UNIT-1

SORTING

Introduction to Sorting

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

Types of Sorting Techniques

Sorting can be classified in two ways:

I. Internal Sorting:

This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated a time in memory is called internal sorting. There is a limitation for internal sorting, they can only process relatively small lists due to memory constraints.

There are 3 types of internal sorts.

1. Selection Sort:

1. Selection sort algorithm
2. Heap sort algorithm

2. Insertion Sort:

    1. Insertion sort algorithm
    2. Shell sort algorithm

3. Exchange Sort:

    1. Bubble sort algorithm
    2. Quick sort algorithm

II. External Sorting:

Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fir into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

- Merge Sort

For a disk, there are three factors contributing to the read/write time:

(1) 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.

(2) Latency time: time until the right sector of the track is under the read/write head.

(3) Transmission time: time to transmit the block of data to/from the disk.

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

First, segments of the input list are sorted using a good internal sort method. These sorted segments, known as runs, are written onto external storage as they are generated.

Second, the runs generated in phase one are merged together until only one run is left.

Example: A list containing 4500 records is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input list is maintained on disk and has a block length of 250 records. We have available another disk that may be used as a scratch pad. The input disk is not to be written on. One way to accomplish the sort using the general function outlined above is to

(1) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs R1 to R6

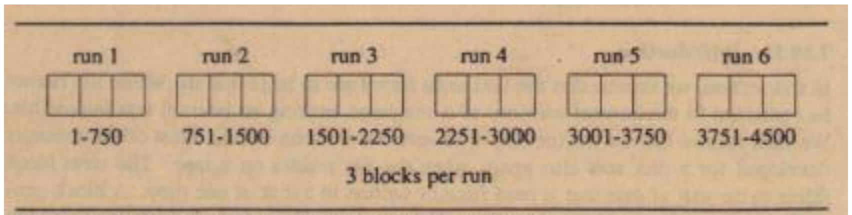
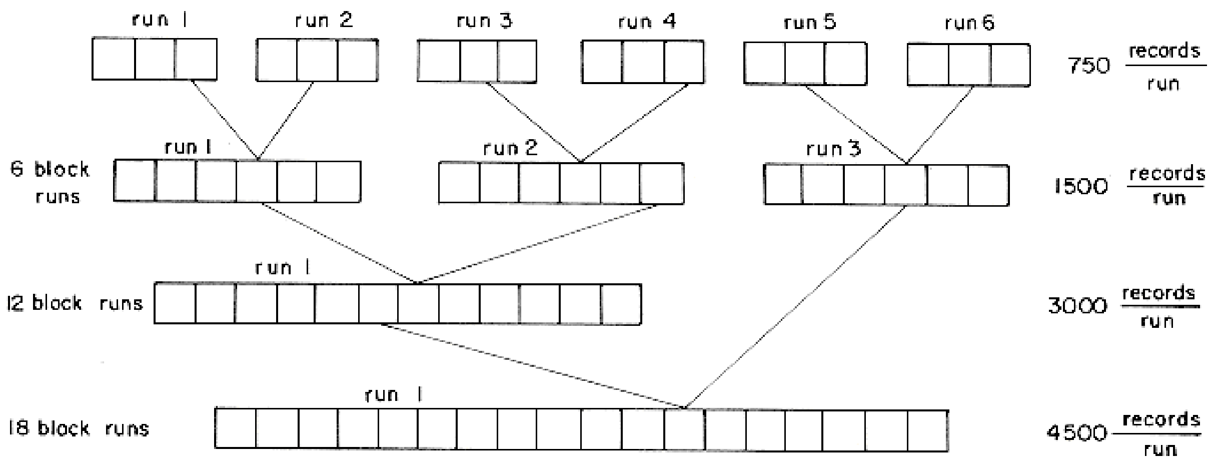
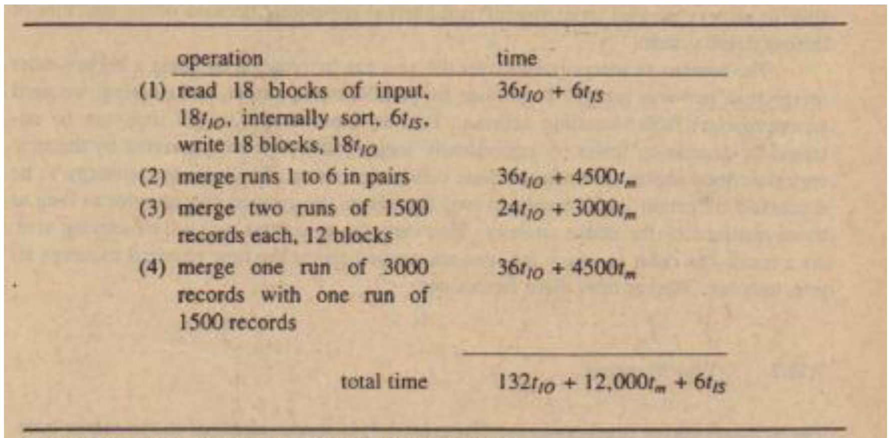


Figure 7.19: Blocked runs obtained after internal sorting

(2) 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 runs R. and R 2: This merge is carried out by first reading one block of each of these runs into input buffers.



We shall assume that each time a block is read from or written onto the disk the maximum seek and latency times are experienced. Although this is not true in general it will simplify the analysis. The computing times for the various operations in our 4500- record example are given in Figure:



	operation	time
(1)	read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
(2)	merge runs 1 to 6 in pairs	$36t_{IO} + 4500t_m$
(3)	merge two runs of 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
(4)	merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
	total time	$132t_{IO} + 12{,}000t_m + 6t_{IS}$

Figure 7.21: Computing times for disk sort example

k- Way Merging

The two-way merge function merge (Program 7.7) is almost identical to the merge function just described (Figure 7.20). In general. if we start with m runs, the! merge tree corresponding to Figure 7.20 will have fiog2m 1+1 levels. for a total of fiOg2m1 passes over the data list. The number of passes over the data can be reduced by using a higher order merge

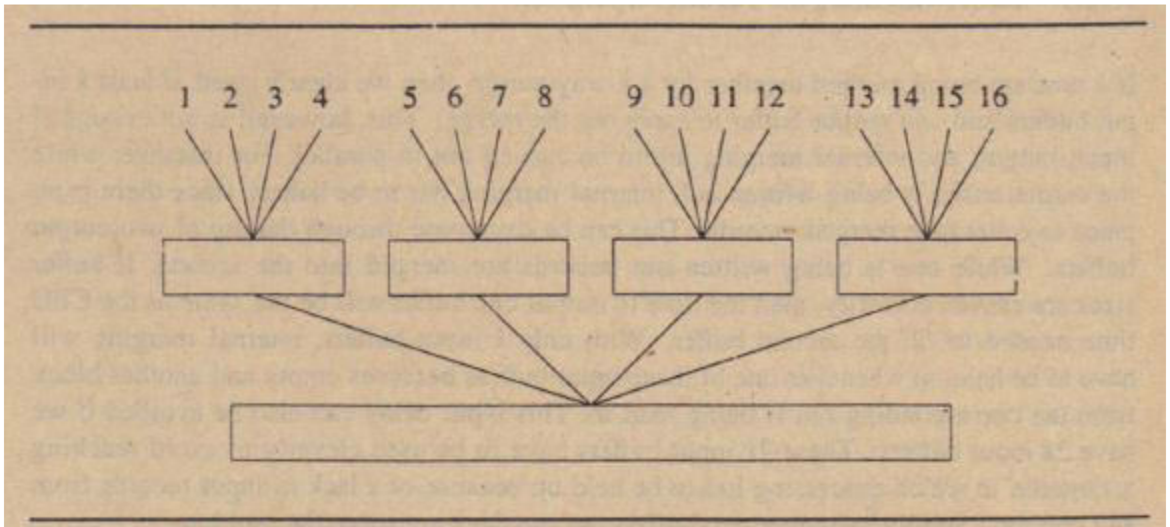


Figure 7.22: A four-way merge on 16 runs

Buffer Handling for Parallel Operation

If k runs are being merged together by a k-way merge, then- we clearly need at least k input buffers and one output buffer to carry out the merge. This, however. is not enough if 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 the 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. If buffer sizes are chosen correctly, then the time to output one buffer will be the same as the CPU time needed to fill the second buffer.

Example: Assume that a two-way merge is carried out using four input buffers, and two output buffers, ou [0] and ou [I]. Each buffer is capable of holding two records. The first few records of run 0 have key value I, 3, 5, 7. 8. 9. The first few records of run I have key value 2,4,6, 15.20.25.

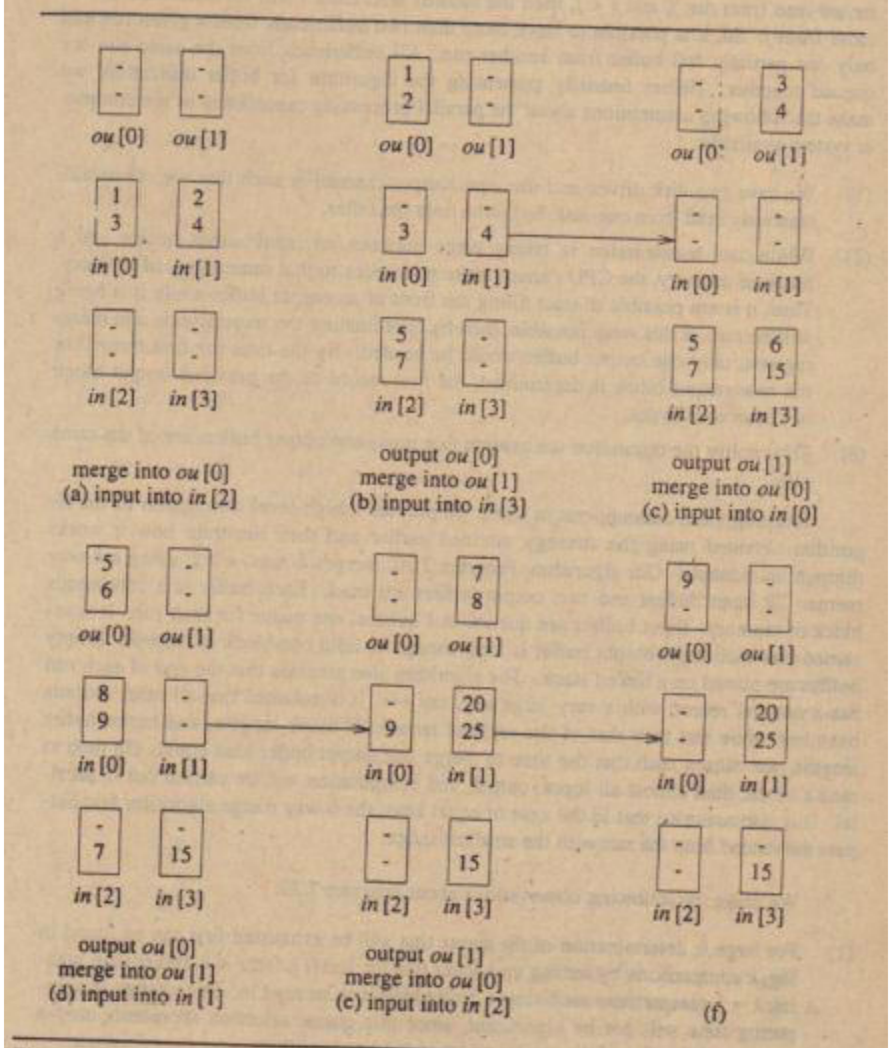


Figure 7.23: Example showing that two fixed buffers per run are not enough for continued parallel operation

Buffering Algorithm:

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.

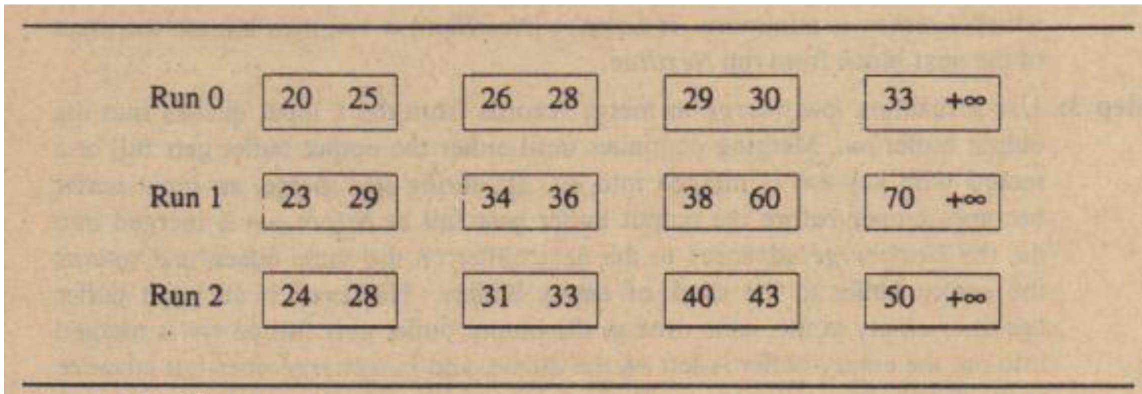
Step 2: Let Last Key [i] be the last key input from run i. Let NextRun be the run for which Last Key is minimum. If LastKey [NextRun] = +∞, then initiate the input of the next block from run NextRun.
Step 3: Use a function k-way merge to merge records from the k input queues into the output buffer ou. Merging continues until either the output buffer gets full or a record with key +∞ is merged into ou. If, during this merge, an input buffer becomes empty before the output buffer gets full or before +∞ is merged into ou, the k-way merge advances to the next buffer on the same queue and returns the empty buffer to the stack of empty buffers. However, if an input buffer becomes empty at the same time as the output buffer gets full or +∞ is merged into ou, the empty buffer is left on the queue, and K-way merge does not advance to the next buffer on the queue. Rather, the 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 the appropriate run. Determine the next run to read from by determining NextRun such that LastKey [NextRun] is minimum.

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

Step 7: Initiate the writing of output buffer ou.

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

Ex: 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. Figure 7.25 shows the status of the input buffer queues, the run from which the next' block is being read, and the output buffer being output at the beginning of each iteration of the loop of Steps 3 through 8 of the buffering algorithm.



Run 0	20 25	26 28	29 30	33 +∞
Run 1	23 29	34 36	38 60	70 +∞
Run 2	24 28	31 33	40 43	50 +∞

Figure 7.24: Three runs

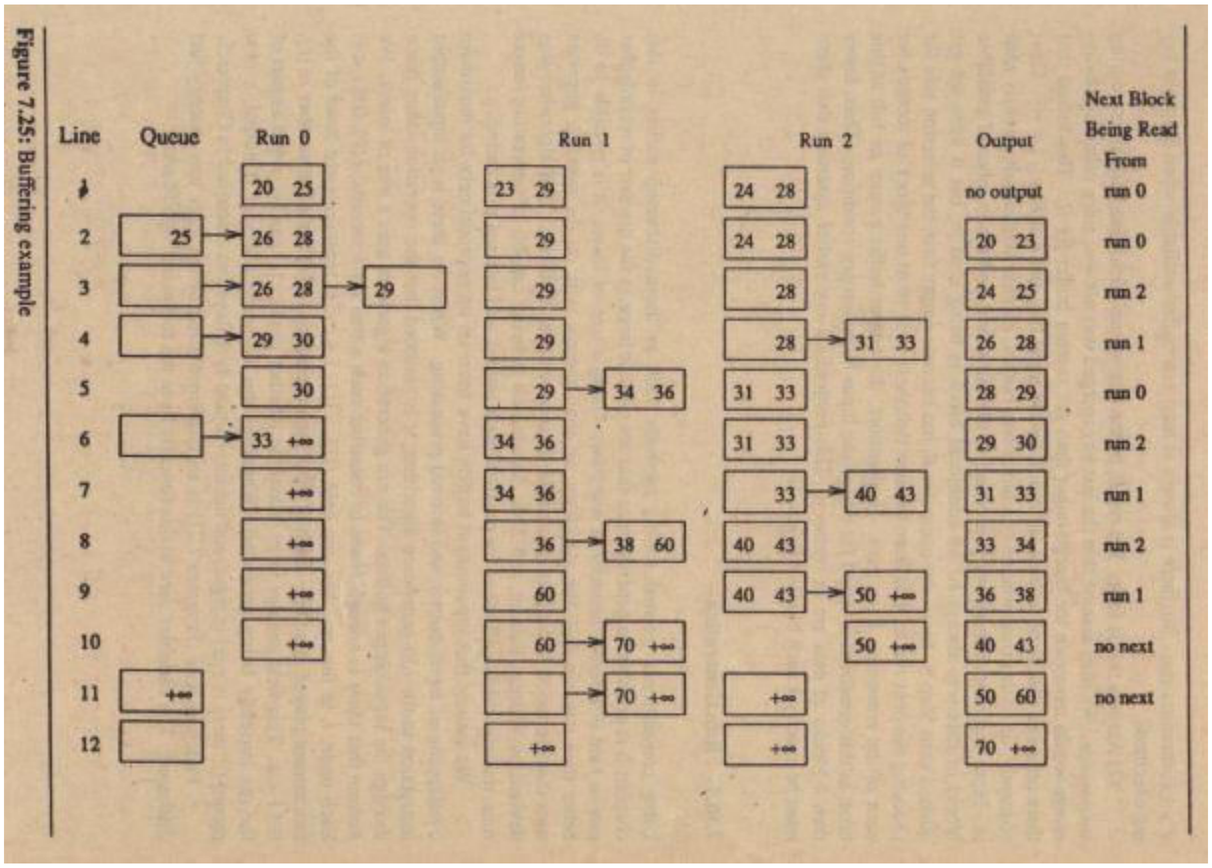


Figure 7.25: Buffering example

Line	Queue	Run 0	Run 1	Run 2	Output	Next Block Being Read 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	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 $+\infty$	34 36	31 33	29 30	run 2
7		$+\infty$	34 36	33 → 40 43	31 33	run 1
8		$+\infty$	36 → 38 60	40 43	33 34	run 2
9		$+\infty$	60	40 43 → 50 $+\infty$	36 38	run 1
10		$+\infty$	60 → 70 $+\infty$	50 $+\infty$	40 43	no next
11	$+\infty$		→ 70 $+\infty$	$+\infty$	50 60	no next
12			$+\infty$	$+\infty$	70 $+\infty$	

Run Generation:

It is possible to generate runs those are only as large as the no of records can be held in internal memory at one time.

Using a tree of losers, it is possible to do better than this.

Winner Trees

Complete binary tree with n external nodes and n - 1 internal nodes.

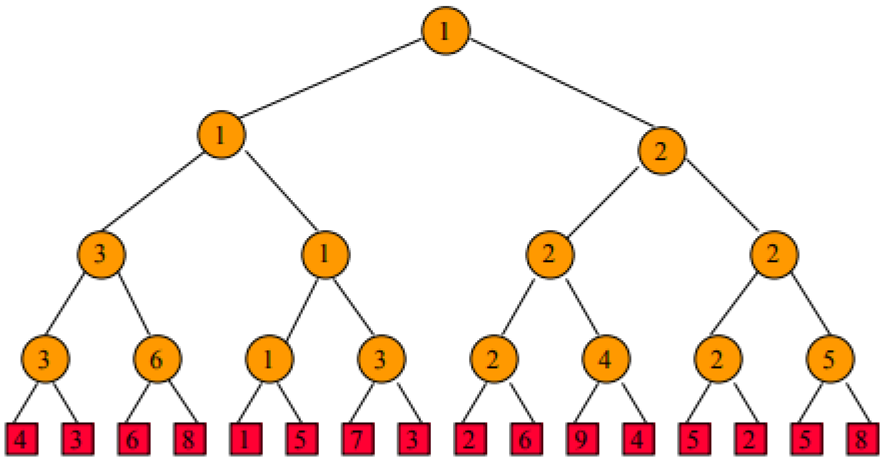
External nodes represent tournament players.

Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.

Root is overall winner.

Ex:

Winner Tree For 16 Players



Smaller element wins => min winner tree.

<u>Time To Sort</u>

• Initialize winner tree.

O(n) time

• Remove winner and replay.

O(log n) time

• Remove winner and replay n times.

O(n log n) time

• Total sort time is O(n log n).

• Actually O (n log n).

Winner Tree Operations

• Initialize

O(n) time

• Get winner

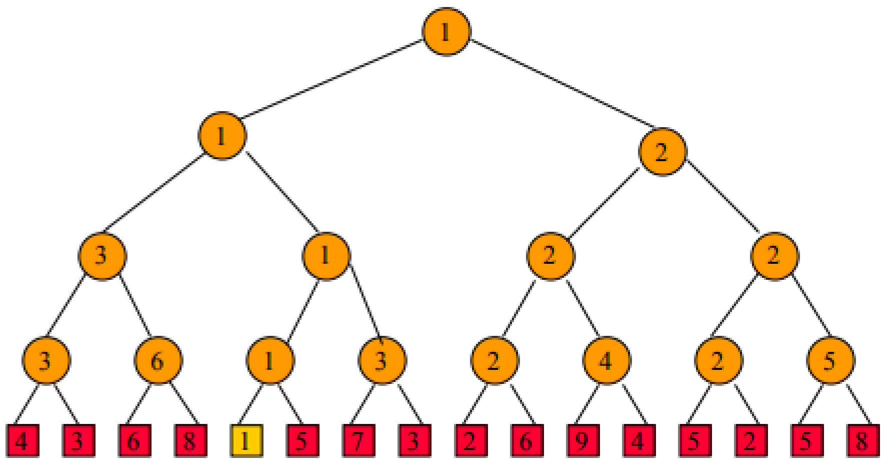
O(1) time

• Remove/replace winner and replay

O(log n) time

more precisely O (log n)

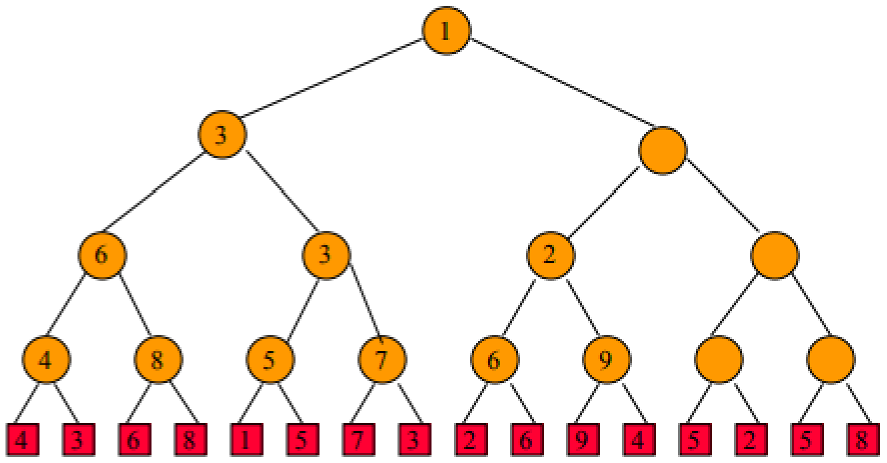
Replace Winner And Replay



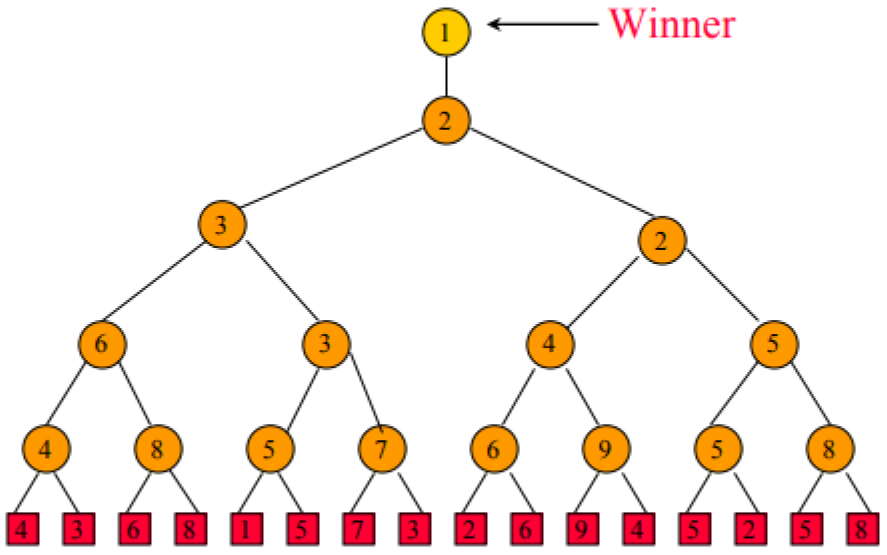
Replace winner with 6.

Loser Tree: Each match node stores the match loser rather than the match winner.

Min Loser Tree For 16 Players



Replace and Replay:



Analysis of runs: When the input list is already sorted. only one run is generated. On the average. the run size is almost 2k. The time required to generate all the runs for an 11 run list is 0(11 log k). as it takes O(log k) time to adjust the loser tree each time a record is output.

Optimal Merging of Runs
The runs generated by function runs may not be of the same size. When tuns are of different size, the run merging strategy employed so far (i.e., make complete passes over the collection of runs) does not yield minimum run times.

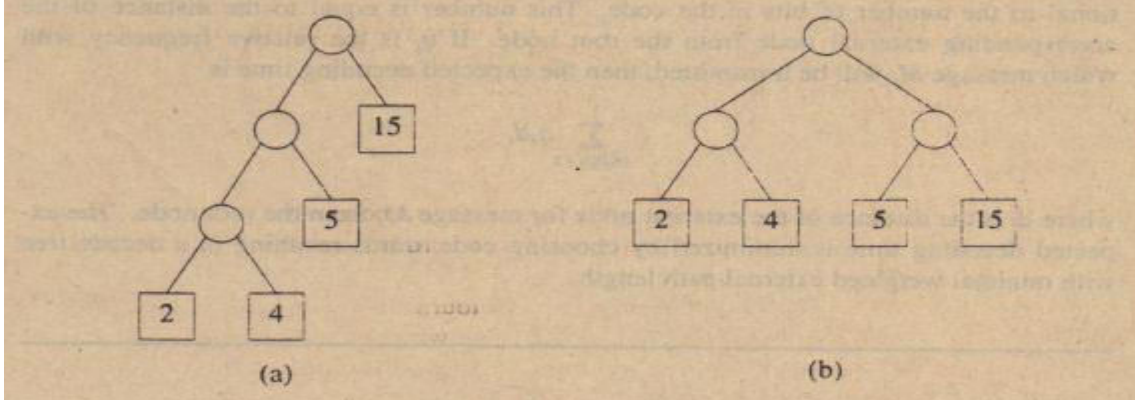


Figure 7.26: Possible two-way merges

The circular nodes represent a two-way merge using as input the data of the children nodes. The square nodes represent the initial runs. We shall refer to the circular nodes as internal nodes and the square ones as external nodes. Each figure is a merge tree.

In the first merge tree. we begin by merging the runs of size 2 and 4 to get one of size 6; next this is merged with the run of size 5 to get a run of size 11; finally this run of size II is merged with the run of size 15 to get the desired sorted run of size 26. When merging is done using the first merge tree. some records are involved in only one merge. and others are involved in up to three merges. In the second merge tree each record is involved in exactly two merges.

Since the time for a merge is linear in the number of records being merged, 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. For the two trees 01 Figure 7.26, the respective weighted external path lengths are

2 . 3 + 4 . 3 + 5 . 2 + 15 . 1 = 43

and

2 . 2 + 4 . 2 + 2 + 5 . 2 + 15 . 2 = 52

another application for binary trees with minimum weighted external path length. Suppose we wish to obtain an optimal set of codes for messages M1………..Mn + 1 Each code is a binary string that will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages.

The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node.

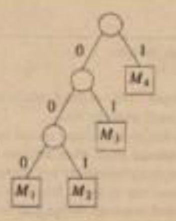


Figure 7.27: A decode tree

If we interpret a zero as a left branch and a one as a right branch. then the decode tree from above figure corresponds to codes 000. 001; 01, and 1 for messages M1. M2. M3. and M4. respectively. These codes are called Huffman codes.

A very nice solution to the problem of finding a binary tree with minimum weighted external path length has been given by D. Huffman.

Ex: Suppose we have the weights q1=2, q2=3, q3=5, q4=7, q5=9a and q6=13. Construct a Huffman tree and find external path length.

NOTE: The number in a circular node represents the sum of weights of external nodes in subtree.

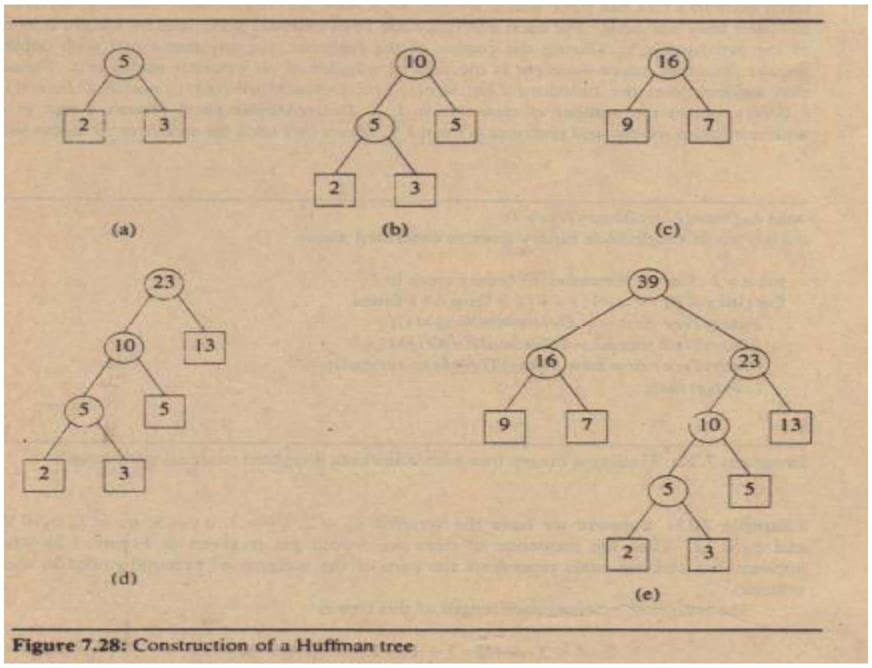


Figure 7.28: Construction of a Huffman tree

The weighted external path length for above tree is:

2.4+3.4+5.3+13.2+7.2+9.2=93

<u>Analysis of Huffman tree:</u>

Heap initialization takes O(n) time.

Push and pop requires only O(log n) time.

Therefore the asymptotic time for the algorithm is: O(n log n).