18

(A 78/12)

# Sorting Presorted Files

Kurt Mehlhorn

Fachbereich Informatik
Universität des Saarlandes
D-6600 Saarbrücken

August 1978

A 78/12

Abstract: A new sorting algorithm is presented. Its running time is $O(n(1+\log(F/n))$ where $F = |\{(i,j);\ i < j \text{ and } x_i < x_j\}|$ is the total number of inversions in the input sequence $x_n\ x_{n-1}\ x_{n-2} \cdots x_2\ x_1$. In other words, presorted sequences are sorted quickly, and completely unsorted sequences are sorted in $O(n \log n)$ steps. Note that $F \leq n^2/2$ always. Furthermore, the constant of proportionality is fairly small and hence the sorting method is competitive with existing methods for not too large n.

## I. Introduction

In this paper we consider the problem of sorting presorted files. Consider the following two permutations of the numbers 1,2,3,...,7:

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

Intuitively speaking, the second permutation is more out of order than the first. A precise notion of this observation is to count the number of inversions; in our example the first permutation has 5 inversions (namely (3,2), (7,5), (7,4), (7,6) and (5,4)) and the second permutation has 15 inversions.

In general, let $x_n\ x_{n-1}\ x_{n-2} \cdots x_3\ x_2\ x_1$ be a sequence of elements $x_p$ from an ordered universe. Define

$$F = |\{(i,j);\ x_i < x_j \text{ and } i < j\}|$$

as the number of inversions in this sequence. Then $0 \leq F \leq n(n-1)/2$. (Note that $F = n(n-1)/2$ if $x_n > x_{n-1} > \cdots > x_2 > x_1$). We take F as a measure for the "sortedness" of sequence $x_n\ x_{n-1} \cdots x_2\ x_1$.

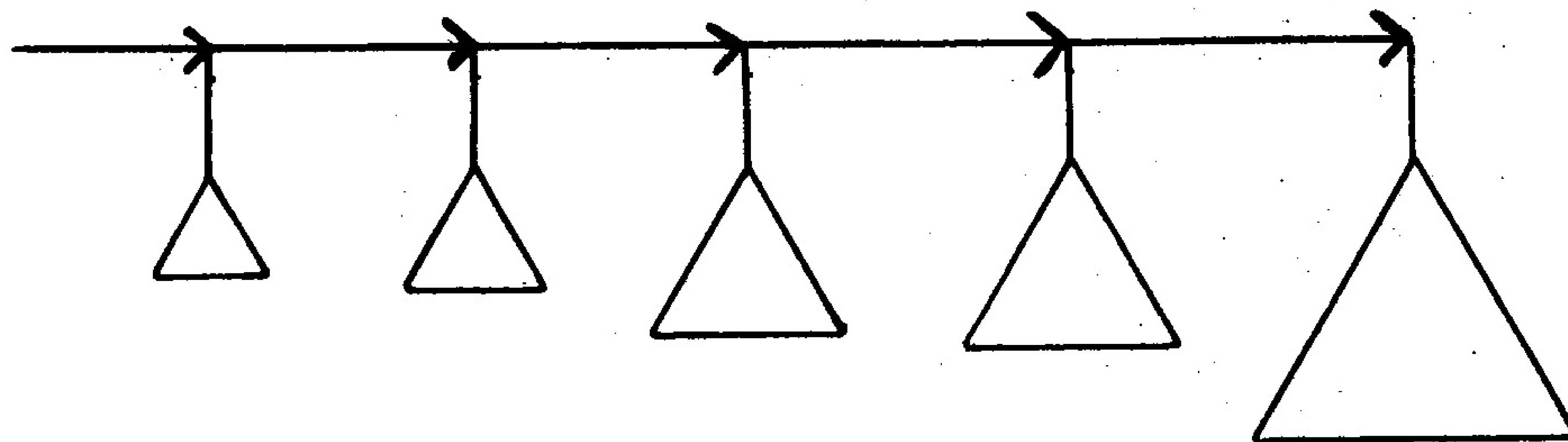
We will sort sequence $x_n\ x_{n-1} \cdots x_2\ x_1$ by an insertion sort, i.e. we start with the sorted sequence consisting of only element $x_1$ and then insert $x_2, x_3 \ldots$ at their proper places. Let

$$f_j = |\{i;\ x_i < x_j \text{ and } i < j\}| \quad .$$

Then $\Sigma f_j = F$. Also $f_j$ is the number of elements smaller than $x_j$ and to the right of $x_j$ in the input sequence. Hence $x_j$ has to be inserted at the $f_j$-th position during the insertion sort. If the input sequence is presorted, i.e. F is small with respect to $n^2/2$, then $f_j$ will tend to be small on the average. Hence most elements will have to be inserted near the front of the sorted sequence.

There is a well-known method for inserting an element into a ordered sequence $y_1 < y_2 < \ldots < y_m$ in time $O(\log t)$ where $t$ is such that $y_t < y \leq y_{t+1}$. This method was used by Fredman and Bentley & Yao and was termed exponential and binary search in Mehlhorn 77. The basic idea is to compare $y$ with $y_{2^i}$ for $i = 0,1,2,\ldots$ until $y \leq y_{2^i}$ (exponential search) and then to perform a binary search on the interval $y_{2^{i-1}},\ldots,y_{2^i}$.

After having found the proper position by exponential + binary search we will have to insert y at the proper place. Note that it will not do to have numbers $y_1,\ldots y_n$ is an array. The following data structure supports the search and the insertion process efficiently: A linear list L of AVL-trees $T_0, T_1, T_2, \ldots$, where $T_i$ has height i.



It supports exponential search since $T_i$ has at least $1.65^i$ leaves. It supports binary search and the insertion process by virtue of AVL-trees. (The actual data structure will be slightly more complicated).

In the next section we introduce the data structure,describe the sorting algorithm and prove that its running time is $O(n(1 + \log(F/n)))$. This is optimal up to a constant factor.

Our algorithm solves a special case of a more general problem considered by Guibas et al and Brown & Tarjan. They study finger trees which represent linear lists in a way, as to make insertion in the vicinity of certain points (called fingers) efficient . In their terminology we have just one finger which is always directed at the front end of the file. However, our data structure solves this special case more efficient. A sorting method arises which is compatible with existing methods for not too large n.

Our data structure has storage requirement 3n vs. 6n for the structure due to Brown & Tarjan. Guibas et al's data structure is based on B-trees of degree at least 25 and hence seems suitable only for very large n. Also, though their algorithms have the same asymptotic behavior, the constant of proportionality for our algorithm is smaller.

## II. The Algorithm

Our algorithm uses the following data structure:

1) Let L be a linear list $L_0, L_1, \ldots, L_k$.

2) Each $L_i$, $0 \leq i \leq k$, is a linear list of AVL-trees $T_{i,o}, T_{i,1}, \ldots, T_{i,\ell_i}$ with $\ell_i \geq 0$. The $L_i$ are called sublists.

3) $T_{i,o}$ is an AVL-tree of height i-1, i or i+1 and $T_{i,j}$, $1 \leq j \leq \ell_i$, is an AVL-tree of height i-1 or i. Furthermore $\ell_o = 0$ and $T_{o,o}$ has height 0 or 1.

A sublist $L_i$ is either clean or dirty. If $\ell_i = 0$ then $L_i$ is clean, if $\ell_i \geq 1$ then $L_i$ is dirty. We use our data structure to store ordered sets S:

3') Let T be an AVL with m leaves and let S be an ordered set with m elements. We store the elements of S in increasing order from left to right in the leaves of T. In an interior node v of T we store the largest element in the left subtree of the tree with root v. Fig. 1 shows an AVL-tree of height 2 and 3 leaves. The set {7,9,13} is stored in it.

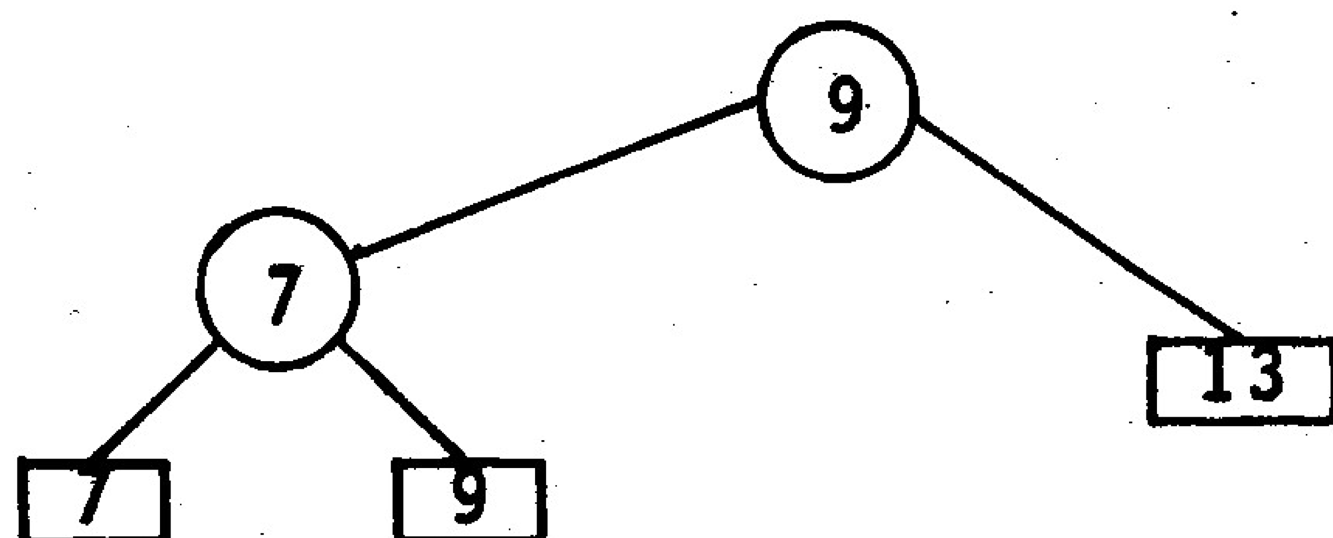



Fig. 1:

Fact 1 (Adel'son-Vel'skii + Landis)

Let T be an AVL-tree of height h and let m be the number of leaves of T. Then

$$F_{h+2} \leq m \leq 2^h$$

where $F_1 = F_2 = 1$ and $F_{i+2} = F_{i+1} + F_i$ are the Fibonacci numbers.

Since $F_i = (\alpha^i - \beta^i)/\sqrt{5}$ with $\alpha = (1+\sqrt{5})/2 \approx 1.618$ and $\beta = (1-\sqrt{5})/2 \approx -0.618$ it follows that

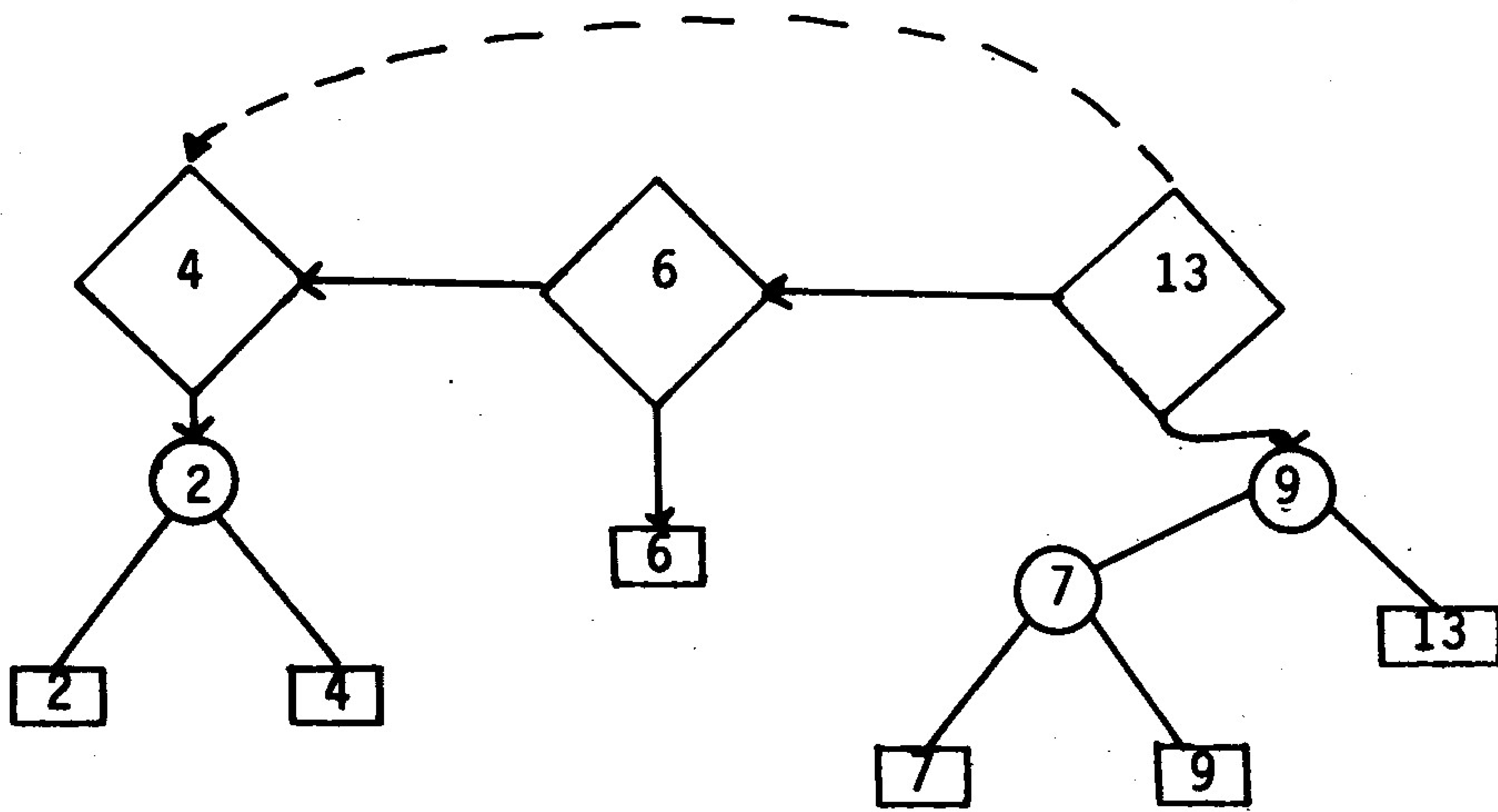
$$(\alpha^{h+2} - \beta^{h+2})/\sqrt{5} \leq m$$

and hence

$$(\alpha^{h+2}-1)/\sqrt{5} \leq m$$

2') Let $L_i$ be a sublist, i.e. $L_i$ is a linear list of AVL-trees $T_{i,o}, T_{i,1}, \ldots, T_{i,\ell_i}$ with $\ell_i \geq 0$. Let $m_j$ be the number of leaves of tree $T_{i,j}$ and let S be a set with $\sum_{j=o}^{\ell_i} m_j$ elements.

We store the $m_o$ largest elements of S in $T_{i,o}$, the next $m_1$ largest elements in $T_{i,1}, \ldots,$ and the $m_{\ell_i}$ smallest elements in $T_{i,\ell_i}$. In the sublist $L_i$ we store pointers to the roots of the trees $T_{i,j}$. Along with the pointer to tree $T_{i,j}$ we also store the largest element stored in Tree $T_{i,j}$, denoted $max_{i,j}$. Fig. 2 shows a list $L_1$ consisting of 3 trees $T_{1,o}$, $T_{1,1}$, $T_{1,2}$ of height 2,0,1 respectively. We have $m_{1,o} = 13$, $m_{1,1} = 6$ and $m_{1,2} = 5$. Note that we draw the




first element of the list at the right end. Also the first element always contains a pointer to the last element.

Fact 2: Let $L_i$ be a sublist. Then the total number of leaves of the trees in sublist $L_i$ is at least $(\alpha^{i+1}-1)/\sqrt{5}$ .

Proof: Since tree $T_{i,o}$ has height at least $i-1$ this follows immediately from Fact 1.

1') Let finally L be the linear list $L_0, L_1, \ldots, L_k$, let $n_i$ be the total number of leaves of trees in sublist $L_i$, $0 \leq i \leq k$, and let S be set with $n = \sum_{i=o}^{k} n_i$ elements. We store the $n_0$ smallest elements of S in $L_0$ as described in 2', the next $n_1$ smallest elements in $L_1, \ldots$ . Fig. 3 shows a possible data structure for set $\{2,5,6,7,9,13,19,21,\infty\}$. We draw list L as consisting of the first elements of the sublists $L_i$. In our example sublists $L_0$ and $L_2$ are clean and $L_1$ is dirty.

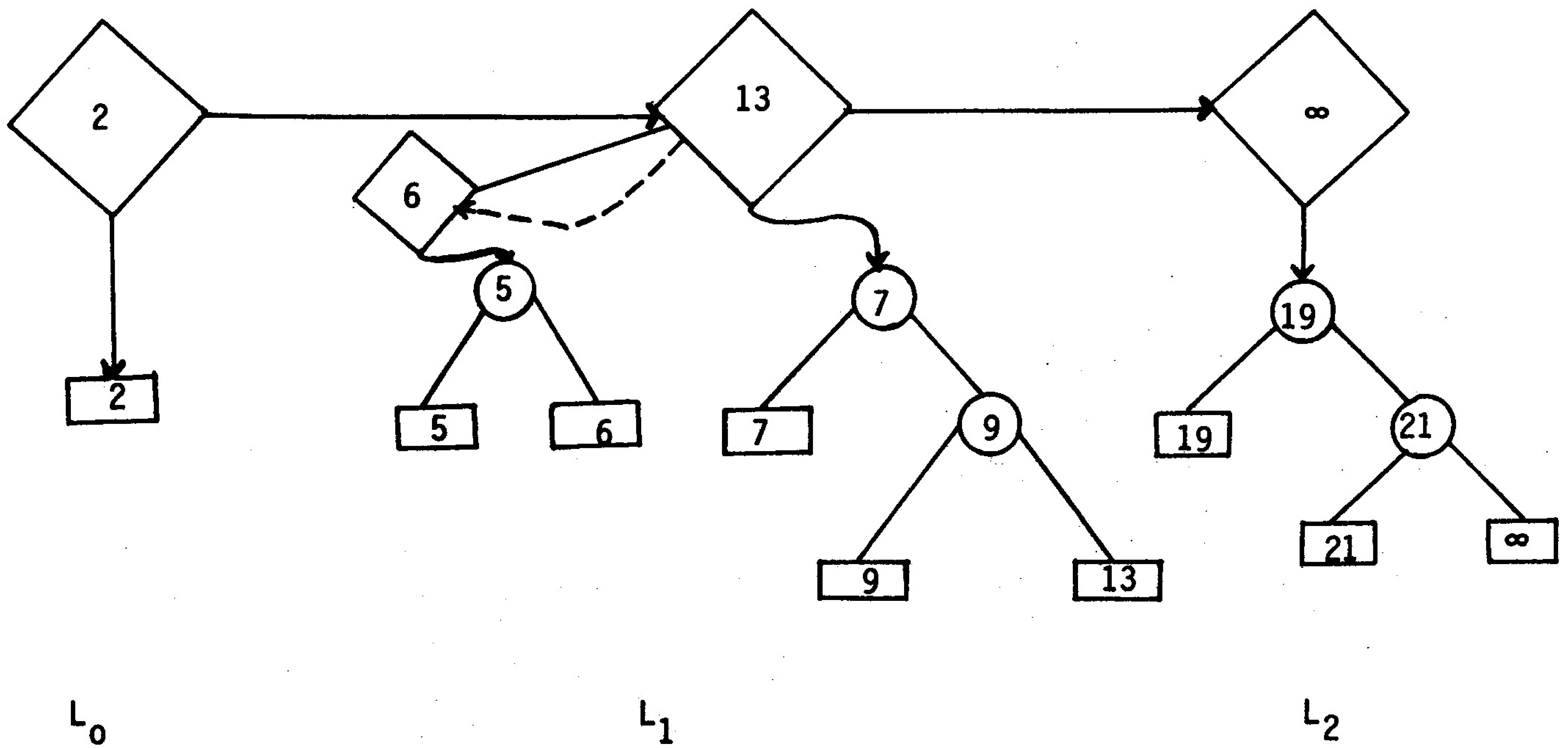



Fig. 3

We use our data structure to do an insertion sort. Let $x_n\ x_{n-1}\ x_{n-2}\ \cdots\ x_2\ x_1\ x_o$ be an unordered list of reals (to be specific). We assume that $x_o = \infty$, i.e. $x_o$ is at least as large as all other elements in the list. This assumption will allow us to eliminate some special cases and is quite customary in sorting.

We start with our data structure for the singleton set $\{x_o\}$, i.e. L consists of sublist $L_o$ only, $L_o$ consists of one tree of height 0 and the only leaf of that tree contains $x_o$. Next we insert $x_1, x_2, x_3, \ldots$ into the structure. Suppose we inserted $x_1, \ldots, x_{p-1}$ and obtained structure L. We want to insert $x_p$ next.

First we locate the sublist $L_i$ in which $x_p$ is to be inserted.

$i \leftarrow 0$

<u>while</u> $x_p > max_{i,o}$ <u>do</u> $i \leftarrow i+1$

Remember that $max_{i,o}$ is the largest element of tree $T_{i,o}$ and hence the largest element stored in sublist i. Suppose L consists of sublists $L_o, L_1, \ldots, L_k$. Then $max_{k,o} = \infty$ and hence the while-loop terminates although we have not included a test for end of list. Let h be the final value of variable i. Then $x_p \leq max_{h,o}$ and either $h = 0$ or $x_p > max_{h-1,o}$.

In our example we would have h = 1 if we try to insert 12 and we would have h = 2 if we try to insert 35.

<u>Definition:</u> Let $f_p := |\{q; q < p \text{ and } x_q < x_p\}|$ be the number of inversions caused by element $x_p$. $f_p$ is the number of elements to the right of $x_p$ yet smaller than $x_p$.

From now on, $\log f_p$ always denotes $\log \max(2, f_p)$.

<u>Lemma 1:</u> Let h be defined as above. Then

$$h = O(\log f_p)$$

<u>Proof:</u> If h = 0 then we have nothing to show. Suppose h > 0. Then $x_p > max_{h-1,o}$.

Furthermore, sublist $L_{h-1}$ contains at least tree $T_{h-1,o}$ which has height at least h-2 and hence at least $F_h \geq (\alpha^h - 1)/\sqrt{5}$ leaves. These leaves contain elements $x_q$ with $q < p$ and $x_q < x_p$. Hence $f_p \geq (\alpha^h - 1)/\sqrt{5}$ . □

At this point we have located sublist $L_h$ and we want to insert $x_p$ into one of the trees $T_{h,j}$, $0 \leq j \leq \ell_h$, on sublist $L_h$. We distinguish whether $L_h$ is clean or dirty.

<u>Case 1:</u> $L_h$ is clean, i.e. $\ell_h = 0$. We insert $x_p$ into AVL-tree $T_{h,o}$ using the standard insertion algorithm for AVL-trees. Tree $T_{h,o}$ has height h-1, h or h+1. After the insertion $T_{h,o}$ has

height h-1, h, h+1, h+2. If its height is $\leq$ h+1 then we are done and proceed to insert $x_{p+1}$. If its height is h+2, then let T' and T" be its left and right subtree and let v be its root. Trees T' and T" have height h or h+1, not necessarily the same. We let T' be the new $T_{h,o}$ and append T" at the end of sublist $L_{h+1}$. If $L_{h+1}$ did not exist,then we also have to create $L_{h+1}$. Note also that the maximal element in T' is stored in v and that the maximal element in T" is the old $max_{h,o}$. Hence we are able to maintain our data structure.

Lemma 2: O(log $f_p$) time units suffice to insert $x_p$ into $T_{h,o}$ and to update the data structure.

Proof: Tree $T_{h,o}$ has height at most h+1. Hence h+1 = O(log $f_p$) (by Lemma 1) time units suffice.

Case 2: $L_h$ is dirty. We will first clean $L_h$ by pairing the trees in it and moving some of the resulting tree to $L_{h+1}$.

Let $L_h$ consist of trees $T_{h,o}$, $T_{h,1}, \ldots, T_{h,\ell_h}$ with $\ell_h \geq 1$. Remember that $T_{h,o}$ has height h-1, h or h+1 and that trees $T_{h,j}$, $1 \leq j \leq \ell_h$, have height h-1 or h.

```
if   height of T_{h,o} is h+1
then  delete T_{h,o} from L_h and append T_{h,o} to the end of L_{h+1};
co  all trees on L_h have height h-1 or h.
while  L_h contains ≥ 3 trees
do begin  delete the first two trees T' and T" from L_h;
          combine them into a new AVL-tree of height h or h+1;
          append T to the end of list L_{h+1}
   end;
```

*co* $L_h$ now contains either one or two trees;

*if* $L_h$ contains two trees T' and T"

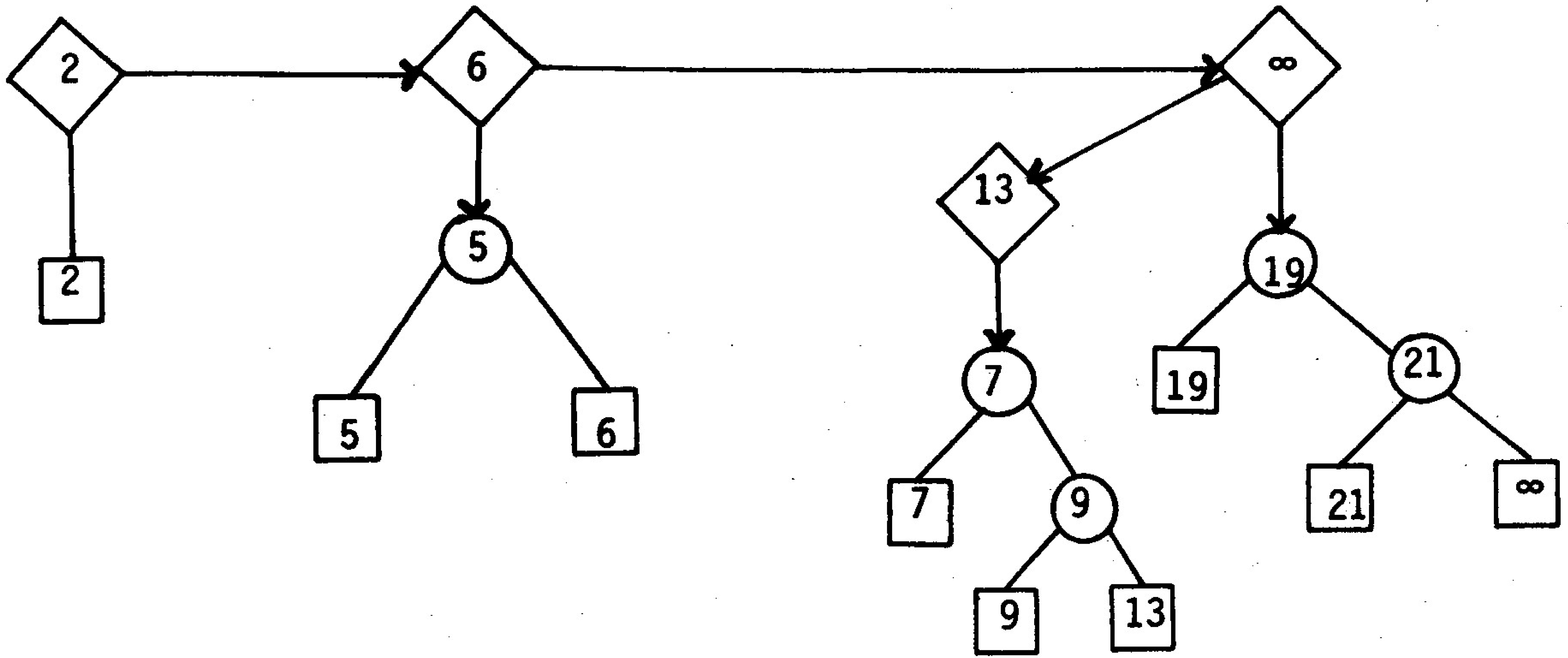
*then* combine T' and T" into a single tree T, delete T' and T" from $L_h$ and make T the only tree of list $L_h$;

At this point $L_h$ is clean, but $L_{h+1}$ may be dirty. Now we try again whether $x_p$ has to be inserted into $L_h$, i.e. we execute $x_p > max_{h,o}$ again. If it has to be inserted into $L_h$ then we proceed as in case 1. Otherwise we increase i by 1 to h+1 and try to insert $x_p$ into $L_{h+1}$. If $L_{h+1}$ is dirty, then we clean $L_{h+1}$ first,... .

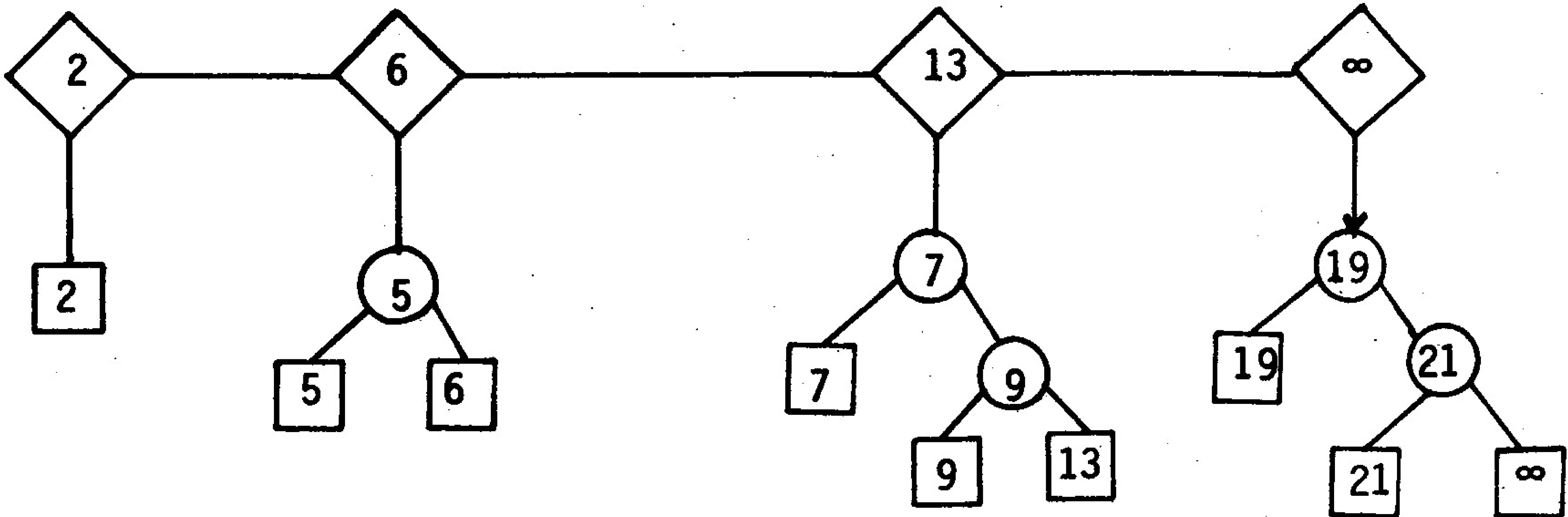
Cleaning $L_h$ means to move at most one tree unchanged from $L_h$ and $L_{h+1}$ and to pair off the remaining trees. Since at least one tree is moved unchanged or at least one pair is formed, the time required to clean $L_h$ is proportional to the number of trees moved unchanged (at most one) plus the number of pairs formed. We will use this fact when analysing the total time complexity of all cleaning processes.

This ends the description of the algorithm.

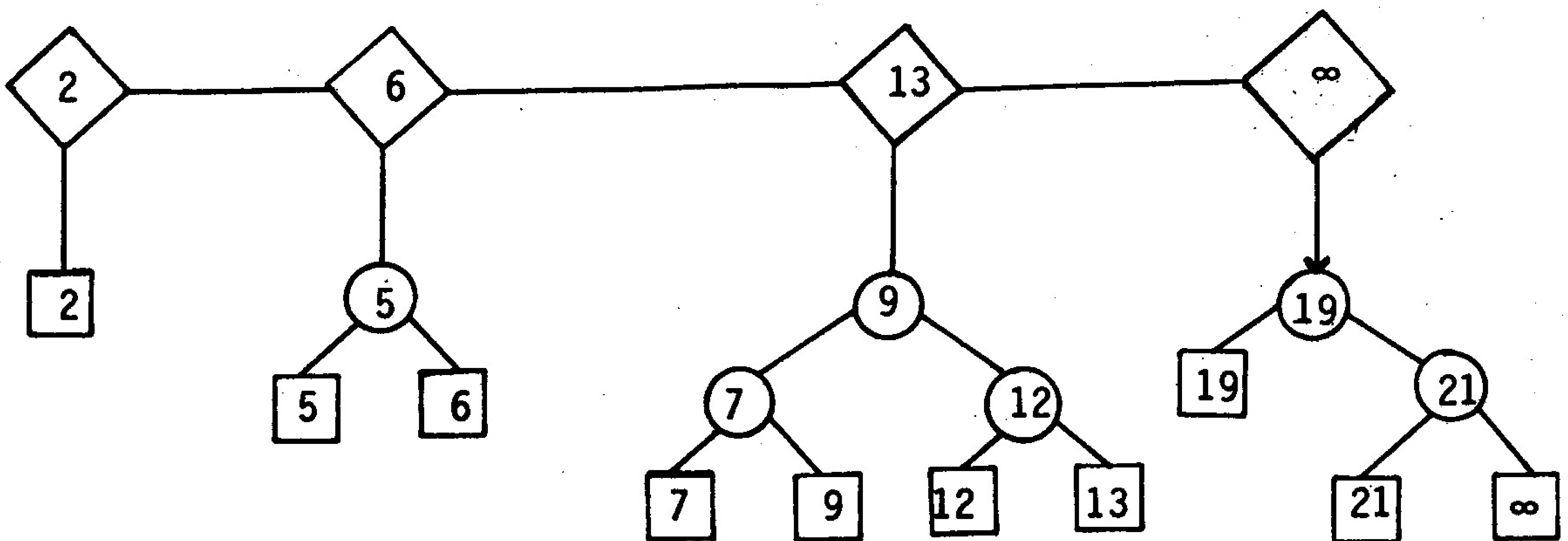
*Example:* Fig. 5 shows the effect of inserting 12 into the structure of Fig. 3. We first determine h = 1. $L_1$ is dirty. Hence $L_1$ is cleaned and the structure in 5a is obtained. Next we find out that i has to be increased to h+1. $L_2$ is dirty and we clean $L_2$. The structure in 5b is obtained. Now i stays stable and $L_2$ is clean. Hence we insert 12 into $T_{2,o}$ and obtain 5c.




5a




5b



It remains to estimate the running time of the algorithm. Suppose we used the algorithm to sort $x_n, x_{n-1}, \ldots, x_1, x_0$. Let S be the total time spent on cleaning and let T be the total time spent outside the cleaning processes.

Note that $T = \sum_{p=1}^{n} O(\log f_p)$ if we never try to insert into a dirty sublist. Suppose now that at some point we try to insert into a dirty sublist $L_h$. Then $L_h$ is cleaned first. The cost of cleaning is allotted to S. Then we find out at the cost of $O(1)$ whether we still have to insert into $L_h$. If we don't have to then we try to insert into $L_{h+1}$. Hence $O(1)$ time units are charged to T for looking at $L_h$ in this case. Hence for the computation of T we may as well assume that we only insert into clean lists and thus $T = \sum_{p=1}^{n} O(\log f_p)$.

<u>Lemma 3:</u> $T = O(n(1 + \log(F/n)))$ where $F = \sum_{p=1}^{n} f_p$

<u>Proof:</u> By the discussion above

$$T = \sum_{p=1}^{n} O(\log f_p)$$

$$= O(n) + O(\sum_{p=1}^{n} \log f_p)$$

$$= O(n) + O(\log \prod_{i=1}^{i} f_i)$$

$$= O(n) + O(n \log(F/n))$$

since $(\prod f_i)^{1/n} \leq (\Sigma f_i)/n$ (the geometric mean is never larger than the arithmetic mean). □

The quantity S is more difficult to estimate. In order to do so we construct a forest $\mathcal{F}$ in parallel to the execution

of the algorithm. There will be one interior node in $\mathfrak{F}$ for every tree moved unchanged and for every pair formed in the cleaning processes. Hence the number of interior nodes of $\mathfrak{F}$ will provide us with a bound for S. Initially $\mathfrak{F}$ consists of one leaf which represents the only tree $T_{o,o}$ in our structure.

a) The leaves of $\mathfrak{F}$: The leaves of $\mathfrak{F}$ will be constructed after inserting new elements into our data structure. Suppose we insert element $x_p$ into clean list $L_h$, i.e. we insert $x_p$ into AVL-tree $T_{h,o}$. After the insertion $T_{h,o}$ is either split or not.

a1) $T_{h,o}$ is not split. Then we create one new leaf which represents the tree $T_{h,o}$ on list $L_h$ after the insertion took place.

a2) $T_{h,o}$ is split into T' and T". T' stays on list $L_h$ and T" is moved to list $L_{h+1}$. We construct two new leaves of forest $\mathfrak{F}$ representing trees T' on list $L_h$ and T" on list $L_{h+1}$ respectively. Since at most two leaves are constructed after inserting an element into our data structure, forest $\mathfrak{F}$ has at most 2n+1 leaves.

b) The interior nodes of $\mathfrak{F}$: The interior nodes are constructed by the cleaning process. Suppose we clean list $L_h$. Then tree $T_{h,o}$ may be moved unchanged to list $L_{h+1}$. If this is the case then we construct one new node with a single son. This new node represents tree $T_{h,o}$ on list $L_{h+1}$. Its single son is the node which represented tree $T_{h,o}$ on list $L_h$. Also pairs of tree of $L_h$ are formed. For every pair formed we construct a new node with two sons. The new node represents the newly formed tree on whatever list it is on (either $L_h$ or $L_{h+1}$) and its two sons are the two nodes which represent the two constituent trees on list $L_h$.

From the construction of forest $\mathfrak{F}$ is it apparent that at every point during the execution of our algorithm every tree in the data structure will be represented by a node of $\mathfrak{F}$. Interior nodes of $\mathfrak{F}$ have either one or two sons.

Lemma 4: Let v be a node of $\mathcal{F}$ having exactly one son w. Then w is either a leaf or has two sons.

Proof: Assume otherwise. Then w has a single son x. x represents some tree T on some list $L_h$. Hence T has height $\leq h+1$. Node w represents the same tree on list $L_{h+1}$. Hence w represents a tree of height $\leq h+1$ on list $L_{h+1}$. Such a tree is never moved unchanged during the cleaning. Hence there can be no node having w as its only son. Contradiction. □

Forest F has at most 2n+1 leaves. Hence it has at most 2n nodes with two sons and hence at most 4n+1 nodes with one son. This shows that at most 8n+2 trees are handled during the whole of the cleaning processes. Hence S = O(n).

We thus proved:

Theorem: Let $x_n\, x_{n-1} \ldots x_1$ be an unordered sequence and let

$$F = |\{(i,j);\ i > j \text{ and } x_i > x_j\}|$$

be the total number of inversions in that sequence. Then our algorithm sorts the sequence in

$$O(n(1 + \log(F/n)))$$

time units. □

Guibas et al have shown that the logarithm of the number of permutations of $\{1,2,\ldots,n\}$ with at least F inversions is $\Omega(n(\log(1+F/n))$. Hence our algorithm is optimal up to a constant factor. This is also true for the algorithms due to Guibas et al and Brown & Tarjan. However, our constant of proportionality is smaller. A first analysis shows that the running time is about 24n log(F/n) + 4on on the machine defined in Mehlhorn 77. Comparing this with the running time of Quicksort which is about 9n log n on the average on that machine shows that

$$24n \log(F/n) + 4on \leq 9n \log n$$

if $$\log(2^{40/24} \cdot F/n) \leq (9/24) \log n$$

if $$F \leq 1/2^{(40/24)} n^{33/24} \approx 0.314 n^{1.375}$$

and hence the algorithm is competitive with Quicksort for $F \leq 0.314n^{1.375}$.

Guibas et al base their data structure on B-trees (Bayer & McCreight) of order (degree of branching) at least 25. Hence it seems unsuitable for small or moderate size n. Brown & Tarjan base their data structure on 2-3 trees. The storage requirement of their structure is 6n. Our data structure requires only 4n storage, a 3n implementation is possible. An obvious implementation of AVL-trees is to use 3 storage cells per node and one storage cell per leaf. This makes 4m-3 storage cells for an AVL-tree with m leaves. In addition, we need for each AVL-tree an element in sublist $L_i$ which points to it. Again three storage cells are required for such an element. Finally, we need two additional storage cells for each sublist $L_i$: a pointer to $L_{i+1}$ and a pointer to the last element in $L_i$. Altogether, $4n+O(\log n)$ storage cells suffice.

Also our algorithm is more time efficient than theirs. This is due to the fact that it uses AVL-trees instead of 2-3 trees and that list L above can be kept in any array.

## III. Conclusion

We presented a new sorting algorithm. Several variations of the general theme are possible.

1) Usage of some other kind of balanced trees instead of AVL-trees, e.g. B-trees [Bayer-McCreight].

2) List L starts with sublist $L_s$, $s \geq 1$, instead of $L_0$. This might remove some overhead.

3) It is conceivable to use the same datastructure recursively to organize list L. After all, list L has length log n and inserting element $x_p$ corresponds to finding the $(\log f_p)$-th position in this list.

4) Usage of random trees instead of balanced trees. This might result in a sorting algorithm with fast average running time.

5) We chose to delay cleaning list $L_{i+1}$ after it became dirty by a split of $T_{i,o}$. It would be possible to clean list $L_{i+1}$ (and $L_{i+2}, L_{i+3}, \ldots$ as necessary) immediately after that split. The same analysis and time bound applies. It requires more study which solution is more efficient with respect to time and space requirements.

## Bibliography

Adelson-Velskii-Landis: "An algorithm for the organization of information", Soviet. Math. Dokl,3,1259-1262, 1962

Aho, Hopcroft & Ullman: "The Design and Analysis of Computer Algorithms", Addison Wesley, 1974

Bayer & McCreight: "Organization and Maintenance of Large Ordered Indizes", Acta Informatica, 1 (1972), 173-189

Bentley & Yao: "An almost Optimal Algorithm for Unbounded Searching", Information Processing Letters, Vol. 5, No. 3, p. 82-87, August 1976

Brown & Tarjan: "A Representation for Linear Lists with Movable Fingers", 10th ACM Symposium on Theory of Computing, p. 19-29, 1978

Fredman, M.L.: "Two applications of a Probabilistic Search Technique: Sorting X + Y and Building Balanced Search Trees", 7th ACM Symposium on Theory of Computing, 1975 240-244

Guibas, Creight, Plass, Roberts : "A new representation for linear lists, 9th ACM Symposium on Theory of Computing, 1977, 49-60

Mehlhorn, K.: "Effiziente Algorithmen", Teubner Studienbücher Informatik, Stuttgart 1977