CS1555 Recitation 13 Solution

Objective: To practice B+-tree and Concurrency Control

Part 1: B+Tree

1. Consider the B+ tree index of order *n* = 5 shown in Figure 10.1.
   - i. Show the tree that would result from inserting a data entry with key 9 into this tree.
   - ii. Show the B+ tree that would result from inserting a data entry with key 3 into the original tree.
   - iii. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible redistribution.
   - iv. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the right sibling is checked for possible redistribution.
   - v. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 46 and then deleting the data entry with key 52.
   - vi. Show the B+ tree that would result from deleting the data entry with key 91 from the original tree.
   - vii. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 59, and then deleting the data entry with key 91.
   - viii. Show the B+ tree that would result from successively deleting the data entries with keys 32, 39, 41, 45, and 73 from the original tree.

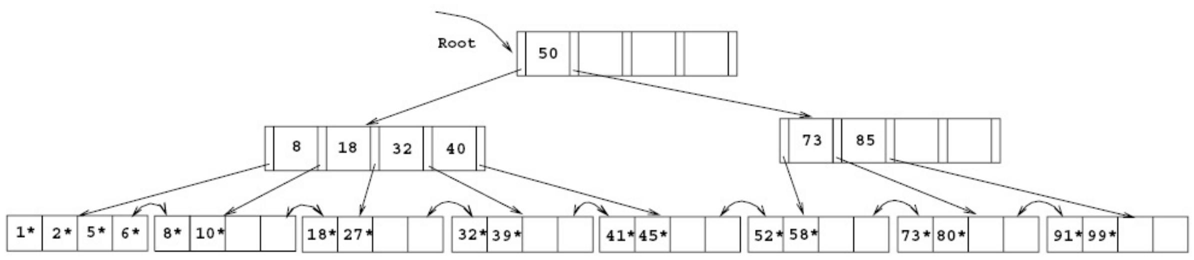


Figure 10.1 Tree for Exercise 10.1

***Answer**:*

i. *The data entry with key 9 is inserted on the second leaf page. The resulting tree is shown in figure 10.2*

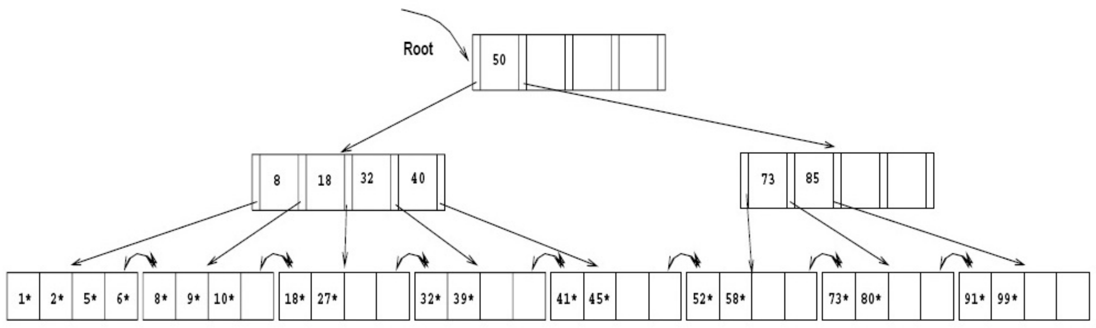


Figure 10.2

ii. *The data entry with key 3 goes on the first leaf page F. Since F can accommodate at most four data entries (n = 5), F splits. The lowest data entry of the new leaf is given up to the ancestor which also splits. The result can be seen in figure 10.3.*

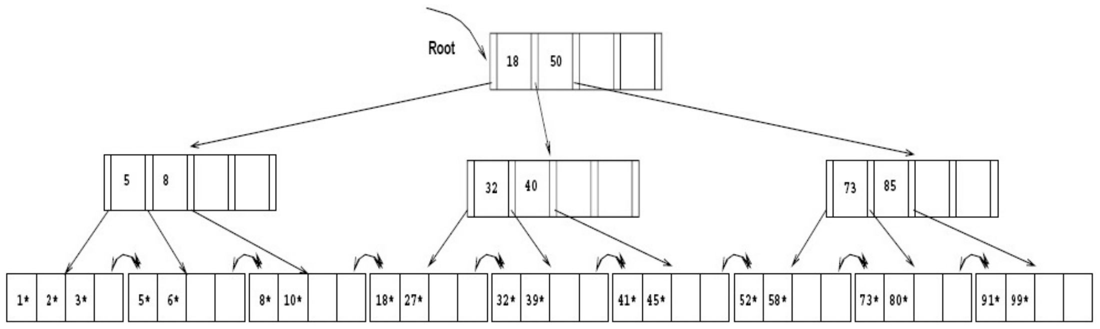


Figure 10.3

iii. *The data entry with key 8 is deleted, resulting in a leaf page N with less than two data entries. The left sibling L is checked for redistribution. Since L has more than two data entries, the remaining keys are redistributed between L and N, resulting in the tree in figure 10.4.*

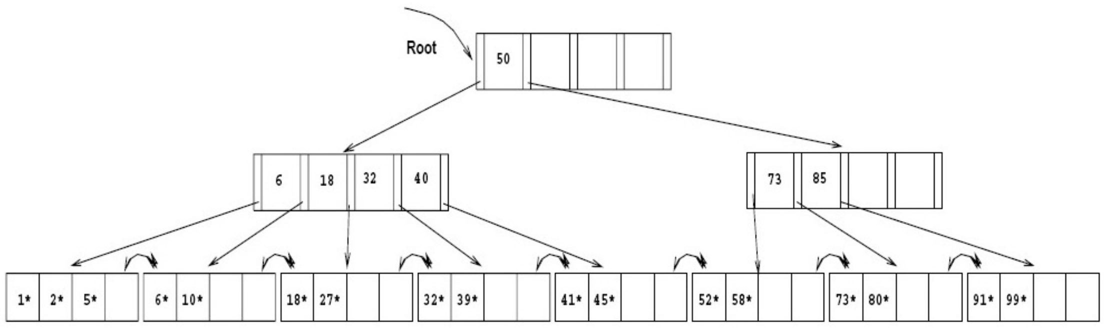


Figure 10.4

iv. *As is part 3, the data entry with key 8 is deleted from the leaf page N. N's right sibling R is checked for redistribution, but R has the minimum number of keys. Therefore the two siblings merge. The key in the ancestor which distinguished between the newly merged leaves is deleted. The resulting tree is shown in figure 10.5.*

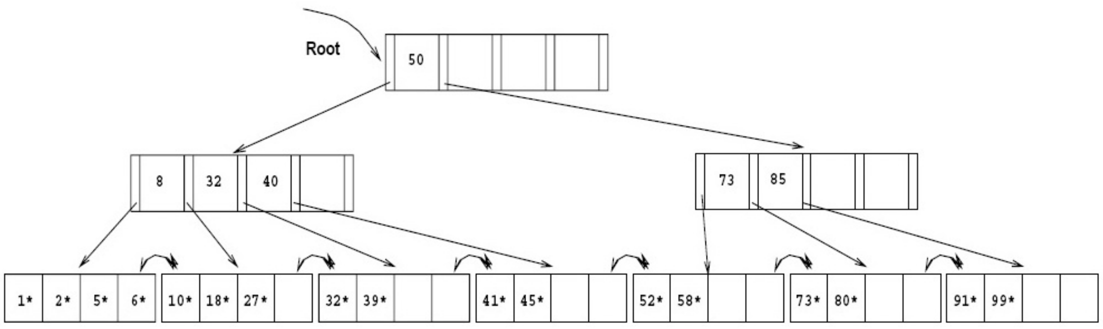


Figure 10.5

v. *The data entry with key 46 can be inserted without any structural changes in the tree. But the removal of the data entry with key 52 causes its leaf page L to merge with a sibling (we chose the right sibling). This results in the removal of a key in the ancestor A of L and thereby lowering the number of keys on A below the minimum number of keys. Since the left sibling B of A has more than the minimum number of keys, redistribution between A and B takes place. The final tree is depicted in figure 10.6.*

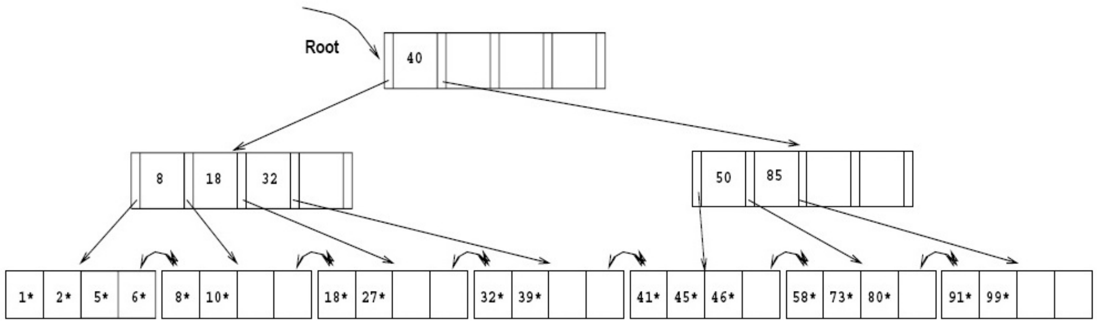


Figure 10.6

vi. *Deleting the data entry with key 91 causes a scenario similar to part 5. The result can be seen in figure 10.7.*

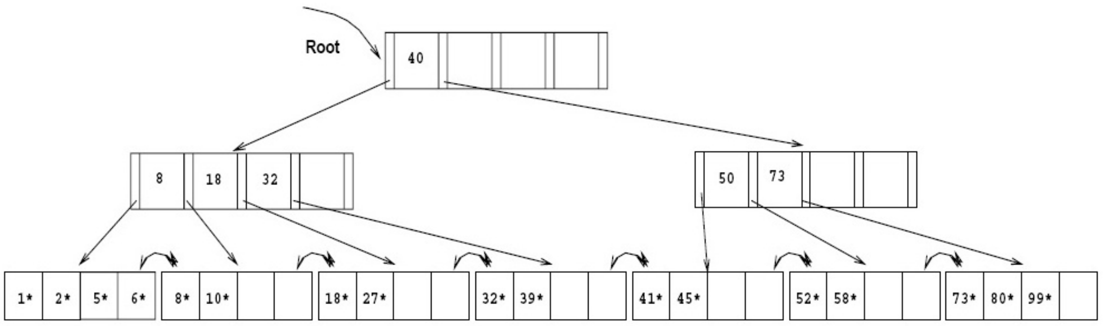


Figure 10.7

vii. The data entry with key 59 can be inserted without any structural changes in the tree. No sibling of the leaf page with the data entry with key 91 is affected by the insert. Therefore deleting the data entry with key 91 changes the tree in a way very similar to part 6. The result is depicted in figure 10.8.

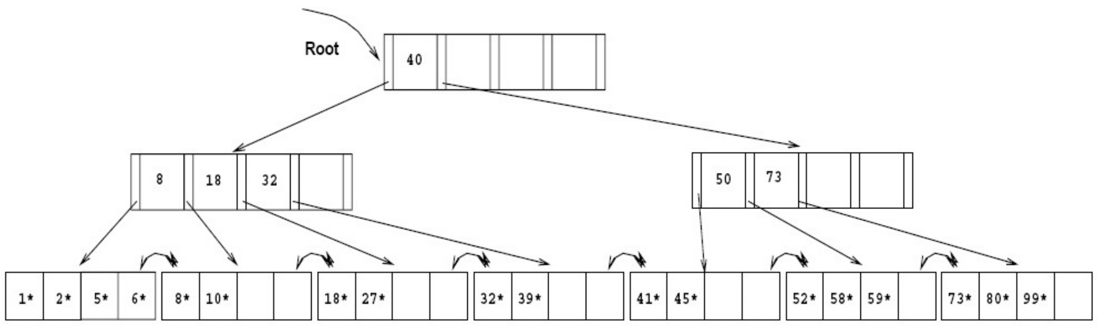


Figure 10.8

viii. Considering checking the right sibling for possible merging first, the successive deletion of the data entries with keys 32, 39, 41, 45 and 73 results in the tree shown in figure 10.9.

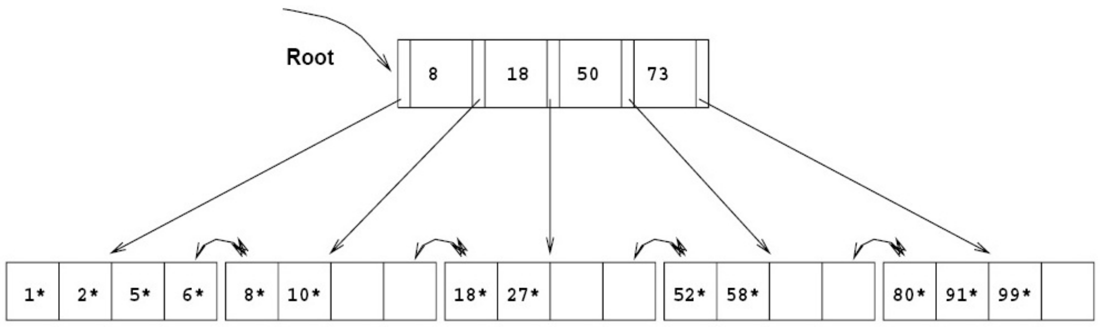


Figure 10.9

Part 2: Concurrency Control

1. Consider the following two transactions:

```
T1:   r1(A) ;                       T2:   r2(C);
      r1(B);                              r2(B);
      If A=0 then B:= B+1;                r2(A);
      w1(B);                              if B>C then {A:= A+1; C:=C+1;}
                                          w2(C)
                                          w2(A)
```

- For each of the following histories/schedules:

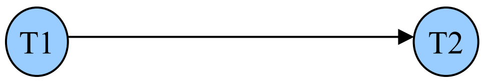
  a) Is it a valid history?
  b) Use *serializability graphs* to check whether it is serializable or not, and if it is, what is the equivalent serial history/schedule

  H1: r1(A) r1(B) r2(C) w1(B) r2(B) r2(A)w2(C) w2(A)
  H2: r1(A) r1(B) r2(C) r2(B) w1(B) r2(A)w2(C) w2(A)

Answer:

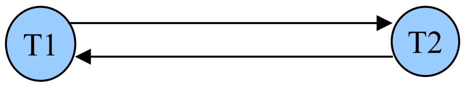
a) *Yes, they are valid histories because the order of statements in each transaction is preserved.*

b) Serializability graph for H1:


$$\begin{tikzcd} T1 \arrow[r] & T2 \end{tikzcd}$$

The graph is acyclic, so the history is serializable. The equivalent serial history is <T1, T2>

Serializability graph for H2:


$$\begin{tikzcd} T1 \arrow[r, shift left] & T2 \arrow[l, shift left] \end{tikzcd}$$

The graph contains a cycle, so it is not serializable.

2. Consider the following history, with lock and unlock statements added for each transaction:

a) Does the history follow 2PL protocol?

Yes. All locks are requested before the data is used and they are released only before the commit step.

b) Did deadlock happen?

We use the wait-for graph to check this, as follows:

T1	T2	Wait-Free

rl1(A)
r1(A)
		yes

rl1(B)
r1(B)
		yes

	rl2(C)
	r2(C)
		yes

	rl2(B)
	r2(B)
		yes, because is a read lock after another read lock from T1

wl1(B)
w1(B)
		no, T1 waits for T2
		(add T1 → T2 to the WFG)

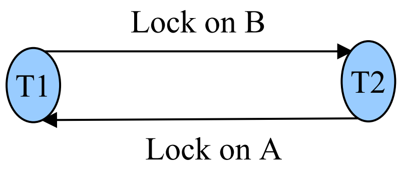
	rl2(A)
	r2(A)
		yes

	wl2(C)
	w2(C)
		yes

	wl2(A)
	w2(A)
		no, T2 waits for T1
		(add T2 → T1 to the WFG)
		(deadlock happens!)

unlock1(A),
unlock1(B)
Commit
	unlock2(C),
	unlock2(A)
	unlock2(B)
	Commit	

The corresponding wait-for graph contains a cycle, which means deadlock has happened.


Lock on B
T1
T2
Lock on A