STABLE SORTING AND MERGING WITH OPTIMAL SPACE AND TIME BOUNDS

by

Luis Trabb Pardo

STAN-CS-74-470
DECEMBER 1974

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Stable Sorting and Merging
With Optimal Space and Time Bounds

Luis Trabb Pardo

Abstract

This work introduces two algorithms for stable merging and stable sorting of files.

The algorithms have optimal worst case time bounds, the merge is linear and the sort is of order $n \log n$. Extra storage requirements are also optimal, since both algorithms make use of a fixed number of pointers. Files are handled only by means of the primitives exchange and comparison of records and basic pointer transformations.

This research was supported by the National Science Foundation grant GJ-36473X. Reproduction in whole or in part is permitted for any purpose of the United States Government.

1. Introduction

An algorithm which rearranges a file is said to be stable if it keeps records with equal keys in their initial relative order. This work presents an algorithm for merging two contiguous files in a stable manner (the PARTITION MERGE). As an immediate application of this, a stable algorithm to sort a file (the PARTITION MERGE SORT) is given.

The algorithms attain optimal worst case bounds with respect to time, the merge is of order n and the sort is of order $n \log n$. Both algorithms require only a fixed number of pointers for auxiliary storage. Furthermore, the algorithms are completely general, in the sense that they treat files as sequences of unmodifiable records, with the keys evaluated from the record contents and not necessarily stored within them.

While D. E. Knuth was preparing his book about sorting techniques, he noted that the known algorithms for stable sorting either were of order n^2 or they used approximately n pointers for additional memory space. Therefore he asked ([Knuth], Section 5.5, exercise 3) whether it was possible to do stable sorting in less time than order n^2, using at most $O(\log n)$ pointers for additional storage. The first progress on this problem was made by R. B. K. Dewar ([Dewar]), who developed a stable sorting algorithm of order $n^{1.5}$, using $O(\log n)$ pointers. Further improvements in the running time were made by V. Pratt ([Pratt]), F. Preparata ([Preparata]), R. Rivest ([Rivest]), and A. Nijenhuis ([Nijenhuis]). E. C. Horvath ([Horvath]) constructed stable merging and sorting algorithms with optimal time and space bounds; however, his algorithms involve the operation of key modification, thus they apply only to files in which the key is explicitly stored within the record.

The algorithms in the present paper make use of a minimum set of primitive operations on files (exchange and comparison) and in this sense appear to offer the final solution to Knuth's problem, except of course for questions dealing with the optimum constants of proportionality in the time and space bounds.

This paper is self-contained; Section 2 introduces the notation and a set of transformations of files upon which the main algorithms are built. In Section 3 the merging strategy is presented, while Section 4 deals with means to keep storage requirements low enough. With all the background of the previous sections, Sections 5 and 6 finally describe the PARTITION MERGE and PARTITION MERGE SORT in full detail, together with their respective analyses.

2. Basic Concepts

This section presents the notation used throughout the paper and describes a set of elementary operations on files that will be used for further definitions of more complex transformations.

2.1 Notation

A record R is a unit of information; its contents cannot be altered.

The key k of a given record R results from the evaluation of a certain function K applied to R

$$k = K(R) \quad .$$

A file $\mathcal{F}$ is a sequence of records

$$\mathcal{F} = \langle R_1, R_2, \ldots, R_i, \ldots, R_n \rangle \quad .$$

Each position in a file has associated with it a pointer value, an integer in the range $[1,n]$.

If i and j are pointers, two primitive operations (and only these) may be used to access the file:

-- an exchange primitive, denoted by exchange(i,j) . An application of exchange(i,j) or of exchange(j,i) transforms $\mathcal{F}$ into

$$\mathcal{F} = \langle R_1, \ldots, R_{i-1}, R_j, R_{i+1}, \ldots, R_{j-1}, R_i, R_{j+1}, \ldots, R_n \rangle \quad ;$$

-- a comparison primitive, denoted by $F(i) \leq F(j)$, whose value is true if and only if $K(R_i) \leq K(R_j)$. Since the other relations $<$, $=$, $\neq$, $\geq$, $>$ can be easily expressed in terms of one or two $\leq$'s they will be used in the definition of algorithms, as a shorthand for the corresponding relation expressed in terms of the $\leq$ primitive.

A block U (of length p) is a subsequence of p consecutive elements of $\mathcal{F}$

$$U = \langle R_m, R_{m+1}, \ldots, R_{m+p-1} \rangle \quad .$$

The length of U will be denoted by $|U|$; thus in the above case $|U| = p$. The block U will be also identified by the pointers to its first and last elements and denoted by $F[m\colon m+p-1]$. The first and last records of U will be $\text{first}(U) = R_m$ and $\text{last}(U) = R_{m+p-1}$. The term prefix (suffix) of U will refer to an initial (final) sequence of contiguous records of the block U .

The number of distinct keys in a block U will be $\lambda(U)$. Obviously $\lambda(U) \leq |U|$, with the case $\lambda(U) = |U|$ corresponding to a block composed of records with distinct keys.

A segment X is a sequence of contiguous blocks X_i

$$X = X_1 X_2 \ldots X_\ell \ldots X_q \quad .$$

A segment will be also regarded as a block with the notations $|X|$, $\text{first}(X)$, $\text{last}(X)$ and $\lambda(X)$ having the previous meaning.

Normally only nondecreasing order will be considered. The predicate ordered(U) is true if and only if the block U is ordered in nondecreasing order.

A stable transformation is a permutation of a file, that preserves the relative order of those records with equal keys. In particular, this work is concerned with two stable transformations:

-- the stable merge of two contiguous ordered blocks U and V denoted by merge(U,V) and

-- the stable sort of a block U (denoted by sort(U)).

In the examples a file will be represented by the actual sequence of records, with the keys explicitly written down.

Example 2.1: Let us assume that the file $\mathcal{F}$ is

```
3 2 3 1 2 5 6 4
A B C D E F G H
```

Then F(4) = D and K(F(4)) = K(D) = 1 . The pointer values range from 1 to 8 . Let U be the block F[2:5] , then

$|U| = 4$, first(U) = F(2) = B ,

last(U) = F(5) = E , $\lambda(U) = 3$ and ordered U is false.

Applying exchange(1,3) or exchange(3,1) yields

```
3 2 3 1 2 5 6 4
C B A D E F G H
```

In this file exchange(1,3) is not a stable transformation, but exchange(4,5) is.

□

Algorithms will be presented as ALGOL-like procedures. The language used will be ALGOL W with the addition of a new type pointer. Pointer values will be operated upon in a similar way as in the case of integers. The inclusion of the type pointer pretends to emphasize that its range depends only on the length of the common file. Thus if the latter consists of n records, only $\lceil \log n \rceil$ bits will be needed to store a pointer value.

For convenience in exposition the algorithms will be written in terms of the operations ' $p+q$ ' , ' $p-q$ ' , ' $p \times q$ ' , ' floor(p/q) ' ,

' ceiling(sqrt(p)) ' and arbitrary comparisons between pointers, but it will be clear that the optimal time and space bounds can also be achieved using only the primitive pointer operations

$$p+1 \quad , \quad p-1 \quad , \quad p+q \quad , \quad p=q \quad \text{and} \quad p := q \quad ,$$

by straightforward modifications.

2.2 Some Basic Transformations Using Minimal Extra Storage

This subsection defines in a precise manner a set of straightforward transformations of blocks and presents time bounds for each of them, so they can be used in the description of more complex algorithms within the rest of this work. None of the algorithms will be recursive, so no 'hidden' pointers are implied.

The reader is referred to Appendix A for a formal description of the algorithm and derivation of time bounds for each transformation.

In the following paragraphs U and V will denote the blocks $U = F[u_1:u_2]$ and $V = F[v_1:v_2]$.

2.2.1 Reversal of a block: REVERSE(u_1,u_2)

An application of REVERSE(u_1,u_2) transforms U into

$$U = \langle R_{u_2}, R_{u_2-1}, \dots, R_{u_1} \rangle \quad .$$

The time bounds are:

$$T_{REV}(U) = O(|U|) \quad . \qquad (2.1)$$

2.2.2 Exchange of blocks of equal length: BLOCK_EXCHANGE(u_1,u_2,v_1,v_2)

Let U and V be non-overlapping blocks of equal length $(|U| = |V|)$. Then an application of BLOCK_EXCHANGE(u_1,u_2,v_1,v_2) or BLOCK_EXCHANGE(v_1,v_2,u_1,u_2) exchanges the contents of U and V , without changing the values of u_1 , u_2 , v_1 , v_2 .

The running time is bounded by

$$T_{BEX}(U,V) = O(|U|) = O(|V|) \quad . \qquad (2.2)$$

2.2.3 Permutation of two contiguous blocks: PERMUTE(u_1,u_2,v_1,v_2)

Let U and V be two contiguous blocks, with U preceding V . That is, the common file is of the form

$$\mathcal{F} = A\,U\,V\,B \qquad \text{(where } A \text{ , } B \text{ are blocks)} \quad .$$

Applying PERMUTE(u_1,u_2,v_1,v_2) yields

$$\mathcal{F} = A\,V\,U\,B \quad ,$$

and the corresponding redefinition of u_1 , u_2 , v_1 and v_2 .
The permuting process is done by application of three successive reversals:

-- first reverse UV yielding $V^R U^R$,

-- then reverse V^R yielding VU^R ,

-- and finally reverse U^R , thus obtaining the permuted pair VU .

Since the reversals are linear so is the permute process:

$$T_{PERM}(U,V) = O(|U| + |V|) \quad . \qquad (2.3)$$

2.2.4 Stable insertion of two contiguous ordered blocks:

INSERT(u_1,u_2,v_1,v_2,f_1,f_2)

Let U and V be two contiguous ordered blocks, that is $\mathcal{F} = A\,U\,V\,B$.
Then INSERT(u_1,u_2,v_1,v_2,f_1,f_2) yields

$$\mathcal{F} = A\,V'\,U\,V''\,B \qquad \text{where } V'V'' = V$$

and

$$\text{last}(V') < \text{first}(U) \leq \text{first}(V'')$$

and sets the pointers in such a way that

$$U = F[u_1:u_2] \quad , \quad V' = F[v_1:v_2] \quad \text{and} \quad V'' = F[f_1:f_2] \quad .$$

Intuitively it can be said that the insertion of U into V moves U as forward as possible, but keeping the transformation stable.

Two basic facts, direct consequences of the above definition, can be stated as claims.

Claim 2.1: Let U , V , V' and V'' be as above, then the insertion of U into V reduces the merge of U and V to the merge of U and V'' , that is

$$\text{merge}(U,V) = V' \text{ merge}(U,V'') \quad . \qquad \square$$

Claim 2.2: Let U and V be as above, and $U = U'U''$. After inserting U'' into V , thus yielding $V' U'' V''$, the merge of U and V is reduced to

$$\text{merge}(U,V) = \text{merge}(U',V') \text{ merge}(U'',V'') \quad . \qquad \square$$

The insertion process consists of

-- a linear search over V in order to find the place where to insert U ,

followed by

-- the permutation of U and V' .

Since both steps can be accomplished in linear time, the time bounds result:

$$T_{INS}(U,V) = O(|U| + |V'|) \quad , \qquad (2.4)$$

or if desired, since $|V'| \leq |V|$

$$T_{INS}(U,V) = O(|U| + |V|) \quad . \qquad (2.5)$$

2.2.5 Direct merge of two contiguous ordered blocks:

BLOCK_MERGE_FORWARD(u_1,u_2,v_1,v_2) and

BLOCK_MERGE_BACKWARD(u_1,u_2,v_1,v_2) .

Let U and V be two contiguous ordered blocks, so $\mathcal{F} = A\,U\,V\,B$. Applying either BLOCK_MERGE_FORWARD(u_1,u_2,v_1,v_2) or

BLOCK_MERGE_BACKWARD(u_1, u_2, v_1, v_2) yields the merge of U and V, thus transforming $\mathcal{F}$ into

$$\mathcal{F} = A \text{ merge}(U,V)\, B \quad .$$

The forward merge is accomplished by an iterative process of insertions of successively smaller suffixes of U into successively smaller suffixes of V. Thus, after a stable insertion of U into V as in Section 2.2.4 yielding $V_1 U V_2$, U is partitioned U_1U_2 (where U_2 is the largest subblock with $\text{first}(U_2) > \text{first}(V_2)$), and the problem reduces to $\text{merge}(U_2, V_2)$. The backward merge is similar, but the insertions are done in a backwards direction.

The time bounds result

-- forward merge

$$T^{(\text{forw.})}_{\text{BLOCKM}}(U,V) = O(|U|\lambda(U)) + O(|V'|) \qquad (2.6)$$

where V' is that prefix of V $(V = V'V'')$ such that $\text{last}(V') < \text{last}(U) \leq \text{first}(V'')$;

-- backward merge

$$T^{(\text{back.})}_{\text{BLOCKM}}(U,V) = O(|V|\lambda(V)) + O(|U''|) \qquad (2.7)$$

where U'' is that suffix of U $(U = U'U'')$ such that $\text{last}(U') \leq \text{first}(V) < \text{first}(U'')$.

Instead of introducing the definitions of V' and U'' the block merge processes could have been bounded by the overall lengths $|V|$ and $|U|$, but these bounds pretend to emphasize the fact that the running time is only a function of the elements that are actually exchanged by the process. That is, no matter how long the suffix V'' (forward merge) or the prefix U' (backward merge) are, the running time for the merge processes doesn't change.

2.2.6 Direct stable sort of a block: STRAIGHT_INSERTION_SORT(u_1,u_2)

This process sorts the block U in a stable manner. Since it must be done with minimal extra storage, the straight insertion sort ([Knuth], Section 5.2.1) is chosen. The only extra storage needed is a fixed amount of pointers.

Time bounds result

$$T_{SORT}(U) = O(|U|^2) \quad . \tag{2.8}$$

3. The Partition Merge Strategy

This section outlines the basic strategy on which the partition merge algorithm is based, without considering either storage requirements or time bounds.

The first subsection introduces the segment insertion process, a stable transformation that is basic to the stable merge, while the second subsection analyzes the strategy itself.

3.1 The Segment Insertion Process

This stable transformation deals with two contiguous ordered blocks U and V, of length equal to a multiple of a given value f. This last condition on the length allows treating U and V as segments of blocks of length f, and thus

$$\begin{aligned} & U = U_1 \dots U_i \dots U_k \\ \text{and} \quad & V = V_1 \dots V_j \dots V_\ell \\ & \qquad \text{for some } k > 0 \text{ and } \ell > 0 , \\ & \text{with the block length} \\ & |U_i| = |V_j| = f \\ & \qquad \text{for } 1 \le i \le k \text{ and } 1 \le j \le \ell . \end{aligned} \tag{3.1}$$

Informally the segment insertion can be described as a permutation of the sequence of blocks $U_1 \dots U_k V_1 \dots V_\ell$ yielding the minimum number of inversions, but, of course, being stable.

In order to characterize such a permutation it can be argued that any block U_i in U cannot go after any block in V that could contain a record with key equal to any key of the records in U_i. Thus a block

U_i should be positioned between the contiguous blocks V_j and V_{j+1} such that

$$\text{last}(V_j) < \text{first}(U_i) \leq \text{last}(V_{j+1}) \quad . \tag{3.2}$$

(In order to make the above equation hold in every case, the fictitious blocks V_0 and $V_{\ell+1}$ must be assumed, with

$$\text{last}(V_0) = -\infty \quad \text{and} \quad \text{last}(V_{\ell+1}) = +\infty \quad .)$$

Since equation (3.2) might yield the same value of j for various consecutive blocks $U_i, U_{i+1}, \ldots, U_{i+p}$, it must also be stated that the permutation must retain the original relative ordering of blocks in U and V. So in this case the final layout will contain the segment $V_j U_i U_{i+1} \cdots U_{i+p} V_{j+1}$.

Example 3.1: As an example, let us consider U and V as below, for a block size $f = 2$:

U					V			
1 2	2 2	2 3	4 5	6 8	1 1	2 3	3 3	5 5
a b	c d	e f	g h	i j	A B	C D	E F	G H
U_1	U_2	U_3	U_4	U_5	V_1	V_2	V_3	V_4

Applying equation (3.2) to U_1 we see that

$$\text{last}(V_0) = -\infty < \text{first}(U_1) = 1 \leq \text{last}(V_1) = 1 \quad .$$

Thus U_1 will go before V_1 . For the blocks U_2 and U_3 ,

$$\text{last}(V_1) = 1 < \text{first}(U_2) = 2 \leq \text{last}(V_2) = 3$$

$$\text{and} \qquad \text{last}(V_1) < \text{first}(U_3) = 2 \leq \text{last}(V_2) \quad ,$$

so U_2 and U_3 will be positioned between V_1 and V_2 , with U_2 preceding U_3 . After considering U_4 and U_5 it can be seen that the final permutation will be

$$\begin{array}{c|c|c|c|c|c|c|c|c}
1\ 2 & 1\ 1 & 2\ 2 & 2\ 3 & 2\ 3 & 3\ 3 & 4\ 5 & 5\ 5 & 6\ 8 \\
a\ b & A\ B & c\ d & e\ f & C\ D & E\ F & g\ h & G\ H & i\ j \\
U_1 & V_1 & U_2 & U_3 & V_2 & V_3 & U_4 & V_4 & U_5
\end{array}$$

□

The final result of the segment insertion can be characterized as the sequence of segments

$$Y_1 Z_1 Y_2 Z_2 \cdots Y_d Z_d \cdots Y_t Z_t \tag{3.3}$$

where $Y_1 Y_2 \cdots Y_d \cdots Y_t = U$ and $Z_1 Z_2 \cdots Z_d \cdots Z_t = V$ and all the segments Y_d and Z_d containing at least one block, with the possible exception of Y_1 and Z_t .

Renaming Y_d and Z_d as

$$\begin{aligned} Y_d &= Y'_d L_d \quad \text{with } |L_d| = f \\ \text{and} \quad Z_d &= F_d Z'_d \quad \text{with } |F_d| = f \end{aligned} \tag{3.4}$$

(that is, L_d is the last block in Y_d and F_d is the first one in Z_d), the following restrictions apply to the layout in equation (3.3)

$$\text{(i)} \quad \text{last}(Z_{d-1}) < \text{first}(Y_d) \ , \quad 1 < d \leq t \tag{3.5}$$

$$\text{and (ii)} \quad \text{first}(L_d) \leq \text{last}(F_d) \ , \quad 1 \leq d \leq t \ . \tag{3.6}$$

The characterization given by equations (3.3) to (3.6) is no more than a formal statement of the initial considerations. Thus in the example considered above,

$$\begin{array}{ll}
Y_1 = U_1 & Z_1 = V_1 \\
Y_2 = U_2U_3 & Z_2 = V_2V_3 \\
Y_3 = U_4 & Z_3 = V_4 \\
Y_4 = U_5 & Z_4 : \ \text{empty} \ .
\end{array}$$

Equations (3.5) and (3.6) state boundary relations between contiguous segments. Somehow they give us the hint that a merge of U and V could be reduced after segment inserting U and V , to a sequence of "local" merges of the pairs of segments Y_d and Z_d . That is the idea underneath the partition merge strategy and so it is the topic of the next subsection.

3.2 <u>Description of the Partition Merge Strategy</u>

Let U and V be two contiguous ordered blocks of length greater than a given value f

$$|U| > f \quad \text{and} \quad |V| > f \quad . \tag{3.7}$$

For the sake of simplicity, and only for the time being, it will be assumed that U is of length equal to a multiple of f

$$|U| = k \cdot f \qquad \text{for} \quad k \geq 1 \quad . \tag{3.8}$$

The partition merge will proceed in the following way:

- Segment insert U and the longest prefix of V of length equal to a multiple of f .
- "Finish up" the merge, by means of local merges.

So, let

$$\begin{aligned} U &= U_1 \cdots U_i \cdots U_k \\ V &= V_1 \cdots V_j \cdots V_\ell \, T_V \end{aligned} \tag{3.9}$$

$$\text{with} \quad |V_i| = |V_j| = f \quad \text{and} \quad |T_V| < f \quad .$$

The <u>segment insertion</u> of U and $V_1 \cdots V_\ell$ yields

$$Y_1 Z_1 \cdots Y_d Z_d \cdots Y_{t-1} Z_{t-1} Y_t Z_t T_V$$

with the segments Y_d and Z_d as described in equations (3.3) to (3.6) of the previous subsection.

In order to analyze the finish up process we shall first consider the rightmost portion of the file, in particular the situation at the boundary of Y_t and Z_t . It is assumed that Z_t is not empty. The case Z_t empty will be quite similar.

By comparing $\text{last}(Y_t)$ with $\text{first}(Z_t)$ two cases may arise:

(i) If $\text{last}(Y_t) \leq \text{first}(Z_t)$ then the segment $Y_t Z_t T_v$ is already in order and, what is more important, in its final position within the merged file. This last statement is a direct consequence of the segment insertion definition, since by equation (3.5)

$$\text{last}(Z_{t-1}) < \text{first}(Y_t) \tag{3.10}$$

and so all records of $Z_1 \ldots Z_{t-1}$ must precede $\text{first}(Y_t)$. But also $\text{last}(Y_{t-1}) \leq \text{first}(Y_t)$ because U was originally in order. Thus, all the elements to the left of $\text{first}(Y_t)$ must precede it, so the above statement is true. Then nothing needs to be done about this segment, and the finish up proceeds by replacing t by t-1 .

(ii) If $\text{last}(Y_t) > \text{first}(Z_t)$ it is going to be necessary to proceed with the finish-up of the segment $Y_t Z_t T_v$, as described below.

The finish up of $Y_t Z_t T_v$ will consist of three steps. In order to describe them, let us adopt the notation of the previous subsection, and for reasons that will be immediately clear, let us rename T_v as C_{t+1} . By doing so, the rightmost portion of the file can be written as

$$Z_{t-1} Y_t' L_t F_t Z_t' C_{t+1} \tag{3.11}$$

where $Y_t' L_t = Y_t$ and $F_t Z_t' = Z_t$ with $|L_t| = |F_t| = f$.

This initial disposition is depicted in Figure 3.1(a). Notice that Figure 3.1 shows the values of the keys along the vertical axis, thus displaying the relative ordering of records.

The first step in the finish up process is to stable insert L_t into F_t, thus transforming $L_t F_t$ into $F'_t L_t F''_t$, such that

$$\text{last}(F'_t) < \text{first}(L_t) \leq \text{first}(F''_t) \quad . \qquad (3.12)$$

Figure 3.1(b) shows the situation after this first step. It can be seen that all the elements in L_t and $F''_t Z'_t C_{t-1}$ are greater or equal to those towards the left of L_t. This last assertion can be formally stated as the following claim.

Claim 3.1: After step 1, $\text{first}(L_t)$ is already in its final position within the merged file, and the overall merge has been reduced to the respective merge of the records to the left and to the right of $\text{first}(L_t)$.

Proof: All the elements to the right of $\text{first}(L_t)$ are greater than or equal to it since

-- those originally in U are greater than or equal to $\text{first}(L_t)$, by the initial order of U;

-- those originally in V are greater than or equal to $\text{first}(F''_t)$, and, by (3.12), it is $\text{first}(L_t) \leq \text{first}(F''_t)$. (The block F''_t is never empty, since $\text{first}(L_t) \leq \text{last}(F_t)$, by equation (3.6), and then by equation (3.12) at least $\text{last}(F_t)$ must belong in F''_t.)

Similarly the elements to the left of $\text{first}(L_t)$ are less than or equal to it:

-- those originally in U by the initial ordering;

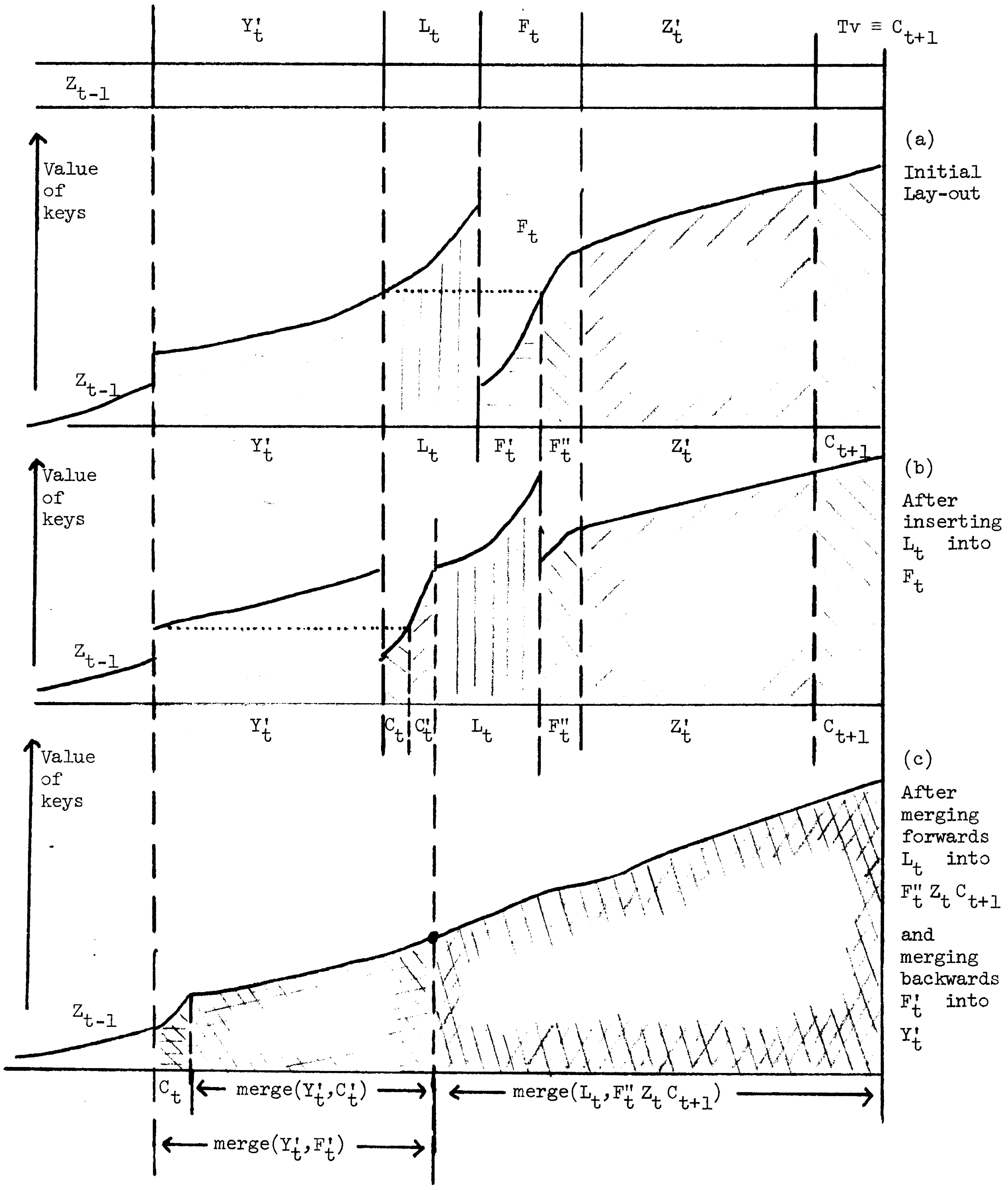


Figure 3.1: "Finish up" merges for the rightmost section of the file.

-- those originally in V are less than or equal to $\mathrm{last}(F'_t)$ and by (3.12)

$$\mathrm{last}(F'_t) < \mathrm{first}(L_t) \quad .$$

(In the case that F'_t resulted empty the first element originally in V to the left of L_t is $\mathrm{last}(Z_{t-1})$, and by equation (3.5) and the initial order of U

$$\mathrm{last}(Z_{t-1}) < \mathrm{first}(Y_t) \leq \mathrm{first}(L_t) \quad .)$$

Hence, the stability of the merge imposes that $\mathrm{first}(L_t)$ remain in its current place, since it was originally in U . And clearly the overall merge is reduced as stated in our claim. □

So, the <u>second step</u> in the finish up is the merge of L_t with $F''_t Z'_t C_{t+1}$.

Now let us consider Y'_t and F'_t , if Y'_t is nonempty. Assume that F'_t is of the form

$$F'_t = C_t C'_t \qquad \text{where} \quad \mathrm{last}(C_t) < \mathrm{first}(Y'_t) \leq \mathrm{first}(C'_t) \quad . \qquad (3.13)$$

(This partition of F'_t is identical to the one that would have been obtained by stable inserting Y'_t into F'_t .)

The third and last step in the finish up process of $Y_t Z_t C_{t+1}$ is the merge of Y'_t and F'_t . But by Claim 2.1 the merge of Y'_t and F'_t yields

$$\mathrm{merge}(Y'_t, F'_t) = C_t \ \mathrm{merge}(Y'_t, C'_t) \quad . \qquad (3.14)$$

If Y'_t is empty, the third step does not take place, and C_t is simply taken to be F'_t .

It is possible now to issue the following claim.

Claim 3.2: After step 3 all the elements to the right of C_t are already in their final position.

Proof: Only the case Y'_t nonempty needs to be considered. When Y'_t is empty the claim follows trivially from Claim 3.1.

Consider $\text{first}(Y'_t)$. By equation (3.13) and the stability of the merge it must occupy the first position in $\text{merge}(Y'_t, C'_t)$. Also by a similar reasoning as in Claim 3.1 (but applying equation (3.13) instead of (3.12)) it can be seen that it is in its final position within the merge. Clearly the rest of the elements in Y'_t and those in C'_t must be placed to the right of $\text{first}(Y'_t)$, and by Claim 3.1 to the left of $\text{first}(L_t)$. Then, all the elements in

$$\text{merge}(Y'_t, C'_t)\ \text{merge}(L_t, F''_t Z'_t C_{t+1})$$

must be in their final positions. □

The final result of the finish up of $Y_t Z_t C_{t+1}$ is shown in Figure 3.1(c).

It is left to the reader to verify that the above process is valid also in the case of empty Z_t . The only difference is that C_{t+1} plays the role of F_t , and F''_t can therefore be empty.

The overall finish up will consist of the application of the above process successively to $Y_t Z_t C_{t+1}, Y_{t-1} Z_{t-1} C_t, \ldots, Y_1 Z_1 C_2$. The proof that this process yields the merge of U and V is a straightforward induction on t , using Claim 3.2.

A remark must be made about the initial restriction on the length of U , given by equation (3.8)

$$|U| = k.f \quad .$$

The general case

$$|U| \bmod f \neq 0$$

can be reduced to the one considered here by partitioning

$$U = U'U'' \tag{3.14}$$

(with $|U'| \bmod f = 0$ and $|U''| < f$) and stable inserting U'' into V , thus yielding $U'V'U''V''$. By Claim 2.2 the overall merge is reduced to

$$\text{merge}(U,V) = \text{merge}(U',V')\ \text{merge}(U'',V'')$$

and now the partition merge strategy can be applied to merge U' and V' .

So, in the general case the partition merge strategy will be:

(a) Insert the suffix U'' into V yielding $U'V'U''V''$.

(b) Segment insert U' into V' .

(c) Finish up the merge of U' and V' : for $d = t,t-1,\ldots,1$:

  (c-1) Stable insert L_d into F_d .

  (c-2) Merge L_d and $F''_d Z'_d C_{d+1}$.

  (c-3) Merge Y'_d and F'_d

(d) Merge U'' and V'' .

To conclude it must be noticed that in all the merge processes, at least one of the blocks to be merged is of length f or less. As it will be seen later this is a key fact in order to achieve linear time bounds.

4. Keeping Storage Requirements Minimal

Any algorithm dealing with files will, at least, need to store some pointer values in order to identify records to be compared and/or exchanged.

That is why an algorithm using only a fixed amount of pointers (and, of course, the space needed to store the file) will be said to have absolute minimum extra storage requirements. Since each pointer requires $\lceil \log_2 n \rceil$ bits, the minimum requirements are $O(\log n)$ bits.

So far, no analysis has been made about extra storage needs for the actual implementation of the partition merge, and it is not obvious how to implement it using only absolute minimum extra storage.

This section introduces the concept of internal buffer, and presents the implementation of another merging technique (the BUFFER MERGE), later used as a local merge for the finish up phase, and an implementation of the segment insertion process.

4.1 The Concept of Internal Buffer

Let B be an ordered block consisting of records with distinct keys, that is

$$\text{ordered}(B) \quad \text{and} \quad \lambda(B) = |B| \quad . \tag{4.1}$$

Then B will be called an internal buffer.

Two useful characteristics of internal buffers may be singled out in advance:

-- Permutations of an internal buffer do not affect the stability of a sorting or merging process (since the internal buffer might always be sorted back in a stable manner). This property is the basis of the BUFFER MERGE technique presented in the next subsection.

-- A given permutation of $|B|$ or less elements can be "stored" in a buffer B by simply permuting its elements correspondingly. This will be the key to the implementation of the segment insertion process, appearing in Subsection 4.3.

Both properties could be used provided an internal buffer is present in the file being processed. Nevertheless, whenever a buffer is needed to process a block U it is possible to rearrange U in order to produce the desired buffer. Such a process will be called <u>buffer extraction</u>.

<u>Definition 4.1</u>: Given an ordered block U , the extraction of a buffer B of at most ℓ records transforms U into U'B , with U' and B also ordered blocks, such that

$$\text{merge}(U',B) = U \tag{4.2}$$

and B is an internal buffer

$$|B| = \lambda(B) \quad \text{and} \quad \text{ordered}(B)$$
$$\text{and} \quad |B| = \min(\ell,\lambda(U)) \quad . \tag{4.3}$$

□

That is, the buffer extraction collects at most ℓ distinct keyed records (or if the block U has only $\lambda(U) < \ell$ records with distinct keys, only $\lambda(U)$ are collected) placing them at the end of the original block; the rest of the records are compressed in U' .

In order to satisfy condition (4.2), for any sequence of records with equal keys in U , the last one is picked, so when merging U' and B , the original block U is obtained.

A similar definition could have been given for an unordered block, but it is not needed for the purposes of the present work.

The buffer extraction technique will be illustrated by means of an example. The reader interested in the actual algorithm and a more detailed analysis is referred to Appendix B.

<u>Example 4.1</u>: A buffer of length at most 5 is needed to process the following block U :

$$\begin{array}{cccccccccccccccccccc}
1&1&1&1&3&3&3&3&4&4&4&4&4&5&6&6&6&7&7&9\\
A&B&C&D&E&F&G&H&I&J&K&L&M&N&O&P&Q&R&S&T
\end{array}$$

In order to extract the buffer we start scanning from left to right until finding the last record with key 1 (that is 1D). This record will be the first in the buffer. We repeat the search, now for the last record in the sequence of those with key 3 :

$$\begin{array}{ccc|c|ccc|c|cc}
1&1&1&1&3&3&3&3&4&\\
A&B&C&D&E&F&G&H&I&\cdots
\end{array}$$

At this point we know that 3H is also going to be in the buffer. So we exchange the previously collected record with the sequence of records with key 3 , except 3H .

Proceeding in a similar manner:

$$\begin{array}{cccccc|cc|cccc|c|cc}
1&1&1&3&3&3&1&3&4&4&4&4&4&5&\\
A&B&C&E&F&G&D&H&I&J&K&L&M&N&\cdots
\end{array}$$

$$\begin{array}{cccccccccc|ccc||c|cc}
1&1&1&3&3&3&4&4&4&4&1&3&4&5&6&\\
A&B&C&E&F&G&I&J&K&L&D&H&M&N&O&\cdots
\end{array}$$

(Notice that in this case the exchange is null, since 5N is the only record with key 5)

$$\begin{array}{cccccccccc|cccc|cc|c|cc}
1&1&1&3&3&3&4&4&4&4&1&3&4&5&6&6&6&7&\\
A&B&C&E&F&G&I&J&K&L&D&H&M&N&O&P&Q&R&\cdots
\end{array}$$

$$\begin{array}{cccccccccccc|ccccc|ccc}
1&1&1&3&3&3&4&4&4&4&6&6&1&3&4&5&6&7&7&9\\
A&B&C&E&F&G&I&J&K&L&O&P&D&H&M&N&Q&R&S&7
\end{array}$$

At this point the collection is finished (we already have an internal buffer of length 5), so the collected buffer is exchanged with the rest of the file to its right, thus obtaining the final configuration

$$\begin{array}{ccccccccccccccc|ccccc} 1&1&1&3&3&3&4&4&4&4&6&6&7&7&9&1&3&4&5&6\\ A&B&C&E&F&G&I&J&K&L&O&P&R&S&T&D&H&M&N&Q\\ \hline \multicolumn{15}{c|}{U'} & \multicolumn{5}{c}{B}\end{array}$$

$\square$

To conclude the present discussion, the following facts (analyzed in Appendix B) must be pointed out:

(i) The buffer extraction technique can be applied to a fixed number of contiguous ordered blocks (in our case we shall be interested in the extraction of a buffer out of the two blocks to be merged);

(ii) The extraction process needs only a fixed amount of pointers as extra storage;

(iii) The time bounds result proportional to

-- the length of the block(s) from which the buffer is extracted;

-- the square of the length of the extracted buffer.

So, in the case of the extraction of a buffer B out of two contiguous ordered blocks U and V the time bounds are

$$T_{BE}(U,V,U',V',B,\ell) = O(|U| + |V|) + O(|B|^2) \quad . \qquad (4.4)$$

4.2 Merging Using an Internal Buffer: The BUFFER MERGE

The BUFFER_MERGE of two contiguous ordered blocks U and V requires an internal buffer B of length

$$|B| \geq \min(|U| , |V|) \qquad (4.5)$$

that is, the buffer length must be greater than or equal to the length of the shortest block to be merged.

Let us assume first that $|V| \leq |B|$. Then the buffer merge can be described as follows:

-- Exchange the contents of V with the first $|V|$ records of B ;

-- "Merge exchange" U and the first $|V|$ records in B ; the result goes in the place previously occupied by UV .

The term "merge exchange" in the above description will be clarified by the following example:

Example 4.2: The figure below shows the contiguous blocks U and V to be merged, and the buffer B :

$$\begin{array}{cccc|cc|c|ccc}
 & U & & & V & & & & B & \\
1 & 4 & 4 & 8 & 2 & 4 & & 1 & 3 & 5 \\
a & b & c & d & A & B & \cdots & \alpha & \beta & \gamma
\end{array}$$

After exchanging V and B we obtain:

$$\begin{array}{cccccc|c|ccc}
 & & U & & & & & & & \\
1 & 4 & 4 & 8 & 1 & 3 & & 2 & 4 & 5 \\
a & b & c & d & \alpha & \beta & \cdots & A & B & \gamma \\
 & & & i & & m & & & j &
\end{array}$$

where the pointers i and j point to the last non-merged element in U and V ; the pointer m points to the first "free" place in UV .

Comparing $F(i)$ and $F(j)$ we decide that $F(i)$ must be the last element of $\text{merge}(U,V)$, so we exchange it with $F(m)$,

$$\begin{array}{ccccc|c|c|ccc}
1 & 4 & 4 & 3 & 1 & 8 & & 2 & 4 & 5 \\
a & b & c & \beta & \alpha & d & \cdots & A & B & \gamma \\
 & & i & & m & & & & j &
\end{array}$$

and update the pointers i and m . (The area to the right of m is the portion of the file already merged.) Now, F(j) is equal to F(i) and so it is the next element to be exchanged:

$$\begin{array}{cccc|cc|c|ccc} 1&4&4&3&4&8&&2&1&5\\ a&b&c&\beta&B&d&\cdots&A&\alpha&\gamma\\ &&i&m&&&&j&& \end{array}$$

Similarly we obtain:

$$\begin{array}{ccc|ccc|c|ccc} 1&4&3&4&4&8&&2&1&5\\ a&b&\beta&c&B&d&\cdots&A&\alpha&\gamma\\ &i&m&&&&&j&& \end{array}$$

$$\begin{array}{cc|cccc|c|ccc} 1&3&4&4&4&8&&2&1&5\\ a&\beta&b&c&B&d&\cdots&A&\alpha&\gamma\\ i&m&&&&&&j&& \end{array}$$

$$\begin{array}{cc|ccccc|c|ccc} &1&2&4&4&4&8&&3&1&5\\ &a&A&b&c&B&d&\cdots&\beta&\alpha&\gamma\\ i,m&&&&&&&j&&& \end{array}$$

At this point all the records originally in V are in their final positions. Thus the remaining prefix of U is also in its proper place and the merge is complete. In the case that U is exhausted before V, the remaining elements of V should be exchanged with the initial position of UV . □

It is important to realize that at any point in the process, the "buffer zone" in UV (that is, the zone filled with elements originally in B) has the same length as the non-merged portion of V . In other words, if the internal buffer B was $F[b_1:b_2]$, the following relation

$$m-i = j - b_1 + 1$$

is an invariant throughout the merge.

It is possible now to formalize the previous description by means of the following procedure:

```
procedure BUFFER_MERGE_BACKWARD (pointer $u_1, u_2, v_1, v_2, b_1, b_2$);
    if $(u_1 \leq u_2) \wedge (v_1 \leq u_2)$ then
      begin comment: both files are nonempty, U is $F(u_1:u_2)$ ,
                     V is $F(v_1:v_2)$ , and $u_2 = v_1-1$ .
                     B is $F(b_1:b_2)$ and $b_2-b_1 \geq v_2-v_1$ ;
        pointer m, i, j;
      comment: exchange contents of V and B;
        BLOCK_EXCHANGE($v_1, v_2, b_1, b_1+v_2-v_1$);
      comment: merge backwards;
        i := $u_2$; j := $b_1 + v_2 - v_1$; m := $v_2$;
        while $(i \geq u_1) \wedge (j \geq b_1)$ do
          begin
            if $F(j) \geq F(i)$
              then begin exchange(j,m);
                         j := j-1
                   end
              else begin exchange(i,m);
                         i := i-1
                   end
            m := m-1;
          end;
      comment: copy remaining portion (if any) of V;
    while $j \geq b_1$ do
      begin exchange(j,m);
            j := j-1; m := m-1
      end
  end buffer-merge-backward;
```

In order to bound the running time the following facts must be considered:

(i) The exchange of V and B takes time proportional to the length of V, that is

$$T_{(i)} = O(|V|) \quad . \tag{4.6}$$

(ii) The merge backwards loop keeps exchanging elements originally in U or V until either one is exhausted. So two cases arise:

(a) V is exhausted first, hence U must be of the form

$$U = U'U'' \tag{4.7}$$

where $\text{last}(U') \leq \text{first}(V) < \text{first}(U'')$, and the exchange takes time

$$T_{(iia)} = O(|U''| + |V|) \quad . \tag{4.8}$$

(b) U is exhausted first, and then V must be

$$V = V'V'' \tag{4.9}$$

with $\text{last}(V') < \text{first}(U) \leq \text{first}(V'')$, with time bounds

$$T_{(iib)} = O(|U| + |V''|) \quad . \tag{4.10}$$

(iii) The copy of the remaining portion of V takes place only if U has been exhausted before V, and that portion happens to be V' as defined in equation (4.9). Thus, with the same cases as above

(a) V is exhausted first:

$$T_{(iiia)} = 0 \quad . \tag{4.11}$$

(b) U is exhausted first:

$$T_{(iiib)} = O(|V'|) \quad . \tag{4.12}$$

The time bounds result

(a) If V is exhausted first

$$\begin{aligned} T_a &= T_{(i)} + T_{(iia)} + T_{(iiia)} \\ &= O(|V|) + O(|U''| + |V|) + O \\ &= O(|U''|) + O(|V|) \quad . \end{aligned} \tag{4.13}$$

(b) If U is exhausted first

$$\begin{aligned} T_b &= T_{(i)} + T_{(iib)} + T_{(iiib)} \\ &= O(|V|) + O(|U| + |V''|) + O(|V'|) \\ &= O(|U|) + O(|V|) \quad . \end{aligned} \tag{4.14}$$

Comparing (4.13) and (4.14) it is possible to write a unique expression for the time bounds as

$$T^{(back)}_{BUFM}(U,V,B) = O(|U''|) + O(|V|) \tag{4.15}$$

$$\text{where} \quad U = U'U'' \quad \text{and} \quad \text{last}(U') \leq \text{first}(V) < \text{first}(U'')$$

since in the case that U is exhausted first according to (4.9) $\text{first}(V) < \text{first}(U)$, and then $U'' = U$ (with U' empty), so T_b reduces to T_a .

Equation (4.15) reiterates a point already considered when discussing the block merge (Subsection 2.2.5): The running time is bounded by the number of elements that are actually exchanged, and hence it is not dependent on the length of the prefix U' (that is, the elements that were already placed in their proper positions before the merge was carried on).

All the previous considerations and a symmetrical algorithm (BUFFER_MERGE_FORWARD) apply to the case in which $|U| \leq |B|$ and U is merged forward into V.

The time bounds result:

$$T^{(forw.)}_{BUFM}(U,V,B) = O(|U|) + O(|V'|) \qquad (4.16)$$

$$\text{where } V = V'V'' \text{ and } last(V') < last(U) \leq first(V'') .$$

4.3 Implementation of the Segment Insertion Process

This subsection describes how the segment insertion can be implemented with the aid of an internal buffer, using as extra storage only a fixed number of pointers.

Recalling the definition stated in Subsection 3.1, the two contiguous ordered segments U and V

$$\begin{aligned} U &= U_1 \dots U_i \dots U_k \\ V &= V_1 \dots V_j \dots V_\ell \end{aligned} \qquad (4.17)$$

$$\text{where } |U_i| = |V_j| = f$$

are transformed into

$$Y_1 Z_1 \dots Y_d Z_d \dots Y_t Z_t$$

where the segments Y_d and Z_d are defined by equations (3.3) to (3.6).

By considering the segments $Z_{d-1} Y_d Z_d Y_d$ as

$$\begin{aligned} Z_{d-1} &= V_{j-r} \dots V_{j-1} \\ Y_d &= U_i \dots U_{i+p} \\ Z_d &= V_j \dots V_{j+q} \\ Y_{d+1} &= U_{i+p+1} \dots U_{i+p+s} \end{aligned} \qquad (4.18)$$

equations (3.5) and (3.6) yield

$$\begin{aligned} &\text{last}(V_{j-1}) \\ &\quad < \text{first}(U_i) \leq \cdots \leq \text{first}(U_{i+p}) \\ &\quad\quad \leq \text{last}(V_j) \leq \cdots \leq \text{last}(V_{j+q}) \\ &\quad\quad\quad < \text{first}(U_{i+p+1}) \quad . \end{aligned} \tag{4.19}$$

Equation (4.19) indicates an easy method to determine the final order of the blocks. Consider sequentially U_1, U_2, etc. until reaching the smallest p with

$$\text{last}(V_1) < \text{first}(U_p) \quad .$$

Then $U_1 \ldots U_{p-1}$ are the first blocks in the final permutation. Now consider V_1, V_2, etc. until reaching the smallest q with

$$\text{first}(U_p) \leq \text{last}(V_q) \quad ,$$

thus establishing that the sequence $V_1 \ldots V_{q-1}$ will come after U_{p-1}. The process is now repeated until U and V are exhausted.

The above process gives us a method to compute the permutation that must be applied to the blocks in UV. But somehow that permutation must be stored before permuting the blocks, since its definition is based on the original ordering of the blocks. Thus the algorithm will have two phases:

-- Compute and "store" the permutation.

-- Permute the blocks.

In order to "store" the permutation, an internal buffer will be used. The key point is that the permutation as defined in (4.19) can be computed by inspecting the blocks in the exact order in which they are going to be permuted. Then it is possible to "remember" the final position of each block by exchanging one of its elements (say the first one) with the element in the buffer that corresponds to its final position (recall that a buffer is an ordered block). After that, the permuting phase becomes simply a sorting process in which each block has as its key the key of its first element. Let us consider the following example.

<u>Example 4.3</u>: Let U and V be as depicted below, with $f = 2$, and let B be a buffer:

$$\begin{array}{cc|cc|cc|cc|cc|c|ccccc}
\multicolumn{2}{c|}{U_1} & \multicolumn{2}{c|}{U_2} & \multicolumn{2}{c|}{U_3} & \multicolumn{2}{c|}{V_1} & \multicolumn{2}{c|}{V_2} & & \multicolumn{5}{c}{B} \\
1 & 2 & 2 & 3 & 3 & 3 & 1 & 1 & 2 & 3 & \cdots & 2 & 3 & 4 & 6 & 8 \\
a & b & c & d & e & f & A & B & C & D & & \alpha & \beta & \gamma & \delta & \varepsilon \\
p & & & & & & & q & & & & m & & & &
\end{array}$$

In order to compute the permutation the pointer p will point to the first element of the block U_i currently being considered, while q will point to the last record in V_j. The pointer m points to the element of B that will be exchanged.

We start by comparing $first(U_1)$ (i.e., $F(p)$) with $last(V_1)$ (i.e., $F(q)$). Since $F(p) \leq F(q)$ we decide that U_1 will be the first block in the permutation. So we mark U_1 by exchanging its first element with the first element of B, obtaining

$$\begin{array}{cc|cc|cc|cc|cc|c|ccccc}
\boxed{\begin{array}{c}2\\ \alpha\end{array}} & \begin{array}{c}2\\ b\end{array} & 2 & 3 & 3 & 3 & 1 & 1 & 2 & 3 & \cdots & 1 & 3 & 4 & 6 & 8 \\
 & & c & d & e & f & A & B & C & D & & a & \beta & \gamma & \delta & \varepsilon \\
 & & p & & & & & q & & & & & m & & &
\end{array}$$

Now since $last(V_1) < first(U_2)$ (that is, $F(q) < F(p)$) V_1 must go before U_1, and so it will be the second block in the permutation. So after marking it and updating the pointers, there results:

$$\begin{array}{|cc|cc|cc|cc|cc|c|ccccc|}
\boxed{2} & 2 & 2 & 3 & 3 & 3 & \boxed{3} & 1 & 2 & 3 & \cdots & 1 & 1 & 4 & 6 & 8 \\
\boxed{\alpha} & b & c & d & e & f & \boxed{\beta} & B & C & D & & a & A & \gamma & \delta & \varepsilon \\
\multicolumn{2}{c}{} & \multicolumn{2}{c}{p} & \multicolumn{2}{c}{} & \multicolumn{2}{c}{} & \multicolumn{2}{c}{q} & \multicolumn{1}{c}{} & \multicolumn{5}{c}{m}
\end{array}$$

After three more marking steps, all the blocks are marked, yielding:

$$\begin{array}{|cc|cc|cc|cc|cc|c|ccccc|}
\multicolumn{2}{|c|}{U_1} & \multicolumn{2}{c|}{U_2} & \multicolumn{2}{c|}{U_3} & \multicolumn{2}{c|}{V_1} & \multicolumn{2}{c|}{V_2} & & \multicolumn{5}{c|}{B} \\
\boxed{2} & 2 & \boxed{4} & 3 & \boxed{6} & 3 & \boxed{3} & 1 & \boxed{8} & 3 & \cdots & 1 & 1 & 2 & 3 & 2 \\
\boxed{\alpha} & b & \boxed{\gamma} & d & \boxed{\delta} & f & \boxed{\beta} & B & \boxed{\varepsilon} & D & & a & A & c & e & C
\end{array}$$

Notice that by inspection of the marking elements we can tell that the permutation is $U_1V_1U_2U_3V_2$.

We proceed now to permute the blocks. As said above, this permutation is simply a sort. But we must choose a sorting method that minimizes the number of exchanges, since they are block exchanges, involving f elements at a time. The "straight selection sort" ([Knuth], Section 5.2.3) is well suited for our purposes. This method looks for the minimal element and exchanges it with the one in the first position, then it does the same but only considering the remaining elements and putting this new minimal in the second position and so on.

After sorting we obtain:

$$\begin{array}{|cc|cc|cc|cc|cc|c|ccccc|}
\boxed{2} & 2 & \boxed{3} & 1 & \boxed{4} & 3 & \boxed{6} & 3 & \boxed{8} & 3 & \cdots & 1 & 1 & 2 & 3 & 2 \\
\boxed{\alpha} & b & \boxed{\beta} & B & \boxed{\gamma} & d & \boxed{\delta} & f & \boxed{\varepsilon} & D & & a & A & c & e & C
\end{array}$$

Finally, we exchange the first element of each block with the corresponding element in B , thus completing the permutation and restoring the original contents of the buffer:

$$\begin{array}{|cccccccccc|c|ccccc|}
1 & 2 & 1 & 1 & 2 & 3 & 3 & 3 & 2 & 3 & \cdots & 2 & 3 & 4 & 6 & 8 \\
a & b & A & B & c & d & e & f & C & D & & \alpha & \beta & \gamma & \delta & \varepsilon
\end{array}$$

$\square$

The following procedure formalizes the above description:

```
procedure SEGMENT_INSERT (pointer $u_1,u_2,v_1,v_2,f,b_1,b_2$);
begin
   pointer m, p, q, r;
   comment: compute the permutation marking the blocks;
      $p := u_1$; $q := v_1 + f - 1$; $m := b_1$;
   mark_U_and_V:
   while $(p < u_2) \wedge (q \leq v_2)$ do
      begin
         if $F(p) \leq F(q)$
            then begin exchange(p,m);
                       $p := p + f$
                 end
            else begin exchange($q - f + 1$, m);
                       $q := q + f$
                 end
         $m := m + 1$
      end;
   comment: mark the blocks of either U or V that haven't been
            marked already;
   mark_remaining_U's:
      while $(p < u_2)$ do
         begin exchange(p,m);
               $p := p + f$; $m := m + 1$
         end;
```

```
mark_remaining_V's:
  while $(q \le v_2)$ do
    begin exchange(q - f + 1 , m);
          q := q + f; m := m + 1
    end;
comment: permute the blocks;
permute_blocks:
  for r := $u_1$ step f until $v_2 - 2 \times f + 1$ do
    begin
      comment: find the block with minimal key;
      m := r;
      for s := r + f step f until $v_2 - f + 1$ do
        if F(s) < F(m) then m := s;
      comment: exchange blocks;
        BLOCK_EXCHANGE(r , r + f - 1 , m , m + f - 1);
    end;
comment: restore the initial key of each block;
restore_keys:
  for s := 1 step 1 until $(v_2 - u_1 + 1)/f$ do
    exchange($u_1 + (s-1) \times f$ , $b_1 - 1 + s$);
end segment_insert;
```

The following analysis establishes time bounds for the segment insertion:

Let N be the number of blocks, namely $(|U| + |V|)/f$.

(i) In order to compute the permutation (and mark the blocks), each block in U and V is compared and marked once (while loops labeled "mark_U_and_V", "mark_remaining_U's" and "mark_remaining V's"). Thus this process is linear in the number of blocks, that is

$$T_{(i)} = O(N) \quad . \tag{4.20}$$

(ii) The permutation process (loop labeled "permute_blocks") can be viewed as follows:

```
for p := 1 until N-1 do
    begin
        Search through the first keys in the p+1, p+2, ..., N-th
            blocks for the minimal one;
        Exchange the p-th block with the one with minimal first key;
    end
```

Since for each value of p the search for the minimal first key takes time $O(N-p)$ and the exchange $O(f)$, the time bounds are

$$T_{(ii)} = \Sigma_{1 \le p \le N-1} (O(N-p) + O(f))$$

$$= O(\frac{N\cdot(N-1)}{2}) + O((N-1)\cdot f) \quad . \tag{4.21}$$

(iii) Restoring keys ("restore_keys" loop) is linear on the number of blocks, so

$$T_{(iii)} = O(N) \quad . \tag{4.22}$$

The overall time bounds result:

$$\begin{aligned} T_{SEGIN}(U,V,f) &= T_{(i)} + T_{(ii)} + T_{(iii)} \\ &= O(N) + O(\frac{N\cdot(N-1)}{2}) + O((N-1)\cdot f) + O(N) \\ &= O(N^2) + O(N\cdot f) \\ &= O((|U| + |V|)^2 / f^2) + O(|U| + |V|) \quad . \end{aligned} \tag{4.23}$$

It is interesting to note that if f is of order $(|U| + |V|)^{1/2}$ or larger, the overall time bounds are linear on the length of UV .

5. The Partition Merge Algorithm

Section 3 presented the partition merge strategy. In Section 4 the necessary tools to keep storage requirements minimal were considered. With that background it is now possible to introduce the partition merge algorithm and bound its running time.

5.1 Description

The algorithm here presented closely follows the process introduced in Section 3, except for the addition of an initial buffer extraction step and, of course, a final merging step to merge back the internal buffer previously obtained. Figures 5.1, 5.2, ... illustrate the process on a particular example.

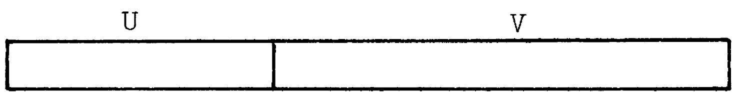
Let U and V be two contiguous ordered blocks to be merged. The following procedure defines the partition merge algorithm:

```
procedure partition_merge (pointer value u ,u ,v ,v );
begin comment: U is F[$u_1$:$u_2$] and V is F[$v_1$:$v_2$];
    pointer n, f, b, $t_1$, $t_2$, $v_3$, $v_4$, $\ell_1$, $\ell_2$, $w_1$, $w_2$, $w_3$, $w_4$, p;
    n := $v_2 - u_1 + 1$;
```

Step 1: Extract an internal buffer of length at most

$$\left\lceil \sqrt{|U| + |V|} \right\rceil .$$

```
buffer_extraction:
    BUFFER_EXTRACT2($u_1$,$u_2$,$v_1$,$v_2$,ceiling(sqrt(n)),$b_1$,$b_2$);
    b := $b_2 - b_1 + 1$; f := floor(n/b);
```

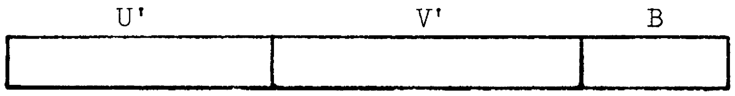


u_1 $\qquad$ u_2	v_1 $\qquad$ v_2
1 2 2 2 4 5 5 5 6 7 9	1 1 1 3 3 3 3 4 4 4 4 4 5 5 6 6 7 7 9
a b c d e f g h i j k	A B C D E F G H I J K L M N O P Q R S
$\|U\| = 11$	$\|V\| = 19$

$$n \quad |U| + |V| = 30$$

$$\left\lceil \sqrt{|U| + |V|} \right\rceil = 6$$

Figure 5.1: Initial layout.



u_1 $\qquad$ u_2	v_1 $\qquad$ v_2	b_1 $\qquad$ b_2
1 2 2 2 4 5 5 5 6 7 9	1 1 3 3 3 4 4 4 4 5 6 7 9	1 3 4 5 6 7
a b c d e f g h i j k	A B D E F H I J K M O Q S	C G L N P R
$\|U'\| = 11$	$\|V'\| = 13$	$\|B\| = 6$

$$|B| = b = 6$$

$$f = \lfloor n/b \rfloor = \lfloor 30/6 \rfloor = 5$$

Figure 5.2: After step 1.

This step transforms UV into U' V' B , where B is an internal buffer of length $b = b_2 - b_1 + 1$.

Let $f = \lfloor n/b \rfloor$.

Step 2: If either |U'| or |V'| has length less than or equal to f , then merge them directly and proceed with the final step (merging back B).

```
check_lengths:
   if $(u_2 - u_1 + 1) \le f$
   then begin
           if $(u_2 - u_1 + 1) > b$
              then BLOCK_MERGE_FORWARD $(u_1,u_2,v_1,v_2)$
              else BUFFER_MERGE_FORWARD $(u_1,u_2,v_1,v_2,b_1,b_2)$;
           go to merge_back_B;
        end
   else if $(v_2 - v_1 + 1) \le f$ then
      begin
         if $(v_2 - v_1 + 1) > b$
            then BLOCK_MERGE_BACKWARD $(u_1,u_2,v_1,v_2)$
            else BUFFER_MERGE_BACKWARD $(u_1,u_2,v_1,v_2,b_1,b_2)$;
         go to merge_back_B;
      end;
```

Notice that depending on the length of the buffer, the algorithm chooses either block_merge or buffer_merge . This choice allows linear running time as will be analyzed below.

Step 3: Prepare things for segment insertion by getting rid of that suffix T_u of U' of length

$$|T_u| = |U'| \bmod f \quad .$$

```
insert_suffix:

    $t_2 := u_2$; $u_2 := u_2 - (u_2 - u_1 + 1)$ mod $f$; $t_1 := u_2 + 1$;
    comment: $U''$ is $F[u_1:u_2]$ and $T_u$ is $F[t_1:t_2]$;
    INSERT $(t_1, t_2, v_1, v_2, v_3, v_4)$;
    comment: $V''$ is now $F[v_1:v_2]$ and $V'''$ is $F[v_3:v_4]$;
```

After the insertion $U'V'$ becomes $U''V''T_uV'''$, where $U''T_u = U'$ and $V''V''' = V'$. By the characteristics of stable insertion the merge of U' and V' is now reduced to the merge of U'' and V'' and that of T_u and V''' .

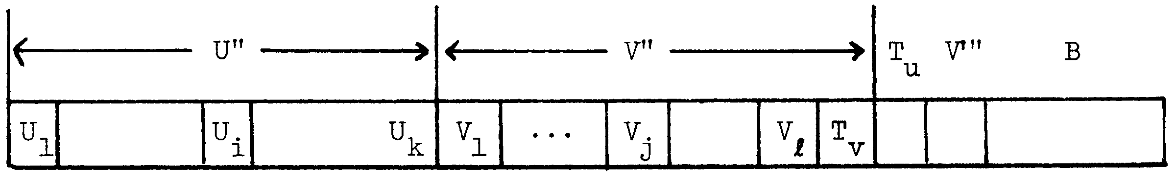
Now $|U''| \bmod f = 0$, by the choice of T_u , so U'' and V'' can be viewed as segments such that:

$$U'' = U_1 \cdots U_i \cdots U_k \qquad \text{and}$$

$$V'' = V_1 \cdots V_j \cdots V_\ell T_v$$

where $|U_i| = |V_j| = f$ for $1 \leq i \leq k$ and $1 \leq j \leq \ell$

and $|T_v| < f$.



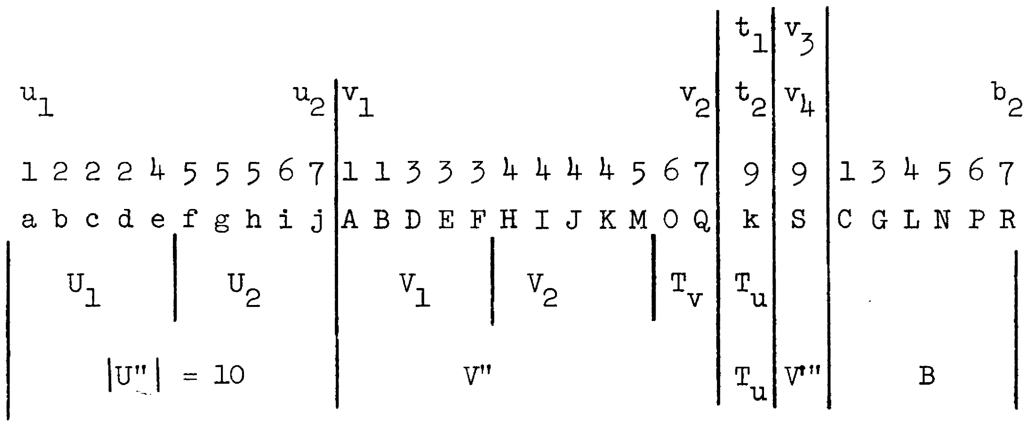

t_1 v_3
u_1 u_2 v_1 v_2 t_2 v_4 b_2
1 2 2 2 4 5 5 5 6 7 1 1 3 3 3 4 4 4 4 5 6 7 9 9 1 3 4 5 6 7
a b c d e f g h i j A B D E F H I J K M O Q k S C G L N P R
U_1 U_2 V_1 V_2 T_v T_u
|U''| = 10
V''
T_u
V'''
B

Figure 5.3: After step 3.

Step 4: Segment insert $U_1 \cdots U_i \cdots U_k$ into $V_1 \cdots V_j \cdots V_\ell$.

```
segment_insertion:
    SEGMENT_INSERT (u1,u2,v1,v2 - (v2 - v1 + 1) mod f , f,b1,b2);
```

The next step will be the finish up process (see Section 3), but some discussion is needed first.

Assume that the layout after the segment insertion is $W_1 \cdots W_m \cdots W_{k+\ell} T_v T_u V''' B$, where $W_1 \cdots W_m \cdots W_{k+\ell}$ corresponds to $Y_1 Z_1 \cdots Y_d Z_d \cdots Y_t Z_t$ as presented in Section 3. Unfortunately there is no explicit information about the way the blocks W_m are grouped to form the segments $Y_d Z_d$. But fortunately the local merges must be performed only on those pairs $Y_d Z_d$ such that $\text{last}(Y_d) > \text{first}(Z_d)$, hence the finish up can be done by repeating the following sequence until the whole segment $W_1 \cdots W_m \cdots W_{k+\ell} T_v$ has been processed:

- In order to locate the next pair $Y_d Z_d$ to be merged, scan to the left until a block W_m , such that

  $$\text{last}(W_m) > \text{first}(W_{m+1})$$

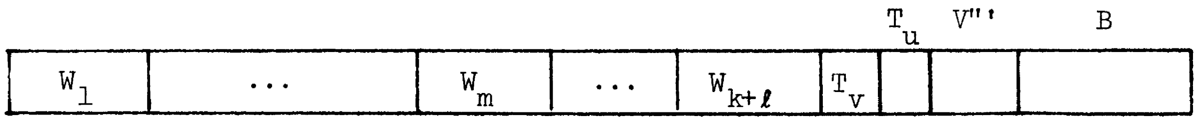
  is found.

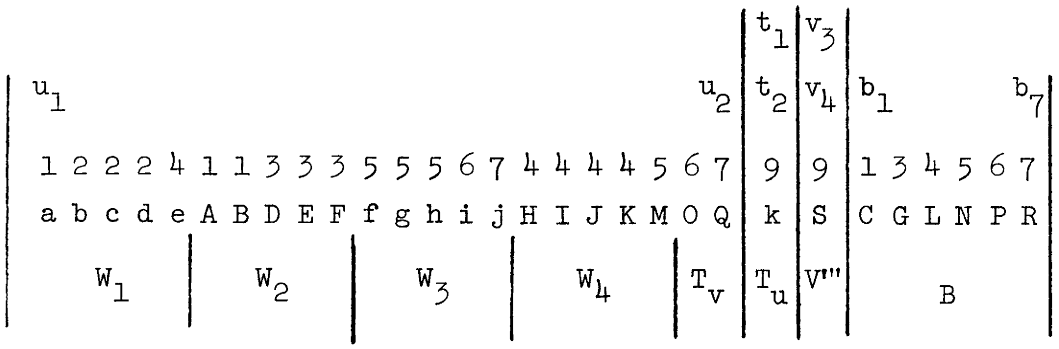
- Perform the local merge:

  1st step: Insert W_m in W_{m+1} , thus transforming $W_m W_{m+1}$ into $W' W_m W''$.

  2nd step: Merge W_m forward.

  3rd step: Merge W' backward.





t_1 v_3
u_1 u_2 t_2 v_4 b_1 b_7
1 2 2 2 4 1 1 3 3 3 5 5 5 6 7 4 4 4 4 5 6 7 9 9 1 3 4 5 6 7
a b c d e A B D E F f g h i j H I J K M O Q k S C G L N P R
W_1 W_2 W_3 W_4 T_v T_u V''' B

In the notation of Section 4,

$$W_1W_2W_3W_4 \equiv Y_1Z_1Y_2Z_2 \qquad (t = 2)$$

with the following grouping:

$$Y_1 = W_1 = U_1 \qquad Z_1 = W_2 = V_1$$

$$Y_2 = W_3 = U_2 \qquad Z_2 = W_4 = V_2 \quad .$$

Figure 5.4: After step 4.

Both definitions result in equivalent operation if the merging method stops once the merge is complete. In this case the bounds are preserved simply by the existing order in the file, thus making unnecessary to keep track of them. In other words, the grouping of $W_1 \cdots W_m \cdots W_{k+\ell}$ into $Y_1Z_1 \cdots Y_dZ_d \cdots Y_tZ_t$ is useful to prove that the algorithm works (and, as will be seen later, to compute its time bounds) but it is not needed to take it into account for implementation purposes.

Step 5: Finish up the merge of $W_1 \ldots W_m \ldots W_{\ell+k} T_v$.

```
finish_up:
   p := $v_2 - (v_2 - v_1 + 1)$ mod f; if p = $v_2$ then p := p-f;
   while p > $u_1$ do
      begin
         comment: find next pair $Y_d Z_d$ to be merged;
            while (p > $u_1$) ∧ (F(p) ≤ F(p+1)) do p := p-f;
         if p > $u_1$ then
            begin comment: local merge;
               comment: $W_m$ is F[$\ell_1$:$\ell_2$] , $W_{m+1}$ is F[$w_1$:$w_2$];
               $\ell_1$ := p - f + 1; $\ell_2$ := p;
               $w_1$ := p+1; $w_2$ := min(p+f , $v_2$);
               INSERT ($\ell_1$,$\ell_2$,$w_1$,$w_2$,$w_3$,$w_4$);
               comment: now W' is F[$w_1$:$w_2$] and W" is F[$w_3$:$w_4$];
               comment: in order to do the merges
                        "forward (of $W_m$)" means F[$w_3$:$v_2$],
                        "backward (of W')" means F[$u_1$:$w_1$-1];
               comment: depending on the size b of the buffer the
                        algorithm chooses:
                            BUFFER_MERGE if b ≥ f
                            BLOCK_MERGE if b < f;
               if b ≥ f
                  then begin
                           BUFFER_MERGE_FORWARD ($\ell_1$,$\ell_2$,$w_3$,$v_2$,$b_1$,$b_2$);
                           BUFFER_MERGE_BACKWARD ($u_1$,$w_1$-1,$w_1$,$w_2$,$b_1$,$b_2$)
                       end
                  else begin
                           BLOCK_MERGE_FORWARD ($\ell_1$,$\ell_2$,$w_3$,$v_2$);
                           BLOCK_MERGE_BACKWARD ($u_1$,$w_1$-1,$w_1$,$w_2$)
                       end;
               p := p-f
            end if_p
      end while_p;
```

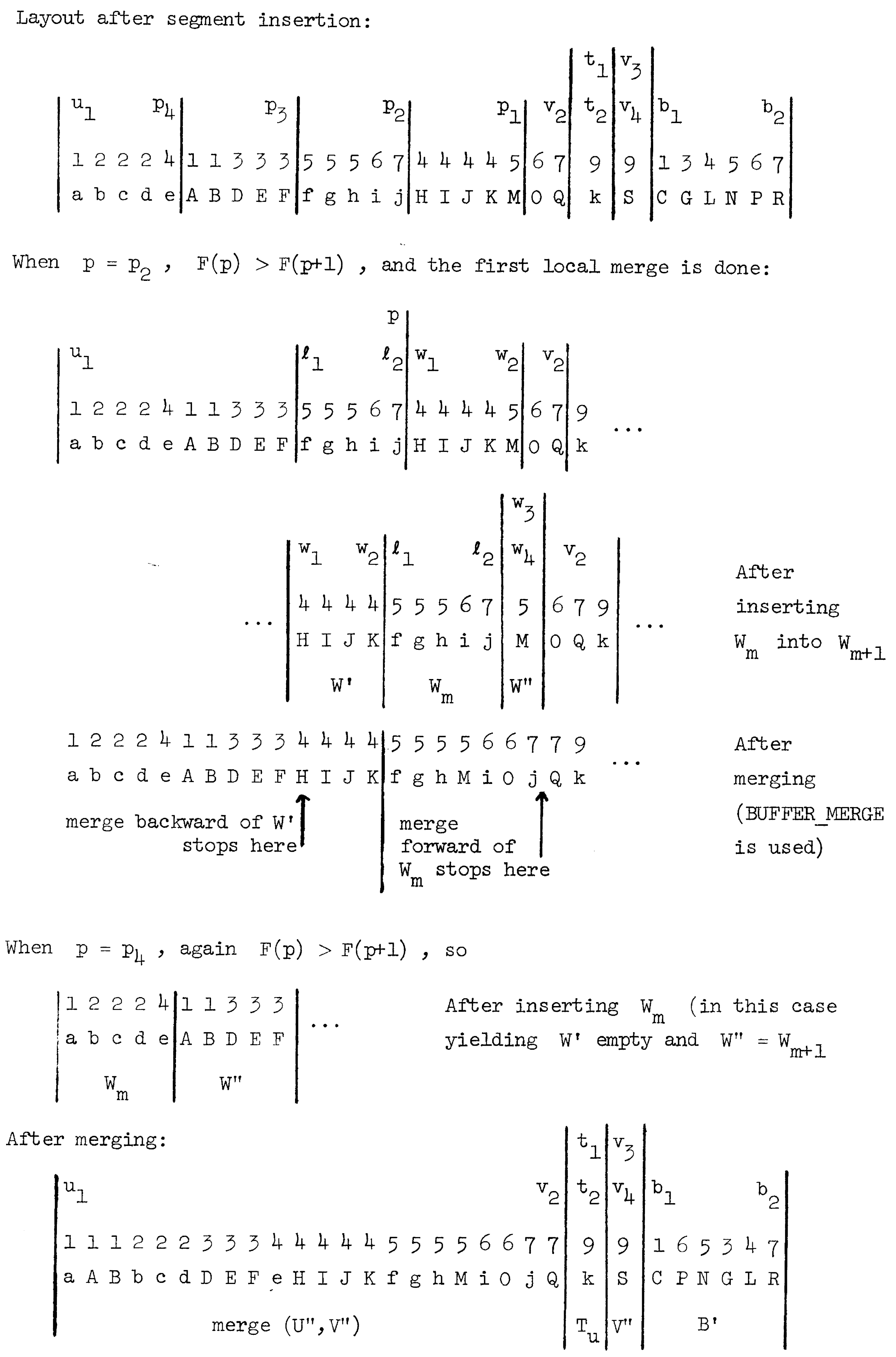

Layout after segment insertion:
u1 p4 p3 p2 p1 v2 t1 t2 v3 v4 b1 b2
1 2 2 2 4 | 1 1 3 3 3 | 5 5 5 6 7 | 4 4 4 4 5 | 6 7 | 9 | 9 | 1 3 4 5 6 7
a b c d e | A B D E F | f g h i j | H I J K M | O Q | k | S | C G L N P R
When p = p2 , F(p) > F(p+1) , and the first local merge is done:
p u1 ℓ1 ℓ2 w1 w2 v2
1 2 2 2 4 1 1 3 3 3 | 5 5 5 6 7 | 4 4 4 4 5 | 6 7 | 9 ...
a b c d e A B D E F | f g h i j | H I J K M | O Q | k ...
w1 w2 ℓ1 ℓ2 w3 w4 v2
... 4 4 4 4 | 5 5 5 6 7 | 5 | 6 7 9 ...
... H I J K | f g h i j | M | O Q k ...
W' Wm W"
After inserting Wm into Wm+1
1 2 2 2 4 1 1 3 3 3 4 4 4 4 | 5 5 5 5 6 6 7 7 9 ...
a b c d e A B D E F H I J K | f g h M i O j Q k ...
merge backward of W' stops here
merge forward of Wm stops here
After merging (BUFFER_MERGE is used)
When p = p4 , again F(p) > F(p+1) , so
1 2 2 2 4 | 1 1 3 3 3 | ...
a b c d e | A B D E F | ...
Wm W"
After inserting Wm (in this case yielding W' empty and W" = Wm+1
After merging:
u1 v2 t1 t2 v3 v4 b1 b2
1 1 1 2 2 2 3 3 3 4 4 4 4 4 5 5 5 5 6 6 7 7 | 9 | 9 | 1 6 5 3 4 7
a A B b c d D E F e H I J K f g h M i O j Q | k | S | C P N G L R
merge (U",V") Tu V" B'

Figure 5.5: The finish up process applied to the example.

It must be noticed that the algorithm chooses either BLOCK_MERGE or BUFFER_MERGE depending on the relative sizes of the blocks and the buffer.

Step 5 transforms the layout into

$$\text{merge}(U'',V'')\ T_u\ V'''\ B'$$

where B' is a permutation of B (and $B' = B$ if block_merge was used in step 5).

Step 6: Merge T_u and V''' .

```
merge_T_u_V''':
    if b ≥ t_2 - t_1 + 1
        then BUFFER_MERGE_FORWARD (t_1,t_2,v_3,v_4,b_1,b_2)
        else BLOCK_MERGE_FORWARD (t_1,t_2,v_3,v_4);
```

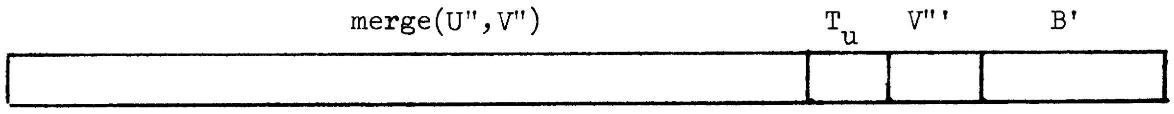
This step completes the merge of U' and V' , thus yielding $\text{merge}(U',V')B'$.

Step 7: Sort B' and merge it backward.

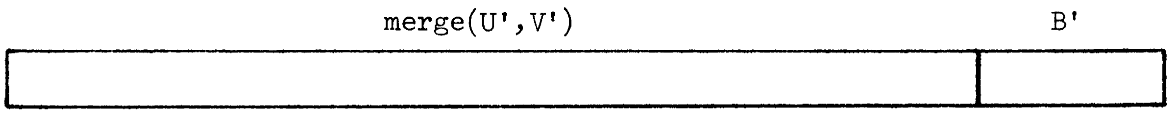
```
merge_back_B:
    STRAIGHT_INSERTION_SORT (b_1,b_2);
    BLOCK_MERGE_BACKWARD (u_1,v_4,b_1,b_2)
end partition_merge;
```

And step 7 finally yields the desired merge of U and V .

Figure 5.6: The general case after step 5 was



and after step 6 there results:



```
|u₁              merge(U',V')               v₄|    B'       |
|                                             |             |
| 1 1 1 2 2 2 3 3 3 4 4 4 4 4 5 5 5 5 6 6 7 7 9 9| 1 6 5 3 4 7 |
| a A B b c d D E F e H I J K f g h M i O j Q k S| C P N G L R |
```

Figure 5.7: Final result.

merge(U,V)

```
1 1 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 6 6 6 7 7 7 9 9
a A B C b c d D E F G e H I J K L f g h M N i O P j Q R k S
```

The storage requirements for the partition merge are the fixed number of pointers declared at the beginning (14 in total, though a more careful usage could have saved some) plus those needed by the different procedures called. Since those procedures (BUFFER_EXTRACT2, BLOCK_MERGE's, BUFFER_MERGE's, INSERT, SEGMENT_INSERT, and STRAIGHT_INSERTION_SORT) require also a fixed amount of pointers (and clearly there is no recursive call involved) the overall storage requirements are absolute minimum, that is $O(\log n)$.

5.2 Time Bounds for the Partition Merge Algorithm

The partition merge is executed as a fixed sequence of steps. The algorithm chooses the sequence in step 2, among the following two possibilities:

(i) If $|U'| \leq f$ or $|V'| \leq f$:

steps 1, 2, 7;

(ii) If $|U'| > f$ and $|V'| > f$:

steps 1, 3, 4, 5, 6, 7.

Hence, calling T_i the time bounds for the i-th step, it results that time bounds for the overall partition merge are either

(i) $T_1 + T_2 + T_7$

or (ii) $T_1 + T_3 + T_4 + T_5 + T_6 + T_7$.

The analysis of time bounds for each step follows.

5.2.1. Time Bounds for Step 1

The buffer extraction is given by equation (4.4)

$$T_{BE}(U,V,U',V',B,\ell) = O(|U| + |V|) + O(|B|^2) \quad .$$

Since the buffer B is restricted to

$$|B| \leq \left\lceil \sqrt{|U| + |V|} \right\rceil \quad ,$$

the time bounds are

$$T_1 = O(|U| + |V|) + O\left(\left\lceil \sqrt{|U| + |V|} \right\rceil^2\right)$$
$$= O(|U| + |V|) \quad . \qquad (5.1)$$

5.2.2. Time Bounds for Step 2

In the case that $|U'| \leq f$ or $|V'| \leq f$, this step yields a merge of U' and V'. Assume $|U'| \leq f$; here two cases must be considered.

(a) $|U'| > b$: this results in BLOCK_MERGE_FORWARD(U',V'), with bounds given by (2.6)

$$T_a = O(|V'|) + O(|U'| \cdot \lambda(U')) \quad . \qquad (5.2)$$

But $f \geq |U'| > b$, and by definition $f = \left\lfloor \frac{|U| + |V|}{b} \right\rfloor$, so $|U| + |V| > b^2$ and then

$$b < \left\lceil \sqrt{|U| + |V|} \right\rceil \quad . \qquad (5.3)$$

Recalling that the buffer extraction in step 1 asked for a buffer of length $\left\lceil \sqrt{|U| + |V|} \right\rceil$, and applying equation (4.3), implies

$$b = \lambda(UV) \quad . \qquad (5.4)$$

Now since $\lambda(U') \leq \lambda(UV)$, it is possible to bound

$$\lambda(U') \le b \quad . \tag{5.5}$$

Hence

$$O(|U'|\lambda(U')) = O(f \cdot b)$$

$$= O\left(\left\lfloor \frac{|U| + |V|}{b} \right\rfloor \cdot b\right) = O(|U| + |V|) \quad . \tag{5.6}$$

Equations (5.2) and (5.6) and the fact that $|V'| \le |V|$ give the final bounds

$$T_a = O(|V'|) + O(|U| + |V|) = O(|U| + |V|) \quad . \tag{5.7}$$

(b) $|U'| \le b$: then it is BUFFER_MERGE_FORWARD(U',V',B) , that yields, by equation (4.16)

$$T_b = O(|U'| + |V'|) = O(|U| + |V|) \quad . \tag{5.8}$$

Clearly, the case $|V'| \le f$ is similar, so by equations (5.7) and (5.8), it is possible to conclude that

$$T_2 = O(|U| + |V|) \quad . \tag{5.9}$$

5.2.3. Time Bounds for Step 3.

The insertion of T_u into V takes time proportional to the sum of both lengths, as stated by equation (2.5),

$$T_3 = O(|T_u| + |V|) = O(|U| + |V|) \quad . \tag{5.10}$$

5.2.4. Time Bounds for Step 4.

The segment insertion process bounds were established in equation (4.23) in Section 4.3. In the case of step 4, U'' was inserted in $V_1 \ldots V_\ell$, so

$$T_4 = T_{SEGIN}(U'', V_1 \ldots V_\ell, f)$$

$$= O((|U''| + |V_1 \ldots V_\ell|)^2 / f^2) + O(|U''| + |V_1 \ldots V_\ell|) \quad . \tag{5.11}$$

Clearly

$$|U''| \leq |U| \quad \text{and} \quad |V_1 \cdots V_\ell| \leq |V| \quad . \tag{5.12}$$

Also

$$|U''| + |V_1 \cdots V_\ell| \leq |U''| + |V''| \leq |U'| + |V'| = |U| + |V| - b \ . \tag{5.13}$$

By definition of f

$$f = \lfloor (|U| + |V|)/b \rfloor > (|U| + |V|)/b - 1 = (|U| + |V| - b)/b \ . \tag{5.14}$$

With (5.12), (5.13) and (5.14), (5.11) becomes

$$T_4 = O\left(\frac{(|U| + |V| - b)^2}{((|U| + |V| - b)/b)^2} \right) + O(|U| + |V|)$$

$$= O(b^2) + O(|U| + |V|) \quad . \tag{5.15}$$

And since $b \leq \left\lceil \sqrt{(|U| + |V|)} \right\rceil$,

$$O(b^2) = O(|U| + |V|) \tag{5.16}$$

thus

$$T_4 = O(|U| + |V|) \quad . \tag{5.17}$$

5.2.5. Time Bounds for Step 5.

In order to compute these bounds, it is convenient to resort to the notation in Section 3.

The segment insertion in step 4 transforms $U''V''$ into $Y_1Z_1 \cdots Y_dZ_d \cdots Y_tZ_tC_{t+1}$, where

$$U'' = Y_1 \cdots Y_d \cdots Y_t$$

$$V'' = Z_1 \cdots Z_d \cdots Z_tC_{t+1} \quad . \tag{5.18}$$

Also, let L_d be the last block of Y_d , and F_d the first one of Z_d , thus renaming

$$Y_d = Y'_d L_d \quad \text{and} \quad Z_d = F_d Z'_d$$

$$\text{with} \quad |L_d| = |F_d| = f \quad . \tag{5.19}$$

The finish up process of step 5 can be viewed as:

```
for d := t step -1 until 1 do
  begin
    insert $L_d$ into $F_d$, transforming $L_d F_d$ into $F'_d L_d F''_d$ ;
    merge $L_d$ forward;
    merge $F'_d$ backward;
  end
```

On the basis of the above description, the time bounds for step 5 result the sum of the time needed for insertions (T_I) plus time to merge forward (T_F) plus time needed to merge backward (T_B) . So

$$T_5 = T_I + T_F + T_B \quad .$$

Time bounds for insertion: Time bounds for insertion of two blocks are given by equation (2.4)

$$T_{INS}(X,Y) = O(|X| + |Y|) \quad ,$$

then

$$T_I = \sum_d T_{INS}(L_d,F_d) = O\left(\sum_d (|L_d| + |F_d|)\right)$$

$$= O\left(\sum_d |L_d|\right) + O\left(\sum_d |F_d|\right) \quad . \tag{5.20}$$

But $|L_d| \leq |Y_d|$ and $|F_d| \leq |Z_d|$, and since

$$\sum_d |Y_d| = |U''|$$

and

$$\sum_d |Z_d| \leq |V''|$$

equation (5.20) becomes

$$T_I = O\left(\sum_d |Y_d|\right) + O\left(\sum_d |Z_d|\right) = O(|U''| + |V''|) \quad . \tag{5.21}$$

Time bounds for merges: The time bounds for block and buffer merge are functions of those records that are actually exchanged (see remarks at the end of Subsections 2.2.5 and 4.2).

Claim 3.2 shows that during the finish up of Y_dZ_d , all the elements to the right of C_{d+1} are already in their final position. Hence when merging L_d forward, it merges into $F''_d Z'_d C_{d+1}$, regardless of how far to the right of C_{d+1} the merge limits point. So in order to bound the running time the process "merge L_d forward" will be regarded as "merge L_d forward into $F''_d Z'_d C_{d+1}$ ".

A quite similar reasoning shows that "merge F'_d backward" is equivalent to "merge F'_d backward into Y'_d ".

There are two cases depending on whether block or buffer merge is used, and they will be analyzed separately:

(a) Case $b \geq f$: BUFFER_MERGE.

By equations (4.15) and (4.16) the time to buffer merge two blocks X and Y (either forward or backward) can be bounded by $O(|X|) + O(|Y|)$, so:

$$T_F = \sum_d O(|L_d|) + \sum_d O(|F''_d Z'_d C_{d+1}|) \tag{5.22}$$

$$T_B = \sum_d O(|F'_d|) + \sum_d O(|Y'_d|) \quad . \tag{5.23}$$

Combining (5.22) and (5.23) and manipulating the lengths properly:

$$T_F + T_B = \sum_d O(|Y'_d L_d|) + \sum_d O(|F'_d F''_d Z'_d|) + \sum_d O(|C_{d+1}|)$$

$$= O(|U''|) + O(|V''|) + O(|V''|) = O(|U''| + |V''|) \quad . \qquad (5.24)$$

(b) Case of $b < f$: BLOCK_MERGE.

Since

$$T^{(forw)}_{BLOCKM}(X,Y) = O(|X|\lambda(X)) + O(Y) \quad ,$$

it is

$$T_F = \sum_d O(|L_d| \cdot \lambda(L_d)) + \sum_d O(|F''_d Z'_d C_{d+1}|)$$

$$= f\, O\left(\sum_d \lambda(L_d)\right) + O(|V''|) \quad , \qquad (5.25)$$

and also

$$T^{(back)}_{BLOCKM}(X,Y) = O(|X|) + O(|Y|\lambda(Y)) \quad ,$$

yielding

$$T_B = \sum_d O(|Y_d|) + \sum_d (|F'_d| \cdot \lambda(F'_d))$$

$$= O(U'') + f\, O\left(\sum_d \lambda(F'_d)\right) \quad . \qquad (5.26)$$

In order to bound the sums in equations (5.25) and (5.26), the following result is needed:

Claim 5.1: Let U be an ordered block. Consider U as a segment of k blocks $U = U_1 U_2 \cdots U_i \cdots U_k$, then

$$\lambda(U) + k \geq \sum_{1 \leq i \leq k} \lambda(U_i) \quad . \qquad (5.27)$$

Proof: Let

$$g(i) = \begin{cases} 1 & \text{if } \mathrm{last}(U_i) = \mathrm{first}(U_{i+1}) \\ 0 & \text{otherwise} \end{cases}$$

for $1 \leq i < k$.

Then clearly

$$\sum_{1 \leq i \leq k} \lambda(U_i) = \lambda(U) + g(1) + \ldots + g(k-1)$$

yielding the claim.

$\square$

Applying (5.27) to the sums in (5.25) and (5.26) yields:

$$\sum_{1 \leq d \leq t} \lambda(L_d) \leq \sum_{1 \leq d \leq t} \lambda(Y_d) \leq \lambda(U'') + t \qquad (5.28)$$

$$\sum_{1 \leq d \leq t} \lambda(F'_d) \leq \sum_{1 \leq d \leq t} \lambda(Z_d) \leq \lambda(V'') + t \quad . \qquad (5.29)$$

Combining (5.25) and (5.26) and using (5.28) and (5.29), the time bounds result:

$$T_F + T_B = O(|U''| + |V''|) + f.O(\lambda(U'') + t) + f\ O(\lambda(U'') + t) \quad . \qquad (5.30)$$

By the same analysis as in 5.2.2, the fact that $f > b$ implies

$$\lambda(UV) = b \quad , \quad \text{and then}$$

$$\lambda(U'') \leq b \quad \text{and} \quad \lambda(V'') \leq b \quad . \tag{5.31}$$

Also t is bounded by the number of blocks (of length f) in U'' and V''

$$t \leq \frac{|U''V''|}{f} \leq \frac{|U'V'|}{\lfloor (|U| + |V|)/b \rfloor} \leq \frac{|U| + |V| - b}{(|U| + |V| - b)/b} = b \; . \tag{5.32}$$

Equations (5.31) and (5.32) and the fact that $f \cdot b \leq |U| + |V|$, applied to (5.30) give

$$T_F + T_B = O(|U''| + |V''|) + f \cdot O(b + b) + f \cdot O(b + b)$$
$$= O(|U| + |V|) \quad . \tag{5.33}$$

In summary:

Time for insertion: equation (5.21) shows that

$$T_I = O(|U''| + |V''|) \quad .$$

Time for merges:

Case $b \geq f$: by equation (5.24)

$$T_F + T_B = O(|U''| + |V''|) \quad .$$

Case $b < f$: by equation (5.33)

$$T_F + T_B = O(|U| + |V|) \quad .$$

Clearly the time bounds for step 5 result

$$T_5 = O(|U| + |V|) \quad . \tag{5.34}$$

5.2.6. Time Bounds for Step 6.

By an analysis completely similar to the one for step 2, the time bounds for the merge of T_u into V'' result

$$T_6 = O(|U| + |V|) \tag{5.35}$$

5.2.7. Time Bounds for Step 7.

By equation (2.8), the time bounds to sort the buffer B are

$$T_S = T_{SORT}(B) = O(|B|^2) = O(|U| + |V|) \quad . \tag{5.36}$$

Also the block merge of B into the rest of the file

$$T^{(back)}_{BLOCKM}(\text{merge}(U',V'), B)$$
$$= O(|\text{merge}(U',V')|) + O(|B|\lambda(B)) \quad . \tag{5.37}$$

But since B is a buffer $|B| = \lambda(B)$ and thus (5.37) becomes

$$T_M = O(|U'| + |V'|) + O(|B|^2) = O(|U| + |V|) \quad . \tag{5.38}$$

Finally

$$T_7 = T_S + T_M = O(|U| + |V|) \quad . \tag{5.39}$$

5.2.8. Overall Time Bounds

Equations (5.1), (5.9), (5.10), (5.17), (5.34), (5.35) and (5.39) show that each single step has time bounds $O(|U| + |V|)$. The conclusion is that the overall process must have also linear bounds, since it consists of a fixed sequence of those steps.

$$T_{PARTM}(U,V) = O(|U| + |V|) \quad . \tag{5.40}$$

6. The Partition Merge Sort

The availability of a linear time merge algorithm gives rise to the possibility of an $(n \cdot \log n)$ time bounded sort. A few slightly different variations of the same basic strategy are possible, and this section presents one of them in detail.

The sorting strategy consists of successive merging passes over the entire block to be sorted, each pass merging pairs of blocks of length $1, 2, 4, \ldots, 2^k, \ldots$ until the entire file is sorted.

6.1 Description

The following procedure sorts a block U , whose first and last elements are pointed to by u_1 and u_2 respectively.

```
procedure partition_merge_sort (pointer value $u_1, u_2$);
begin pointer p, $\ell$;
   comment: $\ell$ is the length of the blocks to be merged;
   $\ell$ := 1;
   while $\ell < u_2 - u_1 + 1$ do
      begin comment: merging of contiguous pairs of blocks of
                     length $\ell$. The pointer p points at the
                     first element of the second block of each pair;
         p := $u_1 + \ell$;
         while $p \leq u_2$ do
            begin
               partition_merge(p-$\ell$ , p-1 , p , min(p+$\ell$-1 , $u_2$));
               p := p + 2*$\ell$
            end
         $\ell$ := 2*$\ell$
      end
end partition_merge_sort;
```

6.2 Time Bounds

Since partition merge is linear on the length of the blocks to be merged, each merging pass results also linear on the length of the block U being sorted, regardless of the value of ℓ. That is, denoting by M_ℓ the time bounds for the merging pass of blocks of length ℓ it is

$$M_\ell = O(|U|) \quad . \tag{6.1}$$

But the merging passes are repeated for lengths

$$\ell = 1, 2, \ldots, 2^i, \ldots, 2^k$$

such that 2^k does not exceed the length of U :

$$2^k < |U| \leq 2^{k+1} \quad . \tag{6.2}$$

So the time bounds for the sorting process are:

$$T_{P\_M\_SORT}(U) = \sum_{0 \leq i \leq k} M_{2^i}(U)$$

$$= \sum_{0 \leq i \leq k} O(|U|) = (k+1)O(|U|) \quad . \tag{6.3}$$

From equation (6.2) it results

$$k+1 = \lceil \log_2 |U| \rceil \quad . \tag{6.4}$$

Finally yielding

$$T_{P\_M\_SORT}(U) = O(|U| \log |U|) \quad . \tag{6.5}$$

7. Conclusions

The most interesting of the results presented here is the PARTITION MERGE algorithm, since as the reader was able to see, the PARTITION MERGE SORT resulted as a straightforward consequence of it.

By analyzing the previously published results, especially the work by Horvath ([Horvath]), it can be concluded that there were two considerations that led to the general result presented here.

First, the utilization of an internal buffer, without any modification of the keys, to "mark" a permutation of a segment, allowed the segment insertion process to be implemented within extra storage bounds of $O(\log n)$ bits.

Secondly, the adaptivity of the algorithm to the characteristics of the file being processed (by proper choice of either BUFFER or BLOCK MERGE) resulted in a linear time "finish up".

It is interesting to note that the operation ' $p+q$ ' on pointers is strictly needed only for the permutation of blocks in the SEGMENT_INSERT process (Section 4.3). All the other sums of pointer values could have been realized by successive ' $p+1$ ' operations within the same time and space bounds. It remains an open question whether these minimum time and space bounds are obtainable only with the primitives ' exchange(p,q) ', ' $F(p) \leq F(q)$ ', ' $p \pm 1$ ', ' $p = q$ ', and ' $p := q$ '.

Acknowledgments

The author is greatly indebted to Professor Donald E. Knuth, not only for his help and encouragement in the present work, but for the many things the author learned from him.

APPENDIX A

Analysis of Basic Transformations

This Appendix presents a detailed analysis of the basic transformations defined in Section 2.2.

Each transformation is defined by means of an ALGOL procedure and the corresponding analysis of the running time bounds is presented.

The blocks U and V are used as parameters, and they correspond to $F[u_1:u_2]$ and $F[v_1:v_2]$ respectively.

In order to allow dealing with empty blocks, an empty block U is represented by $\langle u_1,u_2\rangle$ with $u_2 = u_1-1$. The pointers used to represent the segment UV have $u_2+1 = v_1$ in all cases, even when one of the blocks is empty.

A.1 Reversal of a block: $\text{REVERSE}(u_1,u_2)$

Algorithm:

```
procedure REVERSE (pointer $u_1,u_2$);
for $j := u_1$ step 1 until $(u_2+u_1)/2$ do exchange($j$ , $u_2 - j + u_1$);
```

Time bounds: Clearly

$$T_{REV}(U) = O(|U|) \quad . \tag{A.1}$$

A.2 Exchange of blocks of equal length: BLOCK_EXCHANGE(u_1, u_2, v_1, v_2)

Algorithm:

```
procedure BLOCK_EXCHANGE (pointer $u_1, u_2, v_1, v_2$);
for j := $u_1$ step 1 until $u_2$ do exchange(j , $v_1 + j - u_1$);
```

Restrictions: $|U| = |V|$.

Time bounds: The for loop is executed $|U|$ times, thus

$$T_{BEX}(U,V) = O(|U|) = O(|V|) \quad . \tag{A.2}$$

A.3 Permutation of two contiguous blocks: PERMUTE(u ,u ,v ,v)

Algorithm:

```
procedure PERMUTE (pointer $u_1, u_2, v_1, v_2$);
  begin
    pointer t;
    REVERSE($u_1, v_2$); comment: yields $V^R U^R$;
    comment: exchange pointers;
    t := $v_2$; $v_1$ := $u_1$; $v_2$ := $v_2 - u_2 + u_1 - 1$;
            $u_1$ := $v_2$+1; $u_2$ := t;
    REVERSE($u_1, u_2$); comment: $V^R U$;
    REVERSE($v_1, v_2$); comment: VU;
  end PERMUTE;
```

Restrictions: U and V must be contiguous with U preceding V ; i.e.,

$v_1 = u_2+1$.

Time bounds: Three reverses are executed, all of them linear on the length of the blocks, so

$$T_{REV}(U,V) = O(|U| + |V|) \quad . \qquad (A.3)$$

A.4 Stable insertion of two contiguous ordered blocks:

INSERT(u_1,u_2,v_1,v_2,f_1,f_2)

Algorithm:

```
procedure INSERT (pointer $u_1,u_2,v_1,v_2,f_1,f_2$);
  if $(u_1 \le u_2) \wedge (v_1 \le v_2)$ then
    begin
      comment: search for insertion place;
      $f_1 := v_1$; $f_2 := v_2$;
      while $(f_1 \le v_2) \wedge (F(f_1) < F(u_1))$ do $f_1 := f_1+1$;
      comment: now V' is $F[v_1:f_1-1]$ and V" is $F[f_1:f_2]$;
        $v_2 := f_1-1$; PERMUTE($u_1,u_2,v_1,v_2$);
    end INSERT;
```

Restrictions: $(v_1 = u_2+1)$ and ordered(U) and ordered(V) .

Time bounds: The search compares the elements of V' until reaching the first element of V'' (V' and V'' as defined in Section 2.2.4) and PERMUTE permutes U and V', thus the bounds are

$$T_{INS}(U,V) = O(|U| + |V'|) \qquad (A.4)$$

with $V = V'V''$ and $last(V') < first(U) \leq first(V'')$.

A.5 Direct merge of two contiguous ordered blocks: BLOCK_MERGE

Only BLOCK_MERGE_FORWARD will be considered here.

Algorithm:

```
procedure BLOCK_MERGE_FORWARD (pointer value $u_1,u_2,v_1,v_2$);
  if $(u_1 \leq u_2) \wedge (v_1 \leq v_2)$ then
    begin pointer $x_1,x_2,y_1,y_2,c_1,c_2$;
      $x_1 := u_1$; $x_2 := u_2$; $y_1 := v_1$; $y_2 := v_2$;
      while $(x_1 \leq x_2) \wedge (y_1 \leq y_2)$ do
        begin
          INSERT($x_1,x_2,y_1,y_2,c_1,c_2$);
          comment: any element to the left and including $y_2$
                   is in its final position. The merge is
                   reduced to the merge of $F[x_1:x_2]$ with $F[c_1:c_2]$;
          $y_1 := c_1$; $y_2 := c_2$;
          if $(y_1 \leq y_2) \wedge (F(x_2) > F(y_1))$ then
            begin comment: discard the prefix of $F[x_1:x_2]$
                           already in its final position;
              while $(x_1 \leq x_2) \wedge (F(x_1) \leq F(y_1))$ do $x_1 := x_1+1$;
            end
        end
    end BLOCK_MERGE_FORWARD;
```

Restrictions: $v_1 = u_2+1$ and ordered(U) and ordered(V) .

Time bounds: Let

$$U = U_1 \cdots U_i \cdots U_t \;, \quad |U_i| > 0 \quad \text{for} \quad 1 \leq i \leq t$$

and (A.5)

$$V = V_0\,V_1 \cdots V_i \cdots V_t \;, \quad |V_i| > 0 \quad \text{for} \quad 0 < i < t$$

where

$$\text{last}(U_i) \leq \text{first}(V_i)$$

and (A.6)

$$\text{last}(V_i) < \text{first}(U_{i+1}) \quad .$$

With this notation the merge of U and V can be expressed as

$$\text{merge}(U,V) = V_0\,U_1\,V_1 \cdots U_i\,V_i \cdots U_t\,V_t \quad . \tag{A.7}$$

Furthermore, the block merge process may now be defined as follows:

```
for i := 1 until t do
  begin
    insert $U_i \cdots U_t$ into $V_{i-1}\,V_i \cdots V_t$;
    if i < t
      then search through $U_i$ until reaching the first element
           of $U_{i+1}$;
  end
```

The insertion of $U_i \cdots U_t$ into $V_{i-1}\,V_i \cdots V_t$ yields, according to (A.6), $V_{i-1}\,U_i \cdots U_t\,V_i \cdots V_t$, hence by equation (A.4) it takes time

$$TI_i = O(|V_{i-1}| + |U_i \cdots U_t|) \quad . \tag{A.8}$$

The search to find the first element of U_{i+1} takes time proportional to the length of U_i ,

$$TS_i = O(|U_i|) \quad . \tag{A.9}$$

Thus the overall time bounds result

$$\begin{aligned} T^{(forw.)}_{BLOCKM}(U,V) &= \sum_{1 \le i \le t} TI_i + \sum_{1 \le i < t} TS_i \\ &= \sum_{1 \le i \le t} O(|V_{i-1}| + |U_i \cdots U_t|) + \sum_{1 \le i < t} O(|U_i|) \\ &= O(|V_0 V_1 \cdots V_{t-1}|) + \sum_{1 \le i \le t} O(|U_i \cdots U_t|) \\ &\qquad + \sum_{1 \le i < t} O(|U_i|) \\ &= O(|V_0 V_1 \cdots V_{t-1}|) + \sum_{1 \le i \le t} O(|U_i \cdots U_t|) \quad . \end{aligned} \tag{A.10}$$

Clearly $|U_i \cdots U_t| \le |U|$. Since equation (A.6) implies $last(U_i) < first(U_{i+1})$, the keys in U_i are distinct from the keys in U_{i+1} . Thus t is bounded by $\lambda(U)$, and the sum in (A.10) is

$$\sum_{1 \le i \le t} O(|U_i \cdots U_t|) = O(|U|).t = O(|U|\lambda(U)) \quad . \tag{A.11}$$

Renaming $V_0 \cdots V_{t-1} = V'$ and $V_t = V''$, equation (A.6) yields

$$last(V') < last(U) \le first(V'') \quad . \tag{A.12}$$

And finally the time bounds can be expressed as

$$T^{(forw.)}_{BLOCKM}(U,V) = O(|U|\lambda(U)) + O(|V'|) \quad . \tag{A.13}$$

APPENDIX B

Analysis of the Buffer Extraction Process

The concept of buffer extraction was introduced in Section 4.1. This Appendix presents a slightly more general extraction mechanism and its application in order to produce a buffer from two contiguous ordered blocks.

B.1 The EXTRACT transformation

Let $U = F[u_1:u_2]$ be an ordered block and $M = F[m_1:m_2]$ a buffer (U and M do not overlap). Then an application of $\mathrm{EXTRACT}(u_1,u_2,\ell,b_1,b_2,m_1,m_2)$ transforms U into $U'B$ such that

$$B \text{ is a buffer}, \quad B = F[b_1:b_2] \quad , \quad |B| \le \ell$$
$$U = \mathrm{merge}(U',B) \quad , \tag{B.1}$$

no record in B has a key equal to the key of any record in M (that is, $\forall i: m_1 \le i \le m_2$, $\forall j: b_1 \le j \le b_2$: $F(i) \ne F(j)$), and $|B|$ is as large as possible subject to these conditions.

The extraction is similar to the mechanism presented in the example in Section 4.1, with the addition of a check to avoid collecting any record whose key is already in M .

The following procedure describes the EXTRACT process:

```
procedure EXTRACT (pointer $u_1,u_2,\ell,b_1,b_2,m_1,m_2$);
  begin pointer p,q,s;
    logical procedure is_in_M (pointer q);
      begin
        while $(s \le m_2) \wedge F(s) < F(q))$ do s := s+1;
        if $s > m_2$ then false
                  else $(F(q) = F(s))$
      end is_in_M;
    s := $m_1$; comment:  s will point to successive elements in M;
    $b_1$ := $u_1$; $b_2$ := $u_1$-1; comment:  B is initially empty;
    comment:  collect the buffer;
    while $(b_2 < u_2) \wedge (b_2 - b_1 + 1 < \ell)$ do
      begin
        comment:  set q to point to the next element to be
                  included in B,
             or   set q = $u_2$+1 if no such element exists;
        p := q := $b_2$+1;
        while $(q \leq u_2)$ ∧ is_in_M(q) do q := q+1;
        while $(q < u_2) \wedge F(q) = F(q+1))$ do q := q+1;
        if q $\leq u_2$ then
        begin comment:  permute B and the elements preceding record q;
          q := q-1; PERMUTE($b_1$,$b_2$,p,q);
          $b_2$ := $b_2$+1; comment:  include the record q in B;
        end;
      end;
    comment:  permute B with the elements (if any) to its right;
    p := $b_2$+1; PERMUTE($b_1$,$b_2$,p,$u_2$);
  end EXTRACT;
```

In the above program the procedure is_in_M checks whether a given key is or is not in M . In the following analysis the execution time for a call to is_in_M will be considered fixed, with the proviso that an $O(|M|)$ time is added to the total time bounds. The reason for the above statement is that is_in_M is called upon to check successive keys in U , and thus it needs to run through M only once during the entire execution of EXTRACT.

The buffer B is collected from left to right. Assume that after collecting the first i elements of B the block U has been transformed into

$$U'_i \; B_i \; U''_i \tag{B.2}$$

where U'_i is a prefix of U' ,

B_i is a prefix of B , $|B_i| = i$

and U''_i has not yet been considered.

The execution of EXTRACT now proceeds as follows:

-- search through U''_i until reaching the first record q , such that

$$\forall j: m_1 \leq j \leq m_2: F(q) \neq F(j)) \text{ and } (q = u_2 \text{ or } F(q) < F(q+1)) \;;$$

-- permute B_i with the elements to its right that precede the record q , and append q to B_i thus yielding

$$U'_{i+1} \; B_{i+1} \; U''_{i+1} \quad .$$

The time needed to search is proportional to the difference of lengths between U'_{i+1} and U'_i ,

$$TS_i = c(|U'_{i+1}| - |U'_i|) \quad . \tag{B.3}$$

The permute time is of the order of the length of B_i and the distance between q and the rightmost element of B ,

$$TP_i = d(|B_i| + |U'_{i+1}| - |U'_i|) \quad . \tag{B.4}$$

Hence the overall bounds result

$$T_{EXTR}(U,\ell,B,M) = \sum_{1 \leq i \leq |B|} (TS_i + TP_i) + O(|M|) \quad , \tag{B.5}$$

where the $O(|M|)$ term is the "extra" contribution of is_in_M .

After some manipulation (B.3), (B.4) and (B.5) yield

$$T_{EXTR}(U,\ell,B,M) = O(|B|^2) + O(|U|) + O(|M|) \quad . \qquad (B.6)$$

B.2 Extraction of a buffer from two contiguous ordered blocks:

BUFFER_EXTRACT2($u_1,u_2,v_1,v_2,\ell,b_1,b_2$)

An application of BUFFER_EXTRACT2 produces a buffer B , of length $|B| = \min(\ell, \lambda(UV))$, out of two contiguous ordered blocks U and V , yielding $U' V' B$ where

$$\text{merge}(\text{merge}(U',V'),B) = \text{merge}(U,V) \quad . \qquad (B.7)$$

This transformation is implemented by means of two successive applications of EXTRACT. The following procedure defines the algorithm:

```
procedure BUFFER_EXTRACT2 (pointer $u_1,u_2,v_1,v_2,\ell,b_1,b_2$);
  begin pointer $c_1,c_2$;
    comment: EXTRACT($V,\ell,B,M$) with M empty (thus no restriction
             is imposed on the elements to be collected);
    $c_1$ := 1; $c_2$ := 0;
    EXTRACT($v_1,v_2,\ell,b_1,b_2,c_1,c_2$)
    if $(b_2 - b_1 + 1) < \ell$ then
      begin comment: previous extraction was not enough;
        EXTRACT($u_1,u_2,\ell-(b_2-b_1+1),c_1,c_2,b_1,b_2$)
        PERMUTE($c_1,c_2,v_1,v_2$);
        BLOCK_MERGE_FORWARD($c_1,c_2,b_1,b_2$);
        $b_1$ := $c_1$;
      end
  end BUFFER_EXTRACT2;
```

To analyze the time bounds, two cases must be considered:

(i) The first extraction suffices: then by (B.6)

$$T_{(i)} = O(|B|^2) + O(|V|) \quad . \tag{B.8}$$

(ii) Two extractions are needed: let B_1, $|B_1| = b_1$, be the buffer collected in the first extraction and B_2, $|B_2| = b_2$, the second one; $|B| = b_1 + b_2$. The bounds result

$$\begin{aligned} T_{(ii)} = \; & O(b_1^2) + O(|V|) \\ & + O(b_2^2) + O(|U|) + O(b_1) \\ & + O(b_2) + O(|V|) \\ & + O(b_2\,\lambda(B_2)) + O(b_1) \end{aligned} \tag{B.9}$$

but since B_2 is a buffer $\lambda(B_2) = b_2$, thus (B.9) becomes

$$\begin{aligned} T_{(ii)} &= O(b_1^2) + O(b_2^2) + O(|U| + |V|) \\ &= O(|B|^2) + O(|U| + |V|) \quad . \end{aligned} \tag{B.10}$$

Finally (B.8) and (B.10) yield

$$T_{BE}(U,V,U',V',\ell,B) = O(|U| + |V|) + O(|B|^2) \quad . \tag{B.11}$$

References

[Dewar] Robert B. K. Dewar, "A Stable Minimum Storage Algorithm," Information Processing Letters 2 (April, 1974), 162-164.

[Horvath] Edward C. Horvath, "Efficient Minimum Extra Space Stable Sorting," Ph.D. Thesis, Dept. of Electrical Engineering, Princeton University, (August, 1974).

[Knuth] Donald E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, (Addison-Wesley, December 1973).

[Knuth] Donald E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, (Addison-Wesley, 1973).

[Kronrod] M. A. Kronrod, "An Optimal Ordering Algorithm without a Field of Operation," Dokladi Adad. Nauk SSSR 186 (1969), 1256-1258.

[Nijenhuis] Albert Nijenhuis, private communication (1974).

[Pratt] Vaughan Pratt, private communication (1974).

[Preparata] F. P. Preparata, "A Fast Stable Sorting Algorithm with Absolutely Minimum Storage," Istituto di Science dell'Informazione, Universita di Pisa, Italy (March, 1974).

[Rivest] Ronald Rivest, "A Fast Stable Minimum Storage Sorting Algorithm," IRIA Report No. 43 (December, 1973).