Heap

Heap is a binary tree where each node has at most two children. This binary tree is complete, which means that except the last level of the tree, each level is filled with all possible nodes. The last level is filled from left to right. The arrangement of nodes in the complete binary trees specifies the structure property of heaps.

The min-heap is a complete binary tree where the value at each node is smaller than (or equal to) the values stored in either child. The max-heap is a complete binary tree where the value at each node is larger than (or equal to) the values stored in either child. The hierarchy of elements in min-heap and max-heap specifies the ordering property of heaps. For example, the following tree shows the structure and ordering of values for a max-heap:

Labeling of Nodes

To identify nodes in a heap, they are numbered sequentially level-by-level starting from the root node. At each level, nodes are numbered left to right as shown in the following figure:

Weiss’s codes uses the above labeling for operations applied on heaps. Each numeric label denotes the index of an array that represents a heap. For instance, label 1 denotes index 1 of that array. For the element at index 0, Weiss suggests filling that element with -∞ for min-heap
and $+\infty$ for max-heap.

There’s a relation between the label of a node and the labels of its children: for a node with label $i$, the left child has label $2i$ and the right child has label $2i + 1$ (see figure below):

![Node Labels Diagram](image)

### 0-based Labeling of Nodes

If the nodes are labeled from 0 rather than 1, the heap looks as follows:

![Heap Diagram](image)

Using the 0-based labeling, what are the labels of child nodes if $i$ is the label of the parent node as shown below?

![Child Nodes Diagram](image)

With the 0-based labeling and the structure property of heaps, the nodes of a heap are placed into an array that is indexed by the labels of the nodes.

### Insertion into a Heap

To insert an element in the heap, the element is placed into the bottom level of the tree as left as possible. Then, the element is moved up the tree until the heap ordering property is maintained.

Insert the values 30, 55, and 95 in the heap shown at the beginning section of this lab. Show the values of nodes after each insertion.
Deletion from a Heap

To delete a node from the heap, the root of the tree is removed and is re-placed by the rightmost element at the bottom level of the tree. Then, the element is moved down the tree until the heap ordering property is maintained.

Remove the first two elements from the original heap shown at the beginning section of this lab. Show the values of nodes after each deletion.

Constructing Heap from an Array

Suppose an array of elements that are placed into a complete binary heap but do not maintain the heap ordering property. The ordering property is reinstated by checking the nodes starting from the non-leaf node, which has the largest label in the heap, up to the root node. Each node is moved down the tree until the ordering property is maintained.

For the following questions, assume that elements are placed into a complete binary tree with the 0-based labeling. Also, assume that the ordering property is reinstated for a min-heap.

a. Suppose an array of six elements $a[6]$ is initialized with $a[i] = 10 - i$, where $i$ starts from zero. First, place the array elements into the complete binary tree. Then, reinstate the ordering property on the elements.

b. How should you initialize the above array so that none of the tree nodes needs to move down the tree?

The percolateDown Method

Create a project in Eclipse. Inside the project, create a package called heapsort. Import HeapSort class in the following directory path on MathLAN to the project.

/home/hajiamini/courses/CSC207/code/

The following code is taken from page 818 in Weiss:
The above code moves down nodes in a min-heap using the 1-based node labeling. Besides the relation between indices of children and their parent, the 0-based labeling impacts the relation between the size of heap and the index of the last node of the heap.

Modify the `percolateDown` method for applying to the 0-based heaps. Before modifying the method, answer the following questions. (To answer these questions, you may refer to Deletion from a Heap exercise to run the `percolateDown` method on a new root node after deleting some old root from the heap.)

a. Why `array[hole]` is stored in `tmp`?
b. What does the following test condition of the for-loop do?

```
hole * 2 <= currentSize
```

c. What is the purpose of the following assignment statement?

```
child = hole * 2;
```
d. What is it meant if the following test condition of the if-statement is `false`?

```
child != currentSize
```
e. What does `child++` do?
f. What do the two calls to the `compare` method accomplish?
g. Is the element stored in `array[hole]` lost after executing the following statement?

```
array[ hole ] = tmp;
```
h. What does the `break` statement imply?
Now, open the `HeapSort` class file:

a. `HeapSort` class contains several methods. `heapSort` is the most important method, which is shown on page 826 of the textbook. Review the `heapSort` method to understand how it works.

b. Read the comment and signature of the `percDown` method to understand how it should work.

c. Using the answers to questions a–h and the original implementation of the `percDown` method, implement a new version of this method. This implementation:

   - Uses a max-heap, and
   - Uses the 0-based labeling for manipulating the heap array elements.

Note that you need to use `compareTo` instead of `compare` when comparing elements inside the `percDown` method.

(4 points) Test the new implementation using the main method of `HeapSort` class.

You may find it easier to test the implementation with a smaller heap, e.g., a heap of 7 nodes, and use the Eclipse debugger tool to inspect values for `hole`, `child`, and elements that are swapped together.

Acknowledgement

This lab is based on a material made by Henry Walker.