12.6. B-Trees

12.6.1. B-Trees

This module presents the B-tree. B-trees are usually attributed to R. Bayer and E. McCreight who described the B-tree in a 1972 paper. By 1979, B-trees had replaced virtually all large-file access methods other than hashing. B-trees, or some variant of B-trees, are *the* standard file organization for applications requiring insertion, deletion, and key range searches. They are used to implement most modern file systems. B-trees address effectively all of the major problems encountered when implementing disk-based search trees:

1. The B-tree is shallow, in part because the tree is always height balanced (all leaf nodes are at the same level), and in part because the branching factor is quite high. So only a small number of disk blocks are accessed to reach a given record.

2. Update and search operations affect only those disk blocks on the path from the root to the leaf node containing the query record. The fewer the number of disk blocks affected during an operation, the less disk I/O is required.

3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on range searches.

4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

A B-tree of order m is defined to have the following shape properties:

The root is either a leaf or has at least two children.

Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.

All leaves are at the same level in the tree, so the tree is always height balanced.

The B-tree is a generalization of the 2-3 tree. Put another way, a 2-3 tree is a B-tree of order three. Normally, the size of a node in the B-tree is chosen to fill a disk block. A B-tree node implementation typically allows 100 or more children. Thus, a B-tree node is equivalent to a disk block, and a "pointer" value stored in the tree is actually the number of the block containing the child node (usually interpreted as an offset from the beginning of the corresponding disk file). In a typical application, the B-tree's access to the disk file will be managed using a **buffer pool** and a block-replacement scheme such as **LRU**.

Figure **12.6.1** shows a B-tree of order four. Each node contains up to three keys, and internal nodes have up to four children.

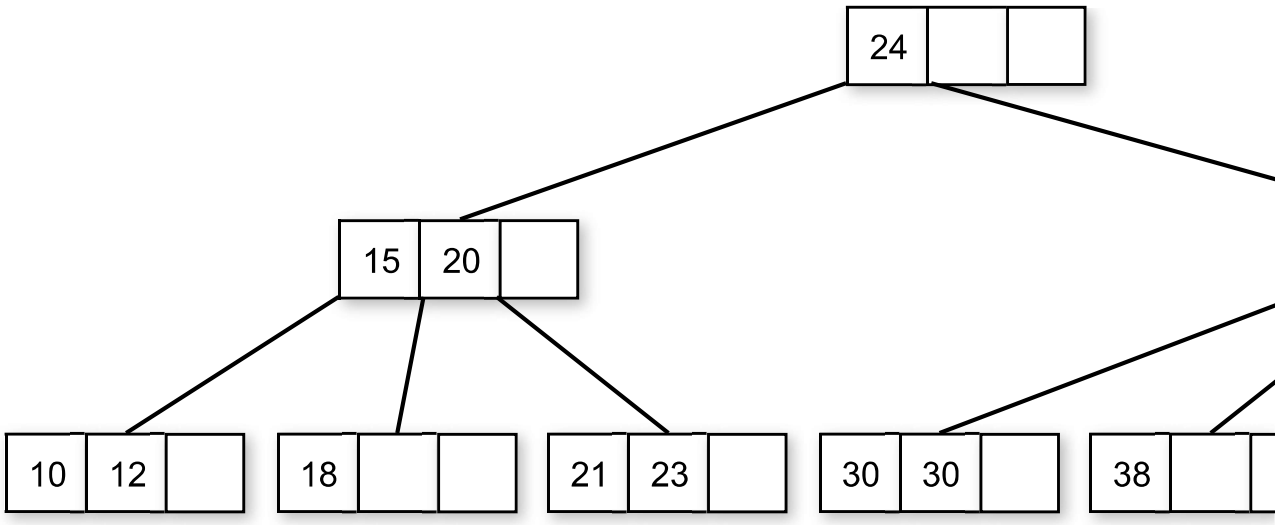


Figure 12.6.1: A B-tree of order four.

Search in a B-tree is a generalization of search in a 2-3 tree. It is an alternating two-step process, beginning with the root node of the B-tree.

1. Perform a binary search on the records in the current node. If a record with the search key is found, then return that record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.

2. Otherwise, follow the proper branch and repeat the process.

For example, consider a search for the record with key value 47 in the tree of Figure **12.6.1**. The root node is examined and the second (right) branch taken. After

examining the node at level 1, the third branch is taken to the next level to arrive at the leaf node containing a record with key value 47.

B-tree insertion is a generalization of 2-3 tree insertion. The first step is to find the leaf node that should contain the key to be inserted, space permitting. If there is room in this node, then insert the key. If there is not, then split the node into two and promote the middle key to the parent. If the parent becomes full, then it is split in turn, and its middle key promoted.

Note that this insertion process is guaranteed to keep all nodes at least half full. For example, when we attempt to insert into a full internal node of a B-tree of order four, there will now be five children that must be dealt with. The node is split into two nodes containing two keys each, thus retaining the B-tree property. The middle of the five children is promoted to its parent.

12.6.1.1. B+ Trees

The previous section mentioned that B-trees are universally used to implement large-scale disk-based systems. Actually, the B-tree as described in the previous section is almost never implemented. What is most commonly implemented is a variant of the B-tree, called the B^+ tree. When greater efficiency is required, a more complicated variant known as the B^* tree is used.

Consider again the **linear index**. When the collection of records will not change, a linear index provides an extremely efficient way to search. The problem is how to handle those pesky inserts and deletes. We could try to keep the core idea of storing a sorted array-based list, but make it more flexible by breaking the list into manageable chunks that are more easily updated. How might we do that? First, we need to decide how big the chunks should be. Since the data are on disk, it seems reasonable to store a chunk that is the size of a disk block, or a small multiple of the disk block size. If the next record to be inserted belongs to a chunk that hasn't filled its block then we can just insert it there. The fact that this might cause other records in that chunk to move a little bit in the array is not important, since this does not cause any extra disk accesses so long as we move data within that chunk. But what if the chunk fills up the entire

block that contains it? We could just split it in half. What if we want to delete a record? We could just take the deleted record out of the chunk, but we might not want a lot of near-empty chunks. So we could put adjacent chunks together if they have only a small amount of data between them. Or we could shuffle data between adjacent chunks that together contain more data. The big problem would be how to find the desired chunk when processing a record with a given key. Perhaps some sort of tree-like structure could be used to locate the appropriate chunk. These ideas are exactly what motivate the B^+ tree. The B^+ tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.

The most significant difference between the B^+ tree and the BST or the standard B-tree is that the B^+ tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search. This means that internal nodes are significantly different in structure from leaf nodes. Internal nodes store keys to guide the search, associating each key with a pointer to a child B^+ tree node. Leaf nodes store actual records, or else keys and pointers to actual records in a separate disk file if the B^+ tree is being used purely as an index. Depending on the size of a record as compared to the size of a key, a leaf node in a B^+ tree of order m might have enough room to store more or less than m records. The requirement is simply that the leaf nodes store enough records to remain at least half full. The leaf nodes of a B^+ tree are normally linked together to form a doubly linked list. Thus, the entire collection of records can be traversed in sorted order by visiting all the leaf nodes on the linked list. Here is a Java-like pseudocode representation for the B^+ tree node interface. Leaf node and internal node subclasses would implement this interface.

```
/** Interface for B+ Tree nodes */
public interface BPNode<Key,E> {
  public boolean isLeaf();
  public int numrecs();
  public Key[] keys();
}
```

An important implementation detail to note is that while Figure **12.6.1** shows internal nodes containing three keys and four pointers, class `BPNode` is slightly different in that

it stores key/pointer pairs. Figure **12.6.1** shows the B^+ tree as it is traditionally drawn. To simplify implementation in practice, nodes really do associate a key with each pointer. Each internal node should be assumed to hold in the leftmost position an additional key that is less than or equal to any possible key value in the node's leftmost subtree. B^+ tree implementations typically store an additional dummy record in the leftmost leaf node whose key value is less than any legal key value.

Let's see in some detail how the simplest B^+ tree works. This would be the "$2-3^+$ tree", or a B^+ tree of order 3.

1 / 28 << < >

Example 2-3+ Tree Visualization: Insert

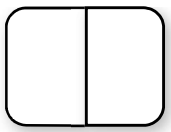


Figure 12.6.2: An example of building a $2-3^+$ tree

Next, let's see how to search.

1 / 10

≪ < >

Example 2-3+ Tree Visualization: Search

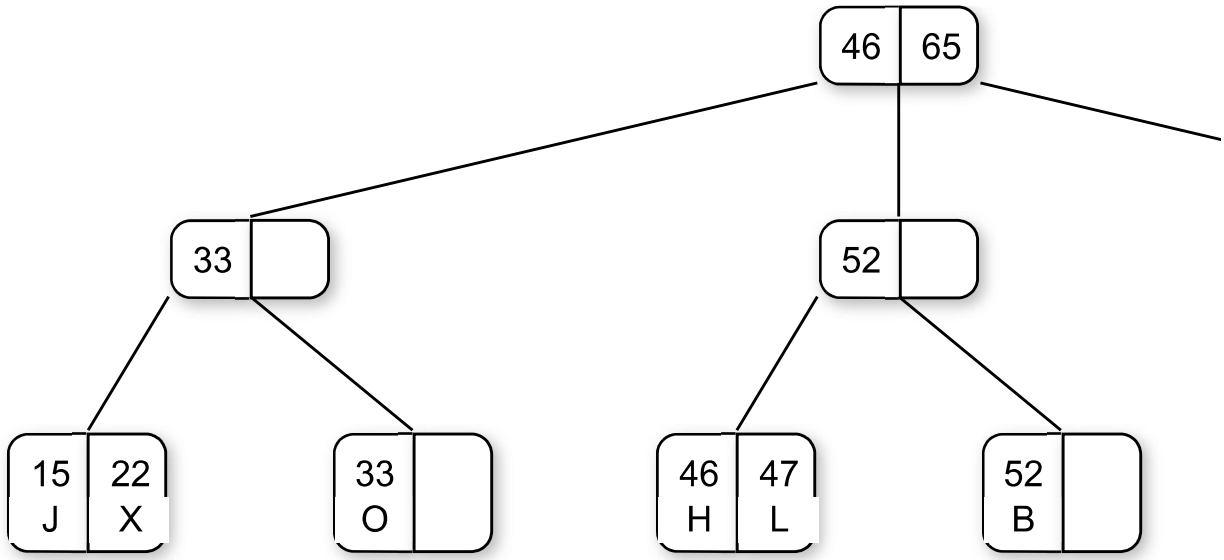


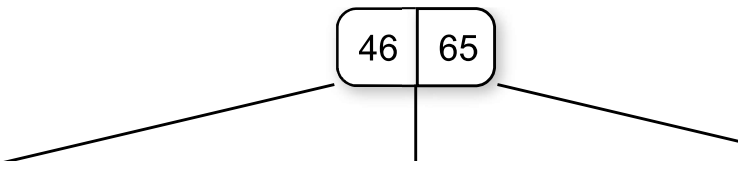
Figure 12.6.3: An example of searching a $2-3^+$ tree

Finally, let's see an example of deleting from the $2-3^+$ tree

1 / 33

≪ < >

Example 2-3+ Tree Visualization: Delete



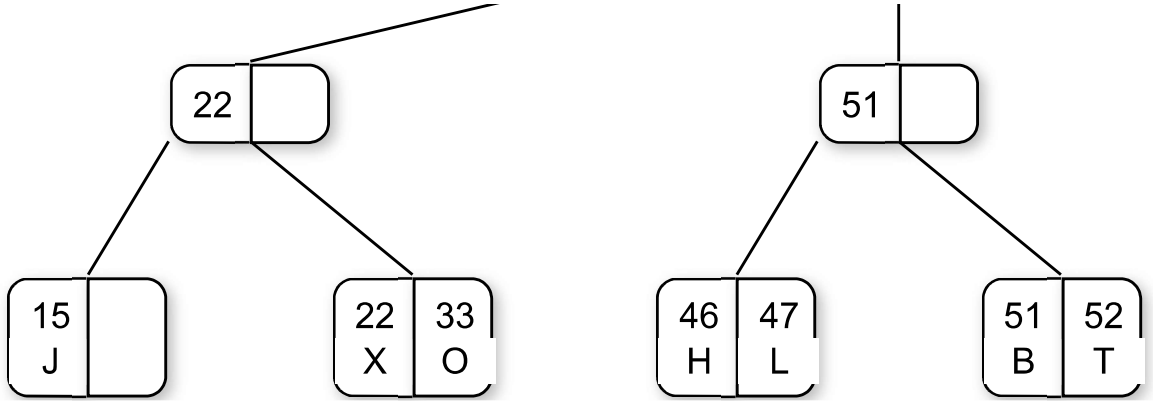


Figure 12.6.4: An example of deleting from a $2-3^+$ tree

Now, let's extend these ideas to a B^+ tree of higher order.

B^+ trees are exceptionally good for range queries. Once the first record in the range has been found, the rest of the records with keys in the range can be accessed by sequential processing of the remaining records in the first node, and then continuing down the linked list of leaf nodes as far as necessary. Figure illustrates the B^+ tree.

1 / 10 ≪ < >

Example B+ Tree Visualization: Search in a tree of degree 4

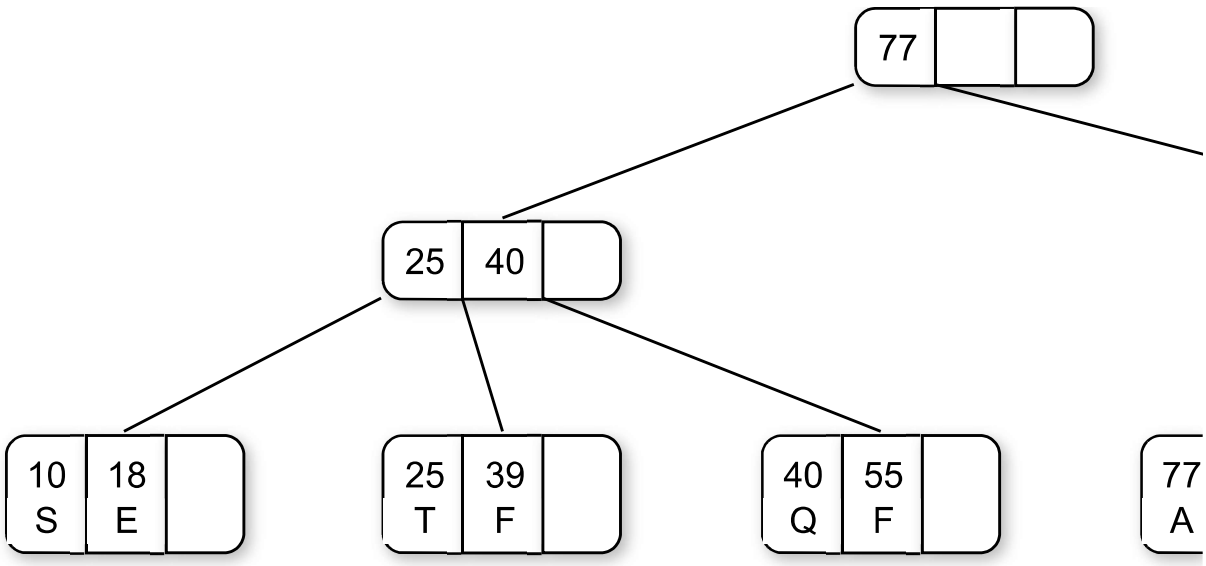


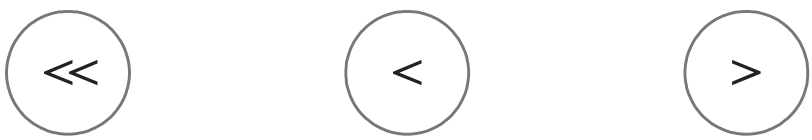
Figure 12.6.5: An example of search in a B+ tree of order four. Internal nodes must store between two and four children.

Search in a B^+ tree is nearly identical to search in a regular B-tree, except that the search must always continue to the proper leaf node. Even if the search-key value is found in an internal node, this is only a placeholder and does not provide access to the actual record. Here is a pseudocode sketch of the B^+ tree search algorithm.

```
private E findhelp(BPNode<Key,E> rt, Key k) {
  int currec = binaryle(rt.keys(), rt.numrecs(), k);
  if (rt.isLeaf()) {
    if ((((BPLeaf<Key,E>)rt).keys())[currec] == k) {
      return ((BPLeaf<Key,E>)rt).recs(currec);
    }
    else { return null; }
  }
  else{
    return findhelp(((BPInternal<Key,E>)rt).pointers(currec), k);
  }
}
```

B^+ tree insertion is similar to B-tree insertion. First, the leaf L that should contain the record is found. If L is not full, then the new record is added, and no other B^+ tree nodes are affected. If L is already full, split it in two (dividing the records evenly among the two nodes) and promote a copy of the least-valued key in the newly formed right node. As with the 2-3 tree, promotion might cause the parent to split in turn, perhaps eventually leading to splitting the root and causing the B^+ tree to gain a new level. B^+ tree insertion keeps all leaf nodes at equal depth. Figure illustrates the insertion process through several examples.

1 / 42



Example B+ Tree Visualization: Insert into a tree of degree 4

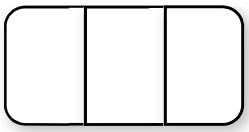


Figure 12.6.6: An example of building a B+ tree of order four.

Here is a a Java-like pseudocode sketch of the B^+ tree insert algorithm.

```
private BPNode<Key,E> inserthelp(BPNode<Key,E> rt,
                                 Key k, E e) {
  BPNode<Key,E> retval;
  if (rt.isLeaf()) { // At leaf node: insert here
    return ((BPLeaf<Key,E>)rt).add(k, e);
  }
  // Add to internal node
  int currec = binaryle(rt.keys(), rt.numrecs(), k);
  BPNode<Key,E> temp = inserthelp(
         ((BPInternal<Key,E>)root).pointers(currec), k, e);
  if (temp != ((BPInternal<Key,E>)rt).pointers(currec)) {
    return ((BPInternal<Key,E>)rt).
               add((BPInternal<Key,E>)temp);
  }
  else{
    return rt;
```

```
    }
  }
```

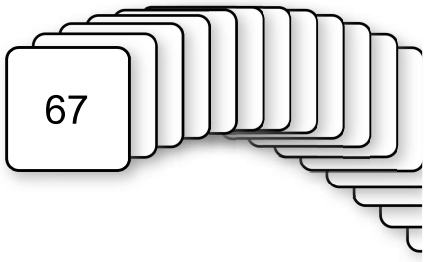
Here is an exercise to see if you get the basic idea of B^+ tree insertion.

B^+ Tree Insertion

Instructions:

In this exercise your job is to insert the values from the stack to the B^+ tree.
Search for the leaf node where the topmost value of the stack should be inserted, and click on that node. The exercise will take care of the rest.
Continue this procedure until you have inserted all the values in the stack.

Undo | Reset | Model Answer | Grade



To delete record R from the B^+ tree, first locate the leaf L that contains R. If L is more than half full, then we need only remove R, leaving L still at least half full. This is demonstrated by Figure .

1 / 23 ≪ < >

Example B+ Tree Visualization: Delete from a tree of degree 4

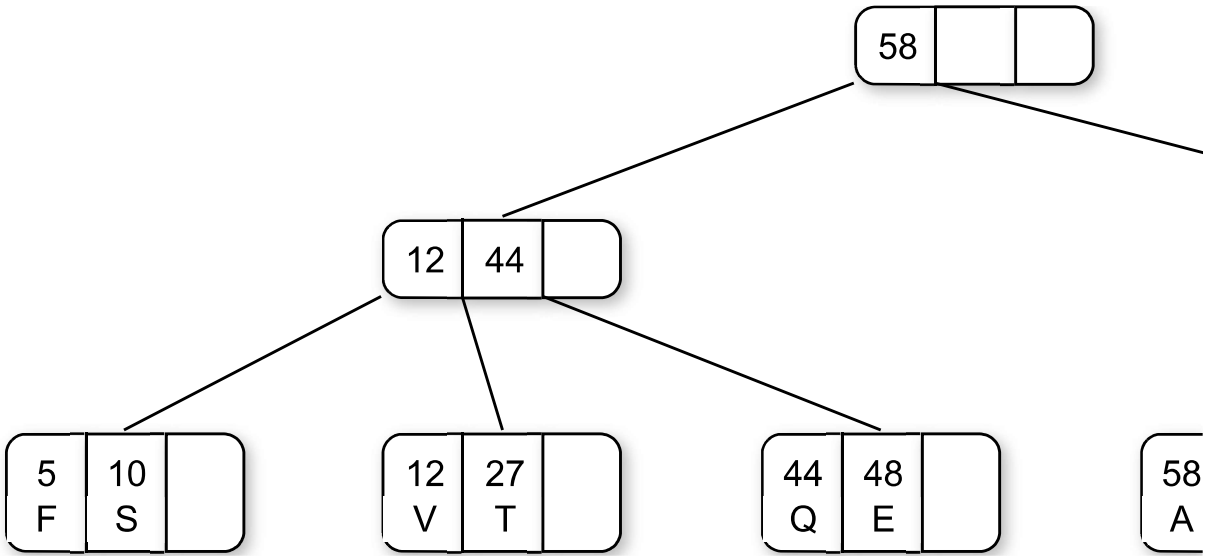


Figure 12.6.7: An example of deletion in a B+ tree of order four.

If deleting a record reduces the number of records in the node below the minimum threshold (called an **underflow**), then we must do something to keep the node sufficiently full. The first choice is to look at the node's adjacent siblings to determine if they have a spare record that can be used to fill the gap. If so, then enough records are transferred from the sibling so that both nodes have about the same number of records. This is done so as to delay as long as possible the next time when a delete causes this node to underflow again. This process might require that the parent node has its placeholder key value revised to reflect the true first key value in each node.

If neither sibling can lend a record to the under-full node (call it N), then N must give its records to a sibling and be removed from the tree. There is certainly room to do this,

because the sibling is at most half full (remember that it had no records to contribute to the current node), and N has become less than half full because it is under-flowing. This merge process combines two subtrees of the parent, which might cause it to underflow in turn. If the last two children of the root merge together, then the tree loses a level.

Here is a Java-like pseudocode for the B^+ tree delete algorithm.

```
/** Delete a record with the given key value, and
    return true if the root underflows */
private boolean removehelp(BPNode<Key,E> rt, Key k) {
  int currec = binaryle(rt.keys(), rt.numrecs(), k);
  if (rt.isLeaf()) {
    if (((BPLeaf<Key,E>)rt).keys()[currec] == k) {
      return ((BPLeaf<Key,E>)rt).delete(currec);
    }
    else { return false; }
  }
  else{ // Process internal node
    if (removehelp(((BPInternal<Key,E>)rt).pointers(currec),
        k)) {
      // Child will merge if necessary
      return ((BPInternal<Key,E>)rt).underflow(currec);
        }
    else { return false; }
  }
}
```

The B^+ tree requires that all nodes be at least half full (except for the root). Thus, the storage utilization must be at least 50%. This is satisfactory for many implementations, but note that keeping nodes fuller will result both in less space required (because there is less empty space in the disk file) and in more efficient processing (fewer blocks on average will be read into memory because the amount of information in each block is greater). Because B-trees have become so popular, many algorithm designers have tried to improve B-tree performance. One method for doing so is to use the B^+ tree variant known as the B^* tree. The B^* tree is identical to the B^+ tree, except for the rules used to split and merge nodes. Instead of splitting a node in half when it

overflows, the B^* tree gives some records to its neighboring sibling, if possible. If the sibling is also full, then these two nodes split into three. Similarly, when a node underflows, it is combined with its two siblings, and the total reduced to two nodes. Thus, the nodes are always at least two thirds full. [1]

Finally, here is an example of building a B+ Tree of order five. You can compare this to the example above of building a tree of order four with the same records.

1 / 33 << < >

Example B+ Tree Visualization: Insert into a tree of degree 5

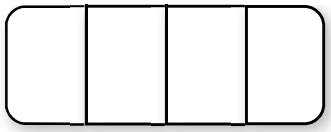


Figure 12.6.8: An example of building a B+ tree of degree 5

Click here for a visualization that will let you construct and interact with a B^+ tree. This visualization was written by David Galles of the University of San Francisco as part of his **Data Structure Visualizations** package.

[1]

This concept can be extended further if higher space utilization is required. However, the update routines become much more complicated. I once worked on a project where we implemented 3-for-4 node split and merge routines. This gave better performance than the 2-for-3 node split and merge routines of the B^* tree. However, the spitting and merging routines were so complicated that even their author could no longer understand them once they were completed!

12.6.1.2. B-Tree Analysis

The asymptotic cost of search, insertion, and deletion of records from B-trees, B^+ trees, and B^* trees is $\Theta(\log n)$ where n is the total number of records in the tree. However, the base of the log is the (average) branching factor of the tree. Typical database applications use extremely high branching factors, perhaps 100 or more. Thus, in practice the B-tree and its variants are extremely shallow.

As an illustration, consider a B^+ tree of order 100 and leaf nodes that contain up to 100 records. A B-B^+ tree with height one (that is, just a single leaf node) can have at most 100 records. A B^+ tree with height two (a root internal node whose children are leaves) must have at least 100 records (2 leaves with 50 records each). It has at most 10,000 records (100 leaves with 100 records each). A B^+ tree with height three must have at least 5000 records (two second-level nodes with 50 children containing 50 records each) and at most one million records (100 second-level nodes with 100 full children each). A B^+ tree with height four must have at least 250,000 records and at most 100 million records. Thus, it would require an *extremely* large database to generate a B^+ tree of more than height four.

The B^+ tree split and insert rules guarantee that every node (except perhaps the root) is at least half full. So they are on average about 3/4 full. But the internal nodes are purely overhead, since the keys stored there are used only by the tree to direct search, rather than store actual data. Does this overhead amount to a significant use of space? No, because once again the high fan-out rate of the tree structure means that the vast majority of nodes are leaf nodes. A **K-ary tree** has approximately $1/K$ of its nodes as internal nodes. This means that while half of a full binary tree's nodes are internal

nodes, in a B^+ tree of order 100 probably only about $1/75$ of its nodes are internal nodes. This means that the overhead associated with internal nodes is very low.

We can reduce the number of disk fetches required for the B-tree even more by using the following methods. First, the upper levels of the tree can be stored in main memory at all times. Because the tree branches so quickly, the top two levels (levels 0 and 1) require relatively little space. If the B-tree is only height four, then at most two disk fetches (internal nodes at level two and leaves at level three) are required to reach the pointer to any given record.

A buffer pool could be used to manage nodes of the B-tree. Several nodes of the tree would typically be in main memory at one time. The most straightforward approach is to use a standard method such as LRU to do node replacement. However, sometimes it might be desirable to “lock” certain nodes such as the root into the buffer pool. In general, if the buffer pool is even of modest size (say at least twice the depth of the tree), no special techniques for node replacement will be required because the upper-level nodes will naturally be accessed frequently.