Wyszukiwanie geometryczne – przeszukiwanie obszarów ortogonalnych – quadtree oraz kd-drzewa

Autorzy: Łukasz Dubiel, Paweł Kruczkiewicz

Ogólny problem

Dane – zbiór punktów P z przestrzeni d-wymiarowej

Zapytanie: dla zadanych xi1, xi2 dla i ze zbioru A zwartego w {0, 1, …, d-1} znaleźć punkty q ze zbioru P takie, że

dla każdego i z A xi1 <= qi <= xi2

Zastosowanie *Range Search*

- geometria obliczeniowa:
- systemy informacji geograficznych
- bazy danych:
  - obiekty z bazy danych mające wiele atrybutów - punkty w wielowymiarowej przestrzeni - zapytanie o obiekty o atrybutach z określonych przedziałów
- projektowanie sieci
- projektowanie układów elektronicznych

Szczególny przypadek - płaszczyzna 2D

Dane – zbiór punktów P na płaszczyźnie.

Zapytanie: dla zadanych x1, x2, y1, y2 znaleźć punkty q ze zbioru P takie, że

$x1 \leq qx \leq x2$, $y1 \leq qy \leq y2$.

Podejście do problemu

- metoda brute force - mało efektywna, bez zastosowań praktycznych
- zastosowanie odpowiednich struktur danych:
  - kd-drzewa - wariant drzewa binarnego, w węzłach wewnętrznych przechowuje proste ortogonalne, z których każda dzieli pewien obszar na dwa wzdłuż pewnej osi,
  - quad tree - drzewo czwórkowe - węzeł wewnętrzny (obszar kwadratowy) ma cztery dzieci (dzieli się na cztery przystające obszary kwadratowe)

Quad Tree - wprowadzenie

Idea - podział obszaru kwadratowego na cztery równe obszary kwadratowe:

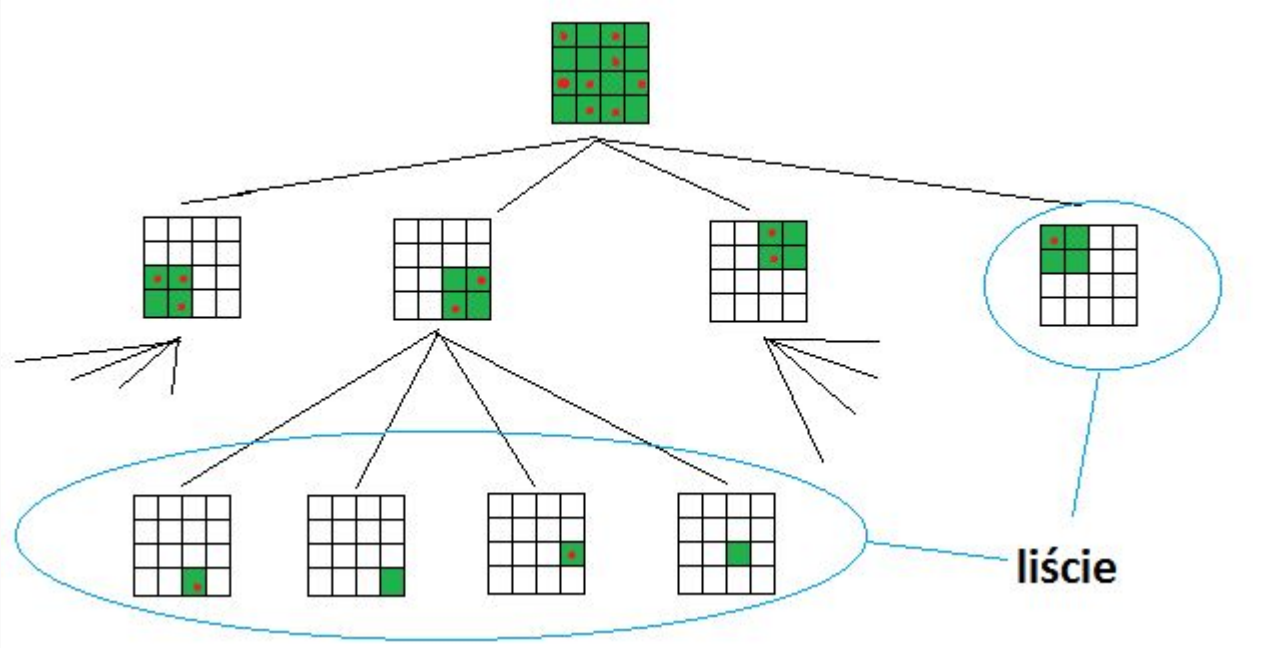
SW, SE, NE, NW

Rozdzielenie punktów do odpowiednich podobszarów

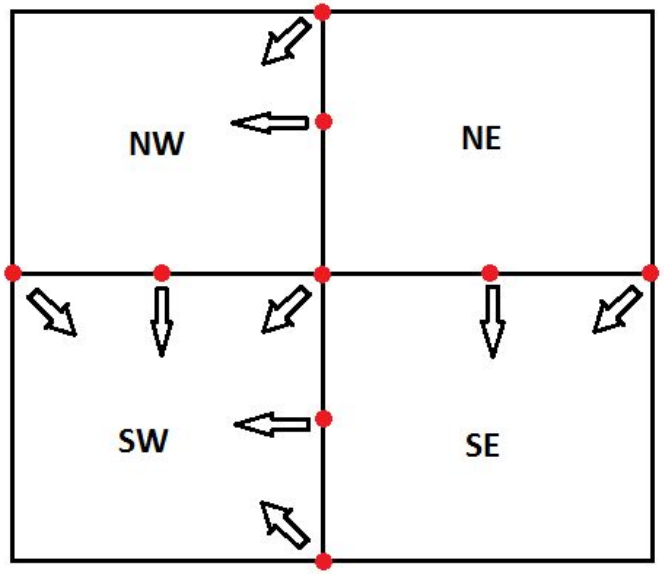
Jeśli dany obszar zawiera nie więcej niż ***nodeCapacity*** punktów np. 1 to staje się liściem

W przeciwnym przypadku ma czworo dzieci

Quad Tree - ilustracja poglądowa



Punkty na liniach podziału



Quad Tree - przykład

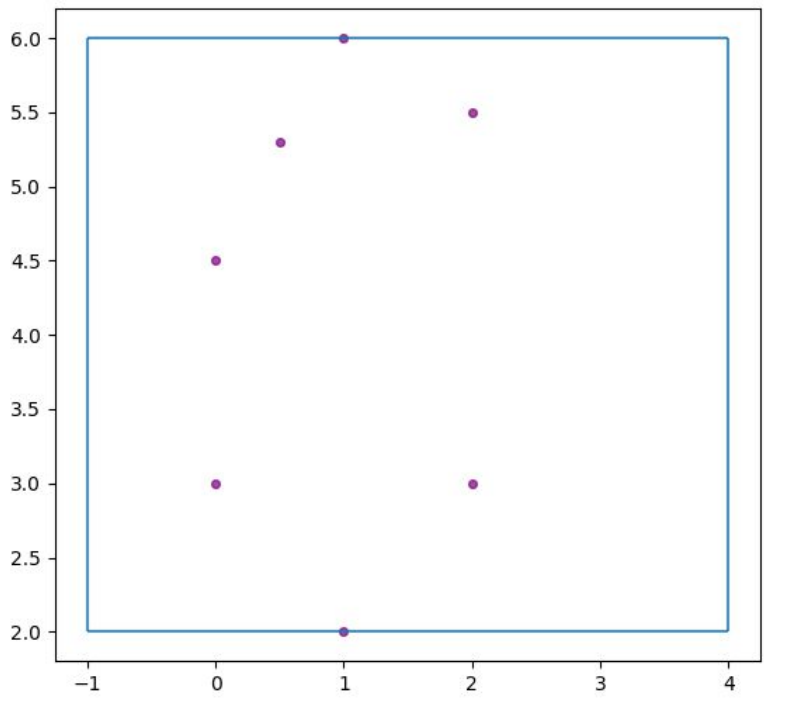
P = [(1, 2), (2, 3), (0, 3), (0, 5), (1, 6), (2, 6), (0.5, 0.5)]

Najpierw wyznaczamy obszar korzenia -

pewien minimalny obszar kwadratowy obejmujący wszystkie punkty, np.

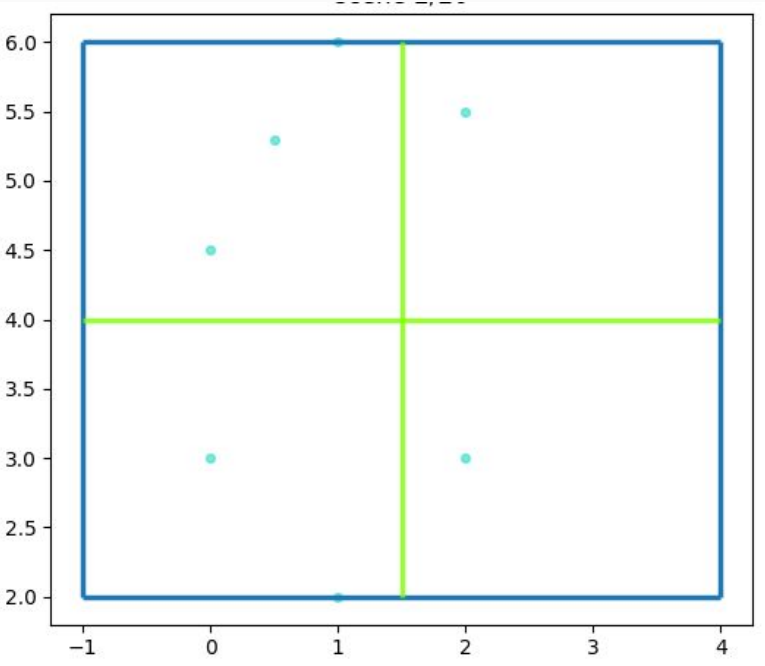
(x1 = -1, x2 = 4, y1 = 2, y2 = 6)

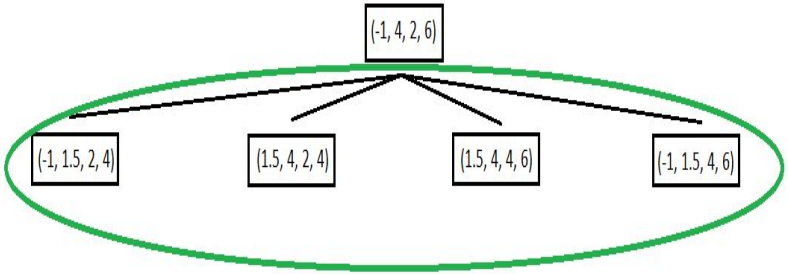
Wizualizacja konstrukcji



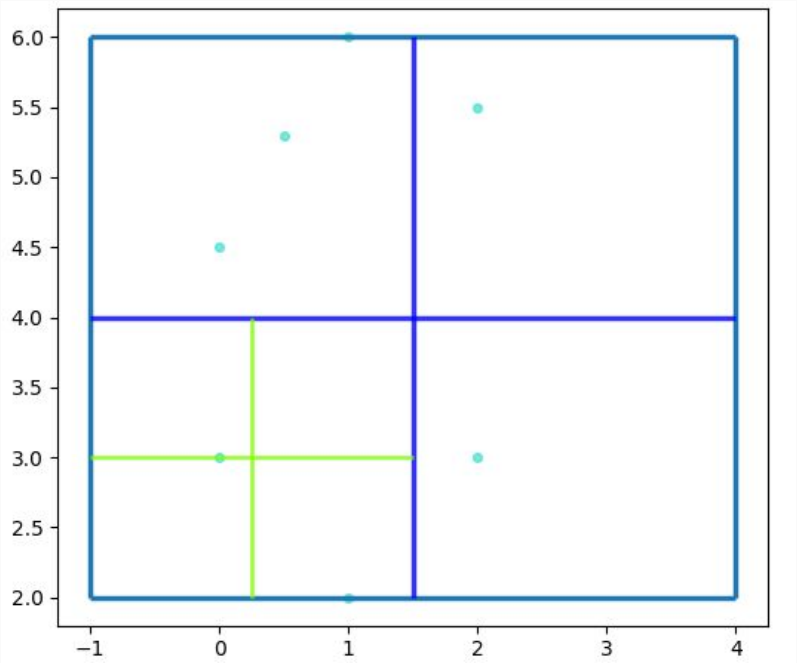
kwadraty - (obszary(x1, x2, y1, y2))
punkty w liściach na czerwono

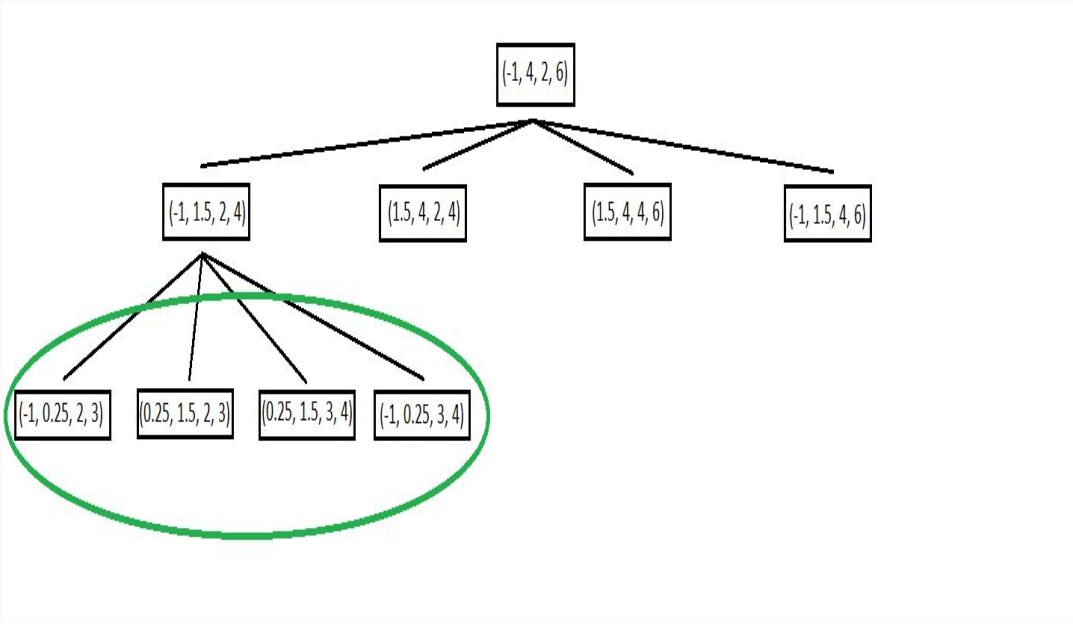
1 podział obszaru root-a

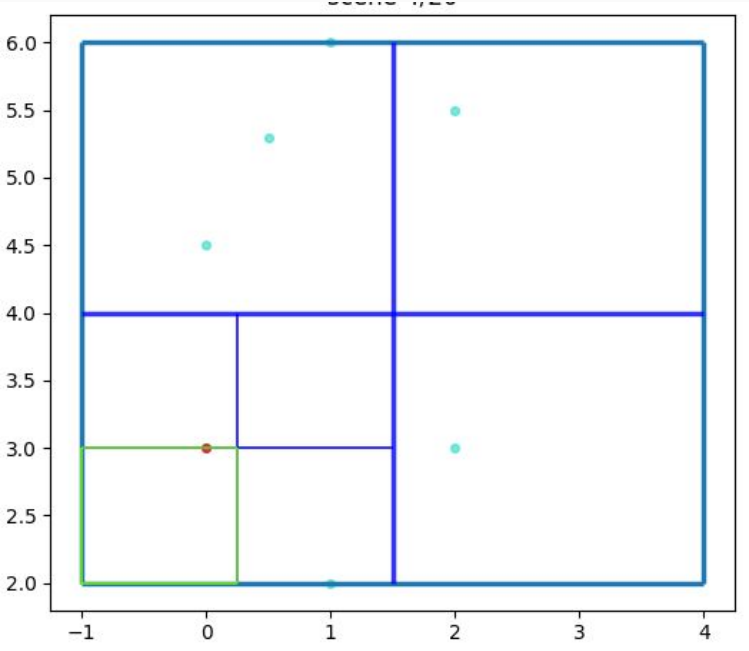


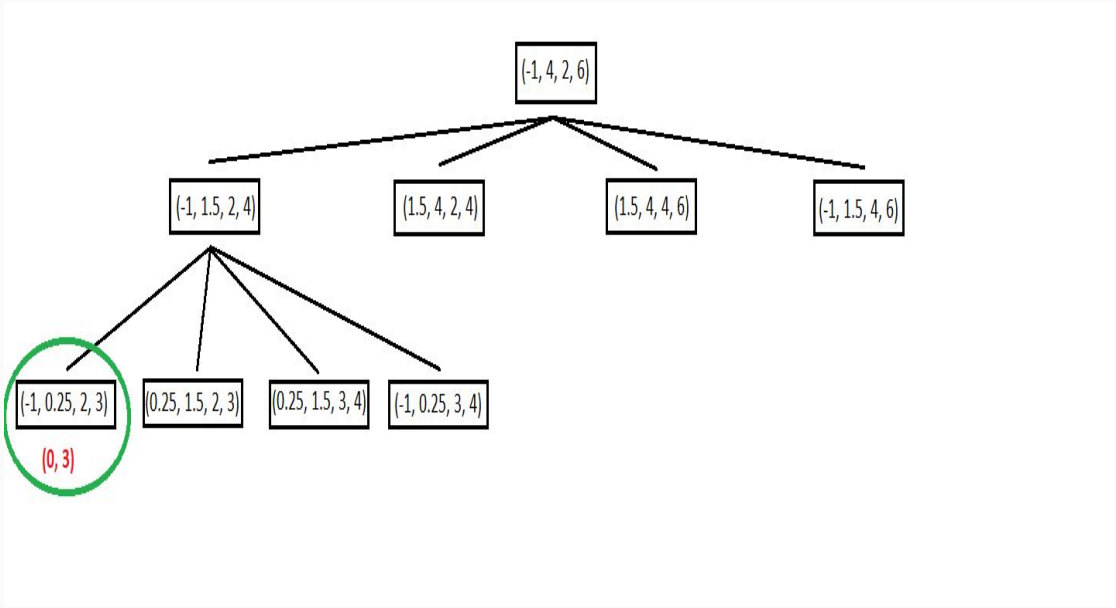


2

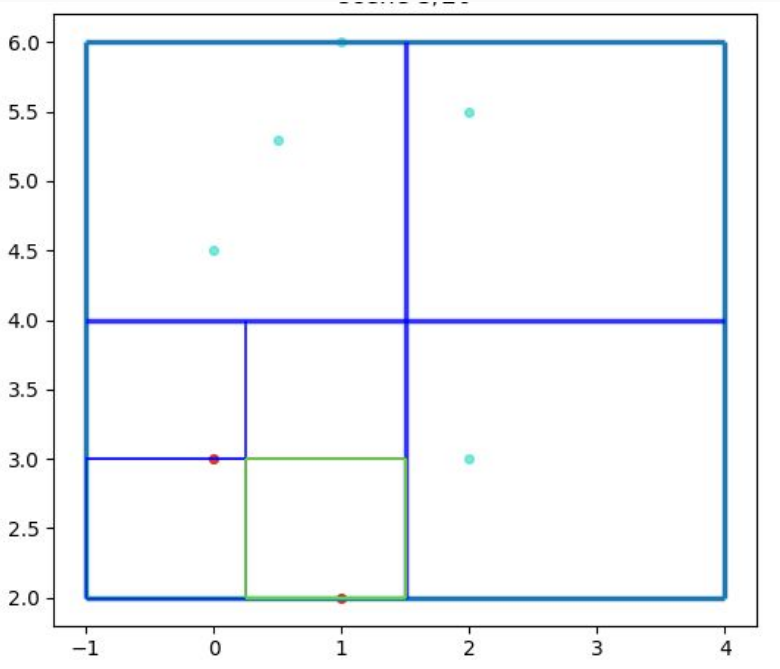

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

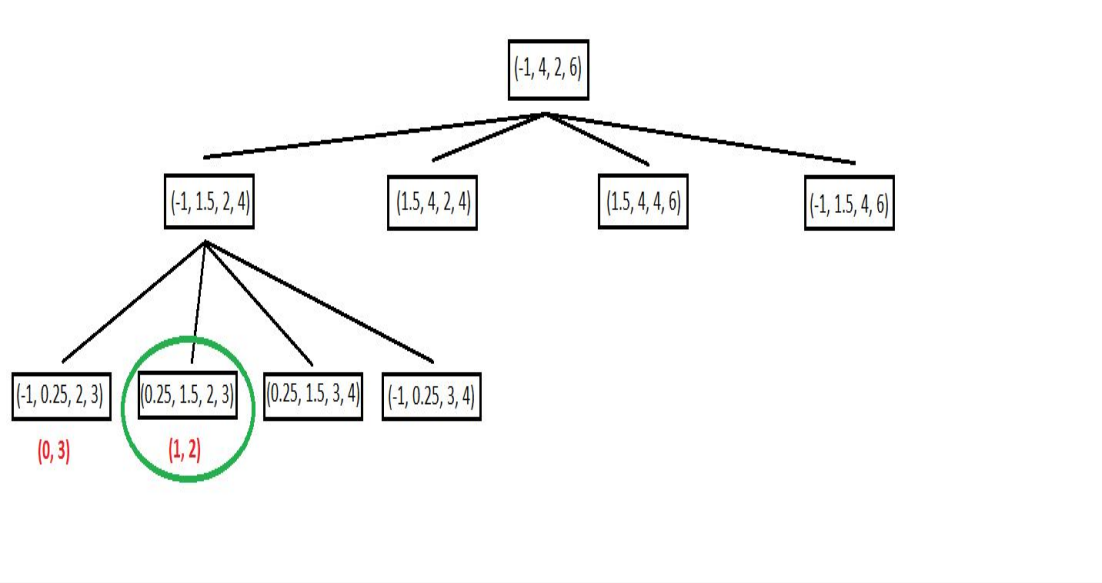

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)

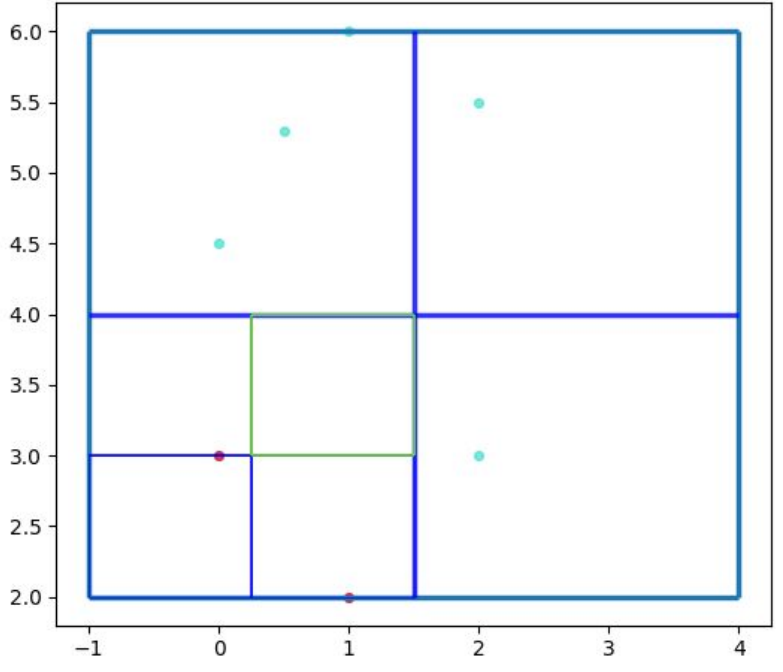

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

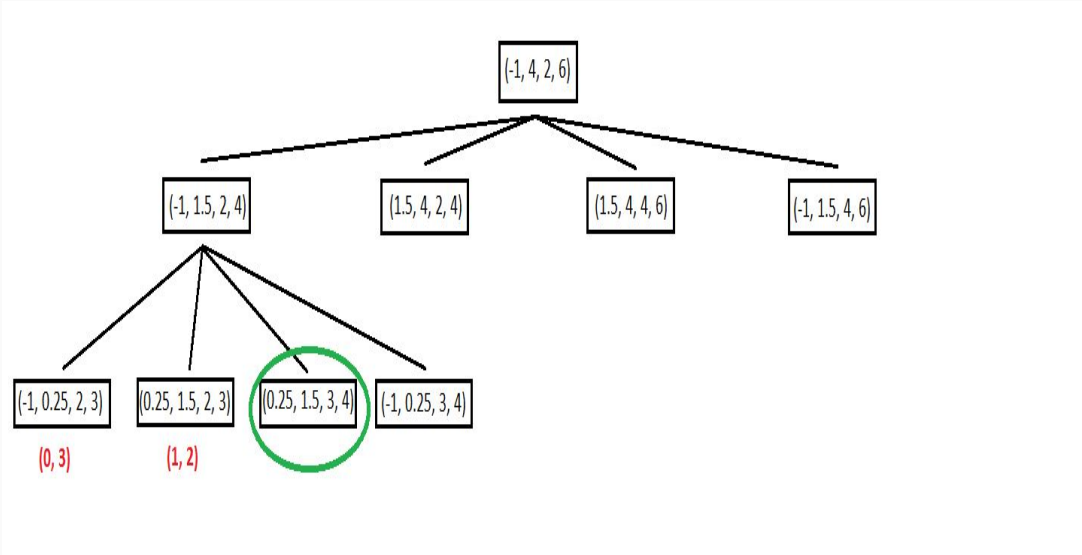

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)

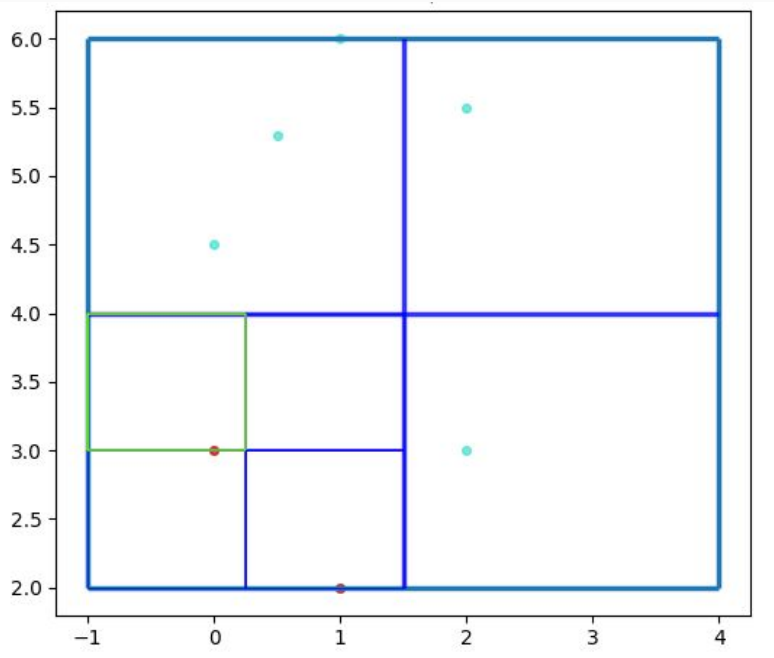
4

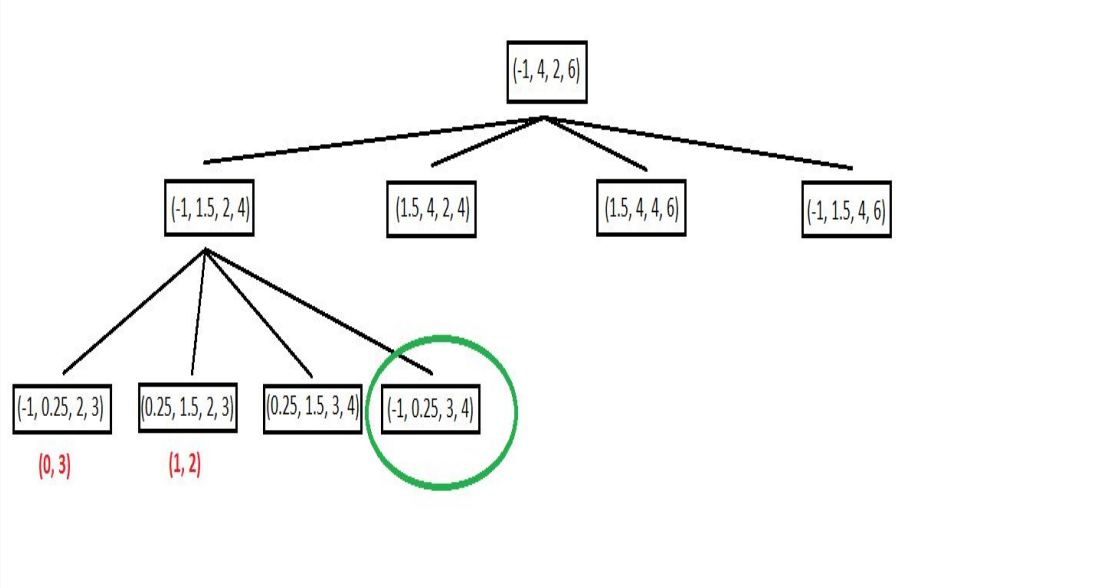

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

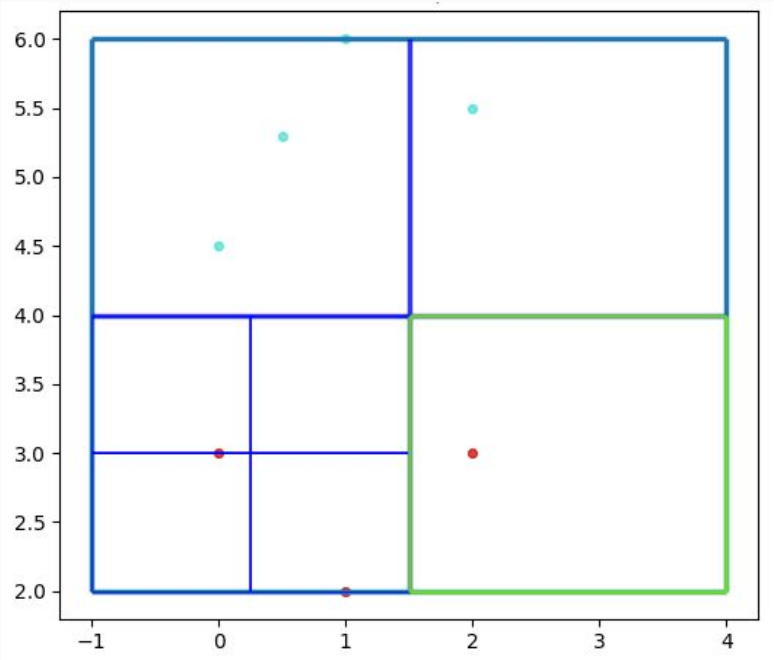

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(0, 3)
(1, 2)

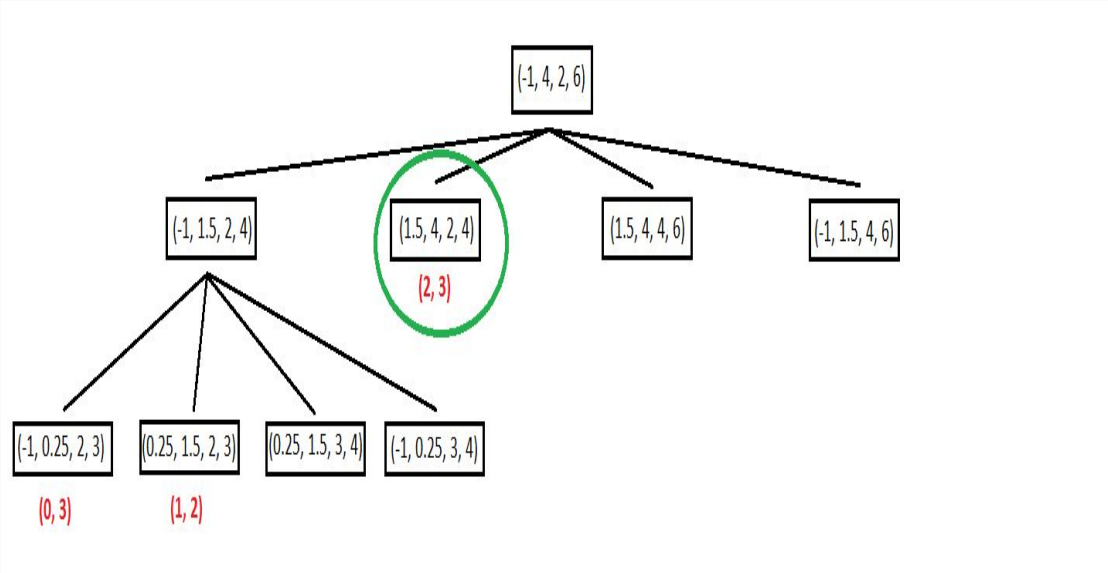

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

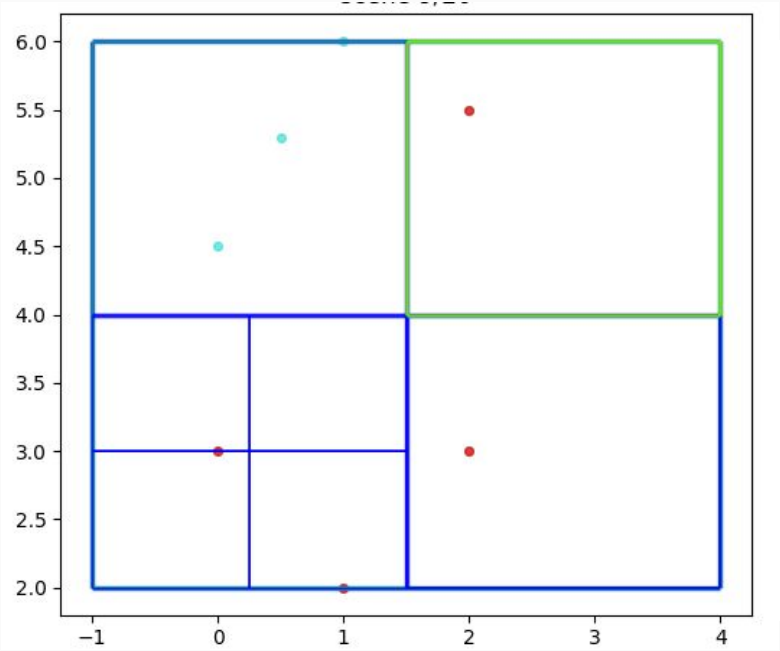

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(0, 3)
(1, 2)

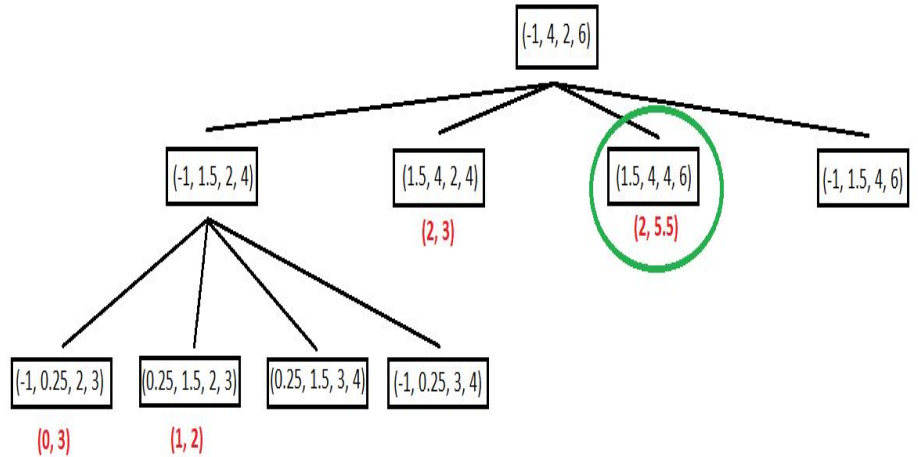

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

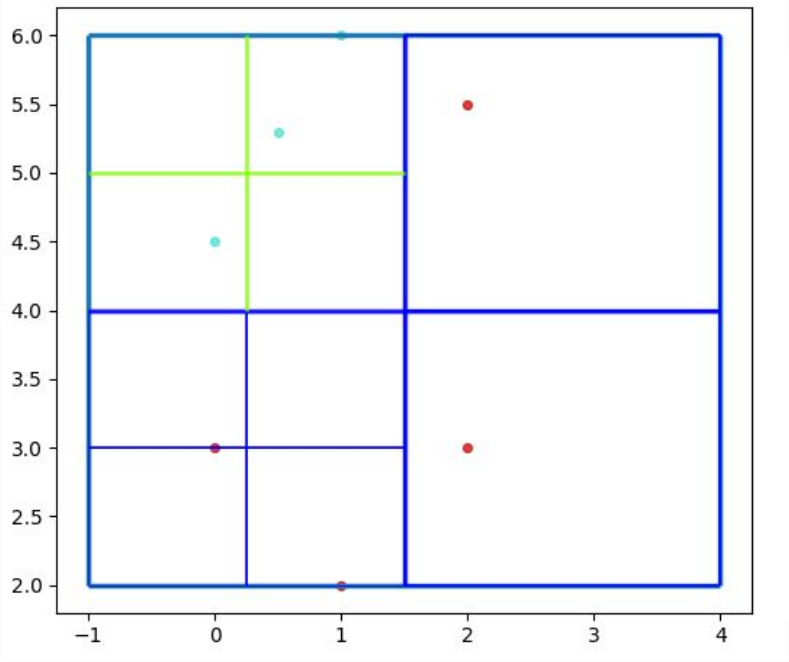

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(0, 3)
(1, 2)

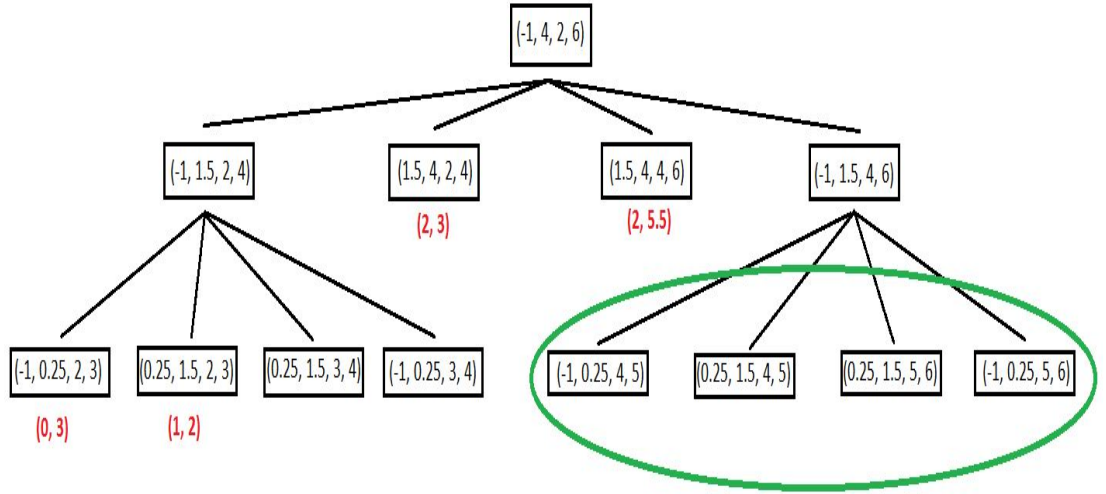

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

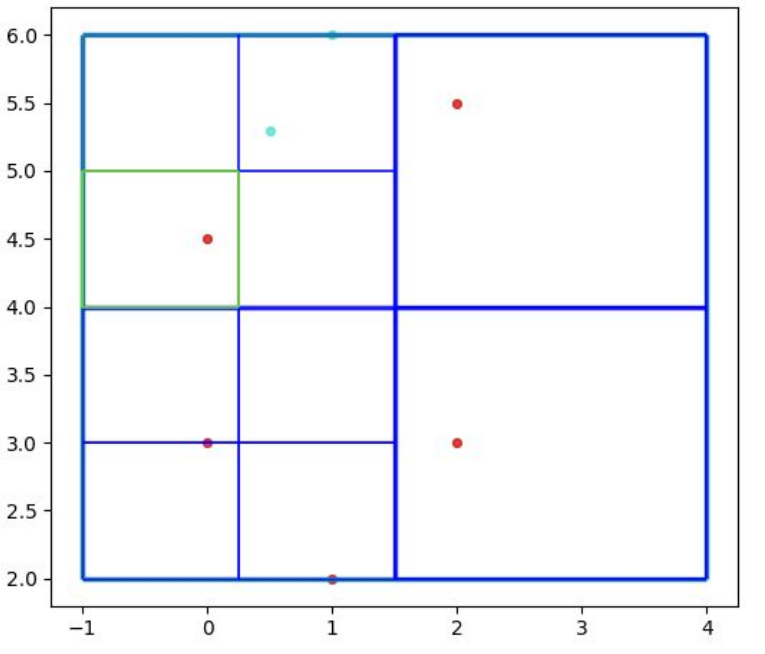

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)

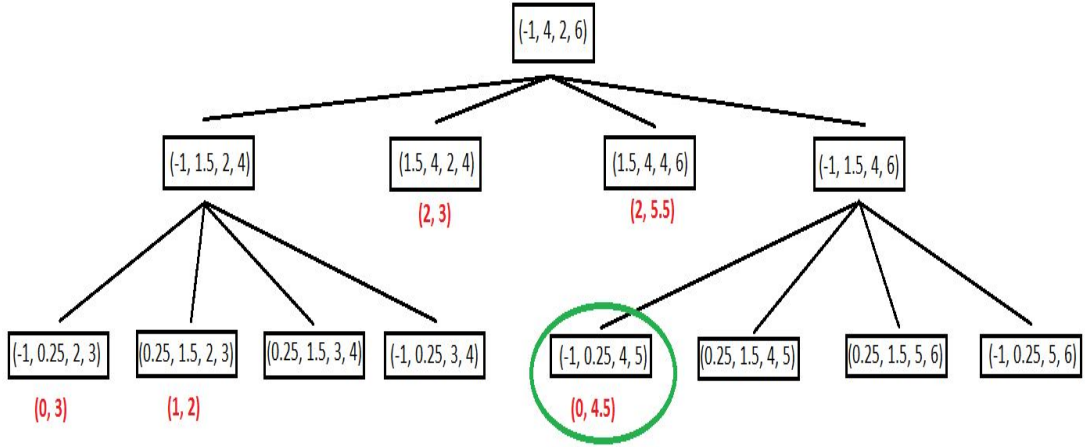

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

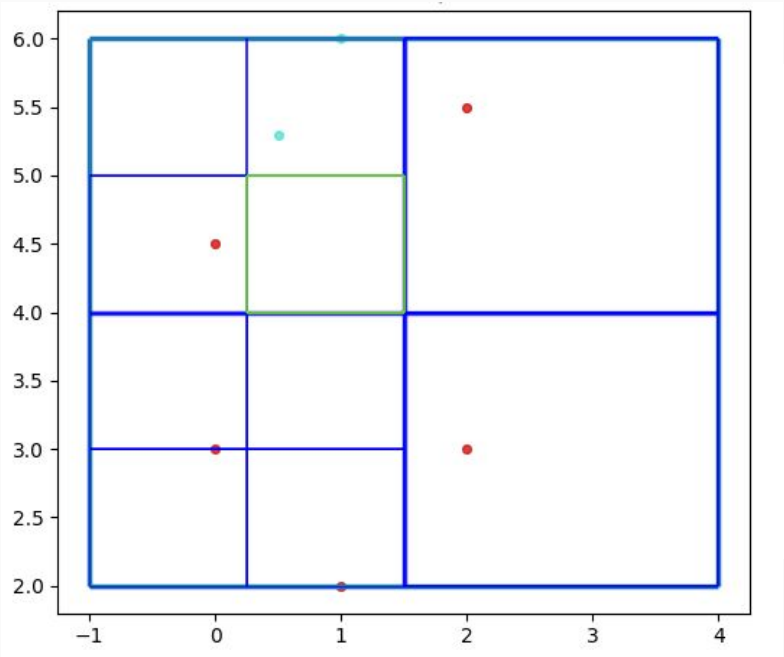

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)

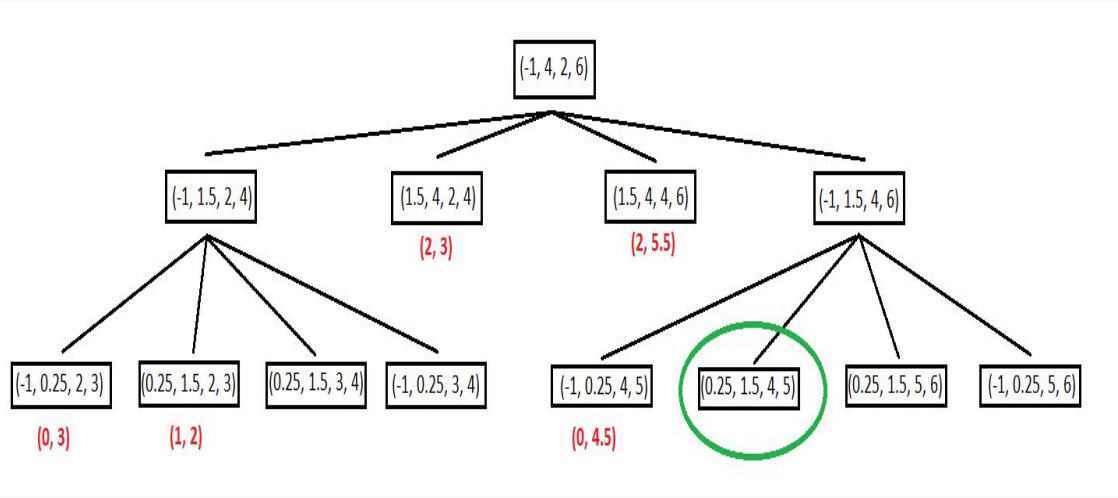

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

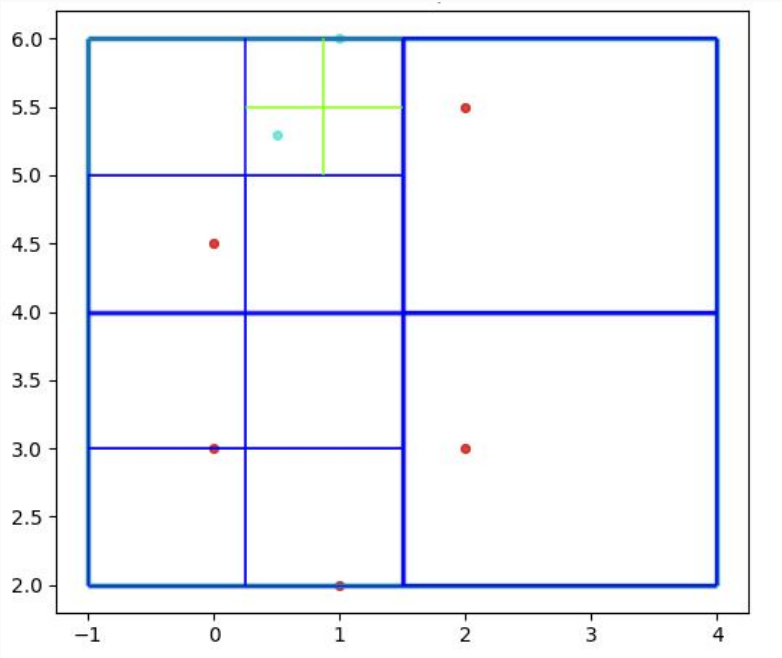

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(2, 3)
(2, 5.5)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0, 3)
(1, 2)

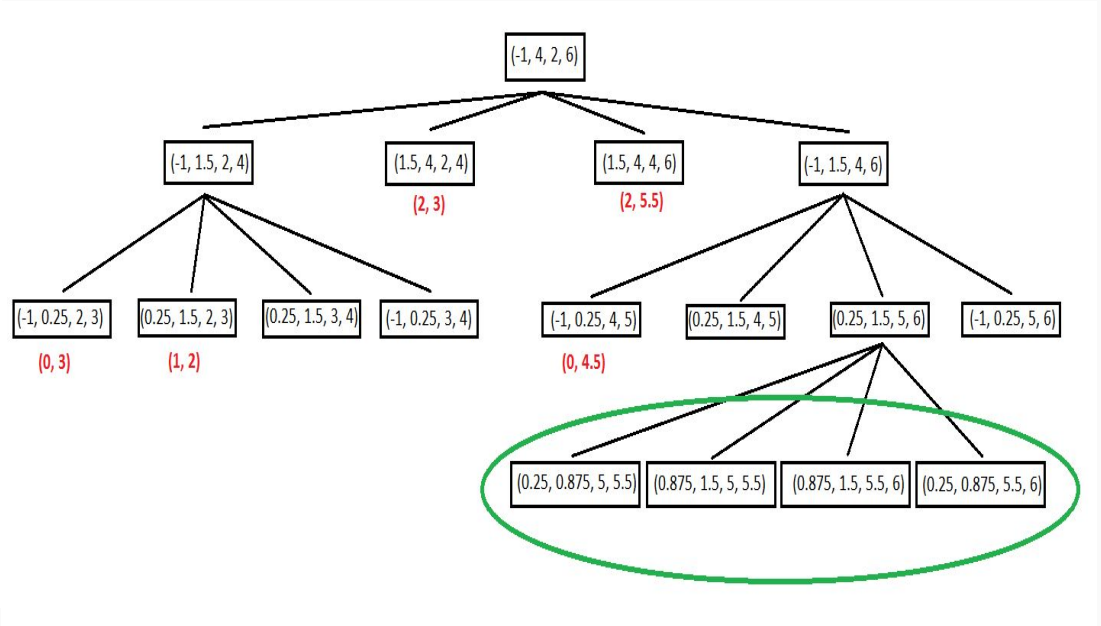

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

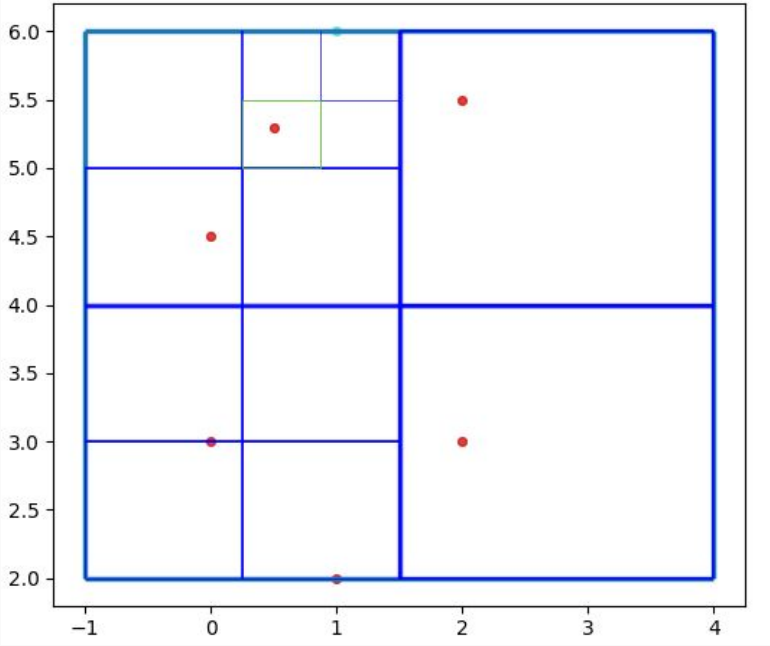

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(2, 3)
(2, 5.5)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0, 3)
(1, 2)
(0, 4.5)

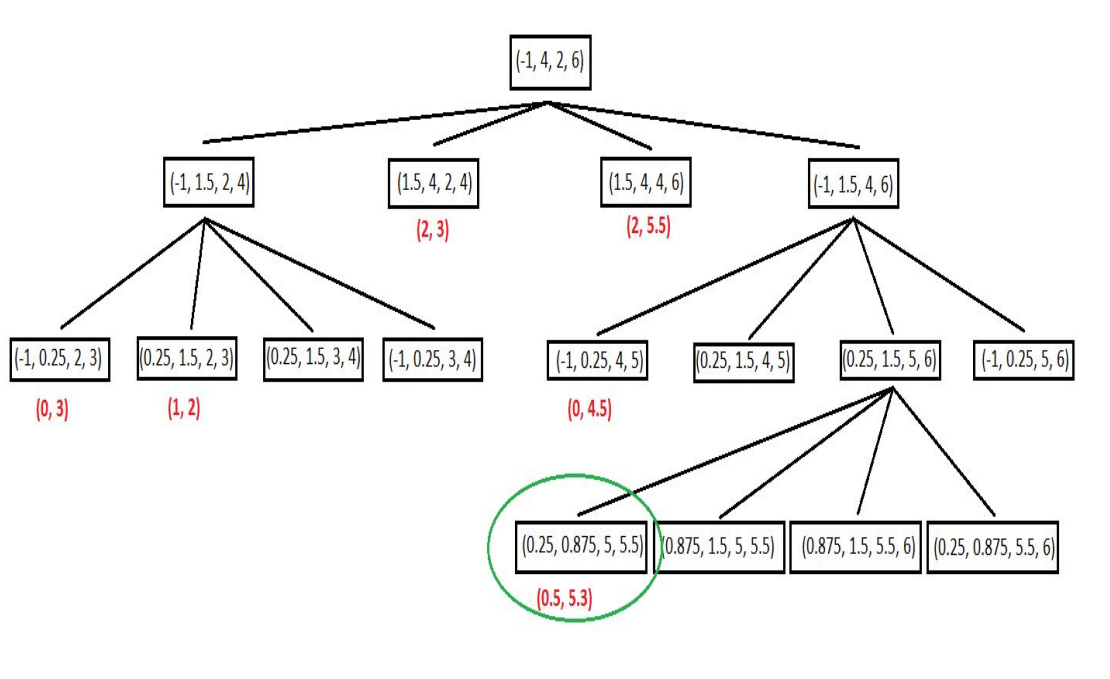

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

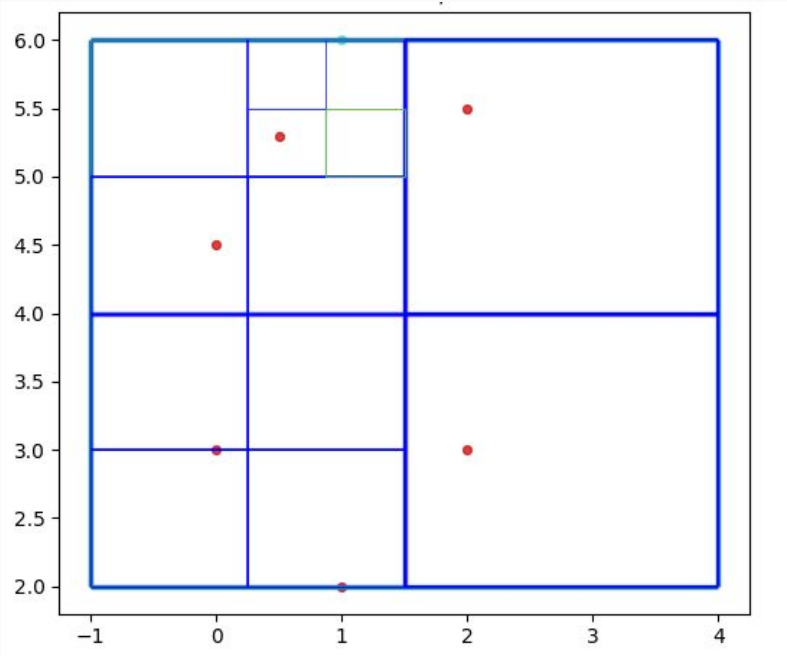

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(2, 3)
(2, 5.5)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0, 3)
(1, 2)
(0, 4.5)

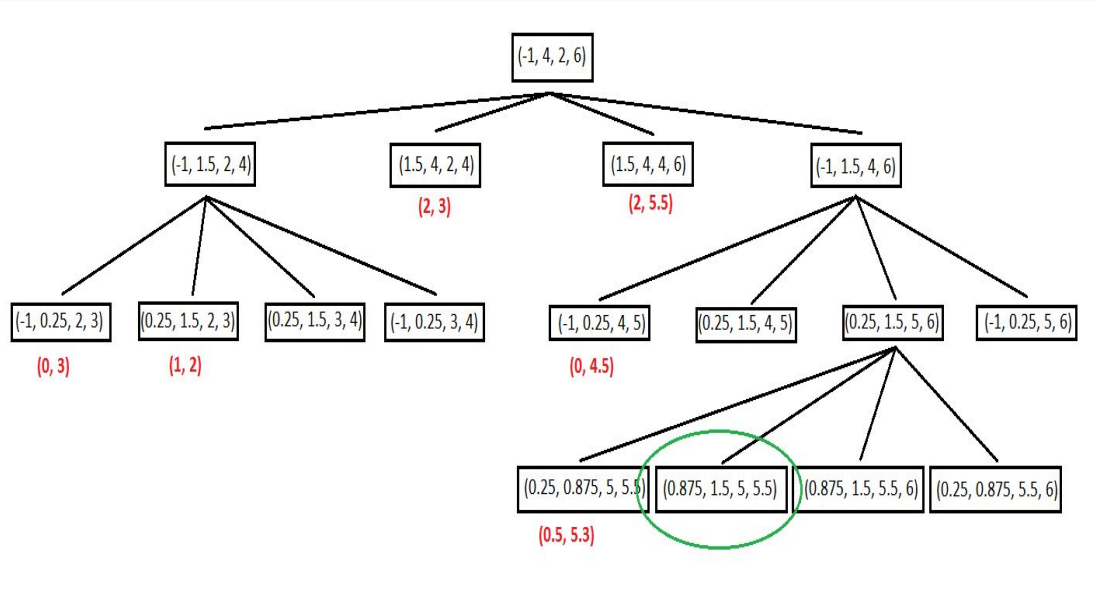

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

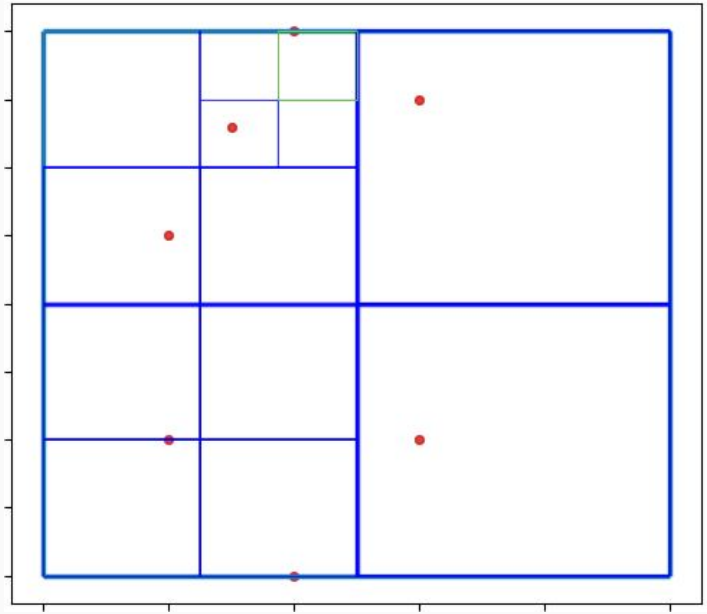

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0, 4.5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0.25, 0.875, 5, 5.5)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(0.25, 0.875, 5.5, 6)

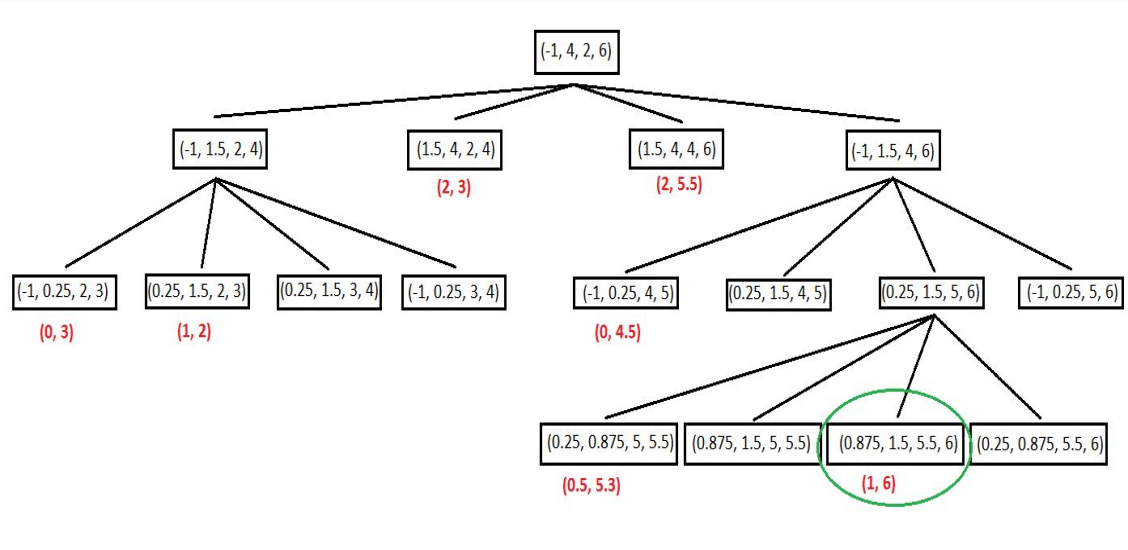

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

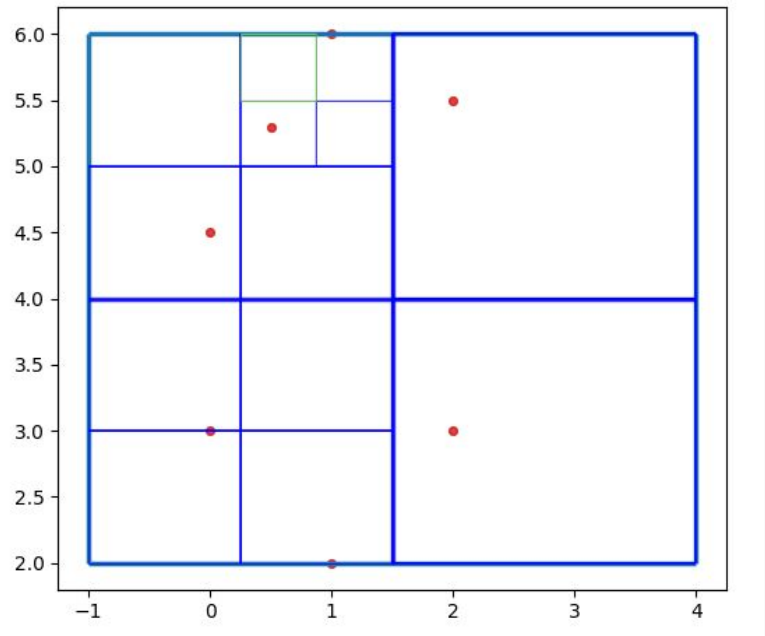

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0, 4.5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0.25, 0.875, 5, 5.5)
(0.5, 5.3)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(0.25, 0.875, 5.5, 6)

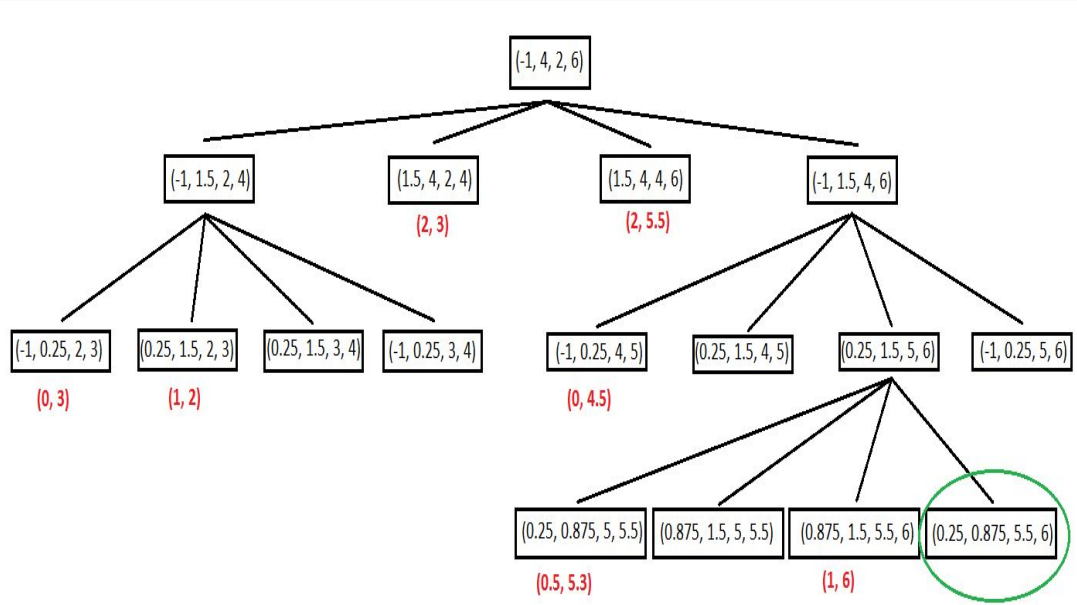

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

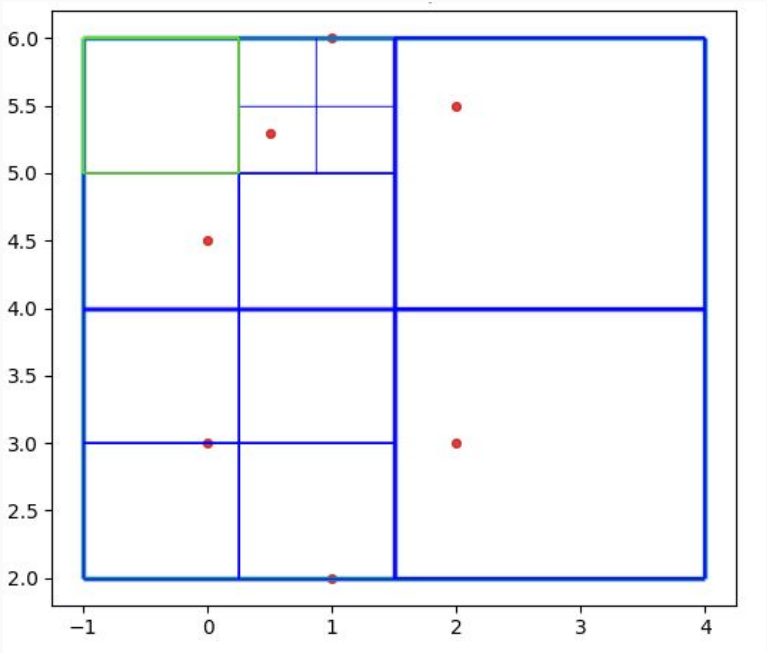

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0, 4.5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0.25, 0.875, 5, 5.5)
(0.5, 5.3)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(0.25, 0.875, 5.5, 6)

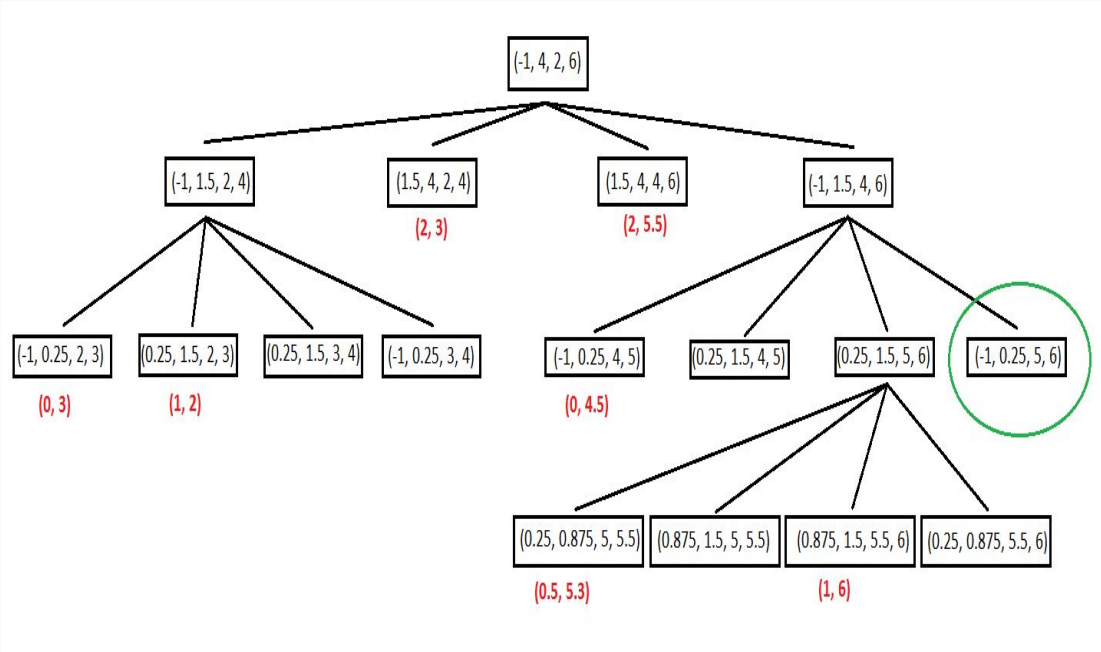


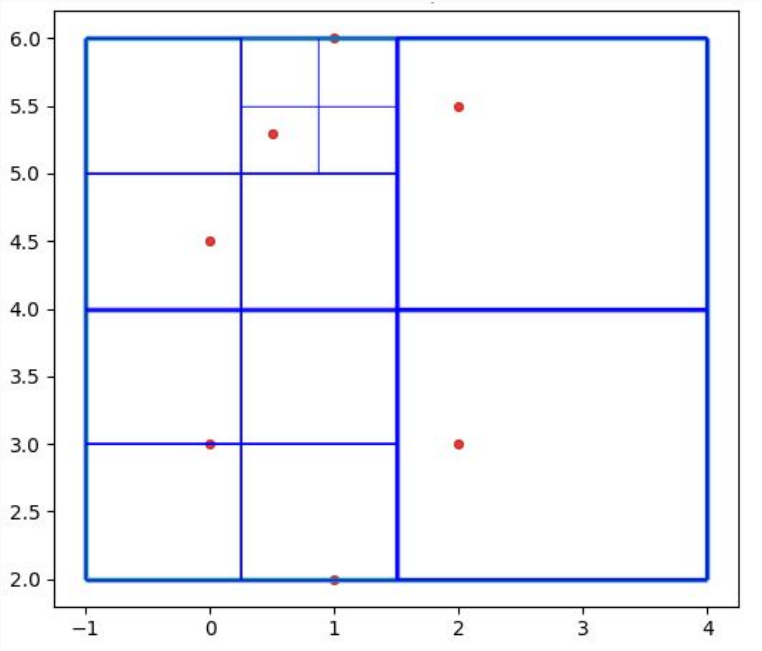

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0, 4.5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0.25, 0.875, 5, 5.5)
(0.5, 5.3)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(1, 6)
(0.25, 0.875, 5.5, 6)

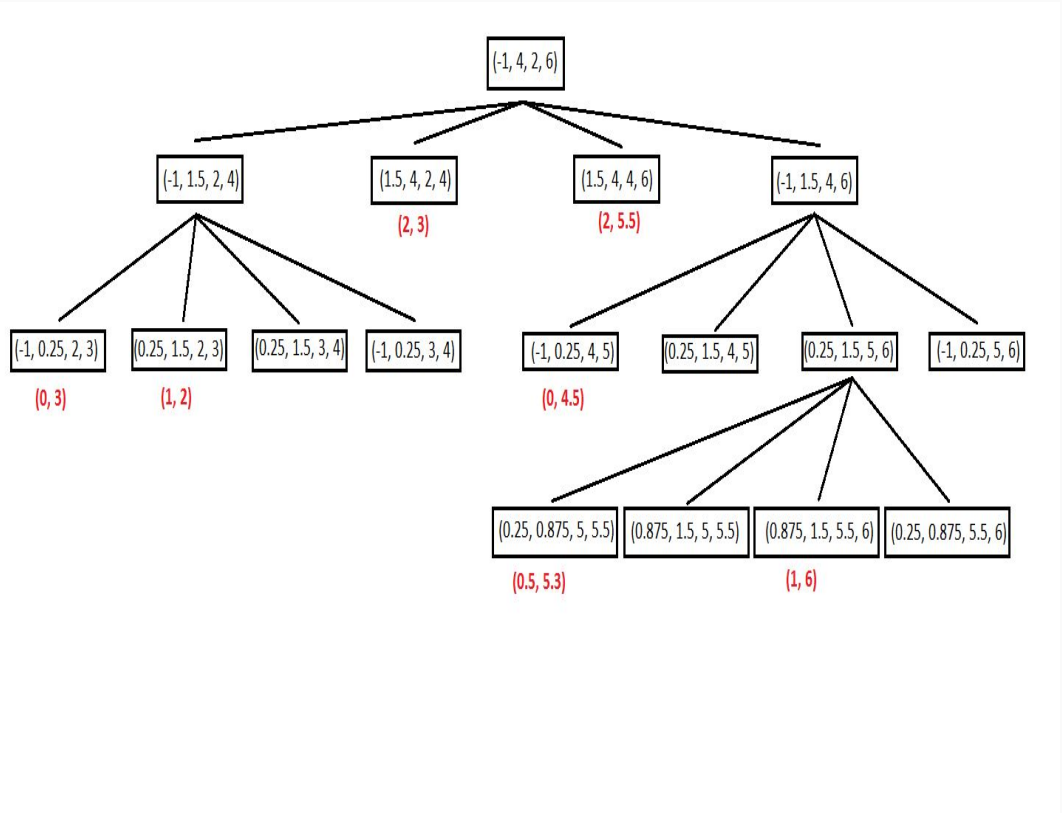

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4


(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(1.5, 4, 4, 6)
(-1, 1.5, 4, 6)
(2, 3)
(2, 5.5)
(-1, 0.25, 2, 3)
(0.25, 1.5, 2, 3)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0, 3)
(1, 2)
(0, 4.5)
(0.25, 0.875, 5, 5.5)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(0.25, 0.875, 5.5, 6)
(0.5, 5.3)
(1, 6)

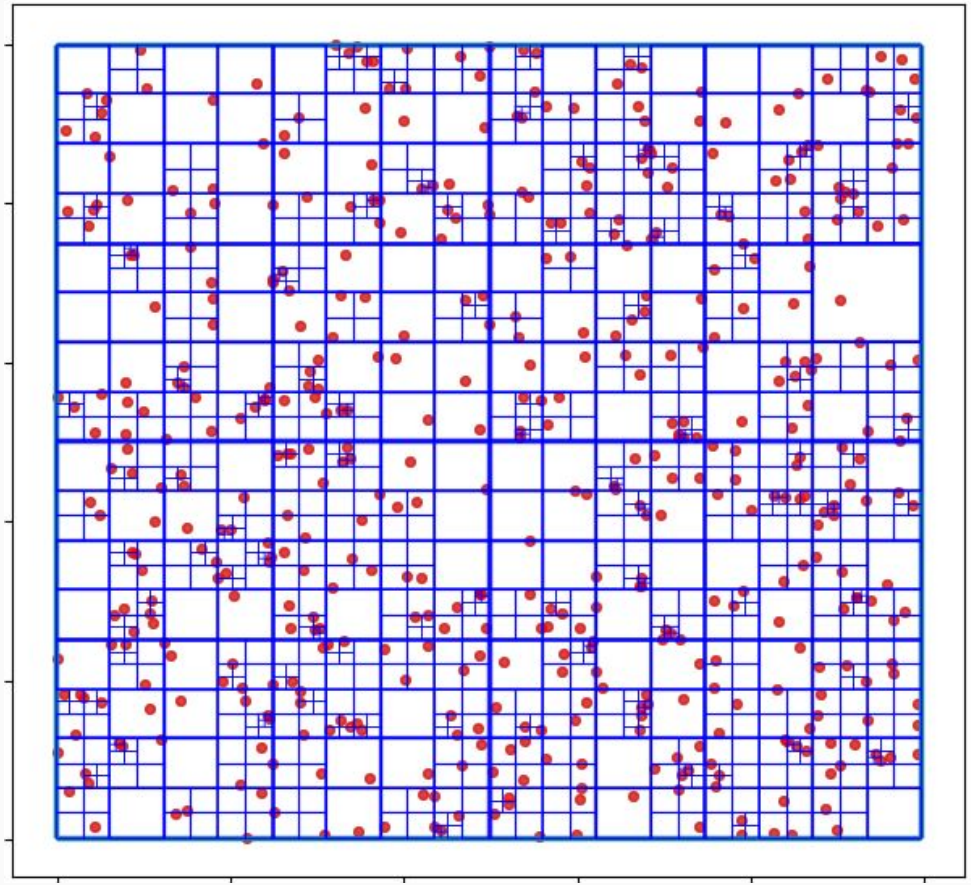

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4


(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0, 4.5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0.25, 0.875, 5, 5.5)
(0.5, 5.3)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(1, 6)
(0.25, 0.875, 5.5, 6)

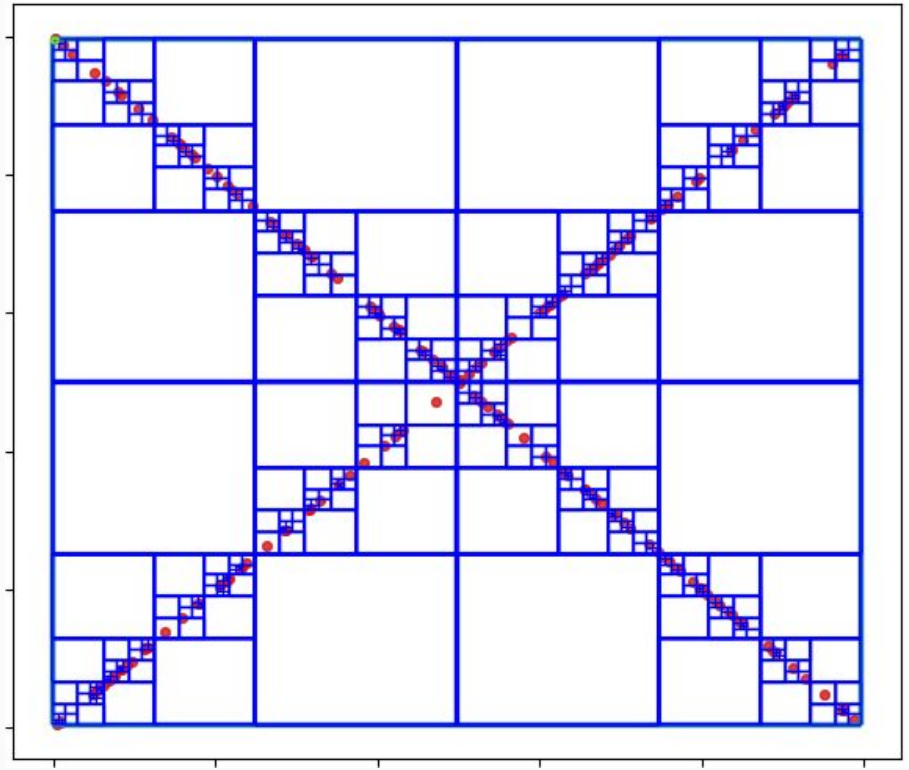

6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
−1
0
1
2
3
4

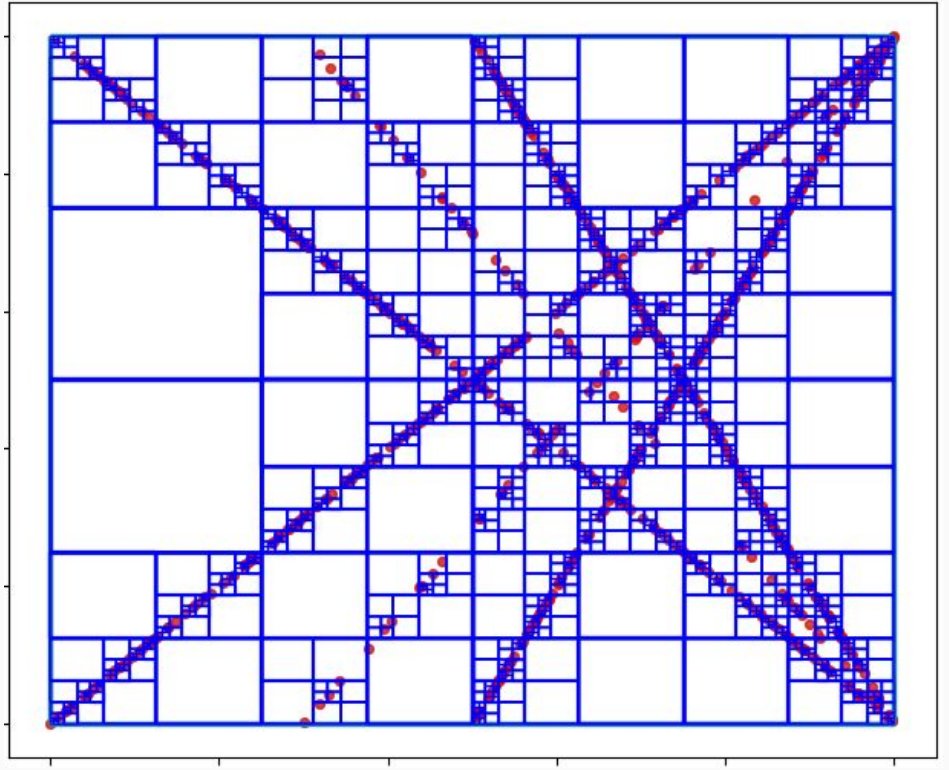

(-1, 4, 2, 6)
(-1, 1.5, 2, 4)
(1.5, 4, 2, 4)
(2, 3)
(1.5, 4, 4, 6)
(2, 5.5)
(-1, 1.5, 4, 6)
(-1, 0.25, 2, 3)
(0, 3)
(0.25, 1.5, 2, 3)
(1, 2)
(0.25, 1.5, 3, 4)
(-1, 0.25, 3, 4)
(-1, 0.25, 4, 5)
(0, 4.5)
(0.25, 1.5, 4, 5)
(0.25, 1.5, 5, 6)
(-1, 0.25, 5, 6)
(0.25, 0.875, 5, 5.5)
(0.5, 5.3)
(0.875, 1.5, 5, 5.5)
(0.875, 1.5, 5.5, 6)
(1, 6)
(0.25, 0.875, 5.5, 6)

Quad Tree - losowy zbiór punktów wewnątrz kwadratu



Quad Tree - losowy zbiór punktów na odcinkach





Quad Tree - konstrukcja pseudokod

```
constructQuadTree(P):
    utwórz root
    oblicz dla P minimalny kwadrat obejmujący
        wszystkie punkty i przypisz go jako obszar root
    splitQTNode(P, root)
```

Quad Tree - konstrukcja pseudokod

```
splitQTNodes(P, v):
    if P ma nie więcej niż nodeCapacity elemntów:
        v.points = P
        v.isLeaf = True
    else:
        v.isLeaf = False
        skonstruuj węzły SW, SE, NE, NW i oblicz dla nich
            odpowiednie obszary
            przypisz je jako dzieci węzła v
        podziel P względem obszarów SW, SE, NE, NW do
            P_SW, P_SE, P_NE, P_NW
        splitQTNodes(P_SW, SW)
        splitQTNodes(P_SE, SE)
        splitQTNodes(P_NE, NE)
        splitQTNodes(P_NW, NW)
```

Złożoność czasowa konstrukcji

O((d+1) * n), gdzie

- n - liczba punktów
- d - wysokość drzewa

  d <= log(s/c) + 3/2, gdzie:

  - s długość boku kwadratu root-a,
  - c - najmniejsza odległość między punktami

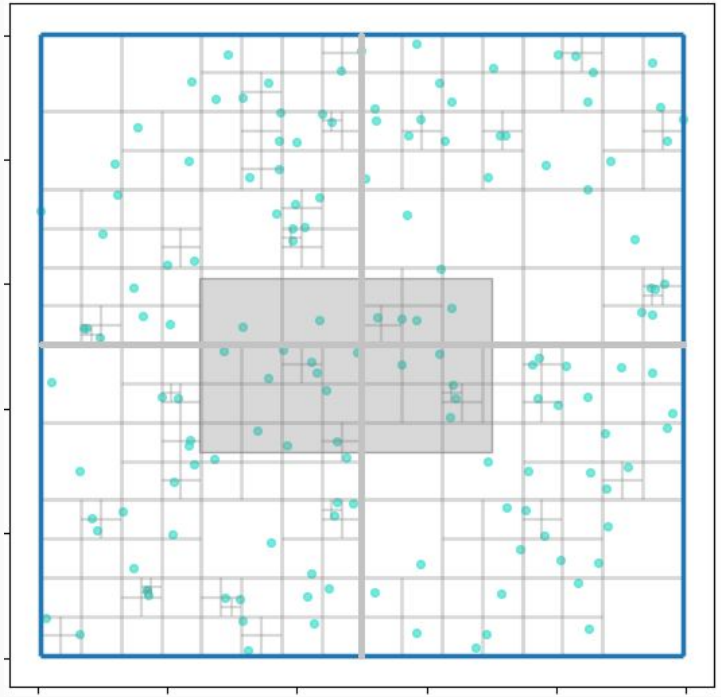
Na każdym poziomie drzewa algorytm musi rozdzielić co najwyżej **n** punktów do obszarów węzłów dzieci, poziomów jest **(d+1)**, więc czas konstrukcji to O((d+1)*n)

Przeszukiwanie ortogonalne

Dane:

- quad tree, drzewo przechowujące punkty
- queryRect, obszar prostokątny do przeszukania

Przeszukując drzewo analizujemy punkty lub dzieci tylko węzłów, których obszary przecinają się z queryRect



Przeszukiwanie - możliwe sytuację

- obszar węzła, w którym się znajdujemy nie przecina się z queryRect:
  - kończymy
- obszar węzła, w którym się znajdujemy zawiera się w queryRect:
  - raportujemy wszystkie punkty z tego obszaru
- obszar węzła, w którym się znajdujemy przecina się z queryRect:
  - jeśli węzeł wewnętrzny, sprawdzamy wszystkie dzieci węzła
  - jeśli liść, sprawdzamy zawieranie się punktów w queryRect

Przeszukiwanie - pseudokod

searchInRange(root, queryRect):

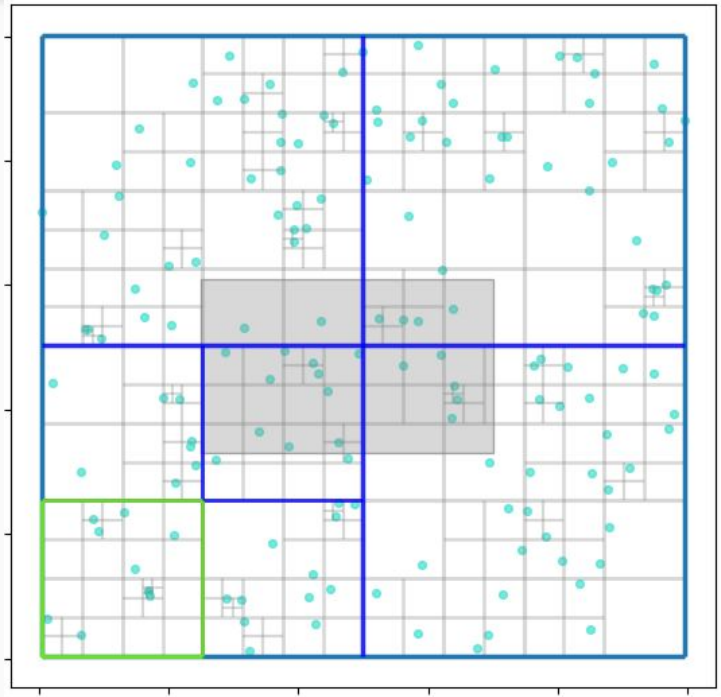
utwórz pustą listę znalezionych punktów **reportedPoints**

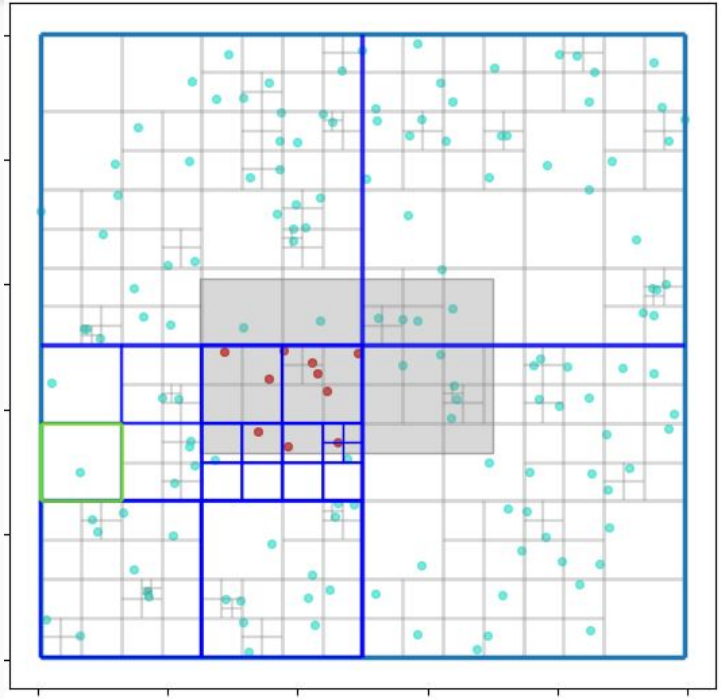
_search(root, reportedPoints, queryRect)

return **reportedPoints**

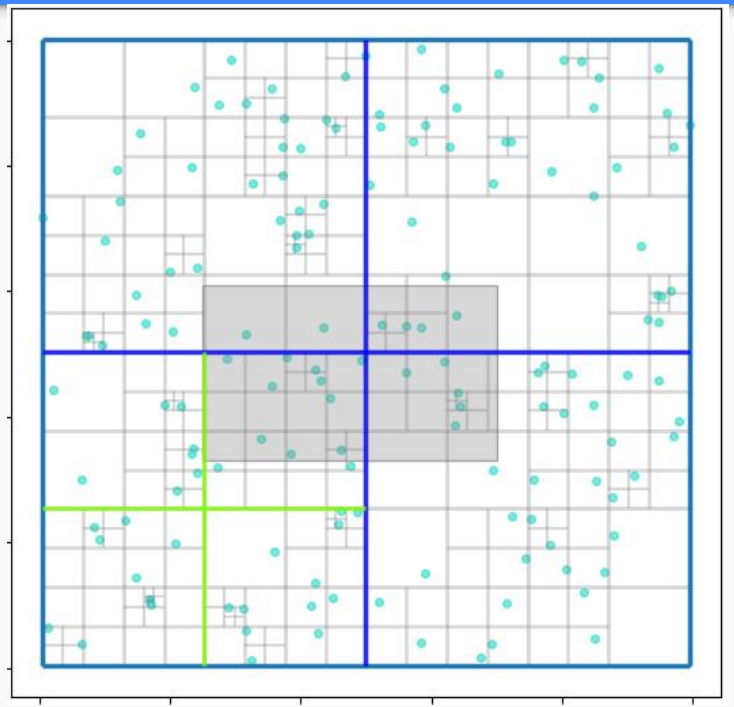
```
_search(v, reportedPoints, queryRect):
    if obszar v nie przecina się z queryRect:
        return
    else:
        if obszar v zawiera się w  queryRect:
            dodaj wszystkie punkty z tego obszaru do reportedPoints
        else:
            if v  jest liściem:
                dodaj do reportedPoints punkty z v zawartę w queryRange
            else:
                _search(v.SW, reportedPoints, queryRect)
                _search(v.SE, reportedPoints, queryRect)
                _search(v.NE, reportedPoints, queryRect)
                _search(v.NW, reportedPoints, queryRect)
```

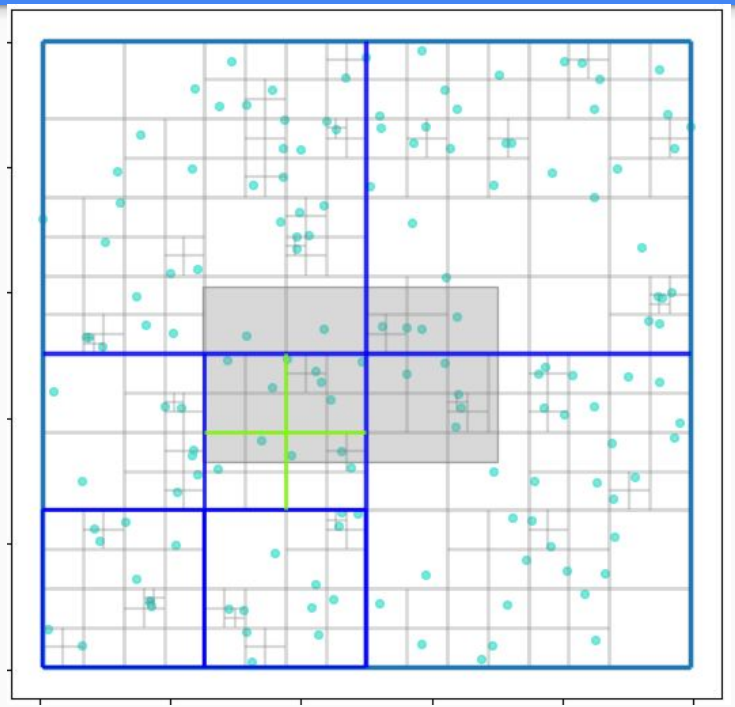
Przeszukiwanie - brak części wspólnej



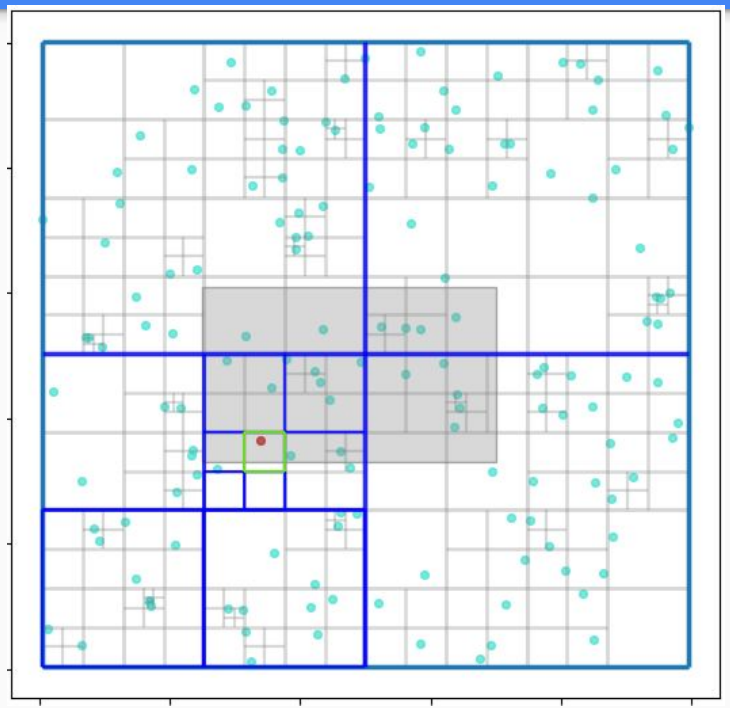


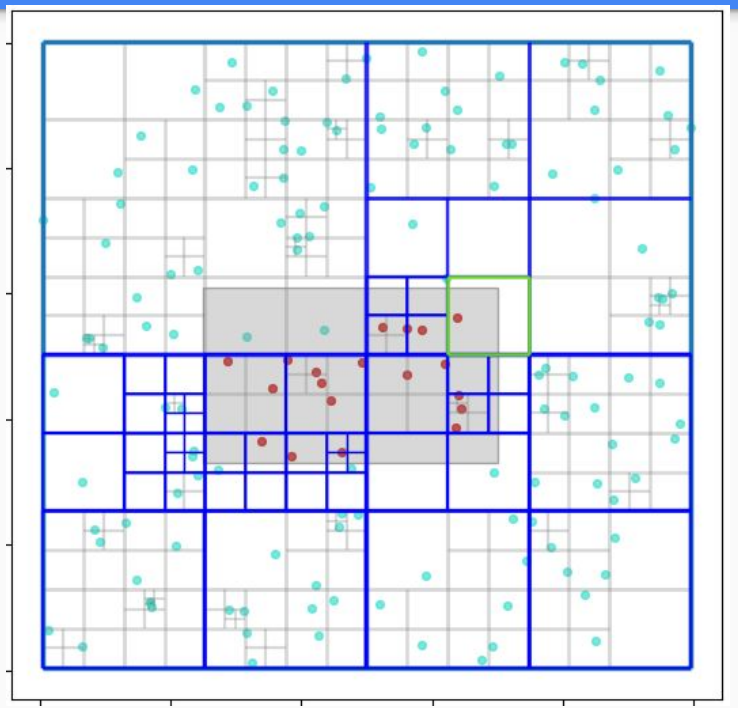
Przeszukiwanie - przecięcie



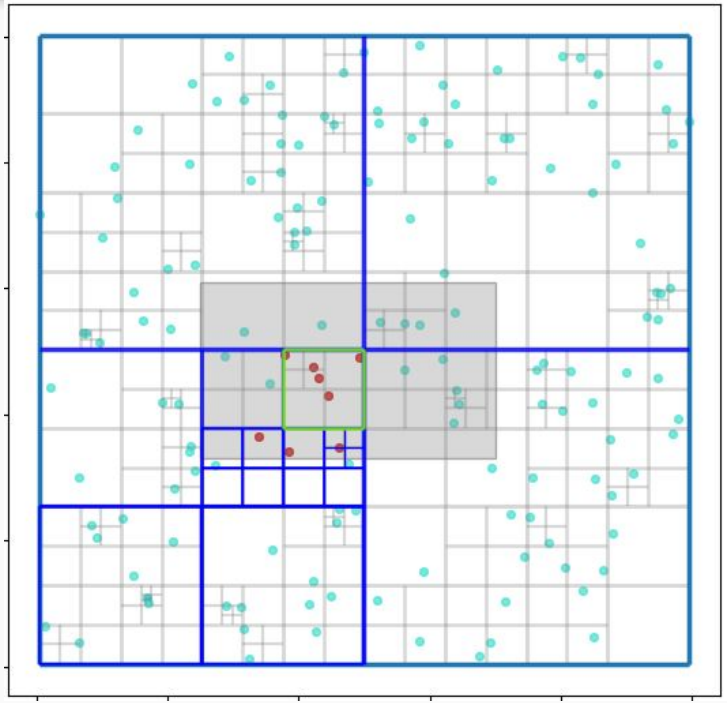


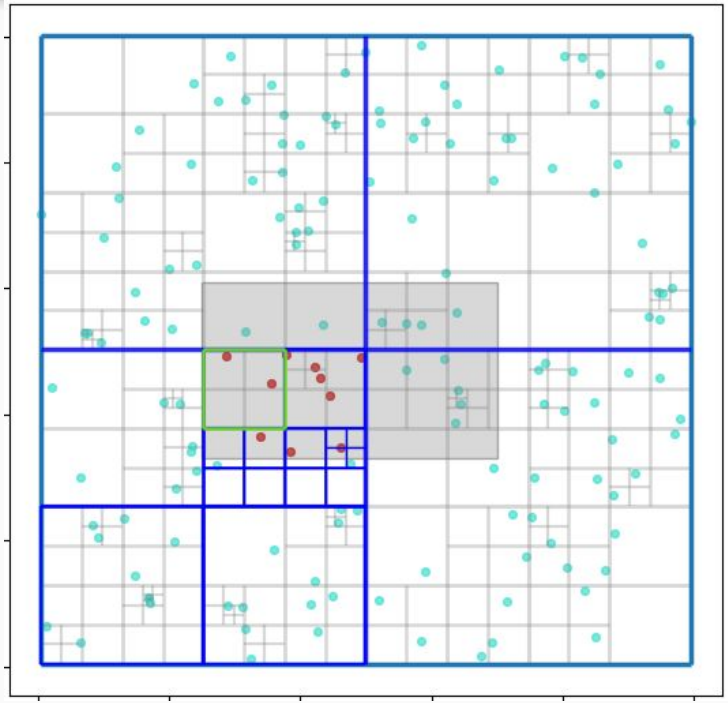
Przeszukiwanie - liść





Przeszukiwanie - zawieranie





Przeszukiwanie - złożoność czasowa

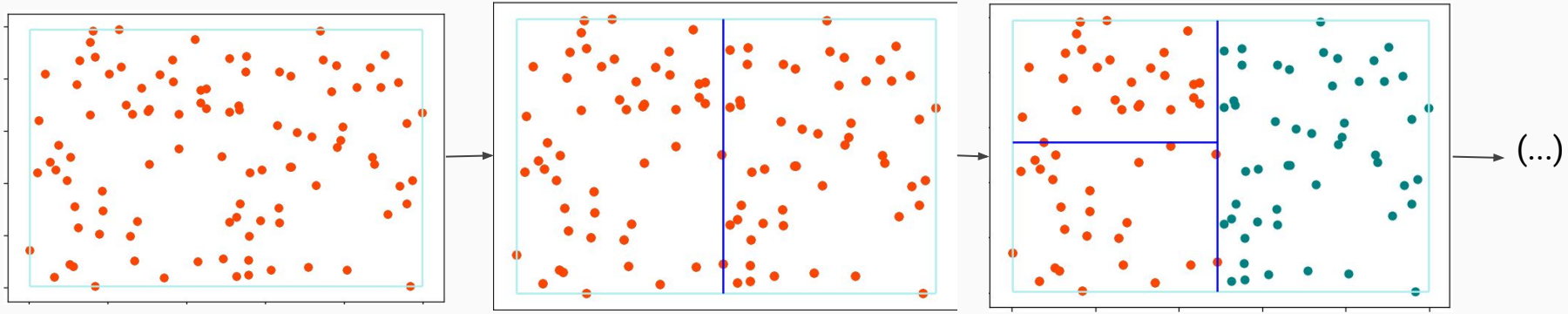
Przeszukiwanie jest silnie zależne od rozkładu punktów na płaszczyźnie:

- najszybsze jest dla rozkładu równomiernego
- jest tym bardziej wolna im większa jest koncentracja punktów wzdłuż linii - gwałtowny wzrost drzewa

Ograniczenie górne : O(n)

KD-Tree - wprowadzenie

Kolejnym z rozwiązań omawianego zagadnienia jest użycie struktury **KD-Tree**. Główną ideą tego rozwiązania jest dokonanie podziału punktów na połowy wg współrzędnej x i y naprzemiennie.



Jak zapisywać taką strukturę?

Ponieważ sposób konstrukcji drzewa jest rekurencyjny, najlepiej zapisywać wyniki w postaci drzewa. Składa się ono z:

Liści - zawierających pojedynczy punkt

Węzłów - zawierających współrzędną przecięcia oraz wskaźniki - lewy i prawy - na punkty o, odpowiednio, mniejszej i większej współrzędnej.

Węzły nie muszą przechowywać osi, względem której dokonano przecięcia - można to obliczyć na podstawie głębokości drzewa

KD-drzewo - ilustracja struktury

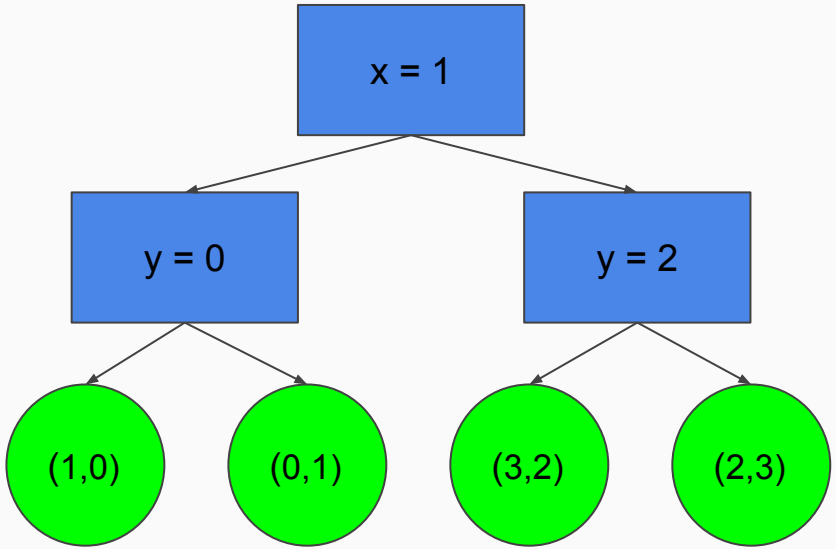
Dla zbioru punktów:

{(0,1), (1,0), (2,3), (3,2)}

Po lewej węzła - punkty mniejsze lub równe

Po prawej - większe

Jest to kwestia umowna



Jak wybrać punkt przecięcia?

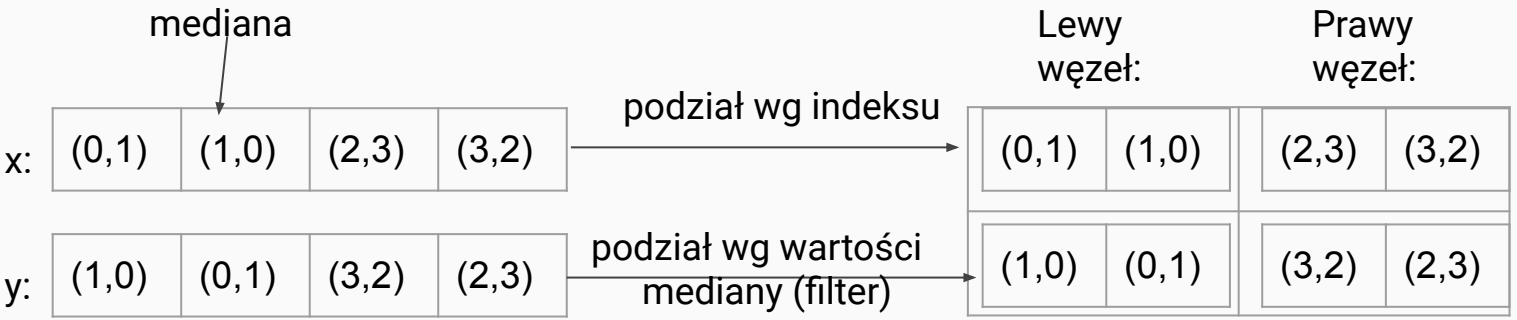
W celu otrzymania drzewa zrównoważonego, musimy dzielić punkty na równe części. W tym celu najlepiej jest się posłużyć medianą*, jednak sortowanie za każdym zejściem rekurencyjnym jest kosztowne obliczeniowo. Należy zatem użyć algorytmu znajdowania mediany (który jest dosyć skomplikowany) bądź posługiwać się dwoma posortowanymi tablicami - jedną wg współrzędnej x, druga wg y.

*Wzorując się na „Computional Geometry”, *Mark de Berg et al.*, przyjmujemy, że mediana parzystej liczby elementów to mniejsza z pośrednich liczb, lecz jest to kwestia umowna. Równie dobre są pozostałe definicje mediany, dopóki odpowiednio przypisujemy równy przecięciu element

Jak dokonać przecięcia?

Dla wcześniejszego przykładu (podział względem x):

1.Znajdujemy medianę 2. Dzielimy tablicę posort. wg x po indeksie mediany
3.Dzielimy tablicę posort. wg y po wartości współrzędnej x mediany



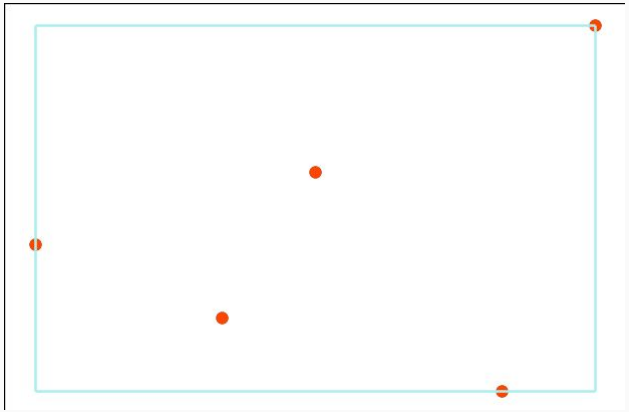
Budowa drzewa - pseudokod

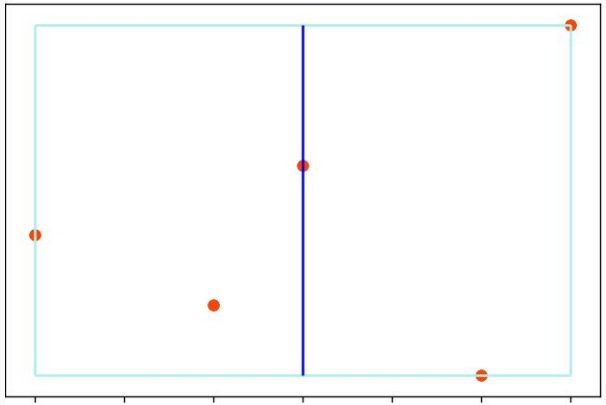
```
INIT_KDTREE(P, depth):
    if P zawiera 1 punkt:
        return liść z punktem z P
    if depth jest parzyste:
        podziel P pionową linią line na zbiory P1 i P2
    else:
        podziel P poziomą linią line na zbiory P1 i P2
    v1 = INIT_KDTREE(P1, depth+1)
    v2 = INIT_KDTREE(P2, depth+1)
    v = węzeł zapamiętujący linię line oraz z synami v1 i v2
    return v
```

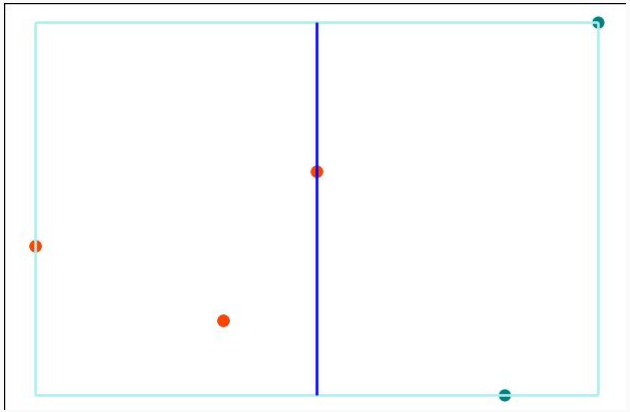
Wizualizacja na zbiorze punktów

Zbiór punktów: { (6,0), (7,5), (4,3), (1,2), (3,1) }

1. Sprawdzamy wielkość zbioru punktów, dzielimy go i schodzimy rekurencyjnie do powstałych dwóch zbiorów punktów

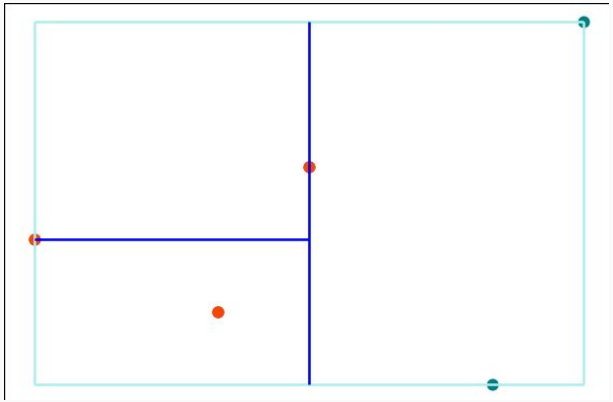


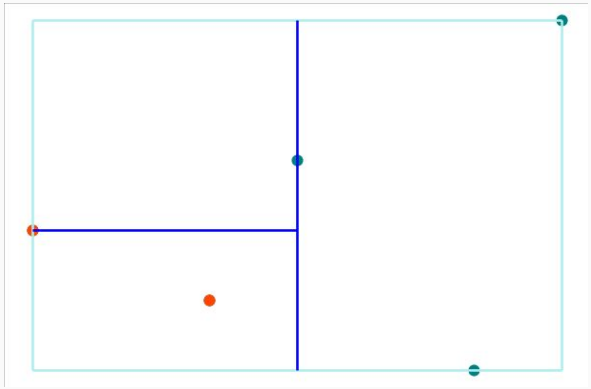




Wizualizacja na zbiorze punktów

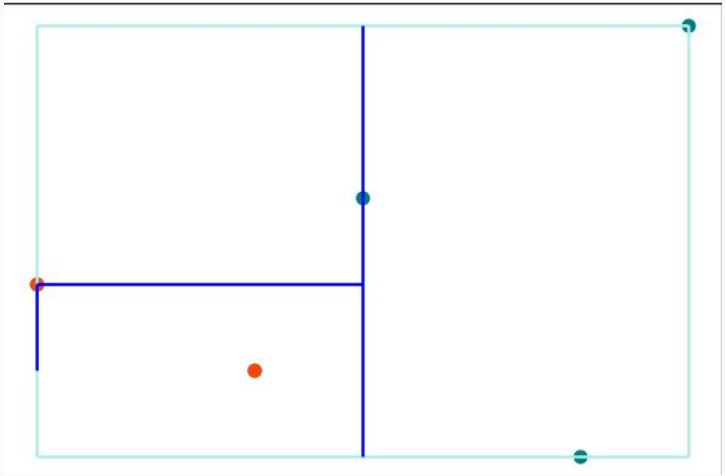
2. Ponownie sprawdzamy wielkość zbioru punktów, dzielimy go i schodzimy rekurencyjnie do powstałych dwóch zbiorów punktów

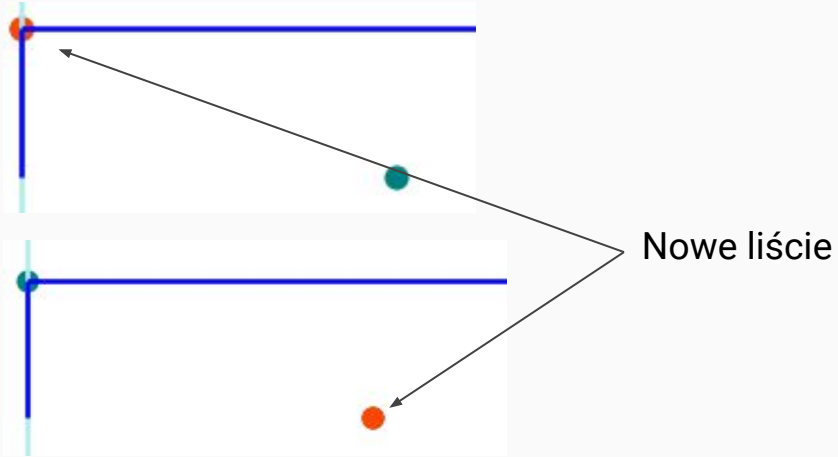




Wizualizacja na zbiorze punktów

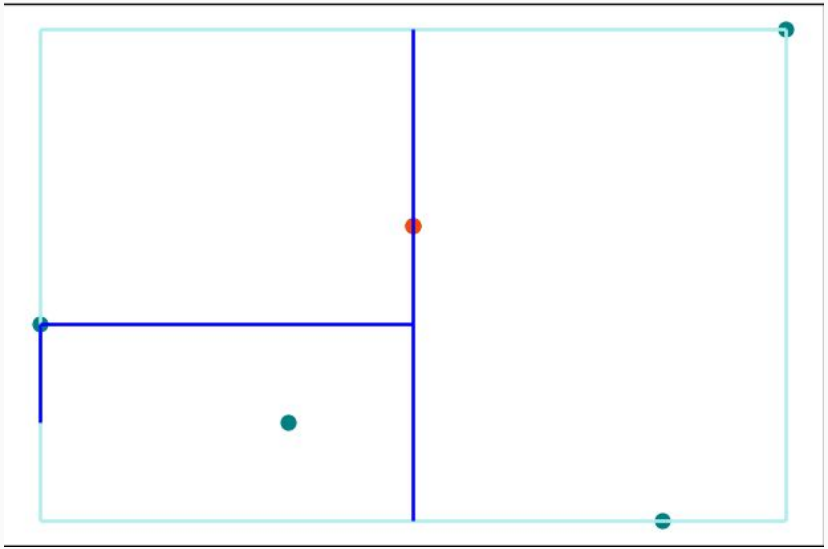
3. Ponownie sprawdzamy wielkość zbioru punktów, dzielimy go i schodzimy rekurencyjnie do powstałych dwóch zbiorów punktów. W tym wypadku zbiory dzieci węzła są jednoelementowe, więc tworzymy z nich liście i cofamy się rekurencyjnie.





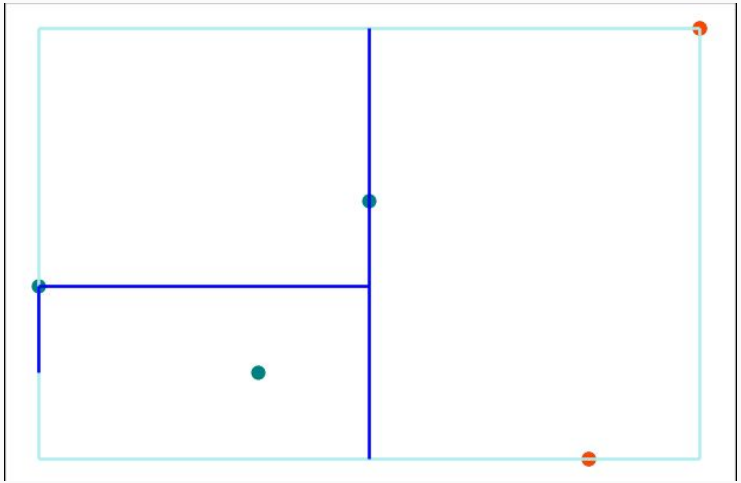
Wizualizacja na zbiorze punktów

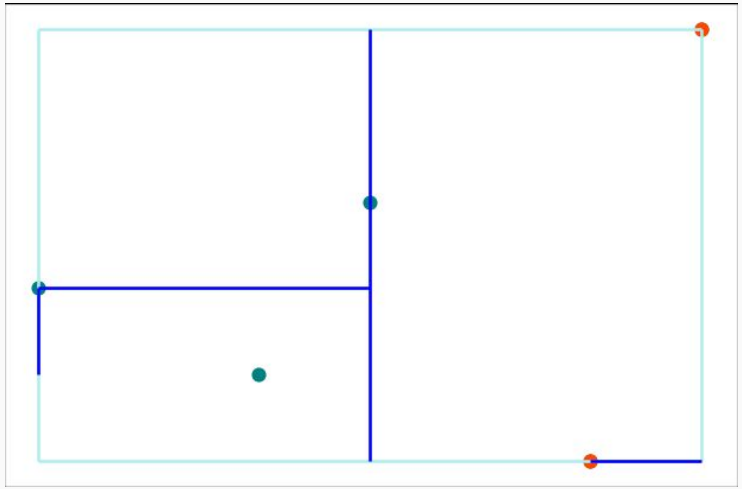
4. Wracamy rekurencyjnie do górnego zbioru. Zawiera jeden punkt, tworzymy liść



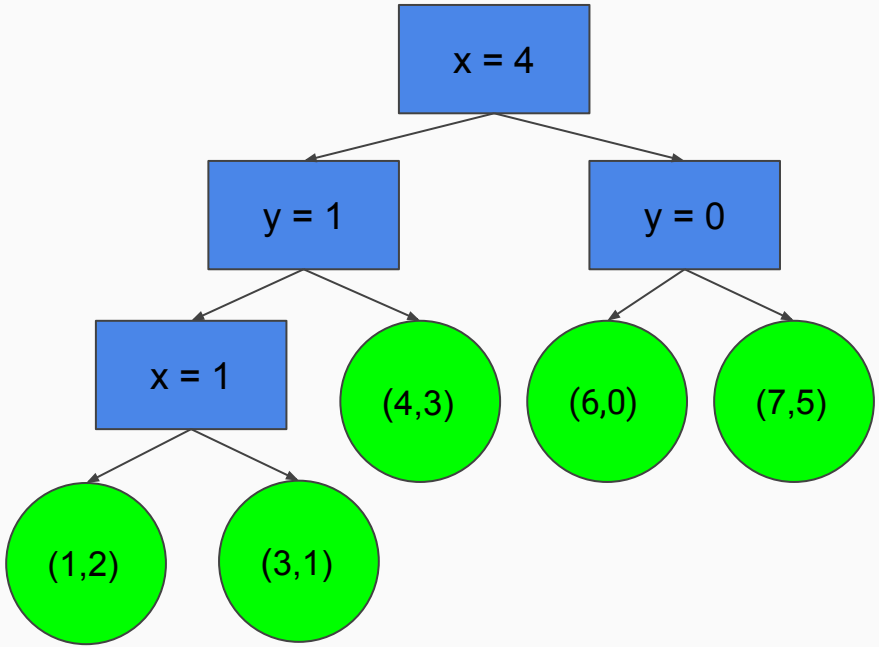
Wizualizacja na zbiorze punktów

5. Wracamy rekurencyjnie do prawego zbioru. Zawiera dwa punkty, tworzymy węzeł, którego dziećmi są liście zawierające poszczególne punkty





Struktura drzewa dla pokazanego przykładu



Złożoność obliczeniowa konstrukcji KD-Tree

Oznaczmy T(n) jako czas konstrukcji KD-Drzewa dla zbioru n punktów. Konstrukcja drzewa jest funkcją rekurencyjną, której krok bazowy (stworzenie liścia) wykonywany jest w czasie O(1), a krok rekurencyjny w czasie O(n) + 2*T(n/2) (podzielenie tablicy + dwa wywołania rekurencyjne dla synów węzła)

Rozwiązawszy powyższe równanie rekurencyjne* otrzymujemy wynik

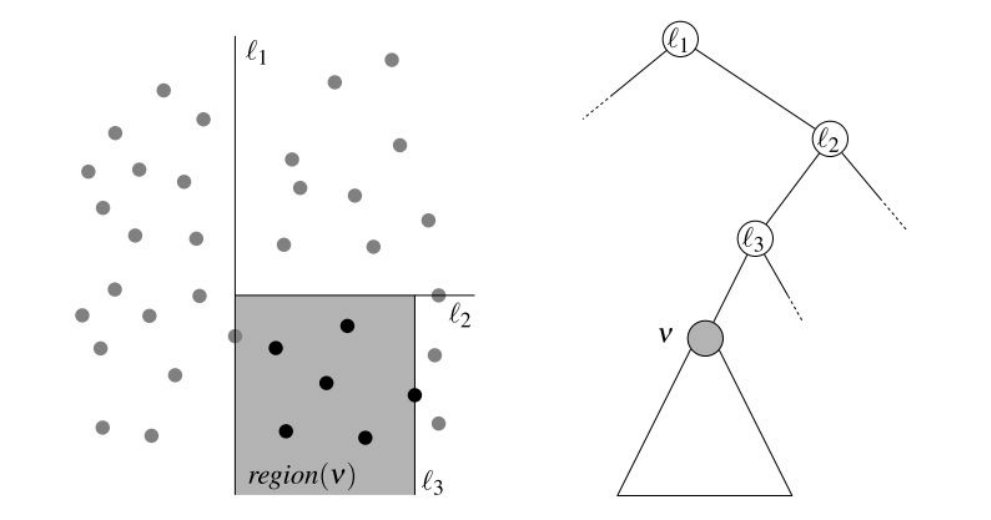
$$O(n \log n)$$

*Np. za pomocą twierdzenia o rekurencji uniwersalnej (https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms))

Przeszukiwanie

Należy pamiętać, że każdy węzeł odpowiada za grupę punktów, których współrzędne możemy ograniczyć za pomocą linii znajdujących się w rodzicach węzła.

Gdy przeszukujemy drzewo **wchodzimy tylko do węzłów, które przecinają się z przeszukiwanym obszarem**



Ilustracja ze strony 102 „Computional Geometry”, Mark de Berg et al., zamieszczona w celach edukacyjnych

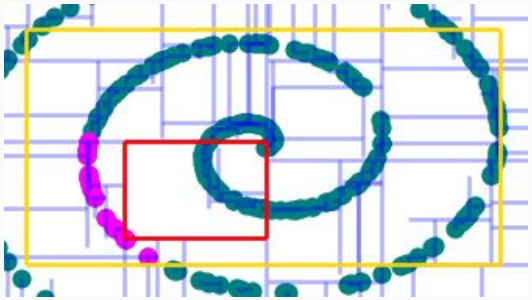
Możliwe scenariusze

Mając na uwadze poprzednie informacje, możemy wyodrębnić trzy interesujące nas sytuacje w trakcie procedury przeszukiwania:

1. Przetwarzany element to węzeł, a obszar węzła zawiera się w obszarze przeszukiwania - wtedy zwracamy wszystkie punkty w węźle
2. Przetwarzany element to węzeł, a obszar węzła i przeszukiwania mają niezerową część wspólną (ale pierwszy nie zawiera się w drugim) - wówczas kontynuujemy procedurę dla przecinającego się węzła
3. Przetwarzany element to liść - sprawdzamy, czy punkt zawiera się w obszarze przeszukiwania i jeśli tak, to dodajemy go do wyniku

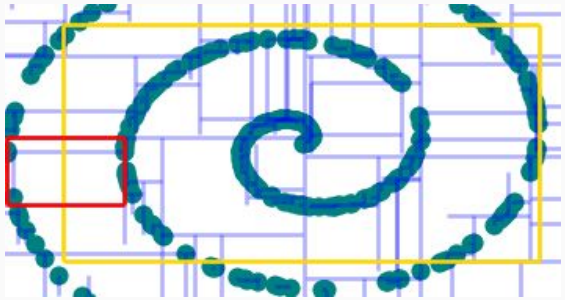
Ilustracja możliwych scenariuszy

Ad. 1 - zawieranie się



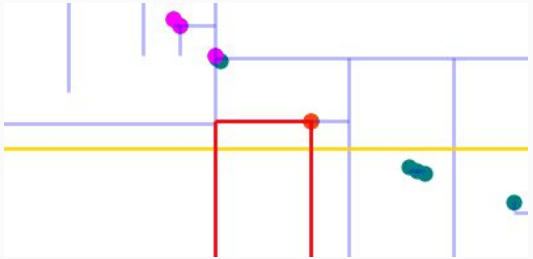
Dodajemy wszystkie punkty wewnątrz

Ad. 2 - przecinanie się



Wchodzimy w węzły-dzieci, który obszar się przecina

Ad. 3 - liść



Sprawdzamy, czy punkt liścia zawiera się w poszukiwanym przedziale i jeśli tak, dodajemy go

Pseudokod przeszukiwania

```
SEARCH_KDTREE(v, R):
if v jest liściem:
    if punkt z v zawiera się w R:
        return v.point

if obszar v.left zawiera się w R:
    REPORT_SUBTREE(v.left, R)
elif obszar v.left przecina R:
    SEARCH_KDTREE(v.left, R)

if obszar v.right zawiera się w R:
    REPORT_SUBTREE(v.right, R)
elif obszar v.right przecina R:
    SEARCH_KDTREE(v.right, R)
```

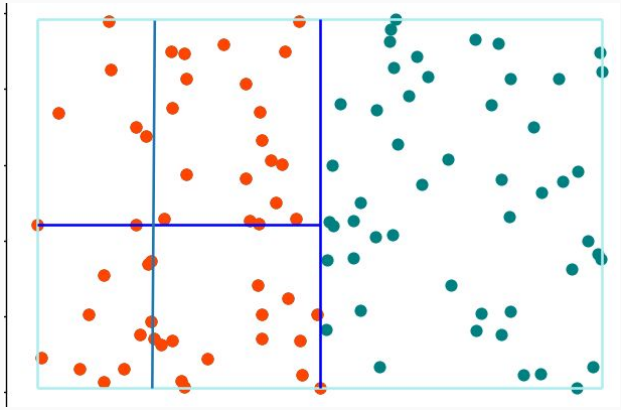
Złożoność obliczeniowa przeszukiwania KD-Tree

Przypomnijmy, że wchodzimy w tyle węzłów, ile jest przecięć obszaru przeszukiwania z obszarami węzłów. Jak zatem obliczyć liczbę przecięć?

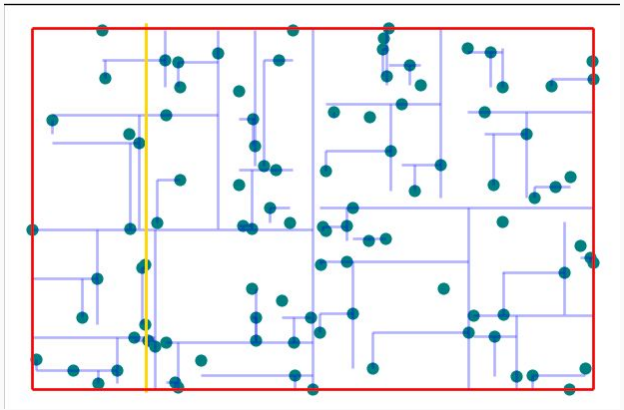
Niech Q(n) będzie liczbą przecięć wertykalną linią l w KD-Tree zawierającym n punktów, którego korzeń zawiera przecięcie wertykalne. Linia l przecina albo lewy, albo prawy obszar, lecz nie oba. Ta sama linia przecina dwoje z czterech obszarów prawnuków węzła, które mają O(n/4) punktów. W prawnukach sytuacja jest analogiczna, więc możemy zapisać zależność rekurencyjną:

$$Q(n) = 2 + 2*Q(n/4) \text{ dla } n>1$$

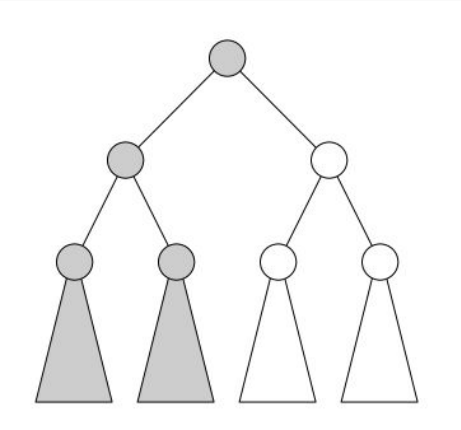
Złożoność obliczeniowa przeszukiwania KD-Tree - ilustracja



Przecięcie 2 z 4 wnuków



Rekurencyjność problemu



Schemat na drzewie

Analogiczne rozumowanie można przeprowadzić dla linii horyzontalnej

Złożoność obliczeniowa przeszukiwania KD-Tree c.d.

Co ważne, dla n = 1 (czyli liścia) liczba przecięć to co najwyżej 1, zatem

$$Q(1) = O(1)$$

Rozwiązując równanie rekurencyjne możemy łatwo obliczyć, że

$$Q(n) = O(\sqrt{n})$$

Dodając na koniec fakt, że gdy obszar przeszukiwania zawiera k punktów, musimy zwrócić je wszystkie, otrzymujemy, że złożoność przeszukiwania to:

$$O(\sqrt{n}) + O(k) = \mathbf{O(\sqrt{n} + k)}$$

Bibliografia/przydatne linki

Bibliografia:
„Computional Geometry”, *Mark de Berg, Otfried Cheng, Marc van Kreveld, Mark Overmars*
Wykłady z Algorytmów Geometrycznych dr Głut

Linki:
https://github.com/pkrucz00/projektGeometryczne.git - link do programu
https://en.wikipedia.org/wiki/Quadtree