COMP2411

Tutorial 5 (Week 12)

Q1.

1. Compare and contrast the three types of single-level indices, i.e., primary index, clustering index, and secondary index, in terms of their applicable conditions, and the kinds of search/queries as supportable.

Types of Single-Level Indexes

Primary Index

- Defined on an ordered data file
- The data file is ordered on a *key field*
- Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
- A similar scheme can use the *last record* in a block

Clustering Index

- Defined on an ordered data file
- The data file is ordered on a *non-key field*
- Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value

Page 31 or 32, Lecture Notes 7.

Q1.

Types of Single-Level Indexes

- **Secondary Index**
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists
  - The secondary index may be on a candidate key field (a unique value in every record), or a non-key field (with duplicate values)
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some **non-ordering field** (ie.,the indexing field) of the data file.
    - The second field is either a **block** pointer or a **record** pointer
    - If we include one entry *for each record* in the data file, then it is called a *dense index*

There can be *many* secondary indexes for the same data file.

Page 36 or 37, Lecture Notes 7.

Q2.

2. Construct a B+ tree for the following set of key values:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assuming that the tree is initially empty, values are added in ascending order, and the order of the tree is 4 and the leaf order is 3.

For the tree derived from the above question, show the form of the tree after each of the following series of operations:

a) Insert 9
b) Insert 10
c) Insert 8
d) Delete 7
e) Delete 3
f) Delete 5

Q2

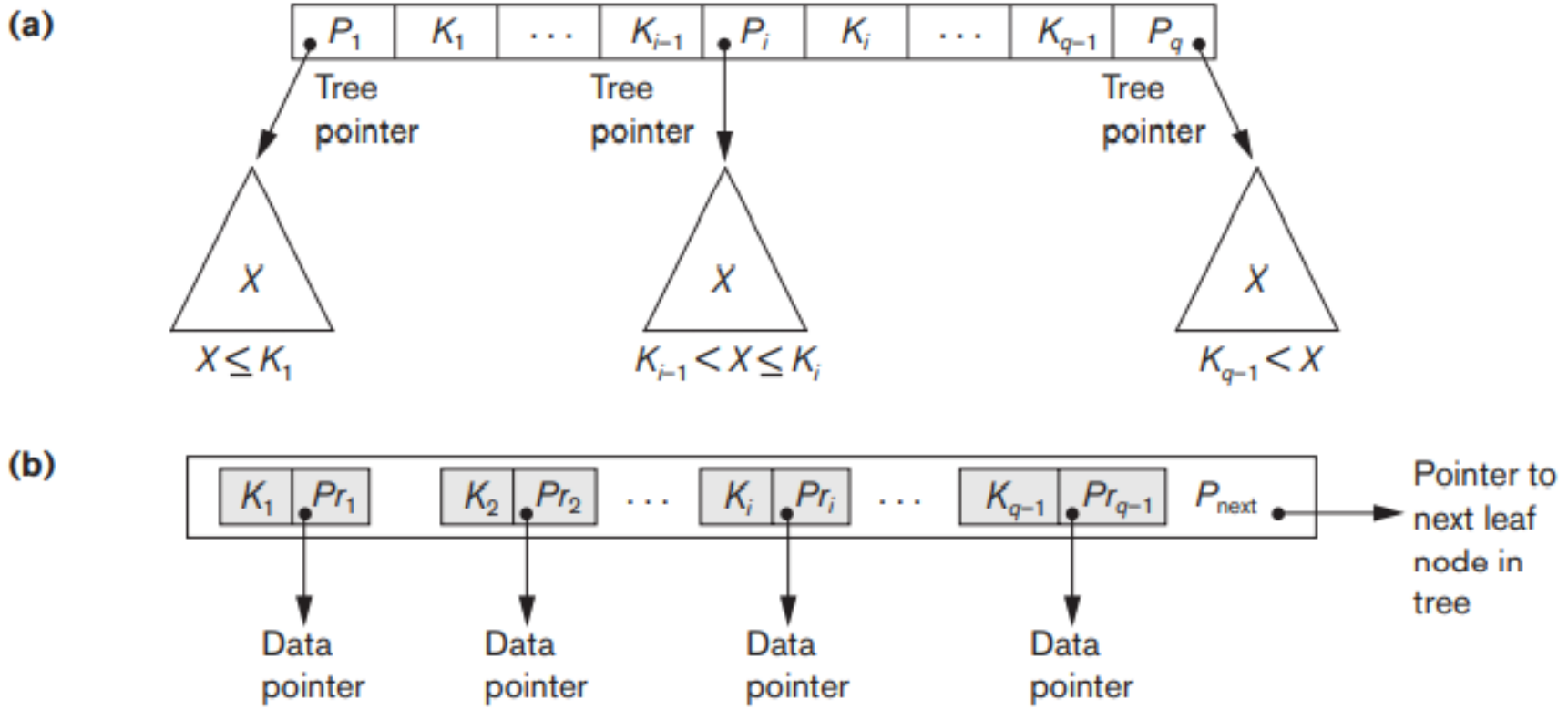

Figure 17.11
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values. (b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.

Page 623, Fundamental Database Systems.

Q2.

17.3.2 B$^+$-Trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B$^+$-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B$^+$-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the B$^+$-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B$^+$-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B$^+$-tree to guide the search. The structure of the *internal nodes* of a B$^+$-tree of order p (Figure 17.11(a)) is as follows:

1. Each internal node is of the form

   $$<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$$

   where $q < p$ and each P_i is a **tree pointer**.
2. Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.
3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 17.11(a)).[12]
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the *leaf nodes* of a B$^+$-tree of order p (Figure 17.11(b)) is as follows:

1. Each leaf node is of the form

   $$<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$$

   where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B$^+$-tree.
2. Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}, q \leq p$.
3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

[12] Our definition follows Knuth (1998). One can define a B$^+$-tree differently by exchanging the $<$ and $\leq$ symbols ($K_{i-1} \leq X < K_i$; $K_{q-1} \leq X$), but the principles remain the same.

Page 622&623, Fundamental Database Systems.

Algorithm 17.3. Inserting a Record with Search Key Field Value K in a B^+-Tree of Order p

```
n ← block containing root node of B+-tree;
read block n; set stack S to empty;
while (n is not a leaf node of the B+-tree) do
    begin
    push address of n on stack S;
        (*stack S holds parent nodes that are needed in case of split*)
    q ← number of tree pointers in node n;
    if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
        then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
        else if K ← n.Kq–1
            then n ← n.Pq
            else begin
                search node n for an entry i such that n.Ki–1 < K ≤ n.Ki;
                n ← n.Pi
                end;
        read block n
    end;
search block n for entry (Ki, Pri) with K = Ki; (*search leaf node n*)
if found
    then record already in file; cannot insert
    else (*insert entry in B+-tree to point to record*)
        begin
        create entry (K, Pr) where Pr points to the new record;
        if leaf node n is not full
            then insert entry (K, Pr) in correct position in leaf node n
            else begin (*leaf node n is full with pleaf record pointers; is split*)
                copy n to temp (*temp is an oversize leaf node to hold extra entries*);
                insert entry (K, Pr) in temp in correct position;
                (*temp now holds pleaf + 1 entries of the form (Ki, Pri)*)
                new ← a new empty leaf node for the tree; new.Pnext ← n.Pnext ;
                j ← ⌈(pleaf + 1)/2⌉;
                n ← first j entries in temp (up to entry (Kj, Prj)); n.Pnext ← new;
                new ← remaining entries in temp; K ← Kj ;
                (*now we must move (K, new) and insert in parent internal node;
                  however, if parent is full, split may propagate*)
                finished ← false;
                repeat
                if stack S is empty
                    then (←no parent node; new root node is created for the tree*)
                        begin
                        root ← a new empty internal node for the tree;
                        root ← <n, K, new>; finished ← true;
                        end
                    else begin
                        n ← pop stack S;
                        if internal node n is not full
                            then
                                begin (*parent node not full; no split*)
                                insert (K, new) in correct position in internal node n;
                                finished ← true
                                end
                            else begin (*internal node n is full with p tree pointers;
                                        overflow condition; node is split*)
                                copy n to temp (*temp is an oversize internal node*);
                                insert (K, new) in temp in correct position;
                                (*temp now has p + 1 tree pointers*)
                                new ← a new empty internal node for the tree;
                                j ← ⌊((p + 1)/2⌋;
                                n ← entries up to tree pointer Pj in temp;
                                (*n contains <P1, K1, P2, K2, ... , Pj–1, Kj–1, Pj>*)
                                new ← entries from tree pointer Pj+1 in temp;
                                (*new contains < Pj+1, Kj+1, ... , Kp–1, Pp, Kp, Pp+1 >*)
                                K ← Kj
                                (*now we must move (K, new) and insert in
                                  parentinternal node*)
                            end
                    end
                until finished
                end;
end;
```

Page 626&627, Fundamental Database Systems.

Q2.

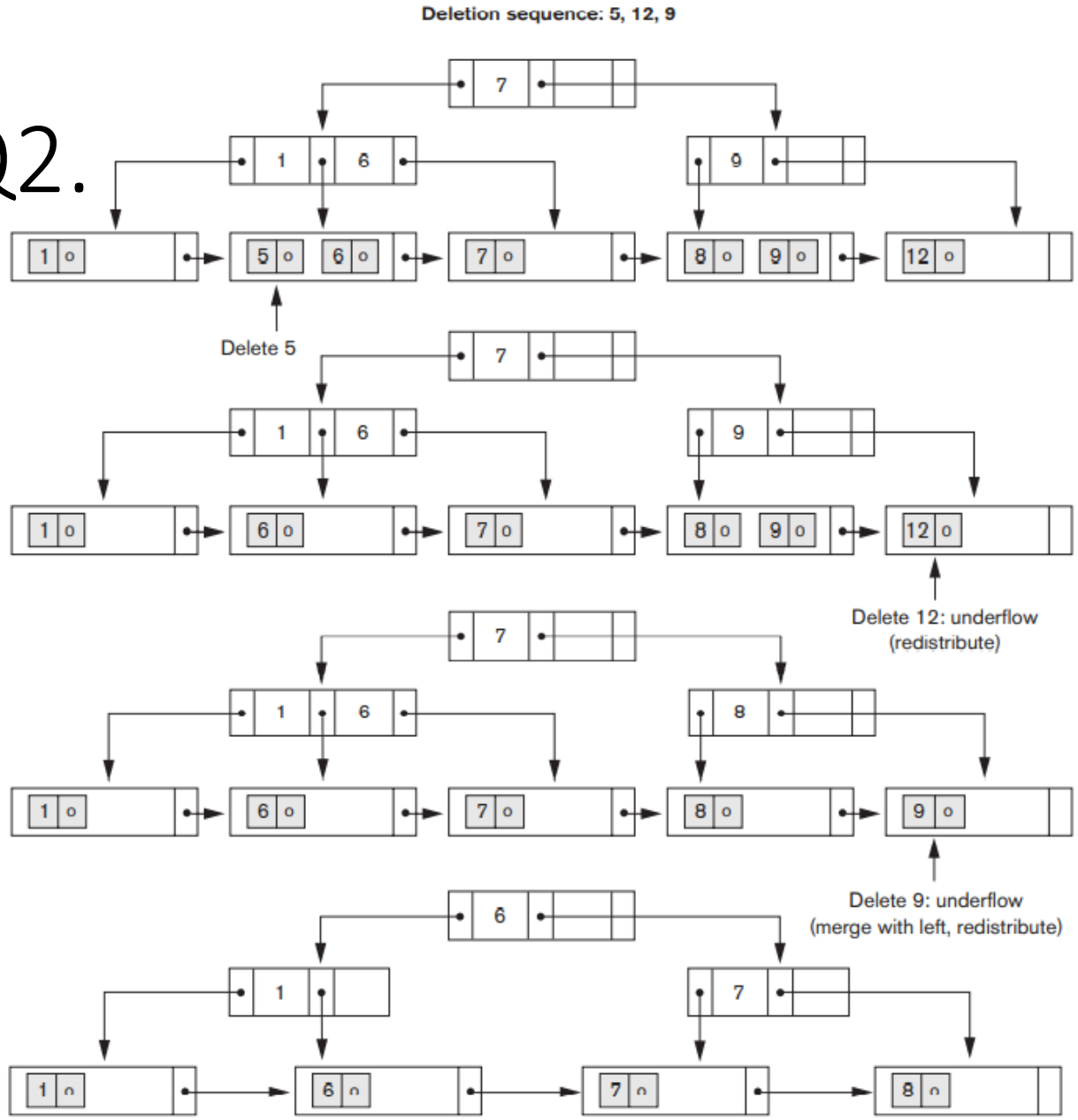

Figure 17.13
An example of deletion from a B^+-tree.

Figure 17.13 illustrates deletion from a B^+-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and its **sibling** so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is also not possible, the three nodes are merged into two leaf nodes. In such a case, underflow may propagate to **internal** nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 17.3. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.[13]

Page 629&630, Fundamental Database Systems.

Q2.

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L'
            Copy L.P_1 ... L.K_{n−1} to a block of memory T that can
                hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L'.P_n = L.P_n; Set L.P_n = L'
            Erase L.P_1 through L.K_{n−1} from L
            Copy T.P_1 through T.K_{⌈n/2⌉} from T into L starting at L.P_1
            Copy T.P_{⌈n/2⌉+1} through T.K_n from T into L' starting at L'.P_1
            Let K' be the smallest key-value in L'
            insert_in_parent(L, K', L')
        end
```

```
procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K_1)
        then insert P, K into L just before L.P_1
        else begin
            Let K_i be the highest value in L that is less than or equal to K
            Insert P, K into L just after L.K_i
        end
procedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'    /* N and N' are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P_1 ... T.P_{⌈(n+1)/2⌉} into P
            Let K'' = T.K_{⌈(n+1)/2⌉}
            Copy T.P_{⌈(n+1)/2⌉+1} ... T.P_{n+1} into P'
            insert_in_parent(P, K'', P')
        end
```

Page 643&644, Database System Concepts.

Q2.


```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
                        pair in N'
                    remove (N'.P_m, N'.K_m) from N'
                    insert (N'.P_m, N'.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_m
                end
            end
            else ... symmetric to the then case ...
        end
    end
```

Figure 14.21 Deletion of entry from a B⁺-tree.

Page 648, Database System Concepts.