```scala
package radixtree

private final case class Edges(nodes: IndexedSeq[Node]) extends WordSet:
  def isEmpty: Boolean  = nodes.isEmpty
  def nonEmpty: Boolean = !isEmpty
  def size: Int  = nodes.foldLeft(0)(_ + _.size)
  def depth: Int =
    if nodes.isEmpty then 0
    else nodes.map(_.depth).max
  def width: Int =
    if nodes.isEmpty then 0
    else math.max(nodes.length, nodes.map(_.width).max)

  def + (word: String): Edges =
    if word.isEmpty then
      val emptyRoot = Node("", Edges(nodes), isWord = true)
      Edges(IndexedSeq(emptyRoot))
    else
      val first = word.head
      val idx = nodes.indexWhere(_.str.head == first)
      if idx < 0 then
        Edges(nodes :+ Node(word, Edges(Vector()), isWord = true))
      else
        val updated = nodes(idx) + (word)
        Edges(nodes.updated(idx, updated))

  def - (word: String): Edges =
    if nodes.isEmpty then this
    else
      val first = word.headOption
      first match
        case None =>
          if nodes.nonEmpty && nodes.head.str.isEmpty then
            Edges(nodes.head.edges.nodes)
          else this

        case Some(c) =>
          val idx = nodes.indexWhere(_.str.head == c)
          if idx < 0 then this
          else
            val updated = nodes(idx) - word
            if updated == null then
              Edges(nodes.patch(idx, Nil, 1))
            else
              Edges(nodes.updated(idx, updated))

  def complete(prefix: String): Edges =
    prefix.headOption match
      case None => this
      case Some(c) =>
        nodes.find(_.str.head == c) match
          case None => Edges(Vector())
          case Some(n) => n.complete(prefix)

  def search(word: String): Int =
    word.headOption match
      case None =>
        if nodes.nonEmpty && nodes.head.str.isEmpty then 1 else -1

      case Some(c) =>
        nodes.find(_.str.head == c) match
          case None => -1
          case Some(n) => n.search(word)

  def contains(word: String): Boolean =
    search(word) == 1

  def find(test: String => Boolean): Option[String] =
    findHelper("", test)

  def exists(test: String => Boolean): Boolean =
```

```scala
    find(test).nonEmpty
  def forall(test: String => Boolean): Boolean =
      !exists(s => !test(s))
  def foreach[U](f: String => U): Unit =
    foreachHelper("", f)
  def fold[A](start: A)(f: (A, String) => A): A =
    foldHelper(start, "", f)

  def allWords: Seq[String] =
    val buf = collection.mutable.ListBuffer[String]()
    nodes.foreach(_.allWords("", buf))
    buf.toList

  def allStrings: Seq[(String, Boolean)] =
    nodes.flatMap(_.dumpStrings)

  def toString(separator: String): String =
    val sb = new StringBuilder
    nodes.foreach(_.toString(0, sb, separator))
    if sb.nonEmpty then sb.result()
    else ""

  // add methods if needed
  private def foreachHelper[U](prefix: String, f: String => U): Unit =
    nodes.foreach(_.foreach(prefix, f))

  private def foldHelper[A](acc: A, prefix: String, f: (A, String) => A): A =
    nodes.foldLeft(acc)((a, n) => n.fold(a, prefix, f))

  private def findHelper(prefix: String, test: String => Boolean): Option[String] =
    nodes.foldLeft(Option.empty[String]) { (acc, node) =>
      acc match
        case some@Some(_) =>
```

```scala
          some
        case None =>
          node.find(prefix, test)
    }
end Edges

private object Edges:
   val empty: Edges = Edges.of()
   def of(nodes: Node*): Edges = new Edges(nodes.toVector)
   // add methods if needed
end Edges

```