

# Dokumentacja projektu semestralnego

Algorytm binarnego drzewa poszukiwań ze wskaźnikiem do rodzica

Autor: Marcin Chamera

## Spis treści:

1. Wprowadzenie z opisem teoretycznym algorytmu
2. Interfejs
3. Implementacja
4. Wyniki testów

### 1. Wprowadzenie z opisem teoretycznym algorytmu

Binarne drzewo poszukiwań (z ang. Binary Search Tree) to specjalny rodzaj drzewa binarnego w którym wartości wszystkich węzłów lewego poddrzewa jakiegokolwiek węzła w drzewie są mniejsze niż wartość tego węzła. Dodatkowo, wartości wszystkich węzłów prawego poddrzewa jakiegokolwiek węzła są większe niż wartość tego węzła. Algorytm BST przedstawiony na laboratoriach został przeze mnie dodatkowo rozszerzony o wskaźnik do rodzica węzła.

### 2. Interfejs

Program został napisany w celu zaprezentowania działania algorytmu i na wyjściu wyświetla jedynie wyniki metod, jakie zostały wywołane w głównej funkcji programu (`__main__`).

### 3. Implementacja

Algorytm, który opisuje ta dokumentacja, został zaimplementowany przy pomocy dwóch klas: `Node` i `BinarySearchTree`. Na poniższych zrzutach ekranu opisane pola i metody tych klas.

```
1 ▼ class Node:
2     '''Klasa reprezentująca węzeł drzewa.'''
3
4 ▼ def __init__(self, data=None, left=None, right=None, parent=None):
5     self.data = data
6     self.right = None
7     self.left = None
8     self.parent = None
9
10    def __str__(self):
11        return str(self.data)
```

W tej implementacji algorytmu wszystkie mechaniki związane z drzewem zostały przeniesione do klasy `BinarySearchTree`. Klasa `Node` służy jedynie do reprezentacji pojedynczego węzła należącego do drzewa. Posiada ona 4 pola:

- `data` – przechowuje jakąś informację zapisaną w węźle (np. liczbę całkowitą, napis etc.)
- `right` – „wskaźnik” do prawego potomka
- `left` – „wskaźnik” do lewego potomka
- `parent` – „wskaźnik” do rodzica

Klasa `Node` zaimplementowana posiada jedynie metody specjalne `__init__`, czyli konstruktor oraz `__str__`, która zamienia klasę na „nieformalny” string.

```

13 class BinarySearchTree:
14     '''Klasa reprezentująca binarne drzewo poszukiwań.'''
15
16     def __init__(self):
17         self.root = None

```

Klasa BinarySearchTree posiada zaimplementowaną jedną metodę specjalną, czyli jej konstruktor `__init__`. Klasa posiada także jedno pole nazwane `root`, które reprezentuje korzeń tego drzewa. Wewnątrz konstruktora korzeń zostaje ustawiony na `None`, to znaczy drzewo zostaje zainicjowane bez żadnego przypisanego do niego węzła.

W dalszej części dokumentacji przy podawaniu ilości przekazywanych parametrów do metody nie będę wspominał o parametrze `self`.

```

19 ▼ def minimum(self, x):
20     '''Zwraca węzeł o najmniejszej wartości 'data' w drzewie o korzeniu x.'''
21     while x.left != None:
22         x = x.left
23     return x

```

Metoda `minimum` wyszukuje w drzewie węzeł o najmniejszej wartości. Przyjmuje jeden parametr `x`, który jest korzeniem wybranego drzewa, w którym szukamy węzła o minimalnej wartości `data`. W pętli `while` przechodzimy kolejno do lewych potomków obecnego węzła `x`, dopóki nie znajdziemy skrajnie lewego potomka w wybranym drzewie. Metoda na koniec zwraca skrajnie lewego potomka.

```

25 def insert(self, node):
26     '''Wstawia węzeł 'node' do drzewa w odpowiednie miejsce.'''
27     y = None
28     temp = self.root
29     while temp != None:
30         y = temp
31         if node.data < temp.data:
32             temp = temp.left
33         else:
34             temp = temp.right
35
36     node.parent = y
37
38     if y == None:
39         self.root = node
40     elif node.data < y.data:
41         y.left = node
42     else:
43         y.right = node

```

Metoda `insert` wstawia nowy węzeł do drzewa. W binarnym drzewie poszukiwań wstawiany węzeł nie może zostać umieszczony w dowolnym miejscu ze względu

na konieczność zachowania struktury takiego drzewa. W celu wstawienia elementu, najpierw należy znaleźć miejsce, w którym struktura drzewa zostanie zachowana. W tym celu został użyty wskaźnik temp, który kolejno przechodzi przez potomków, zaczynając od korzenia. Ostatni węzeł w iteracji pętli należy uczynić rodzicem nowo wstawianego węzła. W tym celu została użyta zmienna y. W wypadku, gdy drzewo nie ma żadnego węzła, nowy węzeł będzie korzeniem tego drzewa, a wskaźnik jego rodzica nie będzie na nic pokazywał. Dlatego na początku wartość y zostaje ustawiona na None. Na końcu algorytm ustawia nowy węzeł jako dziecko y. Jeśli y jest nulle, nowy węzeł będzie korzeniem drzewa. W innym wypadku następuje sprawdzenie czy wartość w data nowego węzła jest większa czy mniejsza niż wartość w data y i odpowiednio będzie on lewym lub prawym dzieckiem.

```
45 def count(self, node):
46     '''Zwraca liczbę węzłów w drzewie.'''
47     if not node:
48         return 0
49     return 1 + self.count(node.left) + self.count(node.right)
```

Metoda count zwraca liczbę węzłów w drzewie. Przyjmuje jeden parametr node. Przy pierwszym wywołaniu metody jako parametr node podaje się korzeń drzewa. W wypadku, gdyby korzeń był pusty, metoda zwraca 0. W przeciwnym wypadku, metoda działa rekurencyjnie. Do liczby obecnie zliczonych węzłów dodaje 1 i wywołuje się dla lewego i prawego potomka węzła node. Gdy wszystkie wywołania metody będą już dla nieistniejących węzłów, zwracana jest liczba zliczonych węzłów.

```
51 def search(self, node, data):
52     '''Szuka w drzewie węzeł o podanej wartości 'data'.'''
53     if node.data == data:
54         return node
55     if data < node.data:
56         if node.left:
57             return self.search(node.left, data)
58     else:
59         if node.right:
60             return self.search(node.right, data)
61     return None
```

Metoda search szuka w drzewie węzeł o danej wartości. Przyjmuje dwa parametry node i data. Metoda działa rekurencyjnie, gdzie przy pierwszym wywołaniu jako parametr node podawany jest korzeń drzewa. Data to szukana wartość. W przypadku, gdy wartość data obecnie rozpatrywanego węzła jest równa wartości data przekazanej w parametrze, ten węzeł jest zwracany. W przypadku, gdy wartość data obecnie rozpatrywanego węzła jest większa od wartości data przekazanej w parametrze, metoda wywołuje siebie samą z tym samym

parametrem data, a jako parametr node przekazywany jest lewy potomek obecnego węzła. W przypadku, gdy wartość data obecnie rozpatrywanego węzła jest mniejsza od wartości data przekazanej w parametrze, metoda wywołuje siebie samą z tym samym parametrem data, a jako parametr node przekazywany jest prawy potomek obecnego węzła. W innym przypadku zwracana jest wartość None.

```
76 def remove(self, data):
77     '''Usuwa z drzewa węzeł o podanej wartości 'data'. '''
78     z = self.search(self.root, data)
79     if z is None:
80         raise Exception('W tym drzewie nie ma wezla o takiej wartosci.')
81     if z.left == None:
82         self.transplant(z, z.right)
83
84     elif z.right == None:
85         self.transplant(z, z.left)
86
87     else:
88         y = self.minimum(z.right) #minimum element in right subtree
89         if y.parent != z:
90             self.transplant(y, y.right)
91             y.right = z.right
92             y.right.parent = y
93
94         self.transplant(z, y)
95         y.left = z.left
96         y.left.parent = y
97     return z
```

Metoda remove usuwa z drzewa wybrany węzeł. Metoda przyjmuje jeden parametr data, który jest wartością, jaką ma węzeł do usunięcia. Do usunięcia węzła z BST wykorzystana została dodatkowo metoda transplant, która pozwoli zamienić wybrane poddrzewo z innym. Innymi słowy, przeszczepi jedno poddrzewo w miejsce innego.

```
63 def transplant(self, u, v):
64     '''Przekleja poddrzewo zakorzenie w węźle v w miejsce poddrzewa
65     zakorzonego w węźle u. '''
66     if u.parent == None:
67         self.root = v
68     elif u == u.parent.left:
69         u.parent.left = v
70     else:
71         u.parent.right = v
72
73     if v != None:
74         v.parent = u.parent
```

Metoda transplant przyjmuje dwa parametry u i v, gdzie v jest korzeniem drzewa, które będzie przeklejone, a u jest korzeniem drzewa, które zostanie zastąpione.

Zadaniem metody `transplant` jest sprawić, aby węzeł `v` stał się potomkiem rodzica węzła `v`, to znaczy jeśli `u` jest lewym potomkiem, to `v` zostanie lewym potomkiem rodzica `u`. Podobnie, jeśli `u` jest prawym potomkiem, wtedy `v` zostanie prawym potomkiem rodzica `u`. Istnieje też możliwość, że `u` nie ma żadnego rodzica, to znaczy `u` jest korzeniem drzewa. W tym przypadku, `v` zostanie korzeniem tego drzewa. Wracając do metody `remove`. Zakładając, że węzeł do usunięcia jest liściem, łatwo można ten węzeł usunąć, ustawiając rodzica tego węzła na `None`. Aby usunąć węzeł z tylko jednym potomkiem przeszczepiamy tego potomka do węzła, co nie zakłóci struktury BST. W przypadku, gdy węzeł do usunięcia ma obu potomków, szukamy najmniejszego elementu prawego poddrzewa tego węzła i podmieniamy go z węzłem do usunięcia. Najmniejszy element prawego poddrzewa będzie albo nie miał potomków, albo będzie miał jednego potomka ponieważ jeśli ma lewego potomka, to wtedy nie będzie najmniejszym elementem. Z tego powodu taki węzeł można spokojnie usunąć.

Przypominając jeszcze raz metodę `remove`:

```
76 def remove(self, data):
77     '''Usuwa z drzewa węzeł o podanej wartości 'data'.'''
78     z = self.search(self.root, data)
79     if z is None:
80         raise Exception('W tym drzewie nie ma wezla o takiej wartosci.')
81     if z.left == None:
82         self.transplant(z, z.right)
83
84     elif z.right == None:
85         self.transplant(z, z.left)
86
87     else:
88         y = self.minimum(z.right)
89         if y.parent != z:
90             self.transplant(y, y.right)
91             y.right = z.right
92             y.right.parent = y
93
94         self.transplant(z, y)
95         y.left = z.left
96         y.left.parent = y
97     return z
```

Na początku wyszukiwany jest węzeł o wartości podanej w parametrze. Następnie sprawdzana jest liczba potomków węzła `z`. Jeśli węzeł `z` nie ma lewego potomka, wtedy węzeł ten ma albo wyłącznie prawego potomka, albo nie ma ich wcale. W obu tych wypadkach, przy pomocy metody `transplant`, przeszczepiany jest do niego jego prawy potomek. Podobnie sprawdzane jest, czy węzeł posiada prawego potomka, czy nie i wykonywane jest przeszczepienie jego lewego potomka w miejsce tego węzła. Jeśli żaden z powyższych warunków nie jest spełniony, oznacza to, że węzeł `z` ma obu potomków. Znajdowany jest wtedy węzeł `z`

minimalną wartością w prawym poddrzewie i przypisywany do zmiennej y. Teraz należy wstawić węzeł y w miejsce z. Najpierw przeklejany jest prawy potomek y w miejsce y, a następnie brane jest prawe poddrzewo węzła z i przeszczepienie go w taki sposób, aby było prawym poddrzewem y. Po tym operacjach, y przeszczepiany jest w miejsce z. Możliwe jest także, że najmniejszy węzeł jest bezpośrednim potomkiem węzła z. W tym wypadku, jedynie y jest przeszczepiany w miejsce z. Po tym, lewy potomek węzła y zostaje lewym potomkiem węzła z.

```
99 ▼ def inorder(self, node):
100     '''Przechodzi przez drzewo w kolejności poprzecznej (in-order)
101     i wyświetla wartość, i rodzica każdego z odwiedzonych węzłów.'''
102 ▼     if node:
103         self.inorder(node.left)
104         print('Węzeł:', node.data, ', Rodzic:',
105               node.parent.data if node.parent is not None else 'None')
106         self.inorder(node.right)
```

Metoda inorder przechodzi przez drzewo w kolejności poprzecznej (in-order) i wyświetla wartość data, i wartość data rodzica każdego z odwiedzonych węzłów. Metoda przyjmuje jeden parametr node, dla którego w ciele metody zostanie wyświetlona wartość jego pola data. Najpierw sprawdzany jest warunek, czy taki węzeł istnieje. Jeśli tak, to najpierw wywoływana jest rekurencyjnie ta metoda, gdzie w parametrze przekazywany jest lewy potomek. Następnie wyświetlana jest wartość data węzła node i wartość data rodzica węzła node, ale tylko jeśli rodzic istnieje (czyli jeśli node nie jest korzeniem). Na końcu wywoływana jest rekurencyjnie ta metoda, gdzie w parametrze przekazywany jest prawy potomek.

#### 4. Wyniki testów

Algorytm został poddany testowi poprawności, które zostały umieszczone w bloku instrukcji warunkowej `if __name == '__main__':`. Testy zostały przeprowadzone, wywołując funkcje dla pewnych konkretnych danych przedstawionych na poniższych zrzutach ekranu.

```

108 if __name__ == '__main__':
109     t = BinarySearchTree()
110
111     a = Node(10)
112     b = Node(20)
113     c = Node(30)
114     d = Node(100)
115     e = Node(90)
116     f = Node(40)
117     g = Node(50)
118     h = Node(60)
119     i = Node(70)
120     j = Node(80)
121     k = Node(150)
122     l = Node(110)
123     m = Node(120)
124
125     nodes = [a, b, c, d, e, f, g, h, i, j, k, l, m]

```

Na początku tworzona jest instancja klasy BinarySearchTree, a po niej paręnaście instancji klasy Node z przykładowymi wartościami. Te instancje klasy Node (węzły) zostały wstawione do listy nodes.

```

127 for node in nodes:
128     t.insert(node)
129
130 print('Po dodaniu wezlow:')
131 t.inorder(t.root)

```

Następnie w pętli węzły wstawiane są do drzewa, po czym wywołana zostaje metoda inorder w celu wyświetlenia aktualnego wyglądu drzewa.

```

Po dodaniu wezlow:
Wezel: 10 , Rodzic: None
Wezel: 20 , Rodzic: 10
Wezel: 30 , Rodzic: 20
Wezel: 40 , Rodzic: 90
Wezel: 50 , Rodzic: 40
Wezel: 60 , Rodzic: 50
Wezel: 70 , Rodzic: 60
Wezel: 80 , Rodzic: 70
Wezel: 90 , Rodzic: 100
Wezel: 100 , Rodzic: 30
Wezel: 110 , Rodzic: 150
Wezel: 120 , Rodzic: 110
Wezel: 150 , Rodzic: 100

```

Tak prezentuje się wynik powyższych instrukcji.

```

133 print('Usunieto wezel o wartosci ', t.remove(10))
134 print('Usunieto wezel o wartosci ', t.remove(120))
135 not_exst_node_val = 130
136 try:
137     t.remove(not_exst_node_val)
138 except Exception as e:
139     z = e
140     print('Proba usuniecia wezla o wartosci', not_exst_node_val)
141     print(z)

```



Następnie wywołana zostaje dwa razy metoda `remove`, a jej wynik zostaje wyświetlony na ekran. Później następuje próba usunięcia węzła o takiej wartości, która nie istnieje w stworzonym drzewie.

```
Proba usuniecia wezla o wartosci 130
W tym drzewie nie ma wezla o takiej wartosci.
```

Tak prezentuje się wynik powyższych instrukcji.

```
143     print('Po usuwaniu wezlow:')
144     t.inorder(t.root)
145
146     print('Liczba wezlow w drzewie:', t.count(t.root))
```

Na końcu zostaje jeszcze raz wywołana metoda `inorder`, w celu wyświetlenia zawartości drzewa po usunięciu dwóch węzłów, a następnie zostaje wywołana metoda `count`.

```
Po usuwaniu wezlow:
Wezel: 20 , Rodzic: None
Wezel: 30 , Rodzic: 20
Wezel: 40 , Rodzic: 90
Wezel: 50 , Rodzic: 40
Wezel: 60 , Rodzic: 50
Wezel: 70 , Rodzic: 60
Wezel: 80 , Rodzic: 70
Wezel: 90 , Rodzic: 100
Wezel: 100 , Rodzic: 30
Wezel: 110 , Rodzic: 150
Wezel: 150 , Rodzic: 100
Liczba wezlow w drzewie: 11
```

Tak prezentuje się wynik powyższych instrukcji.