Chapter 7 Sorting - part 2

South China University of Technology

College of Software Engineering

Huang Min

Sorting Lower Bound

Sorting Lower Bound

- The upper bound for a problem can be defined as the asymptotic cost of the fastest known algorithm. The lower bound defines the best possible efficiency for any algorithm that solves the problem, including algorithms not yet invented.
- Once the upper and lower bounds for the problem meet, we know that no future algorithm can possibly be (asymptotically) more efficient.
- Sorting is O(n log n) (average, worst cases) because we know of algorithms with this upper bound.
- We would like to know a lower bound for all possible sorting algorithms. We will now prove Ω(n log n) lower bound for sorting.

Decision Trees

Decision tree: A decision tree is a binary tree that can model the processing for any algorithm that makes binary decisions.

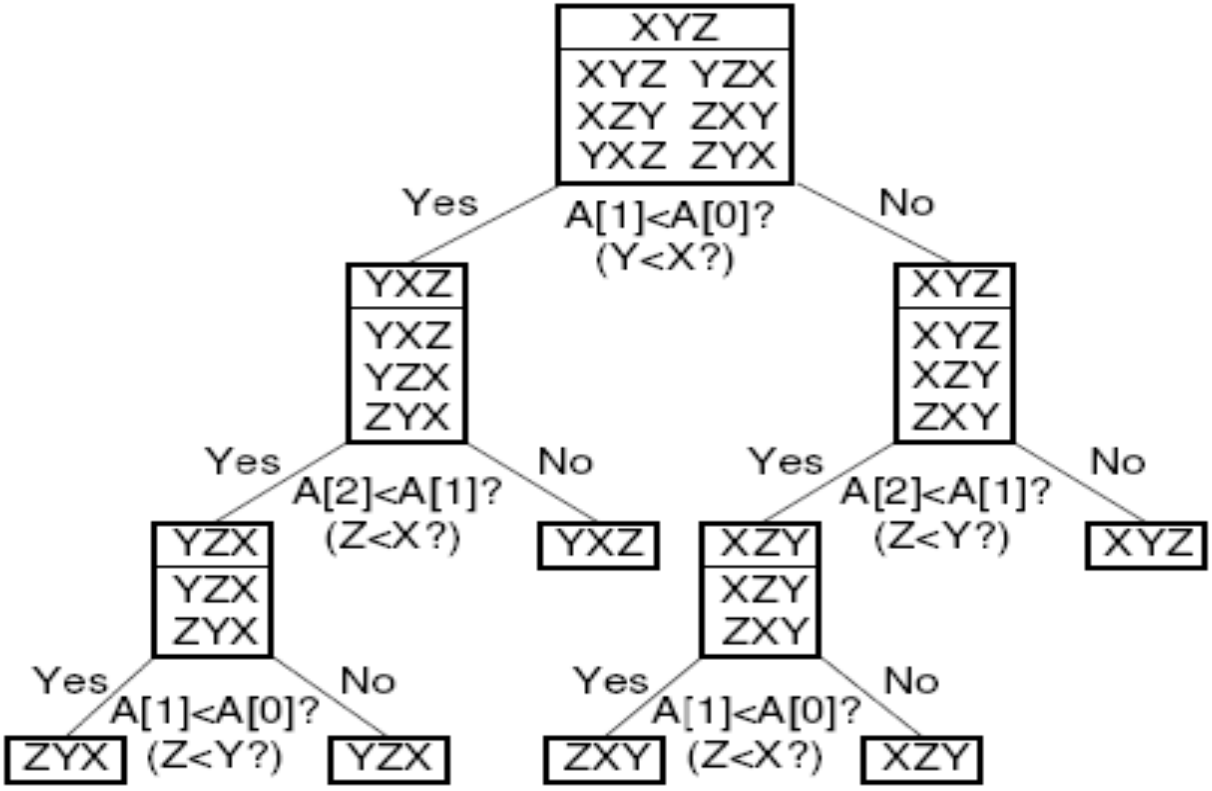


Figure 7.21 Decision tree for Insertion Sort when processing three values labeled X, Y, and Z, initially stored at positions 0, 1, and 2, respectively, in input array A.

Lower Bound Proof

- A sorting algorithm based on comparison can be viewed as for any permutation has been input, and each leaf node of the decision tree corresponds to one permutation.
- There are $n!$ permutations, so the decision tree has $n!$ leaves.
- 1. A binary tree of height n can store at most 2^n -1 nodes.
  2. Equivalently, a binary tree with n nodes is at least (log(n + 1)) levels nodes. (it is a complete binary tree)
- A tree with n nodes has $\Omega(\log n)$ levels, so the tree with n! leaves has $\Omega(\log n!) = \Omega(n \log n)$ levels .

Linear-time sorts: Bucket sort and Radix sort

A simple, efficient sort:

```
for (i=0; i<n; i++)
  B[A[i]] = A[i];
```

Ways to generalize:

- Allow for duplicate values among the keys: Make each bin the head of a list.
- Allow for a key range greater than *n*:

  For example, a set of n records might have keys in the range 1 to 2n, the only requirement is that each possible key value have a corresponding bin in B.

Bucket sort

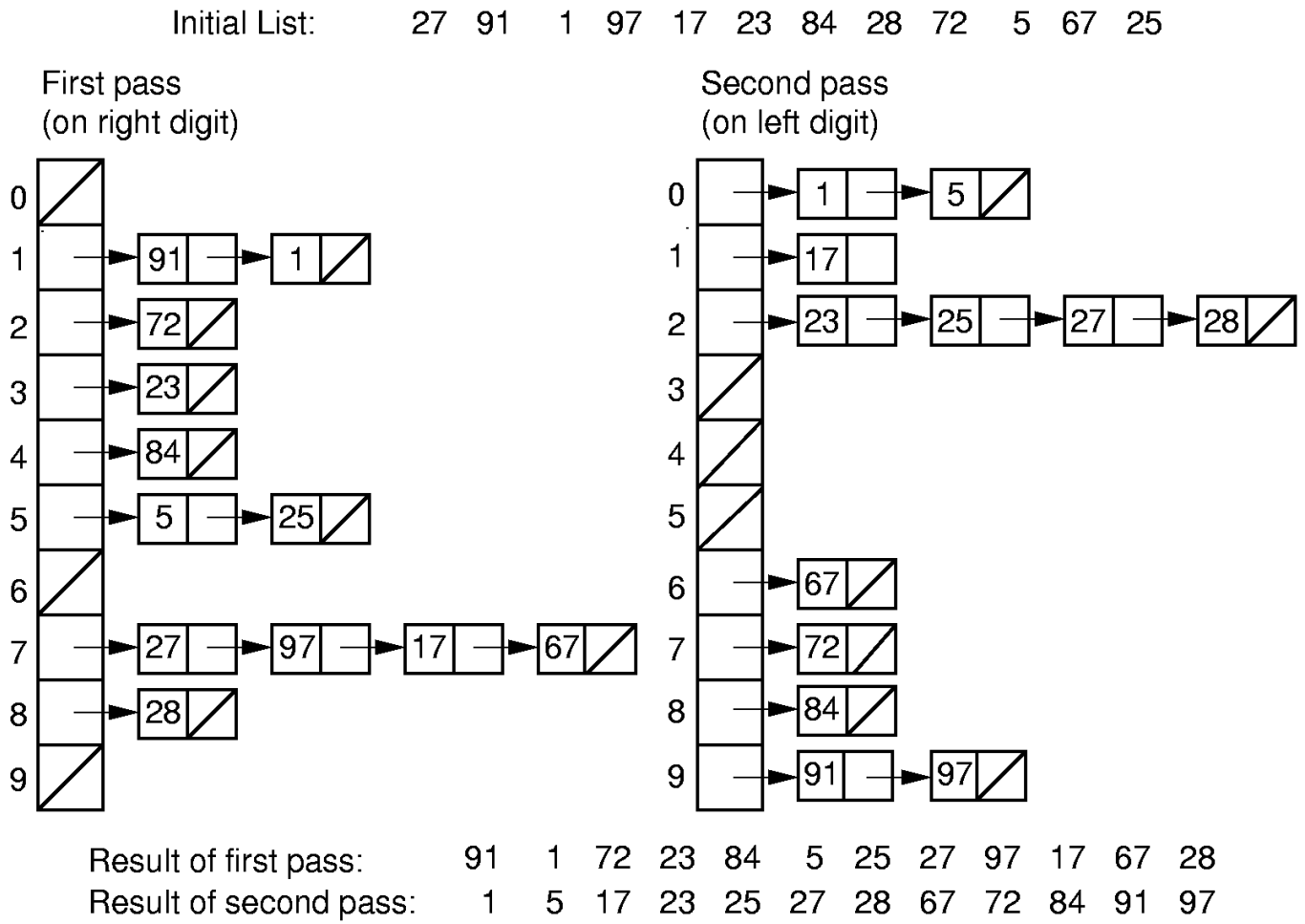
Bucket sort

```
template <typename E, class getKey>
void binsort(E A[], int n) {
  List<E> B[MaxKeyValue];
  E item;
  for (int i=0; i<n; i++)
    B[A[i]].append(getKey::key(A[i]));
  for (int i=0; i<MaxKeyValue; i++)
    for (B[i].setStart();
      B[i].getValue(item); B[i].next())
        output(item);
}
```

Cost: $\Theta(n + \text{MaxKeyValue})$.

Radix Sort

Radix Sort


Initial List: 27 91 1 97 17 23 84 28 72 5 67 25
First pass
(on right digit)
Second pass
(on left digit)
0
1
2
3
4
5
6
7
8
9
91
1
72
23
84
5
25
27
97
17
67
28
1
5
17
23
25
27
28
67
72
84
91
97
Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Radix Sort Example

Initial Input: Array A

27	91	1	97	17	23	84	28	72	5	67	25

First pass values for Count.
rtok = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

End of Pass 1: Array A.

91	1	72	23	84	5	25	27	97	17	67	28

Second pass values for Count.
rtok = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97

Radix Sort

```
template <typename E, typename getKey>
void radix(E A[], E B[],
           int n, int k, int r, int cnt[]) {
  // cnt[i] stores number of records in bin[i]
  int j;
  for (int i=0, rtoi=1; i<k; i++, rtoi*=r) {//For k digits
    for (j=0; j<r; j++) cnt[j] = 0; // Initialize cnt
    // Count the number of records for each bin on this pass
    for (j=0; j<n; j++) cnt[(getKey::key(A[j])/rtoi)%r]++;
    // Index B: cnt[j] will be index for last slot of bin j.
    for (j=1; j<r; j++) cnt[j] = cnt[j-1] + cnt[j];
    // Put records into bins, work from bottom of each bin.
    // Since bins fill from bottom, j counts downwards
    for (j=n-1; j>=0; j--)
      B[--cnt[(getKey::key(A[j])/rtoi)%r]] = A[j];
    for (j=0; j<n; j++) A[j] = B[j]; // Copy B back to A
  }
}
```

Radix Sort Cost

Cost: $\Theta((n+r)k=nk + rk)$

How do *n*, *k*, and *r* relate?

r is usually small and can be viewed as a constant, and if If key range is small, then k can also be viewed as a constant, then this can be Θ (kn)=Θ (n).

More accurately, If there are *n* distinct keys, then the length of a key (the value of *k*) must be at least $\log_r n$.

- Thus, Radix Sort is $\Theta(n \log_r n)$ in general case

External Sorting

External Sorting

Problem: Sorting data sets too large to fit into main memory.

- Assume data are stored on disk drive.

To sort, portions of the data must be brought into main memory, processed, and returned to disk.

An external sort should minimize disk accesses.

Model of External Computation

Secondary memory is divided into equal-sized blocks (512, 1024, etc…)

A basic I/O operation transfers the contents of one disk block to/from main memory.

Primary goal is to minimize I/O operations.

Assume only one disk drive is available.

Key Sorting

Often, records are large, keys are small.

- Ex: Payroll entries keyed on ID number

Approach 1: Read in entire records, sort them, then write them out again.

Approach 2: Read only the key values, store with each key the location on disk of its associated record.

After keys are sorted the records can be read and rewritten in sorted order.

Simple External Mergesort

Quicksort requires random access to the entire set of records.

Better: Modified Mergesort algorithm.

- Process n elements in $\Theta(\log n)$ passes.

A sorted sublist is called a *run*.

Simple External Mergesort

Our approach to external sorting is derived from the Mergesort algorithm.

The simplest form of external Mergesort performs a series of sequential passes over the records, merging larger and larger sublists on each pass.

- The first pass merges sublists of size 1 into sublists of size 2;
- the second pass merges the sublists of size 2 into sublists of size 4; and so on.

Thus, each pass is merging pairs of runs to form longer runs. Each pass copies the contents of the file to another file. is a sketch of the algorithm

Simple External Mergesort

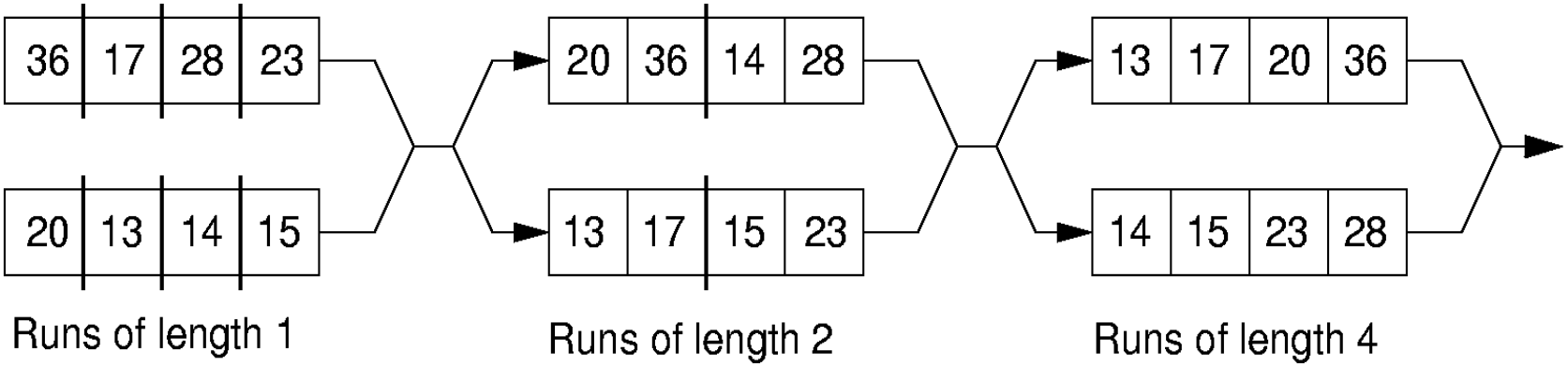


Figure 8.6 A simple external Mergesort algorithm. Input records are divided equally between two input files. The first runs from each input file are merged and placed into the first output file. The second runs from each input file are merged and placed in the second output file. Merging alternates between the two output files until the input files are empty. The roles of input and output files are then reversed, allowing the runlength to be doubled with each pass.

Problems with Simple Mergesort

How can we reduce the number of Mergesort passes?
Not use Mergesort on small runs, read in a block of data and sort it in memory, and then output it as a single sorted run.

Over the years, many variants on external sorting have been presented, but all are based on the following two steps:

- Break the files into large initial runs with Replacement Selection
- Merge the runs together to form a single sorted run file.

Breaking a File into Runs

General approach:

- Read as much of the data of file into memory as possible.
- Perform an in-memory sort such as Quicksort or Heapsort.
- Output this group of sorted records as a single run.

Replacement Selection

- A basic I/O operation is read/write a block. Replacement selection sort is a technology of generating a run as larger as possible based on the basic I/O operations.
- It is a slight variation on Heapsort algorithm.
- Break available memory into an array for the heap, an input buffer, and an output buffer.

Replacement Selection

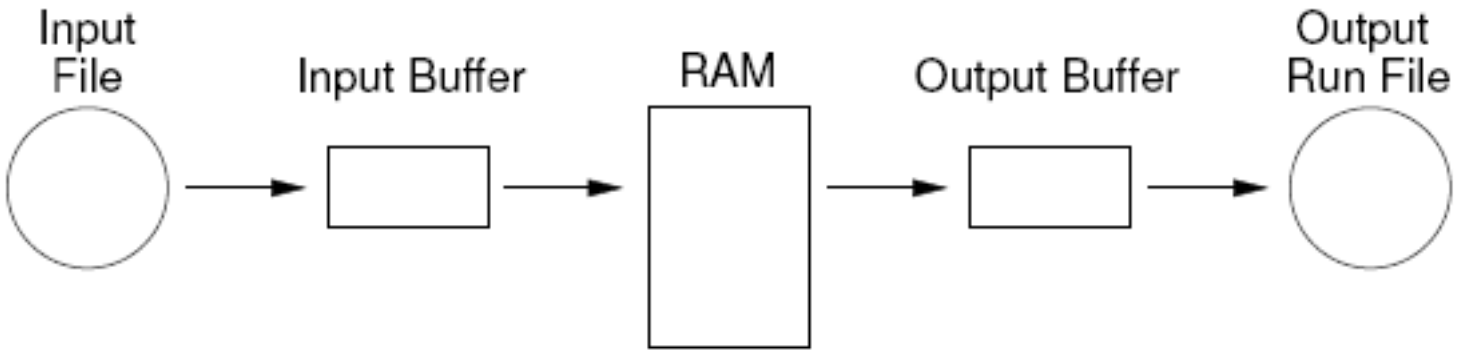


Figure 8.7 Overview of replacement selection. Input records are processed sequentially. Initially RAM is filled with M records. As records are processed, they are written to an output buffer. When this buffer becomes full, it is written to disk. Meanwhile, as replacement selection needs records, it reads them from the input buffer. Whenever this buffer becomes empty, the next block of records is read from disk.

Replacement Selection

1) Fill the array from disk. Set LAST=M-1;
2) Build a min-heap;
3) Repeat until the array is empty:
   a) Send the record with the minimum key value (the root) to the output buffer.
   b) Let R' be the next record in the input buffer. If R's key value is greater than the key value just output, then
      - Then place R at the root.
      - Else replace the root with the record in array position LAST, and place R at position LAST. Set LAST=LAST-1.
   c) Sift down the root to recorder the heap.

RS Example

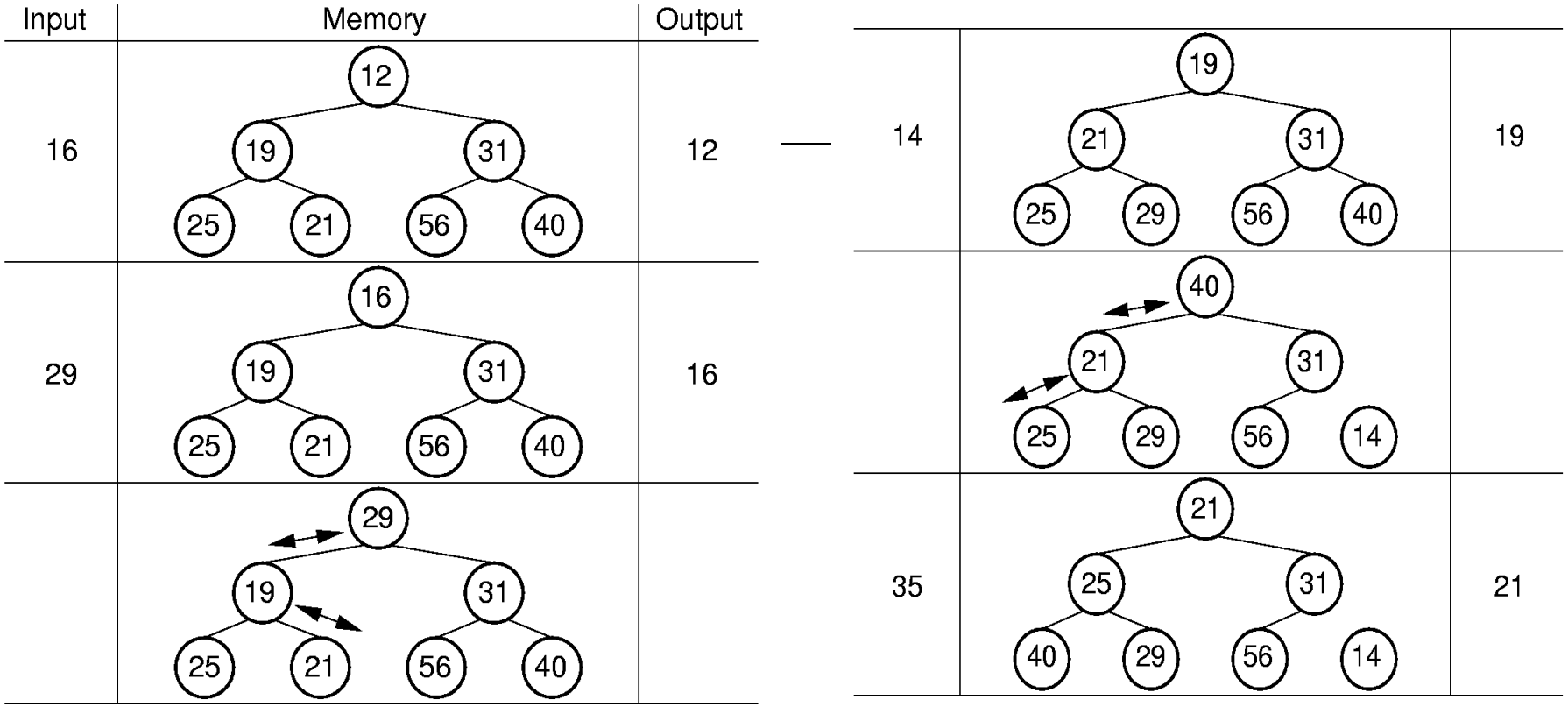


Figure 8.8 Replacement selection example. After building the heap, root value 12 is output and incoming value 16 replaces it. Value 16 is output next, replaced with incoming value 29. The heap is reordered, with 19 rising to the root. Value 19 is output next. Incoming value 14 is too small for this run and is placed at end of the array, moving value 40 to the root. Reordering the heap results in 21 rising to the root, which is output next.

Replacement Selection (3)

1) Only the record whose key value is greater than the last key value output can be added to the heap.
2) The record with smaller key value can not be output as a part of the run, but is stored as the data to be handled for the next run;
   --Heap becomes smaller and smaller and the discarded heap space is used to store the records not be output.
3) If the size of available RAM used by array is M, then the smallest run generated by Replacement selection sort is M and the average run is 2M. (Proven process see “Snowplow analogy”)

Problems with Simple Merge

Simple mergesort: Place runs into two files.

- Merge the first two runs to output file, then next two runs, etc.

Repeat process until only one run remains.

- How many passes for r initial runs?
  $(\log_2 r)$

Need a way to reduce the number of passes.

Multiway Merge

With replacement selection, each initial run is several blocks long.

Assume each run is placed in separate file.

Read the first block from each file into memory and perform an *r*-way merge.

When a buffer becomes empty, read a block from the appropriate run file.

Multiway Merge

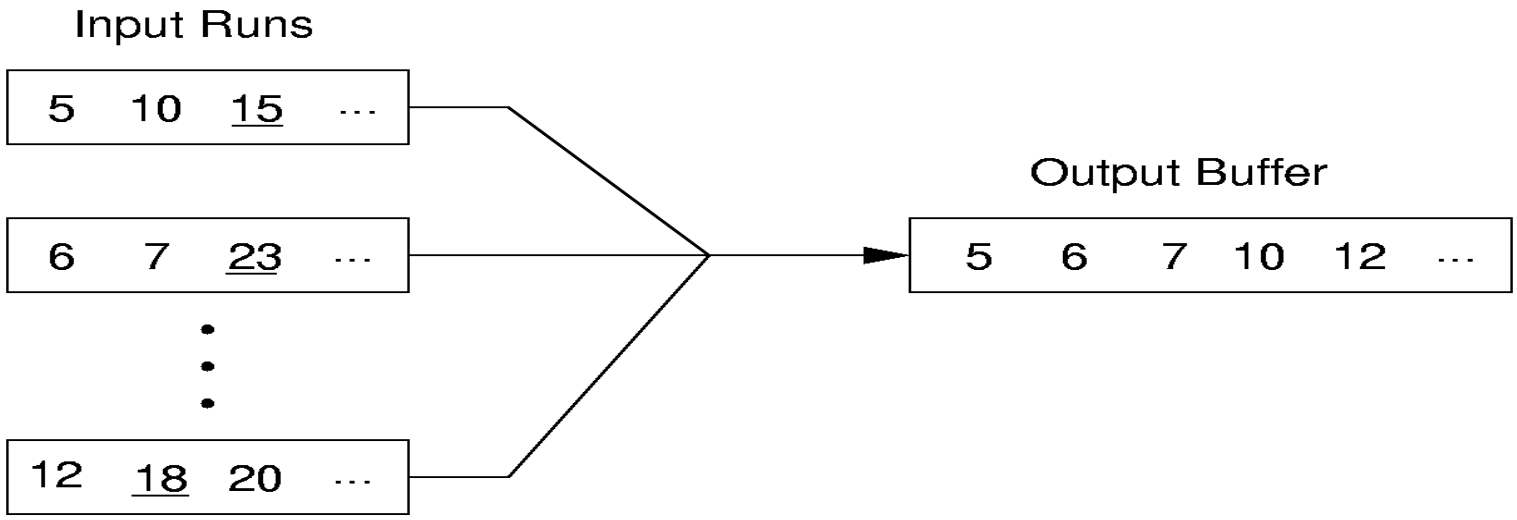


Figure 8.10 Illustration of multiway merge. The first value in each input run is examined and the smallest sent to the output. This value is removed from the input and the process repeated. In this example, values 5, 6, and 12 are compared first. Value 5 is removed from the first run and sent to the output. Values 10, 6, and 12 will be compared next. After the first five values have been output, the "current" value of each block is the one underlined.

Limits to Multiway Merge

Assume working memory is *b* blocks in size.

How many runs can be processed at one time? (b)

The runs are 2*b* blocks long (on average) initially.

How big a file can be merged in one pass? ($2b^2$)

In k merge passes, we process $2b^{(k+1)}$ blocks.

Examples

One example:

0.5MB working memory, a block is 4KB, yielding 128 blocks in working memory. The average run size is 1MB (twice the working memory size). In one pass, 128 runs can be merged. Thus, a file of size 128MB can, on average, be processed in two passes (one to build the runs, one to do the merge) with only 0.5MB of working memory.

Another example:

blocks are 1KB long and working memory is 1MB = 1024 blocks. Then 1024 runs of average length 2MB (which is about 2GB) can be combined in a single merge pass.

General Principles

A good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.

Classification of sorting algorithms

1）插入排序

直接插入排序、希尔排序

2）交换排序

冒泡排序、快速排序

3）选择排序

直接选择排序、堆排序

4）归并排序

5）基数排序

Stableness of sorting algorithms

1）插入排序、冒泡排序、归并排序、分配排序（桶式、基数）都是稳定的排序算法。

2）直接选择排序、堆排序、shell排序、快速排序都是不稳定的排序算法。