PADDED LISTS: SET OPERATIONS IN EXPECTED θ(log log N) TIME *

W. Randolph FRANKLIN
Electrical and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY 12181, U.S.A.

Received 6 August 1979

Padded list, sorting, retrieval, searching, expected time, data structure

1. Introduction

An important problem in database processing is the manipulation of a set of N records with keys chosen from a key space of cardinality much greater than N. This means that a bit map of the set is infeasible. Some typical operations include the following:

(1) finding the record with a given key, if it exists;

(2) finding the record with the next higher (or lower) key than a given record, whose location we may or may not already know;

(3) finding all the records with keys in a certain range;

(4) adding a record to the set and

(5) deleting a record from the set.

The data structure that we choose for our operation must efficiently perform those of the above operations that we need. The times that various data structures take is shown in Table 1. However, there are also other factors to be considered:

(1) if the size of the set is changing, some structures, such as trees, can adapt, while others, such as hash tables and sorted lists, must be reformatted;

(2) some structures, such as hash tables, have average times that are very much better than their worst case times;

(3) some structures, such as trees, require a lot of extra space for auxiliary information, like pointers.

These data structures are summarized in various books such as [1], [2] and [4].

This paper present a new data structure, the *padded list*, that does all the above operations in expected time $T = \theta(\log \log N)$ if the record keys are drawn independently from a known probability distribution. However, if the keys are selected by an adversary, the worst case time for padded lists, like hash tables, is $T = \theta(N)$.

Padded lists, like hash tables, allow a tradeoff between time and space, but a reasonable amount of extra storage used is $\frac{1}{5}$ word per record. This compares favorably with trees that use one word per record for a pointer. The times for padded lists in Table 1 include the cost of periodic reformatting, so they are well adapted to sets that change greatly in size. Finally, padded lists are much easier to implement than structures such as trees which need complicated balancing. Thus padded lists are the best data structure for sets where keys are independent.

2. Data structure

2.1. Initial creation

Assume N records, R_1 to R_N, that we wish to arrange into a padded list. Let k be a small natural number constant. It affects the time-space tradeoff. Let $M = N(1 + 1/k)$. The data structure consists of 2 arrays, A of M words, and B of M bits. Their subscripts range from 1 to M.

* This research was initially prepared for National Science Foundation, Engineering Division, Automation, Bioengineering and Sensing Systems under Grant No. ENG 79-08139.

Table 1
Expected time to perform operations on different data structures

Data structure operation	Hash table	Tree	Sorted list [a,b]	Padded list
Find record	1	log N	log N	log log N
Find next higher record, given location of record	N	log N	1	1
Find next higher record, not given location	N	log N	log N	log log N
Find the P records in given range	P + N	P + log N	P + log N	P + log log N
Add record	1	log N	N	log log N
Delete record	1	log N	log N	log log N

a) In the sorted list, the record is deleted by flagging it. Physical deletion would take time N.
b) The sorted list is assumed to use binary search. Interpolation search would make the log N to be log log N in every case.

Now these arrays A and B are logical arrays that are physically realized as circular buffers A^1 and B^1. The purpose is to allow elements to be assigned from the beginning to the end of A and B as needed. Although the initial mapping is $A_i \leftrightarrow A_i^1$, after many additions to the padded list, A might have rotated two words, for instance to give the mapping

$A_i \leftrightarrow A_{i+2}^1$ for $1 \leqslant i \leqslant M-2$,

$A_i \leftrightarrow A_{i-M+2}^1$ for $i = M-1, M$.

Thus the physical realization would be: $A_{M-1}, A_M, A_1, A_2, \ldots, A_{M-3}, A_{M-2}$. Certain conservation properties are always true of A and B:

(1) $A_1 \leqslant A_2 \leqslant A_3 \leqslant \cdots \leqslant A_M$. This is true for all A_i, whether or not $B_i = 1$;

(2) if the padded list has N records currently, then exactly N of the M $B_i = 1$. The corresponding A_i are the records. If $B_i = 0$, the corresponding A_i is a vacancy that is there as the result of a deletion, or to allow for future additions.

At the initial creation and after each reformatting, one vacancy is left after every k records. That is, $B_i = 0$ for $i = k+1, 2k+2, 3k+3, \ldots$. The A_i for which $B_i = 0$ can be set to A_{i-1} in each case.

Note that various implementation tricks are possible, such as storing B in the sign bits of A, if the keys are positive.

Example 1. Suppose N = 12. Let the set of records be R = 31, 41, 59, 26, 98, 69, 60, 44, 01, 17, 81. Let k = 3 so M = 16. Then the initial padded list is as shown in Table 2. This will be used in future examples.

2.2. Locating a record

Assume that we wish to locate the record R in the padded list, if it is there. The method is:

(1) use an interpolation search [3] on A to find an $A_i = R$. If none exists, then R is not in the list;

(2) if $B_i = 1$, then return i. If not, then we are at a vacancy in A. Find the largest $j < i$ where $B_j = 1$. If $A_j = R$, then return j. Else R is not present;

(3) if there is no $i \leqslant j$ with $B_j = 1$, then R does not exist.

Example 2. If R = 26, then the first probe is at $i = 16 \times \frac{26}{98} = 4$. $A_4 = 26$, but $B_4 = 0$, so we try i = 3.

Table 2

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A_i	01	17	26	26	31	41	44	44	54	59	60	60	69	81	98	98
B_i	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0

This gives $A_3 = 26$ and $B_3 = 1$, so we return 3.

2.3. Locating the next higher record

To find the next higher record, first locate the record as described above, if necessary. Assume it is A_i. Find the smallest $j > i$ with $B_j = 1$ and return A_j. However if we run off the end of A first, then there is no next higher record. The next lower record may be found similarly.

2.4. Locating all records in a range

Locate the lower endpoint record in A. If it is not in A, locate the next larger record. Return records from A, increasing from this record, until the upper end of the range is reached.

2.5. Adding a record

Assume that we wish to add record R to A. Use the interpolation search to find i such that $A_i \leqslant R < A_{i+1}$. If $A_i = R$, then if $B_i = 1$, then R is already in A. Otherwise if $A_i \leqslant R$ and $B_i = 0$, then we have a fortuitous vacancy in A at the right place, and can set $A_i = R$ and $B_i = 1$. Otherwise, we have some shuffling to do. Find the smallest $j > i$ such that $B_j = 0$. Shift A_i to A_{j-1} one up to occupy A_{i+1} to A_j. Do the same with B_i to B_{j-1}. Then if $B_{j+1} = 0$ and $A_{j+1} < A_j$, set $A_{j+1} = A_j$. Repeat for B_{j+2}, B_{j+3}, ... so long as they are 0. This is to keep A in order. Finally, set $A_i = R$ and $B_i = 1$.

It may be that we will run off the top of B before finding $B_i = 0$. In that case, find the smallest $j \geqslant 1$ with $B_j = 0$, move B_1 to B_{j-1} up to B_2 to B_j, shift the start of A and B one up (remember they are circular buffers) so that now A_M is free, shift A_i to A_{M-1} up to A_{i+1} to A_M and similarly for B, and set $A_i = R$ and $B_i = 1$.

2.5.1. Reformating

After βN additions have been done for some fixed β, since the last reformatting, reformat the list so that it has evenly spaced vacancies as described in the section on initial creation. If N is different from when the list was created, then a different amount of storage will be required. Extra storage can be allocated at the end of A and B, if it is available, or else they can be moved to a completely new location as they are reformatted. If they are reformatted in place, then first move all the records in A (for which $B_i = 1$) to the bottom of A, and then space them out. This moves each record twice. It is possible to equalize the spaces in A by only moving each record once, but the method is more complicated, and hence, probably slower and more error-prone.

If the padded list is large and in virtual memory, we have much more flexibility. If the next higher page above A is not in use, we can simply use it. Otherwise, we can allocate new space in virtual memory, transfer A and B, and stop using the old pages so that they are swapped out and effectively cease to exist. On some systems we can explicitly deallocate them.

2.5.2. Examples

Suppose that we wish to add R = 75. The interpolation search yields $A_{13} = 69$, $B_{13} = 1$. However $B_{16} = 0$, so we push A_{14} and A_{15} up one and insert 75 (see Table 3).

Suppose that we next wish to add R = 99. There are no vacancies after $A_{16} = 98$, so we free a word at the bottom of A, rotate the circular buffer by one, and insert 99 (see Table 4).

Suppose that we now wish to reformat the array in place. Since, now N = 14, if k = 3 still, we need M = 18, so there must be some extra space at the end of A^1 and B^1. Note that extra space at the end of the physical arrays A^1 and B^1 means an extra block of space somewhere in the middle of the logical arrays A and B, if they have been rotated. This does not affect anything. In this case, the two extra words will appear between A_{15} and A_{16} (see Table 5). Then we shift the nonempty elements of A down to give:

A: 01, 17, 26, 31, 41, 44, 54, 59,
60, 69, 75, 81, 98, 99, –, –, –, –.

Finally we shift them up, leaving a blank after every

Table 3

i	13	14	15	16
A_i	69	75	81	98
B_i	1	1	1	1

Table 4

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A_i	01	17	26	31	41	44	44	54	59	60	60	69	75	81	98	99
B_i	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1

Table 5

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A_i	01	17	26	31	41	44	44	54	59	60	60	69	75	81	98	–	–	99
B_i	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	–	–	1

Table 6

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A_i	01	17	26	26	31	41	44	44	54	60	69	69	75	81	98	98	99	99
B_i	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	0

third (see Table 6). Here, $N(1 + 1/k)$ is not an integer, so the ceiling was taken. The floor would have also been acceptable.

2.6. Deleting a record

Assume we wish to delete R. Locate it as described above, at $A_i = R$. Set $B_i = 0$. After δN deletions, since the last reformatting for some fixed δ, reformat the padded list as described above. Again, if N is different, it may be desirable to move the list to a different location.

3. Timing

The worst case times to locate or add a record are clearly $T = \theta(N)$. Thus this algorithm is only useful when it is the expected time that is important.

These expected times assume that the record keys are drawn independently from a known statistical distribution. If it is not a uniform distribution, then the keys can be transformed to make it so.

3.1. Initial creation

Initializing the M locations of A and B takes

$$T = \theta(M) = \theta\left(N\left(1 + \frac{1}{k}\right)\right) = \theta(N)$$

since k is constant.

3.2. Locating a record

To locate a record, R, first we search for an $A_i = R$. If $B_i = 1$, we have it, but otherwise we decrement i until $B_i = 1$. The interpolation search takes $T_1 = \theta(\log \log N)$. There are not more than $M - N + \delta N = N(1/k + \delta)$ vacancies in A, so the expected time to find $B_i = 1$ is

$$T_2 = \theta\left(\frac{1}{k} + \delta\right) = \theta(1) .$$

Thus the time to locate a record is $T = T_1 + T_2$:

$$T = T_1 + T_2 = \theta(\log \log N) .$$

3.3. Locating the next higher record

The time here is clearly $T = \theta(\log \log N)$.

3.4. Locating all records in a range

Assume that there are P records in the range, $0 \leqslant P \leqslant N$. It takes $\theta(\log \log N)$ to find the lower bound, and $\theta(P)$ to return all the records, for a total

$$T = \theta(P + \log \log N) .$$

3.5. Adding a record

This is the difficult case; that it can be done efficiently is the central result of this paper. We will consider the case where no deletions have been made since the last reformatting. If there have been some deletions, then the addition will be even faster since there will be fewer records in array A.

Initially we have an array A with N/k vacancies separated by runs of k contiguous records. When we add a record R that falls in a vacancy, there is nothing else to do. On the other hand, if R falls in a run of occupied records, we must shift up from 1 to k records to make room for R. In either case, a vacancy is used up.

After several records have been added it may occur that what were formerly the first P vacancies above R's location are full. Then we have to use the $(P + 1)^{st}$ vacancy, and shift from $Pk + 1$ to $(P + 1)k$ records in A. Thus the time to insert R depends on the average length of the string of occupied former vacancies after R. Since R, by hypothesis, is independently uniformly distributed, these vacancies will be filled randomly. The k records between each original vacancy do not affect the analysis of the problem; they only add a constant factor to the time.

Now the whole process of finding a vacancy is identical to the process of finding an overflow record in hashing with linear probing [2, program L, pp.518–521]. The equivalent size of the hash table is $M - N$. The number of records in the table so far is at most βN so Knuth's load factor, α, is

$$\alpha \leqslant \frac{\beta N}{M - N} = \beta k .$$

However, we are not probing to find a matching hash key, but to find a vacancy. This is equivalent to an unsuccessful hash probe. Thus the average number of probes needed is

$$C \approx \frac{1}{2}\left(1 + \left(\frac{1}{1 - \alpha}\right)^2\right)$$

$$\approx 1 + \alpha + \tfrac{3}{2}\alpha^2 + 2\alpha^3 + \cdots .$$

For example, if $k = 5$ and $\beta = 0.1$, then the first free vacancy is the $C \approx 2\frac{1}{2}^{th}$ one, each of which requires shifting $k + 1 = 6$ records. Thus the average addition in this example requires shifting 15 records which is quite fast with a block move operation. Since β and k are constants, $C = \theta(1)$. Thus, in general the expected work to add a record without reformatting of $T_1 = \theta(\log \log N)$, for the initial find $+\theta(1)$, to find a vacancy $= \theta(\log \log N)$.

Now the reformatting takes time $\theta(N)$ once every βN additions for an average time of $T_2 = \theta(1)$. Thus the total expected time to add a record is

$$T = T_1 + T_2 = \theta(\log \log N) .$$

3.6. Deleting a record

To delete R, we locate $A_i = R$ and set $B_i = 0$. This takes time $T = \theta(\log \log N)$.

4. Future extensions

One problem with padded lists is that in actual cases, series of related operations are often performed on a data structure. A series of additions to the same place may take up to $T = \theta(L)$ for the L^{th} addition. This can be ameliorated by reformatting more often in these cases. If the correlation between successive keys is known, future research might tell how to modify the data structure accordingly.

5. Summary

With periodic reformatting and preplanned gaps, a simple array becomes a padded list that supports the operations of location, adjacent record location, addition, and deletion, in expected time $T = \theta(\log \log N)$. The extra space required is only a fraction of a word per record.

Acknowledgment

This research was supported by the National Science Foundation, Automation, Bioengineering, and Sensing Systems Program, under Grant ENG 79-08139, and by the RPI Engineering Build Program. Their support is gratefully acknowledged.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974).

[2] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973).

[3] Y. Perl, A. Itai and H. Avni, Interpolation search – a log log N search, Comm. ACM 21 (7) (1978) 550–553.

[4] E.M. Reinhold, J. Nievergelt and N. Deo, Combinatorial Algorithms: Theory and Practice (Prentice-Hall, Englewood Cliffs, NJ, 1977).