**Data Structures**

**External Sorting**

**Algorithms**

**Class Notes**

**© Copyright July 2000, 2010, 2012**

**by David Burris, Ph.D., CCP, CSP**

**Permission to use this material for personnel educational purposes by students currently registered in my classes is granted. All other rights reserved.**

**Contents**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | External Sorting Algorithms - Merging | [3](#Merging) |  |
|  | 2 | Sample Sorting Problem for Algorithms | [4](#Problem) |  |
|  | 3 | Two-Way Balanced Sort Merge | [5](#Balanced2Way) |  |
|  | 4 | Polyphase Sort Merge with Distribution Passes | [6](#PolyphaseDistribution) |  |
|  | 5 | Polyphase Sort Based on Fibonacci Numbers | [7](#PolyphaseFib) |  |
|  | 6 | Replacement by Natural Selection | [8](#NaturalSelection) |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

External Sorting Algorithms

Assume that the number of records that must be sorted exceeds the amount of available main memory. When this condition occurs, we must resort to an external sorting algorithm using auxiliary storage. Most auxiliary storage sorting algorithms are based on the principle of merging strings of sorted keys (records). As an example consider the following, which assumes there is only enough main memory to hold two keys (records) at a time in main memory in addition to the code of the program itself.

The general strategy is to read the next available record into main memory from each sorted string. To sort in ascending order, we compare the two keys in main memory (records) and write the smallest key to an auxiliary file. We replace the key that was selected for output with the next available key (in sorted order) from the string in which it appeared. We repeat the comparison, write, and replacement process until out of keys on both files. If one input file empties before the other, then we simply copy the other input files keys to the output file. Note the files do not have to be of the same length. After one merge pass we are guaranteed that the original two sorted strings now constituted a single sorted string. Initially key 16 is selected for output after keys 16 and 18 are read from strings 1 and 2 to fill the main memory buffer.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Main Memory |  | In |  |  |  |  |  |  |  |
|  |  | ~~16~~ | ← | ~~16~~ | 22 | 34 | 36 |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  | ← |  |  |  |  |  |  |  |  |
|  |  | 18 |  | ~~18~~ | 26 | 30 |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 |  |  |  |  |  |  | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 22 |  | ~~16~~ | ~~22~~ | 34 | 36 |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | ~~18~~ | ← | ~~18~~ | 26 | 30 |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 | 18 |  |  |  |  |  | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | ~~22~~ | ← | ~~16~~ | ~~22~~ | 34 | 36 |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 26 |  | ~~18~~ | ~~26~~ | 30 |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 | 18 | 22 |  |  |  |  | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 34 |  | ~~16~~ | ~~22~~ | ~~34~~ | 36 |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | ~~26~~ | ← | ~~18~~ | ~~26~~ | 30 |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 | 18 | 22 | 26 |  |  |  | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 34 |  | ~~16~~ | ~~22~~ | ~~34~~ | 36 |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | ~~30~~ |  | ~~18~~ | ~~26~~ | ~~30~~ |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 | 18 | 22 | 26 | 30 |  |  | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | ~~34~~ | ← | ~~16~~ | ~~22~~ | ~~34~~ | 36 |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | empty |  | ~~18~~ | ~~26~~ | ~~30~~ |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 | 18 | 22 | 26 | 30 | 34 |  | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | ~~36~~ |  | ~~16~~ | ~~22~~ | ~~34~~ | ~~36~~ |  |  |  | String 1 |
|  | ← |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | empty |  | ~~18~~ | ~~26~~ | ~~30~~ |  |  |  |  | String 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 16 | 18 | 22 | 26 | 30 | 34 | 36 | output |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |

**Sort Problem:**

Consider the problem of sorting 7000 keys (records) into ascending order. We assume that there is only enough main memory to hold 1000 records with keys at a time in addition to the code for the sort algorithm. Note that we can read 1000 records into main memory at a time and generate 7 strings of 1000 records each in sorted order, denoted S1, S2, S3, S4, S5, S6, and S7. The main memory sort algorithm used to generate the strings is of no importance to the merge process. Consider using Heap Sort if you do not know the characteristics of the data. The strings can be combined using merging to produce the final sort sequence.

Note that given any two strings of 1000 sorted records, they will merge to form a string of 2000 sorted records. Two strings of 2000 records will merge to form one string of 4000 records. Finally, a string of 4000 records merged with a string of 3000 records will form a string of 7000 records. Note that a null string merged with any string S is just the string S.

**Balanced 2-Way Sort Merge**

One of the most famous ways to accomplish sorting very large files in auxiliary storage is called a 2-way balanced sort merge. We assume 4 auxiliary devices (or four files). Initially, as many records at a time as possible are read into main memory and sorted into strings. These strings are written alternately to the first two devices. Traditionally this is referred to as the sort pass. After the sort pass, the strings for the sample problem would appear on devices 1 and 2 as shown. Devices 3 and 4 would be empty. Following tradition, we assume the devices are tape drives.

**Sort Pass:**

Tape 1 = S1, S3, S5, S7

# Tape 2 = S2, S4, S6

Tape 3 =

Tape 4 =

We now begin the merge passes. The first string on each input device is merged forming a single string consisting of the two input strings in sorted order and place on the output device. This process is repeated alternating the strings on the two output devices. After the merge pass the original input devices are empty. If there was an equal number of original strings all of the same length, then the number of strings is halved and the length of the new strings are twice the size of the original strings. The following shows the results after the first merge pass. The notation S12 means the result of merging S1 and S2.

**1st Merge Pass**

Tape 1 =

# Tape 2 =

Tape 3 = S12, S56

Tape 4 = S34, S7

**2nd Merge Pass**

Tape 1 = S1234

# Tape 2 = S567

Tape 3 =

Tape 4 =

**3rd Merge Pass**

Tape 1 =

# Tape 2 =

Tape 3 =S1234567 and the sort is complete.

Tape 4 =

The number of merge passes for a 2-way balanced sort merge is:

Merge passes = ⎡ log2 S⎤ where S is the number of initial strings.

For example, if the number of initial strings S = 7, then the number of merge passes = ⎡ log2 7⎤ = ⎡ 2.65 ⎤ = 3. The notation ⎡ X ⎤ is the ceiling operator or upper bound of X, e.g., ⎡ 2.003 ⎤ = 3.

Note that to a large extent, the time required to sort is proportional to the number of merge passes. CPU time is typically measured in nano or pico seconds while disk I/O time is measured in milliseconds. Hence the time for the sort is dependent on the number of I/O operations. The CPU processing time is negligible. The merge will go faster it you use more devices. For example, a 3-way balanced sort merge with 6 devices merges three input strings at a time to three output files reducing the total number of strings after each pass by a factor of 3. The problem is the number of devices required to sort versus diminishing returns in performance (time to sort).

## Polyphase Sort Merge

Physical devices (rather than files) required to hold strings during the merge process are very expensive. Hence, a primary motivation is to reduce the number of devices even if time is sacrificed (sort time increases). One method to accomplish this with three devices (files) follows:

Step 1: Distribute the initial runs (strings) alternately on the first two devices during the sort phase, say tape drives T1 and T2.

Step 2: Loop

Merge strings from T1 and T2 onto T3. Stop if T3 consists of a single string, the original data is sorted (exit the loop).

Copy strings from T3 alternately onto drives T1 and T2.

### End Loop

Using the example of 7000 records above sorted into 7 strings (runs) of length 1000 we have after the sort phase the following configuration:

**At end of sort pass:**

Tape 1 = S1, S3, S5, S7

# Tape 2 = S2, S4, S6

Tape 3 =

**At end of 1st merge pass:**

Tape 1 =

# Tape 2 =

Tape 3 = S12, S34, S56, S7

**At end of 1st distribution pass:**

Tape 1 = S12, S56

# Tape 2 = S34, S7

Tape 3 =

**At end of 2nd merge pass:**

Tape 1 =

# Tape 2 =

Tape 3 = S1234, S567

**At end of 2nd distribution pass:**

Tape 1 = S1234

# Tape 2 = S567

Tape 3 =

**At end of 3rd merge pass:**

Tape 1 =

# Tape 2 =

Tape 3 = S1234567

Note that the number of passes over the data = 2 \* ⎡ log2 S⎤ since there is a distribution pass after each merge pass. While this is approximately twice as slow as a 2-way balanced sort merge, it saves the need for one device/temporary file.

## Polyphase Sort Based on Fibonacci Numbers

We desire to speed up the sort process but not introduce additional devices. In general this means that we must reduce the number of merge passes or reduce the number of distribution passes. Recall the Fibonacci Series defined as:

f0 = f1 = 1.

f i+2 = f i+1 + fi for all i > 1.

Hence a few terms include:

f0 = 1, f1 = 1, f2 = 2, f3 = 3, f4 = 5, f5 = 8, f6 = 13, and f7 = 21.

Assume that the number of strings generated by the sort phase is a perfect Fibonacci number. Using three devices, distribute the strings on the first two drives during the sort phase according to the two preceding numbers in the Fibonacci series. The following assumes 13 strings were generated during the sort phase. Initially we place 8 strings on device 1 and 5 strings on device 2 as 8 + 5 = 13. Note that a string may contain thousands of records.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Tape 1** |  | **Tape 2** |  | **Tape 3** |  |  |
|  | 1, 1, 1, 1, 1, 1, 1, 1 |  | 1, 1, 1, 1, 1 |  |  |  | Sort |
|  | 1, 1, 1, |  |  |  | 2, 2, 2, 2, 2 |  | Merge 1 |
|  |  |  | 3, 3, 3 |  | 2, 2 |  | Merge 2 |
|  | 5, 5 |  | 3 |  |  |  | Merge 3 |
|  | 5 |  |  |  | 8 |  | Merge4 |
|  |  |  | 13 |  |  |  | Merge 5 |
|  |  |  |  |  |  |  |  |

We begin by merging 5 strings from tape 1 with 5 strings from tape 2 onto tape 3 producing strings twice the size (2) of the original strings. Note that tape 2 is now empty. In the second merge pass we merge three strings of length 1 on tape 1 with three strings of length 2 on tape two onto tape 3 freeing tape 1. In merge pass three, we merge two strings of length 3 from tape 2 with two strings of length 2 from tape 3 onto tape 1 freeing tape 3. In the 4th merge pass, we merge one string of length 5 from tape 1 with one string of length 3 on tape 2 to tape 3 freeing tape 2. Finally, in merge pass five, we merge one string of length 5 on tape 1 with one string of length 8 on tape 3 to tape 2.

Note that the number of merge passes, 5, is more than the number of merge passes in a 2-way balance sort merge but that not every string is processed in every merge pass. Hence the polyphase sort not only uses fewer devices (files) but also potentially runs faster. Remember that in general, time is proportional to the number of passes over the data, i.e., I/O operations.

The 3 device Polyphase Sort may be extended using generalized Fibonacci Numbers. This is seldom worthwhile as the cost of the additional devices exceeds the improvement in performance. The formula for the number of passes over the data where “e” is the natural log base and S is the number of strings produced in the sort phase is:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Number devices:** | **Number of passes:** |  |
|  | 3 | 1.504 \* log e S + 0.992 |  |
|  | 4 | 1.015 \* log e S + 0.965 |  |

Note that for 21 initial strings, we would place 13 strings on the first device and 8 strings on the second device. If the number of strings is not a perfect Fibonacci number, then we assume dummy strings to reach the next largest Fibonacci number. The placement of the dummy strings on the devices is important since merging dummy strings does not consume time.

## Replacement by Natural Selection

Note that merging does not require strings be of equal length. Furthermore, longer initial strings imply fewer merge passes, hence a faster sort. Most internal memory sorting algorithms are limited to producing strings of fixed size limited to the number of records that will fit in main memory. Assume that we know that the data has a lot of natural order with natural strings frequently exceeding the amount of available main memory. Replacement by Natural Selection using “P” memory locations to hold records produces strings that average 2P in length.

As an example of Replacement by Natural Selection, assume the following sequence of numbers in an input file and that there are only 3 available locations in main memory to hold records in addition to the code for the algorithm. We wish to produce strings in ascending key order. The general approach is to read sufficient records from the input to fill the main memory buffer, i.e., 3 locations in this example. Now output the smallest value in memory to a file replacing it with the next value from the input file. Continue writing the smallest value in the buffer that is larger than the last value output and replacing it with the next value from the input file until the smallest value left in main memory is no longer larger than the previous value output. This marks the end of a string. Using the data in the buffer start a new output string. When the input is exhausted, write the remaining data from the buffer to the output file to complete the final string.

The following shows the keys 56, 18, and 22 from the input file being used in step 1 to fill the 3 key buffer in memory. The smallest key, 18, is then written to the output and replaced by the key 37 from the input. In the second step the key 22 is written to the output and replaced by the key 45 from the input. Note that the first string is completed in step 6 when 72 is written to the output. Many applications write a special sentinel to the output (like ∞)to indicate the end of a string. The double horizontal bar at the end of step 6 is used to indicate the start of the second string. Note that the strings are of length 6 and 5 while the main memory buffer was only of length 3.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Input |  |  | **56** | **18** | **22** | **37** | **45** | **12** | **72** | **10** | **35** | **47** | **23** |
|  | ↓ |  | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Buffer |  |  | **56** | **56** | **56** | **56** | **~~56~~** | **10** | **~~10~~** | **47** | **47** | **47** | **~~47~~** |
|  |  |  | **~~18~~** | **37** | **~~37~~** | **12** | **12** | **12** | **12** | **~~12~~** | **~~23~~** |  |  |
|  |  |  | **22** | **~~22~~** | **45** | **~~45~~** | **72** | **~~72~~** | **35** | **35** | **35** | **~~35~~** |  |
|  |  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Output |  |  | **18** | **22** | **37** | **45** | **56** | **72** | **10** | **12** | **23** | **35** | **47** |
| Strings |  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |