CS 225

Data Structures

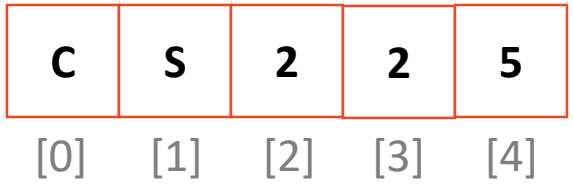
Sept. 21 – Stacks and Queues

Wade Fagen-Ulmschneider

List.h

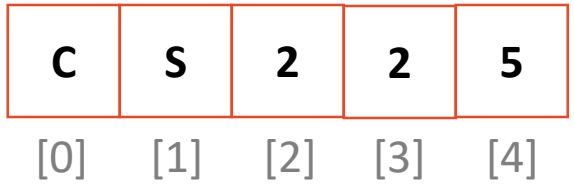
```
 1 #pragma once
 2
 3 template <typename T>
 4 class List {
 5   public:
…      /* ... */
28   private:
29
30
31
32
33 };
```

Array Implementation

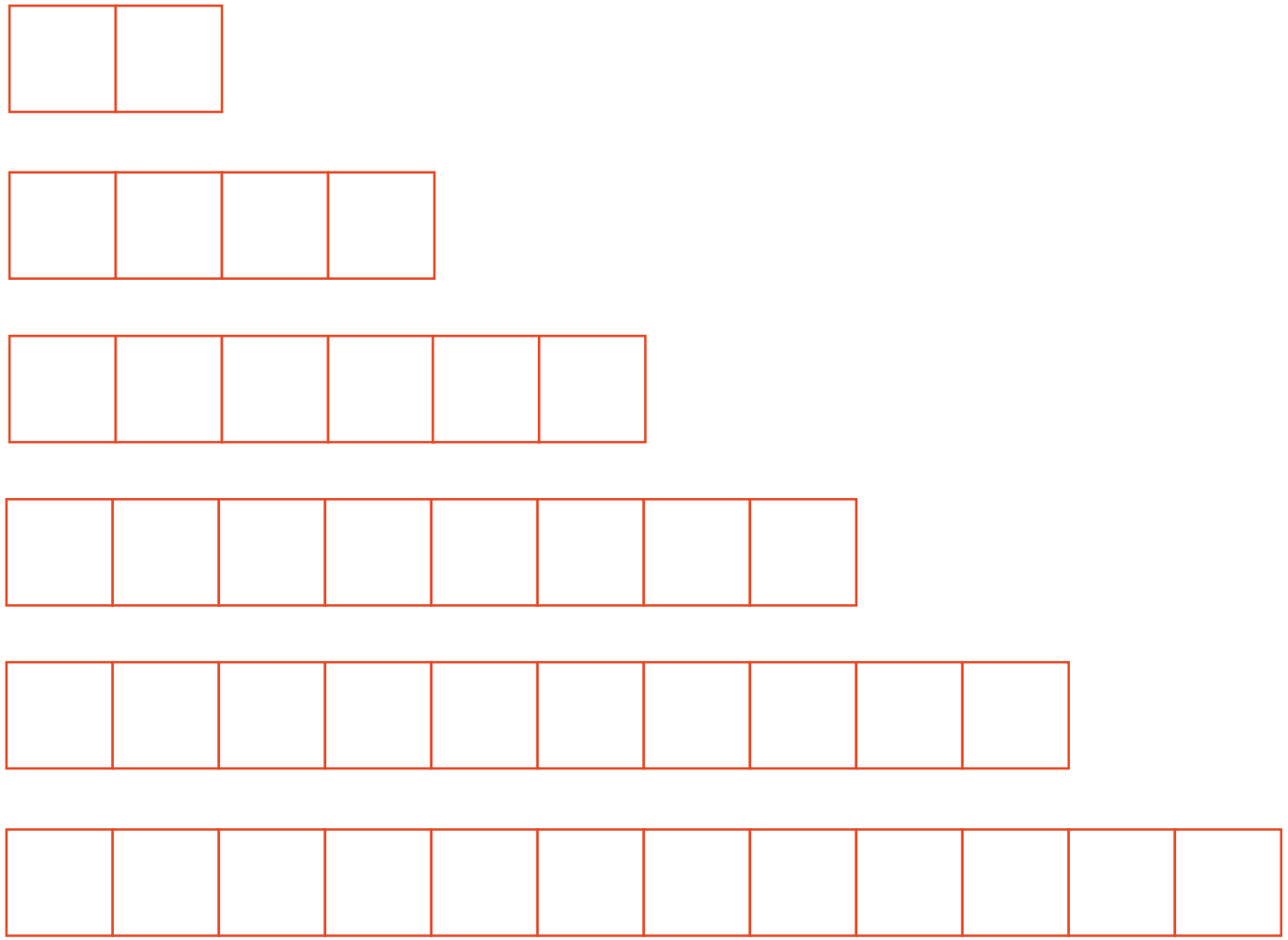


Array Implementation

insertAtFront:

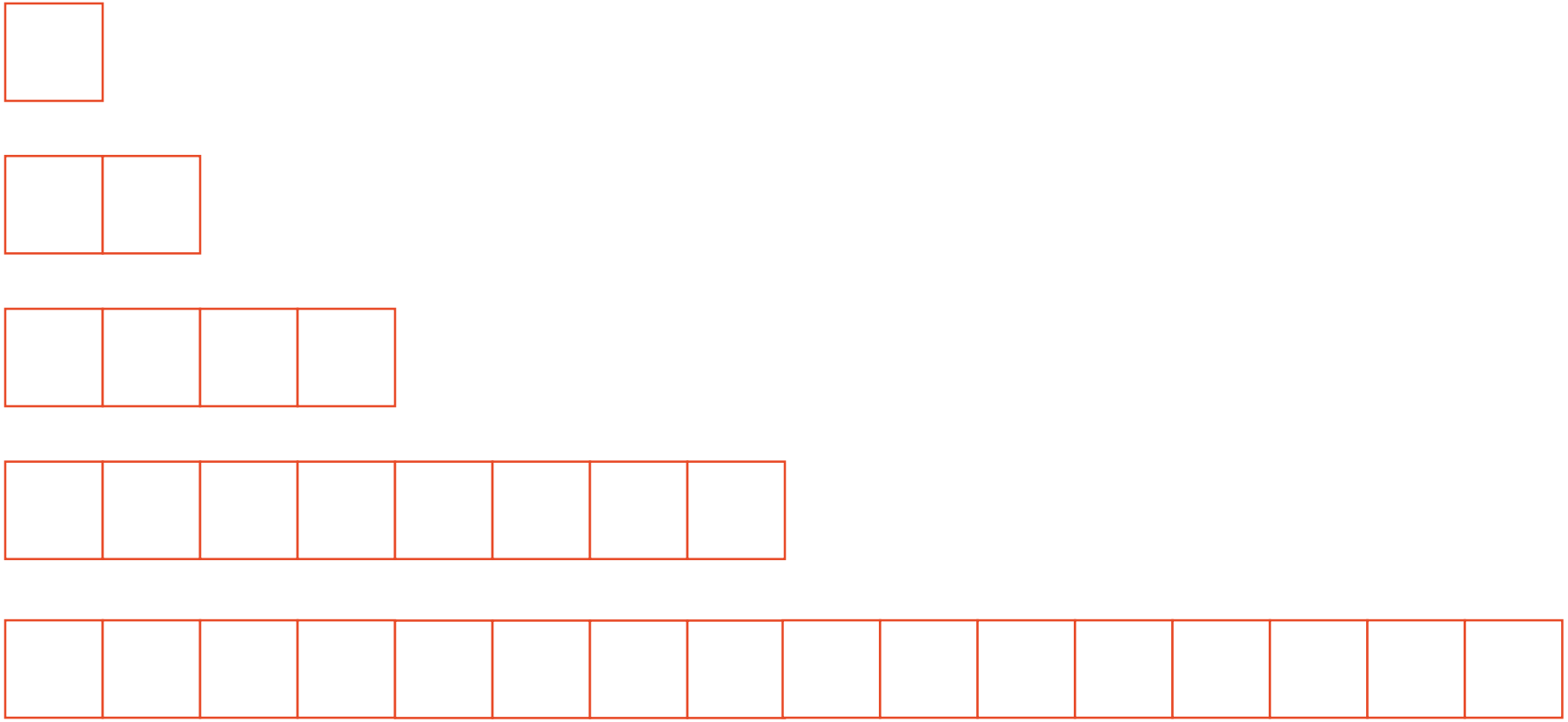


Resize Strategy: +2 elements every time



Resize Strategy: +2 elements every time

Resize Strategy: x2 elements every time



Resize Strategy: x2 elements every time

Array Implementation

	Singly Linked List	Array
Insert/Remove at **front**		
Insert at **given** element		
Remove at **given** element		
Insert at **arbitrary** location		
Remove at **arbitrary** location		

std::vector

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```
(1)

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array. (since C++03)

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function. Extra memory can be returned to the system via a call to `shrink_to_fit()`. (since C++11)

Element access	
`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data` (C++11)	direct access to the underlying array (public member function)

Modifiers	
`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

Capacity	
`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in currently allocated storage (public member function)
`shrink_to_fit` (C++11)	reduces memory usage by freeing unused memory (public member function)

Stack ADT

Queue ADT

Stack.h

```
1  #pragma once
2
3  #include <vector>
4
5  template <typename T>
6  class Stack {
7    public:
8      void push(T & t);
9      T & pop();
10     bool isEmpty();
11
12   private:
13     std::vector<T> list_;
14 };
15
16 #include "Stack.hpp"
```

Stack Implementation

```
 3 template <typename T>
 4 void Stack<T>::push(const T & t) {
 5   list_.push_back(t);
 6 }
 7
 8 template <typename T>
 9 const T & Stack<T>::pop() {
10   const T & data = list_.back();
11   list_.pop_back();
12   return data;
13 }
```

Implications of Design

1.

```
class ListNode {
  public:
    T & data;
    ListNode * next;
    …
```

2.

```
class ListNode {
  public:
    T * data;    …
```

3.

```
class ListNode {
  public:
    T data;    …
```

Implications of Design

	Storage by Reference	Storage by Pointer	Storage by Value
Who manages the lifecycle of the data?			
Is it possible for the data structure to store NULL?			
If the data is manipulated by user code while in our data structure, is the change reflected in our data structure?			
Is it possible to store literals?			
Speed			

Data Lifecycle

Storage by reference:

```
1 Sphere s;
2 myStack.push(s);
```

Storage by pointer:

```
1 Sphere s;
2 myStack.push(&s);
```

Storage by value:

```
1 Sphere s;
2 myStack.push(s);
```

Possible to store NULL?

Storage by reference:

```
class ListNode {
  public:
    T & data;
    ListNode * next;
    ListNode(T & data) : data(data), next(NULL) { }
};
```

Storage by pointer:

```
T ** arr;
```

Storage by value:

```
T * arr;
```

Data Modifications

```
1  Sphere s(1);
2  myStack.push(s);
3
4  s.setRadius(42);
5
6  Sphere r = myStack.pop();
7  // What is r's radius?
```

Speed