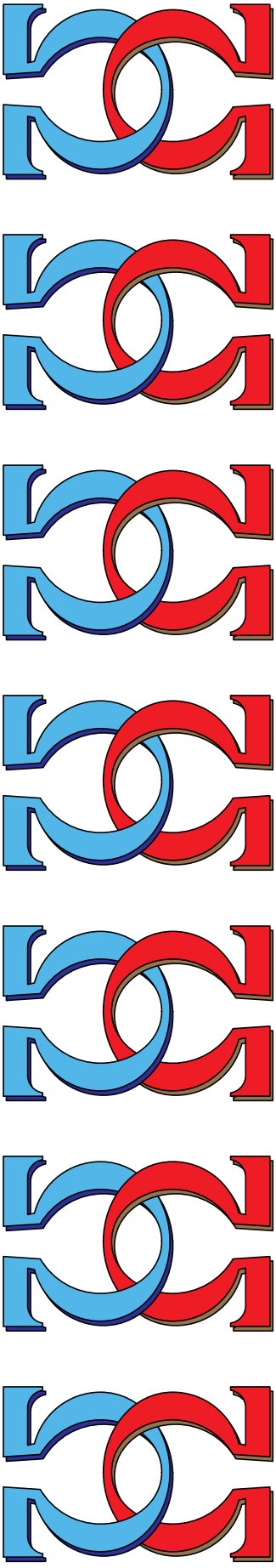


CDMTCS
Research
Report
Series

A Fast Natural Algorithm for Searching

Joshua J. Arulanandham
Cristian S. Calude
Michael J. Dinneen
Department of Computer Science
University of Auckland

CDMTCS-220
June 2003 (revised Oct 2003)

Centre for Discrete Mathematics and
Theoretical Computer Science

A Fast Natural Algorithm for Searching

Joshua J. Arulanandham, Cristian S. Calude, Michael J. Dinneen
Department of Computer Science
The University of Auckland
Auckland, New Zealand
{jaru003,cristian,mjd}@cs.auckland.ac.nz

Abstract

In this note we present two natural algorithms—one for sorting, and another for searching a sorted list of items. Both algorithms work in $O(\sqrt{N})$ time, N being the size of the list. A combination of these algorithms can search an unsorted list in $O(\sqrt{N})$ time, an impossibility for classical algorithms. The same complexity is achieved by Grover's quantum search algorithm; in contrast to Grover's algorithm which is probabilistic, our method is guaranteed correct. Two applications will conclude this note.

1 Introduction

Sorting and searching are fundamental for computer processing, so any attempt to design fast methods for either operation is important. Looking up a name in a telephone directory given a telephone number is exponentially more difficult than looking up a telephone number given a name. Indeed, in the second case $\log N$ steps are enough, but in the first case we need about $N/2$ steps on average and N steps in the worst case. Can we do it better?

In [3] we have proposed *Bead–Sort*, a natural sorting algorithm. Working closely on the same theme, and using a physical mechanism similar to the one used for Bead–Sort, we propose a natural algorithm for searching a sorted list in $O(\sqrt{N})$ time. To perform search on an *unsorted* list, we can first use Bead–Sort to quickly sort the list (in $O(\sqrt{N})$ time as detailed in Section 2) and then apply the proposed search procedure on the resulting sorted list; the combination of the two algorithms works in $O(\sqrt{N})$ time, N being the size of the list. Two applications and a comparison between Grover's algorithm (a quantum algorithm searching an unsorted list in $O(\sqrt{N})$ time) and our procedure will conclude the paper.

In Section 2 we review Bead–Sort; in Section 3 we introduce the natural search technique and prove its correctness with formal arguments; in Section 4 we discuss how the method can be adapted to sort and search databases; in Section 5 we introduce a natural (dynamic) data structure called *BeadStack* (for efficiently finding the minimum

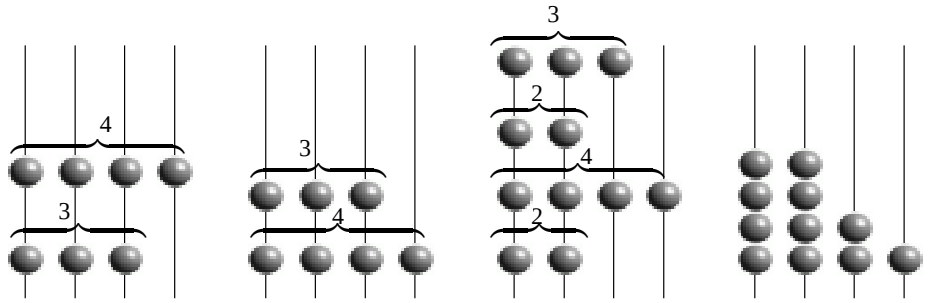


Figure 1: Illustrating Bead–Sort.

and the maximum of a set of integers as well as allowing real–time insertions and deletions). We show that *BeadStack* compares well with the recently proposed *SquareList* data structure. Finally, in Section 6 we compare Grover's algorithm with our algorithm and we finish with some concluding remarks.

2 Bead–Sort

The following is a review of the sorting algorithm we proposed in [3]. The sorting algorithm for positive integers, which we call *Bead–Sort*, is based on a simple natural phenomenon. In what follows, we represent positive integers by a set of *beads* sliding through *rods* as in an abacus (see Figure 1). Figure 1(a) shows the numbers 4 and 3 (represented by beads) attached to rods; beads displayed in Figure 1(a) appear to be suspended in the air, just before they start sliding down. Figure 1(b) shows the state of the *frame* (a *frame* is a structure with the rods and beads) after the beads are allowed to slide down. A row of beads representing number 3 has "emerged" on top of the number 4 (the extra bead in number 4 has dropped down one row). Figure 1(c) shows numbers of different sizes, suspended one over the other (in a random order). We allow beads (representing numbers 3, 2, 4 and 2) to slide down to obtain the same set of numbers, but in a sorted order again (see Figure 1(d)). In this process, the smaller numbers "emerge" above the larger ones and this creates a natural *comparison* (an online animation of the above process can be seen at [1]; for a simulation see [4]).

We now present the Bead–Sort natural algorithm. Consider a set A of n positive integers to be sorted and assume the biggest number in A is m. Then, the frame should have at least m rods and n rows. (See Figure 2; *Rods* (vertical lines) are counted always from left to right and *rows* are counted from top to bottom.[1]) The Bead–Sort algorithm is the following:

The Bead–Sort natural algorithm

`For all` $a \in A$ `drop` a `beads (one bead per rod) along the rods` $1, 2 \ldots, a$.

[1]Note that in [3] we have used the term "level" instead of "row", and counted the levels from bottom to top.

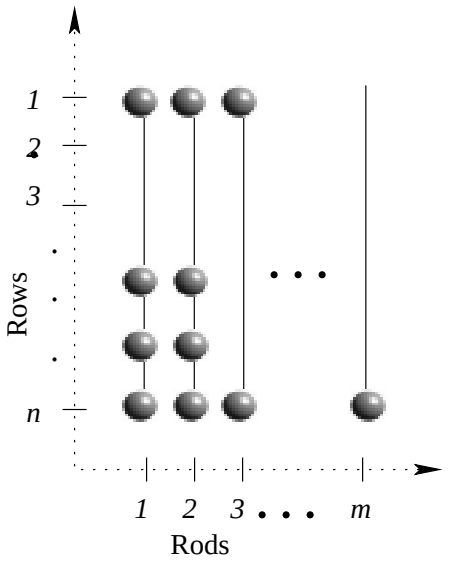


Figure 2: Bead–Sort conventions.

`Finally, the beads, seen row by row, from the` 1^{st} `row to the` n^{th} `row represent` A `in ascending order.`

See [3, 2] for a formal proof of correctness and for various implementations of Bead–Sort. Note that we work with a *row–of–beads* as a basic "data object" (similar to a "byte" or a "word" in a digital computer) and not with a single bead. This means, we can (i) introduce a row–of–beads (consisting of say, n beads) into the rods and also (ii) read a row–of–beads (i.e., read the number of beads in a given row) all at once, in parallel. Gadgets that realize (i) and (ii) can easily be designed in practice.

The time complexity of Bead–Sort is actually the time taken by the beads to settle down in a state of rest. We assume that the whole of the input is first read (imagine the initial state of the frame to be a set of unsorted rows of beads "suspended" in the air[2]), and then the beads are allowed to drop down in parallel. The beads can be viewed as free falling objects accelerating due to gravity.[3] Hence, in the worst case, the time taken by the beads to settle down is $\sqrt{\frac{2h}{g}}$, where h is the height of the rods and g is the acceleration due to gravity. If we fix the height of the rods to be the same as the size (N) of the list, then the time complexity is given by $\sqrt{\frac{2N}{g}}$, i.e. $O(\sqrt{N})$. Note that the complexity analysis of Bead–Sort[3] was done using a different perspective: we have analyzed the complexity by viewing certain actions as "computational steps" and then by *counting* them rather than by *measuring* the time taken for the whole process; treating "dropping of beads" as a single step, we have obtained the complexity $O(1)$ for an implementation of Bead–Sort where all beads are dropped in parallel. The present analysis is more realistic.

[2]Alternatively, one could imagine that the input is read with the frame kept in the horizontal position; it can then be tilted to a vertical position to sort the list by allowing the beads to drop.

[3]The distance d travelled by a free falling object in time t is given by $d = \frac{1}{2}gt^2$, where g is the acceleration due to gravity.

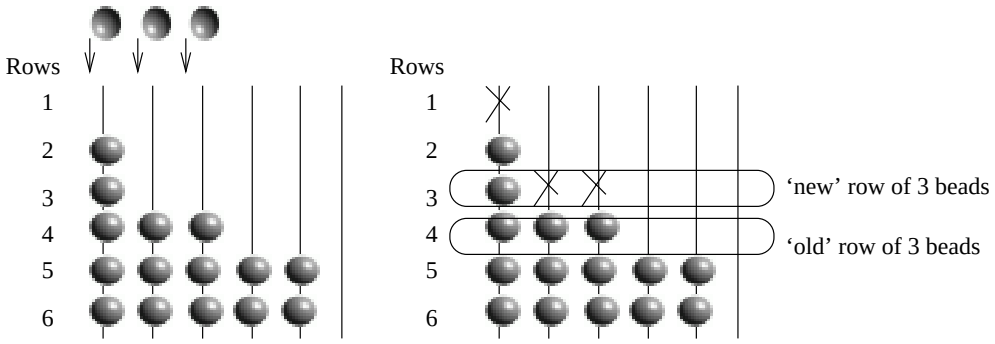


Figure 3: Introducing a new integer into the list.

3 Searching for a needle in a "bead–stack"

Consider the problem of searching for a given integer in a list of positive integers (that is already sorted using Bead–Sort). And, let us imagine that the sorted list is represented in the form of beads in the same "beads–rods" apparatus used for Bead–Sort. We use the following simple observation to do the search. Suppose that the list already contains an integer, say 3 (this means, there is a row of 3 beads in the frame). Now, introduce one more '3' into the list by dropping 3 beads, as before from left to right, one bead per rod. We would eventually find the new integer (3, in our case) just above the other '3' that is already in the list (see Figure 3), thus preserving the sorted order.[4] We can show that, when we introduce an integer n into the list, at least one of the beads representing n (the bead sliding along the n^{th} rod, to be precise) will eventually find itself just above the already existing integer n. The main point in the above illustration is this: the search for the integer n can be reduced to simply introducing a new row of n beads into the list and tracking the last bead as it falls down, say, with some device. The newly introduced beads actually "locate" the integer, if present. But, how would the above method indicate the absence of an integer n in the list? Actually, we would have to drop $n + 1$ "search beads" to determine whether an integer n is in the list or not. In what follows, we present the general (natural) algorithm for searching, and a proof of its correctness.

To do the search, we use an extra apparatus—a tracking mechanism along with the beads and rods. It consists of very thin tapes with markings as shown in Figure 4 (similar to the "measuring tape" or the "inch tape") whose ends are attached to the *search beads*. (*Search beads* are those that are dropped into the list and tracked.) When the search beads are let down, the tapes unfold, exposing the markings on them; the readings seen against the "measurement level" (see Figure 4) at any point of time give the row position of the search beads attached to them. The natural algorithm for searching follows:

The *Bead–Search* natural algorithm

[4]Note, however, that not all the beads that we introduce (call them "search beads") might appear in the newly formed row of 3 beads. In the example shown in Figure 3, the new row of 3 beads contains only two of the search beads.

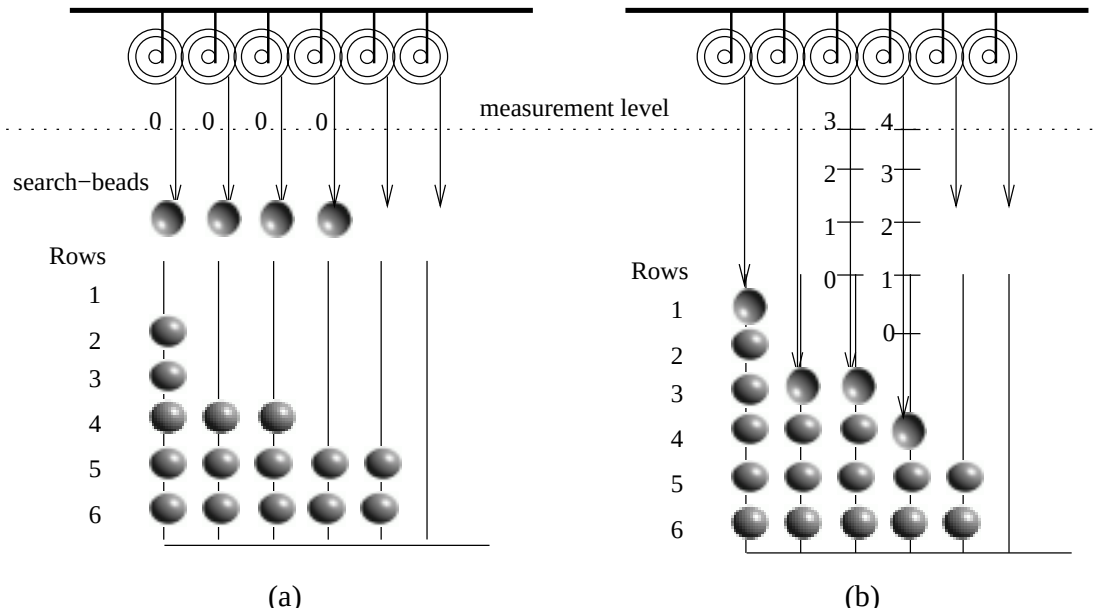


Figure 4: Searching for the integer 3.

To search for an integer n and to determine its location in the list if present, do the following:

1. Drop $n+1$ search beads (one bead per rod) along the rods $1, 2 \ldots, n+1$ (and wait for them to settle down).
2. Compare the readings (taken against the measurement level) of the n^{th} and the $(n+1)^{th}$ search beads. Call them READ(n) and READ$(n+1)$.
3. If READ$(n) =$ READ$(n+1)$, then the integer n is not in the list. If READ$(n) \neq$ READ$(n+1)$, then the integer n is in the list, and can be found on the row READ$(n+1)$.

More precisely, when READ$(n+1) -$ READ$(n) = x$, the integer n occurs in the list x times, starting from row READ$(n)+1$ to row READ$(n+1)$. A few self–explanatory illustrations—searching for integer 3 (present in the list) and 2 (not present in the list) are given in Figures 4 and 5. (Note that the search beads that are dropped down can be pulled up after searching.)

We now show that step 3 in the natural algorithm is indeed correct. In other words, we show that READ$(n) =$ READ$(n+1)$ if and only if the integer n is not in the list.

Suppose, READ$(n) =$ READ$(n+1) = r$. It is clear that the integer n is not on row r, since the bead–position given by the n^{th} rod and the row r is now presently being occupied by the n^{th} search bead. It is also clear that either there is a bead on $(r+1)^{th}$ row, in rod $n+1$ stopping the $(n+1)^{th}$ search bead from dropping further (in which case, there is an integer greater than n on row $r+1$) or, row r is the last row in the mechanical frame. In both the cases, however, n cannot be beneath the row r (note that the list is sorted). Also, the integer n cannot be in one of the rows above r (i.e., the rows 1 to $r-1$). This is because there is no bead in rod n on any of the rows 1 to $r-1$;

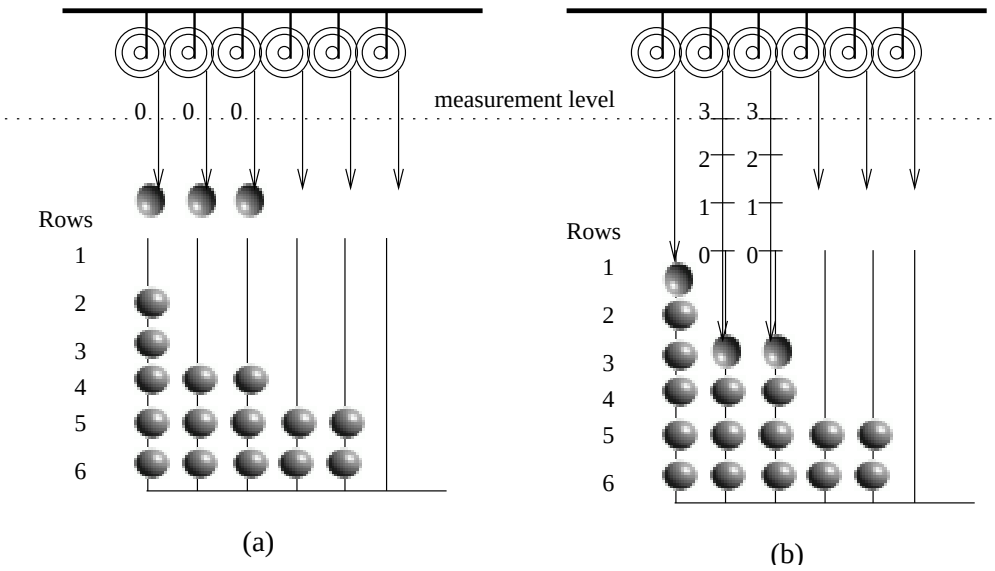


Figure 5: Searching for the integer 2.

otherwise, the n^{th} search bead would not have descended down to the r^{th} row. Thus, the integer n is not in any of the rows. It now follows that when $\text{READ}(n) = \text{READ}(n+1)$, the integer n cannot be in the list. Before proving the converse, let us first observe the following simple fact: for every i, j such that $i < j$, the number of beads in rod i is greater than or equal to the number of beads in rod j (after the beads settle down). This is because we always drop beads from left to right, one bead per rod. Therefore, when $\text{READ}(n) \neq \text{READ}(n+1)$, we can immediately deduce that $\text{READ}(n) < \text{READ}(n+1)$. Now, we are ready to prove the converse. Suppose $\text{READ}(n) \neq \text{READ}(n+1)$, and let $\text{READ}(n+1) = r$. This means, the $(n+1)^{th}$ search bead has dropped to the row r; thus, there is no bead in the $(n+1)^{th}$ rod on all rows starting from 1 to r, except for the search bead. But, since $\text{READ}(n) < \text{READ}(n+1)$, there are beads in the n^{th} rod on rows $r, r-1, r-2, \ldots, \text{READ}(n)+1$. It follows that there is an integer n on all these above rows. I.e., the integer n is present in the list, $\text{READ}(n+1) - \text{READ}(n)$ times. (Indeed, it can also be shown that the list cannot have the integer n in a row other than these.)

The time complexity of this search operation is similar to that of Bead–Sort; it is the time taken by the search beads (dropped in parallel) to settle down in their final positions, and hence the complexity is $O(\sqrt{N})$, as before.

4 Sorting and searching a database

First of all, we observe that Bead–Sort does not rearrange (physically) the rows of beads representing positive integers. For instance, see Figures 1(a) and 1(b): we do find a row of beads representing number 3 on top in Figure 1(b), but this is not the *same* row of beads that we originally used in Figure 1(a) to represent number 3. (They, in fact, still remain at the bottom, even after "sorting" has occurred.) Thus, the "original number 3"

Figure 6: Sorting a database: mere colored beads do not help.

has not moved up at all! This property is both the strength as well as the weakness of Bead–Sort— you do not have to swap or shuffle the (objects that represent) numbers in order to produce a sorting effect; but, the very same property has a negative effect when we attempt to sort a (hypothetical) database like the one shown in Table 1. We now illustrate the fact that ordinary Bead–Sort cannot accomplish this.

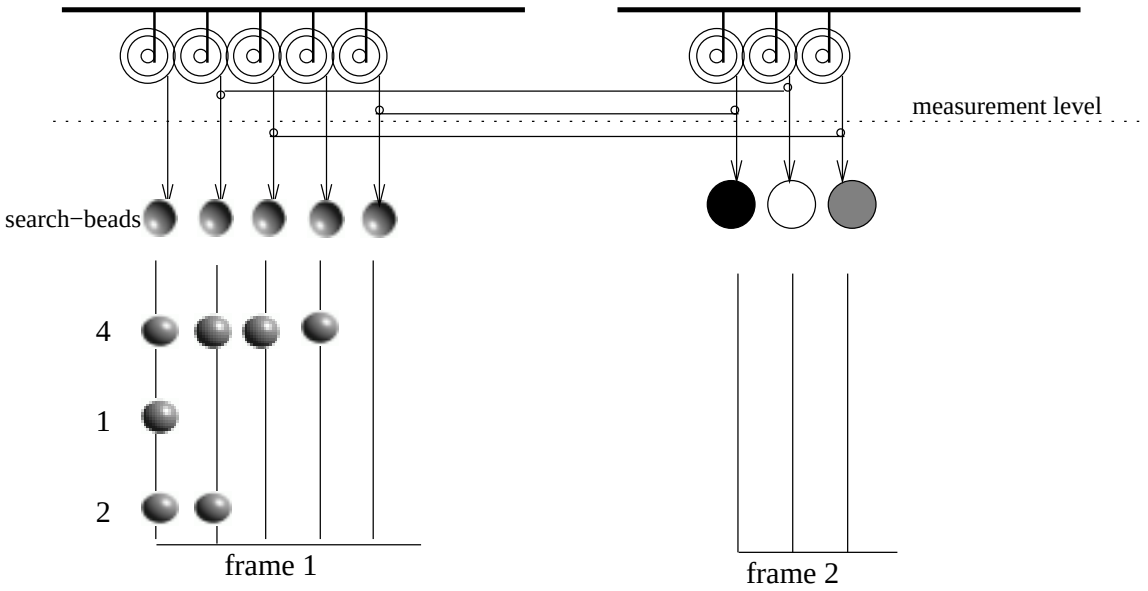
Table 1: Toy database.

customer_ID (Key)	*vehicle_color* (Information associated with key)
4	black
1	white
2	grey

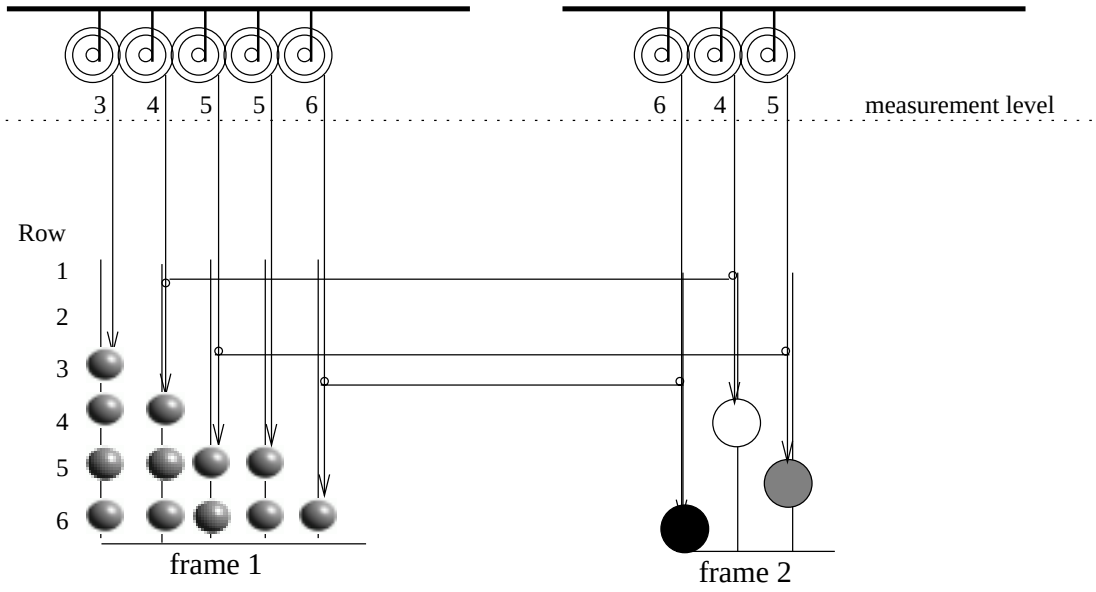
Represent *customer_ID* as usual, with beads. Also, represent *vehicle_color* using, say, the color of the beads as shown in Figure 6. Now, from the resulting "sorted" list, it is clear that we cannot extract the right mapping between the keys and their associated information easily. The mapping is lost, though we have got the keys themselves sorted. However, we can solve this problem in an indirect way as discussed below.

Represent *customer_ID* with rows of beads in frame 1; see Figure 7. (We assume that the keys are unique.) The search beads are ready to slide down along each rod in frame 1 and will be used for a purpose discussed later. Represent *vehicle_color* with a different set of beads (call these "color beads") on a separate frame, i.e. frame 2 as shown in Figure 7; for representing a vehicle color, use one bead with a distinct color.[5] We place the black bead representing the *vehicle_color* "black" (the first entry in the database) on the first rod, the white bead representing "white" (the second entry in the database) on the second rod, and so on. Note that the color beads are not free falling objects, but can be made to drop down by coupling them with the search beads in frame 1.

[5]We use a distinct *colored* bead to represent information associated with a particular key just for the sake of illustration; one could have used "labels" or "tags" (that are stuck on the beads, say) to represent the same.



(a) Initial state (before sorting)



(b) Final state (after sorting)

Figure 7: Sorting a database: a different approach.

Having represented both *customer_ID* and *vehicle_color* individually, we now represent the mapping between them in the following way. For instance, to map the *customer_ID* "4" with *vehicle_color* "black", we just couple the 5^{th} search bead (of frame 1) with the black colored bead; to map *customer_ID* "1" with *vehicle_color* "white", we connect the 2^{nd} search bead to the white colored bead. In general, to map a key n to the information associated with it (i_n), we couple the $(n+1)^{th}$ search bead with the bead on frame 2 representing i_n. (Note that all these are part of the input setting up process, and does not involve a search by itself.)

Now, how do we sort the database? First, sort the *customer_ID*s by allowing the beads on frame 1 to drop down. After they are sorted, allow all search beads in frame 1 to roll down. As detailed in Section 3, the $(n+1)^{th}$ search bead, after settling down, will be exactly on the same row as the *customer_ID* n. (Recall from Section 3 that the reading corresponding to the $(n+1)^{th}$ search bead gives the location of integer n in the list, if present.) Also, the search bead would have pulled down along with it, its "partner", i.e. the color bead representing i_n (the one coupled with it) to exactly the same row, thus aligning each *customer_ID* with its corresponding *vehicle_color*. Indeed we could initiate a search on the database, after the sorting is over.

The major drawback with the above technique is that every time we wish to insert a new (key + information) into the database, we would have to redo the whole alignment procedure once again.

5 The BeadStack Min/Max data structure

In this section we propose a natural (dynamic) data structure called *BeadStack*. The operations of interest are finding the minimum and the maximum of a set of integers, along with insertion and deletion operations. Our data structure has performance comparable to the recently proposed *SquareList* data structure (see [6]). We list the best running times for various "classic" data structures in Table 2 and compare it to BeadStack.

Table 2: Expected performance of common data structures.

Data Structure	Insert	Delete	Find min	Find max
Priority Queue (Heap)	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$
Binomial Heap	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(N)$
Skiplist	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$
Fibonacci Heap	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(N)$
SquareList	$\Theta(\sqrt{N})$	$\Theta(\sqrt{N})$	$\Theta(1)$	$\Theta(1)$
BeadStack	$\Theta(\sqrt{N})$	$\Theta(\sqrt{N})$	$\Theta(1)$	$\Theta(1)$

The BeadStack data structure is our standard collection of beads on rods, where each row of beads (flush left) represents a positive integer. Recall that we work with a *row–of–beads* as a basic data object. The "Find min" operation is simply to return the top row of beads. Likewise, the "Find max" operation is simply to return the bottom

row of beads. These two operations can be done in $\Theta(1)$ time. The insertion operation is done simply by dropping another row of beads on top of the existing stack of integers; it requires $\Theta(\sqrt{N})$ time in the worst case. The deletion operation is done by performing a "search" to find the row containing the integer to delete, followed by the removal of that physical row of beads. As shown in an earlier section, this again can be done in time $\Theta(\sqrt{N})$, which is the cost of the search operation plus a constant time for deletion.

6 Conclusions

The proposed search algorithm works in $O(\sqrt{N})$ time for any unsorted list, an impossibility for classical algorithms. This is a significant complexity reduction: for example, a classical computer will have to look on average at 500,000 items to perform a search in an unsorted list of 1,000,000 items as opposed to only 1,000 required by our algorithm.

Of course, we have discussed an algorithm involving a physical device that might be (impractically) huge, especially when the list size it can handle is large. It compares well with the quantum computer where Grover's quantum algorithm (see [5]) has the same (quantum) time complexity, hence it makes sense to briefly compare these algorithms. First, a common weakness is that in both cases the time complexity refers only to the actual "computational time", i.e. the time necessary to read the input is not taken into consideration. Note that reading the input is not trivial: it requires the preparation of an equally distributed superposition of all possible indices of the items in the list containing the target index in case of Grover's algorithm (takes $O(logN)$ steps) and the set up of beads and their connections in our case (takes $O(N)$ steps). However, if we want the elements of the quantum system to represent an arbitrary database, we need to construct a function which (rapidly) computes the elements of the data base from their indices $1, 2, \ldots, N$; this construction is time consuming. The advantage of our algorithm over Grover's is its deterministic nature: in contrast with the probabilistic nature of Grover's algorithm, a Monte Carlo type of procedure producing fast a probable result (with high probability), our method is guaranteed correct.

7 Acknowledgement

We acknowledge Pulkit Grover for his critical comments on this paper.

References

[1] J. J. Arulanandham. *The Bead–Sort.* Animation, `www.cs.auckland.ac.nz/~j--aru003/BeadSort.ppt`.

[2] J. J. Arulanandham. Implementing Bead-Sort with P systems, *In Proc. 3rd International Conference on Unconventional Models of Computation*, UMC '02, 2002, 115–125.

[3] J. J. Arulanandham, C. S. Calude, M. J. Dinneen. Bead–Sort: A natural sorting algorithm, *EATCS Bull.*, 76 (2002), 153–162.

[4] L. Giavitto, J. Cohen, O. Michel. MGS simulation of Arulanandham–Calude–Dinneen Bead–Sort, `http://www.lami.univ-evry.fr/mgs/ImageGallery/mgs_gallery.html#beadsort`.

[5] L. K. Grover. A fast quantum mechanical algorithm for database search, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, 1996, 212–219.

[6] Mark Sams. The SquareList Data Structure, *Dr. Dobb's Journal*, pages 37–40, May 2003, `http://www.ddj.com`.