Chapter 3

File sorting

In this chapter we will consider sorting a physical file, simply referred to as a file. As usual, we will assume that the file is very large and it cannot fit into the main memory. Using a small number of buffers, a few buckets of the file can be accessed at a time for internal processing required for sorting. This type of sorting is called *external sorting*. External sorting is different from *internal sorting* where the whole file can be stored in the main memory at once.

In order to fix notation, we will also assume that sorting will be done in nondecreasing order (≤) of key values. For the sake of simplicity, in this chapter we will implicitly assume that all buckets in the file are full, that is, the space utilization is 100%. Note that a key may consist of several attributes, and it may be a unique or a nonunique key. Throughout this chapter we will only consider the discrete model of the disk, but some exercises will consider cascades. We assume that the file contains N buckets and B buffers are available to sort it.

To understand external sorting, we need the introduce the concept of a run. A *run* is a sorted segment of the file. Consider the following example.

Example 3.1. Consider the emp file of Figure 1.3. That file is sorted by Name, but not sorted by ID. Suppose we use ID as our key. Then the sequence of records in the file is $\overline{151}$, $\overline{306}$, $\overline{200}$, $\overline{310}$, $\overline{101}$, $\overline{311}$, $\overline{312}$, $\overline{305}$, $\overline{165}$, $\overline{160}$, $\overline{300}$, $\overline{180}$. The sequence consists of the following 7 runs (separated by semicolons): $\overline{151}$, $\overline{306}$; $\overline{200}$, $\overline{310}$; $\overline{101}$, $\overline{311}$, $\overline{312}$; $\overline{305}$; $\overline{165}$; $\overline{160}$, $\overline{300}$; $\overline{180}$.

We face a problem that we may not be aware of the runs that are contained in a given file. For example, we may be given a file that happens to be completely sorted. If we are not aware of this, even to realize that the file is sorted, the least we have to do is read all buckets in the file once, costing us N disk accesses. In order to recognize this problem, we introduce the terms *real run* and *certified run*. A real run refers to a run that is present in a file, irrespective of whether we are aware of it or not. On the other hand, a *certified run* is a sorted segment that we know is a run. When we are given a file without any information about distribution of keys, we may assume that keys are completely out of order, that is, every record forms a run. Thus, the number of certified runs is same as the number of records in the file. The file in Example 3.1 contains 7 real runs and 12 certified runs. The goal of sorting is to reduce the number of certified runs to 1.

We will first present the basis underlying method of external sorting. Although, the cost of processing could be substantial, we do not consider it on a formal basis in this book. We do not formally quantify the number of disk accesses either. After introducing the basic method of sorting, we present some interesting ideas that improve cost of processing and disk accesses.

3.1 Basic underlying method of sorting

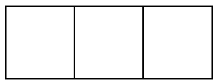
Essentially, external sorting is done in two phases: *creation of runs*, followed by *merging*. In the phase for creation of runs, segments of the file are brought into the main memory and sorted using an internal sort. At the end of this phase, each segment becomes a run. In the merge phase, the runs are continually merged together to form larger runs until the whole file becomes a single run. The two phases are illustrated through the example given below.

Example 3.2. Consider a file containing 1,800 records. Assume that 100 records fit in 1 bucket. Therefore, the file contains 1,800/100 = 18 buckets. Assume that B = 3, that is, we are given 3 buffers to sort this file.

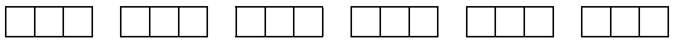
Phase 1: Creation of runs

For creation of runs, it is appropriate to use all the B buffers that are available to us. These buffers are used to read a segment of B buckets from the file, sort it into the main memory, thereby making and certifying it as a single run, and write the segment back to the file.

For sorting the file in Example 3.2, we are given B = 3 buffers. We configure all 3 buffers for reading and writing. The buffer configuration is shown in Figure 3.1(a). In accordance with this buffer configuration, we may view the file to consist of 6 segments of 3 buckets each. This view of the file is shown in Figure 3.1(b).



(a) Buffer configuration:
All 3 buffers configured for input and output



(b) File viewed as 6 segments of 3 buckets each

Figure 3.1 Creation of runs for the file of Example 3.2

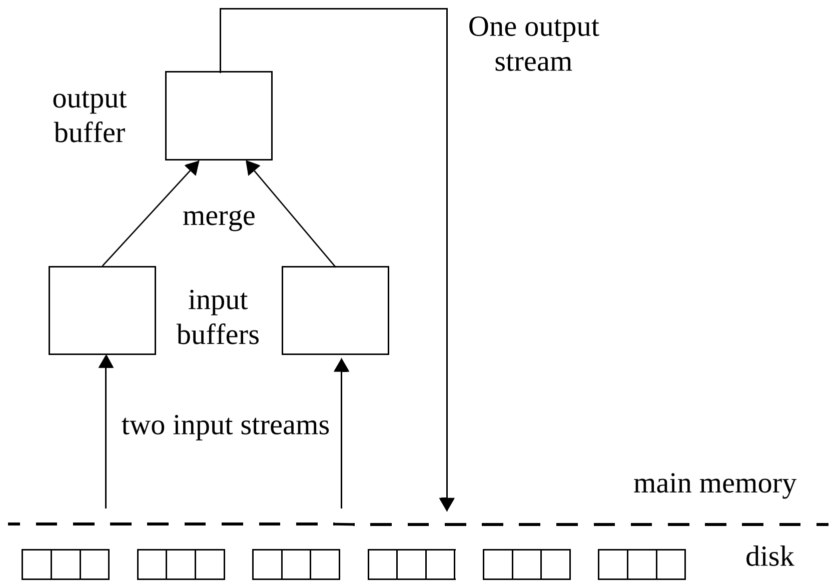
We read the first 3-bucket segment into the buffers, sort the data in these buffers as a single run using an internal sort, and finally write the 3 buffers on the disk. If we do not care about disturbing the original file, we can write the 3-bucket segment in its original location. In other words, the old 3-bucket segment gets overwritten as a 3-bucket run. This procedure is repeated for the remaining 5 segments. In this way we obtain 6 runs of 3 buckets each, which completes the run-creation phase.

Phase 2: Merging

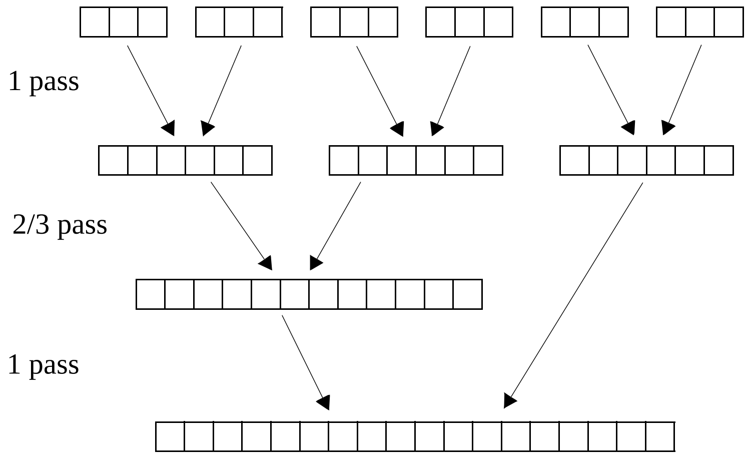
The main idea in merging is that two or more runs can be merged together to produce one larger run. This processes can be repeated on the given runs until one run is left, at which point the file is sorted.

For our running example, let's configure the 3 buffers to setup two input streams and one output stream as shown in Figure 3.2. With this configuration we can merge two runs at a time. We couple each run with one of the 2 input buffers. Then we merge one record at a time by transferring it to the output buffer. The output buffer is written to create a larger run out of the given runs. Whenever one input buffer becomes empty, processing is suspended and the next bucket from the coupled run is read into the buffer. Whenever the output buffer becomes full, the processing is suspended and the buffer is written to the disk. The sequence in which the 3 buffers get read and written is unpredictable. (Also, think what happens when a run coupled to one input stream finishes before the other input stream?)

Merging may require several full and partial passes. For our file, the 2-way merge consists of a binary tree as shown in Figure 3.2. In the first pass we reduce 6 runs of 3 buckets each into 3 runs of 6 buckets each. The second pass is a partial pass; only 2 of the 3 runs can be paired and merged leading to a 12-bucket run. Now we are left with



(a) Merging of runs for the file of Example 3.2



(b) The merge tree

Figure 3.2 Merging

a 12-bucket run and a 6-bucket run. The third pass is a full pass, requiring us to go through the entire file. In this pass the remaining two runs are merged together to form a single 18-bucket run, and the file is now sorted. The complete merging requires 2.67 passes.

The concept of a pass

Concept of a pass has been informally introduced in the previous paragraph. A *pass* consists of reading all buckets, processing all buckets in some appropriate manner, and writing all buckets. Every pass leads to a refinement in the order of records, reducing the number of runs. For a file with N buckets, it is clear that the number of disk accesses for a pass is 2N. On the other hand, the cost of processing in a given pass depends upon the nature of processing required in that pass. In the above example, 1 pass is needed for creation of runs, and then 2.67 passes are needed to do 2-way merging until the file is completely sorted. Therefore, we can say that "the total cost of sorting is 3.67 passes." This figure, 3.67 passes, completely characterizes the number of disk accesses, which is $3.67 \times 2 \times 18 = 132$. The cost of processing per pass during creation of runs and during merging is different because the nature of processing is different. Note that during the partial run, the activity is same as a full pass, except that only two-thirds of the file is handled. The cost, measured at the rate of per bucket, is the same for partial and full passes. Therefore, the number of passes needed to sort a file is a good measure of performance of sorting, especially the disk accesses.

We have stated that we do not formally quantify cost of processing and disk accesses, but we offer some insights into how to reduce such costs. Because the number of disk accesses in a pass is constant (2N), a rule of thumb is to reduce the number of passes that are needed to sort a file. We hope that even if reducing number of passes increases the cost of processing for some passes, the overall cost of processing will still decrease.

m-way merging

As stated above, the input-output cost of merging only depends upon the number of passes. Therefore, a merge tree with a small height is favored, and so it is best to try to use m-way merging for the largest possible value of m. However, the cost of merging one bucket increases with the value of m. This is because for larger value of m, one needs to make more comparisons to choose the smallest key from m input streams. This increase in cost is duly offset by the fact that the number of passes reduce.

Example 3.3. In the previous example, we considered 2-way merging. Now let us assume that we have up to 4 buffers available for merging. As shown in Figure 3.3, we can have a pass of 3-way merging followed by a pass of 2-way merging. Merging can be accomplished in 2 passes, as opposed to 2.67 passes for 2-way merging. Even though the processing cost of a pass is greater for 3-way merging than that of 2-way merging, it is interesting to verify that the overall cost of processing is lower due to reduction in number of passes.

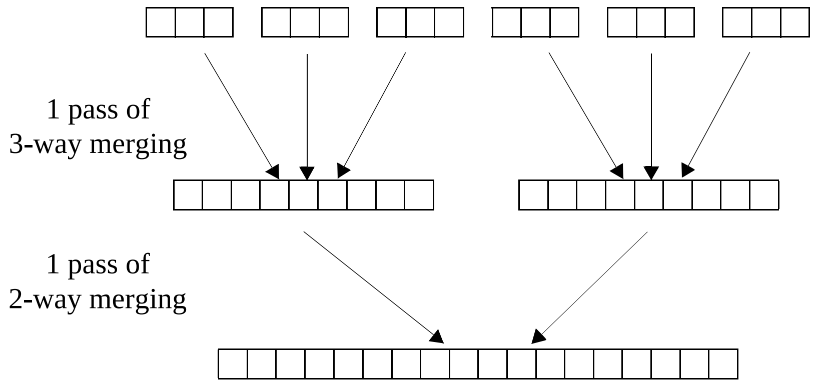


Figure 3.3 Merge tree for Example 3.3

So far we have introduced a basic elements for sorting a file. Considerable improvements can be made in performance of sorting by using some interesting ideas. These ideas are mentioned below, and explained in detail later.

- **Creating large runs.** We have already seen that with B buffers, we can create runs that consist of B buckets. By using a clever technique, we can create runs that are twice as long on average. Although, this does not reduce the cost of creating runs, but it yields fewer runs. The reduced number of runs helps in reducing the cost of merging because fewer passes are needed.
- **Fast m-way merging.** The second improvement is to make m-way merging faster. Ordinarily, choosing the smallest number in m input streams requires us to make $m-1$ comparisons. This can be reduced to $\log_2 m$ comparisons using a clever data structure called tree of losers. This would save processing time that is of critical importance

in sorting. It allows one to do m-way merging with a larger value of m. That reduces the number of passes needed during the merging phase.

• **Pipeline merging.** The third improvement is pipeline processing. Here, input, processing, and output are all done in parallel. Although, it does not save the processing time or the number of disk accesses, it does reduce the time between start and end of sorting by keeping the system dedicated and fully utilized to the task of sorting.

3.2 Creating large runs

The technique for creation of runs outlined in Section is rigid. It pays no attention to the real runs, the runs that are already present in a file. Instead, independently of the state of the file, it reads fixed length segments of records from the file, sorts them and certifies them as runs. Even if the file happens to be sorted to begin with, the fact that it is sorted is never realized.

It is not unreasonable to expect that the certified runs should contain the original real runs. Runs should be expanded and they should not be fragmented. It is natural to expect that the amount of expansion of runs will depend upon the available memory. Using the B buffers that are available to us, by using the technique in the previous section we can create runs that consist of B buckets. Now we give an interesting technique that is known to create runs consisting of 2B buckets on average for random data. In addition, the technique will guarantee that existing runs will not shrink in size.

In order to understand the main idea, we concentrate upon creation of a single run which we will term as the *current run*. Since we want to be able to expand the current run, we have to read more data from the input file. Since the available memory, consisting of B buffers, cannot expand, we must be willing to write a part of the current run in the output file. Since we are only allowed a single pass, the portion of the output run that has been written to the disk will not be read again. This leads to two observations. First, we should be conservative about what we write to the disk: smallest possible values belonging to the current run should be written. Second, the largest value that has been written into the current run becomes a threshold. If an input record has a smaller key value than the threshold, it is obvious that this record cannot be accommodated in the current run and should be deferred to a future run. Since we want everything to be finished in one pass, the record has to be stored in the memory. As we input and process more records, the records that cannot be incorporated in the current run will increase in number. This means that the memory available for composing the current run will decrease and sooner or later it will completely diminish. At this time we have written all the records in the current run and the memory is now filled with the record belonging to the next run. Now, we can start composing the next run by repeating the above process again.

Now we describe how the idea outlined above can be implemented. Heaps (priority queues with the smallest key at the top) are appropriate for such implementation. Recall from any data structures course, that a heap is viewed as a complete binary tree, but stored as an array. As shown in Figure 3.4, the B buffers that are given to us, are partitioned into four logical compartments described below:

• InBuff, a buffer for continual streaming of input of raw data, one bucket at a time.

• OutBuff, a buffer for continual streaming of output to the current run, one bucket at a time.

• Two heaps H_1 and H_2, such that $H_1 \cup H_2$ always occupy $B-2$ buffers. The heap H_1 consists of records that will be accommodated in the current run and H_2 will be used to store records that we are unable to incorporate in the current run. Initially, the heap H_2 is empty. On the other hand, the array H_1 is filled with $B-2$ buckets from the input file and it is then made into a heap. All records of heap H_1 will be included in the current run. The heap H_2 will gradually expand and H_1 shrink. Eventually, H_1 will become empty and H_2 will grow to occupy all $B-2$ buckets. At this point the contents of OutBuff are flushed to the current run. The current run is now completed and it is time to start the next run. This is done simply by interchanging the roles of the heaps H_1 and H_2.

The above description make it clear how the buffers are configured and it outlines the overall strategy. What remains to be seen is how records are transferred from InBuff to heap H_1, heap H_2, and OutBuff. Note that at any given moment we have $H_2 <$ OutBuff $< H_1$. This is meant to express that every key in H_2 is smaller than every key in OutBuff, and every key in OutBuff is smaller than every key in H_1.

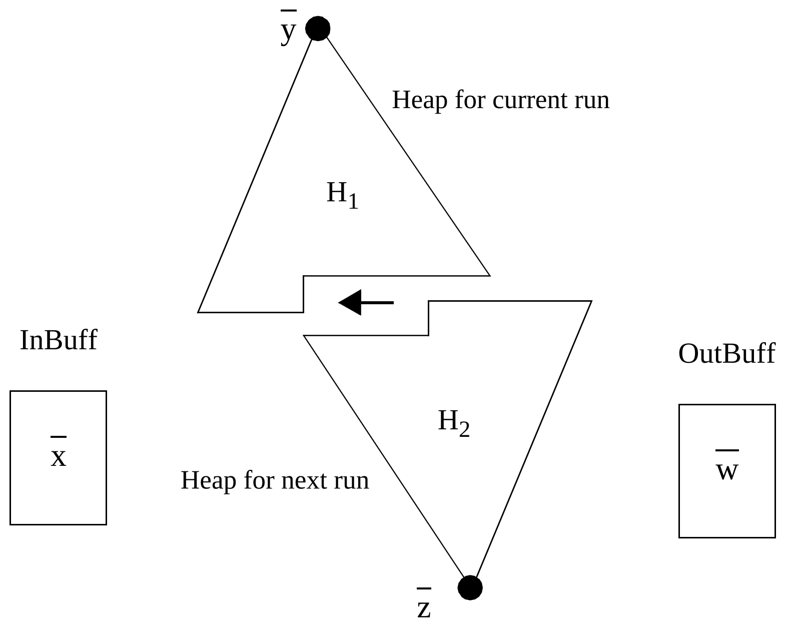


Figure 3.4 Organization of B buufers for creating large runs

Imagine that currently we are composing a run and some records in the run have already been output. As shown in Figure 3.4, suppose we are considering a record $\bar{x}$ in InBuff that is waiting to be processed. Suppose that $\bar{w}$ is the last record deposited to OutBuff. Then, $\bar{w}$ represents a threshold for determining what belongs to the current run and what does not. A record with the key value smaller than w will have to wait for the next run; otherwise it can be accommodated in the current run.

In other words, if $x < w$, the record $\bar{x}$ should be transferred to the heap H_2. In order to do this, heap H_2 has to be expanded by one element. Therefore, the heap H_1 has to be shrunk by one element. To accomplish this, the record $\bar{y}$ at the top of the heap H_1 is transferred to OutBuff. This creates a vacancy at the top of heap H_1. The vacancy is filled by transferring the last element of heap H_1 to the top of H_1. The heap H_1 is now redrawn. The vacated element is now claimed as the last element of heap H_2 and this is where the record $\bar{x}$ is stored. The heap H_2 is also redrawn to complete the process.

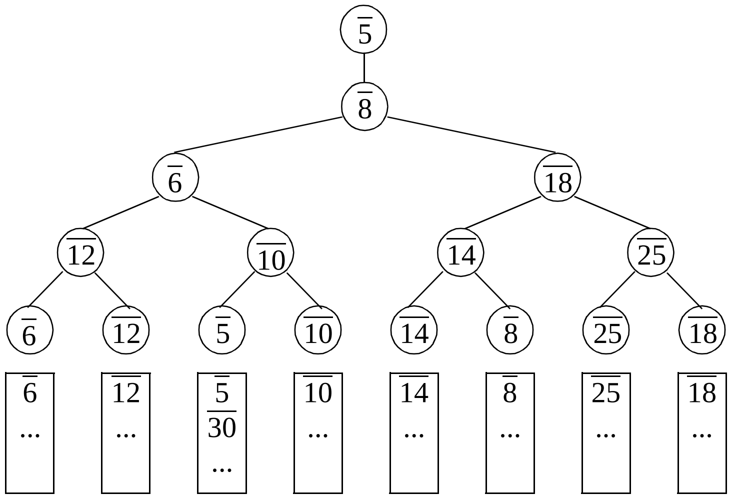
Now assume that $x \geq w$. Do we transfer it to heap H_1 or OutBuff? To decide this y, the key of the record $\bar{y}$ at the top of the heap H_1 will act as the threshold. We have two possibilities. Either $x \leq y$ or $x > y$. If $x \leq y$, then x is $\leq$ every element in heap H_1, and the record $\bar{x}$ is deposited in OutBuff. On the other hand, if $x > y$, then $\bar{x}$ must be absorbed in the heap H_1. Since we cannot expand H_1 in order to accommodate $\bar{x}$, we transfer the record $\bar{y}$ at the top of the heap H_1 to OutBuff. The vacancy is filled by $\bar{x}$ and the heap H_1 is redrawn to complete this process.

In order to implement the heaps, an array of B−2 buffers may be used. Each end of the array can be used for top of a heap. The heap elements are stored inward in the array in opposite directions. The bottoms of the two heaps are consecutive elements of the array. As one heap shrinks the vacancy is easily absorbed by the other heap. At the end of a pass, the roles of the two arrays are swapped.

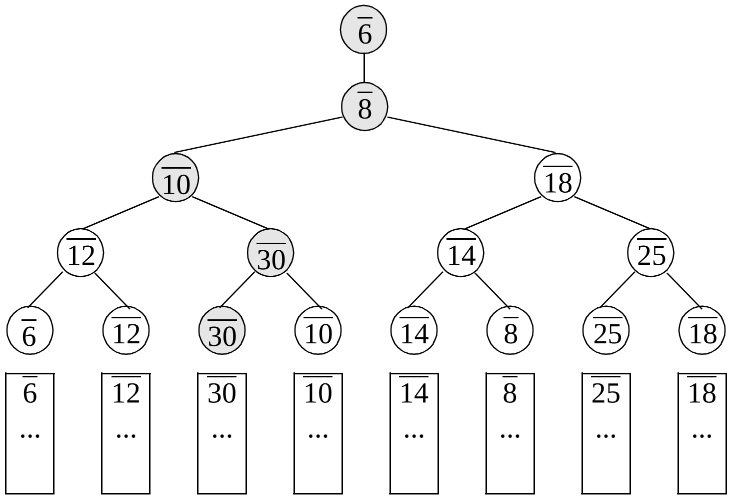
Because of the fact that the H_1 and H_2 occupy only B–2 buckets and not B buckets, the technique as described above will create runs that are on average 2(B−2) buckets long. (This fact is not trivial.) Further improvements can be made at the start of run creation so that the runs are 2B buckets long on average.

3.3 Fast m-way merging

In m-way merging there are m input streams and one output stream. In such merging m+1 buffers are required, one each for m input streams and one for the output stream. One has to continually determine the smallest of m keys of the records at the front of the input stream and transfer the record to the output stream. Ordinarily, this would require m−1 comparisons. However, while making m−1 comparisons, we not only determine the smallest key, but also come across some information about the relative magnitude among other keys. The *tree of losers* helps us "remember" such information; the next smallest of m keys can be determined in only $\log_2 k$ comparisons rather than m−1 comparisons. The tree of losers is a binary tree. Its leaves are the records waiting to be processed in the m input streams.



(a) The first tournament



(b) Subsequent tournaments

Figure 3.5 m-way merging with tree of loosers

To draw the tree, the keys of pairs of records are compared. We refer to such comparison as a *match* between the two records. The record with the smaller key is said to *win* the match. Figure 3.5(a) shows 8-way merging with tree of losers. The tree corresponds to a tournament with the objective of determining the overall winner. The leaves of the tree are the eight records, $\overline{6}$, $\overline{12}$, $\overline{5}$, $\overline{10}$, $\overline{14}$, $\overline{8}$, $\overline{25}$, and $\overline{18}$, from the front of each input stream. The matches are first played between the four pairs of records. The four losers, $\overline{12}$, $\overline{10}$, $\overline{14}$, and $\overline{25}$., of these matches are installed as parents. Now the matches are played among the four winners $\overline{6}$, $\overline{5}$, $\overline{8}$, and $\overline{18}$. The two losers, $\overline{6}$ and $\overline{18}$, are installed as the parents at the next level. In any step, the match is played between the pairs of remaining losers up to that point. So, the last match is played between the two surviving winners $\overline{5}$ and $\overline{8}$. The looser, $\overline{8}$, is installed as the root of the tree. We are left with the overall winner, $\overline{5}$, that is now installed at the helm of the tree. The helm of the tree is simply a spot in the output buffer.

Now, the tournament must be played again to determine the next winner. For this we do not have to start from scratch. The tournament is started at the leaf level, where the winner originated from, and played only along the path going to the root of the tree. For example, in Figure 3.5 the winner, $\overline{5}$, comes from the third input stream. The next record in that input stream is $\overline{30}$. The tournament will now be among $\overline{30}$, $\overline{10}$, $\overline{6}$, and $\overline{8}$. These are the records that lie on the path from the leaf $\overline{30}$ to the root $\overline{8}$ of the tree. The result of the tournament is shown in Figure 3.5(b). The nodes in the path in question are shaded in Figures 3.5(b). In this tournament the looser is left behind and the winner advances. The overall winner, $\overline{6}$ in this case, is installed at the helm of the tree.

The tree of losers reduces the processing time dramatically. If more memory is available, a larger value of m can be chosen. The memory is needed for the m+1 buffers. A small amount of additional memory is needed for the tree. The tree is a complete binary tree (ignoring the helm) that can be stored as an array of nodes. The child and parent pointers can be computed and need not be stored explicitly in these nodes. Moreover, it is not necessary to store records in these nodes, the pointers to the records suffice. In our example we implicitly assumed that m in m-way merging is a power of 2. Some thought is needed if the value of m is arbitrary.

3.4 Pipeline merging: minimizing elapsed time

In this section we will consider how merging can be speed up. We will assume that the system permits disk access and processing can be done in parallel. At this point we will pursue an interesting case when a system has two parallel disks and the system is dedicated to our task of merging. We will assume that for m-way merging, m has been chosen such that the time to process one bucket is same as its access time. Thus we can do merging in a pipeline by performing the following three operations in parallel: read a bucket, process the bucket, and write the bucket. This gives rise to pipeline as shown in Figure 3.6. Therefore, we define a *pipeline operation* to be an operation of the form

$\langle$read α_1, process α_2, write $\alpha_3\rangle$

Note that the operation “process α_2” refers to merging records from m input buffers to fill one output buffer α_2. Thus in this operation m+1 buffers are involved in the processing. A buffer can be involved in only one operation at a time: it can be read, it can be processed, or it can be written. In other words the cost of a pipeline operation is the sum of the costs of its three component operations. However, a pipeline operation allows all three component operations to be performed in parallel, thereby minimizing the elapsed time.

A buffer cannot be involved in I/O and processing at the same time. Therefore, support of uninterrupted m-way merging would require 2k+2 buffers. The behavior of the output stream is simple, and out of the 2k+2 buffers, 2 buffers can be permanently coupled to the output stream. The buffer configuration for the input buffers is interesting and will be discussed next.

A simple strategy, termed *static buffer configuration*, is to couple 2 input buffers permanently to each input stream. Example 3.4 shows that the static buffer configuration does not guarantee uninterrupted pipeline.

The alternative is to use a *dynamic buffer configuration*. This configuration maintains the 2k input buffers in a pool. A buffer is coupled dynamically to a strategically chosen input stream. It might happen that at a given time some input streams have more buffers coupled to them than others, but it can be shown that the dynamic buffer configuration guarantees uninterrupted pipeline merging. (See Exercise 3.10.)

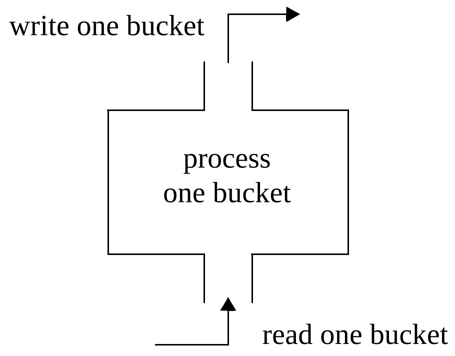


Figure 3.6 Pipeline merging

Example 3.4. This example we show that static buffer configuration does not guarantee uninterrupted pipeline merging. Suppose that for 2-way merging (m =2) we are given 2k+2 = 6 buffers.

Suppose that the 6 buffers are numbered α_1, α_2, α_3, α_4, β_1, and β_2. For 2-way merging we need 2 input streams and 1 output stream. As shown in Figure 3.7, we couple buffers α_1 and α_2 to input stream 1, α_3 and α_4 to input stream 2, and β_1 and β_2 to the output stream. A bucket consists of 2 records. Figure 3.7 also shows 2 runs given to us.

The merging is shown in Figure 3.8. We couple the first run to the first input stream and second to the second input stream. At this point there is nothing in the pipeline. To get the pipeline going, we first read buffers α_1 and α_3. The result at the end of these operations is shown in Figure 3.8(a). Note that the buffers which are involved in read or write operations are shaded. A shaded buffer is not available for processing. Next we read buffer α_2 and merge into buffer β_1. Figure 3.8(b) shows the results at the end of this step.

At this point the pipeline is setup: we can read a bucket, we can process a bucket, and we can write a bucket. The buffers α_1, α_3, α_2, and α_4 are read in a round robin manner. Buffers β_1 and β_2 are likewise written in a round robin manner. The buffer that is not being written is where the processing (merging) is taking place. At this point α_1, α_3, and α_2 have been read, and β_1 has been processed. Next we attempt to execute the following pipeline operations:

$\langle$read α_4, process β_2, write $\beta_1\rangle$
$\langle$read α_1, process β_1, write $\beta_2\rangle$
$\langle$read α_3, process β_2, write $\beta_1\rangle$
$\langle$read α_2, process β_1, write $\beta_2\rangle$

The configuration after each of the first three pipeline operations is shown in parts (c), (d), and (e) of Figure 3.8, respectively. As shown in Figure 3.8(f), the fourth pipeline operation cannot be executed successfully because we have run out of data prematurely in the buffers for the first input stream, even though there is data left in that input stream on the disk. Therefore, the processing in this example has to be suspended for lack of input.

3.5 Summary

In this chapter we introduced external sorting as a two phase process, creation of runs followed by merging. After having given the basis idea, we discussed how to create larger runs and do faster merging. Then we introduced pipeline merging that allows input, processing, and output to be done in parallel. Although, pipeline merging does not reduce the number of disk accesses or the processing time, but helps reduce the elapsed time, i.e., the time from the start to the end of merging. The processing time in sorting could be substantial and it has to be accounted for. We have not given any analytic models to estimate the cost of sorting a file.

We end this chapter with the remark that sometimes the number of disk accesses required for sorting N buckets

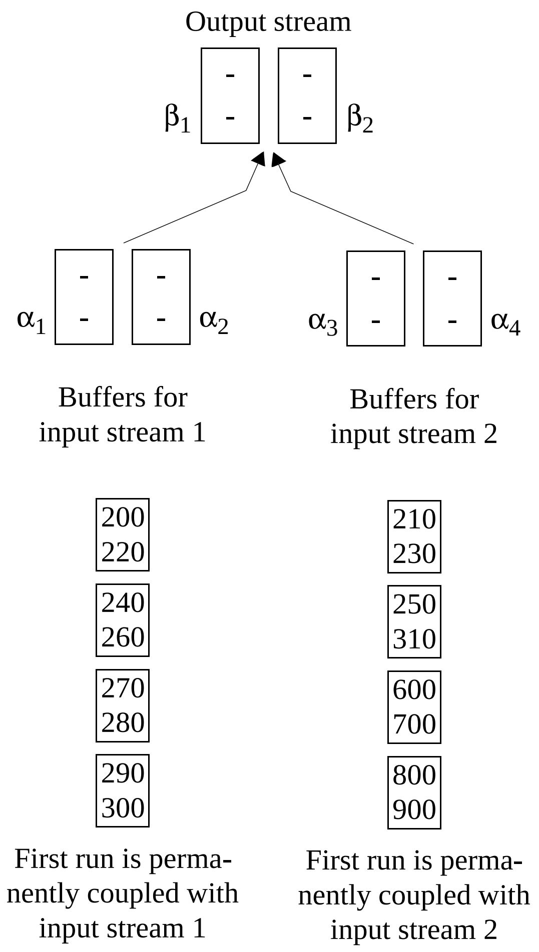

Output stream
β1
β2
-
-
α1
α2
α3
α4
Buffers for input stream 1
Buffers for input stream 2
200
220
240
260
270
280
290
300
210
230
250
310
600
700
800
900
First run is permanently coupled with input stream 1
First run is permanently coupled with input stream 2

Figure 3.7 Merging setup with static buffer configuration

with B buffers is informally estimated as $2N\log_{B-1}N$. In practice we can do much better than that.

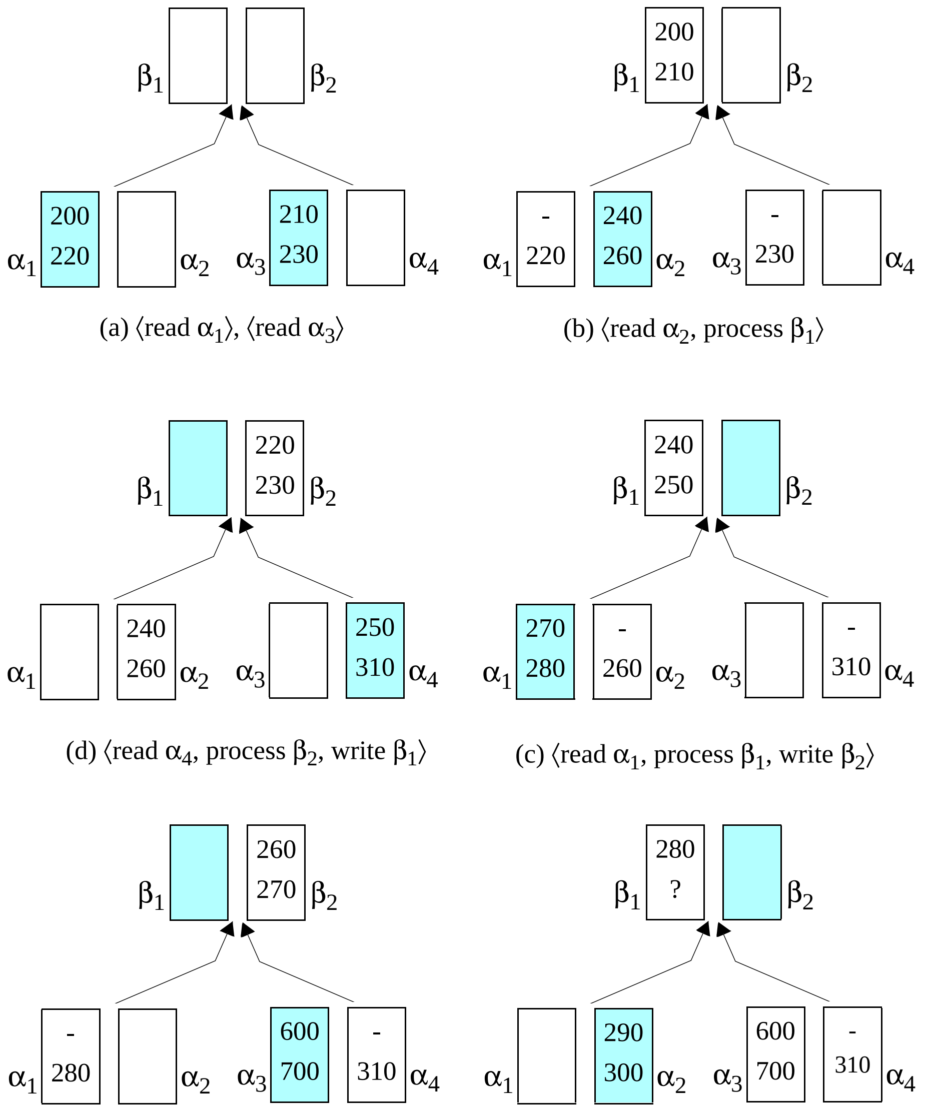
Exercises

3.1. We are given 16807 runs of 5 buckets each, and 8 buffers. To sort the file, give a strategy for merging and calculate the I/O cost of merging.

3.2. We are given a file with 2560 records. Every bucket contains 10 records. For sorting the file we are given 5 buffers. Assuming bucket access time of 75 ms, prove that the I/O cost of sorting the file is 192 seconds.

3.3. We are given 2401 runs of 5 buckets each, and 8 buffers. Give a strategy and calculate the I/O cost of merging to sort the file.

3.4. Suppose a file consists of 1024 runs of 10 bucket each. For merging we have 6 buffers. Suppose bucket access time is 50 ms. Consider the following three options for merging and give an appropriate answer.
(a) Estimate the cost of 5-way merging. To simplify your calculations pad the number of runs being merged so that it is divisible by 5. Do not pad the number of runs in the last pass. The number of runs will follow the following sequence: 1024 changed to 1025, 1025/5 = 205, 205/5 = 41 changed to 45, 45/5 = 9 changed to 10. Now do a 2-way merging.
(b) Estimate the cost of 2-way merging using 2-bucket cascades. Assume that the bucket transfer time is 10 ms.


β_1 β_2
α_1 200 220 α_2 α_3 210 230 α_4
β_1 200 210 β_2
α_1 - 220 240 260 α_2 α_3 - 230 α_4
β_1 220 230 β_2
α_1 240 260 α_2 α_3 250 310 α_4
β_1 240 250 β_2
α_1 270 280 - 260 α_2 α_3 - 310 α_4
β_1 260 270 β_2
α_1 - 280 α_2 α_3 600 700 - 310 α_4
β_1 280 ? β_2
α_1 290 300 α_2 α_3 600 700 - 310 α_4

(a) ⟨read α_1⟩, ⟨read α_3⟩ (b) ⟨read α_2, process β_1⟩

(d) ⟨read α_4, process β_2, write β_1⟩ (c) ⟨read α_1, process β_1, write β_2⟩

(e) ⟨read α_3, process β_2, write β_1⟩ (f) ⟨read α_2, process β_1, write β_2⟩

(Shadowed buffers are involved in I/O and not available for processing.)

Figure 3.8 Merging with static buffer configuration

(b) Perform 2-way pipeline merging. Note that the number of disk accesses in this case will be the same as in part (b). In this case estimate the elapsed time. For simplicity ignore the slack in the pipeline at the beginning and the end for every batch of five runs. (Hint: the elapsed time is simply estimated from the answer of part (b).)

3.5. Given a single disk with 15 ms of seek time, 10 ms of latency, and 2 ms of bucket transfer time. A file with 625 runs of 10 buckets each is stored on the disk. Assume that each run is stored as a cascade. We are given 20 buffers which are to be configured as follows: 5 cascades of 2 buckets each, for 5 input streams, and, 1 cascade of 10 buckets forming a single output stream. Calculate the cost of I/O in sorting the file using merging.

3.6. We are given a single disk for sorting with seek = 20 ms, latency = 10 ms, bucket transfer time = 3 ms. To save access time, we use cascades of multiple buckets for I/O. We assume that all runs are stored as cascades. A file with

625 runs of 10 buckets each, is stored on the disk. For 5-way merging, we have 15 buffers. Each input stream consists of 2 buffers, and the output stream consists of 5 buffers. Calculate the time required to complete the sorting.

3.7. Computer systems sometimes crash because of software failures? How would you modify our strategy for sorting to allow us to salvage as much work before a crash as possible.

3.8. How does the technique described in Section 3.2 handle a file which is already sorted? How about a file that is sorted in the reverse order?

3.9. The technique described in Section 3.2 guarantees that the length of a run will be at least $B-2$ buckets. Improve it so that one is guaranteed that the length of every run is at least B buckets.

3.10. For dynamic buffer configuration for pipeline merging we have stated that an input buffer is coupled to a *strategically* chosen input stream. (a) What is the strategy? (b) Verify that this strategy works for the file of Figure 3.7 (c) Prove that the strategy works in general.

3.11. Suppose N is a exponential power of $B-1$. Show that the number of disk accesses for sorting N buckets with B buffers is no more than $2N\log_{B-1}N$.