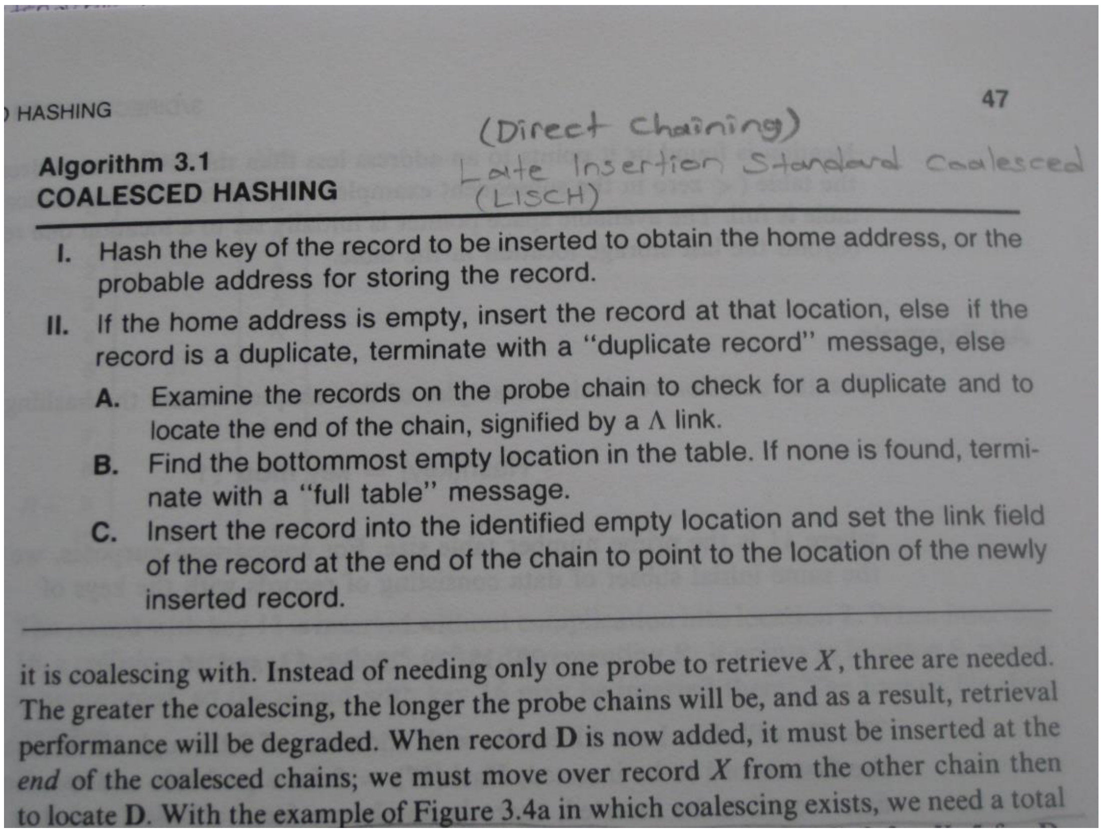
FILE ORGANIZATION AND PROCESSING (Wiley) ALAN L. THARP



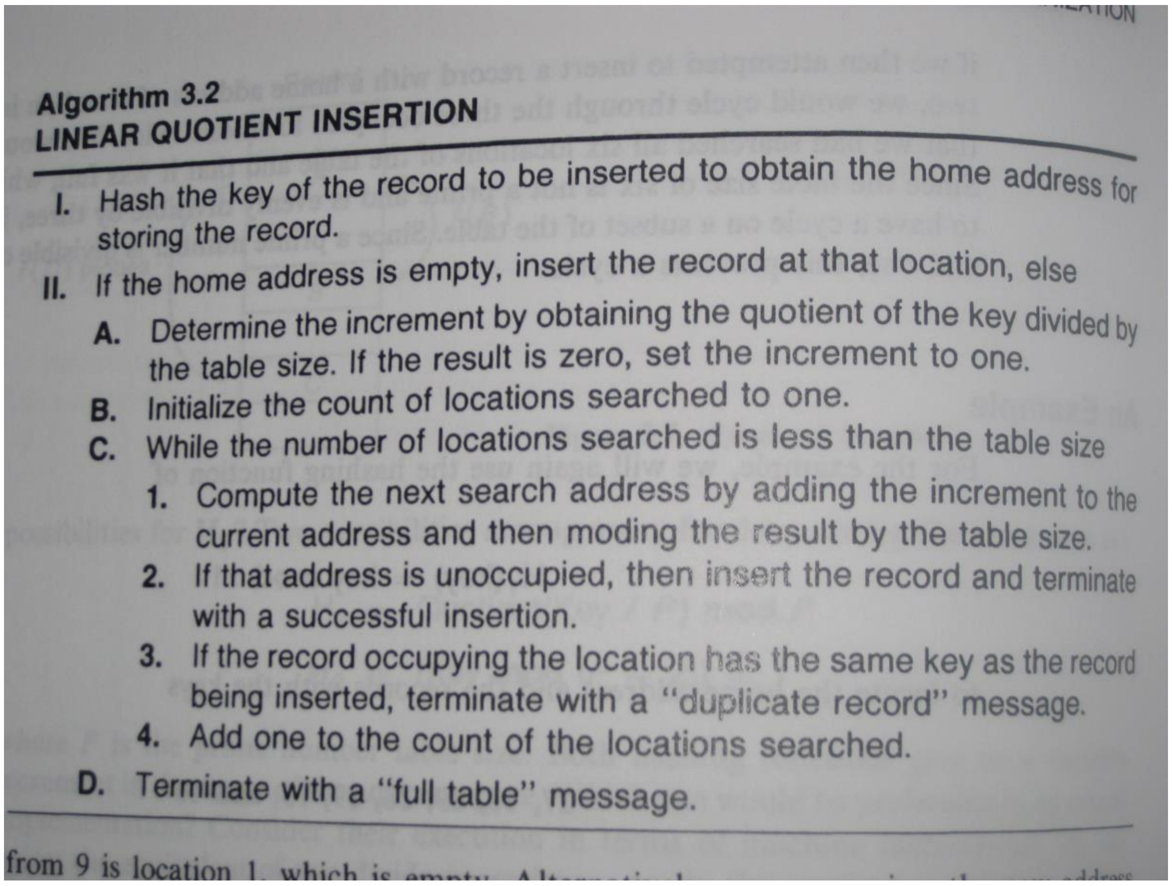
HASHING 47

Algorithm 3.1
COALESCED HASHING

(Direct Chaining)
Late Insertion Standard Coalesced
(LISCH)

```
I.  Hash the key of the record to be inserted to obtain the home address, or the
    probable address for storing the record.
II. If the home address is empty, insert the record at that location, else if the
    record is a duplicate, terminate with a "duplicate record" message, else
    A. Examine the records on the probe chain to check for a duplicate and to
       locate the end of the chain, signified by a Λ link.
    B. Find the bottommost empty location in the table. If none is found, termi-
       nate with a "full table" message.
    C. Insert the record into the identified empty location and set the link field
       of the record at the end of the chain to point to the location of the newly
       inserted record.
```

it is coalescing with. Instead of needing only one probe to retrieve X, three are needed. The greater the coalescing, the longer the probe chains will be, and as a result, retrieval performance will be degraded. When record **D** is now added, it must be inserted at the *end* of the coalesced chains; we must move over record X from the other chain then to locate **D**. With the example of Figure 3.4a in which coalescing exists, we need a total



Algorithm 3.2
LINEAR QUOTIENT INSERTION

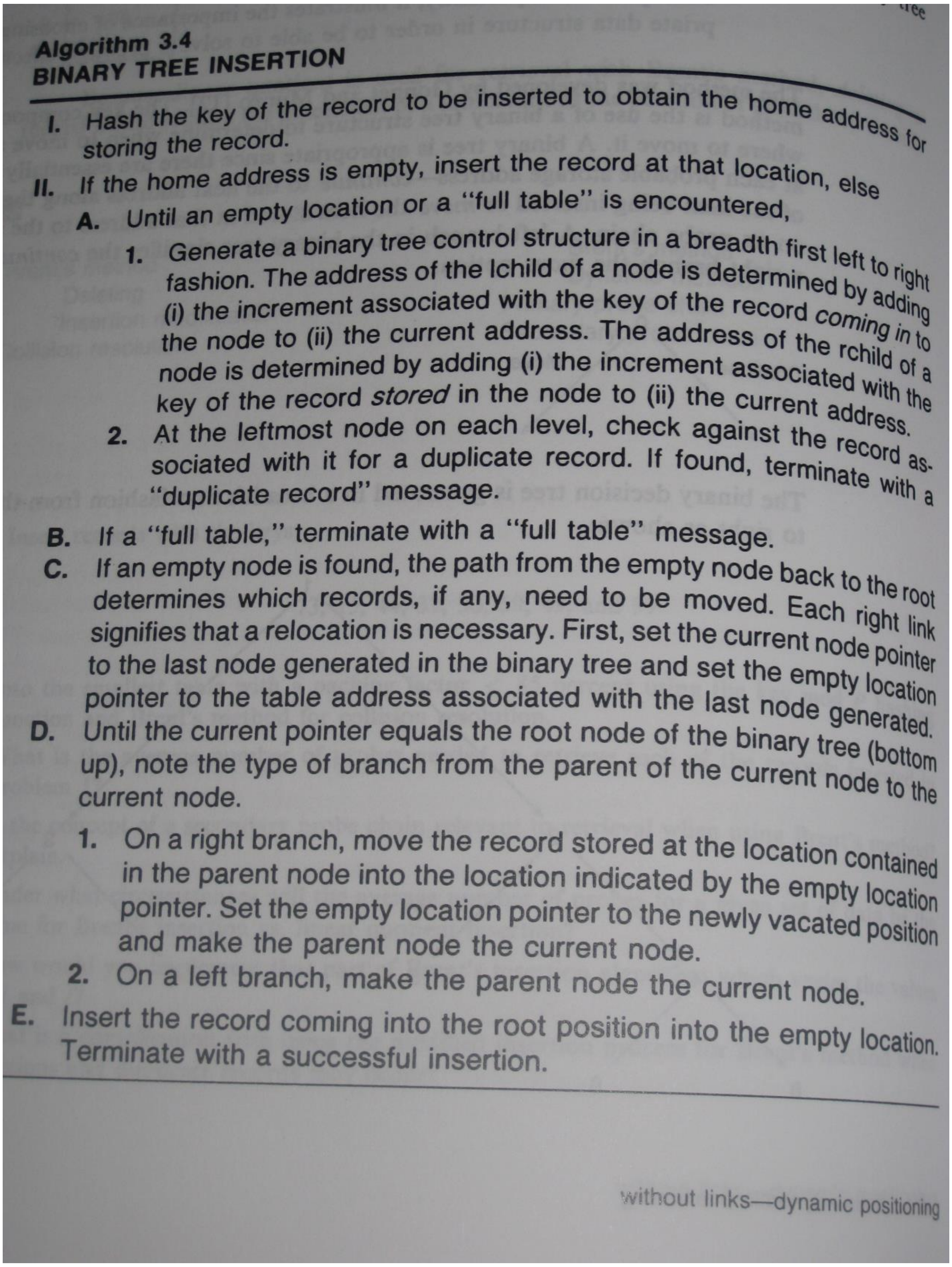
```
I.  Hash the key of the record to be inserted to obtain the home address for
    storing the record.
II. If the home address is empty, insert the record at that location, else
    A. Determine the increment by obtaining the quotient of the key divided by
       the table size. If the result is zero, set the increment to one.
    B. Initialize the count of locations searched to one.
    C. While the number of locations searched is less than the table size
       1. Compute the next search address by adding the increment to the
          current address and then moding the result by the table size.
       2. If that address is unoccupied, then insert the record and terminate
          with a successful insertion.
       3. If the record occupying the location has the same key as the record
          being inserted, terminate with a "duplicate record" message.
       4. Add one to the count of the locations searched.
    D. Terminate with a "full table" message.
```

from 9 is location 1, which is empty.

Algorithm 3.3
BRENT'S METHOD INSERTION

```
I.  Hash the key of the record to be inserted to obtain the home address for
    storing the record.
II. If the home address is empty, insert the record at that location, else
    A. Compute the next potential address for storing the incoming record.
       Initialize $s \leftarrow 2$.
    B. While the potential storage address is not empty,
       1. Check if it is the home address. If it is, the table is full, terminate with
          a "full table" message.
       2. If the record stored at the potential storage address is the same as
          the incoming record, terminate with a "duplicate record" message.
       3. Compute the next potential address for storing the incoming record.
          Set $s \leftarrow s + 1$.
/* Attempt to move a record previously inserted.*/
    C. Initialize $i \leftarrow 1$ and $j \leftarrow 1$.
    D. While $(i + j < s)$
       1. Determine if the record stored at the $i$th position on the primary
          probe chain can be moved $j$ offsets along its secondary probe chain.
       2. If it can be moved, then
          a. Move it and insert the incoming record into the vacated position
             $i$ along its primary probe chain; terminate with a successful inser-
             tion, else
          b. Vary $i$ and/or $j$ to minimize the sum of $(i + j)$; if $i = j$, minimize
             on $i$.
'* Moving has failed. */
    E. Insert the incoming record at position $s$ on its primary probe chain;
       terminate with a successful insertion.
```

nimize $(i + j)$. In the case where $i = j$, we will arbitrarily choose to minimize or
When should we terminate this process of attempting to move an item? When w
no longer achieve a reduction in the number of retrieval probes. Let s be the numbe
probes required to retrieve an item if nothing is moved. We then try all combination
$i + j) < s$ such that we minimize $(i + j)$. On equality, since there would be n
uction in the number of probes, no movement would occur. Figure 3.8 shows th

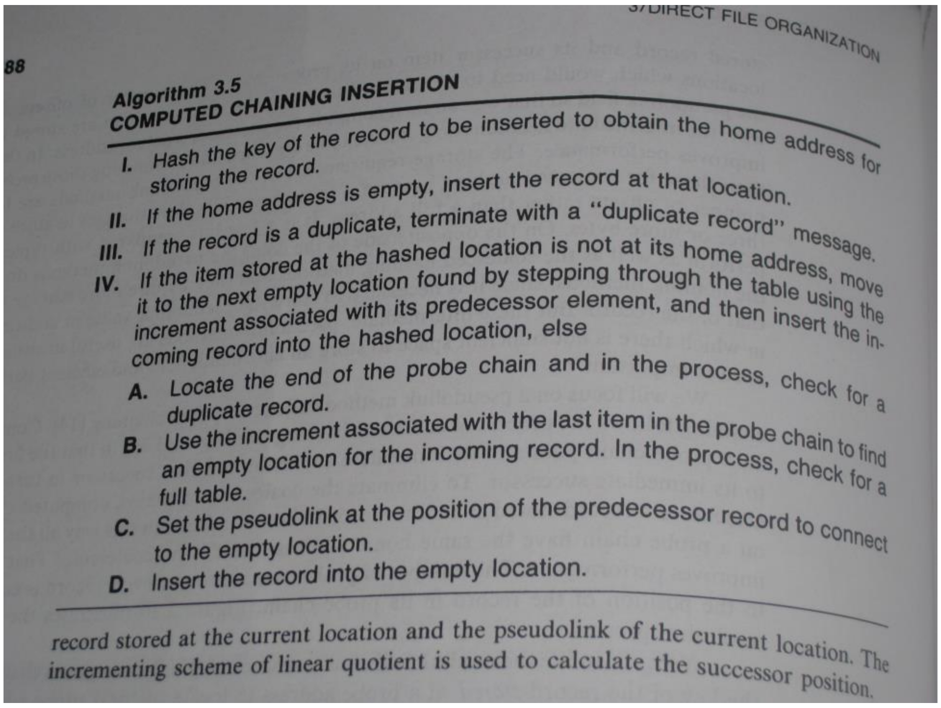


Algorithm 3.4
BINARY TREE INSERTION

I. Hash the key of the record to be inserted to obtain the home address for storing the record.

II. If the home address is empty, insert the record at that location, else

- **A.** Until an empty location or a "full table" is encountered,
  - **1.** Generate a binary tree control structure in a breadth first left to right fashion. The address of the lchild of a node is determined by adding (i) the increment associated with the key of the record *coming in* to the node to (ii) the current address. The address of the rchild of a node is determined by adding (i) the increment associated with the key of the record *stored* in the node to (ii) the current address.
  - **2.** At the leftmost node on each level, check against the record associated with it for a duplicate record. If found, terminate with a "duplicate record" message.
- **B.** If a "full table," terminate with a "full table" message.
- **C.** If an empty node is found, the path from the empty node back to the root determines which records, if any, need to be moved. Each right link signifies that a relocation is necessary. First, set the current node pointer to the last node generated in the binary tree and set the empty location pointer to the table address associated with the last node generated.
- **D.** Until the current pointer equals the root node of the binary tree (bottom up), note the type of branch from the parent of the current node to the current node.
  - **1.** On a right branch, move the record stored at the location contained in the parent node into the location indicated by the empty location pointer. Set the empty location pointer to the newly vacated position and make the parent node the current node.
  - **2.** On a left branch, make the parent node the current node.
- **E.** Insert the record coming into the root position into the empty location. Terminate with a successful insertion.

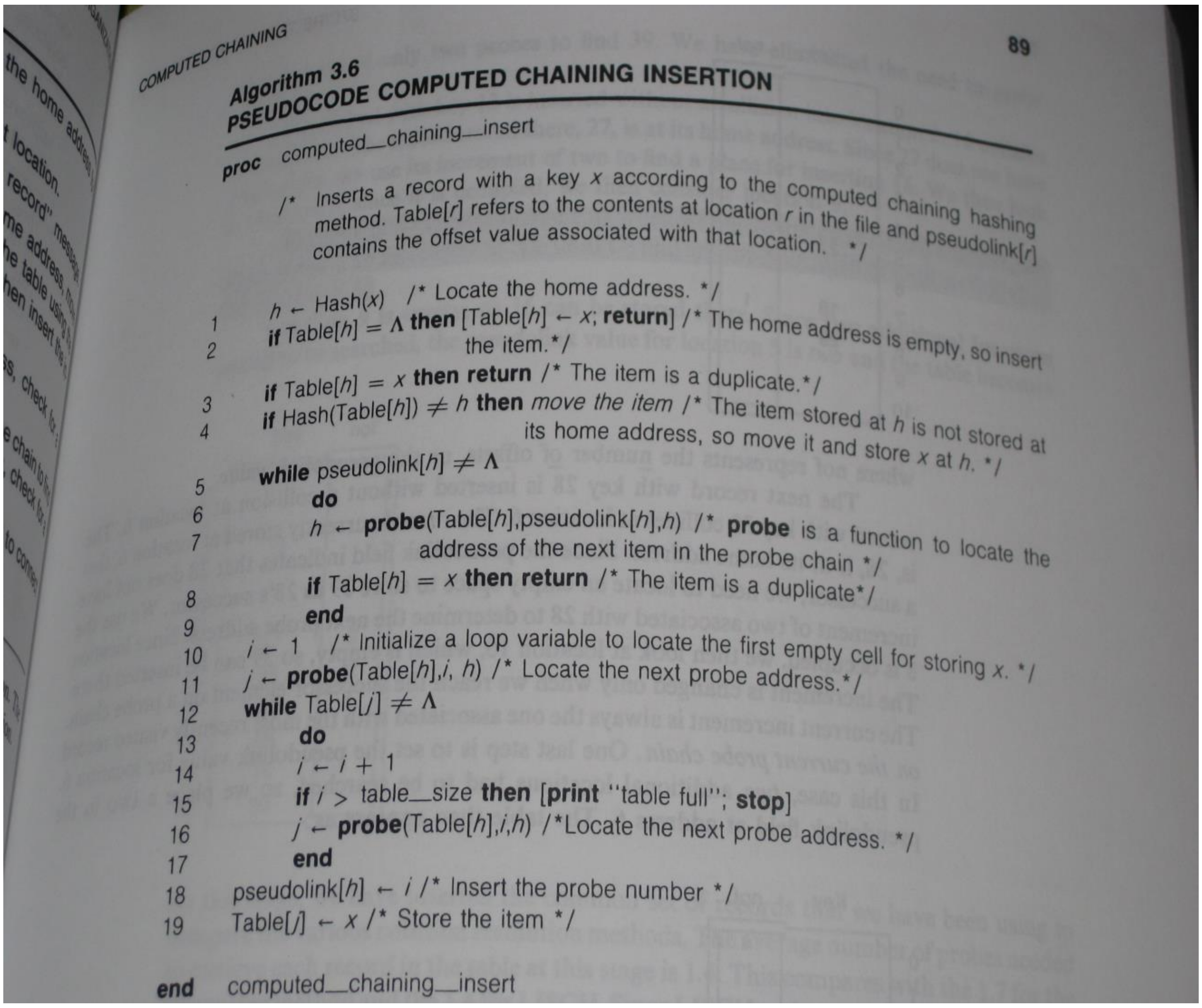


88 3/DIRECT FILE ORGANIZATION

Algorithm 3.5
COMPUTED CHAINING INSERTION

I. Hash the key of the record to be inserted to obtain the home address for storing the record.
II. If the home address is empty, insert the record at that location.
III. If the record is a duplicate, terminate with a "duplicate record" message.
IV. If the item stored at the hashed location is not at its home address, move it to the next empty location found by stepping through the table using the increment associated with its predecessor element, and then insert the incoming record into the hashed location, else
  A. Locate the end of the probe chain and in the process, check for a duplicate record.
  B. Use the increment associated with the last item in the probe chain to find an empty location for the incoming record. In the process, check for a full table.
  C. Set the pseudolink at the position of the predecessor record to connect to the empty location.
  D. Insert the record into the empty location.

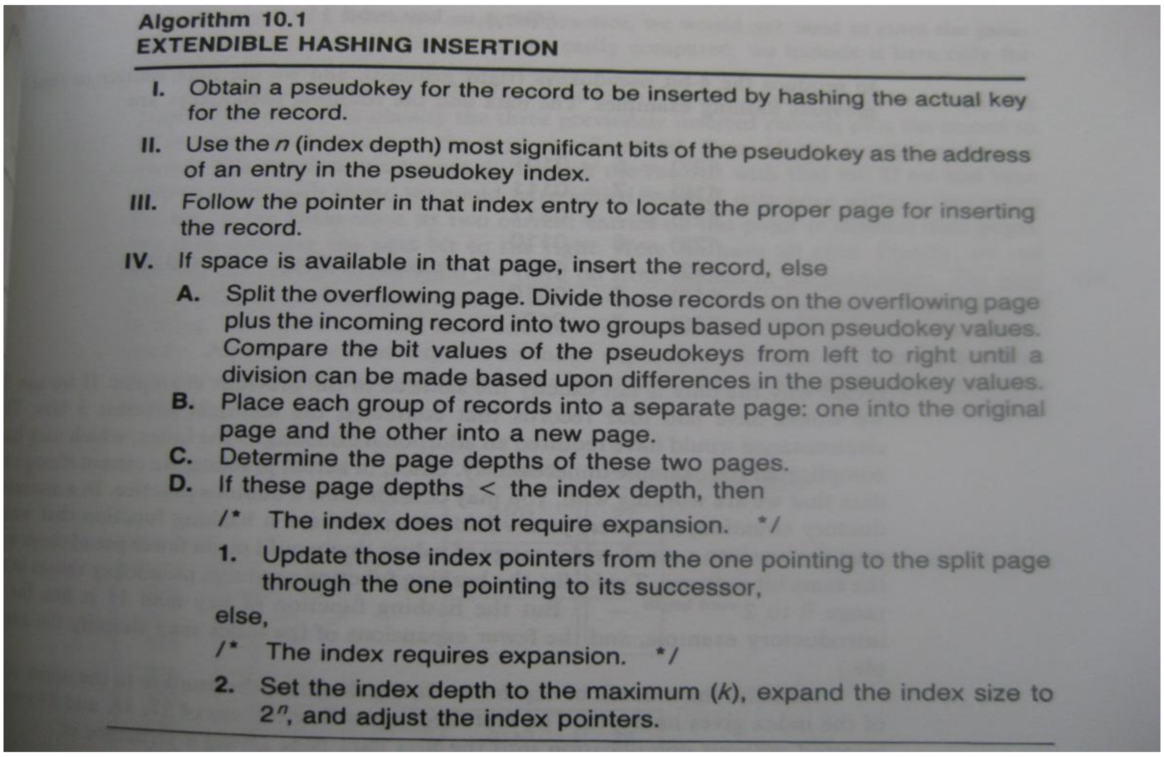
record stored at the current location and the pseudolink of the current location. The incrementing scheme of linear quotient is used to calculate the successor position.



COMPUTED CHAINING 89

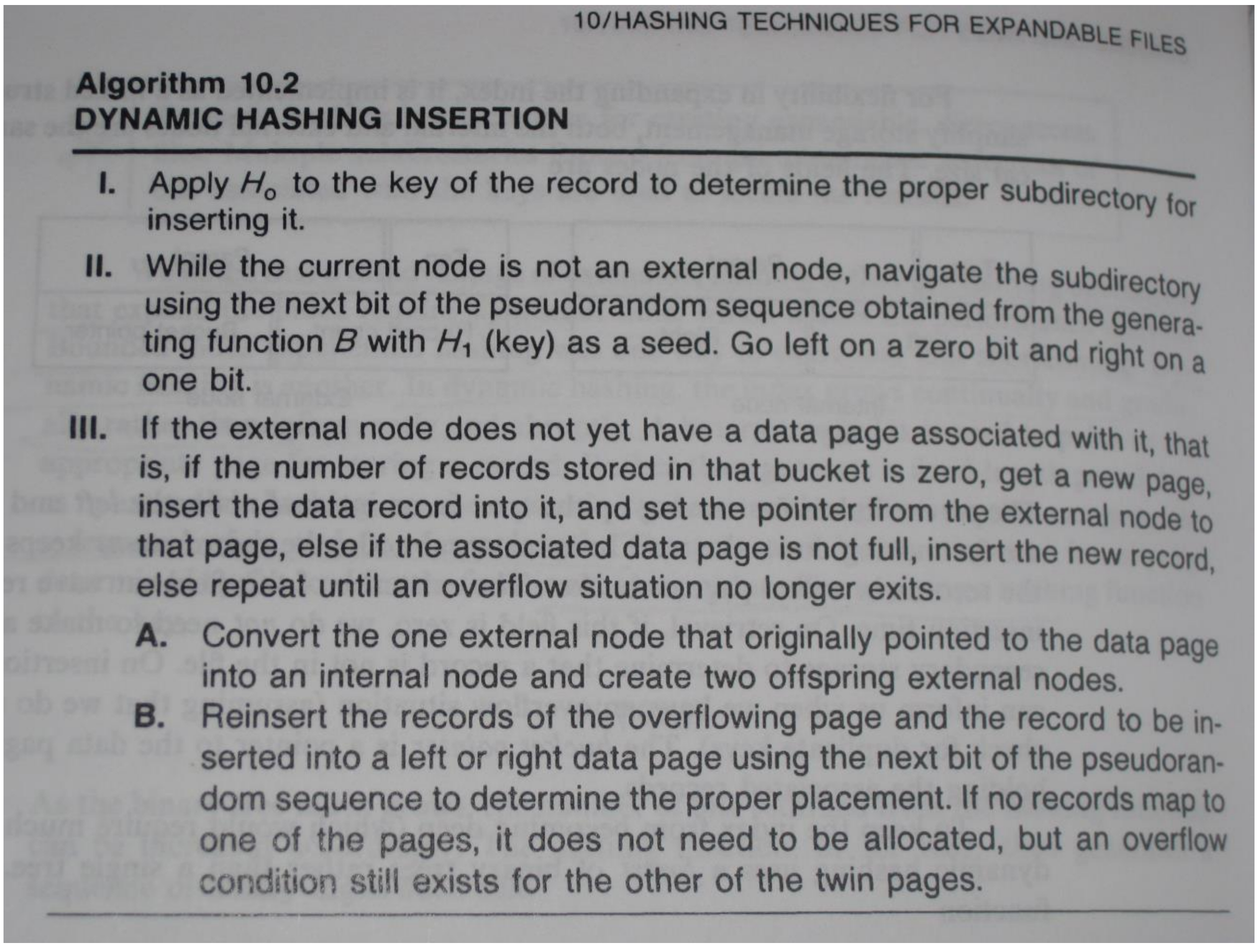
Algorithm 3.6
PSEUDOCODE COMPUTED CHAINING INSERTION

```
proc computed_chaining_insert
        /* Inserts a record with a key x according to the computed chaining hashing
        method. Table[r] refers to the contents at location r in the file and pseudolink[r]
        contains the offset value associated with that location.  */

    h ← Hash(x)   /* Locate the home address. */
    if Table[h] = Λ then [Table[h] ← x; return] /* The home address is empty, so insert
                        the item.*/
    if Table[h] = x then return /* The item is a duplicate.*/
    if Hash(Table[h]) ≠ h then move the item /* The item stored at h is not stored at
                        its home address, so move it and store x at h. */
    while pseudolink[h] ≠ Λ
        do
        h ← probe(Table[h],pseudolink[h],h) /* probe is a function to locate the
                        address of the next item in the probe chain */
        if Table[h] = x then return /* The item is a duplicate*/
        end
    i ← 1   /* Initialize a loop variable to locate the first empty cell for storing x. */
    j ← probe(Table[h],i, h) /* Locate the next probe address.*/
    while Table[j] ≠ Λ
        do
        i ← i + 1
        if i > table_size then [print "table full"; stop]
        j ← probe(Table[h],i,h) /*Locate the next probe address. */
        end
    pseudolink[h] ← i /* Insert the probe number */
    Table[j] ← x /* Store the item */

end computed_chaining_insert
```



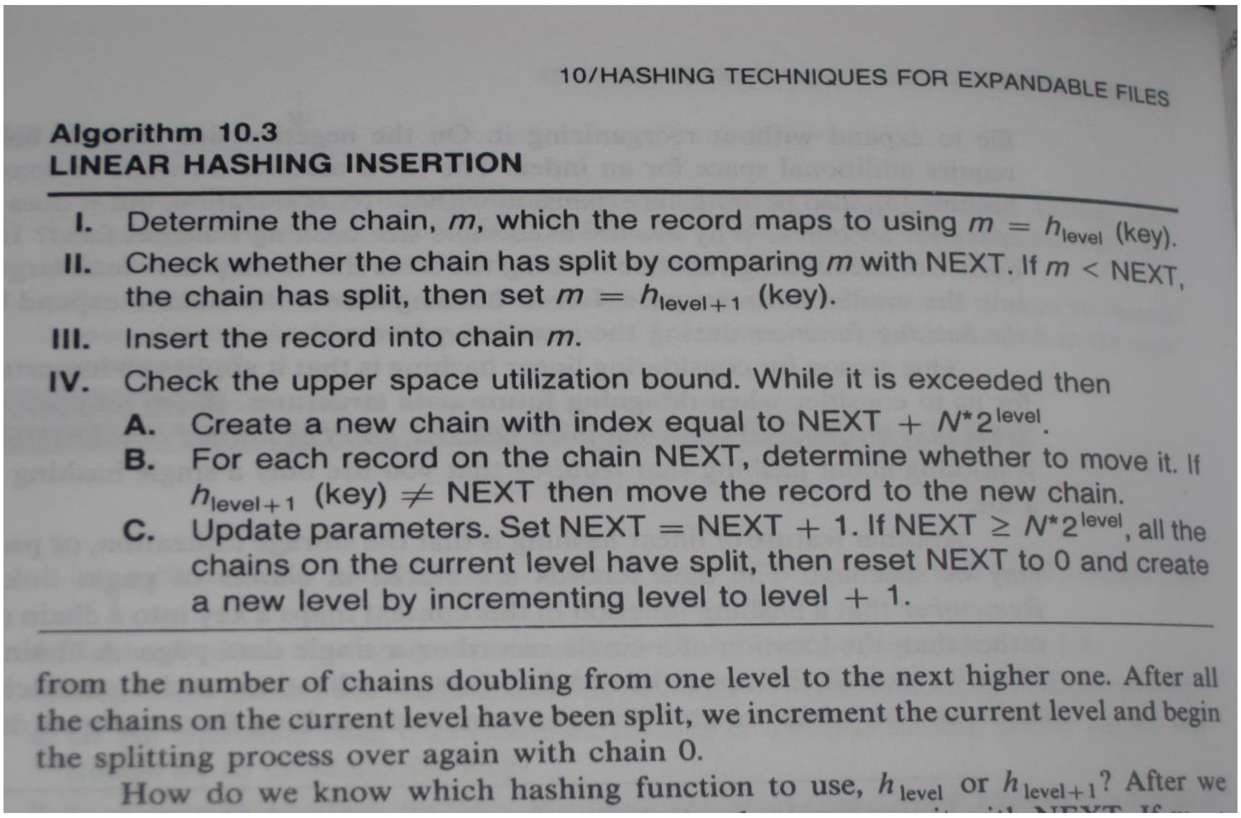
Algorithm 10.1
EXTENDIBLE HASHING INSERTION

```
I.   Obtain a pseudokey for the record to be inserted by hashing the actual key
     for the record.
II.  Use the $n$ (index depth) most significant bits of the pseudokey as the address
     of an entry in the pseudokey index.
III. Follow the pointer in that index entry to locate the proper page for inserting
     the record.
IV.  If space is available in that page, insert the record, else
     A. Split the overflowing page. Divide those records on the overflowing page
        plus the incoming record into two groups based upon pseudokey values.
        Compare the bit values of the pseudokeys from left to right until a
        division can be made based upon differences in the pseudokey values.
     B. Place each group of records into a separate page: one into the original
        page and the other into a new page.
     C. Determine the page depths of these two pages.
     D. If these page depths $<$ the index depth, then
        /* The index does not require expansion. */
        1. Update those index pointers from the one pointing to the split page
           through the one pointing to its successor,
        else,
        /* The index requires expansion. */
        2. Set the index depth to the maximum ($k$), expand the index size to
           $2^n$, and adjust the index pointers.
```



Algorithm 10.2
DYNAMIC HASHING INSERTION

```
I.   Apply $H_0$ to the key of the record to determine the proper subdirectory for
     inserting it.
II.  While the current node is not an external node, navigate the subdirectory
     using the next bit of the pseudorandom sequence obtained from the genera-
     ting function $B$ with $H_1$ (key) as a seed. Go left on a zero bit and right on a
     one bit.
III. If the external node does not yet have a data page associated with it, that
     is, if the number of records stored in that bucket is zero, get a new page,
     insert the data record into it, and set the pointer from the external node to
     that page, else if the associated data page is not full, insert the new record,
     else repeat until an overflow situation no longer exits.
     A. Convert the one external node that originally pointed to the data page
        into an internal node and create two offspring external nodes.
     B. Reinsert the records of the overflowing page and the record to be in-
        serted into a left or right data page using the next bit of the pseudoran-
        dom sequence to determine the proper placement. If no records map to
        one of the pages, it does not need to be allocated, but an overflow
        condition still exists for the other of the twin pages.
```

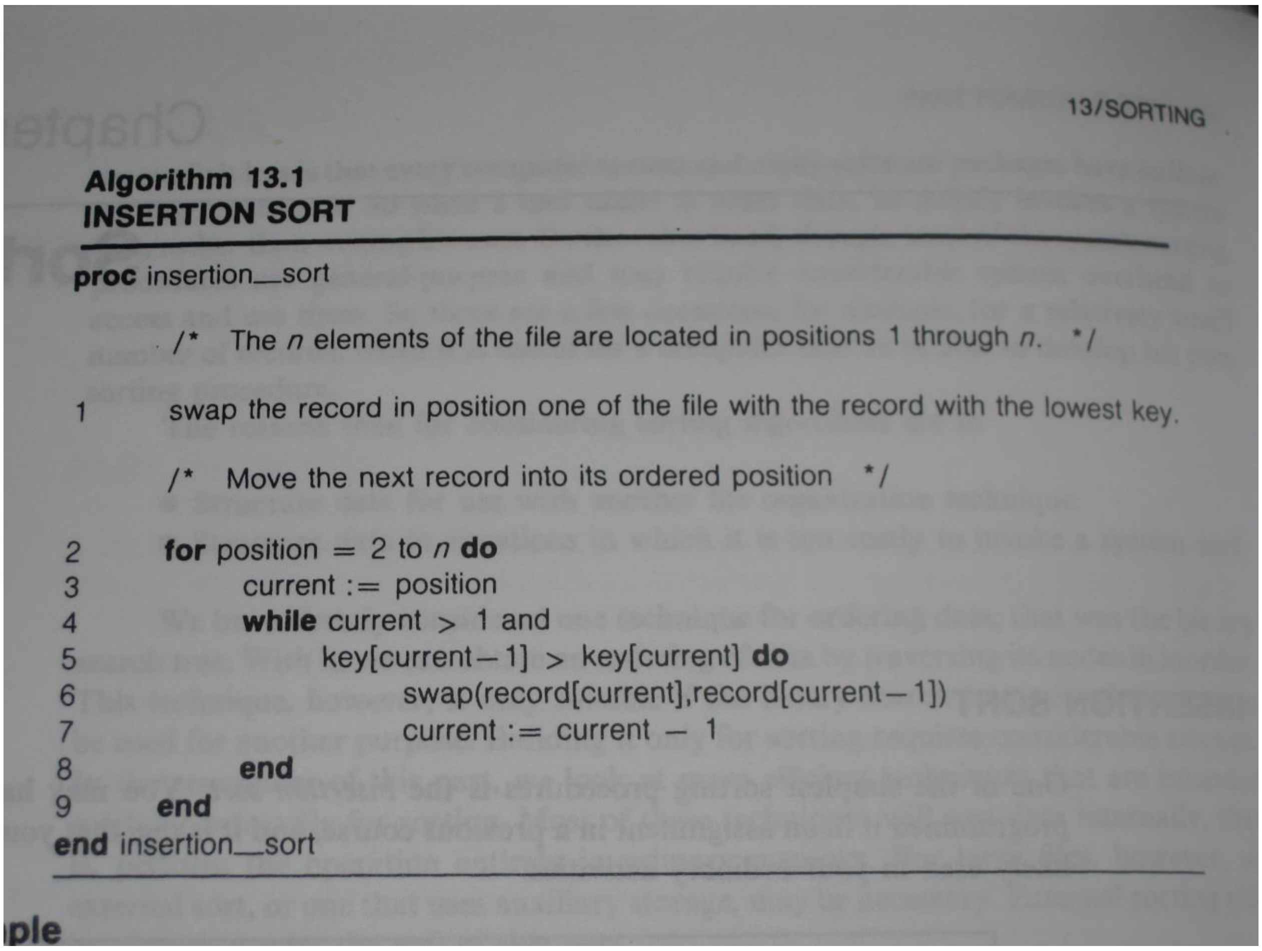


Algorithm 10.3
LINEAR HASHING INSERTION

```
I.   Determine the chain, $m$, which the record maps to using $m = h_{level}$ (key).
II.  Check whether the chain has split by comparing $m$ with NEXT. If $m <$ NEXT,
     the chain has split, then set $m = h_{level+1}$ (key).
III. Insert the record into chain $m$.
IV.  Check the upper space utilization bound. While it is exceeded then
     A. Create a new chain with index equal to NEXT + $N^*2^{level}$.
     B. For each record on the chain NEXT, determine whether to move it. If
        $h_{level+1}$ (key) $\neq$ NEXT then move the record to the new chain.
     C. Update parameters. Set NEXT = NEXT + 1. If NEXT $\geq N^*2^{level}$, all the
        chains on the current level have split, then reset NEXT to 0 and create
        a new level by incrementing level to level + 1.
```

from the number of chains doubling from one level to the next higher one. After all the chains on the current level have been split, we increment the current level and begin the splitting process over again with chain 0.

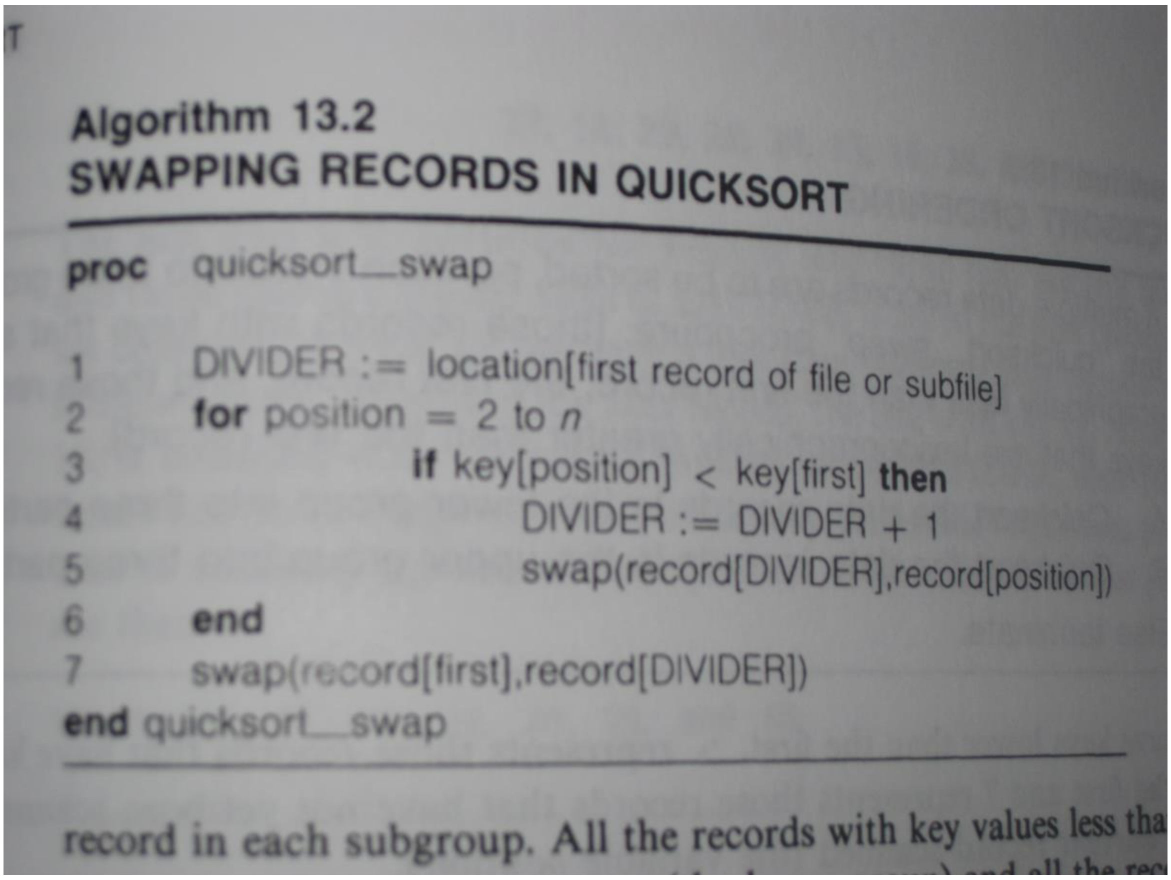
How do we know which hashing function to use, h_{level} or $h_{level+1}$? After we



Algorithm 13.1
INSERTION SORT

```
proc insertion_sort

        /*  The $n$ elements of the file are located in positions 1 through $n$.  */

1       swap the record in position one of the file with the record with the lowest key.

        /*  Move the next record into its ordered position  */

2       for position = 2 to $n$ do
3           current := position
4           while current > 1 and
5                 key[current-1] > key[current] do
6                     swap(record[current],record[current-1])
7                     current := current - 1
8           end
9       end
end insertion_sort
```

ple

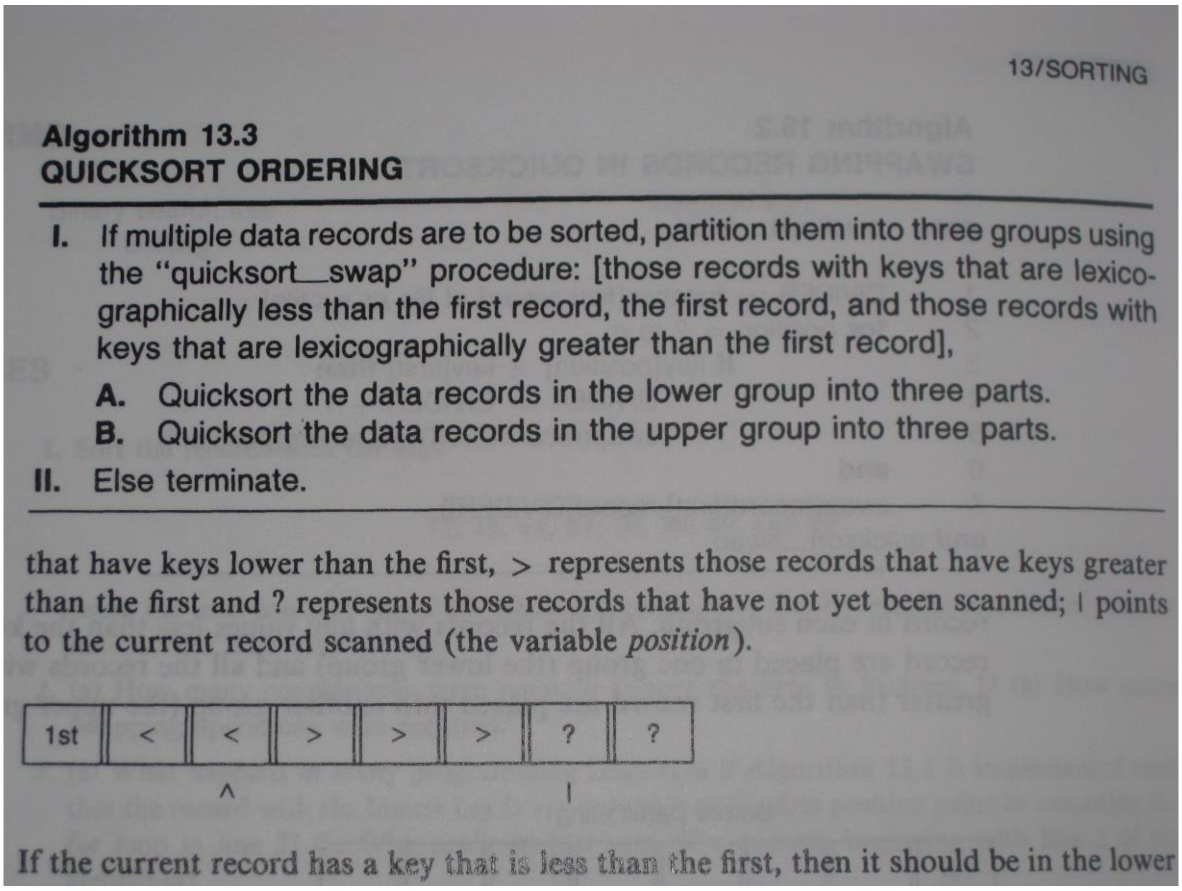


T

Algorithm 13.2
SWAPPING RECORDS IN QUICKSORT

```
proc quicksort_swap

1     DIVIDER := location[first record of file or subfile]
2     for position = 2 to n
3              if key[position] < key[first] then
4                       DIVIDER := DIVIDER + 1
5                       swap(record[DIVIDER],record[position])
6     end
7     swap(record[first],record[DIVIDER])
end quicksort_swap
```

record in each subgroup. All the records with key values less tha



Algorithm 13.3
QUICKSORT ORDERING

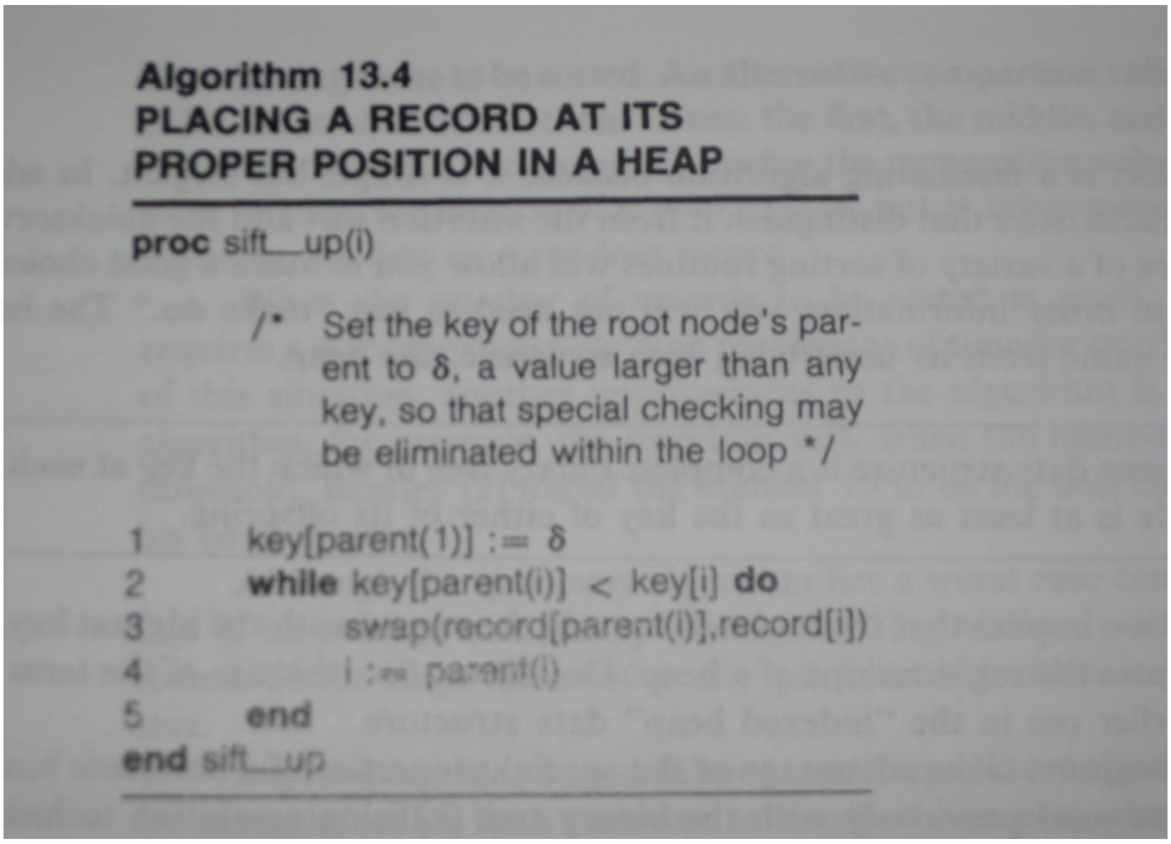
I. If multiple data records are to be sorted, partition them into three groups using the "quicksort_swap" procedure: [those records with keys that are lexicographically less than the first record, the first record, and those records with keys that are lexicographically greater than the first record],
  - **A.** Quicksort the data records in the lower group into three parts.
  - **B.** Quicksort the data records in the upper group into three parts.

II. Else terminate.

that have keys lower than the first, > represents those records that have keys greater than the first and ? represents those records that have not yet been scanned; ı points to the current record scanned (the variable *position*).

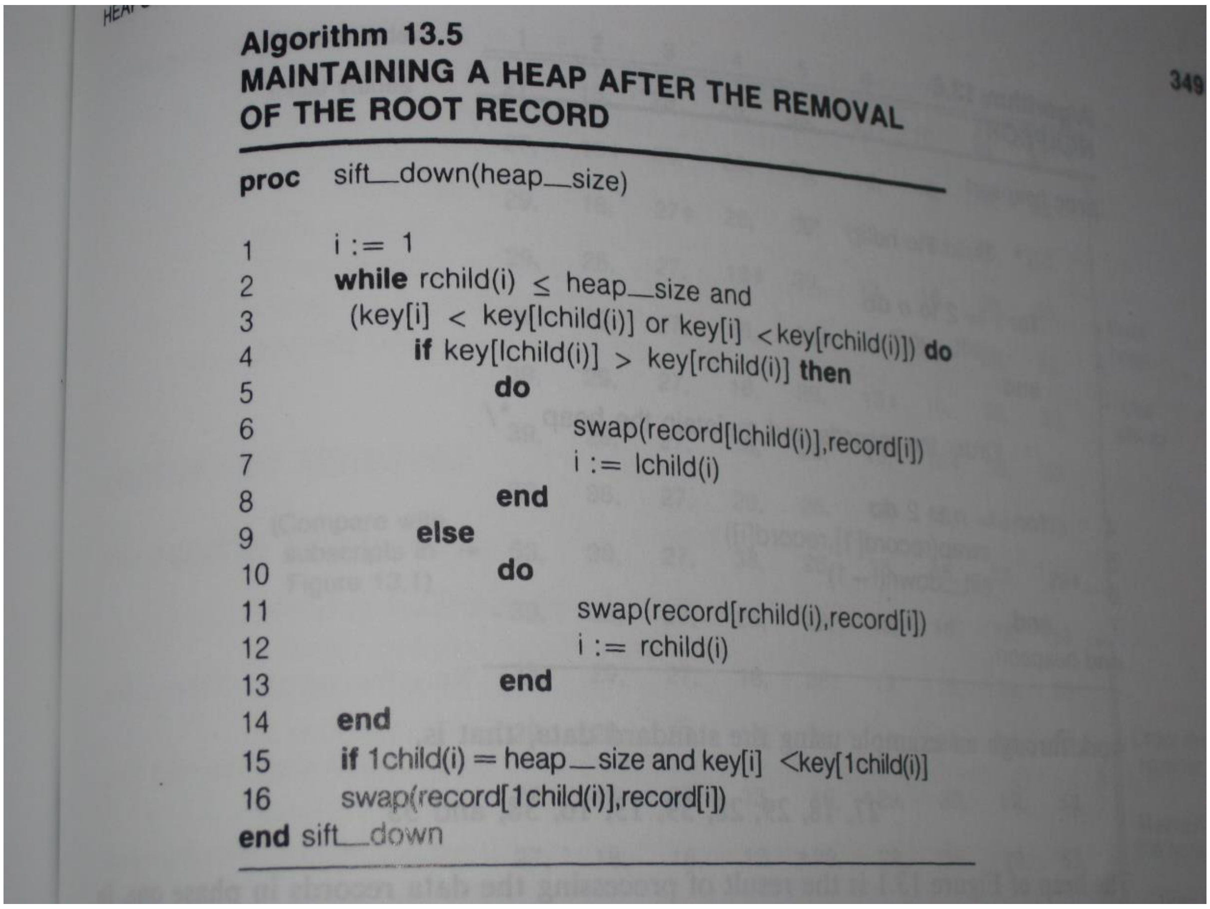
1st	<	<	>	>	>	?	?
		^				ı	

If the current record has a key that is less than the first, then it should be in the lower



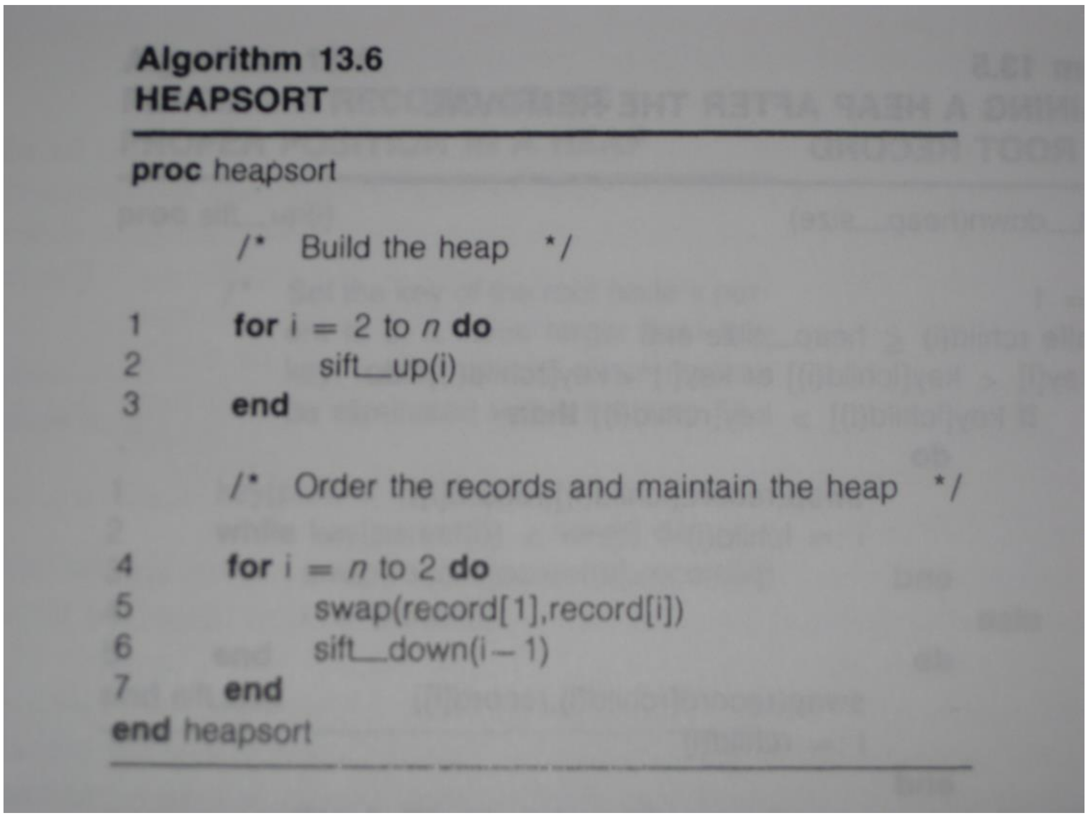
Algorithm 13.4
PLACING A RECORD AT ITS PROPER POSITION IN A HEAP

```
proc sift_up(i)

        /* Set the key of the root node's par-
           ent to $\delta$, a value larger than any
           key, so that special checking may
           be eliminated within the loop */

1       key[parent(1)] := $\delta$
2       while key[parent(i)] < key[i] do
3           swap(record[parent(i)],record[i])
4           i := parent(i)
5       end
end sift_up
```



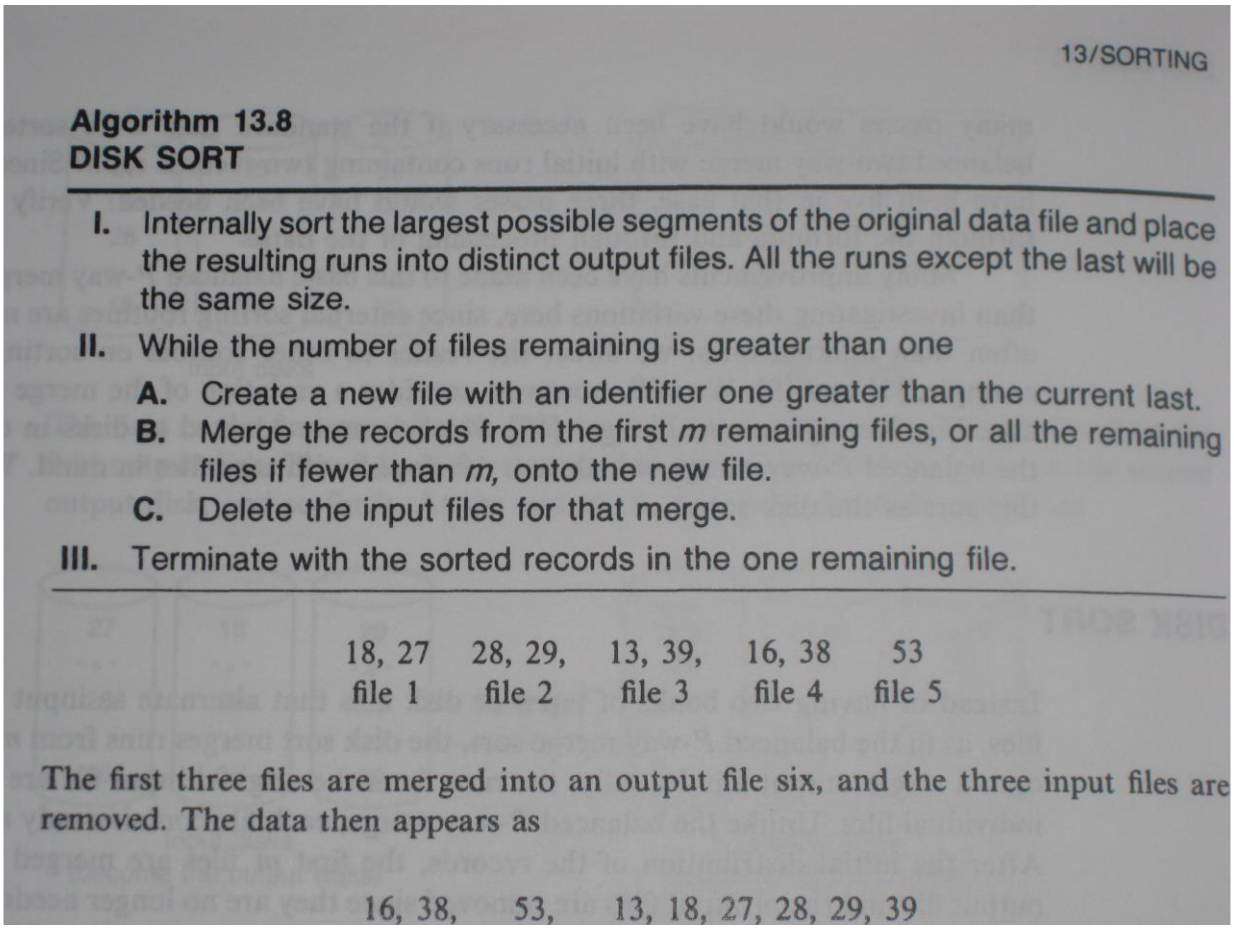
Algorithm 13.5
MAINTAINING A HEAP AFTER THE REMOVAL OF THE ROOT RECORD

```
proc    sift_down(heap_size)

1       i := 1
2       while rchild(i) ≤ heap_size and
3        (key[i] < key[lchild(i)] or key[i] <key[rchild(i)]) do
4              if key[lchild(i)] > key[rchild(i)] then
5                     do
6                            swap(record[lchild(i)],record[i])
7                            i := lchild(i)
8                     end
9              else
10                    do
11                           swap(record[rchild(i),record[i])
12                           i := rchild(i)
13                    end
14      end
15      if 1child(i) = heap_size and key[i] <key[1child(i)]
16      swap(record[1child(i)],record[i])
end sift_down
```



Algorithm 13.6
HEAPSORT

```
proc heapsort

        /*  Build the heap  */

1       for i = 2 to n do
2           sift_up(i)
3       end

        /*  Order the records and maintain the heap  */

4       for i = n to 2 do
5           swap(record[1],record[i])
6           sift_down(i-1)
7       end
end heapsort
```



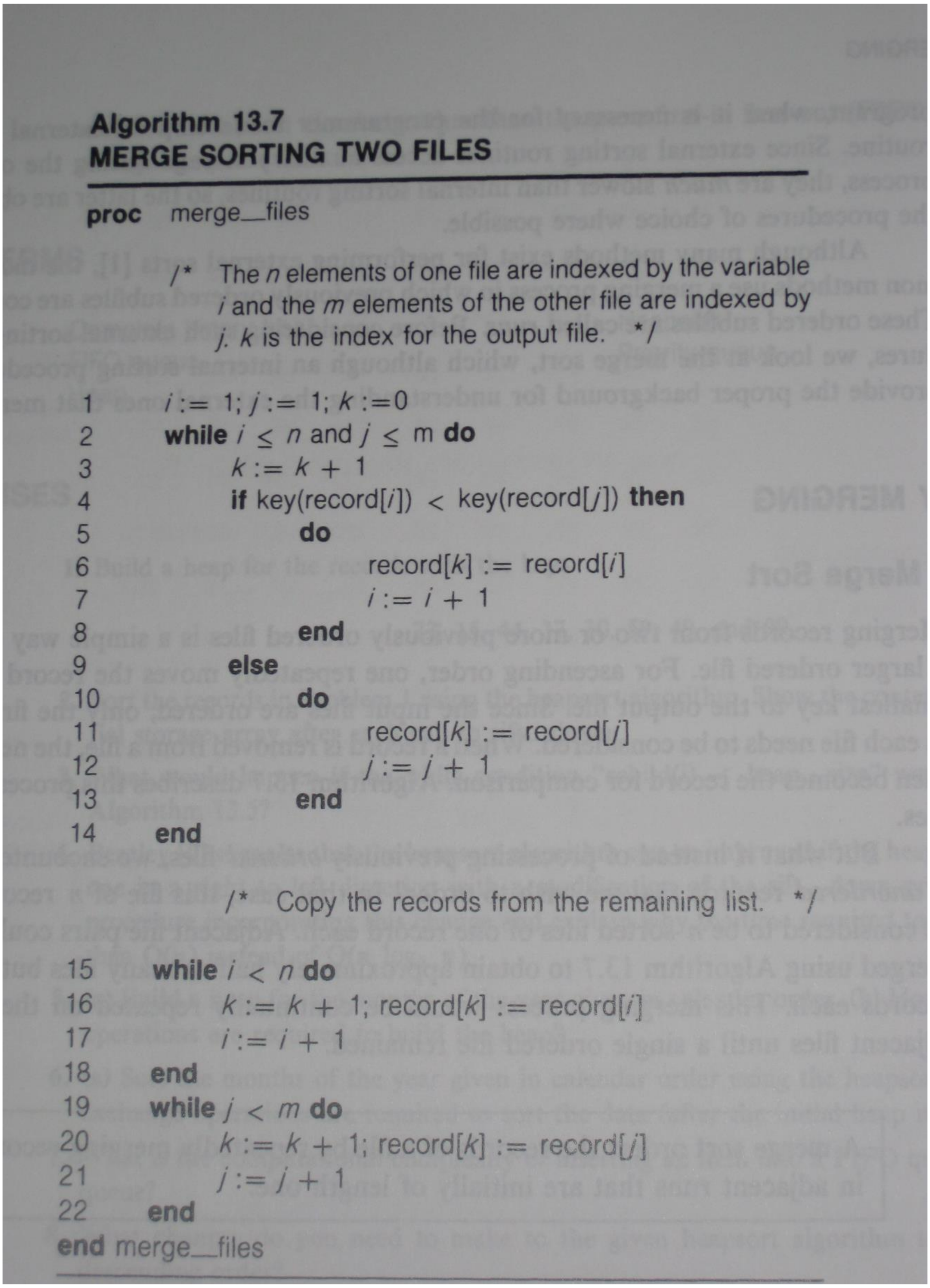
Algorithm 13.8
DISK SORT

- **I.** Internally sort the largest possible segments of the original data file and place the resulting runs into distinct output files. All the runs except the last will be the same size.
- **II.** While the number of files remaining is greater than one
  - **A.** Create a new file with an identifier one greater than the current last.
  - **B.** Merge the records from the first *m* remaining files, or all the remaining files if fewer than *m,* onto the new file.
  - **C.** Delete the input files for that merge.
- **III.** Terminate with the sorted records in the one remaining file.

18, 27	28, 29,	13, 39,	16, 38	53
file 1	file 2	file 3	file 4	file 5

The first three files are merged into an output file six, and the three input files are removed. The data then appears as

16, 38,   53,   13, 18, 27, 28, 29, 39



Algorithm 13.7
MERGE SORTING TWO FILES

```
proc merge_files

        /*  The n elements of one file are indexed by the variable
            i and the m elements of the other file are indexed by
            j. k is the index for the output file.   */

1     i := 1; j := 1; k := 0
2     while i ≤ n and j ≤ m do
3         k := k + 1
4         if key(record[i]) < key(record[j]) then
5             do
6                 record[k] := record[i]
7                 i := i + 1
8             end
9         else
10            do
11                record[k] := record[j]
12                j := j + 1
13            end
14    end

            /*  Copy the records from the remaining list.   */

15    while i < n do
16        k := k + 1; record[k] := record[i]
17        i := i + 1
18    end
19    while j < m do
20        k := k + 1; record[k] := record[j]
21        j := j + 1
22    end
end merge_files
```