International Journal of Computer Applications (0975 – 8887)
Volume 33– No.10, November 2011

A Forest of Hashed Binary Search Trees with Reduced Internal Path Length and better Compatibility with the Concurrent Environment

Vinod Prasad
Sur College of Applied Sciences
Ministry of Higher Education, Sultanate of Oman

ABSTRACT

We propose to maintain a Binary Search Tree in the form of a forest in such a way that – (a) it provides faster node access and, (b) it becomes more compatible with the concurrent environment. Using a small array, the stated goals were achieved without applying any restructuring algorithm. Empirically, we have shown that the proposed method brings down the total internal path-length of a Binary Search Tree quite considerably. The experiments were conducted by creating two different data structures using the same input - a conventional binary search tree, and a forest of hashed trees. Our empirical results suggest that the forest so produced has lesser internal path length and height in comparison to the conventional tree. A binary search tree is not a well-suited data structure for concurrent processing. The evidence also shows that maintaining a large tree in form of multiple smaller trees (forest) increases the degree of parallelism.

Keywords: Binary Search Tree Path Length, Parallel Processing Binary Search Tree, Balanced Tree

1. INTRODUCTION

The Binary Search Tree *(BST)* is a widely used storage medium in the primary memory. A reasonably balanced *BST* provides fast data access and retrieval. Despite its wide popularity, it has few serious problems. First, its shape depends on the nature of the input. Second, it is a sensitive data structure for insertion and deletions -the tree shape could be destroyed, following a series of insertions and deletions. Third, it is a highly incompatible data structure for concurrent processing. Over the years these problems have been investigated extensively. For the tree shape, refer to [1]-[4], [13]-[18]; and, for the parallel processing of the tree, refer to [5]-[11]. Eppinger [12] investigated tree's behavior for a large number of insertions and deletions.

A tree could be maintained in better shape either dynamically or statically. In dynamic maintenance, following each operation tree is inspected, and readjusted, if the operation causes any structural damage. AVL Tree [1], Martin & Ness [2], Red-Black Tree [3], and Splay trees [15] fall into this category. In Splay Trees, the frequently accessed node is pushed towards the root of the tree for future immediate access. Gonnet [12], and Gerasch [17] proposed algorithms to maintain the tree with reduced internal path length. In static maintenance, periodically, the entire tree is taken as input, and some maintenance work is applied. Examples of static maintenance are: Day's algorithm [4], Chang & Iyengar's algorithm [14], and, Stout & Warren's algorithm [16]. Both dynamic and static solutions have their advantages and disadvantages. The dynamic maintenance of the tree is slow because of frequent inspections and rotations. Apart from that, every node in the tree has to store some additional information. Static algorithms, on the other hand, demand lots of extra space. In some cases, the algorithm consumes extra workspace that is twice the size of the input. For example, Chang & Iyengar's algorithm [16] requires an additional array, equal to the size of input, as extra workspace.

As far as the parallel processing of the tree is concerned, substantial work has been done to develop concurrent algorithms, refer to [5]-[11]. In a binary search tree, the root is the only gateway for all the active processes making it difficult to achieve maximum parallelism. Imagine a situation where we have to update the root of the tree. A process performing this operation has to lock the root, making the entire tree unavailable for rest of the processes. No matter how efficient our concurrent algorithms are, other processes have to wait until the previous releases the lock. Ellis [9], [10] presented solutions for concurrent searches and insertions in the 2-3 and AVL trees. Kung and Lehman [11] investigated ordinary binary search trees, and proposed solutions so that the system could support any number of processes performing searches, insertions, deletions, and rotations. To ensure that the searches are never blocked they used special nodes and pointers.

Most of the presented solutions use some kind of locking scheme to allow multiple processes to act upon a single binary search tree simultaneously. The common goal of all proposed solutions was to increase the degree of concurrency by having a lesser portion of the tree locked, and thus exposing a major portion to the rest of the processes. Such algorithms can increase the degree of concurrency up to a certain extent. However, better results could be obtained, if the underlying data structure is modified to accommodate large number of processes. Substantial work has been done on algorithms, but hardly any attempt has been made to create a flexible data structure.

In this paper, we propose a forest of binary search trees to deal with the stated problems - tree balance, and its incompatibility with the concurrent environment. To examine the overall balance of the proposed forest, we have used internal path length *(IPL),* and the height of the tree as measurement parameters. (Concurrency-related issues are discussed in the section 6.)

The height of the tree is the length of the longest path from the root to the leaf. The height is an important parameter to study the worst-case behavior of the tree. For average case analysis, the internal path length of the tree (*IPL)* can be used. *IPL* is the average distance of every node in the tree from the root, and is defined as the sum of the depths of all the nodes in the tree. Thus, for a tree with just one node, the *IPL* is equal to 0, and for a tree with two nodes, the *IPL* is equal to 1. Let I_n be the *IPL* of a tree with '*n'* nodes, and C_n be the average number of comparisons required for a successful search, then we have a relation: $I_n = n\ (C_n - 1)$. From the given relation, it is clear that the *IPL* of the tree directly affects its performance. The lesser the height, the lesser will be the *IPL*, and hence, the faster would be the search. Knuth [19] has given a formula that relates the height of a '*n*' node random binary search tree H_n, and C_n as $C_n = 2(1 + 1/n)\ H_n - 3$.

2. CREATION OF THE FOREST

It is possible to maintain a random *BST* in form of a set of trees called 'forest'. The number of trees in the forest would depend on the application. For the purpose of simulation, we have used a forest of 11 trees. To hold multiple trees, we need multiple roots. We have used an array of pointers of size $k=11$. Each cell of the array acts as a tree root. Like a usual tree, when the tree is empty, each root (*array cell*) points to a null value (refer to Figure 1). When a key has to be inserted or deleted, it is first divided by '*k',* and the remainder is calculated to find the array location. Assuming the array location to be the root of the tree, the desired operation is performed as usual. In other words, keys need to be hashed by the function: *loc=key % k*. Where *'loc'* is the array location to/from which the key has to be inserted/deleted. Using this technique, we get a set of *k* possible trees, which resembles a forest, but acts as a single binary search tree. The only difference between a usual single *BST* and a forest is determining the tree location using hashing. In Figure 1, a forest of two trees is shown. The collective IPL of the forest = the sum of the *IPL*s of all the trees in the forest = 12+6 = 18. Maximum height of the tree in the forest is 3.

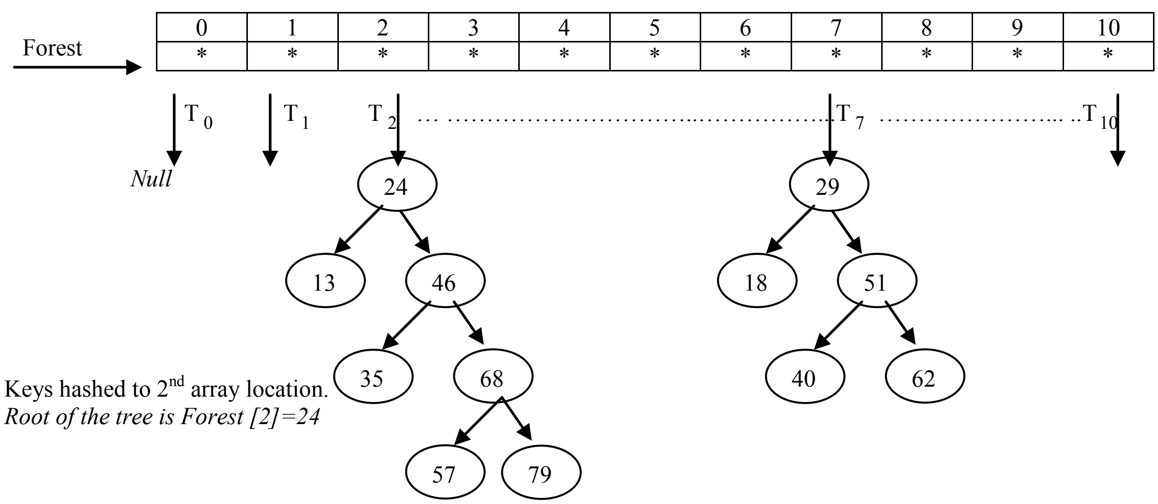

Figure 1: A Forest of Hashed Trees

3. METHODOLOGY

Using the same random input, two different structures were created - a conventional tree, and a forest. For every insertion, a random number was generated that was supplied to two separate algorithms: one that creates a conventional tree; and, the other that creates a forest. Duplicate keys were ignored. Once all the keys were inserted, the *IPL* of both of the structures was calculated. The *IPL* of the conventional tree was calculated as usual by calculating the sum of the depths of all the nodes in the tree. While, for the forest, the *IPL* was calculated by adding all the IPLs of the individual trees in the forest, at the same time, the heights of all the trees in the forest were recorded. For example, in Fig.1, the total *IPL*s of the forest was *12+6 = 18*, the average height of the forest = *(3+2)/2 = 2.5*, the height of the worst tree in the forest = 3. For the division remainder method of hashing, it has been shown that a prime number distributes the keys more uniformly - that is why the array size of 11 was our choice. Another reason was to maintain the array size to be roughly equal to 1% of the input size. We chose the tree size to be *1023 and 2047* so that the forest so produced could also be compared with a perfect, balanced tree, and not just with the random *BST*. (For quick reference, please note that perfect balanced trees with *1023* and *2047* nodes would have heights *9* and *10*, respectively; their *IPL*s would be *8194 and 18434*, respectively.)

4. RESULTS

The following are parameters related to the forest that were used to compare the forest to a conventional and perfect balanced tree:

Average height of the forest = Sum of the heights of all 11 trees in the forest/11.

Height of the worst tree in the forest = A tree with the maximum height in the forest.

Collective path-length of the forest = The sum of the path-lengths of all 11 trees in the forest

Table 1 is obtained as a result of several tests conducted with Borland C++ compiler 5.5 under windows, and GNU C++ compiler 3.4.3 (g++) under Linux. Table 1 (a) shows results for $n = 1023$, and $k = 11$. The average height of the forest was quite close to 9 - somewhat like a perfect, balanced tree with the same number of nodes. If we compare the average height of the forest to the height of the conventional tree, there is a huge reduction (39% to a whopping 52%). The height of the worst tree in the forest was *17*, where the conventional tree went up to *24*. However, the worst tree in the forest would not have same number of nodes as the conventional tree; therefore, comparing their heights is not justifiable, but this will definitely give the worst-case behavior of both of the structures. As far as the *IPL* is concerned, a reduction in the forest *IPL* is more than considerable. Among the *10* sample runs, the worst *IPL* of the forest was recorded as 6623 - that is, far better than 11299 (the *IPL* of the corresponding *BST*), and much better than 8194 (the *IPL* of the perfect, balanced tree with the same number of nodes). The reduction in *IPL* was recorded to be from 39% to 52%. Though the *IPL* is enough for an average case analysis, it would be interesting to apply the formula that relates *IPL* (I_n), and the average number of comparisons required for a successful search C_n. Putting $I_n = 6623$, and $n=1023$, we get for the forest: $C_n = (I_n / n) + 1 = (6623/1023) + 1 = 7.47$. For the corresponding conventional tree: $C_n = (11299/1023) + 1 = 12$. For a perfect, balanced tree with the same number of nodes: $C_n = (8194/1023) + 1 = 8$. Hence, in terms the average number of comparisons required for a successful search, it is quite clear that the forest needs a minimal number of comparisons. These results indicate that the behavior of the forest is far better than that of a random *BST*. If we could ignore the extra cost of time consumed in hashing, on the average, the forest clearly seems to outperform even a perfect, balanced tree.

Obviously, as the size of input increases, we will need a larger array to get similar results. What is the optimal ratio of '*n*' and '*k*'? This could be an interesting question. In our case, presented results in table 1(a) are for 1:93 (input size is 93 times of array size) and, 1(b) is for 1:186. Let's consider a simpler question; how large should an array be, if we wanted to reduce the average height of the forest by 50%? Using random input: we know that the tree height is $O(\lg(n))$. We have k trees, assuming that our hash function distributes keys uniformly among k trees, we would have each tree with a height of approximately $O(\lg(n/k))$, i.e., we have the relationship :- $\lg(n) / \lg(n/k) = 2$

$$\Rightarrow \lg(n) = 2\lg(n/k) \Rightarrow n = (n/k)^2 \Rightarrow k = \sqrt{n}$$

Theoretically, that means for $n = 1023$, to reduce the average height of the forest to half, we need an array of size $\sqrt{1023} = 32$. However, we have seen that the empirical results do not confirm this, and are more encouraging. With an array of a size of $k=11$, and $n=1023$, several runs have shown that, in most of the cases, the reduction in average height of the forest was between 39% to 52%. The total *IPL* of the forest fell to *40%* in comparison to that of the conventional tree. In most of the cases, the total *IPL* of the forest beat even a corresponding perfect, balanced tree.

In Table 1(b), we have provided the results of when input size was doubled, i.e., now $n = 2047$, but array-size is kept unchanged. This would give us some idea of how the performance of a forest is affected, when we increase the size of the input. Comparing the average height of the forest with the height of the conventional tree, it is evident that the reduction in height was still *26% to 48%*. The worst *IPL* of the forest was *16216* - that is, far better than *24133* (the *IPL* of the corresponding *BST*), and still better than *18434* (the *IPL* of the perfect, balanced tree with *2047* nodes). Indeed, there was some deterioration in the performance, but this confirms that the forest still works better when input size is increased to 186 times of the array size.

5. TIME AND SPACE REQUIREMENTS OF THE FOREST

We will compare the time and space requirements of the proposed forest with those of the usual *BST*. On the average, a random BST with n nodes requires $\lg(n)$ time to perform most of the operations. However, in the case of a forest, hashing is required to jump to the correct tree, i.e. additional $O(1)$ is required for hashing. Following this, the entire process is the same as that of a usual *BST* operation. Hence, it is straightforward that, on the average, the forest with n nodes and k trees takes $O(1) + \lg(n/k)$ time. The worst-case behavior of a *BST* is $O(n)$ - which remains the same as for the forest. This happens when all the keys are in sorted order, and hashed to one single tree in the forest, letting the tree grow up to n. A natural question is: what is the additional space requirement of the forest? When compared to a normal *BST*, the forest does not require additional space. Whatever space is used in form of the array is actually used to store the roots of the trees in the forest; hence, the array is not an overhead. From the given results, it is evident that the proposed forest is capable of providing an AVL tree-like performance at the cost of $O(1)$ time. An *AVL* tree does have additional overheads, such as: time consumed in examining the tree balance and restructuring. Apart from that, each node in the *AVL* tree requires two additional bits to store balance information. This is all that is required, in terms of time and space. As a result, the forest behaves as if it was a balanced tree without using any rebalancing algorithm.

The biggest advantage of this technique is that the entire forest acts as if it was a single *BST*. That means we do not need to change any existing *BST*-related algorithm in order to apply with the forest; insertion, deletion and updating algorithms remain the same. The tree traversal requires the roots of the individual tree to be passed to the traversal algorithm. In fact, all the trees in the forest can be traversed in parallel.

6. PARALLEL PROCESSING OF THE FOREST

Another benefit is compatibility with the concurrent environment. The forest of trees is far more compatible with the concurrent environment than a single *BST*. Here, we have a forest with multiple trees, and each tree in the forest is independent, and hence, can be operated independently. No proof is required to demonstrate that a forest allows more numbers of processes to act upon the different trees simultaneously. Without using any locking schemes, *'k'* processes can act simultaneously on *'k'* different trees. From the structure of the forest, it is reasonable to conclude that maintaining the tree this way increases the degree of parallelism by '*k'* times. The choice of *'k'* depends on the required degree of parallelism. For huge data sets and massively parallel systems, we would like to have a greater number of trees in the forest, requiring larger arrays. Although, to reduce the *IPL*, we haven't used any tree-restructuring technique here, still the individual tree in the forest can be restructured for even better results. As stated earlier, in global restructuring, the entire tree is taken as input and restructured. This process may take a lot of time particularly if the tree size is large. But in case of a forest, only a part of the forest (single tree) would be needed for restructuring at a time. In fact, tree restructuring can be done simultaneously with the other operations. Furthermore, a binary search tree is a sensitive data structure, with regard to insertions and deletions. A series of insertions and deletions can harm the balance structure of the tree. Eppinger [10] has shown that performing a large number of insertions and asymmetric deletions increases the *IPL* of a tree to up to $\theta(n\ lg^3(n))$, resulting in the tree being no longer random. In the case of a forest, insertions and deletions will get distributed over the different trees, increasing the immunity of the structure to such operations.

7. CONCLUSION

We have shown that, without losing structural information, a tree could be converted into a forest with reduced internal path length and better compatibility with the concurrent environment. This has been achieved without using any restructuring algorithm. At the cost of hashing, without compromising for space, small modifications in the data structure results in faster node access, and an increased degree of parallelism.

Table : A comparison between the Conventional Tree and the Forest of Trees

Sample run	Conventional Tree		Forest			% Reduction in Height and Path-Length	
	Height	Path-length	Average Height of the Forest	Height of the Worst Tree	Collective Path-length of the Forest	% Reduction in Height	% Reduction in the Path-length
1	18	11018	10.8	12	6193	40	44
2	20	10782	12.0	14	6594	40	39
3	21	11299	11.6	13	6623	44	41
4	24	13728	12.0	17	6542	50	52
5	21	11840	11.1	14	6081	47	48
6	22	11158	12.2	15	6567	44	41
7	20	11185	11.4	16	6467	43	42
8	20	10814	12.2	17	6533	39	40
9	22	11359	12.0	16	6481	45	43
10	24	12174	11.4	13	6414	52	47

a): Number of Nodes: 1023

Sample run	Conventional Tree		Forest			% Reduction in Height and Path-Length	
	Height	Path-length	Average Height of the Forest	Height of the Worst Tree	Collective Path-length of the Forest	% Reduction in Height	% Reduction in the Path-length
1	21	24918	15.0	17	15865	28	36
2	21	24133	15.5	17	16216	26	32
3	25	24179	14.7	19	15603	41	35
4	26	26814	14.9	17	16110	42	40
5	22	24728	14.1	18	15936	36	35
6	27	28267	14.0	17	15207	48	46
7	24	26528	14.5	17	15489	39	42
8	24	25742	13.9	18	15705	42	39
9	25	24478	14.7	18	15508	41	37
10	23	24219	14.5	18	15418	37	36

(b): Number of Nodes: 2047

8. REFERENCES

[1] Adel'son-Vel'skii, G. M, and Landis E. M, 1962. "An Algorithm for the Organization of Information, *Soviet Mathematics Doklady"*, Vol. 3, 1259–1263.

[2] Martin, W. A, and Ness, D. N. 1972, "Optimal Binary Trees Grown with a Sorting Algorithm", *Communication. of the ACM*, 15, 88-93.

[3] Bayer, R. 1972, "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms", Acta Informatica. 1, 290–306.

[4] Day, A. C. 1976, "Balancing a Binary Tree, *Computer Journal"*, 19, 360-361.

[5] Samadi, B. 1976, "B Trees in System with Multiple Users", *Information Processing Letters*, 5 (4), 107-112.

[6] Eswaran, K. P., Gray, J. N., Lorie, R. A, and Traiger, I. L. 1976, "The notions of Consistency and Predicate Locks in a Database System", *Communication of the ACM*, 19(11), 624-633.

[7] Bayer, R., Schkolnick, M. 1977, "Concurrency of Operations on B Trees", *Acta Informatica,* 9(1), 1-21, 1977

[8] Ries, D. R., Stonebreaker, M. 1977, "Effects of Locking Granuality in a Database Management System". *ACM Trans. Database Systems*, 2(3) , 233-246, 1977

[9] Ellis, C. S. 1980. "Concurrent search and insertion in AVL trees". *IEEE Transactions on Computers*, Vol 29, 811–817.

[10] Ellis, C. S. 1980, "Concurrent search and insertion in 2-3 trees". *Acta Informatica*, Vol 14, 63–86, 1980

[11] Kung, H. T., Lehman, P. L. 1980, "Concurrent manipulation of binary search trees". *ACM Transactions on Database Systems*, Vol. 5, 354–382

[12] Eppinger, J. L. 1983, "An Empirical Study of Insertion and Deletion in Binary Search Trees", *Communication of the ACM*, 26, 663-669.

[13] Gonnet, G. H. 1983, "Balancing binary Search Trees by Internal Path Reduction", *Communication of the ACM*, 26(12), 1074-1081.

[14] Chang, H, Iyengar, S. S, 1984, "Efficient Algorithms To Globally Balance a Binary Search Tree", *Communication of the ACM*, 27, 695-702.

[15] Sleator, D. D., Tarjon, R. E, 1985, "Self-Adjusting Binary Search Trees". *Journal of The ACM*, 32(3), 652-686.

[16] Stout, F, Bette, L. W. 1986, "Tree Rebalancing in Optimal Time and Space", *Communication of the ACM,* 29, 902-908.

[17] Gerasch, T. E. 1988, "An insertion algorithm for a minimal internal path length binary search tree". *Communications of the ACM*, Vol.31 (5), 579–585.

[18] Bell, J., Gupta, G. 1993, "An evaluation of self-adjusting binary search tree techniques". *Software Practice and Experience*, Vol. 23(4), 369–382.

[19] Knuth, D. E. 2005, The Art of Computer Programming," *Pearson Education*, Vol. 3, Searching and Sorting.