```java
/** A binary search tree of Comparables. */
public class BinarySearchTree<E extends Comparable<E>>
  implements Parent<E>, Set<E> {

  /** Root node. */
  private BinaryNode<E> root;

  /** A BinarySearchTree is initially empty. */
  public BinarySearchTree() {
    root = null;
  }

  public void add(E target) {
    Parent<E> parent = this;
    BinaryNode<E> node = root;
    int comparison = 0;
    while (node != null) {
      comparison = target.compareTo(node.getItem());
      if (comparison < 0) {     // Go left
        parent = node;
        node = node.getLeft();
      } else if (comparison == 0) { // It's already here
        return;
      } else {                  // Go right
        parent = node;
        node = node.getRight();
      }
    }
    parent.setChild(comparison, new BinaryNode<E>(target));
  }

  public boolean contains(E target) {
    BinaryNode<E> node = root;
    while (node != null) {
      int comparison = target.compareTo(node.getItem());
      if (comparison < 0) {     // Go left
        node = node.getLeft();
      } else if (comparison == 0) { // Found it
        return true;
      } else {                  // Go right
        node = node.getRight();
      }
    }
    return false;
  }

  public BinaryNode<E> getChild(int direction) {
    return root;
  }

  public void remove(E target) {
    Parent<E> parent = this;
    BinaryNode<E> node = root;
    int direction = 0;
    while (node != null) {
      int comparison = target.compareTo(node.getItem());
      if (comparison < 0) {     // Go left
        parent = node;
        node = node.getLeft();
      } else if (comparison == 0) { // Found it
        spliceOut(node, parent, direction);
        return;
      } else {                  // Go right
        parent = node;
        node = node.getRight();
      }
      direction = comparison;
    }
  }

  /**
   * Remove the leftmost descendant of node and return the
   * item contained in the removed node.
   */
  protected E removeLeftmost(BinaryNode<E> node, Parent<E> parent) {
    int direction = 1;
    while (node.getLeft() != null) {
      parent = node;
      direction = -1;
      node = node.getLeft();
    }
    E result = node.getItem();
    spliceOut(node, parent, direction);
    return result;
  }

  public void setChild(int direction, BinaryNode<E> child) {
    root = child;
  }
  public int size() {
    return size(root);
  }
  /** Return the size of the subtree rooted at node. */
  protected int size(BinaryNode<E> node) {
    if (node == null) {
      return 0;
    } else {
      return 1 + size(node.getLeft()) + size(node.getRight());
    }
  }
  /**
   * Remove node, which is a child of parent.  Direction is positive
   * if node is the right child of parent, negative if it is the
   * left child.
   */
  protected void spliceOut(BinaryNode<E> node,
                           Parent<E> parent,
                           int direction) {
    if (node.getLeft() == null) {
      parent.setChild(direction, node.getRight());
    } else if (node.getRight() == null) {
      parent.setChild(direction, node.getLeft());
    } else {
      node.setItem(removeLeftmost(node.getRight(), node));
    }
  }

}
```