Exercise 4

for the lecture

Computational Geometry

Dominik Bendle, Stefan Fritsch, Marcel Rogge and Matthias Tschöpe

December 20, 2018

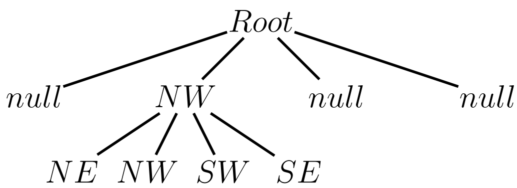
Exercise 1 (Image Compression using Quadtrees) (1 + 3 + 1 + 1 points)

a)

Best Case: The data are clustered in the same region/subtree. ⇒ The tree is filled up subtree by subtree.

Example for the worst case (image filled up with 25 % of data points):

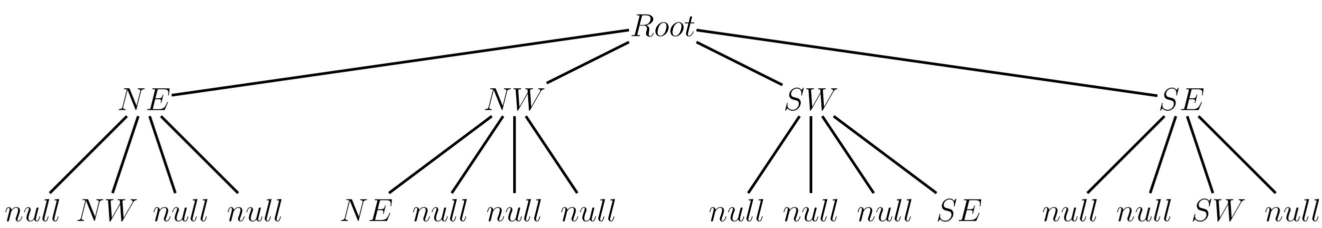
x_1	x_2		
x_3	x_4		



Worst Case: The data points are distributed all over the image and not clustered. ⇒ The data are distributed evenly among the subtrees.

Example for the worst case (image filled up with 25 % of data points):

	x_1	x_2	
	x_3	x_4	



b)
We assume in Worst Case that our squared input image is filled up with at least 25 % of data points. So our tree is a complete 4−regular tree. Since, e is the edge length of the squared image, the depth of our Quad-Tree in the Worst Case is given by $log_4(e^2) = 2 \cdot log_4(e)$. Now, we can compute the number of inner vertices as:

$$\sum_{\ell=0}^{log_4(e^2)-1} 4^\ell \tag{1}$$

Further, we know the number of leafs is:

$$4^{log_4(e^2)} \tag{2}$$

Since, each inner vertex has 5 Pointer (four to its children and one to its parent vertex) except the root node (the root node does not have a parent, therefore we subtract at the end 8 Byte) and each pointer needs 8 Byte, so we need:

$$\left(\sum_{\ell=0}^{log_4(e^2)-1} 4^\ell\right) \cdot 5 \cdot 8 \text{ Byte} - 8 \text{ Byte} = \frac{1}{3}\left(e^2 - 1\right) \cdot 5 \cdot 8 \text{ Byte} - 8 \text{ Byte} \tag{3}$$

for the inner vertices. The memory for the leaf nodes is given by:

$$x \cdot 4^{log_4(e^2)} \cdot 8 \text{ Byte} = x \cdot e^2 \cdot 8 \text{ Byte} \tag{4}$$

where x is the ratio of non-zero data. This leads to the following algorithm:

Algorithm 1 Calculate $\frac{s_{new}}{s_{old}}$ ratio

---

f(e, x)

Input: edge length e and fraction of non-zero data x.

Ouput: ratio $\frac{s_{new}}{s_{old}}$.

1: $s_{old} \coloneqq e^2 \cdot 8\text{Byte}$
2: `size_not_null_data` $\coloneqq x \cdot s_{old}$
3: `tree_size` $\coloneqq \left(\frac{1}{3}(e^2 - 1) \cdot 5 + x \cdot e^2\right) \cdot 8 \text{ Byte} - 8 \text{ Byte}$
4: $s_{new} \coloneqq$ `size_not_null_data` $+$ `tree_size`
5: **return** $\frac{s_{new}}{s_{old}}$

---

c) First we explain the idea of the function g, which computes the storage for the best case. As above, each inner vertex has 5 Pointer (except the root node) and each pointer needs 8 Byte. Similar to the Worst Case, the tree has a maximal depth of $log_4(e^2)$. So, the last inner vertex is on level $log_4(e^2) - 1$. The number of used non-zero data is again $x \cdot e^2$. Hence, we get the following best case function:

Algorithm 2 Calculate $\frac{s_{new}}{s_{old}}$ ratio

g(e, x)

Input: edge length e and fraction of non-zero data x.

Ouput: ratio $\frac{s_{new}}{s_{old}}$.

$s_{old} \coloneqq e^2 \cdot 8\text{Byte}$
`size_not_null_data` $\coloneqq x \cdot s_{old}$
`tree_size` $\coloneqq ((log_4(e^2) - 1) \cdot 5 + x \cdot e^2) \cdot 8 \text{ Byte} - 8 \text{ Byte}$
$s_{new} \coloneqq$ `size_not_null_data` $+$ `tree_size`
return $\frac{s_{new}}{s_{old}}$

<u>Worst Case calculation:</u>

f$(e = 4, x = 0.25)$

$s_{old} \coloneqq 4^2 \cdot 8\text{Byte} = 128 \text{ Byte}$
`size_not_null_data` $\coloneqq 0.25 \cdot s_{old} = 32 \text{ Byte}$
`tree_size` $\coloneqq \left(\frac{1}{3}(4^2 - 1) \cdot 5 + 0.25 \cdot 4^2\right) \cdot 8 \text{ Byte} - 8 \text{ Byte} = 224 \text{ Byte}$
$s_{new} \coloneqq 32 + 224 = 256$
return $\frac{s_{new}}{s_{old}} = \frac{256}{128} = 2$

<u>Best Case calculation:</u>

g(e, x)

$s_{old} \coloneqq 4^2 \cdot 8\text{Byte} = 128 \text{ Byte}$
`size_not_null_data` $\coloneqq 0.25 \cdot s_{old} = 32 \text{ Byte}$
`tree_size` $\coloneqq ((log_4(4^2) - 1) \cdot 5 + 0.25 \cdot 4^2) \cdot 8 \text{ Byte} - 8 \text{ Byte} = 64 \text{ Byte}$
$s_{new} \coloneqq 32 + 64 = 96$
return $\frac{s_{new}}{s_{old}} = \frac{96}{128} = 0.75$

d) We have four approaches how we could save even more memory:
1) We assume, each string needs 8 Byte. Instead of using the strings "NE", "SE", "SW" and "NW", we can use integer values. Since, we only need four values, the standard int32 which can store $2^{32} - 1$ values is enough. If we assume a boolean value need exact one Bit, then we can do better. We could use two boolean values (2 Bit) which decode the values $1, 2, 3$ and 4 as binary value.
2) We can eliminate the pointers which point to the data points by saving the data points directly in the tree.
3) We skip the rule that every node has to have exactly four child nodes to reduce the memory for storing null-pointers. We can achieve this by saving the leafs of a parent node p_i in a list. This list only contains real data points and no null values.
4) Further, for the non-zero-entries, we could try to find duplicates and store them only once.

Exercise 2 (kD-tree Construction) **(2 points)**

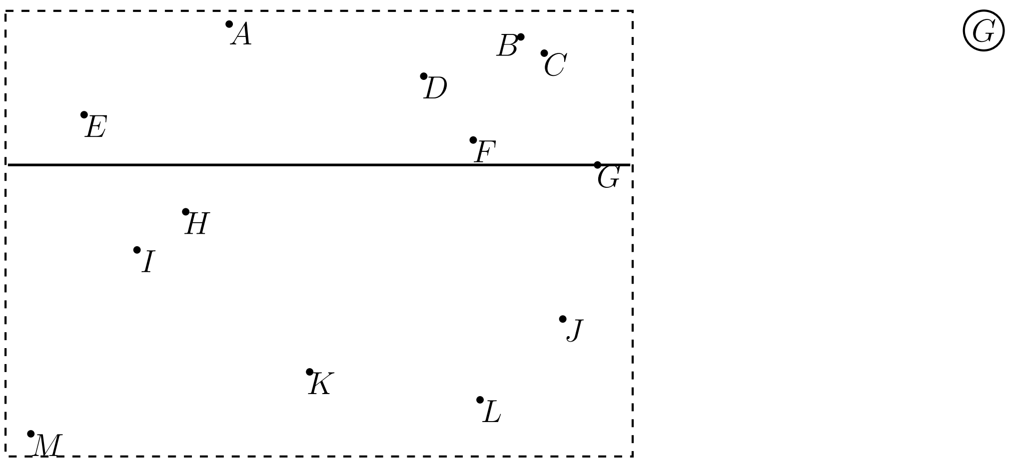
We use two lists x_{sorted} and y_{sorted} and sort them in the x respectively the y direction:

$$\begin{aligned} y_{sorted} &\coloneqq [A, B, C, D, E, F, G, H, I, J, K, L, M] \\ x_{sorted} &\coloneqq [M, E, I, H, A, K, D, F, L, B, C, J, G] \end{aligned} \tag{5}$$

Both list have a length of $len(y_{sorted}) = len(y_{sorted}) = 6$. We shorten the given rules by Ri, where $i \in \{1, 2, 3, 4\}$.

Step 1.

By $R2$ we first have to split in $y-$direction. This means we have to calculate the middle element of the y_{sorted} list, which is G. Hence, G is the first vertex in the $kD-$tree. This yields to the following results:



Step 2.

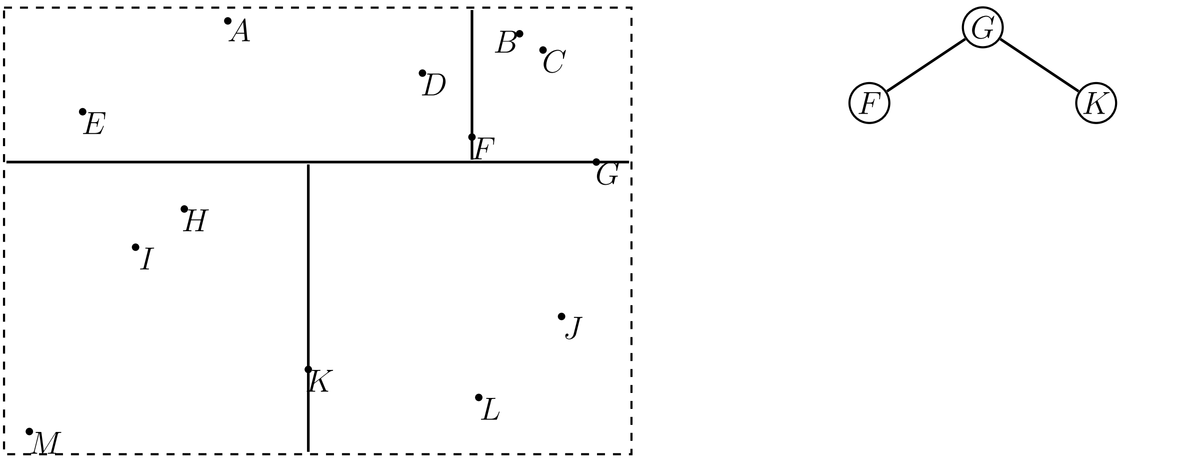
Now we have to split in $x-$direction. Therefore, we take the x_{sorted} list and split them in two new lists x_{above} and x_{below}, where both list are sorted in $x-$direction:

$$\begin{aligned} x_{above} &\coloneqq [E, A, D, F, B, C] \\ x_{below} &\coloneqq [M, I, H, K, J, G] \end{aligned} \tag{6}$$

Since, both lists have even length, we have to use $R4$ to calculate the middle elements:

$$\begin{aligned} n \coloneqq len(x_{above}) = 6 &\implies \text{middle element is } F \\ n \coloneqq len(x_{below}) = 6 &\implies \text{middle element is } K \end{aligned} \tag{7}$$

By $R3$, we have to order the points above, as the left child and the points below as the right child. This gives us:



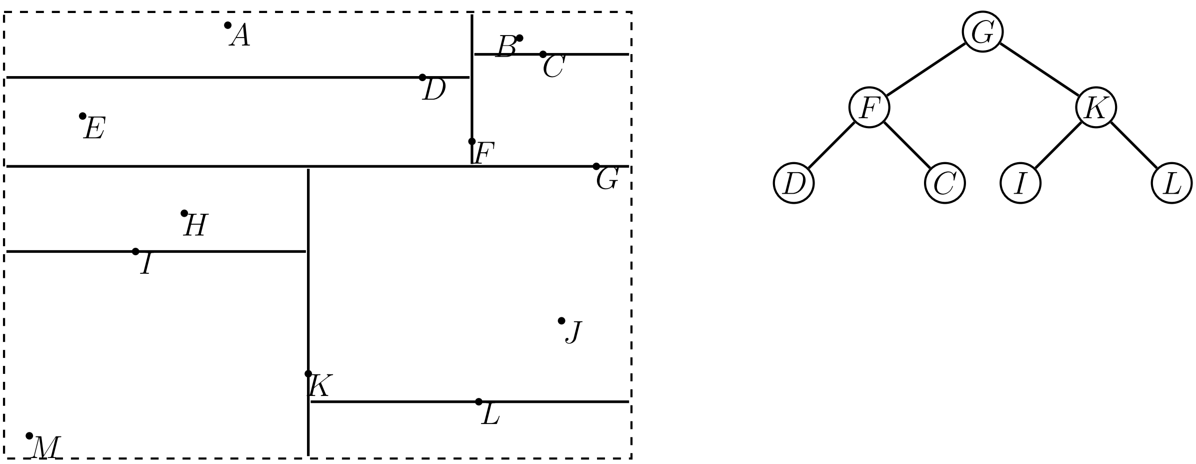
Step 3.
Now, we split in $y-$direction. Hence, we sort x_{above} and x_{below} in $y - direction$ and split each of them into two new lists y_{above_1}, y_{above_2} for the left and right upper part analogously for the part below. Then we get:

$$\begin{aligned} y_{above_1} &:= [A, D, E] \qquad & y_{above_2} &:= [B, C] \\ y_{below_1} &:= [H, I, M] \qquad & y_{below_2} &:= [J, L] \end{aligned} \tag{8}$$

For y_{above_2} and y_{below_2} we have to use $R4$ to calculate the middle elements. Let M be a function which calculates the middle elements for a list, by the given rules, then we get:

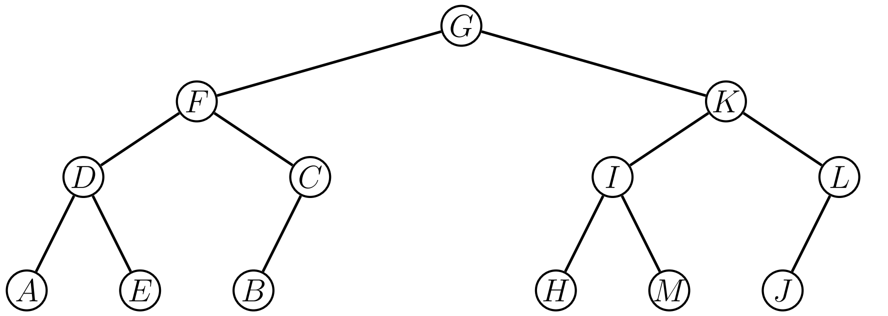
$$\begin{aligned} M(y_{above_1}) &= D \qquad & M(y_{above_2}) &= C \\ M(y_{below_1}) &= I \qquad & M(y_{below_2}) &= L \end{aligned} \tag{9}$$

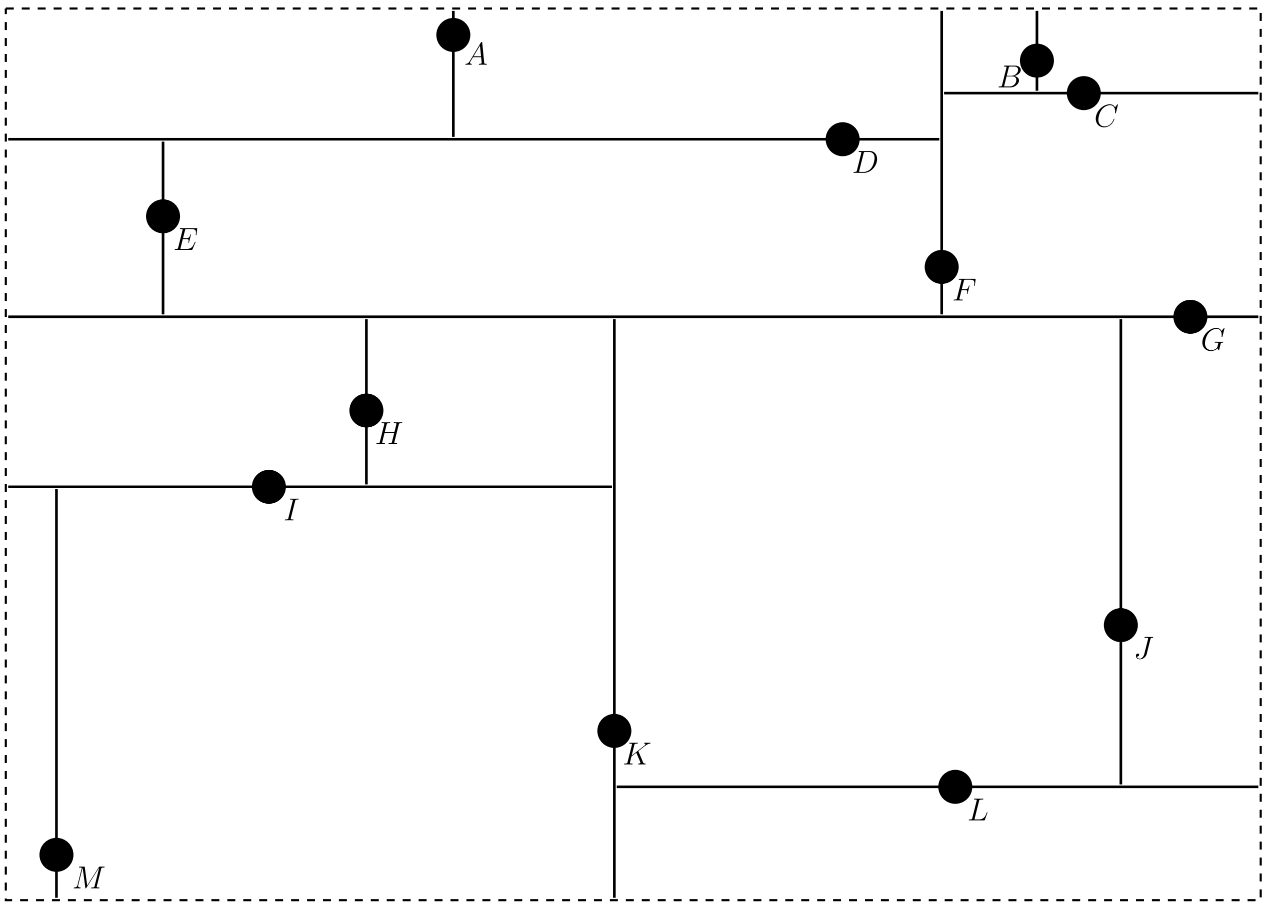
Updating the tree and the lines in the plane yields to:



Step 4.

Now, we are splitting in $x-$direction. Since each remaining list only contains at least one element, we are done after insert those elements in the $kD-$tree. Again, by using $R3$ for inserting the new child nodes, we get:





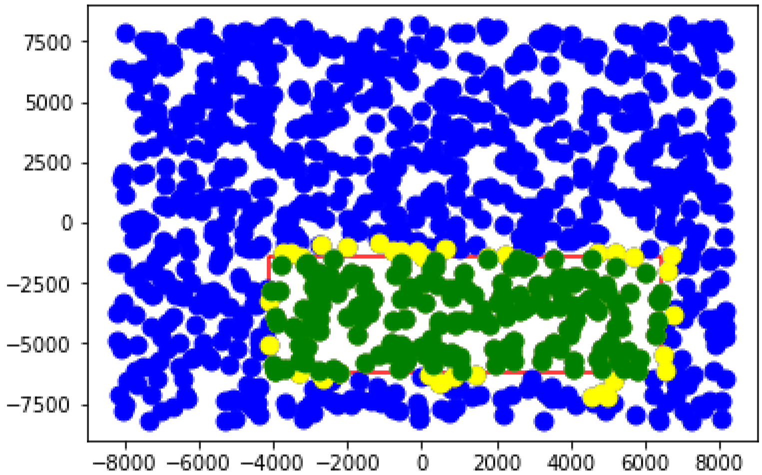
Exercise 3 (Range Search in a kD-tree) **(5 points)**

```
#classes for the KDTree structure class kdNode(object):
    def __init__(self):
        self.value = None
        self.left = None
        self.right = None

class kdLeaf(object):
    def __init__(self):
        self.value = None

k = 2 #given by task
#function for building a KDTree
def kdTree(points, level=0):
    dim = level % k #determines if points should be split by x or y axis
    axis = points[0].dtype.names[dim]
    median = (np.sort(points, order=axis)[len(points)-1][axis] +
              np.sort(points, order=axis)[0][axis]) / 2
    node = kdNode()
    node.value = median
    split_list = splitList(points, median, axis) #function that splits list into left and right
    left_points = split_list[0]
    if len(left_points) == 1:
        node.left = kdLeaf()
        node.left.value = left_points[0]
    else:
        node.left = kdTree(left_points, level+1)
    right_points = split_list[1]
    if len(right_points) == 1:
        node.right = kdLeaf()
        node.right.value = right_points[0]
    else:
        node.right = kdTree(right_points, level+1)
    return node
```

```
looked_at = [] #for the visualization of looked at points
#function for performing rangeSearch on a KDTree
def rangeSearch(node, rec, level=0):
    result = []
    #determine if list should be split by x or y axis
    dim = level % k

    if type(node) is kdLeaf:
        looked_at.append(node.value)
        #check if point is within rectangle range
        if rec[0][0] <= node.value['x'] and rec[0][1] >= node.value['x']:
            if rec[1][0] <= node.value['y'] and rec[1][1] >= node.value['y']:
                result.append(node.value)
    else:
        if node.value > rec[dim][1]:
            #median is to the right of rectangle
            left = rangeSearch(node.left, rec, level+1)
            for e in left:
                result.append(e)
        elif node.value < rec[dim][0]:
            #median is to the left of rectangle
            right = rangeSearch(node.right, rec, level+1)
            for e in right:
                result.append(e)
        else:
            #median is inside the rectangle
            left = rangeSearch(node.left, rec, level+1)
            right = rangeSearch(node.right, rec, level+1)
            for e in left:
                result.append(e)
            for e in right:
                result.append(e)
    return result
```

Example Output for the full program using the 'points_4.ply' dataset:

```
Edges:
(Xmin, Xmax), (Ymin, Ymax)
((-8191, 8163), (-8185, 8177))

Press Enter to start

Random Rectangle:
(left, right), (bottom, top)
((-4101, 6411), (-6176, -1429))

Naive result in 0.00300002098083 seconds
KDTree result in 0.000999927520752 seconds

The kd-approach is 3.00023843586 times faster!

The results are the same!
```



Shown are the min, max values of the dataset. Then the program waits for user input.
After 'Enter' was pressed, a rectangle is randomly determined. In this case with the following dimensions:
$x_{min} = -4101, x_{max} = 6411, y_{min} = -6176, y_{max} = -1429$
Then the time needed to calculate the results and the speedup from naive to kd approach are shown. The results themselves are for space reasons not displayed.
Finally it is checked if the results are the same and a visualization for the kd approach is displayed. The Visualization shows all points of the dataset in blue, points of the result in green, looked at points in yellow, and the random rectangle in red.

Exercise 4 (Nearest Neighbor Search in a kD Tree) (4 points)