stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 89/77 OKTOBER

D. GRUNE

CHOOSING A TAG-LIST ALGORITHM FOR A COMPILER WITH SPECIAL APPLICATION TO THE ALEPH COMPILER

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A10

ACM-Computer Review-Categories: 4.12

Choosing a Tag-list Algorithm for a Compiler with Special Application to the ALEPH Compiler

by

D. Grune

ABSTRACT

Requirements for a tag-list algorithm are formulated. Starting from a very general tag-list algorithm 18 practical versions are developed and their merits judged. Although the final choice (binary search in a diluted table) depends on the details of the application, the main part of this article is not devoted to that final choice itself but rather to ways of reaching it.

KEYWORDS & PHRASES: Compiler design, portability, symbol-table techniques

---

*) This report will be submitted for publication elsewhere.

0. Introduction.

This article is a report on the considerations that went into the choosing of a tag-list algorithm for the ALEPH compiler of the Mathematical Centre, Amsterdam.

Since almost half of the symbols in an ALEPH program are tags (the general pattern being separator, tag, separator, tag, etc.) the implementation of the tag-list algorithm merits some thought.

Much has been written on the question of what is the best tag-list algorithm (see e.g. [1], [2], [3] and [4]). Although the final choice depends on the details of the application, the main part of this article is not devoted to that final choice itself but rather to ways of reaching it.

For the non-ALEPH reader the following is relevant.

ALEPH is a machine-independent language mainly used for compiler writing. The compiler for it is written in ALEPH and bootstrapped from older versions. Two consequences of its machine-independence have influenced this article:

- we do not know the integer values of the characters on the target machine, which has its bearing on hash functions,
- we have no detailed information about the data structure used, so that we have no easy way to add a few bits if the algorithm needs them.

ALEPH has only two built-in data structures, integers and stacks of integers. Both exist in a read/write and a read-only variety. The stacks can also be used as arrays, i.e., they allow direct access through indexing.

Characters are treated as integers. Strings are implemented on an operational basis:

- There exists a routine 'pack string' which will pack a sequence of characters on a stack into a (hopefully smaller) data structure on a second stack; the precise structure of such a packed string is unknown to the ALEPH programmer. The process is reversed by 'unpack string'.
- The maximum length of a packed string is 'max string length', the exact value of which may change from machine to machine.
- Two packed strings can be compared lexicographically by calling 'compare string' which yields a three-way answer smaller/equal/larger.
- If need be the n-th character of a packed string may be obtained through a call of 'string elem'.

1. The requirements.

We shall first make a precise list of requirements and then evaluate in the light of those requirements various implementation techniques, some widely published, some of a more exotic nature.

1.1. Requirement A, the effect.

The algorithm should, when offered a string S, return a pointer to an information block containing a pointer to a string that is equal to S, and when offered the string S again, return this same pointer. This is a slight complication with respect to the algorithms treated in literature where generally a pointer to the string itself is supposed to suffice. It implies that if the algorithm moves entries around, it cannot incorporate the information block in its data structure and the extra link-location needed per entry will be charged against it.

1.2. Requirement B, the nature of the tags.

Tags must be allowed to have sufficient length. Since the tags will be packed by 'pack string', the natural maximum tag length is 'max string length'. Since 'pack string' should, in any reasonable implementation, be able to pack a line of text, we can be sure that 'max string length' has a value large enough not to bother us in this case. So we add to the specifications of the compiler:

- The maximum length of a tag is 'max string length' characters.

No hard upper limit should be set on the number of tags, other than those imposed by integer capacity, memory size, etc. Although the compiler is expected to handle programs with about 1000 tags (order of magnitude), that same compiler should be able to handle 100000 tags, slower and in much more memory but without the need to change some internal constants.

Some tag-list algorithms work less efficiently if the tags are ordered in the first place. Now ordered tags seem unlikely in a program. However, some programmers place the ALEPH rules in alphabetic order to facilitate maintenance; and ordered sequences of tags can occur in sequences of pointer initializations of objects that are in order. (like /a/: letter a, /b/: letter b, etc.). We shall therefore take into consideration the degeneracy caused by ordered input.

We do not take into account the effect of an accidental or intentional "frustration sequence", i.e., a seemingly random sequence that conspires with the hash-function or the randomization scheme so as to give a pathological result. (Schemes that avoid degeneracy at all costs are generally very

expensive).

1.3. Requirement C, efficiency.

As mentioned before, the algorithm will be heavily used. So its efficiency with respect to time is important. Since the tag list has to be in core all the time, and since minimum storage is one of our design goals, its efficiency with respect to memory is important.

It is tempting (and traditional) to measure the time efficiency of a tag-list algorithm in terms of number of string comparisons. It should, however, be pointed out that a simple trick can reduce the number of string comparisons to one per entered string. A hash function is designed that maps tags onto the full range of machine integers; these integers are called the representants of the corresponding tags. The chosen algorithm is then implemented with each string comparison replaced by a comparison of the representant (which can generally be done in one or two machine instructions). Only when the representants are equal, a string comparison is done [5, p. 231].

This scheme, however, has some drawbacks.

- From a machine-independent point of view there is no gain. On a machine where integer comparison and string comparison are equally (in)expensive, we lose. On the CD Cyber the scheme has its merits, especially if the majority of the tags is longer than 10 characters. On the IBM 370 the gain may be marginal.
- The representant has to be calculated, the cost of which may come close to that of a string comparison.
- The representant needs an additional machine word for each string.
- Special measures have to be taken for the "one in maxint" chance that two different tags come up with the same representant.

We shall measure the time in terms of "major actions": string comparison and hash function calculation. It will be assumed that on most machines these will take an order of magnitude more time than the "minor actions" like "increase by 1" or "follow pointer". Moreover, if the number of minor actions per major action is more or less constant, the number of major actions is a good measure of the total amount of work.

Some algorithms need massive amounts of minor actions, not connected to any major action (e.g. table moves). We shall try to take these into account.

In order to compare the efficiency we shall have to make rather explicit assumptions about the input. If we compare the algorithms for 10 tags, we come up with a different proposal

(linear search) from one for 1000000 tags (polyphase tape sort?), neither of which seems a reasonable answer to our present needs.

Like most systems programs, ALEPH programs are not small. Few have less than 400 different tags, none in existence has more than 1500 different tags. This suggest that 1000 is a reasonable test datum. Some counting indicates that such a program contains in total about 4000 tags, many tags occurring only twice whereas some occur a hundred times.

Although a Zipf-law distribution might be nearer to the truth, we shall, for the sake of simplicity, assume that each of the 1000 tags occurs exactly 4 times. We subject the algorithms to two input sequences:

- the random sequence, an arbitrary permutation of (tag1, tag1, tag1, tag1, tag2, tag2, tag2, tag2, ... tag1000, tag1000, tag1000, tag1000),
- the ordered sequence, a similar permutation with the property that the first occurrences of all tags are in alphabetic order.

Literature is full of formulas for the efficiency of a tag-list algorithm for a given number of tags; there seems to be no literature about the influence of the presence of duplicates on these formulas.

The calculations are simple for those algorithms that keep the set of tags in such a form that adding tags does not change the access time of the tags already present: tags are added, in a sense, to the "periphery". Binary trees and unordered lists come in this category.

In these cases a tag is entered by successively comparing it to tags on a path determined by the algorithm until an empty place is found, into which the tag is then inserted. Let cost(i) be the average number of comparisons on such a path in a set of i tags. When this same tag is entered a second time, it follows the same path at the end of which there is one extra comparison to verify that it is indeed the same tag. Number of comparisons: cost(i) + 1.

The total cost of looking up all tags in a set of N different tags each of which occurs M times is then

$$C = \text{sigma}(i,0,N-1,\ \text{cost}(i) + (M-1)*(\text{cost}(i)+1)) =$$

$$= M * \text{sigma}(i,0,N-1,\text{cost}(i)) + (M-1)*N$$

If the same-path requirement mentioned above is not fulfilled (e.g., in ordered linear lists) no simple calculation exists. A solution may be obtained along the following lines:

Solve the recurrency relation

$$V[i+1] = V[i] + \frac{M*(N-V[i])}{M*N - i}$$

where V[i] is the expected number of different tags in the first i tags of a permutation of M*N tags, N of which are different. The fraction is the chance that the (i+1)-st tag is new: there are M*N-i tags after the i-th, N-V[i] of which are new, each occurring in M copies.

This recurrency relation can be solved and yields a formula for V[i] which consists of a sum of repeated products. This gives an estimate of the probability that the i-th tag will be new, which can in turn be used to calculate the total cost C for the most probable input sequence.

The above analysis was deemed far too heavy for the simple use we shall make of its results and a simplified model was used instead.

In the initial phase of entering the M*N tags (almost) all tags are new; in the final phase (almost) all tags are old; in between is an intermediate phase where the cross-over takes place. In our model the intermediate phase is absent, i.e., N new tags are entered, whereafter (M-1)*N old tags are entered.

This is indeed the worst possible sequence, but the best possible sequence (each tag followed directly by its M-1 copes) is much more unlikely. This view is supported by hand-calculation of the case N = 2, M = 3 for an ordered linear list. Out of the 20 permutations 16 need 7 comparisons, 3 need 6 and 1 needs 5.

1.4. Requirement D, the alphabetic listing.

It must be possible to produce an alphabetic listing of the tags present in the tag list. If the algorithm does not keep the tags in alphabetic order, a subsequent quicksort can do the job, at the cost of 11900 major actions (formula 24 in Knuth [1, p. 121]). Such a scheme will be considered a slight disadvantage to the algorithm, since

- it makes a lot of additional code necessary, and may require additional space if the data is not kept in table form,
- it will prompt the user to request an option to switch the alphabetic listing off, since obsolete listings are cheaper.

It is immaterial whether digits are alphabetized before or after the letters.

1.5. Requirement E, removal of tags.

It will not be necessary to remove arbitrary tags from the tag list. In view of the way the ALEPH standard postlude is read, it would be nice if there were a way to withdraw the tag

last entered.

1.6. Requirement F, code-dependency.
Since the tag list will not be prefilled, its structure may depend on the actual character code of the machine.

2. The algorithms.

The tag-search problem can be formulated very generally as follows:
Find a given tag T in the universe of all tags and yield the information appended to it, if present.

In view of the generality of the problem it does not come as a surprise that most tag-list algorithms have essentially the same structure:

a. set an abstract variable S equal to the universe of all tags, each with its information collected so far, and set M to 1.
b. if S contains only one element, yield the information attached to it and stop.
c. ask question Q[M] about the given tag T and save its answer in A.
d. set S to that subset of S that contains all the tags for which question Q[M] yields answer A, increase M by 1 and return to step b.

Reasonable questions Q about T could be:
- how does T compare to a given tag in S?
- what is the M-th character in T?

The only one not to conform to this scheme is the open hash-algorithm: closed hashing, linked hashing, etc., all fit nicely. For a discussion of the discrepancies see 2.5.

In practice the "universe of all tags, each with its information collected so far" is replaced by a connected set of those tags that actually have information attached to them. This slightly complicates the test in step b. In the original algorithm, if we are left with one tag, it must be it, for they were originally all there. Now, however, S may turn out to be empty, in which case the tag must be inserted, or it may contain exactly one tag in which case we must check whether it is the given tag and if not, insert it. This explains why in most algorithms insertion must be done in more than one place.

There are basically two ways of representing a "connected set of items" in a computer: in a contiguous piece of memory (a "table") or in an n-way linked list.

A table is represented by the address of its first item and its length; questions can be asked about any item, e.g., the first, middle or last one. An n-way linked list is represented by the address of an item (the root) which contains n addresses of n other n-way linked lists; questions can be asked about the root only.

A table has the disadvantage that insertion is hard in terms of CPU time, and n-way linked lists need room for n links for each item. This is why trade-off is a recurrent word in discussions about tag-list algorithms.

The exact form of the data structure representing the set of tags is generally determined by the form of the questions Q that will be asked.

The simplest schemes keep Q[M] the same for all M. In order to obtain significant answers all the time, Q will then have to depend on S, which contains only tags. So it must be of the form: "How does T compare to a certain tag in S?". The simplest form of this question is: "Is T equal to the first tag in S?". Combined with the simplest data structure, the table, this leads us to the "simplest" tag-list algorithm: sequential search in an unordered table.

If an algorithm restricts its questions to: "How does one tag compare to a second tag?", we call it a comparison algorithm. If it asks other questions we call it a non-comparison algorithm.

Two types of answers are possible to the question "How does this tag compare to that?": equal/unequal and smaller/equal/larger. The latter answer implies an ordering of the tags which must be supported by the data structure. Since the ALEPH 'compare string' is of this type, methods using this information are preferred.

In most algorithms all Q[M]'s are of the same nature (differ in parameters only). If some of the Q[M]'s are so different from the others as to require different coding (and possibly different data structures) we shall call the algorithm hybrid. We designate the set of questions Q[k], Q[k+1], Q[k+2], ... by Q[k:].

The algorithms reviewed will be characterized by the following five items:

- memory:
  the number of memory locations needed. The minimum is 1000 since this number is required for the pointers to the 1000 tags.

- random input:
  the average number of major actions required to handle a random permutation of 4000 tags of which 1000 are different.

- ordered input:
  the average number of major actions required to handle the ordered version of that permutation.

- alphabetic listing:
  cost (time and memory) of transforming the data structure so that the tags can be accessed in alphabetic order by a simple algorithm.

- removing last item:
  the cost of removing the item last entered (desirable to avoid cluttering up the tag list with unused ALEPH standard externals).

All numbers have been rounded to two-digit accuracy.

2.1. Comparison algorithms on tables.

The combination of tag comparison and contiguous tables leads to the well-known linear, binary and interpolation searches, in which the tag chosen for comparison is the first, the middle and the most probable one, respectively. At first sight these schemes seem doomed to fail since while they work nicely for existing tags, they have no facility for entering new tags, except moving half the table which makes them very expensive. This problem does not occur with unordered tables which we shall discuss first.

2.1.1. Unordered tables.

The only way to find a tag in an unordered table is by linear search. This is the only algorithm we shall meet that requires no overhead in terms of memory locations: the tags are connected through their contiguousness, the information block can be incorporated into the table since items do not move and the table contains no unused entries.

Its characteristics are:

name:	linear search in unordered table
memory:	1000
random input:	2000000
ordered input:	2000000
alph. listing:	12000 (+ 1000 loc)
removing last item:	0

2.1.2. Ordered tables.

Insertion in an ordered table is a problem. Several schemes come to mind:
- Push the tail to the right.
- Put newcomers in an auxiliary table.
- Dilute the table with empty entries.

Pushing the tail to the right makes the process of the order N^2 in terms of minor actions, but it does not cause any overhead for old tags.

Putting newcomers in a separate table and occasionally merging them into the main table does add an N^2 component to the algorithm, in terms of both minor and major actions. Moreover, it needs separate algorithms for searching the main table, searching the auxiliary table and for merging.

Diluting the table with empty entries increases memory requirements slightly but reduces the length of the tail to be pushed aside. When the table becomes too concentrated, more empty entries are added (and stirred well). This seems to add an N^2 component to the algorithm, since all tags will be moved when rediluting takes place.

A more careful analysis, however, shows that this is not so. There are three sources of minor actions to be distinguished:
- redilution,
- pushing aside a short tail when entering a new tag,
- hitting an empty entry when comparing strings.

The exact cost of redilution is not easy to assess; an approximation can be found as follows. We assume that redilution takes place when the table becomes so concentrated that only one out of every s entries is empty, and that afterwards one out of every r entries is empty (this implies $r < s$). Now suppose that redilution has just taken place upon entry of the N-th tag. It started with $N*s/(s-1)$ locations and ended with $N*r/(r-1)$ locations, so it cost $N*r/(r-1)$ moves. This means that the previous redilution ended with $N*s/(s-1)$ locations and consequently cost $N*s/(s-1)$ moves. This shows the ratio between two consecutive redilutions to be

$$\mathrm{rho} = \frac{s}{s-1} * \frac{r-1}{r}$$

We can now calculate the costs of redilutions in a more and more distant past, and if we sum the resulting geometric series

$$N*r/(r-1) * (1 + \mathrm{rho} + \mathrm{rho}^2 + \ldots)$$

to infinity we get for the total cost of rediluting:

$$N * \frac{r}{r-1} * \frac{1}{1-rho} =$$

$$N * \frac{r}{r-1} * \frac{r*(s-1)}{s-r} .$$

The average length of the tail to be pushed aside upon entering a new tag will be about (r+s)/2, and the cost of doing this N times is

$$N * (r+s) / 2 .$$

Likewise the chance of hitting an empty entry when searching for a string is 2/(r+s). This number has to be multiplied by the number of comparisons required by the specific algorithm.

Results of the application of the above to an actual algorithm can be found in 2.1.2.2.

Redilution is easily done in ALEPH by first extending the table with the necessary number of blocks and then from the tail end on moving down 1 empty block followed by r-1 old blocks. If the language does not allow table extension, a new area could be requested, and dilution could be done during copying.

None of these schemes allows the information blocks to be incorporated in the tag blocks, since the location of the latter may change when tags are added.

Three search techniques can be used on an ordered table:
- linear search,
- binary search,
- interpolation search.

Linear and binary search will be treated below.

Interpolation search is based on the idea that tags starting with an A should be sought for at the start of the table and these with Z at the end. The method is suggested by many authors (Knuth [1], p. 416, Tanenbaum [6], p. 307) and fully analysed by Yao and Yao [7]. The general result is that it is only worth while for very large sets of tags. For our case we can understand this as follows.

Although many variants are known, the following method lends itself reasonably well for analysis:

- The most likely position p of the tag is determined by linear interpolation.
- The standard deviation in this position is of the order of sqrt(N). A constant c is chosen and it is determined whether the wanted tag is inside the region

$$[p - c*sqrt(N) : p + c*sqrt(N)] ,$$

or before or after it, at the expense of two comparisons.

If we are lucky, T is inside the region; this reduces the table size to 2*c*sqrt(N), which is a considerable improvement when N is large. If we have bad luck, the range is reduced by about a factor of 2; we could have had this by binary search with one comparison.

The average size of this range is

$$P(c) * 2*c*sqrt(N) + (1-P(c)) * (N - 2*c*sqrt(N))/2$$

where P(c) is the probability that we hit the middle region. We assume P to have normal distribution.

In order to improve our chances we want to increase c; this, however, reduces the effect. There is an optimum value of c (which varies slowly with N) which yields the smallest average range.

Hand calculation shows that for N = 1000 the best value for c is 1.8 and the range is reduced to 140, for 2 comparisons. This is slightly better than binary search which would need almost 3 comparisons to achieve the same effect.

Repeating the procedure for N = 140, we get c = 1.3 and a new range of 35, which is exactly what binary search would have done.

We see that with N near 1000 interpolation search is only useful as Q[1] (Q[2:] being binary search, which makes it hybrid). It will then save us slightly less than 1 comparison. However, it requires code to do linear interpolation on tags, the execution cost of which is probably higher than that of a tag comparison, thus consuming all profit.

Now let's see what happens if N is really large, say N = 1000000. Then c = 3.0 and the first step reduces the range to 7300, at the expense of 2 comparisons and one interpolation, together 3 major actions. This is reduction by a factor of 137, for which binary search would have needed slightly more than 7 major actions. So there is some gain, but it is not impressive: 20 ($= {}^{2}\log 1000000$) major actions for binary search and 16 ($= {}^{2}\log 7300 + 3$) for interpolation as the first step.

2.1.2.1. Linear search in an ordered table.

Since linear search is an N^2 process anyway, there is no reason to choose anything but the simplest insertion method: pushing the rest aside. We obtain the following characteristics:

name:	linear search in ordered table	
memory:	2000	
random input:	1750000	
ordered input:	2000000	
alph. listing:	0	
removing last item:	0	(+ 500 moves)

2.1.2.2. Binary search on an ordered table.

If we use the simple push-aside technique for insertion we get:

name:	binary search in ordered table	
memory:	2000	
random input:	37000	(+ 250000 moves)
ordered input:	37000	
alph. listing:	0	
removing last item:	0	(+ 500 moves)

The number of comparisons has been obtained by calculating cost(i) according to:

```
cost(i) = i is 0 : 0,
          i is odd: cost(entier(i/2)) + 1,
          i is even: i/2 * cost(i/2-1)/(i+1)+
                     (i/2+1) * cost(i/2)/(i+1)
```

and substituting it in formula 1 in 1.3.

We see that for each tag comparison 7 moves must be done. This may be comparable to the work of one comparison.

If we dilute the table with empty blocks, we have to decide on values for r and s, the parameters that govern the redilution as explained in 2.1.2. Here N = 1000 and the number of comparisons is 37000. The following table gives the number of minor actions for various values of r, at those values of s where the total number of minor actions reaches a minimum. Memory requirement goes up by at most 1000/r.

r	s	Redil-uting.	Enter-ing.	Hitting empty.	Total
2	11	4400	6500	5700	16600
5	14	9000	9500	3900	22500
10	25	17800	17500	2100	37400
15	37	26800	25500	1500	53800

We see that at r=10, s=25 the increase in run-time and the increase in memory are both negligible. This yields the following characteristics:

name:	binary search in diluted table (r=10, s=25)	
memory:	2100	
random input:	37000	(+ 37000 moves)
ordered input:	37000	
alph. listing:	0	
removing last item:	0	(+ 500 moves)

2.2. Comparison algorithms on n-way linked structures.

In a linked structure the relation between entries is not through proximity (next address) but through explicit pointers. These do take room, but obviate the need to move entries around, which saves us the pointers to the information blocks.

2.2.1. Linked lists, n = 2.

There is no point in having an unordered list since an unordered table is better. So only the ordered list will be considered. When used with the simple question Q[M]: "How does this tag compare to the first one in the list?", its characteristics are:

name:	ordered linked list
memory:	2000
random input:	1750000
ordered input:	2000000
alph. listing:	0
removing last item:	500

Memory requirements are equal to those of the linear list. Although 1000 locations are needed for linkage, the entries do not move around, thus enabling us to incorporate the information block in the entries. This saves 1000 pointers.

Matthijssen and Uzgalis [12] explain another linear list technique in which different questions are asked. Their questions (tests) recur in groups of length D:

- The first question in such a group is the familiar "How does this tag compare to the first in the chain?"
- The other D-1 questions examine D-1 random entries and check whether any would give a shorter search path than the first entry in the present chain. If so, it is used instead.

The algorithm was originally developed for tag lists that reside on disk and are paged into memory, each page containing D entries. With small values of D it might be useful for lists in ordinary memory; e.g., for D = 2 it can be shown that the number of comparisons is about 2*n*sqrt(n).

We can find a "random" entry by utilizing the fact that the entries are in a contiguous piece of memory: the direct neighbour of the given entry will do.

This results in the following characteristics:

name:	paged search in ordered list
memory:	2000
random input:	260000
ordered input:	260000
alph. listing:	0
removing last item:	500

Taking the preceding entry as the "random" entry will avoid degeneration from ordered input. Better still, use the preceding entry if the present entry number is odd, and the next if it is even; this guarantees that no tag will be tested twice.

2.2.2. Binary trees, n = 3.

Binary trees lend themselves well to be used in conjunction with three-way comparisons. The question Q[M] to be asked about all tags is then: "Is this tag smaller than, equal to or larger than the one in the present root?". First the question is asked about the tag to be entered; this yields the answer smaller/equal/larger; then the same question should be asked about all other tags, to partition them in three groups, but this operation is made trivial by the data organization which allows the three groups to be found by following the left link, looking in the entry, and following the right link respectively.

The classical binary tree algorithm has the following characteristics:

name:	binary tree search
memory:	3000
random input:	47000
ordered input:	2000000
alph. listing:	0
removing last item:	12

The number of comparisons has been obtained by substituting

cost(i) = 2 * sum(k, 1, i+1, 1/k) - 2 * i

in formula 1 in 1.3. The expression for cost(i) is formula 5 derived by Knuth [1, p. 427].

We see that ordered input causes a disproportionate number of comparisons, resulting from the degeneration of the tree into a linear list. Many schemes have been brought forward to remedy this and a comparison of their merits can be found in an article by Baer and Schwab [4]. They all require additional information (generally a few bits) in the entry, which in ALEPH implies allocating another word. This would raise the memory requirement to the unacceptable level of 4000.

There exists, however, a method for avoiding this degeneration which does not require additional space. This method, which has received unfortunately little attention in literature, consists simply of locally randomizing the tree at random moments. The randomization is effected by changing the structure (A b C) d E into A b (C d E), or vice versa, in which b and d are entries and A, C and E are subtrees [8]. The author, G. Kok, advises to apply this transformation on the average once per insertion/look-up. It will, in general, counteract degeneration. It is not resistant to "frustration sequences", but we did not require that anyway.

The transformation involves changing three pointers and keeping track of the last entry visited plus whether we went left or right there. The total cost is estimated at 1 major action per tag offered.

The scheme has the following characteristics:

name:	randomized binary tree
memory:	3000
random input:	51000
ordered input:	51000
alph. listing:	0
removing last item:	12

2.2.3. Multi-way trees, n > 3.

Multi-way trees allow immediate partitioning of the tags into more than 3 groups. Since, however, tag comparison gives at most 3 different answers, this is of no use to us. They do,

however, figure in the hybrid schemes described in 2.4.4.

2.3. Non-comparison algorithms.

Non-comparison questions tend to have many different answers; this implies partitioning the tags into many different groups, which in turn must be split again. A table is not very handy for this, although the following scheme comes to mind:

- Let Q[M] be "What is the M-th character?".
- Let the preceding M - 1 steps have reduced the pertinent part of the table to the interval I.
- Then, set C to the M-th character of the tag to be inserted, find in I the sub-interval J of all tags that have C for their M-th character, and set I to J. Finding this sub-interval can be done by a simple modification of the binary search method.

The actual algorithm is slightly more complicated since it has to take into account the possibility that a tag has no M-th character.

It performs its job doing "minor actions" only; nevertheless it seems to do the same actions as the binary comparison search, except in a different order. The algorithm seems more complicated, but this is because part of the action in binary comparison search is hidden in 'compare string'.

If 'string elem', needed to answer Q[M], does all the testing implied, it will on some implementations count as a "major action". Moreover, only 'compare string' is guaranteed to produce lexicographic ordering: nothing prevents the characters in ALEPH from carrying a parity bit, thus wrecking the alphabetic listing!

The multi-valued answers yielded by non-comparison questions can be handled very well by n-way trees, but only at the expense of extensive memory requirement. If we have 26 links at the root of the tree and 36 links in the other nodes, a tag of length n can be located by n indexing operations, but the memory cost of such a scheme is prohibitive.

Since most of the n links will be nil, it is advantageous to replace the table of n links by a linked list of those links that are not nil. This leads us to schemes like "Patricia" as described by Knuth [1, p. 490].

Comparing memory requirements is non-trivial. Although these schemes need more linkage space, they do not require the text of the tags to be stored in a specific area. Rather, those characters that differentiate a tag from its neighbouring tags are distributed over the tree, where, in machine language, they

can generally be accommodated in open spaces between links. In ALEPH, however, we need separate words for them.

Now, the number of differentiating characters is less than the total number of characters in all tags, so there seems to be an advantage here. But on machines where strings of characters can be stored much more efficiently than single characters (e.g. the CD Cyber), the advantage is soon lost.

Coffman and Eve [9] describe a related scheme. A tag is converted to a unique stream of bits and the question Q[M] yields bits 2M-1 and 2M from this stream. The sub-set consisting of those tags that have the same 2-bit combination can then be found by following one of the four links kept in each entry.

Memory requirement will depend mainly on how soon the bit streams for different tags will start to differ. Van de Lune [10] has analysed this and found that instabilities arise.

2.4. Hybrid algorithms.

The hybridity causes three complications.

- we need code for two different algorithms,
- these generally need different data structures,
- we need a test to determine when to switch.

For a hybrid algorithm to be attractive, it must have advantages which offset these complications.

It is in principle possible to combine any partitioning/search technique with any other at any moment, but some combinations are more advantageous than others.

Reasonable combinations can be found by taking a non-hybrid technique, finding out at what level of splitting up it stops working satisfactorily and then finding a suitable continuation.

2.4.1. Linear search.

For linear search there is no such level.

2.4.2. Binary search on tables.

Although binary search is in theory faster than linear search for all lengths > 1, in practice linear search is faster than binary search for small lengths, the break-even point lying near 20.

So we can make a profit by switching to linear search when the length of the region becomes less than say 10. The test is easy, the data structure is the same and the second algorithm

is simple, so this seems a good technique.

2.4.3. Binary search on trees.

The same applies to binary trees. Here we can have a tree with the usual binary nodes, and with leaves consisting of tables of fixed length T. If a table overflows it splits into two tables and a node. This means that the low-level nodes (which are most numerous and contain many empty links) are concentrated in (much denser) tables, so that we can hope to make a profit in terms of memory. On the other hand, the splitting of the tables causes the entries to move around, which prevents us from incorporating the information blocks in the entries and this will cost us another 1000 linkage locations.

Explicit calculations are required to determine the exact gain. Let the tree have k nodes and k+1 leaves. The tables being 3/4 filled on the average, such a tree holds

$$k + 3/4\ (k+1)\ T$$

tags, which yields for 1000 tags:

$$k = (4000 - 3T)/(4 + 3T).$$

The amount of memory required by this data structure is

$$3k + (k+1)T + 1000,$$

which, for som values of T is:

T	memory
32	2400
16	2460
8	2570
4	2750
0	4000.

We see that reasonable values of T will save about 500 locations, compared to the 3000 needed for normal binary trees.

The run-time cost will not differ considerably from that of the simple binary tree algorithm, as far as random input is concerned. Ordered input will be handled more efficiently since the links in the linear chain now contain T/2+1 tags rather than 1.

Some hand calculation leads us to the following characteristics:

name:	broad-leaf tree		
T:	8	16	32
memory:	2600	2500	2400
random input:	47000	47000	47000
ordered input:	400000	240000	140000
alph. listing:	0	0	0
removing last item:			

complicated; it may have caused a table split !

2.4.4. Non-comparison algorithms on n-way trees.

In 2.3 these were declared prohibitively expensive in terms of memory, mainly because most of the links in the non-root nodes are empty. This suggests using n-way split for Q[1] and use a more memory-conserving technique for Q[2:]. If n is large, questions Q[2:] are applied to a small set only, and the simplest techniques should suffice. Algorithms of this type are known as closed-hash algorithms.

Trees can be ruled out as data structures to support the Q[2:], for three reasons.

- They take 1000 locations more for their internal linkage and these 1000 locations are clearly spent better by increasing n in the n-way split in Q[1].

- They degenerate into linear lists upon ordered input, and to regain their superiority we are forced to apply randomization schemes, which is absurd.

- As mentioned before, linear search is better for small sets, and n should be large enough for the resulting sets to be small.

We obtain the following characteristics for various values of n.

name:	closed hash			
n:	64	256	1024	
memory:	2100	2300	3000	
random input:	34000	14000	9000	
ordered input:	38000	15000	9000	
alph. listing:	12000	12000	12000	(+1000 loc)
removing last item:	0	0	0	

These figures are impressive but do indicate a fairly strong dependence on n: the technique works best if n is not more than an order of magnitude smaller than the number of tags. This suggests that with n = 1024 the performance will begin to deteriorate for programs with more than 10000 (global)

tags.

Creating an alphabetic listing is seen to become a major component in the cost, both in time and in memory. If we are allowed to change the data structure while sorting, we can dispense with the 1000 locations. But it would be nice if we didn't have to sort in the first place.

Since Q[2:] keep alphabetic order, sorting is avoided if Q[1] does so too. This restricts Q[1] to "What are the first k characters/bits of the tag?", resulting in a tag list with thumb indices.

For an existing program containing 1000 tags (actually 1004 of which 4 were discarded at random) the following characteristics were obtained.

name: thumb indices of one letter, n = 26		
Q[2:]:	list	tree
memory:	2000	3000
random input:	150000	30000
ordered input:	170000	170000
alph. listing:	0	0
removing last item:	0	0

name: thumb indices of two letters, n = 702		
Q[2:]:	list	tree
memory:	2700	3700
random input:	47000	19000
ordered input:	52000	52000
alph. listing:	0	0
removing last item:	0	0

The above figures include the cost of answering Q[1] which is estimated at 1 major action per tag.

Since the thumb indices are definitely inferior hash functions, trees are still quite valuable for Q[2:], as is borne out by the above figures. Again the degeneration for ordered input can be offset by randomizing the trees.

2.5. Open-hash algorithms.

The open-hash algorithm (as described, e.g., by Knuth [1, p. 518]) differs fundamentally from the general algorithm given in the beginning of 2 in that it does not depend on persistently decreasing the size of the set of tags the sought tag must be in. The following technique is used instead:

a. set M equal to 1.

b. ask question Q[M] about the given tag T and save the answer in A.
c. set S to that subset of the universe of all tags that contains all the tags for which question Q[M] yields answer A.
d. compare T to the first tag in S that has information attached to it.
e. if they are equal, yield that information and stop, otherwise increase M by 1 and return to step b.

This algorithm only works if it can be guaranteed that eventually (as M increases) each tag will in turn be the "first in subset S". This requires considerable care in the choosing of the sequence Q[M], sometimes supported by non-trivial proofs (as is the case with quadratic hashing [13]).

The method derives its power from the fact that Q[1] can be chosen so that the very first tag we try (at step d and e) has a high probability of being the right one. This is done by making the mesh that Q[1] throws over the universe of all tags so fine that each box in it contains generally only one tag with information.

This n-way fan-out suggests a table of length n as data structure for Q[1]. The questions Q[2:] also split up the tags in n classes but in a different way, i.e., they yield different answers for the same tag; they use the same table of length n.

The requirement that each tag eventually be "the first tag in the selected subset" is then equivalent to the requirement that the sequence of answers to Q[1:] will, for any tag, contain all integer from 0 to n-1.

The easiest way to achieve this is to take for the answer to Q[M] the answer to Q[1] plus M-1, modulo n, since this requires only one calculation of a hash value, the one for Q[1]. A disadvantage is that for two tags for which Q[1] yields the same answer, the whole sequence Q[1:] will be identical. This leads to clusters in the table. Various schemes to avoid them are described by Knuth [1, p. 512].

Because of its high efficiency the open hash algorithm is very popular among compiler writers, but in the form described above it violates requirement B in 1.2: when N tags have been entered there is no room anywhere to put tag number N+1. And requirement B forbids us to present the programmer with an error message "Identifier table full - Compilation abandoned".

So before considering it seriously we have to convince ourselves that there is a reasonable way to remove this hard upper limit to the number of tags.

Hopgood [11] recommends to double the size of the table when it becomes too full. Criteria for "too full" are given in [11].

Changing the table size implies changing the hash function(s) which in turn means restructuring the table; algorithms for this seem to be missing from literature.

The simple approach to restructuring is to allocate a new table of length 2N, siphon the tags one by one from the old table to the new and discard the old table. This causes a (temporary) memory requirement of 3N.

Attempts to restructure a table in situ soon give birth to unduely complicated algorithms, all of which still need considerable working space.

A way out might be provided by the use of a scratch file, but only if all else fails.

Even if we find a solution to the restructuring problem the algorithm still behaves in a non-uniform way, suddenly requiring large amounts of memory upon adding a single tag. So, for the algorithm to be acceptable its other properties must be very good indeed!

The efficiency in terms of major actions (hash value calculations and string comparisons, including those that result from restructuring) depends on the density at which table extension occurs. Like with the diluted tables in 2.1.2 we can define a dilution ratio r such that on the average 1 out of every r entries is empty.

Calculations have been made for r = 2, 3 and 4. The results depend only slightly on the initial table size, the final table size always being 2048.

name:	open extending hash		
r:	2	3	4
memory:	3000	3000	3000
random input:	10100	11000	11800
ordered input:	10100	11000	11800
alph. listing:	12000	12000	12000
removing last item:	0	0	0

The number of major actions has been arrived at by first calculating the total cost of entering 1000 new tags, using formula 22 in [1, p. 521] and taking due account of the table extensions whenever they occur, and adding the cost of entering 3000 old tags, using formula 23 in [1, p. 521].

It should be noticed that the low number of major actions per tag (2.5) is maintained, regardless of the number of tags. This is the only algorithm that is perfectly linear in the number of tags (although sorting ruins this by requiring N

ln(N)).

3. Evaluation.

We have examined 18 algorithms, the results of which are collected in the following table. Deg.fac. stands for the degeneration factor due to ordered input.

name	comparisons	deg.fac	memory
lin. search in unordered table	2000000	1	1000
lin. search in ordered table	1800000	1	2000
bin. search in ordered table	37000	1	2000
bin. search in diluted table	37000	1	2100
ordered linked list	1800000	1	2000
paged search in ordered list	260000	1	2000
binary tree search	47000	40	3000
randomized binary tree	51000	1	3000
broad-leaf tree	47000	5	2500
closed hash 256	26000	1	2300
closed hash 1024	21000	1	3000
thumb ind., one letter, list	150000	1	2000
thumb ind., two letters, list	47000	1	2700
thumb ind., one letter, tree	30000	6	3000
thumb ind., two letters, tree	19000	3	3700
thumb ind., one letter rand.	34000	1	3000
thumb ind., two letters, rand.	23000	1	3000
open extending hash	19000	1	3700

We see that we can safely discard all methods that require more than say 100000 comparisons or have a degeneration factor of more than 3. Binary search in an ordered table needs 250000 moves, so it is not a good candidate either.

This is about as far as science will bring us: choosing between the remaining candidates is an art. Some support can be obtained by considering what special facilities are needed by the various algorithms. This results in the following table.

name	comp	d.f	mem	h.f	q.s	r.s	exp	order
bin. search. dil. t.	37000	1	2100				+	N ln(N)
rand. bin. tree	51000	1	3000			+		N ln(N)
cl. hash 256	26000	1	2300	+	+			N^2
cl. hash 1024	21000	1	3000	+	+			N^2
th. ind., 2L, list	47000	1	2700					N^2
th. ind., 2L, tree	19000	3	3700					N ln(N)
th. ind., 1L, rand.	34000	1	3000			+		N ln(N)
th. ind., 2L, rand.	23000	1	3700			+		N ln(N)
open ext. hash	21000	1	3000	+	+		+	N

where

- h.f means hash function required,
- q.s means quick-sort required,
- r.s means randomization scheme required,
- exp means expander required.

We see that a randomized binary tree can be cheaply improved by adding thumb indices, so we need not consider it further.

Hash functions are a nuissance in a machine-independent ALEPH program. They mean inspecting the tag character by character and doing calculations in double precision to avoid overflow. It might be thought that the words of the packed string may serve if we consider them as bit patterns but this is not true, since these words may contain unused bits which need not be preset in the same way all the time.
This, together with the need for a sorting routine, is no recommendation for the hash algorithms.

Since ALEPH has no modularity, there is a tendency among ALEPH programmers to start tags in the same "conceptual module" with the same prefix. Examples are GEN ACTUAL, GEN ADDRESS, GEN AFTERTHOUGHT, etc. in a code generator. This is ruinous to "thumb indices, 2L, list", but not to "thumb indices, 2L, tree", which would not perform worse than a normal tree. However, if he would put them in alphabetic order (and we don't know he won't), also "thumb indices, 2L, tree" will fail. Safety can then be found in randomizing.

So

- the fastest acceptable algorithm is "thumb indices, two letters, tree", which will, however, degenerate rather badly on some not unlikely input sequences.

- more safety can be found in "thumb indices, two letters, randomized", which is slightly more expensive.

- absolute safety can be reached with "binary search on diluted table", which takes half the memory and twice the time.

4. Conclusion.

- Since our design criteria put frugality of memory use before speed of compilation;
- since alphabetic sequences are not unlikely in ALEPH programs;
- since "binary search on a diluted table" needs much less memory than the others (and progressively so for smaller programs), is absolutely safe and is easy to program;

we conclude that for our application "binary search on a diluted table" is best. It might be useful to switch to linear search for small tables (see 2.4.2).

The ease of removing items may be important for other applications.

5. References.

[1] Knuth, D.E., Sorting and Searching, Vol. 3 of The Art of Computer Programming, Addison Wesley, New York, 1973.

[2] Severance, D.G., Identifier Search Mechanisms, Comp. Surveys, 6, 3, (Sept. 1974), p. 175.

[3] Nievergelt, J., Binary Search Trees and File Organization, Comp. Surveys, 6, 3, (Sept. 1974), p. 195.

[4] Baer, J.-L. and B. Schwab, A Comparison of Tree-Balancing Algorithms, Comm. ACM, 20, 5 (May 1977), p. 322.

[5] Waite, W., Implementing Software for Non-Numerical Applications, Prentice Hall, Englewood Cliffs, N.J., 1973.

[6] Tanenbaum, A.S., Structured Computer Organization, Prentice Hall, Englewood Cliffs, N.J., 1976.

[7] Yao, A.C. and F.F. Yao, The Complexity of Searching an Ordered Random Table, Symp. on Foundations of Comp. Science, IEEE Comp. Soc., 1976.

[8] Kok, G., Alfabetiseren en andere sorteerwerkzaamheden. (Collating and other sorting activities), MR 120/70, Mathematical Centre, Amsterdam (1970), p. 35, (In Dutch).

[9] Coffman, E.G. and J. Eve, File Structures Using Hash Functions, Comm. ACM, 13, 7, (July 1970), p. 427.

[10] Lune, J. van de, On the Asymptotic Behaviour of a Sequence Arising in Computer Science, ZW 31/74, Mathematical Centre, Amsterdam, 1974.

[11] Hopgood, F.R.A., A Solution to the Table Overflow Problem for Hash Tables, Computer Bulletin, 11, 4 (March 1968), p. 297.

[12] Matthijssen, R.L. and R.C. Uzgalis, Some Simple Data Structures in a Paged Environment, The UCLA Comp. Sc. Dept. Quarterly, 4, 1 (Jan 1976), p. 67.

[13] Ackerman, A.F., Quadratic Search for Hash Tables of Size p^n, Comm. ACM, 17, 3 (March 1974), p. 164.