American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN : 2320-0936
Volume-5, Issue-10, pp-244-249
www.ajer.org
Research Paper Open Access

Maintaining Dictionaries under Memory Access Restriction

Dr. Mirza Abdulla
Computer Science Department, AMA International University, Bahrain

ABSTRACT: *We consider the implementation of the dictionary data structure on Random Access Memory (RAM) machines with nonuniform access cost. More specifically, an access to the ith location of the RAM costs $f(i)$, where fis some nondecreasing function. Block data movement can also be catered for by allowing the access cost to a block of data be a function of the length of the data segment as well as $f(i)$, where i is the highest location in that block. This model of computation can capture the cost of data movement, or the I/O complexity of algorithms as well as their computational complexity. We give optimal and efficient worst case lookup and amortized update times under various access costs. We also show how to generalize the results to hold for a wide class of access costs, and give interesting worst case lookup/ amortized update times tradeoff for some of the access costs.*
Keywords: *Dictionary, trees, forests, hierarchical memory*

I. INTRODUCTION

The computational complexity of most algorithms is based largely on the RAM model of computation; the running time of the algorithm depends on the number of arithmetic operations, comparisons made and variable assignments. The cost, unfortunately, ignores the time to access the operands of an operation.

In modern computers, where large data bases and multi-level memory systems are used, data are fetched from slow memory, where data bases usually reside, and moved to fast memory, before any action is taken on these data. The data items are brought to fast memory in blocks, to offset the cost of accessing the slow memory. On the other hand, bringing only one data item from any memory level to fast memory, as the RAM model permits, can have a detrimental effect on the running time of algorithms.

[1], [2], and [3] were the first, to our knowledge, to consider this problem with the RAM model from a theoretical point of view. [1] consider 2 memory level machines, [2] consider hierarchical memory machines, where access to location i costs $f(i)$. [3] improve upon the latter machine model by allowing block transfer cost to be a function of the length of the block plus the access cost to a location in that block. Basically, their model is defined as follows.

DEFINITION [3]. The Hierarchical Memory Machine (HMM) is a Random Access Machine (RAM) operating under the following conditions:

I. The access cost to any location, *i*, costs $f(i)$, where f is a non-decreasing function.

II. The cost of moving a block of length l in locations $[x-l, x]$to locations $[y-l, y]$costs $f(x)+f(y)+1$.

[4] on the other hand made a slight modification to the above definition.

DEFINITION [4]. Given f *an* invertible function, then define $f^{j}(n)$as follows:

a) If f is a sublinear function, then

$f^{j}(n) = n$ for $j = 0$,

$f^{j}(n) = f(f^{j-1}(n))$ for $j > 0$, and

$f^{j}(n) = f^{-1}(f^{j+1}(n))$ for $j < 0$.

b) If f is not a sub linear function, then $f^{j}(n) = f(n/2^{j})$.

DEFINITION [4]. The Hierarchical RAM (HRAM) running under access cost $f(n)$, $f$nondecreasing invertible function, is a RAM, with the exception that access cost to location i, costs $f^{j}(n)$, where $f^{j}(n) < i < f^{j-1}(n)$, when $f$is a sublinear function. If $f$isnot a sublinear function in n, then the access cost to location i is $f(n/2^{j})$, where $n/2^{j} \leq i < n/2^{j-1}$.

In this paper, we will consider the implementation of the dictionary data structure on the HRAM machine. The dictionary is a data structure that supports lookup, insertion and deletion operations. Other operations such as replace can be incorporated as a delete followed by insert. The lookup operation searches for a record whose key matches the given key and returns a yes/no answer, depending on whether the a matching record is found or not. For this operation a prompt response is required, and therefore, the worst case performance of this operation will be studied. The insert operation inserts a new record into the data structure. If a record with the same key already exists, then the insert is treated as a replace operation. The delete operation deletes a record with a matching key from the data structure. If no such record exists, the delete operation behaves as a no operation. Neither the insert, nor the delete operations have to return a response, and therefore, amortized time analysis will be given to these two operations.

The dictionary problem renders itself as a suitable problem to be studied since large data bases are becoming the norm with the ever increasing power of computing. Most of these large data bases reside in slow access memory devices, calling for algorithm time analysis to take into account the I/O complexity of data transfer as well as their computational complexity.

[3] First studied the implementation of dictionaries on the HMM machine. Unfortunately, some of their time bounds were not optimal. In this paper, we improve upon their bounds and provide optimal worst case lookup and amortized update times under various access costs, using the HRAM model of computation. We will be mainly interested in the access costs $\{\log n,\ n^{\alpha}$ for $0 < \alpha < 1,\ n/\log n, n\log n\}$, but the bounds obtained can be shown to hold for a wide class of access costs. Next we will consider HRAMs operating under access cost $f(n) = n^{\alpha}, 0 < \alpha < 1$.

II. THE DATA STRUCTURE

The data structure is built from a sequence of update entries. Each update record is stored together with an indicator field to indicate if it is an insert or a delete entry, and a time stamp. The structure consists of $k = O(\log n / \log\log n)$forests; $F_0, F_1, \ldots, F_k$, where n is the number of update entries in the structure. Entries in each forest F_i, $0 < i < k$, are more recent than those in forest F_{i+1}. There are two types of forest used:

a) Type 1 forests: Each forest F_i of this type consists of only one tree. The size of forest F_i, $|F_i| = 2|F_{i-1}|$.

b) Type 2 forests: Each forest F_i consists of Ign trees. The size of each type 2 forest, F_i, $|F_i| = \log n|F_{i-1}|$. The smallest type 2 forest, $F_{1+\log\log n}$, has size $(\log n)^2$, and the largest type 2 forest has size $|F_j|$, $n/(\log n)^{1+\frac{1}{\alpha}} < |F_j| < n/(\log n)^{\frac{1}{\alpha}}$. Forest $F_{1+\log\log n}$until F_j are all type 2 forests. All the other forests in the structure are type 1 forests.

The reader should note, that from now on we will be using j to refer index of the last type 2 forest, and k to refer to the index of the last forest in the structure.

All the trees in the structure are m-ary trees, where the value of $m$will be defined later. These trees are like B-trees in the sense that the records are held at the leaves, and all leaves of a tree are at the same level, except possibly the right most branches in the most recent tree in a forest, which may contain less data or keys. The internal nodes of these m-ary trees are used only as path finders, for the lookup operation. The internal nodes hold the address of each child and the maximum key value that subtree holds. All entries in the trees are distinct, but different trees (even those belonging to the same forest) may carry duplicate entries. Moreover, the leaves are marked as such to distinguish between them and internal nodes. All trees that belong to the same forest are of the same size, except possibly the last which may contain less. The value of m for each m-ary tree depends on the size of the tree. For a tree of size S, say, we have $m = S^{\alpha}$. All trees have $|1/\alpha|$ levels. Finally, as an implementational detail, the forests are laid in the memory tape so that F_i appears in lower access cost locations than $F_{i+1}, 0 < i < k$. Any other information, such as the index of the first and last type 2 forests is held in fast memory.

III. OPERATIONS ON THE DATA STRUCTURE

Lookup Operation

The trees, starting from the most recent tree, of each forest, F_i , $i = 0, 1, 2, \ldots, k$, is searched and the indicator field of the first match found is checked. If it is an insert record, then a **YES** reply is reported, otherwise a **NO** reply is reported. The code is as follows.

1. Get value of j{index of last type 2 forest} and k{index of last forest in the structure} from fastmemory.
Set answer = **NO**;
2. *for* i =1 to $\log\log n$**do**
search type 1 forest F_i;

if match is found
then if indicator field = INSERT ***then*** answer = YES;
od;
3. ***for*** $i = 1 + \log\log n$ **to** j **do**
Set **s = log *n*;**
while s> 0 do
search tree $T_{i,s}$ {tree s of forest F_i };
if match is found
then if indicator field = INSERT
then Set answer = YES;
Sets = s-1;
od
od,
4. **for** $i = j + 1$ **to** k ***do***
search type 1 forest F_i;
if match is found
then if indicator field = INSERT ***then*** set answer = YES;
od;
5. **return**(answer);

Update Operation

To each update entry, we add two extra fields: A time stamp and an indicator field that indicates whether it is an insert or a delete request. The update entry is merged with forest F_1 , and if an overflow occurs the result is merged with forest F_2 , and so on. Care must be taken when dealing with type 2 forests. If the type 2 forest is not full, i.e., does not have $\log n$trees in it, then add resulting tree as one of the trees of this forest. Otherwise, merge all $\log n$ trees of the forest plus the overflow data. The code for the update follows.

1. Augment the record with time stamp and insert/delete indicator. Call the result temp;
2. **Set**overflow = **True, Set*i* = 0;**
3. ***While*** overflow **do *begin***

temp = merge(temp, F_i);
if |temp| < | F_i|
then *overflow* = **False,**
F_i = temp*;* **end, od,**
Note: *In the above code* IXI *means size of X.*

Re-Placement of Structure

Each time the structure grows from n to n^2 or shrinks from n to $\sqrt{n}$ all the forests in the structure are re-laid according to the new value of $2log\, n$ or $(log\, n)/2$ respectively). Similarly, the first type 1 forests will increase by one or decrease by one, respectively.

This can be done by sorting all the entries according to record key and time stamp. Of these records, only those whose most recent entry is an insert are kept. All others are deleted. The records are then placed as trees of forests of type 1 and 2, as explained earlier.

IV. COMPLEXITY ANALYSIS

The analysis makes use of the following theorem by[4].

Theorem 1. [4]On a HRAM machine with access cost $f(n) = n^{\alpha}, 0 < \alpha < 1$,

a) n items can be read (or brought to location 0) in $O(n \log\log n)$ amortized time,
b) $\log n$ sorted lists, each of length $n/\log n$ can be merged in $O(n \log\log n)$ amortized time,
c) An intermix of n pop/push operations on a stack or a queue can be performed in $O(n \log\log n)$amortized time,
d) Sorting a list of n items can be performed in $O(n \log n)$ time, and
e) Any fixed number of data structures can be maintained without affecting the running time complexity by more than a constant factor.

■

Moreover, it is helpful to give the following lemma, before embarking on the worst case lookup time analysis.

Lemma 2. If accessing an item in a tree in the structure costs $O(n^{\alpha})$ time in the worst case, then the whole tree can be searched in $O(n^{\alpha})$ time in the worst case.

Proof. Bring the root of the tree to fast memory, and perform binary search on it to locate the subtree to be searched next. In performing binary search, we bring the half in which we expect the record to reside to fast memory. This idea is then applied recursively on the root of that subtree, until we reach a leaf, which can be searched to see if the record exists or not in the same manner.
Bringing a node to fast memory should not cost more than $O(n^{\alpha})$time since the length of the node is n^{α} and the worst case access cost to an item in the tree is $O(n^{\alpha})$, the result follows.

■

Lemma 3. *Searching all trees in the dictionary data structure costs no more than $O(n^{\alpha})$ time in the worst case, where n is the size of the data structure.*
Proof. The access cost to any element in the largest type 1 forest in the structure is no more than n^{α}. Now by lemma 2, it follows that searching the largest type 1 forest costs no more than $O(n^{\alpha})$. Moreover, the next largest type 1 forest is smaller by at least a constant factor (2), and therefore the access to that forest is a constant factor less than that of accessing the largest type 1 forest. This argument can then be carried all the way to the smallest type 1 forest. Summing all these times forms a geometric series, which totals to at most $O(n^{\alpha})$time in the worst case.

Next consider the largest type 2 forest in the structure. This forest has no more than $n/(lgn)^{1/\alpha}$records. The next largest type 2 forest is $\log n$ factor smaller. Adding up the space usage for all these forest, we get a value $< 2n/(lgn)^{1/\alpha}$. Thus the worst access time to any item in the largest forest is no more than $(2n/(lgn)^{1/\alpha})^{\alpha} = O(n^{\alpha}/\log n)$. . Type 2 forests can carry *Ign* trees. Thus the overall time to search a type 2 forest is no more than $O\big((n^{\alpha}/\log n)\log n\big) = O(n^{\alpha})$. Now we can apply the same argument as that for type 1 forests to conclude that searching all type 2 forests costs no more than $O(n^{\alpha})$time in the worst case.
Thus the overall time to search all the trees in the structure is bounded above by $O(n^{\alpha})$in the worst case.

■

Theorem 4. *Given* a *dictionary of size n, then worst case lookup time is $O(n^{\alpha})$.*
Proof. The lookup algorithm searches all the trees in the forests starting from F_1 and working its way up to the last tree in the data structure. By lemma 3, this time is bounded above by $O(n^{\alpha})$. In addition,
the algorithm correctly returns a YES response if the record exists. This follows from the fact that the most recently added trees to a forest are searched first, and lower numbered forests are searched first also. Finally, the lower bound follows from the fact that accessing the *nth* data item requires at least n^{α}time.

■

Next, we consider the update operation. The following lemma is helpful.
Lemma 5. *The size of all forests, from F_1 up to and including F_{i-1}, is no more than a constant factor greater than maximum size of any tree in forest F_i , $i \leq k$, and F_k is the largest forest in the structure.*
Proof. If F_i is a type 2 forest, then it is greater by $\log n$ than F_{i-1}. Since type 2 forests can have $\log n$trees, it follows that the maximum size of any tree in F_i is to within a constant factor of the size of forest F_{i-1}. Because, the sum of size of all forests F_1 to F_{i-2} itself is not greater than the size of forest F_{i-1} by more than a constant factor, as the reader may easily verify. Thus the result holds for type 2 forests. Type 1 forests on the other hand, carry only one tree, whose size is at least twice the preceding forest. The sizes of these forests form a geometric series whose sum is not greater than the size of the single tree in $F_i$by more than a constant factor.

■

Theorem 6. *Starting from an empty dictionary, any intermix of n insert and delete operations can be performed in $O(n\log n)$ time in the worst case.*
Proof. Consider the update algorithm given earlier. By lemma 5, the size of all forests preceding F_i is no more than a constant factor greater than any tree in forest F_i . Hence, by Theorem 1 (c), merging all the trees in these forests and the trees in forestF_i , would cost no more than $O(m\log\log m)$, where $m$is the size of forest F_i . Thus, even if charges are made only to the entries in forest F_i , no entry would incur more than $O(\log\log m)$amortized cost for the merge. Now, the reader may easily verify that the update entries in an over-flown forest end up in a higher numbered forest, and that there are at most $O(\log n / \log\log n)$forests in the structure. Thus, it follows that no entry incurs more than $O(\log n)$amortized time, from which the theorem follows. ■

Finally, we show that the re-laying of the structure every time it grows from $n$to n^2or shrinks from $n$to$\sqrt{n}$, does not cause any increase in the asymptotic behavior of the algorithm; i.e., theorem 6 still holds. Here, the cost of re-placing the structure is dominated by the time to sort the entries. By theorem 1 (e), this time is $O(n\log n)$for $n$entries. This amounts to $O(\log n)$time per entry. Now, the dictionary can grow from $\sqrt{n}$to $n$or shrink from $n$to$\sqrt{n}$ only as a result of $n-\sqrt{n} = O(n)$new update entries (insert, and delete respectively). If only these new entries are charged for the replacement of the structure, then none will incur more than $O(\log n)$cost. One last detail, the dictionary as it grows from an empty one to one that contains

$n$entries, may go thru many re-placements in memory. The reader may easily verify, that in the case of dictionaries shrinking, if we make the delete operations pay not only for the sorting this time, but also for the preceding growth from$\sqrt{n}$to n, then the overall time for the re-placement of the structure is no more than $O(\log n)$time per entry. Thus we have.

Theorem 7. *The dictionary data structure can be maintained on a HRAM running under access cost* $f(n) = n^{\alpha}, 0 < \alpha < 1$, *so that lookups can be performed in* $0(n^{\alpha})$ *time in the worst case, and updates in* $O(\log n)$ *amortized time, after* n *operations on an empty structure.* ■

Next, we consider HRAMs running under access costs $f(n) = n/(lgn)^{\beta}$ and $f(n) = n(lgn)^{\beta}, \beta > 0$. The same algorithm and analysis given earlier will still hold, except that the bounds would be different. This follows from the results given by [4].

Theorem 8. [4] *On a HRAM machine with access cost* $f(n) = n/(lgn)^{\beta}, \beta > 0$.

a) n *items can be read (or brought to location* 0*) in* $O(nS(n))$ *amortized time,*
b) $\log n$ *sorted lists, each of length* $n/\log n$ *can be merged in* $O(nS(n))$ *amortized time,*
c) An intermix of n *pop/push operations on a stack or a queue can be performed in* $O(nS(n))$ *amortized time,*
d) Sorting a list of n *items can be performed in* $O(nT(n))$ *time, and*
e) Any fixed number of data structures can be maintained without affecting the running time complexity by more than a constant factor.

Where $S(n) = \log n / \log\log n$ *for access costs* $f(n) = n/(lgn)^{\beta}, \beta > 0$, *and* $S(n) = (lgn)^{1+\beta}, \beta < 0$. *Also* $T(n) = (\log n / \log\log n)^2$ *for the former access costs, and* $T(n) = (lgn)^{2+\beta}, \beta < 0$ *for the latter.* ■

We leave it to the reader to verify the following.

Theorem 9. *The dictionary data structure can be maintained on a HRAM, so that lookups can be performed in* $O(f(n))$ *time in the worst case, and updates in* $O(T(n))$ *amortized time per entry, after* n*operations on an empty structure, where* $T(n)$ *is the function given in theorem 8.* ■

Finally, dictionaries can also be implemented on HRAM running under access cost $f(n) = (\log n)^{\beta}, \beta < 1$, using the same structure given earlier, but at the risk of obtaining a non-optimal lookup time, as is given in the following theorem.

Theorem 10. *The dictionary data structure can be maintained on a HRAM running under access cost* $f(n) = (\log n)^{\alpha}, \alpha < 1$, *so that lookups can be performed in* $O((\log n)^3/\log\log n)$ *time in the worst case, and updates in* $O(\log n)$ *amortized time, after* n *operations on an empty structure.* ■

But the above lookup time can be improved if we use a single B-Tree to carry the update entries. [5] gives the following theorem in this case, which would hold for the dictionary data structure in this case.

Theorem 11. *The B-Tree data structure can be maintained on a HRAM running under access cost* $f(n) = (\log n)^{\alpha}, \alpha < 1$, *so that lookups can be performed in* $O((\log n)^2/\log\log n)$ *time in the worst case, and updates in* $O((\log n)^{1+\varepsilon})$ *amortized time, for some arbitrary constant* $\varepsilon > 0$, *after* n *operations on an empty structure.* ■

Moreover, [6] proved that a dictionary can be maintained on a RAM , so that lookups can be carried out in $O(1)$time in the worst case, but with at most (and at least) $O(n^{\varepsilon})$, for constant $\varepsilon > 0$. Applying this result on a HRAM running under access cost $f(n) = \log n$, we obtain.

Theorem 12. *The dictionary data structure can be maintained on a HRAM running under access cost* $f(n) = \log n$, so *that lookups are performed in* $\Theta(\log n)$ *in the worst case, and updates in* $O(n^{\varepsilon})$, *for constant* $\varepsilon > 0$. ■

The reader can clearly see that there is a worst case lookup/ amortized update time tradeoff. Theorem 10 on the one hand gives optimal update time at the cost of a non-optimal lookup time, and theorem 12 on the other hand gives optimal worst case lookup time but with a non-optimal update time. The lookup bounds given by theorem 11 would be optimal when only trees are considered to carry the structure. The bounds given by theorem 12, on the other hand, would be optimal if only hash tables are considered to carry the dictionary.

V. CONCLUSION

In this paper, we gave and analyzed an algorithm for maintaining dictionaries on HRAMs running under various access costs. These results improve upon those provided by [3] for the access costs they studied ($lgn, n^{\alpha}, 0 < \alpha < 1$). In addition, these results can be generalized to any HRAM running under any

nondecreasing access cost $f(n) > O((lgn)^{\varepsilon})$, constant $\varepsilon > 0$, provided that the time to sort $n$items, $T(n)$, is known for access cost $f(n)$. In such a case it can be shown that the dictionary data structure supporting lookups in $\Theta(f(n))$time in the worst case, and $O(T(n))$amortized update time can be maintained. For access costs $f(n) = O((\log n)^{\varepsilon})$, $\varepsilon > 0$, theorems 10 thru 12 provide interesting worst case lookup and amortized update times tradeoff. It is left as a challenging open problem to find if there exists a structure on which both $O(f(n))$worst case lookup time and $O(lgn)$amortized update time can be maintained, and if not, what is the minimum product of these two times.

REFERENCES

[1]. Aggarwal and J. S. Vitter.*"The Input/Output complexity of sorting and related problems"*. Communications of the ACM, 31(9):1116-1127, 1988.

[2]. A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, "*A Model for Hierarchiecal Memory*", Proceedings of the 19th Annual ACM Symposium on Theory of Computing, ACM Press (1987), 305–314.

[3]. A. Aggarwal, A. K. Chandra, and M. Snir, "*Hierarchiecal Memory with Block Transfer*", Proceedings of the 28th Annual IEEE Symposium on Foundations of Computing, IEEE Computer Society Press (1987), 204–216.

[4]. M. Mirza, "*Data Structures and Algorithms for Hierarchical Memory Machines*", Ph.D. dissertation, Courant Institute of Mathematical Sciences, New York University, 1990.

[5]. Abdulla, Mirza, *"On the I/O complexity of maintaining B-Trees "*, submitted.

[6]. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H.Ronhnert, and R.E. Tarjan, *"Dynamic Perfect Hashing: Upper and LowerBounds"*,29th FOCS, 1988, 524-531.

[7]. Aho, A. V.; Hopcroft, J. E.; and Ullmann, J. D. *"Data Structures and Algorithms*". Reading, MA: Addison-Wesley, pp. 369-374, 1987.

[8]. Bayer, R.; McCreight, E. (1972), "*Organization and Maintenance of Large Ordered Indexes*", ActaInformatica 1 (3): 173–189.

[9]. Beyer, R. "*Symmetric Binary -Trees: Data Structures and Maintenance Algorithms*".ActaInformatica 1, 290-306, 1972.

[10]. Comer, Douglas (June 1979), "*The Ubiquitous B-Tree*", Computing Surveys 11 (2): 123–137, DOI:10.1145/356770.356776,ISSN 0360-0300.

[11]. Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2001), *Introduction to Algorithms* (Second ed.), MIT Press and McGraw-Hill, pp. 434–454, ISBN 0-262-03293-7. Chapter 18: B-Trees.

[12]. Folk, Michael J.; Zoellick, Bill (1992), *File Structures* (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4

[13]. Hong Jia-Wei, H. T. Kung, I/O complexity: "*The red-blue pebble game*", Proceedings of the thirteenth annual ACM symposium on Theory of computing, 1981

[14]. Knuth, Donald (1998), *Sorting and Searching, The Art of Computer Programming*, Volume 3 (Second ed.), Addison-Wesley,ISBN 0-201-89685-0.

[15]. Mond, Yehudit; Raz, Yoav (1985), "*Concurrency Control in B+-Trees Databases Using Preparatory Operations*", VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases: 331–334.

[16]. M. H. Nodine and J. S. Vitter. "*Paradigms for optimal sorting with multiple disks*.", Proc. of the 26th Hawaii Int. Conf. on Systems Sciences, 1993.

[17]. J. D. Ullman and M. Yannakakis. "*The input/output complexity of transitive closure*", Annals of Mathematics and Artificial Intellegence, 1991.

[18]. J. S. Vitter. "*Efficient memory access in large-scale computation*" (invited paper), Symposium on Theoretical Aspects of Computer Science, LNCS 480, 1991.

[19]. J. S. Vitter and E. A. M. Shriver. "*Algorithms for parallel memory, I: Two-level memories*", Algorithmica, 1994.