CHAPTER 11

Indexing and Hashing

Practice Exercises

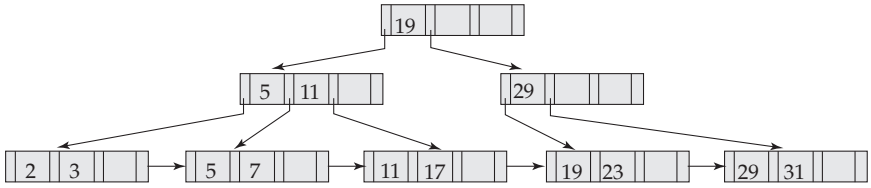
11.1 **Answer:** Reasons for not keeping indices on every attribute include:

- 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.

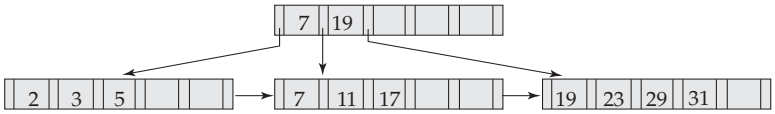
11.2 **Answer:** In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all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

11.3 **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.

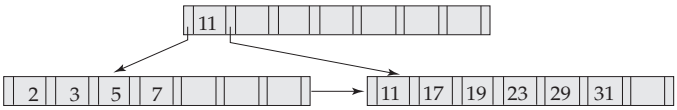
a.



b.



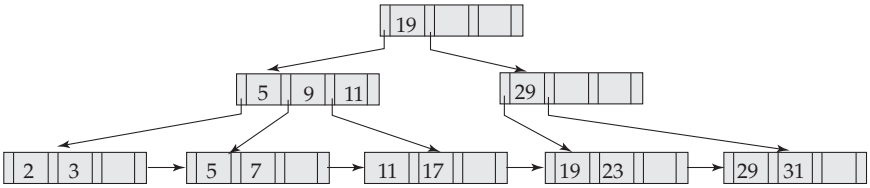
c.



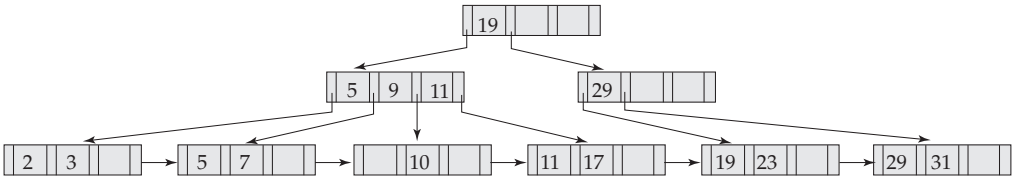
11.4 Answer:

- With structure 11.3.a:

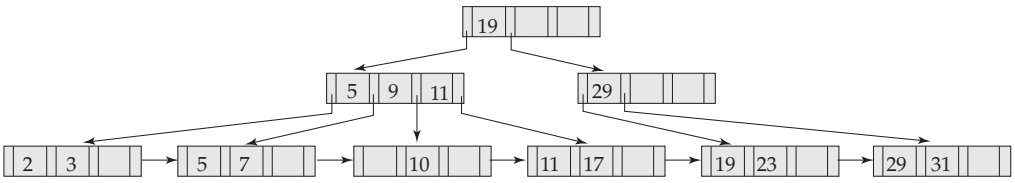
  Insert 9:



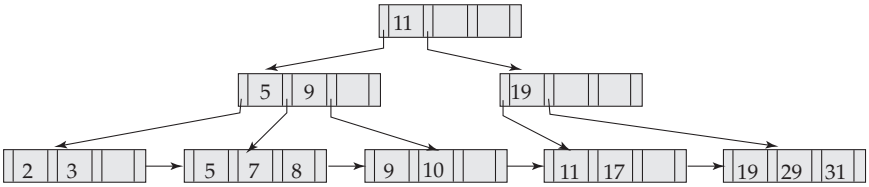
  Insert 10:



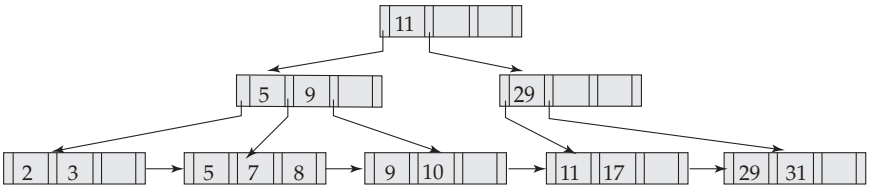
  Insert 8:



Delete 23:

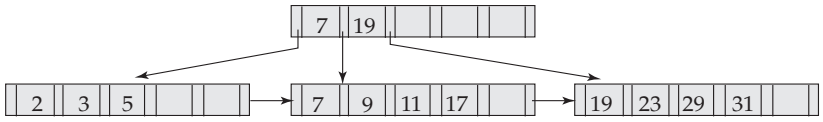


Delete 19:

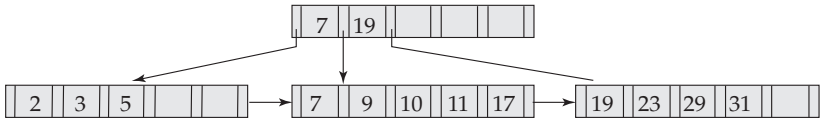


- With structure 11.3.b:

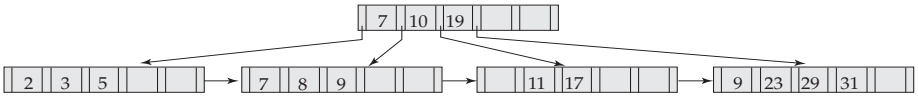
  Insert 9:



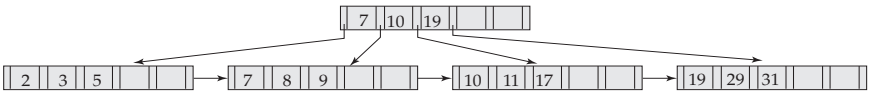
  Insert 10:



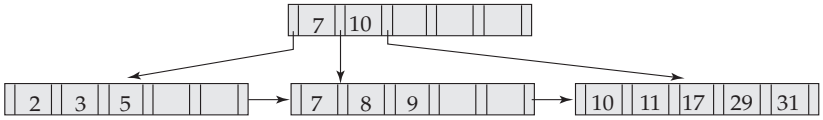
  Insert 8:



Delete 23:

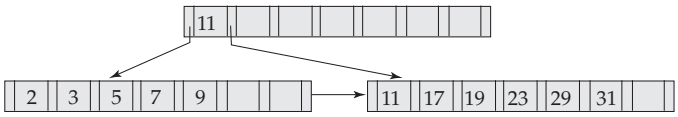


Delete 19:

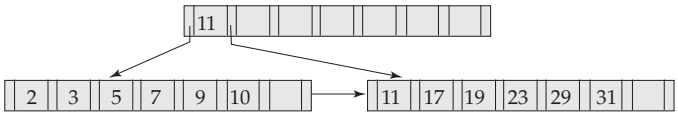


- With structure 11.3.c:

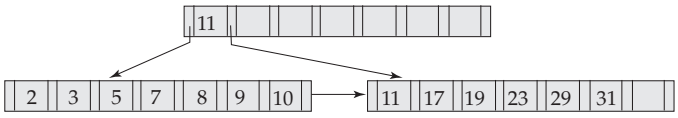
  Insert 9:



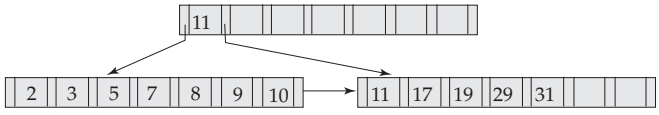
  Insert 10:



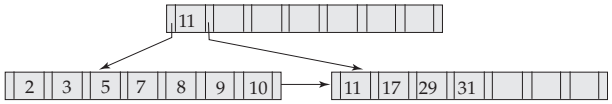
  Insert 8:



  Delete 23:

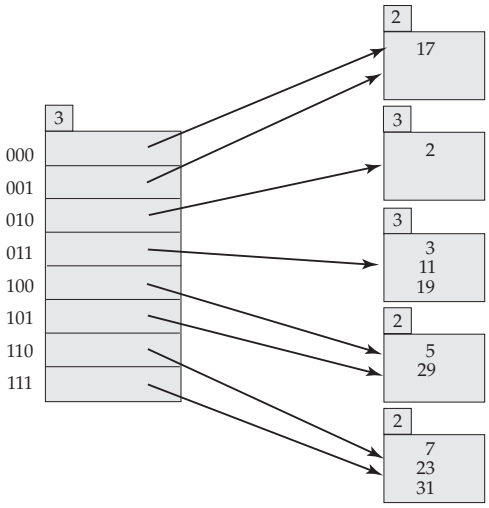


  Delete 19:



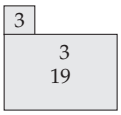
11.5 **Answer:** If there are K search-key values and $m - 1$ siblings are involved in the redistribution, the expected height of the tree is: $log_{\lfloor (m-1)n/m \rfloor}(K)$

11.6 **Answer:** Extendable hash structure



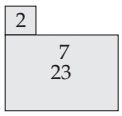
11.7 **Answer:**

a. Delete 11: From the answer to Exercise 11.6, change the third bucket to:

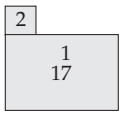


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

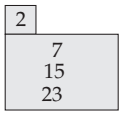
b. Delete 31: From the answer to 11.6, change the last bucket to:



c. Insert 1: From the answer to 11.6, change the first bucket to:



d. Insert 15: From the answer to 11.6, change the last bucket to:



11.8 **Answer:** The pseudocode is shown in Figure 11.1.

11.9 **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. The pseudocode is shown in Figure 11.2.
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.

11.10 **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

```
function findIterator(value V) {
/* Returns an iterator for the search on the value V */
    Iterator iter();
    Set iter.value = V;
    Set C = root node
    while (C is not a leaf node) begin
        Let i = samllest number such that V <= C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m;
        end
        else Set C = C.P_i;
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i then begin
        Set iter.index = i;
        Set iter.page = C;
        Set iter.active = TRUE;
    end
    else if (V is the greater than the largest value in the leaf) then begin
        if (C.P_n.K_1 = V) then begin
            Set iter.page = C.P_n;
            Set iter.index = 1;
            Set iter.active = TRUE;
        end
        else Set iter.active = FALSE;
    end
    else Set iter.active = FALSE;
    return (iter)
}
Class Iterator {
    variables:
        value V /* The value on which the index is searched */
        boolean active /* Stores the current state of the iterator (TRUE or FALSE)*/
        int index /* Index of the next matching entry (if active is TRUE) */
        PageID page /* Page Number of the next matching entry (if active is TRUE)*/

    function next() {
        if (active) then begin
            Set retPage = page;
            Set retIndex = index;
            if (index + 1 = page.size) then begin
                page = page.P_n
                index = 0
            end
            else index = index + 1;
            if (page.K_index ≠ V)
                then active=FALSE;
            return(retPage, retIndex)
        end
        else return null;
    }
}
```

Figure 11.1 Pseudocode for findIterator and the Iterator class

```
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 ≠ 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
```

Figure 11.2 Pseudocode for deletion

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.

11.11 **Answer:** We reproduce the instructor relation below.

ID	*name*	*dept_name*	*salary*
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
$S_4 \cap$ 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.

11.12 **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 takes place on the right node. This makes the occupancy of the leaf nodes to about 50 percent, except 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.

11.13 **Answer:**

a. 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^+-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 build 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^+-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 then are at least half filled.

11.14 **Answer:** In a B^+-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 leafs 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.

a. One way to solve this problem is to rebuild the index to restore sequentiality.

b. i. In the worst case each n-block unit and each node of the B^+-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 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^+-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 comes down by a factor of $n/2$.

iv. Allowing redistribution among the nodes of the same block, does not require additional seeks, where as, in regular B^+-tree 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)