Sorting Jordan Sequences in Linear Time

A 09/84

Kurt Hoffmann/Kurt Mehlhorn
Universität des Saarlandes
Saarbrücken, Germany

Pierre Rosenstiehl
Centre de Mathematique Sociale
Paris, France

Robert E. Tarjan
AT&T Bell Laboratories, Murray Hill, NJ 07974

Abstract

For a Jordan curve C in the plane, let $x_1, x_2, \ldots, x_n$ be the abscissas of the intersection points of C with the x-axis, listed in the order the points occur on C. We call $x_1, x_2, \ldots, x_n$ a Jordan sequence. In this paper we describe an $O(n)$-time algorithm for recognizing and sorting Jordan sequences. The problem of sorting such sequences arises in computational geometry and computational geography. Our algorithm is based on a reduction of the recognition and sorting problem to a list-splitting problem. To solve the list-splitting problem we use level linked search trees.

Sorting Jordan Sequences in Linear Time

Kurt Hoffmann
Kurt Mehlhorn
Universität des Saarlandes
Saarbrücken, Germany

Pierre Rosenstiehl
Centre de Mathematique Sociale
Paris, France

Robert E. Tarjan
AT&T Bell Laboratories
Murray Hill, NJ 07974

1. Introduction

Let C be a Jordan curve in the plane and let $x_1, x_2, \ldots, x_n$ be the abscissas of the intersection points of C with the x-axis, listed in the order the points occur on C. (See Figure 1.) We call a sequence of real numbers $x_1, x_2, \ldots, x_n$ obtainable in this way a Jordan sequence. In this paper we consider the problem of recognizing and sorting Jordan sequences.

[Figure 1]

The Jordan sequence sorting problem arises in at least two different contexts. Edelsbrunner [5] has posed the problem of computing the sorted list of intersections of a simple n-sided polygon with a line. This problem is linear-time equivalent to the problem of sorting a Jordan sequence, since we can represent the line parametrically and compute the list of intersections in the order they occur along the polygon in linear time by computing the intersection of the line with each side of the polygon. (We assume that the sides of the polygon are given in the order they occur along the polygon.) Iri [11] has encountered the problem in the context of computational geography: for two Jordan curves A and B, we are given the list of their intersection points in the order they occur along A and asked to sort them in the order they occur along B, using as a unit-time primitive the operation of comparing two intersection points with respect to their order along B. Any comparison-based algorithm for the Jordan sequence sorting problem will solve Iri's problem as well.

We call a Jordan sequence a Jordan permutation if the sequence consists of the integers 1 through n in some order. Any Jordan permutation determines two nested sets of parentheses [16]. (See Section 2.) It follows that there are at most c^n Jordan permutations of 1 through n, where c is a constant independent of n. This implies by a result of Fredman [6] that Jordan sequences can be sorted in O(n) binary comparisons. Unfortunately the algorithm implied by Fredman's result has non-linear overhead. Our goal is to provide an algorithm that runs in linear time including overhead.

Our approach to the Jordan sequence sorting problem is to convert it into a data manipulation problem that involves repeated splitting of lists. We discuss this transformation in Section 2. In Section 3 we solve the list-splitting problem using an extension of level-linked search trees [4,10,15], thus obtaining a linear-time algorithm for recognizing and sorting Jordan sequences. Section 4 contains some final remarks.

Historical Note. The algorithm presented here was discovered by the first pair of authors and by the second pair of authors working independently. A sketch of the first pair's solution was presented in [7].

2. Jordan Sequences and List-Splitting

Let $x_1, x_2, \ldots, x_n$ be a Jordan sequence, and suppose without loss of generality that the Jordan curve C defining the sequence starts below the x-axis. (If not, reflect it about the x-axis.) Let $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$ be the numbers $x_1, x_2, \ldots, x_n$ permuted into sorted order. Each pair $\{x_{2i-1}, x_{2i}\}$ for $i \in [1..\lfloor n/2 \rfloor]$ corresponds to a part of C starting on the x-axis at x_{2i-1}, rising above it, and returning to the x-axis at x_{2i}. Since C never crosses itself, any two such pairs $\{x_{2i-1}, x_{2i}\}, \{x_{2j-1}, x_{2j}\}$ must nest: if either of x_{2j-1} or x_{2j} lies between x_{2i-1} and x_{2i}, then so does the other. This means that we can construct a set of $\lfloor n/2 \rfloor$ nested parentheses corresponding to the pairs $\{x_{2i-1}, x_{2i}\}$: in the sorted sequence $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$, replace x_{2i-1} and x_{2i} for $i \in [1..\lfloor n/2 \rfloor]$ by a matched left and right parenthesis, with the left parenthesis replacing the smaller of x_{2i-1} and x_{2i} and the right parenthesis replacing the larger. (If n is odd, we merely delete x_n.) Similarly, the pairs $\{x_{2i}, x_{2i+1}\}$ for $i \in [1..\lfloor n/2 \rfloor]$ correspond to a set of $\lfloor n/2 \rfloor$ nested parentheses representing connected parts of C below the x-axis. (See Figure 2.)

[Figure 2]

We need some notation. For a pair $\{x_{i-1},x_i\}$, we define $y_i = \min\{x_{i-1},x_i\}$ and $z_i = \max\{x_{i-1},x_i\}$. Thus $\{y_i,z_i\} = \{x_{i-1},x_i\}$. We say the pair $\{x_{i-1},x_i\}$ <u>encloses</u> a number r if $y_i<r<z_i$. Similarly, $\{x_{i-1},x_i\}$ encloses a pair $\{x_{j-1},x_j\}$ if $y_i<y_j<z_j<z_i$. The <u>parent</u> of a pair $\{x_{i-1},x_i\}$ is the enclosing pair $\{x_{j-1},x_j\}$ with $i = j \bmod 2$ and y_j maximum. With this definition the pairs $\{x_{2i-1},x_{2i}\}$ together with their parent relation define a forest of rooted trees called the <u>upper forest</u> of $x_1,x_2,\ldots,x_n$. Similarly the pairs $\{x_{2i},x_{2i+1}\}$ together with their parent relation define the <u>lower forest</u> of $x_1,x_2,\ldots,x_n$. If $\{x_{i-1},x_i\}$ and $\{x_{j-1},x_j\}$ are siblings in either the upper or lower forest, we order them by putting $\{x_{i-1},x_i\}$ first if $y_i < y_j$. This makes each forest into an ordered forest. We make the forests into trees by adding a dummy pair $\{-\infty,\infty\}$ to each and declaring it to be the parent of any pair not otherwise having a parent. Thus we obtain two trees called the <u>upper tree</u> and the <u>lower tree</u>. (See Figure 3.)

[Figure 3]

To sort a Jordan sequence $x_1,x_2,\ldots,x_n$, we process the numbers x_i in increasing order on i, constructing three objects simultaneously: a sorted list of the numbers so far processed, and the upper and lower trees of the pairs corresponding to the numbers so far processed. Initially the sorted list contains

$-\infty$ and ∞ and the upper and lower trees each consist of the single pair $\{-\infty,\infty\}$. We process x_i by adding pair $\{x_{i-1},x_i\}$ to the appropriate tree (unless $i = 1$) and inserting x_i into the sorted list. The process of inserting $\{x_{i-1},x_i\}$ into its tree provides the approximate location of x_i in the sorted list, so that it can be inserted in $O(1)$ time.

The details of processing x_i are as follows. If $i - 1$ we merely insert x_i in the sorted list between $-\infty$ and ∞. Otherwise, we locate the numbers $r < x_{i-1}$ and $s > x_{i-1}$ adjacent to x_{i-1} in the sorted list. Let $\{x_{j-1},x_j\}$ and $\{x_{k-1},x_k\}$ be the pairs containing r and s such that $i \equiv k \equiv k \bmod 2$. (Either or both of these pairs may be the dummy pair $\{-\infty,\infty\}$.) If both $\{x_{j-1},x_j\}$ and $\{x_{k-1},x_k\}$ enclose x_i, it must be the case that $\{x_{j-1},x_j\} = \{x_{k-1},x_k\} = \{r,s\}$; otherwise $x_1,x_2,\ldots,x_n$ is not a Jordan sequence and we abort the algorithm. If this condition is met, we insert x_i into the sorted list before or after x_{i-1} as appropriate. Also, we construct a new fmaily with parent $\{r,s\}$ and $\{x_{i-1},x_i\}$ as its only child.

On the other hand, suppose one of the pairs, say $\{x_{j-1},x_j\}$, does not enclose x_i. We access the family* containing $\{x_{j-1},x_j\}$ as a child (in the upper tree if i is even, the lower tree if i is odd) and split the list of children into two lists, containing those children enclosed by $\{x_{i-1},x_i\}$ and those not. There may be a child that is a pair having exactly one element (rather than zero or two) enclosed by $\{x_{i-1},x_i\}$; if we find such a pair we abort the algorithm, as $x_1,x_2,\ldots,x_n$ is not a Jordan

---

*By a <u>family</u> we mean a tree node and its list of children.

sequence. Otherwise, we construct a new family with parent $\{x_{i-1},x_i\}$, having as children the children in the old family enclosed by $\{x_{i-1},x_i\}$; in the old family, we replace these removed children by $\{x_{i-1},x_i\}$. (See Figure 4.) Then we insert x_i in the sorted list.

[Figure 4]

If this algorithm successfully processes $x_1,x_2,\ldots,x_n$, then $x_1,x_2,\ldots,x_n$ is indeed a Jordan sequence and is sorted by the algorithm. If the algorithm aborts, then $x_1,x_2,\ldots,x_n$ is not a Jordan sequence.

In order to estimate the running time of the algorithm, we need to say more about the data structures used to implement the method. We represent the sorted list of processed numbers by a doubly-linked list, so that accessing the number before or after a given one or inserting a new number before or after a given one takes O(1) time. We store the numbers x_i in an array indexed on i, so that given i we can in O(1) time access x_i, x_{i-1}, or x_{i+1}. We store each family in the upper or lower forest as a sorted list of the numbers in its constituent pairs. Each number x_i occurs at most four times in such family lists, since it is in at most two pairs ($\{x_{i-1},x_i\}$ and $\{x_i,x_{i+1}\}$), each of which occurs in at most two families (as a parent and as a child).

The crucial operations are those on the family lists. Given a pair $\{x_{i-1},x_i\}$ and a number u_p in a family list $u_1,u_2,\ldots,u_\ell$, such that $u_{p-1}<x_{i-1}<u_p$ or $u_p<x_{i-1}<u_{p+1}$, we must find the number u_q such that $u_q<x_i<u_{q+1}$, test the appropriate pairs containing u_q and u_{q+1} to see if they and $\{x_{i-1},x_i\}$ violate the nesting property, split the family list in two and add x_{i-1} and x_i to the new family lists. (If $u_{p-1}<x_{i-1}<u_p$ and $x_{i-1}<x_i$, the final family lists are $u_1,\ldots,u_{p-1},x_{i-1},x_i,u_{q+1},\ldots,u_\ell$ and $x_{i-1},u_p,\ldots,u_q,x_i$; the other three possible cases are similar.) Then we must insert x_i near u_q in the sorted list of processed numbers. (Numbers x_i fits immediately after u_q unless i is odd and x_1 lies betwe.n u_q and x_i, in which case x_i fits after x_1. This anomaly occurs because x_1 is not represented in the lower tree.)

The total time required by the algorithm is O(n) not counting the time to split family lists and to find the positions at which to split. We shall discuss two ways of implementing family lists. One way is to represent each family list as a circular doubly-linked list. With such a representation the time to insert a new item next to a given one is O(1) and the time to perform the splitting and insertion described above is $O(\min\{|p-q|,\ell-|p-q|\})$. (To find u_q, we begin at u_p and search simultaneously for u_q in both directions around the circular list.)

To estimate the list-splitting and insertion time with this representation, let $T_1(\ell,m)$ be the worst-case time to carry out ℓ successive splittings and insertions on an initial list of size m. Then $T_1(\ell,m)$ is bounded by the following recurrence:

$$T_1(0,m) = 0;$$

$$T_1(\ell,m) \le \max_{\substack{0\le i\le \ell-1 \\ 0\le j\le m}} (T_1(i,j+2) + T_1(\ell-i-1,m-j+2) + O(\min\{j,m-j\})) \text{ for } \ell\ge 1.$$

To estimate $T_1(\ell,m)$, let $T_1'(n)$ satisfy the following recurrence, where the "O" term is the same as that in the bound on T_1:

$$T_1'(1) = O(1);$$

$$T_1'(n) = \max_{1\le i\le n-1} (T_1'(i) + T_1'(n-i) + O(\min\{i,n-i\})) \text{ for } n\ge 2.$$

A straightforward induction shows that $T_1(\ell,m) \le T_1'(4\ell+m)$. It is well-known that $T_1'(n) = O(n \log n)$ [1,14], which implies $T_1(\ell,m) = O((\ell+m)\log(\ell+m))$.

To obtain a time bound for the Jordan sequence sorting algorithm, we note that there are two sets of $\lfloor n/2\rfloor$ list-splittings, on the families of the upper tree and on the families of the lower tree. The initial family list for each tree contains two items. Thus the total time for list-splitting is $2T_1(\lfloor n/2\rfloor,2) = O(n \log n)$, and the total time for the entire algorithm is $O(n \log n)$.

This $O(n \log n)$ bound is no better than what we can obtain using any fast general-purpose sorting method. We can speed up the algorithm by changing our implementation of the family lists. In the next section we shall develop a representation such that the amortized time* to insert a new item next to a given one is O(1) and the amortized time to carry out the list-splitting operation described above is $O(\log \min\{|p-q|, \ell-|p-q|\})$. With this representation, if $T_2(\ell,m)$ is the worst-case time to carry out ℓ successive splittings on an initial list of size m, $T_2(\ell,m)$ is bounded by the following recurrence:

$$T_2(0,m) = 0;$$

$$T_2(\ell,m) \le \max_{\substack{0\le i\le \ell-1 \\ 1\le j\le m}} (T_2(i,j+2) + T_2(\ell-i-1,m-j+2) + O(\log \min\{j,m-j\})) \text{ for } \ell\ge 1.$$

Let $T_2'(n)$ satisfy the following recurrence, where the "O" term is the same as that in the bound on T_2:

$$T_2'(1) = O(1);$$

$$T_2'(n) = \max_{1\le i\le n-1} (T_2'(i) + T_2'(n-i) + O(\log \min\{i,n-i\})) \text{ for } n\ge 2.$$

Then $T_2(\ell,m) \le T_2'(4\ell+m)$ and $T_2'(n) = O(n)$ [15,p.185], from which we obtain that the Jordan sequence sorting algorithm runs in O(n) time.

3. List-Splitting Using Level-Linked Search Trees

In order to represent lists so that splitting is efficient, we shall use an extension of level-linked 2,4 trees

*By amortized time we mean the time of an operation averaged over a worst-case sequence of operations. See [15,18]. We discuss this concept more fully in the next section.

[10,15]. Although our presentation is self-contained, some familiarity with search trees and especially with 2,3 trees [1] or B-trees [2] will help the reader.

A *2,4 tree* is an ordered tree in which all external nodes have the same depth and each internal node has two, three, or four children. We can represent an ordered list of items using a 2,4 tree by storing the items in the external nodes in left-to-right order. In addition, we store in each internal node the maximum item in each of its subtrees except the last. Thus an internal node with d children, which we call a *d-node*, contains d-1 items, which we call *keys*. Each item except the last occurs exactly once as a key. As an exceptional special case, we store the last item in the tree root; otherwise this last item would not appear as a key.

Remark. Although the root contains an extra key, we can avoid using extra space for the root node by maintaining the tree so that the root has at most three children. Small changes in the update algorithms discussed below suffice for this purpose. □

To represent such a tree, we store in each node pointers to its parent and children. With each item we store pointers to its two locations in the tree. In addition, we make the tree *level-linked*: each node points to the nodes preceding and following it at the same height, called its *left neighbor* and its *right neighbor*, respectively. The level links are circular, so that the last node on a level points to the first and vice-versa. The tree root points to itself. (See Figure 5.)

[Figure 5]

Such a data structure is a (circularly) level-linked 2,4 tree. Level-linked 2,4 trees were invented by Huddleston and Mehlhorn [10] as an extension of the level-linked 2,3 trees of Brown and Tarjan [4]. Essentially the same extension was proposed by Maier and Salveter [13]. Our innovation is to make the level links circular, which speeds up access operations and splitting, as we shall see.

The purpose of level links is to make it possible to access any item in the tree starting from any other item in time proportional to the logarithm of the number of items separating them. Suppose x is an item in the tree and we wish to find the smallest item in the tree no smaller than some other item y. Suppose $x < y$. (The case $x > y$ is similar.) Starting at the external node containing x, we follow parent pointers up through the tree until reaching a node e such that e is on the right path* of the tree, or the right neighbor of e contains a key greater than or equal to y, or e is on the left path of the tree and its left neighbor (which is on the right path) contains a key less than or equal to y. We have now located one or possibly two subtrees in which the item sought must appear. (The two-subtree case occurs when y is greater than all keys in e and less than all keys in its right neighbor. Then the desired item is either in the rightmost subtree of e or in the leftmost subtree of its right neighbor, but we cannot in the worst case tell

*By the right path of a tree, we mean the path from the root to the rightmost external node. We define the left path symmetrically. A node is on the right path if and only if it is the root or its right neighbor contains smaller items than it does.

which without searching both.) We search down in the appropriate subtree or subtrees, following child pointers and guided by keys, until finding the desired item. (See Figure 5.)

The time required for such a search is proportional to the maximum height, say h, reached during the search. If n is the total number of items in the tree and d is the number of items greater than x and less than y, then $h = O(\log \min\{d,n-d\})$. To prove this, we note that a 2,4 tree of height i contains at least 2^i items. Let f be the first node reached at height h-1 during the search. The leftmost subtree of the right neighbor of f contains only items greater than x and less than y. Since this subtree is a 2,4 tree of height h-2, we have $d \geq 2^{h-2}$. Similarly the rightmost subtree of the left neighbor of f contains only items less than x or greater than y, which implies $n-d \geq 2^{h-2}$. Combining these bounds, we obtain $h \leq \log \min\{d,n-d\} + 2$. Thus the access operation takes $O(\log \min\{d,n-d\})$ time. This bound improves Brown and Tarjan's $O(\log d)$ bound for access operations in level-linked trees without circular linking.

It remains for us to describe how to update level-linked 2,4 trees. We shall discuss the various update operations only as they affect the tree structure; it is easy to verify that keys and level links can be updated in the claimed time bounds.

Insertion and deletion in 2,4 trees were discussed by Huddleston and Mehlhorn [10] and Maier and Salveter [13]; we shall review their algorithms and analysis (see also [15], Section III.5). To insert a new item x in a tree next to a given one y, we create a new external node to hold x and make its parent the same as that of the external node containing y. This may convert the parent from a 4-node into a 5-node, which is not allowed in a 2,4 tree. We split such a 5-node into a 2-node and a 3-node. This may create a new 5-node, which we split in turn. We continue splitting newly created 5-nodes, moving up the tree, until either the root splits or no new 5-node is created. (See Figure 6.) If the root splits, we create a new root, a 2-node, causing the tree to grow in height by one. The time needed for the insertion is proportional to one plus the number of splits.

[Figure 6]

Deletion is an inverse process, only slightly more complicated. To delete a given item, we destroy the external node containing it. This may make the parent a 1-node. If this 1-node has a neighboring sibling that is a 3-node or a 4-node, we move a child of this neighbor to the 1-node and the deletion stops. (This is called <u>borrowing</u>.) If the 1-node has a neighboring sibling that is a 2-node, we combine the 1-node and the 2-node. (This is called <u>fusing</u>.) Fusing may produce a new 1-node, which we eliminate in the same way. We move up the tree eliminating 1-nodes until either a borrowing occurs or the root becomes a 1-node, which we destroy. (See Figure 7.) The time needed for the deletion is proportional to one plus the number of fusings.

[Figure 7]

In order to obtain a tight estimate of the time for insertions and deletions, we shall amortize, i.e. average the time over a worst-case sequence of operations. Huddleston and Mehlhorn [10] and independently Maier and Salveter [13] did an amortized analysis of insertion and deletion in 2,4 trees. We shall restate their results in the "potential" paradigm [14, Section III.6.1;18].

We define the potential of a collection of 2,4 trees to be twice the number of 4-nodes plus the number of 2-nodes. We define the amortized time of an operation to be the actual time of the operation plus the net increase in potential caused by the operation. With this definition, the total actual time of a sequence of operations is the total amortized time plus the net potential drop over the sequence. If initially there are no trees, then the net potential drop over any sequence is non-positive, since the final potential is non-negative. Thus the sum of the amortized times is an upper bound on the sum of the actual times, and we can use the amortized times as a valid estimate of the complexity of the operations.

Let us define the actual time of an insertion to be one plus the number of splits. Then the amortized time of an insertion is at most three: each split costs one but converts a node that was originally a 4-node into a 2-node and a 3-node, for a net potential drop of one; in addition, the insertion can create one new 2-node or 4-node. Similarly, if we define the time of a deletion to be one plus the number of fusings, the amortized cost is at most two: each fusing costs one but converts two nodes that were originally 2-nodes into a 3-node, for a net potential drop of two; in addition, the deletion can create one new 2-node.

This O(1) amortized time bound for insertion and deletion, derived by Huddleston and Mehlhorn [10] and Maier and Salveter [13], generalizes the O(1) bound per operation derived by Brown and Tarjan [4] for a sequence of pure insertions or a sequence of pure deletions in a 2,3 tree; the extra flexibility of the balance condition in 2,4 trees means that insertions and deletions can be freely intermixed while maintaining the O(1) bound per operation. This advantage of 2,4 trees is crucial in our application.

The O(1) bound for insertion and deletion generalizes to the elimination of a single 5-node or a single 1-node in a 2,4 tree. To eliminate a 5-node, we walk up the path toward the root splitting 5-nodes as in insertion; to eliminate a 1-node, we walk up the path toward the root fusing 1-nodes as in deletion. The ability to eliminate 5-nodes and 1-nodes allows us to devise conceptually straightforward algorithms for joining of 2,4 trees and for a simple form of splitting.

Suppose T_1 and T_2 are 2,4 trees such that all items in T_1 are less than all items in T_2, and we wish to combine the trees to form a single tree representing the concatenation of the lists represented by T_1 and T_2. From the root of T_i for

$i = 1,2$, we can access the largest item in the tree, hence the rightmost external node, and from there the leftmost external node, in $O(1)$ time. To carry out the join, we walk up the right path of T_1 and the left path of T_2 until reaching the root of one or the other. Let h_1 and h_2 be the heights of T_1 and T_2 respectively, and let $h = \min\{h_1,h_2\}$. If $h_1 = h_2$, we create a new 2-node whose children are the roots of T_1 and T_2. If $h_1 < h_2$, we make the root of T_1 a child of the node on the left path of T_2 of height $h + 1$, and eliminate the resulting 5-node if this creates one. We proceed symmetrically if $h_1 > h_2$. The amortized time of concatenation is $O(h)$.

Suppose T is a 2,4 tree containing an item x, and we wish to split T into two trees, T_1 containing all items less than or equal to x, and T_2 containing all items greater than or equal to x. To perform the splitting, we walk up the path from the external node containing x to the root, splitting each node along the path into two, one whose subtrees contain items less than or equal to x and the other whose subtrees contain items greater than or equal to x. Once the root is split, we have trees T_1 and T_2 as desired, except that T_1 may have some 1-nodes along its right path and T_2 may have some 1-nodes along its left path. We eliminate these 1-nodes in top-to-bottom order by iterated fusing. The amortized time needed for such a splitting is $O(h)$, where h is the height of T.

Remark. An alternative way to implement splitting is to use repeated joining [1]. The time bound is the same, but it is worst-case rather than amortized. This is not important for our purposes. $\square$

Now we are ready to consider the kind of splitting needed in the Jordan sequence sorting algorithm. Given a 2,4 tree T and two items $x \leq y$ in it, such that there are d items greater than x and less than y, we wish to split out these d items into a new tree, with the old tree containing the remaining items. We shall describe a method with an amortized running time of $O(\min \log\{d,n-d\}$, where n is the total number of items.

The first step of the splitting is to walk up toward the root concurrently from the external nodes containing x and y until reaching a common node, say e, or two neighboring nodes, say e whose subtree contains x and f whose subtree contains y. To complete the splitting we apply the appropriate one of the following three cases.

<u>Wrap-around</u>: f exists and is the left neighbor of e. (Node e is on the left path and f is on the right path.) Detach the subtree rooted at e, say T^L, from its parent. If the parent becomes a 1-node, eliminate it by repeated fusing. Split T at x into T_1^L containing items less than or equal to x and T_2^L containing the rest. Concatenate T_2^L with what is left of the original tree. Proceed symmetrically on f to obtain T_2^R containing items greater than or equal to y. Concatenate T_1^L and T_2^R.

<u>Single root</u>: f does not exist. Detach the subtree rooted at e, say T^M, from its parent. Split T^M into T_1^M, T_2^M such that T_1^M contains items less than or equal to x. Split T_2^M into T_3^M, T_4^M such that T_4^M contains items greater than

or equal to y. Concatenate T_1^M and T_4^M. If the root of the resulting tree has height less than or equal to h, extend the tree by adding 1-nodes at the top to make it of height h, reattach it in place of e, and eliminate the 1-nodes by repeated fusing. Otherwise the concatenated tree must have a 2-node of height h + 1 as its root. In this case, make the two children of this root children of the original parent of e, and eliminate the resulting 5-node if any by repeated splitting.

Double root: f exists and is the right neighbor of e. This case is similar to the single-root case: we split the subtree with root e at x, split the subtree with root f at y, and combine the pieces in the appropriate way.

If h is the height of node e, then $h = O(\min \log\{d,n-d\})$ by the same argument we used to obtain the bound on access time in level-linked 2,4 trees. In all three cases of splitting it is easy to verify that the amortized running time is $O(h)$.

The repertoire of 2,4 tree operations needed for the Jordan sequence sorting problem consists of access, insertion, and the form of splitting just discussed. The amortized time bounds we have derived for these operations imply an $O(n)$ running time for the sorting algorithm of Section 2, as we have shown there.

4. Remarks

We close this paper with two remarks. First, the kind of list-splitting necessary in the Jordan sequence sorting problem arises in other situations as well. Indeed, it is the most time-consuming part of an early planarity-testing algorithm devised by Hopcroft and Tarjan [8]. By using level-linked 2,4 trees in place of their doubly-linked lists, we reduce the running time of their algorithm from O(n log n) to O(n). This gives a third linear-time planarity-testing algorithm, the others being those of Hopcroft and Tarjan [9] and Lempel, Even, and Cederbaum [12] as implemented by Booth and Leuker [3]. Level-linked 2,4 trees, being a general-purpose list representation, undoubtedly have other applications remaining to be discovered.

Our second remark is that there may be a much simpler way to sort Jordan sequences in linear time: we merely insert the items in the sequence one-at-a-time into a splay tree (a self-adjusting form of binary search tree [17]) and then access them in sorted order. This algorithm certainly runs in O(n log n) time [17]. On the basis of Sleator and Tarjan's dynamic optimality conjecture [17], we conjecture that this algorithm in fact runs in O(n) time. If this is true, then there is a simple linear-time algorithm for recognizing Jordan sequences as well: we run the sorting algorithm until it stops or the O(n) time bound is exceeded. If it finishes within the time bound, we test the parenthesis nesting property, which can be done in O(n) time once the sequence is sorted. If the sorting algorithm runs too long, we stop and declare the sequence non-Jordan. We leave to the reader the problem of proving or disproving that sorting a Jordan sequence using a splay tree takes O(n) time.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[2] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," Acta Info. 1(1972), 173-189.

[3] K. S. Booth and G. S. Leuker, "Testing for the consecutive ones property, interval graphs, and graph planarity using P-Q tree algorithms," J. Comp. Sys. Sci. 13(1976), 335-379.

[4] M. R. Brown and R. E. Tarjan, "Design and analysis of a data structure for representing sorted lists," SIAM J. Comput. 9(1980), 594-614.

[5] H. Edelsbrunner, Problem P36, Bull. EATCS 21(October, 1983), 195.

[6] M. L. Fredman, "How good is the information theory bound in sorting?", Theor. Comp. Sci. 1(1976), 355-361.

[7] K. Hoffman and K. Mehlhorn, "Intersecting a line and a simple polygon," Bull. EATCS 22(February, 1984), 120-121.

[8] J. E. Hopcroft and R. E. Tarjan, "Planarity testing in V log V steps: extended abstract," Information Processing 71, Vol. 1 - Foundations and Systems, North-Holland, Amsterdam (1972), 85-90.

[9] J. E. Hopcroft and R. E. Tarjan, "Efficient planarity testing," J. Assoc. Comp. Mach. 21(1974), 549-568.

[10] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists," Acta Info. 17(1982), 157-184.

[11] M. Iri, private communication.

[12] A. Lempel, S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," Theory of Graphs: International Symposium: Rome, July, 1966, P. Rosenstiehl, ed., Gordon and Breach, New York, 1967, 215-232.

[13] D. Maier and C. Salveter, "Hysterical B-trees," Info. Proc. Lett. 12(1981), 199-202.

[14] K. Mehlhorn, "Nearly optimal binary search trees," Acta Info. 5(1975), 287-295.

[15] K. Mehlhorn, Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching, Springer-Verlag, Berlin, 1984.

[16] P. Rosenstiehl, "Planar permutations defined by two intersecting Jordan curves," Graph Theory and Combinatorics, Academic Press, London, 1984, 259-271.

[17] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees, "J. Assoc. Comp. Mach., to appear.

[18] R. E. Tarjan, "Amortized computational complexity," SIAM J. Alg. Disc. Meth., to appear.

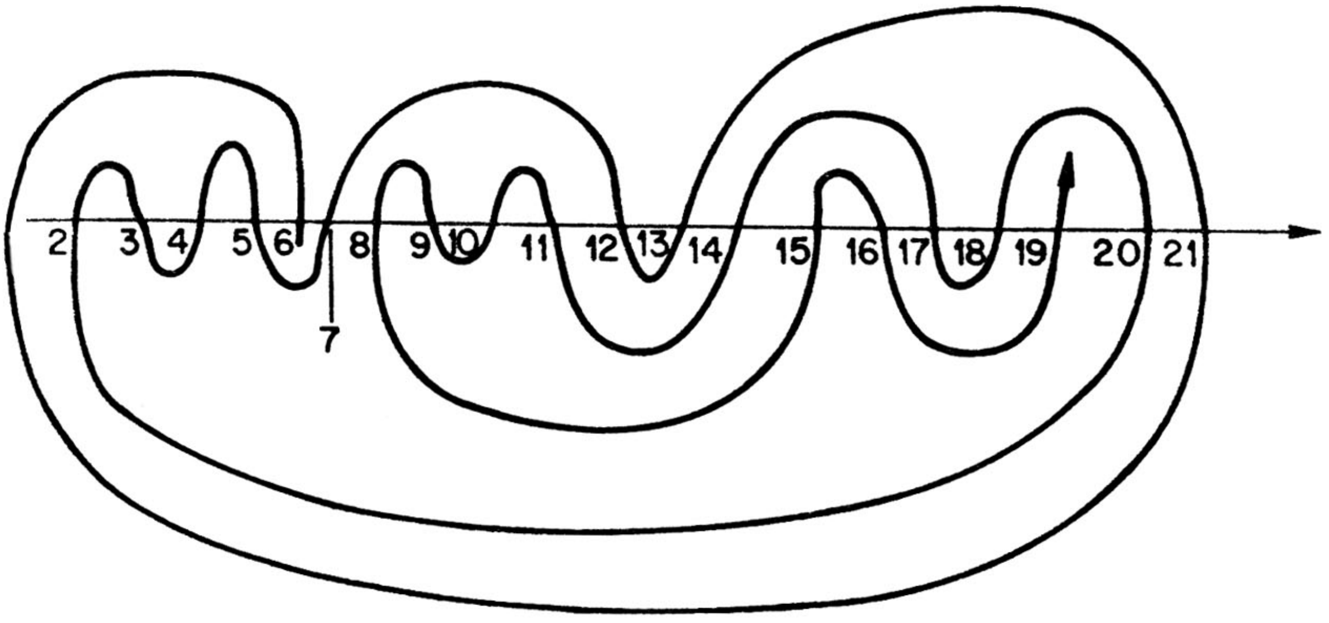


Figure 1. A Jordan curve corresponding to the sequence 6,1,21,13,12,7,5,4,3,2,20,18,17,14,11,10,9,8,15,16,19.

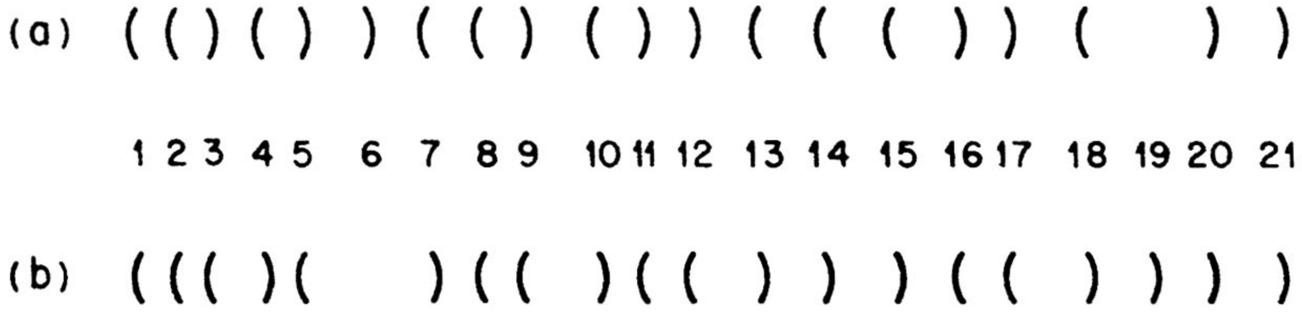


Figure 2. The nested parentheses corresponding to the Jordan curve of Figure 1.

(a) The parentheses corresponding to the pairs $\{x_{2i-1}, x_{2i}\}$.

(b) The parentheses corresponding to the pairs $\{x_{2i}, x_{2i+1}\}$.

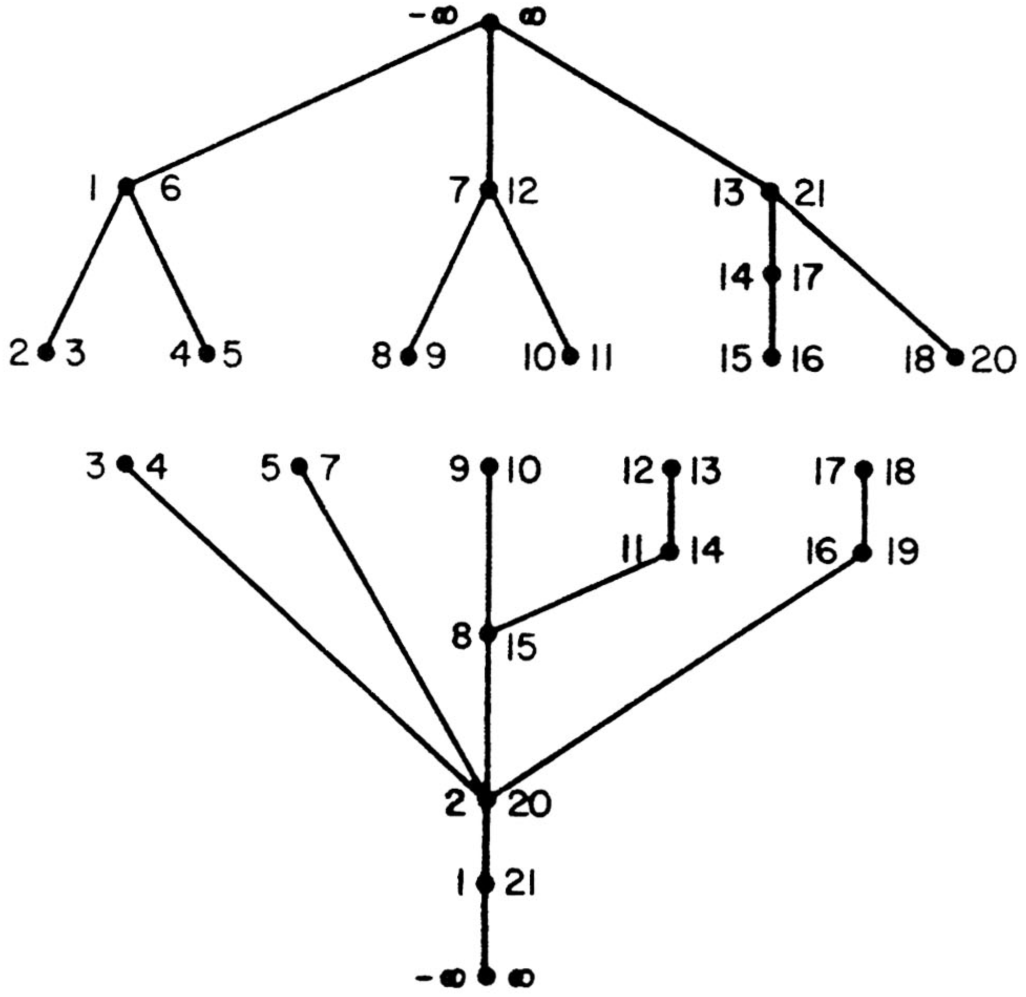
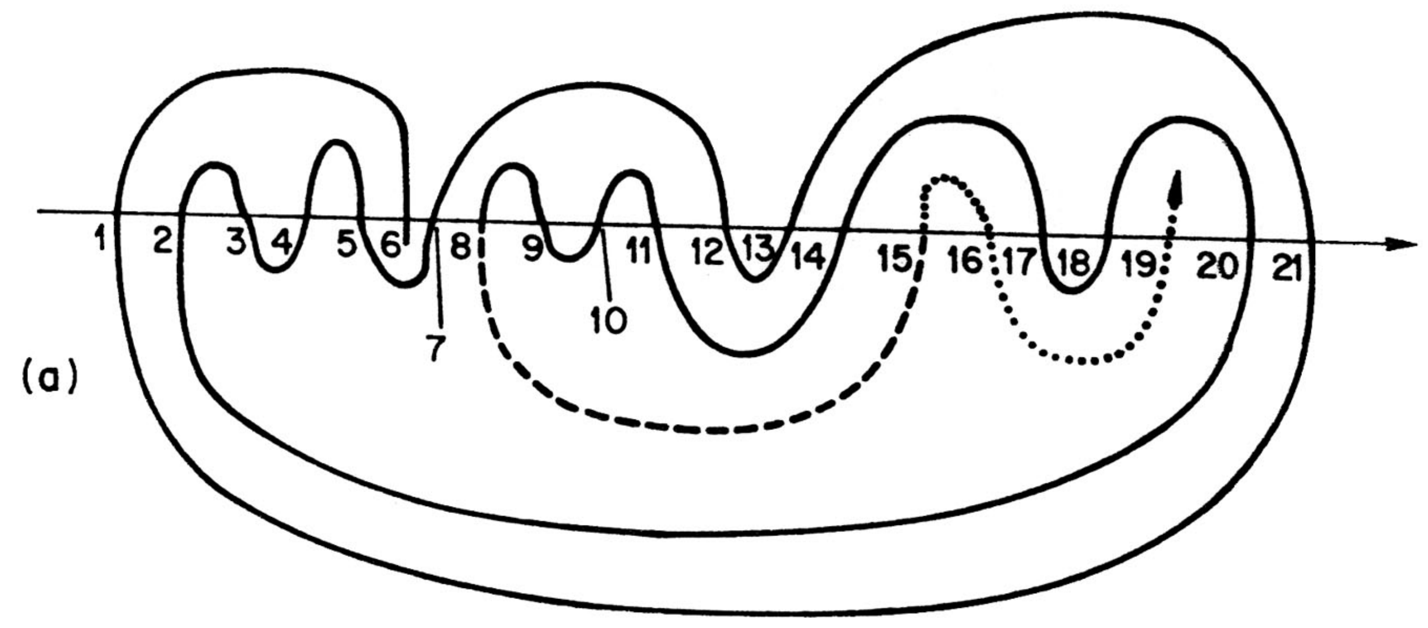


Figure 3. The upper and lower trees for the Jordan curve of Figure 1. The smaller and larger elements of each pair are on either side of the corresponding tree node.



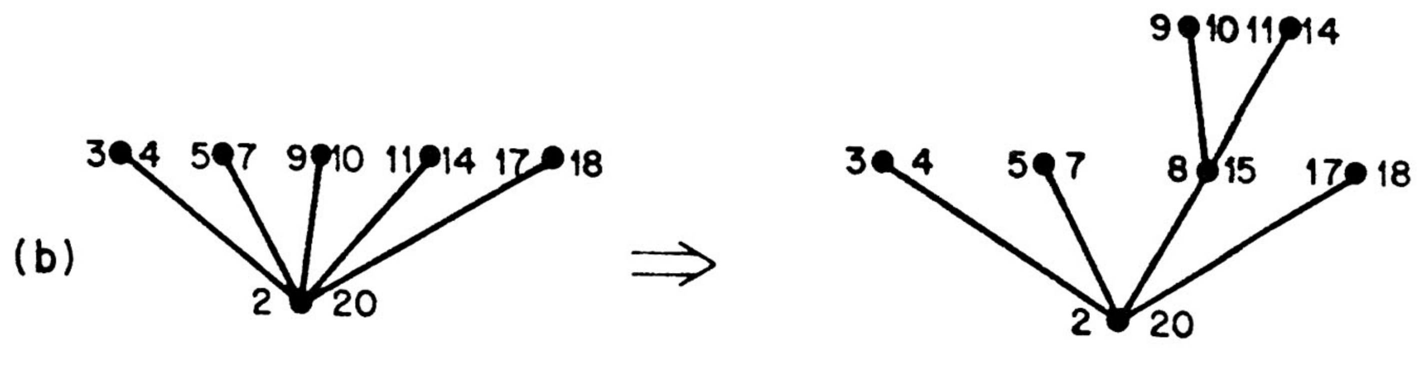


Figure 4. Splitting of a family in the lower tree during processing of 15 in the Jordan sequence of Figure 1.

(a) The Jordan curve. The solid part is already processed, the dashed part is being processed, and the dotted part is to be processed.

(b) The effect of inserting the pair {8,15} into the lower tree. Only the affected family is shown.

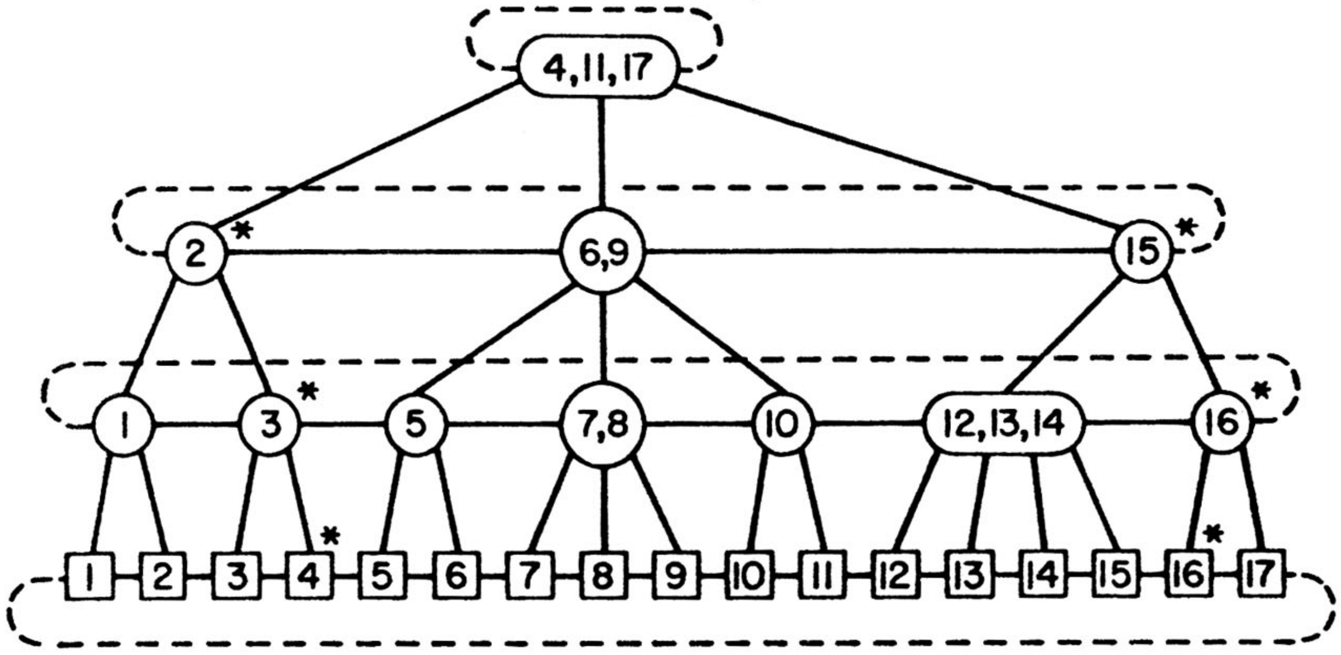


Figure 5. A level-linked 2,4 tree. The edges denote bidirectional links. The circular links are dashed. The starred nodes are those on the access path from 4 to 16.

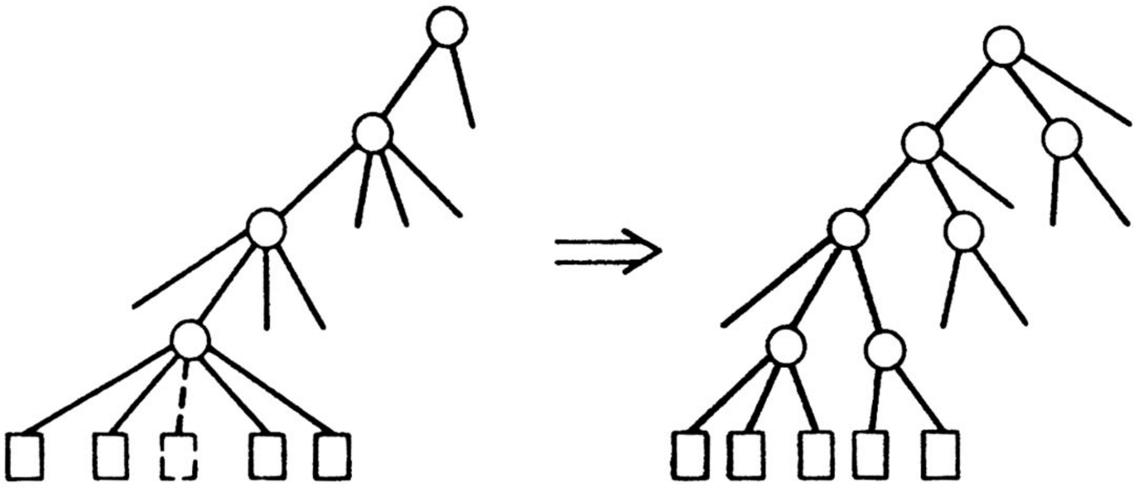


Figure 6. Insertion in a 2,4 tree. Only the affected part of the tree is shown. Three nodes are split.

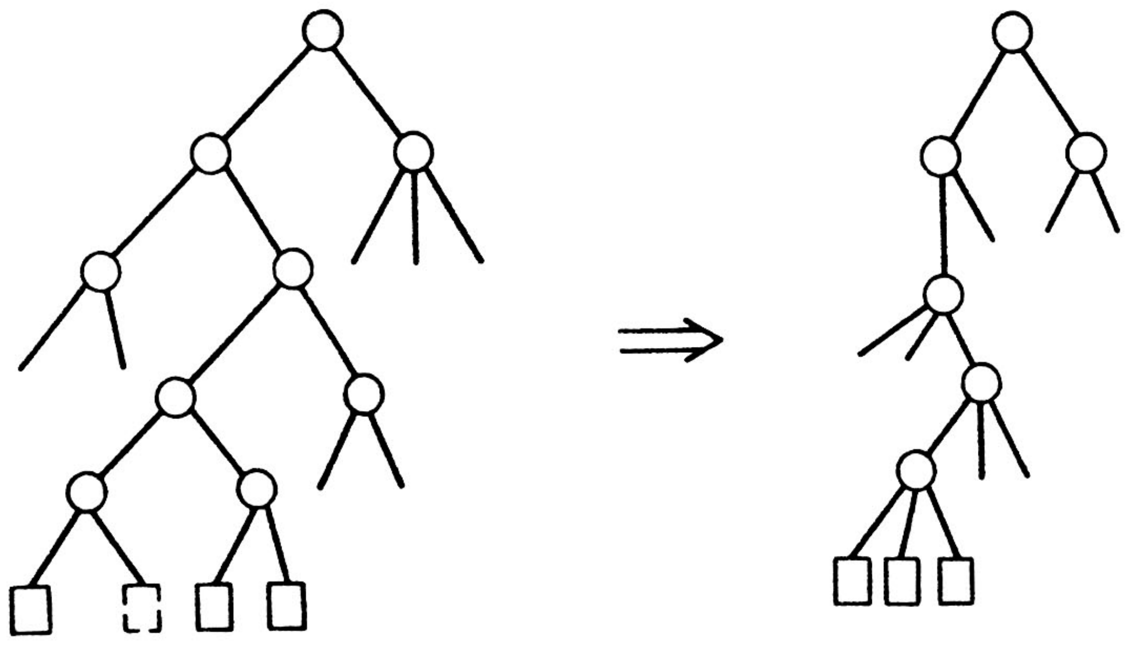


Figure 7. Deletion in a 2,4 tree. Only the affected part of the tree is shown. There are three fusings and one borrowing.