Relatório de Análise e Projeto de Algoritmos - Lista 01

Aluno - Balthazar Paixão

Repositório Github

Códigos da lista

Temas:

```
- Listas
- Pilhas
- Filas
- Recursão
- Complexidade de Algoritmos
- Árvores Binárias
```

Exercício 1

Escreva um programa que apresente os n primeiros números primos a partir do número 1 para um valor n>0 fornecido pelo usuário.

Resposta: A função `get_n_primes` traz os `n` primeiros números primos. Dado um `n` > 0 fornecido pelo usuario, a função itera sobre os números naturais até que a lista de primos tenha o tamanho `n`. Para cada número natural, é verificado se ele é primo. A verificação é feita através da divisão do número por todos os números naturais que o antecedem. Caso o número seja divisível por algum dos números naturais, ele não é primo e o processo se repete. Caso o número não seja divisível por nenhum dos números naturais, ele é primo e o processo se repete. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```python
def get_n_primes(n: int) -> list:
    """Returns a list of the first n prime numbers."""
    primes = []
    while len(primes) < n:
        is_prime = False
        prime = primes[-1] + 1 if primes else 2

        while not is_prime:
            for i in range(2, prime):
                if prime % i == 0:
                    prime += 1
                    break
            else:
                is_prime = True
                primes.append(prime)
    return primes
```

```
if __name__ == "__main__":
    print(get_n_primes(10))
```

Exercício 2

Faça um programa que leia um texto do usuário e conte o número de vogais que aparecem. O texto fornecido deve estar em um arquivo.

Resposta: A função `read_vowels` recebe um input proveniente do usuário e iterando sobre o texto, conta o número de vogais. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```
def read_vowels() -> None:
    txt = input("Digite o texto a ser lido: ")

    vowels = ["a", "e", "i", "o", "u"]

    count_vowels = 0

    for letter in txt:
        if letter in vowels:
            count_vowels += 1
        else:
            continue

    print(f"Total count of vowels is {count_vowels}")


if __name__ == "__main__":
    read_vowels()
```

Exercício 3

Escrever uma função (e um programa que execute tal função) que determine se uma matriz quadrada de dimensão n(n<100) é uma matriz de permutação. Uma matriz quadrada é chamada de matriz de permutação se seus elementos são apenas 0's e 1's e se em cada linha e coluna da matriz existe apena um único valor 1.

Resposta: A função `is_permutation_matrix` recebe uma matriz quadrada de dimensão n x n e verifica se ela é uma matriz de permutação. Para isso, ela itera sobre a matriz e conta o número de 1's em cada linha e coluna. Caso o número de 1's em cada linha e coluna seja igual, a matriz é uma matriz de permutação. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```
import numpy as np


def is_permutation_matrix(matrix: np.array) -> None:
```

```
    """Constraints matrix nxn - n < 100"""
    dim = len(matrix)
    one_col_counts = np.zeros(dim)
    one_row_counts = np.zeros(dim)
    for i in range(dim):
        for j in range(dim):
            if (matrix[i, j] > 1) or (matrix[i, j] < 0):
                break
            elif matrix[i, j] == 1:
                one_row_counts[i] += 1
                one_col_counts[j] += 1
            else:
                continue
    if (one_col_counts == one_row_counts).all():
        print("Is permutation matrix")
    else:
        print("Is not permutation matrix")


if __name__ == "__main__":
    matrix = np.array([[0, 3, 1, 0], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0,
1]])
    is_permutation_matrix(matrix=matrix)
```

Exercício 4

Escreva o algoritmo de busca binária (na forma recursiva e não recursiva) e faça a análise de tempo de execução do pior caso de cada algoritmo.

a) Sem recursão

Resposta: A função `binary_search_non_recursive` recebe uma lista ordenada e um valor e retorna o índice do valor na lista. Para isso, ele sempre pega o valor presente na metade da lista. Caso essevalor seja menor que o que esteja sendo procurado, ele pega a metade da lista a partir do valor atual. Caso o valor seja maior que o que esteja sendo procurado, ele pega a metade da lista até o valor atual. O processo se repete até que o valor seja encontrado, ou não. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```
import time


def binary_search_non_recursive(elems: list, value: int):
    """Given an ordered list named elems"""
    start_time = time.time()
    start = 0
    end = len(elems)
    while start <= end:
        mid = int((start + end) // 2)

        actual = elems[mid]
```

```
        if actual == value:
            break
        elif actual < value:
            start = mid + 1
        else:
            end = mid - 1

    end_time = time.time()
    print(mid)
    print(f"Time elapsed: {end_time - start_time}")


if __name__ == "__main__":
    elems = list(range(1000000))
    binary_search_non_recursive(elems, 999999)
```

b) Com recursão

O algoritmo com recursão tem o mesmo objetivo e a lógica bem parecida com o algoritmo sem recursão. A diferença é que ele chama a si mesmo para fazer a busca ao invés de usar um loop e atualizar o valor em `mid`. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo e considero mais simples que o anterior, sem recursão.

```
import time


def binary_search_recursive(elems: list, value: int):
    """Given an ordered list named elems"""
    start_time = time.time()
    start = 0
    end = len(elems)
    mid = int((start + end) // 2)

    actual = elems[mid]

    if actual == value:
        end_time = time.time()
        print(actual)
        print(f"Time elapsed: {end_time - start_time}")
    elif actual < value:
        binary_search_recursive(elems[mid + 1 :], value)
    else:
        binary_search_recursive(elems[: mid - 1], value)


if __name__ == "__main__":
    elems = list(range(1000000))
    binary_search_recursive(elems, 500000)
```

Exercício 5

Explique por que a declaração “O tempo de execução do algoritmo A é no mínimo O(n^2) ” não tem significado.

Resposta: Dizer que um algoritmo tem no mínimo O(n^2) nos informa apenas o melhor caso do algoritmo em tempo de execução. Para entendermos melhor o funcionamento de um algoritmo é necessário também traçar um limite superior para o tempo de execução, ou seja, o pior caso. Dessa forma é possível entender entre quais intervalos o algoritmo consegue performar.

Exercício 6

Indique para cada par de expressões (A, B) se A é O, o, Ω,Θ e ω de B. Considere: a) (n^3, nlogn) b) (nlogn, n^logn) c) (logn^k, n^logn)

a) A é O(n^3) b) A é Θ(n^logn) e ω(nlogn). c) A é Θ(n^logn) e ω(logn^k).

Exercício 7

Escreva uma função para trocar os elementos m e n de uma lista simplesmente encadeada (m e n podem ser chaves ou mesmo ponteiros para os elementos – a escolha é sua).

Resposta: Para a execução do exercício foi criada a classe referente ao nó e a classe referenta à lista encadeada. Métodos de inserção, remoção e busca foram implementados. O método `switch` recebe dois valores e troca a posição dos nós na lista. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```
import random
import random


class NodeList:
    """Class that represents a node in a linked list"""

    def __init__(self, value: int, next: "NodeList" = None):
        self.value = value
        self.next = next

    def __repr__(self) -> str:
        return f"NodeList({self.value} -> {self.next})"

class LinkedList:
    """Class that represents a linked list"""

    def __init__(self):
        self.head = None

    def __repr__(self) -> str:
        return f"LinkedList({self.head})"

    def __iter__(self):
```

```
        node = self.head
        while node:
            yield node
            node = node.next

    def __len__(self):
        return len(list(iter(self)))

    def print(self):
        for node in self:
            print(node.value, end=" -> ")
        print("None")

    def insert(self, value: int) -> NodeList:
        """Inserts a new node at the beginning of the linked list"""
        node = NodeList(value)
        node.next = self.head
        self.head = node

    def search(self, value: int) -> NodeList:
        """Searches for a node with the given value"""
        node = self.head
        while node and node.value != value:
            node = node.next
        return node

    def delete(self, value: int) -> NodeList:
        node = self.head

        if self.head.value == value:
            self.head = self.head.next

        else:
            prev = None
            actual = self.head

            while actual and actual.value != value:
                prev = actual
                actual = actual.next

            if actual:
                prev.next = actual.next
                actual.next = None
                return actual
            else:
                prev.next = None

    def switch(self, m: int, n: int) -> NodeList:
        """Switches the position of two nodes in the linked list"""

        prev_m = None
        prev_n = None
        actual_m = self.head
        actual_n = self.head
```

```
        while actual_m and actual_m.value != m:
            prev_m = actual_m
            actual_m = actual_m.next

        while actual_n and actual_n.value != n:
            prev_n = actual_n
            actual_n = actual_n.next

        if actual_m and actual_n:
            if prev_m:
                prev_m.next = actual_n
            else:
                self.head = actual_n

            if prev_n:
                prev_n.next = actual_m
            else:
                self.head = actual_n

            temp = actual_m.next
            actual_m.next = actual_n.next
            actual_n.next = temp

if __name__ == "__main__":
    values_to_add = list(range(10))
    random.shuffle(values_to_add)
    m, n = random.sample(values_to_add, 2)
    print(f"m: {m}, n: {n}")
    LinkedList = LinkedList()

    for value in values_to_add:
        LinkedList.insert(value)

    LinkedList.print()
    LinkedList.switch(m, n)
    LinkedList.print()
```

Exercício 8

Escreva uma função void MoveMenor(TipoLista Lista) que, dada uma lista com um número qualquer de elementos, acha o menor elemento da lista e o move para o começo da lista, como exemplificado na figura abaixo. (Obs. Não vale trocar apenas os campos item ou usar uma lista / fila / pilha auxiliar! Você deverá fazer a manipulação dos apontadores para trocar as células de posição).

Resposta: Para a execução do exercício foi criada a classe referente ao nó e a classe referenta à lista encadeada. Métodos de inserção, remoção e busca foram implementados. O método `move_smaller` encontra o menor valor na lista e o move para o início da lista (`self.head`). Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```python
import random


class NodeList:
    """Class that represents a node in a linked list"""
    def __init__(self, value: int, next: "NodeList" = None):
        self.value = value
        self.next = next

    def __repr__(self) -> str:
        return f"NodeList({self.value} -> {self.next})"


class LinkedList:
    """Class that represents a linked list"""

    def __init__(self):
        self.head = None

    def __repr__(self) -> str:
        return f"LinkedList({self.head})"

    def __iter__(self):
        node = self.head
        while node:
            yield node
            node = node.next

    def __len__(self):
        return len(list(iter(self)))

    def print(self):
        for node in self:
            print(node.value, end=" -> ")
        print("None")

    def insert(self, value: int) -> NodeList:
        """Inserts a new node at the beginning of the linked list"""
        node = NodeList(value)
        node.next = self.head
        self.head = node

    def search(self, value: int) -> NodeList:
        """Searches for a node with the given value"""
        node = self.head
        while node and node.value != value:
            node = node.next
        return node

    def delete(self, value: int) -> NodeList:
        if self.head.value == value:
            self.head = self.head.next
```

```
            else:
                prev = None
                actual = self.head

                while actual and actual.value != value:
                    prev = actual
                    actual = actual.next

                if actual:
                    prev.next = actual.next
                    actual.next = None
                    return actual
                else:
                    prev.next = None

        def move_smaller(self):
            """Moves the smallest number to the head of the linked list"""

            prev = None
            smallest = self.head
            current = self.head.next

            while current:
                if current.value < smallest.value:
                    smallest = current
                current = current.next

            if smallest != self.head:
                prev = self.head
                while prev.next != smallest:
                    prev = prev.next

                prev.next = smallest.next
                smallest.next = self.head
                self.head = smallest


    if __name__ == "__main__":
        values_to_add = list(range(10))
        random.shuffle(values_to_add)
        LinkedList = LinkedList()

        for value in values_to_add:
            LinkedList.insert(value)

        LinkedList.print()
        LinkedList.move_smaller()
        LinkedList.print()
```

Exercício 10

Escreva um procedimento não recursivo, com tempo de execução Θ(n) que inverta uma lista simplesmente encadeada de n elementos. Além do custo de armazenar os n elementos, o procedimento não deve gastar mais do que O(1) para inverter a lista.

Resposta: Para a execução do exercício foi criada a classe referente ao nó e a classe referenta à lista encadeada. Métodos de inserção, remoção e busca foram implementados. O método `reverse` inverte a lista encadeada. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo, no entanto o nível é um pouco maior que os exercícios anteriores.

```
import random


class NodeList:
    """Class that represents a node in a linked list"""

    def __init__(self, value: int, next: "NodeList" = None):
        self.value = value
        self.next = next

    def __repr__(self) -> str:
        return f"NodeList({self.value} -> {self.next})"


class LinkedList:
    """Class that represents a linked list"""

    def __init__(self):
        self.head = None

    def __repr__(self) -> str:
        return f"LinkedList({self.head})"

    def __iter__(self):
        node = self.head
        while node:
            yield node
            node = node.next

    def __len__(self):
        return len(list(iter(self)))

    def print(self):
        for node in self:
            print(node.value, end=" -> ")
        print("None")

    def insert(self, value: int) -> NodeList:
        """Inserts a new node at the beginning of the linked list"""
        node = NodeList(value)
        node.next = self.head
        self.head = node
```

```
    def search(self, value: int) -> NodeList:
        """Searches for a node with the given value"""
        node = self.head
        while node and node.value != value:
            node = node.next
        return node

    def delete(self, value: int) -> NodeList:
        if self.head.value == value:
            self.head = self.head.next

        else:
            prev = None
            actual = self.head

            while actual and actual.value != value:
                prev = actual
                actual = actual.next

            if actual:
                prev.next = actual.next
                actual.next = None
                return actual
            else:
                prev.next = None

    def reverse(self):
        previous = None
        current = self.head

        while current:
            next_node = current.next
            current.next = previous
            previous = current
            current = next_node

        self.head = previous


if __name__ == "__main__":
    values_to_add = list(range(10))
    random.shuffle(values_to_add)
    LinkedList = LinkedList()

    for value in values_to_add:
        LinkedList.insert(value)

    LinkedList.print()
    LinkedList.reverse()
    LinkedList.print()
```

Exercício 11

Desenvolva um método para manter duas pilhas dentro de um único vetor linear (um arranjo) de modo que nenhuma das pilhas incorra em estouro até que toda a memória seja usada, e toda uma pilha nunca seja deslocada para outro local dentro do vetor.

Resposta: Tive problemas para entender o que o exercício pedia, está confuso o enunciado.

```
class DoubleStack:
    def __init__(self, size):
        self.size = size
        self.array = [None] * size
        self.top1 = -1
        self.top2 = size

    def push1(self, element):
        if self.top1 < self.top2 - 1:
            self.top1 += 1
            self.array[self.top1] = element
        else:
            print("Stack Overflow")
            exit(1)

    def push2(self, element):
        if self.top1 < self.top2 - 1:
            self.top2 -= 1
            self.array[self.top2] = element
        else:
            print("Stack Overflow")
            exit(1)

    def pop1(self):
        if self.top1 >= 0:
            element = self.array[self.top1]
            self.top1 -= 1
            return element
        else:
            print("Stack Underflow")
            exit(1)

    def pop2(self):
        if self.top2 < self.size:
            element = self.array[self.top2]
            self.top2 += 1
            return element
        else:
            print("Stack Underflow")
            exit(1)

    def print_stack(self):
        print("Stack 1: ", end="")
        for i in range(0, self.top1 + 1):
            print(self.array[i], end=" ")
        print()
```

```
        print("Stack 2: ", end="")
        for i in range(self.size - 1, self.top2 - 1, -1):
            print(self.array[i], end=" ")
        print()


import random


def test_stack_overflow():
    double_stack = DoubleStack(5)
    for _ in range(5):
        double_stack.push1(random.randint(0, 100))
        double_stack.push2(random.randint(0, 100))
        double_stack.print_stack()


def test_stack_underflow():
    double_stack = DoubleStack(11)
    for _ in range(5):
        double_stack.push1(random.randint(0, 100))
        double_stack.push2(random.randint(0, 100))
        double_stack.print_stack()
    double_stack.print_stack()
    double_stack.pop1()
    double_stack.pop1()
    double_stack.pop1()
    double_stack.pop1()
    double_stack.pop1()
    double_stack.pop1()
    double_stack.print_stack()


def test_stack():
    double_stack = DoubleStack(10)
    for _ in range(5):
        double_stack.push1(random.randint(0, 100))
        double_stack.push2(random.randint(0, 100))
        double_stack.print_stack()
    double_stack.print_stack()


def test():
    try:
        test_stack()
        print()
    except:
        print()

    try:
        test_stack_underflow()
    except:
        print()
```

```
    try:
        test_stack_overflow()
    except:
        print()


if __name__ == "__main__":
    test()
```

Exercício 12

Faça um programa para simular um controlador de voo de um aeroporto. Neste programa o usuário deve ser capaz de realizar as seguintes tarefas:

- Listar o número de aviões esperando para decolar;
- Autorizar a decolagem do primeiro avião na fila;
- Adicionar um avião na fila de espera;
- Listar todos os aviões que estão na lista de espera;
- Listar as características do primeiro avião da fila;

Considere que uma estrutura de dados do tipo fila seja usada para manipular os dados e que cada avião possui um nome, um identificador, uma origem e um destino. Se quiser coloque mais informações, nº de passageiros, capacidade, modelo, etc.

Resposta: Para a execução do exercício foi criada a classe referente ao voo e a classe referenta ao controlador de voo. Métodos de inserção, remoção e busca foram implementados. Nenhuma dificuldade foi encontrada ao desenvolver o algoritmo.

```
class Flight:
    def __init__(self, name, id, origin, destination):
        self.name = name
        self.id = id
        self.origin = origin
        self.destination = destination

    def __str__(self):
        return self.name


class FlightController:
    def __init__(self):
        self.queue = []

    def list_waiting_flights(self):
        print("Waiting flights:")
        for flight in self.queue:
            print(flight)
```

```
    def list_waiting_flights_details(self):
        print("Waiting flights details:")
        for flight in self.queue:
            print(
                f"Flight: {flight.name}, ID: {flight.id}, Origin:
{flight.origin}, Destination: {flight.destination}"
            )

    def add_flight(self, flight):
        self.queue.append(flight)

    def authorize_takeoff(self):
        if len(self.queue) > 0:
            flight = self.queue.pop(0)
            print(f"Flight {flight.name} authorized to takeoff")
        else:
            print("No flight waiting for takeoff")

    def list_first_flight(self):
        if len(self.queue) > 0:
            print(f"First flight: {self.queue[0]}")
        else:
            print("No flight waiting for takeoff")

    def list_waiting_flights_size(self):
        print(f"Waiting flights size: {len(self.queue)}")


def test_flight_controller() -> None:
    flight_controller = FlightController()
    flight_controller.list_waiting_flights_size()
    flight_controller.list_waiting_flights()
    flight_controller.list_first_flight()
    flight_controller.authorize_takeoff()

    flight_one = Flight("LATAM 7246", 1, "Rio de Janeiro", "São Paulo")
    flight_two = Flight("GOL 1234", 2, "São Paulo", "Rio de Janeiro")
    flight_three = Flight("AZUL 5678", 3, "São Paulo", "Rio de Janeiro")
    flight_controller.add_flight(flight_one)
    flight_controller.add_flight(flight_two)
    flight_controller.add_flight(flight_three)

    flight_controller.list_waiting_flights_size()
    flight_controller.list_waiting_flights()
    flight_controller.list_first_flight()
    flight_controller.authorize_takeoff()

    flight_controller.list_waiting_flights_size()
    flight_controller.list_waiting_flights()
    flight_controller.list_first_flight()
    flight_controller.authorize_takeoff()

    flight_controller.list_waiting_flights_size()
    flight_controller.list_waiting_flights()
```

```
    flight_controller.list_first_flight()
    flight_controller.authorize_takeoff()

    flight_controller.list_waiting_flights_size()
    flight_controller.list_waiting_flights()


if __name__ == "__main__":
    test_flight_controller()
```

Exercício 13

Quantos antecedentes tem um nó no nível n em uma árvore binária? Prove sua resposta.

Supondo uma árvore binária cheia, o número de antecedentes de um nó no nível n é dado por 2^n - 1. Podemos provar da seguinte forma: a cada nível, o número de nós é dado por 2^n, visto que cada nó tem dois filhos. O número de antecedentes é dado pela soma dos nós de todos os níveis anteriores, ou seja, 2^0 + 2^1 + 2^2 + ... + 2^n-1. Podemos reescrever essa soma como 2^0 + 2^1 + 2^2 + ... + 2^n-1 + 2^n - 2^n. O que nos dá 2^n - 1.

Exercício 14

Implemente um algoritmo que determine se uma árvore binária é: (a) estritamente binária; (b) completa; (c) quase completa

Resposta: Para a execução do exercício foi criada a classe referente ao nó e a classe referenta à árvore binária. Métodos de inserção, remoção e busca foram implementados. Os métodos `is_strictly_binary`, `is_complete` e `is_almost_full` verificam se a árvore é estritamente binária, completa e quase completa, respectivamente. Tive dificuldade em pensar as lógicas de verificação da árvore.

```
import random


class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def is_leaf(self):
        return self.left is None and self.right is None

    def is_full(self):
        return self.left is not None and self.right is not None

    def is_almost_full(self):
        return (
            self.left is not None
            and self.right is None
```

```
            or self.left is None
            and self.right is not None
        )


class BinaryTree:
    def __init__(self):
        self.root = None

    def is_strictly_binary(self):
        return self.is_strictly_binary_recursive(self.root)

    def is_strictly_binary_recursive(self, node):
        if node is None:
            return True
        if node.is_leaf():
            return True
        if node.is_full():
            return self.is_strictly_binary_recursive(
                node.left
            ) and self.is_strictly_binary_recursive(node.right)
        return False

    def is_complete(self):
        return self.is_complete_recursive(self.root)

    def is_complete_recursive(self, node):
        if node is None:
            return True
        if node.is_leaf():
            return True
        if node.is_full():
            return self.is_complete_recursive(node.left) and
self.is_complete_recursive(
                node.right
            )
        return False

    def is_almost_full(self):
        return self.is_almost_full_recursive(self.root)

    def is_almost_full_recursive(self, node):
        if node is None:
            return True
        if node.is_leaf():
            return True
        if node.is_almost_full():
            return self.is_almost_full_recursive(
                node.left
            ) and self.is_almost_full_recursive(node.right)
        return False

    def insert(self, value):
        if self.root is None:
```

```
            self.root = Node(value)
        else:
            self.insert_recursive(self.root, value)

    def insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self.insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                self.insert_recursive(node.right, value)

    def remove(self, value):
        if self.root is None:
            return
        self.remove_recursive(self.root, value)

    def remove_recursive(self, node, value):
        if node is None:
            return None
        if value < node.value:
            node.left = self.remove_recursive(node.left, value)
        elif value > node.value:
            node.right = self.remove_recursive(node.right, value)
        else:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            else:
                node.value = self.find_min(node.right)
                node.right = self.remove_recursive(node.right, node.value)
        return node

    def print_tree(self, node=None, level=0, prefix="Root: "):
        if node is None:
            node = self.root

        if node is not None:
            print(" " * (level * 4) + prefix + str(node.value))
            if node.is_full():
                self.print_tree(node.left, level + 1, "L--- ")
                self.print_tree(node.right, level + 1, "R--- ")
            elif node.left is not None:
                self.print_tree(node.left, level + 1, "L--- ")
            elif node.right is not None:
                self.print_tree(node.right, level + 1, "R--- ")


def test_random_tree() -> None:
```

```
    random_tree = BinaryTree()
    for _ in range(10):
        random_tree.insert(random.randint(0, 100))
    random_tree.print_tree()
    print("Strictly Binary: ", random_tree.is_strictly_binary())
    print("Complete: ", random_tree.is_complete())
    print("Almost Complete: ", random_tree.is_almost_full())


def test_complete_tree() -> None:
    complete_tree = BinaryTree()
    complete_tree.insert(5)
    complete_tree.insert(4)
    complete_tree.insert(7)
    complete_tree.insert(2)
    complete_tree.insert(4)
    complete_tree.insert(6)
    complete_tree.insert(8)
    complete_tree.print_tree()
    print("Strictly Binary: ", complete_tree.is_strictly_binary())
    print("Complete: ", complete_tree.is_complete())
    print("Almost Complete: ", complete_tree.is_almost_full())


def generate_almost_full_tree(levels):
    tree = BinaryTree()
    values = list(range(1, 2 ** (levels - 1) + 1))
    for value in values:
        tree.insert(value)
    return tree


def test_almost_full_tree():
    almost_full_tree = generate_almost_full_tree(4)
    almost_full_tree.print_tree()
    print("Strictly Binary: ", almost_full_tree.is_strictly_binary())
    print("Complete: ", almost_full_tree.is_complete())
    print("Almost Complete: ", almost_full_tree.is_almost_full())


if __name__ == "__main__":
    test_random_tree()
    test_complete_tree()
    test_almost_full_tree()
```

Exercício 15

Duas árvores binárias são similares se elas são vazias ou se elas não são vazias e suas subárvores da esquerda são similares e suas subárvores da direita são também similares. Escreva um programa para determinar se duas árvores binárias são similares

Resposta: Para a execução do exercício foi criada a classe referente ao nó e a classe referenta à árvore binária. Métodos de inserção, remoção e busca foram implementados. O método `is_similar` verifica se duas árvores são similares. Encontrei dificuldade em pensar as lógicas de verificação da árvore.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def is_leaf(self):
        return self.left is None and self.right is None

    def is_full(self):
        return self.left is not None and self.right is not None


class BinaryTree:
    def __init__(self):
        self.root = None

    def is_similar(self, tree):
        return self.is_similar_recursive(self.root, tree.root)

    def is_similar_recursive(self, node1, node2):
        if node1 is None and node2 is None:
            return True
        if node1 is not None and node2 is not None:
            return self.is_similar_recursive(
                node1.left, node2.left
            ) and self.is_similar_recursive(node1.right, node2.right)
        return False

    def insert(self, value):
        if self.root is None:
            self.root = Node(value)
        else:
            self.insert_recursive(self.root, value)

    def insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self.insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                self.insert_recursive(node.right, value)

    def print_tree(self, node=None, level=0, prefix="Root: "):
```

```
        if node is None:
            node = self.root

        if node is not None:
            print(" " * (level * 4) + prefix + str(node.value))
            if node.is_full():
                self.print_tree(node.left, level + 1, "L--- ")
                self.print_tree(node.right, level + 1, "R--- ")
            elif node.left is not None:
                self.print_tree(node.left, level + 1, "L--- ")
            elif node.right is not None:
                self.print_tree(node.right, level + 1, "R--- ")


def test_is_similar():
    tree1 = BinaryTree()
    tree1.insert(3)
    tree1.insert(1)
    tree1.insert(2)
    tree1.insert(4)
    tree1.insert(5)
    tree1.print_tree()

    tree2 = BinaryTree()
    tree2.insert(5)
    tree2.insert(2)
    tree2.insert(3)
    tree2.insert(7)
    tree2.insert(8)
    tree2.print_tree()

    if tree1.is_similar(tree2):
        print("Tree 1 and Tree 2 are similar")
    else:
        print("Tree 1 and Tree 2 are not similar")


def test_is_not_similar():
    tree1 = BinaryTree()
    tree1.insert(3)
    tree1.insert(1)
    tree1.insert(2)
    tree1.insert(4)
    tree1.insert(5)
    tree1.print_tree()

    tree2 = BinaryTree()
    tree2.insert(5)
    tree2.insert(8)
    tree2.insert(3)
    tree2.insert(2)
    tree2.insert(7)
    tree2.print_tree()
```

```
    if tree1.is_similar(tree2):
        print("Tree 1 and Tree 2 are similar")
    else:
        print("Tree 1 and Tree 2 are not similar")


if __name__ == "__main__":
    test_is_similar()
    print()
    test_is_not_similar()
```