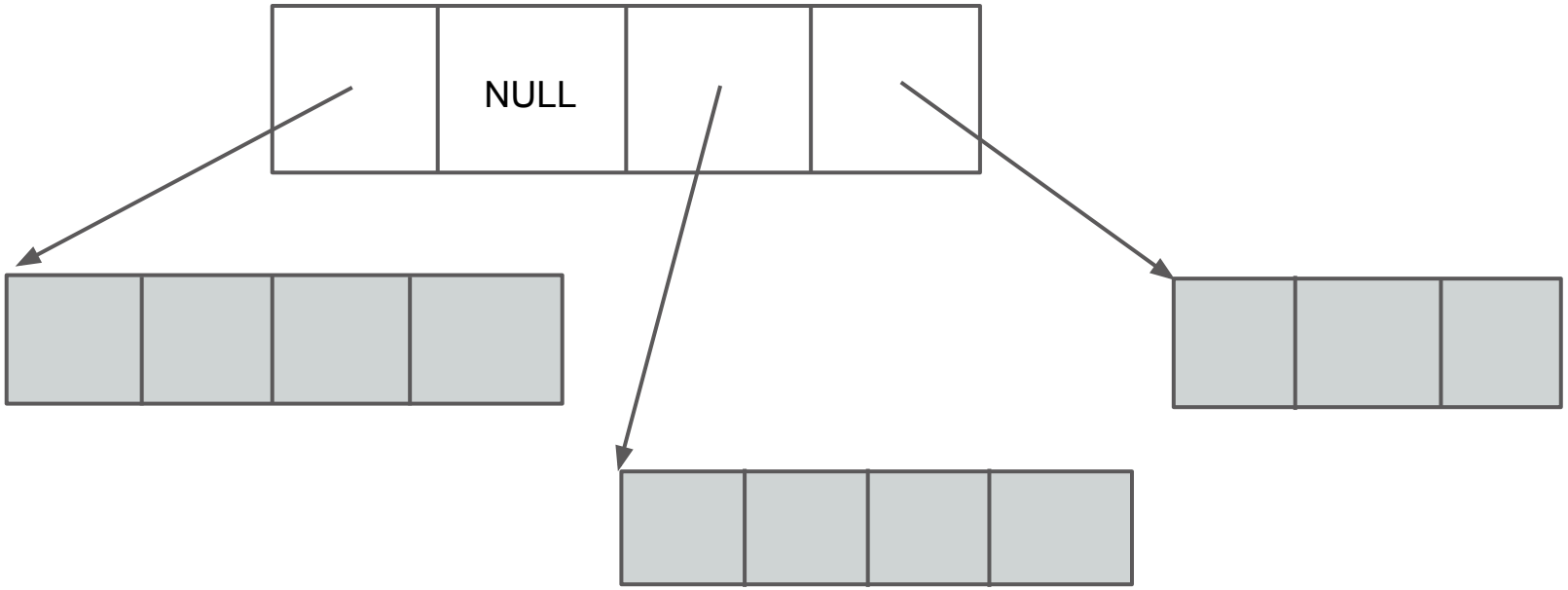
- A deque (pronounced "deck") is a **d**ouble **e**nded **que**ue
- Unlike a vector, it's possible (and fast) to push_front
- The implementation of a deque isn't as straightforward as a vector though

STL <deque>: Usage

Let's look at a quick demo in `STLDeque.cpp`

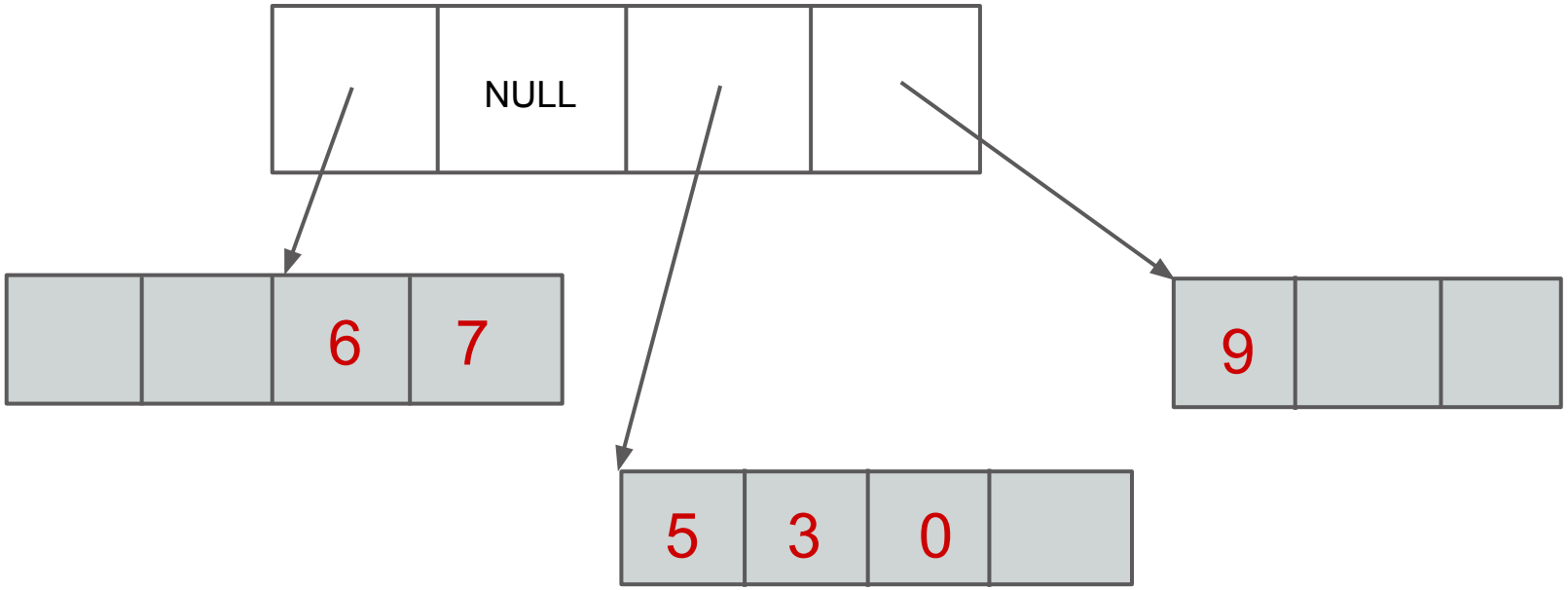
STL <deque>: Implementation

There's no single specification for representing a deque, but it might be laid out something like this



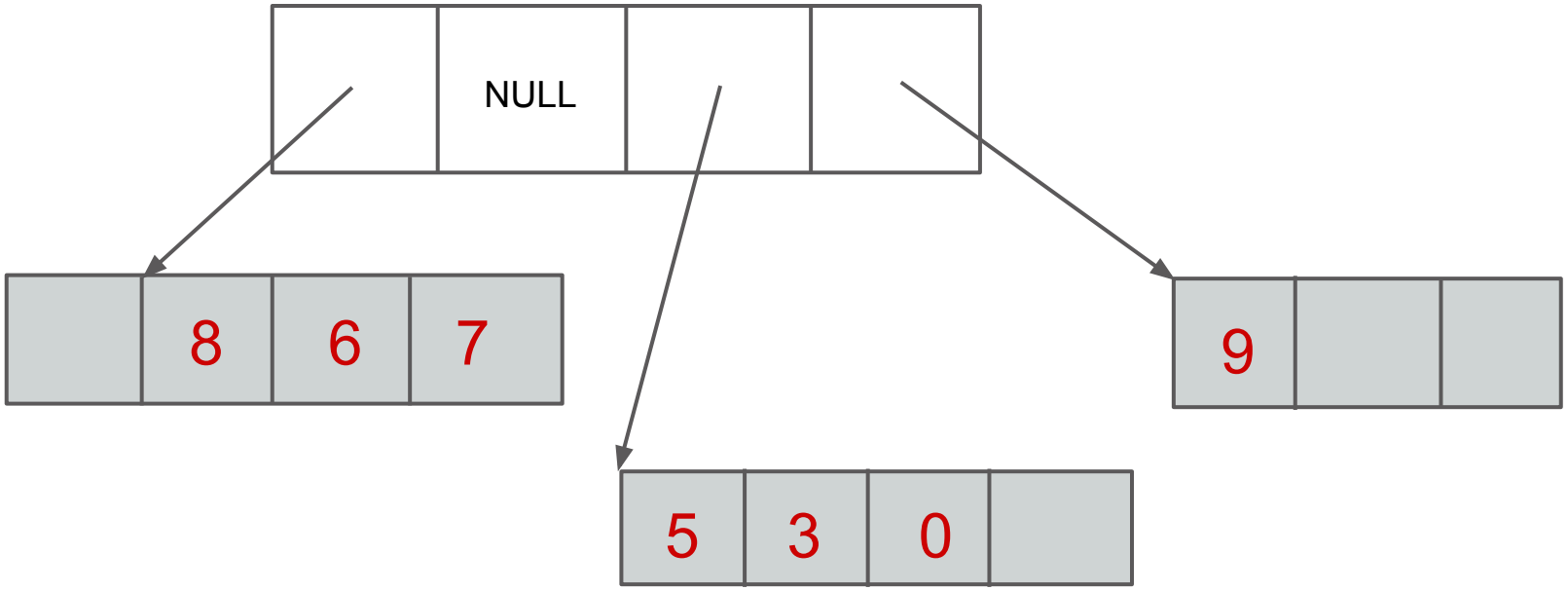
STL <deque>: Implementation

You could support efficient insertion by keeping some reserved space in front of the vector representing the first elements of the deque



STL <deque>: Implementation

You could support efficient insertion by keeping some reserved space in front of the vector representing the first elements of the deque



STL <deque>: Performance

- We can now use the `push_front` function, and it will run much faster than if we had used a vector.
- However, if all you're doing is iterating, resizing, and `push_backing`, then using a vector will be faster.
- Let's see how this looks in real world performance numbers

push_front: vector and deque

```
// Vector test code
vector<int> v;
// Insert at the start of the vector
for (int i = 0; i < N; i++)
   v.insert(v.begin(), i);
// Clear by using pop_front (erase)
for (int i = 0; i < N; i++)
   v.erase(v.begin());
```

push_front: vector and deque

```
// Deque test code
deque<int> d;
// Insert elements using push_front
for (int i = 0; i < N; i++)
   d.push_front(i);
// Clear by using pop_front
for (int i = 0; i < N; i++)
   d.pop_front();
```

push_front: vector and deque

	<vector>	<deque>
N = 1000	0.02	0
N = 10000	2.12	0.01
N = 100000	264.9	0.04
N = 1000000	Years	0.44
N = 10000000	Millenia	5.54

Element Access: vector and deque

```
vector<int> v;
deque<int> d;

for (int i = 0; i < N; i++)
   v[i] = i;

for (int i = 0; i < N; i++)
   d[i] = i;
```

Access: vector and deque

	<vector>	<deque>
N = 1000	0.02	0.14
N = 10000	0.28	1.32
N = 100000	3.02	13.22
N = 1000000	30.84	133.30

push_back: vector and deque

```
// Vector test code
vector<int> v;
// Insert elements using push_back
for (int i = 0; i < N; i++)
   v.push_back(i);
// Clear by using pop_back
for (int i = 0; i < N; i++)
   v.pop_back();
```

push_back: vector and deque

```
// Deque test code
deque<int> d;
// Insert elements using push_back
for (int i = 0; i < N; i++)
   d.push_back(i);
// Clear by using pop_back
for (int i = 0; i < N; i++)
   d.pop_back();
```

push_back: vector and deque

	<vector>	<deque>
N = 1000	0.02	0.02
N = 10000	0.20	0.20
N = 100000	1.98	1.92
N = 1000000	19.9	20.78

Other Sequence Containers

The STL also includes priority queue, queue, and linked list classes, but those aren't too important to us right now.