UNIT I
EXTERNAL SORTING

External Sorting:

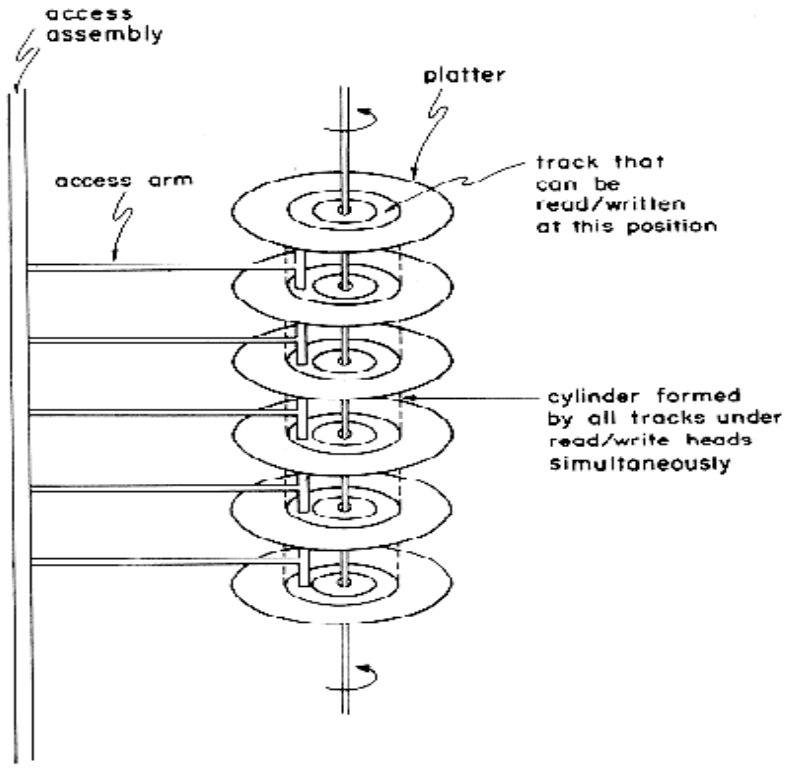
External Sorting is sorting the lists that are so large that the whole list cannot be contained in the internal memory of a computer.

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 form 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.

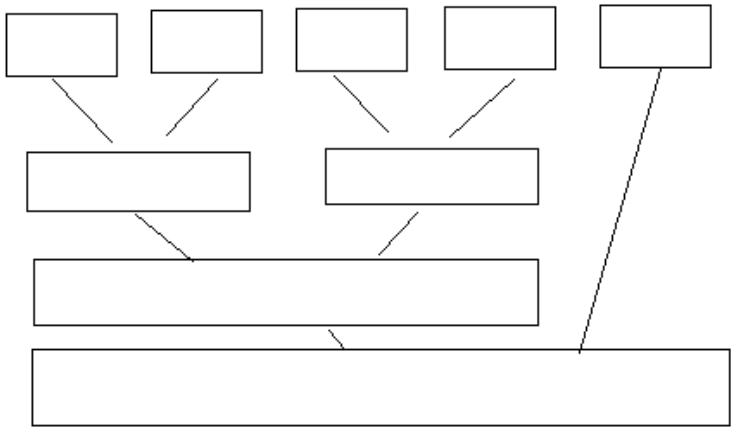
(iii) *Transmission time:* time to transmit the block of data to/from the disk.



Example of External Sorting: 2-way Merge Sort, k-way Merge Sort, Polyphase Merge sort etc.

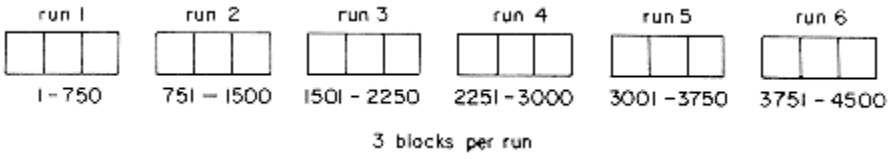
External Sorting with Disks:
The most popular method for sorting on external storage devices is merge sort. This method consists of essentially two distinct phases.
First Phase(Run Generation Phase): Segments of the input file are sorted using a good internal sort method. These sorted segments, known as *runs*, are written out onto external storage as they are generated.
Second Phase(Merge Phase): The runs generated in phase one are merged together following the merge tree pattern , until only one run is left.

Merge tree pattern



Example:
A file containing 4500 records, A_1, . . . ,A_{4500}, is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input file is maintained on disk and has a block length of 250 records.

(i) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs R_1-R_6. A method such as heapsort or quicksort could be used. These six runs are written out onto the scratch 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 runs R_1 and R_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 runs R_1 and R_2 have been merged, R_3 and R_4 and finally R_5 and R_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.

Analysis of External sort to sort 4500 records:

Notations:

t_s = maximum seek time

t_l = maximum latency time

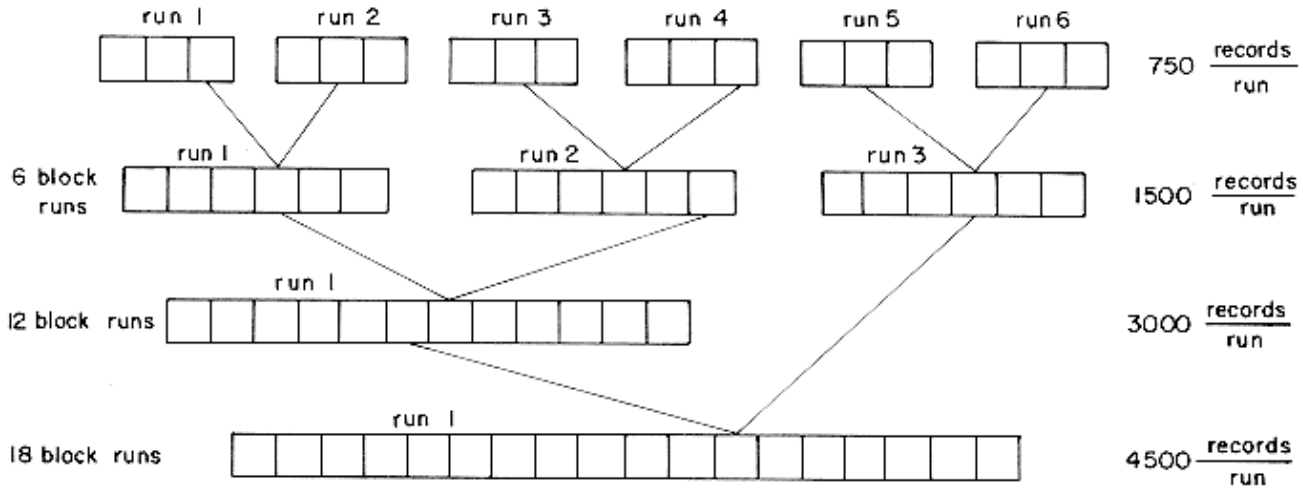
t_{rw} = time to read or write one block of 250 records

$t_{IO} = t_s + t_l + t_{rw}$

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:



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 start.

Let M=maximum number of records that can be sorted & sorted in internal memory at one time.

Algorithm:

Repeat

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

Until data is processed into runs.

Repeat

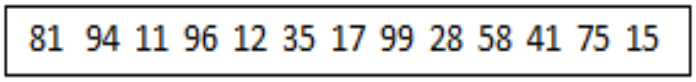
1. Merge two runs into one sorted run twice as long.
2. Write this single run back onto disk

Until all runs processed into runs twice as long

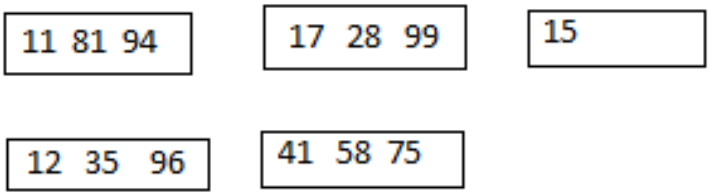
Merge runs again as often as needed until only one large run: The sorted list.

Example:

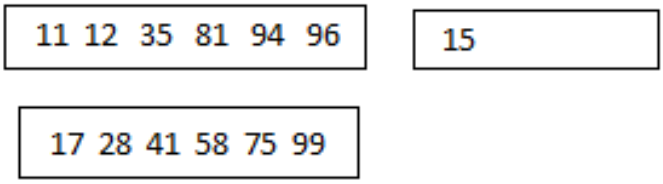
Unsorted Data on Disk



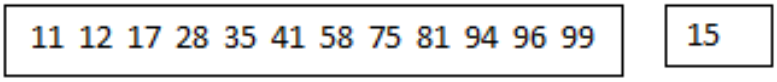
Assume M = 3 (M would actually be much larger, of course.) First step is to read 3 data items at a time into main memory, sort them and write them back to disk as runs of length 3.



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



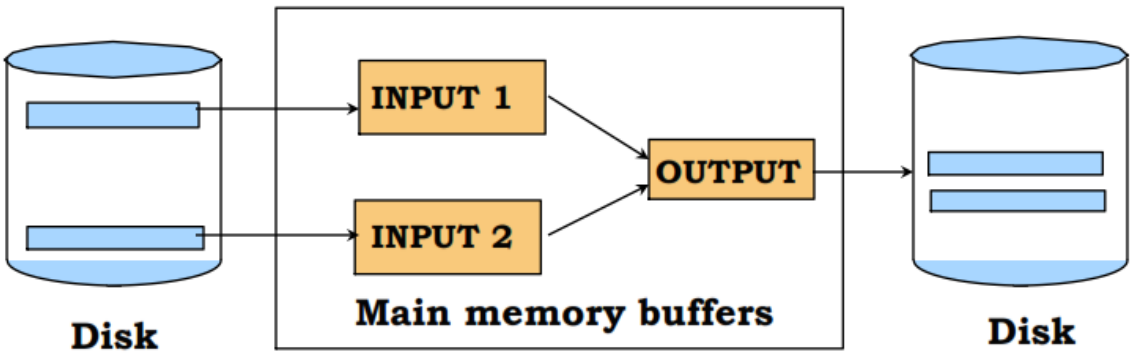
Next step is to merge the runs of length 6 into runs of length 12.



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.

11	12	15	17	28	35	41	58	75	81	84	96	99

2-Way Sort: Requires 3 Buffers



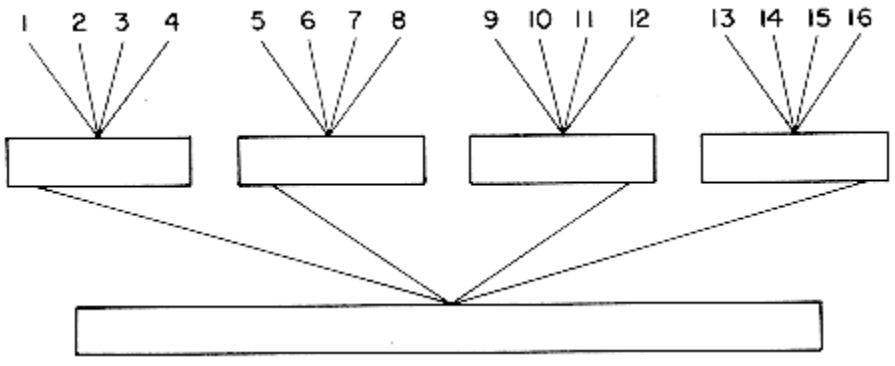
Pass 1: Read a page, sort it, write it.
- only one buffer page is used

Pass 2, 3, …, N etc.:
- Read two pages, merge them, and write merged page
- Requires three buffer pages.

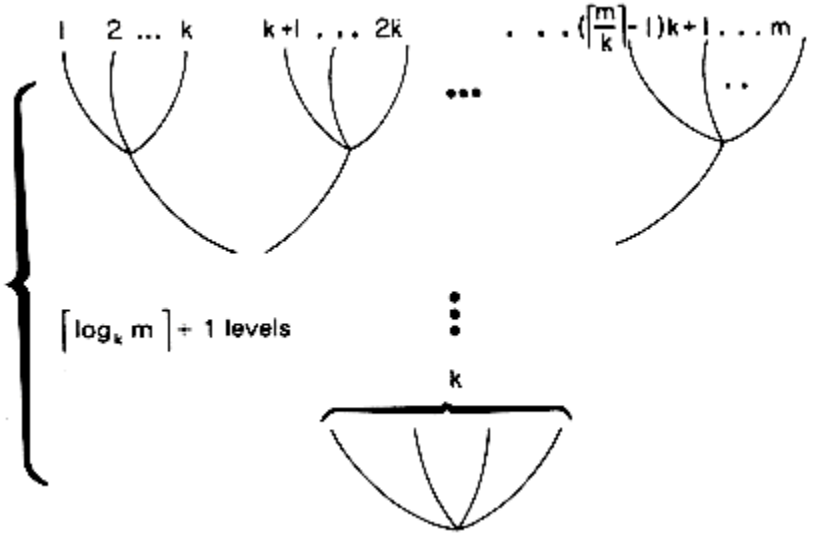
Each pass we read + write each page in file.
N pages in the file => the number of passes = $\lceil \log_2 N \rceil + 1$
So total cost is: $2N(\lceil \log_2 N \rceil + 1)$

Multi-way Merge Sort/ k-way Merge Sort
Instead of a 2-way merge idea is 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 $k \geq 2$. In this case we would simultaneously merge *k* runs together.

A 4-way Merge on 16 Runs



A k-Way Merge



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 $\lceil \log_k m \rceil$ passes over data where as 2-way merge on m runs requires $\lceil \log_2 m \rceil$. Thus, the input/output time may be reduced by using a higher-order merge.

Algorithm:

1. As before, read M values at a time into internal memory, sort, and write as runs on disk
2. Merge K runs:
    1. Read first value on each of the k runs into internal array and build min heap
    2. Remove minimum from heap and write to disk
    3. 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.

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 example 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 easily overcome through the use of two output buffers. While one is being written out, records are merged into the second. In the same way, with only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty and another block from the corresponding run is being read in. This input delay can also be avoided if we have *2k* input buffers.

Ex: Assume that a two way merge is being carried out using four input buffers in[0],in[1],in[2],in[3] buffers and two output buffers ou[0] and ou[1]. Let each buffer is capable of holding two records.

Let run 0 consists of 1, 3, 5, 7, 8, 9 keys

Let run 1 consists of 2, 4, 6, 15, 20, 25 keys

Let in[0] and in[2] are assigned to run[0] and in[1] and in[3] are assigned to run[1].

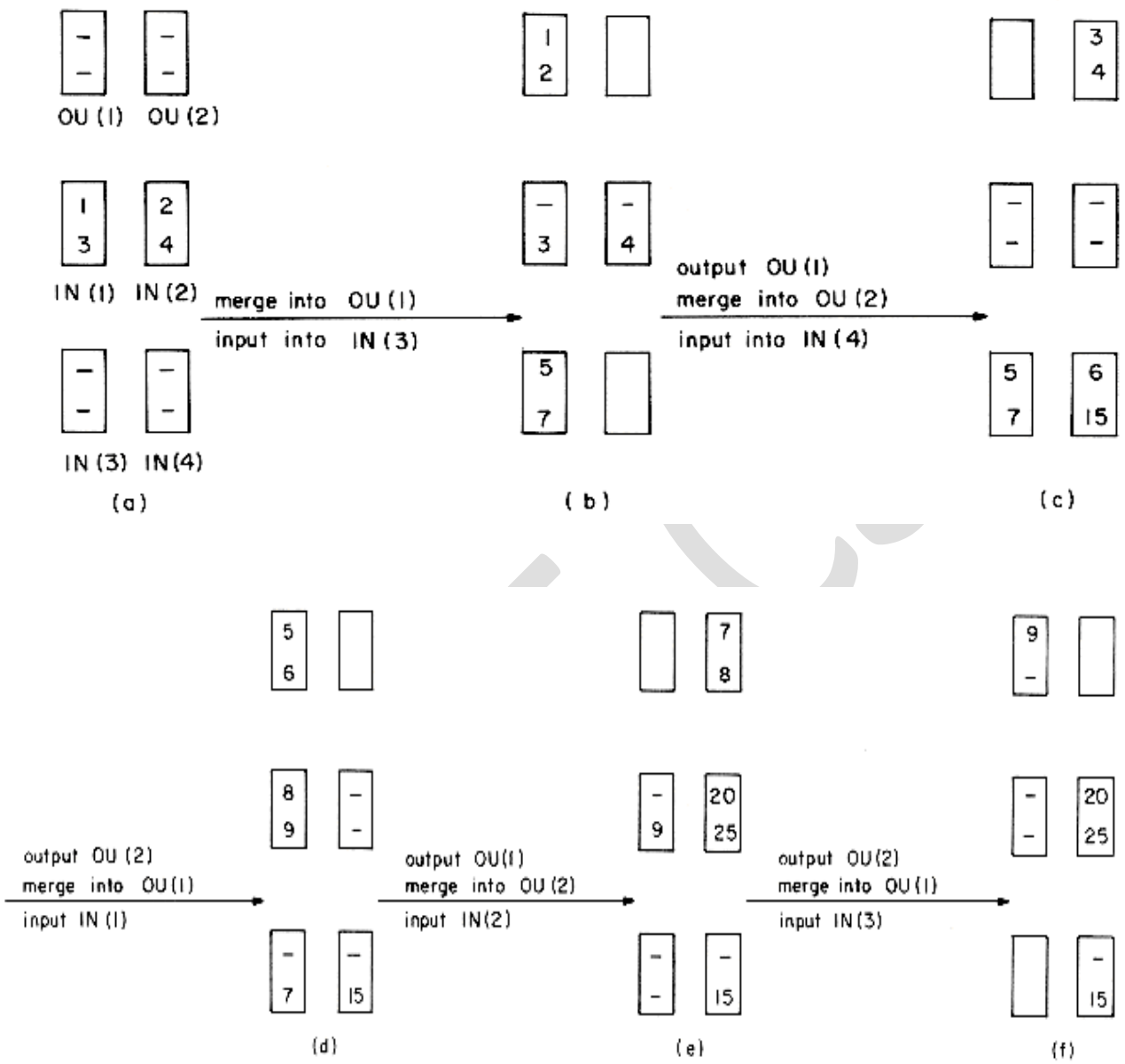
We make the following assumptions about the parallel processing capabilities of the computer system available:

(i) We have two disk drives and the input/output channel is such that it is possible simultaneously to read from one disk and write onto the other.

(ii) While data transmission is taking place between an input/output device and a block of memory, the CPU cannot make references to that same block of memory. Thus, it is not possible to start filling the front of an output buffer while it is being written out. If this were possible, then by coordinating the transmission and merging rate only one output buffer would be needed. By the time the first record for the new output block was determined, the first record of the previous output block would have been written out.

(iii) To simplify the discussion we assume that input and output buffers are to be the same size.

Configuration of buffers is shown in figure



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.

Steps in buffering algorithm(k-way merge 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]!=∞, the initiate the input of the next block from run nextRun.

Step 3: Use a function k-way merge ro 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-way merge 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 out. Set OU to 1-OU.

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

Example to illustrate 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.

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 +∞

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	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. Using a tree of losers, 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.

Run generation using a loser tree

```
template <class T>
 1 void Runs(T *r)
 2 {
 3   r = new T[k];
 4   int *rn = new int[k], *l = new int[k];
 5   for (int i = 0; i < k; i++) {// input records
 6     InputRecord (r [i ]); rn [i ] = 1;
 7   }
 8   InitializeLoserTree ();
 9   T q = l [0]; // tournament winner
10   int rq = 1, rc = 1, rmax = 1; T lastRec = MAXREC;
11   while(1) {// output runs
12     if (rq != rc) {// end of run
13       output end of run marker;
14       if (rq > rmax) return;
15       else rc = rq;
16     }
17     WriteRecord (r [q ]); lastRec = r [q ]; // output record r [q ]
18     // input new record into tree
19     if (end of input) rn [q ] = rmax + 1;
20     else {
21       ReadRecord (r [q ]);
22       if (r [q ] < lastRec)  // new record belongs to next run
23         rn [q ] = rmax = rq + 1;
24       else rn [q ] = rc;
25     }
26     rq = rn [q ];
27     // adjust losers
28     for (t = (k +q) /2; t; t /= 2;) // t is initialized to be parent of q
29       if ((rn [l [t ]] < rq) || ((rn [l [t ]] == rq) && (r [l [t ]] < r [q ])))
30       {// t is the winner
31         swap (q, l [t ]);
32         rq = rn [q ];
33       }
34   }
35   delete [ ] r ; delete [ ] rn ; delete [ ] l;
36 }
```

Program 7.22: Run generation using a loser tree

The variable used in this function have the following significance.
r[i], 0<=i<k -> the k records in the tournament tree.
r[i],1<=i<k -> loser of the tournament played at node i.
l[0] -> winner of tournament
m[i], 0<=i<k -> the run number to which r[i] belongs
rc -> run number for r[q]
rmax -> number of runs that will be generated
lastRec -> last record output
MAXREC -> a record with maximum key possible.

If key[q] < lastkey then r[q] cannot be output as part of the current run rc, as a record with larger key value has already been output in this run. When a tree is being readjusted, a record with lower run number wins over one with a higher run number. When run numbers are equal, the record with lower key value wins. This ensures that records come out of the tree in non-decreasing order of their run number. When we run out of input, a record with run number rmax+1 is introduced. When this record is ready for output, function terminates.

Analysis: When the input list is already sorted, only one run is generated. On the average, the runsize is almost 2k. The time required to generate all the runs for an n runlist is 0(nlogk) as it takes 0(logk) time to adjust the loser tree each time a record is output.

Optimal Merging of Runs

When runs are of different size, the run merging strategy employed so far does not yield minimum runtimes. For example, suppose we have four runs of length 2, 4, 5 and 15 respectively.
Some possible two-way merge

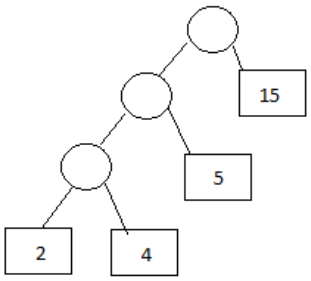


fig a

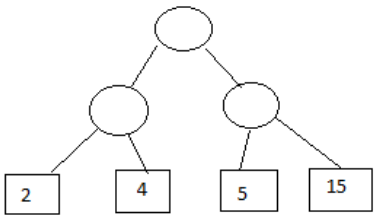


fig b

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.

For fig a. the weighted external path length is 2.3+4.3+5.2+15.1 = 43

For fig a. the weighted external path length is 2.2 + 4.2 + 2 + 5.2 + 15.2 = 52

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

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.

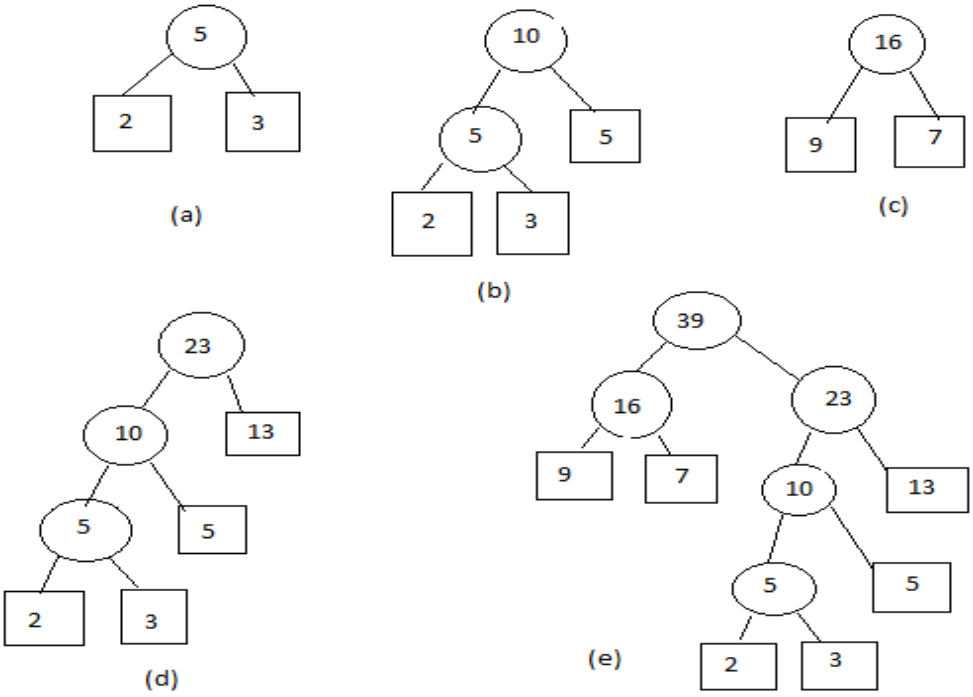
This solution begins with min heap of n single node trees. The single node in each of these trees represents one of the provided qi's and its weight is qi. Huffman's algorithm then repeatedly extracts two minimum-weight trees a and b from min heap, a combines them into a single binary tree c by creating 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 weights q1=2, q2=3, q3=5, q4=7, q5=9 and q6=13. Then the sequence of trees we would get is



The circular node represents the sum of the weights of external nodes in that subtree.
The weighted external path length of this tree is
2.4+3.4+5.3+13.2+7.2+9.2=93

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

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

*****