Lecture 12: B-trees

Read: Section 4.7 of Weiss and 5.2 in Samet.

B-trees: Although it was realized quite early it was possible to use binary trees for rapid searching, insertion and deletion in main memory, these data structures were really not appropriate for data stored on disks. When accessing data on a disk, an entire *block* (or *page*) is input at once. So it makes sense to design the tree so that each node of the tree essentially occupies one entire block. Although it is possible to segment binary trees in a clever way so that many neighboring nodes are squeezed into one disk block it is difficult to perform updates on the tree to preserve this proximity.

An alternative strategy is that of the *B-tree*, which solves the problem by using *multiway trees* rather than binary trees. Standard binary search tree have two pointers, left and right, and a single key value k that is used to split the keys in the left subtree from the right subtree. In a j-ary multiway search tree node, we have j child pointers, $p_1, p_2, \ldots, p_j$, and $j-1$ key values, $k_1 < k_2 < \ldots < k_{j-1}$. We use these key values to split the elements among the subtrees. We assume that the data values k located in subtree pointed to be p_i satisfy $k_i \leq k < k_{i+1}$, for $1 \leq i \leq j$. For convenience of notatoin, we imagine that there are two sentinel keys, $k_0 = -\infty$ and $k_j - +\infty$. See the figure below.

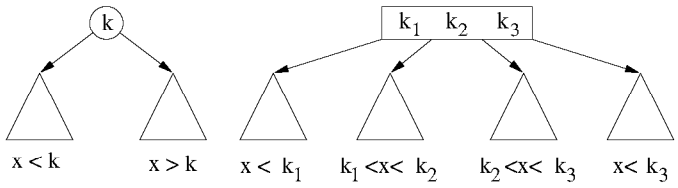

Figure 35: Binary and 4-ary search tree nodes.

B-trees are multiway search trees, in which we achieve balance by constraining the "width" of each node. As with all other balanced structures, we allow for a certain degree of flexibility in our constraints, so that updates can be performed efficiently. A B-tree is described in terms of a parameter m, which controls the maximum degree of a node in the tree, and which typically is set so that each node fits into one disk block.

Our definition differs from the one given in Weiss. Weiss's definition is more reasonable for actual implementation on disks, but ours is designed to make the similarities with other search trees a little more apparent. For $m \geq 3$, *B-tree of order m* has the following properties:

- The root is either a leaf or has between 2 and m children.
- Each nonleaf node except the root has between $\lceil m/2 \rceil$ and m (nonnull) children. A node with j children contains $j-1$ key values.
- All leaves are at the same level of the tree, and each leaf contains between $\lceil m/2 \rceil - 1$ to $m-1$ keys.

So for example, when $m = 4$ each internal node has from 2 to 4 children and from 1 to 3 keys. When $m = 7$ each internal node has from 4 to 7 children and from 3 to 6 keys (except the root which may have as few as 2 children and 1 key). Why is the requirement on the number of children not enforced on the root of the tree? We will see why later.

The definition in the book differs in that all data is stored in the leaves, and the leaves are allowed to have a different structure from nonleaf nodes. This makes sense for storing large

data records on disks. In the figure below we show a B-tree of order 3. In such a tree each node has either 2 or 3 children, and hence is also called a *2-3 tree*. Note that even though we draw nodes of different sizes, all nodes have the same storage capacity.

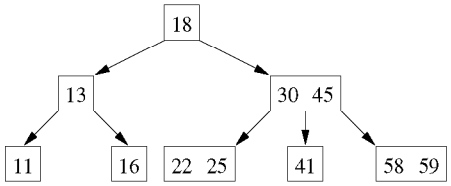

Figure 36: B-tree of order 3, also known as a 2-3 tree.

Height analysis: Consider a B-tree (of order m) of height h. Since every node has degree at most m, then the tree has at most m^h leaves. Since each node contains at least one key, this means that $n \geq m^h$. We assume that m is a constant, so the height of the tree is given asymptotically by

$$h \leq \log_m n = \frac{\lg n}{\lg m} \in O(\log n).$$

This is very small as m gets large. Even in the worst case (excluding the root) the tree essentially is splitting $m/2$ ways, implying that

$$h \leq \log_{(m/2)} n = \frac{\lg n}{\lg(m/2)} \in O(\log n).$$

Thus, the height of the tree is $O(\log n)$ in either case. For example, if $m = 256$ we can store 100,000 records with a height of only three. This is important, since disk accesses are typically many orders of magnitude slower than main memory accesses, it is important to minimize the number of accesses.

Node structure: Unlike skip-lists, where nodes are typically allocated with different sizes, here every node is allocated with the maximum possible size, but not all nodes are fully utilized. A typical B-tree node might have the following Java class structure. In this case, we are storing integers as the elements. We place twice as many elements in each leaf node as in each internal node, since we do not need the storage for child pointers.

```
const int M = ...                   // order of the tree

class BTreeNode {
    int         nChildren;          // number of children
    BTreeNode   child[M];           // children
    int         key[M-1];           // keys
};
```

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 $k_1 < k_2 < \ldots < k_{j-1}$ search (either linearly or by binary search) for x in this list. If you find x in the list, then you are done. Otherwise, determine the index i such that $k_i < x < k_{i+1}$. (Recall that $k_0 = -\infty$ and $k_j = +\infty$.) Then recursively search the subtree p_i. When you arrive at a leaf, search all the keys in this node. If it is not here, then it is not in the B-tree.

Insertion: 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 the leaf is not at full capacity (it has fewer than $m-1$ keys) then we simply insert it and are done. (Note that when m is large this is the typical case.) Note that this will involve sliding keys around within the leaf node to make room for the new entry. Since m is assumed to be a constant, this constant time overhead is ignored.

Otherwise the node *overflows* and we have to take action to restore balance. There are two methods for restoring order in a B-tree.

Key rotation: In spite of this name, this is *not* the same as rotation in AVL trees. This method is quick and involves little data movement. Unfortunately, it is not always applicable. Let's let p be the node that overflows. We look at our immediate left and right siblings in the B-tree. (Note that one or the other may not exist). Suppose that one of them, the right sibling say, has fewer than the maximum of $m-1$ keys. Let q denote this sibling, and let k denote the key in the parent node that splits the elements in q from those in p.

We take the key k from the parent and make it the minimum key in q. We take the maximum key from p and use it to replace k in the parent node. Finally we transfer the rightmost subtree from p to become the leftmost subtree of q (and readjust all the pointers accordingly). At this point the tree is balanced and no further rebalancing is needed. An example of this is shown in the following figure. We have shown all the child links (even at the leaf level where they would not normally be used) to make clearer how the update works at any level of the tree. Key rotation to the right child is analogous.

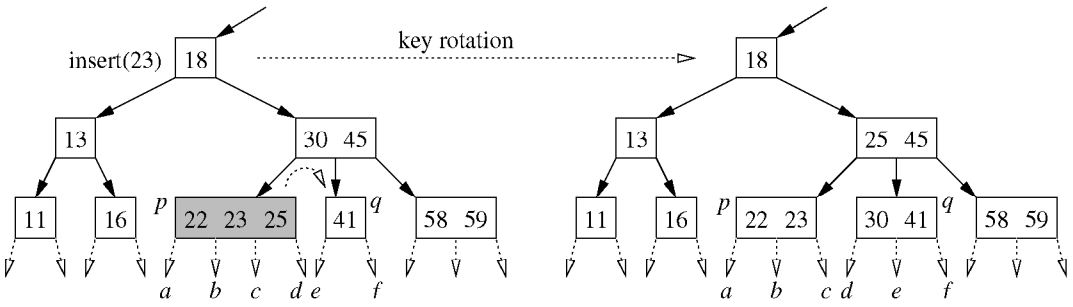

Figure 37: Key rotation.

Node split: If both siblings are full, then key rotation is not possible. Recall that a node can have at most $m-1$ keys and at least $\lceil m/2 \rceil - 1$ keys. Suppose after inserting the new key we have an overfull node with m keys, $k_1 < k_2 < \ldots < k_m$. We split this node into three groups: one with the smallest $\lceil (m-1)/2 \rceil$ keys, a single central element, and one with the largest $\lfloor (m-1)/2 \rfloor$ keys. We copy the smallest group to a new B-tree node, and leave the largest in the existing node. We take the extra key and insert it into the parent.

At this point the parent may overflow, and so we repeat this process recursively. When the root overflows, we split the root into two nodes, and create a new root with two children. (This is why the root is exempt from the normal number of children requirement.) See the figure below for an example. This shows an interesting feature of B-trees, namely that they seem to grow from the root up, rather than from the leaves down.

To see that the new nodes are of proper size it suffices to show that

$$\left\lceil \frac{m}{2} \right\rceil - 1 \;\leq\; \left\lfloor \frac{m-1}{2} \right\rfloor \;\leq\; \left\lceil \frac{m-1}{2} \right\rceil \;\leq\; m.$$

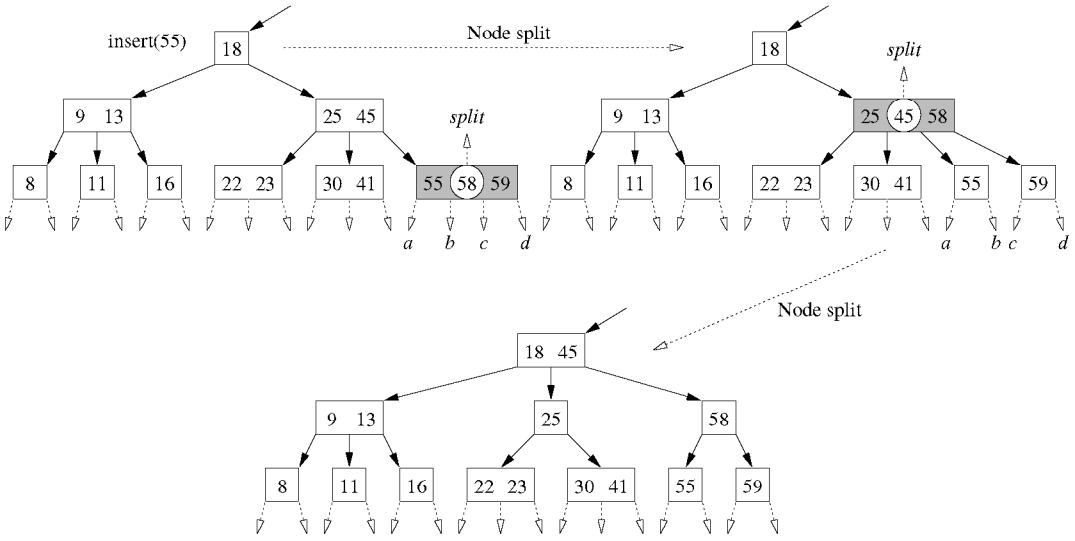

Figure 38: Node split.

The upper bound ($\leq m$) holds for any integer m. To see the lower bound, we observe that if m is odd then

$$\left\lceil \frac{m}{2} \right\rceil - 1 \;=\; \frac{m+1}{2} - 1 \;=\; \frac{m-1}{2} \;=\; \left\lfloor \frac{m-1}{2} \right\rfloor.$$

If m is even then

$$\left\lceil \frac{m}{2} \right\rceil - 1 \;=\; \frac{m}{2} - 1 \;=\; \frac{m-2}{2} \;=\; \left\lfloor \frac{m-1}{2} \right\rfloor.$$

So we actually have equality in each case.

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 need restore the size constraints.

Suppose that after deletion the current node has $\lceil m/2 \rceil - 2$ keys. 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 rotate the extra into this node, and we are done.

Otherwise we know that at least one of our neighbors (say the left) has exactly $\lceil m/2 \rceil - 1$ keys. In this case we need to perform a *node merge*. This is the inverse of a node split. We combine the contents of the current node, together with the contents of its sibling, together with the key from the parent that separates these two. The resulting node has

$$\left(\left\lceil \frac{m}{2} \right\rceil - 2\right) + \left(\left\lceil \frac{m}{2} \right\rceil - 1\right) + 1$$

total keys. We will leave it as an exercise to prove that this is never greater than $m - 1$.

Note that the removal of a key from the parent's node may cause it to underflow. Thus, the process needs to be repeated recursively. In the worst case this continues up to the root. If the root looses its only key then we simply remove the root, and make its root's only child the new root of the B-tree.

An example of deletion of 16 where $m = 3$ (meaning each node has from 1 to 2 keys) is shown in the figure below. (As before, we have shown all the child links, even at the leaf level, where they are irrelevant in order to make clearer how the operation works at all levels.)

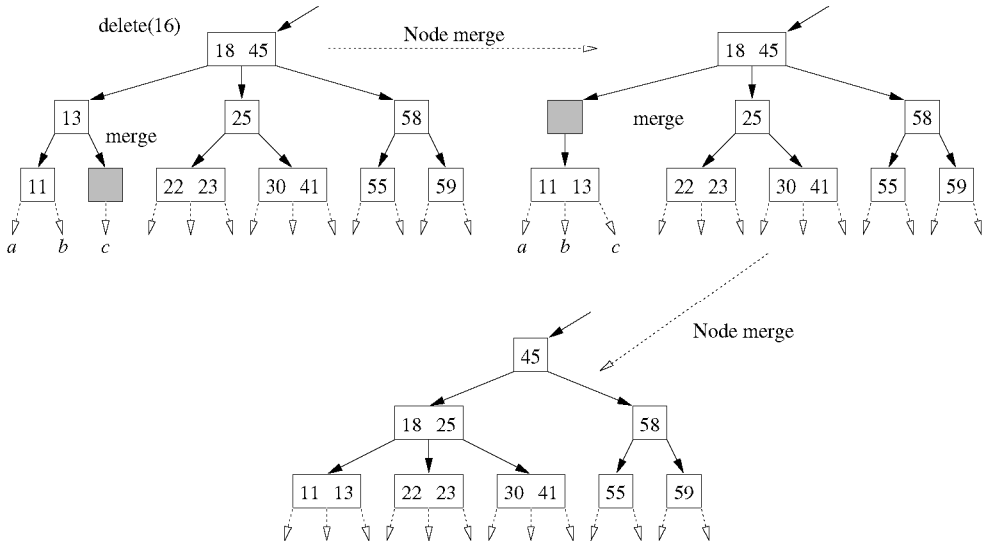

Figure 39: Deletion of key 16.