Merging II

We saw *merging* as an example of a *cosequential processing* algorithm.

We have seen ways to sort *big* files, but we still don't know how to sort *really big* files.

When a file is too big to fit into RAM, and even just the keys don't fit in RAM (or if *keysort* is inappropriate), we need to split the files into smaller files, sort the smaller files, and merge them back together.

Topic	***Folk & Zoellick***
• Mergesort	§ 7.5
• Multiway Merging	§ 7.3.1 (sort of)
• Selection Trees	§ 7.3.2
• Replacement Selection	§ 7.5.6
• A New Heapsort	§ N/A

Run

The smaller files that we split a big file into and sort to be merged later

Sorting Big (big) Files

Sorting files direct from/to disk is disastrous.

If we have enough RAM to hold the whole file, sort it there:

1. `read the entire file into RAM`
2. `do your favourite internal sort`
3. `write the sorted data out to disk`

The *heapsort* is a particularly space-wise algorithm for this sort.

If the file is too big to fit into RAM, maybe we only need to sort the keys:

1. read each record r_i, storing only (k_i, RRN_i) or (k_i, $offset_i$)
2. do your favourite internal sort
3. using the sorted (k_i, RRN_i) or (k_i, $offset_i$) seek to each record in the original file and write it out (or just write out the sorted keys)

But what if not even the keys fit in RAM?

Sorting Bigger Big (big) Files

For the biggest big files, there is no choice but to split the file into smaller files (*runs*) and sort them individually. Then merge them back into one *honkin' big* sorted file.

```
1. As long as there are records left in the file
   a. read as many records as possible into RAM
   b. do a heapsort on those records
   c. write the sorted run out to a temporary file

2. merge the sorted runs
```

If our file has N records and is split into K runs (having about M records each)...

Q: What is the complexity for:

- reading the records:
- sorting the runs:
- writing the runs:
- merging the runs:

MultiMerge

We've already seen how to merge two files (called a two-way merge). Here's the algorithm, slightly adjusted to allow duplicate records to be written.

```
1. Read record $r_1$ from file $F_1$ and record $r_2$ from file $F_2$
2. While records remain in either file
      if $k_1 \leq k_2$
         output $r_1$; read new $r_1$ from $F_1$
      else
         output $r_2$; read new $r_2$ from $F_2$
```

□

The K-way merge is just as simple. Here it is (again, writing as many duplicates as appear in the original runs).

```
1. For i from 1 to K
      Read record $r_i$ from file $F_i$
2. While records remain in any file
      find i such that $k_i$ is the minimum of $k_1$ to $k_K$
      output $r_i$; read new $r_i$ from $F_i$
```

□

The Complexity of MultiMerge

The complexity of two-way merge, as we’ve already seen, is O(M + N).

So what’s the complexity of the K-way merge?

- reading the first record from each file: O(K)
- the main loop: $O(\Sigma M_i) = O(N)$
  - finding the minimum key: O(K)

- total: $O(K) + O(K \times N)$

□

Q: If K << N, $O(K\times N) = O(N)$. But what is the *real* meaning of $O(K\times N)$?

What Happened to O(N) Merging?

As it turns out, K is actually linear with N...
Oh no!

Let’s say we have an illustrated encyclopedia represented as a file of records, one for each article in the encyclopedia. Since the encyclopedia is illustrated, we have pictures (jpegs) in the records:

- 100,000 articles (records)
- 100,000 bytes per record

If we have 100MB RAM, we can fit 1,000 records in RAM at a time. To sort the file would require 100 runs, each containing 1,000 records.

Notice that if we double the number of articles to 200,000 we double the number of runs to 200.

K may be a thousand times smaller than N, but it’s still O(N)

So in the K-way merge, finding the minimum key in K records is O(N). So the K-Way merge is actually $O(N^2)$!

:-b

Merging is Stupid

In the K-way merge, the main loop is executed N times. Every time through the loop, we compare K keys, *even though K-1 of them are the same keys as the last time through the loop.*

We can do better. Here's a new K-way merge algorithm:

```
1. For i from 1 to K
      Read record $r_i$ from file $F_i$ into list[i]
2. Build a binary tree from list[i] such that:
    o the leaves of the tree contain ($k_i$,i)
    o the parent of nodes ($k_i$,i) and ($k_j$,j) contains
         ($k_i$,i) if $k_i \le k_j$
         ($k_j$,j) otherwise
3. While records remain in any file
      output $r_i$ where ($k_i$,i) is the root of the tree
      read new $r_i$ from $F_i$ into list[i]
      update the tree to maintain its order
```

Q: What is the complexity of the *new and improved* K-way merge?

A:

Selection Tree Example

The tree that we build and maintain in the K-way merge is called a *selection tree*. Let’s look at an exmample of merging files with a selection tree. Conveniently, we have 8 runs:

1	2	3	4	5	6	7	8
...	...	...	...	...	...	...	...
22	28	37	29	26	30	35	34
15	24	18	27	19	13	25	21
12	14	16	23	17	10	20	11

1	2	3	4	5	6	7	8
...	...	...	...	...		...	...
22	28	37	29	26	...	35	34
15	24	18	27	19	30	25	21
12	14	16	23	17	13	20	11

We’ve Got Bigger Problems

We’re not out of the woods yet, Dorothy. Let’s see another example.

Let’s say you have access to the CSIS mainframe. (You didn’t hack into it, you got a co-op job there). After two months of reading documentation, you get an email from your supervisor, who is just getting back from vacation. You have to write a program to sort their file of Canadian population records. There are:

- 30 million people (records)
- 100,000 bytes per record (each record has a jpeg of fingerprints)

The 486 they let you work on has 100MB RAM, so you can fit 1,000 records in RAM. To sort the file will require 30,000 runs.

No problem, you say? Learned all about it in CSI 2131, did you?

Don’t forget: the K-way merge must have one record from each run in RAM at the same time.

Q: How much RAM is required to merge the runs?

A:

Running Out of RAM

Somehow we’ve got to decrease the number of runs.

One way to do it would be to take some subset of J runs, merge those runs, then merge the merged runs.

In our CSIS example, we could fit 1,000 records into RAM at a time. That means that we can merge 1,000 runs at a time. We could merge 30 batches of 1,000 runs each, then merge the 30 bigger runs into one big file.

Multi-Step Merging
Merging smaller batches of runs into bigger runs, then merging bigger batches of bigger runs into even bigger runs, until everything is merged into one big file.

Q: Disadvantages?

A:

Q: Advantages?

A:

Back at the Heapsort

Maybe if we were more careful with our use of RAM during the sorting phase, we could build *longer* runs (and therefore *fewer* runs).

Remember the heapsort?

Heap Builder

```
for i from 1 to n
   read the next record r having key k
   put record r at the end of the array

   while k < the key of r’s parent
      exchange record r with its parent
endfor
```

Heap Sorter

```
for i from 1 to n
   write the record in array[1]
   last = n - i
   move the record in array[last+1] to array[1]
      (and call its key k)

   while k > the keys of either of its children
      exchange record r with the child having
         the smaller key
endfor
```

The Heapssort

While heapsorting, a slot in the heap becomes free on each iteration. We *could* use the empty slot to buffer output, but we could also use it to increase the length of runs produced by each sort step.

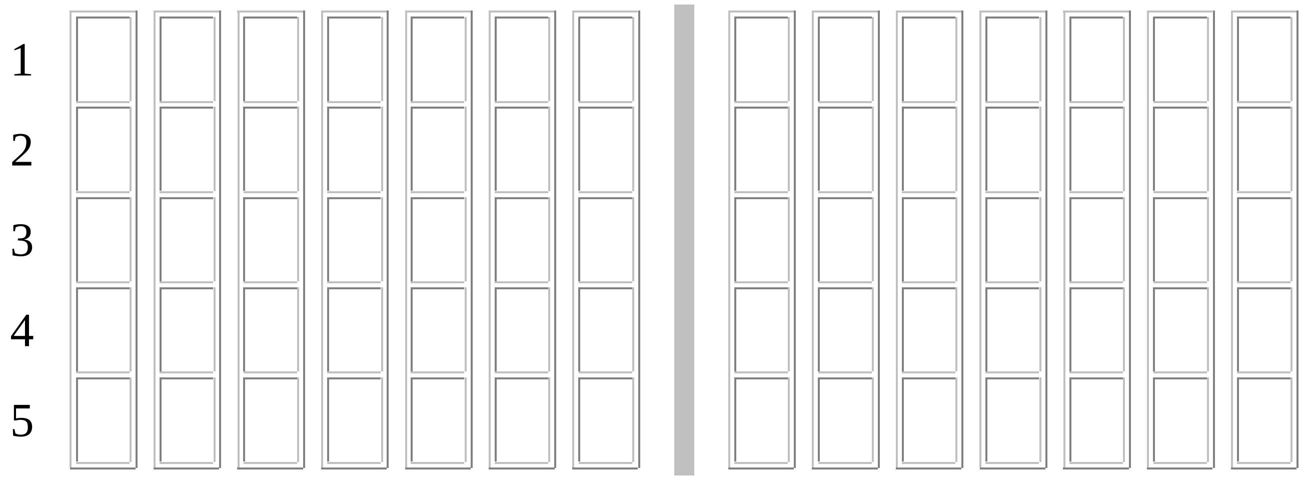
```
for i from 1 to n
   read the next record r having key k
   put record r at the end of the array

   while k < the key of r’s parent
      exchange record r with its parent
endfor
```

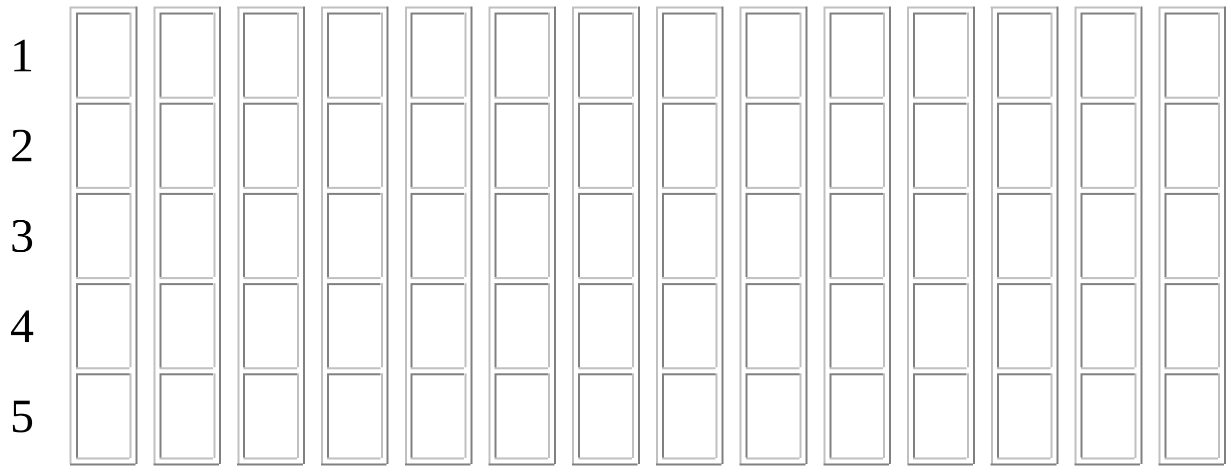
```
last = n
prevkey = lowvalue
while last > 0
   write the record in array[1]
   read the next record r having key k
   if k < prevkey
      move the record in array[last] to array[1]
      call its key k
      put record r into array[last]
      last = last - 1
   else
      put record r into array[1]

   while k > the keys of either of its children
      exchange record r with the child having
         the smaller key
endwhile

when done, start a new run
```

New Heapsort Example

Let’s run through the new heapsort with the following records (only their keys are shown). Unfortunately we only have room for 5 records in the heap.

```
19 5 31 25 18 22 21 24 11 29 15 27 10 6 14 ...
```





Replacement Selection

The nifty adjustment to the heapsort algorithm is called *replacement selection*.

Q: What's with the funny insertion order for records with keys less than the previous key written out?

A:

Q: If we used the original heapsort algorithm inside our mergesort, how many runs would there be (in the example)?

A:

Q: In the worst case, what do you think will be the run size (given a heap that can hold a fixed number *C* of records)?

A:

Q: In the best case, what will be the run size?

A:

Q: On average, what will be the run size?

A: