Working With Lists

```
val fruit = List("apples", "oranges", "pears")
val nums: List[Int] = List(1, 2, 3, 4)
val diag3 =
    List(
      List(1, 0, 0),
      List(0, 1, 0),
      List(0, 0, 1)
    )
val empty = List()
```

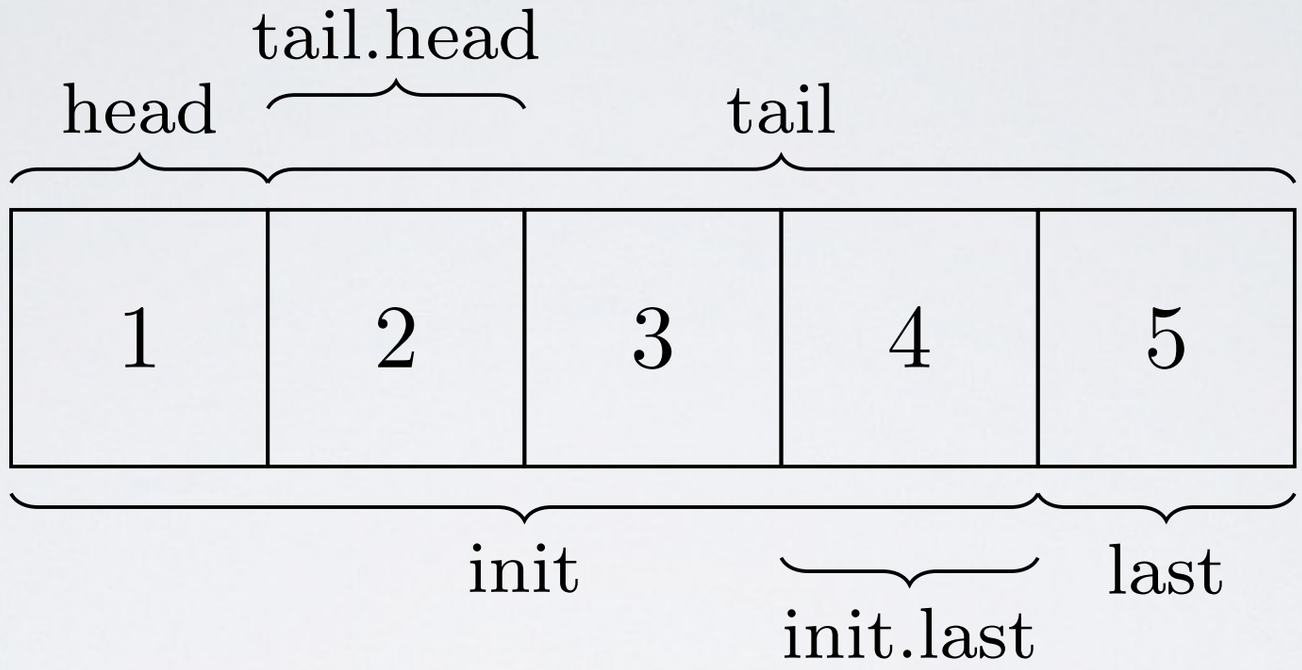
- List are immutable
- Lists have recursive structure
- Lists are homogeneous

Constructing Lists

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

val nums =1::(2::(3::(4::Nil)))

val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil

val empty = Nil
```

- Nil represents the empty list
- :: („cons“) extends list at the front
- :: is right associative => A :: B :: C = A :: (B :: C)

Basic Operations



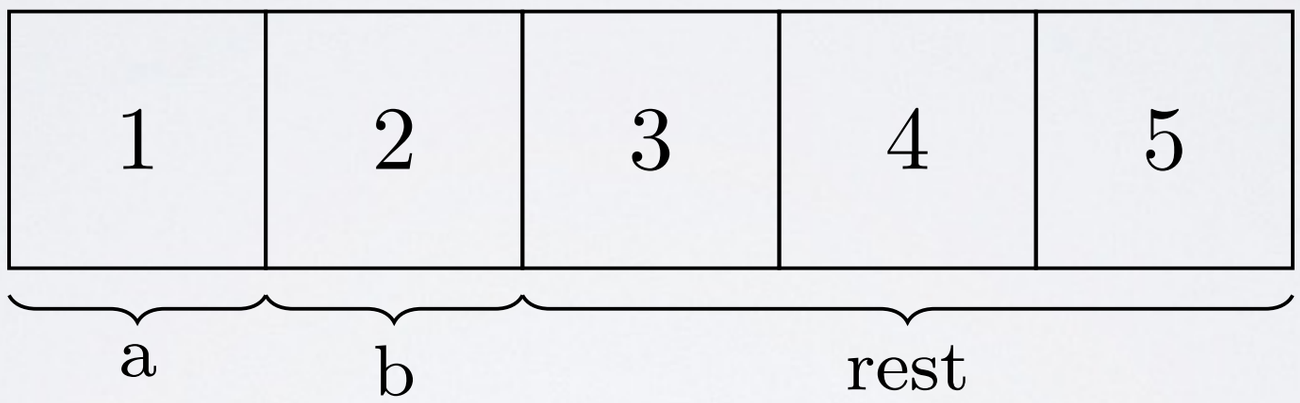
- isEmpty returns true if the list is empty
- head returns the first element
- tail returns everything but the first element
- init returns everything but the last element
- last returns the last element

Insertion Sort

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```

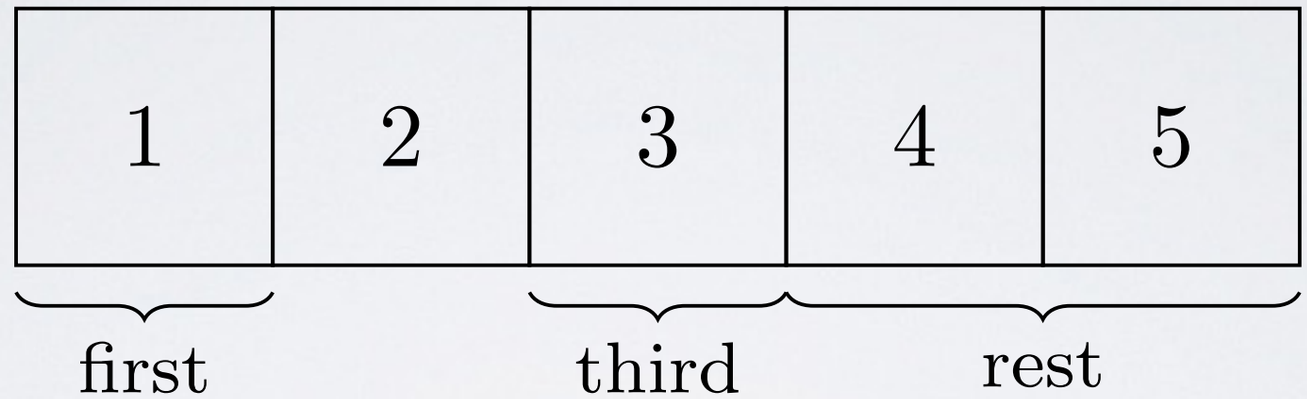
List Patterns

```
val nums: List = List(1, 2, 3, 4, 5)
val a :: b :: rest = nums
```




```
a: Int = 1
b: Int = 2
rest: List[Int] = List(3, 4, 5)
```

List Patterns




```
val nums: List = List(1, 2, 3, 4, 5)
val first :: _ :: third :: rest = nums
```

```
first: Int = 1
third: Int = 3
rest: List[Int] = List(4, 5)
```

List Concatenation

```
val a = List(1, 2, 3)
val b = List(4, 5, 6)
val c = a :: b
    c: List[Any] = List(List(1, 2, 3), 4, 5, 6)
val d = a ::: b
    d: List[Int] = List(1, 2, 3, 4, 5, 6)
```

List Concatenation

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }
```

Common List Operations

- length — returns the number of elements in the list
- reverse — returns a reversed version of the list
- drop — returns everything but the first n elements
- take — returns the first n elements
- splitAt — splits a list into two lists at a specified index
- flatten — takes a list of lists and flattens it out to a single list
- zip — combines two lists into a list of pairs
- unzip — splits a list of pairs into two lists

List Conversions

- toString returns a string representation of the list
- mkString returns a string representation with prefix, seperator and postfix
- toArray converts the list to an array
- iterator creates an iterator for the list

Higher-order List Methods: Map

applies a specified function to each element of a list

```
val alph = ('a' to 'z').toList
alph: List[Char] = List(a, b, c, d, e, …, v, w, x, y, z)
val alphUp = alph.map(_.toUpper)
alphUp: List[Char] = List(A, B, C, D, E, …, V, W, X, Y, Z)
```

Higher-order List Methods: flatMap

applies a specified function that returns a list to each element of the list and flattens (concatenates) the result to one single list

```
val words = List("the", "quick", "brown", „fox")

words map (_.toList)
List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k), List(b, r, o, w, n),
List(f, o, x))

words flatMap (_.toList)
List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x)
```

Higher-order List Methods: foreach

applies a specified procedure to each element of a list

```
var sum = 0
sum: Int = 0

List(1, 2, 3, 4, 5) foreach (sum += _)

sum
Int = 15
```

Higher-order List Methods: filter, partition

filters/partition a list according to a predicate function

```
List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
List[Int] = List(2, 4)

List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
(List[Int], List[Int]) = (List(2, 4), List(1, 3, 5))
```

Higher-order List Methods: find

find the first element satisfying a predicate function

```
List(1, 2, 3, 4, 5) find (_ % 2 == 0)
Option[Int] = Some(2)
```

Higher-order List Methods: dropWhile, takeWhile, span

take/drop elements as long as they satisfy the predicate function

```
List(1, 2, 3, 4, 5) takeWhile (_ <= 2)
List[Int] = List(1, 2)
```

```
List(1, 2, 3, 4, 5) dropWhile (_ <= 2)
List[Int] = List(3, 4, 5)
```

```
List(1, 2, 3, 4, 5) span (_ <= 2)
(List[Int], List[Int]) = (List(1, 2), List(3, 4, 5))
```

Higher-order List Methods: forall, exists

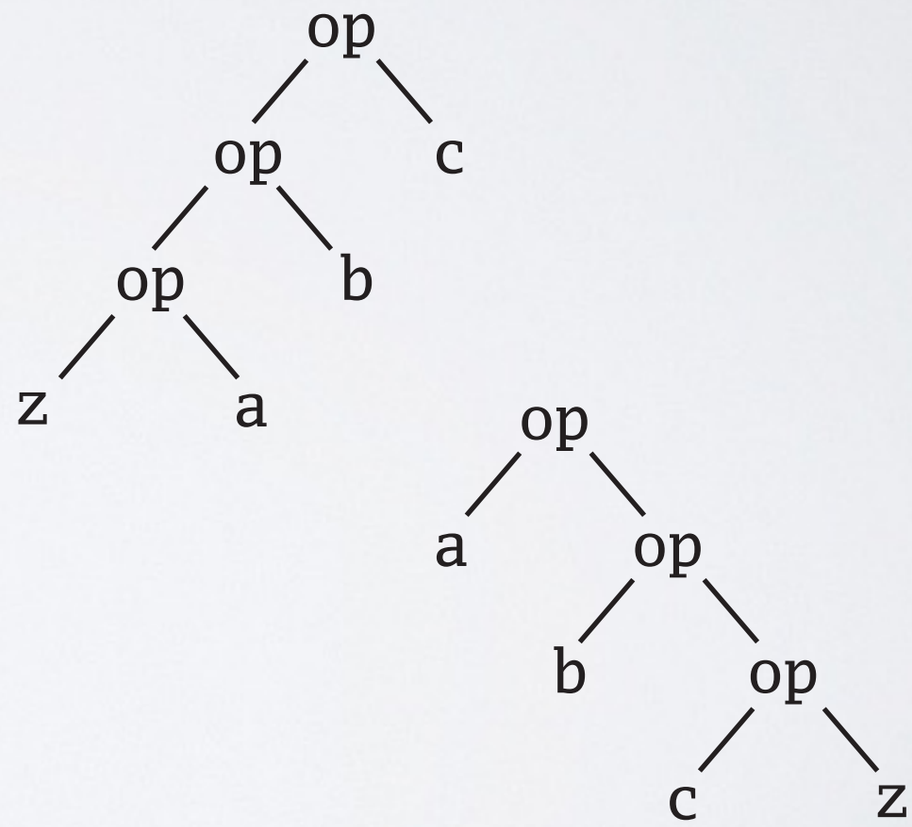
check if every/an element satisfies the predicate function

```
List(1, 2, 3, 4, 5) forall (_ % 2 == 0)
Boolean = false

List(2, 4, 6, 8) forall (_ % 2 == 0)
Boolean = true

List(1, 2, 3, 4, 5) exits (_ % 2 == 0)
Boolean = true
```

Higher-order List Methods: foldLeft (/:) , foldRight (:\)

combine elements with an operator

```
(0 /: List(1, 2, 3, 4, 5)) (_ + _)
Int = 15

(List(1, 2, 5, 3, 4) :\ 0) (_ max _)
Int = 5
```



Higher-order List Methods: sortWith

sorts list according to a function

```
List(1, -3, 4, 2, 6) sortWith (_ < _)
List[Int] = List(-3, 1, 2, 4, 6)

List("the", "quick", "brown", "fox") sortWith (_.length > _.length)
List[java.lang.String] = List(quick, brown, the, fox)
```

Higher-order List Methods: fill, tabulate

fill a list with elements

```
List.fill(5)("x")
List[String] = List(x, x, x, x, x)

List.tabulate(5)(n => "x" * (n+1))
List[String] = List(x, xx, xxx, xxxx, xxxxx)
```