QUESTION 1:

(a) Sector size (1024 bytes) x (#sectors) 128 x (#surfaces) 10 x (#cylinders) 2^12 = 5.12MB.

(b) The min. seek time = 0 ms since this corresponds to the situation that the desired data is on the current track. (Here, we are assuming 0 setup time, since the question doesn't explicitly state the setup time.)

The max. seek time is (2^12 - 1) x 1.002 ms = 4095 x 1.002ms, which is approx. 4.008 s. The average seek time is a third of the maximum and is so 1.336s.

(c) The max. rotational delay is the time for 1 complete revolution, and is = 60/2000 sec.

The average rotational delay is a half of the maximum and is = 60/4000 s = 3/200 s = 15 ms.

QUESTION 2:

File size = 2 x 10^6 x 400 bytes = 800MB = 800MB/4K = 200,000 blocks.
buffer size = 20MB.
Time to read/write a block = time to read 4 sectors = 4/128 x time for one
revolution = 1/32 x 30 ms. = 15/16 ms.
In phase 1, we sort the file one bufferful at a time, producing
40 sorted sublists (40 = 800MB/20MB.)
The time taken for this is approximately time to read and write
all blocks once = 2 x 200,000 x 15/16 ms = 6.25 min.

In phase 2, we will merge all 40 sublists at once. The available buffer space is 20M so we have, say (240 x 2) K sized double buffer for each input
sublist, leaving 20M - (40 x 480K) = 800K (= 400 x 2) for the output double buffer.

In this phase, since we do the merging in one shot, we will read all blocks once and write them back, so the time spent on simply reading/writing the blocks is = 2 x 200,000 x 15/16 = 6.25 min (exactly as for phase 1).

The number of fresh (i.e., random) accesses during the read
is the exactly the number of half-bufferfuls there are in the
file. This is because, with double buffering,
we will at once read half the size of each buffer of each sorted sublist. The number of such chunks is = 800M/240K = 80000/24, so the time spent making these accesses is 80000/24 x (1.336s + 15ms) = 80000/24 x (1.351)s = 80000/24 x 1.351/60 min 75.01 min (approx).

(Recall, we calculated the average seek time
in QUESTION 1(b) as 1.336 s and the average rotational delay in QUESTION 1(c) as 15 ms.)

The number of fresh accesses during the write is the number of chunks in the file that have half the output buffer size = 800M/400K = 2000, so the time spent making these accesses is 2000 x (1.351)/60 min = 45.03 min (approx).

The overall time is of course 6.25 min (phase 1) + 6.25 min (phase 2, sequential) +
75.01 min (phase 2, random reads) + 45.03 min (phase 2, random writes) = 132.54 min.

QUESTION 3:

Assuming each node is implemented using a block (i.e., page), we can pack a maximum of 4000/(16+4) = 200 pairs of keys and pointers in a page. [Note, actual page size of 4K corresponds to 4096 bytes, which will let us pack up to 204 pairs of (key, pointer) pairs. But we ignore this small difference to make the arithmetic simpler.] Thus, the order = min. number of children per internal node (save the root) = 100 and the maximum number of children per internal node = 200.
Size of data file = 2 x 10^6 x 400 = 800M = 800M/4K = 200,000 blocks.

The minimum height B+tree is the one where every node, including the leaves and the root are filled to capacity. Since data entries are the records themselves, this corresponds to having 200,000 leaves. with a fanout of 200, we can cover these leaves with a B+tree of height = ceil(log_200 200000) = ceil(2.3) = 3. [Note: fanout = #children per internal node.]

The maximum height B+tree corresponds to the situation when every node is filled to its minimum allowable occupancy level. For the root, this means one key and two pointers, whereas for every other node, including the leaves, this means "50% full". Specifically, for internal nodes other than the root, this translates to 100 pointers whereas for the leaves this means they contain half as many records as they possibly can. This in turn means the effective number of blocks over which the data is spread is doubled, giving us 400,000 blocks. since the root has exactly two children, one way of calculating the height corresponding to this situation is simply 1 + ceil(log_100 200000) = 1 + 3 = 4. this is because each pointer out of the root covers half the total number of data blocks, i.e., 200000. the height of the subtree rooted at each of the root's children is exactly ceil(log_100 200000), to which we must add 1 to get the B+tree's height.

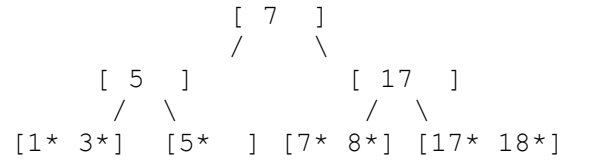
THE FOLLOWING IS NOT EXPECTED, BUT WILL FETCH EXTRA MARKS:

A tighter upper bound of 3 can be shown as follows. To cover 400,000 leaves
we need 400,000/100 = 4,000 parents of leaves, each with a
fanout of 100. Since 100 is the min. allowed fanout for internal nodes
other than the root, we need a minimum of 4,000 parents of leaves.
These in turn necessitate a minimum 4,000/100 = 40 grandparents of
leaves. The only way these grandparents can be covered with a valid B+tree
is with a root having a fanout of 40. This reasoning leads to a tighter
upper bound of 3.
END--"NOT EXPECTED".

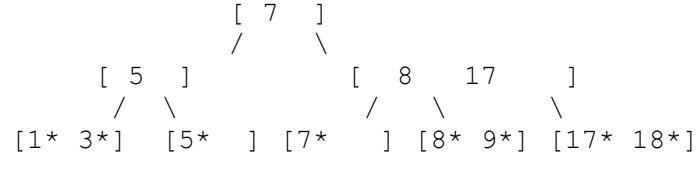
Note: A similar type of "bottom-up" reasoning yields the same minimum
height as calculated above.

(b) The minimum number of blocks needed to hold the B+tree index
corresponds to the situation when every block is filled to capacity.
Let's start with a root with 200 children each of which has 200 children.
this covers 40,000 nodes, so we must go one more level, but then we already
know the number of nodes that must be at the next level, 200,000, the
number of data blocks. so, the index size, not counting the
size of the data itself, is 1 + 200 + 200^2 = 40,201, in number of nodes,
i.e., in number of blocks.

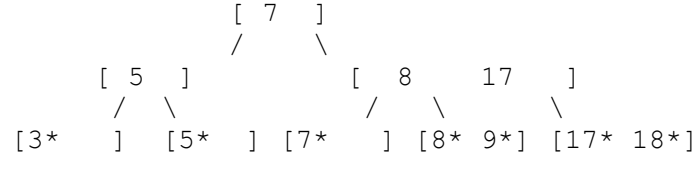
(c) B+tree after inserting 7*:



now, inserting 9* yields:



Now, deleting 1* leads to:



Finally, inserting 3* would leave the B+tree
structure unchanged. whether contents in it
undergoes any change at all depends on:
(a) the representation chosen for data entries
and (b) the key status -- primary or secondary.
If the key is primary, insertion of a duplicate

should be disallowed, so nothing changes in the tree.
If it's a secondary key, then assuming data entries are
(key, list of rids) pairs, we just need to insert an extra
rid in this list for key=3.

(d)
(i) Buckets with addresses 001, 010, and 011 have yet to
be split in the current round.

(ii) Consider inserting 129*. Its hash suffix is 001, so
it would have to be inserted in that bucket, causing an
overflow, and since
Next points to it, it will be split. Because of this,
Next will be advanced to point to bucket 010. For
convenience, here is the modified hash structure.

```
h_1    h_0   primary pages     overflow pages
-----------------------------------------------
000 |  00   [64* | 8* | 48* | ]
001 |  01   [17* | 33* | 41* | 81*]-->[129* | | | ]
010 |  10   [22* | 34* | 26* | 70*]
011 |  11   [43* | 47* | 23* | 27*]
100 |  00   [52* | 44* | | ]
101 |  01   [ | | | ]
```

[Notice that all the entries in bucket 001 still remain there (including
the overflow area), even after splitting. This can happen, depending on
the
distribution of keys we are dealing with.]

(iii) Now inserting 42* would temporarily cause an overflow,
so the bucket pointed to by Next, i.e., bucket 010, should be
split. This will result in:

```
h_1    h_0   primary pages     overflow pages
-----------------------------------------------
000 |  00   [64* | 8* | 48* | ]
001 |  01   [17* | 33* | 41* | 81*]-->[129* | | | ]
010 |  10   [34* | 26* | 42* | ]
011 |  11   [43* | 47* | 23* | 27*]
100 |  00   [52* | 44* | | ]
101 |  01   [ | | | ]
110 |  10   [22* |70*  |     |    ]
```

Next will now point to bucket number 011.