ELSEVIER Science of Computer Programming 24 (1995) 149–158

Science of Computer Programming

The matrix as in-situ data structure

Anne Kaldewaij, Laurens de Vries *

Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Received January 1994; revised August 1994
Communicated by M. Rem

Abstract

It is shown how a matrix can be used to implement a class of dictionaries. Instead of the strong requirement of ascendingness of a linear array, the weaker requirement of ascendingness of a matrix is used. This results in implementations that are efficient in both computation time and storage usage.

Keywords: Data structures; Algorithms; Slope Search; Dictionaries

1. Introduction

We consider implementations of abstract data types that support insertions, deletions, the membership test, and computation of the minimum. These operations operate on an initially empty bag (multiset) V of elements from a totally ordered set. The operations are described as follows

member(x)	$x \in V$
ins(x)	$V := V + \{x\}$
del(x)	$V := V - \{x\}$
min(x)	the minimum of V

where $\{x\}$ denotes the singleton bag containing x. When only the first three operations are specified, the abstract data type is called a *dictionary*. Binary search trees are common implementations of dictionaries (cf. [3]). Without the membership test and with deletions restricted to the minimum of V, the abstract data type is called a priority

* Corresponding author. Email: laurens@win.tue.nl.

SSDI 0167-6423(94)00032-8

queue. The well-known heap can be used as an implementation of the priority queue. The heap is usually implemented as an *in-situ data structure*: a single array is used to store the elements of V, and no pointers or other auxiliary arrays are used. The size of the array determines the number of elements that V may have. The disadvantage of such a heap is that an efficient implementation for the membership test does not exist.

Yet another in-situ representation of a bag is an ascending array. Computation of the minimum is an $O(1)$ operation and for the membership test the binary search yields an $O(\log n)$ program for a bag with n elements. Insertions and deletions, however, have $O(n)$ time complexity.

In this paper we show how the data type described above can be implemented as an in-situ data structure, using a single two-dimensional array (a matrix). For a bag of n elements the following time complexities are obtained:

member(x)	$O(\sqrt{n})$
ins(x)	$O(\sqrt{n})$
del(x)	$O(\sqrt{n})$
min(x)	$O(1)$

Overview. Preliminaries are presented in Section 2. In Section 3 and Section 4 programs for insertion and deletion are derived. These operations affect the part of the matrix that is used to store the elements of V. How the shape of this part of the matrix can be controlled is the topic of Section 5. In Section 6, we show how the solutions can be transformed in such a way that the elements of the bag are stored in an initial segment of a linear array.

2. Preliminaries

We use array $m[1..N, 1..N]$ to hold the values of bag V. Hence, N^2 is an upper bound for the size of V. To obtain efficient programs for the operations on V, we decide to maintain matrix m ascending in both arguments. The ascendingness of m will be an invariant of all operations. For an ascending matrix, the searching paradigm known as *Saddleback Search*, cf. [1], or as *Slope Search*, cf. [2], is applicable. This technique (for details refer to [2, Section 8.10]) yields for the membership test:

$p, q, b := 1, N, \text{false};$
do $\neg b \wedge p \neq N+1 \wedge q \neq 0$
  $\rightarrow$ if $m[p,q] < x \rightarrow p := p+1$
    [] $m[p,q] > x \rightarrow q := q-1$
    [] $m[p,q] = x \rightarrow b := \text{true}$
    fi
od

This program has post-conditions

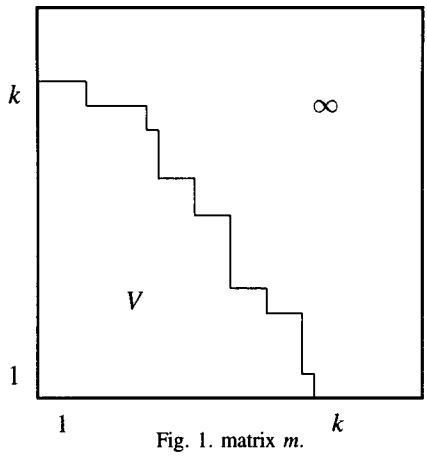

Fig. 1. matrix m.

$$b \equiv (\exists i,j:\ 1 \leqslant i \leqslant N \wedge 1 \leqslant j \leqslant N:\ m[i,j] = x) \quad \text{and}$$
$$b \Rightarrow m[p,q] = x$$

The repetition performs at most $2N$ steps. Hence, the time complexity is linear to the square root of the size of the matrix. Since we aim at a time complexity related to n, the number of elements of V, we will see to it that all elements of V are located within a $k \times k$ part of the matrix, where k is approximately the square root of n. As a result, only this $k \times k$ square has to be searched. This second design decision is implemented as follows.

The elements of matrix m that do not hold a value of bag V will have value ∞ (cf. Fig. 1). The values of m different from ∞, containing the values of V, are within a $k \times k$ square. In Section 5 we show how insertion and deletion can be implemented in such a way that the shape of the set of matrix elements contributing to V is "as square as possible".

The fact that m is ascending in both arguments, can be expressed as

$$(\forall p,q:\ 1 \leqslant p \wedge 1 \leqslant q:\ m[p-1,q] \uparrow m[p,q-1] \leqslant m[p,q] \leqslant m[p+1,q] \downarrow m[p,q+1])$$

where $a \uparrow b$ and $a \downarrow b$ denote the maximum and minimum of a and b. To avoid problems with the boundary of m, we define $m[0,q]$ and $m[p,0]$ as $-\infty$; similarly, $m[i,q]$ and $m[p,i]$ are defined ∞ for $i > N$.

Assignment $m[p,q] := x$ does not violate the ascendingness of m if and only if

$$m[p-1,q] \uparrow m[p,q-1] \leqslant x \leqslant m[p+1,q] \downarrow m[p,q+1]$$

In particular, for a pair (u,v) satisfying

$$m[u,v] = m[p-1,q] \uparrow m[p,q-1] \quad \text{or}$$
$$m[u,v] = m[p+1,q] \downarrow m[p,q+1]$$

assignment $m[p,q] := m[u,v]$ does not violate the ascendingness of m.

3. Insertion

Insertion of a value will cause the change of a matrix element from ∞ to a value different from ∞. Let us denote the set of matrix elements different from ∞ by U. Since the shape of U should be as square as possible, we decide the following:

> An insertion will cause a change from ∞ to a value different from ∞ for precisely one element of the matrix. The choice for this element will depend only on the shape of U; it is independent of the value that is inserted.

Matrix element (a, b) to be used for the next insertion will be at the border of U. More precisely, it will satisfy predicate *Ins* defined as

$$Ins: \quad m[a,b] = \infty \ \wedge \ m[a-1,b] \neq \infty \ \wedge \ m[a,b-1] \neq \infty$$

How this invariant is maintained by insertions and deletions is shown in Section 5. For the moment, we assume that program variables a and b satisfy *Ins*. To change $m[a,b]$ from ∞ to a value different from it, assignment

$$m[a,b] := m[a-1,b] \uparrow m[a,b-1]$$

is appropriate. As observed in the previous section, such an assignment does not violate the ascendingness of m. It establishes

$$V = V' + \{m[a,b]\}$$

where V' denotes the initial value of V. The post-condition of ins(x) is

$$V = V' + \{x\}$$

On account of these observations, we introduce program variables p and q with P_0, as accompanying invariant for a repetition, defined by

$$P_0: \quad V = V' + \{m[p,q]\}$$

Initialization is $p, q := a, b$. Post-condition $V = V' + \{x\}$ is established by $m[p,q] := x$. For the invariance of m's ascendingness, the pre-condition of this assignment should satisfy

$$m[p-1,q] \uparrow m[p,q-1] \leqslant x \leqslant m[p+1,q] \downarrow m[p,q+1]$$

Since $m[a,b+1] = \infty$ and $m[a+1,b] = \infty$, initially $x \leqslant m[p+1,q] \downarrow m[p,q+1]$ holds, which we add as a second invariant of the repetition:

$$P_1: \quad x \leqslant m[p+1,q] \downarrow m[p,q+1]$$

As guard of the repetition, we then have

$$m[p-1,q] \uparrow m[p,q-1] > x$$

For the sake of convenience, we introduce variables u and v with accompanying invariants

P_2 : $(u,v)=(p,q-1) \vee (u,v)=(p-1,q)$

P_3 : $m[u,v] = m[p-1,q] \uparrow m[p,q-1]$

Using P_3, the guard of the repetition can be replaced by $m[u,v] > x$. It implies, due to the ascendingness of m, $x \leqslant m[u+1,v] \downarrow m[u,v+1]$, i.e., $P_1(p,q := u,v)$. To ensure $P_0(p,q := u,v)$ as well, statement $m[p,q] := m[u,v]$ should precede assignment $p,q := u,v$. This yields the following program for ins(x) :

$p,q := a,b$;
if $m[p,q-1] \leqslant m[p-1,q] \rightarrow u,v := p-1,q$
$[]$ $m[p-1,q] \leqslant m[p,q-1] \rightarrow u,v := p,q-1$
fi;
$m[p,q] := m[u,v]$;
$\{P_0 \wedge P_1 \wedge P_2 \wedge P_3\}$
do $m[u,v] > x$
$\rightarrow$ $m[p,q] := m[u,v]$; $p,q := u,v$;
if $m[p,q-1] \leqslant m[p-1,q] \rightarrow u := u-1$
$[]$ $m[p-1,q] \leqslant m[p,q-1] \rightarrow v := v-1$
fi
od;
$m[p,q] := x$

In each step of the repetition $u+v$ decreases by 1, hence, execution of this repetition takes at most $a+b$ steps. When the values of V are within a $k \times k$ square, this is at most $2k$.

Note that the first assignment $m[p,q] := m[u,v]$ is superfluous (it is only used to avoid complicated invariants) and can be removed.

4. Deletion

Deletion of an element of V will decrease the number of elements different from ∞ by one. Since the shape of U should be as square as possible, we decide the following (cf. insertion):

> A deletion will cause a change from a value different from ∞ to ∞ for precisely one element of the matrix. The choice for this element will depend only on the shape of U; it is independent of the value that is removed.

Matrix element (c,d) to be used for the next deletion will be close to the border of U. More precisely, it will satisfy predicate *Del*, defined as

Del: $m[c,d] \neq \infty \wedge m[c+1,d] = \infty \wedge m[c,d+1] = \infty$

How this invariant is maintained by insertions and deletions is shown in Section 5. For the moment, we assume that program variables c and d satisfy *Del*. The assignment

$m[c,d] := \infty$ does not violate the ascendingness of the matrix. Furthermore, it satisfies

$$\{\ V = A \wedge m[c,d] = z\ \}\ m[c,d] := \infty\ \{\ V = A - \{z\}\ \}$$

Let x be the element of V to be removed. If we take $A = V' - \{x\} + \{z\}$ in the above Hoare-triple, then the post-condition simplifies to $V = V' - \{x\}$, which is to be established by del(x).

The remaining problem is now to establish as precondition for $m[c,d] := \infty$:

$$V = V' - \{x\} + \{z\} \ \wedge\ m[c,d] = z$$

This condition expresses formally that z is added to V at the expense of x. It is clear that this requires an assignment of the form

$$m[p,q] := z$$

for some p and q. The corresponding pre-condition of this assignment is

(1) $V = V' - \{x\} + \{m[p,q]\} \ \wedge\ m[c,d] = z$

The ascendingness of m imposes as additional precondition

(2) $m[p-1,q] \uparrow m[p,q-1] \leqslant z \leqslant m[p+1,q] \downarrow m[p,q+1]$

Condition (1) can easily be satisfied using the member operation, to establish $m[p,q] = x$, followed by the assignment $z := m[c,d]$. This suggests the introduction of a repetition with (1) as invariant and a guard whose negation implies (2). This approach is similar to the one used in the ins operation. As a matter of fact, ins can be viewed as a special case of this problem, where $x = \infty$. There is one important difference, however. The values of x and z are not related in any way. So, it is not clear whether p and q must be increased or decreased in order to obtain (2). To make the direction of change of p and q predetermined, we introduce a case analysis. In case $z \leqslant x$, we have $z \leqslant m[p,q]$ as additional invariant of a repetition in which p and q are decreased, as in the ins program. In case $z \geqslant x$, we have $z \geqslant m[p,q]$ as additional invariant of a symmetric repetition in which p and q are increased. This results in the following program for del(x):

```
"compute (p,q), such that m[p,q]=x, using membership";
z := m[c,d];
if z ⩽ x
    → if m[p,q−1] ⩽ m[p−1,q] → u,v := p−1,q
      [] m[p−1,q] ⩽ m[p,q−1] → u,v := p,q−1
      fi;
      do m[u,v] > z
          → m[p,q] := m[u,v]; p,q := u,v;
            if m[p,q−1] ⩽ m[p−1,q] → u := u−1
            [] m[p−1,q] ⩽ m[p,q−1] → v := v−1
```

```
                fi
        od
[] z ⩾ x
    → if m[p,q+1] ⩾ m[p+1,q] → u,v := p+1,q
      [] m[p+1,q] ⩾ m[p,q+1] → u,v := p,q+1
      fi;
      do m[u,v] < z
          → m[p,q] := m[u,v]; p,q := u,v;
            if m[p,q+1] ⩾ m[p+1,q] → u := u+1
            [] m[p+1,q] ⩾ m[p,q+1] → v := v+1
            fi
      od
fi;
m[p,q] := z;
m[c,d] := ∞
```

5. The shape of U

With respect to insertion, the effect on the set U of elements of m different from ∞ is $U := U \cup \{(a,b)\}$, where (a,b) satisfies

Ins: $m[a,b] = \infty \wedge m[a-1,b] \neq \infty \wedge m[a,b-1] \neq \infty$

With respect to deletion, the effect on the set U is $U := U \setminus \{(c,d)\}$, where (c,d) satisfies

Del: $m[c,d] \neq \infty \wedge m[c+1,d] = \infty \wedge m[c,d+1] = \infty$

We let set U depend on the size n of V only. Hence, we try to find a function f: $[1..\infty) \rightarrow [1..\infty) \times [1..\infty)$, such that

$$U = \{f(i) \mid 1 \leqslant i \leqslant n\}$$

Element $f(n)$ of matrix m is added by an insertion into a bag of $n-1$ elements; it is removed by a deletion from a bag of n elements. Hence, as additional invariants, we have

n = the size of V,
$(a,b) = f(n+1)$,
$n > 0 \Rightarrow (c,d) = f(n)$

To procedure ins(x) the following statements are added at the end

```
c,d := a,b;
(a,b) := f(n+2);
```

11	.	.	.	.
7	12	.	.	.
4	8	13	.	.
2	5	9	14	.
1	3	6	10	15

(a)

17	19	21	23	25
10	12	14	16	24
5	7	9	15	22
2	4	8	13	20
1	3	6	11	18

(b)

17	18	19	20	25
10	11	12	16	24
5	6	9	15	23
2	4	8	14	22
1	3	7	13	21

(c)

Fig. 2. Three suitable enumerations of the cells of the matrix.

$n := n+1$

Procedure del(x) is extended at the end by

$a, b := c, d$;
if $n > 1 \rightarrow (c, d) := f(n-1)$
[] $n = 1 \rightarrow$ skip
fi;
$n := n-1$

Fig. 2 shows three suitable functions which satisfy the requirement that the shape of U should be within a small square. We consider the enumeration of Fig. 2(b), for which we have

$$f(1) = (1,1)$$

$$f(n+1) = \begin{cases} (b,a) & \text{if } a < b \\ (1,b+1) & \text{if } a = b, \text{ where } (a,b) = f(n) \\ (b+1,a) & \text{if } a > b \end{cases}$$

It is also possible to express $f(n-1)$ in terms of $f(n)$:

$$f(1) = (1,1)$$

$$f(n-1) = \begin{cases} (d-1,d-1) & \text{if } c=1 \wedge d>1 \\ (d,c-1) & \text{if } c>1 \wedge c \leqslant d \text{ where } (c,d) = f(n) \\ (d,c) & \text{if } c>d \end{cases}$$

With this choice of f, U is within a square with sides $c \uparrow d$, which is equal to $\lceil \sqrt{n} \rceil$. Hence, for the membership test (cf. the program in Section 2), only $m[1..k, 1..k]$ plays a role, where $k = c \uparrow d$.

The time complexity of this implementation is in some sense optimal with respect to the class of implicit data structures that are based on a fixed partial order on the storage locations. Let T_s be the maximum number of steps needed to search for a given element and let T_u be the maximum number of steps needed for any update to the data structure. Then, it is possible to prove that $T_s \times T_u$ is at least n (cf. [4]). So, this implementation provides a good balance between the worst-case time needed for searching and the worst-case time needed for changing the data structure.

This concludes the implementations of the operations. Note that as initialization, all matrix elements will be set to ∞. In the next section, we show how set U can be mapped to a initial segment of a linear array. Initialisation will then be a constant time operation (a single assignment). This transformation shows how inspection of pairs (p,q) where $m[p,q] = \infty$ can be avoided. Furthermore, in the linear version, only storage needed to represent V is needed.

6. Linearization

Instead of matrix m, a linear array $h[1..N^2]$ can be used to store the values of V, in such a way that

$$V = \{h[i] \mid 1 \leqslant i \leqslant n\}.$$

As with the heap, the elements of V are stored in a prefix of h. Since f maps $[1..n]$ to the set of matrix elements used for a bag of n elements, we choose the inverse f^{-1} to map positions in the matrix to positions in h. For $1 \leqslant i \wedge 1 \leqslant j \wedge (i,j) \notin U$ we have $f^{-1}(i,j) > n$. Since $m[i,0]$ and $m[0,j]$ are defined as $-\infty$, we have the following correspondence between matrix m and array h:

$$m[i,j] = \begin{cases} -\infty & \text{if } i = 0 \vee j = 0, \\ h[f^{-1}(i,j)] & \text{if } 1 \leqslant i \wedge 1 \leqslant j \wedge f^{-1}(i,j) \leqslant n, \\ \infty & \text{if } 1 \leqslant i \wedge 1 \leqslant j \wedge f^{-1}(i,j) > n. \end{cases}$$

It is not difficult to obtain an explicit expression for f^{-1}. For the enumeration of Fig. 2(b), we have

$$f^{-1}(i,j) = \begin{cases} (j-1)^2 + 2i - 1 & \text{if } i \leqslant j, \\ (i-1)^2 + 2j & \text{if } i > j. \end{cases}$$

To relate the f^{-1}-values of $(i{+}1,j)$, $(i,j{+}1)$, $(i{-}1,j)$, and $(i,j{-}1)$ to the f^{-1}-value of (i,j), one can use the following relations, which follow from this expression. For $f^{-1}(i,j) = k$, we have

$f^{-1}(i{+}1,j)$	$f^{-1}(i,j{+}1)$	$f^{-1}(i{-}1,j)$	$f^{-1}(i,j{-}1)$	case
$k+2$	$k+2j-1$	$k-2$	$k-2j+3$	$i<j$
$k+2i$	$k+2j-1$	$k-2$	$k-1$	$i=j$
$k+2i-1$	$k+1$	$k-2i+2$	$k-2$	$i=j+1$
$k+2i-1$	$k+2$	$k-2i+3$	$k-2$	$i>j+1$

A complete transformation of membership, insert and delete in terms of h is left to the reader. Note that the initialization (V empty) is $n := 0$ and that $h[1]$ is the minimum of V.

7. Concluding remarks

We have shown how dictionaries and priority queue operations can be implemented as an in-situ data structure, using an ascending matrix. The derivation of the programs for the operations in terms of a matrix is relatively easy. Once the programs have been derived, a linear array representation can be obtained using a transformation function.

The choice of the enumeration of the matrix elements defines the transformation function. If one chooses the enumeration of Fig. 2(a), a data structure is obtained that is similar to the *beap* (biparental heap) of [4]. Note that this choice uses only one half of the matrix. In [4], the beap is also used as basis for a more sophisticated implicit data structure with $O(n^{1/3} \log n)$ execution time. This implementation, however, requires that all elements are distinct.

References

[1] D. Gries, *The Science of Programming*, Texts and Monographs in Computer Science (Springer, New York, 1981).

[2] A. Kaldewaij, *Programming: The Derivation of Algorithms*, Prentice Hall International Series in Computer Science (Prentice Hall, London, 1990).

[3] K. Mehlhorn, *Data structures and algorithms 1: Sorting and Searching* (Springer, Berlin, 1984).

[4] J.I. Munro and H. Suwanda, Implicit data structures for fast search and update, *J. Comput. System Sci.* **21** (1980) 236–250.