! **Exercise 1.6 :** On the assumptions of Exercise 1.5(a), what is the average number of disk I/O's to find and retrieve the ten records with a given search-key value, both with and without the bucket structure? Assume nothing is in memory to begin, but it is possible to locate index or bucket blocks without incurring additional I/O's beyond what is needed to retrieve these blocks into memory.

Exercise 1.7 : Suppose we have a repository of 1000 documents, and we wish to build an inverted index with 10,000 words. A block can hold ten word-pointer pairs or 50 pointers to either a document or a position within a document. The distribution of words is Zipfian; the number of occurrences of the ith most frequent word is $100000/\sqrt{i}$, for $i = 1, 2, \ldots, 10000$.

a) What is the averge number of words per document?

b) Suppose our inverted index only records for each word all the documents that have that word. What is the maximum number of blocks we could need to hold the inverted index?

c) Suppose our inverted index holds pointers to each occurrence of each word. How many blocks do we need to hold the inverted index?

d) Repeat (b) if the 400 most common words ("stop" words) are *not* included in the index.

e) Repeat (c) if the 400 most common words are not included in the index.

Exercise 1.8 : If we use an augmented inverted index, such as in Fig. 10, we can perform a number of other kinds of searches. Suggest how this index could be used to find:

a) Documents in which "cat" and "dog" appeared within five positions of each other in the same type of element (e.g., title, text, or anchor).

b) Documents in which "dog" followed "cat" separated by exactly one position.

c) Documents in which "dog" and "cat" both appear in the title.

2 B-Trees

While one or two levels of index are often very helpful in speeding up queries, there is a more general structure that is commonly used in commercial systems. This family of data structures is called *B-trees*, and the particular variant that is most often used is known as a *B+ tree*. In essence:

- B-trees automatically maintain as many levels of index as is appropriate for the size of the file being indexed.
- B-trees manage the space on the blocks they use so that every block is between half used and completely full.

In the following discussion, we shall talk about "B-trees," but the details will all be for the B+ tree variant. Other types of B-tree are discussed in exercises.

2.1 The Structure of B-trees

A B-tree organizes its blocks into a tree that is *balanced*, meaning that all paths from the root to a leaf have the same length. Typically, there are three layers in a B-tree: the root, an intermediate layer, and leaves, but any number of layers is possible. To help visualize B-trees, you may wish to look ahead at Figs. 11 and 12, which show nodes of a B-tree, and Fig. 13, which shows an entire B-tree.

There is a parameter n associated with each B-tree index, and this parameter determines the layout of all blocks of the B-tree. Each block will have space for n search-key values and $n+1$ pointers. In a sense, a B-tree block is similar to the index blocks introduced in Section 1.2, except that the B-tree block has an extra pointer, along with n key-pointer pairs. We pick n to be as large as will allow $n+1$ pointers and n keys to fit in one block.

Example 10: Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then we want to find the largest integer value of n such that $4n + 8(n+1) \leq 4096$. That value is $n = 340$. $\square$

There are several important rules about what can appear in the blocks of a B-tree:

- The keys in leaf nodes are copies of keys from the data file. These keys are distributed among the leaves in sorted order, from left to right.
- At the root, there are at least two used pointers.[2] All pointers point to B-tree blocks at the level below.
- At a leaf, the last pointer points to the next leaf block to the right, i.e., to the block with the next higher keys. Among the other n pointers in a leaf block, at least $\lfloor (n+1)/2 \rfloor$ of these pointers are used and point to data records; unused pointers are null and do not point anywhere. The ith pointer, if it is used, points to a record with the ith key.

[2]Technically, there is a possibility that the entire B-tree has only one pointer because it is an index into a data file with only one record. In this case, the entire tree is a root block that is also a leaf, and this block has only one key and one pointer. We shall ignore this trivial case in the descriptions that follow.

- At an interior node, all $n+1$ pointers can be used to point to B-tree blocks at the next lower level. At least $\lceil (n+1)/2 \rceil$ of them are actually used (but if the node is the root, then we require only that at least 2 be used, regardless of how large n is). If j pointers are used, then there will be $j-1$ keys, say $K_1, K_2, \ldots, K_{j-1}$. The first pointer points to a part of the B-tree where some of the records with keys less than K_1 will be found. The second pointer goes to that part of the tree where all records with keys that are at least K_1, but less than K_2 will be found, and so on. Finally, the jth pointer gets us to the part of the B-tree where some of the records with keys greater than or equal to K_{j-1} are found. Note that some records with keys far below K_1 or far above K_{j-1} may not be reachable from this block at all, but will be reached via another block at the same level.

- All used pointers and their keys appear at the beginning of the block, with the exception of the $(n+1)$st pointer in a leaf, which points to the next leaf.

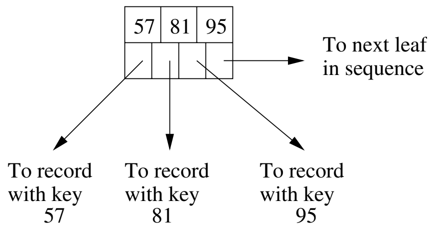

Figure 11: A typical leaf of a B-tree

Example 11: Our running example of B-trees will use $n = 3$. That is, blocks have room for three keys and four pointers, which are atypically small numbers. Keys are integers. Figure 11 shows a leaf that is completely used. There are three keys, 57, 81, and 95. The first three pointers go to records with these keys. The last pointer, as is always the case with leaves, points to the next leaf to the right in the order of keys; it would be null if this leaf were the last in sequence.

A leaf is not necessarily full, but in our example with $n = 3$, there must be at least two key-pointer pairs. That is, the key 95 in Fig. 11 might be missing, and if so, the third pointer would be null.

Figure 12 shows a typical interior node. There are three keys, 14, 52, and 78. There are also four pointers in this node. The first points to a part of the B-tree from which we can reach only records with keys less than 14 — the first of the keys. The second pointer leads to all records with keys between the first and second keys of the B-tree block; the third pointer is for those records

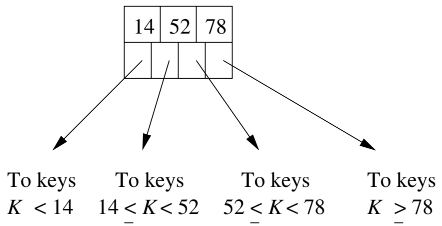

Figure 12: A typical interior node of a B-tree

between the second and third keys of the block, and the fourth pointer lets us reach some of the records with keys equal to or above the third key of the block.

As with our example leaf, it is not necessarily the case that all slots for keys and pointers are occupied. However, with $n = 3$, at least the first key and the first two pointers must be present in an interior node. □

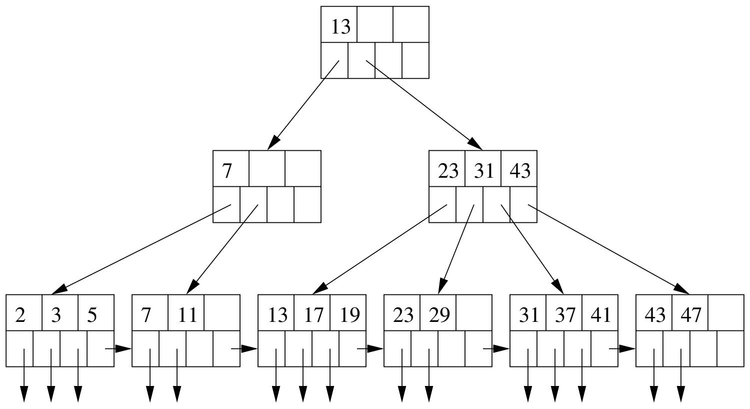

Figure 13: A B-tree

Example 12: Figure 13 shows an entire three-level B-tree, with $n = 3$, as in Example 11. We have assumed that the data file consists of records whose keys are all the primes from 2 to 47. Notice that at the leaves, each of these keys appears once, in order. All leaf blocks have two or three key-pointer pairs, plus a pointer to the next leaf in sequence. The keys are in sorted order as we look across the leaves from left to right.

The root has only two pointers, the minimum possible number, although it could have up to four. The one key at the root separates those keys reachable via the first pointer from those reachable via the second. That is, keys up to 12 could be found in the first subtree of the root, and keys 13 and up are in the second subtree.

If we look at the first child of the root, with key 7, we again find two pointers, one to keys less than 7 and the other to keys 7 and above. Note that the second pointer in this node gets us only to keys 7 and 11, not to *all* keys ≥ 7, such as 13.

Finally, the second child of the root has all four pointer slots in use. The first gets us to some of the keys less than 23, namely 13, 17, and 19. The second pointer gets us to all keys K such that $23 \leq K < 31$; the third pointer lets us reach all keys K such that $31 \leq K < 43$, and the fourth pointer gets us to some of the keys ≥ 43 (in this case, to all of them). □

2.2 Applications of B-trees

The B-tree is a powerful tool for building indexes. The sequence of pointers at the leaves of a B-tree can play the role of any of the pointer sequences coming out of an index file that we learned about in Section 1. Here are some examples:

1. The search key of the B-tree is the primary key for the data file, and the index is dense. That is, there is one key-pointer pair in a leaf for every record of the data file. The data file may or may not be sorted by primary key.

2. The data file is sorted by its primary key, and the B-tree is a sparse index with one key-pointer pair at a leaf for each block of the data file.

3. The data file is sorted by an attribute that is not a key, and this attribute is the search key for the B-tree. For each key value K that appears in the data file there is one key-pointer pair at a leaf. That pointer goes to the first of the records that have K as their sort-key value.

There are additional applications of B-tree variants that allow multiple occurrences of the search key[3] at the leaves. Figure 14 suggests what such a B-tree might look like.

If we do allow duplicate occurrences of a search key, then we need to change slightly the definition of what the keys at interior nodes mean, which we discussed in Section 2.1. Now, suppose there are keys $K_1, K_2, \ldots, K_n$ at an interior node. Then K_i will be the smallest new key that appears in the part of the subtree accessible from the $(i+1)$st pointer. By "new," we mean that there are no occurrences of K_i in the portion of the tree to the left of the $(i+1)$st subtree, but at least one occurrence of K_i in that subtree. Note that in some situations, there will be no such key, in which case K_i can be taken to be null. Its associated pointer is still necessary, as it points to a significant portion of the tree that happens to have only one key value within it.

[3]Remember that a "search key" is not necessarily a "key" in the sense of being unique.

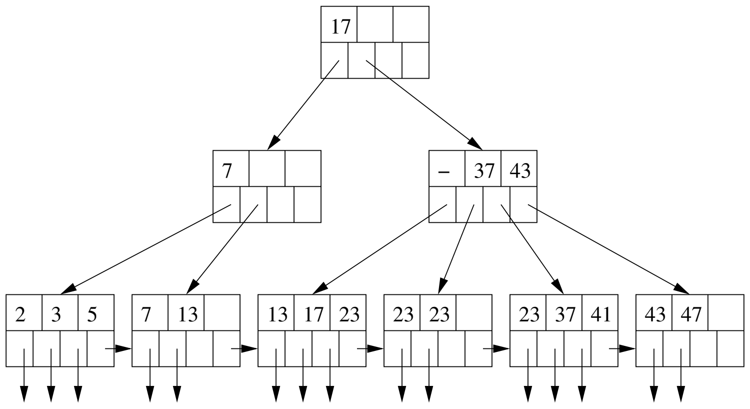

Figure 14: A B-tree with duplicate keys

Example 13: Figure 14 shows a B-tree similar to Fig. 13, but with duplicate values. In particular, key 11 has been replaced by 13, and keys 19, 29, and 31 have all been replaced by 23. As a result, the key at the root is 17, not 13. The reason is that, although 13 is the lowest key in the second subtree of the root, it is not a *new* key for that subtree, since it also appears in the first subtree.

We also had to make some changes to the second child of the root. The second key is changed to 37, since that is the first new key of the third child (fifth leaf from the left). Most interestingly, the first key is now null. The reason is that the second child (fourth leaf) has no new keys at all. Put another way, if we were searching for any key and reached the second child of the root, we would never want to start at its second child. If we are searching for 23 or anything lower, we want to start at its first child, where we will either find what we are looking for (if it is 17), or find the first of what we are looking for (if it is 23). Note that:

- We would not reach the second child of the root searching for 13; we would be directed at the root to its first child instead.
- If we are looking for any key between 24 and 36, we are directed to the third leaf, but when we don't find even one occurrence of what we are looking for, we know not to search further right. For example, if there were a key 24 among the leaves, it would either be on the 4th leaf, in which case the null key in the second child of the root would be 24 instead, or it would be in the 5th leaf, in which case the key 37 at the second child of the root would be 24.

□

2.3 Lookup in B-Trees

We now revert to our original assumption that there are no duplicate keys at the leaves. We also suppose that the B-tree is a dense index, so every search-key value that appears in the data file will also appear at a leaf. These assumptions make the discussion of B-tree operations simpler, but is not essential for these operations. In particular, modifications for sparse indexes are similar to the changes we introduced in Section 1.3 for indexes on sequential files.

Suppose we have a B-tree index and we want to find a record with search-key value K. We search for K recursively, starting at the root and ending at a leaf. The search procedure is:

BASIS: If we are at a leaf, look among the keys there. If the ith key is K, then the ith pointer will take us to the desired record.

INDUCTION: If we are at an interior node with keys $K_1, K_2, \ldots, K_n$, follow the rules given in Section 2.1 to decide which of the children of this node should next be examined. That is, there is only one child that could lead to a leaf with key K. If $K < K_1$, then it is the first child, if $K_1 \leq K < K_2$, it is the second child, and so on. Recursively apply the search procedure at this child.

Example 14: Suppose we have the B-tree of Fig. 13, and we want to find a record with search key 40. We start at the root, where there is one key, 13. Since $13 \leq 40$, we follow the second pointer, which leads us to the second-level node with keys 23, 31, and 43.

At that node, we find $31 \leq 40 < 43$, so we follow the third pointer. We are thus led to the leaf with keys 31, 37, and 41. If there had been a record in the data file with key 40, we would have found key 40 at this leaf. Since we do not find 40, we conclude that there is no record with key 40 in the underlying data.

Note that had we been looking for a record with key 37, we would have taken exactly the same decisions, but when we got to the leaf we would find key 37. Since it is the second key in the leaf, we follow the second pointer, which will lead us to the data record with key 37. □

2.4 Range Queries

B-trees are useful not only for queries in which a single value of the search key is sought, but for queries in which a range of values are asked for. Typically, *range queries* have a term in the `WHERE`-clause that compares the search key with a value or values, using one of the comparison operators other than `=` or `<>`. Examples of range queries using a search-key attribute k are:

```
SELECT * FROM R   SELECT * FROM R
WHERE R.k > 40;   WHERE R.k >= 10 AND R.k <= 25;
```

If we want to find all keys in the range $[a, b]$ at the leaves of a B-tree, we do a lookup to find the key a. Whether or not it exists, we are led to a leaf where

a could be, and we search the leaf for keys that are a or greater. Each such key we find has an associated pointer to one of the records whose key is in the desired range. As long as we do not find a key greater than b in the current block, we follow the pointer to the next leaf and repeat our search for keys in the range $[a, b]$.

The above search algorithm also works if b is infinite; i.e., there is only a lower bound and no upper bound. In that case, we search all the leaves from the one that would hold key a to the end of the chain of leaves. If a is $-\infty$ (that is, there is an upper bound on the range but no lower bound), then the search for "minus infinity" as a search key will always take us to the first leaf. The search then proceeds as above, stopping only when we pass the key b.

Example 15: Suppose we have the B-tree of Fig. 13, and we are given the range $(10, 25)$ to search for. We look for key 10, which leads us to the second leaf. The first key is less than 10, but the second, 11, is at least 10. We follow its associated pointer to get the record with key 11.

Since there are no more keys in the second leaf, we follow the chain to the third leaf, where we find keys 13, 17, and 19. All are less than or equal to 25, so we follow their associated pointers and retrieve the records with these keys. Finally, we move to the fourth leaf, where we find key 23. But the next key of that leaf, 29, exceeds 25, so we are done with our search. Thus, we have retrieved the five records with keys 11 through 23. □

2.5 Insertion Into B-Trees

We see some of the advantages of B-trees over simpler multilevel indexes when we consider how to insert a new key into a B-tree. The corresponding record will be inserted into the file being indexed by the B-tree, using any of the methods discussed in Section 1; here we consider how the B-tree changes. The insertion is, in principle, recursive:

- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level. We may thus recursively apply this strategy to insert at the next level: if there is room, insert it; if not, split the parent node and continue up the tree.
- As an exception, if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level; the new root has the two nodes resulting from the split as its children. Recall that no matter how large n (the number of slots for

keys at a node) is, it is always permissible for the root to have only one key and two children.

When we split a node and insert it into its parent, we need to be careful how the keys are managed. First, suppose N is a leaf whose capacity is n keys. Also suppose we are trying to insert an $(n+1)$st key and its associated pointer. We create a new node M, which will be the sibling of N, immediately to its right. The first $\lceil (n+1)/2 \rceil$ key-pointer pairs, in sorted order of the keys, remain with N, while the other key-pointer pairs move to M. Note that both nodes N and M are left with a sufficient number of key-pointer pairs — at least $\lfloor (n+1)/2 \rfloor$ pairs.

Now, suppose N is an interior node whose capacity is n keys and $n+1$ pointers, and N has just been assigned $n+2$ pointers because of a node splitting below. We do the following:

1. Create a new node M, which will be the sibling of N, immediately to its right.

2. Leave at N the first $\lceil (n+2)/2 \rceil$ pointers, in sorted order, and move to M the remaining $\lfloor (n+2)/2 \rfloor$ pointers.

3. The first $\lceil n/2 \rceil$ keys stay with N, while the last $\lfloor n/2 \rfloor$ keys move to M. Note that there is always one key in the middle left over; it goes with neither N nor M. The leftover key K indicates the smallest key reachable via the first of M's children. Although this key doesn't appear in N or M, it is associated with M, in the sense that it represents the smallest key reachable via M. Therefore K will be inserted into the parent of N and M to divide searches between those two nodes.

Example 16 : Let us insert key 40 into the B-tree of Fig. 13. We find the proper leaf for the insertion by the lookup procedure of Section 2.3. As found in Example 14, the insertion goes into the fifth leaf. Since this leaf now has four key-pointer pairs — 31, 37, 40, and 41 — we need to split the leaf. Our first step is to create a new node and move the highest two keys, 40 and 41, along with their pointers, to that node. Figure 15 shows this split.

Notice that although we now show the nodes on four ranks to save space, there are still only three levels to the tree. The seven leaves are linked by their last pointers, which still form a chain from left to right.

We must now insert a pointer to the new leaf (the one with keys 40 and 41) into the node above it (the node with keys 23, 31, and 43). We must also associate with this pointer the key 40, which is the least key reachable through the new leaf. Unfortunately, the parent of the split node is already full; it has no room for another key or pointer. Thus, it too must be split.

We start with pointers to the last five leaves and the list of keys representing the least keys of the last four of these leaves. That is, we have pointers P_1, P_2, P_3, P_4, P_5 to the leaves whose least keys are 13, 23, 31, 40, and 43, and

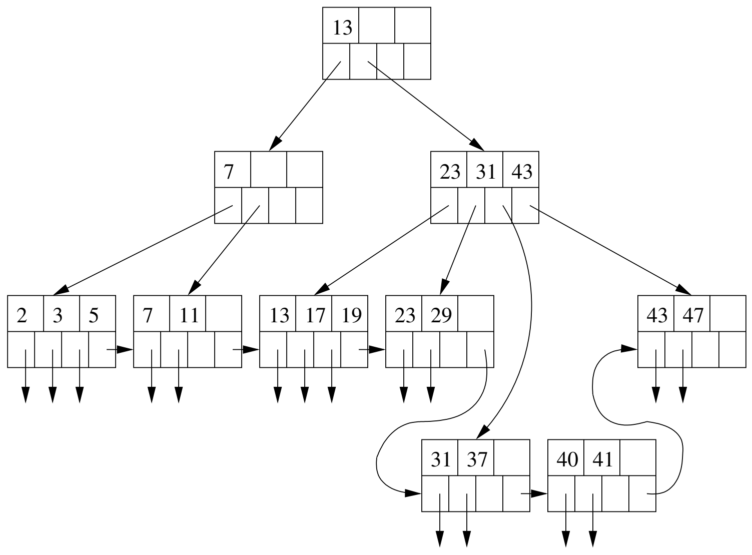

Figure 15: Beginning the insertion of key 40

we have the key sequence 23, 31, 40, 43 to separate these pointers. The first three pointers and first two keys remain with the split interior node, while the last two pointers and last key go to the new node. The remaining key, 40, represents the least key accessible via the new node.

Figure 16 shows the completion of the insert of key 40. The root now has three children; the last two are the split interior node. Notice that the key 40, which marks the lowest of the keys reachable via the second of the split nodes, has been installed in the root to separate the keys of the root's second and third children. □

2.6 Deletion From B-Trees

If we are to delete a record with a given key K, we must first locate that record and its key-pointer pair in a leaf of the B-tree. This part of the deletion process is essentially a lookup, as in Section 2.3. We then delete the record itself from the data file, and we delete the key-pointer pair from the B-tree.

If the B-tree node from which a deletion occurred still has at least the minimum number of keys and pointers, then there is nothing more to be done.[4] However, it is possible that the node was right at the minimum occupancy before the deletion, so after deletion the constraint on the number of keys is

[4]If the data record with the least key at a leaf is deleted, then we have the option of raising the appropriate key at one of the ancestors of that leaf, but there is no requirement that we do so; all searches will still go to the appropriate leaf.

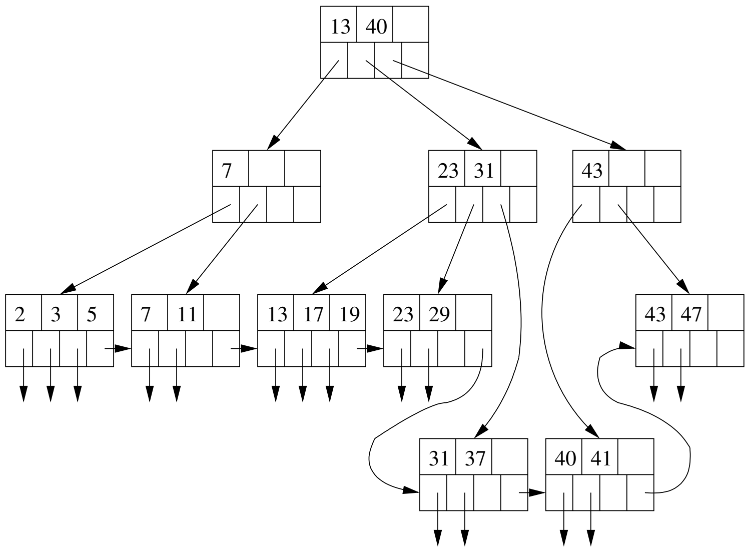

Figure 16: Completing the insertion of key 40

violated. We then need to do one of two things for a node N whose contents are subminimum; one case requires a recursive deletion up the tree:

1. If one of the adjacent siblings of node N has more than the minimum number of keys and pointers, then one key-pointer pair can be moved to N, keeping the order of keys intact. Possibly, the keys at the parent of N must be adjusted to reflect the new situation. For instance, if the right sibling of N, say node M, provides an extra key and pointer, then it must be the smallest key that is moved from M to N. At the parent of M and N, there is a key that represents the smallest key accessible via M; that key must be increased to reflect the new M.

2. The hard case is when neither adjacent sibling can be used to provide an extra key for N. However, in that case, we have two adjacent nodes, N and a sibling M; the latter has the minimum number of keys and the former has fewer than the minimum. Therefore, together they have no more keys and pointers than are allowed in a single node. We merge these two nodes, effectively deleting one of them. We need to adjust the keys at the parent, and then delete a key and pointer at the parent. If the parent is still full enough, then we are done. If not, then we recursively apply the deletion algorithm at the parent.

Example 17: Let us begin with the original B-tree of Fig. 13, before the insertion of key 40. Suppose we delete key 7. This key is found in the second leaf. We delete it, its associated pointer, and the record that pointer points to.

The second leaf now has only one key, and we need at least two in every leaf. But we are saved by the sibling to the left, the first leaf, because that leaf has an extra key-pointer pair. We may therefore move the highest key, 5, and its associated pointer to the second leaf. The resulting B-tree is shown in Fig. 17. Notice that because the lowest key in the second leaf is now 5, the key in the parent of the first two leaves has been changed from 7 to 5.

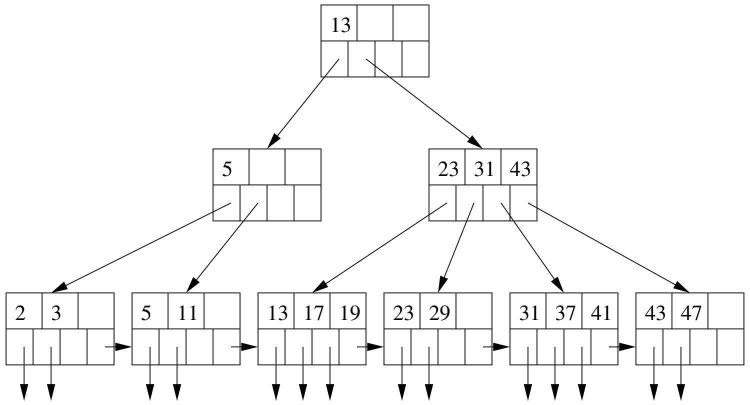

Figure 17: Deletion of key 7

Next, suppose we delete key 11. This deletion has the same effect on the second leaf; it again reduces the number of its keys below the minimum. This time, however, we cannot take a key from the first leaf, because the latter is down to the minimum number of keys. Additionally, there is no sibling to the right from which to take a key.[5] Thus, we need to merge the second leaf with a sibling, namely the first leaf.

The three remaining key-pointer pairs from the first two leaves fit in one leaf, so we move 5 to the first leaf and delete the second leaf. The pointers and keys in the parent are adjusted to reflect the new situation at its children; specifically, the two pointers are replaced by one (to the remaining leaf) and the key 5 is no longer relevant and is deleted. The situation is now as shown in Fig. 18.

The deletion of a leaf has adversely affected the parent, which is the left child of the root. That node, as we see in Fig. 18, now has no keys and only one pointer. Thus, we try to obtain an extra key and pointer from an adjacent sibling. This time we have the easy case, since the other child of the root can afford to give up its smallest key and a pointer.

The change is shown in Fig. 19. The pointer to the leaf with keys 13, 17,

[5]Notice that the leaf to the right, with keys 13, 17, and 19, is not a sibling, because it has a different parent. We could take a key from that node anyway, but then the algorithm for adjusting keys throughout the tree becomes more complex. We leave this enhancement as an exercise.

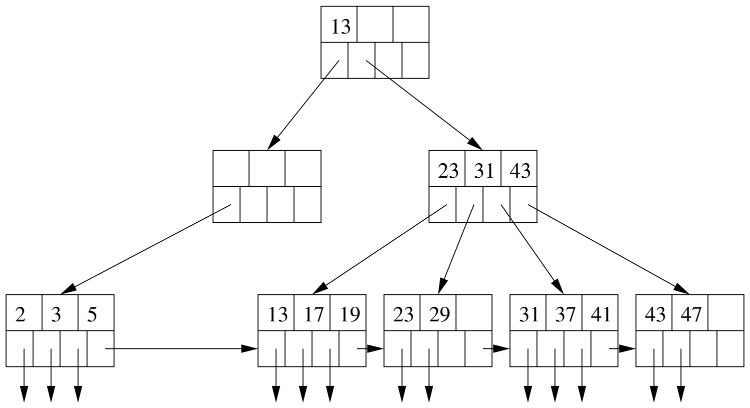

Figure 18: Beginning the deletion of key 11

and 19 has been moved from the second child of the root to the first child. We have also changed some keys at the interior nodes. The key 13, which used to reside at the root and represented the smallest key accessible via the pointer that was transferred, is now needed at the first child of the root. On the other hand, the key 23, which used to separate the first and second children of the second child of the root now represents the smallest key accessible from the second child of the root. It therefore is placed at the root itself. □

2.7 Efficiency of B-Trees

B-trees allow lookup, insertion, and deletion of records using very few disk I/O's per file operation. First, we should observe that if n, the number of keys per block, is reasonably large, then splitting and merging of blocks will be rare events. Further, when such an operation is needed, it almost always is limited to the leaves, so only two leaves and their parent are affected. Thus, we can essentially neglect the disk-I/O cost of B-tree reorganizations.

However, every search for the record(s) with a given search key requires us to go from the root down to a leaf, to find a pointer to the record. Since we are only reading B-tree blocks, the number of disk I/O's will be the number of levels the B-tree has, plus the one (for lookup) or two (for insert or delete) disk I/O's needed for manipulation of the record itself. We must thus ask: how many levels does a B-tree have? For the typical sizes of keys, pointers, and blocks, three levels are sufficient for all but the largest databases. Thus, we shall generally take 3 as the number of levels of a B-tree. The following example illustrates why.

Example 18 : Recall our analysis in Example 10, where we determined that 340 key-pointer pairs could fit in one block for our example data. Suppose

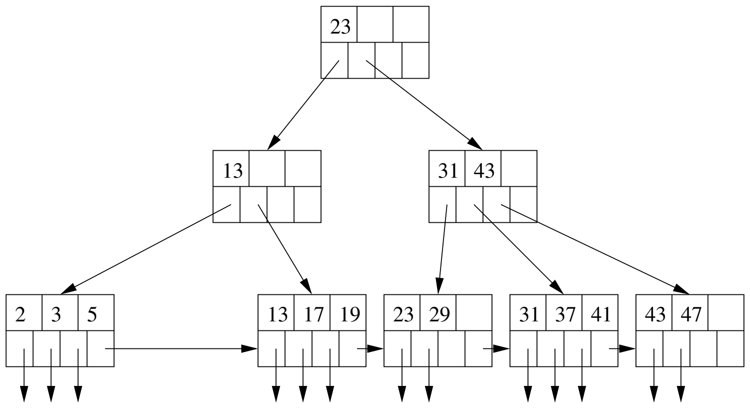

Figure 19: Completing the deletion of key 11

that the average block has an occupancy midway between the minimum and maximum, i.e., a typical block has 255 pointers. With a root, 255 children, and $255^2 = 65025$ leaves, we shall have among those leaves 255^3, or about 16.6 million pointers to records. That is, files with up to 16.6 million records can be accommodated by a 3-level B-tree. □

However, we can use even fewer than three disk I/O's per search through the B-tree. The root block of a B-tree is an excellent choice to keep permanently buffered in main memory. If so, then every search through a 3-level B-tree requires only two disk reads. In fact, under some circumstances it may make sense to keep second-level nodes of the B-tree buffered in main memory as well, reducing the B-tree search to a single disk I/O, plus whatever is necessary to manipulate the blocks of the data file itself.

2.8 Exercises for Section 2

Exercise 2.1 : Suppose that blocks can hold either ten records or 99 keys and 100 pointers. Also assume that the average B-tree node is 70% full; i.e., it will have 69 keys and 70 pointers. We can use B-trees as part of several different structures. For each structure described below, determine (*i*) the total number of blocks needed for a 1,000,000-record file, and (*ii*) the average number of disk I/O's to retrieve a record given its search key. You may assume nothing is in memory initially, and the search key is the primary key for the records.

a) The data file is a sequential file, sorted on the search key, with 10 records per block. The B-tree is a dense index.

b) The same as (a), but the data file consists of records in no particular order, packed 10 to a block.

Should We Delete From B-Trees?

There are B-tree implementations that don't fix up deletions at all. If a leaf has too few keys and pointers, it is allowed to remain as it is. The rationale is that most files grow on balance, and while there might be an occasional deletion that makes a leaf become subminimum, the leaf will probably soon grow again and attain the minimum number of key-pointer pairs once again.

Further, if records have pointers from outside the B-tree index, then we need to replace the record by a "tombstone," and we don't want to delete its pointer from the B-tree anyway. In certain circumstances, when it can be guaranteed that all accesses to the deleted record will go through the B-tree, we can even leave the tombstone in place of the pointer to the record at a leaf of the B-tree. Then, space for the record can be reused.

c) The same as (a), but the B-tree is a sparse index.

! d) Instead of the B-tree leaves having pointers to data records, the B-tree leaves hold the records themselves. A block can hold ten records, but on average, a leaf block is 70% full; i.e., there are seven records per leaf block.

e) The data file is a sequential file, and the B-tree is a sparse index, but each primary block of the data file has one overflow block. On average, the primary block is full, and the overflow block is half full. However, records are in no particular order within a primary block and its overflow block.

Exercise 2.2 : Repeat Exercise 2.1 in the case that the query is a range query that is matched by 1000 records.

Exercise 2.3 : Suppose pointers are 4 bytes long, and keys are 12 bytes long. How many keys and pointers will a block of 16,384 bytes have?

Exercise 2.4 : What are the minimum numbers of keys and pointers in B-tree (*i*) interior nodes and (*ii*) leaves, when:

a) $n = 10$; i.e., a block holds 10 keys and 11 pointers.

b) $n = 11$; i.e., a block holds 11 keys and 12 pointers.

Exercise 2.5 : Execute the following operations on Fig. 13. Describe the changes for operations that modify the tree.

a) Lookup the record with key 41.

b) Lookup the record with key 40.

c) Lookup all records in the range 20 to 30.

d) Lookup all records with keys less than 30.

e) Lookup all records with keys greater than 30.

f) Insert a record with key 1.

g) Insert records with keys 14 through 16.

h) Delete the record with key 23.

i) Delete all the records with keys 23 and higher.

Exercise 2.6 : When duplicate keys are allowed in a B-tree, there are some necessary modifications to the algorithms for lookup, insertion, and deletion that we described in this section. Give the changes for: (a) lookup (b) insertion (c) deletion.

! Exercise 2.7 : In Example 17 we suggested that it would be possible to borrow keys from a nonsibling to the right (or left) if we used a more complicated algorithm for maintaining keys at interior nodes. Describe a suitable algorithm that rebalances by borrowing from adjacent nodes at a level, regardless of whether they are siblings of the node that has too many or too few key-pointer pairs.

! Exercise 2.8 : If we use the 3-key, 4-pointer nodes of our examples in this section, how many different B-trees are there when the data file has the following numbers of records: (a) 6 (b) 10 **!!** (c) 15.

! Exercise 2.9 : Suppose we have B-tree nodes with room for three keys and four pointers, as in the examples of this section. Suppose also that when we split a leaf, we divide the pointers 2 and 2, while when we split an interior node, the first 3 pointers go with the first (left) node, and the last 2 pointers go with the second (right) node. We start with a leaf containing pointers to records with keys 1, 2, and 3. We then add in order, records with keys 4, 5, 6, and so on. At the insertion of what key will the B-tree first reach four levels?

3 Hash Tables

There are a number of data structures involving a hash table that are useful as indexes. We assume the reader has seen the hash table used as a main-memory data structure. In such a structure there is a *hash function* h that takes a search key (the *hash key*) as an argument and computes from it an integer in the range 0 to $B-1$, where B is the number of *buckets*. A *bucket array*, which is an array indexed from 0 to $B-1$, holds the headers of B linked lists, one for each bucket of the array. If a record has search key K, then we store the record by linking it to the bucket list for the bucket numbered $h(K)$.