Homework 5: Max Heaps

● Graded

Student

Vidit Dharmendra Pokharna

Total Points

83 / 100 pts

Autograder Score

98.0 / 100.0

Failed Tests

Checkstyle (8/10)

Question 2

Feedback & Manual Grading

-15 / 0 pts

✓ **– 5 pts** Efficiency 1

✓ **– 5 pts** Efficiency 2

✓ **– 5 pts** Efficiency 3

[-2] Checkstyle

[-15] Efficiency - line 132, when we add, we only need to perform one "upheap", not an entire buildHeap to fix the tree!
- also line 154, remove you only need to call downheap once on index 1
- also line 170, also for getMax, there's no need to do buildheap since the tree should be valid

Great work! -Tomer (□■_■)

Autograder Results

Autograder Output

If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs

Checkstyle (8/10)

```
MaxHeap.java:
        line: 147, column23  - Expression can be simplified. [SimplifyBooleanExpressionCheck]
        line: 167, column23  - Expression can be simplified. [SimplifyBooleanExpressionCheck]
```

Submitted Files

MaxHeap.java

Download

```java
import java.util.ArrayList;
import java.util.NoSuchElementException;

/**
 * Your implementation of a MaxHeap.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators: LIST ALL COLLABORATORS YOU WORKED WITH HERE
 *
 * Resources: LIST ALL NON-COURSE RESOURCES YOU CONSULTED HERE
 */
public class MaxHeap<T extends Comparable<? super T>> {

    /*
     * The initial capacity of the MaxHeap when created with the default
     * constructor.
     *
     * DO NOT MODIFY THIS VARIABLE!
     */
    public static final int INITIAL_CAPACITY = 13;

    /*
     * Do not add new instance variables or modify existing ones.
     */
    private T[] backingArray;
    private int size;

    /**
     * Constructs a new MaxHeap.
     *
     * The backing array should have an initial capacity of INITIAL_CAPACITY.
     */
    public MaxHeap() {
        backingArray = (T[]) new Comparable[INITIAL_CAPACITY];
    }

    /**
     * Creates a properly ordered heap from a set of initial values.
     *
     * You must use the BuildHeap algorithm that was taught in lecture! Simply
     * adding the data one by one using the add method will not get any credit.
     * As a reminder, this is the algorithm that involves building the heap
     * from the bottom up by repeated use of downHeap operations.
     *
     * Before doing the algorithm, first copy over the data from the
     * ArrayList to the backingArray (leaving index 0 of the backingArray
     * empty). The data in the backingArray should be in the same order as it
     * appears in the passed in ArrayList before you start the BuildHeap
     * algorithm.
     *
     * The backingArray should have capacity 2n + 1 where n is the
     * number of data in the passed in ArrayList (not INITIAL_CAPACITY).
     * Index 0 should remain empty, indices 1 to n should contain the data in
     * proper order, and the rest of the indices should be empty.
     *
     * Consider how to most efficiently determine if the list contains null data.
     *
     * @param data a list of data to initialize the heap with
     * @throws java.lang.IllegalArgumentException if data or any element in data
     *                                            is null
     */
    public MaxHeap(ArrayList<T> data) {
        if (data == null) {
            throw new IllegalArgumentException("The arraylist is null");
        }
        backingArray = (T[]) new Comparable[2 * data.size() + 1];
        for (int a = 0; a < data.size(); a++) {
            if (data.get(a) == null) {
                throw new IllegalArgumentException("The arraylist contains a null value");
            }
            backingArray[a + 1] = data.get(a);
        }
        size = data.size();
        for (int a = size / 2; a > 0; a--) {
            downHeap(a);
        }
    }

    /**
     * Helper method to build heap by comparing down
     * @param indice index to downheap
     */
    private void downHeap(int indice) {
        boolean flag = true;
        while (indice * 2  <= size && flag) {
            int compare = indice * 2;
            if (indice * 2 + 1 <= size) {
                if (backingArray[indice * 2].compareTo(backingArray[indice * 2 + 1]) < 0) {
                    compare++;
                }
            }
            if (backingArray[compare].compareTo(backingArray[indice]) > 0) {
                T temp = backingArray[indice];
                backingArray[indice] = backingArray[compare];
                backingArray[compare] = temp;
                indice = compare;
            } else {
                flag = false;
            }
        }
    }

    /**
     * Adds the data to the heap.
     *
     * If sufficient space is not available in the backing array (the backing
     * array is full except for index 0), resize it to double the current
     * length.
     *
     * @param data the data to add
     * @throws java.lang.IllegalArgumentException if data is null
     */
    public void add(T data) {
        if (data == null) {
            throw new IllegalArgumentException("The data provided has a null value and cannot be added");
        }
        if (size + 1 >= backingArray.length) {
            int length = backingArray.length;
            T[] tempBackingArray = (T[]) new Comparable[2 * length];
            for (int a = 0; a < length; a++) {
                tempBackingArray[a] = backingArray[a];
            }
            backingArray = tempBackingArray;
        }
        backingArray[size + 1] = data;
        size++;
        for (int a = size / 2; a > 0; a--) {
            downHeap(a);
        }
    }

    /**
     * Removes and returns the root of the heap.
     *
     * Do not shrink the backing array.
     *
     * Replace any unused spots in the array with null.
     *
     * @return the data that was removed
```

```java
     * @throws java.util.NoSuchElementException if the heap is empty
     */
    public T remove() {
        if (isEmpty() == true) {
            throw new NoSuchElementException("The heap is empty and therefore, no max value can be found");
        }
        T remove = backingArray[1];
        backingArray[1] = backingArray[size];
        backingArray[size] = null;
        size--;
        for (int a = size / 2; a > 0; a--) {
            downHeap(a);
        }
        return remove;
    }

    /**
     * Returns the maximum element in the heap.
     *
     * @return the maximum element
     * @throws java.util.NoSuchElementException if the heap is empty
     */
    public T getMax() {
        if (isEmpty() == true) {
            throw new NoSuchElementException("The heap is empty and therefore, no max value can be found");
        } else {
            for (int a = size / 2; a > 0; a--) {
                downHeap(a);
            }
            return backingArray[1];
        }
    }

    /**
     * Returns whether or not the heap is empty.
     *
     * @return true if empty, false otherwise
     */
    public boolean isEmpty() {
        if (backingArray[1] == null) {
            return true;
        }
        return false;
    }

    /**
     * Clears the heap.
     *
     * Resets the backing array to a new array of the initial capacity and
     * resets the size.
     */
    public void clear() {
        backingArray = (T[]) new Comparable[INITIAL_CAPACITY];
        size = 0;
    }

    /**
     * Returns the backing array of the heap.
     *
     * For grading purposes only. You shouldn't need to use this method since
     * you have direct access to the variable.
     *
     * @return the backing array of the list
     */
    public T[] getBackingArray() {
        // DO NOT MODIFY THIS METHOD!
        return backingArray;
    }

    /**
     * Returns the size of the heap.
     *
     * For grading purposes only. You shouldn't need to use this method since
     * you have direct access to the variable.
     *
     * @return the size of the list
     */
    public int size() {
        // DO NOT MODIFY THIS METHOD!
        return size;
    }
}

```