```java
import dsa.BasicST;
import dsa.LinkedQueue;
import stdlib.StdIn;
import stdlib.StdOut;

public class ArrayST<Key, Value> implements BasicST<Key, Value> {
   private Key[] keys;     // keys in the symbol table
   private Value[] values; // the corresponding values
   private int n;          // number of key-value pairs

   // Constructs an empty symbol table.
   public ArrayST() {
       keys = (Key[]) new Object[2];
       values = (Value[]) new Object[2];
       n = 0;
   }

   // Returns true if this symbol table is empty, and false otherwise.
   public boolean isEmpty() {
       return n == 0;
   }

   // Returns the number of key-value pairs in this symbol table.
   public int size() {
       return n;
   }

   // Inserts the key and value pair into this symbol table.
   public void put(Key key, Value value) {
       if (key == null) {
           throw new IllegalArgumentException("key is null");
       }
       if (value == null) {
           throw new IllegalArgumentException("value is null");
       }

       for (int i = 0; i < n; i++) {
           if (key.equals(keys[i])) {
               values[i] = value;
               return;
           }
       }
```

```
    if (n == keys.length) {
        resize(2 * keys.length);
    }

    keys[n] = key;
    values[n] = value;
    n++;
}

// Returns the value associated with key in this symbol table, or null.
public Value get(Key key) {
    if (key == null) {
        throw new IllegalArgumentException("key is null");
    }
    for (int i = 0; i < n; i++) {
        if (key.equals(keys[i])) {
            return values[i];
        }
    }
    return null;
}

// Returns true if this symbol table contains key, and false otherwise.
public boolean contains(Key key) {
    if (key == null) {
        throw new IllegalArgumentException("key is null");
    }
    for (int i = 0; i < n; i++) {
        if (key.equals(keys[i])) {
            return true;
        }
    }
    return false;
}

// Deletes key and the associated value from this symbol table.
public void delete(Key key) {
    if (key == null) {
        throw new IllegalArgumentException("key is null");
    }
```

```
        for (int i = 0; i < n; i++) {
            if (key.equals(keys[i])) {
                for (int j = i; j < n - 1; j++) {
                    keys[j] = keys[j + 1];
                    values[j] = values[j + 1];
                }
                keys[n - 1] = null;
                values[n - 1] = null;
                n--;

                if (n > 0 && n == keys.length / 4) {
                    resize(keys.length / 2);
                }
                return;
            }
        }
    }

    // Returns all the keys in this symbol table.
    public Iterable<Key> keys() {
        LinkedQueue<Key> queue = new LinkedQueue<Key>();
        for (int i = 0; i < n; i++) {
            queue.enqueue(keys[i]);
        }
        return queue;
    }

    // Resizes the underlying arrays to capacity.
    private void resize(int capacity) {
        Key[] tempk = (Key[]) new Object[capacity];
        Value[] tempv = (Value[]) new Object[capacity];
        for (int i = 0; i < n; i++) {
            tempk[i] = keys[i];
            tempv[i] = values[i];
        }
        values = tempv;
        keys = tempk;
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        ArrayST<String, Integer> st = new ArrayST<>();
```

```
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys()) {
            StdOut.println(s + " " + st.get(s));
        }
    }
}
```