UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

An Implicit Data Structure for the Dictionary Problem that Runs in Polylog Time

J. Ian Munro

Data Structuring Group
CS-84-20

August, 1984

AN IMPLICIT DATA STRUCTURE FOR THE DICTIONARY PROBLEM THAT RUNS IN POLYLOG TIME †

J. Ian Munro

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada

ABSTRACT

We introduce a data structure that requires only one pointer for every k data values and permits the operations search, insert and delete to be performed in $O(k \log n)$ time. This structure is used to develop another that requires no pointers and supports insert, delete and search in $O(\log^2 n)$ time.

---

† This work was supported by NSERC grant A8237 and, while the author was visiting the University of Washington, NSF grant MCS7609212A.

1. The Problem and the Results

Pointers are routinely used to indicate relationships between keys in a data structure. Their use is very often crucial in implementing flexible and efficient algorithms. Their explicit representation, however, can contribute significantly to the space requirements of a structure. A natural question to ask is whether pointers are inherently necessary for structures as efficient as AVL [1] trees in maintaining a dictionary (operations of insert, delete and search). One might think so; however, we note that the heap [2,8] is an ideal structure for a priority queue, requiring no pointers and work of the order of information theoretic lower bound to maintain this data type. In this paper we focus on implicit (or pointer free) data structures for the dictionary problem. This problem was first explicitly studied by Munro and Suwanda [6]. They noted that, by using the "usual" pairing function, n data values could be stored in consecutive locations but be thought of as occupying the locations above the minor diagonal in a square array. By maintaining the data in this way, they showed searches and updates can be performed in $O(n^{1/2})$ time and the structure can grow or shrink smoothly. They also showed that, if the elements are kept in any fixed partial order by their values, the product of search and update times must be $\Omega(n)$, i.e. their structure is more or less optimal in that class. They concluded by suggesting the idea of storing components of a structure in an arbitrary cyclic shift of sorted order. Such an organization is not a partial order, and they were able to reduce the search time to $O(\log n)$ with an insert/delete cost of $O(n^{1/2} \log n)$, or alternatively $O(n^{1/3} \log n)$ for any operation. Frederickson [5] extended the use of rotated lists to achieve $O(\log n)$ search and $O(2^{\sqrt{2 \log n}})$ insert and delete time. This paper solves the main problem left open in [6] by demonstrating that there is an implicit data structure for the dictionary problem that runs in polylog time. The search time is a bit more than Frederickson's, but the update cost is a dramatic improvement. In this work, as in previous work on the problem it is assumed all data values are distinct. This assumption is not necessary for the proof of Theorem 2, but is required for the titular result, Theorem 1.

There are two related results in this paper

Theorem 1: *There is an implicit data structure that supports insertions, deletions, and searches in a worst case time of* $O(\log^2 n)$.

This is based on what one might call a semi-implicit structure, namely

Theorem 2: *There is a data structure that requires only one word of structural information for every* k *data values and supports insertions, deletions and searches in* $O(k \log n)$ *time.*

In the next section we will outline our semi-implicit structure (k data values per pointer where k is parameterizable). In section 3 the update procedures are described, and in section 4 it is shown how this can be converted to a fully implicit structure.

2. The Structure

A natural approach to the semi-implicit version of our problem is to lump r ($r = \Theta(k)$) consecutive data values into a single node along with a constant number of pointers, flags and counters. These nodes can be arranged in an AVL tree [1] or some similar structure. Searches, of course, are trivial. The difficulty is that, as elements are added, single values, rather than r consecutive values, have to be appended. We will outline an interesting way to overcome this problem that has eluded previous investigations. A key insight, as we shall see, is to avoid the temptation to divide and conquer in the usual way, but simply to "subtract and conquer".

The data structure is depicted in Figure 1. Each node, except perhaps the bottom one, contains precisely r data values in sorted order. The structure consists of an AVL tree (which we will refer to as level 0) and a sequence of (generally) shorter linear doubly linked lists of nodes of the same format as the AVL tree. The lists can be viewed as levels 1 through $O(\log n)$. The data values in any node are of contiguous ranks among the values at that or any subsequent level. The crucial invariant on the structure is:

> *The elements in the leftmost node at any level precede all those at subsequent levels. There are at least r (except for the bottom level) and at most $3r-1$ data values in subsequent levels that are greater than the values in a given node but less than those in the following node at its level (or following the rightmost node at any level).*

From this invariant it follows that at least a quarter and at most half of the elements at or below a given level are actually at that level. It follows, then, that there are between $\log_2 n$ and $\log_{4/3} n$ levels.

Each node will contain a pointer to the node at the level immediately below that contains the smallest value greater than those in the given node or is immediately to the left of the gap through which this value falls. It is also helpful to retain the number of elements falling in the gap to the right of the node. The space requirements per node, in addition to data, are easily met with 3 pointers, 1 counter and 4 flags. If one believes that a pointer and a counter have the same

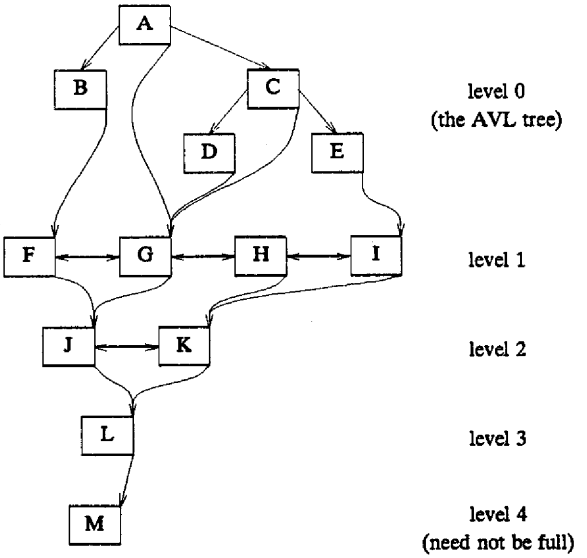

Figure 1 - The Data Structure

Notes: Labels on nodes are for purposes of the explanations below and are unrelated to the key values. Gap pointers are denoted by the splines originating in the bottom right corners of nodes.

Invariants: The values in any node are of contiguous ranks among those at or below their level. The nodes at each level are linked (as binary search tree or linear list) in increasing order by data values. There are at least r and at most $3r$-1 values lying between those in consecutive nodes (such as A and D) except perhaps between the bottom two. These values lie along the vertical path from the leftmost of the pair to the bottom node (A to M) or within 2 nodes to the right of this path.

The Bottom Level: By convention, the node at the bottom level is the only node in use which need not be completely full. This permits smooth growth and shrinkage of the structure, as it is the only node which can be deleted or created.

cost, this is exactly twice the cost of the structural information per *value* in an AVL tree.

With this information, it should be easy to see that:

Lemma 2.1: *The structure outlined above supports searches in* $O(\log n)$ *time.*

3. Updates

Insertions and deletions are most easily viewed as two phase operations. In the first phase, the update is actually made, and the gap sizes are updated, and so the invariant may be violated along a path ($O(\log n)$ nodes) through the structure. The second phase re-establishes the invariant.

More specifically, if an element is to be inserted, then on the first pass a path through the structure is taken as if a search were being performed (gap sizes are also incremented along the search path) until the appropriate position is found for the value. If this occurs in a full node (i.e. not in the bottom node or in the bottom node when it is full) the data values in that node are shifted to accommodate the new value and displace the largest in the node. The insertion procedure continues from this point with the displaced value. The first phase of the deletion procedure is completely analogous, as gap sizes are decremented and the discovery of the element to be removed from a specific node initiates the transfer of the first element in the gap following that node into the node.

Before describing the second or rebalancing phase in detail, the choice of gap size should be explained. A simplified version of the rebalancing will then be given before presenting the faster version.

The "original" idea was simply to restrict gaps from containing more than r data values. Unfortunately, each node should "know" where its gap begins, even if there are no elements in the gap. If there were no elements in a gap, then an arbitrary number of "empty gaps" could "begin" in the same node. If this node is moved, an unbounded number of pointers would have to be changed. Insisting on gap sizes of at least 1 means at most $O(r)$ nodes at any level refer to the same node at the level below. Insisting on at least r, and so at most $3r-1$ (an element short of 2 minimal gaps plus one node full), implies at most 4 nodes at a level refer to a given node at the level below. Hence each node in the list structures could have back pointers to the nodes above that refer to it. This may simplify the process but is not necessary and we will not make explicit use of it.

In a binary search tree the basic tool for local rebalancing is the rotation. In

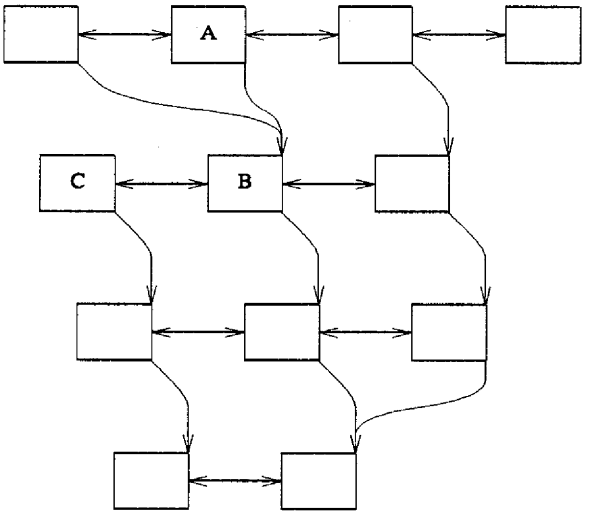

Figure 2 - Part of the Structure Before Rebalancing
Forced by Gap A Being too Large

our structure, the analogous tool is that of moving a node up one level to reduce a gap and down one level to increase one. Observe the effect of an upward move on gap sizes. In Figure 2, the values in node B together with those of gap A (the gap following node A) are of contiguous ranks among those at or below the level of B. Assume gap A violates the invariant by being of size at least $3r$. The middle r elements of gap A are moved into B, and those displaced from B together with others from gap A are put in increasing order in the remainder of gap A. Now B is moved up to be the right neighbour of A and the links are established appropriately, as suggested in Figure 3. Observe there are no changes in gap sizes except

$$|\text{new gap A}| = \lfloor \tfrac{1}{2}(|\text{gap A}| - r) \rfloor$$
$$|\text{new gap B}| = \lceil \tfrac{1}{2}(|\text{gap A}| - r) \rceil$$
$$|\text{new gap C}| = |\text{gap C}| + |\text{gap B}| - r$$

If $3r \leq |\text{gap A}| \leq 7r-2$ then new gap A and new gap B are of valid sizes, while new gap C can be as large as $6r-1$. (The insertion could have left gap B with $3r$ elements and gap C with $3r-1$.) The reshuffle may be applied to the new gap of C, and subsequent levels if necessary. The cost of moving node B is $O(r)$ for actual data moves plus $O(\log n)$ to scan down in search of gap

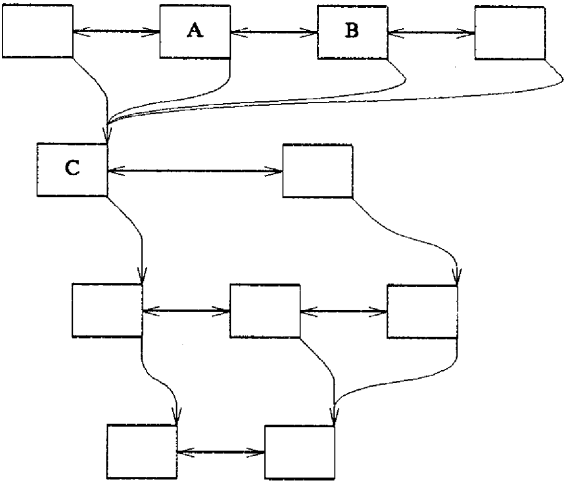

Figure 3 - The Same Portion of After Shifting
Values and Promoting Node B

elements. Since the rebalancing may have to be applied at each level, this leads to an $O(r \log n + \log^2 n)$ insertion procedure.

As it happens, this is adequate to achieve the main result, Theorem 1. However, to demonstrate the bound claimed in Theorem 2, we must avoid the $\Theta(\log n)$ traversals of a significant portion of the original search path. An extra preprocessing pass can be used to sort all potentially movable elements, and so avoid the problem. Recall that only gap sizes along the search path can be invalidated by the first pass. Note, however, that as a result of moving a node up a level, a new gap of up to $6r - 1$ values could be created. It will be necessary to access those up to and including the middle r elements of that gap. These elements are found in the next node on the search path, in that node's two right neighbours, and in the gaps following these three nodes. Relevant elements that fall into any of these gaps will, ultimately, be found somewhere along the search path, or within two nodes to right of it. It is necessary, then, and also suffices, to consider the nodes and values along a column of width (at most) three nodes, bounded to the left by the search path. This sort is achieved in linear time by a simple bottom up procedure. At each stage, we retain a reference to the leftmost element inserted at the most recently added level. Adding a new level consists of inserting three nodes each consisting of elements of contiguous ranks as the

structure is currently constituted. Furthermore, these elements fit within $O(1)$ nodes of the retained reference. Some reshuffling of individual values will occur, as the new elements are hooked in. However, the step at each level requires $O(1)$ pointer moves and inspections, and $O(r)$ data inspections and moves. The full sorting process will take $O(r \log n)$ time, as claimed, with only $O(\log n)$ pointer inspections or changes.

In order to restructure the column back into the data structure, some information, in addition to data values, must be retained during the sort phase. This includes the sizes of the gaps following the (at most) three nodes at each level that may be altered. Designate these as `g1`, `g2` and `g3`. Furthermore, a reference to the node to the right of the column (if any) is required, as this could be the only pointer to such a node. This information can be retained in a linear linked list using pointers, counters and flags not required for the sorted data. (Note this list can be viewed as a sequence of records of information about levels from top to bottom. Two independent linked lists, the data list and this level structure list, share the same set of nodes.) Some other structural information will be required, but it can be derived as the reorganization pass moves from one level to the next. This includes:

`left`, a reference to the node immediately to the left of the path node (`left = null` if the path node is leftmost on its level)

`g0`, the size of the gap to the left of the path node (`g0 = 0` if `left = null`)

`path`, a flag set to `false` if the path node has been moved up to the next level (as in the $\Theta(\log^2 n)$ scheme) and `true` if it is still in the column. `path` is initialized to `true` for the top level

`lp`, a reference to the smallest data value that is still in the data list and is greater than the largest element in node `left` (`lp` refers to the smallest remaining value if `left = null`)

Having performed this preprocessing and performed the initialization for the top level, the rebalancing proceeds top down, and more or less as suggested in the $\Theta(\log^2 n)$ scheme. The outline below provides a high level description of the procedure. The function `insertnode(xx)` indicates that the `r` data values from position `xx` through `xx + r - 1` are to be shifted into a node that is logically removed from the data linked list and inserted in the appropriate place in the data structure. It should be obvious from the context where the node should go and what pointers must be updated. The tedious details are omitted. After the execution of `insertnode`, the parameter `xx` refers to the data value immediately following the last one moved into the main structure. References to data values will be treated as integers. incrementing such a pointer moves the reference to the next data value that may or may not be in the same node.

```
begin
  for each level of the structure from top
    to bottom do
  begin
    if path or (left¬=null) then
    begin
      if path then
      begin
        align:=lp+g0;
        insertnode(align)
      end
      else g1:=g0+g1;
      if g1>=3r then
      begin
        a1:=align+(g1-r)/2;
        insertnode(a1);
        align:=align+g1-r;
        path:=false
      end
      else begin
        align:=align+g1;
        path:=true
      end
    end
    else begin
      insertnode(1);
      if (g1+g2)>=3r then
      begin
        a1:=(g1+g2-r)/2;
        insertnode(a1);
        align:=g1+g2-r;
        path:=false
      end
      else begin
        align:=g1;
        path:=true
      end
    end;
    working from position align apply
      insertnode() to the 2 right
      neighbours of the path node
      (if 2 neighbours exist);
```

```
    reset lp and draw new g0,g1,g2 and g3
      values;
  end;
  if a node was promoted to the top level
    then apply the appropriate AVL (or
    other) strategy to restore balance
end.
```

Some brief mention should be made of the `else` clause executed when `path=false and left=null`. Under this condition, the leftmost r data values must form a node at the current level, but taking those elements may leave too small a gap before the "expected position" of the right neighbour of this new path element.

Each iteration through the procedure outlined above requires $O(r)$ data moves and $O(1)$ pointer or counter inspections and manipulations. The total balancing cost is, then, $O(r \log n)$ time including $O(\log n)$ pointer or gap counter operations.

The deletion process is analogous to the insertion procedure. We have already sketched the first pass. The second or rebalancing phase may move nodes down in the structure if gap sizes drop below r. A slight twist is necessary, in that if a gap becomes too small (size $r-1$) and the next gap to the right is not of minimal size (i.e. more than r elements) then we simply shift an element from the intervening node into the small gap, and move the net larger element into that node. This local patch requires only $O(r)$ time (given the sorting pass) and alters no gaps below. Demotion of a node occurs only if we have a subminimal and a minimal gap together. The demotion produces a gap of size $3r-1$.

Assuming a pointer and a counter each take a word of memory, and that 4 flags can be encoded in one word, the choice $r = 5k$ leads to Lemma 3.1, and so with Lemma 2.1, Theorem 2 follows.

Lemma 3.1: *The structure outlined above requires only n/k words of structural information and supports insertions and deletions in $O(k \log n)$ time including only $O(\log n)$ operations of following or changing pointers.*

4. A Fully Implicit Structure

Setting r to $8 \log n + 8$ we can construct an implicit structure using data values to encode pointers. There are a couple of key points in this construction. First, pointers are represented by a sequence of $2 \log n$ data values. The sequences are stored more or less in sorted order, but odd-even pairs of values are

stored in increasing order to denote a 0 and decreasing order to denote 1. Reading or updating a pointer, then, takes $O(\log n)$ time. The suggested value of r permits 3 pointers, a gap counter and 4 flags per node. Assuming $\log n$ does not change very much due to updates, this gives us $O(\log^2 n)$ algorithms for the basic operations.

The second issue is handling the growth or shrinkage of n. This is done by maintaining $\log \log n$ distinct structures of sizes 2^{2^i} (i = some constant, ... $\lfloor \log \log n \rfloor$) plus one more of the appropriate size (all logarithms used are taken to base 2). In any such structure the logarithm of the number of elements in it and all smaller ones is fixed to within a factor of 2, and hence the node size may be fixed. The cost of a search in each of these structures is dominated by that of a search in the largest, due to the double exponential growth of the structure size. Insertions are always made on the last structure, and a deletion is accomplished by moving an element from the last structure into the one in which the element to be deleted is found. A final detail is to adopt the convention that the bottom (partial) node occupy the last few locations of memory in use. This permits smooth growth and shrinkage.

From Lemmas 2.1 and 3.1, Theorem 1 follows.

5. Conclusions

Do pointers give you more than simpler code and a constant factor in run time? Granted these are extremely important features, but one feels pointers must give more in the dictionary problem. I make no conjecture either way. Suwanda and I were genuinely surprised when we realized that organization other than a partial order was crucial. A few researchers have conjectured that Frederickson's scheme was optimal and that no polylog solution for the dictionary problem existed. He has demonstrated its optimality in a restricted class [4]. The space saving construction is interesting, and its essence may be of use in other environments. The fully implicit structure, however, is the main result. Given that it effectively uses pointers, but charges $O(\log n)$ for each use, one sees that the approach we have taken will go no farther.

A few other results follow quickly. If data values have explicit probabilities of access, one can take advantage of our structures (particularly the $\log \log n$ separate structures for the fully implicit version). The sequence of structures of doubly exponentially increasing size can also be used if the data values come from underlying but unknown distribution. A "self organizing implicit" structure along the lines of Frederickson's [3] can be formed.

There are several related, interesting, and perhaps tractable problems.

- Give a better upper bound on the number of pointers necessary to support $O(\log n)$ searches and polylog updates. Can this be reduced to, say, $O(n^{1/2})$?
- Show, as Allan Borodin has suggested, that no implicit structure can support searches in $O(\log n)$ time while requiring only a constant (!) number of *moves* for an update.
- Give an easily implementable polylog solution to the implicit dictionary problem. This would be of interest even if the bound held only for the average case. Munro and Poblete [7] have suggested a candidate, but the analysis of its behaviour remains open.

6. References

[1] Adel'son-Vel'skii, G.M., and Landis, Y.M., An Algorithm for the Organization of Information, *Dokl. Akad. Nauk. SSR 146*, 263-266.

[2] Floyd, R.W., Algorithm 245: Treesort 3, *CACM 7*, 701 (1964).

[3] Frederickson, G.N., Implicit Data Structures for the Weighted Dictionary Problem, *Proc. 22nd IEEE Symp. on Foundations of Computer Science*, 133-139, Oct. 1981.

[4] Frederickson, G.N., Recursively Rotated Orders and Implicit Data Structures: A Lower Bound, Tech. Rep. CS-82-01, Dept. of Computer Science, Pennsylvania State University, University Park, Pa., 1982.

[5] Frederickson, G.N., Implicit Data Structures for the Dictionary Problem, *JACM 30*, 1, 80-94 (1983).

[6] Munro, J.I., and Suwanda, H., Implicit Data Structures for Fast Search and Update, *J. Computer Syst. Sci 21*, 236-250 (1980).

[7] Munro, J.I., and Poblete, P.V., Searchability in Merging and Implicit Data Structures, *Proc. 10th ICALP*, 527-535, July 1983, appears as *Lecture Notes in Computer Science 154, Springer-Verlag.*

[8] Williams, J.W.J., Algorithm 232: Heapsort, *CACM 7*, 347-348 (1964).