Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Maciej Mularski

Nr albumu: 1154810

Struktury danych dla problemu wyszukiwania zakresowego

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2021

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Chcę podziękować wszystkim, którzy udzielili mi wsparcia w trakcie tworzenia tej pracy, a w szczególności Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za nieocenioną pomoc w analizie przedstawionych rozwiązań, oraz za zaangażowanie przy dopracowywaniu szczegółów tej pracy.

Streszczenie

Wyszukiwanie zakresowe polega na przeszukiwaniu wstępnie przygotowanych danych, aby pozyskać wszystkie elementy znajdujące się w szukanym zakresie. W pracy rozważane jest wyszukiwanie zakresowe punktów w jednym i w wielu wymiarach, gdzie zakresem jest odcinek na osi liczbowej lub wielościan w n-wymiarowej przestrzeni, którego boki są równoległe do osi układu współrzędnych.

Często wykorzystywaną strukturą danych dla wydajnego wyszukiwania zakresowego jest drzewo zakresowe. Struktura ta ma formę zbalansowanego drzewa binarnego, które przechowuje dane w liściach, a w każdym węźle, który nie jest liściem, przechowuje maksymalną wartość swojego lewego poddrzewa. W pracy przedstawiono szczegółowy opis problemu wyszukiwania zakresowego, oraz implementację drzewa zakresowego w języku Python.

Stworzono dwie implementacje drzewa zakresowego na bazie zmodyfikowanego drzewa AVL. Pierwsza implementacja jest przeznaczona do przechowywania i wyszukiwania punktów w jednym wymiarze. Druga implementacja działa dla punktów w n-wymiarach, przy czym powyżej jednego wymiaru duplikaty są zliczane, a nie przechowywane osobno. Struktury te zostały przetestowane pod względem poprawności i rzeczywistej wydajności, dzięki czemu mogą być używane w programach wymagających wydajnego wyszukiwania zakresowego. Kod działa w Pythonie 2.7 i Pythonie 3.

Słowa kluczowe: wyszukiwanie zakresowe, drzewo przedziałowe, drzewo zakresowe, drzewo AVL

English title: Data structures for the range searching problem

Abstract

Range searching is finding objects in a range, where a set of objects is preprocessed in advanve. In this work objects are points in one or more dimensions, and the range is an interval on the number line or a polyhedron in an n-dimensional space, where the query consists of intervals in each of those dimensions.

A data structure often used in efficient range searching is a range tree. This structure has the form of a balanced binary tree, which stores data in leaves and in every node that is not a leaf stores the maximum value of its left subtree. This work describes the range searching problem and Python implementation of a dynamic range tree, which is based on a modified AVL tree.

Two implementations of range trees are created. The first one is focused on storage and acting on points in one dimension. The second one works for points in an n-dimensional space and duplicates are counted, not stored, in order to reduce the size of the tree structure. These structures have been tested for correctness and real efficiency. It can be used in other programs where efficient range searching is required. The code can be used in both Python 2.7 and Python 3.

Keywords: range searching, interval tree, range tree, AVL tree

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest problem wyszukania zakresowego, oraz struktury danych pomagające rozwiązać ten problem. Wyszukiwanie zakresowe polega na przetworzeniu danego zbioru danych w taki sposób, aby szybko odnaleźć elementy leżące w różnych zakresach przeszukiwania. Dokładny opis różnych odmian tego problemu podamy w rozdziale 2.

Wyszukiwanie zakresowe jest jednym z fundamentalnych zagadnień w geometrii obliczeniowej. Problem wyszukiwania zakresów i struktury danych, które go rozwiązują, są podstawowym tematem geometrii obliczeniowej. Zastosowania problemu pojawiają się w obszarach obejmujących systemy informacji geograficznej (GIS) i projektowania wspomaganego komputerowo (CAD), oraz baz danych (SQL).

W tej pracy skupimy się na wyszukiwaniu zakresowym w jednym, a następnie w wielu wymiarach, gdzie wyszukiwane będą punkty leżące odpowiednio w danym odcinku na osi liczbowej, lub w wielościanie w n-wymiarowej przestrzeni, przy czym boki takiej figury są równoległe do osi układu współrzędnych. Jest to prostopadłe wyszukiwanie zakresowe (ang. *orthogonal range searching*).

Przykładowe struktury danych wykorzystywane do wyszukiwania zakresowego są następujące [2]:

— Drzewo przedziałowe (ang. *interval tree*) [3]. Stosowane do odcinków na osi liczbowej.
— Drzewo zakresowe (ang. *range tree*) [4]. Stosowane do punktów w d wymiarach.
— Drzewo k-d (ang. *k-d tree*) [5]. Stosowane do punktów w d wymiarach.
— Drzewo czwórkowe (ang. *quadtree*) [6]. Stosowane do punktów na płaszczyźnie.
— Lista z przeskokami (ang. *skip list*) [7]. Może być stosowana do liczb w jednym wymiarze.

Podstawowe informacje na temat problemu przeszukiwania zakresowego zaczerpnięto z klasycznej książki Cormena, Leisersona, Rivesta i Steina [21], a także z innych źródeł.

1.1. Cel pracy

Celem pracy jest analiza i implementacja w języku Python [1] wybranych struktur danych wykorzystywanych do wyszukiwania zakresowego. Język Python umożliwia łatwą prezentację przygotowanego kodu źródłowego, jak również uruchamianie go w różnych środowiskach.

Podczas przygotowywania pracy tworzenie kodu źródłowego było oparte tylko na założeniach teoretycznych, ponieważ nie została znaleziona gotowa implementacja tychże założeń.

Przygotowana implementacja drzewa zakresowego umożliwia przeszukanie zbioru danych w czasie $O(\log^d n + k)$, gdzie n oznacza liczbę elementów w strukturze, a k liczbę elementów znalezionych podczas wyszukiwania. Struktura ta umożliwia dynamiczne przeszukiwanie zbioru, oznacza to, że dane mogą zmienić się w czasie istnienia takiego drzewa.

1.2. Struktura pracy

Praca została podzielona na trzy główne części.

Część pierwsza (rozdział 2) to wprowadzenie do tematyki przeszukiwania zakresowego. Przedstawione zostaną podstawowe pojęcia potrzebne do zrozumienia problemu przeszukiwania zakresowego, jak i również opis samego problemu. Dodatkowo zostaną przedstawione struktury, które umożliwiają rozwiązanie takiego problemu.

Część drugą (rozdział 3) stanowi szczegółowy opis dostępnych metod, które są zaimplementowane w strukturze.

W ostatniej części (rozdział 4) znajdują się opisy wraz z samymi algorytmami, które zostały wykorzystane w strukturze umożliwiającej przeszukiwanie zakresowe.

W dodatkach zamieszczone zostały wyniki testów algorytmów struktur, które umożliwiają wydajne przeszukiwanie zakresu w zbiorze.

2. Problem wyszukiwania zakresowego

Rozdział zawiera podstawowe definicje i twierdzenia, które są niezbędne do zrozumienia problemu wyszukiwania zakresowego.

2.1. Struktury danych

Struktura danych jest to sposób organizacji danych, najczęściej w pamięci komputera, dla polepszenia wydajności algorytmów [10]. Zaliczamy do nich takie struktury jak:

— tablica,
— lista powiązana (ang. *linked list*),
— sterta (ang. *heap*),
— słownik (ang. *dictionary*),
— drzewo (ang. *tree*).

Ponadto można do nich zaliczyć jednostki koncepcyjne takie jak nazwisko, bądź adres zamieszkania.

2.2. Przetwarzanie danych

Przetwarzanie danych polega głównie na "gromadzeniu i przetwarzaniu elementów danych w celu uzyskania znaczących informacji" [23]. Przykładem przetwarzania danych może być obliczanie orbit satelitów, prognozowanie pogody, wyszukiwanie lokalizacji na mapie. Cykl przetwarzania informacji w kontekście komputerów i przetwarzania komputerowego ma cztery etapy: wprowadzanie, przetwarzanie, wyprowadzanie i przechowywanie [24].

2.3. Wyszukiwanie zakresowe

Definicja: *Wyszukiwanie zakresowe* głównie polega na odpowiednim przetworzeniu zbioru S w celu określenia, które elementy ze zbioru S znajdują się w obszarze zdefiniowanym przy pomocy zapytania, zwanym inaczej *zakresem*, który spełnia odpowiednie kryteria [13].

Na przykład, jeśli S jest bazą danych pracowników, która zawiera kolumny odpowiadające wynagrodzeniu oraz wieku, tymże problemem jest znalezienie tych pracowników, którzy mają wiek pomiędzy a_1 i a_2, oraz ich zarobki znajdują się pomiędzy s_1 i s_2.

2.3.1. Rodzaje problemów wyszukiwania zakresowego

Istnieje kilka odmian problemu wyszukiwania zakresowego, a dla różnych odmian mogą być konieczne różne struktury danych [16]. Aby uzyskać efektywne rozwiązanie, należy sprecyzować kilka aspektów problemu [2]:

(a) **Typy obiektów:** Algorytmy zależą od tego, czy S składa się z punktów, linii, odcinków prostych, ramek, wielokątów, itd. Najprostszymi i najczęściej badanymi obiektami do wyszukiwania są punkty.
(b) **Typy zakresów:** Zakresy zapytań muszą również pochodzić z wcześniej określonego zbioru. Niektóre dobrze zbadane zbiory zakresów i nazwy odpowiednich problemów to prostokąty wyrównane do osi (przeszukiwanie zakresów ortogonalnych), sympleksy, półprzestrzenie, sfery lub okręgi.
(c) **Typy zapytań:** Jeśli potrzeba znaleźć listę wszystkich obiektów, które znajdują się w zakresie, taki problem nazywa się *raportowaniem zakresu*, a zapytanie nazywa się *zapytaniem raportującym*. Czasami jednak wymagana jest tylko liczba obiektów, które znajdują się w zakresie. W takim przypadku problem ten nazywa się *zliczaniem zakresów*, a zapytanie jest określane jako *zapytanie zliczające*. *Zapytanie o pustość zbioru* natomiast informuje, czy istnieje co najmniej jeden obiekt, który znajduje się w zakresie.
(d) **Wyszukiwanie w zakresie dynamicznym lub statycznym:** W ustawieniu statycznym zbiór S jest znany z góry, natomiast w ustawieniu dynamicznym obiekty mogą być wstawiane lub usuwane między zapytaniami.
(e) **Przeszukiwanie zakresu offline:** Zarówno zestaw obiektów, jak i cały zestaw zapytań są znane z góry.

2.3.2. Wyszukiwanie zakresu ortogonalnego

W przypadku wyszukiwania zakresu ortogonalnego zbiór S składa się z n punktów w d wymiarach, a zapytanie składa się z przedziałów w każdym z tych wymiarów. W związku z tym zapytanie składa się z wielowymiarowego prostokąta wyrównanego do osi. Zakładając, że wielkość rezultatu będzie równa k, Jon Bentley użył drzewa $k-d$, aby osiągnąć (w notacji Big O) $O(n)$ przestrzeni i $O(n^{1-1/d}+k)$ czas zapytania [17]. Bentley zaproponował również użycie drzewa zakresowego, co poprawiło czas zapytań do $O(\log^d n + k)$, ale zwiększyło przestrzeń do $O(n \log^{d-1} n)$ [18].

2.3.3. Wyszukiwanie zakresu dynamicznego

Podczas wyszukiwania w zakresie statycznym zbiór S jest znany z góry i nie można go zmienić. W zakresie dynamicznym wstawianie oraz usuwanie punktów jest możliwe pomiędzy zapytaniami. W przyrostowej wersji problemu dozwolone są tylko wstawienia, podczas gdy wersja dekrementalna umożliwia tylko usuwanie. W przypadku ortogonalnym Mehlhorn i Näher stworzyli strukturę danych do wyszukiwania zakresu dynamicznego, która wykorzystuje *dynamic fractional cascading*, aby osiągnąć $O(n \log n)$ przestrzeni oraz $O(\log n \log \log n + k)$ czas zapytania [19]. Zarówno przyrostowe,

jak i malejące wersje problemu można rozwiązać za pomocą czasu zapytania $O(\log n + k)$.

2.4. Drzewo AVL

Definicja: Drzewo AVL jest to zrównoważone binarne drzewo poszukiwań (BST), w którym wysokość lewego i prawego poddrzewa każdego węzła różni się co najwyżej o jeden [8]. Ta różnica wysokości nazywa się *współczynnikiem wyważenia*.

Sposób balansowania: Jeśli zostanie wykryte, że współczynnik wyważenia będzie miał wartość większą niż 1 lub mniejszą niż -1, drzewo traci własność AVL i potrzebne jest przebalansowanie go poprzez rotację węzłów.

Istnieją cztery rodzaje rotacji - dwie pojedyncze oraz dwie podwójne:

— Rotacja pojedyncza LL (ang. *Left-Left*),
— Rotacja pojedyncza RR (ang. *Right-Right*),
— Rotacja podwójna RL (ang. *Right-Left*),
— Rotacja podwójna LR (ang. *Left-Right*).

Oznaczenia RR, LL, RL i LR określają sposób połączenia węzłów przed wykonaniem rotacji. Jeśli węzły łączą się prawymi krawędziami, mamy do czynienia z przypadkiem RR, jeśli tylko lewymi, to jest to przypadek LL. Mieszane połączenia są oznaczane jako LR i RL [9].

Rotacje pojedyńcze przebiegają w taki sposób, że węzeł środkowy staje się rodzicem dla pozostałych węzłów. Rotacje podwójne przebiegają w taki sposób, że doprowadzamy do sytuacji, gdzie wymagana jest rotacja pojedyncza i wykonujemy ją analogicznie.

Wykorzystanie w pracy: W pracy został wykorzystany sposób balansowania i założenia drzewa AVL przy tworzeniu implementacji algorytmu dodwania i usuwania elementów z drzewa zakresowego.

2.5. Drzewo zakresowe

Opis: W tej pracy skupiamy się na implementacji wyszukiwania punktów w jednym oraz wielu wymiarach. Do tego celu zostanie użyte drzewo zakresowe. Drzewo zakresowe jest to uporządkowane drzewo binarne zawierające listę punktów. Umożliwia to efektywne raportowanie wszystkich punktów w danym zakresie i jest zwykle używane w dwóch lub wyższych wymiarach. Drzewa zakresowe zostały wprowadzone przez Bentleya w 1979 roku [12].

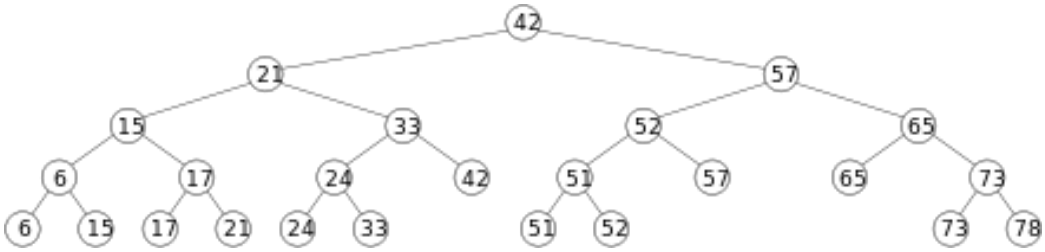
Złożoność: Drzewo zapewnia złożoność czasową dla wyszukań $O(\log^d n + k)$ oraz złożoność pamięciową $O(n \log^{d-1} n)$, gdzie n jest liczbą punktów przechowywanych w drzewie, d to wymiar każdego punktu, a k to liczba punktów zgłoszonych przez dane zapytanie.

Bernard Chazelle ulepszył je do złożoności czasowej $O(\log^{d-1} n + k)$ i złożoność pamięciową $O(n(\frac{\log n}{\log \log n})^{d-1})$ [16].

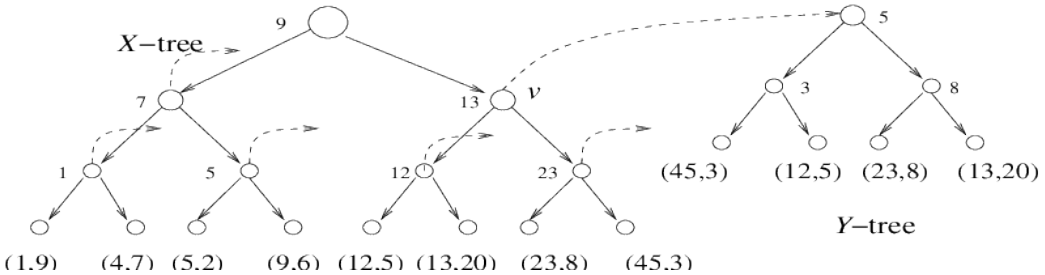
Struktura: Drzewo zakresowe na zbiorze punktów w jednym wymiarze jest zrównoważonym drzewem wyszukiwania binarnego na tych punktach o dodatkowych założeniach:

— Punkty przechowywane w drzewie są przechowywane w liściach drzewa,
— Każdy węzeł wewnętrzny przechowuje największą wartość zawartą w jego lewym poddrzewie.

Drzewo zakresowe na zbiorze punktów w wymiarach d jest rekurencyjnie zdefiniowanym wielopoziomowym drzewem wyszukiwania binarnego. Każdy poziom struktury danych jest binarnym drzewem wyszukiwania w jednym z wymiarów d. Pierwszy poziom to binarne drzewo wyszukiwania na pierwszej ze współrzędnych przechowywanych przez tą strukturę. Każdy wierzchołek v tego drzewa zawiera powiązaną strukturę, która jest $(d-1)$-wymiarowym drzewem zakresowym na ostatnich $(d-1)$-rzędnych punktów przechowywanych w poddrzewie v [4].



Rysunek 2.1. Drzewo zakresowe dla jednego wymiaru ze strony [4].



Rysunek 2.2. Przykładowe drzewo zakresowe dla dwóch wymiarów. Przykład pochodzi z publikacji [26].

Konstrukcja: 1-wymiarowe drzewo zakresowe na zbiorze n punktów jest drzewem wyszukiwania binarnego, które można skonstruować w czasie $O(n \log n)$. Drzewa zakresowe o wyższych wymiarach są konstruowane rekurencyjnie, konstruując zrównoważone drzewo wyszukiwania binarnego na pierwszej współrzędnej ze zbioru punktów, a następnie, dla każdego wierzchołka v tego

drzewa, konstruując $(d-1)$-wymiarowe drzewo zakresowe na pozostałych współrzędnych zawartych w poddrzewie wierzchołka v. Konstruowanie drzewa zakresowego w ten sposób wymaga $O(n \log^d n)$ czasu. Ten czas budowy można poprawić dla 2-wymiarowych drzew zakresowych do $O(n \log n)$ [20].

Niech S będzie zbiorem n dwuwymiarowych punktów. Jeśli S zawiera tylko jeden punkt, zwracany jest liść zawierający ten punkt. W przeciwnym razie konstruowana jest powiązana struktura ze zbiorem S, która jest jednowymiarowym drzewem zakresowym przechowującym współrzędne y punktów ze zbioru S. Niech x_m będzie medianą współrzędnych x tych punktów. Niech S_L będzie zbiorem punktów ze współrzędną x mniejszą lub równą x_m i niech S_R będzie zbiorem punktów o współrzędnej x większej niż x_m. Rekurencyjnie konstruowane jest v_L, które jest dwuwymiarowym drzewem zakresów dla S_L oraz v_R, które jest dwuwymiarowym drzewem zakresowym dla S_R. Następnie tworzony jest wierzchołek v, którego lewy potomek to v_L, a prawy to v_R. Jeśli posortujemy punkty według współrzędnych y na początku algorytmu i zachowamy tę kolejność podczas dzielenia punktów według współrzędnej x, możemy skonstruować powiązane struktury każdego poddrzewa w czasie liniowym. Skraca to czas konstruowania dwuwymiarowego drzewa zakresów do $O(n \log n)$, a także skraca czas konstruowania d-wymiarowego drzewa zakresowego do $O(n \log^{d-1} n)$ [12].

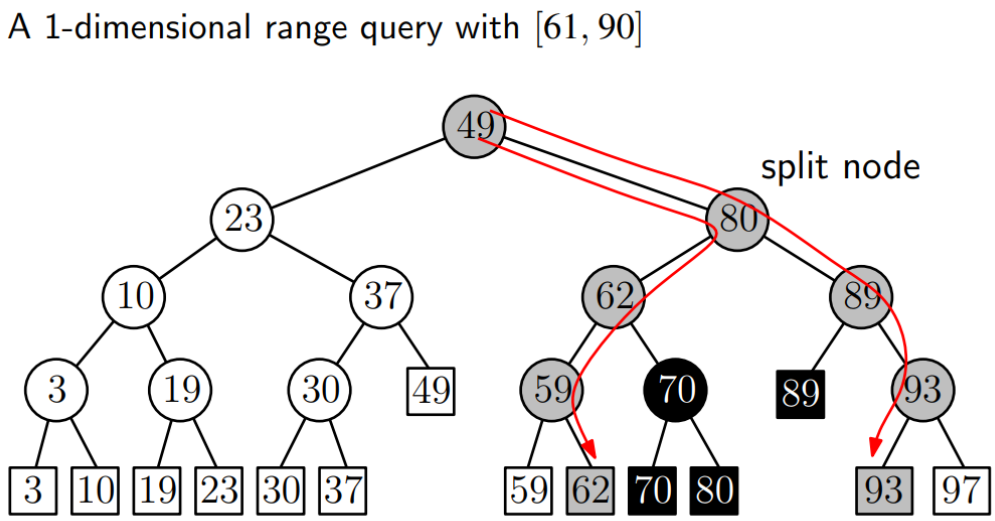
Zapytanie o zakres: Zapytanie o zakres w drzewie zakresowym zwraca zestaw punktów, które znajdują się w przekazanym w zapytaniu przedziale. Aby znaleźć punkty leżące w przedziale $[x_1, x_2]$, zaczynamy od wyszukania granic przedziału x_1 i x_2. W którymś wierzchołku drzewa ścieżki wyszukiwania do x_1 i x_2 zaczną się rozchodzić. Niech v_{split} będzie ostatnim wspólnym wierzchołkiem, który łączy te dwie ścieżki wyszukiwania. Dla każdego wierzchołka v na ścieżce od v_{split} do x_1, jeśli wartość przechowywana w wierzchołku v jest większa, niż wartość przechowywana w wierzchołku x_1, sprawdzany jest każdy punkt w prawym poddrzewie wierzchołka v. Jeśli v jest liściem tego drzewa, element przez niego przechowywany jest dodawany do zestawu, o ile taki element znajduje się wewnątrz przedziału $[x_1, x_2]$. Podobnie jest dla wszystkich punktów wzdłuż ścieżki od v_{split} do x_2, przechowywanych w lewych poddrzewach wierzchołków z wartościami mniejszymi niż x_2, które są liściami i znajdują się wewnątrz przedziału $[x_1, x_2]$ [22].

Ponieważ drzewo zakresowego jest zrównoważonym drzewem binarnym, ścieżki wyszukiwania do x_1 i x_2 mają długość $O(\log n)$. Przejście przez wszystkie punkty przechowywane w poddrzewie wybranego wierzchołka może się odbyć w czasie liniowym przy użyciu dowolnego algorytmu przechodzenia po drzewie. Wynika z tego, że czas wykonania zapytania o zakres wynosi $O(\log n + k)$, gdzie k jest liczbą punktów znajdujących się w interwale zapytania, a n liczbą elementów przechowywanych w drzewie.

Zapytania o zakres w d-wymiarach są podobne. Zamiast sprawdzać wszystkie punkty przechowywane w poddrzewach wykonuje się zapytanie o zakres w $(d-1)$-wymiarowym drzewie powiązanym z każdym poddrzewem wyższego wymiaru. Ostatecznie zostanie wykonane 1-wymiarowe zapytanie o zakres i zostaną zwrócone odpowiednie punkty. Ponieważ kwerenda d-wymiarowa składa się z $O(\log n)$ $(d-1)$-wymiarowych zapytań o zakres, wynika z tego,

że czas potrzebny do wykonania d-wymiarowego zapytania o zakres wynosi $O(\log^d n + k)$, gdzie k jest liczba punktów znajdujących się w interwale zapytania. Można ten czas zredukować do $O(\log^{d-1} n + k)$ przy użyciu metody *fractional cascading* [20].

W tejże metodzie dane dla ostatniego wymiaru są przechowywane w posortowanej tablicy, osobnej dla każdego węzła. Każda wartość przechowuje dwa wskaźniki, jeden dla wartości z tablicy lewego potomka węzła, oraz odpowiednio drugi dla wartości z tablicy prawego potomka węzła. Obie te tablice zawierają się w tablicy przechowywanej przez rodzica tych węzłów [14]. Jednakże metoda ta wymaga zaawansowanego użycia wskaźników dla danych przechowywanych w ostatnim wymiarze, przez co najlepiej sprawdzi się ona dla struktur statycznych, które po utworzeniu nie będą zmieniały swojej struktury.



Rysunek 2.3. Przykładowe działanie algorytmu przeszukiwania zakresowego dla drzewa zakresowego w jednym wymiarze. Przykład pochodzi ze slajdów [25].

3. Implementacja

Ten rozdział poświęcony został interfejsowi drzew zakresowch z punktu widzenia użytkownika. Przedstawimy poszczególne operacje dostępne dla tej implementacji oraz zasady ich użytkowania.

3.1. Drzewo zakresowe jednowymiarowe

W tej sekcji zostanie przedstawione drzewo zakresowe, które można wykorzystać wyłącznie do przeszukiwania punktów w jednym wymiarze. Jest ono zoptymalizowane pod tą liczbę wymiarów.

3.1.1. Importowanie

Import klasy RangeTree() możliwy jest na kilka sposobów. Zalecanym ze względu na prostotę późniejszego użycia jest:

Listing 3.1. Importowanie drzewa zakresowego.

```
>>> from rangetreesAVL import RangeTree
```

3.1.2. Tworzenie nowego drzewa

Drzewo można utworzyć na kilka różnych sposobów. Jako pierwszy argument należy podać listę z elementami, które będą dodane do drzewa. Jako drugi argument należy podać czy podana lista jest posortowana rosnąco. Domyślna wartość False oznacza, że podana lista nie jest posortowana, a wtedy konstruktor wykona sortowanie. Można również nie podawać żadnych argumentów, wtedy zostanie utworzone puste drzewo. Jako argumenty drzewo może przyjmować typy liczbowe z porządkiem (**int**, **float**).

Listing 3.2. Sposoby tworzenia drzewa

```
>>> rangetree = RangeTree([42, 21, 57, 15, 33, 52, 65, 6, 17, 24, 51, 73, 78], False)
>>> rangetree2 = RangeTree()
>>> rangetree3 = RangeTree([21, 10.4])
```

3.1.3. Dodawanie nowego elementu do drzewa

W celu dodania nowego elementu do drzewa należy użyć metody insert(). Przyjmuje ona jeden argument, którym jest dodawana do drzewa wartość.

Listing 3.3. Dodawanie nowego elementu do drzewa jednowymiarowego.

```
>>> rangetree.insert(5)
```

3.1.4. Usuwanie elementu z drzewa

W celu usunięcia elementu z drzewa należy użyć metody delete(). Jej argumentem jest wartość elementu, który ma zostać usunięty.

Listing 3.4. Usuwanie elementu z drzewa jednowymiarowego.

```
>>> rangetree.delete(5)
```

3.1.5. Wyszukiwanie najmniejszego oraz największego elementu w drzewie

Interfejs klasy RangeTree() udostępnia narzędzia pozwalające w prosty i czytelny sposób odnaleźć najmniejszą oraz największą wartość w drzewie. Są to odpowiednio metody minimum_value() oraz maximum_value(), które nie przyjmują argumentów, a zwracają szukaną wartość.

Listing 3.5. Znajdowanie najmniejszego i największego elementu drzewa jednowymiarowego.

```
>>> rangetree.minimum_value()
6
>>> rangetree.maximum_value()
78
```

3.1.6. Wyszukiwanie liczby w drzewie

Aby sprawdzić, czy interesująca nas wartość znajduje się w drzewie, należy użyć metody find(). Jako argument podajemy wyszukiwaną wartość. Metoda zwraca węzeł z wartością, jeśli takowy istnieje, lub None, jeśli nie istnieje.

Listing 3.6. Wyszukiwanie liczby w drzewie jednowymiarowym.

```
>>> rangetree.find(5) # zwrocone None
>>> rangetree.find(21)
<nodesAVL.Node object at 0x7f6fd5154b50>
```

3.1.7. Wyszukiwanie liczb z podanego zakresu

Aby znaleźć wszystkie wartości z zakresu, który nas intersuje, należy użyć metody range_searching(). Funkcja przyjmuje dwa argumenty, są nimi początek oraz koniec zakresu, w którym użytkownik chce znaleźć wartości. Zwracana jest lista zawierająca wszystkie elementy drzewa z podanego zakresu.

Listing 3.7. Wyszukiwanie zakresowe w drzewie jednowymiarowym.

```
>>> rangetree.range_searching(0, 100)
[6, 15, 17, 21, 24, 33, 42, 51, 52, 57, 65, 73, 78]
>>> rangetree.range_searching(0, 1)
[]
>>> rangetree.range_searching(10, 22)
[15, 17, 21]
```

3.1.8. Wyświetlanie zawartości drzewa

Drzewo umożliwia wyświetlenie elementów, które są w nim przechowywane. W tym celu należy użyć metody print_tree(). Do konsoli zostanie wpisana zawartość drzewa w kolejności *preorder* oraz liczbę duplikatów dla danego punktu. Metoda nie przyjmuje żadnych argumentów.

Listing 3.8. Wyświetlanie zawartości drzewa jednowymiarowego.

```
>>> rangetree.print_tree()
>>> rangetree.print_tree()
point: 6 number of duplicates: 1
point: 15 number of duplicates: 1
point: 17 number of duplicates: 1
point: 21 number of duplicates: 1
point: 24 number of duplicates: 1
point: 33 number of duplicates: 1
point: 42 number of duplicates: 1
point: 51 number of duplicates: 1
point: 52 number of duplicates: 1
point: 57 number of duplicates: 1
point: 65 number of duplicates: 1
point: 73 number of duplicates: 1
point: 78 number of duplicates: 1
```

3.1.9. Pozyskiwanie zawartości drzewa

Drzewo posiada zaimplementowe metody, przy pomocy których możliwe jest otrzymanie listy wartości w kolejności *inorder*, *preorder*, oraz *postorder*. Metody nie przyjmują żadnych argumentów.

Listing 3.9. Metody pozyskiwania zawartości drzewa jednowymiarowego.

```
>>> rangetree.inorder_traversal()
[6, 6, 15, 15, 17, 17, 21, 21, 24, 24, 33, 33, 42, 42, 51, 51,
    52, 52, 57, 57, 65, 65, 73, 73, 78]
>>> rangetree.preorder_traversal()
[42, 21, 15, 6, 6, 15, 17, 17, 21, 33, 24, 24, 33, 42, 52, 51,
    51, 52, 65, 57, 57, 65, 73, 73, 78]
>>> rangetree.postorder_traversal()
[6, 15, 6, 17, 21, 17, 15, 24, 33, 24, 42, 33, 21, 51, 52, 51,
    57, 65, 57, 73, 78, 73, 65, 52, 42]
```

3.2. Drzewo zakresowe wielowymiarowe

W tej sekcji zostanie przedstawione drzewo zakresowe, które można wykorzystać do przeszukiwania punktów w wybranej przez użytkowanika liczbie wymiarów. Po wprowadzeniu pierwszej wartości do drzewa zostaje ustalone w jakiej liczbie wymiarów będą przechowywane dane. Nie ma możliwości zmiany tej wielkości, dopóki struktura będzie istniała.

3.2.1. Importowanie

Import klasy RangeTree() możliwy jest na kilka sposobów. Zalecanym ze względu na prostotę późniejszego użycia jest:

Listing 3.10. Importowanie drzewa zakresowego wielowymiarowego.

```
>>> from rangetree_x_dimension import RangeTree
```

3.2.2. Tworzenie nowego drzewa

Drzewo można utworzyć na kilka różnych sposobów. Jako argument należy podać listę z elementami, które będą dodane do drzewa. Punkty muszą być przechowywane w liście wewnątrz głównej listy, np. jeśli chcemy utworzyć drzewo przechowywyujące dane w dwóch wymiarach, należy podać przy tworzeniu listę o następującym schemacie [(x1, y1), (x2, y2), (x3, y3)]. Wymiar drzewa jest ustalony po wczytaniu pierwszego punktu. Można również nie podać żadnych argumentów, wtedy zostanie utworzone puste drzewo. Jako argumenty drzewo może przyjmować typy liczbowe z porządkiem (**int**, **float**).

Listing 3.11. Sposoby tworzenia drzewa wielowymiarowego

```
>>> rangetree = RangeTree([(1, 10), (2, 20), (3, 9)]) # drzewo dwuwymiarowe
>>> rangetree2 = RangeTree([(1, 10), (3, 30), (2, 20), (40, 4), (−2, 2)]) # drzewo dwuwymiarowe
>>> rangetree3 = RangeTree()
>>> rangetree4 = RangeTree([(21, 10.4, 5)]) # drzewo trzywymiarowe
```

3.2.3. Dodawanie nowego elementu do drzewa wielowymiarowego

W celu dodania nowego elementu do drzewa należy użyć metody insert(). Przyjmuje ona jeden argument, którym jest dodawana do drzewa wartość. Punkt musi zawierać się w wymiarze drzewa.

Listing 3.12. Dodawanie nowego elementu do drzewa.

```
>>> rangetree2.insert((5, 4))
```

3.2.4. Usuwanie elementu z drzewa

W celu usunięcia elementu z drzewa należy użyć metody delete(). Jej argumentem jest wartość elementu, który ma zostać usunięty. Metoda zwraca True jeśli wartość została usunięta, lub False, jeśli nie została usunięta.

Listing 3.13. Usuwanie elementu z drzewa.

```
>>> rangetree2.delete((6, 5))
False
>>> rangetree2.delete((5, 4))
True
```

3.2.5. Wyszukiwanie najmniejszego oraz największego elementu w drzewie

Interfejs klasy RangeTree() udostępnia narzędzia pozwalające w prosty i czytelny sposób odnaleźć najmniejszą oraz największą wartość w drzewie

dla wybranego wymiaru. Są to odpowiednio metody minimum_value() oraz maximum_value(), które przyjmują jako argument wymiar w jakim ma być szukana wartość najmniejsza oraz największa, a następnie ją zwracają.

Listing 3.14. Znajdowanie najmniejszego i największego elementu dla wybranego wymiaru w drzewie wielowymiarowym.

```
>>> rangetree2.minimum_value(2)
-2
>>> rangetree2.minimum_value(1)
2
>>> rangetree2.maximum_value(2)
40
>>> rangetree2.maximum_value(1)
30
```

3.2.6. Wyszukiwanie liczb z podanego zakresu

Aby znaleźć wszystkie wartości z zakresu, który nas intersuje, należy użyć metody range_searching(). Funkcja przyjmuje tyle argumentów, ile jest wymiarów w drzewie, są nimi początek oraz koniec zakresu dla każdego wymiaru, w którym użytkownik chce znaleźć wartości. Zwracana jest lista zawierająca wszystkie elementy drzewa z podanego zakresu.

Listing 3.15. Wyszukiwanie zakresowe.

```
>>> rangetree.range_searching((0, 10), (2, 3))
[]
>>> rangetree.range_searching((0, 10), (20, 30))
[(2, 20)]
>>> rangetree.range_searching((0, 10), (0, 100))
[(1, 10), (2, 20), (3, 9)]
```

3.2.7. Wyświetlanie zawartości drzewa

Drzewo umożliwia wyświetlenie elementów, które są w nim przechowywane. W tym celu należy użyć metody print_tree(). Wypisywanie zacznie się od drzewa najwyższego wymiaru, a następnie zostanie wypisane poddrzewo przechowywane przez każdy węzeł drzewa większego wymiaru. Funkcja będzie przechodziła przez węzły każdego drzewa z osobna w kolejności *preorder* oraz wypisze do konsoli zawartość węzła. Dodatkowo dla łodygi zostanie wypisany wymiar w jakim ona się znajduje, a dla liścia zostanie wypisana liczba duplikatów. Metoda nie przyjmuje żadnych argumentów.

Listing 3.16. Wyświetlanie zawartości drzewa wielowymiarowego.

```
>>> rangetree.print_tree()
stalk node value: 1 dimension: 2
stalk node value: 10 dimension: 1
stalk node value: 9 dimension: 1
leaf node point: (3, 9) number of duplicates: 1
leaf node point: (1, 10) number of duplicates: 1
leaf node point: (2, 20) number of duplicates: 1
```

```
stalk node value: 1 dimension: 2
leaf node point: (1, 10) number of duplicates: 1


stalk node value: 2 dimension: 2
stalk node value: 9 dimension: 1
leaf node point: (3, 9) number of duplicates: 1
leaf node point: (2, 20) number of duplicates: 1

stalk node value: 2 dimension: 2
leaf node point: (2, 20) number of duplicates: 1


stalk node value: 3 dimension: 2
leaf node point: (3, 9) number of duplicates: 1
```

3.2.8. Pozyskiwanie wszystkich punktów drzewa

Aby pozyskać wszystkie punkty, które są przechowywane w drzewie, należy użyć metody get_all_values(). Metoda nie przyjmuje żadnych argumentów.

Listing 3.17. Pozyskanie wszystkich punktów drzewa wielowymiarowego.

```
>>> rangetree.get_all_values()
[(1, 10), (2, 20), (3, 9)]
>>> rangetree2.get_all_values()
[(−2, 2), (1, 10), (2, 20), (3, 30), (40, 4)]
```

3.2.9. Pozyskiwanie zawartości drzewa

Drzewo posiada zaimplementowe metody, przy pomocy których możliwe jest otrzymanie dla drzewa najwyższego wymiaru listy wartości, oraz listy punktów przechowywanych w poddrzewach danego węzła. Przechodzenie przez węzły drzewa najwyższego wymiaru może odbywać się w kolejności *inorder*, *preorder*, oraz *postorder*. Metody nie przyjmują żadnych argumentów.

Listing 3.18. Metody pozyskiwania zawartości drzewa wielowymiarowego.

```
>>> rangetree.inorder_traversal()
[[1, [(1, 10)]], [1, [(1, 10), (2, 20)]], [2, [(2, 20)]], [2, [(3, 9), (1, 10), (2, 20)]], [3, [(3, 9)]]]
>>> rangetree.preorder_traversal()
[[2, [(3, 9), (1, 10), (2, 20)]], [1, [(1, 10), (2, 20)]], [1, [(1, 10)]], [2, [(2, 20)]], [3, [(3, 9)]]]
>>> rangetree.postorder_traversal()
[[1, [(1, 10)]], [2, [(2, 20)]], [1, [(1, 10), (2, 20)]], [3, [(3, 9)]], [2, [(3, 9), (1, 10), (2, 20)]]]
```

4. Algorytmy

W tej części znajdują się opisy algorytmów służących do tworzenia struktur, oraz poszukiwania w nich danych z zadanego zakresu. Sekcja zostanie podzielona na opisanie działania algorytmów dla drzewa jednowymiarowego oraz dla drzewa wielowymiarowego.

4.1. Drzewo zakresowe jednowymiarowe

W tej sekcji znajdują się opisy najważniejszych algorytmów używanych w drzewie jednowymiarowym.

4.1.1. Tworzenie drzewa

Algorytm ma za zadanie stworzenie drzewa zakresowego z przekazanych do niego danych.

Dane wejściowe: Wartości w formie listy, które zostaną dodane do drzewa.

Opis algorytmu: Algorytm dzieli przekazaną listę zawierającą posortowane punkty na dwie równe połowy S_l i S_r, tworzy węzeł v i nadaje mu wartość ostatniego punktu z listy S_l. Następnie powtarza ten krok dla lewego (v_l) jak i prawego (v_r) potomka utworzonego węzła przekazując im lewą (S_l) i odpowiednio prawą (S_r) część listy do momentu, gdy przekazywana lista będzie zawierała więcej niż jeden element.

Aby nie tworzyć nowej podlisty przy każdym wywoływaniu metody, przekazywana jest referencja do oryginalnej listy, oraz para indeksów wskazujących na początek i koniec potrzebnej podlisty.

Złożoność: $O(n)$.

Listing 4.1. Główna metoda do tworzenia drzewa jednowymiarowego.

```
1  def __build_tree(self, data, left_idx, right_idx, dict_data):
2      if right_idx - left_idx == 0:
3          node = Node(data[left_idx])
4      else:
5          xm = (right_idx - left_idx) // 2 + left_idx
6          new_left_idx = xm + 1
7          new_right_idx = xm
8          node = Node(data[xm])
9          if new_right_idx - left_idx >= 0:
10             node.left = self.__build_tree(data, left_idx, new_right_idx, dict_data)
11         if right_idx - new_left_idx >= 0:
12             node.right = self.__build_tree(data, new_left_idx, right_idx, dict_data)
```

```
13          node.height = 1 + max(self.__get_height(node.left),
14                                self.__get_height(node.right))
15          node.num_of_duplicates = dict_data[node.data]
16          return node
17
```

4.1.2. Dodawanie nowych elementów do drzewa

Algorytm ma za zadanie dodać do drzewa punkt podany w argumencie.

Dane wejściowe: Wartość dodawana do drzewa.

Opis algorytmu: Algorytm jest podzielony na dwie funkcje: główną - insert() oraz pomocniczą __insert_node(). Rozpoczynamy od sprawdzenia czy drzewo posiada już korzeń, jeśli takowego nie ma, to tworzymy go i dodajemy do niego wartość.

Jeśli korzeń istnieje, to przechodzimy do funkcji pomocniczej. W funkcji pomocniczej przy pomocy wywołań rekurencyjnych wchodzimy do odpowiednich gałęzi w drzewie w celu dotarcia do odpowiedniego liścia, do którego będzie dodana przekazana wartość.

Jako, że drzewo zakresowe zakłada, że w łodygach ma znajdować się największa wartość z lewego poddrzewa, dodawane są dwa punkty o takich samych wartościach, jeden jako łodyga i drugi jako liść będący lewym potomkiem tejże łodygi.

W dalszej kolejności następuje przebalansowanie drzewa, jeśli zajdzie taka potrzeba.

Uwagi: Funkcja pozwala na dodawanie duplikatów.

Złożoność: $O(\log n)$.

4.1.3. Usuwanie elementów z drzewa

Algorytm ma za zadanie usuąć z drzewa punkt podany w argumencie.

Dane wejściowe: Wartość usuwana z drzewa.

Opis algorytmu: Algorytm jest podzielony na dwie funkcje główną - delete() oraz pomocniczą __delete(). Po sprawdzeniu, czy został podany opowiedni format węzła, przechodzimy do funkcji pomocniczej.

W funkcji pomocniczej przy pomocy wywołań rekurencyjnych wchodzimy do odpowiednich gałęzi w drzewie w celu dotarcia do odpowiedniego liścia, który będzie usunięty. Jako, że drzewo zakresowe zakłada, że w łodygach ma znajdować się największa wartość z lewego poddrzewa, usuwane są dwa punkty o takich samych wartościach, jeden jako łodyga znaleziona podczas poszukiwania liścia, oraz drugi jako liść.

W dalszej kolejności następuje przebalansowanie drzewa, jeśli zajdzie taka potrzeba.

Uwagi: W przypadku, gdy nie ma takiej wartości w drzewie, jaką chcemy usunąć, funkcja nie usunie nic i po dotarciu do liścia zakończy się jej działanie. W przypadku zaistnienia duplikatów, zostanie zaaktualizowana informacja o ich liczbie.

Złożoność: $O(\log n)$.

4.1.4. Przeszukiwanie zakresowe na bazie range tree

Algorytm ma za zadanie znaleźć wszystkie punkty z podanego zakresu.

Dane wejściowe: Granice zakresu przeszukiwania x1 i x2.

Opis algorytmu: Algorytm jest podzielony na trzy etapy. Wpierw sprawdzana jest poprawność argumentów oraz czy drzewo nie jest puste.

Następnie w fukcji __find_x_split() szukana jest łodyga, która jest ostatnią wspólną łodygą przed rozdzieleniem się ścieżek.

Jeśli łodyga okaże się liściem, oraz wartość wewnątrz liścia będzie się znajdowała w podanym przedziale, jest ona zwracana oraz kończy się działanie algorytmu. W innym przypadku kończymy działanie zwracając pustą listę.

Jeśli łodyga nie jest liściem, wywoływane są dwie funkcje pomocnicze __range_searching_left() oraz __range_searching_right(), które to mają za zadanie dotrzeć do liści znajdujących się odowiednio na lewo oraz na prawo od łodygi rozdzielającej, których wartości znajdują się we wskazanym przedziale. Działają one na zasadzie, że jeśli wartość przechowywana w łodydze jest większa niż x1, dodawane są wszystkie liście w prawym poddrzewie tejże łodygi. I podobnie jeśli wartość przechowywana w łodydze jest mniejsza niż x2, dodawane są wszystkie liście w lewym poddrzewie tejże łodygi.

Uwagi: W przypadku zaistnienia duplikatów zostaną one potraktowane jako osobne wartości.

Złożoność: $O(\log n + k)$, gdzie n jest liczbą punktów w drzewie, a k jest liczbą punktów znajdujących się w podanym zakresie.

Listing 4.2. Główna funkcja do przeszukiwania zakresowego w drzewie jednowymiarowym.

```
def range_searching(self, x1, x2):
    if x1 > x2:
        raise ValueError("Wrong scope")
    res = []
    if self.root is None:
        return res
    x_split = self.__find_x_split(self.root, x1, x2)
    if x_split.is_leaf():
        if x1 <= x_split.data <= x2:
            for _ in range(x_split.num_of_duplicates):
                res.append(x_split.data)
    else:
        if x_split.left is not None:
            self.__range_searching_left(x_split.left, x1, res)
```

```
15          if x_split.right is not None:
16              self.__range_searching_right(x_split.right, x2, res)
17      return res
```

4.2. Drzewo zakresowe wielowymiarowe

W tej sekcji znajdują się opisy najważniejszych algorytmów używanych w drzewie wielowymiarowym.

4.2.1. Tworzenie drzewa

Algorytm ma za zadanie stworzenie drzewa zakresowego z przekazanych do niego danych.

Dane wejściowe: Lista punktów wielowymiarowych, które zostaną dodane do drzewa.

Opis algorytmu: Algorytm zaczyna się od sprawdzenia, czy podane punkty są poprawne. Następnie zostaje wybrany punkt stanowiący środkowy element przekazanej listy, a sama lista zostaje podzielona na dwie równe części i zostaje rekurencyjnie wywołany algorytm tworzenia drzewa. Jeśli aktualny wymiar jest większy niż pierwszy, zostaje utworzone poddrzewo, które będzie przechowywane w węźle wyższego wymiaru.

Podczas dodawania następuje sortowanie przekazanych punktów po odpowiednich wymiarach.

Uwagi: Struktura pozwala na dodawanie duplikatów.

Złożoność: $O(n \log^d n)$.

4.2.2. Dodawanie nowych elementów do drzewa

Algorytm ma za zadanie dodać do drzewa punkt podany w argumencie.

Dane wejściowe: Punkt dodawany do drzewa.

Opis algorytmu: Algorytm zaczyna się od sprawdzenia, czy zostały przekazane poprawne dane. Następnie tworzony jest węzeł opowiadający liściowi, oraz dodawany jest do niego punkt, a sam węzeł przybiera wartość punktu dla wymiaru drzewa. Czyli jeśli mamy punkt $(1, 10)$ to drzewo węzeł dla drzewa drugiego wymiaru przyjmie wartość 1.

Następnie jeśli wymiar drzewa jest większy niż jeden, zostaje utworzone poddrzewo o wymiarze niższym o jeden.

Potem utworzony węzeł przekazany jest do drzewa, gdzie rekurencyjnie przechodzi do miejsca, gdzie zostanie umieszczony.

Po drodze jeśli napotkany zostanie wezeł o takiej samej wartości, zostanie zaaktualizowana informacja o liczbie duplikatów.

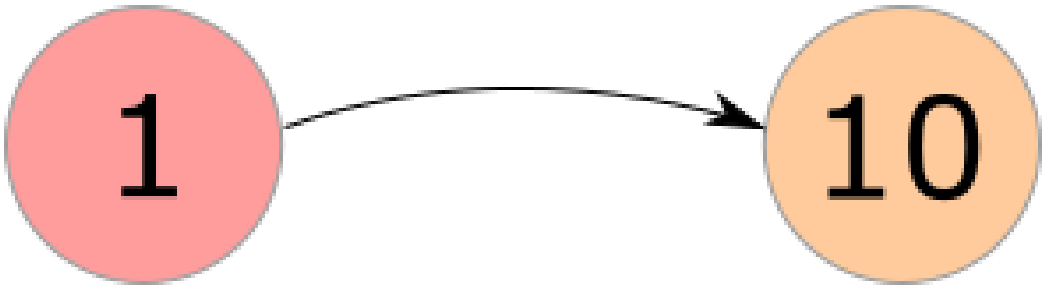
Uwagi: Funkcja pozwala na dodawanie duplikatów.

Złożoność: Złożoność opisanego powyżej schematu wynosi $O(\log n)$ i ponieważ jest on powtarzany dla każdego wymiaru z osobna, złożoność obliczeniowa całości wynosi $O(\log^d n)$.

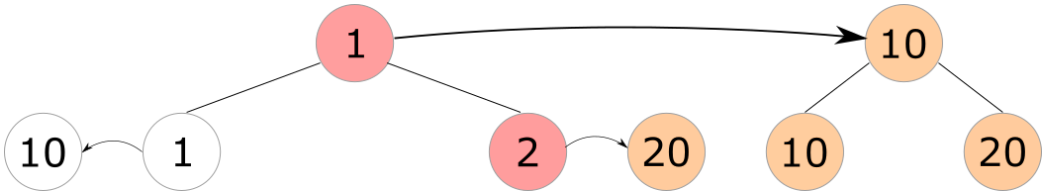
Obrazowe przedstawienie działania algorytmu: Rysunki 4.1, 4.2, 4.3 przedstawiają wygląd drzewa dwuwymiarowego po dodaniu trzech punktów: $(1, 10)$, $(2, 20)$ i $(3, 9)$.

Kolorem czerwonym zostały zaznaczone węzły dla pierwszego wymiaru, w których następuje zmiana podczas algorytmu dodawania nowej wartości.

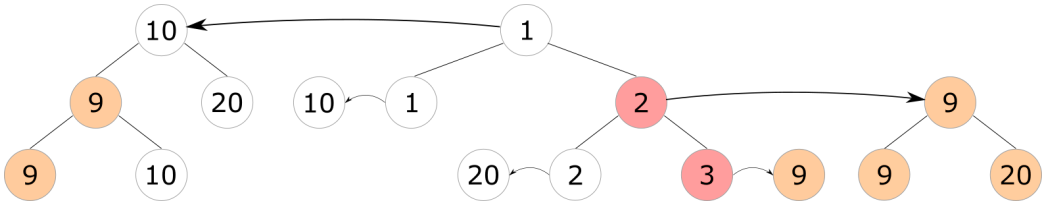
Kolorem pomarańczowym natomiast zostały oznaczone węzły dla drzew drugiego wymiaru, w których nastąpi zmiana.



Rysunek 4.1. Wygląd drzewa po dodaniu punktu $(1, 10)$.



Rysunek 4.2. Wygląd drzewa po dodaniu punktu $(2, 20)$.



Rysunek 4.3. Wygląd drzewa po dodaniu punktu $(3, 9)$.

4.2.3. Usuwanie elementów z drzewa

Algorytm ma za zadanie usuąć z drzewa punkt podany w argumencie.

Dane wejściowe: Punkt, który zostanie usuwanięty z drzewa.

Opis algorytmu: Algorytm zaczyna się od sprawdzenia, czy zostały przekazane poprawne dane. Następnie algorytm rekurencyjnie przechodzi przez drzewo w poszukiwaniu wartości, która ma zostać usunięta. Jeśli dotrze do liścia wyższego wymiaru, algorytm przechodzi do poddrzewa przechowywanego w tym liściu.

Jeśli drzewo znajduje się w pierwszym wymiarze oraz wartość przechowywana przez liścia nie odpowiada wartości poszukiwanej, algorytm kończy swoje działanie zwracając False.

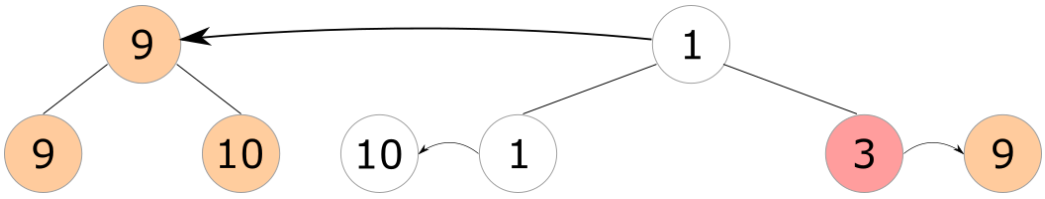
Jeśli zostanie znaleziona szukana wartość, algorytm przekazuje rodzicowi tego węzła informacje o usunięciu liścia i następuje zaaktualizowanie liczby duplikatów, oraz usunięcie łodygi odpowiadającej usuniętemu liściowi.

Złożoność: Złożoność opisanego powyżej schematu wynosi $O(\log n)$ i ponieważ jest on powtarzany dla każdego wymiaru z osobna, złożoność obliczeniowa całości wynosi $O(\log^d n)$.

Obrazowe przedstawienie działania algorytmu: Kolorem czerwonym zostały zaznaczone węzły dla drugiego wymiaru, w których następuje zmiana podczas algorytmu usuwania wartości.

Kolorem pomarańczowym natomiast zostały oznaczone węzły dla drzew pierwszego wymiaru, w których nastąpi zmiana.

Na rysunku 4.4 widoczne jest drzewo po działaniu metody delete(). Usunięty został punkt $(2, 20)$ z drzewa utworzonego na rysunku 4.3.



Rysunek 4.4. Wygląd drzewa po usunięciu punktu $(2, 20)$.

4.2.4. Przeszukiwanie zakresowe na bazie range tree

Algorytm ma za zadanie znaleźć wszystkie punkty z podanego zakresu.

Dane wejściowe: Do algorytmu przekazywane są zakresy, dla których mają zostać wyszukane punkty przechowywane w drzewie dla każdego wymiaru z osobna.

Opis algorytmu: Algorytm (4.3) zaczyna się od sprawdzenia, czy zostały przekazane poprawne dane. Następnie dla drzewa w aktualnym wymiarze w metodzie `__find_x_split()` wyszukiwany jest węzeł rozdzielający `x_split`.

Jeśli znaleziony węzeł jest liściem sprawdzane jest to, czy jego wartość znajduje się w przekazanym przedziale. Jeśli tak, to algorytm przechodzi do wyszukania w niższym wymiarze, o ile taki jest przechowywany w węźle.

Jeśli wezeł jest łodygą, wywoływane są dwie metody `__range_searching_left()` oraz `__range_searching_right()`, które odpowiednio przeszukują lewe oraz prawe poddrzewo węzła `x_split`.

Metody te przechodzą przez drzewo w kierunku liścia, oraz wywołują przeszukiwanie zakresowe dla drzew o niższym wymiarze, o ile takie istnieje. Jeśli metody te są w drzewie dla ostatniego wymiaru, po dotarciu do liścia punkt przez niego przechowywany zostaje dodany do listy, która zostanie zwrócona użytkownikowi po zakończeniu działania metody.

Złożoność: $O(\log^d n + k)$, gdzie n jest liczbą punktów w drzewie, a k jest liczbą punktów znajdujących się w podanym zakresie.

Listing 4.3. Główna funkcja do przeszukiwania zakresowego w drzewie wielowymiarowym.

```
1   def range_searching(self, *ranges):
2
3       reversed_dimension = self.__get_point_for_dimension(ranges, True)
4
5       if len(ranges) - reversed_dimension != self.__get_dimension() or len(ranges[0]) != 2:
6           raise ValueError("Wrong provided ranges number")
7       if ranges[reversed_dimension][0] > ranges[reversed_dimension][1]:
8           raise ValueError("Wrong provided range for dimension", self.__get_dimension())
9
10      res = []
11      if self.root is None:
12          return res
13
14      x_split = self.__find_x_split(self.root, ranges[reversed_dimension][0],
15                                    ranges[reversed_dimension][1])
16      if (x_split.is_leaf() and
17          ranges[reversed_dimension][0] <= x_split.get_value() <= ranges[reversed_dimension][1]):
18          if self.__get_dimension() == 1:
19              for _ in range(x_split.num_of_duplicates):
20                  res.append(x_split.get_point())
21          else:
22              res.extend(x_split.get_subtree().range_searching(*ranges))
23      else:
24          if x_split.left:
25              self.__range_searching_left(x_split.left, ranges, res)
26          if x_split.right:
27              self.__range_searching_right(x_split.right, ranges, res)
28      return res
29
```

5. Podsumowanie

Celem pracy było przedstawienie problemu wyszukiwania zakresowego i stworzenie implementacji drzewa zakresowego w języku Python, z wykorzystaniem narzędzi z biblioteki standardowej. Dużą trudnością okazało odpowiednie przebalansowanie drzewa z zachowaniem odpowiednich powiązań z poddrzewami, jak i informacją o liczbie duplikatów. Jednakże udało się ten problem rozwiązać, przez co wielkość struktury dla większej liczby wymiarów jest zależna od różnych danych, ponieważ dla duplikatów jest przechowywana tylko informacja o ich liczbie, a nie zduplikowane punkty. Struktura może zostać użyta jako podstawa do stworzenia bardzo dużej bazy danych z szybkim pozyskiwaniem informacji o elementach w niej zawartych. Dodatkowo stanowi ona dużą pomoc w problemie, który jest nieodłączną częścią pracy osób zajmujących się tematyką przeszukiwania dużych zbiorów danych.

W pierwszej części pracy (rozdz. 2) przedstawionio krótki opis problemu przeszukiwania zakresowego, oraz teoretyczny opis założeń działania drzewa zakresowego dla jednego oraz wielu wymiarów. Informacje teoretyczne o wyglądzie oraz działaniu drzewa zakresowego zostały zaczerpnięte z książki Bentley'a [12], oraz innych artykułów naukowych i prezentacji osób zajmujacych się tą tematyką. Dodatkowo zamieszczono grafiki w celu lepszego zrozumienia wyglądu drzewa, oraz działania algorytmu przeszukiwania zakresowego dla różnej liczby wymiarów.

W części drugiej (rozdz. 3) została przedstawiona implementacja drzewa zakresowego w postaci klasy RangeTree(). Podano opis metod, przykłady użycia i zwracanych wartości. W osobnym rozdziale 4 opisano algorytmy kryjące się za metodami, podano ich złożoność obliczeniową. Ponadto zamieszczona została wizualizacja zmian w wyglądzie drzewa zakresowego podczas dodawania i usuwania elementów.

Praca pozwala czytelnikowi zrozumieć podstawy działania drzew zakresowch, oraz sposobu na wydajne rozwiązanie problemu przeszukania wybranego zakresu danych. Stworzona implementacja została starannie przetestowana (dodatek A) i z powodzeniem sprawdza się w wyszukiwaniu elementów znajdujących się w podanym zakresie dla dużej liczby wielowymiarowych danych. Kod działa prawidłowo w Pythonie 2.7 i Pythonie 3.

A. Testy algorytmów

W tym rozdziale opracowane są wyniki testów złożoności algorytmów. Wynikiem testu wydajności jest wykres zależności czasu działania algorytmu od liczności zbioru punktów przekazanych do struktury. Testy były przeprowadzone na komputerze z procesorem Intel Core i5 3.7GHz.

Algorytmy zostały dodatkowo przetestowane przy pomocy testów jednostkowych z użyciem modułu unittest [11].

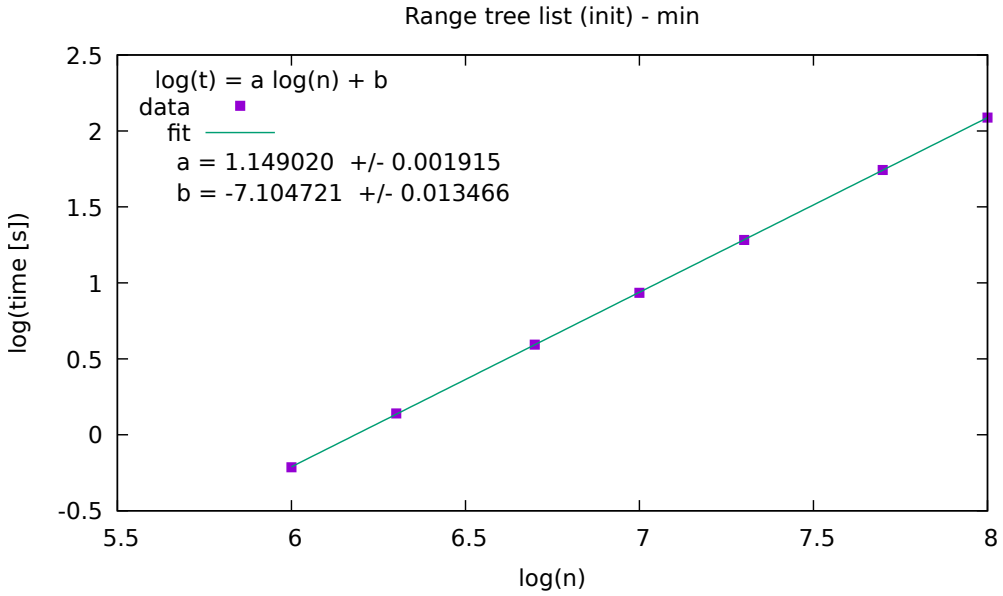
A.1. Drzewo zakresowe - lista posortowana

W celu porównania działania stworzonych struktur przygotowana została prosta implementacja drzewa zakresowego. Wykorzystuje ona standardową listę z biblioteki języka Python, gdzie elementy na liście są stale posortowane, a operacje na liście są wykonywane z użyciem modułu bisect. Operacje wyszukiwania danego elementu lub wyszukiwania zakresowego mają złożoność logarytmiczną.

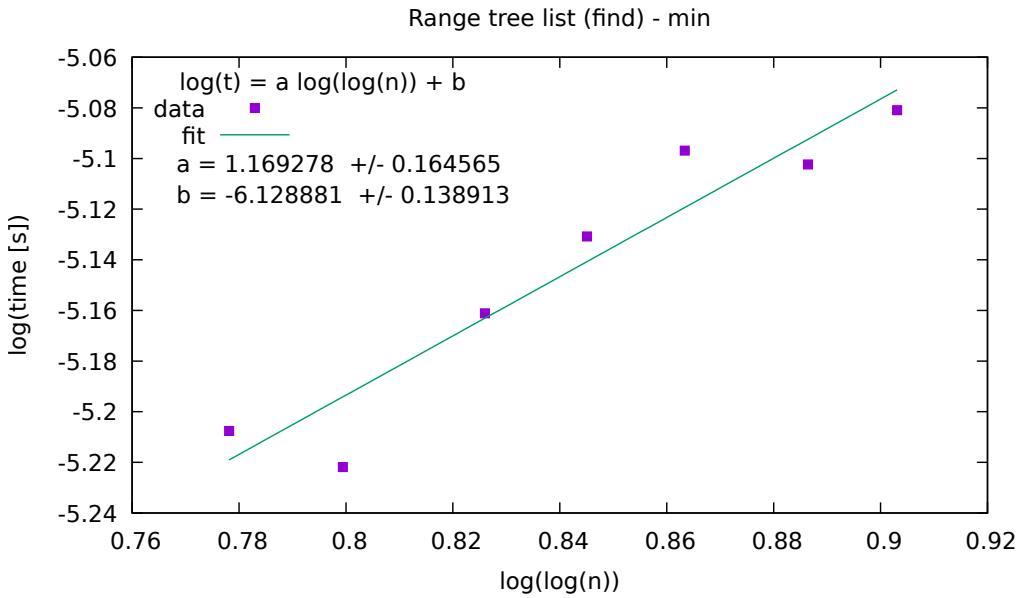
Tworzenie struktury. Teoretyczna złożoność algorytmu tworzenia drzewa wynosi $O(n \log n)$. Widoczny na wykresie A.1 współczynnik $a = 1.149(2)$ oznacza, że złożoność tejże struktury jest ponad $O(n)$ i jest zbliżona do teoretycznej.

Wyszukiwanie jednego liścia z drzewa. Teoretyczna złożoność algorytmu wyszukiwania wartości w drzewie wynosi $O(\log n)$. Widoczny na wykresie A.3 współczynnik wynosi $a = 1.17(16)$, co oznacza, że złożoność wynosi $O(\log(n))$ i jest to złożoność toretyczna. Zmierzone czasy były bardzo krótkie (milisekundy), przez co wpływ innych procesów w systemie operacyjnym przyniósł pewne zaszumienie wyników.

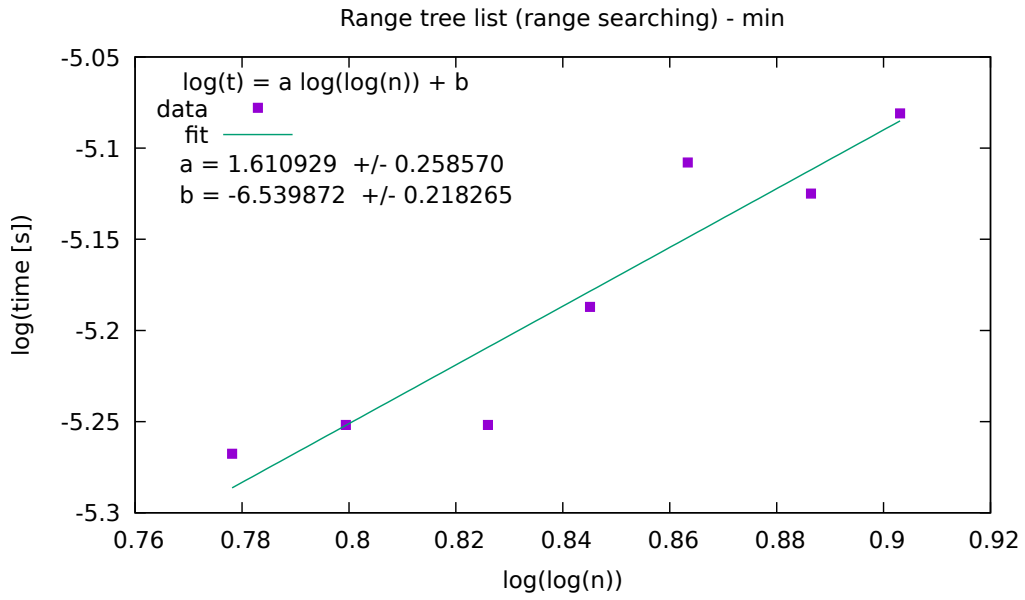
Przeszukiwanie zakresowe. Teoretyczna złożoność algorytmu wyszukiwania punktów przechowywanych w drzewie wynosi $O(\log n + k)$, gdzie k jest liczbą punktów znalezionych podczas przeszukiwania drzewa. Widoczny na wykresie A.3 współczynnik wynosi $a = 1.61(26)$, co oznacza, że złożoność jest powyżej $O(\log(n))$, oraz że jest nieco większa niż podczas wyszukiwania jednego punktu A.3. Możemy stwierdzić, że pojawia się zależość od k, czyli liczby znalezionych punktów, przez co jest ona zbliżona do teoretycznej. Również w tym przypadku zmierzone czasy były bardzo krótkie (milisekundy), przez co widoczne jest zaszumienie pochodzące od systemu operacyjnego.



Rysunek A.1. Wyniki pomiarów tworzenia drzewa zakresowego dla jednego wymiaru na bazie listy.



Rysunek A.2. Wyniki pomiarów wyszukiwania odpowiedniego liścia drzewa zakresowego dla jednego wymiaru na bazie listy.



Rysunek A.3. Wyniki pomiarów wyszukiwania zakresowego drzewa zakresowego dla jednego wymiaru na bazie listy.

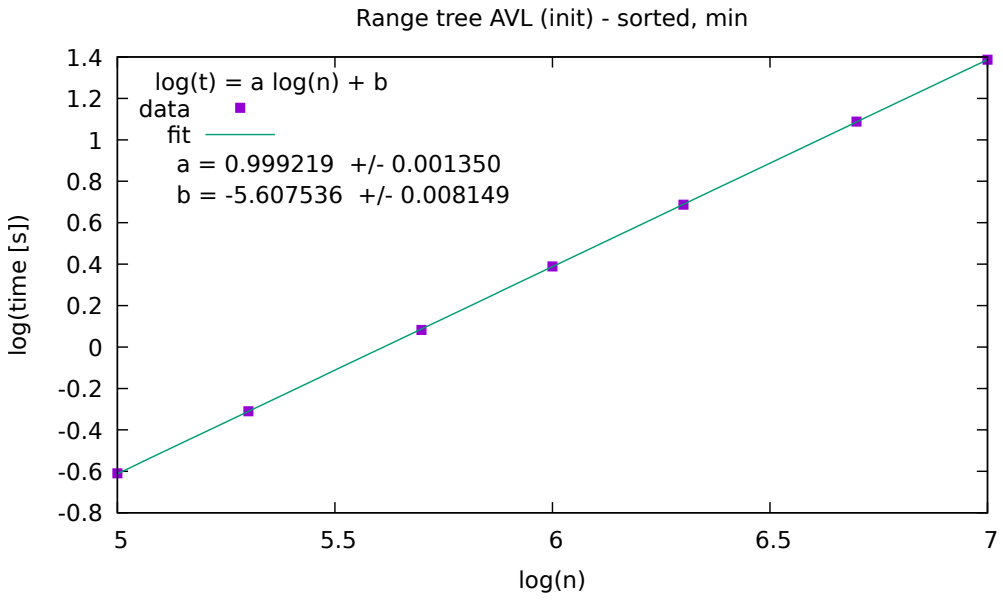
A.2. Drzewo zakresowe - AVL-jednowymiarowe

Struktura została stworzona na bazie drzewa AVL i zoptymalizowana pod działanie w jednym wymiarze.

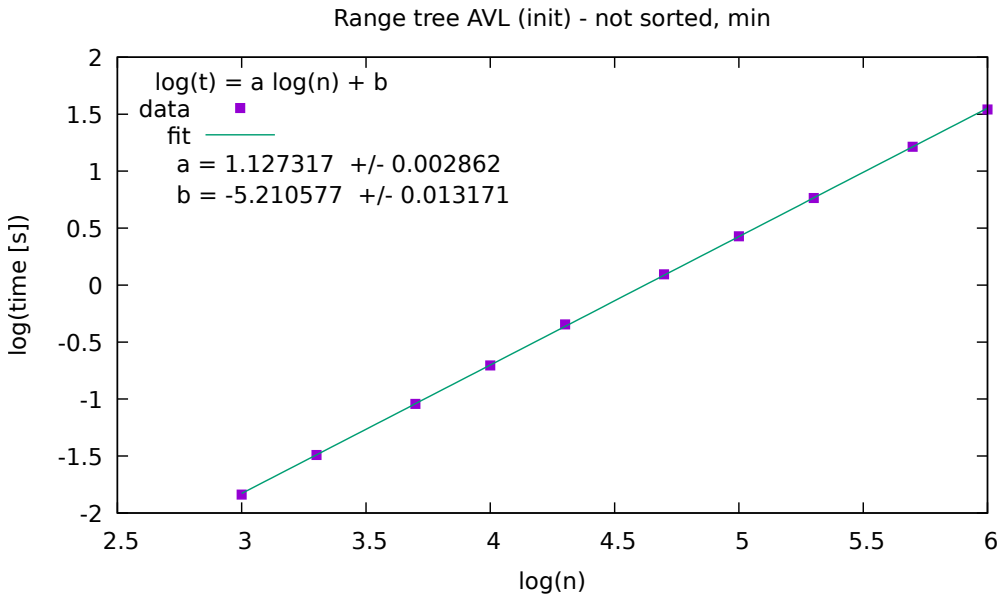
Tworzenie struktury. Teoretyczna złożoność algorytmu tworzenia drzewa jednowymiarowego wynosi $O(n \log n)$. Jeśli punkty wykorzystane do tworzenia drzewa są wcześniej posortowane, wtedy można otrzymać złożoność $O(n)$. Widoczny na wykresie A.4 współczynnik $a = 0.9992(14)$ wskazuje na to, że złożoność jest liniowa. Dodatkowo przetestowane zostało tworzenie drzewa dla nieposortowanych danych A.5. W tym przypadku współczynnik $a = 1.127(3)$ okazał się wyższy niż dla danych posorotwanych, a jego wartość wskazuje na to, że wystąpiła dodatkowa zależność od etapu sortowania danych podczas tworzenia drzewa.

Wyszukiwanie jednego liścia z drzewa. Teoretyczna złożoność algorytmu wyszukiwania wartości w drzewie wynosi $O(\log n)$. Widoczny na wykresie A.6 współczynnik $a = 0.795(78)$ oznacza, że uzyskany czas podczas testów potwierdza złożoność teoretyczną.

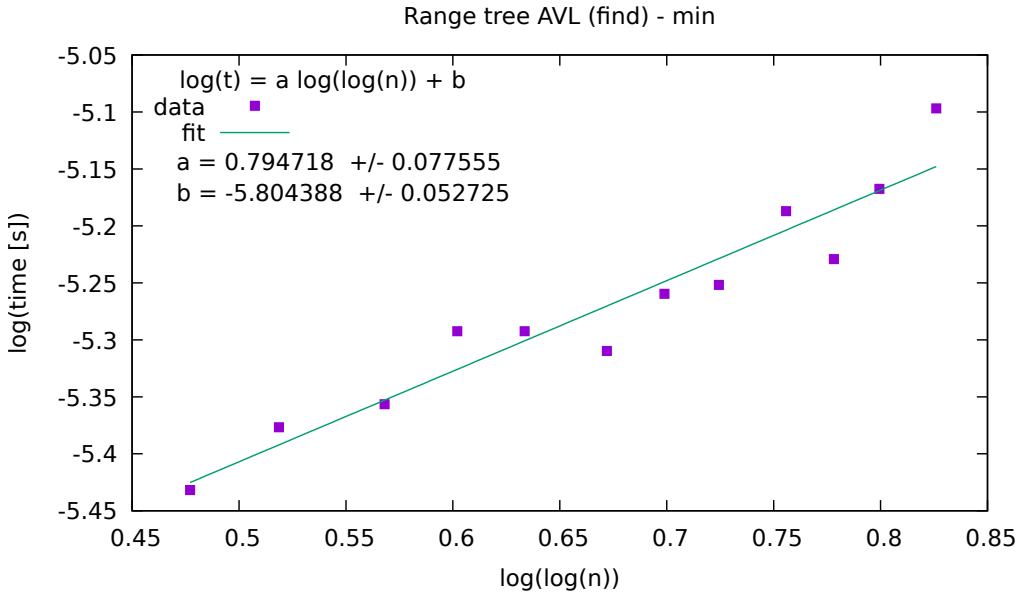
Przeszukiwanie zakresowe. Teoretyczna złożoność algorytmu wyszukiwania wartości w drzewie wynosi $O(\log n + k)$, gdzie k jest liczbą punktów znalezionych podczas przeszukiwania drzewa. Widoczny na wykresie A.7 współczynnik $a = 0.962(43)$ oznacza, że uzyskana złożoność spełnia założenia teoretyczne.



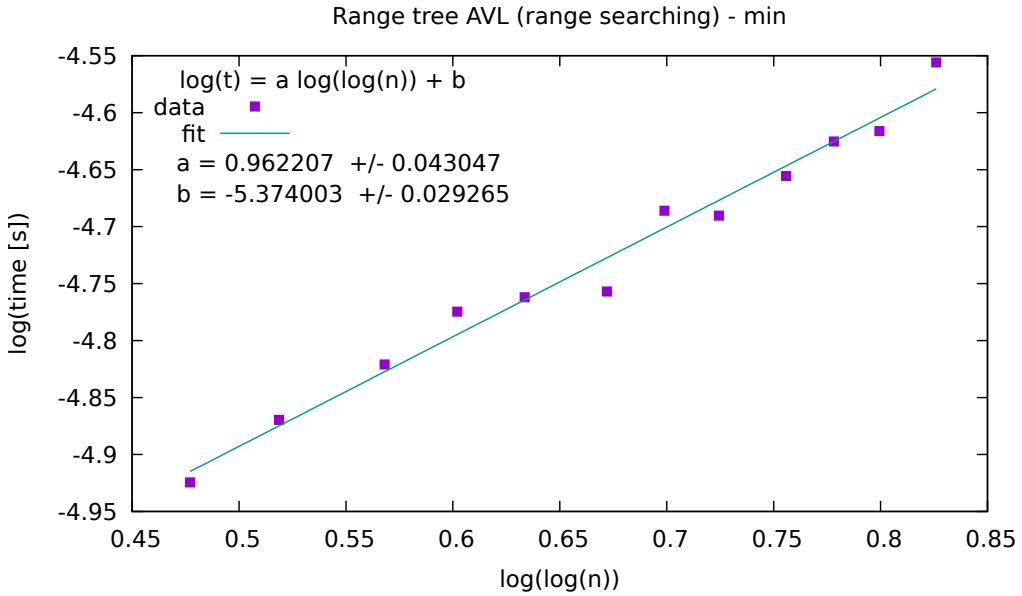
Rysunek A.4. Wyniki pomiarów tworzenia drzewa zakresowego dla jednego wymiaru na bazie drzewa AVL przy użyciu posortowanych danych.



Rysunek A.5. Wyniki pomiarów tworzenia drzewa zakresowego dla jednego wymiaru na bazie drzewa AVL przy użyciu nieposortowanych danych.



Rysunek A.6. Wyniki pomiarów wyszukiwania odpowiedniego liścia drzewa zakresowego dla jednego wymiaru na bazie drzewa AVL.



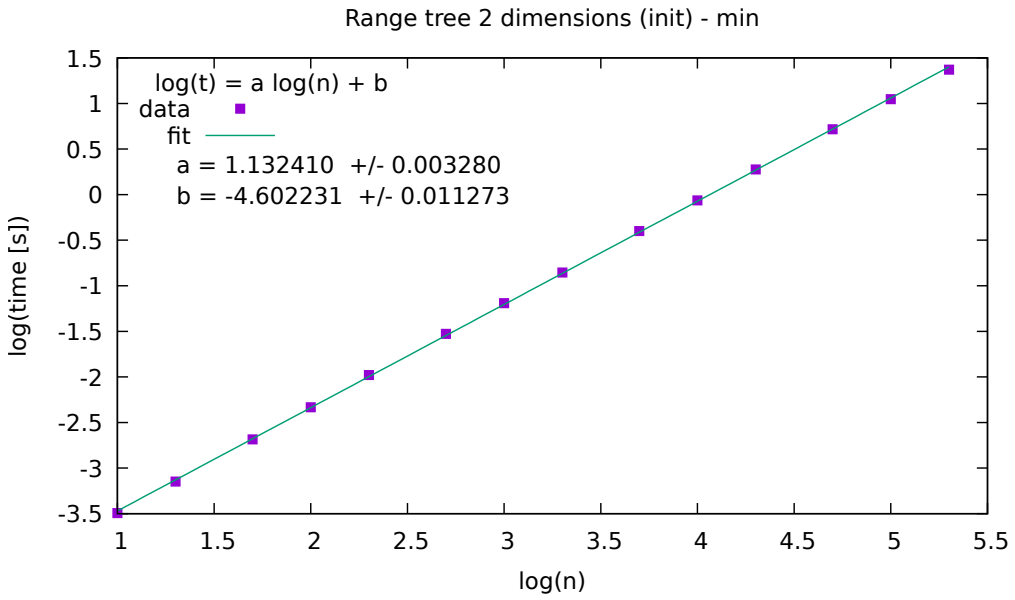
Rysunek A.7. Wyniki pomiarów wyszukiwania zakresowego drzewa zakresowego dla jednego wymiaru na bazie drzewa AVL.

A.3. Drzewo zakresowe - AVL-wielowymiarowe

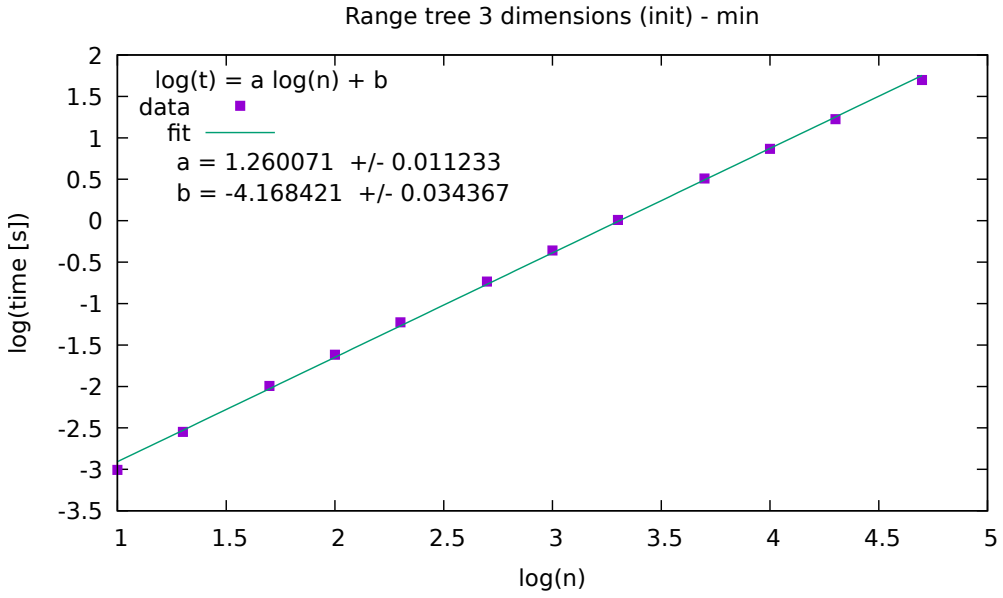
Drzewo zakresowe umożliwia przechowywanie wielowymiarowych punktów. Zostały przeprowadzone testy dla 2, 3, 4 oraz 5 wymiarów. Dla 4 i 5 wymiarów ilość danych została zmniejszona przez ograniczenia sprzętu, na jakim była ta struktura testowana.

Tworzenie struktury. Teoretyczna złożoność algorytmu tworzenia drzewa $O(n \log^d n)$, gdzie d oznacza liczbę wymiarów. Jeśli wprowadzane do drzewa punkty są odpowiednio posortowane, wtedy można otrzymać złożoność $O(n \log^{d-1} n)$, gdzie d oznacza liczbę wymiarów. Widoczne na wykresach A.8, A.9, A.10, A.11 współczynniki $a > 1$ pokazują, że złożoność rośnie dla kolejnych wymiarów, co potwierdza istnienie zależności od liczby wymiarów wstawianych punktów.

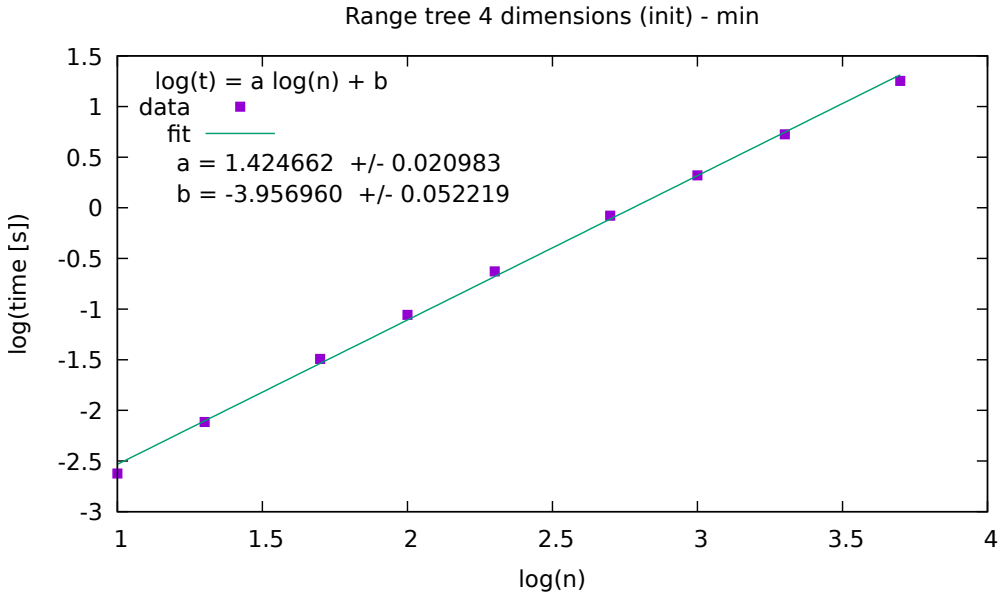
Przeszukiwanie zakresowe. Teoretyczna złożoność algorytmu tworzenia drzewa jednowymiarowego wynosi $O(\log^d n + k)$, gdzie k jest liczbą punktów znalezionych podczas przeszukiwania drzewa. Współczynniki a na wykresach A.12, A.13, A.14, A.15 rosną wraz z dodaniem kolejnego wymiaru, co potwierdza istnienie zależności od liczby wymiarów wstawianych punktów.



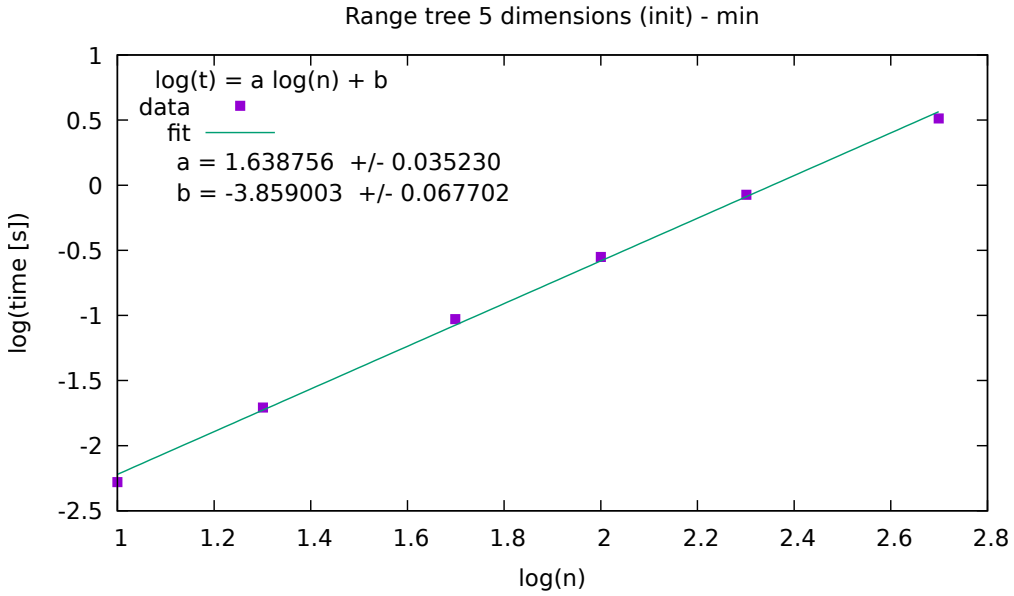
Rysunek A.8. Wyniki pomiarów tworzenia drzewa zakresowego dla dwóch wymiarów.



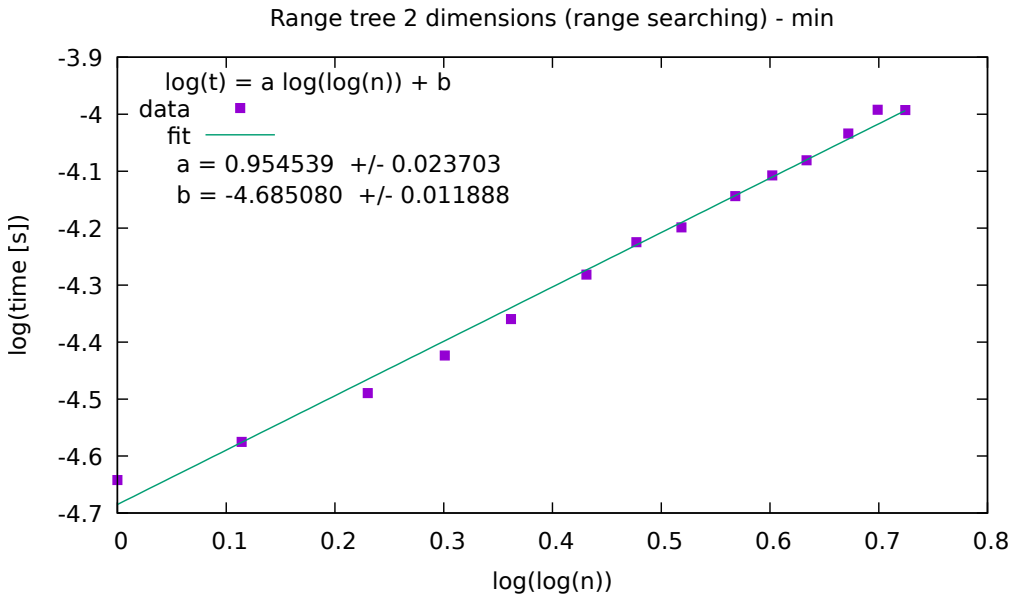
Rysunek A.9. Wyniki pomiarów tworzenia drzewa zakresowego dla trzech wymiarów.



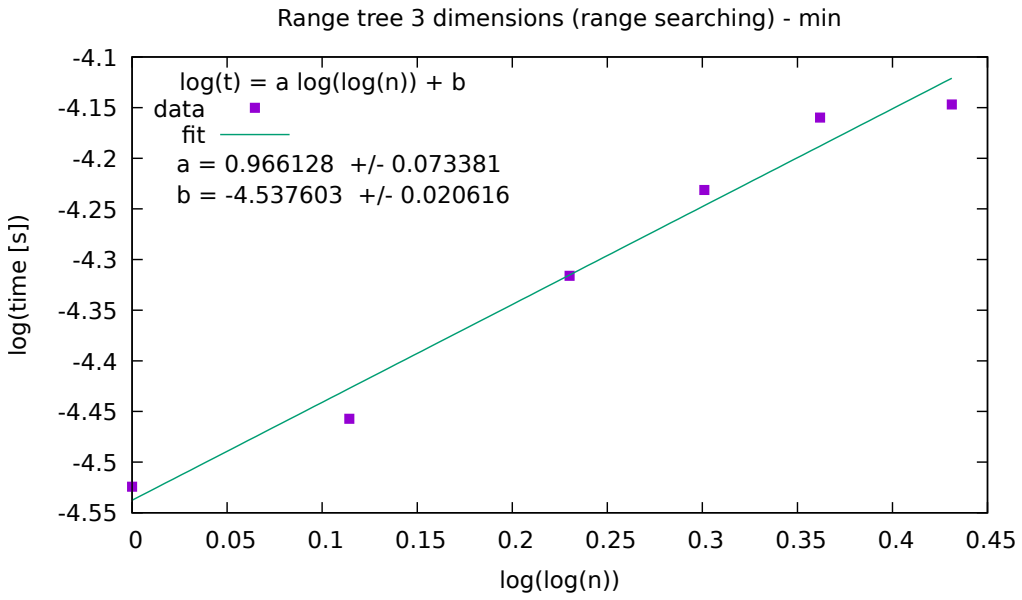
Rysunek A.10. Wyniki pomiarów tworzenia drzewa zakresowego dla czterech wymiarów.



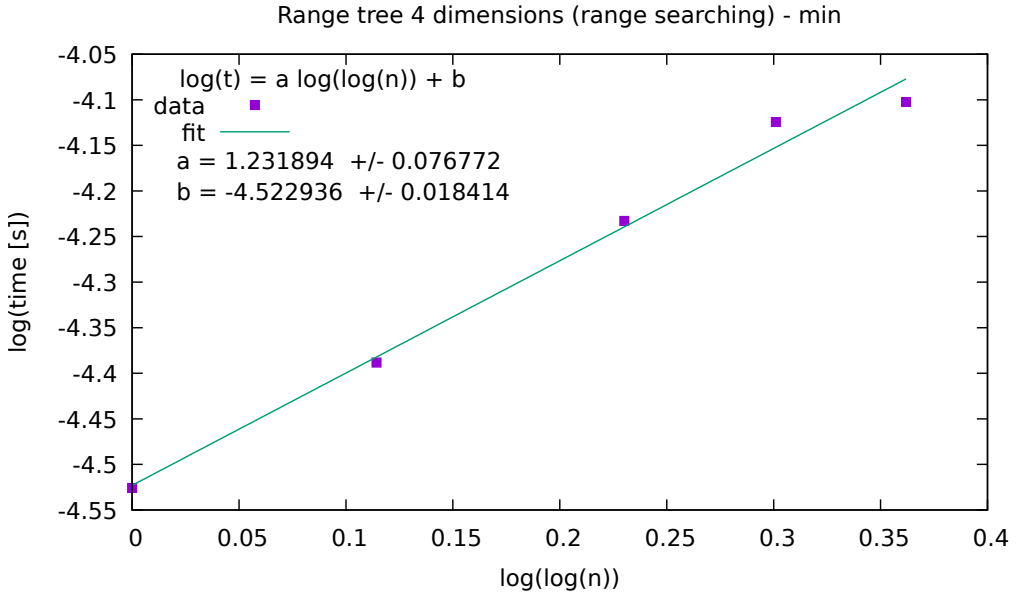
Rysunek A.11. Wyniki pomiarów tworzenia drzewa zakresowego dla pięciu wymiarów.



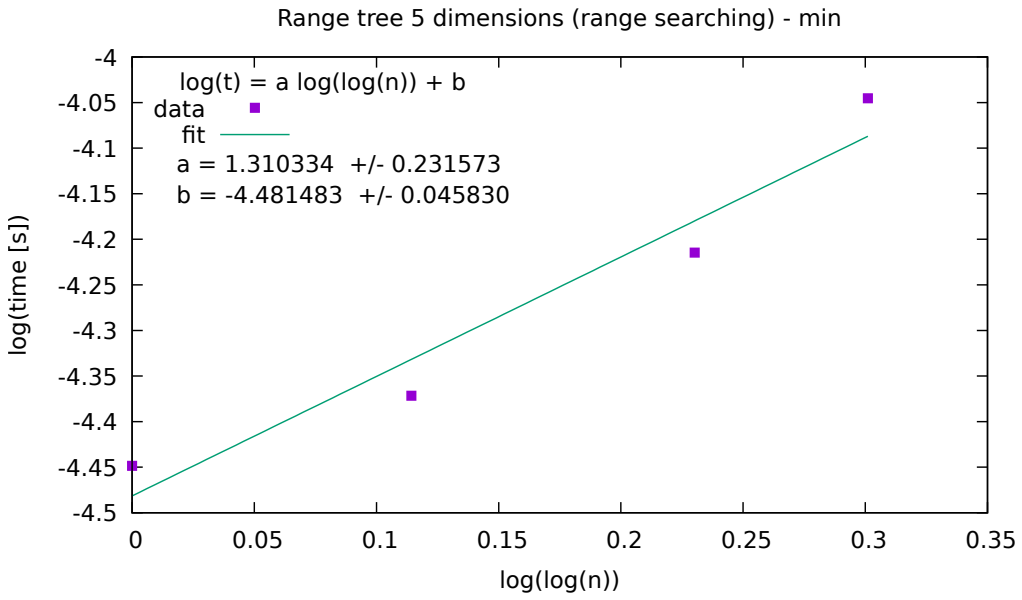
Rysunek A.12. Wyniki pomiarów wyszukiwania zakresowego drzewa zakresowego dla dwóch wymiarów.



Rysunek A.13. Wyniki pomiarów wyszukiwania zakresowego drzewa zakresowego dla trzech wymiarów.



Rysunek A.14. Wyniki pomiarów wyszukiwania zakresowego drzewa zakresowego dla czterech wymiarów.



Rysunek A.15. Wyniki pomiarów wyszukiwania zakresowego drzewa zakresowego dla pięciu wymiarów.

Bibliografia

[1] Python Programming Language - Official Website, https://www.python.org/.

[2] Wikipedia, Range searching, 2021, https://en.wikipedia.org/wiki/Range_searching.

[3] Wikipedia, Interval tree, 2021, https://en.wikipedia.org/wiki/Interval_tree.

[4] Wikipedia, Range tree, 2021, https://en.wikipedia.org/wiki/Range_tree.

[5] Wikipedia, K-d tree, 2021, https://en.wikipedia.org/wiki/K-d_tree.

[6] Wikipedia, Quadtree, 2021, https://en.wikipedia.org/wiki/Quadtree.

[7] Wikipedia, Skip list, 2021, https://en.wikipedia.org/wiki/Skip_list.

[8] Wikipedia, AVL tree, 2021, https://en.wikipedia.org/wiki/AVL_tree.

[9] Jerzy Wałaszek, *Drzewa AVL*, 2021, https://eduinf.waw.pl/inf/alg/001_search/0119.php.

[10] Paul E. Black, *data structure*, 2004, https://xlinux.nist.gov/dads/HTML/datastructur.html.

[11] Python Docs, *unittest - Unit testing framework*, 2021, https://docs.python.org/3/library/unittest.html.

[12] Jon Louis Bentley, *Decomposable searching problems*, 1979, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.880.3974&rep=rep1&type=pdf.

[13] Oracle Corporation, *Range Searching*, 2010, https://docs.oracle.com/cd/E19509-01/820-2719/cnfg_index-range-srch_c/index.html.

[14] Subhash Suri, UC Santa Barbara https://sites.cs.ucsb.edu/~suri/cs235/RangeSearching.pdf.

[15] Bernard Chazelle, *Lower Bounds for Orthogonal Range Searching*, 1989, https://www.cs.princeton.edu/~chazelle/pubs/LBOrthoRangeSearchReporting.pdf.

[16] Pankaj K. Agarwal and Jeff Erickson, *Geometric Range Searching and Its Relatives*, 1997, https://jeffe.cs.illinois.edu/pubs/pdf/survey.pdf

[17] Bentley Jon, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 1975.

[18] Bentley Jon, *Multidimensional divide-and-conquer*, Communications of the ACM, 1980.

[19] Mehlhorn Kurt, Stefan Näher, *Dynamic fractional cascading*, Algorithmica 5, 215-241 (1990).

[20] de Berg Mark, Cheong Otfried, van Kreveld Marc, Overmars Mark, *Computational Geometry*, 2008.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[22] George S. Lueker, *A data structure for orthogonal range queries*, IEEE Computer Society, 1978,
https://doi.org/10.1109/SFCS.1978.1

[23] French Carl, *Data Processing and Information Technology*, 1996,
https://books.google.de/books?id=zVCdg7Tg6-AC&q=inauthor:%22Carl+French%22&pg=PR2&redir_esc=y

[24] Price Ron, *A List of the Steps in the Information Processing Cycle*, sciencing.com, 2021,
https://sciencing.com/info-10014979-list-steps-information-processing-cycle.html

[25] Alper Ungor, *Computational Geometry*, University of Florida, 2018,
https://www.cise.ufl.edu/class/cot5520sp18/CG_RangeTrees.pdf

[26] Martel Chip, Nuckolls Glen, Gertz Michael, Devanbu Premkumar, Kwong April, Stubblebine Stuart, *A General Model for Authentic Data Publication*, University of Florida, 2001,
https://www.cise.ufl.edu/class/cot5520sp18/CG_RangeTrees.pdf