Chapter 8 Sorting

Sorting Large Amounts of Data in Memory

If the entire file fits:

1. Read the entire file from disk into memory
2. Sort the records using a standard sorting procedure, such as shell-sort.
3. Write the file back to disk.

This is faster than sorting the file on disk because each record is read from disk only once and written to disk only once.

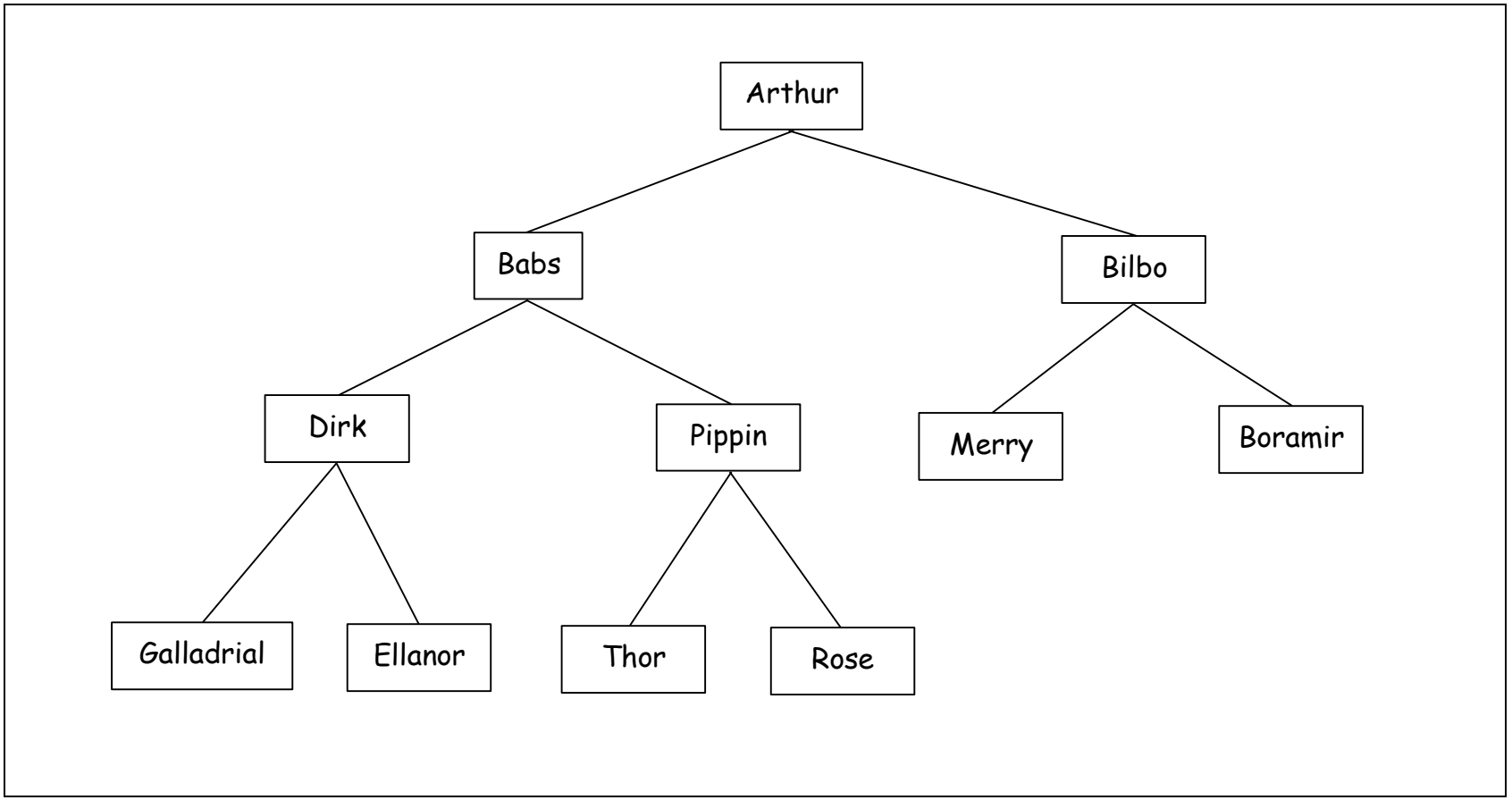
If we can overlap some of the reading with some of the writing, or start sorting before all the data is read, we can speed up the sort.

Heap-sort can start sorting before all of the data is read in.

All the keys are kept in a *Heap,* a binary tree with the following properties:

1. Each node has a single key, and that key is greater than or equal to the key at its parent node.
2. It is a *complete* binary tree
    a. all leaves are on no more than two levels
    b. all keys on the lower level are in the leftmost position
3. Because of 1 and 2, storage of the tree can be allocated sequentially as an array so that
    a. the root node has index 1
    b. the left child of the node with index *j* is at 2 * *j*
    c. the right child of the node with index *j* is at 2 * *j* + 1
    d. conversely, the index of the parent of node *j* is at $\lfloor j/2 \rfloor$

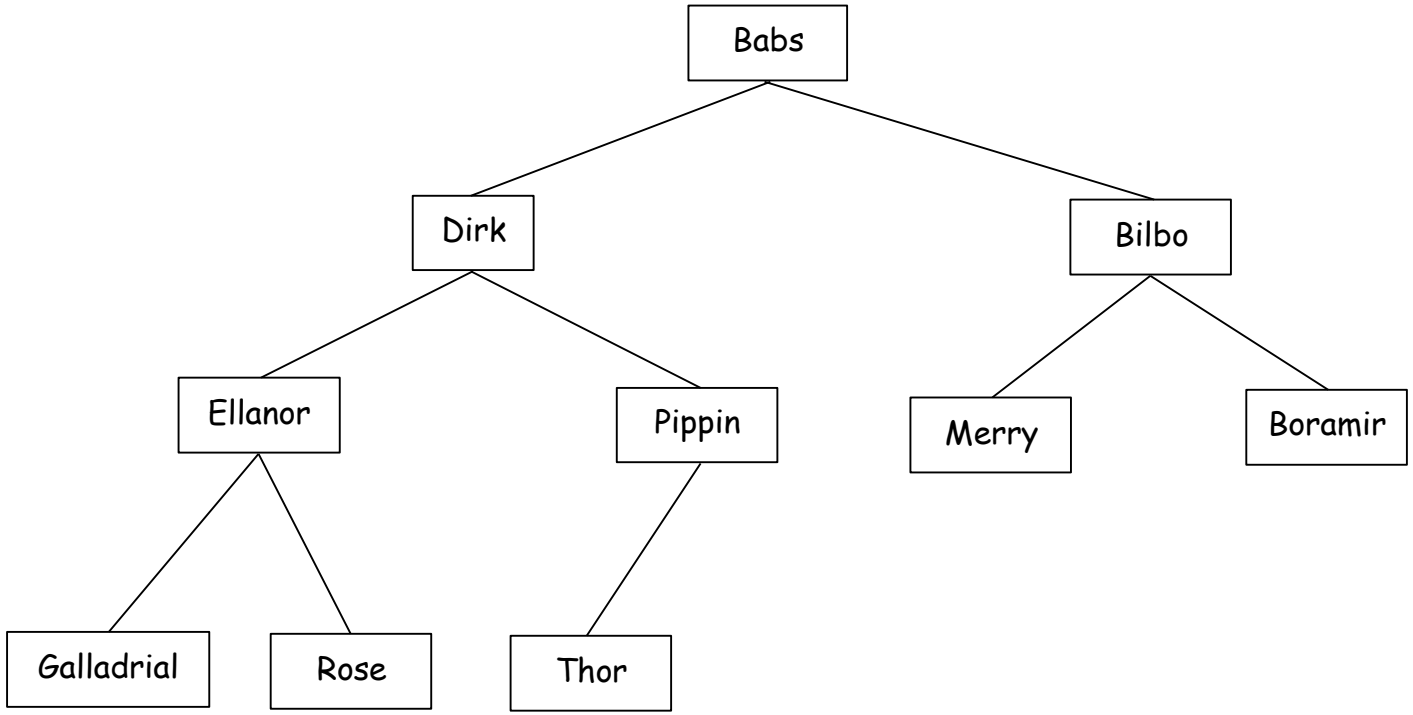
You should be familiar with heap sort from data structures class.



0	1	2	3	4	5	6	7	8	9	10
Arthur	Babs	Bilbo	Boramir	Dirk	Ellanor	Galladrial	Merry	Pippin	Rose	Thor

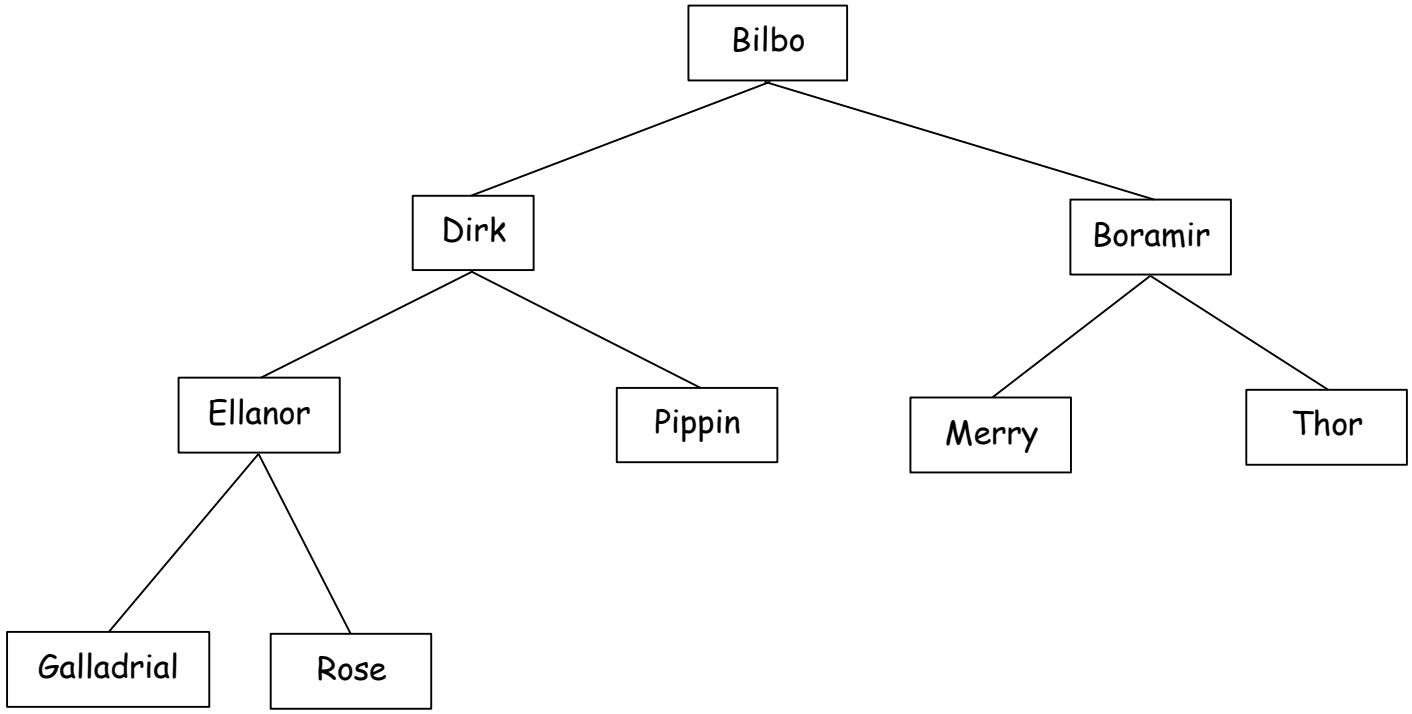
In Alpha Order

Output Athur and heapify



0	1	2	3	4	5	6	7	8	9	10
Babs	Dirk	Bilbo	Ellanor	Pippin	Merry	Boramir	Galladrial	Rose	Thor	

Output Babs and heapify

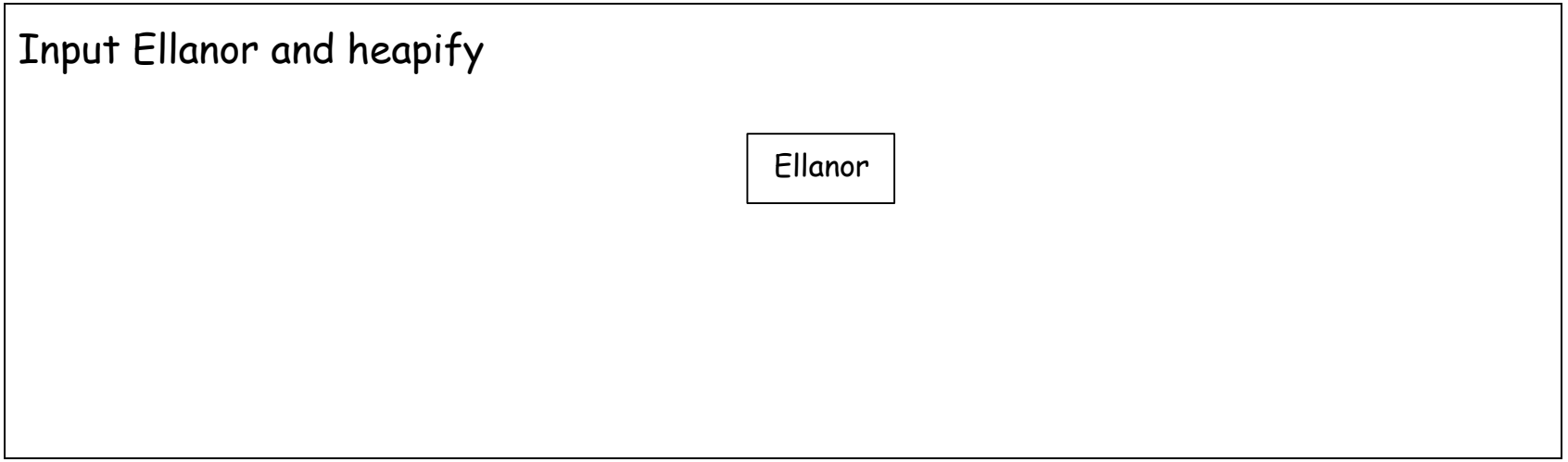


0	1	2	3	4	5	6	7	8	9	10
Bilbo	Dirk	Boramir	Ellanor	Pippin	Merry	Thor	Galladrial	Rose		

Continue this process until the list is output in order.

For this to work we need to build the original heap as we read in the file.

Input Ellanor and heapify

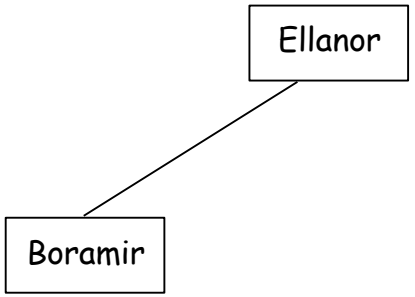


0	1	2	3	4	5	6	7	8	9	10
Ellanor										

Input Stream

Boramir	Bilbo	Galladrial	Pippin	Merry	Babs	Dirk	Arthur	Thor	Rose	

Input Boramir and heapify

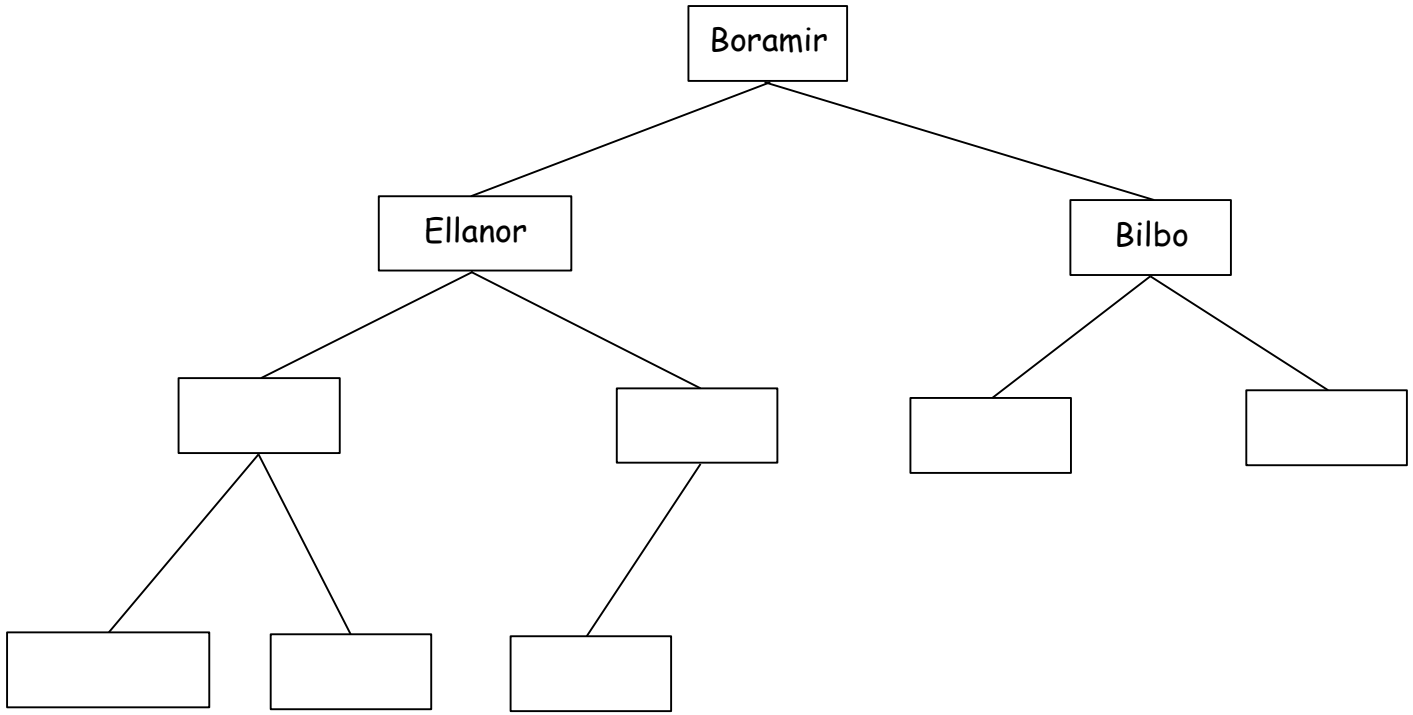


0	1	2	3	4	5	6	7	8	9	10
Ellanor	Boramir									

Input Stream

Bilbo	Galladrial	Pippin	Merry	Babs	Dirk	Arthur	Thor	Rose		

Lets finish this



Input Stream

Galladrial	Pippin	Merry	Babs	Dirk	Arthur	Thor	Rose			

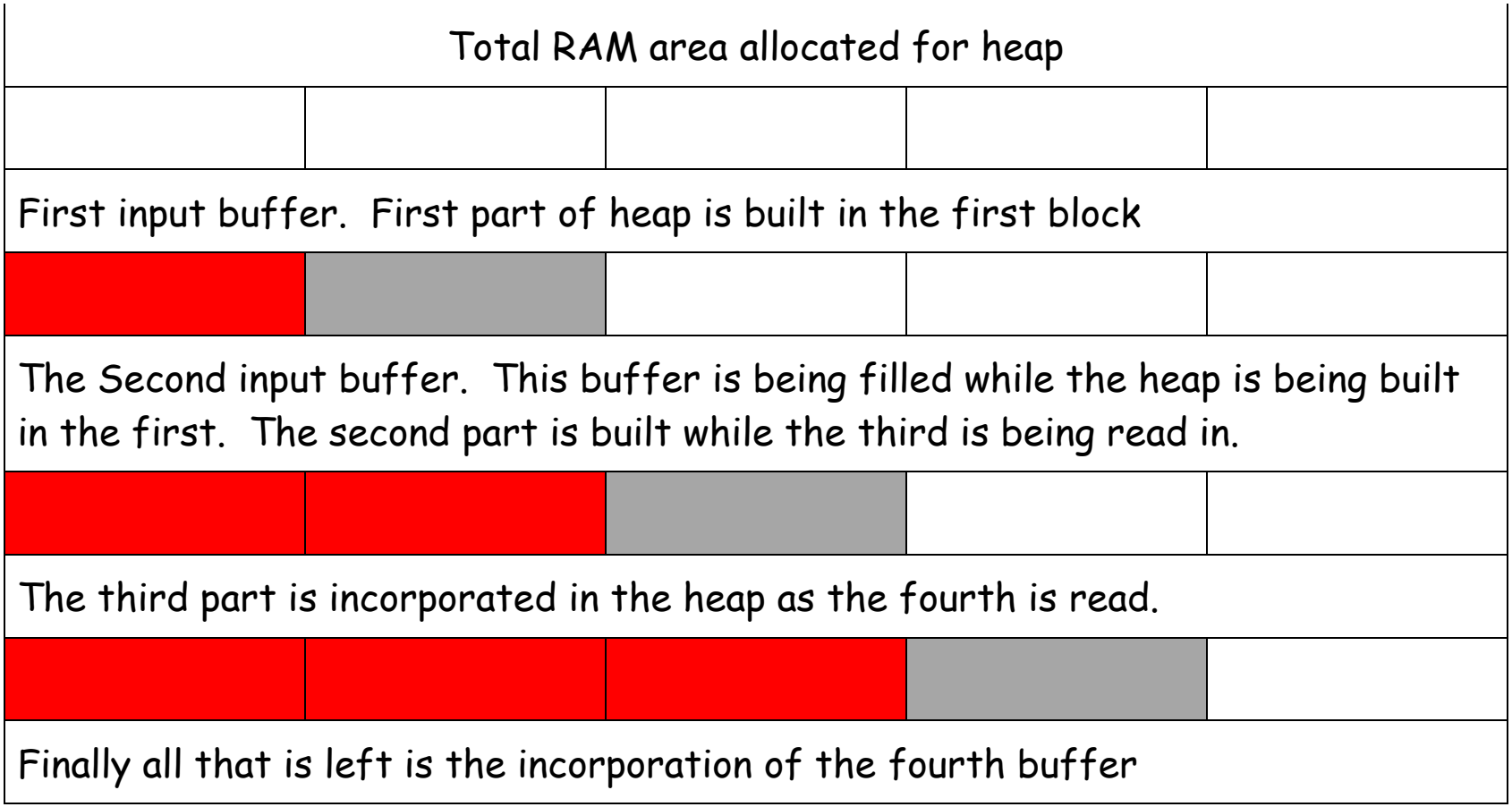
How do we make the input overlap with the heap-building procedure?

Read a block of records into the input buffer, then heapify that block

And at the same time, start reading the next block into a buffer just beyond the last record in this block.

Now the next record is exactly where it needs to be.

The only thing left after reading the last block is its incorporation into the heap.


Total RAM area allocated for heap
First input buffer. First part of heap is built in the first block
The Second input buffer. This buffer is being filled while the heap is being built in the first. The second part is built while the third is being read in.
The third part is incorporated in the heap as the fourth is read.
Finally all that is left is the incorporation of the fourth buffer

Now that we have the heap, we can write it out in sorted order.

Repeat these steps until all have been output:

1. Determine the value of the key in the first position of the heap.
2. Move the largest value in the heap into the first position, decrease the number of elements in the heap,
3. Reorder the heap by exchanging the largest element with the smaller of its children and moving it down the tree until it is either a leaf or larger than both its children.

As soon as we have identified a block of records that can be written, we start writing them.

Note that all I/O is sequential.

Next: What do we do if the file doesn't fit in memory?

Using Merge to sort files that don't fit in memory

Keysort

Once the keys are sorted we can use the index to read each record in and then write it out. If we do this in the key order we will have the data records in key order in the file.

Now the size of the file can be larger but is still limited by how many key records will fit in memory

So really large files are still a problem

We need another approach

8,000,000 records 100 bytes/record 10 bytes/key

The file is 800 megabytes.

We might have 10 megabytes of memory available for buffers.

All the keys require 80 megabytes so we can't sort all the keys in memory.

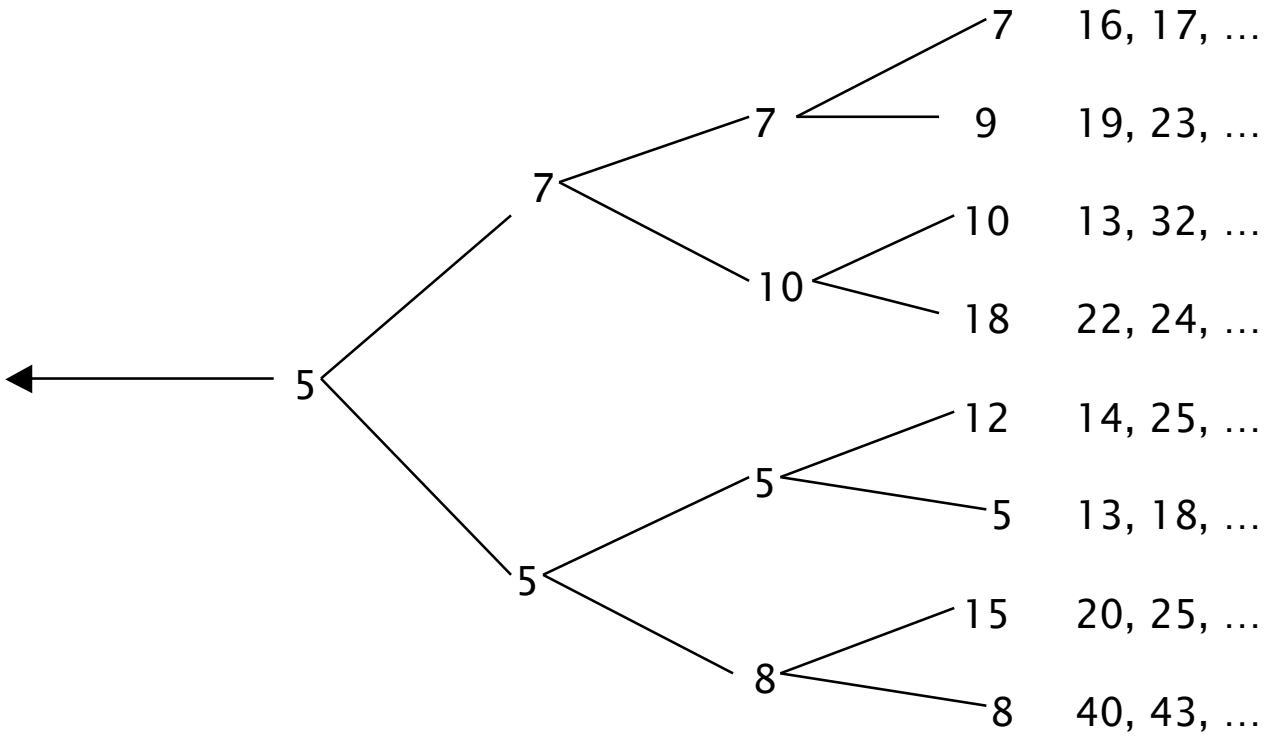
So we can use the following technique:

1. Create sorted sublists using heapsort
2. Merge the sublists using k-way merge of section 8.3

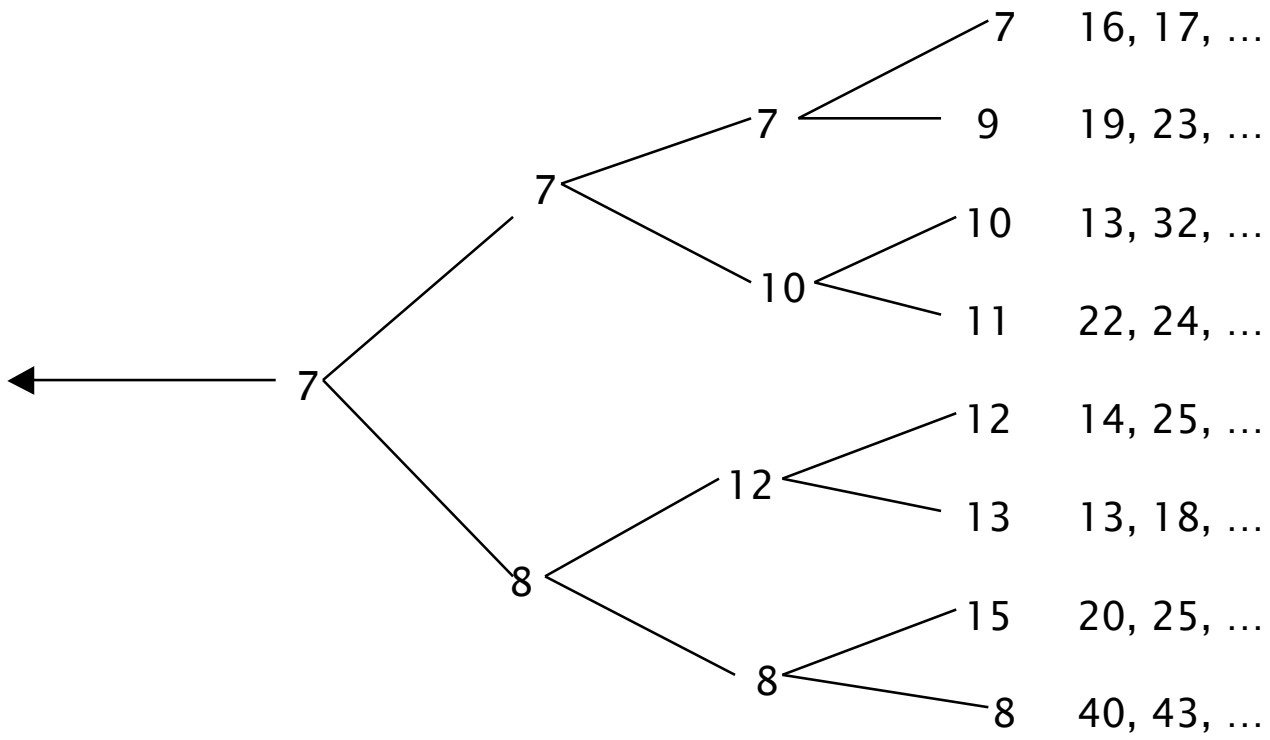
In our example we have: $\frac{10{,}000{,}000 \text{ bytes of memory}}{100 \text{ bytes per record}} = 100{,}000 \text{ records}$

So, we would have 80 runs each containing 100,000 records and our merge would be an 80-way merge.

Merge Tree:



Send the 5 to the sorted list, remove the 5 and rebuild the tree.



Now you can send the 7 to the sorted list, remove the 7 and rebuild the list.

This solution has the following features.

1. Merge can be extended to sort files of any size.
2. Reading during the creation of the runs is sequential so it is faster than keysort which requires a seek for each record
3. Reading the runs during the merge is also sequential although switching from run to run requires a seek
4. Heapsort can be used to overlap reading and sorting
5. Tournament Sort to avoid k = # of runs comparisons at each round of the sort.

How much time does this take? There are 4 steps:

1. Reading Records into Memory for Sorting and Forming Runs

   80 – 10 megabyte chunks

   access time per block = seek time + rotational delay

   = 8 msec + 3 msec = 11 msec

   transfer rate = 14,500 bytes/msec

   So for our example we have 80 seeks and 800 megabytes to transfer

Access:	80 seeks * 11 msec	=	1 sec
Transfer:	800 megabytes @ 14,500 bytes/msec	=	60 sec
Total:			61 sec

2. Writing Sorted Runs to Disk

   This is the reverse of reading above so is 61 sec

3. Reading Sorted Runs into Memory for Merging

   80 runs * 80 seeks/run = 6400 seeks

   6400 seeks * 11 msec/seek = 70 seconds for access time

   We still are transferring 800 megabytes so transfer time is 60 seconds

   This gives 2 minutes 10 seconds

4. Writing the Sorted File to Disk

   Assume 200,000 byte output buffers

$$\frac{800{,}000{,}000 \text{ bytes}}{200{,}000 \text{ bytes per seek}} = 4000 \text{ seeks}$$

   So, the total seek and rotational time is 4000 * 11 msec = 44 seconds

   Transfer time is still 60 seconds to the total is 1 minute 44 seconds

The total is then 356 seconds or almost 6 minutes.

What happens if we have a file that is 10 times larger?

- The sort phase uses sequential I/O so this phase uses minimum time
- So we need to work on the *Merge Phase* to speed up our algorithm
- In fact we work on the *read step* of the merge phase
- Cut down on the number of random access reads during the merge phase

80,000,000 records

8,000 megabytes

800 way merge

800 runs * 800 seeks/run = 640,000 seeks

	Number of seeks	Amount transferred (megabytes)	Seek + Rotation time	Transfer Time (seconds)	Total Time (seconds)
Merge Reading	640,000	8,000	7,040	600	7,640
Merge Writing	40,000	8,000	440	600	1,040
Totals	680,000	16,000	7,480	1,200	8,680

In general for a K-way merge of K runs the buffer size is

$$\left(\frac{1}{k}\right) \times \text{size of memory} = \left(\frac{1}{k}\right) \times \text{size of each run}$$

There are K runs (i.e. K times through the entire file)

Each of the K runs requires K accesses.

So, sort merge is $O(K^2)$ seeks and since $K = rN$ for some r the merge is $O(N^2)$

So What Do We Do?

- Hardware Based Solutions
- Software Based Solutions
  - work on merge
  - work on length of runs
  - overlap I/O operations

HARDWARE BASED SOLUTIONS

- Increase Memory – fewer runs and fewer seeks/run
  - 80,000,000 record file with 10 megabytes of memory requires 2 hrs 6 min.
  - 40 megabytes 400,000 records/run means 200 runs
  - 4000 seeks resulting in 16 minutes 40 seconds for the sort.

- Increase the Number of Disk Drives
  - This reduces the movement of the disk between runs
  - 800 drives means 800 seeks as opposed to 640,000 seeks
  - 7040 seconds becomes 1 second

- Increase the number of I/O channels
  - give each drive its own I/O channel
  - Complete overlap of I/O