Lista de exercícios Estrutura de Dados 2

Nome: Richard Henry Hartmann – RGM:

29701961Curso: Ciência da Computação –

Turma: 4B

richardhartmann2@gmail.com

Universidade da Cidade de São Paulo (UNICID) - Rua Cesário Galeno, 448/475 SãoPaulo – SP – Brasil – CEP: 03071-000

Abstract. *The objective of this project is to provide a solution for a particular investigation into circular lists in Data Structures 2, using the knowledge acquired in classes and innovative problem-solving techniques. Using the complementary materials offered by professor Juliano Ratusznei, it is possible to understand the essential components of the task, divide it into manageable parts and, finally, reach a coherent resolution that satisfies the problem at hand.*

Resumo. O objetivo deste projeto é fornecer uma solução para uma investigação particular sobre listas circulares em Estruturas de Dados 2, utilizando os conhecimentos adquiridos nas aulas e técnicas inovadoras de resolução de problemas. Utilizando os materiais complementares oferecidos pelo professor Juliano Ratusznei, é possível compreender os componentes essenciais da tarefa, dividi-la em partes gerenciáveis e, por fim, chegar a uma resolução coerente que satisfaça o problema em questão.

São Paulo – 2023

Descrição do exercício a ser solucionado:

- Implemente uma lista circular com identificador da composição de trens do metrô a qual tenha as seguintes características:
  - Alocação de memória dinâmica para os vagões inseridos.
  - Uma função de busca de vagões por trem.
  - Contagem da quantidade de vagões presentes na composição.
  - Remoção de vagões.
  - Adição de vagões.
  - O último vagão deve permitir a acoplagem de máquinas ou vagões.

Resolução do exercício – Codificação:

```
# Definindo a classe Vagao para representar um vagão individual
class Vagao:
    def __init__(self, numero):
        self.numero = numero
        self.proximo = None

# Definindo a classe ComposicaoTrem para representar a composição de trens
class ComposicaoTrem:
    def __init__(self):
        self.primeiro = None    # Ponteiro para o primeiro vagão na composição
        self.ultimo = None      # Ponteiro para o último vagão na composição
        self.tamanho = 0        # Armazena a quantidade de vagões na composição

    # Função para adicionar um vagão à composição
    def adicionar_vagao(self, numero):
        novo_vagao = Vagao(numero)  # Cria um novo vagão com o número especificado
        if not self.primeiro:
            self.primeiro = novo_vagao
            self.ultimo = novo_vagao
        else:
            self.ultimo.proximo = novo_vagao
```

```
        novo_vagao.proximo = self.primeiro  # Tornando a lista circular
        self.ultimo = novo_vagao
    self.tamanho += 1  # Incrementando a contagem de vagões

# Função para remover um vagão da composição
def remover_vagao(self, numero):
    if not self.primeiro:
        print("A composição está vazia.")
        return
    if self.primeiro.numero == numero:
        if self.primeiro == self.ultimo:
            self.primeiro = None
            self.ultimo = None
        else:
            self.primeiro = self.primeiro.proximo
            self.ultimo.proximo = self.primeiro
        self.tamanho -= 1
        print(f"Vagão {numero} removido com sucesso.")
        return

    vagao_anterior = self.primeiro
    vagao_atual = self.primeiro.proximo

    while vagao_atual != self.primeiro:
        if vagao_atual.numero == numero:
            vagao_anterior.proximo = vagao_atual.proximo
            self.tamanho -= 1
            print(f"Vagão {numero} removido com sucesso.")
            return
        vagao_anterior = vagao_atual
```

```
            vagao_atual = vagao_atual.proximo

        print(f"Vagão {numero} não encontrado na composição.")

    # Função para buscar vagoes por trem
    def buscar_vagoes_por_trem(self, numero_trem):
        if not self.primeiro:
            print("A composição está vazia.")
            return

        vagoes_encontrados = []
        vagao_atual = self.primeiro

        while True:
            if vagao_atual.numero // 100 == numero_trem:
                vagoes_encontrados.append(vagao_atual.numero)
            vagao_atual = vagao_atual.proximo
            if vagao_atual == self.primeiro:
                break

        if not vagoes_encontrados:
            print(f"Nenhum vagão encontrado para o trem {numero_trem}.")
        else:
            print(f"Vagoes do trem {numero_trem}: {', '.join(map(str,
vagoes_encontrados))}")

    # Função para contar a quantidade de vagões na composição
    def contar_vagoes(self):
        return self.tamanho
```

```
    # Função para acoplar uma máquina ou vagão (adiciona um vagão ao final da composição)
    def acoplar_maq_vagao(self, numero_maq_vagao):
        self.adicionar_vagao(numero_maq_vagao)

    # Representação em string da composição de trens
    def __str__(self):
        if not self.primeiro:
            return "A composição está vazia."
        composicao_str = []
        vagao_atual = self.primeiro

        while True:
            composicao_str.append(str(vagao_atual.numero))
            vagao_atual = vagao_atual.proximo
            if vagao_atual == self.primeiro:
                break

        return " -> ".join(composicao_str)

# Exemplo de uso interativo com input do usuário:
composicao = ComposicaoTrem()

while True:
    print("\nOpções:")
    print("1. Adicionar vagão")
    print("2. Remover vagão")
    print("3. Buscar vagoes por trem")
    print("4. Contar vagoes")
    print("5. Acoplar máquina/vagão")
    print("6. Exibir composição")
```

```
    print("7. Sair")

    escolha = input("Escolha uma opção: ")

    if escolha == '1':
        numero_vagao = int(input("Digite o número do vagão a ser adicionado: "))
        composicao.adicionar_vagao(numero_vagao)
    elif escolha == '2':
        numero_vagao = int(input("Digite o número do vagão a ser removido: "))
        composicao.remover_vagao(numero_vagao)
    elif escolha == '3':
        numero_trem = int(input("Digite o número do trem para buscar vagoes: "))
        composicao.buscar_vagoes_por_trem(numero_trem)
    elif escolha == '4':
        print(f"Quantidade de vagoes na composição: {composicao.contar_vagoes()}")
    elif escolha == '5':
        numero_maq_vagao = int(input("Digite o número da máquina/vagão a ser
acoplado: "))
        composicao.acoplar_maq_vagao(numero_maq_vagao)
    elif escolha == '6':
        print("Composição de trens:")
        print(composicao)
    elif escolha == '7':
        break
    else:
        print("Opção inválida. Tente novamente.")
```

Execução do exercício – Resolução

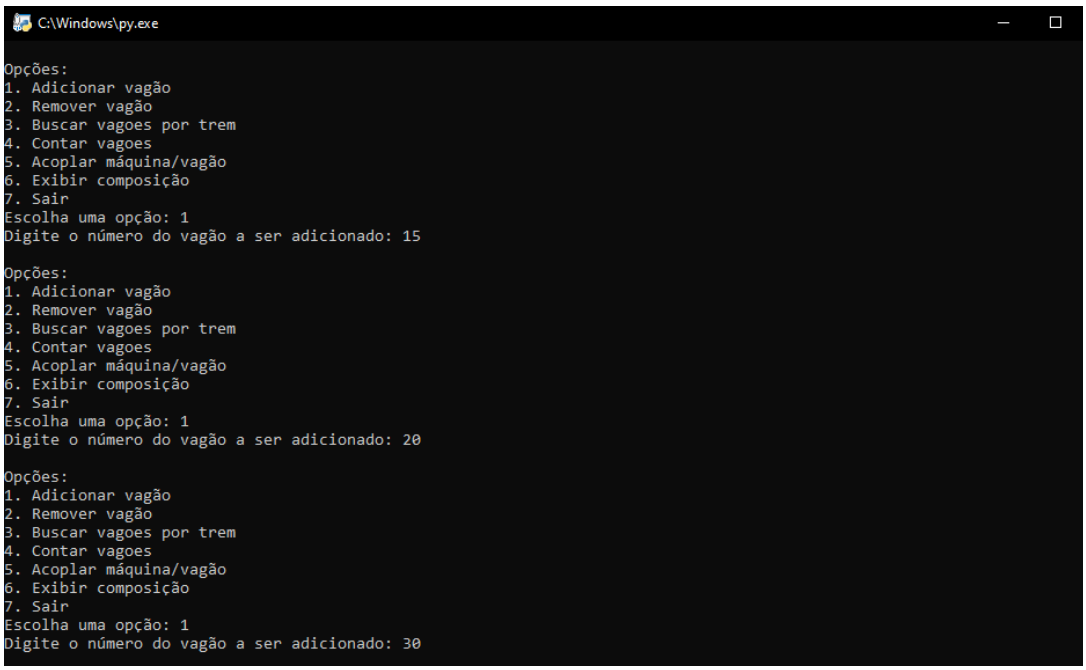

Figura 1: tela da solução do exercício – adicionando vagões (15, 20 e 30).

Figura 2: tela da solução do exercício – removendo vagões (20).

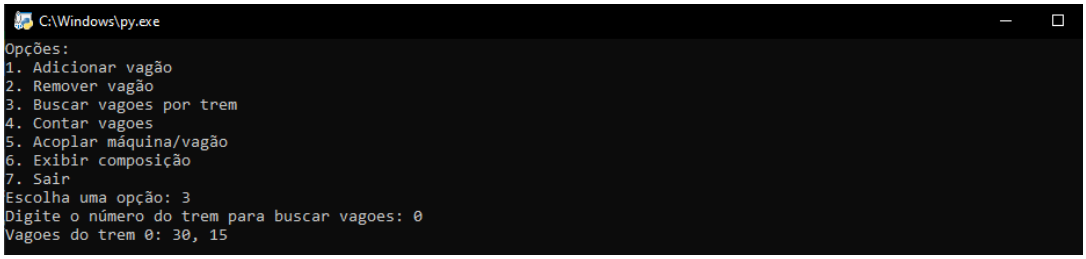

Figura 3: tela da solução do exercício – buscando vagões (30 e 15).

Figura 4: tela da solução do exercício – contando vagões (2).

Figura 5: tela da solução do exercício – acoplando máquinas/vagões à composição (0).

Figura 6: tela da solução do exercício – exibindo a composição (15, 30 e 0).

Descrição da aprendizagem obtida através da problemática:

Inicialmente, o código emprega classes para exemplificar vagões singulares e a configuração dos trens. Isso serve para ilustrar como as classes Python podem ser utilizadas com o propósito de representar entidades tangíveis de maneira estruturada. Isto é crucial em termos de gerenciamento e compartimentação de dados e funcionalidades. Um método intrigante de lidar com configurações de trens é a utilização de uma lista circular. Isto envolve conectar o vagão final ao vagão inicial, estabelecendo um circuito fechado. Isso serve como um exemplo de como estruturas de dados complexas podem ser empregadas para resolver problemas específicos com sucesso.

O código fornece informações sobre a criação de instâncias de vagões por meio da alocação dinâmica de memória, bem como o gerenciamento dessa alocação quando vagões são adicionados ou removidos da composição. Esse conhecimento é essencial para a compreensão da manipulação de memória em linguagens de programação. No desenvolvimento de software, a inclusão da interação do usuário é um aspecto crucial. Ao incorporar a entrada do usuário, o programa se torna interativo, permitindo que os usuários insiram valores e selecionem ações a serem executadas. Tal operação é

fundamental para a criação de aplicativos fáceis de usar e para o aprimoramento do conjunto de habilidades como desenvolvedor de software.

Por fim, o código exibe as etapas necessárias para a execução de operações fundamentais de lista, incluindo, mas não se limitando à adição ou remoção de elementos, busca por itens específicos e determinação da contagem de carros presentes na composição. Tais operações são críticas para a manipulação de listas e diversas outras estruturas de dados.

Relatório final

O desenvolvimento do código para representar uma composição de vagões e trens em Python proporcionou diversas aprendizagens significativas. O uso de classes e objetos permitiu a modelagem eficaz de vagões e composições, enquanto a implementação de uma lista circular demonstrou como estruturas de dados complexas podem resolver problemas de forma convincente. A alocação dinâmica de memória, interatividade com o usuário e manipulação de listas foram habilidades cruciais desenvolvidas. A organização do código em classes e funções promoveu a legibilidade e manutenção, com comentários detalhados para esclarecimento. No geral, esse projeto exemplifica a aplicação prática de conceitos de programação em Python e serve como base valiosa para projetos futuros.

Referências

ALGORITMOS EM PYTHON. Listas. 201-. Disponível em: <https://algoritmosempython.com.br/cursos/algoritmos-python/estruturas-dados/listas-encadeadas/>. Acesso em: 11 set. 2023.

EDUCATION WIKI. Ponteiros em Python - Tipo de ponteiros - Operaçoes aritimeticas. 20-. Disponível em: <https://pt.education-wiki.com/7701009-pointers-in-python>. Acesso em: 13 set. 2023.

JR, Valdir Stumm. FILAS E PILHAS EM PYTHON. 25 set. 2022. Disponível em: <https://pythonhelp.wordpress.com/2012/09/25/filas-e-pilhas-em-python/>. Acesso em: 19 ago. 2023.

JULIANO, Juliana. RECURSÃO EM PYTHON DE UM JEITO FÁCIL COM EXEMPLOS. 15 jun. 2020. Disponível em: <https://academiahopper.com.br/recursao-python/>. Acesso em: 28 ago. 2023.

MARCILIO. Alocação Dinâmica – Listas Ligadas. 14 out. 2020. Universidade de São Paulo, pág 1 a 24. (e-book). Acesso em: 11 set. 2023.

SANTOS, Jean Carlo. Big data e gerenciamento de memória em Python. 201-. Disponível em: <http://paginapessoal.utfpr.edu.br/jeansantos/artigos/big-data-e-gerenciamento-de-memoria-em-python>. Acesso em: 19 ago. 2023.