Performance of Different Lookup Methods on a Hash Table

Alexander Alvarez

HT P1 2024

Introduction

The time efficiency of different lookup methods on different implementations of a hash table is to be measured. The lookup methods will be benchmarked and their implementations will be discussed.

The general behaviours of the hash table data structure are described as well as how the operations were created and implemented for benchmarking.

Hash table

A hash table data structure manages data using an array where the stored objects are identified by specific key-values. These keys are then processed using a hash function which maps a key to an index in the array.

A CSV file is provided with a sorted order of zip codes that will act as the keys in the following tests.

String or integer key using linear and binary search

The first benchmark tests the linear and binary search on two implementations of the hash table. The first implementation uses a string as a key while the other converts the string to an integer.

The data from the CSV file are stored in the array in a desceding order, starting from the top of the file. The size of the array, denoted `max`, is the amount of entries added to the array.

The linear search method is a `while` loop that for each index compares a given key to the key in a stored object. The index starts at `0` and increments by 1 for each iteration. The method returns `true` if the given key is equal to the stored key. The loop breaks and returns `false` if the index value reaches the `max` value or if the key in the stored object is greater than the given key.

```
public boolean linSearch(String key){
  int index = 0;
  while(i < this.max && this.postnr[index].code.compareTo(key) < 0)
    if(this.postnr[index++].code.equals(key)) return true;
  return false;
}
```

The binary search is the same as the one from a previous assignment but modified to also work with a string value. The benchmark for linear and binary search is run using both string and integers as keys. The keys selected for the benchmark are the first and last objects in the array. The benchmark yelds the following results displayed in table 1.

Search	**key**	**integer**	**string**
Linear	111 15	$7.7ns$	$6.8ns$
Linear	984 99	$15000ns$	$46000ns$
Binary	111 15	$23ns$	$90ns$
Binary	984 99	$17ns$	$65ns$

Table 1: Runtime of linear and binary search using string and integer keys.

Using integers as keys is faster than using strings. Binary search is more efficient when it comes to larger arrays with the time complexity $O(logn)$ which we already know. The time for linear search increases as the size increases which gives the complexity $O(n)$. The keys used for the following tests will be integers due to being faster.

Key as index

The second benchmark tests a lookup method on an implementation of the hash table where an object is stored in an index value that is equal to the object's key value.

The array size needs to be greater than every key value to enable indices with the same value as the keys to be used. The array is given the size `100000` since the highest possible key value is 99999.

The constructor in this implementation inserts the objects from the CSV file to the array indices that corresponds to the objects' keys. The lookup method is then a simple `if` statement that returns `true` if an object exists at a given index and `false` if that index is empty.

Running the benchmark with different keys where some don't exist in the CSV file, yields the results displayed in table 2.

The keys that exists averages towards $9.7ns$ while the keys that don't exist averages towards $6.6ns$. The complexity is nonetheless $O(1)$ regardless of whether the keys exists. This is a much more efficient time compared

key	runtime
111 15	$11ns$
199 99	$6.8ns$
211 22	$9.8ns$
454 32	$9.7ns$
454 54	$6.6ns$
820 01	$6.6ns$
984 99	$9.7ns$

Table 2: Runtime of lookup method where key = index.

to binary search. However, the problem is that the array is mostly empty which is an inefficient use of memory. This is where the hash function comes in.

Hash function and collisions

The third benchmark tests an implementation where a hash function maps the objects' keys to indices in the array.

The hash function returns an index value equal to the key value modulo some value `m` ($index = key(\mathrm{mod}m)$). A collision occurs when two different keys has the same hash index. Some code was provided to test out collisions with different values for `m`. Table 3 displays the amounts of hash indices that `m` maps a number of keys to. Column 1 shows the amount of indices with only one key mapped to them, column 2 shows the amount of indices where two keys collide and so on.

m	1	2	3	4	5	6	7	8	9
10000	2000	1100	550	330	200	99	56	39	9
20000	4200	1400	510	190	50	0	0	0	0
12345	5000	1800	310	33	1	0	0	0	0
17389	6300	1400	160	3	0	0	0	0	0
13513	5400	1700	290	12	0	0	0	0	0
13600	3400	1600	610	230	56	0	0	0	0
14000	3100	1300	610	320	130	1	0	0	0

Table 3: Amounts of hash indices containing (1-9) keys based on `m` value.

An `m` value with fewer collisions is better since we want as many keys as possible to have a unique hash index. The `m` value is also the array size because no index value will be greater than `m` post operation. The value `13513` will be used as `m` for the following test due to being relatively small and having relatively few collisions.

Using hash function

Collisions can be handled in many ways using buckets, among other methods. This implementation utilizes buckets as linear nodes to handle collisions.

The constructor retrieves a key and generates a hash index. A node containing the key and it's data is stored in the hashed index if that index is empty. If a node already exists, then the constructor checks if that node's reference is empty and so on until an empty reference is found at which the new node is stored.

The lookup method generates the corresponding hash index with the provided key. The provided key is compared to the key stored in the node and its references at that index for which true is returned if there's a match. False is returned if there was no match or if the index was empty.

```
public boolean lookup(Integer key){
  Integer hashIndex = (key % M);
  if(this.postnr[hashIndex] != null){
    Area current = this.postnr[hashIndex];
    while(current != null){
      if(current.code.compareTo(key) == 0) return true;
      current = current.next;
    }
  }
  return false;
}
```

This implementation has an access time with the complexity $O(1)$ in the first case where no more than 1 node is present at an index. The complexity becomes $O(n)$ in the second case where n is the amount of nodes in the same index. This is due to the method having to linearly search through all the nodes with the same index to find the node with the matching key.

There are other ways of handling collisions as previously mentioned. These other ways impacts the complexity in the second case where there are multiple elements in the same index. This impact depends on the method used.