Midterm
Spring 2018
Example Solutions

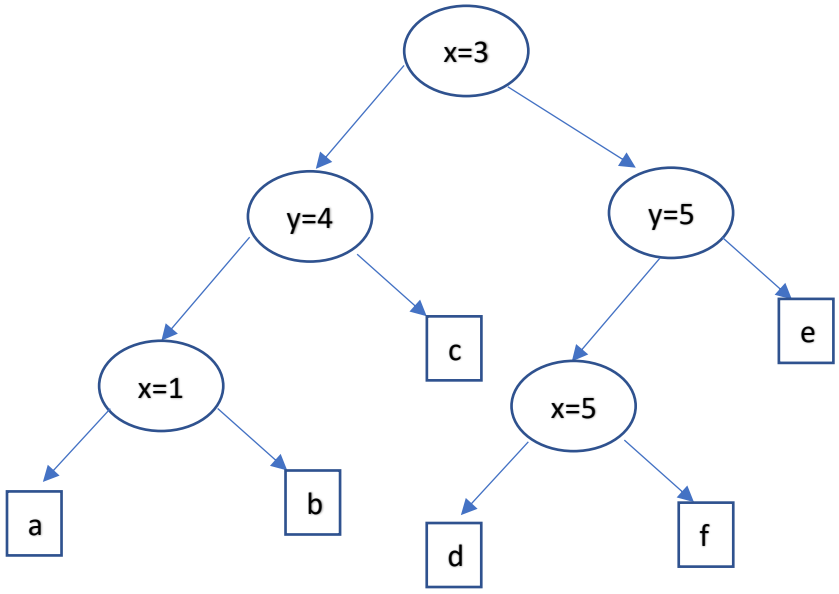
Problem 1

1. We have k=3, se we need to split the space into 8x8 cells. The point (15, 65) is located in the (1, 6) cell. So, the Z-value for this cell is: (001, 110) => 010110 = 22

2. We have to find all the cells that intersect the query. It is easy to see that the query intersects the following cells and we compute their z-avlues:
   (0,0) = (000, 000) => 000000 = 0
   (1,0) = (001, 000) => 000010 = 2
   (0,1) = (000, 001) => 000001 = 1
   (1,1) = (001, 001) => 000011 = 3
   (0,2) = (000, 010) => 000100 = 4
   (1,2) = (001, 010) => 000110 = 6

   So, the 2D range query is mapped to two ranges, one is [0, 4] and the other [6,6].

3. We need to create the kd-tree for this dataset. We start with the x-axis and we find the median value. You can create slightly different trees depending if you put the point before or after the line and above or below. We chose here to put the point before or below the line. So, the first line is x=3 and on the left we have a, b,c and on the right we have d, e, f. We split recursively left and right set, now using the y dimension and so on.



For the NN query Q=(5,4) you need to visit nodes x=3, y=5, x=5, node d and node f. The answer is d.
Notice that after you find d, you still need to check the right child of x=5 because there may be a point there that can be closer to the query than d. So, we read f and we find that d is still the NN.

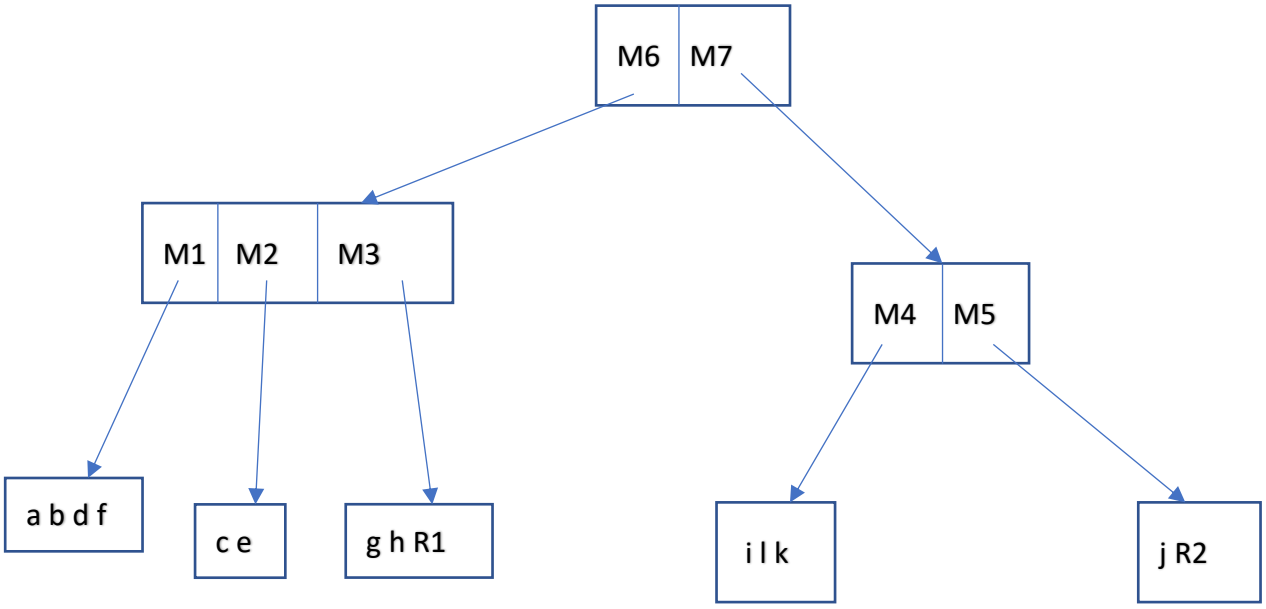
4) We will use the Hilbert curve since it has better clustering properties than Z-curve.

The idea is to store together with each MBR the H-value (Hilbert value) of its center. Then, when we need to insert a new MBR, we compute its H-value and we insert it in a way that is similar to B+-tree, by using only its H-value and the H-values of the MBRs. Then, after we insert it into the leaf node, we update the MBRs of its parents similar to the traditional R-tree.

The node splitting algorithm is also simple. We order the MBRS using their H-value and we split using these values (put the first half entries in one node and the rest in the second node).

Problem 2

1) There are many ways to create a valid R-tree on the dataset. Next is one example. The important thing is to make sure that the R-tree that you create is valid.



Where the MBRs for Mi are:
M1 [(1,3), (3,5)], M2[(1,7)(2,8)], M3[(3,1),(5,3)], M4[(6,0), (7,1)], M5(6,5),(8,7)], M6[(1,1), (5,8)], M7[(6,0), (8,8)]

2) Using the above R-tree, we can answer the queries as follows:
For range query: we start from the root and we visit the nodes that intersect the query and retrieve the objects that intersect the query. For Q, we visit: Root, M6, M7, M3, M5 and the answer is R2 and R1 (if you assume that when a query "touches" an MBR there is no intersection, then the answer is only R2, which is also correct under your assumption).

For NNQ, we use the Best First NN algorithm. We visit: Root, M7, M6, M3, and the answer is R1.

3) We use the algorithm from the TP NN query. First, to find the current NN we have to visit: Root, M6, M3 and the answer is g. So, the current NN is g.

To answer the TP NN query, we have to compute the influence time of the MBRs and objects that we visit. The influence time of Root, M6 and M3 are 0 and we visit them first.
The influence time of M2 is the next smaller and we visit it next. The M2 contains a that has the smallest influence time equal to 1.

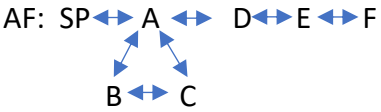
So, the answer is: R={g}, T={1}, C={a} and we had to visit: Root, M6, M3, and M2.

Problem 3

1) We use the insertion algorithm of the Snapshot Index and we get:

A [1,*)	B [5,15)	C [9,24)	D [15,*)	E [21,*)
<1,[1,*)> <2,[2,*)> <3,[3,*)> <4,[4,*)>	<5,[5,13)> <6,[6,15)> <7,[7,14)> <8,[8,15)>	<9,[9,24)> <10,[10,18)> <11, [11, 17)> <12,[12,24)>	<6,[15,*)> <16,[16,*)> <19,[19, *)> <20, [20,*)>	<21,[21,*)> <22,[22,*)> <23, [23,*)> <9, [24,*)>

F [25,*)
<25,[25,*)>



AT: (1,A), (5,B), (C,9), (D, 15), (E, 21),(F,25)

To answer the query at 21, we need to visit: E, D, A, C, B, SP

2) For the All Version query, we need to do some modifications to the Snapshot Index. This idea is discussed in the Snapshot Index paper:
"With a minor modification, the different versions of a given key can be linked together so that queries of the form: "find all versions of a given key" can be addressed in $O(\log_b n + v)$ I/Os, where v represents the number of such versions. Each object record is augmented with one pointer. When an object is deleted, its aid is not deleted from the hashing function, rather it is marked as deleted. When a new version of this object is born, the hashing function can locate the previous version, thus it is possible to have a pointer from the record of the new version to the record of the previous version. When the first copy of an object is created due to the copying procedure, its record will point to the first record of the current version. Later copies of this version can thus point directly to the first record of their version. Thus, the above query can avoid searching all copies of versions. "

3) After these operations we have:

R1

<3,[10,*), A>
<15,[15,22),B>
<15,[22, *), D>
<21,[22,23),E>
<21,[23,*),F>
<37,[15,23),C>

A

<3,[10,*)>
<4,[4,*)>
<6,[5,*)>

B

<15,[2,*)>
<19,[6,*)>
<21,[9,*)>
<25,[7,18)>
<28,[21,*)>
<30,[20,*)>

C

<37,[8,23)>
<38,[11,16)>
<49,[12,*)>

D

<15,[2,*)>
<16,[22,*)>
<19,[6,*)>

E

<21,[9,*)>
<28,[21,*)>
<30,[20,*)>

F

<21,[9,*)>
<28,[21,*)>
<30,[20,*)>
<49,[12,*)>

Problem 4

1) The distance is DTW(X,Y) = 9 and the matrix of the DTW(X,Y) is:

5	6	8	10	7	9
3	4	6	6	7	11
2	4	4	4	7	11
2	3	3	3	6	11
1	2	2	2	7	13
X / Y	3	1	1	6	7

2) The Euclidean distance is more efficient to compute (O(n)) and easier to index.
DTW is more expressive and can handle better accelerations/decelerations over time and time series with different lengths. However, it takes $O(n^2)$ to compute in general.

Also, Euclidean distance is a **special case** of DTW when we use the main diagonal to compute the distance (or we impose warping window length or constraint δ=0) . (see the example above or X and Y).