# Analysis of a Randomized Data Structure for Representing Ordered Sets[1]

Jon Louis Bentley
Departments of Computer Science
and Mathematics
Carnegie-Mellon University
Pittsburgh, PA 15213

Donald F. Stanat
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

J. Michael Steele
Department of Statistics
Stanford University
Stanford, CA 94305

7 October 1981

## Abstract

Janko has described an efficient randomized sorting algorithm for ordered sets. In this paper we extend the basic data structure of his algorithm to a randomized data structure for representing ordered sets, and give a precise combinatorial analysis of the time required to perform various operations. In addition to a practical data structure, this work gives a new probabilistic lower bound and an instance of a problem whose randomized complexity is provably less than its deterministic complexity.

[1]This research was supported in part by the Office of Naval Research under Contract N00014-76-C-0370 and in part by the National Science Foundation under Grant MCS-78-07736.

## Table of Contents

## 1. Introduction

In this paper we will consider the problem of maintaining a set from a totally ordered domain under the operations Member, Insert, Delete, Predecessor, Successor, etc. The basic data structure we will use to represent such a set is due to Janko [1976, 1977]: a sorted linked list implemented by the two arrays `Link[0..N]` and `Value[1..N]` (where `Link[0]` points to the first element in the list).

In Section 2 we will define the problem more precisely and see that the worst-case complexity of performing a Member search is linear in $N$. In Section 3 we will see a randomized algorithm for Member searching that requires only $\sim 2N^{1/2}$ comparisons, and in Section 4 we will see a lower bound that shows that $\sim(2N)^{1/2}$ comparisons are necessary, on the average. Those sections deal only with Member searching; other operations on sets are considered in Section 5. Finally, conclusions are offered in Section 6.

## 2. The Problem and Its Deterministic Complexity

The data structure we will study was briefly mentioned in the introduction; we will now describe it in more detail. It is a sorted linked list implemented in contiguous storage by the two arrays `Link[0..N]` and `Value[1..N]`. The pointer `Link[0]` points to the first element of the list, `Value[Link[0]]`; the next element can be found in `Value[Link[Link[0]]]`, and so forth. The end of the list is denoted by an element whose `Link` field contains $-1$. Furthermore, we will insist that the array is *dense*: `Value[1..N]` must contain $N$ elements of the represented set. The sortedness of the linked list implies that if `Link[I]` is not $-1$, then

`Value[I]` $\leq$ `Value[Link[I]]`.

We will often refer to `Value[I]` and `Link[I]` together as node I. The following figure illustrates the array representation of the sorted linked list <2.6, 3.1, 4.1, 5.3, 5.8, 5.9, 9.7>.

| I | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Value[I] | | 3.1 | 4.1 | 5.9 | 2.6 | 5.3 | 5.8 | 9.7 |
| Link[I] | 4 | 2 | 5 | 7 | 1 | 6 | 3 | −1 |

It is clear that performing a Member search in such an array requires accessing at most $N$ elements of the array (either by following `Link` fields through the list or simply by iterating through `Value` fields of the array).

We will now show that in the worst case, linear time is necessary to locate whether a given element is in the list. We will assume that a (deterministic) search algorithm is composed of operations of the following types.

1. Determine the index of the node at the head of the list (by accessing `Link[0]`). There is one operation of this type.

2. Determine the `Value` of node I, for $1 \leq I \leq N$ (by accessing `Value[I]`). There are $N$ operations of this type.

3. Determine the successor of node I, for $1 \leq I \leq N$ (by accessing `Link[I]`), where node I was previously accessed by an operation of Type 1, 2 or 3. (There are $N$ operations of this type, but only $r$ of them are valid when $r$ nodes have been accessed by previous operations.) ***Is this a drag? Can we remove it?***

We will assume that each of the above operations has unit cost. (Note that if operations of Type 3 have no cost then binary search can be used to solve the problem in logarithmic time.)

Our model assumes that a protagonist specifies a sequence of the above operations while an adversary ensures that $N$ operations will be required. We will assume that the adversary knows the value of the key the protagonist seeks, which we will call V, and that other key values may be assigned arbitrarily by the adversary. We will describe a strategy that enables the adversary to delay returning V until the protagonist has specified a sequence of $N$ operations.

To aid in his task, the adversary maintains a partial specification of an ordered list that is consistent with his responses to date. The data structure initially consists of two empty linked lists, which we will call the "bottom list" and the "top list". ***Finish

## 3. A Randomized Algorithm

In this section we will consider the randomized complexity of the searching problem. We will investigate an algorithm derived from Janko's [1976]; it uses the fact that the list is stored in contiguous storage to sample randomly several elements of the sorted list, and then follows the links from the one that is closest to the desired element in a sequential scan. Janko analyzed this method statistically using approximations; we will study a precise combinatorial analysis showing that if we sample $k$ elements then the number of comparisons made in the sequential scan is $\sim N/(k+1)$. Choosing $k$ as $N^{1/2}$ shows that $\sim 2N^{1/2}$ comparisons suffice on the average to find the desired element.

The algorithm we will study is described in the following pseudo-Pascal program that searches for the element T in the data structure described in the previous section.

```
P := 0
for I := 1 to K do
    J := Random(1,N)
    if Value[J] < T and Value[J] > Value[P] then
        P := J
while Value[Link[P]] < T do
    P := Link[P]
if Value[Link[P]] = T then
    (* T is in Link[P] *)
  else
    (* T is not present; we could insert it after P *)
```

**Program 1.** Search for T in a linked list.

The operation of the above program is straightforward. The `for` loop randomly samples $k$ elements of the array, and stores in `P` the pointer to the node of greatest `Value` among all sampled nodes with `Value` less than `T`. (The function `Random(1,N)` returns an integer chosen uniformly from the range $1..N$.) After that, the `while` loop follows `Link` fields until it either finds `T` or an element greater than `T`; the appropriate processing of the element could be inserted in place of the comments in the `if` statement. The above code assumes that `Value[0]` is $-\infty$ (this is the zeroth node in the list) and that `Value[-1]` is $\infty$ (this is the last node in the list); those assumptions are not necessary for the algorithm, but they do make the code simpler and more efficient.

We will turn now to an analysis of the running time of Program 1. If $k$ is an input parameter to the procedure, then the only variable in the running time is the number of comparisons made by the `while` loop. That number is a function of two conditions: the position of `T` in the list, and the random numbers returned by the random number generator. Our analysis will be worst-case in one sense: we will assume that the search object `T` is greater than every element in the list (that is, than all elements of `Value[1..N]`). The most important part of our analysis will be probabilistic: we will assume that all $N^k$ possible sequences of outputs from the random number generator are equally likely. Notice that no assumptions are made regarding the distribution of the elements of `Value[1..N]` or of their order in the array.

The following lemma reduces the analysis of Program 1 to a straightforward combinatorial problem. The lemma assumes the definition that $M_{k,N}$ denotes the minimum of $k$ integers chosen uniformly with replacement from $1..N$.

Lemma 3.1: Reduction to minimum.

The distribution of the number of comparisons made by the `while` loop in Program 1 to find a search value `T` that is greater than all elements of `Value[1..N]` is identical to the distribution of $M_{k,N}$.

**Proof:**

The key to this lemma is the permutation function *Perm* defined as follows for any valid assignment of `Link` values (*Perm* is bijective on the integers in $1..N$). Let $j$ be the unique integer such that `Link[`$j$`]`

$= -1$; then $Perm(j) \triangleq 1$. Note that if the variable `P` in Program 1 is $j$ after the `for` loop, then the `while` loop will make exactly one comparison. In general, if $Perm(i) = k$ and $\texttt{Link}[m]=i$, then $Perm(m) \triangleq k+1$. It is easy to establish by induction the fact that if the variable `P` in Program 1 contains $j$ after the `for` loop, then the `while` loop will make $Perm(j)$ comparisons. Furthermore, the `for` loop will set `P` to be that $j$ with minimum value of $Perm(j)$. Thus *Perm* can be viewed as mapping array indices into their distances from the end of the linked list, and Program 1 chooses the minimum index of the $k$ it samples.

To establish the lemma we use the fact that choosing a random integer from $1..N$ and applying an arbitrary permutation function gives a random integer from $1..N$.

The above lemma has reduced the analysis of Program 1 to the combinatorial problem of analyzing the distribution of $M_{k,N}$; the next lemmas do exactly that.

Lemma 3.2: Expectation of minimum.

The expected value of the minimum of $k$ elements chosen randomly with replacement from $1..N$ is

$$E[M_{k,N} = (\sum_{1\le k\le N} i^k) \; / \; N^k \; .$$

**Proof:**

The key observations in the proof are that there are a total of $N^k$ $k$-arrangments of $1..N$, and that $(N-+1)^k$ of those arrangements have a minimum value of $i$. The above expectation then follows by straightforward algebra.

Using standard approximations with the above lemma shows that for fixed $k$ and large $N$, we have $E[M_{k,N}] \sim N/(k+1)$. However, we will be concerned with the case that k] varies as a function of $N$ (and specifically, when $k$ grows as (@iN)$^{1/2}$); we therefore need the following lemma.

Lemma 3.3: Expectation of growing $k$.

For integer $N > 2$ and real $a$ such that $N^{1/2}a \ge 2$,

$$N^{1/2}/a - 1/2 \;\le\; E[M_{N^{1/2}a,N}] \;\le\; N^{1/2}/a \;+\; 1/a \;+\; 2/(a^3N^{1/2}).$$

**Proof Sketch:**

The proof is by the Euler-McLaurin formula, and is omitted from these proceedings for brevity.

With the above lemmas in hand, it is easy to establish the following theorem.

Theorem 3.4: Analysis of Program 1.

If the variable `K` in Program 1 is equal to $N^{1/2}$, then the program makes $2N^{1/2}$ + O(1) accesses of the `Value` array.

**Proof:**

Immediate from Lemmas 3.1 and 3.3.

## 4. A Lower Bound on the Randomized Complexity

Janko [1976] proved that the $\sim 2N^{1/2}$ comparisons of the algorithm of Section 3 are optimal over the class of all strategies that first perform a number of random probes and then sequentially search from the best. In this section we will show a weaker lower bound on the randomized complexity of searching a linked list that holds in a more robust model. We will first define a probabilistic game such that the number of operations in the game is exactly the number of values accessed by the searching algorithm, and then prove a lower bound on the number of operations in the game.

We will now study two variants of a probabilistic game; the second variant corresponds exactly to the problem of searching in a linked list, while the first version contains more information than the searching problem.

**Definition 4.1:** Probabilistic games $G_1$ and $G_2$.

Both variants $G_1$ and $G_2$ of this game involve two integers, $i$ and $N$; the value of $N$ remains fixed throughout the game, and the value of $i$ is originally $N$. The goal of the player is to reduce the value of $i$ to zero in the minimal expected number of steps. The value of the game $G_1$ is defined to be the expected number of steps used by an optimal strategy and is denoted by $G_1(N)$; similarly the value of the game $G_2$ is denoted by $G_2(N)$. A step consists of performing one of the following two operations.

- D (for Decrement): Replace $i$ by $i-1$.
- G (for Guess): Choose $j$ to be an integer uniform from $1..N$ and replace $i$ by $j$ if $j \leq i$.

There are two variants of the game: in the first variant, $G_1$, the player knows at all times the current value of $i$. In the second variant, $G_2$, $i$ is unknown and the player is notified when $i$ becomes zero.

The game $G_2$ models the problem of searching a linked list of $N$ elements if we assume only the operations of randomly probing an element of the list (corresponding the the G operation) and following a link (corresponding to D). The initial condition (that $i=N$) corresponds to having a pointer to the initial element of the list, and the final condition assumes that the search element is greater than all elements in the list.

The analysis of Section 3 shows that in game $G_2$, where $i$ is unknown, the strategy of performing $N^{1/2}$ G operations followed by a series of D operations performs a total of $\sim 2N^{1/2}$ operations, on the average. This establishes the upper bound that $G_2(N) \leq \sim 2N^{1/2}$. We conjecture that this bound is tight, but we cannot prove that. In the rest of this section we will prove a lower bound for $G_2$ by proving a lower bound on $G_1$; since the game $G_1$ contains more information than $G_2$, a lower bound on the former implies a lower bound on the latter (that is, $G_1(N) \leq G_2(N)$). Specifically, we will demonstrate an optimal strategy for $G_1$ that shows that $G_1(N) = \sim(2N)^{1/2}$. These results together identify the value of $G_2(N)$ to within a constant factor of $2^{1/2}$.

We will use dynamic programming to establish the optimal strategy for the game $G_1$. We will let the function $F(i,N)$ denote the number of operations required by the optimal strategy for $G_1$ for given values of $i$

and $N$. The following theorem establishes a recurrence for $F(i,N)$.

**Theorem 4.2:** A dynamic programming recurrence.

The function $F(i,N)$ satisfies the dynamic programming recurrence

$$F(i,N) = 1 + \min\{ F(i-1,N),\ (1/N) \cdot \sum_{1 \le j < i} F(j,N) + (N-i+1) \cdot F(i,N)/N \},$$

with the boundary condition that $F(0,N) = 0$.

**Proof:**

The boundary condition states that the problem has been solved once $i$ is zero. The recursive part minimizes the expected value for arbitrary values by choosing the operation with minimum expected cost. If the D operation is chosen, the value of $i$ is decreased by one. If the G operation is chosen, there are two possible outcomes. With probability $(N-i+1)/N$, the value of $i$ is unchanged; with probability $1/N$, it assumes each of the $i-1$ positive integers less than $i$. The recursion is valid by the principle of optimality.

Note that the optimal strategy corresponding to the recurrence must know the value of i to choose which operation to perform at each step.

To express the solution to the above recurrence, we will let $T_k$ denote the $k^{th}$ triangular number, $k \cdot (k-1)/2$.

Theorem 4.3: Solution of the recurrence.

If $N = T_m + k$, where $0 \le k < m$, then

$$F(i,N) = F(i, T_m + k) = \begin{cases} i-1 & \text{for } i \le m \\ m + k/m & \text{for } i > m \end{cases}$$

Note that $T_m$ is the largest triangular number not greater than $N$.

**Proof:**

The proof for a fixed value of $N$ is a straightforward inductive argument on $i$, and is omitted in these proceedings for brevity.

Because $T_m \sim m^2/2$, the value of $m$ in Theorem 4.3 is $\sim(2N)^{1/2}$. This establishes the claim that $G_1(N) = F(N,N) = \sim(2N)^{1/2}$.

The solution to the dynamic programming recurrence implies the optimal strategy for the game $G_1$. Let $m$ be the largest triangular number less than $N$ (as in Theorem 4.3); if the current value of $i$ is greater than $m$, then the optimal strategy performs a G, otherwise it performs a D. It is interesting to observe that both the solution to the recurrence and the optimal strategy remain unchanged if the game is modified so that the G operation always sets $i$ to the random integer $j$.

## 5. A General Data Structure

In previous sections we have considered only the problem of searching the linked list to determine if it contains a given element. It is easy to perform many other set operations on this structure; the following list summarizes those operations, and describes their costs in terms of the number of `Value` elements accessed.

- **Member:** The previous sections studied the problem of searching to determine whether a given element is a member of the set represented by the linked list. *Cost:* $\sim 2N^{1/2}$.

- **Insert:** A new element can be inserted in the list by using Program 1. *Cost:* $\sim 2N^{1/2}$.

- **Delete:** The first step in deleting an element is to find that element by a search algorithm like that in Program 1, and then modify the `Link` field of its predecessor to point to its successor; that step makes $\sim 2N^{1/2}$ references to the `Value` array. The next step must patch the "hole" created in the dense array by moving the last element of the array to the vacant position; searching for the last element requires $\sim 2N^{1/2}$ references. *Cost:* $\sim 4N^{1/2}$.

- **Predecessor:** The element immediately preceding a given element can be found by a modification of Program 1. *Cost:* $\sim 2N^{1/2}$.

- **Successor:** The element immediately succeeding a given element can be found by a modification of Program 1. *Cost:* $\sim 2N^{1/2}$.

- **Minimum:** The minimum element in the set is pointed to by `Link[0]`. *Cost:* 1.

- **Maximum:** The maximum element in the set can be found by a modification of Program 1. *Cost:* $\sim 2N^{1/2}$.

Each of the above operations is straightforward to implement given the model of Program 1 and basic techniques for dealing with data structures for searching (described, for example, by Knuth [1973]). Furthermore, the simplicity of that code implies that the constant factors of the running time of the program will be relatively slow (Janko [1976] describes an implementation in detail). The only deviation that a programmer should make from the code of Program 1 deals with the random number generation: since many random number generators are very expensive, it might be preferable to use some other approach to sample the $k$ elements.

This data structure is superior to all of those described in Knuth [1973] for certain applications; the salient attributes of such applications are listed briefly.

- Space is important. This structure uses only one extra word of storage per element, while binary search trees use at least two extra words, and various hashing schemes use varying amounts of extra storage. However, the storage for this structure must be available in a single contiguous block.
- The "orderedness" operations of Successor, Predecessor, Min and Max are frequent; these are not possible in most hashing schemes.
- Insertions and deletions are frequent. If the data structure changes rarely, binary search in a

sorted array is very efficient.

- Program simplicity is important. Each operation on this structure requires only about a dozen lines of code, while some operations on balanced binary search trees require over one hundred lines of code.
- Run time is important for problems of medium size (where medium means that N is between, say, 100 and 10000). If $N$ is below that range, simple sequential strategies are probably efficient enough. If $N$ is above that range, then the logarithmic search time of binary search will be necessary for many applications. When $N$ is in that range, though, the low constant factors of this structure will make it competitive with binary search trees.

## 6. Conclusions

The following list summarizes our extensions to Janko's work.

- We have extended the problem from sorting an ordered set to maintaining an ordered set under various set operations. The data structure has a number of desirable properties both theoretically and practically (it uses little space, has good randomized time and is easy to implement).

- We have tightened Janko's upper bounds on Member searching to give an exact combinatorial analysis of the procedure.

- We have given a lower bound on the problem of searching a linked list with random probes that is weaker than Janko's but holds in a more robust model.

- We have interpreted Janko's construction as a randomized algorithm; that class of algorithms was identified as a class only after Janko's [1976] paper was published. This problem provides an example of a problem whose randomized complexity is provably less than its deterministic complexity; few examples of such problems are currently known.

This work raises a number of open problems. Perhaps the most obvious is to tighten the bounds on the randomized complexity of the searching problem (using the game $G_2$); the bounds we present are a factor of $2^{1/2}$ apart. We conjecture that the game $G_2$ of Section 4 has the value of $\sim 2N^{1/2}$. A broader open problem is to explore additional randomized data structures, and the application of other probabilistic games to the analysis of randomized algorithms and data structures.

## References

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

Janko, W. [1976]. "A list insertion sort for keys with arbitrary key distribution," *ACM Transactions on Mathematical Software* 2, 2, June 1976, pp. 143-153.

Janko, W. [1977]. "An insertion sort for uniformly distributed keys based on stopping theory," *International*

*Computing Symposium 1977*, April 1977, North-Holland Publishing, pp. 373-379.

Knuth, D. E. [1973]. *The Art of Computer Programming, volume 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts.