# Sorting a File of Records

Consider the problem of sorting a large file, stored on disk, containing a large number of logical records.

If the file is very large at all then it will be impossible to load all of the records into memory at once, and so the conventional in-memory sorting techniques will not work.

Obviously, some subset of the records must be in memory during the sort, and each record must be loaded into memory at some time.

However, at any given time, most of the records will exist only in secondary storage.

Aside from correctness, the principal goal of external sorting is to minimize the amount of time spent on disk accesses.

## Keys vs Records

"Keys are small but records are large."

E.g., consider sorting a file of book records by ISBN. A book record may contain a dozen or more fields, and occupy several hundred bytes. An ISBN will occupy at most 13 bytes.

Clearly we can store more key values in memory than we can store entire records.

So we could read a portion of the file, and build an in-memory list of key values, which could then be sorted using one of the techniques already discussed, such as Quicksort.

Of course, we must also store an address (file offset) along with the key value in order to locate the record.

The sorted list of keys could be used as an index for the unsorted records, or the records could be read and re-written in sorted order.

Note that it will often be impossible to store all of the keys in memory at once…

## Simple Mergesort

We could sort a file of records as follows:

1 Partition the file into two files, say F1 and F2.

2 Read in a block from each file.

3 Take the first record from each block and write them in sorted order (to a new file, say M1).

4 Repeat the process with the record from each block, but write to a second file, say M2.

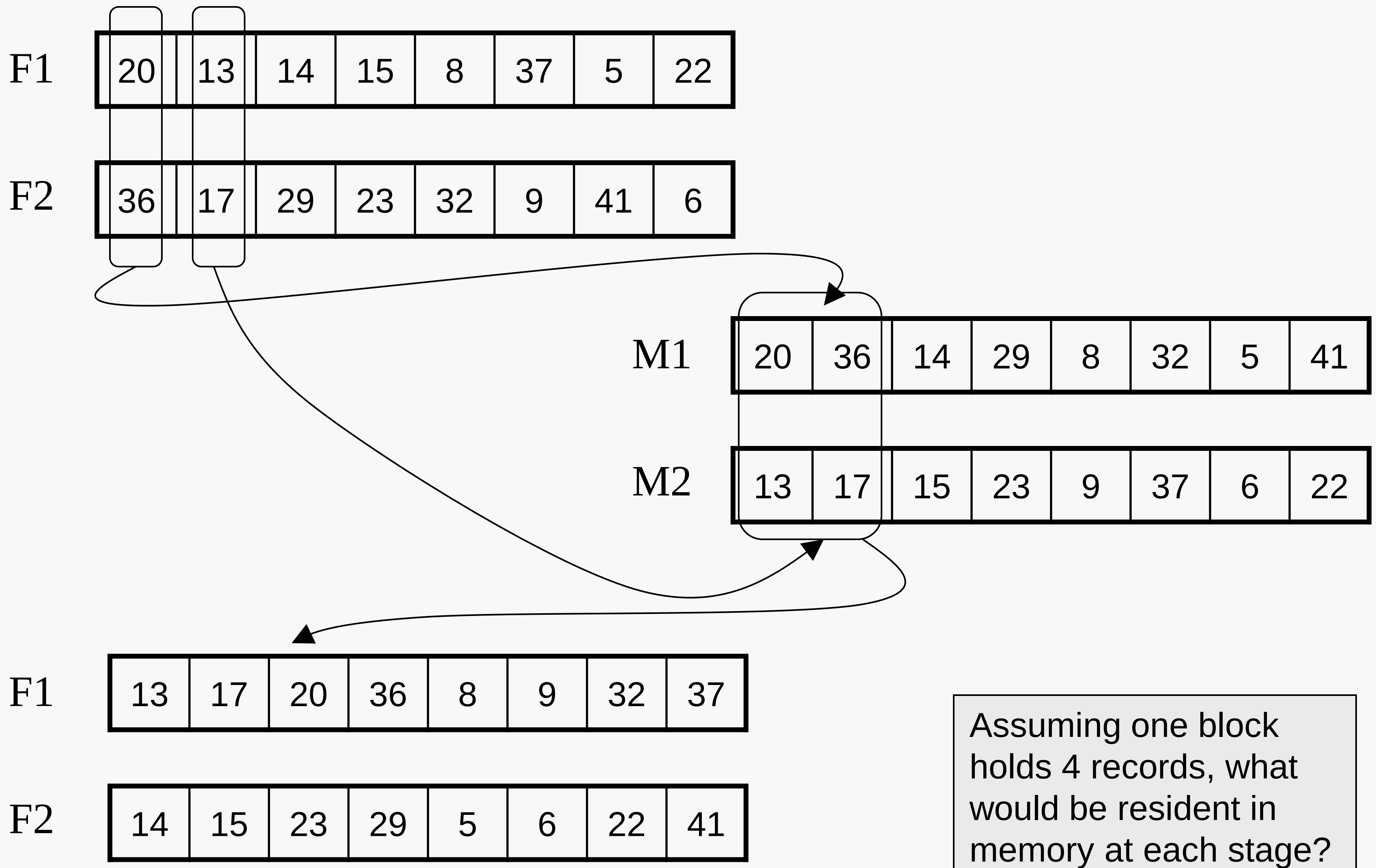
5 Repeat until F1 and F2 have been depleted, reading in new blocks as necessary.

M1 and M2 consist of ordered pairs of records.

6 Repeat steps 2-5, but merge pairs of records instead of single records, producing sorted runs of length four.

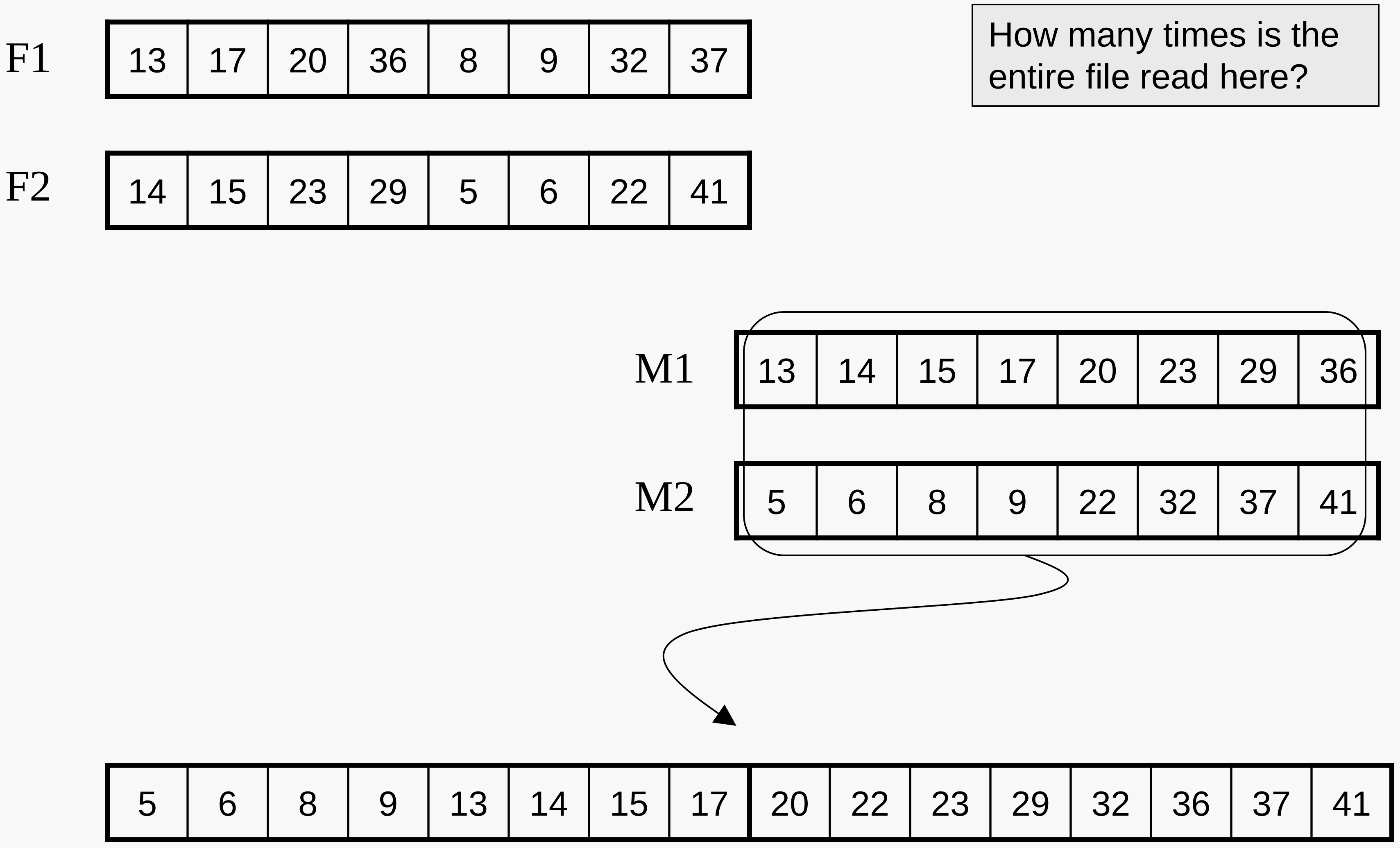
7 Continue, building longer and longer runs.

## Illustration



Assuming one block holds 4 records, what would be resident in memory at each stage?

## Illustration



How many times is the entire file read here?

## Reducing the Number of Passes

We can improve performance if we reduce the number of passes. The inefficiency lies in the early stages, as run lengths grow from 1 to 2 to 4 to 8 to …

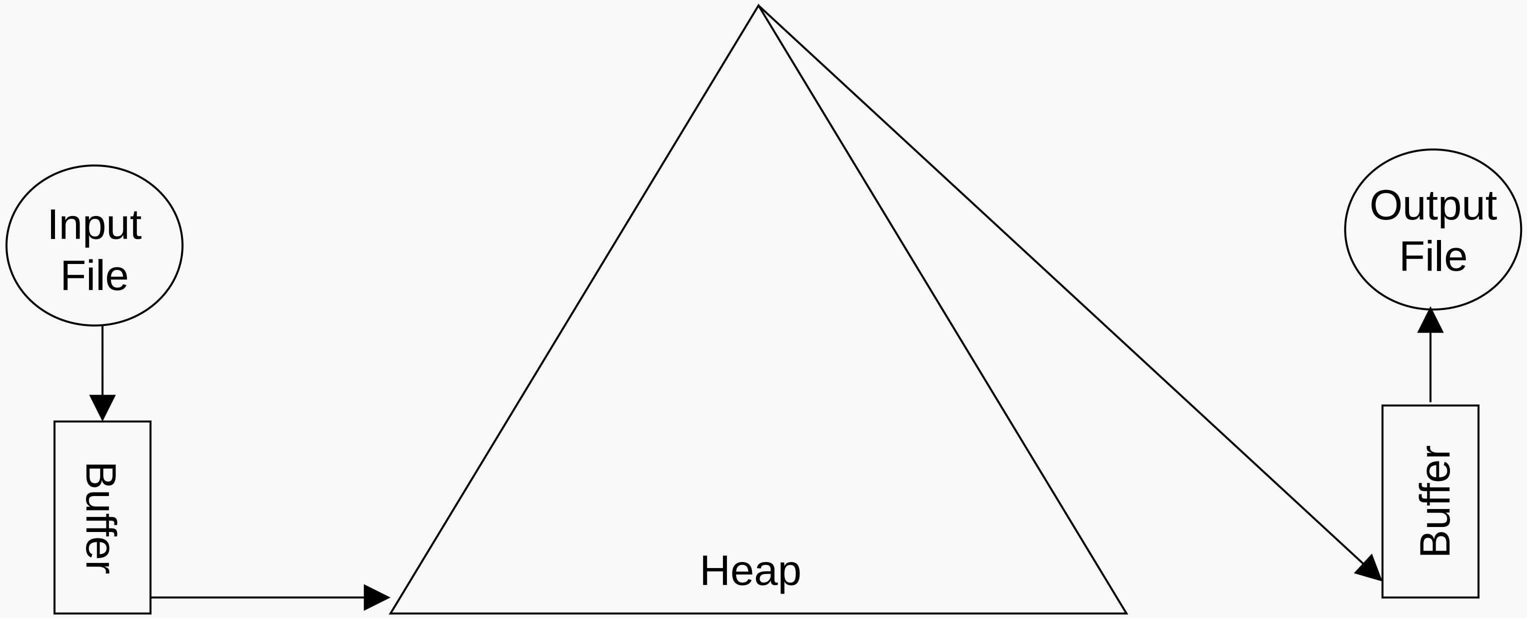
The problem is: how can we efficiently build long runs to which we may efficiently apply the merging algorithm?

One solution: read as much of the file as possible into memory at once, then sort that and write it to a new file. Repeat the process until the entire file has been read. This will build runs whose length corresponds to the amount of memory we can allot to hold records.

Surprisingly it's possible to do substantially better than that, producing initial runs that are roughly twice as long as memory will store at once…

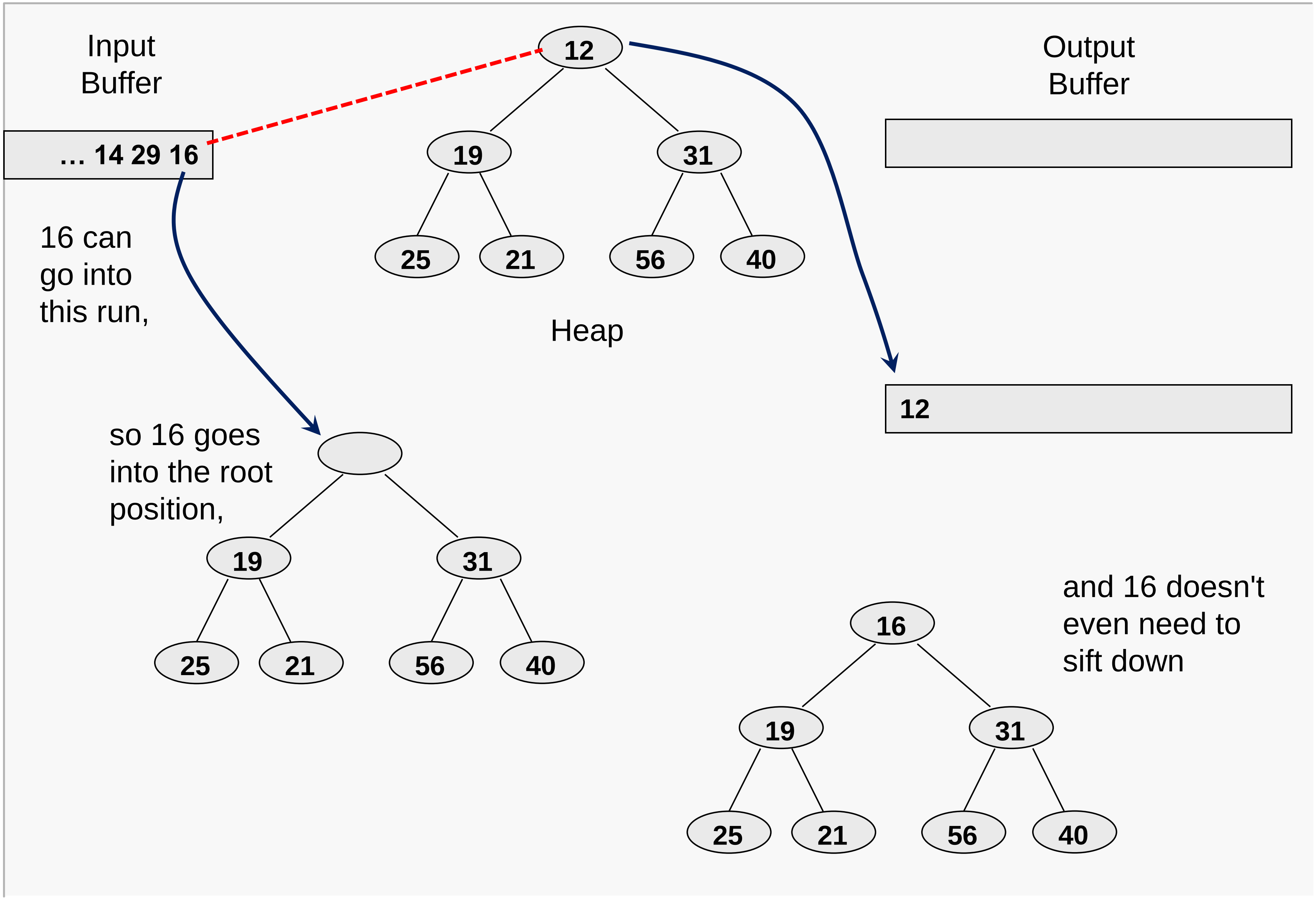
## Replacement Selection

Allocate a portion of memory for a min-heap, an input buffer and an output buffer:



Fill the heap, heapify, and delete the root to the output buffer. If the next key in the input buffer is larger than the old root, make it the new root, else…

## Illustration

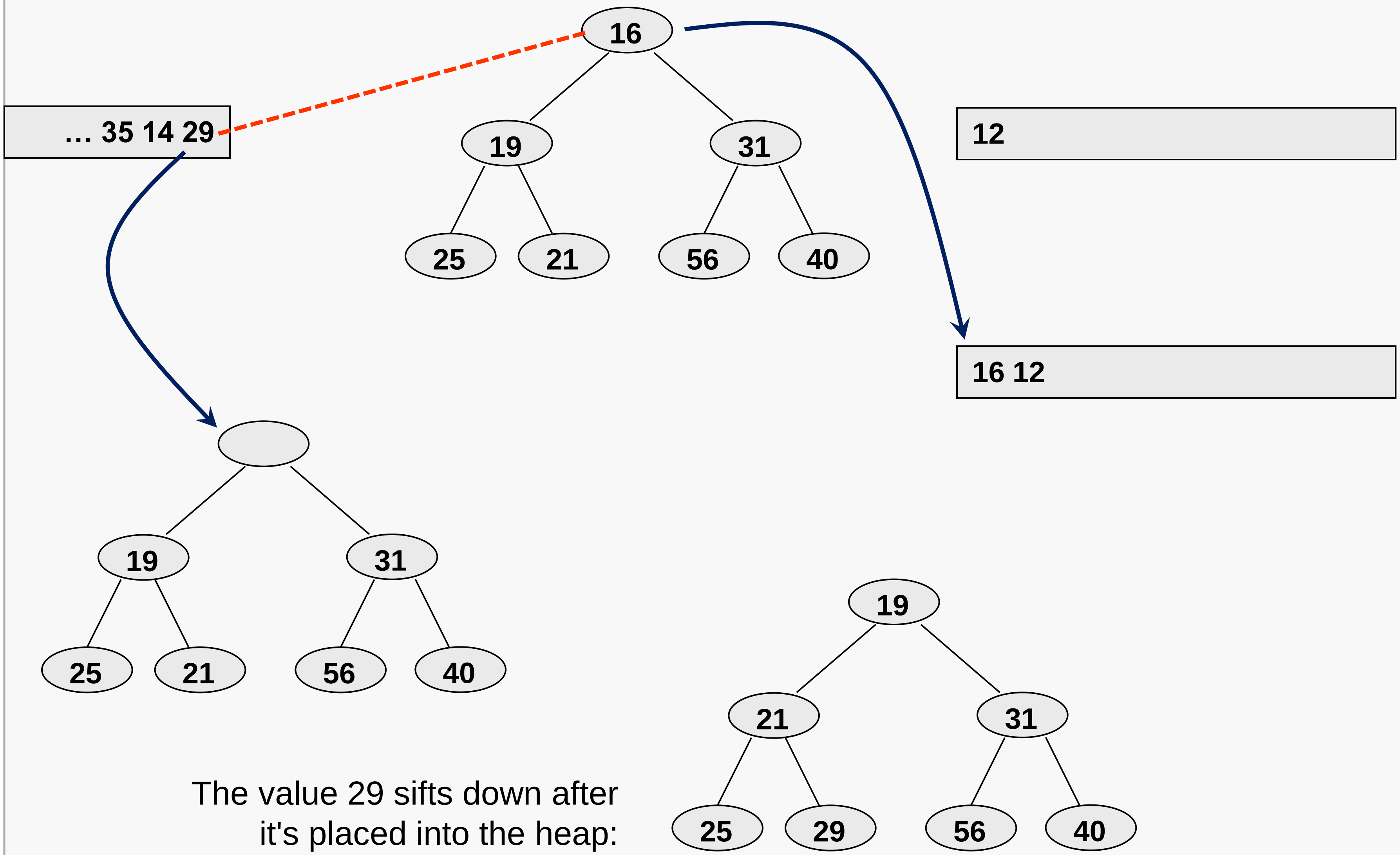


16 can go into this run,

so 16 goes into the root position,

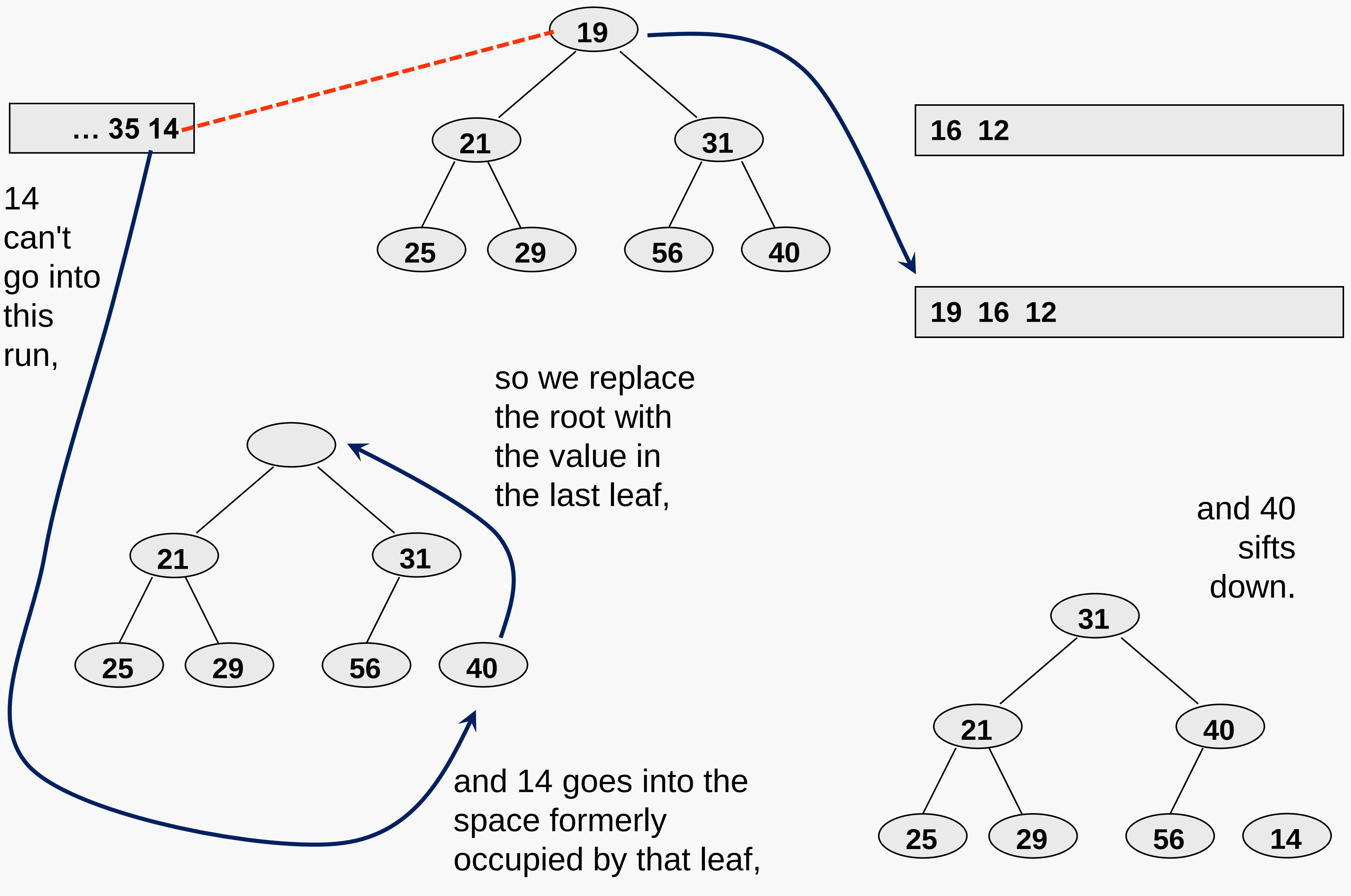
and 16 doesn't even need to sift down

## Illustration



The value 29 sifts down after it's placed into the heap:

## Illustration



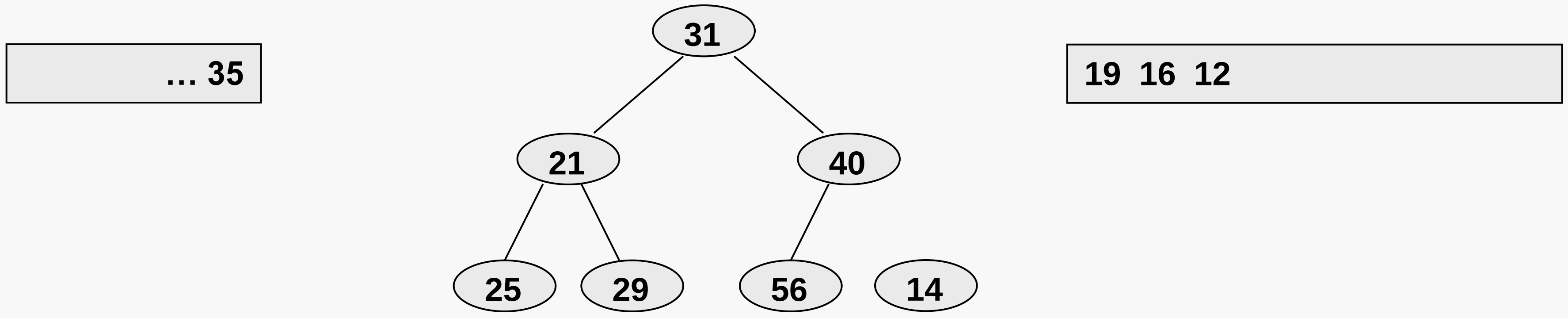
14 can't go into this run,

so we replace the root with the value in the last leaf,

and 14 goes into the space formerly occupied by that leaf,

and 40 sifts down.

# Illustration



… and so forth … when the heap becomes empty, we just flush the buffer out to the file, heapify the array (which is where?) and begin a new run…

… if we shrink the heap 50% of the time, we'll build a run that's got twice the number of elements the heap can hold.