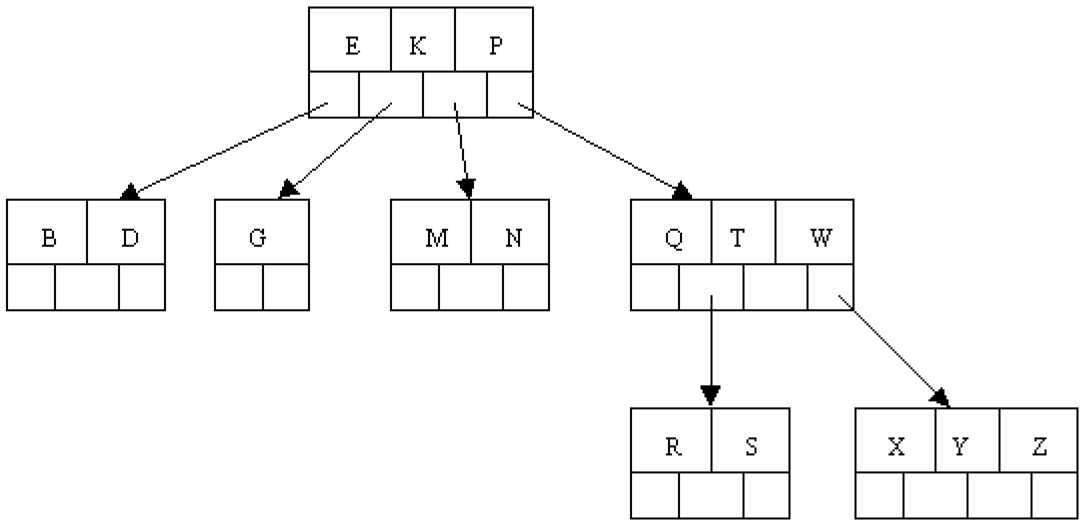
B-Trees

Introduction

A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fairly large amount of data at once (perhaps 1024 bytes).

Definitions

A multiway tree of order m is an ordered tree where each node has at most m children. For each node, if k is the actual number of childen in the node, then k - 1 is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is called a multiway search tree of order m. For example, the following is a multiway search tree of order 4. Note that the first row in each node shows the keys, while the second row shows the pointers to the child nodes. Of course, in any useful application there would be a record of data associated with each key, so that the first row in each node might be an array of records where each record contains a key and its associated data. Another approach is to have the first row of each node contain an array of records where each record contains a key and a record number for the associated data record, which is found in another file. This last method is often used when the data records are large. The example software will use the first method.



What does it mean to say that the keys and subtrees are "arranged in the fashion of a search tree"?

Then a multiway search tree of order 4 has to fulfill the following conditions related to the ordering of the keys:

- The keys in each node are in ascending order.
- At every given node (call it Node) the following is true:
  - The subtree starting at record Node.Branch[0] has only keys that are less than Node.Key[0].
  - The subtree starting at record Node.Branch[1] has only keys that are greater than Node.Key[0] and

  at the same time less than Node.Key[1].
  - The subtree starting at record Node.Branch[2] has only keys that are greater than Node.Key[1] and at the same time less than Node.Key[2].
  - The subtree starting at record Node.Branch[3] has only keys that are greater than Node.Key[2].
- Note that if less than the full number of keys are in the Node, these 4 conditions are truncated so that they speak of the appropriate number of keys and branches.

This generalizes in the obvious way to multiway search trees with other orders.

A B-tree of order m is a multiway search tree of order m such that:

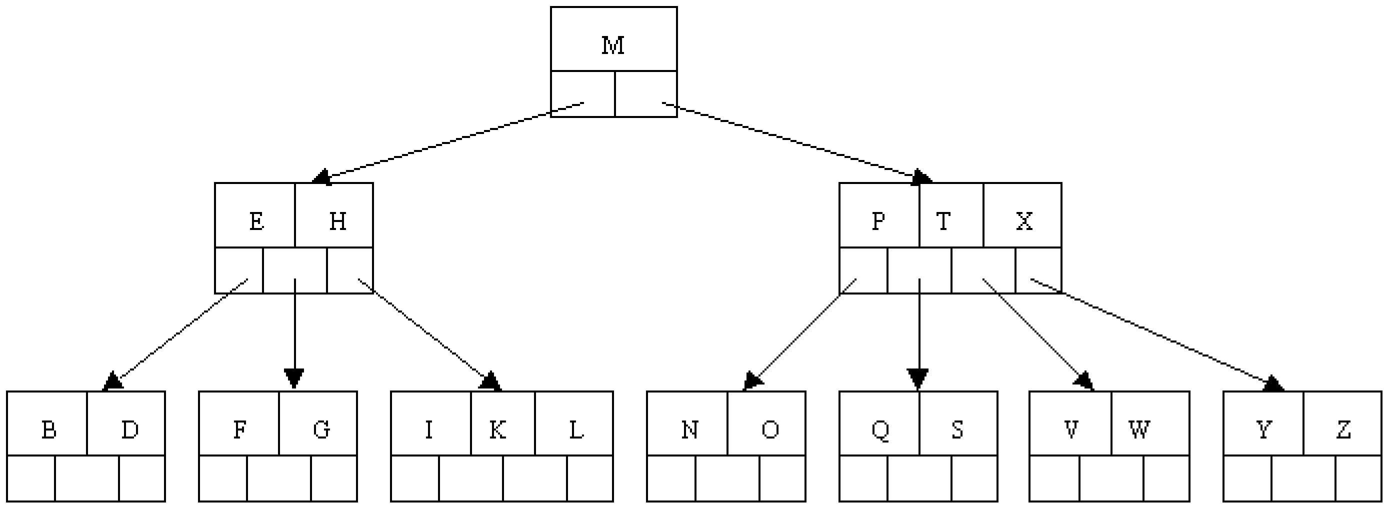
- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least ceil(m / 2) (nonempty) children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least ceil(m / 2) - 1 keys.

Note that ceil(x) is the so-called ceiling function. It's value is the smallest integer that is greater than or equal to x. Thus ceil(3) = 3, ceil(3.35) = 4, ceil(1.98) = 2, ceil(5.01) = 6, ceil(7) = 7, etc.

A B-tree is a fairly well-balanced tree by virtue of the fact that all leaf nodes must be at the bottom. Condition (2) tries to keep the tree fairly bushy by insisting that each node have at least half the maximum number of children. This causes the tree to "fan out" so that the path from root to leaf is very short even in a tree that contains a lot of data.

Example B-Tree

The following is an example of a B-tree of order 5. This means that (other that the root node) all internal nodes have at least ceil(5 / 2) = ceil(2.5) = 3 children (and hence at least 2 keys). Of course, the maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys). According to condition 4, each leaf node must contain at least 2 keys. In practice B-trees usually have orders a lot bigger than 5.



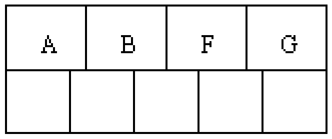
Operations on a B-Tree

Question: How would you search in the above tree to look up S? How about J? How would you do a sort-of "in-order" traversal, that is, a traversal that would produce the letters in ascending order? (One would only do such a traversal on rare occasion as it would require a large amount of disk activity and thus be very slow!)

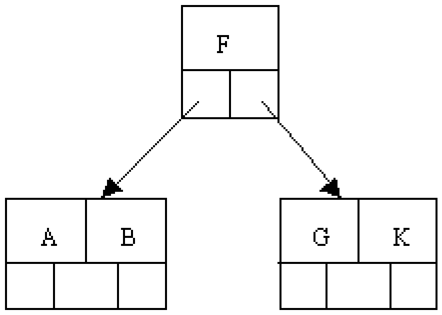
Inserting a New Item

The insertion algorithm proceeds as follows: When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.) Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

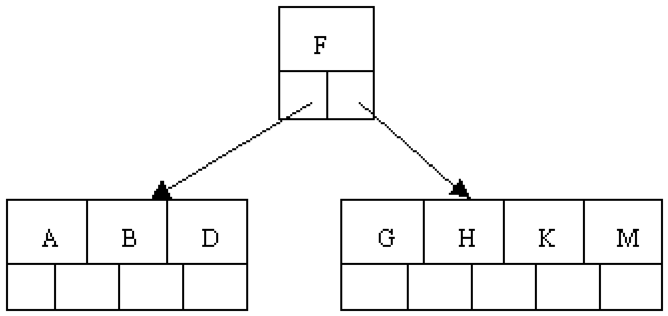
An example: Insert the following letters into what is originally an empty B-tree of order 5: A G F B K D H M J E S I R X C L N T U P. Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



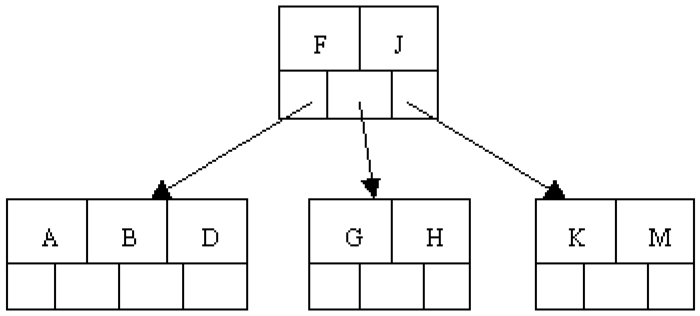
When we try to insert the K, we find no room in this node, so we split it into 2 nodes, moving the median item F up into a new root node. Note that in practice we just leave the A and B in the current node and place the G and K into a new node to the right of the old one.



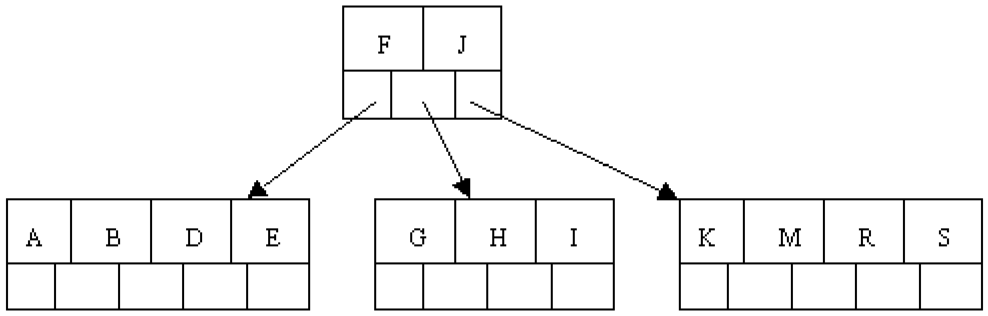
Inserting D, H, and M proceeds without requiring any splits:



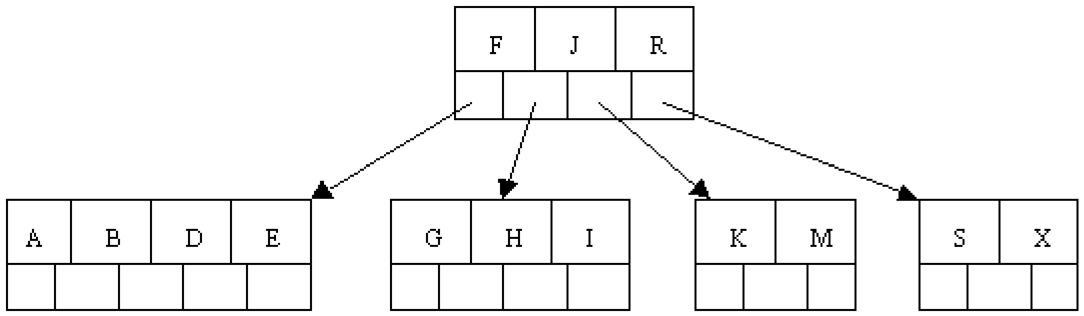
Inserting J requires a split. Note that J happens to be the median key and so is moved up into the parent node.



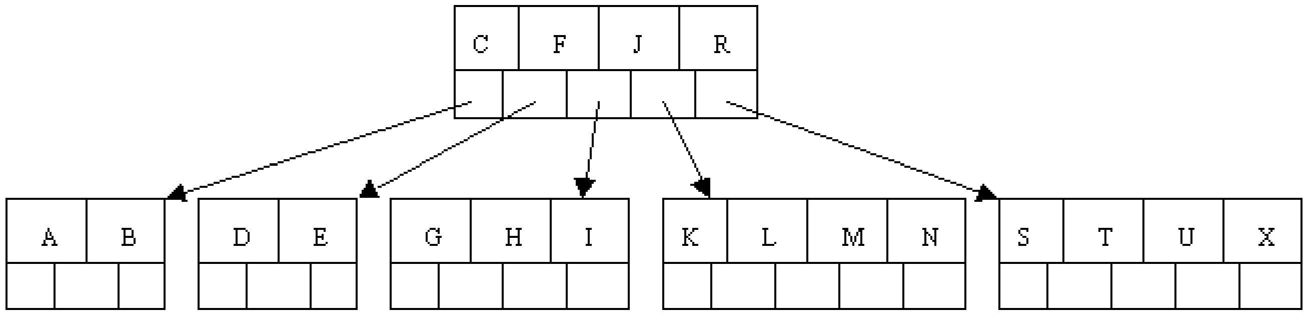
The letters E, S, I, and R are then added without needing any split.



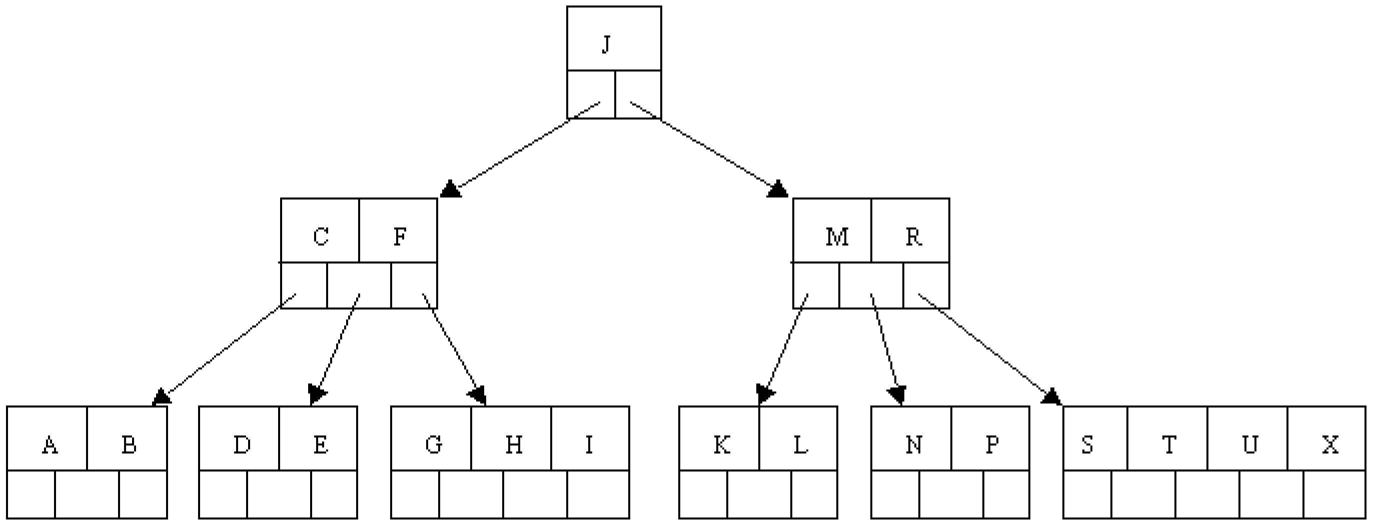
When X is added, the rightmost leaf must be split. The median item R is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



The insertion of C causes the leftmost leaf to be split. C happens to be the median key and so is the one moved up into the parent node. The letters L, N, T, and U are then added without any need of splitting:

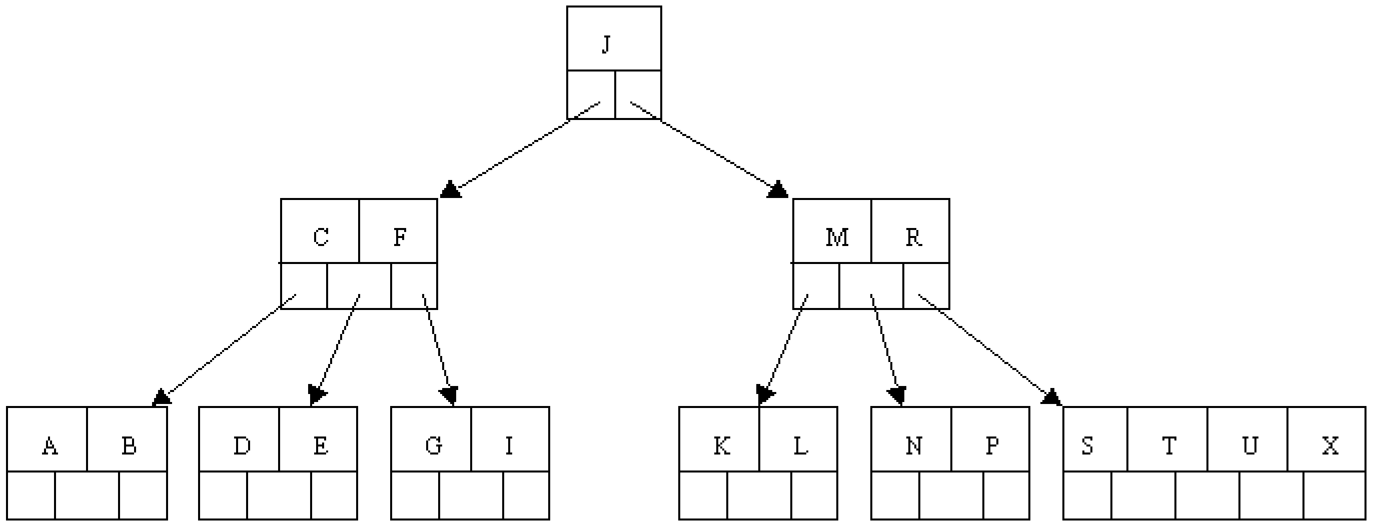


Finally, when P is added, the node with K, L, M, and N splits sending the median M up to the parent. However, the parent node is full, so it splits, sending the median J up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains C and F.

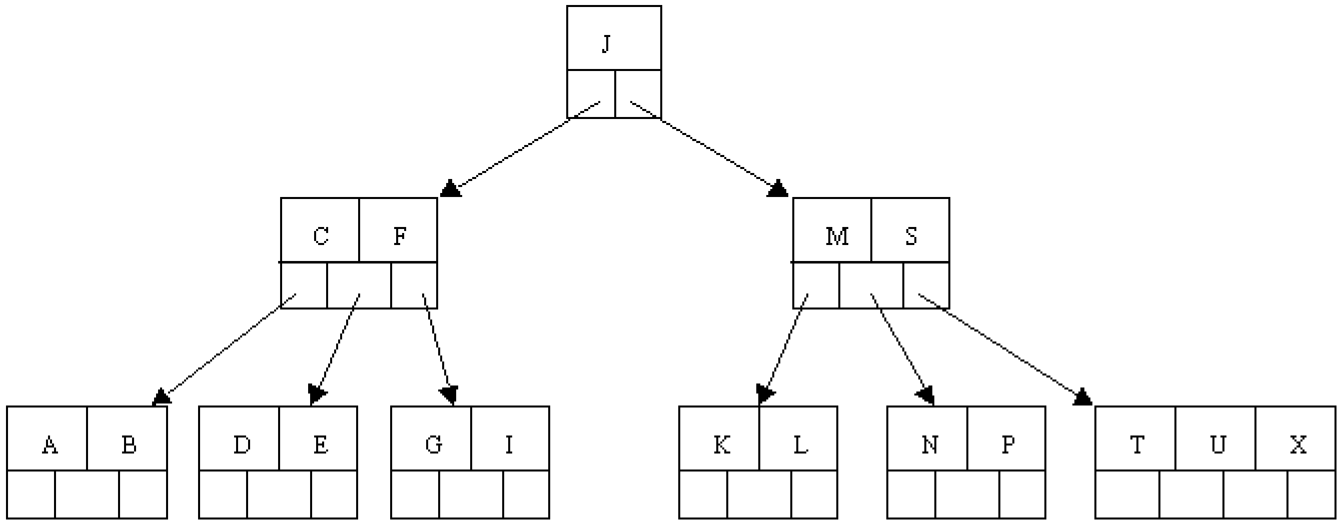


Deleting an Item

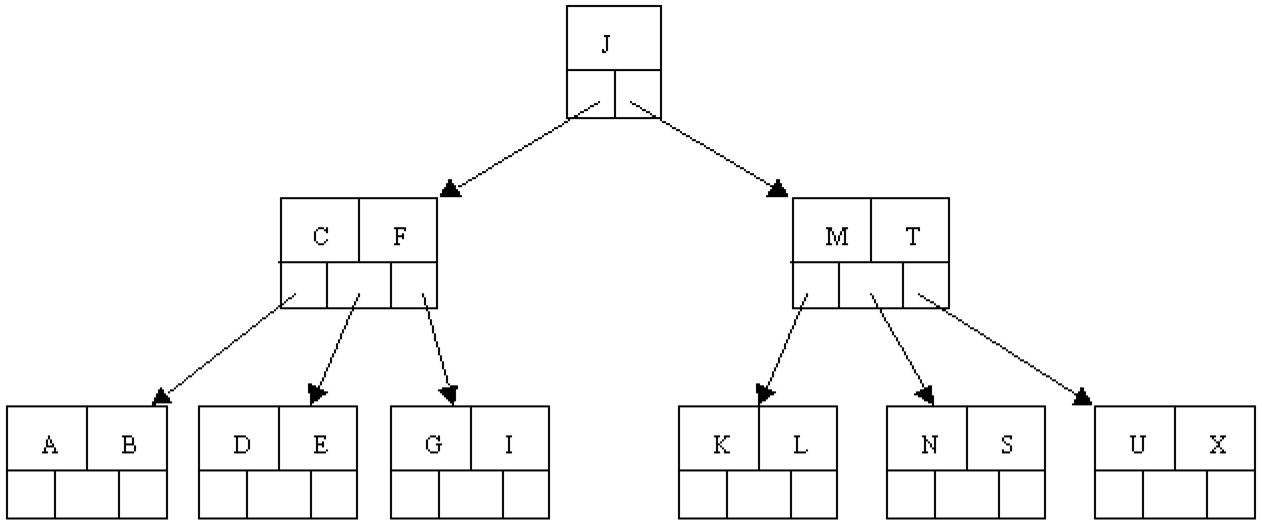
Let's again follow an example. In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the I over where the H had been. This gives:



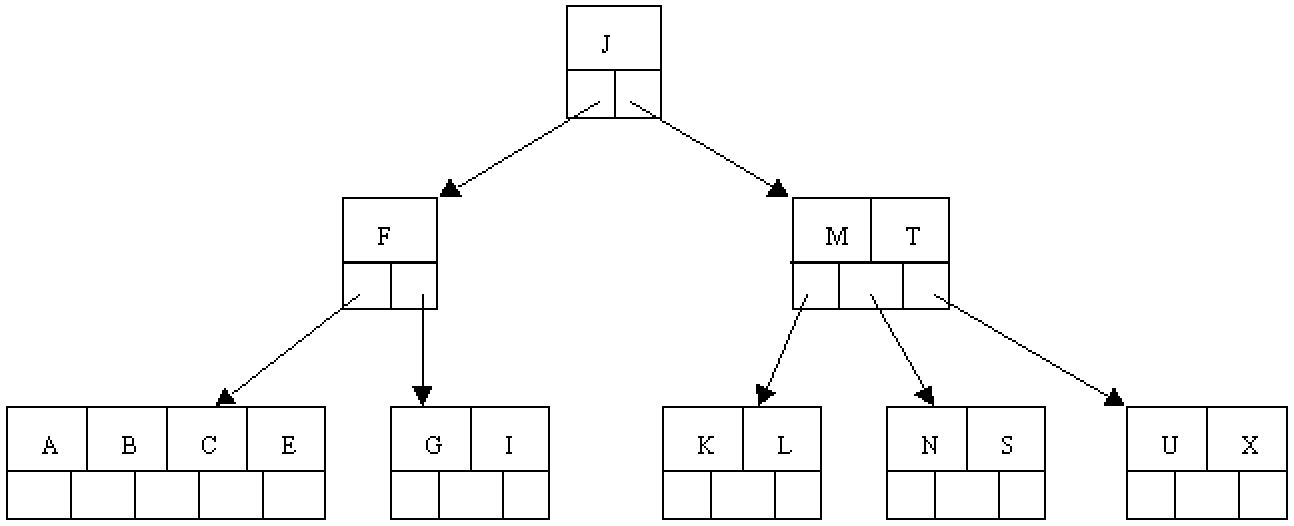
Next, delete the R. Since R is not in a leaf, we find its successor (the next item in ascending order), which happens to be S, and move S up to replace the R. That way, what we really have to do is to delete S from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method.



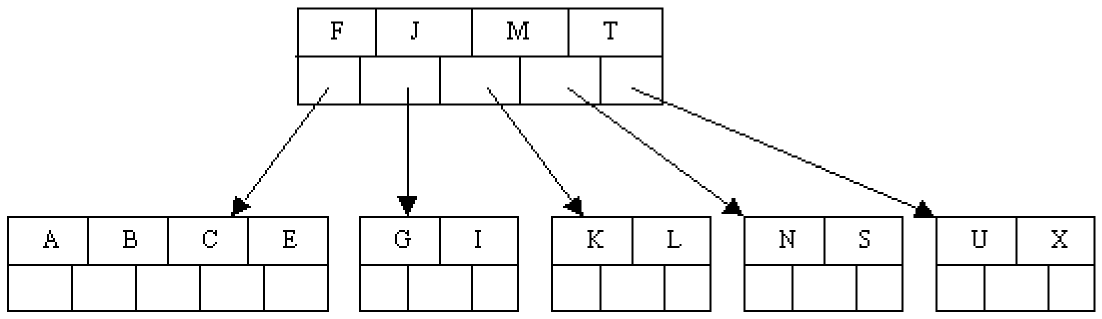
Next, delete P. Although P is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor of P, which is S, is moved down from the parent, and the T is moved up.



Finally, let's delete D. This one causes lots of problems. Although D is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing E with the leaf containing A B. We also move down the C.

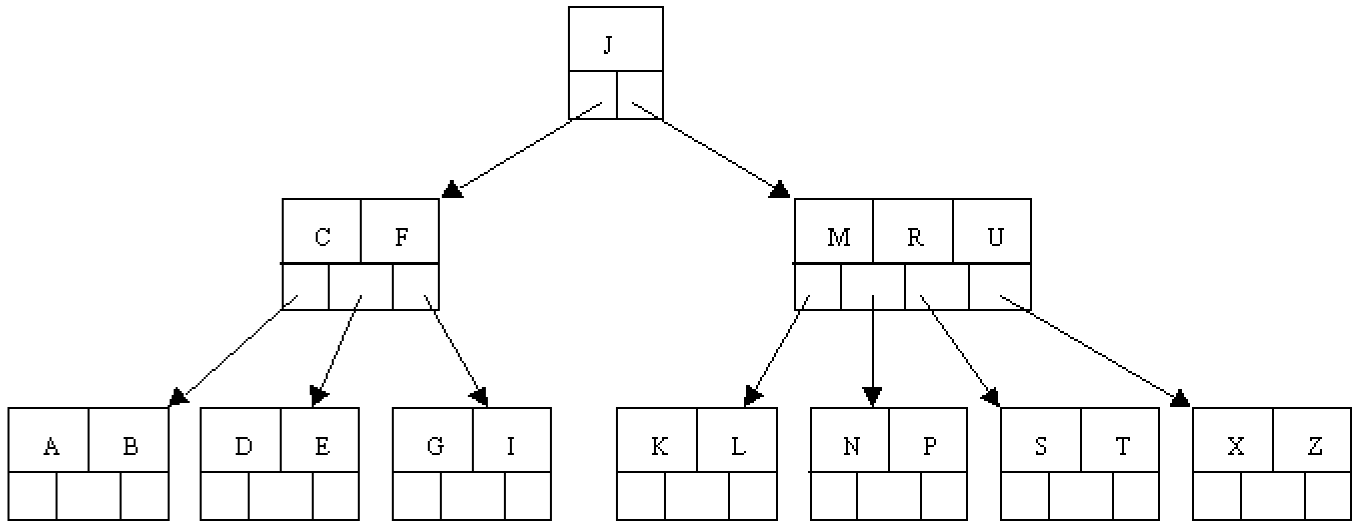


Of course, you immediately see that the parent node now contains only one key, F. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with M T) had one more key in it. We would then move J down to the node with too few keys and move the M up where the J had been. However, the left subtree of M would then have to become the right subtree of J. In other words, the K L node would be attached via the pointer field to the right of J's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the J from the parent. In this case, the tree shrinks in height by one.

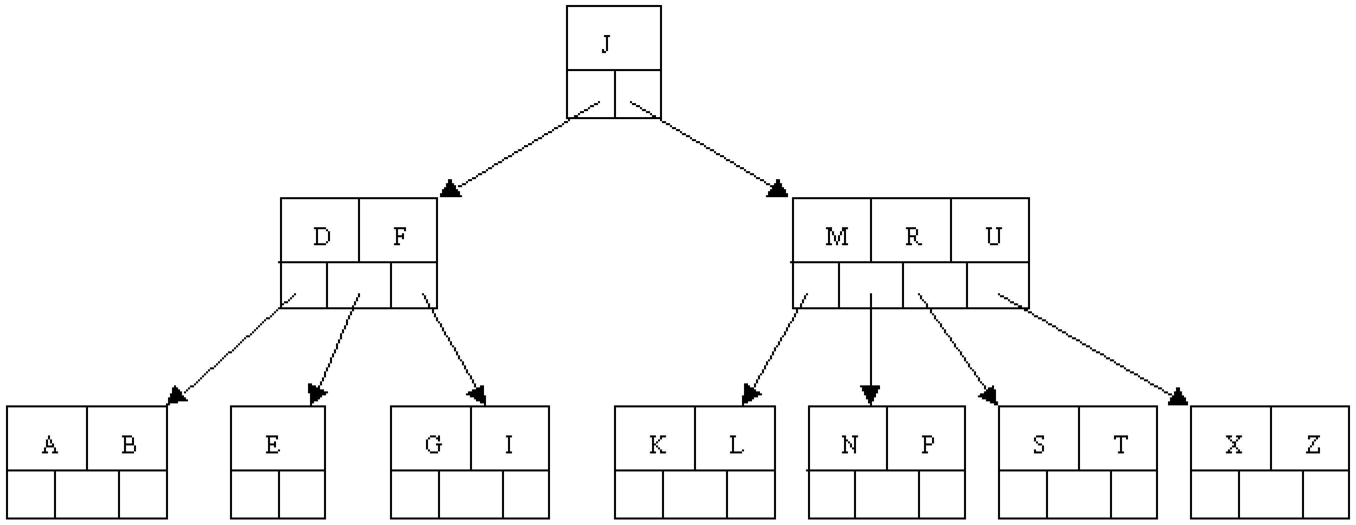


Another Example

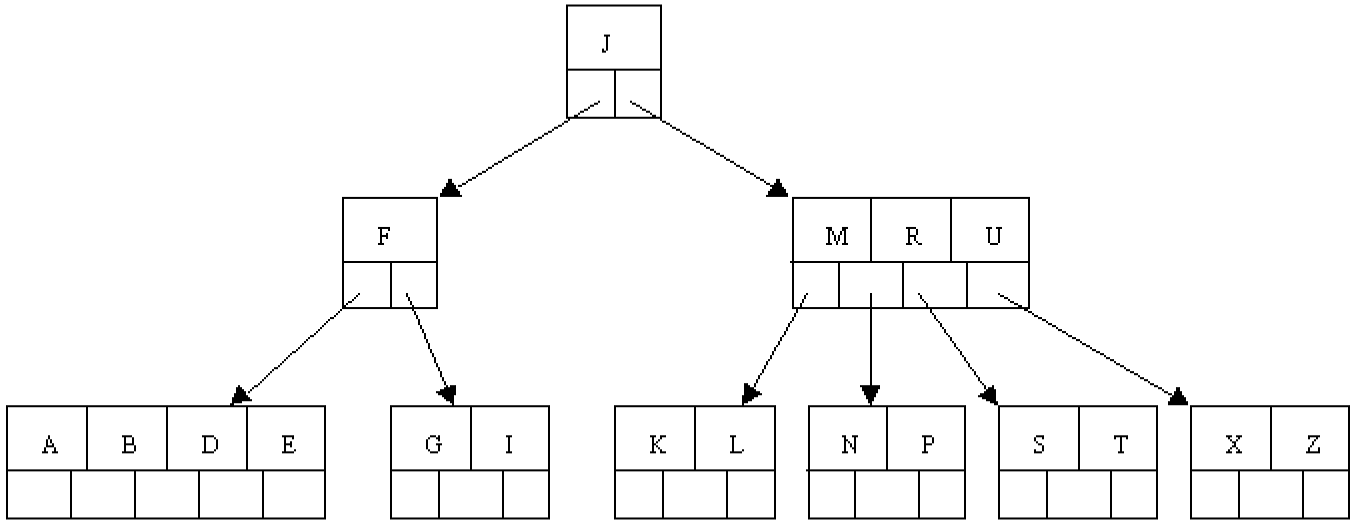
Here is a different B-tree of order 5. Let's try to delete C from it.



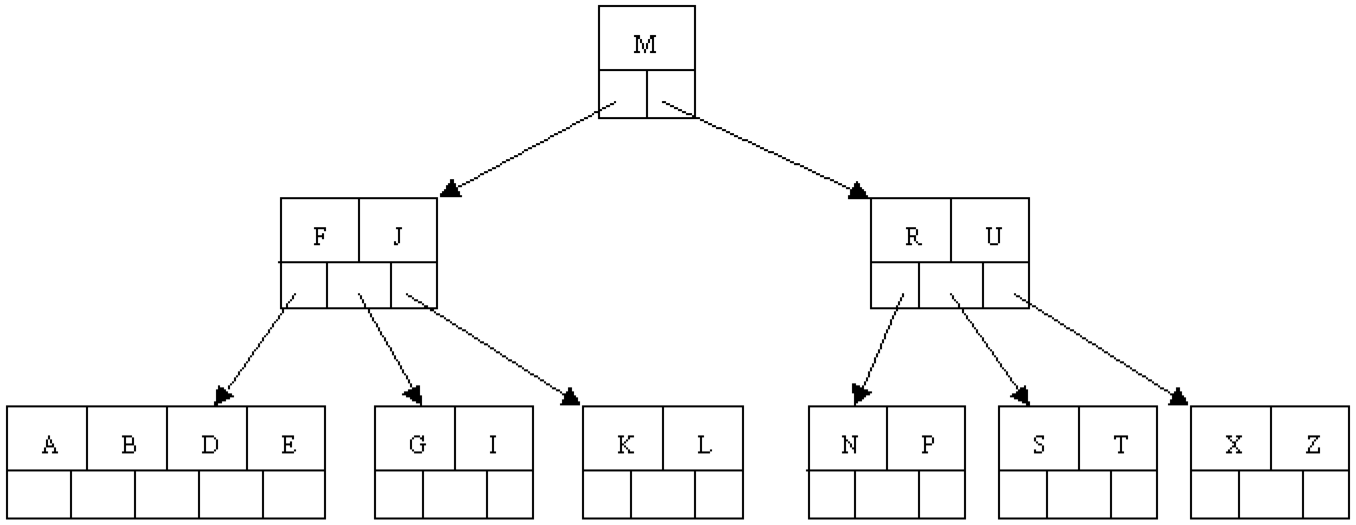
We begin by finding the immediate succesor, which would be D, and move the D up to replace the C. However, this leaves us with a node with too few keys.



Since neither the sibling to the left or right of the node containing E has an extra key, we must combine the node with one of these two siblings. Let's consolidate with the A B node.



But now the node containing F does not have enough keys. However, its sibling has an extra key. Thus we borrow the M from the sibling, move it up to the parent, and bring the J down to join the F. Note that the K L node gets reattached to the right of the J.



Variations

The B-tree and variations on it are commonly used in large commercial databases to provide quick access to the data. In fact, he says that they are "the standard file organization for applications requiring insertion, deletion, and key range searches". The variant called the B+ tree is the usual one. Another variant is the B* tree, which is very similar to the B+ tree, but tries to keep the nodes about two-thirds full at a minimum.

In a B+ tree, data records are only stored in the leaves. Internal nodes store just keys. These are used for directing a search to the proper leaf. If a target key is less than a key in an internal node, then the pointer just to its left is followed. If a target key is greater or equal to the key in the internal node, then the pointer just to its right is followed. The leaves are also linked together so that all of the data records in the B+ tree can be traversed in ascending order, just by going through all of the nodes in this linked list along the bottom level of the tree.

As an example, consider a B+ tree of order 100, whose leaves can contain up to 100 records. A 2 level B+ tree can store up to 10,000 records. A 3 level B+ tree can store up to 1 million records. A 4 level B+ tree can store up to 100 million records. To get faster access to the data, the root node is commonly kept in main memory. Maybe even the child nodes of the root can fit in main memory. Thus one can find one of 100 million records with only 2 or 3 disk reads.

Note that a B-tree (or B+ tree, etc.) can be used as an index file. The actual data records are stored elsewhere on disk. When one looks up and finds a target key, what one finds is the target key and an associated pointer (disk block number or whatever) to the appropriate data record. This means that one data file could have several such indices, each giving an ordering by a different key field, something highly desirable to have.

References:

Data Structures & Program Design, 2nd ed. Robert L. Kruse. Prentice-Hall (1987). There is also a third edition (1994) as well as a new text, Data Structures and Program Design in C++, authored by Kruse and Alexander J. Ryba (1999).
A Practical Introduction to Data Structures and Algorithm Analysis Clifford A. Shaffer. Prentice-Hall (1997).

Adapted from: http://cis.stvincent.edu/carlsond/swdesign/btree/btree.html