SPRAWOZDANIE - LISTA 6

Małgorzata Kowalczyk

Kamil Kowalski

18.01.2022

Zadanie 1

W tym zadaniu naszym celem było stworzenie klasy implementującą binarne drzewa przeszukiwań, która dodatkowo dba o poprawne przetwarzanie powtarzających się kluczy. Skorzystaliśmy z klas `class TreeNode` oraz `class BinarySearchTree` i udoskonaliliśmy je, dodając atrybut counter oraz odpowiednio zmieniając metody `def put(self, key, val)` i `def delete(self, key)` .

Dodatkowo napisaliśmy również metodę `def display_tree(self)` , która wyświetla nam nasze drzewo oraz `def print_BST(self)` , która pokazuje nam jakie klucze, z jakimi wartościami znajdują się na naszym drzewie oraz informuje nas o liczności występowania danego klucza.

Przykładowe działanie naszego programu:

In [2]:
```
bst = BinarySearchTree()
```

In [3]:
```
bst.put(3,123)
bst.put(6,7)
bst.put(4,100)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 3
key: 3 value: 123 counter: 1
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1


--> 3
          --> 4
     --> 6
```

Dodajemy jeszcze raz klucz 3 z tą samą wartością. Counter zwiększa się o jeden.

In [4]:
```
bst.put(3,123)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 3
key: 3 value: 123 counter: 2
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1


--> 3
          --> 4
     --> 6
```

Gdy dodajemy klucz, który znajduje się na naszym drzewie, ale z inną wartością, zmienia się ona na nową. Tutaj klucz 3 posiadał najpierw wartość 123, a po zmianie 1500. Counter zwiększa się o jeden.

In [5]:
```
bst.put(3,1500)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 3
key: 3 value: 1500 counter: 3
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1


--> 3
          --> 4
     --> 6
```

Dodajemy nowy klucz 10 o wartości 200. Nasze drzewo składa się teraz z 4 elementów.

In [6]:
```
bst.put(10, 200)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 4
key: 3 value: 1500 counter: 3
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1
key: 10 value: 200 counter: 1


--> 3
          --> 4
     --> 6
          --> 10
```

Teraz sprawdzimy działanie metody `delete(self, key)` . W tym celu usuwamy klucz 3. Counter zmniejsza się o jeden.

In [7]:
```
bst.delete(3)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 4
key: 3 value: 1500 counter: 2
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1
key: 10 value: 200 counter: 1


--> 3
          --> 4
     --> 6
          --> 10
```

Chcemy pozbyć się 3 z naszego drzewa. Usuwamy więc nasz klucz dwukrotnie. Rozmiar drzewa zmniejsza się o jeden.

In [8]:
```
bst.delete(3)
bst.delete(3)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 3
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1
key: 10 value: 200 counter: 1


     --> 4
--> 6
     --> 10
```

Gdy chcemy usunąć klucz, którego nie ma w naszym drzewie, program zwraca błąd.

In [9]:
```
bst.delete(1)
print("Rozmiar naszego drzewa: " + str(bst.length()))
bst.print_BST()
bst.display_tree()
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-9-ce4d45b54d1a> in <module>
----> 1 bst.delete(1)
      2 print("Rozmiar naszego drzewa: " + str(bst.length()))
      3 bst.print_BST()
      4 bst.display_tree()

<ipython-input-1-dfdfd54c3a54> in delete(self, key)
    186                 self._get(key,self.root).counter -= 1
    187         else:
--> 188             raise KeyError('Error, key "' + str(key) + '" not in tree')
    189 
    190     def __delitem__(self, key):

KeyError: 'Error, key "1" not in tree'
```

Następnie usuwamy wszystkie elementy.

In [10]:
```
bst.delete(10)
bst.delete(4)
bst.delete(6)
print("Rozmiar naszego drzewa: " + str(bst.length()))
```

```
bst.print_BST()
bst.display_tree()
```

```
Rozmiar naszego drzewa: 0
```

Możemy także wizualizować drzewa o większej ilości elementów.

In [10]:
```
bst1 = BinarySearchTree()
bst1.put(3,123)
bst1.put(6,7)
bst1.put(4,100)
bst1.put(10,1)
bst1.put(8,10)
bst1.put(20,5)
bst1.put(14,2)
bst1.put(10,1) #drugi raz ten sam klucz
bst1.put(18,2)
bst1.put(2,9)
bst1.put(7,1)
bst1.put(22,2)
bst1.put(10,1) #trzeci raz ten sam klucz
bst1.put(3,124) #drugi raz ten sam klucz, ale z nową wartością
bst1.put(1,1)
print("Rozmiar naszego drzewa: " + str(bst1.length()))
bst1.print_BST()
bst1.display_tree()
```

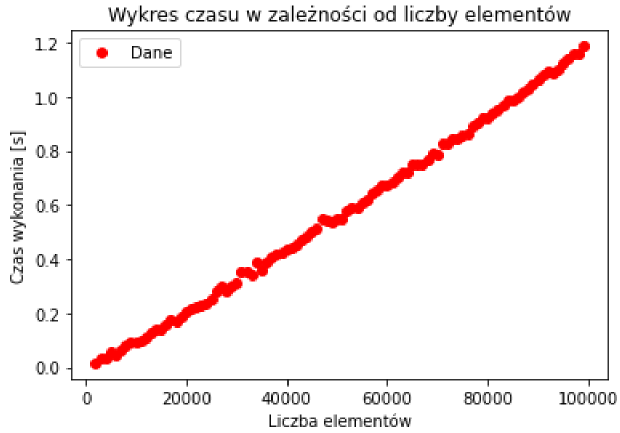
```
Rozmiar naszego drzewa: 12
key: 3 value: 124 counter: 2
key: 2 value: 9 counter: 1
key: 1 value: 1 counter: 1
key: 6 value: 7 counter: 1
key: 4 value: 100 counter: 1
key: 10 value: 1 counter: 3
key: 8 value: 10 counter: 1
key: 7 value: 1 counter: 1
key: 20 value: 5 counter: 1
key: 14 value: 2 counter: 1
key: 18 value: 2 counter: 1
key: 22 value: 2 counter: 1


          --> 1
     --> 2
--> 3
          --> 4
     --> 6
                    --> 7
               --> 8
          --> 10
                    --> 14
                         --> 18
               --> 20
                    --> 22
```

Zadanie 2

Analiza eksperymentalna

Będziemy chcieli pokazać, że funkcja `def sortHeap(data_list)` sortuje listę w czasie $O(n \log n)$. W tym celu, na początku zbadajmy czas wykonywania funkcji w zależności od długości listy. Wyniki zapisaliśmy w pliku *execution_times.csv*.

In [26]:
```
n = [1000 * i for i in range(2, 100)]
execution_times = []
with open('execution_times.csv', newline='') as f:
    reader = csv.reader(f)
    execution_times = [float(i) for i in list(reader)[0]]
plot(n, execution_times)
```

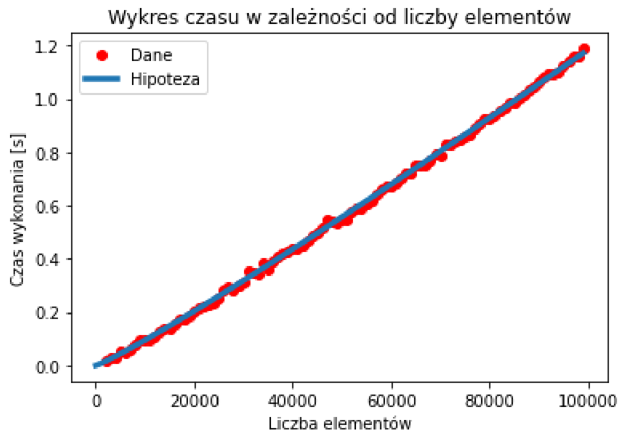


Ponieważ spodziewamy się, czasu $O(n \log n)$, dane na wykresie przybliżamy funkcją

$$T(N) = a \cdot N \log n. \tag{1}$$

Wykorzystujemy funkcję *curve_fit* z pakietu *scipy*, aby obliczyć wartość a oraz sprawdzić, czy przybliżenie naszej funkcji jest właściwe.

In [27]:
```
popt, pcov = curve_fit(fit_fun, n, execution_times)
hypothesis_plot(n, execution_times, fit_fun, popt)
print(popt[0])
```




```
1.0286939332989803e-06
```

Jak widzimy, udało się znaleźć pasującą funkcję. Wynika stąd, że czas wykonania naszego algorytmu opisuje funkcja

$$T(N) = 0,0000010286939 \cdot N \log N. \tag{2}$$

Zadanie 3

W tym zadaniu mieliśmy zaimplementować kopiec binarny o ograniczonej wielkości n - czyli przechowujący n najważniejszych (największych) wartości. Skorzystaliśmy z klasy `class BinHeap` podanej na wykładzie odpowiednio modyfikując funkcję `def insert(self, value)` oraz `def build_heap(self, alist)` .

Przykładowe działanie naszego programu:

Tworzymy kopiec o maksymalnym rozmiarze równym 6. Na początku budujemy go tylko z 5 elementami.

In [13]:
```
bin_heap = BinHeap(6)
print("Czy kopiec jest pusty: "+str(bin_heap.is_empty()))
```

```
Czy kopiec jest pusty: True
```

In [14]:
```
bin_heap.build_heap([9,5,16,2,8])
print(bin_heap)
print("Rozmiar naszego kopca: " + str(bin_heap.size()))
print("Czy kopiec jest pusty: "+str(bin_heap.is_empty()))
```

```
[2, 5, 16, 9, 8]
Rozmiar naszego kopca: 5
Czy kopiec jest pusty: False
```

Przy pomocy `insert(self, value)` dodajemy kolejny element - 12. Mamy pełny kopiec składający się z 6 elementów.

In [15]:
```
bin_heap.insert(12)
```

```
print(bin_heap)
print("Rozmiar naszego kopca: " + str(bin_heap.size()))
```

```
[2, 5, 12, 9, 8, 16]
Rozmiar naszego kopca: 6
```

Teraz spróbujemy dodać element - 7. Jest większy niż najmniejszy element na kopcu, zatem zastąpi on 2. Teraz najmniejszą wartością będzie 5, a rozmiar pozostaje niezmienony.

In [16]:
```
bin_heap.insert(7)
print(bin_heap)
print("Rozmiar naszego kopca: " + str(bin_heap.size()), end = " ")
```

```
[5, 8, 7, 9, 16, 12]
Rozmiar naszego kopca: 6
```

Gdy będziemy dodawać element mniejszy, niż wszystkie wartości na kopcu, elementy pozostaną na nim niezmienione.

In [17]:
```
bin_heap.insert(1)
print(bin_heap)
print("Rozmiar naszego kopca: " + str(bin_heap.size()), end = " ")
```

```
[5, 8, 7, 9, 16, 12]
Rozmiar naszego kopca: 6
```

Następnie usuniemy najmniejszą wartość przy pomocy metody `del_min(self)` .

In [18]:
```
bin_heap.del_min()
print(bin_heap)
print("Rozmiar naszego kopca: " + str(bin_heap.size()), end=" ")
```

```
[7, 8, 12, 9, 16]
Rozmiar naszego kopca: 5
```

Z racji tego, że teraz nasz kopiec nie jest pełny, możemy dodać mniejszy element np. 1.

In [19]:
```
bin_heap.insert(1)
print(bin_heap)
print("Rozmiar naszego kopca: " + str(bin_heap.size()), end=" ")
```

```
[1, 8, 7, 9, 16, 12]
Rozmiar naszego kopca: 6
```

Gdy od razu tworzymy kopiec z większą liczbą elementów, niż jest to dozwolone, program od razu zwraca n największych wartości.

Dla $n = 5$

In [20]:
```
bin_heap2=BinHeap(5)
bin_heap2.build_heap([9,5,6,2,8,1])
print(bin_heap2)
print("Rozmiar naszego kopca: " + str(bin_heap2.size()), end=" ")
```

```
[2, 5, 6, 9, 8]
Rozmiar naszego kopca: 5
```

Dla $n = 2$

In [21]:
```
bin_heap3=BinHeap(2)
bin_heap3.build_heap([50,3,13,9,10,7,15,4])
print(bin_heap3)
print("Rozmiar naszego kopca: " + str(bin_heap3.size()), end=" ")
```

```
[15, 50]
Rozmiar naszego kopca: 2
```

Dla $n = 5$

In [22]:
```
bin_heap4=BinHeap(5)
bin_heap4.build_heap([50,3,13,9,10,7,15,4])
print(bin_heap4)
print("Rozmiar naszego kopca: " + str(bin_heap4.size()), end=" ")
```

```
[9, 10, 13, 50, 15]
Rozmiar naszego kopca: 5
```

Dla $n = 7$

In [45]:
```
bin_heap5=BinHeap(7)
bin_heap5.build_heap([50,3,13,9,10,7,15,4])
print(bin_heap5)
print("Rozmiar naszego kopca: " + str(bin_heap5.size()), end=" ")
```

```
[4, 9, 7, 50, 10, 15, 13]
Rozmiar naszego kopca: 7
```

Zadanie 4

W tym zadaniu naszym celem było napisanie programu liczącego pochodne funkcji. Jest on stworzony w oparciu o drzewa wyprowadzenia. Założenia:

- Program obsługuje symbole +, -, *, /, ^.
- W programie można obliczać pochodne funkcji $\sin, \cos, \exp, \ln$.
- W wyrażeniach funkcji można stosować liczby całkowite lub litery np. $F(x) = a \cdot x + b$.
- Po wpisaniu wzoru funkcji, powinniśmy wpisać zmienną według której program ma obliczyć pochodną.
- Można stosować dowolne złożenia wyżej wymienionych funkcji, np. $\cos(\sin(x^2))$

W programie znajdują się następujące funkcje i klasy:

- `class Stack` - Standardowa klasa implementująca stos, używana już w poprzednich listach.
- `class Binary_tree` - Klasa implementująca drzewa binarne omawiana na wykładzie, dodatkowo dodaliśmy metody `def insert_right_tree(self, tree)` oraz `def insert_left_tree(self, tree)`, które umożliwiają bezpośrednie dołączanie drzewa z lewej lub prawej strony.
- `def remove_blank_space(list)` - Funkcja zmieniająca zawartość listy *list*. Jeśli znajdują się w niej puste nawiasy, to zostaną one usunięte.
- `def unpack_list(tree_list)` - Funkcja poprawiająca czytelność wyświetlanego wyniku końcowego. Domyślnie każdy z elementów drzewa zostaje opakowany w nawias np. $((5) + (x))$, po uruchomieniu tej funkcji wyrażenie jest wyświetlane jako $(5 + x)$.
- `def print_function(tree)` - Funkcja przetwarzająca drzewo na zapis matematyczny w postaci listy, gdzie każdy element jest osobnym symbolem.
- `def parse_function(expresion)` - Funkcja, która przetwarza podaną jako string funkcję matematyczną na listę, którą później można przekształcić na drzewo.
- `def build_tree(parsed_function)` - Funkcja tworząca drzewo z poszczególnych elementów wyrażenia matematycznego podanych w liście. Symbole na liście podane są w kolejności naturalnego zapisania funkcji np. `['x', '+', '5']`
- `def cut_parsed_list(parsed_list)` - Wykorzystujemy ją, gdy zamiast całego drzewa chcemy wziąć tylko jego fragment. Funkcja skraca listę do takiej z której możemy zbudować mniejsze drzewo. Elementy na liście podane są według kolejności preorder.
- `def build_tree_from_parsed(parsed_list)` - Funkcja tworząca drzewo z poszczególnych elementów wyrażenia matematycznego podanych w liście. Symbole na liście podane są w kolejności preorder np. `['+', 'x', '5']`.
- `def differential_tree(tree, symbol)` - Główna funkcja przetwarzająca listę z elementami wyrażenia matematycznego na drzewo funkcji pochodnej.
- `def calculate_derivative(function)` - Funkcja pomocnicza wywołująca funkcje w odpowiedniej kolejności.

Opis działania programu

Na początku parsujemy podanego przez użytkownika stringa, funkcja `parse_function` tworzy listę z odpowiednio rozdzielonymi wyrażeniami. Funkcja musi rozpoznać, kiedy w wyrażeniu pojawiają się takie wyrażenia jak $\sin, \cos, \exp, \ln$ lub liczby.
Następnym krokiem jest stworzenie drzewo wprowadzenia z powstałej listy. Przy pomocy stosu, tworzymy drzewo w podobny sposób, jak było to prezentowane na wykładzie. Dodatkowo musimy zwrócić uwagę, że poza operatorami dwuargumentowymi w naszej funkcji mogą wystąpić operatory jednoargumentowe takie jak $\sin, \cos, \exp, \ln$.
Stworzone drzewo wprowadzenia posłuży nam przy tworzeniu drzewa pochodnej, które jest wynikiem wywołania funkcji `differential_tree`. Na początku tworzymy listę preorder_tree, w której przechowujemy elementy naszego wyrażenia w kolejności preorder. Następnie przechodzimy po elementach tej listy i w zależności od symbolu modyfikujemy drzewo pochodnej.
Często potrzebujemy znać pochodną tylko części naszego wyrażenia. W takich przypadkach, dzięki kolejności preorder, jesteśmy w stanie skrócić naszą listę (funkcja `cut_parsed_list`) i następnie zbudować z niej mniejsze drzewo (funkcja `build_tree_from_parsed`), które możemy przekształcić na pochodną.
Po stworzeniu drzewa pochodnej, sprowadzamy je z powrotem do listy dzięki funkcji `print_function` oraz usuwamy część niepotrzebnych nawiasów (funkcje `remove_blank_space` oraz `unpack_list`).
Poniżej zamieszczamy przykładowe działania naszego programu.

In [69]:
```
calculate_derivative(('cos(sin(((x^2))))', 'x'))
```

```
F(x)= (cos((sin(((x^2))))))
F'(x)= ((-1*(sin(sin)))*((cos)*((1*(x^(2-1)))*2)))
```

In [49]:
```
calculate_derivative(('exp(y^2)+(5*y)', 'y'))
```

```
F(y)= ((exp(y^2))+(5*y))
F'(y)= (((exp(y^2))*((1*(y^(2-1)))*2))+((0*y)+(5*1)))
```

In [62]:
```
calculate_derivative(('cos(sin(z))', 'z'))
```
```
F(z)= (cos((sin(z+1))))
F'(z)= ((-1*(sin(sin(z+1))))*((cos(z+1))*(1+0)))
```

In [50]:
```
calculate_derivative(('exp(x^2)', 'x'))
```
```
F(x)= (exp(x^2))
F'(x)= ((exp(x^2))*((1*(x^(2-1)))*2))
```

In [52]:
```
calculate_derivative(('(sin(x))/(exp(x))', 'x'))
```
```
F(x)= ((sin(x))/((exp(x))))
F'(x)= (((((cosx)*1)*(expx))-((sinx)*((expx)*1)))/((expx)^2))
```

In [53]:
```
calculate_derivative(('ln(z^2)+5', 'z'))
```
```
F(z)= ((ln(z^2))+5)
F'(z)= (((1/(z^2))*((1*(z^(2-1)))*2))+0)
```

In [54]:
```
calculate_derivative(('((x^2)+5)^10', 'x'))
```
```
F(x)= (((x^2)+5)^10)
F'(x)= (((((1*(x^(2-1)))*2)+0)*(((x^2)+5)^(10-1)))*10)
```

In [55]:
```
calculate_derivative(('((x^a)+b)^c', 'x'))
```
```
F(x)= (((x^a)+b)^c)
F'(x)= (((((1*(x^(a-1)))*a)+0)*(((x^a)+b)^(c-1)))*c)
```

In [56]:
```
calculate_derivative(('(a*5)*(a*x)', 'a'))
```
```
F(a)= ((a*5)*(a*x))
F'(a)= ((((1*5)+(a*0))*(a*x))+((a*5)*((1*x)+(a*0))))
```

In [57]:
```
calculate_derivative(('(9*(x^3))+(8*(x^2))+(7*(2*x))+(6*x)', 'x'))
```
```
F(x)= ((((9*(x^3))+(8*(x^2)))+(7*(2*x)))+(6*x))
F'(x)= (((((0*(x^3))+(9*((1*(x^(3-1)))*3)))+((0*(x^2))+(8*((1*(x^(2-1)))*2))))+((0*(2*x))+(7*((0*x)+(2*1)))))
+((0*x)+(6*1)))
```

In [58]:
```
calculate_derivative(('sin((x^3)+2)', 'x'))
```
```
F(x)= (sin((x^3)+2))
F'(x)= ((cos((x^3)))*(((1*(x^(3-1)))*3)+0))
```

In [60]:
```
calculate_derivative(('cos(x+(2*x))+3', 'x'))
```
```
F(x)= ((cos(x+(2*x)))+3)
F'(x)= (((-1*(sin(x+(2*x))))*(1+((0*x)+(2*1))))+0)
```

Linki

https://github.com/github-kamilk/AiSD/tree/main/Lista_6