```scala
package radixtree

import tinyscalautils.lang.InterruptibleConstructor

private final case class Node(str: String, edges: Edges, isWord: Boolean) extends InterruptibleConstructor:
  def size: Int  = (if isWord then 1 else 0) + edges.size
  def depth: Int =  1 + edges.depth
  def width: Int = {
    if edges.isEmpty then 1
    else math.max(edges.nodes.length, edges.width)
  }

  def isEmpty: Boolean = if str.isEmpty then true else false

  def nonEmpty: Boolean = !isEmpty

  def +(word: String): Node =

    val common = longestPrefix(word, str)

    if common == str.length then
      val rest = word.drop(str.length)
      if rest.isEmpty then
        Node(str, edges, isWord = true)
      else
        copy(edges = edges + rest)
    else
      val shared = str.take(common)
      val oldSuffix = str.drop(common)
      val newSuffix = word.drop(common)

      if newSuffix.isEmpty then
        Node(shared, Edges(Vector(Node(oldSuffix, edges, isWord))), isWord = true)
      else
```

```scala
        Node(shared,
          Edges(Vector(Node(oldSuffix, edges, isWord), Node(newSuffix, Edges(Vector()), isWord = true))), isWord = false)

  def -(word: String): Node =
    val common = longestPrefix(word, str)

    if common < str.length then
      this
    else
      val rest = word.drop(str.length)
      if rest.isEmpty then
        if edges.nodes.isEmpty then null
        else Node(str, edges, isWord = false).canonicalize
      else
        val updatedEdges = edges - rest
        Node(str, edges = updatedEdges, isWord).canonicalize

  private def canonicalize: Node =
    if isWord then this
    else
      edges.nodes match
        case IndexedSeq() => null
        case IndexedSeq(single) =>
          Node(str + single.str, single.edges, single.isWord)
        case _ =>
          this

  def complete(prefix: String): Edges =
    val common = longestPrefix(prefix, str)

    if common < str.length || edges.isEmpty then
      edges + str.drop(prefix.length)
    else
```

```scala
      val rest = prefix.drop(str.length)
      if rest.isEmpty then edges
      else edges.complete(rest)


  def search(word: String): Int =
    val common = longestPrefix(word, str)

    if common < str.length then 0
    else
      val rest = word.drop(str.length)
      if rest.isEmpty then
        if isWord then 1 else 0
      else edges.search(rest)

  def foreach[U](prefix: String, f: String => U): Unit =
    val word = prefix + str
    if isWord then f(word)
    edges.nodes.foreach(_.foreach(word, f))

  def fold[A](acc: A, prefix: String, f: (A, String) => A): A =
    var a = acc
    val word = prefix + str
    if isWord then a = f(a, word)
    edges.nodes.foldLeft(a)((cur, n) => n.fold(cur, word, f))

  def find(prefix: String, test: String => Boolean): Option[String] =
    val current = prefix + str

    if isWord && test(current) then
      return Some(current)

    edges.nodes.foldLeft(Option.empty[String]) { (acc, child) =>
```

```scala
      acc match
        case some@Some(_) => some
        case None => child.find(current, test)
    }

  def dumpStrings: Seq[(String, Boolean)] =
    (str -> isWord) +: edges.nodes.flatMap(_.dumpStrings)

  def allWords(prefix: String, buf: collection.mutable.ListBuffer[String]): Unit =
    val word = prefix + str
    if isWord then buf += word
    edges.nodes.foreach(_.allWords(word, buf))

  def toString(depth: Int, sb: StringBuilder, separator: String): Unit =

    sb.append(separator * depth)
    sb.append(str)
    if isWord then sb.append("*")
    sb.append("\n")

    edges.nodes.foreach(_.toString(depth + 1, sb, separator))
end Node

// helper method: length of the longest common prefix of two strings
private def longestPrefix(str: String, word: String): Int =
  val m = math.min(str.length, word.length)
  var i = 0
  while i < m && str(i) == word(i) do i += 1
  i
```