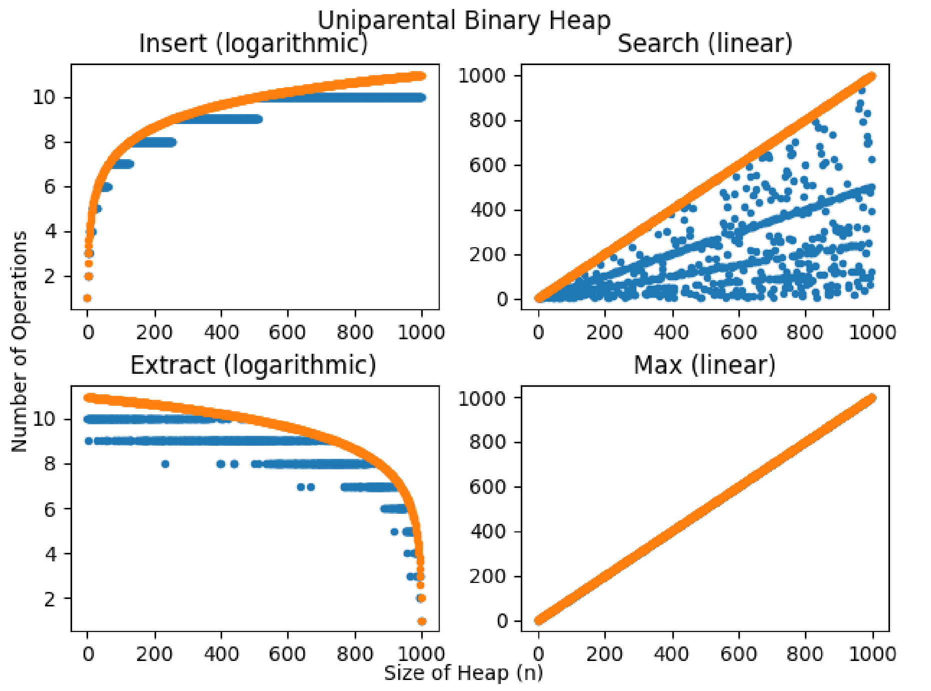
Kyle Maclean
Priority Queues
December 2021

1 Description of implementation

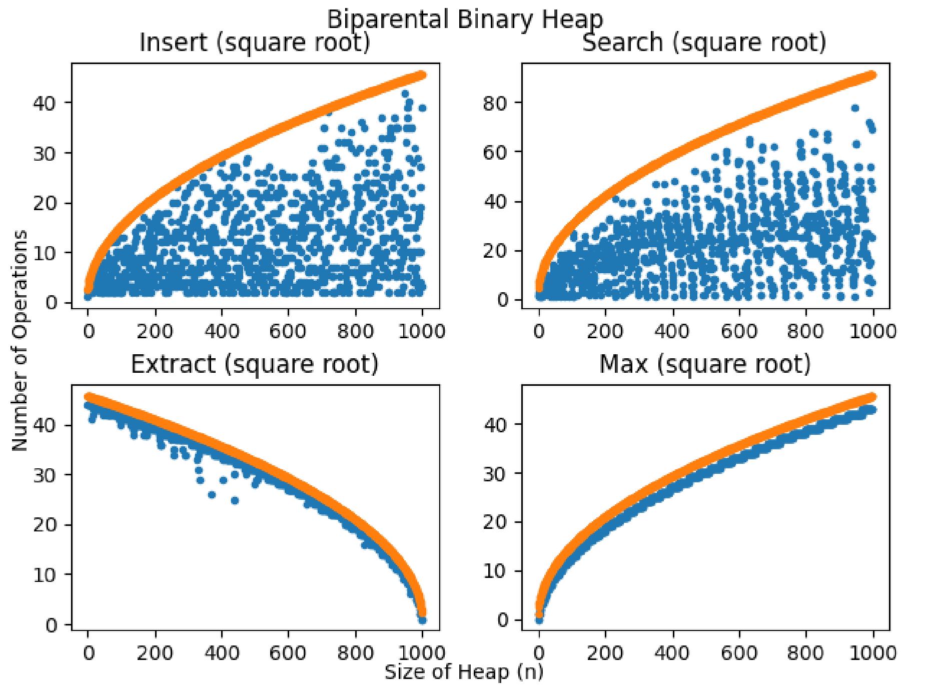
The code is presented in a single Python file, `pq1.py`. The user-facing functions are the first to be defined, and all of their helper functions are defined below all of them. For efficiency, functions that manipulate the heaps operate in-place. Therefore, they do not return a new heap, so the user must just continue to use the same heap that was passed in as argument to the manipulation function.
We define the two datastructures unambiguously as *uniparental binary heaps* and *biparental binary heaps*. Since there are similarities between them, they are made to share the functions which operate on them to reduce code duplication. The `min` function is identical for both datastructures, but for the other four, a parameter, `parents` must be passed to them with a value either of `1` or `2` to describe how to interpret the heap (as uniparental or biparental, respectively) and how to perform the operation.
The user-facing functions are wrappers for functions which also return a `time` value, which represents the number of recursions or iterations taken to perform the operation (used during the efficiency tests). The user-facing functions discard this value in order to provide a simple interface.

2 Results of testing

Each of the five operations has a correctness and efficiency test that can be found in `pq2.py`. Each of these tests repeats the same procedure for each of the parent-types by parametrically assigning appropriate arguments and specialised functions.
The correctness checks all assert expected outputs which have been constructed by hand. The validity checks are performed using the `is_valid` function from `pq1.py`, which itself is tested in `pq2.py` using hard-coded, hand-validated heaps. The correctness checks for `insertion` and `extraction` also use randomly-generated heaps in addition to the hand-crafted test cases.
The efficiency tests assert that the time taken for each operation is less than or equal to the appropriate function on the heap size, which are defined in `get_time_bound`. The function `test_plot_performance` generates the graphs that are displayed in Figure 1. The procedure it follows is to insert 1,000 random numbers into the two kinds of heaps. After every insertion, a search is performed to find the newly-inserted key. This allows an insertion with the appropriate number of ascending swaps to get the inserted element to the correct position and a search with the appropriate number of "steps" along the "columns" and "rows" of the upper-left triangular matrix in the case of biparental binary heaps, or, simply checking every value linearly in the case of a uniparental binary heap to find the key - of the heap at every size up to 1,000. Then, we find the maximum of every heap created from removing the last key 1,000 times. This allows checking for the maximum in a heap of every size up to 1,000. Then, we extract the minimum and merge the resulting sub-heaps to allow extracting from heaps of every size from 1,000 down.



(a) Note that the `max` function's empirical performance follows the bound tightly (the blue dots are underneath the orange dots).



(b) Note that the scale of the `search` function's y-axis is double that of the `insert` and `extract` functions'. This is because the number of comparisons required is bounded by twice the square root of the number of elements. Of course, this constant factor is ignored when expressing asymptotic complexity with Big-Oh.

Figure 1: Plotting the performance of four operations for the two different parent-types of binary heaps. The `min` function is excluded because it is constant time for both types. The blue scattered dots represent randomised empirical measurements of the number of recursions/iterations that were performed on heaps of different sizes. The orange scattered dots represent the theoretical Big-Oh bound on the time the operations should take for heaps of given sizes.