Estruturas de Dados e Algoritmos de Busca e Ordenação

Ed. v1.0.0

HISTÓRICO DE REVISÕES

VERSÃO	DATA	MODIFICAÇÕES	NOME
v1.0.0	5/11/2014	cap5: Adição Calculadora Polonesa como aplicação de Pilha; Revisão das imagens para impressão;	Eduardo
v0.3.4	1/10/2014	Adição de links no histórico de revisão; Solicitando feedback ao final de cada capítulo; Prefácio: explicando como baixar os códigos do livro;	Eduardo
v0.3.3	6/8/2014	Revisão do Capítulo 5; Utilização de monospace em funções e variáveis; Inclusão de links para os códigos no github; Remoção de referência à Linguagem livre de contexto;	Eduardo
v0.3.2	1/8/2014	Revisão do Capítulo 4; Correção ortográfica;	Eduardo
v0.3.1	31/7/2014	Padronização visual dos Objetivos; Remoção de capítulos não escritos; Inclusão da Ficha Catalográfica;	Eduardo
v0.3.0	26/3/2014	Revisão do Capítulo 3;	Eduardo
v0.2.0	21/3/2014	Revisão do Capítulo 2;	Eduardo
v0.1.0	27/2/2014	Capítulo 1 Revisado.	Alexandre

Sumário

Prefácio

Sumário

BAIXANDO A VERSÃO MAIS NOVA DESTE LIVRO

Acesse https://github.com/edusantana/estruturas-de-dados-livro/releases para verificar se há uma versão mais nova deste livro (versão atual: **v1.0.0**). Você pode consultar o Histórico de revisões, na início do livro, para verificar o que mudou entre uma versão e outra.

Público alvo

O público alvo desse livro são os alunos de Licenciatura em Computação, na modalidade à distância [1]. Ele foi concebido para ser utilizado numa disciplina de *Estrutura de Dados*, no terceiro semestre do curso.

Como você deve estudar cada capítulo

- Leia a visão geral do capítulo
- Estude os conteúdos das seções
- Realize as atividades no final do capítulo
- Verifique se você atingiu os objetivos do capítulo

NA SALA DE AULA DO CURSO

[1] Embora ele tenha sido feito para atender aos alunos da Universidade Federal da Paraíba, o seu uso não se restringe a esta universidade, podendo ser adotado por outras universidades do sistema UAB.

- Tire dúvidas e discuta sobre as atividades do livro com outros integrantes do curso
- Leia materiais complementares eventualmente disponibilizados
- Realize as atividades propostas pelo professor da disciplina

Caixas de diálogo

Nesta seção apresentamos as caixas de diálogo que poderão ser utilizadas durante o texto. Confira os significados delas.

Nota
Esta caixa é utilizada para realizar alguma reflexão.

Dica
Esta caixa é utilizada quando desejamos remeter a materiais complementares.

Importante
Esta caixa é utilizada para chamar atenção sobre algo importante.

Cuidado
Esta caixa é utilizada para alertar sobre algo que exige cautela.

Atenção
Esta caixa é utilizada para alertar sobre algo potencialmente perigoso.

Os significados das caixas são apenas uma referência, podendo ser adaptados conforme as intenções dos autores.

Vídeos

Os vídeos são apresentados da seguinte forma:

Figura 1: Como baixar os códigos fontes: http://youtu.be/Od90rVXJV78

Nota
Na **versão impressa** irá aparecer uma imagem quadriculada. Isto é o qrcode (http://pt.wikipedia.org/wiki/C%C3%B3digo_QR) contendo o link do vídeo. Caso você tenha um celular com acesso a internet poderá acionar um programa de leitura de qrcode para acessar o vídeo.
Na **versão digital** você poderá assistir o vídeo clicando diretamente sobre o link.

Compreendendo as referências

As referências são apresentadas conforme o elemento que está sendo referenciado:

Referências a capítulos
Prefácio [v]

Referências a seções
“Como você deve estudar cada capítulo” [v], “Caixas de diálogo” [vi].

Referências a imagens
Figura 2 [ix]

Nota
Na **versão impressa**, o número que aparece entre chaves “[ ]” corresponde ao número da página onde está o conteúdo referenciado. Na **versão digital** do livro você poderá clicar no link da referência.

Códigos e comandos

Os códigos ou comandos são apresentados com a seguinte formação:

```
echo "Hello Word"
```

No exemplo a seguir, temos outra apresentação de código fonte. Desta vez de um arquivo *helloworld.c*, que se encontra dentro do diretório *livro/capitulos/code/prefacio*. O diretório *prefacio* indica o capítulo onde o código está relacionado.

Código fonte code/prefacio/helloworld.c

helloworld.c

```
/* Hello World program */

#include<stdio.h>   // ❶

main()
{
    printf("Hello World");   // ❷ imprime "Hello Word" na tela.
}
```

Baixando os códigos fontes

Recomendamos duas formas de acessar os códigos fontes contidos neste livro:

Acesso on-line individual
https://github.com/edusantana/estruturas-de-dados-livro/tree/v1.0.0/livro/capitulos/code

Baixando zip contendo os códigos

Versão atual
https://github.com/edusantana/estruturas-de-dados-livro/archive/v1.0.0.zip

Nota
Independente do método utilizado para acessar os arquivos, os códigos fontes estão organizados por capítulos no diretório `livro/capitulos/code`.

Feedback

Você pode contribuir com a atualização e correção deste livro. Ao final de cada capítulo você será convidado a fazê-lo, enviando um feedback como a seguir:

Feedback sobre o capítulo
Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?
Acesse https://github.com/edusantana/estruturas-de-dados-livro/issues/new para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**prefacio**) e a versão do livro (**v1.0.0**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Nota

A seção sobre o feedback, no guia do curso, pode ser acessado em: https://github.com/-edusantana/guia-geral-ead-computacao-ufpb/blob/master/livro/capitulos/livros-contribuicao.adoc.

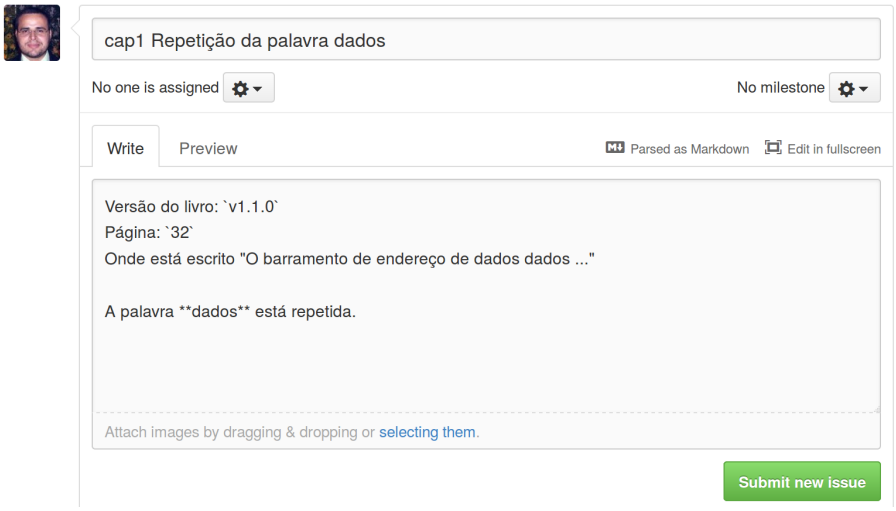


Figura 2: Exemplo de contribuição

Capítulo 1

Introdução

Sumário

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender a importância do uso de estruturas de dados e os conceitos de abstração e encapsulamento
- Conhecer a estrutura básica que será utilizada ao longo do livro para representar um elemento em nossas estruturas de dados
- Compreender o princípio da indução matemática

Computadores podem armazenar e processar grandes volumes de dados. Estruturas de dados formais possibilitam aos programadores gerenciar conceitualmente os relacionamentos entre os dados utilizados por um determinado programa.

Muitas vezes, estruturas de dados específicas nos permitem fazer mais, por exemplo, ordenar mais rápido um grande volume de dados. Noutras vezes, utilizamos certas estruturas de dados para fazermos menos, por exemplo, encontrar um determinado elemento em uma coleção utilizando o menor número possível de comparações.

Estruturas de dados também oferecem garantias sobre a complexidade algorítmica relacionada às suas operações básicas. Portanto, escolher estruturas de dados apropriadas é crucial para o desenvolvimento de qualquer software.

Atenção
A escolha das estruturas de dados a serem utilizadas tem papel fundamental no desenvolvimento de qualquer software. Uma escolha errada pode implicar em perdas de desempenho e em uma maior dificuldade na implementação.

Estruturas de dados representam abstrações de alto nível e podem fornecer operações para lidar com grupos de dados, incluindo formas para adicionar, remover ou localizar elementos. Uma estrutura de dados que fornece operações para manipular seus dados é denominada **tipo abstrato de dados** (muitas vezes abreviado por TAD). Tipos abstratos de dados podem minimizar dependências no código, o que é importante quando é preciso realizar alterações no programa. Tais estruturas abstraem do programa os detalhes de baixo nível, permitindo que o programador concentre-se unicamente em utilizar a estrutura sem precisar conhecer, necessariamente, detalhes sobre sua implementação. Além disso, estruturas de dados que apresentam um conjunto comum de operações de alto nível podem ser substituídas umas pelas outras sem maiores impactos no restante do código de uma aplicação.

Nota
Uma estrutura de dados que fornece operações de alto nível para manipular seus elementos é denominada **tipo abstrato de dados**.

As principais linguagens de programação estão equipadas com um conjunto de tipos de dados, como números inteiros e de ponto flutuante, que nos permitem lidar com dados para os quais o processador do computador oferece suporte nativo. Estes tipos de dados pré-definidos são abstrações do que o processador realmente oferece, uma vez que escondem detalhes sobre sua execução e limitações.

Por exemplo, quanto utilizamos números de ponto flutuante estamos preocupados principalmente com seu valor e com quais operações podem ser aplicadas a eles.

Considere o trecho de código abaixo que calcula o comprimento da hipotenusa de um triângulo com catetos a e b.

Cálculo do comprimento da hipotenusa de um triângulo retângulo de catetos a e b

```
double c = sqrt(a * a + b * b)
```

O código de máquina gerado a partir do código acima utilizaria padrões comuns para computar e acumular o resultado. De fato, estes padrões são tão repetitivos que levaram a criação de linguagens de programação de alto nível justamente para evitar a redundância e para permitir que os programadores possam se preocupar com o que está sendo computado ao invés do como está sendo computado.

Dois conceitos muito úteis e inter-relacionados têm um papel fundamental neste avanço:

Encapsulamento
: Acontece quando um padrão comum de codificação é agrupado em uma unidade com nome único e que pode ser parametrizada, fornecendo um entendimento de mais alto nível sobre este padrão. Por exemplo, a operação de multiplicação requer dois valores fonte e escreve o valor do produto destes dois valores em um determinado destino. A operação é parametrizada por seus dois valores fonte e pelo destino único.

Abstração
: É um mecanismo utilizado para esconder detalhes de implementação dos usuários. Quando multiplicamos dois números não precisamos saber qual a técnica utilizada pelo processador para efetivamente multiplicar os números, precisamos apenas conhecer as propriedades da operação de multiplicação.

Uma linguagem de programação é tanto uma abstração da máquina quanto uma ferramenta para encapsular seus detalhes internos. Quando a linguagem de programação encapsula suficientemente

bem o funcionamento da máquina um programa escrito nesta linguagem pode ser compilado em várias arquiteturas diferentes.

Neste livro nós levamos os conceitos de abstração e encapsulamento um passo adiante. Na medida em que as aplicações começam a ficar mais complexas, as abstrações fornecidas pelas linguagens de programação se mostram insuficientes e precisam ser estendidas, exigindo a construção de novas abstrações a partir das já disponíveis ou a partir de novas abstrações já adicionadas à linguagem. Cada vez que adicionamos um novo nível de abstração o programador perde acesso aos detalhes de implementação do nível inferior. Apesar dessa perda de acesso parecer ruim, ela na verdade ajuda a lidar com a complexidade, pois, no final, estamos interessados em resolver o problema.

Abstrações e encapsulamento nos permitem pensar em níveis mais altos e isso é fundamental para conseguirmos resolver problemas cada vez mais complexos.

1.1 A estrutura básica de um Nó

A primeira estrutura de dados que vamos estudar é chamada *Nó*. Esta é a estrutura básica que utilizaremos para construir todas as demais estruturas de dados tratadas neste livro.

Um *Nó* é um recipiente ao qual adicionamos um elemento do nosso conjunto de dados e um apontador para o próximo *Nó* (que pode não existir). O código a seguir apresenta a definição da estrutura de um *Nó*.

Nota
Utilizaremos a linguagem C em todo o código fonte apresentado neste livro. Além disso, por convenção, nomes de tipos, variáveis e funções serão definidos em inglês. Portanto, a nossa estrutura de *Nó* se chama *Node*.

Código fonte code/capitulo-01/node.h

Definição da estrutura de um Nó e das suas operações básicas.

```
1  #ifndef ELEMENT_T
2  #define ELEMENT_T 1
3  typedef int Element; // ❶
4  #endif
5
6  #ifndef NODE_H_
7  #define NODE_H_ 1
8
9  typedef struct Node_ { //❷
10         Element value; // ❸
11         struct Node_*   next; // ❹
12 } Node;
13 #endif
14
15 Node* makeNode(Element v, Node *next); // ❺
16
17 Element getValue(Node *n); // ❻
18
19 Node* getNext(Node *n); // ❼
20
```

```
21 Node* setValue(Node *n, Element v); // ❽
22
23 Node* setNext(Node *n, Node *newNext); // ❾
```

❶, ❶ Tipo básico de dado a ser armazenado na estrutura

❷, ❷ Estrutura básica de um Nó

❸ Armazena o valor contido neste nó

❹ Apontador para o próximo (do inglês, next) Node, possivelmente null

❺ Cria um nó com valor v e next como próximo nó

❻ Retorna o valor contido no nó n

❼ Retorna o apontador para o próximo nó

❽ Altera o valor contido no nó n

❾ Faz o apontador de próximo de nó n apontar para newNext

O tipo `Element` foi definido como `int`, no entanto ele poderia representar outros tipos números, cadeias de caracteres, objetos, funções ou mesmo outros nós. Essencialmente, pode representar qualquer tipo presente na linguagem.

Neste momento, estamos preocupados apenas que o *Nó* possa armazenar valores de alguma forma. O tipo do dado, neste momento, não é relevante. Em algumas linguagens de programação, o tipo não precisa ser definido (linguagens dinamicamente tipadas como Scheme, Smalltalk ou Python). Em outras linguagens (estaticamente tipadas como C), o tipo necessita ser definido explicitamente. Há ainda uma terceira opção, na qual a decisão sobre o tipo do elemento pode ser atrasada até o momento em que o tipo é efetivamente utilizado (linguagens que suportam tipos genéricos como C++ e Java).

Cada uma das operações especificadas pode ser implementada de forma relativamente simples:

Código fonte code/capitulo-01/node.c

Implementação das operações básicas aplicáveis à estrutura de Nó

```
1  #include "node.h"
2  #include <stdlib.h>
3
4  /**
5   *       Cria um nó com valor v e next como próximo nó
6   */
7  Node* makeNode(Element v, Node *next) {
8          Node *n = (Node *) malloc(sizeof(Node));
9          n->value = v;
10         n->next = next;
11         return n;
12 }
13
14 /**
15  *       Retorna o valor contido no nó n
16  */
```

```
17 Element getValue(Node *n) {
18         return n->value;
19 }
20
21 /**
22  *      Retorna o apontador para o próximo nó
23  */
24 Node* getNext(Node *n) {
25         return n->next;
26 }
27
28 /**
29  *      Altera o valor contido no nó n
30  */
31 Node* setValue(Node *n, Element v) {
32         n->value = v;
33         return n;
34 }
35
36 /**
37  *      Faz o apontador de próximo de nó n apontar para newNext
38  */
39 Node* setNext(Node *n, Node *newNext) {
40         n->next = newNext;
41         return n;
42 }
```

Neste momento, estamos mais preocupados com as operações e a estratégia de implementação do que com a estrutura propriamente dita e sua implementação de baixo nível. Por exemplo, estamos mais preocupados com o requisito de tempo especificado, que determina que todas as operações devam tomar tempo $O(1)$. As implementações acima atendem este requisito uma vez que a quantidade de tempo que cada operação toma é constante. Uma outra forma de pensar sobre operações com tempo constante é vê-las como operações cuja análise independe de qualquer variável.

Nota
A notação $O(1)$ será formalmente apresentada no próximo capítulo. Por hora, é suficiente assumir que ela significa tempo constante.

Uma vez que o nó representa apenas um recipiente para um valor e para um apontador para o próximo nó, não deve ser surpreendente o quão trivial são a estrutura de dados propriamente dita e a implementação de suas operações.

1.2 Encadeando Nós

Apesar da estrutura de um nó ser bastante simples, ela nos permite resolver problemas bastante interessantes. Porém, primeiro vamos dar uma olhada em um programa que não precisa utilizar a nossa estrutura de nó.

O programa a seguir lê da entrada padrão uma série de números inteiros até encontrar o final da entrada e imprime o maior valor lido e a média de todos os números:

Código fonte code/capitulo-01/maxemedia.c

Código para encontrar o maior elemento e a média de um conjunto de valores lidos da entrada padrão

```c
#include <stdio.h>

#define INF 2147483647

int main() {
    int total = 0;
    int count = 0;
    int largest = -1*(INF-1);
    int i;
    while( (scanf("%d",&i) == 1 )) {
        count++;
        total += i;
        if( i > largest) largest = i;
    }
    printf( "Valor máximo: %d\n", largest);
    if(count != 0)
        printf( "Média: %d\n", (total/count));
}
```

Considere agora resolver uma tarefa similar: ler uma série de números até o final da entrada e imprimir o maior número e a média de todos os números que são divisores inteiros do maior número.

Este problema é diferente do problema anterior porque é possível que o maior número seja o último a ser fornecido, portanto, se precisamos calcular a média dos divisores do maior número precisaremos armazenar todos os números lidos. Poderíamos utilizar variáveis para armazenar os números lidos porém esta só seria uma solução se soubéssemos, a priori, quantos números seriam fornecidos.

Por exemplo, suponha que criemos, para armazenar o estado do programa, 200 variáveis inteiras, cada uma com 64 bits. Mesmo a implementação mais eficiente só poderá computar os resultados para 2^{64*200} entradas diferentes. Apesar desse ser um número muito grande de combinações, uma lista com 300 números de 64 bits exigiria ainda mais bits para ser codificada.

Ao invés de criar uma solução com limitações que dificultam a implementação (como ter apenas um número constante de variáveis), podemos utilizar as propriedades da nossa abstração de nó para nos permitir armazenar tantos números quanto nos permita a memória do computador:

Código fonte code/capitulo-01/maxemedia2.c

Código para encontrar o maior elemento de um conjunto de valores lidos da entrada padrão e a média dos divisores deste elemento

```c
#include <stdlib.h>
#include <stdio.h>
#include "node.c"

#define INF 2147483647  //Maior valor que um inteiro pode assumir.

int main() {
        int count = 0;
```

```
9           int total = 0;
10          int largest = -1*(INF-1);
11          int i;
12          Node *nodes = NULL;
13          while( (scanf("%d",&i) == 1 )) {
14                  nodes = makeNode(i, nodes);
15                  if( i > largest) largest = i;
16          }
17          printf( "Valor máximo: %d\n", largest);
18          while( nodes != NULL ) {
19                  i = getValue(nodes);
20                  if( largest % i == 0 ) {
21                          total += i;
22                          count++;
23                  }
24                  nodes = getNext(nodes);
25          }
26          if(count != 0)
27                  printf( "Média: %d\n", (total/count));
28  }
```

No código acima, se **n** números forem lidos, a função `makeNode` será invocada **n** vezes. Isto exige que **n** nós sejam criados (o que requer espaço de memória suficiente para armazenar o valor e o apontador para o próximo nó), portanto, o requisito de memória da solução é da ordem O(n).

De forma similar, construímos uma **cadeia de nós** e iteramos ao longo de toda a cadeia, o que requer O(n) passos para criar a cadeia e outros O(n) passos para iterar sobre ela.

Note que quando iteramos sobre os números na cadeia, estamos na verdade acessando-os em ordem inversa a da entrada. Por exemplo, assuma que os números na entrada do programa são 4, 7, 6, 30 e 15. Após encontrar o final da entrada, a cadeia de nós terá a seguinte forma:

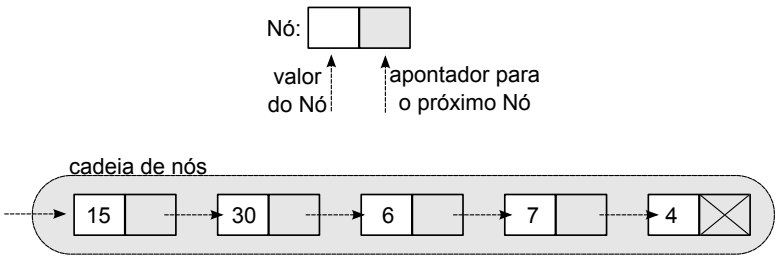


Figura 1.1: Cadeia com n nós

Tais cadeias são mais comumente conhecidas como listas encadeadas. No entanto, geralmente preferimos pensar nelas em termos de listas ou sequências, que são conceitos de mais alto nível: o conceito de encadeamento é apenas um detalhe de implementação. Apesar de uma lista poder ser construída por encadeamentos, neste livro abordaremos várias outras formas para implementar uma lista. No momento, nos preocupamos mais com as possibilidades de utilização da nossa estrutura de nó do que com uma das formas como ela pode ser utilizada.

Dica

O algoritmo anterior utiliza apenas as funções `makeNode`, `getValue` e `getNext`. Modifique o código apresentado utilizando a função `setNext` para gerar uma cadeia que mantém a ordem original da entrada ao invés de invertê-la.

1.3 O Princípio da Indução

As cadeias que podemos construir a partir de um nó são demonstrações do princípio da indução matemática:

Indução Matemática

1. Suponha que você tenha uma propriedade dos números naturais *P*(n)
2. Se você puder provar que quando *P*(n) é verdade *P*(n+1) também precisa ser verdade, então
3. Tudo que você precisa fazer para provar que *P*(n) é verdade para qualquer número natural *n* é mostrar que *P*(1) é verdade.

Por exemplo, seja a propriedade *P*(n) a afirmação de que "é possível construir uma cadeia para armazenar *n* números". Esta é uma propriedade dos números naturais pois a sentença faz sentido para valores específicos de *n*:

- É possível construir uma cadeia para armazenar 5 números
- É possível construir uma cadeia para armazenar 100 números
- É possível construir uma cadeia para armazenar 1.000.000 de números

Ao invés de provar que podemos construir uma cadeia de comprimento 5, 100 ou 1 milhão, podemos provar a afirmação geral *P*(n). Vamos ver como provar o passo 2 descrito acima, denominado **Hipótese Indutiva**:

- Assuma que *P*(n) é verdade, ou seja, que é possível construir uma cadeia com *n* elementos. Agora precisamos mostrar que *P*(n+1) também é verdade.
- Assuma que *chain* é o primeiro nó de uma cadeia com *n* elementos. Assuma que *i* é algum número que nós gostaríamos de adicionar à cadeia fazendo com que seu tamanho aumente para *n+1*.
- O código a seguir pode fazer isso para nós

```
Node *biggerChain = makeNode(i,chain);
```

- Agora `i` é o valor do primeiro nó da cadeia denominada `biggerChain`. Se a cadeia `chain` tinha `n` elementos então `biggerChain` precisa ter `n + 1` elementos.

O passo 3 mais acima é denominado **Caso Base**. Vamos ver como podemos provar que *P*(1) é verdade.

- Precisamos mostrar que $P(1)$ é verdade, ou seja, que podemos criar uma cadeia com um elemento.
- O código a seguir poder fazer isso para nós

```
Node *chain = makeNode(i, NULL);
```

O princípio da indução nos diz agora que provamos que é possível construir uma cadeia com `n` elementos para qualquer valor de `n` $\geq$ `1`. Como isso é possível? Provavelmente, a melhor forma de pensar sobre a indução é que ela é na verdade uma forma de criar uma fórmula que descreve um número infinito de provas. Depois que provamos que a afirmação é verdade para $P(1)$, o caso base, podemos aplicar a hipótese indutiva para mostrar que $P(2)$ é verdade. Sabendo que $P(2)$ é verdade, podemos novamente aplicar a hipótese indutiva para mostrar que $P(3)$ é verdade. O princípio nos diz que não há nada que nos impeça de continuar fazendo isso repetidamente, portanto, podemos assumir que P é verdade para qualquer `n` $\geq$ `1`.

A indução pode parecer a primeira vista uma forma estranha de prova, mas é uma técnica extremamente útil. O que faz está técnica ser tão útil é que podemos pegar um problema que a princípio parecer ser bem difícil "provar que P(n) é verdade para qualquer n $\geq$ 1"e quebrá-lo em afirmações mais simples e mais fáceis de provar. Tipicamente os casos base são fáceis de provar porque não são afirmações gerais. Portanto, a maior parte do trabalho se concentra em provar que a hipótese indutiva é válida, o que muitas vezes envolve reformular as afirmações de forma que elas possam ser utilizadas em uma prova de validade para $n + 1$.

Você pode pensar no valor contido em um nó como o caso base e no apontador para o próximo nó como a hipótese indutiva. Da mesma forma que na indução matemática, podemos quebrar o difícil problema de armazenar um número arbitrário de elementos em um problema muito mais simples, que é armazenar um único elemento é ter um mecanismo para adicionar novos elementos.

1.4 Recapitulando

Neste capítulo nós vimos a importância da utilização de estruturas de dados e conhecemos dois conceitos fundamentais para o desenvolvimento de software: **encapsulamento** e **abstração**.

Implementamos uma estrutura de dados rudimentar, chamada nó, e a utilizamos para construir cadeias para armazenar uma quantidade arbitrária de elementos. Essa estrutura rudimentar será a base para todas as estruturas de dados que abordaremos ao longo deste livro.

Finalmente, vimos como o princípio da **indução matemática** está associado à concepção de estruturas em cadeia.

No próximo capítulos estudaremos o conceito de **notação assintótica**, o que nos permitirá estimar a quantidade de tempo necessária para a execução de um algoritmo e a quantidade de memória necessária para armazenar uma determinada estrutura de dados.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?

Acesse https://github.com/edusantana/estruturas-de-dados-livro/issues/new para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**capitulo-01**) e a versão do livro (**v1.0.0**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 2

Notação Assintótica

Sumário

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender a noção de comparação assintótica de funções;
- Conhecer as principais notações utilizadas para denotar a relação entre a complexidade assintótica de duas funções;
- Ser capaz de identificar a classe de crescimento assintótico de uma determinada função utilizando a notação O;

2.1 Introdução

Um problema pode ter várias soluções algorítmicas. Para poder escolher a melhor dentre as diversas opções, é preciso ser capaz de julgar quanto tempo cada solução precisa para resolver o problema. Mais precisamente, você não precisa saber o tempo de execução de uma solução em minutos ou segundos mas deve ser capaz de, dadas duas soluções, decidir qual delas é a mais rápida.

Importante

Complexidade assintótica é uma forma de expressar o componente principal relacionado ao tempo de execução de um algoritmo utilizando unidades abstratas de trabalho computacional.

Considere, por exemplo, um algoritmo para ordenar uma pilha de cartas de baralho que percorre repetidas vezes a pilha a procura da menor carta. A complexidade assintótica deste algoritmo é o *quadrado* do número de cartas na pilha. Este componente quadrático é o termo principal da fórmula que conta quantas operações o algoritmo precisa para ordenar a pilha de cartas e ele mostra, por exemplo, que se dobramos o tamanho da pilha, o trabalho necessário para ordená-la é quadruplicado.

A formula exata para o custo de execução do algoritmo é um pouco mais complexa e contém detalhes adicionais necessários para entender a complexidade do algoritmo. Em relação ao problema de ordenar a pilha e cartas, o pior caso acontece quando a pilha se encontra em ordem decrescente pois nessa situação a busca pela menor carta sempre teria que ir até o final da pilha. A primeira busca iria varrer as 52 cartas do baralho, a próxima varreria as demais 51, a seguinte 50 e assim por diante. Portanto, a formula para o custo de execução seria $52 + 51 + 50 + \ldots + 2$. Sendo N o número de cartas, a formula geral é $2 + \ldots + N - 1 + N$, o que é igual a $((1 + N) \times N)/2) - 1 = ((N^2 + N)/2) - 1 = (1/2)N^2 + (1/2)N - 1$. Note que o termo N^2 domina a expressão, sendo a chave para comparação do custo entre dois algoritmos.

Assintóticamente falando, quando N tende ao infinito, a soma $2 + 3 + \ldots + N$ se aproxima cada vez mais da função puramente quadrática $(1/2)N^2$. Além disso, com um valor de N muito grande, o fator constante $1/2$ pode ser desprezado. Com isso, podemos dizer que o algoritmo tem complexidade $O(n^2)$.

Nota

O algoritmo de ordenação descrito acima é **muito ineficiente**. Veremos no decorrer do livro algoritmos bem mais eficientes que este, com tempos de execução sub-quadráticos.

Considere agora a situação em que é necessário comparar a complexidade de dois algoritmos. Seja $f(n)$ o custo, no pior caso, de um dos algoritmos, expresso como uma função do tamanho da entrada n, e $g(n)$ a função que descreve o custo, no pior caso, do outro algoritmo.[1] Se para todos os valores de $n \geq 0$, $f(n) \leq g(n)$, dizemos que o algoritmo com função de complexidade f é **estritamente mais rápida** do que g.

Porém, de forma geral, nossa preocupação em termos de custo computacional está mais focada nos casos onde n é muito grande; portanto, a comparação entre $f(n)$ e $g(n)$ para valores pequenos de n é menos significativa do que a comparação entre $f(n)$ e $g(n)$ para n maior que um determinado limiar.

Nota

Estamos discutindo *limites* para o desempenho de algoritmos ao invés de medir a *velocidade exata* de cada um. O número exato de passos necessários para ordenar a nossa pilha de cartas (com nosso algoritmo simplório) depende diretamente da ordem em que as cartas se encontram inicialmente. O tempo necessário para executar cada um dos passos depende diretamente da velocidade do processador, do conteúdo da memória cache e de vários outros fatores.

[1] Por exemplo, considerando dois algoritmos de ordenação, $f(10)$ e $g(10)$ seriam as quantidades máximas de passos que cada um dos algoritmos precisaria para ordenar uma lista com 10 itens.

2.2 A notação Big-O

A notação *O* (Big-O) é o método formal utilizado para expressar o *limite superior* no tempo de execução de um algoritmo.[2] É uma medida da maior quantidade de tempo que um determinado algoritmo pode precisar para completar sua tarefa.

De maneira mais formal, para duas funções não-negativas, $f(n)$ e $g(n)$, se existe um inteiro n_0 e uma constante $c > 0$ tal que para todos os inteiros $n > n_0$, $f(n) \leq cg(n)$, então $f(n)$ tem complexidade *big oh* $g(n)$, denotada por:

$f(n) = O(g(n))$ — $g(n)$ *limita superiormente* $f(n)$

No gráfico a seguir é possível observar visualmente o comportamento das funções f e g. Note que para $n > n_0$, $f(n) \leq cg(n)$.

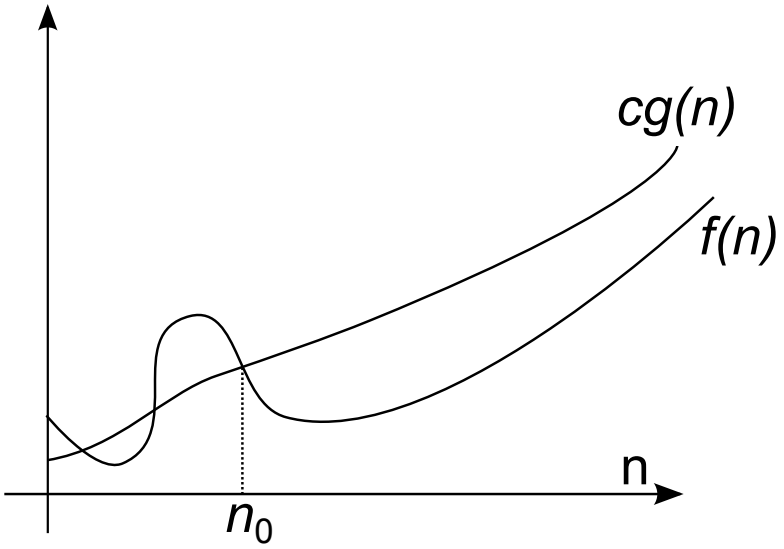


Figura 2.1: Notação *O*

Exemplo 2.1 Funções com complexidade $O(n^2)$

$f(n) = n$
$f(n) = n/1000$
$f(n) = n^{1.9}$
$f(n) = n^2$
$f(n) = n^2+n$
$f(n) = 1000n^2 + 50n$

Importante
Lembre-se que a notação *O* denota o **limite superior** da função quando *n* for muito grande. Consequentemente qualquer função quadrada *limita superiormente* uma função linear, considerando os casos em que ambas forem não negativas.

[2] Do inglês, pronuncia-se *Big ou*. Também é encontrado na literatura por "big oh".

Exemplo 2.2 Funções com complexidade superior a $O(n^2)$

$f(n) = n^3$
$f(n) = n^{2.1}$
$f(n) = 2^n$

Exemplo 2.3 Complexidade de alguns algoritmos

Os exemplos apresentados a seguir são **informativos**, não é esperado que você já possua o conhecimento para calcular as complexidades deste algoritmos:

1. Imprimir uma lista de n itens na tela, examinando cada um deles uma única vez = $O(n)$
2. Receber uma lista de n elementos e dividi-la ao meio sucessivas vezes até que reste um único elemento = $O(\log n)$
3. Receber uma lista de n elementos e comparar cada elemento com todos os outros elementos = $O(n^2)$

Exemplo 2.4 Mostre que $2n + 8 = O(n^2)$

Sejam $f(n) = 2n + 8$ e $g(n) = n^2$. Podemos encontrar uma constante n_0 tal que $2n + 8 \leq n^2$?
Analisando as funções podemos ver que para $n_0 \geq 4$, $2n + 8 \leq n^2$.
Uma vez que queremos generalizar essa expressão para valores de n grandes, e valores pequenos para n (1, 2 e 3) não são importantes, pode-se dizer que $f(n)$ é mais rápida que $g(n)$, ou seja, $f(n)$ é limitada por $g(n)$. Portanto, $f(n) = O(n^2)$.

Existem alguns atalhos para encontrar o limite superior de uma função. No exemplo anterior, podemos remover todas as constantes da expressão pois, eventualmente, para algum valor c, elas se tornam irrelevantes. Isso faz $f(n) = 2n$. Além disso, para conveniência na comparação, podemos também remover constantes multiplicativas, neste caso, o 2, fazendo $f(n) = n$. Com isso, podemos dizer que $f(n) = O(n)$, e isso nos dá um limite mais *apertado* para f.

2.2.1 Exercícios

1. Faz sentido dizer que $f(n) = O(n^2)$ para $n \geq 4$?
2. É verdade que $10n = O(n)$?
3. É verdade que $10n^{55} = O(2^n)$?
4. É verdade que $n^2 + 200n + 300 = O(n^2)$?
5. É verdade que $n^2 - 200n - 300 = O(n)$?
6. É verdade que $(3/2)n^2 + (7/2)n - 4 = O(n)$?
7. É verdade que $(3/2)n^2 + (7/2)n - 4 = O(n^2)$?
8. É verdade que $n^3 - 999999n^2 - 1000000 = O(n^2)$?
9. Prove que $n = O(2^n)$.

10. Prove que $n = O(2^{n/4})$.

Dica
Nas questões 9 e 10, use indução matemática para provar que $n \leq 2^n$ e que $n \leq 2^{n/4}$ para todo n suficientemente grande.

Nota
Os algoritmos e estruturas de dados apresentadas neste livro terão sua complexidade temporal e espacial apresentadas utilizando a notação *O*. Porém, existem várias outras notações utilizadas para descrever complexidade assintótica. A seguir veremos algumas dessas notações e o seu significado.

2.3 Notação Big-Omega

A notação Ω (*omega*) é utilizada para expressar o *limite inferior* no tempo de execução de um algoritmo. É uma medida da menor quantidade de tempo que um determinado algoritmo pode precisar para completar sua tarefa.

Para duas funções não negativas, $f(n)$ e $g(n)$, se existe um inteiro n_0 e uma constante $c > 0$ tal que para todos os inteiros $n \geq n_0$, $f(n) \geq cg(n)$, $f(n)$ tem complexidade *Omega* $g(n)$, denotado por:

$f(n) = \Omega(g(n))$ — *g(n) limita inferiormente f(n)*

Esta é praticamente a mesma definição para a notação *O*, com a exceção de que $f(n) \geq cg(n)$ implica em dizer que $g(n)$ representa o **melhor caso** para $f(n)$, como ilustrado na Figura 2.2 [14] Neste caso, $g(n)$ descreve o *melhor* que pode acontecer para um determinado tamanho de entrada.

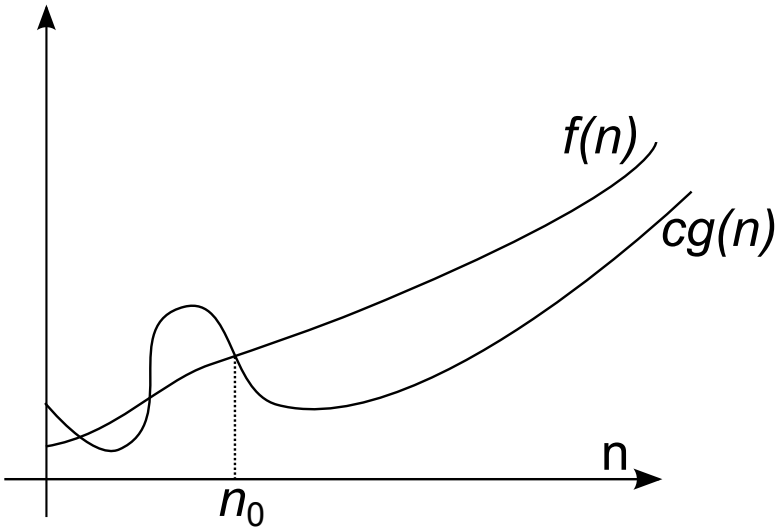


Figura 2.2: Notação Ω

Exemplo 2.5 Funções com complexidade $\Omega(n)$

$f(n) = n$
$f(n) = n/1000$
$f(n) = n^{1.9}$
$f(n) = n^2$
$f(n) = n^2+n$
$f(n) = 1000n^2 + 50n$

Exemplo 2.6 Funções com complexidade inferior a $\Omega(n^2)$

$f(n) = n$
$f(n) = n^{1.9}$
$f(n) = log(n)$

2.3.1 Exercícios

1. Prove que $100 \lg n - 10n + 2n \lg n = \Omega(n \lg n)$.
2. É verdade que $2^{n+1} = \Omega(2\hat{}n)$?
3. É verdade que $3^n = \Omega(2^n)$?

2.4 Notação Theta

A notação Θ (*theta*) é utilizada para definir um *limite assintótico restrito* para o tempo de execução de um algoritmo. É uma medida que define tanto limites superiores quanto inferiores para o crescimento da função $f(n)$.

Sejam duas funções não-negativas $f(n)$ e $g(n)$, $f(n)$ é dita theta de $g(n)$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$, denotado por:

$$f(n) = \Theta(g(n))$$

Isto significa que a função $f(n)$ é limitada tanto superior quanto inferiormente, pela função $g(n)$, conforme ilustrado na Figura 2.3 [16].

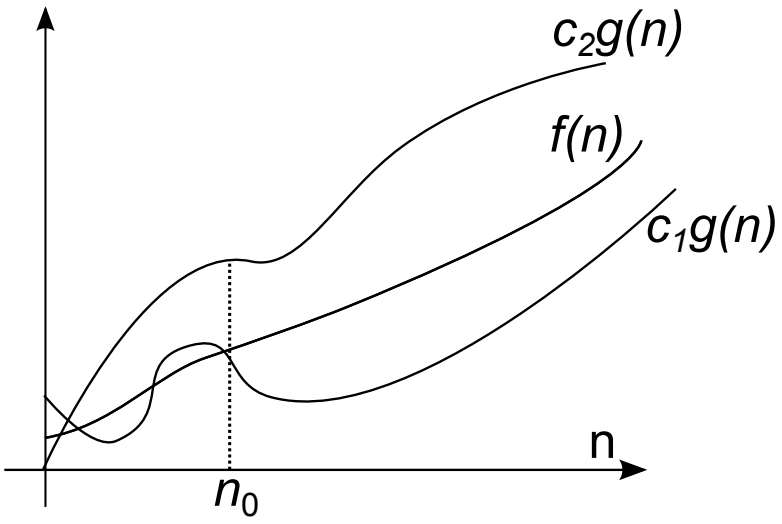


Figura 2.3: Notação Θ

2.4.1 Exercícios

1. É verdade que $(3/2)n^2 + (7/2)n^3 - 4 = \Theta(n^2)$?
2. É verdade que $9999n^2 = \Theta(n^2)$?
3. É verdade que $n^2/1000 - 999n = \Theta(n^2)$?
4. É verdade que $log_2 n + 1 = \Theta(log_{10} n)$?

2.5 Outras notações

2.5.1 Notação Little-o

A notação *o* representa uma versão mais solta da notação *O*. Nesta notação, *g(n)* limita a função *f(n)* superiormente mas não representa um limite assintótico restrito.

Sejam duas funções não-negativas, *f(n)* e *g(n)*, *f(n)* é dita *little o* de *g(n)* se e somente se $f(n) = O(g(n))$ mas $f(n) \neq \Theta(g(n))$, denotando-se por $f(n) = o(g(n))$.

2.5.2 Notação Little-omega

A notação ω (*pronunciada little omega*) representa uma versão mais solta da notação Ω. Nesta notação *g(n)* limita a função *f(n)* inferiormente mas não representa um limite assintótico restrito.

Sejam duas funções não-negativas, *f(n)* e *g(n)*, *f(n)* é dita *little omega* de *g(n)* se e somente se $f(n) = \Omega(g(n))$ mas $f(n) \neq \Theta(g(n))$, denotando-se por $f(n) = \omega(g(n))$.

2.6 Notação Assintótica e Análise de Complexidade

Tempo de execução não é a única métrica de interesse ao analisarmos a complexidade de um algoritmo. Existem também fatores de interesse relacionados ao espaço de memória utilizado. Geralmente, há uma relação direta entre tempo de execução e quantidade de memória utilizada. Se pensarmos na quantidade de tempo e espaço que um programa utiliza como uma função do tamanho da entrada podemos analisar como estas métricas evoluem quando mais dados são introduzidos no programa.

Isto é particularmente importante no projeto de estruturas de dados pois geralmente desejamos utilizar uma estrutura que se comporte de forma eficiente quando grandes volume de dados são fornecidos. Entretanto, é importante compreender que nem sempre algoritmos que são eficientes para grandes volumes são simples e eficientes para pequenos volumes de dados.

Importante
A **notação assintótica** é geralmente utilizada como uma forma conveniente de expressar o que acontece com uma função no pior e no melhor caso.

Por exemplo, se você deseja escrever uma função que pesquisa um arranjo de números em busca do menor valor:

Código para encontrar o menor elemento de um arranjo

```
1 int findMin(int array[], int size) {
2         int min = INT_MAX;
3         int i;
4         for(i = 0; i < size; i++) {
5                 if(array[i] < min)
6                         min = array[i];
7         }
8         return min;
9 }
```

Independente do quão grande seja o arranjo, sempre que a função `findMin()` é executada é preciso inicializar as variáveis `min` e `i` e retornar a variável `min` ao fim da execução. Portanto, podemos considerar estes trechos do código como constantes e ignorá-los no cálculo da complexidade.

Então, como podemos utilizar a notação assintótica para descrever a complexidade da função `findMin()` ? Se a função for utilizada para encontrar o valor mínimo de um arranjo com 87 elementos o `for` realizará 87 iterações, mesmo que o menor elemento seja o primeiro a ser visitado. De maneira análoga, para `n` elementos o `for` realizará `n` iterações. Portanto, podemos dizer que a função `findMin()` tem tempo de execução *O(n)*.

O que dizer agora acerca da função abaixo:

Código fonte code/capitulo-02/findMinAndMax.c

Função para encontrar o menor e maior elementos de um arranjo

```
1 void findMinAndMax(int array[], int size, int * minimum,
2     int * maximum) {
3   int min = INT_MAX;
4   int max = INT_MIN;
```

```
5
6     //Encontra o valor mínimo no arranjo
7     for(int i=0; i < size; i++) {
8       if(array[i] < min)
9         min = array[i];
10    }
11
12    //Encontra o valor máximo no arranjo
13    for(int i=0; i < size; i++) {
14      if(array[i] > max)
15        max = array[i];
16    }
17
18    // Retorna os valores por referência
19    *minimum=min;
20    *maximum=max;
21  }
```

Qual função melhor representa o tempo de execução da função `findMinAndMax()`? Existem dois laços e cada um realiza n iterações sendo assim o tempo de execução é claramente $O(2n)$. Uma vez que 2 é uma constante ele pode ser ignorado e o tempo de execução da função pode ser descrito como $O(n)$.

2.7 Recapitulando

Neste capítulo vimos uma noção sobre a comparação assintótica de funções e conhecemos as principais notações utilizadas para expressar o relacionamento assintótico entre duas funções.

Uma analogia imprecisa entre a comparação assintótica de duas funções f e g e a relação entre seus valores é dada a seguir:

$f(n) = O(g(n)) \approx f(n) \leq g(n)$

$f(n) = \Omega(g(n)) \approx f(n) \geq g(n)$

$f(n) = \Theta(g(n)) \approx f(n) = g(n)$

$f(n) = o(g(n)) \approx f(n) < g(n)$

$f(n) = \omega(g(n)) \approx f(n) > g(n)$

Estas notações, particularmente a notação O, serão utilizadas no decorrer deste livro para expressar complexidade temporal de algoritmos e complexidade espacial de estruturas de dados.

No próximo capítulo iniciamos efetivamente o estudo de estruturas de dados abordando os arranjos (*arrays*) unidimensionais, também denotados *vetores* e arranjos multi-dimensionais, que podem ser denominados de *matrizes* no caso de termos apenas duas dimensões.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?

Acesse https://github.com/edusantana/estruturas-de-dados-livro/issues/new para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**capitulo-02**) e a versão do livro (**v1.0.0**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 3

Arranjos

Sumário

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender o conceito e funcionamento de um arranjo
- Implementar funções utilizando arranjos unidimensionais e multi-dimensionais em uma linguagem de programação

3.1 Introdução

Um arranjo (do inglês *array*) é uma estrutura de dados linear utilizada principalmente para armazenar dados similares. É um método particular para armazenar elementos de um conjunto de dados *indexável*. Os elementos são armazenados sequencialmente em blocos dentro do arranjo. Cada elemento é referenciado através de um índice.

O índice, normalmente um número, é utilizado para endereçar um elemento do arranjo. Por exemplo, se desejamos armazenar informações sobre cada um dos dias do mês de Agosto, precisaríamos criar um arranjo capaz de armazenar 31 valores, um para cada dia do mês. As regras de indexação dependem da linguagem de programação utilizada, porém, a maioria das linguagens utiliza 0 ou 1 como índice para o primeiro elemento do arranjo.

O conceito de arranjo pode parecer desafiador para leitores com pouca experiência em programação mas é, na verdade, algo bastante simples. Pense em um caderno com páginas numeradas de 1 até 12. Cada página pode conter ou não dados. O caderno é um arranjo de páginas. Cada página é um *elemento* do arranjo de páginas denominado `caderno`. Podemos extrair informações de forma

programática de uma página nos referindo a ela através de seu índice (`caderno[4]` é uma referência ao conteúdo da quarta página do arranjo `caderno`.).

Figura 3.1: Um caderno (*arranjo*) com 12 páginas (*elementos*)

Arranjos podem ser também multi-dimensionais — ao invés de acessar elementos de uma lista unidimensional, os elementos são acessados através de dois ou mais índices, de forma semelhante a uma matriz.

Arranjos multi-dimensionais são tão simples quanto o arranjo do nosso exemplo anterior. Para visualizar um arranjo multi-dimensional imagine um calendário. Cada página do calendário, com índices variando de 1 a 12, representa um mês. Cada mês contém aproximadamente 30 dias, que são seus elementos. Cada dia pode possuir informações associadas a ele: se é feriado, a lua etc. Portanto, de forma programática, `Calendario[4][15]` seria uma referência ao 15º dia do 4º mês. Portanto, teríamos um arranjo bi-dimensional. Para visualizar um arranjo tridimensional basta quebrar cada dia em 24 horas, como uma Agenda de Compromissos. Desta forma, teríamos algo como `Agenda[4][15][9]` para nos referir a 9ª hora do 15º dia do 4º mês.

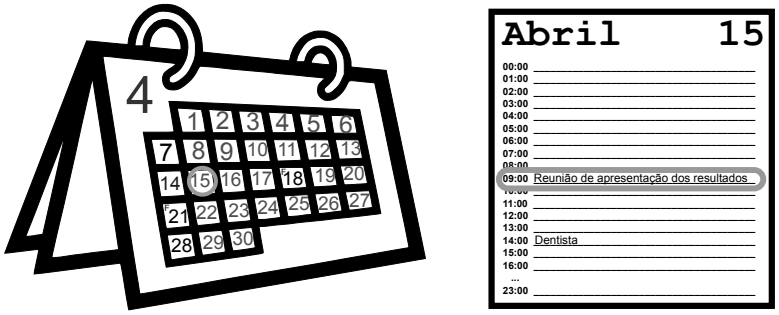


Figura 3.2: Arranjos bi-dimensional (Calendario) e tri-dimensional (Agenda)

3.2 Operações de manipulação em arranjos unidimensionais

No código a seguir apresentamo as três operações básicas para manipulação de arranjos unidimensionais.

Código fonte code/capitulo-03/array.h

Definição das operações básicas para manipulação de arranjos unidimensionais

```
1 #ifndef ELEMENT_T
2 #define ELEMENT_T 1
3
4 typedef int Element;
5
6 #endif
7
8 /**
```

```
 * Cria um arranjo de n elementos indexados de 0 a n - 1, inclusive.
 */
Element * makeArray(int n);

/**
 * Retorna o valor do elemento de índice index.
 * O valor de index precisar estar no intervalo 0 <= index <= (n - 1)
 */
Element getValueAt(Element *, int index);

/**
 * Altera o valor do elemento de índice index no arranjo para o
 * valor newValue.
 */
void setValueAt(Element *, int index, Element newValue);
```

Arranjos garantem tempo *constante* para acessos de leitura e escrita: *O(1)*. Várias operações de consulta nos arranjos possuem complexidade linear, *O(n)*, tais como encontrar o menor valor, encontrar o maior valor, encontrar o índice de um determinado elemento, etc.

A implementação de arranjos é bastante eficiente na maioria das linguagem de programação uma vez que suas operações computam o endereço de cada elemento do arranjo através de uma função simples baseada no *endereço base* do arranjo.

A implementação de arranjos difere bastante de uma linguagem de programação para outra. Algumas linguagens permitem que um arranjo seja redimensionado automaticamente ou que contenha elementos de diferentes tipos (como em Perl). Outra linguagens são mais estritas e exigem que o tipo e tamanho do arranjo sejam conhecidos em tempo de compilação (como em C).

Arranjos são geralmente mapeados diretamente para espaços contíguos de memória e são, portanto, a estrutura de armazenamento "mais natural"na maioria das linguagens de programação de alto nível.

Arranjos lineares simples são a base para grande parte das demais estruturas de dados. De tal forma que várias linguagens de programação não permitem que o programador aloque memória para qualquer outra estrutura além dos arranjos. Isso é uma forma de garantir que todas as demais estruturas de dados sejam implementados utilizando arranjos como base.[1]

3.2.1 Tipos e Limites

Os índices de um arranjo precisam ser de um mesmo tipo. Geralmente é utilizado o tipo inteiro padrão da linguagem, mas existem linguagens como *Ada* e *Pascal* que permitem a utilização de qualquer tipo discreto como índice de um arranjo. Linguagens de *script*, como *Python*, permitem que qualquer tipo possa ser utilizado como índice para um arranjo (*arranjos associativos*).

Cada arranjo possui um limite **inferior** e **superior** para os seus índices. Algumas linguagens de programação fixam o valor do limite inferior em *0* (*C, C++, C#, Java*) ou 1 (*FORTRAN 66*). Outras linguagens (*Ada, PL/I, Pascal*) permitem que o limite inferior seja livremente definido, inclusive com valores negativos.

O terceiro aspecto relacionado ao acesso a arranjos é a **checagem de validade do índice** e a definição do que acontece quando um índice inválido é fornecido. Este é um ponto muito importante

[1] A exceção seriam as listas encadeadas, que podem ser implementados utilizados objetos alocados individualmente. Porém, também é possível implementar uma lista encadeada utilizando um arranjo.

pois softwares maliciosos como *vírus* e *vermes* geralmente se aproveitam de falhas de checagem nos limites de arranjos para contaminar programas legítimos.

Existem três opções mais comuns para checagem de limites em arranjos:

1. A maioria das linguagens (*Ada, PL/I, Pascal, Java, C#*) checa os limites do arranjo e lança uma exceção quando se tenta acessar um elemento fora do arranjo.
2. Algumas poucas linguagens (*C, C++*) não checam por acessos fora dos limites do arranjo e simplesmente retornam algum valor arbitrário quando um elemento inválido é acessado.
3. Linguagens de *script* geralmente expandem automaticamente o tamanho do arranjo quando dados fora do limite inicial do arranjo são acessados.

3.2.2 Declarando Arranjos

A declaração de um arranjo difere bastante de uma linguagem de programação para outra e é afetada diretamente pelas funcionalidades fornecidas por cada uma delas.

A forma de declaração mais simples ocorre quando a linguagem define um valor mínimo e tipo fixo para os índices. Se você precisa de um arranjo para armazenar seus rendimentos mensais você poderia declará-lo em linguagem *C* da seguinte forma:

Declaração de um arranjo unidimensional na linguagem C

```
typedef double Rendimento[12];
```

Esta declaração define um novo tipo denominado *Rendimento* que representa um arranjo de números em ponto flutuante com os índices dos elementos variando de `0` a `11`.[2]

Algumas linguagens permitem que se especifique o valor mínimo e máximo e o tipo do índice. A declaração nesse caso se torna um pouco mais complexa. A seguir apresentamos um exemplo de declaração de arranjos na linguagem *Ada*:

Declaração de um arranjo em linguagem Ada

```
type Mes is range 1 .. 12;
type Rendimento is array(Mes) of Float;
```

3.2.3 Exercícios

1. Forneça uma implementação com complexidade *O(1)* para as três operações básicas para manipulação de arranjos unidimensionais definidas neste capítulo: `makeArray`, `getValueAt` e `setValueAt`.

[2] Este tipo poderia armazenar os rendimentos de uma pessoa durante um ano.

Dica

Para auxiliá-lo na resolução deste exercício preparamos uma proposta de solução que você **pode** adotar, utilizando os seguintes arquivos:

- code/capitulo-03/array.h
- code/capitulo-03/operacoes_basicas_setembro1999.c

Nesta proposta você necessita fornecer a implementação das funções indicadas. A proposta possui testes (`assert`) que podem serem utilizados para verificar sua solução.
Lembre-se que a seção “Baixando os códigos fontes” [viii] explica como baixar e acessar os códigos fonte do livro.

2. Assuma que estas funções são a única forma de acesso aos elementos de um arranjo. Como estas funções poderiam ser utilizadas para manipular elementos em um arranjo bi-dimensional?

Dica

Para auxiliá-lo na resolução deste exercício preparamos uma proposta de solução que você **pode** adotar, utilizando os seguintes arquivos:

- code/capitulo-03/array.h
- code/capitulo-03/operacoes_basicas_setembro1999_bidimencional.c

Nesta proposta você deve fornecer o corpo da implementação das funções de acesso bidimencional.

3. Novamente assumindo que estas funções são a única forma de acesso aos elementos de um arranjo, forneça implementações com complexidade *O(n)* para as seguintes funções:

   a. Encontrar o menor elemento de um arranjo unidimensional
   b. Encontrar o maior elemento de um arranjo bi-dimensional
   c. Encontrar o índice de um elemento de arranjo unidimensional (retornando `-1` se o elemento não for encontrado).

Dica

Para auxiliá-lo na resolução deste exercício preparamos uma proposta de solução que você **pode** adotar, utilizando os seguintes arquivos:

- code/capitulo-03/array.h
- code/capitulo-03/operacoes_de_consulta.c

Nesta proposta você deve utilizar a propriedade `temperatura` para realização das consultas. Então, o “menor elemento” significa o “dia com a menor temperatura”.

3.3 Recapitulando

Arranjos são, em muitas linguagens de programação, a estrutura de dados base para a construção de outras estruturas.

Neste capítulos aprendemos um pouco sobre o funcionamento destas estruturas e tivemos a oportunidade de praticar a implementação de código utilizando arranjos unidimensionais e multi-dimensionais.

No próximo capítulo começaremos a estudar uma família de estruturas de dados bastante semelhantes às estruturas que já estudamos (cadeia no capítulo 1 e arranjo no capítulo 3) mas que oferecem um nova gama de funcionalidades e oportunidades: as estruturas de lista.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?

Acesse https://github.com/edusantana/estruturas-de-dados-livro/issues/new para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**capitulo-03**) e a versão do livro (**v1.0.0**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 4

Estruturas de Lista e Iteradores

Sumário

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Compreender o conceito de lista encadeada
- Conhecer diferentes variações e implementações de listas encadeadas
- Ser capaz de escolher entre estas variações a mais adequada para resolver determinados tipos de problemas.

Vimos até este momento duas estruturas de dados diferentes que permitem armazenar uma sequência ordenada de elementos. No entanto, tais estruturas apresentam *interfaces* distintas. A estrutura de **arranjo** utiliza as funções *setElement()* e *getElement()* para acessar e alterar elementos. Já a **cadeia** requer a utilização da função *getNext()* até que o nó desejado seja encontrado e, neste momento, a utilização das funções *getValue()* e *setValue()* para acessar ou modificar seu valor.

Imagine agora que você escreveu um código utilizando uma dessas estruturas e de repente chegou a conclusão que precisaria, na verdade, utilizar a outra. Você precisará percorrer todo o código já escrito e alterar as funções de acesso à estrutura de dados. Isso, dependendo do tamanho do programa, pode ser uma operação bastante custosa.

Felizmente, existe uma forma de concentrar essas alterações em um único local: utilizando tipos abstrato de dados. Iniciamos então apresentando dois novos *TADs*: uma *lista* e um *iterador*. A seguir encontramos a definição desses novos tipos abstratos de dados.

Exemplo 4.1 Definição do tipo abstrato de dados *List*

Código fonte code/capitulo-04/list.h

```
/**
 * Retorna um iterador que aponta para o primeiro elemento da lista.
 * Tem complexidade O(1).
 */
Iterator getBegin(List list);

/**
 * Retorna um iterador que aponta para o elemento após o último
 * elemento da lista.
 * Tem complexidade O(n).
 */
Iterator getEnd(List list);

/**
 * Adiciona um elemento no início da lista.
 * Tem complexidade O(1).
 */
List prepend(List list, Element newElement);

/**
 * Adiciona um elemento imediatamente após o elemento apontado por it.
 * Tem complexidade O(n).
 */
void insertAfter(Iterator it, Element newElement);

/**
 * Remove o primeiro elemento da lista.
 * Tem complexidade O(1).
 */
List removeFirst(List list);

/**
 * Remove o elemento imediatamente após o elemento apontado por it.
 * Tem complexidade O(n).
 */
void removeAfter(Iterator it);

/**
 * Retorna 1 se a lista estiver vazia.
 * Possui uma implementação default.
 * Tem complexidade O(n).
 */
int isEmpty(List list);

/**
 * Retorna o número de elementos da lista.
 * Possui uma implementação default.
 * Tem complexidade O(n).
 */
```

```
50 int getSize(List list);
51
52 /**
53  * Retorna o e-nésimo elemento da lista, contando a partir do 0.
54  * Possui uma implementação default.
55  * Tem complexidade O(n).
56  */
57 Element getNth(List list, int n);
58
59 /**
60  * Altera o valor do e-nésimo elemento da lista, contando a partir
61  * do 0. Possui uma implementação default.
62  * Tem complexidade O(n).
63  */
64 void setNth(List list, int n, Element newElement);
```

Um *iterador* é uma outra abstração que encapsula tanto o acesso a um único elemento quanto a movimentação ao longo da lista. Sua interface é bastante similar a interface da estrutura de *Nó* apresentada na introdução, mas como ele é um tipo abstrato, diferentes tipos de listas podem fornecer implementações diferentes.

Exemplo 4.2 Definição do tipo abstrato de dados *Iterator*

Código fonte code/capitulo-04/iterator.h

```
1  /**
2   * Retorna o elemento da lista apontado por este iterador.
3   * Tem complexidade O(1).
4   */
5  Element getElement(Iterator it);
6
7  /**
8   * Altera o valor do elemento apontado por este iterador.
9   * Tem complexidade O(1).
10  */
11 void setElement(Iterator it, Element newElement);
12
13 /**
14  * Faz o iterador apontar para o próximo elemento da lista.
15  * Tem complexidade O(1).
16  */
17 Iterator moveNext(Iterator it);
18
19 /**
20  * Retorna 1 se os dois iteradores apontam para o mesmo elemento
21  * da lista. Tem complexidade O(1).
22  */
23 int equal(Iterator it0, Iterator it1);
```

Há vários outros aspectos na definição do tipo abstrato de dados *List* que requerem maior detalhamento. Inicialmente, note que a operação *getEnd()* retorna um iterador que "aponta para o elemento após o último elemento" da lista. Este requisito complica um pouco a implementação mas permite que se forneçam implementações como a que se segue:

Exemplo 4.3 Uma implementação para a operação *getEnd()*

```
/**
 * Retorna um iterador que aponta para o elemento após o último
 * elemento da lisa. Tem complexidade O(n).
 */
Iterator getEnd(List list) {
  Iterator it = getBegin(list);
  while(it != NULL) {
    moveNext(it);
  }
  return it;
}
```

Em segundo lugar, cada operação apresenta sua complexidade no pior caso. Qualquer implementação do tipo abstrato *List* deve garantir ser capaz de implementar estas operações pelo menos tão rápido quanto o especificado na interface. Porém, a maioria das implementações será bem mais eficiente que o especificado. Por exemplo, em uma implementação de lista utilizando a cadeia de nós a operação `insertAfter()` terá complexidade *O(1)*.

Finalmente, algumas operações especificam que possuem uma implementação *default*. Isso significa que elas podem ser implementadas em termos de outras operações mais primitivas. Elas são incluídas no tipo abstrato de dados de forma que certas implementações possam fornecer versões mais eficientes. Por exemplo, a implementação *default* da operação `getNth()` tem complexidade *O(n)* porque ela precisa percorrer todos os elementos da lista até alcançar o *enésimo*. Porém, em uma implementação de lista baseada em arranjos, essa operação tem complexidade *O(1)* utilizando a operação `getElement()`. A seguir são apresentas as implementações *default* para as operações `isEmpty()`, `getSize()`, `getNth()` e `setNth()`.

Implementações default para as operações isEmpty(), getSize(), getNth() e setNth() do tipo abstrato de dados List

```
/**
 * Retorna 1 se a lista estiver vazia. Tem complexidade O(1).
 */
int isEmpty(List list) {
  return getBegin(list) == getEnd(list);
}

/**
 * Retorna o número de elementos da lista. Tem complexidade O(n).
 */
int getSize(List list) {
  Iterator it = getBegin(list);
  int size = 0;
  while( it != NULL) {
    moveNext(it);
    size++;
  }
  return size;
}

/**
```

```
22  * Retorna o e-nésimo elemento da lista, contando a partir do 0.
23  * Tem complexidade O(n).
24  */
25 ItemType getNth(List list, int n) {
26   Iterator it = getBegin(list);
27   int i;
28   for( i = 0; i < n; i++) {
29     moveNext(it);
30   }
31   return getValue(it);
32 }
33
34 /**
35  * Altera o valor do e-nésimo elemento da lista, contando a partir
36  * do 0. Tem complexidade O(n).
37  */
38 void setNth(List list, int n, ItemType newItem) {
39   Iterator it = getBegin(list);
40   int i;
41   for( it = 0; i < n; i++) {
42     moveNext(it);
43   }
44   setValue(it,newItem);
45 }
```

4.1 Implementações

Para podermos utilizar o tipo abstrato *List* é preciso fornecer uma implementação concreta de sua interface. Existem duas estruturas de dados que naturalmente implementam o tipo abstrato *List*: a cadeia de nós apresentada no capítulo 1, comumente denominada *Lista Encadeada*, e uma extensão dos arranjos apresentados no capítulo 2, denominado *Vector*, que possui a capacidade de se redimensionar para acomodar mais elementos.

Nesta seção apresentaremos várias implementações concretas para o tipo abstrato de dados *List*.

4.1.1 Lista Encadeada

O código a seguir apresenta a definição do tipo abstrato de dados *LinkedList*. A definição da lista é baseada na estrutura básica de *Nó* que apresentamos no Capítulo 1. A definição de *LinkedList* apresenta uma operação adicional, `createLinkedList()`, utilizada para criar uma nova lista.

Inicialmente a lista esta vazia, portanto, a criação consiste simplesmente em criar um *Nó* para representar a cabeça da lista e atribuir o valor `NULL` a este nó.

Nesta implementação de lista, o iterador "um após o final da lista"é simplesmente um nó com valor `NULL`. Para entender o motivo, pense no que aconteceria se você tivesse um iterador para o último elemento da lista e invocasse a operação `moveNext()`.

A seguir apresentamos o código com todas as operações de uma *LinkedList*.

Exemplo 4.4 Implementação das operações de uma *LinkedList*

Código fonte code/capitulo-04/linkedlist.c

```c
#include "../capitulo-01/node.c"
#include "listiterator.c"
#include <stdlib.h>

/** Definição dos tipos LinkedList e LL_Iterator **/

#ifndef ELEMENT_T
#define ELEMENT_T 1
/**
 * Tipo básico de dado a ser armazenado na estrutura.
 */
typedef int Element;
#endif

#ifndef LL_ITEMTYPE
#define LL_ITEMTYPE 1
/**
 * Define o tipo de elemento a ser armazenado na lista encadeada
 */
typedef Node LL_ItemType;
#endif

#ifndef LINKEDLIST
#define LINKEDLIST 1
/**
 *  Define o tipo abstrato LinkedList
 */
typedef LL_ItemType* LinkedList;
#endif

#ifndef LL_ITERATOR
#define LL_ITERATOR 1
/**
 * Define o tipo abstrato LL_Iterator
 */
typedef LinkedList LL_Iterator;
#endif

/**
 * Retorna um iterador de lista (será definido em breve) que aponta
 * para o primeiro elemento da lista. Tem complexidade O(1).
 */
LL_Iterator getBegin(LinkedList list) {
  LL_Iterator it = list;
  return it;
}

/**
 * Retorna um iterador que aponta para o elemento após o último
```

```
 * elemento da lista. Tem complexidade O(1).
 */
LL_Iterator getEnd(LinkedList list) {
  return NULL;
}

/**
 * Adiciona um elemento no início da lista.
 * Tem complexidade O(1).
 */
LinkedList prepend(LinkedList list, Element newElement) {
  return makeNode(newElement,list);
}

/**
 * Adiciona um elemento imediatamente após o elemento apontado por it.
 * Tem complexidade O(1).
 */
void insertAfter(LL_Iterator it, Element newElement) {
  LL_ItemType* newItem = makeNode(newElement,it->next);
  it->next = newItem;
}

/**
 * Remove o primeiro elemento da lista.
 * Tem complexidade O(1).
 */
LinkedList removeFirst(LinkedList list) {
  LinkedList list0 = list;
  // apontador de cabeça da lista apontará para o próximo elemento
  list = list->next;
  free(list0); // libera primeiro elemento
  return list;
}

/**
 * Remove o elemento imediatamente após o elemento apontado por it.
 * Tem complexidade O(n).
 */
void removeAfter(LL_Iterator it) {
  LinkedList next_of_iterator = it->next;
  it->next = it->next->next; // liga anterior com o próximo do próximo
  free(next_of_iterator); // libera quem era o próximo de it
}

/**
 * Retorna 1 se a lista estiver vazia. Tem complexidade O(n).
 */
int isEmpty(LinkedList list) {
  return getBegin(list) == getEnd(list);
}

/**
```

```
103  * Retorna o número de elementos da lista. Tem complexidade O(n).
104  */
105 int getSize(LinkedList list) {
106   LL_Iterator it = getBegin(list);
107   int size = 0;
108   while( it != NULL) {
109     moveNext(it);
110     size++;
111   }
112   return size;
113 }
114 
115 /**
116  * Retorna o e-nésimo elemento da lista, contando a partir do 0.
117  * Tem complexidade O(n).
118  */
119 Element getNth(LinkedList list, int n) {
120   LL_Iterator it = getBegin(list);
121   int i;
122   for( it = 0; i < n; i++) {
123     moveNext(it);
124   }
125   return getValue(it);
126 }
127 
128 /**
129  * Altera o valor do e-nésimo elemento da lista, contando a partir
130  * do 0. Tem complexidade O(n).
131  */
132 void setNth(LinkedList list, int n, Element newElement) {
133   LL_Iterator it = getBegin(list);
134   int i;
135   for( it = 0; i < n; i++) {
136     moveNext(it);
137   }
138   setValue(it,newElement);
139 }
```

Nota

Observem a presença do prefixo *LL* em alguns tipos de dados, particularmente, no *LL_Iterator*. Isso é necessário para evitar conflitos com outras implementações de iteradores que apresentaremos ao longo do curso.

4.1.1.1 Inserindo na lista

Observemos agora o funcionamento das duas operações que inserem elementos na lista (ver Figura 4.1 [34]). A primeira delas, a operação `prepend()`, insere um novo item no começo da lista. A implementação apresentada realiza essa operação fazendo o apontador de próximo, do novo item, apontar para a *cabeça* da lista e fazendo a *cabeça* da lista ser o novo item inserido.

A segunda operação para inserção de elementos na lista, `insertAfter()`, insere um novo item após o item apontado por um iterador.

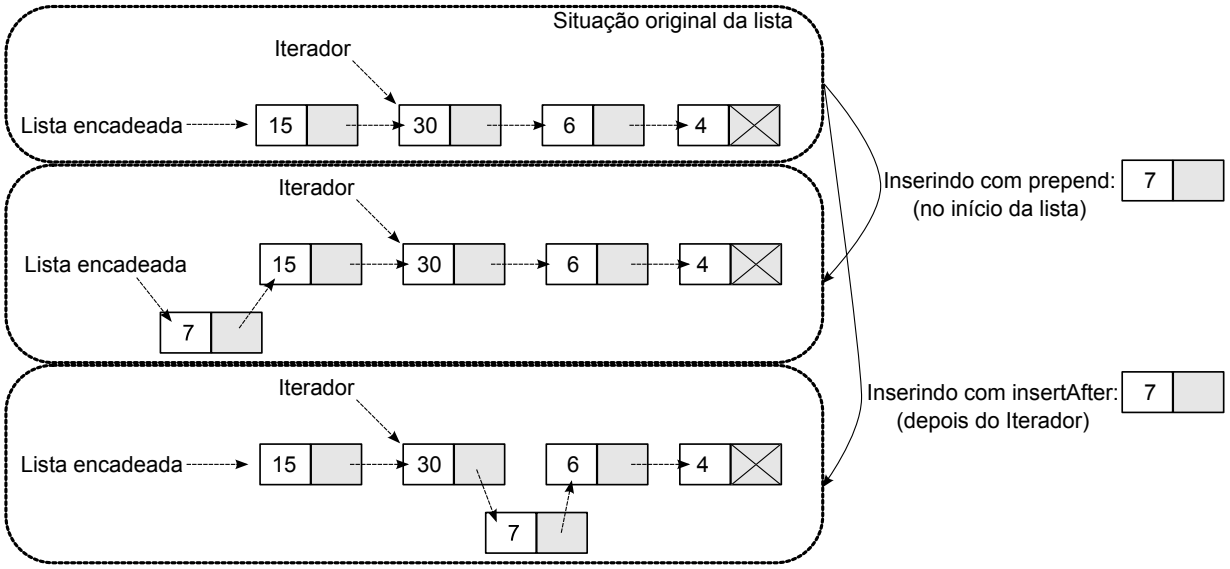


Figura 4.1: Ilustração das duas operações de inserção: `prepend` e `insertAfter`

4.1.1.2 Removendo da Lista

As operações de remoção `removeFirst` e `removeAfter` funcionam de maneira análoga. No caso da operação `removeFirst`, o apontador de cabeça da lista é alterado para apontar para o próximo elemento da lista, fazendo com que não haja mais qualquer referência para o antigo primeiro elemento. Após essa alteração, é preciso liberar a memória alocada previamente para armazenar o item. Isso é feito utilizando a função `free()`.

No caso da operação `removeFirst`, a operação altera o *apontador de próximo* do item referenciado pelo iterador para que aponte para um elemento mais adiante (`it->next =it->next->next`). Em seguida, é necessário liberar o elemento, como no caso anterior.

Atenção

Não liberar memória utilizada para armazenar dados não mais utilizados é um erro comum e que pode levar a um vazamento de memória (*memory leak*). Isso pode levar a uma falha na execução do programa por esgotamento da memória livre disponível. Algumas linguagem de programação, como *Java* por exemplo, implementam um mecanismo automático para liberação de memória sempre que um objeto passa a não ser mais referenciado. Esse mecanismo é conhecido como coletor de lixo ou *garbage collector*.

Em nossa implementação de *LinkedList* o iterador nada mais é do que um apontador para uma estrutura de *Nó*. A implementação de suas operações faz uso das propriedades da estrutura básica que introduzimos no capítulo 1, conforme a listagem de código abaixo.

Exemplo 4.5 Implementação de um iterador para uma *LinkedList*

Código fonte code/capitulo-04/listiterator.c

```c
#include "../capitulo-01/node.h"

/** Definição dos tipos LinkedList e LL_Iterator **/

#ifndef ELEMENT_T
#define ELEMENT_T 1
/**
 * Tipo básico de dado a ser armazenado na estrutura.
 */
typedef int Element;
#endif

#ifndef LL_ITEMTYPE
#define LL_ITEMTYPE 1
/**
 * Define o tipo de elemento a ser armazenado na lista encadeada
 */
typedef Node LL_ItemType;
#endif

#ifndef LINKEDLIST
#define LINKEDLIST 1
/**
 * Define o tipo abstrato LinkedList
 */
typedef LL_ItemType* LinkedList;
#endif

#ifndef LL_ITERATOR
#define LL_ITERATOR 1
/**
 * Define o tipo abstrato LL_Iterator
 */
typedef LinkedList LL_Iterator;
#endif

/**
 * Retorna o elemento da lista apontado por este iterador.
 * Tem complexidade O(1).
 */
Element getElement(LL_Iterator it) {
  return getValue(it);
}

/**
 * Altera o valor do elemento apontado por este iterador.
 * Tem complexidade O(1).
 */
void setElement(LL_Iterator it, Element newElement) {
  setValue(it, newElement);
}

```

```
53 /**
54  * Faz o iterador apontar para o próximo elemento da lista.
55  * Tem complexidade O(1).
56  */
57 LL_Iterator moveNext(LL_Iterator it) {
58   return getNext(it);
59 }
60
61 /**
62  * Retorna 1 se os dois iteradores apontam para o mesmo elemento
63  * da lista. Tem complexidade O(1).
64  */
65 int equal(LL_Iterator it0, LL_Iterator it1) {
66   return it0 == it1;
67 }
```

Nota

A operação `equal()` faz uma comparação de apontadores para saber se dois iteradores são iguais. Isso significa comparar os endereços de memória das localidades referenciadas pelos dois iteradores.

4.1.1.3 Encontrando o maior elemento

Voltemos agora para o exemplo que vimos no capítulo 1 sobre como encontrar o maior elemento de um conjunto de valores lidos da entrada padrão e a média dos divisores deste elemento. Vamos alterar a nossa implementação anterior para utilizar a nossa nova estrutura de dados *LinkedList*.

Utilizando uma LinkedList para encontrar o maior elemento de um conjunto de valores e a média dos divisores deste elemento

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <limits.h>
4  #include "linkedlist.c"
5
6  #define INF 2147483647 //Maior valor que um inteiro pode assumir.
7
8  int main() {
9    int count = 0;
10   int total = 0;
11   int largest = INT_MIN;
12   Element i;
13
14   LinkedList list = NULL;
15   LL_Iterator it;
16
17   printf("Digite números para adicionar à lista (CTRL+D para sair): ") ↩
       ;
18
19   while( (scanf("%d",&i) == 1 ) ) {
```

```
20      list = prepend(list, i);
21      if( i > largest) largest = i;
22    }
23    printf( "Valor máximo: %d\n", largest);
24
25    it = getBegin(list);
26    while( it != NULL ) {
27      i = getElement(it);
28      //if( largest % i == 0 ) {
29        total += i;
30        count++;
31      //}
32      it = moveNext(it);
33    }
34    if(count != 0)
35      printf( "Média: %d\n", (total/count));
36  }
```

4.1.1.4 Exercícios

1. Escreva um programa que recebe um conjunto de números inteiros de tamanho desconhecido da entrada padrão e imprime os números lidos na ordem inversa da leitura.

2. Implemente uma função que receba como parâmetro uma lista encadeada e um valor inteiro `n` e divida a lista em duas, de tal forma que a segunda lista comece no primeiro nó logo após a primeira ocorrência de `n` na lista original.

3. Implemente uma função que recebe duas listas encadeada `lista1` e `lista2` como parâmetros e retorna uma terceira lista `lista3` obtida pela concatenação de `lista2` ao final de `lista1`.

4. Implemente uma função que, dados uma lista encadeada e um inteiro não negativo `n`, remova da lista seus `n` primeiros nós e retorne a lista resultante. Caso `n` seja maior que o comprimento da lista, todos os seus elementos devem ser removidos e o resultado da função deve ser uma lista vazia.

5. Escreva um programa que recebe um conjunto de números inteiros de tamanho desconhecido da entrada padrão e imprime os números lidos em ordem crescente.

6. Modifique a estrutura *LinkedList* apresentada para que ela se comporte como uma **lista circular**, ou seja, o último elemento da lista deve ter um apontador para o primeiro elemento.

4.1.2 Vetor

A seção anterior apresentou a estrutura de lista denominado *LinkedList*, que foi construída tomando por base a estrutura de nó que foi introduzida no capítulo 1 deste livro. A principal característica desta estrutura é sua capacidade de aumentar dinamicamente seu tamanho, a medida em que novos elementos vão sendo inseridos na lista. Nesta seção, estudaremos outro tipo de lista, denominada *Vector*, que não é baseada na estrutura de nó que vimos anteriormente.

Um *Vector* (vetor) é uma estrutura de lista que utiliza, em vez da estrutura encadeada vista anteriormente, um **arranjo** para armazenar seus elementos.

Antes de apresentarmos a implementação de um vetor, vejamos primeiro como funciona o seu iterador. Isso deixará a implementação do vetor adiante mais clara.

Exemplo 4.6 Implementação de um iterador para um *Vector*

Código fonte code/capitulo-04/viterator.c

```
1  /** Definição dos tipos Vector e V_Iterator **/
2
3  #ifndef ELEMENT_T
4  #define ELEMENT_T 1
5  /**
6   * Tipo básico de dado a ser armazenado na estrutura.
7   */
8  typedef int Element;
9  #endif
10
11 #ifndef VECTOR
12 #define VECTOR 1
13 /**
14  * Define o tipo abstrato Vector
15  */
16 typedef struct {
17   Element *data;    //Dados armazenados no vector
18   int     size;     //Quantidade de dados armazenados
19   int     capacity; //Capacidade máxima do vector
20 } Vector;
21 #endif
22
23 #ifndef V_ITERATOR
24 #define V_ITERATOR 1
25 /**
26  * Define o tipo abstrato V_Iterator
27  */
28 typedef struct {
29   Vector *vector; //Referência ao vector associado
30   int    index;   //Índice para acessos
31 } V_Iterator;
32 #endif
33
34 /**
35  * Retorna o elemento do vector apontado por este iterador.
36  * Tem complexidade O(1).
37  */
38 Element getElement(V_Iterator *it) {
39   return it->vector->data[it->index];
40 }
41
42 /**
43  * Altera o valor do elemento apontado por este iterador.
44  * Tem complexidade O(1).
45  */
```

```
46 void setElement(V_Iterator *it, Element newElement) {
47   it->vector->data[it->index] = newElement;
48 }
49
50 /**
51  * Faz o iterador apontar para o próximo elemento da lista.
52  * Tem complexidade O(1).
53  */
54 V_Iterator *moveNext(V_Iterator *it) {
55   it->index++;
56   return it;
57 }
58
59 /**
60  * Retorna 1 se os dois iteradores apontam para o mesmo elemento
61  * do vector. Tem complexidade O(1).
62  */
63 int equal(V_Iterator *it0, V_Iterator *it1) {
64   return (it0->index  == it1->index) && (it0->vector == it1->vector);
65 }
```

A definição para a estrutura de um *Vector* é reproduzida abaixo. Uma vez que é ineficiente sempre manter o arranjo utilizado para armazenar com o tamanho exato (pense em quantas vezes seria preciso redimensionar o arranjo para acomodar novos elementos), armazenamos na estrutura dados sobre o tamanho do arranjo (campo `size`), indicando a quantidade de elementos atualmente armazenados, e sobre a capacidade (campo `capacity`), que representa o número total de espaços no arranjo. Os índices válidos para acesso aos elementos do *Vector* estarão sempre no intervalo `0` até `capacity - 1` porém só representam elementos válidos os encontrados entre 0 e `size -1`.

```
1 typedef struct {
2   Element *data; //Dados armazenados no vector
3   int size;      //Quantidade de dados armazenados
4   int capacity;  //Capacidade máxima do vector
5 } Vector;
```

Um iterador para um *Vector* é uma estrutura que encapsula um *Vector* e um *índice de acesso*. Para evitarmos manter muitas cópias dos dados na memória principal o, *V_Iterator* armazena apenas uma referência para o *Vector* ao qual está relacionado, e esta referência é utilizada para acesso aos dados e campos do *Vector*.

```
1 typedef struct {
2   Vector *vector; //Referência ao vector associado
3   int index;      //Índice para acessos
4 } V_Iterator;
```

Os métodos de acesso do *V_Iterator* (*getElement* e *setElement*) utilizam esta referência, ao *Vector* associado, para poder acessar ou modificar os valores de elementos armazenados no *Vector*.

É importante notar o funcionamento do método `equal`, que compara dois *V_Iterators* e retorna 1 se eles forem iguais. Dois *V_Iterators* são iguais se eles estão associados a um mesmo *Vector* e referenciam o mesmo elemento deste *Vector*.

```
1 /**
```

```
 * Retorna 1 se os dois iteradores apontam para o mesmo elemento
 * do vector. Tem complexidade O(1).
 */
int equal(V_Iterator *it0, V_Iterator *it1) {
  return (it0->index  == it1->index) && (it0->vector == it1->vector);
}
```

Uma vez apresentadas as estruturas de *Vector* e *V_Iterator* e a implementação das operações que nos permitem iterar[1] sobre os elementos armazenados em um *Vector* podemos passar a estudar a implementação do *Vector* propriamente dita, ilustrada a seguir.

Exemplo 4.7 Implementação de um *Vector*

Código fonte code/capitulo-04/vector.c

```
#include "viterator.c"
#include <stdlib.h>
#include <math.h>

/**
 * Método auxiliar para a criação de um vector que possui
 * capacidade inicial igual a 16.
 */
Vector createVector() {
  Vector vector;
  vector.data = (Element *) malloc(16 * sizeof(Element));
  vector.size = 0;
  vector.capacity = 16;
  return vector;
}

/**
 * Retorna o número de elementos do vector. Tem complexidade O(1).
 */
int getSize(Vector *vector) {
  return vector->size;
}

/**
 * Retorna 1 se o vector estiver vazio. Tem complexidade O(1).
 */
int isEmpty(Vector *vector) {
  return getSize(vector) == 0;
}

/**
 * Retorna o e-nésimo elemento do vector, contando a partir do 0.
 * Tem complexidade O(1).
 */
Element getNth(Vector *vector, int n) {
  return vector->data[n];
}
```

[1] Não confundir **iterar** com **interação**. **Iterar** está relacionado com repetir, enquanto **interação** está relacionado com interagir.

```
38
39 /**
40  * Altera o valor do e-nésimo elemento do vector, contando a partir
41  * do 0. Tem complexidade O(1).
42  */
43 void setNth(Vector *vector, int n, Element newElement) {
44   vector->data[n] = newElement;
45 }
46
47 /**
48  * Retorna um iterador de vector que aponta para o primeiro elemento
49  * do vector. Tem complexidade O(1).
50  */
51 V_Iterator getBegin(Vector *vector) {
52   V_Iterator it;
53   it.vector = vector;
54   it.index = 0;
55   return it;
56 }
57
58 /**
59  * Retorna um iterador que aponta para o elemento após o último
60  * elemento do vector. Tem complexidade O(1).
61  */
62 V_Iterator getEnd(Vector *vector) {
63   V_Iterator it;
64   it.vector = vector;
65   it.index = vector->size;
66   return it;
67 }
68
69 /**
70  * Método auxiliar para garantir que o vector sempre terá
71  * capacidade suficiente para acomodar a inserção de um novo elemento
72  */
73 void ensureCapacity(Vector *vector, int newCapacity) {
74
75   // Se a capacidade atual for suficiente para acomodar o novo
76   // elemento não fazemos nada.
77   if(vector->capacity >= newCapacity) return;
78
79   //Se a capacidade não for suficiente, devemos criar um novo arranjo.
80   //O novo arranjo terá, no mínimo, o dobro da capacidade anterior.
81   int cap = vector->capacity * 2;
82   if(newCapacity > cap) cap = newCapacity;
83
84   Element *newData = (Element *) malloc(cap * sizeof(Element));
85
86   for( int i = 0; i < cap; i++)
87     newData[i] = 0;
88
89   //E copiar os elementos do arranjo antigo para o novo.
90   for( int i = 0; i < vector->size; i++ )
```

```
91      newData[i] = vector->data[i];
92
93    //Liberar a memória utilizada para armazenar o antigo arranjo.
94    free(vector->data);
95
96    //E alterar o vector para utilizar o novo arranjo.
97    vector->data = newData;
98
99    //por fim, atualizamos a nova capacidade do vector.
100   vector->capacity = cap;
101 }
102
103 /**
104  * Adiciona um elemento imediatamente após o elemento apontado por it.
105  * Tem complexidade O(n).
106  */
107 void insertAfter(V_Iterator *it, Element newElement) {
108   // Garantindo que o vector pode receber um novo elemento.
109   ensureCapacity(it->vector, it->vector->size + 1);
110
111   // deslocando todos os elementos entre as posições index + 1 e
112   // size uma casa para a direita.
113   for(int i = it->vector->size; i > it->index + 1; i-- )
114     it->vector->data[i] = it->vector->data[i - 1];
115
116   // Inseção do novo elemento na posição seguinte a apontada
117   // pelo Iterador
118   it->vector->data[it->index+1] = newElement;
119
120   //Atualizando o tamanho.
121   it->vector->size++;
122 }
123
124 /**
125  * Remove o elemento imediatamente após o elemento apontado por it.
126  * Tem complexidade O(n).
127  */
128 void removeAfter(V_Iterator *it) {
129   // Deslocando todos os elementos entre as posições index + 1 e
130   // size uma casa para a esquerda
131   for(int i = it->index + 1; i < (it->vector->size - 1); i++ )
132     it->vector->data[i] = it->vector->data[i + 1];
133
134   // Atualizando o tamanho.
135   it->vector->size--;
136 }
137
138 /**
139  * Adiciona um elemento no início do vector.
140  * Tem complexidade O(n).
141  */
142 void prepend(Vector *vector, Element newElement){
143   V_Iterator it = getBegin(vector); // it aponta para o primeiro
```

```
144   it.index--; //Faz o iterador apontar para o elemento localizado
145               // antes do primeiro elemento do vector
146   insertAfter(&it, newElement); // insere no início (depois de antes)
147 }
148
149 /**
150  * Remove o primeiro elemento do vector.
151  * Tem complexidade O(n).
152  */
153 void removeFirst(Vector *vector) {
154   V_Iterator it = getBegin(vector); // it aponta para o primeiro
155   it.index--;  //Aponta para o elemento antes do primeiro elemento
156   removeAfter(&it); // remove o primeiro (depois de antes do primeiro)
157 }
```

Na implementação de *Vector* apresentada temos quatro grupos de funções. O primeiro grupo, formado pelas funções `getSize`, `isEmpty`, `getNth` e `setNth` representam as funções mais simples, que são utilizadas para checar a quantidade de elementos em um *Vector* e para consultar ou alterar valores de elementos armazenados no *Vector* utilizando um índice de acesso. Também podemos incluir neste grupo inicial a função `createVector` que, como o próprio nome indica, serve para criar um nova instância de um *Vector*. Cada nova instância é criada com uma capacidade inicial para acomodar 16 elementos. Este valor foi escolhido de forma arbitrária e não limita de forma alguma a utilidade ou funcionamento de um *Vector*.

O segundo grupo, formado pelas funções `getBegin` e `getEnd`, oferece operações para criar um *V_Iterator* associado a um determinado *Vector*.

Vejamos a definição da função `getBegin` abaixo.

```
1  /**
2   * Retorna um iterador de vector que aponta para o primeiro
3   * elemento do vector. Tem complexidade O(1).
4   */
5  V_Iterator getBegin(Vector *vector) {
6    V_Iterator it;
7    it.vector = vector;
8    it.index = 0;
9    return it;
10 }
```

Esta função cria um novo *V_Iterator* que armazena uma referência para um *Vector* e que tem seu índice de acesso iniciado com 0, ou seja, o índice que aponta para o primeiro elemento do *Vector*. A sua função irmã, `getEnd`, funciona de forma análoga, porém, o índice de acesso é iniciado com o valor `size`, ou seja, o valor do índice do elemento que se encontra **após o último elemento** armazenado no *Vector*, conforme a especificação do tipo abstrato de dados *List*.

O terceiro grupo, formado pelas funções `insertAfter`, `removeAfter`, `prepend` e `removeFirst`, é responsável pela inserção e remoção de elementos do *Vector*. Todas estas funções têm complexidade *O*(n) porque podem exigir que a movimentação de todos os `n` elementos de um *Vector* para acomodar a inserção de um novo elemento ou para eliminar um espaço em branco deixado pela remoção de um elemento. Também faz parte deste grupo a função auxiliar `ensureCapacity`, que é utilizada para garantir que o *Vector* terá capacidade suficiente para acomodar a inserção de um novo elemento.

Vejamos então, inicialmente, o funcionamento da função auxiliar `ensureCapacity`, cuja definição é apresentada novamente abaixo:

```
/**
 * Método auxiliar para garantir que o vector sempre terá
 * capacidade suficiente para acomodar a inserção de um novo elemento
 */
void ensureCapacity(Vector *vector, int newCapacity) {

  // Se a capacidade atual for suficiente para acomodar o novo
  // elemento não fazemos nada.
  if(vector->capacity >= newCapacity) return;

  //Se a capacidade não for suficiente, devemos criar um novo arranjo.
  //O novo arranjo terá, no mínimo, o dobro da capacidade anterior.
  int cap = vector->capacity * 2;
  if(newCapacity > cap) cap = newCapacity;

  Element *newData = (Element *) malloc(cap * sizeof(Element));

  for( int i = 0; i < cap; i++)
    newData[i] = 0;

  //E copiar os elementos do arranjo antigo para o novo.
  for( int i = 0; i < vector->size; i++ )
    newData[i] = vector->data[i];

  //Liberar a memória utilizada para armazenar o antigo arranjo.
  free(vector->data);

  //E alterar o vector para utilizar o novo arranjo.
  vector->data = newData;

  //por fim, atualizamos a nova capacidade do vector.
  vector->capacity = cap;
}
```

O que esta função faz é garantir que o *Vector* terá capacidade suficiente para acomodar a inserção de um novo elemento. Para isso, ela primeiro checa se a capacidade atual é suficiente para suportar uma inserção. Se esse for o caso, nada preciso ser feito. Porém, se a capacidade não for suficiente é preciso aumentar o tamanho do *Vector*. Isso é feito criando um novo arranjo para acomodar os dados já armazenados no *Vector* e que tenha capacidade para receber mais elementos. Para isso, criamos um novo arranjo com capacidade no mínimo duas vezes maior que a atual e copiamos todos os elementos atualmente armazenados no *Vector* para este novo arranjo. Por essa razão, esta operação acaba tendo complexidade $O(n)$.

Importante

Uma implementação mais inocente desta função poderia simplesmente criar um novo arranjo com capacidade exatamente igual a solicitada. Para entender porque esta opção seria ineficiente pense sobre o que aconteceria se começassem a inserir elementos um a em um laço. Uma vez que excedamos a capacidade inicial do *Vector* cada novo elemento inserido iria exigir que todos os elementos do arranjo existente fossem copiados. É por essa razão que a implementação apresentada, no mínimo, dobra a capacidade do arranjo utilizado quando ele precisa crescer.

Podemos agora passar para a implementação das operações `insertAfter` e `removeAfter`. Estas funções, como veremos a seguir, são as únicas que efetivamente alteram o tamanho de um *Vector*.

A função `insertAfter`, re-apresentada abaixo, é responsável por inserir um novo elemento no *Vector* na posição seguinte àquela representada pelo índice do *V_Iterator*, ou sela ela insere um elemento na posição `index + 1`

```
/**
 * Adiciona um elemento imediatamente após o elemento apontado por it.
 * Tem complexidade O(n).
 */
void insertAfter(V_Iterator *it, Element newElement) {
  // Garantindo que o vector pode receber um novo elemento.
  ensureCapacity(it->vector, it->vector->size + 1);

  // deslocando todos os elementos entre as posições index + 1 e
  // size uma casa para a direita.
  for(int i = it->vector->size; i > it->index + 1; i-- )
    it->vector->data[i] = it->vector->data[i - 1];

  // Inseção do novo elemento na posição seguinte a apontada
  // pelo Iterador
  it->vector->data[it->index+1] = newElement;

  //Atualizando o tamanho.
  it->vector->size++;
}
```

Para fazer essa inserção a função precisa primeiro abrir o espaço necessário. Para isso, deve deslocar todos os elementos entre a posição `index + 1` e `size` uma casa para a direita, deixando a posição `index + 1` pronta para receber um novo elemento. Depois que os elementos são deslocados, basta alterar o valor do elemento apropriado no arranjo.

Complexidade da função insertAfter no Vetor

Portanto, no pior caso, representando pela inserção na primeira posição, a inserção de um elemento no *Vector* tem complexidade *O*(n), pois haverá uma deslocamento de `n` elementos.

A função `removeAfter`, re-apresentada abaixo, funciona de forma análoga. Só que neste caso, os elementos devem ser deslocados para a esquerda para sobre-escreverem o elemento removido. Em seguida, basta atualizar o tamanho do *Vector* para refletir a remoção de um elemento.

```c
/**
 * Remove o elemento imediatamente após o elemento apontado por it.
 * Tem complexidade O(n).
 */
void removeAfter(V_Iterator *it) {
  // Deslocando todos os elementos entre as posições index + 1 e
  // size uma casa para a esquerda
  for(int i = it->index + 1; i < (it->vector->size - 1); i++ )
    it->vector->data[i] = it->vector->data[i + 1];

  // Atualizando o tamanho.
  it->vector->size--;
}
```

Finalmente, nas funções `prepend` e `removeFirst` utilizamos um artifício para evitar duplicação de código. A função `prepend` é utilizada para inserir um elemento no começo do *Vector*, ou seja, antes do primeiro elemento. Para fazer isso, nós criamos um *V_Iterator* apontando para o primeiro elemento do *Vector* utilizando a função `getBegin` e em seguida decrementamos o seu índice. Com isso, temos um *V_Iterator* que "aponta" para a posição imediatamente antes do início do *Vector*. Com isso, basta utilizarmos a operação `insertAfter` para inserirmos um elemento após esta posição, no começo do *Vector* como queríamos.

O mesmo artifício é utilizado na implementação da operação `removeFirst`, que remove o elemento localizado após o elemento que se encontra uma posição antes do início do *Vector*.

4.1.2.1 Encontrando o maior elemento utilizando *Vector*

Finalizamos esta seção apresentando uma nova versão da nossa solução para encontrar o maior elemento e a média dos seus divisores utilizando a nossa implementação de *Vector*.

Exemplo 4.8 Utilizando um *Vector* para encontrar o maior elemento de um conjunto de valores e a média dos divisores deste elemento

Código fonte code/capitulo-04/maxemedia4.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "vector.c"

int main() {
  int count = 0;
  int total = 0;
  int largest = INT_MIN;
  Element i;

  Vector vector = createVector();
  V_Iterator it = getEnd(&vector);

  while( (scanf("%d",&i) == 1 ) ) {
    if(vector.size == 0 )
      prepend(&vector,i);
    else {
```

```
19        insertAfter(&it,i);
20        moveNext(&it);
21      }
22      if( i > largest) largest = i;
23    }
24
25    printf( "Valor máximo: %d\n", largest);
26    it = getBegin(&vector);
27
28    while( it.index < vector.size ) {
29      i = getElement(&it);
30  //    if( largest % i == 0 ) {
31        total += i;
32        count++;
33  //    }
34      moveNext(&it);
35    }
36    if(count != 0)
37      printf( "Média: %d\n", (total/count));
38  }
```

4.1.2.2 Exercícios

1. Implemente uma função que recebe dois *vectors* `v1` e `v2` como parâmetros e retorna um terceiro *vector* obtido pela concatenação de `v2` ao final de `v1`.

2. Implemente uma função `append` que adiciona para adicionar um elemento no **final** de um *Vector*.

3. Implemente uma função `reverse` para inverter a ordem dos elementos em um *Vector*.

4.2 Recapitulando

Para pode escolher a estrutura de lista correta para resolver um determinado problema precisamos ter alguma idéia sobre o que vamos precisar fazer com os dados.

- Nosso programa jamais precisará lidar com mais do que 100 itens de dados ou é possível que tenhamos que manipular milhões de registros?
- Como os dados serão acessados? Sempre em ordem cronológica? Sempre ordenados por nome? Sem ordem?
- Sempre iremos adicionar/remover elementos no começo ou no final da estrutura? Ou precisaremos fazer inserções ou remoções no meio da estrutura?

Precisamos então tentar encontrar um equilíbrio entre os vários requisitos. Se precisarmos acessar frequentemente os dados de 3 formas diferentes, escolha uma estrutura de dados que permite estes 3 tipos de acesso de forma não muito lenta. Não escolha uma estrutura que será insuportavelmente lenta para um dos tipos de acesso independente de quão rápida ela será para os outros dois.

Geralmente, a solução mais simples para um problema é utilizar uma estrutura de arranjo unidimensional.

Na maioria das vezes, as vantagens em se utilizar uma *LinkedList* são as desvantagens em se utilizar um *Vector* e vice-versa.

- Vantagens em se utilizar um *Vector*

    1. Índices: acesso rápido a qualquer elemento da estrutura. Na *LinkedList* é preciso percorrer toda a lista para se acessar um determinado elemento.
    2. Acesso mais rápido: de forma geral, é mais rápido acessar um elemento em um arranjo do que em uma *LinkedList*.

- Vantagens em se utilizar uma *LinkedList*

    1. Redimensionamento: a lista pode crescer dinamicamente sempre que seja precisa copiar elementos.
    2. Inserção/Remoção: É simples inserir ou remover elementos no meio de uma lista.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?

Acesse https://github.com/edusantana/estruturas-de-dados-livro/issues/new para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**capitulo-04**) e a versão do livro (**v1.0.0**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 5

Pilhas e Filas

Sumário

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender o funcionamento de pilhas e filas
- Ser capaz de implementar operações para manipulação de elementos de uma pilha e de uma fila
- Ser capaz de decidir quando cada uma destas estruturas deve ser utilizada na resolução de problemas.

Neste capítulo estudaremos duas estruturas de dados que possuem regras específicas para inserção e remoção de elementos: **Pilhas** e **Filas**. Estas estruturas são fundamentais para o desenvolvimento de uma vasta gama de algoritmos em diversas áreas de computação.

Seu funcionamento é baseado em princípios derivados do mundo real. Uma pilha representa um conjunto de elementos que são empilhados um acima do outro, como uma pilha de cartas de baralho, uma pilha de pratos, uma pilha de moedas, uma pilha de panquecas, etc. Já uma fila representa um conjunto de elementos que são enfileirados como pessoas em uma fila de cinema ou requisições em uma fila de escalonamento.

Compreender e ser capaz de utilizar corretamente estas duas estruturas de dados representam um requisito básico para a resolução de um grande número de problemas como veremos mais adiante.

5.1 Pilhas

Uma pilha (*stack*, em inglês) é uma estrutura de dados básica que pode ser compreendida logicamente como uma estrutura linear (uma única dimensão) representada por uma pilha física de objetos na qual a inserção e remoção de elementos *sempre ocorre no topo da pilha*. O conceito básico pode ser ilustrado se pensarmos em nosso conjunto de dados como uma pilha de pratos ou uma pilha de livros, da qual só se pode remover ou adicionar no topo da pilha. Este tipo de estrutura é largamente utilizada em programação e em diversas áreas da computação.

A implementação básica de uma pilha é denominada LIFO (*Last In First Out*) que significa 'o último a entrar é o primeiro a sair', o que reflete a forma como os dados são acessados.

Existem duas operações fundamentais para a manipulação de elementos de uma pilha. São elas:

- Inserir um elemento no topo da pilha (`push`)
- Remover um elemento do topo da pilha (`pop`)

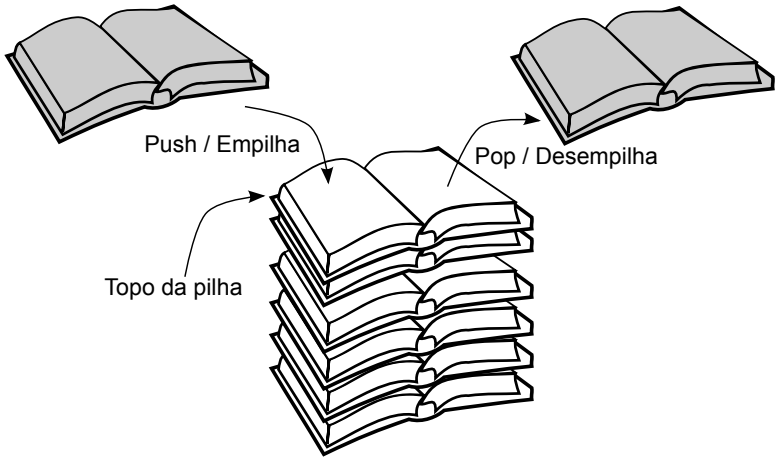


Figura 5.1: Operações fundamentais para manipulação de pilhas: `push` e `pop`

Além dessas duas operações, é comum uma implementação de pilha apresentar também operações acessórias como:

Tabela 5.1: Funções complementares em pilhas

Função	Descrição
`top`	Consultar o elemento no topo da pilha sem removê-lo.
`isEmpty`	Checar se a pilha está vazia.
`isFull`	Checar se a pilha está completa.
`getSize`	Consultar a quantidade de elementos na pilha.

Apresentamos então a seguir a definição das operações fundamentais e complementares para manipulação de pilhas.

Exemplo 5.1 Definição das operações de uma `stack`

Código fonte code/capitulo-05/stack.h

```
/**
 * Insere um elemento no topo da pilha. Tem complexidade O(1).
 */
Stack push(Stack stack, Element item);

/**
 * Retorna o elemento no topo da pilha sem removê-lo.
 * Tem complexidade O(1).
 */
Element top(Stack stack);

/**
 * Remove o elemento no topo da pilha. Tem complexidade O(1).
 */
Stack pop(Stack stack);

/**
 * Retorna 1 se a pilha estiver vazia e 0 caso contrário.
 * Tem complexidade O(1).
 */
int isEmpty(Stack stack);

/**
 * Retorna 1 se a pilha estiver completa e 0 caso contrário.
 * Tem complexidade O(1).
 */
int isFull(Stack stack);

/**
 * Retorna o número de elementos na pilha. Tem compleidade O(n).
 */
int getSize(Stack stack);
```

5.1.1 Implementação com Lista Encadeada

A forma mais simples de implementar uma Pilha é utilizando uma lista encadeada para armazenar os seus elementos.

Exemplo 5.2 Implementação de `stack` utilizando uma *LinkedList*

Código fonte code/capitulo-05/stack.c

```
#include "../capitulo-04/linkedlist.c"

#ifndef STACK_T
#define STACK_T 1

typedef LinkedList Stack;

```

```
#endif

/**
 * Insere um elemento no topo da pilha. Tem complexidade O(1).
 */
Stack push(Stack stack, Element item) {
  return prepend(stack,item);
}

/**
 * Retorna o elemento no topo da pilha sem removê-lo.
 * Tem complexidade O(1)
 */
Element top(Stack stack) {
  return getElement(getBegin(stack));
}

/**
 * Remove o elemento no topo da pilha. Tem complexidade O(1).
 */
Stack pop(Stack stack) {
  return removeFirst(stack);
}

/**
 * Retorna sempre 0 uma vez que esta implementação se baseia em uma
 * lista encadadeada.
 */
int isFull(Stack stack) {
  return 0;
}
```

Nesta implementação apresentamos apenas as duas operações fundamentais e duas das operações acessórias. A razão para isso é que como nossa implementação de `stack` se baseia em uma *LinkedList*, as demais operações já estão implementadas para este tipo de estrutura e podem ser re-utilizadas para pilhas. Sendo assim, não precisamos re-implementar as operações `isEmpty` e `getSize`.

A operação `isFull` foi implementada apenas para retornar `0` uma vez que podemos assumir que uma lista encadeada nunca está completa.

A operação `push`, responsável por inserir um elemento no topo da pilha, é implementada utilizando a função `prepend`, que insere um elemento no começo da lista encadeada.

```
/**
 * Insere um elemento no topo da pilha. Tem complexidade O(1).
 */
Stack push(Stack stack, Element item) {
  prepend(stack,item);
}
```

Já a operação `pop`, responsável pela remoção de um elemento do topo da pilha, é implementada utilizando a operação `removeFirst`, que remove o primeiro elemento da lista encadeada.

```
/**
```

```
 * Remove o elemento no topo da pilha. Tem complexidade O(1).
 */
Stack pop(Stack stack) {
  removeFirst(stack);
}
```

Finalmente, temos a operação `top`, que retorna o elemento no topo da pilha sem removê-lo. Essa operação pode ser facilmente implementada utilizando a função `getBegin`, que retorna um iterador para o primeiro elemento da lista.

```
/**
 * Retorna o elemento no topo da pilha sem removê-lo.
 * Tem complexidade O(1).
 */
Element top(Stack stack) {
  return getElement(getBegin(stack));
}
```

5.1.2 Aplicações de Pilhas

Pilhas podem ser utilizadas para resolver vários tipos de problemas. Alguns destes problemas são listados a seguir.

5.1.2.1 Convertendo um Número Decimal em Binário

Nota
Você já estudou a conversão de números decimais para binário na disciplina Introdução a Computação. Se sentir necessidade de revisar este conteúdo consulte a seção “Conversão de números da base 10 para uma base b qualquer” no livro da disciplina.

A lógica para converter um número de decimal para binário é a seguinte:

1. Leia um número
2. Itere (enquanto o número for maior que 0)
    a. Encontre o resto da divisão do número por 2
    b. Imprima o resto
    c. Divida o número por 2
3. Fim da iteração

No entanto, há um problema com esta lógica. Suponha que queremos converter o número decimal `23` para binário. Usando esta lógica (ver Figura 5.2 [54]), obteremos o resultado `11101` ao invés de `10111`, que seria o resultado correto.

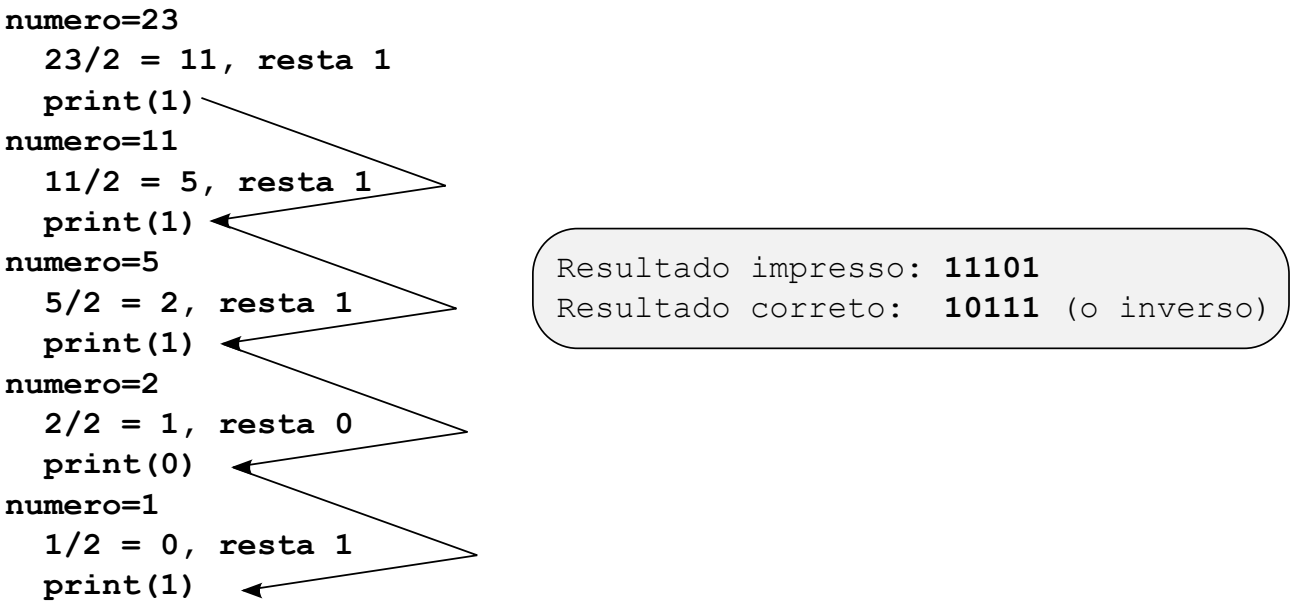


Figura 5.2: Execução da lógica sobre o decimal 23

Para resolver este problema podemos utilizar uma pilha fazendo uso de sua propriedade LIFO. Ao invés de irmos imprimindo os restos das divisão por dois ao longo do algoritmo, nós empilhamos esse resto na pilha. Depois que todo o número tiver sido convertido vamos desempilhando digito a digito da pilha e imprimindo (ver Figura 5.3 [54]). Desta maneira, conseguimos converter corretamente um número decimal para binário. O Exemplo 5.3 [54] implementa este algoritmo.

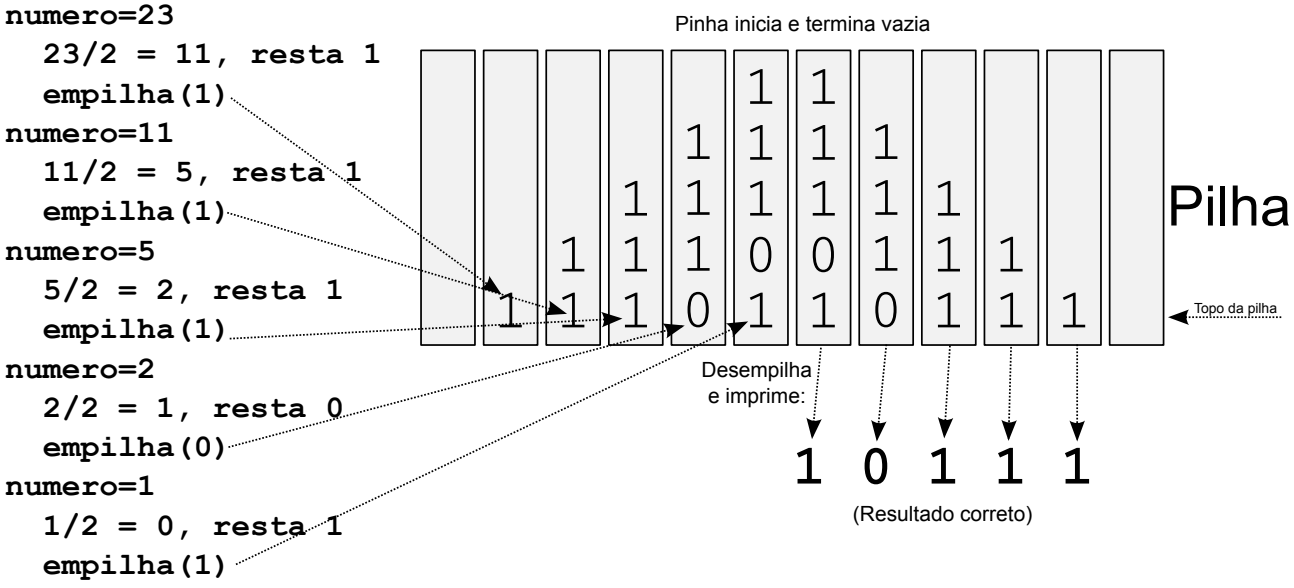


Figura 5.3: Execução da lógica utilizando uma pilha para impressão

Exemplo 5.3 Convertendo um número decimal para binário

Código fonte code/capitulo-05/decbin.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```c
#include "stack.c"

int main() {

  int numero, digito;
  Stack stack;

  printf("Digite um número decimal: ");
  scanf( "%d", &numero);

  //Realiza a conversão, empilhando os digitos
  while( numero > 0 ) {
    digito = numero % 2;
    stack = push(stack,digito);
    numero = numero / 2;
  }

  //Desempilha e imprime os digitos.
  while( !isEmpty(stack)) {
    digito = top(stack);
    stack = pop(stack);
    printf( "%d", digito);
  }
  printf( "\n");

  return 0;
}
```

5.1.2.2 Torres de Hanoi

Figura 5.4: Torres de Hanoi

Uma das mais interessantes aplicações de pilhas é a sua utilização para resolver um quebra-cabeças conhecido como Torres de Hanoi. De acordo com uma antiga história Brâmane a idade do universo é calculada de acordo com o tempo gasto por um grupo de monges, trabalhando 24 horas por dia, para mover 64 discos de uma torre para para outra, segundo o seguinte conjunto de regras:

1. Você só pode mover um disco por vez
2. Uma terceira torre pode ser utilizada para armazenamento temporário
3. Você não pode colocar um disco de diâmetro `X` sobre um disco com diâmetro `Y` se `X > Y`

Assumimos que A é a primeiro torre, onde os discos estão originalmente localizados, B é a segunda torre, para armazenamento temporário, e C é a terceira torre, para onde os discos devem ser movidos de acordo com as regras.

A seguir apresentamos a sequência de passos necessários para resolver o problema das Torres de Hanoi com 3 discos.

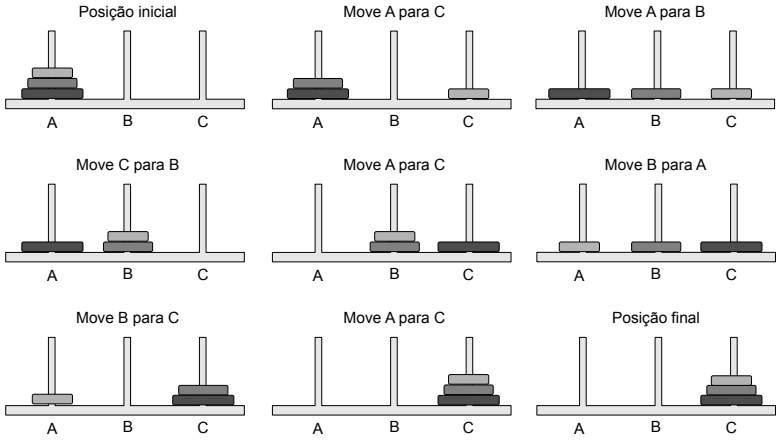


Figura 5.5: Movimentos da Torre de Hanoi com 3 discos

A sequência de passos para resolver o problema com 3 discos é:

1. Move disco de A para C
2. Move disco de A para B
3. Move disco de C para B
4. Move disco de A para C
5. Move disco de B para A
6. Move disco de B para C
7. Move disco de A para C

Se, por outro lado, tivéssemos 4 discos, a sequência de passos seria:

1. Move disco de A para B
2. Move disco de A para C
3. Move disco de B para C
4. Move disco de A para B

5. Move disco de C para A
6. Move disco de C para B
7. Move disco de A para B
8. Move disco de A para C
9. Move disco de B para C
10. Move disco de B para A
11. Move disco de C para A
12. Move disco de B para C
13. Move disco de A para B
14. Move disco de A para C
15. Move disco de B para C

Ou seja, para resolver o problema com n discos são necessários 2^n - 1 passos.

A seguir apresentamos uma solução recursiva para o problema das Torres de Hanoi. Esta solução utiliza implicitamente uma estrutura de pilha, através de chamadas recursivas de função, para armazenar resultados intermediários e imprimir a ordem correta de movimentação dos discos.

Exemplo 5.4 Solução recursiva para o problema das Torres de Hanoi

Código fonte code/capitulo-05/hanoi.c

```c
#include <stdio.h>

/**
 * Solução recursiva para o problema das Torres de Hanoi.
 * Move n discos da torre SOURCE para a torre DEST, usando INTERM
 * como intermediário.
 */
void hanoi(int n, char source, char dest, char interm) {
  if( n > 0 ) {
    hanoi(n - 1, source, interm, dest);
    printf("Move disco de %c para %c\n", source, dest);
    hanoi(n - 1, interm, dest, source);
  }
}

int main() {
  hanoi(3, 'A', 'C', 'B');
  return 0;
}
```

Importante
É possível modificar esta solução para resolver o problema de forma iterativa, sem recursividade, utilizando uma estrutura de pilha.

A complexidade da solução apresentada é $O(2^n)$. Portanto, fica óbvio que este problema só pode ser resolvido para valores pequenos de n (geralmente `n<=30`). No caso dos monges, o número de movimentos necessários para transferir os 64 discos, de acordo com as regras apresentada, será *18.446.744.073.709.551.615*, o que, com certeza, deve levar um bocado de tempo.

5.1.2.3 Calculadora pós-fixada

As calculadoras comuns utilizam a notação *infixada*, inserindo o operador da função entre os dois números que desejamos realizar a operação, exemplo: `13+2=`. O resultado da operação só aparece depois que pressionamos outro operador ou =.

Existem outros tipos de calculadoras (científicas e financeiras) que funcionam de forma diferente. Nelas inserimos primeiros os números e depois os operadores, esta notação é conhecida como Polonesa Inversa ou *pós-fixada*, exemplo: `13ENTER2+` ou também `13ENTER2ENTER+`.

Veja a diferença entre estas duas notações na Tabela 5.2 [58].

Tabela 5.2: Diferença entre notação *infixada* e *pós-fixada*

infixada	***pós-fixada*** [1]	**Resultado**
`2 + 3`	`2 3 +`	`5`
`2 + 3 * 4`	`2 3 4 * +`	`14`
`2 + (3 * 4)`	`2 3 4 * +`	`14`
`2 x 3 + 4`	`2 3 * 4 +`	`10`
`2 x (3 + 4)`	`2 3 4 + *`	`14`

O algorítmo da calculadora pós-fixada utiliza uma pilha para realização dos cálculos, veja um exemplo a seguir:

```
Leia uma entrada
  Se a entrada for um número
    Empilhe o número lido
  Se a entrada for um operador
    Desempilhe dois valores da pilha
    Realize a operação lida utilizando os dois valores
    Empilhe o resultado na pilha
    Imprime o resultado do topo da pilha
```

Em outras palavras, você pode imaginar que os operandos (os números) estão numa pilha, quando um operador (o sinal) for encontrado, basta desempilhar dois operandos, realizar a operação entre eles, empilhar o resultado (colocando-o no topo da pilha) e por último, imprime o resultado lendo do topo da pilha. A Tabela 5.3 [59] ilustra a execução da seguinte operação pós-fixada: `2 3 4 + *`.

[2] Os ENTERs foram omitidos.

Tabela 5.3: Execução da operação pós-fixada: `2 3 4 + *`

Leitura	Ações	Pilha[3]
`2`	empilha o valor lido: `2`	`[2]`
`3`	empilha o valor lido: `3`	`[3] 2`
`4`	empilha o valor lido: `4`	`[4] 3 2`
`+`	desempilha `4` e `3`, empilha a soma deles e imprime topo da pilha	`[7] 2`
`*`	desempilha `7` e `2`, empilha a multiplicação deles e imprime topo da pilha	`[14]`

5.1.2.4 Avaliação de Expressões

Expressões matemáticas podem ser representadas nas notações *pré-fixada*, *pós-fixada* e *infixada*. A conversão de uma notação para a outra pode ser feita utilizando uma pilha. De fato, muitos compiladores utilizam pilhas para analisar a sintaxe de expressões e blocos de programa antes de efetuar a tradução em código de baixo nível.

A expressão em notação infixada `(((2 * 5) -(1 * 2))/(11-9))` ao ser avaliada deve resultar no valor `4`. Para avaliar este tipo de expressão devemos analisar 5 tipos de caracteres:

- Abertura de parênteses
- Números
- Operadores
- Fechamento de parênteses
- Quebra de linha

O processamento de uma expressão com essa envolve um conjunto de operações com uma pilha. Um algoritmo para a avaliação de expressões é descrito a seguir:

1. Ler caractere da entrada
2. Ações a serem tomadas para cada caractere:
    a. Abertura de parênteses: inserir no topo da pilha e repetir passo 1
    b. Número: inserir no topo da pilha e repetir passo 1
    c. Operador: inserir no topo da pilha e repetir passo 1
    d. Fechamento de parênteses: remover elemento do topo da pilha.
        i. Se o caractere for uma abertura de parênteses, descartar e repetir passo 1.
        ii. Se não, devemos remover os próximos três elementos do topo da pilha. O primeiro elemento será denominado `op2`, o segundo `op` e o terceiro `op1`. Devemos então avaliar a expressão `op1 op op2`, onde `op1` e `op2` são números e `op` é um operador. O resultado dessa expressão deve então ser inserido no topo da pilha

[4] Em destaque o topo da pilha entre colchetes: `[]`.

e. Quebra de linha: remover o elemento do topo da pilha e imprimir o resultado

Exemplo do funcionamento deste algoritmo:

Dada a expressão `(((2 * 5) -(1 * 2)) /(11 -9))` o conteúdo da pilha durante sua avaliação seria o seguinte:

Símbolo na entrada	Conteúdo da pilha	Operação
(	(	
(	((	
(	(((	
2	(((2	
*	(((2 *	
5	(((2 * 5	(((2*5
)	((10	2 * 5 = 10 e `push`
-	((10 -	
(	((10 - (	
1	((10 - (1	
*	((10 - (1 *	
2	((10 - (1 * 2	
)	((10 - 2	1 * 2 = 2 e `push`
)	(8	10 - 2 = 8 e `push`
/	(8 /	
(	(8 / (	
11	(8 / (11	
-	(8 / (11 - 9	
)	(8 / 2	11 - 9 = 2 e `push`
)	4	8 / 2 = 4 e push
`\n`		`pop` e imprime resultado

5.1.2.5 Exercícios

1. Implemente uma da calculadora pós-fixada utilizando uma pilha. A calculadora deve:

   - Realizar operações apenas com números inteiros;
   - Utilizar os operandos "+ - * /" para as operações de soma, subtração, multiplicação e divisão inteira respectivamente;
   - Imprimir o valor do topo da pilha se nenhuma entrada for digitada;

   Dica
   Você pode utilizar a funções `atoi(char *nptr)` para converter um string para `int`.

2. Implemente um programa capaz de avaliar expressões matemáticas em notação infixada

3. Implemente um solução para o problema das Torres de Hanoi sem utilizar recursividade

5.2 Filas

Uma fila (*queue* em inglês) é uma outra variação lógica de uma estrutura de armazenamento linear que representa um enfileiramento de elementos. O conceito também reflete uma estrutura básica encontrada no mundo real, utilizado geralmente para disciplinar o acesso a algum tipo de recurso.

O funcionamento padrão de uma fila é FIFO (*First In First Out*) que significa *o primeiro a entrar é o primeiro a sair*.

De forma análoga às pilhas, existem duas operações básicas para gerenciar a inserção e remoção de elementos de uma fila. São elas:

- Inserir um elemento no final da fila (`enqueue`)
- Remover um elemento do começo da fila (`dequeue`)

A figura abaixo ilustra o funcionamento destas operações.

Figura 5.6: Funcionamento das operações `enqueue` e `dequeue`

Assim como no caso das pilhas, as filas também possuem 4 operações complementares:

Tabela 5.4: Funções complementares em pilhas

Função	Descrição
`front`	Consultar o elemento na cabeça da fila sem removê-lo.
`isEmpty`	Checar se a fila está vazia.
`isFull`	Checar se a fula está completa.
`getSize`	Contar a quantidade de elementos na fila.

Apresentamos a seguir a definição das operações fundamentais e complementares para manipulação de filas.

Exemplo 5.5 Definição das operações de uma `queue`

Código fonte code/capitulo-05/queue.h

```c
/**
 * Insere um elemento no final da fila . Tem complexidade O(1)
 */
void enqueue(Queue *queue, Element item);

/**
 * Retorna o elemento na cabeça da fila sem removê-lo.
 * Tem complexidade O(1).
 */
Element front(Queue *queue);

/**
 * Remove o elemento na cabeça da fila. Tem complexidade O(1).
 */
Element dequeue(Queue *queue);

/**
 * Retorna 1 se a fila estiver vazia e 0 caso contrário.
 * Tem complexidade O(1).
 */
int isEmpty(Queue *queue);

/**
 * Retorna 1 se a fila estiver completa e 0 caso contrário.
 * Tem complexidade O(1).
 */
int isFull(Queue *queue);

/**
 * Retorna o número de elementos na fila. Tem compleidade O(1).
 */
int getSize(Queue *queue);
```

5.2.1 Implementação baseada na nossa estrutura de Nó.

Mais uma vez, a forma mais simples de se implementar uma Fila é utilizar uma lista encadeada como estrutura base para armazenar os seus elementos e escrever as rotinas que disciplinam a inserção e remoção de elementos da estrutura.

Porém, desta vez, seguiremos uma abordagem um pouco diferente da adotada na nossa implementação para a estrutura de pilha. Ao invés de basearmos nossa implementação na utilização direta da estrutura *LinkedList*, daremos uma passo para trás no nível de abstração e utilizaremos diretamente a estrutura de Nó definida no capítulo 1 deste livro. A razão para isso é arejar a cabeça do leitor mostrando uma nova variação de estrutura de dados primária.

A seguir vemos uma implementação de Fila baseada na nossa estrutura de Nó.

Exemplo 5.6 Implementação das operações de uma `queue` utilizando a estrutura *Node*

Código fonte code/capitulo-05/queue.c

```c
#include "../capitulo-01/node.c"
```

```
2
3  #ifndef QUEUE_T
4  #define QUEUE_T 1
5
6  typedef struct {
7    Node *head;
8    Node *tail;
9    int size;
10 } Queue;
11
12 #endif
13
14 /**
15  * Retorna 1 se a fila estiver vazia e 0 caso contrário. Tem
16  * complexidade O(1)
17  */
18 int isEmpty(Queue *queue) {
19   return queue->size == 0;
20 }
21
22 /**
23  * Insere um elemento no final da fila. Tem complexidade O(1)
24  */
25 void enqueue(Queue *queue, Element item) {
26   if(isEmpty(queue)) {
27     queue->head = makeNode(item,NULL);
28     queue->tail = queue->head;
29   }
30   else {
31     Node *nn = makeNode(item,NULL);
32     queue->tail->next = nn;
33     queue->tail = nn;
34   }
35   queue->size++;
36 }
37
38 /**
39  * Retorna o elemento na cabeça da fila sem removê-lo.
40  * Tem complexidade O(1)
41  */
42 Element front(Queue *queue) {
43   return getValue(queue->head);
44 }
45
46 /**
47  * Remove o elemento na cabeça da fila. Tem complexidade O(1)
48  */
49 Element dequeue(Queue *queue) {
50   Element v = getValue(queue->head);
51   Node *nn = queue->head;
52   queue->head = queue->head->next;
53   queue->size--;
54   if(isEmpty(queue))
```

```
55      queue->tail = queue->head;
56    free(nn);
57    return v;
58  }
59
60  /**
61   * Retorna sempre 0 uma vez que esta implementação se baseia em uma
62   * lista encadadeada.
63   */
64  int isFull(Queue *queue) {
65    return 0;
66  }
67
68  /**
69   * Retorna o número de elementos na fila. Tem compleidade O(1).
70   */
71  int getSize(Queue *queue) {
72    return queue->size;
73  }
```

A primeira coisa a observar nesta implementação é a definição do tipo abstrato de dados `Queue`. Este tipo, ilustrado no segmento de código a seguir, apresenta dois apontadores para Nó. O primeiro apontador é utilizado para referenciar a *cabeça da fila* enquanto que o segundo referencia sua *cauda*. Essa necessidade já seria esperada se considerarmos que os elementos são inseridos e removidos de locais diferentes da fila. Além disso, a estrutura armazena também um contador de itens inseridos na fila, utilizado para tornar trivial a implementação das operações `isEmpty` e `getSize`.

Definição do Tipo Abstrato de Dados queue

```
1  typedef struct {
2          Node *head;
3          Node *tail;
4          int size;
5  } Queue;
```

A primeira operação fundamental de uma fila, `enqueue`, utilizada para inserir elementos no final da fila, tem sua implementação reapresentada a seguir.

Implementação da operação enqueue

```
1  /**
2   * Insere um elemento no final da fila. Tem complexidade O(1)
3   */
4  void enqueue(Queue *queue, Element item) {
5    if(isEmpty(queue)) {
6      queue->head = makeNode(item,NULL);
7      queue->tail = queue->head;
8    }
9    else {
10     Node *nn = makeNode(item,NULL);
11     queue->tail->next = nn;
12     queue->tail = nn;
13   }
14   queue->size++;
```

```
15 }
```

Nesta implementação é possível observar que há dois casos a serem tratados. O primeiro caso envolve a inserção de elementos em uma lista vazia. Nesta situação, o elemento é inserido na *cabeça da lista* e este único elemento passa então a ser tanto o primeiro quanto o último elemento da lista.

O segundo caso acontece quando inserimos elementos em uma lista não-vazia. Nesta situação, o elemento deve ser inserido após o último elemento já presente na lista, referenciado pelo apontador de cauda (`tail`). Em seguida este apontador deve ser atualizado para apontar para o novo último elemento da lista.

A implementação do método que remove elementos de uma fila, `dequeue`, ilustrada a seguir, é um pouco mais simples.

Implementação da operação dequeue

```
1  /**
2   * Remove o elemento na cabeça da fila. Tem complexidade O(1).
3   */
4  Element dequeue(Queue *queue) {
5    Element v = getValue(queue->head);
6    Node *nn = queue->head;
7    queue->head = queue->head->next;
8    queue->size--;
9    if(isEmpty(queue))
10     queue->tail = queue->head;
11   free(nn);
12   return v;
13 }
```

Esta operação salva o valor do elemento armazenado na *cabeça da fila* em uma variável temporária e em seguida atualiza o *apontador de cabeça* para que ele aponte para o próximo elemento na fila (“a fila anda”). Após a atualização do contador de elementos na fila, checa-se se a fila está vazia. Se isso ocorrer, atualizamos também o apontador de cauda.

Como mencionado anteriormente, a implementação das operações acessórias é trivial e não será discutida em detalhes aqui. Uma observação diz respeito a função `isFull`, que sempre retorna 0. Isso acontece por assumirmos que a memória sempre permitirá a criação de um novo Nó para inserção no final da fila, portanto, a fila nunca estará completa.

5.2.2 Aplicações de Filas

Filas são, assim como pilhas, estruturas de dados que sempre permeiam o desenvolvimento de algoritmos. Aplicações que requerem sua utilização são muito frequentes e muitas vezes envolvem problemas de otimização e escalonamento, como os que estudaremos na disciplinas de Sistemas Operacionais.

Além disso, uma aplicação clássicas das filas é sua utilização para o cálculo de distâncias.

5.2.2.1 Cálculo de Distâncias

Imagine 6 cidades numeradas de 0 a 5 e interligadas por estradas de mão única. (É claro que você pode trocar “6” pelo seu número favorito.) As ligações entre as cidades são representadas por uma matriz A da seguinte maneira:

`A[i][j]` vale `1` se existe estrada da cidade `i` para a cidade `j` e vale `0` em caso contrário.

Suponha que a matriz tem zeros na diagonal, embora isso não seja importante.

Um exemplo desta matriz, que chamaremos de matriz de conectividade, é apresentado a seguir.

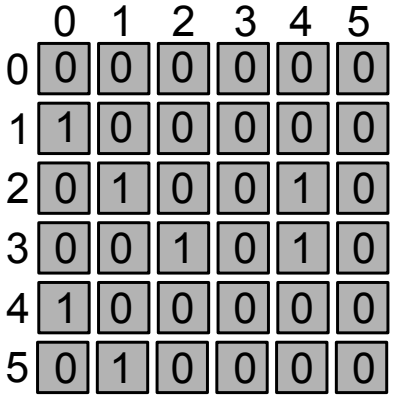
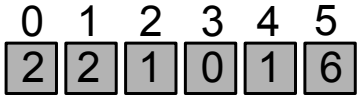


Figura 5.7: Matriz de conectividade

A distância de uma cidade `c` a uma outra `j` é o menor número de estradas que devo percorrer para ir de `c` a `j`. Nosso problema: dada uma cidade `c`, determinar a distância de `c` a cada uma das demais cidades.

As distâncias serão armazenadas em um vetor `d`: a distância de `c` a `j` será `d[j]`. Que fazer se é impossível chegar de `c` a `j`? Poderíamos dizer nesse caso que `d[j]` é infinito. Mas é mais limpo e prático dizer que `d[j]` vale 6, pois nenhuma distância `real` pode ser maior que `5`. Se adotarmos `c` igual a `3` no exemplo acima, teremos `d` igual a:



Eis a ideia de um algoritmo que usa o conceito de fila para resolver nosso problema:

- uma cidade é considerada ativa se já foi visitada mas as estradas que começam nela ainda não foram exploradas;
- mantenha uma fila das cidades ativas;
- em cada iteração, remova da fila uma cidade i e insira na fila todas as cidades vizinhas de i que ainda não foram visitadas.

A seguir temos um programa que implementa este algoritmo para encontrar a menor distância entre a cidade 0 e as demais 5 cidades. Para fins de ilustração, a matriz de conectividade é gerada de forma aleatória.

Exemplo 5.7 Programa que utiliza uma fila para encontrar a menor distância entre um grupo de cidades

Código fonte code/capitulo-05/dist.c

```c
#include "queue.c"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Recebe uma matriz A que representa as interligações entre
// cidades 0,1,..,5: há uma estrada (de mão única) de i a j
// se e só se A[i][j] == 1.  Devolve um vetor d que registra
// as distâncias da cidade c a cada uma das outras: d[i] é a
// distância de c a i.

int *distancias( int A[6][6], int c) {
   int *d, j;

   Queue fila;
   fila.size = 0;
   fila.head = NULL;
   fila.tail = NULL;

   d = malloc( 6 * sizeof (int));
   for (j = 0; j < 6; ++j)
     d[j] = 6;

   d[c] = 0;

   enqueue(&fila,c); // c entra na fila

   while (!isEmpty(&fila)) {
      int i, di;

    i = dequeue(&fila);  // i sai da fila

      di = d[i];
      for (j = 0; j < 6; ++j)
         if (A[i][j] == 1 && d[j] >= 6) {
            d[j] = di + 1;
            enqueue(&fila,j); // j entra na fila
         }
   }

   return d;
}

int main() {

  int matriz[6][6];
  int *dist;

  srand(time(NULL));
```

```
50
51    //Gerando uma matriz de conectividade aleatória.
52    printf( "Matriz de conectividade: \n");
53    for(int i = 0; i < 6; i++) {
54      printf( "Cidade [%d]: ", i);
55      for(int j = 0 ; j < 6; j++) {
56        if( i != j )
57          matriz[i][j] = (rand()%2 == 0);
58        else
59          matriz[i][j] = 0;
60        printf( "%d ", matriz[i][j]);
61      }
62      printf("\n\n");
63    }
64
65    //Calculando a distância mínima da cidade 0 para as demais cidades
66    dist = distancias(matriz,0);
67
68    printf( "Distância mínima da cidade 0 para as demais cidades:\n");
69    for( int i = 0; i < 6; i++)
70      printf( "%d " , dist[i]);
71    printf("\n");
72
73    return 0;
74  }
```

5.2.2.2 Exercícios

1. Seria possível implementar uma estrutura de fila com apenas um apontador para a cabeça da fila? Justifique sua resposta.
2. Quais seriam os impactos dessa alteração na complexidade assintótica das suas funções?
3. Quais seriam os impactos da remoção do campo `size` na complexidade assintótica das funções de manipulação de filas?

Dica
A funcionalidade de obter o tamanho da fila é importante, por isto, ao remover o campo `size` você deverá adotar outra estratégia que continue provendo a mesma funcionalidade: retornar o tamanho da fila.

5.3 Recapitulando

É importante fazer neste momento um paralelo entre o funcionamento de estruturas FIFO, como as filas, e as estruturas LIFO, como as pilhas. A figura abaixo ilustra bem estas diferentes. Estruturas do tipo FIFO preservam a *ordem de chegada* dos elementos enquanto que em estruturas do tipo LIFO retornam a *ordem inversa*.

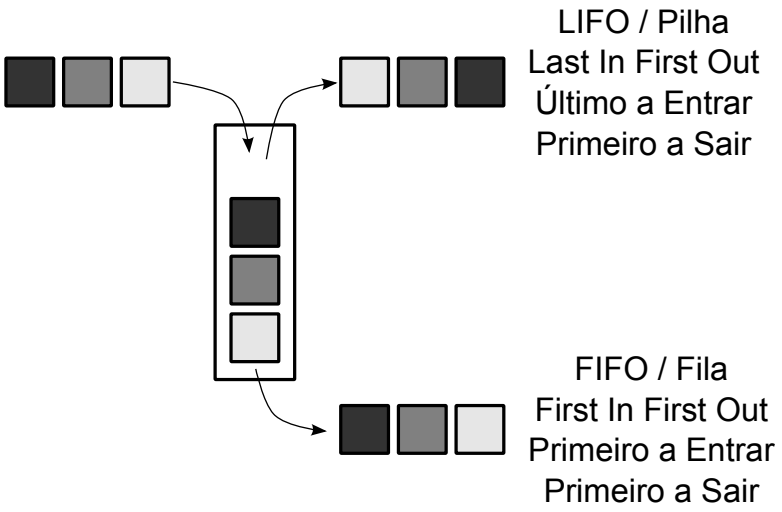


Figura 5.8: Comparação do funcionamento de estruturas LIFO e FIFO

Observamos que em estruturas LIFO os elementos são inseridos e removidos da mesma extremidade da estrutura. Em se tratando de uma pilha, são inseridos e removidos do topo da pilha. Já no caso de uma estrutura FIFO, os elementos são inseridos em uma extremidade (cauda da fila) e removidos da outra extremidade (cabeça da fila).

Estas são duas das principais estruturas de dados que estudaremos e, apesar de apresentarem um funcionamento bastante simples, são amplamente utilizadas na Ciência da Computação para a implementação de soluções para os mais diversos problemas, desde a implementação de jogos até a resolução de intrincados problemas de escalonamento.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?

Acesse https://github.com/edusantana/estruturas-de-dados-livro/issues/new para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**capitulo-05**) e a versão do livro (**v1.0.0**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 6

Índice Remissivo