Contemporary Access Structures Under Mixed Workloads

ALEX DELIS[1] AND QUANG LEVIET[2]

[1]Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201, USA
[2]School of Information Systems, Queensland University of Technology, Brisbane, QLD 4001, Australia
Email: ad@naxos.poly.edu

Modern high-performance computing systems and databases are implemented under the assumption that a very large proportion of the data used can now be maintained in volatile memory. In this paper, we compare experimentally two recently proposed self-adjusting access structures that can be used to organize data in such settings, namely, the Skip-List (SL) and the Binary B-Tree (BB-Tree). We examine the scalability of these two methods against both mixed and pure-query workloads. Our experiments reveal the behaviour of SLs and BB-Trees under diverse environments and varying data requirements.

Received May 9, 1996; revised July 28, 1997

1. INTRODUCTION

A wide variety of applications require main-memory access structures that demonstrate both fast response times and easy maintenance [1–4]. The performance of the Binary Search Tree (BT), a widely used structure, deteriorates significantly to almost O(n) when the structure is created by ordered, partially ordered or skewed input data sets [1, 5, 6]. In realistic settings, such data sets appear often [7–10]. To prevent poor response time, various other classes of trees have been proposed, based on height balance.

One such example is the AVL-tree which guarantees logarithmic search time at the expense of node rotations [11–13]. However, AVL maintenance routines are complicated to implement and the cost of maintaining balance may become rather high in an environment with continuous updates of skewed data items [14, 15]. Hairy trees [16] are built using an elegant insertion mechanism and avoid costly global re-balancing operations. This is achieved by performing local balancing in the vicinity of newly inserted nodes. Deletions could be more difficult to carry out as they may trigger global tree reorganizations.

Confronting similar problems but following a probabilistic approach, Pugh introduced a self-adjusting access method termed the Skip-List (SL) [17, 18]. Analysis indicates that SLs may perform well under skewed data sets and provide shorter response times than AVL, 2–3 and self-adjusting trees [18, 19]. In the worst-case scenario, the SL may produce O(n) access times. However, this is argued to be highly improbable [18].

The BB-Tree access method, introduced in [20], follows a more conventional structural approach to offer fast query-response times. The proposed method is based on a simple observation that limits the number of required restructuring operations. This observation suggests that before breaking up a node, we have to ensure that only right-hand edges are in the same level. In this way, the re-balancing of the tree can be achieved with only two simple operations. The BB-Tree is maintained by a set of elegant and simple-to-implement routines. The two main reported advantages of the BB-Tree are easy coding and satisfactory performance [20].

Recently, a number of studies have analysed issues pertinent to the SL structure from a theoretical viewpoint. Such analyses include the study of the path length, the behaviour of an optimized search algorithm and the development of a limit theory for SLs [19, 21, 22]. However, at this stage and from the experimental point of view, we are not aware of any large-scale study that attempts to evaluate empirically not only the performance of the SL but also to compare it with new promising self-adjusting access methods. Limited experimentation in [20] suggests that the Deterministic Skip-List [23]—a worst-case efficient variant of SL—gives poorer performance than the BB-Tree. Relatively small-size structures (up to 10 000 items) were used in a Pascal and Unix environment. In [20], it is also argued that the SL offers similar results if compared with the BB-Tree.

In this paper, we carry out a large-scale comprehensive experimental study in the Unix environment in order to obtain a clear understanding regarding the merits of both the SL and BB-Tree, and see how these access structures behave under the presence of both queries and updates (i.e. mixed workloads). Large, skewed and mixed-data workloads of varying compositions are used and the sensitivity of the access structures to distributions of input data is examined. Our experience indicates that the BB-Tree offers elegant maintenance routines. If these routines are implemented efficiently, they yield very competitive response times in a number of cases. On the other hand, the SL demonstrates consistently better creation times and improved response times for all experiments that involve mixed workloads with frequent updates. The SL also has minimal space overhead

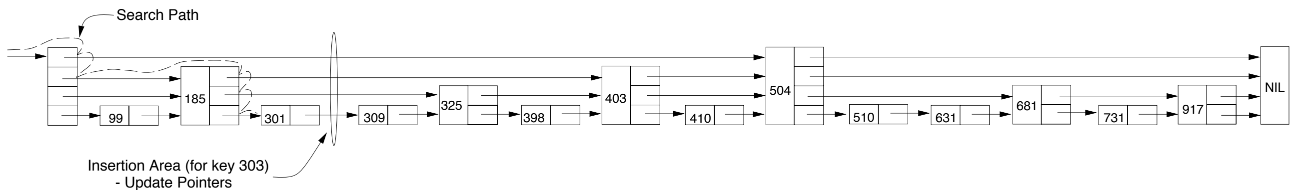

FIGURE 1. The Skip-List structure.

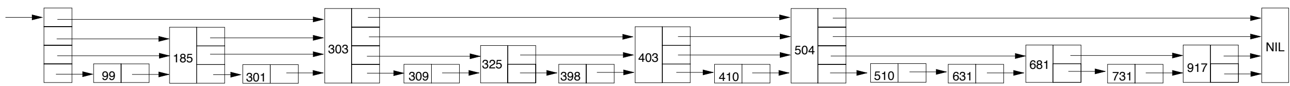

FIGURE 2. The Skip-List structure after the insertion of item '303'.

requirements. Measurements with regular BTs are used as the baseline case whenever necessary.

This paper is organized as follows. Section 2 outlines the main features of the two access methods. Section 3 describes our goals and discusses the design of our experiments, while Section 4 presents some of our experimental results. Conclusions can be found in the final section.

2. KEY FEATURES OF THE ACCESS METHODS

The SL is a probabilistic variation of the linked-list in which each node has a different number of forwarding pointers. The number of pointers at each node is generated randomly when the node is inserted and ranges between 1 and a *MaxLevel* number. *MaxLevel* is often defined in conjunction with the number of items in the SL [17]. A NIL node with a *MaxValue* is placed at the very end of the list. Figure 1 shows the key features of the structure.

Searching of the SL is initiated at the header cell and follows the highest possible pointer until no further progress can be made. At this point, the value at the current node is greater than that of the search key. Subsequently, the searching moves down one pointer level at the header node and the above process is repeated. This continues until pointer level one is reached. Searching at this level will either reveal the cell of the structure that maintains the key if it exists, or indicate that the search has failed. The dashed line in Figure 1 indicates the path traversed in order to locate the node with key '301'.

The main idea for both insertion and deletion operations is first to search for the correct location and then splice the structure with the assistance of an auxiliary *Update* vector containing pointers from the traversed path. The *Update* array simply keeps track of the forwarding pointers encountered thus far. Let us assume that we need to insert an item whose key value is '303'. In Figure 1, the forwarding pointers pertinent to the item to be inserted ('303') are circled. In particular, the nth entry of this auxiliary array contains a pointer to the right-most node of level n or higher to the left of the location of the insertion/deletion [18]. To insert a key value, a new node with a random number of pointers is created and inserted into the appropriate position. This random number underlines the probabilistic nature of the SL. Figure 2 shows the insertion of a node with key '303' after four forwarding pointers have been generated with the help of the *Update* vector.

On the other hand, the BB-Tree could be characterized as a binary representation of the 2–3 tree. BB-Tree nodes maintain a record of balancing information along with their data. This balancing information simulates the behaviour of the 2–3 tree nodes and is called the 'level' of the node [20]. The bottom layer of the structure has balance equal to 1. The root of the tree has the maximum level in the structure. Figure 3 depicts a BB-Tree. Each node contains the level information (next to the key). Note that only right-hand edges are allowed to be in the same level (i.e. '325' and '631').

Rearrangement occurs if there are more than two nodes with the same level value. This situation corresponds to the case of an overflowing node in a 2–3 tree. Sibling nodes belonging to the same level can be connected with left- and right-hand edges called 'horizontal edges'. In order to maintain balance in the BB-Tree, two cases have to be dealt with: firstly, all the horizontal left-hand edges have to be eliminated; secondly, the tree has to be rearranged if more than two siblings exist with the same level values (i.e. over-floating tree nodes at the same level). The former action both checks and corrects against 'skewed' internal node arrangement in the BB-Tree and the latter provides for the balanced expansion of the tree upwards through splitting. Therefore, only two operations are required to maintain the BB-Tree, namely *Skew*(n) and *Split*(n), where n is a tree node. The former extends horizontal left-hand edges beneath n. The latter splits the pseudo-node n if it is too large by augmenting the level of every other node.

The searching of the BB-Tree is similar to that of the binary tree. Insertions and deletions are constructed around the *Skew*( ) and *Split*( ) operations since updates are likely to violate the balancing relationship among the tree's nodes. Insertions occur initially at the first (lowest) level of the structure. Subsequently, the tree is traversed from this new node to the root and at each node both the *Skew*( ) and

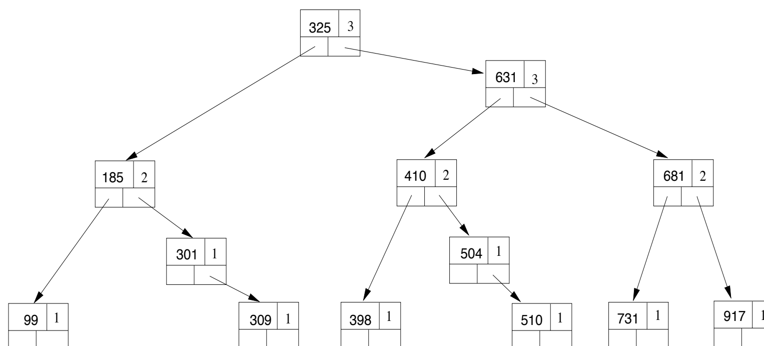

FIGURE 3. The BB-Tree structure.

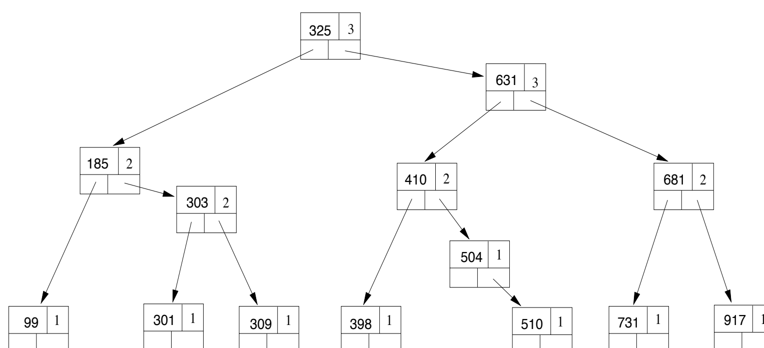

FIGURE 4. The BB-Tree structure after the insertion of item '303'.

Split( ) operations are applied to rectify possible imbalances. Figure 4 shows the resulting structure if a node with the key '303' is inserted. Node deletion from the lowest level is followed by a traversal up to the root of the tree. While ascending in the structure, the level of the current node is checked against the levels of its children. If the level of the current node differs from a child by two then the level of the node is reduced by two and the *Skew*( ) and *Split*( ) operations are performed. To handle deletions of internal nodes, two additional global pointers are used to keep track of the traversal [20].

3. EXPERIMENTAL SETTINGS AND WORKLOADS

In order to secure a fair comparison, we took particular care to design software modules that reflect the algorithms presented in [18, 20]. In addition, we implemented on our own an iterative version of the maintenance routines for the BB-Tree to avoid possible delays due to extensive recursive operations. This version, termed 'Iterative BB-Tree', is implemented around a stack of pointers that is allocated statically by the driver routine of the package. ANSI C was used for the implementation of the access methods and the `gcc` compiler helped us compile our packages on a number of Sun Microsystems platforms. The experimental results reported here were generated with a Sun IPX SparcStation having 64 Mbyte of main memory and running Solaris 2.4-System V Release 4.0. Our experiments were also run on a number of various other configurations, namely a Sun IPC equipped with 24 Mbyte of main memory, running Sun OS-4.1.3; a SparcStation20 with 64 Mbyte, running Sun OS-4.1.4 and finally a Sparc ULTRA1 workstation with 96 Mbyte of RAM, running Solaris 2.5-System V Release 4.0. Results with these three configurations were consistent with those reported here.

The data sets used in the experiments were varied along three dimensions: size, type and workload composition. We created structures with sizes ranging from 1000 to 200 000 entries, as we were particularly interested in developing a thorough understanding of the methods' behaviour under very large data requirements. The data types of the nodes used in our experiments were integer, long (real) numbers, strings with an average length of 20 characters and structures of 50 bytes each (including the key). For brevity, we only present results compiled with nodes with a size of 20 bytes (pointers and level information not included). Structures were created from both ordered (input data arranged in ascending and descending order) and random data sets. Once the data were in place, queries were carried out and response-time statistics were collected. A complete set of queries retrieved all data items in a structure. Finally, batch jobs consisting of both queries and updates were submitted against the structures in order to study the behaviour of the access methods under mixed workloads [12]. We considered as updates either deletion or insertion operations.

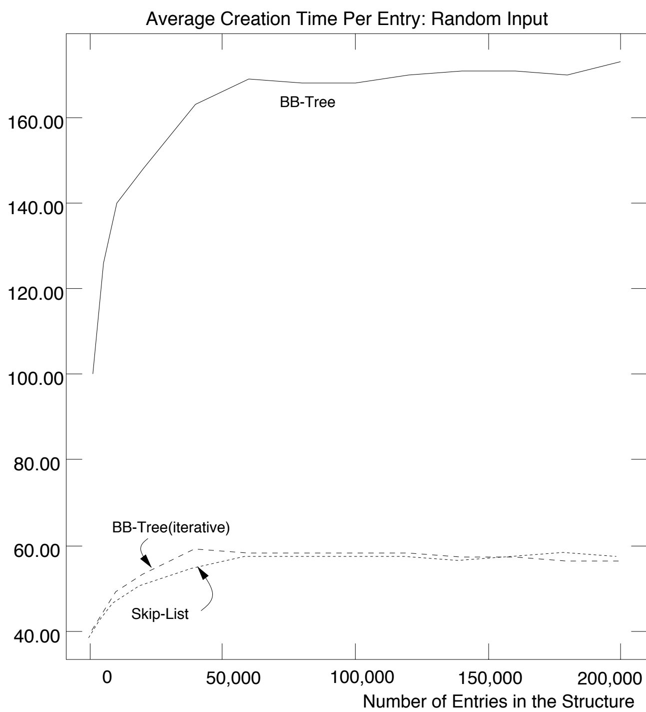

FIGURE 5. Average insertion time per entry—structure created from input in random order.

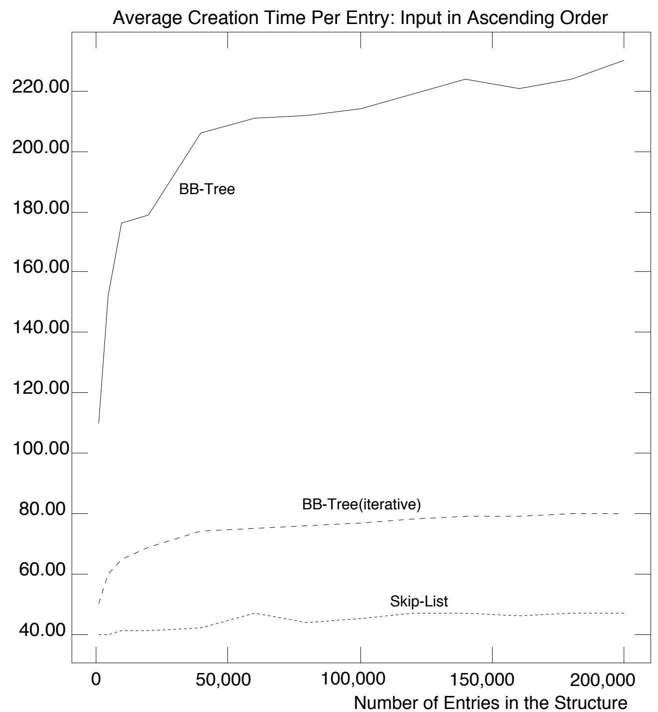

FIGURE 6. Average insertion time per entry—structure created from input in ascending order.

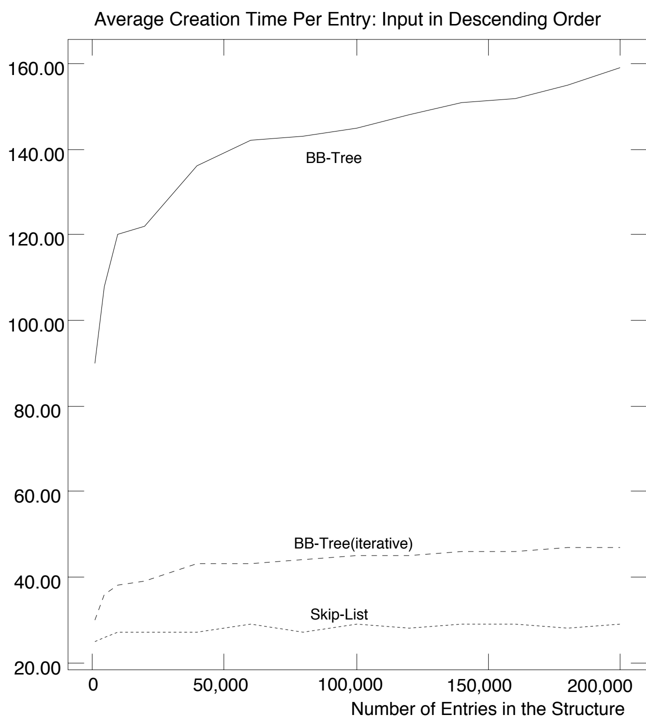

FIGURE 7. Average insertion time per entry—structure created from input in descending order.

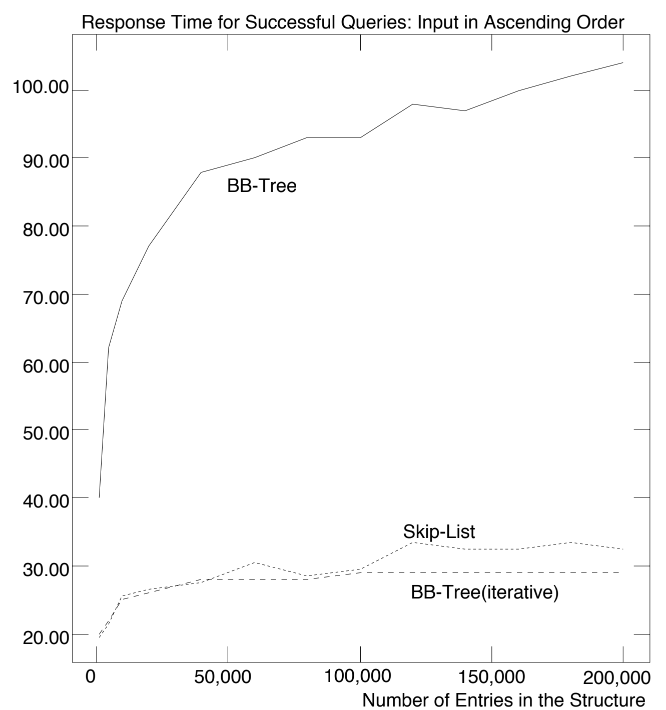

FIGURE 8. Response time for successful searches—structure created from input in ascending order.

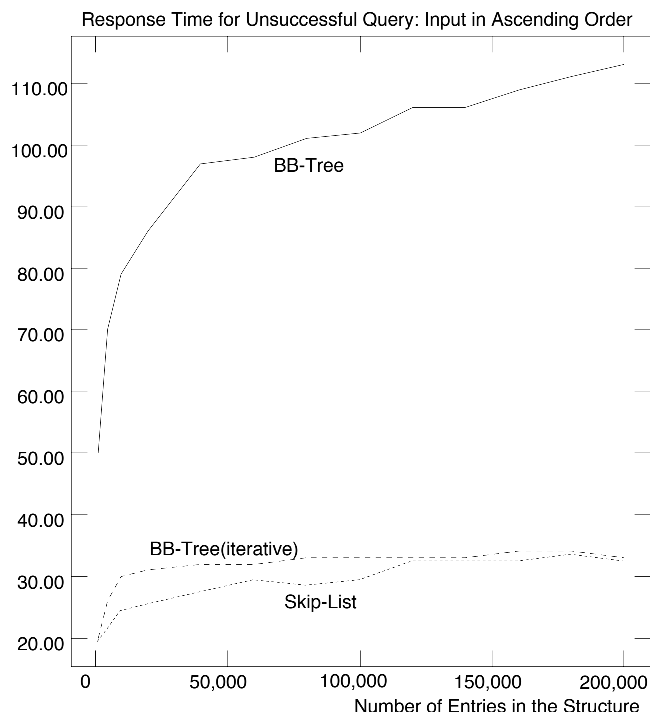

FIGURE 9. Response time for unsuccessful searches—structure created from input in ascending order.

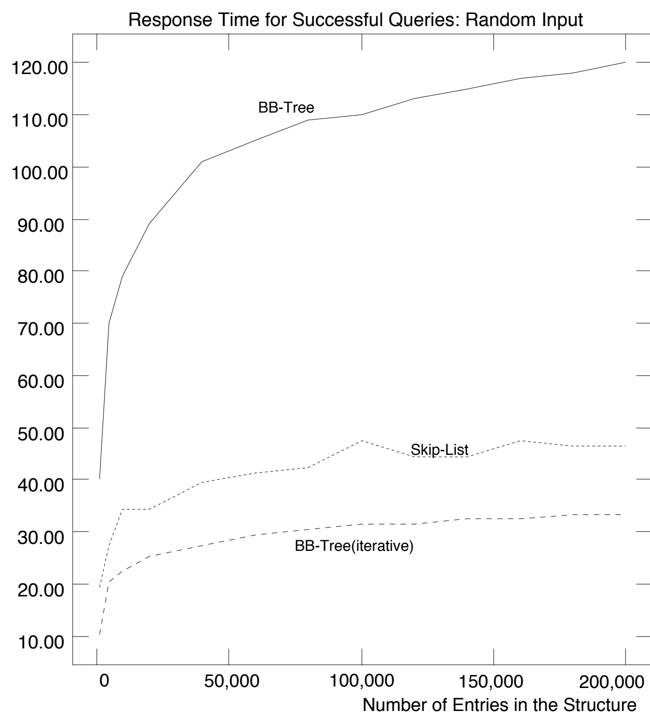

FIGURE 10. Response time for successful searches—structure created from input in random order.

Our experimentation commenced by building the data structures for the various sizes and by comparing the average costs involved. Measurements from experiments with regular BTs were used as the baseline case whenever necessary. In the second phase, both successful and unsuccessful searches for all different types of input data were performed on static structures. Finally, three experiments were carried out using mixed workloads. Each of these workloads consisted of both queries (i.e. searches) and updates (i.e. deletions and insertions) mixed randomly in a pre-determined ratio. The three workloads were: 80% queries and 20% updates (denoted as 80–20%), 50% queries and 50% updates (denoted as 50–50%), and 20% queries and 80% updates (denoted as 20–80%). The objective of these experiments is to examine the performance of the methods in diverse settings and is in line with previous work [12]. The first workload corresponds to environments with conventional database processing requirements [1, 24]. The other two workloads examine the performance of the methods under discussion in highly dynamic environments. In such settings, frequent modifications of structures occur. Dynamic environments of this form are managed by real-time systems [25], main-memory databases [26, 27], rule-based systems [28] and directory maintenance systems for mobile telephony [29].

The *clock*( ) system call [30] was used to measure the average response time (μs) per operation (insertion, query/search, deletion). Our results were compiled by executing a large number of iterations for each individual experiment. We computed the average times observed and the standard deviations. In all experiments involving the two implementations of the BB-Tree (the proposed recursive [20] and our own iterative implementation), the standard deviation was found to be $<1\%$. For almost all the experiments involving SLs, the standard deviation was found to be $<3.5\%$ (a very few experiments produced larger deviations with a maximum observed value of 4.55%). The higher deviation values obtained in the case of SL are attributed to the probabilistic nature of this method. The experiments were conducted with a single user on the system during the night so that we maintain an environment free of interference from other processes. It is worth mentioning that even under usual daytime resource contention conditions, we obtained approximately similar trends to those reported here.

4. EMPIRICAL RESULTS

Figure 5 shows the average time needed (μs) to insert an entry into structures using SL, BB-Tree and Iterative BB-Tree routines as the number of entries ranges from 1000 to 200 000. Entries are identified by unique value keys. Items to be inserted into the structures are created and if necessary randomized off-line (before the measurement of the build-up time commences). The same random data sets are used to create not only the SL and the BB-Tree but also the Binary-Tree baseline structure. From Figure 5, it is apparent that the SL consistently produces a much faster average insertion time than the recursive BB-Tree. However, the iterative implementation of the BB-Tree produces response times directly comparable with those attained by the SL. The average time to insert an item in

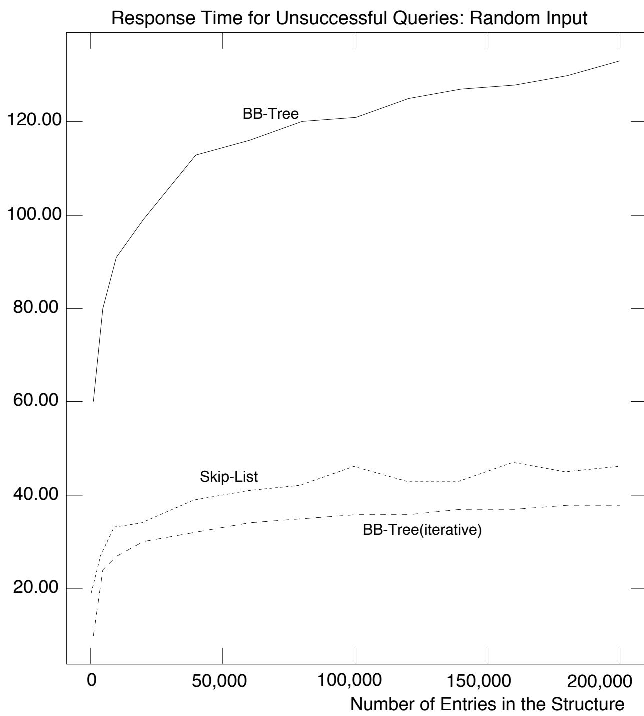

FIGURE 11. Response time for unsuccessful searches—structure created from input in random order.

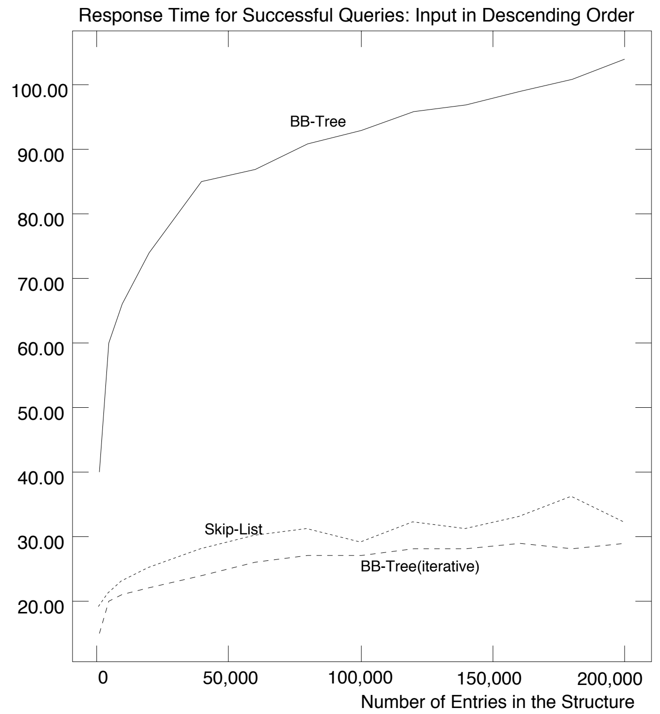

FIGURE 12. Response time for successful searches—structure created from input in descending order.

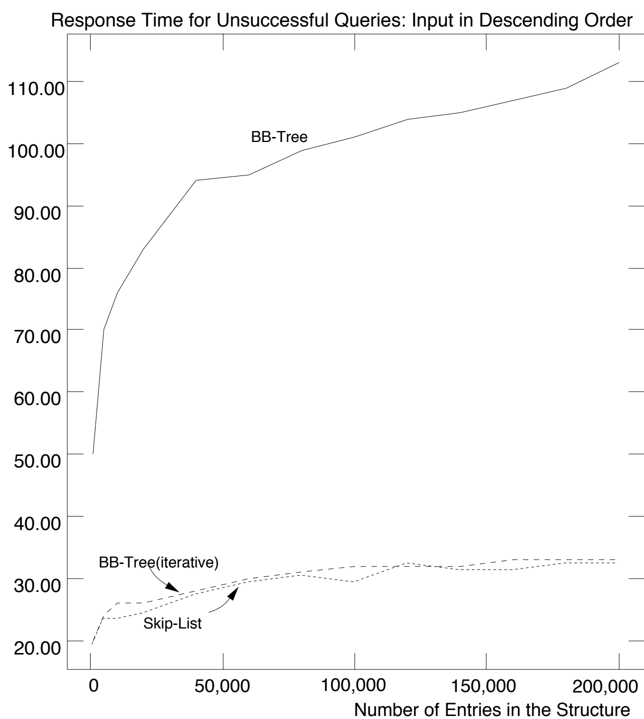

FIGURE 13. Response time for unsuccessful searches—structure created from input in descending order.

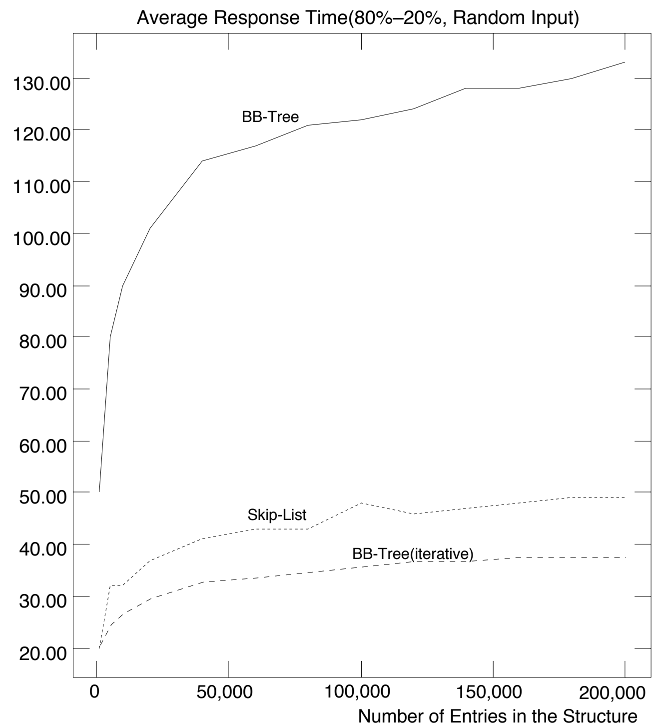

FIGURE 14. Response time for 80–20% mixed workload and structure created from random input.

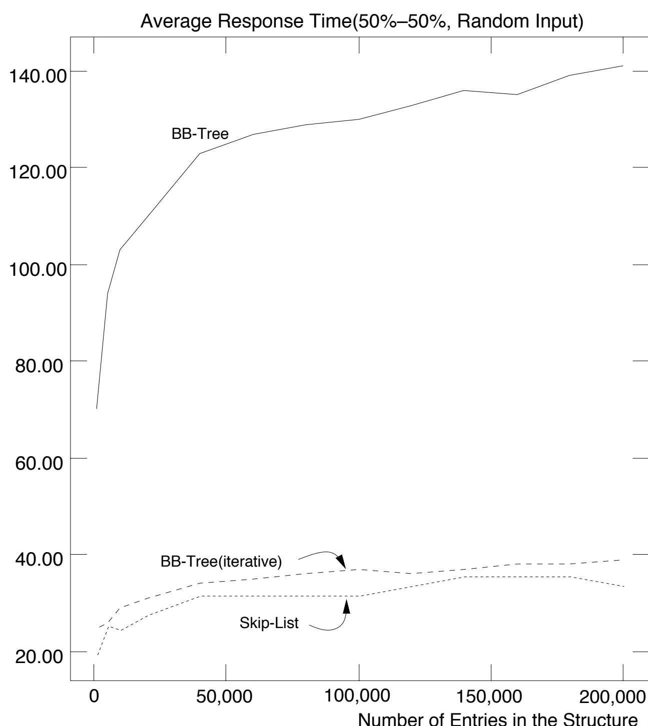

FIGURE 15. Response time for 50–50% mixed workload and structure created from random input.

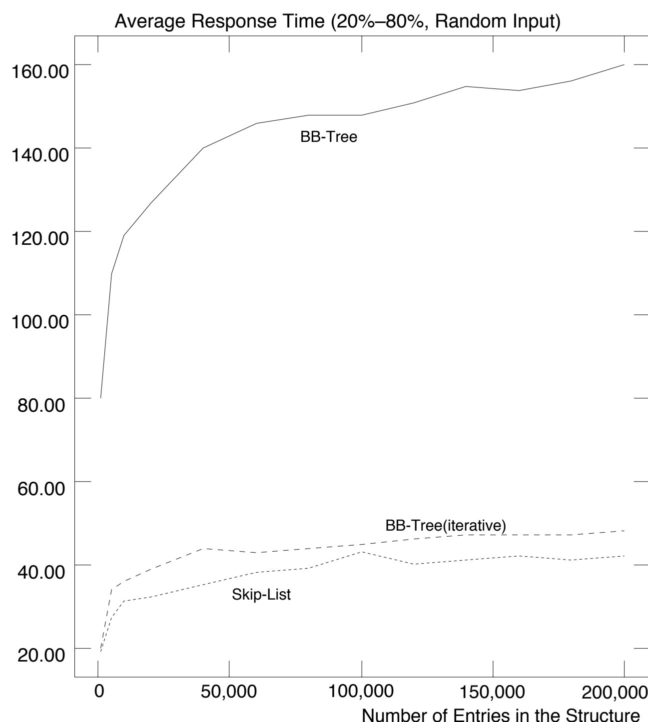

FIGURE 16. Response time for 20–80% mixed workload and structure created from random input.

a structure of 100 000 entries was 168 μs using the BB-Tree routines, 59 μs using the iterative BB-Tree routines and finally 58 μs for SL.

Figures 6 and 7 depict the average time required to insert an entry as the size of the structure ranges up to 200 000 entries in the case of ordered input data sets. Input sets are ordered in ascending (Figure 6) and descending fashion (Figure 7). The SL offers extremely competitive response times for this form of insertion. Here, the SL insertions are more than four times faster than those achieved by the BB-Tree and two times faster than the rates obtained by the Iterative BB-Tree. This is mainly due to the extensive BB-Tree re-balancing required for the ordered input data sets. Table 1 summarizes the responses obtained for up to 20 000 entries and includes measurements compiled with the baseline BT structure. Since BT is a non-self-organizing access method, it suffers as the size of the ordered data set becomes large. Indeed, for more than 30 000 entries we were unable to compile any results as a number of our platforms were thrashing. Although for ordered input (both cases), the BB-Tree is definitely superior to its BT counterpart, it still lags behind the SL.

Once the data structures are set up, they are subjected to read-only requests (queries). All items present in a structure are retrieved and response times are computed. Figures 8–13 show the average response times for both successful and unsuccessful searches in the presence of structures created by both ordered and random data sets. SL offers a 3–4 times faster response time than the recursive version of BB-Tree for both successful and unsuccessful searches. Although the SL in general performs more comparisons, it avoids the inherent overheads of the recursive programming style used to encode the routines of the original BB-Tree proposal. However, in this query-only setting, the iterative version of the BB-Tree out-performs the SL as the structure remains balanced and the above-mentioned two types of overheads are avoided. Only when structures were created from skewed input data was the SL able to perform slightly better than our iterative version of the BB-Tree. The average improvement in query response time for the iterative BB-Tree over the SL throughout the space of the experiment was calculated as 46.36% (successful search) and 25.29% (unsuccessful search) in the case of structures created by randomly ordered input sets.

The time required to search an entry in a standard Binary-Tree is orders of magnitude worse than that of a BB-Tree. For instance, for a structure consisting of 15 000 entries, a successful search takes on average 84 μs in the BB-Tree, 24 μs in the iterative BB-Tree, 32 μs in the SL and 89 303 μs for the Binary-Tree (structures constructed from random input data).

Table 2 summarizes the space overhead in bytes required for the BB-Tree, the SL and the Binary-Tree as the number of entries increases from 1000 to 200 000 nodes (the data field is 20 bytes long per node). The BB-Tree shows 50% more storage overhead than the BT due to the additional integer used for the node balancing information. On the other hand, the SL demands notably less space than both BB and BT as the forwarding pointers of its structure cells are highly utilized. Ordered input sets also generate notably

TABLE 1. Insertion response time (μs) for up to 20 000 entries (input in ascending order)

Number of entries	1000	2000	5000	10 000	20 000
BT (iterative)	5630	11 815	30 184	62 342	93 563
BB-Tree	110	140	152	176	179
BB-Tree (iterative)	50	55	60	65	69
SL	40	45	40	41	41

TABLE 2. Space overhead (byte) for BT, BB-Tree and SL

Number of entries	2000	10 000	20 000	50 000	100 000	150 000	200 000
BT	16 000	80 000	160 000	400 000	800 000	1 200 000	1 600 000
BB-Tree	24 000	120 000	240 000	600 000	1 200 000	1 800 000	2 400 000
BB-Tree (iterative)	25 200	121 200	241 200	601 200	1 201 200	1 801 200	2 401 200
SL (input in random order)	11 208	53 988	105 932	247 564	437 412	588 308	702 052
SL (input in ascending order)	11 048	55 412	110 868	277 192	555 188	835 408	1 112 592
SL (input in descending order)	11 108	55 708	111 192	278 252	556 276	833 468	1 115 276

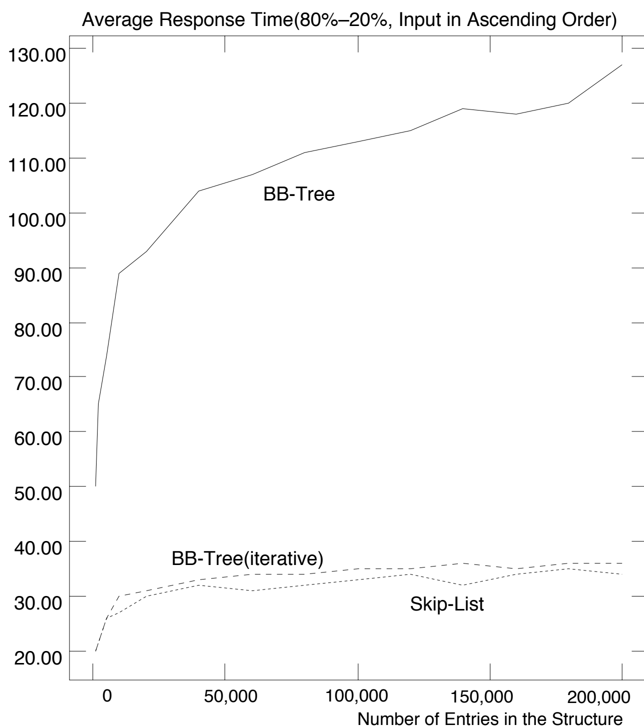

FIGURE 17. Response time for 80–20% mixed workload and structure created from input in ascending order.

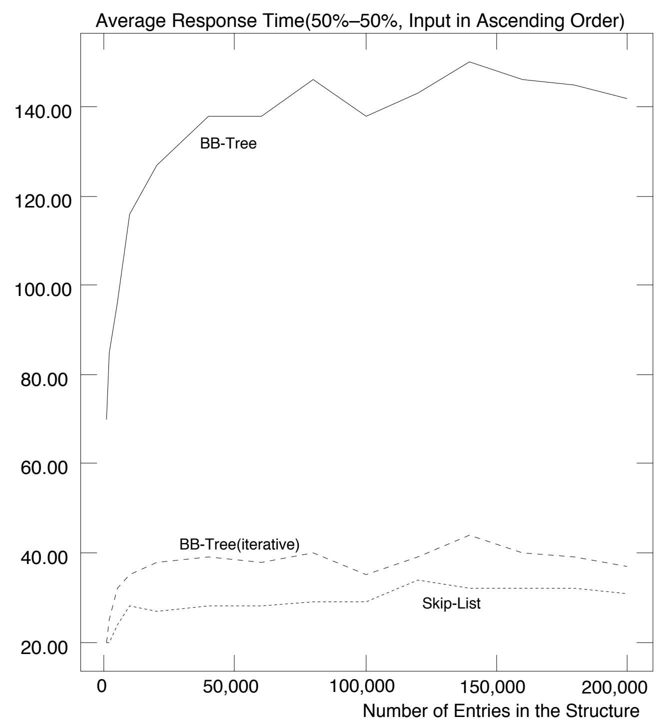

FIGURE 18. Response time for 50–50% mixed workload and structure created from input in ascending order.

increased space overheads for the SL if compared with the space requirements of random input sets.

Figures 14–22 show results from the three experiments with mixed workloads. Deletions and insertions participate equally in the update percentages of the workloads. These mixes of searches and modifications are submitted to the structures created by SL and BB-Trees. At the end of each experiment, the structure under examination was reconstructed so that we avoid possibly undesirable ripple effects in the course of our measurements. In all these graphs, the SL response times are 3–7 times better than those achieved by its recursive BB-Tree counterpart. Although the iterative implementation of BB-Tree performs much better than the original proposal, it is still inferior to SL in the light of skewed input and mixed workloads that involve significant and/or heavy updating. More specifically for

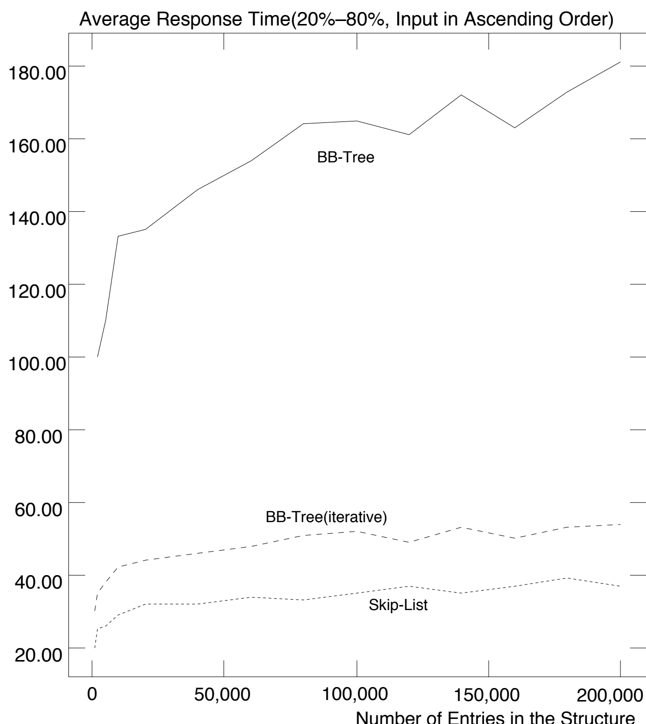

FIGURE 19. Response time for 20–80% mixed workload and structure created from input in ascending order.

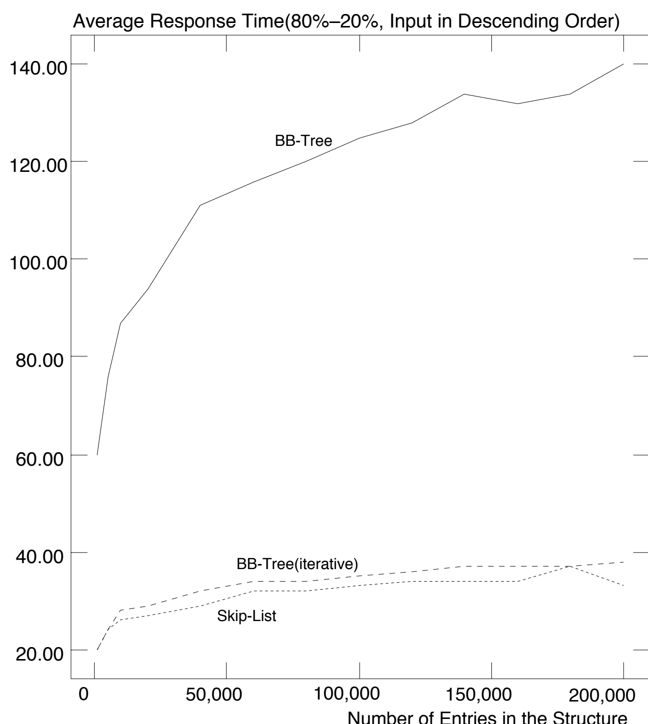

FIGURE 20. Response time for 80–20% mixed workload and structure created from input in descending order.

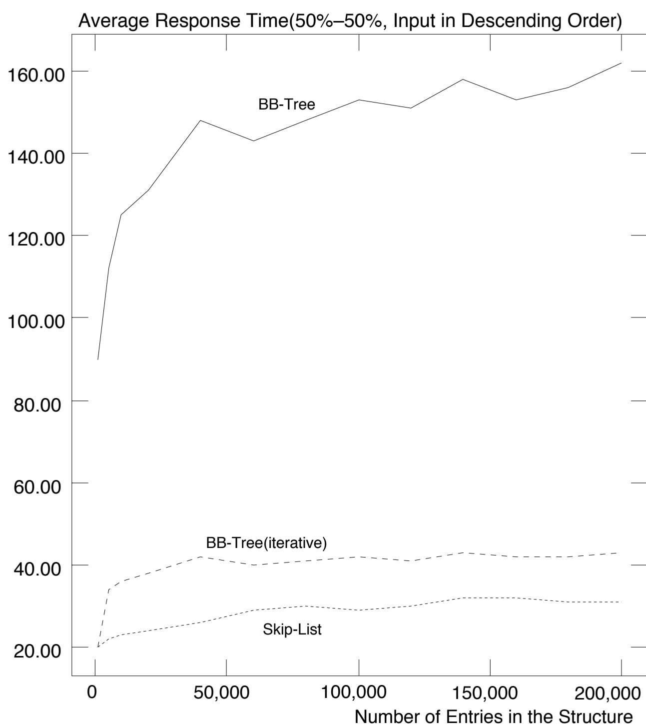

FIGURE 21. Response time for 50–50% mixed workload and structure created from input in descending order.

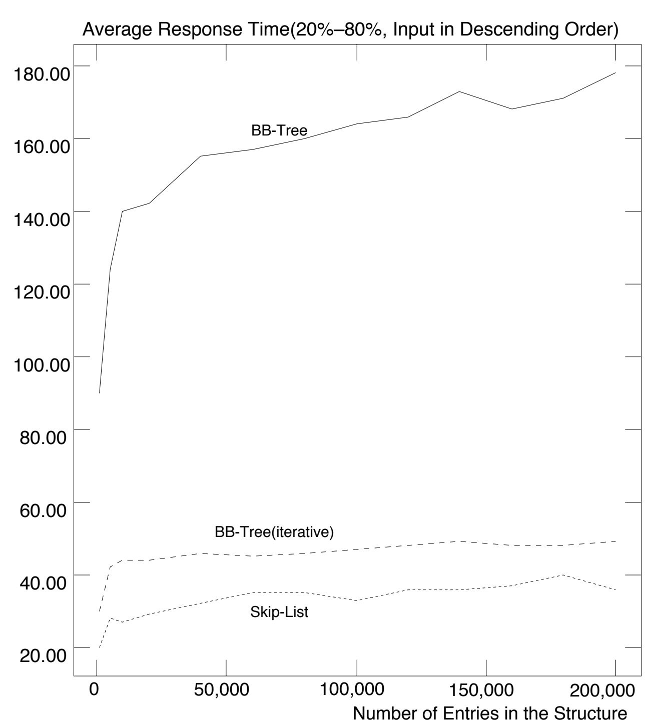

FIGURE 22. Response time for 20–80% mixed workload and structure created from input in descending order.

50 000 entries in Figure 19, SL offers an average of 33 μs operation response, the BB-Tree produces 153 μs, and the iterative BB-Tree implementation results in 49 μs. For a structure of 200 000 entries (Figure 19), the SL achieves 37 μs and the two versions of BB-Tree 181 and 54 μs, respectively.

As we move from mostly read-type (80–20% job composition) to mostly update-type workloads (20–80%), the response times by the BB-Tree become longer, as expected. Nevertheless, the performance of both Iterative and SL appears to level-off for more than 50 000 entries at 35 μs. The required extensive re-balancing of the BB-Tree (especially in the 20–80% workload), as well as the delays due to recursive calls of the initially proposed implementation, contribute to this picture.

Figure 14 depicts the only case where the iterative BB-Tree implementation consistently offers better performance than SL in the presence of mixed workloads. In particular, SL produces on average 27.40% worse response times than the iterative implementation of the BB-Tree. This is due to the fact that a large majority of the operations are queries whose short response times dilute the costs of lengthy and infrequent updates in structures created by random inputs. For instance in Figure 14 and for a 200 000-item structure, the SL gives an average response time of 49 μs while the iterative BB-Tree produces 37 μs.

5. CONCLUSIONS

The ever-increasing sizes of databases call for indexing techniques that furnish very short response times in spite of the high performance of contemporary computer systems. In this paper, we have experimentally evaluated two modern access structures that provide for efficient accessing of voluminous main-memory resident data, namely the probabilistic SL and the BB-Tree. Apart from analytical results [18, 19], there are few and limited experimental results to provide an understanding about the effectiveness of the above two access methods in diverse settings [18, 20]. In addition to the initially proposed recursive maintenance routines for BB-Tree [20], we have created an iterative implementation of this access method. Our experiments focus on large data sets constructed from random and skewed input sequences as well as mixed workloads. To the best of our knowledge, no benchmarking with mixed workloads has been presented before.

Our major results are:

1. In environments where data rarely change, the iterative version of the BB-Tree consistently outperforms the SL for queries on structures created from randomly ordered input data.
2. The SL provides a much better alternative for mixed workloads with a high volume of updates. The use of the SL is also advantageous when the input data used are highly skewed.
3. The iterative implementation of BB-Trees presents very competitive response times in the light of workloads with a limited number of updating operations and structures generated from randomly ordered data. In this type of environment, we have found that the SL offers inferior response times.
4. Under large space requirements, the SL demonstrates the smallest space overhead of the structures considered. This is due to high utilization of the forwarding pointers of the SL cells. Our experience also indicates that it is practically impossible to create a degenerate SL.
5. The recursive maintenance routines of the BB-Tree produced consistently inferior performance than the SL in all our experiments.

ACKNOWLEDGEMENTS

The authors are grateful to the anonymous referees for their comments and suggestions, to Mikhail Reznikov for providing counter implementations, and to Stephen Milliner for commenting on earlier versions of the manuscript. This work was partially supported by the Center for Advanced Technology in Telecommunications (CATT) in Brooklyn, NY.

REFERENCES

[1] Litwin, W., Roussopoulos, N., Levy, G. and Hong, W. (1991) Trie hashing with controlled load. *IEEE Trans. Software Engineering*, **SE-17**, 678–691.

[2] Walker, A. and Wood, D. (1976) Locally balanced binary trees. *Comp. J.*, **19**, 322–325.

[3] Manolopoulos, Y. and Kollias, J. G. (1989) Expressions for completely and partly unsuccessful batched search of sequential and tree-structured files. *IEEE Trans. Software Engineering*, **SE-15**, 794–799.

[4] Özden, B., Biliris, A., Rastogi, R. and Silberschatz, A. (1994) A low-cost storage server for movie on demand databases. In *Proc. 20th Very Large Databases Conf.*, Santiago, Chile.

[5] Frost, R. A. and Peterson, M. M. (1982) A short note on the binary search tree. *Comp. J.*, **25**, 158.

[6] Bayer, R. (1972) Symmetric binary B-trees: data structure and maintenance algorithms. *Acta Informatica*, **1**, 290–306.

[7] Rahm, E. (1993) Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. Database Syst.*, **18**, 333–337.

[8] Palmer, M. and Zdonik, S. (1991) Fido: a cache that learns to fetch. In *Proc. 17th Int. Conf. on Very Large Databases*, Barcelona.

[9] Knuth, D. (1973) *Sorting and Searching: the Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading, MA.

[10] Gabarro, J., Martinez, C. and Messeguer, X. (1996) A design of parallel dictionary using skip lists. *Theor. Comp. Sci.*, **158**, 1–33.

[11] Adelson-Velskii, G. M. and Landis, E. M. (1962) An algorithm for the organization of information. *Dokl. Akademia SSSR*, **146**, 1259–1262.

[12] Bell, J. and Gupta, G. (1993) An evaluation of self-adjusting binary search tree techniques. *Software Practice Experience*, **23**, 369–382.

[13] Gonnet, G. and Baeza-Yates, R. (1991) *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, MA.

[14] Singhal, M. (1990) Update transport: a new technique for update synchronization in replicated database systems. *IEEE Trans. Software Engineering*, **SE-16**, 1325–1336.

[15] Ciciani, B., Das, D., Iyer, B. and Yu, P. (1990) A hybrid distributed centralized system structure for transaction processing. *IEEE Trans. Software Engineering*, **SE-16**, 791–806.

[16] Koster, C. H. A. and Van Der Weide, T. P. (1995) Hairy search trees. *Comp. J.*, **38**, 691–694.

[17] Pugh, W. (1989) *A Skip List Cookbook*. Technical Report UMIACS-TR-89-72.1. The University of Maryland, College Park, MD.

[18] Pugh, W. (1990) Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, **33**, 668–676.

[19] Kirschenhofer, P., Martinez, C. and Prodinger, H. (1995) Analysis of an optimized search algorithm for skip lists. *Theor. Comp. Sci.*, **144**, 199–220.

[20] Andersson, A. (1993) Balanced search trees made simple. In *3rd Workshop on Algorithms and Data Structures (WADS)*, Montreal, Canada.

[21] Kirschenhofer, P. and Prodinger, H. (1994) The path length of random skip lists. *Acta Informatica*, **31**, 775–792.

[22] Devroye, L. (1992) A limit theory for random skip lists. *Ann. Appl. Probab.*, **2**, 597–609.

[23] Munro, J., Papadakis, T. and Sedgewick, R. (1992) Deterministic skip lists. In *3rd Ann. ACM Symp. on Discrete Algorithms*, Orlando, FL, pp. 367–375.

[24] Gray, J. (ed.) (1991) *The Benchmark Handbook: for Database and Transaction Processing Systems*. Morgan Kaufmann, San Mateo, CA.

[25] Kao, B. and Garcia-Molina, H. (1996) Scheduling soft real-time jobs over dual non-real-time servers. *IEEE Trans. Parallel Distrib. Syst.*, **PDS-7**, 56–68.

[26] Salem, K. and Garcia-Molina, H. (1990) System M: a transaction processing testbed for memory resident data. *IEEE Trans. Knowledge Data Engineering*, **KDE-2**, 161–172.

[27] Eich, M. (1987) A classification and comparison of main-memory database recovery techniques. In *Proc. IEEE Int. Conf. on Data Engineering*, pp. 332–339.

[28] Ceri, S. and Windom, J. (1992) Production rules in parallel and distributed database environments. In *Proc. 19th Int. Conf. on Very Large Databases*, Vancouver, BC, Canada.

[29] Qiu, X. and Li, V. (1995) Performance analysis of PCS mobility management database system. In *Proc. 4th Int. Conf. on Computer Communications and Networks (ICCCN'95)*, Las Vegas, NV.

[30] Sun Microsystems (1992) *Sun OS 4.1.3 Answer Book*. SunSoft.