Assignment 4: Heap Sort

Date: Februari 28th 2018
Deadline: March 7th 2018 23:59

Objectives

You must implement an object-oriented `Heap` implementation in combination with the heap-sort algorithm.

Requirements

You are left relatively free, regarding your implementation, as long as it maintains the following requirements:

- The structure consists of at least a `Node` class and a Max `Heap` class, where the `Heap` class uses said `Node` classes for its innner structure. The contents of this `Heap` class are inspired by the wikipedia page on `Heap_(data_structure)`, which you can find at the bottom of this document. These classes should at least have the following functionality. How you implement them exactly is mostly up to you:
  - Node (heap.py):
    - Can store its parent
    - Can store its children (order is important, and a maximum of 2)
    - Can store its value
    - Can be easily compared to other Node objects or numbers using all comparison operations. For example:

```
if node1 < node2:
  print("I can check against fellow Node objects!")
if node1 >= 20:
  print("I can check numbers too")
```

  - Heap (heap.py):
    - Stores the head node (you can obviously store more if you want to)
    - Minimally has the following functions (**hint** you are free to create many more to help you):
      - find_max(self): Finds the maximum item of the `Heap` (a.k.a. `peek`)
      - insert(self, number): Adds a new element to the `Heap`, and preserves the `Heap` properties (a.k.a. `push`)
      - extract_max(self): Returns the node with the maximum value from the `Heap`, after removing it (a.k.a. `pop`)
      - build_heap(input_list): Takes a list of elements and generates a single `Heap` from these, using a series of `insert` commands. (Although a more efficient version is possible, with an object-oriented approach this is significantly more difficult to implement, therefore the less efficient version with complexity O(n log n) is allowed).
      - size(self): Returns the current size of the `Heap`
      - is_empty(self): returns true if the `Heap` has zero elements, else False
      - delete(self, node): Deletes a node from the `Heap` (in the proper way)

    - __sift_up(self, node): Move a node up in the tree, as long as needed.
    - __sift_down(self, node): Move a node down in the tree, as long as needed.
    - __find_second_to_last_tail(self): Starting from the `tail_node` of the `Heap`, finds the second-to-last node in the `Heap`. It will be necessary to update the tail_node when you remove the current tail. (**hint**: You can always find the tail_node by iterating through the `Heap`, but in many cases it might be possible that there is a quicker way, by starting from the current tail_node.)
    - __find_first_free_parent(self): Starting from the `tail_node` of the `Heap`, finds the first parent that has the open slot that should receive a new child. It will be necessary to find this parent when you insert a new node on the heap. (**hint**: The hint from the previous line might still be relevant here, although it now applies to the parent instead of a new tail_node.)
- Does NOT maintain a list of all contained nodes. Using the array-representation for the `Heap` is also explicitly forbidden.
- When the `Heap` is printed, it prints out the values of its elements in breadth-first order. For example take the following drawing:

```
   10
   /\
  9  8
 / \ /
1  2 3
>> print(heap1)
10 9 8 1 2 3
```

Details on the input and output formats

- The heap-sort algorithm, should take a `Heap`, and return the sorted output. The original content of the `Heap` does not have to be preserved. The sorting algorithm takes a sequences of integers separated by spaces as command-line input, and print the sorted sequence (separated by spaces).

  Example:

```
$ python3 heap_sort.py 3 3 2
2 3 3
$ python3 heap_sort.py 3 5 4 1 2 9 10 20
1 2 3 4 5 9 10 20
```

- Spaces can be safely ignored

  Example:

```
$ python3 heap_sort 4  5   3 2
2 3 4 5
```

You must submit your work as a tarball [1]. Next to the source code, your archive must contain a text file named “AUTHORS” containing your name and Student ID.

Getting started

1. Download the tarball, that includes the checks, these instructions, and the empty python files you will fill as part of the assignment.

2. Understand what a `Heap` is, and how it works. On internet you find many `array` implementations, you can not use these. Your implementation must be object-oriented.
3. Familiarize yourself with the heap-sort algorithm. Also consider this algorithm when designing your `Heap` class.
4. Write a bunch of valid and invalid expressions to serve as test input for your program and add these to `check_heap_sort.py`.

Testing

A small set of tests is provided for the `Heap` sort algorithm. The file `check_heap_sort.py` contains a basic set of testcases for the sorting algorithm.

The test sets provided are not complete. Add your own tests to make sure that your implementation of the data structure and the algorithm is correct.

Grading

Your grade starts from 0, and the following tests determine your grade:

- +0,5pt if you have submitted an archive in the right format with an `AUTHORS` file.
- +0,5pt if your code runs without errors and you have made an effort for the `Heap` and the sorting algorithm.
- +1pt Node class contains all the required functionality, and works properly.
- +1pt `find_max` , `insert`, `extract_max`, `size`, `build_heap`, and `is_empty` work as intended.
- +1pt `__sift_up` Works as intended
- +1.5pt `__sift_down` Works as intended
- +1.5pt `delete` Works as intended
- +0.5pt The `Heap` has a proper constructor (essentially the `create-heap` command from wikipedia)
- +0.5pt `__find_second_last` function works efficiently (you can do it in N, but there are smarter ways)
- +0.5pt Heap_sort algorithm works with your object-oriented `Heap` approach.
- +1.5pt Passes all the command-line test-cases that TAs came up with.
- -2pt if you used any python libraries to complete the `Node`, `Heap`. For the heap-sort only `argparse` would be allowed.
- -10pt if you used the built-in sort function or priority-queue, or any similar built-in function that does the same.

See also

- Heap https://en.wikipedia.org/wiki/Heap_(data_structure)
- Heapsort (Array-notation)

  https://en.wikipedia.org/wiki/Heapsort

  https://www.geeksforgeeks.org/heap-sort/

1 http://bfy.tw/Gdfn