Option #1: Linear Probing

Scott Miner

Colorado State University – Global Campus

Abstract

```
Keys:
[[46, 6, 27, 45, 36, 42, 16, 6, 21, 36]]
```

Figure 1. Keys inserted into hash tables that use linear and double-hashing algorithms

```
Linear Probe: initial table:
   --------
 0:|  16   |
 1:|  6    |
 2:|  42   |
 3:|  21   |
 4:|  36   |
 5:|  45   |
 6:|  46   |
 7:|  6    |
 8:|  27   |
 9:|  36   |
   --------
```

Figure 2. Results from inserting ten items into a hash table using a linear probing sequence

```
Double Hash: initial table:
   --------
 0:| E/S   |
 1:|  45   |
 2:|  46   |
 3:|  36   |
 4:|  36   |
 5:|  27   |
 6:|  6    |
 7:|  6    |
 8:|  21   |
 9:|  42   |
10:|  16   |
   --------
```

Figure 3. Results from inserting ten items into a hash table using a double-hashing sequence

```
=====================
Linear Probe Results
=====================
        Key  Probes
0        46       0
1         6       1
2        27       1
3        45       0
4        36       3
5        42       0
6        16       4
7         6       5
8        21       2
9        36       8
Total   NaN      24
```

Figure 4. Number of probes needed to insert ten items in a hash table with linear probing

```
=====================
Double Hash Results
=====================
        Key  Probes
0        46       0
1         6       0
2        27       0
3        45       0
4        36       0
5        42       0
6        16       1
7         6       1
8        21       6
9        36       2
Total   NaN      10
```

Figure 5. Number of probes needed to insert ten items in a hash table with double-hashing

```
Numbers Inserted
----------------
   0   1   2   3   4   5   6   7   8  ...  91  92  93  94  95  96  97  98  99
0  33  50  82  75  49  97  18  22  89  ...  39  16  65  79  37  35  85  59  51

[1 rows x 100 columns]
                       92  93  94  95  96  97  98  99 Total % Total Dec.  Time (sec)  \
Probe Sequence Result
Linear Probing Key     16  65  79  37  35  85  59  51     0     -146.93%  0.02270940
               Probes   0  46  33   4   7  28  55   0   442     -146.93%  0.02270940
Double-Hashing Key     16  65  79  37  35  85  59  51     0       59.50%  0.02160530
               Probes   0  10   6  15  12   6  13   0   179       59.50%  0.02160530

                      % Time Dec.  Num. Lists  Num. Elements  Hash Table Size
Probe Sequence Result
Linear Probing Key          -5.11%          1            100              211
               Probes       -5.11%          1            100              211
Double-Hashing Key           4.86%          1            100              211
               Probes        4.86%          1            100              211
```

Figure 6. Results after inserting 100 unique random integers into linear probing and double-hashing hash tables

```
Numbers Inserted
----------------
   0    1    2    3    4    5    6  ...  993  994  995  996  997  998  999
0  285  704  471  819   59  298  195  ...   44  136  276  975  738  907  363

[1 rows x 1000 columns]
                      992 993  994  995  996  997  998  999  Total % Total Dec.  \
Probe Sequence Result
Linear Probing Key     682  44  136  276  975  738  907  363      0     -174.94%
               Probes  410  16    1    0  118  356  188    5  32273     -174.94%
Double-Hashing Key     682  44  136  276  975  738  907  363      0       63.63%
               Probes  105   0    9    0   24   93   62    0  11738       63.63%

                       Time (sec) % Time Dec.  Num. Lists  Num. Elements  Hash Table Size
Probe Sequence Result
Linear Probing Key     0.58818540     -26.23%           1           1000             1931
               Probes  0.58818540     -26.23%           1           1000             1931
Double-Hashing Key     0.46596410      20.78%           1           1000             1931
               Probes  0.46596410      20.78%           1           1000             1931
```

Figure 7. Results after inserting 1,000 unique random integers into linear probing and double-hashing hash tables

```
Numbers Inserted
----------------
   0     1     2     3     4     5   ...  9994  9995  9996  9997  9998  9999
0  6984  1910  8585  2392  4366  1176  ...   700  5176  1261  1973  7160  1624

[1 rows x 10000 columns]
                      9992  9993  9994  9995  9996  9997  9998  9999    Total % Total Dec.  \
Probe Sequence Result
Linear Probing Key     2203  5613   700  5176  1261  1973  7160  1624        0     -156.09%
               Probes     6     2     0     0    28    38  3595    26  3938455     -156.09%
Double-Hashing Key     2203  5613   700  5176  1261  1973  7160  1624        0       60.95%
               Probes     3     0  1568     4     4    31  3538  1437  1537904       60.95%

                      Time (sec) % Time Dec.  Num. Lists  Num. Elements  Hash Table Size
Probe Sequence Result
Linear Probing Key    75.67305950     -23.78%           1          10000            17341
               Probes 75.67305950     -23.78%           1          10000            17341
Double-Hashing Key    61.13698650      19.21%           1          10000            17341
               Probes 61.13698650      19.21%           1          10000            17341
```

Figure 8. Results after inserting 10,000 unique random integers into linear probing and double-hashing hash tables

Option #1: Linear Probing

For his fifth critical thinking assignment in CSC506: Design and Analysis of Algorithms, the student illustrates the linear probing method of hashing and discusses how to overcome its shortcomings. Lysecky and Vahid (2019) define hash tables as data structures that map unordered items into array locations known as *buckets*. Hashing functions often use the modulo operator, %, which returns the integer remainder after dividing two numbers, to compute bucket indices from item keys. For instance, when searching for available buckets, the linear probing algorithm searches items consecutively, one after another, and assumes the following form: $newLocation = (startingValue + stepSize) \% arraySize$, where the step size is incremented linearly after each search (Bello *et al.*, 2014).

Lysecky and Vahid (2019) describe collisions as occurring when a hash table attempts to insert an element into an index already occupied. Open addressing is a collision resolution technique that stores new items in subsequently available buckets. Nimbe *et al.* (2014) describe the advantages of open addressing, including not needing to use additional data structures to store elements and being efficient storage-wise. Some demerits include needing to flag bucket states (e.g., empty-since-start vs. empty-after-removal), requiring unique keys, and choosing proper table sizes. Table sizes are often chosen using the smallest prime number ≥ $\frac{\text{Number of elements in the table}}{\text{Desired load factor}}$ or the smallest prime number ≥ $n \times 2$, where n is the number of elements in the table (Lysecky & Vahid, 2019).

Bello *et al.* (2014) write that the load factor is one possible threshold to determine when to resize a hash table. The load factor is the number of elements in the hash table divided by the number of buckets. Bello *et al.* found that the average lookup cost of a hash table search function was "nearly constant as the load factor increases from 0 up to .7 or so" (p. 685), at which point the "table's speed drastically degrades" (p. 686). Therefore, the length of a probe sequence is proportional to the $\frac{\text{load factor}}{(1 - \text{load factor})}$. Furthermore, the time complexity of a hash table resize is *O*(*n*) since all items from the old array need to be re-inserted into the new array (Lysecky & Vahid, 2019).

Hash tables with linear probing fall under the open addressing collision resolution category. When a hash table with linear probing attempts to insert an item into a bucket already occupied, the algorithm searches linearly for the next available bucket, inserting the item if an empty bucket is found. If the algorithm reaches the hash table's end, probing restarts at the zeroth index (Lysecky & Vahid, 2019). Figure 1 shows a list of ten elements inserted into a hash table with linear probing. Figure 2 shows the initial hash table after inserting all elements, and Figure 4 shows the number of probes needed to insert each element along with the total number of probes. Flajolet *et al.* (1998) define *displacement* as the circular distance between an element's inserted location and its initial hash value. Thus, displacement measures the cost of inserting or searching for an element in a hash table. Moreover, *total displacement* measures the *construction cost* of a hash table and is the total sum of all item displacements.

Flajolet *et al.* (1998) describe *sparse tables* as those with filling ratios, n/m, less than 1, where m represents the number of table locations and n the number of keys. When using hash tables with linear probing, the construction cost of a sparse table has an average of *O*(*n*) and a standard deviation of $O(\sqrt{n})$. In contrast, *full* $(m = n)$ and *almost full* $(m = n - 1)$ tables have average construction costs of $O(n^{3/2})$ and standard deviations of the same order, indicating that late insertions into full or nearly full tables have highly dispersed superlinear costs. Additional disadvantages of linear probing sequences include primary clustering, long probing sequences, and deteriorated performance caused by large clusters. *Primary clustering* occurs when all initial hash values produce the same probe sequence for a given constant (Luo & Heileman, 2003).

One way to overcome these challenges is to use *rehashing*, also known as *double-hashing*, which is another open addressing collision resolution technique that eliminates clustering problems by using a second hash function to compute probing sequences: $\left(h1(key) + i \times h2(key)\right) \, mod \; (tablesize)$ (Bello *et al.*, 2014). The second hash function has two requirements: (a) it must never evaluate to zero and (b) it must ensure that all buckets can be probed (Bello *et al.*, 2014; Luo & Heileman, 2003). Figure 3 shows the results of inserting ten items into a hash table with double-hashing. The hash table's initial size is a primary number: 11. Once all items have been inserted, bucket zero remains marked as empty-since-start since no keys were ever inserted into it. Figure 5 shows the number of probes needed to insert each key into the double-hashing hash table, along with the total number of probes. The double-hashing probe sequence used only ten probes compared to the linear probing sequence, which used 24. By eliminating the primary clustering problem seen in linear probing, the double-hashing technique improved the cost construction of the hash table.

Figures 6 – 8 expand upon this idea, showing the results after inserting 100, 1,000, and 10,000 elements into linear probing and double-hashing hash tables. The images display the number of probes needed to insert the last eight elements in each probing sequence, along with the sizes of each hash table and performance metrics. The double-hashing sequence reduced the performance time of the linear probing sequence by approximately 20% and reduced the number of probes by approximately 60%. In conclusion, this paper illustrated the linear probing method in hashing, explained its performance analysis, and discussed how to overcome its shortcomings using rehashing.

References

Bello, S. A., Liman, A. M., Gezawa, A. S., Garba, A., & Ado, A. (2014). *COMPARATIVE ANALYSIS OF LINEAR PROBING, QUADRATIC PROBING, AND DOUBLE-HASHING TECHNIQUES FOR RESOLVING COLLISION IN A HASH TABLE*. *5*(4), 3.

Flajolet, P., Poblete, P., & Viola, A. (1998). On the Analysis of Linear Probing Hashing. *Algorithmica*, *22*(4), 490–515. https://doi.org/10.1007/PL00009236

Luo, W., & Heileman, G. L. (2003). Comparison of Different Open Addressing Hashing Algorithms. *Computers and Their Applications*, 1–4.

Lysecky, R., & Vahid, F. (2019). *Data Structures Essential: Pseudocode with Python Examples*. Zyante Inc. (zyBooks.com).

Nimbe, P., Ofori Frimpong, S., & Opoku, M. (2014). An Efficient Strategy for Collision Resolution in Hash Tables. *International Journal of Computer Applications*, *99*(10), 35–41. https://doi.org/10.5120/17411-7990