Algoritmos de Inserção em Estruturas de Dados para Maratonas de Programação em Python

Este documento apresenta algoritmos de inserção em diversas estruturas de dados, fundamentais para maratonas de programação.

1. Inserção em Listas (Arrays)

1.1 Inserção em Lista não ordenada

python

```python
def inserir_lista(lista, elemento, posicao=None):
    """
    Insere um elemento em uma lista na posição especificada.
    Se a posição não for especificada, insere no final.

    Complexidade de tempo:
    - Melhor caso (inserção no final): O(1)
    - Pior caso (inserção no início): O(n)

    Complexidade de espaço: O(1)
    """
    if posicao is None:
        lista.append(elemento)  # Inserção no final - O(1)
    else:
        lista.insert(posicao, elemento)  # Pode ser O(n) se precisa deslocar elementos

    return lista

# Exemplo de uso
numeros = [1, 2, 3, 5, 6]
inserir_lista(numeros, 4, 3)  # Insere 4 na posição 3
print(f"Lista após inserção: {numeros}")

inserir_lista(numeros, 7)  # Insere 7 no final
print(f"Lista após inserção no final: {numeros}")
```

1.2 Inserção em Lista ordenada (mantendo a ordem)

```
python
```

```python
def inserir_ordenado(lista, elemento):
    """
    Insere um elemento em uma lista ordenada mantendo a ordem.

    Complexidade de tempo:
    - Busca: O(log n) usando busca binária
    - Inserção: O(n) no pior caso (deslocamento)
    - Total: O(n)

    Complexidade de espaço: O(1)
    """
    # Busca binária para encontrar a posição de inserção
    esquerda, direita = 0, len(lista) - 1
    posicao_insercao = 0

    while esquerda <= direita:
        meio = (esquerda + direita) // 2

        if lista[meio] < elemento:
            esquerda = meio + 1
            posicao_insercao = esquerda
        else:
            direita = meio - 1

    # Insere o elemento na posição correta
    lista.insert(posicao_insercao, elemento)
    return lista

# Usando o módulo bisect (mais eficiente)
import bisect

def inserir_ordenado_bisect(lista, elemento):
    """
    Insere um elemento em uma lista ordenada usando o módulo bisect.

    Complexidade de tempo: O(n)
    Complexidade de espaço: O(1)
    """
    bisect.insort(lista, elemento)
    return lista

# Exemplo de uso
numeros_ordenados = [1, 3, 5, 7, 9]
```

```
inserir_ordenado(numeros_ordenados, 4)
print(f"Lista ordenada após inserção: {numeros_ordenados}")

numeros_ordenados = [1, 3, 5, 7, 9]
inserir_ordenado_bisect(numeros_ordenados, 6)
print(f"Lista ordenada após inserção com bisect: {numeros_ordenados}")
```

2. Inserção em Listas Encadeadas (Linked Lists)

2.1 Implementação básica de Lista Encadeada

```
python
```

```python
class No:
    """Nó de uma lista encadeada."""
    def __init__(self, valor):
        self.valor = valor
        self.proximo = None

class ListaEncadeada:
    """Implementação de uma lista encadeada simples."""
    def __init__(self):
        self.cabeca = None

    def inserir_inicio(self, valor):
        """
        Insere um elemento no início da lista.

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        novo_no = No(valor)
        novo_no.proximo = self.cabeca
        self.cabeca = novo_no

    def inserir_final(self, valor):
        """
        Insere um elemento no final da lista.

        Complexidade de tempo: O(n)
        Complexidade de espaço: O(1)
        """
        novo_no = No(valor)

        # Se a lista estiver vazia
        if self.cabeca is None:
            self.cabeca = novo_no
            return

        # Percorre até o último nó
        atual = self.cabeca
        while atual.proximo:
            atual = atual.proximo

        atual.proximo = novo_no
```

```
def inserir_apos(self, no_anterior, valor):
    """
    Insere um elemento após um nó específico.

    Complexidade de tempo: O(1)
    Complexidade de espaço: O(1)
    """
    if no_anterior is None:
        print("O nó anterior não pode ser None")
        return

    novo_no = No(valor)
    novo_no.proximo = no_anterior.proximo
    no_anterior.proximo = novo_no

def inserir_posicao(self, valor, posicao):
    """
    Insere um elemento em uma posição específica.

    Complexidade de tempo: O(n)
    Complexidade de espaço: O(1)
    """
    novo_no = No(valor)

    # Caso especial: inserção no início
    if posicao == 0:
        self.inserir_inicio(valor)
        return

    # Encontra o nó anterior à posição desejada
    atual = self.cabeca
    for i in range(posicao - 1):
        if atual is None:
            print("Posição fora dos limites")
            return
        atual = atual.proximo

    # Verifica se chegou ao fim da lista
    if atual is None:
        print("Posição fora dos limites")
        return

    # Insere o novo nó após o atual
    novo_no.proximo = atual.proximo
```

```
            atual.proximo = novo_no

    def imprimir(self):
        """Imprime todos os elementos da lista."""
        atual = self.cabeca
        elementos = []
        while atual:
            elementos.append(str(atual.valor))
            atual = atual.proximo
        print(" -> ".join(elementos))

# Exemplo de uso
lista = ListaEncadeada()
lista.inserir_inicio(3)
lista.inserir_inicio(2)
lista.inserir_inicio(1)
lista.inserir_final(4)
lista.inserir_posicao(2.5, 2)  # Insere 2.5 na posição 2
lista.imprimir()
```

2.2 Inserção em Lista Duplamente Encadeada

```
python
```

```python
class NoDuplo:
    """Nó de uma lista duplamente encadeada."""
    def __init__(self, valor):
        self.valor = valor
        self.anterior = None
        self.proximo = None

class ListaDuplamenteEncadeada:
    """Implementação de uma lista duplamente encadeada."""
    def __init__(self):
        self.cabeca = None
        self.cauda = None

    def inserir_inicio(self, valor):
        """
        Insere um elemento no início da lista.

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        novo_no = NoDuplo(valor)

        # Se a lista estiver vazia
        if self.cabeca is None:
            self.cabeca = novo_no
            self.cauda = novo_no
            return

        novo_no.proximo = self.cabeca
        self.cabeca.anterior = novo_no
        self.cabeca = novo_no

    def inserir_final(self, valor):
        """
        Insere um elemento no final da lista.

        Complexidade de tempo: O(1) com ponteiro para a cauda
        Complexidade de espaço: O(1)
        """
        novo_no = NoDuplo(valor)

        # Se a lista estiver vazia
        if self.cabeca is None:
```

```
            self.cabeca = novo_no
            self.cauda = novo_no
            return

        # Insere após a cauda atual
        self.cauda.proximo = novo_no
        novo_no.anterior = self.cauda
        self.cauda = novo_no

    def inserir_apos(self, no_anterior, valor):
        """
        Insere um elemento após um nó específico.

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        if no_anterior is None:
            print("O nó anterior não pode ser None")
            return

        novo_no = NoDuplo(valor)

        # Se inserindo no final
        if no_anterior == self.cauda:
            self.inserir_final(valor)
            return

        # Atualiza ponteiros
        novo_no.proximo = no_anterior.proximo
        no_anterior.proximo.anterior = novo_no
        novo_no.anterior = no_anterior
        no_anterior.proximo = novo_no

    def inserir_antes(self, no_posterior, valor):
        """
        Insere um elemento antes de um nó específico.

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        if no_posterior is None:
            print("O nó posterior não pode ser None")
            return
```

```
        # Se inserindo no início
        if no_posterior == self.cabeca:
            self.inserir_inicio(valor)
            return

        # Insere antes do no_posterior
        self.inserir_apos(no_posterior.anterior, valor)

    def imprimir(self):
        """Imprime todos os elementos da lista."""
        atual = self.cabeca
        elementos = []
        while atual:
            elementos.append(str(atual.valor))
            atual = atual.proximo
        print(" <-> ".join(elementos))

# Exemplo de uso
lista_dupla = ListaDuplamenteEncadeada()
lista_dupla.inserir_inicio(2)
lista_dupla.inserir_inicio(1)
lista_dupla.inserir_final(4)
lista_dupla.inserir_final(5)
lista_dupla.inserir_apos(lista_dupla.cabeca.proximo, 3)  # Insere 3 após o 2
lista_dupla.imprimir()
```

3. Inserção em Pilhas (Stacks)

```
python
```

```python
class Pilha:
    """Implementação de uma pilha usando lista."""
    def __init__(self):
        self.itens = []

    def esta_vazia(self):
        """Verifica se a pilha está vazia."""
        return len(self.itens) == 0

    def empilhar(self, item):
        """
        Insere um item no topo da pilha (push).

        Complexidade de tempo: O(1) amortizado
        Complexidade de espaço: O(1)
        """
        self.itens.append(item)

    def desempilhar(self):
        """
        Remove e retorna o item do topo da pilha (pop).

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        if self.esta_vazia():
            raise Exception("Pilha vazia")
        return self.itens.pop()

    def topo(self):
        """Retorna o item no topo sem removê-lo."""
        if self.esta_vazia():
            raise Exception("Pilha vazia")
        return self.itens[-1]

    def tamanho(self):
        """Retorna o número de itens na pilha."""
        return len(self.itens)

# Exemplo de uso
pilha = Pilha()
pilha.empilhar(1)
pilha.empilhar(2)
```

```
pilha.empilhar(3)
print(f"Topo da pilha: {pilha.topo()}")
print(f"Item desempilhado: {pilha.desempilhar()}")
print(f"Novo topo: {pilha.topo()}")
```

4. Inserção em Filas (Queues)

```
python
```

```python
from collections import deque

class Fila:
    """Implementação de uma fila usando deque."""
    def __init__(self):
        self.itens = deque()

    def esta_vazia(self):
        """Verifica se a fila está vazia."""
        return len(self.itens) == 0

    def enfileirar(self, item):
        """
        Insere um item no final da fila (enqueue).

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        self.itens.append(item)

    def desenfileirar(self):
        """
        Remove e retorna o item da frente da fila (dequeue).

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        if self.esta_vazia():
            raise Exception("Fila vazia")
        return self.itens.popleft()

    def frente(self):
        """Retorna o item da frente sem removê-lo."""
        if self.esta_vazia():
            raise Exception("Fila vazia")
        return self.itens[0]

    def tamanho(self):
        """Retorna o número de itens na fila."""
        return len(self.itens)

# Exemplo de uso
fila = Fila()
```

```
fila.enfileirar(1)
fila.enfileirar(2)
fila.enfileirar(3)
print(f"Frente da fila: {fila.frente()}")
print(f"Item desenfileirado: {fila.desenfileirar()}")
print(f"Nova frente: {fila.frente()}")
```

5. Inserção em Árvores Binárias de Busca (BST)

```
python
```

```python
class NoArvore:
    """Nó de uma árvore binária de busca."""
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita = None

class ArvoreBinariaBusca:
    """Implementação de uma árvore binária de busca."""
    def __init__(self):
        self.raiz = None

    def inserir(self, valor):
        """
        Insere um valor na árvore.

        Complexidade de tempo:
        - Melhor caso (árvore balanceada): O(log n)
        - Pior caso (árvore degenerada): O(n)

        Complexidade de espaço:
        - O(h) onde h é a altura da árvore, devido às chamadas recursivas
        """
        self.raiz = self._inserir_recursivo(self.raiz, valor)

    def _inserir_recursivo(self, no, valor):
        """Função auxiliar para inserção recursiva."""
        # Se o nó for None, cria um novo nó
        if no is None:
            return NoArvore(valor)

        # Insere na subárvore apropriada
        if valor < no.valor:
            no.esquerda = self._inserir_recursivo(no.esquerda, valor)
        elif valor > no.valor:
            no.direita = self._inserir_recursivo(no.direita, valor)

        # Retorna o nó atualizado
        return no

    def inserir_iterativo(self, valor):
        """
        Insere um valor na árvore de forma iterativa.
```

```
        Complexidade de tempo: O(h) onde h é a altura da árvore
        Complexidade de espaço: O(1)
        """
        novo_no = NoArvore(valor)

        # Se a árvore estiver vazia
        if self.raiz is None:
            self.raiz = novo_no
            return

        atual = self.raiz
        pai = None

        while atual:
            pai = atual
            if valor < atual.valor:
                atual = atual.esquerda
            elif valor > atual.valor:
                atual = atual.direita
            else:
                # Valor já existe na árvore
                return

        # Insere o novo nó como filho do pai
        if valor < pai.valor:
            pai.esquerda = novo_no
        else:
            pai.direita = novo_no

    def busca(self, valor):
        """
        Busca um valor na árvore.

        Complexidade de tempo: O(h) onde h é a altura da árvore
        Complexidade de espaço: O(1) para versão iterativa, O(h) para recursiva
        """
        atual = self.raiz

        while atual:
            if valor == atual.valor:
                return True
            elif valor < atual.valor:
                atual = atual.esquerda
```

```
            else:
                atual = atual.direita

        return False

    def percurso_em_ordem(self):
        """Retorna uma lista com os valores em ordem crescente."""
        resultado = []
        self._em_ordem_recursivo(self.raiz, resultado)
        return resultado

    def _em_ordem_recursivo(self, no, resultado):
        """Função auxiliar para percurso em ordem."""
        if no:
            self._em_ordem_recursivo(no.esquerda, resultado)
            resultado.append(no.valor)
            self._em_ordem_recursivo(no.direita, resultado)

# Exemplo de uso
bst = ArvoreBinariaBusca()
valores = [50, 30, 70, 20, 40, 60, 80]

for valor in valores:
    bst.inserir(valor)

print(f"Árvore em ordem: {bst.percurso_em_ordem()}")
print(f"Valor 40 está na árvore? {bst.busca(40)}")
print(f"Valor 55 está na árvore? {bst.busca(55)}")
```

6. Inserção em Heaps (Montes)

```
python
```

```python
class MinHeap:
    """Implementação de um heap mínimo."""
    def __init__(self):
        self.heap = []

    def pai(self, i):
        """Retorna o índice do pai do nó i."""
        return (i - 1) // 2

    def esquerda(self, i):
        """Retorna o índice do filho esquerdo do nó i."""
        return 2 * i + 1

    def direita(self, i):
        """Retorna o índice do filho direito do nó i."""
        return 2 * i + 2

    def troca(self, i, j):
        """Troca os elementos nas posições i e j."""
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def subir(self, i):
        """
        Restaura a propriedade de heap movendo o elemento para cima.

        Complexidade de tempo: O(log n)
        Complexidade de espaço: O(1)
        """
        pai = self.pai(i)

        if i > 0 and self.heap[pai] > self.heap[i]:
            self.troca(i, pai)
            self.subir(pai)

    def descer(self, i):
        """
        Restaura a propriedade de heap movendo o elemento para baixo.

        Complexidade de tempo: O(log n)
        Complexidade de espaço: O(1)
        """
        menor = i
        esq = self.esquerda(i)
```

```
        dir = self.direita(i)

        if esq < len(self.heap) and self.heap[esq] < self.heap[menor]:
            menor = esq

        if dir < len(self.heap) and self.heap[dir] < self.heap[menor]:
            menor = dir

        if menor != i:
            self.troca(i, menor)
            self.descer(menor)

    def inserir(self, valor):
        """
        Insere um novo valor no heap.

        Complexidade de tempo: O(log n)
        Complexidade de espaço: O(1)
        """
        self.heap.append(valor)
        self.subir(len(self.heap) - 1)

    def extrair_min(self):
        """
        Remove e retorna o menor elemento (a raiz).

        Complexidade de tempo: O(log n)
        Complexidade de espaço: O(1)
        """
        if not self.heap:
            return None

        minimo = self.heap[0]
        ultimo = self.heap.pop()

        if self.heap:
            self.heap[0] = ultimo
            self.descer(0)

        return minimo

    def tamanho(self):
        """Retorna o número de elementos no heap."""
        return len(self.heap)
```

```
    def esta_vazio(self):
        """Verifica se o heap está vazio."""
        return len(self.heap) == 0

    def minimo(self):
        """Retorna o menor elemento sem removê-lo."""
        if not self.heap:
            return None
        return self.heap[0]

# Usando heapq (biblioteca padrão)
import heapq

def usar_heapq():
    """Exemplo usando o módulo heapq para operações de heap."""
    # Criar heap a partir de uma lista
    lista = [4, 2, 8, 1, 5, 3]
    heapq.heapify(lista)  # Transforma a lista em um heap
    print(f"Heap após heapify: {lista}")

    # Inserir elemento
    heapq.heappush(lista, 0)
    print(f"Heap após inserir 0: {lista}")

    # Extrair o menor elemento
    menor = heapq.heappop(lista)
    print(f"Menor elemento: {menor}")
    print(f"Heap após extração: {lista}")

    # Inserir e extrair em uma operação
    heapq.heappushpop(lista, 6)  # Insere 6 e extrai o menor
    print(f"Heap após heappushpop: {lista}")

    # Retorna os n menores elementos
    tres_menores = heapq.nsmallest(3, lista)
    print(f"Os três menores elementos: {tres_menores}")

# Exemplo de uso do MinHeap
heap = MinHeap()
valores = [4, 2, 8, 1, 5, 3]

for valor in valores:
    heap.inserir(valor)
```

```
print(f"Mínimo: {heap.minimo()}")
print(f"Extraindo elementos em ordem:")
while not heap.esta_vazio():
    print(heap.extrair_min(), end=" ")
print()

# Exemplo usando heapq
usar_heapq()
```

7. Inserção em Grafos

```
python
```

```python
class Grafo:
    """Implementação de um grafo usando lista de adjacência."""
    def __init__(self, direcionado=False):
        self.lista_adj = {}
        self.direcionado = direcionado

    def adicionar_vertice(self, vertice):
        """
        Adiciona um vértice ao grafo.

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        if vertice not in self.lista_adj:
            self.lista_adj[vertice] = []

    def adicionar_aresta(self, origem, destino, peso=1):
        """
        Adiciona uma aresta ao grafo.

        Complexidade de tempo: O(1)
        Complexidade de espaço: O(1)
        """
        # Adiciona vértices se não existirem
        self.adicionar_vertice(origem)
        self.adicionar_vertice(destino)

        # Adiciona a aresta
        self.lista_adj[origem].append((destino, peso))

        # Se o grafo for não direcionado, adiciona a aresta na direção oposta
        if not self.direcionado:
            self.lista_adj[destino].append((origem, peso))

    def vertices(self):
        """Retorna a lista de vértices."""
        return list(self.lista_adj.keys())

    def arestas(self):
        """Retorna a lista de arestas."""
        arestas = []
        for origem in self.lista_adj:
            for destino, peso in self.lista_adj[origem]:
```

```
                # Evita duplicatas em grafos não direcionados
                if self.direcionado or origem <= destino:
                    arestas.append((origem, destino, peso))
        return arestas

    def adjacentes(self, vertice):
        """Retorna a lista de vértices adjacentes a um vértice."""
        return [destino for destino, _ in self.lista_adj.get(vertice, [])]

    def imprimir(self):
        """Imprime a representação do grafo."""
        for vertice in self.lista_adj:
            adjacentes = [f"{destino}(peso:{peso})" for destino, peso in self.lista_ad
            print(f"{vertice} -> {', '.join(adjacentes)}")

# Exemplo de uso
grafo = Grafo(direcionado=True)
grafo.adicionar_aresta('A', 'B', 2)
grafo.adicionar_aresta('A', 'C', 3)
grafo.adicionar_aresta('B', 'C', 1)
grafo.adicionar_aresta('B', 'D', 5)
grafo.adicionar_aresta('C', 'D', 2)

print("Grafo direcionado:")
grafo.imprimir()

# Grafo não direcionado
grafo_nd = Grafo(direcionado=False)
grafo_nd.adicionar_aresta('A', 'B', 2)
grafo_nd.adicionar_aresta('A', 'C', 3)
grafo_nd.adicionar_aresta('B', 'C', 1)

print("\nGrafo não direcionado:")
grafo_nd.imprimir()
```

8. Inserção em Tabelas Hash (Dicionários)

```
python
```

```python
class TabelaHash:
    """Implementação simples de uma tabela hash com endereçamento aberto."""
    def __init__(self, tamanho=10):
        # Inicializa com valores None
        self.tamanho = tamanho
        self.tabela = [None] * tamanho
        self.elemento_count = 0

    def hash_funcao(self, chave):
        """Função de hash simples para strings e números."""
        if isinstance(chave, str):
            # Somar os valores ASCII de cada caractere
            return sum(ord(c) for c in chave) % self.tamanho
        elif isinstance(chave, (int, float)):
            return int(chave) % self.tamanho
        else:
            # Para outros tipos, usar o hash padrão do Python
            return hash(chave) % self.tamanho

    def inserir(self, chave, valor):
        """
        Insere um par chave-valor na tabela hash.
        Usa sondagem linear para resolução de colisões.

        Complexidade de tempo:
        - Melhor caso: O(1)
        - Pior caso: O(n) com muitas colisões

        Complexidade de espaço: O(1)
        """
        # Verificar se a tabela está cheia
        if self.elemento_count >= self.tamanho * 0.7:  # Fator de carga de 70%
            self._redimensionar()

        indice = self.hash_funcao(chave)
        posicao_inicial = indice

        # Sondagem linear
        while self.tabela[indice] is not None:
            # Atualiza o valor se a chave já existe
            if self.tabela[indice][0] == chave:
                self.tabela[indice] = (chave, valor)
                return
```

```
            # Move para a próxima posição
            indice = (indice + 1) % self.tamanho

            # Se já verificou todas as posições, a tabela está cheia
            if indice == posicao_inicial:
                return

        # Insere o par chave-valor
        self.tabela[indice] = (chave, valor)
        self.elemento_count += 1

    def _redimensionar(self):
        """Duplica o tamanho da tabela hash."""
        tabela_antiga = self.tabela
        self.tamanho *= 2
        self.tabela = [None] * self.tamanho
        self.elemento_count = 0

        # Reinsere todos os elementos
        for item in tabela_antiga:
            if item is not None:
                self.inserir(item[0], item[1])

    def buscar(self, chave):
        """
        Busca um valor pela chave.

        Complexidade de tempo:
        - Melhor caso: O(1)
        - Pior caso: O(n) com muitas colisões
        """
        indice = self.hash_funcao(chave)
        posicao_inicial = indice

        while self.
```