Implementing Hash Tables

Names: Cornelis Boon(10561145), Tim Groot(10165673)

Introduction

A hash table, also known as a hash map, is a data structure in the form of an array that not only has values in them, but each value corresponds with a so called key. A hash table is, most of the times, far more effective for putting and getting information to or from an array than a regular one. The worst possible time complexity for hash tables is *O(n)*, meaning linear. The key that corresponds the a place in the unordered associative array is calculated using a hash function. This function can range from simple functions like $key - value \bmod N$ to more complex ones like MAD[1] and binary operations.
The goal of this assignment is to benchmark our own implementation of hash tables with *SpellChecker.java*. One implementation uses closed-addressing, also known as chaining. To solve collisions and the other implementations you will need to use some form of open-addressing, such as linear probing, quadratic probing or double hash probing.

Implementation discussion

For the implementation of the two hashtables, we have created the following files. Per file, we will discuss shortly about why they have been implemented the way they are, if this is needed.

Interfaces

For the various classes we have written a few interfaces:

Entry:
An interface for key-value objects.

HashTable:
An interface for the hash table classes

[1]Multiple, Add, Divide

Chaining

The process of hashing with collision chaining is done by three files: *HashTable-Closed.java*, *ChainedList.java* and *ChainedEntry.java*

HashTableClosed:
: Main class which implements the *HashTable* interface. Passes the get and put-requests from Spellchecker to the appropriate ChainedList.

ChainedList:
: Class that implements a double *LinkedList*. Every item is a *ChainedEntry*. Handles the get and put-requests that it gets from HashTable Closed.

ChainedEntry:
: *ChainedEntry* implements the Entry interface. It is a node within the *ChainedList* with a reference to the node ahead and behind it. With these references, the List can be searched from both ends at the same time, thus it requires only half the search time to find an entry.

Open addressing

Open addressing uses the files *OpenEntry.java* and *HashTableOpen.java*

OpenEntry:
: An implementation of the Entry interface. A simple key-value pair class.

HashTableOpen:
: A class that implements the HashTable interface. Consists of get and put methods, as well as a resize method for cases when the initially chosen size ends up being not large enough.

Design choices

Chaining

For the hash table that uses collision chaining, we have implemented double linked lists. We have chosen this over a single linked list so that in the case that a small hash size is specified, which would make the linked lists quite large, these lists still can be fairly easily searched from both ends.

Open addressing

In open addressing, there is only room for one value per slot, unlike the chained implementation that uses linked lists. Due to this, collisions will have to be solved differently. To work around this, we have implemented a linear probing solution. This means that if a collision happens, it will try to find the next best empty slot to store the key-value pair in. It does this by:

Step 1: Iterating from the index that has been calculated to the end of the table.

Step 2: If step 1 wasn't successful, it will then try the part of the table it missed by iterating from the start up to the calculated index.

Step 3: If step 2 is unsuccessful, it will re-size the table, calculate a new index and try step 1 and 2 again.

Experiments and benchmarks

We have produced the following benchmarks using the four parameters hash method, array size, initial value and multiplier. The program is run with the command:
java -Xint SpellChecker txt/british-english-insane txt/origin-of-species-ascii.txt Array Size Method

Figure 1 shows the benchmarks done on *origin-of-species-ascii.txt* with varying array size. The initial and multiplier values are constant here.
Figure 2 shows different combinations of small and large, prime and non-prime numbers for *Initial* and *Multiplier*.
The last figure, Figure 3 shows the effect of a varying load factor on the array size and calculation time.

Figure 1: Varying *Array Size*:

Linear Probing / Chaining	Array Size	Initial	Multiplier	Put(ms)	Get(ms)	Load Factor
Linear Probing	18000	11	31	51233	77883	0.554068
Chaining	18000	11	31	13243	3736	35.460333
Linear Probing	36000	11	31	56756	79662	0.554068
Chaining	36000	11	31	13840	4748	17.730167
Linear Probing	566000	11	31	30555	86935	0.563857
Chaining	566000	11	31	8930	3166	1.127714

Figure 2: Varying *Initial* and *Multiplier*:

Linear Probing / Chaining	Array Size	Initial	Multiplier	Put(ms)	Get(ms)	Load Factor
Linear Probing	9000	11	31	37460	56441	0.554086
Chaining	9000	11	31	13691	3496	70.92
Linear Probing	9000	3	99983	16504	69523	0.554068
Chaining	9000	3	99983	16014	4465	70.920667
Linear Probing	9000	12	32	62650	61155	0.554068
Chaining	9000	12	32	28015	4171	70.92
Linear Probing	9000	2	132932	18119	5966	0.554068
Chaining	9000	2	132932	17821	3824	70.920667

Figure 3: Varying *Load Factor*:

Linear Probing / Chaining	Array Size	Initial	Multiplier	Put(ms)	Get(ms)	Load Factor
Linear Probing	638286	11	31	45541	252716	1.0
Chaining	638286	11	31	8155	2887	1.0
Linear Probing	800000	11	31	9371	64187	0.8
Chaining	800000	11	31	9257	3454	0.8
Linear Probing	912000	11	31	8273	61504	0.7
Chaining	912000	11	31	8666	3196	0.7

Conclusion

From the benchmarks shown in Figures 1-3, we have concluded the following about open- and closed addressing:

- When the array size is increased and the initial and multiplier values are unchanged (Figure 1), the time it takes to put an item into that array does not necessarily increase for linear probing. For closed addressing with linked lists, one might even argue from these benchmarks that the time decreases. To verify this, however, more tests are needed. Retrieving an item from an increasingly bigger array may take a little longer for linear probing, but stays the same for chaining. The complexity here for linear probing is even less than linear, since the calculation time increases linearly, while the array increases exponentially.

- Benchmarks with varying prime and non-prime numbers are shown in Figure 2. The first and fifth row clearly show that prime numbers are a better choice, since they are far more rare when the hash function uses modulo.
  Furthermore, linear probing turns out to be a lot faster in some scenario's when the multiplier becomes greater.

- Benchmarks where the load factor is kept constant are shown in Figure 3. The execution time of the put and get requests for the chaining implementation remains the same regardless of the load factor. *Linear Probing*, however, shows a vast decrease in time, when the load factor decreases. This is likely due to that when there is a large enough size dedicated for the array, such that a resize is not needed, and there is enough open slots left in the array, the hash table can process requests more easily.