THE RECONSTRUCTION AND OPTIMIZATION OF TRIE HASHING FUNCTIONS

Leen Torenvliet and P. van Emde Boas

University of Amsterdam, Depts of Mathematics and Computer Science
Roetersstraat 15, 1018 WB Amsterdam, the Netherlands.

Abstract: We propose an adaptation to the trie hashing algorithm published by W. Litwin in 1980. This adaptation extends the algorithm so that it will save necessary information on secondary storage to reconstruct the hashing function after loss of information (E.G., system crash or termination of a find/insert program). An algorithm is given to reconstruct the trie from the information saved, and another for optimizing the reconstructed trie. A mathematical analysis of the trie data structure is given, making visible the essential structural properties of these tries; based on this analysis correctness of the algorithms presented can be established.

0. INTRODUCTION.

In 1980 Litwin [4] proposed a new algorithm for hashing. Contrary to usual hashing this algorithm stores records in order. Furthermore the file may be highly dynamic, even entirely resulting from a sequence of insertions. The load factor stays typically about 70% . The search for a record with a given key is performed in only one disk access. Litwin claims: "No other technique attaining such performance is known." The algorithm dynamically creates a hashing function which might be represented by a kind of trie [3]. During insertions and searches this trie is entirely kept in core. Hence after a system crash or termination of the program that constructs the trie, the entire construction giving access to the file is lost.

As files may attain millions of records, it seems reasonable to think that the information in the file may not only be of use to the running program but may be used by subsequent programs, which are updating this information. Even if there is only one program which builds this information and runs forever, a system crash might spoil the created hashing function and we would be forced to start all over again from scratch. Each program that wants to update the information in the file would have to reconstruct the trie giving access to this file. There are essentially two ways in which we can enable programs to perform this reconstruction:

First the original construction method can be used by reinserting all records on the disk. This is a rather slow and tedious job which usually will require many disk accesses. Alternatively we can, during construction of the original trie, take steps to save on secondary storage information necessary for reconstruction of the trie. Assuming that the original trie may be kept in core entirely, regaining this information will cost only one disk access.

Our paper investigates this second approach. In addition to the reconstruction, we would like the new trie to be as efficient as possible with respect to search time in the trie.

The paper is organized as follows: Section 1 gives an outline of Litwin's original paper. Section 2 presents a mathematical formalization of the underlying information structure which enables the trie to perform its role. The main invariant of the trie date structure is extracted, and the information necessary for reconstruction is obtained. The correspondence between the tries described by Litwin and their mathematical formalizations are the subject of section 3 . Based upon this correspondence the reconstruction and optimization algorithms are obtained and seem to be correct. Due to lack of space known performance analyses are omitted. Section 4 presents some test results. Finally, in section 5 concluding remarks and indications of subjects for further research are provided.

1. LITWIN'S TRIE HASHING ALGORITHM.

We start this section with a list of definitions of basic concepts which are used in the sequel.

1.1 Definitions and notations.

1.1.1 *Digits* - given a finite, totally ordered alphabet Σ the elements of Σ are called *digits*; Σ contains a minimal element denoted '_' *(space)* and a maximal element denoted ':'.

1.1.2 *Strings* - the elements of Σ^* will be called *strings*.

1.1.3 *Length* - the *length* of the string $s = s_0 s_1 \ldots s_k$, denoted $l(s)$ equals k. The empty string has length -1.

1.1.4 *Key space* - the subset $K \subset \Sigma^*$ will be called a *key space* provided K consists of all strings over Σ of length n for some natural number n. This number n is called the *length* of K, and members of K are denoted $k = k_0 k_1 \ldots k_n$. The length of K is denoted $l(K)$. By convention strings s with $l(s) < n$ are considered as members of K by extending them with spaces: $s = s_0 \ldots s_{l(s)}$ becomes $k = s_0 \ldots s_{l(s)} '\_' \ldots '\_'$.

1.1.5 *Lexicographical order* - given the ordering $<$ on Σ^* the *Lexicographical order*, denoted $<_\Sigma$ on Σ^* is defined by:
$\forall_{s,s' \in \Sigma^*}[s <_\Sigma s' \Leftrightarrow \exists_j[\forall_{i<j}[s_i = s'_i] \;\&\; s_j < s'_j]]$.
This induces a total order on a key space K.

1.1.6 *Initial segment* - an element $s = s_0 \ldots s_k$ is called an *initial segment* of $s' = s'_0 \ldots s'_m$ iff $k \leq m$ and $s_i = s'_i$ for $0 \leq i \leq k$. If $k < m$ then s is called a *proper initial segment* of s', notation $s \subset s'$.

1.1.7 *Buckets* - records will be stored in order of their keys on secondary storage called *disk*. A disk is divided into cells called *buckets*. Buckets are numbered $0,1,2,\ldots$. The number of a bucket is called its *bucket address*. Each bucket may contain the same number of records called *bucket capacity*, denoted b.

1.1.8 *Search* - a search for a record consists of two steps:

i) an address computation by an algorithm called *key-to-address-transform (ktat)*

ii) a *disk access* bringing to the main memory called *core* a bucket containing at most b records for examination.

1.1.9 *Collision* - a *collision* occurs when a record is inserted into a bucket which is already full since it contains b records. This bucket will be divided in two parts. The key whose position in the ordered sequence of $b+1$ keys is closest to $(b+1)/2$ will be called the *middle key*, notation c'.

1.1.10 *Nodes* - a *node* is a structured value with the following fields:

i) two pointers (UP,LP) called *upper-* and *lower pointer* respectively

ii) a pair called *digit field* (DF) consisting of a *digit number* (DN) in $\mathbb{N}$ and a *digit value* (DV) in Σ.

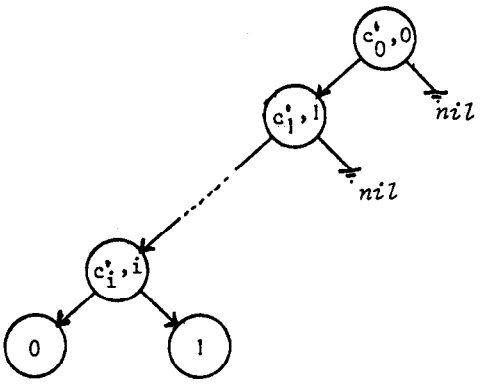
Pointers may be either a reference to a node or bucket address, or a special value *nil* indicating that the pointer refers to nothing.

1.1.11 *Trie* - a *trie* is the dynamic data structure composed of nodes, which is constructed according to the algorithms given in 1.2.

1.1.12 *Weight* - the *weight* of a trie T, denoted $w(T)$ equals the number of nodes in the trie.

A definition like 1.1.11 may look strange to the mathematically trained reader, but actually, given the way Litwin presents his structure, it is the only one possible (short of performing the analysis presented in sections 2 and 3 of our paper). In reality a trie will be a structure based upon a binary tree. As a consequence a large amount of standard terminology on binary

trees will be used in the sequel. In particular we use phrased like *father, son, left son, right son, ancestor, descendant, left (right) subtrie,* etcetera. We can freely speak about the weight of subtries. In particular, for a trie T we denote by $w_L(T)$ and $w_R(T)$ the weight of its left and right subtrie. The concrete tries described in this section will be called *Litwin tries* in the sequel to discriminate them from the formal tries introduced in section 2.



1.2 Construction of the trie.

Litwin described his data structure by providing a verbose description how the structure is created by a sequence of insertions. We rather strictly follow his description.

1.2.1 Initial stage.

As long as no keys are inserted into the file, we allocate a single bucket 0 and assume that the entire file is hashed onto address 0. We thus provide storage for up to b records. The trie initially is empty, and the pointer to its root points directly to address 0.

1.2.2 First collision.

When the b+1-st record is inserted into the file the first collision occurs. At this stage the trie becomes non empty. We proceed as follows:

1.2.2.1) Select the shortest sequence of digits such that for some of the b+1 records $c = c_0 \cdots c_{l(K)}$ one has $c'_0 \cdots c'_i <_\Sigma c_0 \cdots c_i$.

1.2.2.2) Allocate bucket 1 and insert all keys satisfying the condition from 1.2.1.1 into bucket 1; the other keys remain inserted in bucket 0.

1.2.2.3) Create i+1 nodes, and set their values according to the diagram below:

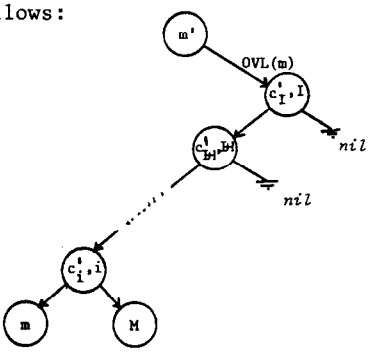
1.2.3 Further collisions.

As further insertions occur buckets which have been allocated will overflow. Such a bucket will be divided in two. We proceed as follows: Let m' be the node containing a pointer to the overflown bucket; we denote this pointer by OVL(m'). Let m be the address of the overflown bucket, and let M be the number of buckets already allocated.

1.2.3.1) Select the initial segment $c'_0 \cdots c'_i$ of of the middle key of minimal length, which can serve to split the bucket as in 1.2.2.1.

1.2.3.2) Compute the address for the key $c'_0 \cdots c'_i$ using the key to address transform algorithm described in 1.2.4; however, during this search an additional counting is performed. We have an integer I initially equal to 0. Each time we encounter in the address computation a node with DN = I and DV = c_I we let I := I+1.

1.2.3.3) Create i-I+1 nodes and set their values as follows:



1.2.3.4) Allocate bucket M and divide the b+1 records in the overflown bucket between the buckets m and M as indicated in 1.2.2.3. Set M := M+1.

The purpose of the address computation in 1.2.3.2 is the following. Litwin's key to address transform algorithm builds at every node of the path traversed a string against which the key search for is compared. The selected splitting key $c'_0 \ldots c'_i$ must be buildable by this algorithm. The nodes that enable us to build the string $c'_0 \ldots c'_{I-1}$ (which may be empty), are found to be already present in the trie; the remaining ones are created by step 1.2.3.3..

1.2.4 Key to address transform.

At any stage during the creation of the trie we assume that any key is stored in the bucket where it would be inserted if it was not there yet, and that every new record will be inserted into the bucket where it would have been if it had been subject to all splits already performed. This rule leads to the following algorithm:

Let c be the searched key, whereas T is the trie considered. The pointer p is initialized to the root of T. The strings s and t initially are empty.

```
if M = 0 then return 0 # no nodes created yet #
else result := -1 ;
   while result < 0
   do if DN(p) ≥ 1 then s:=s0...sDN(p)-1DV(p)
                   else s:=DV(p)
      endif; c' := c0...cDN(p) ;
      if s <Σ c' then p:= UP(p) ; s:=t
               else p:= LP(p) ; t:=s
      endif;
      if p is a bucket address then result
      := p endif
   endwhile;
   return result
endif;
```

During the algorithm s denotes the string against which the current input key c is tested to direct the further search. The string t represents the cumulative information gathered during the search so far. The statements s:=t and t:=s are needed for the correct performance of the algorithm since the information found in the current node must be discarded when the algorithm chooses the upper-pointer. These two statements were not included in the original paper by Litwin.

In order to gain further insight in the operation of the algorithms and the trie data structure the reader might consult Litwin's original paper. It is, however, difficult to infer from this paper the essential properties on which the correctness of the trie structure and its supporting algorithms are based. Such insight is needed if one wants to investigate the problems of reconstruction and optimization discussed in the introduction.

The formalization of the trie structure proposed in the next section is an attempt to locate exactly the fundamental properties underlying Litwin's proposal.

2. INFORMATION SAVING FOR RECONSTRUCTION.

2.1 Formalization of tries.

Before we can give tools for saving information necessary for reconstruction of the trie, we must determine which information needs to be saved. To do this we look at the key-to-address-transform algorithm presented in section 1.2.4. This algorithm builds strings to compare to (a prefix of) the searched key and then chooses the upper- or lower-pointer depending on the outcome of the comparison. When choosing the upper-pointer the information found in the current node is discarded; in the other case the information found in all previous nodes is "trimmed" - all previous information gathered in nodes with DN ≥ DN(current node) is discarded.

Now, evidently, if two keys k and k' in the key are hashed into different buckets, the key-to-address-transform algorithm somewhere must build a string $c_0 \ldots c_i$ on the path in the trie it follows, such that, if compared against this string, both keys k and k' behave differently for the first time. Assuming without loss of generality (WLOG) that $k <_\Sigma k'$ the condition expressing this behaviour can be expressed as follows:

2.1.1.1) for $j=0,\ldots,i$ $k_j = c_j$ or $\exists_{j \le i}[k_j < c_j]$

2.1.1.2) for $j=0,\ldots,i-1$ $k'_j = c_j$ & $k'_i < c_i$.

If the above condition is fulfilled at the node where $c_0 \ldots c_i$ is built and if k and k' arrive at this node then the algorithm will choose the lower-pointer in the case of k and the upper-pointer in the case of k'. Since this is the only way in which two strings can become separated, it seems reasonable to assign to the Litwin trie a *formal trie* consisting of all strings which can be built by the key-to-address-

transform. We then obtain a system of probes on which we can decide whether or not two keys are hashed into the same bucket.

2.1.2 Definition: A *(formal) Key-to-address-transform algorithm* on a key space K is a function $f:K \to \mathbb{N}$.

Whenever appropriate we use the abbreviation *ktat*.

2.1.3 Definition: A *(formal) Trie* T on a key space K is an ordered pair (V,f) with:

2.1.3.1) $V \subset \Sigma^*$ with 1) $\forall t = t_0 \ldots t_n \in V.[t_n \neq ':']$
2) $\forall t \in V[l(t) \leq l(K)]$
3) $t \neq t' \in V.[t \not\subset t']$.

2.1.3.2) f a key-to-address-transform algorithm on K such that
$\forall k < k' \in K[f(k) \neq f(k')$
$\leftrightarrow \exists t = y_0 \ldots t_n \in V.[\exists_{i \leq n}[k_0 \ldots k_i \leq_\Sigma t_0 \ldots t_i$
$\& \; k'_0 \ldots k'_{i-1} = t_0 \ldots t_{i-1} \; \& \; k'_i > t_i]]]$.

The property expressed by 2.1.3.2 will be described by the phrase " k is separated from k' by t at position i ", and will be notated $k <t,i] k'$.

2.1.4 Definition: An element $k \in \Sigma^*$ is a *prefix* in a trie $T = (V,F)$ iff there exists a string $t \in V$ such that $k \subseteq t$.

2.1.5 Definition: Two tries $T = (V,F)$ and $T' = (V',f')$ on a key space K are called *equivalent* iff their key-to-address-transform algorithms are equal:
$\forall k \in K[f(k) = f'(k)]$.

2.2 Consequences of the definitions.

The first consequence of the formal definitions above is the lemma below. It expresses the fact that for every string s which is not a prefix in T, a pair of keys separated by s can be found that are hashed to the same address.

2.2.1 Lemma: Let $T = (V,F)$ be a trie on a key space K, let $k \in K$ and let $s \in \Sigma^*$ be an initial segment of k which is not a prefix in T. If $s_{l(s)} \neq ':'$ then there exists a key k' such that $k <s,l(s)] k'$ but $f(k) = f(k')$.

proof: Let $s = s_0 \ldots s_n$. WLOG we may assume that the number n is minimal so $s_0 \ldots s_{n-1}$ is a prefix in t. Note that $s_0 \ldots s_{n-1}$ may be empty. Let $W = \{t \in V \mid s_0 \ldots s_{n-1} \subset t\}$. Again W may be empty. Since s is no prefix in T there exists no t in W with $t_n = s_n$, so one of the following cases arises:

case a) for some t' in W $t'_n > s_n$
case b) all t in W have $t_n < s_n$.

First consider case a; WLOG we select t'' in W with $t''_n > s_n$ and t'' minimal in the lexicographical order with this property. Then we let $k' = k_0 \ldots k_{n-1} t''_n k_{n+1} \ldots k_{l(K)}$. Now clearly $k <s,n] k'$. Assume that $f(k) \neq f(k')$; by condition 2.1.3.2 there must exist a string $\hat{t} \in V$ with $k <t,i] k'$ for some number i. But since k and k' are equal up to digit $n-1$ we must have $i \geq n$ and $k_0 \ldots k_{n-1} = \hat{t}_0 \ldots \hat{t}_{n-1}$, hence $\hat{t} \in W$. By the choice of t'' we have for all $t' \in W$ either $t'_n \leq k_n < k'_n$ or $k_n < k'_n \leq t'_n$, so $i = n$ is excluded as well. But if $i > n$ then by condition 2.1.3.2 one has $\hat{t}_n = k_n = s_n$ contradicting the fact that s is not a prefix in T. So $\hat{t}$ does not exist and hence $f(k) = f(k')$.

In case b we take $k' = k_0 \ldots k_{n-1} ':' k_{n+1} \ldots k_{l(K)}$. Since $s_n \neq ':'$ we again have $k <s,n] k'$, whereas the assumption that $f(k) \neq f(k')$ leads to a contradiction again. This completes the proof of the lemma. □

By force of this lemma we are able to isolate the main invariant of the trie data structure:

2.2.2 Proposition: Let $T = (V,f)$ and $T' = (V',f')$ be a pair of equivalent tries on a key space K. Then $\forall_{s \in \Sigma^*}[$ s is a prefix in $T \leftrightarrow s$ is a prefix in T'].

proof: Let $s = s_0 \ldots s_n$ be a prefix in T which is not a prefix in T'. WLOG we assume that $n = l(s)$ is minimal. There exists a t in V with $t_0 \ldots t_n = s_0 \ldots s_n$. First assume $s_n \neq ':'$. By lemma 2.2.1 there exist keys $k <_\Sigma k'$ such that $k <s,n] k'$ but $f'(k) = f'(k')$, so by equivalency we have $f'(k) \neq f'(k')$ as well.

Contradiction.

In the case that $s_n = ':'$ we consider the string $t = t_0 \ldots t_{l(t)}$; by 2.1.3.1 $t_{l(t)} \neq ':'$ and therefore $l(t) > n$. Let $t' = t_0 \ldots t_{l(t)-1}\,':'$ then by 2.1.3.2 $f(t) \neq f(t')$. On the other hand $f'(t) \neq f'(t')$ would imply that $t_0 \ldots t_n$ is a prefix in T' quod non. Contradiction again. □

2.2.3 Corollary: For equivalent tries $T = (V,f)$ the set V is uniquely determined.

proof: If T and T' are equivalent tries, and $t \in V \setminus V'$ then t is a prefix in T so there must exist a string t' in V' with $t \subset t'$ but $t \neq t'$. Again t' is a prefix in T' so t' must be a prefix in T as well, so there exists a string t'' in V with $t' \subset t''$. But now $t \subset t''$ for a pair of different strings in T contradicting 2.1.3.1.3. □

A trie $T = (V,f)$ defines an equivalence relation on the key space K by $k \simeq_T k'$ iff $f(k) = f(k')$. We can extend this equivalence relation to a partial ordering $<_T$ on K by: $k <_T k'$ iff $(k <_\Sigma k' \;\&\; f(k) \neq f(k'))$,

2.2.4 Claim: The ordering $<_T$ is linear.

proof: We only need to prove the transitivity of $<_T$. Assume therefore that k,k' and k'' are keys with $k <_T k'$ and $k' <_T k''$. The transitivity of the order $<_\Sigma$ implies $k <_\Sigma k''$. Furthermore, since $f(k') \neq f(k'')$ there exists a string t' in V and an integer i such that $k' <t',i]\, k''$. By the definition of lexicographical order we have $k <_\Sigma k' \Rightarrow k_0 \ldots k_i <_\Sigma k'_0 \ldots k'_i$ whence $k <t',i]\, k''$ as well, so $f(k) \neq f(k'')$. □

By the above ordering $<_T$ the equivalence classes determined by the trie are linearly ordered. Since f is determined by V, the entire structure is determined by V. Denote the collection of equivalence classes by K/T. The number of elements in K/T denoted $\#K/T$, depends on V only. In fact one has:

2.2.5 Proposition: Let $\tilde{V} = \{s \in \Sigma^* \mid s$ is a prefix in T and $s_{l(s)} \neq ':'\}$, then $\#K/T = \#\tilde{V} + 1$.

proof: 1) $\#K/T \leq \#\tilde{V} + 1$. This follows immediately from the fact that $<_T$ is linear: Let $\#V = n$, and suppose that $\#K/T \geq n+2$. Select a sequence of $n+2$ keys $k^{(1)},\ldots,k^{(n+2)}$ from different equivalence classes in K/T, such that $i < j$ implies $k^{(i)} <_T k^{(j)}$. Each pair $k^{(i)} <_T k^{(i+1)}$ is separated by some prefix p in T which implies $p_0 \ldots p_{l(p)-1} = k^{(i+1)}_0 \ldots k^{(i+1)}_{l(p)-1}$ and $p_{l(p)} < k^{(i+1)}_{l(p)}$. This implies that $p_{l(p)} \neq ':'$; moreover the prefixes p involved must be distinct. Since there are $n+1$ pairs in the sequence there must be at least $n+1$ different prefixes in T not ending in $':'$. Contradiction. □

2) $\#K/T \geq \#\tilde{V} + 1$. We prove this inequality by providing $\#\tilde{V} + 1$ keys which are mutually inequivalent. Let $n = \#\tilde{V}$, and let $\tilde{V}$ consist of the strings $t^{(0)},\ldots,t^{(n)}$ in lexicographical order. We define the set $\tilde{K} = \{k^{(0)},\ldots,k^{(n)}\}$ by letting:

$$k^{(0)}_j = '\_' \quad \text{for } j = 0,\ldots,l(K)$$
$$k^{(i)}_j = t^{(i)}_j \quad \text{for } j < l(t^{(i)})$$
$$k^{(i)}_j = \text{the successor of } t^{(i)}_j \text{ in } \Sigma \quad \text{for } j = l(t^{(i)})$$
$$k^{(i)}_j = '\_' \quad \text{for } l(t^{(i)}) < j \leq l(K).$$

Clearly $\tilde{K} = n+1$, so it suffices to prove that for $i \neq j$ we have $f(k^{(i)}) \neq f(k^{(j)})$. Assume that $k^{(i)} <_\Sigma k^{(j)}$, and let m be the least number such that $k^{(j)}_m > k^{(i)}_m$. Now by construction $m \leq l(t^{(i)})$ so either $m = l(t^{(j)})$, in which case we have $k^{(i)} <t^{(j)},m]\, k^{(j)}$, or $m < l(t^{(j)})$, in which case $k^{(i)} <_\Sigma t^{(j)}$ and a fortiori $k^{(i)} <t^{(j)},m]\, k^{(j)}$. In both cases one obtains $f(k^{(i)}) \neq f(k^{(j)})$. □

The reader should observe that the linear ordered collection of equivalence classes K/T is nothing but the chain of buckets into which the key space is hashed by the trie, where the

lexicographical order coincides with the pre-order in the trie.

2.3 Storage of information.

We have seen above that the information needed for rebuilding a trie equivalent to the original trie in the sense of definition 2.1.5 is the collection of strings V, combined with a sequence of bucket addresses. The set V determines the "formal buckets" K/T, together with their linear order, but fails to indicate to which bucket address such equivalence classes are mapped by the ktat-algorithm f.

The algorithms in section 1 now can be modified in such a way that they will save and update this information on disk; each time the trie is altered by the allocation of a new bucket, the information on disk is modified. For this purpose we reserve two areas on disk, henceforth denoted NS and BS. Here NS will store a collection of strings and BS a sequence of integers.

Initially NS = $\emptyset$ and BS = (0). Each time we select the proper initial segment of a middle key $c_0' \ldots c_i'$ in step 1 of algorithms 1.2.2 or 1.2.3, we extend NS by this string $c_0' \ldots c_i'$, removing from NS the string $s \subset c_0' \ldots c_i'$ provided such a string is present in NS. The modification of BS is somewhat trickier, since a new bucket may be allocated either by splitting an overflown existing bucket, or by inserting a first key in a bucket located at a position in the trie where one had a *nil*-pointer before. In both cases the sequence BS is updated by either inserting the new bucket M after the number corresponding to the overflown bucket, or by replacing the *nil*-address by address M+1.

In the second case we face the problem that there may occur several *nil*-pointers in the trie, and one must indicate which one should be updated. We propose the following solution: instead of having a single value *nil* we use a variable *NIL*, initialised at -1; whenever a new value *nil* should be assigned to some pointer p we perform the actions p := NIL; NIL := NIL-1. Now in case the insertion algorithm finds a bucket address < 0 it will allocate a new bucket M+1 at this point in the trie, and increase M by 1; next it will insert the proper disk address in the sequence BS at the position taken by the negative bucket address found. This modification has as a consequence that the key-to-address-transform algorithm 1.2.4 must be recoded, since the outcome of the test "result < 0" no longer is valid. Instead, by use of a tag-bit one must discriminate between nodes and bucket addresses as values of pointers.

The update on BS seems to require a sequential search on BS for locating the update position. In the case of a split a sequential scan seems needed anyhow for the copying due to the insertion in the middle. In the case of a *nil*-replacement a local update will do, provided we can have direct access to the update position. This position can be computed during the ktat-algorithm provided we reserve additional storage at every node for storing weight information (IE., the number of nodes in the left subtrie). With this additional information a single *nil*-value will again suffice.

Since in our paper the primal purpose of storing BS and NS on disk is the availability of this information for reconstruction purposes, we abstain from investigating the organization of this information in detail. During the running of the algorithm we assume to have a copy of BS and NS available in core as well. If core is scarce it may be necessary to store BS and NS on disk only, which will lead to additional problems concerning the organization of this information on disk.

Obviously, the reloading of NS and BS requires at least one disk-access. If more disk-accesses are needed they would be required anyhow for bringing the reconstruction information into core. From the assumption made that the entrie trie can be kept in core, it seems reasonable to infer that no further disk-accesses are needed for performing the reconstruction. Note that one does not need to have BS and NS in core at the same time.

3. RECONSTRUCTION AND OPTIMIZATION.

3.1 Equivalence between Litwin tries and formal tries.

In the previous section we derived a necessary condition for two formal tries to be equivalent; they should have the same set of strings: V = V'. In the present section we address the following two questions:

1) In which way does a Litwin trie determine a formal trie?
2) In which way can one obtain from a formal trie a Litwin trie which is equivalent in the sense that its corresponding formal trie is the one given.

It turns out that the correspondence between Litwin tries and formal tries is many-one. Consequently, question 2 can be refined to the following two subproblems:

2a) (Reconstruction) Given a formal trie, construct as easily as possible an equivalent Litwin trie
2b) (Optimization) Given a formal trie, construct a Litwin trie which is optimally balanced.

We describe a greedy algorithm for solving 2b. We conjecture that this algorithm indeed yields an optimal trie, but due to the way in which subsequent moves in the greedy algorithm depend on previous moves - a phenomenon which does not occur in the superficially related problem of the reconstruction of an optimally balanced binary search tree [1] - we are unable to prove

this. Clearly, an optimal trie can be obtained by a back-tracking algorithm, but we abstain from proposing such a method.

3.2 From Litwin tries to formal tries.

The key-to-address-transform algorithm 1.2.4 is the ultimate tool which decides whether two keys k and k' are mapped into different buckets. Let $k <_\Sigma k'$. These strings are mapped into different buckets iff the algorithm, having arrived at some node p with $DV(p) = x$, $DN(p) = n$ will move left while transforming k whereas it moves right when transforming k'. If s is the string built at p by the ktat-algorithm, we observe that this occurs iff $k <s,n] k'$, and if this situation never occurred on the path from the root to p before, since otherwise the paths traversed by k and k' would have diverged at some earlier stage.

Conversely, the only way a prefix w in a formal trie can perform its separation task in a Litwin trie, is by occurring as string s built at some node p in a Litwin trie be the ktat-algorithm; moreover, there should exist a pair of keys k and k' such that the ktat-algorithms arrive at p transforming both k and k', such that k and k' are separated by s. To express this crucial role of the string s we define:

3.2.1 Definition: Suppose that the key-to-address-transform algorithm follows a path from the root to node p and has, upon arriving in p, formed string $s = s_0 \ldots s_m$; let $DN(p) = n$ and $DV(p) = x$; then node p is said to *represent* the string $t = s_0 \ldots s_{n-1} x$ in the trie.

3.2.2 Definition: Given a Litwin trie constructed as described in section 1.2 its corresponding formal trie consists of all maximal strings represented at some node p in T.

In a Litwin trie the buckets are assigned to leaves, whereas in a formal trie buckets are obtained as equivalence classes in K/T. In both cases there exists a linear order on the set of buckets, being the trie traversal order for the Litwin trie and the order $<_T$ for the formal trie. In order that the formal and the Litwin trie show equal behaviour these two orders should coincide.

3.2.3 Proposition: The sequence order of buckets met in preorder traversal of a Litwin trie equals the order $<_T$ on the set K/T obtained from its corresponding formal trie.

proof: Let k and k' be a pair of keys hashed into buckets B and B' such that B precedes B' in preorder traversal. The ktat-algorithm, having arrived at some node p goes left for k and right for k'. This is equivalent to saying that $k <s,n] k'$ for the string s represented at p where $n = l(s)$. This exactly means $k <_T k'$.

3.3 From formal tries to Litwin tries.

Assuming correct behaviour of the key-to-address-transform algorithm on a well-designed Litwin trie, it is clear that 3.2.2 is a well-defined notion. On the other hand, we could extend notion 3.2.1 by investigating the behaviour of the ktat-algorithm in a trie consisting of an arbitrary collection of nodes, disregarding the question whether the algorithm would ever reach certain nodes while processing a key. We simply inspect what the algorithm would do in case some genie forces it to traverse some path from the root to some node p. Under these circumstances the following observation can be made:

3.3.1 Observation: The string s represented at p is determined entirely by the values $DN(p)$, $DV(p)$ in combination with the string s' represented at the lowest ancestor q of p in the path from the root to p where a lower-pointer was taken.

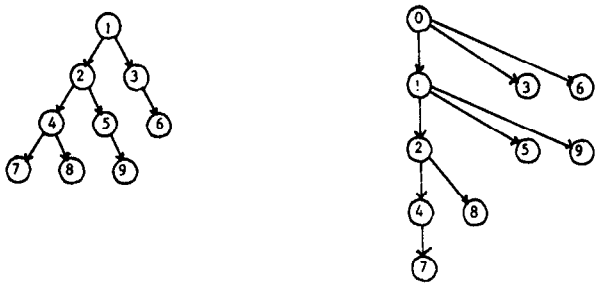
This observation is based upon the fact that information gathered at some node is discarded when the upper-pointer is followed.

The observation shows that the structure of a binary tree present in a trie is less suitable for investigating the interactions of the various nodes. In order to get some grasp on the relation between nodes p and q above, we introduce on the nodes of a Litwin trie another tree structure, called the *godfather tree*.

3.3.2 Definition: The *right (left) spine* of a binary tree is the maximal path consisting of right (left) pointers only, starting at the root. A right (left) spine in some tree is the right (left) spine in some subtree.

3.3.3 Definition: Given a binary tree its *godfather tree* is obtained as follows: each node becomes the son of its *godfather* - the lowest ancestor of which it is a left-descendant. The different sons of a godfather form the right spine of its left subtree; they are ordered as sons in the order in which they occur on this right spine. A new root is introduced which becomes the godfather of all nodes on the right spine of the original tree - these nodes have no godfather in the binary tree.

This is a well-known transformation transforming binary trees into ordered forests, where we have added the new root to obtain a tree instead of a forest. The transformation is easily inverted. Given an ordered tree a binary tree is obtained by transformaing the ordered list of sons of node p into the right spine of its left subtree (and discarding the root which obtains no right son). See the figure below:



3.3.4 A binary tree with corresponding godfather tree.

Observation 3.3.1 shows that the string represented by some node in a Litwin trie is not determined by the entire path between the node and the root but by the nodes on its godfather path to the new root. In order to make 3.2.1 meaningful for nodes on the right spine we must give the new root in the godfather tree a DN-value and a DV-value as well. We take DN = -1 and DV = '_'; as a consequence the new root represents the empty string, and the ktat-algorithm will always select the lower-pointer, IE., it will enter the real trie.

Next consider the situation where a node p with $DN(p) = n$, $DV(p) = x$ has a godfather q representing the string $s_0 \ldots s_m$; we have $m = DN(q)$. Now p will represent the string $s_0 \ldots s_{n-1} x$, but this is meaningful only if $n \leq m+1$; otherwise a string "with holes" will arise. This observation leads to a further restriction on the structure of Litwin tries:

3.3.5 Observation: For every node p with godfather q one has $DN(p) \leq DN(q)+1$; consequently all nodes on the right spine have $DN(p) = 0$.

From the fact that on the godfather path to the root the length of the represented string never decreases by more than 1 we obtain by a simple induction on string length:

3.3.6 Proposition: If string $s = s_0 \ldots s_n$ is represented at some node p in the trie then the initial segment $s_0 \ldots s_{n-1}$ is represented somewhere on the godfather path of p to the new root.

Consequently, if a string s is represented then so will be all its initial segments.

The numerical observation 3.3.5 still does not cover all structure present in a Litwin trie, for the fact that every node might represent a good looking string does not yet enforce that this node will ever separate a pair of keys. To see this assume that the ktat-algorithm, while processing a key k, arrives at node p where string s is represented. Suppose that the lower-pointer is selected, indicating that either $k <_\Sigma s$ or $s \subset k$. Next we arrive at some node q representing a string t such that $s' <_\Sigma t$ for every extension $s' \supseteq s$. Then the lower-pointer is selected at q regardless of which string k brought us there, so node q is useless - its right-hand subtrie will never be entered.

Similarly, it is useless to represent in the right-hand subtrie of p any string t such that $t <_\Sigma s$ or $s \subseteq t$, since then the lower-pointer at q will never be selected. Finally, it is meaningless to represent an initial segment of s in any subtrie of p since it again would lead to a test with one impossible outcome. This leads to the following observation.

3.3.7 Observation: Let the nodes p and q in a Litwin trie represent strings s and t respectively. If q occurs in the left-hand subtrie of p then either $t <_\Sigma s$ or $s \subset t$; if q occurs in the right-hand subtrie of p then $s <_\Sigma t$ and $s \not\subset t$. If q occurs below p in the trie then $t \not\subset s$.

3.3.8 Proposition: In a Litwin trie, satisfying condition 3.3.7, every string s is represented at most once in the trie.

proof: Condition 3.3.7 clearly excludes the case that a string is represented twice on a path from the root to some node in the tree. So if string t is represented at two different nodes q and q' we may assume that q and q' are unrelated. Let p denote the lowest common ancestor of q and q' in the trie, and let s denote the string represented at p. Since q and q' are unrelated we have $q \neq p \neq q'$. Assume WLOG that q occurs in the left subtrie of p. By 3.3.7 we have $t <_\Sigma s$ or $s \subset t$ since t is represented by q. But we also have $s <_\Sigma t$ and $s \not\subset t$ since t is represented by q' in the right subtrie of p. Contradiction. □

3.3.9 Corollary: For every Litwin trie the correspondence between nodes and prefixes of its formal trie will be one-one.

Under this correspondence the empty prefix is mapped onto the new root of the godfather tree, sowe'd better disregard the empty string from being a prefix.

There remains the step of showing that for a given formal trie a Litwin trie can be defined,

which moreover should satisfy the conditions 3.3.5 and 3.3.7. Such a trie now is easily obtained by first constructing its godfather tree and subsequently transforming it into a binary tree.

3.3.10 Theorem: For every formal trie $T = (V,f)$ there exists a Litwin trie T' satisfying conditions 3.3.5 and 3.3.7, such that T is the formal trie corresponding to T'.

proof: The godfather tree $\tilde{T}$ of T' is obtained as follows: the nodes of $\tilde{T}$ are the prefixes of strings in V including the empty one. The godfather of every nonempty prefix $x_0 \ldots x_n$ in $\tilde{T}$ is the prefix $x_0 \ldots x_{n-1}$ in $\tilde{T}$. Brothers are ordered lexicographically. At node $x = x_0 \ldots x_n$ in $\tilde{T}$ we let $DN(x) = n$ and $DV(x) = x_n$. For the root r we have $DN(r) = -1$ and $DV(r) = '\_'$. Transforming $\tilde{T}$ into a binary trie by the standard construction yields the required Litwin trie T'.

By construction the trie T' satisfies 3.3.5; in fact we have for node p with godfather q that $DN(p) = DN(q) + 1$. Moreover, by construction node x represents string x. Hence, the left subtrie of x only represents extensions $y \supset x$, whereas the right subtrie of x only represents strings y with $x <_\Sigma y$ and $x \not\supset y$. All initial segments of x form precisely the godfather path from x to the new root r. □

The following property of Litwin tries can be obtained from the structural constraints 3.3.5 and 3.3.7 as well:

3.3.11 Proposition: The value $DN(p)$ is nondecreasing if p traverses a right spine in a Litwin trie.

proof: Assume to the contrary that there exists a right spine $p_1, p_2, \ldots, p_a$ such that for some j $DN(p_j) > DN(p_{j+1})$. Take j minimal with this property. Let q be the common godfather of all nodes p_i and let $s = s_0 \ldots s_n$ be the string represented at q, so $n = DN(q)$. Let $m = DN(p_j)$, $m' = DN(p_{j+1})$. By 3.3.5 we have $m \leq n+1$; combined with $m > m'$ this yields $m' \leq n$. Let $x = DV(p_j)$, $x' = DV(p_{j+1})$. We conclude that the nodes represent strings $s_0 \ldots s_{m-1} x$ and $s_0 \ldots s_{m'-1} x'$ respectively. Since p_{j+1} occurs in the right subtrie of p_j one has $s_0 \ldots s_{m-1} x <_\Sigma s_0 \ldots s_{m'-1} x'$ with inclusion being excluded both by the inequality $m' < m$ and by 3.3.7. On the other hand p_{j+1} occurs in the left subtrie of q so we have $s_0 \ldots s_{m'-1} x' <_\Sigma s_0 \ldots s_n$, again the case of inclusion $s_0 \ldots s_n \subset s_0 \ldots s_{m'-1} x'$ being excluded by $m' \leq n$. But, after truncating these two lexicographical inequalities upto length $m' \leq m-1, n$ we obtain:

$$s_0 \ldots s_{m'} <_\Sigma s_0 \ldots s_{m'-1} x' <_\Sigma s_0 \ldots s_{m'}$$

yielding a contradiction. □

3.4 The reconstruction algorithm.

We will now present an algorithm which rebuilds a trie from the sets NS and BS satisfying the structural constraints on Litwin tries 3.3.5 and 3.3.7. As the proof of theorem 3.3.10 is constructive we could easily convert this proof to a reconstruction algorithm. We will bypass the godfather tree in our construction; as the reader may easily verify the results are the same.

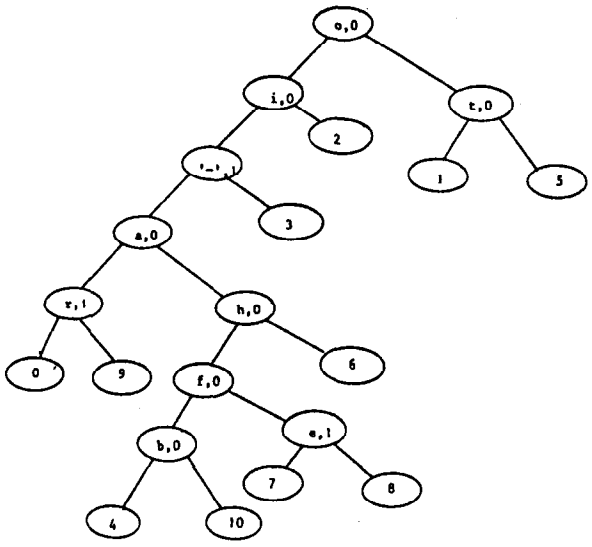
We define a recursive procedure GENTR that builds a left subtrie for a node p and returns a pointer to the root of this subtrie. The parameters are a set of strings L and an integer level.

```
procedure GENTR ( L,level )
begin declare pointer p ; p:=nil ;
for c in Σ from ':' down to '_'
do L_c := {v∈L | v_level = c} ;
   if L_c ≠ ∅
   then create node n_c ; UP(n_c):= p ;DV(n_c):= c ;
                          DN(n_c):= level ;
                          LP(n_c):= GENTR(L_c, level+1) ;
   endif
endfor;
return p
end .
```

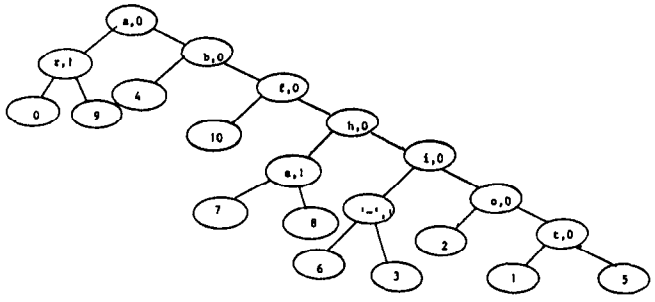
Procedure GENTR is called by GENTR(NS,0). Next we assign bucket addresses to the *nil*-pointers by:

1) set i:= 0 ;
2) traverse the trie in preorder and for each pointer p = *nil* encountered set p := BS[i] and i := i+1 .

3.4.1 Example: In the paper by Litwin a trie was presented with alphabetical keys, constructed by insertion of the most frequently used English words inserted in the order of decreasing frequency. After construction it looks like:



If we apply the reconstruction algorithm to this trie we get:



The sets NS and BS belonging to this trie are:

NS = {ar,b,f,he,i'_',o,t}

BS = {0,9,4,10,7,8,6,3,2,1,5} .

3.5 Transformations on tries for optimization.

Given a formal trie V and a Litwin trie T such that V is the formal trie corresponding to T, we investigate now how much freedom there remains in modifying T preserving the equivalence with V. Clearly, the structural properties 3.3.5 and 3.3.7 must be preserved as well. The same holds for the consequences of the properties like 3.3.6, 3.3.8 and 3.3.11 .

Consider a node p in the trie representing string $s_0 \ldots s_n$. By 3.3.6 the string $s_0 \ldots s_{n-1}$ is represented at some node q which is located on the godfather path between p and the new root. It is not excluded by 3.3.6 that this godfather path between p and q contains other nodes as well. However, it is not difficult to see that these intermediate nodes only can represent extensions s' with $s_0 \ldots s_{n-1} \subset s'$ and $s_n < s'_n$.

The following observation is a direct consequence of 3.3.7 and 3.3.8 :

3.5.1 Given a Litwin subtrie T representing a collection of strings W, the string s represented at the root r of T determines completely the weight of the left- and right-hand subtrie of r .

This follows, since the nodes in T are in one-one correspondence with strings in W in such a way that strings s' satisfying $s' <_\Sigma s$ or $s \subset s'$ are represented in the left subtrie of r, whereas the right-hand subtrie only represents nodes with $s <_\Sigma s'$ and $s \not\subset s'$. This split is determined entirely by the string s .

In the case of the construction of an optimal binary search tree a condition like 3.5.1 trivially holds. This leads to the following well-known greedy algorithm: "select as the root a node yielding an optimal balance between the weights of the left- and right-hand subtrees and perform this optimization recursively on both subtrees". We would like to apply the same greedy approach in our case. However, condition 3.3.6 imposes constraints on the possible choices of the root. In fact, previous choices of roots of embracing subtries affect these constraints. For this reason we no longer can prove that the greedy approach, which underlies the algorithm presented below, indeed yields an optimal trie. We conjecture this to be the case. We can prove, however, that the trie obtained is equivalent to the one given.

As with the reconstruction algorithm, it is advantageous to explain the optimization by considering its functioning on the godfather tree structure. We need the following concept:

3.5.2 Definition: A node p in the trie (godfather tree) is called *undisturbed* if all its descendants in the godfather tree are undisturbed, and have DN-values > DN(p) .

Note that the nodes in the left subtree of an undisturbed node p represent extensions of the string s represented at p ; moreover, these extensions only are represented in the left subtree of p by 3.3.6 and 3.3.8 .

It is clear that the new root of the godfather tree corresponding to the trie built by the reconstruction algorithm in section 3.4 is undisturbed.

The greedy optimizing strategy which we have in mind, goes as follows:

(1) Select one of the nodes in the trie to be the new root, in such a way that for this node r we have $|w_L(r) - w_R(r)|$ as small as possible.

(2) Perform a local transformation, distributing the nodes in the left and right subtries, which preserves the equivalence;
(3) recursively apply steps (1) and (2) on both subtries of the new root.

Note that the new root in step 1 must be eligible, given the structural constraints on Litwin tries. Note moreover that the transformation in step 2 will affect both the trie and its godfather tree since the two determine each other in a unique way.

In the sequel we shall call a node for which the actions 1 and 2 have been performed a *processed* node. The greedy algorithm will process the nodes in the trie in a top-down manner.

Some further explanation is needed concerning the phrase "local transformation" in step 2. Such a transformation amounts to an assignment of new values to a small finite collection of pointers in the trie, involving nodes close to either the old or the new root in the subtrie considered.

The equivalence between the two Litwin tries involved (before and after the transformation) requires that the sets of strings represented are equal. Given the one-one correspondence between prefixes in the formal trie and nodes in the Litwin tries, and given the fact that the prefix represented at node p determines the values of DN(p) and DV(p) these values are preserved as well. We can use these values as "tags" for identifying these nodes during the transformation, even though a given DN - DV value pair can occur many times in the trie. All our transformations will leave the values of the DN- and the DV - fields invariant, and only update LP- and UP - pointers. The string represented at a node gives this node a unique identity. which enables us to trace its position during the transformation.

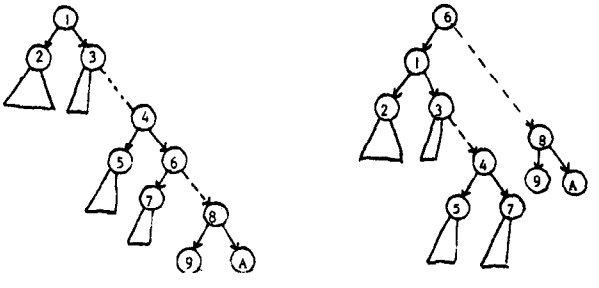
Consider a node p in the left subtrie of some node q. Under which circumstances can it become the root of the left subtrie of p while the set of left descendants of p remains unchanged? In general, this question has no clearly described answer. For example, the original root of the left subtrie of p may have been moved to the interior of this subtrie by some transformations, where it still is eligible of becoming the root by undoing these transformations.

In the sequel we present a lemma which states that all nodes on the right spine of the left subtrie of p are candidate roots. If moreover, some additional structural information is available it can be shown that these nodes are the only candidate roots.

For the right subtrie a similar question should be answered. However, in this case the role performed by node p becomes empty, since p does not occur as godfather for any node in its right subtrie. The same holds for all nodes in between p and its godfather. Hence, in order to investigate the candidate roots of the right-hand subtrie of p we will consider the situation where this right-hand subtrie is the left-hand subtrie of the godfather of p, and where all intermediate nodes, including p, together with their left descendants are eliminated. Under this interpretation the lemma announced above becomes meaningful for right subtries as well.

3.5.3 Lemma: Let p be a node and let T be the left subtrie of p, then for every node q on the right spine of T there exists a local transformation of T turning q into the root of T, which preserves the strings represented at these nodes. If moreover all godfather-sons of p are undisturbed then these godfather-sons are the only nodes eligible in the trie T for being the new root of T.

proof: We first describe a transformation on the left subtrie T of p, which turns an arbitrary node q on the right spine of T (IE., a godfather-son of p) into the new root of p, and which leaves the strings represented at every node invariant. The transformation is indicated in figure 3.5.4 below:



3.5.4 A local transformation.

The transformation makes q the left-hand son of p. The right subtree of q is unchanged. The left subtrie of q is inserted as a whole as right subtrie of the lowest node in the right spine of p's truncated left subtrie, which subtrie, after this modification, becomes the left subtrie of q.

We show that this transformation has the required properties. The only node that obtains left descendants which it did not have before, is node q. In terms of the godfather tree this means that q becomes the godfather of those brothers that preceded q in the order of p's godfather-sons. No other godfather relations are changed, from which it follows that all strings represented at nodes are unchanged, except for those elder brothers of q. It suffices therefore to show that these elder brothers represent the same string before and after the transformation.

Let q' be such an elder brother of q, and let s,t and s' be the strings represented by q,p and q' respectively. By 3.3.11 we

have $DN(q') = l(s') \leq l(s) = DN(q)$. Before the transformation we have:

$s = t_0 \cdots t_{DN(q)-1}{}^{DV(q)}$ and
$s' = t_0 \cdots t_{DN(q')-1}{}^{DV(q')}$.

After the transformation we have:

$s = t_0 \cdots t_{DN(q)-1}{}^{DV(q)}$ and
$s' = s_0 \cdots s_{DN(q')-1}{}^{DV(q')} = t_0 \cdots t_{DN(q')-1}{}^{DV(q')}$.

This shows that these strings are unchanged.

In order to prove the second part of the lemma we assume that all nodes on the right spine of p's left subtrie are undisturbed. Let q be one of these godfather-sons of p and let q' be a proper descendant of q in T. We show that q' cannot become the root of T without disturbing the strings represented at some nodes.

Let s and s' denote the strings represented at q and q'. Since q is undisturbed we have $s \subset s'$ before the transformation. By 3.3.6 after the transformation q must occur on the godfather path from q' to the root. However, if q' becomes the root of T then p becomes the godfather of q', so q would have to become a godfather ancestor of p, showing that the node q would have to be placed outside T after the transformation. Aside from no longer being a transformation that transforms the subtrie T inside itself, actually we can show that such a position outside T is impossible as well. Consider the string represented by p, say t. Since q occurs in the left subtrie of p by 3.3.7 we have $s \neq t$ and either $s <_\Sigma t$ or $t \subset s$. After the transformation q becomes a godfather ancestor of p and we have $t \neq s$ and either $t <_\Sigma s$ or $s \subset t$. This clearly yields a contradiction. □

Lemma 3.5.3 shows that the set of candidate roots of a (left) subtrie is restricted to the right spine of this subtrie, provided all these candidates are undisturbed. Starting with the trie resulting from the reconstruction algorithm from section 3.4 we know that all nodes are undisturbed. We need to show that the transformation from the previous proof leaves the undisturbedness of most nodes invariant.

3.5.5 Observation: Let p be a node such that all its godfather-sons are undisturbed, and let q be some node on the right spine of p's left subtrie. Then the transformation 3.5.4 performed in such a way that q becomes the root of p's left subtrie leaves the undisturbedness of all nodes in p's left subtrie invariant except for the new root q.

The proof of this observation is evident, given the fact that transformation 3.5.4 leaves all the DN-fields invariant, and given the fact that the only node obtaining godfather descendants which it had not before, is the new root q.

This observation is used in the next section to prove that our greedy optimization strategy investigates at each step all presently available candidates for being the root.

3.6 The optimization algorithm.

Having explained all necessary concepts and tools in the previous section we now come to a specification of our greedy optimization method. Our greedy method selects some node yielding an optimal balance on the right spine of the trie considered, turns this node into the new root by performing transformation 3.5.4 and calls itself recursively on both resulting subtries of the new root (if nonempty).

The method is described by the following recursive procedure OPT; its input parameter is a pointer to the trie to be optimized.

```
procedure OPT(p)
begin declare pointer p1,p2,new root ; integer M ;
p1 := p2 := p ; M := w(p)-1 .
while w_R(p1) > M/2 do p1 := UP(p1) endwhile ;
if p1 ≠ p then
   while UP(p2) ≠ p1 do p2 := UP(p2) endwhile ;
   if (w_R(p2) - M/2)   (M/2 - w_R(p1))
     then new root := p2
     else new root := p1
   endif ;
   if new root ≠ p then
     p1 := p ; while UP(p1) ≠ new root do
                              p1 := UP(p1) endwhile
     p2 := LP(new root) ; LP(new root) := p .
                              UP(p1) := p2 ; p:=newroot
   endif
endif ;
if LP(newroot) is not a bucket address then
                    then OPT(LP(newroot)) endif ;
if UP(newroot) is not a bucket address
                    then OPT(UP(newroot)) endif
end .
```

Note that this procedure does nothing except for the recursive calls in case it finds out that the present root is already optimal.

It follows from lemma 3.5.3 that this algorithm yields an equivalent trie to the given one, since transformation 3.5.4 preserves the corresponding formal trie. Next consider the case that the input trie is obtained as the result of the reconstruction algorithm in section 3.4. Then the recursive top-down strategy preserves the following invariant:

3.6.1 Invariant: All nodes in the trie are either undisturbed or processed; all ancestors of processed nodes are processed.

As a consequence, given the fact that the invariant is established by having the reconstructed trie as input, we see that by 3.5.3 all possible candidate roots are located on the right spine of the trie given as an argument to OPT, and that this property holds for the subtries given as argument in the recursive calls as well. This shows that our optimization method is optimal within the class of greedy algorithms.

From the above we may conclude that our optimization method always finds a trie which is a binary trie in the class $[\frac{1}{2},\frac{3}{4}]$ in the sense of Mehlhorn [5], unless we find at some level a subtrie that consumes more than half of the nodes available. But in this extreme case we can do no better than shifting all these nodes either in the left or in the right subtrie, without violating some of the structural constraints on Litwin tries.

4. TEST RESULTS.

The reconstruction and optimization algorithms were coded in a program of approximately 1000 lines of C, and run on a PDP-11 under UNIX%. Tries were obtained by generating up to 5000 strings (with possible repetitions), both by random generation, or by processing existing text files. Bucket sizes 10 and 20 were used. The results indicate that the original trie is reasonable, the reconstructed trie being extremely unbalanced, whereas the optimized trie has an average path length of appr 2% up to 7% less than the original one.

Keys inserted: Intern. nodes	Max. Pathl.	Average Pathl.	Max. W_L-W_R	Average W_L-W_R	Average $\lvert W_L-W_R\rvert$
Constr	8	6.595	3	0.609	0.902
Recon 548:42	37	19.428	40	-16.634	16.634
Optim	7	6.476	1	- 0.292	0.488
Constr	9	6.837	15	1.119	1.452
Recon 558:43	40	21.325	39	-18.667	18.667
Optim	7	6.511	1	- 0.452	0.500
Constr	9	7.269	13	0.661	1.241
Recon 892:63	43	24.066	57	-20.850	20.850
Optim	8	7.066	2	- 0.367	0.700
Constr	11	7.965	26	1.071	1.894
Recon 1013:88	39	24.667	82	-20.337	20.771
Optim	11	7.583	5	- 0.108	0.831
Constr	16	10.902	172	1.546	3.114
Recon 5154:423	57	34.789	410	-28.814	29.513
Optim	19	10.175	19	- 0.0332	1.4916

% Unix is a trademark of Bell Laboratories

5. DISCUSSION.

In his paper Litwin introduced trie hashing by describing the algorithms in a verbose style, claiming a high performance. He did not present any theory on the data structures involved. Our present research was motivated by a question asked by H. van Emde Boas - Lubsen, after a presentation by Litwin in Jan 1980; she asked whether it was possible to reuse the structure after a system crash. The authors acknowledge to her this inspiring idea which led them to several unexpected theoretical results.

It turns out that the reconstruction algorithm is not too difficult to invent. The optimization method requires some more effort. The main problem, however, seems to show that these algorithms are correct. Hence, one must show that these algorithms preserve the relevant properties which were never defined by Litwin at all. Since the meaning of a Litwin trie, operationally is given by its key-to-address transform algorithm, it is reasonable to declare two tries to be equivalent in case they yield the same ktat algorithm. The theory developed in sections 2 and 3 represents the result of 1½ years of struggle by the authors with the problem of translating this operational equivalence in structural properties of the trie, based on which the correctness of our algorithms can be proved.

We believe that our experience gives a nice illustration of how non-trivial data structures require a rather complex mathematical analysis in order that interesting algorithms operating on them can be shown to be correct. Theoretical Computer Science can be practical!

After submission of our paper we were informed about another investigation related to Litwin tries by J. Dur & S. Koopmans, IBM DCSSC, Uithoorn [2]. Although their motivation was different from ours (their purpose being data compression, rather than optimizing search time), they arrived at a structure which at a theoretical level is similar to our reconstructed tries, except for the trivial modification of selecting for equal keys the upper-pointer instead of the lower one. Actually this choice would clean up the theory in as far as the complex condition $s \leq_\Sigma s'$ or $s' \subset s$ can be replaced by the simple condition $s <_\Sigma s'$; Litwin's choice of moving left on equal keys and the traditional definition of lexicographical order don't fit together well.

Some remaining open problems are:

5.1 The optimization algorithm is correct for arbitrary Litwin tries, but in general it will not investigate all possible candidate roots of a left subtrie. Can one obtain simple (heuristic) tools for locating candidate roots outside the right spine of the left subtrie? We have a method for locating such candidates on the left spine but no idea about the value of this heuristics.

5.2 The paper uses a number of hypotheses on the environment. We assume that the entire trie fits in core. Moreover, we assume that reconstruction information on disk (NS and BS) can be brought in core in a single disk access and can be processed in core together with the trie under reconstruction. Both assumptions seem to become unreasonable for large-scale applications. The problems on the proper paging and segmentation

methods required for obtaining a reasonable behaviour under circumstances where these assumptions are invalid, are left for further research. Possible solutions are the logging of transactions or the implementation on a virtual core machine.

5.3 To our knowledge the reconstruction problem has not been investigated for any dynamic data structure before. It might be interesting to know whether the overall strategy followed here is applicable to other structures, in order that they may be protected against system crashes and other nasty habits of computers.

5.4 Prove or disprove our conjecture that our optimization algorithm yields an optimal equivalent trie.

REFERENCES.

1 Aho, A.V., J.E. Hopcroft & J.D. Ullman, *The design and analysis of computer algorithms*, Addison Wesley 1974.

2 Dur, J. & S. Koopmans, *Cactus Index*, Internal Report IBM DCSSC, May 1983, Uithoorn, the Netherlands.

3 Knuth, D.E., *The art of computer programming vol. 3*, Addison Wesley 1973.

4 Litwin, W., *Trie Hashing*, Sirius Map I-013, Inst. Nat. de Recherche en Informatique et Automatique, 1980. Also in P.M.G. Apers, ed., *Colloquium Databankorganisatie, deel 2*, MC Syllabus 46.2 (1982), pp. 27-52.

5 Mehlhorn, K., *Dynamic Data Structures*, in J.W. de Bakker & J. van Leeuwen eds., *Foundations of Computer Science III, part 1*, MC Tracts 108 (1979), pp. 71-96.