Dokumentacja projektu semestralnego

Algorytm binarnego drzewa poszukiwań ze wskaźnikiem do rodzica

Autor: Marcin Chamera

Spis treści:

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

```
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)
```

opisane pola i metody tych klas.

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

Klasa Node zaimplementowane posiada jedynie metody specjalne __init__, czyli

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

konstruktor oraz __str__, która zamienia klasę na „nieformalny” string.

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 nullem, 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      zakorzenionego 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('Wezel:', 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.

```
115     def DSW(self):
116       '''Doprowadza drzewo do postaci zrównoważonej.'''
117       if self.root:
118         self.create_backbone(self.root, self.root)
119         self.create_perfect_BST()
```

Metoda DSW doprowadza drzewo do postaci zrównoważonej powodując, że wysokość drzewa jest rzędu O(logn), gdzie n to liczba węzłów drzewa. Jeśli drzewo nie jest puste, to znaczy jeśli posiada korzeń, wywołuje dwie metody: create_backbone i create_perfect_BST mające za zadanie zrównoważyć drzewo.

```
121     def create_backbone(self, root, top):
122       '''Zamienia drzewo w listę przez wielokrotne użycie metody rotate_right.'''
123       left_child = None
124       parent = top
125       while parent:
126         left_child = parent.left
127         if left_child:
128           root = self.rotate_right(root, parent)
129           parent = left_child
130         else:
131           parent = parent.right
132       self.root = root
```

Metoda create_backbone zamienia drzewo w listę przez wielokrotne użycie metody rotate_right. Takie drzewo sprowadzone do postaci listy zwane jest kręgosłupem. Jeśli zmienna parent posiada lewego potomka, wykonywana jest rotacja tego potomka względem zmiennej parent (czyli lewy potomek zostaje ojcem węzła parent). Zmienna parent zostaje przesunięta wtedy do nowo powstałego rodzica. W wypadku, gdy zmienna parent nie posiadała lewego potomka, zostaje ona przesunięta w miejsce swojego prawego potomka.

```
134   def rotate_right(self, root, top):
135     '''Rotacja wierzchołków w prawo z zachowaniem struktury BST.'''
136     if top.left is None:
137       return root
138     node = top.left
139     top.left = node.right
140     if node.right:
141       node.right.parent = top
142     node.parent = top.parent
143     if top.parent is None:
144       root = node
145     elif top == top.parent.right:
146       top.parent.right = node
147     else:
148       top.patent.left = node
149     node.right = top
150     top.parent = node
151     return root
```

Metoda rotate_right rotuje wierzchołki w prawo z zachowaniem struktury BST. Przyjmuje dwa parametry root i top. Rotacja odbywa się wokół krawędzi pomiędzy węzłami root i top w drzewie.

```
153   def rotate_left(self, root, top):
154     '''Rotacja wierzchołków w lewo z zachowaniem struktury BST.'''
155     if top.right is None:
156       return root, top
157     node = top.right
158     top.right = node.left
159     if node.left:
160       node.left.parent = top
161     node.parent = top.parent
162     if top.parent is None:
163       root = node
164     elif top == top.parent.left:
165       top.parent.left = node
166     else:
167       top.parent.right = node
168     node.left = top
169     top.parent = node
170     return root
```

Metoda rotate_left rotuje wierzchołki w lewo z zachowaniem struktury BST. Przyjmuje dwa parametry root i top. Rotacja odbywa się wokół krawędzi pomiędzy węzłami root i top w drzewie.

```
172   def create_perfect_BST(self):
173     '''Przywraca kształt drzewa poprzez wielokrotne wywołanie metody
174     make_rotations.'''
175     root = self.root
176     n = self.count(self.root)
177     m = int(pow(2, floor(log(n+1, 2)))-1)
178     root = self.make_rotations(root, n - m)
179     while m > 1:
180       m //= 2
181       root = self.make_rotations(root, m)
182     self.root = root
```

Metoda create_perfect_BST przywraca kształt drzewa poprzez wielokrotne wywołanie metody make_rotations. Zmienna m jest liczbą potrzebną do wyznaczenia ilości obrotów przy pierwszym obiegu algorytmu.

```
184     def make_rotations(self, root, bound):
185         '''Wykonuje wielokrotne lewe rotacja na co drugim węźle, względem jego
186         rodzica.'''
187         parent = root
188         for _ in range(bound):
189             if parent:
190                 root = self.rotate_left(root, parent)
191                 if parent.parent:
192                     parent = parent.parent.right
193         return root
```

Metoda make_rotations wykonuje wielokrotne lewe rotacje na co drugim węźle, względem jego rodzica. Metoda przyjmuje dwa parametry root i bound.

```
195     def height(self, root):
196         '''Zwraca wysokość drzewa.'''
197         if root == None:
198             return 0
199         return 1 + max(self.height(root.left), self.height(root.right))
```

Metoda height zwraca wysokość drzewa. W wypadku, gdy drzewo jest puste, zwraca 0. Gdy drzewo nie jest puste wykonuje się rekurencyjnie przechodząc przez poddrzewa lewego i prawego potomka.

4. Wyniki testów

```
201  import unittest
202
203  class TestBinarySearchTree(unittest.TestCase):
204    def setUp(self):
205      self.t = BinarySearchTree()
206      self.nodes = [Node(10), Node(20), Node(30), Node(100), Node(90), Node(40),
207        Node(50), Node(60), Node(70), Node(80), Node(150), Node(110), Node(120)]
208      for node in self.nodes:
209        self.t.insert(node)
210
211    def test_print(self):
212      self.assertEqual(str(Node(10)), '10')
213
214    def test_count(self):
215      self.assertEqual(self.t.count(self.t.root), 13)
216
217    def test_search(self):
218      self.assertTrue(self.t.search(self.t.root, 50))
219      self.assertFalse(self.t.search(self.t.root, 130))
220
221    def test_remove(self):
222      self.t.remove(10)
223      self.assertEqual(self.t.count(self.t.root), 12)
224      self.assertRaises(Exception, lambda: self.t.remove(10))
225
226    def test_height(self):
227      self.assertEqual(self.t.height(self.t.root), 10)
228
229    def test_DSW(self):
230      self.t.DSW()
231      self.assertEqual(self.t.height(self.t.root), 4)
232
233    def tearDown(self): pass
234
235  if __name__ == '__main__':
236      unittest.main()
```

Do zweryfikowania poprawności algorytmu zostały przeprowadzone testy jednostkowe przy pomocy modułu unittest. Przeprowadzone testy zostały przedstawione powyżej na zrzucie ekranu. Poniżej można zobaczyć pomyślny wynik tych testów.

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
[Finished in 0.2s]
```

5. Referencje

https://www.python.org/

https://www.codesdope.com/course/data-structures-binary-search-trees/

http://www.geekviewpoint.com/python/bst/dsw_algorithm

https://pl.wikipedia.org/wiki/Algorytm_DSW

https://ufkapano.github.io/download/Ewelina_Matusiewicz_2017.pdf