Technische Universität München
Fakultät für Informatik
Lehrstuhl für Algorithmen und Komplexität
Prof. Dr. Harald Räcke
Richard Stotz

Winter term 2019/20
Assignment 09
December 9, 2019

Efficient Algorithms and Data Structures I

*Deadline: December 16, 10:15 am in the **Efficient Algorithms** mailbox.*

Homework 1 (5 Points)

Santa Claus asks you to insert the following values in a Cuckoo-Hashing hashtable. The first hash function is

$$h_1(x) = (4x + 7 \bmod 17) \bmod 9 \ ,$$

the second hash function is

$$h_2(x) = (3x + 4 \bmod 19) \bmod 9 \ .$$

Both hash tables use size 9, `maxsteps` is set to 5. Initially, the hash table looks as follows:

	0	1	2	3	4	5	6	7	8
T_1	9	7	18				19		

	0	1	2	3	4	5	6	7	8
T_2					3		10		

Santa wants to see how the hash table looks like after each insert. Always carry out each operation on the result of the previous operation.

(a) Insert 17

(b) Insert 11

(c) Insert 4

Note: For this exercise, you do not need to show any intermediate steps.

Homework 2 (5 Points)

A huge array T contains garbage data and should be used as a *dictionary* of pointers. The dictionary must support the following operations

- Insert(p, k): store data $p \neq 0$ with key k,
- Search(k): return the data associated with key $k \in [|T|]$ and ERROR if no pointer with that key is in the data structure,

- Delete(k): remove the data associated with key k.

Ideally, we would wipe all entries of T initially by setting them to 0; then Insert(p, k) could simply store p at position $T[k]$. This, however, is very expensive because T is huge.
Describe a scheme for implementing a direct-address dictionary on T.
Each stored object should use $\mathcal{O}(1)$ space; the operations SEARCH, INSERT, and DELETE should take $\mathcal{O}(1)$ time each; and initializing the data structure should take $\mathcal{O}(1)$ time.
Example: When accessing $T[i]$, your scheme must detect whether the content of $T[i]$ is garbage or actual data. If $T[i]$ is valid, your scheme must provide a way to access – in constant time – the pointer associated with $T[i]$.
Hint: You may use a constant number of additional arrays. These arrays can be treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.

Homework 3 (6 Points)

Consider a system in which we have two different hashing functions, i.e. each key is mapped to one of its two valid positions.
Prove the following statement:
It is possible to distribute any n keys without collision **if and only if** there is no set S of keys with $|S| \leq n$ so that the keys in S have at most $|S| - 1$ alternative positions in the two hash tables.
Note that each key has exactly two positions, one for each hash table. You may assume that no two keys have both positions identical.
Example: If key k_1 has position 1 in table T_1 and position 4 in table T_2, while key k_2 has positions 6 and 4 in T_1 and T_2, respectively, then k_1 and k_2 have 3 alternative positions.

Homework 4 (5 Points)

Consider classes of hash functions $\mathcal{H}$ over a finite universe U into $\{0, 1, \ldots, n-1\}$ with $|U|, n > 1$.
Professor Räcke claimed in the lecture that if $|U| > n$, then for any class $\mathcal{H}$, at least one pair $u_1, u_2 \in U$ of distinct elements has

$$\Pr[h(u_1) = h(u_2)] \geq \frac{1}{n} \ .$$

Here, the probability is over the choice of the hash function h drawn uniformly at random from the family $\mathcal{H}$.

1. Prove Prof. Räcke wrong by providing a counterexample to his claim, i.e., a class of hash functions $\mathcal{H}$ violating the claim.

2. Correct the statement by showing that for any family of hash functions

$$\Pr[h(u_1) = h(u_2)] \geq \frac{1}{n} - \frac{1}{|U|} \ .$$

Tutorial Exercise 1

Consider a binary heap H implemented with a binary tree data structure (as implemented in the lectures) containing n items. Design an algorithm to find the k-th smallest item in H in $O(k \log k)$ time.

Tutorial Exercise 2

A *soft-heap* is a priority queue that performs insert and delete-min in $\mathcal{O}(\log 1/\varepsilon)$ steps for any $0 < \varepsilon < 1/2$. This is achieved at the expense of "corrupting" keys, i.e. increasing them: At any time, at most εn keys in the heap are corrupted, where n is the total number of elements ever inserted into the heap. After any operation, the soft heap returns a list of newly corrupted keys.

1. We want to select the kth smallest element in a list of n elements. Let $\varepsilon = 1/3$. Use a soft heap to find an element whose rank is between $n/3$ and $2n/3$ in linear time. Apply this procedure recursively to find the kth smallest element in linear time.

   Hint: Quick, select an element near the median!

2. Using the result from Homework 2 and Part 1, show that an element of rank k in a binary heap can be returned in $\mathcal{O}(k)$ steps.

> Minimum Spanning Tree is the textbook example of a matroid optimization problem for which, as prevailing wisdom goes, "greed is good." Nonsense. This misguided view held back progress for years; that is, until KKT came along and showed greed for what it was: rubbish. My algorithm reaffirmed that finding.
>
> - B. Chazelle