W BST złożoność czasowa zależy od wysokości drzewa i ilości węzłów. Dla nierównoważonego drzewa BST wynosi O(n)

```
                 #########################################################
                                          # BST
                 #########################################################
class Node:
    def __init__(self, data = None, par = None):
        self.data = data
        self.left = self.right = None
        self.parent = par

class Tree:
    def __init__(self):
        self.dummy = Node('u')
        self.root = self.dummy.right
        # do korzenia dodajemy sztucznego rodzica

    def find(self, node, value):
        if node is None:
            return None, False

        if value == node.data:
            return node, True

        if value < node.data:
            if node.left:
                return self.find(node.left, value)

        if value > node.data:
            if node.right:
                return self.find(node.right, value)

        return node, False

    def append(self, obj):
        if not isinstance(obj, Node):
            obj = Node(obj)
        if self.root is None:
            self.root = obj
            self.root.parent = self.dummy

        s, fl_find = self.find(self.root, obj.data)

        if not fl_find and s:
            if obj.data < s.data:
                s.left = obj
                obj.parent = s
            else:
                s.right = obj
                obj.parent = s
```

Drzewo Splay to rodzaj drzewa w którym ostatnio dostępne elementy są przesuwane na szczyt drzewa. Złożoność czasowa wynosi O(log n).

```python
                  ########################################################
                                              # SPLAY TREE
                  ########################################################
class NodeSplay:
    def __init__(self, data = None, par = None):

        self.data = data
        self.left = self.right = None
        self.parent = par
class SplayTree:

    def __init__(self):
        self.root = None

    def right_rotate(self, x):

        y=x.left
        x.left=y.right

        if y.right:
            y.right.parent = x
        y.parent = x.parent

        if x.parent is None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y

        y.right = x
        x.parent = y

    def left_rotate(self, x):

        y = x.right
        x.right = y.left

        if y.left:
            y.left.parent = x
        y.parent = x.parent

        if x.parent is None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
```

```
        y.left = x
        x.parent = y


    def splay(self, x):
        while x.parent:
            if not x.parent.parent: # jesli x jest dzieckiem root'a
                if x.parent.left == x:
                    self.right_rotate(x.parent) # obrot w prawo
                else:
                    self.left_rotate(x.parent) # obrot w lewo

            elif x.parent.left == x and x.parent.parent.left == x.parent: # obrot 2x w prawo
                self.right_rotate(x.parent.parent)
                self.right_rotate(x.parent)

      elif x.parent.right == x and x.parent.parent.right == x.parent: # obrot 2x w lewo
                self.left_rotate(x.parent.parent)
                self.left_rotate(x.parent)

            elif x.parent.left == x and x.parent.parent.right == x.parent: # obrot w prawo potem lewo
                self.right_rotate(x.parent)
                self.left_rotate(x.parent)

            else:      # obrot w lewo potem w prawo
                self.left_rotate(x.parent)
                self.right_rotate(x.parent)

    def find(self, key):
        node = self.root
        while node:
            if key < node.data:
                if not node.left:
                    break
                node = node.left

            elif key > node.data:
                if not node.right:
                    break
                node = node.right

            else:
                return node
        return node

    def search(self, key):
        node = self.find(key)
        if node and node.data == key:
            self.splay(node)
            return node
        return None
```

```python
    def insert(self, key):
        if not self.root:
            self.root = NodeSplay(key)
            return

        node = self.find(key)

        if node.data == key:
            self.splay(node)
            return

        new_node = NodeSplay(key)
        new_node.parent = node

        if key < node.data:
            node.left = new_node
        else:
            node.right=new_node

        self.splay(new_node)
```

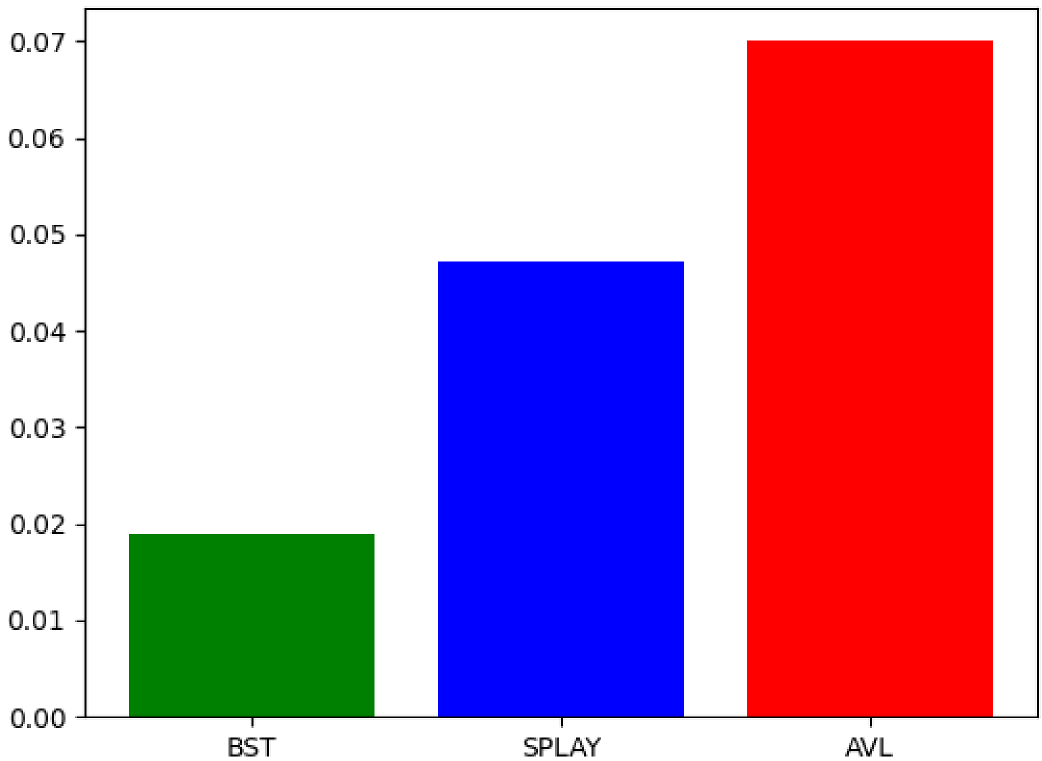
AVL tree to rodzaj drzewa, które jest samobalansujące. Różnica wysokości między lewym i prawym poddrzewem dowolnego węzła w AVL tree jest nie większa niż 1. Dzięki temu operacje wstawiania, usuwania i wyszukiwania mają gwarantowaną złożoność czasową O(log n

```python
                  ########################################################
                                        # AVL TREE
                  ########################################################

class NodeAVL:
    def __init__(self, data, par = None):
        self.data = data
        self.right = self.left = None
        self.height = 1
        self.parent = par
class AVLTree:
    def __init__(self):
        self.root = None

    def _height(self, node):
        if node:
            return node.height
        else:
            return 0
    # Aktualizuje wysokość node na podstawie wysokości jego potomków
    def update_height(self, node):
        if node:
            node.height = 1 + max(self._height(node.left), self._height(node.right))
    # Oblicza współczynnik równowagi node
```

```python
    def balance_factor(self, node):
        if node:
            return self._height(node.left) - self._height(node.right)
        else:
            return 0

    def right_rotate(self, y):
        x = y.left # Ustawienie x jako lewe dziecko y
        T2 = x.right # T2 jest prawym poddrzewem x, które zostanie przesunięte

        # Rotacja
        x.right = y # x staje się rodzicem y
        y.left = T2 # T2 staje się lewym dzieckiem y

        # Aktualizacja wysokości
        self.update_height(y)
        self.update_height(x)

        return x

    def left_rotate(self, x):
        y = x.right
        T2 = y.left

        # Rotacja
        y.left = x
        x.right = T2

        # Aktualizacja wysokości
        self.update_height(x)
        self.update_height(y)

        return y

    def rebalance(self, node):
        self.update_height(node)
        balance = self.balance_factor(node)

        # Lewy przypadek
        if balance > 1:
            if self.balance_factor(node.left) < 0:  # Lewo-prawo przypadek
                node.left = self.left_rotate(node.left)
            return self.right_rotate(node)

        # Prawy przypadek
        if balance < -1:
            if self.balance_factor(node.right) > 0:  # Prawo-lewo przypadek
                node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

    def _insert(self, node, key):
        if not node:
```

```
            return NodeAVL(key)
        if key < node.data:
            node.left = self._insert(node.left, key)
        else:
            node.right = self._insert(node.right, key)

        return self.rebalance(node)

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _search(self, node, key):
        if not node or node.data == key:
            return node
        if key < node.data:
            return self._search(node.left, key)
        return self._search(node.right, key)

    def search(self, key):
        return self._search(self.root, key)
```

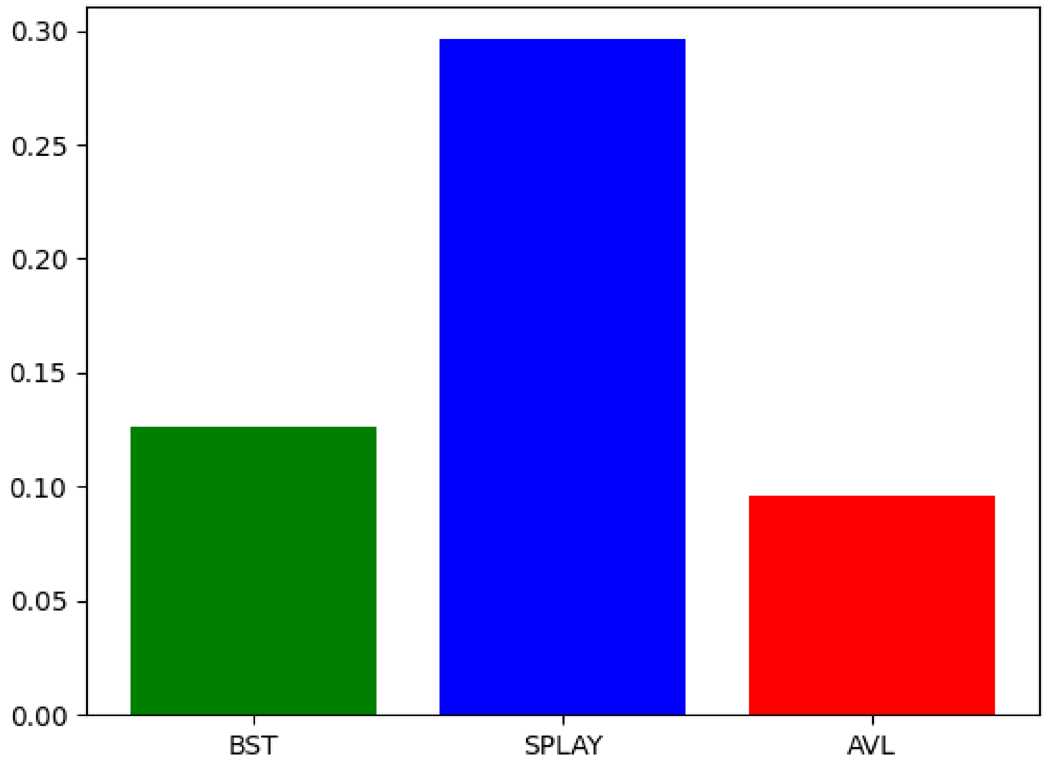
Szybkość tworzenia się drzewa



Drzewo BST tworzy się najszybciej, po nim SPLAY a najdłużej AVL.

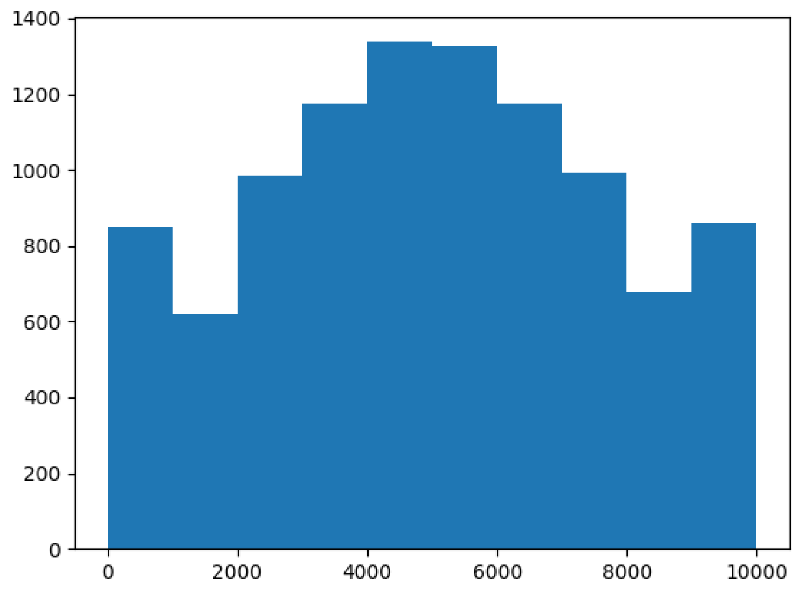
WYSZUKIWANIE

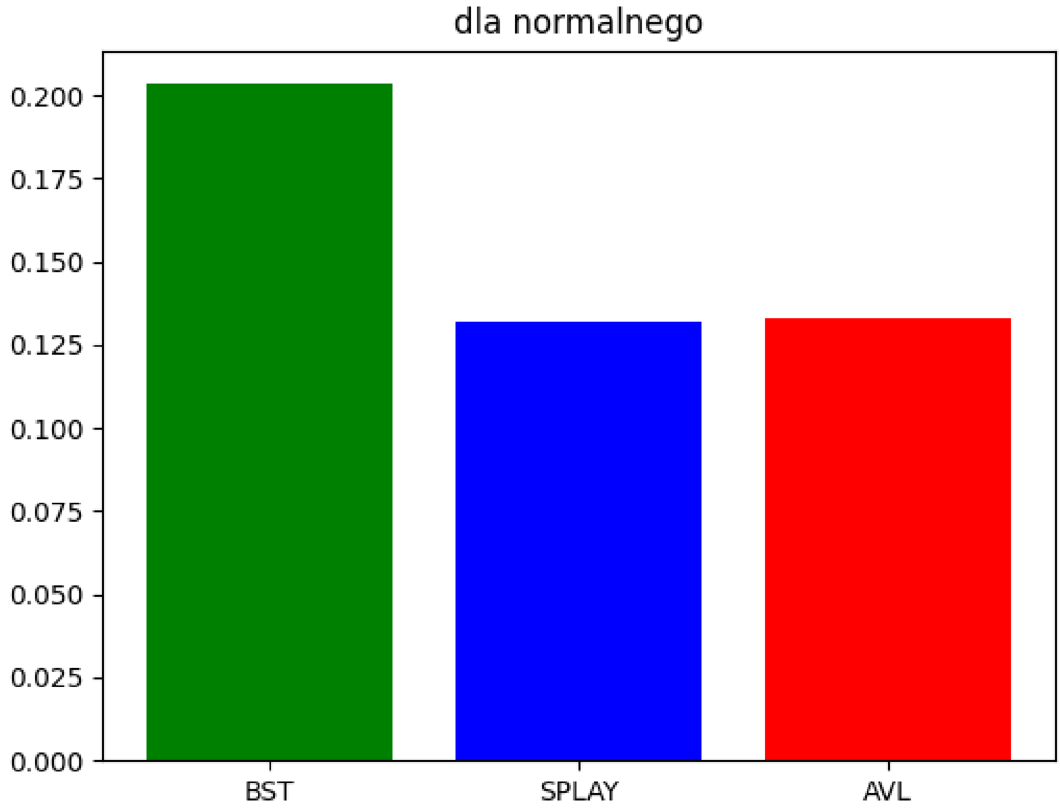
ROZKLAD JEDNOSTAJNY



W wyszukiwaniu losowych liczb dla rozkładu jednostajnego najlepiej radzą sobie BST - 0.01939s i AVL- 0.07133s których czasy są bardzo podobne, za to SPLAY - 0.04556 potrzebuje ponad 2 razy więcej czasu niż BST.

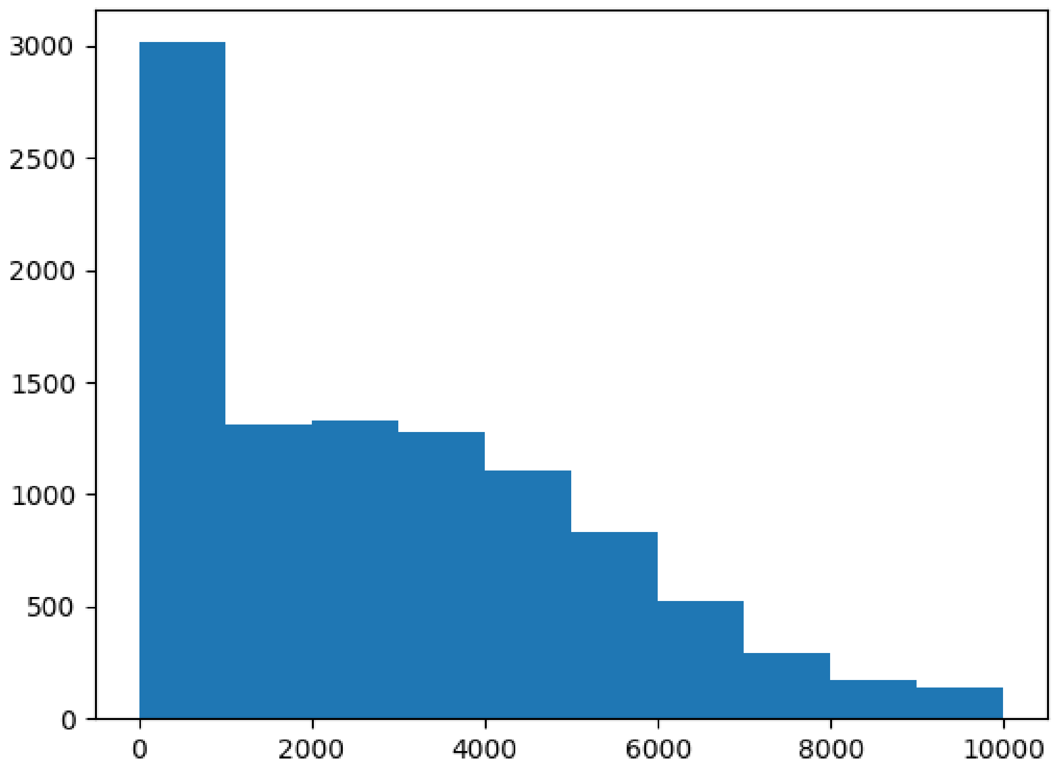
ROZKLAD NORMALNY

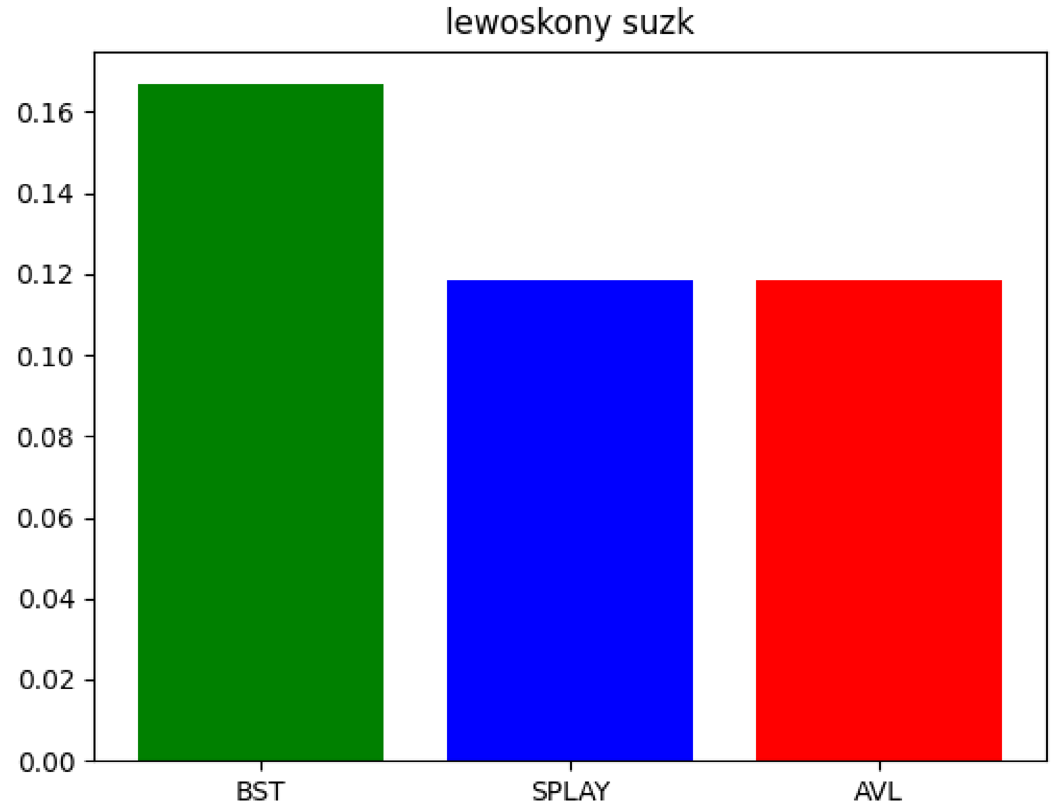




W wyszukiwaniu liczb dla rozkładu normalnego najlepiej radzi sobie i SPLAY - 0.13201s i AVL - 0.13332S .
BST - 0.17015s wypadł najgorzej

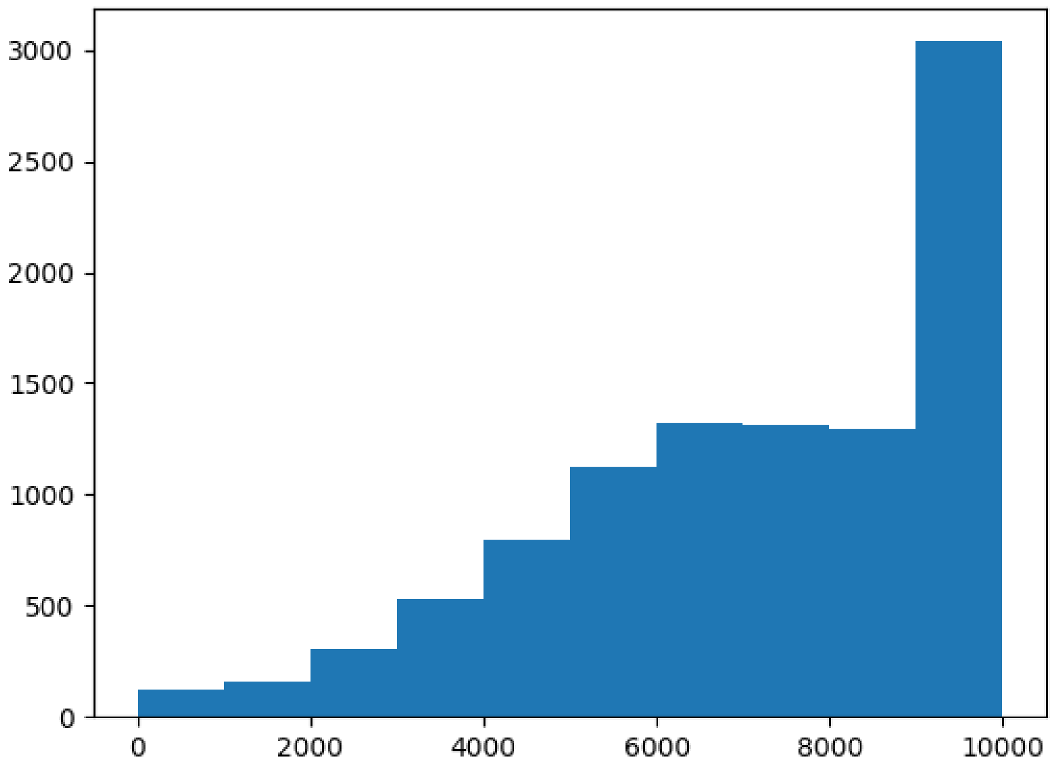
ROZKLAD LEWOSKOSNY

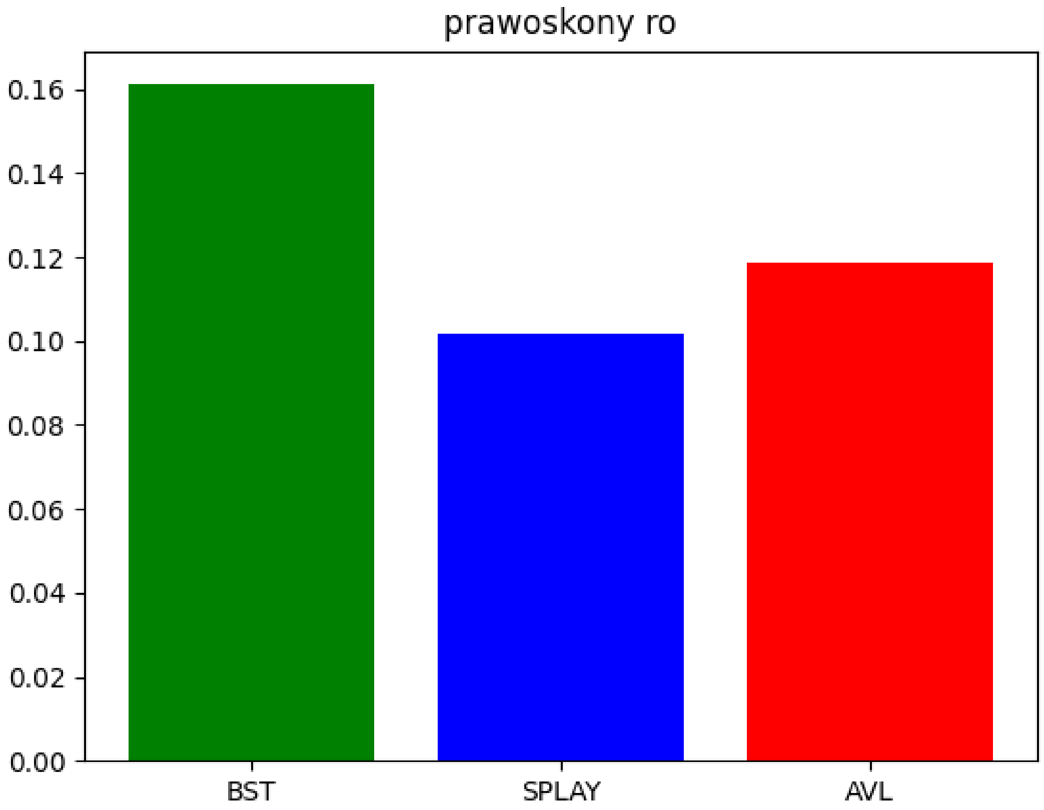




W wyszukiwaniu liczb dla rozkładu lewoskośnego wyniki SPLAY i AVL są praktycznie identyczne. BST potrzebował najwięcej czasu.

ROZKAL PRAWOSKOŚNY





Najlepiej wypada SPLAY , po nim AVL i na końcu BST

Podsumowanie

Drzewa AVL zapewniają najspójniejszą wydajność wyszukiwania dzięki samobalansującej się strukturze, ale są najwolniejsze w tworzeniu. Drzewa Splay są efektywne w scenariuszach, gdzie często dostępne elementy są wielokrotnie wyszukiwane, mimo że ich czas wyszukiwania jest zmienny. Drzewa BST są najszybsze w tworzeniu, lecz ich wydajność wyszukiwania znacznie spada w przypadku nierównoważonych drzew.