Towards a Dynamic Optimal Alphabetic Tree

A.A. Belal M.S. Selim S.M.Arafat
Department of Computer Science and Control
University of Alexandria,EGYPT
belala@usa.net

Abstract

Binary trees have a wide variety of applications in computer science and information systems. Fast algorithms for building all kinds of binary trees in *O(n log n)* time do exist. However, no existing algorithm makes it possible to insert in (or delete from) the tree without losing its optimality. In this paper, we propose an algorithm to insert into or delete from a weighted binary alphabetic tree in **linear time** *keeping the tree optimal after insertion or deletion*. We show that both insertion and deletion of a node can be done in *O(n)* time provided its weight is not bigger than the higher weight of its two neighbouring nodes. This algorithm makes it possible to have a dynamic optimal alphabetic tree with reasonable complexity.

Key words : Optimal alphabetic trees,insertion and deletion ,linear time alphabetic trees.

1 Introduction

Binary trees have received a considerable attention in computer science research. It saves a lot of time to search for data stored in a binary tree structure; it also minimizes code lengths to use special binary trees. Some tree building algorithms assume an equally weighted node tree, in which case an optimal tree means a balanced one. However, when the nodes of the tree, both internal and external, have different access frequencies, it becomes important to assign a different weight for each node. In this case, an optimal tree is the one with minimal cost, where the cost is the summation of the products of the node weight by the node level over all nodes. Optimal trees of this kind can be built in $O(n^2)$ time complexity [9]. For the simpler case where only external nodes have weights, the optimal tree can be found in time *O(n log n).* An example is the Huffman tree[7] which is widely used in coding and information theory. A more constrained kind is the alphabetic tree, also called an insertion tree, where nodes must appear in their original order in the final tree[4,5].

In the last few years more results on optimal alphabetic trees (OAT) were reported in the literature. The equivalence of the OAT to optimal binary search trees was reported in [2]. The use of the 1-dimensional O(n log n) algorithm for 2-dimensional information retrieval was considered in [1,3]. The search for sub O(n log n) algorithms for the OAT problem was also recently reported [8,10]. Linear time algorithms were also presented in [6] for some special weight sequences.

In this paper, we give an O(n) time algorithm to insert an element into an OAT with n-nodes, keeping the resulting (n+1) nodes tree optimal. The idea is to emulate the effect of rebuilding the optimal (n+1) node tree using the Hu-Tucker algorithm [5,9] and making use of the information already obtained during the process of building the previous n-node tree. First, we show in detail how to achieve this when the new weight is inserted at the boundary of the tree, then we extend the algorithm to insertion in any arbitrary position. After introducing the algorithm for insertion, the one needed for deletion or even changing weights is simply done through insertion of a *negative* weight node.

2 Insertion at the Boundary

In this section, we examine the problem of building a new OAT with (n+1) nodes by inserting a new node at the boundary of a given n-node OAT. The old tree is obtained using the Hu_Tucker algorithm which proceeds in three phases.

Phase1

At each step, the two nodes that constitute a local minimum compatible pair (lmcp) are combined until a tree is obtained.

Phase 2

Uses the tree built in phase1 to find the level of each node.

Phase 3

Use a stack algorithm to construct an alphabetic binary tree based on the node levels.

Both phases 2,3 take time O(n). Phase1 can also be done in O(n) time for some special weight sequences, but will require O(n log n) time for a general sequence of weights. In the next section we will show how at each stage the known "*lmcp*" of the old tree can be used to find the "*lmcp*" for the new tree using only a constant number of steps at every stage, thus leading to a linear time algorithm for finding the new OAT.

Given a set of weights q_1,q_2 ..,q_n let q_{n+1} be the weight of the new node. The existence of the new node, $\mathbf{q_{n+1}}$, will not affect the process of combining nodes into (*lmcp*)'s until the node $\mathbf{q_n}$, is selected say with node **x**, at this point we will start building a list containing the rest of the *lmcp* for the new tree. The entries in this list will be called new nodes and by the rules of the Hu-Tucker combination phase, are always sorted. We have two possibilities (ties are broken in the Hu-Tucker algorithm with the convention that the node at the left has a smaller weight)

<u>a) $x < q_{n+1}$:</u> One comparison needed for this step

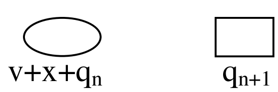
Then **x** is combined with $\mathbf{q_n}$ as before and we have the following sequence of nodes, shown in Fig.1 and (x+qn)becomes the first entry in the new nodes list and is marked as the head of the list.

Fig.1 one new node

Since the list of new nodes is always sorted, it is the head of the list that may combine to form a *lmcp* for the new tree, creating a new node and leaving its position as head of the list to the next node in line.

Let (**v**,**w**) be the pair that forms the *lmcp* for the old tree in the next step (assume **v** to the left of **w**). Then if w is $x+q_n$ then the *lmcp* of the new tree at this stage will be the smallest of the sums $(v,x+q_n)$, (v,q_{n+1}), $(x+q_n,q_{n+1})$ giving the weight sequences in Figs.2a,2b,2c respectively, the old weight sequence is shown in Fig. 2d.

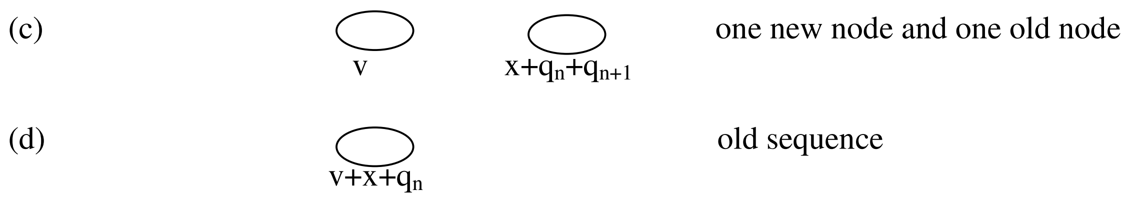
(a)



one new node

(b)

two new nodes



Figs. 2a - 2d

In Fig.2a the new node ($v+x+q_n$) becomes the head of the list. In Fig.2b, the node($v+q_{n+1}$)is added to the list of new nodes. The node ($x+q_n$) remains head of the list. For Fig.2c, the node ($x+q_n+q_{n+1}$)is now the head of the list. Node v in the weight sequence of Fig.2c will be called an old node, this is a node that existed in previous stages of the old tree weights sequence and reappeared in the new sequence at this stage. No more than one old node will ever appear in any weight sequence.

If the node ($x+q_n$) does not participate in forming the *lmcp* (v,w) then the *lmcp* for the new tree must be the smallest sum among the pairs (v,w), (v,q_{n+1}), (w,q_{n+1}) which can be decided in 2 comparisons and two adds provided that there are no external nodes between w and x.
Or in the case when w,x are separated by external nodes, the new *lmcp* is the smallest sum of the two pairs (v,w) and ($x+q_n,q_{n+1}$) which requires 1 comparison and 1 add. Fig.3 shows the resulting weight sequence for all the possible cases together with the weight sequence of the old tree at this stage.

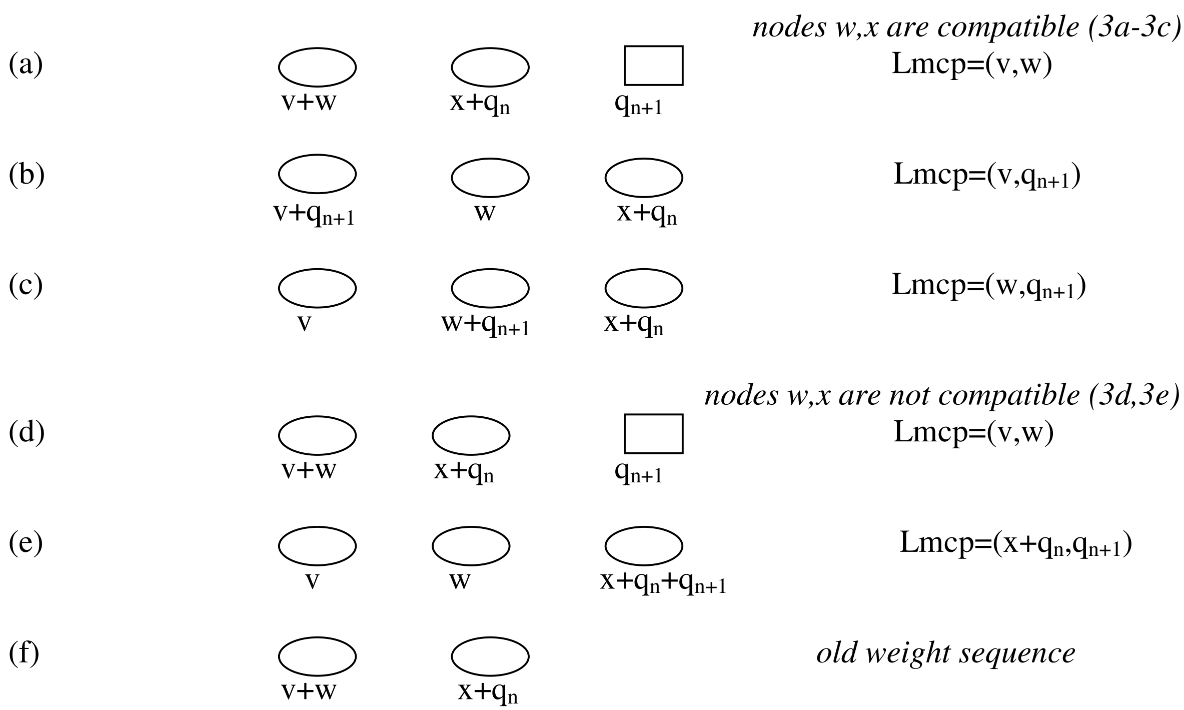


Fig. 3a-3f

Examining Fig.3, we see that the new weight sequences in Figs. 3a,3d only differ from the sequence of the old tree at this stage (Fig .3f) in the existence of the node qn+1, which is a similar situation as that of Fig.1 of the previous stage. Fig 3b,3c each has two new nodes and one old node. The head of the list is $(x+q_n)$in both case. In Fig. 3e, the node $(x+q_n+q_{n+1})$ becomes head of the list, note that the pair (v,w) is now a *lmcp* and must be combined into the node (v+w).

We now go one further step with the cases of Figs. 3b,3c,3e and show that the *lmcp* for the new tree can be obtained from the *lmcp* of the old tree at a given stage by only examining the head of the new nodes list and one old node if it exists. This means that the *lmcp* of the new tree can be obtained in a constant number of steps at each stage.

It suffices to consider the new sequence of Fig.3b and let the pair (v`,w`) be the *lmcp* for the next step (Fig .4).

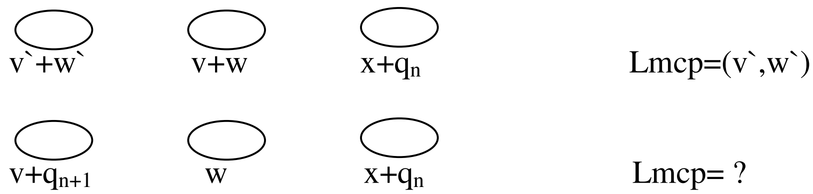


Fig. 4

To find the *lmcp* for the new sequence of weights, a key observation is to note that the old node (w) must belong to the new *lmcp* since it was part of the previously chosen lmcp (v+w),an old node if it exists must combine. This means that, the new *lmcp* is the minimum of the three sums (w,w`), (w,v`) and (w,head of the new nodes list) which requires two steps to find. The head of the list at this stage is $(x+q_n)$. The case when the *lmcp* = (w,v`) is shown in Fig. 5.

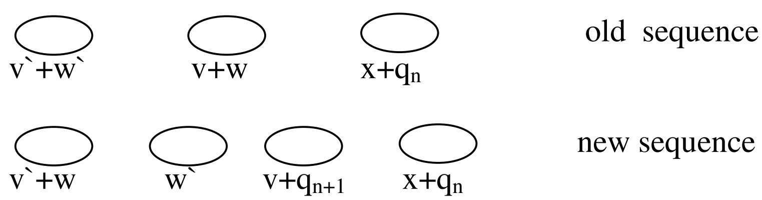


Fig. 5

The important point here is that the newly formed new node (v` +w) is of higher weight than the previously formed ones by the combination rules of the Hu-Tucker algorithm. The number of old nodes in the sequence will never exceed one and the new nodes are always sorted and therefore at most two nodes, one old node and the smallest new node, will affect the choice of the *lmcp* at each stage.

We must mention here that the previous discussion assumed that all considered nodes were compatible and can be combined since this gives the maximum number of steps needed to find the *lmcp*. The presence of external nodes will not contribute more than a linear time factor to the whole algorithm.
To see this, assume
weight sequence of Fig.4,then we simply use the *lmcp* (*lmcp* of the new tree at this stage and go to the next stage. This is possible because we know that amongst all the *lmcp* and
hence, by a result in [6],it will still form a *lmcp* later become

stage. The only additional work that needs to be done here is when we eventually reach the case of Fig.5 when the nodes are no longer incompatible. Then the currently formed *lmcp,* may be smaller in value than the *lmcp* formed when external nodes were present and must be inserted in its right position to keep the order of the *lmcp* consistent. If there number is k_1, this can be done in $\lceil \log k_1 \rceil$ steps. Therefore, the contribution to the whole algorithm will be no worse than

$\lceil \log k_1 \rceil + \lceil \log k_2 \rceil \quad \lceil \log k_r \rceil$ with $k_1 + k_2 \quad +k_r = n$ .This is no worse than O(n).

b) <u>$x > q_{n+1}$:</u>

The new node q_n+q_{n+1} is formed and the old node **x** reappears. At this step, the newly formed node has to appear in its correct order to keep the tree consistent. This is done by inserting it in the sequence of all previously formed *lmcp* . These *lmcp* together with the one inserted remain *lmcp* for the new tree. The list of new nodes is started with the node (q_n+q_{n+1}) as its head. Node **x** is now an old node and we consider the next *lmcp*, the one that comes after ($x+q_n$), of the old tree.

Identical arguments to those given for the case $x < q_{n+1}$ show that at every stage of forming the new OAT, the *lmcp* can be found in constant time.

3 Insertion in the Middle

The only case that will hinder the analysis given in the last section when a new node q is inserted between nodes *l*,*r* will arise when (*l*,*r*) become a *lmcp* in the old tree and neither *l* nor *r* combines with q to form the new *lmcp*, so we avoid this case.

Result 1: A new node q can be inserted in linear time between nodes *l*, *r* provided q is not bigger than both *l*, *r*.

4 Deletion

Deleting a node q between nodes *l*, *r* is accomplished by inserting the negative weight node (-q) between them. This works fine as long as the two nodes q, -q are guaranteed to be combined in the final tree thus forming a node of weight zero. The required tree is obtained by removing the node of weight zero and its two children and decrementing by one the level of the node that formed a *lmcp* with the zero node.

It is known [6] that for the consecutive four nodes a, b, c, d the two middle nodes will combine in the final tree if:

$a>c$, $b \le d$ and $(b+c) < \max(a,d)$

then for the sequence of four weights *l*, q,-q, *r* the condition reduces to $q \le r$ while for the sequence *l*,-q, q, *r* the condition is $l > q$. By noticing that when two consecutive nodes are equal then it does not really matter which one of the two gets deleted, then deletion will still work fine when $l = q$ and therefore deletion can be done provided that q is not bigger than both *l*, r.

Result 2: A node q between nodes *l*, r can be deleted in linear time provided q is not bigger than both *l*, r.

5 Conclusion

Optimal alphabetic trees can be constructed in linear time for some special weight sequences. The algorithm presented in this paper will allow us to do a constant number of linear time insertions and deletions on those trees thus expanding the number of OAT that can be constructed in linear time.

References

[1] Ahmed M.A., Belal A.A. and Ahmed K.M., "Optimal insertion in two-dimensional arrays", *International Journal of Information Sciences*, 99(1/2) : 1-20, June 1997.

[2] Andersson A., "A note on searching in a binary search tree", *Software-Practice and Experience*, 21(10) : 1125-1128, 1991.

[3] Belal A.A., Ahmed M.A., Arafat S.M., "Limiting the search for 2-dimensional optimal alphabetic trees", *Fourth International Joint Conference on Information Sciences*, October 1998.

[4] Garcia A.M. and Wachs M.L., "A new algorithm for minimum cost binary trees", *SIAM Journal on Computing*, 6(4): 622-642, 1977.

[5] Hu T.C. and Tucker A.C., "Optimal computer search trees and variable-length alphabetic codes", *SIAM Journal on Applied Mathematics*, 21(4): 514-532, 1971.

[6] Hu T.C., Morgenthaler J.D., "Optimum alphabetic binary trees", Combinatories nd Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Volume 1120 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp.234-243.

[7] Huffman D.A., "A method for the construction of minimum redundancy codes", *Proceedings of the IRE*, 40: 1098-1101, 1952.

[8] Klawe M.M. and Mumey B., "Uper and lower bounds on constructing alphabetic binary trees", *Proceedings of the 4th Annual AVM-SIAM Symposium on Discrete Algorithms*, page 185-193, 1993.

[9] Knuth D.E., "*The Art of Computer Programming, Volume 3: Sorting and Searching*", Addison-Wesley, Reading, MA, 1973.

[10] Przytycka T.M. and Larmore L.L., "The optimal alphabetic tree problem revisited", *Journal of Algorithms*, 28(1): 1-20, June 1997.