Sample Solutions

Question 1
Number of EMP data page = 100,000. Each leaf node of B+-tree can store about 510 keys (order is 255). Note that some space is wasted under this computation. You could have packed more entries (but we simply follow the standard formula).
So, there are 1961 leaf nodes (assuming the leaf nodes are 100% full), and the tree has 3 levels (root, internal node, leaf nodes). Hash index should have at least 142857 (=1,000,000/7) primary pages/buckets (in practice, the number would be higher).

a) Using the sorted file requires accessing 10% of the data file (=10,000 I/O). Using B+-tree requires accessing at least 10% of the data file (because of the additional overhead in accessing the index). Hashing is at best 100000 (if we hash each of the values assuming sno is of integer type). In the worst case, hashing is not applicable – need to scan the whole file! So, method 1 is the best.
b) Obvious. Hashing is the best, requires 1 I/O (format 1) (If we have used format 2, then it would be 2 I/Os). B+-tree will need 4 I/Os. If we have access to the first page of the file, and it is chained, then we need 2 pages at most. But, in general, this may not be the case, e.g., if we are looking for sno = 429580, then it may take a long while to search. Binary search may be possible (with some additional assumptions). Otherwise, we really need a scan till we find the value.
c) B+-tree is expected to perform best.
d) Accessing the sorted file is the best.

Note: Your answer may be slightly different if you used a different order or assume leaf nodes are less than 100% full.

Question 2
1. 6,000,000 records, each 200 bytes
   10,000 single-record accesses
   100 range queries, each accessing 0.005% of the file, i.e., 300 records

<u>Hash</u>
Assume that records are stored in hash table.

bucket size = 4096 bytes => 20 records
since no overflow, and 70% load factor ==> each bucket contains 14 records only.
Total number of buckets = 400,000+

for 10,000 single-record accesses, cost = 10,000 I/O (i.e., 1 I/O per access)
for 100 range queries, we need 300 accesses per query (assumption: BookIDs are integers and they are consecutive)
so cost = 100*300 = 30,000

So, total cost = 40,000 I/O.

Note that there is a certain range whereby it is cheaper to scan the entire file per range query. For example, if the range queries access more than 20% of the data, then scanning the file would be faster!

B+-tree
Assume that (key,ptr) pairs are stored in leaf nodes.

each node = 4096 bytes.
let order be d => 2d*8 + (2d+1)*4 <= 4096 => d = 170
=> each node can store at most 340 keys.
since each node is 70% full, we have each node storing 238 keys
(and 239 pointers).

=> at leaf level, we have 6,000,000/238 = 25211 nodes
=> at level above leaf, we have 25211/239 = 105 nodes
=> next level is the root.
=> the tree has 3 levels.

for 10,000 single-record accesses, cost = 10,000*(3+1) = 40,000
for each range query, we need to traverse 2 leaf nodes, and 22 data
nodes (assuming data are clustered).
so, the cost for 100 range queries = 100*(3+1+22) = 2600

total = 42,600

Note: It is possible that 3 leaf nodes are accessed. It is also possible that 23 data nodes are accessed. As long as you can understand and justify your answers, then its fine.

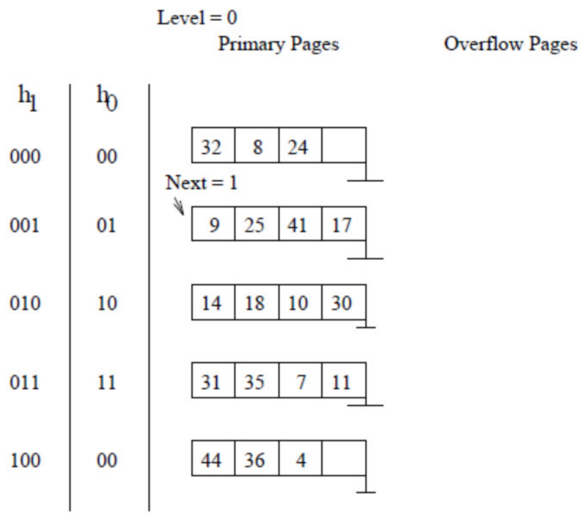
the winner is the hashing scheme (given the assumptions). NOTE: If we did not make any assumptions, each range query would require 400000+ I/Os (need to scan all buckets!) and the hashing scheme would have been the loser.

The loser will outperform the winner when the number of range queries increases or the range increases, or the number of single-record queries reduces.

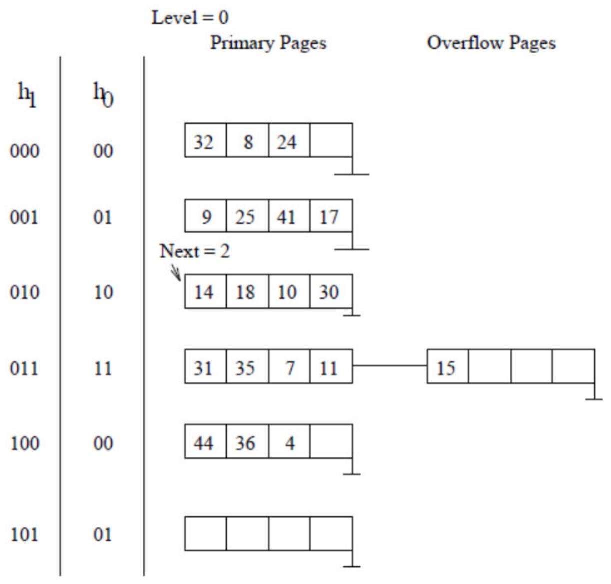
If hashing loses (by relaxing the assumption), then it’s the other way round – increase the number of single-record queries and reduce the number of range queries may help.

Question 3
(a) If the last item that was inserted had a hashcode h0(keyvalue) = 00 then it caused a split (as there are no overflow pages for any of the other buckets), otherwise,
any value could have been inserted.

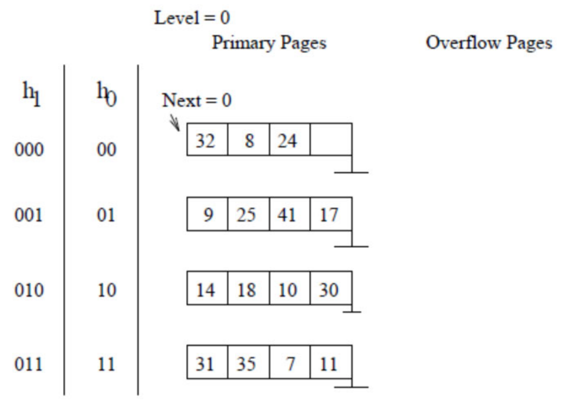
(b) Hash index after inserting 4.



Hash index after inserting 15.



(c) Hash index after deleting 36 and 44 into the original tree.



(d) The following constitutes the minimum list of entries to cause two overflow pages in the index :

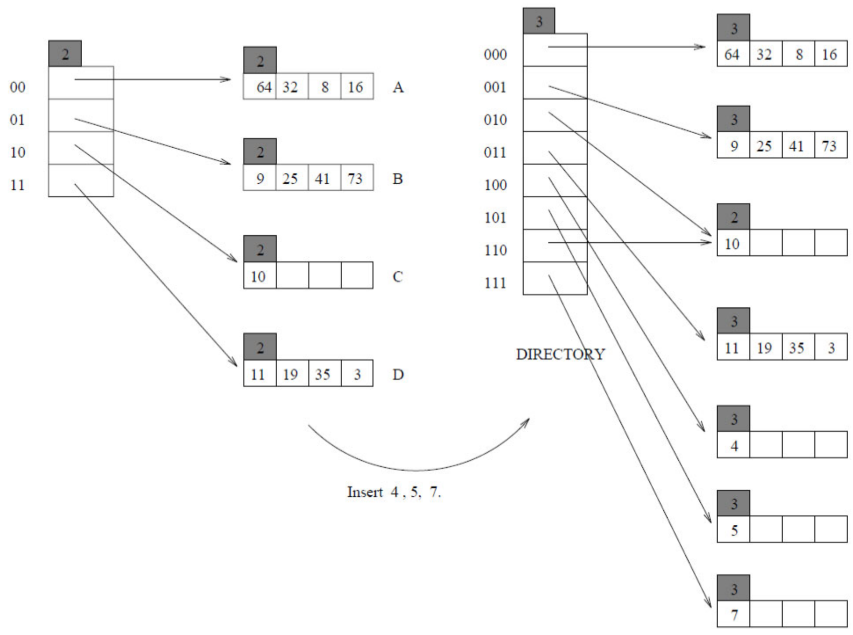
63, 127, 255, 511, 1023

The first insertion causes a split and causes an update of Next to 2. The insertion of 1023 causes a subsequent split and Next is updated to 3 which points to this bucket. This overflow chain will not be redistributed until three more insertions (a total of 8 entries) are made.

Note that this answer is not unique. You can also insert 1 to bucket(01); since 9 (1001), 25 (11001), 41 (101001), 17 (10001) are all 001, there will be no splitting. If you keep inserting "001"-entries, there will be 2 overflow pages to be created, so this is another valid answer.

Question 4.

Each one of the insertions (4, 5, and 7) causes a split with the initial split also causing a directory split. But none of these insertions redistribute the already existing data entries into the new buckets. So when we delete these data entries in the reverse order (actually the order doesn't matter) and follow the full deletion algorithm we get back the original index.



2
00
01
10
11
2
64 32 8 16
A
2
9 25 41 73
B
2
10
C
2
11 19 35 3
D
Insert 4, 5, 7.
3
000
001
010
011
100
101
110
111
DIRECTORY
3
64 32 8 16
3
9 25 41 73
2
10
3
11 19 35 3
3
4
3
5
3
7