CS3334 Hash Table Project Report

Name: Le Trung Hai
SID: 57072995

1. Introduction

Hash table is an abstract data type for holding keys or key-value pairs such that the average time complexity for searching, inserting, and deleting is $\Theta(1)$. In this project, we implement a hash table for storing `unsigned` in C++ using a few different methods and compare their performance characteristics.

In general, the characteristic of a hash table is determined by two main factors:

- Hashing scheme
- Collision resolution

Other factors such as load factor threshold, table size, resizing strategy, etc. also affect the time-space performance of a hash table.

2. Existing implementations

Before diving in, here is the table summarizing how hash table is implemented in some common programming languages and libraries.

Language/library	Hashing	Collision	Reference
C++ (MSVC)	Specialized for each type, FNV-1a	Separate chaining	[1]
C++ (gcc)	Specialized for each type, MurmurHashUnaligned2	Separate chaining	[2]
C++ (Abseil's Swiss Tables)	Multiplicative hashing, CityHash	Quadratic probing	[3]
Go	Native AES hash, fallback to wyhash	Separate chaining	[4], [5]
Java (OpenJDK)	Multiple algorithms (MD5, CRC32, Xorshift)	Separate chaining	[6], [7]
CPython	Custom	Open addressing	[8]
Rust	SipHash-1-3	Quadratic probing	[9]

3. Project details

This project implements three types of collision resolutions and two types of hash function, resulting in six different kinds of hash table. They derive from the following two abstract classes:

```
struct Hasher {
    unsigned hash(unsigned x);
    void set_size(unsigned size);
private:
    unsigned table_size;
}

struct HashTable {
    void Insert(unsigned x);
    void Delete(unsigned x);
    unsigned* Search(unsigned x);
}
```

The two hashing methods are:

- Division hashing: `Hasher_Div`
- Multiplicative hashing: `Hasher_Mul`

The three collision resolutions schemes are:

- Separate chaining (using linked list): `HashTable_SC`
- Open addressing (linear probing): `HashTable_LP`
- Open addressing (quadratic probing): `HashTable_QP`

3.1 Hash function

Division hashing

The division hashing (or modular hashing) method here is the simplest possible: given a key x and table size M, the hash value is

$$h(x) = x \bmod M.$$

Since all hash tables implemented here use power-of-2 for size, this is effectively using the least significant bits of x, discarding the higher order bits. This leads to a very easy way to produce collision: any x and y congruent modulo M will collide.

Multiplicative hashing

Slightly more sophisticated, multiplicative hashing computes the hash value for a key x and table size M as

$$h(x) = \lfloor M \cdot \{xa\} \rfloor,$$

where a is a real number and $\{xa\} = xa - \lfloor xa \rfloor$ is the fractional part of xa. This can be approximated alternatively by setting A close to $2^w \times a$, in which case we have

$$h(x) = \left\lfloor M \cdot \frac{xA}{2^w} \right\rfloor.$$

If $M = 2^p$, then this can be calculated very quickly by `h(x) = x * A >> (w - p)`. Here, the table size is a power of 2, so we opt for the final formula for speed and simplicity.

The problem here is which constant a or A to choose. Ideally, it should be a number mixing both 0 and 1 in its bit representation. Knuth reccommends $a = (\sqrt{5} - 1)/2$, the inverse golden ratio, for its nice property.[10]

3.2 Collision resolution

Separate chaining

The buckets in `HashTable_SC` is an array of `Entry`, which contains the data and a pointer to the chain.

```
struct Entry {
    unsigned data;
    Entry* next;
}
```

If collision occurs when inserting a key, we traverse the chain, returns early if the key is found, or else append it to the end of the chain. In effect, each slot in the bucket array is a singly linked list. Deletion and searching works like with a normal linked list.

The time complexities of the hash table operations depends on the length of the chains, which in turn depend on the data distribution and the chosen hash function. For uniformly random data, even division hashing can guarantee $\Theta(1)$ average time complexity for all three operations, since the chain length is then relatively short.

The space complexity is $\Theta(n)$. The space utilization rate here is rather abysmal: since `unsigned` is usually 4 bits while `Entry*` takes up 8 bits, only a third of memory is for storing data. With struct padding and table resizing, the real rate is even lower.

Linear probing

The buckets in `HashTable_LP` is an array of `Entry`, which contains the data and a boolean flag indicating if this slot had been occupied before.

```
struct Entry {
    unsigned data;
    bool had_value;
}
```

Given the hash value h and the table size M, the probing sequence is

$$h, h+1, h+2, h+3, h+4, \dots \mod M.$$

If collision occurs when inserting a key, we traverse the probing sequence, returns early if the key is found, until we encounter a never-occupied slot and insert the key in. Searching basically works the same, without actually inserting the key. Deletion also follows the same probing process, in which if we encounter the key, then its `data` is set to a sentinel value indicating null, but the `had_value` field is left intact. This is because deletion creates a hole in the probing sequence, so we need some indicator to ensure subsequent searching works.

One thing to note is the load factor L: for insertion to succeed, L must be smaller than 1, preferably 3/4 (deleted values also contribute to the load factor).

The time complexities of the hash table operations depends on hash value distribution. For nicely separated hash values, we can guarantee $\Theta(1)$ average time complexity for all three operations, since the probing length is then relatively short. As the load factor approaches 1, the worst case complexity becomes $O(n)$, since the elements are clustered together.

The space complexity is $\Theta(n)$. The memory usage is somewhat better, with 4/5 of the memory is for storing data. (However, array alignment means this number is probably only 1/2). More importantly, using an array here gives us locality of reference and is cache-friendly, which can mean better performance than the chaining version.

Quadratic Probing

The implementation of `HashTable_QP` is almost identical to that of `HashTable_LP`, with the only difference being the probing sequence. Here, we use the triangular sequence as generated by the quadratic polynomial $p(i) = i(i+1)/2$, which yields

$$h, h+1, h+3, h+6, h+10, \ldots \mod M.$$

Here, M being a power of 2 is crucial, since it means this sequence for i ranging from 0 to $M-1$ gives us all residues modulo M.[11] Consequently, insertion will always succeed, as long as there is an empty slot in the table.

3.3 Table size and load factor

The default table size for all three types is 32. When the load factor exceeds the threshold, the table size is doubled, and rehashing occurs. During rehashing, a new bucket array is allocated, while the old one is deleted, invalidating all previous pointers returned by `Search`. The default load factor threshold is 3/4.

Resizing is crucial for the open addressing ones, since insertion only works when their load factor is low enough. Rehashing in open addressing tables also marks deleted slots as never-occupied, reducing the load factor further.

Due to rehashing, the worst case complexity of insertion in all hash tables is $O(n)$, though the armotized cost is still $O(1)$.

4. Compile and run

4.1 Requirements

gcc, make, bash

These commands are for bash on Linux (and possibly macOS).

4.2 Compilation

In the project root are two files: `main.cpp`, which runs a hash table implementation on stdin, and `gen.cpp`, which generates test input files. To compile and run the generator, run

```
make
bin/gen SEED NUMOPERATIONS KEYUPPERBOUND
```

Then, to execute the hash table program on one of the input files, run

```
cat test/input-0.txt | bin/main X
```

where `X` indicates the hash table implementation we want. The allowed options are:

- 0: `std::unordered_set`, default hashing
- 1: separate chaining, division hashing
- 2: separate chaining, multiplicative hashing
- 3: linear probing, division hashing
- 4: linear probing, multiplicative hashing
- 5: quadratic probing, division hashing
- 6: quadratic probing, multiplicative hashing

5. Benchmarks

5.1 Test cases

Running `bin/gen` will generate 4 test input files under the `test` folder:

- `input-0.txt`
  - Random numbers
  - Operations: 40% search, 40% insert, 20% delete
- `input-1.txt`
  - Numbers congruent modulo 256, e.g. $1, 1 + 256, 1 + 256 \cdot 2, \ldots$
  - Operations: 40% search, 40% insert, 20% delete
- `input-2.txt`
  - Numbers in a range, e.g. $2507, 2508, 2509, \ldots$
  - Operations: 40% search, 40% insert, 20% delete
- `input-3.txt`
  - Random numbers
  - Operations: 40% search, 60% insert

To generate the correct output files, we can run

```
cat test/input-0.txt | bin/gen 0 > test/output-0.txt
```

which will use `std::unordered_set`.

5.2 Benchmarking

A script `benchmark.sh` can be used to benchmark any specific implementation. Run

```
./benchmark.sh X [NUM]
```

to benchmark implementation X (from 0 to 6) NUM times (defaulting to 50 times).

The default command to generate input is `bin/gen 191 100000 100000`; that is, the seed is 191, the number of operations is 100000, and the upper bound for key is 100000. The time taken is measured by the `time` bash builtin, and output as a sum of the user and system time.

5.3 Results

For $i = 0, 1, \ldots, 6$, we run `./benchmark.sh i` 5 times, then take the average time in seconds. The results are as follow

	`input-0.txt`	`input-1.txt`	`input-2.txt`	`input-3.txt`
`std::unordered_set`, default hashing	2.542	2.078	2.323	2.392
separate chaining, division hashing	2.335	2.870	2.190	2.169
separate chaining, multiplicative hashing	2.345	2.134	2.268	2.156
linear probing, division hashing	2.416	2.574	2.449	2.168
linear probing, multiplicative hashing	2.313	2.220	2.259	2.071
quadratic probing, division hashing	2.358	2.591	2.216	2.133
quadratic probing, multiplicative hashing	2.291	2.244	2.297	2.102

From this, we can make a few observations.

Firstly, most of the handmade implementations give better or comparable performance to the standard library one. While this is reassuring, the standard implementation must works with any data type, and so is subject to many more constraints.

Secondly, in almost all cases, multiplicative hashing tables perform better than division hashing counterparts. This is especially true with `input-1.txt`, where the run time of separate chaining tables decreases by 25% (from 2.870 to 2.134). We can see the use of a good hash function matters greatly here, as it is very easy to create collision with division hashing.

Finally, even though division hashing is bad for certain data distributions, for uniformly random data it is generally a good hash function, as seen in the first and last column of the table. A few other languages also make the same observation, such as Python, which hashes integers to themselves.

6. Conclusion

Overall, this has been an interesting project. A few remarks I draw from developing this project are

- Resizing is important for separate chaining to maintain its performance. At first, the separate chaining table had a fixed size at 256, and the performance was not as good, even though it can still work correctly.
- Fuzzing tests are useful for testing correctness. When I write the `Delete` methods for the open addressing tables, the program passed the test on OJ, but failed the fuzzing tests, from which I could identify the problem.

Unlike some data structures, hash table has a lot of factors impacting its characteristics, and there are many considerations for making a good hash table. There isn't enough time to explore the myriad of options for hashing method, collision resolution, bucket array allocations, resizing strategy, lookup optimization, etc. Still, I have fun doing this project.

---

1. unordered_map at microsoft/STL↩
2. What is the default hash function used in C++ std::unordered_map?↩
3. raw_hash_set.h at abseil/abseil-cpp↩
4. map.go at golang/go↩
5. alg.go at golang/go↩
6. HashTable.java at openjdk/jdk↩
7. synchronizer.cpp at openjdk/jdk↩
8. dictobject.c at python/cpython↩
9. map.rs source↩
10. Donald Knuth, "The Art of Computer Programming, Volume 3, Sorting and Searching", section 6.4, page 516-517.↩
11. This is also used in Abseil, as seen here. For a proof, see the comment in `hasher_mul.cpp`.↩