|  |
| --- |
| **CIS 450/550 Database and Information Systems**  **Summer 2019** |

**Homework 4 (Due by 21:59:59 EDT on June 24)**

This homework is about transactions and indexing. You should read the assigned readings from the textbook (Chapter 16:1-6, 10: 3-8).

## Part 1: Transactions (50 points)

1. (15 points)

Consider a database with objects X and Y, and assume that there are two transactions T1 and T2. Transaction T1 reads X and Y and then writes X. Transaction T2 reads X and Y and then writes X and Y.

##### (5 points)

Give an interleaved schedule of T1 and T2 that has a write-read conflict, a read-write conflict and a write-write conflict. Show where these conflicts occur.

|  |  |
| --- | --- |
| T1 | T2 |
| R1(X) |  |
|  | R2(X) |
|  | R2(Y) |
|  | W2(X) |
|  | W2(Y) |
| R1(Y) |  |
| W1(X) |  |

This interleaved schedule has write-read conflict: W2(Y)R1(Y), read-write conflict: R1(X)W2(X) and R2(X)W1(X), write-write conflict: W2(X)W1(X).

1. (5 points)

Is your schedule serializable? Why or why not?

It is not serializable as there is a circle between T1 and T2 in precedence graph

And the conflicts here are T1->T2 (R1(X)W2(X)), T2-> T1 (R2(X)W1(X), W2(Y)R1(Y), W2(X)W1(X))

1. (5 points)

Is there any interleaved schedule of T1 and T2 that is achievable using strict two-phase locking (2PL) without lock upgrades?

T1: XLOCK1(X), R1(X), SLOCK1(Y), R1(Y), W1(X), REL(X,Y)

T2: XLOCK2(X), R2(X), XLOCK2(Y), R2(Y), W2(X), W2(Y), REL(X,Y)

If we start with T1, then we acquire a key XLOCK on object X and it will be released when transaction T1 commit. But if we want to start T2 we need to get XLOCK on X also, but it won’t be released before the end of T1, so we cannot start T2 before the end of T1. If we begin with T2 is the same. So the interleaved schedule is not achievable with strict two-phase.

2. (20 points)

Consider the following sequence of actions:

R1(X), W2(X) W2(Y), W3(Y), W1(Y), Commit(T1), Commit(T2) Commit (T3)

For each of the following concurrency control mechanisms, show how it handles the sequence by introducing lock acquisitions (SLOCKi(X), XLOCKi(X)) and releases (RELi(X))

Again, assume that the DBMS processes actions in the order given by your schedule. A lock is acquired just before it is used, and if a transaction is blocked all of its actions are queued until it is resumed; the DBMS continues with the next action in the schedule of an unblocked transaction.

Assume that the timestamp of Ti is i.

1. (5 points) Strict 2PL with timestamps used for deadlock prevention using a **wait-die** policy (see Section 17.4.1). Assume that the lower the timestamp the higher the transaction’s priority; that is, the oldest transaction has the highest priority these are kills

|  |  |  |
| --- | --- | --- |
| T1 |  |  |
| SLOCK1(X) |  |  |
| R1(X) |  |  |
|  | T2 |  |
|  | Require XLOCK2(X), but fail because SLOCK1(X) holds by T1 |  |
|  | T2>T1, DIE |  |
|  |  | T3 |
|  |  | XLOCK3(Y) |
|  |  | W3(Y) |
| Require XLOCK1(Y), but fail because XLOCK3(Y) holds by T3 |  |  |
| T1<T3, WAIT |  |  |
|  |  | COMMIT3, REL3(Y) |
| XLOCK1(Y) |  |  |
| W1(Y) |  |  |
| COMMIT1, REL1(X,Y) |  |  |

1. (5 points) Strict 2PL with timestamps used for deadlock prevention using a **wound-wait** policy (see Section 17.4.1). Assume that the lower the timestamp the higher the transaction’s priority; that is, the oldest transaction has the highest priority

|  |  |  |
| --- | --- | --- |
| T1 |  |  |
| SLOCK1(X) |  |  |
| R1(X) |  |  |
|  | T2 |  |
|  | Require XLOCK2(X), but fail because SLOCK1(X) holds by T1 |  |
|  | T2>T1, WAIT |  |
|  |  | T3 |
|  |  | XLOCK3(Y) |
|  |  | W3(Y) |
| Require XLOCK1(Y), but XLOCK3(Y) holds by T3 |  |  |
|  |  | T1<T3, DIE |
| XLOCK1(Y) |  |  |
| W1(Y) |  |  |
| COMMIT1,REL1(X,Y) |  |  |
|  | XLOCK2(X) |  |
|  | W2(X) |  |
|  | XLOCK2(Y) |  |
|  | W2(Y) |  |
|  | COMMIT2, REL2(X,Y) |  |

1. (5 points) Strict 2PL with deadlock **detection** (show the wait-for graph in case of deadlock, see Section 17.4). this is block not kill

SLOCK1(X), R1(X), {T2 is blocked by T1 as it requires XLOCK2(X), so T2->T1}, XLOCK3(Y), W3(Y), {T1 is blocked by T3 as it requires XLOCK1(Y), so T1->T3}, COMMIT3, REL3(Y), XLOCK1(Y), W1(Y), COMMIT1, REL1(X,Y), XLOCK2(X,Y), W2(X), W2(Y), COMMIT2, REL2(X,Y)

The wait for graph is T2->T1, T1->T3

1. (5 points) **Conservative** (and strict) 2PL, in which the transaction obtains al locks it will ever need when it begins, or blocks waiting for those locks to become available. Locks are held until the end of the transaction.

SLOCK1(X), XLOCK1(Y), R1, {T2 is blocked and waiting by T1 as it requires XLOCK2(X)}, {T3 is blocked and waiting by T1 as it requires XLOCK3(Y)}, W1(Y), COMMIT1, REL1(X,Y), XLOCK2(X,Y), W2(X), W2(Y), {T3 is blocked and waiting by T2 as it requires XLOCK3(Y)}, COMMIT2, REL2(X,Y), XLOCK3(Y), W3(Y), COMMIT3

3. (15 points)

Consider the following actions taken by transaction T1 on database objects X and Y :

R1(X); W1(X); R1(Y ); W1(Y )

Assume that the initial database state is X = 10, Y = 20 and that T1 decrements X by 5 and increments Y by 5. Run in isolation, T1 would yield a final state of X = 5, Y = 25. Further, assume that transaction T1 executes using isolation level REPEATABLE READ.

1. Give an example of another transaction T2 such that the result of running it concurrently with T1 under isolation level READ COMMITTED is different than what is achievable when running T2 under isolation level REPEATABLE READ. By “different”, we mean that the values read or written are different in the two executions. Specify the common schedule of T1 and T2 complete with lock and unlock operations, give concrete values for the data read and written, and explain why the results are different.

Schedule: R2(X), R1(X), W1(X), R1(Y), W1(Y), COMMIT1, R2(X), COMMIT2

|  |  |
| --- | --- |
| T1 REPEATABLE READ | T2 READ COMMITTED |
|  | SLOCK2(X) |
|  | R2(X) X=10 |
|  | REL2(X) |
| XLOCK1(X) |  |
| R1(X) X=10 |  |
| W1(X) X=5 |  |
| XLOCK1(Y) |  |
| R1(Y) Y=20 |  |
| W1(Y) Y=25 |  |
| COMMIT1 |  |
| REL1(X,Y) |  |
|  | SLOCK2(X) |
|  | R2(X) X=10 |
|  | REL2(X) |
|  | COMMIT2 |

|  |  |
| --- | --- |
| T1 REPEATABLE READ | T2 REPEATABLE READ |
|  | SLOCK2(X) |
|  | R2(X) X=10 |
| T1 requires XLOCK1(X), but fails because SLOCK2(X) is held by T2, T1 blocks |  |
|  | R2(X) X=10 |
|  | REL2(X) |
|  | COMMIT2 |
| XLOCK1(X) |  |
| R1(X) X=10 |  |
| W1(X) X=5 |  |
| XLOCK1(Y) |  |
| R1(Y) Y=20 |  |
| W1(Y) Y=25 |  |
| COMMIT1 |  |
| REL1(X,Y) |  |

If T2 is under READ COMMITTED, it will gets X=10 for the first time and X=5 for the second time. If T2 is under REPEATABLE READ, it will gets X=10 for the first time and X=10 for the second time. It gets two different values because 1. The transaction T1 is blocked in the second situation (it cannot get the lock it needs) 2. Transaction T2 is repeatable read under the second situation, which means it will always get X=10 once it reads.

1. Give an example of another transaction T3 (which could be the same as T2 above) such that the result of running it concurrently with T1 under isolation level READ UNCOMMITTED is different than what is achievable when running T3 under isolation level READ COMMITTED. Specify the common schedule of T1 and T3 complete with lock and unlock operations, give concrete values for the data read and written, and explain why the results are different.

Schedule: R1(X), R2(X), COMMIT2, W1(X), R1(Y), W1(Y), COMMIT1

|  |  |
| --- | --- |
| T1 REPEATABLE READ | T3 READ UNCOMMITTED |
| XLOCK1(X) |  |
| R1(X) X=10 |  |
|  | R2(X) X=10 |
|  | COMMIT2 |
| W1(X) X=5 |  |
| XLOCK1(Y) |  |
| R1(Y) Y=20 |  |
| W1(Y) Y=25 |  |
| COMMIT1 |  |
| REL1(X,Y) |  |
|  |  |

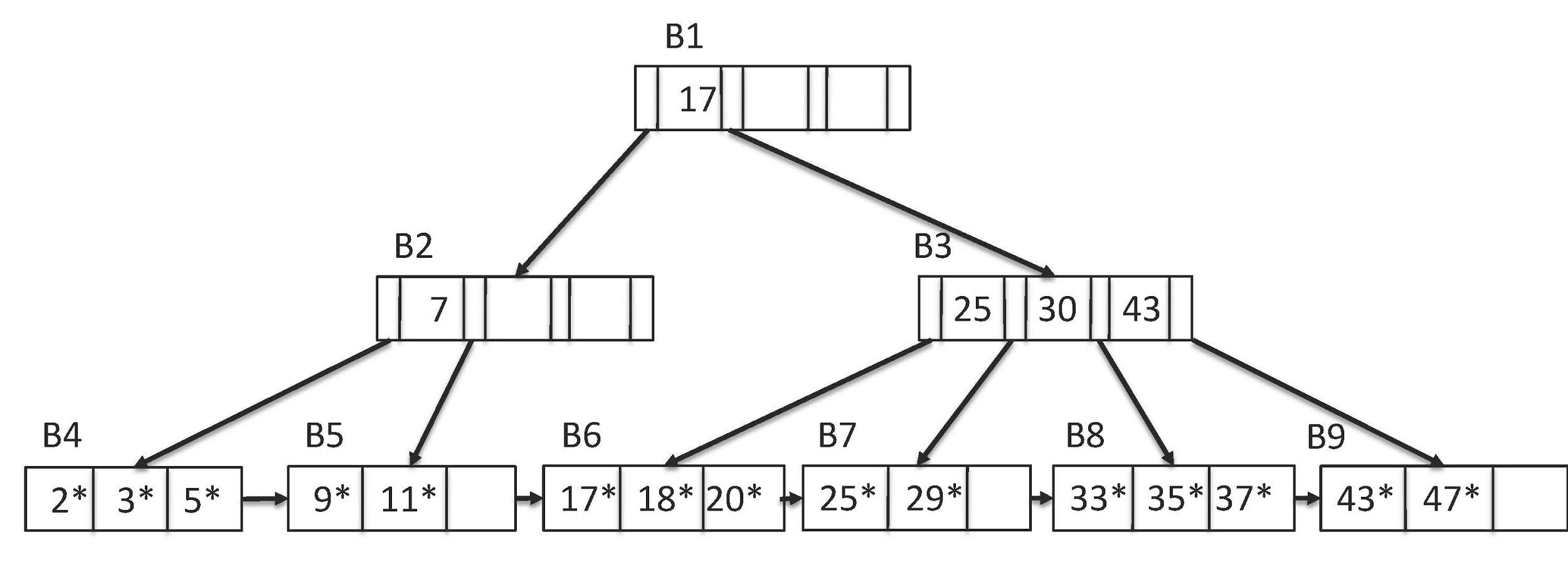
|  |  |
| --- | --- |
| T1 REPEATABLE READ | T3 READ COMMITTED |
| XLOCK1(X) |  |
| R1(X) X=10 |  |
|  | T2 requires SLOCK2(X), but fails because XLOCK1(X) is held by T1, T2 blocks |
| W1(X) X=5 |  |
| XLOCK1(Y) |  |
| R1(Y) Y=20 |  |
| W1(Y) Y=25 |  |
| COMMIT1 |  |
| REL1(X,Y) |  |
|  | SLOCK2(X) |
|  | R2(X) X=5 |
|  | REL2(X) |
|  | COMMIT2 |

If T3 is under READ UNCOMMITTED, it will gets X=10. If T3 is under READ COMMITTED, it will gets X=5. It gets two different values because 1. The transaction T2 under READ UNCOMMITTED does not require any locks for read, it can read dirty data in other transactions. 2. Transaction T2 under READ COMMITTED will be blocked when T1 is executing, and it can only reads X when T1 is finished, that is when X is modified to 5.

## Part 2: Storage and Indexing (50 points)

4. (25 points)

Consider the following B+ tree, where the index blocks are identified as B1, B2, B3, and the data entry blocks are B4,..., B9 (using Alternative 2). The minimum capacity of an index block is one key value (2 pointers), and the minimum capacity of a data entry block is 2 data entries.



Answer the following questions. For each of the following lookup operations, describe which blocks would be read and in which order. ***For each of the insert and delete operations, apply them to the given B+ tree*** and describe (or draw) the resulting tree. You can ignore inserting/deleting the record into/from the data file, and just focus on the data entries.

If you give a handwritten answer, please make sure that it is readable!

1. Lookup record with key 15.

B1->B2->B5. 15<17, so it goes to find in the left sub-tree of the root 17 in B1. Then, 15>7, so it goes to find in the right sub-tree of node 7 in B2. It goes to the leaf node B5, 9<11<15, and 15 does not exist in the leaf node B5. So, key 15 does not exist in this B+ tree.

1. Lookup record with key 43.

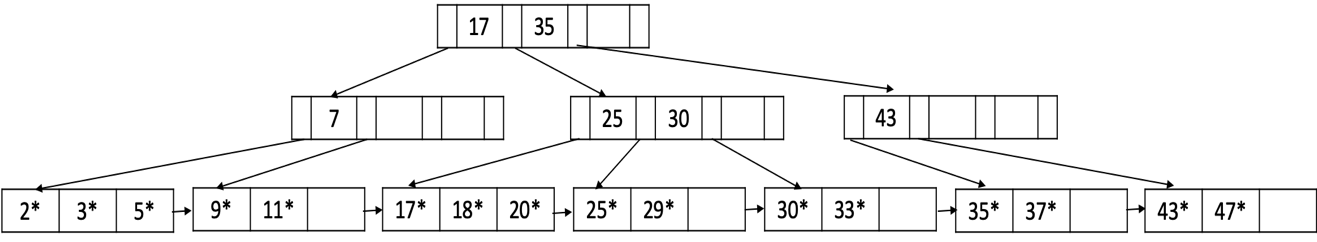
B1->B3->B9. 43>17, so it goes to find in the right sub-tree of the root 17 in B1. Then, 43 exist in the node B3, so it goes to find it in the right sub-tree of node 43. Key 43 exist in the leaf node B9 as the first key in this block. So, key 43 is in the first leaf node of B9 block in this B+ tree.

1. Lookup records with keys in the range 3 to 21.

B1->B2->B4->B5->B6->B7. 3<17, so it goes to find in the left sub-tree of the root 17 in B1. Then, 3<7, so it goes to find in the left sub-tree of node 7 in B2. It goes to the leaf node B4, 3 is in the second leaf node of block B4. So, a start points to the second node of B4, which is key=3. Then an end pointer goes to the next leaf node, 5<21, so it goes to the next leaf node (data entry). It iterates if the key of leaf node equal to or less than 21. When it goes to the first leaf node in block B7, 25>21, the iteration end and the end pointer points to the previous leaf node of 25, which is 20. So, the results are 3, 5, 9, 11, 17, 18, 20.

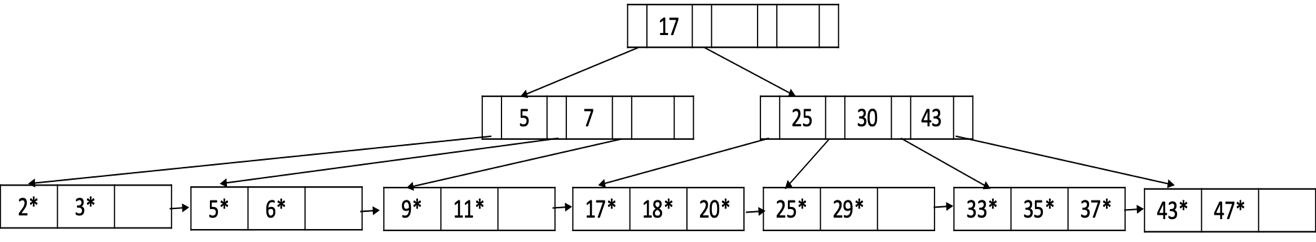
1. Insert record with key 30.

Go through the leaf nodes we find 29<30<33, so it should be insert between 29 and 33. The common node of the two sibling blocks (B7, B8) is 30 in B3. 30=30, so it should be insert into the right subtree of node 30, which is B8 and before 33. There is no empty leaf node, so, B8 needs to be split into 30, 33 and 35, 37 two subsets. The middle key 35 needs also to be pop up to the parent block, which is B3. And B3 is still full, so B3 also needs to be split into 25, 30 and 35, 43. Then pop up the middle one which is 35. There is empty space here, so we can put 35 here. The result is:

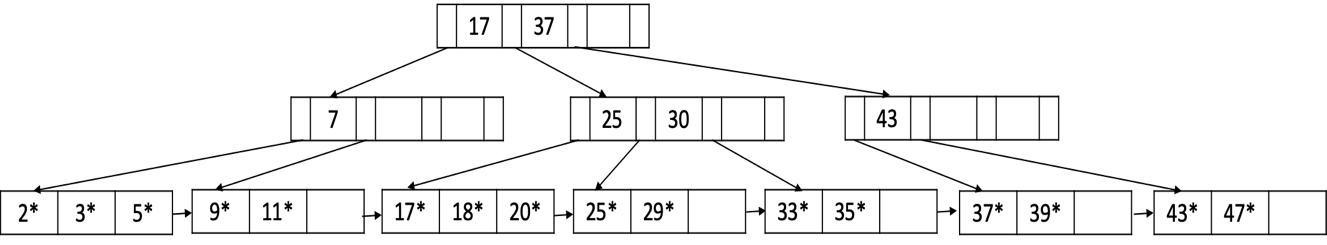


1. Insert record with key 6.

Go through the leaf nodes we find 5<6<9, so it should be insert between 5 and 9. The common node of the two sibling blocks (B4, B5) is 7 in B2. 6<7, so it should be insert into the left subtree of node 7, which is B4 and after 5. There is no empty leaf node, so, B4 needs to be split into 2, 3 and 5, 6 two subsets. The middle key 5 needs also to be pop up to the parent block, which is B2. And B2 has empty space, so it can be insert into B2. 5<7, so it needs to be insert into the first node of B2 and the second node is 7. The result is:



1. Insert record with key 39.



1. Delete record with key 33.

A close up of a logo

Description automatically generated

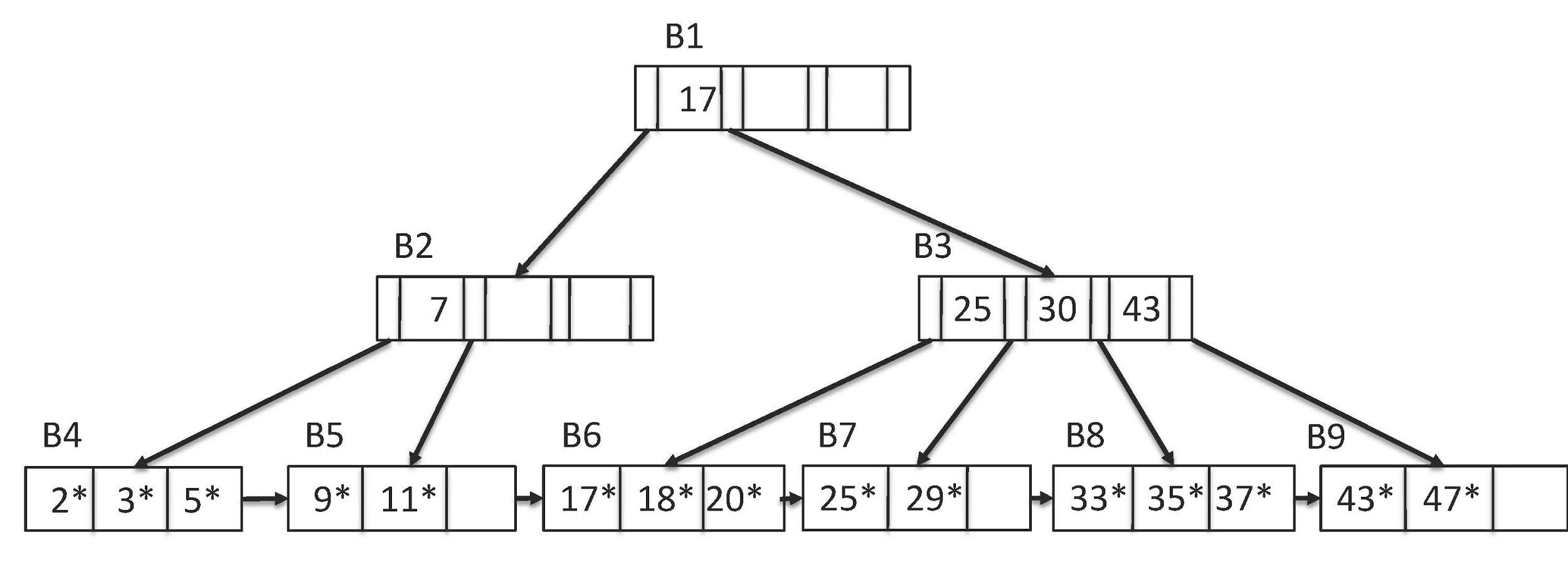
1. Delete record with key 11.

A close up of a logo

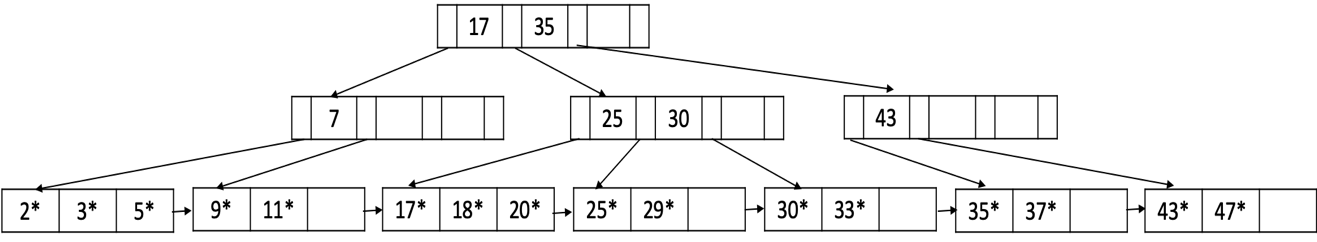
Description automatically generated

##### **Extra Credit (10 points)**

What is the minimum number of additional insertions that would cause the root to split? Explain your answer saying where the insertions are occuring. Assume that the keys are real-valued, not integers.



1. From the beginning B+ tree, we know that at least 3 values needs to be inserted into the B1 block for the root the split, to make the inserted key value in leaf node to pop up to the root, we can choose to insert a value between 17 and 25 or 30 and 43. We insert 30 as an example:



2. Now, there is no possible key that we can insert it and pop up to the root node. So, we start with the leaf nodes’ block that has most nodes and the parent block has most nodes. So we can insert a value between 17 and 25. Suppose we insert 21:

A close up of a logo

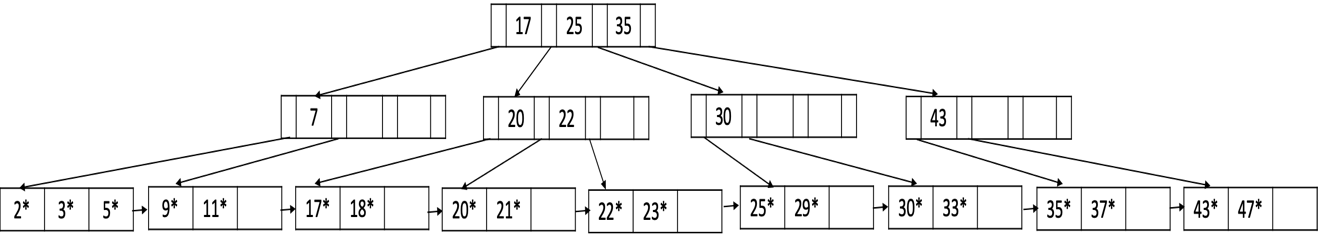
Description automatically generated

3. Now we can insert any value between 17 and 35, it will fill up the any middle four blocks but will not pop up value to the parent blocks. Suppose we have 22:

A close up of a logo

Description automatically generated

4. Now we insert value between 20 and 25, it will cause the split of leaf node block and its parent block and add a new value to the root. Suppose we have 23:



5. Now we can insert a value between 20 and 22 or 22 and 25 to fill up one of the blocks (either 20,21 or 22,23), suppose we have 22.5:

A close up of a logo

Description automatically generated

6. Now we can add a value between 22 and 25 to split it. Suppose we have 24:

A close up of a logo

Description automatically generated

7. Now we can add any value between 11 and 25 to fill up any four of the blocks (17,18; 20,21; 22,22.5; 23,24;). Suppose we have 19:

A close up of a logo

Description automatically generated

8. Now we add any value between 17 and 20 will cause a split on the root. Suppose we have 19.5:

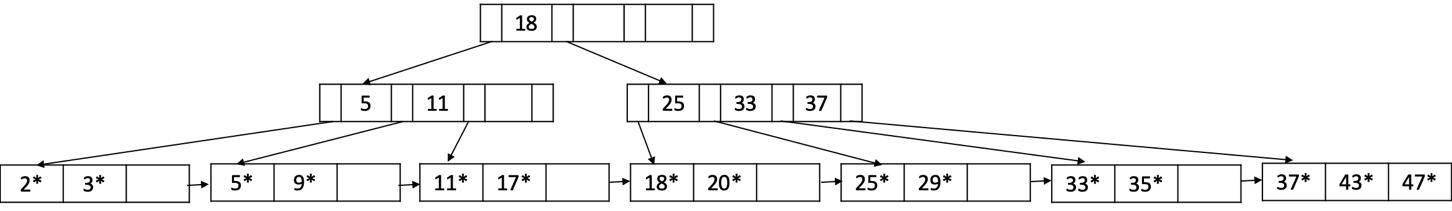
A close up of a logo

Description automatically generated

So we have at least 8 insertion.

##### 5. (10 points)

##### Suppose you wish to bulk load the records shown at the leaves of the tree in the previous question: 2, 3, 5, 9, 11, 17, 18, 20, 25, 29, 33, 35, 37, 43, 47. Since the initial step in doing so is to sort the records, that step has already been performed. Assuming that each data entry page holds a maximum of 3 and a minimum of 2 records, show what the resulting tree would look like after bulk-loading (see Ch. 10.8.2).

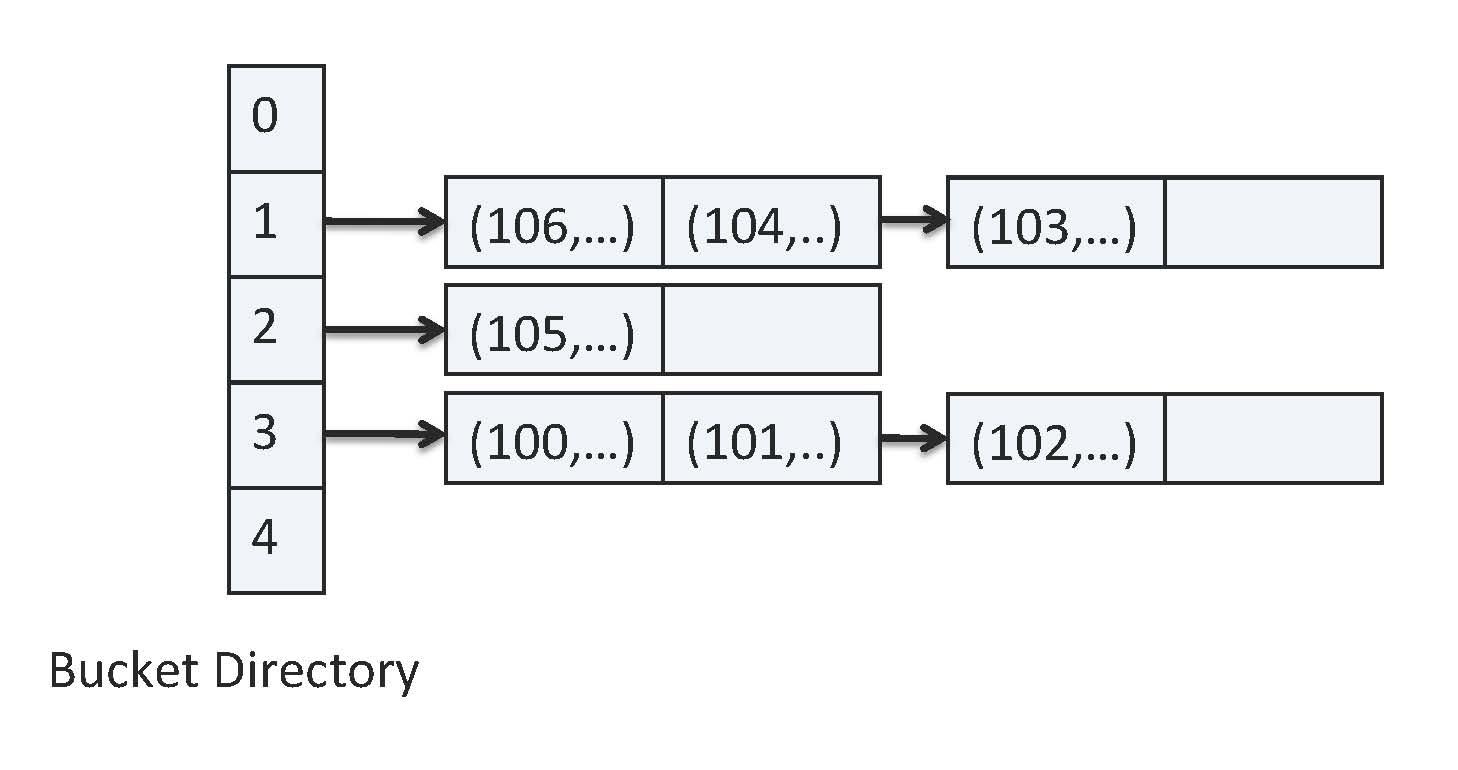


6. (15 points)

Consider the following relational instance of the Internet Movie Database(IMDb).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **imdbID** | **title** | **yearReleased** | **gross(M)** | **rating** |
| 100 | Practical Magic | 1998 | 46 | 7 |
| 101 | Dark Knight | 2008 | 532 | 9 |
| 102 | Beetle Juice | 1998 | 73 | 7 |
| 103 | Heartbreakers | 2001 | 40 | 6 |
| 104 | Shrek | 2001 | 267 | 8 |
| 105 | The Simpsons Movie | 2007 | 183 | 8 |
| 106 | The Holiday | 2006 | 63 | 7 |

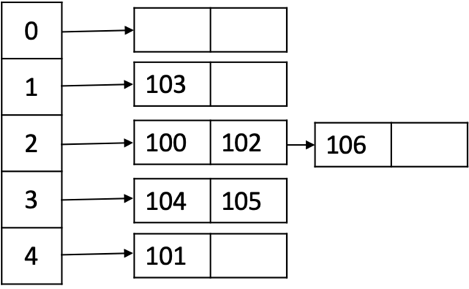
You are storing the relation as an index organized file using hashing. The hash index (a.k.a. bucket directory) can fit on one page, but only two records of IMDb can fit on one page. The following shows what the structure would look like using the hash function yearReleased mod 5. Note that each bucket entry *i* points to a heap of records that hash to *i*. Any page after the first in a heap is called an overflow page (e.g. the one containing (103,...)).



1. (3 points)

Show what this structure would look like using the hash function

h(rating)= (rating) mod 5.



1. (4 points)

Using the hash function in your favorite programming language, calculate a hash using title and distribute over 5 buckets. Show your code as well as the resulting hash structure.

hashtitle=[[] for i in range(0,5)]

title=["Practical Magic", "Dark Knight","Beetle Juice","Heartbreakers","Shrek","The Simpsons Movie","The Holiday"]

for item in title:

hashindex=hash(item)%5

if item not in hashtitle[hashindex]:

hashtitle[hashindex].append(item)

print(hashtitle)

[['Practical Magic', 'Shrek'], ['Beetle Juice', 'The Simpsons Movie'], ['Dark Knight', 'The Holiday'], ['Heartbreakers'], []]

1. (4 points)

List three queries that you expect will be frequently executed over this table.

1. SELECT \* FROM IMDB WHERE yearReleased>=2000 AND yearReleased<=2010

2. SELECT \* FROM IMDB WHERE rating >=8

3. SELECT TOP 10 PERCENT FROM IMDB ORDER BY gross(M) DESC

1. (4 points)

Assuming now that the number of buckets is much bigger than 5, what field/fields do you think should be used to create a hash index and why?

Title can be a good choice, because almost every movie title is different from each other, so it has a very wide range. Then, by using hash(title), an integer is generated from the movie title. We can divide this integer by the number of buckets. This will give us a wide range even distribution hash index on title. A large bucket number and an even distribution of data in buckets is good for the performance of hash index.