# Lab 02 - Trie Tree and Sufix Tree

In [1]:
```
from timeit import default_timer as timer
```

## Sample data

Sample input data:

- "bbb$" (również w wersji, gdzie znak 'b' został powtórzony więcej razy)
- "aabbabd"
- "ababcd"
- "abcbccd"
- 1997_714_head.txt file

In [2]:
```
data_full = ['bbb$', 'bbbbbbbbbb$', 'aabbabd', 'ababcd', 'abcbccd',
             open('1997_714_head.txt', mode='r', encoding='utf-8').read()]
```

Checking if every dataset has unique character (marker) at it's end. If not - add it.

In [3]:
```
potencial_markers = ['$', '#', '@', '*', '^', '%', '(', ')', '-', '=', '_', '+']

for data in data_full:
    last_char = data[-1]
    n = len(data)

    for i in range(0, n-1):
        if last_char == data[i]:
            last_char = ''
            break

    if last_char == '':
        for marker in potencial_markers:
            if marker not in data:
                data += marker
                break
        else:
            raise Exception("No suitable marker found!")
```

## Implementation of trie tree structure

In [4]:
```
class trie_node:
    def __init__(self, ch = None):
        self.char = ch
        self.children = {}

    def create_path(self, text, index):
        if index == len(text):
            return
        elif text[index] not in self.children.keys():
            self.children[text[index]] = trie_node(text[index])
        self.children[text[index]].create_path(text, index + 1)

    def search(self, text, index):
        if len(text) == index:
            return True

        if text[index] in self.children.keys():
            return self.children[text[index]].search(text, index+1)
        return False

class trie_tree:
    def __init__(self, text):
        self.root = trie_node()
        for i in range(len(text)):
            self.root.create_path(text, i)

    def find_occurence(self, s):
        return self.root.search(s, 0)
```

In [5]:
```
cnt = 0

class sufix_edge:
    def __init__(self, p, q):
        self.start_ = p
        self.end_   = q

class sufix_node:
    def __init__(self):
        self.children = {}

    def build_path(self, text, p, q):
        char = text[p]
        if char not in self.children.keys():
            # jeśli brak wspólnego sufixu to tworzymy nową krawędź
            self.children[char] = (sufix_edge(p, q), sufix_node())
        else:
            edge, node = self.children[char]
            edge_len = edge.end_ - edge.start_ + 1
            text_len = q - p + 1

            eq_len = 1
            # jak długi jest wspólny sufix?
            for i in range(1, min(edge_len, text_len)):
                index_edge = edge.start_ + i
                index_text = p + i
                if text[index_edge] != text[index_text]:
                    break
                eq_len += 1

            # jeśli wspólny prefix pokrywa całą krawędź (a konkretnie tekst jaki reprezentuje ta krawędź)
            if eq_len == edge_len:
                # jeśli wspólny prefix nie pokrywa całego tekstu, który chcemy dodać
                # do danego węzła to kontynuujemy od kolejnego węzła
                if eq_len != text_len:
                    node.build_path(text, p + edge_len, q)
                return

            # jeśli wspólny prefix nie pokrywa całej krawędzi musimy dokonać jej rozdzielenia
            new_node = sufix_node()
            self.children[char] = (sufix_edge(edge.start_, edge.start_ + eq_len-1), new_node)
            new_node.children[text[edge.start_ + eq_len]] = (sufix_edge(edge.start_ + eq_len, edge.end_), node)

            # jeśli nie pokrywa całej krawędzi I całego tekstu, który chcemy dodać to zaczynamy dodawanie
            # kolejnego fragmentu tekstu od węzła rozdzielającego, który przed chwilą wstawiliśmy
            if eq_len != text_len:
                new_node.build_path(text, p+eq_len, q)

    def search(self, pattern, text):
        if pattern == '':
            return True
        if pattern[0] not in self.children.keys():
            return False

        edge, next_node = self.children[pattern[0]]
        m = edge.end_ - edge.start_ + 1
        n = len(pattern)
        if n <= m:
            return text[edge.start_ : edge.start_ + n] == pattern

        return text[edge.start_ : edge.end_+1] == pattern[:m] and next_node.search(pattern[m:], text)


class sufix_tree:
    def __init__(self, text):
        self.text = text
        self.root = sufix_node()
        self._len = len(text)
        global cnt

        for i in range(self._len-1):
            cnt += 1
            self.root.build_path(self.text, i, self._len-1)

    def search(self, pattern):
        _len = len(pattern)

        return self.root.search(pattern, self.text)
```

## Time comparison

In [7]:
```
for i in range(len(data_full)):
    data = data_full[i]
    len_data = len(data)
    print("Test #" + str(i+1), end = ' : ')
    if len_data >= 20:
        print(data[0:20].replace("\n", "\\n") + "... (too long)")
    else:
        print(data.replace("\n", "\\n"))

    trie_start = timer()
    tree = trie_tree(data)
    trie_end = timer()


    print("Creation of trie tree : " + str(f'{trie_end - trie_start : .7f}') + " seconds")

    sufix_start = timer()
    tree = sufix_tree(data)
    sufix_end = timer()

    print("Creation of sufix tree: " + str(f'{sufix_end - sufix_start : .7f}') + " seconds")

    print('')
```

```
Test #1 : bbb$
Creation of trie tree :  0.0011590 seconds
Creation of sufix tree:  0.0000194 seconds

Test #2 : bbbbbbbbbb$
Creation of trie tree :  0.0000307 seconds
Creation of sufix tree:  0.0000337 seconds

Test #3 : aabbabd
Creation of trie tree :  0.0000192 seconds
Creation of sufix tree:  0.0000154 seconds

Test #4 : ababcd
Creation of trie tree :  0.0000223 seconds
Creation of sufix tree:  0.0000106 seconds

Test #5 : abcbccd
Creation of trie tree :  0.0000283 seconds
Creation of sufix tree:  0.0000200 seconds

Test #6 : \n\n\n\nDz.U. z 1998 r. ... (too long)
Creation of trie tree :  8.0458083 seconds
Creation of sufix tree:  0.4240891 seconds
```

Różnice w czasie generowania struktur w przypadku tekstów krótkich nie są różnicami znacznymi. Różnice stają się wyraźne, gdy tworzymy struktury w oparciu o długie teksty. Spowodowane jest to prawdopodobnie tym, że drzewo sufixów przy nieznalezieniu jakiegokolwiek wspólnego początku tworzy po prostu pojedyńczą krawędź, podczas gdy drzewo trie przechodzi zawsze znak po znaku dodając kolejne węzły. Drzewo sufixów nie radzi sobie natomiast najlepiej z tekstami, gdzie dużo sufixów ma wspólne prefixy, co może być spowodowane przez dodawanie duzych ilości węzłów rozszczepiających istniejące już krawędzie, lub błędy w implementacji popełnione przez niesfornego programistę.

In [ ]:
```
```