Dynamic Perfect Hashing: Upper and Lower Bounds

Martin Dietzfelbinger*
Universität–GH–Paderborn
Fachbereich 17
4790 Paderborn, F. R. G.

Anna Karlin†
DEC Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301

Kurt Mehlhorn‡
Max-Planck-Institut
für Informatik
6600 Saarbrücken, F. R. G.

Friedhelm Meyer auf der Heide*
Universität–GH–Paderborn
Fachbereich 17
4790 Paderborn, F. R. G.

Hans Rohnert‡
Siemens AG
8000 München 83, F. R. G.

Robert E. Tarjan†
Princeton University
Dept. of Computer Science
Princeton, NJ 08544
and NEC Research Institute

Revised Version, January 7, 1990
`Final version will appear in SIAM J. Computing`

Abstract

The dynamic dictionary problem is considered: provide an algorithm for storing a dynamic set, allowing the operations insert, delete, and lookup. A dynamic perfect hashing strategy is given: a *randomized* algorithm for the dynamic dictionary problem that takes $O(1)$ worst-case time for lookups and $O(1)$ amortized expected time for insertions and deletions; it uses space proportional to the size of the set stored. Furthermore, lower bounds for the time complexity of a class of *deterministic* algorithms for the dictionary problem are proved. This class encompasses realistic hashing-based schemes that use linear space. Such algorithms have amortized worst-case time complexity $\Omega(\log n)$ for a sequence of n

*partially supported by DFG Grant Me 872/1-4.

†Research at Princeton University partially supported by NSF grants DCR-8605962 and STC88-09648 and ONR Contract N00014-87-K-0467.

‡partially supported by DFG grant Me 620/6-1 and ESPRIT-project ALCOM. K. Mehlhorn and H. Rohnert were affiliated with the Universität des Saarlandes when this research was done.

insertions and lookups; if the worst-case lookup time is restricted to k then the lower bound becomes $\Omega(k \cdot n^{1/k})$.

Key words. data structures, dictionary problem, hashing, universal hashing, randomized algorithm, lower bound.

AMS(MOS) subject classifications. 68P05, 68P10, 68Q20.

1 Introduction

A *dictionary* over a *universe* $U = \{0, 1, \ldots, N-1\}$ is a partial function S from U to some set I. The operations *Lookup*(x), *Insert*(x, i), and *Delete*(x) are available on a dictionary S; *Lookup*(x) returns $S(x)$, *Insert*(x, i) adds x to the domain of S and sets $S(x)$ to i, and *Delete*(x) removes x from the domain of S. In the following, the "information field" $S(x)$ associated with the "key" x in the dictionary will be ignored; thus, S is identified with its domain and regarded as a (dynamic) set. There are two major techniques for implementing dictionaries: trees and hashing.

For a static set S (no updates), Fredman, Komlós, and Szemerédi [FKS84] described a hashing technique that achieves linear storage (in n) and constant query time for all N and n, where n is the size of S.

In this paper (Section 2), we present an extension of their scheme to the dynamic situation, wherein membership queries are processed in constant worst-case time, insertions and deletions are processed in constant expected amortized time, and the storage used at any time is proportional to the number of elements currently stored in the dictionary. The algorithm is randomized; the averaging involved in the analysis is over choices made by the algorithm and not over the sequence of operations.

Besides solutions that use (balanced) search trees, several other approaches to the dynamic dictionary problem have been proposed, some of which lead to expected or average constant time per instruction. Aho and Lee [AL86] presented a scheme achieving the same time and storage bounds as our algorithm. However, in order to prove these bounds, they require that the items being inserted are chosen uniformly at random from the universe of possible elements.

Carter and Wegman [CW79] proposed *universal hashing* as a way of avoiding assumptions on the distribution of input values. This approach works particularly well in combination with the idea of "continuous rehashing" introduced by Brassard and Kannan [BK88]. In this way an algorithm is obtained that needs linear space and expected constant time for each single instruction. However, for n keys being stored in the dictionary in a scheme of this kind the best upper bound known on the expected *worst* *case time* for an instruction (i. e., the length of the longest chain in the resulting hash table with chaining) is $O(\log n / \log\log n)$ (cf. [DM90b, S89]), and it can be argued that it is $\Omega(\log n / \log\log n)$ no matter what universal class is used. In fact, this lower bound even holds in the case of uniform hashing, where one assumes that the hash values for different keys are chosen uniformly at random [G81, MV84].

In contrast, our algorithm guarantees constant time for each membership query.

When we say that no assumption is made about the sequence of operations, we mean that the sequence is arbitrary, but fixed before the algorithm starts running. In essence, all that is needed for the analysis is that the sequence of operations be independent of the random choices made by the algorithm. Thus, we require that the party that chooses the sequence of operations not use any knowledge on these random choices to determine which items to insert in the table.

In the second part of the paper (Sections 4 and 5), we consider the case that we have to deal with an adversary that knows the random choices made by the algorithm, or equivalently, that the algorithm is deterministic. We prove an $\Omega(\log n)$ lower bound on the amortized worst-case time complexity for any deterministic solution to the dictionary problem which is solely based on hashing and uses only linear space. Furthermore, if we assume the worst-case lookup time to be bounded by k, the amortized worst-case complexity is $\Omega(k \cdot n^{1/k})$.

Remark 1.1 Some of the lower bounds that hold for the model considered in Sections 4 and 5 are bigger than the $O(\log n)$ worst-case bound guaranteed by balanced search trees. This results from the fact that our model is defined so as to cover only pure hashing strategies. In [MNR90], which was motivated by the first version of the present paper, a lower bound of $\Omega(n \log\log n)$ for n insertions is shown in a stronger lower bound model that encompasses both hashing strategies and search trees.

In Section 3, some general facts concerning the performance of universal classes of hash functions consisting of polynomials of constant degree or variants thereof are established. These results have proved useful for variations of the scheme presented in this paper, which yield constructions of dynamic dictionaries for parallel and distributed machine models as well as further improvements of the sequential scheme [DM89, DM90a, DM90b].

2 Dynamic perfect hashing

We begin by reviewing the FKS scheme for statically storing a set S of size n. Let $\mathcal{H}_s = \{h : U \to \{1, \ldots, s\} \mid h(x) = (kx \bmod p) \bmod s,\ 1 \le k \le p-1\}$, where p is prime and $p \ge N$. The scheme has two levels. At the top level, a hash function partitions the elements being stored into s sets. The second level consists of a perfect hash function for each of these sets. Specifically, a function h chosen uniformly at random from $\mathcal{H}_s$ is used to partition the set S into s blocks. Let $W_j^h = \{x \in S \mid h(x) = j\}$; the superscript h is omitted when h is understood. Fredman, Komlós, and Szemerédi show that if a function h is chosen from $\mathcal{H}_s$ uniformly at random then

$$E\left(\sum_{0 \le j < s} \binom{|W_j|}{2}\right) \le \frac{n(n-1)}{s} \tag{2}$$

(where $E(X)$ denotes the expectation of the random variable X), and consequently that

$$\Pr\left(\sum_{0 \le j < s} \binom{|W_j|}{2} < \frac{2n(n-1)}{s}\right) \ge \frac{1}{2}. \tag{1}$$

Choosing $s = 2(n-1)$, relation $(*)$ implies that for at least half of the functions $h \in \mathcal{H}_s$ one has

$$\sum_{0 \le j < s} \binom{|W_j|}{2} < n.$$

Such a function is used to partition S into blocks W_j, $0 \le j < s$. For each block W_j one uses relation $(*)$ with $s_j = \max\{1, 2|W_j|(|W_j| - 1)\}$. It follows that for at least half of the functions $h \in \mathcal{H}_{s_j}$ one has

$$\sum_{0 \le l < s_j} \binom{|W_{j,l}|}{2} < 1,$$

where $W_{j,l} = \{x \in W_j \mid h(x) = l\}$, i. e., $|W_{j,l}| \le 1$ for all l. For each j therefore at least half of the functions in $\mathcal{H}_{s_j}$ are injective on W_j. One uses one such function for each W_j. The total space requirement is linear since

$$\sum_{0 \le j < s} s_j \le s + 4 \cdot \sum_{0 \le j < s} \binom{|W_j|}{2} = O(n)$$

by the choice of the hash functions.

For the dynamic case, we use the standard doubling method to deal with the fact that we do not know in advance how big the top-level table or any of the subtables will get.

Suppose that n is the current number of elements stored in the table. The FKS scheme in use will accommodate up to M elements. The value of M will initially be set to $(1+c) \cdot n$ for some $c > 0$ and as n changes will never be more than $\frac{1+c}{1-c} \cdot n$. Let $s(M)$, to be specified, be the number of sets into which the top level hash function is to partition the elements of S. The function h will be a random element of $\mathcal{H}_{s(M)}$. Thus, the set S is partitioned by h into the subsets $W_j = \{x \in S \mid h(x) = i\}$, $0 \le j < s(M)$.

Let T_j be the block of memory used for storing W_j. The amount of space allocated to T_j is s_j, where $s_j = 2m_j(m_j - 1)$, and m_j is the maximal size of W_j the current table T_j is meant to manage. The quantity m_j is always as least as big as $|W_j|$ and is at most twice the number of all elements ever mapped to j by the current top level function h. The subset W_j is resolved within T_j by using a perfect hash function h_j from $\mathcal{H}_{s_j}$. If the value k_j specifies which hash function h_j is being used, then $x \in W_j$ is stored in location $(k_j x \bmod p) \bmod s_j$ of subtable T_j. It will be arranged that the following condition is always satisfied:

$$\sum_{0 \leq j < s(M)} s_j \;\leq\; \frac{32M^2}{s(M)} + 4M \; . \tag{7}$$

The parameter $s(M)$ will be chosen to be $\Theta(n)$ so that the right hand side of this equation is $O(n)$. We will see that this guarantees that the total space used is linear in the number of elements currently stored in the table.

The algorithm can be specified more precisely as described in the program given in Figures 1 and 2 below. The variable *count* keeps track of the number of updates performed in the hash table of the present size M. From time to time it becomes necessary to restructure the whole table. This is the case when *count* reaches M or when $(**)$ becomes wrong. In both cases, we start a *new phase*, resetting M to the new value $(1+c)\cdot n$, where n is the number of elements currently stored in the dictionary; the variable *count* is set to n, so that the system is able to perform up to $c \cdot n$ updates before the beginning of the next phase. Deletions are performed by attaching a tag "deleted" to the table entry to be erased; only when a new level-1 hash function h or a new hash function h_j for the subtable T_j is chosen, do we drop the elements with a tag "deleted" from T_j.

Let us first analyze the space needed by the scheme. By a *phase* we mean the time period during which one level-1 hash function h is "in use": a phase starts when some h is chosen and ends when the next level-1 function is chosen, either in the same or the subsequent call to *RehashAll*. Phases that only consist of choosing an h to find out that h does not satisfy condition $(**)$ are called *degenerate*. During any phase that starts with n keys being stored in the dictionary the number of keys will never drop below $(1-c)\cdot n$, since at most $c \cdot n$ updates are made. Thus, the following lemma is sufficient to prove the claimed space bound.

Lemma 2.1 *The memory space used during a phase that starts with n keys being stored in the dictionary is $O(n)$.*

Proof: The lemma is obviously true for degenerate phases, since $s(M) = \Theta(n)$. Thus, we assume that a function h is chosen that satisfies condition $(**)$, and determine how big the table T has to be to accommodate all versions of all subtables. For $0 \leq j < s(M)$, let $\bar{m}_j$ denote the final capacity of T_j, that is, the value of m_j at the end of the phase, and let $\bar{s}_j = 2\bar{m}_j(\bar{m}_j - 1)$ be the final size of T_j. The previous versions of T_j (if there were any) had capacity $\frac{1}{2}\bar{m}_j$, $\frac{1}{4}\bar{m}_j$, $\ldots$. Since, for $l \geq 0$,

$$2 \cdot (2^{-l} \cdot \bar{m}_j)(2^{-l} \cdot \bar{m}_j - 1) \leq 4^{-l} \cdot 2\bar{m}_j(\bar{m}_j - 1) = 4^{-l} \cdot \bar{s}_j,$$

```
procedure Insert(x);
  count ← count + 1;
  if count > M
  then
    RehashAll(x);
  else
    j ← h(x);
    if position h_j(x) of subtable T_j contains x
      then
        if x is marked "deleted" then remove this tag;
      else (* x is new for W_j *)
        b_j ← b_j + 1;
        if b_j ≤ m_j
          then (* size of T_j sufficient *)
            if position h_j(x) of T_j is empty
              then
                store x in position h_j(x) of T_j;
              else
                go through the subtable T_j, put all elements
                not marked "deleted" into a list L_j, and
                mark all positions of T_j empty;
                append x to list L_j; b_j ← length of L_j;
                repeat h_j ← randomly chosen function in H_{s_j}
                until h_j is injective on the elements of list L_j;
                for all y on list L_j store y in position h_j(y) of T_j;
          else (* T_j is too small *)
            m_j ← 2 · max{1, m_j}; s_j ← 2m_j(m_j − 1);
            if condition (**) is still satisfied
              then (* double capacity of T_j *)
                allocate new space, namely s_j cells, for new subtable T_j;
                go through old subtable T_j, put all elements
                not marked "deleted" into a list L_j,
                and mark all positions empty;
                append x to list L_j; b_j ← length of L_j;
                repeat h_j ← randomly chosen function in H_{s_j}
                until h_j is injective on the elements of list L_j;
                for all y on list L_j store y in position h_j(y) of T_j;
              else (* level-1-function h "bad" *)
                RehashAll(x);
```

Figure 1: Insertion

```
procedure RehashAll(x);
    (* RehashAll(x) is either called by Insert(x), and then x ∈ U,
        or by Delete(x), and then x = −1. RehashAll(x) builds a new table
        for all elements currently in the table plus x (if x ∈ U).          *)
  go through the whole table T, put all elements not tagged "deleted"
  into a list L, count them, and mark all positions in T "empty";
  if x ∈ U then append x to L;
  count ← length of list L;
  M ← (1 + c) · max{count, 4};
  repeat h ← randomly chosen function in H_{s(M)};
    for all j, 0 ≤ j < s(M), do form a list L_j of all x ∈ L with h(x) = j;
    for all j, 0 ≤ j < s(M), do
      b_j ← length of list L_j;  m_j ← 2 · b_j;  s_j ← 2m_j(m_j − 1);
  until condition (**) is satisfied;
  for all j, 0 ≤ j < s(M), do
    allocate space s_j for subtable T_j;
    repeat h_j ← randomly chosen function in H_{s_j}
    until h_j is injective on the elements of list L_j;
    for all x on list L_j do store x in position h_j(x) of T_j;

procedure Delete(x);
  count ← count + 1;
  j ← h(x);
  if position h_j(x) of subtable T_j contains x
    then mark x as "deleted"
    else return(x is not a member of S);
  if count ≥ M
    then (* start new phase *)
      RehashAll(−1);

procedure Lookup(x);
  j ← h(x);
  if position h_j(x) of subtable T_j contains x (not marked "deleted")
    then return("x is a member of S")
    else return("x is not a member of S");

procedure Initialize;
  T ← an empty table;
  RehashAll;
```

Figure 2: Setup, Deletion, Lookup, and Rehashing

the total number of cells occupied by all versions of all subtables T_j is bounded by

$$\sum_{0\le j<s(M)} \sum_{l\ge 0} 4^{-l} \cdot \bar{s}_j \;=\; \frac{4}{3}\cdot \sum_{0\le j<s(M)} \bar{s}_j \;\le\; \frac{4}{3}\cdot\left(\frac{32M^2}{s(M)} + 4M\right).$$

The last inequality holds since the algorithm makes sure that condition (**) remains valid throughout the phase.

The space required by the header table is at most $5s(M)$, since the jth entry of the header table need only contain a pointer to T_j, the variables s_j, b_j, and m_j, and the number k_j that describes the hash function h_j. If we let $s(M) = \frac{8}{15}\sqrt{30}\cdot M$, the space needed by the subtables and the header table taken together is bounded by $\frac{4}{3}\cdot\left(\frac{32M^2\cdot 15}{8\sqrt{30}\cdot M} + 4M\right) + 5\cdot\frac{8}{15}\sqrt{30}\cdot M = \frac{16}{3}(\sqrt{30}+1)\cdot M < 35\cdot(1+c)\cdot n$, which proves the lemma. □

Now we turn to the time bounds. Note first that membership queries do not interfere with the time analysis, since they are executed in constant time in the worst case. Thus, there is no harm in assuming that there are no membership queries at all. Note further that instructions that are executed in constant time (i. e., deletions in any case and insertions if they do not cause a subtable T_j to be rearranged) can be safely ignored, since they will not invalidate an overall linear time bound. Thus, we only need to worry about the time spent for installing new level-1 functions at the beginning of a phase (in *RehashAll*), and for constructing new versions of the subtables T_j (in *RehashAll* or in *Insert*).

Lemma 2.2 *The expected time for a phase that starts with n keys being stored in the dictionary is $O(n)$.*

Proof: Consider the call to *RehashAll* in which the phase starts. Clearing the old table (header table and the subtables) and building up the list L takes time $O(n)$, since by 2.1 the old table occupies only space $O(n)$. Time linear in n suffices to construct the sublists L_j, to compute the values b_j, m_j, and s_j, $0 \le j < s(M)$, and to compute $\sum_{0\le j<s(M)} s_j$. Thus, if the phase is degenerate, it takes $O(n)$ time in the worst case. In a non-degenerate phase h initially satisfies (**). By the remarks immediately preceding Lemma 2.2, we only have to estimate the time spent for installing new hash functions h_j for the subtables. Fix some j, and split the phase, as far as T_j is concerned, into subphases, one subphase being defined as a maximal time period in which the capacity m_j and hence the size s_j of T_j have a fixed value. We need the following observation:

Claim: Assume a hash function h_j is chosen for T_j at the beginning or in the course of a subphase. Then the probability that h_j stays in use until the end of the subphase exceeds $\frac{1}{2}$.

Proof of claim: Let the capacity of T_j during the phase be m_j. Let W_j be the set of keys x contained in the list L_j when h_j is chosen. Let V_j be the set of the first

$m_j - |W_j|$ *different* keys x in the sequence of the *Insert* instructions to be executed next that satisfy $h(x) = j$ and do not occur in W_j. Then, by relation $(*)$, table size $s_j = 2m_j(m_j - 1)$ (for $m_j \neq 0$) implies that with probability exceeding $\frac{1}{2}$ the elements of $W_j \cup V_j$ will be mapped by h_j to different locations in T_j. If this happens, the way b_i is changed and repeated keys are treated in *Insert* and *Delete* implies that h_j stays in use until b_j grows beyond m_j; that is, until the end of the subphase. □

By the claim, the probability that u or more hash functions h_j are used in a single subphase is at most $2^{-(u-1)}$, and hence the expected number of functions h_j chosen during the subphase is bounded by 2. Thus, the expected cost for installing new hash functions h_j during a subphase in which T_j has size s_j is $O(s_j)$. Exactly as in the proof of Lemma 2.1 we get an overall bound of $O(M) = O(n)$ for the expected time for installing new hash functions h_j, $0 \le j < s(M)$, by summing over all subtable sizes and all j, and using $(**)$. □

In order to finish the time analysis, we will show in the following two lemmas that there will not be too many phases. Fix some phase, and let S be the set of elements stored in the table at the beginning of the phase (whose number is n) together with those that occur in the next $c \cdot n$ update instructions to be executed (repeated elements are only counted once). Let $M = (1+c)\cdot n$; clearly, $|S| \le M$. For $h \in \mathcal{H}_{s(M)}$ chosen at random, define $W_j = \{x \in S \mid h(x) = j\}$, $0 \le j < s(M)$.

Lemma 2.3 *(a)* *With probability exceeding $\frac{1}{2}$ we have*

$$\sum_{0 \le j < s(M)} 4|W_j|(2|W_j| - 1) < \frac{32M^2}{s(M)} + 4M.$$

(b) If the inequality in (a) is satisfied for the level-1 function h chosen at the beginning of the phase, then the phase ends with the variable count reaching M; i. e., the phase comprises $c \cdot n$ updates.

Proof: (a) In the situation just described, relation $(*)$ reads

$$\Pr\left(\sum_{0 \le j < s(M)} \binom{|W_j|}{2} \le \frac{2M(M-1)}{s(M)}\right) \ge \frac{1}{2}.$$

The claim follows by a simple transformation, using the obvious inequality $\sum_{0 \le j < s(M)} |W_j| = |S| \le M$.

(b) It is immediate from the way the variables b_j and m_j are initialized in *RehashAll* and updated in *Insert*, and from the fact that only keys from S can occur in the phase, that $b_j \le |W_j|$, and hence $m_j \le 2|W_j|$, throughout the phase. Since $s_j = 2m_j(m_j - 1)$ throughout the algorithm, we see that the inequality in (a) entails that $(**)$ stays valid throughout the phase. □

Lemma 2.4 *Suppose that RehashAll is called at a time when $n \geq 1$ keys are stored in the dictionary. Then the (expected) time needed until the first call to RehashAll after $c \cdot n$ updates have been performed is $O(n)$.*

Proof: Consider an arbitrary phase that starts before the next $c \cdot n$ updates have been processed. The number of keys in the table at the beginning of this phase is n', where $(1-c) \cdot n < n' < (1+c) \cdot n$. By Lemma 2.3(a)(b), the probability that during this phase $c \cdot n'$ updates are performed exceeds $\frac{1}{2}$. Since $c \cdot n - |n'-n| \leq c \cdot n'$ no matter if $n' < n$ or $n' \geq n$, this means that the probability that this phase extends further than the $c \cdot n$ updates we are considering is at least $\frac{1}{2}$. Thus, the expected number of phases needed to perform these $c \cdot n$ updates is not more than 2. Each phase occurring starts with n' keys, $(1-c) \cdot n < n' < (1+c) \cdot n$, and takes $O(n)$ steps (expected) by 2.2. This finishes the proof of 2.4 and the time analysis. □

Lemmas 2.1–2.4 taken together yield the following result.

Theorem 2.5 *Dynamic perfect hashing, as described by the algorithm in this section, uses linear space, needs constant time for membership queries, and has $O(1)$ expected amortized insertion and deletion cost.* □

Remark 2.6 Obviously, the space bound $35(1+c)n$ proved in 2.1 is not satisfactory from a practical point of view. There are many conceivable ways of reducing the space bound, by varying the parameters fixed in the algorithm, by using slightly different hash functions, or by adapting more involved schemes, e. g., that described in [FKS84], which achieves an $n + o(n)$ space bound in the static case. Most of these variations will increase the bounds on the expected computation time, but this does not necessarily mean that the time requirements observed in practice will grow significantly. M. Wenzel [W90] has implemented a variant of the scheme described above. In his implementation the universe U is $\{0, 1, \ldots, 2^{31}-1\}$; the space requirements are kept small by avoiding the use of subtables if $|W_j|$ is small. He reports that the space requirements of his implementation are comparable to those of balanced trees and that the running time is superior to search trees provided n is moderately large ($n \geq 1000$). We refer the reader to [W90] for details.

3 Higher order hash functions

In this section we generalize inequality (†) from Section 2 (which originated in [FKS84]) to polynomials of degree larger than 1, and note some consequences of this generalization. These extensions have proved useful since the first version of this paper appeared as [DKM88], see, e. g., [DM89, DM90a, DM90b]. In order to formulate the result in a slightly more general way than just for polynomials, we recall a definition given originally in [WC79], and studied further (with varying notation) for example in [MV84, S89].

Definition 3.1 ([WC79]) *Let $\mathcal{H}$ be a collection of functions h with domain D and range R. Let $c > 0$ and $k \in I\!N$. The class $\mathcal{H}$ is called* (c, k)-universal *if for all sequences $x_1, \ldots, x_k$ of different elements of D, all sequences $y_1, \ldots, y_k$ of elements of R, and randomly chosen $h \in \mathcal{H}$*

$$\Pr(h(x_i) = y_i \textit{ for } 1 \le i \le k) \le \frac{c}{|R|^k}.$$

(Alternatively, such classes have been called c strongly k universal or $(k)_c$-independent.)

Examples: (a) [WC79] If F is a finite field, we may let $D = R = F$; then

$$\mathcal{H} = \{h \mid h(x) = \sum_{0 \le i < k} a_i\, x^i \text{ for } x \in F,\ a_0, \ldots, a_k \in F\}$$

is $(1, k)$-universal. This holds since for each sequence of k different arguments in F and k prescribed values there is exactly one polynomial of degree at most $k - 1$ that interpolates through these argument-value pairs.
(b) [WC79, MV84] If $\mathcal{H}$ is (c, k)-universal and $r : R \to R'$ is such that $|r^{-1}(j)| \le d$ for all j then the (multi)set $\mathcal{H}' = \{r \circ h \mid h \in \mathcal{H}\}$ is $(\hat{c}, k)$-universal, for $\hat{c} = c \cdot (d|R'|/|R|)^k$.
(c) A direct consequence of (a) and (b): If p is prime, and $1 \le s \le p$, then for $D = \{0, \ldots, p-1\}$ and $R = \{0, \ldots, s-1\}$ the set

$$\mathcal{H}_s^k = \left\{\ h : D \to R\ \middle|\ h(x) = \Bigl(\sum_{0 \le i < k} a_i\, x^i \bmod p\Bigr) \bmod s,\ 0 \le a_0, \ldots, a_{k-1} < p\ \right\}$$

is (c, k)-universal, for $c = (\lceil p/s \rceil \cdot s/p)^k \le (1 + s/p)^k$.
(d) For the finite field $D = R = GF(p^l)$, p prime, $l \ge 1$, we obtain $(1, k)$-universal classes with $|R'| = p^{l'}$, $1 \le l' \le l$, by combining (a) with a suitable function $r : R \to R'$. (See [MV84] for further examples.)

In the following, we assume that D and $R = \{0, \ldots, s-1\}$ are fixed, and that $\mathcal{H}$ is a class of functions from D to R. Let a set $S \subseteq D$ be fixed, $|S| = n$, and let x_0 be an element of $D - S$. For $h \in \mathcal{H}$ and $0 \le j < s$ we define $B_j^h = \{x \in S \mid h(x) = j\}$ and $b_j^h = |B_j^h|$; further, we define $B_{x_0}^h = \{x \in S \mid h(x) = h(x_0)\}$ and $b_{x_0}^h = |B_{x_0}^h|$. Assume that h is chosen uniformly at random from $\mathcal{H}$. (In the notation, we drop the superscript h.) For arbitrary $z \in I\!R$, $k \ge 0$, we let $(z)_k$ denote the "falling factorial" $z(z-1)\ldots(z-k+1)$.

Lemma 3.2 *If $\mathcal{H}$ is (c, k)-universal for D and R, then*

(a) $E((b_j)_k) \le c \cdot \dfrac{(n)_k}{s^k} \le c \cdot \left(\dfrac{n}{s}\right)^k$, *for* $0 \le j < s$;

(b) $E((b_{x_0})_{k-1}) \le c \cdot \dfrac{(n)_{k-1}}{s^{k-1}} \le c \cdot \left(\dfrac{n}{s}\right)^{k-1}$.

Proof: For $l \ge 1$, let $(S)_l$ denote the set $\{(x_1, \ldots, x_l) \in S^l \mid x_1, \ldots, x_l \text{ different}\}$.

(a) Fix j, and define random variables $X_{x_1,\ldots,x_k}, (x_1, \ldots, x_k) \in (S)_k$, by

$$X_{x_1,\ldots,x_k} = \begin{cases} 1, & \text{if } h(x_1) = \ldots = h(x_k) = j, \\ 0, & \text{otherwise.} \end{cases}$$

Then $E(X_{x_1,\ldots,x_k}) = \Pr(X_{x_1,\ldots,x_k} = 1) \le c/s^k$, since $\mathcal{H}$ is (c,k)-universal. On the other hand, it is clear that

$$(b_j)_k = \sum_{(x_1,\ldots,x_k)\in(S)_k} X_{x_1,\ldots,x_k}.$$

Consequently,

$$E((b_j)_k) = \sum_{(x_1,\ldots,x_k)\in(S)_k} E(X_{x_1,\ldots,x_k}) \le |(S)_k| \cdot \frac{c}{s^k} = (n)_k \cdot \frac{c}{s^k}.$$

(b) The proof is similar to the one given in (a). Define random variables $Y^j_{x_1,\ldots,x_{k-1}}$, $(x_1, \ldots, x_{k-1}) \in (S)_{k-1}$, $0 \le j < s$, by

$$Y^j_{x_1,\ldots,x_{k-1}} = \begin{cases} 1, & \text{if } h(x_1) = \ldots = h(x_{k-1}) = h(x_0) = j, \\ 0, & \text{otherwise.} \end{cases}$$

Then $E(Y^j_{x_1,\ldots,x_{k-1}}) \le c/s^k$, since $\mathcal{H}$ is (c,k)-universal. Further,

$$(b_{x_0})_{k-1} = \sum_{(x_1,\ldots,x_{k-1})\in(S)_{k-1}} \sum_{0\le j<s} Y^j_{x_1,\ldots,x_{k-1}}.$$

Taking expected values, we get

$$E((b_{x_0})_{k-1}) \le |(S)_{k-1}| \cdot s \cdot \frac{c}{s^k} = (n)_{k-1} \cdot \frac{c}{s^{k-1}},$$

as claimed. □

A hash function h is called *l-perfect* for S if $b^h_j \le l$ for all j, $0 \le j < s$, i.e., if no block B^h_j has size exceeding l.

Corollary 3.3 *In the situation of Lemma 3.2, if we further assume that $s \ge n$, we have:*

(a) $\Pr(h$ is $(k-1)$-perfect$) \ge 1 - (c/k!) \cdot n \cdot (n/s)^{k-1}$. *In case $s = n^{1+1/(k-1)}$ this probability exceeds $1 - c/k!$.*

(b) $E\Big(\sum_{0\le j<s} (b_j)^k\Big) \le c_k \cdot n$, *for some constant c_k. (Here $(b_j)^k$ is the k-th power of b_j.)*

(c) $\Pr\Big(\sum_{0\le j<s} (b_j)^k \le 2c_k \cdot n\Big) \ge \frac{1}{2}$, *for* c_k *as in (b).*

Corollary 3.4 *In the situation of Lemma 3.2 we have:*

(a) For $0 \le j < s$ *arbitrary:*

$$\Pr(b_j \ge u) \le \begin{cases} c\cdot(e^{u-1}/u^u)\cdot(n/s)^u, & \text{for } 1 \le u < k; \\ c\cdot(e^{k-1}/u^k)\cdot(n/s)^k, & \text{for } k \le u. \end{cases}$$

In particular, for $s \ge n$ *and* $u \ge k$, *we have* $\Pr(b_j \ge u) = O(u^{-k})$.

(b)

$$\Pr(b_{x_0} \ge u) \le \begin{cases} c\cdot(e^{u-1}/u^u)\cdot(n/s)^u, & \text{for } 1 \le u < k-1; \\ c\cdot(e^{k-2}/u^{k-1})\cdot(n/s)^{k-1}, & \text{for } k-1 \le u. \end{cases}$$

In particular, for $s \ge n$ *and* $u \ge k-1$, *we have* $\Pr(b_{x_0} \ge u) = O(u^{-(k-1)})$.

(**Note:** The special case $k = u$ in 3.4(a) has already been analyzed in [MV84].)

Proof of Corollary 3.3:
(a) We estimate the probability that h is not $(k-1)$-perfect.

Clearly

$$\Pr(\exists j : b_j \ge k) = \Pr(\exists j : (b_j)_k \ge k!) \le s \cdot \max\{\Pr((b_j)_k \ge k!) \mid 0 \le j < s\}.$$

By 3.2(a) and the Markov inequality the last term is bounded above by $s\cdot(c/k!)\cdot(n/s)^k = (c/k!)\cdot n\cdot(n/s)^{k-1}$.

(b) Let $J = \{\, j \mid b_j \le k-1\}$. Since $\sum_{j\in J} b_j \le |S| = n$, it follows from elementary considerations that $\sum_{j\in J}(b_j)^k \le \lfloor\frac{n}{k-1}\rfloor\cdot(k-1)^k + (n - \lfloor\frac{n}{k-1}\rfloor\cdot(k-1))^k \le n\cdot(k-1)^{k-1}$.

We need the following simple fact:

Claim: If $z \ge k$, then $z^k/(z)_k < e^{k-1}$.

(*Proof of claim:*

$$\frac{z^k}{(z)_k} = \prod_{j=1}^{k-1}\frac{z}{z-j} \le \prod_{j=1}^{k-1}\frac{k}{k-j} = \frac{k^{k-1}}{(k-1)!} = \sum_{l=0}^{k-1}\binom{k-1}{l}\cdot\frac{(k-1)^l}{(k-1)!} < \sum_{l=0}^{k-1}\frac{(k-1)^l}{l!} < e^{k-1}.)$$

Thus, we may write:

$$E\Big(\sum_{0\le j<s} b_j^k\Big) \le E\Big(\sum_{j\in J} b_j^k + \sum_{j\notin J} b_j^k\Big) \le n\cdot(k-1)^{k-1} + e^{k-1}\cdot E\Big(\sum_{0\le j<s}(b_j)_k\Big).$$

By 3.2, we obtain for $c_k = (k-1)^{k-1} + ce^{k-1}$ that $E(\sum_{0\le j<s} b_j^k) \le c_k \cdot n$, as claimed.

(c) is immediate from (b). □

Proof of Corollary 3.4:
(a) Assume first that $u \ge k$. Then, by 3.2(a),

$$\Pr(b_j \ge u)\cdot (u)_k = \Pr((b_j)_k \ge (u)_k)\cdot (u)_k \le E((b_j)_k) \le c\cdot \left(\frac{n}{s}\right)^k,$$

whence we get

$$\Pr(b_j \ge u) \le \frac{c}{(u)_k}\cdot \left(\frac{n}{s}\right)^k.$$

By the claim in the proof of 3.3(b), this implies

$$\Pr(b_j \ge u) \le \frac{c\cdot e^{k-1}}{u^k}\cdot \left(\frac{n}{s}\right)^k.$$

In case $1 \le u \le k-1$ it is easily seen that $\mathcal{H}$ is also (c,k)-universal. Applying the above result yields the desired estimate $\Pr(b_j \ge u) \le c\cdot e^{u-1}\cdot (n/s)^u/u^u$.

(b) The argument is exactly the same as in (a). Just use 3.2(b) instead of 3.2(a). □

4 Optimal lower bounds for the deterministic case

In this and the following section we consider *deterministic* algorithms for the dictionary problem that are based on hashing, and lower bounds on their performance. It will turn out that such deterministic algorithms must be much slower than the randomized algorithms described in the preceding sections.

As a basis for our lower bound proofs we introduce a simplified, abstract type of algorithm. Such algorithms maintain the following data structure D. If $S \subseteq U$ is the set of elements in the dictionary, then D consists of a rooted tree whose leaves are labelled with the elements of S. The inner nodes are labelled with hash functions whose values correspond to the edges leaving the node. In order to access a key $x \in S$, one starts at the root and repeatedly evaluates the hash function at the current node (with x as argument) to determine the edge to be followed out of the node until a leaf is reached. This leaf has label x. This data structure generalizes the one used in Section 2, where two hash functions had to be evaluated to access a key. We count one step for the evaluation of a hash function.

In more detail, the data structure can be described as follows. D is a rooted tree in which each inner node v is labelled with a hash function $h_v : U \to \{0,1,\ldots,m_v-1\}$, with $m_v \ge 2$, and has m_v children, one for each value of h_v. Each $x \in U$ determines a path from the root to a leaf. This path is given by $w_0, w_1, \ldots, w_r$, where w_0 is the root, w_{t+1} is the $h_{w_t}(x)$-th child of w_t, for $0 < t < r$, and w_r is a leaf. We say that D is a dictionary for $S = \{x_0, \ldots, x_n\} \subseteq U$ if each leaf contains exactly one of the x_i. To each node v of D we associate the set $A(v) \subseteq U$ of keys that are "sent to" v by the hash functions on the path from the root to v. We define inductively: $A(v) = U$ for v the root, and $A(v_q) = \{x \in A(v) \mid h_v(x) = q\}$ for $0 \leq q < m_v$ where v_q, $0 \leq q < m_v$, are the children of v.

For our lower bound arguments, we will consider only insertions. To insert a key $x_{n+1} = x \in U$ into a dictionary D for S, we follow the path $w_0, w_1, \ldots, w_r$ determined by x, and for some node v on this path (determined by the algorithm) perform a *rehashing* at v, which means that we choose a new perfect hash function h_v for $A(v) \cap (S \cup \{x\})$. Thus, all $|A(v) \cap (S \cup \{x\})|$ children of v become leaves, and to each of them corresponds exactly one element of $A(v) \cap (S \cup \{x\})$. Such a rehashing *must* be performed for exactly one node v on the path. The cost of such an insertion is $\mathrm{depth}(v) + |A(v) \cap (S \cup \{x\})|$. The cost of inserting $x_1, x_2, \ldots, x_n \in U$ into a dictionary D is the sum of the costs of the single insertions. Note that we assume that D initially contains one element x_0 in a leaf, with no root.

Remark 4.1 When a rehashing at v is performed, a perfect hash function for $A(v) \cap (S \cup \{x\})$ is given at linear cost; in addition, setting up the hash table, i. e., the subtree of depth 1, for this set has linear cost as well. This assumption excludes search trees that use an order on the universe U to define the way keys are distributed at nodes, as well as other schemes involving cleverly chosen hash functions that can be extended to additional keys at low cost while keeping the function injective.

Remark 4.2 (a) We require that collisions are resolved immediately by rehashing. In particular, we do not allow forming chains, i. e., linked lists, at the leaves of the tree as is done in many hashing schemes. But the absence of this restriction would not change the lower bounds by much. If we were to allow chaining, inserting n elements would cost n steps, because we could insert each element at the head of the chain, which would mean constant time per insertion. To justify our model, we have to consider tasks with insertions *and* lookups. If after inserting x we include a lookup for the element at the end of the chain into which x was inserted, then this lookup costs essentially as much as rehashing at the leaf to which the chain belongs. Thus algorithms for insertions and lookups, with chaining allowed, are as least as costly as algorithms without chaining for insertions only.

(b) One could ask if it would be advantageous to also allow rehashings at nodes v that do not lie on the path determined by the x just being inserted. But it is easily checked that the algorithm does not become slower if such rehashings are performed at the time when the last element of $S \cap A(v)$ is inserted into D. Thus it is justified not to admit such "spontaneous" rehashings.

Remark 4.3 *The role of space limitations.* In the description of the data structure D, we have not introduced the concept of the space used by D. On the other hand, some space restriction is necessary, since using the identity function as the hash function at the root would make all rehashing superfluous.

If we assume that storing a hash function h_v together with the corresponding table takes space $O(m_v) = O(|range(h_v)|)$, then $\sum_{v \,\mathrm{node\,in}\, D} m_v$ is a reasonable measure for the space used by D. In our description of the data structure D we assumed that every leaf of D contained an element of S, so for every h_v and every $j \in \{0, \ldots, m_v - 1\}$ there is some $x \in S$ with $h_v(x) = j$. Since in every rooted tree with $n + 1$ leaves and outdegree at least two the number of edges is bounded by $2n$, our data structure D satisfies $\sum_v m_v \leq 2n$, which means that it needs linear space.

If the algorithm were allowed to use hash functions h_v with range larger than $|A(v) \cap S|$ when rehashing at node v, then the lower bounds given in the theorems below would still hold, with constants smaller by a factor of $\frac{1}{4}$ than those in the theorems. We only have to assume that the space used by D is not too large in relation to the size of the universe U (namely $|U| \geq (S(n) 2 \log n)^{2 \log n} \cdot (n + 1)$ in Theorem 4.4 and $|U| \geq (S(n)/k)^k$ in Theorem 4.6 for $S(n) = \sum_v m_v$). We shall comment on this in more detail below when the adversary strategies for the lower bound proofs are discussed.

We want to study the following quantities.

$T(n)$	$=$	worst-case (amortized) cost incurred by an optimal algorithm to insert n elements.
$T_{\max}(n)$	$=$	worst-case cost needed for a single insertion or membership query in a sequence of n instructions.
$T_k(n)$	$=$	worst-case amortized cost needed by an optimal algorithm to insert n elements, if the depth of the tree is not allowed to exceed k, i. e., if the worst case lookup time is k.

The following three theorems sum up the results (upper and lower bounds) concerning these three quantities. Theorem 4.4 shows that amortized time $O(n)$ for n insertions cannot be achieved in the deterministic case, but that a slowdown by a factor $\log n$ is unavoidable. Theorem 4.5 shows that in any case there will be single instructions that are very costly. If we demand constant lookup time to be guaranteed, Theorem 4.6 shows that this can only be achieved by many costly rehashings.

Theorem 4.4

(a) $T(n) \geq (n+1) \cdot \log(n+1)$, *if* $|U| \geq (n/\log n)^{2\log n} \cdot (n+1)$.

(b) $T(n) \leq 3(n+1)\log(n+1)$.

Theorem 4.5

(a) $T_{\max}(n) = \sqrt{n}$, *if* $|U| \geq 2(\sqrt{n})^{\sqrt{n}}$.

(b) If only algorithms with a total cost smaller than $f(n) \cdot n$ *for* n *insertions are considered, and* $|U| \geq (\frac{n}{f(n)})^{2f(n)} \cdot (n+1)$, *then* $T_{\max}(n) = \Omega(n/f(n))$.

Theorem 4.6

(a) $T_k(n) \geq (k/e) \cdot n^{1+1/k}$*for* $n \geq e^k$, *if* $|U| > (2n/k)^k$.

(b) $T_k(n) \leq d_k \cdot n^{1+1/k}$ *for all sufficiently large* n, *where the constants* d_k *can be chosen to satisfy* $d_k \sim k/e$. *(Here* $e = 2.71828\ldots =$ *Euler's constant.)*

The *proofs* of the theorems will be given in the next section.

Remark 4.7 If we reconsider the randomized algorithm presented in Section 2, we see that randomization is only used for constructing perfect hash functions at expected linear cost. Thus, if we give such hash functions at guaranteed linear cost, we should obtain a deterministic algorithm that is as least not slower than the randomized one. This seems to contradict our lower bounds! To resolve this paradox, consider adversary strategies for the randomized computation model. Here the adversary has to determine the moves of the strategy without knowledge of the outcomes of the coin flips of the algorithm to be executed. This means that the data structure produced by the algorithm cannot be taken into consideration by the adversary. But this is what happens in the deterministic case and what makes the adversary as strong as indicated in the lower bounds for the deterministic model.

Remark 4.8 Theorem 4.5 gives a lower bound for our model that is bigger than the $O(\log n)$ worst-case bound for single instructions guaranteed by implementations of dictionaries as balanced search trees. This is an effect of the quite severe restriction that rehashing at a node v has cost linear in the size of the subtree rooted at v. (Cf. Remark 1.1.)

5 Proofs of the lower bounds

This section contains the proofs of the theorems stated in Section 4.

5.1 The adversary strategy

For proving the lower bounds, we apply an adversary argument in each case. Let us first give a general description of the adversary strategy. Initially, the tree D contains one element x_0. The adversary chooses, step by step, the element x_i to be inserted next. Basically, x_i is always chosen in such a way that it has to follow a longest path in D.

In order to always be able to find such an element x_i, we must make sure that the set of elements of U that belongs to such a longest path is not empty. The aim of the adversary is to build up long paths $w_0, w_1, w_2, \ldots$ in the tree and to make sure that the sets $A(w_0), A(w_1), A(w_2), \ldots$ are as large as possible. Thus, if a decision is to be made which path to choose, the adversary will, at each node v, choose that child q of v that maximizes $|h_v^{-1}[q] \cap A(v)|$, $0 \leq q < m_v$. (If there is a tie, the smallest such q is chosen.) For the sake of simplicity of notation, we will assume that $q = 0$ always has this property. (If this is not the case, renumber the children of v.)

Assumption 5.1 *For all trees D ever built by the algorithms and for all nodes v of D, the set $A(v) \cap h_v^{-1}[0]$ is maximal (w. r. t. cardinality) among $A(v) \cap h_v^{-1}[q]$, $0 \leq q < m_v$.*

We will regard the child number 0 of v as the leftmost child of v, and define the leftmost path and the leftmost leaf in D accordingly (always follow the edge to child 0).

Simple adversary strategy: Choose $x_1 \neq x_0$ arbitrarily. For $i > 1$, assume that $x_1, \ldots, x_{i-1}$ have been inserted and that a tree D has been set up by the algorithm. Then let x_i be an arbitrary element of $A(v) - \{x_0, x_1, \ldots, x_{i-1}\}$, where v is the leftmost leaf of D.

Note that all elements inserted follow the leftmost path in D. This path grows as the result of inserting x_i if the algorithm chooses to perform a rehashing only at the leftmost leaf, or it is cut off at v if the algorithm performs a rehashing at an inner node v of the leftmost path.

Remark 5.2 We have made the assumption that in all nodes v of D all values of h_v are used by members of S (cf. Remark 4.3). This has the effect that each insertion causes a collision at some node, at the latest at the leaf reached by the newly inserted element, and hence causes a rehashing. If some values of h_v are not used by elements of S, it may happen that when x_i is inserted, it reaches a leaf that is not already occupied by a key from $\{x_0, \ldots, x_{i-1}\}$, hence no rehashing is necessary. However, observe that out of two subsequent insertions performed according to the adversary strategy at least one must cause a rehashing somewhere along the leftmost path. It is then seen that all lower bounds proved below hold under the assumption that not n but $2n$ keys are inserted, because they cause at least n rehashings.

The following lemma makes precise how big U has to be in order to guarantee that some suitable x_i is available in each step of the adversary strategy.

Lemma 5.3 *Let v be a node on the leftmost path in D, and let the depth of v in D be r. Then*

(a) $$|A(v)| \geq |U| \Big/ \left(\frac{2n}{r}\right)^r .$$

(b) If we drop the assumption (cf. 4.3) that for all nodes v in the tree $A(v) \cap S \neq \emptyset$, and regard $s(D) = \sum_{v \text{ node in } D} m_v$ as a measure for the space needed by D, then for v as in (a)

$$|A(v)| \geq |U| \Big/ \left(\frac{s(D)}{r}\right)^r .$$

Proof: Let $w_0, w_1, \ldots, w_r = v$ be the path from the root w_0 to v. By definition, $|A(w_0)| = |U|$; further, $|A(w_{t+1})| \geq |A(w_t)|/m_{m_t}$, by Assumption 5.1. Thus, $|A(v)| \geq |U|/(\prod_{t=0}^{r-1} m_{w_t})$. Obviously, $\sum_{t=0}^{r-1} m_{w_t} \leq s(D)$. From this it is easily seen that the denominator $\prod_{t=0}^{r-1} m_{w_t}$ cannot be larger than $(s(D)/r)^r$. This proves (b). As noted already in 4.3, if $A(v) \cap S \neq \emptyset$ for all nodes v in D, then $s(D) \leq 2n$. This proves (a). $\Box$

Lemma 5.4 *Let $\bar{T}(n)$ denote the minimal number of steps needed by any algorithm for inserting n elements, if these elements are chosen according to the simple adversary strategy. (In particular, the algorithm has to admit the simple adversary strategy, which means that for each $i < n$ we have that after inserting x_i the set $A(v) - \{x_0, x_1, \ldots, x_i\}$ is nonempty, for v the leftmost leaf in D.) Then*

$$\bar{T}(n) \geq (n+1)\log(n+1).$$

Proof: (Induction on n.) Fix such an algorithm for n elements. Clearly, $\bar{T}(0) = 0$, $\bar{T}(1) = 2$ (rehashing at the root is forced). Let $n > 1$. Let $1 \leq i \leq n$ where i is maximal such that x_i is inserted by rehashing at the root. (Such an i exists, since this applies to $i = 1$.) Inserting $x_1, \ldots, x_{i-1}$ costs at least $\bar{T}(i-1)$, by the definition of $\bar{T}$, inserting x_i costs $i+1$, inserting $x_{i+1}, \ldots, x_n$ costs at least $n - i + \bar{T}(n-i)$, since the hash function at the root has to be evaluated for $x_{i+1}, \ldots, x_n$, and all these elements are sent into the leftmost subtree and have to be inserted there, and are chosen according to the simple adversary strategy with respect to this subtree. (Note that this subtree already has an element.) Thus

$$\bar{T}(n) \geq \bar{T}(i-1) + (i+1) + (n-i) + \bar{T}(n-i).$$

By the induction hypothesis, this entails

$$\bar{T}(n) \geq i\log i + (n+1-i)\log(n+1-i) + n + 1,$$

and the right hand side of the last inequality is at least $(n+1)\log(n+1)$, since the function $y\log y + (n+1-y)\log(n+1-y)$ attains its minimum in the range $1 \leq y \leq n$ in $y = (n+1)/2$. $\Box$

5.2 Proof of Theorem 4.4

We first consider the lower bound (part(a)). We would like to use the adversary strategy described above. However, to provide for the case that the leftmost path in D becomes very long and U is not as big as demanded in 5.3, we must slightly change the adversary strategy: We choose x_i so that it aims at the $\lfloor 2\log n\rfloor$-th node on the leftmost path in D.

Modified adversary strategy: Choose $x_1 \neq x_0$ arbitrarily. For $i > 1$, assume that $x_1, \ldots, x_{i-1}$ have been inserted and that a tree D has been set up by the algorithm. Let $w_0, w_1, \ldots, w_r$ be the path from the root to the leftmost leaf in D. Choose x_i to be an arbitrary element of $A(w_{r'}) - \{x_0, \ldots, x_{i-1}\}$, where $r' = \min(r, \lfloor 2\log n\rfloor)$.

By Lemma 5.3, this strategy will work as long as $|U|/(\frac{2n}{2\log n})^{2\log n} \geq n+1$, i.e., $|U| \geq (\frac{n}{\log n})^{2\log n} \cdot (n+1)$.

Define

$$L = \{\, x_i \mid 1 \leq i \leq n,\ \text{depth}(v) \geq 2\log n \text{ for the vertex } v \text{ in } D \text{ at which rehashing is performed when } x_i \text{ is inserted}\,\}.$$

Clearly, for each $x_i \in L$ the cost of evaluating the hash functions on the way down to v alone is at least $2\log n$. We determine a lower bound for inserting the elements in $\{x_{i_1}, x_{i_2}, \ldots, x_{i_{n'}}\} = \{x_1, \ldots, x_n\} - L$ into the tree as follows. (Here, $n' = n - |L|$.) Observe that if we disregard all elements $x_i \in L$ and all inner nodes at depth $\geq 2\log n$ in the computation for $x_1, \ldots, x_n$, then we obtain a computation in which $x_{i_1}, \ldots, x_{i_{n'}}$ are inserted into a dictionary that always has depth smaller than $2\log n$, and $x_{i_1}, \ldots, x_{i_{n'}}$ are chosen according to the simple adversary strategy considered in Lemma 5.4. Thus we may conclude from Lemma 5.4 that inserting $x_{i_1}, \ldots, x_{i_{n'}}$ has cost at least $(n'+1)\log(n'+1)$. Altogether we get

$$\begin{aligned} T(n) &\geq |L| \cdot 2\log n + (n - |L| + 1) \cdot \log(n - |L| + 1) \\ &\geq \min_{0 \leq y \leq n-1} (y \cdot 2\log n + (n-y+1) \cdot \log(n-y+1)). \end{aligned}$$

For $n \geq 4$, the minimum is attained for $y = 0$; hence $T(n) \geq (n+1)\log(n+1)$. For $n = 1, 2, 3$, the lower bound in Theorem 4.4 is obvious. This finishes the proof of 4.4(a).

To prove the upper bound in Theorem 4.4 (part (b)), we use the following algorithm for arbitrary n: Perform a global rehashing (i.e., a rehashing at the root) for x_i if i is a power of 2. Choose the hash functions h_v, for v the root, in such a way that $|h_v^{-1}[q]| = 1$ for all $q > 0$; then all insertions that do not cause a rehashing at the root go into the leftmost subtree, to which the same algorithm is applied recursively. Let $\tilde{T}(n)$ = cost of this algorithm when applied to n elements. By inspection, $\tilde{T}(1) = 2$, $\tilde{T}(2) = 5$, $\tilde{T}(3) = 8$. We claim that $\tilde{T}(n) \leq 3(n+1)\log(n+1)$ for all n. Fix $n \geq 4$ and let $t = \lfloor \log n \rfloor$. We split $x_1, \ldots, x_n$ into three groups and two single elements:

– inserting $x_1, \ldots, x_{2^{t-1}-1}$ costs $\tilde{T}(2^{t-1}-1)$;

– inserting $x_{2^{t-1}}$ costs $2^{t-1}+1$;

– inserting $x_{2^{t-1}+1}, \ldots, x_{2^t-1}$ costs $\tilde{T}(2^{t-1}-1)+(2^{t-1}-1)$;

– inserting x_{2^t} costs 2^t+1;

– inserting $x_{2^t+1}, \ldots, x_n$ costs $\tilde{T}(n-2^t)+(n-2^t)$.

Thus, by the induction hypothesis,

$$\tilde{T}(n) \leq 2+2^{t-1}+n+2\cdot 3\cdot 2^{t-1}\log(2^{t-1})+3\cdot(n-2^t+1)\log(n-2^t+1).$$

With $2+2^{t-1}+n \leq 3\cdot 2^t$ it follows that

$$\tilde{T}(n) \leq 3\cdot 2^t\log(2^t)+3\cdot(n-2^t+1)\log(n-2^t+1);$$

hence, by the convexity of the function $y\log y$, we get $\tilde{T}(n) \leq 3(n+1)\log(n+1)$, as desired.

This finishes the proof of 4.4(b).

5.3 Proof of Theorem 4.5

(a) Apply the simple adversary strategy from Section 5.1. If at some time the leftmost path in the tree becomes longer than $\sqrt{n}$ then at least one insertion had cost $\sqrt{n}$. Otherwise, the assumption $|U| \geq 2(\sqrt{n})^{\sqrt{n}}$ guarantees, by Lemma 5.3, that the adversary strategy can be carried out. Only nodes on the leftmost path have children, hence there must be one node on the leftmost path that has at least $\sqrt{n}$ children. Thus, the cost of the last rehashing at this node was at least $\sqrt{n}$.

(b) Apply the modified adversary strategy from the proof of Theorem 4.4, for $r' = \min\{r, 2f(n)\}$. At most $n/2$ keys can be inserted below level $2f(n)$, by the overall time bound; hence at least $n/2$ will be above that level. In levels smaller than $2f(n)$, only nodes on the leftmost path can have children; as in (a) it follows that one insertion must have had cost at least $n/2f(n)$.

5.4 Proof of Theorem 4.6

5.4.1 The lower bound

Let an arbitrary algorithm for inserting $x_1, \ldots, x_n$ (into a table that initially contains one element x_0) be given. We use the simple adversary strategy from Section 5.1.

From Lemma 5.3 we know that the assumption $|U| \geq (2n/k)^k$ is sufficient to ensure that $|A(v)| \geq 2$ for v the leftmost leaf of D, and hence that the strategy is always applicable under this assumption.

For $k \geq 1$, $n \geq 1$ define $\hat{T}_k(n)$ = the minimal number of steps needed by any algorithm to insert $x_1, \ldots, x_n$ chosen according to the adversary strategy. Clearly, $T_k(n) \geq \hat{T}_k(n)$. Trivially, $\hat{T}_k(0) = 0$ for all $k \geq 1$.

Lemma 5.5 *$\hat{T}_k(n)$ satisfies the following inequalities.*

(a) $\hat{T}_1(n) = (n+1)(n+2)/2 - 1$, for $n \geq 0$.

(b) $\hat{T}_k(n) \geq \min\Big\{ l + \sum_{j=1}^{l}(ja_j + \hat{T}_{k-1}(a_j - 1)) \mid l \geq 1, a_1, \ldots, a_l \in I\!N, \sum_{j=1}^{l} a_j = n \Big\}$, for $n \geq 2$, $k \geq 2$.

Proof: (a) If $k = 1$, then every element x_i is inserted by rehashing at the root, which has cost $i+1$. Thus $\hat{T}_1(n) = \sum_{i=1}^{n}(i+1) = (n+1)(n+2)/2 - 1$.

(b) Let $x_1, \ldots, x_n$ be inserted, chosen according to the simple adversary strategy. Consider an algorithm that for inserting these elements needs $\hat{T}_k(n)$ steps. Let $x_{i_0}, x_{i_1}, \ldots, x_{i_{l-1}}$ be those elements that are inserted by global rehashing, i.e., by constructing a new perfect hash function at the root. (For x_1 this is forced, hence $i_0 = 1$.) Also, let $i_l = n+1$. Note that between global rehashings the elements $x_{i_{j-1}+1}, \ldots, x_{i_j - 1}$ are chosen so that they are all sent to the subtree rooted at the leftmost child of the root of D, and that insertions into this subtree are performed according to some strategy for $i_j - i_{j-1} - 1$ elements and depth $k-1$; further, after the insertion of $x_{i_{j-1}}$ this subtree already has one element. By the definition of $\hat{T}_k(n)$, inserting these elements into the subtree has cost at least $\hat{T}_{k-1}(i_j - i_{j-1} - 1)$. In addition, for these elements the hash function at the root has to be evaluated, which has cost $i_j - i_{j-1} - 1$. Inserting x_{i_j}, $j = 0, 1, \ldots, l-1$, has cost $i_j + 1$. Thus the total cost is

$$\begin{aligned}
\hat{T}_k(n) &\geq \sum_{j=1}^{l}(i_j - i_{j-1} - 1) + \hat{T}_{k-1}(i_j - i_{j-1} - 1)) + \sum_{j=1}^{l-1}(i_j + 1) \\
&= \sum_{j=1}^{l} \hat{T}_{k-1}(i_j - i_{j-1} - 1) + \sum_{j=1}^{l} i_j.
\end{aligned}$$

Let $a_{l+1-j} = i_j - i_{j-1}$ for $1 \leq j \leq l$. Then $\sum_{j=1}^{l} a_j = i_l - i_0 = n$, and $\sum_{j=1}^{l} i_j = l \cdot i_0 + \sum_{j=1}^{l}(l+1-j) \cdot a_{l+1-j}$; hence

$$\hat{T}_k(n) \geq \sum_{j=1}^{l}(j \cdot a_j + \hat{T}_{k-1}(a_j - 1)) + l.$$

This proves part (b). □

The proof of Theorem 4.6 is completed by the following lemma.

Lemma 5.6

$$\hat{T}_k(n) \;\geq\; g_k(n+1)$$

for all $k \geq 1$, $n \geq 0$, where

$$g_k(y) = \begin{cases} 0, & \text{for } y = 0; \\ y \ln y, & \text{for } 0 < y \leq e^k; \\ \dfrac{k}{e} \cdot y^{1+1/k}, & \text{for } e^k < y. \end{cases}$$

For the *proof* of this lemma see the Appendix. It is a technical argument based solely on the inequalities of Lemma 5.5.

5.4.2 The upper bound

We will describe an algorithm for inserting n elements $x_1, \ldots, x_n \in U$ into a table (which initially contains one element x_0) so that the depth of the resulting tree never becomes larger than k. As in the proof of Theorem 4.4(b), the hash function h_v chosen for a vertex v always satisfies $|h_v^{-1}[q] \cap A(v)| = 1$ for all $q > 0$. This means that subsequent elements that are inserted in the subtree rooted at v are always sent to the leftmost subtree of v. Let

$$d_1 = 1, \quad d_k = k \cdot \frac{k+1}{k+2} \cdot \left(\frac{d_{k-1}}{k-1}\right)^{(k-1)/k}, \quad \text{for } k > 1.$$

Then $d_k = k \cdot \left(\prod_{q=2}^{k}((q+1)/(q+2))^q\right)^{1/k}$. As an abbreviation, let $b_k = d_k/k$.

Algorithm for a table of depth at most k (Inductive description):

$k = 1$: Insert each element by global rehashing.

$k > 1$: Let $i_t = \lceil \sum_{s=1}^{t}(s/(k \cdot b_{k-1}))^{k-1} \rceil$, for $t = 0, 1, 2, 3, \ldots$. Insert the elements $x_1 = x_{i_1}, x_{i_2}, x_{i_3}, \ldots$ by global rehashing; that is, by establishing a new hash function h_v at the root v. Between these global rehashings the elements $x_{i_{t-1}+1}, \ldots, x_{i_t-1}$ all go into the leftmost subtree of the root. Apply the algorithm for depth at most $k-1$ to this subtree, for these $i_t - i_{t-1} - 1$ elements.

It is obvious that this algorithm always maintains a tree of depth at most k, hence a lookup time of k is guaranteed. We only have to analyze the time required for insertions. For $k \geq 1$, $n \geq 0$, let

$$\tilde{T}_k(n) \;=\; \text{cost of inserting } x_1, \ldots, x_n \text{ into a table, which initially has one element, using the algorithm just described.}$$

(Note that for the cost of the algorithm it is irrelevant which particular elements $x_1, \ldots, x_n$ are inserted.) To finish the proof of Theorem 4.6(b), we just have to show the following.

Lemma 5.7

(a) $\tilde{T}_k(n) \leq d_k \cdot n^{1+1/k}$ *for all* $n \geq n_k$*, for* n_k *large enough (for all* $k \geq 1$*).*

(b) $\displaystyle\lim_{k\to\infty} \frac{d_k}{k/e} = 1.$

Proof: (a) (Induction on k.)

Initial step ($k = 1$): Obviously, $\tilde{T}_1(n) = (n+1)(n+2)/2 - 1 \leq n^2$ for $n \geq 3$.

Induction step ($k > 1$): Assume $\tilde{T}_{k-1} \leq d_{k-1} \cdot n^{k/(k-1)}$ for all $n \geq n_{k-1}$. Now let n be fixed, n large enough. Define $t_0 = \min\{t \geq 1 \mid i_t > n\}$, for the sequence i_t, $t \geq 1$, defined in the strategy. We first estimate t_0. Clearly, by the definition of i_t and t_0 we have

$$\sum_{s=1}^{t_0-1} s^{k-1} \;\leq\; (k \cdot b_{k-1})^{k-1} \cdot n \;<\; \sum_{s=1}^{t_0} s^{k-1},$$

hence (by estimating the sums by integrals and taking k-th roots),

$$t_0 - 1 \;\leq\; k \cdot b_{k-1}^{(k-1)/k} \cdot n^{1/k} \;<\; t_0 + 1 \tag{3}$$

In the following, we estimate $\tilde{T}_k(i_{t_0} - 1)$, which certainly is an upper bound for $\tilde{T}_k(n)$. We let $i_0 = 0$. Then inserting the element x_{i_t} (by global rehashing) has cost $i_t + 1$, for $t = 1, 2, \ldots, t_0 - 1$; inserting the elements $x_{i_{t-1}+1}, \ldots, x_{i_t - 1}$ has cost $(i_t - i_{t-1} - 1) + \tilde{T}_{k-1}(i_t - i_{t-1} - 1)$, for $t = 1, 2, \ldots, t_0$. Thus,

$$\tilde{T}_k(n) \;\leq\; \sum_{t=1}^{t_0} ((i_t - i_{t-1} - 1) + \tilde{T}_{k-1}(i_t - i_{t-1} - 1)) + \sum_{t=1}^{t_0-1} (i_t + 1),$$

or, after a trivial transformation,

$$\tilde{T}_k(n) \;\leq\; \sum_{t=1}^{t_0} ((t_0 + 1 - t)(i_t - i_{t-1}) + \tilde{T}_{k-1}(i_t - i_{t-1} - 1)).$$

Substituting the induction hypothesis $\tilde{T}_{k-1}(n') \leq d_{k-1} \cdot (n')^{k/(k-1)}$, for $n' \geq n_{k-1}$, into this inequality yields

$$\tilde{T}_k(n) \leq \sum_{t=1}^{t_0} ((t_0 + 1 - t)(i_t - i_{t-1} - 1) + d_{k-1}(i_t - i_{t-1} - 1)^{k/(k-1)}) + \tfrac{1}{2} t_0(t_0+1) + E_k,$$

for some constant E_k (needed to make up for the error caused by replacing $\tilde{T}_{k-1}(i_t - i_{t-1} - 1)$ by $d_{k-1}(i_t - i_{t-1} - 1)^{k/(k-1)}$ for t so small that $i_t - i_{t-1} - 1 < n_{k-1}$). By the definition of i_t we clearly have $i_t - i_{t-1} - 1 \leq (t/(kb_{k-1}))^{k-1}$; furthermore, from the bounds on t_0 in (‡) it follows that $t_0^2 = O((n^{1/k})^2) = O(n)$. Thus,

$$\begin{aligned}\tilde{T}_k(n) &\leq \sum_{t=1}^{t_0}\left[\left(\frac{t}{kb_{k-1}}\right)^{k-1} \cdot (t_0 + 1 - t) + d_{k-1} \cdot \left(\frac{t}{kb_{k-1}}\right)^{k}\right] + O(n) \\ &= (kb_{k-1})^{1-k} \cdot \left(\sum_{t=1}^{t_0}(t_0+1)t^{k-1} - \sum_{t=1}^{t_0} t^k/k\right) + O(n).\end{aligned}$$

We substitute the two inequalities $\sum_{t=1}^{t_0} t^{k-1} \leq (t_0+1)^k/k$ and $\sum_{t=1}^{t_0} t^k \geq t_0^{k+1}/(k+1)$ (obtained by replacing the sums by integrals), and simplify, noting that $(t_0+1)^{k+1} = t_0^{k+1} + O(t_0^k)$. In this way we get

$$\tilde{T}_k(n) \leq \frac{1}{k+1} \cdot (kb_{k-1})^{1-k} \cdot t_0^{k+1} + O(t_0^k) + O(n).$$

By (‡), we have $t_0^k = O((n^{1/k})^k) = O(n)$ and furthermore that $t_0^{k+1} = k^{k+1} \cdot b_{k-1}^{k-1/k} \cdot n^{1+1/k} + O(t_0^k)$. Hence

$$\begin{aligned}\tilde{T}_k(n) &\leq \frac{1}{k+1} \cdot (kb_{k-1})^{1-k} \cdot k^{k+1} \cdot b_{k-1}^{k-1/k} \cdot n^{1+1/k} + O(n) \\ &= \frac{k^2}{k+1} \cdot b_{k-1}^{(k-1)/k} \cdot n^{1+1/k} + O(n).\end{aligned}$$

For n large enough, this implies

$$\tilde{T}_k(n) \leq k \cdot \frac{k+1}{k+2} \cdot b_{k-1}^{(k-1)/k} \cdot n^{1+1/k} = d_k \cdot n^{1+1/k},$$

and this is what we wanted to show.

(b) By definition, $\dfrac{d_k}{k} = \left(\prod_{q=2}^{k}\left(\dfrac{q+1}{q+2}\right)^q\right)^{1/k}$. Recall that

$$\left(\frac{q+1}{q+2}\right)^{q+2} \leq \frac{1}{e} \leq \left(\frac{q+1}{q+2}\right)^{q+1}$$

for all q, and hence

$$\frac{d_k}{k} \cdot \left(\prod_{q=2}^{k}\left(\frac{q+1}{q+2}\right)^2\right)^{1/k} \leq \left(\frac{1}{e}\right)^{(k-1)/k} \leq \frac{d_k}{k} \cdot \left(\prod_{q=2}^{k}\left(\frac{q+1}{q+2}\right)\right)^{1/k}.$$

Clearly,

$$\lim_{k\to\infty}\left(\prod_{q=2}^{k}\frac{q+1}{q+2}\right)^{1/k} = \lim_{k\to\infty}\left(\frac{3}{k+1}\right)^{1/k} = 1,$$

and thus $\lim_{k\to\infty} \dfrac{d_k}{k} = \dfrac{1}{e}$, as claimed. □

References

[AL86] Aho, H. V., and Lee, D., Storing a dynamic sparse table, *Proc. of the 27th IEEE FOCS*, 1986, pp. 55–60.

[BK88] Brassard, G., and Kannan, S., The generation of random permutations on the fly, *Information Processing Letters* **28** (1988) 207–212.

[CW79] Carter, J. L., and Wegman, M. N., Universal classes of hash functions, *J. Comput. Syst. Sci.* **18** (1979) 143–154.

[DKM88] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E., Dynamic perfect hashing: Upper and lower bounds, *Proc. of the 29th IEEE FOCS*, 1988, pp. 524–531; also: Tech. Report No. 282, Fachbereich Informatik, Universität Dortmund, 1988.

[DM89] Dietzfelbinger, M., and Meyer auf der Heide, F., An optimal parallel dictionary, *Proc. of ACM Symp. on Parallel Algorithms and Architectures*, 1989, pp. 360–368.

[DM90a] Dietzfelbinger, M., Meyer auf der Heide, F., How to distribute a dictionary in a complete network, *Proc. of the 22nd ACM STOC*, 1990, pp. 117–127.

[DM90b] Dietzfelbinger, M., and Meyer auf der Heide, F., A new universal class of hash functions, and dynamic hashing in real time, *Proc. of 17th ICALP*, Springer LNCS 443, 1990, pp. 6–19.

[FKS84] Fredman, M. L., Komlós, J., and Szemerédi, E., Storing a sparse table with $O(1)$ worst case access time, *J. ACM* **31**(3), 1984, 538–544.

[G81] Gonnet, Gaston H., Expected length of the longest probe sequence in hash code searching, *J. ACM* **28**(3) (1981) 289–304.

[M84] Mehlhorn, K., *Data Structures and Algorithms*, Vol. 1, Springer Verlag, Berlin, 1984.

[MNR90] Mehlhorn, K., Näher, S., and Rauch, M., On the complexity of a game related to the dictionary problem, *SIAM J. Comput.* **19**(5) (1990) 902–906.

[MV84] Mehlhorn, K., and Vishkin, U., Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memory, *Acta Informatica* **21** (1984) 339–374.

[S89] Siegel, A., On universal classes of fast hash functions, their time-space tradeoff, and their applications, *Proc. of the 30th IEEE FOCS*, 1989, pp. 20–25.

[T83] Troutman, J. L., *Variational calculus with elementary convexity*, Springer Verlag, New York, 1983.

[WC79] Wegman, M. N., and Carter, J. L., New classes and applications of hash functions, *Proc. of the 20th IEEE FOCS*, 1979, 175–182.

[W90] Wenzel, M., Eine Implementierung von Dynamic Perfect Hashing, Diplomarbeit, Universität des Saarlandes, 1990.

A Appendix

A.1 Proof of Lemma 5.6

We show the following: If the functions $T_k, k \geq 1$, satisfy the inequalities stated in Lemma 5.5, that is, $T_k(0) = 0$ for all $k \geq 1$, and

(a) $T_1(n) \geq (n+1)(n+2)/2 - 1$, for all $n \geq 1$,

(b) $T_k(n) \geq \min\Big\{ 1 + \sum_{j=1}^{l} (ja_j + T_{k-1}(a_j - 1)) \;\Big|\; l \geq 1, a_1, \ldots, a_l \in \mathbb{N}, \sum_{j=1}^{l} a_j = n \Big\}$,
for all $n \geq 1, k \geq 2$,

then the functions T_k satisfy the assertion of Lemma 5.6; that is,

$$T_k(n) \geq g_k(n+1)$$

for all $k \geq 1, n \geq 0$, where, for $k \geq 1$,

$$g_k(y) = \begin{cases} 0, & \text{if } y = 0; \\ y \ln y, & \text{if } 0 < y \leq e^k; \\ (k/e) \cdot y^{1+1/k}, & \text{if } e^k < y. \end{cases}$$

We proceed by induction on k. For $k = 1$, it is easily checked that $g_1(n+1) \leq (n+1)(n+2)/2 - 1$ for all $n \geq 0$. Thus, let $k > 1$, and assume the claim to be true for $k - 1$; that is, $T_{k-1}(n) \geq g_{k-1}(n+1) = g(n+1)$, for all $n \geq 0$. (From here on, we will write g for g_{k-1}.) For $n = 0$, the claim is trivially satisfied. Let $n \geq 1$ be fixed. By assumption (b) above and the induction hypothesis, we may fix some $l \geq 1$ and a sequence $a = (a_1, \ldots, a_l)$ of natural numbers with $\sum_{j=1}^{l} a_j = n$ and

$$T_k(n) - 1 \geq \sum_{j=1}^{l} (ja_j + T_{k-1}(a_j - 1)) \geq \sum_{j=1}^{l} (ja_j + g(a_j)). \tag{1}$$

We want to find a lower bound on the last sum in (1). The first step we take is to transform sums to integrals and sequences of natural numbers to real functions. The sequence a may be regarded as equivalent to the piecewise constant function $f_a\colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$ defined by

$$f_a(x) = \begin{cases} a_j, & \text{if } j - 1 \leq x < j,\ j = 1, \ldots, l, \\ 0, & \text{if } l \leq x < \infty. \end{cases}$$

The condition $\sum_{j=1}^{l} a_j = n$ translates to $\int_0^\infty f_a(x)\, dx = n$, and the sum in (1) can be expressed as

$$\sum_{j=1}^{l} (ja_j + g(a_j)) = \int_0^\infty \Big(x f_a(x) + g(f_a(x)) \Big)\, dx + \frac{n}{2}. \tag{2}$$

Our aim is now to find a lower bound on the integral in (2). To this end, we transform the minimization problem a little further: instead of piecewise constant functions such as f_a we will consider continuous functions.

Definition A.1

(a) Let $\mathcal{D}$ be the class of all continuous functions $f\colon I\!R_0^+ \to I\!R^+$ (strictly positive) so that $\int_0^\infty f(x)\,dx = n$ and so that $\lim_{x\to\infty} e^x f(x)$ exists and is positive.

(b) Let $G\colon I\!R_0^+ \times I\!R_0^+ \to I\!R$ be defined by $G(x,y) = xy + g(y) = xy + g_{k-1}(y)$.

(c) For $f \in \mathcal{D}$ let $I(f) = \int_0^\infty G(x, f(x))\,dx$.
(Note that the condition $\lim_{x\to\infty} e^x f(x) > 0$ ensures that the integral exists.)

It is easy to see that for any given $\varepsilon > 0$ the piecewise constant function f_a can be approximated by some $f_{a,\varepsilon} \in \mathcal{D}$ in such a way that

$$I(f_{a,\varepsilon}) < \varepsilon + \int_0^\infty x f_a(x) + g(f_a(x))\,dx. \tag{3}$$

Now it follows from (1), (2), (3), and the fact that $f_{a,\varepsilon} \in \mathcal{D}$ for all $\varepsilon > 0$ that

$$T_k(n) - \left(\frac{n}{2} + 1\right) \;\geq\; \inf\{\, I(f) \mid f \in \mathcal{D}\,\}. \tag{4}$$

The following proposition establishes the existence of a function $f_0 \in \mathcal{D}$ that realizes this infimum; moreover it provides an equation for f_0 that will enable us to calculate f_0 explicitly. Then we may evaluate $I(f_0)$ to obtain the desired lower bound on $T_k(n)$. The proposition is proved by reducing the problem of minimizing $I(f)$ over $\mathcal{D}$ to a standard situation treated in the Calculus of Variations. (The details of this proof, which will be given in the second part of the appendix, are irrelevant for the rest of the argument.)

Proposition A.2 *There is a unique function $f_0 \in \mathcal{D}$ so that*

$$I(f_0) = \min\{\, I(f) \mid f \in \mathcal{D}\,\}. \tag{5}$$

Moreover, there is some constant $A \in I\!R$ so that f_0 satisfies

$$\left.\frac{\partial}{\partial y} G(x,y)\right|_{y=f_0(x)} \;=\; A\,, \quad \textit{for all } x \in I\!R_0^+. \tag{6}$$

Our next goal is to use (6) in order to obtain an expression for f_0. First, we calculate A. By the definition of G, we have that $\frac{\partial}{\partial y} G(x,y) = x + g'(y)$, and hence (6) becomes

$$x + g'(f_0(x)) = A\,, \quad \text{for } x \geq 0. \tag{7}$$

It follows easily from the definition of $g = g_{k-1}$ that

$$g'(y) = \begin{cases} 1 + \ln y, & \text{if } 0 < y \le e^{k-1}, \\ (k/e) \cdot y^{1/(k-1)}, & \text{if } e^{k-1} \le y < \infty. \end{cases}$$

Obviously, $g'(y)$ is a strictly increasing function of y with range $I\!R$, and the inverse of g' is given by

$$(g')^{-1}(z) = \begin{cases} e^{z-1}, & \text{if } -\infty < z \le k, \\ (ez/k)^{k-1}, & \text{if } k \le z < \infty \end{cases} \tag{8}$$

Thus (7) can be transformed to

$$f_0(x) = (g')^{-1}(A - x), \quad \text{for } x \ge 0. \tag{9}$$

Since $f_0 \in \mathcal{D}$, we have (using the explicit formula (8) for $(g')^{-1}$):

$$n = \int_0^\infty (g')^{-1}(A - x)\, dx = \begin{cases} e^{A-1}, & \text{if } A \le k, \\ (e^{k-1}/k^k) \cdot A^k, & \text{if } A \ge k. \end{cases} \tag{10}$$

We may now solve (10) for A to obtain

$$A = \begin{cases} 1 + \ln n, & \text{if } n \le e^{k-1}, \\ (n/e^{k-1})^{1/k} \cdot k, & \text{if } n \ge e^{k-1}. \end{cases} \tag{11}$$

Now, finally, we are in a position to evaluate $I(f_0)$. First, we substitute (9) into the definition of $I(f_0)$ (see Definition A.1(b)(c)) to obtain

$$I(f_0) = \int_0^\infty x \cdot (g')^{-1}(A - x) + g((g')^{-1}(A - x))\, dx. \tag{12}$$

Case 1: $n \ge e^{k-1}$. Then $A - k \ge 0$, and we get from (12), by substituting (8) and the definition of $g = g_{k-1}$, that

$$\begin{aligned} I(f_0) &= \int_0^{A-k} x \cdot (e(A-x)/k)^{k-1} + \frac{k-1}{e} \cdot (e(A-x)/k)^k\, dx \\ &\quad + \int_{A-k}^\infty x \cdot e^{A-x-1} + e^{A-x-1} \cdot \ln(e^{A-x-1})\, dx. \end{aligned}$$

The second integral evaluates to $(A-1)e^{k-1}$, the first one equals

$$\begin{aligned} & A \cdot \left(\frac{e}{k}\right)^{k-1} \cdot \int_0^{A-k} (A-x)^{k-1}\, dx - \frac{e^{k-1}}{k^k} \cdot \int_0^{A-k} (A-x)^k\, dx \\ = \; & A \cdot e^{k-1} \cdot \frac{1}{k^k} \cdot (A^k - k^k) - \frac{e^{k-1}}{k^k} \cdot \frac{1}{k+1}(A^{k+1} - k^{k+1}) \\ = \; & A^{k+1} \cdot e^{k-1} \cdot \frac{1}{k^k} \cdot \frac{k}{k+1} - A \cdot e^{k-1} + e^{k-1} \cdot \frac{k}{k+1}. \end{aligned}$$

Altogether,

$$I(f_0) = A^{k+1} e^{k-1} \cdot \frac{1}{k^k} \cdot \frac{k}{k+1} - \frac{1}{k+1} \cdot e^{k-1}.$$

Substituting the value $A = (n/e^{k-1})^{1/k} \cdot k$ given by (11) into the last equation and using the fact that $\frac{k}{k+1} \geq e^{-1/k}$ results in

$$I(f_0) = \frac{k}{k+1} \cdot e^{-1+1/k} \cdot k \cdot n^{1+1/k} - \frac{e^{k-1}}{k+1} \geq \frac{k}{e} \cdot n^{1+1/k} - \frac{e^{k-1}}{k+1}.$$

In combination with (4) and (5) this yields

$$T_k(n) \geq \frac{k}{e} \cdot n^{1+1/k} - \frac{e^{k-1}}{k+1} + \frac{n}{2} + 1. \tag{13}$$

Elementary estimates show that the right hand side of (13) is bounded from below by $(n+1)\ln(n+1)$, if $n+1 \leq e^k$, and $(k/e) \cdot (n+1)^{1+1/k}$, if $n+1 \geq e^k$. This proves the inequality $T_k(n) \geq g_k(n+1)$ in Case 1.

Case 2: $n \leq e^{k-1}$. Then $A - k \leq 0$, and we get from (12) and (8) that

$$I(f_0) = \int_0^\infty x \cdot e^{A-x-1} + e^{A-x-1} \cdot \ln(e^{A-x-1})\, dx = (A-1) \cdot e^{A-1}.$$

We now substitute the value $A = 1 + \ln n$ from (11) to obtain that $I(f_0) = n \ln n$. In combination with (5) and (4) this entails that

$$T_k(n) \geq n \ln n + \frac{n}{2} + 1.$$

Elementary estimates show that the right hand side of this inequality is bounded from below by $(n+1)\ln(n+1) = g_k(n+1)$. This finishes the proof of Lemma 5.6. □

A.2 Proof of Proposition A.2

We sketch a proof of Proposition A.2 stated in the first part of this appendix. We reduce the proposition to a standard theorem from the Calculus of Variations. First, instead of dealing with conditions defined by the *integrals* of the functions in class $\mathcal{D}$ (see Definition A.1(a)) we need conditions on the *values* of the functions considered at the boundaries of the interval. For this, we consider the integral functions $x \mapsto \int_0^x f(\xi)\, d\xi$, for $f \in \mathcal{D}$, $x \in \mathbb{R}_0^+$. Second, we transform the unbounded interval $\mathbb{R}_0^+$ to the bounded interval $[0, 1]$ by means of the transformation $x = x(t) = -\ln(1-t)$, for $0 \leq t < 1$, with inverse transformation $t = t(x) = 1 - e^{-x}$, for $0 \leq x < \infty$.

Definition A.3

(a) *Let $\mathcal{E}$ be the class of all functions $\varphi\colon [0,1] \to \mathbb{R}_0^+$ that have a continuous derivative $\frac{d}{dt}\varphi(t) = \varphi'(t) > 0$ in $[0,1]$ and satisfy $\varphi(0) = 0$ and $\varphi(1) = n$.*

(b) *Let $H\colon [0,1] \times \mathbb{R}_0^+ \to \mathbb{R}$ be defined by*

$$H(t,z) = \begin{cases} \frac{1}{1-t} \cdot G(-\ln(1-t), (1-t)z)\,, & \text{if } z > 0 \text{ and } 0 \le t < 1\,; \\ z \ln z\,, & \text{if } z > 0 \text{ and } t = 1\,, \end{cases}$$

where $G(x,y) = xy + g(y)$ is as in Definition A.1(b).

(c) *For $\varphi \in \mathcal{E}$ let $J(\varphi) = \int_0^1 H(t, \varphi'(t))\,dt$.*
(The following lemma implies that the integral is well-defined.)

Lemma A.4 *The function H from the previous definition is continuous, and for each fixed $t \in [0,1]$ the function $z \mapsto \frac{\partial^2}{\partial z^2} H(t,z)$, $z \in \mathbb{R}$, is continuous and strictly positive, excepting for $t \neq 1$ and $y = e^{k-1}/(1-t)$.*

Proof: Straightforward verification. □

Lemma A.5 *There is a bijection between $\mathcal{D}$ and $\mathcal{E}$ given by the mappings $f \mapsto \varphi_f$ and $\varphi \mapsto f_\varphi$, where*

$$\varphi_f(t) = \begin{cases} \displaystyle\int_0^{-\ln(1-t)} f(\xi)\,d\xi\,, & \text{if } 0 \le t < 1\,; \\ n\,, & \text{if } t = 1\,, \end{cases}$$

and

$$f_\varphi(x) = \tfrac{d}{dx}\varphi(1 - e^{-x}) = \varphi'(1 - e^{-x}) \cdot e^{-x}\,, \text{ if } 0 \le x \le \infty.$$

Moreover, we have $I(f) = J(\varphi_f)$, for all $f \in \mathcal{D}$.

Proof: Straightforward verification. □

We now need the following theorem, which is obtained by combining Proposition (3.10) and Theorem (3.7) from [T83].

Theorem A.6 *If $H = H(t,z)$ is continuous on $[0,1] \times \mathbb{R}_0^+$ and if, for each $t \in [0,1]$, the function $z \mapsto \frac{\partial^2}{\partial z^2} H(t,z)$ is continuous and positive (except possibly at a finite set of z-values), then there is exactly one function $\varphi_0 \in \mathcal{E}$ that minimizes $\int_0^1 H(t, \varphi'(t))\,dt$ on $\mathcal{E}$. Moreover, this function φ_0 satisfies $\frac{\partial}{\partial z} H(t,z)\Big|_{z=\varphi_0'(t)} = \text{const}$, for $t \in [0,1]$.* □

By Lemma A.4, the function H from Definition A.3 satisfies the hypothesis of this theorem, and hence there is a unique function $\varphi_0 \in \mathcal{E}$ that minimizes $J(\varphi)$ over $\mathcal{E}$; moreover, there is some $A \in I\!R$ with $\frac{\partial}{\partial z}H(t,z)\Big|_{z=\varphi_0'(t)} = A$ for all $t \in [0,1]$. By Lemma A.5, the function $f_0 = f_{\varphi_0}$ minimizes $I(f)$ over $\mathcal{D}$. It remains to establish Equation (6).

By Definition A.3(b) we have $\frac{\partial}{\partial z}H(t,z) = \frac{\partial}{\partial y}G(-\ln(1-t),y)\Big|_{y=(1-t)z}$, and hence

$$A = \frac{\partial}{\partial z}H(t,z)\Big|_{z=\varphi_0'(t)} = \frac{\partial}{\partial y}G(-\ln(1-t),y)\Big|_{y=(1-t)\varphi_0'(t)}, \text{ for } 0 \leq t \leq 1. \tag{14}$$

By Lemma A.5 we have $f_0(x) = \frac{d}{dx}\varphi_0(1-e^{-x}) = \varphi_0'(1-e^{-x}) \cdot e^{-x} = \varphi_0'(t) \cdot (1-t)$, under the bijection $t \mapsto x(t) = -\ln(1-t)$. Hence, Equation (14) entails that $\frac{\partial}{\partial y}G(x,y)\Big|_{y=f_0(x)} = A$ for all $x \in I\!R_0^+$, as claimed. This finishes the proof of Proposition A.2.