Міністерство освіти України

Київський Національний Університет імені Тараса Шевченка

Механіко-математичний факультет

Кафедра алгебри і комп’ютерної математики

КУРСОВА РОБОТА на тему:

# «Нумерація дерев та конвертація у бінарні та троїчні дерева пошуку»

Виконав студент 1 курсу магістратури,
групи Комп’ютерна математика
Тищенко Тимофій Андрійович

Науковий керівник: Бородін Віктор Анатолійович

Київ – 2023

**Зміст**


1. Вступ
2. Теоретичні основи

    2.1. Нумерація бінарних дерев

    2.2. Числа Каталана та їх зв'язок з кількістю бінарних дерев

    2.3. Алгоритм нумерації бінарних дерев

    2.4. Бінарні дерева пошуку (BST)

    2.5. Троїчні дерева пошуку (TST)

3. Реалізація алгоритмів та структур даних

    3.1. Опис структури проекту

    3.2. Клас TreeConverter

    3.3. Клас TreePrinter

    3.4. Допоміжні функції

4. Аналіз та результати

    4.1. Візуалізація дерев за допомогою бібліотеки NetworkX

    4.2. Візуалізація дерев за допомогою бібліотеки NetworkX

5. Висновки
6. Список використаних джерел

## 1. Вступ

Актуальність теми дослідження алгоритмів нумерації дерев обумовлена їх важливістю у галузі інформаційних технологій. Дерева є фундаментальними структурами даних, які використовуються для представлення ієрархічних відношень та організації даних. Нумерація дерев дозволяє однозначно ідентифікувати кожне дерево в межах певного класу та ефективно генерувати дерева за заданим номером.

Метою даної роботи є дослідження та реалізація алгоритмів нумерації дерев, зокрема, бінарних дерев. Також, розглядаються бінарні дерева пошуку (BST) та троїчні дерева пошуку (TST) як доповнення до основної теми. Завданнями роботи є:

1. Вивчення теоретичних основ нумерації бінарних дерев.
2. Реалізація алгоритмів нумерації бінарних дерев та генерації дерева за номером.
3. Дослідження BST та TST як прикладів застосування бінарних дерев.
4. Аналіз ефективності реалізованих алгоритмів та візуалізація дерев.

Робота складається з трьох основних розділів. У першому розділі розглядаються теоретичні основи нумерації бінарних дерев, а також BST та TST. Другий розділ присвячений реалізації алгоритмів нумерації та генерації дерев, а також перетворенню дерев у BST та TST. У третьому розділі проводиться аналіз ефективності реалізованих алгоритмів та демонструється візуалізація дерев.

## 2. Теоретичні основи

### *2.1. Нумерація бінарних дерев*

Нумерація бінарних дерев - це спосіб присвоєння унікального номера кожному бінарному дереву з певною кількістю вершин. Це дозволяє однозначно ідентифікувати кожне дерево та встановити взаємно однозначну відповідність між деревами та їх номерами.

Ідея нумерації бінарних дерев полягає в тому, що кожне дерево отримує номер на основі кількості вершин та номерів його лівого та правого піддерев. Нумерація здійснюється за наступними правилами:

1. Пусте дерево (дерево з 0 вершин) має номер 0.
2. Дерево з однією вершиною має номер 1.
3. Для дерева з n вершинами (n > 1), його номер визначається на основі номерів лівого та правого піддерев.

Розглянемо фрагмент коду з класу `TreePrinter`, який реалізує нумерацію бінарних дерев:

```
def print_tree(self, order_num):
    if order_num == 0:
        return ""
    node_num = 1
    while self.tree_num_sum[node_num] <= order_num:
        node_num += 1
    return self.print_tree_helper(order_num - self.tree_num_sum[node_num - 1],
node_num)

def print_tree_helper(self, order_num, node_num):
    if node_num == 0:
        return ""
    order_before = 0
    for children_node_nums in self.children_combinations[node_num]:
        tree_num_product = self.get_tree_num_product(children_node_nums)
        if order_before + tree_num_product > order_num:
            break
        order_before += tree_num_product
    children_trees = [self.print_tree_helper(children_order_nums[child],
children_node_nums[child]) for child in range(self.max_children)]
    output = f'{order_num + self.tree_num_sum[node_num - 1]}('
    for ind, child_tree in enumerate(children_trees):
        if ind > 0:
            output += ','
        if child_tree:
            children_trees[ind] = f'({child_tree})'
            output += f'{child_tree}'
        else:
            output += '0'
    output += ')'
    return output
```

Метод `print_tree` приймає номер дерева `order_num` та повертає рядкове представлення дерева з цим номером. Спочатку визначається кількість вершин `node_num` у дереві на основі `order_num` та масиву `tree_num_sum`, який містить сумарну кількість дерев до певної кількості вершин.

Метод `print_tree_helper` рекурсивно генерує рядкове представлення дерева. Він обчислює номери лівого та правого піддерев на основі `order_num` та комбінацій кількості вершин у піддеревах. Далі рекурсивно викликається для лівого та правого піддерев і формується результуючий рядок з номером дерева та рядковими представленнями піддерев.

### *2.2. Числа Каталана та їх зв'язок з кількістю бінарних дерев*

Числа Каталана - це послідовність натуральних чисел, яка має широке застосування в комбінаториці та теорії графів. n-е число Каталана `C(n)` визначає кількість структурно унікальних бінарних дерев з `n` вершинами.

У класі `TreePrinter` числа Каталана обчислюються за допомогою методу `_initialize`:

python

Copy code

```python
def _initialize(self):
    self.tree_num[0] = self.tree_num_sum[0] = 1
    for node_num in range(1, self.max_nodes + 1):
        self.tree_num[node_num] = 0
        children_combs = get_children_combinations(node_num,
self.max_children)
        self.children_combinations.append(children_combs)
        for children_node_nums in children_combs:
            tree_num_product = self.get_tree_num_product(children_node_nums)
            self.tree_num[node_num] += tree_num_product
        self.tree_num_sum[node_num] = self.tree_num_sum[node_num - 1] +
self.tree_num[node_num]
```

Метод `_initialize` обчислює кількість дерев з певною кількістю вершин (`tree_num`) та їх сумарну кількість до певної кількості вершин (`tree_num_sum`). Для кожної кількості вершин `node_num` генеруються всі можливі комбінації кількості вершин у піддеревах за допомогою функції `get_children_combinations`. Далі для кожної комбінації обчислюється добуток кількості дерев у піддеревах (`tree_num_product`), який додається до загальної кількості дерев з `node_num` вершинами.

*2.3. Алгоритм нумерації бінарних дерев*

Алгоритм нумерації бінарних дерев базується на рекурсивному обчисленні номерів дерев з урахуванням номерів їх лівого та правого піддерев.

Розглянемо метод `get_tree_num_product` з класу `TreePrinter`, який обчислює добуток кількостей дерев для заданої конфігурації дочірніх вершин:

```
def get_tree_num_product(self, children_node_nums):
    tree_num_product = 1
    for child_node_num in children_node_nums:
        tree_num_product *= self.tree_num[child_node_num]
    return tree_num_product
```

Цей метод приймає список `children_node_nums`, який містить кількості вершин у кожному піддереві, та обчислює добуток кількостей дерев (`tree_num_product`) для цієї конфігурації дочірніх вершин.

Метод `print_tree_helper`, який ми розглянули раніше, використовує `get_tree_num_product` для обчислення номерів лівого та правого піддерев на основі `order_num` та комбінацій кількості вершин у піддеревах.

*2.4. Бінарні дерева пошуку (BST)*

Бінарне дерево пошуку (BST) - це впорядковане бінарне дерево, в nl/k,nkljp;якому для кожної вершини виконується властивість: ключі всіх вершин у лівому піддереві менші за ключ даної вершини, а ключі всіх вершин у правому піддереві більші за ключ даної вершини.

Розглянемо фрагмент коду з класу `TreeConverter`, який відповідає за конвертацію дерева у BST:

```
def to_bst(self):
    values = self._extract_values()
    values = sorted(set(values) - {0})  # Remove duplicates and 0, then sort
    self._build_bst(values)
    return self

def _build_bst(self, values):
    if not values:
        return
    mid = len(values) // 2
    self.value = values[mid]
```

```
    left_values = values[:mid]
    right_values = values[mid+1:]

    if left_values:
        self.children[0] = TreeConverter(left_values[len(left_values) // 2],
max_children=2)
        self.children[0]._build_bst(left_values)
    else:
        self.children[0] = None

    if right_values:
        self.children[1] = TreeConverter(right_values[len(right_values) // 2],
max_children=2)
        self.children[1]._build_bst(right_values)
    else:
        self.children[1] = None
```

Метод `to_bst` спочатку викликає метод `_extract_values`, який отримує всі значення з дерева. Потім значення сортуються, видаляються дублікати та значення 0. Далі викликається метод `_build_bst`, який рекурсивно будує BST на основі відсортованих значень.

Метод `_build_bst` вибирає середній елемент зі списку значень `values` як корінь дерева. Потім рекурсивно будує ліве та праве піддерева на основі лівої та правої частин списку значень.

*2.5. Троїчні дерева пошуку (TST)*

Троїчне дерево пошуку (TST) - це впорядковане дерево, в якому кожна вершина має три дочірні вершини: ліву, середню та праву. TST використовується для зберігання та пошуку рядків.

Розглянемо фрагмент коду з класу `TreeConverter`, який відповідає за конвертацію дерева у TST:

```
def to_tst(self):
    values = self._extract_values()
    values = [v for v in values if v != 0]  # Remove 0 values
    values.sort()
    self._build_tst(values)
    return self

def _build_tst(self, values):
    if not values:
        return
```

```
    mid = len(values) // 2
    self.value = values[mid]
    left_values = [v for v in values[:mid] if v < self.value]
    center_values = [v for v in values if v == self.value]
    right_values = [v for v in values[mid+1:] if v > self.value]

    if left_values:
        self.children[0] = TreeConverter(left_values[len(left_values) // 2],
max_children=3)
        self.children[0]._build_tst(left_values)
    else:
        self.children[0] = None

    if len(center_values) > 1:
        self.children[1] = TreeConverter(center_values[len(center_values) //
2], max_children=3)
        self.children[1]._build_tst(center_values[len(center_values) // 2 +
1:])
    else:
        self.children[1] = None

    if right_values:
        self.children[2] = TreeConverter(right_values[len(right_values) // 2],
max_children=3)
        self.children[2]._build_tst(right_values)
    else:
        self.children[2] = None
```

Метод `to_tst` спочатку викликає метод `_extract_values`, який отримує всі значення з дерева. Потім значення сортуються та видаляються значення 0. Далі викликається метод `_build_tst`, який рекурсивно будує TST на основі відсортованих значень.

Метод `_build_tst` вибирає середній елемент зі списку значень `values` як корінь дерева. Потім розділяє значення на ліві (`left_values`), центральні (`center_values`) та праві (`right_values`) відносно значення кореня. Рекурсивно будує ліве, середнє та праве піддерева на основі відповідних списків значень.

Сподіваюсь, це допоможе краще зрозуміти теоретичні основи та їх зв'язок з реальним кодом проекту. Якщо є якісь запитання або потрібні додаткові пояснення, дайте мені знати.

*2.5. Функції пошуку в BST та TST*

Однією з ключових операцій в бінарних деревах пошуку (BST) та троїчних деревах пошуку (TST) є пошук елемента за ключем. Розглянемо метод `search` з класу `TreeConverter`, який виконує пошук значення як у BST, так і в TST:

```
def search(self, value):
    if self is None or self.value == 0:
        return False
    if self.value == value:
        return True
    if len(self.children) == 2:  # BST logic
        if value < self.value and self.children[0]:
            return self.children[0].search(value)
        elif value > self.value and self.children[1]:
            return self.children[1].search(value)
    elif len(self.children) == 3:  # TST logic
        if value < self.value and self.children[0]:
            return self.children[0].search(value)
        elif value > self.value and self.children[2]:
            return self.children[2].search(value)
        elif value == self.value and self.children[1]:
            return self.children[1].search(value)
    return False
```

Метод `search` приймає значення `value`, яке потрібно знайти в дереві. Пошук виконується рекурсивно, починаючи з кореня дерева.

Якщо дерево є BST (має два дочірні вузли), то:

- Якщо поточний вузол має шукане значення, метод повертає `True`.
- Якщо шукане значення менше за значення поточного вузла, пошук продовжується в лівому піддереві.
- Якщо шукане значення більше за значення поточного вузла, пошук продовжується в правому піддереві.

Якщо дерево є TST (має три дочірні вузли), то:

- Якщо поточний вузол має шукане значення, метод повертає `True`.
- Якщо шукане значення менше за значення поточного вузла, пошук продовжується в лівому піддереві.
- Якщо шукане значення більше за значення поточного вузла, пошук продовжується в правому піддереві.

- Якщо шукане значення дорівнює значенню поточного вузла, пошук продовжується в середньому піддереві.

Якщо значення не знайдено, метод повертає `False`.

## 3. Структура проекту

### *3.1. Опис структури проекту*

Проект складається з наступних файлів:

- `constants.py`: Містить константу `NODE_NUM_MAX`, яка визначає максимальну кількість вершин у бінарному дереві.
- `tree_converter.py`: Містить клас `TreeConverter`, який відповідає за перетворення дерев з рядкового представлення у внутрішнє представлення та навпаки, а також за конвертацію дерев у BST та TST.
- `tree_printer.py`: Містить клас `TreePrinter`, який відповідає за нумерацію бінарних дерев та генерацію дерева за номером.
- `main.py`: Головний файл проекту, який демонструє використання класів `TreeConverter` та `TreePrinter`, а також візуалізацію дерев.

### *3.2. Клас TreeConverter*

Клас `TreeConverter` відповідає за перетворення дерев з рядкового представлення у внутрішнє представлення та навпаки, а також за конвертацію дерев у BST та TST.

Основні методи класу:

- `from_string(tree_str, max_children)`: Перетворює дерево з рядкового представлення у внутрішнє представлення.
- `to_string()`: Перетворює дерево з внутрішнього представлення у рядкове представлення.
- `to_bst()`: Конвертує дерево у BST.
- `to_tst()`: Конвертує дерево у TST.
- `search(value)`: Виконує пошук значення у дереві (BST або TST).
- `visualize_networkx()`: Візуалізує дерево за допомогою бібліотеки NetworkX.

### *3.3. Клас TreePrinter*

Клас `TreePrinter` відповідає за нумерацію бінарних дерев та генерацію дерева за номером.

Основні методи класу:

- `print_tree(order_num)`: Повертає рядкове представлення дерева з номером `order_num` у нумерації.
- `_initialize()`: Ініціалізує допоміжні масиви для обчислення кількості дерев з певною кількістю вершин.
- `get_tree_num_product(children_node_nums)`: Обчислює добуток кількостей дерев для заданої конфігурації дочірніх вершин.
- `print_tree_helper(order_num, node_num)`: Рекурсивно генерує дерево з номером `order_num` та `node_num` вершинами.

### *3.4. Допоміжні функції*

Проект містить наступні допоміжні функції:

- `get_children_combinations(N, m)`: Генерує всі можливі комбінації кількості дочірніх вершин для вершини з `N` вершинами та `m` дочірніми вершинами.
- `get_indices(linear_index, dimensions)`: Перетворює лінійний індекс у багатовимірні індекси на основі розмірностей.

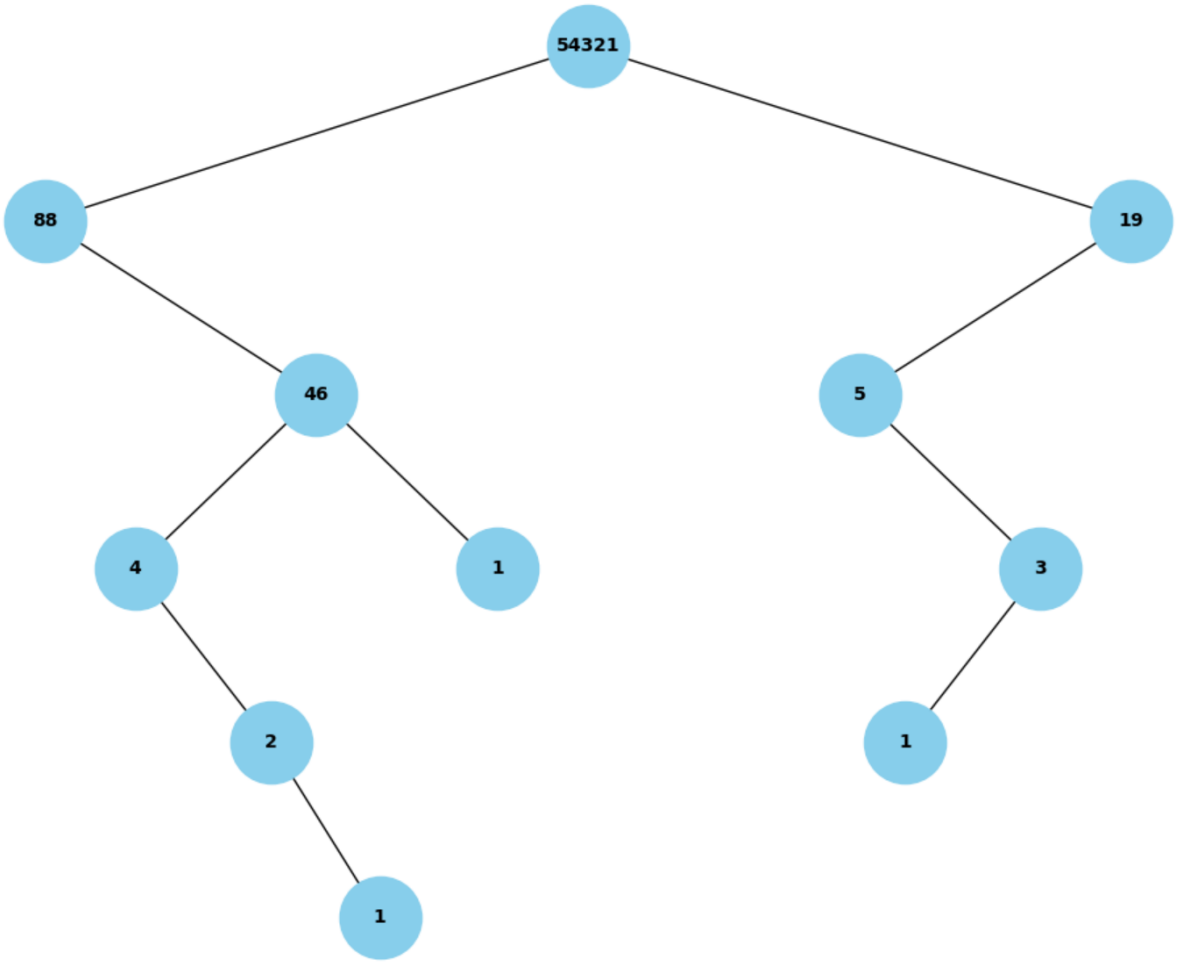
## 4. Аналіз та результати

### *4.1. Візуалізація дерев за допомогою бібліотеки NetworkX*

Для кращого розуміння структури та властивостей дерев, було використано бібліотеку NetworkX для візуалізації різних типів дерев.

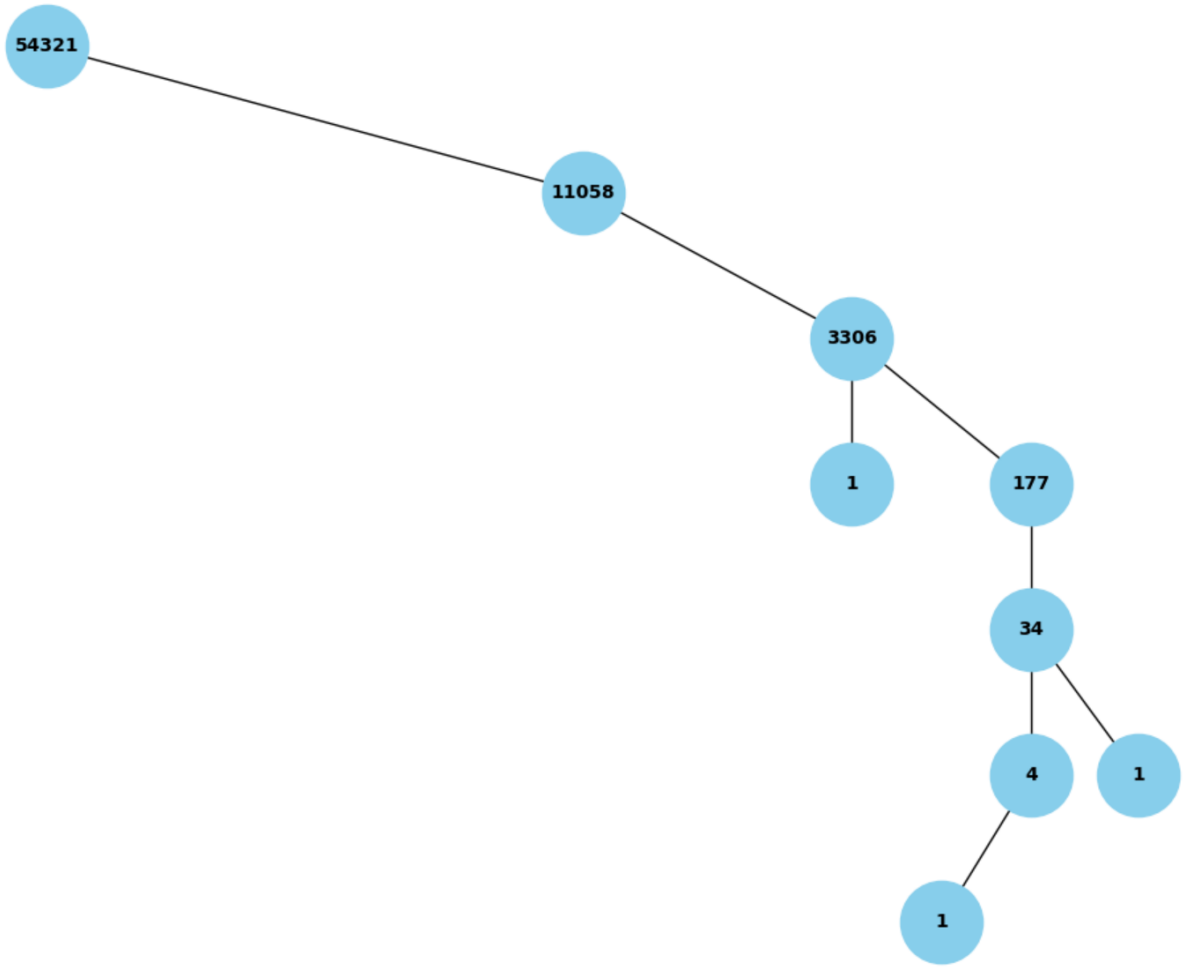
*Приклад нумерації бінарних дерев*

```
Please specify maximum number of children for each node: 2
54321
54321(88(0,46(4(0,2(0,1(0,0))),1(0,0))),19(5(0,3(1(0,0),0)),0))
```



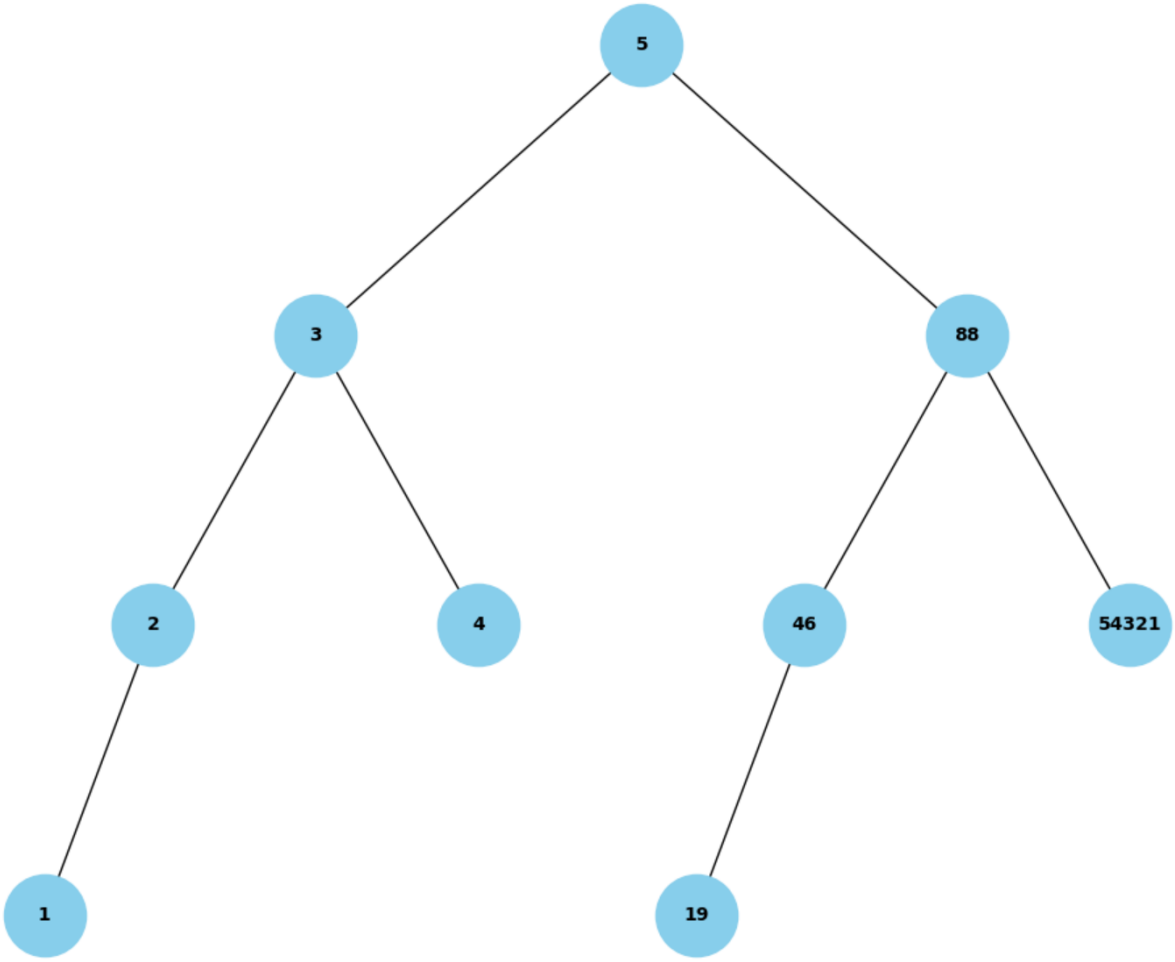

*Приклад нумерації троїчних дерев*

```
Please specify maximum number of children for each node: 3
54321
54321(0,0,11058(0,0,3306(0,1(0,0,0),177(0,34(0,4(1(0,0,0),0,0),1(0,0,0)),0))))
```



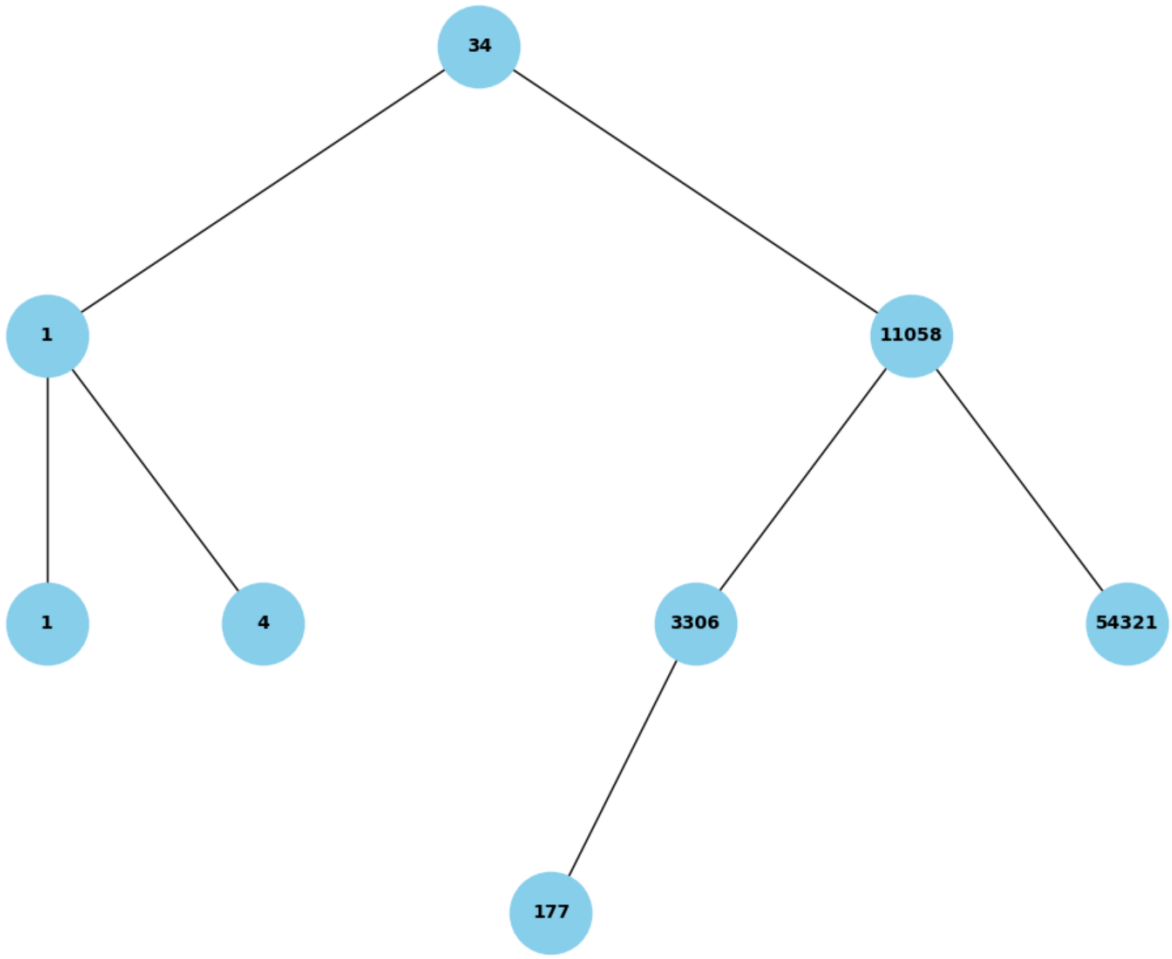

На зображеннях показано приклад нумерації бінарних та троїчних дерев з номером 54321.

*Приклад бінарного дерева пошуку (BST)*




Зображення демонструє приклад бінарного дерева пошуку (BST) створеного за допомогою конвертації бінарного дерева під номером 54321. У BST ключі всіх вершин у лівому піддереві менші за ключ кореня, а ключі всіх вершин у правому піддереві більші за ключ кореня. Така структура забезпечує ефективний пошук, вставку та видалення елементів.

*Приклад троїчного дерева пошуку (TST)*




На зображенні показано приклад троїчного дерева пошуку (TST) створеного за допомогою конвертації троїчного дерева під номером 54321. TST використовується для зберігання та пошуку рядків. Кожна вершина має три дочірні вершини: ліву, середню та праву.

*Приклад пошуку числа у троїчному дереві пошуку*

```
Enter a value to search in TST (or enter 0 to stop): 1
Value found in the TST.
Enter a value to search in TST (or enter 0 to stop): 2
Value not found in the TST.
Enter a value to search in TST (or enter 0 to stop): 11058
Value found in the TST.
Enter a value to search in TST (or enter 0 to stop): 124
Value not found in the TST.
Enter a value to search in TST (or enter 0 to stop): 4
Value found in the TST.
```

На зображені видно приклад пошуку чисел у троїчному дереві пошуку створеного за допомогою конвератації троїчного дерева під номером 54321.

*4.2. Варіанти застосування дерев*

Дерева мають широке застосування у різних галузях інформаційних технологій. Ось кілька прикладів використання дерев:

- Індексація та пошук даних у базах даних та файлових системах.
- Реалізація алгоритмів стиснення даних, таких як алгоритм Хаффмана.
- Представлення ієрархічних структур, таких як DOM-дерево у веб-розробці.
- Організація простору імен у програмуванні та системах управління файлами.
- Реалізація алгоритмів маршрутизації у комп'ютерних мережах.

Розуміння властивостей та алгоритмів роботи з деревами є важливим для ефективного використання цих структур даних у різноманітних застосуваннях.

## 5. Висновки

У даній роботі було досліджено алгоритми нумерації бінарних дерев та розглянуто бінарні дерева пошуку (BST) і троїчні дерева пошуку (TST) як приклади застосування бінарних дерев.

Основні результати роботи:

1. Вивчено теоретичні основи нумерації бінарних дерев, зокрема, числа Каталана та їх зв'язок з кількістю структурно унікальних бінарних дерев.
2. Реалізовано алгоритми нумерації бінарних дерев та генерації дерева за номером.
3. Досліджено властивості та операції з BST та TST, а також реалізовано алгоритми конвертації дерев у ці структури.
4. Використано бібліотеку NetworkX для візуалізації різних типів дерев та проілюстровано їх структуру та властивості.
5. Розглянуто варіанти застосування дерев у різних галузях інформаційних технологій.

Результати роботи показують, що алгоритми нумерації бінарних дерев дозволяють однозначно ідентифікувати кожне дерево та ефективно генерувати дерева за заданим номером. BST та TST є важливими структурами даних, які забезпечують ефективний пошук, вставку та видалення елементів за певними критеріями.

Подальші напрямки дослідження можуть включати:

- Оптимізацію алгоритмів нумерації та генерації дерев для роботи з великими деревами.
- Дослідження інших типів дерев, таких як АВЛ-дерева, червоно-чорні дерева тощо.
- Застосування дерев у конкретних прикладних задачах, таких як індексація даних, стиснення інформації, маршрутизація у мережах тощо.
- Реалізація конвертації дерева назад у число, для можливого подальшого кодування зображень.

## 6. Використані джерела


https://basecamp.eolymp.com/en/problems/2176

https://en.wikipedia.org/

https://www.geeksforgeeks.org/binary-search-tree-data-structure/

https://www.geeksforgeeks.org/ternary-search-tree/

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.