Vector vs Balanced BST

Vectors are a common data structure to store data. However, they often perform poorly when we want our data to be sorted, because data must be shifted on every insert, resulting on linear time complexity. On the other hand, a balanced binary search tree, or multiset, can be a better option due to its logarithmic insertions and deletions. We will implement a sorted vector with a binary search to find the insertion index to see if it will compare to the logarithmic time of balanced binary search trees.

Hypothesis

We hypothesize that implementing a sorted vector with a binary search will be faster than the linear time complexity of a standard vector but will still perform poorly compared to a balanced binary search tree. Although using a binary search can quickly find the insertion index, all the elements proceeding the index must still be shifted on each insertion. So, although the search will likely be logarithmic time, the shifting will still be linear time in the worst case. A balanced binary search tree does not have to shift elements on every insertion and will maintain its expected logarithmic time complexity.

Methods

Full code can be found at this GitHub repo

Design: For this study, we compare the run time performance of a vector with binary search and a balanced binary search tree. We chose to implement the vector with a standard Python list and wrote a binary search algorithm to find the insertion index of a given element. We chose to represent the balanced binary search tree with a sortedcontainer.sortedlist because it is implemented with 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
multiset = sortedlist.SortedList()
vector = []
```

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*. For the list, we will use binary search on every element of the array to find the insertion index then call the insert operation on the list. This procedure is given as follows:

```
    for i in arr:
        # Run binary search to find index to insert element
        left = 0
        right = len(datastructure) - 1
        mid = 0
        while left <= right:
            mid = (left + right) // 2
            if datastructure[mid] < i:
```

```
                left = mid + 1
            elif datastructure[mid] > i:
                right = mid - 1
            else:
                break
        # Insert element
        datastructure.insert(mid, i)
```

For the balanced binary search tree, the insert operation will be called on every element of the array. The insert operation for the balanced binary search tree is as follows:

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

Data collection:

A data set was constructed for each container to store the run times for the insert operations. They were initialized as empty dictionaries to be populated with keys of *n* values and keys being the associated run times. They were initialized as follows:

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

We then measured the insertion times for a large range of input sizes. We increased input size by a factor of 10 for each test until the time to insert *n* elements exceeded 3 seconds. Prior to measuring the insertion times, random data was needed. We constructed an array of size *n*, filled with random integers between 0 and *n* for each iteration of the test. The seeding value was set to a constant to ensure consistent results between tests.

We used the measure_times function to implement the tests, which 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:
    # Increase size of input
    n = n*10
    # 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 multiset
    if type(datastructure) == sortedlist.SortedList:
        start = time.time()
        for i in arr:
            datastructure.add(i)
        end = time.time()
        # Store run time
        dataset[n].append(end - start)

    # Measure run time if data structure is a vector
    else:
        start = time.time()
        for i in arr:
            # Run binary search to find index to insert element
            left = 0
            right = len(datastructure) - 1
            mid = 0
            while left <= right:
                mid = (left + right) // 2
                if datastructure[mid] < i:
                    left = mid + 1
                elif datastructure[mid] > i:
                    right = mid - 1
                else:
                    break
            # Insert element
            datastructure.insert(mid, i)
        end = time.time()
        # Store run time
        dataset[n].append(end - start)
```

The function will choose the appropriate insertion procedure based on the type of the data structure that is passed to it. For the balanced binary search tree (sortedlist), it will call the add() method for every element in our array. Otherwise, it will perform a binary search for the insertion index of each element of the random array into our list, then use the insert() method to insert at the index we found.

We repeated this process 10 times to ensure validity and avoid data skews.

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

    # Reset data structures
    vector = []
    multiset = sortedlist.SortedList()
```

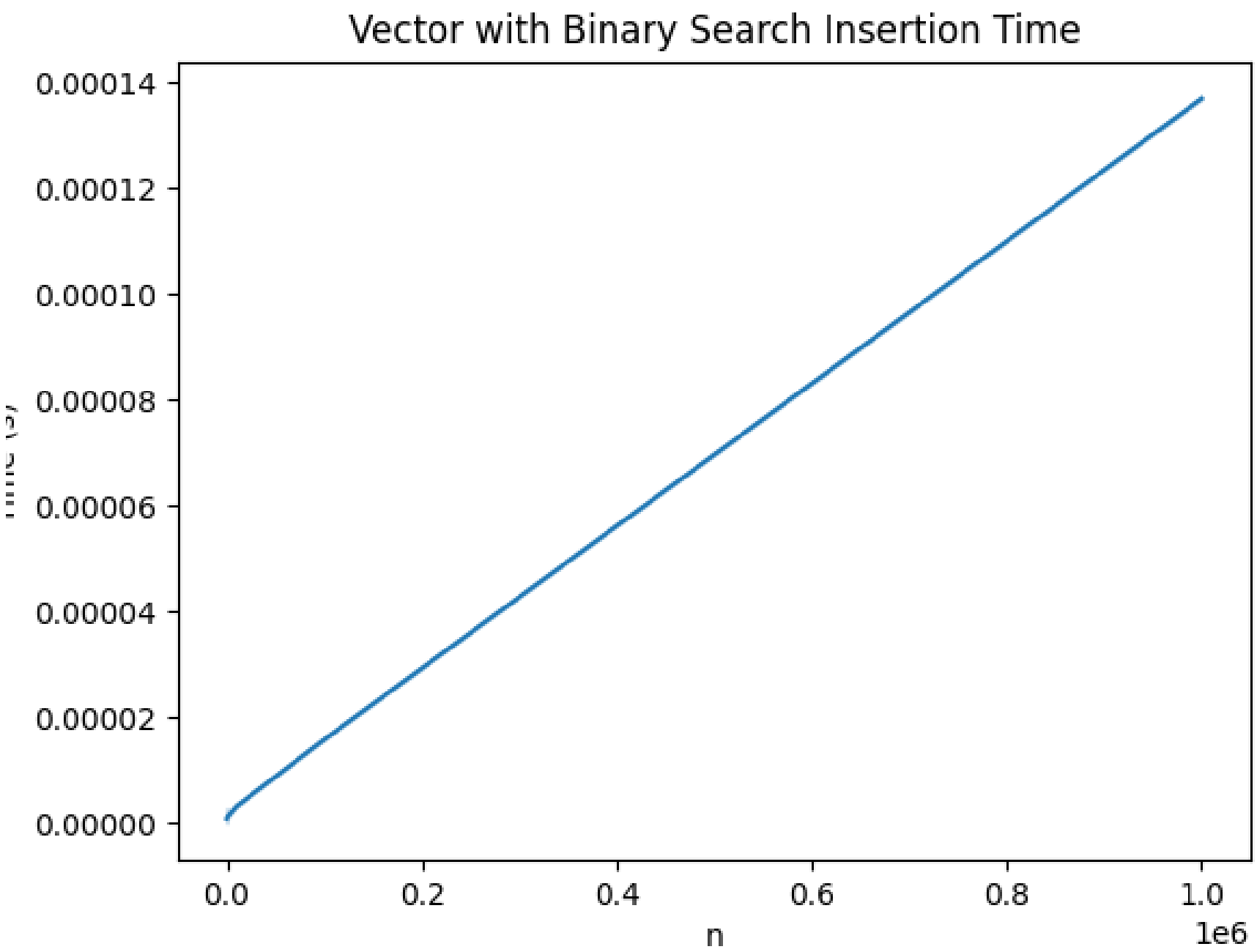
Finally, we construced pandas DataFrames for each dataset to plot the results with matplotlib's lineplot.

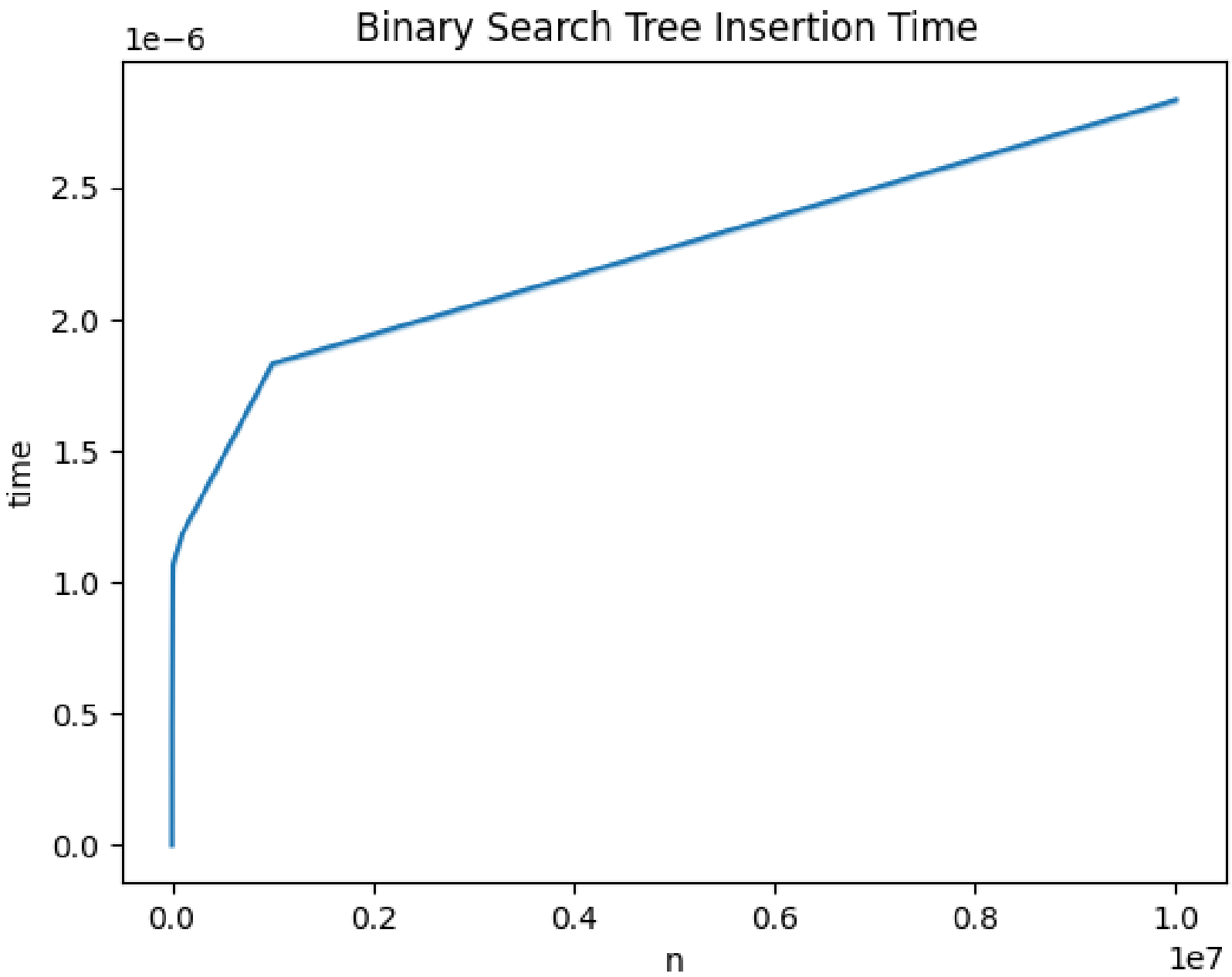
```
# Plot results
df = pd.DataFrame.from_dict(vector_times, orient='index')
df = df.transpose()
df = df.melt(var_name='n', value_name='time')
df['n'] = df['n'].astype(int)
df['time'] = df['time'].astype(float)
sns.lineplot(x='n', y='time', data=df)
plt.title('Vector with Binary Search Insertion Time')
plt.savefig('vector_insertion.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('bst_insertion.png')
plt.clf()
```

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

Results

The insertion run times for each data structure were stored in separate data sets. Each data set contained run times for a wide range of values of n. The datasets were formatted for matplotlib's lineplot. The results are given in the following graphs, where the x-axis is the input size, n, and the y-axis is the average run time per element for the insertion operation.


Vector with Binary Search Insertion Time
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
0.0
0.2
0.4
0.6
0.8
1.0
n
1e6


Binary Search Tree Insertion Time
1e−6
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
1e7
n

Discussion

The results of our experiment show that the vector with binary search performed significantly slower than the balanced binary search tree for all values of *n*. Although the vector performed poorly compared to the balanced binary search tree, the performance was sub-linear, a great improvement from the expected run time of a sorted list implemented without binary search.

Although the vector with binary search performed well, the shifting operation required for every insertion could not be avoided and significantly slowed down the operation. The binary search can efficiently find the insertion index within logarithmic time, but when we insert an element into a vector, all the proceeding elements must be shifted over by 1. This is a linear time operation and bottlenecks our binary search's faster time complexity. For very large data sets, such as the ones we tested, many elements may need to be shifted if the insertion index is somewhere in the middle of the list. This is often the case when we try to insert randomly generated data, as we did.

The balanced binary search tree, on the other hand, maintained logarithmic run time complexity for insertions due to its efficient binary search and not having to shift elements on each insert. The balanced binary search tree, therefore, was able to perform insertions at a rate of approximately 2 orders of magnitude faster than the vector with binary search (10^{-6}seconds per element and 10^{-4}seconds per element, respectively).

Conclusion

Under the conditions tested, a sorted vector implemented using binary search to locate insertion indices will perform at a sublinear time complexity, but cannot perform nearly as well as the logarithmic insertions of a balanced binary search tree due to the expensive shifting required for every insertion.

Sources and Tools:

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

Quick Sort Source Code - GitHub link

Full source code - GitHub link