XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

A Simple Implicit Dictionary with Polylog Average Cost

Ricardo A. Baeza-Yates
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1 *

Abstract

We study different deletion algorithms in a simple implicit data structure that had already known $O(\log^2 n)$ worst case time for searches and insertions. Experimental results shows that the dominant term in the deletion cost is the number of moves, giving a time complexity of $O(\log^3 n)$.

1 Introduction

Bentley *et al.* [1] introduced a simple implicit data structure based in sorted lists. They store a set of n elements in n consecutive storage locations which are viewed as $\lceil \log(n+1) \rceil$ or fewer sorted lists. [1] There is one list of length 2^i for each i such that the i^{th} bit of the binary representation of n is 1.

In this work we summarize known results of this structure and we discuss the application of this structure to the dictionary problem, in particular to support deletions. We use the RAM model, counting comparisons and movements of data values.

2 Search and Insertion

Using binary searches we have a $\log^2 n/2$ worst case search algorithm. If the lists are searched in decreasing order of size, we have a $2 \log n$ comparison algorithm on the average [1].

Inserting an element requires only the addition of a list of size 1. If there is already a list of size 1, the two are merged into a list of size 2. This process continues as long as possible. In the worst case, an insertion produces at most $\log n$ merges, and then $O(n)$ comparisons and moves in the worst case. However, any sequence of n insertions requires at most $n \log n/2$ comparisons and the same number of moves. That is, the amortized worst case of n insertions is $O(\log n)$ time per insertion [1].

*This work was supported by the Institute of Computer Research of the University of Waterloo and by the University of Chile.

[1]All logarithms are in base 2

If we spread out the merging cost between n insertions, and we maintain at the same time $O(\log n)$ searching in each list, we have a $O(\log n)$ worst case bound per insertion with a more complicated algorithm [4].

Hence, we have a structure that supports insertions in $O(\log n)$ worst case time and searches in $O(\log^2 n)$ worst case time and $O(\log n)$ average case time. Given the simplicity of the structure, this result is very interesting compared with the current best implicit structure for the dictionary problem, that supports the same operations and deletions in $O(\log^2 n)$ worst case time [3].

3 Deletion

Proposed Algorithm

In [1] a few approaches to deletions were suggested. For example, flagging elements. However, this implies 1 extra bit per element (that is, the structure is not implicit any more) and a small increase in run-time. In [4] a different approach was suggested. If we use a redundant binary system (with the digits 0, 1, and 2, where 2 means 2 lists of that size) we can guarantee that there is always at least one list of each possible size. We can see this as having the elements represented in 2 structures, an adding (bitwise) the number of elements of each structure we get the redundant binary code. In general, there is more than one representation in this encoding for each number. We use the one that does not have any 0 digit. The algorithm for a deletion is

```
Delete( a, x )
Sorted_list a;
Element x;
{
   Find and remove x, this leaves a hole in a list with 2**r elements
   for( i = r-1; i > 0; i = i-1 )
   {
       Find a[k] < x < a[k+1] in a list of size 2**i
       ~~Replace x in list r by a[k] or a[k+1]~~ and reorder the list
       x = a[k] or a[k+1]
       r = i
   }
   If there is no lists of length 1; initiate a sequence of "unmerges"
      analogous to the merges required by an insertion
}
```

Clearly, we have three different costs. First, the searching cost to find x. This will be $O(\log^2 n)$ comparisons. Second, the moves and comparisons to "unmerge" if at the end of the deletion no lists of size 1 exists. Using the same idea when inserting, we spread out the cost of the unmerges, having an $O(\log n)$ worst case cost for "unmerges". Here, we will have lists being unmerged and lists being merged. In some cases, an insertion will transform a list in an "unmerging" state to the normal state. The same with deletions, and a list being merged. To keep these states we will need 2 other digits, however, we still need only $O(\log n)$ bits to represent the structure.

Note that the "unmerge" cost is independent of the sequence of replacements used and the cost of searching x is independent of the algorithm used to replace x. Hence, these costs will be always $O(\log^2 n)$ in the worst case and we will not discuss them any more.

Finally, we have the moves and comparisons to find the replacement elements and to replace x and reorder the lists. This is the interesting cost.

We will concentrate our analysis into the number of moves. Clearly, in the worst case, the number of moves is $O(n)$. What about the average? We can divide the number of moves in two parts. The first one, or *basic moves* ($\overline{M}_b$) will account for all the replacements. In the worst case and in the average, we have $O(\log n)$ replacements in the proposed algorithm. The second part, will account for all the moves necessary to reorder the lists (*internal moves*, $\overline{M}_i$). The analysis for this quantity is not trivial. If we assume that the distribution in all the lists remain uniform, we would expect 1 or 2 extra moves per list, giving a $O(\log n)$ time on the average. However, experimental results does not support this assumption [2,4].

The number of comparisons needed to find the replacements $\overline{C}_r$ depends on the algorithm used, and will be analyzed later. The number of comparisons needed for the reordering ($\overline{C}_o$) will be proportional to the number of moves. More specifically

$$\overline{M}_i \leq \overline{C}_o \leq \overline{M}_i + 2\overline{M}_b$$

because we have one comparison for each internal move performed, and in the worst case, we have 2 test comparisons for each basic move.

Replacement Heuristics

The first issue in the given algorithm, is how to replace x. If we have more than one list of the same size, we have 2 obvious choices:

1. Random choice of one of the lists

2. Choose the list that provides the closest element to x

Obviously, the second choice will be better, and only increases the search time by at most a factor of 2.

Suppose that we have now only one list. If we have two choices for replacing x, which one we use? We look at 4 possible alternatives:

1. Random: We flip a coin to choose which element we use.

2. Alternation: We alternate between the right and the left element.

3. Closest: We choose the one closer to x. That is, the one that minimizes the absolute difference between x and the element.

4. Nearest to one side: We choose the one that is nearest to one of the ends of the list. The idea is that (hopefully) if we have to replace the minimum or the maximum of a list, the probability of reordering will be smaller.

Figure 1 shows the experimental results for update cycles (see next section) for $n = 2^{11} - 1 = 2047$ elements. Clearly, the closest heuristic outperforms all the others, and the last heuristic (surprisingly) is the worst one. In the following, we use the *closest* element heuristic unless we explicitly state something different.

Update Cycles

In the following, we assume that n is of the form $2^k - 1$ (the basic problems are captured without loss of generality) and to measure the number of moves we will consider a sequence of updates (deletion/insertion pairs). We delete a randomly chosen element and we insert a random value from a uniform distribution in the range $[0..1]$. For this type of n, the insertion is very simple, because after the deletion there is no list of size 1.

In this case, we have exactly $k-1$ basic moves in the worst case, and

$$\overline{M}_b = \sum_{i=1}^{k-1} \frac{i\, 2^i}{n} = k - 2 + \frac{k}{n} = \log(n+1) + O(1)$$

on the average. We will use this measure to check our empirical results. In the worst case, the number of internal moves is bounded by

$$0 \leq M_i \leq n - k = n - \log(n+1)$$

The analysis of the expected number of internal moves is not simple, even for the first update. In [4] a simple analysis gives $4/5\ (\log(n+1) - 2)$ internal moves for the first update, and the experimental results supported the hypothesis that the constant was $1/2$ instead of $4/5$.

Experimental results in [2,4] showed that the average number of internal moves was not logarithmic and a bound of $\log^2 n/2$ was conjectured. The structure seemed to converge after $O(n)$ updates to a non-uniform distribution, where clusters of elements appeared. Figure 2 shows the change in the distribution variance (initially uniform) for the lists of size 16 to 1024 for $n = 2047$ with several updates. Clearly, the distribution is less uniform for small lists. The number of updates necessary to reach the stable distribution is approximately $3n$ and after that small oscillations continue to exist.

Exact Analysis for Small n

For the analysis we distinguish each possible configuration of the lists using only the ranks (not the values) of the elements, and we associate each configuration to a state in a Markov chain. For example if $n = 3$, the possible configurations are $[(1,2)(3)]$, $[(1,3)(2)]$, and $[(2,3)(1)]$. For example $[(1,2)(3)]$ means that the elements of rank 1 (minimum) and 2 are in the list of size 2, and the maximum element (rank 3) is in the list of size 1. If we have more than one list of each size, then we use a lexicographical order on those lists. Because each list is sorted and each element has a unique rank, each list must be an ascendant sequence of ranks. For $n = 2^k - 1$ the total number of configurations is

$$\binom{n}{2^{k-1}} \prod_{i=1}^{k-2} \binom{n - \sum_{j=i+1}^{k-1} 2^j}{2^i}$$

When we delete an element, we delete its rank from the configuration and we rank the remaining elements again. If we have that 2 choices are possible for a replacement, and both ranks are at the same distance to x, we assign the same probability to each element ($1/2$) of being the closest one. In the case of an insertion, we append a singleton, and we rank all the elements again. Each possible rank of the new singleton is equally likely, because the overall distribution of all the elements is always uniform.

For example, if we delete the element of rank 3 from $[(2,3)(1)]$ we obtain $[(1,2)()]$. Now, if we insert a big element, we obtain $[(1,2)(3)]$. Figure 3 shows the Markov chain for $n = 3$. The solution in this case is very easy. In the steady state of the Markov chain (that is, asymptotically in the number of updates), the probability of being in each state is the same ($1/3$). In all the states with probability $2/3$ we have a basic move and in states 1 and 3 with probability $1/3$ we have an internal move. This gives an average of $2/3$ basic moves and $2/9$ internal moves per update.

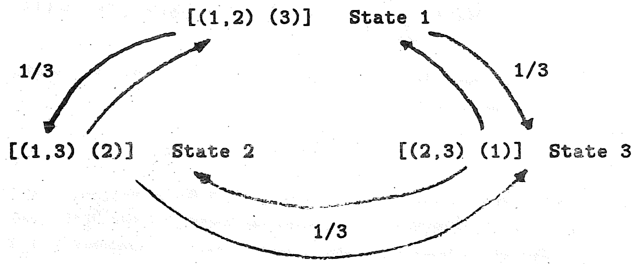

Figure 3: Markov chain for $n = 3$.

Table 1 shows the expected number of moves for $n < 8$ with the experimental results between parentheses. In some cases we give real numbers because the fractional values are too long to be printed.

n	Number of States	Binary Redundant Code	Basic Moves ($\overline{M}_b$)		Internal Moves ($\overline{M}_i$)	
3	3	11	2/3	(.667± .001)	$2/9 = .\bar{2}$	(.2223± .0001)
4	6	12	1/2		$5/52 = .096154$	
5	15	21	4/5		$4/15 = .2\bar{6}$	
6	45	22	2/3		.140665	
7	105	111	10/7=1.42857	(1.429± .002)	.58253	(.5984 ± .0003)
8	420	112				

Note that the number of moves is not monotonic between $n = 3$ and $n = 7$ (but it is monotonic in n of the form $2^k - 1$). Also, the experimental results (95% confidence interval, 100 runs, 50000 updates) are in good agreement with the analysis.

Figures 4a and 4b gives the experimental results for different k and different number of updates. Figure 5 shows the asymptotic expected number of basic moves and the theoretical results; the agreement is perfect. Figure 6 shows the asymptotic expected number of internal moves. A least squares approximations in a model of the form $a(\log^x(n+1)-1)$ gives $x = 2.8617$ and $a = 0.0184$ using 14 values. The error is approximately 0.717. Ineluding lower order terms, we found that a model of the form

$$(a\log^2(n+1)+b)(\log(n+1)-1)$$

was better (note that again we have only 2 parameters). In fact the error was only 0.103, giving $a = 0.0127$ and $b = 0.209$ (this curve is also given in Figure 6). Then, we have

$$\overline{M}_i = 0.013\log^3(n+1) - O(\log^2(n+1))$$

This suggests that the number of internal moves per list is $O(\log^2 n)$. Therefore, the number of internal moves is the dominant cost (asymptotically). The constant of the dominant term is so small that the number of comparisons to find the replacements,

$$\overline{C}_r = \log^2(n+1)/2 + O(\log n)$$

is bigger than $\overline{M}_i$ for $n < 4 \times 10^{11}$. The overall cost for this deletion algorithm is $O(\log^3 n)$ moves and comparisons.

Improving the number of internal moves

If the internal moves are the dominant cost, what we really want is to minimize the number of internal moves to remove the singleton. In general, the optimal sequence of replacements depends on the current state of the data structure. For that reason, we are using a heuristic that will try to work well for almost all the cases.

In the *closest* heuristic is implicitly assumed that a good sequence of replacements must look each time to the next smaller list. However, in general this is not true. May be, the optimal sequence (in the sense of minimal number of internal moves) is to go to some list, and then go back to a bigger one and so on (see Figure 7, case a). Also, it is possible that the optimal sequence visits a list more than once (see Figure 7, case b) or does not visit all the lists (see Figure 7, case c).

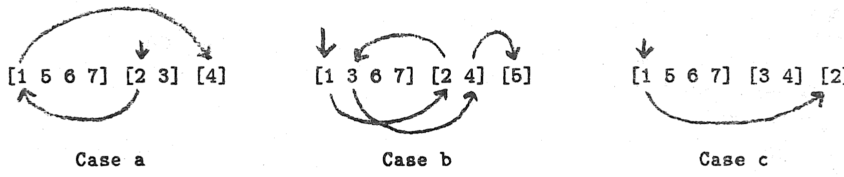

Figure 7: Examples of optimal sequences of replacements.

Intuitively, on the average, the *closest* heuristic seems to be a good approach. However, we have only a 50% chance that the closest element is in the next list rather than in all the others smaller lists (assuming a uniform distribution). One way to increase this chance, it is to look to the closest in all the smaller lists and then use that element. The drawback, is that now the complexity of the searches is $O(\log^3 n)$ and we are looking for something better. One compromise between both schemas, it is to look at the 2 next smaller lists for the replacement element (that is a 75% chance to find the closest one). This introduces at most a factor of 2 in the searching time and hopefully will give a more stable behavior to the structure. This seems to happen based in the experimental results. Now, approximately only $2n$ updates are needed to reach the stable distribution and the oscillations that persist seems to be smaller. The period of the oscillations also seems to be shorter.

Suppose now, that the replacement element (the closest of 2 lists) is in the next list with probability p, and with probability $1-p$ in the next one. If the distribution were uniform, we would expect $p = 2/3$. However, in general, p depends on n, and on the size of the current list. Hence, the expected number of replacements, if we delete an element in a list of size 2^j, is given by the recurrence

$$T(j) = 1 + pT(j-1) + (1-p)T(j-2)$$

with $T(1) = 1$ and $T(0) = 0$. The solution is

$$T(j) = \frac{j}{p-2} + \frac{1-p}{(2-p)^2}(1-(p-1)^j)$$

Then, the expected number of basic moves is given by

$$\overline{M}_b = \sum_{i=1}^{k-1} T(i)\frac{2^i}{n} = \frac{2p^2-7p+6}{(3-2p)(p-2)^2}\log(n+1) + \frac{9-9p+2p^2}{(3-2p)(p-2)^2} + O\left(\frac{\log n}{n}\right)$$

For $p = 2/3$ we have $3\log(n+1)/4 - 21/16 + o(1)$.

The same Markovian analysis for $n = 7$ (in the other cases there are no differences) gives $\overline{M}_b = 1.21101$ and $\overline{M}_i = .50284$. This approach minimizes the number of internal moves for this n, however does not minimizes the total number of moves. In fact, if we always use the singleton as the replacement element, for $n = 7$, we have $\overline{M}_b = 6/7 = .85714$ and $\overline{M}_i = 2/9$. For small n this heuristic is better and we will use this fact ahead.

The analysis for $n = 7$ suggests a value of $p \approx .62$ for this case instead of $2/3$. However, the experimental values $\overline{M}_b = 1.262 \pm .006$ and $\overline{M}_i = .5332 \pm .0009$ indicates a value of $p \approx .71$. For bigger n, p seems to approach $2/3$ (see Figure 5 for the experimental and theoretical curve with $p = 2/3$). Figures 8a and 8b shows the experimental results. The asymptotic values for the internal moves are shown in Figure 6.

For a model of the form $a\log^x(n+1)$ we obtained $x = 2.575$ and $a = 0.0247$ with an error of 0.458. Using a model of the form $(a\log^{3/2}(n+1)+b)(\log(n+1)-1)$ we obtain a much smaller error (0.021) with $a = 0.0308$ and $b = 0.0714$ (this curve is shown in Figure 6). Hence, the dominant cost is still the number of moves, but the complexity is now

$$\overline{M}_i = 0.031\log^{5/2}(n+1) - O(\log^{3/2} n)$$

This suggests that the number of moves in each list is $O(\log^{3/2} n)$. The number of comparisons $\overline{C}_r = \log^2(n+1) + O(\log(n+1))$ is bigger than the number of internal moves for $n < 3 \times 10^{334}$ (!!).

Improving the overall cost

An optimal algorithm in the number of comparisons will be any method that does not search for a replacement element. For example, to use always the singleton to replace the hole. In general, this is not a good idea, because the number of internal moves is then linear on average. Considering the singleton as a random element to be inserted in a uniform distribution, we have that the number of internal moves in a list of size s is

$$M(s) = \frac{1}{s}\sum_i \frac{1}{s+1}\sum_j |i-j| = \frac{s-1}{3}$$

where i are all possible positions for x, and j are all possible positions for the singleton. The expected number of internal moves is then

$$\overline{M}_i = \sum_{i=2}^{k-1} \frac{2^i\ M(2^i)}{n} = \frac{n-1}{9}$$

Experimental results are in good agreement with this formula. Both curves are shown in Figure 6.

If we use a cost function of the form

$$Cost = \alpha(\overline{M}_i + \overline{M}_b) + \beta(\overline{C}_o + \overline{C}_r)$$

we may find a hybrid algorithm that optimizes this cost. Suppose $\alpha = \beta = 1$. In that case, the overall cost of the dumb algorithm is better for small n. Based on empirical results the dumb algorithm is better than the original deletion algorithm for $n < 200$ (approximately). Based in the empirical models and our analysis the overall cost for the original algorithm is better than the new algorithm proposed for any $n < 1 \times 10^{17}$. Hence, for practical values of n, the original algorithm runs faster (see Figure 9).

4 Final Remarks

For large n, we can combine all the different ideas presented in this paper. For example, intuitively the biggest number of internal moves, on average, will be in the biggest list. Hence, why not trying to find the closest one on the next 2 lists in that case? Also, if we are deleting from a small list, why not using the singleton element for the replacement. All these decisions will depend on the application.

For implementation details we refer the reader to [1]. A simple non-implicit implementation (but still using less space than a search tree or other classic dictionary structure) is to use a table of pointers of size $O(\log n)$. The total space is then $n + O(\log n)$.

References

[1] Bentley, J., Detig, D., Guibas, L. and Saxe, J. "An Optimal Data Structure for Minimal Storage Dynamic Member Searching", Dept. of Computer Science, Carnegie-Mellon University, 1978.

[2] Chan, V. "Simulation of Insertion and Deletion in Binomial List", Master Essay, Dept. of Computer Science, University of Waterloo, 1982.

[3] Munro, J.I., "An Implicit Data Structure for the Dictionary Problem that Runs in Polylog Supporting Insertion, Deletion and Search in $O(\log^2 n)$ Time", *JCSS* 33 (1986), 66-74.

[4] Munro, J.I. and Poblete, P., "Searchability in Merging and Implicit Data Structures", *Proc. ICALP*, Lecture Notes in Computer Science 154, Springer-Verlag, New York/Berlin, 1983, 527-535.

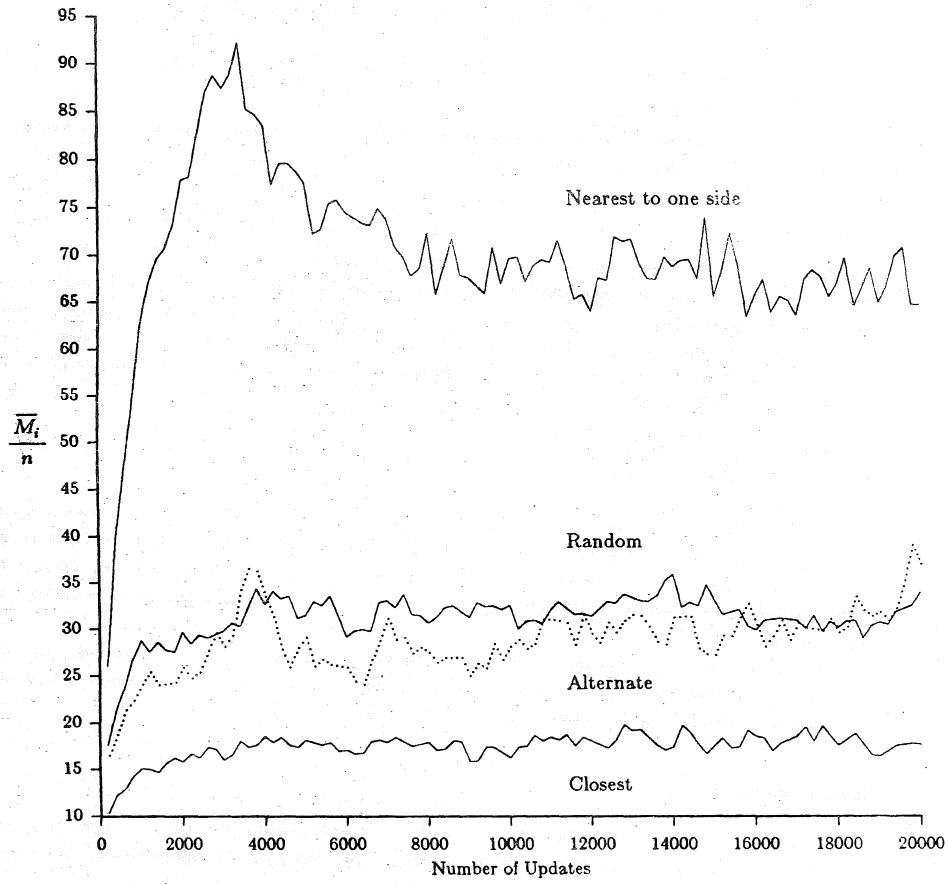

Figure 1. Simulation results for different heuristics (n=2047).

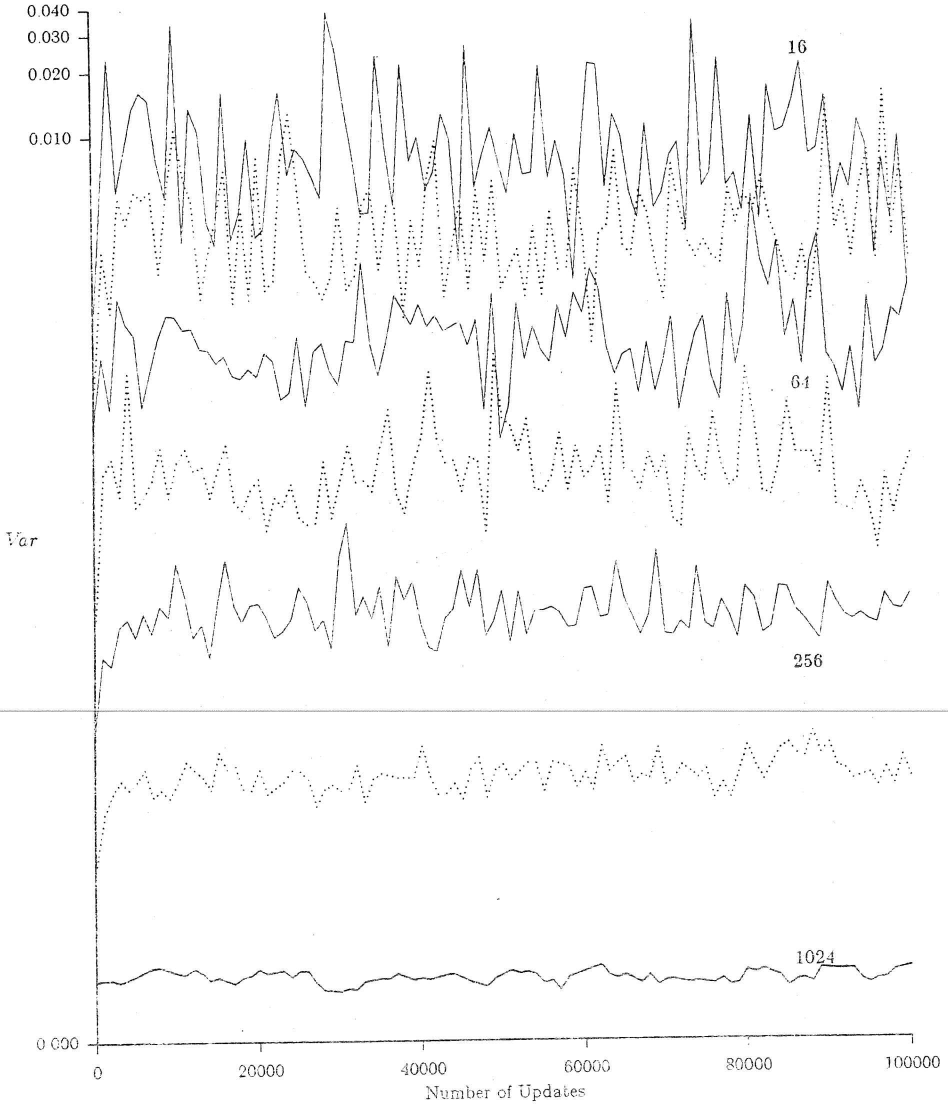

Figure 2. Simulation results for the variance on each list for $n=2047$.

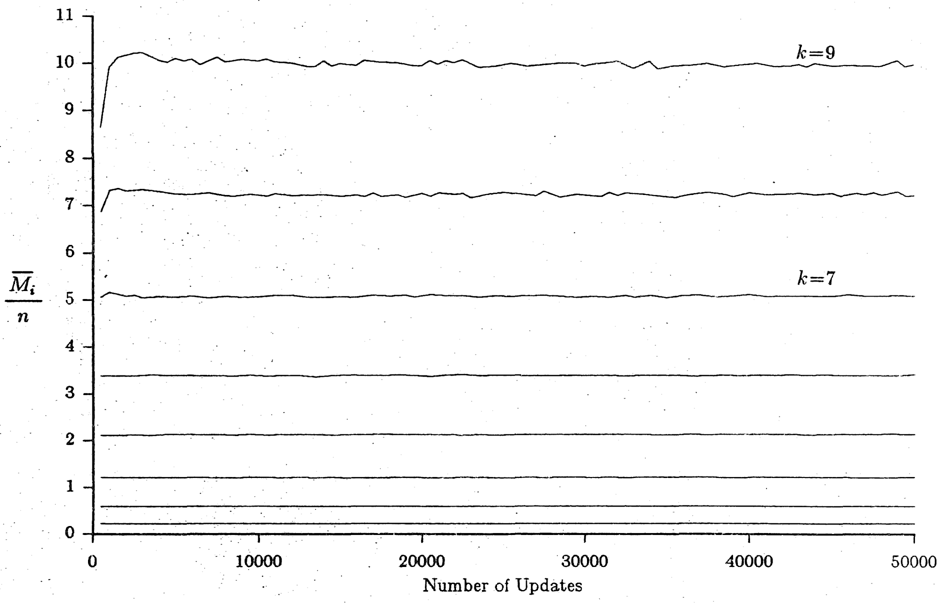

Figure 4a. Simulation results for $n=2^k-1$, $k=2..9$

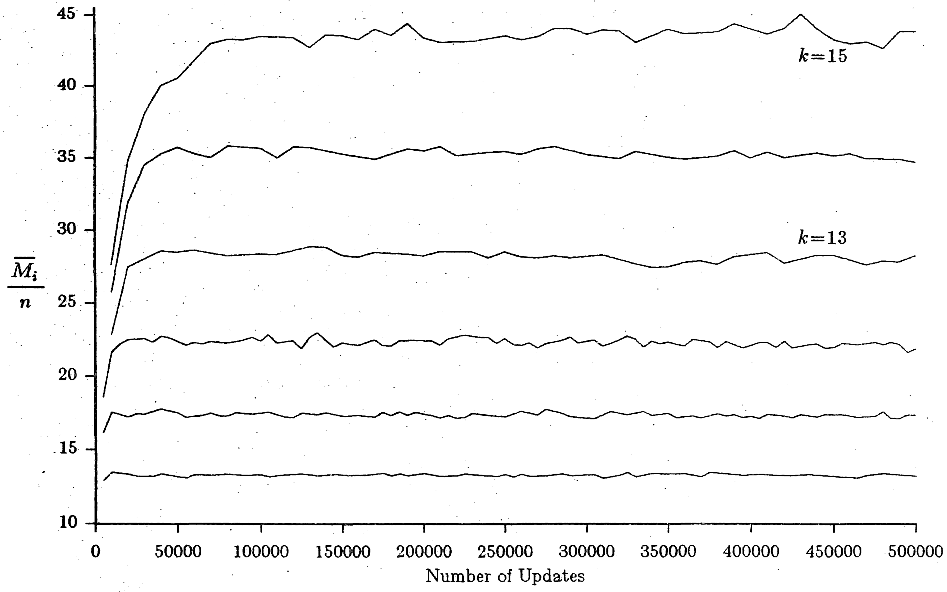

Figure 4b. Simulation results for $n=2^k-1$, $k=10..15$

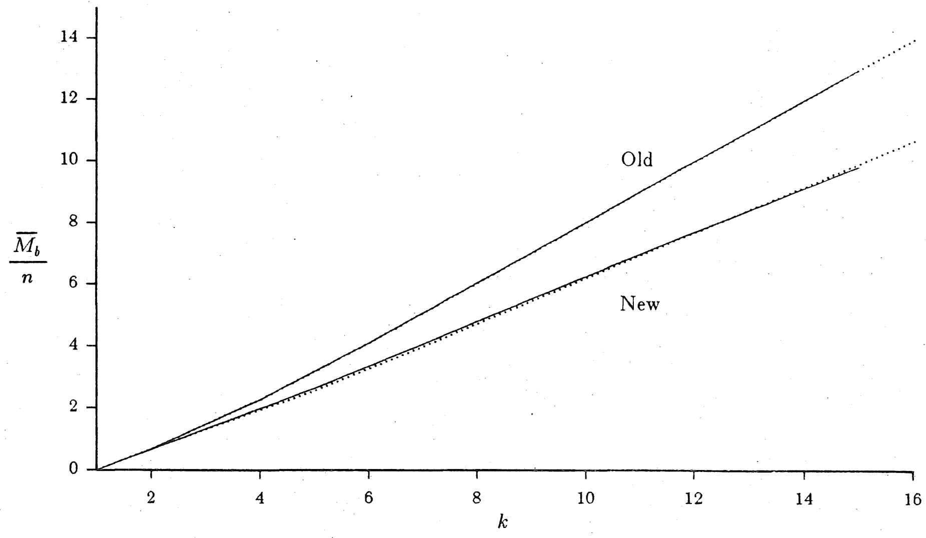

Figure 5. Asymptotic simulation results for $n=2^k-1$, $k=2..15$

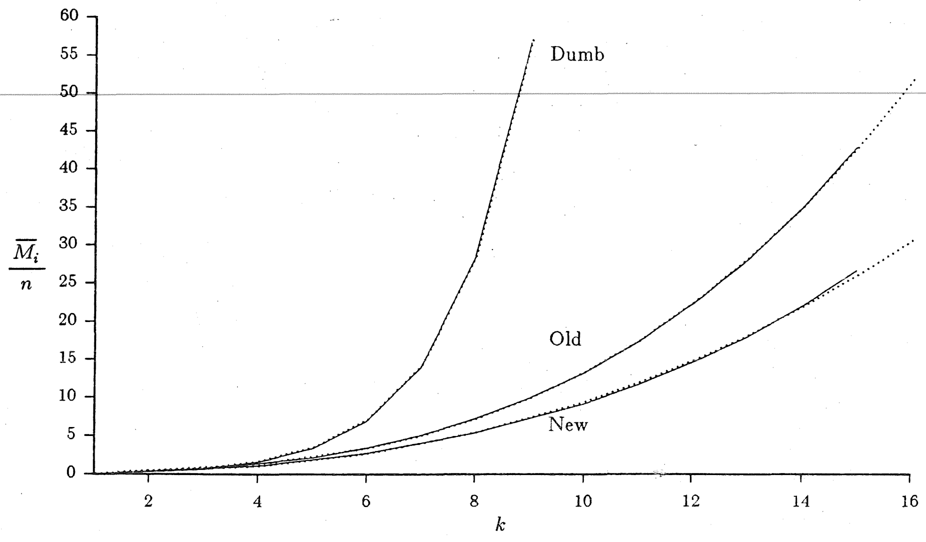

Figure 6. Asymptotic simulation results for $n=2^k-1$, $k=2..15$

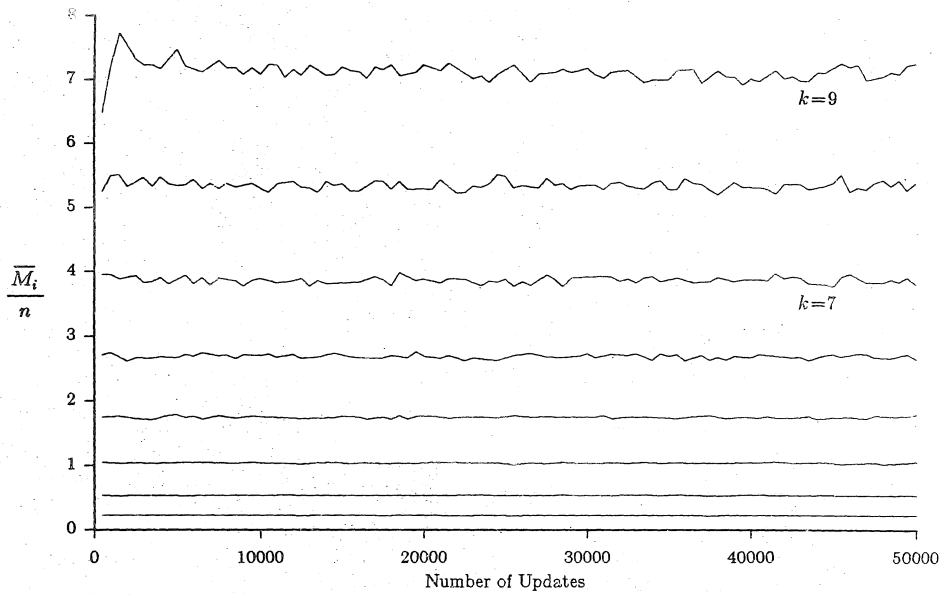

Figure 8a. Simulation results (new algorithm) for $n=2^k-1$, $k=2..9$

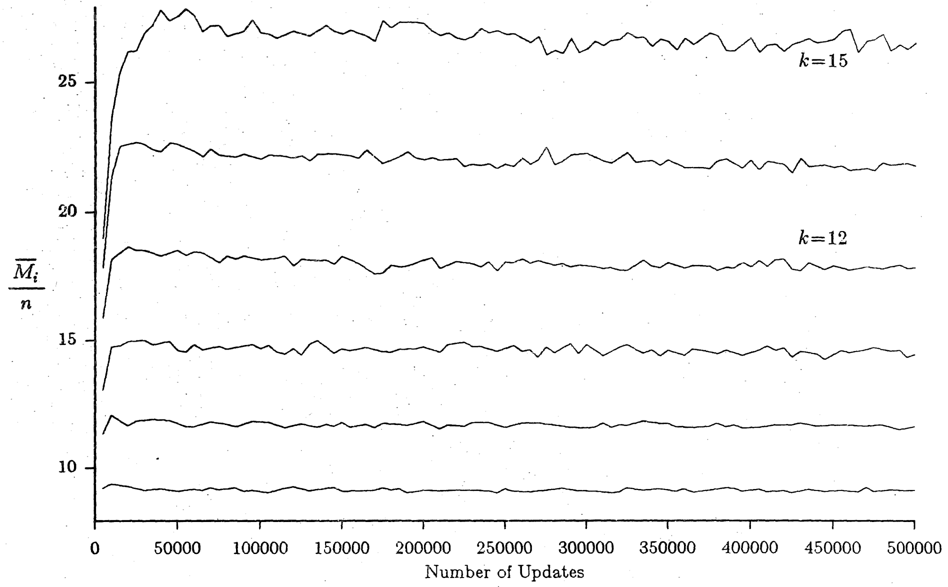

Figure 8b. Simulation results (new algorithm) for $n=2^k-1$, $k=10..15$

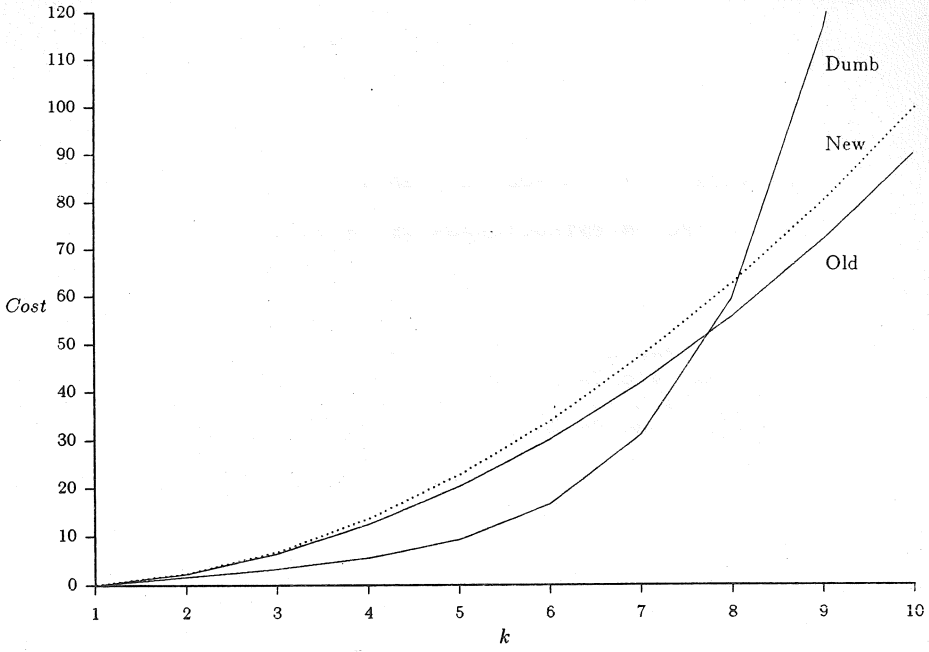

Figure 9. Total experimental cost for $n=2^k-1$, $k=2..9$