Information Processing Letters 35 (1990) 183–189
North-Holland

7 August 1990

BOUNDED ORDERED DICTIONARIES IN O(log log N) TIME AND O(n) SPACE

Kurt MEHLHORN and Stefan NÄHER

Fachbereich 14, Universität des Saarlandes, 6600 Saarbrücken, FRG

Communicated by T. Lengauer
Received 22 August 1989
Revised 27 March 1990

In this paper we show how to implement bounded ordered dictionaries, also called bounded priority queues, in O(log log N) time per operation and O(n) space. Here n denotes the number of elements stored in the dictionary and N denotes the size of the universe. Previously, this time bound required O(N) space.

Keywords: Data structures, computational complexity, dictionary, hashing, search tree

1. Introduction

A *bounded ordered dictionary* D on $[0..N-1]$ is a partial function from $[0..N-1]$ to some set I, called the information type of D, together with the following operations:

$D.init(N)$	initializes D to the empty function
$D.defined(x)$	checks whether D is defined on x
$D.lookup(x)$	returns $D(x)$
$D.insert(x, i)$	adds x to the domain of D and sets $D(x)$ to i
$D.delete(x)$	removes x from the domain of D
$D.size(\ )$	returns $\lvert dom\ D \rvert$
$D.locate(x)$	returns the minimal $y \in dom\ D$ with $x \leqslant y$, if there is such a y, and -1 otherwise
$D.min(\ )$	returns $min\ dom\ D$
$D.max(\ )$	returns $max\ dom\ D$
$D.suc(x)$	is applicable only to $x \in dom\ D$ and returns the minimal $y \in dom\ D$ with $x < y$, if there is such a y, and -1 otherwise
$D.pred(x)$	is applicable only to $x \in dom\ D$ and returns the maximal $y \in dom\ D$ with $y < x$, if there is such a y, and -1 otherwise

In this paper we show

Theorem 1.1. *Let N be a prime. Then a bounded ordered dictionary D on $[0 \ldots N-1]$ can be implemented such that operations init, defined, lookup, size, min, max, successor and predecessor take time* O(1) *and insert, delete and locate take time* O(log log N). *The space requirement is linear in the size of D. The* O(1) *time bounds and the time bound for locate are worst case, the bounds for insert and delete are amortized and randomized, i.e., the averaging involved in the analysis is over choices made by the algorithm and not over the input sequence.*

This result extends and unifies the work of van Emde Boas et al., Van Emde Boas, Johnson, Karlsson and Willard. In [8,9,3] it is shown that the time bounds can be achieved using O(N) space and that a space

bound $O(N^\epsilon)$ can be achieved at the cost of multiplying the time bounds by $1/\epsilon$ for any $\epsilon > 0$. In [10] the space bound $O(|dom\ D|)$ is achieved for static dictionaries, and in [4,11] the space $O(|dom\ D|)$ and an $O(\sqrt{\log N})$ time bound is achieved. Many applications of dictionaries require insertions and deletions but cannot afford $O(N)$ storage. For these applications we improve the time bound from $O(\sqrt{\log N})$ to $O(\log \log N)$.

Our solution is based on van Emde Boas' stratified trees. The $O(N)$ storage requirement of stratified trees results from the use of arrays of size N. Arrays are a particularly simple implementation of *bounded dictionaries*, i.e., of partial functions on $[0..N-1]$ with operations *init, insert, delete, lookup* and *defined*, which supports all five operations in time O(1). We use dynamic perfect hashing [1] instead of arrays.

Fact [1]. *For prime N, dynamic perfect hashing realizes bounded dictionaries in constant time per operation. The time bound for init, lookup and defined is worst case, the bound for insert and delete is amortized and randomized. Furthermore the algorithm uses* $O(1)$ *amortized space per insertion and deletion, i.e., n insertions and deletions use $O(n)$ space. The model of computation is the unit-cost RAM where the arithmetic operations* $+$, $-$, $*$, mod, div *on integers of value at most N take unit time. For more details see* [1,2].

This paper is organized as follows. In Section 2 we prove the theorem above. Section 3 contains the complete programs. They are written in C++ and are part of the LEDA library [6]. Section 4 offers a short conclusion.

2. The construction

A bounded ordered dictionary D on $[0..N-1]$, N prime, is realized by a data structure consisting of four parts: the integer N, an ordered list L of the pairs $(x, D(x))$ with x in the domain of D, a bounded dictionary E from $[0..N-1]$ to the items of list L and a stratified tree of order $k_0 = \lceil \log N \rceil$ for the domain of D. The domain of E is equal to the domain of D; also for each $x \in dom\ D$, $E(x)$ is the list item containing the pair $(x, D(x))$, i.e., the position of $(x, D(x))$ in L.

It is clear that using the ordered list L and the bounded dictionary E the operations *defined*(x), *lookup*(x), *suc*(x) and *pred*(x) take constant time. Also, initialization, insertions and deletions take expected constant time when restricted to these components.

The stratified tree [8] of order k_0 is used to support the locate operation on the domain of D. In general, a stratified tree of order k supports the operations *locate*, *insert*, and *delete* on subsets of $[0..2^k-1]$ in time $O(\log k)$; $k \geqslant 1$. Our exposition of stratified trees follows [5, p. 290–296]; it is somewhat simpler however and it uses bounded dictionaries instead of arrays.

We need the following notations. For an integer $x \in [0..2^k-1]$, $k \geqslant 2$, let $x1 \in [0..2^{\lfloor k/2 \rfloor}-1]$, $x2 \in [0..2^{\lceil k/2 \rceil}-1]$ be such that $x = x1 \cdot 2^{\lceil k/2 \rceil} + x2$. A *stratified tree* of order $k \leqslant k_0$ for a set $S \subseteq [0..2^k-1]$ consists of the eight components N, k, *size*, *mi*, *ma*, D, *top* and *bot*, cf. Program 3.1. The integer variables N, k, *size*, *mi*, *ma* contain N, k, the cardinality $|S|$ of S, and the minimal and maximal element of S respectively. If $S = \emptyset$ then $mi = ma = -1$. The bounded dictionary D on $[0..N-1]$ has its domain equal to S; it is never used for lookups but only for defined queries (and hence its information type I is irrelevant). The variable *top* is a pointer to a stratified tree of order $\lceil k/2 \rceil$ and *bot* is a bounded dictionary from $[0..N-1]$ to pointers to stratified trees. If $k = 1$ or *size* $\leqslant 1$ then *top* = **nil** and *bot* is the empty dictionary. If $k \geqslant 2$ and *size* $\geqslant 1$ then *top* points to a stratified tree of order $\lceil k/2 \rceil$ for the set $S' = \{x1: x \in S\}$ and *bot* maps each $s \in S'$ to a pointer to a stratified tree of order $\lfloor k/2 \rfloor$ for the set $S''(s) = \{x2: \exists x \in S \text{ with } x1 = s\}$.

It is clear that a stratified tree of order k for the empty set can be constructed in constant time and that operations *min* and *max* take constant time, cf. Program 3.2. Let us turn to operations *locate*, *insert* and *delete* next. A program for *locate*(x) is shown in Program 3.3.

If x exceeds ma then we return -1, if $x \leqslant mi$ then we return mi and if $mi < x \leqslant ma$ and $size \leqslant 2$ then we return ma. So let us assume that $mi < x \leqslant ma$ and $size \geqslant 3$. Then $k \geqslant 2$. We first check whether $x1 \in S'$ (by calling $bot.lookup(x1)$ and comparing the result with **nil**) and, if so, checking whether $x2 \leqslant max\ S''(x1)$ (by comparing $x2$ with the ma-field of $bot.lookup(x1)$). If both conditions hold then we locate $x2$ in $S''(x1)$, say this yield z, and return $x1 \cdot 2^{\lfloor k/2 \rfloor} + z$. If one condition does not hold then we locate $x1$ in S', say this yields $z \in S'$, and return $z \cdot 2^{\lfloor k/2 \rfloor} + min\ S''(z)$. In either case, we spend constant time and generate one recursive call on a stratified tree of order $\lceil k/2 \rceil$ or $\lfloor k/2 \rfloor$. The running time is therefore O(log k).

The insertion algorithm is given in Program 3.4. After checking whether x belongs to S we distinguish cases. If $k = 1$ or $size = 0$ then we only have to update the fields mi, ma, $size$ and D. All of this takes time O(1). Calls with $k = 1$ or $size = 0$ are henceforth called trivial. So let us assume $k > 1$ and $size \geqslant 1$. If $size = 1$, let $S = \{y\}$. We create and initialize a stratified tree $T0$ of order $\lceil k/2 \rceil$ and assign it to top, create and initialize a stratified tree $T1$ of order $\lfloor k/2 \rfloor$ and insert ($x1$,pointer to $T1$) into bot. If $x1 \neq y1$ then we create and initialize another stratified tree $T2$ of order $\lfloor k/2 \rfloor$ and insert ($y1$,pointer to $T2$) into bot. All of this takes time O(1). We then insert $x1$ into $T0$ and $x2$ into $T1$. If $x1 = y1$ then we also insert $y2$ into $T1$ and if $x1 \neq y1$ then we insert $y1$ into $T0$ and $y2$ into $T2$. In either case at most one recursive call is nontrivial. Finally, if $size \geqslant 2$ we insert $x1$ into the top-structure. Also, if $x1 \notin S'$ then we create a stratified tree $T1$ of order $\lceil k/2 \rceil$, insert $x2$ into $T1$ and ($x2$,pointer to $T1$) into bot. If $x1 \in S'$ then we only insert $x2$ into the tree $bot(x1)$. We conclude as above that in either case only one of the two recursive calls of *insert* is nontrivial. Thus running time is O(log k).

Deletion is also quite simple, cf. Program 3.5. If x is not present in the stratified tree there is nothing to do. Otherwise we remove x from dictionary D and decrease the size field. We now distinguish cases. If the new size is 0 then we only have to set mi and ma to -1. If the new size is 1 then we either set mi to ma or vice versa whatever is appropriate. If $k > 1$ then we delete $x1$ and $x2$ from the substructures and deallocate the substructures. All of this takes time O(1). Finally, if the new size is at least two and hence $k > 1$ then we delete $x2$ from $S''(x1)$. If $S''(x1)$ becomes empty then we also delete $x1$ from the dictionary bot and the stratified tree top. We also recompute mi and ma in the obvious way, i.e., ma is set to the maximal element of $S''(z)$ where z is the maximal element of S'. The running time of the deletion algorithm is O(log k) since at most one recursive call is nontrivial.

We summarize the discussion in

Lemma 2.1. *Operations Insert, Delete and Locate on stratified trees take* O(log log N) *time, initialization takes time* O(1).

We discuss the space requirement next. An insertion or deletion into a stratified tree allocates space O(log log N) outside the calls to bounded dictionaries and generates O(log log N) calls to bounded dictionaries. Each such call uses O(1) amortized space. Thus a sequence of m insertions and deletions into a stratified tree runs in space O(m log log N). The standard halving-and-doubling technique allows us to relate the space requirement to the size of the set stored. Let S be a set of size n and suppose that we process a sequence of $n/2$ insertions and deletions on S. This yields a set T of size between $n/2$ and $3n/2$. We can simulate this sequence by first inserting the elements of S one by one and then processing the insertions and deletions. Then storage requirement and running time is O(n log log N). We can now start the next cycle with set T.

Lemma 2.2. *A stratified tree for a set of size n uses space* O(n log log N).

It is easy to reduce the storage requirement to O(n) as described by van Emde Boas. We only have to divide S into segments of size between log log N and 4 log log N. Segments are stored as linked lists and only the first element of each segment is stored in the stratified tree. Thus only O(n/log log N) elements

are stored in the stratified tree and hence the space bound is $O(n)$. A locate operation first uses the stratified tree to locate the appropriate segment and then completes the operation by linear search. An insertion or deletion proceeds similarly. In addition if the subsegment becomes too long or short it is split into two subsegments or merged with a neighboring sequence as in B-trees.

3. Programs

```
class stratified_tree {
int N;                                          // prime integer, universe size for
                                                // bounded dictionaries
int k;                                          // stratified tree over [0..2^k − 1]
int size;                                       // number of elements stored
int mi;                                         // minimal element stored
int ma;                                         // maximal element stored
stratified_tree * top;                          // pointer to top structure
bounded_dictionary (stratified_tree *) bot;     // a dictionary for the nonempty
                                                // bottom structures
bounded_dictionary(int) D;
};
```

Program 3.1. The components of a stratified_tree. A *bounded_dictionary*(I) is a partial function from $[0..N-1]$ to I.

```
stratified_tree :: stratified_tree(int order, int p)
//we first construct the bounded dictionaries bot and D
: bot(p), D(p)
{ // and then initialize all the variables
  N = p;
  k = order;
  size = 0;
  mi = ma = −1;
  top = nil;
}
```

Program 3.2. Initialization of a stratified_tree. The integer p must be a prime with $p \geqslant 2^{order} - 1$. In C++, construction of an object and initialization of its components have to take place simultaneously. Also the name of the constructor is the name of the class itself. Thus *stratified_tree* :: *stratified_tree* denotes the construction operation of class *stratified_tree*. Also for syntactical reasons the calls of the constructors *bot*(p) and $D(p)$ appear outside the body $\{\ldots\}$.

```
int stratified_tree :: locate(int x)
{ if (x > ma) return − 1;
  if (x ⩽ mi) return mi;
  if (size > 2)
  { int x1 = high_bits(x);
    int x2 = low_bits(x);
    stratified_tree * strptr = bot.lookup(x1);
    if ((strptr ≠ nil) && (x2 ⩽ strptr → ma))
```

```
        return x1 * 2^⌊k/2⌋ + (srtptr → locate(x2));
      else { int z = top → locate(x1);
             return z · 2^⌊k/2⌋ + bot.lookup(z) → mi;
           }
    }
    else return ma;
  }
```

Program 3.3. Operation locate

```
void stratified_tree :: insert(int x)
{   if (D.defined(x)) return;
    int x1 = high_bits(x);
    int x2 = low_bits(x);
    if (k = 1) { if (size = = 0) mi = ma = x;
                 else if (x > mi) ma = x;
                      else        mi = x;
               }
    else
    switch(size) {
    case 0: { mi = ma = x;
              break;
            }
    case 1: { int y = mi;
              if (x > mi) ma = x;
              else mi = x;
              int y1 = high_bits(y);
              int y2 = low_bits(y);
              top = new stratified_tree(⌈k/2⌉, N);
              top → insert(x1);        // a trivial call
              stratified_tree * strptr = new stratified_tree(⌊k/2⌋, N);
              strptr → insert(x2);     // a trivial call
              bot.insert(x1, strptr);
              if (x1 = = y1)
                 strptr → insert(y2);
              else { top → insert(y1);
                     strptr = new stratified_tree(⌊k/2⌋, N);
                     strptr → insert(y2);     // a trivial call
                     bot.insert(y1, strptr);
                   }
              break;
            }
    default: { if (x > ma) ma = x;
               if (x < mi) mi = x;
               top → insert(x1);
               if (!bot.defined(x1))
                  { stratified_tree * strptr = new stratified_tree(⌊k/2⌋, N);
                    strptr → insert(x2);
```

```
                bot.insert(x1, strptr);
              }
            break;
          }
    }
    size++;
    D.insert(x);
  }  // end insert
```

Program 3.4. Operation insert

```
void stratified_tree::del(int x)
{ if(!D.defined(x)) return;
  int x1 = high_bits(x);
  int x2 = low_bits(x);
  size--;
  D.del(x);
  switch(size) {
  case 0: { mi = ma = -1;
            break;
          }
  case 1: { if (x == mi) mi = ma;
            else ma = mi;
            if (k > 1)
            { top → del(x1);
              bot.lookup(x1) → del(x2);
              delete top;
              top = nil;
              bot.del(x1);
            }
            break;
          }
  default:{ bot.lookup(x1) → del(x2);
            if (bot.lookup(x1) → size( ) == 0)
            { bot.del(x1);
              top → del(x1);
            }
            int z = top → ma;
            ma = z · 2^⌊k/2⌋ + bot.lookup(z) → ma;
            z = top → mi;
            mi = z · 2^⌊k/2⌋ + bot.lookup(z) → mi;
            break;
          }
  }
}  // end del
```

Program 3.5. Operation delete

4. Conclusion

We have shown that the space requirement of the bounded ordered dictionary of van Emde Boas can be reduced from $O(N)$ to $O(n)$ while maintaining the $O(\log \log N)$ time bounds. This makes the data structure feasible even for very large N.

References

[1] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert and R. Tarjan, Upper and lower bounds for the dictionary problem, in: *Proc. 29th IEEE Symposium on Foundations of Computer Science* (1988).

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

[3] D. Johnson, A priority queue in which initialization and queue operations take time $O(\log \log D)$ time, *Math. System Theory* **15** (1982) 295–309.

[4] R.G. Karlsson, Algorithms in a restricted universe, Rept. CS-84-50, Department of Computer Science, University of Waterloo, Waterloo, Ont. (1984).

[5] K. Mehlhorn, *Data Structures and Algorithms I* (Springer, Berlin, 1984).

[6] K. Mehlhorn and St. Näher, LEDA, a library of efficient data types and algorithms, in: *Proc. MFCS '89* (1989) 88–106.

[7] P. van Emde Boas, An $O(n \log \log n)$ on-line algorithm for the insert-extract-min problem, Rept. 74, Department of Computer Science, Cornell University, Ithaca, NY (1974).

[8] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6** (1977) 80–82.

[9] P. van Emde Boas, R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977) 99–127.

[10] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, *Inform. Process. Lett.* **17** (1983) 81–89.

[11] D. Willard, New trie data structures which support very fast search operations, *J. Comput. System Sci.* **28** (1984) 395–419.