

A Closer Look At Lists

Principles of Functional Programming

Lists Recap

Lists are the core data structure we will work with over the next weeks.

Type: `List[Fruit]`

Construction:

```
val fruits = List("Apple", "Orange", "Banana")  
val nums = 1 :: 2 :: Nil
```

Decomposition:

```
fruits.head      // "Apple"  
nums.tail        // 2 :: Nil  
nums.isEmpty     // false
```

```
nums match  
  case x :: y :: _ => x + y    // 3
```

List Methods (1)

Sublists and element access:

<code>xs.length</code>	The number of elements of <code>xs</code> .
<code>xs.last</code>	The list's last element, exception if <code>xs</code> is empty.
<code>xs.init</code>	A list consisting of all elements of <code>xs</code> except the last one, exception if <code>xs</code> is empty.
<code>xs.take(n)</code>	A list consisting of the first <code>n</code> elements of <code>xs</code> , or <code>xs</code> itself if it is shorter than <code>n</code> .
<code>xs.drop(n)</code>	The rest of the collection after taking <code>n</code> elements.
<code>xs(n)</code>	(or, written out, <code>xs.apply(n)</code>). The element of <code>xs</code> at index <code>n</code> .

List Methods (2)

Creating new lists:

<code>xs ++ ys</code>	The list consisting of all elements of <code>xs</code> followed by all elements of <code>ys</code> .
<code>xs.reverse</code>	The list containing the elements of <code>xs</code> in reversed order.
<code>xs.updated(n, x)</code>	The list containing the same elements as <code>xs</code> , except at index <code>n</code> where it contains <code>x</code> .

Finding elements:

<code>xs.indexOf(x)</code>	The index of the first element in <code>xs</code> equal to <code>x</code> , or <code>-1</code> if <code>x</code> does not appear in <code>xs</code> .
<code>xs.contains(x)</code>	same as <code>xs.indexOf(x) >= 0</code>

Implementation of last

The complexity of head is (small) constant time.

What is the complexity of last?

To find out, let's write a possible implementation of last as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) =>
  case y :: ys =>
```

Implementation of last

The complexity of head is (small) constant time.

What is the complexity of last?

To find out, let's write a possible implementation of last as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys =>
```

Implementation of last

The complexity of head is (small) constant time.

What is the complexity of last?

To find out, let's write a possible implementation of last as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
```

Implementation of last

The complexity of head is (small) constant time.

What is the complexity of last?

To find out, let's write a possible implementation of last as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
```

So, last takes steps proportional to the length of the list xs.

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) => ???
  case y :: ys => ???
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) =>
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) => List()
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) => List()
  case y :: ys => y :: init(ys)
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])  
  def ++ (ys: List[T]): List[T] =
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])  
  def ++ (ys: List[T]): List[T] = xs match  
    case Nil =>  
    case x :: xs1 =>
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])  
  def ++ (ys: List[T]): List[T] = xs match  
    case Nil => ys  
    case x :: xs1 =>
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])  
  def ++ (ys: List[T]): List[T] = xs match  
    case Nil => ys  
    case x :: xs1 => x :: (xs1 ++ ys)
```


Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])  
  def ++ (ys: List[T]): List[T] = xs match  
    case Nil => ys  
    case x :: xs1 => x :: (xs1 ++ ys)
```

What is the complexity of concat?

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])  
  def ++ (ys: List[T]): List[T] = xs match  
    case Nil => ys  
    case x :: xs1 => x :: (xs1 ++ ys)
```

What is the complexity of concat?

Answer: $O(xs.length)$

Implementation of reverse

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])  
  def reverse: List[T] = xs match  
    case Nil =>  
    case y :: ys =>
```

Implementation of reverse

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])  
  def reverse: List[T] = xs match  
    case Nil => Nil  
    case y :: ys =>
```

Implementation of reverse

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])  
  def reverse: List[T] = xs match  
    case Nil => Nil  
    case y :: ys => ys.reverse ++ List(y)
```

Implementation of reverse

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])  
  def reverse: List[T] = xs match  
    case Nil => Nil  
    case y :: ys => ys.reverse ++ List(y)
```

What is the complexity of reverse?

Implementation of reverse

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])  
  def reverse: List[T] = xs match  
    case Nil => Nil  
    case y :: ys => ys.reverse ++ List(y)
```

What is the complexity of reverse?

Answer: $O(xs.length * xs.length)$

Can we do better? (to be solved later).

Exercise

Remove the n 'th element of a list `xs`. If n is out of bounds, return `xs` itself.

```
def removeAt[T](n: Int, xs: List[T]) = ???
```

Usage example:

```
removeAt(1, List('a', 'b', 'c', 'd')) > List(a, c, d)
```


Exercise (Harder, Optional)

Flatten a list structure:

```
def flatten(xs: List[Any]): List[Any] = ???
```

```
flatten(List(List(1, 1), 2, List(3, List(5, 8))))  
  > res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```