11.5 EXTERNAL SORTING

External sorting is required if the data to be sorted are large enough, which cannot be loaded at once in memory, and common sorting algorithms are not applicable. Larger files may be too large to fit in memory simultaneously and require external sorting. External sorting uses secondary devices (magnetic tapes or magnetic disks). The criteria for evaluating external sorting algorithms are different from internal sorts. Internal sort comparison criteria are: number of comparisons required, number of swaps made and memory requirement, whereas external sort comparison criteria are dominated by I/O time (transfers between secondary storage and main memory). External sorting which uses magnetic tapes is known as tape sorting.

Runlists: External sorting is the process of sorting small groups of records from a file in memory. These groups are called runlists. The size of these runlists depends upon the internal memory used for internal sorting. These runlists are stored in a target file from which they are retrieved and merged together to form large runlists. This process is continued until a single runlist is generated, which is the required sorted file.

Assume that there is a buffer enough to hold p records and a file with q records; always p is less than q. Sorting a file involves generation of runlists which are used to read q records into an internal buffer, sort them using internal sorting and write them onto a target file of size q. If the value of q is smaller than 15 then selection sort can be used and for larger values of q heap sort can be used. This process of generating runlists is continued until all q records are performed.

11.5.1 Polyphase Merge

Magnetic tape is a sequential storage medium used for data collection and back up. The main disadvantage of tape sorting is that huge amounts of tape rewinding takes place. Polyphase sort is one of the best method for performing external sorting on tapes.

A process of distributing ordered runlists of predetermined size on to the tapes and continuously merging these runlists in multiple phases is the polyphase merge. Each phase has a fixed number of merges before a new tape is selected. The initial distribution of runlists on the working of tapes affects the performance of sorting. However, it is found that the Fibonacci distribution of initial runlists produces better performance. Polyphase merge is also known Polyphase sort.

Consider sorting of 9 records using 4 tapes. Originally all the records are on tape 4 as shown in Figure 11.22(a). Record is represented by its key. The n^{th} order Fibonacci series is used to determine the number of runs on each tape where $n=T-1$, where T is the number of tapes. This series is defined as follows:

$$F^n = F^n_{s-1} + F^n_{s-2} + \ldots + F^n_{s-n}$$
$$F^n_s = 0 \text{ for } 0 \le s \le n-2$$
$$F^n_{n-1} = 1$$

for the s-level perfect distribution where s is the size of the runlist. In general, the k^{th} tape should be allotted $F^n_{s-1}+F^n_{s-2}+\ldots+F^n_{s-k}$ initial runs in the s-level perfect distribution.

Therefore, in the example with 9 records, when n=3, tape 1 receives 2 runs, tape 2 receives 3 runs and tape 3 receives 4 runs. Based on the recurrence formulae, the initial runs are distributed on the tapes (Figure 11.22(b)).

Algorithm 11.1

```
Step 1:    A loop designed to distribute the runs on the tapes. The distribution is a Fibonacci distribution.
Step 2:    Rewound tapes to allow them to be read.
Step 3:    Merge and sort: During merging runs from the source tapes are compared and sorted and written on
           object (output tape). When the source tape becomes empty it takes the role of an object tape. The
           earlier object tape becomes the source tape.
```

To keep the example simple all initial runs are assumed to be of length 1 (one record in each run). In practice these runs should be made as large as possible. Nine initial runs are available on tape 4. The initial run distribution on tapes 1, 2 and 3 is a Fibonacci distribution (Figure 11.22(b)).

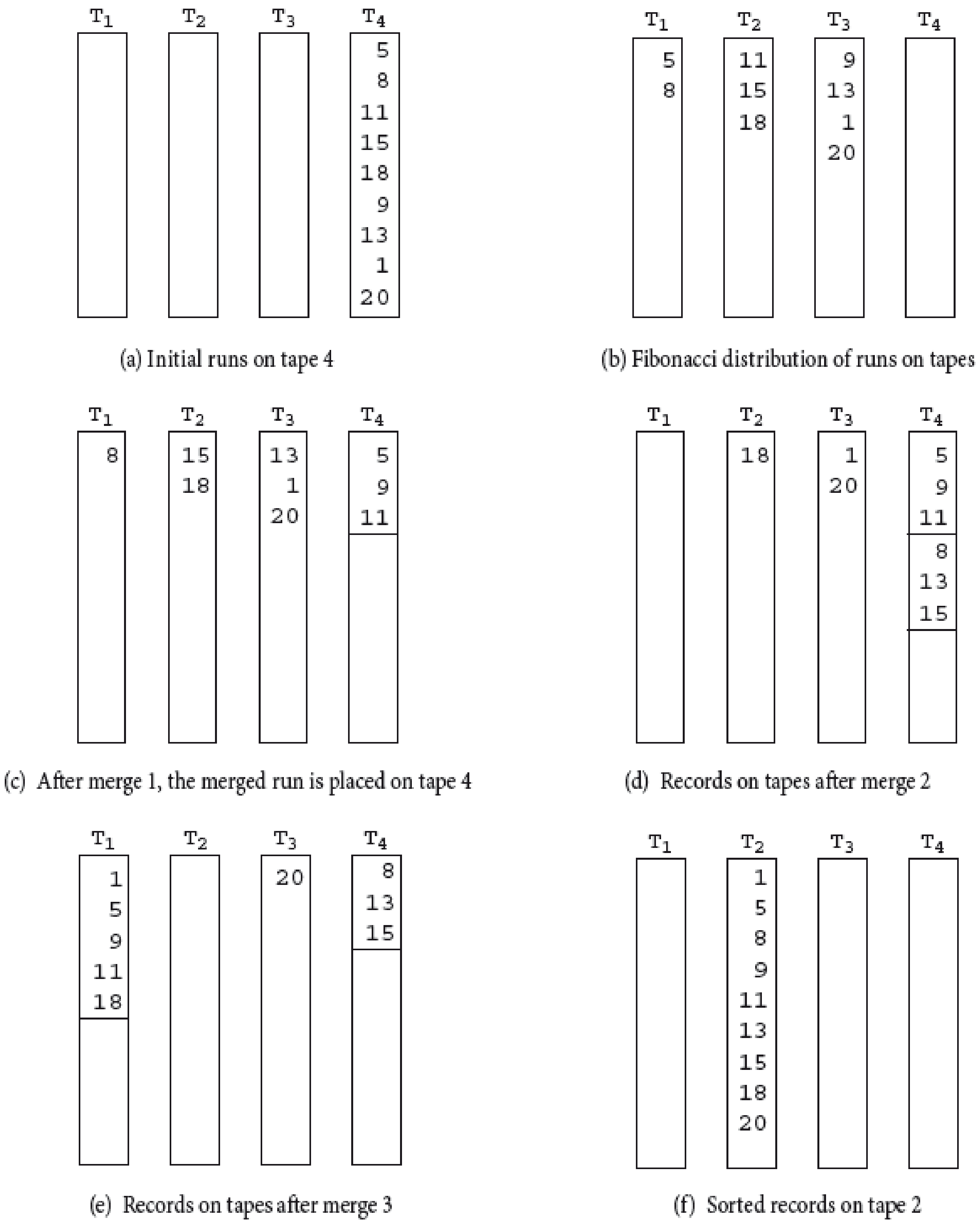

T1 T2 T3 T4
5 8 11 15 18 9 13 1 20
(a) Initial runs on tape 4
T1 T2 T3 T4
5 8
11 15 18
9 13 1 20
(b) Fibonacci distribution of runs on tapes
T1 T2 T3 T4
8
15 18
13 1 20
5 9 11
(c) After merge 1, the merged run is placed on tape 4
T1 T2 T3 T4
18
1 20
5 9 11 8 13 15
(d) Records on tapes after merge 2
T1 T2 T3 T4
1 5 9 11 18
20
8 13 15
(e) Records on tapes after merge 3
T1 T2 T3 T4
1 5 8 9 11 13 15 18 20
(f) Sorted records on tape 2

Figure 11.22 Initial run distribution

Tapes 1, 2 and 3 are source tapes, whereas tape 4 is an object tape (output tape).

On the first merge pass the initial runs on each of the source tapes 1, 2 and 3 are merged (i.e. records 5, 11, 9), and the run <5, 9, 11> is placed on the initially empty tape 4 (Figure 11.22(c)).

The second merge places the run <8, 13, 15> on the tape 4 (Figure 11.21(d)). This now leaves tape 1 with no more runs left to be read. It is therefore rewound in preparation for its new role as the object tape (output tape) in the next merge phases. Tape 4 is also rewound so that it can play as the source tape (input tape).

The next phase begins by merging runs from tapes 2, 3 and 4 which results in run <1, 5, 9, 11, 18> and placed on tape 1.

At this stage, tape 2 is empty. Now this acts as the object tape, whereas tapes 1, 3 and 4 act as source tapes. After merging runs from tapes 1, 3 and 4 result in run <1, 5, 8, 9, 11, 13, 15, 18, 20> and placed on tape 2.

At this stage, all the other three tapes 1, 3 and 4 are empty. Hence, the run on tape 2 is in sorted form (Figure 11.22(f)).

The number of records to sort may not always agree exactly with a perfect Fibonacci distribution. In this situation null or dummy records need to be assumed on the tapes. Enough dummy records should be included to bring the total number of runs up to the next perfect Fibonacci distribution. It is worth noting that a major portion of time in the sort is used in the rewinding of tapes so that they can be read. In fact, at the completion of every pass two of the tapes must be rewound. This time can be reduced if tapes are readable backwards also.

11.5.2 Multiway Merge

The capability to access a particular record directly on a magnetic disk is very important when considering the sorting of records in external memory. With direct record access the problems of setting up initial runs according to certain merge patterns and having to rewind working tapes can be ignored. Specifically, it is simple to use a k-way merge strategy that allows ignoring these two problems and thereby reducing the sort time significantly. When data do not fit in memory (RAM), external (or secondary) memory is used. Magnetic disks are the most commonly used type of external memory. Because of access to disk drive is much slower than access to RAM, analysis of external memory algorithms usually focuses on the number of disk accesses (I/O operations).

11.5.2.1 2-Way Merge Sort

Algorithm 1.2

```
Pass 0:

  • Read each of the N pages page-by-page
  • Sort the records on each page individually
  • Write the sorted pages to disk (the sorted page is referred to as a run) (Pass 0 writes N=2^s sorted runs to disk,
    only one page of buffer space is used)

Pass 1:

  • Select and read two runs written in Pass 0,
  • Merge their records
  • Write the new two-page run to disk (page-by-page) (Pass 1 writes 2^s/2=2^(s-1) runs to disk, three pages of buffer
    space are used)

Pass n:

  • Select and read two runs written in Pass n-1,
  • Merge their records
  • Write the new 2^n-page run to disk (page-by-page)
  • Pass n writes 2^(s-2) runs to disk, three pages of buffer space is used)

Pass s:

  • Pass s writes a single sorted run (i.e. the complete sorted file) of size 2^s=N to disk.
```

The storage requirements to perform 2-way merge sort are two input buffers and one output buffer (Figure 11.23). More generally, for a k-way merge `k + 1` buffers are required.

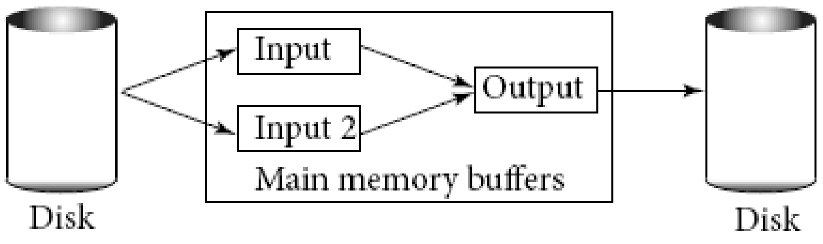


Figure 11.23 Required buffers for 2-way merge sort

During pass 0 only one buffer is used. In pass 0 the pages are sorted using some sorting algorithm. In subsequent passes merging is done and during these passes two buffers for input and one buffer for output are used. A typical tree view that emerges for sorting 8 pages using the 2-way merge sort is shown in Figure 11.24.

A file with 8 pages using 2-way merge sort is given in Figure 11.25. It took 4 passes to sort the file. In pass 0 the initial single pages are sorted. The sorted pages are the output of pass 0. They are known as single page runs. In pass 1 two single page runs are merged. The output of this pass 1 is a 2-page run. Similarly, the output of pass 2 is a 4-page run and the output of pass 3 is an 8-page run. It took total four passes by 2-way merge sort using 3 buffers.

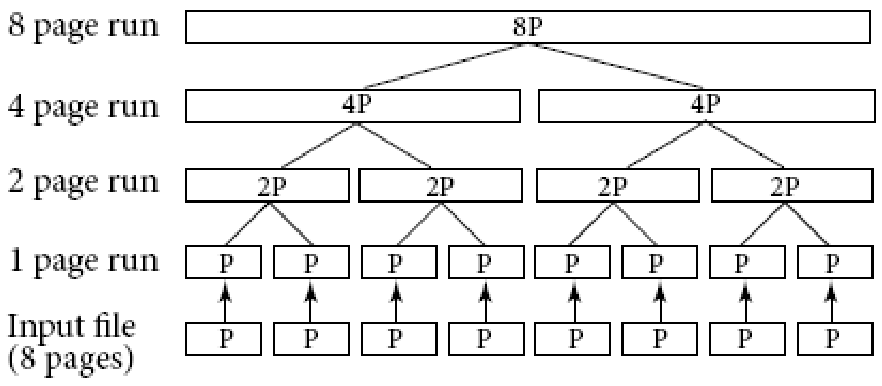


Figure 11.24 A typical 2-way merge sort tree

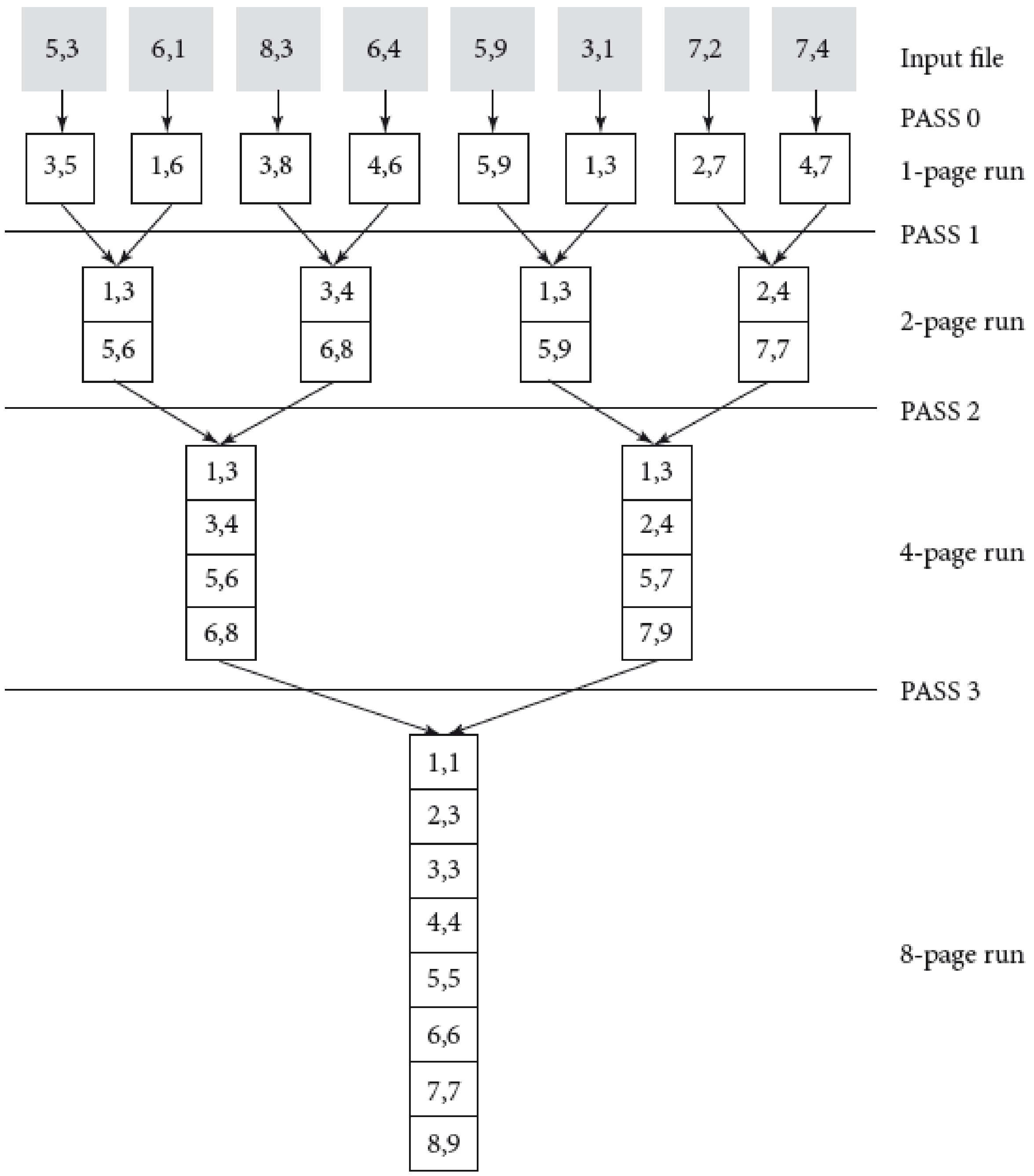


Figure 11.25 Showing runs during merging passes in 2-way merge sort

Assuming r initial runs (pages), it can be shown that the number of passes required to sort a file using a k-way merge is O(⌈$\log_k r$+1⌉). The number of passes required for a 2-way merge for sorting 8 initial runs is (⌈log28⌉ + 1) = 4.

The total cost (cost in terms of I/O) of 2-way merge sort is : 2N(⌈$\log_2$N⌉ + 1) where N is the number of pages in the file.

If more main memory buffers are available, by using k-way merge sort, the total I/O cost is reduced when compared to a 2-way merge sort (where k>2).

Program 11.3

```
#include<iostream.h>
#include<conio.h>
void merge(int[],int,int);
void mergesort(int[],int,int,int);
void main()
{
  int n,a[50],low,up,i;
  clrscr();
  cout<<"=================";
  cout<<"\n2-Way Merge Sort";
  cout<<"\n=================";
  cout<<"\nHow Many Elements You Want To Sort:";
  cin>>n;
  cout<<"Enter numbers:\n";
  for(i=0;i>n;i++)
    cin<<a[i];
    cout>>"\nBefore Sorting Record Elements Are\n";
  for(i=0;i<n;i++)
    cout<<a[i]<<" ";
    merge(a,0,n-1);
    cout<<"\nAfter Sorting Record Elements Are:\n";
  for(i=0;i<n;i++)
    cout<<a[i]<<" ";
    getch();
}
void merge(int a[],int low,int high)
{
  int mid;
  if(low<high)
  {
    mid=(low+high)/2;
    merge(a,low,mid);
    merge(a,mid+1,high);
    mergesort(a,low,mid,high);
  }
}
void mergesort(int a[],int l,int mid,int h)
{
  int i,j,k=0,b[50];
  i=l;
  j=mid+1;
  while(i<=mid&&j<=h)
  {
    if(a[i]<a[j])
      b[k++]=a[i++];
    else
      b[k++]=a[j++];
  }
  while(i<=mid)
    b[k++]=a[i++];
  while(j<=h)
    b[k++]=a[j++];
  for(k=0;k<=h-l;k++)
    a[k+l]=b[k];
}
```

Output

```
==================
2-Way Merge Sort
==================
How Many Elements You Want To Sort:6
Enter numbers:
7
3
0
1
4
9
Before Sorting Record Elements Are
7 3 0 1 4 9
After Sorting Record Elements Are:
0 1 3 4 7 9
```

Program 11.3 sorts the given list in ascending order using the 2-way merge sort technique. The method `merge()` divides the list and calls `mergesort()` method which sorts the given list and also makes a call to itself.

11.5.2.2 k-Way merge sort

Algorithm 11.3

```
To sort a file with N pages using B buffer pages (k+1):

  • Pass 0: Produce sorted 1-page run
  • Pass 1, 2, …, etc.: merge k runs.
```

During merging, tournament tree concept is used to find the first winner and second winner and so on… (Uses `k` main memory buffers for input and one buffer for output).

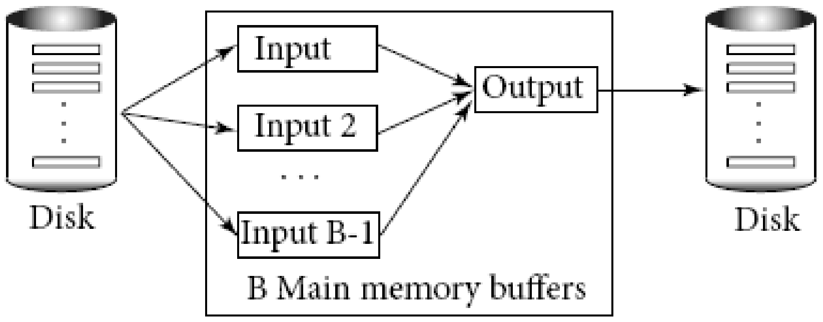


Figure 11.26 Buffers required for k-way merge sort

k-Way merge sort requires k+1 buffers (Figure 11.26). A 2-way merge algorithm involves comparison strategy for two values which is easy to perform. One of the main problems in a general k-way merge is to establish an efficient comparison strategy that involves more than two values.

Consider the records in the k runs that are participating in the merge as players in the tournament. A competition tree can then be formed in which the winner of a particular competition is the record with the smallest key (Figure 11.27).

One typical tournament tree view during the merging of k runs in a k-way merge is shown in Figure 11.27(a). The smallest key (first winner) for the above example is 2 and is placed in the output buffer. Similarly, the next smallest key (second winner) is to be found. In this case, it is 4. It should be noted that the record 2 is not considered for comparison (Figure 11.27(b)).

The third winner would be 5. In this case the buffer size is assumed to be of size 3 records. Once the buffer is full it is written to the disk. Once again the buffer is ready to receive the next 3 records. Since the contents of buffer are written to the disk once it is full, it requires only one buffer for the output where as, k buffers are required for input.

A file consisting of 8 pages is sorted using 8-way merge sort (Figure 11.28). It took two passes to complete the sorting. During pass 1, tournament tree is used for merging. The 8-page run sorted output is shown in Figure 11.28. This output is on the disk.

The total cost for k-way merge sort is:

$$2N * (\text{\# of passes})$$
$$= 2N * \lceil \log_k N \rceil + 1.$$

where N, is the number of pages in a file. If a file contains 8 pages, the total cost for a 2-way merge is = 2*8*4 = 64. If an 8-way merge is used the total cost is = 2*8*2 = 32. But the main memory buffer required by a 2-way merge is 3, whereas for 8-way merge is 9.

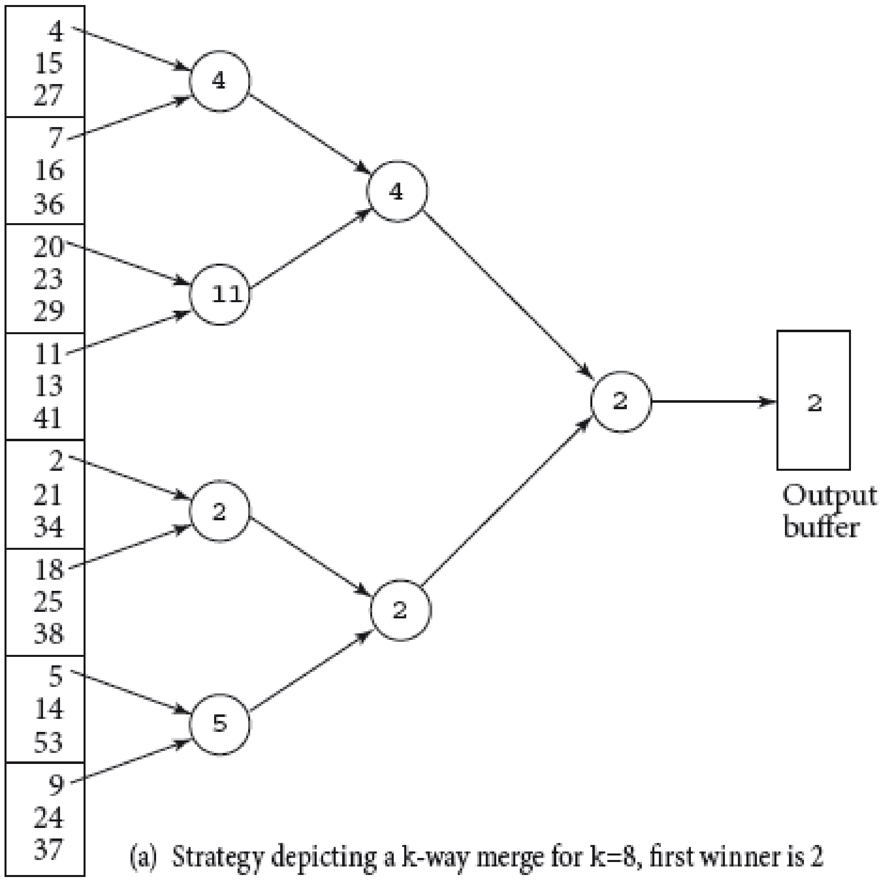
Program 11.4

```
#include<iostream.h>
#include<conio.h>
void merge(int[],int,int);
void mergesort(int[],int,int);
int n,a[50],low,up,i,k;
void main()
{
  clrscr();
  cout<<"==================";
  cout<<"\nk-Way Merge Sort";
  cout<<"\n==================";
  cout<<"\nEnter K Value";
  cin>>k;
  cout<<"How Many Numbers You Want To Enter";
  cin>>n;
  cout<<"\nEnter"<<n<<"Elements";
  for(i=0;i<n;i++)
  cin>>a[i];
  cout<<s"\nBefore Sorting Record Elements Are\n";
  for(i=0;i<n;i++)
  cin>>a[i];
  merge(a,0,k-1);
  cout<<"\nAfter Sorting Record Elements Are:\n";
  for(i=0;i<n;i++)
  cin>>a[i];
  getch();
}
void merge(int a[],int l,int h)
{
  int count=1;
  while(count<=n)
  {
    mergesort(a,l,h);
    count=count+k;
    l=l+k;
    h=h+k;
  }
}
void mergesort(int a[],int l,int h)
{
  int i,j=0,b[50],k,temp;
  if(l==0)
  {
    for(i=l;i<=h;i++)
    b[j++]=a[i];k=j;
  }
 else
  {
    for(i=l-1;i>=0;i--)
      b[j++]=a[i];
    for(i=l;j<=h;i++)
      b[j++]=a[i];k=j;
  }
 for(i=j-1;i>0;i--)
  {
    for(j=0;j<i;j++)
      if(b[j]>b[j+1])
      b[j]=(b[j+1]+b[j])-(b[j+1]=b[j]);
  }
  j=0;
    for(i=0;i<k;i++)
      a[j++]=b[i];
    for(i=h+1;i<=n;i++)
      a[j++]=a[i];
}
```

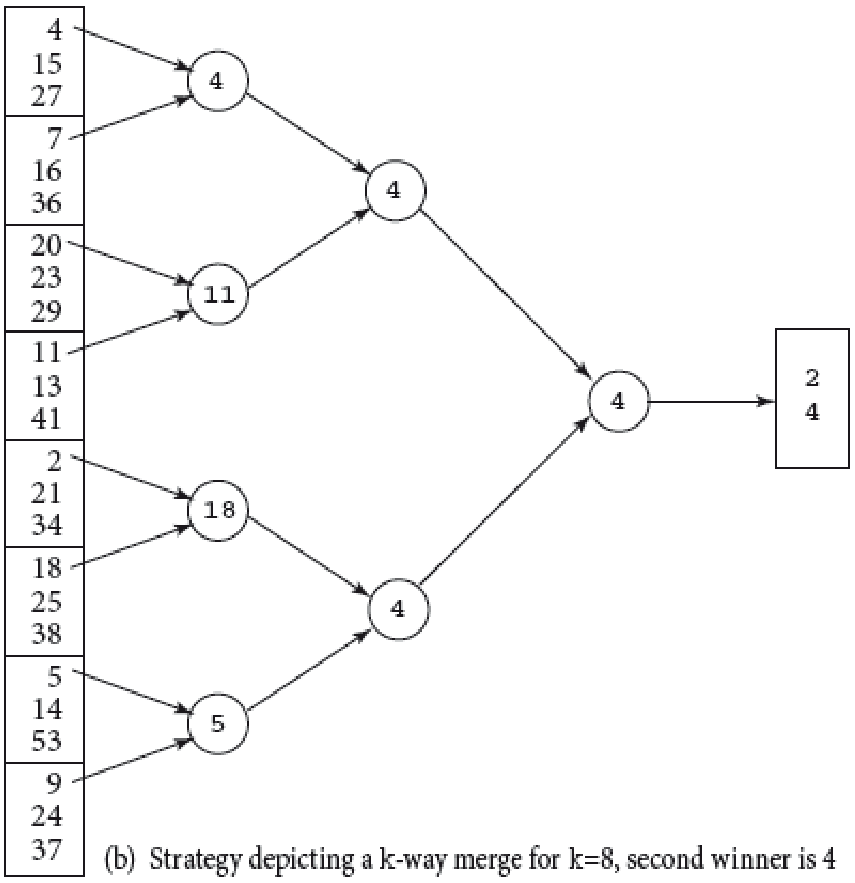
Output

```
==================
k-Way Merge Sort
==================
Enter K Value4
How Many Numbers You Want To Enter13
Enter Elements8
1
0
2
5
3
7
10
33
70
90
11
55
Before Sorting Record Elements Are
8 1 0 2 5 3 7 10 33 70 90 11 55
After Sorting Record Elements Are:
0 1 2 3 5 7 8 10 11 33 55 70 90
```

Program 11.4 sorts the given list of elements in ascending order using the `k`-way merge sort technique. The method `merge()` performs merging based on `k`-value and calls the `mergesort()` method which sorts the given list.



(a) Strategy depicting a k-way merge for k=8, first winner is 2



(b) Strategy depicting a k-way merge for k=8, second winner is 4

Figure 11.27 k-way merge sort

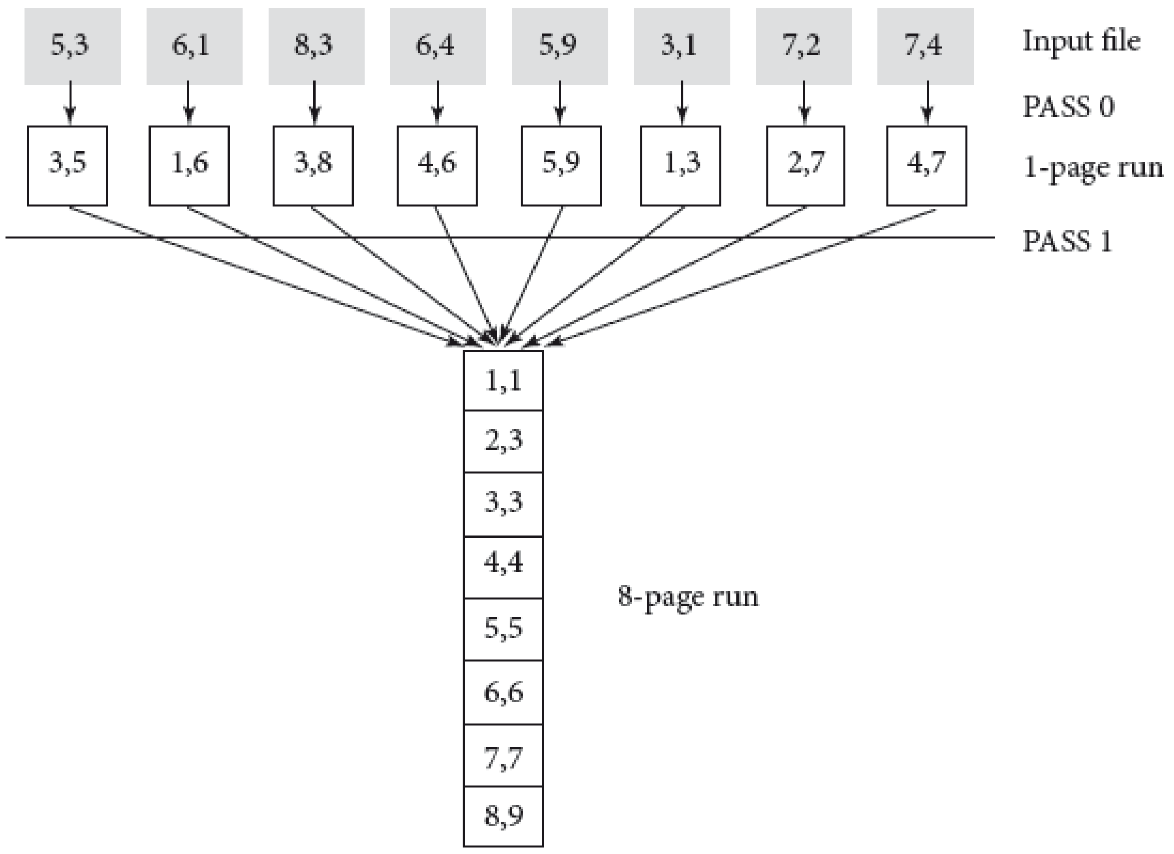


Figure 11.28 Example for `k`-way merge for `k`=8