schulz_lab04

April 17, 2021

1 Lab 4: Heaps

David Schulz

1.1 Introduction

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.

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

1.1.1 Summary

We learned how to implement the various operations involved with heaps, as well as how their best-, average-, and worst-case inputs differed and affected the run-times of building heaps, inserting new values, and extracting the max values.

1.2 Evaluating Impact of List Order on Performance

[1]:
```
import heap as hp
import matplotlib.pyplot as plt
import numpy as np
import random
import time

list_sizes = [100, 1000, 10000, 100000]
numbers = []

for i in range(0, 100000):
    numbers.append(random.random())

sort = numbers[:]
sort.sort()
reverse = numbers[:]
```

```
reverse.sort(reverse=True)
```

[2]:
```
def bench_build(lst, n_trials, n_nums):
    times = []
    for i in range(n_trials):
        start = time.perf_counter()
        lst2 = lst[:n_nums] # copy the input list so that we don't modify it
        hp.build_max_heap(lst2, n_nums)
        end = time.perf_counter()
        elapsed = end - start
        times.append(elapsed)
    return np.mean(times)
```

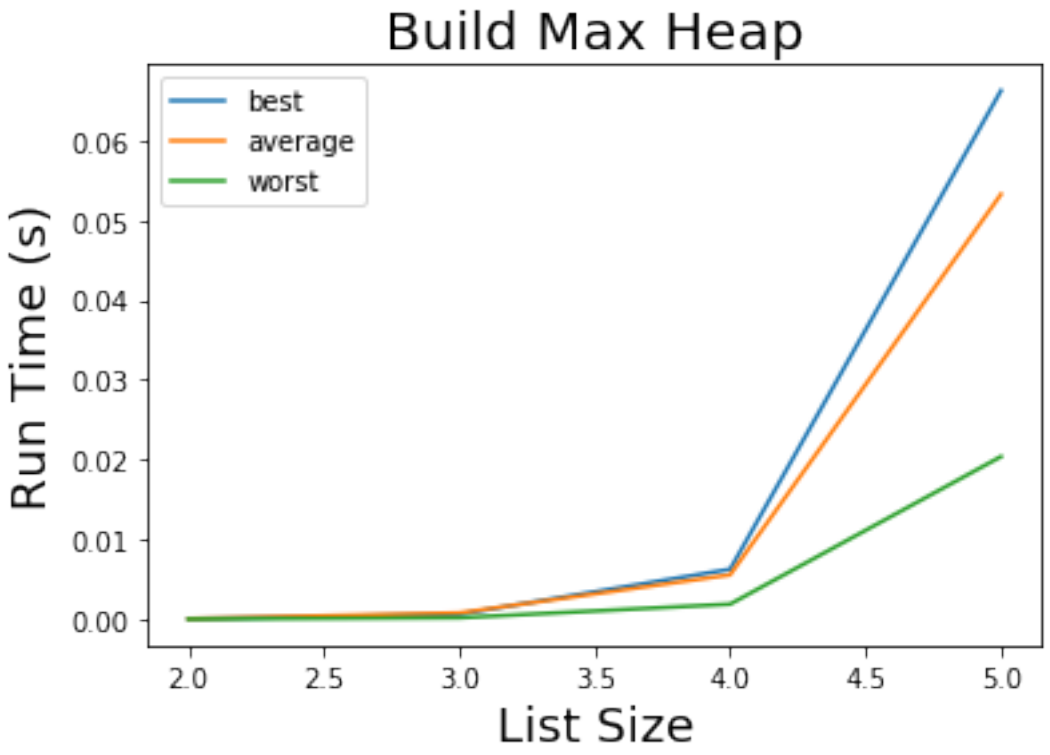
[3]:
```
build_shuffle = []
for size in list_sizes:
    build_shuffle.append(bench_build(numbers, 10, size))

build_sorted = []
for size in list_sizes:
    build_sorted.append(bench_build(sort, 10, size))

build_reverse = []
for size in list_sizes:
    build_reverse.append(bench_build(reverse, 10, size))
```

[4]:
```
plt.plot(np.log10(list_sizes), build_sorted, label="sorted")
plt.plot(np.log10(list_sizes), build_shuffle, label="randomly permuted")
plt.plot(np.log10(list_sizes), build_reverse, label="reverse sorted")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Build Max Heap", fontsize=20)
plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x7f20046f8f60>
```



1.3 Evaluating Impact of Building a Heap in One Step vs Inserting Each Element

[5]:
```
def bench_insert(lst, n_trials, n_nums):
    times = []
    for i in range(n_trials):
        start = time.perf_counter()
        lst2 = lst[:n_nums] # copy the input list so that we don't modify it
        heap = []
        for value in lst2:
            hp.max_heap_insert(heap, len(heap), value)
        end = time.perf_counter()
        elapsed = end - start
        times.append(elapsed)
    return np.mean(times)
```

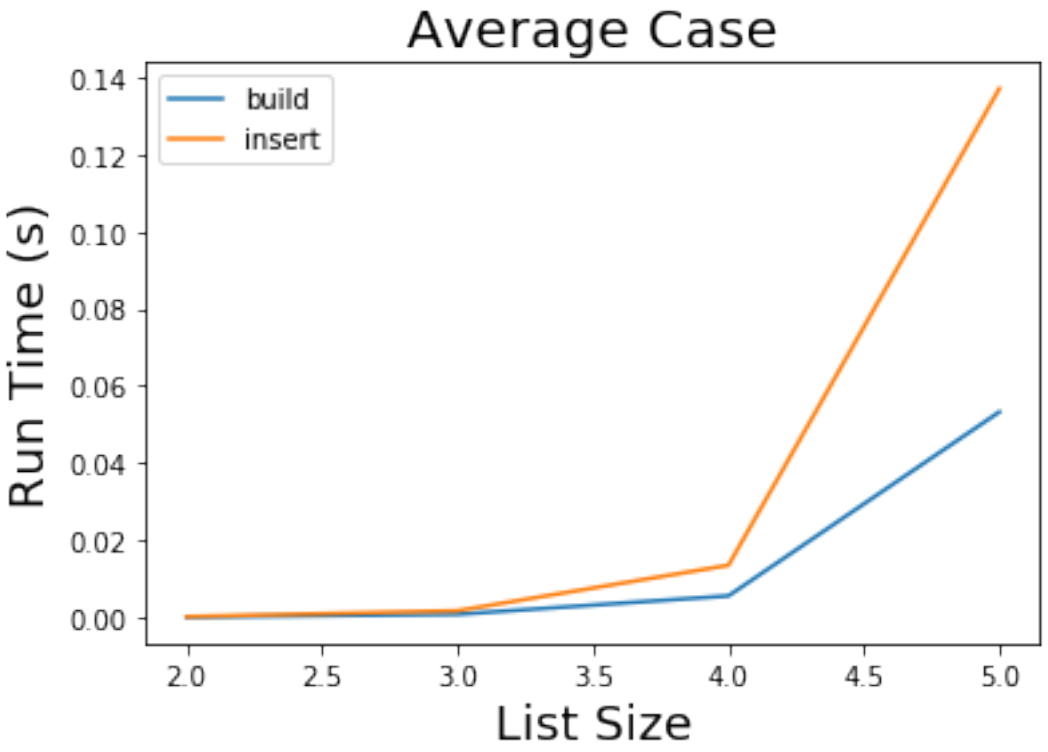
[6]:
```
insert_shuffle = []
for size in list_sizes:
    insert_shuffle.append(bench_insert(numbers, 10, size))

insert_sorted = []
for size in list_sizes:
    insert_sorted.append(bench_insert(sort, 10, size))
```

```
insert_reverse = []
for size in list_sizes:
    insert_reverse.append(bench_insert(reverse, 10, size))
```

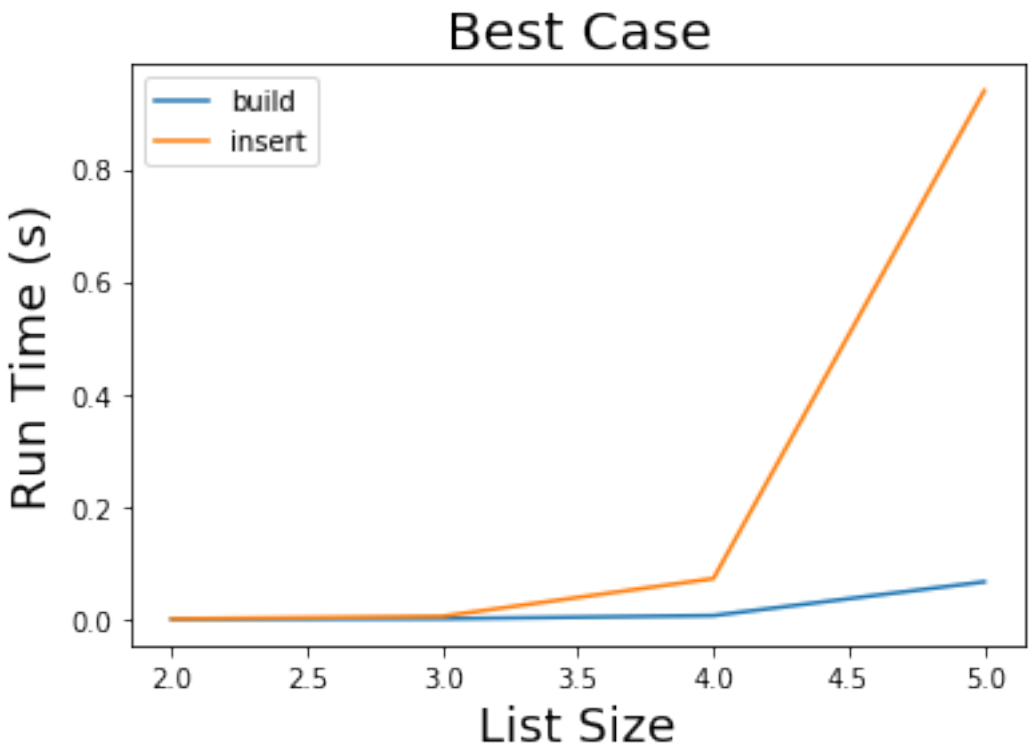
[7]:
```
plt.plot(np.log10(list_sizes), build_shuffle, label="build")
plt.plot(np.log10(list_sizes), insert_shuffle, label="insert")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Randomly Permuted", fontsize=20)
plt.legend()
```

[7]:
```
<matplotlib.legend.Legend at 0x7f2003443c50>
```



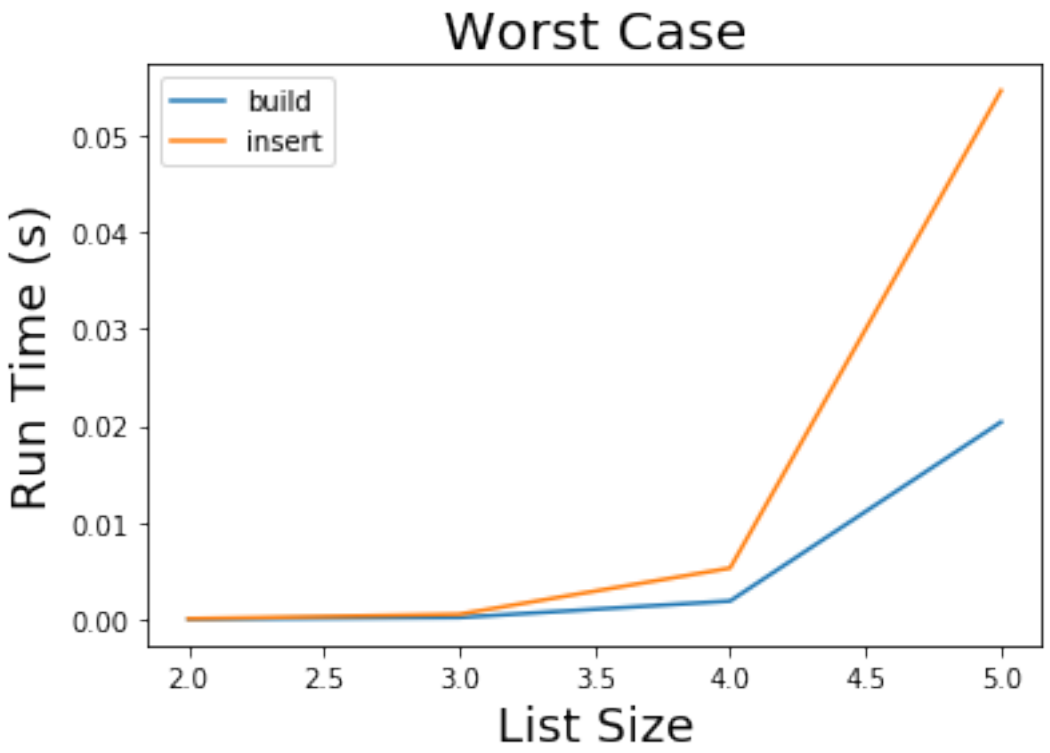
[8]:
```
plt.plot(np.log10(list_sizes), build_sorted, label="build")
plt.plot(np.log10(list_sizes), insert_sorted, label="insert")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Already Sorted", fontsize=20)
plt.legend()
```

[8]:
```
<matplotlib.legend.Legend at 0x7f20033d1fd0>
```



[9]:
```
plt.plot(np.log10(list_sizes), build_reverse, label="build")
plt.plot(np.log10(list_sizes), insert_reverse, label="insert")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Already Reverse Sorted", fontsize=20)
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f20033a7390>
```



1.4 Evaluating Run-Time of Extraction vs Insert

[10]:
```
def bench_extract(lst, n_trials, n_nums):
    times = []
    for i in range(n_trials):
        heap = lst[:n_nums] # copy the input list so that we don't modify it
        heap_size = len(heap)
        hp.build_max_heap(heap, heap_size)
        start = time.perf_counter()
        while heap_size > 0:
            hp.heap_extract_max(heap, heap_size)
            heap_size -= 1 # heap_extract_max() is not decreasing the value␣
↪here for some reason
        end = time.perf_counter()
        elapsed = end - start
        times.append(elapsed)
    return np.mean(times)
```

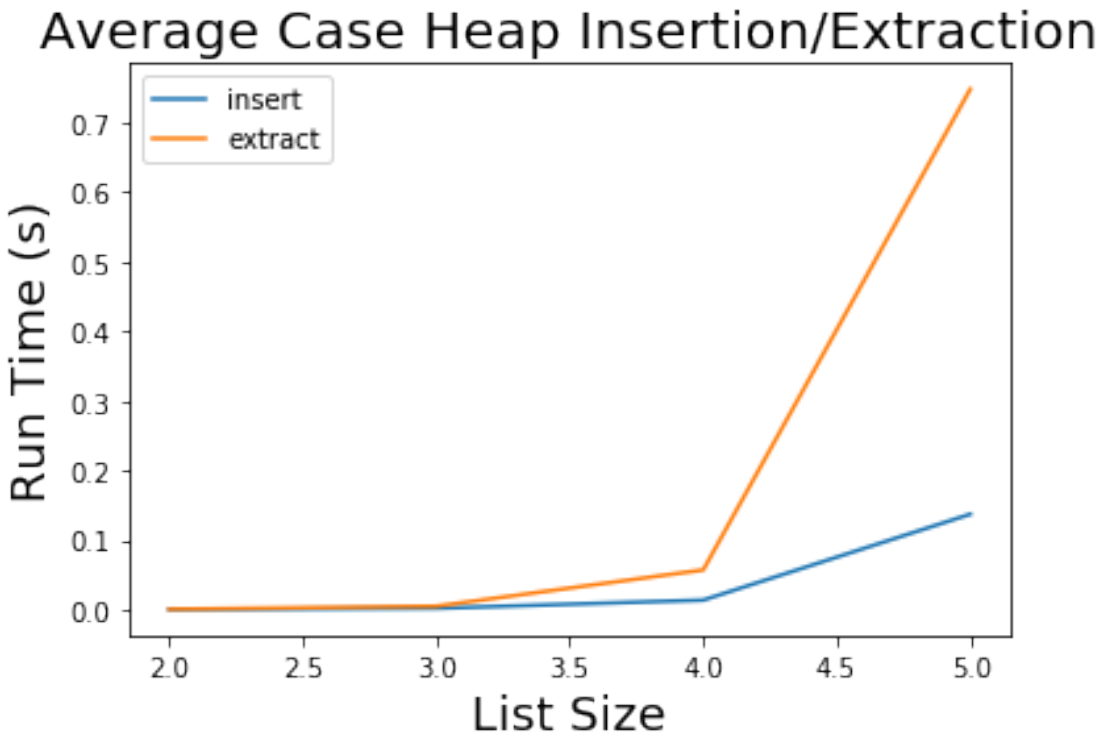
[11]:
```
extract_shuffle = []
for size in list_sizes:
    extract_shuffle.append(bench_extract(numbers, 10, size))
```

```
extract_sorted = []
for size in list_sizes:
    extract_sorted.append(bench_extract(sort, 10, size))

extract_reverse = []
for size in list_sizes:
    extract_reverse.append(bench_extract(reverse, 10, size))
```

[12]:
```
plt.plot(np.log10(list_sizes), insert_shuffle, label="insert")
plt.plot(np.log10(list_sizes), extract_shuffle, label="extract")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Average Case Heap Insertion/Extraction", fontsize=20)
plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x7f20033247b8>
```



1.5 Reflection Questions

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

    - Yes, the time varies. It turns out that what you would expect to be the "best-case" input is actually the worst-case and vice versa.

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.)

    - The structure of a heap as a list somewhat resembles a max-to-min sort, with the absolute max value always being first. When a sorted list (min-to-max) is used as input for creating a heap, it must do "bubbling" the most, since the larger values must be carried from the end to the beginning of the list.

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

    - Building the entire heap at once is faster because the time complexity of building a heap is only O(n), while the time complexity of inserting a single element is O(logn), making the time complexity of inserting one at a time for the whole array O(nlogn).

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

    - Inserting an element is faster because although the worst-case time complexities for both is O(logn), insertion's average-case time complexity is actually only O(1).