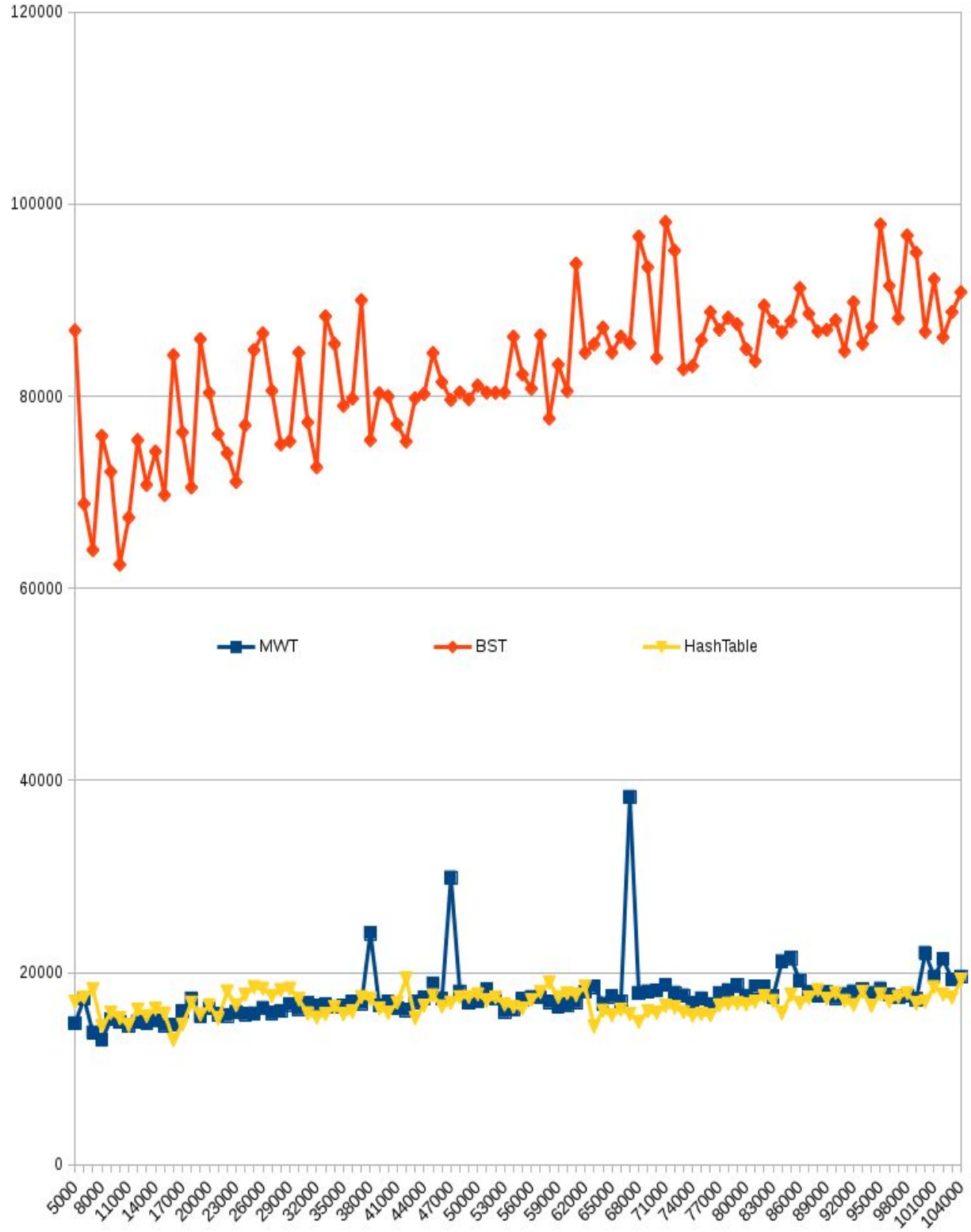
Final Report Kening Zhang Gang Yang
Part3



In class we learened that

	HashTable	MultiWay Tree	BST
Average Case Time	O(1)	O(k)	O(logN)

Which is consistent to my test result.
First, for HashTable, we don't see a clear trend of increasing in average case time with the increase in case number. The time is fluctuating within the range: [15000,19000] without specific outlier, which means the average case time does not depend on the number of words inserted.

Second is Multiway Tree. The average case time is increasing with a low slope as the number of words inserted increases. Three obvious outlier can be clearly found. We think that this is the outcome that the multiway tree has an average case time of O(k) , where k is the longest word inserted. With more words inserted, we are more likely to get a longer word, therefore increasing the average case time.

Finally is Binary Search Tree. the pattern of the increase in average case time can be observed easily. If we draw a least square curve within the data, the slope of the curve would increase slower as the words inserted increases. The reason to explain it can only be that the average case time of BST is O(logN) since the second derivative of log Function decreases as N increases.

Part4

A. Describe how each function works.

HashFunction1: Simply add up all the ASCII code of each characters in the string, and then mode the size of the hashTable, which is twice of number of words to get the position to insert in the hash table.

// Source:textbook 5.2.3 https://stepik.org/lesson/31712/step/3?unit=11896

HashFunction2: A little bit fancy but similar to HashFunction1. It also takes out the ASCII code of each characters. Then it multiples weight 31^(n-1), 31^(n-2)...31^(0) where n is the length of the string to each characters and adds up to mode the size of the hashTable.

// Source: Online https://www.cs.cmu.edu/~adamchik/15-121/lectures/Hashing/hashing.html

B. I verify if hash function works by writing test case of different string: the basic case with three letters, the case with empty string and the case with extreme long characters. I calculate the hash value by hand and then write test to see if the outcome of two hash functions is the same as I expected.

For example:

```
// Test Case2 to check if hash function works for empty space
std::string s2("I I");
if(hashValue1(s2,50)==28)
{
        cout<<"Success in hashValue1"<< endl;
}
else
{
  cout<< "Fail in hashValue1" << endl;
}

if(hashValue2(s2,50)==18)
{
        cout<<"Success in hashValue2"<< endl;
}
else
{
  cout<< "Fail in hashValue2" << endl;
}
cout<< hashValue1(s2,50)<<endl;
cout<< hashValue2(s2,50)<<endl;
```

C.

Hash1		1000	2000	3000	4000	5000
	#hits	#slots rec	#slots rec	#slots rec	#slots rec	#slots receiving the
	0	1252	2804	4514	6306	8152
	1	547	675	667	633	613
	2	158	328	417	445	403
	3	38	128	213	266	316
	4	3	46	122	192	217
	5	1	14	39	85	151
	6	1	4	20	38	78
	7		1	7	25	41
	8			1	7	14
	9				3	9
	10					5
	11					0
	12					1
Average Time		1.315	1.6045	1.88433	2.126	2.4402
Worst Case		6	7	8	9	12

Hash2		1000	2000	3000	4000	5000
	#hits	#slots rec	#slots rec	#slots rec	#slots rec	#slots receiving the #hits
	0	1221	2425	3684	4901	6138
	1	605	1225	1750	2360	2929
	2	133	286	465	602	760
	3	35	55	86	113	146
	4	6	7	14	23	23
	5		2	0	1	3
	6			1		1
Average Time		1.454	1.436	1.468	1.463	1.4684
Worst Case		4	5	6	5	6

D. HashFunc2 is better.

When the number of inserted words is below 1000, the hash function1 is more efficient that the average time is less than that of hash function2. However, as long as the words is increasing, the average time of hash function does not increase anymore and keeps constant while that hash function has a bad performance with large number of hits and longer average time.

It matches my expectation because hashFunc2 involves more complex algorithm including the power function to make the data spread out. Therefore it makes sense to have less collisions.