B+Tree (Bottom Up) Construction.

- Efficiently build a B+tree index for sorted data existing within a file.
- Data is read one “node” at a time into a buffer. These nodes become the “external” leaf nodes in the B+tree. These nodes may have some spare space for some future insertions.

B+Tree (Bottom Up) Construction II

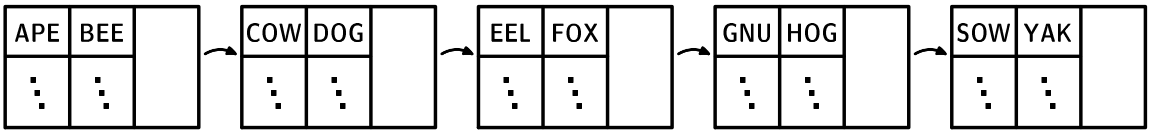
1. Read first data node. Create first internal (order m) root node. The address of data node is the root node's first pointer value.

2. Read next data node and put it's smallest key into the lowest level, not at capacity, internal node. If all internal nodes are at capacity, create a new root, put address of previous root in the first pointer with the key. If the current entry is above the parent-to-the-data level, create new internal nodes between the entry and the data level. The address of each node becomes its parent's first/next pointer value. When memory fills, flush any node except the current ancestors of the leaf node presently being processed. Continue step 2 until all data nodes have been read.

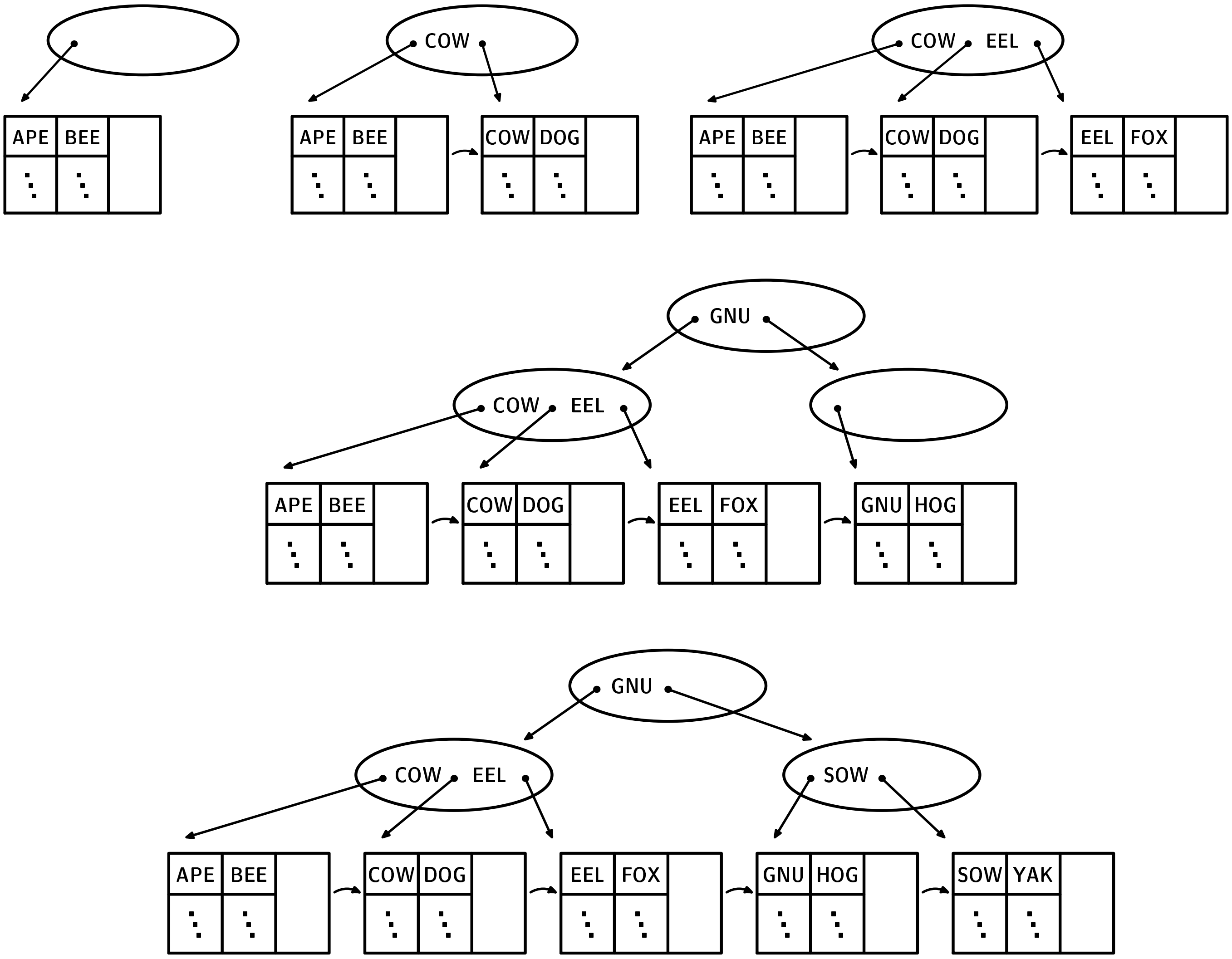
3. Clean Up. The rightmost internal nodes at some levels may be too empty. Start at bottom of tree and work up. (Consider only internal nodes.)

B+Tree (Bottom Up) Construction III

File contains 10 records, accessed as 5 nodes each containing 2 records and one spare slot.



sequence of order 3 B+trees



External Sorting I

Two phases – distribution & merge.

Distribution: input file is read and sorted into *sequences* as long as possible. The sequences are each written to output files.

Merge: read some number of sequences, combining (merging) them into a single longer sequence that is written to output files. Continue merging until one sequence contains all the data.

- Costs are primarily the time to read and write the data.

Several approaches:

tape drive sorting motivates many.

Distribution: heapsort or replacement selection.

Merge: P-way.

HeapSort I

0. Allocate two buffers (input and output).

1. Read (fill) the input buffer. Heap Order the input buffer. Fetch data from input buffer and put into output buffer until the input buffer is empty. Write (empty) the output buffer. Continue step 1 until all input data has been read.

Heap Sort II

	input					output				
input	10	21	15	5	6					
heap	5	6	15	21	10					
move		6	15	21	10	5				
sift	6	10	15	21		5				
move		10	15	21		5	6			
sift	10	21	15			5	6			
move		21	15			5	6	10		
sift	15	21				5	6	10		
move		21				5	6	10	15	

. . .

HeapSort III

Run Time –

Buffers capacity: m data items

Input data: n data items

creates n/m sequences.

Access data from a file sequentially, each data item requires $t_{\text{sequential}}$ ms. The random access time is t_{random} ms. Read or write data in $t_{\text{sequential}}\ n + t_{\text{random}}\ n/m$ ms.

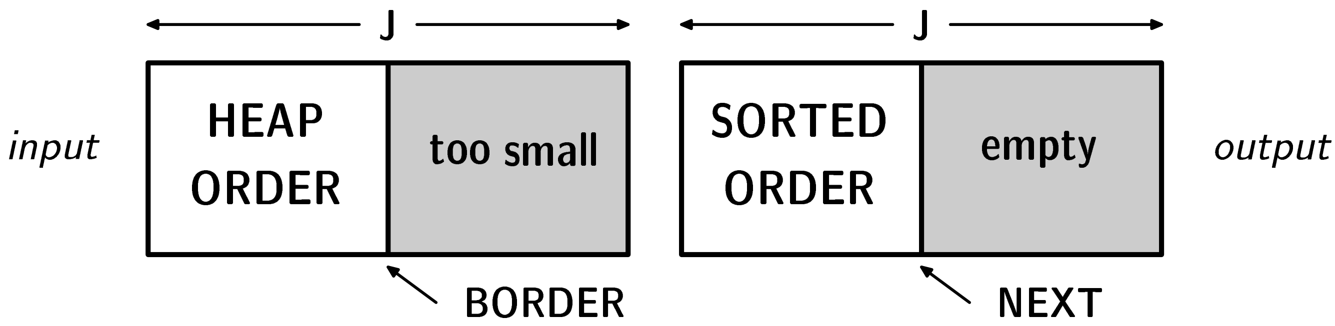
Internal comparing, fetching, & swapping data items requires t_{internal} ms.

Each input buffer requires proportional to $m \log(m)$ operations. The total internal time will be

$$t_{\text{internal}}\ n \log(m) \text{ ms.}$$

Replacement Selection Distribution

File F contains n records. Create k segments of $J = n/k$ records. Two buffers, both contain J records.



1. Read J records sequentially from F into input buffer.
2. Heap Order these records.
3. Move smallest record from heap order portion of input buffer to output buffer. *Sift* record R just to the left of BORDER, if any. Read another record $R1$ from F into the slot just vacated to the left of BORDER. If $R1$ no less than R, *insert* $R1$ into Heap, otherwise include in the too small pool. If nothing exists to the left of BORDER, output buffer, close file, open next segment file, and continue with step 2 on the too small records.

Replacement Selection Distribution II

	input					output				
input	10	21	15	5	6					
heap	5	6	15	21	10					
move		6	15	21	10	5				
sift	6	10	15	21		5				
read	6	10	15	21	8	5				
insert	6	8	15	21	10	5				
move		8	15	21	10	5	6			
sift	8	10	15	21		5	6			
read	8	10	15	21	3	5	6			

$\ddots$

Replacement Selection Distribution III

insert New record in "last" position; others already in heap order. Move into heap order by swapping with parents. (a priority queue insert operation)

sift Record from "last" heap order position moved to root and then swapped with smaller child until in heap order again; (a priority queue deletion operation)

Theorem: (Knuth/Moore) With randomly ordered input, replacement selection distribution creates initial segments with average size twice the input buffer size.

"Snowplow argument."

P-way Merge

- P segments combined into one longer segment.
- Buffer space limits P.

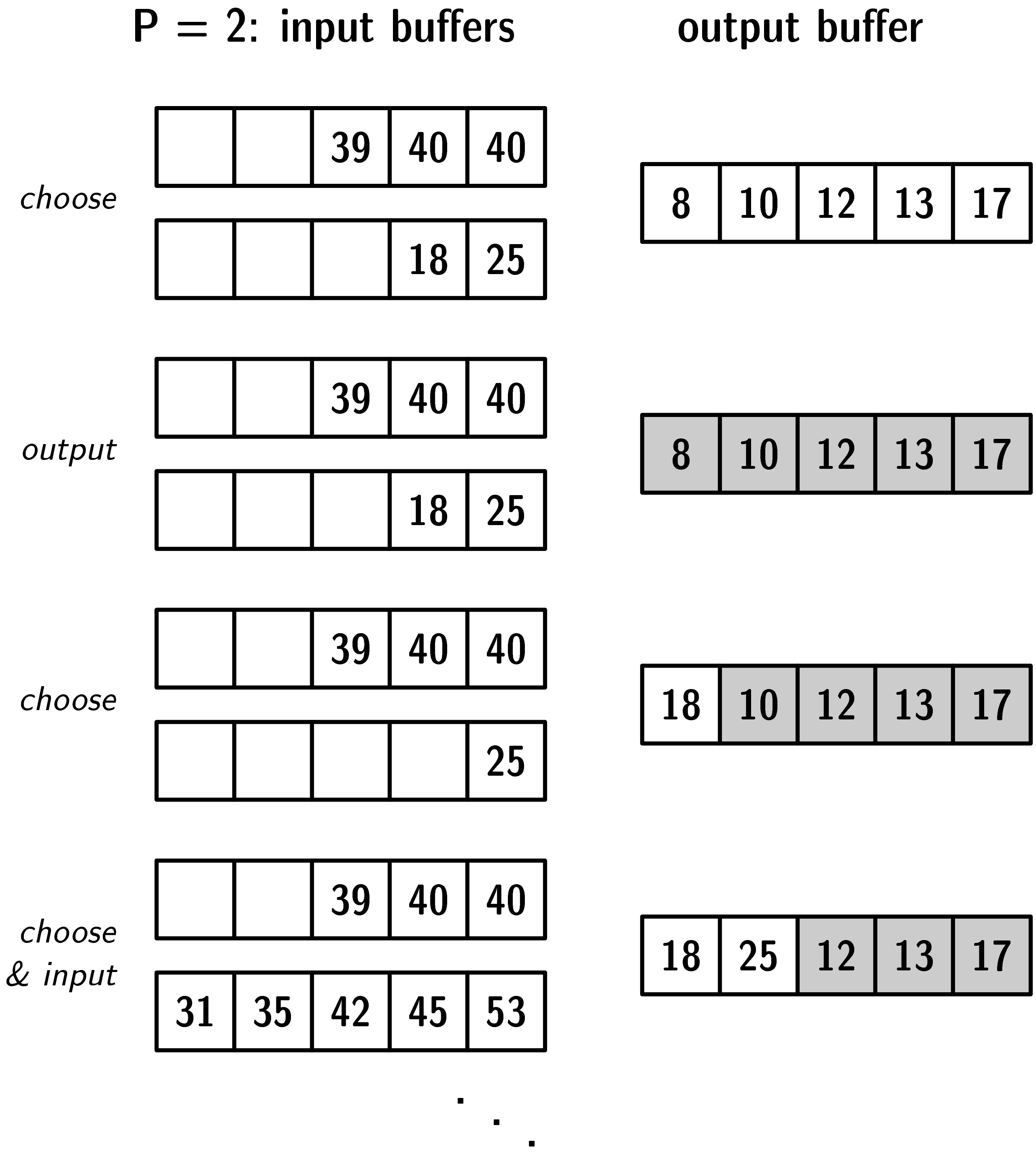
  Space divided into P+1 equal-sized buffers (J records each.)

1. For each of the $i \leq$ P segments, read J records sequentially into the associated buffer.
2. Move smallest remaining record in i buffers to output buffer.
3. If a buffer empties, read another J records from segment associated with the empty buffer. If the segment is empty, then decrement i.
4. If i is zero, write output buffer and terminate. If the output buffer is full, write it. Continue with step 2.

P-way Merge II

	P = 2: input buffers					output buffer				
input	10	17	39	40	40					
	8	12	13	18	25					
choose	10	17	39	40	40	8				
		12	13	18	25					
choose		17	39	40	40	8	10			
		12	13	18	25					
choose		17	39	40	40	8	10	12		
			13	18	25					
choose		17	39	40	40	8	10	12	13	
				18	25					

P-way Merge III



P-way Merge IV

initial segments: m

average number of records per segment: n

total buffer record capacity: k.

P-way merge time (j segments): $T_P(j)$

A single segment is read into buffer of size $k/(P{+}1)$; **each read will comprise a random access since we will be reading from several segments. The total number of reads for a segment is** $n/(k/(P{+}1)) = (P{+}1)*n/k$. **The records of each segment are eventually written back to disk; the same number of write operations is necessary as well. The other time for accessing the single segment is the record transfer time for both the read and write operation: there are** $2n$ **such transfers for a single segment.**

For P segments:

$$T_P(P) = 2P(P{+}1)*n/k \; t_{\text{random}} + 2Pn \; t_{\text{sequential}} \text{ ms.}$$

P-way Merge V

Merge all m segments in time $T_P(m)$ and create m/P longer segments to merge at the next "level".

The sorting process continues doing more P-way merges until there is exactly one segment remaining.

level	length	number
0	n	m
1	nP	m/P
2	nP^2	m/P^2
	$\ddots$	
$\log_P(m)$	nm	1

Time for level i: (same for all levels)

$$2(m/P^i)(P+1)*nP^i/k\ t_{\text{random}} + 2(m/P^i)nP^i\ t_{\text{sequential}} \text{ ms.}$$

$$= 2m(P+1)*n/k\ t_{\text{random}} + 2mn\ t_{\text{sequential}} \text{ ms.}$$

P-way Merge VI

The total time for the P-way merge operation will be

$$\log_P(m)\{\ 2m(P+1)n/k\ t_{\text{random}} + 2mn\ t_{\text{sequential}}\ \}$$

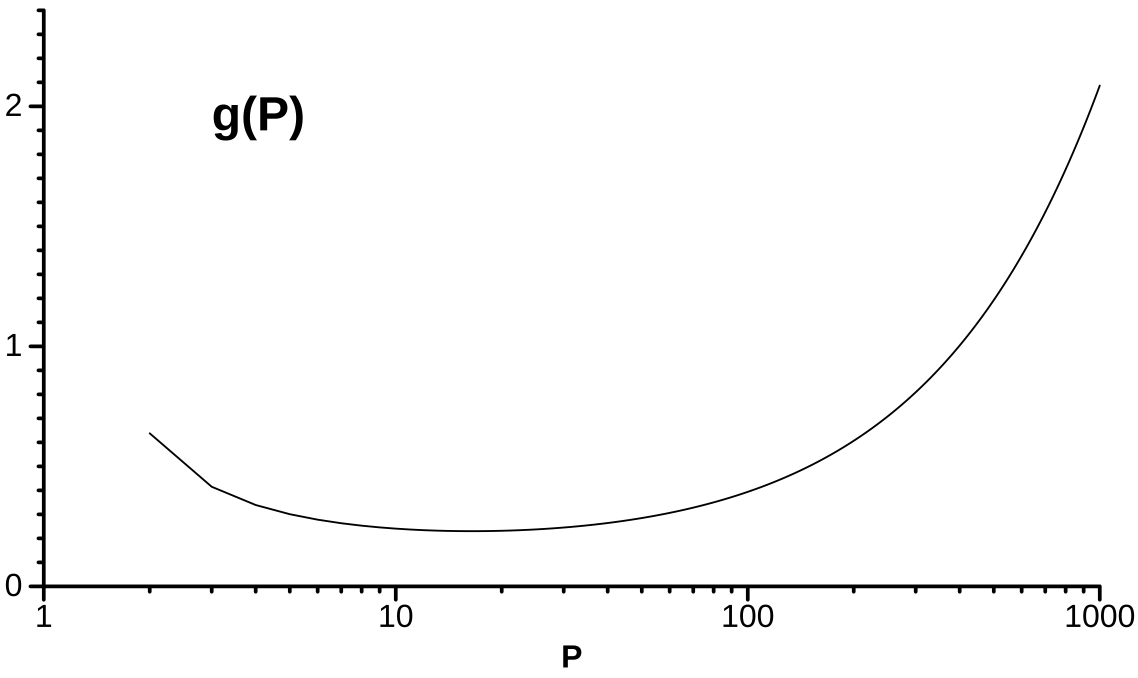
- Best P found by taking the derivative with respect to P. Must solve

$$P\ \ln(P) - P = 1 + \frac{k\ t_{\text{sequential}}}{t_{\text{random}}}$$

The solution depends upon the memory size k, and the two disk times t_{random} and $t_{sequential}$.

P	$P\ ln(P)$ - P	P	$P\ ln(P)$ - P
3	0.3	4	1.5
5	3.0	6	4.8
7	6.6	8	8.6
9	10.8	10	13.0
15	25.6	20	39.9
25	55.5	30	72.0
35	89.4	40	107.6
45	126.3	50	145.6

P-way Merge VII



$$k = 1000,\ t_{\text{random}} = 14\text{ms},\ t_{\text{sequential}} = 0.4\ \text{ms}.$$

$$\mathbf{g}(P) = \frac{1}{\ln(P)}\left(\frac{(P+1)}{\mathbf{k}} \cdot t_{\text{random}} + t_{\text{sequential}}\right)$$

The total merge time is $2\,n\,m \ln(m)\,\mathbf{g}(P)$