Testing Details

Unit Testing: Dealing with Scale

- **Best practice** is to test individual ***units*** or ***components*** of software
  - Test one class's operation at a time
  - This is known as ***unit testing***

Unit Testing:

This is the kind of testing we will do in this course.

- **Best practice** is to test individual ***units*** or ***components*** of software
  - Test one class's operation at a time
  - This is known as ***unit testing***

Unit Testing:

And the unit being tested is known as the *unit under test*

- **Best practice** is to test individual ***units*** or ***components*** of software
  - Test one class's operation at a time
  - This is known as ***unit testing***

Testing On a Larger Scale

- ***Integration testing*** – Is when testing involves multiple components (classes) that are put together to form a larger subsystem

- ***System testing*** – Is when the entire end-user system comes under test

Testing Functional Correctness

- What does it mean for a program unit to be ***correct***?

Testing Functional Correctness

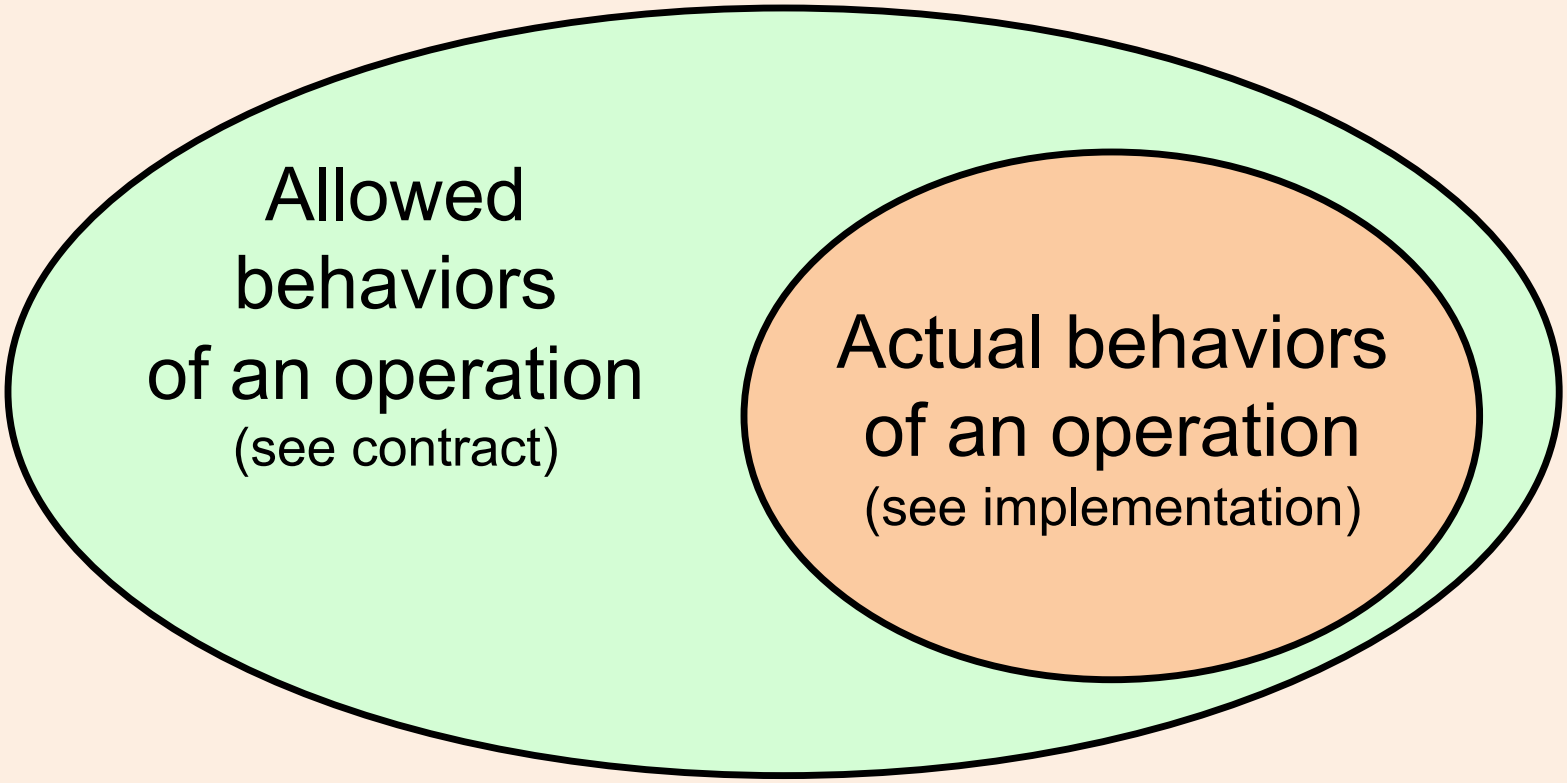
- What does it mean for a program unit to be ***correct***?

- The following answers are vague
  - It does what it is supposed to do.
  - It doesn’t do what it is not supposed to do.

“Supposed To Do”?

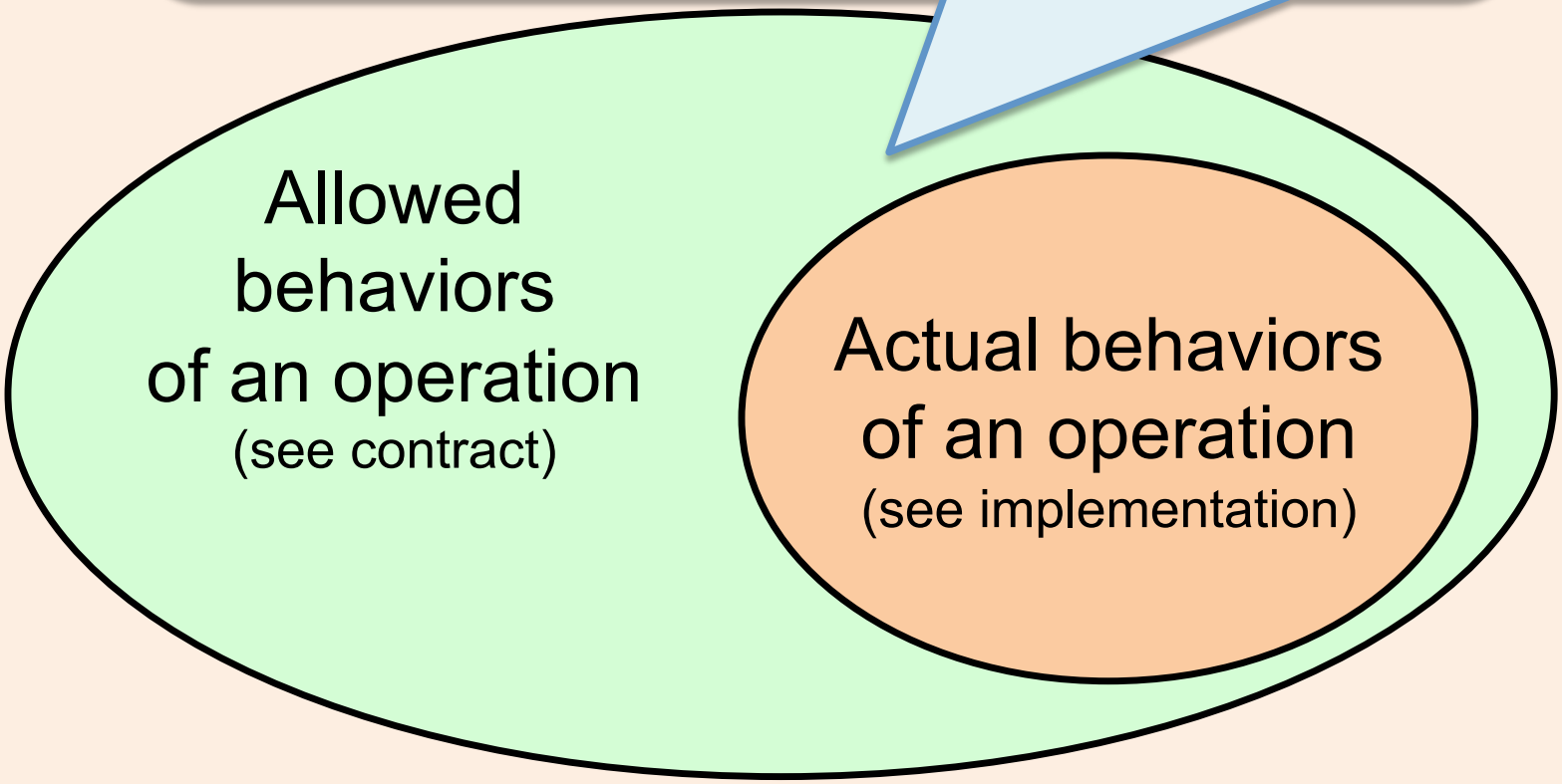
- How do we know what an operation *is supposed to do*, and what it is *not supposed to do*?
  - We look at the operation’s ***contract***, which is a ***specification*** of its ***intended behavior***

Allowed & Actual Behaviors



A

Each point in this space is a ***legal input*** with a corresponding ***allowable result***. Represented as 2-tuples: (legal input, allowable result)



Example: Queue's *length* Contract

```
Integer length(void);
   //! preserves self
   //! ensures: length = |self|
```

Example: Queue's *length* Contract

```
Integer length(void);
   //! preserves self
   //! ensures: length = |self|
```

This means:
"*length* returns a count of the number of items currently in the queue"

Example: Client of Queue

```
#include "Wrapper.h”
#include "Queue\Queue1.hpp”

typedef Queue1<Integer> QueueOfInteger;

int main(int argc, char* argv[])
{
   QueueOfInteger q1;

   cout << q1.length();
}
```

Example: Client of Queue

```
#include "Wrapper.h”
#include "Queue\Queue1.hpp”

typedef Queue1<Integer> QueueOfInteger;

int main(int argc, char* argv[])
{
   QueueOfInteger q1;

   cout << q1.length();
}
```

Instantiate Queue1 with Integer to create QueueOfInteger

Example: Client of Queue

```
#include "Wrapper.h”
#include "Queue\Queue1.hpp”

typedef Queue1<Integer> QueueOfInteger;

int main(int argc, char* argv[])
{
   QueueOfInteger q1;

   cout << q1.length();
}
```

Declare variable q1 from QueueOfInteger

Example: Client of Queue

```
#include "Wrapper.h”
#include "Queue\Queue1.hpp”

typedef Queue1<Integer> QueueOfInteger;

int main(int argc, char* argv[])
{
   QueueOfInteger q1;

   cout << q1.length();
}
```

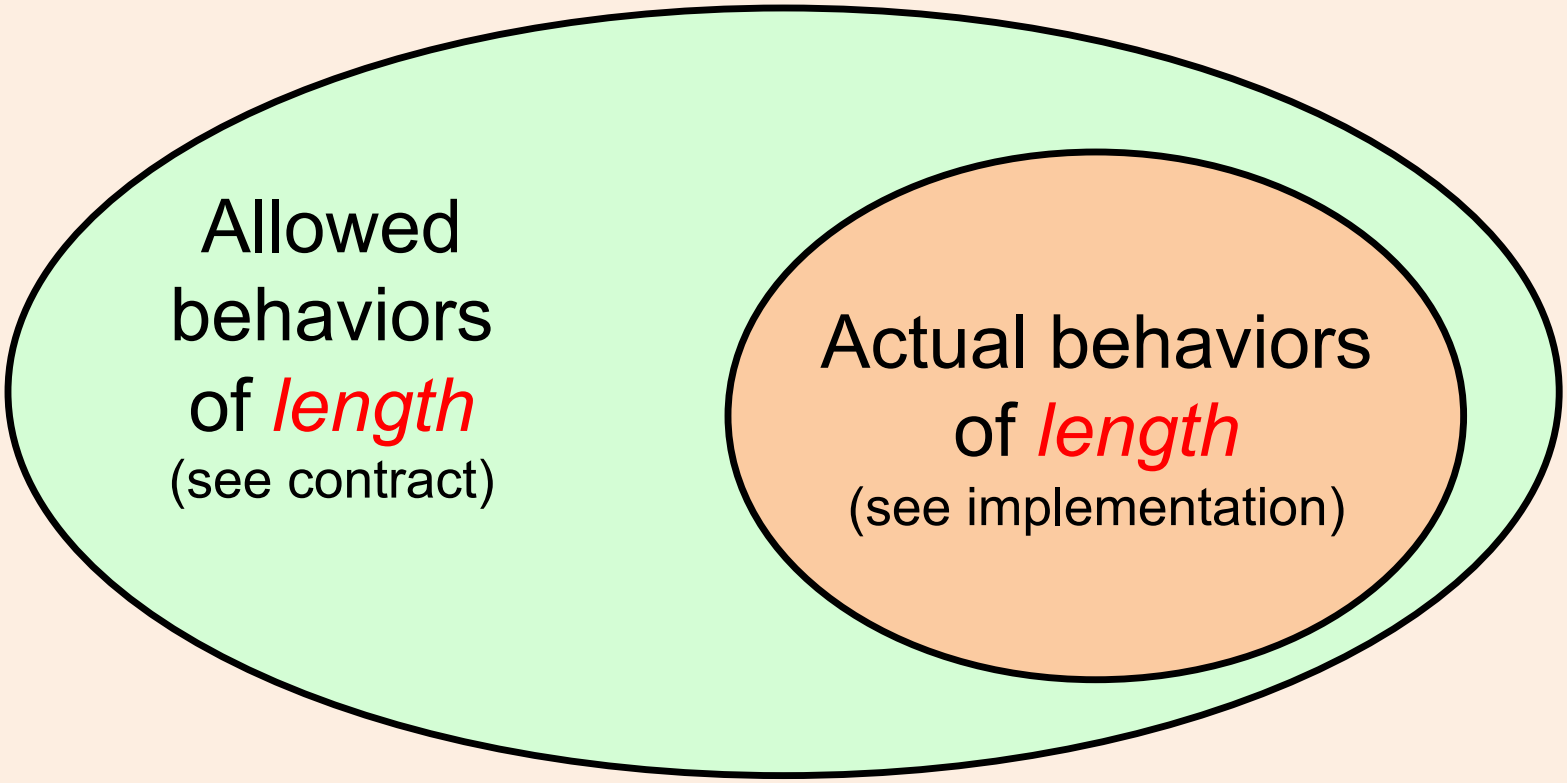
Ensures clause of Queue’s constructor initializes q1 = <>

Example: Client of Queue

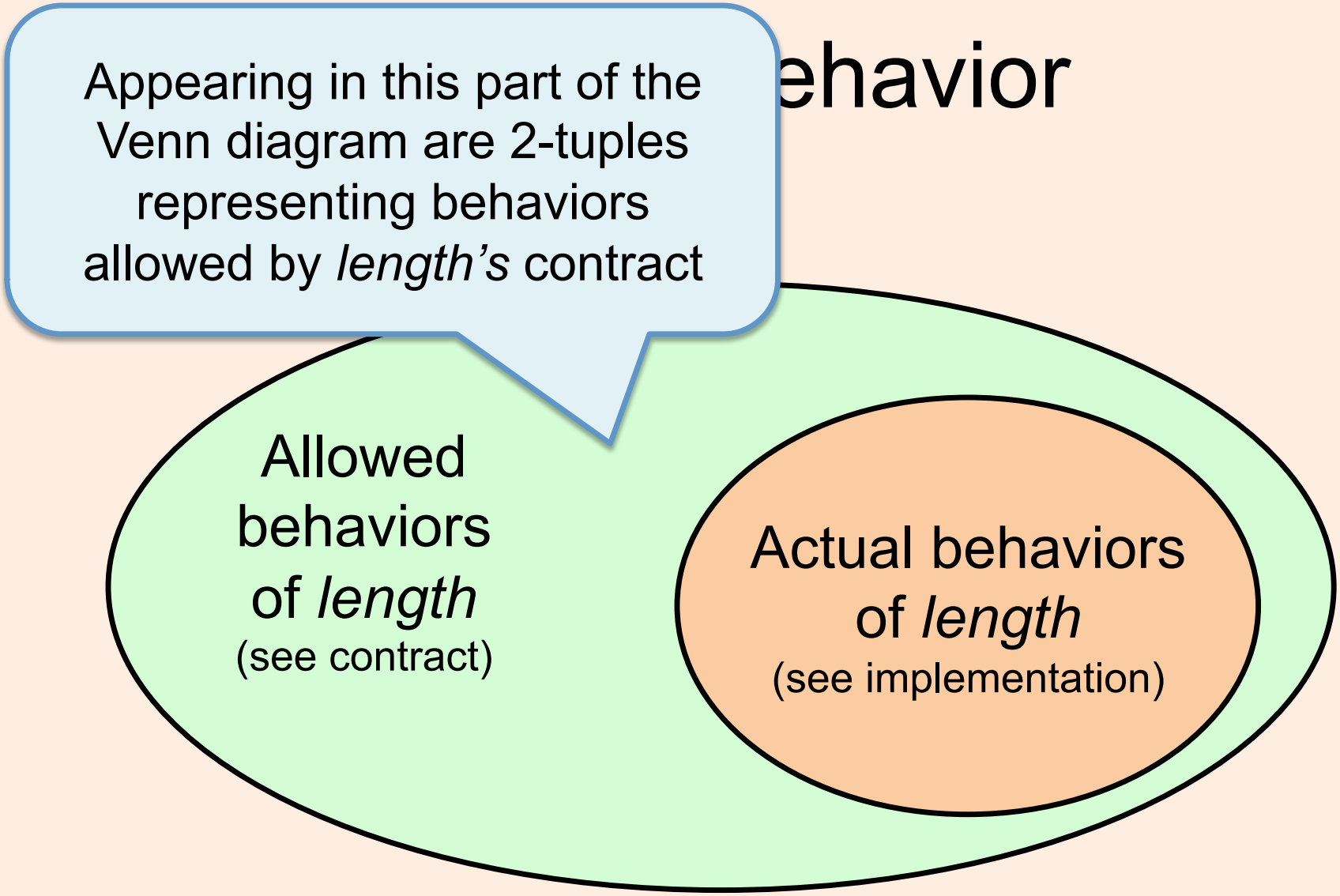
```
#include "Wrapper.h”
#include "Queue\Queue1.hpp”

typedef Queue1<Integer> QueueOfInteger;

int main(int argc, char* argv[])
{
   QueueOfInteger q1;

   cout << q1.length();
}
```

Ensures clause of Queue’s length is:
length = |q1|

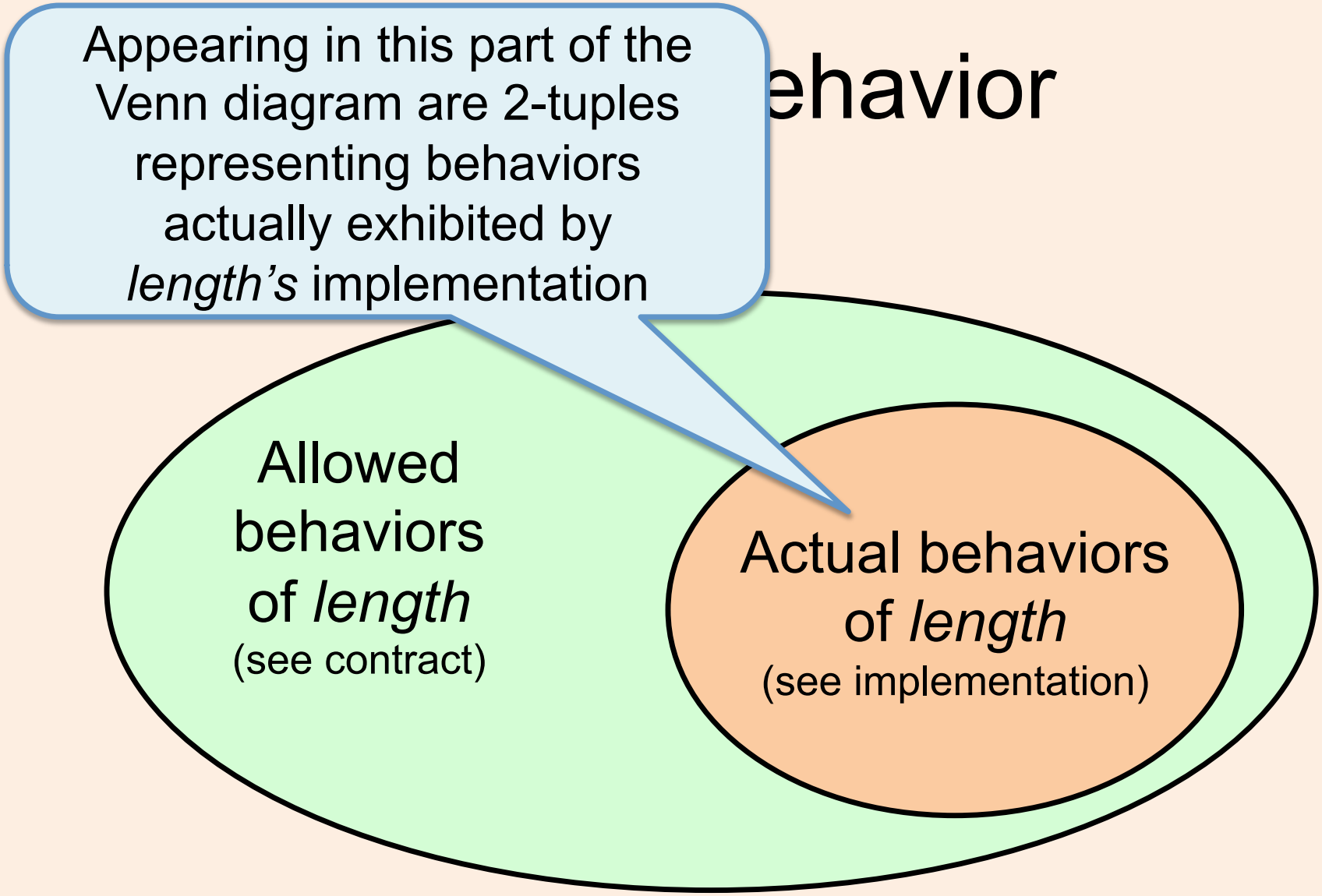
Example: *length's* Behavior



ehavior


Appearing in this part of the Venn diagram are 2-tuples representing behaviors allowed by *length's* contract
Allowed behaviors of *length* (see contract)
Actual behaviors of *length* (see implementation)

ehavior


Appearing in this part of the Venn diagram are 2-tuples representing behaviors actually exhibited by *length's* implementation
Allowed behaviors of *length* (see contract)
Actual behaviors of *length* (see implementation)

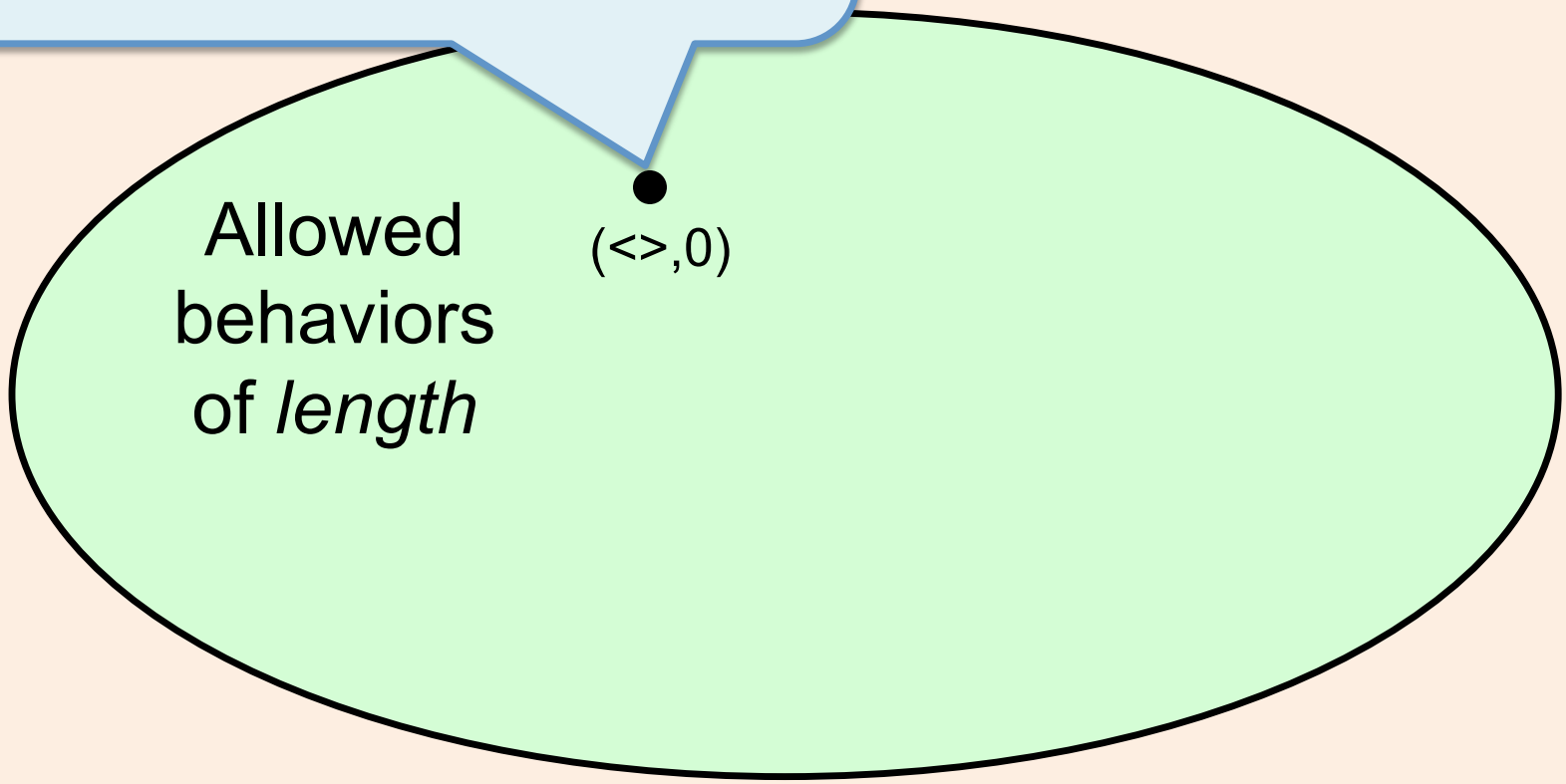
ehavior

For the moment, let's focus on the behaviors allowed by *length's* contract

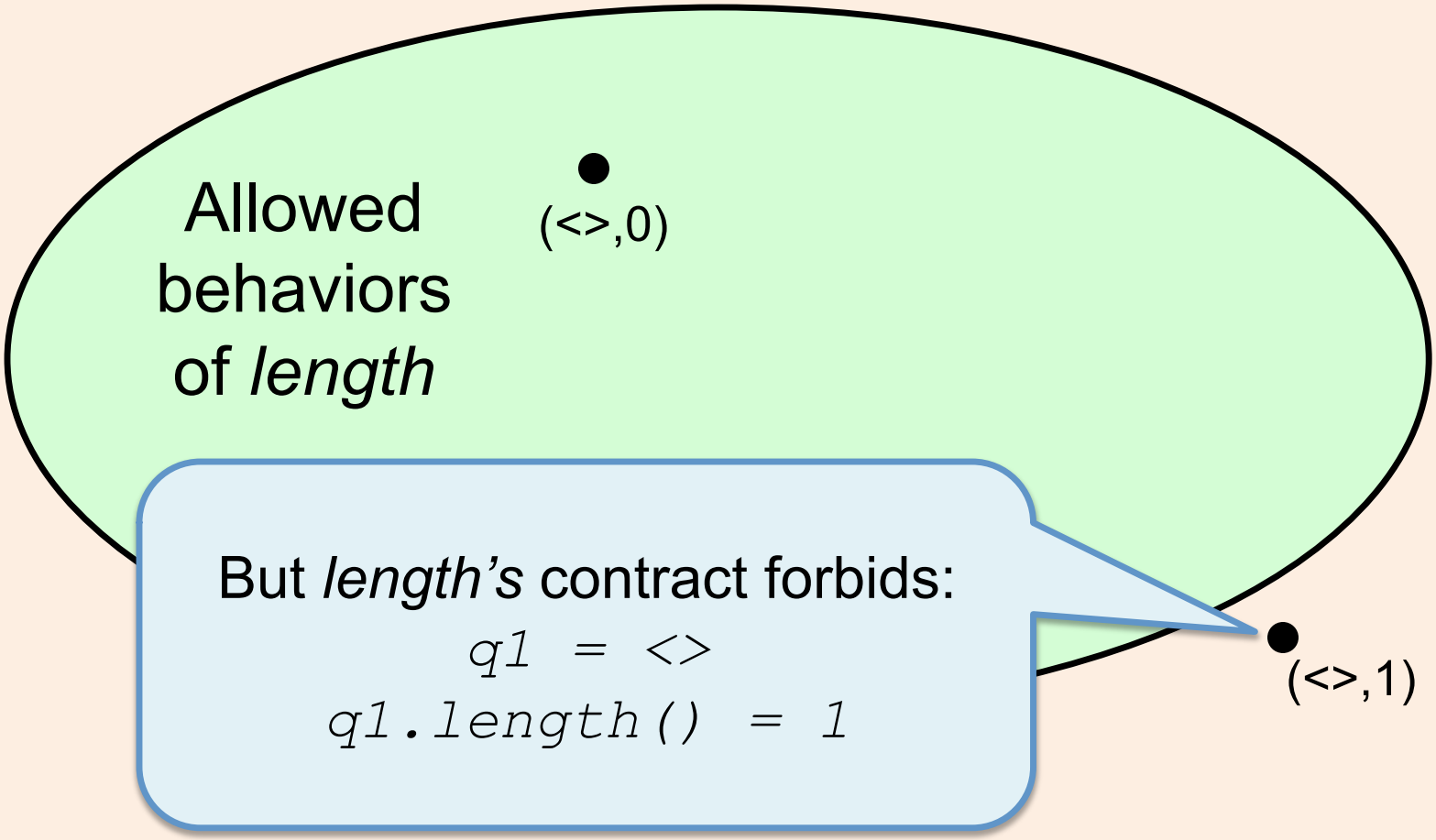
Allowed behaviors of *length*
(see contract)

ehavior

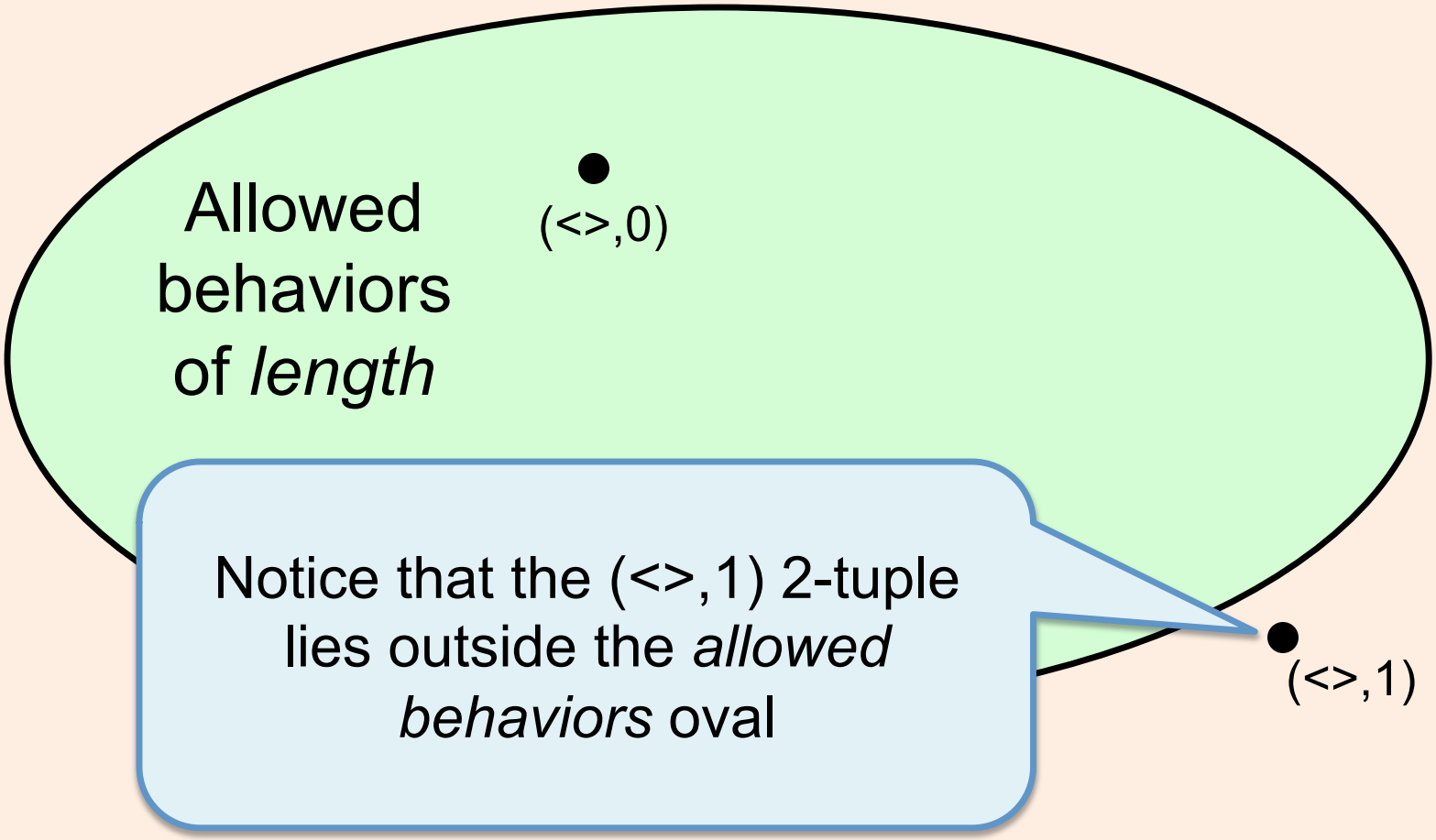
By *length's* contract:
if `q1 = <>`
then `q1.length() = 0`



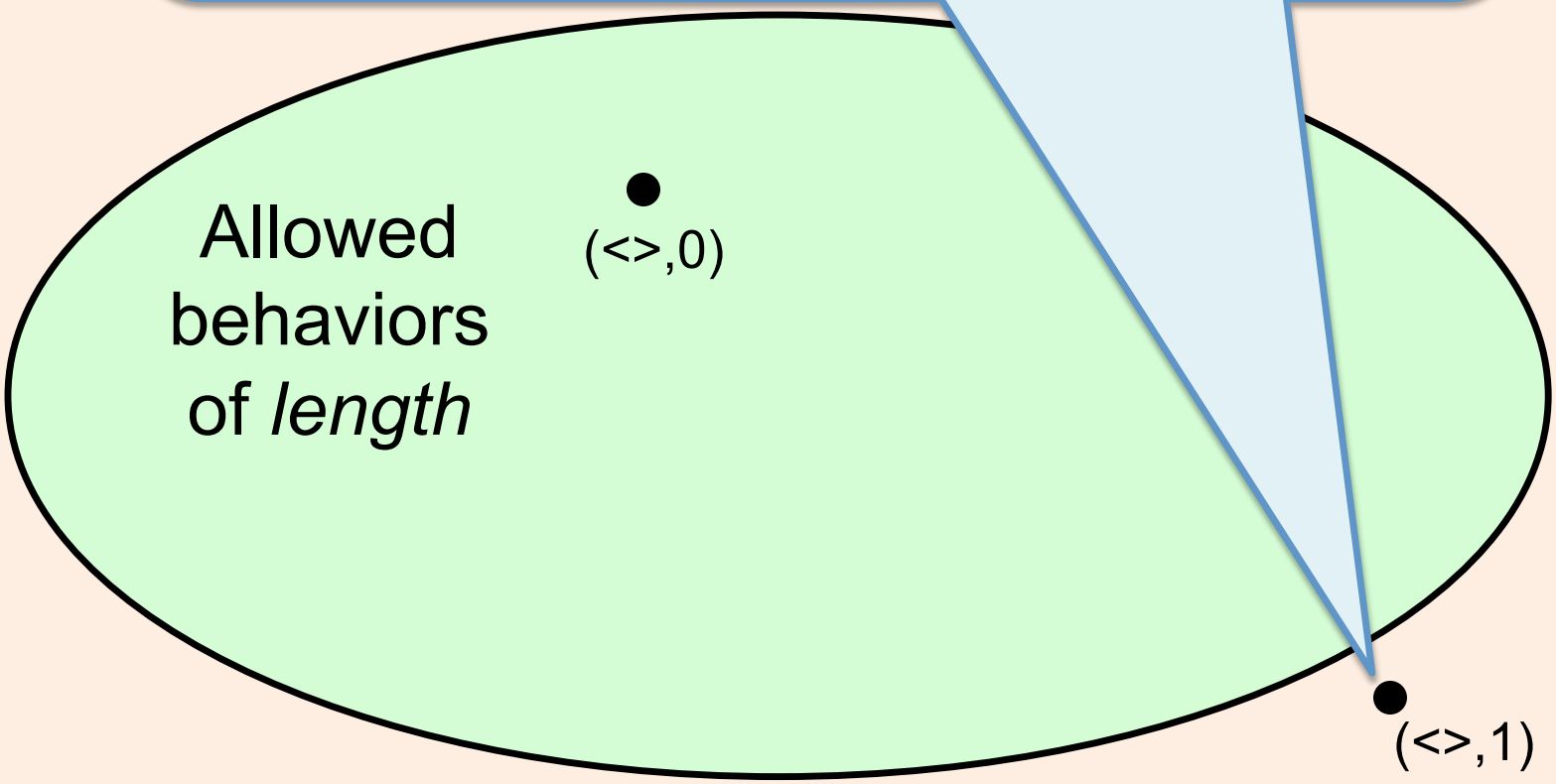
length's Behavior



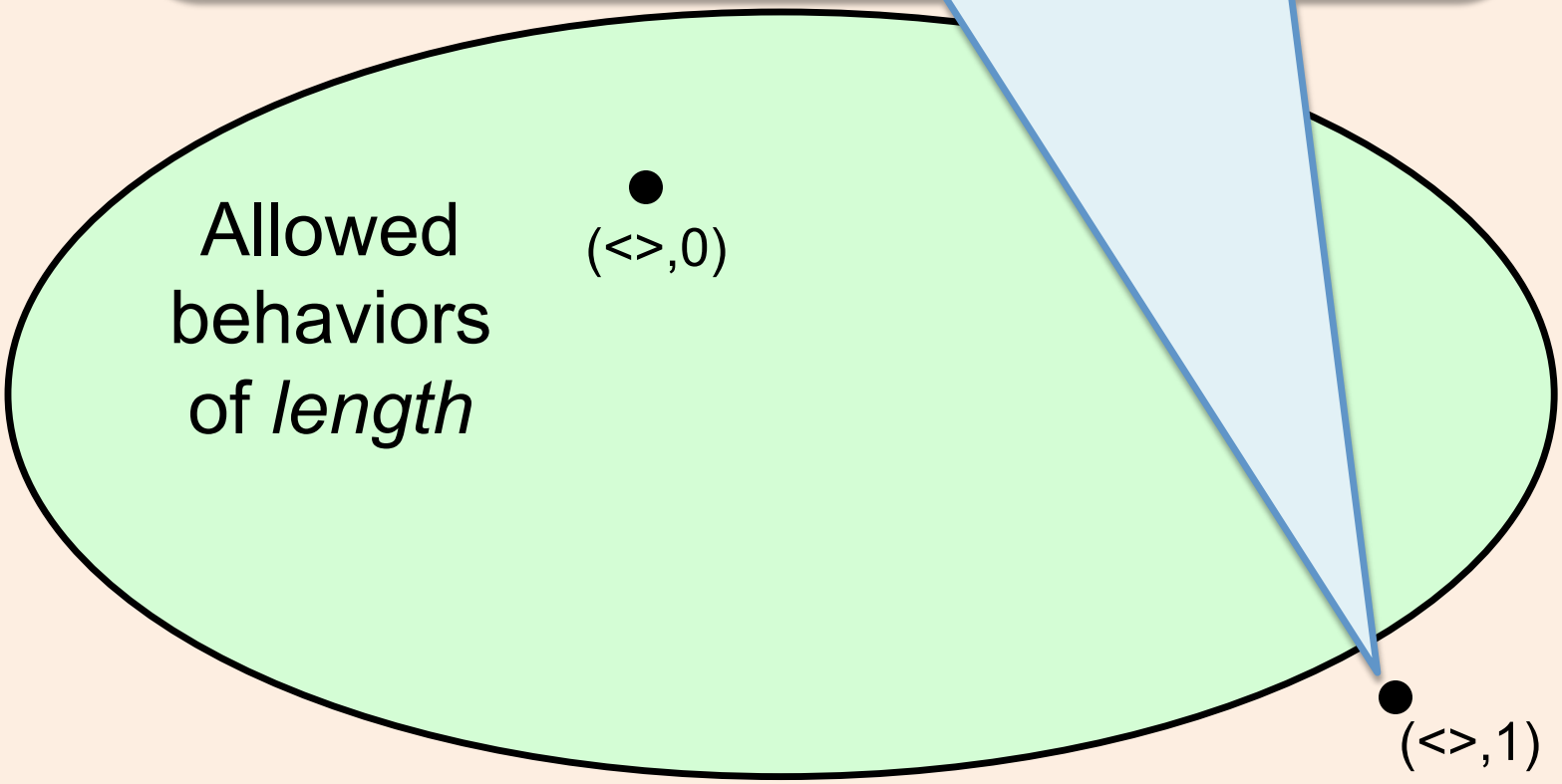
length's Behavior



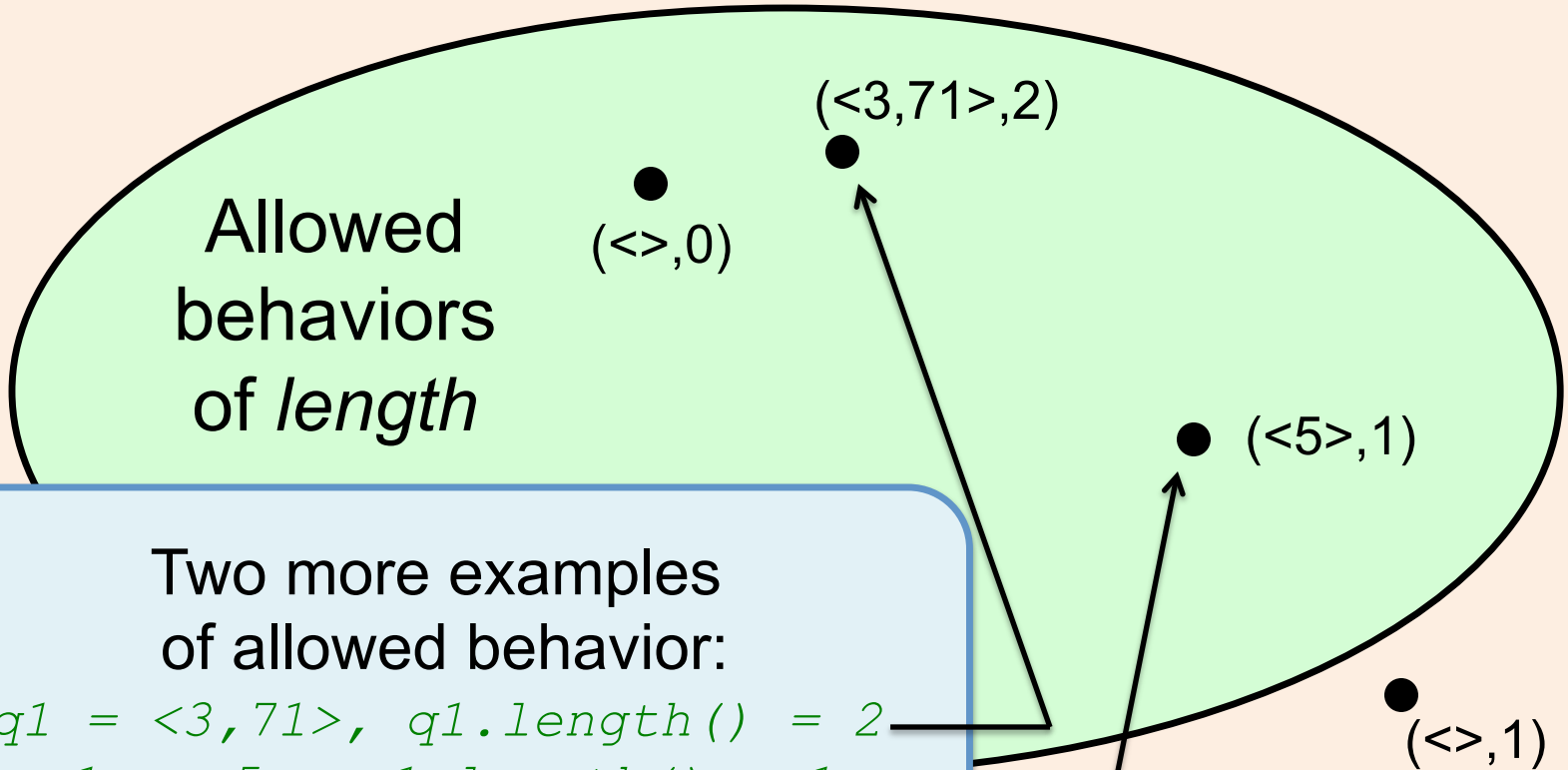
Each point outside the oval is a ***legal input*** with a corresponding ***not allowable result***. These are also represented as 2-tuples: (legal input, not allowable result)



We call these *counterexamples*



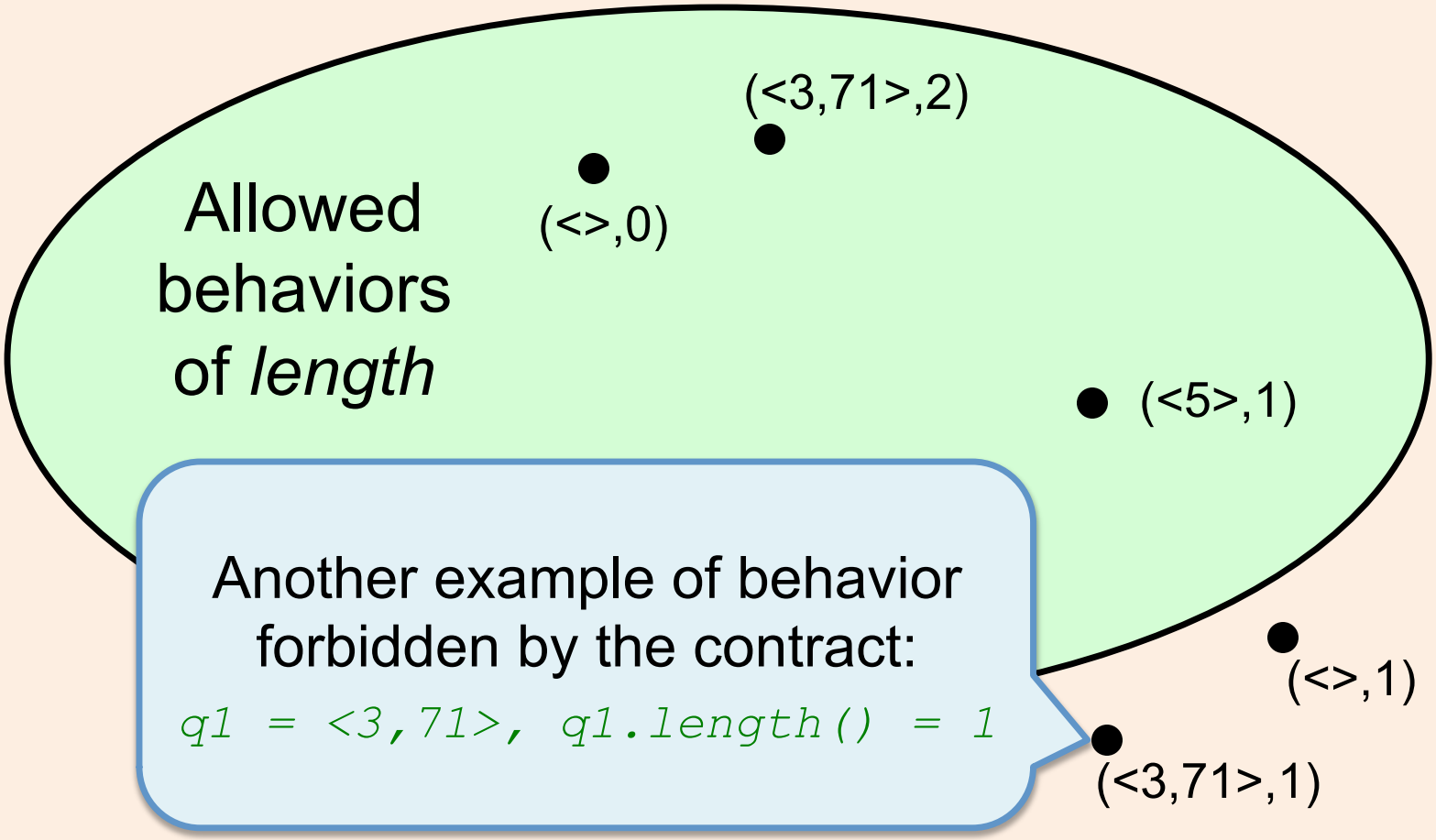
length's Behavior



Two more examples of allowed behavior:

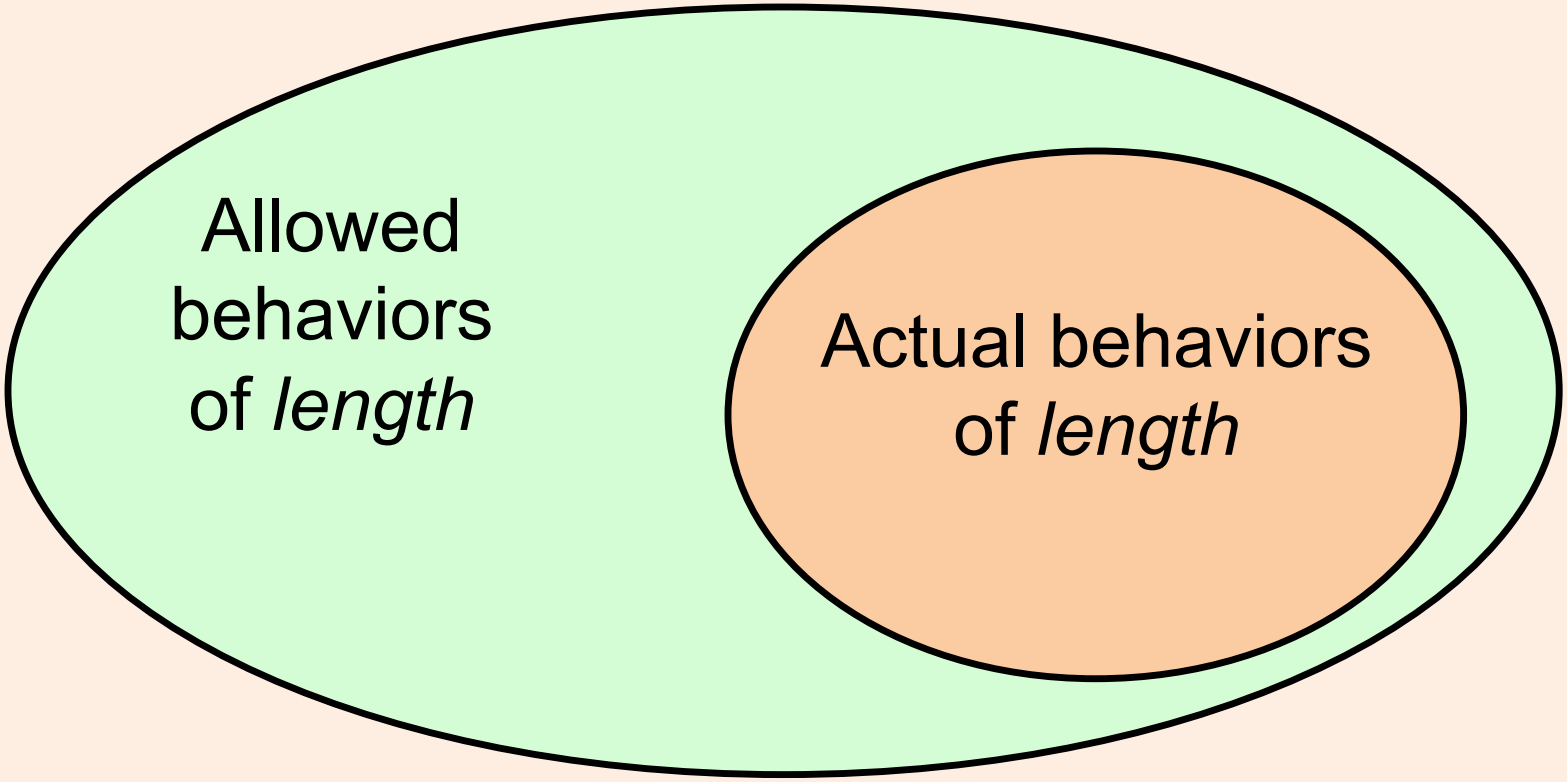
```
q1 = <3,71>, q1.length() = 2
q1 = <5>, q1.length() = 1
```

length's Behavior



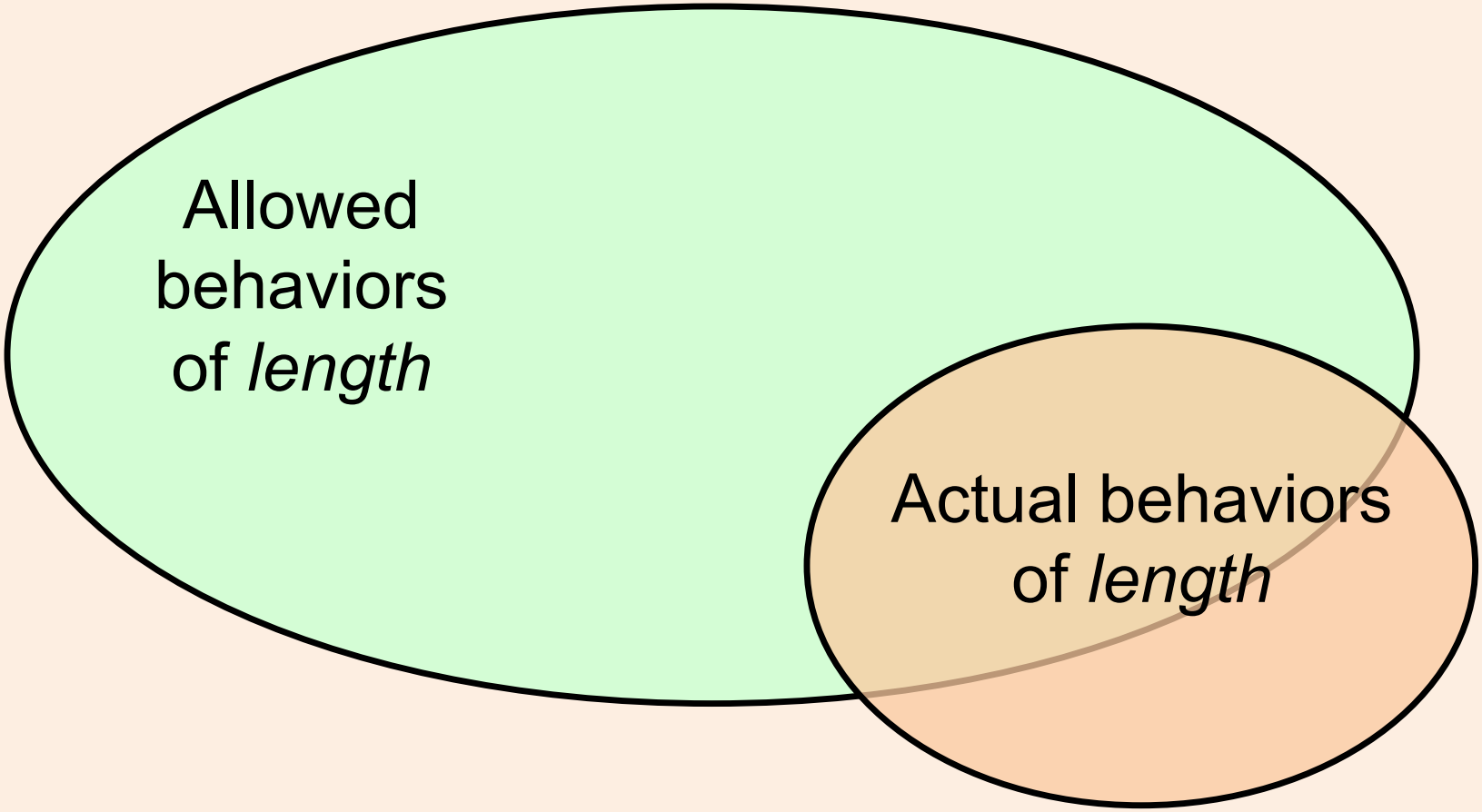
Definition of Correctness

- The implementation is ***correct*** if *actual* is a subset of *allowed*.



Definition of Defective Code

- The implementation is ***incorrect*** (or defective) if *actual* is not a subset of *allowed*.



A Possible Implementation of Queue's *length*

```
Integer length(void)
{
  return 1;
} // length
```

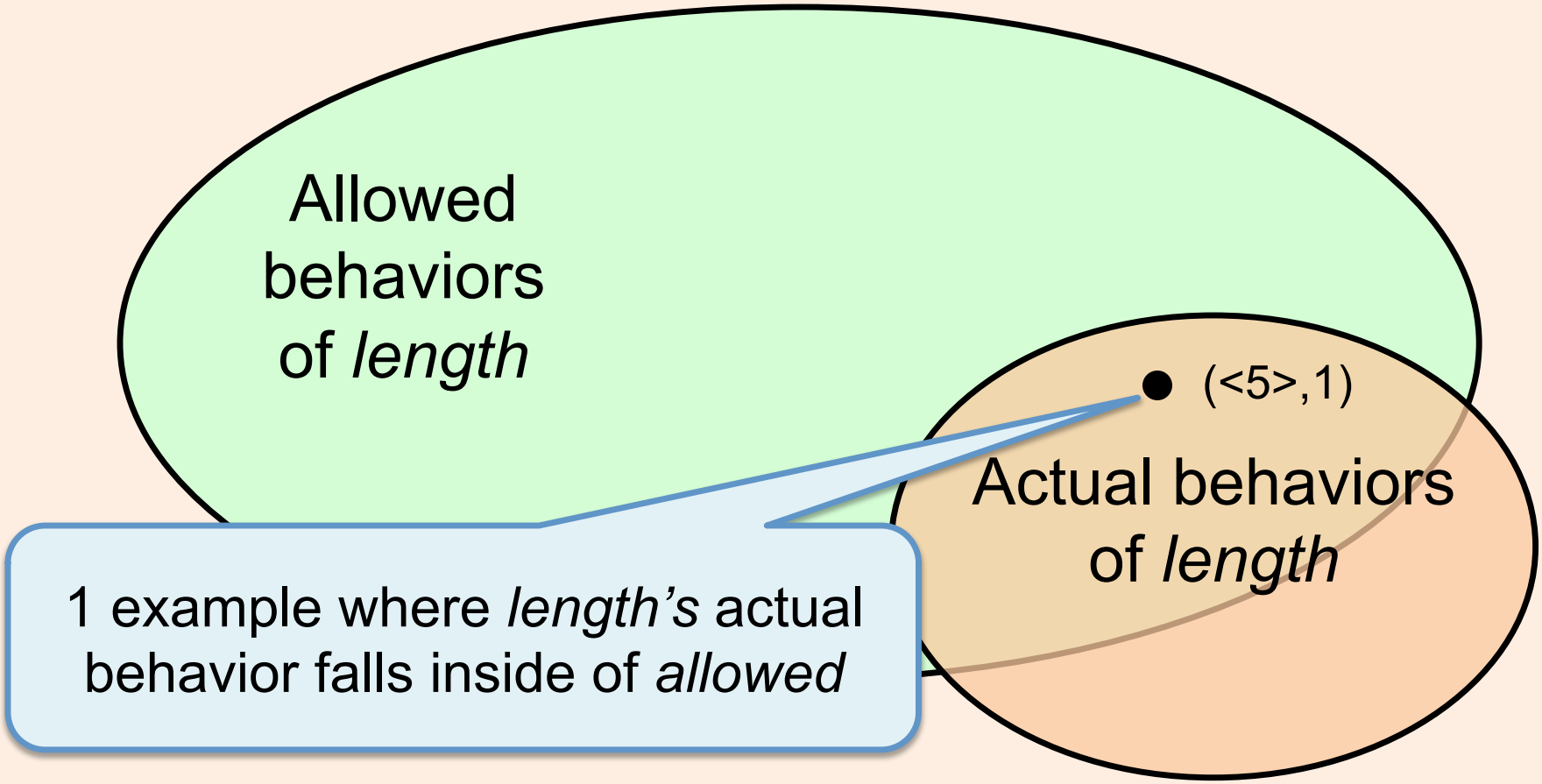
A Possible Implementation of Queue's *length*

```
Integer length(void)
{
  return 1;
} // length
```

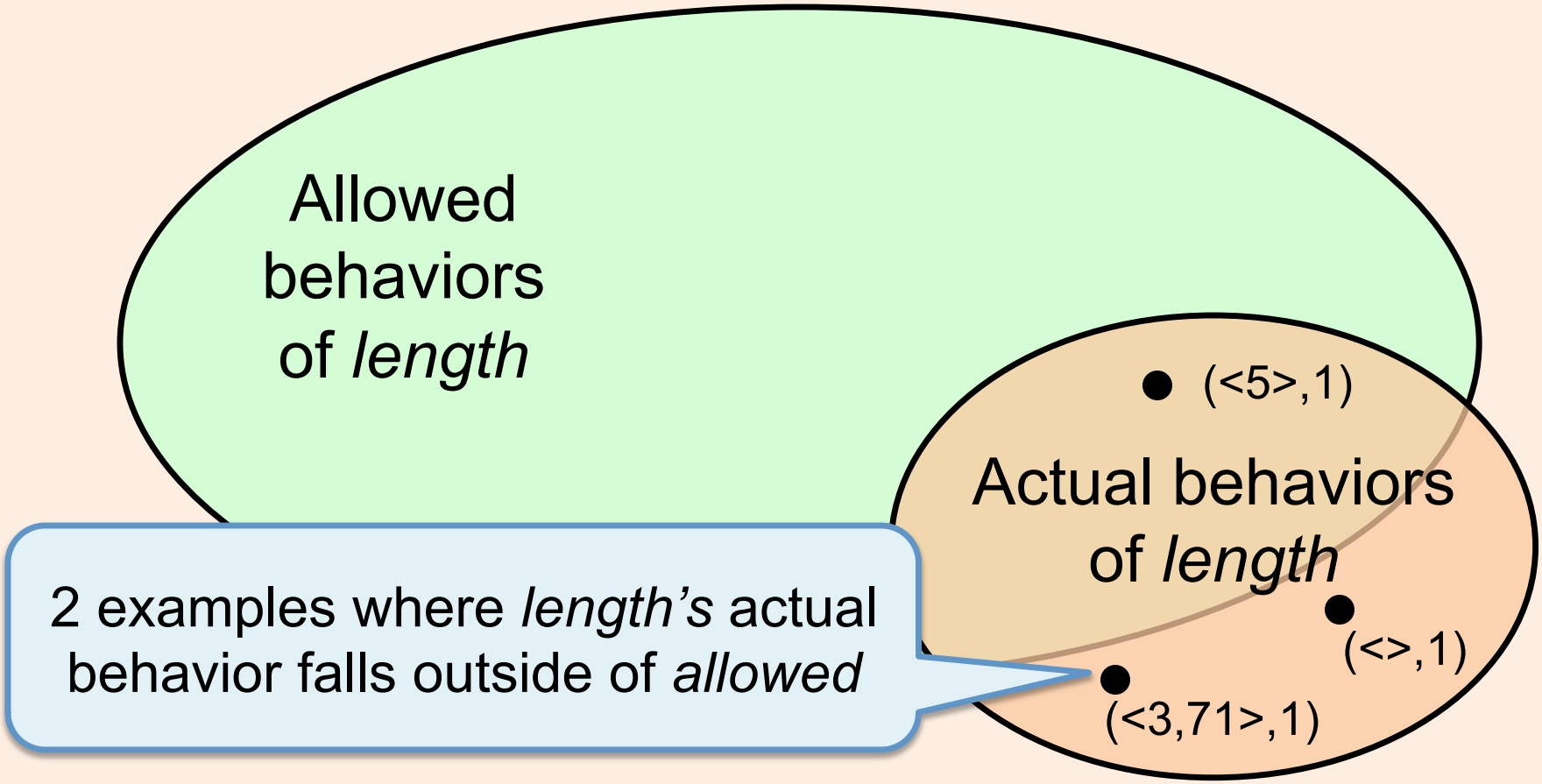
This implementation always returns 1

Is this implementation correct?

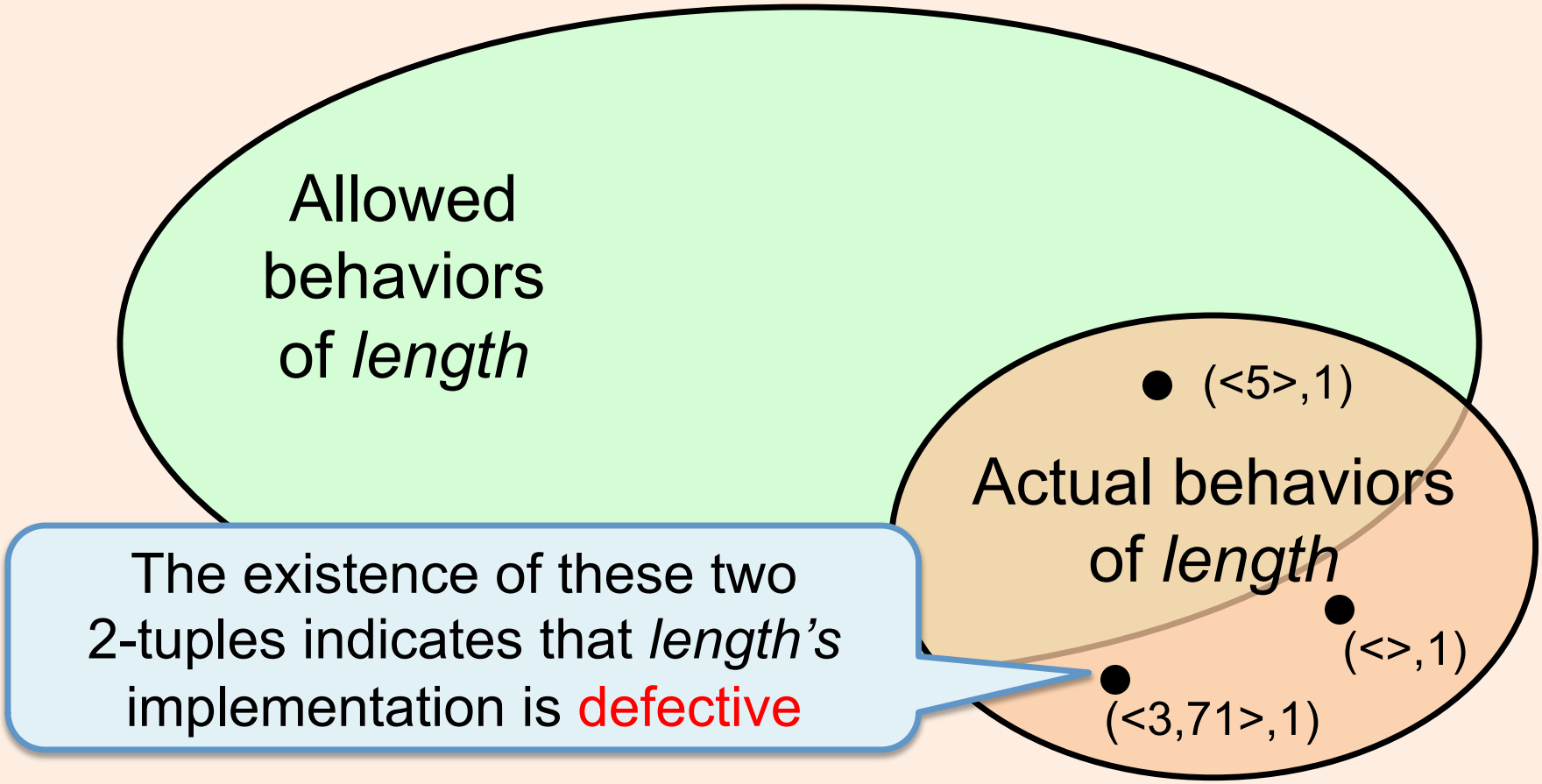
Defective Implementation



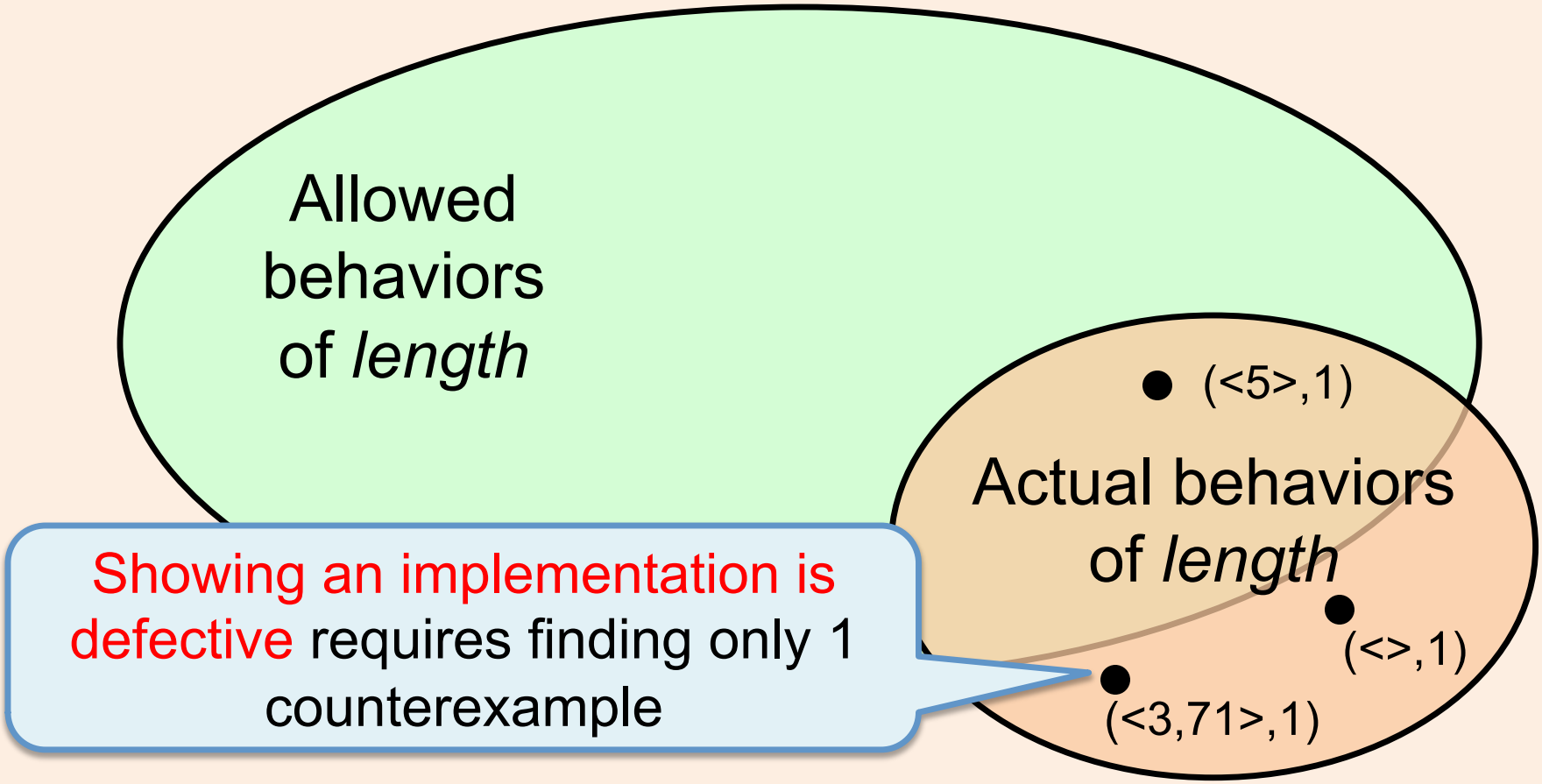
Defective Implementation



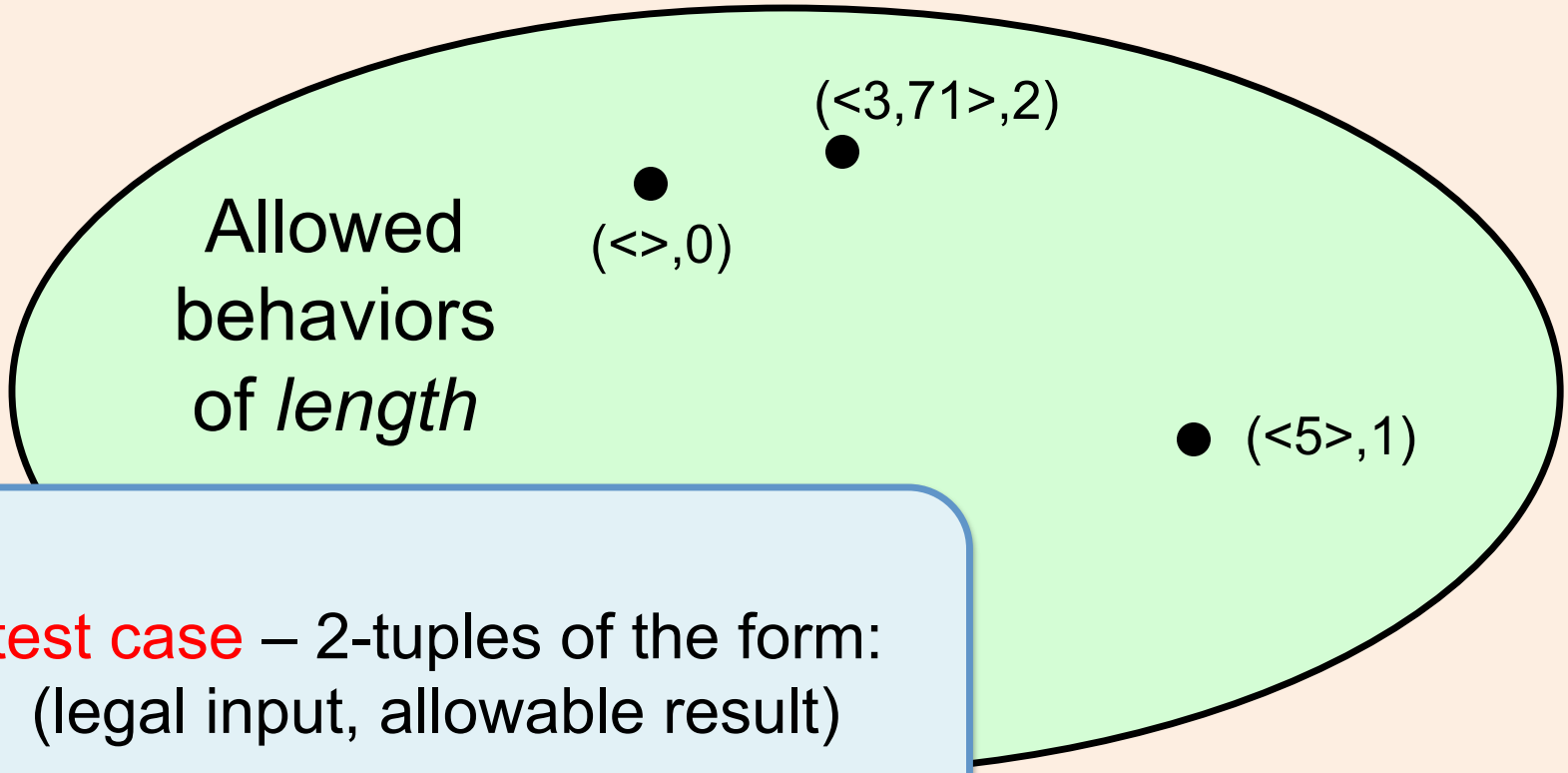
Defective Implementation



Defective Implementation

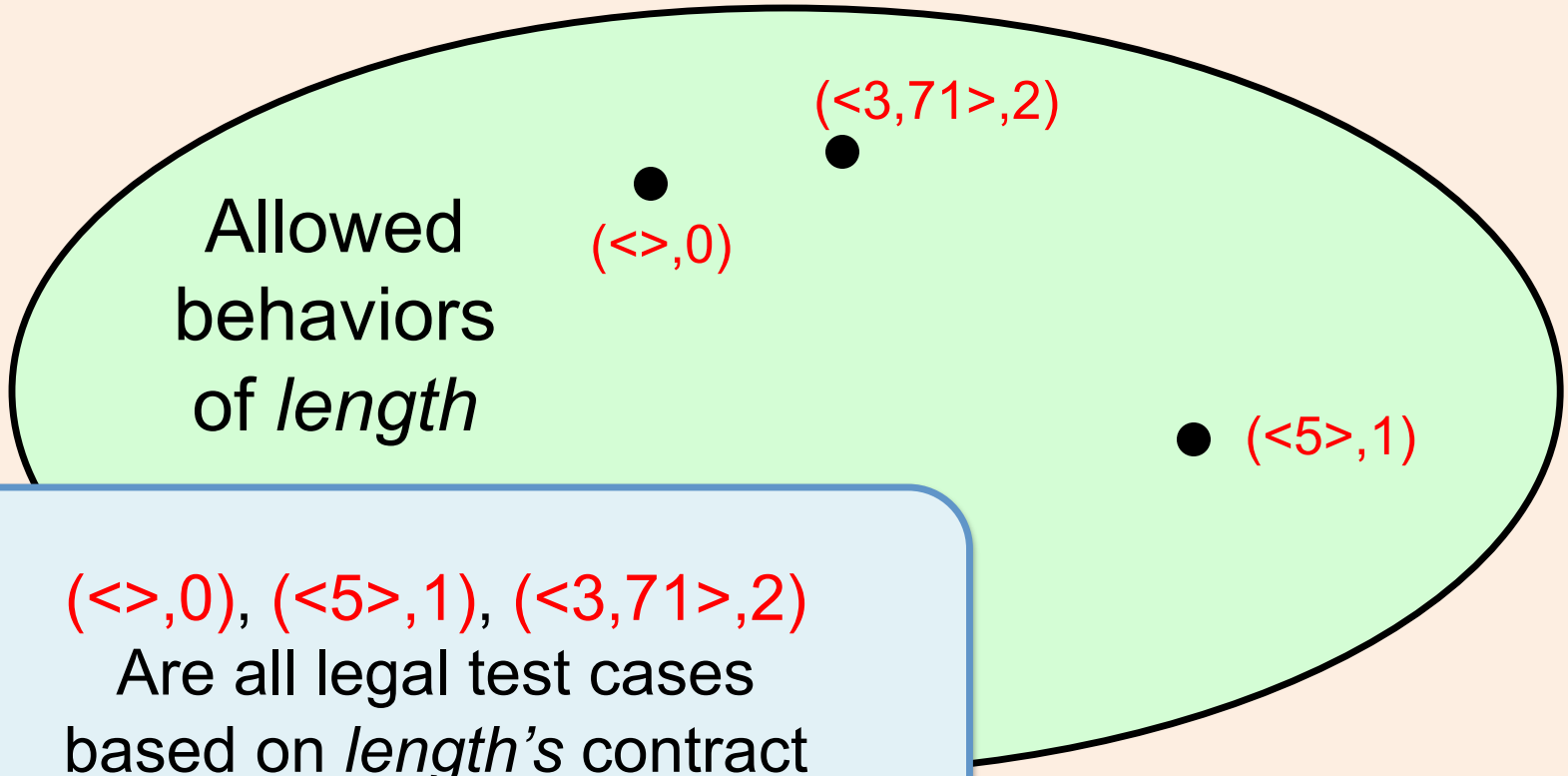


Test Cases



test case – 2-tuples of the form:
(legal input, allowable result)

Test Cases



(<>,0), (<5>,1), (<3,71>,2)
Are all legal test cases based on *length's* contract

Testing Concepts

- ***actual behavior space*** – consists of all 2-tuples of the form (legal input, allowable result) and (legal input, not allowable result)
- ***allowable result*** – an output from an operation's implementation satisfying the operation's ensures clause
- ***allowed behavior space*** – consists of 2-tuples of the form (legal input, allowable result)
- ***correct implementation*** – when the actual behavior space is a subset of the allowed behavior space
- ***counterexample*** – consists of a 2-tuple of the form (legal input, not allowable result)
- ***defective implementation*** – when the actual behavior space is not a subset of the allowed behavior space
- ***integration testing*** – when a subsystem comprising multiple classes are under test
- ***legal input*** – an input to an operation satisfying the operation's requires clause
- ***not allowable result*** – an output from an operation's implementation (on a legal input) that does not satisfy the operation's ensures clause
- ***showing an implementation is defective*** – requires finding only one counterexample
- ***system testing*** – when the entire end-user system is under test
- ***test case*** – consists of a 2-tuple of the form (legal input, allowable result) based on the unit under test's contract
- ***unit testing*** – testing one operation at a time
- ***unit under test*** – the operation being tested

Credits

- These slides were adapted from slides obtained from Dr. Bruce W. Weide and Dr. Paolo Bucci.

- Drs. Weide & Bucci are members of the Resolve/Reusable Software Research Group (RSRG) which is part of the Software Engineering Group in the Department of Computer Science and Engineering at The Ohio State University.