

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 11 – Hashing

(Week 13, starting 11 April 2016)

1. Hash Functions

A good hash function is essential for good hash table performance. A good hash function is easy/efficient to compute and will evenly distribute the possible keys. Comment on the flaw (if any) of the following hash functions. Assume the load factor α = number of keys / table size = 0.7 for all the following cases:

- (a) The hash table has size 100. The keys are positive even integers. The hash function is
$$h(\text{key}) = \text{key} \% 100$$
- (b) The hash table has size 49. The keys are positive integers. The hash function is
$$h(\text{key}) = (\text{key} * 7) \% 49$$
- (c) The hash table has size 100. The keys are non-negative integers in the range of [0, 10000]. The hash function is
$$h(\text{key}) = \lfloor \sqrt{\text{key}} \rfloor \% 100$$
- (d) The hash table has size 13. The keys are integers in the range of [0, 100]. The hash function is
$$h(\text{key}) = \text{key} \% 13$$
Collision resolution uses double hashing with secondary hash function
$$h_2(\text{key}) = \text{key} \% 7$$
- (e) The hash table has size 1009. The keys are valid email addresses. The hash function is
$$h(\text{key}) = (\text{sum of ASCII values of each of the last 10 characters}) \% 1009$$
See <http://www.asciitable.com> for ASCII values
- (f) The hash table has size 101. The keys are integers in the range of [0, 1000]. The hash function is
$$h(\text{key}) = \lfloor \text{key} * \text{random} \rfloor \% 101, \text{ where } 0.0 \leq \text{random} \leq 1.0$$

Answer

- (a) No key will be hashed directly to **odd-numbered slots** in the table, resulting in wasted space, and a higher number of collisions than necessary.
Aside from the hash function, the hash table size may not be good as it is not a prime number. If there are many keys that have identical last two digits, then they will all be hashed to the same slot, resulting in many collisions.
- (b) All keys will be hashed only into slots 0, 7, 14, 21, 28, 35 and 42. Also, as in (a), the hash table size is not a prime number.
- (c) Keys are **not uniformly** distributed. Note that $h(1 \leq \text{key} \leq 3) = 1$, i.e. 3 keys are hashed to slot 1; $h(4 \leq \text{key} \leq 8) = 2$, i.e. 5 keys are hashed to slot 2; $h(9 \leq \text{key} \leq 15) = 3$, i.e. 7 keys are hashed to slot 3, and so on. The difference between successive squares (e.g. 4, 9, 16, 25) increases as the numbers get larger, leading to **many more keys being mapped to the larger-indexed slots**. Also, as in (a), the hash table size is not a prime number.
- (d) The **output of the secondary hash function equals 0** when the input is a multiple of 7. For example, insert 7 followed by 98 ($98 = 13 * 7 + 7$). 7 will be inserted into the slot at index 7. The home address of 98 will also be 7, a collision. However, the output of the secondary hash function will be 0 ($98 \% 7 = 0$). This will create an **infinite loop while resolving the collision**.

- (e) Keys are **not evenly distributed** because many email addresses have the **same domain names**, e.g. "hotmail.com", "gmail.com". Many email addresses will be hashed to the same slot, resulting in many collisions.
- (f) This hash function is **not deterministic**. The hash function does not work because, while using a **given key** to retrieve an element after inserting it into the hash table, we **cannot always reproduce the same address**.

2. Hashing and Collision Resolution

Consider a hash table with size $m = 13$. First, the following 11 keys are **inserted** into the table in the following sequence:

10 100 32 45 58 126 3 29 200 400 0

Next, the following 4 operations are performed in sequence, assuming that **lazy deletion** is used:

Delete(45), Delete(4), Find(58) and Find(45)

- (a) Show the contents of the hash table after each insert, delete and find operation. Use a simple hash function $h(\text{key}) = \text{key} \% m$, and **linear probing** to resolve collisions.
- (b) Repeat (a) by using the following hash function and **quadratic probing** to resolve collisions.

```
int hash(int key, int m) {
    int sum = digitSum(key); //digitSum adds all digits of key,
    return sum % m;          //e.g. key = 126, sum = 1+2+6 = 9.
}
```

Answer

- (a) Insertion of the 11 keys:

key	Hash Table Slot												
	0	1	2	3	4	5	6	7	8	9	10	11	12
10											10		
100										100	10		
32							32			100	10		
45							32	45		100	10		
58							32	45	58	100	10		
126							32	45	58	100	10	126	
3				3			32	45	58	100	10	126	
29				3	29		32	45	58	100	10	126	
200				3	29	200	32	45	58	100	10	126	
400				3	29	200	32	45	58	100	10	126	400
0	0			3	29	200	32	45	58	100	10	126	400

<u>key</u>	<u>Home Addr</u>	<u>Probe Sequence</u>	<u>Number of Probes</u>
10	10	10	1
100	9	9	1
32	6	6	1
45	6	6, 7	2
58	6	6, 7, 8	3
126	9	9, 10, 11	3
3	3	3	1
29	3	3, 4	2
200	5	5	1
400	10	10, 11, 12	3
0	0	0	1

Average number of probes = $19 / 11 = 1.73$

Delete(45)

<u>key</u>	<u>Home Addr</u>	<u>Probe Sequence</u>	<u>Number of Probes</u>
45	6	6, 7	2

With lazy deletion, the deleted element is marked, without affecting the other elements.

Hash Table Slot												
0	1	2	3	4	5	6	7	8	9	10	11	12
0			3	29	200	32	45	58	100	10	126	400

Delete(4)

The key 4 is **not found** in the hash table, so the hash table remains the same.

<u>key</u>	<u>Home Addr</u>	<u>Probe Sequence</u>	<u>Number of Probes</u>
4	4	4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 1	11

We keep probing for 4 till we **hit a truly empty slot**.

Find(58)

Eventually found.

<u>key</u>	<u>Home Addr</u>	<u>Probe Sequence</u>	<u>Number of Probes</u>
58	6	6, 7, 8	3

Although the key 45 is logically “deleted”, we have to **continue probing past** it.

Find(45)

The key 45 is not found in the hashtable.

<u>key</u>	<u>Home Addr</u>	<u>Probe Sequence</u>	<u>Number of Probes</u>
45	6	6, 7, 8, 9, 10, 11, 12, 0, 1	9

Again, we have to **continue probing past** the lazily deleted element. We do not even examine its key.

(b) Insertion of the 11 keys:

key	Hash Table Slot												
	0	1	2	3	4	5	6	7	8	9	10	11	12
10		10											
100		10	100										
32		10	100			32							
45		10	100			32				45			
58	58	10	100			32				45			
126	58	10	100			32				45	126		
3	58	10	100	3		32				45	126		
29	58	10	100	3		32				45	126	29	
200	58	10	100	3		32	200			45	126	29	
400	58	10	100	3	400	32	200			45	126	29	
0	58	10	100	3	400	32	200			45	126	29	0

key	Home Addr	Probe Sequence	Number of Probes
10	1	1	1
100	1	1, 2	2
32	5	5	1
45	9	9	1
58	0	0	1
126	9	9, 10	2
3	3	3	1
29	11	11	1
200	2	2, 3, 6	3
400	4	4	1
0	0	0, 1, 4, 9, 3, 12	6

Average number of probes = $20 / 11 = 1.82$

Delete(45)

key	Home Addr	Probe Sequence	Number of Probes
45	9	9	1

Again, with lazy deletion, the deleted element is marked, without affecting the other elements.

Hash Table Slot												
0	1	2	3	4	5	6	7	8	9	10	11	12
58	10	100	3	400	32	200			45	126	29	0

Delete(4)

The key 4 is **not found** in the hash table, so the hash table remains the same.

key	Home Addr	Probe Sequence	Number of Probes
4	4	4, 5, 8	3

Remember, as we are calculating the next index to probe, add the next square **starting** from the **home address**!

Find(58)

Found.

key	Home Addr	Probe Sequence	Number of Probes
58	0	0	1

Find(45)

The key 45 is not found in the hash table.

key	Home Addr	Probe Sequence	Number of Probes
45	9	9, 10, 0, 5, 12, 8	6

Remember to **continue probing past** the lazily deleted element.

3. Binary Search on Sorted Array vs Lookup in Hash Table

An array named `values` contains 511 distinct integers in ascending order. The following code is used to determine if an integer `key` exists within `values`:

```
int leftIndex = 0;
int rightIndex = values.length - 1;
while (leftIndex <= rightIndex) {
    int middleIndex = (leftIndex + rightIndex) / 2;
    if (values[middleIndex] == key) {
        return true;
    } else if (values[middleIndex] < key) {
        leftIndex = middleIndex + 1;
    } else {
        rightIndex = middleIndex - 1;
    }
}
return false;
```

Restriction for all parts: Search for each `key` independently. Do NOT reuse the result from one search in another. Do NOT search for two keys at once.

- (a) Compute the total number of **key comparisons** necessary to locate all 511 values in the array using the above loop. A key comparison occurs whenever we evaluate one of:
- `if (values[middleIndex] == key) { ...`
 - `} else if (values[middleIndex] < key) { ...`
- (b) Now suppose that the 511 distinct integers are already stored in a hash table with size p , where p is a prime. The hash function is $h(key) = key \% p$. We use **separate chaining** to resolve collisions. We assume that all the 511 integers are uniformly distributed, i.e. there are either $\left\lceil \frac{511}{p} \right\rceil$ or $\left\lfloor \frac{511}{p} \right\rfloor$ nodes in each chain.

Determine the minimal p , i.e. the smallest possible hash table, such that the total number of key comparisons necessary to access all 511 values in this hash table is smaller than or equal to the result in (a). Keep in mind that even though the first node in the chain is the value we are searching for, we still need a key comparison. Checking whether there is a node after the current does NOT count as a key comparison.

- (c) One of the problems with *separate chaining* is that long chains slow down the search, because we can only compare keys sequentially. One suggestion is: instead of using linked list as a **chain**, use a **sorted array**.

Each slot in the hash table contains a reference to a sorted array. The arrays may have different sizes. When finding whether a key exists in the hash table, we can use the hash function and table size p in (b), but resolve collisions by inserting the colliding key into the appropriate sorted array.

As all arrays are sorted, we can now apply binary search to improve the efficiency. Suppose the 511 integers are already stored in the new hash table, does this improve searching over (a) and (b)? Also, is this design good, and why or why not?

Answer

(a)

During binary search:

Searching for `values[255]`, the middle element, requires 1 key comparison.

Searching for `values[127]` or `values[383]`, the middle elements of the first 255 and last 255 elements respectively, requires 3 key comparisons.

Four keys in values require 5 key comparisons... and so on.

Therefore, the total number of comparisons to find all 511 keys is:

$$1 \times 1 + 2 \times 3 + 4 \times 5 + 8 \times 7 + 16 \times 9 + 32 \times 11 + 64 \times 13 + 128 \times 15 + 256 \times 17 = \mathbf{7683}$$

(b)

Let c = length of each chain in the hash table. Since the 511 integers are uniformly distributed:

$$c \approx \frac{511}{p} \quad \dots\dots\dots(1)$$

After finding the home address using the hash function, we have to traverse the list to find `key`. In each chain:

The first integer requires 1 key comparison,

The second integer requires 2 key comparisons, ...

The last integer in the chain requires c key comparisons.

Hence, the total number of key comparisons for all elements in **ONE** chain is:

$$1 + 2 + 3 + \dots + c = \frac{c(c+1)}{2}$$

There are a total of p **chains** in the hash table, and we want to find p such that the total number of key comparisons is less than 7683:

$$\frac{pc(c+1)}{2} < 7683$$

As we have already computed the value of c in equation (1), we have

$$p > 17.58$$

Therefore, the smallest prime to be chosen as the hash table size is $p = \mathbf{19}$.

(c)

Yes, searching is improved because on average, and in the worst case, fewer key comparisons are needed. To find one element, $O(\log c)$ key comparisons are needed, instead of $O(c)$ key comparisons.

In this example, we know the total size = 511, and the length of each chain $[c]$ or $[c]$, beforehand. Therefore, we can allocate all the p arrays without wasting any space.

However, in general, this is not a good design. Two main drawbacks are listed:

Maintaining sorted order slows insertion

In this case, the keys are already in the hash table. In reality, data is usually added, removed, and searched. Maintaining sorted order for every newly added entry is time-consuming. With a sorted array, we need $O(c)$ time to insert a new entry, as opposed to $O(1)$ time using the typical separate chaining with linked list.

Using (static-sized) array with unknown input size

In this case, we know how long each chain (array) should be. We will not waste any memory space allocating the arrays. In reality, we usually cannot predict the capacity of the hash table. If this happens, then we need to either allocate a much longer array and waste space, or allocate a smaller array and waste time reallocating memory for a new array, copying the contents over, when the array is full.

Note: Question 4 will not be discussed during tutorial. You may discuss it on the IVLE forum.

4. Finding Combinations

Given an **odd** integer k , and a list of n people, each having a name and amount of money, **find all pairs** of people who have $\$k$ in total. For each pair found, print their (name, amount). You need not print the other permutation of the same pair of people. Every person has a unique name.

Example

Given $k = 9$ and the following list of 5 people:

(Alice, 4), (Bob, 3), (Charlie, 7), (Kenneth, 5), (Victor, 2)

The output should be:

(Alice, Kenneth)
(Charlie, Victor)

Restriction: Use a `java.util.HashMap` to solve the problem. You may also use other data structures along with the `HashMap` if really necessary.

```
class Person {
    private String _name;
    private int _amount;
    public Person(String name, int amount) {
        _name = name; _amount = amount;
    }
    public String getName() { return _name; }
    public int getAmount() { return _amount; }
}

public void findAllPairs(List<Person> people, int targetAmount) {}
```

- (a) Each person has a unique amount of money. Design an $O(n)$ algorithm to solve the problem.
- (b) More than one person may have the same amount of money. Analyze the time complexity of your solution.

Answer

(a)

As there are no duplicate amounts, for every person, there will be either 0 or 1 other person to pair with. `java.util.HashMap` maps a unique key to a value, which does not need to be unique. Using a person's amount as key and name as value, we can find another person with the corresponding amount in $O(1)$ average-case time.

For each person with V amount of money, we find whether there is another person with amount $(k - V)$ in the `HashMap`. If such a person exists, we want the person's name. Iterating through the list of n persons, we use the `get()` and `put()` methods, which each costing $O(1)$ average time. Therefore, the algorithm runs in $O(n)$ time.

Current Person	HashMap Key: Value	HashMap get()	Print Pair of Persons
(Alice, 4)		5 (because $k - 4 = 5$) Not found	
(Bob, 3)	4: Alice;	6 (because $k - 3 = 6$) Not found	
(Charlie, 7)	4: Alice; 3: Bob;	2. Not found	
(Kenneth, 5)	4: Alice; 3: Bob; 7: Charlie;	4. Found Alice	(Alice, Kenneth)
(Victor, 2)	4: Alice; 3: Bob; 7: Charlie; 5: Kenneth;	7. Found Charlie	(Charlie, Victor)

```
public void findAllPairs(List<Person> people, int targetAmount) {
    HashMap<Integer, String> map = new HashMap<Integer, String>();
    for (Person currPerson : people) {
        int currAmt = currPerson.getAmount();
        String currName = currPerson.getName();
        String otherName = map.get(targetAmount - currAmt);
        if (otherName != null)
            System.out.println(
                "(" + otherName + ", " + currName + ")");
        map.put(currAmt, currName);
    }
}
```

(b)

For each person, we still want to search for another person with the corresponding amount of money in $O(1)$ average-case time. However, many such people may exist. Imagine if there is a (James, 4) in the above list, Kenneth should form pairs with two people instead of just one.

To avoid overwriting Alice in the HashMap, we need the value to be a Collection of names instead. This is similar to resolving collisions using the separate chaining technique. We can now store multiple names that have the same amount of money.

```
public void findAllPairs(List<Person> people, int targetAmount) {
    HashMap<Integer, LinkedList<String>> map =
        new HashMap<Integer, LinkedList<String>>();
    for (Person currPerson : people) { // build HashMap and chains
        int currAmt = currPerson.getAmount();
        String currName = currPerson.getName();
        LinkedList<String> names = map.get(currAmt);
        if (names == null) {
            names = new LinkedList<String>();
            map.put(currAmt, names);
        }
        names.addLast(currName);
    }
    for (Person currPerson : people) { // find matches to curr person
        int currAmt = currPerson.getAmount();
        String currName = currPerson.getName();
        LinkedList<String> names = map.get(targetAmount - currAmt);
        if (names != null)
            for (String otherName : names)
                System.out.println(
                    "(" + currName + ", " + otherName + ")");
        map.get(currAmt).remove(currName); // eliminate symmetry
    }
}
```

Picture the scenario where $\frac{n}{2}$ persons have X amount of money, the other $\frac{n}{2}$ persons have Y amount of money, and $X + Y = k$. In this worst case, there are a total of $\frac{n}{2} \times \frac{n}{2}$ pairs of people, with each list being traversed sequentially. This means the time complexity would be $O(n^2)$.

