Chapter 16: Spatial indexing

Package installation

This chapter covers spatial indexing with *KD-trees*, *Quadtrees* and *R-trees*. The package requirement for these spatial indexes are the `scipy.spatial`, `pyqtree` and `rtree` modules respectively.

Anaconda

If you have Anaconda installed, the `scipy` package was installed together with, you only need to install `pyqtree` and `rtree`. Open the *Anaconda Prompt* and type in:

```
conda install -c conda-forge pyqtree rtree
```

Python Package Installer (pip)

If you have standalone Python3 and Jupyter Notebook install, open a command prompt / terminal and type in:

```
pip3 install scipy pyqtree rtree
```

You most likely have already installed `rtree`, as it was an optional dependency for `geopandas` in [Chapter 11 (11_spatial_vector.pdf)](11_spatial_vector.pdf).

Process the dataset

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities, and should be familiar from [Chapter 15 (15_graph_spanning_tree.pdf)](15_graph_spanning_tree.pdf).

In [1]:

```
import geopandas as gpd

cities = gpd.read_file('../data/hungary_cities.shp')
display(cities)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

Minimal bounding box

Calculate the minimal bounding box for all the points! (We will use it later.)

In [2]:

```
def get_x(point):
    return point.x

def get_y(point):
    return point.y

# Calculating the minimal bounding box
min_x = min(cities['geometry'], key = get_x).x # or cities.geometry
max_x = max(cities['geometry'], key = get_x).x
min_y = min(cities['geometry'], key = get_y).y
max_y = max(cities['geometry'], key = get_y).y

print("Bounding box: ({0:.1f}, {1:.1f}) - ({2:.1f}, {3:.1f})".format(min_x, min_
y, max_x, max_y))
```

```
Bounding box: (431339.2, 48431.5) - (934944.4, 359044.9)
```

Lambda functions *(optional)*

Python lambdas are little, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions.

Lambda functions can have any number of arguments but only one expression. The evaluated expression is the return value of the function.

A lambda function in python has the following syntax:

```
lambda arguments: expression
```

Lambda functions can be used wherever function objects are required.

In [3]:

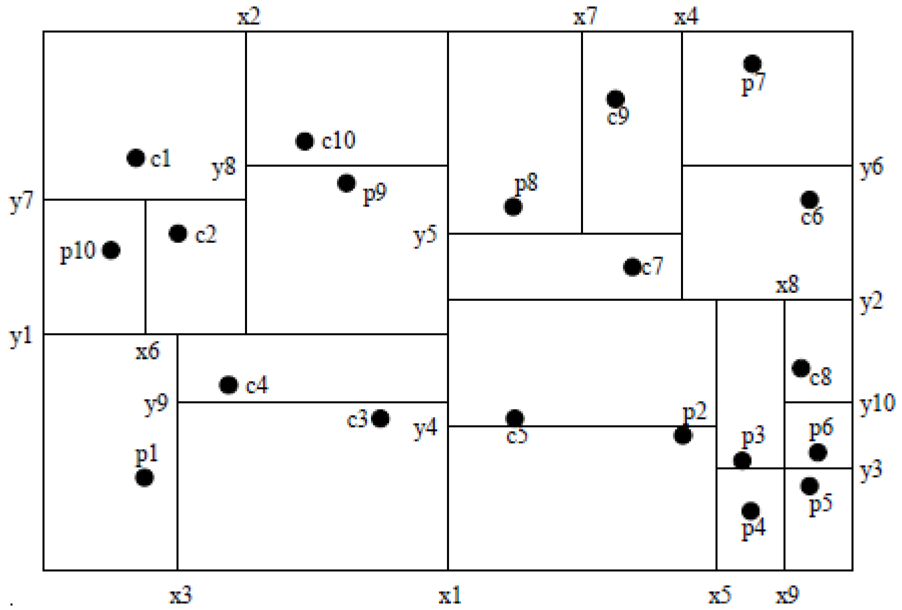
```
# Calculating the minimal bounding box
min_x = min(cities['geometry'], key = lambda p: p.x).x
max_x = max(cities['geometry'], key = lambda p: p.x).x
min_y = min(cities['geometry'], key = lambda p: p.y).y
max_y = max(cities['geometry'], key = lambda p: p.y).y

print("Bounding box: ({0:.1f}, {1:.1f}) - ({2:.1f}, {3:.1f})".format(min_x, min_
y, max_x, max_y))
```

```
Bounding box: (431339.2, 48431.5) - (934944.4, 359044.9)
```

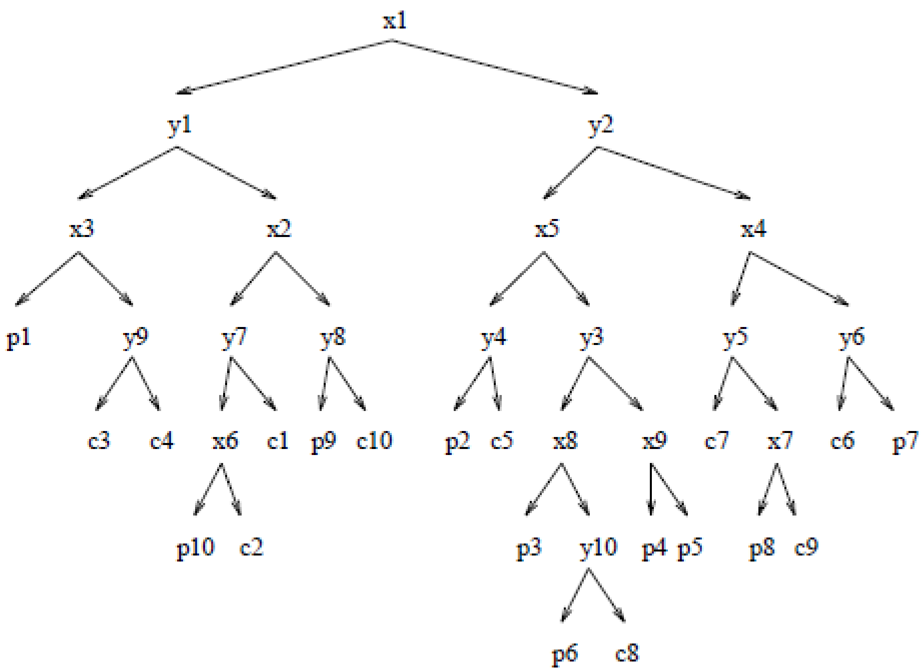
KdTree

A [kdTree (https://en.wikipedia.org/wiki/K-d_tree)](https://en.wikipedia.org/wiki/K-d_tree) (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a k-dimensional space. KdTrees are especially useful for searches involving a multidimensional search key, e.g. nearest neighbor searches and range searches.

Example KdTree:



Representation:



Select a random city and create a point which we will query later.

In [4]:

```
import random
random.seed(42) # for reproducibility

idx = random.randint(0, len(cities) - 1)
city = cities.iloc[idx]
print(city)
```

```
Id                                  2620
County                              PEST
City                         Szigethalom
Status                              town
KSH                                13277
geometry    POINT (646998.8 219076.5)
Name: 2619, dtype: object
```

Create the *query point*, by slightly distorting the location of the selected city.

In [5]:

```
from shapely.geometry import Point

city_point = city.geometry
query_point = Point(city_point.x + 1, city_point.y + 2)

print("City location: {0}".format(city_point))
print("Query location: {0}".format(query_point))
```

```
City location: POINT (646998.8 219076.5)
Query location: POINT (646999.8 219078.5)
```

Construct the KD-Tree

The `scipy` module can construct KD-Tree from a list of points, where each point is represented by a 2 element list or tuple.

In [6]:

```
points = [(p.x, p.y) for p in cities['geometry']]
print(points[:10])
```

```
[(610046.8, 187639.0), (577946.1, 89280.8), (721963.7, 273880.3), (8
12129.2, 331508.2), (809795.6, 331138.3), (791113.0, 341953.5), (808
664.6, 328230.8), (792853.4, 338292.6), (817486.0, 356056.1), (76721
4.3, 237868.5)]
```

Now the *KD-Tree* can be constructed.

In [7]:

```
import scipy.spatial
kdtree = scipy.spatial.KDTree(points)
```

Pointwise query

Query the closest neighbor to the query point.

In [8]:

```
print("City location: {0}".format(city_point))
print("Query location: {0}".format(query_point))

dist, idx = kdtree.query(query_point)

print("Closest neighbor: distance = {0:.4f}, index = {1}, point = {2}".format(di
st, idx, points[idx]))
print("Closest neighbor city: {0}".format(cities.iloc[idx]['City']))
```

```
City location: POINT (646998.8 219076.5)
Query location: POINT (646999.8 219078.5)
Closest neighbor: distance = 2.2361, index = 2619, point = (646998.
8, 219076.5)
Closest neighbor city: Szigethalom
```

Query the 3 closest neighbors to the query point.

In [9]:

```
distances, indices = kdtree.query(query_point, k = 3)

print("Query location: {0}".format(query_point))
print("3 closest neighbors:")
for i in range(len(indices)):
    idx = indices[i]
    dist = distances[i]
    print("{0}. neighbor: distance = {1:.4f}, index = {2}, point = {3}, city =
{4}".format(i+1, dist, idx, points[idx], cities.iloc[idx]['City']))
```

```
Query location: POINT (646999.8 219078.5)
3 closest neighbors:
1. neighbor: distance = 2.2361, index = 2619, point = (646998.8, 219
076.5), city = Szigethalom
2. neighbor: distance = 3087.9825, index = 2864, point = (643968.9,
219669.5), city = Tököl
3. neighbor: distance = 3250.9858, index = 2622, point = (649095.2,
221564.1), city = Szigetszentmiklós
```

Query the 50 closest neighbors to the query point within 10km.

In [10]:

```
distances, indices = kdtree.query(query_point, k = 50, distance_upper_bound = 10
000)
print("Distance list: %s" % distances)
print("Index list: %s" % indices)
```

```
Distance list: [2.23606798e+00 3.08798248e+03 3.25098578e+03 4.37588
711e+03
 5.82197378e+03 5.82989477e+03 6.07146723e+03 6.09635922e+03
 6.88039701e+03 7.86611138e+03 8.30077594e+03 8.36841190e+03
 9.92969764e+03            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf            inf            inf
            inf            inf]
Index list: [2619 2864 2622 2678 1619  971  660 2618 2547  646  733
586   95 3147
 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 31
47
 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 3147 31
47
 3147 3147 3147 3147 3147 3147 3147 3147]
```

Most likely will only find less than 50 neighbors in a 10km range, but the index list has still 50 elements. For the invalid elements the `indices[i]` is not a valid index, but instead equals to `len(cities)` . So with a simple check we can detect the end of the valid results.

In [11]:

```
valid_indices = [idx for idx in indices if idx < len(cities)]
print(valid_indices)
```

```
[2619, 2864, 2622, 2678, 1619, 971, 660, 2618, 2547, 646, 733, 586,
95]
```

In [12]:

```
print("50 closest neighbors within 10km:")
for i in range(len(valid_indices)):
    idx = valid_indices[i]
    dist = distances[i]
    print("{0}. neighbor: distance = {1:.1f}, index = {2}, location = {3}, city
 = {4}".format(i+1, dist, idx, points[idx], cities.iloc[idx]['City']))
```

```
50 closest neighbors within 10km:
1. neighbor: distance = 2.2, index = 2619, location = (646998.8, 219
076.5), city = Szigethalom
2. neighbor: distance = 3088.0, index = 2864, location = (643968.9,
219669.5), city = Tököl
3. neighbor: distance = 3251.0, index = 2622, location = (649095.2,
221564.1), city = Szigetszentmiklós
4. neighbor: distance = 4375.9, index = 2678, location = (651262.9,
220065.6), city = Taksony
5. neighbor: distance = 5822.0, index = 1619, location = (646007.0,
213341.8), city = Majosháza
6. neighbor: distance = 5829.9, index = 971, location = (644860.0, 2
24501.5), city = Halásztelek
7. neighbor: distance = 6071.5, index = 660, location = (651533.1, 2
15039.7), city = Dunavarsány
8. neighbor: distance = 6096.4, index = 2618, location = (643902.1,
213827.8), city = Szigetcsép
9. neighbor: distance = 6880.4, index = 2547, location = (640127.5,
218744.8), city = Százhalombatta
10. neighbor: distance = 7866.1, index = 646, location = (653626.9,
223316.1), city = Dunaharaszti
11. neighbor: distance = 8300.8, index = 733, location = (640833.1,
224635.0), city = Érd
12. neighbor: distance = 8368.4, index = 586, location = (651301.4,
211900.3), city = Délegyháza
13. neighbor: distance = 9929.7, index = 95, location = (647222.5, 2
09151.3), city = Áporka
```

Exercise

Task 1: Implement a linear search for the closest point instead of using a *KD-Tree*!

In [13]:

```
def find_closest(points, query):
    min_dist = None
    min_point = None
    for point in points:
        dist = point.distance(query)
        if min_dist is None or dist < min_dist:
            min_dist = dist
            min_point = point
    return min_point

print("City location: {0}".format(city_point))
print("Query location: {0}".format(query_point))
closest_point = find_closest(cities['geometry'], query_point)
print("Closest location: {0}".format(closest_point))
```

```
City location: POINT (646998.8 219076.5)
Query location: POINT (646999.8 219078.5)
Closest location: POINT (646998.8 219076.5)
```

Task 2: Compare the execution time of the linear search and the spatial index query (logarithmic asymptotic complexity) approach!

Hint: import the `time` module to record the timestamp before and after the execution of the desired algorithm:

```
start = time.time()
# ... measured code ...
end = time.time()
print("Execution time: {0:.6f}s".format(end-start))
```

In [14]:

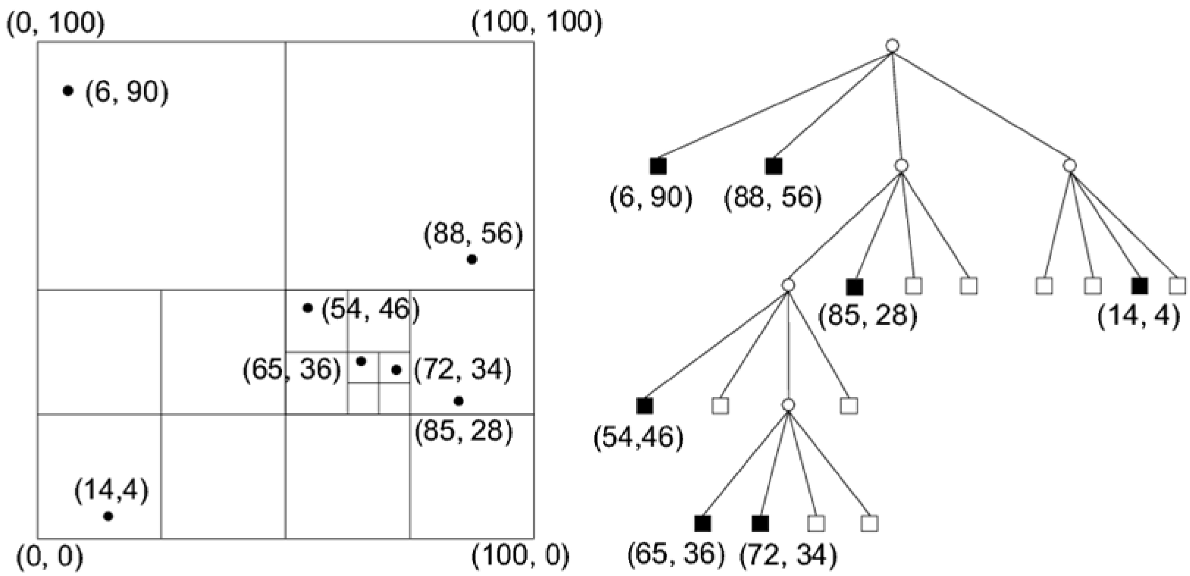
```
import time

start = time.time()
find_closest(cities['geometry'], query_point)
end = time.time()
print("Linear search execution time: {0:.6f}s".format(end-start))

start = time.time()
kdtree.query(query_point)
end = time.time()
print("KD-tree search execution time: {0:.6f}s".format(end-start))
```

```
Linear search execution time: 0.018702s
KD-tree search execution time: 0.000258s
```

Quadtree

A [quadtree (https://en.wikipedia.org/wiki/Quadtree)](https://en.wikipedia.org/wiki/Quadtree) is a tree data structure in which each internal node has exactly four children. The 3 dimensional analog of quadtree is the [octree (https://en.wikipedia.org/wiki/Octree)](https://en.wikipedia.org/wiki/Octree).

Quadtree example:



Create a 10x10km query area around a point.

In [15]:

```
query_area_size = 10000
query_area = (
    query_point.x - query_area_size/2,
    query_point.y - query_area_size/2,
    query_point.x + query_area_size/2,
    query_point.y + query_area_size/2
)
print("Query area: {0}, side length = {1:.1f} km".format(query_area, query_area_
size / 1000))
```

```
Query area: (641999.8, 214078.5, 651999.8, 224078.5), side length =
10.0 km
```

Construct the Quad-tree

In [16]:

```
import pyqtree

quadtree = pyqtree.Index(bbox=(min_x, min_y, max_x, max_y))
for i in range(len(points)):
    obj = { "id": i, "point": points[i] }
    quadtree.insert(obj, points[i]) # object, bbox
```

Note: for a polygon, the first argument should be the indexed object (e.g. the polygon itself), and the second argument should be the bounding box of the polygon.

Areawise query

In [17]:

```
matches = quadtree.intersect(query_area)
print("Matches: {0}".format(matches))
```

```
Matches: [{'id': 660, 'point': (651533.1, 215039.7)}, {'id': 2619,
'point': (646998.8, 219076.5)}, {'id': 2622, 'point': (649095.2, 221
564.1)}, {'id': 2678, 'point': (651262.9, 220065.6)}, {'id': 2864,
'point': (643968.9, 219669.5)}]
```

In [18]:

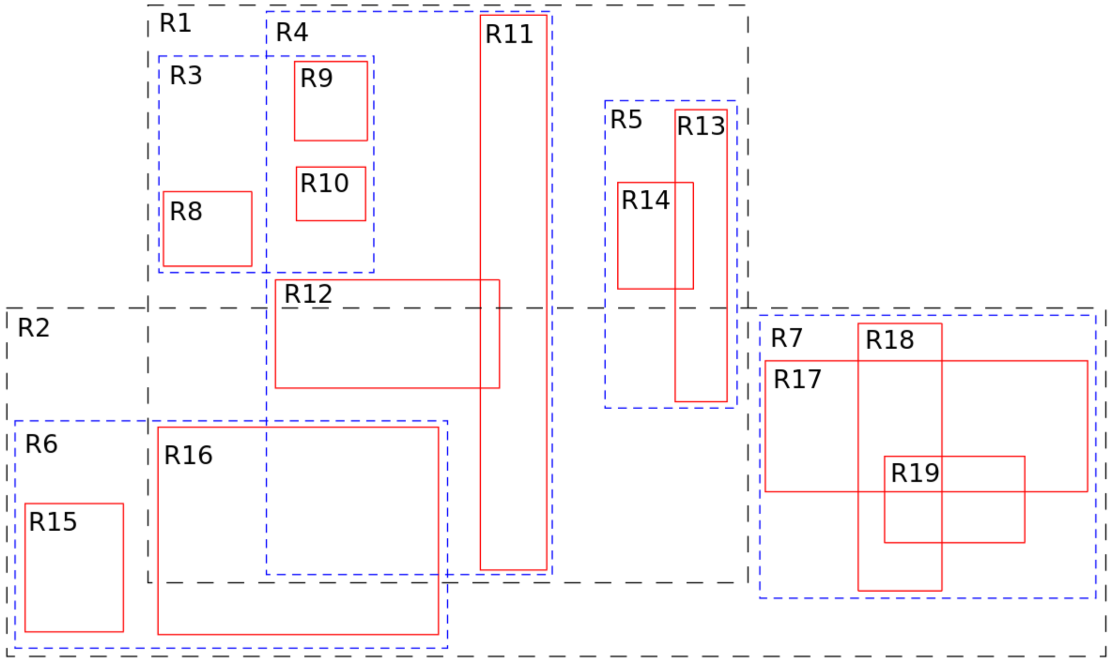
```
for obj in matches:
    print("Index: {0}, Location: {1}, City: {2}".format(obj['id'], obj['point'],
 cities.iloc[obj['id']]['City']))
```

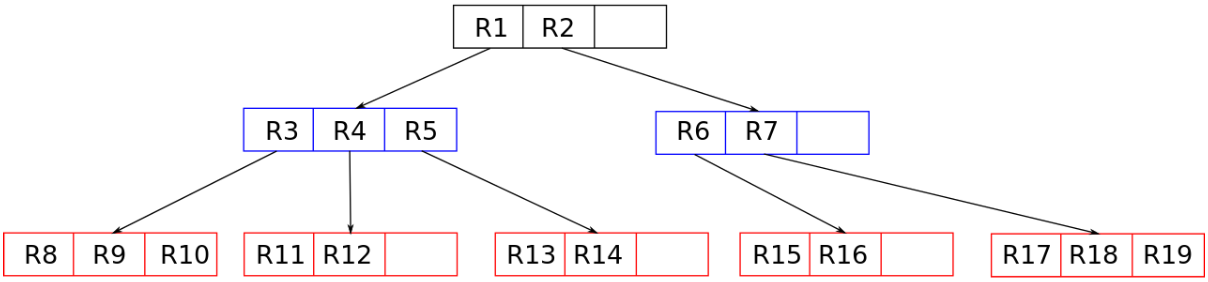
```
Index: 660, Location: (651533.1, 215039.7), City: Dunavarsány
Index: 2619, Location: (646998.8, 219076.5), City: Szigethalom
Index: 2622, Location: (649095.2, 221564.1), City: Szigetszentmiklós
Index: 2678, Location: (651262.9, 220065.6), City: Taksony
Index: 2864, Location: (643968.9, 219669.5), City: Tököl
```

R-Tree

Inspired by the [B-tree (https://en.wikipedia.org/wiki/B-tree)](https://en.wikipedia.org/wiki/B-tree) for scalara data, he key idea of the [R-tree (https://en.wikipedia.org/wiki/R-tree)](https://en.wikipedia.org/wiki/R-tree) index structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree, The *"R"* in R-tree stands for rectangle.

R-tree for 2 dimensional data:





We will use the same `query_area` for demonstration, as before with the *Quadtree*.

Construct the R-Tree

In [19]:

```
from rtree import index as rtree_index

rtree = rtree_index.Index()
for i in range(len(points)):
    rtree.insert(i, points[i]) # index, bbox
```

Areawise query

In [20]:

```
matches = rtree.intersection(query_area)
print("Matches: {0}".format(list(matches)))
```

```
Matches: [2622, 2678, 2619, 2864, 660]
```

In [21]:

```
matches = rtree.intersection(query_area)
for idx in matches:
    city = cities.iloc[idx]
    print("Index: {0}, Location: {1}, City: {2}".format(idx, city['geometry'], c
ity['City']))
```

```
Index: 2622, Location: POINT (649095.2 221564.1), City: Szigetszentm
iklós
Index: 2678, Location: POINT (651262.9 220065.6), City: Taksony
Index: 2619, Location: POINT (646998.8 219076.5), City: Szigethalom
Index: 2864, Location: POINT (643968.9 219669.5), City: Tököl
Index: 660, Location: POINT (651533.1 215039.7), City: Dunavarsány
```

GeoPandas integration

If the `rtree` module is installed, the `geopandas` module utilizes an *R-tree* in the background to spatially index the spatial objects in a *GeoDataFrame*.

This spatial index can be accessed directly as the `sindex` property of the *GeoDataFrame*:

In [22]:

```
print(cities.sindex)
matches = cities.sindex.intersection(query_area)
print("Matches: {0}".format(list(matches)))
```

```
rtree.index.Index(bounds=[431339.156, 48431.5, 934944.4, 359044.9],
size=3147)
Matches: [660, 2619, 2864, 2678, 2622]
```

The *R-Tree* spatial index is also used by the `sjoin()` and `clip()` function of *geopandas*.