

# CHAPTER 14



## Indexing

Solutions for the Practice Exercises of Chapter 14

### Practice Exercises

#### 14.1

**Answer:**

Reasons for not keeping indices on every attribute include:

- Every index requires additional CPU time and disk I/O overhead during inserts and deletions.
- 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).
- Each extra index requires additional storage space.
- 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.

#### 14.2

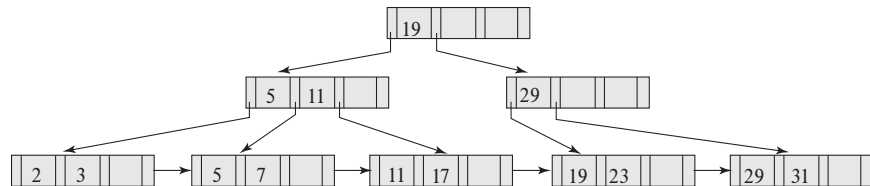
**Answer:**

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 the 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.

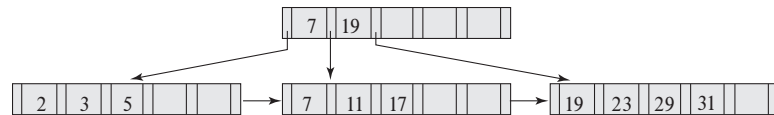
**14.3****Answer:**

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

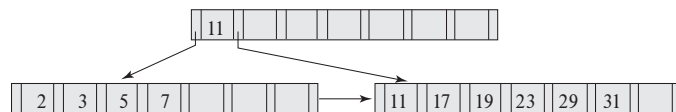
a.



b.

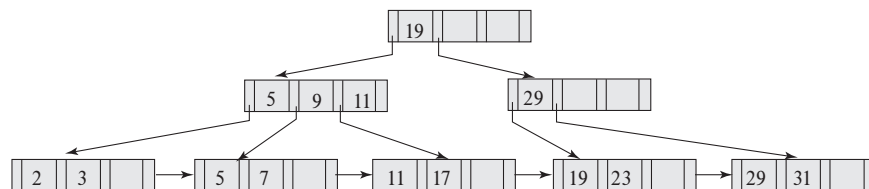


c.

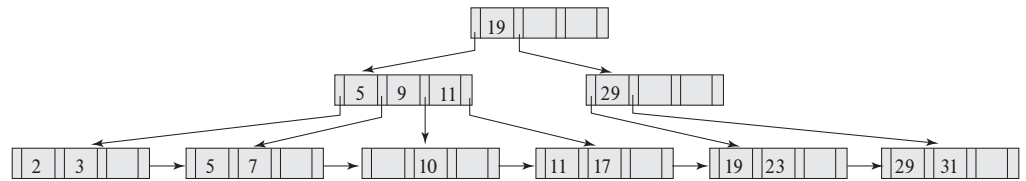
**14.4****Answer:**

- With structure Exercise 14.3.a:

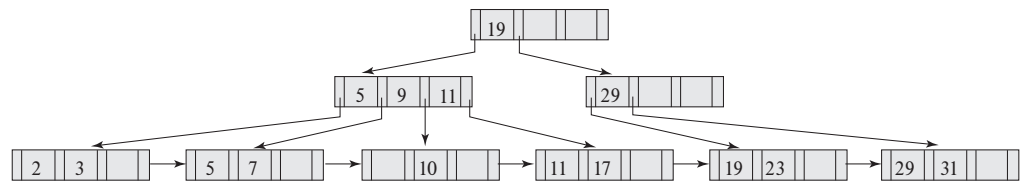
Insert 9:



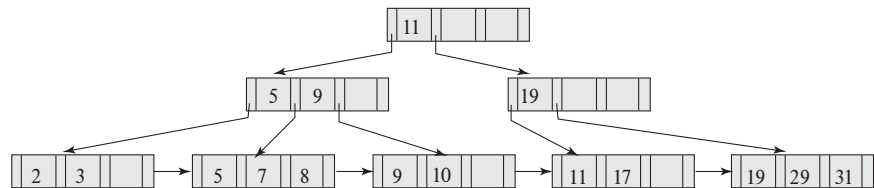
Insert 10:



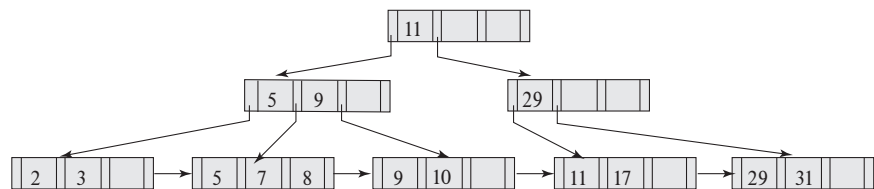
Insert 8:



Delete 23:

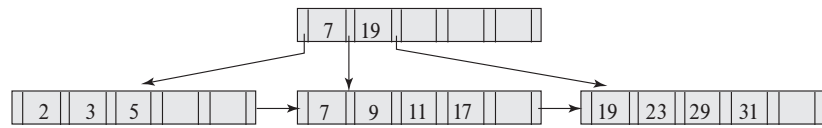


Delete 19:

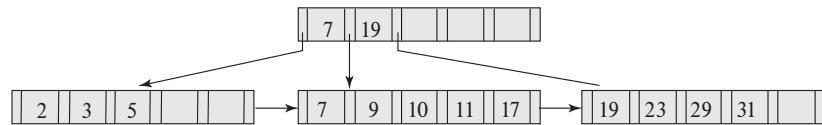


- With structure Exercise 14.3.b:

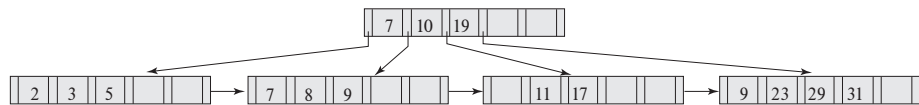
Insert 9:



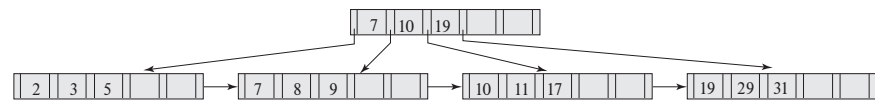
Insert 10:



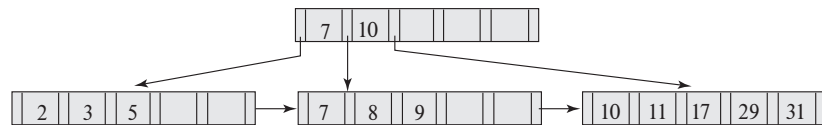
Insert 8:



Delete 23:

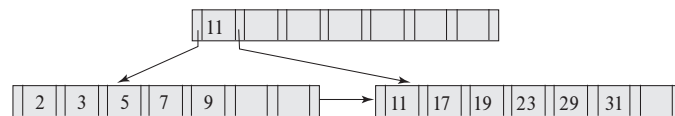


Delete 19:

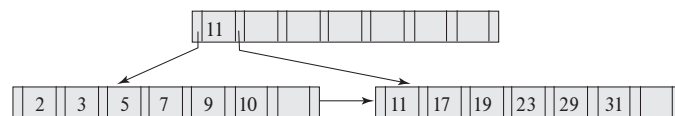


- With structure Exercise 14.3.c:

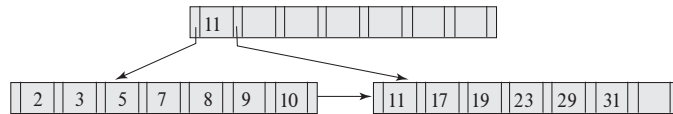
Insert 9:



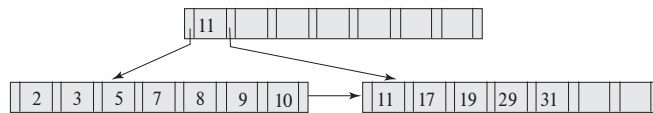
Insert 10:



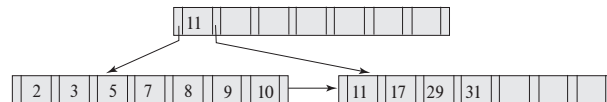
Insert 8:



Delete 23:



Delete 19:



14.5

**Answer:**

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

14.6

**Answer:**

FILL IN

14.7

**Answer:**

If the index entries are inserted in ascending order, the new entries get directed to the last leaf node. When this leaf node gets filled, it is split into two. Of the two nodes generated by the split, the left node is left untouched and the insertions take place on the right node. This makes the occupancy of the leaf nodes about 50 percent except for the last leaf.

If keys that are inserted are sorted in descending order, the above situation would still occur, but symmetrically, with the right node of a split never getting touched again, and occupancy would again be 50 percent for all nodes other than the first leaf.

14.8

**Answer:**

- The cost to locate the page number of the required leaf page for an insertion is negligible since the non-leaf nodes are in memory. On the leaf

level it takes one random disk access to read and one random disk access to update it along with the cost to write one page. Insertions which lead to splitting of leaf nodes require an additional page write. Hence to build a B<sup>+</sup>-tree with  $n_r$  entries it takes a maximum of  $2 * n_r$  random disk accesses and  $n_r + 2 * (n_r/f)$  page writes. The second part of the cost comes from the fact that in the worst case each leaf is half filled, so the number of splits that occur is twice  $n_r/f$ .

The above formula ignores the cost of writing non-leaf nodes, since we assume they are in memory, but in reality they would also be written eventually. This cost is closely approximated by  $2 * (n_r/f)/f$ , which is the number of internal nodes just above the leaf; we can add further terms to account for higher levels of nodes, but these are much smaller than the number of leaves and can be ignored.

- b. Substituting the values in the above formula and neglecting the cost for page writes, it takes about  $10,000,000 * 20$  milliseconds, or 56 hours, since each insertion costs 20 milliseconds.

c.

```

function insert_in_leaf(value  $K$ , pointer  $P$ )
    if (tree is empty) create an empty leaf node  $L$ , which is also the root
    else Find the last leaf node in the leaf nodes chain  $L$ 
    if ( $L$  has less than  $n - 1$  key values)
        then insert ( $K, P$ ) at the first available location in  $L$ 
    else begin
        Create leaf node  $L1$ 
        Set  $L.P_n = L1$ ;
        Set  $K1$  = last value from page  $L$ 
        insert_in_parent(1,  $L$ ,  $K1$ ,  $L1$ )
        insert ( $K, P$ ) at the first location in  $L1$ 
    end

```

```

function insert_in_parent(level  $l$ , pointer  $P$ , value  $K$ , pointer  $P1$ )
  if (level  $l$  is empty) then begin
    Create an empty non-leaf node  $N$ , which is also the root
    insert( $P$ ,  $K$ ,  $P1$ ) at the starting of the node  $N$ 
    return
  else begin
    Find the right most node  $N$  at level  $l$ 
    if ( $N$  has less than  $n$  pointers)
      then insert( $K$ ,  $P1$ ) at the first available location in  $N$ 
    else begin
      Create a new non-leaf page  $N1$ 
      insert ( $P1$ ) at the starting of the node  $N$ 
      insert_in_parent( $l + 1$ , pointer  $N$ , value  $K$ , pointer  $N1$ )
    end
  end

```

The insert\_in\_leaf function is called for each of the value, pointer pairs in ascending order. Similar function can also be built for descending order. The search for the last leaf or non-leaf node at any level can be avoided by storing the current last page details in an array.

The last node in each level might be less than half filled. To make this index structure meet the requirements of a B<sup>+</sup>-tree, we can redistribute the keys of the last two pages at each level. Since the last but one node is always full, redistribution makes sure that both of them are at least half filled.

#### 14.9

##### Answer:

- a. In a B<sup>+</sup>-tree index or file organization, leaf nodes that are adjacent to each other in the tree may be located at different places on disk. When a file organization is newly created on a set of records, it is possible to allocate blocks that are mostly contiguous on disk to leaf nodes that are contiguous in the tree. As insertions and deletions occur on the tree, sequentiality is increasingly lost, and sequential access has to wait for disk seeks increasingly often.
- b.
  - i. In the worst case, each  $n$ -block unit and each node of the B<sup>+</sup>-tree is half filled. This gives the worst-case occupancy as 25 percent.
  - ii. No. While splitting the  $n$ -block unit, the first  $n/2$  leaf pages are placed in one  $n$ -block unit and the remaining pages in the second  $n$ -block unit. That is, every  $n$ -block split maintains the order. Hence, the nodes in the  $n$ -block units are consecutive.

- iii. In the regular B<sup>+</sup>-tree construction, the leaf pages might not be sequential and hence in the worst-case, it takes one seek per leaf page. Using the block at a time method, for each  $n$ -node block, we will have at least  $n/2$  leaf nodes in it. Each  $n$ -node block can be read using one seek. Hence the worst-case seeks come down by a factor of  $n/2$ .
- iv. Allowing redistribution among the nodes of the same block does not require additional seeks, whereas in regular B<sup>+</sup>-trees we require as many seeks as the number of leaf pages involved in the redistribution. This makes redistribution for leaf blocks efficient with this scheme. Also, the worst-case occupancy comes back to nearly 50 percent. (Splitting of leaf nodes is preferred when the participating leaf nodes are nearly full. Hence nearly 50 percent instead of exact 50 percent)

**14.10****Answer:**

Indices on any attributes on which there are selection conditions; if there are only a few distinct values for that attribute, a bitmap index may be created, otherwise a normal B<sup>+</sup>-tree index.

B<sup>+</sup>-tree indices on primary-key and foreign-key attributes.

Also indices on attributes that are involved in join conditions in the queries.

**14.11****Answer:**

If there have been no updates in a while, but there are a lot of index look ups on an index, then entries at one level, say  $i$ , can be merged into the next level, even if the level is not full. The benefit is that reads would then not have to look up indices at level  $i$ , reducing the cost of reads.

**14.12****Answer:**

The idea of buffer trees can be used with any tree-structured index to reduce the cost of inserts and updates, including spatial indices. In contrast, LSM trees can only be used with linearly ordered data that are amenable to merging. On the other hand, buffer trees require more random I/O to perform insert operations as compared to (all variants of) LSM trees.

Write-optimized indices can significantly reduce the cost of inserts, and to a lesser extent, of updates, as compared to B<sup>+</sup>-trees. On the other hand, the index lookup cost can be significantly higher for write-optimized indices as compared to B<sup>+</sup>-trees.

**14.13****Answer:**

We reproduce the instructor relation below.



<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- a. Bitmap for *salary*, with  $S_1$ ,  $S_2$ ,  $S_3$  and  $S_4$  representing the given intervals in the same order

$S_1$	0	0	1	0	0	0	0	0	0	0	0	0
$S_2$	0	0	0	0	0	0	0	0	0	0	0	0
$S_3$	1	0	0	0	1	0	0	1	0	0	0	0
$S_4$	0	1	0	1	0	1	1	0	1	1	1	1

- b. The question is a bit trivial if there is no bitmap on the *dept\_name* attribute. The bitmap for the *dept\_name* attribute is:

Comp. Sci.	1	0	0	0	0	0	1	0	0	0	1	0
Finance	0	1	0	0	0	0	0	0	1	0	0	0
Music	0	0	1	0	0	0	0	0	0	0	0	0
Physics	0	0	0	1	0	1	0	0	0	0	0	0
History	0	0	0	0	1	0	0	1	0	0	0	0
Biology	0	0	0	0	0	0	0	0	0	1	0	0
Elec. Eng.	0	0	0	0	0	0	0	0	0	0	0	1

To find all instructors in the Finance department with salary of 80000 or more, we first find the intersection of the Finance department bitmap and  $S_4$  bitmap of *salary* and then scan on these records for salary of 80000 or more.

Intersection of Finance department bitmap and  $S_4$  bitmap of *salary*.

$S_4$	0	1	0	1	0	1	1	0	1	1	1	1
Finance	0	1	0	0	0	0	0	0	1	0	0	0
$S_4 \cap \text{Finance}$	0	1	0	0	0	0	0	0	1	0	0	0

Scan on these records with salary 80000 or more gives Wu and Singh as the instructors who satisfy the given query.

**14.14**

**Answer:**

FILL IN

**14.15**

**Answer:**

Start with regions with very small radius, and retry with a larger radius if a particular region does not contain any result. For example, each time the radius could be increased by a factor of (say) 1.5. The benefit is that since we do not use a very large radius compared to the minimum radius required, there will (hopefully!) not be too many points in the circular range query result.