A LINEAR TIME ALGORITHM FOR THE LOWEST COMMON ANCESTORS PROBLEM

Dov Harel

University of California, Irvine
Irvine, CA 92717

Technical Report #155
August 1980

Keywords: Lowest common ancestors, balanced trees, threads linear algorithms, union find, indexing, random access machine, pointer machine, reference machine.

CR categories: 3.73, 3.74, 4.34, 5.25.

This research was supported by Earl C. Anthony Fellowship and National Science Foundation grant MCS79-04997.

A LINEAR TIME ALGORITHM FOR THE LOWEST COMMON ANCESTORS PROBLEM

Dov Harel

Abstract

We investigate two lowest common ancestors (LCA) problems on trees. We give a linear time algorithm for the off-line problem, on a random access machine (RAM). The half-line problem is one in which LCA queries on a fixed tree are arriving on line. We extend our RAM algorithm to answer each individual query in O(1) time, with O(n) preprocessing time. Tarjan observed that this result helps to explicate the difference in power between RAM and pointer machines. We also show how to modify our algorithm to achieve a linear preprocessing time, optimal query time, algorithm on a reference machine.

1.0 Introduction

In [AHU76] three lowest common ancestors (LCA) problems are discussed. In the first one, which is called the on-line problem, we assume that queries, which are pairs of vertices, arrive on line, and the tree changes dynamically. An O(n log n) algorithm is given for the execution of n LCA queries and LINK commands. The on-line problem is discussed also by [M79], where some improvements of the previous result are given, mainly with regard to space complexity. The latter paper also gives an application of the on-line algorithm to detection of negative cycles in sparse graphs.

The second problem which is discussed in [AHU76], the off-line problem, is one in which we are allowed to look at the entire sequence of queries before producing any answers. The algorithm given there, which is also referred to in exercise 4.38 of [AHU74], is a well-known use of the disjoint UNION_FIND algorithm, which is analyzed in [T75,T77]. An n-query problem has time complexity $O(n\,\alpha(n))$, where $\alpha(n)$ is related to an inverse of Ackermann's function. In [T77] it is shown that $\Theta(n\,\alpha(n))$ operations are necessary for solving the union-find problem on a reference machine. The model of computation used, reference machines, does not allow indexing,

although Tarjan conjectured that this restriction is not essential for the lower bound. A natural question is whether $O(n\,\alpha(n))$ is a lower bound for the lowest common ancestors problem as well. In this paper, section 3, we show that the answer is negative if we allow the use of indexing. The problem investigated in section 3 is somewhat more general than the version of [AHU76] in that we do not assume that the number of queries is equal to the size of the tree.

In section 4 we discuss a version of a problem of intermediate difficulty from [AHU76]. We assume that the tree is fixed, and that queries are arriving on-line. Following [T80] we will refer to this problem as the half-line problem. We give an $O(n)$ preprocessing time algorithm which answers each LCA query in $O(1)$ time, which is an extension of our off-line algorithm to the half-line case. Tarjan [T80] gives a lower bound of $\Omega(\log \log n)$ time per query on the performance of a reference machine [T77], and thus observes that our results help to explicate the difference in power between random access machines and pointer manipulating machines. We also show how to apply our methods to pointer machines to get an $O(\log \log n)$ query time with only $O(n)$ preprocessing time. The previous algorithm [AHU76] requires $\Theta(n \log \log n)$ preprocessing time.

2.0 Threads and Inorder Numbers

Throughout this paper, we assume that each node in a binary tree has zero or two children. By a complete binary tree we mean a binary tree all of whose leaves are at the same depth. All logarithms are to the base two.

In order to search for lowest common ancestors efficiently we will introduce threads in our binary trees, which are, in fact, the same threads used in [HL79]. These are very similar to the threads which were used by [GMPR77] for B-trees. If x is a leaf node, define its right thread, written R_THREAD(x), to be its lowest right ancestor, that is, its successor in inorder. If x is an internal node, define R_THREAD(x) to be its rightmost leaf descendant, that is, the last node we visit if we repeatedly follow right links from x. The left thread of x, L_THREAD(x), is defined symmetrically. Note that those threads are different from the traditional threads which appear commonly in the literature [PT60,K68,S80] in that traditional threads are kept only at leaves.

In addition to threads, we will keep with each node x its inorder number, written NUM(x), which is the position of x in the inorder traversal sequence of the tree. That is, NUM(x)=i iff x is the i^{th} node in the sequence.

Using threads and inorder numbers we can compute a number of other quantities in O(1) time. For example, let RIGHTMOST(x) (resp. LEFTMOST(x)) be the rightmost (resp. leftmost) descendant of x. Then RIGHTMOST(x) could be calculated as

if x is a leaf then x else R_THREAD(x);

Given two nodes x and y in T with NUM(x) $\leq$ NUM(y) the interval I(x,y) is defined by I(x,y) = {z in T | NUM(x) $\leq$ NUM(z) $\leq$ NUM(y)}. The distance between x and y in T, denoted d(x,y), is the cardinality of I(x,y), and is easily calculated by d(x,y) = NUM(y) - NUM(x) + 1. The reason is that NUM is a 1-1 correspondence between I(x,y) and the interval [NUM(x),NUM(y)] in the natural numbers domain. For a node x of T we will denote by T(x) the subtree of T rooted at x. The rank of x, written r(x), is defined differently for binary or general trees. For binary trees we use the traditional definition [NR73,L79] r(x) is defined to be one plus the number of descendants of x. For general trees we use r(x) to denote the number of leaf descendants of x. The reader is warned that the ranks used in this paper are not to be confused with the rank groups that are used in [AHU76] in a partial analysis of the union-find structure.

Given two nodes x and y we can in O(1) time answer questions like "is x an ancestor of y?" Let x_1 (resp. x_2) be LEFTMOST(x) (resp. RIGHTMOST(x)) and let y_1 and y_2 defined similarly with respect to y; then x is an ancestor of y iff T(y) is a subtree of T(x) iff [NUM(y_1),NUM(y_2)] is a subset of [NUM(x_1),NUM(x_2)], which could be answered in O(1) time.

Let LRA(x) (resp. LLA(x)) denote the lowest right ancestor (resp. the lowest left ancestor) of x; then LRA(x) is computable by

LRA(x) = R_THREAD(RIGHTMOST(x));

Notice that either LRA(x) or LLA(x) is the parent of x, and thus PARENT(x) can be found in O(1) time. Notice also that the successor of x in inorder, written NEXT(x), is computable in O(1) time by

```
if x is a leaf then R_THREAD(x)
    else LEFTMOST(RIGHT(x));
```

Many of the above remarks apply to threaded binary search trees with keys playing the role of the inorder numbers. In [HL79] a finger search procedure is given which does not require neighbor pointers, but uses threads instead. It is also shown in the above paper how to update the threads with a constant amount of effort per rotation in a binary tree; thus threads may be superior to neighbor pointers in situations where rotations are used to rebalance the tree (e.g. AVL trees [AVL62,K73,S80], BB(α) trees [NR73], the Dichromatic Framework [GS78] etc.)

It will sometimes be convenient to add the tree as a parameter of the function. For example LRA(x,T) means the lowest right ancestor of x in T. This notation is useful when some nodes are elements of more than one tree. Also, if T is some tree, and x is a node of T we will use T(x) to denote the subtree of T rooted at x.

3.0 The Off-Line Problem

By the off-line lowest common ancestors problem, abbreviated hereafter as OFLCA, we mean the following. Given a tree T on n nodes, and a list Q of q pairs of vertices of T, which we call queries, find for each pair (x,y) the lowest common ancestor of x and y in T. We assume that we are allowed to read the entire list Q, as well as the tree T, before producing an answer to any of the queries.

In section 3.1 we give an $O(n + q)$ algorithm for the OFLCA problem on complete binary trees, and in section 3.2 we show how to extend the algorithm in 3.1 to any balanced binary tree. Section 3.3 contains a linear transformation of the problem on general trees to an OFLCA problem on union trees, and finally in section 3.4 we show how to linearly transform the problem on union trees to an OFLCA problem on balanced binary trees.

3.1 A Linear Time Algorithm for Complete Binary Trees

Let T be a complete binary tree of height h, and let Q be a list of q queries on T. We will show how to answer all the queries in linear time in the size of the input, that is, in O(n + q) time. First notice that, without loss of generality, we may assume that for any query (x,y), x and y are not ancestors of each other, since using threads such queries could clearly be answered in linear time. Second, we may assume, again without loss of generality, that queries pertain only to leaves. That is, if q=(x,y) is a query, we may assume that x and y are leaves, since otherwise we can replace x by LEFTMOST(x), and y by LEFTMOST(y), without affecting the answer to the query.

To illustrate the idea behind our algorithm, we first show how to answer a single query efficiently, in a complete threaded binary tree with inorder numbers. Let x and y be leaves in T, such that x precedes y in inorder (i.e. NUM(x) $\leq$ NUM(y)), and let d be the linear distance between x and y in T, d = d(x,y). Examine the following procedure.

```
procedure FIND_LCA0(x,y);
begin
  v := x;
  while v is not an ancestor of y do v := LRA(v);
  return v;
end;
```

Claim 0. The procedure FIND_LCA0(x,y) returns the lowest common ancestor of x and y, in O(log d) steps.

Proof. Assume that the while loop of FIND_LCA0 was entered at least once. Let v_0 be the last value v accepts before it subtends y, and let v_1 be the right child of v_0. Then the complete subtree rooted at v_1 lies between x and y. Since clearly the time complexity of the program is O(log $r(v_0)$) and $r(v_0)$ is O(d) the claim follows. []

In order to answer all queries in linear time, we will use bucketting techniques which will first give incorrect, approximate answers. From each approximate answer, the correct answer is computable in O(1) time per query, with the aid of the threads.

The algorithm consists of two main procedures, the first of which is called PARTITION, and is responsible for the following two tasks. First, it threads the tree in the fashion discussed in section 2, calculating and storing inorder numbers, ranks, and heights at all the nodes. It then uses those fields to

a) Partition the tree into h sets C_i which correspond to the levels of T. $C_i = \{x \mid r(x) = 2^i\}$, and C_i is ordered by inorder.

b) Partition the queries into h buckets, denoted by B_i, with $B_i = \{q=(x,y) \text{ in } Q \mid 2^i \leq d(x,y) < 2^{i+1}\}$, and B_i is ordered by increasing NUM(x).

Clearly, all the above could be accomplished in $O(n + q)$ time. (For example (b) can be accomplished by a two pass radix sort on the queries, queries using $\langle \lfloor \log d(x,y) \rfloor , \text{NUM}(x) \rangle$ as the key of the query (x,y).)

The procedure FIND_CA finds, for each query q=(x,y) in B_i, the ancestor of x in C_i, written C_ANCESTOR(x,i). For each i, we do this by merging the queries in B_i into the cut C_i in such a way that each $v \in C_i$ is followed by all the queries q=(x,y) in Q for which x is a descendant of v. This merge takes $O(|C_i| + |B_i|)$ time. Finally the simple procedure FIND_LCA1 computes the lowest common ancestor of x and y, LCA(x,y), from C_ANCESTOR(x,i) in one or two steps. The procedures PARTITION, FIND_CA, and FIND_LCA1 are given in the appendix.

To see that the time complexity of FIND_CA is linear, notice that during the i^{th} iteration of the main for loop, v can be advanced at most $|B_i|$ steps, and q can be advanced up to a total of $|C_i|$ steps, giving a total time of $O(|C_i|+|B_i|)$ for the i^{th} iteration through the main loop. Summing up from 1 to h, the $|C_i|$-s sum up to n, since C_i, i=1...h is a partition of T, and the $|B_i|$-s sum up to q, since B_i, i=1...h is a partition of Q. Thus the total time spent by FIND_CA is $O(n + q)$.

Let q=(x,y) be a query in B_i, and let u be C_ANCESTOR(x,i) in C_i.

Observation 1. u is not an ancestor of y.

Proof. $r(u) = 2^i$ so the cardinality of T(u) is 2^i-1. The cardinality of the interval I(x,y), on the other hand, is $d(x,y) \geq 2^i$, so I(x,y) can not be a subset of T(u), and thus y is not a descendant of u. []

Observation 2. Let v = LRA(u) and w = LRA(v). Then w is an ancestor of y.

Proof. Denote the right child of w by z. $r(z) \geq 2^{i+1}$ and so $|T(z)| \geq 2^{i+1}-1$. If w is not an ancestor of y then T(z) lies strictly between x and y, which implies that $d(x,y) \geq |T(z)| + 2 > 2^{i+1}$. This is a contradiction to the fact that (since (x,y) is in B_i) $d(x,y) < 2^{i+1}$. []

It follows from the above argument that FIND_LCA1(x,y) correctly computes the lowest common ancestor of x and y, which completes the proof of the following lemma.

Lemma 1. Our algorithm solves a q-query OFLCA problem on an n-node complete binary tree in O(n + q) time.

It is worth noting at this point that the concepts and algorithms in this section extend naturally to BB(α) trees [NR73], which in fact motivated my approach. Further extensions are given in the following sections.

3.2 Extending the Result to Balanced Binary Trees

In this section we show how to extend the result of the previous section to any balanced binary tree. By a balanced binary tree we mean a tree T, on n nodes, with height h(T) which is O(log n). Let T be a balanced binary tree on n nodes. Let h be the height of T, and assume $h \leq k \log n$ for some constant k. We will see later that the constant k may appear as a multiplicative constant in the time complexity of our modified algorithm, depending on the model of computation we use.

We will first try the following simple minded approach. let T^* denote the complete binary tree of height h. We can look at T^* as an extension of T, and clearly we can answer the queries on T as queries on T^*. Since we have a linear algorithm for T^*, it may seem that this algorithm is also a linear algorithm for

T. The trouble is that T^* have many more nodes than T, in fact may have $\Theta(n^k)$ nodes, rather than O(n).

Although the above approach fails, a refinment of this same idea yields a linear algorithm for T. The idea is that in fact we have to traverse only the original tree T, and the cut sets C_i, are the levels of T. The only necessary change is that we have to use the ranks, and the inorder numbers, with respect to T^*, rather than with respect to T. Those, however, can be calculated in linear time in the size of T. Clearly $r(x,T^*) = 2^{h-depth(x)+1}$ and so the T^*-ranks of the nodes in T can be calculated in O(n) time. A slightly more interesting claim is that we can assign $NUM(x,T^*)$ to nodes in T without traversing T^*, but rather by traversing T only. Procedure COMPUTE_NUM is given in the appendix.

Lemma 2. Procedure COMPUTE_NUM(T,T^*) correctly computes $NUM(x,T^*)$ for any x in T.

Proof. Recall that each node in T has zero or two children. For such trees the set of leaves is dense in their inorder sequence, thus at any point during the run of COMPUTE_NUM exactly one of x and y is a leaf. Going from a leaf x to a non leaf y we add $r(RIGHT(x),T^*)$ which is one plus the number of nodes between x and y in T^*. A similar argument holds when we go from a non leaf x to a leaf y. []

In order to establish the extension of our algorithm to T we have to make certain modifications. First, we have to bear in mind that although we may assume that queries are about leaves of T, those nodes may very well be nonleaves in T^*. The following lemma shows that this is not a real problem.

Lemma 3. Let x and y be two nodes of T which are not ancestors of each other, with x preceding y in inorder. Let $d(x,y,T^*) = NUM(y,T^*) - NUM(x,T^*) + 1$. If $2^i \leq d(x,y,T^*) < 2^{i+1}$ then at least one of those two nodes has an ancestor v in C_i, which is not an ancestor of the other.

Proof. Assume that $r(x,T^*) \leq r(y,T^*)$, that is, x lies on a lower level than y in T^*. We will show that in this case x has an ancestor in C_i which is not an ancestor of y. (The case $r(y,T^*) \leq r(x,T^*)$ is symmetrical.)

First notice that by an argument similar to the one in Lemma 2, x and y do not have a common ancestor in C_i. We now have to show that x has an ancestor in C_i. Notice that since both $T^*(\text{RIGHT}(x))$ and $T^*(\text{LEFT}(y))$ lie strictly between x and y, in the inorder sequence of T^* we have

$$r(x,T^*) \leq r(\text{RIGHT}(x),T^*) + r(\text{LEFT}(y),T^*) < 2^{i+1}$$

and since $r(x,T^*)$ is an integral power of two it must be that $r(x,T^*) \leq 2^i$.

[]

Another fine point which arises when tuning our algorithm to general binary balanced trees, is that the distances in T^* between nodes of T, $d(x,y,T^*)$, are $\Theta(n^k)$. If our model of computation allows us to compute logarithms in O(1) time this causes no problem. What if our repertoire consists of addition, subtraction, multiplication, and division only? Since the problem is off line, we may sort the distances $d(q,T^*)$, in linear time, using a k-pass radix sort, and by traversing the ordered list, remembering increasing powers of 2, we can compute all the required logarithms in O(k n) time. In fact, we can do better than that. By first building a table of exponents of size n, we get $\lfloor \log j \rfloor$, for $2 \leq j \leq n$ in O(1) time by indexing into the table. For numbers $2 \leq d \leq n^k$, we can find the representation of d as a base n number, by performing k divisions by n, and by examining the highest nonzero coefficient we can figure $\lfloor \log d \rfloor$ exactly. This trick is important when dealing with the half-line problem.

For this version of the problem it is convenient to assume that if (x,y) is a query then $r(x,T^*) < r(y,T^*)$, rather than to assume NUM(x) < NUM(y). We can partition T into the ordered C_i-sets in linear time. Likewise if

$$B_i = \{q=(x,y) \mid 2^i \leq d(x,y,T^*) < 2^{i+1}\},$$

then we can partition the queries into the buckets B_i, such that each B_i is ordered by increasing NUM(x), in O(n + q) time. Combining these remarks with Lemma 2 and Lemma 3, we get

Lemma 4. There exists a simple modification of the algorithm in section 3.1 which solves the OFLCA problem for balanced binary trees in O(n + q) time.

3.3 Linear Reduction of General Trees to Union Trees.

In this section we show how to transform the OFLCA problem on general trees to an OFLCA problem on union trees, thereby achieving height which is O(log n). By a union tree we mean a tree which is built as a result of repeated UNION operations, using the well-known UNION_FIND structure, with the weighted union rule, but without executing any FIND operations. Using our previous convention, when we have two nodes, x and y, which are members of two different trees, T_1 and T_2, we use the notation $LCA(x,y,T_i)$ for the lowest common ancestor of x and y in T_i, for i=1,2.

Let T be some general tree, not necessarily a binary tree. As was mentioned before for binary trees, we may assume that all the queries on T are about leaves. We will build a union tree, which will be called U, with the following properties.

a) The nodes of U are the leaves of T. With each node of U we keep some additional information, as will be specified later.

b) Using the information stored in U, we can compute efficiently lowest common ancestors in T from lowest common ancestors in U.

We will now see how to build U, from T, in linear time. The heart of the procedure is a postorder traversal of T. At each node of T we keep a pointer, SET(x), to the root of the union tree containing the set of all the leaves below x which have been visited so far. When visiting a node x of T, we union the sets SET(y), over all the children y of x. The use of the UNION structure for representing the sets is the standard one, and thus we will suppress the details. In addition to the standard weight fields and parent pointers, we keep the following information at each leaf u of T:

a) The set of u's children in U, on a doubly linked list.

b) T_NODE(u) which is a pointer to the node x in T which was visited at the point in time when u became a child of its parent in U.

c) TIME(u) which is an integer designating the point in time, during the traversal of T, when u became a child of its parent in U. If we imagine that children are added in a left-to-right fashion, then TIME(u) is the postorder number of u in U (with the exception of the root r of U for which TIME(r) = 0).

Throughout the rest of this section we will identify the nodes of U with the sets represented by them. We assume that the time variable t is initialized to 0. The procedures BUILD_SET(x) and UNION(u,v,x) are given below. U is the result of calling BUILD_SET on the root of T.

```
procedure BUILD_SET(x);
begin
 if x is a leaf then
  begin
    T_NODE(x) := null;
    TIME(x) := 0;
    SET(x) := {x};
  end else
  begin
    SET(x) := Ø;
    for each child y of x do
    begin
      BUILD_SET(y);
      SET(x) := UNION(SET(x),SET(y),x);
    end;
  end;
end;

procedure UNION(u,v,x);
begin
  comment union the sets represented by the nodes u and v, as roots of the
    respective union trees, while visiting the node x of T.
  perform the weighted union of u and v;
  wlog assume that u became a child of v;
  add u at the end of the children list of v;
  t := t + 1;
  TIME(u) := t;
  T_NODE(u) := x;
  return v;
end;
```

Lemma 5. Let u and v be two nodes of U (leaves of T) and let w be the lowest common ancestor of u and v in U, LCA(u,v,U). We will distinguish between two cases.

a) w is either u or v. Assume without loss of generality that w = u, and let w_1 be the child of w which is an ancestor of v. Then T_NODE(w_1) is the lowest common ancestor of u and v in T, LCA(u,v,T).

b) w ≠ u and w ≠ v. Let w_0 and w_1 be the children of w which are the ancestors of u and v respectively, and assume that TIME(w_0) < TIME(w_1). Then T_NODE(w_1) = LCA(u,v,T).

Proof sketch. Examine the U forest at the point in time t, during the traversal of T, just prior to adding w_1 as a child of w. If at time t we were at the node x of T, then x is the lowest common ancestor in T of any pair (u',v') such that u' is in the subtree U(w) and v' is in U(w_1). Since while adding w_1 as a child of w we set T_NODE(w_1) := x, the claim follows. []

The following notation will be useful. If w is an ancestor of u in U define the tree function ANC(u,w) on U by:

```
ANC(u,w) := if u ≠ w then the ancestor of u in U
                          which is a child of w in U else null;
```

Let L denote the set of leaves of U. To complete the reduction we have to compute (ANC(u,w) , ANC(v,w)) for each pair (u,v) in L, where w = LCA(u,v,U). To do this in linear time we can compute for each w in U the set

$$A(w) = \{(u,v) \mid (u,v) \text{ is in } L \text{ and } w = LCA(u,v,U)\},$$

and A(w) is ordered by increasing postorder numbers, POST_NUM(u). We also keep for each node w in U the list of its children C(w) ordered from left to right. By traversing A(w) and C(w) simultaneously, we can compute ANC(u,w) for every u which is the first coordinate of some query q=(u,v) in A(w), in time $O(|A(w)| + |C(w)|)$. By summing up over all the nodes w in U we get O(n + q). The argument is similar to the one given for the linearity of FIND_CA. Since we can repeat the above argument with the sets A(w) ordered by the second coordinate clearly we can compute (ANC(u,w) , ANC(v,w)) for each query (u,v) in L, where w = LCA(u,v,U), in O(n + q) time. As a conclusion from Lemma 5 and the above argument, we get

Lemma 6. Given a set of q pairs of leaves of T and their lowest common ancestors in U, we can compute the lowest common ancestors in T in O(n + q) time.

3.4 Linear Reduction of Union Trees to Balanced Binary Trees

Let U be a union tree on n nodes. Note that each node in U may have arbitrarily many children; we want to convert it into a binary tree. For the purpose of this discussion, we will color the nodes of U in green. We will add nodes, which we shall refer to as red nodes, in between green nodes, to get a tree which we shall refer to as a binarization of U, as follows. If u is a node of U, with k children in U, we cut off all the children of u, make u a root of a red binary tree with k leaves, and then replace the leaves of that tree by the original green children of u.

Lemma 7. In O(n) time we can compute a binarization B of U, such that the height of B is bounded by three times the height of U.

Proof. We will define the rank r(x,U) to be the number of leaves in the subtree of U which is rooted at x. To binarize U we will classify its nodes according to their rank groups by defining:

$$G_i = \{x \mid 2^{i-1} \leq r(x,U) < 2^i\}$$

Thus every node x belongs to rank group $\lfloor \log r(x,U) \rfloor + 1$ which will be referred to as the index of x in U. We will assume that with each node u of U we keep a linked list L(u) of non-empty buckets B_i. The i^{th} bucket B_i contains the set of children of u which belong to rank group G_i. We further assume that each list L(u) is ordered by increasing order of the indices i. This representation could be achieved in linear time by a two pass radix sort on U, using $< DF\_NUM(PARENT(x)) , \lfloor \log r(x,U) \rfloor + 1 >$ as the sorting key for node x. To binarize U we process its nodes in postorder. While at node x we pair its children in two phases.

During the first phase, which is called BALANCED_PAIRING, we repeatedly eliminate pairs of nodes from the i^{th} bucket by making the two nodes the children of a new red node, which is inserted into the $i+1^{st}$ bucket. At the end of this process we insert the remaining single element of B_i (if any) into an initially empty queue Q. At the end of phase 1 all the elements of the queue belong to different rank groups.

During the second phase, which is called SKEWING, we put the elements of Q as the right children of new red nodes which lie on a left leaning path which is rooted at x. The path is created so that the heavy elements of Q lie closer to the root x. The procedures PAIR, BALANCED_PAIRING, and SKEWING are given in the appendix.

One can easily convince himself that we spend only O(1) time for introducing a new red node, and thus that the time it takes to binarize all of U is O(n).

To see that the height of the resulting binary tree B is logarithmic in n we use a technique which was suggested to us by Scott Huddleston and is similar to techniques of Guibas and Sedgewick [GS78]. We label the edges of B by 0's and 1's as follows: if x is a balanced node then label the edges which correspond to its children by a 0. We distinguish between balanced nodes which are the nodes which were introduced during the BALANCED_PAIRING phase, and skewed nodes which are the nodes which are introduced during the skewing phase. If x is skewed then label its left child edge by a 0 and its right child edge by a 1. Let $X = x_0x_1...x_k$ be some path from the root x_0 of B to some leaf x_k, and let p_i be the label of the edge from x_{i-1} to its child x_i. To bound k we will count separately the number of 0's and 1's in the string $p = p_0p_1...p_k$. Note that if $p_i = 0$ then x_i belongs to a lower rank group than its parent x_{i-1}. Since there are only $\lceil \log n \rceil$ rank groups the number of 0's in p is bounded by log n. Likewise, since blocks of consecutive ones are separated by zeroes there are at most log n blocks of consecutive ones. Finally, note that if $p_i = p_{i+1} = 1$ then x_i must be green (i.e. a node of U) and since there are at most log n nodes on any path from x_0 to a leaf of U the total number of consecutive ones is also bounded by log n. Thus the number of 1's in p is bounded by 2 log n and $k \leq 3 \log n$. []

Since we can keep at each red node a pointer to its lowest green ancestor, we can easily compute lowest common ancestors in U from lowest common ancestors in B. Now to answer q queries on a general tree T, we first transform T into U, and then binarize U into B, by Lemma 7. By Lemma 4 we can answer the queries on B in linear time, and thus we can compute the lowest common ancestors in U in linear time. Finally by Lemma 6 we can compute the lowest common ancestors in T from the lowest common ancestors in U, in linear time and so we have:

Theorem 1. The OFLCA problem on general trees is doable in linear time on a random access machine.

4.0 The Half-Line Problem

By the half-line problem we mean the following. Given a fixed tree T on n vertices we assume that queries, namely pairs (x,y) of vertices of T, are arriving on-line. Whenever a pair of vertices arrives we want to compute their lowest common ancestor. It may require a substantial amount of time to answer a single query if all we are given is the initial tree. If however we are allowed to preprocess the tree we may be able to answer each individual query much faster. We are interested in the time it takes to answer each individual query, which we will refer to as query time, and in the time it takes to preprocess the tree, which we will refer to as preprocessing time. We note that the problem is equivalent to the problem of intermediate difficulty which is presented in [AHU76], in which all the LINK commands preceded all the LCA commands. In that case the size of the tree is the length of the linking sequence (assuming that the sequence results in a single tree) and the time it takes to build the tree is proportional to the number of LINK commands. Thus the preprocessing time corresponds to the time it takes to process the LINK-ing sequence, which was $\Theta(n \log \log n)$ in the above paper. The time for answering a single query in the above paper was $\Theta(\log \log n)$. We will see that the half-line problem helps to explicate the difference in power between pointer manipulating machines and random access machines, as was suggested to us by R. E. Tarjan [T80]. He showed that in fact the above stated query time is the best possible for a pointer manipulating machine. We will show how by using our techniques on a RAM one can greatly improve over the above stated time bounds. In fact there is an algorithm that uses $O(n)$ preprocessing time and has $O(1)$ query time. After reading an earlier draft of this paper, where we claimed an $O(n \log^* n)$ algorithm, Tarjan [T80] encouraged us to obtain a linear algorithm.

In section 4.1 we discuss the half-line problem on complete binary trees. This section includes Tarjan's lower bound for pointer manipulating machines, and the improvement achievable using indexing on a RAM. In section 4.2 we extend the results of section 4.1 to general trees. Section 4.3 includes an improvement over previous results [AHU76] for pointer manipulating machines. The preprocessing time which is $\Theta(n \log \log n)$ in the above paper could be improved to be $O(n)$ without increasing the $O(\log \log n)$ query time, which is optimal by Tarjan's theorem.

4.1 The Half-Line Problem on Balanced Binary Trees

We will start this section by proving a lower bound on the query time on a pointer manipulating machine. Following Tarjan [T77,T80] we make the following assumptions about the model of computation. We assume that the tree is represented by a linked structure, with each tree vertex represented by a single node (record). The structure may contain additional nodes not representing any tree vertex. With each node we keep a fixed number of pointers, independent of the tree size. We may take this number to be 2.

Theorem 2 [T80]. Let T be a complete binary tree with n leaves. Any pointer machine requires $\Omega(\log \log n)$ time to answer an LCA query in the worst case, independent of the representation of the tree.

Proof. The key point is that from any node in our structure at most $2^{j+1}-1$ nodes are accessible in j steps or less. Let k be such that all LCA queries can be answered in k steps or less. For each leaf x of T let A_x denote the set of nodes representing tree vertices which are accessible from x in k steps or less. Let w be a node of T and u and v be its left and right children. We claim that either w belongs to A_x for every leaf x in T(u), or w belongs to A_y for every leaf y of T(v). Otherwise there would be a pair of leaves (x,y), x in T(u), and y in T(v), such that w = LCA(x,y) but w is not accessible from either x or y in less than k+1 steps, contradicting the choice of k. We conclude that w occurs in at least half the sets A_x of its descendants x. If w belongs to the i^{th} level for $0 < i \leq h = \log n$ then w has 2^i leaf descendants, and thus w occurs in at least 2^{i-1} A_x sets. Since there are 2^{h-i} nodes at level i we see that the i^{th} level contributes

$$2^{h-i}\, 2^{i-1} = 2^{h-1} = n/2$$

occurrences to the union of the sets A_x. There are log n levels, and so if L represents the set of leaves of T we have:

$$\sum_{x \in L} |A_x| \geq n/2 \log n$$

Since for any leaf x we have $|A_x| \leq 2^{k+1}$ we get:

$$n\, 2^{k+1} \geq n/2 \log n$$

which implies that

$$k \geq \log \log n - 2$$

[]

An interesting fact is that a complete tree, which is used in the proof of the lower bound for pointer manipulating machines, is perhaps the "easiest" case for a random access machine. This result is more precisely stated by the following lemma. This lemma is just a special case of the much more general result which will be proved later (Theorem 3). The reason for stating this result separately is that its proof is so simple, and yet the main idea is perhaps helpful for understanding the proof of the general case.

Lemma 8. Let T be a complete binary tree with n leaves. There exists an algorithm which achieves O(n) preprocessing time, and O(1) query time.

Proof. As we have seen before (observations 1 and 2 of section 3.1) in order to answer an LCA query in O(1) time it is sufficient to answer in O(1) time queries of the following form:

> given a leaf x of T, and an integer d, find an ancestor y of x with $r(y) = 2^i$, where $i = \lfloor \log d \rfloor$.

If we keep the nodes of T of rank 2^i, ordered from left to right in an array of length $n / 2^i$ we could index directly into the appropriate ancestor of x. All we need is the position p of x in the left-to-right order of the leaves of T, and we could find the required y by using $\lceil p / 2^i \rceil$ as an index. Note that we do not assume that the log of d is computable in O(1) time. We use a table lookup to compute logarithms, and clearly setting up the table takes only linear time. One can easily verify that we could accomplish all the preprocessing we need in linear time. []

4.2 The Half-Line Problem on General Trees

We will now tackle the general case. Let T be some tree of size n, U be the corresponding union tree, and B be the binarization of U. It may be convenient to think of T, U, and B as three separate trees, although in our notation we will assume that each leaf of T is a node of both U and B. From our exposition in section 3 it follows that in order to answer LCA queries on T in O(1) time it is sufficient to be able to answer in O(1) time queries of the following two types:

Type 0 query:

> Given a leaf x_0 of B, and an integer m_0, $2^i \leq m_0 < 2^{i+1}$, find an ancestor y_0 of x with $r(z_0, B^*) = 2^i$.

Type 1 query:

Given a leaf x_1 of U, and an ancestor z_1 of x_1 in U, find the child y_1 of z_1 which is an ancestor of x_1 in U.

The reason queries of type 0 are of interest is that given two leaves u and v of T, from the answers to certain queries of type 0 which are computable in O(1) time we can find w' = LCA(u,v,B) in O(1) time (section 3.2). From w' we can compute w = LCA(u,v,U) in O(1) time (section 3.4). Queries of type 1 are of interest because, roughly speaking, they allow us to compute the children of w which subtend u and v. Formally, we can compute the pair (ANC(u,w) , ANC(v,w)) in O(1) time, and from this pair LCA(u,v,T) is computable in constant time (section 3.3).

Some definitions concerning balance in trees will be useful. Given a general tree T, and a constant c, a node x of T will be called Locally Balanced of degree c, LB(c), if the height of the subtree of T rooted at x is bounded by c times log |T(x)|, i.e., $h(T(x)) \leq c \log |T(x)|$. We call a tree T Globally Balanced of degree c, GB(c) if every node x of T is LB(c). The class of GB trees is very wide and it includes all the types of balanced trees which are commonly used for storage of information in computer science, such as B-trees, AVL-trees, and BB(α) trees. In particular note that U is GB(1) and its binarization B is GB(3). Note that the rank r(x) of a node x is a good measure of the size of the subtree T(x) which is rooted at x, i.e., that $r(x) = \Theta(|T(x)|$, for all the balanced trees used in this paper. Thus we may replace |T(x)| by r(x) which is convenient because of the additive property of ranks; if C is the set of children of x then:

$$r(x) = \sum_{y \in C} r(y)$$

Let V be some general (not necessarily binary) GB(c) tree. We define a basic query on V to be a query of the following form:

Basic query:

Given a leaf x of V, and an integer d, find the ancestor y of x in V with DEPTH(y) = d.

We will now show that in fact both type 0 and type 1 queries are reducible to basic queries.

Lemma 9. If we could answer basic queries on a GB tree in O(1) time, then we could answer type 0 and type 1 queries in O(1) time.

Proof. Let x_0, m_0, and y_0 be as in the definition of a type 0 query; let x_1, z_1, and y_1 be as in the definition of type 1 query; and finally let x, d, and y be as in the definition of a basic query.

To answer the type 0 query (x_0,m_0), take V = B, $x = x_0$ and $d = h - i$, where h is the height of B, and $i = \lfloor \log m_0 \rfloor$. Return $y_0 = y$ where y is the answer to the corresponding basic query.

To answer the type 1 query (x_1,z_1), take V = U, $x = x_1$, and $d = \text{DEPTH}(z_1) + 1$. Let y be the answer to the basic query thus obtained and return $y_1 = y$. []

For the rest of this section let V denote some general GB(c) tree on n vertices. We will show how to preprocess the tree V in O(n) time such that the answer to each basic query is computable in O(1) time. Roughly speaking the idea is to "peel off" some layers of fringe trees from V such that the remaining "core tree" is small enough to index into it from its leaves. For the fringe trees we will build a complete look-up table for each type of tree which could appear as a fringe tree. By a suitable numbering of the fringe trees we can first index into the appropriate table to find the postorder number of the required node in its fringe tree. We can then index into the required node in the given fringe tree, all in O(1) time. The fringe trees will be small enough so that the total space used for the tables in still O(n).

We now give a more formal description of our method. Given some tree T, a set C of its vertices will be called a tree cut for T if every path P from the root of T to a leaf intersects C exactly once (i.e. $|P \cap C| = 1$). We pick the numbers a_0, a_1, a_2, and a_3 as follows:

$$a_0 = 1; \quad a_1 = \log \log n; \quad a_2 = \log n; \quad a_3 = n$$

For $i < 3$ we define the i^{th} ply boundary in V by:

$$B_i = \{x \mid r(x) \leq a_i < r(\text{PARENT}(x))\}$$

The ply boundaries are similar to the ones used in [HL79] for BB(α) trees. One can easily verify that B_i is a tree cut for V, for i = 0, 1, or 2. Notice that

for general trees the rank of a node is defined to be the number of leaves below it and thus B_0 is in fact the set of leaves of V. It will be convenient to add the set consisting only of the root r of V as a fourth boundary B_3. For $i \in \{0,1,2\}$ we define the i^{th} ply P_i by:

$$P_i = \{x \mid x \text{ has a descendant in } B_i \text{ and an ancestor in } B_{i+1}\}$$

Note that $P_i \cap P_{i+1} = B_{i+1}$. For i=1,2 we would like to index from boundary nodes in B_i into their ancestors in P_i. Since the width of P_i is $\Theta(\log a_{i+1}) = \Theta(a_i)$ we would have to associate an array of size $\Theta(a_i)$ with each boundary node. On the other hand, $|B_i|$ is not necessarily $O(n/a_i)$, and so the space complexity of such a scheme may be excessive. We encounter a similar problem if we try to associate arrays with parents of nodes in B_i only. The following definition helps to overcome this problem. We call a node y <u>principal node of order i</u> or simply principal if y is in P_i and all its children are in B_i.

<u>Lemma 10</u>. If x is any node in $P_i - B_i$ then x has a principal descendant in P_i.

<u>Proof</u>. By our assumption $r(x) > a_i$. If all the children of x have rank $\leq a_i$ then x is principal. Otherwise x has at least one child y with $r(y) > a_i$. Since y is lower than x the proof follows by a simple induction on the height of x above B_i. []

As we will see later associating arrays only with principal nodes of orders 1 or 2 is sufficient for the purpose of indexing into nodes in $P_1 \cap P_2$. At the same time this scheme requires only a linear amount of space.

Let x be a principal node of order i, (i=1 or 2), and let y be its ancestor in B_{i+1}. With each such x we associate an array $A_x[0 : c \log a_{i+1}]$ which contains pointers to the ancestors of x in P_i as follows; if $0 \leq d \leq c \log a_{i+1}$ then:

$A_x[d]$ = if x has an ancestor $z \in P_i$ with DEPTH(z,V(y)) = d
then z else null;

Notice that $c \log a_{i+1}$ is a sufficient length since by our assumption V is a GB(c) tree. With each leaf of V we keep pointers to all its boundary ancestors. With each boundary node x in B_i we keep the following fields:

1) DEPTH(x) = the depth of x in V;

2) CPN(x,i) = if PARENT(x) $\in P_i$ then the Closest Principal Node of order i below PARENT(x) else null;

Notice that the P_0 could be represented as the disjoint union:

$$P_0 = \bigcup_{y \in B_1} V(y)$$

The trees V(y) where $y \in B_1$ are the fringe trees. The core tree is what is left after pruning off the fringe trees. To answer the basic query (x,d) on V we first figure out in which ply we have to look for the answer. We notice that the fringe trees are so small that in fact we could enumerate all the possible types of those trees and build a complete lookup table for each type in O(n) total time, while preprocessing the tree. This use of precomputation is reminiscent of the four Russians algorithm for boolean matrix multiplication [ADKF70,AHU74] and was suggested to me by George Lueker. We now give the details of the construction.

We first give a numbering system for general trees which will be used for numbering the fringe trees. Let T be some rooted unlabeled tree of size k. The degree of a vertex x in T, deg(x) is defined to be the number of its children. Let $x_1x_2\ldots x_k$ be the postorder sequence of the vertices of T. We define the signature s of T to be the sequence $s = s_1s_2\ldots s_k$ where $s_i = \deg(x_i)$.

Lemma 11. T is reconstructible from its signature in linear time.

The proof is easy and is omitted. Notice that although the length of a signature is $\Theta(k \log k)$ it could easily be compressed to O(k), for example by using 0's as separators, and blocks of 1's of length d to represent the degree d. Since

$$\sum_{x \in T} \deg(x) = k - 1$$

this representation requires O(k) bits. Lemma 11 shows that two trees with the same signatures are isomorphic. We shall refer to the signature of a tree also as the type of the tree. The previous argument shows that the number of trees of size k is $O(2^k)$, and thus the number of trees of size at most k is $O(2^k)$. If we take $k = \log \log n$ we see that the number of trees which can appear as fringe trees is o(n). For each type s tree, of size at most k, we build a table $C_s[1{:}k\ ,\ 0{:}k]$ of size $O(k^2)$. If we identify node x with its postorder number POST_NUM, then

$C_s[i,d]$ = if i has an ancestor j in T of depth d then j else 0;

A possible implementation of those tables will use a three dimensional array $C[1{:}k\ 2^k, 1{:}k, 0{:}k]$. With each $x \in B_1$ we keep:

a) A field SIGNATURE(x) which contains the signature of T(x).

b) An array $B_x[1 : |T(x)|]$ in which the i^{th} entry, $B_x[i]$ contains a pointer to the i^{th} node of T(x) in postorder.

Assuming that we have all the above fields and tables the following procedure will answer a basic query on V in O(1) time.

```
procedure BASIC_ANSWER(x,d);
begin comment assuming x is a leaf of V and d is an integer 0 ≤ d ≤ DEPTH(x)
  return the ancestor y of x at depth d;
  for j := 0 to 3 do x_j := the ancestor of x in B_j;
  k := the smallest i s.t. DEPTH(x_i) ≤ d ≤ DEPTH(x_{i+1});
  v := x_k;  u := x_{k+1};
  d' := d - DEPTH(u);
  comment the required y is in ply P_k between the boundary nodes u and v, and
    its depth in V(u) is d';
  if k = 0 then
  FRINGE_SEARCH:
  begin
      i := POST_NUM(x);
      s := SIGNATURE(u);
      j := C_s[i,d'];
      y := B_u[j];
  end else
  if d = DEPTH(v) then y := v else
  CORE_SEARCH:
  begin
      z := CPN(v,k);
      y := A_z[d'];
  end;
  return y;
end;
```

Lemma 12. BASIC_ANSWER(x,d) computes the ancestor of x in V at depth d in O(1) time.

Proof. The correctness of the FRINGE_SEARCH block follows from the definitions of signatures and of the tables C_s and B_u for signatures s and for $u \in B_1$. The correctness of the block CORE_SEARCH follows from the definitions of principal nodes, of the CPN fields, and of the arrays A_z for principal nodes z of order 1 or 2. Notice that once we have entered the CORE_SEARCH block we know that $d < DEPTH(v)$ where $v = x_k$ is in B_k. We can deduce that the required y is an ancestor of PARENT(v) in P_k. Furthermore, since $z = CPN(v)$ is a principal node which is a descendant of PARENT(v) it must be that y is an ancestor of z in P_k which lies d' levels down from $u = x_{k+1}$. Thus by the definition of the array A_z we know that $A_z[d'] = y$. []

Lemma 13. Our algorithm requires O(n) space and preprocessing time.

Proof. By close inspection of our structure one can easily verify that the preprocessing time is linear in the amount of space used. We therefore have to show that the tables A_x, B_y, and C_s require a total of O(n) space. We will represent the total space used for the tables by:

$$S = S_A + S_B + S_C$$

where S_A, S_B, and S_C represent the space used up by the corresponding arrays. Let PR_i denote the set of principal nodes of order i.

To bound S_A notice that we associated arrays A_x with principal nodes of order i, for i = 1 or 2. From the definition it follows that two principal nodes of order i subtend disjoint subtrees. Since the rank of a principal node of order i is at least a_i we have:

$$|PR_i| \leq n / a_i \qquad (1)$$

For i = 1 or 2 the array A_y associated with each principal node y of order i is bounded by:

$$|A_y[0 : c \log a_{i+1}]| = 1 + c \log a_{i+1} = O(a_i) \qquad (2)$$

We thus get:

$$S_A = \sum_{y \in PR_1} |A_y| + \sum_{y \in PR_2} |A_y|$$

Lemma 12. BASIC_ANSWER(x,d) computes the ancestor of x in V at depth d in O(1) time.

Proof. The correctness of the FRINGE_SEARCH block follows from the definitions of signatures and of the tables C_s and B_u for signatures s and for $u \in B_1$. The correctness of the block CORE_SEARCH follows from the definitions of principal nodes, of the CPN fields, and of the arrays A_z for principal nodes z of order 1 or 2. Notice that once we have entered the CORE_SEARCH block we know that $d < DEPTH(v)$ where $v = x_k$ is in B_k. We can deduce that the required y is an ancestor of PARENT(v) in P_k. Furthermore, since $z = CPN(v)$ is a principal node which is a descendant of PARENT(v) it must be that y is an ancestor of z in P_k which lies d' levels down from $u = x_{k+1}$. Thus by the definition of the array A_z we know that $A_z[d'] = y$. []

Lemma 13. Our algorithm requires O(n) space and preprocessing time.

Proof. By close inspection of our structure one can easily verify that the preprocessing time is linear in the amount of space used. We therefore have to show that the tables A_x, B_y, and C_s require a total of O(n) space. We will represent the total space used for the tables by:

$$S = S_A + S_B + S_C$$

where S_A, S_B, and S_C represent the space used up by the corresponding arrays. Let PR_i denote the set of principal nodes of order i.

To bound S_A notice that we associated arrays A_x with principal nodes of order i, for i = 1 or 2. From the definition it follows that two principal nodes of order i subtend disjoint subtrees. Since the rank of a principal node of order i is at least a_i we have:

$$|PR_i| \leq n / a_i \qquad (1)$$

For i = 1 or 2 the array A_y associated with each principal node y of order i is bounded by:

$$|A_y[0 : c \log a_{i+1}]| = 1 + c \log a_{i+1} = O(a_i) \qquad (2)$$

We thus get:

$$S_A = \sum_{y \in PR_1} |A_y| + \sum_{y \in PR_2} |A_y|$$

Lemma 12. BASIC_ANSWER(x,d) computes the ancestor of x in V at depth d in O(1) time.

Proof. The correctness of the FRINGE_SEARCH block follows from the definitions of signatures and of the tables C_s and B_u for signatures s and for $u \in B_1$. The correctness of the block CORE_SEARCH follows from the definitions of principal nodes, of the CPN fields, and of the arrays A_z for principal nodes z of order 1 or 2. Notice that once we have entered the CORE_SEARCH block we know that $d < DEPTH(v)$ where $v = x_k$ is in B_k. We can deduce that the required y is an ancestor of PARENT(v) in P_k. Furthermore, since $z = CPN(v)$ is a principal node which is a descendant of PARENT(v) it must be that y is an ancestor of z in P_k which lies d' levels down from $u = x_{k+1}$. Thus by the definition of the array A_z we know that $A_z[d'] = y$. []

Lemma 13. Our algorithm requires O(n) space and preprocessing time.

Proof. By close inspection of our structure one can easily verify that the preprocessing time is linear in the amount of space used. We therefore have to show that the tables A_x, B_y, and C_s require a total of O(n) space. We will represent the total space used for the tables by:

$$S = S_A + S_B + S_C$$

where S_A, S_B, and S_C represent the space used up by the corresponding arrays. Let PR_i denote the set of principal nodes of order i.

To bound S_A notice that we associated arrays A_x with principal nodes of order i, for i = 1 or 2. From the definition it follows that two principal nodes of order i subtend disjoint subtrees. Since the rank of a principal node of order i is at least a_i we have:

$$|PR_i| \leq n / a_i \qquad (1)$$

For i = 1 or 2 the array A_y associated with each principal node y of order i is bounded by:

$$|A_y[0 : c \log a_{i+1}]| = 1 + c \log a_{i+1} = O(a_i) \qquad (2)$$

We thus get:

$$S_A = \sum_{y \in PR_1} |A_y| + \sum_{y \in PR_2} |A_y|$$

and from (1) and (2) we have:

$$= O(n / a_1)\, O(a_1) + O(n / a_2)\, O(a_2) = O(n)$$

To bound S_B notice that the space used up by each fringe tree $T(x)$ for $x \in B_0$ is $O(r(x))$ and so clearly S_B is $O(n)$.

To bound S_C we note that there are at most $O(2^{\log \log n})$ types of fringe trees. Each one requires a table of size $O((\log \log n)^2)$. Thus the total space which is used up is:

$$O(\log n\, (\log \log n)^2) = o(n)$$ []

The main result of this section is summarized in the following theorem which is a consequence of Lemmas 12 and 13.

Theorem 3. Our algorithm requires $O(n)$ preprocessing time and answers each individual query in $O(1)$ time.

This last result is an extention of Theorem 1 to the half-line case. It is an improvement over an $O(n \log^* n)$ preprocessing time, $O(\log^* n)$ query time in an earlier version of this paper. Tarjan [T80] suggests an alternative proof, possibly by using the table compression technique of Tarjan and Yao [TY79].

4.3 More on Pointer Manipulating Machines

In 4.1 we showed that pointer machines require $\Omega(\log \log n)$ time per LCA query in the worst case (Theorem 2). Aho, Hopcroft, and Ullman [AHU76] gave an $O(n \log \log n)$ preprocessing time and $O(\log \log n)$ query time algorithm which is implementable on a pointer machine. It may be of interest to ask whether $\Omega(n \log \log n)$ is a lower bound on the preprocessing time by any pointer machine algorithm for the half-line problem. We now show that this is not the case by observing that it is possible to tailor our algorithm to a pointer machine to achieve optimal query time with linear preprocessing time.

Theorem 4. There is a pointer machine algorithm which achieves O(n) preprocessing time with O(log log n) query time.

Proof Sketch. We first observe that a pointer machine can partition a GB tree into the plies described in 4.2 in linear time. For principal nodes y of degree 2 we can replace the arrays A_y by balanced trees, and so instead of indexing in O(1) time we use a log log n search in a balanced tree. The B_1 boundary as well as the tables for the fringe trees can be eliminated since in a GB subtree of size O(log n) a brute force upwards search takes O(log log n) time to answer basic queries. We notice also that computing log d for $d \leq n$ can be done by an O(log log n) search in a preset balanced tree. Those arguments show that given the union tree U, and its binarization B, we can answer LCA queries on the original tree T in O(log log n) time after preprocessing U and B in linear time. U is built in linear time by our algorithm, which requires no indexing. Also, if the the children of each node in U are given in order of increasing ranks then our algorithm computes the binarization B in linear time without use of indexing.

It remains to show that we can order children in U in order of increasing ranks in linear time on a pointer machine. The key point is to observe that the distribution of ranks in the U tree is skewed enough to enable such a linear sort. We use a variant of the finger sort of [GMPR77] to sort all the vertices in U in order of increasing ranks. We first build a complete tree T_0 of size $m = 2^j$, where $j = \lceil \log n \rceil$, with leaves numbered from 1 to n. The leaves of T_0 represent buckets. We keep a finger at the leftmost leaf of the tree which is numbered by one. Given a node x of U with rank r, we insert it into the r^{th} bucket by starting a finger search for this bucket from the finger. The search takes O(log r) time for a node of rank r. We notice that if $x \in U$ then $r(x) \leq w(x) = |U(x)|$. For a union tree U with n vertices and e edges one can easily prove by induction on the structure that

$$\sum_{x \in U} \log w(x) \leq e$$

and thus the total sorting time is O(n + e) = O(n) □

An interesting problem which is still open is whether the off-line problem is doable in linear time on a pointer machine.

Acknowledgements

I would like to thank my advisor George Lueker for his encouragement, many enjoyable discussions, and helpful suggestions. In particular his suggestion greatly simplified the proof of Theorem 3. I also wish to thank Scott Huddleston for his suggestion which simplified the proof of Lemma 7. Finally, I am grateful to Robert Tarjan for his contributions and helpful suggestions concerning the half-line case. In particular Theorem 2 is due to him, and his suggestions encouraged me to improve over a weaker version of Theorem 3.

References

[ADKF70] Arlazarov, V. L., Dinic, E. A., Kronrod, M. A., and Faradzev, I. A., "On Economical Constraction of the Transitive Closure of a Directed Graph," Dokl. Akad. Nauk SSSR 194 (1970), pp. 487-488. English translation in Soviet Math. Dokl. 11:5 pp. 1209-1210.

[AHU74] Aho, A., Hopcroft, J., and Ullman, J., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[AHU76] Aho, A., Hopcroft, J., and Ullman, J., "On Finding Lowest Common Ancestors in Trees," SIAM J. Comput. 5:1 (March 1976), pp. 115-132.

[AVL62] Adel'son-Vel'skii, G. M., and Landis, Y. M., "An Algorithm for the Organization of Information," Dokl. Akad. Nauk SSSR 146 (1962), pp. 263-266. English translation in Soviet Math. Dokl. 3, 1962 pp. 1259-1262.

[GMPR77] Guibas, L. J., McCreight, E. M., Plass, M. F., and Roberts, J. R., "A New Representation for Linear Lists," Proc. Ninth Annual ACM Symposium on Theory of Computing (May 1977), pp. 49-60.

[GS78] Guibas, L. J., and Sedgewick, R., "A Dichromatic Framework for Balanced Trees," 19-th FOCS (October 1978), pp. 8-21.

[HL79] Harel, D., and Lueker, G., "A Data Structure with Movable Fingers and Deletions," Technical Report #145, University of California at Irvine, December 1979.

[K68] Knuth, D. E., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

[K73] Knuth, D. E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[L79] Lueker, G. S., "A Transformation for Adding Range Restriction Capability to Dynamic Data Structures for Decomposable Searching Problems," Technical Report #129, Dept. of ICS, University of California at Irvine, February 1979.

[M79] Maier, D., "An Efficient Method for Storing Ancestor Information in Trees," SIAM J. Comput. 8:4 (1979), pp. 599-618.

[NR73] Nievergelt, J., and Reingold E. M., "Binary Search Trees of Bounded Balance," SIAM J. Comput. 2:1 (1973), pp. 33-43.

[PT60] Perlis, A. J., and Thornton, C., "Symbol Manipulation by Threaded Lists," CACM 3:4 (April 1960), pp. 195-204.

[S80] Standish, T. A., Data Structure Techniques, Addison-Wesley, Reading, Mass., 1980.

[T75] Tarjan, R. E., "Efficiency of a Good But Not Linear Set Union Algorithm," JACM 22:2 (April 1975), pp. 215-225.

[T77] Tarjan, R. E., "Reference Machines Require Non-Linear Time to Maintain Disjoint Sets," 9-th STOC (1977), pp. 18-29.

[T80] Tarjan, R. E., Private comunication, June 4, 1980.

[TY79] Tarjan, R. E., and Yao, A. C., "Storing a Sparse Table," CACM 22:11 (November 1979), pp. 606-611.

Appendix: Pidgin Algol for the Algorithms.

We use the following notation [S80]. Let QUEUE be a queue and x be an element.

QUEUE <= x means: add x at the rear of QUEUE.

x <= QUEUE means: delete the element at the front of QUEUE, and assign its value to x.

```
procedure PARTITION(T,L);
begin comment assuming every query in Q is of the form q=(x,y) where x and y are
  leaves, and NUM(x) < NUM(y), partition T into h levels C_i, and Q into h
  buckets B_i;
  thread the tree;
  for each node x of T store the inorder number of x in NUM(x), and initialize
    L(x) to the empty list;
  for each query q=(x,y) calculate d(q)=d(x,y)=NUM(y) - NUM(x) + 1;
  while Q is not empty do delete q=(x,y) from Q and add it to L(x);
  comment at this point, for every node x, L(x) consists of all the queries of
    the form (x,y) for some y;
  x := the first node of T in inorder;
  while x ≠ do
  begin
    i := log r(x);
    add x at the end of C_i;
    while L(x) is not empty do
    begin
      let q=(x,y) be the first query in L(x);
      j := ⌊ log d(x,y) ⌋ ;
      delete q from L(x) and add it at the end of B_j;
    end;
    x := NEXT(x);
  end;
  comment at this point C_i is the i^th level, ordered by inorder. Also,
    B_i = {queries(x,y) | 2^i ≤ d(x,y) < 2^(i+1)}, and B_i is ordered by increasing
    NUM(x);
end;

procedure FIND_LCA1(x,y);
begin
  v := C_ANCESTOR(x,i);
  v := LRA(v);
  if v subtends y then return v else return LRA(v);
end;
```

```
procedure FIND_CA(T,C_i,B_i)
begin assuming the tree is partitioned into log n cuts and the queries are
  partitioned into log n buckets, by PARTITION, such that NEXT_C(v) (resp.
  NEXT_B(q)) is the next element in the appropriate cut (resp. bucket) compute
  for each q=(x,y) in B_i the ancestor of x in C_i, and store it in
  C_ANCESTOR(x,i);
  for i := 1 until h do
  begin
    v := the first node in C_i;
    q := the first query in B_i;
    if q ≠ null then
    begin
      while v ≠ null and v is not an ancestor of x do v := NEXT_C(v);
      while q ≠ null and v is an ancestor of x do
      begin
        (x,y) := q;
        C_ANCESTOR(x,i) := v;
        q := NEXT_B(q);
      end;
    end;
  end;
end;

procedure COMPUTE_NUM(T,T*)
begin
  x := the first node of T in inorder;
  NUM(x,T*) := r(LEFT(x),T*);
  y := NEXT(x,T);
  while y ≠ null do
  begin
    NUM(y,T*) := if x is a leaf of T then NUM(x,T*) + r(RIGHT(x),T*)
                                     else NUM(x,T*) + r(LEFT(y),T*);
    x := y;
    y := NEXT(x,T);
  end;
end;
```

```
procedure BALANCED_PAIRING(x);
begin
  initialize QUEUE to the empty queue;
  while L(x) is not empty do
  begin
    B_i := the next bucket in L(x);
    delete B_i from L(x);
    while |B_i| ≥ 2 do
    begin
      delete two nodes u and v from B_i;
      PAIR(u,v,w);
      add w to B_{i+1};
    end;
    if B_i ≠ Ø then QUEUE <= the only element of B_i;
  end;
end;

procedure SKEWING(x);
begin
  y_0 <= QUEUE;  i := 0;
  while QUEUE is not empty do
  begin
    x_i <= QUEUE;
    PAIR(y_i,x_i,y_{i+1});
    i := i+1;
  end;
  replace y_i by x;
end;

procedure PAIR(u,v,w);
begin
  create a new red node w;
  make u the left child of w;
  make v the right child of w;
end;
```