JNTUK396.BLOGSPOT.COM

dmg

ADS

Unit - 1

Jntuk396.blogspot.com

External sorting

Lab : ..... Expt : ..... Date : .....

Introduction

→ The term sorting means arranging.

→ The elements of an array, that they are placed in some relevant order which may be either ascending or descending.

→ There are two types of sorting

① Internal sorting

② External sorting.

① internal sorting:-

→ Internal sorting is related with the ordering of elements present in a file stored in computer's memory.

→ few of the internal sorting techniques widely used are bubble sort, insertion sort, selection sort, heap sort, quick sort, merge sort and radix sort.

→ The internal sorting techniques are inefficient for sorting these huge files as they require the entire file to be present in the main memory of the computer, which is practically impossible. So, the need for external sorting methods arises to sort large files.

② External sorting

→ External sorting is applied when there is voluminous data to sorted that cannot fit in the memory.

→ Because of their large volumes, the files are stored in external storage device like magnetic tapes, magnetic disks, etc.

Where Quality matters...

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

internal sorting techniques. Each of the sorted block of records is called a *run*. Now, the file is viewed as a collection of runs. The runs are written on to the external storage devices as and when required. The most popular method for sorting files on external storage devices is *external merge sort* which is more efficient in gathering and processing the runs stored on external devices.

.2 k–WAY MERGE SORT

External merge sort is performed in two phases. The first phase involves the run generation and the second phase involves the merging of runs to form a larger run. This run generation is repeated and merging is continued till a single run is generated with the sorted file as its outcome. If k runs are merged at a time, the external merge sort is known as a *k–way merge sort*. To understand the k–way merge sort better, first let us first look at the 2–way merge sort with an example.

1.2.1 2-Way Merge Sort

The k–way merge sort where $k = 2$ is a 2–way merge sort. In 2–way merge sort, 2 runs are merged at a time to generate a single run twice as long. The merging process is repeated until a single run is generated.

Technique

For easy understanding, consider that there are 6000 records to be sorted and the internal memory capacity is 500 records. Let $R_{i\ j}$ represent the jth run in the ith pass. The generated runs in the first pass are $R_{1\ 1}$ to $R_{1\ 12}$. In the first pass, $R_{1\ 1}$ and $R_{1\ 2}$ are merged resulting in run $R_{2\ 1}$ which consists of the sorted list of first 1000 records. The next two runs $R_{1\ 3}$ and $R_{1\ 4}$ are merged resulting in $R_{2\ 2}$. Likewise, four other runs will be merged in the second pass resulting in runs $R_{2\ 1}$ to $R_{2\ 6}$. Similarly, in the third pass, $R_{2\ 1}$ and $R_{2\ 2}$ are merged to form $R_{3\ 1}$. Likewise, two other runs are generated resulting in runs $R_{3\ 1}$ to $R_{3\ 3}$. In the fourth pass, $R_{3\ 1}$ and $R_{3\ 2}$ are merged to form run $R_{4\ 1}$. The last run $R_{3\ 3}$ will be taken as it is to $R_{4\ 2}$. In fifth pass, $R_{4\ 1}$ and $R_{4\ 2}$ runs are merged to form run $R_{5\ 1}$, the final sorted file. The run generation using 2–way merge sort is shown in Fig. 1.1.

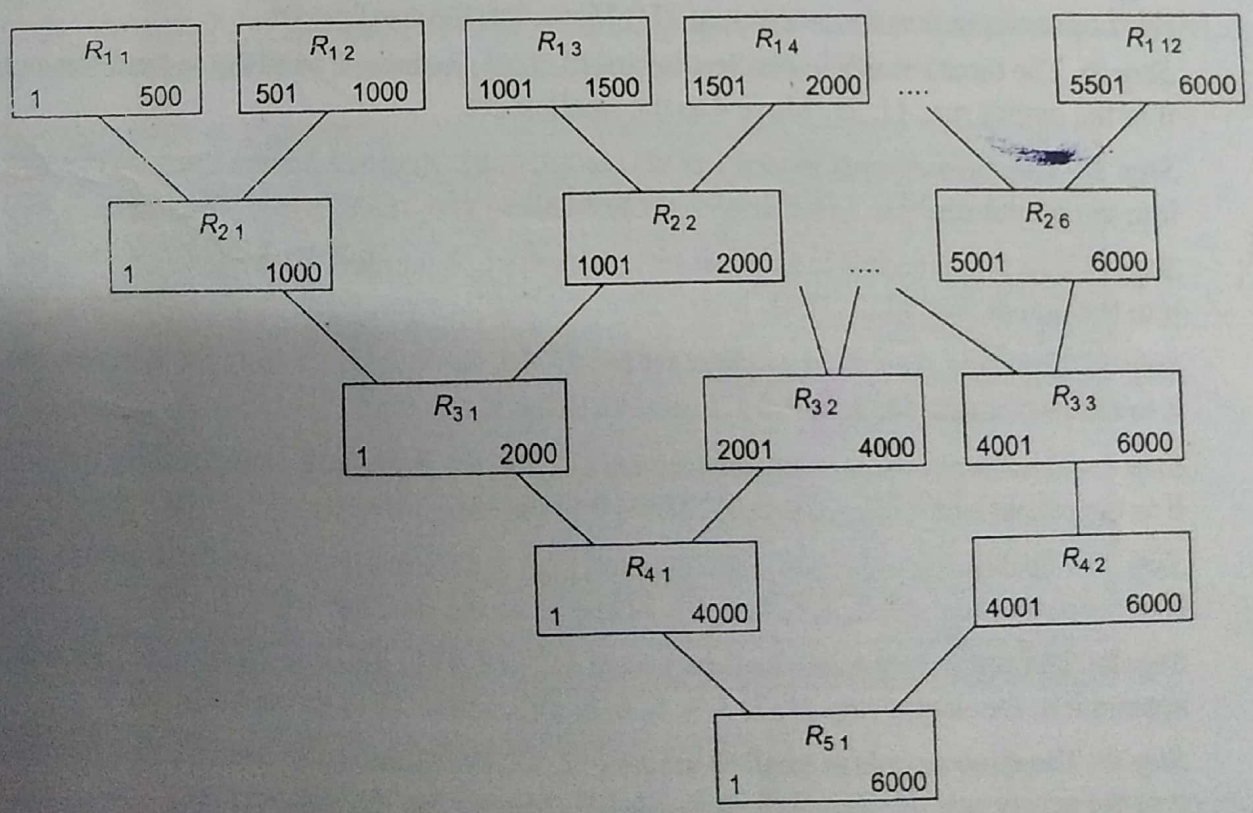


Figure 1.1 2–way merge sort

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

Example 1.1 Consider 6000 records are available on a disk which are to be sorted. In the internal memory of the computer, only 500 records can be resided. The block size of the disk is 100 records. Sort the file using 2–way merge sort.

Solution The steps involved in sorting the file are as follows:

Step 1 Read five blocks of data, i.e., totally 500 records at a time from the file residing on the disk to internal memory. Sort these blocks using any internal sorting technique to generate 12 runs (runs = 6000 records/500 records). These runs are written back on to the disk after sorting.

Step 2 Merge 2 runs at a time to generate a new run in the next pass with size twice as long. Until all runs in the pass are processed.

Step 3 Repeat Step 2 until a single run is generated.

Step 4 EXIT

1.2.2 3–Way Merge Sort

The k–way merge sort where $k = 3$ is a 3–way merge sort. In 3–way merge sort, 3 runs are merged at a time to generate a single run thrice as long. The merging process is repeated until a single run is generated. Considering the same Example 1.1 for 3–way merge sort yields the runs as shown in Fig. 1.2.

In the first pass, $R_{1\,1}$ to $R_{1\,3}$ are merged resulting in run $R_{2\,1}$, which consists of the sorted list of first 1500 records. The next three runs $R_{1\,4}$, $R_{1\,5}$, $R_{1\,6}$ are merged resulting in $R_{2\,2}$. Likewise, four runs will be emerging in the second pass, i.e., $R_{2\,1}$ to $R_{2\,4}$. Similarly, in the third pass, $R_{2\,1}$ to $R_{2\,3}$ are merged to form $R_{3\,1}$. The last run $R_{2\,4}$ will be taken as it is to $R_{3\,2}$. In fourth pass, $R_{3\,1}$ and $R_{3\,2}$ runs are merged to form run $R_{4\,1}$, the sorted output.

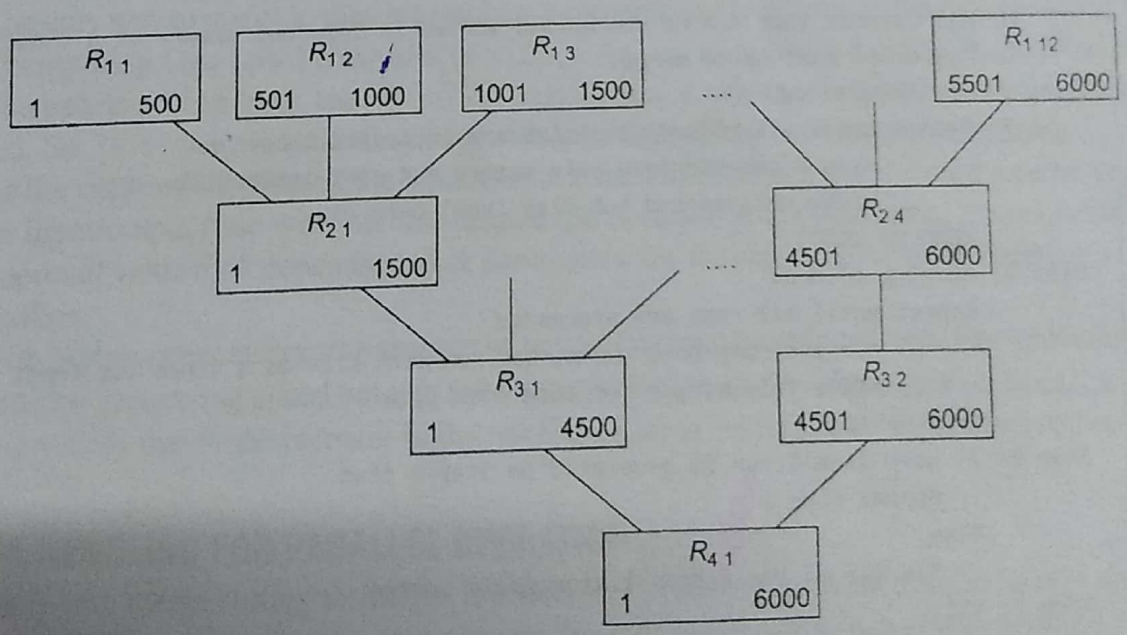


Figure 1.2 3–way merge sort

1.2.3 4–Way Merge Sort

The k–way merge sort where $k = 4$ is a 4–way merge sort. In 4–Way merge sort, 4 runs are merged at a time to generate a single run four times as long. The process is repeated until a single run is generated. Considering again example 1 for 4–way merge sort, yields the runs as shown in Fig. 1.3.

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

In the first pass, if the records are divided into 12 runs, 500 records each, then we have runs generated as $R_{1\,1}$ to $R_{1\,12}$. In the second pass, $R_{1\,1}$ to $R_{1\,4}$ are merged to form $R_{2\,1}$. In the same way $R_{2\,2}$ and $R_{2\,3}$ are generated by merging $R_{1\,5}$, $R_{1\,6}$, $R_{1\,7}$, $R_{1\,8}$ and $R_{1\,9}$, $R_{1\,10}$, $R_{1\,11}$, $R_{1\,12}$ respectively. In the second pass all the runs ($R_{2\,1}$, $R_{2\,2}$, and $R_{2\,3}$) are merged together to form the final sorted run, i.e., $R_{3\,1}$.

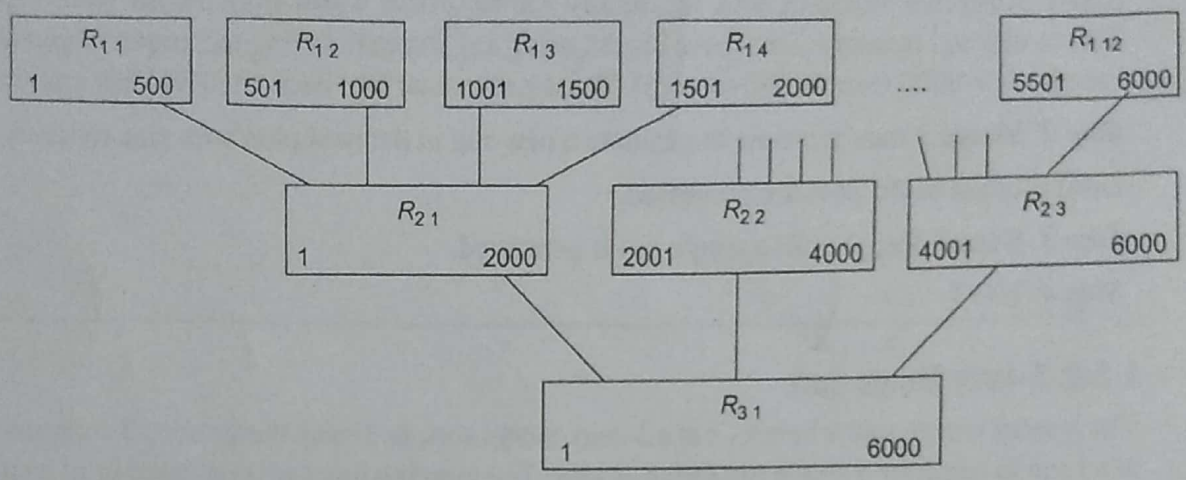


Figure 1.3 4–way merge sort

From the above we observe that as we increase the number of runs to be merged, there is a substantial decrease in the number of passes. To summarize, k–way merge combines k runs into one sorted run where k ≥2. The algorithm for k–way merge sort is shown in Fig. 1.4.

```
k-way_mergesort( File1, M, k)
     File1: unsorted file on disk
     M: max records that can be stored and sorted in internal memory at a time
     k: number of runs to be merged
Step 1: Run Generation
        Repeat until all records in File1 are processed into runs
                Read M records into main memory and sort internally
                Write this sorted sub-list (run) onto disk
        [END OF LOOP]
Step 2: Merging of Runs
        Repeat until all runs are processed
                Merge k runs into one sorted run with size as k times the input run size
                Write this single run back onto disk
        [END OF LOOP]
Step 3: If more than 1 run is generated in Step 2 then
           Repeat Step 2
        Else
           The file on the external storage is sorted
Step 4: EXIT
```

Figure 1.4 Algorithm for k–way merge sort

1.2.4 Merging Runs

As the k–way merge sort includes merging of runs in every pass, let us now look at the merging process in more detail. The simplest merging technique is the 2–way merge, where two sorted runs are merged into one run. In a 2-way merge, firstly, we need to identify the first smallest record of

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

each run and place it in the smallest set. The smallest of the two records in the smallest set is the smallest record overall. This record is put in the output run and removed from the corresponding run. The next smallest record in the corresponding run is moved to the smallest set. This process is repeated until the two initial runs are empty. The k–way merge behaves identically, but at each step the smallest of k records is selected and removed from its run.

Example 1.2 Consider 3 runs with 4 records each as shown in Fig 1.5. Implement the 3-way merge sort technique.

Run 1	3	5	12	15								
Run 2	2	4	10	17								
Run 3	1	6	8	18								
Output Run	1	2	3	4	5	6	8	10	12	15	17	18

Figure 1.5 Three runs ready to be merged and the output run resulting from the merge

Solution

- Consider the smallest record of each run and add it to the smallest set: {3, 2, 1}.
- Take the smallest record of the smallest set, 1, add it to the output run and delete it from the original run. At this point, the output run is {1}. The step-by- step process of merging the three runs of Fig. 1.5 is as follows:

Step 1: The three records in the smallest set are {3, 2, 1}. Remove the smallest record1, from the third run and put it in the output run: {1}. Move 6 to the smallest set.

Step 2: The three records in smallest set are {3, 2, 6}. Remove 2 from the second run and append it to the output run: {1, 2}. Move 4 to the smallest set.

Step 3: The three records in smallest set are {3, 4, 6}. Remove 3 from the first run and append it to the output run: {1, 2, 3}.Move 5 to the smallest set.

Step 4: The three records in smallest set are {5, 4, 6}. Remove 4 from the second run and append it to the output run: {1, 2, 3, 4}. Move 10 to the smallest set.

Step 5: The three records in smallest set are {5, 10, 6}. Remove 5 from the first run and append it to the output run: {1, 2, 3, 4, 5}. Move 12 to the smallest set.

Step 6: The three records in smallest set are {12, 10, 6}. Remove 6 from the third run and append it to the output run: {1, 2, 3, 4, 5,6}. Move 8 to the smallest set.

Step 7: The three records in smallest set are {12, 10, 8}. Remove 8 from the third run and append it to the output run: {1, 2, 3, 4, 5, 6, 8}. Move 18 to the smallest set.

Step 8: The three records in smallest set are {12, 10, 18}. Remove 10 from the second run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10}. Move 17 to the smallest set.

Step 9: The three records in smallest set are {12, 17, 18}. Remove 12 from the first run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10, 12}. Move 15 to the smallest set.

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

Step 10: The three records in smallest set are {15, 17, 18}. Remove 15 from the first run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10, 12, 15}.The first run is now empty, the merge follows as a 2-way merge instead of a 3-way merge.

Step 11: The two top records are {17, 18}. Remove 17 from the second run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17}. Now, the second run is also empty, only the third run remains non-empty.

Step 12: The records of the last run are {18} and are appended to the output run and the final run is obtained {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17, 18}.

The result of merging the three initial runs is {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17, 18}.

1.2.5 Time Complexity

To analyze the time complexity of external sorting, the three factors influencing the read/write time from a disk must be known. They are as follows:

- **Seek time:** It is the time taken to position the read/write heads on to the required cylinder and it depends on the number of cylinders across which the head moves.
- **Latency time:** It is the time taken to position the head on to the right sector on the disk.
- **Transmission Time:** It is the time taken to transmit the block of data to/from the disk.

To perform analysis over the complexity of external sort, the following terms are used:

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of records

t_{IO} = time to input or output one block ($t_s + t_l + t_{rw}$)

t_{IS} = time taken to internally sort the records in memory

nt_m = time taken to merge n records from input buffers to the output buffer

Here, we assume that every time a block of records is read from or written onto the disk, the maximum seeks and latency times are utilized. However, this may not be true in reality, but this assumption simplifies the analysis. The computing time for various operations performed in Example 1.1, to sort 6000 records stored in disk is given in Fig. 1.6.

Operations	Time
Step 1: Read 60 blocks of records from the input file, $60t_{IO}$, perform internal sort with time taken as $12t_{IS}$. Again write back 60 blocks on to the disk, 60 t_{IO}	$120\ t_{IO} + 12t_{IS}$
Step 2: Merge runs 1 to 12 of 500 records each and every two runs at a time.	$120\ t_{IO} + 6000t_m$
Step 3: Merge two runs of 1000 records each, with a total of 20 blocks.	$120\ t_{IO} + 6000t_m$
Step 4: Merge two runs of 2000 records each, with a total of 40 blocks.	$80t_{IO} + 4000t_m$
Step 5: Merge one run of 4000 records with one run of 2000 records.	$120t_{IO} + 6000t_m$
Total Computing Time =	$560t_{IO} + 22000t_m + 12t_{IS}$

Figure 1.6 Computing times for operations performed in Example 1.1

From Fig. 1.6, it is evident that the computing time for the sorting process is mainly dependent on the number of passes made over the data to get it sorted. The first pass involves the data that

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

is divided into runs after the internal sort is carried out on the data with size equal to the capacity of the internal memory.

The merging of runs requires 3⅔ passes over the entire set of records. In merging, the first pass is used to merge 12 runs of length 500 records each yielding to 6 runs of length 1000 records each. The time taken for this merge is $6000t_m$. The second pass is used to merge 6 runs yielding 3 runs of length 2000 records each. The time taken for this merge is $6000t_m$. In the third pass, only 2 runs of the 3 are merged, i.e., merge 2 runs of length 2000 records each yielding to a run of length 4000 records and another run of length 2000 records. This is considered as 2/3 of the pass. The time taken for this merge is $4000t_m$. The fourth pass is used to merge two runs, one run of length 4000 records and the other with 2000 records yielding to a single run of length 6000 records. The time taken for this merge is $6000t_m$.

The total number of passes required for sorting is 4⅔ (1 pass for generating runs and 3⅔ passes for merging the runs). Therefore, the total input and output time is $2 \times (4⅔) \times 60t_{IO} = 560t_{IO}$. The time required for merging is $3⅔ \times 6000t_m = 22000t_m$.

Due to the strong correlation of computing time with the number of passes made over the data, the complexity of the k–way merge sort is mainly concerned with the counting of number of passes required.

In general if we start with m runs in a 2–way merge, $\log_2 m + 1$ passes are required to sort the records. The number of passes can be reduced with the use of a higher-order merge, i.e., k–way merge with $k \geq 2$. Therefore, a k–way merge with m runs requires $\log_k m + 1$ passes for sorting the data. However when we use higher-order merge, k–runs of length $s_1, s_2, s_3 \ldots\ldots, s_k$ cannot be merged in $O(\Sigma_1^k s_k)$ time as the next record to be output next is not confined to the smallest key of the two possibilities in a 2–way merge but instead need to consider all the k possibilities, i.e., the k smallest keys present in the k–runs.

The only way to merge k–runs is to make $k-1$ comparisons to determine the next record to output. The computing time now becomes $O((k-1)\Sigma_1^k s_k)$. As $\log_k m$ passes are made, the total number of key comparisons is $n(k-1) \log_k m = n(k-1) \log_2 m / \log_2 k$, where n is the number of records to be sorted. So, $(k-1)/\log_2 k$, is the factor by which number of key comparisons increases.

As the value of k increases, the input/output time will decrease. But, beyond a certain value of k, the input/output time will increase despite the decrease in the number of passes being made. The optimal value for k depends on disk parameters and the amount of internal memory available for buffers.

With higher-order merge, the amount of I/O needed may be reduced with no significant loss in the internal processing speed but it is not as much as indicated by the reduction to $\log_k m$ passes. This is mainly due to the increase in the number of input buffers in k–way merge increases with the k.

1.3 BUFFER HANDLING FOR PARALLEL OPERATION

As the k–way merge sorting technique is introduced in the previous section, let us now discuss in more detail the process of merging runs internally using buffers. In k–way merge sorting technique, we need at least k–input buffers and 1–output buffer for performing the merge operation. The major steps involved in the k–way merge sort are:

- **Read:** Load the records from runs into input buffers.
- **Merge:** Perform merge operation on input buffers and store the result in output buffer.
- **Write:** Store the records from the output buffer onto the disk.

In the previous section, the read, merge, and write operations are shown to be performed sequentially. Instead of performing these operations in sequence, we can opt for performing these operations in parallel order to improve the efficiency, i.e., we can perform read, merge, and write

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

operations simultaneously. But k–input buffers and 1–output buffer strategy is not sufficient to handle the parallel operations.

While the output buffer is being written on to the disk, it is not possible to perform the merge operation as there is no place to collect the merged records. So, we need 2 output buffers, one for write operation and one for merge operation. Similarly, if the content is being loaded into the input buffer, merging will be delayed. This can be handled by taking 2 input buffers. If one input buffer is busy with read operation, then the other input buffer is used for merge operation. Therefore, we require 2k–input buffers and 2–output buffers to handle the operations in parallel, i.e., double buffering strategy is used. The parallel operations are demonstrated in Example 1.3.

Example 1.3 Using buffer handling for parallel operations, implement 2–way merge for the runs shown below. Consider the block size as 2 and each buffer can hold 2 records.

Run 1

2	4	6	8	9	10

Run 2

3	5	7	16	21	26

Solution As 2–way merge is to be used, the input buffers used are 4 and the output buffers used are 2. Let `in[1]` `in[2]` represent the input buffers for loading the records from Run1 and `in[3]`, `in[4]` represent the input buffers for loading the records from Run2. Let `ou[0]` and `ou[1]` represent the output buffers. Initially all the input and output buffers are empty.

Step 1: **Read**–Load `in[1]` with a block from Run1. The configuration of buffers after read operation is shown in Fig. 1.7.

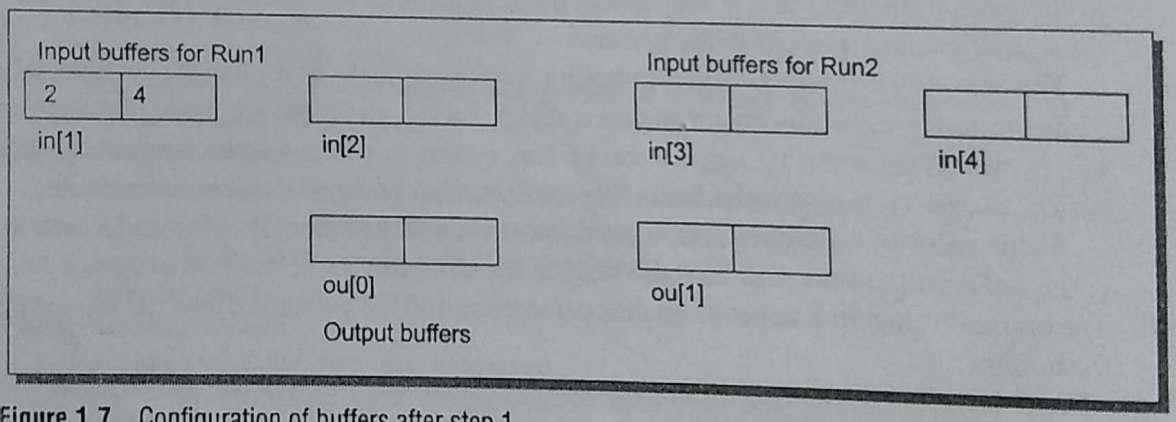


Figure 1.7 Configuration of buffers after step 1

Step 2: **Read**–Load `in[3]` with a block from Run2. The configuration of buffers after read operation is shown in Fig. 1.8.

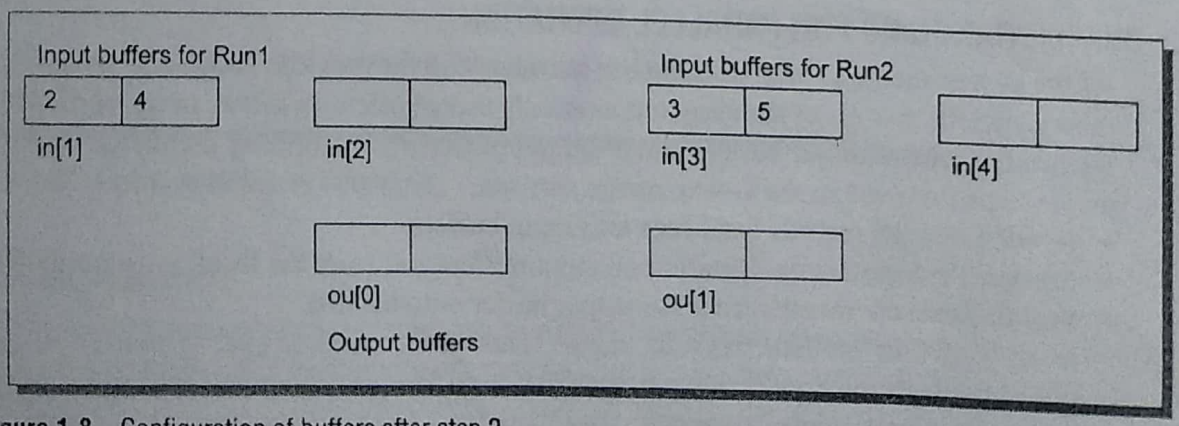


Figure 1.8 Configuration of buffers after step 2

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

Step 3: **Read**–Load `in[2]` with a block from Run1.
Merge–Perform merge operation on input buffers `in[1]` and `in[3]`. Store the merged record in ou[0]. The configuration of buffers after read and merge operations is shown in Fig. 1.9.

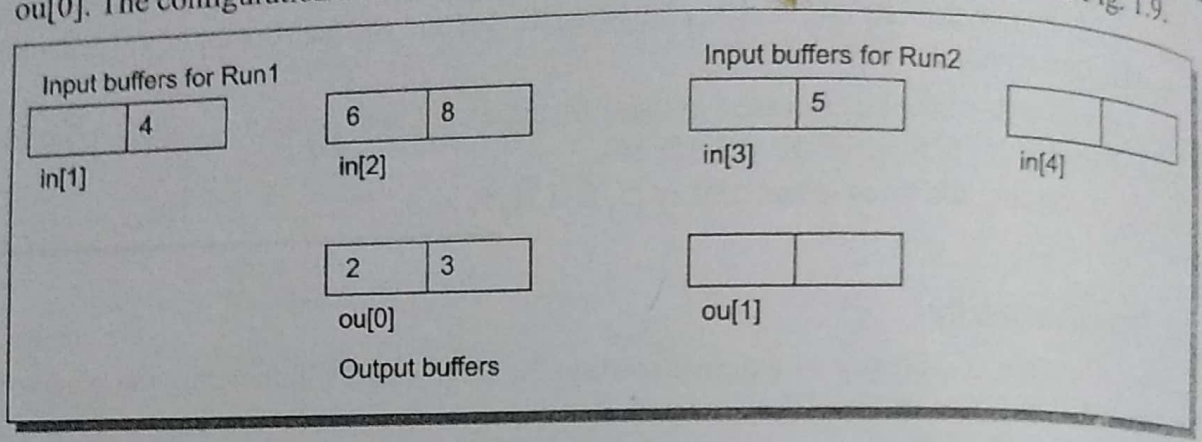


Figure 1.9 Configuration of buffers after step 3

Step 4: **Read**–Load `in[4]` with a block from Run2.
Merge–Perform merge operation on input buffers `in[1]`, `in[2]` and `in[3]`. Store the merged record in `ou[1]`.
Write–Store the content in `ou[0]` onto the disk. The configuration of buffers after read, merge, and write operations is shown in Fig. 1.10.

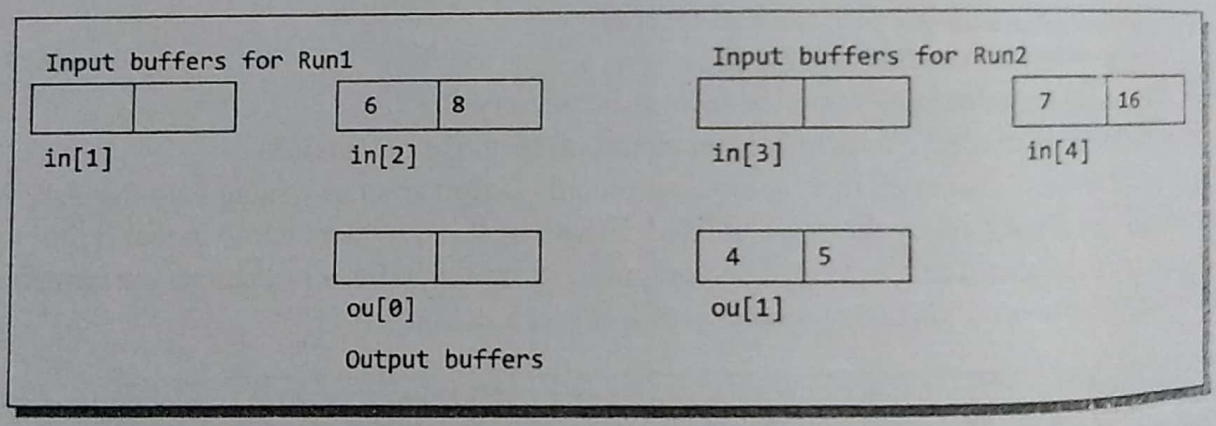


Figure 1.10 Configuration of buffers after step 4

Step 5: **Read**–Load `in[1]` with a block from Run1.
Merge–Perform merge operation on input buffers `in[2]` and `in[4]`. Store the merged record in `ou[0]`
Write–Store the content in `ou[1]` onto the disk. The configuration of buffers after read, merge, and write operations is shown in Fig 1.11.

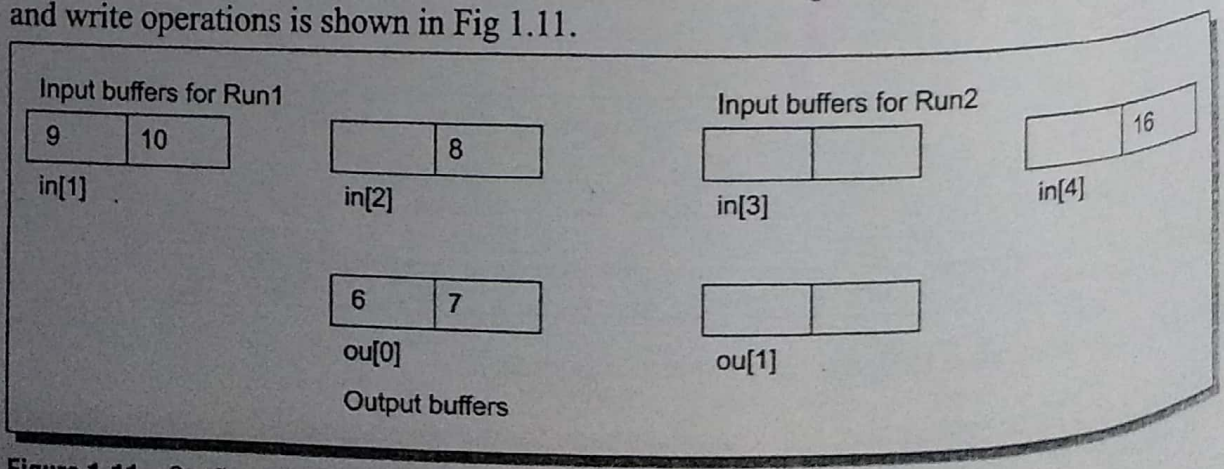


Figure 1.11 Configuration of buffers after step 5

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

Jntuk396.blogspot.com

Step 6: **Read**–Load `in[3]` with a block from Run2.
Merge–Perform merge operation on input buffers `in[1]`, `in[2]`, and `in[4]`. Store the merged record in `ou[1]`
Write–Store the content of `ou[0]` onto the disk. The configuration of buffers after read, merge, and write operations is as shown in Fig. 1.12.

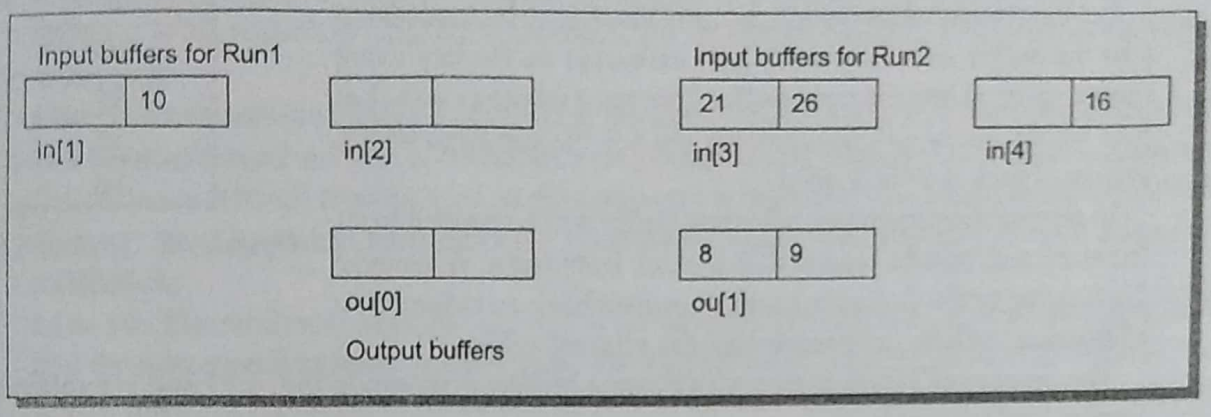


Figure 1.12 Configuration of buffers after step 6

Step 7: **Merge**–Perform merge operation on input buffers `in[1]`, `in[3]`, and `in[4]`. Store the merged record in `ou[0]`.
Write–Store the content in `ou[1]` onto the disk. The configuration of buffers after merge and write operations is shown in Fig. 1.13.

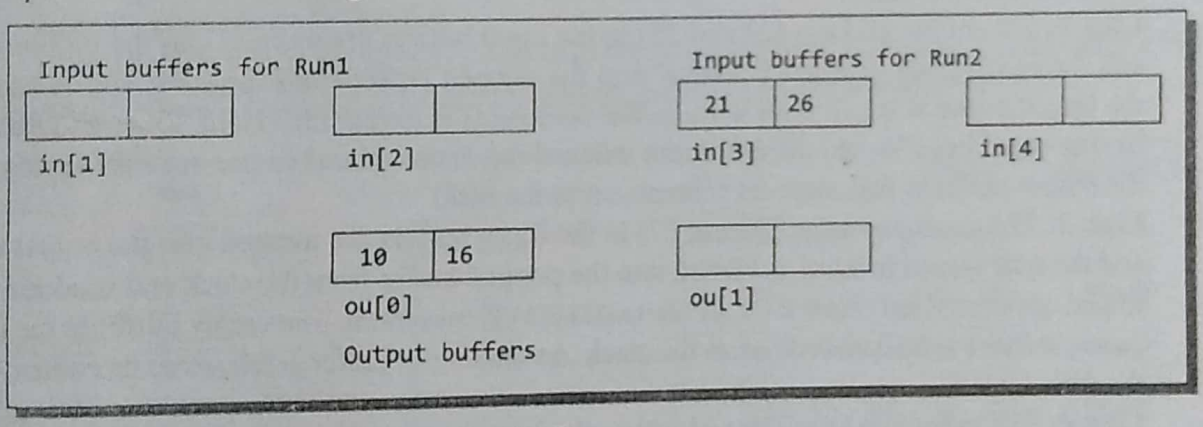


Figure 1.13 Configuration of buffers after step 7

After Step 7, merging will be delayed until another record is loaded from Run1 into `in[1]`.

In the above example, even though we used 2 buffers per run, we reached a situation in which processing has to be held up due to lack of input records from the run. Simply assigning 2 buffers per run does not solve the problem. Therefore, input buffers have to be assigned to the runs cleverly to avoid the input delay, i.e., an individual buffer may be assigned to any run depending upon need. This type of buffer that is assigned to a run when needed is called a floating buffer and the k–way merge algorithm using floating buffers is called *buffering algorithm.*

In the buffer assignment strategy of floating buffers, at any time there will be at least one input buffer for each run. The remaining buffers will be filled on a priority basis. The following assumptions are made for the k–way merge with floating buffers (buffering algorithm):

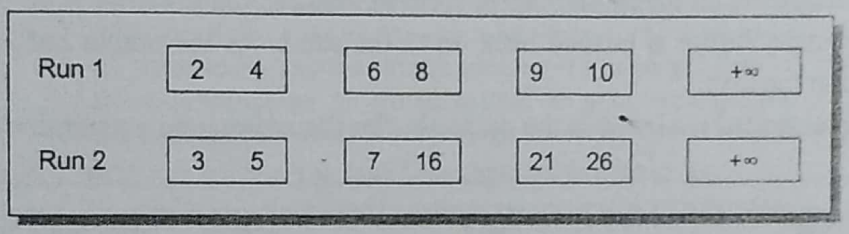
1. For performing the operations of reading from a disk and writing onto a disk simultaneously, we take two disk drives.
2. While read or write operation is being processed, the CPU cannot make reference to the same block of memory, i.e., the same output buffer cannot be used for filling in and writing out. If this is possible, only one output buffer is sufficient.

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

3. Input and output buffers are of the same size.
4. End of each run has a sentinel record with a very large key, say $+\infty$ and all other records have key value less than that of sentinel record.
5. The time to merge into an output buffer equals the time to read a block.
6. In case of equal keys, the run with smallest index is chosen for the next read operation.
7. Input buffers are queued in k queues, one queue for each run. Empty buffers are placed on a linked stack.

The process of implementing k–way merge using floating buffers is demonstrated in detail in Examples 1.4 and 1.5.

Example 1.4 Implement k–way merge with floating buffers for the runs shown below. Assume that the last block of each run is loaded with the sentinel record with key $+\infty$. The blocks in Run1 and Run2 are as follows:



Solution As we have considered two runs, `k = 2` and there will be 2 linked queues one for each run, which are initially empty. There are four floating buffers (`b1`, `b2`, `b3`, and `b4`) initially placed on the stack as in the Fig. 1.14. Let `N` be the run from which the next block is to be loaded and `s` be the buffer size. Here, `s = 2`. Let `LastKey[i]` be the key value recently placed into the input buffer from run i where `1 ≤ i ≤ k`. Let `N = i` where `LastKey[i]` is minimum and `N` is the run from which the next block is to be loaded. In the next load operation, a floating buffer will be popped from the stack and then it will be loaded with a block from run `N`. Whenever the floating buffer is empty, it will be pushed back on to the stack of floating buffers. Let `M` represent the sentinel value.

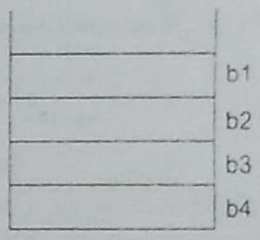


Figure 1.14 Stack of 4 floating buffers

The process of 2–way merge with floating buffers is shown in Fig. 1.15 and is explained in a line-by-line manner below.

Line 1: Take two floating buffers (one for each run) from the stack of floating buffers and assign a buffer to each run. Load the first block from Run1 into floating buffer `b1` and first block from Run2 into floating buffer `b2`. `LastKey[1] = 4` and `LastKey[2] = 5`. Here, `N = 1`, as the `LastKey[1]` is minimum. The next block is to be loaded from Run1.

Line 2: The minimum keys (2 and 3) in the input buffers are merged into the output buffer and the next record in Run1 is loaded into the popped buffer `b3` and is added to the linked queue at Run1. Now `N = 2`, as the `LastKey[2] = 5` is minimum when compared to `LastKey[1]`. As the output buffer is full, store its contents on to the disk.

Line 3: The minimum keys (4 and 5) in the input buffers are merged into the output buffer and the next record in Run2 is loaded into the popped buffer `b4` and is added to the linked queue at Run2. Now `N = 1`, as the `LastKey[1]` is minimum. The empty buffer `b1` is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 4: The minimum keys (6 and 7) in the input buffers are merged into the output buffer and the next record in Run1 is loaded into the popped buffer `b1` and is added to the linked queue at

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

Run1. Now `N = 1`, as the `LastKey[1]` is minimum. The empty buffer `b2` is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 5: The minimum keys (8 and 9) in the input buffers are merged into the output buffer and the next record `M` (sentinel record) in Run1 is loaded into the popped buffer `b2` and is added to the linked queue at Run1. Now `N = 2`, as the `LastKey[2]` is minimum. The empty buffer `b3` is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 6: The minimum keys (10 and 16) in the input buffers are merged into the output buffer and the next record in Run2 is loaded into the popped buffer `b3` and is added to the linked queue at Run2. Now `N = 2`, as the `LastKey[2]` is minimum. Finally, the empty buffers `b1`, `b4` are pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 7: The minimum keys (21 and 26) in the input buffers are merged into the output buffer and the next record `M` (sentinel record) in Run2 is loaded into the popped buffer and is added to the linked queue at Run2. As all the input buffers consists sentinel record there is no load step further. Finally, the empty buffer is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 8: As the sentinel record is placed in the output buffer the entire process terminates.

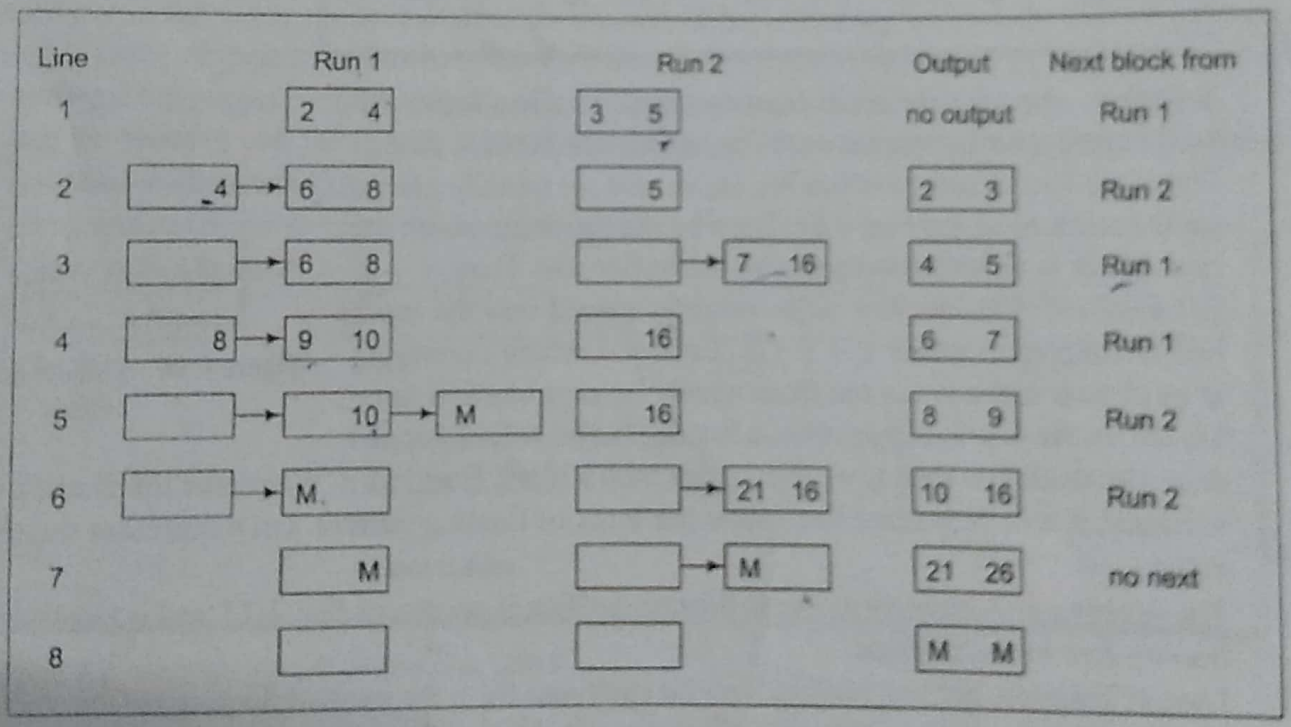


Figure 1.15 The status of linked queues for 2 runs

Example 1.5 Implement k–way merge with floating buffers for the runs shown below. Assume that the last block of each run is loaded with the sentinel record with key $+\infty$. The blocks in Run1, Run2, and Run3 are as follows:

Run 1	22	27	28	30	31	32	35	$+\infty$
Run 2	25	31	36	38	40	62	72	$+\infty$
Run 3	26	30	33	35	42	45	52	$+\infty$

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

Solution As we have considered three runs, `k = 3` and there will be 3 linked queues, one for each run which are initially empty. There are six floating buffers (`b1`, `b2`, `b3`, `b4`, `b5`, and `b6`) initially placed on the stack as in the Fig. 1.16.

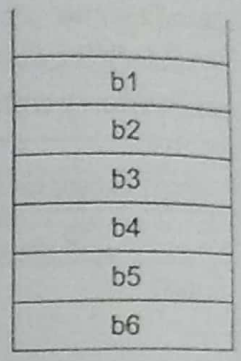


Figure 1.16 Stack of 6 floating buffers

Let `N` be the run from which the next block is to be loaded and `s` be the buffer size. Here `s = 2`. Let `LastKey[i]` be the key value recently placed into the input buffer from run `i` where `1 ≤ i ≤ k`. Let `N = i` where `LastKey[i]` is minimum and `N` is the run from which the next block is to be loaded.

In the next load operation a floating buffer will be popped from the stack and then be loaded with a block from run `N`. Whenever the floating buffer is empty it will be pushed back on to the stack of floating buffers. Let `M` represents the sentinel value.

The process of 3–way merge with floating buffers is shown in Fig. 1.17 and is explained in a line-by-line manner below.

Line 1: Take three floating buffers (one for each run) from the stack of floating buffers and assign a buffer to each run. Load the first block from Run1 into floating buffer `b1` and first block from Run2 into floating buffer `b2` and first block from Run3 into floating buffer `b3`. `LastKey[1] = 27`, `LastKey[2] = 31` and `LastKey[3] = 30`. Here `N = 1`, as the `LastKey[1]` is minimum. The next block is to be loaded from Run1.

Line 2: The minimum keys (22 and 25) in the input buffers are merged into the output buffer and the next record in Run1 is loaded into the popped buffer from the stack and is added to the linked queue at Run1. Now `N = 1`, as the `LastKey[1]` is minimum. (Here, Run1 and Run3 are having equal keys 30, but the minimum indexed run is considered as per our assumptions). As the output buffer is full, store its contents on to the disk.

Line 3: The minimum keys (26 and 27) in the input buffers are merged into the output buffer and the next record in Run1 is loaded into the popped buffer from the stack and is added to the linked queue at Run1. Now `N = 3`, as the `LastKey[3]` is minimum. The empty buffer in the linked queue at Run1 is pushed back on to the stack. As the output buffer is full, store its contents onto the disk.

Line 4: The minimum keys (28 and 30) in the input buffers are merged into the output buffer and the next record in Run3 is loaded into the popped buffer from the stack and is added to the linked queue at Run3. Now `N = 2`, as the `LastKey[2]` is minimum. The empty buffer in the linked queue at Run1 is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 5: The minimum keys (30 and 31) in the input buffers are merged into the output buffer and the next record in Run2 is loaded into the popped buffer from the stack and is added to the linked queue at Run2. Now set `N = 1`, as the `LastKey[1]` is minimum. As the output buffer is full, store its contents on to the disk.

Line 6: The minimum keys (31 and 32) in the input buffers are merged into the output buffer and the next record in Run1 is loaded into the popped buffer from the stack and is added to the linked queue at Run1. Now set `N = 3`, as the `LastKey[3]` is minimum. The empty buffer in the linked queue at Run1 is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 7: The minimum keys (33 and 35) in the input buffers are merged into the output buffer and the next record in Run3 is loaded into the popped buffer from the stack and is added to the

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

linked queue at Run3. Now set N = 2, as the LastKey[2] is minimum. As the output buffer is full, store its contents on to the disk.

Line 8: The minimum keys (35 and 36) in the input buffers are merged into the output buffer and the next record in Run2 is loaded into the popped buffer from the stack and is added to the linked queue at Run2. Now set N = 3, as the LastKey[3] is minimum. The empty buffer in the linked queue at Run3 is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 9: The minimum keys (38 and 40) in the input buffers are merged into the output buffer and the next record in Run3 is loaded into the popped buffer from the stack and is added to the linked queue at Run3. Now set N = 2, as the LastKey[2] is minimum. The empty buffer in the linked queue at Run2 is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 10: The minimum keys (42 and 45) in the input buffers are merged into the output buffer and the next record in Run2 is loaded into the popped buffer from the stack and is added to the linked queue at Run2. Now, the minimum key of the last loaded records from each runs is $+\infty$ (M) which indicates there is no next load operation. The empty buffer in the linked queue at Run3 is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 11: The minimum keys (52 and 62) in the input buffers are merged. As there is no next (N value) in the previous step, no load operation is performed and no need to update N. The empty buffer in the linked queue at Run2 is pushed back on to the stack. As the output buffer is full, store its contents on to the disk.

Line 12: The minimum keys (72 and M) in the input buffers are merged into the output buffer. As the sentinel recorded is loaded merged the process terminates with this step.

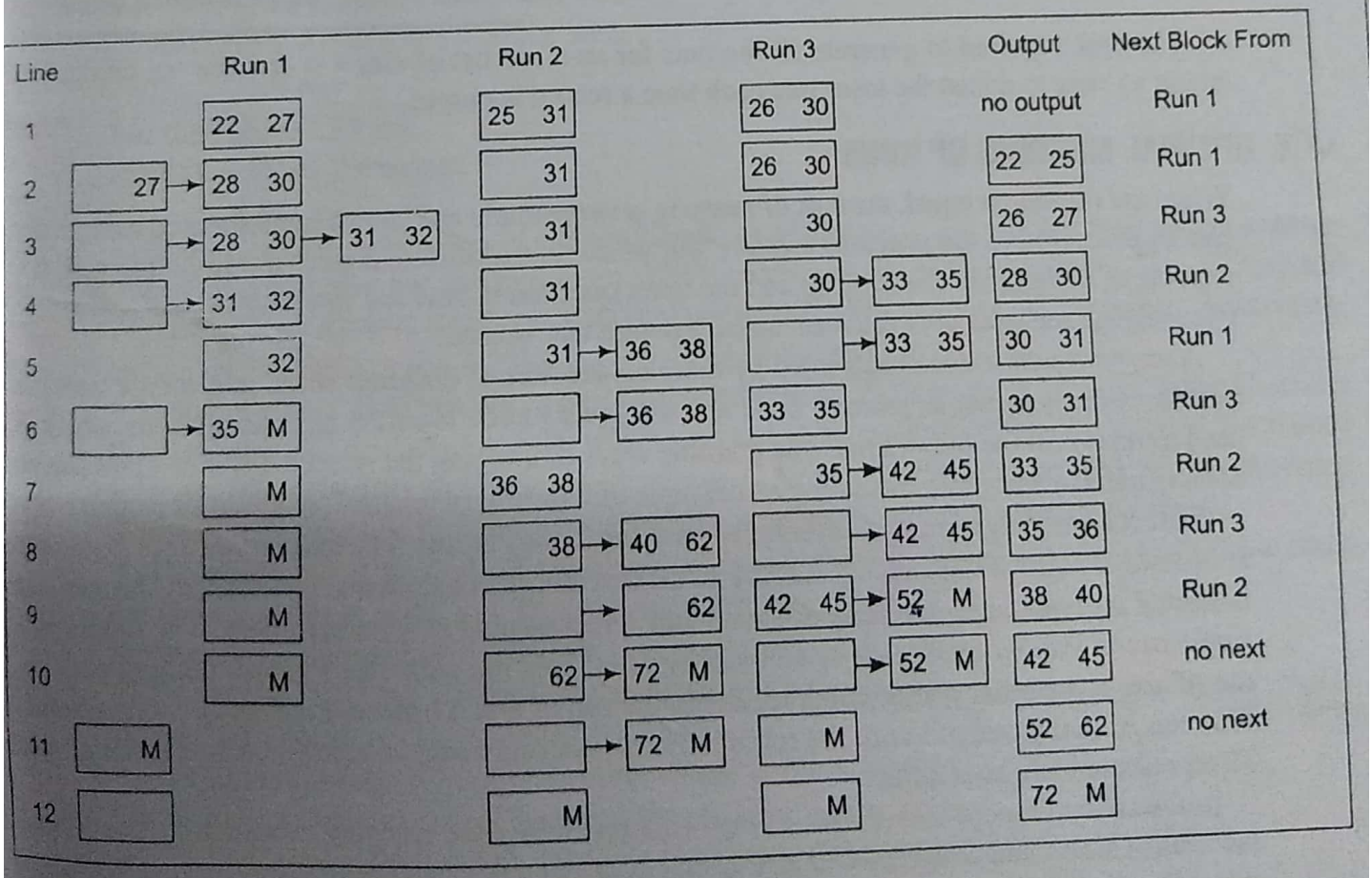


Figure 1.17 Status of linked queues for 3 runs

Scanned by CamScanner

The algorithm depicting the process of k–way merge using floating buffers is presented in Fig. 1.18.

```
Let k be the no. of runs, s be the buffer size
Step 1: Set up k linked queues, insert the first block of each run in an input buffer
        attached to the corresponding queue. The remaining k input buffers are pushed
        into a linked stack of empty input buffers. Let the output buffer be represented
        as ou.
Step 2: Let LastKey[i] be the key value recently placed into the input buffer from run i
        where 1 ≤ i ≤ k. Let N = i where LastKey[i] is minimum. N is the run from which the
        next block is to be loaded. If LastKey[N] ≠ +∞, then insert the next block from
        run N into free input buffer.
Step 3: Select s minimum keys from all the input buffers, merge them into the output
        buffer ou and initiate the writing of ou onto disk. Merging continues until
        the output buffer gets full. If an input buffer becomes empty before the output
        buffer gets full, the next buffer on the same queue is considered and the empty
        buffer is returned to the stack of empty buffers. However, if an input buffer
        becomes empty and the output buffer gets +∞ merged into it, the empty buffer is
        left on the queue. If the output buffer gets +∞ merged into it, the contents of
        ou are copied onto disk, the merge terminates and the control goes to Step 6.
Step 4: Wait for any ongoing disk input/output to complete.
Step 5: Repeat Steps 2 to 4 until there is no input block to be read.
Step 6: EXIT
```

Fig 1.18 Algorithm for k–way merge with floating buffers

1.4 RUN GENERATION

With the existing internal sorting techniques, it is possible to generate runs whose size is equal to the number of records that can be held in the internal memory. However, by using the loser's tree, it can be performed in a better way. A *loser tree* can be understood clearly by knowing firstly about winner trees. A *winner tree* is a complete binary tree which helps us to obtain the smallest key from a set of keys where each node represents the smaller of its two children. The root consists of the smallest value in the entire tree.

The construction of the winner tree may be compared with the tournament tree. In a tournament tree, each external node represents a player and each internal node represents the winner of the match played between the players represented by its children nodes. Similarly, each internal node of the winner tree represents the winner of a tournament and the root node is the overall winner of the entire tournament. Each internal node consists of smallest value of its children and the overall winner is the key with the smallest value in the entire tree. The keys or records are present in the external nodes or leaf nodes only. All the internal nodes consist of only the pointers to the winners of each tournament played in the respective levels of the tree. Here, in the k–way merge sort, leaf nodes of the winner tree consists of the first record from the k runs being merged. The concept of merging using loser trees can be better understood using the following example.

Example 1.6 Consider merging of 4 runs using winner tree and loser tree for the runs shown below.

Scanned by CamScanner

Run 1	Run2	Run 3	Run 4
15	10	24	8
25	18	56	12
30	20	68	15
46	93	76	28

Solution To construct winner tree, all the first keys from all the 4 runs will be initially loaded into the external nodes as shown in Fig. 1.19.

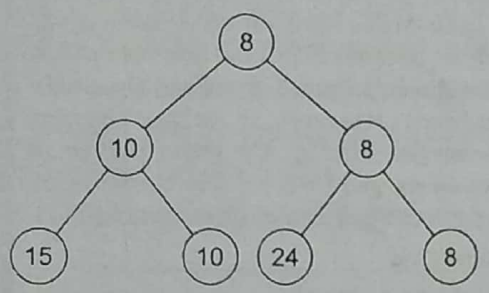


Figure 1.19 Winner tree

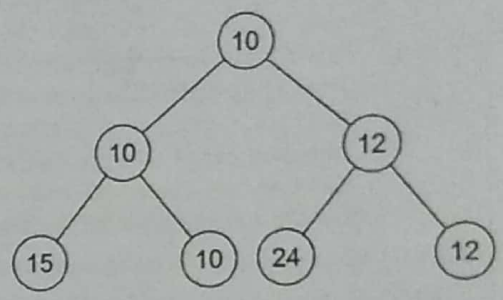


Figure 1.20 Winner tree after first record is output

The tournament is played between the sibling nodes and the resulting winner (smallest key) is placed in the parent node. The root node consists of the overall winner of the tournament which is the result of tournament played between the winners of the previous level in the tree. Every new comparison takes place at every level of the tree, starting from the leaf nodes towards the root. In the Fig. 1.19, the internal nodes 10 and 8 represent the winner of the respective tournaments played between 15, 10, and 24, 8 respectively. The root node of the winner tree consists of the overall winner, i.e., 8. This value pointed by root is stored onto the disk.

When a record is output in the winner tree, the next record to be placed in the external node is taken from the run from which the smallest record obtained as the output in the previous pass. Hence, in this example, the smallest record at root, 8, is taken from Run4. So, the next value that is input is to be taken from Run4. The next value in Run4 to be input is 12. The value 12 is compared against the other values which are basically the losers in the previous pass. The winner tree constructed after first record is output is shown in Fig. 1.20. The process repeats until all the elements from the runs are processed.

Another efficient way of generating runs is by using loser trees in which the number of comparisons is reduced. The winner tree can be restructured in such a way that instead of placing pointers to winners as internal nodes, pointers to losers can be placed as internal nodes. Such trees are called as *loser trees*. A special node is added on top of the root node to represent the overall winner of the tournament. The leaf nodes of loser tree consist of the first records of all the runs as in winner tree.

The loser tree for Example 1.6 is shown in the Fig. 1.21. The internal nodes 15 and 24 represent the losers of the respective tournaments played between 15, 10 and 24, 8 respectively. The root node consists of loser of the winner nodes 10 and 8, i.e., 10. The overall winner consists of the winner of 10 and 8 (i.e., 8).

Scanned by CamScanner

In the first step, the overall winner is the output and therefore the value 8 is copied onto the disk. The next value is to be considered from Run4 as the output value 8 is from Run4. The value to be input is now 12. This is copied into the leaf nodes. But once the overall winner is output, the records with which the input needs to play these tournaments are readily available in the parent nodes. As a result, the sibling nodes are not accessed when the tournaments are being played.

The loser tree after the first record is output is shown in Fig. 1.22. The value 12 in leaf node is compared with its parent 24 and not compared with its sibling 24. The root node consists of loser of the nodes 10 and 12, i.e., 12. The overall winner consists of the winner of 10 and 12 (i.e., 10). The process repeats until all the elements from the runs are processed.

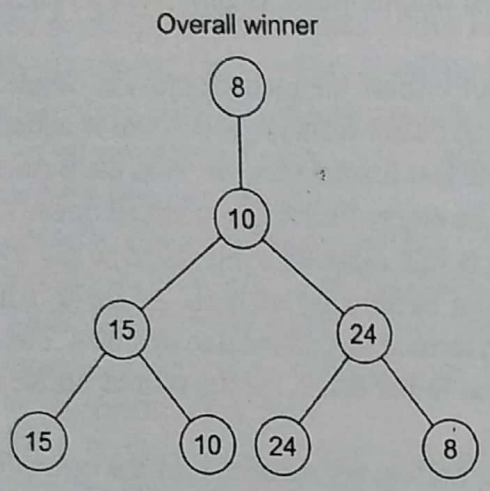


Figure 1.21 Loser tree

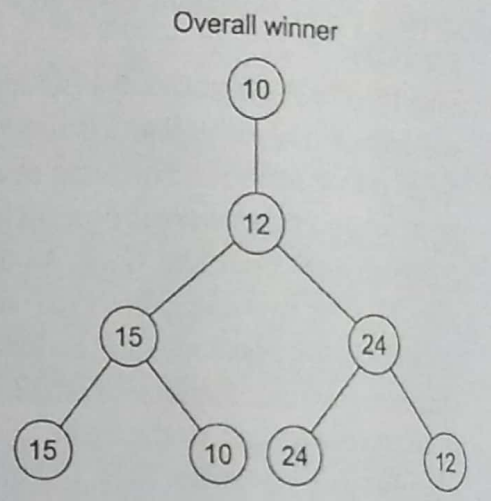


Figure 1.22 Loser tree after the first record is output

The time required to generate all the runs for an n run list of size `k` is `O(n log k)`, as it takes `O(log k)` time to adjust the loser tree each time a record is output.

1.5 OPTIMAL MERGING OF RUNS

When the run size is equal, the task of merging is very simple and can be implemented using loser tree as discussed in the previous section. But when runs of different sizes are to be merged, the process of merging is challenging and the main question is how the merging can be optimized. For merging the runs with different sizes, a merge tree is used.

A *merge tree* is a tree constructed by merging the runs of different sizes. In a merge tree, the circular node is called as internal node and the square node is called as external node which is used to represent the initial runs. The possible ways of merging the runs of different sizes can be demonstrated clearly by considering an example of four runs of length 3, 6, 8, and 14 respectively.

The runs are merged in two different ways using 2–way merge. The two Merge trees generated are shown in Fig. 1.23(a) and Fig. 1.23(b). In the first merge tree shown in Fig. 1.23 (a), the merging is started by combining the runs of size 3 and 6 represented as external nodes. The output is a single run of size 9 represented as an internal node. This run with size 9 is next merged with the run of size 8 (external node) which results in the run of size 17 represented as an internal node. This run is finally merged with the run of size 14(external node) to result in a single run of size 31 represented

Huffman worked on this problem and came out with a solution by constructing a Binary tree with minimum weighted external path length called the *Huffman tree*. The Huffman algorithm is applied for constructing the Huffman tree. The two main activities done using the Huffman algorithm are constructing a Huffman tree and finding the Huffman codes by traversing the nodes from root to leaf nodes in the Huffman tree. In the Huffman tree, the leaf nodes represent the characters and the internal nodes represent the intermediary values. The process of constructing a Huffman tree and finding the Huffman codes is demonstrated Example 1.7.

Note **HUFFMAN TREE** Huffman coding is an entropy encoding algorithm developed by David A. Huffman that is widely used as a lossless data compression technique. The Huffman coding algorithm uses a variable length code table to encode a source character where the variable-length code table is derived on the basis of the estimated probability of occurrence of the source character.

The key idea behind Huffman algorithm is that it encodes the most common characters using shorter strings of bits than those used for less common source characters.

The algorithm works by creating a binary tree of nodes that are stored in an array. A node can be either a leaf node or an internal node. Initially, all the nodes in the tree are at the leaf level and store the source character and its frequency of occurrence (also known as weight).

While the internal node is used to store the weight and contains links to its child nodes, the external node contains the actual character. Conventionally, a '0' represents following the left child and a '1' represents following the right child. A finished tree that has n leaf nodes will have n − 1 internal nodes.

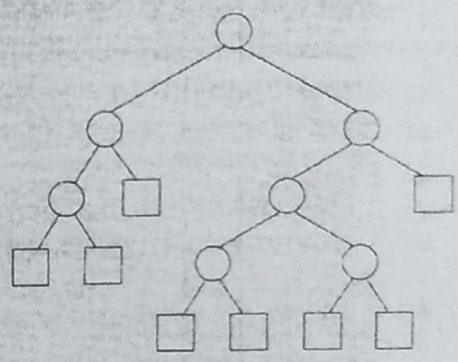


Figure 1.25 Binary tree

The running time of the algorithm depends on the length of the paths in the tree. So, before going into further details of Huffman coding, let us first learn how to calculate the length of the paths in the tree. The *external path length* of a binary tree is defined as the sum of all path lengths summed over each path from the root to an external node. The internal path length is also defined in the same manner. The *internal path length* of a binary tree is defined as the sum of all path lengths summed over each path from the root to an internal node. Look at the binary tree given in Fig. 1.25.

The internal path length, $L_I = 0 + 1 + 2 + 1 + 2 + 3 + 3 = 12$

The external path length, $L_E = 2 + 3 + 3 + 2 + 4 + 4 + 4 + 4 = 26$

Note that, $L_I + 2 * n = 12 + 2 * 7 = 12 + 14 = 26 = L_E$

Thus, $L_I + 2n = L_E$, where n is the number of internal nodes. Now if the tree has n external nodes and each external node is assigned a weight, then the weighted path length P is defined as the sum of the weighted path lengths.

Therefore, $P = W_1L_1 + W_2L_2 + \ldots + W_nL_n$

where W_i and L_i are the weight and path length of an external node N_i.

Scanned by CamScanner

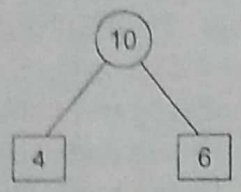


Figure 1.26 Binary tree with nodes q_1 and q_2

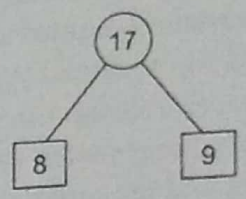


Figure 1.27 Binary tree with nodes q_3 and q_4

Step 3: Consider the next two minimum nodes from the list. The values of q3 and q4 are minimum. Construct a tree with q3 as left child and q4 as right child. Create an internal node with value as the sum of the values of left and right child, i.e., 17 as shown in Fig. 1.27. Replace q3 and q4 in the list with this value 17. The list now contains 10, 17, q5 = 15, and q6 = 28.

Step 4: Consider the next two minimum nodes from the list. The values 10 and q5 are minimum. Construct the tree with the internal node 10 as left child and q5 as right child. Create an internal node with value as the sum of the values of left and right child, i.e., 25 as shown in Fig. 1.28. Replace 10 and q5 in the list with this value 25. The list now contains 17, 25, and q6 = 28.

Step 5: Consider the next two minimum nodes from the list. The values 17 and 25 are minimum. Construct the tree with the internal node 17 as left child and 25 as right child. Create an internal node with value as the sum of the values of left and right child, i.e., 42 as shown in Fig 1.28. Replace 17 and 25 in the list with this value 42. The list now contains, 42 and q6 = 28.

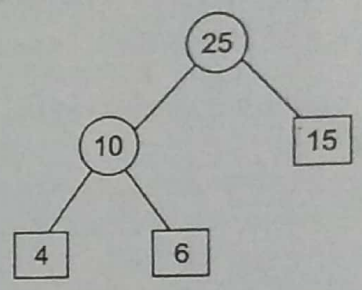


Figure 1.28 Binary tree after q5 is inserted

Step 6: Consider the next two minimum nodes from the list. The values 28 and 42 are minimum. Construct the tree with the internal node 42 as right child and 28 as left child. Create an internal node with value as the sum of the values of left and right child, i.e., 70 as shown in Fig. 1.30. Replace 28 and 42 in

Scanned by CamScanner

The WEPL = 4*4+6*4+15*3+8*3+9*3+28*1

= 16+24+45+24+27+28

= 164

Considering '0' represents the left child and '1' represents the right child of the Huffman tree as shown in Fig. 1.31, the Huffman Code for each of the character is known by traversing the Huffman tree from the root node to the character leaf node. The Huffman codes are q1=1100, q2=1101, q3=100, q4=101, q5=111, q6=0.

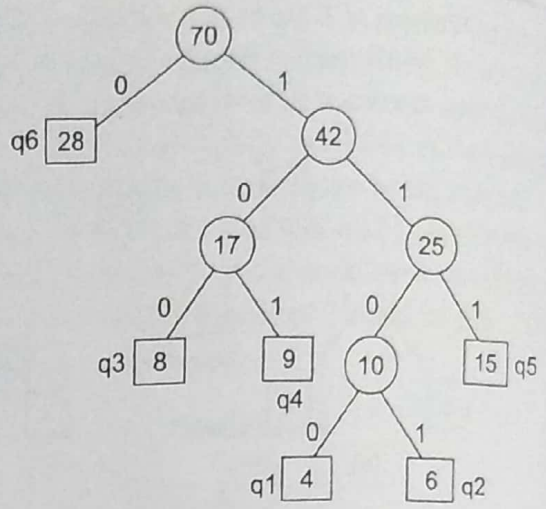


Figure 1.31 Huffman tree with codes

The algorithm to construct Huffman tree is shown in Fig. 1.32.

```
Step 1: Use the given characters in the list and create a leaf node for each character.
Step 2: Select two nodes with minimum values in the given list of characters.
Step 3: Construct a binary tree by creating a new internal node with value as the sum
of two minimum values from the list. The minimum value of the list is considered as a
left child and the next minimum value is considered as a right child. Replace the two
nodes selected in the list with the internal node value.
Step 4: Repeat Steps 2 and 3 until the list has only one value.
Step 5: EXIT
```

Figure 1.32 Algorithm for constructing Huffman tree

The time complexity for constructing a Huffman tree is `O(n log n)`, where `n` is the number of unique characters.

PROGRAMMING EXAMPLE

1. Write a C++ program to construct a Huffman tree.

```
#include<iostream>
#include<vector>
#include<string>
#include <bits/stdc++.h>
using namespace std;
class Huffman;
//A Tree Node
class node
{
  node *Left;
  node *Right;
  double value;
  string character;
  string Code;
  friend class Huffman;
};
class Huffman
{
```

Scanned by CamScanner

```
            cout<<"  "<<root1->character<<"\t\t"<<root1->Code<<endl;
    }
   else
    {
            //Appending 0 for left child
            root1->Left->Code =str.append("0");
            str.erase(str.end()-1);
            //Appending 1 for right child
            root1->Right->Code = str.append("1");
            str.erase(str.end()-1);
            BFS(root1->Left,str.append("0"));
            str.erase(str.end()-1);
            BFS(root1->Right,str.append("1"));
            str.erase(str.end()-1);
            }
            }
void Huffman::SetHuffmanCode()
{
            int size,i;
            double F;
            string tempString="";
            cout<<"Enter the number of characters to encode "<<endl;
            cin>>size;
            for(i=0;i<size;i++)
            {
                        cout<<"Enter character to be encoded and its value "<<endl;
                        node InternalNode;
                        cin>>tempString;
                        cin>>F;//Reading values
                        InternalNode.value=F;
                        InternalNode.character=tempString;
                        InternalNode.Left=NULL;
                        InternalNode.Right=NULL;
                        V.push_back(InternalNode);
            }
            node root = SetHuffmanTree();//build Huffman tree
            cout<<"Character"<<"\t"<<"Code"<<endl;
            BFS(&root,"");//visiting arbitrary nodes
}
int main()
{
            Huffman h;
            h.SetHuffmanCode();
            return 0;
}
```

Output

```
Enter the number of characters to encode!
6
Enter character to be encoded and its value
q1        4
Enter character to be encoded and its value
q2        6
Enter character to be encoded and its value
q3        8
Enter character to be encoded and its value
q4        9
```

Scanned by CamScanner

```
Enter character to be encoded and its value
q5        15
Enter character to be encoded and its value
q6        28
Character Code
q6                    0
q3                    100
q4                    101
q1                    1100
q2                    1101
q5                    111
```

Points to Remember

- External sorting is a term for a class of sorting algorithms that can handle massive amounts of data.
- In external sorting, each of the sorted block of records is called a run.
- k–way merge algorithms are a specific type of sequence merge algorithms that specialize in taking in multiple sorted lists and merging them into a single sorted list.
- In k–way merge, read, write and merge operations are performed in serial. In order to improve its efficiency, these operations are carried out in parallel by using buffer handling.
- The input delay in buffer handling can be overcome with the help of floating buffers.
- A winner tree is a complete binary tree which helps us to obtain the smallest key from a set of keys where each node represents the smaller of its two children.
- The weighted external path length (WEPL) of a tree T is the total merge time calculated by adding the product of weight of an external node and the depth of the external node from the root node.
- Using Huffman algorithm, a Huffman tree with minimum WEPL can be constructed.

Exercises

Review Questions

1. Write an algorithm for k–way merge sort.
2. Explain the step by step process of merging runs in k–way merge sort.
3. Write an algorithm for k–way merge sort with floating buffers.
4. Explain the importance of loser trees in run generation.
5. Explain the steps involved in constructing a Huffman tree.
6. Illustrate the importance of buffer handling for parallel operations in k–way merge sort.
7. Elaborate the usage of Huffman tree in the process of optimal merging of runs in k–way merge sort.
8. Consider an example of six characters q1 to q6 with the following values of each node q1=5, q2 = 9, q3 = 12, q4 = 13, q5 = 16. Construct the minimum binary tree using huffman algorithm.

Programming Exercises

1. Write a program to implement huffman tree.
2. Write a program to implement loser tree.

Multiple-choice Questions

1. In external sorting, number of runs that can be merged in every pass are called:
   (a) Degree of merging (b) Degree of passing
   (c) Degree of sorting (d) Degree of runs
2. Files that can fit in available buffer space in phase of external sorting must be read into:
   (a) Multilevel indexes (b) Search nodes
   (c) Main memory (d) Processing unit
3. Procedure of sorting algorithms for larger records that does not fit in main memory and are stored on disk is classified as:
   (a) Parser sorting (b) External sorting
   (c) Internal sorting (d) Secondary sorting

Scanned by CamScanner