Mergesort

- Mergesort is a O(n log n) worst case sorting algorithm
- A variant of the "standard" mergesort algorithm is effective for minimizing external memory access when working with large data sets
- Note: Some of the content of this presentation is from course material provided for the text:

Data Structures and Algorithms in Java(4th edition)

by Michael T. Goodrich and Roberto Tamassia

Mergesort

```
mergesort(data)
    mergesort(left half of data);
    mergesort(right half of data);
    merge(data);
```

Mergesort

Mergesort is a divide and conquer algorithm with runtime complexity characterized by the recurrence relation:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + f(n)$$

where f(n) denotes the cost of merging.

Mergesort

```
mergesort(data)
    mergesort(left half of data);
    mergesort(right half of data);
    merge(data);
```

Divide

Mergesort

```
mergesort(data)
    mergesort(left half of data);
    mergesort(right half of data);
    merge(data);
```

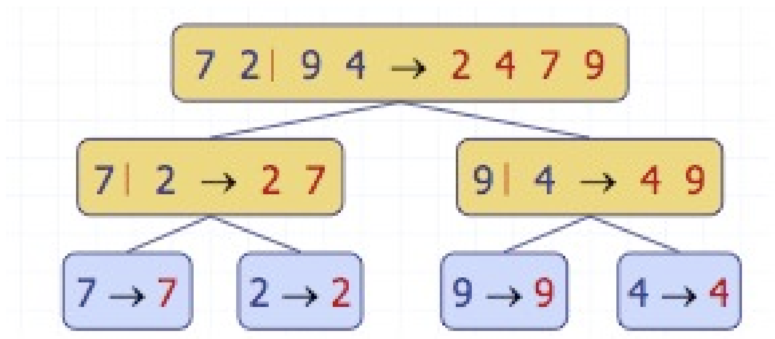
Conquer

Mergesort

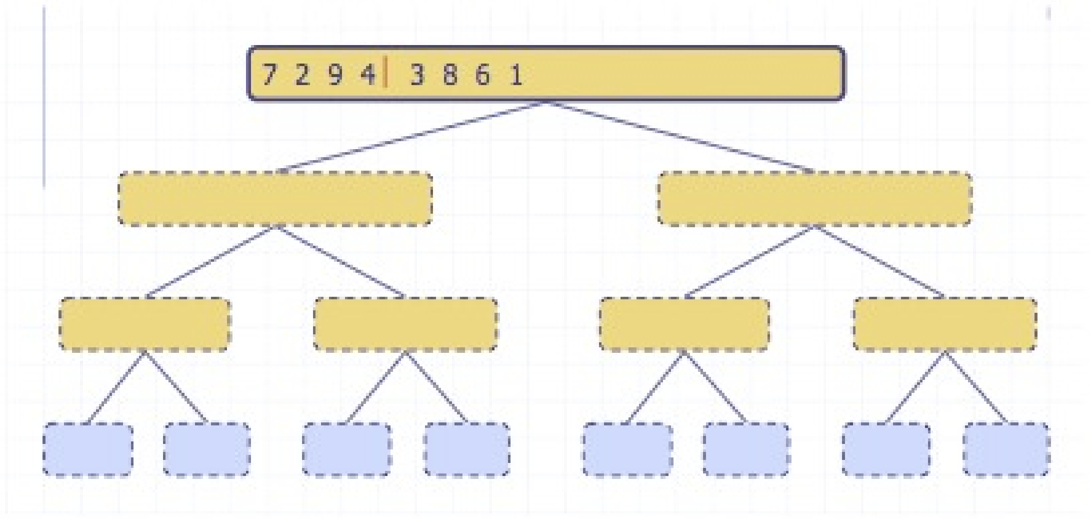
```
merge(data, first, last){
    mid = (first + last)/2;
    i1 = 0;
    i2 = first;
    i3 = mid + 1;
    while (both left and right sub-arrays of data contain elements){
    if (data[i2] < data[i3])
        temp[i1++] = data[i2++];
        else temp[i1++] = data[i3+];
    }
    copy the remaining elements of data into temp
    copy temp back into data
}
```

Thus the cost of merging is O(n)

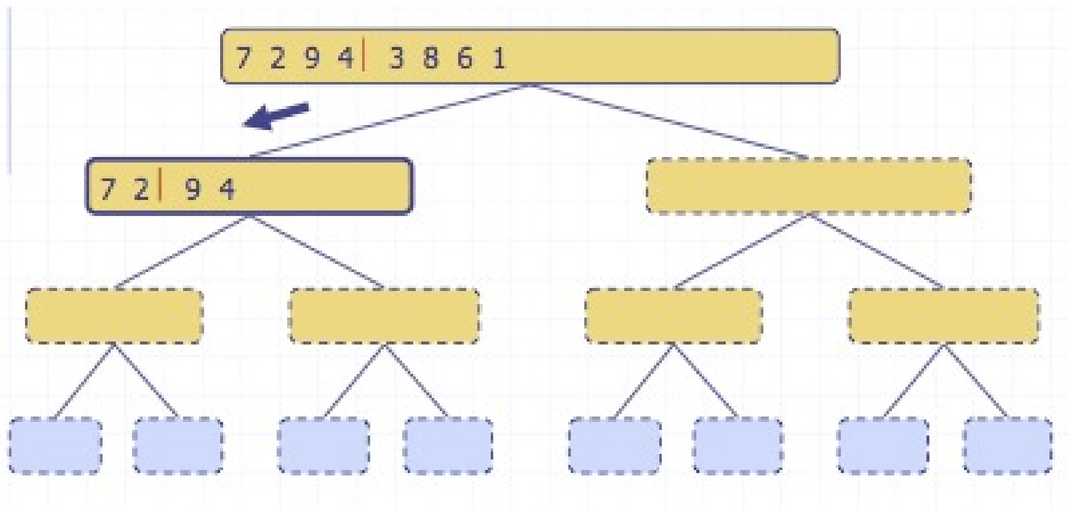
A "cartoon" description of mergesort



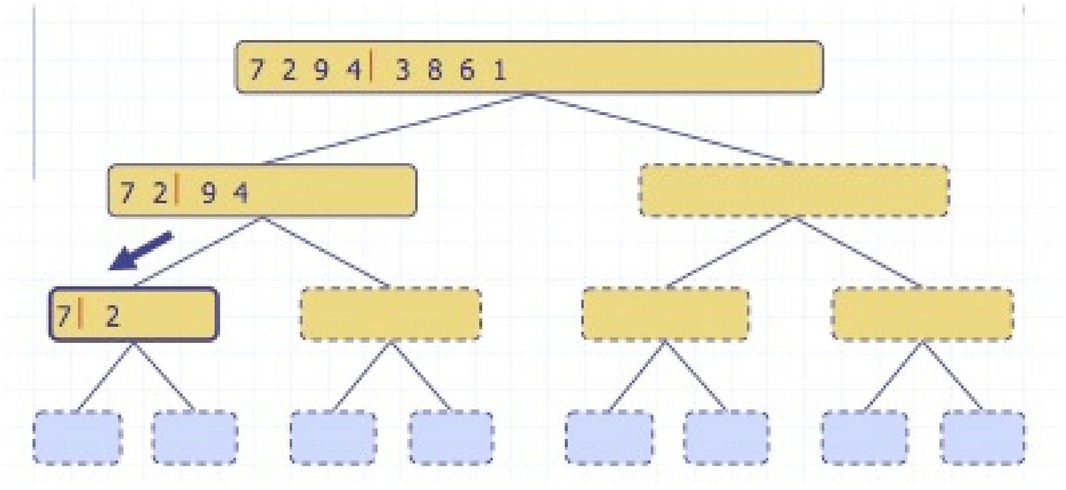
A "cartoon" description of mergesort



A "cartoon" description of mergesort

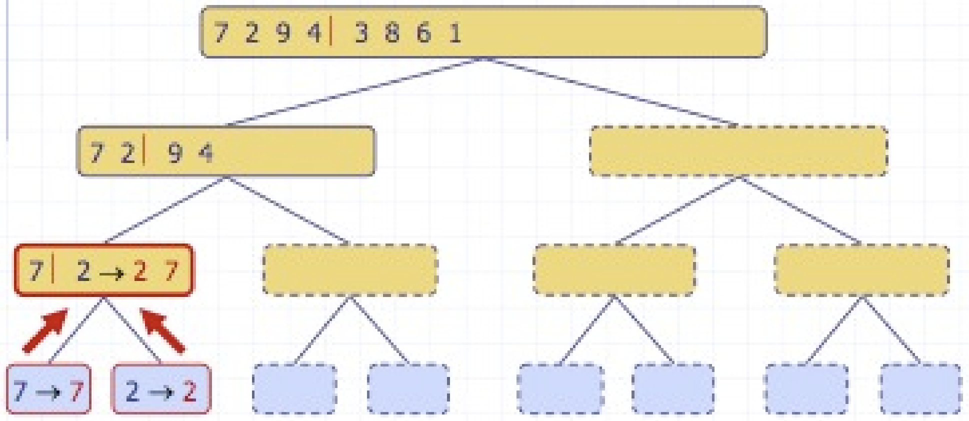


A "cartoon" description of mergesort



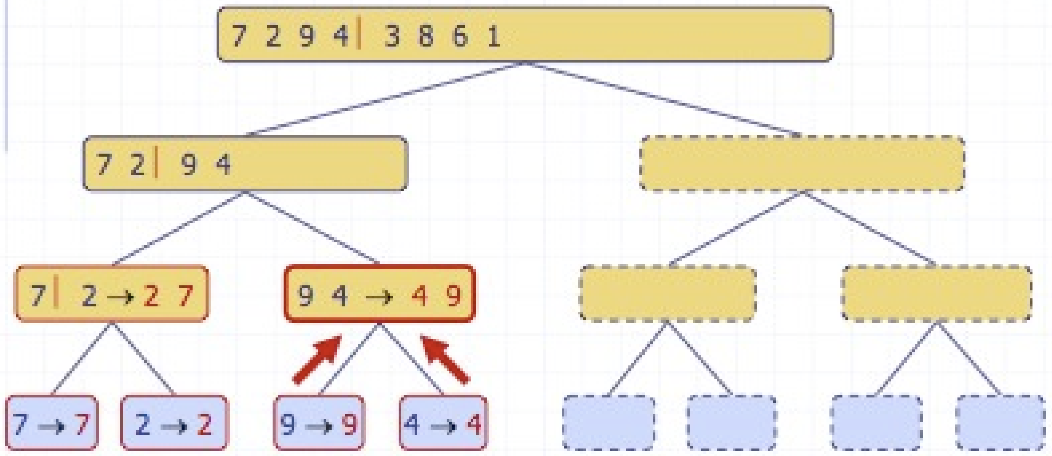
A "cartoon" description of mergesort

- Merge



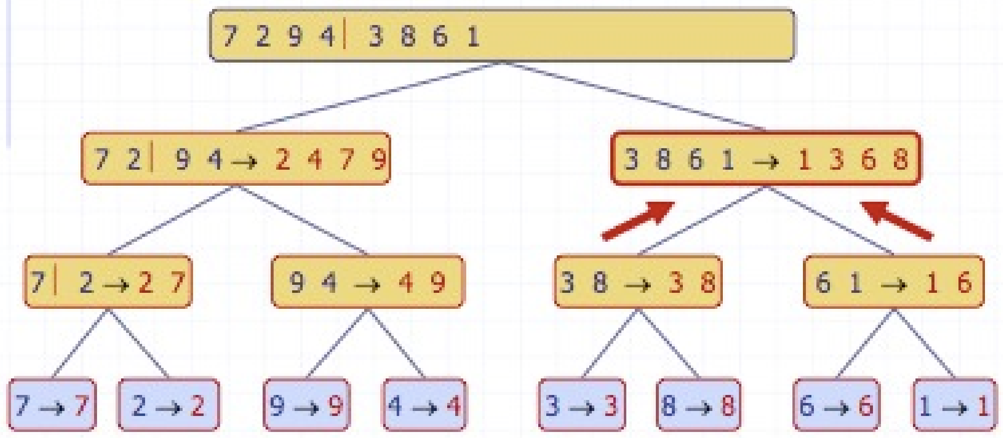
A "cartoon" description of mergesort

- Recursive call, ..., base case, merge



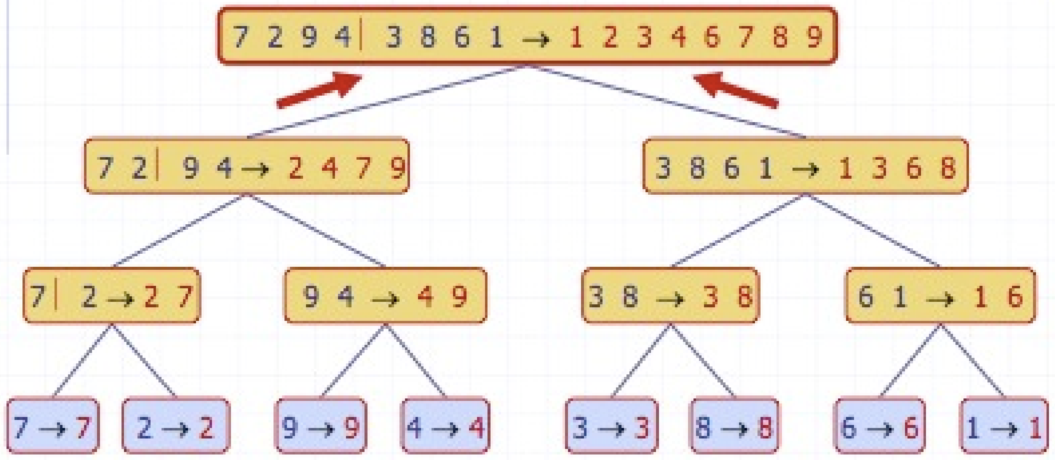
A "cartoon" description of mergesort

- Recursive call, …, merge, merge



A "cartoon" description of mergesort

- Merge



Mergesort

To solve the mergesort recurrence we make some simplifying assumptions. Let $n = 2^k$, and let's simply charge n for merging n items.
Thus we have:

$$T(2^k) = 2T(2^{k-1}) + 2^k$$

with the base case

$$T(2) = 2$$

We can solve the recurrence by expanding.

$$\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 2^k \\
&= 2(2T(2^{k-2}) + 2^{k-1}) + 2^k \\
&= 2^2T(2^{k-2}) + 2^k + 2^k \\
&\;\;\vdots \\
&= 2^{k-1}\,T(2) + 2^k \ldots + 2^k + 2^k \\
&= k2^k
\end{aligned}$$

So $T(n) = n \log_2 n$.

Mergesort

Consider the case where we want to sort a collection of data, however, the data does not fit into main memory, and we are concerned about the number of external memory accesses that we need to perform. Our data, consists of *N* *blocks or pages*, and the amount of free internal memory is B blocks.

Mergesort

Let us first determine the number of external memory accesses used by mergesort.

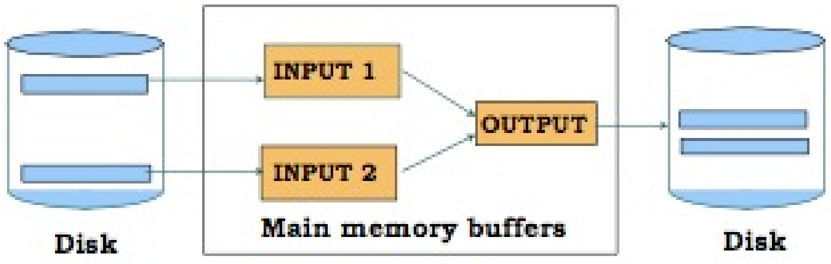
In pass 0 we sort the data into $\lceil N/B \rceil$ runs of size B using an efficient internal sorting algorithm.

In pass 1 we can merge the data into runs of size 2B.
Notice how we partition internal memory into 3 buffers.

In pass 2 we can merge the data into runs of size 4B.

⋮

In pass p we can merge the data into runs of size 2^pB.

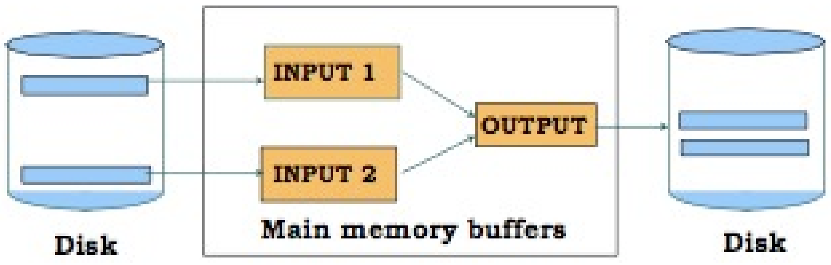


When p is such that $2^pB \geq N$ we have completely sorted the data. Notice that we perform 2 memory accesses (one read and one write) for each block of data per pass. So the total number of external memory accesses is

$$2N(p+1)$$

or

$$2N(\lceil (\log_2 N) \rceil + 1)$$



Mergesort

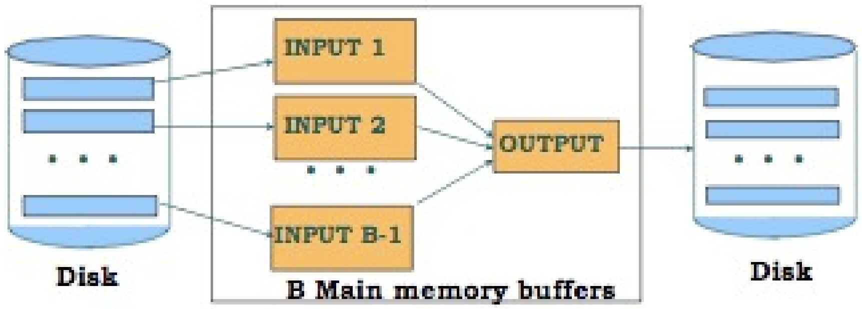
Rather than perform a two way merge we can merge k sorted runs per pass. With a 2 way merge the number of passes p is related to $\log_2 N$. If we perform a k way merge then the number of passes will be related to $\log_k N$. Thus we would like to make k as large as possible.

When p is such that $k^p B \geq N$ we have completely sorted the data. Notice that we perform 2 memory accesses (one read and one write) for each block of data per pass. So the total number of external memory accesses is

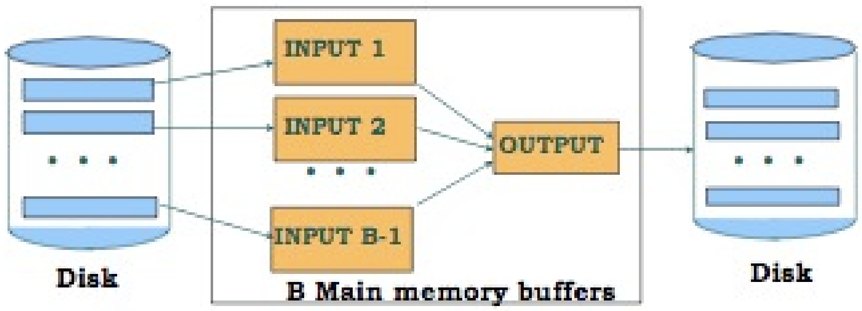
$$2N(p+1)$$

or

$$2\,N(\lceil(\log_k N)\rceil + 1)$$



To maximize the value of k and make the best use of memory we set k to B-1. This gives makes use of all B blocks of memory, B-1 input buffers and 1 output buffer.



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Summary

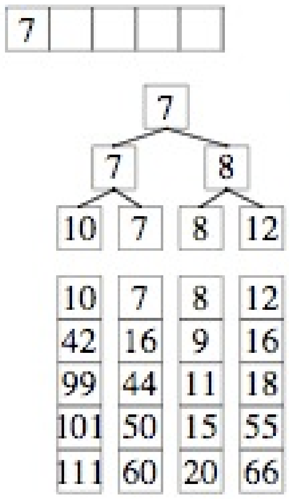
- External sorting is important; DBMS*may dedicate part of buffer pool for sorting!
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages). Later passes: ***merge*** runs.
  - # of runs merged at a time depends on ***B***, and ***block size***.
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs rarely more than 2 or 3.

*DBMS Data Base Management System

Multi-way merging

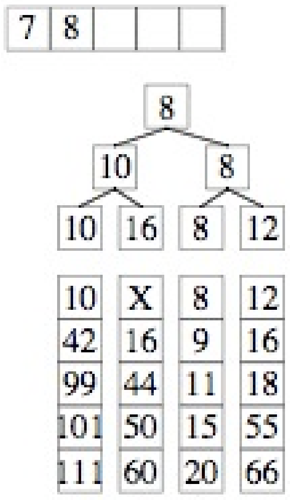
One detail remains, and that is how do we accomplish a k-way merge.

Multi-way merging



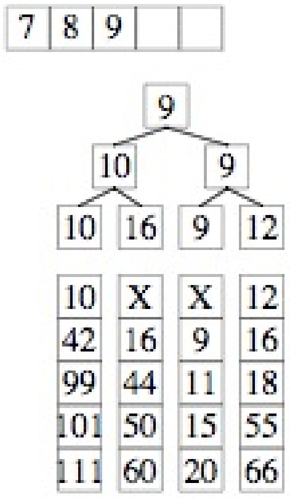
One method uses a *tournament tree* to determine the item that is put onto the output buffer.

Multi-way merging

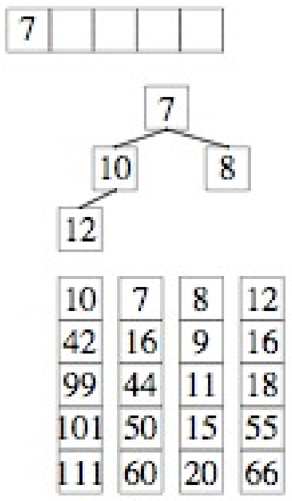


The buffer that provides the winner puts a new element into the tree.

Multi-way merging

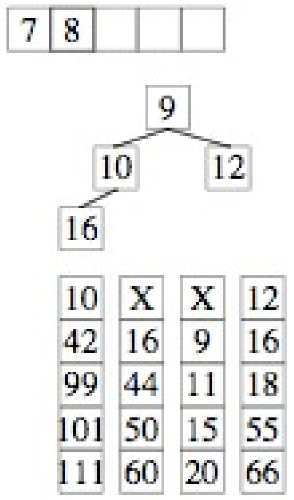


Multi-way merging

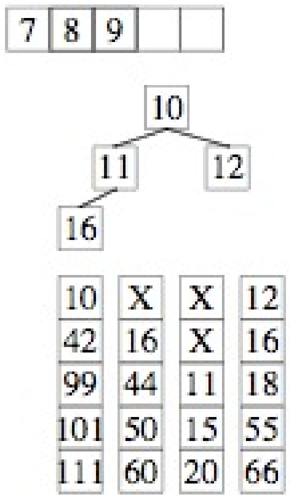


Or we can use a min heap to determine the item that is put onto the output buffer.

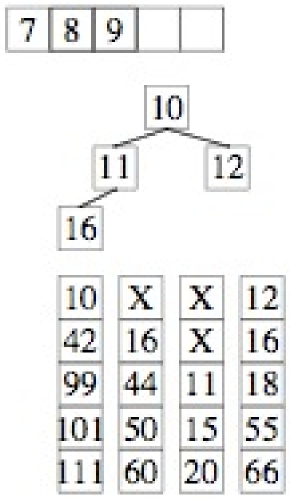
Multi-way merging



Multi-way merging



Multi-way merging



In either case a k-way merge is performed in O(log k) operations.