Name

CWID

Quiz

1

Feb 13th, 2017
Due Feb 27th, 11:59pm

Quiz 1: CS525 - Advanced Database Organization

Results

Please leave this empty! 1.1 1.2 1.3 1.4 Sum

Instructions

- **You have to hand in the assignment using your blackboard**
- **This is an individual and not a group assignment**
- Multiple choice questions are graded in the following way: You get points for correct answers and points subtracted for wrong answers. The minimum points for each questions is **0**. For example, assume there is a multiple choice question with 6 answers - each may be correct or incorrect - and each answer gives 1 point. If you answer 3 questions correct and 3 incorrect you get 0 points. If you answer 4 questions correct and 2 incorrect you get 2 points. ...
- For your convenience the number of points for each part and questions are shown in parenthesis.
- There are 4 parts in this quiz
  1. SQL
  2. Relational Algebra
  3. Index Structures
  4. Result Size Estimation

Part 1.1 SQL (Total: 31 + 10 bonus points Points)

Consider the following social network database schema and example instance. The example data should not be used to formulate queries. SQL statements that you write should return the correct result for every possible instance of the schema!

user

name	userID	age	state	postVisibility
Peter Petersen	ppman	17	IL	friend
Gert Gertsen	supergert	19	CA	public
Alice Altmann	aa21	21	IL	friendOfFriend
Oma Omasen	grannyO	67	WI	public

friends

userID	friendID	since
ppman	supergert	2014
supergert	ppman	2014
grannyO	aa21	2011
aa21	supergert	2015

posts

postID	userID	message	date	refPostID
1	ppman	Weather is great lately!	2017-05-23	NULL
2	supergert	...boooring	2017-05-23	1
3	aa21	not over here!	2017-05-24	1
4	grannyO	baked some nice cake for @aa21	2017-07-01	NULL

likes

userID	postID	date
supergert	2	2017-05-23
aa21	4	2017-07-07

Hints:

- Attributes with black background are the primary key attributes of a relation
- The attributes `userID` and `friendID` or relation `friends` are both foreign keys to relation `user`.
- The attribute `userID` of relation `posts` is a foreign key to relation `user`.
- The attribute `userID` of relation `likes` is a foreign key to relation `user`.
- The attribute `refPostID` of relation `posts` is a foreign key to relation `posts`.
- The attribute `postID` of relation `likes` is a foreign key to relation `posts`.

Question 1.1.1 (2 Points)

Write an SQL query that returns for each post the total number of responses (other posts that have this post as a `refPostID`) as a rolling count. For this question you do not have to consider indirect responses (posts that respond to a responding post).

Solution

```
SELECT p1.postID, count(*) OVER (PARTITION BY p1.postID ORDER BY date ASC)
FROM posts p1 LEFT OUTER JOIN posts p2 ON (p1.postID = p2.refPostID)
```

Question 1.1.2 (2 Points)

Write a query that returns for each user (their *name*) the average number of likes each of the users post has received.

Solution

```
SELECT count(*) / count(DISTINCT p.postID) AS avglikes, name
FROM user u LEFT OUTER JOIN posts p USING (userID)
            LEFT OUTER JOIN likes l USING (postID)
GROUP BY u.name
```

Question 1.1.3 (6 Points)

For each user determine the number of users that can see their posts. The visibility of posts is determined by the value of attribute `postVisibility`: `public` means everybody can see the users posts, `friend` limits visibility to direct friends of a person, and `friendOfFriends` also allows friends of a friend of the poster to see their posts.

Solution

```
WITH directFriends AS (
    SELECT userID, friendID
    FROM friend f
  ),
  indirectFriends AS (
    SELECT d.userID, f.friendID
    FROM directFriends d, friends f
    WHERE d.friendID = f.userID
    UNION
    SELECT * FROM directFriends
  )
SELECT u.userID,
       CASE WHEN postVisibility = 'public'
            THEN (SELECT count(*) FROM user)
            WHEN postVisibility = 'friend'
            THEN (SELECT count(*) FROM directFriends f WHERE u.userID = f.userID)
            WHEN postVisibility = 'friendOfFriend'
            THEN (SELECT count(*) FROM indirectFriends f WHERE u.userID = f.userID)
       END AS numVis
FROM user u
```

Question 1.1.4 (5 Points)

Write an SQL query that returns posts to which all users in the social network have responded to.

Solution

```sql
WITH responders AS (
    SELECT userID, refPostID AS postID
    FROM posts
    WHERE refPostID IS NOT NULL
  ),
  potentialResponders AS (
    SELECT u.userID, postID
    FROM user u, posts p
  )
  missedResponders AS (
    SELECT postID
    FROM
      (potentialReponders
      EXCEPT
      responders)
  )
SELECT p1.postID
FROM posts p1
WHERE p1.postID NOT IN (SELECT postID
                        FROM missedResponders);
```

of course a double `NOT EXISTS` would work too!

Question 1.1.5 (4 Points)

Write an SQL Query that returns pairs of users that are friends (direct), but have not responded to each others posts yet.

Solution

```
WITH responded AS (
    SELECT p1.userID AS posterID, p2.userID AS responseID
    FROM posts p1, posts p2
    WHERE p1.userID = p2.refPostID
  ),
  symResp AS (
    SELECT * FROM responded
    UNION
    SELECT respondID AS posterID, posterID AS responseID FROM respond
  )
SELECT userID, friendID
FROM friends f
WHERE (userID, friendID) NOT IN (SELECT * FROM symResp);
```

of course one can also use not exists

Question 1.1.6 (3 Points)

Write an SQL query that returns the number of posts that either contain the word "cake" or are direct or indirect responses to posts with the word "cake" in it.

Solution

```
WITH cakePosts(postID) AS (
    SELECT postID FROM posts WHERE message LIKE '%cake%'
    UNION ALL
    SELECT p.postID FROM cakePosts c, posts p WHERE c.postID = p.refPostID
  )
SELECT count(*) FROM cakePosts;
```

Question 1.1.7 (2 Points)

Write an SQL query that returns users that have not posted any message yet.

Solution

```
SELECT userID
FROM user u
WHERE u.userID NOT IN (SELECT userID FROM posts p);
```

Question 1.1.8 (4 Points)

Return the 3 messages (attribute `message` of `posts`) with the most likes.

Solution

```
SELECT *
FROM
  (SELECT message
  FROM (SELECT count(l.postID) AS likes, p.message
        FROM posts p LEFT OUTER JOIN likes l USING (postID)
        GROUP BY p.message) AS pl
  ORDER BY likes DESC)
WHERE ROWNUM < 4;
```

Question 1.1.9 (4 Points)

Write a query that returns users ranked by an activity score. The activity score of a user is the average of their activity score per year. An activity score per year is computed based on the number of new friends the user has made, the number of new posts the user has posted, and the number of new likes from that user. These three measures are then combined into weighted sum (weights are 30% for friends, 50% for posts, and 20% for likes).

Solution

```
WITH newFriends AS (
    SELECT userID, count(*) AS fC, since AS y
    FROM friends
    GROUP BY userID, since),
  newPosts AS (
    SELECT userID, count(*) AS pC, year(date) AS y
    FROM posts
    GROUP BY userID, year(date)),
  newLikes AS (
    SELECT userID, count(*) AS lC, year(date) AS y
    FROM likes
    GROUP BY userID, year(date)),
  yearScore AS (
    SELECT 0.3 * fC + 0.5 * pC + 0.2 * lC AS yearScore, y, userID
    FROM newFriends NATURAL JOIN newPosts NATURAL JOIN newLikes),
  avgScore AS (
    SELECT userID, AVG(yearScore) AS activity
    FROM yearScore
    GROUP BY userID)

SELECT *
FROM avgScore
ORDER BY activity DESC;
```

Question 1.1.10 Optional Bonus Question (10 Bonus Points)

Write a calculator (interpreter of reverse polish notation expressions) as a recursive SQL query. The expression to be evaluated by your calculator are stored in a relation `stack(pos,type,elem)` where `pos` stores the order of stack entries, `type` is the type of entry (operation or operand), and `elem` stores the value of an operand (if type is `op`) or value of an operand (if type is `val`). You have to support the following arithmetic operations: addition and multiplication. Your calculator query should return the final result of evaluating the expression stored in relation `stack`. Recall that reverse polish notation is a way to represent arithmetic expressions without the need for parentheses. This is achieved by changing the order in which elements of an arithmetic operation are written down, i.e., instead of `operand operator operand` (e.g., $4 + 3$) the format is `operator operand operand` (e.g., $+4\,3$). An example instance of the `stack` relation for arithmetic expression `* 3 + 4 5` (that is $3 * (4 + 5)$) are shown below.

stack

pos	type	op
0	op	*
1	val	3
2	op	+
3	val	4
4	val	5

result

result
27

Solution

Code written for Oracle

```
CREATE TABLE stac (pos int, typ char(1), op VARCHAR(20));

INSERT INTO stac VALUES (0,'o','*');
INSERT INTO stac VALUES (1,'v','3');
INSERT INTO stac VALUES (2,'o','+');
INSERT INTO stac VALUES (3,'v','4');
INSERT INTO stac VALUES (4,'v','5');
COMMIT;

WITH res(it,pos,typ,op,val,oPos) AS (
  SELECT 1 AS it,
         ROW_NUMBER() OVER (ORDER BY pos DESC) - 1 AS pos,
         typ,
         op,
         CASE WHEN typ = 'v' THEN CAST(op AS NUMBER) ELSE -1 END AS val,
         0 as oPos
  FROM stac
  UNION ALL
  SELECT it + 1 AS it,
         CASE WHEN s.typ = 'o' THEN r.pos - 2 ELSE r.pos END AS pos,
         CASE WHEN r.pos = 2 AND s.typ = 'o' THEN  'v' ELSE r.typ END AS typ,
         r.op,
         CASE
          WHEN r.pos = 2 AND s.typ = 'o' AND s.op = '+' THEN
            SUM(val) OVER (PARTITION BY it
                           ORDER BY r.pos ASC
                           ROWS BETWEEN 2 PRECEDING AND 1 PRECEDING)
          WHEN r.pos = 2 AND s.typ = 'o' AND s.op = '*' THEN
            SUM(val) OVER (PARTITION BY it
                           ORDER BY r.pos ASC
                           ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
            * SUM(val) OVER (PARTITION BY it
                             ORDER BY r.pos ASC
                             ROWS BETWEEN 2 PRECEDING AND 2 PRECEDING)
          ELSE val
         END AS val,
         oPos + 1
  FROM res r, rstac s
  WHERE oPos = s.pos AND r.pos >= 0
)
SELECT val
FROM res
WHERE it = (SELECT max(it) FROM res)
      AND pos = 0;
```

Part 1.2 Relational Algebra (Total: 29 Points)

Question 1.2.1 Relational Algebra (2 Points)

Write a relational algebra expression over the schema from the SQL part (part 1) that returns the names of users from Illinois that are older than 21 years and have postVisibility set to public.

Solution

$$\pi_{name}(\sigma_{state=IL \wedge age>21 \wedge postVisibility=public}(\mathbf{user}))$$

Question 1.2.2 Relational Algebra (4 Points)

Write a relational algebra expression over the schema from the SQL part (part 1) that returns users (their userID) that have no friends from Illinois.

Solution

$$\mathbf{hasIL} \leftarrow projection_{userID}(\mathbf{user} \bowtie \mathbf{friends} \bowtie \pi_{userID \rightarrow friendID}(\sigma_{state=IL}(\mathbf{user})))$$
$$\mathbf{q} \leftarrow \pi_{userID}(\mathbf{user}) - \mathbf{hasIL}$$

Question 1.2.3 Relational Algebra (4 Points)

Write a relational algebra expression over the schema from the SQL part (part 1) that returns the state and name of users which have posted at least 3 messages.

Solution

$$\pi_{userID,state}(\sigma_{count(*)>2}(\ _{userID}\alpha_{count(*)}(\mathbf{posts})) \bowtie \mathbf{user})$$

Question 1.2.4 SQL → Relational Algebra (5 Points)

Translate the SQL query from Question 1.1.2 into relational algebra (bag semantics).

Solution

$$\begin{aligned}
\mathbf{joins} &\leftarrow \mathbf{user} \bowtie \mathbf{posts} \bowtie \pi_{postID}(\mathbf{likes}) \\
\mathbf{cnt} &\leftarrow\ _{name}\alpha_{count(1)}(\mathbf{joins}) \\
\mathbf{dcnt} &\leftarrow\ _{name}\alpha_{count(1)}(\delta(\pi_{postID,name}(\mathbf{joins}))) \\
\mathbf{counts} &\leftarrow cnt \bowtie dcnt \\
\mathbf{q} &\leftarrow \pi_{c/d \rightarrow avgLikes,name}(counts)
\end{aligned}$$

Question 1.2.5 SQL → Relational Algebra (5 Points)

Translate the SQL query from question 1.1.4 into relational algebra (bag semantics).

Solution

$$\begin{aligned}
\mathbf{res} &= \pi_{userID,refPostID \rightarrow postID}(\mathbf{posts}) \\
\mathbf{pot} &= \pi_{userID}(\mathbf{user}) \times \pi_{postID}(\mathbf{posts}) \\
\mathbf{miss} &= \pi_{postID}(\mathbf{pot} - \mathbf{res}) \\
\mathbf{q} &= \pi_{postID}(\mathbf{posts}) - \mathbf{miss}
\end{aligned}$$

Question 1.2.6 SQL → Relational Algebra (5 Points)

Translate the SQL query from question 1.1.5 into relational algebra (bag semantics).

Solution

$$\begin{aligned}
\textbf{res} &= \pi_{userID,responseID}(\pi_{userID,postID}(\textbf{posts}) \bowtie \pi_{userID \rightarrow responseID, refPostID \rightarrow postID}(posts)) \\
\textbf{sym} &= \textbf{res} \cup (\pi_{userID \rightarrow responseID, responseID \rightarrow userID}(\textbf{res}) \\
\textbf{q} &= \pi_{userID,friendID}(\textbf{friends}) - \textbf{sym}
\end{aligned}$$

Question 1.2.7 Equivalences (4 Points)

Consider the following relation schemas (primary key attributes are underlined):
$R(\underline{A}, B)$, $S(\underline{B}, C)$, $T(\underline{C}, D)$.

Check equivalences that are correct under **set semantics**. For example $R \bowtie R \equiv R$ should be checked, whereas $R \equiv S$ should not be checked.

- ■ $\alpha_{sum(A)}(R) \equiv \alpha_{sum(X)}({}_{B}\alpha_{sum(A) \rightarrow X}(R))$
- ❑ ${}_{B}\alpha_{sum(A)}(R \bowtie S) \equiv {}_{B}\alpha_{sum(A)}(R) \bowtie S$
- ■ $\sigma_{B<10}({}_{B}\alpha_{sum(A)}(R)) \equiv {}_{B}\alpha_{sum(A)}(\sigma_{B<10}(R))$
- ❑ ${}_{B,C}\alpha_{sum(A)}(R \bowtie S) \equiv {}_{B}\alpha_{sum(A)}(R) \bowtie {}_{C}\alpha_{sum(A)}(R)$
- ■ $R \triangleright S \equiv R - (\pi_{A,B}(R \bowtie S))$
- ■ $\sigma_{B=3 \vee B=5}(R) \equiv \sigma_{B=3}(R) \cup \sigma_{B=5}(R)$
- ❑ $R - (S - R) \equiv S - (R - S)$
- ■ $R - (R - S) \equiv S - (S - R)$

Part 1.3 Index Structures (Total: 30 Points)

Assume that you have the following table:

Item

SSN	name	age
1	Pete	13
2	Bob	23
44	John	49
43	Joe	45
45	Alice	77
42	Lily	3
88	Gertrud	29
89	Heinz	14

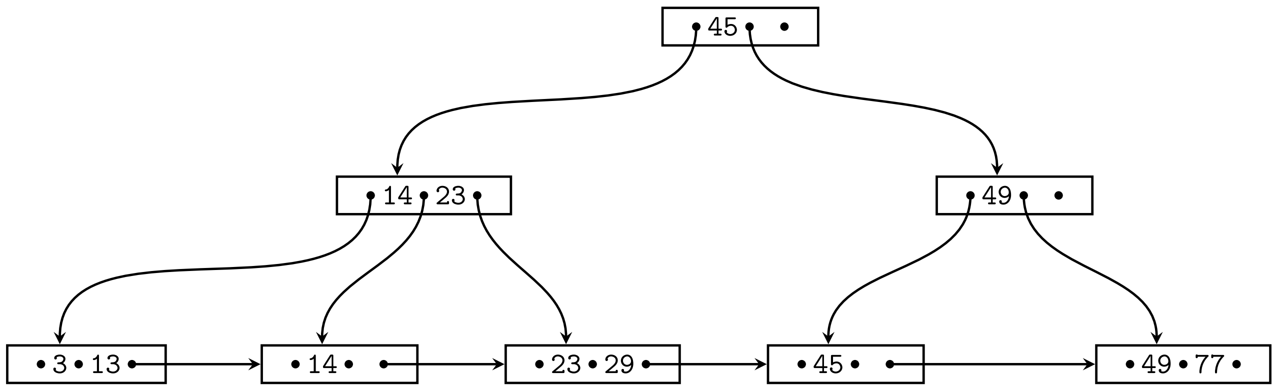
Question 1.3.1 Construction (12 Points)

Create a B+-tree for table *Item* on key *age* with $n = 2$ (up to two keys per node). You should start with an empty B+-tree and insert the keys in the order shown in the table above. Write down the resulting B+-tree after each step.

When splitting or merging nodes follow these conventions:

- **Leaf Split**: In case a leaf node needs to be split during insertion and n is even, the left node should get the extra key. E.g, if $n = 2$ and we insert a key 4 into a node [1,5], then the resulting nodes should be [1,4] and [5]. For odd values of n we can always evenly split the keys between the two nodes. In both cases the value inserted into the parent is the smallest value of the right node.
- **Non-Leaf Split**: In case a non-leaf node needs to be split and n is odd, we cannot split the node evenly (one of the new nodes will have one more key). In this case the "middle" value inserted into the parent should be taken from the right node. E.g., if $n = 3$ and we have to split a non-leaf node [1,3,4,5], the resulting nodes would be [1,3] and [5]. The value inserted into the parent would be 4.
- **Node Underflow**: In case of a node underflow you should first try to redistribute values from a sibling and only if this fails merge the node with one of its siblings. Both approaches should prefer the left sibling. E.g., if we can borrow values from both the left and right sibling, you should borrow from the left one.

Solution

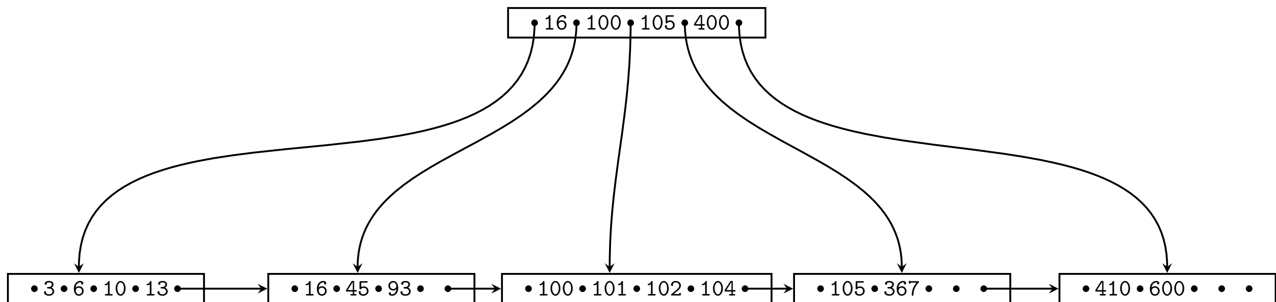

45
14 23
49
3 13
14
23 29
45
49 77

Question 1.3.2 Operations (10 Points)

Given is the B+-tree shown below ($n = 4$). Execute the following operations and write down the resulting B+-tree after each operation:

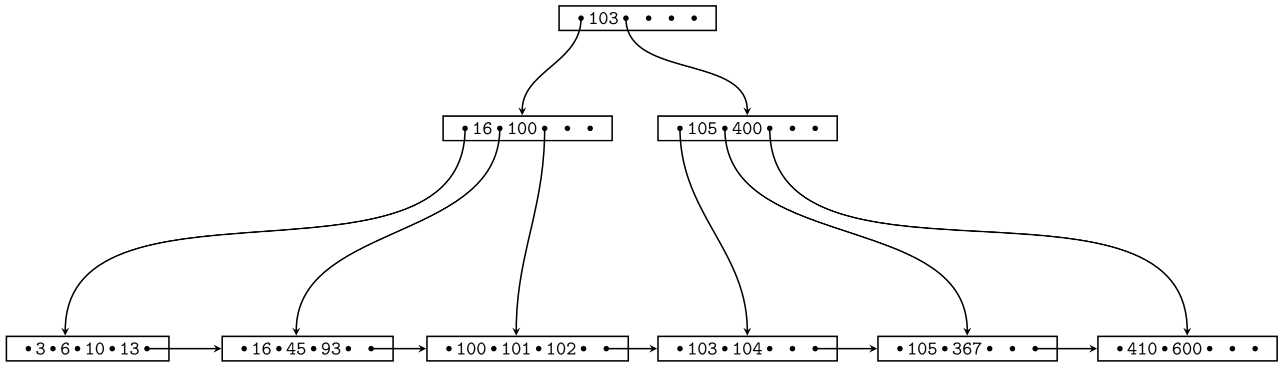
insert(103), insert(59), insert(60), delete(102), delete(103), delete(101)

Use the conventions for splitting and merging introduced in the previous question.

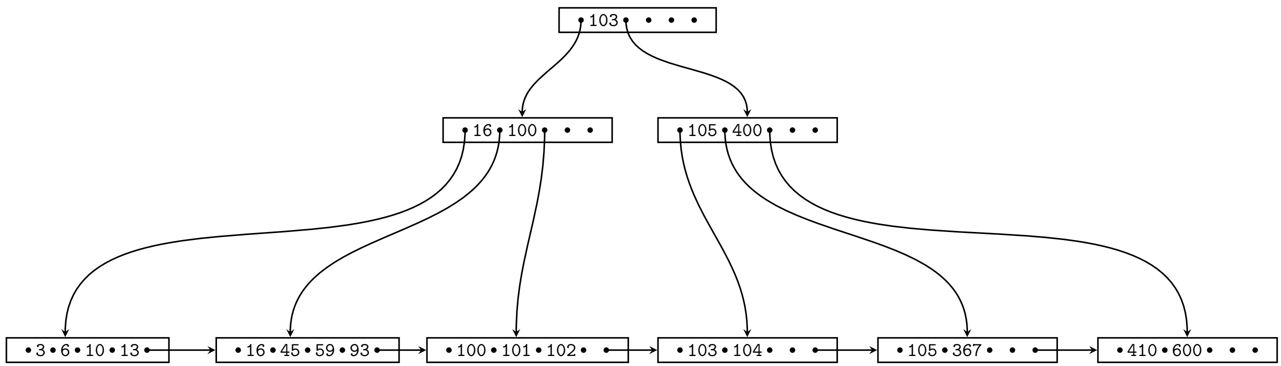


Solution

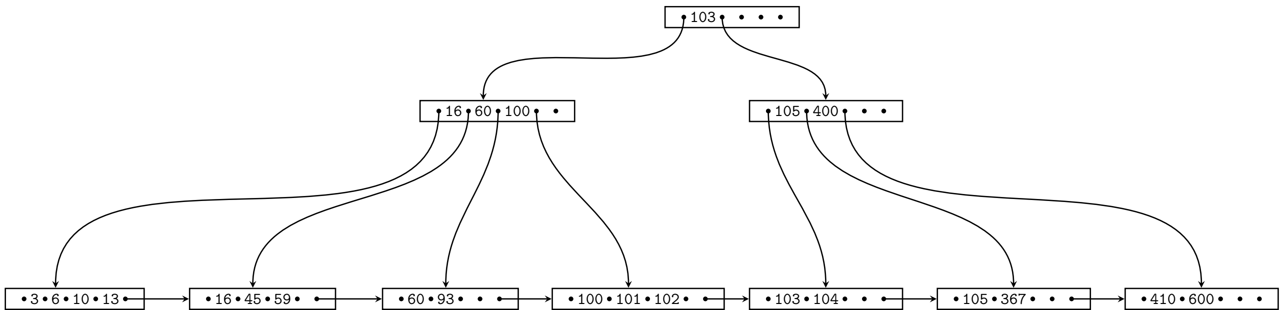
Insert 103



Insert 59

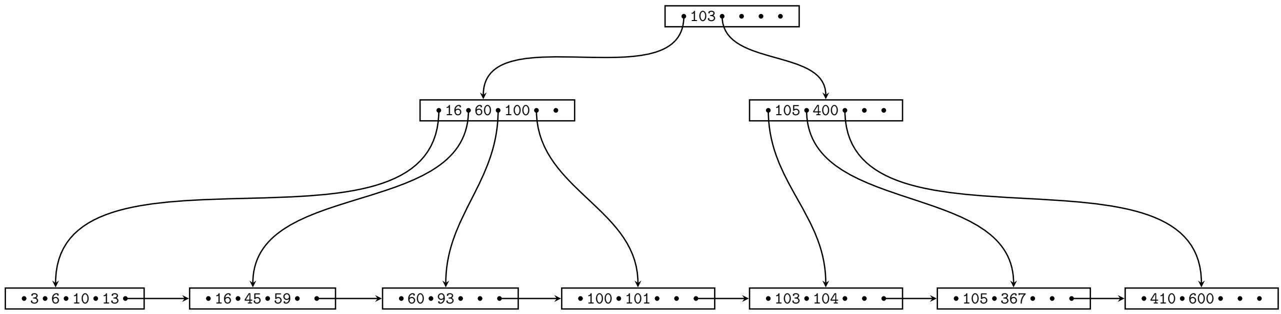


Insert 60

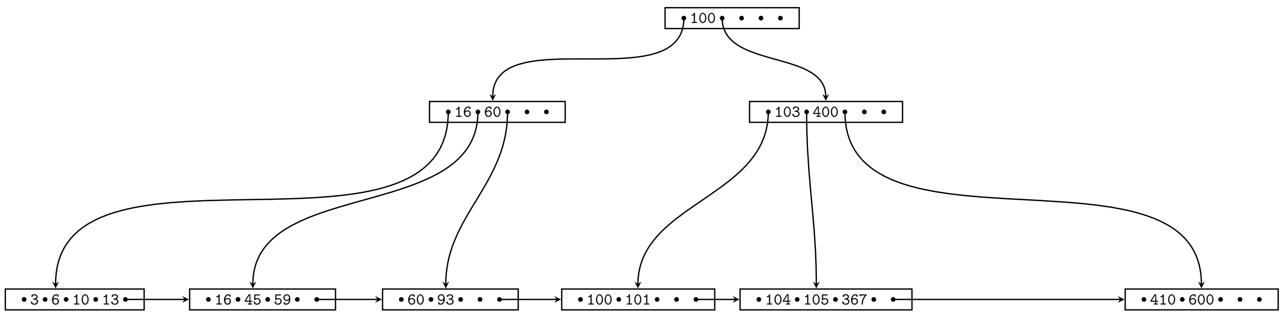


Solution

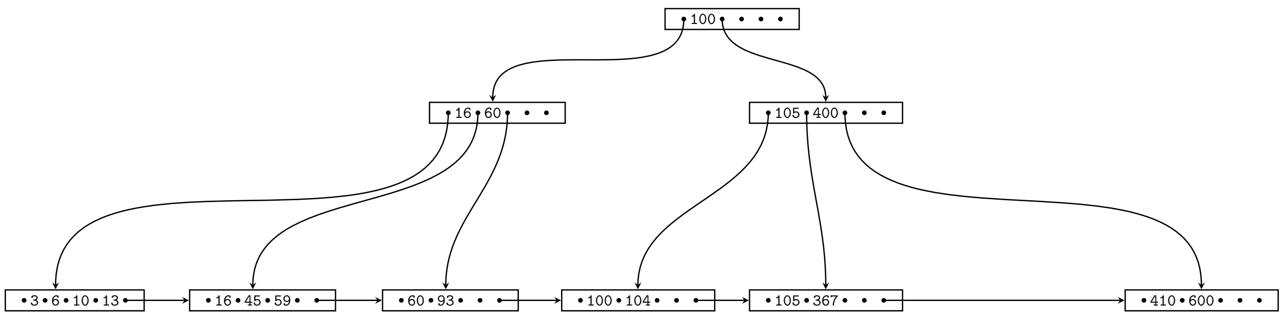
Delete 102



Delete 103



Delete 101



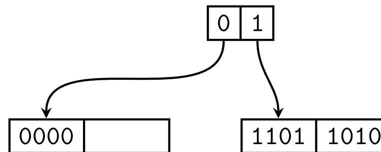
Question 1.3.3 Extensible Hashing (8 Points)

Consider the extensible Hash index shown below that is the result of inserting values 0, 1, and 2. Each page holds two keys. Execute the following operations

`insert(8),insert(5),insert(6),insert(3)`

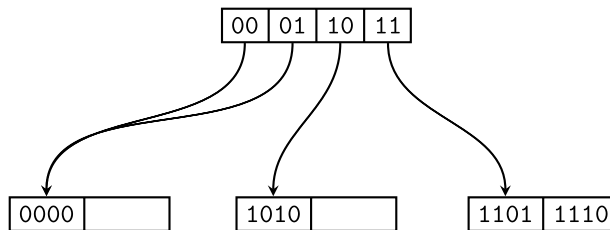
and write down the resulting index after each operation. Assume the hash function is defined as:

x	h(x)
0	1101
1	0000
2	1010
3	1100
4	0001
5	0000
6	1010
7	0111
8	1110

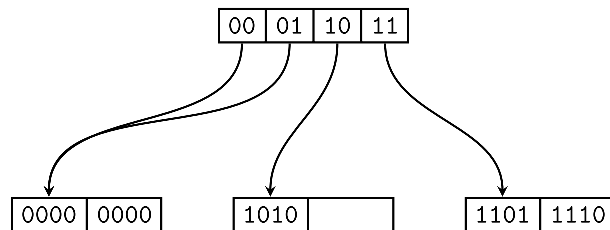


Solution

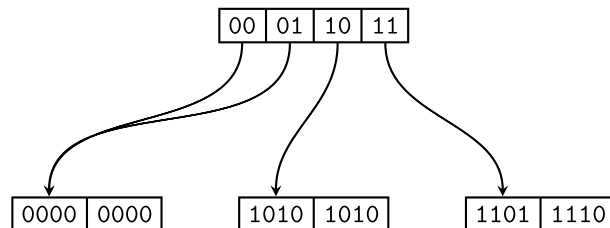
insert(8)



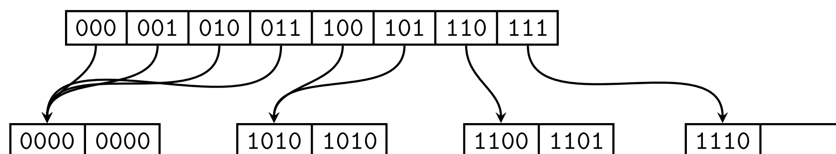
insert(5)



insert(6)



insert(3)



Part 1.4 Result Size Estimations (Total: 10 Points)

Consider a table `lecture` with attributes `title`, `campus`, `topic`, `roomSize`, a table `student` with `name`, `major`, `age`, and a table `attendsLecture` with attributes `student` and `lecture`. `attendsLecture.student` is a foreign key to `student`. Attribute `lecture` of relation `attendsService` is a foreign key to of relation `lecture`. Given are the following statistics:

$$T(lecture) = 200 \qquad T(student) = 30,000 \qquad T(attendsLecture) = 600,000$$
$$V(lecture, title) = 200 \qquad V(student, name) = 30,000 \qquad V(attendsLecture, student) = 25,000$$
$$V(lecture, campus) = 3 \qquad V(student, major) = 10 \qquad V(attendsLecture, lecture) = 150$$
$$V(lecture, topic) = 10 \qquad V(student, age) = 30$$
$$V(lecture, roomSize) = 20$$

Question 1.4.1 Estimate Result Size (3 Points)

Estimate the number of result tuples for the query $q = \sigma_{major=CS}(student)$ using the first assumption presented in class (values used in queries are uniformly distributed within the active domain).

Solution

$$T(q) = \frac{T(student)}{V(student, major)} = \frac{30,000}{10} = 3,000$$

Question 1.4.2 Estimate Result Size (3 Points)

Estimate the number of result tuples for the query $q = \sigma_{age>30 \wedge age \leq 50}(person)$ using the first assumption presented in class. The minimum and maximum values of attribute age are 20 and 59.

Solution

$$T(q) = \frac{50 - 30}{max(age) - min(age) + 1} \times T(student) = \frac{20}{40} \times 30,000 = 15,000$$

Question 1.4.3 Estimate Result Size (4 Points)

Estimate the number of result tuples for the query q below using the first assumption presented in class.

$$q = (student \bowtie_{name=student} attendsLecture \bowtie_{lecture=title} \sigma_{topic=CS}(lecture)$$

Solution

$$\begin{aligned}
q_1 &= \sigma_{topic=CS}(lecture) \\
T(q_1) &= = \frac{T(lecture)}{V(lecture, topic)} = \frac{200}{10} = 20 \\
V(q_1, title) &= V(lecture, title) = 200 \\
T(q) &= \frac{T(student) \times T(attendsLecture) \times T(q_1)}{max(V(student, name), V(attendsLecture, student)) \times max(V(attendsLecture, lecture), V(q_1, title))} \\
&= \frac{30,000 \times 600,000 \times 20}{max(30,000, 25,000) \times max(150, 20)} = \frac{600,000 \times 20}{150} = 80,000
\end{aligned}$$