Algorithms & Data Structures
SI3 - Polytech Nice-Sophia - Edition 2018

Lab #5: Priority Queues, Heaps

This assignment will give you practice about priority queues and heaps. In this assignment, you can use some provided test class for interactive testing.

Part 1: write up questions

- Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap
- Show the result of using the linear-time algorithm to build a binary heap using the same input
- Show the result of performing three `deleteMin` operations in the heap of the previous question
- Show the following regarding the inverse extremum item in the heap (that is the minimum for a max-heap or the maximum for a min-heap):
  - It must be at one of the leaves
  - There are exactly N/2 leaves
  - Every leaf must be examined to find it

Part 2: binary heap implementation

Download the file Lab5.py and put it in the suitable package in your IDE project.

In this part you have to implement a binary heap class following what we learned during lecture but with the new following rules:

- your binary heap must be parametrized by the ordering on the elements (b`key`)
- you are using a list, i.e. the first index 0 will be the index of the extreme element.

Complete the provided class template `BinaryHeap` by writing all the methods

Part 3: deleting in a binary heap

Assuming values in a binary heap are all distinct, we want to implement the *delete* operation which deletes a given element in the heap (not necessary the extreme element).

- give a lower bound for the complexity of the *delete* operation
- write the method `delete` in the same class `BinaryHeap` as in part 2
- give the detailed worst case complexity of the `delete` method

We now assume that the elements in the heap are not unique and it is likely that the heap contains many occurrences of the element to delete.

- give a different method to delete all the occurrences of a given element in the heap

- write the method `deleteAll` in the same class `BinaryHeap` of part 2

Part 4: dynamic median-finding

Given a finite set of comparable values, the *median* of this set is the value separating the higher half of the set from the lower half. For example, the *median* value of the set { 25, 56, 13, 61, 48, 79, 37 } is 48 and the median of { 1, 2, 3, 4} is 3.

- design a data type that supports the following operations:
  - *__len__*: gives the current number of elements. Complexity must be constant (i.e. in Θ(1))
  - *add*: adds a new element in the data structure. Complexity must be in O(log(N)) where N is the number of elements currently in the data structure
  - *median*: returns the median of the element currently in the data structure. Complexity must be constant (i.e. in Θ(1))
  - *pop_median:* returns and deletes the median of the element currently in the data structure. Complexity must be in O(log(N)) where N is the number of elements

This can be achieved by using **two heaps**, a max-heap to store the values *less than or equal to* the median, and a min-heap to store the values *greater than or equal to* the median. In that way, the median value is always the extreme element of the max-heap!

- give the implementation of this data type by completing the provided class template `DynamicMedian`

Part 4: *d*-heaps

A simple generalization of the binary heap is a *d*-heap, which is exactly like a binary heap except that all nodes have *d* children (thus, a binary heap is a 2-heap).

- give the height of a *d*-heap containing *N* elements
- discuss the pros and cons of *d*-heaps versus 2-heaps
- write all the methods in a class named `DHeap`: as far as the implementation is concerned, a *d*-heap is just like a binary heap, except it has a new attribute *d* (the number of children of nodes).