```scala
package replist

import scala.collection.immutable.Stream.Empty


def prepend[A](rl: RepList[A], value: A): RepList[A] =
  Cons(value, rl)

def headCopied[A](rl: RepList[A], n: Long): RepList[A] = rl match {
  case Empty =>
    throw new NoSuchElementException("Empyt list has no head to copy")
  case Cons(h, tail) => Repeat(n, Cons(h, tail))
  case Repeat(k, Cons(h, tail)) => Repeat(n + k, Cons(h, tail))
}

def head[A](rl: RepList[A]): A = rl match {
  case Empty =>
    throw new NoSuchElementException("Empty list does not have head")
  case Cons(h, _) =>
    h
  case Repeat(_, Cons(h, _)) =>
    h
}

def tail[A](rl: RepList[A]): RepList[A] = rl match {
  case Empty =>
    throw new NoSuchElementException("Empty list does not have tail")
  case Cons(h, tail) =>
    tail
  case Repeat(n, Cons(h, tail)) =>
    tail
}

```

```scala
@tailrec
def last[A](rl: RepList[A]): A = rl match {
  case Empty =>
    throw new NoSuchElementException("Empty list does not have last element")
  case Cons(h, Empty) =>
    h // base case
  case Cons(h, tail) =>
    last(tail) // recursive case on non repeat list
  case Repeat(n, Cons(h, Empty)) =>
    h // base case on repeat list
  case Repeat(n, Cons(h, tail)) =>
    last(tail) // recursive case on repeat list
}

def length[A](rl: RepList[A]): Long = {

@tailrec
def loop(rl: RepList[A], sum: Long): Long = rl match {
  case Empty => sum
  case Cons(h, tail) => loop(tail, sum + 1)
  case Repeat(n, Cons(h, tail)) => loop(tail, sum + n)
}

loop(rl, 0)
}

def take[A](rl: RepList[A], n: Long): RepList[A] = (n, rl) match {
  case (n, l) if n >= length(l) =>
    rl //short cut if n > length(rl)
  case (n, tail) if n <= 0 =>
    Empty //base case, returns Empty to recursive case once n == 0
  case (_, Empty) =>
    Empty //should be un-reachable but just in case
  case (n, Cons(h, tail)) =>
    Cons(h, take(tail, n - 1)) //recursive case for non
```

```scala
 repeat list
  case (n, Repeat(k, Cons(h, tail))) if n <= k =>
    Repeat(n, Cons(h, Empty)) //recursive case for a repeat list when n < number of repeats
  case (n, Repeat(k, Cons(h, tail))) =>
    Repeat(k, Cons(h, take(tail, n - k))) //recursive case for a repeat list with less than n repeats
}

def drop[A](rl: RepList[A], n: Long): RepList[A] = (n, rl) match {
  case (n, l) if n >= length(l) =>
    Empty //short cut if n > length(rl)
  case (n, l) if n <= 0 =>
    l //base case, returns remaining list to recursive case once n == 0
  case (n, Empty) =>
    Empty //should be un-reachable but just in case
  case (n, Cons(h, tail)) =>
    drop(tail, n - 1) //recursive case for non repeat list
  case (n, Repeat(k, Cons(h, tail))) if k == 2 =>
    drop(Cons(h, tail), n - 1) //recursive case for a repeat list that will become a cons
  case (n, Repeat(k, Cons(h, tail))) =>
    drop(Repeat(k - 1, (h, tail)), n - 1) //recursive case for a repeat list that will still be a repeat list
}

def isEmpty[A](rl: RepList[A]): Boolean = rl match {
  case Empty => true
  case _ => false
}

def nonEmpty[A](rl: RepList[A]): Boolean = {
  !isEmpty(rl)
}
```

```scala

@tailrec
def contains[A](rl: RepList[A], target: A): Boolean = rl match {
  case Empty => false
  case Cons(h, _) if h == target => true
  case Cons(h, l) if h != target => contains(l, target)
  case Repeat(_, Cons(h, l)) => contains(Cons(h, l), target)
}

def count[A](rl: RepList[A], target: A): Long = {
  @tailrec
  def loop(rl: RepList[A], target: A, count: Long): Long = rl match {
    case Empty => count
    case Cons(h, tail) if h == target => loop(tail, target, count + 1)
    case Cons(h, tail) if h != target => loop(tail, target, count)
    case Repeat(n, Cons(h, tail)) if h == target => loop(tail, target, count + n)
    case Repeat(n, Cons(h, tail)) if h != target => loop(tail, target, count)
  }

  loop(rl, target, 0)
}
def removeAll[A](rl: RepList[A], target: A): RepList[A] = rl match {
  case Empty => Empty
  case (Cons(h, tail)) if h == target => removeAll(drop(rl, 1), target)
  case (Cons(h, tail)) if h != target => Cons(h, removeAll(tail, target))
  case Repeat(n, Cons(h, tail)) if h == target => removeAll(drop(rl, n), target)
  case Repeat(n, Cons(h, tail)) if h != target => Repeat(n, (h, removeAll(tail, target)))
}

@tailrec
def getAt[A](rl: RepList[A], i: Long): A = rl match {
  case Empty =>
    throw new NoSuchElementException("Empty list has no entry at index $i")
  case Cons(h, tail) if i == 0 => h
  case Cons(h, tail) if i > 0 => getAt(tail, i - 1)
  case Repeat(n, Cons(h, tail)) if i < n => h
  case Repeat(n, Cons(h, tail)) if i >= n => getAt(tail, i - n)
}

def toList[A](rl: RepList[A]): List[A] = rl match {
  case Empty => Nil
  case Cons(h, tail) => h :: toList(tail)
  case Repeat(n, Cons(h, tail)) => h :: toList(Repeat(n - 1, Cons(h, tail)))
}

def toSet[A](rl: RepList[A]): Set[A] = rl match {
  case Empty => Set.empty[A]
  case Cons(h, tail) => toSet(tail) + h
  case Repeat(n, Cons(h, tail)) => toSet(tail) + h
}

def concat[A](rl1: RepList[A], rl2: RepList[A]): RepList[A] = rl1 match {
  case Empty => rl2
  case Cons(h, tail) => Cons(h, concat(tail, rl2))
  case Repeat(n, Cons(h, tail)) => Repeat(n, Cons(h, concat(tail, rl2)))
}

@tailrec
```

```scala
def reverse[A](rl: RepList[A]): RepList[A] = {
  def loop(rl: RepList[A], revrs: RepList[A]): RepList[A] = rl match {
    case Empty => revrs
    case Cons(h, tail) => loop(tail, prepend(revrs, h))
    case Repeat(n, Cons(h, tail)) => loop(tail, Repeat(n, Cons(h, revrs)))
  }

  loop(rl, Empty)
}

def sort[A: Ordering](rl: RepList[A]): RepList[A] = ???

```