Hashing (Κατακερματισμός)

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

Efficient implementation of ADT Map

- We need fast **equality search**
- Balanced trees
  - AVL / B-trees / Red-black / …
  - Store `(key, value)` in each node
- Or any efficient implementation of **ADT Set**
  - Store `(key, value)` as elements in the set
- The above provide search in $O(\log n)$
  - But also ordered traversal, which is **not needed**!
- Can we do better?
  - Yes, using hashing!

Hashing

- We need to store a `(key, value)` pair
- Idea: use the **key** as an **index** in an array
- This is easy if `key` is a **small integer**
  - Insert: simply store `value` in `array[key]`
  - Find: read `array[key]`
- **Problem:** does not work when `key` is large (or not an integer)
  - Solution: apply a **hash function** that transforms **keys** to **indexes**

Example

- Keys: integers, eg $1,\ 3,\ 18$
- Store data in an array of size $M = 7$
  - called a **hash table**
- Use a simple **hash function**

$$h(k) = k \mod 7$$

- A pair `(key, value)` is stored at index $h(\texttt{key})$

Table T after Inserting keys $2, 10, 14, 19$

Table T	
0	14
1	
2	2
3	10
4	
5	19
6	

- Keys are stored in their **hash addresses**
- The cells of the table are often called **buckets (κάδοι)**

Insert 24

Table T	
0	14
1	
2	2
3	10
4	
5	19
6	

- **Collision**, $h(24) = 3$ is already taken
- **Resolution policy**
  - look at lower locations of the table to find a place for the key

Insert 24

Table T		
0	14	
1	$\mathbf{24}$	$\leftarrow$ 3rd probe
2	$\mathbf{2}$	$\leftarrow$ 2nd probe
3	10	$\leftarrow$ 1st probe
4		
5	19	
6		

$h(24) = 3$

Insert 23

Table T		
0	14	$\leftarrow$ 3rd probe
1	24	$\leftarrow$ 2nd probe
2	2	$\leftarrow$ 1st probe
3	10	
4		
5	19	
6	23	$\leftarrow$ 4th probe

$h(23) = 2$

Open Addressing

- **Open addressing**
  - The method of inserting colliding keys into empty locations
- **Probe**
  - The inspection of each location
  - The locations we examined are called a **probe sequence**
- **Linear probing**
  - Examine consecutive addresses

Double Hashing

- **Double hashing** uses non-linear probing by computing different probe decrements for different keys using a second hash function $p(Ln)$.
- Let us define the following probe decrement function:

$$p(n) = \max(1, \frac{n}{7})$$

Insert 24

Table T		
0	14	$\leftarrow$ 2nd probe
1		
2	2	
3	10	$\leftarrow$ 1st probe
4	24	$\leftarrow$ 3rd probe
5	19	
6		

$h(24) = 3$

We use a probe decrement of $p(24) = 3$

Insert 23

Table T		
0	14	
1		
2	2	$\leftarrow$ 1st probe
3	10	
4	24	
5	19	
6	23	$\leftarrow$ 2th probe

$h(23) = 2$

We use a probe decrement of $p(23) = 3$

Collision Resolution by Separate Chaining

- The method of **collision resolution by separate chaining (χωριστή αλυσίδωση)** uses a linked list to store keys at each table entry.
- This method should not be chosen if space is at a premium, for example, if we are implementing a hash table for a mobile device.

Example

Table T		
0	14	
1		
2	$\mathbf{2}$	$\rightarrow \mathbf{23}$
3	10	$\rightarrow 24$
4		
5	19	
6		

Good Hash Functions

- Suppose T is a hash table having entries whose addresses lie in the range 0 to $M - 1$.
- An **ideal hashing function** $h(k)$ maps keys onto table addresses in a **uniform and random** fashion.
- In other words, for any arbitrarily chosen key, any of the possible table addresses is equally likely to be chosen.
- Also, the computation of a hash function should be **very fast.**

Collisions

- A **collision** between two keys k and k' happens if, when we try to store both keys in a hash table T both keys have the same hash address $h(k) = h(k')$.
- Collisions are relatively frequent even in sparsely occupied hash tables.
- A good hash function should **minimize collisions**.
- The **von Mises paradox**: if there are more than 23 people in a room, there is a greater than 50% chance that two of them will have the same birthday $(M = 365)$.

Primary clustering

- Linear probing suffers from what we call **primary clustering (πρωταρχική συσταδοποίηση)**.
- A **cluster** (**συστάδα**) is a sequence of adjacent occupied entries in a hash table.
- In open addressing with linear probing such clusters are formed and then grow bigger and bigger. This happens because all keys colliding in the same initial location trace out identical search paths when looking for an empty table entry.
- Double hashing does not suffer from primary clustering because initially colliding keys search for empty locations along **separate** probe sequence paths.

Ensuring that Probe Sequences Cover the Table

- In order for the open addressing hash insertion and hash searching algorithms to work properly, we have to guarantee that every probe sequence used can probe **all locations** of the hash table.
- This is obvious for linear probing.
- Is it true for double hashing?

Choosing Table Sizes and Probe Decrements

- If we choose the table size to be a **prime number (πρώτος αριθμός)** M and probe decrements to be positive integers in the range $1 \leq p(k) \leq M$ then we can ensure that the probe sequences cover all table addresses in the range 0 to $M - 1$ exactly once.

Good Double Hashing Choices

- Choose the table size M to be a **prime number**, and choose probe decrements, any integer in the range 1 to $M - 1$.
- Choose the table size M to be a **power of 2** and choose as probe decrements any **odd integer** in the range 1 to $M - 1$.
- In other words, it is good to **choose probe decrements to be relatively prime with** M

Deletion

- The function for deletion from a hash table is left as an exercise.
- But notice that **deletion poses some problems**.
- If we delete an entry and leave a table entry with an empty key in its place then we destroy the validity of subsequent search operations because a search terminates when an empty key is encountered.
- As a solution, we can leave the deleted entry in its place and mark it as deleted (or substitute it by a special entry “available”). Then search algorithms can treat these entries as not deleted while insert algorithms can treat them as deleted and insert other entries in their place.
- However, in this case, if we have many deletions, the hash table can easily become clogged with entries marked as deleted.

Load Factor

The **load factor (συντελεστής πλήρωσης)** α of a hash table of size M with N occupied entries is defined by

$$\alpha = \frac{N}{M}$$

- The load factor is an important parameter in characterizing the performance of hashing techniques.

Performance Formulas

- Hash table of size M with exactly N occupied entries
  - load factor $\alpha = \frac{N}{M}$
- C_N : average number of probes during a **successful search**
- C'_N : average number of probes during an **unsuccessful search**
  - or insertion

Efficiency of Linear Probing

- For **open addressing with linear probing**, we have the following performance formulas:

$$C_N = \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

$$C_N\text{'} = \frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$$

- The formulas are known to apply when the table T is up to 70% full (i.e., when $a \leq 0.7$).

Efficiency of Double Hashing

- For **open addressing with double hashing**, we have the following performance formulas:

$$C_N = \frac{1}{a} \ln \frac{1}{1-\alpha}$$

$$C'_N = \frac{1}{1-\alpha}$$

Efficiency of Separate Chaining

For **separate chaining**, we have the following performance formulas:

$$C_N = 1 + \frac{1}{2}\alpha$$

$$C'_N = \alpha$$

Important

Important consequence of these formulas:

- The performance depends **only on the load factor** α
- **Not** on the **number of keys** or the size of the table

Theoretical Results: Apply the Formulas

- Let us now compare the performance of the techniques we have seen for different load factors using the formulas we presented.
- Experimental results are similar.

Successful Search

Load Factors

	0.10	0.25	0.50	0.75	0.90	0.99
Separate chaining	1.05	1.12	1.25	1.37	1.45	1.49
Open/linear probing	1.06	1.17	1.50	2.50	5.50	50.5
Open/double hashing	1.05	1.15	1.39	1.85	2.56	4.65

Unsuccessful Search

Load Factors

	0.10	0.25	0.50	0.75	0.90	0.99
Separate chaining	0.10	0.25	0.50	0.75	0.90	0.99
Open/linear probing	1.12	1.39	2.50	8.50	50.5	5000
Open/double hashing	1.11	1.33	2.50	4.00	10.0	100.0

Complexity of Hashing

- Use a hash table that is never more than **half-full** ($\alpha \leq 0.50$)
- If the table becomes more than half-full, we can expand the table by choosing a new table twice as big and by **rehashing** the entries in the new table.
- Suppose also that we use one of the hashing methods we presented.
- Then the previous tables show that successful search can never take more than 1.50 key comparisons and unsuccessful search can never take more than 2.50 key comparisons.
- So the behaviour of hash tables is independent of the size of the table or the number of keys, hence the complexity of searching is $O(1)$

Complexity of Hashing

- To enumerate the entries of a hash table, we must first sort the entries into ascending order of their keys. This requires time $O(n \log n)$ using a good sorting algorithm.
- Insertion takes the same number of comparisons as an unsuccessful search, so it has complexity $O(1)$ as well.
- Retrieving and updating also take $O(1)$ time.

Important observations

1. It **can** happen that all entries **hash to the same index**
    - So the **worst-case** complexity of search/insert is $O(n)$
    - But the **average-case** remains $O(1)$
        - Under the assumption of a good hash function
2. **Rehashing** takes $O(n)$ time
    - So the **real-time** complexity of insert is $O(n)$
    - But it happens rarely
        - So the **amortized-time** complexity is $O(1)$
        - Similarly to a dynamic array

Complexity, summary

Search	Worst-case	Average-case
Real-time	$O(n)$	$O(1)$
Amortized-time	$O(n)$	$O(1)$

Insert	Worst-case	Average-case
Real-time	$O(n)$	$O(n)$
Amortized-time	$O(n)$	$O(1)$

Load Factors and Rehashing

- Experiments and average case analysis suggest that **we should maintain**
  - $\alpha < 0.5$ for open addressing schemes
  - $\alpha < 0.9$ for separate chaining
- With open addressing, as the load factor grows beyond 0.5 and starts approaching 1, clusters of items in the table start to grow as well.
- **At the limit, when a is close to 1, all table operations have linear expected running times** since, in this case, we expect to encounter a linear number of occupied cells before finding one of the few remaining empty cells.

Load Factors and Rehashing

- If the load factor of a hash table goes significantly above a specified threshold, then it is common to require the table to be resized to regain the specified load factor. This process is called **rehashing (ανακατακερματισμός)** or **dynamic hashing (δυναμικός κατακερματισμός).**
- When rehashing to a new table, a good requirement is having the new array's size be **at least double** the previous size.

Summary: Open Addressing or Separate Chaining?

- Open addressing schemes save space but they are not faster.
- As you can see in the above theoretical results (and corresponding experimental results), the separate chaining method is either competitive or faster than the other methods depending on the load factor of the table.
- So, **if memory is not a major issue, the collision handling method of choice is separate chaining**.

Comparing ADT Map implementations

	Search	Insert	Delete	Ordered traversal
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Hashing	$O(1)$	$O(1)$	$O(1)$	$O(n \log n)$

Choosing a Good Hash Function

- Ideally, a hash function will map keys **uniformly and randomly** onto the entire range of the hash table locations with each location being equally likely to be the target of the function for a randomly chosen key.

Example of a Bad Choice

- **Keys**
  - Strings of 3 ASCII characters
  - 24-bit integer containing the 3 8-bit bytes
- Use open addressing with double hashing.
- Select a table size $M = 2^8 = 256$
- Define our hashing function as $h(k) = k \mod 256$

Example

- This hash function is a poor one because it **selects the low-order character of the three-character key** as the value of $h(k)$
- If the key is 321 , when considered as a 24-bit integer, it has the numerical value $3 \times 256^2 + 2 \times 256^1 + 1 \times 256^0$
- Thus when we do the modulo 256 operation, we get the value 1

Example

- “Similar” keys create collisions

$$h(AAA) = h(ABA) = h(ACA) = h(BAA) = \dots$$

- Thus this hash function will create and preserve clusters instead of **spreading** them as a good hash function will do.
- Hash functions should take into account **all the bits of a key**, not just some of them.

Hash Functions

Hash function $h(k)$ as consisting of two actions:

- **Hash code**
  - Map the key $k$to an integer
- **Compression function**
  - Map the hash code to the range of indices 0 to $M - 1$

Hash Codes

- The first action that a hash function performs is to take an arbitrary key and map it into an integer value.
- This integer need not be in the range 0 to $M - 1$ and may even be negative, but we want the set of hash codes to **avoid collisions**.
- If the hash codes of our keys cause collisions, then there is no hope for the compression function to avoid them.

Hash Codes in C

- The hash codes described below are based on the assumption that the number of bits of each data type is known.

Converting to an Integer

- For any data type that is D represented using at most as many bits as our integer hash codes, we can simply **take an integer interpretation of the bits as a hash code** for elements of D.
- Thus, for the C basic types `char`, `short int` and `int`, we can achieve a good hash code simply by **casting** this type to `int`.

Converting to an Integer

- On many machines, the type `long int` has a bit representation that is twice as long as type `int`.
- One possible hash code for a long element is to simply cast it down to an int.
- But notice that this hash code **ignores half of the information** present in the original value. So if many of the keys differ only in these bits, they will **collide** using this simple hash code.
- A **better hash code**, which takes all the original bits into consideration, **sums** an integer representation of the high-order bits with an integer representation of the low-order bits.

Converting to an Integer

- In general, if we have **an object x whose binary representation can be viewed as a k-tuple** of integers $(x_0, x_1, \ldots, x_{k-1})$, we can form a hash code for as $\sum_{i=0}^{k-1} x_i$
- **Example**: Given any floating-point number, we can sum its mantissa and exponent as long integers and then apply a hash code for long integers to the result.

Summation Hash Codes

- The summation hash code, described above, is **not a good choice** for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{k-1})$ where the order of the x_i's is significant.
- **Example**: Consider a hash code for a string s that sums the ASCII values of the characters in s. This hash code produces lots of unwanted collisions for common groups of strings e.g., `temp01` and `temp10`.
- A better hash code should take the order of the x_i's into account.

Polynomial Hash Codes

- Let be an integer constant such that $a \neq 1$
- We can use the polynomial

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

  as a hash code for $(x_0, x_1, \ldots, x_{k-1})$.
- This is called a **polynomial hash code**.
- To evaluate the polynomial we should use the efficient **Horner's method**:

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots + a(x_1 + a x_o)) \ldots))$$

Polynomial Hash Codes

- Experiments show that in a list of over 50,000 English words, if we choose $a = 33,\ 37,\ 39,\ 41$ we produce **less than seven collisions** in each case.
- For the sake of speed, we can apply the hash code to only a fraction of the characters in a long string.

Polynomial Hash Codes

```
// dbj2 hash function

uint hash_string(Pointer value) {
  uint hash = 5381;

  for (char* s = value; *s != '\0'; s++)
        hash = (hash * 33)+ *s;

  return hash;
}
```

Polynomial Hash Codes

- In theory, we first compute a polynomial hash code and then apply the compression function *modulo M*
- The previous hash function takes the modulo M **at each step.**
- The two approaches are the same because the following equality holds for all a, b, x, M that are nonnegative integers:

$$(((ax) \mod M) + b) \mod M = (ax + b) \mod M$$

- The approach of the previous function is preferable because, otherwise, **we get errors with long strings** when the polynomial computation produces overflows (try it!).

Hashing Floating Point Quantities

- We can achieve a better hashing function for floating point numbers than casting them down to `int` as follows.
- Assuming that a `char` is stored as an 8-bit byte, we could **interpret a 32-bit float as a four-element character array** and use the hashing functions we discussed for strings.

Some Applications of Hash Tables

- Databases
- Symbol tables in compilers
- Browser caches
- Peer-to-peer systems and torrents (distributed hash tables)

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Chapter 11
- M.T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition. Chapter 9
- R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος. Κεφάλαιο 14