CS240: Homework 5

Eli Zupke

November 12, 2017

The GitHub folder can be found at https://github.com/Trainzack/CS240/tree/master/Homework%205%20Dictionary.

Contents

1 src/cs240/Test.java

```java
package cs240;

import java.util.Iterator;

public class Test {

    private final static int TEST_SIZE = 10;

    public static void main(String args[]) {

        DictionaryInterface<Integer, String> dict = new SortedDictionaryStaticArray<>(TEST_SIZE);
        dictionaryTest(dict, "Static Array Dict");
        dict = new SortedDictionaryLinkedData<>();
        dictionaryTest(dict, "Linked Data Dict");

    }

    public static void dictionaryTest(DictionaryInterface<Integer, String> dict, String listName) {

        System.out.println("==================");
        System.out.println("Testing " + listName);
        System.out.println("==================");

        Integer[] keyData = new Integer[TEST_SIZE];
        String[] valueDataA = new String[TEST_SIZE];
        String[] valueDataB = new String[TEST_SIZE];

        for (int i = 0; i < TEST_SIZE; i++) {
            keyData[i] = new Integer(i);
            valueDataA[i] = "A:" + i;
            valueDataB[i] = "B:" + i;
        }

        // Add items to the dictionary

        boolean valueCheck = true;
        boolean containmentCheck = true;
        boolean sizeCheck = true;
        boolean replacementCheck = true;
        boolean removalCheck = true;
        boolean emptyCheck = true;

        boolean iteratorHasNext = true;
        boolean iteratorNext = true;
        boolean iteratorRemove = true;

        // The dictionary should be empty
        emptyCheck = booleanLatch(emptyCheck, dict.isEmpty());
        // Add the even numbers to the dictionary
        for (int i = 0; i < TEST_SIZE; i+= 2) {

            // System.out.println("Adding " + i);
            String v = dict.add(keyData[i], valueDataA[i]);

            // We are replacing nothing, so this should always return null.
            replacementCheck = booleanLatch(replacementCheck, v == null);

            // Check to make sure the size of the dictionary is correct
            sizeCheck = booleanLatch(sizeCheck, dict.getSize() == (i / 2) + 1);

            // The dictionary should not be empty
            emptyCheck = booleanLatch(emptyCheck, !dict.isEmpty());

            for (int j = 0; j <= i; j += 2) {
                // Ensure that the dictionary contains the things we put in
                containmentCheck = booleanLatch(containmentCheck, dict.contains(keyData[j]));
                // Ensure that the value is correct
                valueCheck = booleanLatch(valueCheck, dict.getValue(keyData[j]) == valueDataA[j]);
            }

            // Ensure that the dictionary does not contain the things we didn’t put in
            for (int j = i + 2; j < TEST_SIZE; j += 2) {
                if (dict.contains(keyData[j])) {
                    containmentCheck = false;
                }
            }

        }

        // Add all of the numbers to the dictionary, but with the B values.
        for (int i = 0; i < TEST_SIZE; i++) {
```

```
        // System.out.println("====================");
        // System.out.println("Adding " + i);
        String v = dict.add(keyData[i], valueDataB[i]);

        // dict.getSize(); // only do this because of testing side-effect

        if (i % 2 == 0) {
            // We are replacing the values we entered earlier
            replacementCheck = booleanLatch(replacementCheck, v == valueDataA[i]);
        } else {
            // We are replacing nothing, so this should always return null.
            replacementCheck = booleanLatch(replacementCheck, v == null);
        }


        for (int j = 1; j <= i; j += 2) {
            // Ensure that the dictionary contains the odd numbers we put in
            containmentCheck = booleanLatch(containmentCheck, dict.contains(keyData[j]));
            // Ensure that the value is correct
            valueCheck = booleanLatch(valueCheck, dict.getValue(keyData[j]) == valueDataB[j]);
        }

        // Ensure that the dictionary does not contain the things we didn't put in
        for (int j = i + 2; j < TEST_SIZE; j += 2) {
            booleanLatch(containmentCheck, !dict.contains(keyData[j]));
        }
    }


    Iterator<Integer> iK = dict.getKeyIterator();
    Iterator<String> iV = dict.getValueIterator();
    if (iK == null || iV == null) {
        iteratorHasNext = false;
        iteratorNext = false;
        iteratorRemove = false;
    } else {
        for (int i = 0; i < TEST_SIZE; i++) {
            iteratorHasNext = booleanLatch(iteratorHasNext, iK.hasNext() && iV.hasNext());
            Integer k = iK.next();
            String v = iV.next();
            iteratorNext = booleanLatch(iteratorNext, k == keyData[k] && v == valueDataB[i]);
        }
        iteratorHasNext = booleanLatch(iteratorHasNext, !iK.hasNext() && !iV.hasNext());
    }

    for (int i = 0; i < TEST_SIZE; i++) {
        String v = dict.remove(keyData[i]);
        removalCheck = booleanLatch(removalCheck, v == valueDataB[i] && dict.getSize() == (TEST_SIZE - i-1));
    }

    // The dictionary should be empty
```

```java
        emptyCheck = booleanLatch(emptyCheck, dict.isEmpty());

        // Fill up the dictionary again.
        for (int i = 0; i < TEST_SIZE; i++) {
            dict.add(keyData[i], valueDataA[i]);
        }

        iK = dict.getKeyIterator();

        for (int i = TEST_SIZE; i > 0; i--) {
            iK.next();
            iK.remove();
            iteratorRemove = booleanLatch(iteratorRemove, dict.getSize() == i - 1);
        }
        // Fill up the dictionary again.
        for (int i = 0; i < TEST_SIZE; i++) {
            dict.add(keyData[i], valueDataB[i]);
        }

        iV = dict.getValueIterator();

        for (int i = TEST_SIZE; i > 0; i--) {
            iV.next();
            iV.remove();
            iteratorRemove = booleanLatch(iteratorRemove, dict.getSize() == i - 1);
        }


        printTestResult(removalCheck, listName, "Removal Check");
        printTestResult(emptyCheck, listName, "Empty Check");
        printTestResult(sizeCheck, listName, "Size Check");
        printTestResult(valueCheck, listName, "Value Check");
        printTestResult(containmentCheck, listName, "Containment Check");
        printTestResult(iteratorHasNext, listName, "Iterator Check (Has Next)");
        printTestResult(iteratorNext, listName, "Iterator Check (Next)");
        printTestResult(iteratorRemove, listName, "Iterator Check (Remove)");
    }


    /**
     * Print the results of a test
     * @param result Whether the test passed
     * @param name The name of the test
     */
    public static void printTestResult(boolean result, String listName, String name) {
        if (result) System.out.println(listName + " passed " + name +".");
        else System.err.println(listName + " failed " + name + "!");
    }

    /**
     * Returns the value of a flag such that the value of the flag is set to false if the new data is set to false,
     * but is never returned to true.
     * @param flag The flag whose value we are returning
     * @param newData The new data we are getting
     * @return The new value of the flag
     */
    private static boolean booleanLatch(boolean flag, boolean newData) {
        if (flag && newData) {
            return true;
        }
        return false;
    }

    /**
     * Take an array of objects, and print them out nicely.
     * @param l The array to print
     */
    public static void printArray(Object[] l) {

        System.out.print("[");
        for (int i = 0; i < l.length; i++) {
            System.out.print(l[i]);
            if (i+1 < l.length) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }



}
```

2 src/cs240/DictionaryInterface.java

```
package cs240;

import java.util.Iterator;
/**
   An interface for a dictionary with distinct search keys.
   @author Frank M. Carrano
   @author Timothy M. Henry
   @version 4.0
 */
public interface DictionaryInterface<K, V>
{
    /** Adds a new entry to this dictionary. If the given search key already
       exists in the dictionary, replaces the corresponding value.
       @param key    An object search key of the new entry.
       @param value  An object associated with the search key.
       @return  Either null if the new entry was added to the dictionary
                or the value that was associated with key if that value
                was replaced. */
    public V add(K key, V value);

    /** Removes a specific entry from this dictionary.
       @param key  An object search key of the entry to be removed.
       @return  Either the value that was associated with the search key
```

```java
                or null if no such object exists. */
    public V remove(K key);

    /** Retrieves from this dictionary the value associated with a given
       search key.
       @param key  An object search key of the entry to be retrieved.
       @return  Either the value that is associated with the search key
                or null if no such object exists. */
    public V getValue(K key);

    /** Sees whether a specific entry is in this dictionary.
       @param key  An object search key of the desired entry.
       @return  True if key is associated with an entry in the dictionary. */
    public boolean contains(K key);

    /** Creates an iterator that traverses all search keys in this dictionary.
       @return  An iterator that provides sequential access to the search
                keys in the dictionary. */
    public Iterator<K> getKeyIterator();

    /** Creates an iterator that traverses all values in this dictionary.
       @return  An iterator that provides sequential access to the values
                in this dictionary. */
    public Iterator<V> getValueIterator();

    /** Sees whether this dictionary is empty.
       @return  True if the dictionary is empty. */
    public boolean isEmpty();

    /** Gets the size of this dictionary.
       @return  The number of entries (key-value pairs) currently
                in the dictionary. */
    public int getSize();

    /** Removes all entries from this dictionary. */
    public void clear();
} // end DictionaryInterface
```

3 src/cs240/SortedDictionaryStaticArray.java

```java
package cs240;

import java.util.Iterator;
import java.util.NoSuchElementException;


/**
 * Implements the Sorted Dictionary ADT using a fixed size array. Keys in this dictionary are sorted ascendingly.
 * @author Eli Zupke
 *
 * @param <K> The type that will be used as keys in this dictionary
 * @param <V> The type that will be used as values in this dictionary
 */
public class SortedDictionaryStaticArray<K extends Comparable<? super K>, V> implements DictionaryInterface<K, V> {

    // Used to keep track of where the last element of the dictionary is stored.
    private int end;
    private int capacity;

    // These two arrays hold the keys and the values. The corresponding value of each key will be the entry in the value array with the same index.
    private K[] keyArray;
    private V[] valueArray;

    /**
     * Creates a new sorted dictionary via static array.
     * @param capacity The maximum number of key-value pairs in this dictionary.
     */
    public SortedDictionaryStaticArray(int capacity) {

        this.capacity = capacity;

        // The dictionary starts at zero, so start the end variable pointing at -1 (empty)
        end = -1;

        // Instantiate the arrays for both the keys and values.
        @SuppressWarnings("unchecked")
        K[] tempKeyArray = (K[])new Comparable[capacity]; // Unchecked cast
        keyArray = tempKeyArray;

        @SuppressWarnings("unchecked")
        V[] tempValueArray = (V[])new Comparable[capacity]; // Unchecked cast
        valueArray = tempValueArray;

    }

    @Override
    public V add(K key, V value) {
        // These store keys and values in the event that we need to add the key in the middle of the array.
        K curKey = null;
        V curValue = null;

        // Go down the array until we get to a value greater than the one we're adding, then move the rest down
        int i = 0;
        // System.out.println("Inserting " + value.toString());
        //Test.printArray(keyArray); Test.printArray(valueArray);

        for (; i < getSize() + 1; i++) {
            if (keyArray[i] == key) {
                // We already have the key, it seems.

                // Hold on to the old value, replace it with the new one, then return it.
                V returnValue = valueArray[i];
                valueArray[i] = value;

                return returnValue;
            } else if (keyArray[i] == null) {
                // System.out.println("TEST END");
                // We got to the end of the array, so let's place it at the end!
                valueArray[i] = value;
                keyArray[i] = key;
                end++;
                // We've added the element, so leave.
                return null;
            } else if (keyArray[i].compareTo(key) > 0) {
                // System.out.println("TEST GREATER");
                // We have found where to place our key, so let's do it!
                K tempKey = keyArray[i];
                V tempValue = valueArray[i];

                keyArray[i] = key;
                valueArray[i] = value;

                curKey = tempKey;
                curValue = tempValue;

                // Since we now know that we need to expand the array, but don't know whether we have enough room, let's check
                ensureCapacity();
                end++;
                break;
            }
        }
        // If we get here, then we know that we went through the last else if,
        // and we still need to move the remaining values over one index.

        for (i += 1; i < getSize(); i++) {
            // Move the next group of values
            K tempKey = keyArray[i];
            V tempValue = valueArray[i];

            keyArray[i] = curKey;
            valueArray[i] = curValue;

            curKey = tempKey;
            curValue = tempValue;
        }

        return null;
    }

    @Override
    public V remove(K key) {

        V value = null;

        // Declare the index variable outside the loop, so we can continue where we left off in the next one
        int i = 0;

        // Find the key, store its value, and stop the loop.
        // If it gets to the end, then the next loop will not be entered, and we will return null.
        for (; i <= end; i++) {
            if (keyArray[i] == key) {
                value = valueArray[i];
                break;
            }
        }

        // If we didn't find the key, then we can stop now
        if (value == null) {
            return null;
        }

        // Otherwise, move the rest of the values back.
        for (; i < end; i++) {
            // Move the next group of values
            keyArray[i] = keyArray[i+1];
            valueArray[i] = valueArray[i+1];
        }
        keyArray[end] = null;
        valueArray[end] = null;
        // Finally, reduce the end index by one.
        end--;

        return value;
    }

    @Override
    public V getValue(K key) {

        // Sequential search the key array for the key we are looking for.
        for (int i = 0; i < capacity; i++) {
            if (keyArray[i] == key) {
                // We found what we're looking for.
                return valueArray[i];
            }
        }
        // We couldn't find the key we were looking for.
        return null;
    }

    @Override
    public boolean contains(K key) {

        // Sequential search the key array for the key we are looking for.
        for (int i = 0; i < capacity; i++) {
            if (keyArray[i] == key) {
                return true;
            }
        }
        return false;
    }

    @Override
    public Iterator<K> getKeyIterator() {
        return new StaticArrayIterator<K>(true);
    }

    @Override
    public Iterator<V> getValueIterator() {

        return new StaticArrayIterator<V>(false);
    }

    private class StaticArrayIterator<I> implements Iterator<I> {

        // Whether this is an iterator of keys (if true) or values (if false)
        boolean key;

        // index is the index of the value we just gave.
        private int index = -1;

        // whether there is an element we can remove.
        boolean canRemove = false;

        StaticArrayIterator(boolean _key) {
            super();
            key = _key;
        }

        @Override
        public boolean hasNext() {
            return index < end;
        }

        // Because I will always equal K or V, and we know which one it will equal, we can do this cast.
        @SuppressWarnings("unchecked")
        @Override
        public I next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            index++;
            canRemove = true;
            if (key) {
                return (I)keyArray[index];
            } else {
                return (I)valueArray[index];
            }

        }

        @Override
        public void remove() {
            if (!canRemove) {
                throw new IllegalStateException();
            }
            canRemove = false;

            SortedDictionaryStaticArray.this.remove(keyArray[index]);
            // Because we removed an element, we need to move our index backwards
            index--;

        }
    }


    @Override
    public boolean isEmpty() {
        return getSize() == 0;
    }

    @Override
    public int getSize() {
        // The size of the dictionary is always equal to the position of the end index plus 1.
        return end + 1;
    }

    @Override
    public void clear() {

        // Dereference everything in both arrays
        for (int i = 0; i < capacity; i++) {
            keyArray[i] = null;
            valueArray[i] = null;
        }

        // Move the end index back to before the start of the array.
        end = -1;
    }

    /**
     * Test to see if we have room to add an element.
     * Throws an IndexOutOfBoundsException if the array is full.
     */
    private void ensureCapacity() {
        if (end + 1 >= capacity) {
            throw new IndexOutOfBoundsException("Max array size reached!");
        }
    }
}
```

4 src/cs240/SortedDictionaryLinkedData.java

```java
package cs240;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * An implementation of the Sorted Dictionary ADT using linked data
 * @author Eli Zupke
 *
 * @param <K> The type that will be used as keys in this dictionary
 * @param <V> The type that will be used as values in this dictionary
 */
public class SortedDictionaryLinkedData<K extends Comparable<? super K>, V> implements DictionaryInterface<K, V> {

    // This is the first node in the dictionary.
    private Node front;

    private class Node {

        K key;
        V value;
        Node next;

        public Node(K _key, V _value) {
            key = _key;
            value = _value;
            next = null;
        }
    }

    @Override
    public V add(K key, V value) {

        // Special case. This node simply becomes the front.
        if (front == null) {
            front = new Node(key, value);
            // we replaced nothing, so return null.
            return null;
        }

        // Special case. This node becomes the front.
        if (key.compareTo(front.key) < 0) {
            Node newNode = new Node(key, value);
            newNode.next = front;
            front = newNode;
            return null;
        }

        Node prevNode = null;
        Node curNode = front;

        while (curNode != null) {
            if (curNode.key == key) {
                // The key we’re adding already exists in our dictionary, so go ahead and replace the value.
                V oldValue = curNode.value;
                curNode.value = value;
                return oldValue;
            } else if (key.compareTo(curNode.key) < 0) {
                // We need to insert the key here.
                Node newNode = new Node(key, value);
                newNode.next = curNode;
                prevNode.next = newNode;
                return null;
            }

            prevNode = curNode;
            curNode = curNode.next;
        }
        // We have got to the end of the linked data, but we still haven’t added the new pair.
        // Therefore, we add it to the end.
        Node newNode = new Node(key, value);
        prevNode.next = newNode;
        return null;
    }

    @Override
    public V remove(K key) {

        // Special case: the node we want to remove is the front
        if (front != null && front.key == key) {
            V oldValue = front.value;
            front = front.next;
            return oldValue;
        }

        Node prevNode = null;
        Node curNode = front;
        while (curNode != null) {
            if (curNode.key == key) {
                // We’ve found the key in our dictionary
                V oldValue = curNode.value;
                prevNode.next = curNode.next;
                return oldValue;
            }
            prevNode = curNode;
            curNode = curNode.next;
        }
        return null;
    }

    @Override
    public V getValue(K key) {

        Node curNode = front;

        // Sequential search the list until we get to the node that we need.
        while (curNode != null) {
            if (curNode.key == key) {
                return curNode.value;
            }
            curNode = curNode.next;
        }

        return null;
    }

    @Override
    public boolean contains(K key) {

        Node curNode = front;

        while (curNode != null) {
            if (curNode.key == key)
                return true;
            curNode = curNode.next;
        }

        return false;
    }


    @Override
    public Iterator<K> getKeyIterator() {
        return new StaticArrayKeyIterator();
    }

    private class StaticArrayKeyIterator implements Iterator<K> {

        // The node we just gave
        Node prevNode = null;
        // The node we are about to give
        Node curNode = front;


        @Override
        public boolean hasNext() {
            return curNode != null;
        }

        @Override
        public K next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            prevNode = curNode;
            curNode = curNode.next;
            return prevNode.key;

        }

        @Override
        public void remove() {
            if (prevNode == null) {
                throw new IllegalStateException();
            }
            SortedDictionaryLinkedData.this.remove(prevNode.key);

        }
    }

    @Override
    public Iterator<V> getValueIterator() {

        return new StaticArrayValueIterator();
    }

    private class StaticArrayValueIterator implements Iterator<V> {

        // The node we just gave
        Node prevNode = null;
        // The node we are about to give
        Node curNode = front;


        @Override
        public boolean hasNext() {
            return curNode != null;
        }

        @Override
        public V next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            prevNode = curNode;
            curNode = curNode.next;
            return prevNode.value;

        }

        @Override
        public void remove() {
            if (prevNode == null) {
                throw new IllegalStateException();
            }
            SortedDictionaryLinkedData.this.remove(prevNode.key);

        }
    }

    @Override
    public boolean isEmpty() {
        return front == null;
    }

    @Override
    public int getSize() {
        int size = 0;

        Node curNode = front;

        // System.out.println("==========");
        while (curNode != null) {
            // System.out.println(curNode.value);
            size++;
            curNode = curNode.next;
        }

        return size;
    }

    @Override
    public void clear() {
        front = null;

    }

}
```