APPENDIX A

Overview of B-Trees

Tree-based searching methods involve making branches based on the results of comparison operations. When the volume of data is very large, we find that often the tree is too large to fit in memory at once. Therefore, the tree is stored on many disk pages, and some of the pages might not be in memory at the time the comparison is made. In such an environment, the branches in the tree contain disk addresses, and it is not uncommon for a comparison operation to result in a page fault. This calls for search techniques known as *external searching*. The B-tree and its variants are the most commonly used external searching techniques and are the subject of this appendix.

When the tree is a binary tree, the number of page faults can be quite high. For example, for a binary tree of N records, the number of comparison operations is minimized if the binary tree is complete (a concept similar to global balance). In this case, we need to make approximately $\log_2 N$ comparisons before reaching a leaf node. If N is 2 million, then we will make 21 comparison operations, which potentially could cause 21 page faults. Thus, binary trees are not good for external searching.

Our goal is to minimize the number of page faults. There are several ways of achieving this goal. One way is to aggregate the results of the comparisons by delaying the branch operation to the missing page. For example, when N is 2 million, we can transform the complete binary tree of depth 21 to a complete tree of depth 7, where groups of seven key values are formed at every three levels of the tree. The result is that each node of the new tree has eight branches instead of two, and the effect is more like an 8-ary tree (see Figure A.1). Thus, we will have at most 7 page faults (instead of 21) when retrieving the data. It is important to note that we have separated the comparison operations from the branching process. This is clear in the figure, where we see the locality of the comparisons—that is, a branch is only attempted once the appropriate sequence of three comparisons has been executed.

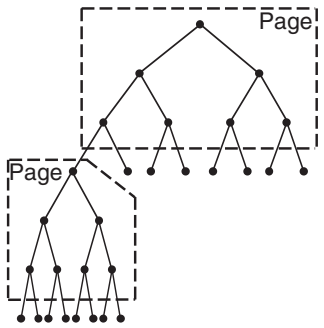

Figure A.1
8-ary tree.

The small page sizes (i.e., seven nodes), as in the example above, are widely used to improve cache locality of main memory data structures in modern processors where cache lines are typically 256 bytes at the present [746]. Nevertheless, when the above scheme is used, we usually aggregate more than seven key values. The aggregation is such that the result is stored on a page. It is not inconceivable to store as many as 500 key values on one of the pages. For example, if we store 511 key values on a page, then we have 512 branches and with at most three page faults, we can access over 128 million records. In fact, this is the basis of the *indexed-sequential file organization* (ISAM) [705] for storing data on disk. In this case, the first branch indicates a cylinder address, the second branch indicates a track along the cylinder, and the third address indicates the address of the record.

Another way to minimize the page faults is to use a multiway tree instead of a binary tree. A *multiway tree* is a generalization of a binary search tree where at each

APPENDIX B

Linear Hashing

Linear hashing addresses the situation that arises when the hash table (usually consisting of buckets) is full or when the hash chains in a chaining method (attributed to Luhn by Knuth [1046, p. 547]) get too long. One solution is to expand the size of the hash table. Since the number of buckets (or chains in a chaining method) is usually fixed by the hash function, we must modify our hash function to generate more buckets. This modification should ideally require a minimal amount of rehashing—that is, only a few records should have to be moved to new buckets. Generally, most hashing methods do not allow the number of buckets to grow gracefully.

Knott [1040] suggests storing the buckets using a trie (see the definition in the opening section of Chapter 1) in the form of a binary tree (e.g., Figure B.1(a)). In this scheme, dealing with bucket overflow is rather trivial since the bucket is split into two parts (e.g., bucket B in Figure B.1(a) is split into buckets E and F in Figure B.1(b)). The problem with such a scheme is that accessing a bucket at depth l requires l operations. However, the tree is usually kept in memory, and thus these l operations are considerably faster than a disk access, and, in fact, only one disk access is required.

A more general formulation of Knott's approach is described by Larson [1104], who makes use of m binary trees, each of which contains just one leaf node initially. These leaf nodes correspond to buckets with a finite capacity c. Given a key k, applying a hash function h to k (i.e., $h(k)$) yields the binary tree that contains k. The collection of m one-node binary trees is analogous to a hash table with m buckets. As items are added to the hash table, the buckets corresponding to the leaf nodes become full, and thus they are expanded by splitting them, thereby creating two new buckets for each split. Subsequent insertions may cause these buckets also to fill up, thereby resulting in further splits. The splitting process is represented by a binary tree.

This method is analogous to the separate chaining method,[1] where, instead of placing the overflow buckets in a chain (i.e., linked list), we organize each bucket and its set of overflow buckets as a binary tree whose leaf nodes consist of these buckets. Thus, we have m binary trees. Note also that, unlike separate chaining, all buckets have the same capacity.

At this point, it is hard to see the advantage of using the binary tree organization of the overflow buckets over the separate chaining approach unless we can impose some access structure on the binary tree that enables us to avoid examining all of the buckets in the worst case of a search (whether or not it is successful). Larson [1104] proposes to

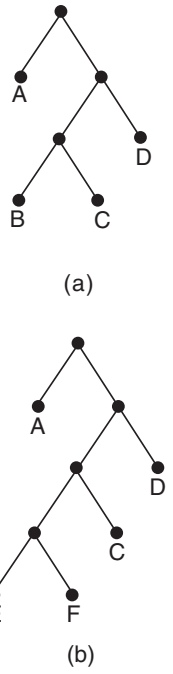

Figure B.1
Example illustrating the representation of a hash table as a binary tree and (b) the result of splitting bucket B into buckets E and F upon overflow.

[1] Briefly, in this case, we have a table of m hash values and pointers to m hash lists—that is, one for each hash value. The hash value is commonly referred to as a *bucket address* since, frequently, more than one key value is associated with it. The hash lists are used to store all of the records whose key values have the same hash value.

APPENDIX C

Spiral Hashing

One of the drawbacks of linear hashing (see Appendix B) is that the order in which the buckets are split is unrelated to the probability of the occurrence of an overflow. In particular, all the buckets that are candidates for a split have the same probability of overflowing. Moreover, the expected cost of retrieving a record and updating the table of active buckets varies cyclically in the sense that it depends on the proportion of the buckets that have been split during a bucket expansion cycle (see Exercises 6–13 in Appendix B).

Proposed by Martin [1247], spiral hashing [324, 384, 1106, 1247, 1324] is a technique whose performance (i.e., the expected search length in a bucket) has been observed to be independent of the number (or fraction) of the buckets that have been split. Thus, it is said to have uniform performance (see Exercise 23). This is achieved by distributing the records in an uneven manner over the active buckets. Moreover, it always splits the bucket that has the highest probability of overflowing.

Spiral hashing is similar in spirit to linear hashing. To simplify our explanation, we identify the buckets by their addresses. The central idea behind spiral hashing is that there is an ever-changing (and growing) address space of active bucket addresses that is moving forward (e.g., Figure C.1). This is in contrast to linear hashing in which the active bucket addresses always range from 0 to $m - 1$. Recall that in linear hashing, when the bucket at address s is split, then a new bucket is created at location m, and the previous contents of bucket s are rehashed and inserted into buckets s and m, as is appropriate.

In the following discussion, we assume that a bucket can be split into r buckets rather than just two as was the case in our explanation of linear hashing. r is called the growth factor. We assume further that r is an integer; however, this restriction can be lifted. Let s and t denote the addresses of the first and last active buckets, respectively, so that each bucket with address i ($s \leq i \leq t$) is active. Define the storage utilization factor, τ, in the same way as for linear hashing—that is, the ratio of the number of records in the file to the number of positions available in the existing primary and overflow buckets. A bucket is split whenever τ exceeds a predetermined value, say α. When τ falls below a predetermined value, say β, buckets should be merged (i.e., the bucket-splitting process should be reversed). However, this has not been the subject of much research, and we ignore β here (see Exercise 5).

When the bucket at address s is split, say into r buckets, then r new buckets are allocated starting at bucket address $t + 1$, and the contents of bucket s are rehashed into these new buckets. Bucket address s is no longer used (but see Exercise 7), and the active bucket addresses now range from $s + 1$ to $t + r$. As we will see, the manner in which the hashing function is chosen guarantees that the expected load (i.e., the expected number of records per bucket) is always at a maximum at bucket s and is at a minimum at bucket