CHAPTER 12

Indexing and Hashing

Exercises

12.2 Answer: Reasons for not keeping several search indices include:

a. Every index requires additional CPU time and disk I/O overhead during inserts and deletions.

b. Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).

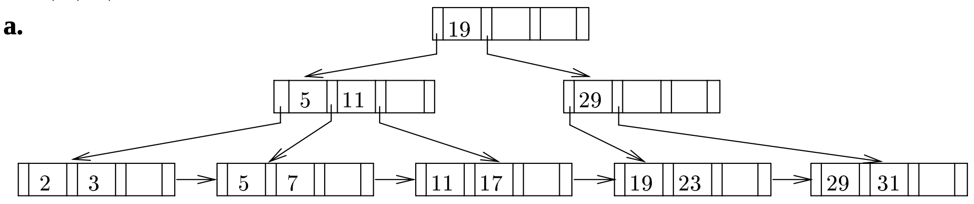
c. Each extra index requires additional storage space.

d. For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore database performance is improved less by adding indices when many indices already exist.

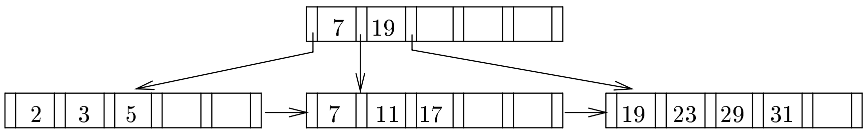
12.4 Answer: In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient.

12.5 Answer: The following were generated by inserting values into the B$^+$-tree in ascending order. A node (other than the root) was never allowed to have fewer than $\lceil n/2 \rceil$ values/pointers.

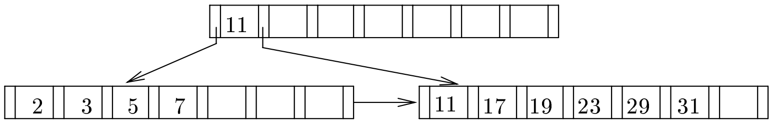
a.



b.



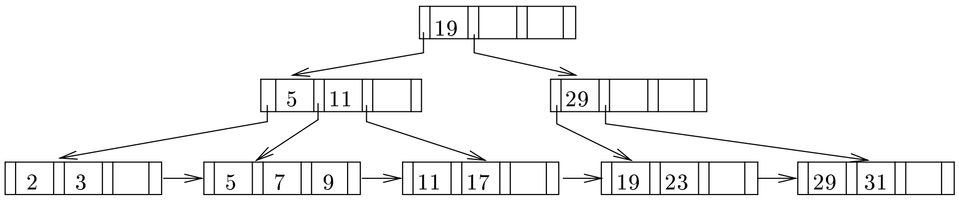
c.



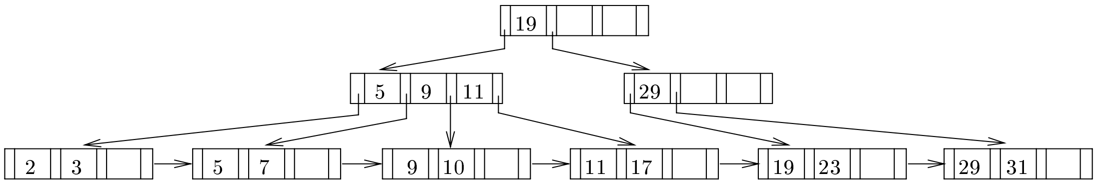
12.7 Answer:

- With structure 12.5.a:

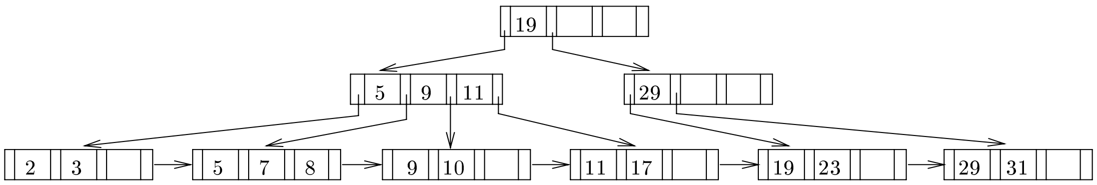
  Insert 9:



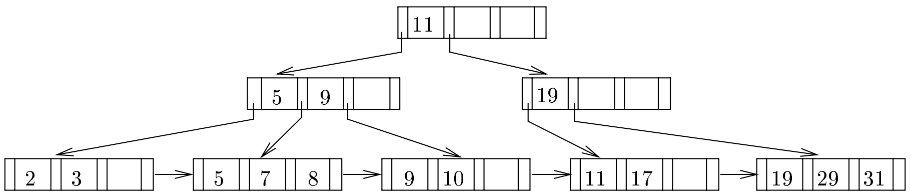
  Insert 10:



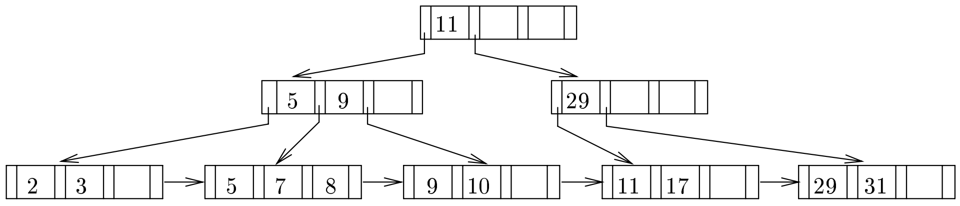
  Insert 8:



  Delete 23:

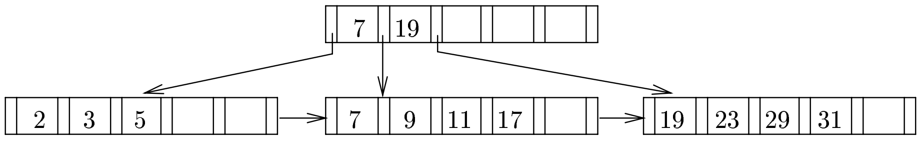


  Delete 19:

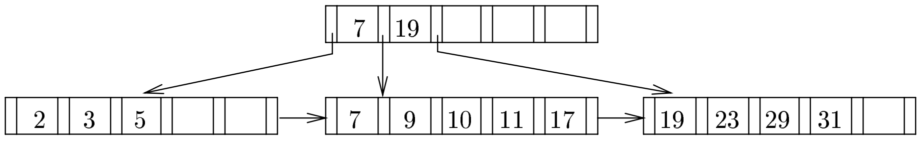


- With structure 12.5.b:

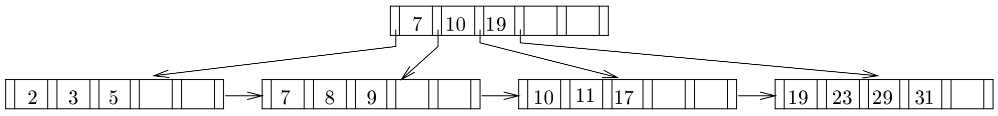
  Insert 9:



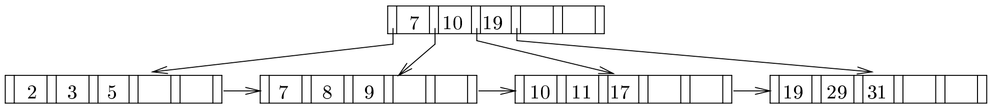
  Insert 10:



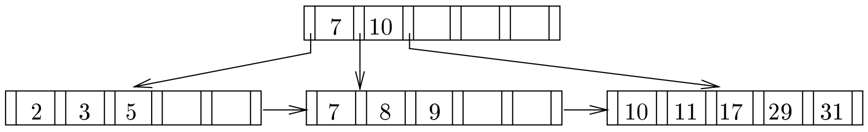
  Insert 8:



  Delete 23:

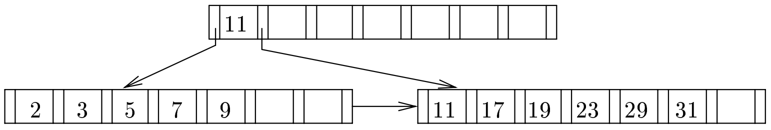


  Delete 19:

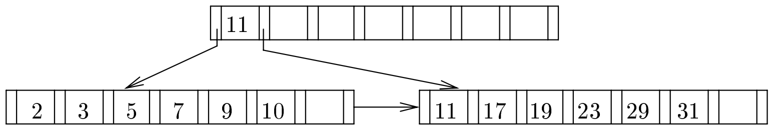


- With structure 12.5.c:

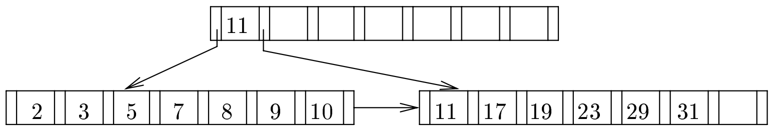
  Insert 9:



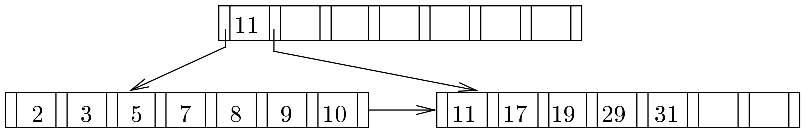
  Insert 10:



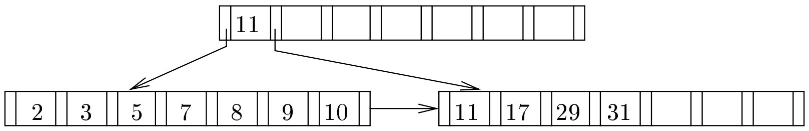
Insert 8:



Delete 23:



Delete 19:



12.8 Answer: If there are K search-key values and $m - 1$ siblings are involved in the redistribution, the expected height of the tree is: $log_{\lfloor (m-1)n/m \rfloor}(K)$

12.9 Answer: The algorithm for insertion into a B-tree is:

Locate the leaf node into which the new key-pointer pair should be inserted. If there is space remaining in that leaf node, perform the insertion at the correct location, and the task is over. Otherwise insert the key-pointer pair conceptually into the correct location in the leaf node, and then split it along the middle. The middle key-pointer pair does not go into either of the resultant nodes of the split operation. Instead it is inserted into the parent node, along with the tree pointer to the new child. If there is no space in the parent, a similar procedure is repeated.

The deletion algorithm is:

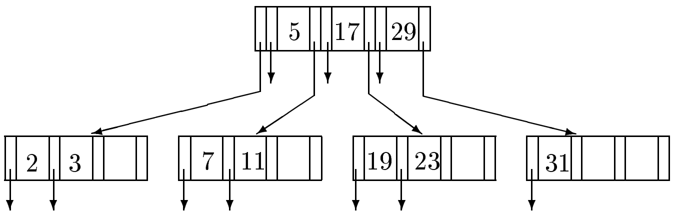
Locate the key value to be deleted, in the B-tree.

a. If it is found in a leaf node, delete the key-pointer pair, and the record from the file. If the leaf node contains less than $\lceil n/2 \rceil - 1$ entries as a result of this deletion, it is either merged with its siblings, or some entries are redistributed to it. Merging would imply a deletion, whereas redistribution would imply change(s) in the parent node's entries. The deletions may ripple upto the root of the B-tree.

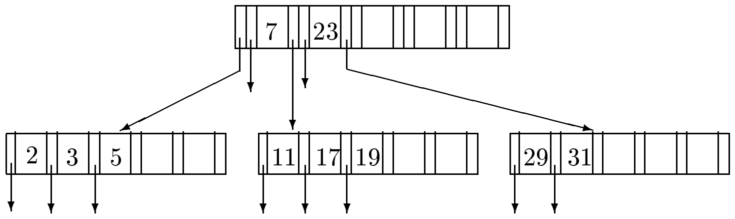
b. If the key value is found in an internal node of the B-tree, replace it and its record pointer by the smallest key value in the subtree immediately to its right and the corresponding record pointer. Delete the actual record in the database file. Then delete that smallest key value-pointer pair from the subtree. This deletion may cause further rippling deletions till the root of the B-tree.

Below are the B-trees we will get after insertion of the given key values. We assume that leaf and non-leaf nodes hold the same number of search key values.

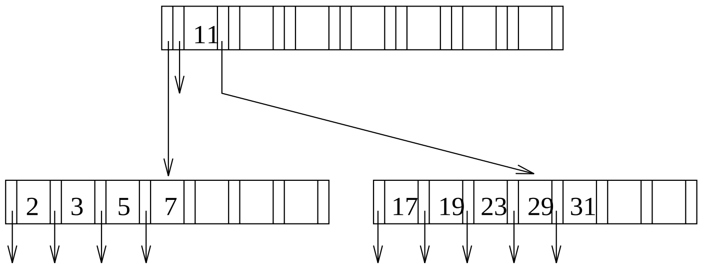
a.



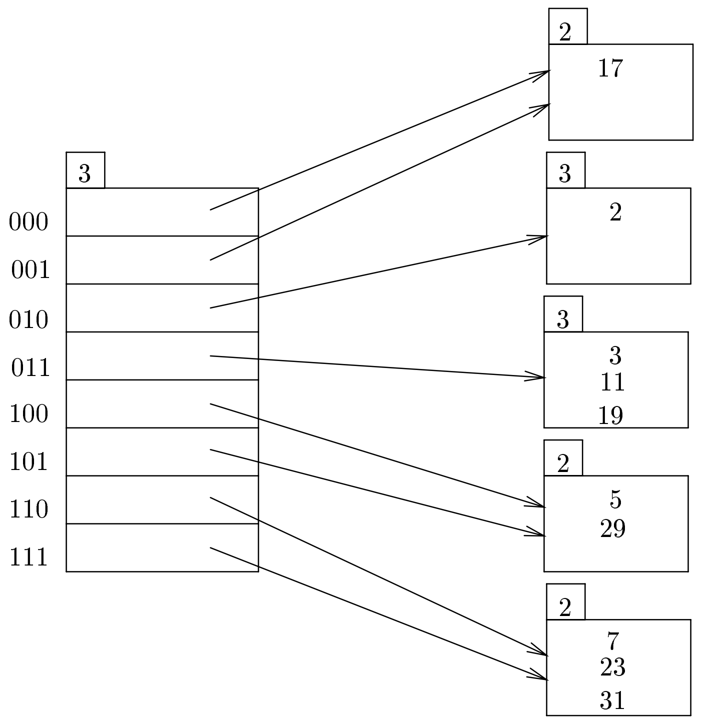
b.



c.

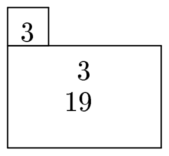


12.12 Answer:



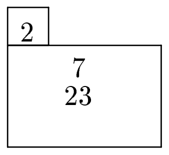
12.13 Answer:

a. Delete 11: From the answer to Exercise 12.12, change the third bucket to:

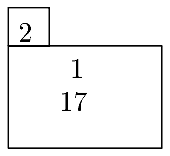


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

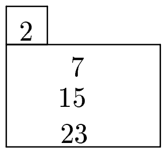
b. Delete 31: From the answer to 12.12, change the last bucket to:



c. Insert 1: From the answer to 12.12, change the first bucket to:



d. Insert 15: From the answer to 12.12, change the last bucket to:



12.14 Answer: Let i denote the number of bits of the hash value used in the hash table. Let **BSIZE** denote the maximum capacity of each bucket.

```
delete(value $K_l$)
begin
    $j$ = first $i$ high-order bits of $h(K_l)$;
    delete value $K_l$ from bucket $j$;
    coalesce(bucket $j$);
end

coalesce(bucket $j$)
begin
    $i_j$ = bits used in bucket $j$;
    $k$ = any bucket with first $(i_j - 1)$ bits same as that
        of bucket $j$ while the bit $i_j$ is reversed;
    $i_k$ = bits used in bucket $k$;
    if($i_j \neq i_k$)
        return; /* buckets cannot be merged */
    if(entries in $j$ + entries in $k > $ BSIZE)
        return; /* buckets cannot be merged */
    move entries of bucket $k$ into bucket $j$;

    decrease the value of $i_j$ by 1;
    make all the bucket-address-table entries,
    which pointed to bucket $k$, point to $j$;

    coalesce(bucket $j$);
end
```

Note that we can only merge two buckets at a time. The common hash prefix of the resultant bucket will have length one less than the two buckets merged. Hence we look at the buddy bucket of bucket j differing from it only at the last bit. If the common hash prefix of this bucket is not i_j, then this implies that the buddy bucket has been further split and merge is not possible.

When merge is successful, further merging may be possible, which is handled by a recursive call to *coalesce* at the end of the function.

12.15 Answer: If the hash table is currently using i bits of the hash value, then maintain a count of buckets for which the length of common hash prefix is exactly i.

Consider a bucket j with length of common hash prefix i_j. If the bucket is being split, and i_j is equal to i, then reset the count to 1. If the bucket is being split and i_j is one less that i, then increase the count by 1. It the bucket if being coalesced, and i_j is equal to i then decrease the count by 1. If the count becomes 0, then the bucket address table can be reduced in size at that point.

However, note that if the bucket address table is not reduced at that point, then the count has no significance afterwards. If we want to postpone the reduction, we have to keep an array of counts, i.e. a count for each value of common hash prefix. The array has to be updated in a similar fashion. The bucket address table can be reduced if the i^{th} entry of the array is 0, where i is the number of bits the table is using. Since bucket table reduction is an expensive operation, it is not always advisable to reduce the table. It should be reduced only when sufficient number of entries at the end of count array become 0.

12.18 Answer: We reproduce the account relation of Figure 12.25 below.

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Bitmaps for *branch-name*

Brighton	1	0	0	0	0	0	0	0	0
Downtown	0	1	1	0	0	0	0	0	0
Mianus	0	0	0	1	0	0	0	0	0
Perryridge	0	0	0	0	1	1	1	0	0
Redwood	0	0	0	0	0	0	0	1	0
Round hill	0	0	0	0	0	0	0	0	1

Bitmaps for *balance*

L_1	0	0	0	0	0	0	0	0	0
L_2	0	0	0	0	1	0	0	0	1
L_3	0	1	1	1	0	0	1	1	0
L_4	1	0	0	0	0	1	0	0	0

where, level L_1 is below 250, level L_2 is from 250 to below 500, L_3 from 500 to below 750 and level L_4 is above 750.

To find all accounts in Downtown with a balance of 500 or more, we find the union of bitmaps for levels L_3 and L_4 and then intersect it with the bitmap for Downtown.

Downtown	0	1	1	0	0	0	0	0	0
L_3	0	1	1	1	0	0	1	1	0
L_4	1	0	0	0	0	1	0	0	0
$L_3 \cup L_4$	1	1	1	1	0	1	1	1	0
Downtown	0	1	1	0	0	0	0	0	0
Downtown $\cap (L_3 \cup L_4)$	0	1	1	0	0	0	0	0	0

Thus, the required tuples are A-101 and A-110.