10

Indexing

Many large-scale computing applications are centered around data sets that are too large to fit into main memory. The classic example is a large database of records with multiple search keys, requiring the ability to insert, delete, and search for records. Hashing provides outstanding performance for such situations, but only in the limited case in which all searches are of the form "find the record with key value K." Many applications require more general search capabilities. One example is a range query search for all records whose key lies within some range. Other queries might involve visiting all records in order of their key value, or finding the record with the greatest key value. Hash tables are not organized to support any of these queries efficiently.

This chapter introduces file structures used to organize a large collection of records stored on disk. Such file structures support efficient insertion, deletion, and search operations, for exact-match queries, range queries, and largest/smallest key value searches.

Before discussing such file structures, we must become familiar with some basic file-processing terminology. An **entry-sequenced file** stores records in the order that they were added to the file. Entry-sequenced files are the disk-based equivalent to an unsorted list and so do not support efficient search. The natural solution is to sort the records by order of the search key. However, a typical database, such as a collection of employee or customer records maintained by a business, might contain multiple search keys. To answer a question about a particular customer might require a search on the name of the customer. Businesses often wish to sort and output the records by zip code order for a bulk mailing. Government paperwork might require the ability to search by Social Security number. Thus, there might not be a single "correct" order in which to store the records.

Indexing is the process of associating a key with the location of a corresponding data record. Section 8.5 discussed the concept of a key sort, in which an **index file** is created whose records consist of key/pointer pairs. Here, each key is associated with a pointer to a complete record in the main database file. The index file

could be sorted or organized using a tree structure, thereby imposing a logical order on the records without physically rearranging them. One database might have several associated index files, each supporting efficient access through a different key field.

Each record of a database normally has a unique identifier, called the **primary key**. For example, the primary key for a set of personnel records might be the Social Security number or ID number for the individual. Unfortunately, the ID number is generally an inconvenient value on which to perform a search because the searcher is unlikely to know it. Instead, the searcher might know the desired employee's name. Alternatively, the searcher might be interested in finding all employees whose salary is in a certain range. If these are typical search requests to the database, then the name and salary fields deserve separate indices. However, key values in the name and salary indices are not likely to be unique.

A key field such as salary, where a particular key value might be duplicated in multiple records, is called a **secondary key**. Most searches are performed using a secondary key. The secondary key index (or more simply, **secondary index**) will associate a secondary key value with the primary key of each record having that secondary key value. At this point, the full database might be searched directly for the record with that primary key, or there might be a primary key index (or **primary index**) that relates each primary key value with a pointer to the actual record on disk. In the latter case, only the primary index provides the location of the actual record on disk, while the secondary indices refer to the primary index.

Indexing is an important technique for organizing large databases, and many indexing methods have been developed. Direct access through hashing is discussed in Section 9.4. A simple list sorted by key value can also serve as an index to the record file. Indexing disk files by sorted lists are discussed in the following section. Unfortunately, a sorted list does not perform well for insert and delete operations.

A third approach to indexing is the tree index. Trees are typically used to organize large databases that must support record insertion, deletion, and key range searches. Section 10.2 briefly describes ISAM, a tentative step toward solving the problem of storing a large database that must support insertion and deletion of records. Its shortcomings help to illustrate the value of tree indexing techniques. Section 10.3 introduces the basic issues related to tree indexing. Section 10.4 introduces the 2-3 tree, a balanced tree structure that is a simple form of the B-tree covered in Section 10.5. B-trees are the most widely used indexing method for large disk-based databases, and for implementing file systems. Since they have such great practical importance, many variations have been invented. Section 10.5 begins with a discussion of the variant normally referred to simply as a "B-tree." Section 10.5.1 presents the most widely implemented variant, the B^+-tree.

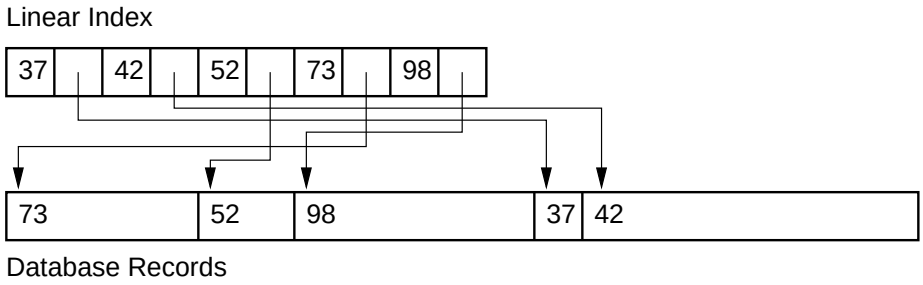

Figure 10.1 Linear indexing for variable-length records. Each record in the index file is of fixed length and contains a pointer to the beginning of the corresponding record in the database file.

10.1 Linear Indexing

A **linear index** is an index file organized as a sequence of key/pointer pairs where the keys are in sorted order and the pointers either (1) point to the position of the complete record on disk, (2) point to the position of the primary key in the primary index, or (3) are actually the value of the primary key. Depending on its size, a linear index might be stored in main memory or on disk. A linear index provides a number of advantages. It provides convenient access to variable-length database records, because each entry in the index file contains a fixed-length key field and a fixed-length pointer to the beginning of a (variable-length) record as shown in Figure 10.1. A linear index also allows for efficient search and random access to database records, because it is amenable to binary search.

If the database contains enough records, the linear index might be too large to store in main memory. This makes binary search of the index more expensive because many disk accesses would typically be required by the search process. One solution to this problem is to store a second-level linear index in main memory that indicates which disk block in the index file stores a desired key. For example, the linear index on disk might reside in a series of 1024-byte blocks. If each key/pointer pair in the linear index requires 8 bytes (a 4-byte key and a 4-byte pointer), then 128 key/pointer pairs are stored per block. The second-level index, stored in main memory, consists of a simple table storing the value of the key in the first position of each block in the linear index file. This arrangement is shown in Figure 10.2. If the linear index requires 1024 disk blocks (1MB), the second-level index contains only 1024 entries, one per disk block. To find which disk block contains a desired search key value, first search through the 1024-entry table to find the greatest value less than or equal to the search key. This directs the search to the proper block in the index file, which is then read into memory. At this point, a binary search within this block will produce a pointer to the actual record in the database. Because the

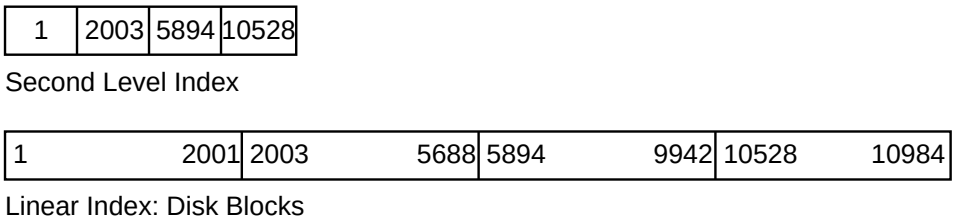

Figure 10.2 A simple two-level linear index. The linear index is stored on disk. The smaller, second-level index is stored in main memory. Each element in the second-level index stores the first key value in the corresponding disk block of the index file. In this example, the first disk block of the linear index stores keys in the range 1 to 2001, and the second disk block stores keys in the range 2003 to 5688. Thus, the first entry of the second-level index is key value 1 (the first key in the first block of the linear index), while the second entry of the second-level index is key value 2003.

second-level index is stored in main memory, accessing a record by this method requires two disk reads: one from the index file and one from the database file for the actual record.

Every time a record is inserted to or deleted from the database, all associated secondary indices must be updated. Updates to a linear index are expensive, because the entire contents of the array might be shifted. Another problem is that multiple records with the same secondary key each duplicate that key value within the index. When the secondary key field has many duplicates, such as when it has a limited range (e.g., a field to indicate job category from among a small number of possible job categories), this duplication might waste considerable space.

One improvement on the simple sorted array is a two-dimensional array where each row corresponds to a secondary key value. A row contains the primary keys whose records have the indicated secondary key value. Figure 10.3 illustrates this approach. Now there is no duplication of secondary key values, possibly yielding a considerable space savings. The cost of insertion and deletion is reduced, because only one row of the table need be adjusted. Note that a new row is added to the array when a new secondary key value is added. This might lead to moving many records, but this will happen infrequently in applications suited to using this arrangement.

A drawback to this approach is that the array must be of fixed size, which imposes an upper limit on the number of primary keys that might be associated with a particular secondary key. Furthermore, those secondary keys with fewer records than the width of the array will waste the remainder of their row. A better approach is to have a one-dimensional array of secondary key values, where each secondary key is associated with a linked list. This works well if the index is stored in main memory, but not so well when it is stored on disk because the linked list for a given key might be scattered across several disk blocks.

Jones	AA10	AB12	AB39	FF37
Smith	AX33	AX35	ZX45	
Zukowski	ZQ99			

Figure 10.3 A two-dimensional linear index. Each row lists the primary keys associated with a particular secondary key value. In this example, the secondary key is a name. The primary key is a unique four-character code.

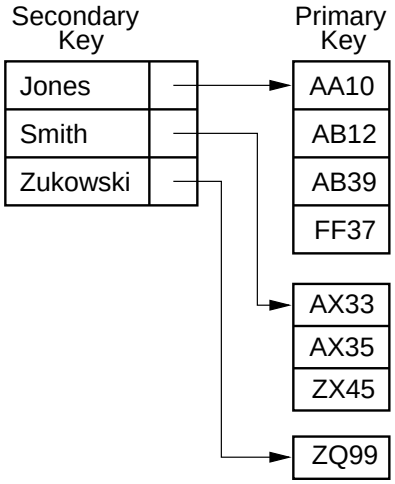

Figure 10.4 Illustration of an inverted list. Each secondary key value is stored in the secondary key list. Each secondary key value on the list has a pointer to a list of the primary keys whose associated records have that secondary key value.

Consider a large database of employee records. If the primary key is the employee's ID number and the secondary key is the employee's name, then each record in the name index associates a name with one or more ID numbers. The ID number index in turn associates an ID number with a unique pointer to the full record on disk. The secondary key index in such an organization is also known as an **inverted list** or **inverted file**. It is inverted in that searches work backwards from the secondary key to the primary key to the actual data record. It is called a list because each secondary key value has (conceptually) a list of primary keys associated with it. Figure 10.4 illustrates this arrangement. Here, we have last names as the secondary key. The primary key is a four-character unique identifier.

Figure 10.5 shows a better approach to storing inverted lists. An array of secondary key values is shown as before. Associated with each secondary key is a pointer to an array of primary keys. The primary key array uses a linked-list implementation. This approach combines the storage for all of the secondary key lists into a single array, probably saving space. Each record in this array consists of a

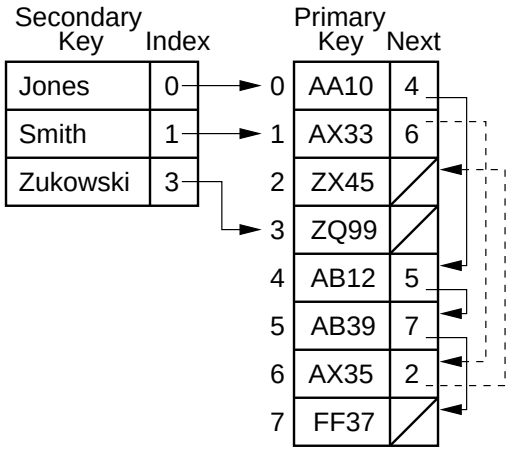

Figure 10.5 An inverted list implemented as an array of secondary keys and combined lists of primary keys. Each record in the secondary key array contains a pointer to a record in the primary key array. The **`next`** field of the primary key array indicates the next record with that secondary key value.

primary key value and a pointer to the next element on the list. It is easy to insert and delete secondary keys from this array, making this a good implementation for disk-based inverted files.

10.2 ISAM

How do we handle large databases that require frequent update? The main problem with the linear index is that it is a single, large array that does not adjust well to updates because a single update can require changing the position of every key in the index. Inverted lists reduce this problem, but they are only suitable for secondary key indices with many fewer secondary key values than records. The linear index would perform well as a primary key index if it could somehow be broken into pieces such that individual updates affect only a part of the index. This concept will be pursued throughout the rest of this chapter, eventually culminating in the B^+-tree, the most widely used indexing method today. But first, we begin by studying ISAM, an early attempt to solve the problem of large databases requiring frequent update. Its weaknesses help to illustrate why the B^+-tree works so well.

Before the invention of effective tree indexing schemes, a variety of disk-based indexing methods were in use. All were rather cumbersome, largely because no adequate method for handling updates was known. Typically, updates would cause the index to degrade in performance. ISAM is one example of such an index and was widely used by IBM prior to adoption of the B-tree.

ISAM is based on a modified form of the linear index, as illustrated by Figure 10.6. Records are stored in sorted order by primary key. The disk file is divided

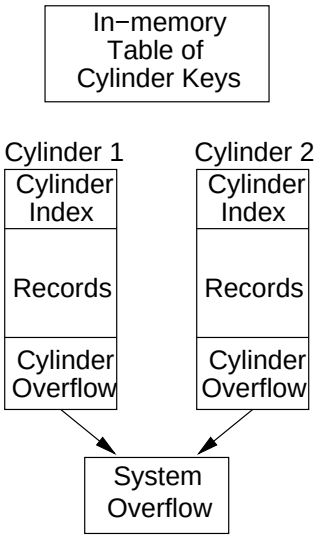

Figure 10.6 Illustration of the ISAM indexing system.

among a number of cylinders on disk.[1] Each cylinder holds a section of the list in sorted order. Initially, each cylinder is not filled to capacity, and the extra space is set aside in the **cylinder overflow**. In memory is a table listing the lowest key value stored in each cylinder of the file. Each cylinder contains a table listing the lowest key value for each block in that cylinder, called the **cylinder index**. When new records are inserted, they are placed in the correct cylinder's overflow area (in effect, a cylinder acts as a bucket). If a cylinder's overflow area fills completely, then a system-wide overflow area is used. Search proceeds by determining the proper cylinder from the system-wide table kept in main memory. The cylinder's block table is brought in from disk and consulted to determine the correct block. If the record is found in that block, then the search is complete. Otherwise, the cylinder's overflow area is searched. If that is full, and the record is not found, then the system-wide overflow is searched.

After initial construction of the database, so long as no new records are inserted or deleted, access is efficient because it requires only two disk fetches. The first disk fetch recovers the block table for the desired cylinder. The second disk fetch recovers the block that, under good conditions, contains the record. After many inserts, the overflow list becomes too long, resulting in significant search time as the cylinder overflow area fills up. Under extreme conditions, many searches might eventually lead to the system overflow area. The "solution" to this problem is to periodically reorganize the entire database. This means re-balancing the records

[1] Recall from Section 8.2.1 that a cylinder is all of the tracks readable from a particular placement of the heads on the multiple platters of a disk drive.

among the cylinders, sorting the records within each cylinder, and updating both the system index table and the within-cylinder block table. Such reorganization was typical of database systems during the 1960s and would normally be done each night or weekly.

10.3 Tree-based Indexing

Linear indexing is efficient when the database is static, that is, when records are inserted and deleted rarely or never. ISAM is adequate for a limited number of updates, but not for frequent changes. Because it has essentially two levels of indexing, ISAM will also break down for a truly large database where the number of cylinders is too great for the top-level index to fit in main memory.

In their most general form, database applications have the following characteristics:

1. Large sets of records that are frequently updated.
2. Search is by one or a combination of several keys.
3. Key range queries or min/max queries are used.

For such databases, a better organization must be found. One approach would be to use the binary search tree (BST) to store primary and secondary key indices. BSTs can store duplicate key values, they provide efficient insertion and deletion as well as efficient search, and they can perform efficient range queries. When there is enough main memory, the BST is a viable option for implementing both primary and secondary key indices.

Unfortunately, the BST can become unbalanced. Even under relatively good conditions, the depth of leaf nodes can easily vary by a factor of two. This might not be a significant concern when the tree is stored in main memory because the time required is still $\Theta(\log n)$ for search and update. When the tree is stored on disk, however, the depth of nodes in the tree becomes crucial. Every time a BST node *B* is visited, it is necessary to visit all nodes along the path from the root to *B*. Each node on this path must be retrieved from disk. Each disk access returns a block of information. If a node is on the same block as its parent, then the cost to find that node is trivial once its parent is in main memory. Thus, it is desirable to keep subtrees together on the same block. Unfortunately, many times a node is not on the same block as its parent. Thus, each access to a BST node could potentially require that another block to be read from disk. Using a buffer pool to store multiple blocks in memory can mitigate disk access problems if BST accesses display good locality of reference. But a buffer pool cannot eliminate disk I/O entirely. The problem becomes greater if the BST is unbalanced, because nodes deep in the tree have the potential of causing many disk blocks to be read. Thus, there are two significant issues that must be addressed to have efficient search from a disk-based

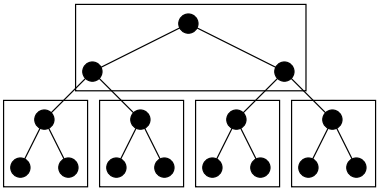

Figure 10.7 Breaking the BST into blocks. The BST is divided among disk blocks, each with space for three nodes. The path from the root to any leaf is contained on two blocks.

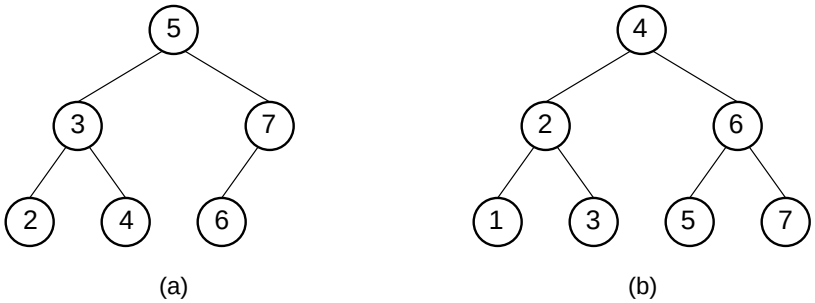

Figure 10.8 An attempt to re-balance a BST after insertion can be expensive. (a) A BST with six nodes in the shape of a complete binary tree. (b) A node with value 1 is inserted into the BST of (a). To maintain both the complete binary tree shape and the BST property, a major reorganization of the tree is required.

BST. The first is how to keep the tree balanced. The second is how to arrange the nodes on blocks so as to keep the number of blocks encountered on any path from the root to the leaves at a minimum.

We could select a scheme for balancing the BST and allocating BST nodes to blocks in a way that minimizes disk I/O, as illustrated by Figure 10.7. However, maintaining such a scheme in the face of insertions and deletions is difficult. In particular, the tree should remain balanced when an update takes place, but doing so might require much reorganization. Each update should affect only a few blocks, or its cost will be too high. As you can see from Figure 10.8, adopting a rule such as requiring the BST to be complete can cause a great deal of rearranging of data within the tree.

We can solve these problems by selecting another tree structure that automatically remains balanced after updates, and which is amenable to storing in blocks. There are a number of balanced tree data structures, and there are also techniques for keeping BSTs balanced. Examples are the AVL and splay trees discussed in Section 13.2. As an alternative, Section 10.4 presents the **2-3 tree**, which has the property that its leaves are always at the same level. The main reason for discussing the 2-3 tree here in preference to the other balanced search trees is that it naturally

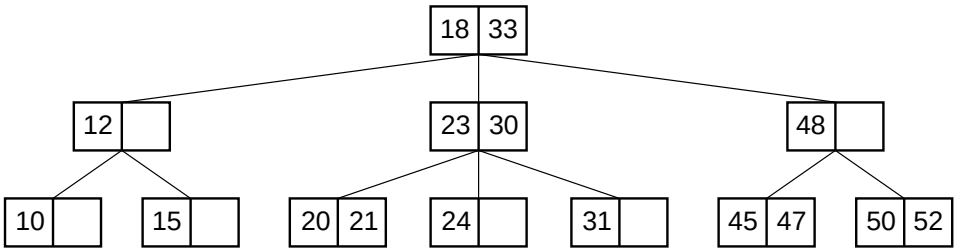

Figure 10.9 A 2-3 tree.

leads to the B-tree of Section 10.5, which is by far the most widely used indexing method today.

10.4 2-3 Trees

This section presents a data structure called the 2-3 tree. The 2-3 tree is not a binary tree, but instead its shape obeys the following definition:

1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys). Hence the name.
3. All leaves are at the same level in the tree, so the tree is always height balanced.

In addition to these shape properties, the 2-3 tree has a search tree property analogous to that of a BST. For every node, the values of all descendants in the left subtree are less than the value of the first key, while values in the center subtree are greater than or equal to the value of the first key. If there is a right subtree (equivalently, if the node stores two keys), then the values of all descendants in the center subtree are less than the value of the second key, while values in the right subtree are greater than or equal to the value of the second key. To maintain these shape and search properties requires that special action be taken when nodes are inserted and deleted. The 2-3 tree has the advantage over the BST in that the 2-3 tree can be kept height balanced at relatively low cost.

Figure 10.9 illustrates the 2-3 tree. Nodes are shown as rectangular boxes with two key fields. (These nodes actually would contain complete records or pointers to complete records, but the figures will show only the keys.) Internal nodes with only two children have an empty right key field. Leaf nodes might contain either one or two keys. Figure 10.10 is an implementation for the 2-3 tree node.

Note that this sample declaration does not distinguish between leaf and internal nodes and so is space inefficient, because leaf nodes store three pointers each. The techniques of Section 5.3.1 can be applied here to implement separate internal and leaf node types.

```
/** 2-3 tree node implementation */
class TTNode<Key extends Comparable<? super Key>,E> {
  private E lval;        // The left record
  private Key lkey;        // The node's left key
  private E rval;        // The right record
  private Key rkey;        // The node's right key
  private TTNode<Key,E> left;   // Pointer to left child
  private TTNode<Key,E> center; // Pointer to middle child
  private TTNode<Key,E> right;  // Pointer to right child

  public TTNode() { center = left = right = null; }
  public TTNode(Key lk, E lv, Key rk, E rv,
                TTNode<Key,E> p1, TTNode<Key,E> p2,
                TTNode<Key,E> p3) {
    lkey = lk; rkey = rk;
    lval = lv; rval = rv;
    left = p1; center = p2; right = p3;
  }

  public boolean isLeaf() { return left == null; }
  public TTNode<Key,E> lchild() { return left; }
  public TTNode<Key,E> rchild() { return right; }
  public TTNode<Key,E> cchild() { return center; }
  public Key lkey() { return lkey; }  // Left key
  public E lval() { return lval; }  // Left value
  public Key rkey() { return rkey; }  // Right key
  public E rval() { return rval; }  // Right value
  public void setLeft(Key k, E e) { lkey = k; lval = e; }
  public void setRight(Key k, E e) { rkey = k; rval = e; }
  public void setLeftChild(TTNode<Key,E> it) { left = it; }
  public void setCenterChild(TTNode<Key,E> it)
    { center = it; }
  public void setRightChild(TTNode<Key,E> it)
    { right = it; }
```

Figure 10.10 The 2-3 tree node implementation.

From the defining rules for 2-3 trees we can derive relationships between the number of nodes in the tree and the depth of the tree. A 2-3 tree of height k has at least 2^{k-1} leaves, because if every internal node has two children it degenerates to the shape of a complete binary tree. A 2-3 tree of height k has at most 3^{k-1} leaves, because each internal node can have at most three children.

Searching for a value in a 2-3 tree is similar to searching in a BST. Search begins at the root. If the root does not contain the search key K, then the search progresses to the only subtree that can possibly contain K. The value(s) stored in the root node determine which is the correct subtree. For example, if searching for the value 30 in the tree of Figure 10.9, we begin with the root node. Because 30 is between 18 and 33, it can only be in the middle subtree. Searching the middle child of the root node yields the desired record. If searching for 15, then the first step is

```
private E findhelp(TTNode<Key,E> root, Key k) {
  if (root == null) return null;          // val not found
  if (k.compareTo(root.lkey()) == 0) return root.lval();
  if ((root.rkey() != null) && (k.compareTo(root.rkey())
       == 0))
    return root.rval();
  if (k.compareTo(root.lkey()) < 0)       // Search left
    return findhelp(root.lchild(), k);
  else if (root.rkey() == null)           // Search center
    return findhelp(root.cchild(), k);
  else if (k.compareTo(root.rkey()) < 0)  // Search center
    return findhelp(root.cchild(), k);
  else return findhelp(root.rchild(), k); // Search right
}
```

Figure 10.11 Implementation for the 2-3 tree search method.

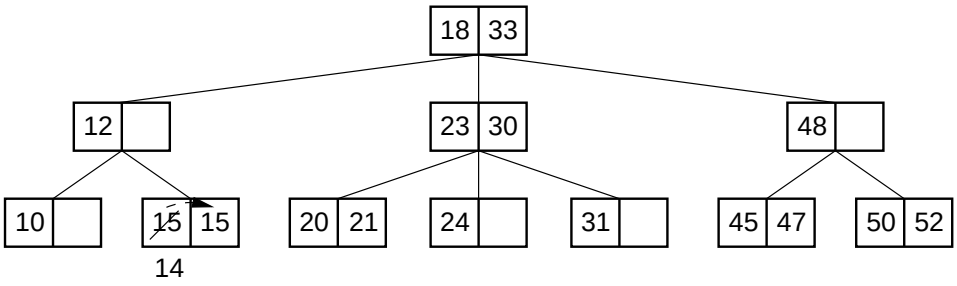

Figure 10.12 Simple insert into the 2-3 tree of Figure 10.9. The value 14 is inserted into the tree at the leaf node containing 15. Because there is room in the node for a second key, it is simply added to the left position with 15 moved to the right position.

again to search the root node. Because 15 is less than 18, the first (left) branch is taken. At the next level, we take the second branch to the leaf node containing 15. If the search key were 16, then upon encountering the leaf containing 15 we would find that the search key is not in the tree. Figure 10.11 is an implementation for the 2-3 tree search method.

Insertion into a 2-3 tree is similar to insertion into a BST to the extent that the new record is placed in the appropriate leaf node. Unlike BST insertion, a new child is not created to hold the record being inserted, that is, the 2-3 tree does not grow downward. The first step is to find the leaf node that would contain the record if it were in the tree. If this leaf node contains only one value, then the new record can be added to that node with no further modification to the tree, as illustrated in Figure 10.12. In this example, a record with key value 14 is inserted. Searching from the root, we come to the leaf node that stores 15. We add 14 as the left value (pushing the record with key 15 to the rightmost position).

If we insert the new record into a leaf node L that already contains two records, then more space must be created. Consider the two records of node L and the

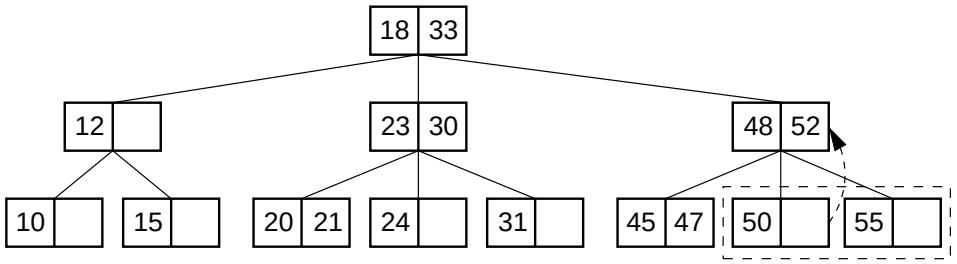

Figure 10.13 A simple node-splitting insert for a 2-3 tree. The value 55 is added to the 2-3 tree of Figure 10.9. This makes the node containing values 50 and 52 split, promoting value 52 to the parent node.

record to be inserted without further concern for which two were already in L and which is the new record. The first step is to split L into two nodes. Thus, a new node — call it L' — must be created from free store. L receives the record with the least of the three key values. L' receives the greatest of the three. The record with the middle of the three key value is passed up to the parent node along with a pointer to L'. This is called a **promotion**. The promoted key is then inserted into the parent. If the parent currently contains only one record (and thus has only two children), then the promoted record and the pointer to L' are simply added to the parent node. If the parent is full, then the split-and-promote process is repeated. Figure 10.13 illustrates a simple promotion. Figure 10.14 illustrates what happens when promotions require the root to split, adding a new level to the tree. In either case, all leaf nodes continue to have equal depth. Figures 10.15 and 10.16 present an implementation for the insertion process.

Note that **`inserthelp`** of Figure 10.15 takes three parameters. The first is a pointer to the root of the current subtree, named **`rt`**. The second is the key for the record to be inserted, and the third is the record itself. The return value for **`inserthelp`** is a pointer to a 2-3 tree node. If **`rt`** is unchanged, then a pointer to **`rt`** is returned. If **`rt`** is changed (due to the insertion causing the node to split), then a pointer to the new subtree root is returned, with the key value and record value in the leftmost fields, and a pointer to the (single) subtree in the center pointer field. This revised node will then be added to the parent, as illustrated in Figure 10.14.

When deleting a record from the 2-3 tree, there are three cases to consider. The simplest occurs when the record is to be removed from a leaf node containing two records. In this case, the record is simply removed, and no other nodes are affected. The second case occurs when the only record in a leaf node is to be removed. The third case occurs when a record is to be removed from an internal node. In both the second and the third cases, the deleted record is replaced with another that can take its place while maintaining the correct order, similar to removing a node from a BST. If the tree is sparse enough, there is no such record available that will allow all nodes to still maintain at least one record. In this situation, sibling nodes are

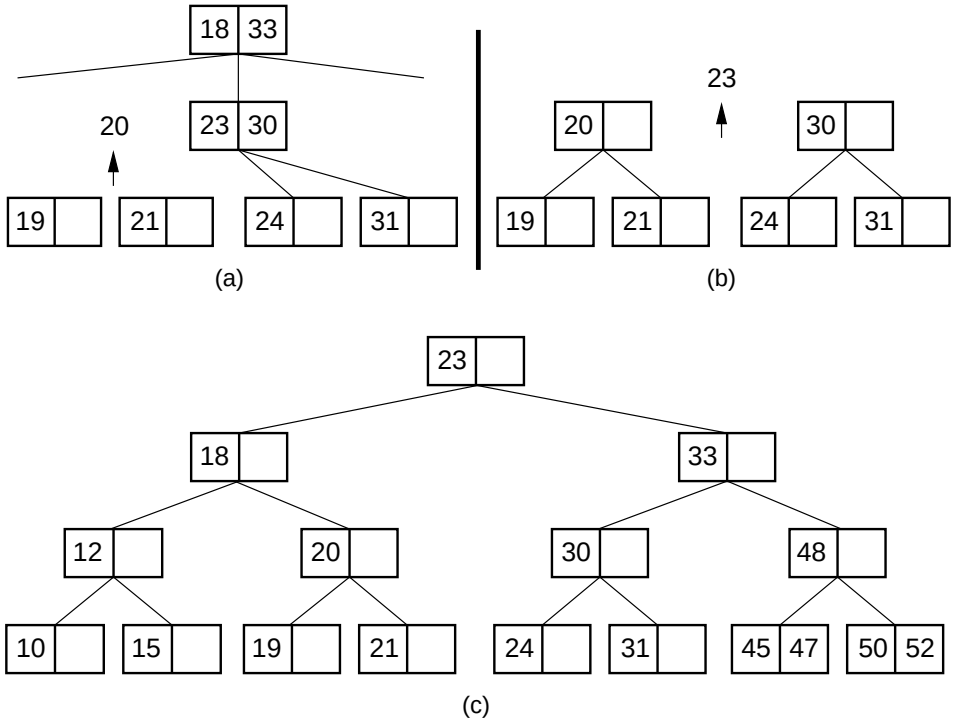

Figure 10.14 Example of inserting a record that causes the 2-3 tree root to split. (a) The value 19 is added to the 2-3 tree of Figure 10.9. This causes the node containing 20 and 21 to split, promoting 20. (b) This in turn causes the internal node containing 23 and 30 to split, promoting 23. (c) Finally, the root node splits, promoting 23 to become the left record in the new root. The result is that the tree becomes one level higher.

merged together. The delete operation for the 2-3 tree is excessively complex and will not be described further. Instead, a complete discussion of deletion will be postponed until the next section, where it can be generalized for a particular variant of the B-tree.

The 2-3 tree insert and delete routines do not add new nodes at the bottom of the tree. Instead they cause leaf nodes to split or merge, possibly causing a ripple effect moving up the tree to the root. If necessary the root will split, causing a new root node to be created and making the tree one level deeper. On deletion, if the last two children of the root merge, then the root node is removed and the tree will lose a level. In either case, all leaf nodes are always at the same level. When all leaf nodes are at the same level, we say that a tree is **height balanced**. Because the 2-3 tree is height balanced, and every internal node has at least two children, we know that the maximum depth of the tree is $\log n$. Thus, all 2-3 tree insert, find, and delete operations require $\Theta(\log n)$ time.

```
private TTNode<Key,E> inserthelp(TTNode<Key,E> rt,
                                 Key k, E e) {
  TTNode<Key,E> retval;
  if (rt == null) // Empty tree: create a leaf node for root
    return new TTNode<Key,E>(k, e, null, null,
                             null, null, null);
  if (rt.isLeaf()) // At leaf node: insert here
    return rt.add(new TTNode<Key,E>(k, e, null, null,
                                    null, null, null));
  // Add to internal node
  if (k.compareTo(rt.lkey()) < 0) { // Insert left
    retval = inserthelp(rt.lchild(), k, e);
    if (retval == rt.lchild()) return rt;
    else return rt.add(retval);
  }
  else if((rt.rkey() == null) ||
          (k.compareTo(rt.rkey()) < 0)) {
    retval = inserthelp(rt.cchild(), k, e);
    if (retval == rt.cchild()) return rt;
    else return rt.add(retval);
  }
  else { // Insert right
    retval = inserthelp(rt.rchild(), k, e);
    if (retval == rt.rchild()) return rt;
    else return rt.add(retval);
  }
}
```

Figure 10.15 The 2-3 tree insert routine.

10.5 B-Trees

This section 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. B-trees are always height balanced, with all leaf nodes at the same level.
2. Update and search operations affect only a few disk blocks. The fewer the number of disk blocks affected, 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 searches due to locality of reference.
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.

```
/** Add a new key/value pair to the node. There might be a
    subtree associated with the record being added. This
    information comes in the form of a 2-3 tree node with
    one key and a (possibly null) subtree through the
    center pointer field. */
  public TTNode<Key,E> add(TTNode<Key,E> it) {
    if (rkey == null) { // Only one key, add here
      if (lkey.compareTo(it.lkey()) < 0) {
        rkey = it.lkey(); rval = it.lval();
        right = center; center = it.cchild();
      }
      else {
        rkey = lkey; rval = lval; right = center;
        lkey = it.lkey(); lval = it.lval();
        center = it.cchild();
      }
      return this;
    }
    else if (lkey.compareTo(it.lkey()) >= 0) { // Add left
      center = new TTNode<Key,E>(rkey, rval, null, null,
                                 center, right, null);
      rkey = null; rval = null; right = null;
      it.setLeftChild(left); left = it;
      return this;
    }
    else if (rkey.compareTo(it.lkey()) < 0) { // Add center
      it.setCenterChild(new TTNode<Key,E>(rkey, rval, null,
                            null, it.cchild(), right, null));
      it.setLeftChild(this);
      rkey = null; rval = null; right = null;
      return it;
    }
    else { // Add right
      TTNode<Key,E> N1 = new TTNode<Key,E>(rkey, rval, null,
                                      null, this, it, null);
      it.setLeftChild(right);
      right = null; rkey = null; rval = null;
      return N1;
    }
  }
```

Figure 10.16 The 2-3 tree node **add** method.

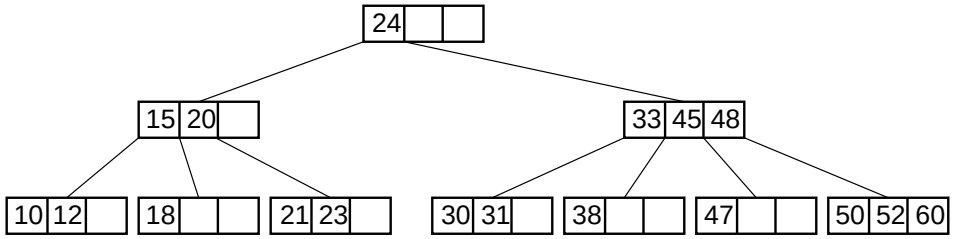

Figure 10.17 A B-tree of order four.

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 (see Section 8.3).

Figure 10.17 shows a B-tree of order four. Each node contains up to three keys, and internal nodes have up to four children.

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 10.17. 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.

10.5.1 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, nor is the 2-3 tree as described in Section 10.4. 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.

When data are static, 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

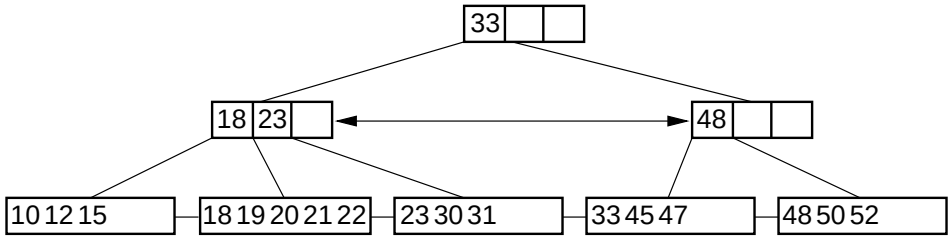

Figure 10.18 Example of a B$^+$-tree of order four. Internal nodes must store between two and four children. For this example, the record size is assumed to be such that leaf nodes store between three and five records.

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 10.17 shows internal nodes containing three keys and four pointers, class **BPNode** is slightly different in that it stores key/pointer pairs. Figure 10.17 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.

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 10.18 illustrates the B$^+$-tree.

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

```
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);
}
```

Figure 10.19 Implementation for the B$^+$-tree search method.

access to the actual record. To find a record with key value 33 in the B$^+$-tree of Figure 10.18, search begins at the root. The value 33 stored in the root merely serves as a placeholder, indicating that keys with values greater than or equal to 33 are found in the second subtree. From the second child of the root, the first branch is taken to reach the leaf node containing the actual record (or a pointer to the actual record) with key value 33. Figure 10.19 shows a pseudocode sketch of the B$^+$-tree search algorithm.

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 10.20 illustrates the insertion process through several examples. Figure 10.21 shows a Java-like pseudocode sketch of the B$^+$-tree insert algorithm.

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 10.22.

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. Figure 10.23 illustrates the process.

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

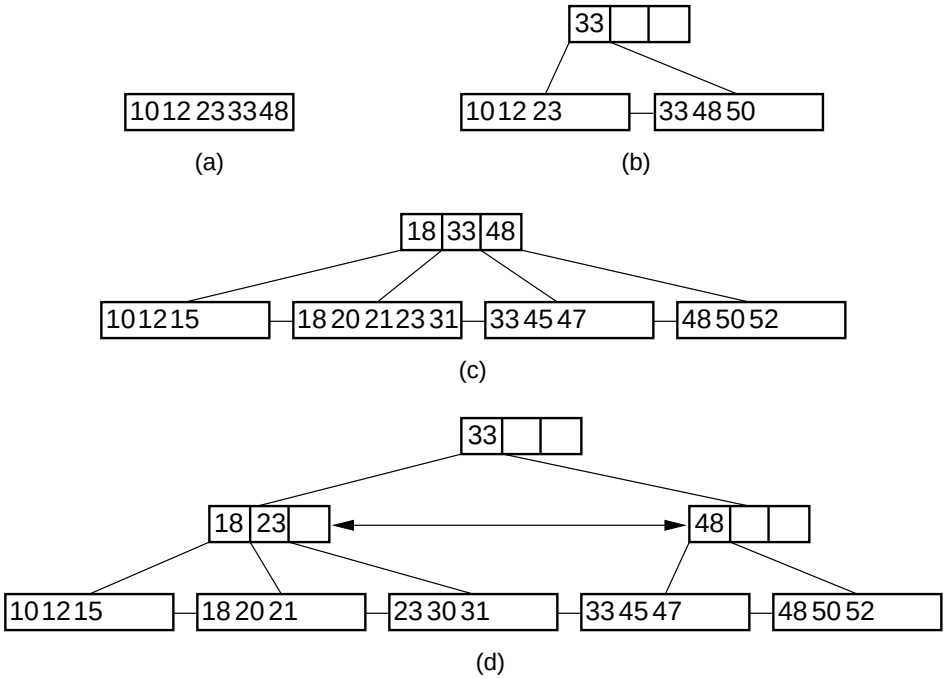

Figure 10.20 Examples of B^+-tree insertion. (a) A B^+-tree containing five records. (b) The result of inserting a record with key value 50 into the tree of (a). The leaf node splits, causing creation of the first internal node. (c) The B^+-tree of (b) after further insertions. (d) The result of inserting a record with key value 30 into the tree of (c). The second leaf node splits, which causes the internal node to split in turn, creating a new root.

```
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;
}
```

Figure 10.21 A Java-like pseudocode sketch of the B^+-tree insert algorithm.

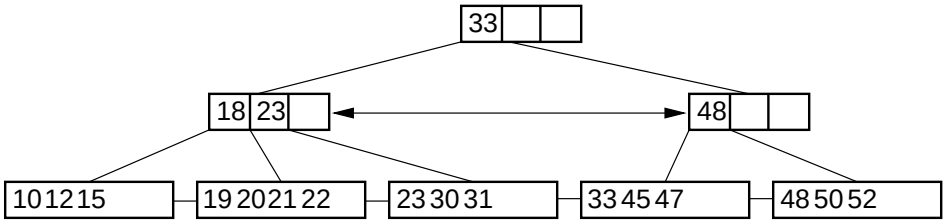

Figure 10.22 Simple deletion from a B$^+$-tree. The record with key value 18 is removed from the tree of Figure 10.18. Note that even though 18 is also a placeholder used to direct search in the parent node, that value need not be removed from internal nodes even if no record in the tree has key value 18. Thus, the leftmost node at level one in this example retains the key with value 18 after the record with key value 18 has been removed from the second leaf node.

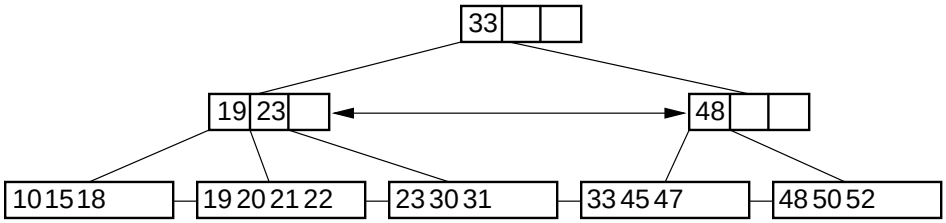

Figure 10.23 Deletion from the B$^+$-tree of Figure 10.18 via borrowing from a sibling. The key with value 12 is deleted from the leftmost leaf, causing the record with key value 18 to shift to the leftmost leaf to take its place. Note that the parent must be updated to properly indicate the key range within the subtrees. In this example, the parent node has its leftmost key value changed to 19.

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. Figure 10.24 illustrates the node-merge deletion process. Figure 10.25 shows Java-like pseudocode for the B$^+$-tree delete algorithm.

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

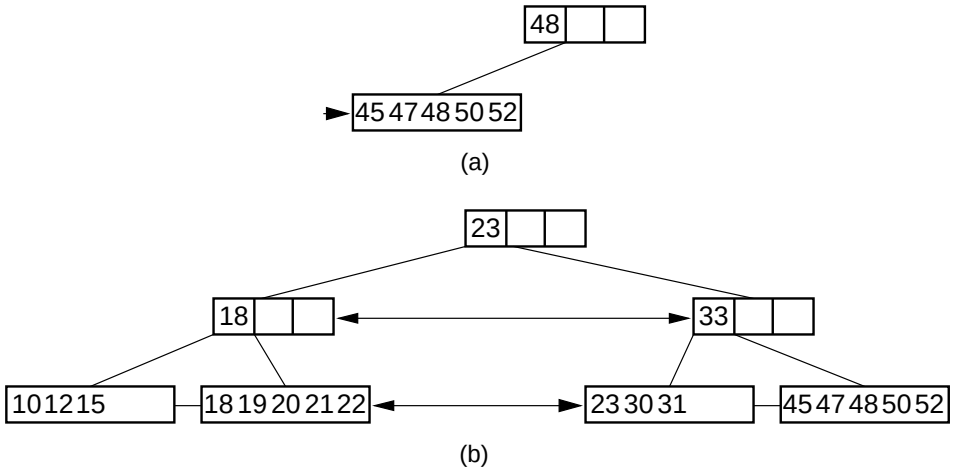

Figure 10.24 Deleting the record with key value 33 from the B$^+$-tree of Figure 10.18 via collapsing siblings. (a) The two leftmost leaf nodes merge together to form a single leaf. Unfortunately, the parent node now has only one child. (b) Because the left subtree has a spare leaf node, that node is passed to the right subtree. The placeholder values of the root and the right internal node are updated to reflect the changes. Value 23 moves to the root, and old root value 33 moves to the rightmost internal node.

```
/** 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;
}
```

Figure 10.25 Java-like pseudocode for the B$^+$-tree delete algorithm.

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.[2]

10.5.2 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^+-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. Recall (from Section 6.4) that a full 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

[2] 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!

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.

10.6 Further Reading

For an expanded discussion of the issues touched on in this chapter, see a general file processing text such as *File Structures: A Conceptual Toolkit* by Folk and Zoellick [FZ98]. In particular, Folk and Zoellick provide a good discussion of the relationship between primary and secondary indices. The most thorough discussion on various implementations for the B-tree is the survey article by Comer [Com79]. Also see [Sal88] for further details on implementing B-trees. See Shaffer and Brown [SB93] for a discussion of buffer pool management strategies for B^+-tree-like data structures.

10.7 Exercises

10.1 Assume that a computer system has disk blocks of 1024 bytes, and that you are storing records that have 4-byte keys and 4-byte data fields. The records are sorted and packed sequentially into the disk file.

(a) Assume that a linear index uses 4 bytes to store the key and 4 bytes to store the block ID for the associated records. What is the greatest number of records that can be stored in the file if a linear index of size 256KB is used?

(b) What is the greatest number of records that can be stored in the file if the linear index is also stored on disk (and thus its size is limited only by the second-level index) when using a second-level index of 1024 bytes (i.e., 256 key values) as illustrated by Figure 10.2? Each element of the second-level index references the smallest key value for a disk block of the linear index.

10.2 Assume that a computer system has disk blocks of 4096 bytes, and that you are storing records that have 4-byte keys and 64-byte data fields. The records are sorted and packed sequentially into the disk file.

(a) Assume that a linear index uses 4 bytes to store the key and 4 bytes to store the block ID for the associated records. What is the greatest

number of records that can be stored in the file if a linear index of size 2MB is used?

(b) What is the greatest number of records that can be stored in the file if the linear index is also stored on disk (and thus its size is limited only by the second-level index) when using a second-level index of 4096 bytes (i.e., 1024 key values) as illustrated by Figure 10.2? Each element of the second-level index references the smallest key value for a disk block of the linear index.

10.3 Modify the function **`binary`** of Section 3.5 so as to support variable-length records with fixed-length keys indexed by a simple linear index as illustrated by Figure 10.1.

10.4 Assume that a database stores records consisting of a 2-byte integer key and a variable-length data field consisting of a string. Show the linear index (as illustrated by Figure 10.1) for the following collection of records:

397	Hello world!
82	XYZ
1038	This string is rather long
1037	This is shorter
42	ABC
2222	Hello new world!

10.5 Each of the following series of records consists of a four-digit primary key (with no duplicates) and a four-character secondary key (with many duplicates).

3456	DEER
2398	DEER
2926	DUCK
9737	DEER
7739	GOAT
9279	DUCK
1111	FROG
8133	DEER
7183	DUCK
7186	FROG

(a) Show the inverted list (as illustrated by Figure 10.4) for this collection of records.

(b) Show the improved inverted list (as illustrated by Figure 10.5) for this collection of records.

10.6 Under what conditions will ISAM be more efficient than a B^+-tree implementation?

10.7 Prove that the number of leaf nodes in a 2-3 tree with height k is between 2^{k-1} and 3^{k-1}.

10.8 Show the result of inserting the values 55 and 46 into the 2-3 tree of Figure 10.9.

10.9 You are given a series of records whose keys are letters. The records arrive in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the 2-3 tree that results from inserting these records.

10.10 You are given a series of records whose keys are letters. The records are inserted in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the tree that results from inserting these records when the 2-3 tree is modified to be a 2-3$^+$ tree, that is, the internal nodes act only as placeholders. Assume that the leaf nodes are capable of holding up to two records.

10.11 Show the result of inserting the value 55 into the B-tree of Figure 10.17.

10.12 Show the result of inserting the values 1, 2, 3, 4, 5, and 6 (in that order) into the B$^+$-tree of Figure 10.18.

10.13 Show the result of deleting the values 18, 19, and 20 (in that order) from the B$^+$-tree of Figure 10.24b.

10.14 You are given a series of records whose keys are letters. The records are inserted in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the B$^+$-tree of order four that results from inserting these records. Assume that the leaf nodes are capable of storing up to three records.

10.15 Assume that you have a B$^+$-tree whose internal nodes can store up to 100 children and whose leaf nodes can store up to 15 records. What are the minimum and maximum number of records that can be stored by the B$^+$-tree with heights 1, 2, 3, 4, and 5?

10.16 Assume that you have a B$^+$-tree whose internal nodes can store up to 50 children and whose leaf nodes can store up to 50 records. What are the minimum and maximum number of records that can be stored by the B$^+$-tree with heights 1, 2, 3, 4, and 5?

10.8 Projects

10.1 Implement a two-level linear index for variable-length records as illustrated by Figures 10.1 and 10.2. Assume that disk blocks are 1024 bytes in length. Records in the database file should typically range between 20 and 200 bytes, including a 4-byte key value. Each record of the index file should store a key value and the byte offset in the database file for the first byte of the corresponding record. The top-level index (stored in memory) should be a simple array storing the lowest key value on the corresponding block in the index file.

10.2 Implement the 2-3$^+$ tree, that is, a 2-3 tree where the internal nodes act only as placeholders. Your 2-3$^+$ tree should implement the dictionary interface of Section 4.4.

10.3 Implement the dictionary ADT of Section 4.4 for a large file stored on disk by means of the B$^+$-tree of Section 10.5. Assume that disk blocks are 1024 bytes, and thus both leaf nodes and internal nodes are also 1024 bytes. Records should store a 4-byte (`int`) key value and a 60-byte data field. Internal nodes should store key value/pointer pairs where the "pointer" is actually the block number on disk for the child node. Both internal nodes and leaf nodes will need room to store various information such as a count of the records stored on that node, and a pointer to the next node on that level. Thus, leaf nodes will store 15 records, and internal nodes will have room to store about 120 to 125 children depending on how you implement them. Use a buffer pool (Section 8.3) to manage access to the nodes stored on disk.