Deleting a Data Entry from a B+ Tree

- Start at root, find leaf $L$ where entry belongs.
- Remove the entry.
  - If $L$ is at least half-full, done!
  - If $L$ has only $d-1$ entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as $L$).
    - If re-distribution fails, merge $L$ and sibling.
- If merge occurred, must delete entry (pointing to $L$ or sibling) from parent of $L$.
- Merge could propagate to root, decreasing height.
Current B+ Tree

Delete 19*
Delete 20*
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

- Deleting 19* is easy.
- Deleting 20* is done with re-distribution.
  Notice how middle key is *copied up*.
... And Then Deleting 24*

- Must merge.
- Observe `toss’ of index entry (on right), and `pull down’ of index entry (below).
Example of Non-leaf Re-distribution

- Tree is shown below during deletion of $24^*$.  
- In contrast to previous example, can re-distribute entry from left child of root to right child.
After Re-distribution

- Intuitively, entries are re-distributed by `pushing through' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we’ve re-distributed 17 as well for illustration.
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic`; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt, David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.
**Prefix key compression**

Compress to ‘Dav’ or ‘Davi’

<table>
<thead>
<tr>
<th>Daniel Lee</th>
<th>David Smith</th>
<th>Devarakonda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dante Wu</td>
<td>Darius Rex</td>
<td>...</td>
</tr>
</tbody>
</table>
If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.

Bulk Loading can be done much more efficiently.

Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.
Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)

- Much faster than repeated inserts, especially when one considers locking!
Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.

- Option 2: Bulk Loading
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.
A Note on `Order’

- Order (d) concept replaced by physical space criterion in practice (`at least half-full’).
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3)).
Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (F) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
  - Typically, 67% occupancy on average.
  - If data entries are data records, splits can change rids!
Summary (Contd.)

- Key compression increases fanout, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.