CS/SE 2XB3 — Lab 5

Due Friday, Feb 26th, 11:59pm

Submit the lab on Avenue. Note, Avenue must receive your lab by the due date. **Do not leave your submission to the last minute! Late submissions, even by seconds, will not be graded.** You have been warned.

This lab is worth (59/7)% of your final grade in the course. Read this document carefully and completely. **Whenever I ask you to discuss something, I implicitly mean to include that discussion in your lab report.** Include all timing experiments in your `code.py` file.

Purpose

The goals of this lab are as follows:

1. Implement and compare different ways to build heaps.
2. Explore a k–heap implementation.

Submission

You will submit your lab via Avenue. You will submit your lab as three separate files:

- code.py
- heap.py
- k_heap.py
- report.pdf (or docx, etc.)

Only one member of your lab group will submit to Avenue – see the section below for more details on that. Your report .pdf will contain all information regarding your group members, i.e. name, students number, McMaster email, and enrolled lab section. This will be given on the title page of the report. For the remainder of this document, read carefully to gauge what other material is required in the report. Your report should be professional and free from egregious grammar, spelling, and formatting errors. Moreover, all graphs/figures in your report should be professional and clear. You may lose grades if this is not the case. Organize the report in a such a way to make things easy for the TA to grade. You are not doing yourself any favors by making your report difficult to mark!

Building Heaps [50%]

Depending which source you read/what lectures you go to, you may have different terms for the same heap methods. Heapify, bubble-up, and swim all refer to the same procedure. Bubble-down and sink refer to the same procedure. In your 2C03 lectures, I believe you are using sink/swim.

See the `heap.py` file posted with this lab. You will note that the `build_heap` (1, 2, and 3) methods are left unimplemented. You will explore three different ways to build a heap and compare them to each other. Remember, initially, a heap is arbitrary list of values. To build a heap, we must take these arbitrary values and ensure they have the heap properties. Below are three ways you will build a heap.

1. `build_heap_1` Bottom-up: This is the method we saw in lecture. It exploits the fact that we know all leaves (roughly half of the heap) are already "heaps". It proceeds by working backwards (bottom-up) and calls sink on all remaining nodes.

2. `build_heap_2` Brick-by-brick: We know we can insert a single value into a heap. So start with an empty heap, and build the heap by inserting one value in the list at a time.

3. `build_heap_3` Sink-top-down: In lecture, we saw that calling sink/heapify on every node does not guarantee that the resulting structure is a heap. But, informally, it does make it more "heapish". Implement `build_heap_3` such that it calls sink on every node (in a top down fashion), then checks to see if the resulting structure is a heap, if it isn't a heap yet, it calls sink on all the nodes again. To implement this method, you should also implement a helper method `is_heap` which returns a boolean value.

From 2C03 you should have seen that sink and insert both have worst case $O(\log n)$ complexity. Knowing that, make an estimation on the complexity of the three methods above. Discuss your reasoning in your report. You do not need to comment on small performance details (like which one is better) just overall asymptotic behaviour.

Now, using timeit, run some experiments to gauge the performance of the three methods. Do your findings agree with what you originally estimated? Discuss why this may or may not be the case. Use the data to verify/justify your claims. Put all experiment related code in your code.py file.

For `build_heap_3`, can you identify an element of the implementation that causes poor performance? Could you think of a way to improve it? Hint: what would be the maximum number of times you need to heapify every node before the whole thing is a heap?

k–Heap [50%]

For this section you will partially implement a k-heap. As you may suspect a k–heap is similar to a traditional heap but instead of nodes having up to two children, each node has up to k

children. All heap properties still hold! That is, the parent is greater than or equal to all its children, and the heap is built in a "complete" fashion.

In the `k_heap.py` file, partially complete the class by implementing the following methods. All methods are similar to that of a traditional heap, with of course the exception of up to k children:

- `__init__(self, values, k)`: Initializes a k–heap from a list of values. Assume $k > 1$.
- `build_heap(self, values)`
- `parent(self, i)`: Returns the index of the parent of the node at i
- `children(self, i)`: Returns a list of all the indices of the children of the node at index i.
- `sink(self)`: Same as the sink methods you have seen before, but with k children instead of 2.

You do not need to run any experiments on your implementation. However, discuss the advantages/disadvantages of using a k–heap instead of a traditional one. In your discussion state what you believe the asymptotic complexity of sink to be. You do not need to give a formal proof of your claim, but justify it with a reasonable explanation.