Hash Tables vs Binary Search Trees

Hash Tables and balanced Binary Search Trees are very commonly used data structures. Hash tables are known for their amortized constant time insertions, deletions, and searches. Balanced binary search trees are known for efficient $O(\log_2 n)$ time complexity for searches and traversals.

Hypothesis

In theory, a balanced binary search tree should be slower at insertions than a hash table due to its time complexity, however, hash tables can degrade with poorly written hash functions that result in collisions. We decided to investigate how much faster a hash table really is.

We hypothesize that a hash table will perform faster insertions compared to a balanced binary search tree. As input size grows large, the hash table's constant time complexity for insertions will significantly outperform a balanced binary search tree's $O(\log_2 n)$ time complexity.

Therefore, it is also hypothesized that hash tables will be able to reach a larger input size before reaching the 3 second time limit. The hash table will likely be able to reach an input size at least an order of magnitude greater than a balanced binary search tree due to the large difference in time complexity between constant time and logarithmic time at large values of n.

Methods

Full code can be found at this GitHub repo

Design:

For this study, we compared the run times of hash tables versus balanced binary search trees for insertion operations. We chose Python's dictionary class to represent hash tables, as the standard Python dictionary is implemented using hash tables. We chose Sorted Container's SortedList to represent a balanced binary search tree because it is implemented using a balanced binary search tree. Both options were concluded to be reasonable representations of the data structures we aimed to compare.

The data structures were initialized as follows:

```
# Data structures
hashtable = {}
multiset = sortedlist.SortedList()
```

These data structures will be used when executing run time tests. An array of size n will be constructed from random integers from 0 to n. The insert operation will be called on every element in the array and time to insert will be recorded. For the hash table, the insert operation is as follows:

```
    for i in arr:
        datastructure[i] = i
```

For the multiset/balanced binary search tree, the insert operation is as follows:

```
        for i in arr:
            datastructure.add(i)
```

Data collection:

To compare the insertion times of hash tables and binary search trees, we used Python's dictionary and the sortedcontainers.sortedlist library. They were initialized to empty containers. We then measured the insertion times of both data structures for a large range of input sizes, until the data structure took >= 3 seconds to insert *n* elements.

Prior to measuring insertion times for *n* elements, we created random arrays of size *n*, containing random integers between 0 and *n*. Python's random module was used to generate these numbers, along with a constant seed to ensure consistent results. The size of this array started at 10 and was increased by a factor of 10, until the time to insert *n* elements was greater than or equal to 3 seconds. For each value of *n* tested, we recorded the average time to insert per element into a dataset for the respective data structure. The measure_times function was used to implement this experiment, and is given as follows:

```
def measure_times(datastructure, dataset):
    """
    Measures the run time of a function for different sizes of input and stores the results in a dictionary.
    :param datastructure: Data structure to measure
    :param dataset: Dictionary mapping size of input to list of run times
    """

    start = 0
    end = 0
    random.seed(0)
    n = 10
    while end - start < 3:
        # Generate random array of size n
        arr = [random.randint(0, n) for _ in range(n)]

        if dataset.get(n) == None:
            dataset[n] = []

        # Measure run time if data structure is a dictionary
        if type(datastructure) == dict:
            start = time.time()
            for i in arr:
                datastructure[i] = i
            end = time.time()
            dataset[n].append((end - start) / n) # Average time per element
        # Measure run time if data structure is a sorted list
        else:
            start = time.time()
            for i in arr:
                datastructure.add(i)
            end = time.time()
            dataset[n].append((end - start) / n)
        # Increase size of input
        n = n*10
    print(type(datastructure),"  n: ",n)
```

The data sets used to keep track of the run times for each data structure are initialized as empty dictionaries, where the keys are the input sizes and the values of run times.

```
# Dictionary mapping size of input to list of run times
hashtable_times = {}
multiset_times = {}
```

We repeated the process 10 times to ensure validity and avoid outliers skewing the data.

```
for i in range(10):
    # Measure run times
    measure_times(hashtable, hashtable_times)
    measure_times(multiset, multiset_times)

    # Reset data structures
    hashtable = {}
    multiset = sortedlist.SortedList()
```

Finally, we constructed pandas DataFrames for both runtime datasets and created lineplots for each data structure and saved them as images.

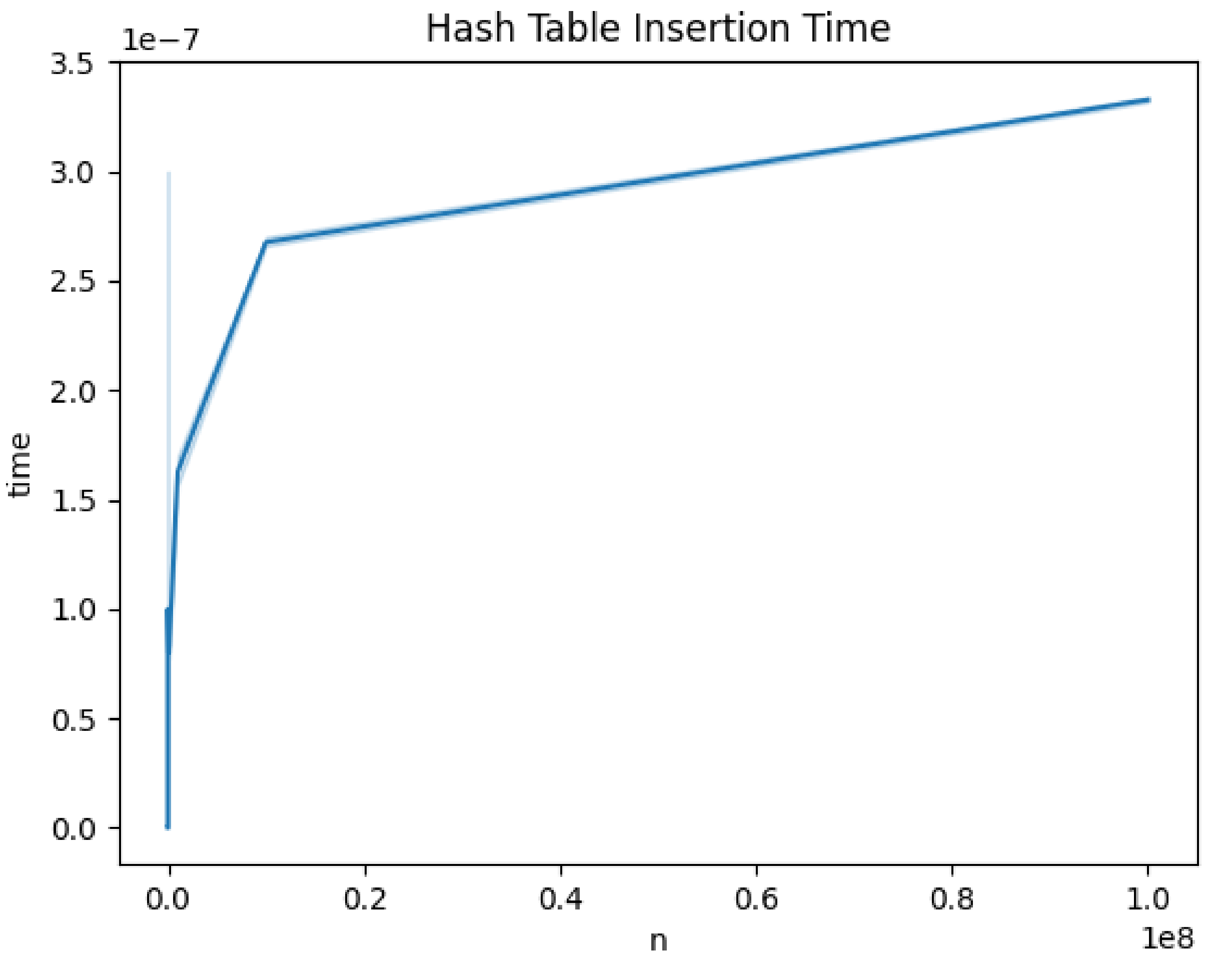
```
# Plot results
df = pd.DataFrame.from_dict(hashtable_times, orient='index')
df = df.transpose()
df = df.melt(var_name='n', value_name='time')
df['n'] = df['n'].astype(int)
df['time'] = df['time'].astype(float)
sns.lineplot(x='n', y='time', data=df)
```

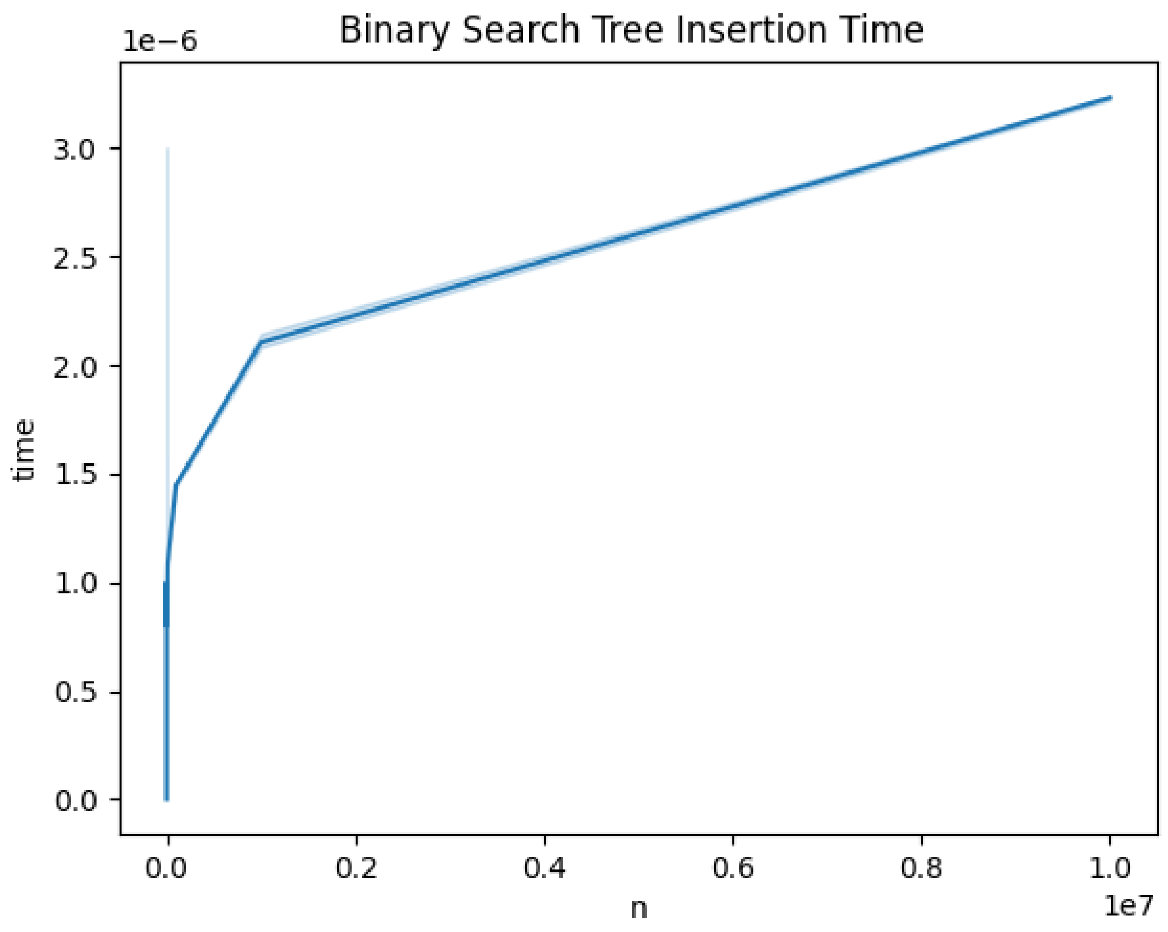
```
plt.title('Hash Table Insertion Time')
plt.savefig('ht.png')
plt.clf()

df = pd.DataFrame.from_dict(multiset_times, orient='index')
df = df.transpose()
df = df.melt(var_name='n', value_name='time')
df['n'] = df['n'].astype(int)
df['time'] = df['time'].astype(float)
sns.lineplot(x='n', y='time', data=df)
plt.title('Binary Search Tree Insertion Time ')
plt.savefig('bt.png')
plt.clf()
```

The experiment was executed on a computer with a Ryzen 5 2600 and 16GB of main memory.

Results

For each data structure, the insertion run times for a large range of *n* were recorded in the respective data sets. *n* was increased until the time to insert all elements took at least 3 seconds. The results are shows in the follows graphs, where the *x*-axis represents input size, *n*, and the y axis represents the average time to insert one element (units shown on graphs).


Hash Table Insertion Time
1e−7
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
time
0.0
0.2
0.4
0.6
0.8
1.0
1e8
n



Discussion

Our results show that the hash table outperformed the balanced binary search tree at insertion by approximately one order of magnitude. Not only was the hash table approximately 10 times faster at insertion at these large values of n, but it was able to handle an input size an order of magnitude larger than the balanced binary search tree before crossing our 3 second time limit. These results validate both aspects of our initial hypotheses.

The hash table's run time complexity allowed it to significantly outperform the balanced binary search tree for insertions. The amortized constant time hashing function is much faster than the balanced binary search tree's logarithmic time insertion because it must traverse the tree every time an insert occurs. On the other hand, Python's dictionary will simply hash the element to a bucket where it can be placed in a linked list. Although this can result in a worst-case time complexity of $O(n)$, if all the elements are hashed to the same location, the average, or amortized, time complexity is approximately $O(1)$ because the hash function is well written and will hash different objects to different buckets. The balanced binary search cannot perform insertions at this speed because it must first traverse its tree to find the spot to insert an element. This can be up to logarithmic time. For a random array such as the one we used, the elements will often be inserted all over the tree, which requires many traversals.

As a result, the hash table was able to insert up to 10^8elements before exceeding our 3 second time limit, while staying in the order of magnitude of 10^{-7}seconds per element. The balanced binary search tree inserted

10^7elements before exceeding the 3 second limit and remained in the order of magnitude of 10^{-6}seconds per element.

Conclusion

Under the conditions tested, hash tables greatly outperform balanced binary search trees at the insertion operation. It performed approximately 10 times faster at the values of *n* we tested and was able to insertion approximately 10 times more elements before timing out.

Sources and Tools:

Python Sorted Containers

https://seaborn.pydata.org/generated/seaborn.lineplot.html

Full source code - GitHub link