Towards a Dynamic Optimal Binary Alphabetic Tree

Ahmed A. Belal
Department of Computer Science & Automatic Control
Faculty of Engineering
University of Alexandria
Egypt

Mohamed S. Selim
Department of Computer Science & Automatic Control
Faculty of Engineering
University of Alexandria
Egypt

Shymaa M. Arafat
Computer Science Department
Faculty of Computer & Information Sciences
Ain Shams University
Egypt

Abstract

Optimal alphabetic binary trees have a wide variety of applications in computer science and information systems. Fast algorithms for building such trees in *O(n log n)* 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 an optimal 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 and allows us to expand the domain of weight sequences whose optimal alphabetic trees can be obtained in linear time.

Key words:

Optimal alphabetic tree, 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 natural 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 binary 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 binary 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 binary 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].It was recently shown in [6] that linear time algorithms for building optimal binary alphabetic trees are possible 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 The Hu-Tucker Algorithm

The algorithm proceeds in three phases

Phase1 : Combination

This is where most of the work is done.Every node before combining is a square node also called external node.If we let $\mathbf{q_i}$ denotes the weight of the node or the node itself then when two square nodes $\mathbf{q_i},\mathbf{q_j}$ combine they form a circular node also called internal node with weight $\mathbf{q_i+q_j}$ occupying the position of the left child.

Due to the alphabetic constraint,two nodes can only combine if they form a compatible pair.Two nodes in a sequence form a compatible pair if they are adjacent in the sequence or if all nodes between them are internal nodes.Among all compatible pairs in a weight sequence ,the one having the minimum weight is called the minimum compatible pair.

To break ties, the Hu-Tucker algorithm uses the convention that the node on the left has a smaller weight.

A pair of nodes ($\mathbf{q_j},\mathbf{q_k}$) is a **local minimum compatible pair** (*LMCP*) if

$\mathbf{q_i} > \mathbf{q_k}$ for all nodes q_i compatible with q_j

$\mathbf{q_j} \leq \mathbf{q_l}$ for all nodes q_l compatible with q_k

The first phase of the algorithm keeps forming *LMCP*s until a tree is formed.

Phase 2 : Assigning Levels

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

Phase 3 : Reconstruction

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

Both phases 2,3 take time *O(n)* while phase1 requires *O(n logn)* time.It was shown in [6] that phase1 can also be done in *O(n)* time for some special classes of weight sequences, as for example the increasing weight sequence $q_1 \leq q_2 \leq \ldots \leq q_n$.

In the next section we will show how after inserting a new node at the boundary of a known OAT, the new OAT is obtained in linear time. Using the known *LMCP*s of the old tree, the *LMCP* of the new tree at each stage can be obtained in *O(1)* time from a weight sequence of no more than **5 nodes** , thus leading to a linear time algorithm for finding the new OAT.

3 Insertion at the Boundary

Given a sequence of external nodes $\mathbf{q_1,q_2,\ldots,q_n}$ whose OAT is known,we show how to obtain in linear-time the OAT for the sequence $\mathbf{q_1,q_2,\ldots,q_n,q_{n+1}}$.

All we actually need to know, to find the new OAT is the sequence of *LMCP*s for the old tree which is formed during phase1 (the combination phase) of the Hu-Tucker algorithm.We will show how this sequence of *LMCP*s, which we will call the old tree list, can be used to obtain each *LMCP* for the new tree in *O(1)* time .This sequence of *LMCP*s will be called the new tree list and is thus obtained in *O(n)* time.Phases 2,3 of the Hu-Tucker algorithm each requiring *O(n)* time can now be applied to the new tree list to find the required OAT. Fig.1 shows the original node sequence and its associated old tree list with nodes $\mathbf{q_i,q_j}$ forming the first *LMCP*.

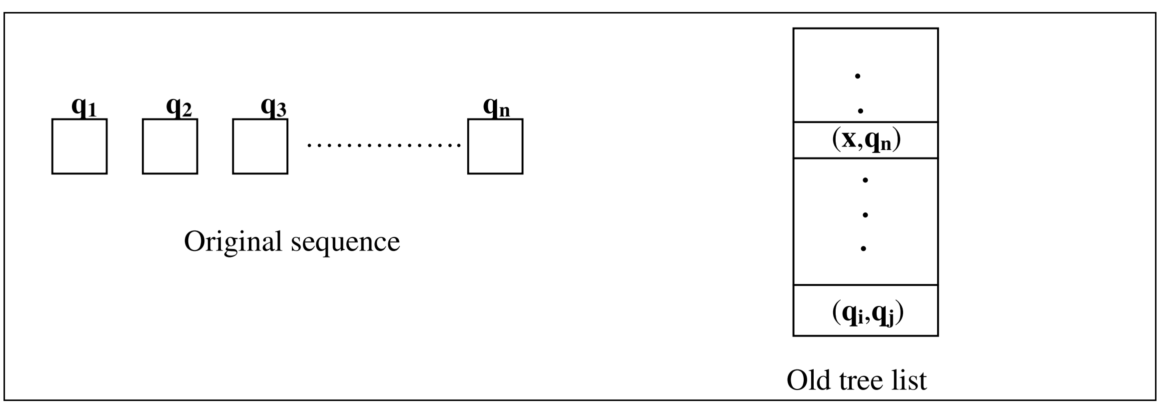


Fig.1

Because of the rules for combining nodes into *LMCP*s, the old tree list is in sorted order and has (n-1) entries.

Since the node $\mathbf{q_{n+1}}$ is inserted at the boundary of the original sequence to the right of $\mathbf{q_n}$, it cannot combine with any other node before the external node $\mathbf{q_n}$ does. Therefore the node $\mathbf{q_{n+1}}$ will not start to affect the formation of the *LMCP*s for the new tree until $\mathbf{q_n}$ appears in an entry in the old tree list. Previous entries in the old tree list will still be valid *LMCP*s for the new tree and will appear in the same order in the new tree list.

With this understanding we let the old tree list only contain the entries starting with the entry where $\mathbf{q_n}$ appears for the first time, combining say with node $\mathbf{x}$.The corresponding weight sequence for this list will now contain a mixture of internal(circular)and external(square)nodes with node $\mathbf{q_n}$ still being an external node.This old tree list will be used to determine a sorted list of entries for the new tree which can now be merged with the list of *LMCP*s that had formed before $\mathbf{(x,q_n)}$ to obtain the required list for the sequence $\mathbf{q_1,q_2,\ldots,q_n,q_{n+1}}$ of square nodes.

The first entry in the old tree list is $\mathbf{(x,q_n)}$.To find the first entry in the new tree list we must find the *LMCP* from the working sequence containing the nodes $\mathbf{x,q_n,q_{n+1}}$ as shown in Fig.2.

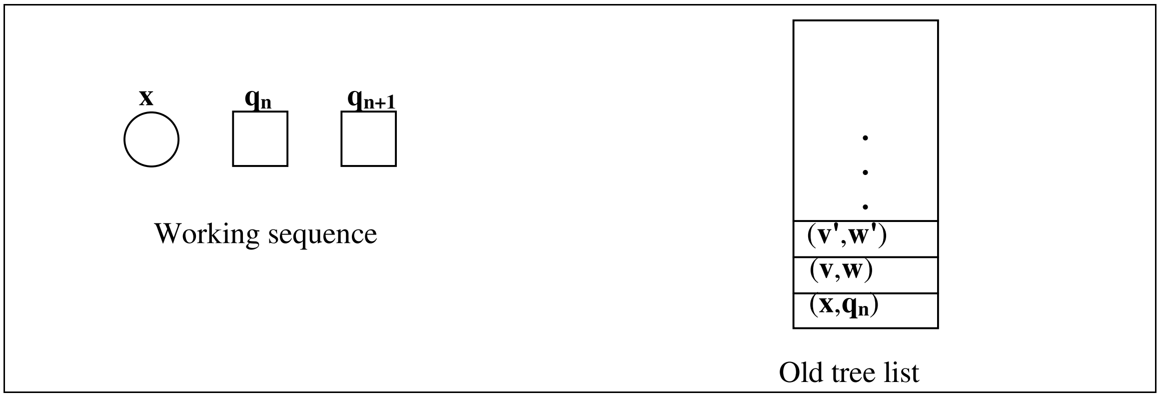


Fig.2

If $\mathbf{x}<\mathbf{q}_{n+1}$ then $\mathbf{x},\mathbf{q}_n$ combine as before to form an *LMCP*. The working sequence will now contain the nodes $\mathbf{v},\mathbf{w}$ of the next entry in the old tree list and the node $\mathbf{q}_{n+1}$, Fig.3a.

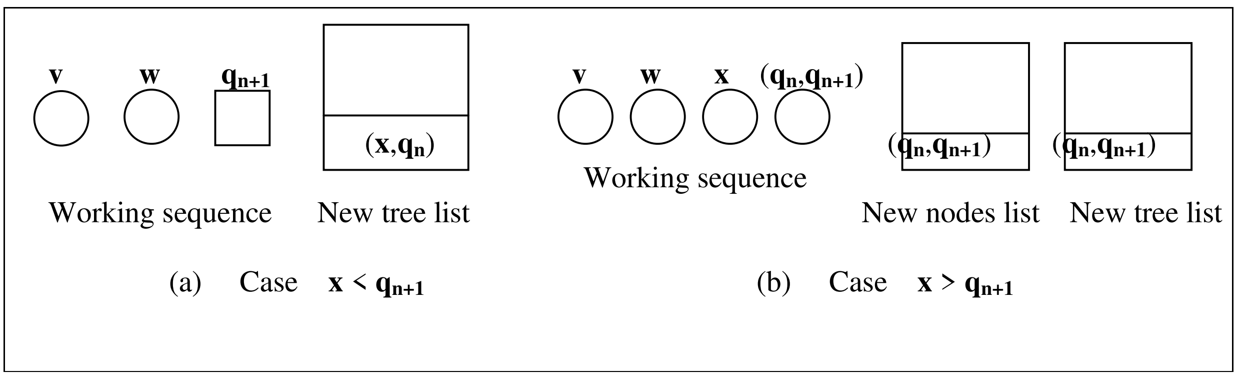


Fig.3

On the other hand, if $\mathbf{x}>\mathbf{q}_{n+1}$ then node $\mathbf{q}_n$ will combine with node $\mathbf{q}_{n+1}$ to form the *LMCP* and node **x** will be left in the working sequence.We call **x** a *left-over node*.Since $(\mathbf{q}_n,\mathbf{q}_{n+1})$ is a new *LMCP* that did not exist in the old tree list ,it is also entered in another list we call the new nodes list, Fig.3b.This list is needed to achieve a linear time algorithm.

New nodes list

During the course of the algorithm, and regardless of whether **x** is smaller or larger than $\mathbf{q}_{n+1}$, if an entry for the new tree list is found that was not present in the old tree list ,it must also be entered in *the new nodes list*. The new nodes list will thus contain all the entries in the new tree list except those entries that were present in the old tree list, and due to the combination rules for forming an *LMCP* the new nodes list is always sorted and its smallest two entries may affect the formation of an entry for the new tree list at any stage. A special marker should be used to indicate the head of the list (smallest element) position at every stage. When the head of the list is involved in a combination it leaves its position as head of the list to the next element in line.

The new nodes list has another important property. Since all its entries are *LMCP*s then if its two smallest entries combine at any stage to form an entry in the new tree list, the value for this newly formed *LMCP* must be larger than the largest entry in the new nodes list.The head of the list now beomes the next smallest and their combination is entered at the tail of the list.

New tree list

The new tree list, because it may contain entries from the old tree list, is not necessarily sorted but can be made sorted in linear time by simply identifying those entries that are in the new tree list but not in the new nodes list and merging them with the new nodes list.

Going back to Fig.2, if nodes $\mathbf{q_n}$,$\mathbf{q_{n+1}}$ combine as in Fig.3b leaving node $\mathbf{x}$ in the working sequence, the node ($\mathbf{x}$,$\mathbf{q_n}$) that had formed in the old tree list is no longer a valid node for the new tree and must therefore be deleted from all future entries in the old tree list in which it appears. This can be achieved by adding a pointer in every entry to the parent of the resulting node. This information is readily available for the old tree.The process of deleting the nodes will not take more than linear time for the whole algorithm,since the number of entries in the old tree list is just (*n-1*).

Another piece of information is also needed to be able to determine which nodes are allowed to combine together in the working sequence. This can be done by attaching to each node the identities of the first non-compatible node to its right and its left. For example in the working sequence a,b,c,.. of external nodes if the nodes a=1,b=2 where the first to combine, their entry would look something like a:1,b:2,3,*,c,p indicating that the formed node ab with value 3 can combine with any node to its left ,c is the first non-compatible node to its right and p is a pointer to its parent node.For reasons of clarity, these additional pieces of information are not shown in the figures.

Without loss of generality the following discussion is directed to the more general case of Fig.3b,the results and arguments apply as well to the case of Fig.3a when the inserted node eventually combines.

To find the next entry in the new tree list following the entry ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) in Fig.3b,the working sequence must clearly contain the left-over node $\mathbf{x}$, the new formed node ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) and the next *LMCP* from the old tree list, nodes $\mathbf{v}$,$\mathbf{w}$. It is important to note that *the left-over node* $\mathbf{x}$ *is compatible with the new node* ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$).The left-over node $\mathbf{x}$ will remain in the working sequence until it combines with the new node ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) forming another new node and leaving no left-over nodes in the working sequence, or by combining with either **v** or **w** creating another new node say (**w**,**x**) and leaving **v** as a left-over node.When the new nodes list contain more entries,the smallest two must always be brought into the working sequence before an entry to the new tree list can be determined.The following lemma provides an important observation.

Lemma 1: *The left-over node* v *is compatible to both nodes in the new nodes list.*

A left-over node gets created when a pair of nodes forming an entry in the old tree list is brought to the working sequence and one of them combines with a different node forming a new node and leaving its partner. The left-over node must therefore be compatible with the new node that its partner just formed. Node **v** is therefore compatible with node (**w**,**x**) and since node **x** is compatible with node ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) the result follows.

The result generalizes; if **v** now combines with **w'** to create a third entry in the new nodes list and leaves **v'** as a left-over node, then **v'** will be compatible to all three entries in the new nodes list.

The following lemma limits the number of left-over nodes

Lemma 2: *At most one left-over node exists in the working sequence at any stage.*

We prove this for the first created left-over node, then show that identical analysis applies to the most general case. The first left-over node will be created when the nodes $\mathbf{q_n}$, $\mathbf{q_{n+1}}$ combine. The working sequence will then, without loss of generality, contain the left-over node **x**, the only entry in the new nodes list ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) and the nodes **v**,**w** from the current entry in the old tree list as in Fig.3b. We will show that neither of the nodes **v**,**w** can combine with the node ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) leaving behind another left-over node with node **x**.

To show this, we go one step backwards when nodes **x**, $\mathbf{q_n}$ combined in the old tree list and consider all the different possibilities.

The square node $\mathbf{q_n}$ is the right-most node. Suppose that both nodes **v**, **w** are located between nodes **x**,$\mathbf{q_n}$ as in Fig.4a., then since **x** combined with $\mathbf{q_n}$ in the old tree, both nodes **v**, **w** must be circular nodes also compatible with $\mathbf{q_n}$ and therefore each must have a value larger than the value of **x** otherwise they would have combined with $\mathbf{q_n}$ in place of **x**. Fig.5a shows the position after nodes $\mathbf{q_n}$, $\mathbf{q_{n+1}}$ combine. With **v**, **w** larger than **x**, neither can combine before **x** does in Fig.5a. Node **x** in Figs.4a,5a can either be circular or square.

Fig.4b gives the second possibility with **x** assumed a square node, if **x** was circular case 4a applies. Here **w** must be circular and compatible with $\mathbf{q_n}$, and must have value larger than **x**. Node **v** is prevented from combining with ($\mathbf{q_n},\mathbf{q_{n+1}}$) in Fig. 5b, by the external node **x**. Node **v** can either be circular or square in this case.

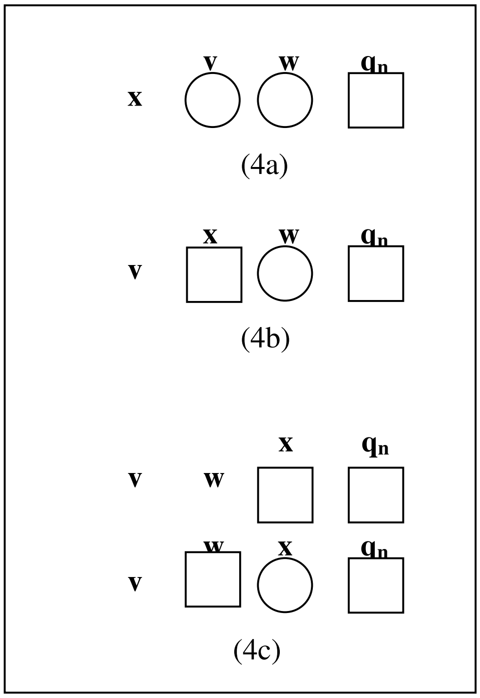


Fig.4

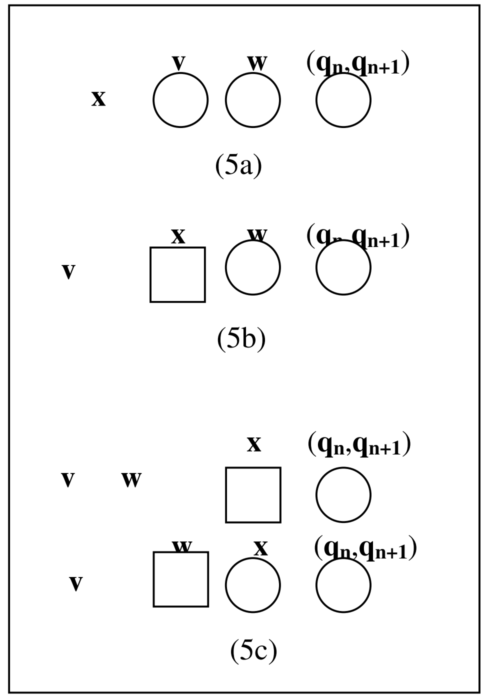


Fig.5

The last case is considered in Fig.4c with nodes **v**,**w** to the left side of **x**. In the first row of Fig.4c, node **x** is assumed to be a square node with **v**,**w** either circular or square. Node **x** will then prevent both **v**,**w** from combining with node ($\mathbf{q_n},\mathbf{q_{n+1}}$) in the first row of Fig.5c. The second row in Fig.4c considers the case when **x** is a circular node and **v** is either circular or square. If node **w** is circular then case 4a applies, so **w** is assumed square. If **w** is compatible with **x** in the second row of Fig.4c, it must also be compatible with $\mathbf{q_n}$ and therefore must have weight larger than that of **x**, and hence it cannot combine with ($\mathbf{q_n},\mathbf{q_{n+1}}$) in the second row of Fig.5c before **x** does. Node **v** cannot reach node ($\mathbf{q_n},\mathbf{q_{n+1}}$) because **w** is a square node.

The most general case is provided after node **x** combines with **w** in Fig.3b, to form the entry (**w**,**x**) in both the new tree list and the new nodes list , leaving **v** as a left-over node in the working sequence. Fig.6 shows the various lists and the working sequence at this stage.

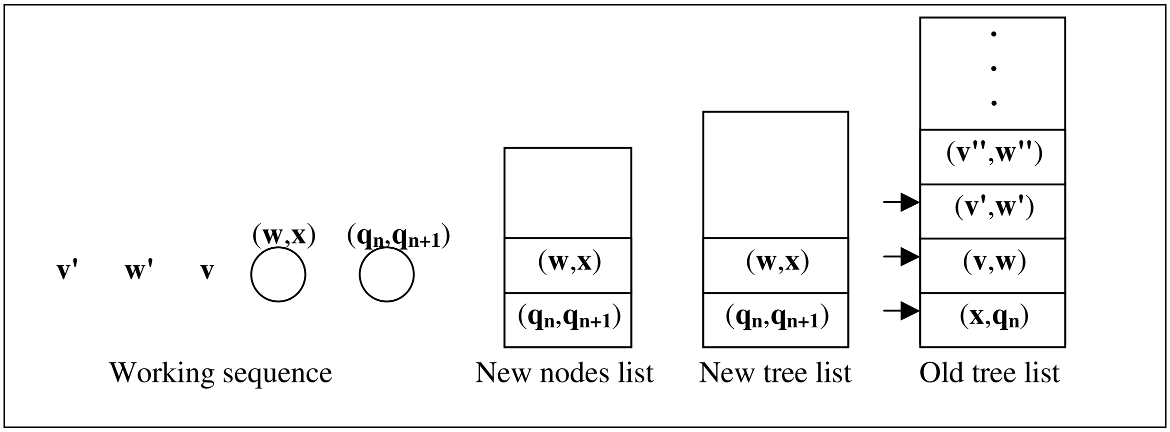


Fig. 6

By lemma 1,the two nodes ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) and (**w**,**x**) from the new nodes list are compatible to each other and since they are both circular nodes, we can safely place them next to one another in the working sequence with ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) occupying the right-most position. An identical argument to the one given before will show that neither **v'** nor **w'** can combine with either ($\mathbf{q_n}$,$\mathbf{q_{n+1}}$) or (**w**,**x**) leaving another left-over node with **v**.The argument for one new node holds for two compatible circular new nodes..

With lemma 2 assuring us of no more than one left-over node in the working sequence at any stage, every *LMCP* for the new tree can thus be obtained by examining at most five nodes: the smallest two entries in the new nodes list, one possible left-over node and one or two nodes from the old tree list entry as shown in the working sequence of Fig.6, as either **v'** or **w'** might have been deleted in a previous step. The next entry (**v''**,**w''**) cannot affect the formation of the *LMCP* at this stage and is not needed in the working sequence.The reason for this follows a similar argument as that given for lemma 2. Neither **v''** nor **w''** can combine before **v'**, nor can they combine with **v'** because of the existence of **w'** and the left-over node **v**. Node **v'** must consider **w'** and **v** before thinking of combining with either **v''** or **w''**. The same goes for **w'** so it is not necessary to look at the next entry (**v''**,**w''**) from the old tree list. Node **v'** was prevented from combining with either **v''** or **w''** because of its partner node **w'** and the left-over node **v**, but what if node **v'** had no partner node **w'** and the working sequence had no left-over node then in this case the next entry (**v''**,**w''**) from the old tree list must be examined, again giving at most *five* nodes in the working sequence, hence lemma 3.

Lemma 3 : *Every LMCP of the new tree can be determined by examining no more than five nodes in the working sequence.*

Blocked *LMCP*s

One last case must be considered and luckily it will just simplify matters. When an entry say (v’,w’)is read from the old tree list into the working sequence as in Fig.6 and both nodes v’,w’ are not compatible with the remaining nodes in the working sequence, we say that (v’,w’)is a blocked *LMCP* and is simply removed from the working sequence and entered as an *LMCP* in the new tree list.The reason for this is the fact that since amongst all nodes compatible with v’,it was w’ that combined with it to form an *LMCP* for the old tree at this stage ,then w’ will still combine with v’ for the new tree since no more members were added to the set of nodes compatible with v’. This blocked *LMCP must be removed from the working sequence and entered in the new tree list before any combination involving the remaining nodes in the working sequence can be considered as it could be affected by the next entry in the old tree list.*

In conclusion, the general rule is simple. To form an *LMCP* the working sequence must contain at least two compatible nodes from the old tree list (no more than 3 are ever needed) with at least one of them compatible to the nodes from the new nodes list.

The following example demonstrates most of these points. Fig.7 gives an 18 node weight sequence and the *LMCP* list of its OAT showing only the first six entries. It is required to find the OAT after the node Z=2 is inserted at the right end of the given weight sequence.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
30	20	18	20	3	9	2	8	8	1	9	3	6	3	4	2	1	6

Original sequence

. . .	
(g,h)	10
(pq,r)	9
(l,m)	9
(i,j)	9
(n,o)	7
(p,q)	3

LMCP list

Fig. 7

Our old tree list will contain a list of *LMCP*s beginning with the entry (pq,r) in Fig.7. This list, together with the weight sequence of nodes corresponding to it are shown in Fig.8. For ease of illustration the nodes in this figure are renamed as the sequence A,B,C,..,N, node a appears as node A, node pq as M and node r as N.

A	B	C	D	E	F	G	H	I	J	K	L	M	N
30	20	18	20	3	9	2	8	9	9	9	7	3	6

Reduced weight sequence

(AGHMNIK,BCDEFJL)	153
(BC,DEFJL)	86
(A,GHMNIK)	67
(D,EFJL)	48
(B,C)	38
(GHMN,IK)	37
(EF,JL)	28
(GH,MN)	19
(I,K)	18
(J,L)	16
(E,F)	12
(G,H)	10
(M,N)	9

Old tree list

Fig.8

To determine the first entry in the new tree list, the working sequence will contain the nodes M,N and the inserted node Z. The new tree list, the new nodes list and the working sequence are shown in Fig.9 for the first four steps of the algorithm. The old tree list at the end of the four steps is also shown where an asterisk denotes an entry that was deleted.

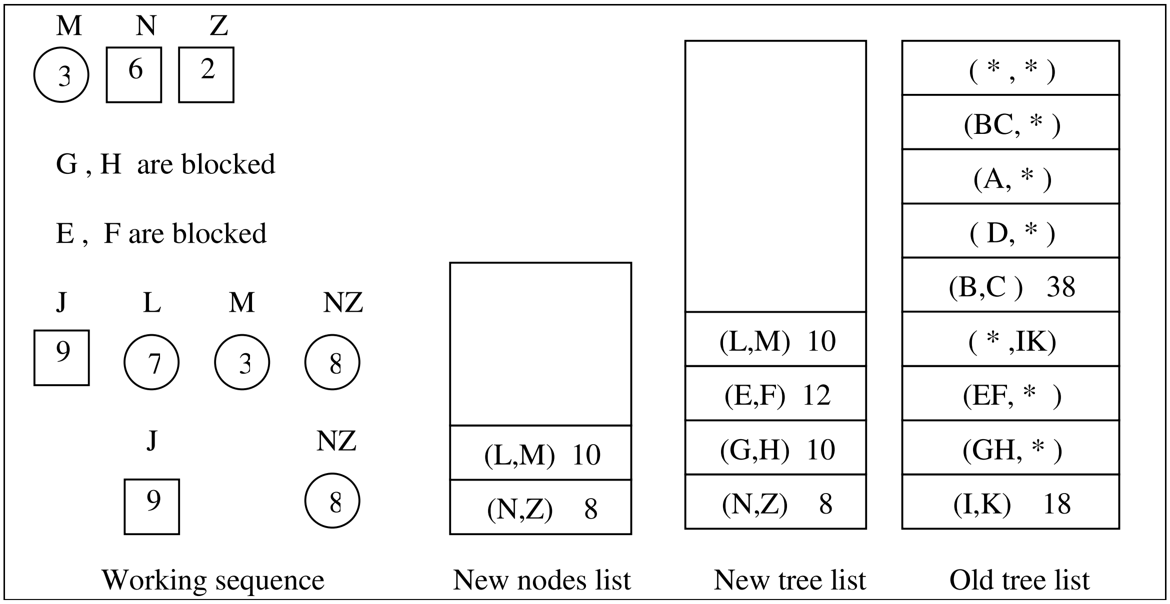
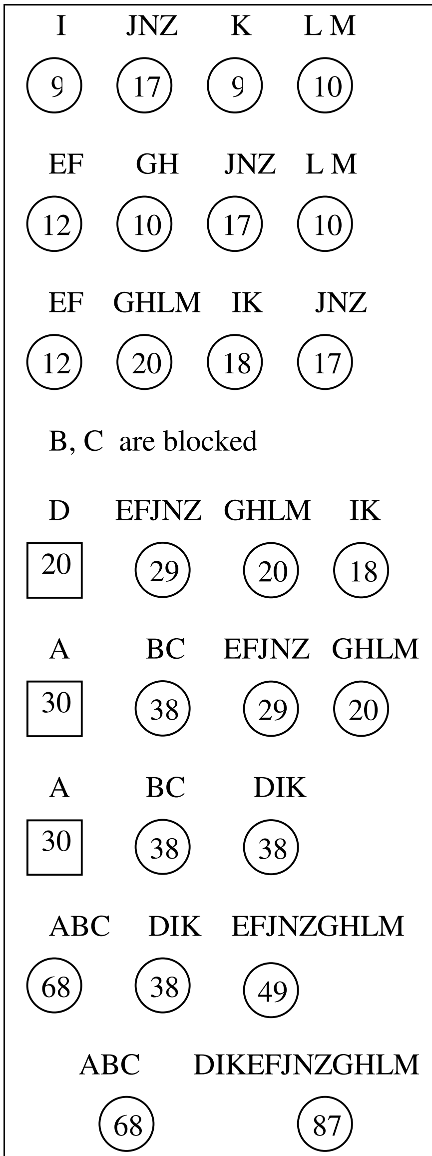


Fig. 9

Next we add the entries I=9,K=9 and (L,M)=10 to the working sequence and find the LMCP (J,NZ) =17.
The remaining steps and the final lists are shown in Fig.10.

The new tree list of Fig.10 is not sorted. To obtain a sorted list we look at the difference between this list and the new nodes list. This difference will produce a list of sorted entries that are members of the old tree list. In our example, these will be the four entries (G,H)10, (E,F)12, (I,K)18 and (B,C)38. Merging these entries with the new nodes list provides a sorted new tree list. Fig.11 shows this sorted new tree list together with the part of the original *LMCP* list of Fig.7 below the entry (pq,r).The two lists of Fig.11 are now merged to get the required final *LMCP* list of the new optimal alphabetic tree.



Working sequence

(ABC,DIKEFJNZGHLM)	155
(DIK,EFJNZGHLM)	87
(A,BC)	68
(EFJNZ,GHLM)	49
(D,IK)	38
(EF,JNZ)	29
(GH,LM)	20
(J,NZ)	17
(L,M)	10
(N,Z)	8

New nodes list

	(ABC,DIKEFJNZGHLM)	155
	(DIK,EFJNZGHLM)	87
	(A,BC)	68
	(EFJNZ,GHLM)	49
	(D,IK)	38
→	(B,C)	38
	(EF,JNZ)	29
	(GH,LM)	20
→	(I,K)	18
	(J,NZ)	17
	(L,M)	10
→	(E,F)	12
→	(G,H)	10
	(N,Z)	8

New tree list

Fig.10

(ABC,DIKEFJNZGHLM)	155
(DIK,EFJNZGHLM)	87
(A,BC)	68
(EFJNZ,GHLM)	49
(D,IK)	38
(B,C)	38
(EF,JNZ)	29
(GH,LM)	20
(I,K)	18
(J,NZ)	17
(E,F)	12
(L,M)	10
(G,H)	10
(N,Z)	8

Sorted new tree list

. . .	
(l,m)	9
(i,j)	9
(n,o)	7
(p,q)	3

Lower part of original *LMCP* list

Fig. 11

4 Insertion in the Middle

The main result of this section is entirely based on the analysis given in the previous section where we showed that when a node is inserted at the boundary of a weight sequence,the new OAT can be obtained from the old one in linear time.

Now assume that a new node **q** is to be inserted not at the boundary but between the two nodes **l**,**r**. When either node **l** or node **r** first appear in an entry of the old tree list we proceed as before to form the entries for the new tree list and the new nodes list.Both lemmas 1,2 are easily seen to apply in this case.

To be able to proceed with the formation of the *LMCP*s for the new tree we must make sure that when two nodes **L**,**R** on opposite sides of **q** appear together as an *LMCP* in the old tree list then either **L** or **R** combines with **q** to form the new *LMCP*, otherwise we get stuck and cannot proceed as before. Of course we may become lucky and this does not occur until the very end, nodes combine nicely on each side of q forming a single node on its left and a single node on its right. But we may not be that lucky,and therefore must prevent such an occurrence during the entire course of building the new tree list.

Condition 1: *The new node **q** to be inserted between nodes **l**, **r** cannot be bigger than both **l**,**r**.*

With this condition we ensure that at the stage when any two nodes **L**,**R** to the left and right of **q** respectively, combine to form an *LMCP* for the old tree,then either **L** or **R** must combine with **q** to form the *LMCP* for the new tree and we can proceed as before.

The same general rule for finding *LMCP*s when insertion is at the boundary still applies exactly when a node q is inserted in the middle *but to both the left and right sides of q.*

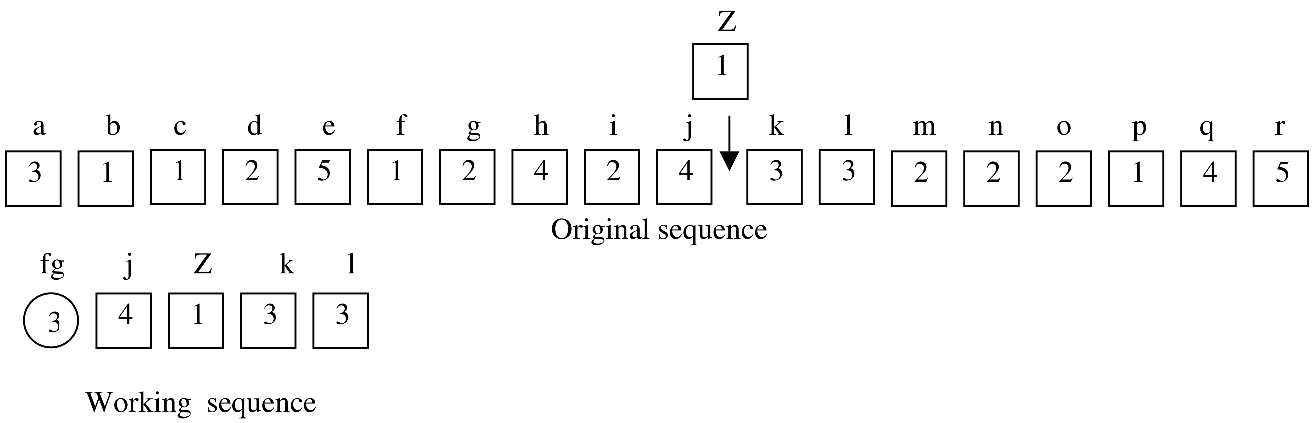
Lemma 4 :*Every LMCP for the new tree can be determined by examining no more than eight nodes.*

When insertion is at the boundary the maximum number of nodes in the working sequence at any stage is five, as shown in Fig.6 with w' compatible with the left-over node v.It was shown that the next entry (v'',w'')from the old tree list when compatible with either v' or w' cannot affect the formation of the *LMCP* at this stage .When both v'',w'' are not compatible with either v' or w' it is a blocked *LMCP* and again cannot affect the formation of an entry at this stage. Being blocked from v',w' make them blocked from the rest of the nodes in the working sequence. Not so when insertion is not at the boundary, the pair of nodes v'',w'' may still be blocked from v',w' but can reach the other nodes in the working sequence from the right side,as in Fig.6 with v a square node.

To find the *LMCP* at any stage, the working sequence will contain the smallest two nodes in the new nodes list, one left-over node and at most two nodes to the left of q and two nodes to the right of q from the old tree list for a total of at most seven nodes. When there is no left-over node then we may need up to three nodes to the left and three nodes to the right of q from the old tree list for a total of at most eight nodes.

Implementation

To be able to identify nodes to the left and right of q,the old tree list is divided into two lists the left list and the right list containing respectively the entries lying to the left and right of q.If an entry in the old tree list is formed from one node to the left of q and one node to the right of q then the entry is split with the left node entered in the left list and the right node in the right list.This is demonstrated for the 18 node sequence of Fig.12 with the node Z=1 to be inserted between nodes j,k.



(abcdklfgjmnop,eqhir)	47
(abcdkl,fgjmnop)	27
(eq,hir)	20
fgj	7
abcd	7
hi	6
e	5
(a,bcd)	7

Left list

mnop	7
kl	6
r	5
q	4
(mn,op)	7

Right list

(abcdklfgjmnop,eqhir)	47
(abcdkl,fgjmnop)	27
(eq,hir)	20
(fgj,mnop)	14
(abcd,kl)	13
(hi,r)	11
(e,q)	9
(mn,op)	7
(fg,j)	7
(a,bcd)	7
(k,l)	6

Old tree list

(h,i)	6
(m,n)	4
(bc,d)	4
(o,p)	3
(f,g)	3
(b,c)	2

Lower part of *LMCP* list

Fig. 12

The lower part of the *LMCP* list contains all the *LMCP*s formed before either j or k combine. Since the insertion is between nodes j and k,the two *LMCP*s (k,l) and (fg,j) involving the two nodes j,k in the old tree list are placed in the working sequence with node Z and the remaining *LMCP*s of the original tree are divided into the left and the right lists. The left list will contain node j and all the nodes to its left while the right list will contain node k and all nodes to its right. The first working sequence will contain the nodes Z,k,l,fg,j.Since (k,l) appeared before (fg,j) in the old tree list and both these entries must be brought to the working sequence,all entries in the left list forming before the entry (fg,j) will represent blocked *LMCP*s and will remain as valid entries in the new tree list ,in our example the entry (a,bcd). The different working sequences and the final new nodes and new tree lists are shown in Fig.13.The new tree list contains all the entries in the new nodes list together with entries from both the left and right lists and is therefore not necessarily sorted. These entries from the left and right lists can be easily identified and marked with an (L) or an (R).The list of entries marked with (L), the list of entries marked with (R) and the new nodes list are merged together to produce a sorted new tree list. This new tree list is next merged with the lower part of the *LMCP* list to find the required list of *LMCP*s for the new weight sequence. All this merging is done in *O(n)* time, hence the following result

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**.*

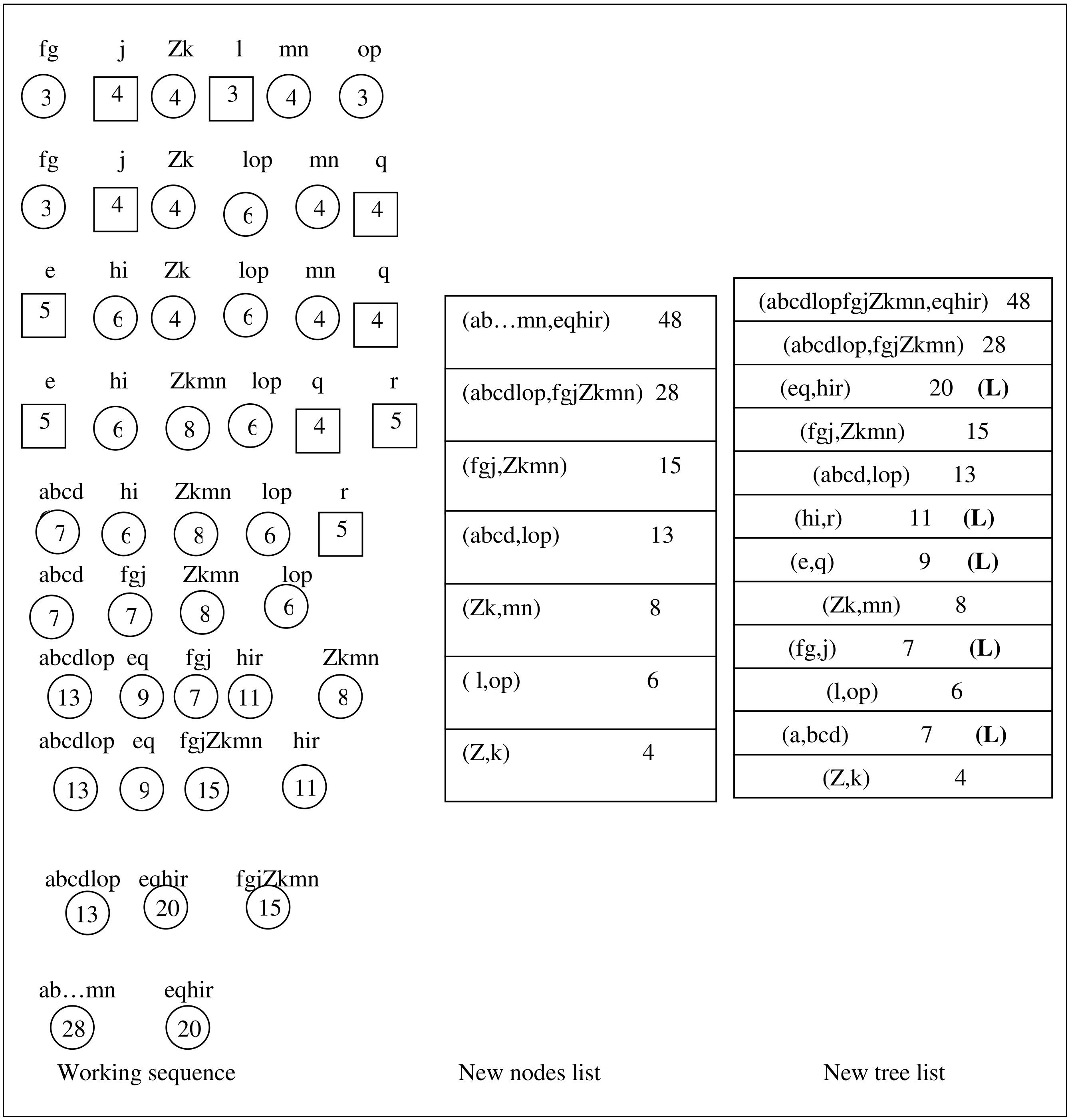


Fig.13

5 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 an *LMCP* with the zero node.

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

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

Therefore for the sequence of four weights **l**, **q**,**-q**, **r** the condition reduces to $\mathbf{q} \leq \mathbf{r}$ while for the sequence **l**,**-q**, **q**, **r** the condition is $\mathbf{l} > \mathbf{q}$. By noticing that when two consecutive nodes are equal 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

Binary 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 such trees 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", Combinatorics and 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 ACM-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.