B+Tree (Bottom Up) Construction. I

- Efficiently build a B+tree index for sorted data.
- 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.

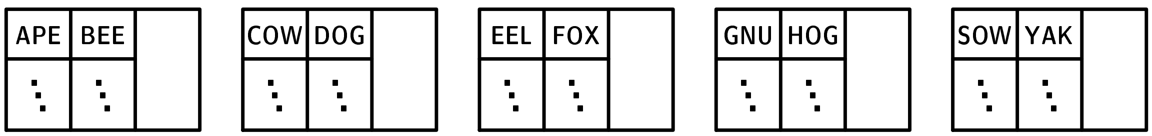
2. Read next data node; if its smallest key SK is larger than the previous router value, then look at the key in the rightmost end of the previous external node. Morover, if it is the same as SK, look for the leftmost occurrence of SK. Put SK and its leftmost address into the not full, internal node at the lowest level. 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. Whenever a node fills, write the node. Otherwise, if SK equal to the previous router value, then do nothing for this external node. 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 the top of tree and work down ensuring sufficient occupancy. (Consider only internal nodes.)

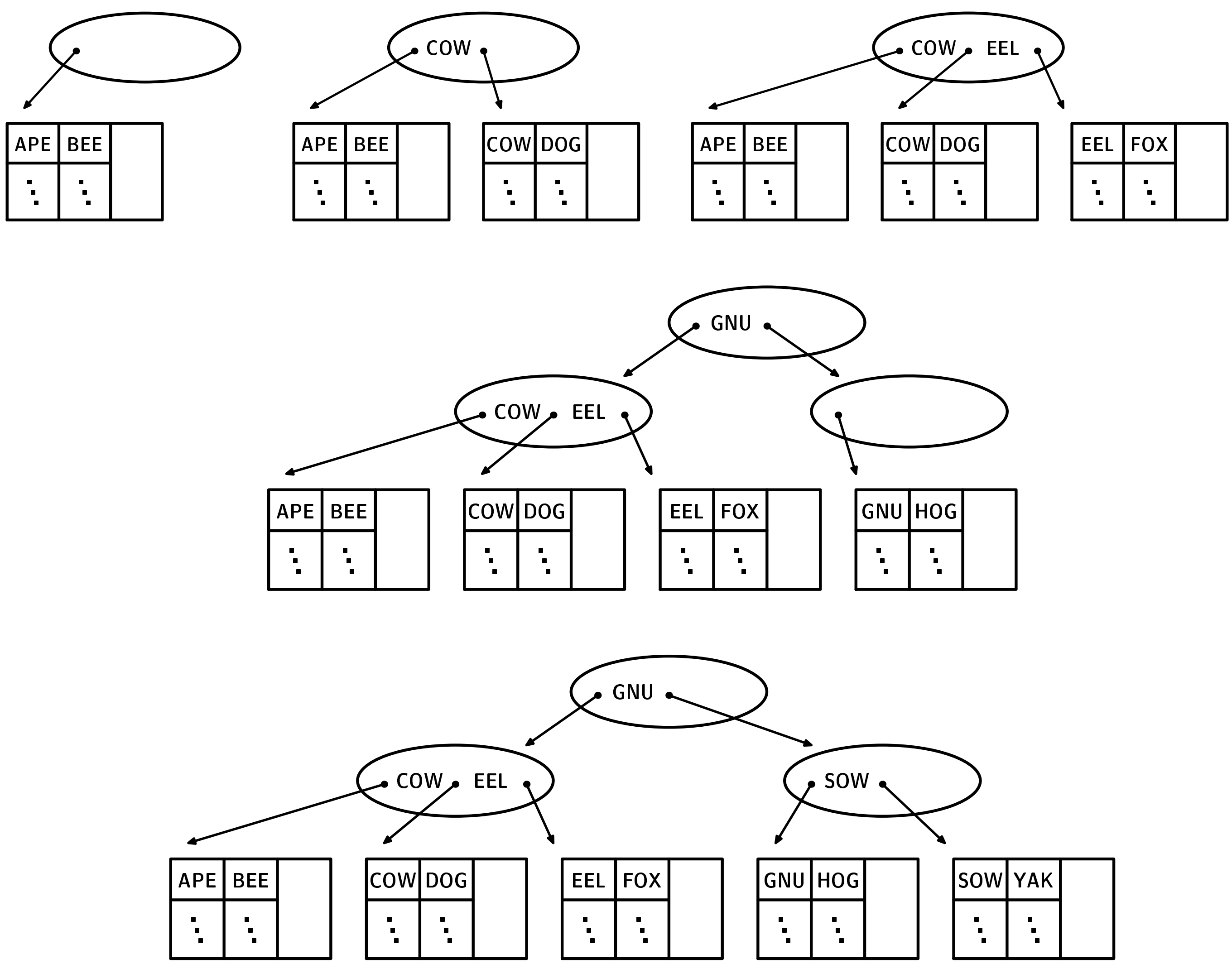
B+Tree (Bottom Up) Construction III

Data file contains 10 records, accessed as 5 nodes

each containing 2 records and one spare slot.

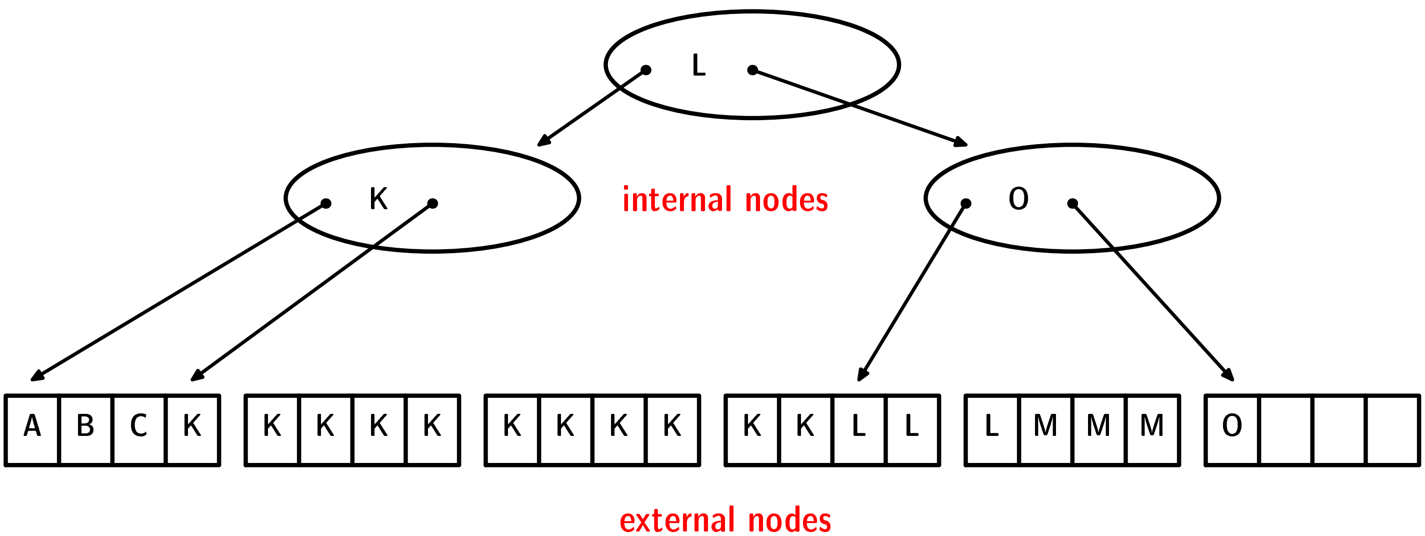


sequence of order 3 B+trees



B+Tree (Bottom Up) Construction IV

Data file contains 21 summary records, accessed as 6 nodes each containing space for 4 summary records; internal nodes have order three.



External Sorting V

Two phases – distribution & merge.

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

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

- Run-time is 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, put into output buffer and Sift until the input buffer is empty. Write (empty) the output buffer. Continue step 1 until all input data has been read.

HeapSort II

- **Heap Order:**

```
int array [ 1 .. n ] ;

for ( i = 2 ; i <= n ; i++ )
    for ( j = i ;
      j > 1 && array[j] < array[j/2] ; j >>= 1 )
        swap ( array[j] , array[j/2] ) ;
```

- **Sift:**

```
array[1] = array[n--] ;

for ( i = 2 ; i <= n ; i <<= 1 ) {
    if ( i < n && array[i] > array[i+1] ) i++ ;
    if ( array[i/2] > array[i] )
        swap ( array[i/2] , array[i] ) ;
    else break ;
}
```