Algorytmy i stuktury danych - Lista 6

Alicja Myśliwiec - gr. wtorek 7:30

In [2]:
```python
import numpy as np
import random
import time
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

Zad. 1 Stwórz własną klasę implementującą binarne drzewa przeszukiwań. Zadbaj o poprawne przetwarzanie powtarzających się kluczy.

In [3]:
```python
class TreeNode:
    def __init__(self, key, value='default', right_child=None, left_child=None, parent=None):
        self.key = key
        self.payload = [value, 0]  # [wartość; liczba pokazuje ile razy dany klucz został wprowadzony]
        self.rc = right_child
        self.lc = left_child
        self.parent = parent

    def has_rc(self):
        return self.rc

    def is_rc(self):
        return self.parent and (self.parent.rc == self)

    def has_lc(self):
        return self.lc

    def is_lc(self):
        return self.parent and (self.parent.lc == self)

    def is_root(self):
        return not self.parent

    def is_leaf(self):
        return not (self.rc or self.lc)

    def has_any_children(self):
        return self.rc or self.lc

    def has_both_children(self):
        return self.rc and self.lc

    def splice_out(self):
        if self.is_leaf():
            if self.is_lc():
                self.parent.lc = None
            else:
                self.parent.rc = None
        elif self.has_any_children():
            if self.has_lc():
                if self.is_lc():
                    self.parent.lc = self.lc
            else:
                self.parent.rc = self.rc
                self.rc.parent = self.parent
        elif self is not None:
            if self.is_lc():
                self.parent.lc = self.rc
            else:
                self.parent.rc = self.rc
            self.rc.parent = self.parent

    def find_min(self):
        current = self
        while current.has_lc():
            current = current.lc
        return current

    def find_successor(self):
        successor = None
        if self.has_rc():
            successor = self.rc.find_min()
        else:
            if self.parent:
                if self.is_lc():
                    successor = self.parent
                else:
                    self.parent.rc = None
                    successor = self.parent.find_successor()
                    self.parent.rc = self
        return successor

    def replace_node_data(self, key, value, lc, rc):
        self.key = key
        self.payload = value
        self.lc = lc
        self.rc = rc
        if self.has_lc():
            self.lc.parent = self
        if self.has_rc():
            self.rc.parent = self

# ----- funckja dodana--------

    def is_overload(self):
        """
        Funkcja sprawdza, czy dany klucz został wprowadzony więcej niż raz. Jeśli tak, zwraca 'True'.
        """
        return self.payload[1] > 0
```

In [117…
```python
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()

    def put(self, key, val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = TreeNode(key, val)
        self.size += 1

    def _put(self, key, val, current_node):
        if key < current_node.key:
            if current_node.has_lc():
                self._put(key, val, current_node.lc)
            else:
                current_node.lc = TreeNode(key, value=val, parent=current_node)
        elif key == current_node.key:  # nadpisanie klucza
            current_node.payload[0] = val
            current_node.payload[1] += 1
        else:
            if current_node.has_rc():
                self._put(key, val, current_node.rc)
            else:
                current_node.rc = TreeNode(key, value=val, parent=current_node)

    def __setitem__(self, key, val):
        self.put(key, val=val)

    def get(self, key):
        if self.root:
            res = self.find_node(key, self.root)
            if res:
                return res.payload[0]
            else:
                return None
        else:
            return None

    def get_count(self, key):
        if self.root:
            res = self.find_node(key, self.root)
            if res:
                return res.payload[1]
            else:
                return None
        else:
            return None

    def find_node(self, key, current_node):
        if not current_node:
            return None
        elif current_node.key == key:
            return current_node
        elif key < current_node.key:
            return self.find_node(key, current_node.lc)
        else:
            return self.find_node(key, current_node.rc)

    def __getitem__(self, key):
        return self.get(key)

    def __contains__(self, key):
        if self.find_node(key, self.root):
            return True
        return False

    def delete(self, key):
        if self.size > 1:
            node_to_remove = self.find_node(key, self.root)
            if node_to_remove:
                if node_to_remove.is_overload():
                    node_to_remove.payload[1] -= 1
                else:
                    self.remove(node_to_remove)
                    self.size -= 1
            else:
                raise KeyError('Key not in the tree')
        elif self.size == 1 and self.root.key == key:
            if self.root.is_overload():
                self.root.payload[1] -= 1
            else:
                self.root = None
                self.size -= 1
        else:
            raise KeyError('Key not in the tree')

    def remove(self, current_node):
        if current_node.is_leaf():
            if current_node == current_node.parent.lc:
                current_node.parent.lc = None
            else:
                current_node.parent.rc = None
        elif current_node.has_both_children():
            successor = current_node.find_successor()
            successor.splice_out()
            current_node.key = successor.key
            current_node.payload = successor.payload
        else:
            if current_node.has_lc():
                if current_node.is_lc():
                    current_node.lc.parent = current_node.parent
                    current_node.parent.lc = current_node.lc
                elif current_node.is_rc():
                    current_node.lc.parent = current_node.parent
                    current_node.parent.rc = current_node.lc
                else:
                    current_node.replace_node_data(current_node.lc.key, current_node.lc.payload,
                                                   current_node.lc.lc, current_node.lc.rc)
            else:
                if current_node.is_lc():
                    current_node.rc.parent = current_node.parent
                    current_node.parent.lc = current_node.rc
                elif current_node.is_rc():
                    current_node.rc.parent = current_node.parent
                    current_node.parent.rc = current_node.rc
                else:
                    current_node.replace_node_data(current_node.rc.key, current_node.rc.payload,
                                                   current_node.rc.lc, current_node.rc.rc)

    def __delitem__(self, key):
        self.delete(key)

# ---------------------- wizualizacja -----------------------

    def show_tree(self, current_node, list_to_print=None, lvl=0):
        if list_to_print is None:
            list_to_print = []

        if current_node is not None:
            self.show_tree(current_node.rc, list_to_print, lvl + 1)
            text = "{}---{}".format('      ' * lvl, current_node.key)
            list_to_print.append(text)
            self.show_tree(current_node.lc, list_to_print, lvl + 1)
            return list_to_print
        else:
            pass

    def print_tree(self):
        if self.root:
            list_to_print = self.show_tree(self.root)
            for line in list_to_print:
                print(line)
        else:
            raise NameError("There's no tree to show here yet")
```

In [103…
```python
tree = BinarySearchTree()
key_list = [7, 4, 9, 5, 8, 1, 3]
value_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
for i in range(len(key_list)):
    tree[key_list[i]] = value_list[i]
tree.print_tree()
```

```
      ---9
            ---8
---7
            ---5
      ---4
                  ---3
            ---1
```

In [104…
```python
tree[9] = 'h'
tree[6] = 'i'
tree.print_tree()
```

```
      ---9
            ---8
---7
                  ---6
            ---5
      ---4
                  ---3
            ---1
```

In [105…
```python
tree.get_count(9)    # wprowadzone 2 razy - stąd wartość 1
```

Out[105… 1

In [106…
```python
4 in tree
```

Out[106… True

In [107…
```python
tree.delete(4)
4 in tree
```

Out[107… False

In [108…
```python
tree.print_tree()
```

```
      ---9
            ---8
---7
            ---6
      ---5
                  ---3
            ---1
```

In [109…
```python
tree.delete(4)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-109-cdd8da9fc365> in <module>
----> 1 tree.delete(4)

<ipython-input-102-a3b85f7ea3ff> in delete(self, key)
     87                     self.size -= 1
     88             else:
---> 89                 raise KeyError('Key not in the tree')
     90         elif self.size == 1 and self.root.key == key:
     91             if self.root.is_overload():

KeyError: 'Key not in the tree'
```

In [110…
```python
tree[6]
```

Out[110… 'i'

Sprawdzimy, czy relacje pomiędzy kluczami są poprawne.

In [111…
```python
tree = BinarySearchTree()
key_list = [12, 7, 6, 9, 1, 5, 0, 7, 15, 8, 17, 16]
for i in range(len(key_list)):
    tree[key_list[i]] = i    # wartosc nie ma tu dla nas znaczenia
tree.print_tree()
```

```
            ---17
                  ---16
      ---15
---12
            ---9
                  ---8
      ---7
            ---6
                        ---5
                  ---1
                        ---0
```

In [112…
```python
tree.root.key  #12
```

Out[112… 12

In [113…
```python
tree.root.lc.rc.lc.key   #8
```

Out[113… 8

In [114…
```python
tree.root.rc.parent.key   #12
```

Out[114… 12

In [115…
```python
tree.root.find_min().key  #0
```

Out[115… 0

In [118…
```python
a = BinarySearchTree()
a.print_tree()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-118-b93ed115cdb4> in <module>
      1 a = BinarySearchTree()
----> 2 a.print_tree()

<ipython-input-117-e3f89d7434a4> in print_tree(self)
    154                 print(line)
    155         else:
--> 156             raise NameError("There's no tree to show here yet")

NameError: There's no tree to show here yet
```

Zad. 2 Zaimplementuj kopiec binarny. Korzystając z tego kopca napisz funkcję sortującą listę elementów w czasie $O(n \log n)$. Przeprowadź analizę eksperymentalną czasu wykonania algorytmu.

In [51]:
```python
class BinHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def size(self):
        return self.current_size

    def is_empty(self):
        return self.current_size == 0

    def perc_up(self, i):
        while i // 2 > 0:
            if self.heap_list[i] < self.heap_list[i // 2]:
                tmp = self.heap_list[i // 2]
                self.heap_list[i // 2] = self.heap_list[i]
                self.heap_list[i] = tmp
            i //= 2

    def find_min(self):
        return self.heap_list[1]

    def min_child(self, i):
        if i * 2 + 1 > self.current_size:
            return i * 2
        else:
            if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def perc_down(self, i):
        while (i * 2) <= self.current_size:
            mc = self.min_child(i)
            if self.heap_list[i] > self.heap_list[mc]:
                tmp = self.heap_list[i]
                self.heap_list[i] = self.heap_list[mc]
                self.heap_list[mc] = tmp
            i = mc

    def del_min(self):
        retval = self.heap_list[1]
        self.heap_list[1] = self.heap_list[self.current_size]
        self.current_size -= 1
        self.heap_list.pop()
        self.perc_down(1)
        return retval

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        self.perc_up(self.current_size)

    def buildHeap(self, build_list):
        i = len(build_list) // 2
        self.current_size = len(build_list)
        self.heap_list = [0] + build_list[:]
        while i > 0:
            self.perc_down(i)
            i -= 1

    def __str__(self):
        txt = "{}".format(self.heap_list[1:])
        return txt
```

In [54]:
```python
heap_test = BinHeap()
heap_test.buildHeap([2, 5, 3, 1, 1, 8, 4, 6])
print(heap_test)
```

```
[1, 1, 3, 2, 5, 8, 4, 6]
```

In [56]:
```python
def sort_heap(data_list):
    heap = BinHeap()
    heap.buildHeap(data_list)
    return [heap.del_min() for _ in range(len(data_list))]
```

In [58]:
```python
sort_heap([2, 5, 3, 1, 1, 8, 4, 6])
```

Out[58]: [1, 1, 2, 3, 4, 5, 6, 8]

Tak jak w przypadku listy 5, w pliku .py czasy zapisuje do pliku, by nie program ich nie liczył za każdym razem. Tutaj skorzystam z uzyskanych wcześniej danych.

In [14]:
```python
def random_sort_time(n):
    random_data = [random.randint(-100, 100) for _ in range(n)]
    start = time.time()
    sort_heap(random_data)
    end = time.time()
    time_data = end - start
    return time_data

#def time_check_and_save(n_list, file_name):
#    times = []
#    for num in n_list:
#        times.append(random_sort_time(num))
#    file = open(file_name, 'w')
#    file.write(str(times))
#    file.close()
#
#
#def get_times(list_of_data, file_name):
#    if not os.path.exists(file_name):
#        time_check_and_save(list_of_data, file_name)
#
#    file = open(file_name, 'r')
#    times = eval(file.read())
#    return times
```

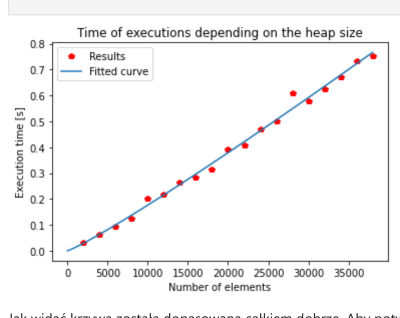
Wiemy, że funkcja powinna sortować elementy w czasie $O(n \log n)$. Sortujemy poprzez usuwanie najmniejszego elementu z kopca i wstawienie go do nowej listy, co kosztuje nas za każdym razem $\log n$. Dlatego też, koszt posortowania n-elementowego kopca to $n \log n$. Sprawdzimy to, dopasowując krzywą $an \log n + b$ do uzyskanych wyników.

In [15]:
```python
def func(n, a, b):
    return a * n * np.log(n) + b
```

In [16]:
```python
def plot_hypothesis(x, y, func, popt):
    x2 = np.arange(1, x[-1])
    plt.plot(x, y, 'rp', label="Results")
    plt.plot(x2, func(x2, *popt), label="Fitted curve")
    plt.xlabel("Number of elements")
    plt.ylabel("Execution time [s]")
    plt.legend(loc='upper left')
    plt.title("Time of executions depending on the heap size")
    plt.show()
```

In [17]:
```python
trial_data = [2000 * n for n in range(1, 20)]
trial_times = [0.03125, 0.0625, 0.09375, 0.125, 0.203125, 0.21875, 0.265625, 0.28125, 0.3125, 0.390625, 0.40625
```

In [18]:
```python
popt, pcov = curve_fit(func, trial_data, trial_times)
plot_hypothesis(trial_data, trial_times, func, popt)
```



Jak widać krzywa została dopasowana całkiem dobrze. Aby potwierdzić czas z jakim sortujemy, spróbujemy przyrównać wartość dopasowywującej krzywej do czasu dla liczby n większej, niż obejmuje wykres.

In [29]:
```python
n = 50000
ex_time = random_sort_time(50000)
```

In [30]:
```python
fitted_func = func(n, *popt)
```

In [31]:
```python
print("", ex_time, "\n", fitted_func)
```

```
 1.082148790359497
 1.0345255545867635
```

In [35]:
```python
n = 60000
ex_time = random_sort_time(60000)
```

In [36]:
```python
fitted_func = func(n, *popt)
```

In [37]:
```python
print("", ex_time, "\n", fitted_func)
```

```
 1.2706091403961182
 1.262193977512508
```

Z uzyskanych wyników możemy wywnioskować, że udało się zaimplementować funckje sotrującą w czasie $O(n \log n)$.

Zad. 3 Zaimplementuj kopiec binarny o ograniczonej wielkości n. Innymi słowy, stwórz strukturę przechowującą n najważniejszych (największych) wartości.

In [122…
```python
class LimitedBinHeap:

    def __init__(self, limit):
        self.heap_list = [0]
        self.current_size = 0
        self.limit = limit

    def size(self):
        return self.current_size

    def is_empty(self):
        return self.current_size == 0

    def perc_up(self, i):
        while i // 2 > 0:
            if self.heap_list[i] < self.heap_list[i // 2]:
                tmp = self.heap_list[i // 2]
                self.heap_list[i // 2] = self.heap_list[i]
                self.heap_list[i] = tmp
            i //= 2

    def find_min(self):
        return self.heap_list[1]

    def min_child(self, i):
        if i * 2 + 1 > self.current_size:
            return i * 2
        else:
            if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def perc_down(self, i):
        while (i * 2) <= self.current_size:
            mc = self.min_child(i)
            if self.heap_list[i] > self.heap_list[mc]:
                tmp = self.heap_list[i]
                self.heap_list[i] = self.heap_list[mc]
                self.heap_list[mc] = tmp
            i = mc

    def del_min(self):
        retval = self.heap_list[1]
        self.heap_list[1] = self.heap_list[-1]
        self.current_size -= 1
        self.heap_list.pop()
        self.perc_down(1)
        return retval

    def insert(self, k):

        if self.current_size < self.limit:
            self.heap_list.append(k)
            self.current_size += 1
            self.perc_up(self.current_size)
        else:
            if self.find_min() > k:
                raise ValueError("Given value is too small")
            else:
                self.del_min()
                self.insert(k)

    def build_heap(self, build_list):

        size = len(build_list)
        if size <= self.limit:
            i = size // 2
            self.current_size = size
            self.heap_list = [0] + build_list[:]
            while i > 0:
                self.perc_down(i)
                i -= 1
        else:
            limit_list = build_list[:self.limit]
            self.build_heap(limit_list)

            for k in build_list[self.limit:]:
                self.insert(k)

    def __str__(self):
        return str(self.heap_list[1:])
```

In [123…
```python
heap = LimitedBinHeap(7)
heap.build_heap([4, 7, 2, 8, 0, 6, 3, 12, 7, 5])
print(heap)
```

```
[4, 7, 5, 8, 7, 12, 6]
```

In [124…
```python
heap.current_size
```

Out[124… 7

In [125…
```python
heap.insert(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-125-dc47ddbab507> in <module>
----> 1 heap.insert(3)

<ipython-input-122-1174a076843a> in insert(self, k)
     57         else:
     58             if self.find_min() > k:
---> 59                 raise ValueError("Given value is too small")
     60             else:
     61                 self.del_min()

ValueError: Given value is too small
```

In [126…
```python
heap.insert(9)
heap.insert(13)
print(heap)
```

```
[6, 7, 9, 8, 7, 12, 13]
```

Podczas dodawania elementów do ograniconego kopca, jego struktura również zostaje zachowana.

Zad. 4 Napisz funkcję, która na wejściu przyjmuje drzewo wyprowadzenia jakiegoś wyrażenia matematycznego, a na wyjściu zwraca pochodną tego wyrażenia względem podanej zmiennej.

Zaczne od zaimplementowania struktur takich jak stak i drzewo binarne, które pomogą w napisaniu właściwego kodu.

In [38]:
```python
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)


class BinaryTree:
    def __init__(self, root_obj):
        self.key = root_obj
        self.left_child = None
        self.right_child = None

    def insert_left(self, new_node):
        if self.left_child is None:
            self.left_child = BinaryTree(new_node)
        else:
            tree = BinaryTree(new_node)
            tree.left_child = self.left_child
            self.left_child = tree

    def insert_right(self, new_node):
        if self.right_child is None:
            self.right_child = BinaryTree(new_node)
        else:
            tree = BinaryTree(new_node)
            tree.right_child = self.right_child
            self.right_child = tree

    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_value(self, obj):
        self.key = obj

    def get_root_value(self):
        return self.key

    def __str__(self):
        return self.key
```

In [39]:
```python
def parse_string(p_str, unknown):
    p_str = p_str.replace(" ", "")
    plist = []
    while len(p_str) > 0:
        if p_str[:3] in ['sin', 'cos', 'exp']:
            plist.append(p_str[:3])
            p_str = p_str[3:]
        elif p_str[:2] == 'ln':
            plist.append(p_str[:2])
            p_str = p_str[2:]
        elif p_str[0] in ['(', ')', '*', '+', '-', '/', '^', unknown]:
            plist.append(p_str[0])
            p_str = p_str[1:]
        elif p_str[0].isdigit():
            plist.append(p_str[0])
            p_str = p_str[1:]
    return plist
```

In [41]:
```python
parse_string("cos(3*x) -x^3", "x")
```

Out[41]: ['cos', '(', '3', '*', 'x', ')', '-', 'x', '^', '3']

LINK GITHUB

https://github.com/AlutkaMalutka/Programowanie_python/tree/main/semestr_3/Lista_6-3s-