Solutions Manual

Chapter 14

Section 14.1

Exercise 14.1.1

(a) For dense index we need a key-pointer pair for each record, and so will need $\frac{n}{10}$ blocks. For the data, we will need $\frac{n}{3}$ blocks, and so the total number of blocks is $\frac{13n}{30}$.

(b) For the sparse index we need a key-pointer pair for each of the data block, and so will need $\frac{n}{30}$ blocks. For the data, we will need $\frac{n}{3}$ blocks, and so the total number of blocks is $\frac{11n}{30}$.

Exercise 14.1.2

(a) For dense index we need a key-pointer pair for each record, and so will need $\frac{n}{200 \cdot 0.8}$ blocks. For the data, we will need $\frac{n}{30 \cdot 0.8}$ blocks, and so the total number of blocks is $\frac{23n}{480}$.

(b) For the sparse index we need a key-pointer pair for each of the data block, and so will need $\frac{n}{30 \cdot 0.8 \cdot 200 \cdot 0.8}$ blocks. For the data, we will need $\frac{n}{30 \cdot 0.8}$ blocks, and so the total number of blocks is $\frac{161n}{3840}$.

Exercise 14.1.3

(a) We need the same amount of blocks for the data as in 14.1.1 (a). We will need $log_{10}n$ levels of the index with total number of blocks of $1 + 10 + 10^2 + 10^3 + 10^{log_{10}n} = \frac{n-1}{9}$. The total is then $\frac{4n-1}{9}$.

(b) We need the same amount of blocks for the data as in 14.1.1 (b). We will need $log_{10}\frac{n}{3}$ levels for the index with total number of blocks of $1 + 10 + 10^2 + 10^3 + 10^{log_{10}\frac{n}{3}} = \frac{n-3}{27}$. The total is then $\frac{10n-3}{27}$.

Exercise 14.1.4

Since the records could start at any location in the block the number of blocks needed is $\left\lceil \frac{m}{10} \right\rceil + 1$ (e.g. for m=0 we would need one block and for m=10 we would need two blocks). The average of this function over $[0, m]$ is $\left\lceil \frac{m}{20} \right\rceil + 1$.
If movies were randomly distributed over the large number of blocks, the average would be $m + 1$.

Exercise 14.1.5

(a) We would need 1000 (for records) + $\frac{3000}{50}$ (for pointers) + $\frac{300}{10}$ (for pointer-value pairs), for the total of 1090.
Without buckets we would need key-value pair for each record, which is 300 blocks (for the total of 1300).

(b) We still need the same amount of blocks for records and the bucket pointers. The only variable here is the number of blocks for the pointer-value pairs. If all values are different, we need 300 blocks (for total of 1360), and if all values are the same, we need 1 block (for the total of 1061).

Exercise 14.1.6

In the best case, we will only need to read one index file block to locate the key-value pair we need. Further, in the best case, the records with the same value would be clustered (i.e. 3 records per block). Therefore, we would need 1 (for

index) + 1 (for pointers) + $\left\lceil \frac{10}{3} \right\rceil$ = 6 I/O's.
In the worst case, we would need to read all of the index file to locate the key-value pair, and records would not be clustered (i.e. 1 record per block). Therefore, we would need 30 + 1 + 10 = 41 I/O's.
The average is then $\frac{47}{2}$ = 23.5 I/O's.
Without the buckets, the best case would be 5 I/O's and the worst case would be 310 I/O's, for the average of 157.5 I/O's.

Exercise 14.1.7

(a) The average number of words per document is $\frac{\sum_{n=1}^{10000} \frac{100000}{\sqrt{n}}}{1000} \approx 20000$

(b) In the worst case, each word will appear in each document at least once, therefore, the maximum number of blocks we would need is: $\frac{10000 \cdot 1000}{50}$ (for the bucket pointers) + $\frac{10000}{10}$ (for the word-pointer pairs) = 201000.

(c) There are total of 20000000 word occurrences (from (a)), so we would need $\frac{20000000}{50}$ (for the bucket pointers) + $\frac{10000}{10}$ (for the word-pointer pairs) = 401000.

(d) The worst case is still the same (each word appears in each document). Thus, we get $\frac{(10000 - 400) \cdot 1000}{50}$ (for the bucket pointers) + $\frac{10000 - 400}{10}$ (for the word-pointer pairs) = 192960.

(e) There are total of 16000000 word occurrences (from (a) minus the 400 most common), so we would need $\frac{16000000}{50}$ (for the bucket pointers) + $\frac{10000}{10}$ (for the word-pointer pairs) = 321000.

Exercise 14.1.8

(a) We could first construct a relation C containing the bucket entries for "cat", and relation D, containing the bucket entries for "dog". Then we would

join C and D picking only the document pointers such that D.type = C.type and D.document = C.document and D.position and C.position are within 5 of one another. Then use resulting document pointers to retrieve the documents.

(b) Similar to (a) except the join predicate for the position field would be D.position = C.position + 2, and perhaps removing the D.type = C.type join predicate if we want to be insensitive to the type boundaries.

(c) Again, similar to (a), but could retrieve buckets with type = 'title' only. Then join C and D using C.document = D.document.

Section 14.2

Exercise 14.2.1

(a) We would need $\frac{1000000}{10} = 100000$ blocks for the data + $\frac{1000000}{69} = 14493$ blocks for the leaf nodes + $\frac{14493}{70} = 208$ blocks for the next B-tree level, $\frac{208}{70} = 3$ for the next level, and one block for the root. The total would be 114705 blocks. We would need 5 I/O's (4 for the B-tree levels + data page).

(b) Same as (a)

(c) We would need $\frac{1000000}{10} = 100000$ blocks for the data + $\frac{100000}{69} = 1450$ blocks for the leaf nodes + $\frac{1450}{70} = 21$ blocks for the next B-tree level, , and one block for the root. The total would be 101472 blocks. We would need 4 I/O's (3 for the B-tree levels + data page).

(d) We would need $\frac{1000000}{7} = 142858$ blocks for the leaf nodes + $\frac{142858}{70} =$ 2041 blocks for the next level + $\frac{2041}{70} = 30$ blocks for the next level, , and one block for the root. The total would be 144930 blocks. We would need 4 I/O's (4 for the B-tree levels).

(e) We would need $\frac{1000000}{15} = 66667$ blocks for the primary data + 66667 for the overflow blocks, $\frac{66667}{69} = 967$ blocks for the leaf nodes + $\frac{967}{70} = 14$ blocks for the next B-tree level, and one block for the root. The total would be 134316 blocks. We would need 3 I/O's for the B-tree levels + average of $1 + 1 \cdot \frac{1}{3}$ for the total of $4\frac{1}{3}$.

Exercise 14.2.2

Note that the number of blocks does not change. We only need to calculate the average number of disk I/O's.

(a) We would need 4 I/O's to get to the first leaf block. To match 1000 records we need to examine $\frac{1000}{69} = 15$ leaf blocks. Since the file is sorted on the search key, we only need to read 100 blocks of data, therefore we need 4 + 15 + 100 = 119 I/O's.

(b) We would need 4 I/O's to get to the first leaf block. As in (a) we need to have 15 I/O's for the leaf blocks. Since the file is not sorted, we must read potentially 1000 data blocks for the total of 1019 (it is possible to average 550 + 15 + 4 = 569).

(c) We would need 3 I/O's to get to the first leaf block. Since the file is sorted on the search key, to match 1000 records, we only need to read 100 blocks, therefore we need one more for the next leaf block and 100 I/O's for the data blocks. The total is 104.

(d) We would need 3 I/O's to get to the first leaf block. Since the file is sorted on the search key, to match 1000 records, we only need to read 143 blocks. The total is 146.

Exercise 14.2.3

$12n + 4(n + 1) \leqslant 16384$, so 1023 keys and 1024 pointers.

Exercise 14.2.4

(a) 5 keys and 6 pointers for the interior nodes, 5 keys and 5 pointers in the leaf nodes

(b) 5 keys and 6 pointers for the interior nodes, 6 keys and 6 pointers in the leaf nodes

Exercise 14.2.5

(a) Start at the root. $41 \geqslant 13$ so follow the second pointer. $31 \leqslant 41 < 43$ so follow the third pointer. We find one of the keys is 41 so we follow the third pointer to the data block.

(b) Start at the root. $40 \geqslant 13$ so follow the second pointer. $31 \leqslant 40 < 43$ so follow the third pointer. We do not find any of the keys is 40, so we find that there are no records with key 40.

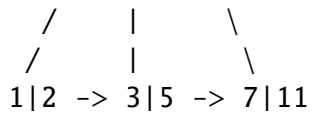
(c) Start at the root. $20 \geqslant 13$ so follow the second pointer. $20 < 23$ so follow the first pointer. No keys in the range are found so follow the next leaf pointer. 23 is in the range, so follow the first pointer to the data block. Key 29 is in the range so follow second pointer to the data block. No more keys on the leaf so follow next leaf pointer. Key 31 is not in the range so we are done.

(d) Start at the root. Follow leftmost pointers until the leaf node. For each of the keys, if it less than 30, follow the pointer to the data block. Thus we will follow data pointers for keys 2, 3, 5. Since $30 \geqslant 5$ follow the next leaf pointer and repeat. Thus we will follow the data pointers for keys 7, 11, then next leaf, 13, 17, 19, next, 23, 29, next. $30 < 31$ so we are done.

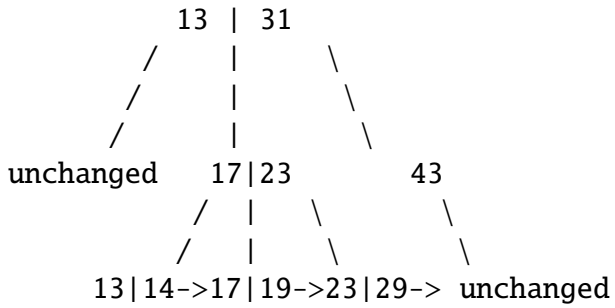
(e) Start at the root. $30 \geqslant 13$ so follow the second pointer. $23 \leqslant 30 < 31$ so follow the second pointer. We find none of the keys is greater than 30 so we follow the next leaf pointer. We find 31, which is the first key greater than 30 so we follow every pointer to data starting from 31 to the end of all keys in the leaves (i.e. we will follow data pointers for keys: 31,37,41,43,47).

(f) We search for key 1, this brings us to the leave with keys 2,3,5. There is no room to insert since we now have keys: 1,2,3,5. So we must split. We create new node and move the highest two keys: 3 and 5 to that node. We add a key 3 to the parent interior level to point to the newly created block. The result is:

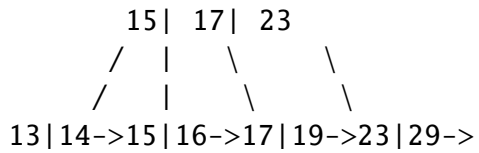
```
3   |   7
```



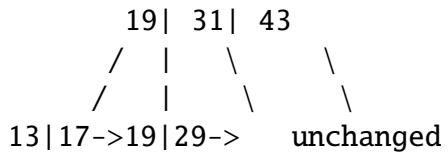
(g) We search for the key 14, this brings us to the leave with keys 13,17,19. There is no room for 14 so we split. We create new block and move keys 17 and 19 there. The parent interior node (keys 23,31,43) has no room so we must split. We create new node and move key 43 there and key 31 to the root. The result is:



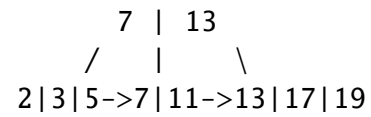
Next we search for the key 15. This brings us to the leave with keys 13,14. We insert 15. No other changes.
Next we search for the leave for key 16. This brings us to the leave with keys 13,14,15. We must split. We create new node and move keys 15,16 there. We add key 15 to the parent interior node. The result is:



(h) We search for the key 23. This brings us to the leave node with keys 23, 29. We delete key 23 and borrow the highest key (19) from the neighboring leave node. We also update the parent interior node to include key 19. The result is:



(i) As in (h) we locate the leave node with keys 23,29. We delete the node and all leave nodes that follow. We also update the parent interior node to remove the deleted keys. This results in an empty node (node that had keys 23,31,43). We merge the two adjacent nodes (node with key 7 and the empty node adding key 13). The result is:



Exercise 14.2.6

(a) Follow the same lookup procedure as for ”non-duplicate” B-tree. If the key is found, follow through the current (and subsequent leaves to the right) leave looking for more keys with the same value. Once a different key value is reached, stop.

(b) Follow the lookup logic from (a). Then follow the through the current leave (and subsequent leaves to the right) until either an empty spot is available (in which case we insert and are done) or a different key value is reached (in which case we follow the normal insert logic to insert the new key between the last key of the same value and the first key with a different value).

(c) Follow the lookup logic from (a). Then delete all keys that have the search value, following the normal delete process.

Exercise 14.2.7

When borrowing from the left adjacent non-sibling node, no additional logic in the existing algorithm is required. When borrowing from the right adjacent non-sibling node, we must update the root to replace the borrowed key value with the next smallest key value after the borrowed key value. In addition, because the now

unoccupied entry where the borrowed key was residing will not be claimed (since all searches for smaller keys will go to the left side of the root), we should move the keys left (either all keys in the node or only the now the smallest key of the right hand side of the root).

Exercise 14.2.8

Interior nodes must have at least 2 pointers and at least 1 key and leaf nodes must have at least 2 pointers and at least 2 keys (the max is 3 keys).

(a) The only possible layouts for the leaf nodes are 2-2-2 and 3-3. For both cases the next level must be the root. So there are two different B-trees.

(b) The only possible layouts for the leaf nodes are: 2-2-2-2-2, 2-2-3-3, 2-3-2-3, 2-3-3-2, 3-2-2-3, 3-2-3-2, 3-3-2-2. For the case with 4 leaves, we can only have either root only (with 4 pointers), or two interior nodes (with 2 pointers) and the root. For the 2-2-2-2-2 case, we must have the two interior nodes and the root, but there are two possible cases for the interior node, 2-3 or 3-2. Therefore the total number of B-trees is 14.

(c) The maximum number of leaf nodes we could have is 7 where one node must have 3 keys and the rest of the nodes must have 2 keys (e.g. 2-2-2-2-2-2-3). There are ${}_7C_1 = 7$ combinations. For each of these combinations we can have 5 possible layouts for the interior nodes: 2-2-3, 2-3-2, 3-2-2, 4-3, 3-4. Thus the total for this case is $7 \cdot 5 = 35$.
The next smaller number of leaf nodes is 6 where three nodes must have 3 keys and three nodes must have 2 keys (e.g. 2-2-2-3-3-3). There are ${}_6C_3 = 20$ combinations. For each of those, there are 4 possible layouts for the interior nodes: 2-2-2, 3-3, 2-4, 4-2. The total is $20 \cdot 4 = 80$
The next smaller number of leaf nodes is 5, where all nodes must have 3 keys. And there are only 2 possible layouts for the interior nodes: 2-3, 3-2. The total is 2.
The grand total is, therefore, 117.

Exercise 14.2.9

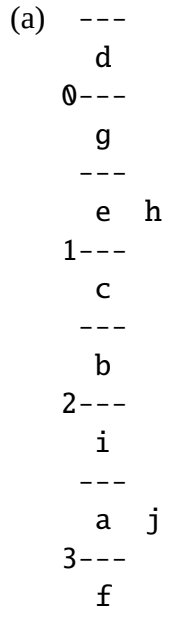
When we split the leaf blocks we divide pointers 2 and 2, so at the time of 4^{th} level added we will have only pairs of keys at each leaf. Similarly, because we split 3 and 2 for interior node, we will have all interior nodes with 3 pointers except for

the rightmost interior block which will have 2 pointers. Thus, we will have a root, then level with 3-2 then level with 3-3-3-3-2, for the total of 14 pointers to the leaf nodes, so the total number of keys in the leaves is 28. Therefore, a 4^{th} level is introduced when 28^{th} key is inserted.

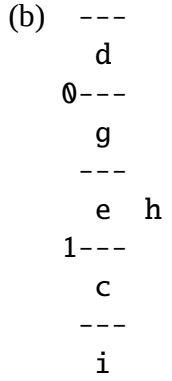
Section 14.3

Exercise 14.3.1

(a)



(b)

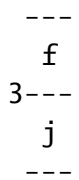


```
    2---

     ---
      f
    3---
      j
     ---

(c)  ---
      d  k
    0---
      g
     ---
      e  h  l
    1---
      c
     ---
      i
    2---
      m
     ---
      f  n
    3---
      j
     ---

(d)  ---
      g
    0---
      k
     ---
      e  l
    1---
      h
     ---
      i
    2---
      m
```



Exercise 14.3.2

For deletion, we would locate the record by applying hash function and then searching within the bucket (following the overflow linked list if needed). If deleting from the overflow list we simply remove the entry from the linked list, connecting the previous to next. If we are deleting from the bucket itself, we would first move up the entries below the deleted entry and then move the first entry from the overflow list (if any) into the empty spot.
The disadvantage of this approach is that it takes longer time to delete a record, but the advantage is that potentially less blocks could be used for the records and also the search would be much faster since we would maintain the sorted order (assuming insert also works by inserting in the sorted order).

Exercise 14.3.3

(a) For lookup and deletion we simply need to continue looking for other records with the same key in the current bucket. No changes for insert.
One of the potential issues is that if there are many duplicates, there will be many unused buckets and the buckets that are used would have long overflow lists.

(b) For lookup and deletion we not only need to continue looking for other records with the same key in the current bucket, but also check other buckets that match the key value based on the smaller number of bits. For instance, if current j is 3, and we are trying to locate the records with key 0000, we need to search on 000, then also on 00 and then also on 0.
For insert, when we split, for the $j < i$ case, if key is a duplicate, instead of splitting again, we use the second block to save the key but do not increment the ”nub” value.
One of the potential issues is that if there are many duplicates we will have frequent splits. Also searching becomes tedious since we need to read more than one block and follow several buckets.

(c) same as (a).

Exercise 14.3.4

(a) Since each integer can be represented as $10a + b$ where $0 \leqslant b < 10$ and so its square modulo 10 is the same as b^2 modulo 10. The squares of 0 to 9 modulo 10 are: 0, 1, 4, 9, 6, 5, 6, 9, 4, 1, and so the buckets for 2,3,7,8 will always be empty and buckets 1,4,6 are twice as likely to be hit.

(b) The problem here is that every number divisible by 4 will go to bucket 0, which is about 25% of all numbers.

(c) There does not appear to be any good values for B to make this hash function useful.

Exercise 14.3.5

In order for all members of the block to go to the same created block, they must have the same bit at the $(j+1)^{st}$ position. The probability of all n+1 records having the same bit is $\left(\frac{1}{2}\right)^{n+1}$.

Exercise 14.3.6

(a)

```
      ---
      0000
000
      0001
      ---
      0010
001
      0011
      ---
      0100
010
      0101
      ---
```

```
        0110
    011
        0111
        ---
        1000
    100
        1001
        ---
        1010
    101
        1011
        ---
        1100
    110
        1101
        ---
        1110
    111
        1111
        ---
```

(b) `i = 3, n = 6, r = 16`

```
        ---
        0000
    000 1000

        ---
        0001
    001 1001

        ---
        0010  1110
    010 0110
        1010
        ---
        0011  1111
    011 0111
        1011
```

```
        ---
        0100
   100  1100

        ---
        0101
   101  1101

        ---
```

(c)
```
        ---
        0001
   000
        0000
        ---
        0011
   001
        0010
        ---
        0101
   010
        0100
        ---
        0111
   011
        0110
        ---
        1001
   100
        1000
        ---
        1011
   101
        1010
        ---
        1101
   110
        1100
```

```
        ---
        1111
   111
        1110
        ---
```

(d) `i = 3, n = 8, r = 16`

```
        ---
        1000
   000 0000

        ---
        1001
   001 0001

        ---
        1010
   010 0010

        ---
        1011
   011 0011

        ---
        1100
   100 0100

        ---
        1101
   101 0101

        ---
        1110
   110 0110

        ---
        1111
   111 0111
```

Exercise 14.3.7

We could use double indirection by having hash table point to pointers to records rather than the records themselves. Thus, we would be moving pointers and not the records.
We could leave forwarding addresses in buckets when we split them.

Exercise 14.3.8

(a) The number of buckets with twice as many keys would be $n - 2^{\lfloor log_2 n \rfloor}$ and so the number of overflow blocks would be $\frac{cnk - nk}{n + 2^{\lfloor log_2 n \rfloor}}$. Total number of blocks would be $n + \frac{cnk - nk}{n + 2^{\lfloor log_2 n \rfloor}}$.

(b) The λ in this case is ck (i.e. $p(i) = \frac{e^{-ck} ck^i}{i!}$), and the number of overflow blocks per bucket is not known. Assuming we have x overflow blocks per bucket, the expected number of blocks would be

$$n + n \sum_{x=1}^{cn} x \cdot \sum_{y=1}^{y=k} \frac{e^{-ck} ck^{xk+y}}{(xk+y)!}$$

Exercise 14.3.9

In the best case, all buckets will have a number of records that is divisible by 100. Then, the number of blocks needed is 10000.
In the worst case, we could have only one record in each of 999 buckets. Since we are not sharing blocks among buckets, we would need 999 blocks for these buckets. The remaining bucket has 999001 records, which requires 9991 blocks, for a total of 10,990 blocks.

Section 14.5

Exercise 14.5.1

(a) *hd* dimension: 210, *speed* dimension: 1.9, 2.15, 2.7, 3.0

(b) *hd* dimension: 210, *speed* dimension: 2.15, 2.7, 3.0

(c) The function result would be a 2-bit value with first bit being zero when *hd* value is less than 210 and one otherwise, and the second bit being zero when *speed* is less than 2.7 and one otherwise.

Exercise 14.5.2

hd dimension: partition among grids 210 and 260, *speed* dimension: partition among 2.15 and 2.7, *ram* dimension: partition among 1000 and 2000. This will divide the data equally with one point per bucket.

Exercise 14.5.3

We could use the same function as in 14.5.1 (c), adding a third bit to the result. the third bit would be zero for *ram* values less than 1024 and would be one otherwise.

Exercise 14.5.4

A line in either dimension is equally good since it will not split any other buckets. For instance, we pick a line along *speed* 2.6.

Exercise 14.5.5

(a) 25

(b) The distance is 15.8, so the other buckets that need to be examined are: (80,200), (120,200), (80,150), (100,150), (120,150).

Exercise 14.5.6

(a) $p2^m + (1-p)2^{n-m}$

(b) $m = \frac{1}{2} log_2 \frac{(p-1)2^n}{p}$ (or $\frac{1}{2}n + log_2 \sqrt{\frac{p-1}{p}}$)

Section 14.6

Exercise 14.6.1

(a)
```
1.42 --> 512
1.86 --> 2048
2.00 --> 1024
2.10 --> 512
2.20 --> 1024
         2048
2.66 --> 1024
2.80 --> 1024
         2048
3.20 --> 512
         1024
```

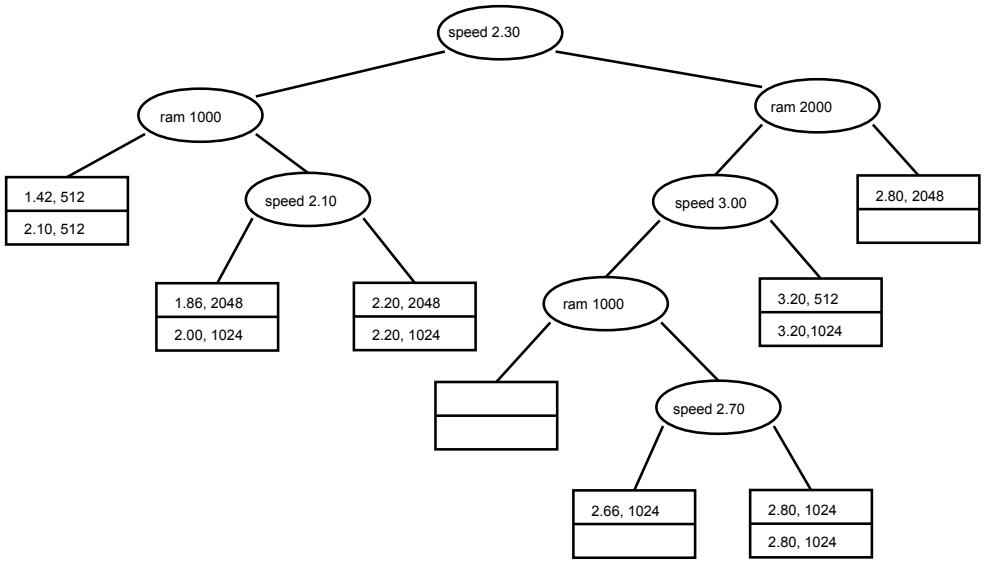
(b)
```
512  --> 80
         250
1024 --> 160
         200
         250
         320
2048 --> 160
         250
         300
```

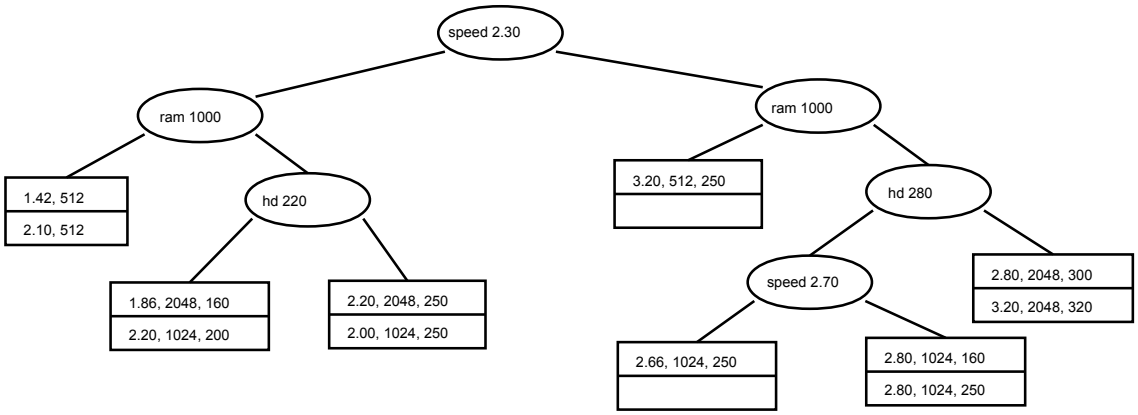
(c)
```
1.42 --> 512  --> 80
1.86 --> 2048 --> 160
2.00 --> 1024 --> 250
2.10 --> 512  --> 250
2.20 --> 1024 --> 200
         2048 --> 250
2.66 --> 1024 --> 250
2.80 --> 1024 --> 160
                  250
```

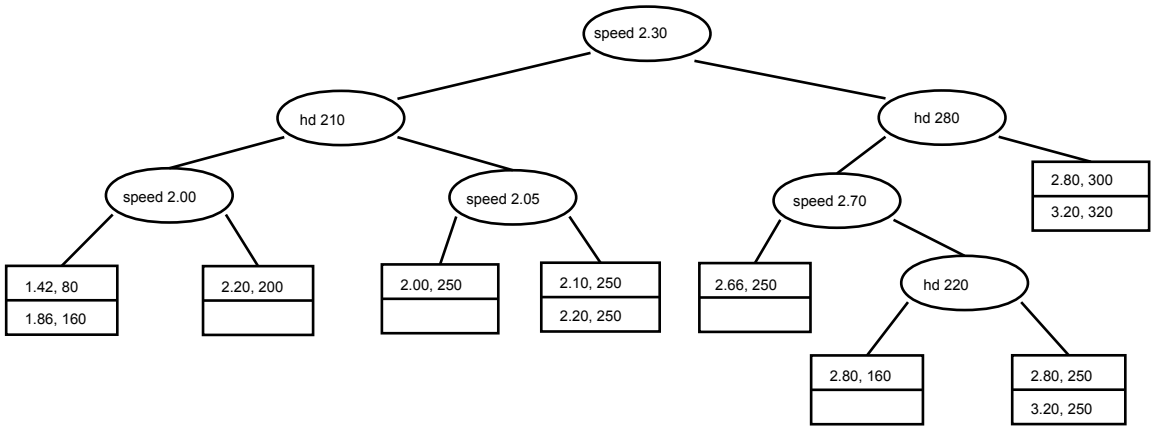
```
          2048 --> 300
3.20 --> 512  --> 250
          1024 --> 320
```

Exercise 14.6.2

Diagrams for (a), (b), and (c) follow:


speed 2.30
ram 1000
ram 2000
1.42, 512
2.10, 512
speed 2.10
speed 3.00
2.80, 2048
1.86, 2048
2.00, 1024
2.20, 2048
2.20, 1024
ram 1000
3.20, 512
3.20,1024
speed 2.70
2.66, 1024
2.80, 1024
2.80, 1024


speed 2.30
ram 1000
ram 1000
1.42, 512
2.10, 512
hd 220
3.20, 512, 250
hd 280
1.86, 2048, 160
2.20, 1024, 200
2.20, 2048, 250
2.00, 1024, 250
speed 2.70
2.80, 2048, 300
3.20, 2048, 320
2.66, 1024, 250
2.80, 1024, 160
2.80, 1024, 250


speed 2.30
hd 210
hd 280
speed 2.00
speed 2.05
speed 2.70
2.80, 300
3.20, 320
1.42, 80
1.86, 160
2.20, 200
2.00, 250
2.10, 250
2.20, 250
2.66, 250
hd 220
2.80, 160
2.80, 250
3.20, 250

Exercise 14.6.3

(a) For the dense index on x we would need 100/10 = 10 blocks, and so for the sparse index of x we would need 1 block. Therefore, we need two disk I/O's to get to the y index. For each x there are 100 y values and so the dense index on y will need 10 blocks, and the sparse index will need one block. We would need two disk I/O's to get to the y value. The total number of I/O's is then 2+2+1(for the data) = 5.

(b) For the dense index on y we would need 1000/10 = 100 blocks, and so for the sparse index on y we would need two levels (10 blocks and 1 block). Therefore, we need 3 disk I/O's to get to the x index. For each y there are 10 values of x and so we just need a one block dense index for x. The total disk I/O's is then 3+1+1 (for the data) = 5.

(c) We would want to buffer the top 11 blocks of the tree (root + 10 intermediate blocks of the next level after the root). This means that picking x as the first index is better since the whole index would be in memory and the queries where predicate x = C is false could be answered without any additional I/O's.

Exercise 14.6.4

To evaluate $20 \leqslant x \leqslant 35$ we need to read the root, then 3 blocks for the range (11-20, 21-30, 31-40). That's 4 I/O's. For each of the x values qualified we need to evaluate $200 \leqslant y \leqslant 350$ which means reading the root block and then 3 blocks for the range (101-200, 201-300, 301-400). The total is then, $4+1\cdot4+10\cdot4+10\cdot4 = 88$.

Exercise 14.6.5

(a) $150 \leqslant salary < 300$ and $age < 47$

(b) $80 \leqslant salary < 150$ and $age < 60$

Exercise 14.6.6

The changes would be: 1) right child of the "Salary 80" would be a new node "Age 50" with left child pointing to a block with data (20,110) and the right child pointing to a block with data (50,100) and (50,120), and 2) right child of "Age

35” would point to a new node ”Salary 400” with left child pointing to a block with data (45,350) and the right child pointing to the block with the data (35,500) and (40, 400).

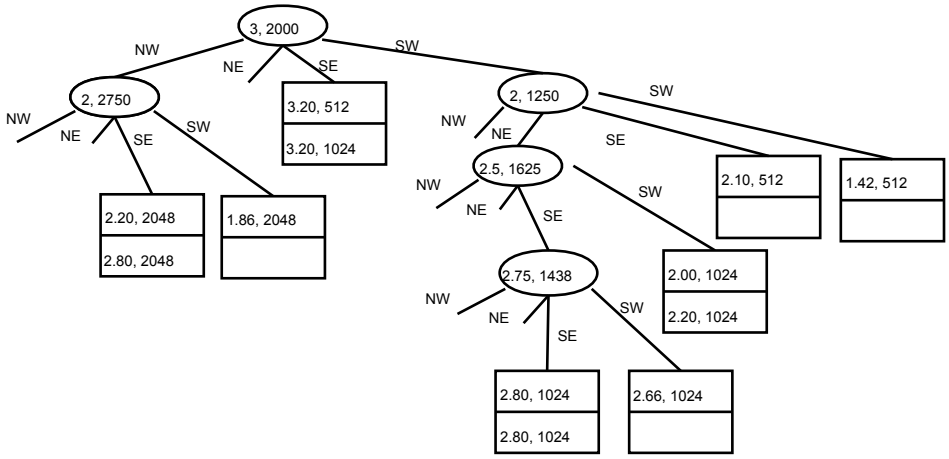
Exercise 14.6.7

(a) Since the tree is balanced, $n = 2^k$, where k is some natural number representing the number of levels in the tree (not counting the root)(really $log_2 n$). And since we reduce the number of leafs that we consider by a factor of 2 for each attribute that has the value specified and this happens every other level, we will examine $\frac{2^k}{2^{\frac{k}{2}}} = 2^{\frac{k}{2}} = \sqrt{n}$.

(b) This is a generalization of (a) where we will reduce the number of leafs we consider by a factor of 2 every m out of d times. Thus, we will examine $\sqrt[d]{n^{d-m}}$ leafs.

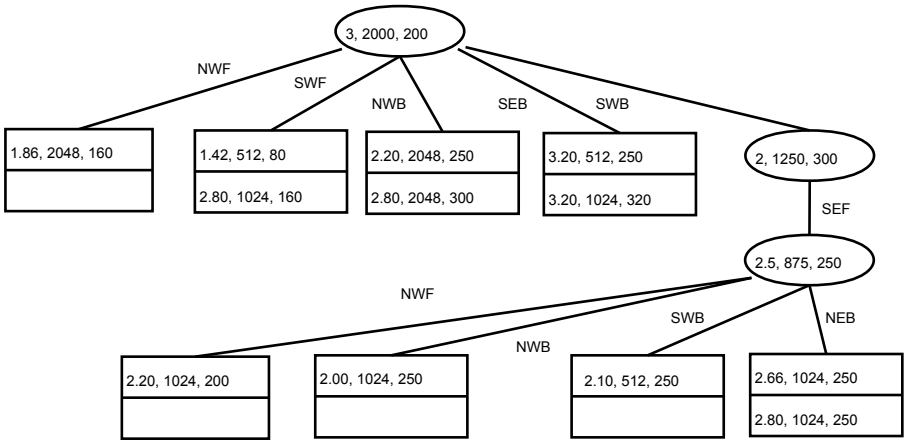
(c) For partitioned hash table, we will only need to examine 2^x of the buckets, where x is the number of bits used for all d dimensions minus the number of bits used in m dimensions that are for the specified attributes. Note that that compared to (b), which is tied to the total number of leaves n (and thus, to the number of records), the hash table is tied to the number of bits for the keys. Thus, we cannot make a strong arithmetic comparison between (b) and (c) without making some further assumptions. For instance, if we assume we use only one bit per key for each of the d keys, then the comparison becomes a simple $log_2 n$ vs. d. Which puts the hash table in advantage when relation is large. In general, the comparison is $\frac{log_2 n}{d}$ for kd tree vs. x, where x is the average number of bits per keys for all $d - m$ attributes that are not specified in the query.

Exercise 14.6.8



Exercise 14.6.9

We use the compass designations for the quadrants and for the children of the nodes, as in the book. However, for the third dimension we use F (forward) and B (back) for points with z-coordinates less than and greater than the splitting line respectively. To save space, we do not show edges that have no child.


3, 2000, 200
NWF
SWF
NWB
SEB
SWB
1.86, 2048, 160
1.42, 512, 80
2.80, 1024, 160
2.20, 2048, 250
2.80, 2048, 300
3.20, 512, 250
3.20, 1024, 320
2, 1250, 300
SEF
2.5, 875, 250
NWF
SWB
NEB
NWB
2.20, 1024, 200
2.00, 1024, 250
2.10, 512, 250
2.66, 1024, 250
2.80, 1024, 250

Exercise 14.6.10

We can not do it. For instance, when all the points are in a line, there will always be empty squares.

Exercise 14.6.11

The R-tree would have a root and three levels. Since the overlays do not affect the number of nodes that need to be examined on a given level, the number of blocks that need to be read is 3.

Section 14.7

Exercise 14.7.1

All keys are in ascending order.

(a) uncompressed:

```
001000000000
000000000010
010000000000
000000001000
000000110000
100000000000
000100000101
000011000000
```

compressed:

```
1010
11101010
01
11101000
11011000
```

```
00
101111010101
11010000
```

(b) uncompressed:

```
011010000000
100101101001
000000010110
```

compressed:

```
010001
0010100100011010
1101110100
```

(c) uncompressed:

```
001000000000
000000000011
000000100000
110110011000
000000000100
000001000000
```

compressed:

```
1010
1110101000
110110
00000100101000
11101001
110101
```

Exercise 14.7.2

```
100000001000 OR 000000010000 = 100000011000
110000000000 OR 001000000000 OR 000100000000 = 111100000000
100000011000 AND 111100000000 = 100000000000
```

So, only one record found (25,60).

Exercise 14.7.3

(a) $125000m$

(b) There would be a variable run of zeros from 0 to m-1, followed by the $\frac{1000000}{m} - 1$ runs of m-1 zeros.
The total bytes would be $\frac{1}{8}\left(2mlog_2\frac{m-1}{2} + (\frac{1000000}{m} - 1)(2 + 2log_2(m-1))\right)$.

Exercise 14.7.4

We could encode the length in m bits followed by m zeros. For instance, using 20 bits (10 bits plus 10 zeros), we can encode length of up to 1024.

Exercise 14.7.5

(a) 0100110111110101

(b) 0011010111011010100001

(c) 101111101010110101