Assignment 3 – solutions

Problem 1.

1. Some database operations (e.g., certain implementations of the join relational algebra operator) require repeated sequential scans of a relation. Suppose that there are 10 frames available in the buffer pool, and the file to be scanned has 11 or more pages (i.e., at least one more than the number of available pages in the buffer pool). Using LRU, every scan of the file will result in reading in every page of the file! In this situation, called 'sequential flooding', LRU is the worst possible replacement strategy.
2. Recall the example I gave in the class for a main memory with 5 frames and file with 6 pages.

Problem 2.

For Alternative (2), we use the following notation <A, (p,s)> with the meaning that A is the search key, (p, s) is the record id, where p is the page number of the entry and s is the location on page p of the entry. For Alternative (3), the notation is the same, but with the possibility of additional rid's listed.

1. Contradiction. At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
2. The order of the entries is **not** significant in the index.
   <Andy, (1, 1)>, <Dave, (1, 2)>, <David, (1, 3)> <Eli, (2, 1)>, <John, (2, 2)>
3. The order of the entries is **not** significant in the index.
   Data entries with the same key are now bucketed together. However, keys are all distinct.
   <Andy, (1, 1)>, <Dave, (1, 2)>, <David, (1, 3)>, <Eli, (2, 1)>, <John, (2, 2)>
4. The order is **important** because the order of the entries in the index is the same as the order of data records in the file.
   <Andy, record with SID = 53>, <Dave, record with SID = 54>, <David, record with SID = 44>, <Eli, record with SID = 32>, <John, record with SID = 20>
5. The order is **important** because the order of the entries in the index is the same as the order of data records in the file.
   <Andy, (1, 1)>, <Dave, (1, 2)>, <David, (1, 3)>, <Eli, (2, 1)>, <John, (2, 2)>
6. The order is **important** because the order of the entries in the index is the same as the order of data records in the file. Since our keys are all distinct there will be one entry per distinct key in a bucket/list.
   <Andy, (1, 1)>, <Dave, (1, 2)>, <David, (1, 3)>, <Eli, (2, 1)>, <John, (2, 2)>
7. Contradiction. At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
8. The order of the entries is **not** significant in the index.
   <10k, (1, 1)>, <10k, (2, 2)>, <20k, (1, 2)>, <20k, (2, 1)>, <30k, (1, 3)>
9. The order of the entries is **not** significant in the index. Records with the same key are placed in the same list.
   <10k, (1, 1), (2, 2)>, <20k, (1, 2), (2, 1)>, <30k, (1, 3)>
10. Contradiction. At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)

11. We cannot build a clustered index using Alternative (2) since the order of the records sorted by SALARY is not the same as that on NAME.
12. Same as above.
13. Option 4 is the best. The other good options are options 5 and 6, but they have an additional level of indirection from index to file, which can increase the number of I/Os.
14. We need to do a full scan of the records. We have to choose from among Options 8 and 9. Either of them is good because they give us a index-only approach, which leads to a significant smaller number of I/Os.

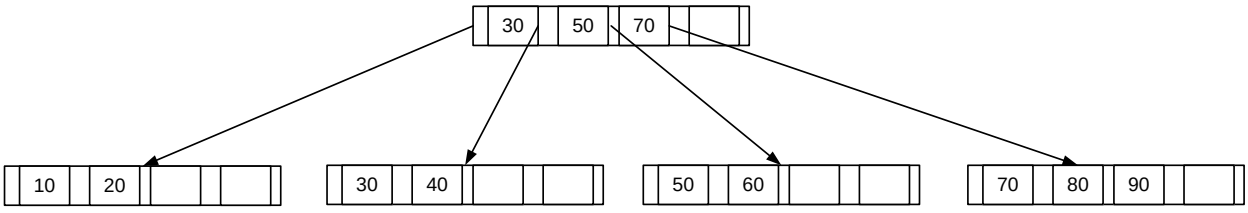
Problem 3.

1. Bytes/track = (bytes/ sector) x (sectors/track) = 1024 x 100 = 102400 bytes = 100KB.
2. Bytes/surface = (bytes/track) x (tracks/surface) = 100KB x 4000 = 400,000KB.
3. Bytes/disk = (bytes/surface) x (surfaces/disk) = 400,000 x 10 x 2 = 80,000,000KB.
4. 4000, i.e., same as the number of tracks.
5. One complete rotation takes 1/7200 in a minute = 1/7200 x 60 seconds ≈ 0.0083 seconds = 8.3 ms. The average rotational delay is half of the rotation time, i.e., 4.15 ms.
6. A track has 100KB. It takes about 8.3ms to make a revolution. Hence, transfer rate is 100KB/8.3ms ≈ 12.05 KB/ms.

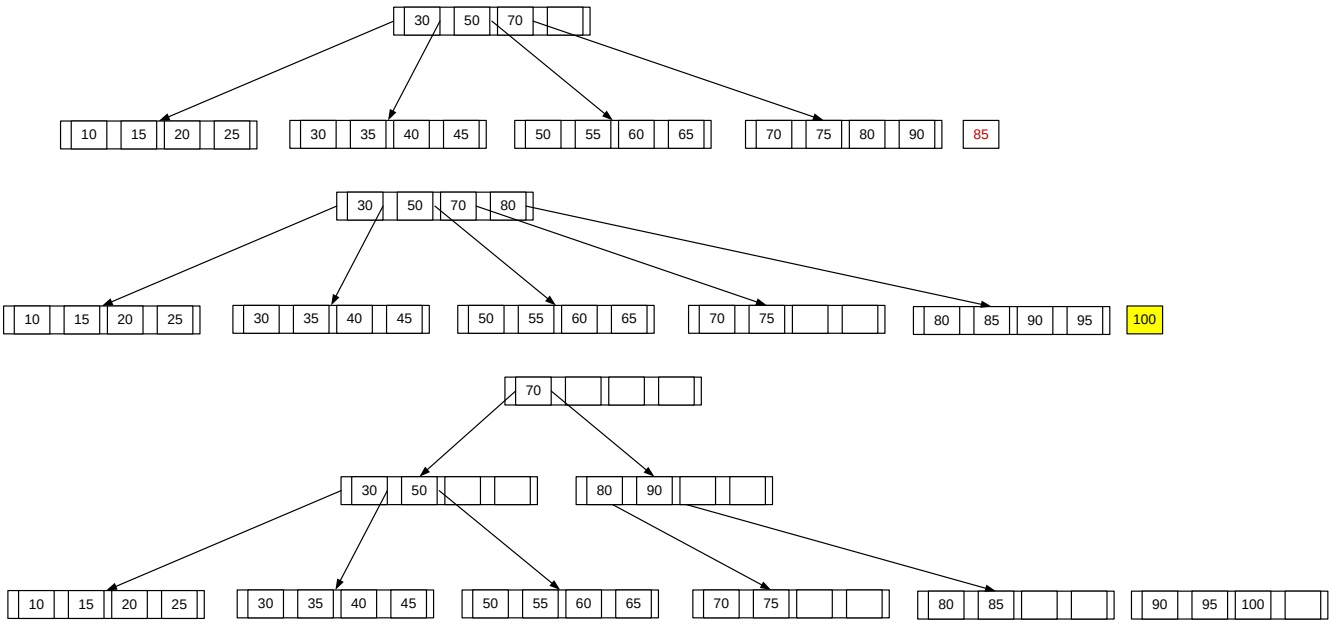
   If you are asked to give the TOTAL transfer time then this is given by:
   Total transfer time = seek time + latency + transfer time = 10ms + 4.15ms + 12.05ms = 26.2ms

Problem 4.

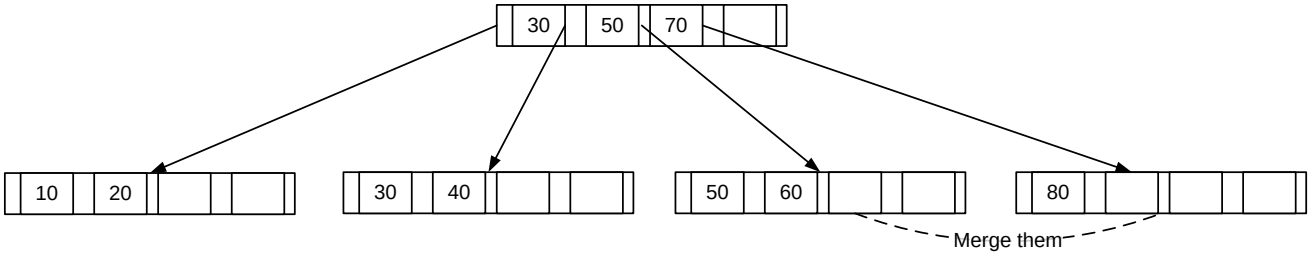
1. See figure below.



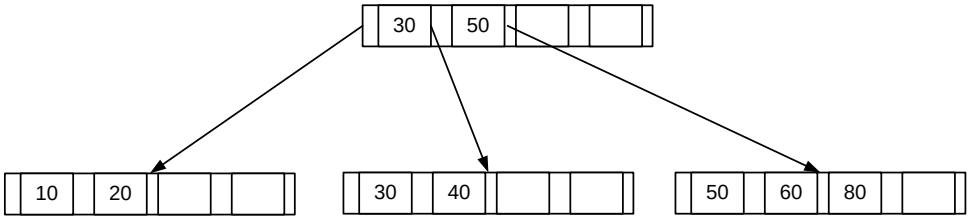
2. See figure below.


30
50
70
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
90
85
30
50
70
80
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
70
30
50
80
90
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100

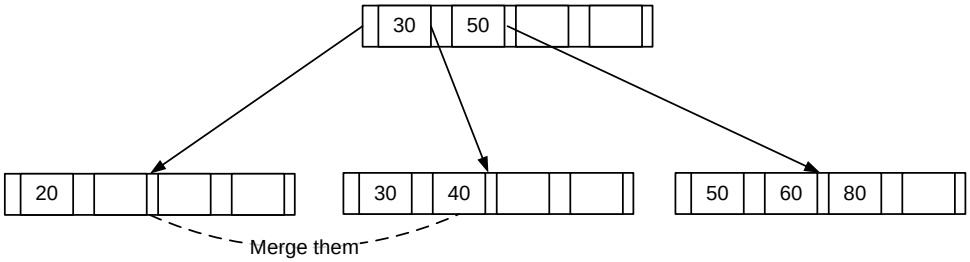
3. See figure below



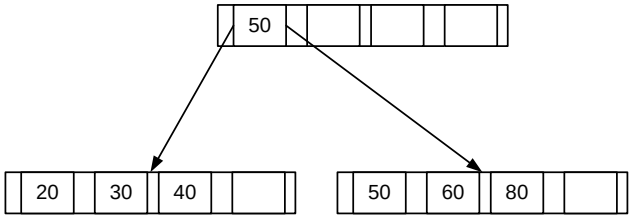
After delete 70 and 90, before merge.



After delete 70 and 90.



After delete 10 and before merge.



Final tree after 70, 90 and 10, in this order, are deleted.