HW5
CSC403 Data Structure II
Harry Chen

Test the performance of Separate Chaining ST and Linear Probing ST on Flight Data

Part I: Data description and generation

I created a Flight Object to be the Key of the symbol table, The Flight object contains 4 parts: flight company name which are composed by 2 capital letters, 4 digits flight number range in 1000 to 9999, origin and destination airport which are represented by 3 capital letters. An example of my flight object is "UA-6427-OHR-DTW". For the value of the key, I used flight duration.

I randomly generate those four parts by creating several functions which are listed below:

1. generatename(int size): By using this function, I can generate either 2 or 3 letter to represent the flight company name or destination and origin name.
2. generatenumber(int lo, int hi): By using this function, I can generate a number in user defined range.
3. generatenamepool(int poolsize, int wordsize): This function will return a list contain non-duplicated name either for flight company name or airport.
4. Randomchoose(String[] pool): randomly choose a name from the pool.
5. generateflight (): This function will return a flight object with 3 name parts coming from the pool and number created by "generatenumber()" function.

The uniqueness of the key and the hashing index collision depend on how you specify the poolsize argument in generatenamepool.

Part II: Generate two test cases.

In order to test and compare the performance of put and get operations using Separate Chaining and linear probing container, I developed the following test strategy.

1. Populate empty container to size N: using auxiliary ST to avoid counts I/O time.
2. Insert N+1st key to size N container: In order to make the result more visible, I made the insert number in proportion to the table size, for example, inserting 1 flight when table size is 1000, inserting 10 flights when table size is 10000, inserting 100 flights when table size is 100000. Same strategy is applied when testing get 1 value from size N table.
3. Compare the performance of those two when get and put operations mix together. What I did is simply replace "test.put(key,0)" to "test.put(key, st.get(key))" so that every put operation need to do a get operation first to find a value in auxiliary array.

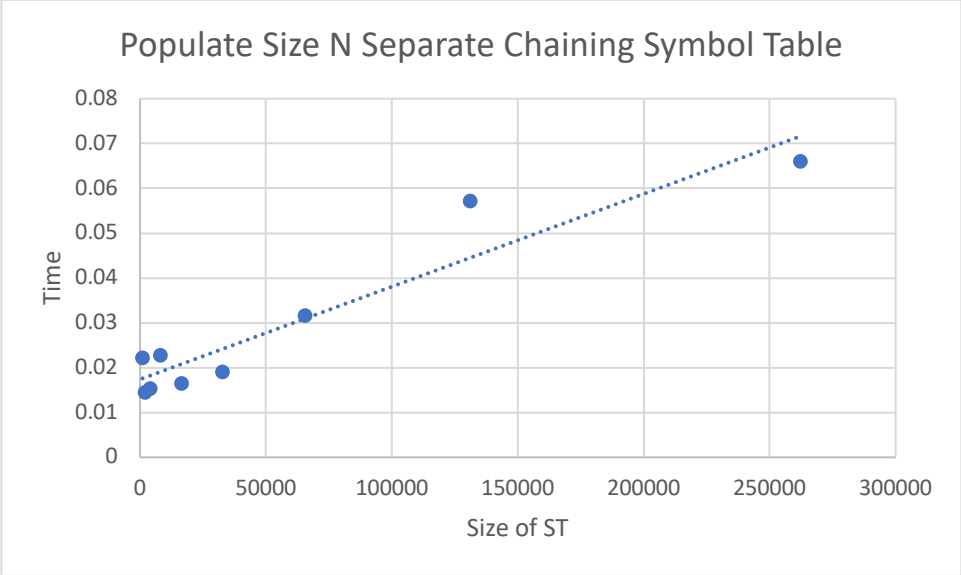
HW5
CSC403 Data Structure II
Harry Chen

Part III: Test result

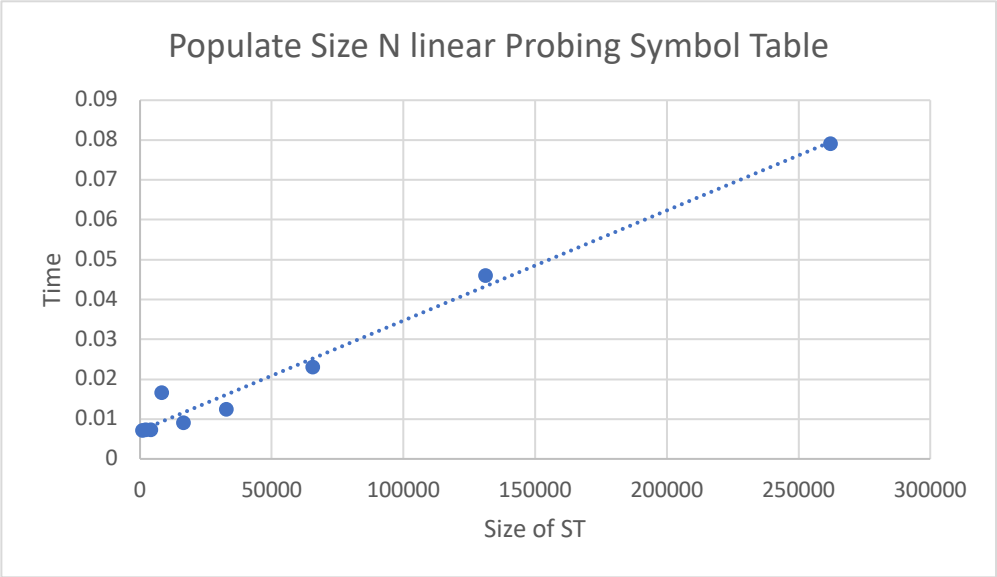
1. Time — Size table

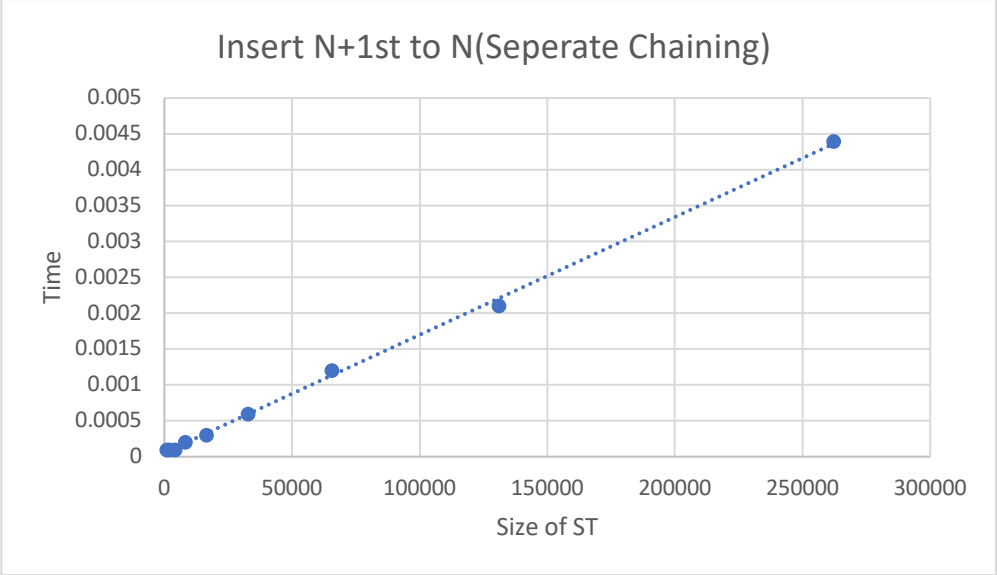
	SC(From empty)	LP(from empty)	Mixture SC	Mixture LP	putSC (N+1st)	PutLP (N+1st)	getSC (1)	GetLP (1)
1024	0.0222	0.0071	0.0189	0.0069	0.0001	0	0	0
2048	0.0144	0.0074	0.0155	0.0067	0.0001	0	0	0
4096	0.0153	0.0074	0.0179	0.0072	0.0001	0.0001	0.0001	0
8192	0.0227	0.0166	0.0165	0.0084	0.0002	0.0001	0	0
16384	0.0164	0.0091	0.0172	0.0108	0.0003	0.0002	0	0
32768	0.0191	0.0125	0.0338	0.0187	0.0006	0.0007	0	0
65536	0.0316	0.023	0.0413	0.0246	0.0012	0.0012	0	0
131072	0.0571	0.046	0.0449	0.0407	0.0021	0.0025	0.0001	0
262144	0.066	0.0791	0.0723	0.0922	0.0044	0.0045	0	0.0001

2. Time — Size dot plot and regression

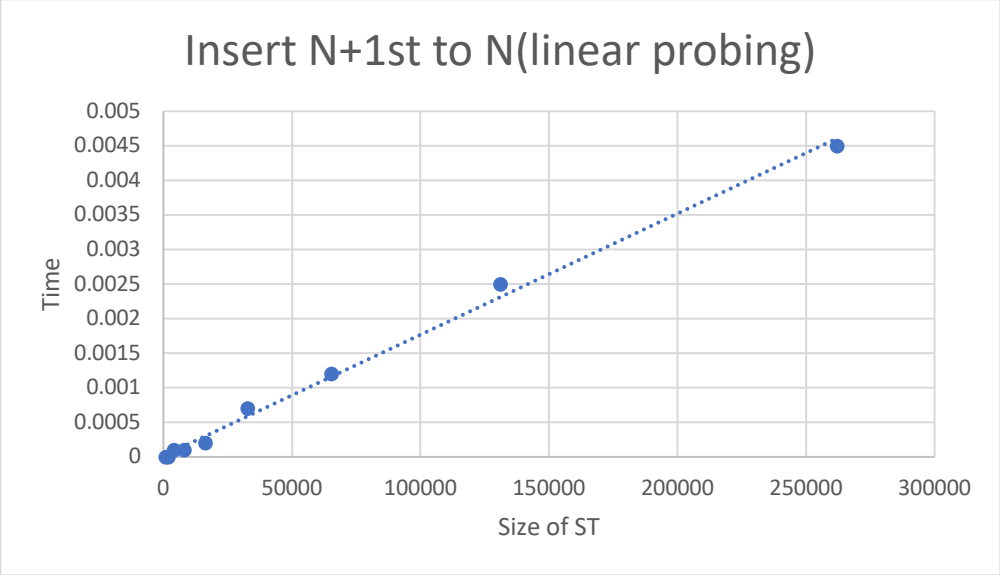


HW5
CSC403 Data Structure II
Harry Chen





HW5
CSC403 Data Structure II
Harry Chen



Part IV: Analysis

1. What is the algorithmic complexity of creating/populating a hash table of size N?
   According the graph, both SC and LP took linear time, O(N)
2. What is the algorithmic complexity of inserting (put) the N+1St key into a hash table of size N?

   Depend on the uniqueness of data and how hashing function hash the key, the performance varies. If there are too much collision happened, it will take more time on searching. In my case, to insert relatively small amount data only takes time in millisecond for both SC and LP, but time-cost do increase in proportion to the table size.
3. What is the algorithmic complexity of searching (get) for a key in a hash table of size N?
   In my case, they both take constant time, cause no matter how large the table size is, they take <.0001 time to find the correspond value of the target key.
4. Which Hash implementation performs better on the scenario of a mixture of a large number (but near equal) number of puts and gets?

SC	**0.0189**	**0.0155**	**0.0179**	**0.0165**	**0.0172**	**0.0338**	**0.0413**	**0.0449**	**0.0723**
LP	0.0069	0.0067	0.0072	0.0082	0.0108	0.0187	0.0246	0.0407	0.0922

See through the above data table, LP always takes less time than SC.

HW5
CSC403 Data Structure II
Harry Chen

Part V: Limitation & Improvement

I list the limitations and Improvement in the following list.

Limitation	Improvement

1. **Timer should not be embedded together!**
 	
`for (int N=1024; N < 300000; N*=2) {`
`Stopwatch sw = new Stopwatch()`
`….`
`for (int r = 1; r <= reps; r++) {`
 	
`……`
`for ( Flight key: lp.keys()) {`
`Stopwatch sw2 = new`
`……`
`for(int j =0; j<N/100;j++){`
`……`
`Stopwatch sw3 = new`
`for(int q =0; q<N/100;q++) {`
`}`
	Use System clock to count time elapsed by doing "Start – end" in each part of the experiment might be a good choice.

2. **Generate different size namepool to make the result more generic.**
 	
`String[] Airportpool = generatenamepool(50,3);`
`String[] Flightnamepool = generatenamepool(30,2);`
`int number = generatenumber(1000,9999);`
	What if we limited the flight company only in "UA(United Airline)" "AA(American Airline)" "SA(Sprint Airline)" "JA(Jetblue Airline)", shrinked the flight number range only between 1000 and 1500 or followed some pattern, limited the airport to top 20 main airports U.S.A domestic. In this way, Hash function may hash the key not very uniformly distributed, so that more collision will happens which will cause the result vary.