UNIT- I : EXTERNAL SORTING

External Sorting, Introduction, Definition, 2-way Merging, 4-way merging, k-way merging, buffer handling for parallel operation, Run Generation, Optimal merging of Runs

---

Arranging the elements in ascending order or in descending order is called **Sorting**. Sorting techniques are broadly categorized into two.

a) Internal Sorting and
b) External Sorting.

Internal Sorting: All the records that are to be sorted are stored in main memory. The sorting algorithm that is used to sort records stored in main memory is said to be internal sorting.

Ex: **Bubble sort,**
Insertion sort,
Selection sort,
Quick sort,
Heap sort,
Merge sort etc..

Example 1) The basic merging algorithm takes two input arrays A and B an output array C and three counters Aptr, Bptr and Cptr which are initially set to the beginning of their respective arrays. The smaller of A[Aptr] and B[Bptr] is copied to the next entry in C, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to C.

A	B	C
→ 1	→ 2	→ 1
13	15	2
24	27	13
26	38	15
		24
		26
		27
		38

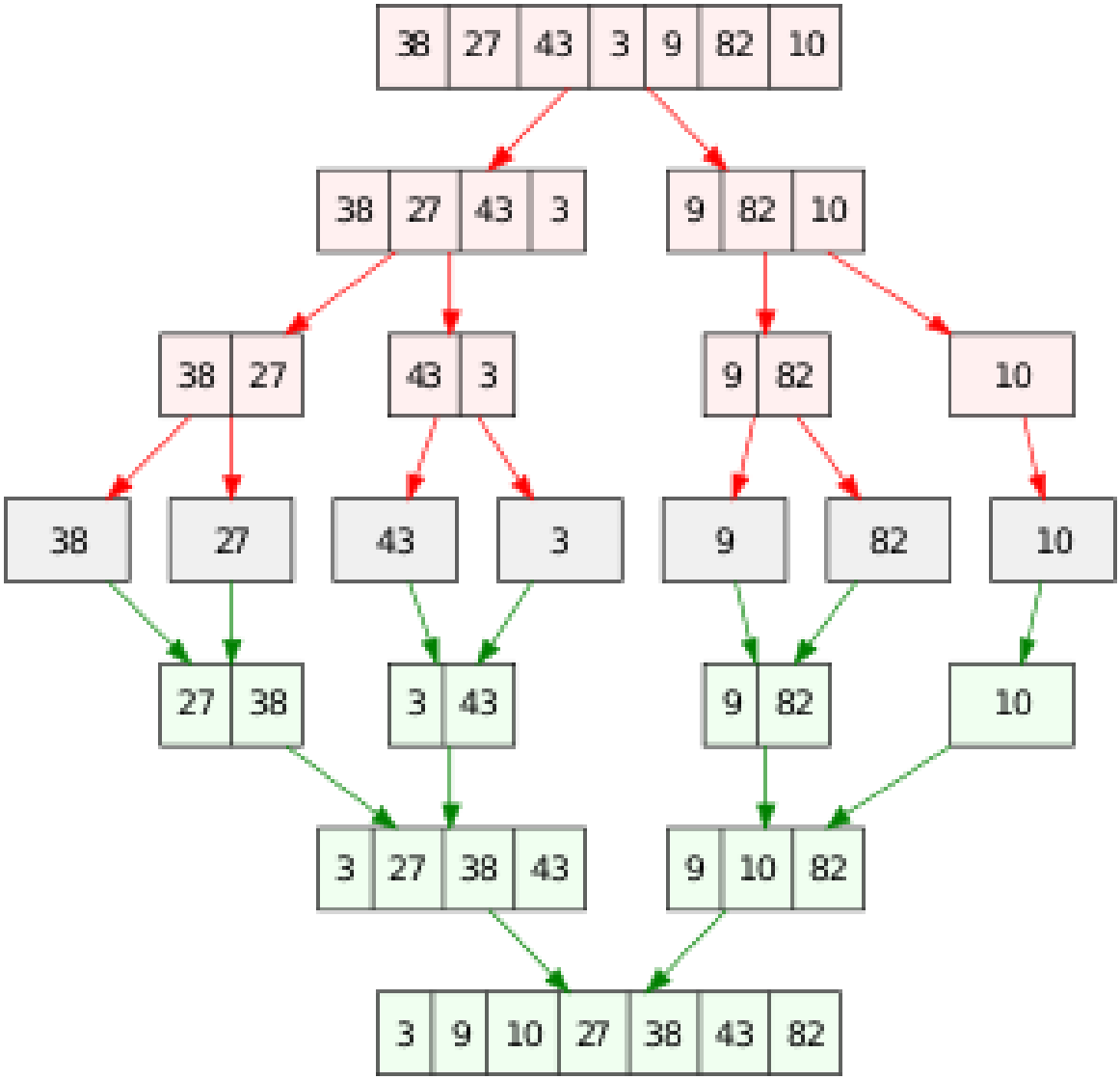
Example 2) Sorting an array using merge sort technique.

To sort an array using merge sort, go on splitting the array into sub arrays such that all the sub arrays are sorted.

We know that an array consisting of a single element is sorted.

So we split array into sub arrays of size 1.

Then we start merging sub arrays as shown below:

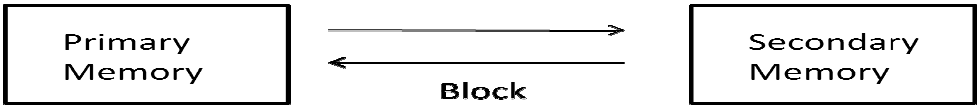


External Sorting: Some sorts that cannot be performed in main memory and must be done on disk or tape. This type of sorting is known as External Sorting.

External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.

Thus, external sorting algorithms are external memory algorithms and thus applicable in the external memory model of computation.

We assume that the lists to be sorted are so large that an entire list cannot be contained in the internal memory of a computer making an internal sorting impossible. In such cases we use external sorting.



The most popular method for sorting on external storage devices is merge sort. This method consists of two Phases :

First, segments of input list are sorted using internal sorting method. These sorted segments are known as runs are written onto the external storage.

Second, the runs generated in phase 1 are merged together.

Assume that the list (or file) to be sorted resides on a disk. The term block refers to the unit of data that is read from or written to a disk at one time. A block generally consists of several records. For a disk, there are three factors contributing to read/write time:

i) **Seek time:** Time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.

ii) **Latency time:** Time until the right sector of the track is under the read/write head.

iii) **Transmission time**: Time to transmit the block of data to/from the disk.

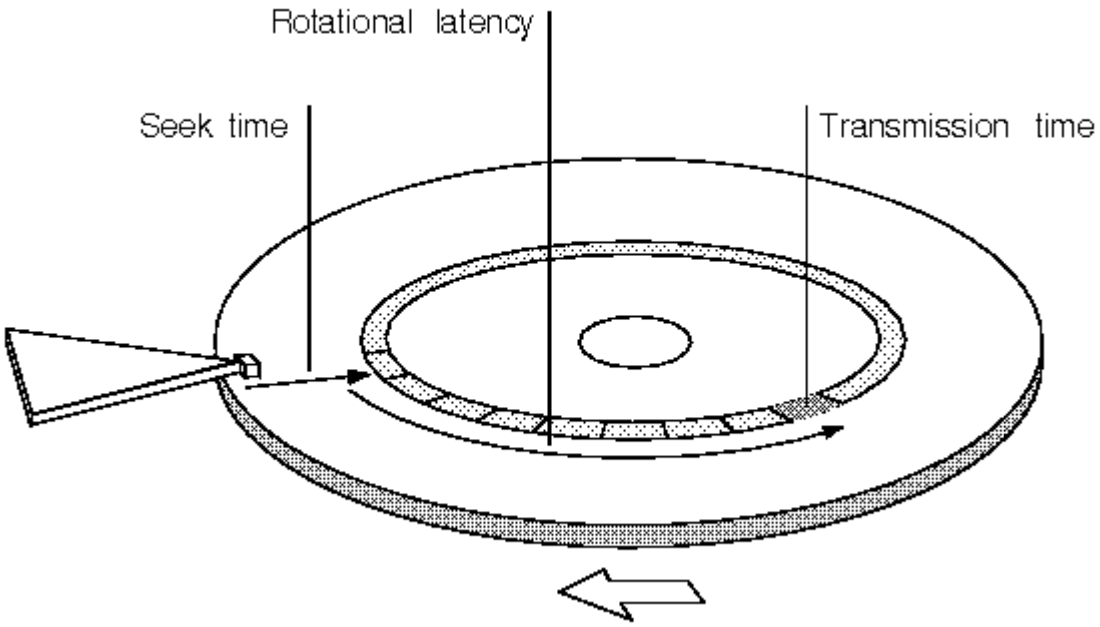
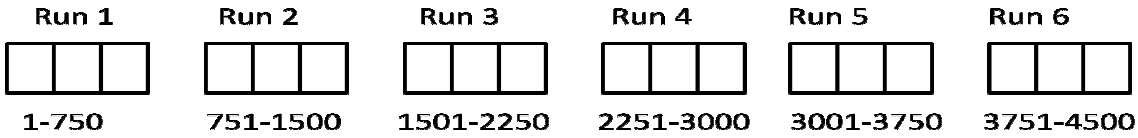


Fig) Secondary storage device- Disk- scheduling

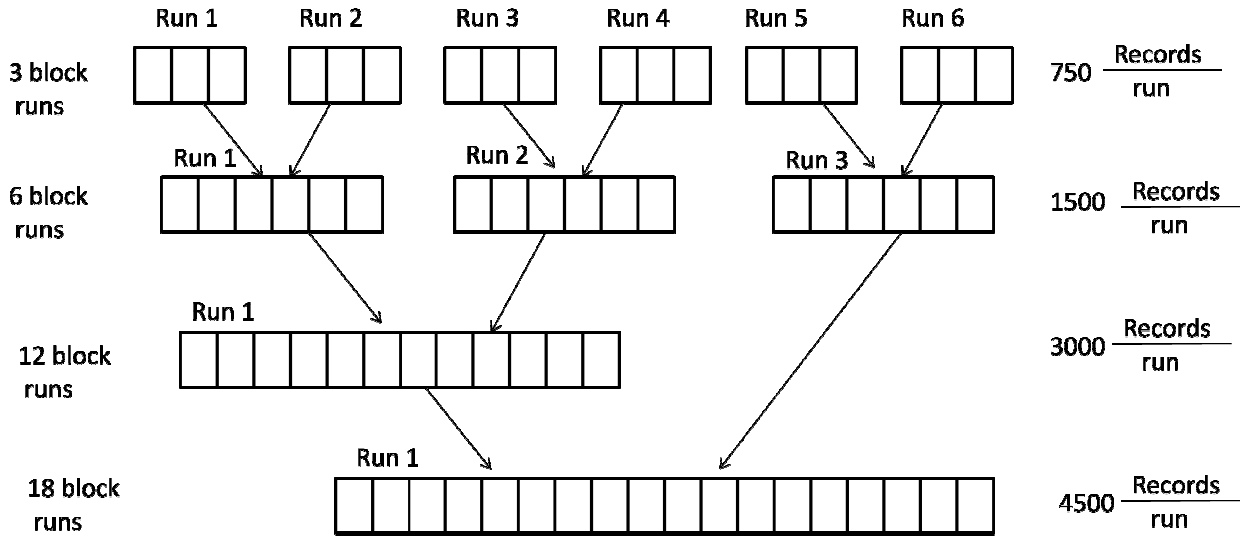
Example : A list containing 4500 records to be sorted. Assume that the internal memory is capable of sorting 750 records. The input file is maintained on disk and has a block length of 250 records.

The term block refers to the unit of data that is read from or written to a disk at one time. So, we can store 3 blocks at a time.



i) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs Run 1 to Run 6. A method such as heapsort or quicksort could be used. These six runs are written out onto the disk.

ii) Set aside three blocks of internal memory, each capable of holding 250 records. Two of these blocks will be used as input buffers and the third as an output buffer. Merge run-1 and run-2. This is carried out by first reading one block of each of these runs into input buffers. Blocks of runs are merged from the input buffers into the output buffer. When the output buffer gets full, it is written out onto disk. If an input buffer gets empty, it is refilled with another block from the same run. After run-1 and run-2 have been merged, run-3 and run-4 and finally run-5 and run-6 are merged. The result of this pass is 3 runs, each containing 1500 sorted records of 6 blocks. Recursively merging we can get desired sorted list.



2-way Merge with initial 6 Runs

A 2-way merge with 6 runs contains 4 levels and 3 passes.

	Levels	Passes
m=6	$\log_2 6 + 1$	$\log_2 6$
k=2	$\log_2 2^3 + 1$	$\log_2 2^3$
levels= $\log_k m+1$	3+1	3
passes= $\log_k m$		

Analysis of External Sorting to sort 4500 records:

Notations:

t_s = maximum seek time
t_l = maximum latency time
t_{rw} = time to read or write one block of 250 records
$t_{IO} = t_s + t_l + t_{rw}$
t_{IS} = time to internally sort 750 records
n t_m = time to merge n records from input buffers to the output buffer

The computing time for the various operations are:

SNo	Operation	Time
1	Read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$ write 18 blocks, $18t_{IO}$	$36\ t_{IO} + 6\ t_{IS}$
2	merge runs 1-6 in pairs	$36\ t_{IO} + 4500\ t_m$
3	merge 2 runs of 1500 records each, 12 blocks	$24\ t_{IO} + 3000\ t_m$
4	merge one run of 3000 records with one run of 1500 records	$36\ t_{IO} + 4500\ t_m$
	Total Time	$132\ t_{IO} + 12000\ t_m + 6\ t_{IS}$

2-way Merging/ Basic External Sorting Algorithm

Assume unsorted data is on disk at the beginning.
Let M=maximum number of records that can be stored & sorted in internal memory at one time.

Algorithm:

Step 1. Read M records into main memory & sort internally.
Step 2. Write this sorted sub-list into disk. (This is one "run").

Repeat step 1 and step 2 until data is processed into runs.

Step 3 . Merge two runs into one sorted run .
Step 4. Write this single run back onto disk

Repeat step 3 and step 4 until all runs processed and stored back in secondary memory.
Merge runs again as often as needed until only one large run: The sorted list.

Buffer Handling

2-way merge requires 3 buffers
2 input buffers and one output buffer

Two runs are going to merge into single array
Elements of run 1 will be placed in first input buffer
Elements of run 2 will be placed in second input buffer
Smallest of input buffers will be placed into output buffer
Content of output buffer will be sent back to disk.

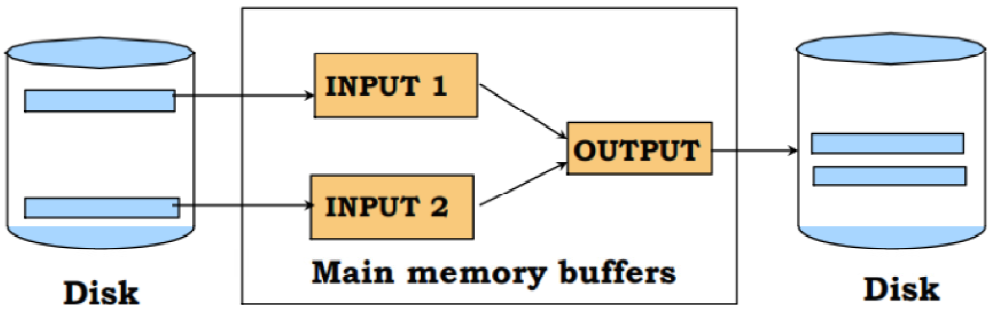


Fig) Buffer Handling

Example-1: Sorting unsorted array using 2 - way merge

Unsorted data on disk

81 94 11 96 12 35 17 99 28 58 41 75 15

Assume M = 3 (M would actually be much larger, of course.) First step is to read 3 data items at a time into main memory sort them and write them back to disk as runs of length 3.

Run 1	Run 2	Run 3	Run 4	Run 5
81 94 11	96 12 35	17 99 28	58 41 75	15

Sort runs and store back in to disk

Run 1	Run 2	Run 3	Run 4	Run 5
11 81 94	12 35 96	17 28 99	41 58 75	15

Next step is to merge the runs of length 3 into runs of length 6.

Run 1	Run 2	Run 3
11 12 35 81 94 96	17 28 41 58 75 99	15

Next step is to merge the runs of length 6 into runs of length 12

Run 1	Run 2
11 12 17 28 35 41 58 75 81 94 96 99	15

Next step is to merge the runs of length 12 into runs of length 24. Here we have less than 24, so we're finished. Sorted list is on disk.

11 12 15 17 28 35 41 58 75 81 94 96 99

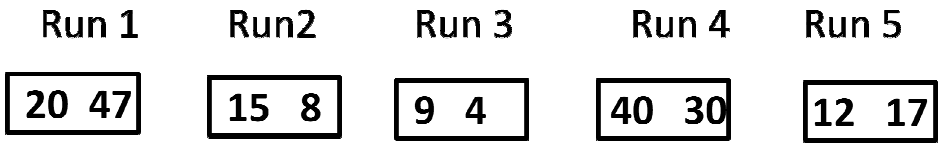
Final sorted Array

Example-2 : Sorting unsorted array using 2 - way merge

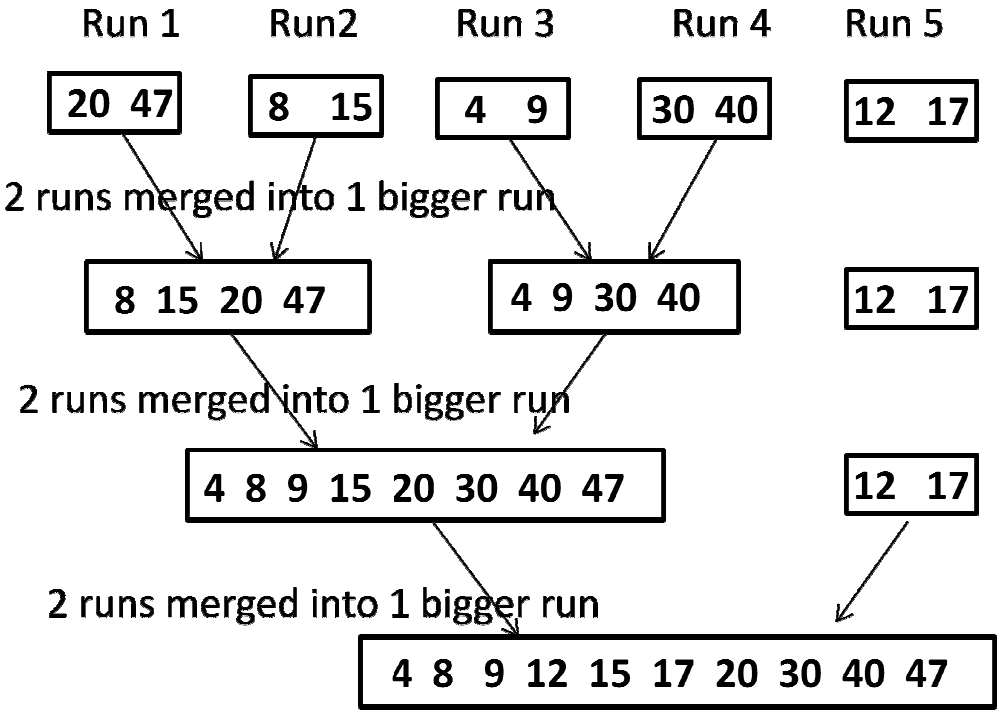
Unsorted array in the disk : 20 47 15 8 9 4 40 30 12 17

Assume that every block can store 2 data values.

The input array is divided into 5 runs



Individual runs are sorted and stored back in Disk.



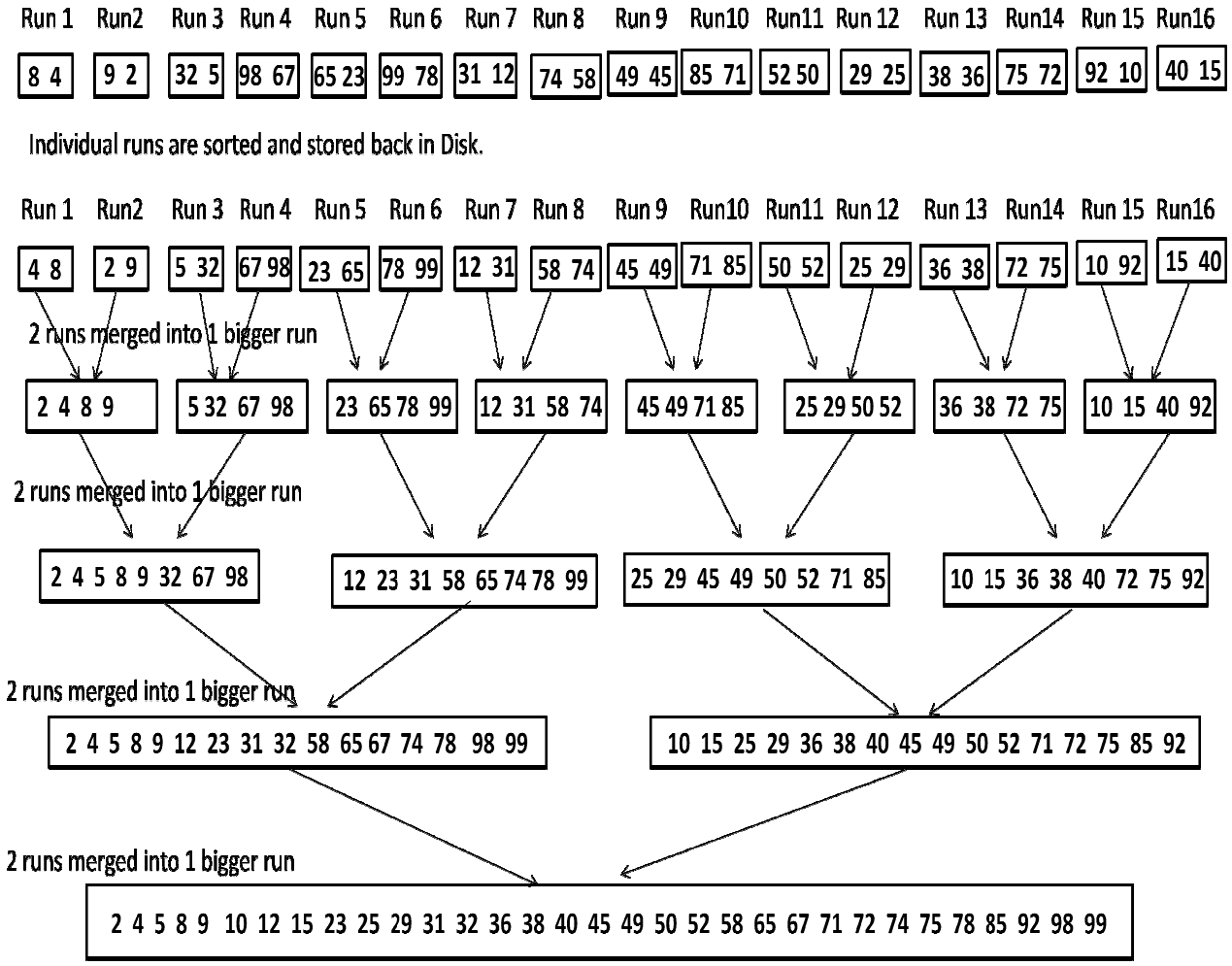
Final sorted array

Example-3) 2-way merge sort :: Consider the following data file containing 32 data items stored in a disk.

8 4 9 2 32 5 98 67 65 23 99 78 31 12 74 58 49 45 85 71 52 50 29 25 38 36 75 72 92 10 40 15

The file contains 32 data items. Assume that we send 2 data items into main memory to generate runs. Initially every run may contain 2 data items. As the input file contains 32 data items and divided into 16 runs. Each run has 2 data items. Each run is sorted and again stored back in secondary storage device.

In every level 2 runs will be merged to form a bigger run.



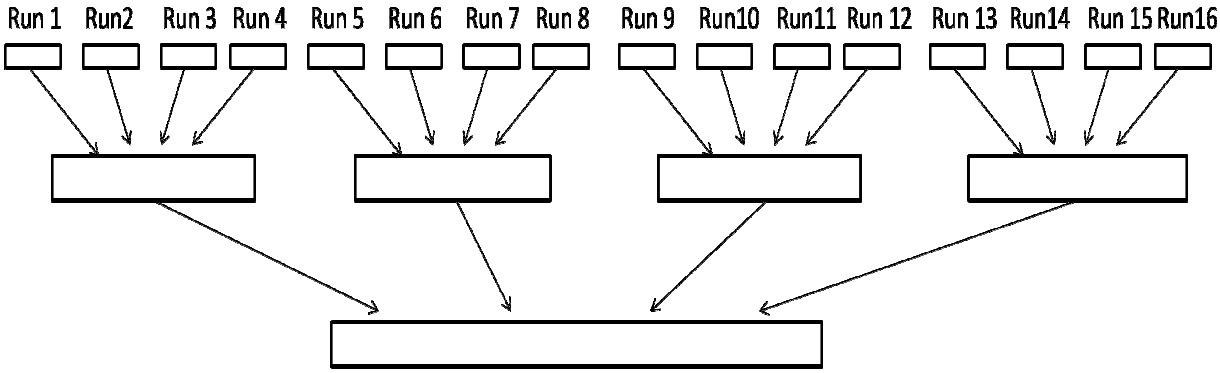
Final sorted file stored in disk using 2-way merge sort

K - Way merging

The two way merge function was just described. If we start with m runs, the merge tree will have $\log_2 m+1$ levels and $\log_2 m$ passes.

The number of passes over the data can be reduced by using a higher order merge. A K-way merge on m runs requires $\log_k m + 1$ levels and $\log_k m$ passes. Thus the input or output time may be reduced by using a higher order merge.

Assume that there is a big data file which can divided into 16 runs or 16 sub files. A 4-way merge can be described as follows.



A 4-way merge on 16 runs

If we start with m runs it will lead to 3 levels and 2 passes

Levels = $\log_k m + 1$
$\log_4 16+1$
$\log_4 4^2+1$
$2+1$
3

passes = $\log_k m$
$\log_4 16$
$\log_4 4^2$
2

m=16 **k=4**

Instead of a 2-way merge, let us discuss how to do a K-way merge. The number of passes over the data can be reduced by using a higher order merge, i.e., k-way merge for k >= 2. In this case we would simultaneously merge k runs together.

The number of passes over data for 16 runs using 2-way merge is 4 where as the number of passes over data for 16 run using k-way merge is 2. In general k-way merge on m runs requires $\log_k m$ passes over data where as 2-way merge on m runs requires $\log_2 m$. Thus, the input/output time may be reduced by using a higher-order merge.

Algorithm k-way merge:

Step 1. Read M values at a time into internal memory, sort, and write as runs on disk

Step 2. Merge K runs:

i. Read first value on each of the k runs into internal array and build min heap

ii. Remove minimum from heap and write to disk

iii. Read next value from disk and insert that value on heap

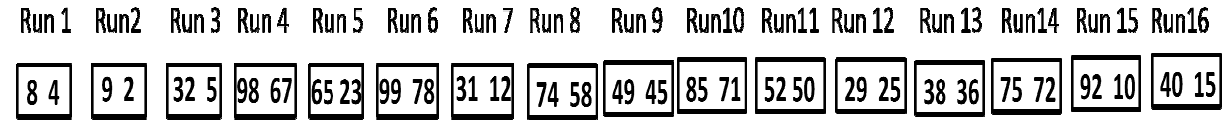
Repeat steps until all first K runs are processed

Repeat merge on larger & larger runs until have just one large run: sorted list.

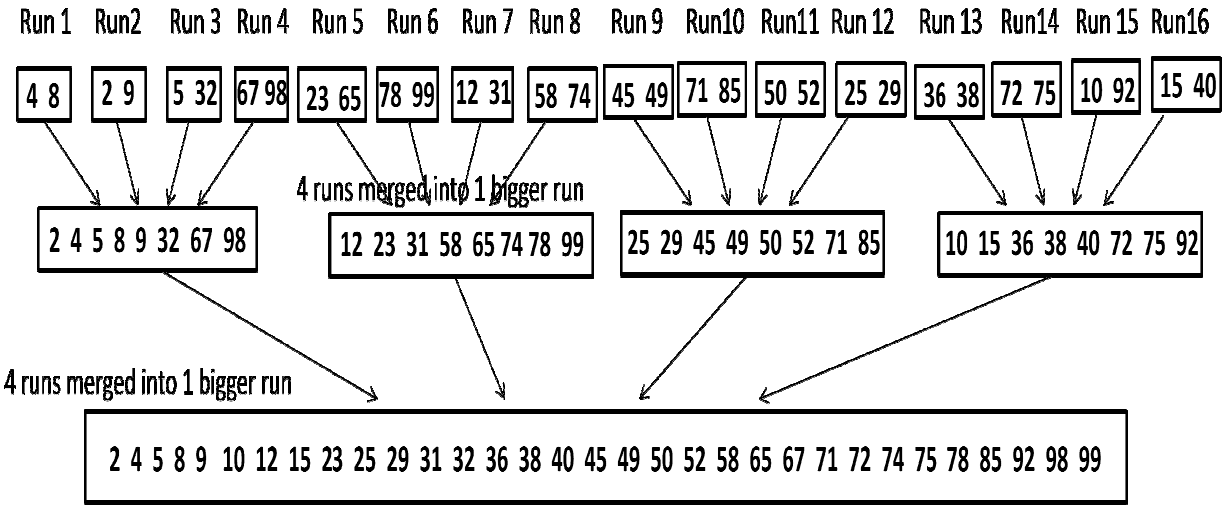
Example: Let us take the unsorted data file stored in secondary storage device

8 4 9 2 32 5 98 67 65 23 99 78 31 12 74 58 49 45 85 71 52 50 29 25 38 36 75 72 92 10 40 15

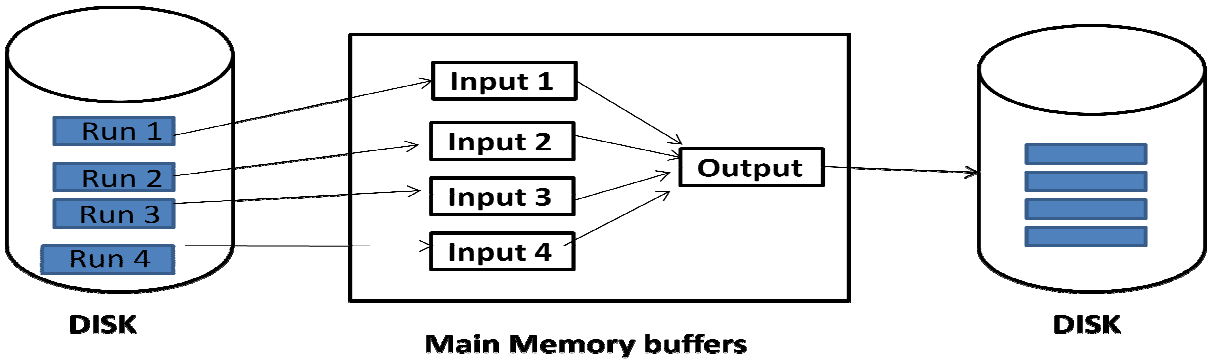
The file contains 32 data items. Assume that we send 2 data items into main memory to generate runs. Initially every run may contain 2 data items. As the input file contains 32 data items and divided into 16 runs. Each run has 2 data items. Each run is sorted and again stored back in secondary storage device.



Individual runs are sorted and stored back in Disk.



Buffer handling



In going to a higher order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_k m$ passes. This is so because the number of input buffers needed to carry out a k-way merge increases with k.

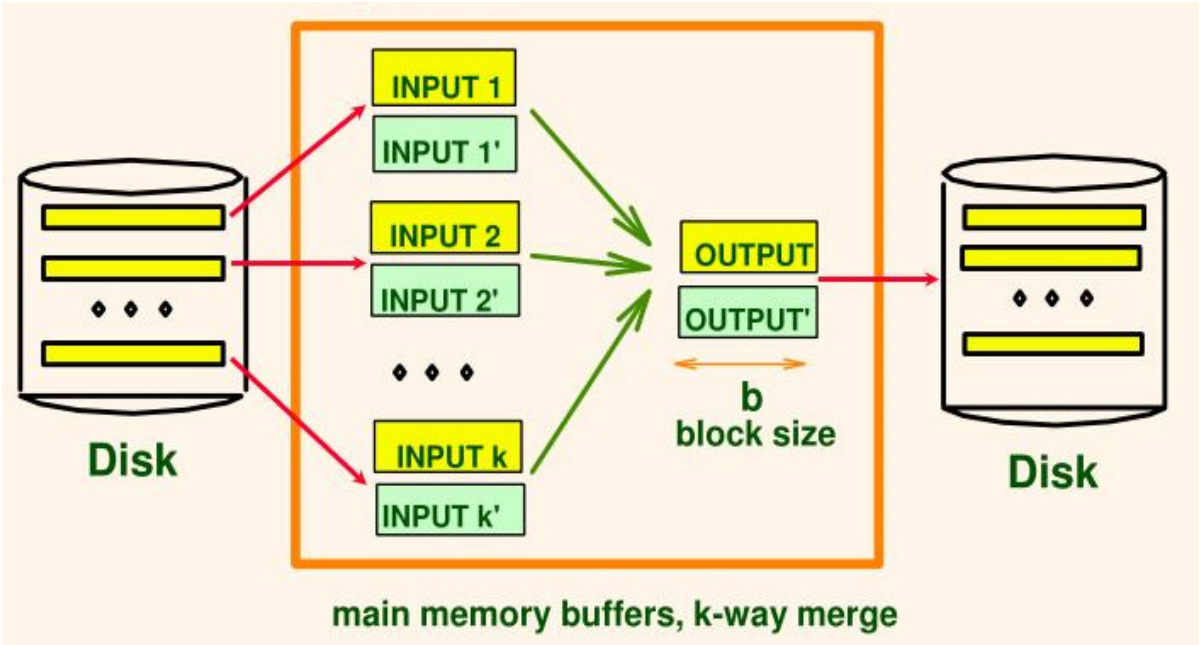
Though k + 1 buffers are sufficient, the use of 2k + 2 buffers is more desirable. The optimal value for k clearly depends on disk parameters and the amount of internal memory available for buffers.

The total time needed per level of merge tree is $O(n \log_2 k)$

Number of levels = $O(\log_k m)$

The asymptotic internal processing time becomes $O(n \log_2 k \ \log_k m)=O(n \log_2 m)$

This is independent of k.



Buffer handling for parallel operation

In k-way merge we clearly need k input buffers and one output buffer to carry out the merge. This however, is not enough to input, output and internal merging are to be carried out in parallel.

For instance, while the output buffer is being written out, internal merging has to be halted, since there is no place to collect merged records. This can be overcome through the use of two output buffers. While one is being written out, records are merged into the second.

Example: assume that a two way merge is carried out using four input buffers.
in[i] 0 <= i <= 3 i.e. in[0], in[1], in[2] and in[3].
and two output buffers ou[0] and ou[1].
Each buffer is capable of holding 2 records.

The first few records of run 0 have key values 1, 3, 5, 7, 8, 9.
The first few records of run 1 have key values 2, 4, 6, 15, 20, 25.
Buffers in[0] and in[2] are assigned to run 0
Buffers in[1] and in[3] are assigned to run 1

We start the merge by reading in one buffer load from each of the two runs. At this time the buffers have the configuration of fig (a).

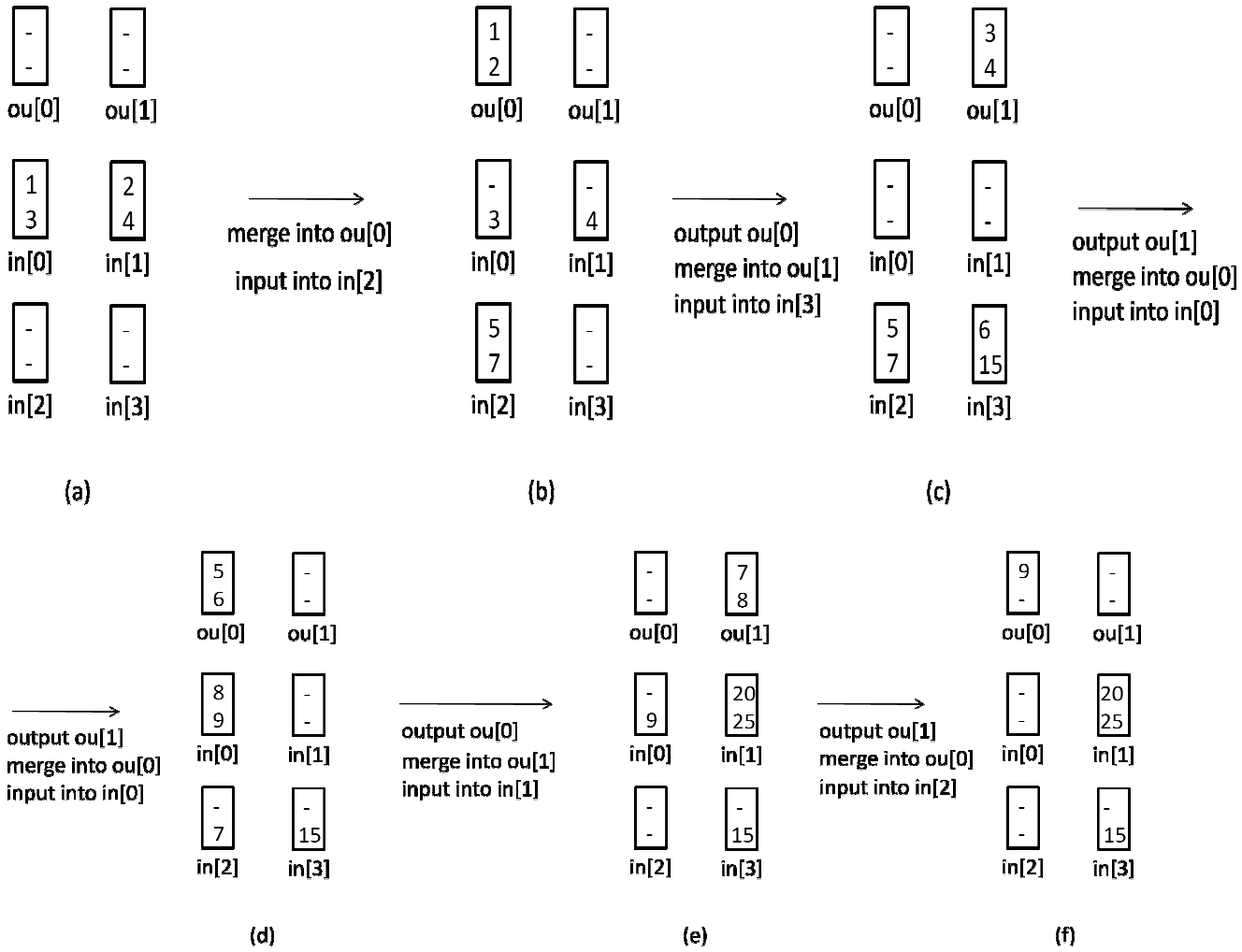
Now run 0 and run 1 are merged using records from in[0] and in[1]. In parallel with this, the next buffer load from run 0 is input

When ou[0] is full , we have the situation of fig (b).
Next we simultaneously output ou[0], input into in[3] from run 1 and merge into ou[1]. When ou[1] is full we have the situation of fig (c).

Continuing in this way, we reach the configuration of fig (e). we now begin to output ou[1] , input from run 0 into in[2] and merge into ou[0]. During the merge all records from run 0 gets used before ou[0] gets full.

Merging must now be delayed until the inputting of another buffer load from run 0 is completed.

The first few records of run 0 have key values **1, 3, 5, 7, 8, 9.**
The first few records of run 1 have key values **2, 4, 6, 15, 20, 25.**
Buffers in[0] and in[2] are assigned to run 0
Buffers in[1] and in[3] are assigned to run 1



Example showing that two fixed buffers per run are not enough for continued parallel operation.

In the above example, it is clear that 2k input buffers are fixed, then we cannot assign two buffer per run. The buffer must be floating in the sense that an individual buffer may be assigned to any run depending upon need. In buffer assignment strategy, there will be at any time be at least one input buffer containing records from each run. The remaining buffers will be filled on priority basis.

k-way merge with floating buffers algorithm:

This algorithm merges k-runs, k>=2, using a k-way merge. 2k input buffers and 2 output buffers are used. Each buffer is a contiguous block of memory. Input buffers are queued in k queues, one queue for each run. It is assumed that each input/output buffer is long enough to hold one block of records. Empty buffers are placed on a linked stack. This algorithm assumes that the end of each run has sentinel record with a very large key, say $+\infty$. It is assumed that all the record key value less than that of the sentinel record. Time taken to merge, time to output block equals the time to read a block, then almost all input, output and computation will be carried out parallel.

k-way merge algorithm with floating buffers

Step 1: Input the first block of each of the k runs, setting up k linked queues, each having one block of data. Put the remaining k input blocks into a linked stack of free input blocks. Set OU to 0.

Step 2: Let lastkey[i] be the last key input from run i. Let nextRun be the run for which lastkey is minimum. If lastkey[nextRun]!=+∞, then initiate the input of the next block from run nextRun.

Step 3: Use a function k-wayMerge to merge records from k input queues into the output buffer OU. Merging continues until either the output buffer gets full or a record with key +∞ is merged into OU. If, an input buffer becomes empty before output buffer gets full, k-way merge advances to the next buffer on same queue and returns the empty buffer to the stack of empty buffer. If an input buffer becomes empty at the same time as the output buffer gets full, the empty buffer is left on the queue, k-wayMerge does not advance to the next buffer on queue. Merge terminates.

Step 4: Wait for any ongoing disk input/output to complete.

Step 5: If an input buffer has been read, add it to the queue for appropriate run. Determine the next run to read from determining NextRun such that lastKey[nextRun] is minimum.

Step 6: If lastKey[nextRun]!=+∞, the initiate reading the next block from run nextRun into a free input buffer.

Step 7: Initiate the writing of output buffer ou. Set ou to 1 - ou.

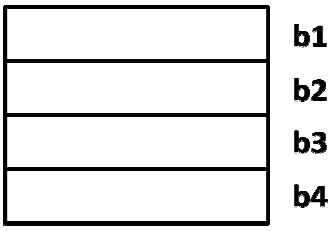
Step 8: If a record with key +∞ has been not been merged into the output buffer, go back to step 3. Or wait for the ongoing write to complete and then terminate.

Example to illustrate k-way merge with floating buffers algorithm:

Example 1) 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 +∞. The blocks in Run1 and Run2 are as follows:

Run 1	2 4	6 8	9 10	+∞
Run 2	3 5	7 16	21 26	+∞

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. Let N be the run from which the next block is to be loaded and s be the buffer size. Here, s = 2.



Stack of 4 floating buffers

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.

The process of 2–way merge with floating buffers is shown in Fig 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.

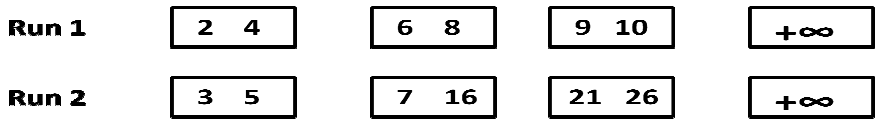
Line 4: The minimum keys (6 and 7) in the input buffers are merged into the output buffer and the next record in Run1 is loaded into the popped buffer b1 and is added to the linked queue atRun1. Now N=1, as the LastKey[1] is minimum. The empty buffer b2 is pushed back on to the stack. As the output buffer 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.



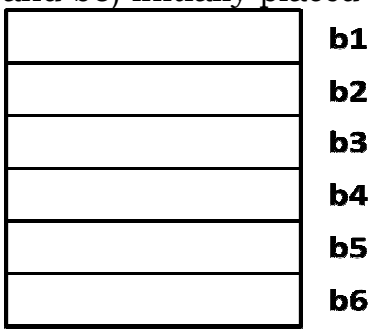
Line	Run 1	Run 2	OUTPUT	Next Block From
1	2 4	3 5	No Output	Run 1
2	4 → 6 8	5	2 3	Run 2
3	→ 6 8	→ 7 16	4 5	Run 1
4	8 → 9 10	16	6 7	Run 1
5	→ 10 → M	16	8 9	Run 2
6	→ M	→ 21 26	10 16	Run 2
7	M	→ M	21 26	No Next
8			M M	------

Fig) The status of linked queues for 2 runs

Example 2) 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 +∞. The blocks in Run1, Run2, and Run3 are as follows:

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

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.



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. 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 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

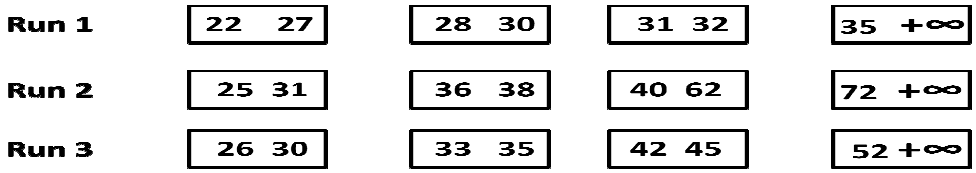
Line 9: The minimum keys (38 and 40) in the input buffers are merged into the output buffer and the next record in Run3 is loaded into the popped buffer from the stack and is added to the linked queue at Run3. Now set N=2, as the LastKey[2] is minimum. The empty buffer in the linked queue at Run2 is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

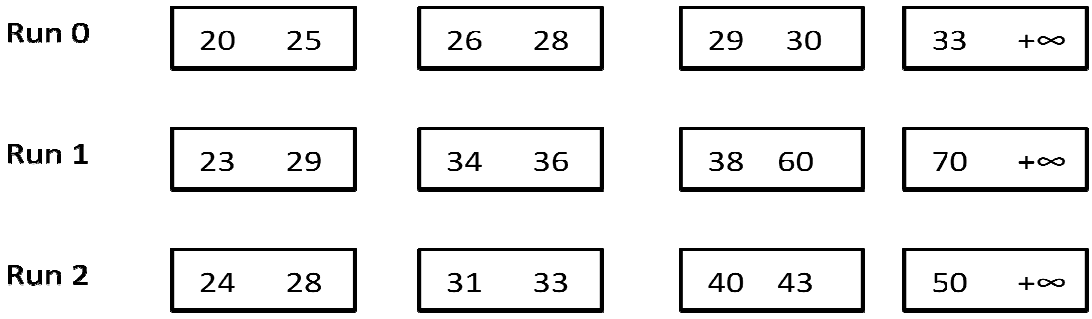


Line	Run 1	Run 2	Run 3	OUTPUT	Next Block From
1	22 27	25 31	26 30	No Output	Run 1
2	27 → 28 30	31	26 30	22 25	Run 1
3	→ 28 30 → 31 32	31	30	26 27	Run 3
4	→ 31 32	31	30 → 33 35	28 30	Run 2
5	32	31 → 36 38	→ 33 35	30 31	Run 1
6	→ 35 M	→ 36 38	33 35	30 31	Run 3
7	M	36 38	35 → 42 45	33 35	Run 2
8	M	38 → 40 62	→ 42 45	35 36	Run 3
9	M	→ 62	42 45 → 52 M	38 40	Run 2
10	M	62 → 72 M	→ 52 M	42 45	No Next
11	M	→ 72 M	M	52 62	No Next
12		M	M	72 M	-----

Figure) Status of linked queues for 3 runs

Example 3) To illustrate k-way merge with floating buffers algorithm:

Let us consider three way merge on three runs. Each run consists of four blocks of two records each; the last key in the fourth block of each of these three runs is +∞. We have six input buffers and two output buffers.



Line	Queue	Run 0	Run 1	Run 2	Output	NextBlock from
1		20 25	23 29	24 28	No Output	Run 0
2	25	→ 26 28	29	24 28	20 23	Run 0
3		→ 26 28 → 29 30	29	28	24 25	Run 2
4		→ 29 30	29	28 → 31 33	26 28	Run 1
5		30	29 → 34 36	31 33	28 29	Run 0
6		→ 33 M	34 36	31 33	29 30	Run 2
7		M	34 36	33 → 40 43	31 33	Run 1
8		M	36 → 38 60	40 43	33 34	Run 2
9		M	60	40 43 → 50 M	36 38	Run 1
10		M	60 → 70 M	50 M	40 43	No next
11	M		→ 70 M	M	50 60	No next
12			M	M	70 M	

Run Generation

Using conventional internal sorting methods, it is possible to generate runs that are only as large as the number of records that can be held in internal memory at one time.

Run generation is the first phase of external-memory sorting, where the objective is to scan through the data, reorder elements using a small buffer of size M , and output runs (contiguously sorted chunks of elements) that are as long as possible. i.e. **Segments of the input file are sorted using a good internal sort method. These sorted segments, known as runs, are written out onto external storage as they are generated.**

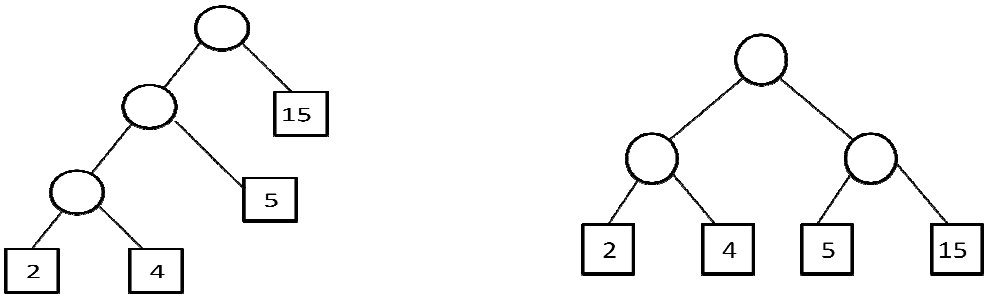
Using a tree of loosers, it is possible to do better than this. This algorithm was devised by Walters, Painter and Zalk. This algorithm generates runs that are twice as long as obtainable by conventional methods (long runs). In addition, this algorithm will allow for parallel input, output and internal processing.

Optimal Merging of Runs

The runs generated by function may not be of the same size.

When runs are of different size, the run merging strategy employed so far does not yield minimum run times. For Example suppose we have **4 runs** of length **2 4 5 and 15** respectively.

Some possible two way merge trees



(a) (b)

Possible two way merges (Merge Trees)

In the above figure circular nodes as internal nodes and the square nodes as external nodes. The **total merge time** is obtained by **summing the products of the run lengths and the distance from the root** of the corresponding external nodes. This sum is called the weighted external path length.

The circular nodes represent a two way merge using as input the data of the children nodes.
The square nodes represent the initial runs.

In the first merge tree, we begin by merging the runs of size 2 and 4 to get run of size 6. This is merged with 5 to get a run size of 11. Finally this run of size 11 is merged with run of size 15 to get desired sorted run of size 26.

This sum is called the weighted external path length

First tree

= 2x3+4x3+5x2+15x1
= 6 + 12 + 10 + 15
= 43

Second tree

= 2x2 + 4x2 + 5x2 + 15x2
= 4 + 8 + 10 + 30
= 52

The total merge time of fig (a) is less than the total merge time of fig (b).

There is good solution to the problem of finding a binary tree with minimum weighted external path. D.Huffman has given the solution. Huffman tree gives minimum weighted external path.

This solution begins with min heap of n single node trees. The single node in each of these trees represents one of the provided qi's and its weight is qi. Huffman's algorithm then repeatedly extracts two minimum weight trees a and b from min heap, a combines them into a single binary tree c by creating a new root whose left and right sub trees are a and b, and inserts c into the min-heap. After n-1 rounds of this extract, combine, insert process, the minheap will be left with single binary tree which is asserted to be a binary tee with minimum weighted external path length.

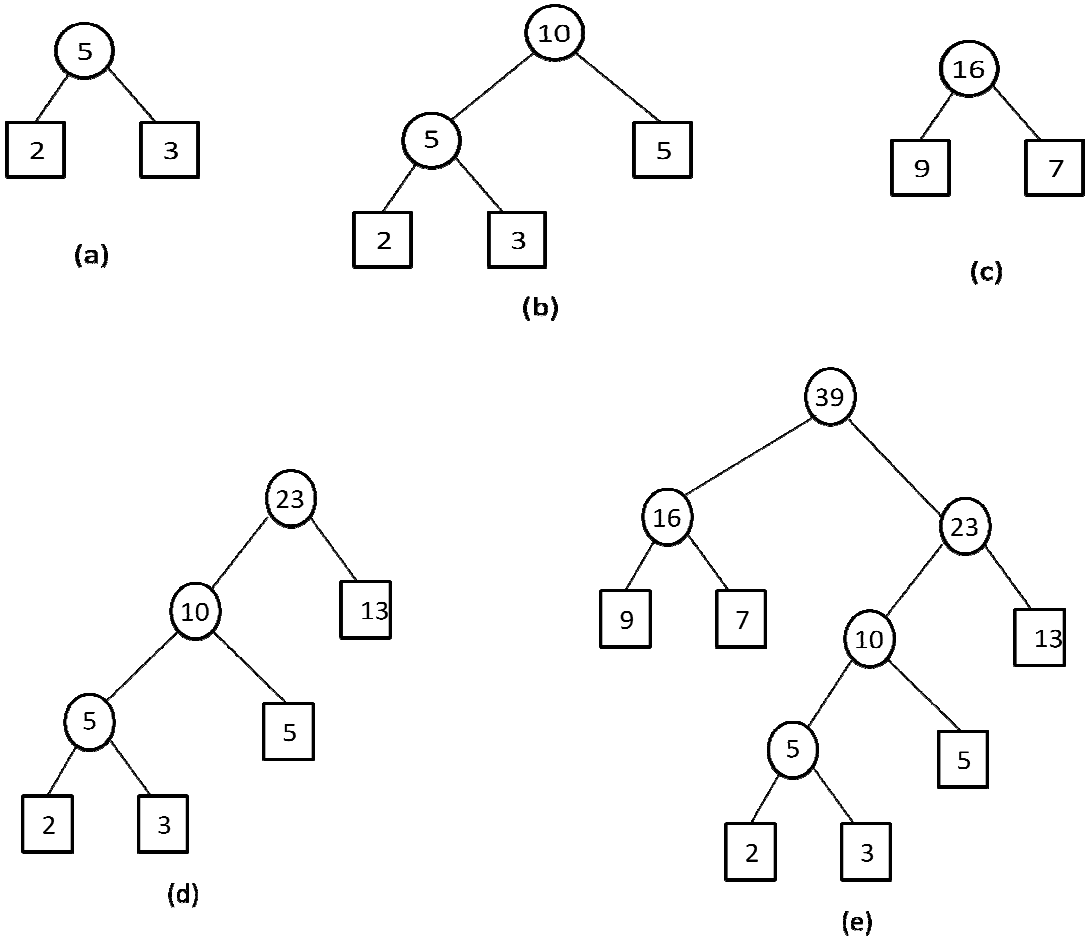
Huffman Function

```
template <class T>
void Huffman ( MinHeap<TreeNode<T>*> heap, int n)
{
   for (int i=0;i<n-1;i++)
    {
       TreeNode<T> *first=heap.Pop();
       TreeNode<T> *second=heap.Pop();
       TreeNode<T>*bt=new BinaryTreeNode<T>(first, second, first.data+second.data);
       Heap.Push(bt);
    }
}
```

Example:

Suppose we have the weights **q1 = 2 q2=3 q3=5 q4=7 q5=9 q6=13**

The sequence of trees we would get is given. The number in circle represents the sum of weights of external nodes



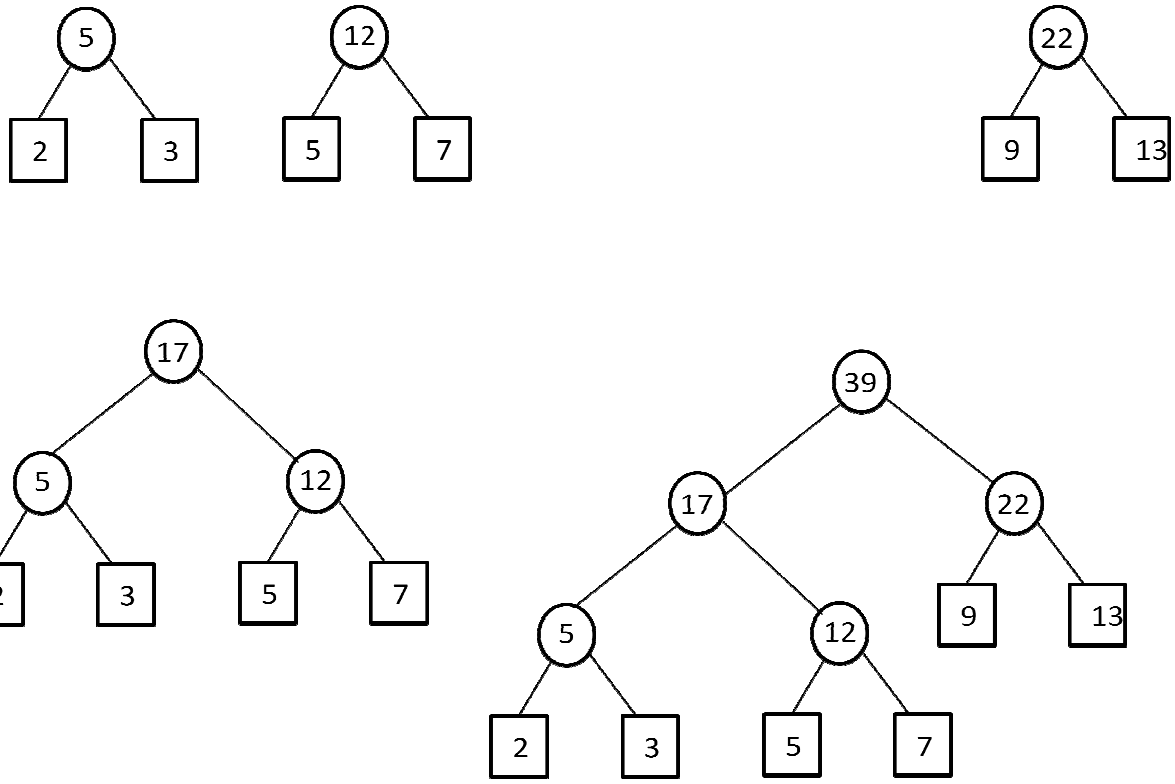
Construction of a Huffman tree

The weighted external path length of this tree is
= 2x4 + 3x4 + 5x3 + 13x2 + 9x2 + 7x2
= 8 + 12 + 15 + 26 + 18 + 14
= 93

Heap initialization takes O(n) times
The main for loop executes n-1 times
Each call to push and pop requires O(log n) times.
Hence, the asymptotic computing time for the algorithm is O(n log n).

Weighted external path lengths of a Complete Binary Tree

q1 = 2 q2=3 q3=5 q4=7 q5=9 q6=13



Complete binary tree

The weighted external path length of this tree is
= 2x3 + 3x3 + 5x3 + 7x3 + 9x2 + 13x2
= 6 + 9 + 15 + 21 + 18 + 26
= 95

By comparison, the best complete binary tree has weighted path length is 95.

Analysis of Huffman:

The main loop is executed n-1 times.
Each call to Pop and Push requires 0(log n) time.
Hence asymptotic computing time is 0(n log n).

QUESTIONS FROM PREVIOUS QUESTION PAPERS

1) Define External sorting

External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.

Thus, external sorting algorithms are external memory algorithms and thus applicable in the external memory model of computation.

In internal sorting all the data to sort is stored in memory at all times while sorting is in progress. In external sorting data is stored outside memory (like on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into main memory entirely.

The most popular method for sorting on external storage devices is merge sort. This method consists of two Phases:

First, segments of input list are sorted using internal sorting method. These sorted segments are known as runs are written onto the external storage.

Second, the runs generated in phase 1 are merged together.

2) Why it is difficult to adopt internal sorting methods for external sorting?

Internal sorting methods are used on data that is stored in only one table completely stored in the main memory. Whereas external sorting the data to be sorted is stored on a secondary storage device and the entire data is not fit in the main memory at once. So we need to have separate sorting techniques other the internal sorting methods to sort and store data in secondary storage devices.

3) Write and explain k-way merge algorithm with floating buffers.

Pages 15 to 21

4) Discuss the drawbacks of k-way merging with higher k-values

Page 9 &10

5) Prove that the asymptotic internal processing time of k-way merge is independent of k.

Page 12

6) Write and explain external merge sort algorithm.

External sorting

External sorting is a technique in which the data is stored on the secondary memory, in which part by part data is loaded into the main memory and then sorting can be done over there. Then this sorted data will be stored in the intermediate files. Finally, these files will be merged to get a sorted data. Thus by using the external sorting technique, a huge amount of data can be sorted easily. In case of external sorting, all the data cannot be accommodated on the single memory, in this case, some amount of memory needs to be kept on a memory such as hard disk, compact disk and so on.

The requirement of external sorting is there, where the data we have to store in the main memory does not fit into it. Basically, it consists of two phases that are:

1. **Sorting phase:** This is a phase in which a large amount of data is sorted in an intermediate file.

2. **Merge phase:** In this phase, the sorted files are combined into a single larger file.

Two-Way Merge Sort

Two-way merge sort is a technique which works in two stages which are as follows here:
Stage 1: Firstly break the records into the blocks and then sort the individual record with the help of two input tapes.

Stage 2: In this merge the sorted blocks and then create a single sorted file with the help of two output tapes.

By this, it can be said that two-way merge sort uses the two input tapes and two output tapes for sorting the data.

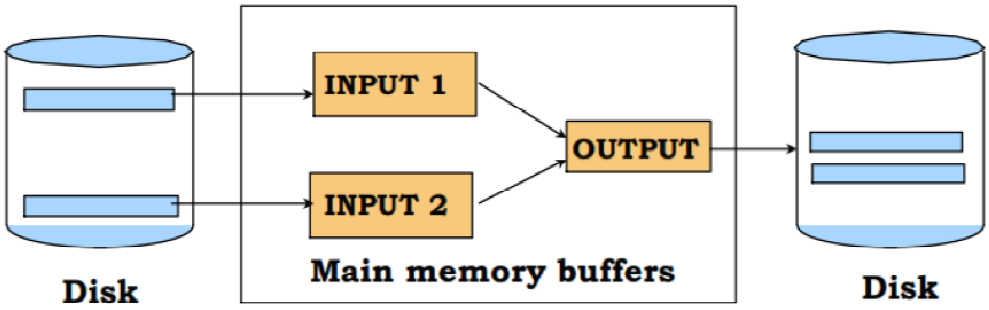
Algorithm for Two-Way Merge Sort:

Step 1) Divide the elements into the blocks of size M. Sort each block and then write on disk.

Step 2) Merge two runs

1. Read first value on every two runs.
2. Then compare it and sort it.
3. Write the sorted record on the output tape.

Step 3) Repeat the step 2 and get longer and longer runs on alternates tapes/disk. Finally, at last, we will get a single sorted list.



Analysis: This algorithm requires log(N/M) passes with initial run pass. Therefore, at each pass the N records are processed and at last we will get a time complexity as O(N log(N/M).

7) Suppose an external sorting method involves merging 4 runs of lengths 2, 4, 5 and 15 respectively. Present all possible ways of 2-way and 3-way merging for those runs and identify the merge with optimal performance.

Page 22

8) Explain the need for external sorting.

External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

9) Assume a list containing 4500 records is to be sorted using a computer with internal memory capable of sorting at most 750 records at a time and the input list is maintained on a disk that has block length of 250 records. For this scenario explain how external sorting may be performed to accomplish the task.

Page 4 & 5

10) Derive an expression for the total time required to perform the external sorting mentioned in the above question, with a detailed note on each operation.

Page 5

11) What is the key difference between internal sorting and external sorting?

Page 3

12) Explain the buffering algorithm for k-way merge with floating buffers.

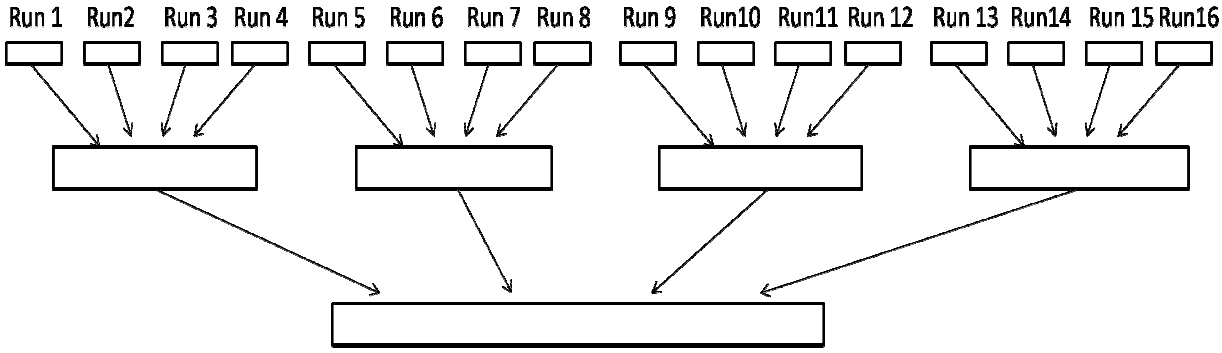
Page 15 to page 21

13) Show the Huffman function generates a binary tree of minimal weighted external path length.

Page 16 & 17

14) What is k-way merging? What are the pros and cons of having higher k-values? How to select a k-value to maximize the performance?

Instead of a 2-way merge, let us discuss how to do a K-way merge. The number of passes over the data can be reduced by using a higher order merge, i.e., k-way merge for k >= 2. In this case we would simultaneously merge k runs together.



A 4-way merge on 16 runs

The number of passes over data for 16 runs using 2-way merge is 4 where as the number of passes over data for 16 run using k-way merge is 2. In general k-way merge on m runs requires $\log_k m$ passes over data where as 2-way merge on m runs requires $\log_2 m$. Thus, the input/output time may be reduced by using a higher-order merge.

Algorithm k-way merge:

1. Read M values at a time into internal memory, sort, and write as runs on disk

2. Merge K runs:
   i. Read first value on each of the k runs into internal array and build min heap
   ii. Remove minimum from heap and write to disk
   iii. Read next value from disk and insert that value on heap

Repeat steps until all first K runs are processed.

Repeat merge on larger & larger runs until have just one large run: sorted list.

In going to a higher order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_k m$ passes. This is so because the number of input buffers needed to carry out a k-way merge increases with k. Though k + 1 buffers are sufficient, the use of 2k + 2 buffers

is more desirable. The optimal value for k clearly depends on disk parameters and the amount of internal memory available for buffers.

15) Present the algorithm for run generation using a looser tree.

Page 34

16) Define latency time and transaction time

Latency Time : Time until the right sector of track is under the read/write head.
Transaction time: Time to transmit the block of data to/from the disk.

17) Define seek time

Time taken to position the read/write heads to correct cylinder. This will depend on the number of cylinders across which the heads have to move.

18) Explain the external sorting technique using a list containing 4500 records and internal memory capable of sorting at most 750 records block length is 250 records. Analyze the complexity of above external sort technique.

Page 4 & 5

19) Briefly explain buffer handling for parallel operation with suitable example.

Page 13 & page 14

20) Explain step by step process of k-way sorting technique with 900 megabytes of data using 100 megabytes of RAM.

For example, for sorting 900 megabytes of data using only 100 megabytes of RAM:

Algorithm:

Step 1)Read 100 MB of the data in main memory and sort by some conventional method, like quicksort.

Step 2) Write the sorted data to disk.

Step 3) Repeat steps 1 and 2 until all of the data is in sorted 100 MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.

Step 4) Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)

Step 5) Perform a 9-way merge and store the result in the output buffer. Whenever the output buffer fills, write it to the final sorted file and empty it.

Whenever any of the 9 input buffers empties, fill it with the next 10 MB of its associated 100 MB sorted chunk until no more data from the chunk is available. This is the key step that makes external merge sort work externally -- because the merge algorithm only makes one pass sequentially through each of the chunks, each chunk does not have to be loaded completely; rather, sequential parts of the chunk can be loaded as needed.

21) Define disk block

The term block refers to the unit of data that is read from or written to a disk at one time. A block generally consists of many records.

22) Using Huffman function, how to find a binary tree of minimal weighted external path length? Explain with suitable example.

Page 24 & 25

23) Write algorithm for k-way merge with floating buffers.

Page 15 to page 21

Merge k sorted arrays

Example:

Input: k = 3, n = 4

```
arr[][] = {
            {1, 3, 5, 7},
            {2, 4, 6, 8},
            {0, 9, 10, 11}
          };
```

Output: 0 1 2 3 4 5 6 7 8 9 10 11

A **simple solution** is to create an output array of size n*k and one by one copy all arrays to it. Finally, sort the output array using any O(n Log n) sorting algorithm. This approach takes O(nk log $_n$k) time.

One efficient solution is to first merge arrays into groups of 2. After first merging, we have k/2 arrays. We again merge arrays in groups, now we have k/4 arrays. We keep doing it unit we have one array left. The time complexity of this solution would O(nk log k). How? Every merging in first iteration would take 2n time (merging two arrays of size n). Since there are total k/2 merging, total time in first iteration would be O(nk). Next iteration would also take O(nk). There will be total O(log k) iterations, hence time complexity is O(nk log k)

Another efficient solution is to use Min Heap. This Min Heap based solution has same time complexity which is O(nk Log k). But for different sized arrays, this solution works much better.

Following is detailed algorithm.

1. Create an output array of size n*k.

2. Create a min heap of size k and insert 1st element in all the arrays into the heap

3. Repeat following steps n*k times.

a) Get minimum element from heap (minimum is always at root) and store it in output array.

b) Replace heap root with next element from the array from which the element is extracted. If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Direct *k*-way merge

In this case, we would simultaneously merge k-runs together.

A straightforward implementation would scan all k arrays to determine the minimum. This straightforward implementation results in a running time of Θ(kn). Note that this is mentioned only as a possibility, for the sake of discussion. Although it would work, it is not efficient.

We can improve upon this by computing the smallest element faster. By using either heaps, tournament trees, or splay trees, the smallest element can be determined in O(log k) time. The resulting running times are therefore in O(n log k).

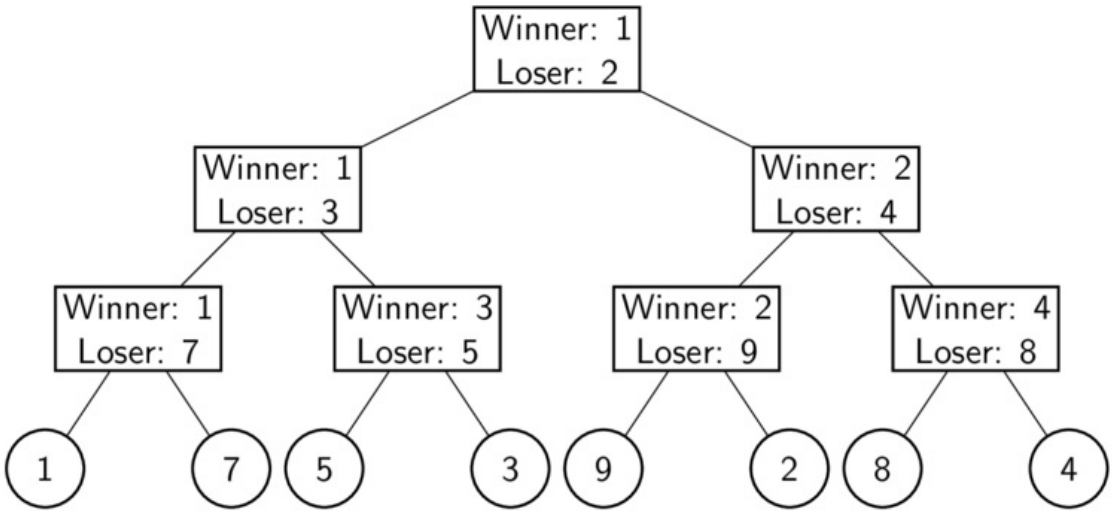
The heap is more commonly used, although a tournament tree is faster in practice. A heap uses approximately 2*log(k) comparisons in each step because it handles the tree from the root down to the bottom and needs to compare both children of each node. Meanwhile, a tournament tree only needs log(k) comparisons because it starts on the bottom of the tree and works up to the root, only making a single comparison in each layer. The tournament tree should therefore be the preferred implementation.

Heap

The heap algorithm allocates a min-heap of pointers into the input arrays. Initially these pointers point to the smallest elements of the input array. The pointers are sorted by the value that they point to. In an O(k) preprocessing step the heap is created using the standard heapify procedure. Afterwards, the algorithm iteratively transfers the element that the root pointer points to, increases this pointer and executes the standard decrease key procedure upon the root element. The running time of the increase key procedure is bounded by O(log k). As there are n elements, the total running time is O(n log k).

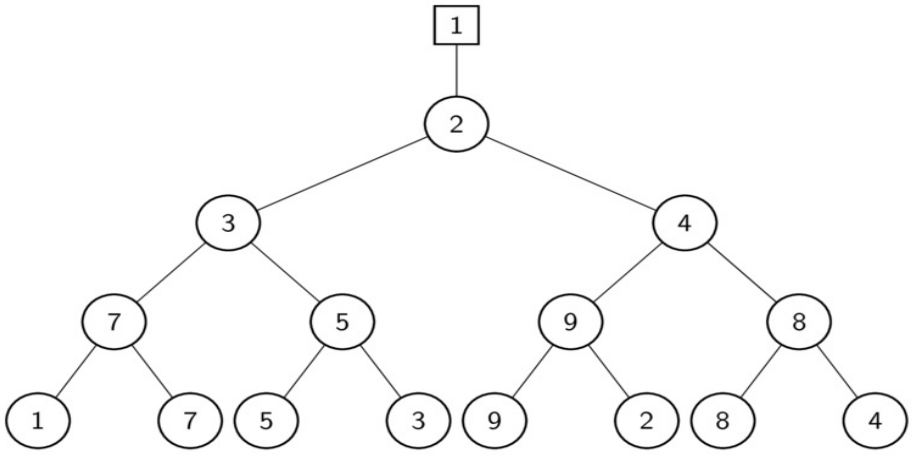
Note that the operation of replacing the key and iteratively doing decrease-key or sift-down are not supported by many Priority Queue libraries such as C++ stl and Java. Doing an extract-min and insert function is less efficient.

Tournament Tree



Tournament Tree

The Tournament Tree is based on an elimination tournament, like it can be found in sports. In each game, two of the input elements compete. The winner is promoted to the next round. Therefore, we get a binary tree of games. The list is sorted in ascending order, so the winner of a game is the smaller one of both elements.



Looser Tree

For k-way merging, it is more efficient to only store the looser of each game (see image). The data structure is therefore called a looser tree. When building the tree or replacing an element with the next one from its list, we still promote the winner of the game to the top. The tree is filled like in a sports match but the nodes only store the looser. Usually, an additional node above the root is added that represents the overall winner. **Every leaf stores a pointer to one of the input arrays.** Every inner node stores a value and an index. The index of an

inner node indicates which input array the value comes from. The value contains a copy of the first element of the corresponding input array.

The algorithm iteratively appends the minimum element to the result and then removes the element from the corresponding input list. It updates the nodes on the path from the updated leaf to the root (*replacement selection*). The removed element is the overall winner. Therefore, it has won each game on the path from the input array to the root. When selecting a new element from the input array, the element needs to compete against the previous loosers on the path to the root. When using a loser tree, the partner for replaying the games is already stored in the nodes. The looser of each replayed game is written to the node and the winner is iteratively promoted to the top. When the root is reached, the new overall winner was found and can be used in the next round of merging.

The images of the tournament tree and the looser tree in this section use the same data and can be compared to understand the way a looser tree works.

Algorithm

A tournament tree can be represented as a balanced binary tree by adding sentinels to the input lists and by adding lists until the number of lists is a power of two. The balanced tree can be stored in a single array. The parent element can be reached by dividing the current index by two.

When one of the leaves is updated, all games from the leaf to the root are replayed. Additionally, the number of lists to merge is assumed to be a power of two.

Optimal Merging of Runs of different sizes

Merge Tree

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. (a) and Fig. (b). In the first merge tree shown in Fig. (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 as a root node.

In the second merge tree shown in Fig. (b), firstly the merging proceeds by combining the runs of size 3 and 6 represented as external nodes to result in a run of size 9 represented as an internal node. The next step is to merge the next external nodes, i.e., runs of size 8 and 14 to result in a run of size 22 represented as an internal node. Finally, the two internal node are merged, i.e., merge the run of size 9 with run of size 22 to obtain a single run of size 31 represented as a root node.

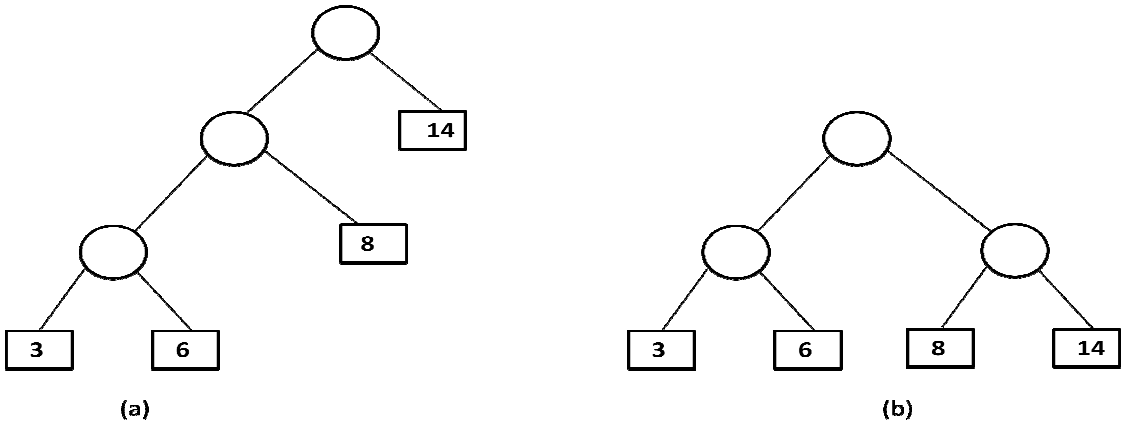


Fig) Two way merge Tree

Now the question is which among the two is optimal, i.e., which type of merge has the minimum merging time. This can be easily obtained by calculating the weighted external path length. 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. The weight of the external node is the *run size*. The depth of the external node is the length of the path from the root node to the external node.

The WEPL can be calculated using the following equation.

WEPL(T) = S(weight of external node i) * (depth of node i from the root)

For the Figure (a), WEPL(T) = 3 * 3 + 6*3 + 8*2 + 14*1 = 57

For the Figure (b), WEPL(T) = 3*2+6*2+8*2+14*2 = 62

The cost of a k–way merge of n runs of length qi where 1≤ i ≤ n is reduced by utilizing a merge tree of degree k that has the minimum WEPL.

Applying the same concept to Fig. (a) and Fig.(b), we observe that Fig.(a) has minimum WEPL value of 57. Therefore, the merging technique followed in Fig.(a) is considered as *optimal merge*.

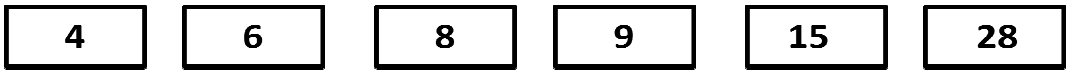
Huffman Tree

Construct a Huffman tree for the following values

q1 = 4, q2 = 6, q3 = 8, q4 = 9, q5 = 15, q6 = 28.

Solution

Step 1: Create six leaf nodes for the six characters as shown below.



Step 2: Select two minimum nodes from the given character list. In this example q1 and q2 values are minimum. Construct a Huffman tree as shown in Fig. a) with q1 as left child and q2 as right child. Calculate the internal node value as sum of the left and right child, i.e., 10. Replace q1 and q2 in the list with this value 10.

The list now contains 10 (internal node value) in Fig. (a), q3 = 8, q4 = 9, q5 = 15, and q6 = 28.

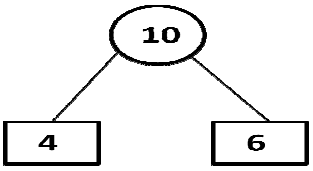


Fig a) Binary tree with nodes q1 and q2

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. (b). Replace q3 and q4 in the list with this value 17. The list now contains 10, 17, q5 = 15, and q6 = 28.

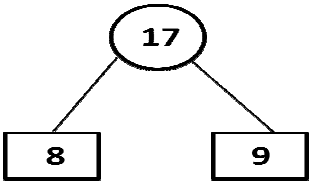


Fig b) Binary tree with nodes q3 and q4

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. (c). Replace 10 and q5 in the list with this value 25. The list now contains 17, 25, and q6 = 28.

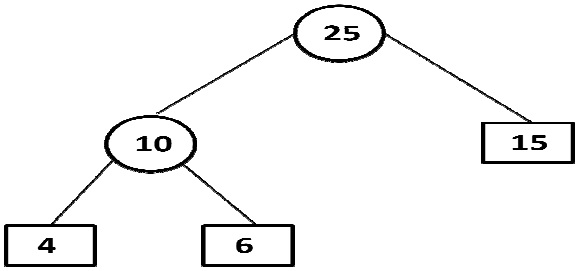


Fig c) Binary tree after q5 inserted

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 (d). Replace 17 and 25 in the list with this value 42. The list now contains, 42 and q6 = 28.

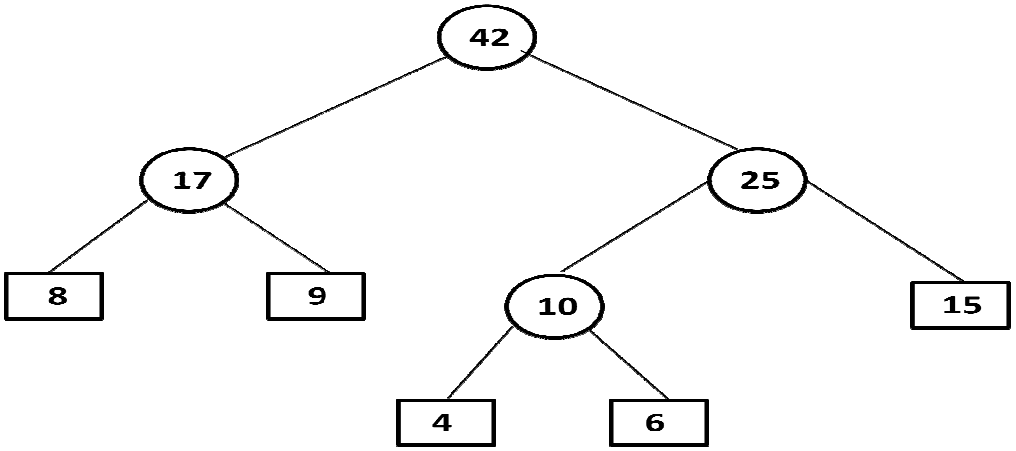


Fig d) Binary tree after 17 and 25 are inserted

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. e. Replace 28 and 42 in the list with this value 70. The list now contains 70.

Consider the next two minimum nodes from the list. The list consists of only one value. The process stops and the final binary tree constructed is the Huffman tree shown in Fig. (e).

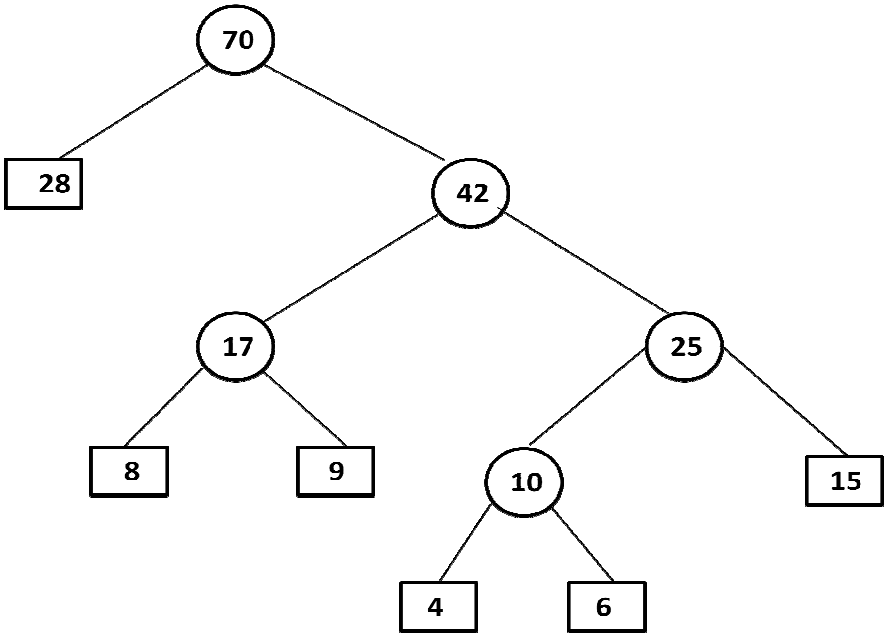


Fig e) Huffman Tree

The Huffman tree in Fig. (e), constructed using Huffman algorithm has a minimum WEPL.

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

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

= 164