A Comparison of Dictionary Implementations

Mark P Neyer

April 10, 2009

1 Introduction

A common problem in computer science is the representation of a *mapping* between two sets. A mapping $f : A \rightarrow B$ is a function taking as input a member $a \in A$, and returning b, an element of B. A mapping is also sometimes referred to as a dictionary, because dictionaries map words to their definitions. Knuth [?] explores the map / dictionary problem in Volume 3, Chapter 6 of his book *The Art of Computer Programming.* He calls it the problem of 'searching,' and presents several solutions.

This paper explores implementations of several different solutions to the map / dictionary problem: *hash tables*, *Red-Black Trees*, *AVL Trees*, and *Skip Lists*. This paper is inspired by the author's experience in industry, where a dictionary structure was often needed, but the natural C# hash table-implemented dictionary was taking up too much space in memory. The goal of this paper is to determine what data structure gives the best performance, in terms of both memory and processing time. AVL and Red-Black Trees were chosen because Pfaff [?] has shown that they are the ideal balanced trees to use. Pfaff did not compare hash tables, however. Also considered for this project were *Splay Trees* [?].

2 Background

2.1 The Dictionary Problem

A *dictionary* is a mapping between two sets of items, K, and V. It must support the following operations:

1. **Insert** an item v for a given key k. If key k already exists in the dictionary, its item is updated to be v.
2. **Retrieve** a value v for a given key k.
3. **Remove** a given k and its value from the Dictionary.

2.2 AVL Trees

The AVL Tree was invented by G.M. Adel'son-Vel'skiĭ and E. M. Landis, two Soviet Mathematicians, in 1962 [?]. It is a self-balancing binary search tree data structure. Each node has a *balance factor*, which is the height of its right subtree minus the height of its left subtree. A node with a balance factor of -1,0, or 1 is considered 'balanced.' Nodes with different balance factors are considered 'unbalanced', and after different operations on the tree, must be rebalanced.

1. **Insert** Inserting a value into an AVL tree often requires a tree search for the appropriate location. The tree must often be re-balanced after insertion. The re-balancing algorithm 'rotates' branches of the tree to ensure that the balance factors are kept in the range [-1,1]. Both the re-balancing algorithm and the binary search take $O(\log n)$ time.

2. **Retrieve** Retrieving a key from an AVL tree is performed with a tree search. This takes $O(\log n)$ time.
3. **Remove** Just like an insertion, a removal from an AVL tree requires the tree to be re-balanced. This operation also takes $O(\log n)$ time.

2.3 Red-Black Trees

The Red-Black Tree was invented by Rudolf Bayer in 1972 [**?**]. He originally called them ”Symmetric Binary B-Trees”, but they were renamed ”Red-Black Trees” by Leonidas J. Guibas and Robert Sedgewick in 1978 [**?**]. Red-Black trees have nodes with different ’colors,’ which are used to keep the tree balanced. The colors of the nodes follow the following rules:

1. Each node has two children. Each child is either red or black.
2. The root of the tree is black
3. Every leaf node is colored black.
4. Every red node has both of its children colored black.
5. Each path from root to leaf has the same number of black nodes.

Like an AVL tree, the Red-Black tree must be balanced after insertions and removals. The Red-Black tree does not need to be updated as frequently, however. The decreased frequency of updates comes at a price: maintenance of a Red-Black tree is more complicated than the maintenance of the AVL tree.

1. **Insert** Like an AVL Tree, inserting a value into a Red-Black tree is done with a binary search, followed by a possible re-balancing. Like the AVL tree, the re-balancing algorithm for the Red-Black tree ’rotates’ branches of the tree to ensure that some constraints are met. The difference is that the Red-Black Tree’s re-balancing algorithm is more complicated. The search and re-balancing both run in $O(\log n)$ time.
2. **Retrieve** Retrieving a key from a Red-Black tree is performed with a binary search. This takes $O(\log n)$ time.
3. **Remove** Just like an insertion, a removal from a Red-Black tree requires the tree to be re-balanced. The search for the key to be removed, along with the re-balancing, takes $O(\log n)$ time.

2.4 Skip Lists

Skip Lists are yet another structure designed to store dictionaries. The skip list was invented by William Pugh in 1991 [**?**]. A skip list consists of parallel sorted linked lists. The ’bottom’ list contains all of the elements. The next list up is build randomly: For each element, a coin is flipped, and if the element passes the coin flip, it is inserted into the next level. The next list up contains the elements which passed *two* coin flips, and so forth. The skip list requires little maintenance.

1. **Insert** Insertion into a skip list is done by searching the list for the appropriate location, and then adding the element to the linked list. There is no re-balancing that must be done, but because a search must be performed, the running time of insertions is the same as the running time for searches : $O(\log n)$.
2. **Retrieve** Retrieving a key from a Red-Black tree is performed with a special search, which first traverses the highest list, until an element higher than the search target is found. The search goes back one element of the highest list, then goes down one level and continues. This process goes on until the bottom list is found, and the element is either located or it is found that the element is not part of the list. how long does this search take? Pugh showed that it takes, on average, $O(\log n)$ time.

3. **Remove** Removal of an item from the skip list requires no maintenance, but it again requires a search. This means that the removal takes $O(\log n)$ time.

2.5 Hash tables

Hash tables were first suggested by H. P. Luhn, in an internal IBM memo in 1953 [**?**]. Hash tables use a *hash function* $h : K \rightarrow V$ to compute the location of a given value v in a table. The function is called a 'hash function' because it 'mixes' the data of its input, so that the output for similar inputs appears totally unrelated. When two members of K, say k_1 and k_2 have the same hash value, i.e. $h(k_1) = h(k_2)$, then we say there is a hash *collision*, and this collision must be resolved.

There are two main methods of collision resolution. Under the chaining method, each entry in the hash table contains a linked list of 'buckets' of keys with the same hash value. When a lookup is performed in the hash table, first the appropriate bucket is found, and then the bucket is searched for the correct key. If there are M lists of buckets in the hash table, and there are N total entries, we say that the hash table has a *load factor* of N/M. When the load factor gets too high, it makes sense to create a new hash table with more bucket lists, to reduce the time of each operation.

The second method of collision resolution is called 'open addressing'. Under open addressing, each hash function hashes to not one but a series of addresses. If an entry is performed and the first address in the hash table is occupied, the next is probed. If it is occupied, the next is probed, and so on, until an empty address is found. Hash tables with open addressing therefore have a total maximum size M. Once the the load factor (N/M) exceeds a certain threshold, the table *must* be recopied to a larger table.

1. **Insert** Inserting a value into a Hash table takes, on the average case, $O(1)$ time. The hash function is computed, the bucked is chosen from the hash table, and then item is inserted. In the worst case scenario, all of the elements will have hashed to the same value, which means either the entire bucket list must be traversed or, in the case of open addressing, the entire table must be probed until an empty spot is found. Therefore, in the worst case, insertion takes $O(n)$ time.
2. **Retrieve** Retrieving a key from a Red-Black tree is performed by computing the hash function to choose a bucket or entry, and then comparing entries until a match is found. On average, this operation takes $O(1)$ time. However, like insertions, in the worst case, it can take $O(n)$ time.
3. **Remove** Just like an insertion, a removal from a hash table is $O(1)$ in the average case, but $O(n)$ in the worst case.

3 Experiment

The Microsoft .NET platform was chosen for this experiment because it is of interest to the author's experience in industry, and due to the availability of a precise memory measurement debugging library, namely *sos.dll.* The hash table chosen was the *System.Collections.hash table* class, and the AVL tree chosen was based upon an implementation found at http://www.vcskicks.com. The Red-Black tree was based upon an implementation found at http://www.devx.com/DevX/Article/36196, and the Skip List used was based upon an implementation found at http://www.codeproject.com/KB/recipes/skiplist1.aspx. The tests were performed in rounds, with the dictionary size in each round varying from 5 entries to 5120 entries. For each round a dictionary was created and then populated with random elements. The time it took to create the dictionary, and memory taken up were then measured. Next, the dictionary was queried randomly, 100 times for each entries in the dictionary. The time it took to perform these queries was recorded.

Measurement of the size of the hash table was made difficult by the fact that the .NET runtime uses hash tables internally, and the measuring tools did not filter these tables out. Therefore, when memory taken up by instances of the Hashtable or the Hashtable+bucket[] class was taken, a baseline for the number of existing hash tables and their size was computed based upon memory used when no instance of the Hashtable class were created for the experiment. This baseline memory usage was then subtracted from all future measurements of memory usage. The experimental data is presented both as it was gathered, as well as in a processed form to account for the existing hash tables.

4 Results

Table **??** compares the memory consumption of the different implementations of dictionaries. For all but the most trivial of dictionaries, the hash table outperforms AVL and Red-Black Trees, but loses to the Skip List.

Table **??** compares the time it took to create thedifferent dictionary implementations. The AVL tree is the clear loser, while Red-Black Trees, Hash tables, and Skip Lists all appear to take roughly the same amount of time.

Table **??** compares lookup time in the different structures. Once again, Hash Tables are the superior choice in terms of fast lookup time, with skip lists coming in second and red-black trees a close third. The clear loser is, again, the AVL tree.

5 Conclusion

It is very clear now that, for random access dictionaries, hash tables are superior to both trees and skip lists. The primary advantage of the later group of data structures is that an they allow for sorted access to data. A hash table's speed, on the other hand, depends upon its ability to store the elements 'randomly', i.e. with no relation to the relative values of the keys.

Among trees, it is clear that a Red-Black tree is superior to an AVL tree. The Red-Black tree allows itself to be more 'unbalanced' than an AVL tree, and, as a result, does not need to be re-balanced as frequently. A Skip list, however, appears to be a better choice than either of these two trees. Its memory footprint is smaller than any of the other structures, and in construction time it is comparable to a Hash table. For random queries, it performs about as well as the Red-Black tree.

6 Bibliogrpahy

References

[1] G.M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1926.

[2] R. Bayer. Symmetric binary b-trees: Data structures and maintenance algorithms. *Acta Informat.*, 1:290–306, 1972.

[3] L. J. Guibas and R Sedgewick. A dichromatic framework for balanced trees. *IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[4] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Co., Philippines, 1973.

[5] Ben Pfaff. Performance analysis of BSTs in system software. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 410–411, New York, NY, USA, 2004. ACM.

[6] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[7] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

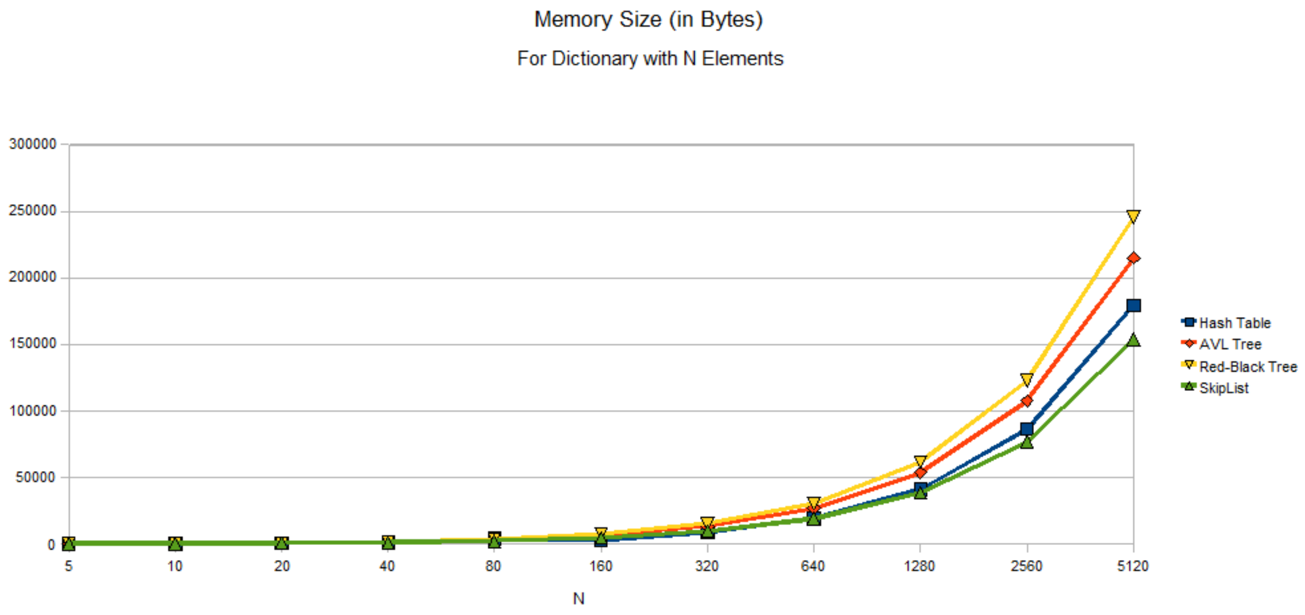
7 Data

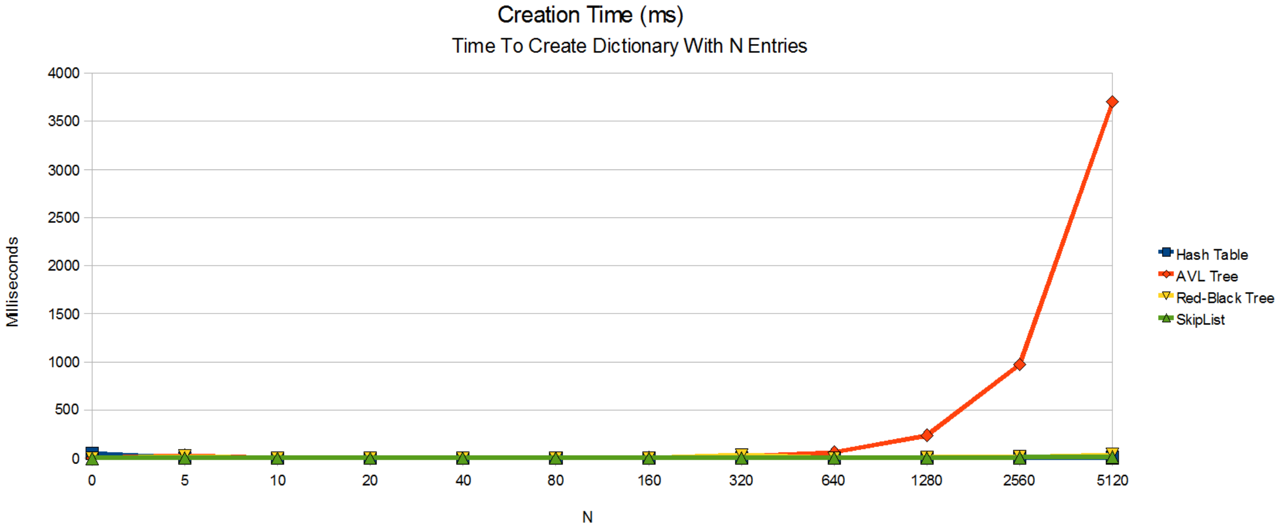
Number of Entries	Hash Table	AVL Tree	Red-Black Tree	Skip List
5	330	176	224	160
10	344	456	544	360
20	776	876	1024	660
40	1784	1716	1984	1260
80	4088	3396	3904	2460
160	3066	6756	7744	4860
320	9330	13476	15424	9660
640	19424	26916	30784	19260
1280	41168	53796	61504	38460
2560	86264	107556	122944	76860
5120	179504	215076	245824	153660

Table 1: Dictionary Memory Size In Bytes

Number of Entries	Hash Table	AVL Tree	Red-Black Tree	Skip-list
5	2	30	24	4
10	1	4	1	1
20	1	2	1	1
40	1	2	1	2
80	1	3	2	1
160	2	6	2	1
320	2	18	33	2
640	2	62	4	3
1280	2	238	9	4
2560	4	975	14	7
5120	6	3704	38	16

Table 2: Dictionary Creation Time In Milliseconds



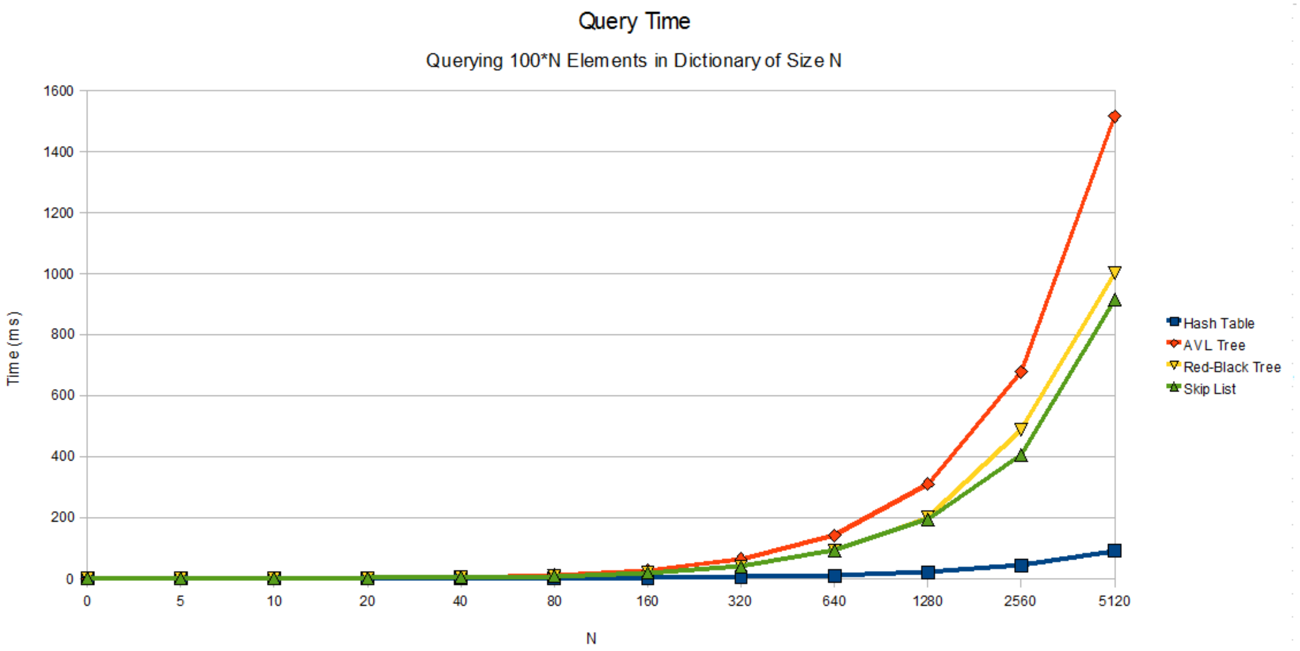


Number of Entries	Hash Table	AVL Tree	Red-Black Tree	Skip List
5	0	1	1	1
10	0	1	1	0
20	0	2	2	2
40	1	5	4	4
80	1	13	9	9
160	3	28	19	19
320	6	64	41	41
640	10	143	92	93
1280	21	310	202	195
2560	44	679	489	406
5120	91	1517	1002	914

Table 3: Dictionary Lookup Speed In Milliseconds

Size of Table	Number of Tables	Number of Bucket Arrays	Size of Bucket Arrays	Average Size of Bucket Arrays	Corrected Size of Bucket Arrays	Total Size (Bytes)
0	56	64	18264	285.38	285	341
5	54	55	15096	274.47	274	330
10	56	65	18552	285.42	288	344
20	54	56	15816	282.43	720	776
40	54	56	16824	300.43	1728	1784
80	54	56	19128	341.57	4032	4088
160	54	55	17832	324.22	3010	3066
320	54	55	24096	438.11	9274	9330
640	54	56	34464	615.43	19368	19424
1280	54	56	56208	1003.71	41112	41168
2560	54	56	101304	1809	86208	86264
5120	54	56	194544	3474	179448	179504

Table 4: Hash Tables: Memory Size in Bytes


Query Time
Querying 100*N Elements in Dictionary of Size N
Time (ms)
1600
1400
1200
1000
800
600
400
200
0
0
5
10
20
40
80
160
320
640
1280
2560
5120
N
Hash Table
AVL Tree
Red-Black Tree
Skip List