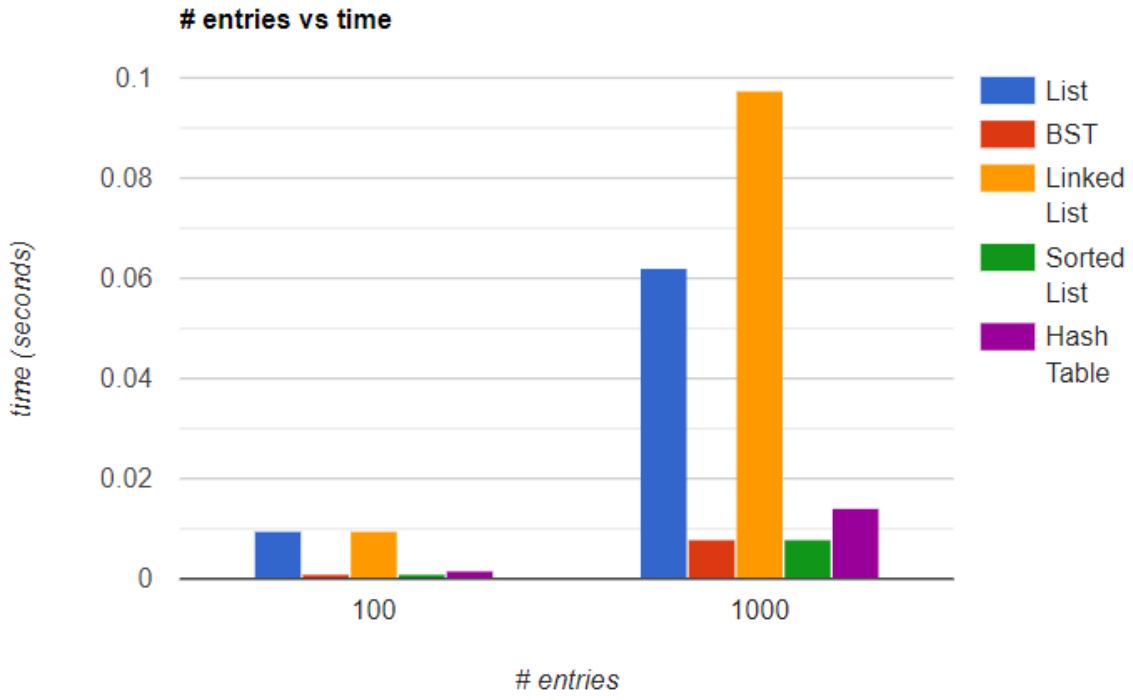
Data Table:

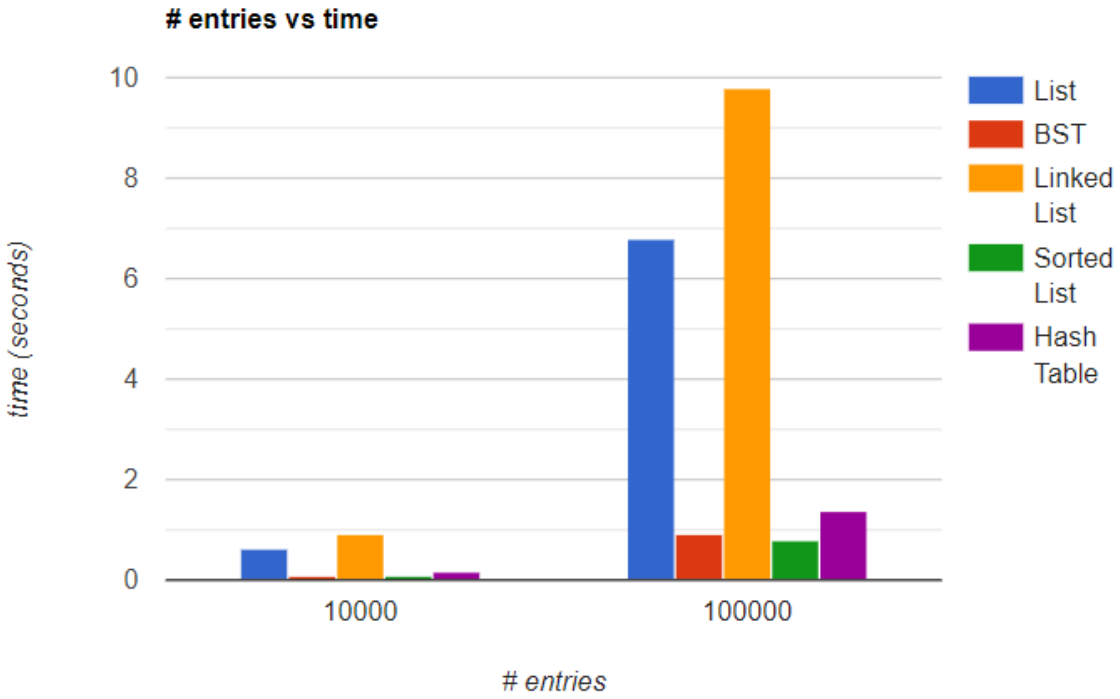
List		**BST**		**Linked List**		**Sorted List**		**Hash Table**	
# entries	time (seconds)	# entries	time (seconds)	# entries	time (seconds)	# entries	time (seconds)	# entries	time (seconds)
100	0.009502887726	100	0.001000642776	100	0.009503126144	100	0.000999212265	100	0.0015001297
1000	0.06201100349	1000	0.008001327515	1000	0.09751749039	1000	0.008001565933	1000	0.01400256157
10000	0.6341123581	10000	0.07951426506	10000	0.9346666336	10000	0.0820145607	10000	0.1475257874
100000	6.792708158	100000	0.9116621017	100000	9.80674386	100000	0.8086440563	100000	1.365242243

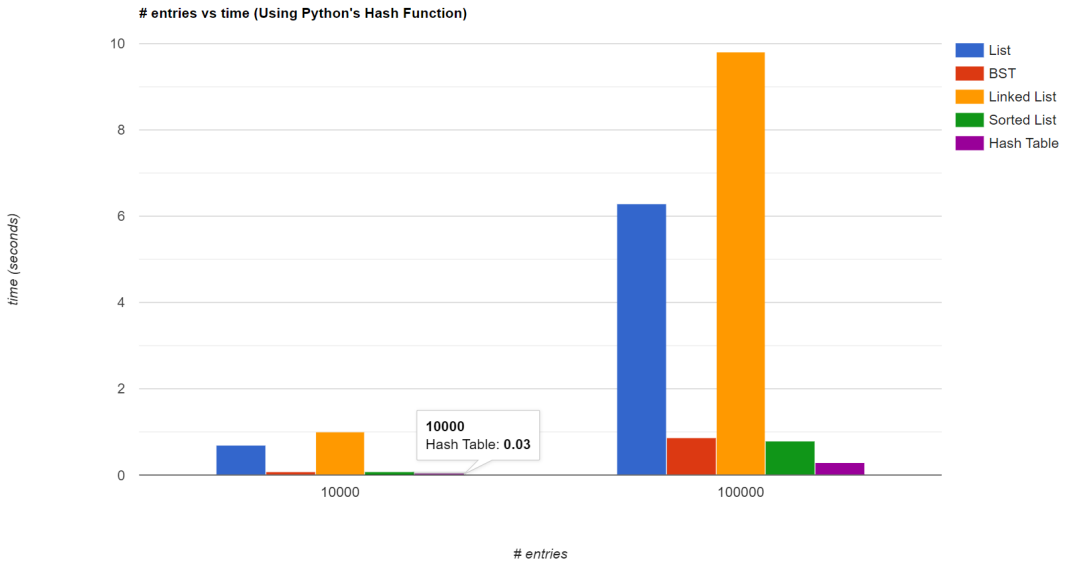
Using Python's built-in hash function (only 10k and 100k queries):

List		**BST**		**Linked List**		**Sorted List**		**Hash Table**	
# entries	time (seconds)	# entries	time (seconds)	# entries	time (seconds)	# entries	time (seconds)	# entries	time (seconds)
10000	0.6836197376	10000	0.08201527596	10000	1.00267601	10000	0.08551454544	10000	0.029504776
100000	6.285598278	100000	0.8676493168	100000	9.802712917	100000	0.7871365547	100000	0.2805495262

Graphs:


entries vs time
0.1
0.08
0.06
0.04
0.02
0
time (seconds)
100
1000
entries
List
BST
Linked List
Sorted List
Hash Table


entries vs time
10
8
6
4
2
0
time (seconds)
10000
100000
entries
List
BST
Linked List
Sorted List
Hash Table



Data Observations:

With 1000 queries, the general trend of the order of time taken to search for each data structure is as follows: linked list, list, hash table, binary search tree, and sorted list. The linked list and list implementations both take O(n) in the average and worst cases since you must iterate through both lists to find the given element. However, with more queries, the linked list takes a much longer time than the list implementation, at 100,000 entries, the list took ~6.8 seconds while the linked list took ~9.8 seconds, this could be attributed to the time it takes to jump around in memory due to the use of node pointers. The third slowest data structure was the hash table. I was surprised to see that the hash table would perform worse compared to the binary search tree and sorted list since in the best case, the hash table can search in O(1) time. The reason why the hash table was not performing as efficiently was due to the quality of the hash function. As a group, we chose to do a mod function with the sum of each key's ascii values, coupled with the open hashing implementation, this could lead to more collisions, resulting in a longer search time due to a sequential search of the linked lists. The binary search tree was marginally slower than the sorted list, with both producing similar results. This is because the BST and sorted list in the average case can search in O(log(n)). Since the BST property ensures that the left subtrees/children are less than or equal to the parent node, allowing the data to be searched to be

cut in half based on the value of the key. The sorted list is similar to a binary search and can be utilized to achieve O(log(n)).

Final conclusion on the best data structure to use:

I think that the best data structure to use would be the hash table, only if the hash function is efficient enough to limit collisions. In the last graph, I used Python's built-in hash function, and it was faster than the sorted list at both 10,000 and 100,000 search entries. At 100,000 entries, the hash table implementation searched at ~0.28 seconds while the sorted list searched at ~0.79 seconds. With the most efficient hash function, the hash table implementation can search in O(1) versus the sorted list searching in O(log(n)), with the exception of the best case of the sorted list being O(1) as well.