Assignment 6 – Selected Model Answers

EXERCISE 1. The waste of space in hashing with chaining is due to empty lists in some entries of the representing array A. For a random hash function determine the expected number of empty entries in the hash table as a function of the number m of possible hash values and the size n of the represented set.

SOLUTION. Let S be the set represented by A with the hash function $h : S \to \{0, \ldots, m-1\}$ and $|S| = n$.

For $0 \leq i \leq m-1$ we use a Bernoulli-distributed random variable X_i with

$$X_i = \begin{cases} 1 & \text{if } A[i] = [] \\ 0 & \text{else} \end{cases} \qquad \text{and let} \quad X = \sum_{i=0}^{m-1} X_i$$

be the random variable for the number of empty entries in the array A. We have $A[i] = []$ iff $h(e) \neq i$ holds for all $e \in S$. We have $P(h(e) = i) = 1/m$, as h is a random hash function. Then $P(h(e) \neq i) = 1 - \frac{1}{m} = \frac{m-1}{m}$. Furthermore, the values of $h(e)$ are independent of each other, so the probability that $h(e) \neq i$ holds for all $e \in S$ is $\left(\frac{m-1}{m}\right)^n$ independent from the value i.

With this we compute the expectation

$$E(X) = \sum_{i=0}^{m-1} E(X_i) = \sum_{i=0}^{m-1} P(A[i] = []) = m \left(\frac{m-1}{m}\right)^n .$$

EXERCISE 2.

(i) Assume you have a large file consisting of triples (transaction, price, customer ID). Explain how to compute the total payment due for each customer. Your algorithm should run in linear time.

(ii) Show how to scan a hash table for hashing with chaining and for hashing with linear probing, i.e. return all values stored in the table. What is the running time of your solutions?

SOLUTION.

(i) Let S be the set of triples in the file. We construct a hash table, in which entries are (pointers to) triples $(t, p, c) \in S$ with a unique transaction number t, a price p, and a customer ID c. Let $T > 0$ and $C > 0$ be constants, and assume two hash functions h_1 and h_2, where h_1 assigns a value in $h_1(t) \in \{0, \ldots, T-1\}$ to a transaction number, and h_2 assigns a value in $h_2(c) \in \{0, \ldots, C-1\}$ to a customer ID.

We define a new hash function $h(t, p, c) = T \cdot h_2(c) + h_1(t)$. We need to compute $P(c) = \sum_{(t,p,c) \in S} p$. Then all $(t, p, c) \in S$ can be found by scanning the hash table starting from

the entry at position $T \cdot h_2(c)$ and continuing at least to position $T \cdot h_2(c) + T - 1$. We use hashing with linear probing, so the scan terminates, when *undef* is found.

The maximum number of table entries examined is $\leq T + |S|$, so the complexity is in $O(|S|)$, i.e. the search requires time linear in the number of transactions.

(ii) As we do not know which hash values in $\{0, \ldots, m-1\}$ are used, all these values need to be explored. For hashing with linear probing it is appropriate to scan the representing array from position $A[0]$ to $A[m-1]$. If the entry is x different from *undef* or a placeholder, x is appended to a result list, which initially is empty. For hashing with chaining the lists ℓ_i stored in $A[i]$ are concatenated. As *append* and *concat* can be executed in constant time on doubly-linked lists, the time complexity in in $O(m)$ in both cases, where m is the size of the hash table.

EXERCISE 3. A simple variant of *linear hashing* exploits a fixed maximum length l of the lists stored in a table entry. For this fix a number m and a hash function h with values in $\{0, \ldots, m-1\}$. The hash table will have a size $M \geq m$. Then define inductively hash functions h_i with $h_0 = h$ and $h_{j+1}(e) = h_j(e) + m \cdot 2^j$, so h_j takes $2^j \cdot m$ different hash values. The hash value of an element e is $h_j(e)$, where j is maximal with $h_j(e) \leq M - 1$.

(i) Show that whenever an element e is given, it suffices to compute at most two hash values.

(ii) Define a *rehash* operation that comes into action, when the maximum list length l will be exceeded by insertion of a new element e. In this case use h_{j+1} to distribute the elements of the list into two sublists and increase M accordingly. If necessary, split other lists as well according to the new value M.

(iii) Show that if it is permitted to use also h_{j-1} instead of h_j to determine the hash value of an element e, then rehashing can be done in a lazy way by splitting only one list.

SOLUTION.

(i) As the size of the hash table is M, we take $L = \lfloor \log_2 M/m \rfloor$, which gives $2^L \cdot m \leq M < 2^{L+1} \cdot m$. For given e we therefore compute first $h_{L+1}(e)$. If this is $< M$, the hash value of e is $h_{L+1}(e)$, otherwise it is $h_L(e)$.

(ii) If the list ℓ stored in the hash table in position $h_j(e)$ is to be split due to an overflow, we built the list ℓ_1 containing all $e' \in \ell$ with $h_{j+1}(e') = h_j(e')$ as well as the list ℓ_2 containing all $e' \in \ell$ with $h_{j+1}(e') = h_j(e') + 2^j \cdot m$. ℓ_1 is the new list stored in position $h_j(e)$, whereas ℓ_2 is the new list stored in position $h_{j+1}(e)$.

Accordingly M is increased to $M' = h_{j+1}(e)$. For positions $k \neq h_j(e)$ it becomes possible that the hash value $h(f) = k$ (which is $\leq M - 1$ before the split) satisfies $k < M'$. In this case the list stored in position k is also split.

(iii) If the cascaded splitting in (ii) is not done, then a list in position $h_j(e)$ may contain values e', where $h_j(e') = h_j(e)$, but $h_{j+1}(e') < M$, i.e. the maximality of the index j is violated. Such a value e' cannot be found, if we only use a hash function h_j for which j is maximal with $h_j(e') < M$. However, if also $h_{j-1}(e')$ is computed, then e' can be found in the list in position $h_{j-1}(e')$.

EXERCISE 4.

(i) An alternative possibility is to reorganise the hash table during insertions, known as *Robin Hood hashing* is to check for two elements competing for the same position, how far away they are stored from the ideal position given by their hash value. Then the chaining search is applied to the element closer to this ideal position. Implement this method and compare the performance with the performance of the methods on the HASHSET class.

(ii) A *map* M is a partial function that is defined on a set K, i.e. we can write M as a set of *key-value pairs* (k, v) with unique keys $k \in K$. *Memoisation* is a programming technique useful for cases, where the same call of a function is used multiple times. In order to avoid costly recomputation it uses a map to store results from previous calls.

(a) Define a class HASHMAP in analogy to HASHSET to represent maps. As keys uniquely determine key-value pairs let the hash values only depend on the keys of such pairs.

(b) Use your class HASHMAP for memoisation on an example of your choice—you could use a naive algorithm for computing Fibonacci numbers, the towers of Hanoi problem, or just the location of elements on a given position in a doubly-linked list.

SOLUTION. See the C++ header and program files in the archives `Ass6_Ex4isolution.zip` and `Ass6_Ex4iisolution.zip`.