# A New Representation For Linear Lists

by

Leo J. Guibas, Edward M. McCreight
*Xerox Palo Alto Research Center*

and

Michael F. Plass, Janet R. Roberts
*Stanford University*

## 1. Introduction

We present a new data structure for maintaining a set of records in a linear list according to their key values. This data structure has the property that we can keep a number of *fingers* at points of interest in the key space (e.g., the beginning or the end of the list), so that access and modification in the neighborhood of a finger is very efficient. More precisely, if we have $f$ fingers, then we can access, insert, and delete records at a distance $p$ away from one of the fingers in time $O(f + \log p)$. As an important special case, we have a representation for linear lists in which access or modification within $p$ records of the beginning of the list can be accomplished with cost $O(\log p)$. Our structure is based on B-trees ([1], [4, p.473], [6]) with a regularity condition asserted on certain paths, and appears to be eminently suited for maintaining a set of records when accesses are not randomly distributed over the key space but instead have a (perhaps time-varying) locale of interest. By keeping a finger in this locale we can get better performance in the worst case than is possible with any other known technique. In a different vein, we can use the same structure with *Insertion-Sort* to efficiently sort nearly ordered files. For example, if we know that the file has no more than $O(n \log n)$ inversions, then our algorithm will sort it in time $O(n \log \log n)$ in the worst case, which we can prove is best possible.

Each finger in the key space generates a *finger path*. This is the path to the root of the tree, from the leaf closest to the finger. We measure distance to a leaf by the distance to the closest key contained in the leaf, in any metric appropriate for our particular key space. (If there are two equidistant leaves, we arbitrarily choose one.) The finger paths are the paths on which we will maintain our regularity condition. The condition serves to guarantee that nodes which are too full or too empty do not pile up very close to each other. Thus the splitting and recombining operations necessary for maintaining the B-tree never have to propagate too far. Our problem is analogous to that of having to handle long carry propagation sequences in adding 1 to a number in binary. The average number of carries propagated in adding 1 to an integer of size $n$ is well known to be 1, when averaged over all integers from 1 to $n$. On the other hand, any particular addition may have to propagate a carry as many as $\log n$ steps. In solving the general problem we also obtain a way of representing integers so that 1 (or any power of 2) can be added/subtracted from our number while propagating at most 1 carry/borrow. This is quite similar to algorithms for generating all subsets of an $n$ element set with constant effort expended between successive sets ([2]).

In the Section 2 we discuss the general structure of our B-tree. Since we propose to search the tree from a leaf upwards, additional links need to be introduced. In Section 3 we show how to obtain our result for the case of one finger. A key idea is the construction of a number representation behaving as described above, which we can use to model the propagation of modifications in the B-tree along the finger path. In Section 4 we generalize the structure so that several fingers in the key space can be maintained, with the advantage that access is cheap in the neighborhood of each finger. Finally in Section 5 we present some implementation notes and applications, mostly to sorting.

## 2. Searching A B-tree From The Bottom Up

Any access or modification operation to our data structure begins with the search for a key. This will be the key being accessed, added, or deleted. Such a search is made always with reference to a particular finger. The search begins at the leaf node on the finger path, and ascends this path, while simultaneously examining certain nodes neighboring the path *in the direction of the search* (that is, if the key is larger than the finger then we look at nodes which are right neighbors of the path nodes, else we look at left neighbors). We stop ascending when we discover a node which subtends a range of the key space in which the key being sought lies. From that node we now search downwards, in the normal B-tree fashion. The details of this operation are described below, but this summary should suffice to motivate some of the additional structure we will need to introduce in each node of the B-tree.

Besides the usual keys and downward links, we will need to introduce the following fields in each node t of the B-tree:

*father:* pointer to father node

*lNbr:* pointer to t's left neighbor at the same level (can be omitted if this neighbor is t's left brother)

*rNbr:* pointer to t's right neighbor at the same level (can be omitted if this neighbor is t's right brother)

*lThread:* if t is a non-leaf node, then a pointer to the leftmost leaf t subtends, else a pointer to the internal node containing t's predecessor key in the linear order

*rThread:* if t is a non-leaf node, then a pointer to the rightmost leaf t subtends, else a pointer to the internal node containing t's successor key in the linear order

*leaf:* flag indicating whether or not t is a leaf node

We also assume that downwards pointers to subtrees that subtend some finger are *tagged* in some manner.

Suppose we are searching for key $k$ starting from finger $f$. Then the upwards part of the search can be described as follows. We assume $k > f$, the case $k < f$ being exactly dual. (If $k = f$, we are done).

UpwardSearch:

```
t ← leaf node on f's path;
WHILE t does not subtend k DO
   BEGIN s ← t.rNbr
   IF s = NIL THEN DONE
   COMMENT we hit the right spine of the tree;
      ELSE IF s subtends k THEN
         BEGIN
         t ← s;
         DONE;              COMMENT exit the while loop;
         END;
      ELSE t ← t.father
   END;
```

At this point t subtends $k$ or we have established that $k$ is larger than any key present in the tree and t is a node on the tree's right spine. In either case we can begin a regular downwards search looking for $k$.

For a node x let x.*smallest* and x.*largest* denote respectively its smallest and largest keys. Then the test "t subtends $k$" can be performed as the test "$k$ in Range(t)", where

Range(t) = IF t.*leaf* THEN [t.*smallest*,t.*largest*]
ELSE [t.*lThread.smallest*,t.*rThread.largest*];

Given any two nodes x, y in the tree, note that range gives us a O(1) test for deciding if x is an ancestor of y, namely whether Range(y) is a subset of Range(x).

LEMMA 2.1: If the key $k$ is $p$ keys away from $f$, then the UpwardSearch terminates in time O(log $p$).

*Proof:* It suffices to show that if the upwards search ascends by $l$ levels on the finger path, then $l$ = O(log $p$). This, however, follows from the fact that by needing to ascend from level $l$-1 to level $l$ on the finger path, we have established the existence of an entire subtree of height $l$-1 (namely the tree rooted at s in the above algorithm), all of whose keys lie between $f$ and $k$. ∎

We now consider insertions and deletions. We assume the reader is familiar with the way these operations are normally implemented in B-trees ([4, p.473, & p.679]). Recall that deletion of a non-leaf key is accomplished by replacing the key to be deleted with its successor in symmetric order and then deleting the successor. Such a successor must be a leaf key. Thus in what follows we can assume that we are dealing with insertion or deletion of a key in a leaf node. It is well-known that such an operation may cause the leaf node to overflow or underflow its allowed capacity, in which case an adjustment is necessary. Such adjustments may in turn propagate upwards in the tree.

We next discuss four types of operations that preserve the B-tree structure and which may be invoked to restore the density of a node to the allowed range. In this section we confine our attention to showing how these operations may be performed while still maintaining the above data structure for the tree. We postpone until the next two sections a discussion of *when* these operations are to be invoked, and on *what* nodes. The four operations are *splitting*, *absorbing*, *sharing*, and *relieving*. The first three of them can be performed in O(1) time, whereas *relieving* requires O($l$) time, where $l$ is the level of node to which it is being applied. These bounds will be evident from the algorithms described below.

## 2.1 SPLITTING

A node t splits when its capacity is exceeded. When a node with $x$ keys splits, two new nodes are generated, having $\lfloor x/2 \rfloor$ and $\lceil x/2 \rceil - 1$ keys respectively. The remaining middle key is inserted into the father and both newly created nodes become its offspring. Let $k$ be the middle key, and let lk and rk

respectively denote the downward links to its left and right in the original node. The algorithm for splitting is shown below, omitting the standard B-tree manipulations.

Split:

```
IF t.leaf THEN
    BEGIN
    IF t.father.lThread = t THEN
        BEGIN COMMENT t is the leftmost leaf of its father;
        s ← NewNode();
        <copy right half of t's keys and links into s>;
        <delete right half of t's keys and links from t,
            including k>;
        COMMENT note that the lThread of nodes pointing to
            t as their leftmost leaf need not be modified;
        r ← t.rNbr;  t.rNbr ← s;  r.lNbr ← s;
        s.lNbr ← t;  s.rNbr ← r;
        s.lThread ← s.rThread ← s.father ← t.father;
        s.leaf ← TRUE;
        <insert k and link to s in t.father>;
        <tag links to s and t in father according to whether
            s or t contain finger links>;
        COMMENT recall that links are tagged exactly when
            they are part of finger paths;
        END
    ELSE IF t.father.rThread = t THEN ....
    COMMENT this is the exact dual case -- note that for any
        non-leaf node y, y.lThread ≠ y.rThread, so only one
        of the above conditions can succeed;
    ELSE ....
    COMMENT this is the easy case -- either of the above two
        bodies of code will work;
    END
ELSE COMMENT t not a leaf;
BEGIN
s ← NewNode();
<copy right half of t's keys and links into s>;
<link new offspring of s to s>;
<delete right half of keys and links from t, including k>;
r ← t.rNbr;  t.rbnr ← s;  IF r ≠ NIL THEN r.lNbr ← s;
IF 1k.leaf THEN
    BEGIN COMMENT splitting at level 1;
    t.rThread ← 1k;
    1k.rThread ← t.father
    s.lThread ← rk;
    rk.lThread ← t.father
    END
ELSE BEGIN
    t.rThread ← 1k.rThread;
    1k.rThread.rThread ← t.father;
    s.lThread ← rk.lThread;
    rk.lThread.lThread ← t.father;
    END;
    s.lNbr ← t;  s.rNbr ← r;
    s.father ← t.father;
    s.leaf ← FALSE;
    <insert k and link to s in t.father>;
    <appropriately tag links to s and t in father, according
        to whether s or t contain finger links>;
    END;
END;
```

Figure 2.1 illustrates *splitting* at a leaf.

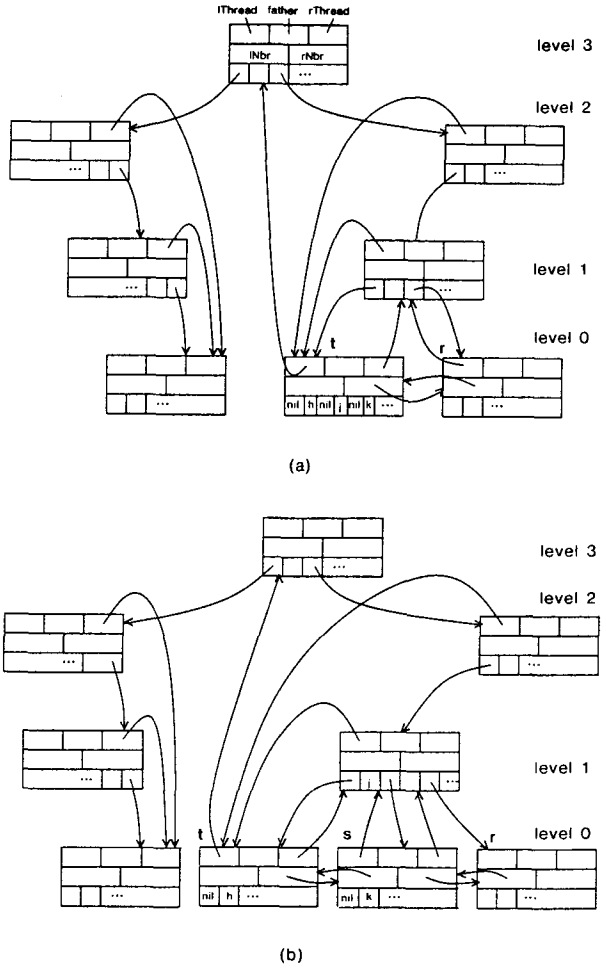



Figure 2.1. Splitting a Leaf
(a) before
(b) after

## 2.2 ABSORBING

A node which is underflowing may be brought into range by absorbing one of its immediate (e.g. left or right) brothers. If node t had $x$ keys originally and absorbs a brother having $y$ keys, then t will have $x+y+1$ keys after the absorption (one key $k$ comes down from t.*father* to separate t's links from those of its brother.

```
Absorb: COMMENT we describe the case of absorbing our
    right brother s (the left brother case is entirely
    analogous). Note that this operation is the reverse of
    splitting;

BEGIN
<copy k and the links and keys of s to the right end of t>;
<link old offspring of s to t>;
t.rNbr ← s.rNbr; s.rNbr.lNbr ← t;
IF t.leaf THEN
    BEGIN COMMENT this case has exactly the same
        complications as in splitting -- that is we must make
        sure that if either leaf of the pair entering this
        operation is an extremal leaf of its father, then it is
        the one that absorbs the other;
    ...
    END
ELSE BEGIN
    t.rThread.rThread ← t; s.lThread.lThread ← t;
    END;
t.rThread ← s.rThread;
<tag t.father's link to t if either of the links to t or s were
    previously tagged>;
Release(s);
END;
```

Figure 2.2 illustrates *absorbing* at level *i*.

### 2.3 Sharing

A node which is underflowing may also be brought into range by absorption, followed by a split; that is, the node and its brother evenly split all keys they collectively have. If node t has $x$ keys and the brother has $y$ keys, then after sharing, t will have $\lfloor (x+y)/2 \rfloor$ and the brother $\lceil (x+y)/2 \rceil$ keys. From the above descriptions of *splitting* and *absorbing* it should be clear how *sharing* can be performed.

### 2.4 Relieving

An overflowing node whose left (or right) neighbor is not a brother, may be brought into range by passing its leftmost (or rightmost) subtree to the neighbor. Similarly, if the node is underflowing, a subtree may be moved to it from its neighbor. This operation is similar to sharing, only it happens between adjacent nodes at the same level which are *not* brothers (and only one subtree is involved). It is for performing this operation that the leaf threads are kept in the tree.

```
Relieve: COMMENT node t passes its leftmost subtree to its
    neighbor and non-brother s -- all other cases are analogous.

BEGIN
IF t.leaf THEN
    BEGIN
    r ← t.lThread; COMMENT same as s.rThread;
    k ← key in r separating s from t;
    m ← leftmost key of t;
    <move k down to the right end of s>;
    <replace k by m in r>;
    <delete m from t>;
    END
ELSE BEGIN
    r ← t.lThread.lThread;
    COMMENT same as s.rThread.rThread;
    k ← key in r separating s from t;
    u ← leftmost subtree of t;
    m ← leftmost key of t;
    <move k and u to the right end of s>;
    u.father ← s;
    <replace k by m in r>;
    <delete u, m from t>;
    v ← right brother of u under t;
    COMMENT right brother must exist;
    w ← s.rThread;
    IF u.leaf THEN
        BEGIN
        s.rThread ← u; u.rThread ← r;
        t.lThread ← v; v.lThread ← u;
        END
    ELSE BEGIN
        s.rThread ← u.rThread; u.rThread.rThread ← r;
        t.lThread ← v.lThread; v.lThread.lThread ← r;
        END;
    COMMENT now swap left leaves;
    x ← u.lThread;
    y ← v.lThread;
    <swap the contents of nodes x and y>;
    <correct downward links in x.father and y.father>;
    <traverse left spines from u and v, while swapping
        lThread pointers>;
    COMMENT note this requires O(l) steps, where l = level of
        t;
    COMMENT we have interchanged x and y so that lThread
        links will still be valid for nodes which are ancestors
        of t;
    COMMENT now swap right leaves;
    z ← u.rThread;
    COMMENT swap z and w -- do exactly as above, but with
        rThreads instead;
    ...
    END;
END;
```

Figure 2.4 illustrates this operation.

We see that *relieving* can be expensive, since it is one that materially affects the path structure of the tree. As we will see in the next section, this operation will be invoked at most once in any particular search.

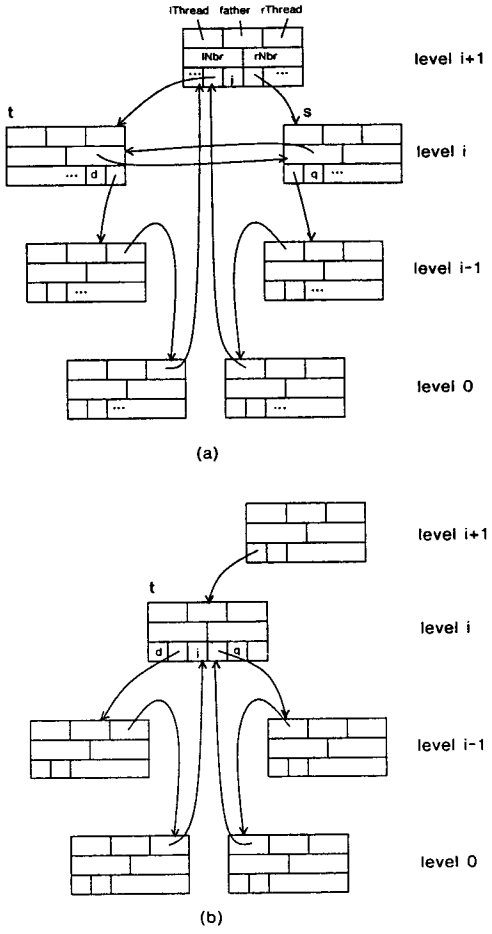
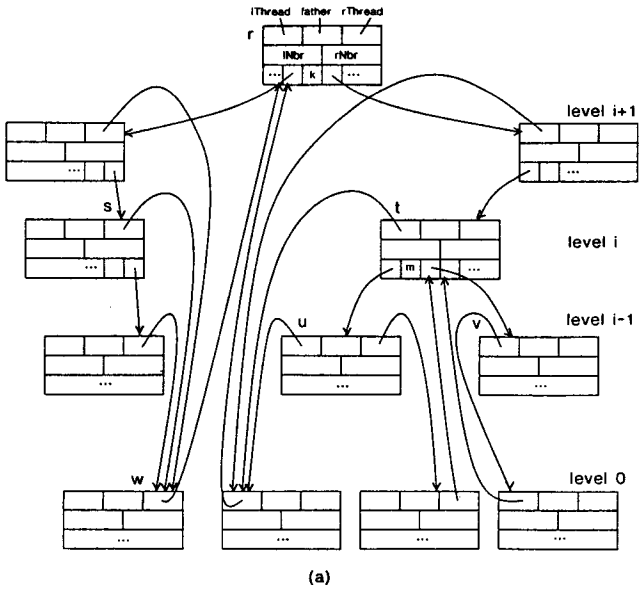



Figure 2.2. Absorbing a Node at Level i
(a) before
(b) after



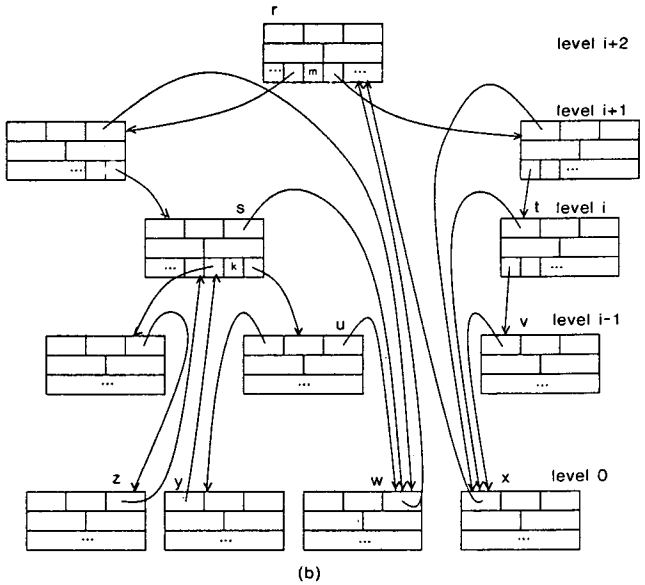




Figure 2.4. Relieving a Node at Level i
(a) before
(b) after

## 3. Maintaining Regularity On One Finger Path

In this section we confine our attention to maintaining exactly one finger in the tree. It is well known that in a B-tree with $n$ keys an insertion or deletion may occasionally require as many as O(log $n$) fixup operations of the kind we described in the previous section. It is clear that we will need to introduce some additional structure into the B-tree in order to guarantee our worst case O(log $p$) bound. The additional structure we impose is a regularity constraint on the densities of nodes on the finger path, essentially requiring that nodes which are nearly full (or nearly empty) cannot be too close to each other. There is a similarity between the situation in which a path of many contiguous full nodes requires many successive splittings, and the phenomenon in binary addition where a long string of 1's will force a carry to propagate very far. The regularity constraint described below was motivated by considering number systems in which a 1 could be added or subtracted from any digit position, while having to propagate at most a fixed small number of carries.

In our B-tree we will allow each node to contain between $\lfloor m/3 \rfloor - 1$ and $m$ keys, where $m$ is a fixed positive integer. We assign to each node a *state* (also called a *digit*, by analogy with a number system) in accordance with his density level:

| Density Level (in keys) | State | Digit |
|---|---|---|
| $\lfloor m/3 \rfloor - 1$ | underflow | U |
| $\lfloor m/3 \rfloor$ | near underflow | NU |
| $[\lfloor m/3 \rfloor + 1, m-2]$ | buffer | B |
| $m-1$ | near overflow | NO |
| $m$ | overflow | O |

We must remark that, as with all B-trees, we cannot demand a minimum density of the root node. Such a node may have only one key. This, however, will cause us no difficulty.

Let us consider what can happen to the tree during an insertion. The case of a deletion is entirely analogous. An insertion begins at a leaf. A sequence of splittings may now be started, which propagates upwards. Let t be the node found by *UpwardSearch* in the previous section. Then t subtends the leaf where the insertion begins. We know further that if $l$ is the level of t, then $l = O(\log p)$. If t is on the finger path, or t is the brother of a node on the finger path, then the successive splittings either terminate before they reach the finger path, or they reach it at level $l$ or $l+1$ respectively. If t satisfies neither of the above conditions then t must be a non-brother neighbor of a node on the finger path. In this case the splitting is not allowed to propagate past t. Should we need to split, we perform a *relieve* operation instead, transfering the burden of splitting to a node on the finger path. It is clear that our work up to now is bounded by $O(l)$.

The disturbance has now reached the finger path. As we will see below, our regularity condition will allow us to restore all necessary constraints in time O(1). We now summarize when the various operations of splitting, absorbing, sharing and relieving are to applied.

A node always splits if it has $m+1$ keys. A node on the finger path with $m$ keys will split if it receives the message *topple*, as described below. Absorbing and sharing are mutually exclusive operations; we call either operation a *borrow*. A node that needs to borrow will absorb its brother if the brother has no more than $\lfloor m/2 \rfloor$ keys. It will share with its brother if the brother has more than $\lfloor m/2 \rfloor$ keys. A node (other than the root) with $\lfloor m/3 \rfloor - 2$ keys must borrow. A node on the finger path with $\lfloor m/3 \rfloor - 1$ keys will borrow if it receives the message *topple*. Finally a node performs a *relieve* operation (giving or taking a subtree to/from a non-brother neighbor) exactly in the one situation encountered above.

LEMMA 3.1. If $m \geq 24$, after node t performs any of the four operations *split*, *absorb*, *share*, or *relieve*, all affected or newly created nodes at t's level will be in state B, with the possible exception of t's neighbor in a *relieve* operation.

*Proof:* The proof is a tedious but straighforward verification of simple inequalitites, and is omitted. ∎

We are ready to model the situation on the finger path. Regard the sequence of states of nodes on the finger path as a sequence of digits extending from right to left. The state of the leaf node corresponds to the rightmost digit. The regularity condition on the finger path can now be expressed by the following two invariants.

INVARIANT 1. Let the current state of the path correspond to a string of digits $\sigma$. If $\sigma = \alpha O \beta O \gamma$, where $\alpha$, $\beta$, $\gamma$ denote substrings, then in fact $\beta$ is non-empty and contains at least one U, NU, or B. The leftmost (e.g. highest in the tree) such digit will be called a *buffer digit for overflow*, denoted by state BO. (We assume $O \notin \beta$).

INVARIANT 2. If $\sigma = \alpha U \beta U \gamma$, then $\beta$ is non-empty and it contains at least one O, NO, or B digit. The leftmost such is called a *buffer digit for underflow*, denoted by BU. (We assume that $U \notin \beta$).

We postulate an imaginary U and O to the right of the rightmost digit, so as to force the invariants to apply always to the right end of the string.

How do we maintain these invariants? In an insertion or deletion the above discussion has demonstrated that there will be a *first* position in which a finger path can be affected. At that moment a node on the finger path will (1) receive or lose one key (and its associated subtree), or (2) be used in a borrow operation as the brother. Because of the symmetry between a node and its brother in borrowing, however, we see that the effect on the finger path will be the same as if it were the node on the finger path that underflowed (and who then borrowed from his brother). Thus case (2) reduces to case (1). In terms of our digit string model our problem then becomes: *How do we maintain the invariants accross an operation of adding or subtracting* 1 *to the density level of a node on the path?*

We call the act of splitting or borrowing of a node on the finger path a *topple*. A toppling node must be a U or O. A topple always renders the toppled digit a B, and then adds/subtracts a 1 to/from the following digit (on the left), according to whether the toppled node was a O or an U. We are now adequately prepared to deal exclusively with the model.

Shown below is a set of rules for adding a 1 into an arbitrary position s which maintain the above invariants while causing at most one topple. The idea is that when we add a 1 in a certain interval between two O's, if we are to the right of the buffer digit in the interval, then we topple ourselves (in case we have generated a new O), else we topple the next O to our left. For subtracting 1 we use the exact dual rules.

Let s be the position where we are adding/subtracting. Let ro be the leftmost position to the right of s where an O occurs. Let lo be the rightmost position to to the left of s where an O occurs, and let fo be the next O to the left after that. (Either lo, or lo and fo may fail to exist - but then our arguments below will be only easier.) Finally, let bo denote the buffer for overflow digit to the right of lo. (Position fo need only be considered in the verification of the correctness of the rules below). Schematically we have

```
O  ... O ...   BO ... O
fo     lo      bo    ro
```

We number the digit positions from right to left, the rightmost digit corresponding to the leaf node. Let ru, lu, fu, bu be correspondingly defined for underflows.

### Propagation Rules

For each digit transition, the required fixups for invariants 1 and 2 are shown. A fixup, if necessary, always consists of one topple operation. The parenthetical remarks below suggest how the roles played by various digits change; the implementation is postponed until the next section.

#### Add 1

| Digit Transition | Fixup Required |
|---|---|
| U → NU | 1: none<br>2: (remove U) |
| NU → B | 1: none<br>2: (update BU) |
| B → NO | 1: if we are the bo digit then topple lo<br>2: none |
| NO → O | 1: if we are to the left of bo then topple lo, else topple self<br>2: none |
| O | topple self |

#### Subtract 1 (exact dual)

| Digit Transition | Fixup Required |
|---|---|
| O → NO | 1: (remove O)<br>2: none |
| NO → B | 1: (update BO)<br>2: none |
| B → NU | 1: none<br>2: if we are the bu digit then topple lu |
| NU → U | 1: none<br>2: if we are to the left of bu then topple lu, else topple self |
| U | topple self |

#### Topple O

| Digit Transitions | Fixup Required |
|---|---|
| U O → NU B | 1: (remove O)<br>2: (remove U) |
| NU O → B B | 1: (remove O)<br>2: (update BU) |
| B O → B B | 1: (remove O)<br>2: none |
| B O → NO B | 1: (O to BO)<br>2: none |
| NO O → O B | 1: (new O and BO)<br>2: none |

#### Topple U (exact dual)

| Digit Transitions | Fixup Required |
|---|---|
| O U → NO B | 1: (remove O)<br>2: (remove U) |
| NO U → B B | 1: (update BO)<br>2: (remove U) |
| B U → B B | 1: none<br>2: (remove U) |
| B U → NU B | 1: none<br>2: (U to BU) |
| NU U → U B | 1: none<br>2: (new U and BU) |

**LEMMA 3.2. The above rules preserve invariants 1 and 2.**

*Proof:* We show only the argument for the Add 1 operation, that for subtraction being exactly dual. Invariant 1 can be destroyed by an Add 1 operation only if a BO, NO, or O digit is affected. We need to distinguish the following cases:

Case 1: The digit at s is O.

Then we topple at s. If bo = s+1, then, after toppling, s can serve as the BO between ro and lo. Else if we create an O at s+1, then position bo is unaffected and can serve as BO between s+1 and lo. Further position s which is now B can serve as BO for ro and s+1. Finally, if neither happens, then position bo can still serve as BO between ro and lo.

Case 2: bo > s > ro.

Problems arise only if we create a new O. But then this O will topple and we are reduced exactly to the argument of case 1.

Case 3: s = bo.

By our rules we topple lo. Thus at lo we now have a B, and the argument proceeds as in case 1.

Case 4: lo > s > bo.

We must create a new O. According to the rules, we topple lo. The digit at bo can now serve as BO between ro and s. To the left of position s the situation is ok by the argument of case 1.

Finally note that an Add 1 operation can never destroy invariant 2. If position s is a U, then the digit at bu can serve as BU between ru and lu. If s is a NU, then if s > bu, we have a new BU at s. Else the operation has no effect whatsoever. ∎

*Remark.* From the above argument it is clear that an O or U digit can be toppled at any time without destroying invariants 1 or 2.

The reader will undoubtedly be wondering of how we propose to keep track of all the O, U, BO, and BU digits. It is clear that an operation such as "topple lo" cannot be implemented by a brute force search upwards on the path to find the next O digit, if we want to maintain our O(log $p$) time bound; we must clearly save a link to it. The parenthetical remarks in the above rules serve to indicate what kind of updating may be necessary in a data structure maintaining the positions of the O, U, BO, and BU digits. We have purposefully avoided being very specific on this point in this section, since we will have to deal with this problem in the considerably more difficult and interesting case of maintaining the above invariants on several finger paths simultaneously. This is the topic of the next section.

## 4. Regularity On Multiple Finger Paths

In this section we describe how to maintain $f$ fingers in the key space so that access or modification to the tree a distance of $p$ keys away from a finger can be done with cost $O(f + \log p)$. In our tree there are now $f$ finger paths of interest. We will maintain the invariants of the previous section on all finger paths simultaneously. This presents considerable complications, since the fixup operations that are necessary to maintain the invariants on one path may in fact destroy the invariants on other paths. For example, a segment of the path we are currently working on might be

$$\ldots\ O_2\ \text{NO}\ \text{NO}\ \text{NO}\ \text{B}\ O_1\ \ldots$$

(recall that *left* in this string corresponds to *higher* in the tree). Now some other path may join this one at B, and that B may be the only possible BO below $O_2$ on that path. If we decide to topple $O_1$, transforming B to NO and $O_1$ to B, we maintain the invariants for our path, but in the process we destroy invariant 1 for the path joining at B.

For any insertion or deletion operation in the multiple finger situation, there is a first finger path that will be affected (if any), as the sequence of modifications propagates upwards. (If the first affected node on a finger path is in fact a juncture node for several finger paths, we arbitrarily choose one of them as the first). This first path will be called the *principal path.* If the originator of the modification operation chose wisely the finger on which his operation is based, that finger's path will be the principal path. On the other hand, our algorithms will work no matter from which finger the operation is attempted. The principal path can easily be determined during the course of the search for the key. It is the finger path through the node from which a tagged (finger) link was last traversed during the downward part of the search.

There is another small variation in the multiple finger case. The *relieve* operation need be performed only if the non-brother neighbor involved is on no finger path, i.e. has no tagged links.

We are faced with the question of how fixups on the principal path may affect the invariants on other paths. To handle this issue we impose the following constraint.

Non-Interference Condition. A node which is at the juncture of two or more finger paths cannot be an O or U node. Furthermore, it cannot serve as a BO or BU node for any of the paths involved.

From now on we will assume that the operation when the principal path was encountered was an *Add* 1. The subtraction case is exactly dual. Let t be the first node on the principal path affected by the modifications. We now perform a *toppling sequence* on the principal path path. A toppling sequence consists of toppling the next available O above t until none of the (one or two) digits affected by the topple are juncture nodes with other finger paths. At that point we will say that the toppling sequence *fizzles.* Note that by a remark of the previous section a toppling sequence maintains the invariants on the principle path. We show below that when the toppling sequence has fizzled all our regularity constraints are valid.

Lemma 4.1. A toppling sequence is never longer that $2f$-1 steps.

*Proof:* A fixup or topple operation affects at most two digits on the principal path. Those digits are always the node where the operation is applied and its father. At most $f$-1 nodes on the principal path can be juncture node where other finger

paths join. Thus in at most $2(f-1)+1$ topples the sequence fizzles. ∎

LEMMA 4.2. After a toppling sequence has fizzled, the invariants on all finger paths and the non-interference condition are valid.

*Proof:* Let $\pi$ denote any finger path other that the principal one. We may assume that $\pi$ joins the principal path no higher than the highest digit affected by the last topple. (Otherwise $\pi$ has not been affected). Note that $\pi$ joins below the digit being toppled, since the toppling sequence fizzles at this point. Thus the last topple can be regarded as a topple on $\pi$ also. If $\pi$ joins the principal path at or below t, there is clearly no problem. If $\pi$ joins the principal path above t, then the toppling sequence has quaranteed that up to now there is no O node in the shared portion of $\pi$ and the principal path. The last topple, being wholly in $\pi$, assures us that invariant 1 will now be satisfied on $\pi$. Invariant 2 was never affected. Furthermore, any new B, or B and O digits created by the last topple cannot be at juncture nodes, and therefore the non-interference condition is satisfied. ∎

We next discuss the data structures necessary for maintaining the regularity conditions on the finger paths. Nodes on finger paths that are in states O or U will need to contain some additional information. By duality we confine our attention to O nodes for the moment. For each O node t there is a first O node above it on the path to the root (or none exist). This higher O node will be called t's *successor*. In general t itself will be the successor of several O nodes. These will be said to be the O nodes *visible* below t and will be maintained in a doubly linked linear list, in the order in which they occur in a symmetric order traversal of the tree. An O node t then has the following fields:

| | |
|---|---|
| *successor* | pointer to next O on path to root |
| *lNext* | link to previous O node visible from t's successor (in the linear order) |
| *rNext* | link to next O node visible from t's successor |
| *firstVisible* | leftmost O node visible below t |
| *buffer* | pointer to BO node between t and t.*successor* |

A topple always changes the toppling node t to a B. At that moment the O nodes visible below t will become visible below the next higher O node that existed or has been created on the the way to the root. In an O toppling sequence this will happen repeatedly, and the various chains of visible nodes need to be merged together. The lists of two successive toppling nodes can be merged in O(1) time. Let `t1`, `t2` denote the two toppling nodes (`t2` is higher than `t1`). Then the merge operations consists of replacing `t1` as a node in `t2`'s list by `t1`'s list. We assume that in a toppling sequence this list merging happens as we go along. See figure 4.

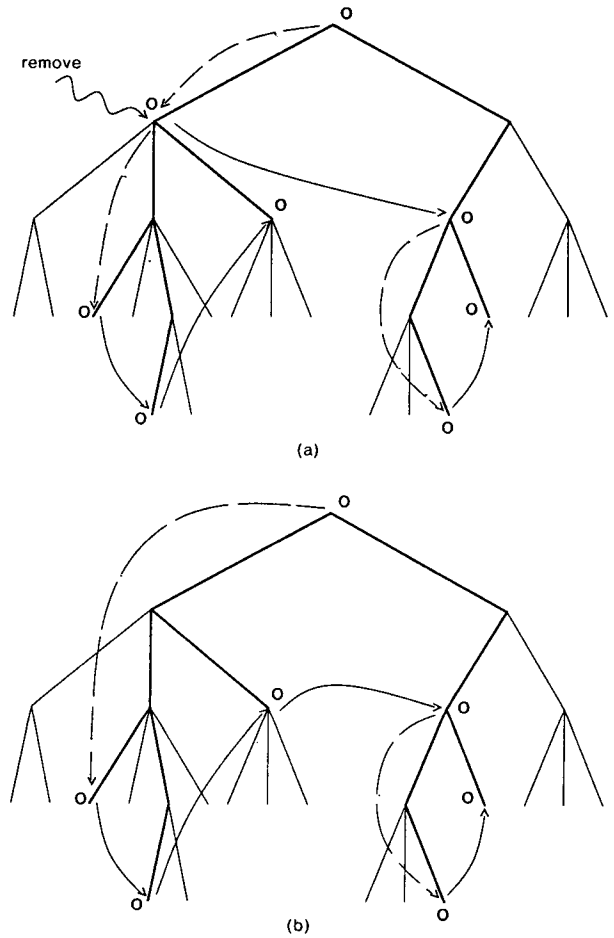



Figure 4. Merging Two Visible Node Lists (showing only forward pointers)
(a) before
(b) after

The reader should refer to the topple O table of the previous section for the discussion that follows. Assume we are partially through the toppling sequence. A new topple operation that removes an O creates a new successor and buffer node for all O nodes on the current list, namely the *successor* and *buffer* nodes of the toppling O. A topple that destoys the old BO digit while creating a new one at the toppling node is similar. This time the successor O for all nodes in the list is still the toppling O's successor, but the new buffer node for the list O's is the toppling node itself. Finally if the topple creates a new O digit, then that O node gets the toppling node's succsessor and buffer, while the nodes in the list get this new O as their successor and the toppling digit as their buffer. We do not perform any of these manipulations, however, until the last

topple has occurred. (Except for list merging, as noted above.) We simply maintain the current candidate buffer and successor nodes through every topple. Since toppling O's can remove U's and/or create new BU's a similar manipulation is done with the U nodes.

```
ToppleInSequence: COMMENT t is the toppling node, s its
    father;

BEGIN CASE toppleType OF

(U O → NU B):       Osuc ← t.successor;
                    Obuf ← t.buffer;
                    Usuc ← s.successor;
                    Ubuf ← s.buffer;

(NU O → B B):       Osuc ← t.successor;
                    Obuf ← t.buffer;
                    IF s is above Ubuf THEN Ubuf ← s;
                    COMMENT recall from section 2 that we
                        can decide if x is a descendant of
                        y in the tree in constant time;

(B O → B B):        Osuc ← t.successor;
                    Obuf ← t.buffer;

(B O → NO B):       Osuc ← t.successor;
                    Obuf ← IF t.buffer = s THEN t ELSE
                        t.buffer;

(NO O → O B):       Osuc ← s;
                    Obuf ← t;

ENDCASE;
<also do list merging>;

COMMENT total cost of this operation is O(1);
END;
```

After performing the above no more than $2f-1$ times the topple sequence has fizzled.

```
FinalTopple: COMMENT do in addition to the above;

BEGIN
    <merge current O node list with those visible below
    Osuc>;
<merge current U node list with those visible below Usuc>;
<set the successor and buffer fields in all nodes on the O list
    that are descendants of s to Osuc and Obuf respectively>;
COMMENT fizzling implies that s and t have the same
    descendants on finger paths;
<set the successor and buffer fields in all nodes on the U list
    that are descendants of s to Usuc and Ubuf respectively>;
COMMENT any two nodes on the same list cannot be
    descendants one of the other -- thus the length of each
    list is bounded by the number of finger paths f, and
    therefore the cosst of the above operations is O(f);
<attach visible O node list to Osuc>;
<attach visible U node list to Usuc>;

IF toppleType = (NO O → O B) THEN
    BEGIN
    s.successor ← t.successor;
    s.buffer ← t.buffer;
    <replace node t by node s in t.successor's visible O list>;
    COMMENT as above this costs O(f);
    END;

COMMENT total cost of this operation is O(f);

END;
```

We have now proved the main result of this paper:

THEOREM 4.1. There is a data structure for maintaining $f$ fingers into the key space so that access and modification a distance of $p$ keys away from one of the fingers can be done in time $O(f + \log p)$.

We end this section with a few words on how new fingers may be added or old fingers may be abandoned. Abandoning a finger can clearly be done simply by traversing the path to the root and untagging all the links until another finger path is encountered. Adding a finger may require, in addition to tagging, a sequence of topple operations, so that the invariants will be valid. It is not hard to see that the invariants will always be satisfied if we start from the new finger leaf and topple any node we encounter on the way to the root, until we hit another finger path. We can now handle matters exactly as above. Thus if our tree has $n$ keys, a finger can be added or abandoned with a one-time worst case cost of $O(\log n)$.

## 5. Notes And Applications

In the previous sections we have endowed each node of our B-tree with a large number of links. This may be an acceptable overhead if we take $m$ large, think of the B-tree nodes as corresponding to pages on the disk, and measure cost by the number of pages we have to access. From a practical viewpoint, however, the overhead of our algorithms is unacceptable in most other situations, particularly if we think of the entire tree as being kept in core and we measure cost by CPU cycles. Of course we set ourselves ambitious aims. We are able to maintain an arbitrary number fingers, and further add and abandon fingers at will with a reasonable cost. Our description was aimed more at conceptual conciseness than amenability to implementation. The more limited requirements of any particular situation can lead to considerable simplifications.

For example, if the fingers are fixed for all time, then we can clearly dispense with the additional links in all nodes except those near finger paths. In any practical situation it is clear that the number of fingers we wish to maintain will be small. We want to have fingers only when we have a high expectation in the locality of references in the key space. Our multiple finger problems may also be compounded by not knowing which finger to use for a given access. It is, however, a very common situation that there is a locale in the key space in which we are particularly interested for accesses and modifications. Such a locale may even be slowly varying with time. For such applications our structure has much to recommend it. An important case is that of maintaining a finger at the left end of the key space. Then we obtain a representation for linear lists such that the following operations take O(log $p$) steps in the worst case, where $p$ is the distance of our record from the beginning of the list

(1) accessing the $p$-th record;
(2) inserting before the $p$-th record;
(3) deleting the $p$-th record.

Clearly records may be searched for either by key or by position in the list. In this case we can completely dispense with all extra links except for father links along the left spine of the tree. Also $m$ can now be taken to be as small as 6 for nodes on the left spine and as small as 3 for other nodes (we need not maintain the same density bounds for all nodes in the tree). The reader is invited to work out this special case as an exercise. A similar solution for this special case using AVL trees as opposed to B-trees is also possible ([5]).

We now present a number of applications.

a. *Priority Queues*

Note that our structure allows us to implement priority queues (or deques) [4, p.149] with the property that deleting the smallest (or largest) element takes constant effort, while the insertion cost is logarithmic. The usual implementations of priority queues using heaps or leftist trees [4, p.150] do not have this property.

b. *Sorting Nearly Ordered Files*

We can advantageously use the data structure presented in Section 3 for sorting, using a variant of *Insertion-Sort* [4, p.81]. To sort a file we insert it sequentially into the list, while maintaining a finger at the high end of the key space. Let us call this Phase 1. In Phase 2 we simply pull out the file in sorted order from the structure, by always deleting the smallest element. For a file of size $n$ the cost of Phase 2 is always $O(n)$. If our file is already relatively in order, then we expect that the cost of Phase 1 will not be too large, as we will often be inserting near the end of the list.

To be more precise: suppose our file is restricted to have no more than $F(n)$ *inversions* [4, p.11]. Let the $i$-th element of the file go to the $c_i$-th position from the rear in the list, when inserted. Then the cost of Phase 1 is $O(\Sigma \log c_i)$. But note that $\Sigma\ c_i$ = number of inversions in the file $\leq F(n)$. From elementary calculus we know that to maximize $O(\Sigma \log c_i) = O(\log \Pi\ c_i)$ subject to the above constraint, we should take all $c_i$ to be equal. It follows that the cost of Phase 1 is bounded by $O(n \log \max\{2, F(n)/n\})$, and this in fact bounds the entire cost of our sorting method. (The max enters the formula since each element costs us at least O(1).)

This implies that if, for example, we restrict ourselves to files with no more than $O(n \log n)$ inversions, then we can sort in worse case time $O(n \log \log n)$. If we have no *a priori* information about the file, then $F(n) = O(n^2)$, and we get the usual $O(n \log n)$ sorting bound [4, p.183].

Furthermore we can show that, in any situation where we have advance knowledge of a bound on the number of inversions, the performance of our algorithm is best possible, to within a constant factor. (We don't need the knowledge in advance of the sort; just in advance of the analysis). We prove this by computing a lower bound on the number of permutations with no more than $F(n)$ inversions. Let $I_n(k)$ denote the number of permutations of $n$ elements with $k$ inversions, and let $p_n(k)$ denote the number of partitions of the integer $k$ into parts not exceeding $n$, and finally let $p(k)$ denote the number of partitions of $k$ (unrestricted). Using generating functions [3, p.15] we prove the following equation:

$$\sum_{0 \leq k \leq t} I_n(k)\ p_n(t-k) \quad = \quad \binom{n+t-1}{t}$$

From this we obtain that the number of permutations with no more than $t$ inversions is bounded below by

$$\binom{n+t-1}{t} \ / \ p(t) \quad .$$

(Since $p(t) \geq p_n(t)$.) Now if we let $t = F(n)$ and use the well-known asymptotic formula [5]

$$p(t) \quad \sim \quad \exp(\pi(2t/3)^{1/2})/4t3^{1/2}$$

we obtain that the logarithm of the number of permutations with at least $F(n)$ inversions is $\Omega(n \log (1 + F(n)/n))$. Combining this with the fact that every element of the permutation must be examined in order to sort it, we have proved:

**THEOREM 5.1.** Our data structure can be used to sort a file with an *a priori* bound on the number of inversions with a cost that is within a costant factor of the optimum possible. If the file has no more than $F(n)$ inversions, then the order of growth of this cost is $\Omega(n \log \max\{2, F(n)/n\})$.

The above counting argument can be strengthened to give us good estimates for the number of permutations with no more than $t$ inversions. Using these estimates, a previous non-constructive result [3] implies the *existence* of a sorting algorithm attaining the asymptotic performance of ours.

We conclude with the suggestion that it may be possible to use our structure to take advantage of other kinds of non-randomness in a file to be sorted.

*Acknowledgement:* The authors wish to thank Prof. D. E. Knuth for suggesting the problem discussed at the beginning of this section.

## 6. REFERENCES

[1] R. Bayer and E. McCreight, "Organization and Maintainance of Large Ordered Indexes", *Acta Informatica* 1 (1972), 173-189

[2] J. Bitner, G Ehrlich, and E. Reingold, "Efficient Generation of Binary Reflected Gray Code and Its Applications", *CACM*, 19, (1976), 517-521

[3] M. Fredman, "Sorting $X + Y$ and Building Balanced Search Trees", *Proc. 7-th Ann. ACM Symp. Theor. Comp* (1975), 240-244

[4] D. Knuth, *The Art of Computer Programming*, vol. III, Sorting and Searching, Addison-Wesley (1975)

[5] D. Knuth, Unpublished CS204 Lecture Notes, *Stanford University*, 1976. (See also *The State of the Art of Computer Programming*, STAN-CS-76-551, #292 and #358).

[6] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall (1975), 246-257

[7] H. Rademacher, "On the Partition Function $p(n)$", *Proc. London Math. Soc.* 43, 241-254