AIMAN SIDDIQUA – 2K18/MC/008

DATABASE MANAGEMENT SYSTEM (MC-302)

ASSIGNMENT-3

1. (i) Write-read conflict

T_1: R(X) R(Y) W(X)

T_2: R(X) R(Y) W(X) W(Y)

(ii) Read-write conflict

T_1: R(X) R(Y) W(X)

T_2: R(X) R(Y) W(X) W(Y)

(iii) Write-write conflict

T_1: R(X) R(Y) W(X)

T_2: R(X) R(Y) W(X) W(Y)

(iv) Write-Read conflict : T_2 will not get a lock on X until T_1 commits.

Read-write conflict : T_1 will not get Exclusive lock on X until T_2 commits.

Write-write conflict : If T_2 doesn't commit T_1 doesn't get a lock.

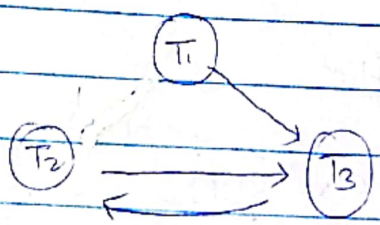
2. (i) $R_1(X)$ $R_2(X)$ $W_1(X)$ $W_2(X)$

Not serializable, Not conflict Serializable, Not view-serializable, Recoverable, Avoids cascading aborts. NOT strict

(2) $W_1(X)\ R_2(Y)\ R_1(Y)\ R_2(X)$

Serializable, conflict serializable, view serializable,
Does not avoid cascading aborts, NOT strict.
Recoverability can't be decided since commit or abort sequence is not specified.

(3) $R_1(X)\ R_1(Y)\ W_1(X)\ R_2(Y)\ W_3(Y)\ W_1(X)\ R_2(Y)$



NOT conflict serializable
View and serializability cannot be decided.
NOT avoids cascading aborts
NOT strict
Recoverability cannot be decided since commit or abort sequence is not specified.

(4) $R_1(X)\ W_2(X)\ W_1(X)\ Abort_2\ Commit_1$

Serializable, conflict serializable & view serializable
Recoverable, Avoids cascading aborts
NOT strict

(5) $R_1(X)\ W_2(X)\ W_1(X)\ Commit_2\ Commit_1$

Serializable, NOT Conflict Serializable, view Serializable
Recoverable, Avoids cascading aborts
NOT strict.

(6) $W_1(X)$ $R_2(X)$ $W_1(X)$ $Commit_2$ $Abort_1$!

Serializable, view serilizable, conflict serizable
NOT recoverable, DOES NOT avoid cascading aborts
NOT strict

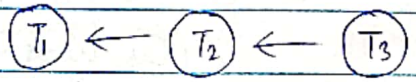
(7) $R_1(X)$ $W_3(X)$ $commit_3$ $W_1(Y)$ $commit_1$ $R_2(Y)$
$W_2(Z)$ $Commit_2$
Serializable, view serializable, conflict serializable
Recoverable, Avoids cascading aborts, Strict.

(8) $R_1(X)$ $W_2(X)$ $W_1(X)$ $R_3(X)$ $Commit_1$ $Commit_2$ $Commit_3$
Serializable, view serializable, NOT conflict serializable
Recoverable, DOES NOT avoid cascading aborts,
NOT strict.

3. (i)

Time	t_1	t_2	t_3	t_4	t_5	t_6	t_7
T_1	X(A)						S(C)
T_2			S(B)	S(C)		S(A)	
T_3		S(C)			X(B)		
LM	G	G	G	G	B	B	G

(ii) Wait-for graph for the lock requests at time-tick t_7.

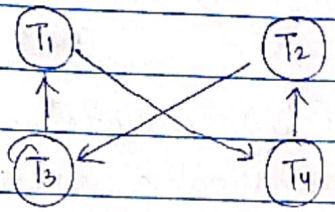


(iii) A deadlock does not exists because there is no cycle in the dependency graph.

(4.) (i)

Time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
T_1	X(B)			S(A)				
T_2					X(D)	X(C)		
T_3			S(C)				X(B)	
T_4		X(A)						S(D)
ZM	G	G	G	B	G	B	B	B

(ii) Wait-for graph for the lock requests.



A deadlock exists because there is a cycle in the dependency graph.

(iii) X(A) at t_2 : G — $T_1 < T_2 < T_3 < T_4$

S(C) at t_3 : G

S(A) at t_4 : A (T_1 aborts)

X(D) at t_5 : G

X(C) at t_6 : A (T_2 aborts)

X(B) at t_7 : G (No locks held on B since T_1 was aborted)

S(D) at t_8 : G (No locks held on D since T_2 was aborted)

(iv) X(A) at t_2 : G.

S(C) at t_3 : G

S(A) at t_4 : ~~(T_5 aborts)~~

X(D) at t_5 : G

X(C) at t_6 : B

X(B) at t_7 : G1 (T_1 aborts)

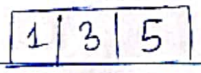
S(D) at t_8 : G1 (T_2 aborts)

5. B-Tree

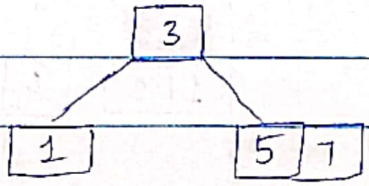
(a) Insert : 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

Insert 1 :

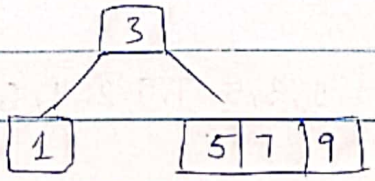
Insert 3, 5:



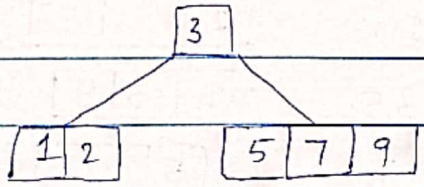
Insert 7:



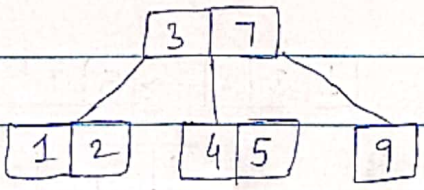
Insert 9 :



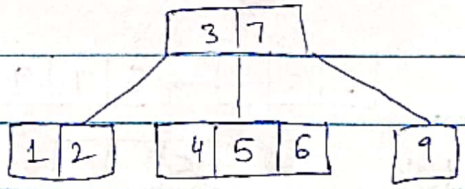
Insert 2:



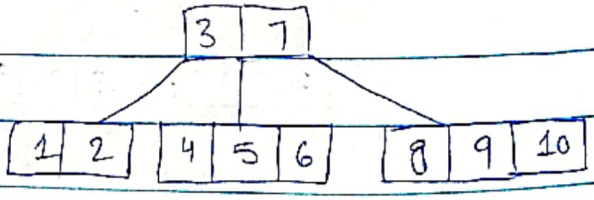
Insert 4:



Insert 6:

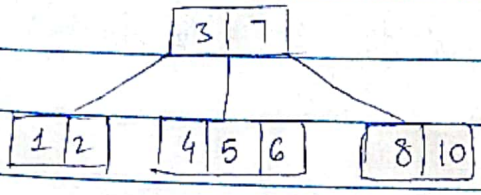


Insert 8 and 10

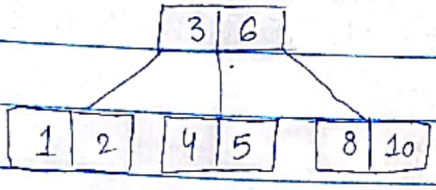


(b) Delete 9, 7, 8

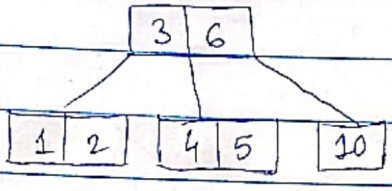
Delete 9:



Delete 7:



Delete 8:

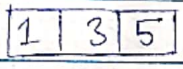


B+ TREES.

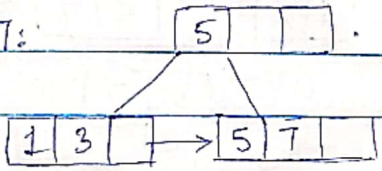
(a) Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

Insert 1:

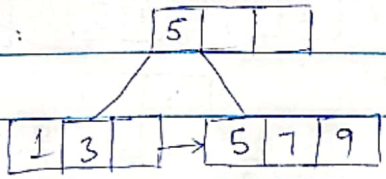
Insert 3, 5:



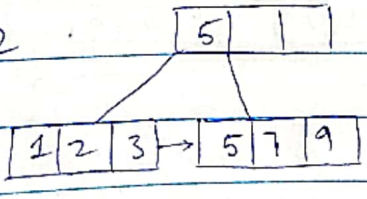
Insert 7:



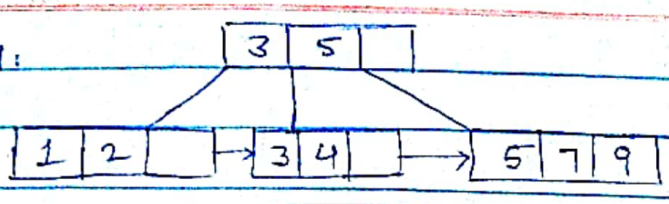
Insert 9:



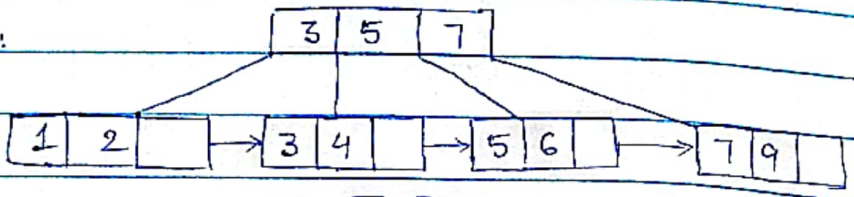
Insert 2



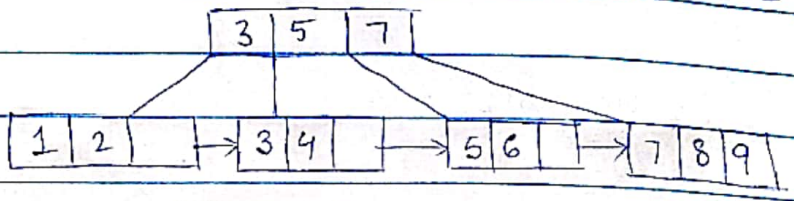
Insert 4:



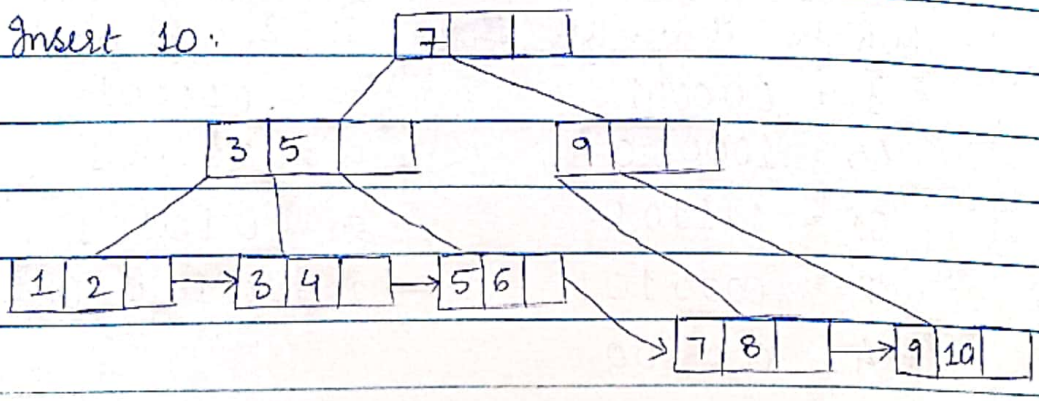
Insert 6:



Insert 8:

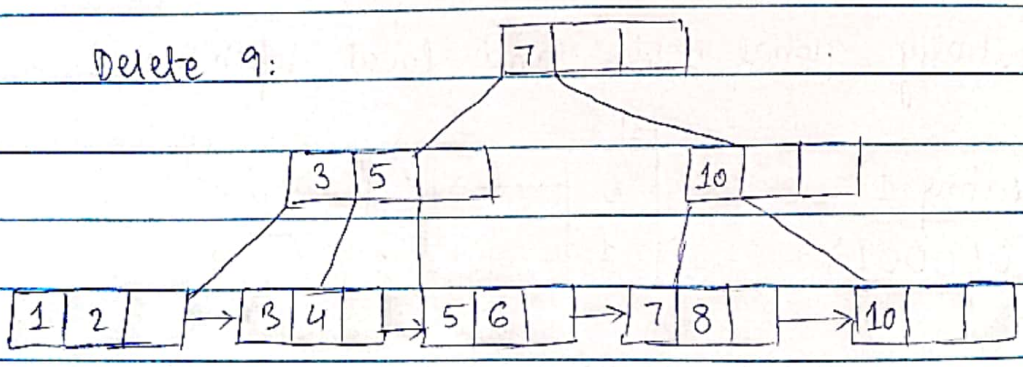


Insert 10:

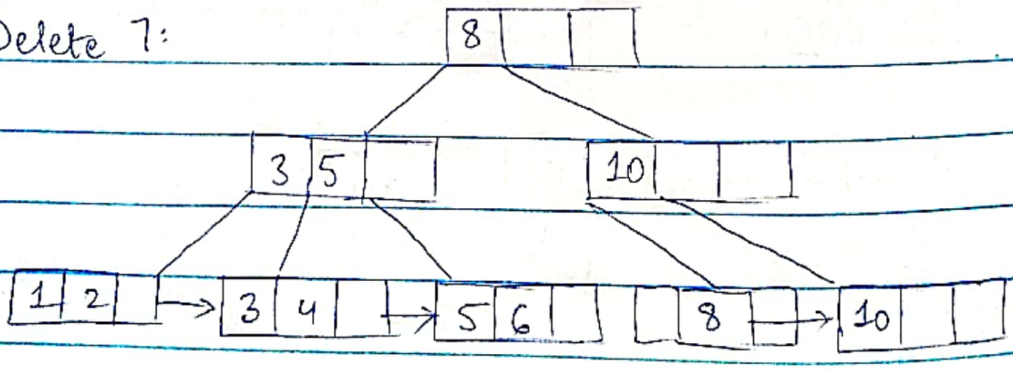


(b) Delete 9, 7, 8

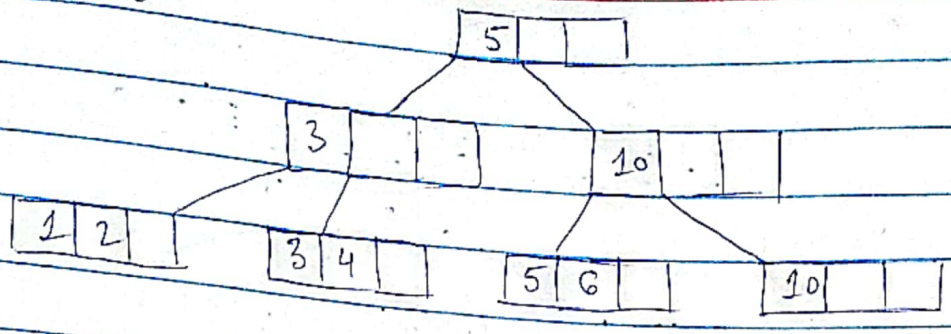
Delete 9:



Delete 7:



Delete 8 :



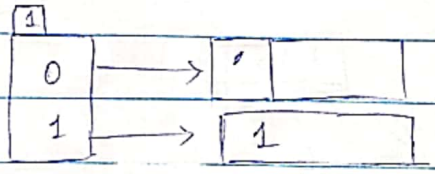
(6) Bucket Size = 3.

Hash the elements : 1, 40, 61, 2, 4, 15, 30, 17, 9, 20, 26

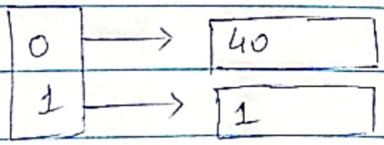
1 = 000001
40 = 101000
61 = 111101
2 = 000010
4 = 000100
15 = 001111
30 = 011110
17 = 010001
9 = 001001
20 = 010100
26 = 011010

Initially global depth and local depth is 1.

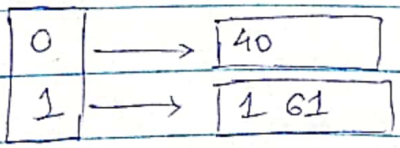
Inserting 1 (000001)



Inserting 40 (101000)



Inserting 61 (111101)



Inserting 2 (000010)

0	→	40 2
1	→	1 61

Inserting 4 and 15 (000100) (00 1111)

0	→	40 2 4
1	→	1 61 15

Inserting 30 : (011110) Will result in overflow. Directory expansion takes place. Global depth = 2

00	→	40 4
01	→	1 61
10	→	2 30
11	→	15

Inserting 17 : (010001)

00	→	40 4
01	→	1 61 17
10	→	2 30
11	→	15

Inserting 9 : (001001) Will result in overflow. Directory expansion takes place. Global Depth = 3

000	→	40
001	→	1 17 9
010	→	2 26
011		
100	→	4 20
101	→	61
110	→	30
111	→	15

Inserting 20 and 26

Ans

7. (a) STATIC HASHING

In static hashing, the resultant data bucket address will always be the same.

For eg. If we generate an address for EMP_ID = 103 using the hash function mod(5) then it will always result in the same bucket address 3.

Hence in static hashing the number of data buckets in memory remains constant throughout.

(b) Most common hash functions used:

(i) Hashing by Division - The mod method.

The key is mapped into one of the slots by taking the remainder of key divided by table_size.

(ii) Hashing by Multiplication.

- The key K is multiplied by a constant real number c in the range $0 < c < 1$ and the fractional part is extracted. (K * c)
- This value is multiplied by table size m and floor of the result is taken.

$$h(K) = \text{floor}(m * \text{frac}(K * C))$$

(c) Common collision resolution techniques:

(i) REHASHING

It invokes a secondary hash function which is applied continuously until an empty slot is found.

(ii) CHAINING

It builds a linked list of items whose key hashes to the same value.

(iii) OPEN ADDRESSING

In open addressing we keep probing until an empty slot is found.

It can be done in one of the 3 ways

1. Linear Probing
2. Quadratic Probing
3. Double Hashing.

(d) STATIC HASHING	DYNAMIC HASHING
1) A fixed no. of bucket is allocated to a file to store the records	1) It allows the number of buckets to vary dynamically.
2) It uses a fixed hash function to partition the set of all possible search-key values into subsets and then maps each subset to a bucket.	2) It uses a second stage of mapping to determine the bucket associated with some search-key value.
3) Performance degrades due to the bucket overflow.	3) Performance does not degrade as the file grows.
4) In static hashing the resultant data bucket address is always the same.	4) In dynamic hashing the data buckets change depending on the records