CSC-361
Lab 8

A Comparison of Heap Implementations (Part I)

In this lab we are interested in comparing implementations of two priority queues (heaps). In particular, we are interested in comparing two operations: (1) building a heap and (2) extracting the minimum value.

The MinHeap Interface

Our implementation of min-heaps will follow the `MinHeap` interface below.

```
public interface MinHeap<T>
{
    /* Construct a heap from a set of values and corresponding set of keys */
    public void build(List<T> values, List<Double> keys);

    /* Add to the heap */
    public void insert(HeapNode<T> node);

    /* Remove and return the node corresponding to the minimum key */
    public HeapNode<T> extractMin();

    /* Return the node corresponding to the minimum key */
    public HeapNode<T> peekMin();

    public boolean isEmpty();
    public int size();
    public void clear();
}
```

All classes that implement the `MinHeap` interface must use the `HeapNode` class: a public aggregator class storing data, a priority key, and a (reflection) index of its position in the underlying array implementation.

```
public class HeapNode<T>
{
    public T      _data;
    public double _key;
    public int    _index;    // index in the min-heap
    // ...
}
```

Expandable Heap

Fundamentally, heaps need to be a data structure that have the capability to expand. However, most implementations of a heap come in the form of an array-based implementation; hence, expansion is a problem. For this lab we will use an abstract heap class that implements some of the `MinHeap` operations on an array. For example, the `ExpandableHeapBase` class builds a heap (with `build`) by calling insert for each pair of `<key, value>` pairs; the iterative implementation of the `build` code is shown below.

```
/* A base class for an array-based implementation of a heap.
 * All implementations are naive: no special ordering of nodes based on keys.
 */
public abstract class ExpandableHeapBase<T> implements MinHeap<T>
{
    protected HeapNode<T>[] _heap;
    protected int           _size;
    protected final int     _MIN_CAPACITY = 10;

    public void build(List<T> values, List<Double> keys)
    {
        int sz = Math.min(values.size(), keys.size());
        for (int i = 0; i < sz; i++)
        {
            this.insert(new HeapNode<T>(values.get(i), keys.get(i)));
        }
    }
    // ...
}
```

We expand the heap using code similar to Java API `ArrayList` implementation. In particular, we implement our own `ensureCapacity` method in which we create a new, `150%` larger array and copy the contents of the old array into the new array.

```
/* Increases the capacity of this expandable array-based data structure, if
 * necessary, to ensure  that it can hold at least the number of elements
 * specified by the minimum capacity argument.
 * @param   minCapacity   the desired minimum capacity.
 */
@SuppressWarnings("unchecked")
protected void ensureCapacity(int minCapacity)
{
    int oldCapacity = _heap.length;

    // If the user is checking that the container is large enough already
    if (minCapacity <= oldCapacity) return;

    //
    // Enlarge and copy
    HeapNode<T> oldData[] = _heap;
    int newCapacity = (oldCapacity * 3)/2 + 1;
    if (newCapacity < minCapacity) newCapacity = minCapacity;
    _heap = (HeapNode<T>[])new HeapNode[newCapacity];
    System.arraycopy(oldData, 0, _heap, 0, _size);
}
```

Therefore, defining the `insert` method in a `MinHeap` should always immediately call `ensureCapacity`.

What You Need To Do: Implement Two Min-Heaps

You are to implement two minimum heaps that inherit from the `ExpandableHeapBase` class. The `UnsortedListMinHeap` class is a naïve implementation in which the nodes stored in the underlying array are unsorted. Second, you are to implement a `SortedListMinHeap` class in which the elements in the underlying array are ordered from least to greatest. For your implementation of `SortedListMinHeap`, the insert operation has to be $O(n)$ operation (a la 'insert' in insertion sort). For the `build` operation it suffices to copy and then sort the array (`Arrays.sort`).

What You Need To Do: Timing in Main and Reporting Timing Operations

In a `main` method in a `Main` class, you are to implement code to time two operations (`build` and `extractMin`) in both the unsorted and sorted heaps. Your input to these functions should be in a random order: use `Collections.shuffle` to achieve the desired level of randomness. Report the results of these timings in the header comment of each file. That is, we expect to see a table similar to the following in comments in each corresponding file. In addition, please also report the efficiency in O-notation for each of these operations in both files.

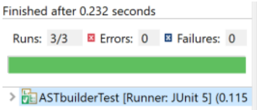
	Build	ExtractMin
5000	1	3
10000	2	6
50000	3	9
100000	4	12
200000	5	15
…		
	O(?)	O(?)

Provided Code

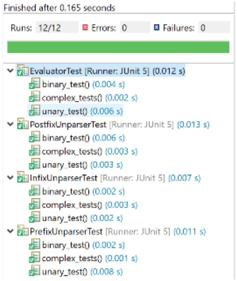
Basic junit testing for min-heaps is provided in the test source code folder. Some utility classes are provided in the `utilities` package. Otherwise, please review the provided heap-based functionality.

JUnit Testing

No output should be produced by your tests; we are seeking only a 'green' output indication in Eclipse. Make sure your screenshot shows success of all junit testing methods and not just the overall summary. For example, the image on the left is bad (and on the right is good).



```
Finished after 0.232 seconds
Runs: 3/3    Errors: 0    Failures: 0
> ASTbuilderTest [Runner: JUnit 5] (0.115
```



```
Finished after 0.165 seconds
Runs: 12/12    Errors: 0    Failures: 0
v EvaluatorTest [Runner: JUnit 5] (0.012 s)
    binary_test() (0.004 s)
    complex_tests() (0.002 s)
    unary_test() (0.006 s)
v PostfixUnparserTest [Runner: JUnit 5] (0.013 s)
    binary_test() (0.006 s)
    complex_tests() (0.003 s)
    unary_test() (0.003 s)
v InfixUnparserTest [Runner: JUnit 5] (0.007 s)
    binary_test() (0.002 s)
    complex_tests() (0.003 s)
    unary_test() (0.002 s)
v PrefixUnparserTest [Runner: JUnit 5] (0.011 s)
    binary_test() (0.002 s)
    complex_tests() (0.001 s)
    unary_test() (0.008 s)
```

Submitting: Source Code

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable String messages output, if failure occurs.

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly into these files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each method should have a comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions; Javadoc comments are ideal. Each *block* of code (3-4 or more lines in sequence) in a function should be commented.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files.

Submitting: Proof of Program Execution

Execute your code and take a screenshot of the associated junit window and output console (with no output). Place these screenshots into a word processing document (Word, OpenOffice, GoogleDocs, etc.). If multiple screenshots are necessary, label each clearly. Please make sure to crop and enlarge the screenshots so that the picture and / or text is clear (and doesn't strain my old eyes).

Create a PDF of this document and call it `evidence.pdf`.

Source Code Files

Please submit only `Main`, `UnsortedListMinHeap`, and `SortedListMinHeap`.

Final Submission File

In a folder named `lab`, place (1) the source code files and (2) `evidence.pdf`. Zip folder `lab` and label that zip file as `lab.zip`. This zip file is to be submitted via Moodle.

Please be reminded that following instructions explicitly and submitting well-formatted documents (with consistent fonts and typeset equations) is an important part of professionalism.

Only one person in your pair needs to submit the final product.