Part 2 - B-Tree
Using B-Trees and B+-Trees as Dynamic Multi-level Indexes

- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.

- In B-Tree and B+-Tree data structures, each node (can) correspond to a disk block.

- Each node is kept between half-full and completely full.

- An insertion into a node that is not full is quite efficient; if a node is full the insertion causes a split into two nodes.

- Splitting may propagate to other tree levels.

- A deletion is quite efficient if a node does not become less than half full.

- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes.

Difference between B-tree and B+-tree:

- In a B-tree, pointers to data records exist at all levels of the tree.

- In a B+-tree, all pointers to data records exists at the leaf-level nodes.

- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree
B-Trees

- Similar to indexed sequential files, B-trees can be accessed both sequentially and directly.

- B in B-tree does not stand for binary tree. (Some say it stands for Bayer. One of the original authors.)

- Binary trees have a *branching factor* of 2 however, B-trees can have many. (B-tree is multi-way search tree)

- Traditional trees grow from their leaves (*Top down*)

- In a B-tree, new nodes are inserted into the leaf level however, if the B-tree must increase in height, it is the root level or top of the tree that changes. (*Bottom up*)
  - **Advantage of Bottom up growth:**
    - Automatically preserves the balance of the tree!
    - No rotation is necessary.
B-tree Definitions:

- **Node:**
  - A B-tree node consists of one or more records and links.
  - Typically (N) records and (N+1) links.

- **Capacity Order:**
  - Also known as simply “order”
  - The order of the tree is the number of records that may be stored in a node minus 1.
  - **For example** when the order \( p = 3 \) you can have 2 key values and 3 pointers in the node.

**Example:**  *(B-tree of “Order” 3)*

<table>
<thead>
<tr>
<th>Link</th>
<th>Key (Record)</th>
<th>Link</th>
<th>Key (Record)</th>
<th>Link</th>
</tr>
</thead>
</table>

- **Note:**
  - The order of a B-tree usually depends upon parameters such as Block Size, Page Size and record size.
  - Block Size is physical, and Page Size is logical and determined by the operating system.
• A B-tree with a Capacity Order “p” has:
  
  • # of keys:
    \[ \frac{p}{2} \leq \text{keys} \leq p-1 \]  // except for the root node which has between 1 and p-1 keys.

  • # of links:
    \[ \frac{p}{2} \leq \text{links} \leq p \]  // except for the root which has between 2 and p links.

• Minimum Capacity Rule:
  • All leaf except for the ROOT node must be at least half full.

• All leaf nodes in a B-tree are at the same level.
Example: (B-tree of “Order” 2)

<table>
<thead>
<tr>
<th>Link</th>
<th>Key (Record)</th>
<th>Link</th>
</tr>
</thead>
</table>

Example: (B-tree of “Order” 3)

<table>
<thead>
<tr>
<th>Link</th>
<th>Key (Record)</th>
<th>Link</th>
<th>Key (Record)</th>
<th>Link</th>
</tr>
</thead>
</table>

- Note that in the above figure both the key and the record are stored in the data portion.

- Alternatively, the key to the data, as well as a pointer to the record or pointer to a block which contains the record can be stored in the b-tree node. See below

<table>
<thead>
<tr>
<th>Link</th>
<th>Key</th>
<th>Pointer to Record</th>
<th>Link</th>
</tr>
</thead>
</table>
**Inserting into a B-tree:**

- Similar to a binary search tree.

  Search the tree
  
  if the record is found
  
  Print “duplicate”
  
  Else
  
  Read the node into memory (a block)
  
  If (the block just read has enough room for the new record)
  
  Insert the New record in the node (in sorted order)
  
  Write the node back out.
  
  Else // there is not enough room in that node
  
  Split the node into two nodes.
  
  Move the middle record to the next higher level.
Example -1:

Insert the following items in a B-tree of order 1. (2 keys, 3 links)

(cat, ant, dog, cow, rat, pig and gnu)
Example -2

Insert the following items in a B-tree of order 2. (4 keys, 5 links)

(80, 50, 100, 90, 60, 65, 70, 75, 55, 64, 51, 76, 77, 78, 200, 300, 150)
Advantages of B-Trees:

- No rotation is necessary
- Natural balancing property
- Bottom up growth instead of the typical top-down in regular trees.
- Performance $O(\log_d N)$ worst case performance where, “d” is the “capacity order” (# of records stored in a node)
- Each node can hold many records (as much as you can store in a block)
- I/O is done at node (block/page) level.
- Relatively flat hierarchy which provides more efficiency:
  - A B-tree of capacity order 50 containing on million records requires at most 4 retrieval probes to locate a record.

  $O(\log_{50} 1000_000)$

  $1000_000 = 50^x$

  $x \approx 4$

- In-order (LNR) traversal of a B-tree provides a sorted list.
Disadvantages of Basic B-Trees:

- Storage utilization of B-tree is not optimal. (Simulation has shown about 69% of the storage is used. Elmasri & Navathe 5th edition)

- Insertion and Deletions are more complex.
Deletion of Records from B-tree

물을 지우는 B-트리

😊 The reverse of an insertion.

1) The B-tree index structure must be preserved

2) The minimum space utilization must be maintained. (All nodes except the root must be at least half full.)

Delete Algorithm:

1) Find the record and delete it!

2) if (deleted-record in a non-leaf node)
   * Replace the deleted record with its in-order successor
   * if (removing the in-order successor violates the minimum capacity constraint (perform step 4)

3) if (deleted-record in leaf_node) and (minimum capacity == OK)
   * We are done!
   * No more nodes are affected.
   * Rewrite the node back to disk

4) if (deleted-record in leaf_node) and (minimum capacity != OK)
   * Redistribute the node from one of its siblings.

   Case 1: Borrow one or more records from a sibling,
           (as long as they themselves do not violate the minimum capacity rule!)

   Case 2: Coalesce with a sibling

   Case 3: If necessary, shrink the tree by one level, or borrow from the parent.
Deleting from a Non-Leaf Node:

1) Replace the deleted record with its in-order successor.

   Example: Delete 84

   ** This process will replace 84 with 87.

   * if (removing the in-order successor violates the minimum capacity constraint (perform step 4, see above)
Resolving the Minimum Capacity Constraint: (Case 1: Redistribution)

Delete 60

** After deleting 60, node A will violate the minimum capacity constraint.

Apply Case 1: (Redistribute or borrow from sibling)

Note: The comparison record (85) has been changed with (90) to preserve the lexicographic ordering.
Resolving the Minimum Capacity Constraint: (Case 2: Coalescing)

Delete 60

** Redistribution is not possible. (Violates the minimum capacity rule)

Apply Case 2: (Coalesce with a sibling node)

** Combine node A, the comparison record (85) and node B in the same node.
Note 1: If removing a record from a parent node would cause its capacity constraint to be violated, then a redistribution or coalescing would have to be performed with one of its sibling nodes.

Note 2: If the root has only one record and that record becomes coalesced with its child node(s), then the root is deleted and the level of the tree is decreased by one.