Michał Szczurek - lab 2

1. Przygotowanie danych

Do ostatniego tekstu dodano na koniec $ jako marker. W pozostałych rolę markera spełnia znak d.

In [5]:
```python
f = open("1997_714_head.txt", "r", encoding="UTF-8")
data = ["bbbd","aabbabd","ababcd","abcbccd", f.read()]

data[4] = data[4] + "$"
```

In [6]:
```python
print (data)
```

```
['bbbd', 'aabbabd', 'ababcd', 'abcbccd', '\n\n\n\nDz.U. z 1998 r. Nr 144, poz. 930\n
\n                                      \n                                      \n
\n                                  USTAWA\n                        z dnia 20 listopada 1998 r.\n
\n        o zryczałtowanym podatku dochodowym od niektórych przychodów\n                     osiąganych przez oso
by fizyczne\n                                    \n                                 Rozdział 1\n
Przepisy ogólne\n                                    \n                                 Art. 1.\nUstawa reguluj
e opodatkowanie zryczałtowanym podatkiem dochodowym niektórych\nprzychodów (dochodów) osiąganych przez osoby fizyczne
prowadzące pozarolniczą\ndziałalność gospodarczą oraz przez osoby duchowne.\n
\n                                  Art. 2.\n1. Osoby fizyczne osiągające przychody z pozarolniczej działalności\n
gospodarczej opłacają zryczałtowany podatek dochodowy w formie:\n   1) ryczałtu od przychodów ewidencjonowanych,\n
2) karty podatkowej.\n2. Osoby duchowne, prawnie uznanych wyznań, opłacają zryczałtowany podatek\n  dochodowy od przy
chodów osób duchownych.\n3. Wpływy z podatku dochodowego opłacanego w formie ryczałtu od przychodów\n  ewidencjonowan
ych oraz zryczałtowanego podatku dochodowego od przychodów\n  osób duchownych stanowią dochód budżetu państwa.\n4. Wp
ływy z karty podatkowej stanowią dochody gmin.\n                                   \n
Art. 3.\nPrzychodów (dochodów) opodatkowanych w formach zryczałtowanych nie łączy się z\nprzychodami (dochodami) z in
nych źródeł podlegającymi opodatkowaniu na\npodstawie ustawy z dnia 26 lipca 1991 r. o podatku dochodowym od osób\nfi
zycznych (Dz. U. z 1993 r. Nr 90, poz. 416 i Nr 134, poz. 646, z 1994 r. Nr\n43, poz. 163, Nr 90, poz. 419, Nr 113, p
oz. 547, Nr 123, poz. 602 i Nr 126,\npoz. 626, z 1995 r. Nr 5, poz. 25 i Nr 133, poz. 654, z 1996 r. Nr 25, poz.\n11
3, Nr 87, poz. 395, Nr 137, poz. 638, Nr 147, poz. 686 i Nr 156, poz. 776, z\n1997 r. Nr 28, poz. 153, Nr 30, poz. 16
4, Nr 71, poz. 449, Nr 85, poz. 538, Nr\n96, poz. 592, Nr 121, poz. 770, Nr 123, poz. 776, Nr 137, poz. 926, Nr 13
9,\npoz. 932-934 i Nr 141, poz. 943 i 945 oraz z 1998 r. Nr 66, poz. 430, Nr 74,\npoz. 471, Nr 108, poz. 685 i Nr 11
7, poz. 756), zwanej dalej "ustawą o podatku\ndochodowym".\n$']
```

2. Struktura Trie

In [7]:
```python
class Node:
    def __init__(self, char, depth):
        self.children = {}
        self.char = char
        self.depth = depth # used for giving node unique label, in order to draw tree plot (which did not work in th
e end ;))
        # self.end = False if we want to have only suffixes in trie

    def add_child(self, char):
        d[char] = Node(char)

    def has_child(self, char):
        return char in self.children
```

In [8]:
```python
class Trie:

    def __init__(self,text):
        self.root = Node("$",0)
        for i in range(len(text)-1,-1,-1):
            self.add(text[i:])
            #print(text[i:])

    def add(self, text):
        i = 0
        node = self.root
        while i < len(text) and node.has_child(text[i]):
            node = node.children[text[i]]
            i += 1
        while i < len(text):
            node.children[text[i]] = Node(text[i],node.depth + 1)
            node = node.children[text[i]]
            i += 1
        node.end = True

    def contains(self,text):
        i = 0
        node = self.root
        while i < len(text) and node.has_child(text[i]):
            prev = node
            node = node.children[text[i]]
            i += 1
        return i == len(text) # and node.end == True
```

3. Drzewo suffiksowe

In [9]:
```python
class Suffix_node:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.children = {}

    def __repr__(self):
        return f"({self.start}, {self.end})"
    def __str__(self):
        return f"({self.start}, {self.end})"

    def print(self):
        for key, child in self.children.items():
            print(child)
        for key, child in self.children.items():
            print("======")
            print("Children of: " + str(child))
            child.print()
```

In [10]:
```python
class Suffix_tree:
    def __init__(self, text):
        self.root = Suffix_node(-1, -2) # special values for root It is important that start > end for "contains" to
work
        self.text = text
        for i in range(len(text)-1,-1,-1):
            self.add(i, self.root)
            # self.root.print() debbug
            # print("========================")

    def add(self, curr_id, curr_node):
        next_node = curr_node.children.get(self.text[curr_id])

        if next_node is None: # create new node representing text[i:]
            new_node = Suffix_node(curr_id, len(self.text)-1)
            curr_node.children[self.text[curr_id]] = new_node
            return

        i = next_node.start
        while i <= next_node.end and curr_id < len(self.text) and self.text[curr_id] == self.text[i]:
            i += 1
            curr_id += 1

        if curr_id == len(self.text): # whole new suffix is already represented
            return # new suffix already in tree, nothing to do

        elif i == next_node.end + 1: # index out of nodes range
            self.add(curr_id , next_node) # add remaining part of suffix to child
            return

        else: # text[curr_id] != text[i], we need to split branch
            old_branch = Suffix_node(i, next_node.end)
            old_branch.children = next_node.children
            new_branch = Suffix_node(curr_id, len(self.text)-1)
            next_node.end = i-1
            next_node.children = {}
            next_node.children[self.text[curr_id]] = new_branch
            next_node.children[self.text[i]] = old_branch
            return

    def contains(self, pattern, curr_node = None):
        if curr_node == None:
            curr_node = self.root

        i = 0
        while i + curr_node.start <= curr_node.end and i < len(pattern):
            if self.text[curr_node.start + i] != pattern[i]:
                return False
            else:
                i += 1

        if i == len(pattern):
            return True

        else: # search in child
            next_node = curr_node.children.get(pattern[i])
            if next_node == None:
                return False
            return self.contains(pattern[i:], next_node)
```

4. Testowanie poprawności algorytmów

4.0 Uniwersalna funkcja testująca

In [11]:
```python
def test(trie, suffix, expected_true, expected_false):
    for pattern in expected_true:
        if trie.contains(pattern) is False:
            print("Error FALSE in trie for " + pattern)
            return False
        if suffix.contains(pattern) is False:
            print("Error FALSE in suffix tree for " + pattern)
            return False

    for pattern in expected_false:
        if trie.contains(pattern) is True:
            print("Error TRUE in trie for " + pattern)
            return False
        if suffix.contains(pattern) is True:
            print("Error TRUE in suffix tree for " + pattern)
            return False
    print("OK")
    return True
```

4.1 Testy dla 1. tekstu:

In [12]:
```python
a_trie = Trie(data[0])
a_suffix = Suffix_tree(data[0])
expected_true = ["bb", "bd", "bbb","bbbd","b"]
expected_false = ["a","db","bbbbbbb"]
test(a_trie, a_suffix , expected_true, expected_false)
```

```
OK
```

Out[12]: True

4.2 Testy dla 2. tekstu:

In [13]:
```python
b_trie = Trie(data[1])
b_suffix = Suffix_tree(data[1])
expected_true = [data[1][i:j] for i in range(len(data[1])) for j in range(len(data[1]))]
expected_false = ["aaa","db","bbbbbbb", "aba"]
test(b_trie, b_suffix , expected_true, expected_false)
```

```
OK
```

Out[13]: True

4.3 Testy dla 3. tekstu:

In [14]:
```python
c_trie = Trie(data[2])
c_suffix = Suffix_tree(data[2])
expected_true = [data[2][i:j] for i in range(len(data[2])) for j in range(len(data[2]))]
expected_false = ["aaa","db","bbb", "abaa", "dc", "baba"]
test(c_trie, c_suffix , expected_true, expected_false)
```

```
OK
```

Out[14]: True

4.4 Testy dla 4. tekstu:

In [15]:
```python
d_trie = Trie(data[3])
d_suffix = Suffix_tree(data[3])
expected_true = [data[3][i:j] for i in range(len(data[3])) for j in range(len(data[3]))]
expected_false = ["aaa","db","bbb", "aba", "dc", "baba", "cccc"]
test(d_trie, d_suffix , expected_true, expected_false)
```

```
OK
```

Out[15]: True

4.5 Testy dla 5. tekstu:

In [16]:
```python
e_trie = Trie(data[4])
e_suffix = Suffix_tree(data[4])
expected_true = ["anych oraz zryczałtowanego podatku dochodowego od przychodów",
                 "Nr 133, poz. 654, z 1996 r. Nr 25,",
                 "20 listopada 1998 r.",
                 "a"]
expected_false = ["aaa","db","bbb", "aba", "dc", "baba", "cccc"]
test(e_trie, e_suffix , expected_true, expected_false)
```

```
OK
```

Out[16]: True

4. Badanie czasów budowy struktur

In [17]:
```python
print("TESTY DLA DRZEWA TRIE\n")
for i in range(len(data)):
    print(f"Test prędkości dla {i+1}. tekstu")
    %timeit Trie(data[i])
```

```
TESTY DLA DRZEWA TRIE

Test prędkości dla 1. tekstu
8.63 µs ± 500 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
Test prędkości dla 2. tekstu
21.8 µs ± 1.6 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
Test prędkości dla 3. tekstu
19.3 µs ± 1.83 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
Test prędkości dla 4. tekstu
24.9 µs ± 1.72 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
Test prędkości dla 5. tekstu
3.07 s ± 139 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [18]:
```python
print("TESTY DLA DRZEWA SUFIKSOWEGO\n")
for i in range(len(data)):
    print(f"Test prędkości dla {i+1}. tekstu")
    %timeit Suffix_tree(data[i])
```

```
TESTY DLA DRZEWA SUFIKSOWEGO

Test prędkości dla 1. tekstu
7.38 µs ± 272 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
Test prędkości dla 2. tekstu
13.1 µs ± 1.3 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
Test prędkości dla 3. tekstu
9.6 µs ± 384 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
Test prędkości dla 4. tekstu
11.1 µs ± 530 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
Test prędkości dla 5. tekstu
25.7 ms ± 936 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Z powyższych testów wynika, że czas potrzebny na stworzenie drzewa sufiksowego jest istotnie mniejszy od czasu budowy drzewa Trie pomimo takiej samej złozoności teoretycznej. Ponadto warto zwrócić, uwagę, na fakt, że drzewa sufiksowe zajmują istotnie mniej pamięci.