Imperative Programming 2:
Polymorphism 3

Hongseok Yang
University of Oxford

Polymorphism 1, 2

- Principles.
- Two kinds of polymorphism in Scala.
  1. Ad-hoc poly. -- Overloading.
  2. Parametric poly. -- Generics & subtyping.

Polymorphism 3

- Example of generics and variance.
- Persistent covariant generic queue.

Learning outcome

- Can develop covariant generic classes.
- Can explain how mutable states interact with variance of type parameters.
- Can implement an efficient persistent queue.

Persistent data structure

- Immutable.
- Updates are done by creating new objects.
- Very common in Scala -- Default List, Map, Set in Scala are all immutable.
- More efficient than you might think.

Persistent data structure

hyang — java — 56×19

```
scala> val l1 = List(0)
l1: List[Int] = List(0)

scala> val l2 = 1::l1
l2: List[Int] = List(1, 0)

scala> val l3 = 2::l2
l3: List[Int] = List(2, 1, 0)

scala> l1
res2: List[Int] = List(0)

scala> l3.tail
res3: List[Int] = List(1, 0)

scala> l3.head
res4: Int = 2

scala>
```

Persistent data structure

hyang — java — 58×21

```
import collection.immutable.Queue

scala> val q1 = Queue(0)
q1: scala.collection.immutable.Queue[Int] = Queue(0)

scala> val q2 = q1.enqueue(1)
q2: scala.collection.immutable.Queue[Int] = Queue(0, 1)

scala> val q3 = q2.enqueue(2)
q3: scala.collection.immutable.Queue[Int] = Queue(0, 1, 2)

scala> q1
res5: scala.collection.immutable.Queue[Int] = Queue(0)

scala> q1.tail
res6: scala.collection.immutable.Queue[Int] = Queue()

scala> q3.head
res7: Int = 0

scala>
```

Persistent data structure

- Immutable.
- Updates are done by creating new objects.
- Very common in Scala -- Default List, Map, Set in Scala are all immutable.
- More efficient than you might think.

[Q] Why do you think this is the case?

Exercise: Implement a persistent generic Queue

```
class Queue[T] ... {
  def head ...
  def tail ...
  def enqueue ...
}
```

Hint

```
class IQueue(private val contents: List[Int]) {
  def head: Int = contents.head
  def tail: IQueue = new IQueue(contents.tail)
  def enqueue(x: Int): IQueue = new IQueue(contents :+ x)
}
```

Exercise: Implement a persistent generic Queue

```
class Queue[T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue[T](contents.tail)
  def enqueue(x: T): Queue[T] = new Queue[T](contents :+ x)
}
```

Hint

```
class IQueue(private val contents: List[Int]) {
  def head: Int = contents.head
  def tail: IQueue = new IQueue(contents.tail)
  def enqueue(x: Int): IQueue = new IQueue(contents :+ x)
}
```

Exercise: Implement a persistent generic Queue

```
class Queue[T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue(x: T): Queue[T] = new Queue(contents :+ x)
}
```

Hint

```
class IQueue(private val contents: List[Int]) {
  def head: Int = contents.head
  def tail: IQueue = new IQueue(contents.tail)
  def enqueue(x: Int): IQueue = new IQueue(contents :+ x)
}
```

Exercise: Implement a persistent generic Queue

```
class Queue[T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue(x: T): Queue[T] = new Queue(contents :+ x)
}
```

Hint

```
class IQueue(private val contents: List[Int]) {
  def head: Int = contents.head
  def tail: IQueue = new IQueue(contents.tail)
  def enqueue(x: Int): IQueue = new IQueue(contents :+ x)
}
```

Almost the same because of Scala's generic libraries.

Exercise: Implement a persistent generic Queue

```
class Queue[T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue(x: T): Queue[T] = new Queue(contents :+ x)
}
```

[Q1] What should we do to make Queue covariant?

Exercise: Implement a persistent generic Queue

```
class Queue[+T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue[U >: T](x: U): Queue[U] =
    new Queue(contents :+ x)
}
```

[Q1] What should we do to make Queue covariant?

Exercise: Implement a persistent generic Queue

```
class Queue[+T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue[U >: T](x: U): Queue[U] =
    new Queue(contents :+ x)
  def filter(f: T=>Boolean): Queue[T] =
    new Queue(contents filter f)
  def mapTwice(f: T=>T): Queue[T] =
    new Queue((contents map f) map f)
}
```

[Q1] What should we do to make Queue covariant?

[Q2] Can we add filter? What about mapTwice?

Exercise: Implement a persistent generic Queue

```
class Queue[+T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue[U >: T](x: U): Queue[U] =
    new Queue(contents :+ x)
  def filter(f: T=>Boolean): Queue[T] =
    new Queue(contents filter f)
  def mapTwice(f: T=>T): Queue[T] =
    new Queue((contents map f) map f)
}
```

Yes

No

[Q1] What should we do to make Queue covariant?

[Q2] Can we add filter? What about mapTwice?

Exercise: Implement a persistent generic Queue

```
class Queue[+T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue[U >: T](x: U): Queue[U] =
    new Queue(contents :+ x)
  def filter(f: T=>Boolean): Queue[T] =
    new Queue(contents filter f)
  def mapTwice[U >: T](f: U=>U): Queue[U] =
    new Queue((contents map f) map f)
}
```

[Q1] What should we do to make Queue covariant?

[Q2] Can we add filter? What about mapTwice?

Exercise: Implement a persistent generic Queue

```
class Queue[+T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue[U >: T](x: U): Queue[U] =
    new Queue(contents :+ x)
  def filter(f: T=>Boolean): Queue[T] =
    new Queue(contents filter f)
  def mapTwice[U >: T](f: U=>U): Queue[U] =
    new Queue((contents map f) map f)
}
```

[Q1] What should we do to make Queue covariant?

[Q2] Can we add filter? What about mapTwice?

[Q3] The enqueue operation is very slow. How can we improve it?

Exercise: Implement a persistent generic Queue

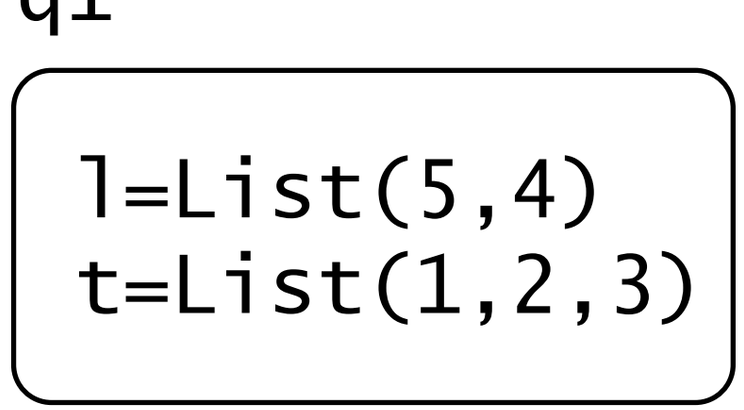
```
class Queue[+T](private val contents: List[T]) {
  def head: T = contents.head
  def tail: Queue[T] = new Queue(contents.tail)
  def enqueue[U >: T](x: U): Queue[U] =
    new Queue(contents :+ x)


}
```

[Q1] What should we do to make Queue covariant?
[Q2] Can we add filter? What about mapTwice?
[Q3] The enqueue operation is very slow. How can we improve it?

Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`



Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```

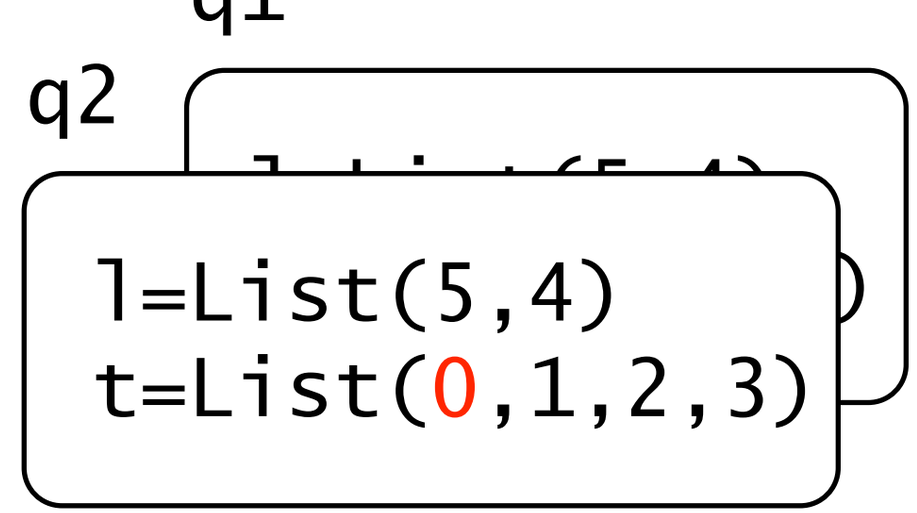
q1

```
l=List(5,4)
t=List(1,2,3)
```

Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

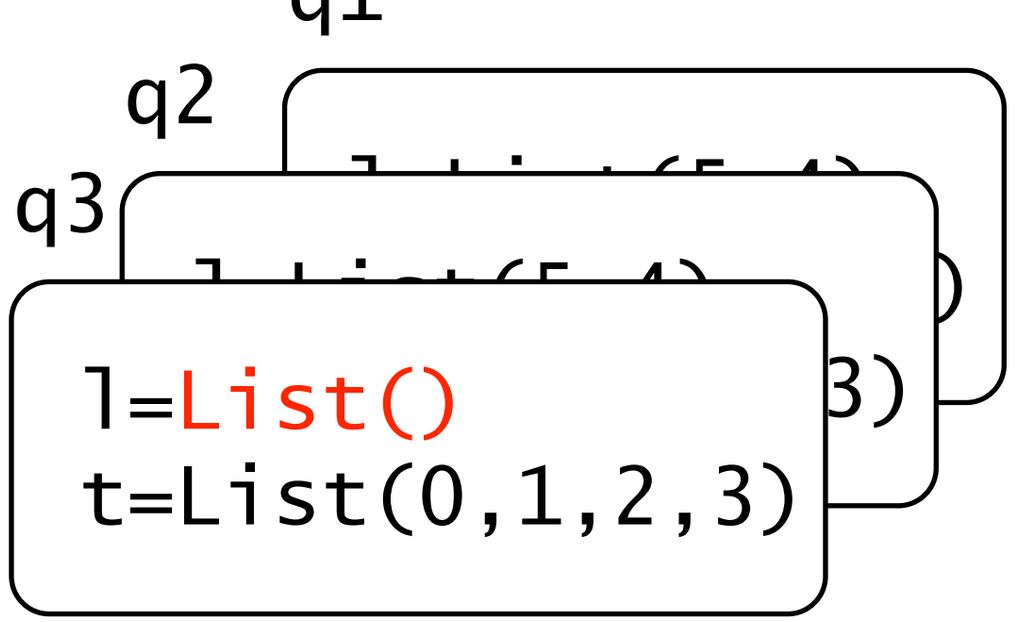
```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```



Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

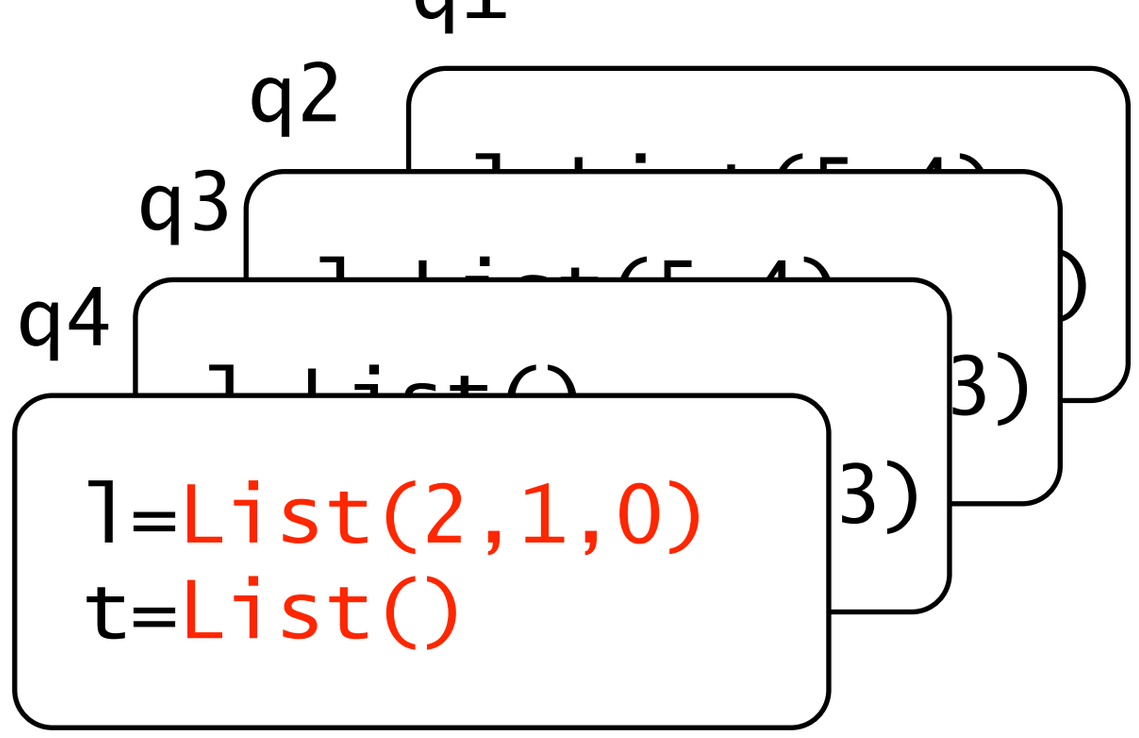
```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```



Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

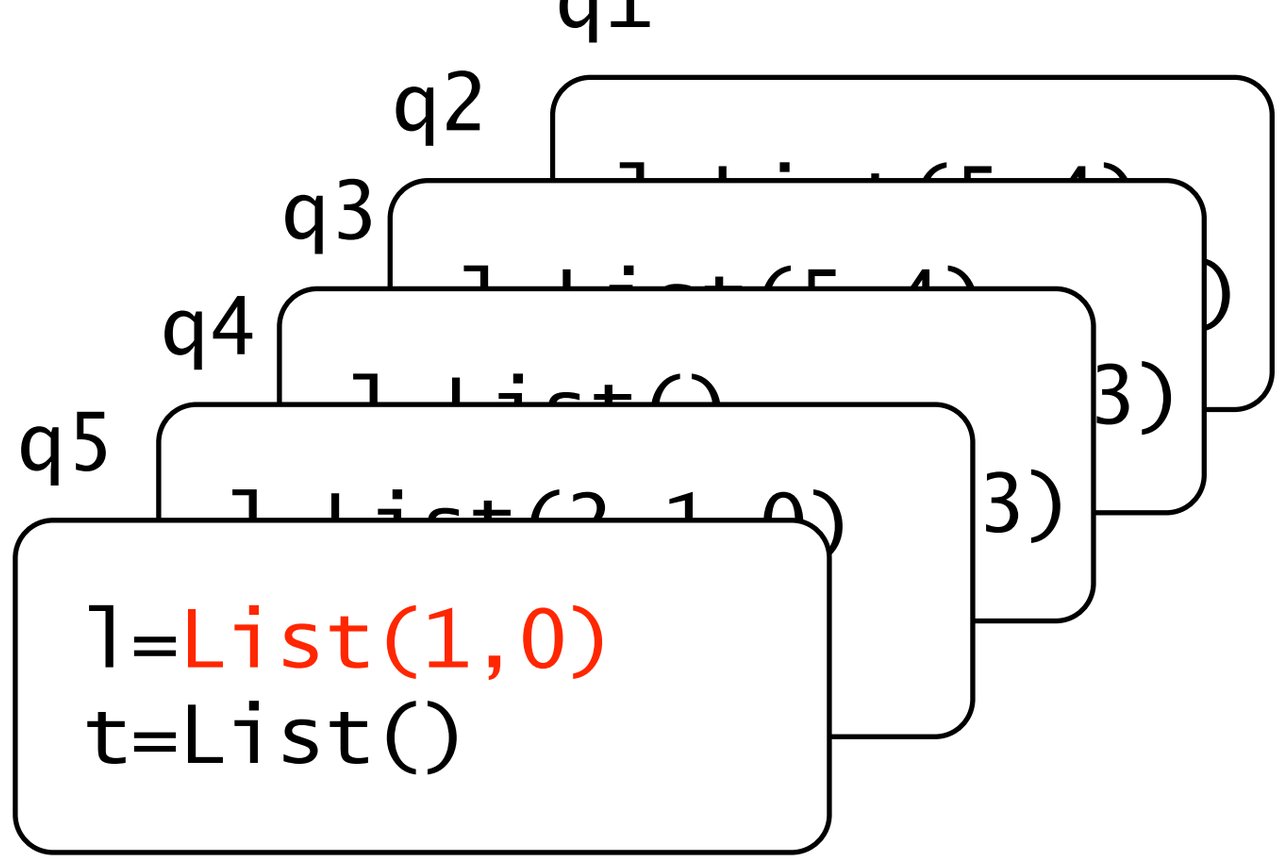
```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```



Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

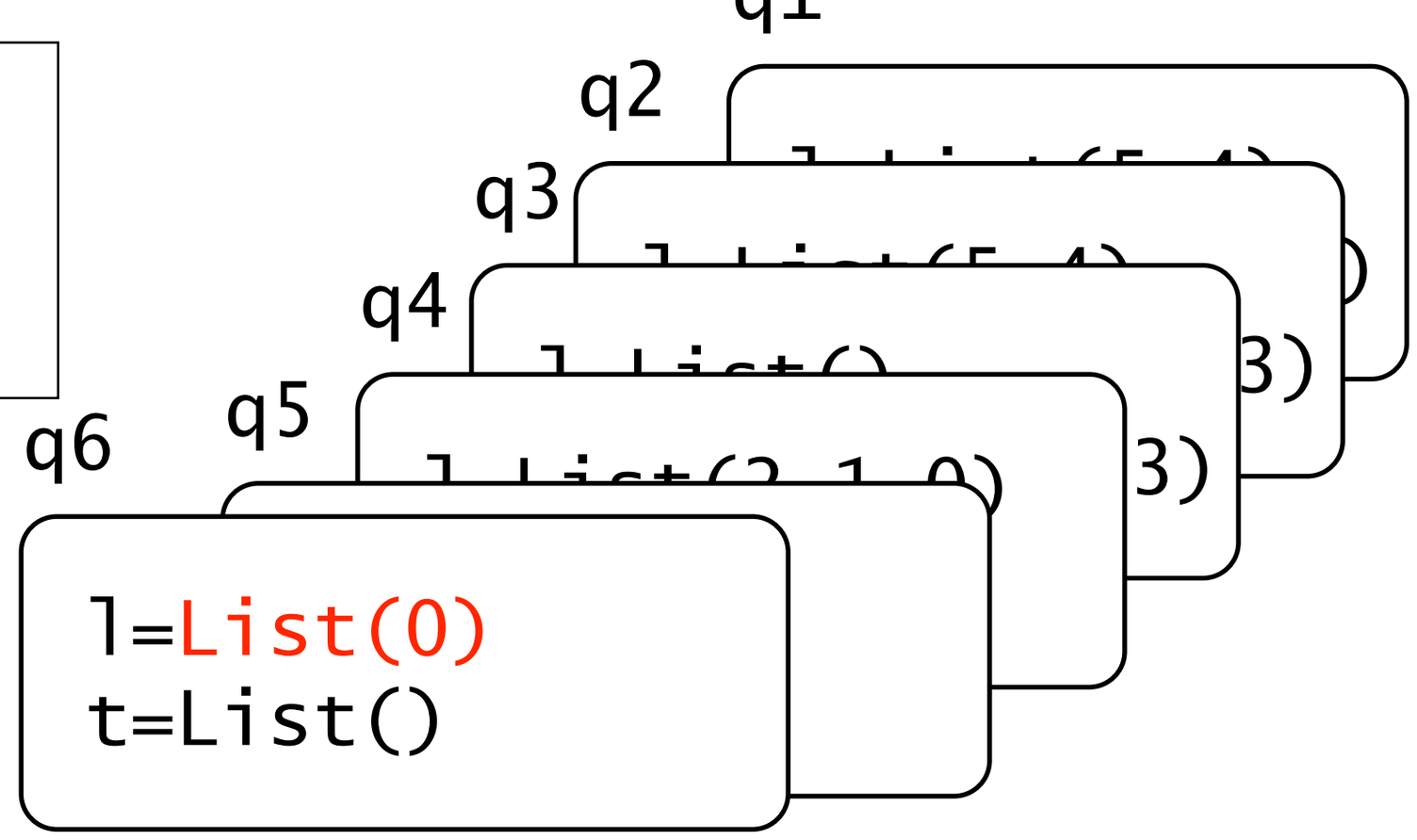
```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```



Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```

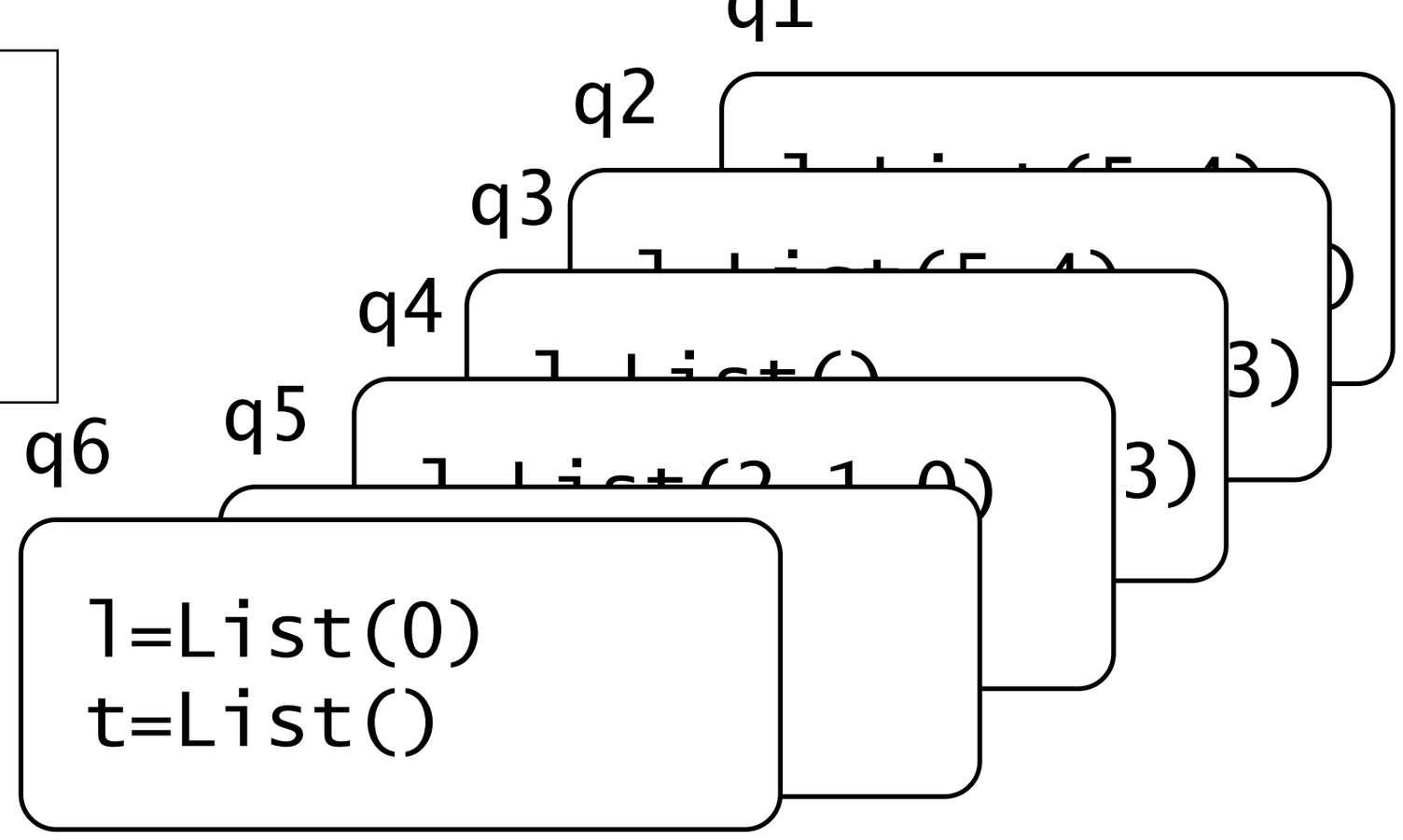


Key idea: Maintaining two lists

- Use two lists `l` and `t` for storing values.
- Invariant: `contents == l ++ t.reverse`

```
val q2 = q1.enqueue(0)
val q3 = q2.tail.tail
val q4 = q3.tail
val q5 = q4.tail
val q6 = q5.tail
```

Many tail operations benefit from a single reverse.



Implementing the idea

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {



  def head: T = ...
  def tail: FQueue[T] = ...



  def enqueue[U >: T](x: U): FQueue[U] = ...

}
```

Implementing the idea

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {



  def head: T = ...
  def tail: FQueue[T] = ...



  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x::t)
}
```

Implementing the idea

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {
  private def mirror =
    if (l.isEmpty) new FQueue(t.reverse, Nil)
    else this
  def head: T = mirror.l.head
  def tail: FQueue[T] = ...



  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x::t)
}
```

Exercise:
Implement the tail method

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {
  private def mirror =
    if (l.isEmpty) new FQueue(t.reverse, Nil)
    else this
  def head: T = mirror.l.head
  def tail: FQueue[T] = ...

  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x::t)
}
```

Exercise:
Implement the tail method

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {
  private def mirror =
    if (l.isEmpty) new FQueue(t.reverse, Nil)
    else this
  def head: T = mirror.l.head
  def tail: FQueue[T] = {
    val q = mirror
    new FQueue(q.l.tail, q.t)
  }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x::t)
}
```

Exercise:
Implement the tail method

[Q1] Any bad case still?
[Q2] What should we do?

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {
  private def mirror =
    if (l.isEmpty) new FQueue(t.reverse, Nil)
    else this
  def head: T = mirror.l.head
  def tail: FQueue[T] = {
    val q = mirror
    new FQueue(q.l.tail, q.t)
  }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x::t)
}
```

Exercise:
Implement the tail method

```
class FQueue[+T](
  private val l: List[T],
  private val t: List[T]
) {
  private def mirror =
    if (l.isEmpty) new FQueue(t.reverse, Nil)
    else this
  def head: T = mirror.l.head
  def tail: FQueue[T] = {
    val q = mirror
    new FQueue(q.l.tail, q.t)
  }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x::t)
}
```

[Q1] Any bad case still?
[Q2] What should we do?

3 reversals!

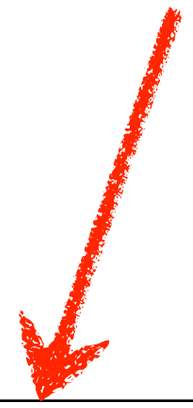
```
val q = new FQueue(Nil, List(6,5,4,3,2,1,0))
println(q.head); println(q.head); println(q.head)
```

```
val q = new FQueue(Nil, List(6,5,4,3,2,1,0))
println(q.head); println(q.head); println(q.head)
```

q

```
var l=List()
var t=List(6,5,4,3,2,1,0)
```

(1) Make l and t mutable var, not val.

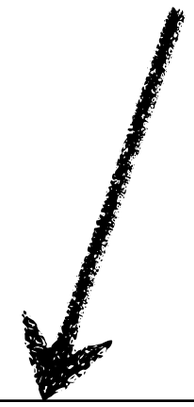


```
val q = new FQueue(Nil, List(6,5,4,3,2,1,0))
println(q.head); println(q.head); println(q.head)
```

q

```
var l=List(0,1,2,3,4,5,6)
var t=List()
```

(1) Make l and t mutable var, not val.



```
val q = new FQueue(Nil, List(6,5,4,3,2,1,0))
println(q.head); println(q.head); println(q.head)
```

(2) Store the result of reversal by updating l and t.

q

```
var l=List(0,1,2,3,4,5,6)
var t=List()
```

(1) Make l and t mutable var, not val.

```
val q = new FQueue(Nil, List(6,5,4,3,2,1,0))
println(q.head); println(q.head); println(q.head)
```

(2) Store the result of reversal by updating l and t.

(3) Reuse the stored result.

q

```
var l=List(0,1,2,3,4,5,6)
var t=List()
```

(1) Make l and t mutable var, not val.

```
val q = new FQueue(Nil, List(6,5,4,3,2,1,0))
println(q.head); println(q.head); println(q.head)
```

(2) Store the result of reversal by updating l and t.

(3) Reuse the stored result.

Invisible state change -- Client programs of FQueue cannot notice the change for l and t.

Implementation of invisible state change

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() = ...

  def head: T =
  def tail: FQueue[T] = ...
  def enqueue[U >: T](x: U): FQueue[U] = ...

}
```

We use mutable state.

Implementation of invisible state change

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() = ...

  def head: T = ...
  def tail: FQueue[T] = ...
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
}
```

We use mutable state.

But still update by creating a new object.

Implementation of invisible state change

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = ...
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
}
```

We use mutable state.

But still update by creating a new object.

Exercise:
Implement the tail method

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = ...
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
}
```

We use mutable state.

But still update by creating a new object.

Exercise:
Implement the tail method

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = { mirror(); new FQueue(l.tail,t) }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
}
```

We use mutable state.

But still update by creating a new object.

Exercise:
Implement the tail method

Doesn't compile.

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = { mirror(); new FQueue(l.tail,t) }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
}
```

We use mutable state.

But still update by creating a new object.

Exercise:
Implement the tail method

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = { mirror(); new FQueue(l.tail,Nil) }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
  def reset(q: FQueue[Any]) { q.l = List(10) }
}

val q = new FQueue[Boolean](Nil,Nil); q.reset(q)
println(q.head && q.head)
```

Doesn't compile.

We use mutable state.

But still update by creating a new object.

Exercise:
Implement the tail method

```
class FQueue[+T](
  private var l: List[T],
  private var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = { mirror(); new FQueue(l.tail,Nil) }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)

}
```

Doesn't compile.
[Q] Fix this.

We use mutable state.

But still update by creating a new object.

Exercise:
Implement the tail method

```
class FQueue[+T](
  private[this] var l: List[T],
  private[this] var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = { mirror(); new FQueue(l.tail,Nil) }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)

}
```

Doesn't compile.
[Q] Fix this.

We cannot update

But still update by creating a new object.

All "generic" var fields should be private to the object.

Exercise:
Implement the tail method

```
class FQueue[+T](
  private[this] var l: List[T],
  private[this] var t: List[T]
) {
  private def mirror() =
    if (l.isEmpty) { l = t.reverse; t = Nil }
  def head: T = { mirror(); l.head }
  def tail: FQueue[T] = { mirror(); new FQueue(l.tail,Nil) }
  def enqueue[U >: T](x: U): FQueue[U] =
    new FQueue(l, x :: t)
  def reset(q: FQueue[Any]) { q.l = List(10) }
}
```

Doesn't compile.
[Q] Fix this.

Compiler reports an access error at `q.l`

All "generic" var fields should be private to the object.

Covariant type parameter T

- T can appear only in a positive position.
- A mutable variable involving T should be private to the object.
- This means that the mutable part of the object is invisible to the outside.
- Usually, covariant type parameters work well with classes for immutable objects.

Last question

- Can we implement a persistent queue with O(log n) or O(1) complexity in the worst case?

Last question

- Can we implement a persistent queue with O(log n) or O(1) complexity in the worst case?
- Yes. If you are interested, see the below paper by Chris Okasaki:

  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.8825

Summary

- Developed a persistent covariant generic queue with O(1) amortised cost.
- Interaction between mutable state and variance.
- Read Chapter 19.