**Lab 4: Heaps**

**Learning Outcomes**

* Implement sorting algorithms such as heap sort
* Apply asymptotic time complexity analysis to choose among competing algorithms

**Overview**

Heaps are data structures for efficiently tracking the smallest or largest item in a collection. Heaps can be repurposed for sorting. Once a heap is implemented, a collection of elements can be sorted by (1) adding every element to the heap and (2) pop each element off of the heap and add to a list. Since the pop operation of the heap will return the smallest (or largest) element of the remaining elements, the elements will be returned in sorted (reversed sorted) order.

You are going to implement a binary max-heap data structure and then use it to implement heap sort.

The binary max-heap data structure and the heapsort algorithms are described in Chapter 6 of *Introduction to Algorithm*s, 3rd ed. by Cormen, Leiserson, Rivest, and Stein.

**Implement a Binary Max-Heap**

You have been provided the file test\_heap.py, which contains unit tests designed to help you check the correctness of your implementation. In the file heap.py

1. Implement the following functions:

* void build\_max\_heap(A)
* void heap\_extract\_max(A)
* void max\_heap\_insert(A, key)
* void heapsort(A)

and the associated helper functions:

* void \_heap\_increase\_key(A, i, key)
* void \_max\_heapify(A, i)

You can test your code by running:

$ python test\_heap.py

**Benchmark Your Max Heap**

Next, you are going to benchmark your heap. In particular, you are going to compare insertion and deletion times for sorted, reverse sorted, and randomly-shuffled lists of numbers.

2. Create a Jupyter notebook and name it **lastname\_lab04**. The notebook should have a title, your name, and an introduction.

3. In the notebook, import the heap, matplotlib, numpy, random, and time modules.

4. **Evaluating Impact of List Order on Performance.** Benchmark inserting elements into your max heap by creating heaps from lists of 100; 1,000; 10,000; and 100,000 numbers. Create the lists from random numbers using random.random(). Use random.shuffle() to generate a random permutation of the list and the list.sort() method to sort the list in order and reverse order (using the reverse keyword). Run each benchmark 10 times (trials). Create the heap using the void build\_max\_heap(A) function.

Use Matplotlib to plot the list sizes versus the run times using a line plot. There should be one line for each heap creation scenario (randomly permuted, sorted, and reverse sorted). Label each line and create a legend. You may need to take the log10 of the list sizes to get a nice graph.

5. **Evaluating Impact of Building Heap in One Step vs Inserting Each Element.** Benchmark inserting elements into your max heap by creating heaps from lists of 100; 1,000; 10,000; and 100,000 numbers. Create the lists from random numbers using random.random(). Use random.shuffle() to generate a random permutation of the list and the list.sort() method to sort the list in order and reverse order (using the reverse keyword). Run each benchmark 10 times (trials). Create the heap by calling void max\_heap\_insert(A, key) on each element.

For each scenario, create a plot of the run time of using build\_max\_heap(A) and max\_heap\_insert(A, key) (3 total plots). Use Matplotlib to plot the list sizes versus the run times using a line plot. There should be one line for each function. Label each line and create a legend. You may need to take the log10 of the list sizes to get a nice graph.

6. **Evaluating Run-Time of Extraction vs Insert.** Benchmark extracting the max elements from a heap using lists of 100; 1,000; 10,000; and 100,000 numbers. Create the lists from random numbers using random.random(). Use the void build\_max\_heap(A) function to create the heaps. Then, benchmark calling void heap\_extract\_max(A) until the heap is empty. Run each benchmark 10 times (trials).

Create a plot of the run times from calling max\_heap\_insert(A) with lists of randomly-permuted numbers and heap\_extract\_max(A) to extract each element. Use Matplotlib to plot the list sizes versus the run times using a line plot. There should be one line for each function. Label each line and create a legend. You may need to take the log10 of the list sizes to get a nice graph.

**Analyze Heap Operation Run Times**

Heapsort uses a heap to sort a list of numbers. Each element is added to the heap. Once the elements have been added, each element is removed in turn. The elements will be returned in largest to smallest size and can be placed in order in the list in an appropriate fashion.

7. Answer the following questions in your notebook:

1. Does the heap creation run time vary between the three scenarios? Which cases are the best-, average-, and worst-cases?

2. Why do you think the best case is faster than the worst case? (Hint: think about how a single element is added to the heap and the process for "bubbling" that element up or down to maintain the heap invariant property.)

3. Which is faster: building the entire heap at once or inserting one element at a time? Why?

4. Is extracting an element from or inserting an element into the heap faster? Why?

I will be will looking for the following:

* That the tests pass
* An introduction (including your own statement of the problem) and a written summary of your results at the top of the notebook in Markdown. Make sure to put your name at the top of the notebook.
* That your line plots look reasonable.
* Reasonable attempts to answer the questions.

**Submission Instructions**

Save the notebook as an PDF named **lastname\_lab04.pdf**. Submit a zip file containing your heap.py file and your PDF through Canvas.

**Rubric**

|  |  |
| --- | --- |
| Followed submission instructions | 10% |
| Heap implementation passes tests | 30% |
| Correctly ran the 3 benchmarks | 30% |
| Plot of the run times | 10% |
| Answers to questions | 20% |