CS 3851 Algorithms - Lab 4: Heaps

Stuart Harley

Introduction

In this lab we are implementing max-heaps and the associated functions max_heapify, build_max_heap, heapsort, insert, extract_max, and increase_key. We then benchmark the time it takes to build a heap using build_max_heap. We then benchmark the time it takes to build a heap by individually inserting each element to compare that to the time it took using build_max_heap. Finally we benchmark the time it takes to individually extract each element from a heap to compare that to the time it takes to individually insert each element.

We show that building a heap all at once is faster than building a heap by individually inserting each element. We also show that inserting an element is on average faster than extracting the max element.

Importing Libraries

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

from heap import *
from test_heap import *
```

Testing Implementation of Heaps

In [2]:
```
!python test_heap.py
```

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Creating Lists to use for Benchmarking

For benchmarking, we will evaluate 3 cases: list is already sorted, list is randomly permuted, and list is already sorted in reverse order.

Creating lists of 100, 1000, 10000, and 100000 numbers. Creating the lists from random numbers. Then shuffling them to generate a random permutation of the list or sorting them in regular/reverse order to create the desired cases.

_p, _o, _r refers to the list being randomly permutated, sorted in order, and sorted in reverse order respectively.

In [3]:
```
l100 = [random.random() for i in range(100)]
random.shuffle(l100) # shuffling the list to generate a random permutation
l100_p = l100[:]
l100.sort() # sorting the list in order
l100_o = l100[:]
l100.sort(reverse=True) # sorting the list in reverse order
l100_r = l100[:]
```

In [4]:
```
l1000 = [random.random() for i in range(1000)]
random.shuffle(l1000) # shuffling the list to generate a random permutation
l1000_p = l1000[:]
l1000.sort() # sorting the list in order
l1000_o = l1000[:]
l1000.sort(reverse=True) # sorting the list in reverse order
l1000_r = l1000[:]
```

In [5]:
```
l10000 = [random.random() for i in range(10000)]
random.shuffle(l10000) # shuffling the list to generate a random permutation
l10000_p = l10000[:]
l10000.sort() # sorting the list in order
l10000_o = l10000[:]
l10000.sort(reverse=True) # sorting the list in reverse order
l10000_r = l10000[:]
```

In [6]:
```
l100000 = [random.random() for i in range(100000)]
random.shuffle(l100000) # shuffling the list to generate a random permutation
l100000_p = l100000[:]
l100000.sort() # sorting the list in order
l100000_o = l100000[:]
l100000.sort(reverse=True) # sorting the list in reverse order
l100000_r = l100000[:]
```

Benchmarking the Build_Max_Heap Function

Within the heapsort algorithm, the first step is to call build_max_heap which internally using the max_heapify function to build the heap from a list. We are benchmarking the build_max_heap function in this section.

We are running each benchmark 10 times (trials).

In [7]:
```
n_trials = 10
run_times_best_b = []
run_times_average_b = []
run_times_worst_b = []
```

In [8]:
```
# 100 elements
times = []
for i in range(n_trials):
    lst2 = l100_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_b.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 7.566490385215729e-05 sec
100 elements: Sorted in Order
Average Time: 5.9246298042126e-05 sec
100 elements: Sorted in Reverse
Average Time: 4.301409935578704e-05 sec
```

In [9]:
```
# 1000 elements
times = []
for i in range(n_trials):
    lst2 = l1000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l1000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l1000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_b.append(np.mean(times))
```

```
1000 elements: randomly permuted
Average Time: 0.0009652703010942787 sec
1000 elements: Sorted in Order
Average Time: 0.0007216750032966957 sec
1000 elements: Sorted in Reverse
Average Time: 0.0002224027004558593 sec
```

In [10]:
```
# 10000 elements
times = []
for i in range(n_trials):
    lst2 = l10000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l10000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l10000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_b.append(np.mean(times))
```

```
10000 elements: randomly permuted
Average Time: 0.006587520902394317 sec
10000 elements: Sorted in Order
Average Time: 0.007007107703248039 sec
10000 elements: Sorted in Reverse
Average Time: 0.002382382398354821 sec
```

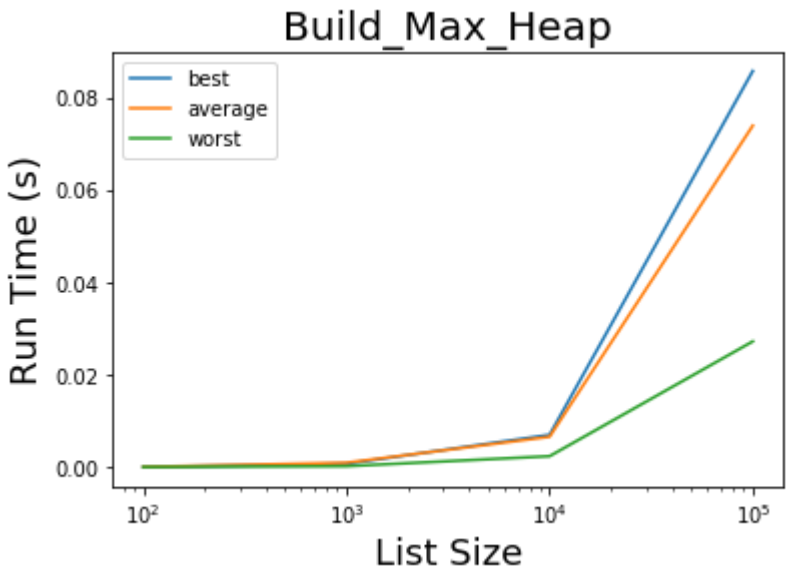
In [11]:
```
# 100000 elements
times = []
for i in range(n_trials):
    lst2 = l100000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_b.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    build_max_heap(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_b.append(np.mean(times))
```

```
100000 elements: randomly permuted
Average Time: 0.07390777799882926 sec
100000 elements: Sorted in Order
Average Time: 0.08574449819861911 sec
100000 elements: Sorted in Reverse
Average Time: 0.027225346196792087 sec
```

Plotting the Benchmarked Run Times of Build_Max_Heap

In [12]:
```
list_sizes = [100, 1000, 10000, 100000]
```

In [13]:
```
plt.plot(list_sizes, run_times_best_b, label="best")
plt.plot(list_sizes, run_times_average_b, label="average")
plt.plot(list_sizes, run_times_worst_b, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Build_Max_Heap", fontsize=20)
plt.legend()
plt.xscale('log')
```



Benchmarking Building the Heap by Inserting Each Element Individually

Instead of calling heapsort and therefore using the build_max_heap and max_heapify functions to build the heap, we are inserting each element into the heap individually.

In [14]:
```
run_times_best_i = []
run_times_average_i = []
run_times_worst_i = []
```

In [15]:
```
# 100 elements
times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l100_p)):
        max_heap_insert(lst2, l100_p[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l100_o)):
        max_heap_insert(lst2, l100_o[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l100_r)):
        max_heap_insert(lst2, l100_r[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 0.0002097948017762974 sec
100 elements: Sorted in Order
Average Time: 0.0004102754988707602 sec
100 elements: Sorted in Reverse
Average Time: 5.7606399059295656e-05 sec
```

In [16]:
```
# 1000 elements
times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l1000_p)):
        max_heap_insert(lst2, l1000_p[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l1000_o)):
        max_heap_insert(lst2, l1000_o[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l1000_r)):
        max_heap_insert(lst2, l1000_r[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
1000 elements: randomly permuted
Average Time: 0.0028093932080082594 sec
1000 elements: Sorted in Order
Average Time: 0.006551173198386095 sec
1000 elements: Sorted in Reverse
Average Time: 0.0006425094994483516 sec
```

In [17]:
```
# 10000 elements
times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l10000_p)):
        max_heap_insert(lst2, l10000_p[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l10000_o)):
        max_heap_insert(lst2, l10000_o[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l10000_r)):
        max_heap_insert(lst2, l10000_r[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
10000 elements: randomly permuted
Average Time: 0.017722812999272718 sec
10000 elements: Sorted in Order
Average Time: 0.0944838326977333 sec
10000 elements: Sorted in Reverse
Average Time: 0.0065598061948549 sec
```

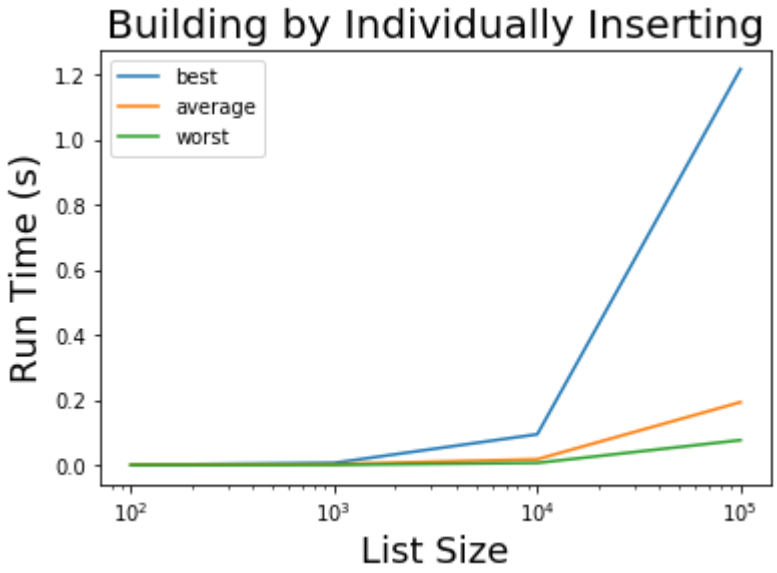
In [18]:
```
# 100000 elements
times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l100000_p)):
        max_heap_insert(lst2, l100000_p[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l100000_o)):
        max_heap_insert(lst2, l100000_o[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = []
    start = time.perf_counter()
    for i in range(len(l100000_r)):
        max_heap_insert(lst2, l100000_r[i])
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
100000 elements: randomly permuted
Average Time: 0.19317745179287157 sec
100000 elements: Sorted in Order
Average Time: 1.2169948480965105 sec
100000 elements: Sorted in Reverse
Average Time: 0.076445330894785 sec
```

Plotting the Benchmarked Run Times of Build_Max_Heap vs. Building the Heap by Inserting Each Element Individually

First plotting all the cases of individually adding the elements to the heap to visualize the difference between the 3 cases.

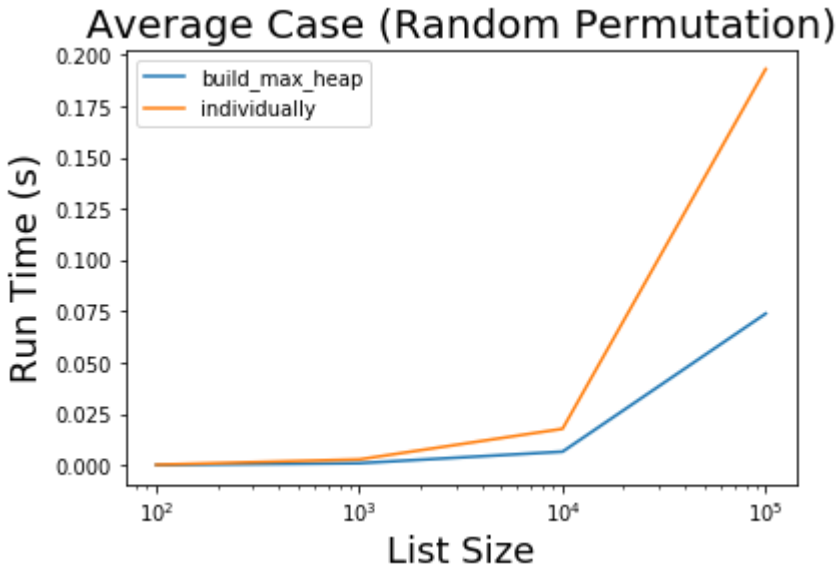
In [19]:
```
plt.plot(list_sizes, run_times_best_i, label="best")
plt.plot(list_sizes, run_times_average_i, label="average")
plt.plot(list_sizes, run_times_worst_i, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Building by Individually Inserting", fontsize=20)
plt.legend()
plt.xscale('log')
```



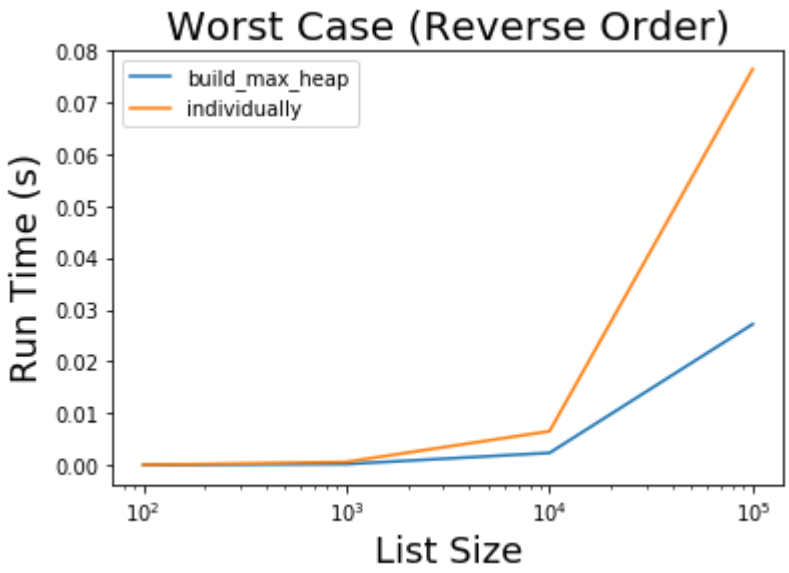
Creating one plot for each case comparing build_max_heap and building the heap by inserting each element individually.

In [20]:
```
plt.plot(list_sizes, run_times_best_b, label="build_max_heap")
plt.plot(list_sizes, run_times_best_i, label="individually")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Best Case (Sorted Order)", fontsize=20)
plt.legend()
plt.xscale('log')
```

In [21]:
```
plt.plot(list_sizes, run_times_average_b, label="build_max_heap")
plt.plot(list_sizes, run_times_average_i, label="individually")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Average Case (Random Permutation)", fontsize=20)
plt.legend()
plt.xscale('log')
```



In [22]:
```
plt.plot(list_sizes, run_times_worst_b, label="build_max_heap")
plt.plot(list_sizes, run_times_worst_i, label="individually")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Worst Case (Reverse Order)", fontsize=20)
plt.legend()
plt.xscale('log')
```



Benchmarking the Run Time of Extraction

Building heaps using the build_max_heap function, then benchmarking the time it takes to extract every element from the heap.

In [23]:
```
run_times_e = []
```

In [24]:
```
# 100 elements
times = []
for i in range(n_trials):
    lst2 = l100_p[:] # copying the input list so we don't modify it
    build_max_heap(lst2)
    start = time.perf_counter()
    for i in range(len(l100_p)):
        heap_extract_max(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_e.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 0.0003647240024292842 sec
```

In [25]:
```
# 1000 elements
times = []
for i in range(n_trials):
    lst2 = l1000_p[:] # copying the input list so we don't modify it
    build_max_heap(lst2)
    start = time.perf_counter()
    for i in range(len(l1000_p)):
        heap_extract_max(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_e.append(np.mean(times))
```

```
1000 elements: randomly permuted
Average Time: 0.004385412501869723 sec
```

In [26]:
```
# 10000 elements
times = []
for i in range(n_trials):
    lst2 = l10000_p[:] # copying the input list so we don't modify it
    build_max_heap(lst2)
    start = time.perf_counter()
    for i in range(len(l10000_p)):
        heap_extract_max(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_e.append(np.mean(times))
```

```
10000 elements: randomly permuted
Average Time: 0.06174333980306983 sec
```

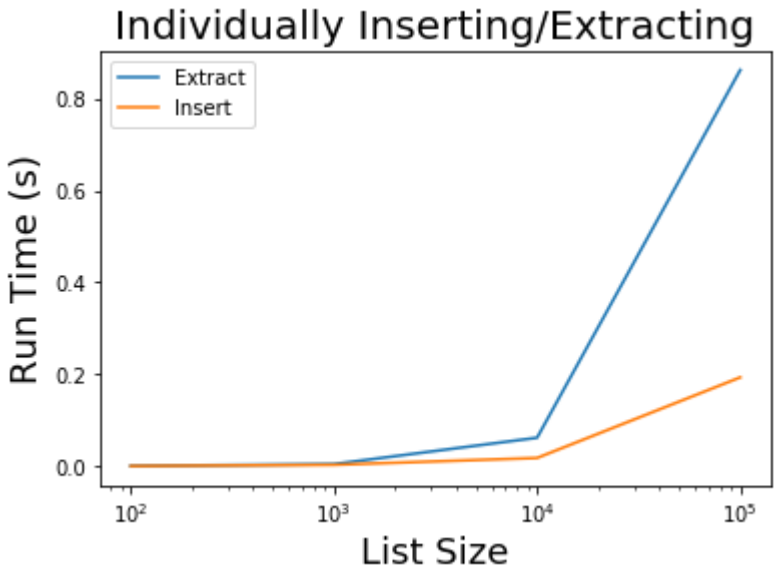
In [27]:
```
# 100000 elements
times = []
for i in range(n_trials):
    lst2 = l100000_p[:] # copying the input list so we don't modify it
    build_max_heap(lst2)
    start = time.perf_counter()
    for i in range(len(l100000_p)):
        heap_extract_max(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_e.append(np.mean(times))
```

```
100000 elements: randomly permuted
Average Time: 0.8621004673012067 sec
```

Plotting the Run Times of Inserting Each Element Individually vs Extracting Each Element Individually

In this comparison we use the average case of inserting each element individually since that is the average run time that would occur.

In [28]:
```
plt.plot(list_sizes, run_times_e, label="Extract")
plt.plot(list_sizes, run_times_average_i, label="Insert")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Individually Inserting/Extracting", fontsize=20)
plt.legend()
plt.xscale('log')
```



Reflection Questions

1) Heap creation varied between the 3 cases in both build_max_heap and when inserting individually; best (sorted in order), average (random permutation), and worst (reverse order). As expected, the average case was in the middle but the "best" case was actually the worst and the "worst" case was actually the best.

2) The best case for creating the heap ("worst" case: reverse sorted order) performed the best because of the way heaps are created. Heaps are formed with the smallest values at the bottom and the largest at the top. Therefore, when the list is in reverse sorted order, all of the numbers are already in a valid location for a max-heap.

3) Building a heap all at once is more efficient that inserting one at a time. When building a heap all at one time, when only have to perform a max_heapify call on half (n/2) of the elements; all those that are not leaf elements. Once you are done, the entire heap will be in a correct order. When you insert each element individually, you have to balance the heap every (n) time(s).

4) Inserting an element is faster than extracting an element in a heap. When inserting an element, there is a chance the element you are inserting will already be in a valid location. Or maybe you only need to shift it by a couple of rows rather than send it all the way to the top of the heap. However, when you extract an element from the heap, because you are switching it with the last element in the heap, that element is most likely going to have to be sent all the way back down to the bottom row of the heap. So you end up having to perform more switches on average than when inserting.

Conclusion

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. This is all done in an efficient way such that heapsort ends up being $\Theta(n\log n)$.