CMSC 420: Lecture 17
B-Trees

Data Structures for External Memory: While binary trees are great data structures for ordered dictionaries stored in main memory, these data structures are really not appropriate for data stored on external memory systems (i.e., disks). When accessing data on a disk, latency (that is, the delay waiting for the transfer to begin) is significant, but an entire *block* (or "*page*") of memory input at once. So it makes sense to design the tree so that each node of the tree essentially occupies one entire page.

This idea applies more generally to modern memory systems, which are organized hierarchically. Memory is partitioned into various levels of caches, with each successive level having higher latency and larger page size. The data structure we will discuss today is appropriate whenever memory can be accessed efficiently in blocks.

This suggests the generalization of *multiway search trees*, where each node is allocated so that it coincides with a single page. Each node of a standard binary search tree store a single key value and two children left and right storing keys that are smaller than and greater than this key, respectively. In a *j-ary* multiway search tree node, a node stores references to j different subtrees, $T_1, \ldots, T_j$ and contains $j-1$ key values, $a_1 < \ldots < a_{j-1}$, such that subtree T_i stores nodes whose key values x such that $a_{i-1} < x < a_i$. (To handle the boundary cases, let's make the convention that $a_0 = -\infty$ and $a_j = +\infty$, but these are not stored in the node.)

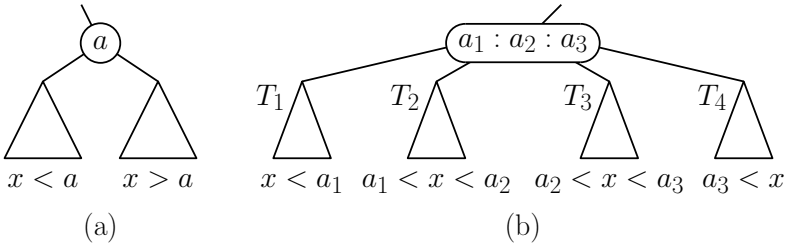

Fig. 1: Binary and 4-ary search tree nodes.

B-trees: B-trees are multiway search trees, in which we achieve balance by constraining the "width" of each node. B-trees were first introduced way back in 1970 by Rudolf Bayer and Edward McCreight. They have proven to be very popular. The 2-3 tree that we studied in an earlier lecture is a special case (indeed, the smallest special case) of a B-tree. Numerous modifications and adaptations of B-trees have been developed over the years. We will present one, fairly simple, formulation. (Later in the lecture we will discuss a particularly popular variant, called B+ trees.)

For any integer $m \geq 3$, a *B-tree of order m* is a multiway search tree has the following properties (see Fig. 2):

- The root is either a leaf or has between two and m children.
- Each node except the root has between $\lceil m/2 \rceil$ and m children (which may be empty, that is `null`). A node with j children contains $j-1$ key.
- All leaves are at the same level of the tree.

The 2-3 tree that we presented earlier is an example of a B-tree of order 3. The typical fan-out values for B-trees are quite large. For example, B-trees of order of around 100 are common

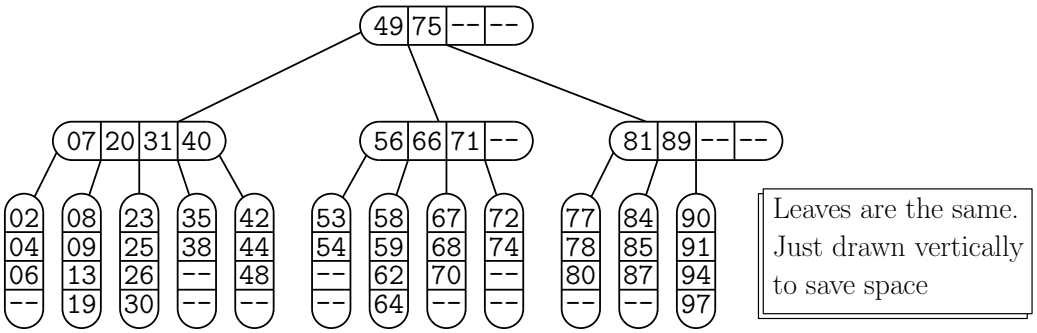

Fig. 2: B-tree of order 5. (Each node, except possibly the root has 3–5 children.)

in practice. A node in such a tree has between 50 and 100 children and holds between 49 and 99 keys. Of course, with such high fan-outs, the heights of the tree are quite small.

Height Analysis: The following theorem show that as fan-out of a B-tree grows, the height of the tree decreases.

Theorem: A B-tree of order m containing n keys has height at most $\frac{\lg n}{\gamma}$, where $\gamma = \lg \frac{m}{2}$.

Proof: The formal proof is a bit messy, so we will instead prove a simpler result. Let's let n denote the number of keys just at the leaf level. (This is reasonable, because in any multiway tree, a constant fraction of the nodes are leaves, and as the degree increases, the fraction increases.) Let $h \geq 1$ denote the height of our tree, and let $N(h)$ denote the smallest possible number of keys in the leaves of this tree. Since $N(h) \leq n$, it suffices to show that $h \leq \frac{\lg N(h)}{\gamma}$.

Recall that the degree of a node is its number of children. The root node has degree at least two, and all other nodes have degree at least $d = m/2$. Thus, there are at least two nodes at depth 1, and with each additional level the number increases by a factor of d. Thus, there are at least $2d$ nodes at depth 2, $2d^2$ at depth 3, and generally $2d^{h-1}$ at depth h. Each of these leaf nodes contains at least d keys, implying the total number of keys at the leaf level is at least $2d^h$. Thus, we have

$$\begin{aligned} & N(h) \;\geq\; 2d^h \;\geq\; d^h \\ \Longrightarrow\quad & \lg N(h) \;\geq\; \lg(d^h) \;=\; h \lg d \qquad \text{(taking lg on both sides)} \\ \Longrightarrow\quad & \frac{\lg N(h)}{\lg d} \;\geq\; h. \end{aligned}$$

Recalling that $\gamma = \lg \frac{m}{2} = \lg d$, a tree with n keys has height at least $h \leq \frac{\lg N(h)}{\gamma}$, as desired.

For example, when $m = 100$, this implies that the height of the B-tree is not greater than $(\lg n)/5.6$, that is, it is 5.6 times smaller than a binary search tree. For example, this means that you can store over a 100 million keys in a search structure of height roughly five!

Node structure: Although B-tree nodes can hold a variable number of items, this number generally changes dynamically as keys are inserted and deleted. Therefore, every node is allocated with the maximum possible size, but most nodes will not be fully utilized. (Experimental

B-Tree Node

```
final int M = ...                   // order of the B-tree

class BTreeNode {
    int         nChildren;          // current number of children (from M/2 to M)
    BTreeNode   child[M];           // children pointers
    Key         key[M-1];           // keys
    Value       value[M-1];         // values
}
```

studies show that B-tree nodes are on average about 2/3 utilized.) The code block below shows a possible Java implementation of a B-tree node implementation.

Note that 2-3 trees and 2-3-4 trees discussed in earlier lectures are special cases (when $M = 3$ and $M = 4$, respectively.)

Search: Searching a B-tree for a key x is a straightforward generalization of binary tree searching. When you arrive at an internal node with keys $a_1 < a_2 < \ldots < a_{j-1}$ search (either linearly or by binary search) for x in this list. If you find x in the list, then we have found x. Otherwise, determine the index i such that $a_{i-1} < x < a_i$. (Recall that $a_0 = -\infty$ and $a_j = +\infty$.) Then recursively search the subtree T_i. When you arrive at a leaf, search all the keys in this node. If it is not here, then x is not in the B-tree.

Since nodes may be quite wide, should we employ a fast search method such as binary search? Well, you could, but normally latency times (the time needed to load a node from external memory) are typically so much higher than processing times that simple linear search is fast enough.

Restructuring: In an earlier lecture, we showed how to restructure 2-3 trees. We had three mechanisms: splitting nodes, merging nodes, and subtree adoption. We will generalize each of these operations to general B-trees.

Key Rotation (Adoption): Recall that a node in a B-tree can have from $\lceil m/2 \rceil$ up to m children, and the number of keys is smaller by one. As a result of insertion or deletion, a node may acquire one too many ($m + 1$ children and hence, m keys) or one too few ($\lceil m/2 \rceil - 1$ children and hence, $\lceil m/2 \rceil - 2$ keys).

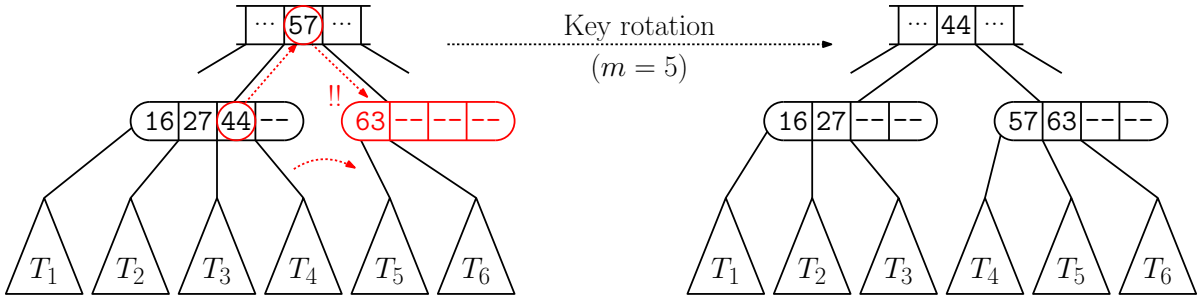

Fig. 3: Key rotation for a B-tree of order $m = 5$.

The easiest way in which to remedy the imbalanced is to move a child into or from one of your siblings, assuming that you have a sibling can absorb this change. This is called

key rotation (or as I call it, *adoption*). For example, in Fig. 3, the node in red has too few children, and since its left sibling can spare a child, we move this node's rightmost child over, sliding the associated key value up to the parent and we take the parent's key value.

This operation is not always possible, because it depends on the existence of a sibling with a proper number of keys. Because allocating and deallocating nodes is a relatively expensive operation, *this is the preferred rebalancing operation.*

Node Splitting: As the result of insertion, a node may acquire one too many children ($m+1$ children and hence, m keys). When this happens and key rotation is not available, we split the node into two nodes, one having $m' = \lceil m/2 \rceil$ children and the other having the remaining $m'' = m + 1 - \lceil m/2 \rceil$ children. (For example, if $m = 8$, when a node has $m + 1 = 9$ children it is split into one of size $m' = 4$ and the other $m'' = 5$.) Clearly, the first node has an acceptable number of children. The following lemma demonstrates that the other node has an acceptable number of children as well.

Lemma: For all $m \geq 2$, $\lceil m/2 \rceil \leq m + 1 - \lceil m/2 \rceil \leq m$.

Proof: If m is even, then $\lceil m/2 \rceil = m/2$, and the middle expression in the inequality reduces to $m + 1 - m/2 = m/2 + 1$. Thus, the claim is equivalent to

$$\frac{m}{2} \;\leq\; \frac{m}{2} + 1 \;\leq\; m,$$

which is clearly true for any even $m \geq 2$. On the other hand, if m is odd then $\lceil m/2 \rceil = (m+1)/2$, and the middle expression in the inequality reduces to $m + 1 - (m + 1)/2 = (m + 1)/2$. Thus, the claim is equivalent to

$$\frac{m+1}{2} \;\leq\; \frac{m+1}{2} \leq m,$$

which is also clearly true for any $m \geq 1$.

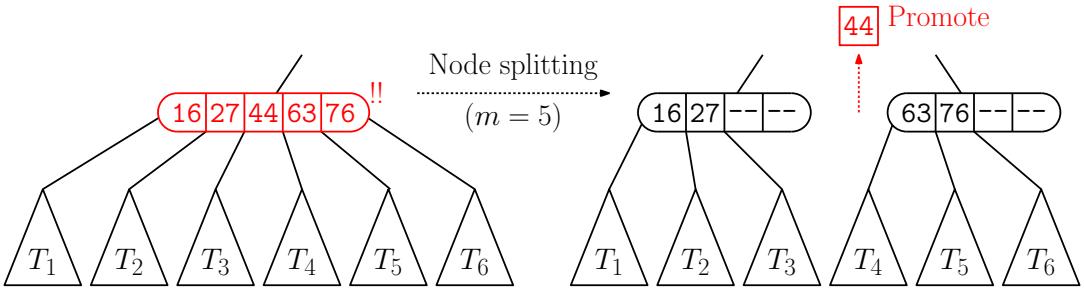

Fig. 4: Node splitting for a B-tree of order $m = 5$.

Returning to node splitting, we create two nodes and distribute the smallest m' subtrees to the first and the remaining m'' to the second node (see Fig. 4). Among the $m - 1$ keys, $m' - 1$ smallest keys go with the first node and the $m'' - 1$ largest keys go with the other node. Since $(m' - 1) + (m'' - 1) = m - 2$, we have one extra key that does not fit into either of these nodes. This node is promoted to the parent node. (As with 2-3 trees, if we do not have a parent, we create a new root node with this single key and just two children. By the way, this is the reason that we allowed the root to have fewer than $\lceil m/2 \rceil$ children.)

Since the parent acquires an extra key and extra child, the splitting process may propagate to the parent node.

Node Merging: As the result of deletion, a node may have one too few children ($\lceil m/2 \rceil - 1$ children and hence, $\lceil m/2 \rceil - 2$ keys). When this happens and key rotation is not available, we may infer that its siblings have the minimum number $\lceil m/2 \rceil$ children. We merge this node with either of its siblings into a single node having a total of $m' = (\lceil m/2 \rceil - 1) + \lceil m/2 \rceil = 2\lceil m/2 \rceil - 1$ children. The following lemma demonstrates that the resulting node has an acceptable number of children.

Lemma: For all $m \geq 2$, $\lceil m/2 \rceil \leq 2\lceil m/2 \rceil - 1 \leq m$.

Proof: If m is even, then $\lceil m/2 \rceil = m/2$, and the middle expression in the inequality reduces to $2(m/2) - 1 = m - 1$. Thus, the claim is equivalent to

$$\frac{m}{2} \;\leq\; m - 1 \;\leq\; m,$$

which is easily true for any $m \geq 2$. On the other hand, if m is odd then $\lceil m/2 \rceil = (m+1)/2$, and the middle expression in the inequality reduces to m. Thus, the claim is equivalent to

$$\frac{m+1}{2} \;\leq\; m \;\leq\; m,$$

which is easily true for any $m \geq 1$.

Returning to node merging, we merge the two nodes into a single node having m' children (see Fig. 5). The number of keys from the two initial nodes is $\lceil m/2 \rceil - 2 + \lceil m/2 \rceil = 2\lceil m/2 \rceil - 2 = m' - 2$, which is one too few. We demote the appropriate key from the parent's node to yield the desired number of keys.

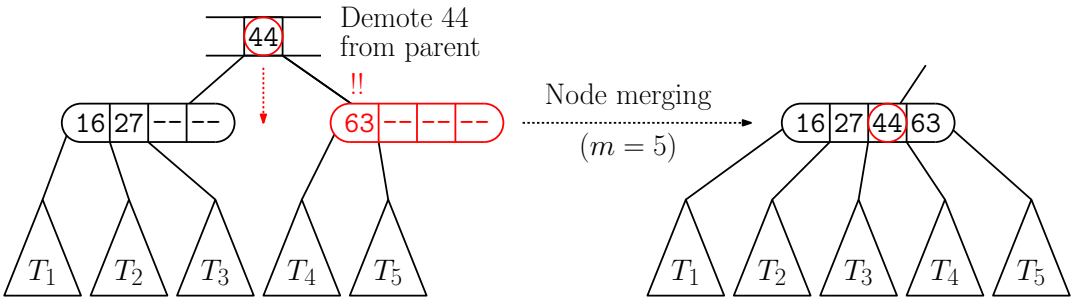

Fig. 5: Node merging for a B-tree of order $m = 5$.

Since the parent has lost a key and a child, the merging process may propagate to the parent node.

Given these operations, we can now describe how to perform the various dictionary operations.

Insertion: In the case of 2-3 trees, we would always split a node when it had too many keys. With B-trees, creating nodes is a more expensive operation. So, whenever possible we will try to employ key rotation to resolve nodes that are too full, and we will fall back on node splitting only when necessary.

To insert a key into a B-tree of order m, we perform a search to find the appropriate leaf into which to insert the node. If we find the key, then we signal a duplicate-key error. Otherwise, if the leaf is not at full capacity (it has fewer than $m - 1$ keys) then we simply insert it and are done. Note that this will involve sliding keys around within the leaf node to make room

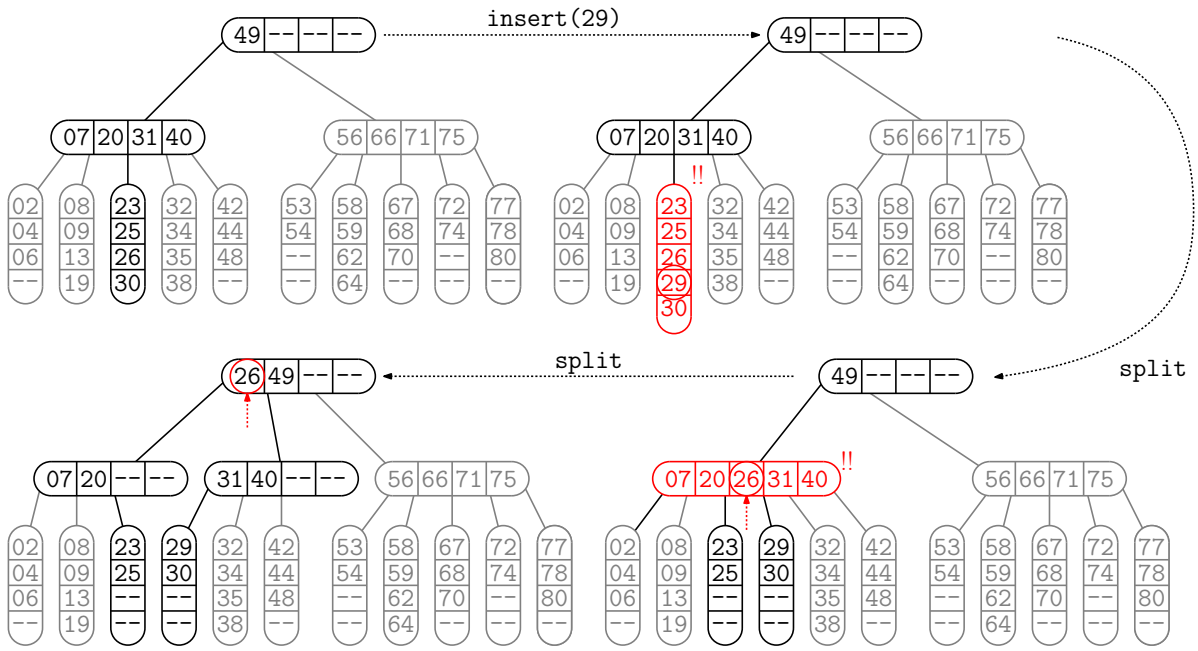

Fig. 6: Insertion of key 29 (order $m = 5$). It is inserted into the leaf node $(23, 25, \ldots)$ which splits, causing the middle key 26 to be promoted to the parent, which also splits, causing the middle key 26 to be promoted to the root.

for the new entry, but since m is assumed to be a constant (e.g., the size of one disk page), we ignore this extra cost.

Otherwise the node *overflows* and to remedy the situation, we first check whether either sibling is less than full. If so, we perform a rotation moving the extra key and child into this sibling. Otherwise, we perform a node split as described above (see Fig. 6). When this happens, the parent acquires a new child and new key, and thus the splitting process may continue with the parent node.

Deletion: As in binary tree deletion we begin by finding the node to be deleted. We need to find a suitable replacement for this node. This is done by finding the largest key in the left child (or equivalently the smallest key in the right child), and moving this key up to fill the hole. This key will always be at the leaf level. This creates a hole at the leaf node. If this leaf node still has sufficient capacity (at least $\lceil m/2 \rceil - 1$ keys) then we are done.

Otherwise, we have an underflow situation at this node. As with insertion we first check whether a *key rotation* is possible. If one of the two siblings has at least one key more than the minimum, then we rotate the extra key into this node, and we are done (see Fig. 7).

If this is not possible, then any siblings of ours must have the minimum number of $\lceil m/2 \rceil$ children, and so we can apply a node node merge (see Fig. 8).

The removal of a key from the parent's node may cause it to underflow. Thus, the process may need to be repeated recursively up to the root. If the root now has only one child, and we make this single child the new root of the B-tree.

B+ trees: B-trees have been very successful, and a number of variants have been proposed. A particularly popular one for disk storage is called a *B+ tree*. The key differences with the

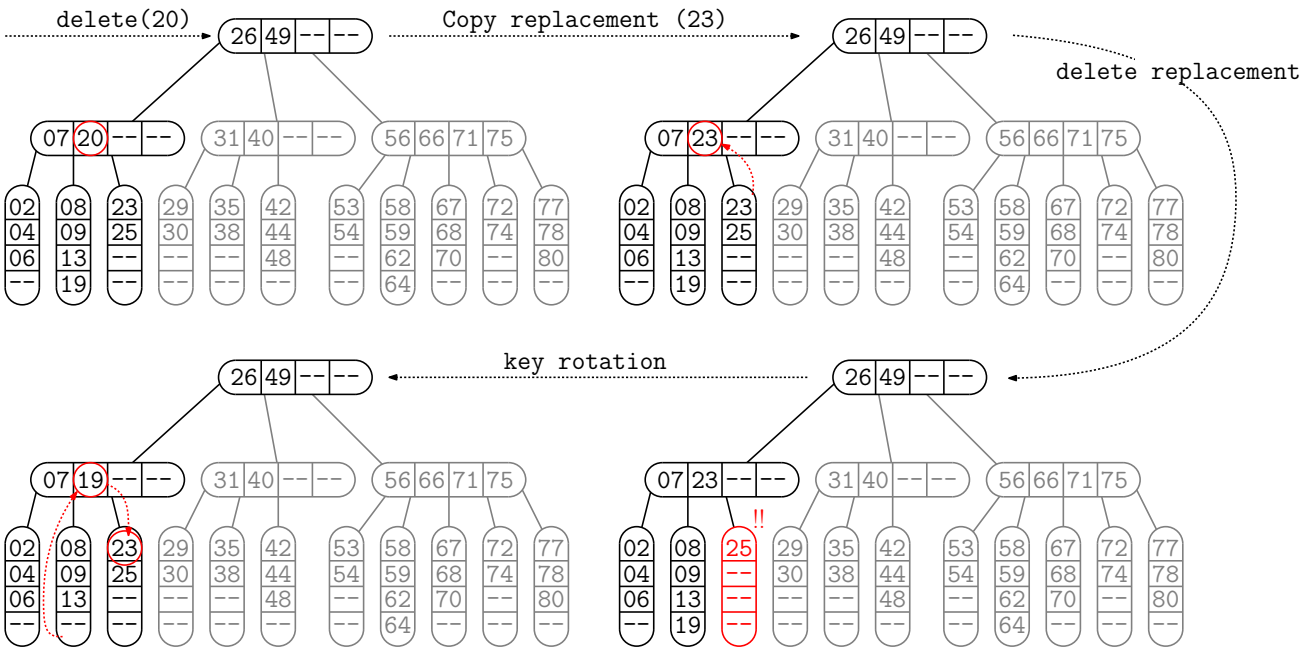

Fig. 7: Deletion of key 20 (order $m = 5$). Since 20 is not a leaf, we find the replacement 23 (inorder successor) and copy it there. We then delete 23 from its leaf, which underflows. The sibling rotates the key 19 to the parent and the parent gives up its key 23.

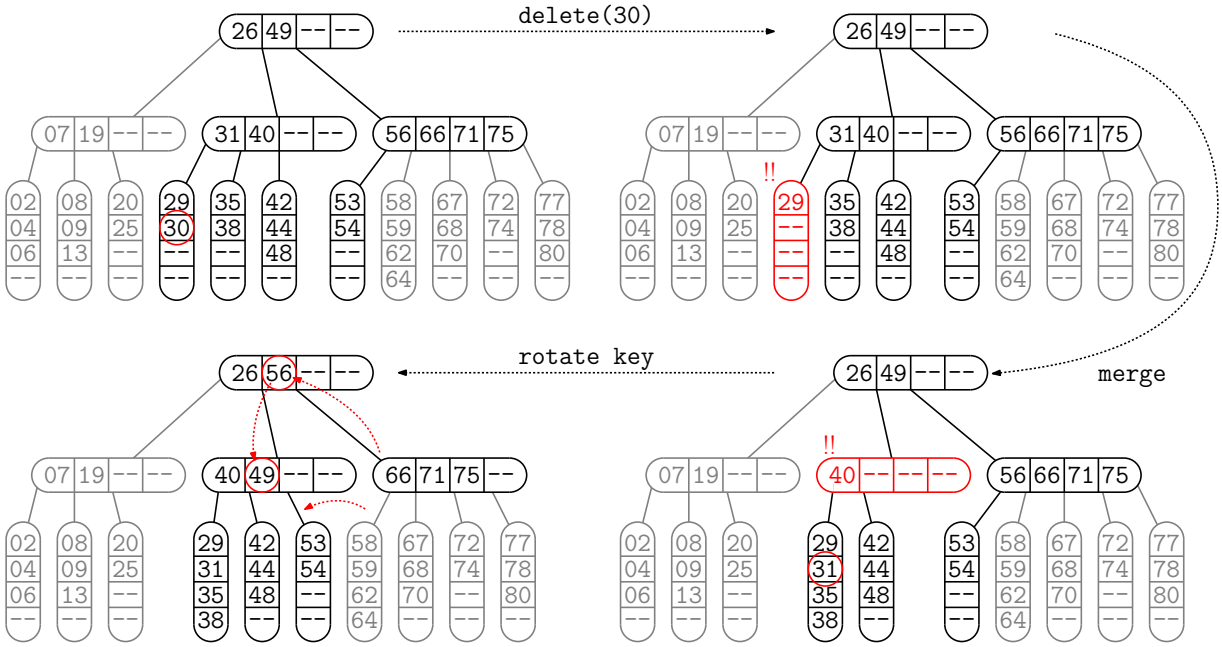

Fig. 8: Deletion of key 30 (order $m = 5$). Its leaf is underfull, but its sibling cannot give up a key, so we merge with $(35, 38)$, demoting the middle key 31 from the parent. Now the parent 40 is underfull, but its right sibling can spare a key, so 56 is rotated to the parent and 49 is rotated down.

standard B-tree as the following:

- Internal and leaf nodes are different in structure:
  - Internal nodes store keys only, no values. The keys in the internal nodes are used solely for locating the leaf node containing the actual data, so it is not necessary that every key appearing in an internal node need correspond to an actual key-value pair.
  - All the key-value pairs are stored in the leaf nodes. There is no need for child pointers. (This also saves space.)
- Each leaf node has a *next-leaf* pointer, which points to the next leaf in sorted order.

Storing keys only in the internal nodes saves space, and allows for increased fan-out. This means the tree height is lower, which reduces number of disk accesses. Thus, the internal nodes are merely an *index* to locating the actual data, which resides at the leaf level. (The policy regarding which keys a subtree contains are changed. Given an internal node with keys $\langle a_1, \ldots, a_{j-1} \rangle$, subtree T_j contains keys x such that $a_{i-1} < x \leq a_i$.)

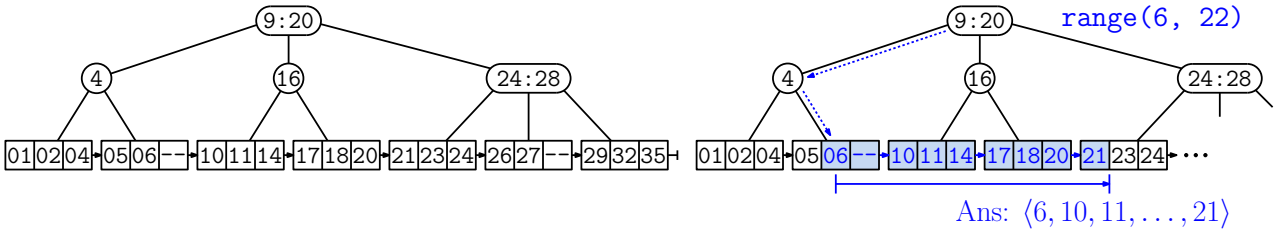

Fig. 9: B+ tree of order $m = 3$, where leaves can hold up to 3 keys.

The next-leaf links enable efficient *range reporting* queries. In such a query, we are asked to list all the keys in a range $[x_{\min}, x_{\max}]$. We simply find the leaf node for $x_{\min}$ and then follow next-leaf links until exceeding $x_{\max}$.