Universitetet i Oslo
Institutt for Informatikk
A. Maus, R.K. Runde, I. Yu

INF2220: algorithms and data structures

Series 3

Topic Map and Hashing (Exercises with hints for solution)

Issued: 7. 09. 2016

Classroom

Exercise 1 (Hash table complexity) What is the complexity of finding order information, such as max, min, or range of stored values from a hash table?

Solution: [of Exercise 1] Hash tables are not really made for that. In particular, they don't contain or represent *order* information. In a way, a good *hashing function* should best be completely unordered ....

Anyway, since that's the case, there's no other way to determine the maximum etc than just going throught the whole table and find out. So the complexity is $O(n)$. Note there is no distinction between best/worst/average case, one needs to seach through all of it, as said.

Exercise 2 (Extendible hashing) Assume that you have an empty hashtable, and that $M = 4$. Show the hash table you get by inserting the following numbers: {000100, 001000, 100000, 011011, 111000, 010100, 101010, 010101, 111001, 001010, 000111, 001001, 101110, 100100}

Solution: [of Exercise 2] See figure.ex

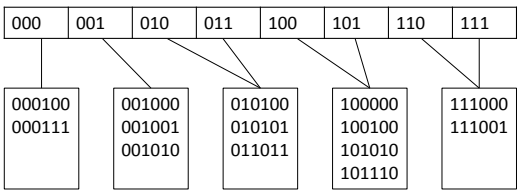

Figure 1: solution of 2

Exercise 3 (Deletion from a hash table) Explain how deletion is performed both with probing and separate chaining hash-tables.

Exercise 4 (Hash table behavior) Given the input {`42,39,57,3,18,5,67,13,70,26`}, and a fixed table size of 13, and a hash-function `H(X) = X mod 13`, show the resulting

1. Linear probing hash-table
2. Separate chaining hash-table

Solution: [of Exercise 4] One may mention that exercises like that are not uncommon in earlier exams. Not just for hash-tables, of course. Note: the solution of last year had an error (13 and 26 should have swapped place).

1. *linear probing:* That one is pretty simple: just calculate mod 13, if the slot is filled ("conflict"), move (typically) to the right until you find a free slot. Note that it's convenient (as is the case in Java) for those modulo-based hashing functions, that the first slot of the array is index by 0.[1] The hash table is rather small, but perhaps one get's a feeling what the problem of *primary clustering* is. One can compare that with the figure below (with external chaining): for instance the linked list at slot 0 is of length 3 and hence "grows" in the cluster starting at 2 etc. such that in the end it's one big cluster. As a hint for an exam: if the task is (like here) to make a separate chain and an internal probing hash table *with the same data,* it seems slightly faster an less error prone to do the separate chain first. Of course: one cannot first do the separate chains and *only* look at the result to construct the other one .... The order of the original insertions plays a role (which is not completely preserved in the separate chain HT).

elements:	39	13	67	42	3	57	18	5	70	26			
i:	0	1	2	3	4	5	6	7	8	9	10	11	12

2. *Separate chaing hash table:*

   The solution here is slightly inefficient. As u can see from the linked list, when inserting an element, it's done *at the end.* There is no reason to do that. It's to be expected that it's slightly more efficient to insert it at the head, avoiding list traversal. In practice there is perhaps not such a big difference. If one has a good hashing function (and consequently not too heavy clustering) the linked lists should be reasonably short. If they get too long one would increase the size of the array. But still, no need to insert at the end. In the book [Weiss, p. 198, Fig 5.10] the function `List.add` method (from the Java lib) is used, where the documentation states

   > "Appends the specified element to the end of this list (optional operation)."

   It's to be expected that the Java implementation of linked list is more subtle that a naive (single-)linked list, so the above remark may not be too relevant. If, however, one makes the linked list from scratch, adding it at the beginning is more appropriate. Note: remembering "esoteric" details from the Java library such as "remember that the add function adds to the end" as far as Java is concerned is not exam material. Knowing that adding to a linked list at the beginning or the end can make a difference (depending on how it's represented) is something one should be aware of.

[1] Some older/weired languages may start with 1, in which case one has to "shift" the mod-function by one.

i:	0	1	2	3	4	5	6	7	8	9	10	11	12
List of elements:	↓ 39 ↓ 13 ↓ 26		↓ 67	↓ 42 ↓ 3		↓ 57 ↓ 18 ↓ 5 ↓ 70							

Exercise 5 (Hash table behavior) Given the input {`4371,1323,6173,4199,4344,9679,1989`}, and a fixed table size of 10, and a hash-function `H(X) = X mod 10`, show the resulting

1. Linear probing hash-table
2. Quadratic probing hash-table
3. Separate chaining hash-table

Solution: [of Exercise 5]

1. Linear probing hash-table

elements:	9679	4371	1989	1323	6173	4344				4199
i:	0	1	2	3	4	5	6	7	8	9

2. Quadratic probing hash-table.

elements:	9679	4371		1323	6173	4344			1989	4199
i:	0	1	2	3	4	5	6	7	8	9

3. Separate chaining hash-table: that one is of course almost too boring.

i:	0	1	2	3	4	5	6	7	8	9
List of elements:		↓ 4371		↓ 1323 ↓ 6173	↓ 4344					↓ 4199 ↓ 9679 ↓ 1989

Exercise 6 (Hash table behavior) Given the input {`15,78,56,25,19,38,57,76,34,53,72,91`}, and a fixed table size of 19, and a hash-function `H(X) = X mod 19`,

1. show the resulting Quadratic probing hash-table
2. show the resulting Double probing hash-table. Note that you have to first find the *largest prime number* which is *smaller* than the size of the hash-table.

Solution:

1. show the resulting Quadratic probing hash-table

elements:	19	38	78		57	53	25			76			72	91		15	34		56
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

2. show the resulting double probing hash-table. Double probing is described at page 203.

elements:	19		78	53			25	34		76	91	57		38		15	72		56
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Exercise 7 (Complexity questions: Binary hash map) A class `BinaryHashMap` serves as a basis for the first 3 questions in this assignments. The internal array (of lists) which is used by `BinaryHashMap` is of length 2. Consequently, the (very basic) hash-function hashes all keys with an even length to 0 and all keys with an odd length to 1. The following complexity questions should be anwered with big-O notation, both for average case and worst case.

1. What is the complexity of locating an element in an unordered list?
2. What is the complexity of locating an element inside the BinaryHashMap?

   `binHash` is an object of BinaryHashMap.

   ```
   Object o = binHash.get(‘‘a_key’’)
   ```

   Hint: the elements are distributed among two unordered lists.
3. What is the complexity of inserting an element into the BinaryHashMap?

   ```
   binHash.put(‘‘some_key’’, some_obj_pointer);
   ```

   Hint: keys are unique, we cannot simply add it to one of our internal lists.[2]
4. Let M be the number of list-pointers internally inside a hash-table, assume that we have a hash-function which is perfect. I.e., if we fill it with M key-value pairs, we have zero collisions and all our internal lists contain one elements each.

   What is the complexity of locating an element in a hash-table with a perfect hash-function, if it contains N elements, and it has M internal list pointers. I.e., *big-O* expressed with N and M.

Solution: [for Exercise 7]

1. finding elements in an unordered list:
   - worst case: $O(n) = n$. The worst case is that the element is the *last* one we inspect (equally if the element is *not* there; in that case we have to check them all as well until we are sure).

[2] It's not a "law of nature" that hash tables as concept work only with unique keys. However, the binary hash map of this exercise should be a degenerate version of Java's hash map (`java.util.HashMap<K,V>`. It's a hash table implementation for a "map", which a function assocuating values to keys (so a *mapping* from keys to values. Hence, keys must be unique. Note in passing: the `put` method from Java's hash map overrides the old value (if there's one) with the newly inserted and gives back the old value (if there has been one). This detail does not affect the complexity.

- average case: $O(n) = (n/2)/2 = n/4$. On average, one has to check half the list to find an element. According to the definitions we learnt, the factors $1/2$ are irrelevant in the asymptotic considerations underlying the "big-o" notations. So a correct answer is $O(n)$ as well, or "linear complexity". Similarly for the other questions.

2. finding an element in the given "binary" hash map

   - worst case: $O(n) = n$: the worst case is, that the hash table is completely "unbalanced" in the sense that all elements are chained to one slot, and furthermore that the element is at the end of that list.
   - average case: $O(n) = (n/2)/2 = n/4$. One has two lists with $n/2$ elements on average, and you have to check half the list on average to find an element.

3. Insertion

   (a) worst case $O(n) = n$

   (b) average case O(n) = n/4

   We first have to find the element as before, i.e. the same penalty as before. Updating the pointer to the new element can be done in constant time and therefore can be ignored.

4. The number of elements in internal lists are distributed perfectly as described is: N / M.

   The complexity of finding an element in a list with length N/M is

   (a) worst case: N/M

   (b) average case: $(N/M)/2 = N/2M$

Lab

Exercise 8 We are going to implement a few functions inside the class `BinaryHashMap`

1. Implement the function: `boolean remove(String key)`, which returns false if the elements is not present inside BinaryHashMap true otherwise.

2. Implement the function: `String[] keys()`, which returns all keys inside the `BinaryHashMap`.

   (HINT: we know how large this array has to be from `this.size`)

3. Implement the function: `Object[] toArray()`, which should return all values from the hash-table.

Exercise 9 Write an implementation for a hash table which uses *separate chaining* to handle hash collision. The implementation should include *inserting*, *deleting*, and *searching* an element in the hash table.