MATHEMATICAL CENTRE TRACTS 81, 1976, 37-147

THE COMPLEXITY OF DATA ORGANIZATION

J. VAN LEEUWEN *
State University of New York, Buffalo, USA

0. INTRODUCTION

0.1. The way *data* is presented to the computer and represented in the system can be a dominating factor in the performance of the computer as a data-processing tool.

By its very nature the area of data-organisation extends from the study of combinatorial algorithms to the design of information management systems, and it is too broad to be conveyed in a series of six lectures only.

We decided to emphasize a few topics in concrete complexity theory which have recently received much attention and which have lead to a number of fundamental techniques and new results which may be applicable to many problems which computer-programmers encounter.

0.2. The present lectures neither emphasize (say) a specific domain in computational complexity nor give you the design-philosophy of relational database, but instead we will concentrate on useful *techniques* (tricks?) in data-organisation which can bring improvements in many computer-programs performing data-manipulations of some sort.

We have specifically avoided to present results which are already adequately treated in such excellent texts as Knuth [43],[44] or Aho, Hopcroft, and Ullman [3].

The reader is assumed to have some familiarity with computer-programming and the fundamental information-structures as discussed in Knuth [43] or Wirth [66]. Roberts [54] gives a useful survey of the basic file-organisation techniques.

---

*) Present address: Dept. of Computer Science, Whitmore Laboratory, the Pennsylvania State University, University Park, Pennsylvania 16802.

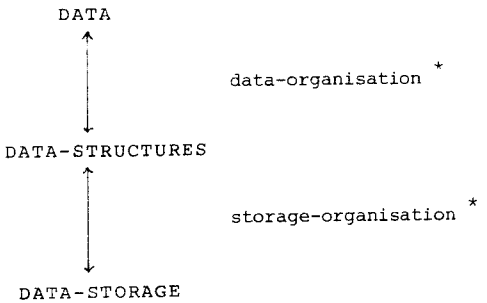


fig.1. Some terminology

* Sometimes called "file-organisation" and "data-organisation" respectively.

1. EFFICIENCY VERSUS DATA-REPRESENTATION

1.1. Let us start from an intuitive concept of *data*. The user is largely responsible for *data-collection*, and must make sure that he gathers the information needed for a successful data-processing system.

1.2. To facilitate *retrieval* and efficient *manipulation* information should not be stored randomly, but is preferably organised in structures which allow for easy acess and (say) deletion and addition of data at all times.

1.3. Computer storage is divided into directly addressable *core-storage* ("main memory") and "supplementary" or *secondary storage* ("auxiliary memory"). Data is usually stored in auxiliary memory on mass-storage devices like magnetic tapes, magnetic drums, (magnetic) cards, data-cells, or disc-packs, and only small parts will be held for processing in main memory at a time. In time-shared systems the data will necessarily be segmented, with due restrictions on size per segment.

1.4. The user is not likely to be concerned with the "hardware" of a data management system, but is using an intermediate language instead. This can be a "host"-language (i.e. some general purpose programming language) providing helpful primitive data-structures and a flexible mode-definition mechanism for creating more complex structures, or a special *data-language*

Data-structures are merely the model of data-storage as observed by the user. The "system" presumably interpretes his model in "real" storage.

1.5. Information is usually provided in small packages of logically connected data called *records*. The *fields* of a record must be specified, and may consist of other records. The information in a record can be accessed by using the *field names* as *selectors*.

1.6. Some fields may be used (or added) for uniquely identifying a record. The contents of such fields together form the *primary key* of a record.

Later we shall simply identify a record and its primary key (as it is the only part our algorithms will be using), and define it to be *an ordered k-tuple* $(b_1,b_2,\ldots,b_k)$ *of values* taken from some ordered set R.

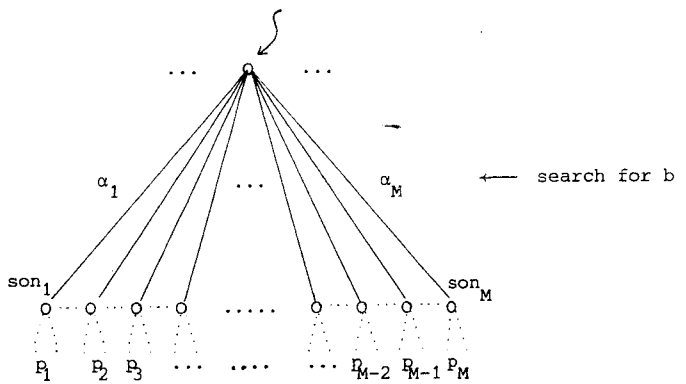
1.7. Consider an arbitrary, unstructured collection of (perhaps) randomly stored records

$$V = \{v_1, v_2, \ldots, v_N\} \subseteq R^k.$$

Numerous *address-calculation schemes* exist for "finding" a record when its primary key is presented (Knuth [44]).

1.8. The simplest method would be to interprete keys as *indices*, and to store the addresses in a *table* (fig. 2). *Sequential search* is easy to program, but will require an average of N/2 probes. If keys were sorted in *lexicographic order*, then presumably some kind of binary search over "outgoing" edges for each successive component of the key would reduce the search-time to ~ log N (at worst) *per component*. Fredman [31] indicated how an information-theoretic argument can help to improve it further. (The proof we develop here seems new.)

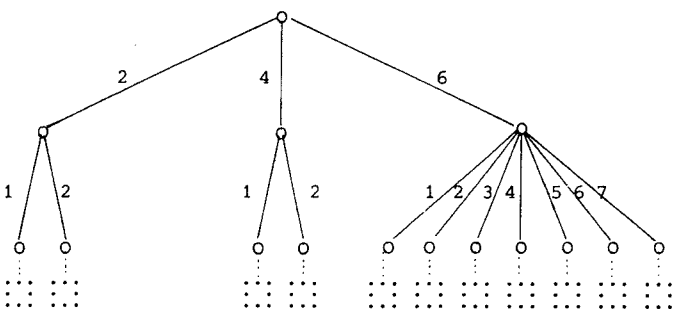
1.9. Consider an arbitrary node reached in progressing down the tree



$$\alpha_1 < \alpha_2 < \ldots < \alpha_M$$

2	1
2	2
4	1
4	2
6	1
6	2
6	3
6	4
6	5
6	6
6	7

fig.2. A table and a (lexicographic) tree



Suppose "son_i" has p_i descendant leaves, $p_i \in \mathbb{N}$. In searching for an i such that $\alpha_i = b$ (sending us off to son_i), it is useful in probing to emphasize more frequently occurring α's.

The proper formulation makes use of a binary *search-tree*: each node in such a tree carries a query

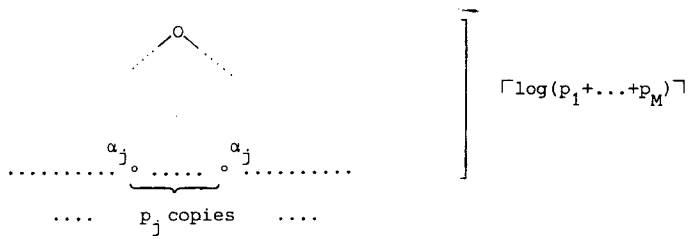
```
if b ≤ X   then  Y  else Z

      ↑          ↑       ↑
      ⋮          ⋮       ⋮
  some α-value   ⋮       ⋮
                 ⋮       ⋮
                 ⋮       ⋮
                 :.......go left,
                         go right, or
                         "is son_i"      (found it !)
```

<u>LEMMA</u>. *There exists a search-tree for finding* b *in*

$$\leq \lceil \log(p_1+\ldots+p_M) \rceil - [\log(p_i+1)] + 1 \text{ steps,}$$

where i *is such that* $b = \alpha_i$.

Proof. Make p_j copies of α_j (for each $j = 1,\ldots,M$), and build a binary tree of minimum path-length



Call a node j-*critical* if and only if all its descendant leaves carry α_j and there is no node less deep in the tree with that property. For each j these are one or two j-critical nodes.

Make it a search-tree by assigning appropriate queries *topdown* such that the search for some b (presumably $= \alpha_i$) is always directed to an

i-critical node. All nodes that are irrelevant now (thus, not on a search-path to some critical node) may be *purged*, after which the tree may have to be condensed back to binary form (thus reducing the search-length of some paths even further).

The identity of b is uncovered at the father of a critical node (and it obviously is the fastest way of identifying it).

The height of the "father" is bounded by the smallest t such that $d_t = 0$ in the following recurrence

$$d_0 = p_i$$

$$d_s = \left[\frac{d_{s-1}-1}{2}\right], \qquad \text{for } s = 1,2,\ldots$$

which is $t = [\log(p_i+1)]$. Thus a decision is found after $\lceil \log(p_1+\ldots+p_M) \rceil - [\log(p_i+1)] + 1$ steps. □

1.10. The resulting search-tree (after all unnecessary information has been purged) takes only little more space than was needed at the node of 1.9 anyway, and should be substituted. The original lexicographic tree becomes a cascade of local search-trees, and it enables one to find the address of a record $(b_1,b_2,\ldots,b_k)$ quickly.

<u>THEOREM</u>. *A record can be identified within* $\sim \log N + 2k$ *queries.*

Proof. Follow the trail of identifying $(b_1,\ldots,b_k)$ (fig. 3).

By lemma 1.9 the total number of queries needed is bounded by

$$\sum_{0}^{k-1} (\lceil \log N_i \rceil - [\log(N_{i+1}+1)] + 1) =$$

$$= \lceil \log N_0 \rceil + \sum_{1}^{k-1} (\lceil \log N_i \rceil - [\log(N_i+1)]) - 1 + k$$

$$\cong \log N + 2k. \qquad \square$$

Recall that the original, unbalanced search-strategy did cost us about $\sim \log N$ queries *per component*.

1.11. Large collections of logically related records are hardly ever stored

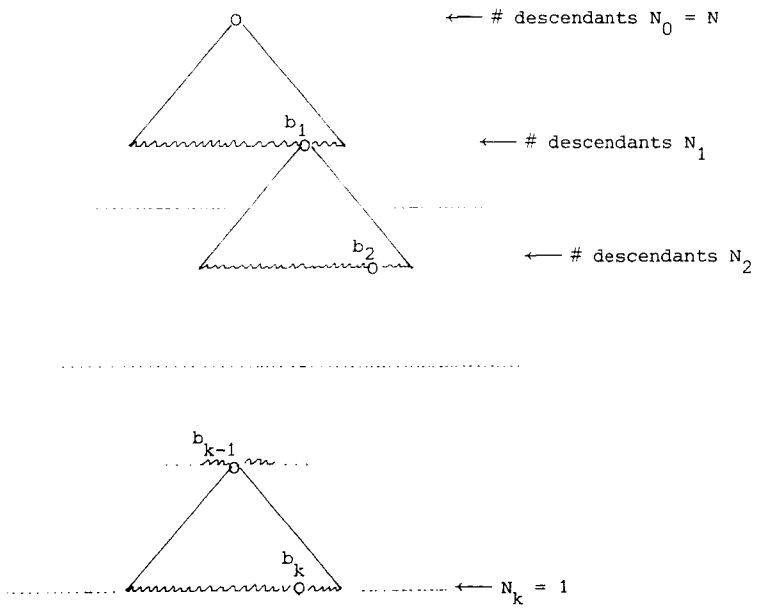


fig.3. Cascade

randomly, and are normally arranged in one or more perhaps specifically structured units called *files*. The reason for *file-design* (or *file-organisation*) is to store data appropriately on available storage-media to allow for efficient handling of the operations which the user requests.

1.12. In modern terminology a collection of stored operational files used by the application systems of some particular enterprise is called a *database* (Engles [25]).

1.13. Files can be distinguished by the supporting retrieval mechanism. The basic types are

(i) *sequential files*

- A record can be accessed only by scanning the entire file for the point where it is stored. It is common to *batch* successive record-

requests, and to avoid the need for rewinding the file for each individual request.

(ii) *random access files*

- The address of a record is determined from its primary key with a simple (perhaps tabulated) function, and access to its location follows (almost) immediately.

(iii) *indexed (sequential) files*

- A derived key is used as index in a directory structure to yield the *bucket* where a record must be, and ordinary retrieval from the bucket follows.

1.14. An important task in programming consists of finding the best data-organisation for an application in order that the data can be processed most efficiently.

However, the data may not be available in most desirable form or may be of much larger volume than can fit in core, and it can be equally important to modify algorithms for an application to work better and better under *given* constraints on the data-organisation.

These two directions of research together form the domain of *the complexity-theory of data organisation*.

1.15. An interesting example to show how a given data organisation can affect the choice of an algorithm is *Warren's transitive closure algorithm* for 0 - 1 matrices (Warren [63]).

We shall first consider the more traditional algorithm of Warshall (Warshall [64]) for this task.

1.16. An $n \times n$ 0 - 1 matrix $M = (m_{ij})$ can always be interpreted as the adjacency-matrix of a directed graph G_M on $\{1,\ldots,n\}$ where

$$m_{ij} = 1 \longleftrightarrow i \circ \longrightarrow \circ\, j$$

The 0 - 1 matrix $M^* = (m^*_{ij})$ is called the *transitive closure* of M if and only if its coefficients satisfy

$$m^*_{ij} = 1 \longleftrightarrow \text{there is a path } i \to \ldots \to j \text{ (of length } \geq 0) \text{ in}$$

1.17. Most transitive closure algorithms can be distinguished by the stepwise manner in which all paths $i \to \ldots \to j$ are built up. One only needs to consider paths free of repeating nodes.

1.18. In Warshall's algorithm one builds M^* in stages $M^*(s)$ where

$$m^*_{ij}(s) = 1 \longleftrightarrow \text{there is a path } \underbrace{i \to \ldots\ldots \to j}_{\in \{1,\ldots,s\}} \text{ (of length } \geq 0) \text{ in } G_M .$$

1.19. Obviously $M^*(0) = M$ and $M^*(n) = M^*$.

1.20. For $s = 1,\ldots,n$ one can construct $M^*(s)$ from the previous stage by observing that

$$m^*_{ij}(s) = m^*_{ij}(s-1) \vee \underbrace{m^*_{is}(s-1)}_{\text{fixed factor for row } i} \cdot m^*_{sj}(s-1)$$

Denoting the i-th row of $M^*(s)$ by $M^*_i(s)$ we obtain the rule

$$M^*_i(s) := \underline{\text{if}}\ m^*_{is}(s-1)\ \underline{\text{then}}\ M^*_i(s-1) \vee M^*_s(s-1)\ \underline{\text{else}}\ M^*_i(s-1).$$

The algorithm can now be formulated as

```
"let M* be M";
for s := 1 to n do
for i := 1 to n do
M*_i := if m*_is then M*_i ∨ M*_s
od
od;
```

1.21. Warshall's algorithm works very well if each row of the matrix can

be *packed* in a single word and words can be "or"-ed directly.

As a contrast it is of interest to evaluate the algorithm for (very) large matrices which are stored externally in a sequential file. (It also serves as an adequate model for a paging environment).

Let consecutive records correspond to consecutive rows of the matrix, and assume that one can store ~ 2 records (at least) in memory at a time.

We shall use the number of records "paged in" by the algorithm as a measure for its complexity.

1.22. The algorithm must now be formulated as

```
rewind;
for s := 1 to n do
scan for record s;
M*_s := get record;
rewind;
for i := 1 to n do
M*_i := get record
if m*_is then reset record;
            M*_i := M*_i ∨ M_s;
            put M*_i
fi
od;
rewind
od;
```

and it follows that $\sim n^2 + n$ records must be read into memory.

1.23. With the given (sequential) organisation of the data Warshall's algorithm immediately becomes less attractive because the need for the coefficient m^*_{is} forces one to read in *every* record even if there is no subsequent action on the record.

In an algorithm recently proposed by Warren ([63]) this has been eliminated, and records are paged in only when necessary. (Our proof seems to be new.)

1.24. In Warren's algorithm M^* is built up in *two* passes.

Pass I yields an intermediate matrix $M^{\otimes}$ in stages $M^{\otimes}(s)$ where

$$m^{\otimes}_{ij}(s) = 1 \longleftrightarrow \text{there is a path}$$

$$\underbrace{i \to \ldots\ldots\ldots \to j}_{\in\ \{1,\ldots,i-1\}}$$

for all $i \le s$

and pass II subsequently yields M^* in stages $M^*(s)$ where

$$m^{*}_{ij}(s) = 1 \longleftrightarrow \text{there is a path}$$

$$\underbrace{i \overset{\otimes}{\to} \ldots\ldots\ldots \overset{\otimes}{\to} j}_{\in\ \{i+1,\ldots,n\}}$$

for all $i \le s$.

1.25. Obviously $M^{\otimes}(1) = M$, $M^*(0) = M^{\otimes}(n)$, and $M^*(n-1) = M^*$.

For $s = 2,\ldots,n$ one can construct $M^{\otimes}(s)$ from the previous stage by updating row s. One can similarly construct $M^*(s)$ from the previous stage for $s = 1,\ldots,n-1$.

1.26. To compute $m^{\otimes}_{sj}(s)$ it is helpful to conceive of intermediate coefficients $m^{\otimes}_{sj}(i,s)$ where

$$m^{\otimes}_{sj}(i,s) = 1 \longleftrightarrow \text{there is a path}$$

$$\underbrace{s \to \ldots\ldots\ldots \to j}_{\in\ \{1,\ldots,i\}}$$

$(0 \le i < s)$.

Obviously $m^{\otimes}_{sj}(0,s) = m^{\otimes}_{sj}(s-1)$, and the values of $m^{\otimes}_{sj}(i,s)$ for $i = 1,\ldots,s-1$ can be computed using the rule

$$m^{\otimes}_{sj}(i,s) = m^{\otimes}_{sj}(i-1,s) \vee m^{\otimes}_{si}(i-1,s) \cdot m^{\otimes}_{ij}(s-1)$$

fixed factor for the row.

Denoting the s-th row of $M^{\otimes}(s)$ by $M^{\otimes}_s(s)$ we obtain the iteration


```
for i := 1 to s-1 do
if m⊗si(s) then M⊗s(s) := M⊗s(s) ∨ M⊗i(s-1)
fi
od
```

(where $M^{\otimes}_i(s-1) \equiv M^{\otimes}_i(s)$).

1.27. To compute $m^*_{sj}(s)$ it is similarly helpful to conceive of intermediate coefficients $m^*_{sj}(i,s)$, where

$$m^*_{sj}(i,s) = 1 \longleftrightarrow \text{there is a path}$$

$$\underbrace{s \overset{\otimes}{\to} \ldots\ldots\ldots \overset{\otimes}{\to} j}_{\in\ \{s+1,\ldots,i\}}$$

$(s \le i \le n)$.

Obviously $m^*_{sj}(s,s) = m^{\otimes}_{sj}(n)$, and the values of $m^*_{sj}(i,s)$ for $i = s+1, \ldots,n$ can be computed using the rule

$$m^*_{sj}(i,s) = m^*_{sj}(i-1,s) \vee m^*_{si}(i-1,s) \cdot m^*_{ij}(s-1)$$

fixed factor for the row.

We can therefore obtain $m^*_{sj}(s)$ from the iteration

```
for i := s+1 to n do
if m*si(s) = 1 then M*s(s) := M*s(s) ∨ M*i(s-1)
fi
od                    (where M*i(s-1) ≡ M*i(s) ≡ M*i(0)).
```

1.28. The entire algorithm can now be formulated as

```
rewind;
scan for record 2;
for s := 2 to n do
M*s := get record;
rewind;
for i := 1 to s - 1 do
```

```
        if m*_si then M*_i := get record;
                      M*_s := M*_s ∨ M*_i
             else scan for next record
        fi
        od;
        put M*_s
        od;
        rewind;
        for s := 1 to n - 1 do
        M*_s := get record
        for i := s + 1 to n do
        if m*_si then M*_i := get record;
                      M*_s := M*_s ∨ M*_i
             else scan for next record
        fi
        od;
        rewind;
        scan for record s;
        put M*_s
        od;
```

1.29. The number of records "paged in" by Warren's algorithm is bounded by

$$2(n-1) + \#\text{off-diagonal ones in } M^*$$
$$\sim n^2 + n - \#\text{zero's in } M^*$$

which shows that *it behaves better than Warshall's the more sparse* M^* *is.*

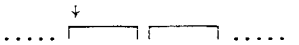
1.30. It follows that the design of good algorithms which use a pre-determined data-organisation can lead to non-trivial questions which the theory must consider just as well, and in the complexity-theory of data organisation one has to pay attention to the trade-offs between new data-structures and new algorithms.

2. FILE-MERGING

2.1. Sequential files are *tape-like.*

We shall assume that all records have equal size and that records are stored with small, detectable intermissions

2.2. There is a *file-pointer*, normally positioned at the beginning of a record

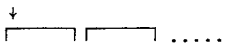


2.3. We shall later consider essentially random access files which allow an arbitrary (but per application bounded) number of pointers.

2.4. Any general purpose programming language provides some or all of the following operations on a sequential file:

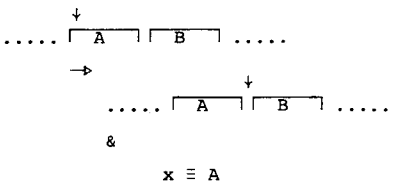
rewind

position the file-pointer back to the beginning of the tape



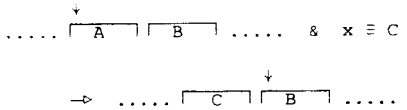
x := get record

the contents of the record currently pointed at is read into record-variable x and the pointer is advanced to the beginning of the next record



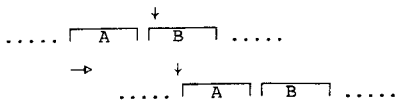
put x

the contents of record-variable x is stored into the current record-position on the tape. *The operation is usually allowed only if the pointer is located at the end of the file*, otherwise we assume it simply overwrites the previous record contents



reset record

positions the pointer at the beginning of the immediately preceeding record (and void at the beginning of the tape)



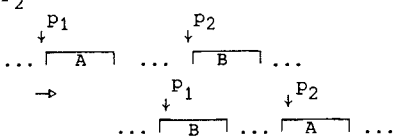
and occasionally one may want to use derived operations like

```
scan for b
        repeat
            x := get record
        until eof ∨ x.key = b;
```

(or an equivalent interpretation of it).

2.5. On files with more than one pointer we shall in fact allow primitive operations such as

swap (p_1,p_2)



and perhaps a boolean-valued instruction

ordered (p_1,p_2)

yielding <u>true</u> iff p_1 is before p_2 and p_1.key $\leq$ p_2.key.

EXAMPLE. A random access file

with n records can be inverted in linear time by

```
while p2 > p1 do swap (p1,p2);
                 p1 := p1 + s;
                 p2 := p2 - s
              od;
```

(where s is the record size).

2.6. A (sequential) file is said to be *ordered* if and only if the keys of consecutive records form a nondecreasing chain.

2.7. The process of combining two ordered (sequential) files into a single ordered (sequential) file is called *merging*.

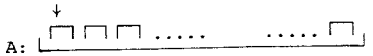
A merging procedure is said to be *stable* if and only if it leaves the relative order of records with the same key in the original files unchanged. (See Knuth [44].)

2.8. It is now reasonably well-understood how to merge two files in case some auxiliary tapes are available (Wirth [66]). It is less obvious how to proceed with minimal storage-requirements, and in particular how to do a linear time merge without the use of an unbounded number of (perhaps hidden) pointers or links.

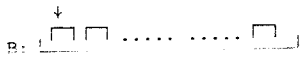
This problem was first cited in Knuth ([44], p.388) and solved in 1974 by Horvath [40] and by Luis Trabb Pardo [58].

To acquire a better appreciation for the specific constraints of the problem we shall first consider a linear time two tape merge-procedure of Floyd & Smith [28].

2.9. Suppose we are given two ordered sequential files of records of a certain size s



N records, but perhaps
only $n \le N$ with a distinct key.



M records, but perhaps
only $m \le M$ with a distinct key.

We shall assume very little about s, but only require that it provides enough bits for the distinct keys:

$$s \ge \log(n+m).$$

2.10. The idea is to copy file A (at the end of B) and file B (at the end of A) and insert the proper sequence numbers in intermediate records first, showing in what order the file-members must be assembled.

An immediate problem shows up if we want to perform this in linear time: the sequence-numbers for file A (and similarly for file B) will occupy

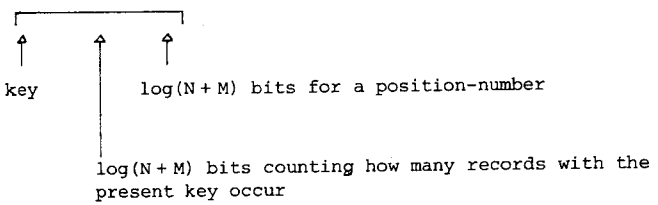
$$\sim N \cdot \log(N+M) \text{ in tape-length}$$

and this may not at all be bounded in terms of the original file-length

$$\sim N \cdot s$$

which could be as little as $N \cdot \log(n+m)$.

2.11. Instead of complete sequence-numbers we shall therefore insert intermediate records once for each distinct key only (a common trick in *data-reduction*), but augment the information per record to



This will require an extra space of only

$$\sim n\,(s + 2\log(N+M))$$

$$\le N \cdot s + 2n\log(N+M)$$

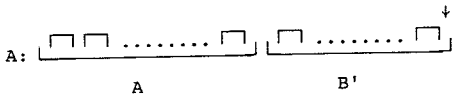
$$\le N \cdot s + \gamma' \cdot (N+M)\log n$$

$$\le \gamma \cdot (Ns + Ms)$$

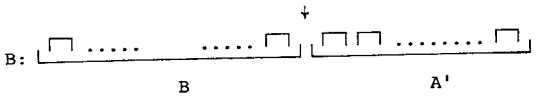
for tape A (and similarly for tape B), which is now bounded by a constant factor in the length of the original files.

2.12. The first step of the algorithm will copy A and insert occurrence-numbers for all distinct keys.

After a similar procedure is applied to copy tape B into B' at the end of tape A we have



containing $M + m$ records.



containing $N + n$ records

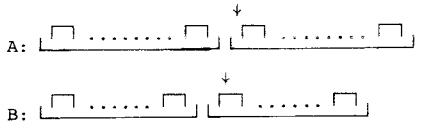
and the time required so far is

$$\underset{\text{rewind}}{\sim Ns + Ms} + \underset{\text{scan B}}{Ms} + \underset{\text{scan + copy A}}{3Ns} + \underset{\text{copy onto A'}}{Ns + \gamma(Ns+Ms) + Ns} +$$

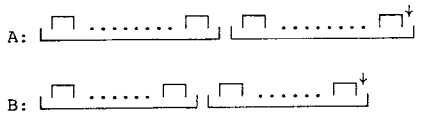
$$+ \underset{\text{rewind over A' and B}}{\gamma(Ns+Ms) + Ms} + \underset{\text{scan + copy B}}{3Ms} + \underset{\text{copy onto B'}}{Ms + \gamma(Ns+Ms)}$$

$$\sim (6+3\gamma)\,Ns + (7+3\gamma)\,Ms.$$

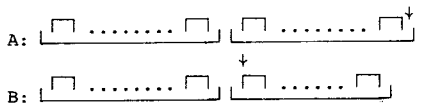
2.13. Rewind over B'



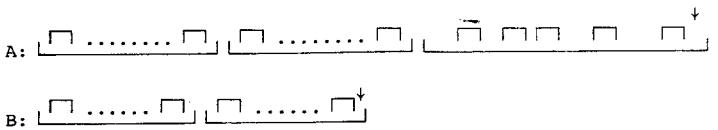
and perform a "dummy" merge of A and B by looking at the intermediate, accounting records only and inserting the appropriate position-number for the first record of each distinct key in the proper field.



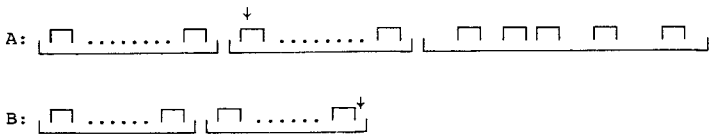
Rewind over A'



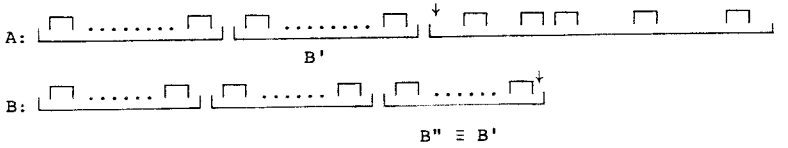
and copy the records of A (taken from A') at the end of tape A in the proper relative final position (which we can determine from the information in the accounting records which should *not* be copied over)



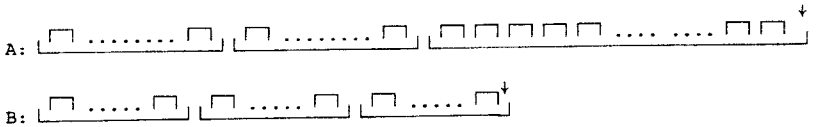
Rewind tape A to the beginning of B'



and copy all of B' onto tape B:



Finally, rewind over B" and copy the records of B (taken from B") into their final positions on tape A (again using the information from the accounting records)



The merged version of the original files appears as the last block on tape A.

2.14. We have now obtained the essential result of Floyd & Smith.

THEOREM. *The two-tape merge-procedure is stable and requires time linear in the length of the merged files.*

Proof. Stability follows from the construction. The time required for all successive stages adds up to

$$\sim (18 + 13\gamma)(Ns + Ms). \qquad \square$$

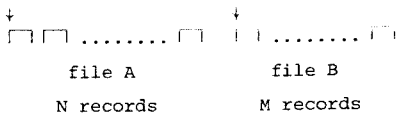
2.15. Note that the algorithm requires no third tape, and we could have used the given space even more economically had we overwritten parts which were at some stage in the algorithm not needed anymore. We seem to be close to an answer of Knuth's problem, but there is apparently no way to eliminat the need for all accounting records without thereby causing a non-linear increase of time.

It doesn't seem to be easier for random access files either, and therefore the following result of Luis Trabb Pardo [58] is of interest.

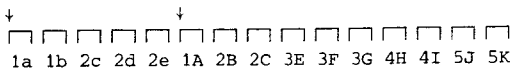
THEOREM. *There is an algorithm to merge two "rigid" random access files in a stable way in linear time and no auxiliary storage using only a bounded number of pointers.*

2.16. Trabb Pardo's algorithm contains a number of interesting contributions which we shall present in a simplified form to better demonstrate the ideas.

Assume that the files are stored as one chunk

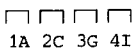


EXAMPLE.



2.17. A *block* of consecutive records is called an *internal buffer* if and only if it is fully ordered and contains only records with a distinct key.

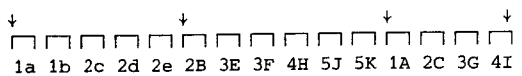
EXAMPLE.



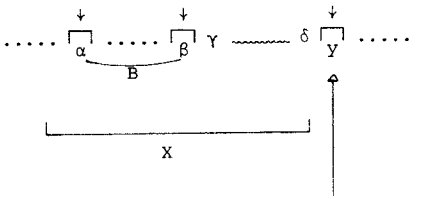
2.18. The first phase consists of *extracting* an internal buffer of size $\sim \sqrt{N+M}$ from the given files (which we always can if there are enough distinct keys).

The buffer-elements can be collected by scanning the files from left to right, picking a new larger record only if it is the *last* sample with the current key in the row (to ensure stability later).

EXAMPLE



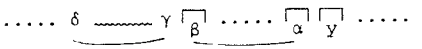
The buffer-elements are kept together in a "growing" block B sliding steadily to the right, while a pointer advances to search for a next candidate



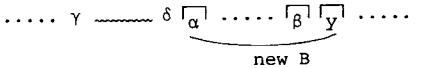
suppose this record must be assembled next.

Then:

reverse X



and reverse the marked parts



and we can proceed until B is full size. B is then exchanged in the same way to the end of the file.

The time required for this phase is bounded by

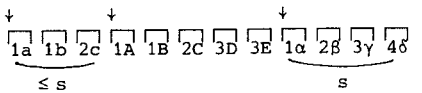
$$\sim N + M \quad + \quad \left(\sqrt{N+M}\right)^2$$

each stretch of elements can be involved in reversion once

but the elements of the buffer are involved in it perhaps as many times as its number of elements.

2.19. It could very well happen that file A is "small" with respect to $s = \sqrt{N+M}$.

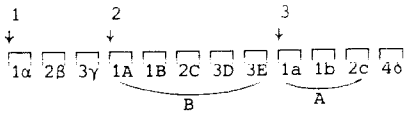
EXAMPLE.



(If it is so for B we do the next procedure "backwards".)

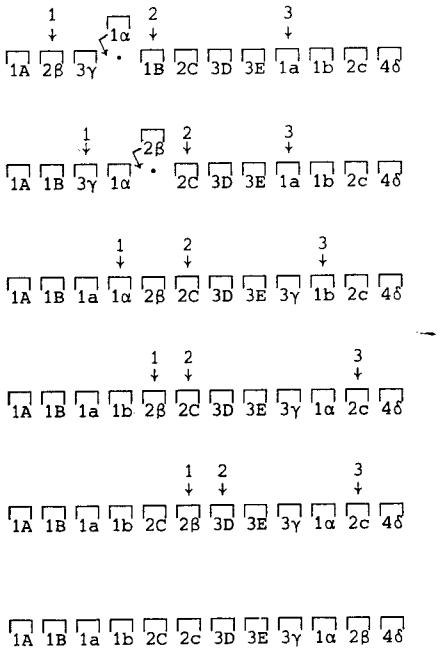
Exchange A with the first part of the buffer (in linear time using the reversion-trick):

EXAMPLE.



and merge B and A into place in a *stable* manner by moving pointers 2 and 3 steadily rightwards, each time exchanging the smallest with the (buffer-) record under 1.

EXAMPLE.



The buffer-records act as "dummies" which keep making place for records merged into position, and in the end they recollect in the buffer-zone.

Sort the buffer back into order, and complete the procedure (in linear time) by merging the buffer back into the file (how to do it will

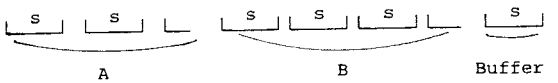
follow later).

It is now clear why an internal buffer is essential: it is a group of records which we can move around to provide a space where needed and which we can bring back into stable order by just sorting!

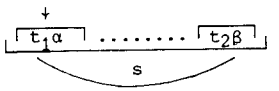
(Another application of the buffer will follow next.)

2.20. Let us now assume that both A and B are "long". Think of A and B as divided into blocks of size s.

EXAMPLE.



It is convenient to have a notation for the first and last key of a block of known size:



X

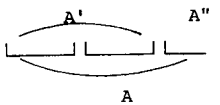
$\text{first}(X) = t_1$

$\text{last}(X) = t_2$

The values can be picked up using only two probes into the block.

2.21. If the size of A is no integer multiple of s

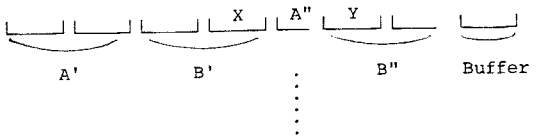
EXAMPLE.



then get rid off the left-over part A" by merge-exchanging it (as a block) into B as far right as possible, i.e., after the *last* block X in B for which

$$\text{last}(X) < \text{first}(A").$$

EXAMPLE.



$$\text{last}(X) < \text{first}(A'')$$
$$\text{first}(A'') \le \text{last}(Y).$$

X can be found by a simple scan, and using the reversion-trick the exchange follows in linear time.

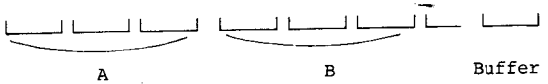
The task is now reduced to

- merge A' and B'
- merge A" and B"

(where the last part follows as in 2.19). Note that the records will appear in final position if we do the merge in place.

2.22. Without loss of generality we can therefore assume that A has a neat block-arrangement.

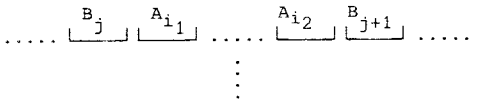
EXAMPLE.



Let the blocks of A and B be $A_1,\ldots,A_k$ and $B_1,\ldots,B_\ell$. Since $s \sim \sqrt{N+M}$, $k+\ell \le s$.

2.23. The idea is to first "merge" the A-blocks (as a block) in between the B-blocks as far to the right as possible while preserving stable order.

EXAMPLE.



$$\text{last}(B_j) < \text{first}(A_{i_1})$$
$$\text{first}(A_{i_2}) \le \text{last}(B_{j+i}).$$

In order to find the proper order of all blocks we shall first do a "dummy" pass and "assign position-numbers" (as in the Floyd & Smith algorithm).

The trick that makes it work in no extra space is to *use the increasingly ordered buffer-records as "position-numbers"* which we simply exchange into the first position of a block while temporarily storing the original first record of the block into the buffer-position corresponding to its position in the final order.

2.24. In this way we reduced "merging" blocks to a *sort* on the first element of each block. Use a simple straight insertion method.

Swapping each next "minimal" block into place (by a record-wise exchange) takes s steps, and searching for the next, i-th minimal block may take up to $k+\ell-i$ probes. The total time is bounded by

$$\sim (k+\ell)s + \sum_{1}^{k+\ell} (k+\ell-i) \sim s^2 \sim N+M$$

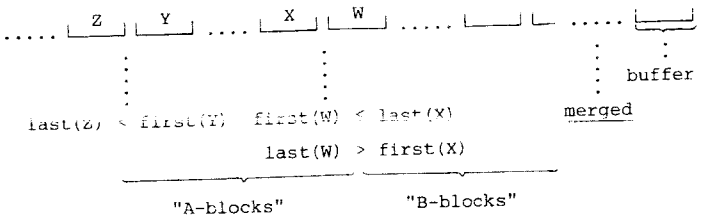
and is therefore linear.

Finally, in a left-to-right scan we exchange the original first records back into the blocks while the buffer is automatically restored at the same time.

2.25. It seems that we only need to re-hash neighbouring blocks somewhat to get records in the proper place (finally). We shall do so from right-to-left.

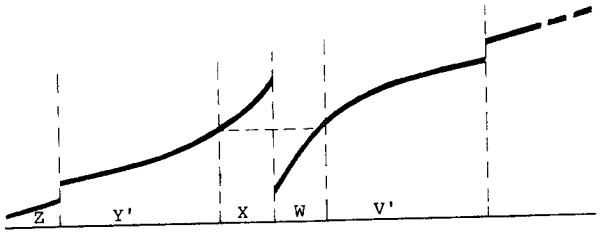
We didn't "mark" what the A- and B-blocks were (it doesn't matter as long as we merge correctly), but one can recover the partition by inspecting the order-relation between the first and last record of blocks while scanning to the left.

EXAMPLE.



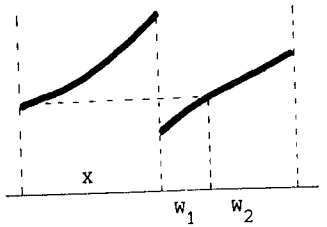
The situation can be recovered in time linear in the length of the stretch of elements involved. Note the L-block that may drag along. A graph helps to show the precise order-relation between the parts.

EXAMPLE.



2.26. Consider XW

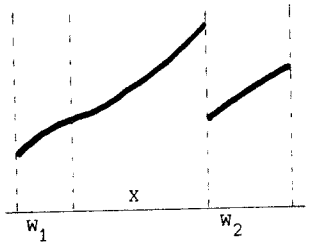
EXAMPLE.



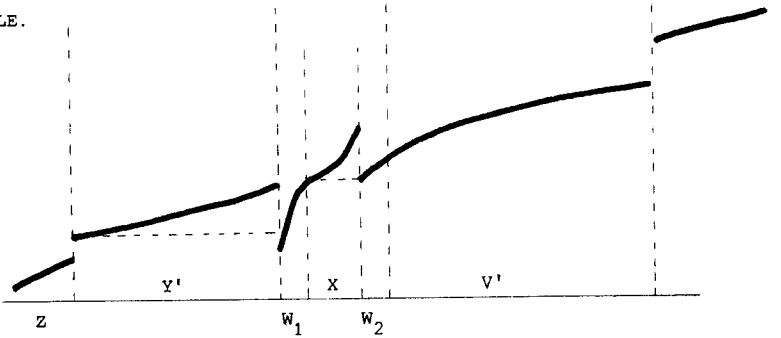
and use the reversion-trick to exchange X (as a block) as far to the right into W as possible.

EXAMPLE.

yielding

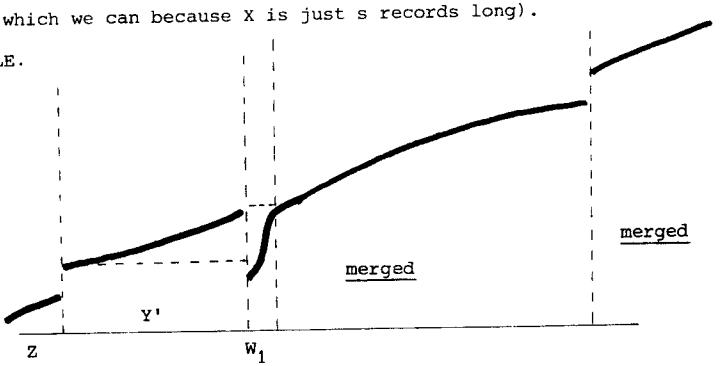


EXAMPLE.



2.27. Now *merge* X *into* W_2V' in a stable manner *using the buffer-trick from* 2.19 (which we can because X is just s records long).

EXAMPLE.

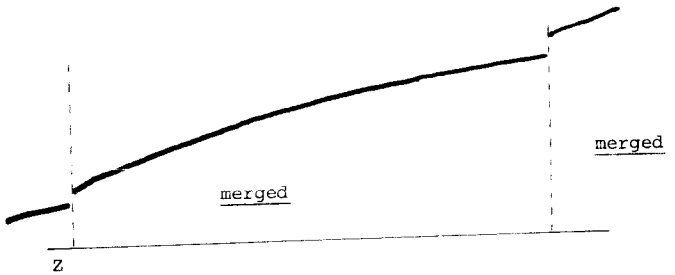


and we finished off "half" of the zone in time just linear in the number of elements involved.

Don't resort the buffer again each time we do such a phase.

With the same technique one can merge Y' and W_1 in a stable manner and "finish" the entire zone.

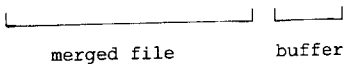
EXAMPLE.



Proceed left and locate the next zone, thus continuing and completing the merge in linear time and a bounded number of pointers.

2.28. In the last phase of the algorithm one should resort the buffer (in time $\sim s^2$).

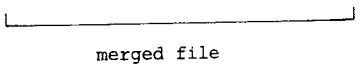
EXAMPLE.



and insert the buffer back into the file again, leaving each buffer-record at the far right of the stretch of file-records with the same key to ensure final stability.

With the reversion-trick one can steadily slide the buffer to the left and absorp it in linear time and no extra space.

EXAMPLE.



2.29. This completes the essential part of Trabb Pardo's minimal requirement stable merge procedure. (What should happen if no full size buffer can be extracted is left to the reader, but all necessary techniques to use have been presented.) Several steps were due to Kronrod and to Horvath.

2.30. Note that an "in-place" linear time merge procedure immediately yields

an "in-place" and minimal requirement $O(n \log n)$ sorting method by merging the data into adjacent blocks of a size which is an increasing power of two.

3. TABLES AND BALANCED TREES

3.1. The choice of a proper data-organisation for a task depends on how the user wants to operate on the data.

In the design-philosophy of programming languages there is a tendency to make *data-structuring* more and more automatic. Ultimately the user should be able to specify the *axioms* and *rules* for the operations on the data in some formal language, and the system should determine a consistent (and preferably efficient) internal representation. This has been studied theoretically (see Spitzen & Wegbreit [55]), and it is anticipated for PASCAL in terms of graph-specification primitives (Gehani [33]).

3.2. Perhaps the simplest requirement on a (random access) file or *table* is that one can perform

find (k)
- locate the record with key k -

insert (k)
- put the record with key k into the table -

for all keys quickly.

Since the universe of key-values may be much larger than the set of keys actually occurring in an application, an efficient and easy computable *key-to-address transformation* should presumably "chop down" a key into an index in a small *hash-table* where the record-address is (or will be) listed.

Such *hash-functions* h(k) are almost by definition many-to-one, and it may happen that different keys are mapped onto the same *hash-address*.

The anomaly in insertion occurring when a hash-address is already occupied is called *collision*.

3.3. Collisions can be resolved by entering records in an appropriate *overflow-area*.

Another method (called *open addressing*) is to search the table for an opening "nearby" by systematically probing locations at a distance of

incr(k)

further and further down the table. The *hash-increment function* is assumed to be positive, and is presumably chosen to lead to an opening fast.

The case when

incr(k) = F(h(k))

for all keys and some easy function F is sometimes called *double-hashing*.

3.4. Suppose we maintain a table

item[0 : M-1]

in this manner, where item[i] contains a key k such that h(k) = i if such a key was ever added and probably 0 otherwise. (It is strictly true only until collisions have occurred.)

One can *search* for k with

```
        visits := 0;
        probe := h(k);
        while visits < M do
        x := item (probe);
        if x = 0 then not found & exit for L fi;
        if x = k then found & exit for L fi;
        probe := probe - incr(k) (mod M);
        visits := visits + 1
        od;
        table full;
    L:
```

and *insert* k with essentially the same routine after replacing "not found" by

item (probe) := k.

In order that *all* locations of the table be probed in searching *we demand*

that incr(k) *is relatively prime to* M.

3.5. A hashing-method should avoid *clustering* of records (eliminating collisions from occurring too often), and is therefore usually dependent on an estimated probability distribution of the keys.

3.6. Amble & Knuth [5] proposed in 1973 to maintain *ordered hash-tables*.

It does not mean that the table itself is entirely ordered, but that large parts of the *chains are*.

One can search for k with

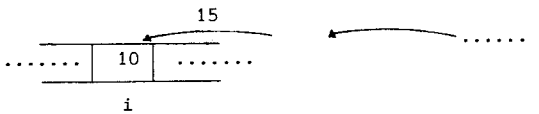
```
        visits := 0;
        probe := h(k);
        step := incr(k);
        while visits < M do
        x := item (probe);
        if x < k then not found & exit for L fi;
        if x = k then found & exit for L fi;
        probe := probe - step (mod M);
        visits := visits + 1
        od;
        table full;
    L:
```

The kind of arrangement does not specifically make successful searches faster, but *it tends to detect unsuccessful searches much earlier*.

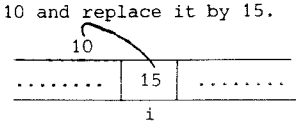
3.7. If a set of keys were inserted in an empty table from largest to smallest (in that order) with the algorithm of 3.4, then an ordered hash-table results.

The insertion of randomly presented keys into an ordered hash-table is somewhat trickier.

EXAMPLE. Suppose we want to move 15 into position i now occupied by 10.



Then we simply remove 10 and replace it by 15.



and try to re-insert 10. (Note that the replacement of an item by a *larger* item leaves all existing chains through that location consistent.)

Re-inserting 10 by back-chaining from h(10) on will bring us back to i again, so we better start the re-insertion procedure from position i on right-away. Note that for consistency the stepsize must become incr(10) now.

One can insert k in an ordered hash-table with

```
        visits := 0;
        probe := h(k);
        step := incr(k);
        key := k;
        while visits < M do
        x := item (probe);
        if x = 0 then item (probe) := key & exit for L fi;
        if x < key then item (probe) := key;
                        step := incr(x)
                                        fi;
        if x = key then found & exit for L fi;
        key := x;
        probe := probe - step (mod M);
        visits := visits + 1
        od;
        overflow;
    L:
```

Since the x-value keeps descending M is still a valid bound on the search-length.

Verify that insertion in an ordered hash-table indeed leaves another ordered hash-table!

3.7. The arrangement of N keys (N < M) in an ordered hash-table is *unique*.

3.8. Entirely different requirements of a data-organisation occur in *ordered* files or in structuring the directory of indexed files where one would like to perform operations

```
        find (k)
        insert (k)
```

but also

```
        delete (k)
                - remove the item with key k -
```

and perhaps

```
        split file at k
                - separate the file in a part with items < k
                  and a part with items ≥ k -
```

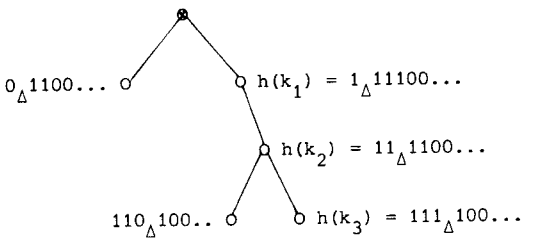
as efficiently as possible.

3.9. Deleting elements in a hash-table could break up existing chains and make items further down the chain inaccessible unless a cumbersome reorganisation procedure is applied. Most implementations therefore avoid *direct addressing schemes* and use *binary trees* instead.

3.10. An interesting intermediate file-organisation suggested by Coffman & Eve [19] results if we choose a hash-function which maps keys onto bit-strings.

One can interprete bitstrings as coding paths down a tree and maintain the table as a *hash-tree*. To insert k we follow h(k) bit-after-bit and enter it into the first open node encountered, unless the key was found on the way.

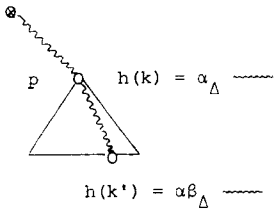
EXAMPLE.



Note the part of the hash-code used to enter k_1, k_2 and k_3. (0 ≡ "left", 1 ≡ "right").

Collisions are automatically avoided except when h(k) is "too short" to reach a distinguishing position (in which case an overflow-area must be entered).

3.11. Deleting k from its position p in the tree is easy when it is a leaf, but otherwise



one can delete an arbitrary leaf k' from its subtree and move it up into p to keep the hash-tree *consistent*, which is very easy also (provided k' is not too far away).

3.12. The average search-time in a hash-tree with N items will be ~ log N or less (assuming a uniform distribution), but series of inserts and deletes can seriously impair it and cause long and inefficient paths in the tree.

3.13. Various kinds of *binary search tree* organisations have been proposed which remain *balanced* throughout.

In such trees the file-elements are normally arranged in left-to-right order at the leaves, and the internal nodes contain appropriate queries

<u>if</u> key < k <u>then</u> left <u>else</u> right

which enable one to retrieve stored items in a top-down search (see also 1.9). To maintain a balance in a tree with N items one must design insert- and delete-procedures which keep the worst case distance from the root to any leaf ~ log N.

Known balanced tree-models include AVL-trees (Adelson-Velskii & Landis [1]), various kinds of B-trees (Bayer [8]), 2-3 trees (Hopcroft, see [3]), and 1-2 trees (Maurer & Wood [45]).

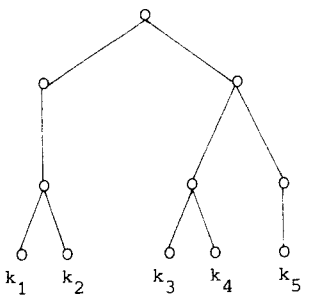
We shall consider a simplified kind of 1-2 tree recently proposed by Ottmann & Six [49].

3.14. A binary (search) tree is called an *HB-tree* if and only if

(i) all leaves have equal depth,

(ii) each node with only one son has a brother with two sons.

EXAMPLE.



The root of an HB-tree must have two sons (unless it is a leaf). If a node has only one son, say p, then p must have two sons or else be a leaf. It follows that there can be

<u>no</u>  - <u>chains</u>

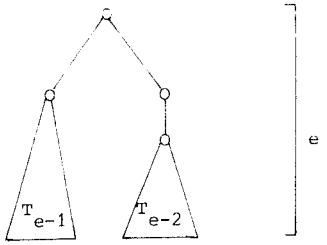
and HB-trees must always be quite "dense".

THEOREM. *The depth* d *of any HB-tree with* N *leaves satisfies*

$$\lceil \log N \rceil \le d < 1.44 \ldots \log(N+1) - 0.32 \ldots$$

Proof. (Our argument differs from Ottman & Six [49]). In a binary tree of depth d we always have $N \le 2^d \longrightarrow d \ge \lceil \log N \rceil$.

Let T_e $(e \ge 0)$ be an HB-tree of depth e with the smallest possible number of leaves. It follows that (up to symmetry) T_e must be of the form



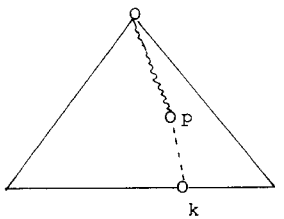
for $e \ge 2$, and T_e must have F_{e+2} leaves by induction (where F_i denotes the i-th Fibonacci-number). The depth d of a tree with N leaves satisfies therefore

$$\frac{\phi^{d+2}}{\sqrt{5}} - 1 < F_{d+2} \le N. \qquad \square$$

3.15. Observe that *HB-trees are AVL-trees in which all paths* from the root down the tree *are made equally long by inserting extra nodes of degree one* at the appropriate places. The effect, however, is that the operations for inserting or deleting items are much easier to explain.

3.16. One can insert k in the following manner.

Find the position in the tree where k must be inserted among the leaves.



If p had only one son then it now has two and the structure remains an HB-tree. If p had two sons already then it now has three, and we must

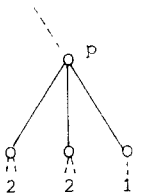
enact a procedure *split* p.

It is now advantageous to *pretend* that all leaves have two sons (for reasons of consistency only).

The *invariant* maintained throughout the algorithm is that

> *split* p is called if and only if p has three sons at least two of which have degree 2.

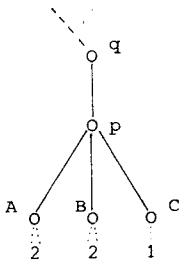
EXAMPLE.



Thus at least one of the outermost sons of p must have degree 2 and we shall designate one as the *active* son (on the *active* side of p).

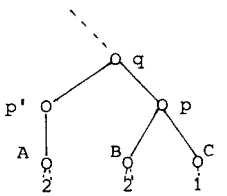
3.17. If p is the only son of q (say)

EXAMPLE



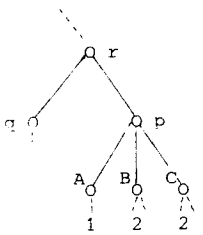
then create a new brother p' on the active side of p, connect the active son of p to p' and finish.

EXAMPLE.



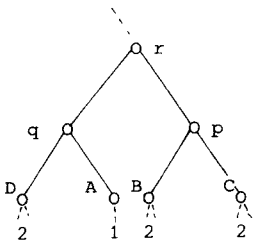
If p has a brother q and father r

EXAMPLE



then make the son of p on the q-side a son of q if q has degree one (which makes an HB-tree even if A had degree one) and finish

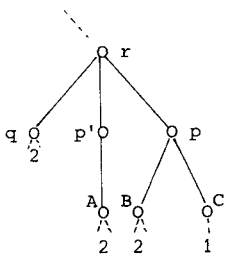
EXAMPLE



(as originally only son of q D must have degree 2, see 3.14)

and otherwise (when q has degree 2) create a new brother p' directly on the active side of p, make the active son of p the son of p', and continue with *split* r (observing that *the invariant is preserved*!).

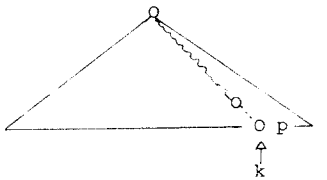
EXAMPLE



If p has no father we must have reached the root of the tree and must create a new root one level higher.

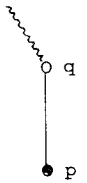
3.18. One can delete k in the following manner.

Find the position among the leaves where k is located.



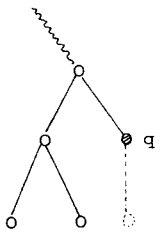
If p is the only son of q

EXAMPLE



then delete p

EXAMPLE



(note that q must have a brother of degree 2)

and continue to *delete* q.

After this initial phase the invariant maintained by the algorithm is that

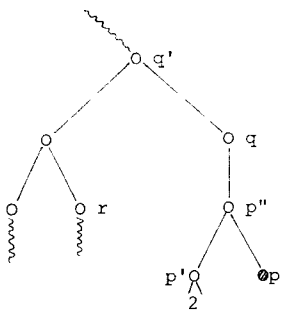
> *delete* p is called if and only if p must be deleted and it has a brother of degree 2.

3.19. (Our deletion-procedure differs slightly from Ottman & Six.)

In a *delete* p the order between p and his brother is irrelevant, so we shall always assume the brother to be on the "convenient" side.

If the father of p is the only son of q

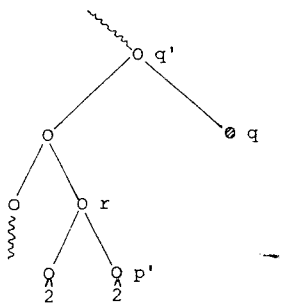
EXAMPLE



(this is the implied global structure)

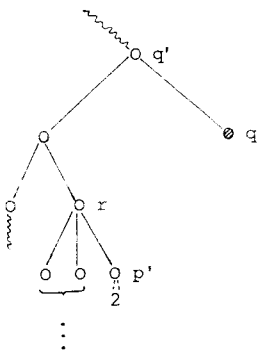
then make p' a son of r in case r has degree one, omit p and p", and continue with *delete* q (observing that the invariant is preserved).

EXAMPLE.



and otherwise (when r has degree 2) we make p' a son of r also while dropping p and p" but observe that now the invariant is satisfied for *split* r.

EXAMPLE

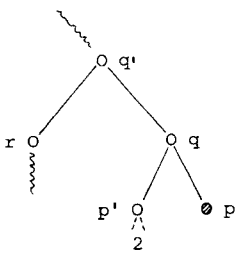


the original sons of r cannot both have degree one.

If *split* r (etcetera) runs out before q' is reached we continue with *delete* q (observing that the invariant will hold), otherwise we omit q and finish.

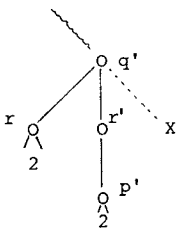
If the father of p has a brother

EXAMPLE



then make p' a son of r, omit p, and continue with *delete* q in case r had degree one, and otherwise (when r has degree 2) construct a new brother r' between r and q, make p' the son of r' and omit p and q, and finish.

EXAMPLE.



If the father of p is no son of any node we must have reached the top of the tree, and we can simply omit p and make his brother the new root.

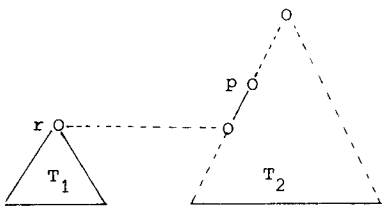
3.20. The procedures show

THEOREM. *One can execute find, insert, and delete-instructions on an arbitrary HB-tree with* N *leaves in* ~ log N *steps per instruction.*

3.21. Given two HB-trees T_1 and T_2 (in that order) one can merge the corresponding sets in the following manner.

Assume that T_1 is smaller than T_2 (and modify the algorithm accordingly otherwise). Find the element on the left-side of T_2 equally high as root(T_1).

EXAMPLE



Make r a son of p. If p had one son before it now has two and we can finish, otherwise the invariant for *split* p is enacted and an HB-tree results only after some additional moves.

Merging two trees is thus also bounded by ~ log N (or, more precisely, by depth(T_2) - depth(T_1)).

HB-trees can now also support instructions

split at k

in ~ log N steps as follows. Cut the tree along the path down to k in two parts, each part consisting of correctly ordered but loose hanging HB-trees. Merge the smaller trees into the larger continuously.

3.21. The idea of HB-trees can be generalised following similar steps to k-ary search trees (Maurer, Ottmann & Six [46]).

3.22. All previous tree-structures could be used also for implementing

disjoint *unordered files* or *sets*.

The simplest operations data-structures for sets should support efficiently include

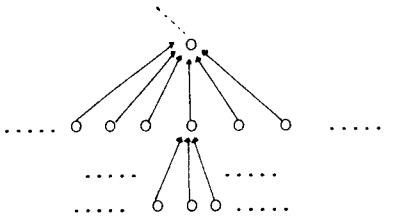
union (A,B,C)
- merge the (disjoint) sets named A and B into a new set named C -

find (k)
- determine the name of the set to which k currently belongs -

A variety of new, helpful programming techniques for *set-manipulation programs* developed during and after a systematic study of Aho, Hopcroft, and Ullman [3] (chapter 4, see also Knuth [44]).

Since we shall hardly ever need a search-facility in such applicatio we shall abandon the strict binary trees in favor of more general and *perhaps more compact* trees in which for each node only a father-link (if any) is listed.

EXAMPLE.



3.23. One can search the root of such a tree from any interior node on using

```
visitednode := startnode;
while (p := visitednode.father; p ≠ nil) do
visitednode := p
od;
root := visitednode;
```

but it is impossible to start at the root and search the tree in opposite direction.

It is therefore convenient to keep a fixed directory where set-elements are and store the set-name at the root of the tree (which also makes updating set-names easy).

We shall maintain a directory which yields for any set-name the

address of the corresponding root.

3.24. Union-find programs obviously benefit from *balancing* the trees.

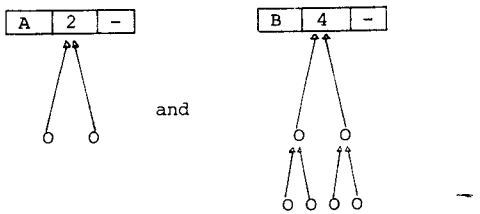
Let the *weight* of a node be equal to the number of leaves in its subtree.

Build the trees from items of

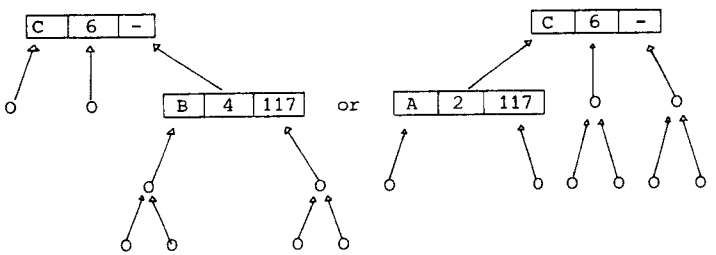
```
type node = record
                name: a character-string;
              weight: an integer;
              father: ↑ node;
            end;
```

The idea of balancing in performing a "union" is to *never merge a tree of larger weight into a tree of smaller weight.*

EXAMPLE. Note the difference in merging.



into



(Note also that in merging some of the original set-names are automatically "lost").

The objective of this rule is to keep the path-length in the merged trees down, thus permitting efficient find's at any moment.

3.25. A union (A,B,C) instruction can be executed by

```
set a := ↑ root A;
set b := ↑ root B;
if set a.weight > set b.weight then p := set a;
                                    set a := set b;
                                    set b := p
fi;
set b.name := C;
set b.weight := set b.weight + set a.weight;
set a.father := set b;
```

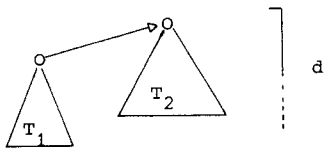
followed by an update of the set-name directory.

3.26. Given n one-element sets initially, it follows that

THEOREM. *The given organisation can support* m *(less than* n*) union-find instructions in* ~ log m *steps per instruction.*

Proof. Unions take ~ constant time. The time required by find's is bounded by ~ the maximal path-length in any tree built during the algorithm.

We observe that any tree of depth d inthe forest under construction must have weight $\geq 2^d$. This is true initially (for d = 0), and whenever two trees are merged into a tree of depth d later



either T_2 already had depth d and weight $\geq 2^d$ by induction or T_1 has depth d - 1 and

$$\text{weight}(T_2) \underset{\text{because of balancing}}{\geq} \text{weight}(T_1) \underset{\text{by induction}}{\geq} 2^{d-1}$$

which means that the resulting tree has

$$\text{weight} \geq 2^{d-1} + 2^{d-1} = 2^d$$

also.

The maximal path-length d therefore satisfies $2^d \lesssim m \longrightarrow d \sim \log m$. □

3.27. In merging, one set will always loose its original name (see 3.24) and because the result of balancing is rather unpredictable we may not know in advance which one. Note also that under circumstances the same root may be renamed more than once.

It is therefore impossible in the given implementation to perform

$lca(k_1,k_2)$
- find the *node* (or name) of the least recent ("smallest") set to which k_1 and k_2 were made to belong together, and undefined if no such set exists -

on-line.

Aho, Hopcroft, and Ullman [4] proved that one may execute ~ m union-find-lca instructions online in an average of ~ log m steps per instruction.

Using a different data-structure for forests we have been able to reduce this to a less than logarithmic bound (Van Leeuwen [60]).

THEOREM. *One may execute* ~ m *union-find-lca instructions in* ~ loglog m *steps per instruction on the average.*

(We note that Aho, Hopcroft, and Ullman [4] considered a slightly more general type of merge-instruction to occur in conjunction with lca's than we permitted here.)

4. PATH COMPRESSION

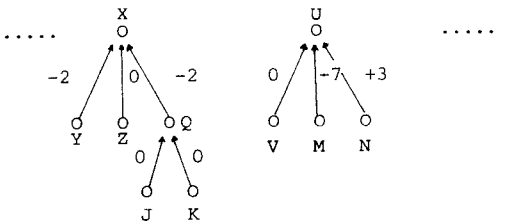
4.1. The *logical* development of set-manipulation programs often benefits from representing sets as trees, even though an actual implementation may

use different structures like arrays. It relates to a similar aspect of information management systems where the *data-model* of the user may be very different from the organisation of the physical database.

Trees were already used as data-structures in the early fifties when "sorting and searching" emerged as an area of intensive study.

In 1964 the idea was used by Galler & Fischer [32] in improving earlier work on Arden, Galler, and Graham [6] on MAD and aspects of FORTRAN-implementation. They considered how one can efficiently assign addresses to EQUIVALENCEd variables in FORTRAN-programs at compile-time, and used trees which actually carried names of variables in all nodes (including array-identifiers).

EXAMPLE. EQUIVALENCE (U[3],M[10],N) is represented, for example, among



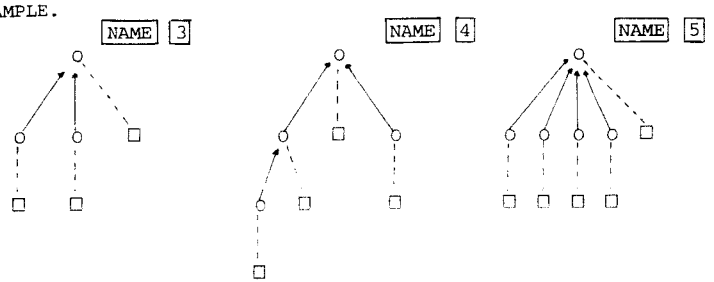
4.2. Trees have been successfully applied in programs for manipulating *disjoint* sets, which occur for instance when one is building *classes of an equivalence-relation* or in a variety of *graph-algorithms*.

4.3. There are only a few programming tricks for making ordinary set-manipulation programs more efficient, but some of these techniques are very powerful and fundamental.

Following an approach of Aho, Hopcroft, and Ullman [3] we shall study union-find programs for more abstractly presenting an analysis of *balancing* and *path-compression*.

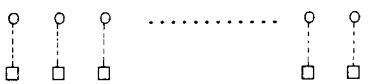
4.4. A collection of disjoint trees (representing disjoint sets) is called a *forest*. We shall assume that set-elements are attached to leaves only.

EXAMPLE.



The *weight* gives the number of set-elements stored in a tree. Directories tell where elements are stored, and where for each set-name the corresponding root is, but to execute a "find" one must explicitly follow father-links as in 3.23.

4.5. Assume (for simplicity) that we start with n singleton sets



and execute m finds and (at most) n - 1 unions.

Unions take constant time each, but the cost for finds is to be *the number of edges* traversed in each find-path.

Let F be the set of edges which will ever appear in the finds. (Note that these edges may not all be present in the current forest at some moment.)

4.6. A direct implementation could cost time ~ mn in worst case.

Balancing helps, and a rule as in 3.24 (*the weighted union rule*) for resolving unions reduces execution-time to ~ m log n in worst case.

Tritter (according to Knuth [43]) and also McIlroy & Morris (according to Hopcroft) apparently first realized how one can reduce the cost of (future) finds. In executing a find(k) one should save all nodes encountered on the find-path in a stack and attach all *directly* to the root once it is located (see fig.7). It is sometimes called the *collapsing rule*, but nowadays better known as *path-compression*.

4.7. It is clear that balancing and path-compression (or even path-compression alone) should eventually lead to *very* low trees and thus reduce the

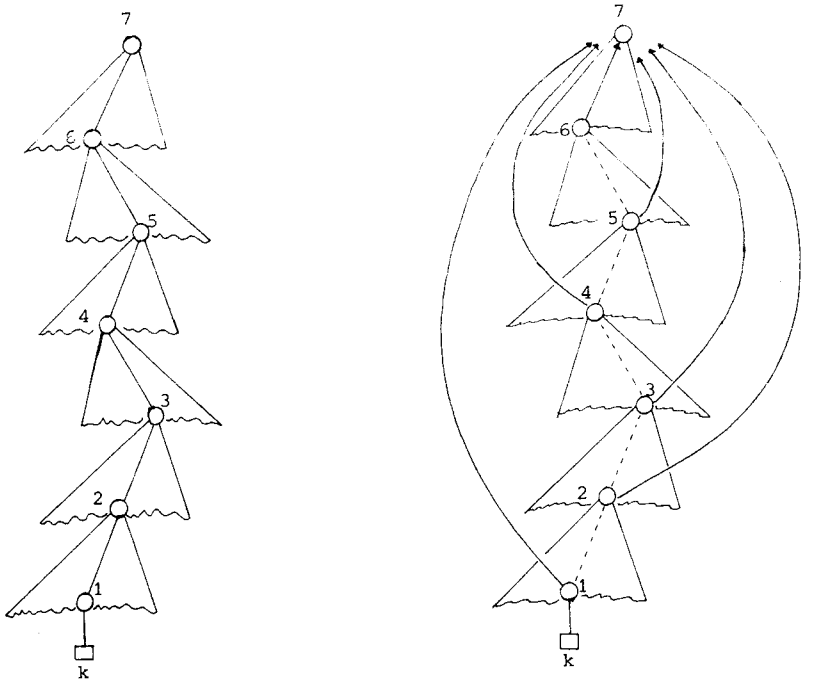


fig.7. The effect of path-compression in find(k)

cost for later finds substantially, but the precise analysis requires a new and important accounting-technique.

Paterson proved in 1972 that when only path-compression (and *no* balancing) is used and m ~ n, the execution-time is still bounded by ~ n log n. Hopcroft & Ullman [39] found an elegant level-crossing argument soon after and proved that balancing and path-compression together reduce the execution-time to ~ m $\log^*$ n, which is *almost* linear. The ideas were carefully analysed further by Tarjan [56] who finally succeeded showing that

> The cost for m finds(m ≥ n) and n - 1 unions with balancing and path-compression is *practically* linear in m.

Our presentation of the key-ideas in the analysis will differ somewhat from Tarjan's.

4.8. Consider arbitrary programs of m finds and n - 1 unions which use *some* strategy for each instruction first.

Nodes can gradually climb to higher and higher levels.

A "find" (*even when path-compression is used*) *does not change the current location of a root*, but it can make that some interior nodes are pulled up to a shorter distance of the root.

It is therefore important to know where the roots will be during the algorithm, which is *entirely* determined by the unions!

4.9. *Pretend* that we do all unions first (obeying whatever strategy there was), and see where all nodes will eventually finish.

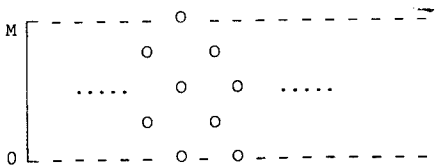
Leave all nodes where they are (hanging up in the air at their respective levels), but remove all connecting edges and start the program again.

Unions are now easy (just insert the connecting edge again), and we can concentrate entirely on what finds do in the model.

4.10. The level-number of a node v is called its *rank* r(v).

Let the maximum rank be M.

EXAMPLE.

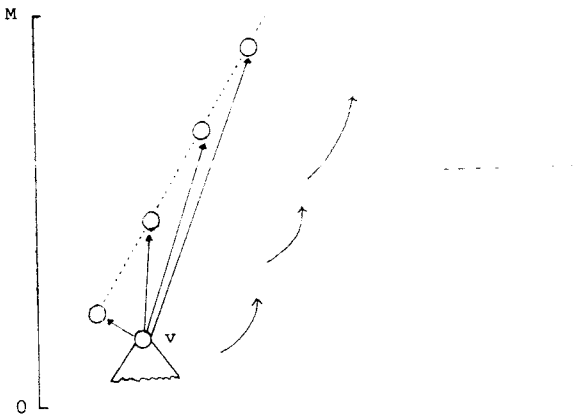


4.11. In a find with path-compression interior nodes get connected to nodes higher up, and *their (new) father always remains of higher rank*.

It means that at any moment the ranks along an upward path must form an *increasing* sequence.

4.12. We can estimate #F by counting how many times each node v ever occurring on a find-path can stretch its father-link to a higher node.

EXAMPLE.



We shall do so by counting how many times eventually occurring edges

can cross distinguished levels of a *grid*.

4.13. The crucial step in Tarjan's analysis is to work with not just one grid, but to consider k + 1 grids

$$A(i,*) \qquad (0 \le i \le k)$$

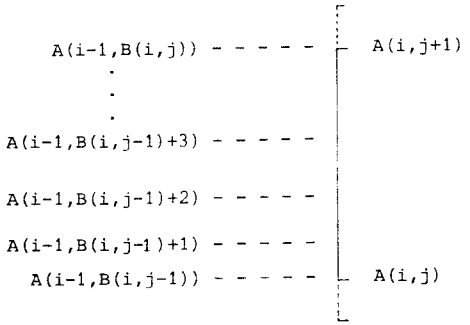
instead, where A(i-1,*) is a *refinement* of A(i,*). Thus we require the existence of a *monotone* function B(*,*) such that

$$A(i,j) = A(i-1,B(i,j-1)) \qquad (i \ge 1),$$

where we define B(i,-1) = 0 for consistency.

Let $a_B(i,x) = \min\{j \mid A(i,j) > x\}$. Thus $a_B(i,M)$ gives how many levels in each grid there are.

EXAMPLE.



4.14. The finest grid is A(0,*) defined by

$$A(0,x) = cx \qquad (c \geq 2).$$

Note that any point ever occurring on a find-path must cross this grid after being pulled up c - 1 times.

The next grids all begin with

$$A(i,0) = 0.$$

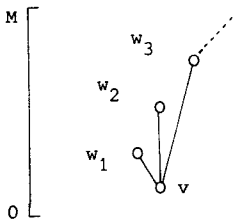
Let the number of nodes v with $A(i,j) \leq r(v) < A(i,j+1)$ be s(i,j), and assume that

$$s(i,j) \leq \frac{f(n)}{g(i,j)} \qquad (i \geq 1).$$

4.15. For each node v we should estimate how many edges

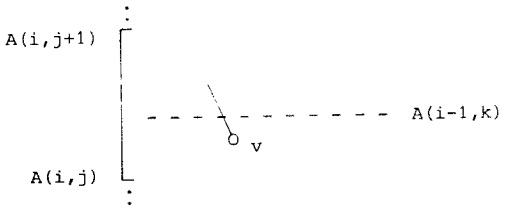
there can be on later find-paths.

EXAMPLE.



Follow v as it "bumps" into lower grids while rising first until it crosses a next grid A(i-1,*) for the first time.

EXAMPLE.

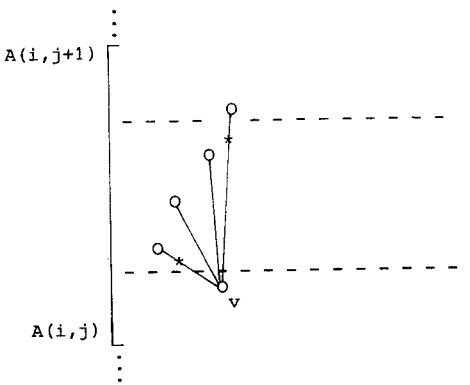


Watch v to see when it gets attached to a node across the next level of A(i,*), *in the meantime* "charging the i-th grid" one edge whenever v gets attached to a node across yet another level of A(i-1,*)

- thus the i-th grid can get charged at most B(i,j) - B(i,j-1) times -

and "charging the find" whenever v is in between but on its way

EXAMPLE



It means that we partition F into sets

$$F_0 = \{(v,w) \in F \mid \exists_j\ A(0,j) \leq r(v) < r(w) < A(0,j+1)\}$$

$$F_i = \{(v,w) \in F \mid \exists_j\ A(i,j) \leq r(v) < r(w) < A(i,j+1) \text{ and } \exists_\ell\ r(v) < A(i-1,\ell) \leq r(w)\}$$

(for $1 \leq i \leq k$)

$$F_{k+1} = \{(v,w) \in F \mid \exists_\ell\ r(v) < A(k,\ell) \le r(w)\}$$

and divide the charges (i.e. the edge-count) between

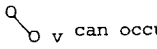
$$F_i - L_i \quad \text{and} \quad L_i \qquad (0 \le i \le k+1),$$

where

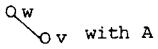
$$L_i = \{(v,w) \in F_i \mid \text{on its find-path } (v,w) \text{ is the last edge in } F_i\}$$

(thus all edges in $F_i - L_i$ are *not* last and must contribute a new crossing across the level crossed by the last).

4.16. Clearly $\#L_i \le m$, because each find can get charged at most one last edge.

4.17. An edge  can occur in $F_0 - L_0$ at most $c-1$ times, because after that many times v *must* be attached to a node across the first level of $A(0,*)$ (and the edges are charged to F_1 or F_2 etc.). Thus

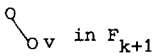
$$\#(F_0 - L_0) \le (c-1)n.$$

4.18. The points v on edges  with $A(i,j) \le r(v) < r(w) < A(i,j+1)$ in $F_i - L_i$ (see 4.15) can contribute at most $B(i,j) - B(i,j-1)$ each, and thus for each $1 \le i \le k$

$$\#(F_i - L_i) \le \sum_{j=0}^{*} (B(i,j) - B(i,j-1))s(i,j) \le$$

$$\le \sum_{j=0}^{*} \frac{B(i,j) - B(i,j-1)}{g(i,j)} f(n) \le h(i)\cdot f(n)$$

for some function h.

4.19. For each node v there can be at most as many edges  in $F_{k+1} - L_{k+1}$ as there are lines to cross in the k-th grid. Thus

$$\#(F_{k+1} - L_{k+1}) \le a_B(k,M)\cdot n\ .$$

4.20. Adding up all estimates the total charge for m finds and $n-1$ unions

is bounded by

$$(k+2)m + cn + \sum_{1}^{k} h(i)\cdot f(n) + a_B(k,M)\cdot n\ ,$$

and all we have to do is choose the grids such that the expression is as small as possible.

4.21. It is important to note that the bound was derived *independent* of the strategy for unions. We can immediately obtain Paterson's result as follows.

THEOREM. *The cost for* n *finds and* $n-1$ *unions using only path-compression (and no balancing) is* $\sim n \log n$.

Proof. Let $B(i,j) = c\cdot(j+1)$ such that by induction we have $A(i,j) = c^i\cdot j$.

Since no balancing is used, the "best" estimate for $s(i,j)$ is just some $\frac{f(n)}{g(i,j)}$ with $f(n) = n$ and $\sum_{j=0}^{*} \frac{1}{g(i,j)} \sim 1$.

It follows that $h(i) = c$ for $1 \le i \le k$, and also $a_B(k,M) \sim M/c^k$.

By 4.20 the total cost for n finds and $n-1$ unions is bounded by $\sim ckn + Mn/c^k$. Let $c = 2$ and choose $k \sim \log M$. It follows that the cost is bounded by $\sim n \log M$, which could be as worse as $\sim n \log n$ for an unbalanced tree. □

Fischer [27] proved in fact that $n \log n$ is tight to within a constant factor.

4.22. When a proper form of balancing is used we get $M \sim \log n$, and it immediately follows from 4.21 that the cost for n finds and $n-1$ unions is at least reduced to $\sim n \log\log n$.

Tarjan showed that one can get much better bounds from 4.20 in this case.

Remember from 3.26 that when the weighted union rule is used (or indeed any other rule for proper balancing) a node of rank r must lead a tree with at least 2^r leaves. The number of rank r nodes is therefore bounded by $n/2^r$, and it follows that

$$s(i,j) \le \sum_{r=A(i,j)}^{A(i,j+1)-1} \frac{n}{2^r} \le \frac{2n}{2^{A(i,j)}} \quad \left(= \frac{f(n)}{g(i,j)}\right).$$

Thus we must choose B such that we can somehow reasonably bound

$$\sum_j \frac{B(i,j) - B(i,j-1)}{2^{A(i,j)}} .$$

4.23. *Big trick*: one can choose $B(i,j) = A(i,j)$ for $j \geq 1$ and $B(i,0) = 1$. (We could have choosen even larger B's but this will do to get extremely sharp results already).

4.24. The grids we now have are defined by

$$A(0,x) = cx$$

$$A(i,0) = 0$$
$$A(i,1) = c$$
$$A(i,j+1) = A(i-1,A(i,j)) \qquad (i,j \geq 1)$$

which yields an analog of *Ackerman's function* with an equally super-fast, not primitive recursively bounded growth-behaviour.

Observe that

$$A(1,x) = c^x$$

$$A(2,x) = \left.\begin{matrix} c^{c^{\cdot^{\cdot^{\cdot^{c}}}}} \end{matrix}\right] |x| \quad , \text{ etc.}$$

4.25. Since we can now choose $h(i) = 2$ the total cost for m finds and $n-1$ unions is bounded by

$$\sim k(m+n) + cn + a_A(k,\log n)n$$

where one should notice that already for $k = 2$ the function $a_A(k,\log n)$ is *very* slowly growing.

For $k = 2$ and $c = 2$ we get Hopcroft & Ullman's result that m finds and $n-1$ unions take at most time $\sim m + n \log^* n$.

Define a "functional inverse" $\alpha(m,n)$ of the A-function in 4.24 by

$$\alpha(m,n) = \min\{j \mid A(j,\tfrac{m}{n}j) > \log n\}.$$

We obtain a form of Tarjan's result

THEOREM. *The cost for* m *finds and* $n-1$ *unions using balancing and path-compression is bounded by* $(m+n)\alpha(m,n)$.

Proof. Choose $k \sim \alpha(m,n)$, and observe that $a_A(k,\log n) \leq \frac{m}{n}\alpha(m,n)$.
$\longrightarrow$ bound $\sim (m+n)\alpha(m,n)$. □

Assuming that in practical situations usually $m \geq n$ it follows that the time-bound is

$$\sim m\,\alpha(m,n)$$

and that is *practically* linear. Tarjan [56] proved that the algorithm is not completely linear.

4.26. There are numerous applications of Tarjan's theorem in data-organisation, showing that all kinds of algorithms can be made practically linear with balancing and path-compression.

Hopcroft & Karp [38] proved in 1971 that the equivalence of two finite automata with at most n states can be determined in $\sim n \log n$ steps, but one may now show

THEOREM. *The equivalence of* n*-state finite automata can be determined in* $\sim n\,\alpha(n,n)$ *steps.*

Proof. Let $M_1 \oplus M_2 = \langle S_1 \oplus S_2, \Sigma, \delta, \{s_1, s_2\}, F\rangle$ and consider the equivalence-relation E determined by

$$p\ E\ q \longleftrightarrow \forall_x (\delta(p,x) \in F \longleftrightarrow \delta(q,x) \in F).$$

M_1 and M_2 are equivalent if and only if $s_1\ E\ s_2$. It means that the smallest relation R satisfying

(i) $s_1\ R\ s_2$
(ii) $p\ R\ q \longrightarrow \forall_{\sigma\in\Sigma}\ \delta(p,\sigma) R \delta(q,\sigma)$

must be contained in E.

Thus M_1 and M_2 are equivalent if and only if we can build R-classes as if it were equivalence-classes without ever getting a final and a non-final state together in the same class.

Prepare classes with names $\in \{1,\ldots,2n\}$ and attributes $\in$ {FINAL, NON-FINAL} with the following algorithm

```
classes ⟵ a family of 2n disjoint sets of one
          element each:
                 set_i ⇝ [ state i | attr. ]

queue ⟵ (s_1,s_2);
while queue ≠ φ do
detach (p,q) from queue;
name_1 := find(p);
name_2 := find(q);
if name_1 ≠ name_2 then
    if name_1·attrib ≠ name_2·attrib then
            M_1 and M_2 are not equivalent & signal fi
          else union(name_1,name_2,name_1);
               attach (δ(p,σ),δ(q,σ)) to queue
               for each σ ∈ Σ
fi
od;
```

When M_1 and M_2 are equivalent the algorithm will produce exactly all equivalence-classes of admissible states, but if they aren't it will find the clash.

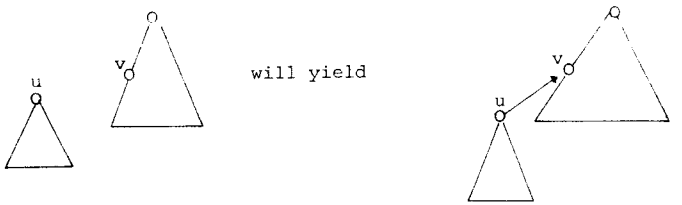
There can be at most $\sim n$ unions, and thus at most $\sim n \cdot \#\Sigma$ pairs of states will ever get onto the queue. We need to execute at most n unions and $O(n)$ finds, assuming that $\#\Sigma$ is fixed. □

4.27. An interesting and instructive application of path-compression was found by Aho, Hopcroft, and Ullman [4] in studying programs which maintain a forest with the following instructions

merge(u,v)
- to be executed only if u is a root and v is not in the tree with root u. The instruction makes u a son of v -

depth(v)
- determine the distance of v to the root of the tree to which it currently belongs -

EXAMPLE. A merge(u,v) applied to



where as a result depth(u) := depth(v) + 1.

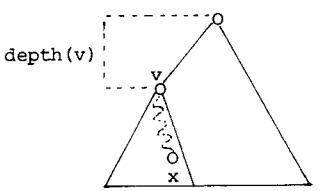
Even balancing may not seem straightforward here as it might destroy the ability to determine depth correctly if it was applied directly to the *real forest*.

In addition to the real forst we shall therefore maintain an auxiliary *shadow forest* which does support depth-instructions quickly, and we shall give a proof of

THEOREM. *One can execute* $n-1$ *merge- and* m *depth-instructions* $(m \geq n)$ *in* $\sim m\,\alpha(m,n)$ *steps.*

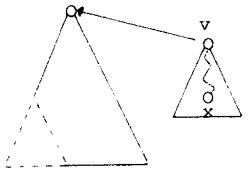
4.28. A depth-instruction looks like a find, but if you want to move an internal node v

EXAMPLE



and make it a direct son of the root (as we would in path-compression), then one must attach a *relative displacement* ("disp") to it which gets added to the depth-count when we pass v from x on our (shortened) way up to the root.

EXAMPLE.



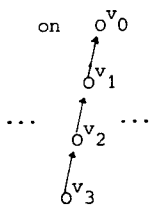
disp(v) = depth(v) - 1

4.29. Suppose we start with a forest of n singletons, and maintain the structure with nodes of type

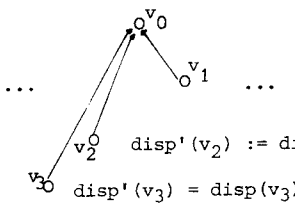
```
type node = record
                w:    number of descendant leaves;
             disp:    current displacement;
           father:    ↑ node;
              end;
```

In executing a "depth(v)" we can do path-compression as in a find, provided we adjust the displacement of all nodes pulled up to the root appropriately.

EXAMPLE. A depth(v_3) on



should result in



$disp'(v_2) := disp(v_2) + disp(v_1) + 1$

$disp'(v_3) = disp(v_3) + disp'(v_2) + 1$

with $depth(v_3) = disp'(v_3) + 1 + disp(v_0)$.

The following iterative algorithm will work

```
stack S ⟵ φ;
p := ↑v;
if p = root then depth := p.disp & HALT fi;
while p ≠ root do
push p;
p := p.father
od;
root := p;
p := pop S;
while stack ≠ φ do
q := pop S;
q.father := root;
q.disp := q.disp + p.disp + 1;
p := q
od;
depth := p.disp + 1 + root.disp
```

(Note that the w.field of internal nodes is not up-dated and becomes unreliable, but fortunately we only need its integrity for the root.)

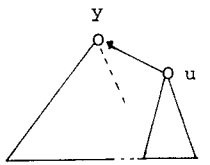
4.30. To execute a merge(u,v) we first check in the shadow forest that the instruction is permissible by executing

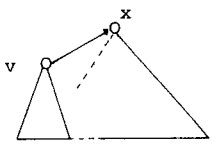
depth(u)
depth(v)

(with path-compression).

We should find that depth(u) = 0, and also get as a side-result the roots x and y of the trees where u and v currently belong. Check $x \neq y$.

EXAMPLE.

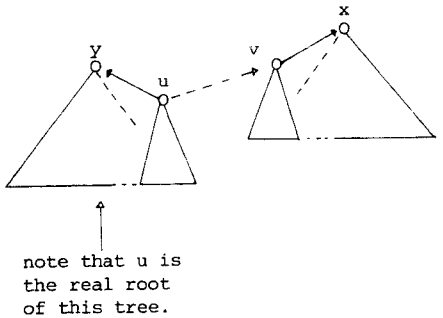




(Note that the roots in the shadow-forest could differ from the roots in the real forest.)

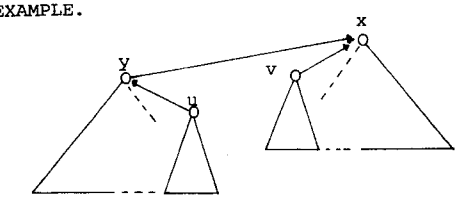
With some care the weighted union rule can be applied to the merge as reflected in the shadow forest.

EXAMPLE. The merge(u,v) should come up with a correctly displaced structure equivalent to



4.31. If y.weight ≤ x.weight then we may just as well make y a direct son of x, provided we adjust the displacement of y to correct for all nodes in that tree.

EXAMPLE.



where disp'(y) = disp(y) + 1 + disp(v).

Note for instance that

$$\begin{aligned} \text{depth}(u) &:= \text{disp}(u) + 1 + \text{disp}'(y) + 1 + \text{disp}(x) = \\ &= \text{disp}(u) + 1 + \text{disp}(y) + 1 + \text{disp}(v) + 1 + \text{disp}(x) = \\ &= 1 + \text{disp}(v) + 1 + \text{disp}(x) \end{aligned}$$

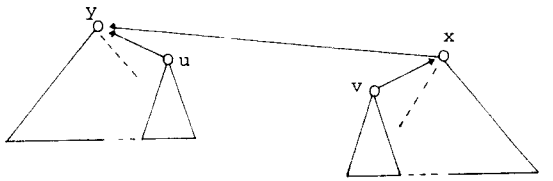
which is what is should be.

One can implement this case simply with

```
      x.w := x.w + y.w;
   y.disp := y.disp + 1 + v.disp;
```

4.32. If y.weight > x.weight then "balancing" requires that we attach x to y instead. Adjusting the displacements is slightly trickier now.

EXAMPLE.



where now disp'(y) = disp(y) + 1 + disp(v) + 1 + disp(x) and for proper correction disp'(x) = disp(x) - disp'(y) - 1.

We should execute

```
      y.w := y.w + x.w;
   y.disp := y.disp + 1 + v.disp + 1 + x.disp;
   x.disp := x.disp - y.disp - 1;
```

(and the order of the instructions is important!).

4.33. The given implementation of m depths and n - 1 merges actually requires ~ 2n extra auxiliary depth-instructions, but since m ≥ n the time needed is the same as for the underlying finds and unions on the shadow-forest. Apply Tarjan's theorem.

4.34. A generalisation of the techniques for merge-depth programs for other functions over trees was formulated recently by Tarjan [57]. He showed for instance that one can *verify* a minimal spanning tree on an n-node graph with m edges in time ~ m α(m,n).

5. ASSOCIATIVE SEARCH STRUCTURES

5.1. The known techniques for data-structuring can usually be combined somehow by a creative programmer into an acceptable organisation for efficiently handling records on an identifying primary key, but additional, non-trivial considerations may be needed to work it out for accessing records on *secondary* keys.

In such applications a user can issue *data-requests* (or *queries*) asking the system to produce all data-items from a file or data-base which possess a desired combination of *attributes*.

Queries are typically of the form

- list all records with property P in workspace W -

where P may be specified by some formula in relational algebra or, say, as a predicate in a calculus-based language like Codd's data-sublanguage ALPHA [18].

The process of answering data-requests is called *data-retrieval*. (One may speak of re-three-val after probing the data-base twice.)

5.2. An important task in data-base management consists of organising information such that all queries in a known language-system can be handled quickly. One may also require that in realistic applications this property must be maintained under a choice of admissible operations on the data-set.

The task usually reduces to

- resolving the nearly always present *time/storage trade-offs* favourably -

and in close connection to it

- finding the optimal degree of *redundancy* needed in the data-base -

5.3. There may be an advantage in keeping lists with instant answers to basic queries, and it is conceivable that in such situations one record has to occur on many lists.

Note that redundancies in the data-model can possibly be avoided in the physical data-base.

5.4. A common principle learns not to store a data-item where it is not needed, and not to duplicate a data-item when you can avoid it.

The use of *pointers* tends to be successful exactly for this reason, but one should not forget that storing many pointers may require much space also.

5.5. One may wish to avoid storing explicit pointers and try to "hack" a pointer (as a record key) in little pieces which one can search in a "small" auxiliary table first.

If at least one piece is not found then there is no point in searching the original, much larger file.

5.6. A technique of Bloom [10] suggests to maintain a long bit-string

$$\alpha_0 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \alpha_M \qquad \text{(some M)}$$

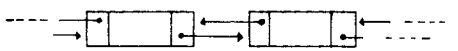
and to use hashing functions $h_1, \ldots\ldots, h_s$ as follows. *Mark that record* k *is stored by setting bits* $\alpha_{h_1(k)}, \ldots\ldots, \alpha_{h_s(k)}$ *to* 1, as if it were unique coordinates for k.

Provided $\binom{M+1}{s} > n$, there are good chances that this method can distinguish many records.

5.7. In cases where pointers are used to *link* elements together in a (linear) list, the need for extra space seems unavoidable.

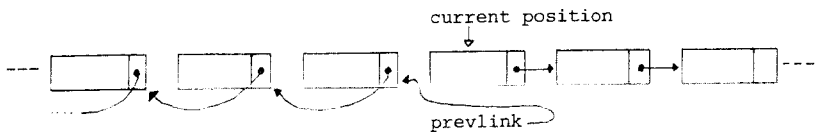
EXAMPLE. Doubly linked linear lists are usually built from elements of

```
type block = record
               ℓlink : ↑block;
               info  :  ...
               rlink : ↑block;
             end;
```



Nevertheless (depending on the application) it may take little effort occasionally to eliminate much of the pointers even here.

EXAMPLE. Suppose a doubly linked list was choosen in order to traverse a file from left-to-right *and back*. Then one can save 50% in pointer-space by using a linked structure



One can move one block to the right with

```
if (next := pos.link; next = nil) then exit fi;
pos.link := prevlink;
prevlink := pos;
pos      := next;
```

and one can move one block back (to the left) with

```
if (next := prevlink; next = nil) then exit fi;
prevlink := next.link;
next.link := pos;
pos      := next;
```

5.8. It is common programming practice also sometimes to add a pointer into records as a means of referring to a part in memory where auxiliary information relevant to each individual record is stored.

It is probably much better to do so than to list the auxiliary information explicitly in each record in detail, especially in cases where many records *share* the same data.

Thus *pointers are used here to eliminate redundancy*, and we pay only one additional field per record (instead of many).

There are many variants of this idea; one may want to eliminate the pointers altogether and use a table at a "known" location instead in which is listed where the auxiliary data for each record can be found (but then again, at the cost of a table-search).

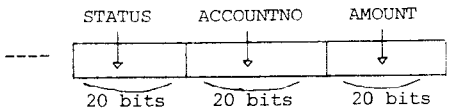
5.9. In an interesting paper Bobrow [11] pointed out very recently that the problem of locating "associated information" may be attacked by making better use of the one and only attribute of a record that comes for free: its *address*!

It is there anyway, and *it occupies no space within the record*.

The idea is that one can simply *hash the address* of a record (serving as its primary key) into a unique position of a table, and immediately find what we need.

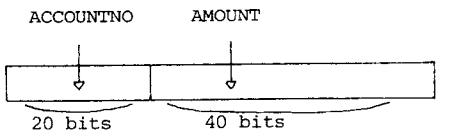
The technique of *hash-linking* can be applied in various forms.

EXAMPLE. A company may keep records



for its customers which exactly fit in one word for amounts not exceeding 20 bits of storage.

The range of AMOUNT is conceivably larger, and in programming there is always the tendency to allocate enough space to handle "the worst case". Assuming that amounts *usually* fit in 20 bits one should not allocate more words for each record, but instead *hash-link* a cell to a separate, much smaller table with "special purpose" records



whenever its amount overflows (which presumably doesn't happen very often). An "all ones"-field in the original record can signal that it is hash-linked.

As long as there are no collisions one may compress the hash-table even further because it isn't necessary to keep keys in the table (a hash-link will immediately lead to the right spot). Otherwise one might mark a record to indicate that it collided, and that one must follow a hash-link into an overflow-area where this time identifying keys are available.

5.10. The methods for locating "secondary information" in a data-file get more involved when we consider querying a data-base.

One might distinguish the following types of queries (following Rivest [53])

- exact match queries
  locate the record with key k

- component match queries
  locate all records in which the i-th attribute of the key is j

- partial match queries
  locate all records for which s specified attributes match given data (and the remaining attributes can be arbitrary)

- range match queries
  locate all records for which specified attributes are within a given range

- good match queries
  locate all records which have a key differing from k by at most a given tolerance

- best match queries
  locate all records which have a key with least distance to k

In more general, *boolean queries* one can specify records by means of an arbitrary propositional formula over the attributes.

5.11. Most queries seem to force a search through the entire data-base, and there is great need for a specific data-organisation (perhaps one for each single type of queries) to support faster retrieval. It is the object of current research in software and in hardware systems to implement *associative search* in a best possible way.

There are probably natural limits to how efficient data-retrieval can be. If one specifies more and more conditions on the attributes, then the amount of work to test each record *increases* while the actual number of records satisfying the query probably *decreases*.

5.12. Consider the following, typical *library search* problem.

Suppose records are classified on k out of many possible keywords,

and suppose we want to store records in such a way that we can later find all records with s ($\leq$ k) attributed keywords quickly without having to inspect each element of the data-base individually.

How can such a goal be achieved?

Common solutions are based on an appropriate partitioning of the data-base in various *buckets*.

5.13. Apparently Gustafson [35] systematically investigated such problems first, and in 1969 he suggested the following variant of hashing for obtaining an acceptable solution.

Choose an "ordinary" hash-function

$$h: \{\text{keywords}\} \longrightarrow \{0,\ldots\ldots,b-1\}$$

for some b > k, and build a "super" hash-function

$$H: \text{unordered k-tuples} \longrightarrow \text{b-bit numbers}$$

classifying each record with attribute-set $\{\beta_1,\ldots\ldots,\beta_k\}$ in the bucket with address $\alpha_0\ldots\ldots\alpha_{b-1}$ where

$$\alpha_i = 1 \longleftrightarrow \exists_j\ h(\beta_j) = i.$$

We thus hash each key into a bit-position, and sometimes speak of *hash-coding*. Assuming a more or less uniform distribution, buckets will have $\sim N/\binom{b}{k}$ elements each.

5.14. To search for all records which contain keywords $\gamma_1,\ldots\ldots,\gamma_s$ (some $s \leq k$) one should first determine the *partial* address $\alpha_0\ldots\alpha_{b-1}$ with

$$\alpha_i = \begin{cases} 1, & \text{if } \exists_j\ h(\gamma_j) = i \\ *, & \text{otherwise} \end{cases}$$

and then fill in the *'s with k - s ones and b - k zeros in all possible ways to obtain the names of all buckets which one would have to inspect only.

On the average one would have to scan through no more than

$$\sim \binom{b-s}{k-s} \cdot \frac{N}{\binom{b}{k}} = \frac{\binom{k}{s}}{\binom{b}{s}} \cdot N =$$

$$= (k-s+1) \ldots\ldots (b-s) \frac{k!}{b!} N \text{ elements,}$$

which indeed tends to be a very small fraction of the entire data-base for $s \sim O(k)$.

The method is promising for small k.

5.15. We note in passing that methods for partial match retrieval based on *intersecting inverted lists* will not be discussed here (see Engles [25] or Roberts [54]).

5.16. In 1971 Rivest (see Knuth [44], Rivest [52]) suggested another, simple hash-coding algorithm for partial match retrieval in a data-base.

Suppose now that keys are ordered k-tuples, and assume it simply are length k strings over some alphabet Σ with $\#\Sigma = \sigma$.

Determine an "ordinary" hash-function

$$h: \Sigma \longrightarrow \text{b-bit codes}$$

(where perhaps $2^b < \sigma$), and build a super hash-code

$$H: \Sigma^k \longrightarrow \text{c-bit numbers}$$

with $c = kb$ by defining

$$H(\beta_1 \ldots\ldots \beta_k) = h(\beta_1) \ldots\ldots h(\beta_k).$$

There will be $\sim N/2^c$ elements per bucket on the average.

5.17. Let a partial match query be represented as a length k string

$$\gamma = \gamma_1 \ldots\ldots \gamma_k$$

where for each i

$$\gamma_i \in \Sigma \text{ ("specified") or } \gamma_i = * \text{ ("unspecified")},$$

and suppose there are s stars.

As in Gustafson's method, again, we only have to search in the $(2^b)^s$ buckets with a name matching γ in the specified positions.

It means that on the average we must inspect

$$\sim \frac{N}{2^c} \cdot 2^{bs} = 2^{b(s-k)} N \quad (= \sigma^{s-k} N)$$

elements, which is

$$\sim 2^{s \cdot \log \sigma} = \sigma^s \sim N^{s/k}$$

for $b \sim \log \sigma$ and a full data-base with $N \sim \sigma^k$.

5.18. Provided enough extra storage is available there is a simple variant of Rivest's algorithm from which one may expect an even better behaviour.

Partition each length k key in m fields of base-σ numbers of $\frac{k}{m}$ digits

$$\underbrace{\quad A_1 \quad}\ \underbrace{\quad A_2 \quad}\ \ldots\ldots\ldots\ \underbrace{\quad A_{k/m} \quad}$$

and store each record with such a key in bucket A_j of the j-th data-base for *each* $1 \le j \le m$.

We thus list each record many times in our data-model, but it will appear that *the redundancy will pay off in greater retrieval-efficiency*.

5.19. In a partial match query with a total of s stars there must be at least one field with $\le s/m$ stars. Among these choose the field with the fewest number of stars (say field j).

One may expect that the j-th *data-base will have the elements we search for concentrated in the smallest number of buckets.*

Roughly, there are $\sim N/\sigma^{k/m}$ elements per bucket and we have to scan through

$$\sim N/\sigma^{k/m} \cdot \sigma^{s/m} = \sigma^{s-k/m} \cdot N$$

elements. This compares favourably with the bound $\sim \sigma^{s-k} \cdot N$ in 5.17 (although a more precise analysis would be needed to support the apparent conclusion).

5.20. One may similarly develop other methods, and it becomes of interest to study how the various hash-coding algorithms for partial-match retrieval compare in efficiency.

Rivest [52] (also [53]) answered this question for a specific class of hashing algorithms, and we shall give a simplified approach to make his result understandable.

5.21. Each hash-code storage schema consists of a super hash-function

$$H: \Sigma^k \longrightarrow \text{a finite set of buckets}$$

(the algorithm of 5.18 is *not* in this category).

The relation "having the same H-value" gives a partitioning of Σ^k, and the various equivalence-classes may be called *H-blocks*.

Now consider how we answer partial-match queries γ with s stars. The algorithm must return the names of all H-blocks which contain at least one record matching γ in the specified positions.

Rivest suggested to "measure" H by the *average* number of H-blocks needed to answer s-star queries.

5.22. Suppose there are Q s-star queries.

On the average we need to inspect a number of blocks equal to

$$\frac{1}{Q} \sum_{\gamma} (\text{\# H-blocks with a } \gamma\text{-match}) =$$

$$= \frac{1}{Q} \sum_{B} (\text{\# s-star } \gamma\text{'s touching B}),$$

where B ranges over all H-blocks.

Each s-star query occurs at least once in this summation, and it is counted more than once if and only if it hits more than one block.

It is conceivable (and Rivest proved it for "balanced" hash-coding algorithms) *that the average behaviour will come out best once the "answer" to as many partial-match queries as is practical is contained in one block.*

It means that preferably H-blocks must allow a decomposition into

$$\Sigma_1 \times \Sigma_2 \times \ldots\ldots \times \Sigma_k$$

where for each i: $\Sigma_i = \Sigma$ or $\#\Sigma_i = 1$.

It holds for the algorithms of Gustafson and Rivest, and will hold for later algorithms also.

5.23. Based on these heuristics it seems promising to consider a general type of (hash-coded) *partial-match file design* which consists of a table linking *independent* key-*patterns* and bucket-addresses

PATTERN	BUCKET
⋮	⋮
*1**001*1*	25
⋮	⋮

where

(i) each row contains s stars and k-s symbols from Σ;

(ii) any two key-patterns (rows) differ in at least one specified position (which means that buckets will be disjoint).

Technically each entry in the table is an s-star partial-match query, but obviously other queries can be answered from it also. If no entry matches a record-query then no records of that type are in the data-base.

The way stars are distributed in the key-patterns can strongly influence the average performance of the file-design. Rivest [53] and later Burkhard [14] (see also Bentley & Burkhard [9]) studied several special designs with an alleged approximately optimal behaviour.

5.24. One may assume that entries in the table (as strings over $\Sigma \cup \{*\}$) are listed in *radix-sorted* order (a general storage method suggested by Hildebrand & Isbitz [36] as early as 1959). It is likely that the best results will be implied when the entire data-base is stored in a compatable manner.

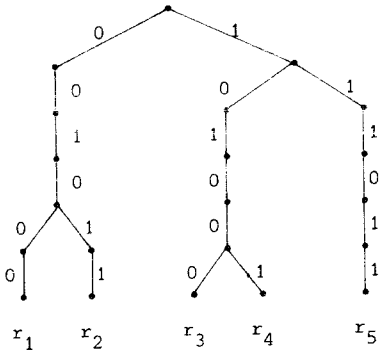
The idea is that in such a scheme it becomes unnecessary to store the table separately, and its structure (or rather, the structure of its buckets) is implied by a straightforward digit-wise search where each "*" is interpreted as a "branch in both directions"-instruction.

5.25. Assume that $\Sigma = \{0,1\}$ (for simplicity).

When length-k strings of 0's and 1's are entered into a tree where at each node "0" is interpreted as "left" and "1" as "right", each record-key becomes a code for the path to a unique leaf where the corresponding record-address can be stored.

The idea is independently due to de la Briandais [23] and Fredkin [30], and the resulting storage structure is known as "*trie memory*" (pronounced usually as "try-memory", although the editor of the section "Techniques" in the 1960-issue of the Communications pointed out in a footnote to Fredkin's paper that "trie" apparently was derived from the word re-TRIE-val).

EXAMPLE. The trie corresponding to {001000,001011,101000,101001,111011} is



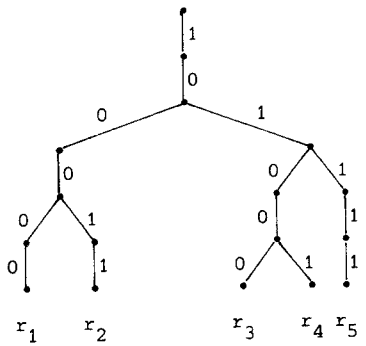
Tries are often used when there are records with variable-length keys also.

5.26. Despite the clarity of the file-structure, many programmers observed that tries can be somewhat inefficient in their use of storage.

One reason is that in entering keys we may not have made sufficient use of the observation that some strings could have had identical segments of bits which we should have followed first.

DEFINITION. A π-*trie* (where $\pi \in S_k$) is the storage-structure resulting after entering keys γ with bits ordered as $\gamma_{\pi(1)}\gamma_{\pi(2)}\cdots\gamma_{\pi(k)}$.

EXAMPLE. The 341256-trie for {001000,001011,101000,101001,111011} (see 5.25) is



It has only 12 internal nodes, as compared to the 123456-trie in 5.25 which

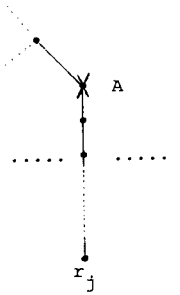
has 16.

It is unlikely that there will be a polynomial-time bounded algorithm to find an optimal π-trie for a file, since Comer & Sethi [20] recently proved the following interesting result

THEOREM. *The problem to determine a π-trie with the smallest number of internal nodes for a file is NP-complete.*

5.27. Given tries (or π-tries) can be *optimized* in various, simple ways.

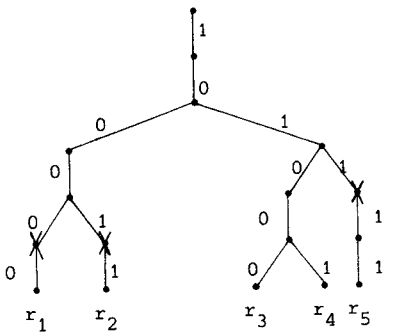
Observe, for instance, that one could eliminate chains of the form



(which Comer & Sethi call *leaf-chains*), and *prune* the tree in A at the earliest moment that r_j is uniquely identified.

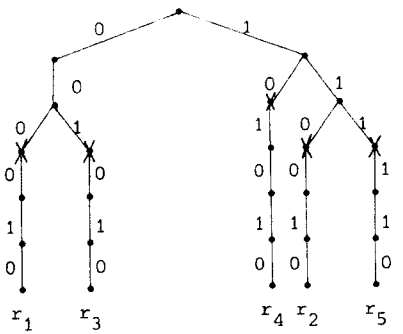
The idea is that the remaining bits of the key need not take up trie-storage and can be checked elsewhere.

EXAMPLE. The pruned 341256-trie for {001000,001011,101000,101001,111011} is



It seems to require an entirely different strategy to determine the best possible *pruned* π-trie for a file.

EXAMPLE. The 651234-trie for {001000,001011,101000,101001,111011} is

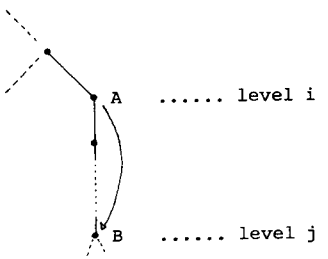


It has as many as 21 internal nodes, but the pruned version has only 5.

Comer & Sethi [20] proved also

THEOREM. *The problem to determine a pruned π-trie with the smallest number of internal nodes for a file is* NP-*complete.*

5.28. A next step would to collapse long chains



in the interior of the trie also, by adding an instruction at node A to skip over bits $\gamma_{\pi(i+1)},\ldots\ldots,\gamma_{\pi(j)}$ and to continue immediately with bit $\gamma_{\pi(j+1)}$.

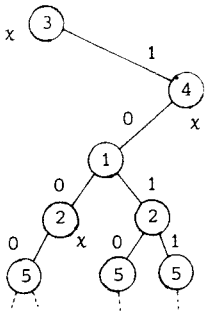
The idea was apparently first proposed by Morrison [48] in 1968, and

implemented in his system PATRICIA. Testing the skipped bits of the key afterwards avoids the need for ineffective trie-storage, and the technique leads to the most compact π-trie.

The problem to determine an "optimal" compacted π-trie is meaningless if the number of internal nodes is the only measure, and one should now also take the degree of balancing into account.

5.29. In a π-trie we could have inscribed the tested bit-positions in the corresponding nodes.

EXAMPLE.



Clearly in partial-match retrieval some bits in the input will be unspecified, and when a * is found one must descend down two branches (if there are two) simultaneously (and so on ..), thus implicitly defining what bucket is searched.

5.30. Carrying this idea one step further, Burkhard [15] recently formulated a type of generalised trie which seems an appropriate structure for partial-match retrieval from large indexed files.

A (k,s) *PM-trie* is a complete binary tree with 2^{k-s} leaves in which

(i) each leaf corresponds to a bucket;

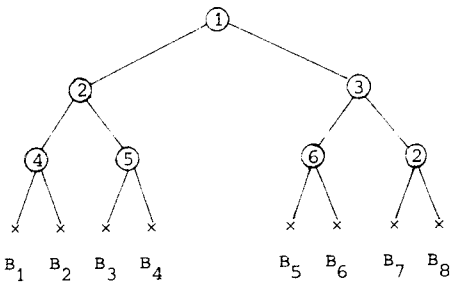
(ii) each internal node is labeled by one out of k possible bit-positions;

(iii) on no path from the root to a leaf there is a bit-position which is tested more than once.

(Thus each such path is like an s-star query.)

Technically each (k,s) PM-trie directly implies a (k,s) partial-match file-design in the sense of 5.23, but not conversely.

EXAMPLE.



is a (6,3) PM-trie.

To test how good a PM-trie is we shall measure how many buckets must be searched *in worst case* to answer a t-star query ($t \le k$).

The result is likely to depend on the way tests are distributed over the tree, and the task is to find "good" (k,s) PM-tries for use on files with k-bit entries.

EXAMPLE. The query ***010 in the given (6,3)-trie leads into 5 buckets.

The query *1**** leads into 5 buckets also, despite the fact that it is "almost unspecified".

5.31. In early 1975 Burkhard [14] first formulated and analysed a type of "good" (2n+1,n) PM-tries. We shall present a modified version which was shortly later suggested by Dubost & Trousse [24].

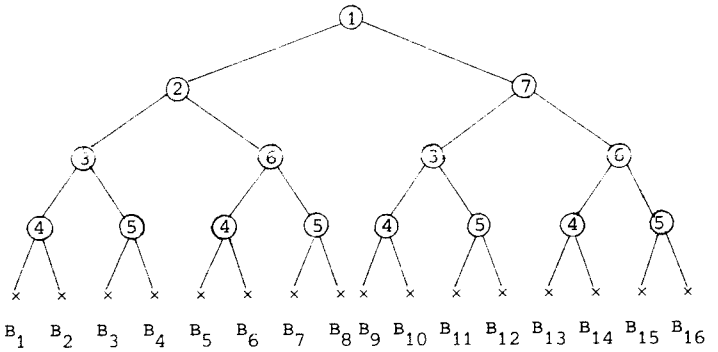
DEFINITION. A (2n+1,n) PM-trie is called a BDT-trie of order n if and only if

(i) the root (at level 1) has label 1;

(ii) for each node in level i ($1 \le i \le n$) the left-son is labeled $i+1$ and the right-son is labeled $2n+2-i$.

Observe that the tested bit-positions at the nodes may be computed while descending down the tree very easily.

EXAMPLE. The BDT-trie of order 3 is



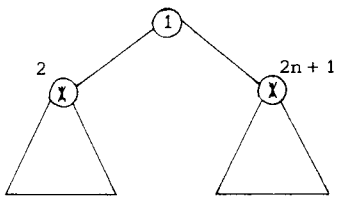
Observe that in a BDT-trie of order n all paths have length $n+1$, and there is an obvious regularity in the assignment of tests.

The labels of two brothers always add to $2n+3$.

Let T_n denote the BDT-trie of order n.

LEMMA. *For all* $n \ge 1$

T_n =

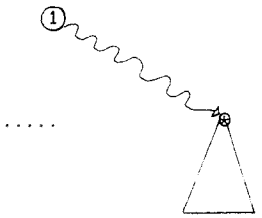


where each marked sub-tree is T_{n-1} *with* 1 *added to all labels (and* 2n *to the root of the right copy).*

5.32. Formulas for the worst case performance of BDT-tries were first given by Burkhard [14]. We shall outline an interesting proof-technique due to Dubost and Trousse.

The problem is how to visualize what buckets can be entered in a t-star query.

Suppose we first follow the trie-path until we hit the first bit-test which probes a star.



It means we narrowed the search for buckets to a small sub-trie.

However, each sub-trie (essentially) is a BDT-trie for sub-keys of the original file.

*Thus we reduced the task to finding the maximal number of buckets reached in a t-star query with a * in position* 1 (it will turn out to be an increasing function in n).

5.33. A *consistent assignment* with t stars in a BDT-trie is a complete marking of all (internal) nodes with "stars" and "dots" such that

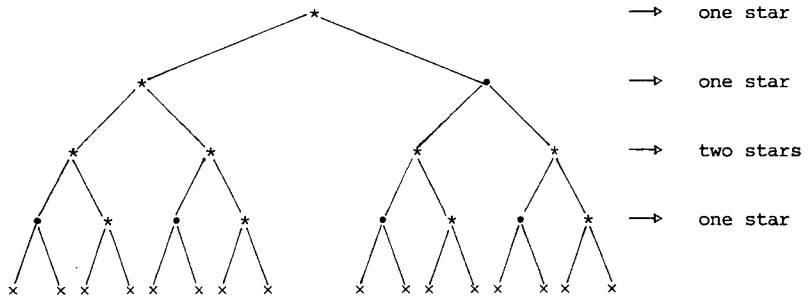
(i) the root is marked with * .

(ii) If a left-son in level i is marked with a *, then so are all left-sons in level i (which is reasonable since they probe the same bit-position).
If it happens, then the left-sons in level i are said to *contribute one star*.

(iii) If a right-son in level i is marked with a * then all are, and the right-sons are said to contribute one star also.

(iv) The number of contributed stars adds up to t.

EXAMPLE.



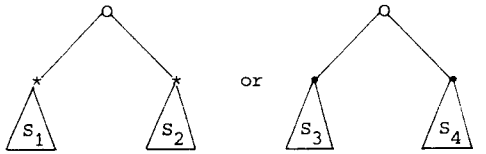
The search-structure in a BDT-trie of order 3 for a query ***•**** (where

dots mean *defined* bits). No matter what, 9 buckets will be visited on any 5-star query of this pattern.

The reason for introducing this concept is that *each t-star query implies a consistent assignment with t stars and conversely*.

5.34. For symmetry-reasons we may assume that if a level i > 1 contributes one star then it is contributed by the left-sons in that level.

Observe further that if two brothers are identically marked

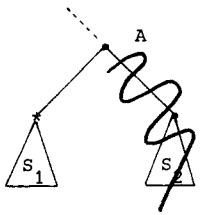


then the entire, corresponding sub-trees are identical in marking (i.e., $S_1 \equiv S_2$ and $S_3 \equiv S_4$ respectively).

5.35. Given a consistent assignment one can easily determine the largest possible number of buckets generated by a corresponding query.

Collapse and prune the tree according to the following rules.

- for each dotted node with not identically marked sons, purge the sub-tree of the dotted son



Because the bit at A is specified we would enter either S_1 *or* S_2. The sub-trees are identically marked except at their root, so entering S_1 will definitely lead into a larger number of buckets.

- for each dotted node with two identically marked sons, purge one of its subtrees also.

  The left- and right-subtrees are identical, and since we enter only one on the specified bit we need only count buckets in one sub-tree.

One could go one step further and collapse all chains of dotted nodes left.

The number of leaves in the resulting tree is exactly the number of buckets to be visited.

5.36. *It follows that the number of buckets visited by t-star queries of the same *-pattern is the same.*

5.37. Consider first how many buckets result when in the original, consistent assignment each level contributes *one* star.

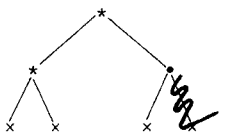
LEMMA. *For BDT-tries of order* n *this number is* F_{n+3}.

Proof. By induction.

For n = 0 we have

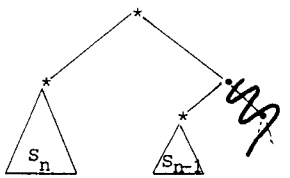
$\longrightarrow 2 \equiv F_3$

and for n = 1 we have (remember: only one star per level)



$\longrightarrow 3 \equiv F_4$

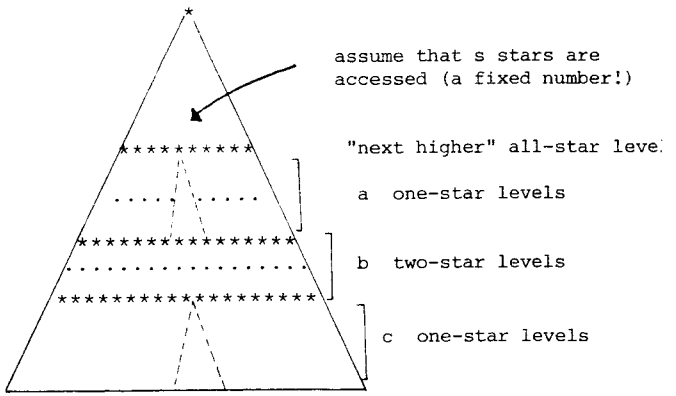
For n + 1 the marked trie-structure S_{n+1} of this form decomposes as



and the number of buckets adds up to $F_{n+3} + F_{n+2} = F_{n+4} \equiv F_{(n+1)+3}$. □

5.38. Next we consider consistent assignments in which each level contributes at least one star, and the number of levels indeed contributing two stars is assumed "fixed".

Examine the "lowest" layer of two-star levels in such an assignment



The number of buckets visited is

$$s \cdot 2^b \cdot F_{a+3} \cdot F_{c+3}$$

(using the previous lemma). Assuming b fixed, and observing that

$$s \cdot 2^b \cdot F_{a+3} \cdot F_{c+3} \leq s \cdot 2^b \cdot F_3 \cdot F_{a+c+3}$$

it follows that *the number of buckets is maximal if* a = 0. We immediately obtain

LEMMA. *In a consistent assignment with one- and two-star levels the number of buckets is largest if all two-star levels occur together and precede all one-star levels.*

5.39. In a general consistent assignment there can be dotted levels also, and we shall now analyse where they have to occur for obtaining the largest number of buckets.

Our argument here differs from the proof of Dubost and Trousse who

tried to "collapse" non-contributing levels in the "worst possible" way.

Let

$w_n(t)$ = the maximum number of buckets visited with a t-star query on a BDT-trie of order n.

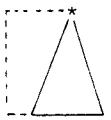
5.40. Observe first that the number of buckets for an assignment does not change if we add non-contributing levels at the bottom of the trie.

The idea is to make a given assignment "*worse*" by an exchange operation on the levels *without changing the order of the trie and without changing how many contributed stars there are*. (Thus the assignment remains a t-star query).

Do the following reductions whenever levels of the indicated form occur.

(i)

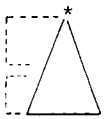
one-star and two-star levels



⇒

two-star levels

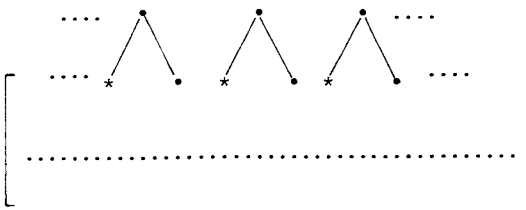
one-star levels



This is the simple transformation we proved in 5.38. It can be applied to subtrees also, *provided* the same transformation brings profit in all subtrees at this level (viz. when it is a two-star level).

(ii)

one-star and two star levels



It is easy to see that the worst case occurs if from the dotted level we

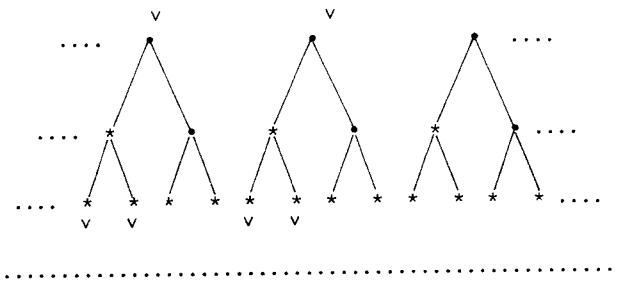
always proceed to the starred son. Thus we can apply (i) in the subtrees and should have all two-star levels immediately follow the initial one-star level

⇒

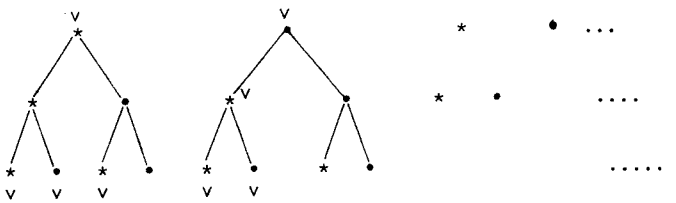
two-star levels

one-star levels

(iii)

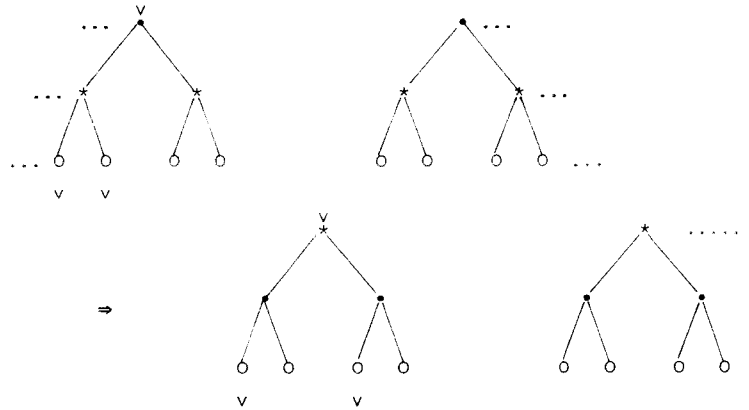


From each node in the dotted level we reach two stars two levels lower. Now borrow a star from the two-star level and make the dotted level into a one-star level:



Considering how one can enter the formerly dotted level to get a worst case now, it follows that at least as many buckets are entered now as we entered before.

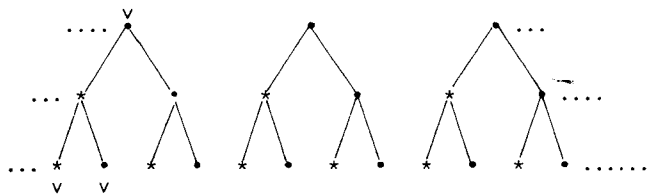
(iv)



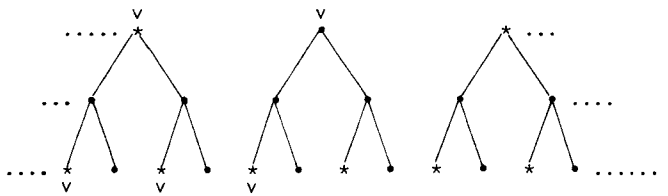
A two-star level and a dotted level can always be interchanged. (With the previous transformations *it now follows that for the worst case to occur all two-star levels must occur at the top of the tree*).

(v) Anywhere in the tree the number of consecutive dotted levels can be reduced to 1, provided we add the deleted levels at the bottom (to preserve the order of the tree)

(vi)

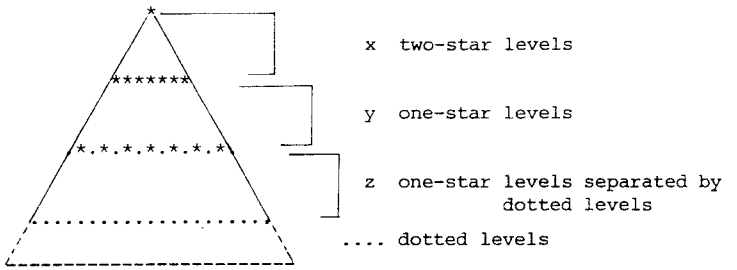


It is always "better" to have the dotted level occur in between the two one-star levels.



Note that this transformation together with the other tries to sift the dotted levels down and use them to separate as many one-star levels as it can.

5.14. If we perform these operations then we reduce a given assignment to a kind of normal form for the worst case:



where $1 + x + y + 2z \le n + 1$ (the number of levels)

and $1 + 2x + y + z = t$ (the number of stars).

5.42. We can now combine all information and prove Burkhard's theorem

THEOREM. *For t-star queries on a BDT-trie of order* n

$$w_n(t) = \begin{cases} 2^t & \text{for } 0 \le t \le n+1 - \lceil \frac{n}{2} \rceil \\ 2^{n+1-t} F_{2t-n+1} & \text{for } n+1 - \lceil \frac{n}{2} \rceil \le t \le n+1 \\ 2^{t-n-1} F_{2n+4-t} & \text{for } n+1 \le t \le 2n+1. \end{cases}$$

Proof. For the worst case stars must be distributed as in the tree of 5.41. We reach 2^x nodes in the last all-star level (which could be the root if $x = 0$), F_{y+1} stars and F_y dots from each of these in the last of the consecutive one-star levels, and in the last part of the tree we get into 2^{z+1} buckets from a star and 2^z buckets from a dot.

It follows that we must maximize the total of

$$2^x \cdot F_{y+1} \cdot 2^{z+1} + 2^x \cdot F_y \cdot 2^z = 2^{x+z}(2F_{y+1} + F_y)$$
$$= 2^{x+z} \cdot F_{y+3} \text{ buckets,}$$

under the condition that

$$x + y + 2z \le n$$

$$2x + y + z = t - 1. \qquad \square$$

5.43. In a subsequent generalization Burkhard [14] formulated another, more flexible type of PM-trie in which he abstracts the essential features that make BDT-tries work.

DEFINITION. A (2n+1,n) PM-trie is called a G-trie of order n if and only if
(i) any label j ($1 \le j \le 2n+1$) occurs in a unique level;
(ii) brothers carry different labels.

Burkhard proves that the expressions of 5.42 are upperbounds for order n G-tries in general.

Burkhard [16] also develops similar kinds of tries for files where keys are over an arbitrary alphabet Σ.

6. PATTERN-MATCHING

6.1. One may describe *pattern-matching* as the task of finding one or more sub-structures which meet a certain specification in given larger structures of some kind (usually in on-line manner).

Pattern-matching operations occur in lexical analysis, in searching library-files, in text-editing and in symbol-manipulation, and although it may seem easy we often need non-trivial data-organisations to perform the operations quickly without undesirable overhead.

6.2. Pattern-matching is undoubtedly the most powerful feature of the SNOBOL 4 programming language, and it often occurs in conjunction with *replacement* (see Griswold [34]).

Although we shall restrict the algorithms to strings and one-dimensional patterns, one could also consider pattern-matching problems for arrays and trees (see Karp, Miller, and Rosenberg [41]).

6.3. Let all strings considered be over the alphabet Σ with $\#\Sigma = \sigma$.

The simplest algorithm to find all length d substrings of

$$x_1 x_2 x_3 \ldots\ldots\ldots\ldots\ldots\ldots x_n \qquad (n \ge d)$$

makes use of a *shift-register* containing

$$\text{state} = |x_{i+1}|\sigma^{d-1} + |x_{i+2}|\sigma^{d-2} + \ldots + |x_{i+d}|$$

$$\longleftrightarrow \quad \ldots x_{i+1} x_{i+2} \ldots\ldots x_{i+d} \ldots$$

current position of scanner

Suppose that σ^d *buckets* $B_{\frac{\sigma^d-1}{\sigma-1}}, \ldots\ldots, B_{\sigma \cdot \frac{\sigma^d-1}{\sigma-1}}$ are allocated, one for each possible string of length d. Then one can collect all information from x in one scan as follows

```
state := |x_1|σ^(d-1) + |x_2|σ^(d-2) + ... + |x_d|;
enter 1 in B_state;
position := d;
while position < n do
position := position + 1;
state := state·σ + |x_position| (mod σ^d);
enter position - d+1 in B_state
od;
```

In the end $B_{|y_1|\sigma^{d-1} + \ldots + |y_d|}$ will contain all positions $i+1$ such that

$$x_{i+1} \ldots\ldots x_{i+d} = y_1 \ldots\ldots y_d.$$

The algorithm essentially traverses a finite state automaton with a very special transition-structure (known as a de Bruijn graph in combinatorics) which could have been computed and stored in advance. The approach is of reasonable complexity only if arithmetic modulo σ^d can be performed quickly.

6.4. More common pattern-matching algorithms of low complexity develop and use *positional information* of non-numeric nature.

A typical procedure for classifying and locating all length d substrings of x makes use of the following equivalence-relation

positions i and j of x are *s-equivalent* if and only if

$$x_i \cdots\cdots x_{i+s-1} = x_j \cdots\cdots x_{j+s-1}.$$

Our representation of equivalence-classes will differ from Karp, Miller, and Rosenberg [41], and simplify the final algorithm somewhat.

6.5. Let the *name* of an equivalence-class be the smallest element it contains. CLASS[i] = j means that position i belongs to the s-equivalence class named j.

The elements of an equivalence-class will be *linked* together *in ascending order* using NEXT. The end of a NEXT-chain will be marked by a negative integer which yields the beginning address of another class-chain, and 0 if all classes have been linked together.

One could enumerate all positions in equivalence classes with

```
position := 1;
repeat
alpha := CLASS[position];
print position;
while NEXT[position] > 0 do
position := NEXT[position];
print position
od;
end of class alpha;
position := - NEXT[position]
until position = 0;
```

6.6. The problem is how one can construct the s - equivalence classes in this representation quickly. The idea is an inductive approach based on the following result.

LEMMA. *For all* $r \le s$ *and* i *and* j, $i \sim_{r+s} j \longleftrightarrow i \sim_s j \;\&\; i+r \sim_s j+r$.

Proof. By overlap. □

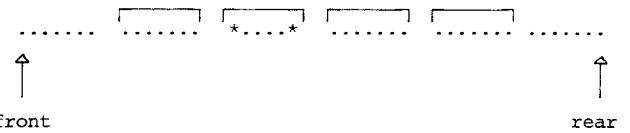
We can now show

THEOREM. *Given the representation of s-equivalence classes one can construct the* $r+s$*-equivalence classes for any* $r \le s$ *in linear time.*

Proof. Assume that we have n queues $Q_1,\ldots,Q_n$ available (one for each equivalence class).

Read out all s-equivalence classes (which yields the elements per class in ascending order) and *attach each position* i *so encountered at the rear of* Q_j, *where* j = CLASS[i+r] (provided $i+r+s-1 \le n$).

In the end each Q_j is either empty or made up of blocks



where each block simply consists of the items attached to Q_j while reading out an entire s-class. Note that each block can be identified because it is the longest stretch of elements with the same particular CLASS-value.

Observe now that i and j are in the same block if and only if $i \sim_s j$ and CLASS[i+r] = CLASS[j+r], thus by the lemma if and only if $i \sim_{r+s} j$. Thus the blocks exactly are the $r+s$-equivalence classes.

One can now obtain the representation of $r+s$-classes by reading out the queues block after block. The first element of a block by definition is its name (since it is smallest), and it can be retained and put into CLASS while passing through the block. The classes can be linked in the order in which the queues are emptied. (Start with the block containing 1, which must occur up-front somewhere.) □

6.7. It follows that

THEOREM. *One can construct the representation of s-equivalence classes for* x *in* $\sim \sigma \cdot n + n \log s$ *steps.*

Proof. The 1-equivalence classes follow by a simple scan of x in $\sim \sigma \cdot n$ steps. (There is one chain for each distinct symbol of x.) If the representation of the r-equivalence classes is known, then we can construct the 2r-equivalence classes in linear time. Thus the following algorithm will do

```
r := 1;
record the r-classes;
while 2r < s do
r := 2r;
record the r-equivalence classes
od;
record the (s-r)+r-classes from the r-classes;
```

□

6.8. To find all substrings of length d in x with the given algorithm works pleasantly in $\sim n \log d$ steps (assuming a fixed alphabet). A direct search for the classes with more than one element gives all repeated patterns of length d in about the same number of steps.

The same principle as in 6.7 can be used to show

THEOREM. *One can find all longest repeated substrings of* x *in* $\sim \sigma \cdot n + n \log n$ *steps.*

Proof. Karp, Miller, and Rosenberg [41] observed that the problem is equivalent to finding the *largest* r such that there is at least one r-class with more than one element.

Determine the 1-classes first in $\sim \sigma \cdot n$ steps, and then find r with binary search after a usual "doubling" procedure has located a lower bound d and upper bound 2d for it. □

One can develop similar algorithms based on s-equivalence to determine for instance whether x is of the form y^k for some $k \geq 2$ and y.

6.9. The idea of positional equivalences is clearly of great help in solving pattern-matching problems, but it seems to be efficient only when we search *all* patterns of some kind. Note that the algorithm in 6.8 does *not* simplify (for instance to complexity $\sim \sigma n$) in case we just want *any* longest repeated substring.

Another draw-back is that the algorithms become more inefficient (in storage and search-time) in case we must consider and "remember" several s-equivalences at the same time.

6.10. Weiner [65] proposed a very interesting idea for extracting exactly all the information you need to know from the various s-equivalences.

Let us assume that x always has a fixed marker at the end which is not occurring elsewhere

$$x_1 x_2 \cdot \cdot \cdot \cdot x_n \$.$$

DEFINITION. y is called the *substring-identifier* for position i if and only if it is the smallest u with the property that u occurs nowhere else in x except at position i.

EXAMPLE. The substring-identifiers for the various positions in

bbababba$

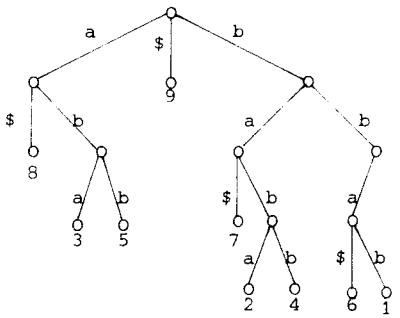
are

1	bbab
2	baba
3	aba
4	babb
5	abb
6	bba$
7	ba$
8	a$
9	$

The trick of adding $ at the end of x guarantees that *each position has a substring-identifier*, and one can observe that *for each* i *the substring-identifier is precisely equal to the length-r substring at* i *for the smallest* r *such that there are no other positions in the* r*-equivalence class containing* i.

6.11. Clearly no substring-identifier can be a proper prefix of another, and one can enter all substring-identifiers consistently in a σ+1-ary *position-tree*.

EXAMPLE. The position-tree for bbababba$ (the previous example) is



The path with y ends at the leaf with label i if and only if y is the substring-identifier of position i.

6.12. With the algorithm of Karp, Miller, and Rosenberg it follows that

THEOREM. *The position-tree for* x *can be constructed in* $\sim \sigma \cdot n \cdot L$ *steps, where* L *is the length of the longest substring identifier.*

Proof. Construct the s-equivalence classes for s = 1,2,...,L. Each time a new relation is constructed, record the current s-value for all positions which now form a singleton equivalence class. Use the table afterwards to enter the corresponding substring identifiers in the tree in $\sim \sigma L$ steps per entry. □

The theorem is not the best possible result though. A position-tree with many internal nodes (as many as $\sim n^2$ are possible even though it can only have n leaves) must contain long straight chains

which one can compress, resulting in a *compacted position-tree* of only $\sim O(n)$ nodes.

Weiner [65] proved that one may construct the compacted tree directly in only *linear* time. At the SWAT-meeting in Iowa City it was promptly called "the algorithm of 73", because it took that many pages to explain. (Easier descriptions are now available, see Aho, Hopcroft, and Ullman [3].)

6.13. Once the position-tree is known all kind of pattern-matching questions can be answered quickly.

Consider the problem of determining a longest repeated substring y of x. Any such y must be the *proper* prefix of some substring-identifier. Thus one must choose a y which leads to a lowest interior node in the position-tree (and any such y qualifies).

EXAMPLE. A longest repeated substring of bbababba$ (see 6.11) is bab.

To find a longest common (contiguous) substring of x and y one can construct the position-tree for

x ¢ y $

and solve the task by searching for a lowest interior node with at least one son representing a position in x and one son representing a position in y.

6.14. The position-tree can be used also to solve the authentic pattern-matching problem to determine whether given string y is a substring of x.

Let

y = y[1]y[2] .... y[m]

x = x[1]x[2] .....x[n]

with $m \le n$.

Before we see a direct algorithm for this problem it is of interest to look at the method SNOBOL 4 uses

```
position := 0;
check := 1;
while position < n - m + 1 do
```

```
while x[position+check] = y[check] do
check := check + 1
until check > m;
if check > m then match found & exit fi;
position := position + 1;
check := 1
od;
```

The method may require in worst case ~ n·m steps. Morris & Pratt [47] apparently first realized that these is a *linear* time pattern-matching algorithm. At the same time ~ 1970 the result followed from a general, linear time algorithm for simulating arbitrary 2-way deterministic pushdown-automata found by Cook [21], which prompted Knuth & Pratt to write a report entitled "automata-theory can be useful" [42].

6.15. Consider how one might improve the previous algorithm.

Suppose the algorithm has been in action for a while

$$\cdots\cdots\ \overbrace{x_{k-j+1}\cdots\cdots x_k} \qquad \text{(match)}$$

$$y_1\cdots\cdots\cdots y_j$$

and now finds that $x_{k+1} \neq y_{j+1}$

$$\cdots\cdots\cdots\cdots\cdots\cdots\overbrace{}^{x_k}\ \ \overset{\downarrow}{x_{k+1}}$$

$$y_1\cdots\cdots\cdots y_j \quad y_{j+1}$$

Instead of trying again from the very beginning one should *continue at the largest* i *such that*

$$\cdots\cdots\cdots\cdots\cdots\cdots\overbrace{}^{x_k}\ \ \overset{\downarrow}{x_{k+1}}$$

$$y_1\cdots\cdots\cdots y_i \quad y_{i+1} \qquad \underline{\text{with}}\ x_{k+1} = y_{i+1}$$

which is exactly the largest i *such that*

$$y_1\cdots\cdots\cdots\cdots\cdots\cdots\overbrace{}^{y_j}\ \cdots$$

$$y_1\cdots\cdots\cdots y_i \quad y_{i+1} \qquad \underline{\text{with}}\ x_{k+1} = y_{i+1}$$

Candidate i's can therefore be determined based solely on y.

6.16. The following concept now appears relevant.

DEFINITION. The *overlap-identifier* h(j) of position j in y is the largest i < j such that

$$\cdots\cdots\cdots\cdots\cdots\cdots\overbrace{}^{y_j}$$

$$\overbrace{\cdots\cdots\ y_i}$$

Observe that always h(1) = 0. Let h(0) = 0 for consistency.

EXAMPLE. The overlap-identifiers of babaaba are

b	a	b	a	a	b	a
0	0	1	2	0	1	2

LEMMA. *The overlap-identifiers can be determined in ~ m steps.*

Proof. Suppose h(1),...,h(j) have been determined and we want h(j+1) next. We know

$$\cdots\cdots\cdots\cdots\cdots\cdots\overbrace{}^{y_j}\ \ y_{j+1}$$

$$\cdots\cdots\cdots\ y_{h(j)}$$

and can put h(j+1) = h(j)+1 in case $y_{j+1} = y_{h(j)+1}$, but otherwise we must back up further

$$\cdots\cdots\cdots\cdots\cdots\cdots\overbrace{}^{y_j}\ \ y_{j+1}$$

$$\cdots\cdots\cdots\cdots\cdots y_{h(j)}$$

$$\cdots\cdots\cdots y_{h(h(j))}$$

$$\vdots$$

and further until such a match is found. The algorithm can be formulated as

```
h(1) := 0;
j := 1;
while j < m do
```

```
i := h(j);
while y[j+1] ≠ y[i+1] & i > 0 do
i := h(i)
od;
h(j+1) := if y[j+1] ≠ y[i+1] then 0 else i+1;
j := j+1
od;
```

Consider the total cost of this algorithm as we keep sliding a copy of y across along y itself. As long as y[j+1] and y[i+1] do not match we can charge the cost for a step to position y[j-i+1] (and note here that i steadily decreases). If there is a match we charge the cost to y[j+1] and continue with an increased value of j. No position can get charged more than twice. Thus the algorithm takes only a linear number of steps. □

6.17. Once the overlap-identifiers are determined we can enact the *on-line* pattern-match across x. The algorithm (see 6.15) becomes

```
    match := 0;
    scan := 1;
    while scan ≤ n do
    if x[scan] = y[match+1] then goto L fi;
    while x[scan] ≠ y[match+1] do
    match := h(match)
    until match = 0;
L:  match := if x[scan] = y[match+1] then match+1 fi;
    if match = m then pattern found & exit fi
    scan := scan+1
    od;
```

By changing the "exit" into

```
record the match at scan;
match := h(match);
```

one can use the algorithm to locate *all* (possibly overlapping) occurrences of y. Observe that each time around the loop we either match a next posi-

tion of x or move the beginning of y up, thus "pointing" at a position of x for the second time (but we will never point at it later after we passed it). Thus the algorithm requires time bounded by ~ 2n.

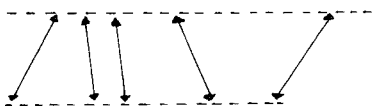
Adding the work we obtain

THEOREM. *One can determine whether* y *is a substring of* x *in* ~ n + m *steps.*

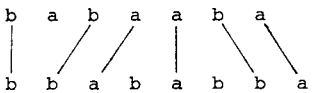
6.18. Hirschberg [37] considered an interesting variant of the longest common substring problem, now known as *the maximal common subsequence problem.*

DEFINITION. z is a maximal common subsequence of x and y if and only if
(i) $z \le x$ & $z \le y$;
(ii) there is no w with $|w| > |z|$ such that $w \le x$ & $w \le y$.

The problem typically occurs when a maximal correspondence between two data-arrays must be found, and is encountered in comparing molecular structures and for instance in *problem-solving*. Learning systems classify problems by a series of features in order of decreasing importance, and to find out to what extent two problems are similar (or "analogical") one tries to establish the best possible sequence of common features



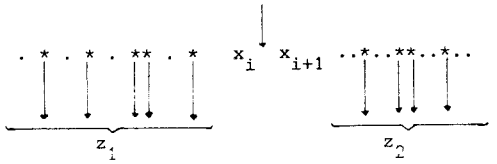
EXAMPLE. A longest common subsequence of babaaba and bbababba is bbaaba



The problem poses some interesting questions of time- and storage-utilization (the "simplicity" of the task is again deceptive). We shall prove Hirschberg's result (in a somewhat different presentation) that there is a data-organisation such that

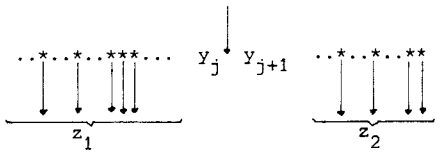
THEOREM. *The maximal common subsequence of* x *and* y *can be found in* ~ n m *steps and* ~ n + m *space.*

6.19. If z is to be a longest common sequence then for each position i in x



where $z_1 z_2 = z$

there must be a corresponding "split"-position j in y



such that there is no position k in y for which the maximal common subsequence of

$$x[1] \ldots\ldots x[i] \quad \text{and} \quad y[1] \ldots\ldots y[k]$$

concatenated with the maximal common subsequence of

$$x[i+1] \ldots\ldots x[n] \quad \text{and} \quad y[k+1] \ldots\ldots y[m]$$

is longer than z.

This criterion immediately suggests an algorithm to find z. Split x in two parts

$$x[1] \ldots\ldots x[\lfloor \tfrac{n}{2} \rfloor] \quad \text{and} \quad x[\lfloor \tfrac{n}{2} \rfloor + 1] \ldots\ldots x[n]$$

and find a position j such that the longest common subsequences of

$$x[1] \ldots\ldots x[\lfloor \tfrac{n}{2} \rfloor] \quad \text{and} \quad y[1] \ldots\ldots y[j]$$

and

$$x[\lfloor \tfrac{n}{2} \rfloor + 1] \ldots\ldots x[n] \quad \text{and} \quad y[j+1] \ldots\ldots y[m]$$

determined by a recursive application of the algorithm concatenate to a longest possible string.

6.20. The common trick in *dynamic programming* learns to compute the value of j quickly first (without actually computing what the common subsequences

are), and then do a specific recursive call to do the hard work of finding the subsequences.

The following concept is now needed.

DEFINITION. For $i \le j$ and $k \le \ell$ let $M_{ik}[j,\ell]$ be the length of the maximal common subsequence of

$$x[i] \ldots\ldots x[j] \quad \text{and} \quad y[k] \ldots\ldots y[\ell] \ .$$

LEMMA. *One can compute* $M_{ik}[j,\ell]$ *in* $\sim (j-i)(\ell-k)$ *time and only* $\sim (\ell-k)$ *space.*

Proof. One can determine $M_{ik}[i,k], M_{ik}[i,k+1], \ldots\ldots, M_{ik}[i,\ell]$ by comparing symbol after symbol with x[i] in $\sim \ell - k$ steps.

Assuming that $M_{ik}[s-1,k], M_{ik}[s-1,k+1], \ldots\ldots, M_{ik}[s-1,\ell]$ have been determined, one can compute a next row with

```
M_ik[s,k] := if y[k] occurs in x[i]...x[s] then 1 else 0;
for t := k + 1 to ℓ do
M_ik[s,t] := if x[s] = y[t] then M_ik[s-1,t-1]+1
             else MAX(M_ik[s-1,t],M_ik[s,t-1]);
od;
```

One must compute $j - i + 1$ rows. □

6.21. By building $M_{ik}[j,\ell]$ from "right-to-left" one can similarly compute rows

$$M_{ik}[j,\ell], M_{i\ k+1}[j,\ell], \ldots\ldots, M_{i\ell}[j,\ell]$$

in the same time and space bounds.

6.22. We can now develop the recursive procedure (due to Hirschberg) FIND[i,j,k,ℓ,α] determining α as a longest common subsequence of

$$x[i] \ldots\ldots x[j] \quad \text{and} \quad y[k] \ldots\ldots y[\ell]$$

```
procedure FIND[i,j,k,ℓ,α]
begin
if i = j or k = ℓ then determine α directly fi;
half := i + (j-i)/2;
compute row M_ik[half,k],M_ik[half,k+1],.....,M_ik[half,ℓ];
compute row M_(half+1 k)[j,ℓ],M_(half+1 k+1)[j,ℓ],.....,M_(half+1 ℓ)[j,ℓ];
split := an index k ≤ t ≤ ℓ such that
        M_ik[half,t] + M_(half+1 t+1)[j,ℓ]
        is maximal (over all choices of t);
FIND[i,half,k,split,α_1];
FIND[half+1,j,split+1,ℓ,α_2];
return α := α_1α_2
end;
```

It is easy to make the procedure iterative and to see that only linear space is needed.

In the time-analysis we shall use that for integers $x,y \geq 1$ and $A \geq 2B + C$ (A,B, and C positive)

$$A\ xy \geq B(x+y) + C.$$

Assume as an induction-hypothesis that FIND[i,j,k,ℓ,α] requires only

$$a_1(j-1)(\ell-k) + a_2[(j-i)+(\ell-k)] + a_3$$

time.

Choosing $a_2 > 1$ and $a_3 \geq 1$ appropriately it holds when $i = j$ and/or $k = \ell$.

For the induction-step we shall assume (by 6.20 and 6.21) that the computation of a row $M_{ik}[j,*]$ or $M_{i*}[j,\ell]$ takes

$$b_1(j-i)(\ell-k) + b_2[(j-i)+(\ell-k)] + b_3$$

time.

Then the time for FIND[i,j,k,ℓ,α] is bounded by

$$\begin{aligned}
&\mathrm{const}_1 + 2b_1(j-i)(\ell-k) + 2b_2[(j-i)+(\ell-k)] + 2b_3 + \\
&+ \mathrm{const}_2(\ell-k) + a_1\frac{j-i}{2}(\ell-k) + a_2[(j-i)+(\ell-k)] + a_3 + \\
&+ \mathrm{const}_3 =
\end{aligned}$$

$$\begin{aligned}
&= \left(\frac{a_1}{2} + 2b_1\right)(j-i)(\ell-k) + (a_2+2b_2+\mathrm{const}_2)[(j-i)+(\ell-k)] + \\
&+ (a_3+2b_3+\mathrm{const}_1+\mathrm{const}_3) \leq \\
&\leq \left(\frac{a_1}{2} + 2b_1+2a_2+4b_2+2\,\mathrm{const}_2 + a_3+2b_3+\mathrm{const}'\right)(j-i)(\ell-k).
\end{aligned}$$

If we choose a_1 such that

$$a_1 \geq \frac{a_1}{2} + 2b_1 + 2a_2 + 4b_2 + a_3 + 2b_3 + \mathrm{const}''$$

(which we can) then the assumption is consistent. □

6.23. One may now ask if there is perhaps a faster than quadratic algorithm to find maximal common subsequences. Aho, Hirschberg, and Ullman [2] proved under weak assumptions that in general there isn't.

Questions of *optimality* require that we somehow delimit the class of algorithms under consideration.

Consider straight-line programs which extract information from the input only by *cross-comparisons* and count how many queries

"is x[p] = y[q]"

are needed in worst case to find a longest common subsequence.

Note that if $x^{(1)}$ and $y^{(1)}$ and $x^{(2)}$ and $y^{(2)}$ yield the same answer query after query such algorithms generate the same computation in both instances and locate the longest common subsequence in identical manner. We shall exploit this fact to prove that if such algorithms query the input only a "few" times they cannot distinguish different inputs correctly.

6.24. We can immediately show

LEMMA. *Any algorithm for solving the longest common subsequence problem needs* $\geq n + m - 1$ *queries in worst case.*

Proof. Consider the sequence of queries (*all yielding no*) generated on

$$x = \underbrace{0\ 0\ \ldots\ldots\ 0}_{n} \quad \text{and} \quad y = \underbrace{1\,1\ \ldots\ldots\ 1}_{m}$$

which have an "empty" solution.

Construct a *bipartite* graph G connecting the positions queried in each step.

If G has at least two connected components, then change the symbols of the positions of x and y in one component to 1 and 0 (respectively). The queries don't change and the algorithm still yields "empty" despite that there now is a non-trivial common subsequence.

It follows that G must be a connected graph with $n + m$ vertices. □

6.25. For $\sigma = 2$ (say alphabet $\Sigma = \{0,1\}$) the given bound cannot be improved. The positions of x and y can be identified in only $n + m - 1$ comparisons as follows.

Think of x[1] as " 0 " and ask "is x[1] = y[1]" to determine if we can think of y[1] as " 0 " also or should consider it as " 1 ". Identify x[2].....x[n] by asking "is x[*] = y[1]", and y[2].....y[m] by asking "is x[1] = y[*]".

For $\sigma > 2$ it doesn't work, and we can indeed improve the lemma to a quadratic bound.

THEOREM. *For* $\sigma > 2$ *any algorithm for solving the longest common subsequence problem requires* $\geq nm$ *queries in worst case.*

Proof. Construct the same graph G as in 6.24, and suppose it has $< nm$ lines. Then there must be a pair p,q *not* connected and thus not queried in the algorithm.

Change x[p] and y[q] into " 2 " (which we can because a third symbol is available). It does *not* affect the answer of any query asked and the algorithm still comes up with "empty" despite that the correct result must be a non-trivial string. □

REFERENCES (some references are not explicitly cited in the text).

[1] ADEL'SON-VEL'SKII, G.M., and Y.M. LANDIS, An algorithm for the organisation of information, Soviet Math. Dokl. 3 (1962) 1259-1262.

[2] AHO, A.V., D.S. HIRSCHBERG, and J.D. ULLMAN, Bounds on the complexity of the longest common subsequence problem, Conf. Record 15th Annual IEEE Symp. Switching and Automata Theory, New Orleans (1974) 104-109, also: JACM 23 (1976) 1-12.

[3] AHO, A.V., J.E. HOPCROFT, and J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass. (1974).

[4] AHO, A.V., J.E. HOPCROFT, and J.D. ULLMAN, On finding lowest common ancestors in trees, Proc. 5th Annual ACM symp. Theory of Computing, Austin (1973) 253-265, also: SIAM J. Computing 5 (1976 115-132.

[5] AMBLE, O., and D.E. KNUTH, Ordered hash-tables, Stanford University, Stanford (1973) STAN-CS-73-367, also: Computer J. 17 (1974) 135-142.

[6] ARDEN, B.W., B.A. GALLER, and R.M. GRAHAM, An algorithm for equivalence declarations, CACM 4 (1961) 310-314.

[7] BAYER, R., Binary B-trees for virtual memory, Proc. ACM SIGFIDET Workshop, San Diego (1971) 219-235.

[8] BAYER, R., Symmetric binary B-trees: data-structures and maintenance algorithms, Acta Inf. 1 (1972) 290-306.

[9] BENTLEY, J.L. and W.A. BURKHARD, Heuristics for partial-match retrieval data-base design, Inf.Proc. Letters 4 (1976) 132-135.

[10] BLOOM, B.H., Space/time trade-offs in hash-coding with allowable errors CACM 13 (1970) 422-426.

[11] BOBROW, D.G., A note on hash-linking, CACM 18 (1975) 413-415.

[12] BOOTH, K.S., and G.S. LUEKER, Linear algorithms to recognize interval-graphs and test for the consecutive ones property, Proc. 7th Annu ACM Symp. Theory of Computing, Albuquerque (1975) 255-265.

[13] BOOTH, K.S., and G.S. LUEKER, PQ-tree algorithms, Dept. of Electr. Engineering and Computer Sc., Univ. of California, Berkeley (1975) (preprint).

[14] BURKHARD, W.A., Hashing and trie-algorithms for partial match retrieval, Dept. of Applied Physics and Information Sc., Unif. of California San Diego, La Jolla (1975) TR-2.

[15] BURKHARD, W.A., Partial-match queries and file-design, Proc. ACM Conf. on Very Large Data-bases (1975) 523-525.

[16] BURKHARD, W.A., Associative retrieval trie hash-coding, Dept. of Applied Physics and Information Sc., Univ. of California/

San Diego, La Jolla (1975) TR-6, also: Proc. 8th Annual ACM Symp. Theory of Computing, Hershey (1976) (to appear).

[17] CODD, E.F., A Relational model of data for large shared data banks, CACM 13 (1970) 377-387.

[18] CODD, E.F., A data-base sublanguage founded on the relational calculus, Proc. ACM SIGFIDET Workshop on data-description, access, and control (1971).

[19] COFFMAN, E.G., and J. EVE, File-structures using hashing functions, CACM 7 (1970) 427-432,436.

[20] COMER, D., and R. SETHI, NP-completeness of tree-structured index minimization, Dept. of Computer Sc., Pennsylvania State Univ., College Park (1975) (preprint).

[21] COOK, S.A., Linear time simulation of deterministic two-way pushdown automata, Proc. IFIP Congress 71, TA-2, North-Holland Publ. Comp., Amsterdam (1971) 172-179.

[22] DATE, C.J., *An introduction to data-base systems,* the Systems Progr. Series, Addison-Wesley, Reading, Mass. (1975).

[23] DE LA BRIANDAIS, R., File searching using variable length keys, Proc. Western Joint Computer Conf. (1959) 295-298.

[24] DUBOST, P., and J-M. TROUSSE, Software implementation of a new method of combinatorial hashing, Stanford University, Stanford (1975) STAN-CS-75-511.

[25] ENGLES, R.W., A tutorial on data-base organisation, Annual Review in Automatic Progr. 7 (1972) 1-64.

[26] EVE, J., On computing the transitive closure of a relation, Stanford University, Stanford (1975) STAN-CS-75-508.

[27] FISCHER, M.J., Efficiency of equivalence algorithms, in: R.E. MILLER, and J.W. THATCHER (ed.), *Complexity of Computer Computations,* Plenum Press, New York (1972) 153-168.

[28] FLOYD, R.W., and A.J. SMITH, A linear time two-tape merge, Stanford University, Stanford (1972) STAN-CS-72-285, also: Inform. Proc. Letters 2 (1974) 123-125.

[29] FOSTER, D.V., Elementary file organisations: a survey, Duke University, Durham (1975) CS-1975-6.

[30] FREDKIN, E., Trie memory, CACM 3 (1960) 490-499.

[31] FREDMAN, M.L., Two applications of a probabilistic search technique: sorting X + Y and building balanced search trees, Proc. 7th Annual ACM Symp. Theory of Computing, Albuquerque (1975) 240-244.

[32] GALLER, B.A., and M.J. FISCHER, An improved equivalence algorithm, CACM 7 (1964) 301-303, 506.

[33] GEHANI, N., private communication, SUNY/Buffalo, Amherst (1976).

[34] GRISWOLD, R., *String and list processing in SNOBOL 4: techniques and applications,* PH Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, N.J. (1974).

[35] GUSTAFSON, R.A., A randomized combinatorial file-structure for storage and retrieval systems, Ph.D. thesis, Univ. of South Carolina (1969).

[36] HILDEBRAND, P., and H. ISBITZ, Radix-exchange: an internal sorting method for digital computers, JACM 6 (1959) 156-163.

[37] HIRSCHBERG, D.S., A linear space algorithm for computing maximal common subsequences, CACM 18 (1975) 341-343.

[38] HOPCROFT, J.E., and R.M. KARP, An algorithm for testing the equivalence of finite automata, Cornell University, Ithaca (1971) TR-71-114.

[39] HOPCROFT, J.E., and J.D. ULLMAN, Set merging algorithms, SIAM J. Computing 2 (1973) 294-303.

[40] HORVATH, E.C., Efficient stable sorting with minimal extra space, Proc. 6th Annual ACM Symp. Theory of Computing, Seattle (1974) 194-215.

[41] KARP, R.M., R.E. MILLER, and A.L. ROSENBERG, Rapid identification of repeated patterns in strings, trees, and arrays, Proc. 4th Annual ACM Symp. Theory of Computing, Denver (1972) 125-136.

[42] KNUTH, D.E., and V.R. PRATT, Automata theory can be useful, Stanford University, Stanford (1971) STAN-CS-71- .

[43] KNUTH, D.E., *The art of computer programming* I: *Fundamental algorithms*, Addison-Wesley, Reading, Mass. (1969).

[44] KNUTH, D.E., *The art of computer programming* III: *Sorting and searching*, Addison-Wesley, Reading, Mass. (1973).

[45] MAURER, H.A., and D. WOOD, Zur Manipulation von Zahlenmengen, Inst. f. Angew. Informatik u. formale Beschreibungsverfahren, Univ. Karlsruhe (1975) Bericht 34.

[46] MAURER, H.A., TH. OTTMAN, and H.W. SIX, Implementing dictionaries using binary trees of very small height, Inst. f. Angew. Informatik u. formale Beschreibungsverfahren, Univ. Karlsruhe (1975) Bericht 37.

[47] MORRIS, J.H., and V.R. PRATT, A linear time pattern-matching algorithm, Computing Centre, Univ. of California, Berkeley (1970) TR-40.

[48] MORRISON, D.R., PATRICIA-practical algorithm to retrieve information coded in alphanumeric, JACM 15 (1968) 514-534.

[49] OTTMAN, TH., and H.W. SIX, Eine neue Klasse von Ausgeglichenen Binärbäumen, Inst. f. Angew. Informatik u. formale Beschreibungsverfahren, Univ. Karlsruhe (1975) Bericht 35.

[50] PORTER, T., and I. SIMON, Random insertion into a priority queue structure, Stanford University, Stanford (1974) STAN-CS-74-460.

[51] RIVEST, R.L., On self-organising sequential search heuristics, Conf. Record 15th Annual IEEE Symp. Switching and Automata Theory, New Orleans (1974) 122-126, also: CACM 19 (1976) 63-67.

[52] RIVEST, R.L., On hash-coding algorithms for partial - match retrieval, Conf. Record 15th Annual IEEE Symp. Switching and Automata Theory, New Orleans (1974) 95-103.

[53] RIVEST, R.L., Partial-match retrieval algorithms, SIAM J. Computing 5 (1976) 19-50.

[54] ROBERTS, D.C., File organisation techniques, in: *Adv. in Computers* 12 (1972) 115-174.

[55] SPITZEN, J., and B. WEGBREIT, The verification and synthesis of data-structures, Acta Inf. 4 (1975) 127-144.

[56] TARJAN, R.E., Efficiency of a good but not linear set union algorithm, JACM 22 (1975) 215-225.

[57] TARJAN, R.E., Applications of path-compression on balanced trees, Stanford University, Stanford (1975) STAN-CS-75-512.

[58] TRABB PARDO, L., Stable sorting and merging with optimal space and time bounds, Stanford University, Stanford (1974) STAN-CS-74-470.

[59] VAN EMDE BOAS, P., Preserving order in a forest in less than logarithmic time, Proc. 16th Annual IEEE Symp. Foundations of Computer Sc., Berkeley (1975) 75-84.

[60] VAN LEEUWEN, J., On finding lowest common ancestors in less than logarithmic average time, Symp. New Directions and Recent Results in Algorithms and Complexity, Carnegie-Mellon Univ., Pittsburgh (1976) 107.

[61] VAN LEEUWEN, J., *The complexity of data-organisation*, 2nd Adv. Course on Foundations of Computer Science, Amsterdam (1976).

[62] VAN LEEUWEN, J., On the construction of Huffman-trees, in: S. MICHAELSON & R. MILNER (ed), 3rd *Int. Colloq. on Automata/languages/Programming*, Edinburgh (1976), 382-410.

[63] WARREN, H.S., A modification of Warshall's algorithm for the transitive closure of binary relations, CACM 18 (1975) 218-220.

[64] WARSHALL, S., A theorem on Boolean matrices, JACM 9 (1962) 11-12.

[65] WEINER, P., Linear pattern-matching algorithms, Conf. Record 14th Annual IEEE Symp. Switching and Automata Theory, Iowa City (1973) 1-11.

[66] WIRTH, N., *Algorithms + data-structures = programs*, PH Series in Automatic Progr., Prentice-Hall, Englewood Cliffs, N.J. (1976).