Lab 11 — Heaps of Fun

Data Structures and Algorithms – Spring 2024

---

1. Download the files for today's lab (**lab11_files.zip**) from the class web page, and open **Heap.java**. This is the skeleton of a parameterized class for representing heaps of comparable elements. Your task is to complete the implementation. A Heap object stores its elements in an internal ArrayList<E>, where E indicates the element type. The constructor and toString() methods are already provided for you. Notice that Heap extends the class HeapTester<E>. This precompiled class contains code that will automatically test your Heap methods to verify that they're working correctly.

2. I've also provided a pre-compiled program HeapDemo that generates animated demos of inserting and removing elements from a Heap. In this case, the elements are Strings, which are compared alphabetically. The program shows the binary-tree representation of the Heap in a popup window, and the corresponding array of elements in the DrJava interactions window. To run the program, simply type `java HeapDemo` at the DrJava prompt and follow the instructions.

3. You should implement each of the **Heap** methods in the bulleted list below, in the order shown. There are two different ways to test your code. The first option is to use the **InteractiveHeapTest.java** program to test your heap methods interactively. This program displays a visual representation of a heap in a window, using a HeapViewer, which supports the following methods:

hv = new HeapViewer()	constructs a new heap display window, which can be resized interactively
hv.draw(*heap*)	draws a Heap<String> in the window
hv.verify(*heap*)	verifies that the Heap<String> elements satisfy the heap property
hv.randomHeap(*num*)	returns a new Heap<String> containing *num* randomly chosen strings
hv.close()	closes the window
hv.setFontSize(*fontSize*)	changes the size of the font used to display the heap elements

   Currently, if you run InteractiveHeapTest, it will not update the heap as expected, since most of the methods of the Heap class are not yet implemented.

   The second way to test your Heap methods (all at once) is to simply execute the **Heap.java** file directly by clicking *Run* in DrJava or typing `java Heap` at the command line. The Heap class inherits its `main` method from the HeapTester<E> superclass, so you should NOT add your own `main` method to Heap, because it would override the tester program. When you run Heap, the tester program checks each Heap method and reports the first problem it encounters, showing the results that were expected and the results that actually occurred. Here are all of the methods that you'll need to implement:

- **int heapSize()** should return the number of elements currently in the heap. Remember that index position 0 is not used, so the size of the heap is one less than the size of the ArrayList.

- **boolean isEmpty()** should return true if the heap is empty, or false otherwise.

- **E peek()** should return the smallest element in the heap, without removing it, or `null` if the heap is empty.

- **E getElement(int *i*)** should return the element at index position *i* in the heap.

- **int lastIndex()** should return the index position of the last element in the heap.

- **int parentIndex(int *i*)** should return the index position of the parent of element number *i*.

- **int leftChildIndex(int *i*)** should return the index position of the left child of element number *i*.

- **int rightChildIndex(int *i*)** should return the index position of the right child of element number *i*.
- **boolean hasLeftChild(int *i*)** should return true if element number *i* has a left child, or false otherwise.
- **boolean hasRightChild(int *i*)** should return true if element number *i* has a right child, or false otherwise.
- **E getParent(int *i*)** should return the parent element of element number *i*.
- **E getLeftChild(int *i*)** should return the left child element of element number *i*.
- **E getRightChild(int *i*)** should return the right child element of element number *i*.
- **void swap(int *i*, int *j*)** should swap the heap elements at positions *i* and *j*.
- **boolean hasSmallerChild(int *i*)** should return true if element number *i* has at least one child that is smaller than element number *i*.
- **int indexOfSmallestChild(int *i*)** should return the index position of the smallest child of element number *i*, assuming that element *i* is not a leaf. If both child elements are present and equal to each other, the index position of the *right child* should be returned.
- **void add(E *newElement*)** should add *newElement* to the heap, modifying the arrangement of elements to restore the heap property if necessary. Here is an outline of the insertion process:

```
add the new element to the end of the heap
set the current position to be the new element's position
while current position is not the root and current element < parent element:
    swap the current and parent elements
    update the current position to be the parent position
```

- **E remove()** should remove and return the smallest element from the heap, modifying the arrangement of elements to restore the heap property if necessary. Here is an outline of the removal process:

```
if the heap contains just one element:
    delete and return the root element
else
    retrieve the smallest element from the root
    delete the last element from the heap and put it at the root
    fix the heap by pushing the new root element down the tree as follows:
        set the current position to be the root position
        while the current element has a smaller child:
            choose the smallest child of the current element
            swap the current element with its smallest child
            set the current position to be the smallest child's position
    return the smallest element
```