Algoritmos e Estruturas de Dados

Listas Lineares e Análise de Complexidade

Sumário

---

1. Análise de Complexidade

1.1 Notação Big-O

A notação Big-O descreve o comportamento assintótico de um algoritmo, ou seja, como o tempo de execução ou espaço de memória cresce em função do tamanho da entrada (n).

Principais Complexidades:

- **O(1) - Constante:** Tempo de execução não depende do tamanho da entrada
  - Exemplo: acesso a elemento em array por índice
- **O(log n) - Logarítmica:** Tempo cresce logaritmicamente
  - Exemplo: busca binária em lista ordenada
- **O(n) - Linear:** Tempo cresce proporcionalmente ao tamanho da entrada
  - Exemplo: busca sequencial em lista
- **O(n log n) - Linearítmica:** Comum em algoritmos de ordenação eficientes
  - Exemplo: MergeSort, QuickSort (caso médio)
- **O(n^2) - Quadrática:** Tempo cresce com o quadrado do tamanho
  - Exemplo: BubbleSort, SelectionSort
- **O(2^n) - Exponencial:** Tempo dobra a cada elemento adicionado
  - Exemplo: algoritmos de força bruta

1.2 Análise de Melhor, Pior e Caso Médio

- **Melhor Caso:** Situação mais favorável para o algoritmo
- **Pior Caso:** Situação menos favorável (geralmente mais importante)
- **Caso Médio:** Comportamento esperado em situações típicas

1.3 Complexidade de Espaço

Além do tempo, é importante analisar o espaço de memória utilizado:

- **Espaço Auxiliar:** Memória extra além da entrada
- **Espaço In-Place:** Algoritmos que usam O(1) de espaço auxiliar

---

2. Listas Lineares - Conceitos Fundamentais

2.1 Definição

Uma lista linear é uma estrutura de dados que armazena elementos em sequência, onde cada elemento (exceto o primeiro) tem exatamente um predecessor e cada elemento (exceto o último) tem exatamente um sucessor.

Propriedades:

- Ordem: os elementos mantêm uma ordem específica
- Homogeneidade: todos os elementos são do mesmo tipo
- Tamanho variável: pode crescer ou diminuir dinamicamente

2.2 Operações Fundamentais

Toda implementação de lista deve suportar:

1. **Inserção:** adicionar elemento em posição específica
2. **Remoção:** remover elemento de posição específica
3. **Busca:** localizar elemento por valor ou posição
4. **Acesso:** recuperar elemento em posição específica
5. **Percurso:** visitar todos os elementos

2.3 Tipos de Implementação

As duas principais formas de implementar listas são:

- **Alocação Sequencial (Estática):** usa arrays contíguos na memória
- **Alocação Encadeada (Dinâmica):** usa nós conectados por ponteiros

---

3. Alocação Sequencial (Listas Estáticas)

3.1 Estrutura de Dados

```
Estrutura ListaSequencial:
    elementos: array[MAX] de TipoElemento
    tamanho: inteiro
```

Características:

- Elementos armazenados em posições consecutivas de memória
- Acesso direto por índice: O(1)
- Tamanho máximo predefinido (ou realocação dinâmica)

3.2 Operações e Complexidade

3.2.1 Inserção

Inserção no Final:

```
Inserir_Final(L, elemento):
    se L.tamanho = MAX então
        retornar ERRO (lista cheia)
    L.elementos[L.tamanho] ← elemento
    L.tamanho ← L.tamanho + 1
```

- **Complexidade:** O(1)

Inserção em Posição Arbitrária:

```
Inserir_Posicao(L, elemento, pos):
    se L.tamanho = MAX então
        retornar ERRO
    para i de L.tamanho-1 até pos faça
        L.elementos[i+1] ← L.elementos[i]
    L.elementos[pos] ← elemento
    L.tamanho ← L.tamanho + 1
```

- **Melhor Caso:** O(1) - inserção no final
- **Pior Caso:** O(n) - inserção no início (requer deslocar todos elementos)
- **Caso Médio:** O(n)

3.2.2 Remoção

Remoção de Posição Arbitrária:

```
Remover_Posicao(L, pos):
    se pos ≥ L.tamanho então
        retornar ERRO
    para i de pos até L.tamanho-2 faça
        L.elementos[i] ← L.elementos[i+1]
    L.tamanho ← L.tamanho - 1
```

- **Melhor Caso:** O(1) - remoção do final
- **Pior Caso:** O(n) - remoção do início
- **Caso Médio:** O(n)

3.2.3 Busca

Busca Sequencial:

```
Buscar(L, valor):
    para i de 0 até L.tamanho-1 faça
        se L.elementos[i] = valor então
            retornar i
    retornar NÃO_ENCONTRADO
```

- **Melhor Caso:** O(1) - elemento está na primeira posição
- **Pior Caso:** O(n) - elemento não existe ou está no final
- **Caso Médio:** O(n)

Busca Binária (lista ordenada):

```
Busca_Binaria(L, valor):
    inicio ← 0
    fim ← L.tamanho - 1
    enquanto inicio ≤ fim faça
        meio ← (inicio + fim) / 2
        se L.elementos[meio] = valor então
            retornar meio
        senão se L.elementos[meio] < valor então
            inicio ← meio + 1
        senão
            fim ← meio - 1
    retornar NÃO_ENCONTRADO
```

- **Complexidade:** O(log n)

3.2.4 Acesso

Acesso por Índice:

```
Acessar(L, pos):
    se pos ≥ L.tamanho então
        retornar ERRO
    retornar L.elementos[pos]
```

- **Complexidade:** O(1)

3.3 Vantagens e Desvantagens

Vantagens:

- Acesso direto e rápido por índice: O(1)
- Simplicidade de implementação
- Boa localidade de referência (cache-friendly)
- Menor overhead de memória (sem ponteiros)

Desvantagens:

- Tamanho máximo fixo (ou custo de realocação)
- Inserção/remoção ineficiente: O(n)
- Desperdício de memória se tamanho máximo não for utilizado
- Realocação custosa quando capacidade é excedida

---

4. Alocação Encadeada (Listas Dinâmicas)

4.1 Estrutura de Dados

4.1.1 Lista Simplesmente Encadeada

```
Estrutura Nó:
    dado: TipoElemento
    proximo: ponteiro para Nó

Estrutura ListaEncadeada:
    inicio: ponteiro para Nó
    tamanho: inteiro
```

Características:

- Cada nó contém dado e ponteiro para próximo nó
- Último nó aponta para NULL
- Acesso sequencial obrigatório

4.1.2 Lista Duplamente Encadeada

```
Estrutura NóDuplo:
    dado: TipoElemento
    proximo: ponteiro para NóDuplo
    anterior: ponteiro para NóDuplo

Estrutura ListaDupla:
    inicio: ponteiro para NóDuplo
    fim: ponteiro para NóDuplo
    tamanho: inteiro
```

Características:

- Navegação bidirecional
- Facilitação de certas operações (remoção, por exemplo)
- Maior uso de memória (dois ponteiros por nó)

4.2 Operações e Complexidade

4.2.1 Inserção (Lista Simples)

Inserção no Início:

```
Inserir_Inicio(L, elemento):
    novo ← alocar Nó
    novo.dado ← elemento
    novo.proximo ← L.inicio
    L.inicio ← novo
    L.tamanho ← L.tamanho + 1
```

- **Complexidade:** O(1)

Inserção no Final:

```
Inserir_Final(L, elemento):
    novo ← alocar Nó
    novo.dado ← elemento
    novo.proximo ← NULL

    se L.inicio = NULL então
        L.inicio ← novo
    senão
        atual ← L.inicio
        enquanto atual.proximo ≠ NULL faça
            atual ← atual.proximo
        atual.proximo ← novo

    L.tamanho ← L.tamanho + 1
```

- **Complexidade:** O(n) - sem ponteiro para fim
- **Com ponteiro para fim:** O(1)

Inserção em Posição Arbitrária:

```
Inserir_Posicao(L, elemento, pos):
    se pos = 0 então
        Inserir_Inicio(L, elemento)
        retornar

    novo ← alocar Nó
    novo.dado ← elemento

    atual ← L.inicio
    para i de 0 até pos-2 faça
        se atual = NULL então
            retornar ERRO
        atual ← atual.proximo

    novo.proximo ← atual.proximo
    atual.proximo ← novo
    L.tamanho ← L.tamanho + 1
```

- **Complexidade:** O(n) - necessário percorrer até a posição

4.2.2 Remoção (Lista Simples)

Remoção do Início:

```
Remover_Inicio(L):
    se L.inicio = NULL então
        retornar ERRO

    temp ← L.inicio
    L.inicio ← L.inicio.proximo
    liberar temp
    L.tamanho ← L.tamanho - 1
```

- **Complexidade:** O(1)

Remoção por Valor:

```
Remover_Valor(L, valor):
    se L.inicio = NULL então
        retornar ERRO

    se L.inicio.dado = valor então
        Remover_Inicio(L)
        retornar

    atual ← L.inicio
    enquanto atual.proximo ≠ NULL faça
        se atual.proximo.dado = valor então
            temp ← atual.proximo
            atual.proximo ← temp.proximo
            liberar temp
            L.tamanho ← L.tamanho - 1
            retornar
        atual ← atual.proximo

    retornar NÃO_ENCONTRADO
```

- **Melhor Caso:** O(1) - elemento no início
- **Pior Caso:** O(n) - elemento no final ou não existe
- **Caso Médio:** O(n)

4.2.3 Busca

Busca Sequencial:

```
Buscar(L, valor):
    atual ← L.inicio
    pos ← 0
    enquanto atual ≠ NULL faça
        se atual.dado = valor então
            retornar pos
        atual ← atual.proximo
        pos ← pos + 1
    retornar NÃO_ENCONTRADO
```

- **Complexidade:** O(n)
- Busca binária não é eficiente em listas encadeadas

4.2.4 Acesso

Acesso por Posição:

```
Acessar(L, pos):
    atual ← L.inicio
    para i de 0 até pos-1 faça
        se atual = NULL então
            retornar ERRO
        atual ← atual.proximo
    retornar atual.dado
```

- **Complexidade:** O(n)

4.3 Lista Duplamente Encadeada - Diferenças Importantes

Inserção no Final - O(1)

```
Inserir_Final(L, elemento):
    novo ← alocar NóDuplo
    novo.dado ← elemento
    novo.proximo ← NULL
    novo.anterior ← L.fim

    se L.fim ≠ NULL então
        L.fim.proximo ← novo
    senão
        L.inicio ← novo

    L.fim ← novo
    L.tamanho ← L.tamanho + 1
```

- **Complexidade:** O(1) - devido ao ponteiro para fim

Remoção de Nó Específico - O(1)

```
Remover_No(L, no):
    se no.anterior ≠ NULL então
        no.anterior.proximo ← no.proximo
    senão
        L.inicio ← no.proximo

    se no.proximo ≠ NULL então
        no.proximo.anterior ← no.anterior
    senão
        L.fim ← no.anterior

    liberar no
    L.tamanho ← L.tamanho - 1
```

- **Complexidade:** O(1) - se já temos o nó

4.4 Lista Circular

Uma variação onde o último nó aponta para o primeiro (simples) ou ambos se conectam (dupla).

Características:

- Útil para implementar buffers circulares
- Não há "fim" definido
- Cuidado com loops infinitos ao percorrer

4.5 Vantagens e Desvantagens

Vantagens:

- Tamanho dinâmico (cresce conforme necessário)
- Inserção/remoção eficiente no início: O(1)
- Sem necessidade de realocação
- Não desperdiça memória

Desvantagens:

- Acesso sequencial obrigatório: O(n)
- Maior overhead de memória (ponteiros)
- Pior localidade de referência (cache)
- Mais complexa de implementar
- Possibilidade de vazamento de memória

---

5. Comparação entre Implementações

5.1 Tabela Comparativa de Complexidade

Operação	Sequencial	Encadeada Simples	Encadeada Dupla
Acesso por índice	O(1)	O(n)	O(n)
Busca	O(n)	O(n)	O(n)
Busca (ordenada)	O(log n)	O(n)	O(n)
Inserção início	O(n)	O(1)	O(1)
Inserção fim	O(1)	O(n)*	O(1)
Inserção meio	O(n)	O(n)	O(n)
Remoção início	O(n)	O(1)	O(1)
Remoção fim	O(1)	O(n)	O(1)
Remoção meio	O(n)	O(n)	O(n)

*O(1) se mantiver ponteiro para o fim

5.2 Uso de Memória

Alocação Sequencial:

- Memória = n × tamanho_elemento + overhead_array
- Overhead mínimo
- Possível desperdício se capacidade > tamanho

Alocação Encadeada Simples:

- Memória = n × (tamanho_elemento + tamanho_ponteiro)
- Overhead de 1 ponteiro por elemento

Alocação Encadeada Dupla:

- Memória = n × (tamanho_elemento + 2 × tamanho_ponteiro)
- Overhead de 2 ponteiros por elemento

5.3 Quando Usar Cada Implementação

Use Alocação Sequencial quando:

- Acesso frequente por índice

- Tamanho máximo conhecido e razoável
- Poucas inserções/remoções
- Busca binária necessária
- Memória cache é importante

Use Alocação Encadeada quando:

- Tamanho imprevisível ou muito variável
- Muitas inserções/remoções (especialmente no início)
- Não há necessidade de acesso aleatório
- Economia de memória é crítica (sem desperdício)

---

6. Exercícios e Questões de Prova

6.1 Análise de Complexidade

Q1: Qual a complexidade da seguinte função?

```
função Exemplo(A, n):
    para i de 0 até n-1:
        para j de 0 até n-1:
            se A[i] > A[j]:
                trocar(A[i], A[j])
```

Resposta: $O(n^2)$ - dois loops aninhados que percorrem n elementos cada

Q2: Analise a complexidade:

```
função Busca(L, x):
    meio = tamanho(L) / 2
    se L[meio] = x:
        retornar meio
    para i de 0 até tamanho(L)-1:
        se L[i] = x:
            retornar i
    retornar -1
```

Resposta: O(n) - apesar da verificação inicial, o loop percorre todos elementos

6.2 Listas Sequenciais

Q3: Uma lista sequencial com capacidade 100 contém 50 elementos. Qual o custo (pior caso) para:

- a) Inserir no início? **O(n) = O(50)**
- b) Inserir no final? **O(1)**
- c) Remover posição 25? **O(n) = O(25)**

Q4: Por que busca binária é mais eficiente que busca sequencial em listas ordenadas? **Resposta:** Busca binária elimina metade dos elementos a cada comparação (O(log n)), enquanto busca sequencial verifica elemento por elemento (O(n)).

6.3 Listas Encadeadas

Q5: Qual a vantagem de manter um ponteiro para o último nó em uma lista simplesmente encadeada? **Resposta:** Permite inserção no final em O(1), ao invés de O(n).

Q6: Por que remover um nó específico é O(1) em lista duplamente encadeada, mas requer O(n) em lista simples? **Resposta:** Na lista dupla, temos acesso direto ao nó anterior através do ponteiro, permitindo ajustar os links diretamente. Na lista simples, é necessário percorrer desde o início para encontrar o nó anterior.

Q7: Implemente uma função para inverter uma lista simplesmente encadeada.

```
Inverter(L):
    anterior ← NULL
    atual ← L.inicio

    enquanto atual ≠ NULL faça:
        proximo ← atual.proximo
        atual.proximo ← anterior
        anterior ← atual
        atual ← proximo

    L.inicio ← anterior
```

Complexidade: O(n) tempo, O(1) espaço

6.4 Questões Conceituais

Q8: Compare memória e performance entre lista sequencial de 1000 elementos (50% ocupada) e lista encadeada com 500 elementos.

Resposta:

- Sequencial: desperdiça memória (500 posições vazias), mas acesso O(1)
- Encadeada: usa apenas memória necessária + overhead de ponteiros, acesso O(n)
- Escolha depende do padrão de acesso

Q9: É possível implementar uma pilha (LIFO) eficientemente com ambas as estruturas? **Resposta:** Sim. Em ambas, operações push e pop podem ser O(1) se operarmos em uma extremidade apropriada (topo para sequencial, início para encadeada).

Q10: Qual estrutura é melhor para implementar um deque (fila dupla)? **Resposta:** Lista duplamente encadeada, pois permite inserção/remoção O(1) em ambas extremidades. Lista sequencial requer O(n) para operações no início.

Resumo para Prova

Pontos-Chave para Memorizar

Complexidades Essenciais:

- Acesso sequencial: O(1), encadeada: O(n)
- Inserção/remoção início: sequencial O(n), encadeada O(1)
- Busca: ambas O(n), mas sequencial ordenada permite O(log n)

Trade-offs Fundamentais:

- Sequencial: rápido acesso, lenta modificação
- Encadeada: rápida modificação (extremidades), lento acesso

Memória:

- Sequencial: pode desperdiçar, sem overhead de ponteiros
- Encadeada: usa exato necessário, overhead de ponteiros

Quando Usar:

- Acesso aleatório frequente → Sequencial
- Inserções/remoções frequentes → Encadeada
- Tamanho previsível → Sequencial
- Tamanho imprevisível → Encadeada

Material desenvolvido para preparação de provas de AED Bons estudos!