Pedro Kantek

Estrutura de Dados e arquivos

Unicenp
Dez/2007
Curitiba

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Sumário

©88-08, Pedro Kantek

 ©88-08, Pedro Kantek

©88-08, Pedro Kantek

Se te pedirem um programa, faz dois, e deita fora o primeiro (Pedro Guerreiro)

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Capítulo 1

Questões Operacionais

1.0.1 Regras da disciplina

Estas são as regras propostas para a cadeira de Estrutura de Dados no corrente ano.

1. Em todas as aulas haverá um exercício prático individual e diferente sobre o assunto da aula que será dada.
2. O exercício em questão deverá ser devolvido até o último dia do bimestre letivo.
3. Eventuais discussões quanto à nota lançada e formas de solução do exercício também poderão ter lugar aqui.
4. Não haverá segunda chamada para exercícios perdidos.
5. Não serão aceitos xerox ou fax de exercícios. Apenas o exercício que contiver o nome do aluno será aceito.
6. A média bimestral será obtida fazendo a média aritmética da prova bimestral (com peso 4) e a nota dos exercícios. Esta será a média de todos os exercícios do bimestre. A nota dos exercícios entrará na média bimestral com peso 6.
7. Na aula seguinte à da prova o professor distribuirá um extrato no qual constarão as notas obtidas por cada aluno em cada exercício individual, além da nota da prova.

1.0.2 Portal

No Portal Unicenp (www.unicenp.edu.br) estão os textos base de todas as aulas da disciplina, em um livro de nome LIVROED.PDF.

1.0.3 Exercícios Individuais

código

São gerados usando um pacote denominado Algoritmos Vivos, que está em desenvolvimento desde o ano de 1992, quando começou a ser usado ainda na Universidade Católica do Paraná, para a disciplina de estrutura de dados. Hoje é uma coleção de mais de 135 exercícios abrangendo as áreas de estruturas de dados, tópicos avançados em informática e inteligência artificial. Cada exercício possui um código que o remete a uma classificação de temas dentro da Ciência da Computação, e uma seqüência que informa qual o posição do exercício dentro do ano letivo.

Por exemplo, o exercício a ser feito hoje, tem o código VIVO213a (VIVO é a identificação do pacote gerador de exercícios, o número 2 remete à estruturas de dados especiais e o sub-número 13 remete a um exercício em especial. A letra "a" indica o primeiro programa dentro do workspace).

Seqüência

cada exercício tem um identificador único, chamado seqüência com as seguintes informações: **aaTOM101**, que deve ser assim interpretada:

aa Ano atual

ED disciplina de ED

M ou N Turno

1 1^o bimestre do ano letivo

01 Primeiro exercício deste bimestre

Aqui, a lista de exercícios previstos para o ano, como ela está em 8 de fevereiro de 2006:

Revisão de logaritmos

código: 215
finalizado em 06/01/03
Exercício prévio para o estudo da complexidade algorítmica. São apresentadas as bases da potência, da radiciação e dos logaritmos. Em cada um destes temas, há 3 exercícios numéricos.

Problema da partição

código: 231
finalizado em 17/07/02
Um algoritmo aproximado (guloso) para resolver o problema da partição. Dada uma coleção de objetos, cada um com seu peso, dividor o conjunto em 2 subconjuntos disjuntos, de modo que a soma dos dois subconjuntos seja igual. Este problema é NP.

Requisitos de um automóvel

código: 232
finalizado em 21/01/01
Aplicação de algoritmo guloso: qual o menor conjunto de veículos (cada um com a sua lista de habilidades e seu preço) que satisfaz a uma lista de requisitos do comprador ?

Habilidades de um jogador de futebol

código: 233
finalizado em 18/01/01
Aplicação de algoritmo guloso: qual o menor conjunto de jogadores que satisfazem a todas as habilidades que um clube necessita ?

©88-08, Pedro Kantek

Prática de recursividade

código: 251
finalizado em 18/04/00
3 funções recursivas e que se chamam entre si são definidas. Algumas variáveis globais também. Finalmente pede-se que o aluno siga as funções e reproduza as suas saídas.

Anagramas

código: 252
finalizado em 18/04/00
Pede que o aluno construa os anagramas com um determinado conjunto de letras. O algoritmo é recursivo. Ainda na versão 1.

Passeio do cavalo

código: 253
finalizado em 18/04/00
O conceito de *backtracking* é mostrado através de um exemplo: o passeio completo do cavalo sobre um tabuleiro de xadrez. Dado um tabuleiro $n \times n$ (do qual o 8×8 é um caso particular) e dada uma casa inicial pede-se qual a seqüência de passos do cavalo de maneira a passar em todas as casas do tabuleiro sem repetir nenhuma. Finalmente uma instância do problema é oferecida ao aluno, pedindo-se uma parte do caminho feito pelo cavalo. Ótimo candidato a implementação.

3 funções recursivas

código: 254
finalizado em 24/07/03
São mostradas 3 funções recursivas: uma ordenação recursiva, o cálculo da potência recursiva (aquela que abaixa a complexidade normal de $O(n)$ para $O(log\ n)$) e o algoritmo do mdc de Euclides. Pede-se que o aluno simule as recursões e informe o que acontece após 2 ou 3 passos de recursão.

Árvore quad

código: 255
finalizado em 04/02/02
Apresenta a árvore quad como um exemplo de estrutura de dados recursiva. Visto o conceito, dá-se uma árvore armazenada em uma matriz e pede-se que o aluno construa o desenho correspondente. Depois dá-se um desenho e pede-se a matriz. No fim o aluno é surpreendido pelo fato de que os 2 exercícios referem-se à mesma árvore (e portanto a resposta de um é o enunciado do outro e vice versa), mas só quem gastou algum tutano descobre isto.

Casamento Estável

código: 256
finalizado em 11/04/06
Descreve o algoritmo (conforme mostrado pelo Niklaus Wirth no livro Algoritmos e Estruturas de dados, pág. 129) e apresenta dois conjuntos de 10 elementos cada, e suas respectivas preferências. São pedidas as perdas em dois casos: casamentos com minimização de perdas do primeiro conjunto e também do segundo conjunto. Este

exercício tem que ser implementado. É quase impossível resolvê-lo à mão, por conta da recursividade

Sudoku para computadores

código: 257
finalizado em 08/09/06
Descreve o sudoku e apresenta o algoritmo recursivo que o resolve. O aluno é convidado a seguir o algoritmo e preencher apenas a primeira linha do sudoku. Esta restrição é para que o aluno consiga seguir o algoritmo sem se enredar em chamadas recursivas.

Estratégias de alocação

código: 271
finalizado em 26/10/01
Descreve as estratégias de alocação next fit, first fit, best fit, worst fit e alocação offline. Depois uma certa arrumação de sacos de compra em um mercado é pedida em cada uma das estratégias.

Alocação em um teatro

código: 272
finalizado em 15/05/01
Um teatro usa uma estrutura encadeada para reservar lugares, além de guardar informações sobre os compradores nessa mesma estrutura. Dado um teatro parcialmente vendido, pede-se que o aluno responda perguntas sobre o teatro e sobre os compradores de lugares.

Uma família representada em uma estrutura de dados

código: 273
finalizado em 18/02/02
Uma família com 5 gerações é mostrada através de uma estrutura encadeada. Trata-se de uma matriz de 50 linhas e 6 colunas informando as relações parenterais na família. Pede-se que o aluno desenhe a árvore genealógica, além de responder a diversas perguntas sobre a família. Este é o exercício inaugural em "estrutura de dados" e serve para mostrar a lógica dos algoritmos vivos.

Alocação do DOS

código: 291
finalizado em 04/05/00
Mostra-se o mecanismo que o DOS usava (usa) para gerenciar alocações de espaço em disco. Uma série de criação, modificação e apagamento de arquivos é pedida ao aluno. Ao final, pede-se o estado do diretório e da FAT do disco.

Alocação do UNIX

código: 292
finalizado em 04/05/00
Mostra-se o mecanismo que o UNIX usa para gerenciar alocações de espaço em disco. Uma série de criação, modificação e apagamento de arquivos é pedida ao aluno. Ao

final, pede-se o estado dos blocos em disco. O superbloco não é pedido, mas deve ser montado pelo aluno para ajudar a responder.

Alocação de memória pelo LISP

código: 293
finalizado em 06/11/01
Descreve-se como o LISP aloca memória para as variáveis e as funções que são nele definidas. O conceito de célula, CONS e CAR de cada célula é descrito. Finalmente é dada uma lista LISP e pede-se que o aluno faça o mapeamento interno da memória, simulando o que o LISP faz. O exercício pede a representação interna da lista dada.

Simulação de pilhas

código: 311
finalizado em 02/05/00
Dadas os algoritmos de empilhamento e desempilhamento e dadas 5 pilhas e sugeridas 50 operações de empilhamento e desempilhamento nestas 5 pilhas pede-se, ao final, o que cada uma das 5 pilhas contém. O aluno deve responder a 15 perguntas sobre os conteúdos das pilhas.

Pilhas em baixo nivel

código: 312
finalizado em 02/05/03
São dados os algoritmos de criação, inclusão e exclusão em pilhas tanto seqüenciais como encadeadas, em baixo nivel (instruções semelhantes ao assembler). Diversas operações são feitas em um bloco de memória e ao final pede-se informações sobre como ficou esse bloco de memória. Este é um dos exercícios mais difíceis da coleção.

Notação polonesa reversa

código: 313
finalizado em 02/05/00
Uma aplicação de pilhas. São mostrados s algoritmos de conversão entre notação convencional (infixa) e as notações pré e pós-fixa. Depois é mostrado o algoritmo de cálculo de expressão pós-fixa. O exercício mostra uma expressão parentizada convencional e pede a expressão RPN correspondente. Depois pede que a expressão RPN seja calculada.

Passeio do rato no laberinto

código: 314
finalizado em 19/05/00
Uma aplicação de pilhas. Dado um laberinto, um rato é colocado em uma certa posição. Um algoritmo de busca da saída é dado. Pede-se a simulação do caminho percorrido pelo rato. A pilha é usada para memorizar posições de bifurcação, permitindo ao rato retornar quando encontrar becos sem saída.

Envoltória convexa de Graham

código: 315
finalizado em 13/01/03
Aplicação de pilhas. Dados um conjunto de pontos no plano, pede-se quais são os pontos

da envoltória convexa. (em outros termos: quais os pontos que seriam tocados por um elástico que englobasse o conjunto). É uma aplicação de geometria computacional interessante.

Construção de um compilador

código: 318
finalizado em 18/12/05
Uma aplicação de pilhas. Dadas uma linguagem fonte (estruturada, contendo enquanto, repita, para e se) e uma objeto (contendo apenas testes e desvios incondicionas e condicionais) e dada uma descrição do processo de compilação usado, dá-se um programa fonte e pede-se que o aluno simule a compilação apresentando o objeto gerado

Engenharia reversa de compiladores

código: 319
finalizado em 02/02/06
Este exercício descreve as regras de uma linguagem fonte e sua correspondente linguagem objeto, e dai pede duas coisas

1. Dado um programa objeto, qual foi o seu programa fonte ?

2. Dado um fonte e seu objeto compilado a menos de 2 comandos, quais são estes dois comandos ?

Deve ser usado logo após o exercício 318

Filas seqüenciais

código: 321
finalizado em 12/06/00
Dadas os algoritmos de enfileiramento e desenfileiramento e dadas 5 filas e sugeridas 50 operações nestas 5 filas pede-se, ao final, o que cada uma das 5 filas contém. O aluno deve responder a 15 perguntas sobre os conteúdos das filas seqüenciais.

Filas seqüenciais circulares

código: 322
finalizado em 12/06/00
Dadas os algoritmos de enfileiramento e desenfileiramento e dadas 5 filas circulares e sugeridas 50 operações nestas 5 filas pede-se, ao final, o que cada uma das 5 filas contém. O aluno deve responder a 15 perguntas sobre os conteúdos das filas sequenciais circulares.

Filas em baixo nivel

código: 323
finalizado em 03/05/03
São dados os algoritmos de criação, inclusão e exclusão em filas tanto seqüenciais circulares como encadeadas, em baixo nivel (instruções semelhantes ao assembler). Diversas operações são feitas em um bloco de memória e ao final pede-se informações sobre como ficou esse bloco de memória. Este é um exercícios bem difícil.

Ordenação topológica usando filas

código: 324
finalizado em 30/05/00
Dados todos os componentes de roupa que uma pessoa normal veste em um dia de frio e chuva, pede-se ao aluno que ache a seqüência de operações para vestir-se. O algoritmo de ordenação topológica é mostrado no exercício.

Truco em listas encadeadas

código: 331
finalizado em 27/06/00
Uma memória encadeada contendo os dados de uma partida de truco é mostrada. Pede-se o resultado das 3 rodadas e da partida, a partir dos dados da memória e das regras oficiais (?) do truco. Este exercício é uma festa em sala de aula.

4 erros em listas encadeadas

código: 332
finalizado em 23/02/01
São descritos os erros: loop na lista, endereços de próximo registro inválidos, junção de 2 listas distintas e dados inválidos na lista. Dada um bloco de memória que contém 4 listas (cada uma com um dos erros acima) pede-se que o aluno localize cada lista e cada erro. Não é um exercício trivial.

Criação e manuseio de 5 listas encadeadas

código: 333
finalizado em 01/08/00
Uma biblioteca contendo 25 livros com 5 encadeamentos (título do livro, idioma, preço, páginas e cidade de edição) é mostrada. Pede-se que o aluno inclua 2 novos livros, acertando os 5 encadeamentos de cada livro. Usam-se cursores ao invés de apontadores para permitir a execução e correção do exercício.

Listas duplamente encadeadas em baixo nivel

código: 341
finalizado em 29/04/01
Descreve-se uma lista duplamente encadeada e são dados os algoritmos de manutenção (os algoritmos não cabem na folha de exercício, eles estão apenas no texto da aula). Após 13 operações pede-se como ficou o bloco de memória.

Manuseio de listas duplamente encadeadas

código: 342
finalizado em 01/08/00
São mostradas 3 listas: de paises, de idiomas e de cidades, as 3 com encadeamentos duplos usando cursores. Feitas inclusões e exclusões nas 3 listas, pede-se ao final como elas ficaram.

Representação e caminhamento em árvores binárias

código: 411
finalizado em 07/08/00

Este exercício mostra 3 representações de árvores binárias (usando cursores, usando apontadores em um bloco de memória e usando parênteses. O exercício mostra 3 árvores (uma em cada metodologia de representação) e pede que o aluno faça os 3 caminhamentos (em-ordem, pré-ordem e pós-ordem) nas 3 árvores apresentadas.

Conversão de árvores n-árias em binárias

código: 412
finalizado em 01/07/02
O exercício mostra uma árvore de grau 6 usando cursores e pede que o aluno escreva uma árvore binária equivalente.

Balanceamento em árvores ABP

código: 421
finalizado em 14/07/00
Um desenho de uma ABP desbalanceada é apresentado pedindo-se ao aluno que refaça o desenho balanceando a árvore. Depois disso pede-se o caminhamento em largura da árvore.

Caminhamento e inclusões em ABPs

código: 431
finalizado em 11/03/00
Uma árvore é apresentada através de sua matriz de cursores. Pede-se que o aluno a desenhe e a seguir responda diversas questões sobre ela.

ABPs: busca, criação e altura

código: 432
finalizado em 18/06/02
Cerca de 8 ABPs devem ser construídas a partir de uma alimentação sequencial de itens. A seguir sobre cada uma das árvores são feitas diversas perguntas que devem ser respondidas.

Inclusões e exclusões em ABPs

código: 433
finalizado em 07/08/00
Os algoritmos de inclusão e de exclusão são mostrados. A seguir 3 árvores compartilhando a mesma matriz de cursores são mostradas. 28 operações são aplicadas sobre as árvores. Ao final pede-se o caminhamento sobre as 3 árvores.

Árvore binária de pesquisa com as letras do nome

código: 434
finalizado em 11/03/00
Pede que o aluno construa uma árvore de pesquisa binária com as letras de seu nome. Ainda na versão 1.

Árvores binárias de partilha

código: 435
finalizado em 04/08/03
As ABPas são estruturas que associam a eficiência de uma ABP equilibrada garantindo ao mesmo tempo que as chaves mais acessadas estarão mais próximas da raiz do que as menos acessadas. Com isso a complexidade média, que é $O(log\ n)$ no caso das ABPs torna-se ainda menor. Os algoritmos de criação e de busca são mostrados e o aluno deve construir uma ABPa.

Tries

código: 436
finalizado em 09/08/03
As tries são árvores muito eficientes para funcionar como dicionários. A folha mostra 2 implementações possíveis (usando-se árvores de grau k, onde k é a cardinalidade do alfabeto e usando-se árvores binárias). É dado um pequeno alfabeto e o aluno deve construir as 2 tries.

Aquecimento em árvores B

código: 441
finalizado em 02/07/02
Apenas exemplos de árvores B são mostrados sem maior preocupação com o rigor na definição. Algumas inclusões *ad hoc* são feitas e o aluno é convidado a antecipar alguns resultados.

Árvore B com grau t=3

código: 442
É apresentada a definição formal da árvore B e os algoritmos de criação são mostrados. O aluno é convidado a construir uma árvore B de grau t=3 com 40 números.

Simulando índices do SYBASE

código: 443
Um exemplo real de árvores B. É dado um arquivo com 8 registros e 5 índices, sendo um esparso e 4 não granulados. O aluno deve construir as páginas de índices do SYBASE (que são árvores B) e responder a perguntas sobre elas.

xBase

código: 444
finalizado em 10/08/01
O ambiente xBase (originalmente criado pelo dBase) é descrito com seus arquivos DBF, DBT e NDX. Este último (que é uma árvore B) é descrito com algum detalhe. Finalmente os alunos recebem um *dump* de memória contendo um arquivo NDX real e devem responder a perguntas sobre ele.

B árvore com t=5

código: 445
Uma B-árvore com t=5 é apresentada parcialmente preenchida e o aluno é convidado a

 ©88-08, Pedro Kantek

nela incluir mais 20 chaves. Depois são feitas perguntas sobre a árvore que já sofreu as inclusões.

Codificação e decodificação

código: 450
finalizado em 13/03/01
Este exercício é uma preparação para a compressão de Huffman. Solicita conversões entre binário, hexadecimal, ASCII. São criados 2 códigos novos: o ORTELINO (no qual cada caracter usa 5 bits) e o TROCALETRA (no qual cada caracter usa um número variável de bits. No mínimo 4 e no máximo 8 bits por caracter). Conversões de um para o outro e do outro para o um são pedidas.

Compressão usando o algoritmo de Huffman

código: 451
finalizado em 01/01/93
O algoritmo de Huffman é apresentado através de um simples exemplo. As etapas do algoritmo são descritas. Finalmente, apresenta-se um universo comum a todos os exercícios (o que garantirá a mesma árvore para todos) e pede-se a compressão e a descompressão usando essa árvore de 2 falas de peças de Shakespeare.

Huffman, árvores distintas

código: 452
finalizado em 01/01/94
Na continuação do exercício 451, novas frases de Shakespeare devem ser convertidas e desconvertidas. Mas agora, cada aluno terá um universo de caracteres diferentes, fazendo com que cada um tenha que construir a sua própria árvore.

Huffman, árvore dinâmica

código: 453
finalizado em 16/02/04
A construção de uma árvore dinâmica de Huffman é mostrada passo a passo ao mesmo tempo que os textos vão sendo comprimidos e/ou descomprimidos. Agora as mensagens são curtas (tipicamente 8 caracteres). Os algoritmos não são mostrados, estando apenas no texto da aula.

Heap - construção e pesquisa

código: 461
Um HEAP é formalmente definido e são pedidos 4 exercícios aos alunos abordando todas as facetas de um heap.

Algoritmos de ordenação - 1^{a}parte

código: 511
finalizado em 03/01/03
Apresenta os algoritmos de ordenação bolha, bolha-estrela, inserção e seleção. Comparativos entre os 4 métodos em termos de tempo são mostrados. O aluno deve responder a questões sobre o estado das variáveis internas alguns passos após o início de cada algoritmo.

©88-08, Pedro Kantek

Algoritmos de ordenação - 2^{a}parte

código: 512
finalizado em 04/01/03
Apresenta os algoritmos de ordenação shell, heap e quick fazendo um comparativo entre eles e com os algoritmos do exercício 511. O aluno deve responder a questões sobre o estado das variáveis internas alguns passos após o início de cada algoritmo.

Redes de comparação e de ordenação

código: 513
finalizado em 02/01/03
Estabelece a discussão de ordenação em ambientes multiprocessados, baixando o melhor custo da ordenação que era $O(n \times log_2\ n)$ para $O(log_2^2\ n)$. Perguntas são feitas sobre o estado da rede de ordenação.

Tabelas de dispersão: hash tables

código: 521
Este exercício introduz o conceito de tabelas de dispersão e pede ao aluno que construa duas delas: a primeira com áreas primária e secundária separadas e com encadeamento entre elas. A segunda com apenas área primária e dispondo as colisões à direita do elemento destino. Discussões sobre funções hash adequadas também são conduzidas.

Busca em cadeias: algoritmo de Boyer-Moore

código: 531
finalizado em 24/01/02
O exercício começa descrevendo o algoritmo de força bruta para realizar esta tarefa, mostrando a quantidade de testes necessários para conduzir a pesquisa em uma pequena frase. A seguir as características do algoritmo de Boyer-Moore são apresentadas. Finalmente uma busca é conduzida usando este algoritmo. O exercício permite concluir pela superioridade do método apresentado.

Busca em tabelas

código: 532
finalizado em 01/09/03
Este é um exercício importante da série ao permitir a comparação de diversos métodos de busca em tabelas. São mostrados o método de busca linear, o de busca linear em tabela ordenada, o de busca linear com sentinela, o busca através de tabelas hash, a busca binária, e a busca usando uma árvore binária de pesquisa. Dados numéricos para um determinado problema são mostrados, comentados e comparados. Finalmente, os alunos são convidados a usar todos os métodos e apresentar a quantidade de testes em cada um.

Grafos usando listas encadeadas de arestas

código: 611
finalizado em 14/07/02
Um ambiente de descrição de grafos usando arestas encadeadas em uma estrutura similar a uma memória é mostrado. A seguir pede-se que o aluno estabeleça a matriz de caminhos mínimos (ainda de maneira intuitiva: é fácil, o grafo tem apenas 6 vértices

e 20 arestas). Depois pergunta-se quantos pares de vértice tem caminhos que custam menos do que a ligação direta e qual o diâmetro (o maior valor dentro da matriz de caminhos mínimos) do grafo.

Representação e caminhamento em grafos

código: 612
O exercício mostra a representação de grafos através de matrizes de adjacência e de incidência. Depois são mostrados algoritmos de caminhamento em profundidade e em largura. É dado um grafo de 14 vértices e pede-se que o aluno monte os caminhos em largura e em profundidade.

Caminho mínimo: algoritmo de Dijkstra

código: 621
O algoritmo é mostrado e são feitas 2 simulações: uma pequena com um grafo de 5 vértices e uma maior com um grafo de 10 vértices. A seguir o aluno deve fazer o mesmo com um grafo de 10 vértices.

Caminho mínimo 1:1

código: 622
Pede que o aluno calcule o caminho mínimo entre dois vértices de um dado grafo. Ainda na versão 1.

Caminho mínimo: algoritmo de Floyd-Warshall

código: 623
Este algoritmo calcula os caminhos mínimos de todos os vértices para todos os vértices, tendo complexidade $O(n^3)$. Primeiro é apresentado um exemplo simplório de 3 vértices e depois outro maior de 6 vértices que é feito em conjunto professor e alunos. Finalmente, os alunos são convidados a resolver um grafo de 7 vértices, devendo responder o valor do caminho para 5 pares de vértices aleatoriamente escolhidos.

Caminho mínimo: roteamento através de Floyd modificado

código: 624
O algoritmo de roteamento de Floyd é mostrado e um exemplo de 5 vértices é resolvido. Os alunos são convidados a resolver um grafo de 7 vértices devendo responder a 3 perguntas sobre custos dos caminhos e a 3 perguntas sobre as rotas dos caminhos achados. Este exercício deve ser feito após o exercício 623.

Cálculo de fluxos: Ford-Fulkerson, Edmonds-Karp

código: 632
finalizado em 15/07/02
Descreve-se o problema de análise de fluxos. É como se um grafo indicasse a vazão de conexões entre pontos diversos de uma planta. A pergunta a responder é qual a vazão máxima entre 2 pontos da planta ? É resolvida uma instância de 7 pontos e pede-se que o aluno resolva outra instância inédita de 7 pontos.

Ordenação topológica usando grafos

código: 633
finalizado em 22/10/01
O sequenciamento de um curso de informática hipotético é apresentado com suas disciplinas e pré-requisitos. A seguir o algoritmo que o resolve é apresentado e o aluno deve achar qual o caminho ideal (a seqüência) de disciplinas que deve ser cursada.

Árvore de cobertura mínima: algoritmo de Kruskal

código: 634
finalizado em 18/01/02
A árvore de cobertura mínima é o menor grafo que conecta todos os pontos de uma determinada planta. Sempre tem a característica de uma árvore, daí a razão do nome. Um algoritmo (guloso) é mostrado e o aluno convidado a calcular uma ACM para um grafo de 10 vértices.

Coloração de grafos

código: 635
finalizado em 19/08/03
A característica NP do problema é ressaltada. A seguir um exemplo real de tempos de um semáforo em um cruzamento hipotético de 5 ruas é mostrado. Depois o algoritmo (guloso) é mostrado e o aluno convidado a colorir um grafo de 12 vértices.

Aplicação de grafos: quebra cabeça dos tijolos

código: 636
finalizado em 29/10/05
Um dominó contendo 10 peças e 3 animais em cada peça é apresentado. Mostra-se como montá-lo em um dominó. O problema pode ser convertido em buscar um caminho hamiltoniano em um grafo de 10 vértices (1 a cada peça) e com 15 arestas (cada vértice está ligado a outros 3, representando os animais de cada peça. Se o problema puder ser representado em um grafo de Petersen, não haverá solução, caso contrário haverá. O aluno é convidado a tentar resolver 2 dominós diferentes.

Método PERT-CPM

código: 640
finalizado em 15/02/06
Este exercício propõe um projeto formado de eventos e tarefas, estas com suas durações e principalmente com a lista de precedências (uma tarefa só pode ser começada depois que outras terminarem). O exercício ensina o algoritmo do caminho crítico e pede que o aluno calcule a duração do projeto proposto e quais os eventos (e tarefas) que estão no caminho crítico

Algoritmo Húngaro

código: 650
finalizado em 01/05/07
Um problema de otimização combinatória. Trata-se de uma lista de pessoas e de uma lista de tarefas. Cada combinação pessoa-tarefa tem um custo associado. O algoritmo informa, ao final, qual a combinação que garante custo mínimo ao conjunto.

©88-08, Pedro Kantek

Torneio de Tenis

código: 660
finalizado em 07/08/07
Um exemplo da estratégia dividir para conquistar. Trata-se de construir uma tabela de jogos envolvendo todos os oponentes jogando entre si no menor número possível de dias. O exercício não pede a tabela, já que esta pedida é muito dificil de ser feita sem implementação (fica como ótimo desafio), mas apenas a seqüência de chamadas recursivas à função.

Além destes exercícios gerados pelo pacote Algoritmos Vivos, haverá outros extraídos da literatura ou propostos ad-hoc.

©88-08, Pedro Kantek

1.1 Bibliografias para Estruturas de Dados

Wir86 WIRTH, Niklaus. Algoritmos e Estruturas de Dados. Tradução do livro [Wir76].

Wir76 WIRTH, Niklaus. Algorithms + Data Structures = Programs. Um dos melhores livros de estruturas de dados e algoritmos.

Szw94 SZWARCFITER, Jayme e MARKENSON, Lilian. LTC.

Excelente referência em árvores. Boa lista de exercícios.

Cor02 CORMEN, Thomas et alli. Algoritmos, Teoria e Prática. Editora Campus.

O livro usado por 9 entre 10 universidades no mundo.

Knu73a KNUTH, Donald. The Art of Computer Programming. Vol 1: Fundamental Algorithms. Addison Wesley Publishing Co.

Knu73b KNUTH, Donald. The Art of Computer Programming. Vol 2: Seminumerical Algoritms. Addison Wesley Publishing Co.

Knu73c KNUTH, Donald. The Art of Computer Programming. Vol 3: Sorting and Searching. Addison Wesley Publishing Co.

Os livros seminais na área.

Ata99 ATTALAH, Mikhail. (org). Algorithms and Theory of Computation Handbook. CRC Press.

Talvez o mais completo e moderno livro de algoritmos.

Aho83 AHO, Alfred et alli. Data Structures and Algorithms.

Excelente texto.

Dew89 DEWDNEY, A.K. The Turing Omnibus. Computer Science Press.

um livro iluminado. Apresenta cerca de 60 tópicos com profundidade e clareza.

Laf99 LAFORE, Robert. Aprenda Estrutura de Dados e algoritmos em 24 horas. Editora Campus.

Exemplos dados em C++. Acompanha CD com exemplos escritos em JAVA. Alguma confusão conceitual.

Ziv93 ZIVIANI, Nívio. Projeto de Algoritmos com implementações em Pascal e C.

Bom livro introdutório.

Vil93 VILLAS, Marcos Vianna et alli. Estruturas de Dados. Editora Campus.

Pouco ambicioso.

Sch97 SCHILDT, Herbert. C Completo e Total. Editora Makron Books.

possui alguns códigos em C bastante interessantes.

Kyl00 KYLE Loudon. Dominando algoritmos com C. Editora Ciência Moderna.

Exemplos todos em C. Se aproveita alguma coisa.

Ten95 TENNENBAUM, Aaron et alli. Estruturas de Dados usando C. Makron Books.

privilegia a linguagem em detrimento do algoritmos, mas...

Kru84 KRUSE, Robert. Data Structures and Program Design. Prentice Hall.

Texto introdutório.

Ter91 TERADA, Routo. Desenvolvimento de algoritmos e estruturas de dados.McGraw-Hill, 1991.

Livro telegráfico, mas bom na área de complexidade.

Vel86 VELOSO, Paulo et alli. Estruturas de Dados. Editora Campus.

Bem despretencioso.

San01 SANTOS, Clésio Saraiva dos. Tabelas: Organização e Pesquisa. Série Livros Didáticos da UFRGS. Ed. Sagra Luzzato.

Boa apresentação do assunto tabelas.

Sed88 SEDGEWICK, Robert. Algorithms. Addison Wesley Publishing Co.

bom livro.

Lips86 LIPSCHUTZ, Seymour. Data Structures. Série Schaum. Ed McGraw Hill.

bons exercícios.

Sin98 SINGH, Simon. O último Teorema de Fermat. Ed. Record.

Um romance magnífico sobre a história da matemática.

Sin01 SINGH, Simon. O Livro dos Códigos. Record.

Excelente guia introdutório à criptografia.

Mao03 MAOR, Eli. e: A História de um número.

Boa descrição dos logaritmos e do cálculo.

Str97 STRATHERN, Paul. Turing e o computador em 90 minutos. Jorge Zahar Editora. (Unicenp: 004.S899t = 4.andar).

O autor, pelo autor

Meu nome é Pedro Luis Kantek Garcia Navarro, conhecido como Kantek, ou Pedro Kantek. Nasci em Curitiba há mais de 50 anos. Sou portanto brasileiro, curitibano e coxa-branca com muito orgulho, mas sendo filho de espanhóis (meus 7 irmãos nasceram lá), tenho também a nacionalidade espanhola. Aprendi a falar em *castellano*, o português é portanto meu segundo idioma. Estudei no Colégio Bom Jesus e quando chegou a hora de escolher a profissão, lá por 1972, fui para a engenharia civil, mas sem muita convicção. Durante a copa do mundo de futebol de 1974 na Alemanha, ao folhear a Gazeta do Povo, achei um pequeno anúncio sobre um estágio na área de processamento de dados (os nomes informática e computação ainda não existiam). Lá fui eu para um estágio na CELEPAR, que hoje, mais de 30 anos após, ainda não acabou. Na CELEPAR já fui de tudo: programador, analista, suporte a BD (banco de dados), suporte a TP (teleprocessamento), coordenador de auto-serviço, coordenador de atendimento, ... Atualmente estou na área de governo eletrônico. Desde cedo encasquetei que uma boa maneira de me obrigar a continuar estudando a vida toda era virar professor. Comecei essa desafiante carreira em 1976, dando aula num lugar chamado UUTT, que não existe mais. Passei por FAE, PUC e cheguei às Faculdades Positivo em 1988. Sou o decano do curso de informática e um dos mais antigos professores da casa. Na década de 80, virei instrutor itinerante de uma empresa chamada CTIS de Brasília, e dei um monte de cursos por este Brasil afora (Manaus, Recife, Brasília, Rio, São Paulo, Fortaleza, Floripa, ...). Em 90, resolvi voltar a estudar e fui fazer o mestrado em informática industrial no CEFET. Ainda peguei a última leva dos professores franceses que iniciaram o curso. Em 93 virei mestre, e a minha dissertação foi publicada em livro pela editora Campus (Downsizing de sistemas de Informação. Rio de Janeiro: Campus, 1994. 240p, ISBN:85-7001-926-2). O primeiro cheque dos direitos autorais me manteve um mês em Nova Iorque, estudando inglês. Aliás, foi o quarto livro de minha carreira de escritor, antes já havia 3 outros (MS WORD - Guia do Usuário Brasileiro. Rio de Janeiro: Campus, 1987. 250p, ISBN:85-7001-507-0, Centro de Informações. Rio de Janeiro: Campus, 1985. 103p, ISBN:85-7001-383-3 e APL - Uma linguagem de programação. Curitiba. CELEPAR, 1982. 222p). Depois vieram outros. Terminando o mestrado, rapidamente para não perder o fôlego, engatei o doutorado em engenharia elétrica. Ele se iniciou em 1994 na UFSC em Florianópolis. Só terminou em 2000, foram 6 anos inesquecíveis, até porque nesse meio tive que aprender o francês - mais um mês em Paris aprendendo-o. Finalmente virei engenheiro, 25 anos depois de ter iniciado a engenharia civil. Esqueci de dizer que no meio do curso de Civil desistí (cá pra nós o assunto era meio chato...) em favor de algo muito mais emocionante: matemática. Nessa época ainda não havia cursos superiores de informática. Formei-me em matemática na PUC/Pr em 1981. Em 2003, habilitei-me a avaliador de cursos para o MEC. Para minha surpresa, fui selecionado e virei delegado do INEP (Instituto Nacional de Pesquisas Educacionais) do Governo Brasileiro. De novo, visitei lugares deste Brasilzão que sequer imaginava existirem (por exemplo, Rondonópolis, Luiziânia, Rio Grande, entre outros), sempre avaliando os cursos na área de informática: sistemas de informação, engenharia e ciência da computação. Atualmente estou licenciado da PUC e no UNICENP respondo por 4 cadeiras: Algoritmos (1. ano de sistemas de informação), Estrutura de Dados (2.ano, idem) e Tópicos Avançados em Sistemas de Informação (4.ano, idem), além de Inteligência Artificial (último ano de Engenharia da Computação). Já fiz um bocado de coisas na vida, mas acho que um dos meus sonhos é um dia ser professor de matemática para crianças: tentar despertá-las para este mundo fantástico, do qual – lastimavelmente – boa parte delas nunca chega sequer perto ao longo de sua vida.

©88-08, Pedro Kantek

O autor, há muitos anos, quando ainda tinha abundante cabeleira

©88-08, Pedro Kantek

Capítulo 2

Complexidade

2.1 Complexidade computacional

É a medida do desempenho estimado de um determinado algoritmo. A complexidade nos diz quanto de recurso é necessário para executar um algoritmo. Cuidado que não é esse o conceito trivial de complexidade. No dia-a-dia uma coisa é complexa se é difícil. Aqui uma coisa é complexa se consome muito recurso.

2.1.1 Como analisar um algoritmo

Existem basicamente alguns critérios para analise de um algoritmo, quais sejam:

Correção o algoritmo gera resultados corretos qualquer que seja a entrada ?

Generalidade o algoritmo se presta a uma classe ampla de entradas, ou só funciona para uma específica ?

Clareza e legibilidade Pode ser entendido por alguém que não o seu autor (ou até mesmo por este) ?

simplicidade o algoritmo é inteligível ou é uma demonstração da [suposta] habilidade intelectual do seu autor ?

2.1.2 Custo

O custo já teve maior importância em dias passados, quando computadores tinham menos recursos e eram muito mais caros. Hoje, tal interesse é menor, mas mesmo assim, importante. Existem algoritmos que para rodar exigem computadores quase infinitos, e é fácil perceber, que por maior que sejam tais máquinas, elas nunca serão infinitas.

Como se mede custo ? Basicamente em função dos requisitos de tempo de processamento e de espaço de memória alocada. A análise real dos requisitos de um algoritmo é complexa e nem sempre tem significado prático. (Pode mudar o computador, o SO, a linguagem, o compilador, os parâmetros de instalação,...). Prefere-se, ao invés disso ter uma medida aproximada, comparável e matematicamente tratável.

Uma característica importante é que a eficiência de um algoritmo é função do tamanho de sua entrada. Em geral tempo e espaço podem ser negociados (ou seja, é possível diminuir o tempo de um algoritmo aumentando o espaço alocado e vice versa). O espaço, costuma ser uma função linear - ou no mínimo de crescimento inferior ao do tempo

- do tamanho da entrada, e já que tempo e espaço podem - em princípio - se intercambiados, este estudo será centrado no tempo. Assim, quando nada se falar em contrário, a eficiência de um algoritmo será sempre em relação ao seu tempo de execução. Quanto ao tempo, a próxima questão é como medi-lo?

A tentação é a de buscar um cronômetro e começar a medir, mas para isso, precisamos ter os algoritmos funcionando implementados em um programa, e supostamente é isso que queremos evitar. Só pretendemos construir aquele que for mais eficiente. Portanto, nada feito. A próxima abordagem seria medir as instruções de máquina de cada um e inferir o seu tempo total. Também é muito complicado e impraticável. Resta a solução mais simples, que é a de propor um modelo simplificado de computador, ainda que teórico.

A primeira proposta poderia ser uma máquina de Turing. Como ele demonstrou qualquer problema computacional pode ser resolvido por uma máquina de Turing. Assim, para analisar um algoritmo bastaria construir a MT correspondente à sua solução e nela olhar:

a quantidade de estados o que nos daria o custo em espaço ocupado.

a quantidade de transições que a máquina realizasse para resolver uma instância do problema. Esta medida nos daria o custo em tempo.

Em que pese a sua absoluta generalidade e a possibilidade de medir ao mesmo tempo, tempo e espaço, a MT acaba não sendo prática já que algoritmos mesmo simples, acabam gerando MTs muito grandes e complicadas. Além disso uma MT não tem nenhuma similaridade com o programa real de computador que implementa a solução do algoritmo proposto

Usar-se-á um modelo (conhecido por alguns autores como Máquina RAM), no qual as unidades de tempo e de espaço são simplificadas. Eis as regras do modelo proposto

- Considerar apenas as instruções que são exaustivamente repetidas ao longo do algoritmo (tipicamente dentro de loops). Com isso desprezamos aquelas instruções que são executadas uma ou poucas vezes.
- Considerar cada uma dessas instruções repetidas como tendo tempo unitário. Isso nos livra de considerar a máquina, o sistema operacional, a linguagem etc etc.
- Considerar o espaço ocupado como $C \times q$, onde q é a quantidade de variáveis e C é o espaço ocupado por uma variável básica.

Veremos a seguir que tais simplificações não invalidam o tipo de análise que se fará a seguir. Antes de prosseguir, um alerta: em geral algoritmos mais simples e fáceis de entender e codificar tem menos eficiência do que algoritmos mais complexos e sofisticados (se não para escrever, pelo menos para entender). Essa ressalva é feita, porque tempo de computador não é tudo. Há que se considerar o tempo de escrever e depurar o programa, bem como o número de vezes em que este programa rodará. Mais ainda, se o tamanho da instância de entrada permanece pequeno, toda esta análise pode ser deixada de lado. Infelizmente, nada se pode dizer quanto ao valor desse "pequeno", mas ele pode ser tomado intuitivamente.

Em resumo: pequenas instâncias, ou programas que rodarão uma única vez, merecem os algoritmos mais simples que existam (desde que funcionem corretamente, bem entendido). Entretanto, quando o software é grande, crucial em alguma aplicação, complexo ou será executado milhões de vezes por dia, aí então vale a pena uma discussão sobre eficiência.

©88-08, Pedro Kantek

2.2 Complexidade Computacional

Computacionalmente, a eficiência de um algoritmo é medida por uma grandeza chamada complexidade computacional do algoritmo. Para este estudo a complexidade será uma medida inexata, porém digna de manipulações e comparações. A inexatidão decorre do fato de que medidas exatas demandariam trabalhos ciclópicos (tipicamente resolver inúmeras instâncias do problema, compará-las e obter uma lei geral de formação), além do que só seriam válidas para contextos muito determinados, o que de antemão impediria sua conveniente generalização. Mas, as dificuldades não devem nos desanimar. Parafraseando Graham,

> "não precisamos insistir em tudo ou nada" [Gra95, pág. 320].

Podemos supor que para um dado algoritmo que tem n entradas, (n podendo ser quantidade de registros, de pares de números, de símbolos em uma cadeia, de partições em um conjunto, não importa), terá um tempo de execução que pode ser representado por uma função de n. Tipicamente, quanto maior é n, também maior é esta $f(n)$. A dificuldade acima referida é quanto ao real formato e comportamento de $f(n)$, mas esse conhecimento poderá ser menosprezado aqui.

O que nos interessa é que pode-se formar uma série de funções típicas em n, na qual todas tenderão ao infinito quando n tender, mas *com velocidades diferentes*. É esse ritmo de crescimento que nos interessa. Essa série poderia ser

$$k \ll \log(n) \ll n \ll n \times \log(n) \ll n^2 \ll n^3 \ll 2^n \ll 3^n \ll n!n^n$$

Na qual o símbolo $\ll$ indica "cresce mais rápido". Usar-se-á a notação O (denotada $O(n)$, ou big-O de n), introduzida na matemática por Paul Bachmann em 1894.

Antes de estudar a função $O(n)$, vamos ver alguns exemplos intuitivamente

Seja o algoritmo

x ← x+1

Assim colocado, sua complexidade $O(n)$ é igual a 1, já que a instância da entrada (o número de variáveis x diferentes é de apenas uma).

Entretanto, se o algoritmo for

para k de 1 até 100 **faça**
  x ← x+1
fimpara

A complexidade $O(n)$ será n (n neste caso é 100), já que aumentando-se n aumenta-se de maneira proporcional o tempo de execução.

Se o algoritmo for

para k de 1 até 100 **faça**
  para j de 1 até 100 **faça**
    x←x+1
  fimpara{j}
fimpara{k}

A complexidade $O(n)$ será n^2 (n quadrado), já que aumentando-se n, o requisito de tempo aumenta com o quadrado de n.

Como comentado acima, note-se que se fizeram três simplificações importantes:

O tempo de $x \leftarrow x + 1$ é um. Não importa se segundos, milésimos de segundo ou o que quer que seja. Todas as instruções são iguais em termos de tempo, o que é uma simplificação e tanto (sabemos que $x + 3$, é dezenas de vezes mais rápido que $x \div 3$). Não são contadas todas as instruções. Apenas aquelas que sofrem iterações, ou seja, que são repetidas a cada ciclo.

 ©88-08, Pedro Kantek

2.2.1 Formalização

Diz-se que uma função $f(n)$ é da ordem de $g(n)$, e escreve-se $f(n) = O(g(n))$ se existirem dois inteiros positivos a e b, tais que $f(n) \leq a(g(n))$, para todo $n \leq b$. Assim posto, percebe-se que não existe uma única função $g(n)$, existem infinitas delas, variando a e b, bem como $g(n)$. Para esta análise buscar-se-á uma função $g(n)$ que tenha coeficiente unitário e que se aproxime o máximo possível de $f(n)$. Daqui, segue-se que se $f(n)$ for um polinômio, pode-se escolher a $g(n)$ como o termo de seu mais alto grau.

Por exemplo, suponhamos que $f(n) = n^3 + 4n^2 + 8$. Diante do acima exposto pode-se afirmar que $f(n)$ é $O(n^3)$, uma vez que quando n cresce, a importância do primeiro termo do polinômio, claramente suplanta as dos outros dois. Mas, atente-se que isto não é verdade para valores pequenos de n. No limite, se $n = 1$, o termo mais importante é $4n^2$ e não n^3, já que 4 é maior do que 1.

Costuma-se atribuir os nomes de ordem polinomial para função que tem $O(n^k)$, e ordem exponencial para função que tem $O(k^n)$. Segundo Tenenbaun,

> "devido à incrível taxa de crescimento das funções de ordem exponencial, os problemas que exigem algoritmos de tempo exponencial para sua solução são considerados intratáveis no atual ambiente de computação, ou seja tais problemas não podem ser solucionados com precisão, exceto nos casos mais simples". [Ten95, pág. 418].

A análise é complicada, devido ao fato de que o tempo gasto por um algoritmo não é - em geral - função única do tamanho de sua entrada. Outras características dessa entrada também devem ser levadas em conta. No caso mais famoso deste fenômeno, estão os algoritmos de classificação. O algoritmo A pode ter um desempenho melhor do que B se a ordem dos elementos a classificar é aleatória, mas em contrapartida, se apenas uma pequena parcela desses elementos, digamos 2% deles, se encontra fora de ordem, não seria de estranhar que B fosse muito melhor do que A. E, note-se que nos dois casos, o tamanho da instância de n é o mesmo nos dois casos.

2.3 Análise de Casos

Desse fato, nascem as análises de casos. Tipicamente podem-se estudar três: a pior (chamada pessimista), a aleatória (chamada média) e a melhor (chamada otimista). Tradicionalmente, as análises matemáticas do fenômeno concentraram-se nos casos pessimistas, pois eles representam o "teto"da complexidade, isto é, o seu valor máximo. Ainda hoje, essa análise é válida. Suponha que uma refinaria de petróleo informatizada, operando em tempo real, precisa uma varredura de sinais de temperatura em no máximo 50 milissegundos. De nada adianta garantir que o algoritmo A conseguirá tempos de 10 milissegundos em 99% dos casos, embora possa levar 100 milissegundos em 1% dos casos. Aqui, se necessita um algoritmo, digamos B que embora tenha tempo médio de 40 milissegundos (portanto aparentemente pior do que A), mas que consiga um tempo máximo de 50 milissegundos. Por razões óbvias, o caso pessimista precisa ser considerado, sob pena de ter que chamar os bombeiros. (e agüentar o prejuízo...).

Mas, nos problemas do dia-a-dia, não tão críticos assim, costuma-se usar a abordagem do tempo médio, uma vez que se nada podemos precisar quanto à maneira com que os dados de entrada virão, podemos presumir que eles se distribuirão aleatoriamente, e casos demorados serão compensados por outros casos rápidos. A análise otimista não desperta muito interesse teórico.

Para encerrar esta abordagem conceitual, definem-se duas cotas superiores para um dado problema. A chamada cota superior de complexidade, é a complexidade do melhor algoritmo que resolve o problema. Denota-se essa cota por $CS(n)$. Para certos

©88-08, Pedro Kantek

problemas, entretanto pode-se definir a cota inferior de complexidade, chamada $CI(n)$, que é inerente e intrínseca ao problema considerado, isto é, independe do algoritmo utilizado, e garantidamente nunca poderá ser ultrapassada para baixo, por qualquer outro algoritmo que venha a ser inventado. A $CI(n)$ é característica do problema, enquanto $CS(n)$ é característica do melhor algoritmo (disponível até o momento) que resolve o problema. Obviamente, o objetivo final dos cientistas da computação é igualar as duas complexidades, dado que, para inúmeros problemas hoje existentes $CS(n) > CI(n)$.

Os símbolos usados: pessimista $\rightarrow \Omega$, exata (entre dois limites conhecidos) $\rightarrow \Theta$ e melhor $\rightarrow O$. Para mais detalhes, veja a figura 3.1 (pág 34) do livro do Cormen (Cor02). Veja também a nota do capítulo à pág. 49.

2.3.1 Classes

Diante do exposto, os diversos problemas a resolver mediante algoritmos computacionais podem ser incluídos em uma de 4 classes:

- Problemas com complexidade linear, ou seja, com $O(n)$. (note que este caso é um particular da classe seguinte, a polinomial)
- Problemas com complexidade polinomial, ou seja, com $O(n^x)$, onde $x > 1$. Este tipo de problema é conhecido como de classe P (de polinomial)
- Problemas que aparentemente não tem complexidade exponencial, mas para os quais não se conhecem (ainda) algoritmos com complexidade polinomial. Este tipo de problema pertence à classe NP (de não polinomial).
- Problemas que intrinsecamente requerem uma quantidade exponencial de operações, ou seja, com $O(k^n)$.

Uma boa aproximação trivial para as classes linear e polinomial (n^2) é dada pelas nossas operações triviais da matemática. Vejamos

Quanto custa somar dois números de n dígitos ? Resposta: n

Quanto custa multiplicar dois números de n dígitos ? Resposta n^2.

2.3.2 O problema P versus NP

Este é um dos problemas do milênio: provar se $P = NP$ ou se $P \neq NP$. Sua solução vale US$ 1.000.000 [1] A classe NP acima citada, foi uma tentativa de criar uma classe intermediária entre a polinomial e a exponencial.

Duas estratégias são usualmente usadas para resolver instâncias não triviais deste problema

- contentar-se com soluções aproximadas, usando por exemplo, o algoritmo guloso
- olhar características do problema, buscando informações externas ao modelo para simplificá-lo, eliminando algumas alternativas por absurdas.

Por exemplo, em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

[1] vide em www.claymath.org as regras do prêmio e a descrição formal deste e de mais 6 problemas: a hipótese de Riemann, a teoria de Yang-Mills (hipótese de lacuna de massa), as equações de Navier-Strokes, a conjectura de Poincaré, a conjectura de Birch, Swinnerton e Dyer e a conjectura Hodge)

Fazem parte desta, problemas cuja verificação de instâncias do problemas é simples, mas que são tão demorados pela necessidade de examinar quantidades estupendas de alternativas (o problema do caixeiro viajante é um bom exemplo).

O nome oficial da classe NP é **Processos em tempo não deterministicamente exponencial**. O uso da palavra "não determinístico" em um ambiente de computadores (os seres determinísticos por natureza) sugere que esta classificação é teórica. A idéia é imaginar um computador que diante de muitas escolhas, faça sempre a mais adequada (seria um computador sortudo). Imaginando existir este computador, a solução desta classe de problemas seria polinomial, já que mesmo escolhendo bem, ele ainda teria que verificar se o seu palpite foi o correto.

Embora a classe seja teórica, ela é importante por duas razões:

1. A grande maioria dos problemas exponenciais que surgem na indústria e na administração é do tipo NP
2. Em 1971, Stephen Cook escreveu um artigo mostrando um obscuro problema NP e demonstrando matematicamente que se o seu problema pudesse ser resolvido em tempo P, outros problemas NP também o poderiam. Ele chamou esta propriedade de completude NP.

Em resumo: um problema é NP-completo se a descoberta de uma eventual solução P para ele implicar que todos os problemas NP-completos poderão usar também uma solução P.

Mais tarde, Richard Karp mostrou que a maioria dos problemas NP até então conhecidos (o do caixeiro viajante, o de seqüenciamento de tarefas, etc) eram NP-completos.

Lembrar se ainda não se sabe se $P = NP$ ou se $P \neq NP$. Essa dúvida ainda vale 1000 milhares de doletas.

Uma tabela comparativa

#	Complexidade esperada	Tempo esperado em função do tamanho da entrada	ex	tempo para n=10	tempo para n=100
1	1	não importa o tamanho de n, o tempo é constante e igual a 1	1	1	1
2	$\log_2 n$	Para um tamanho n, o algoritmo demora $\log_2 n$	2	3.3	6.6
3	n	Tempo linear em função de n	3	10	100
4	$n.\log_2 n$	Tempo linear multiplicado pelo logaritmo na base 2 de n	4	33.2	664.4
5	n^2	Tempo quadrático em função de n	5	100	10^5
6	n^3	Tempo cúbico em função de n	6	1.000	10^6
7	2^n	2 elevado à potência n	7	1024	12.10^{29}
8	3^n	3 elevado à potência n	8	59049	51.10^{46}
9	$n!$	fatorial de n	9	3628800	93.10^{156}
10	n^n	n elevado à n	10	1010	10^{200}

Exemplos citados na tabela anterior

1. 1: imprima X[0]
2. 1: **enquanto** N > 1 **faça**
   2: imprima X[N]
   3: N ← N div 2

```
    4: fimenquanto
3.  1: para K de 0 até N-1 faça
    2:    imprima X[K]
    3: fimpara
4.  1: para K de 0 até N-1 faça
    2:    J ← 1
    3:    enquanto J < N faça
    4:       ...
    5:       J ← J × 2
    6:    fimenquanto
    7: fimpara
5.  1: para K de 0 até N-1 faça
    2:    para J de 0 até N-1 faça
    3:       ...
    4:    fimpara
    5: fimpara
6.  1: para K de 1 até N faça
    2:    para J de 1 até N faça
    3:       para L de 1 até N faça
    4:          ...
    5:       fimpara
    6:    fimpara
    7: fimpara
7.  1: K ← 2^K
    2: para J de 0 até K-1 faça
    3:    ...
    4: fimpara
8.  1: K ← 3^K
    2: para J de 0 até K-1 faça
    3:    ...
    4: fimpara
9.  1: K ← 1
    2: para J de 2 até N faça
    3:    K ← K × J
    4: fimpara
    5: para L de 0 até K-1 faça
    6:    ...
    7: fimpara
10. 1: K ← 1
    2: para J de 1 até N faça
    3:    K ← K × N
    4: fimpara
    5: para L de 0 até K-1 faça
    6:    ...
    7: fimpara
```

Atente-se que esta tabela não apresenta as únicas complexidades possíveis. Como vimos, elas são infinitas. Os coeficientes do termo de mais alto grau de f(n) sempre foram tomados como unitários na tabela acima. Geralmente é isso que se faz, mas nada impede, que diante de um caso prático real, tais coeficientes sejam tomados no

 ©88-08, Pedro Kantek

seu valor real, que não necessariamente é 1. É a eterna polêmica entre generalidade e especificidade, sempre presente em estudos a respeito de desempenho computacional - e em muitos outros campos da computação.

A figura abaixo foi retirada da pág. 18 de [Trem84]

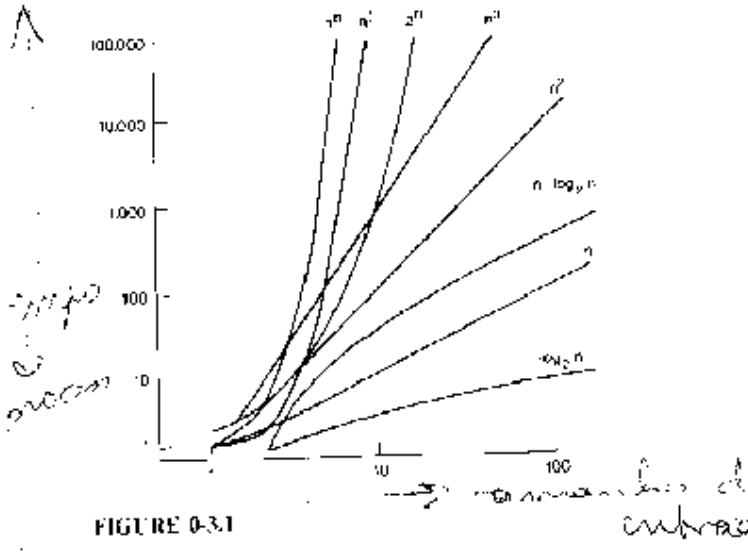


Figura 2.1: gráfico de desempenho em função do tamanho da entrada

Alguns exemplos numéricos

Verificar existência de duplos, comparando cada elemento com todos os outros

```
 1: Algoritmo
 2: inteiro I, K, NUM, CTD
 3: inteiro VET [100]
 4: leia VET
 5: para I de 0 até 99 faça
 6:    NUM ← VET[I]
 7:    CTD ← 0
 8:    para K de 0 até 99 faça
 9:       se VET[K] = NUM então
10:          CTD++
11:       fimse
12:    fimpara
13:    se CTD ≠ 1 então
14:       imprima NUM
```

©88-08, Pedro Kantek

15: **fimse**
16: **fimpara**

Este algoritmo tem desempenho de $O(n) = n^2$.

Fazer pesquisa binaria em um dicionário

1: Algoritmo PesqDicionário
2: Universo ← dicionário
3: **enquanto** universo > 1 página **faça**
4: Dividir universo em duas partes
5: **se** Palavra procurada < divisor do universo **então**
6: Universo ← primeira parte
7: **senão**
8: Universo ← segunda parte
9: **fimse**
10: **fimenquanto**
11: Pesquisa seqüencial na pagina

$O(n) = \log_2 n$ (onde n é o número de páginas do dicionário).

c. Somatório dos elementos do vetor M

1: Algoritmo teste
2: inteiro M [100]
3: inteiro I, SOM
4: SOM ← 0
5: leia M
6: **para** I de 0 até 99 **faça**
7: SOM ← SOM + M[I]
8: **fimpara**
9: imprima SOM

$O(n) = n$

Nestes 3 exemplos, vimos os 3 casos mais comuns de complexidade, que são: n, n^2 e $\log_2 n$.

Um contra exemplo

1: K ← 1
2: **enquanto** K $\neq$ 1 **faça**
3: imprima 33
4: **fimenquanto**

Claramente aqui há um erro de programação, já que K não tem o seu valor alterado dentro do enquanto. Se não fosse um erro, poder-se-ia falar em complexidade infinita, já o laço nunca se encerra.

Seja o seguinte trecho de código

1: J ← 1
2: **enquanto** (J ≤ N) **faça**
3: J ← J × B
4: execute tarefa...
5: **fimenquanto**

Supondo N e B inteiros e positivos, pergunta-se qual a complexidade associada a este trecho de algoritmo. A resposta passa pela análise do comportamento de J. Olhando seus valores, tem-se
$J = 1, B, B^2, B^3, B^4, ...$
Com isso, a complexidade $O(n)$ associada é $O(\log_B N)$.

Existem alguns algoritmos que tem complexidades associadas um pouco diferentes daquelas tradicionais. Veja-se por exemplo, o algoritmo "método de fatoração de campo

 ©88-08, Pedro Kantek

numérico para números primos", desenvolvido por Pollard, há não muitos anos. Já foi usado para fatorar com sucesso o número RSA-130 (um número primo com 130 dígitos de comprimento). É o método mais poderoso para isso e sua complexidade é $O(e^{c(\log n)^{1/3}(\log\log n)^{2/3}})$

Existem 3 valores relevantes para c, a depender do número que se quer fatorar. Tais valores são $c = 1.526285...$, $c = 1.922999...$ e $c = 1.901883$ (Fonte: Eric Weinsstein´s world of mathematics)

EXERCÍCIO 1 Preencha a tabela a seguir, para os valores sugeridos de N e para as principais complexidades apresentadas

N	1	$\log_2 N$	N	$N.\log_2 N$	N^2	N^3	2^N
2							
3							
4							
5							
10							
20							
30							

EXERCÍCIO RESOLVIDO 1 Escreva o algoritmo de um método (ou função) que receba um vetor de n números de ponto flutuante, calcule e devolva o vetor das médias prefixadas do vetor de entrada.

Definição: O vetor de médias prefixadas do vetor $[1..n]$ é o vetor $B[1..N]$ onde

$$B[i] = \frac{\sum_{j=1}^{j=i} A_j}{i}$$

Por exemplo, seja o vetor$A = 1, 2, 3, 4, 5$, seu vetor de médias prefixadas B será $B = 1, 1.5, 2, 2.5, 3$.

Estratégia 1

Seja o algoritmo

Entrada: Um vetor X contendo n inteiros
Saída: Um vetor R contendo n médias (ponto flutuante)

1: pflutuante R[n] função MédiaPrefixada (int X[n])
2: inteiro i,j,a
3: $i \leftarrow 1$
4: **enquanto** $i \leq n$ **faça**
5: $a \leftarrow 0$
6: $j \leftarrow 1$
7: **enquanto** $j < i$ **faça**
8: $a \leftarrow a + X[j]$
9: $j++$

©88-08, Pedro Kantek

10: **fimenquanto**
11: $R[i] \leftarrow a \div i$
12: $i++$
13: **fimenquanto**

A pergunta é: qual a complexidade do algoritmo acima ? Existem 2 laços no algoritmo. O mais externo varia entre 1 e n, gerando uma complexidade $O(n)$. Já o mais interno varia entre 1 e i. Na média, i chegará até a metade de n, gerando $O(n/2)$. Multiplicando ambas e extraindo a constante, chega-se à complexidade $O(n^2)$.

Estratégia 2

Seja agora um segundo algoritmo, considerando o seguinte fato: Ao calcular a media prefixada $A[i]$, temos a seguinte fórmula $A[i] = (X[1] + X[2] + ... + X[i]) \div i$.

Para o cálculo da média prefixada de $A[i+1]$, a fórmula é $A[i+1] = (X[1] + X[2] + ... + X[i] + X[i+1]) \div i + 1$

A constatação óbvia é que a soma de todas as parcelas da entrada anteriores ao índice atual pode ser guardada e não precisa ser recalculada a cada vez. Com isto, vai-se ao segundo algoritmo:

Entrada: Um vetor X contendo n inteiros
Saída: Um vetor R contendo n médias (ponto flutuante)
1: pflutuante R[n] função MédiaPrefixada (int X[n])
2: pflutuante s
3: $i \leftarrow 1$
4: $s \leftarrow 0$
5: **enquanto** $i \leq n$ **faça**
6: $s \leftarrow s + X[i]$
7: $R[i] \leftarrow s \div i$
8: $i++$
9: **fimenquanto**

Para este segundo algoritmo, a complexidade esperada é apenas $O(n)$. Comparando-se os dois desempenhos para um vetor de 100.000 números, no primeiro caso, ter-se-iam $100.000 \times 50.000 = 5.000.000.000$ operações, enquanto no segundo seriam apenas 100.000 operações. Em um computador que fizesse 10.000 operações por segundo, o primeiro algoritmo rodaria em 138 horas, enquanto o segundo o faria em 10 segundos. E, ambos calculam rigorosamente a mesma coisa.

EXERCÍCIO RESOLVIDO 2 Seja o problema de calcular a seqüência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Sua definição formal:
$F(1) = 1$
$F(2) = 1$
$F(n) = F(n-2) \times F(n-1)$

 ©88-08, Pedro Kantek

Solução Recursiva
1: inteiro função FIB(inteiro n)
2: **se** n < 3 **então**
3: devolva 1
4: **senão**
5: devolva FIB(n-2) + FIB(n-1)
6: **fimse**
Comentário: funciona, mas é terrivelmente ineficiente pelo recálculo exagerado e desnecessário.

Solução Iterativa
1: inteiro função FIB2(inteiro n)
2: inteiro I
3: inteiro X[n]
4: X[1] $\leftarrow$ X[2] $\leftarrow$ 1
5: **para** I = 3 até n **faça**
6: X[n] $\leftarrow$ X[n-2] + X[n-1]
7: **fimpara**
8: devolva X[n]
Comentário: funciona, tem complexidade $O(n)$, mas exige um vetor de n posições.

Solução Iterativa Melhor
1: inteiro função FIB3(inteiro n)
2: inteiro A, B, C
3: A $\leftarrow$ B $\leftarrow$ 1
4: **para** I = 3 até n **faça**
5: C $\leftarrow$ A + B
6: A $\leftarrow$ B
7: B $\leftarrow$ C
8: **fimpara**
9: devolva A
Comentário: funciona, tem complexidade $O(n)$, só exige 3 inteiros.

EXERCÍCIO 2 Para cada algoritmo a seguir, infira qual a complexidade esperada na sua execução:

a)
1: **para** K de 1 até N **faça**
2: **para** J de K até N **faça**
3: imprima 22
4: **fimpara**
5: **fimpara**

b)
1: K $\leftarrow$ 2
2: **enquanto** K $<$ N **faça**
3: **para** J de 1 até K **faça**
4: imprima 22
5: **fimpara**
6: K $\leftarrow$ K $\times$ K
7: **fimenquanto**

c)
1: K $\leftarrow$ N

©88-08, Pedro Kantek

```
2: enquanto K > 0 faça
3:    J ← N
4:    enquanto J > 0 faça
5:       imprima 22
6:       J ← J - 2
7:    fimenquanto
8:    K ← K - 4
9: fimenquanto
```

d)

```
1: K ← N
2: repita
3:    K–
4:    J ← K × K
5:    para I de 1 até J faça
6:       imprima 22
7:    fimpara
8: até K < (N div 2)
```

e) [Aho83, pág. 33]

```
1: para i de 1 até n faça
2:    para j de 1 até n faça
3:       C[i,j] ← 0
4:       para k de 1 até n faça
5:          C[i,j] ← C[i,j] + A[i,k] × B[k,j]
6:       fimpara
7:    fimpara
8: fimpara
```

f) idem

```
1: para i de 1 até n-1 faça
2:    para j de i+1 até n faça
3:       para k de 1 até j faça
4:          1+1
5:       fimpara
6:    fimpara
7: fimpara
```

g) idem

```
1: para i de 1 até n faça
2:    se impar(i) então
3:       para j de i até n faça
4:          x++
5:       fimpara
6:       para j de 1 até i faça
7:          y++
8:       fimpara
9:    fimse
10: fimpara
```

EXERCÍCIO 3 idem. Ordene as seguintes funções por sua taxa de crescimento (usando $\ll$) (a) n; (b) $\sqrt{n}$; (c) $\log_2 n$; (d) $\log_2 \log_2 n$; (e) $\log_2^2 n$; (f) $n/\log_2 n$; (g) $\sqrt{n}\log_2^2 n$; (h) $(1/3)^n$; (i) $(3/2)^n$; (j) 17.

EXERCÍCIO 4 idem. Assuma que o parâmetro n é uma potência positiva de 2 (isto é: 2, 4, 8, 16...) Descubra a fórmula que dá y em função de n.

```
1: y ← 0
2: x ← 2
3: enquanto (x < n) faça
4:    x ← 2 × x
5:    y++
6: fimenquanto
7: imprima y
```

EXERCÍCIO 5 Faça uma análise comparativa entre uma busca linear em um vetor com e sem sentinela. Se não souber o que é sentinela, pergunte!

EXERCÍCIO 6 Escreva um algoritmo que leia um vetor de 200 inteiros e ache os 2 maiores valores presentes no vetor. A complexidade máxima admitida é $O(n) = n$, onde n=200.

Acompanhe a discussão a seguir...

N é primo ? [Swa91, devidamente corrigido..., págs. 16 a 20]

1ª Estratégia: Se N $\geq$ 4 é divisível por um fator entre 2 e (n-1) então, N é composto

```
1: lógico função EPRIMO1 (inteiro N)
2: lógico PRIMO
3: inteiro FATOR
4: PRIMO ← .V.
5: se N > 3 então
6:    FATOR ← 2
7:    enquanto PRIMO ∧ (FATOR ≤ N - 1) faça
8:       se N mod FATOR = 0 então
9:          PRIMO ← .F.
10:      senão
11:         FATOR++
12:      fimse
13:   fimenquanto
14: fimse
15: devolva PRIMO
16: fim função
```

2ª Estratégia: Se N for par (e diferente de 2) é composto, senão se for divisível por fator impar entre 3 e N-1 é composto, senão é primo. Se N < 4 é primo.

```
1: lógico função EPRIMO2 (inteiro N)
2: lógico PRIMO
3: inteiro FATOR
4: PRIMO ← ¬ ((N ≠ 2) ∧ (N mod 2 = 0))
5: se PRIMO ∧ (N > 4) então
6:    FATOR ← 3
7:    enquanto PRIMO ∧ (FATOR ≤ N - 1) faça
8:       se N mod FATOR = 0 então
9:          PRIMO ← .F.
10:      senão
11:         FATOR ← FATOR + 2
12:      fimse
13:   fimenquanto
14: fimse
```

15: devolva PRIMO
16: fim função

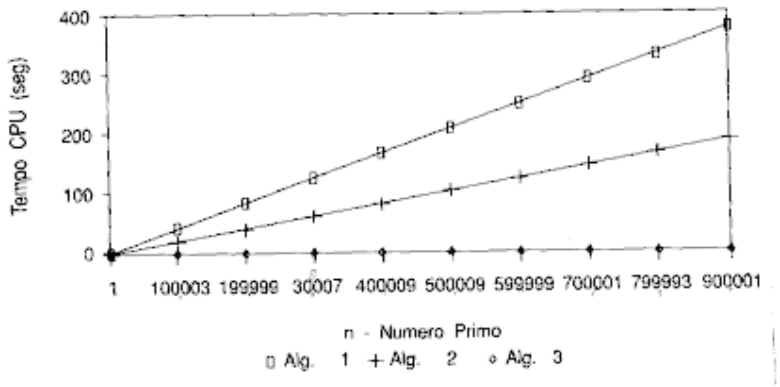
3ª Estratégia: Só precisamos testar o divisor de N até a raiz quadrada de N...

```
1: lógico função EPRIMO3 (inteiro N)
2: lógico PRIMO
3: inteiro FATOR
4: PRIMO ← ¬ ((N ≠ 2) ∧ (N mod 2 = 0))
5: se PRIMO ∧ (N > 4) então
6:    FATOR ← 3
7:    enquanto PRIMO ∧ (FATOR ≤ √N) faça
8:       se N mod FATOR = 0 então
9:          PRIMO ← .F.
10:      senão
11:         FATOR ← FATOR + 2
12:      fimse
13:   fimenquanto
14: fimse
15: devolva PRIMO
16: fim função
```

N	PRIMO1	PRIMO2	PRIMO3
10	10	5	3
100	100	50	10
1000	1000	500	32
10000	10000	5000	100

Testando estes 3 algoritmos para verificar se 10 números (100003, 199999, 300007, 400009, 500009, 599999, 700001, 799993 e 900001), tem-se o seguinte gráfico

Figura 2.2: gráfico de desempenho das 3 funções estudadas



 ©88-08, Pedro Kantek

Para finalizar, a soma de tempo dos 10 números é 186 seg. (alg. 1), 93 seg. (alg. 2) e 0,13 seg. (alg. 3)

EXERCÍCIO 7 Qual é o menor valor de n de tal maneira que o algoritmo de complexidade $O(100n^2)$ roda em menos tempo do que outro algoritmo equivalente de complexidade $O(2^n)$ no mesmo computador ?

EXERCÍCIO 8 Suponha que você tem vários algoritmos que devem processar um vetor. Esse vetor, rodando no computador em questão pode ser processado à razão de 10.000 posições (de vetor) por segundo, para qualquer um dos algoritmos. A pergunta que se faz, é qual é o maior tamanho do vetor que pode ser processado pelos algoritmos a seguir em cada uma das unidades de tempo da tabela. Note que cada algoritmo tem uma determinada complexidade, sempre estabelecida em função do tamanho n do vetor.

	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\log_2 n$							
$\sqrt{n}$							
n							
$n \times \log_2 n$							
n^2							
n^3							
2^n							
$n!$							

Respostas

©88-08, Pedro Kantek

	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\log_2 n$	10^8	3.6×10^{11}	1.2×10^{15}	7.4×10^{17}	6.7×10^{20}	9.9×10^{22}	9.9×10^{26}
$\sqrt{n}$	10^8	3.6×10^{11}	1.2×10^{15}	7.4×10^{17}	6.7×10^{20}	9.9×10^{22}	9.9×10^{26}
n	10.000	600.000	36×10^6	8.64×10^8	2.5×10^{10}	3.1×10^{11}	3.1×10^{13}
$n \times \log_2 n$	1003	39350	17×10^5	34×10^6	8×10^8	9×10^9	8×10^{11}
n^2	100	774	6.000	29.393	160.996	561.566	5.615.665
n^3	21	84	330	952	2.959	6.806	31.593
2^n	13	19	25	29	34	38	44
$n!$	7	9	10	12	13	14	15

2.4 Principais classes

EXERCÍCIO 9 Considere a seguinte lista de funções, todas dadas em função de n. Suponha que elas representam medidas de desempenho de algoritmos em função do tamanho da entrada n. Ordene-as em ordem de crescimento, em função do crescimento de n.

$6n \log_2 n$	2^{100}	$\log_2 \log_2 n$	$\log_2^2 n$
$2^{\log_2 n}$	2^{2^n}	$\lceil \sqrt{n} \rceil$	$n^{0.01}$
$\frac{1}{n}$	$4 \times n^{3/2}$	$3 \times n^{0.5}$	$5 \times n$
$\lfloor 2n \log_2^2 n \rfloor$	2^n	$nlog_4 n$	4^n
n^3	$n^2 \log_2 n$	$4^{\log_2 n}$	$\sqrt{\log_2 n}$

Dica: quando em dúvida ao analisar as funções $f(n)$ e $g(n)$, considere o desempenho de $\log f(n)$ e de $\log g(n)$ ou então de $2^{f(n)}$ e de $2^{g(n)}$. (Extraído de [Goo02]).

EXERCÍCIO 10 Estabeleça uma notação grande-O para os seguintes trechos de código:

```
1: X ← 0
2: para (I = 0; I < N; I++) faça
3:    X++
4: fimpara
```

```
1: X ← 0
2: para (I = 0; I < N; I++) faça
3:    para (K = 0; K < N; K++) faça
4:       X++
5:    fimpara
6: fimpara
```

```
1: X ← 0
2: para (I = 0; I < N; I++) faça
3:    para (K = 0; K < (N × N); K++) faça
4:       X++
5:    fimpara
6: fimpara
```

```
1: X ← 0
2: para (I = 0; I < N; I++) faça
3:    para (K = 0; K < I; K++) faça
4:       X++
5:    fimpara
6: fimpara
```



```
1: X ← 0
2: para (I = 0; I < N; I++) faça
3:    para (K = 0; K < (I × I); K++) faça
4:       para (M = 0; M < K; M++) faça
5:          X++
6:       fimpara
7:    fimpara
8: fimpara
```

```
1: X ← 0
2: para (I = 0; I < N; I++) faça
3:    para (K = 0; K < (I × I); K++) faça
4:       se (I mod K = 0) então
5:          para (M = 0; M < K; M++) faça
6:             X++
7:          fimpara
8:       fimse
9:    fimpara
10: fimpara
```

extraído de [Wei97]

EXERCÍCIO 11 Dados dois algoritmos equivalentes (que resolvem o mesmo problema) denominados A e B e que tem respectivamente complexidades exatas (sem constantes indeterminadas) expressas por p e q, informe para qual tamanho da entrada N, A é mais eficiente do que B

$p(n) = 2n^3 + n^2.\sqrt(n)$ e $q(n) = 2n^2.\log n + 3n$

$p(n) = 2^n + 10.n^10$ e $q(n) = n! + 3^n$

EXERCÍCIO 12 Calcule as complexidades esperadas para as duas funções equivalentes abaixo. Ambas calculam x^n. Qual das duas tende a ser mas rápida para grandes números ?

```
inteiro função poten(inteiro x, n)
   inteiro prod = 1
   inteiro i
   para i de 1 até n
      produto = produto * x
   fim{para}
   retorne produto
fim{função}

inteiro função poten2(inteiro x, n)
   se n = 0
      retorne 1
   senão
      t = poten2(x, chão(n/2))
      se ((n mod 2) = 0)
         retorne t * t
      senão
         retorne x * t * t
      fim{se}
   fim{se}
fim{função}
```

Capítulo 3

Recursividade

3.1 Recursividade

Figura 3.1: Um exemplo de recursividade

3.2 Introdução

> Lei da Homegrown
> Esta feira se realiza todo ano na Holanda, em Amsterdam, na semana do Thanksgiving day (novembro). Uma das leis que rolam lá tem letras monotemáticas: "Fume dois baseados antes de fumar dois baseados e depois fume mais dois baseados".

Alguma coisa é recursiva quando é definida em termos dela própria. O grande apelo que o conceito da recursão traz é a possibilidade de dar uma definição finita para um conjunto que pode ser infinito. Eis um exemplo da aritmética, bem simples, aquele que dá a definição dos números naturais, em termos dos axiomas de Peano:

- O primeiro natural é o zero
- O sucessor de um número natural é um número natural

A recursividade é uma daquelas idéias em informática (como algumas da vida comum) que em geral dividem a população em dois grupos antagônicos: os que são francamente favoráveis, em geral impressionados com a elegância e aparência de simplicidade

dos algoritmos recursivos, e os que a consideram uma frivolidade a ser evitada a todo custo. No caso da recursividade, talvez haja que se acrescentar um terceiro grupo: o daqueles que não a conhecem, ou nunca chegaram a entendê-la, e que talvez por isso com maior facilidade se alinhem no segundo grupo. Uma possível explicação para esse desconforto, vem da sensação de que a recursividade leva o procedimento ao infinito e este conceito tem assustado o homem desde a antigüidade.

Um programador digno do nome terá que conhecê-la, entendê-la e estar pronto a usá-la quando for a melhor (ou as vezes única) alternativa. Feita essa ressalva, fica a critério posterior usá-la ou não. Em geral, salvo raras exceções uma solução recursiva admite um algoritmo equivalente não recursivo. E também em geral, tal solução não recursiva exige maior código, embora este seja mais facilmente entendível, com exceção daquelas (poucas) pessoas que conseguem ver a recursividade como algo claro e límpido.

Mas, afinal, o que é a recursividade ? É a descrição de algo em termos dele mesmo. Só que para que a definição não seja circular (e portanto infinita), a descrição recursiva deve ser dada em termos "menores"que a definição original. É esse "menor"que faz com que em algum momento, o ciclo se encerre a e definição esteja completa. Qual o sentido e o significado desse menor, varia de caso a caso, e é muito difícil dar uma definição geral.

Antes de prosseguir nessas definições, e ver a recursividade na programação, vamos examinar um exemplo coloquial de procedimento recursivo.

Imaginemos ter que achar o significado da palavra "palombeta"no dicionário Aurélio. Obviamente, a ninguém ocorre a tentação de ler seqüencialmente o dicionário até chegar à palavra procurada. (ainda que a leitura descompromissada do Aurélio sempre reserve boas e divertidas surpresas).

Definamos o processo de achar uma palavra num dicionário em termos algorítmicos mais ou menos livres.

```
1:  função procura-palavra (Dicionário, palavra-procurada)
2:  abrir o dicionário na metade
3:  palavra-pivot1 ← a primeira palavra da página impar aberta
4:  palavra-pivot2 ← a última palavra da página par aberta
5:  se (palavra-procurada < palavra-pivot1) então
6:     procura-palavra (primeira-metade do dicionário, palavra-procurada)
7:  fimse
8:  se (palavra-procurada > palavra-pivot2) então
9:     procura-palavra (segunda metade do dicionário, palavra-procurada)
10: fimse
11: se (palavra-procurada > palavra-pivot1 ∧ palavra-procurada < palavra-pivot2) então
12:    ler seqüencialmente as duas páginas
13:    se (palavra-procurada for encontrada) então
14:       imprimir a sua definição
15:       fim do algoritmo
16:    fimse
17:    se (palavra-pivot2 for encontrada) então
18:       imprimir ("palavra procurada não existe")
19:       fim do algoritmo
20:    fimse
21:    fim da leitura seqüencial
22: fimse
```

Vejamos mais de perto esse algoritmo. Ele é recursivo no sentido de que ele chama a ele mesmo. Veja-se que o nome do algoritmo aparece também 2 vezes dentro dele mesmo. Nesse caso, o processo é finito, já que a cada chamada o universo de pesquisa é

menor. Na primeira vez, ele é todo o dicionário, na segunda vez, apenas a metade, na terceira vez, uma quarta parte e assim por diante. Finalmente, há uma condição que estabelece o fim da pesquisa (quando as páginas que contém a palavra procurada forem abertas).

No exemplo do dicionário vimos as 3 características que um problema recursivo tem que ter:

- Um processo que chame a si mesmo
- A garantia de que a cada chamada, o universo de trabalho do processo será "menor"
- Uma condição, que obrigatoriamente ocorrerá, que indique quando terminar.

Outro exemplo, usando uma analogia é a do processo pelo qual um historiador busca informações sobre a guerra do Paraguai, por exemplo. O processo poderia ser:

- obtenha cópia dos documentos que descrevem o período
- estude as informações referentes à guerra
- se qualquer documento menciona outros documentos, obtenha-os e estude-os.

Como se pode ver, o terceiro passo da "receita"nos conduz diretamente a outro ciclo recursivo, onde o processo vai ser novamente aplicado.

3.2.1 As Babushkas

O leitor já deve ter visto aquelas bonecas soviéticas chamadas babushkas. Trata-se de uma coleção de muitas bonecas, rigorosamente iguais, e que cabem uma dentro da outra. Ou seja, supondo que a nossa coleção tenha 10 bonecas, dentro da maior, estão todas as outras 9. Quando se abre a boneca grande, digamos a número 10, encontramos a boneca 9. Quando se abre a 9, temos a 8 e assim por diante, até a última, que em geral é bem pequenina. Nós vamos simular a recursão, neste início onde o assunto é apresentado a um computador que tem uma família de babushkas dentro dele. É nelas que vai se dar o fenômeno da recursão, já veremos como.

Como vimos, um programa recursivo é aquele que chama a si mesmo. Portanto precisamos nos acostumar com a idéia de múltiplas instâncias do mesmo programa, cada uma delas como entidade independente, com seus próprios dados de entrada e saída, convivendo juntas dentro da memória da máquina. É nessa hora que as babushkas vão nos ajudar.

Vamos elaborar um exemplo, passo a passo, para poder ver o funcionamento. Suponhamos a função matemática fatorial. Ela pode ser definida de maneira simples através de um algoritmo trivial (não recursivo), como o seguinte

1: inteiro **função** fatorialI (inteiro N)
2: inteiro I
3: $I \leftarrow 1$
4: **enquanto** $(N > 1)$ **faça**
5: $\quad I \leftarrow I \times N$
6: $\quad N - -$
7: **fimenquanto**
8: devolva I

Entretanto, a definição matemática de fatorial, ela de per si só, já sugere uma abordagem recursiva. Matematicamente, o fatorial de N pode ser definido como

```
Fatorial (N)
   é igual a 1 se N = 0
   é igual a N * Fatorial (N-1) se N ? 0
```


Note-se que esta definição inclui as 3 características acima apontadas como características da recursão:

1. A definição se refere a ela mesmo
2. A cada nova definição, o universo de atuação é menor. No nosso caso, de N passa-se a $N - 1$.
3. Existe uma condição de fim, no caso $N = 0$.

Ou seja, eis um exemplo perfeito para implementar o conceito de recursão. Seja agora a função

1: inteiro **função** FatorialR (inteiro N)
2: **se** $(N = 0)$ **então**
3: $\quad$ devolva 1
4: **senão**
5: $\quad$ devolva ($N \times$FatorialR $(N - 1)$)
6: **fimse**

Salta aos olhos que a definição recursiva é mais elegante, uma vez que:

1. Aplica rigorosamente a definição matemática
2. Não possui iteração explícita
3. Não possui variáveis locais à função

Vamos simular o funcionamento de FatorialR para um determinado valor e vamos usar as babushkas. Estas, sempre obedecerão aos seguintes princípios:

1. Vai se iniciar o processo sempre pela maior boneca. É ela também que receberá os dados iniciais e entregará o resultado final, quando este for calculado.
2. Cada vez que uma nova chamada recursiva for feita, os dados serão entregues à próxima boneca, que será colocada dentro da anterior, recebendo como dados de entrada, aqueles que lhe forem passados pela boneca anterior.
3. Quando o processo acabar, a última boneca que foi colocada, gerará um resultado, sairá de dentro do conjunto e entregará o resultado para o boneca que ficar após sua saída.

Ou seja, agora, podemos imaginar cada boneca como sendo uma instância do programa recursivo. Todas elas são iguais, e no entanto tem existência diferentes. Mais ainda, como e apenas uma cabe dentro da outra, podemos simular facilmente o protocolo de passagem e recepção de dados.

Dito isto, vamos ao exemplo: Suponhamos querer calcular FatorialR de 4. Como é a primeira chamada, vamos entregar esse pedido à maior babushka do conjunto. Ela vai receber o valor 4 e vai analisar o código da função (Babushkas sabem ler algoritmos !). Primeiro, ela vai associar N a 4, que é o valor que lhe foi passado. Indo em frente e olhando a linha 3, verá que $4 \neq 0$ e portanto executará a linha 6. Ao interpretar a linha 6, verá que deve chamar uma nova babushka, passando-lhe o valor de entrada que agora é $4 - 1 = 3$.

A história se repete, a segunda babushka recebe $N = 3$, verifica que $3 \neq 0$, e chama uma nova boneca com $N = 2$.

De novo, a terceira babushka recebe $N = 2$, verifica na linha 4 que $2 \neq 0$, e vai para a linha 6, chamando nova boneca com $N = 1$.

Mais uma vez, a quarta boneca recebe $N = 1$, verifica na linha 4 que $1 \neq 0$, e vai para a linha 6, chamando mais uma colega e passando-lhe o parâmetro $1 - 1 = 0$.

Chega a vez da quinta boneca que recebe o valor $N = 0$.

Agora, ao olhar o código ela verifica que o teste da linha 4 foi satisfeito, e portanto ela deve sair de dentro do conjunto de bonecas e retornar o valor 1, para a quarta boneca

que passará a ficar na vez de receber o processamento. A quarta boneca ao receber o valor 1 (da quinta boneca), multiplica-o pelo valor de N que ela (a quarta boneca) tem guardado, e que é 1 também, dando como resultado 1, e decide que seu trabalho terminou. Desocupa o espaço e entrega seu resultado para a terceira boneca.

Esta, ao receber o valor 1 entregue pela quarta boneca, multiplica-o por 2 (o seu valor de N), acha o resultado final, que é 2, desocupa o lugar e entrega o 2 para a segunda boneca.

A segunda, recebe 2 e multiplica-o pelo N que tem guardado (que é 3), obtendo como resultado final 6. Desocupa o lugar e entrega o valor 6 para a primeira boneca. Que o recebe e o multiplica pelo seu N (que é 4), obtendo o resultado final 24. Como ela é a última boneca, cabe apenas entregar o resultado final a quem a chamou, ou seja ao programa que disparou a primeira chamada recursiva da função.

Vamos a outro exemplo trivial, apenas para penetrar mais nos meandros da recursividade. Seja uma função que recebe um alfanumérico, e o tamanho real ocupado pelo seu conteúdo e imprime-o de trás para diante. Como sempre, antes fazemos a função iterativa (que neste caso é melhor, mais elegante do que a recursiva) já que este ainda é um exemplo didático

```
1: função TRASDIANTEI (TEXTO,TAMANHO)
2: alfanumérico[40] TEXTO
3: inteiro TAMANHO
4: inteiro J
5: para (J = TAMANHO − 1; J = 0; J − −) faça
6:    imprima TEXTO[J]
7: fimpara
```

Agora, a mesma função sendo chamada recursivamente. Note, novamente, que as 3 condições estão presentes, a saber: a função chama a si própria; a cada chamada seu universo é menor e existe uma condição bem clara de fim, qual seja TAMANHO ser igual a zero.

```
1: função TRASDIANTER (TEXTO,TAMANHO)
2: alfanumérico[40] TEXTO
3: inteiro TAMANHO
4: se (TAMANHO ≠ 0) então
5:    imprima TEXTO[TAMANHO-1]
6:    TRASDIANTER (TEXTO, TAMANHO-1)
7: fimse
```

Seja esta função sendo chamada com os parâmetros (”RUIM”, 4). Vamos usar o modelo das babushkas de novo. A maior delas (a primeira), recebe TEXTO = ”RUIM”, e TAMANHO = 4. Como 4 é diferente de zero, ela imprime a letra ”M”e chama a segunda boneca e lhe entrega TEXTO = ”RUIM”, e TAMANHO = 3. De novo, é impresso ”I”e nova babushka (a terceira) é chamada com TEXTO = ”RUIM”, e TAMANHO = 2. De novo, após a impressão de ”U”, nova recursão (quarta boneca) com TEXTO = ”RUIM”, e TAMANHO = 1. Aqui é impresso ”R”e chamada nova babushka (a quinta) com TEXTO = ”RUIM”, e TAMANHO = 0. Esta última, ao receber 0, não faz nada e encerra seu processamento, e devolve o controle para a quarta. Esta recebe o controle, não faz nada e o devolve para a terceira. Idem para a segunda e para a primeira. Esta última ao receber o controle encerra todo o processamento. Se verificarmos o que foi impresso, teremos ”MIUR”, que era o que foi pedido no início.

3.2.2 Um contra exemplo

Vejamos agora um problema, onde claramente e definitivamente a solução iterativa é melhor do que a recursiva, pelo menos no que diz respeito a uso de recursos (tempo e

 ©88-08, Pedro Kantek

memória) de máquina. Seja o seguinte enunciado, adaptado de [Hel86]. Imagine que você comprou um casal de ratinhos da índia recém nascidos. Tais animais se proliferam com grande facilidade. Para simplificar nosso problema, façamos as seguintes suposições: Os ratinhos alcançam sua maturidade sexual após 2 meses de vida, e tem um acasalamento por mês Os ratinhos nunca morrerão Cada casal de ratinhos sempre terá 2 filhos sendo um macho e outro fêmea A questão que se coloca é: ”Quantos ratinhos haverá após 7 meses ?”

Resolvendo a questão passo a passo:

Mês	Quantidade	Explicação
1	2	O casal original
2	2	O casal original que ainda não atingiu a maturidade
3	4	O original mais seu primeiro casal de filhos
4	6	O original, seus dois filhos do mês 3 e mais dois filhos do mês 4
5	10	O original, seus filhos dos meses 3, 4 e 5, e mais os filhos do casal nascido no mês 3
6	16	O original, seus filhos dos meses 3,4,5 e 6, os filhos dos meses 5 e 6 do casal nascido no mês 3, e o primeiro casal de filhos do casal nascido no mês 4
7	26	O original, seus filhos dos meses 3,4,5,6 e 7, os filhos dos meses 5,6 e 7 do casal nascido no mês 3, os filhos 6 e 7 do casal nascido no mês 4, e os dois primeiros casais de filhos dos casais nascidos no mês 5

Obviamente, não tem sentido progredir nessa abordagem. Se nos for pedido o número de ratinhos após 24 meses, não há meios de prosseguir nas contas sem que se entre em desespero.

Vem em nosso auxílio, uma lei de formação que fica bastante clara na análise do quadro acima. Ressalvados os meses 1 e 2, onde o número de ratos é constante, daí em diante pode-se observar que:

ratos (mês n) = ratos (mês n-1) + ratos (mês n-2)

Na verdade, se a conta se fizer não sobre ratos, mas sobre casais de ratos, tem-se a seqüência 1,1,2,3,5,8,13,21,34,65... que é a conhecida série de Fibonacci e que explica inúmeros fenômenos do dia a dia.

Podemos obter uma descrição recursiva para os ratos, através da definição: ratos(n) = 2 se n ? 2 e ratos(n) = ratos(n-1) + ratos(n-2) se n > 2

Como sempre, vamos primeiro fazer a solução iterativa

```
1: inteiro função RATOSI (inteiro N)
2: inteiro RAT[100]
3: inteiro K
4: RAT[0] ← RAT[1] ← 2
5: para (K = 2; /; K ≤ N; /; K + +) faça
6:    RAT[K] ← RAT[K − 1] + RAT[K − 2]
7: fimpara
8: devolva RAT[N]
```

Agora é hora de desenvolver a solução recursiva

```
1: inteiro função RATOSR (inteiro N)
2: se (N ≤ 2) então
3:    devolva 2
4: senão
5:    devolva RATOSR(N − 1) + RATOSR(N − 2)
6: fimse
```

©88-08, Pedro Kantek

Qual o grande problema da solução recursiva? Trata-se do fato de que inúmeros cálculos vão ser repetidos exaustivamente, sem nenhuma necessidade, já que em tese, uma vez obtido, digamos RATOS(5), não teria sentido ele ser recalculado mais vezes. Mas não é o que acontece aqui, senão vejamos:

Imaginemos a chamada RATOSR(6). Ela se desdobra em uma série de chamadas como podemos ver no desenho a seguir

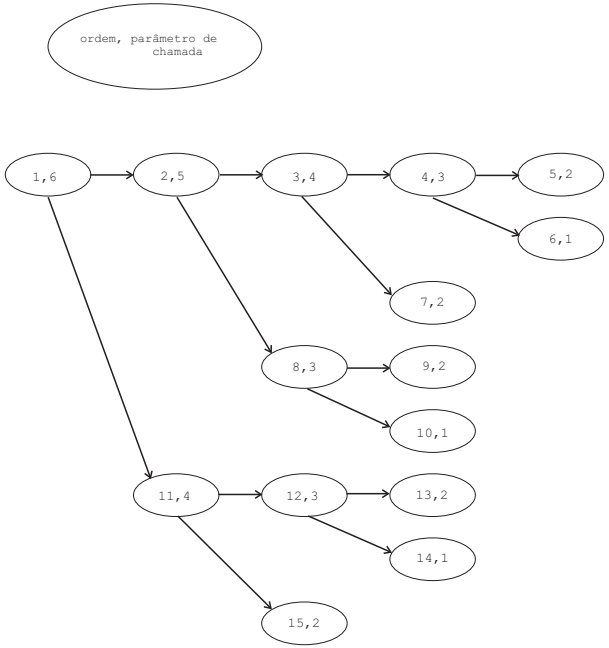


Figura 3.2: desmembramento das chamadas recursivas a RATOS. Dentro de cada elipse, o primeiro número indica a ordem de chamada e o segundo o parâmetro passado para a função

Esta figura deve ser lida olhando-se os números dentro de cada elipse: o primeiro número ordena as diversas execuções, da primeira até a décima quinta. O segundo indica qual o argumento da função RATOS-R quando ela foi chamada. Verifique-se que as elipses que contem 1 ou 2 como parâmetro não geram mais chamadas recursivas, pois esses valores alcançam a condição terminal de recursão. A crítica que se pode fazer a esta função é que ela calcula as mesmas coisas em momentos diferentes. Senão veja-se o cálculo de RATOSR(3), por exemplo. Ele é feito nas etapas 4, 8 e 12. Parece razoável que esse cálculo deveria ser feito uma única vez, já que o resultado nas três vezes, obviamente será o mesmo. Este problema, como foi programado tende a gerar soluções ineficientes e é um péssimo exemplo do uso da recursão.

Qual a complexidade de RATOSR ?

Vamos considerar a unidade de medida da complexidade como uma chamada à função. Olhando o mapa acima, vamos montar uma tabela de número de chamadas em função do parâmetro passado a RATOSR.

Parâmetro	Chamadas
1	1
2	1
3	3
4	5
5	9
6	15

Agora, tentemos achar uma lei de formação para esta tabela. Note-se que cada número a partir da terceira linha é a soma das duas linhas anteriores acrescida de uma unidade. Seguindo-se essa lei, pode-se prosseguir na tabela e temos

Parâmetro	Chamadas
7	25
8	41
9	67
10	109
11	177
12	287

De posse desta tabela, pode-se compará-la com as complexidades mais comuns, usando-se na tabela anexa

N	K	$log_2 N$	N	$N \times log_2 N$	N^2	N^3	2^N	3^N	$fat(N)$
2	1	1.0	2	2.0	4	8	4	9	2
3	1	1.6	3	4.8	9	27	8	27	6
4	1	2.0	4	8.0	16	64	16	81	24
5	1	2.3	5	11.6	25	125	32	243	120
6	1	2.6	6	15.5	36	216	64	729	720
7	1	2.8	7	19.7	49	343	128	2187	5040
8	1	3.0	8	24.0	64	512	256	6561	40320
9	1	3.2	9	28.5	81	729	512	19683	362880
10	1	3.3	10	33.2	100	1000	1024	59049	3628800
11	1	3.5	11	38.1	121	1331	2048	177147	39916800
12	1	3.6	12	43.0	144	1728	4096	531441	479001600

Olhando para RATOSR(12) que é 287, percebemos que este valor é maior do que N^2 (144), e portanto N^2 pode ser descartado. O próximo candidato é N^3, mas se adiantarmos os cálculos (por exemplo até 40), veremos que a seqüência de ratos ultrapassa também N^3. Mais ainda, se olharmos como a série é montada, podemos perceber a maneira exponencial como ela cresce. Responderá corretamente, quem disser que a complexidade da função é K^n, a menos das constantes. Na verdade no livro Matemática Concreta, [Gra95, pág. 218], os autores dão a seguinte fórmula fechada para o k-ésimo termo da seqüência de Fibonacci

$$F_n = \frac{1}{\sqrt{5}}\left(1.61803^n - (-0.61803)^n\right)$$

Note-se que esta fórmula é rápida, mas usa o número aúreo (que não é exato) no

cálculo. Assim, o resultado obrido não será exato e sim aproximado.

A nossa série de ratos é o dobro do número de Fibonacci, mais um a cada geração, a fórmula fechada para os ratos seria um pouco diferente, mas muito parecida, e de qualquer forma, exponencial.

A seqüência de Fibonacci é comum na matemática e ocorre em muitos fenômenos da natureza.

Acompanhe a descrição recursiva e não recursiva do elemento genérico da seqüência:

Definição recursiva $f(0) = 1$
$f(1) = 1$
$f(x+2) = f(x+1) + f(x)$

Definição explícita Para definir o i-ésimo termo da seqüência, pode-se usar a fórmula

$$x_n = 1\frac{1}{\sqrt{5}}\left[\left(1+\frac{\sqrt{5}}{2}\right)^n - \left(1-\frac{\sqrt{5}}{2}\right)^n\right]$$

Uma aplicação prática

Seja o cálculo de um salário líquido segundo a formulação a seguir: Salário Líquido = Salário Bruto - Descontos Descontos = Previdência Social + IR + Contribuição Sindical + Mensalidade da fundação + Plano de saúde

Onde Previdência social é igual a um percentual que varia em função de um suposto SALARIOREFERENCIA, de acordo com a tabela

Número de SALARIOREFERENCIA	Desconto em %
até 5	8,5 %
de 5,0001 até 11	9,5 %
de 11,0001 até 20	11,0 %
acima de 20	12,5 %

O Imposto de Renda (IR) varia também em função de uma tabela parecida, mas onde há uma parcela a subtrair do imposto devido. Por simplicidade, vamos usar o mesmo valor de SALARIOREFERENCIA.

Número de SR	Imposto devido em %	menos a parcela de
até 5	10 %	0
de 5,001 até 12	22 %	0.6 . SR
de 12,001 até 25	30 %	1.5 . SR
acima de 25	40 %	4 . SR

A contribuição sindical é de 1% do salário bruto. A mensalidade da fundação é de 5% do SR para quem ganha até 5 SR (inclusive) e de 12% do SR para quem ganha mais do que 5 SRs O plano de saúde custa 2% do salário bruto, independente de faixa. Todos os descontos incidem sobre o valor bruto inicial.

Claramente, dado um salário bruto e o valor do SR, é razoavelmente fácil calcular o salário líquido. Mas, suponhamos que o nosso problema é o inverso. Dado o líquido, obter o bruto. A função inversa não existe, devido ao uso da tabela do IR como ela se apresenta. É melhor usar outra abordagem, a da tentativa e erro, com tentativas cada vez mais próximas do alvo. Nosso problema é: dado um valor para o salário líquido, (e o SR), calcular qual deveria ser o salário bruto que correspondesse aquele líquido, a menos de um erro menor do que 1 centavo.

 ©88-08, Pedro Kantek

Para resolver esse problema uma possibilidade é criar uma função recursiva que seja chamada pela primeira vez com 2 valores, sendo ambos iguais ao líquido real. O segundo representará uma estimativa (um chute) para o valor do bruto. Quanto mais próximo ele estiver do real, menor o número de recursões. Mas há uma boa razão para começar chutando zero de descontos. [1] A condição de fim está claramente estabelecida. A diferença entre o liquido calculado (a partir do valor estimado) e o líquido real que foi fornecido à função tem que ser menor do que 1 centavo. Quando essa diferença for maior que 1 centavo, a função calculará a diferença entre o líquido real e o líquido calculado. Se a diferença for positiva, isso significa que a estimativa errou por baixo. Se for negativa, errou-se para mais. Como estabelecemos que ambos os valores inicialmente serão iguais, a estimativa sempre errará por baixo, e a diferença será positiva. A nova chamada da mesma função deverá substituir o segundo valor (a estimativa) pela estimativa anterior somada com a diferença encontrada. A função converge para o valor correto entre 10 e 40 recursões na média.

Ao final, a função deve imprimir o salário bruto real, e a quantidade de recursões que foram necessárias.

Para testar uma eventual implementação deste caso, utilize-se a tabela de respostas corretas:

Valor líquido real	Como chamar a função	SR	Valor bruto correto	Núm. de recursões
15,232.00	ACHABRU 15232 15232 2000	2,000	21,789.31	16
64,370.25		5,000	105,450.00	24
5,720.00		1,000	8,000.00	15
3,301.95		1,000	4,270.00	10
31,327.40		1,500	57,320.00	29
35,728.60		1,570	66,600.00	29
24,449.60		1,920	40,000.00	24

Eis um exemplo da aplicação das funções achabru e achaliq. Começando com `1200 achabru 1200`

1200.00	1200.00
1200.00	1426.00
1200.00	1449.73
1200.00	1452.22
1200.00	1452.48
1200.00	1452.51

Continuando com `800 achabru 800`

800.00	800.00
800.00	890.00
800.00	899.00
800.00	899.90
800.00	899.99

[1] Há uma boa razão para agir assim. Por mistérios das leis trabalhistas, pode ocorrer que dois salários brutos diferentes tenham o mesmo líquido. Começando a estimativa por baixo, o algoritmo sempre encontrará o mais baixo bruto que determina o líquido fornecido. Por exemplo, na primeira linha da tabela acima, o valor correto do bruto será 21789,31. Entretanto, um bruto de 22300,00 dará o mesmo líquido, já que os descontos, de alguma maneira são proporcionais ao ganho bruto. Se a função for chamada com estimativa muito grande, ela encontrará o maior valor

©88-08, Pedro Kantek

Mais um exemplo, agora bem simples Seja obter a potência n-ésima de um número dado K, sendo n inteiro e positivo. A função iterativa que executa essa tarefa é simples:

```
1: real função POTENCIAI (real K, inteiro N)
2: inteiro CONTADOR
3: real RESPOSTA
4: RESPOSTA ← K
5: para (CONTADOR = 1; CONTADOR ≤ N; CONTADOR + +) faça
6:    RESPOSTA ← RESPOSTA × K
7: fimpara
8: devolva RESPOSTA
```

Vejamos agora a solução recursiva:

```
1: real função POTENCIAR (real K, inteiro N)
2: se (N < 2) então
3:    devolva K
4: senão
5:    devolva K × POTENCIAR(K, N − 1)
6: fimse
```

EXERCÍCIO 13 Escreva um algoritmo recursivo e outro iterativo para avaliar a multiplicado por b, usando apenas a operação de adição. A e b são inteiros positivos.

EXERCÍCIO 14 A função de Ackerman é definida sobre os não negativos como segue:
$A(m, n) = n + 1$, se m é igual a 0
$A(m, n) = A(m − 1, 1)$, se m é diferente de 0 e n é igual a 0
$A(m, n) = A(m − 1, A(m, n − 1))$, se m e n são diferentes de zero

Calcule o valor de $A(2, 3)$, usando a definição recursiva acima.

3.3 Busca exaustiva e backtracking

Existe um capítulo dentro da solução de problemas chamado busca exaustiva. Trata-se de testar todas as possibilidades de resposta existentes para um dado problema até encontrar a solução procurada, ou concluir pela sua inexistência quando o espaço de pesquisa se esgotar.

Este tipo de abordagem de solução geralmente leva a soluções de complexidade exponencial, pelo que, ela costuma só ser empregada em instâncias pequenas de problemas. Mas mesmo essas, tem coisas interessantes a nos mostrar (Para ver uma abordagem diferente, adaptada a grandes espaços, veja o algoritmo A*, na aula de IA sobre este tópico). Mesmo porque aqui não vai se estudar a solução geral para essa classe de problemas, mas apenas um uso mais que interessante para a recursividade.

Novamente olhemos as 3 características de um problema recursivo (definido em termos de si próprio, universos cada vez menores e uma condição de fim). Todas elas estarão aqui, mas há um fenômeno novo a estudar: O encontro de um beco sem saída. Dito de outra maneira, quando uma das recursões chegar a uma situação que não permitirá chegar a uma boa solução, deveremos voltar à pilha de recursão até encontrar uma nova instância que permita uma nova linha de ação.

Imagine uma analogia com uma árvore (de verdade, dessas que tem uma frondosa copa toda verde, uma macieira, por exemplo). Suponhamos que alguém mande contar todas as folhas dos galhos que não tem nenhuma folha podre. Ou seja: a existência de

 ©88-08, Pedro Kantek

uma folha podre em qualquer posição do galho, permite ignorar todas as folhas (boas e podres) daquele galho.

Começamos pelo tronco, e depois pela sua primeira ramificação e assim sucessivamente até encontrar a primeira folha. Se ela estiver boa, passamos à seguinte do mesmo (sub) galho para contá-la e assim sucessivamente até terminar o galho. Quando encontrada uma folha podre (que é a analogia com o beco sem saída acima), podemos parar a contagem naquele galho e retroceder até o início do galho, ignorando todas as folhas dele - mesmo as boas - e seguindo para o próximo.

Essa estratégia, em computação, recebe o nome de *backtracking*, que literalmente significa abandonar o que se faz e voltar para trás na busca da próxima possibilidade em aberto. Fazer isso sem recursividade é possível (sempre é), mas é tarefa terrível, pela quantidade de *overhead* que introduz nos algoritmos.

3.3.1 As rainhas

Vamos examinar 2 exemplos da técnica, e para ambos vamos nos basear em um tabuleiro e nas peças do xadrez. O xadrez, além de uma das mais belas criações humanas, representa um manancial fabuloso de idéias para trabalhar com algoritmos. Com regras simples e em pequeno número e com um universo pequeno (64 casas, 32 pretas e 32 brancas), ainda assim o xadrez permite grandes e divinas criações humanas.

Se o leitor já ouviu falar de experiências genéticas deve saber que o material preferido dos geneticistas para suas experiências é a *Drosophila melanogaster*, a popular mosca doméstica. Barata, abundante, reproduz-se rápido (11 dias de gestação), tem grandes e manuseáveis cromossomos e alimenta-se de qualquer coisa. Pois bem, o xadrez já foi chamado a *Drosophila* da computação. [Mic85, pág 52].

O primeiro problema é o de colocar 8 rainhas em um tabuleiro de xadrez, sem que uma ataque a outra. Relembrando, rainhas podem atacar qualquer peça que esteja a qualquer distância, desde que na mesma linha ou mesma coluna ou em qualquer uma das duas diagonais que se cruzam na casa ocupada pela rainha. Dito assim, o problema parece trivial, mas não é. Ele não admite solução analítica, e tem complexidade exponencial. Só é tratável pela reduzida dimensão do tabuleiro. Mais tarde, com uma pequena alteração no algoritmo, seremos capazes de listar todas as possibilidades de disposição de 8 rainhas.

O chamado método da força bruta, tentaria encontrar todas as possíveis colocações das rainhas e depois verificaria quais posições são boas e quais não são. Só que mesmo para algo reduzido como um tabuleiro de xadrez, verifica-se que é tarefa hercúlea. Vejamos: são 8 linhas e 8 rainhas, logo cada rainha ocupará uma linha. Como cada linha tem 8 colunas, existem 8 possibilidades em cada linha. Logo, o número total de possibilidades é de $8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 = 8^8 = 16.777.216$ possibilidades. Examinando 1.000 por minuto, o que já é bastante, se necessitariam 279 horas para olhar todas. Precisamos um algoritmo mais inteligente. Vamos olhar o caso das rainhas e depois generalizá-lo na medida do possível.

O algoritmo de backtracking é simples, em princípio. Supondo um tabuleiro como uma matriz de 8×8, as rainhas irão sendo colocadas uma a uma, sempre na primeira casa possível. Começando, vamos colocar a primeira rainha na primeira casa disponível que é a (1,1). Na seqüência, vamos tentar colocar a segunda rainha, na linha 2, na primeira coluna disponível. O resultado desta tentativa é:

coluna 1 Não pode ficar pois esta na mesma coluna da rainha anterior

coluna 2 Não pode pois está na mesma diagonal da anterior.

coluna 3 Pode ficar.

A situação agora pode ser assim visualizada Acompanhando a colocação da terceira rainha (figura **??**)

©88-08, Pedro Kantek

Figura 3.3: Tabuleiro com duas rainhas

Figura 3.4: Tabuleiro com 3 rainhas

coluna 1 Não pode ficar pois esta na mesma coluna da rainha 1.

coluna 2 Não pode pois está na diagonal da rainha 2.

coluna 3 Não pode ficar pois esta na coluna da rainha anterior.

coluna 4 Não pode pois está na diagonal da anterior

coluna 5 Pode, e o tabuleiro fica conforme a figura.

Agora vamos colocar a rainha 4. Acompanhe

coluna 1 Não pode ficar pois esta na mesma coluna da primeira rainha.

coluna 2 Pode

rainha 5: na coluna 4

Figura 3.5: Quarta rainha

Figura 3.6: Quinta rainha

Até aqui, correu tudo bem. O problema começa a surgir agora: onde colocar a rainha 6 ? Note, que ela obrigatoriamente terá que estar na linha 6 (uma vez que são 8 rainhas em 8 linhas), mas não existe lugar para ela na linha 6. Senão vejamos:

coluna 1 ela é atacada pela rainha 1

coluna 2 ela é atacada pela rainha 4

coluna 3 ela é atacada pela rainha 2

coluna 4 atacada pela rainha 5

coluna 5 atacada pela rainha 3

coluna 6 atacada pela rainha 1, via diagonal Principal

coluna 7 atacada pela rainha 2, via diagonal Principal

coluna 8 atacada pela rainha 3, via diagonal Principal

Conclusão algum passo anterior nos trouxe a este beco sem saída.

Só resta retornar e tomar a próxima posição disponível na rainha anterior. Retira-se a rainha 5 e leva-se-á para a próxima posição possível, que é a coluna 8. (Note que ela não pode ocupar a coluna 5, nem a 6 e nem a 7). O tabuleiro fica:

Figura 3.7: A quinta rainha colocada, depois de um *backtracking*

 ©88-08, Pedro Kantek

Novamente não é possível colocar a rainha 6. Logo devemos voltar e retirar a rainha 5 e a 4 (pois a 5, como visto, já esgotou todas as suas possibilidades). A rainha 4 vai para a coluna 7,... e assim por diante.

O fenômeno de ter que voltar para trás, e alterar uma posição anterior, recomeçando todo o trabalho a partir dela, é que recebe o nome de backtracking.

A vantagem da recursividade nesta abordagem é evidente. Cada rainha a colocar, significa uma nova chamada à função de colocação de rainhas. À medida em que elas vão sendo colocadas, as funções vão se empilhando na pilha de recursão, de tal maneira que retornar a uma situação anterior, significa apenas encerrar uma ou mais funções empilhadas (e que conduziram ao beco sem saída), retornando o processamento a partir do ponto desejado.

Antes de prosseguir, para trabalhar diretamente no algoritmo, precisa-se pensar na representação do tabuleiro e da colocação das rainhas nele. A utilização de uma matriz 8×8 de valores lógicos é tentadora, pela similaridade com o problema real. Ficariam facilitadas também as análises de ataque via linha e via coluna. Mas, a análise das diagonais demandaria trabalho excessivo. Há um truque que permitirá economizar os cálculos necessários para verificar ataques pelas diagonais, descrito em [Wir86] e que consiste na verificação de um fenômeno interessante no tabuleiro: Seja um pedaço 4 x 4 de um tabuleiro, com as casas identificadas na base da linha,coluna

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

Para simplicidade de notação, vamos chamar as diagonais paralelas à principal (que vai de 1,1 até 4,4) de diagonais P (de principal) . E as outras paralelas à que vai de 4,1 até 1,4 de diagonais S (de secundária).

Pois a característica interessante é que todas as diagonais S tem como peculiaridade o fato de que a soma da linha com a coluna de cada uma de suas casas é constante. Falando das diagonais S, temos a diagonal 2 (só a casa 1,1), a diagonal 3 (casas 2,1 e 1,2), diagonal 4 (casas 3,1; 2,2 e 1,3) e assim por diante. Note-se que neste pequeno tabuleiro de 4 x 4 existem 7 diagonais do tipo S, que podem ser numeradas de 2 a 8.

Já as diagonais P, tem como característica comum a diferença entre a linha e a coluna serem também constantes para todas as casas da diagonal. Existe a diagonal 3 (formada pela casa 4,1), a 2 (formada por 3,1 e 4,2) e assim por diante. Neste tabuleiro temos 7 diagonais numeradas de 3 até -3.

No tabuleiro de xadrez completo existirão 15 diagonais de cada tipo. As S serão chamadas de 0 até 14 e as do tipo P serão chamadas de -7 até 7.

Assim, em vez de se usar uma matriz 8×8, usar-se-ão 4 vetores, assim estabelecidos

LR Indica qual a coluna onde está a i-ésima rainha. Vetor numérico de 8 inteiro, subtraindo-se 1 do índice para passar de 1 a 8 para 0 a 7.

CR Indica se existe rainha na j-ésima linha. Vetor de 8 lógicos. Também subtraindo-se 1.

DS Indica se existe rainha na k-ésima coluna do tipo S. Vetor de 15 lógicos.

DP Indica se existe rainha na l-ésima coluna do tipo P. Vetor de 15 lógicos, necessitando-se adicionar 7 ao índice para passar de -7 até 7, para 0 até 14.

Nesta organização, posicionar uma rainha na coluna LIN, COL pode ser assim escrita:

$LR[LIN] \leftarrow COL$
$CR[COL] \leftarrow$ .FALSO.
$DS[LIN + COL] \leftarrow$ .FALSO.
$DP[LIN - COL + 7] \leftarrow$ .FALSO.

©88-08, Pedro Kantek

Quando se encontrar uma situação de "beco sem saída", será necessário retirar uma rainha já colocada, e a operação será a oposta daquela acima descrita, a saber:

$CR[COL] \leftarrow$ .VERDADEIRO.
$DS[LIN + COL] \leftarrow$ .VERDADEIRO.
$DP[LIN - COL + 7] \leftarrow$ .VERDADEIRO.

Descobrir se uma posição LIN, COL é boa para uma rainha, equivale a testar:

$CR[COL] \wedge DS[LIN + COL] \wedge DP[LIN + COL + 7]$

Com isto feito, temos o programa

```
1: algoritmo RAINHAS
2: inteiro LIN
3: lógico SUCESSO
4: lógico CR[8]
5: inteiro LR[8]
6: lógico DS[15], DP[15]
7: lógico função RAINHA(LINRAI)
8: inteiro LINRAI
9: inteiro COL
10: lógico RESP
11: COL ← −1
12: repita
13:    COL + +
14:    RESP ← .FALSO.
15:    se (CR[COL] ∧ DS[LINRAI + COL] ∧ DP[LINRAI − COL + 7]) então
16:       LR[LINRAI] ← COL
17:       CR[COL] ←.FALSO.
18:       DS[LINRAI + COL] ← .FALSO.
19:       DP[LINRAI − COL + 7] ← .FALSO.
20:       se (LIN < 7) então
21:          RESP ← RAINHA(LINRAI + 1)
22:          se (RESP =.FALSO.) então
23:             CR[COL] ← .VERDADEIRO.
24:             DS[LINRAI + COL] ← .VERDADEIRO.
25:             DP[LINRAI − COL + 7] ← .VERDADEIRO.
26:          fimse
27:       senão
28:          RESP ← .VERDADEIRO.
29:       fimse
30:    fimse
31: até ((RESP = .VERDADEIRO.) ∨ (COL ≥ 8))
32: devolva RESP
33: fim da função
34: para (LIN = 0; LIN ≤ 7; LIN + +) faça
35:    LR[LIN] ← .VERDADEIRO.
36: fimpara
37: para (LIN = 0; LIN ≤ 14; LIN + +) faça
38:    DS[LIN] ← DP[LIN] ← .VERDADEIRO.
39: fimpara
40: RESP ← RAINHA(1)
41: imprima LR
```


A generalização deste problema, (isto é, encontrar todas as soluções), passa por uma pequena modificação no programa, que paradoxalmente deixa-o mais simples. Uma vez encontrada uma solução boa, em vez de encerrar o algoritmo, imprimir-se-á a resposta e passar-se-á a considerá-la ruim, deixando o barco correr, até o esgotamento completo das soluções.

```
1: algoritmo TODASRAINHAS
2: inteiro LIN
3: lógico SUCESSO
4: lógico CR[8]
5: inteiro LR[8]
6: lógico DS[15], DP[15]
7: lógico função TODARAINHA(LINRAI)
8: inteiro LINRAI
9: inteiro COL
10: lógico RESP
11: COL ← −1
12: repita
13:    COL + +
14:    RESP ← .FALSO.
15:    se (CR[COL] ∧ DS[LINRAI + COL] ∧ DP[LINRAI − COL + 7]) então
16:       LR[LINRAI] ← COL
17:       CR[COL] ←.FALSO.
18:       DS[LINRAI + COL] ← .FALSO.
19:       DP[LINRAI − COL + 7] ← .FALSO.
20:       se (LIN < 7) então
21:          RESP ← TODARAINHA(LINRAI + 1)
22:       senão
23:          imprima LR
24:       fimse
25:       CR[COL] ← .VERDADEIRO.
26:       DS[LINRAI + COL] ← .VERDADEIRO.
27:       DP[LINRAI − COL + 7] ← .VERDADEIRO.
28:    fimse
29: até ((RESP = .VERDADEIRO.) ∨ (COL ≥ 8))
30: devolva RESP
31: fim da função
32: para (LIN = 0; LIN ≤ 7; LIN + +) faça
33:    LR[LIN] ← .VERDADEIRO.
34: fimpara
35: para (LIN = 0; LIN ≤ 14; LIN + +) faça
36:    DS[LIN] ← DP[LIN] ← .VERDADEIRO.
37: fimpara
38: SUCESSO ← TODARAINHA(1)
39: imprima LR
```

A solução ótima

Uma variante do problema anterior, ocorre quando não se desejam todas as soluções possíveis, mas apenas a ótima, assim considerada segundo um critério qualquer e que obviamente depende do problema em questão. Vamos exemplificar a técnica usando ainda o problema das rainhas, e imaginando que o critério de otimalidade poderia ser o menor número de chamadas recursivas à função que calcula a posição final. A mudança é sim-

ples. Dado o algoritmo anterior, ao se gerar uma solução completa, seria gerado também o nível de recursão atingido. Se este nível for menor do que um valor global do algoritmo (e inicializado com um valor bem alto), a solução será guardada e seu nível de recursão ocupará o valor desta variável global. Se o nível for igual ou maior, a solução será desprezada. Ao final, bastará imprimir a solução guardada.

```
1: algoritmo MELHORAINHA
2: inteiro LIN, RECU, LIMRECU, LRS
3: lógico SUCESSO
4: lógico CR[8]
5: inteiro LR[8]
6: lógico DS[15], DP[15]
7: lógico função MELHORAINHA(LINRAI)
8: inteiro LINRAI
9: inteiro COL
10: lógico RESP
11: RECU++
12: COL ← −1
13: repita
14:    COL++
15:    RESP ← .FALSO.
16:    se (CR[COL] ∧ DS[LINRAI + COL] ∧ DP[LINRAI − COL + 7]) então
17:       LR[LINRAI] ← COL
18:       CR[COL] ←.FALSO.
19:       DS[LINRAI + COL] ← .FALSO.
20:       DP[LINRAI − COL + 7] ← .FALSO.
21:       se (LIN < 7) então
22:          RESP ← MELHORAINHA(LINRAI + 1)
23:       senão
24:          se (RECU < LIMRECU) então
25:             LRS ← LR
26:             LIMRECU ← RECU
27:             RECU ← 0
28:          fimse
29:       fimse
30:       CR[COL] ← .VERDADEIRO.
31:       DS[LINRAI + COL] ← .VERDADEIRO.
32:       DP[LINRAI − COL + 7] ← .VERDADEIRO.
33:    fimse
34: até ((RESP = .VERDADEIRO.) ∨ (COL ≥ 8))
35: devolva RESP
36: fim da função
37: para (LIN = 0; LIN ≤ 7; LIN++) faça
38:    LR[LIN] ← .VERDADEIRO.
39: fimpara
40: para (LIN = 0; LIN ≤ 14; LIN++) faça
41:    DS[LIN] ← DP[LIN] ← .VERDADEIRO.
42: fimpara
43: LIMRECU ← ∞
44: RECU ← 0
45: SUCESSO ← MELHORAINHA(1)
46: imprima LRS, LIMRECU
```

 ©88-08, Pedro Kantek

3.4 Os cavalos

Outro problema que pode exemplificar de novo a técnica é o chamado percurso do cavalo. Dado um tabuleiro de xadrez de $n \times n$ casas (e aqui o tabuleiro real 8×8 é um caso particular), e dada uma posição inicial para um cavalo, busca-se saber qual a seqüência de saltos que permitirá ao cavalo passar por todas as casas do tabuleiro parando 1 e só 1 vez em cada uma delas. Note-se que este problema nem sempre tem solução, podendo ocorrer para um dado tabuleiro ou uma dada posição inicial, que não haja tal caminho.

Antes de prosseguir analisando o problema, relembremos os movimentos de um cavalo no tabuleiro de xadrez. Supondo uma matriz de 5×5, e imaginando que o cavalo ocupe a posição central (a 3,3), os saltos que ele pode dar são os seguintes (numerados de 1 a 8).

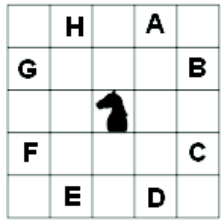


Figura 3.8: Oito movimentos possíveis ao cavalo que esteja na posição 3,3

Note-se que a seqüência de numeração é arbitrária, e serve apenas para orientar a busca de posições aceitáveis para o movimento da peça. Há outro detalhe importante. Os 8 movimentos só são possíveis quando há pelo menos 2 linhas e colunas acima e abaixo e à esquerda e à direita da posição atual. Se o cavalo se encontrar próximo dos limites do tabuleiro, alguns movimentos podem ser inaceitáveis, pois conduziriam o cavalo para fora do tabuleiro. Finalmente, o cavalo não pode ocupar uma casa já ocupada, o que no nosso exemplo significa que ele só pode passar em cada casa uma vez.

Antes de prosseguir na análise do algoritmo, veja-se uma resposta encontrada para um tabuleiro de 5×5, colocando-se o cavalo na posição inicial.

19	8	3	14	25
2	13	18	9	4
7	20	1	24	15
12	17	22	5	10
21	6	11	16	23

A seqüência de numeração indica qual o caminho percorrido pela peça. A configuração foi encontrada após 9908 iterações, aí incluídos os retrocessos necessários quando se chegou a uma posição que não levaria a preencher todo o tabuleiro.

No problema mais geral, O tabuleiro será representado por uma matriz numérica $n \times n$, onde n é a dimensão linear do tabuleiro, inicialmente zerada e a marcação do caminhamento será feita pela utilização da série de números inteiros 1,2,3... . Posto assim, pesquisar se uma casa está livre, isto é, ainda não foi usada será equivalente a perguntar se ela é igual a zero em tal matriz. Esta será global ao algoritmo, e para automatizar os movimentos do cavalo, mais dois vetores de 8 números cada, serão necessários. Olhando para a figura que descreve os 8 movimentos possíveis do cavalo, verifique-se que a obtenção da primeira casa implica em subtrair 1 da linha atual e 2 da coluna atual. Da segunda casa em subtrair 2 da linha atual e 1 da coluna atual. Da casa 3, subtrair 2 da

©88-08, Pedro Kantek

linha e somar 1 à coluna atuais. De formas que podem-se construir 2 vetores, chamados DELTALIN e DELTACOL que indicarão as quantidades a somar ou subtrair da posição atual para obter as 8 posições candidatas a receberem o cavalo.

Os valores para esses vetores serão

$DELTALIN \leftarrow -1\ -2\ -2\ -1\ 1\ 2\ 2\ 1$ e $DELTACOL \leftarrow -2\ -1\ 1\ 2\ 2\ 1\ -1\ -2$

Ou seja, a k-ésima posição de DELTALIN e DELTACOL indicam quanto somar à linha e coluna atuais para obter a linha e coluna da k-ésima posição possível para o cavalo.

A função recursiva que vai permitir a busca do caminho receberá o número do movimento a fazer, seguido da indicação de linha e coluna da última posição ocupada pelo cavalo. No exemplo acima, a função recebeu os parâmetros 2,3,3, significando que o próximo movimento a fazer deveria ser o de número 2, e que o resultado do movimento número 1 era o cavalo colocado na linha 3, coluna 3.

A função retornará uma valor lógico indicando o sucesso (.V.) ou insucesso (.F.) da tentativa daquele movimento.

```
1: Algoritmo CAVALO
2: inteiro MATRIZ[n][n]
3: inteiro DELTALIN[8]
4: inteiro DELTACOL[8]
5: DELTALIN ← −1 − 2 − 2 − 11221
6: DELTACOL ← −2 − 11221 − 1 − 2
7: lógico função ANDACAVALO (NUMMOV, LINCAV, COLCAV)
8: inteiro NUMMOV, LINCAV, COLCAV
9: inteiro K
10: lógico RESP
11: K ← 0
12: repita
13:    RESP ← .FALSO.
14:    LINPOS ← LINCAV + DELTALIN[K]
15:    COLPOS ← COLCAV + DELTACOL[K]
16:    se (LINPOS ≥ 0) ∧ (LINPOS ≤ n − 1) ∧ (COLPOS ≥ 0) ∧ (COLPOS ≤ n − 1)
       então
17:       se (MATRIZ[LINPOS][COLPOS] = 0) então
18:          MATRIZ[LINPOS][COLPOS] ← NUMMOV
19:          se (NUMMOV < n^2) então
20:             RESP ← ANDACAVALO(NUMMOV + 1, LINPOS, COLPOS)
21:             se (RESP = .FALSO. ) então
22:                MATRIZ[LINPOS][COLPOS] ← 0
23:             fimse
24:          senão
25:             RESP ←? .VERDADEIRO.
26:          fimse
27:       fimse
28:    fimse
29:    K + +
30: até ((K ≥ 8) ∨ (RESP ≠ .FALSO.))
31: devolva RESP
32: fim da função
33: MATRIZ ← 0
34: MATRIZ[linhainicial][colunainicial] ← 1
35: se (ANDACAVALO(2, linhainicial, colunainicial)) então
36:    imprima MATRIZ
```



```
37: senão
38:    imprima "Não existe caminho possível"
39: fimse
```

Seja bem vindo, Deep Blue

Algum tempo atrás tivemos a eletrizante (tanta emoção quanto as melhores novelas da televisão - e sem comerciais irritantes no meio) disputa entre Gary Kasparov e Deep Blue. De um lado, um genial enxadrista no vigor da idade e a credencial inconteste de melhor jogador de xadrez vivo. Do outro lado, um computador. Todos os jornais e revistas tem descrito o Deep Blue como um imenso e especializado cérebro eletrônico, como se dizia quando eu era criança e havia a ameaça de um computador ensandecido (e provavelmente comunista) declarar guerra à espécie humana.

Aqui a meu ver reside a falácia maior de toda a questão. Pois todos falam no hardware, descrevem como ele faz 200.000 comparações de jogadas por segundo, de quantos processadores especializados ele tem e por aí vai. Nem uma palavra sobre o software, esse sim, ouro puro, resultado do que de melhor pode a espécie humana produzir em termos de formalismo, concisão e rigor científicos. Aliás, de nada vale poder analisar 200.000 possibilidades por segundo. Há um episódio verídico que ilustra bem esse fato. Certa vez perguntaram a Ricardo Reti (um campeão mundial, grande jogador, exímio na arte de aprontar verdadeiras arapucas nas quais o infeliz oponente era atraído achando que fazia grande negócio...), quantos lances ele precisava analisar para jogar bem. Sua resposta é deliciosa: apenas um. O certo!

Há uma explicação para o fato: o hardware é produto comercial, trabalho coletivo, para o qual convergem programas, enfoques e estudos, é mais engenharia que arte, e seu fabricante tem o maior interesse, e nesse caso todo o direito, de falar à exaustão sobre ele. Já o software... bem o software é mais produção individual, está mais para arte do que para engenharia. É resultado de muitas horas de atividade solitária. Não tem o glamour do hardware, na verdade sequer tem existência física real. É pouco mais que uma abstração. Aliás, é do nosso colega Müller, também conhecido como "o"mestre, a antológica definição: Na dúvida entre o que é hardware e o que é software, jogue os dois para cima. O que cair na sua cabeça e causar dor, é hardware. A equipe de desenvolvimento do Deep Blue é composta por 6 pessoas, sendo sua estrela um aluno de doutorado que propôs, há vários anos atrás um algoritmo mais inteligente para jogar e vencer o xadrez. Dessas pessoas não se fala, mas são elas que literalmente carregam o deep blue no colo. Sem elas, tal máquina era capaz de perder um jogo de damas de quem aprendeu o jogo há duas semanas.

Confesso que fiquei dividido na disputa. Horas torcia pelo deep blue, ou mais especificamente falando, pelos programadores do deep blue, esperando que mais essa barreira fosse rompida e que tivéssemos a prova inconteste de que a inteligência artificial é uma área séria da ciência, com importantes realizações em seu ativo, e o mais importante com belas promessas. Horas torcia pelo Kasparov, último defensor da humanidade frente a invasão das máquinas. Nesses momentos via-me correndo em fuga, diante de um computador enlouquecido gritando te pego, desgraçado.

Alguns podem dizer que o xadrez é mero divertimento, coisa a toa, sem importância. Lembro-me a grande definição do Millor Fernandes, para quem o xadrez é atividade utilíssima, sendo indicada para todas as pessoas que querem ter mais habilidade de... jogar xadrez. Na verdade quem já se sentou diante de um oponente, com a responsabilidade de conduzir seu pequeno

exército em direção ao rei adversário, sabe que naquele pequeno retângulo de madeira há mais estratégia, arte, sacanagem, aprontação, virtude e generalidade que em muitos outros locais mais visados e glamurosos (a Onu ou a OTAN, só para ficar nos mais famosos). É uma pena que nossas escolas, não arrumem um espacinho na sobrecarregada agenda dos alunos para incentivar o xadrez. Pois este é uma das poucas coisas que ao mesmo tempo que auxilia o treinamento do raciocínio abstrato e o desenvolvimento de estratégias sobre um conjunto pequeno e estável de regras, é divertido. A boa nova do xadrez é que a lição de casa não precisa ser chata, aborrecida e cinzenta para produzir resultados. É possível desenvolver-se divertindo-se... pensando bem acho que é por isso que as escolas ignoram solenemente o xadrez...

Para os que desprezam o xadrez, vale a lembrança de que ele já foi chamado de drosóphila melanogaster da inteligência artificial. A drosóphila, é a mosca, esse ser abjeto que nos irrita, aquela da sopa do Raul Seixas. É difícil achar que tal ser tenha alguma utilidade, mas ele é uma dádiva divina para os geneticistas. Seus cromossomos são enormes, facilmente coráveis e estudáveis. Tem um ciclo reprodutivo de 11 dias, come qualquer coisa, é barata e facilmente encontrável. Portanto qualquer cientista que estude a genética há de agradecer pela existência da mosca. Quem se interessa, ainda que de leve pela inteligência artificial, tem que ser grato pela existência do xadrez.

E, para aqueles que acham ser um jogo bobo, vamos a alguns números: Uma partida média tem cerca de 40 movimentos de cada lado. 80 no total. A cada jogada, há cerca de 25 respostas possíveis. Assim se fossemos escrever numa tabela todos os jogos de xadrez possíveis de existirem, essa tabela teria um numero bem grandinho de elementos. Vejamos: no primeiro nível, são 25 possibilidades. Cada uma delas dá origem a outras 25, o que significa 625 possibilidades no segundo lance. No terceiro, são 15.625. No quarto 390.625, no quinto, 9.765.625. Lá pelo lance 60 (na verdade, no lance 30 de cada jogador) as combinações já são 7.523.310.000.000.000.000.000 .000.000.000.000. 000.000.000.000. 000.000.000.000.000.000.000.000. Só para comparação, estimam-se existirem "apenas"1080 elétrons no universo. Esse número de combinações é atingido lá pela altura da jogada 55. As demais ficam por conta do lucro.

Para encerrar este artigo, fica a reflexão. É falsa a afirmativa de que o desafio foi entre homem e máquina. Foi na verdade entre o Kasparov e os 6 programadores. E, lastimo dizer ao campeão mundial, que a turma da informática levou mais essa fatura... (P. Kantek, publicado em jun/98)

3.5 Casamento estável

Seja agora o problema de relacionar 2 conjuntos de elementos segundo um determinado critério. Este problema é conhecido na literatura como o casamento estável, uma vez que se usa da analogia com o casamento humano para apresentá-lo. Mas, obviamente, ele pode ser usado sempre que um problema de características semelhantes se apresentar. Por exemplo, um grupo de alunos buscando professor orientador e um grupo de professores interessados nos alunos mais competentes.

Como o algoritmo é complexo, antes de apresentá-lo, vamos ver um exemplo. Seja um grupo de 5 moças (Luiza, Joana, Maria, Josefa e Walquíria) e outro conjunto de rapazes (Simão, João, Alfredo, Ernesto e Sebastião). Na nossa analogia, os homens do grupo querem se casar com uma (e só uma) das mulheres e as mulheres com um (e um só) dos homens.

 ©88-08, Pedro Kantek

Foi pedido a cada um dos elementos de cada grupo que escreva os nomes das pessoas do grupo oposto com que gostariam de se casar, em ordem decrescente de interesse. Os resultados obtidos foram consolidados na lista de preferência dos homens:

homem	Lista de mulheres com que ele gostaria de casar
Simão	Joana, Josefa, Walquiria, Luiza e Maria
João	Maria, Joana, Luiza, Josefa e Walquíria
Alfredo	Walquíria, Joana, Luiza, Maria e Josefa
Ernesto	Maria, Joana, Josefa, Walquíria e Luiza
Sebastião	Luiza, Joana, Maria, Walquíria e Josefa

Identicamente, foi pedido às mulheres que preparem lista similar. O resultado foi a lista de preferência das mulheres:

mulher	Lista de homens com que ela gostaria de casar
Luiza	João, Simão, Sebastião, Alfredo e Ernesto
Joana	Sebastião, João, Ernesto, Alfredo e Simão
Maria	Simão, Ernesto, João, Alfredo e Sebastião
Josefa	Simão, Sebastião, Alfredo, Ernesto e João
Walquíria	João, Alfredo, Ernesto, Sebastião e Simão

Repare-se que se houver uma relação um para um, entre os dois grupos, o algoritmo é dispensável, basta chamar o padre. Mas não é isso que ocorre aqui. Note-se que, por exemplo João e Ernesto querem se casar com a Maria, enquanto Maria e Josefa querem se casar com Simão. É impossível contentar às 10 pessoas. Alguém terá que ceder. O algoritmo que veremos a seguir, verifica todos os casamentos possíveis apresentando os casamentos estáveis, bem como seu índice de perda para cada um dos casos. O índice de perda é calculado somando-se uma unidade a cada preferência descartada, incluindo a preferência aceita. Ou seja, se num dado casamento, 3 homens conseguiram sua primeira escolha, um deles só conseguiu a segunda e o último só a terceira, a perda dos homens será 8 (os três primeiros perdem 1, o quarto perde 2 e o quinto perde 3, logo 1+1+1+2+3=8). Nesse mesmo casamento, calcula-se também a perda das mulheres que geralmente é diferente da dos homens. Por hipótese, se aqui, apenas duas tivessem conseguido sua primeira opção, duas a segunda e a última apenas a terceira opção, a perda das mulheres seria de 1+1+2+2+3 =9.

O casamento, neste contexto é a formulação de k pares, com um elemento tirado de cada conjunto, onde k é o tamanho de cada um deles. Ele é dito instável quando para pelo menos um dos casais formados vale a regra: H_1 está casado com M_1, e M_2 está casada com H_2. Entretanto, na lista de preferências de H_1, M_2 aparece antes do que M_1. Identicamente, na lista de M_2, H_1 aparece antes do que H_2. Se não houver nenhum casal formado para o qual a regra acima seja válida, o casamento é dito estável.

Em resumo: um casamento é instável houver pelo menos um par onde o homem gostaria de ter casado com outra mulher e esta outra mulher – também casada – preferiria ter casado com o primeiro homem e não com o seu marido

Antes de prosseguir só um aviso: note que a analogia com o casamento humano foi um pouco forçada (as emoções humanas não são tão matemáticas assim), principalmente pelo fato de que as escolhas, uma vez feitas não se alteram em função das atribuições que vão sendo feitas pelo algoritmo. Feita a ressalva, vamos ao algoritmo.

Em primeiro lugar, a representação de dados. Numerados os elementos de cada conjunto de 1 a n, obtêm-se duas matrizes $n \times n$, a matriz da preferência dos homens e a da preferência das mulheres. Para o exemplo acima mostrado, tais matrizes seriam

Preferência dos homens (chamada no algoritmo de PREFHOM)

©88-08, Pedro Kantek

2 4 5 1 3
3 2 1 4 5
5 2 1 3 4
3 2 4 5 1
1 2 3 5 4

Preferência das mulheres (PREFMUL)

2 1 5 3 4
5 2 4 3 1
1 4 2 3 5
1 5 3 4 2
2 3 4 5 1

Será necessário ainda um vetor de homens (chamado HOMEM), de n posições, que conterá em cada uma delas o número da mulher com quem se casará no casamento que está sendo analisado. Enquanto as atribuições forem sendo consideradas estáveis, ele irá sendo preenchido. Embora não fosse necessário, será também usado um vetor chamado MULHER, que conterá o número do homem com quem ela se casará. Ele não é necessário, já que bastaria uma inspeção em HOMEM para saber com qual homem uma dada mulher se casaria. Mas, para simplificar, ambos vão ser criados.

O algoritmo será chamado para o primeiro homem da lista. Este será casado com todas as mulheres (na ordem de sua preferência), preservando-se o primeiro resultado que for estável. Feito isso, passa-se ao segundo homem (eis aqui a recursividade), e assim sucessivamente. Quando todos os homens estiverem casados, o resultado será impresso (com as correspondentes perdas), e se retorna na busca da próxima mulher que continuar tornando estável a união. Quando todas as possibilidades forem exauridas, o algoritmo acaba.

Para controlar a união, precisa-se um vetor de tamanho n formado por lógicos, (chamado SOLTEIRO), que conterá .V. na posição k quando o homem k estiver solteiro ainda e .F. quando ele já estiver casado.

Finalmente, há mais duas matrizes (chamadas CLASHOM e CLASMUL), que podem ser obtidas facilmente a partir das matrizes de entrada. O elemento CLASHOM[x][y] indica qual a classificação da mulher y na lista de preferências do homem x. Identicamente CLASMUL [x][y] indica qual a classificação do homem y na lista de preferências da mulher x.

Para as matrizes acima mostradas, os valores seriam

CLASHOM

4 1 5 2 3
3 2 1 4 5
3 2 4 5 1
5 2 1 3 4
1 2 3 5 4

A primeira linha de CLASHOM deve ser lida como: na preferência do Simão (linha 1), a Luiza (col 1) ocupa a 4ª posição. A joana (col 2), ocupa a 1ª posição. A Maria (col 3) ocupa a 5ª posição e assim por diante. Note- se que esta matriz é obtida facilmente de PREFHOM, logo ao início do processamento.

CLASMUL

2 1 4 5 3
5 2 4 3 1
1 3 4 2 5
1 5 3 4 2
5 1 2 3 4

A linha 3 (a título de exemplo) da matriz acima, deve ser lida como: Na preferência da Maria (linha 3), o Simão (col 1) ocupa a primeira posição. O João (col 2) ocupa

 ©88-08, Pedro Kantek

a terceira posição e assim sucessivamente. Esta matriz também é facilmente obtida a partir de PREFMUL.

O resultado obtido para este conjunto de dados é

sol.	x_1	x_2	x_3	x_4	x_5	p. masc	p. fem
1	4	2	5	3	1	7	10
2	4	1	5	3	2	9	7

A melhor solução para os homens (perda mas=7) é formada por Simão-Josefa (p=2); João-Joana (P=2); Alfredo-Walquiria (P=1), Ernesto-Maria (P=1), Sebastião-Luiza (P=1). Esta mesma solução apresenta as seguintes perdas para as mulheres: Josefa-Simão (P=1); Joana-João (P=2); Walquiria-Alfredo (P=2), Maria-Ernesto (P=2), Sebastião-Luiza (P=3).

A melhor solução para as mulheres (perda fem=7) é formada por Josefa-Simão (P=1); Luiza-João (P=1); Walquiria-Alfredo (P=2); Maria-Ernesto (P=2) e Joana-Sebastião (P=1). Essa mesma solução apresenta as seguintes perdas para os homens: Simão-Josefa (P=2); João-Luiza (P=3); Alfredo-Walquiria (P=1); Ernesto-Maria (P=1) e Sebastião-Joana (P=2).

Dito isso, segue o algoritmo

```
1: função IMPRIME ()
2: inteiro I, PERDAHOM, PERDAMUL
3: PERDAHOM ← PERDAMUL ← 0
4: para (I = 0; I ≤ n − 1; I++) faça
5:    imprima MULHER[I]+1
6:    PERDAHOM ← PERDAHOM + CLASHOM [I] [MULHER[I]] + 1
7:    PERDAMUL ← PERDAMUL + CLASMUL [MULHER[I]] [I] + 1
8: fimpara
9: imprima PERDAHOM, PERDAMUL
10: fim função
```

```
1: função TENTATIVA (HOM)
2: inteiro HOM
3: inteiro K, MUL
4: K ← 0
5: enquanto K < n faça
6:    MUL ← PREFHOM [HOM] [K]
7:    se SOLTEIRO [MUL] ∧ ESTAVEL (HOM, MUL, K) então
8:       MULHER [HOM] ← MUL
9:       HOMEM [MUL] ← HOM
10:      SOLTEIRO [MUL] ← .FALSO.
11:      se HOM < n-1 então
12:         TENTATIVA (HOM+1)
13:      senão
14:         IMPRIME()
15:      fimse
16:      SOLTEIRO[MUL] ← .V.
17:   fimse
18:   K++
19: fimenquanto
```

```
1: lógico função ESTAVEL (HOM, MUL, LIMITE)
2: inteiro HOM, MUL, LIMITE
3: inteiro CANDHOM, CANDMUL, K, ATEONDE
4: lógico RESP
```

©88-08, Pedro Kantek

```
5: RESP ← .V.
6: enquanto ((K < LIMITE) ∧ RESP) faça
7:    CANDMUL ← PREFHOM [HOM] [K]
8:    K++
9:    se (  SOLTEIRO[CANDMUL]) então
10:      RESP ← CLASMUL[CANDMUL] [HOM] > CLASMUL [CANDMUL] [HOMEM[CANDMUL]]
11:   fimse
12: fimenquanto
13: K ← 0
14: ATEONDE ← CLASMUL [MUL] [HOM]
15: enquanto ((K < ATEONDE) ∧ RESP faça
16:   CANDHOM ← PREFMUL [MUL] [K]
17:   K++
18:   se (CANDHOM < HOM) então
19:      RESP ← CLASHOM [CANDHOM] [MUL] > CLASHOM[CANDHOM] [MUL-
         HER[CANDHOM]]
20:   fimse
21: fimenquanto
22: devolva RESP
```

```
1: Algoritmo CASAMENTO
2: inteiro PREFHOM [n] [n]
3: inteiro PREFMUL [n] [n]
4: inteiro CLASHOM [n] [n]
5: inteiro CLASMUL [n] [n]
6: inteiro HOMEM [n]
7: inteiro MULHER [n]
8: lógico SOLTEIRO [n]
9: inteiro I,J
10: SOLTEIRO ← .VERDADEIRO.
11: leia PREFHOM, PREFMUL
12: PREFHOM ← PREFHOM - 1
13: PREFMUL ← PREFMUL - 1
14: para i de 0 até n-1 faça
15:   para j de 0 ate n-1 faça
16:      CLASHOM[I] [PREFHOM[I] [J]] ← J
17:      CLASMUL[I] [PREFMUL[I] [J]] ← J
18:   fimpara
19: fimpara
20: TENTATIVA (0)
```

Um exemplo, extraído de N. Wirth, Algoritmos e Estruturas de Dados, pág. 129. Sejam as seguintes matrizes de preferências

$$\text{PREFHOM}=\begin{matrix}7&2&6&5&1&3&8&4\\4&3&2&6&8&1&7&5\\3&2&4&1&8&5&7&6\\3&8&4&2&5&6&7&1\\8&3&4&5&6&1&7&2\\8&7&5&2&4&3&1&6\\2&4&6&3&1&7&5&8\\6&1&4&2&7&5&3&8\end{matrix}$$

 ©88-08, Pedro Kantek

$$\text{PREFMUL}=\begin{matrix}4&6&2&5&8&1&3&7\\8&5&3&1&6&7&4&2\\6&8&1&2&3&4&7&5\\3&2&4&7&6&8&5&1\\6&3&1&4&5&7&2&8\\2&1&3&8&7&4&6&5\\3&5&7&2&4&1&8&6\\7&2&8&4&5&6&3&1\end{matrix}$$

Tais matrizes deram origem às seguintes:

$$\text{CLASHOM}=\begin{matrix}5&2&6&8&4&3&1&7\\6&3&2&1&8&4&7&5\\4&2&1&3&6&8&7&5\\8&4&1&3&5&6&7&2\\6&8&2&3&4&5&7&1\\7&4&6&5&3&8&2&1\\5&1&4&2&7&3&6&8\\2&4&7&3&6&1&5&8\end{matrix}$$

$$\text{CLASMUL}=\begin{matrix}6&3&7&1&4&2&8&5\\4&8&3&7&2&5&6&1\\3&4&5&6&8&1&7&2\\8&2&1&3&7&5&4&6\\3&7&2&4&5&1&6&8\\2&1&3&6&8&7&5&4\\6&4&1&5&2&8&3&7\\8&2&7&4&5&6&1&3\end{matrix}$$

A partir destes dados, o resultado obtido foi

sol.	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	p. masc	p. fem
1	7	4	3	8	1	5	2	6	16	32
2	2	4	3	8	1	5	7	6	22	27
3	2	4	3	1	7	5	8	6	31	20
4	6	4	3	8	1	5	7	2	26	22
5	6	4	3	1	7	5	8	2	35	15
6	6	3	4	8	1	5	7	2	29	20
7	6	3	4	1	7	5	8	2	38	13
8	3	6	4	8	1	5	7	2	34	18
9	3	6	4	1	7	5	8	2	43	11

A primeira linha, deve ser lida assim: a mulher 7 está casada com o homem 1. A mulher 4 está casada com o 2, a mulher 3 com o homem 3, e assim por diante, até a mulher 6 que está casada com o homem 8.

Para fazer este confuso pareamento, use o seguinte truque:

- Escreva uma lista com os homens em ordem
- embaixo de cada um, escreva o nome da mulher cujo número aparece na posição do vetor considerado.

Para este casamento, a perda dos homens é de 16 e a perda das mulheres é de 32. Da maneira como o algoritmo está construído, a última linha oferece a melhor opção para as mulheres: a mulher 3 está casada com o homem 1, a 6 com o 2, ... , e a 2 com o homem 8. Esta solução apresenta uma perda de 11 para as mulheres.

©88-08, Pedro Kantek

EXERCÍCIO 15 Considere o seguinte algoritmo [Hel86, pág. 206]

```
inteiro função AI(inteiro N)
imprima ("a função entrou com ",N)
se N = 0 então
  devolva 1
senão
  se N = 1 então
    devolva 2
  senão
    se N= 2 então
      devolva 3
    senão
      devolva AI(N-2) × AI(N-4)
    fimse
  fimse
fimse
fim função
algoritmo TESTE
imprima AI(8)
```

Obs: Antes de rodar em computador procure predizer qual vai ser a seqüência de execuções. Use uma pilha para isso.

EXERCÍCIO 16 Considere o seguinte algoritmo [Hel86,207]

```
função R(inteiro X, Y)
se  Y > 0 então
  X++
  Y–
  imprima (X,Y)
  R(X,Y)
  imprima(X,Y)
fimse
```

Quanto é R(5,3) ?

EXERCÍCIO 17 Estude a função que escreve números em OCTAL

```
função IMPROCT (inteiro N)
se N > 0 então
  se rfloor N ÷ 8 > 0 então
    IMPROCT (rfloor N ÷ 8)
  fimse
  imprima (N mod 8)
fimse
```

Quanto é IMPROCT (100) ?

EXERCÍCIO 18 Procure em QUALQUER livro de estrutura de dados os algoritmos abaixo. Faça sua implementação em C ou Pascal e perca umas boas 2 horas entendendo o seu funcionamento:

- Torres de Hanói
- Quicksort
- MDC (definição de Euclides)

 ©88-08, Pedro Kantek

Georg Cantor e o Infinito

Nascido em São Petersburgo, passou a maior parte da vida na Alemanha. Seu pai converteu-se ao protestantismo, e sua mãe nasceu católica, mas ambos eram de ascendência judia. O interesse inicial de Cantor foi pelos argumentos medievais sobre continuidade e infinito. O pai queria que ele fosse engenheiro, mas ele preferiu algo bem mais sutil. Estudou filosofia, física e matemática e doutorou-se em Berlim em 1867, com uma tese sobre a teoria dos números. Desde a Grécia antiga se falava em infinito, mas parece que ninguém sabia direito do que se falava. A primeira definição formal de infinito é devida a Dedekind (um sistema S é infinito quando é semelhante a uma parte própria dele mesmo). Em 1874, Cantor se casou e na sua lua de mel foi à Interlaken onde estava Dedekind. Eles se encontraram e logo depois ele publicou um artigo revolucionário. Concordou com Dedekind na definição de infinito, mas discordou dizendo que há infinitos e infinitos. No caso dos finitos, dizemos que dois conjuntos tem a mesma cardinalidade se seus elementos podem ser colocados em correspondência bi-unívoca. Cantor levou esse mesmo conceito para os conjuntos infinitos. Por exemplo, pode-se provar que há tantos números pares quanto números inteiros. Ambos são infinitos, mas tem a mesma cardinalidade, já que a relação n <-> p, sempre pode ser estabelecida (n é inteiro e p é um par, sendo que p = 2n). Como ambos são conjuntos infinitos, essa relação sempre pode ser estabelecida e com isso, pode-se concluir que ambos conjuntos tem a mesma cardinalidade.

Outro exemplo: sempre se achou que os racionais (na forma p/q) seriam em maior número que os inteiros. Já que entre dois racionais sempre é possível inserir um novo. Graças a um arranjo triangular bem sacado, Cantor provou que os racionais também podem ser postos em equivalência com os inteiros e portanto são enumeráveis e portanto a cardinalidade dos racionais é a mesma dos inteiros. Mesmo os algébricos que são mais gerais que os racionais, continuam tendo a mesma cardinalidade. Pode-se pensar que todos os infinitos tem a mesma cardinalidade, mas novamente Cantor provou que não é verdade. Os reais tem cardinalidade maior. A prova é simples e elegante, por reductio ad absurdum. Suponhamos que eles são contáveis.

Então será possível criar uma tabela do tipo

$$\begin{array}{llll} 0,a_{11} & a_{12} & a_{13} & ... \\ 0,a_{21} & a_{22} & a_{23} & ... \\ 0,a_{31} & a_{32} & a_{33} & ... \\ ... & ... & ... & ... \end{array}$$

Cantor propôs um real $0,b_1b_2b_3...$ construído segundo a regra: sempre que a_{kk} for igual a 1, b_k será 5, e se a_{kk} for diferente de 1, b_k será 1. Esse número assim construído será um real compreendido entre 0 e 1, e no entanto não fará parte da lista que supostamente tinha todos os reais. Logo...

A conclusão é que são os números transcendentes que dão aos reais essa característica de enumerabilidade. Ele chamou a cardinalidade dos inteiros de aleph zero, e a dos reais de continuum. Ainda não se sabe se existem cardinalidades entre ??? e C, embora saiba-se existirem infinitos números de cardinalidade depois de C. Ele trabalhou numa aritmética de números cardinais, onde por exemplo ?+1 ? 1+?, e ?+?=?, onde ? é um número cardinal. Alguns resultados do trabalho de Cantor eram tão paradoxais que ele mesmo escreveu "vejo isso, mas não acredito". Isso aliado à profunda dificuldade que se tinha em trabalhar com o infinito, retardou o reconhecimento merecido por ele. Embora trabalhasse com cuidado e rigor ocasionalmente se entregava a raciocínios teológicos. Era temperamental e sensitivo, ainda mais quando acusado. Em 1884 sofreu o primeiro dos esgotamentos ner-

©88-08, Pedro Kantek

vosos que o perseguiriam pelos 33 anos restantes de sua vida. Morreu em 1918 num hospital para doentes mentais. Como se pode ver, genialidade e loucura andam sempre meio próximas. Entretanto, Hilbert, outro grande matemático do século XX escreveu "Ele entrou onde almas tímidas tinham hesitado. Ninguém nos expulsará do paraíso que Cantor criou para nós".

3.6 Definição de árvore

Embora venha a ser objeto de estudo no terceiro bimestre, para a aula de hoje se necessita uma definição operacional da estrutura de dados conhecida como árvore.

A árvore é uma das estruturas de informação mais utilizadas e úteis. Como tantas coisas na informática a árvore aproveitou um nome já existente para batizar uma coisa nova quando esta foi inventada, e como sempre buscou-se algo que tivesse semelhança com o conceito novo. A estrutura em árvore é uma estrutura hierárquica, na qual um segmento é o principal e dele se derivam todos os demais. Esse mesmo desenho se repete nos segmentos derivados, e a semelhança que existe com uma árvore tradicional ocorre quando se compara esse segmento básico ao tronco de uma árvore, os demais aos galhos, até que finalmente se chega às folhas.

Árvores ocorrem no nosso dia a dia. Veja-se por exemplo, a tabela das semifinais de qualquer copa mundial de futebol.

Outro exemplo de árvore está no sumário deste livro. Veja às págs 2 a 8, como os assuntos do livro se concatenam e formam um todo hierárquico. Outros exemplos: uma expressão aritmética tipo $((1 + 2) \times (3/4)) - 5$. Ou então uma árvore genealógica. Finalmente, um exemplo vistoso e já estudado é o da alocação de espaços em disco pelo sistema operacional UNIX .

Pode aplicar-se então, a seguinte definição: Uma árvore é uma estrutura de informação que é constituída por nodos, que são os itens elementares dos dados, e onde os nodos se hierarquizam de alguma maneira pré-estabelecida.

Esta hierarquia assume características que podem ser estudadas fazendo-se analogia com a relação "pai-filho"(ou mãe-filho, para as feministas).

Diz-se então que dois nodos estão ligados através de uma relação pai ou filho. Por exemplo, se uma árvore contém dois nodos, A e B, pode-se ter A sendo pai de B ou A sendo filho de B.

Todos os nodos (com exceção de um deles) terão um pai. Já os nodos podem ter ou não ter filhos. O único nodo que não tem pai é conhecido como nodo raiz.

Uma segunda definição, esta recursiva para árvore é: Uma árvore A é uma estrutura definida por

- Uma estrutura vazia OU
- um nodo raiz, ao qual estão vinculadas sub-árvores

A seguir algumas definições importantes

raiz O único nodo da árvore que não tem pai

folha Todos os nodos da árvore que não tem filhos

altura da árvore O maior caminho existente entre uma folha e a raiz da árvore

grau da árvore o número máximo de filhos de qualquer nodo da árvore

 ©88-08, Pedro Kantek

3.7 Quad Tree

Esta interessante estrutura apareceu nos anos 70 e assinalou um novo estágio de progresso na computação gráfica. Muitas operações tais como armazenamento, manipulação ou análise de imagens digitais foram melhoradas. Um conjunto completo de novas operações tornou-se possível, sempre pelo expediente simples de subdividir uma imagem digital em quadrantes assinalando cada um desses quadrantes a um nodo de uma árvore, a *Quad Tree* ou Árvore de Quadrantes.

Considere a figura **??**. Trata-se de um gato (e um rato) digitais. Ambos podem ser decompostos (apenas sua imagem) em uma árvore quad. Usando esta estrutura, pode-se armazenar a imagem de maneira muito eficiente, permitindo diversas transformações geométricas sobre ela e permitindo descobrir quais são os objetos (partes) que pertencem ao gato

```
Acompanhe no exemplo: Seja a imagem
16 x 16 pixels (.=branco; x=preto)

                        1 1 1 1 1 1 1
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
    - - - - - - - - - - - - - - - -
 1-. . . . . . . . . . . . . . . .
 2-. . . . . . . . . . . . . . . .
 3-. . . . . . . . . . . . . . . .
 4-. . . . . . . . . . . . . . . .
 5-. . . . . . . . . . . . . . . .
 6-. . . . . . . . . . . . . . . .
 7-. . . . . . . . . . . . . . . .
 8-. . . . . . . . . . x x x x . .
 9-. . . . . . . . . . x x x x . .
10-. . . . . . . . . . x x x x . .
11-. x x x x . . . x x x x x x . .
12-. x x x x . . . x x x x . . . .
13-. x x x x . . . x x x x . . . .
14-. x x x x . . . x x x x . . . .
15-. . . . . . . . . . . . . . . .
16-. . . . . . . . . . . . . . . .
```

```
Esta imagem geraria a seguinte
 rvore QUAD:
   1- .  2  7  16
   2- .  .  3  5
   3- .  .  .  4
   4- .  .  x  x
   5- .  .  6  .
   6- .  .  x  x
   7- 8  10 12 14
   8- .  .  9  x
   9- .  x  .  x
  10- .  .  11 .
  11- x  .  x  .
  12- 13 x  .  .
  13- .  x  .  x
  14- 15 .  .  .
  15- x  .  x  .
  16- 17 18 20 .
  17- .  x  x  x
  18- x  .  19 .
  19- x  x  .  .
  20- x  x  .  .
```

```
Seja agora a imagem de um gato
                        1 1 1 1 1 1 1
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
    - - - - - - - - - - - - - - - -
 1-. . . . . . . . . . . . . . . .
 2-. . . . . . . . . . . . . . . .
 3-. . . . x . . . . . . . . . . .
 4-. . x x x . . . . . . . . . . .
 5-. . x x x x . . . . . . . . . .
 6-. . . . x x x x . . . . . . . .
 7-. . . . . x x x x . . . . . . .
 8-. . . . . x x x x x . . . x x .
 9-. . . . . x . x x x x . . . x .
10-. . . . . x . . x x x . . . x .
11-. . . . . x . . x x x . . x x .
12-x x . . x x . x x x x x x x . .
13-. . . . . x . . x x x . . . x .
14-. . . . . x . . x x x . . . x .
15-. . . . . x x x x x . . . . . .
16-. . . . . x x x x . . . . . . .
```

```
 1-2  12 17 25
 2-3  5  7  9
 3-.  .  .  4
 4-.  .  x  x
 5-.  .  6  .
 6-x  .  x  .
 7-.  8  .  .
 8-x  x  .  .
 9-x  10 11 x
10-.  .  x  x
11-.  x  .  x
12-.  .  13 15
13-.  .  x  14
```

```
16-.  .  x  x
17-18 20 .  .
18-.  .  19 .
19-.  .  x  x
20-21 22 23 24
21-.  x  .  x
22-.  x  .  .
23-.  x  x  x
24-.  .  .  x
25-x  26 .  .
26-.  27 28 29
27-.  x  .  x
28-.  .  x  x
```

©88-08, Pedro Kantek

```
13-. . . . . .  14-.  .  x  .    29-x  x  x  .
14-. . . . . .  15-.  .  .  16
15-. . . . . .
16-. . . . . .
```

Na árvore Quad (a imagem do) nosso gato é dividida em 4 quadrantes (daí o nome) e depois cada quadrante é subdividido em 4 quadrantes e assim por diante. Essa é a razão pela qual ela é estudada no tópico "recursividade". O último estágio da subdivisão é alcançado quando um único pixel torna-se um quadrante. Esta estratégia é mais eficiente quando as dimensões da imagem são valores iguais a uma potência de 2.

Cada quadrante é representado por um nodo na árvore quad. Se o quadrante é todo branco ou preto, o nodo é adequadamente identificado e tratado como nodo folha. Se ao contrário no quadrante existem pixels brancos e pretos misturados, então cria-se um novo nodo na árvore contendo 4 filhos que serão os 4 sub-quadrantes do quadrante. Veja-se a seguir a árvore QUAD do quadrante B do desenho do gato A árvore anexa é uma parte

da imagem do gato quando interpretada adequadamente. Um nodo terminal branco no nível k, contado a partir de baixo representa um quadrante de $2^k \times 2^k$ inteiramente branco. Identicamente para o preto. Há economia: o gato pode ser armazenado em 41 nodos, ao invés de exigir o armazenamento de 16×16 pixels. Para um grande número de imagens este é um coeficiente de economia comum.

Algumas transformações são facilitadas: A rotação da imagem em 90^o é fácil: basta obter os nodos da árvore devidamente rotacionados também (por exemplo, obter DABC ao invés do convencional ABCD).

Para mudar a escala da imagem, basta remover todos os nodos folha da árvore, interpretando o nodo pai destes de maneira apropriada. Este processo diminui a imagem por um fator 2.

Uma operação importante é a localização de componentes. No exemplo, há 2 componentes: o gato e o rato. Esta funcionalidade é muito importante em aplicações médicas, por exemplo, na contagem de células em uma amostra.

O algoritmo localiza um pixel preto e pesquisa vizinhos pretos ao norte, leste, sul e oeste. A pesquisa prossegue montando uma lista de todos os pixels pretos.

3.7.1 Algoritmos

A função que cria a árvore QUAD pode ser assim escrita

```
1: função CRIAQUAD (inteiro NUMERO, array IMA)
2: se linhas(ARRAY) ≤ 2 então
3:    AQUAD[NUMERO;1 2 3 4] ← IMA[1;1],IMA[1;2],IMA[2;1] ,IMA[2;2]
4: senão
5:    Q1 ← quadrante(1,IMA)
6:    Q2 ← quadrante(2,IMA)...
7:    se tudoigual(Q1) então
8:       AQUAD[NUMERO;1] ← IMA[1;1]
9:    senão
10:      PROX++
11:      AQUAD[NUMERO;1] ← PROX
12:      CRIAQUAD (PROX, Q1)
13:   fimse
14: fimse
15: fim função
```


Para chamar, deve-se criar AQUAD como matriz de 64 por 4 contendo zeros,colocar 1 em PROX, e chamar CRIAQUAD(1, GATO).

A função que lê a árvore e cria o desenho é a seguinte: Antes, deve-se criar a matriz VOLTA contendo 16x16 brancos

```
1: função DESENHA (inteiro LINHA, matriz ONDE)
2: inteiro xx ← AQUAD[LINHA]
3: se xx[1] = -1 OU xx[1] = -2 então
4:    VOLTA[retiraquadrante (ONDE, 1)] ← (branco,preto)[abs (xx[1])]
5: senão
6:    xx[1] DESENHA retiraquadrante (ONDE,1)
7: fimse
8: fim função
```

Desafio

- Implementar uma árvore QUAD e guardar a imagem do gato. Comparar os consumos de memória entre a árvore quad e a imagem BMP correspondente.
- Pensar e bolar uma árvore SEX, com grau=6, e na qual a imagem é dividida em 6 sextantes.
- Pense no processo necessário para ampliar uma imagem guardada em árvore QUAD por um fator de 2.

Capítulo 4

Mecanismos de alocação

4.1 Algoritmos de alocação de área livre

4.1.1 First Fit

Este algoritmo percorre a lista de áreas livres disponíveis na memória, até encontrar o primeiro espaço livre que acomode a requisição de memória que foi efetuada. Supondo que uma requisição de 120 inteiros tenha sido feita, este algoritmo entregaria a primeira área livre encontrada que fosse maior ou igual a 120. A sigla FIRST-FIT pode ser traduzida por o "primeiro que cabe ". A principal vantagem dessa estratégia é a rapidez com que as solicitações são atendidas. Como desvantagem pode-se apontar o caráter randômico das alocações, o que deixa ao acaso o mapa geral de alocações.

4.1.2 Best Fit

Este algoritmo percorre toda a lista de áreas livres fazendo a alocação pedida naquela área que minimize a sobra de espaço (a quantidade de inteiros perdidos ao final do bloco alocado). Por exemplo, digamos que uma requisição de 80 inteiros tenha sido feita, e ao percorrer a lista de áreas livres tenham sido encontradas as seguintes: 100, 280, 90, 500 e 320. Este algoritmo escolheria a terceira área, que é a que menor área desocupada deixaria ao final (apenas 10 inteiros). A sigla BEST-FIT pode ser traduzida por "o menor que cabe ". A vantagem dessa estratégia é a conservação de blocos grandes. Por outro lado, ele tem como desvantagem a necessidade de pesquisar toda a lista a cada alocação, além de fragmentar a memória em muitos blocos pequenos, cada um deles podendo ter a sua sobra ao final. Este esquema privilegia as grandes alocações, no sentido de que requisições grandes tem boa chance de serem atendidas.

4.1.3 Worst Fit

Este algoritmo percorre toda a lista de áreas livres e aloca a maior área que tenha sido encontrada na cadeia de áreas livres e que comporte a requisição de espaço solicitada. Usando o mesmo exemplo do caso anterior (solicitando-se 80, e comportando a lista os valores 100, 280, 90, 500 e 320) seria alocada a área de 500 inteiros, que passaria a contar agora apenas com 420 inteiros. A sigla WORST-FIT pode ser traduzida por "o pior que cabe ". A vantagem desta estratégia é que ela não gera blocos muito pequenos, mas em compensação ela tende a fragmentar os blocos grandes, além é claro, de ter que percorrer toda a cadeia de áreas livres para achar "a pior".

4.1.4 Variantes

Como toda algoritmo associado a uma estrutura de dados, as estratégias acima comportam diversas modificações. O objetivo sempre é encontrar um ponto de equilíbrio entre objetivos opostos quais sejam: minimizar o tempo gasto na alocação e maximizar o uso da memória. As variantes mais conhecidas são:

- Existência de um valor mínimo de alocação. Sempre que a diferença entre o valor pedido na alocação e o tamanho do bloco selecionado para atender aquela alocação for menor que um valor fixo, todo o bloco será alocado. Essa modificação, se de um lado desperdiça certas áreas ao final das alocações, por outro lado mantém a lista de blocos livres pequena - ou pelo menos, menor do que seria sem esse conceito.
- Ajuste de tamanho . Antes de se proceder ao atendimento da solicitação de alocação, o tamanho pedido é arredondado - para cima - para um valor múltiplo de alguma constante. Essa estratégia também garante a minimização de pequenos pedaços ao final dos blocos alocados.
- Encontro imediato . Nos algoritmos de BEST-FIT ou WORST-FIT, sempre que na pesquisa é encontrado um bloco com tamanho igual ao pedido (ou com diferença menor que uma dada constante) o bloco é imediatamente alocado, sem ter que se proceder à toda pesquisa na lista de blocos livres.
- Segregação de memória. Os blocos são previamente criados e posteriormente não são divididos. Por exemplo, cria-se uma coleção de blocos de 100 inteiros, outra de 500, outra de 1000 e assim por diante.

4.1.5 Liberação de blocos de memória

Como vimos, para qualquer uma das abordagens de gerência dos blocos de memória é necessária uma estrutura de dados que encadeie as diversas áreas que estão sendo controladas. A questão é qual estrutura usar. São candidatas: pilha, fila e lista em geral. Analisemos cada uma dela.

De início, é necessário ressaltar que as considerações que vem a seguir se aplicam basicamente ao algoritmo BEST-FIT, que é o que mais privilegia a rapidez no acesso e pesquisa à cadeia de áreas livres. Isso não significa que os outros algoritmos não precisem se preocupar com rapidez e facilidade de acesso. Só que nos outros dois, toda a cadeia sempre é percorrida, o que minimiza a questão de achar rapidamente o melhor bloco.

Se a estrutura de controle das áreas livres for uma pilha , e o algoritmo for o FIRST-FIT, o resultado será pobre, no sentido que, como as desalocações são aleatórias, eventualmente será necessário desempilhar uma porção de segmentos - até encontrar aquele que suporte a alocação, e depois será necessário reempilhar todos os que não serviram. A vantagem que se pode apontar é que o bloco final da memória só será fragmentado quando realmente necessário. Isso só ocorrerá quando nenhum dos blocos alocados e já liberados, servir. Já se os blocos tiverem tamanhos pré-alocados, essa pode ser uma boa alternativa de implementação.

Se a estrutura for uma pilha e o algoritmo o WORST-FIT, nem essa vantagem servirá. Uma outra desvantagem da pilha, para qualquer dos algoritmos é que os blocos estarão ordenados por ordem de liberação, que como vimos é aleatório. Isto complica sobremaneira a tarefa de reintegração de blocos.

Se a escolha for uma fila , bem poucas vantagens se podem apontar, a menos que a disciplina de liberação deixe de ser randômica e passe também a obedecer a disciplina FIFO. Continua havendo a dificuldade de reintegração de blocos.

A solução óbvia para este problema é a utilização de uma lista genérica, tal como vimos no exemplo prático acima, e mais do que isso, manter essa lista ordenada por

©88-08, Pedro Kantek

endereços de memória. Agindo dessa maneira, a reintegração de blocos passa a ser simplificada, pois a contigüidade de blocos saltará aos olhos (dois blocos logicamente contíguos serão também fisicamente contíguos) quando a lista for examinada.

4.1.6 Lista ordenada por endereço

Supondo que o primeiro inteiro do bloco livre seja o endereço do próximo, vamos chamar esse inteiro de PROXIMO(X), onde X é o bloco em questão. Vamos chamar o endereço inicial deste bloco de ENDEREÇO(X), e o tamanho do bloco de TAMANHO(X). A lista estará ordenada por ENDEREÇO. Note-se que quando dois blocos forem contíguos fisicamente o PRÓXIMO do primeiro será exatamente igual ao ENDEREÇO do segundo.

Quando um bloco qualquer, digamos Y, for liberado a lista deve ser percorrida, até ser encontrado à esquerda um bloco cujo endereço seja menor que ENDEREÇO(Y) e a direita esteja um bloco cujo endereço seja maior.

```
Exemplificando. Seja a cadeia de blocos livres:
Bloco A: endereço 200, próximo 350, tamanho 100
Bloco B: endereço 350, próximo 600, tamanho 40
Bloco C: endereço 600, próximo 890, tamanho 150
```

E imaginemos a liberação de um bloco K, de endereço 500, com tamanho 50.
Note-se que o local de inserção de K é depois de B e antes de C. Vale a relação dos endereços: B < K < C. Mais ainda note-se que - para rapidez de acesso, o bloco C não precisa ser visitado, já que na relação acima, ele pode ser substituído por PRÓXIMO(B). A relação correta passa a ser: B < K < PRÓXIMO(B). A vantagem dessa relação é que o local de inserção do bloco recém liberado pode ser encontrado examinando-se apenas um elemento por vez.

A inserção pode ser facilmente executada, fazendo-se as seguintes atribuições:

```
PRÓXIMO(bloco recém liberado) := PRÓXIMO (bloco anterior)
PRÓXIMO(bloco anterior) := ENDEREÇO (bloco recém liberado).
```

Vejamos essa atribuição no exemplo de cima:

```
PRÓXIMO(K)  :=  600 (é o PRÓXIMO(B))
PRÓXIMO(B)  :=  500 (é o ENDEREÇO(K)),
e agora a lista passa a ser:

Bloco A: endereço 200, próximo 350, tamanho 100
Bloco B: endereço 350, próximo 500, tamanho 40
Bloco K: endereço 500, próximo 600, tamanho 50
Bloco C: endereço 600, próximo 890, tamanho 150
```

4.1.7 Reintegração de blocos

Suponhamos a existência de dois elementos na lista de blocos livres. Para efeitos didáticos, vamos chamá-los de ESQUERDO e de DIREITO. Nesse momento, vem a liberação de um bloco de endereço intermediário entre ambos, a quem vamos chamar MEIO. Examinar-se-ão agora as condições em que eles podem ser integrados em blocos maiores.

1. O primeiro caso a ser examinado é quando ESQUERDO se integra com MEIO, isto é, quando eles são fisicamente contíguos. Essa condição é dada pela relação
ENDEREÇO(ESQUERDO) + TAMANHO(ESQUERDO)
= PRÓXIMO(ESQUERDO)

Nesse caso, o processo de reintegração será dado pelas seguintes atribuições
TAMANHO(ESQUERDO) ← (TAMANHO(ESQUERDO) +
TAMANHO(PRÓXIMO(ESQUERDO))
PRÓXIMO(ESQUERDO) ← PRÓXIMO(PRÓXIMO(ESQUERDO))

2. O segundo caso é quando MEIO se integra com DIREITO. A condição é dada pela relação
ENDEREÇO(MEIO) + TAMANHO(MEIO) = PRÓXIMO(MEIO)

O processo de integração segue as seguintes atribuições
TAMANHO(MEIO) ← TAMANHO(MEIO) + TAMANHO(PRÓXIMO(MEIO))
PRÓXIMO(MEIO) ← PRÓXIMO(PRÓXIMO(MEIO)).

3. Finalmente, o terceiro caso ocorre quando o bloco MEIO preenche exatamente o espaço existente entre ESQUERDO e DIREITO. Agora os três blocos passarão a formar um único. A condição que estabelece essa situação é
ENDEREÇO(ESQUERDO) + TAMANHO(ESQUERDO) =
PRÓXIMO(ESQUERDO) E
ENDEREÇO(MEIO) + TAMANHO(MEIO) = PRÓXIMO(MEIO)

Note-se que os dois testes acima são rigorosamente os mesmos dos dois casos anteriores, conectados por um "E". Isso sugere, que no processo de reintegração, a primeira coisa que se deve fazer é testar as duas condições, antes de fazer qualquer coisa. Se ambas forem verdadeiras, parte-se logo para o terceiro caso. Se uma delas for falsa, executa-se o caso correspondente (se a primeira for falsa, usa-se o caso 2. Se a segunda for falsa, usa-se o caso 1.

O processo de reintegração segue as seguintes atribuições:
TAMANHO(ESQUERDO) ← TAMANHO(ESQUERDO)+
TAMANHO(MEIO)+TAMANHO(PRÓXIMO(MEIO)
PRÓXIMO(ESQUERDO) ←
PRÓXIMO(PRÓXIMO(PRÓXIMO(ESQUERDO)))

4.1.8 Memória M

Trata-se de uma simplificação didática a fim de estudar o comportamento da memória real. Imagine um computador didático que tem uma memória apenas de 100 inteiros. Como é uma memória pequena ela pode ser facilmente manuseada e vista.

Seu aspecto é

M	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

A memória M é criada pela função AMBICRM (que significa: função do AMBIente que CRia a memória M). Ela devolve 0 se a criação for bem sucedida e -1 senão.

Ela sempre se inicia no endereço zero e se encerra no 99. Apenas números inteiros podem ser armazenados, embora – como é um artefato artificial – as vezes se coloca uma

letra ou um nome curto sem nenhum problema.

O mecanismo de alocação nesta memória é bastante simples. Trata-se de uma variável global intitulada PPL acrônimo que significa Próxima Posição Livre. Quando a memória M é criada, PPL sempre é inicializado com o valor zero. A medida em que alocações vão sendo feitas, PPL vai crescendo.

A função que aloca área recebe o nome AMBIOBT (é uma função do AMBIente que fetua OBTenção de memória). Esta função:

- Recebe um número com a quantidade de inteiros pedidos
- Verifica se os há disponíveis (consultando PPL)
- Se a alocação for possível (há espaço), acerta o PPL e devolve o endereço inicial de alocação
- Se não for possível, retorna -1 e deixa PPL inalterado.

O endereço que AMBIOBT devolve é importante pois passa a ser usado para acessar a área que foi recém alocada.

Como M é uma simplificação de memória real, não há mecanismos de desalocação nem de reaproveitamente de memória, pelo menos neste estágio inicial.

Todos os algoritmos que forem estudados para M, poderão ser usados com alterações mínimas em qualquer memória real.

4.2 Alocação

As primeiras estruturas de dados que aprenderemos são as chamadas listas . O nome lista indica uma estrutura genérica, na qual elementos de formatos iguais ou semelhantes vão sendo usados do desenvolvimento dos algoritmos que os manuseiam. O termo lista é usado de maneira genérica, para indicar estruturas cujos itens elementares se repetem, sempre no mesmo formato. São exemplos de listas dessa natureza, os tipos vetor e matriz, que estudamos anteriormente, a memória M já vista, (na verdade qualquer memória de qualquer computador é uma lista), uma variável do tipo alfanumérica, um arquivo,... todos esses elementos podem ser vistos como listas.

As listas podem ser implementadas basicamente de duas maneiras distintas, a chamada alocação seqüencial e a alocação encadeada . Naturalmente, essa é uma primeira divisão, e novamente aqui, como em todo o universo do software, o limite de implementação e funcionamento das listas sempre é a imaginação humana (ou seja, não há limite). Mas, para efeitos didáticos, e para manter na mente nosso objetivo de complexidade crescente, vamos começar examinando apenas essas duas implementações básicas

4.2.1 Alocação seqüencial

A diferença básica entre uma lista seqüencial e uma lista encadeada é que a primeira tem os elementos pertencentes à lista em posições contíguas da memória. É essa contigüidade ou vizinhança que dá as principais características à alocação seqüencial. Já a alocação encadeada, pressupõe que os elementos poderão se distribuir ao longo da memória, ocupando espaços (buracos) livres, e sem que o elemento de ordem "k"esteja antes do elemento de ordem "k+1", e depois do elemento de ordem "k-1".

Uma lista seqüencial é reconhecida pelos seguintes elementos

- S_i = endereço inicial da lista seqüencial
- C = comprimento de cada elemento da lista

 ©88-08, Pedro Kantek

- N_{max} = número máximo de elementos que a lista poderá conter.

Um exemplo desse tipo de estrutura é a do tipo vetor. Quando definimos um vetor para conter os 12 totais mensais de vendas de uma empresa (portanto ponto flutuante) ao longo de um determinado ano, estamos definindo os 3 itens da lista, a saber:
$S + i$ = a localização do vetor dentro do programa (memória). É o endereço inicial
C = o espaço ocupado por um elemento ponto flutuante da linguagem que estiver sendo usada. É o comprimento
$N_{max} = 12$, já que são 12 meses.

Graças a essas 3 definições, existe uma fórmula básica de acesso aos elementos de uma lista seqüencial e que pode ser assim descrita $S_k = S_i + (k-1) \times C$ onde S_k indica a localização do k-ésimo elemento da lista seqüencial, S_i indica a localização do primeiro elemento da lista (ou mais propriamente, da lista seqüencial), k é o índice do elemento que se deseja acessar e C é o comprimento de cada elemento.

Vejamos um exemplo numérico. Suponhamos uma lista seqüencial que começa no endereço 30 de M, e cujo comprimento dos elementos é de 4 inteiros. A capacidade máxima da lista é de 6 elementos. Qual a posição do 3º elemento dessa lista ? $S_k = S_i + (k-1) \times C$, ou $S_3 = 30 + (3-1) \times 4$, ou $S_3 = 30 + 2 \times 8$, ou $S_3 = 46$. Como o elemento tem 4 inteiros, o terceiro elemento ocupa as posições 46..49.

A principal vantagem da alocação seqüencial é a simplicidade dos algoritmos. Não tanto em termos de escrita (como veremos depois, alocações encadeadas, às vezes geram algoritmos menores e mais elegantes), mas sim em termos de entendimento. É por essa razão, que a maioria dos autores (e este livro segue a tendência) começam o estudo das estruturas de dados por aqui.

Criada uma estrutura seqüencial, é fácil visualizá-la na memória (principalmente se for a nossa M), e é mais fácil ainda entender os mecanismos de manuseio dessa estrutura.

Como tudo na vida tem um preço, a desvantagem da alocação seqüencial é a necessidade prévia de determinar tamanhos máximos para a estrutura antes do início do seu uso. Para estruturas pequenas, esse não é um problema sério, pelo que, para estruturas pequenas, sempre recomendar-se-á esse tipo de alocação. Entretanto, as estruturas usadas em geral ao longo da vida do profissional tendem muito mais para o grande do que para o pequeno. Bancos de dados, estruturas de auxílio à classificação de arquivos, árvores para os mais diversos usos, devoram o recurso memória com apetite sempre insaciável. Um bom resumo para essa história poderia ser: a alocação seqüencial é importante para iniciar o aprendizado das estruturas e também para uso em estruturas reconhecida e sabidamente pequenas, e que se manterão sempre assim. Fora esses casos, a escolha sempre deverá recair sobre estruturas encadeadas.

Até porque, há um outro custo importante associado às estruturas seqüenciais, que é o da manutenção ordenada da lista. Um bom exemplo, poderia ser o de um dicionário. Suponha que um dicionarista qualquer, mantenha seus originais em cadernos anotados e com as folhas totalmente preenchidas. Como incluir uma nova palavra, mantendo a ordem do dicionário ? Talvez, no caso limite o dicionarista fosse obrigado a copiar fisicamente todo o dicionário para outro caderno, simplesmente para incluir uma nova palavra.

No computador as coisas não precisam acontecer assim, porque as memórias podem ser lidas e escritas um sem-número de vezes sem degradação (seria como se o dicionarista tivesse seus textos escritos à lápis, e o caderno nunca se desgastasse), mas o trabalho de deslocar palavras para cima e para baixo, esse, não há como fugir dele. E mesmo assim, em estruturas seqüenciais, às vezes, para inserir um item em ordem, a maneira mais barata (em termos de recursos computacionais) é recopiar toda a lista, em outra área da memória. Uma maneira de implementar uma lista em um vetor é usando o seguinte

©88-08, Pedro Kantek

esquema:

primeiro elemento
segundo elemento
último elemento
parte vazia

Devidamente acompanhado de uma estrutura auxiliar composta de

último usado
tamanho máximo

O vetor é definido originalmente como vazio. Duas variáveis (a rigor uma constante e uma variável) apontam para o último elemento preenchido e para o tamanho total da lista.

Seja a seguinte estrutura

```
1: estrutura LA
2: inteiro ultimo, tamanho; inteiro elemento[10]
3: fimestrutura
4: LA X
```

Como X seria visualizado vazio ?

0	10	?	?	?	?	?	?	?	?	?	?

Após a inclusão de 13, 21 e 7, nessa ordem, X ficaria

3	10	13	21	7	?	?	?	?	?	?	?

Após a exclusão do 7, X ficaria

2	10	13	21	7	?	?	?	?	?	?	?

EXERCÍCIO 19 Suponha a seguinte definição em português estruturado, na forma de variáveis globais estrutura inteiro ultimo, tamanho; inteiro elemento[100] lista lista A,B,C,D
Escreva o algoritmo de uma função que inicializa A vazia e nada retorna

EXERCÍCIO 20 Escreva o algoritmo de uma função que recebe um inteiro e o inclui na próxima posição disponível de A. Devolve zero se for bem sucedida e -1 se houver erro.

EXERCÍCIO 21 Escreva o algoritmo de uma função que recebe um inteiro e o inclui na próxima posição disponível de A. Se A estiver cheia, tenta a inclusão em B. Devolve zero se a inclusão se der em A, 1 se ela se der em B e -1 se houver erro (B também está lotada)

EXERCÍCIO 22 Escreva o algoritmo de uma função que nada receba e devolva .T. se A está vazia e .F. senão.

EXERCÍCIO 23 Escreva o algoritmo de uma função que não receba nenhum parâmetro e devolva 1, 2, 3 ou 4 dependendo de qual lista tiver menos elementos.

EXERCÍCIO 24 Escreva o algoritmo de uma função que receba um número de elementos que se pretende incluir em A e devolva .V. se isso for possível e .F. senão.

EXERCÍCIO 25 Escreva o algoritmo de uma função que não receba parâmetro e tente distribuir os dados de A a começar da última posição pelas listas B, C e D nesta ordem. Se qualquer uma das 3 lotar, os elementos a ela destinados a partir desse ponto devem ser

 ©88-08, Pedro Kantek

desprezados. Exemplo: Sejam as listas (antes)

A	47	3	88	9	2	7		
B	1	11	21	31	41	51		
C	64							
D	4	16	32	65	128	256	512	

E o resultado (depois)

A								
B	1	11	21	31	41	51	7	88
C	64	2	3					
D	4	16	32	65	128	256	512	9

EXERCÍCIO 26 Escreva o algoritmo de uma função que exclua o último elemento da lista D e devolva-o. Caso D estivesse vazia, devolver -1.

EXERCÍCIO 27 Escreva o algoritmo de uma função que exclua os 4 últimos elementos de A, B, C e D e devolva o maior deles. Se alguma das listas estiver vazia, não deve ser considerada. Se as 4 estiverem vazias, devolver -1.

EXERCÍCIO 28 Escreva o algoritmo de uma função que exclua o primeiro elemento de A.

EXERCÍCIO 29 Escreva o algoritmo de uma função que exclua o primeiro elemento de A que seja maior do que 100.

4.2.2 Alocação encadeada

Na alocação encadeada, não há contigüidade física . Os elementos da lista serão inseridos em qualquer posição da memória. Embora, em geral haja critérios de preferência, como proximidade, otimização de espaços livres, início em endereços múltiplos de certa constante, o esquema teórico da alocação encadeada, em tese, permite a inclusão de um novo elemento na lista em qualquer posição da memória.

Se, o próximo elemento de uma lista encadeada não é o vizinho próximo, como saber onde ele está ? A resposta é pelo uso de apontadores . Trocando em miúdos, cada elemento de uma lista encadeada, tem no seu interior o endereço do próximo elemento da lista. De início, salta aos olhos uma desvantagem, que é a de que cada elemento carregue consigo o endereço do seu próximo. Talvez poderíamos chamá-lo de vizinho lógico, já que fisicamente, em geral, um está longe do outro.

Dito isso, o que caracteriza uma lista encadeada são as seguintes informações:

- O endereço do primeiro elemento (também conhecido como "endereço da lista"), e identificado por E_i.
- A posição do "apontador para o próximo"dentro de cada elemento
- Uma determinada constante, que quando colocada como conteúdo desse apontador, indicará que a lista termina aqui (não há próximo elemento).

©88-08, Pedro Kantek

4.2.3 Diferenças (vantagens e desvantagens)

Critério	Alocação Seqüencial	Alocação Encadeada
Espaço realmente ocupado	Não há perda de espaço . Todos as informações armazenadas na lista são relevantes	No mínimo, perde-se uma posição de memória (que aponta para o primeiro), e para cada elemento existente, perde-se outro apontador
Espaço inicialmente alocado	Há necessidade de pré-estabelecer o tamanho da lista, ainda que muitas vezes parte dela fique inutilizada	Enquanto pequena, ela ocupa pouco. Quando fica grande, passa a ocupar mais espaço. Nada é desperdiçado
Crescimento futuro	Não há possibilidade. O tamanho da estrutura com alocação seqüencial é fixo e não pode ser modificado	Enquanto houver memória livre, a estrutura em alocação encadeada pode crescer sem nenhum limite.
Processamento necessário para "visitar"todos os elementos da lista	Facilitado pelo uso da fórmula de recorrência ($E_k = E_i + (k-1) \times C$)	A visitação precisa percorrer todos os elementos da lista
Pesquisa: achar o primeiro elemento da lista que satisfaz determinado critério		
Inclusão de novo elemento em lista desordenada	Havendo espaço na alocação original da lista, a inclusão é fácil.	Havendo memória, a inclusão é fácil
Inclusão de novo elemento em lista ordenada	Complexo. Há necessidade de localizar qual o espaço adequado para esse elemento, e depois, deslocar todos os elementos subsequentes, para gerar esse espaço.	Relativamente fácil. Localizado o ponto de inserção, basta colocar o novo item em qualquer posição de memória, fazer o anterior apontar para ele, e ele apontar para o seguinte na ordem original
Excluir elemento	Complexo. Implica em deslocar todos os elementos que estão depois do elemento retirado. Lembrar que na alocação seqüencial, não podem ficar "buracos"na estrutura. A contigüidade é imprescindível	Basta fazer o elemento anterior ao retirado apontar para o seguinte ao retirado. Dessa maneira, com muita facilidade "salta-se"o elemento retirado.
Classificação de elementos da lista segundo algum critério.	Sempre implica em movimentar todos os itens da lista	São necessários apenas acertos nos apontadores dos elementos da lista
Concatenação de duas listas	Implica na alocação contígua de um espaço capaz de manter as 2 listas, e na movimentação dos elementos das duas, ou no mínimo, de uma.	Apenas o apontador do último elemento da primeira lista precisa ser alterado

Alguns exemplos

Vamos usar a memória M e criar alguns exemplos de listas seqüenciais e encadeadas.

 ©88-08, Pedro Kantek

Vencedores do mundial de futebol Nosso problema, por hipótese, é manter uma lista de campeões do mundo em futebol nas décadas de 60 a 80. Embora usemos M, que por definição só aceita números inteiros, vamos abrir uma exceção: No lugar de identificar os países por um número, vamos usar suas iniciais. O exemplo ficará mais claro, e como M é uma construção particular nossa, isso pode ser feito sem nenhuma agressão às boas regras de uso da memória.

Vamos supor que as coisas tenham ocorrido assim (dados sobre gols marcados são fictícios):

Ano do Campeonato	Campeão	Gols marcados
62	Brasil	17
66	Inglaterra	16
70	Brasil	20
74	Alemanha	22
78	Argentina	21
82	Itália	17
86	Argentina	19

O elemento da estrutura (lista) que será definida, terá a seguinte estrutura:

1: Nome do País Campeão
2: Ano do Campeonato
3: Gols marcados

Alocação seqüencial A lista vai começar no endereço 0. Precisamos definir o tamanho máximo da tabela. Imaginemos querer guardar, mais tarde, a década de 90 também. Os elementos da lista ficam: Si = 0 C = 3 Nmax = 10 (correspondendo aos anos de 62, 66, 70, 74, 78, 82, 86, 90, 94 e 98)

M1	0	1	2	3	4	5	6	7	8	9
0	Br	62	17	In	66	16	Br	70	20	Al
1	74	22	Ar	78	21	It	82	17	Ar	86
2	19	R	R	R	R	R	R	R	R	R
3										

Note que os espaços correspondentes aos campeonatos de 90, 94 e 98 estão apenas reservados (é o significado da letra "R", e note também que esses espaços estão perdidos até serem usados. O tamanho total ocupado pela lista é de 30 elementos de M. A necessidade de manter o espaço reservado, é a de que, em tese, essa memória também está sendo ocupada para outras atividades. Se nada for feito, quando a inclusão do campeão de 1990 for ser feita, provavelmente o endereço 21 de M já esteja ocupado, e então, toda a tabela precisará ser recriada. Para evitar tal transtorno, manteremos a reserva.

Alocação encadeada Como não há garantia de proximidade, a primeira modificação é que o elemento precisará ter uma informação a mais, que é o endereço do próximo elemento. Com isso, ele ficará:

Nome do País Campeão	Ano do Campeonato	Gols marcados	Próximo elemento

Como não há contigüidade, é necessária a informação de onde se encontra o primeiro elemento da lista. Vamos manter tal informação em M[0]. Entretanto, vamos tirar vantagem da capacidade de crescimento dinâmicos das listas e portanto não reservaremos nenhum espaço para os próximos campeões. Nosso indicador de fim de lista será o número 1 negativo (-1). Apenas para tornar mais claro o exemplo, vamos sombrear os elementos de M que representam apontadores. M ficaria:

M2	0	1	2	3	4	5	6	7	8	9
0	1	Br	62	17	5	In	66	16	9	Br
1	70	20	13	Al	74	22	17	Ar	78	21
2	21	It	82	17	25	Ar	86	19	-1	

©88-08, Pedro Kantek

EXERCÍCIO 30 identifique na Memória M acima:

- Os elementos individuais
- O endereço do terceiro elemento
- Os endereços que indicam quais os anos em que a Argentina foi campeã

Sem querer estudar os algoritmos que manuseiam tais estruturas (eles já já vão chegar), vamos só a título de ilustração, mostrar como M1 (seqüencial) e M2 (encadeada) sofreriam alterações se houvesse uma determinação de reordenar as listas M1 e M2, criando M3 e M4, agora por ordem alfabética de país (note que em M1 e M2, elas estão ordenadas por ano do campeonato). A ordem alfabética de países campeões é: Alemanha, Argentina, Brasil, Inglaterra, Itália.

M3	0	1	2	3	4	5	6	7	8	9
0	Al	74	22	Ar	78	21	Ar	86	19	Br
1	62	17	Br	70	20	In	66	16	It	82
2	17	R	R	R	R	R	R	R	R	R
3										

Comparando M1 com M3 (é a mesma informação), percebe-se que todos os elementos da lista tiveram seus locais modificados.

Já M4 (também com os apontadores sombreados) ficaria

M4	0	1	2	3	4	5	6	7	8	9
0	13	Br	62	17	9	In	66	16	21	Br
1	70	20	5	Al	74	22	17	Ar	78	21
2	25	It	82	17	-1	Ar	86	19	1	

Note-se que dos 29 inteiros que compunham a lista em M2, apenas 7 tiveram que ser modificados. Embora possa parecer difícil que a M4 está em ordem ascendente por nome de país, vamos seguir pelo encadeamento (por isso ela é encadeada) e mostrar que isso de fato é verdade.

A lista começa no endereço gravado em M[0], ou seja em M[13]. O elemento que começa em M[13] é Al, 74, 22. O segundo elemento tem seu endereço em M[16] e é 17. O elemento que começa em M[17] é Ar, 78, 21. O próximo elemento começa em M[25]. Ele é Ar, 86, 19. O próximo elemento começa em M[1] e é Br, 62, 17. O seguinte começa em M[9] e é Br, 70, 20. O próximo começa em M[5] e é In, 66, 16. O próximo começa em M[21] e é It, 82, 17. O -1 que termina este elemento indica que ele é o último da lista.

Apenas para concluir esta primeira apresentação, vamos imaginar por hipótese que o título conquistado pela Argentina em 1978, deixe de ser considerado. (Ou seja esse elemento deve ser retirado de M3 e de M4). Eis como ficariam M5 e M6 com a alteração proposta.

M5	0	1	2	3	4	5	6	7	8	9
0	Al	74	22	Ar	86	19	Br	62	17	Br
1	70	20	In	66	16	It	82	17	R	R
2	R	R	R	R	R	R	R	R	R	R
3										

Comparando M3 com M5, percebe-se a mudança de pelo menos 13 elementos de M.

Já M6 ficaria

M6	0	1	2	3	4	5	6	7	8	9
0	13	Br	62	17	9	In	66	16	21	Br
1	70	20	5	Al	74	22	25			
2		It	82	17	-1	Ar	86	19	1	

 ©88-08, Pedro Kantek

Note que, para criar M6, apenas um elemento de M4 precisou ser modificado (o M[16], que passou de 17 para 25). Os elementos M[17..20] foram apagados apenas para mostrar que essa área está liberada. Mas a rigor, mesmo liberada, ela ficaria com os conteúdos antigos.

É hora de encerrar este exemplo. Ele não quer provar que a alocação encadeada é melhor ou pior que a seqüencial, mas se o leitor gastou alguns minutos entendendo os conteúdos das 6 tabelas acima, deve ser capaz de entender a regra básica para decisão entre alocação seqüencial e encadeada. A seqüencial é fácil de ver e entender, mas não é muito eficiente. A Encadeada é mais eficiente, mas é bem mais "dura"de enxergar. Como toda regra básica, esta admitiria um mundo de senões, exceções, mas...contudo semelhantes. Certamente precisamos discutir melhor essa regra (e o faremos adiante), mas ela fica como sinalizadora de percurso daqui para frente.

Mais um exemplo Ordenando alunos por nome . Seja uma sala de aula com 48 carteiras divididos em 6 fileiras de alunos com 8 alunos por fileira. Nosso objetivo é colocar a primeira e a última filas da sala com os alunos ordenados por ordem alfabética do primeiro nome. A primeira fila será ordenada usando alocação seqüencial, e a última em alocação encadeada.

Na primeira fila (alocação seqüencial), nosso procedimento poderá ser:

1. Localizar o aluno cujo nome tenha a menor inicial
2. Trocar esse aluno de lugar com o aluno sentado na primeira carteira
3. Repetir o teste 1 com os alunos restantes (excluído o primeiro)
4. Trocar o aluno vencedor do teste 3 com o aluno sentado na segunda carteira
5. Repetir o ciclo, até que todos estejam em seus lugares corretos

Na última fila (alocação encadeada), vamos precisar apontadores. Usaremos o quadro negro da sala como endereço inicial da lista, e distribuiremos um pedaço de papel para cada aluno da fila. Nosso procedimento agora será:

1. Numerar as carteiras
2. Localizar o aluno cujo nome tenha a "menor"letra inicial e transcrever o número de sua carteira no quadro negro.
3. Localizar o aluno cujo nome ocupe o segundo lugar. Anotar o número da sua carteira no papel do aluno vencedor do teste 2.
4. Prosseguir no ciclo, até o aluno cujo nome ocupe o último lugar. Nesse, anotar um símbolo especial (terminador) no seu papel.

Novamente, uma visão esquemática das vantagens e desvantagens das duas estruturas na tarefa de classificar dados. Na alocação seqüencial, o fenômeno que salta aos olhos é a movimentação físicas (alunos trocando de lugares). Na encadeada, o principal destaque é o overhead introduzido pelos apontadores.

EXERCÍCIO 31 Imagine que ambas as filas contém os seguintes alunos

[1]João	[2]Maria	[3]Pedro	[4]Fábio	[5]Carlos	[6]Willy	[7]Helena	[8]Antônio

Como ficaria a primeira fila ?

E a última fila ? Use o quadro a seguir como apontador para a lista

©88-08, Pedro Kantek

Use a linha de baixo para registrar o apontador para o próximo. Use -1 para indicar o fim da lista.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Operando com listas

Sejam agora algumas operações básicas com as listas que estamos estudando. A operação fundamental que é a de classificação será deixada para mais tarde, até porque já foi suficientemente explorada nos dois exemplos anteriores.

Onde começa ?

- Seqüencial: no endereço E_i
- Encadeada: no endereço apontado pelo endereço S_i.

No exemplo da carteiras acima, a primeira fileira começa na primeira carteira e a última fileira, na carteira cujo número aparece no quadro.

EXERCÍCIO 32 Quem é o primeiro da primeira fila ?

[1]Antônio	[2]Carlos	[3]Fábio	[4]Helena	[5]João	[6]Maria	[7]Pedro	[8]Willy

Quem é o primeiro da última fila ? [8]

[1]João	[2]Maria	[3]Pedro	[4]Fábio	[5]Carlos	[6]Willy	[7]Helena	[8]Antônio
2	3	6	7	4	-1	1	5

Onde termina ?

- Seqüencial: Se a lista está completamente cheia, ela termina em Si + (Nmax -1) * C. Se não estiver completamente cheia, precisaremos
  - guardar em algum lugar, a quantidade real de elementos ocupados, ou
  - usar o conceito de sentinela (um determinado ítem da lista contém um valor próprio que indica que o seu vizinho anterior é o último).
- Encadeada: A lista encadeada precisa ser percorrida até se encontrar o terminador. O item que o tiver é o último. Entretanto, nada impede que se guarde em algum lugar o endereço do último, tal como na alocação seqüencial

Na primeira fileira, a lista termina na última carteira. Na última fileira, a lista termina no aluno que tiver o papel onde esteja escrito -1.

EXERCÍCIO 33 Quem é o último da primeira fila ?

[1]Antônio	[2]Carlos	[3]Fábio	[4]Helena	[5]João	[6]Maria	[7]Pedro	[8] Willy

Quem é o último da última fila ?

[8]

1 João	2 Maria	3 Pedro	4 Fábio	5 Carlos	6 Willy	7 Helena	8 Antônio
2	3	6	7	4	-1	1	5

Inclusão no começo

Inclusão no meio Em geral esta operação é necessária quando a lista está ordenada por algum critério. O local da inclusão portanto é determinado biunivocamente pelo atributo (no nosso caso, o nome do aluno) do elemento que está sendo inserido.

- Seqüencial: Havendo espaço para a inclusão, deve-se localizar o local da inclusão. Feito isso, todos os elementos seguintes devem ser deslocados para trás. Com isso, abre-se o espaço para o novo elemento.
- Encadeado: Incluí-se o elemento em qualquer parte disponível, e guarda-se esse endereço - por hipótese E_k. Localiza-se o elemento imediatamente anterior ao elemento que será incluído. Pega-se o conteúdo do apontador do elemento anterior e coloca-se no apontador de E_k. Finalmente, muda-se o apontador do elemento anterior para o valor E_k.

Supondo que nossa sala ganhou uma nona carteira lá no final. Um novo aluno deve ser incluído na primeira e na última fileira da sala de aula. Na primeira fileira (seqüencial), localizado o ponto de inserção, todos os alunos que estão depois dele precisão se mover 1 carteira para trás. Isso liberará a carteira certa para o novo aluno. Na última fileira (encadeada), primeiro o novo aluno senta na última carteira que estará disponível, e registra-se em algum lugar o número dessa carteira. Localiza-se o ponto de inserção. O aluno novo recebe no seu papel o número que o aluno anterior ao ponto de inserção tinha. Depois o aluno anterior ao ponto de inserção recebe o número que foi registrado lá no começo.

EXERCÍCIO 34 Como ficará a primeira fila após inserir o aluno "Jair"?

1 Antônio	2 Carlos	3 Fábio	4 Helena	5 João	6 Maria	7 Pedro	8 Willy	
1	2	3	4	5	6	7	8	9

Como ficará a primeira fila após inserir o aluno "Jair"?

1 João	2 Maria	3 Pedro	4 Fábio	5 Carlos	6 Willy	7 Helena	8 Antônio	9 Jair

Inclusão no final Este tipo de inclusão em geral ocorre quando a lista não está ordenada. Utiliza-se a inserção no fim, nestes casos, por ser a operação mais barata em termos de recursos computacionais.

- Seqüencial: Havendo espaço para a inclusão, esta é imediata, pela fórmula já vista
- Encadeada: Inclui-se o novo elemento, que receberá o terminador no seu apontador. Localiza-se o terminador anterior e este recebe o endereço do elemento recém incluído.

Na primeira fileira, o aluno novo sentará na última carteira. Na última fileira, o aluno sentará na carteira livre e anotará no seu papel o terminador, e terá seu número anotado em algum lugar. O aluno que tinha o terminador receberá este número guardado.

EXERCÍCIO 35 Como ficará a primeira fila após inserir o aluno "Wilson"?

1 Antônio	2 Carlos	3 Fábio	4 Helena	5 João	6 Maria	7 Pedro	8 Willy	
aqui1	2	3	4	5	6	7	8	9

Como ficará a primeira fila após inserir o aluno "Wilson"?

1 João	2 Maria	3 Pedro	4 Fábio	5 Carlos	6 Willy	7 Helena	8 Antônio	9 Wilson

Visitação de todos os elementos

- Seqüencial: Os elementos serão percorridos seqüencialmente (consecutivamente) do primeiro ao último.
- Encadeada: Iniciando no endereço do início da lista, os elementos serão percorridos, sendo que o endereço do próximo da lista sempre se encontra no elemento atual. A visitação termina quando o terminador for encontrado.

Na primeira fileira, os alunos serão processados em ordem física , da primeira à última carteira. Na última fileira, o processamento começará na carteira cujo número estiver no quadro, e seguirá a seqüência dada pelos papeis de cada aluno. Terminará quando for encontrado o papel com -1.

EXERCÍCIO 36 Indique por meio de setas, qual a seqüência de visitação da primeira fila

1 Antônio	2 Carlos	3 Fábio	4 Helena	5 João	6 Maria	7 Pedro	8 Willy

Indique por meio de setas, qual a seqüência de visitação da última fila

8							
1 João	2 Maria	3 Pedro	4 Fábio	5 Carlos	6 Willy	7 Helena	8 Antônio
2	3	6	7	4	-1	1	5

Exclusão

- Seqüencial: Encontrado o elemento a excluir, todos os subseqüentes deverão ser trazidos para a frente.
- Encadeada: Encontrado o elemento a excluir, o conteúdo do seu apontador deverá ser colocado no apontador do elemento anterior a ele.

Na primeira fileira, saindo um aluno, todos os que ficaram atrás dele, devem passar uma carteira para a frente. Na última fileira, o dono do papel que aponta para quem vai sair deve receber o papel do aluno que está saindo.

EXERCÍCIO 37 Como ficará a primeira fila após excluir o aluno "Carlos"?

1 Antônio	2 Carlos	3 Fábio	4 Helena	5 João	6 Maria	7 Pedro	8 Willy

1	2	3	4	5	6	7	8

Intercalação

- Seqüencial: Uma nova fila deve ser alocada, com tamanho suficiente para conter as duas filas a serem intercaladas. Estas devem estar em alguma ordem, que determine o critério de intercalação .
- Encadeada: Os apontadores devem ser reorganizados para que a nova lista represente a intercalação. Não há necessidade de movimentação física dos dados.

 ©88-08, Pedro Kantek

Dadas a fileira A, seqüencial, com os alunos já em ordem alfabética e a fileira B também, a intercalação de ambas, dará origem a uma nova fileira, na qual todos estarão em ordem. Obviamente todos os alunos deverão trocar de carteira. Já em duas fileiras encadeadas, em ordem, os papéis deveriam ser trocados entre si para refletir a nova lista. Nenhum aluno precisa sair de sua carteira.

EXERCÍCIO 38 Suponhamos que a fileira A (seqüencial) de alunos seja

1 Alberto	2 Carlos	3 Dante	4 Geraldo	5 Ivan

1 Benedito	2 Eduardo	3 Filomena	4 Homero

Como deverá ficar a fileira (seqüencial) intercalada ?

1	2	3	4	5	6	7	8	9

Agora suponhamos as fileiras A e B (encadeadas) de alunos, compartilhando a mesma memória

A

4	

B

6	

1 Carlos	2 Ivan	3 Geraldo	4 Alberto	5 Dante	6 Benedito	7 Homero	8 Filomena	9 Eduardo
5	-1	2	1	3	9	-1	7	8

Como ficaria a fila concatenada ?

1 Carlos	2 Ivan	3 Geraldo	4 Alberto	5 Dante	6 Benedito	7 Homero	8 Filomena	9 Eduardo

Desafio

- Escreva um programa de computador que implemente o teatro estudado na folha ALLA212a. Preencha o teatro com outras configurações e verifique as respostas que o programa apresentará. Compare-as com as encontradas manualmente.

4.3 O jogo de truco

4.3.1 Conceitos

Partida Jogo valendo 12 pontos, conseguidos através das "mãos".

Mão Fração da partida, vale 1 ponto e poderá ter seu valor aumentado a 3, 6, 9 e até 12 pontos, é disputada em melhor de 3 rodadas.

Rodada É a fração da "mão", em cada rodada, os jogadores mostram uma carta.

Truco Para pedir o aumento do valor da "mão"para 3.

Seis ou meio-pau Para pedir o aumento do valor da "mão"para 6.

Nove Para pedir o aumento do valor da "mão"para 9.

©88-08, Pedro Kantek

Doze ou queda Para pedir o aumento do valor da ”mão”para 12.

Cangar, melar, embuchar ou empatar Quando a maior carta de cada dupla (no jogo de duplas), numa determinada rodada, tem o mesmo valor. Valedo também para a maior carta de cada jogador (no jogo mano-a-mano).

Mão de onze Quando uma das equipes conseguir chegar a 11 pontos na partida.

Mão de ferro É a mão-de-onze especial, quando todos conseguem chegar a 11 pontos na partida.

Tento Quando algum jogador vê um de seus oponentes roubando ele pede tento, a fim de requerer pontos na partida.

Manilhas São as 4 maiores cartas do jogo, as únicas dentre as 42 que não podem ser cangadas.

Zap, gato ou zorro É a maior carta do jogo e seu nipe sempre será paus.

Copas, copeta ou copilha É a segunda maior carta e seu nipe sempre será copas.

Espadas ou espadilha É a terceira maior e seu nipe sempre será espadas.

Ouros, pica-fumo ou mole É a quarta maior e seu nipe sempre será ouros.

Vira é a carta que definirá as 4 manilhas.

Esconder a carta Colocar a carta que seria mostrada a todos, no meio do baralho, assim ninguém poderá saber qual era. É como se jogasse de ”lona”.

Lei do truco A equipe que perde a partida por 12 X 0 é obrigada a passar por debaixo da mesa.

Hierarquia das Cartas

Em ordem decrescente, começa pelas 4 manilhas: Zap, Copeta, Espadilha e Pica-fumo. As demais cartas em ordem decrescente são: 3, 2, Ás, K, J, Q, 7, 6, 5 e 4. As manilhas são definidas como sendo uma carta acima de acordo com a ”vira”. Se a ”vira”for 4, as manilhas são: 5 de paus (zap), 5 de copas (copeta), 5 de espadas (espadilha) e 5 de ouros (pica-fumo). Se a vira for 3, as manilhas são os 4 de todos os nipes. Se for o 7, será a Q. Se for o três-e-meio, a manilha será o 4. Se a ”vira”for o J de copas, por exemplo, as manilhas serão o K de paus (zap), K de copas (copeta), K de espadas (espadilha), e K de ouros (pica-fumo). E na sequência, as cartas em ordem decrescente são: os 3 de todos os nipes, os 2 de todos os nipes, os Ás de todos os nipes, os J de paus, espadas e ouros em igual valor, as Q de todos os nipes, os 7, 6, 5 e 4 de todos os nipes. O nipe da ”vira”, não altera no valor de nenhuma carta, bem como os nipes das cartas comuns. No exemplo acima, todos os 3 têm o mesmo valor, bem como os J de paus, espadas e ouros.

A Postura dos Jogadores

No jogo de duplas, os jogadores devem sentar em posições alternadas. Do lado direito e do lado esquerdo deverão estar dois oponentes e à frente, seu parceiro. O jogador postado do lado esquerdo do embaralhador, ”cortará”o baralho e o situado à direita irá começar o jogo.

 ©88-08, Pedro Kantek

Embaralhar

O embaralhador, poderá ver o fundo do baralho no ato de embaralhar, mas não poderá mostrá-lo ao seu companheiro. Não poderá ”abrir”o baralho como um ”leque havaiano”e ficar ordenando as cartas. Após embaralhar, deverá passar o maço ao seu oponente da esquerda que irá ”cortar”o baralho e tirar a ”vira”. Quando receber de volta o maço do oponente da esquerda, deverá começar a distribuição das cartas de cima para baixo, isto se o adversário que ”cortou”, não pedir para ”subir”. Se mandar ”subir”, deverá fazer a distribuição pegando as cartas de baixo para cima.

”Cortar”o Baralho

O jogador que ”corta”o baralho também tira a ”vira”, esta pode ser escolhida antes de devolver o baralho ao embaralhador que irá dar as cartas, mas o jogo só começa quando a ”vira”for conhecida. O ato de ”cortar”, é permitido apenas 3 vezes, não é permitido em hipótese alguma ”cerrar”o baralho. Antes de passar o baralho ao adversário que irá iniciar a distribuição das cartas, é permido olhar a última carta do maço, não sendo permitido mostrar ao parceiro. Coso a última carta seja vista, o jogador que ”cortou”o baralho, não poderá mandar o embaralhador ”subir”as cartas quando estiver distribuindo-as. Feito esse ritual, o baralho deve ser entregue ao embaralhador.

”Queimar”as Cartas

É permitido ”queimar”até 9 cartas. As cartas ”queimadas”, deverão ser mostradas a todos. Só quem poderá queimar é quem ”corta”o baralho ou o embaralhador. As cartas poderão ser ”queimadas”no momento dos ”cortes”ou quando o embaralhador estiver distribuindo-as. No momento dos ”cortes”, o jogador poderá queimar de 1 a 9 cartas, não podendo vê-las antes de ”queimá-las”. Essas cartas poderão ser escolhidas de qualquer parte do baralho, mas não poderá mudar a ordem das cartas, pois se mudar, contará como ”cortes”e poderá ultrapassar o limite de 3 ”cortes”. Se já foram queimadas 9 cartas, o embaralhador não poderá ”queimar”mais nenhuma. Se o jogador que ”corta”o baralho ”queimar”4 cartas, o embaralhador poderá ”queimar”até no máximo 5 cartas.

Dar as Cartas

A primeira carta do maço deverá ser entregue para o oponente à direita. As demais cartas poderão ser entregues na ordem que bem enter o embaralhador até atingir o número de 3 cartas para cada jogador. Durante a distribuição das cartas o embaralhador poderá ”queimá-las”desde que não ultrapasse o número permitido de 9 cartas. Se o embaralhador estiver dando as cartas e uma das cartas de um de seus dois adversários virar, o jogador prejudicado poderá analizar a carta que virou e continuar com ela, ou, poderá ”queimá-la”e pedir uma nova carta. O embaralhador por isso, não levará qualquer punição. Se virar uma carta do embaralhador ou de seu parceiro, não será permitido trocar a carta.

Erro no Número de Cartas

Se forem dadas um número maior de cartas do que o permitido, o jogador que as tiver deverá postar todas as cartas frente a um de seus adversários, que irá ”queimá-las”até restarem 3 cartas. Se for dado um número inferior, o jogador prejudicado deverá informar ao embaralhador que irá completar seu jogo até atingir o número máximo de 3 cartas.

OBS.: Um erro no número de cartas, não significará a perda ou ganho de nenhum ponto.

©88-08, Pedro Kantek

Começando o Jogo

A primeira rodada da "mão", começa com o jogador postado à direita do embaralhador, este deverá mostrar uma de suas 3 cartas. O jogador à direita deste deverá mostrar uma carta, seguindo em sentido anti-horário. Na primeira rodada, não é permitido esconder a carta. Leva a rodada, a dupla que mostrar a maior carta. Leva os pontos da "mão", a dupla que ganhar duas rodadas. Começará a segunda rodada, o jogador que mostrou a maior carta na primeira rodada, jogará em seguida o jogador da direita, depois o jogador da direita deste e por fim, o jogador que ainda não mostrou sua segunda carta. Começará a terceira rodada, caso seja necessária, o jogador que mostrou a maior carta na segunda rodada. Na 2^a e 3^a rodadas, é permitido esconder as cartas.

Cangar o Jogo

O jogo poderá ser cangado em 5 situações:

1. Na 1^a rodada: Se o jogo for cangado na primeira rodada, o desempate será feito na 2^a rodada. Deverá mostrar a carta, primeiramete, o jogador que cangou o jogo na 1^a rodada, não sendo obrigado a mostrar sua maior carta. O jogador da direita será o próximo e o da direita deste o próximo e assim sucessivamente. Com o jogo cangado, é permitido esconder a carta. A dupla que mostrar a maior carta, levará os pontos da "mão", não sendo necessário, jogar a 3^a rodada.
2. Na 2^a rodada: Se o jogo for cangado na 2^a rodada, levará os pontos da "mão"a dupla que ganhou a 1^a rodada, não sendo necessário jogar a 3^a rodada.
3. Na 3^a rodada: Se o jogo for cangado na 3^a rodada, levará os pontos da "mão"a dupla que venceu a 1^{a} rodada.
4. Na 1^a e 2^a rodadas: Se o jogo for cangado em ambas rodadas, levará os pontos da "mão"a dupla que vencer a 3^a rodada.
5. Nas 3 rodadas: A mão termina empatada.

Truco, Meio-pau, Nove e Doze

Pode ser pedido TRUCO a qualquer momento do jogo. A "mão"só estará valendo 3 pontos, se um dos jogadores da dupla adversária aceitar o pedido. O pedido poderá ser aceito dizendo-se: "cai", "venha", "manda", "aceito", ou qualquer outra expressão que dê uma posição de positivismo. O pedido de TRUCO, deverá ser feito de forma bem clara, não podendo ser usadas palavras parecidas, como "troco", "turco", etc, a fim de confundir o oponente. Caso a dupla adversária aceite o TRUCO, a equipe que conquistar a "mão", somará 3 pontos. Se o TRUCO não for aceito, a equipe que fez o pedido somará 1 ponto. Se além de aceitar o TRUCO, a dupla pedir MEIO-PAU, a "mão"só valerá 6 pontos caso a dupla adversária aceite o pedido. Se a dupla "correr", a equipe que pediu MEIO-PAU leva 3 pontos. Se além de aceitar o MEIO-PAU, a dupla pedir NOVE, a "mão"só valerá 9 pontos, caso a dupla adversária aceite o pedido, podendo ainda levar o jogo a DOZE. Se uma dupla corre do NOVE, a dupla que o pediu leva 6 pontos e, se uma dupla corre do DOZE, a dupla que o pediu leva 9 pontos.

Mão-de-onze

A mão-de-onze acontece quando uma das duplas atinge 11 pontos na partida. A dupla que chegou a este patamar tem a regalia de mostrar suas cartas ao seu parceiro. Na mão-de-onze, o jogo já começa trucado ("mão"valendo 3 pontos). Caso a dupla ao analizar

 ©88-08, Pedro Kantek

as cartas que tem em mãos, achar que não será possível vencer a "mão", é permitido "correr"do jogo, dando apenas 1 ponto ao adversário. Se a dupla aceitar jogar a mão-de-onze, esta valerá 3 pontos. Na mão-de-onze não é permitido olhar o fundo do maço, queimar cartas e pedir truco, meio-pau, nove ou doze.

Mão-de-ferro

A mão-de-ferro acontece quando ambas as duplas estão com 11 pontos na partida, sendo assim, quem vencer a mão-de-ferro leva. Na mão-de-ferro não é permitido olhar o fundo do maço, queimar cartas e pedir truco, meio-pau, nove ou doze. Além disso, a "vira"não poderá ser vista, ela deverá ficar separada e com sua face virada para baixo, bem como as 3 cartas de cada jogador. Ninguém poderá ver suas cartas para jogar. O 1^{o} a jogar, mostra uma de suas 3 cartas, depois o 2^{a} mostra, depois o 3^{a} e o 4^{a}. Estando as 4 cartas na mesa, é hora de mostrar a "vira". Mesmo as cartas que estão na mesa, poderão ser manilhas de acordo com a "vira". Depois de ver quem levou a 1^{a} rodada, é hora de jogar a 2^{a}. Ainda serão jogadas no "escuro"a 2^{a} e 3^{a} rodadas, porém a "vira"já estará conhecida.

Tento

O pedido de tento poderá ser feito a qualquer momento do jogo, desde que seja visto e provado que alguém estava roubando. No truco é permitido roubar, mas se for pego pagará um tento pela incompetência de não saber roubar. Ainda será tento se o jogador queimar mais de 9 cartas, cortar o baralho em número de vezes superior ao permitido, jogar com 4 cartas, pedir truco, meio-pau, nove ou doze na mão-de-onze ou mão-de-ferro, mostrar a "vira"ou olhar as cartas antes da hora na mão-de-ferro, se ao invés de falar truco, falar uma palavra parecida para confundir.

O Valor do Tento

O valor do tento depende da situação em que se encontra o jogo. Se a "mão"estiver valendo 1 ponto e for pedido tento, o tento valerá 1 ponto. Se tiver valendo 3 pontos, o tento valerá 3 pontos e assim sucessivamente. Na mão-de-onze, o tento vale 1 ponto se a dupla que estiver com 11 pontos ainda não tiver dito se vai jogar ou "correr". Se o jogo já estiver rolando e for pedido tento, este valerá 3 pontos. Na mão-de-ferro, o tento significa o fim da partida e consequentemente a derrota para a equipe infratora.

4.3.2 Algoritmo

Suponha uma instância da memória conforme descrita no exercício ALLA331a. Supondo que a memória reside no vetor M (de 100 inteiros, com índices variando entre 0 e 99), e que se pretende construir um algoritmo que, lendo M, estabeleça o valor correto para uma matriz T de 3 linhas por 5 colunas, contendo as cartas em jogo, eis uma possibilidade.

Antes, vai-se desenhar – para o completo entendimento – como ficaria a matriz T.

Jogador 1	Jogador 2	Jogador 3	Jogador 4	Resto do baralho
carta 1	carta 1	carta 1	carta 1	carta 1
carta 2	carta 2	carta 2	carta 2	carta 2
carta 3	carta 3	carta 3	carta 3	carta 3

```
1: função MONTAT
2: inteiro T [3] [5]
3: inteiro I, P, J
4: I ← 0
5: enquanto I < 5 faça
```

©88-08, Pedro Kantek

```
6:   P ← M[M[0]+I]
7:   J ← 0
8:   enquanto J < 3 faça
9:     T[J][I] ← M[P]
10:    P ← M[P+1]
11:    J++
12:  fimenquanto
13:  se M[P+1] ≠ -1 ∧ I ≠ 4 então
14:    ... erro: este baralho está errado...
15:  fimse
16:  I++
17: fimenquanto
```

Desafio

- Escreva um programa de computador que simule o funcionamento de uma FAT.
- Use um editor em hexadecimal (HEXEDIT, por exemplo) e altere áreas da FAT e do diretório de um DISQUETE, analisando depois através do uso de diversos programas utilitários, o resultado.

Tome todo o cuidado do mundo (e mais um pouco) antes de alterar áreas de controle de discos, pois o risco de perda total dos dados não é irrelevante. Trabalhe apenas com disquetes

4.4 Alocação de área em disco sob DOS

O algoritmo de alocação de área em disco sob DOS precisa conciliar algumas questões importantes e em certa medida contraditórias, a saber:

1. alocação encadeada, para permitir que arquivos não precisem estar em contigüidade física. Esta característica permite que um arquivo comece hoje, seja fechado e dentro de alguns dias, seja aberto para extensão, podendo crescer (ou diminuir) sem que haja necessidade de transferência física dos dados já gravados. Esta característica permite também um certo grau de (pseudo) paralelismo, através do qual dois ou mais processos podem estar gravando seus arquivos.
2. Eficiência de busca, tanto para buscar um dado quanto para alocar um bloco novo a um arquivo já existente, sem que o sistema perca muito tempo nessas operações. Isto implica em overhead mínimo possível.
3. Facilidade de acessar e alocar blocos, incluindo-se aqui a simplicidade necessária e suficiente para recuperar discos danificados e/ou arquivos excluídos por engano.

Surge agora o conceito de unidade de alocação : Trata-se do quantum de área em disco que é oferecido a qualquer processo sempre que há requisição de espaço. Este é o valor mínimo de alocação. Nunca um arquivo terá a si alocada uma área menor do que esta unidade (embora possa acabar não usando integralmente o que foi alocado). Os americanos chamam esta unidade de alocação de cluster , que foi traduzida para o português como grânulo .

Tipicamente um cluster pode ser de 2048 bytes ou de 1024 bytes. Clusters muito grandes causam fragmentação do disco (pelo desperdício de áreas não ocupadas em cada cluster) enquanto clusters pequenos implicam em grandes tabelas de alocação. Isto posto, e como regra geral, quanto menor o cluster mais eficiente é o sistema.

Áreas em um disco DOS Como já descreveu magistralmente Peter Norton, em seus livros, qualquer disco em DOS está organizado em 4 áreas (eventualmente discos rígidos tem uma quinta área que descreve o volume).

1. Registro de Boot: contém características físicas do disco e um pequeno programa de carga do núcleo do sistema operacional (SO) quando a máquina é ligada e este disco é o especificado para inicializar a máquina.
2. Área de diretório , que comporta diversas informações sobre cada um dos arquivos (nome, tamanho, data de atualização, atributos etc). Aqui interessam-nos duas informações: o nome do arquivo e o cluster em que ele se inicia.
3. Área de FAT (File alocation table). Trata-se de uma tabela que será explicada após ver-se a área 4.
4. Área de dados do disco. É a maior (mais de 95

Agora pode-se voltar à definição da FAT. Ela faz a ligação entre os diversos clusters que compõe um arquivo.

Assim por exemplo, suponhamos que o arquivo AAAA ocupa 4 clusters, o 2, o 7, o 8 e o 10. No diretório, ocorre a entrada: AAAA começa em 2. Na segunda posição da FAT (que equivale ao cluster 2) o sistema coloca o número do próximo cluster do arquivo, no caso 7. Na 7 entrada da FAT está o número do próximo cluster que é o 8. Na oitava entrada consta 10 e finalmente na décima entrada tem-se um terminador de maneira a avisar que a cadeia acaba lá.

Vale ressaltar que o primeiro cluster do arquivo está identificado no diretório. A entrada da FAT correspondente a este cluster inicial contém o endereço do próximo cluster e assim por diante.

Exemplo completo: Seja um disco assim representado

N	T	CI

1	2	3	4	5	6	7	8	9	10	11	12	13

Observações 1. Quando um disco é formatado , é feita a gravação de uma constante em todo o disco. Logo a seguir todo o disco é lido e onde não houver a constante gravada, é porque a superfície de gravação apresenta erros. Quaisquer gravações nesse lugar estarão condenadas a serem impossíveis de recuperar (pois a superfície contém imperfeições magnéticas). A maneira do DOS contornar isso é colocando um valor qualquer na FAT que sinalize estar esse cluster em questão impossibilitado de ser alocado.

2. Quando um cluster é alocado, e não integralmente usado, a área final fica desperdiçada. Entretanto, se no futuro o arquivo crescer de tamanho, antes de haver nova alocação, a área anteriormente desperdiçada é usada até o seu esgotamento completo.

Erros em discos DOS Para esta discussão, suponhamos um disco simplificado composto apenas de área de diretório e FAT. Para a FAT valem as seguintes convenções: 0 = cluster livre; -1 = é o último cluster do arquivo; -2 = cluster defeituosos e que não deve ser alocado para ninguém. K = o arquivo prossegue no k-ésimo bloco.

1. Um cluster é possuído por 2 ou mais arquivos Suponha que a FAT para o exemplo acima seja: 2, 9, 1, 7, -1, 0, 2, 0, -1 e perceba que o cluster 2 pertence tanto ao arquivo A quanto ao B, situação obviamente impossível e que resultou de erro.

2. Um cluster que não está livre e não pertence a nenhum arquivo Suponha que a FAT agora seja: 5, 9, 1, 7, -1, 6, 2 , 0, -1. Repare que o cluster número 6 não pertence nem a A nem a B e no entanto não tem 0 (vazio) na FAT

3. Uma cadeia que não tem terminador Agora a FAT é 5, 9, 1, 7, 0, 0, 2, 0, -1. Veja que o arquivo A não termina

4. Existência de laço (loop) na alocação Suponha A começando em 1 e B começando em 2. Se a FAT for 2, 1, -1, -1, ... temos o loop.

EXERCÍCIO 39 Suponha a seguinte estrutura de uma FAT (formato DOS)

```
1: estrutura ARQUIVO
2: cadeia NOME[11]
3: inteiro TAMANHO
4: inteiro CLINIC
5: fim estrutura
6: estrutura FAT
7: ARQUIVO ARQ[30]
8: inteiro ALOC[300]
9: fim estrutura
10: FAT FAT1
```

Escreva uma função que não receba nada, acesse FAT1 (por hipótese é uma variável global) calcule e devolva a quantidade de clusters que compõe o primeiro arquivo de FAT1. Por hipótese este primeiro arquivo é válido.

EXERCÍCIO 40 Escreva uma função que receba o nome de um arquivo, acesse FAT1 (por hipótese é uma variável global) calcule e devolva a quantidade de clusters que compõe este arquivo em FAT1. Se este arquivo não existir em FAT1 deve devolver 0.

EXERCÍCIO 41 Escreva o algoritmo que acesse FAT1 (por hipótese é uma variável global) calcule o tamanho de todos os arquivos cujo nome seja diferente de espaços e imprima o nome daqueles cuja medida do tamanho seja menor do que o tamanho registrado na FAT. Assuma que cada cluster tem 1024 caracteres.

EXERCÍCIO 42 Suponha uma variável global FAT1, como descrita acima. Algum gaiato entrou em FAT1 e apagou os tamanhos de arquivos. Escreva o algoritmo de uma função que processe FAT1 e devolva o nome do maior arquivo existente lá dentro.

Desafio

- Escreva um programa de computador que simule o funcionamento de uma FAT.
- Use um editor em hexadecimal (HEXEDIT, por exemplo) e altere áreas da FAT e do diretório de um DISQUETE, analisando depois através do uso de diversos programas utilitários, o resultado.

Tome todo o cuidado do mundo (e mais um pouco) antes de alterar áreas de controle de discos, pois o risco de perda total dos dados não é irrelevante. Trabalhe apenas com disquetes

4.5 Alocação em UNIX

Já o sistema operacional UNIX utiliza um esquema completamente distinto para alocações em disco. Similar ao diretório do DOS, há uma área nos discos UNIX onde o sistema coloca a lista de arquivos existentes no disco. Ao lado do nome e dos atributos (quem pode ler, gravar e executar os arquivos) há espaço para a colocação de 13 endereços de blocos de alocação do disco associados a um único arquivo.

 ©88-08, Pedro Kantek

Os 10 primeiros blocos, apontam diretamente para o bloco de dados . Assim se um determinado arquivo contiver apenas até 10 blocos de dados, os endereços desses blocos estarão diretamente colocados na lista de arquivos do disco. (Em contraposição ao DOS que guarda apenas o endereço do *primeiro* bloco de dados). Para arquivos pequenos, este esquema favorece o acesso e a facilidade de endereçamento dos dados.

Quando o arquivo se expandir além dos 10 blocos, passa-se a usar o endereço número 11. Este não aponta para dados e sim aponta para um bloco que é subdividido em apontadores para blocos de dados. Este bloco tem portanto um nível de indireção quando aponta para os dados.

Supondo por exemplo um sistema cujo bloco tenha 2000 bytes e cujos números de bloco tenham 4 bytes. Neste sistema o endereçamento direto (10 blocos iniciais) permite usar 20.000 bytes. Se este espaço for insuficiente, alocar-se-á o bloco 11 e este permitirá endereçar mais 500 blocos de dados (2000 (4 = 500). Ao usar este endereço o arquivo pode chegar a 510 blocos (ou 1.020.000 bytes).

Se o arquivo continuar crescendo, alocar-se-á o endereço 12, que terá dois níveis de indireção. Portanto, o endereço 12 aponta para um bloco de endereços que apontam para blocos de endereços que apontam para blocos de dados.

Usando o mesmo exemplo anterior, ao usar o endereço 12, o sistema passa a poder endereçar 10 blocos (dos endereços diretos) + 500 blocos (do endereço 11) + 500 (500 = 250.000 blocos (do endereço 12), totalizando 250.510 blocos ou 250.510 x 2000 bytes.

Finalmente, se o arquivo se tornar imenso, alocar-se-á o endereço 13, que terá 3 níveis de indireção. Os endereços 1 a 10 continuam sendo usados de maneira direta, o 11 com 1 e o 12 com 2 níveis de indireção e finalmente o 13 com 3 níveis de indireção.

Ainda o mesmo exemplo anterior. Usando o endereço 13, o sistema passa a poder endereçar 10 blocos (dos endereços diretos) + 500 blocos (do endereço 11) + 500 (500 blocos (do endereço 12), + 500 (500 (500 (do endereço 13) totalizando 125.250.510 blocos ou 250.000.000.000 bytes.

Note-se que é uma estrutura encadeada, o que não obriga (nem isso seria possível) a contigüidade física dos dados.

EXERCÍCIO 43 Suponha uma estrutura de alocação tipo UNIX com diretório contendo D_1, D_2, I_1, I_2 e I_3, com blocos de 100 caracteres e endereços de 10 caracteres. Escreva uma função que receba um tamanho de arquivo qualquer e devolva 1, 2, 3, 4, 5 ou 6 dependendo da regra:

- 1 = o diretório do arquivo apenas usará D1
- 2 = usará D1 e D2.
- 3 = D1, D2 e I1
- 4 = D1, D2, I1 e I2
- 5 = D1, D2, I1, I2 e I3
- 6 = o arquivo não poderá ser guardado nesta estrutura, ele terá um tamanho maior do que o máximo permitido. A propósito quanto é este tamanho máximo ?

EXERCÍCIO 44 Imagine uma estrutura de alocação UNIX-like. Escreva o algoritmo de uma função que receba 4 numeros (quantidade de blocos diretos, quantidade de blocos indiretos, quantidade de endereços por bloco e tamanho do bloco em caracteres) e devolva o tamanho máximo de arquivo que se pode armazenar nesta estrutura.

EXERCÍCIO 45 Suponha a seguinte descrição de um diretório UNIX

©88-08, Pedro Kantek

```
1: estrutura FUNIX
2: cadeia NOME[60]
3: inteiro TAM
4: inteiro ENDS[13] // 10 primeiros: diretos; depois I1, I2 e I3.
5: fim estrutura
6: estrutura BLOCO
7: cadeia BL[1024]
8: fim estrutura
9: FUNIX AC[100] // até 100 arquivos
10: BLOCO X[2000] // até 2000 blocos
```

Suponha FUNIX e X globais e que os endereços tem 8 caracteres. Escreva uma função que receba 3 numeros. O primeiro representa um arquivo (em AC) e o segundo um determinado caracter inicial deste arquivo e o terceiro um tamanho qualquer. A função deve imprimir o conteudo do arquivo a partir do caracter dado e pelo tamanho especificado. Por exemplo:

se for lido (3,10000,1000) a função deve imprimir 1000 caracteres do 3o arquivo. a partir de sua posição 10.000. se for lido (1,1,1) será impresso o primeiro caracter do primeiro arquivo.

Desafio

- JOGO DE BARALHO

  1. Criar uma memória M contendo 120 inteiros, e apresentá-la o tempo todo do programa no vídeo.
  2. Criar comandos para iniciar o jogo, jogar 1 vez, jogar n vezes (onde n é fornecido no comando, e n <= 1000) e abortar a execução.
  3. Em cada instante, avisar quantas cartas tem o monte de cada jogador.
  4. Caso o jogo termine, avisar quem ganhou
  5. Embora muito difícil, pode haver empate (prever isso)
  6. O jogo pode entrar em deadlock. Se isso ocorrer, o jogo deve ser abortado.

  Regras do jogo:

  - O jogo é jogado com um baralho convencional de 52 cartas, sem coringas
  - O baralho deve ser embaralhado e dividido igualmente pelos 2 jogadores.
  - Cada jogador levanta uma carta do seu monte. Aquele que tiver a maior carta, leva as cartas viradas para o fim do seu monte.
  - Se as duas cartas forem iguais, cada um deve virar mais uma (que não vale) e mais uma (que vale). A maior leva tudo
  - A ordem das cartas é A, K, J, Q, 10, 9, 8, 7, 6, 5, 4, 3 e 2.
  - O vencedor é aquele jogador que terminar seu monte primeiro

  Implementação:

  1. Usar filas encadeadas para implementar o jogo.
  2. É proibido fazer movimentações de dados, após a distribuição inicial. Apenas apontadores podem ser modificados durante o transcorrer do jogo.
  3. São necessárias (no mínimo) as seguintes listas:
     (a) Fila do jogador 1
     (b) Fila do jogador 2

     (c) Fila da mesa

  Sugestões

  1. Algoritmos para embaralhar : a) Cria-se um vetor com as 52 cartas. Gera-se um randômico entre 1 e 52. A carta selecionada é retirada deste vetor e colocada em outro chamado vetor_embaralhado. O vetor original sofre um deslocamento, para preencher o espaço deixado pelo elemento que saiu. Agora deve-se gerar um aleatório entre 1 e 51, ... e assim por diante

     b. Gera-se o baralho. Gera-se um vetor de 1000 inteiros aleatórios entre 1 e 52. Pego a carta indicada pelo 1^o elemento do vetor aleatório e troco ela de lugar com a carta indicada pelo 1000^o elemento. Refaço para o 2^o e o 999^o. E assim por diante.

4.6 LISP

[1]

> Se você quer ir a París, precisa aprender francês. Se quer ir para a Inteligência Artificial, precisa aprender LISP.

4.6.1 Programação Funcional

Relembrando os paradigmas usuais de programação são:

Imperativo O computador é uma máquina de estados que são alterados através de comandos emitidos pelo programador. É o principal paradigma, por estar fortemente associado à arquitetura de Von Neumann . Seus representantes: FORTRAN, COBOL, BASIC, PL/I, PASCAL, C, CLIPPER.

Orientado a objetos Coleção de objetos cujos estados se alteram pela execução de ações associadas a eles. Representantes (híbridas: C++, Object Pascal; puras: Smaltalk, Java)

Lógico Um programa é uma relação. O resultado é obtido computando-se os valores que satisfazem a uma relação. O representante solitário é o Prolog.

Funcional A computação é obtida pela aplicação de funções (no sentido matemático) a seus argumentos. Exemplos: APL,LISP, Haskell.

Comparando o paradigma funcional com o imperativo, que é o mais conhecido, pode-se afirmar:

- As linguagens funcionais estão baseadas fortemente em funções. Programas são funções. Funções feitas pelo programador depois de prontas são indistinguíveis das funções primitivas.
- O paradigma é a função matemática, com o conceito de uma correspondência biunívoca entre os conjuntos imagem e domínio. (em $y = f(x)$, x é o domínio e y a imagem).
- Minimiza o conceito de "variável" global, substituindo-o por variável local, logo apresentando *transparência referencial*. Isso diminui de maneira quase inacreditável os erros de programa.

[1] Definição alternativa: LISP = Lot of Idiots and Stupids Parenthesis

- minimiza o uso de atribuições. Na programação funcional original não há a atribuição, mas claro que isso é impraticável.
- Não possuem mecanismos de iteração, substituindo-os por recursão ou por mecanismos específicos à linguagem.
- ciclo leia-calcule-imprima. Em geral o computador se comporta como uma (super) calculadora de mão. Isso aumenta a produtividade ao programar, pois a impressão de resultados já está pronta.
- funções de ordem superior: estas são funções que admitem outras funções (primitivas ou do usuário) como parâmetros da função.
- fracamente tipada (LISP só tem átomos e listas, APL redefine tipos ao bel prazer do programador). Em LISP valores (e não variáveis) é que tem tipo. Objetos têm mais de um tipo. Assim, por exemplo o número `140` é do tipo: `fixnum`, `integer`, `rational`, `real`, `number`, `atom`, e `t`.
- conjunto pequeno (mas poderoso) de primitivas
- capacidade de processamento paralelo
- programas e dados são representados igual e armazenados igual também.
- programas são menores e apresentam menos erros. Esta última não é uma característica formal da programação funcional mas é uma conseqüência real do seu uso. A programação funcional permite a *depuração iterativa*. Funções podem ser testadas imediatamente após prontas. Se elas retornam o valor esperado, pode-se confiar nelas. Essa confiança crescente, faz uma enorme diferença ao final.

Os programas trabalham devolvendo valores e não modificando coisas. As funções são chamadas pelos valores que elas retornam, não por seus efeitos colaterais. A função `remove`, por exemplo, toma uma lista e um valor e devolve outra lista na qual o valor foi excluido.

```
> (setf x '(c u r i t i b a))
(C U R I T I B A)
> (remove 'i x)
(C U R T B A)
```

Não se pode dizer que `remove` remove os elementos da lista, pois a lista original permanece intocada.

```
> x
(C U R I T I B A)
```

E, se realmente se quiser eliminar os elementos da lista ? Seria necessário algo como

```
> (setf x (remove 'i x))
```

A programação funcional tenta evitar assinalamentos, como se verá a seguir.
Compare-se uma função LISP com a sua correspondente C

```
; LISP
(defun sum (n)
  (let ((s 0))
  (dotimes (i n s)
    (incf s i))))
```

 ©88-08, Pedro Kantek


```
/* C */
int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s = s + i;
  return(s);
}
```

Seja agora um exemplo de uma função lisp:

```
> (defun li (L1 L2)   ; pergunta se duas
                      ; listas são iguais
    (cond
       ((null L1) (null L2))
       ((null L2) nil)
       ((not (eql (car L1) (car L2))) nil)
       (t (li (cdr L1) (cdr L2)))))
LI
(defun h(x) ; esta funcao e
            ; a estimadora do tarkin
   (setq qtd 0) ; quantidade de concordancias
   (setq obje '(1 2 3 8 9 4 7 6 5)) ; objetivo
         ; final do tabuleiro (9 e branco)
   (mapcar '(lambda (y z)
      (cond ((eql y z) (setq qtd
             (+ qtd 1)))) ; compara
             ; e se igual qtd++
      ) x obje)
   (- 9 qtd)) ; o retorno
   ;        e' 9 - qtdade de concordancias
```

4.6.2 História do LISP

O LISP nasce como filho da comunidade de Inteligência Artificial. Na origem, 4 grupos de pessoas se inseriram na comunidade de IA: os lingüistas (na busca de um tradutor universal), os psicólogos (na tentativa de entender e estudar processos cerebrais humanos), matemáticos (a mecanização de aspectos da matemática, por exemplo a demonstração de teoremas) e os informáticos (que buscavam expandir os limites da ciência). O marco inicial é o encontro no Darthmout College em 1956, que trouxe duas conseqüências importantes: a atribuição do nome (IA) e a possibilidade dos diferentes pesquisadores se conhecerem e terem suas pesquisas beneficiadas por uma interfecundação intelectual. A primeira constatação foi a de que linguagens existentes privilegiavam dados numéricos na forma de arrays. Assim, a busca foi a criação de ambientes que processassem símbolos na forma de listas. A primeira proposta foi IPL (Information Processing Language I, por Newel e Simon em 1956). Era um assemblão com suporte a listas. A IBM engajou-se no esforço, mas tendo gasto muito no projeto FORTRAN decidiu agregar-lhe capacidade de listas, ao invés de criar nova linguagem. Nasceu a FLPL (Fortran List Processing Language). Em 1958, McCarthy começou a pesquisar requisitos para uma linguagem simbólica. Foram eles: recursão, expressões condicionais, alocação dinâmica de listas, desalocação automática. O FORTRAN não tinha a menor possibilidade de atendê-las e assim McCarthy junto com M. Minsky desenvolveram o LISP. Buscava-se uma função Lisp Universal (como uma máquina de Turing Universal). A primeira versão (chamada

©88-08, Pedro Kantek

LISP pura) era completamente funcional. Mais tarde, LISP foi sofrendo modificações e melhoramentos visando eliminar ineficiências e dificuldades de uso. Acabou por se tornar uma linguagem 100% adequada a qualquer tipo de resolução de problema. Por exemplo, o editor EMACS que é um padrão no mundo UNIX está feito em LISP. Inúmeros dialetos LISP apareceram, cada um queria apresentar a sua versão como sendo a melhor. Por exemplo, FranzLisp, ZetaLisp, LeLisp, MacLisp, Interlisp, Scheme, T, Nil, Xlisp, Autolisp etc. Finalmente, como maneira de facilitar a comunicação entre todos estes (e outros) ambientes, trabalhou-se na criação de um padrão que foi denominado Common Lisp. Como resultado o Common Lisp ficou grande e um tanto quanto heterogêneo, mas ele herda as melhores práticas de cada um de seus antecessores.

4.6.3 Introdução e conceitos iniciais

Lisp é uma linguagem dinâmica (originalmente interpretada) cujos programas são pequenos módulos de funcionalidade genérica e com objetivo restrito. Um programa completo é obtido pela combinação desses módulos.

O conceito básico em LISP é o de FORMA (ou fórmula). Isso porque o LISP é como se fosse (é) uma calculadora, à qual são apresentadas fórmulas e ele as responde. Uma fórmula bem comum é um inteiro, por exemplo 8. Se você entregar a ele a fórmula 8, ele devolve o resultado da fórmula, que é 8. Veja no exemplo

```
> 8    ; um inteiro
8
```

Algumas observações deste trecho de conversa com LISP. O sinal > indica a espera por uma entrada. O que aparece depois do > foi digitado pelo operador. Na linha de baixo a resposta de LISP. O sinal de ";"indica que o que vem depois é um comentário e deve ser desprezado pelo LISP.

Este ciclo de entrar fórmula, o LISP avaliá-la e devolver um resultado é o que caracteriza o trabalho do interpretador. Quando você escrever um programa LISP, o programa nada mais será do que um conjunto de fórmulas que serão processadas uma a uma (como se estivessem sendo fornecidas no teclado) enquanto o programa vai sendo executado. Existe um nome técnico para este comportamento que é o ciclo eval-apply-print .

4.6.4 Funções

Uma parte importante das fórmulas são as funções. As mais simples que conhecemos são as aritméticas (+, -, *, /, ...). Vamos considerar cada função como uma caixa que recebe o(s) seu(s) operando(s) e devolve o resultado esperado. Por exemplo, veja uma função adição usual, na figura 4.1 Neste exemplo, verifica-se que a função + recebe 2

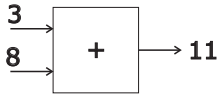


Figura 4.1: Um exemplo da função adição

operandos, no caso o 3 e o 8, e devolve o resultado de 3+8=11. Note que para algumas funções, a ordem de entrada é importante, por exemplo, na figura 4.2.

A função / devolve a parte inteira da divisão dos operandos. Assim, 3/5 é 0, 5/3 é 1, 6/3 é 2 e assim por diante.

 ©88-08, Pedro Kantek

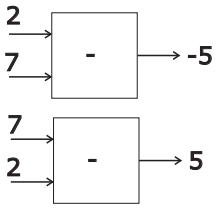


Figura 4.2: A ordem dos operandos é importante

Quando fornecemos ao LISP uma expressão, por exemplo 2 mais 2, estamos pedindo a ele que a avalie esta fórmula. Ele faz isso chamando a função mais e entregando a ela os 2 operandos. Tudo funciona como se a função fosse uma caixa similar a essa aí de cima.

4.6.5 Símbolos e Números

Um número em LISP é qualquer conjunto de dígitos (de 0 a 9), opcionalmente com um sinal (- ou +) e opcionalmente com um ponto separador. Números dividem-se em inteiros e fracionários. Inteiros são entrados como uma seqüência de números opcionalmente precedidos do sinal de menos. Não se colocam aspas. Números fracionários podem ser entrados com o ponto decimal ou com uma barra de fração. Por exemplo, 3/4 é a mesma coisa que 0.75.

Uma classe de funções importante em LISP é a dos predicados, que respondem a uma pergunta e têm como resposta t ou nil. Usualmente, seu nome termina em "P".

O predicado INTEGERP devolve t se o operando é inteiro, nil senão. O predicado NUMBERP devolve t se o operando é número (seja inteiro ou fracionário). O predicado ATOM devolve t se o operando é símbolo ou número, nil senão.

Outro símbolo importante em LISP são as palavras. Servem para nomear variáveis e funções, são usados como dados também. Constróem-se com quaisquer letra, podem ter números, mas sem começar por eles, e também o sinal de hífen.

Finalmente, números e símbolos podem ser considerados como ÁTOMOS.

Existem 4 atributos associados a um símbolo

1. seu nome (um string qualquer)
2. valor corrente: pode ser qualquer dado lisp. O valor default para um valor é o próprio nome do símbolo
3. lista de propriedades (default NIL)
4. função associada: aplicada aos operandos quando este símbolo é chamado como uma função.

A função SYMBOLP retorna t se o operando for um símbolo e nil senão. (symbolp ()) é T, e (symbolp NIL) também é T. Quando quisermos criar um determinado valor corrente para um string que tenha brancos, parênteses ou caracteres especiais, devemos colocar todo o string entre aspas duplas A função SET serve para setar um valor dentro de um símbolo. Exemplo:

```
(set 'vogais '(a e i o u)). ;Escrevendo agora
> vogais  ; <enter>,
a resposta é (a e i o u).
```

©88-08, Pedro Kantek

Já que o primeiro argumento de SET sempre precisa ser precedido de uma aspas, existe a função SETQ que prescinde da primeira aspas, MAS NÃO DA SEGUNDA. É uma maneira mais econômica de escrever o comando SET. Exemplo:

```
(setq vogais '(a e i o u)).
```

Uma variável pode ser usada mais de uma vez em uma única linha. Vale sempre, para efeito de avaliação da fórmula, a hierarquia determinada pelos parênteses. Por exemplo:

```
(setq vogais (cons 'y vogais)).
```

Símbolos t e nil

Existem 2 símbolos especiais em LISP que atendem pelo nome de t (de TRUE) e nil (de vazio). São pela própria definição, sinônimos de sim e de não. Ou também de Verdade e de Falso. Como já vimos, toda função LISP que tem como resposta um t ou um NIL, recebe o nome de predicado. Portanto, um predicado é uma função LISP que responde uma pergunta (com sim ou não). Boa parte dos predicados em LISP tem um nome terminado pela letra P. Por exemplo, NUMBERP, SYMBOLP, ZEROP, ODDP, que funcionam como mostrado na figura 4.3 Então A resposta de um predicado verdadeiro

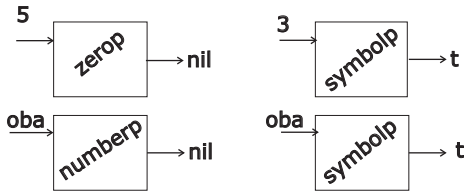


Figura 4.3: Predicados e os símbolos t e nil

é t, de um falso é nil e qualquer coisa diferente de nil pode ser entendida como t. Por exemplo, se eu disser que a resposta de ZEROP(0) é 8, isto formalmente não está errado, pois 8 não é nil, logo 8 pode ser entendido como t (quando resposta a um predicado, é óbvio).

Outro predicado importante é o NOT, que inverte o valor lógico da entrada. Assim NOT de nil é t e NOT de t é nil. Na verdade, o NOT de qualquer coisa (exceto t) é t.

Existe também o predicado EQUAL que devolve t se seus dois operandos são exatamente iguais e nil em caso contrário.

O predicado ODDP (ímpar) devolve t se o seu operando por ímpar e nil senão. O predicado EVENP (par) devolve t se o operando por par e nil senão.

Funções Compostas

Vamos começar definindo a função SOMAUM, que poderia ter a estrutura definida na figura 4.4. Com SOMAUM definido, podemos construir SOMADOIS, conforme a figura 4.5. Usando o critério de encaixar caixas, podemos construir funções cada vez mais ricas.

Se quisermos somar dois números, por exemplo 4 e 5, deveremos escrevem em LISP (+ 4 5) ao invés do nosso usual 4+5. Isto porque a fórmula esperada pelo LISP é uma lista formada por "abre parênteses", operador, operandos... e "fecha parênteses".

Se quisermos calcular $2 + 4 + 6$, em LISP deveremos escrever (+ 2 4 6). É a chamada notação pré-fixada. Note que na aritmética usual, cada sinal de "+"exige 2 operadores, um à esquerda e um à direita. Em LISP podemos ter qualquer número de operadores:

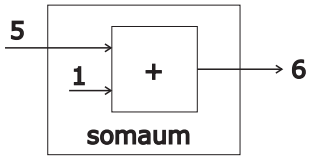


Figura 4.4: Uma função específica de soma um

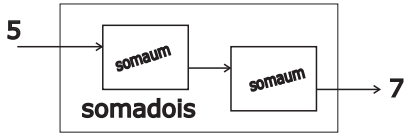


Figura 4.5: Uma função composta

```
>(+)
0
>(+ 2)
2
>(+ 3 7)
10
>(+3 6 9)
18
>(+ 1 2 3 4 5)
15
```

Nada impede que você construa funções compostas, usando várias listas, por exemplo

```
>(+(-7 2) (+2 3)) ; é igual a
10
```

A sequencia de execução do comando acima é:
a) avalia-se - 7 2, que dá como resposta 5
b) avalia-se + 2 3, que dá como resposta 5
c) avalia-se + 5 5, que dá como resposta 10.

EXERCÍCIO 46 Usando as funções +, -, *, / e os predicados EQUAL, NOT, ZEROP, >, <, ODDP, EVENP, escreva as caixas que determinam as seguintes funções:

1. MAIORQUEUMP, que devolve t se o primeiro operando é igual ao segundo operando mais um e nil senão.

```
> (defun mq (x y)
   (if (> x (+ y 1)) t nil))
MQ
> (mq 1 9)
NIL
> (mq 9 1)
T
```

```
> (mq 1 2)
NIL
> (mq 2 1)
NIL
> (mq 3 1)
T
```

2. METADE que devolve a metade do operando fornecido.

```
> (defun mt (x)
(/ x 2.0))
MT
```

3. POSITIVOP, que devolve t se o operando por maior ou igual a zero.

```
> (defun pos (x)
    (or (> x 0) (= x 0)))
POS
> (pos 5)
T
> (pos 0)
T
> (pos -1)
NIL
```

4. DIFERENTEP, que devolva t se os dois operandos forem diferentes (não iguais) e nil senão.

```
> (defun diferentep (x y)
    (not (= x y)))
DIFERENTEP
> (diferentep 5 5)
NIL
> (diferentep 5 4)
T
```

5. MAIOROUIGUALP, que devolva t se o primeiro operando é maior ou igual ao segundo operando e nil senão.

6. MARIAP, que devolve t se o seu operando for MARIA e nil senão.

```
> (defun mariap (x)
    (equal x 'maria))
MARIAP
> (mariap 'maria)
T
> (mariap 'jose)
NIL
```

7. SOMA3 que receba 3 valores e devolva a sua soma.

```
> (defun soma (x y z)
    (+ x y z))
SOMA
> (soma 1 2 3)
6
```

 ©88-08, Pedro Kantek

Domínio e contra-domínio de uma função

O domínio de uma função é o conjunto dos valores possíveis de entrada para a função e seu contra-domínio (ouimagem) é o conjunto de todos os valores possíveis de saída. Assim, o domínio da função ODDP é o conjunto dos números inteiros e sua imagem é o conjunto t, nil.

Define-se o fechamento como uma propriedade que pode se aplicar a alguma função. Uma função é fechada em seu domínio quando os valores de saída podem ser usados como entrada para a mesma função. Por exemplo, a função SOMAUM acima é fechada em seu domínio. A função ZEROP não é.

Função inversa de uma função dada é aquele que ao receber a saida da função dada, produz a entrada original a ela. Por exemplo, a função SOMAUM poderia ter a inversa denominada SUBTRAIUM. Nem toda função tem inversa.

4.6.6 Listas

Uma lista é uma composição de dados. Neste sentido um dado composto (lista) é o antônimo de átomo. Uma lista é um objeto fundamental em LISP (não esqueça o significado de LISP). Tudo em LISP pode ser representado como uma lista. Inclusive funções são representadas como listas (são listas!) em LISP.

Listas são representadas de duas maneiras, a externa e a interna. Para o usuário de LISP apenas a representação externa interessa, mas para quem quer conhecer LISP e programá-lo, ambas são importantes.

Representação externa

A representação externa de uma lista começa com um "abre parênteses", segue por uma lista de átomos – separados por pelo menos um espaço e terminado por um fecha parênteses.

Exemplos de listas

```
(LA NO ALTO DE TANTAS GLORIAS)
Uma lista com 6 elementos
(22)
Lista com um único elemento, o 22
()
Lista vazia
(2 APERTOS 1 BOLACHA 33 3 21)
Lista com 7 elementos
(A B (C D) E)
Lista com 4 elementos. O terceiro, por
sua vez é uma lista de 2 elementos
```

Este último exemplo, merece algum cuidado. Perceba a diferença entre a lista (A B (C D) E) e a lista (A B C D E). A primeira tem 4 elementos e a segunda 5. Esta construção permite a elaboração de listas mais sofisticadas, por exemplo, suponha uma lista de pessoas e para cada pessoa queremos ter o nome, a idade e a cidade de nascimento. Poderia ficar

```
((Paulo 27 Curitiba) (Marília 23 Florianópolis)
     (José 45 Maringá))
```

Quem se sentir desconfortável com esta representação pode usar:

©88-08, Pedro Kantek

```
((Paulo 27 Curitiba)
 (Marília 23 Florianópolis)
 (José 45 Maringá))
```

É a mesma coisa. Outra observação a ser feita neste ponto é que () é para todos os efeitos sinônimo de nil. Embora nil seja um símbolo e () uma lista eles significam a mesma coisa e são intercambiáveis. Portanto (A nil B) é a mesma lista que (A () B).

Note que necessariamente deverá haver uma correção no número e na disposição dos parênteses na fórmula LISP. Verifique os seguintes erros possíveis

```
(LA NO ALTO (DE TANTAS GLORIAS)
dois abre parêntes e só um fecha
)22(
Primeiro fecha e depois abre
())
Um abre e dois fecha
(A B C
abre e não fecha
```

Representação interna

Os elementos de LISP existem em LISP em um ambiente onde há uma tabela de símbolos e também um "saco"de células CONS. As células CONS se unem com o auxílio da tabela de símbolos para formar as listas, que como vimos são o elemento fundamental em LISP.

Uma célula CONS é um conjunto de 2 endereços que apontam para a tabela de símbolos. Elementos da tabela, por sua vez apontam para células CONS que são encadeadas.

O primeiro endereço da célula CONS é chamado CAR e o segundo é chamada CDR. A origem dos nomes é histórica. Na primeira implementação de LISP ainda na década de 50, usou-se um computador IBM 704 e a célula CONS foi construída usando dois dos registradores da máquina (CAR = Contents of the Address part of Register e CDR = Contents of the Decrement part of Register). COMMON LISP possui os sinônimos "first"e "rest"mas parece que ninguém os usa.

Neste desenho vemos o primeiro endereço como sendo o CAR da célula CONS e o segundo como sendo o endereço CDR da mesma célula.

Suponhamos agora querer entender a representação interna da lista

```
(TENHO 22 REAIS)
```

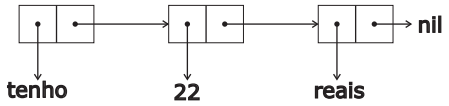


Figura 4.6: A lista (TENHO 22 REAIS) na memória

Os endereços em cada célula CONS referem-se a entradas na tabela de símbolos onde de fato os átomos ("TENHO", 22 e "REAIS") foram definidos. nil refere-se a uma das entradas da tabela que contém nil.

Agora suponhamos a representação interna da seguinte lista

```
(A (B C) D)
```

Eis como ficaria na memória da máquina

 ©88-08, Pedro Kantek

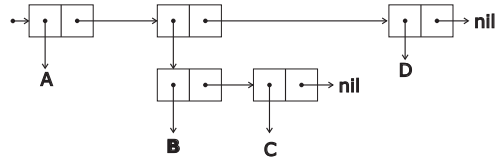


Figura 4.7: A lista (A (B C) D) na memória

EXERCÍCIO 47 Escreva a representação interna das seguintes listas:
1. ((ALBERTO E O) GATO AZUL)
2. (UM DIA (A CASA (AMARELA) VAI) CAIR)
3. (((AÇUCAR)))

EXERCÍCIO RESOLVIDO 3 Exercício sobre alocação de memória pelo interpretador LISP:
Se imaginarmos que a tabela de símbolos começa no endereço 1000 (tamanho de cada entrada=12) e o "saco"de CONS comeca em 2000, (tamanho de cada CONS=8), poderíamos ter a seguinte situação:

```
ENDE-CONT
1000-0580
1012-0639
1024-1057
1036-3358
1048-3777
1060-4155
1072-4450
1084-4466
1096-JTYA
1108-KJFC
2000-L1
1132-OYGZ
1144-QBDF
1156-RZZH
1168-UVKJ
1180-nil
1192-t

ENDE  CAR  CDR
2000- 2008 2016
2008- 1060 1180
2016- 1048 2024
2024- 2032 2040
2032- 1168 1180
2040- 1024 2048
2048- 1156 2056
2056- 2064 2088
2064- 2072 2080
```

©88-08, Pedro Kantek

```
2072- 1108 1180
2080- 1072 1180
2088- 2096 2152
2096- 2104 2112
2104- 1144 1180
2112- 2120 2128
2120- 1012 1180
2128- 1036 2136
2136- 1084 2144
2144- 1000 1180
2152- 2160 1180
2160- 1096 2168
2168- 1132 1180
```

Este exercício terá como resposta:
((4155) 3777 (UVKJ) 1057 RZZH ((KJFC) 4450)((QBDF) (0639) 3358 4466 0580) (JTYA OYGZ))

```
Respostas
1:  (CQNZ LVOX ZLTD (WRWU) (2782) JCRI ((3277) 2464 (BBQH)
    UWYC) 1331 0229 3948 ((3747)))

2:  (WIYY (0895) (BYEQ (VDZL) 4343 ((1379) ((MHDF GLRO)
    1529 (3333) (SOYJ) (1417) ((KIGO) 0371)))))
```

Desafio

- Localize um LISP que seja freeware e execute as funções apresentadas nesta aula.

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Capítulo 5

Pilhas

5.1 Pilhas

5.1.1 Introdução

Pilha é um tipo de estrutura muito comum em algoritmos de computador. Trata-se de um arranjo de dados, no qual tanto a entrada quanto a saída de elementos se dá através de uma única extremidade da estrutura. A melhor comparação para o funcionamento de uma pilha é imaginá-la como sendo uma pilha de pratos no armário da cozinha. Tanto a entrada (colocação de um novo prato limpo) quanto a saída (retirada de um prato para colocar comida sobre ele) se dão pelo topo da pilha. Veja-se uma exemplo real de pilhas na figura **??**. É interessante notar que esse tipo de estrutura implementa uma disciplina de uso conhecida como "último a entrar é o primeiro a sair ". Ela também é conhecida por sua sigla em inglês (LIFO ou last in first out). Uma pilha é um recurso imprescindível em qualquer am-biente de programação. Seja para armazenar parâmetros numéricos a serem usados em operações parentizadas, seja para guardar endereços de retorno em chamadas de sub-rotinas, seja para controlar o uso de recursos compartilhados, seja para transformar roti-nas recursivas em rotinas iterativas, enfim, para um sem-número de tarefas, o uso de uma pilha é imprescindível. Tanto é, que alguns computadores chegaram a implementar instruções nativas (de máquina) que implementam e operam com pilhas.

5.1.2 Pilhas em Alocação Seqüencial reais

Agora, não podemos mais nos valer da memória M, e precisaremos usar estruturas reais (ou pelo menos diretamente implementáveis em computadores reais, usando linguagens comuns). Entretanto, o conceito da pilha permanece rigorosamente o mesmo, similar ao de uma pilha de pratos na cozinha.

A diferença é que agora podemos usar os recursos que a nossa linguagem algorítmica nos fornece. Por exemplo, se a pilha for de tipos básicos (como inteiros ou ponto flutuantes), poderemos usar um simples vetor para fazer o armazenamento dos dados. Note-se que agora, não há necessidade de armazenar todas as informações que vimos no caso anterior. Senão vejamos: a base e o limite da pilha estarão automaticamente estabelecidos no momento da definição do vetor. A mesma consideração vale para o tamanho do elemento. Resta apenas a informação do topo da pilha, que poderá a critério do programador ser armazenado em uma posição específica do vetor (talvez a primeira, ou a última), ou ainda ser um elemento separado na estrutura.

Para simplificar (e clarificar) a notação algorítmica, nos próximos exemplos e exercícios, vamos usar a segunda alternativa, com o topo sendo especificamente definido. Diante disso, uma pilha de 50 inteiros poderia ser assim definida:

```
1: estrutura PILHA1
2: inteiro TOPO
3: inteiro ELEM [50]
4: fim estrutura
```

A criação de uma pilha qualquer (digamos P1), com a estrutura acima, ficaria

```
1: PILHA1 P1
```

Já uma pilha, com 200 elementos e na qual cada elemento tivesse 6 ponto flutuantes, poderia ser assim definida

```
1: estrutura PILHA2
2: inteiro TOPO
3: pflutuante ELEM [200] [6]
4: fim {estrutura}
```

A criação de 3 pilhas, digamos PIL1, PIL2 e PIL3 com esta estrutura ficaria

```
1: PILHA2 PIL1, PIL2, PIL3
```

Finalmente, poderíamos ter uma pilha com uma estrutura heterogênea, digamos com o nome, salário e ano de nascimento de pessoas quaisquer. Se fosse necessário criar 2 pilhas deste tipo, denominadas ADMITIDOS e DEMITIDOS com espaço para 50 elementos em cada uma, a definição da estrutura e das pilhas ficaria

```
1: estrutura PESSOA
2: alfanum NOME [40]
3: pflutuante SALÁRIO
4: inteiro ANONASC
5: fim estrutura
6: estrutura PILHA3
7: inteiro TOPO
8: PESSOA ELEM [50]
9: fim estrutura
10: ...
11: PILHA3 ADMITIDOS, DEMITIDOS
```

Inicialização Aqui não tem muito sentido em falar em criação, uma vez que boa parte do trabalho já foi feita quando da descrição e utilização da estrutura adequada. Prefere-se por isso o termo inicialização, que visa estabelecer a condição inicial de pilha vazia. Como os vetores na nossa linguagem usam como limites 0 e k- 1 (para um vetor de k elementos), o zero será elemento de indexação válido, e portanto usaremos um número negativo (de preferência -1), para indicar que a pilha está vazia.

Para os exemplos e algoritmos a seguir, vamos considerar a pilha acima definida, e repetir aqui sua estrutura e criação.

```
1: estrutura PILHA2
2: inteiro TOPO
3: pflutuante ELEM [200] [6]
4: fim estrutura
5: ...
6: PILHA2 PIL1, PIL2, PIL3
7: inteiro função INICPILHA (PILHA)
8: PILHA2 *PILHA {note que o * é imprescindível pois estruturas são passadas como
   funções através de ponteiros }
9: PILHA.TOPO ← -1 {o indicador de pilha vazia será -1}
```


©88-08, Pedro Kantek

```
10: retorne (1) {indicativo de sucesso}
11: fim função
```

Empilhamento A função de empilhamento tem a responsabilidade de colocar um novo item na pilha, devendo receber como parâmetros o endereço (apontador) da pilha e o endereço (apontador) do vetor de 6 números de ponto flutuante que deverão ser empilhados e podendo devolver uma resposta de "tudo bem", que será, por convenção o valor 1, ou a informação de que a pilha estourou, condição esta que é conhecida como overflow, e que também por convenção será o valor -1. Segue-se o algoritmo

```
 1: estrutura ELEM2
 2: pflutuante ELEM [6]
 3: fim estrutura
 4: inteiro função PILSINS (PILHA, ELEMS)
 5: PILHA2 *PILHA
 6: ELEM2 *ELEMS
 7: inteiro X
 8: se PILHA->TOPO ≥ 199 então
 9:    retorne (-1)
10: senão
11:    PILHA->TOPO++
12:    para X de 0 até 5 faça
13:       PILHA->ELEM[PILHA->TOPO] [X] ← ELEMS[X]
14:    fimpara
15:    retorne (1)
16: fimse
17: fim função
```

Desempilhamento Esta função recebe o endereço recebe como parâmetros o endereço da pilha e devolve os valores -1 (insucesso, neste caso underflow), ou o índice dentro do vetor no qual os elementos deverão ser copiados. Repare que os elementos desempilhados permanecem dentro do vetor, mas como o TOPO é diminuído em uma unidade, eles a rigor passam a fazer parte da área inacessível da pilha.

```
 1: estrutura ELEM2
 2: pflutuante ELEM [6]
 3: fim estrutura
 4: inteiro função DESEMPILHASEQ (PILHA)
 5: PILHA2 *PILHA
 6: inteiro X
 7: se PILHA->TOPO ≤ -1 então
 8:    retorne (-1)
 9: senão
10:    X ← PILHA->TOPO
11:    PILHA->TOPO–
12:    retorne (X)
13: fimse
14: fim função
```

A Pilha Seqüencial Está Vazia ? Receberá um único parâmetro (o endereço da pilha) e devolverá um valor lógico que poderá ser .V. se a pilha estiver vazia e .F. senão. Eis a sua definição:

```
1: lógico função PILHAVAZIASEQ(PILHA)
```

 ©88-08, Pedro Kantek


```
2: PILHA2 *PILHA
3: se PILHA->TOPO ≤ -1 então
4:    retorne (.V.)
5: senão
6:    retorne (.F.)
7: fimse
8: fim função
```

A Pilha Seqüencial Está Cheia ? Testa a condição de overflow da pilha, representada por um topo igual a 199.

```
1: lógico função PILHACHEIASEQ(PILHA)
2: PILHA2 *PILHA
3: se PILHA->TOPO ≥ 199 então
4:    retorne (.V.)
5: senão
6:    retorne (.F.)
7: fimse
8: fim função
```

EXERCÍCIO 48 Escreva uma função que receba três operandos inteiros. Os dois primeiros informam os endereços de 2 pilhas seqüenciais criadas em M. O terceiro operando, se par indica que deve ser desempilhado um elemento da primeira pilha. Se ímpar, é a segunda pilha que deve ser desempilhada. Entretanto, se ao tentar desempilhar uma das pilhas, houver uma condição de "pilha vazia", antes da função retornar - 1, a outra pilha deve ser pesquisa. Se esta também estiver vazia, aí sim, deve- se retornar -1. Se a segunda pilha não estiver vazia, é ela que deve ser desempilhada.

EXERCÍCIO 49 Escreva uma família de funções que implemente uma pilha em alocação seqüencial, com tamanho de elemento variável. Função de criação: deve receber como parâmetro a quantidade total de inteiros da memória que deve ser alocada. Sugere-se o uso da seguinte estrutura de apontadores: Endereço limite inferior da pilha Endereço limite superior da pilha Quantidade de elementos já inseridos na pilha (começa com 0) Endereço do topo Deve devolver o endereço dos apontadores da pilha (se sucesso), ou -1 se insucesso.

Função de empilhamento: Deve receber o endereço da pilha e o endereço inicial dos dados a empilhar. Note-se que nesse segundo endereço, o primeiro inteiro é o tamanho do elemento. A função deve devolver 0 se sucesso e -1 se insucesso.

Função de desempilhamento: Deve receber o endereço da pilha e devolver o endereço do elemento desempilhado (o primeiro inteiro também é o tamanho do elemento), ou -1 se a pilha estiver vazia

Função de consulta a espaço livre: Deve receber o endereço da pilha e devolver a quantidade de inteiros que ainda existem disponíveis na estrutura

Função de "pilha vazia". Deve receber o endereço da pilha e devolver .V. se a pilha estiver vazia ou .F. senão

EXERCÍCIO 50 Escrever um conjunto de funções para manusear uma estrutura de pilha com 3 prioridades diferentes. Na realidade serão 3 pilhas na memória, com as prioridades "A"(a mais alta), "B"e "C"(a mais baixa). As 3 pilhas guardarão elementos de mesmo tamanho. Na criação serão informadas as quantidades que devem ser alocadas para as 3 pilhas. Quando houver o empilhamento de um dado elemento, será fornecido em qual prioridade ele deve ser empilhado. Entretanto, se a pilha da prioridade solicitada, estiver cheia, antes de devolver uma condição de erro, o algoritmo deve tentar empilhá- lo nas pilhas de menor prioridade. No desempilhamento, não se informa prioridade. O elemento a ser desempilhado deverá ser

©88-08, Pedro Kantek

sempre o de maior prioridade disponível. Apenas se as 3 pilhas estiverem vazias é que a função de desempilhamento deve devolver -1.

Função de criação: deve receber 4 inteiros (quantidade de elementos da pilha "A", quantidade de elementos da pilha "B", e quantidade de elementos da pilha "C", bem como o tamanho em inteiros dos elementos). Esta função deve criar um bloco de 4 apontadores, chamado de "super pilha", (porque contém os dados das 3 pilhas), formados por: &A, &B, &C e tamanho do elemento Deve devolver o endereço da estrutura "super pilha"ou -1 se não houver espaço para a criação.

Função de inserção: deve receber o endereço da superpilha, um inteiro indicando em qual das 3 pilhas deve ser tentada a inclusão (0=A, 1=B, 2=C), e finalmente o endereço dos dados do elemento. Deve devolver a identificação da pilha que efetivamente foi usada (0, 1 ou 2), ou -1 se não houver espaço em nenhuma das 3.

Função de desempilhamento: Deve receber o endereço da superpilha e mais um endereço de memória que supostamente está livre e devolver o endereço dos dados desempilhados. No endereço de memória passado (o 2º operando) a função deve colocar a identificação da pilha que foi usada para fazer a retirada (0, 1 ou 2).

Função de consulta: Deve receber o endereço da superpilha, e mais um indicador de qual pilha é a consultada (0, 1 ou 2) e devolver a quantidade de elementos que estão empilhados na pilha informada.

EXERCÍCIO 51 Defina um algoritmo de exclusão de uma pilha encadeada sobre a memória M vista em sala de aula, que não permita que a pilha fique com menos do que 3 elementos. Isto é, antes de fazer a exclusão, a pilha deve verificar se existem 4 ou mais elementos e só se isto for verdadeiro, a exclusão deve ser feita. Se não for a exclusão deve retornar -1. O nome da função deve ser PILE3EXC, ela deve receber o endereço dos apontadores da pilha encadeada, e devolver o endereço dos dados excluídos (depois de acertados os apontadores), ou -1 se a exclusão for impossível.

Os descritores para esta pilha, podem ser

Endereço do topo	Tamanho dos elementos

EXERCÍCIO 52 Suponha que eu desejo criar uma estrutura de dados seqüencial sobre M (vista em sala de aula), que precisa de 8 descritores e 20 inteiros para área de dados. Os três primeiros descritores devem ser zerados, o quarto deve conter o endereço do primeiro descritor, o quinto o endereço do início da área de dados, o sexto o limite superior da área de dados, e o sétimo e o oitavo devem conter 2 parâmetros A e B passados para a função de criação. Defina essa função, com o nome CRIA8. A função deve devolver o endereço do primeiro parâmetro (se ela tiver sucesso) ou -1 senão.

EXERCÍCIO 53 Suponha a existência de uma pilha seqüencial contendo os seguintes descritores

BASE	TOPO	Lim Sup	Tamanho

Suponha também que precisamos uma função que faça auditoria sobre essa pilha. Note que a diferença entre BASE e TOPO precisa ser múltipla do tamanho. Escreva uma função de nome AUDITA, que receba o endereço dessa pilha e retorne 0 se a auditoria resultar OK e -1 senão (valor 1,5)

EXERCÍCIO 54 Imagine a nossa memória M[0..55] formada por inteiros e cujo controle de alocação é feito pela variável MU (que aponta para o próximo inteiro livre na memória M). Tem-se um problema que exige a existência de duas pilhas independentes. Entretanto, estas duas pilhas tem uma característica interessante. Quando uma tende a crescer a outra diminui e vice-versa. Como nossa memória é pequena, nós temos interesse em que estas duas pilhas compartilhem a mesma memória. Chamemos a primeira pilha de P1, e a segunda pilha de P2. Enquanto a primeira cresce em direção aos índices maiores de M, a P2 faz o contrário. Começa lá encima e vem diminuindo. O espaço "vazio"no meio das duas, a rigor não pertence a uma ou outra e sim aquela que primeiro o precisar. Descreva como

 ©88-08, Pedro Kantek

Figura 5.1: Duas pilhas juntas ocupando o mesmo espaço

funcionarão as pilhas na memória. Indique quais são e para que servem os apontadores das duas pilhas. Destaque as condições de overflow e underflow para ambas as pilhas. Pense e escreva algumas vantagens e/ou desvantagens que esta implementação pode acarretar. Por simplicidade do trabalho, suponha que ambas as filas trabalharão com elementos de comprimento igual a 1 inteiro.

EXERCÍCIO 55 Definir em português estruturado, uma função chamada CONSULTAPILHA (PILHA1OU2), que quando chamada retorne o conteúdo do elemento que esta' no topo da pilha indicada, SEM RETIRA-LO. Por exemplo, CONSULTAPILHA (2), consulta a pilha 2, mas não a altera.

EXERCÍCIO 56 Imagine uma pilha seqüencial implementada em C. Suponha que foi usado um vetor de 201 inteiros para implementar os elementos da pilha (cada elemento ocupa 10 inteiros e existe espaço na pilha para 20 elementos). O topo da pilha reside na posição 0 do vetor. Defina as funções de criação da pilha, empilhamento e desempilhamento.

EXERCÍCIO 57 Imagine uma pilha seqüencial implementada em C ou pascal. Suponha que os elementos da pilha ocupam um número variável (entre 1 e 10 inteiros cada um). O primeiro inteiro de cada elemento, diz qual é a quantidade de inteiros do elemento. O topo deve ser definido como uma variável externa à área de dados, embora incluída na struct da linguagem. A função de empilhamento deve receber a identificação da pilha, e um vetor de 11 elementos, no qual o primeiro valor é a quantidade de inteiros a empilhar, e os demais 10 valores serão usados na medida da necessidade. A função de desempilhamento deve devolver o mesmo formato. Condições de erro devem ser reportadas com a devolução de valores negativos como resultado das funções. Valores zero significam OK.

EXERCÍCIO 58 Implemente um conjunto de 3 funções (criação, empilhamento e desempilhamento) sobre M, para uma pilha seqüencial cujos elementos tenham tamanho variável. (Não se esqueça, que o primeiro inteiro do elemento será o seu tamanho). A função de criação só receberá o tamanho total da pilha em inteiros. A função de empilhamento receberá 3 valores: o endereço da pilha, a quantidade de inteiros a empilhar, e o endereço dos dados onde eles deverão ser apanhados. A função de desempilhamento receberá o endereço da pilha e um endereço aleatório em M onde os dados deverão ser despejados. Devolverá a quantidade de inteiros desempilhados ou -1.

Desafio

- Implementar em C (ou em qualquer outra linguagem) as funções básicas de manuseio de pilhas.
- Testar a implementação acima nas condições limite (pilha cheia e pilha vazia).

5.1.3 Pilhas em Alocação Seqüencial em M

Neste tópico, veremos apenas as pilhas implementadas em alocação seqüencial na memória M. Isto não significa que é a única maneira de implementá-las. Trata-se apenas de ir devagar e ver uma coisa de cada vez.

©88-08, Pedro Kantek

Caracterização de uma Pilha Seqüencial em M Como vimos, na alocação seqüencial, no momento de criação da estrutura precisam ser reservadas todas as posições de memória que ela vai utilizar. Em outras palavras, (e isto é uma característica da alocação seqüencial), a pilha terá um comprimento pré-determinado na sua criação. São necessárias quatro informações para caracterizar uma pilha em alocação seqüencial. São elas:

- O endereço do início dos dados da pilha (chamado BASE da pilha ou BP)
- O endereço limite até onde a pilha pode crescer (chamado LIMITE da pilha ou LP)
- O tamanho de cada elemento da pilha (chamado TAMANHO do elemento ou TA)
- A posição do elemento que está no topo da pilha (chamado TOPO da pilha ou TP)

Esses 4 números, que em conjunto recebem o nome de "descritor da pilha ", identificam uma pilha em alocação seqüencial. Usualmente eles são colocados no início da pilha. Note-se que a localização dos 4 valores pode ser no início ou no final, assim como a pilha pode crescer dos menores para os maiores valores ou vice-versa. Como tudo, na programação, o que vale são as convenções do programador. Para evitar confusões desnecessárias, a convenção aqui seguirá será a seguinte:

1. Os 4 valores serão colocados nos endereços mais baixos da pilha
2. A pilha crescerá junto com seus endereços

Para efeito de resumo do descritor, ele tem o seguinte aspecto

BP	LP	TA	TP

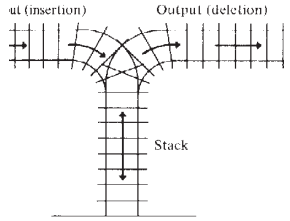


Figura 5.2: Um exemplo real de pilhas

Criação de Pilha Seqüencial em M Estamos com todas as convenções necessárias para definir a função PILSCRI . Ela deve-rá receber dois parâmetros, a quem chamaremos TAMANHO e QTDMAX. O primeiro in-dica a quantidade de inteiros que cada elemento da pilha terá, e o segundo a quantidade máxima de elementos que a pilha poderá ter. Ela devolverá o valor do endereço do primeiro inteiro da pilha (o endereço da BASE) ou o valor -1, se for impossível criá-la. Acompanhe a definição da função:

```
1: inteiro função PILSCRI (inteiro TAMANHO,QTDMAX)
2: inteiro TAMANHO, QTDMAX
3: inteiro X,Y
4: Y ← TAMANHO * QTDMAX {calcula o tamanho para a área de dados}
5: X ← AMBIOBT (Y + 4) {obtém Y + 4 (para o descritor da pilha)}
6: se (X < 0) então
7:    devolva -1 {não houve sucesso na alocação}
8: senão
9:    M[x] ← X + 4 {instala a BASE DA PILHA}
10:   M[x+1] ← M[x] + Y {instala o LIMITE SUPERIOR}
11:   M[x+2] ← TAMANHO {instala o TAMANHO DO ELEMENTO}
12:   M[x+3] ← M[x] {instala o TOPO (que é igual à BASE, pois a pilha está vazia}
13: fimse
14: fim {função}
```

Eis como ficaria uma determinada chamada para a função PILSCRI. Suponhamos M recém criada e inicializada com zeros. Logo, PPL também tem 0. Executando PILSCRI(3,5), vamos acompanhar a execução dos comandos da função:

```
1: CRIAM {M é criada, PPL ← 0}
2: P1 ← PILSCRI (3,5) {TAMANHO = 3; QTDMAX = 5, P1 é o nome da PILHA}
3: Y ← 3 × 5 {Y é 15, o tamanho da área de dados}
4: X ← AMBIOBT (19) {X receberá 0, o endereço inicial da pilha}
5: se (X < 0) então
6:    retorne (-1)
7: senão
8:    M[x] ← X + 4 {M[0] ← 4, o primeiro elemento começa em 4}
9:    M[x+1] ← M[x] + Y {M[1] ← 19, a P1 pode crescer até 18}
10:   M[x+2] ← TAMANHO {M[2] ← 3, o tamanho do elemento é 3}
11:   M[x+3] ← M[x] {M[3] ← 4, como P1 está vazia, o topo é 4}
12: fimse
13: fim função
```

Eis como ficaria M após a execução de CRIAM

`PPL=0`

M	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

Eis como ficaria após P1 ← PILSCRI (3,5). Apenas para facilitar a visualização, serão hachuriados os inteiros reservados para guardar os 5 elementos dessa pilha `PPL=19`

M7	0	1	2	3	4	5	6	7	8	9
0	4	19	3	4	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0
2	0	0	0	0	0	0	0	0	0	0

Nada impede que coexistam mais pilhas na mesma memória. Vejamos o que ocorre nesse exemplo, se em seguida, for criada uma nova pilha através do comando P2 ← PILSCRI(2,8)

Os 16 inteiros reservados para os elementos da segunda pilha também estão hachurados. PPL=39

M8	0	1	2	3	4	5	6	7	8	9
0	4	19	3	4	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	23
2	39	2	23	**0**	**0**	**0**	**0**	**0**	**0**	**0**
3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0
4	0	0	0	0	0	0	0	0	0	0

Inclusão de Elementos em uma Pilha Seqüencial em M (empilhamento ou push) Esta operação também é conhecida como "empilhamento ". Uma vez definida uma pilha seqüencial, de acordo com os padrões acima estabelecidos, temos o seu endereço de início (devolvido pela função PILSCRI), e que para todos os efeitos práticos, deste ponto em diante, passará a identificar esta pilha. Assim, se fizermos P1 ← PILSCRI(x,y), esta pilha será conhecida como PILHA P1, ou simplesmente P1.

Esta informação é necessária, na medida em que é comum a coexistência de mais de uma pilha na memória, e portanto precisamos de um identificador único para cada uma delas. Essa é a função da variável que recebe o resultado de PILSCRI.

A função PILSINS terá como finalidade a colocação de um elemento (uma quantidade pré-determinada de inteiros seqüenciais que já terão que estar gravados em M, naturalmente fora da área da pilha) no topo da pilha. PILSINS receberá dois parâmetros, PILHA e ENDELEM. O primeiro é o endereço inicial da pilha (o valor que foi devolvido pela PILSCRI), e o segundo é o endereço de M a partir do qual serão obtidos os inteiros que juntos integrarão o elemento a ser empilhado. Esta função devolve dois valores possíveis. O valor 1 significará que o empilhamento foi bem sucedido, enquanto o valor -1, indicará uma condição de erro que neste caso sempre é a falta de espaço para o novo elemento, ou seja, a pilha já estava cheia. Isso ocorre quando TP = LP. Eis como ficaria o algoritmo de PILSINS. Relembrando o descritor de uma pilha seqüencial.

BP	LP	TA	TP

```
1: inteiro função PILSINS(inteiro PILHA,ENDELEM)
2: inteiro PILHA, ENDELEM
3: inteiro BP,LP,TA,TP,X
4: BP ← M[PILHA]
5: LP ← M[PILHA + 1]
6: TA ← M[PILHA + 2]
7: TP ← M[PILHA + 3]
8: se (TP + TA ≤ LP) então
9:    X ← 0
10:   enquanto (X ≤ TA) faça
11:      M[TP + X] ← M[ENDELEM + X]
12:      X++
13:   fimenquanto
14:   M[TP] ← M[TP] + M[TA] {cálculo do novo topo}
15:   retorne (1)
16: senão
17:   retorne (-1)
18: fimse
19: fim {função}
```

Note-se que a criação das variáveis BP, LP, TA e TP a rigor não é necessária, já que podemos usar os valores de M, onde tais informações se encontram diretamente dentro da função, desde que nos permitamos usar a dupla indexação. Não há nada na nossa

 ©88-08, Pedro Kantek

linguagem algorítmica que impeça isso, portanto vamos à ela. A função PILSINS2, definida a seguir, faz exatamente a mesma coisa que PILSINS, sem nenhuma modificação, inclusive nos seus códigos de retorno. Procure entender o seu funcionamento.

```
1: inteiro função PILSINS (inteiro P,E)
2: inteiro X
3: se M[P+3] + M[P+2] ≤ M[P+1] então
4:    para X de 0 até M[P+2]-1 faça
5:       M[M[P+3]+X] ← M[E+X]
6:    fimpara
7:    M[P+3] ← M[P+3] + M[P+2]
8:    devolva 0
9: senão
10:   devolva -1
11: fimse
12: fim {função}
```

Retirada de um Elemento da Pilha Seqüencial em M (desempilhamento ou pop) Também conhecida como "desempilhamento ", esta operação retira o elemento que está no topo da pilha e devolve o endereço onde ele se inicia. Esse elemento deve ser imediatamente movido para outro local da memória, uma vez que o local onde ele está - imediatamente ao final da função de desempilhamento, passou a ser um local livre e portanto sujeito a novo empilhamento a qualquer momento. Essa função devolve também 2 códigos distintos. Um valor positivo qualquer indicará sucesso, e o valor em si indica onde está o elemento desempilhado. Já um valor -1, significará condição de erro, que neste caso significa "pilha vazia". Note que esta condição ocorre quando TP = BP.

```
1: inteiro função PILSEXC(inteiro PILHA)
2: inteiro PILHA
3: inteiro BP,LP,TA,TP
4: BP ← M[PILHA]
5: LP ← M[PILHA + 1]
6: TA ← M[PILHA + 2]
7: TP ← M[PILHA + 3]
8: se (TP = BP) então
9:    retorne (-1)
10: senão
11:   M[TP] ← M[TP] - M[TA]
12:   retorne (M[TP])
13: fimse
14: fim {função}
```

Outra maneira

```
1: inteiro função PILSEXC (inteiro P)
2: se M[P+3] = M[P] então
3:    devolva -1
4: senão
5:    M[P+3] ← M[P+3] - M[P+2]
6:    devolva M[P+3]
7: fimse
8: fim {função}
```

Da mesma forma que no caso anterior, poderemos ter PILSEXC2, que fará exatamente a mesma coisa, só que sem usar as 4 variáveis locais.

©88-08, Pedro Kantek

Não é boa norma de programação que os algoritmos que chamarão as funções PILSCRI, PILSINS e PILSEXC manuseiem diretamente a memória M. Não haverá garantia de integridade de M se for permitido a qualquer programa colocar dados lá dentro. Repare que não há nada na linguagem algorítmica que impeça isso (assim como não há nas linguagens convencionais - aquelas que não implementam o conceito de ocultamento de dados). Esse é o princípio de programação em níveis. Por essa razão, necessitamos de uma coleção de novas funções que nos dêem informações sobre uma determinada pilha seqüencial. A seguir, uma rápida palavra sobre tais funções.

A Pilha Seqüencial em M Está Vazia ? Definiremos uma função PILSVAZ, que receberá um único parâmetro (o endereço da pilha) e devolverá um valor lógico que poderá ser .V. se a pilha estiver vazia e .F. senão. Eis a sua definição:

```
1: lógico função PILSVAZ(inteiro PILHA)
2: inteiro PILHA
3: se M[PILHA] = M[PILHA+3] então
4:    retorne (.V.)
5: senão
6:    retorne (.F.)
7: fimse
8: fim {função}
```

Novamente temos uma simplificação a fazer sobre a função PILSVAZ. Acompanhe a definição de PILSVAZ2, que faz a mesma coisa. Entretanto, este tipo de simplificação só será feita aqui, e depois não mais ao longo de todo o livro. A razão é simples. Embora esta função seja menor (e alguns dirão, mais elegante) que a anterior, fica mais fácil ler (e entender) a versão completa, na qual o teste é feito e são apontadas claramente as duas alternativas possíveis. A regra aqui, é "nunca sacrificar a clareza".

```
1: lógico função PILSVAZ2(PILHA)
2: inteiro PILHA
3: retorne (M[PILHA] = M[PILHA + 3])
4: fim {função}
```

A Pilha Seqüencial em M Está Cheia ? Função parecida com a anterior, mas que testa a outra possível condição de erro, qual seja a tentativa de empilhamento em uma pilha já cheia. Eis sua definição, com o nome de PILSCHE.

```
1: lógico função PILSCHE(PILHA)
2: se M[PILHA + 4] = M[PILHA + 2] então
3:    retorne (.V.)
4: senão
5:    retorne (.F.)
6: fimse
7: fim {função}
```

Novamente temos uma simplificação a fazer sobre a função PILSCHE. Acompanhe a definição de PILSCHE2, que faz a mesma coisa.

```
1: lógico função PILSCHE2(PILHA)
2: inteiro PILHA
3: retorne (M[PILHA + 4] = M[PILHA + 2])
4: fim {função}
```

Qual o Tamanho do Elemento da Pilha Seqüencial em M Esta função chamada PILSTAM, receberá como parâmetro o endereço de uma pilha seqüencial e devolverá o tamanho do elemento daquela pilha. Acompanhe a definição:

```
1: inteiro função PILSTAM(PILHA)
```



```
2: inteiro PILHA
3: retorne M[PILHA + 2]
4: fim {função}
```

Exemplos de Pilhas Seqüenciais Faremos agora uma coleção de exemplos, que devem ser seguidos com toda a atenção pelo leitor, para que este perceba o mecanismo de convivência de diversas pilhas dentro de M, compartilhando-a. Após cada execução de uma chamada a qualquer função, M e PPL serão mostrados.

CRIAM `PPL=0`

M9	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

PILHA1 ← PILSCRI(2,3)
... PILHA1 é zero

`PPL=10`

M10	0	1	2	3	4	5	6	7	8	9
0	4	10	2	4	**0**	**0**	**0**	**0**	**0**	**0**
1	0	0	0	0	0	0	0	0	0	0

PILHA2 ← PILSCRI(3,4)
...PILHA2 é 10

`PPL=26`

M11	0	1	2	3	4	5	6	7	8	9
0	4	10	2	4	0	0	0	0	0	0
1	14	26	3	14	**0**	**0**	**0**	**0**	**0**	**0**
2	**0**	**0**	**0**	**0**	**0**	**0**	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0

PILHA3 ← PILSCRI(4,5)
... PILHA3 é 26

`PPL=50`

M12	0	1	2	3	4	5	6	7	8	9
0	4	10	2	4	0	0	0	0	0	0
1	14	26	3	14	0	0	0	0	0	0
2	0	0	0	0	0	0	30	50	4	30
3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
4	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
5	0	0	0	0	0	0	0	0	0	0

PILHA4 * PILSCRI(3,3)
... PILHA4 é 50
`PPL=63`

M14	0	1	2	3	4	5	6	7	8	9
0	4	10	2	4	0	0	0	0	0	0
1	14	26	3	14	0	0	0	0	0	0
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0

PILSINS(PILHA1,80)
...o resultado é 1 (sucesso!)
PPL=63

M15	0	1	2	3	4	5	6	7	8	9
0	4	10	2	6	1	2	0	0	0	0
1	14	26	3	14	0	0	0	0	0	0
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

PILSINS(PILHA2,81)
...o resultado é 1 (sucesso!)
PPL=63

M16	0	1	2	3	4	5	6	7	8	9
0	4	10	2	6	1	2	0	0	0	0
1	14	26	3	17	2	3	4	0	0	0
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

PILSINS(PILHA1,85)
...o resultado é 1 (sucesso!)
PPL=63

M17	0	1	2	3	4	5	6	7	8	9
0	4	10	2	8	1	2	6	7	0	0
1	14	26	3	14	0	0	0	0	0	0
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

PILSINS(PILHA2,91)
...o resultado é 1 (sucesso!)
PPL=63

©88-08, Pedro Kantek

M18	0	1	2	3	4	5	6	7	8	9
0	4	10	2	8	1	2	6	7	0	0
1	14	26	3	20	2	3	4	12	13	14
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

PILSINS(PILHA1,0)
...o resultado é 1 (sucesso!)
PPL=63

M19	0	1	2	3	4	5	6	7	8	9
0	4	10	2	10	1	2	6	7	4	10
1	14	26	3	20	2	3	4	12	13	14
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

PILSINS(PILHA1,3)
... o resultado é -1, já que a PILHA1 está cheia. Note que LP = TP PILSTAM(PILHA2)
... o resultado é 3 (a PILHA2 tem elemento com 3 inteiros) PILSEXC(PILHA2)
... o resultado é 17 (que é o endereço do elemento desempilhado) PPL=63

M20	0	1	2	3	4	5	6	7	8	9
0	4	10	2	10	1	2	6	7	4	10
1	14	26	3	17	2	3	4	12	13	14
2	0	0	0	0	0	0	30	50	4	30
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	54	63	3	54	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

5.1.4 Pilhas em Alocação Encadeada em M

Neste tópico, veremos as pilhas implementadas em alocação encadeada.

Caracterização de uma Pilha Encadeada em M Uma pilha encadeada é caracterizada apenas por 2 inteiros: o endereço do elemento e o tamanho dele.

Criação de Pilha Encadeada em M Para criar uma pilha encadeada, precisamos primeiro obter 2 posições de memória e depois alocar a pilha (colocando o terminador -1) no seu apontador e o tamanho do elemento logo a seguir.

1: inteiro função PILECRI (TAMANHO)
2: inteiro TAMANHO
3: inteiro Y
4: Y ← AMBIOBT (2)
5: **se** (Y < 0) **então**

©88-08, Pedro Kantek

```
    retorne (-1)
senão
    M[Y] ← -1
    M[Y+1] ← TAMANHO
    retorne (Y)
fimse
fim função
```

Vejamos uma simulação da execução da função acima, no caso da criação de uma pilha encadeada P1, com elemento de 3 inteiros, sobre a memória M inicialmente vazia.

```
CRIAM {M é criada, PPL ← 0}
P1 ← PILECRI (3) {TAMANHO = 3}
Y ← AMBIOBT (2) {Y receberá 0, o endereço inicial da pilha}
se (Y < 0) então
    retorne (-1)
senão
    M[0] ← -1 {M[0] ← -1, já que a pilha é criada vazia}
    M[1] ← TAMANHO {M[1] ← 3, o tamanho do elemento é 3}
fimse
fim função
```

Eis como ficaria M após a execução de CRIAM `PPL=0`

M21	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

`PPL=0`

Eis como ficaria após P1 ← PILECRI (3).

`PPL=0`

M22	0	1	2	3	4	5	6	7	8	9
0	-1	3	0	0	0	0	0	0	0	0

`PPL=2`

Nada impede que coexistam mais pilhas na mesma memória. Vejamos o que ocorre nesse exemplo, se em seguida, for criada uma nova pilha através do comando P2 ← PILECRI(8).

`PPL=2`

M23	0	1	2	3	4	5	6	7	8	9
0	-1	3	-1	8	0	0	0	0	0	0

`PPL=4`

Empilhamento em Pilha Encadeada em M

Primeiro, veremos uma implementação mais simples, no qual a pilha encadeada tem elementos de comprimento = 1, e o próprio elemento que vai ser empilhado é oferecido para a função PILEINS1. Com isso, evita-se a transferência de inteiros dentro da memória. Naturalmente, para usar PILEINS1, primeiro uma pilha encadeada deverá ter sido criada com o tamanho 1, com o comando PILECRI(1).

```
inteiro função PILEINS1(PILHA,ELEMENTO)
inteiro PILHA, ELEMENTO
inteiro X
X ← AMBIOBT (M[PILHA+1] + 1) {o resultado, nesse caso, sempre é 2}
se (X < 0) então
    retorne (-1)
senão
    M[X+1] ← ELEMENTO
    M[X] ← M[PILHA]
    M[PILHA] ← X
    retorne (0) {significando sucesso}
fimse
fim função
```

A próxima função de empilhamento encadeado, tornará genérico o tamanho do elemento a empilhar, obrigando a fazer a transferência dos inteiros que o compõe, dentro da memória.

```
inteiro função PILEINS2(PILHA,ENDELEM )
inteiro PILHA, ENDELEM
inteiro X,Y
X ← AMBIOBT (M[PILHA+1] + 1)
se (X < 0) então
    retorne (-1)
senão
    Y ← 0
    enquanto (Y < M[PILHA+1]) faça
        M[X+1+Y] ← M[ENDELEM+Y]
        Y ← Y + 1
    fimenquanto
    M[X] ← M[PILHA]
    M[PILHA] ← X
    retorne (0) {significando sucesso}
fimse
fim função
```

Desempilhamento em Pilha Encadeada em M

Esta função promove a retirada do último elemento que entrou na pilha. Ela recebe o endereço da pilha e devolve o endereço onde está o elemento que saiu. Note que não há movimentação de dados.

```
inteiro função PILEEXC(PILHA)
inteiro PILHA
inteiro X
se M[PILHA] < 0 então
    retorne (-1) {a pilha está vazia}
senão
    X ← M[PILHA]
    M[PILHA] ← M[M[PILHA]]
    retorne (X)
fimse
```

```
11: fim função
```

A Pilha Encadeada em M Está Vazia ?

```
1: lógico função PILEVAZ(PILHA)
2: inteiro PILHA
3: se M[PILHA] ≤ -1 então
4:    devolva (.V.)
5: senão
6:    devolva (.F.)
7: fimse
8: fim função
```

A Pilha Encadeada em M Está Cheia ? Pelas próprias características da pilha encadeada, não tem sentido questionar se ela esta cheia. Esta condição só ocorrerá quando a memória estiver esgotada. Entretanto, como essa situação pode ser devida a um erro de programa (só empilhar elementos, sem desempilhá-los), prevê-se aqui uma consulta a PPL, que no nosso modelo descreve as condições de utilização de M.

```
1: lógico função PILECHE (PILHA)
2: inteiro PILHA
3: se PPL > 99 - M[PILHA+1] então
4:    retorne (.V.)
5: senão
6:    retorne (.F.)
7: fimse
8: fim função
```

Qual o Tamanho do Elemento da Pilha Encadeada em M

```
1: inteiro função PILETAM(PILHA)
2: inteiro PILHA
3: retorne (M[PILHA+1])
4: fim função
```

EXERCÍCIO RESOLVIDO 4 Definindo-se as funções: r = empilha(P,i), r = desempilha (P), r = olha (P), e r = vazia (P) e supondo-se que a função x = olha(P) equivale a x = desempilha (P) seguido de um empilha (P,x), isto é olha(P) devolve o elemento que está no topo de P, SEM RETIRÁ-LO, escreva algoritmo de funções que façam:

- k = f1(P) onde k é o segundo elemento de P. P permanece inalterada.

```
1: inteiro função F1 (pilha P)
2: inteiro PRIM, X
3: PRIM ← desempilha P
4: devolve OLHA(P)
5: X ← empilha P, PRIM
6: fim função
```

- k = f2(P,n) onde k é o n-ésimo elemento de P, que perde os n elementos do topo

```
1: inteiro função F2 (pilha P, inteiro N)
2: inteiro X
3: enquanto (N > 0) faça
4:    X ← desempilha P
5:    N–
6: fimenquanto
7: devolva X
8: fim função
```

- k = f3(P) onde k é o último elemento da pilha, que fica vazia

```
1: inteiro função F3 (pilha P)
2: enquanto ¬ vazia (P) faça
3:    X ← desempilha (P)
4: fimenquanto
5: devolva X
6: fim função
```

- k = f4(P) onde k é o penúltimo elemento da pilha, que permanece só com o último elemento

```
1: inteiro função F4 (pilha P)
2: inteiro ULT, PENULT
3: PENULT ← desempilha (P)
4: ULT ← desempilha (P)
5: enquanto ¬ vazia(P) faça
6:    PENULT ← ULT
7:    ULT ← desempilha (P)
8: fimenquanto
9: empilha (P, ULT)
10: devolve PENULT
11: fim função
```

- k = f5(P) onde k é último elemento da pilha, que permanece inalterada (dica: use outra pilha)

```
1: înteiro função F5 (pilha P)
2: pilha SALVA
3: inteiro Y, X
4: enquanto ¬ vazia(P) faça
5:    X ← desempilha P
6:    Y ← empilha (SALVA, X)
7: fimenquanto
8: devolva X
9: enquanto ¬ vazia(SALVA) faça
10:    X ← desempilha (SALVA)
11:    Y ← empilha (P, X)
12: fimenquanto
13: fim função
```

EXERCÍCIO 59 Escreva uma função que receba o endereço dos apontadores de uma pilha encadeada, e retorne a quantidade de elementos que estão presentemente empilhados nela

EXERCÍCIO 60 Implemente uma superestrutura formada por 2 pilhas encadeadas, e na qual vão ser empilhados elementos formados por um único inteiro. No empilhamento da superestrutura, o elemento que está entrando deve ser empilhado na pilha que menos elementos tiver. Se ambas tiverem número igual, prevalece a pilha 1. Na retirada, será informado qual das duas pilhas (1 ou 2) deve ser usada. Sugestão: Apontadores da superestrutura: &1, &2 Apontadores da pilha 1: &topo, quantidade de elementos na pilha 1 Apontadores da pilha 2: &topo, quantidade de elementos na pilha 2

EXERCÍCIO 61 Escreva um algoritmo que percorra uma lista encadeada, devidamente registrada em "M"e retorne a quantidade de nodos que fazem parte dessa lista. O formato do nodo é

& do próximo	Inteiro 1	Inteiro 2	Inteiro 3

O endereço da lista está em M[4], e o terminador é -1.

EXERCÍCIO 62 Escreva o conjunto de algoritmos necessários para implementar uma "super-pilha", ou um conjunto de pilhas seqüenciais. O algoritmo de criação deve receber o tamanho dos elementos para todas as pilhas (sempre será o mesmo), e a quantidade de pilhas a criar. As pilhas serão numeradas de 1, 2, ... n. A função de empilhamento receberá o endereço da super-pilha, seguida do número da pilha, e o endereço dos dados a empilhar. A função devolverá 0 (se OK), ou -1 se aquela pilha solicitada estiver cheia. A função de desempilhamento receberá o endereço da super-pilha, seguida do número da pilha e devolverá o endereço dos dados, ou -1 se aquela pilha estiver vazia.

EXERCÍCIO 63 Escreva um algoritmo que leia um conjunto de caracteres terminado por um "!". Ao final deve imprimir 1 se o vetor era "adequado"e 0 senão. Um conjunto é adequado se tiver a forma a*b, onde a é qualquer seqüência de caracteres e b é a mesma seqüência EM ORDEM INVERSA. * é * mesmo. Exemplos de seqüências adequadas: *!, 123*321!, 44444*44444!, abbcccbba*abbcccbba!, 1*1!, etc.

Desafio

- Implementar o modelo M e as funções de criação, empilhamento e desempilhamento nas versões encadeada e sequencial.
- Testar o modelo acima nas condições limite (pilha cheia e pilha vazia).
- Manipular diretamente sobre M os descritores das pilhas e observar o comportamento anômalo das funções de inclusão e exclusão.

5.1.5 Aplicações de pilhas

Seja o passeio por um laberinto.

```
1: função ANDA (LABERINTO, LINHAS)
2: MAZE ← laberinto cercado por 111111, acima, abaixo, a esq e a dir
3: MARK ← 0 {mark tem as mesmas dimensoes de maze}
4: CRIAPILHA(P,3) {cria pilha P com elementos de comprimento=3}
5: MARK [ 2 ; 2 ] ← 1
6: CONTADOR ← 0
7: EMPILHA (P,(2,2,0))
8: enquanto (nãovazia(P)) faça
9:    ALFA ← DESEMPILHA(P)
10:   CONTADOR++
11:   se (CONTADOR=LINHAS) então
12:      imprima ALFA
13:   fimse
14:   I ← ALFA[1]
15:   J ← ALFA[2]
16:   MOV ← ALFA[3] + 1
17:   enquanto (MOV ≤ 4) faça
18:      G ← I + MOVE[MOV;1]
19:      H ← J + MOVE[MOV;2]
```

 ©88-08, Pedro Kantek


```
20:      se ((I = 11) ∧ (J = 16)) então
21:         ...ACABOU...
22:      fimse
23:      se ((MAZE[G ; H] = 0) ∧ (MARK[G ; H] = 0)) então
24:         mark[G ; H] ← 1
25:         empilha(P,(I,J,MOV))
26:         MOV ← 0
27:         I ← G
28:         J ← H
29:      fimse
30:      MOV++
31:   fimenquanto
32: fimenquanto
33: fim função
```

EXERCÍCIO 64 Considere o seguinte laberinto (1=parede 0=passagem). Utilize o algoritmo **??** e descubra qual o caminho percorrido para sair do laberinto.

```
0 1 1 1 0 0 0 0 0 1 0 0 0 0 0
0 1 1 1 0 1 1 1 0 1 0 1 1 1 1
0 0 0 0 0 1 1 1 0 1 0 0 0 0 0
1 1 1 1 1 1 1 1 0 1 0 1 1 1 1
1 0 0 0 0 0 0 1 0 1 0 1 0 1 0
1 1 0 1 0 1 0 1 0 0 0 0 0 0 0
1 1 1 1 1 1 0 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Análise de expressão parentizada

Suponha uma expressão do tipo ((A + B) x C) + (((D + 3) / (1 + B)) + 5). Deseja-se um algoritmo que receba a expressão e verifique se a mesma é "integra". Os critérios para esta análise são

- Existe um número igual de abre e fecha parênteses.
- Todo fecha parênteses está precedido por um abre parênteses.

Se para cada símbolo da expressão definirmos uma profundidade sendo ela igual ao número de abre parênteses subtraído dos fecha parênteses até aqui encontrados, as condições acima podem ser rescritas: Ao final da expressão, a profundidade deve ser zero e A profundidade em todos os pontos da expressão tem que ser positiva.

Analisaremos a expressão acima, segundo os critérios sugeridos

```
( ( A + B ) x C ) + ( ( ( D + 3 ) / ( 1 + B ) ) + 5 )
1 2 2 2 2 1 1 1 0 0 1 2 3 3 3 3 2 2 3 3 3 3 2 1 1 1 0
```

Supondo agora um problema levemente modificado, no qual os delimitadores são de três tipos: parênteses, colchetes e chaves, sendo que sempre um terminador deve ser do mesmo tipo que o iniciador. Ou seja a expressão (1 + [2 + 5]) está correta, mas a (1 + [2 + 5)] não está.

Este problema será resolvido usando-se uma pilha. Sempre que um abre- escopo (, [ou { for encontrado ele será empilhado. Quando um fecha escopo),] ou } for encontrado a pilha será examinada. Se o topo da pilha for equivalente ao fecha alguma

©88-08, Pedro Kantek

coisa, desempilha-se o topo e o algoritmo prossegue. Ao final a pilha deve estar vazia. Se o topo não for equivalente ao elemento em estudo, haverá erro. [1]

Conversão de funções recursivas em iterativas

Este problema é usual em informática, no mínimo pelas seguintes razões:

1. algoritmos iterativos em geral são mais eficientes que os equivalentes recursivos.
2. algumas linguagens (COBOL, por exemplo) não suportam recursividade.
3. Necessidade de controlar o processo.

A idéia é criar uma pilha com espaço para guardar todos os parâmetros que são passados à função. Cada vez que ocorreria uma chamada da função a si mesmo, os parâmetros são colocados na pilha. No início apenas o problema original é empilhado e o algoritmo começa desempilhando uma instância. O processamento segue até que a pilha esteja vazia

Transformação e cálculo de expressões RPN

Seres humanos escrevem suas sentenças matemáticas com o uso de chaves, colchetes, parênteses e ainda por cima tem uma hierarquia de operações. Se isto não for usado, temos problemas. Por exemplo:

```
5 + 4 * 3 e igual a 17, ou é
    igual a 27 ???
```

Qualquer compilador precisa entender expressões aritméticas escritas por humanos, logo, todo compilador precisa entender tais sentenças. Será que não haveria uma maneira de se escreverem estas sentenças, sem NENHUM parênteses, e SEM nenhuma hierarquia ?

Esta maneira foi inventada por Jan Lukiziewicz , e passou a ser denominada NOTAÇÃO POLONESA . Agora, mediante a introdução de um caracteres separador de operandos, as sentenças podem ser escritas sem parênteses e sem ambigüidade.

Na notação polonesa, escrevem-se os operandos e antes (prefixada) ou depois (posfixada) a operação que eles sofrerão.

```
Por exemplo, a expressão, 5 * (6 + 2) - 12 / 4,
         ficaria 5,6,2,+,*,12,4,/,-
```

Para que qualquer compilador consiga achar o valor de uma expressão com parênteses, ele sempre:

Primeiro, precisa transformar a expressão em RPN . Depois, deve calcular o valor da expressão RPN. Veja-se a seguir o algoritmo que transforma uma expressão convencional em RPN.

Entrada: a cadeia Q, contendo uma expressão parentizada normal
Saída: A resposta, será a cadeia R (equivalente a Q) expressa em RPN

```
1: cadeia Algoritmo NORMAL2RPN (cadeia Q) {Para este algoritmo funcionar, consideraremos a seguinte escala de prioridades: ; * e /, + e -. Quando duas operações de mesma prioridades existirem faremos da esquerda para a direita, a menos que haja parênteses)}
2: PILHA:pilha
3: Insira "("na pilha
4: Insira ")"ao final de Q
```

[1] Obs: este algoritmo está programado em C na pág. 93 de [Tene95].

 ©88-08, Pedro Kantek


```
5: enquanto houver elementos em Q (pesquisa da esq -> dir) faça
6:   se é operando  então
7:     Coloque-o na resposta R
8:   senão
9:     se é "(" então
10:      empilhe-o
11:    senão
12:      se é o operador (*) então
13:        retire da pilha todos os operadores com precedência igual ou maior a (*)
           colocando-os na resposta R
14:        Adicione (*) na pilha
15:      senão
16:        se é ")" então
17:          Retire da PILHA e coloque na resposta R cada operador ate achar um
             "("
18:          Remova o "("
19:        fimse
20:      fimse
21:    fimse
22:  fimse
23: fimenquanto
24: fim {algoritmo}
```

EXERCÍCIO RESOLVIDO 5 Fazer chinês com 5 * (6 + 2) - 12 / 4 Deve dar: 5,6,2 + * 12, 4 / -

EXERCÍCIO 65 Dado: A+((B*C-(D/E^F)*G)*H)
Indique qual a prioridade de execução destas operações.

A seguir, o algoritmo que calcula o valor de uma expressão RPN.

Entrada: R é uma expressão RPN correta
Saída: um real resultado da expressão, que fica no topo da pilha ao final

```
1: real Algoritmo RESOLVERPN (cadeia R)
2: pilha PILHA
3: real A,B
4: Adicione ")"ao final de R
5: enquanto houver elementos em R (da esquerda -> dir) faça
6:   se é um operando então
7:     empilhe-o
8:   senão
9:     se é um operador (*) então
10:      desempilhe B
11:      desempilhe A
12:      Calcule A (*) B
13:      empilhe o resultado
14:    fimse
15:  fimse
16: fimenquanto
17: fim {algoritmo}
```

EXERCÍCIO RESOLVIDO 6 Chinês com: 5 * (6 + 2) - 12 / 4
Transformado em 5, 6, 2, + * 12, 4 / -
E dará': 5,6,2

©88-08, Pedro Kantek

```
5,8
40,12,4
40,3
37
```

Desafio

- Implementar a saída do laberinto
- Implementar a análise de expressões parentizadas
- Implementar a conversão de expressão normal para RPN e sua posterior solução.

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Capítulo 6

Filas

6.1 Filas

Como vimos as filas tem duas extremidades. Numa se dá a entrada e em outra a saída. Como nas estruturas encadeadas (simplesmente, isto é com único encadeamento), só podemos guardar uma única extremidade, há que se propor uma alternativa. Relembre-se que só se pode guardar uma única extremidade, uma vez que só há um apontador disponível. A alternativa que seria guardar ambas as extremidades forçaria a manutenção de 2 apontadores por elemento, e será objeto de estudo mais para a frente no tópico "listas duplamente encadeadas".

Então não há saída: se se guarda o endereço da entrada (válido para inserções), no momento da saída há que se percorrer o encadeamento até o final. Isto tornará a exclusão lenta em relação à inserção. Se por outro lado a situação inversa for a desejada, (retirada rápida e inserção lenta), deve-se manter o apontador para a saída e percorrer a cadeia no momento da inserção.

6.2 Filas seqüenciais

Suponha filas seqüenciais com as seguintes características:

```
1: estrutura FILA
2: inteiro ENT, SAI
3: caracter LET [5]
4: fim estrutura {Condição de inicialização: ENT=0 e SAI=0}
1: função IF (FILA F, caracter L)
2: inteiro X,Y
3: se F.ENT < 5 então
4:    F.ENT++
5:    F.LET[F.ENT] ← L
6: senão
7:    se F.SAI > 0 então
8:       X ← F.ENT - F.SAI
9:       para Y de 1 até X faça
10:         F.LET[Y] ← F.LET[F.SAI+Y]
11:      fimpara
12:      F.ENT ← F.ENT - F.SAI
13:      F.SAI ← 0
14:      IF (F, L)
```



```
15:   senão
16:      erro (F cheia)
17:   fimse
18: fimse
19: fim {função}
 1: caracter função EF (FILA F)
 2: caracter X
 3: se F.SAI ≥ F.ENT  então
 4:    retorne '*' erro (F vazia)
 5: senão
 6:    F.SAI++
 7:    X ← F.LET[F.SAI]
 8:    se F.SAI = F.ENT então
 9:       F.SAI ← F.ENT ← 0
10:    fimse
11:    devolva X
12: fimse
13: fim {função}
```

```
Siga os exemplos (supondo duas Filas: 1 e 2)

a. IF(1,A)      F1= A - - - - E1=1, S1=0    F2= - - - - - E2=0, S2=0
b. IF(1,B)      F1= A B - - - E1=2, S1=0    F2= - - - - - E2=0, S2=0
c. IF(2,EF(1))  F1= A B - - - E1=2, S1=1    F2= A - - - - E2=1, S2=0
d. IF(2,X)      F1= A B - - - E1=2, S1=1    F2= A X - - - E2=2, S2=0
f. IF(2,Y)      F1= A B - - - E1=2, S1=1    F2= A X Y - - E2=3, S2=0
g. EF(1)        F1= A B - - - E1=0, S1=0    F2= A X Y - - E2=3, S2=0
h. EF(2)        F1= A B - - - E1=0, S1=0    F2= A X Y - - E2=3, S2=1
i. IF(2,EF(1))  F1= A B - - - E1=0, S1=0    F2= A X Y * - E2=4, S2=1
j. IF(1,J)      F1= J B - - - E1=1, S1=0    F2= A X Y * - E2=4, S2=1
k. IF(2,K)      F1= J B - - - E1=1, S1=0    F2= A X Y * K E2=5, S2=1
l. IF(2,M)      F1= J B - - - E1=1, S1=0    F2= X Y * K M E2=5, S2=0
```

6.3 Filas Circulares

Suponha filas circulares com as seguintes características:

```
1: estrutura FILA
2: inteiro ENT, SAI, QT
3: caracter LET [5]
4: fim estrutura
```

Condição de inicialização: ENT=0, SAI=0 e QT=0
Quantidade de elementos válidos: QT

```
 1: função IC (FILA C, caracter L)
 2: se C.QT < 5 então
 3:    C.ENT++
 4:    se C.ENT ≥ 6 então
 5:       C.ENT ← 1
 6:    fimse
 7:    C.QT++
 8:    C.LET[C.ENT] ← L
 9: senão
10:    ... erro ...
11: fimse
12: fim função
 1: caracter função EC (FILA C)
```

©88-08, Pedro Kantek

```
2: caracter X
3: se C.QT > 0 então
4:    C.SAI++
5:    se C.SAI ≥ 6 então
6:       C.SAI ← 1
7:    fimse
8:    X ← C[C.SAI]
9:    C.QT–
10:   devolva X
11: senão
12:   devolva '*'
13: fimse
14: fim função
```

```
Siga os exemplos (supondo duas Filas Circulares: 1 e 2)

a.IC(1,A)     F1= A - - - - E=1 S=0 Q=1  F2= - - - - - E=0 S=0 Q=0
b.IC(1,B)     F1= A B - - - E=2 S=0 Q=2  F2= - - - - - E=0 S=0 Q=0
c.IC(2,EC(1)) F1= A B - - - E=2 S=1 Q=1  F2= A - - - - E=1 S=0 Q=1
d.IC(2,X)     F1= A B - - - E=2 S=1 Q=1  F2= A X - - - E=2 S=0 Q=2
e.IC(2,Y)     F1= A B - - - E=2 S=1 Q=1  F2= A X Y - - E=3 S=0 Q=3
f.EC(1)       F1= A B - - - E=2 S=2 Q=0  F2= A X Y - - E=3 S=0 Q=3
g.IC(2,EC(1)) F1= A B - - - E=2 S=2 Q=0  F2= A X Y * - E=4 S=0 Q=4
h.IC(1,J)     F1= A B J - - E=3 S=2 Q=1  F2= A X Y * - E=4 S=0 Q=4
i.IC(2,K)     F1= A B J - - E=3 S=2 Q=1  F2= A X Y * K E=5 S=0 Q=5
j.EC(2)       F1= A B J - - E=3 S=2 Q=1  F2= A X Y * K E=5 S=1 Q=4
k.IC(1,EC(2)) F1= A B J X - E=4 S=2 Q=2  F2= A X Y * K E=5 S=2 Q=3
l.IC(2,W)     F1= A B J X - E=4 S=2 Q=2  F2= W X Y * K E=1 S=2 Q=4
```

Desafio

- Implementar os algoritmos de fila seqüencial.

6.4 Filas em baixo nível

Fila Seqüencial

Haverá um bloco composto por 5 inteiros, a saber:

- endereço limite da entrada
- endereço atual da entrada
- endereço limite da saída
- endereço atual da saída
- tamanho de cada elemento

A seguir, a função que cria uma fila seqüencial em M.

```
1: inteiro função FILSCRI (inteiro TAMANHO, QTD)
2: inteiro X
3: X ← AMBIOBT (5 + TAMANHO × QTD)
4: se X ≥ 0 então
5:    M[X+1] ← M[X+2] ← M[X+3] ← X + 5
6:    M[X] ← 5 + X + (TAMANHO × QTD)
7:    M[X+4] ← TAMANHO
8:    devolve X
9: senão
```

 ©88-08, Pedro Kantek


```
10:   devolva -1
11: fimse
12: fim função
```

Agora, o algoritmo de inserção em uma fila seqüencial em M

```
1: inteiro função FILSINC (inteiro FILA, ENDEREÇO)
2: se M[FILA+1] + M[FILA+4] ≤ M[FILA] então
3:    para Y de 0 até M[FILA+4] - 1 faça
4:       M[M[FILA+1]+Y] ← M[ENDEREÇO + Y]
5:    fimpara
6:    M[FILA+1] ← M[FILA + 1] + M[FILA+4]
7:    devolve 0
8: senão
9:    se M[FILA+3]>M[FILA+2]
10:   para Y de 0 até (M[FILA+1] - M[FILA+3])-1 faça
11:      M[M[FILA+2]+Y]←M[M[FILA+3]+Y]
12:   fimpara
13:   M[FILA+3] ← M[FILA+2]
14:   M[FILA+1] ← M[FILA+2] + Y + 1
15:   devolva FILSINC (FILA, ENDEREÇO)
16: senão
17:   devolva -1
18: fimse
19: fim função
```

E, a função de exclusão em fila seqüencial.

```
1: inteiro função FILSEXC (inteiro FILA)
2: se M[FILA+1] > M[FILA+3] então
3:    SALVA ← M[FILA+3]
4:    M[FILA+3] ← M[FILA+3] + M[FILA+4]
5:    se  M[FILA+1] = M[FILA+3] então
6:       M[FILA+1] ← M[FILA+3] ← M[FILA + 2]
7:    fimse
8:    devolva SALVA
9: senão
10:   devolva -1
11: fimse
12: fim função
```

Condições especiais para filas seqüenciais:

L_e	E	L_s	S	T

- Fila Cheia: $E = L_e \wedge S = L_s$
- Fila vazia: $E = S$

Caracterização de uma Fila Encadeada em M No modelo que aqui vai ser implementado em M, usar-se-á a seguinte fila:

1. alocação encadeada (portanto: o limite de tamanho é a memória, e a alocação se dá no momento da inclusão)
2. Tamanho de elemento fixo, mas determinável quando da criação da fila
3. inserções na extremidade controlada pelo apontador, portanto rápidas

©88-08, Pedro Kantek

4. exclusões na outra extremidade, o que implica em percorrer a cadeia, até localizar o terminador. O elemento que o tiver deve ser o excluído.

Esta fila será identificada pelo seguinte bloco de descritores:

endereço do último elemento a entrar (ou -1 em caso de fila vazia	Tamanho do elemento em inteiros

Criação de Fila Encadeada em M A função a seguir, recebe o tamanho do elemento que deverá ser armazenado na fila a criar, e devolve o endereço dos descritores da fila, ou -1 em caso de erro. Eis o algoritmo

```
1: inteiro função FILFCRI tamanho
2: inteiro X
3: X ← AMBIOBT (2) {1 para o apontador e outra para o tamanho}
4: se (X < 0) então
5:    devolva -1
6: senão
7:    M[X] ← -1 {condição inicial de fila vazia}
8:    M[X+1] ← tamanho
9:    devolva X
10: fimse
11: fim função
```

Enfileiramento em Fila Encadeada em M Esta função recebe o endereço da fila (devolvido na criação) e o endereço dos dados a enfileirar e devolve 0 em caso de sucesso e -1 de fracasso. Eis o algoritmo

```
1: inteiro função FILFINS (inteiro FILA, inteiro ENDE)
2: inteiro TAMA, X, K
3: TAMA ← M[FILA + 1]
4: X ← AMBIOBT (TAMA + 1)
5: se (X < 0) então
6:    devolva -1
7: senão
8:    M[X] ← M[FILA]
9:    M[FILA] ← X
10:   para K de 0 até TAMA-1 faça
11:      M[X+1+K] ← M[ENDE+K]
12:   fimpara
13:   devolva 0
14: fimse
15: fim função
```

Desenfileiramento em Fila Encadeada em M Neste caso, a função deve receber o endereço da fila (devolvido na criação) e deve devolver o endereço dos dados que compõe o elemento. Caso a fila esteja vazia, deve devolver -1.

```
1: inteiro função FILFEXC (inteiro FILA)
2: se M[FILA] = -1 então
3:    devolva -1
4: senão
5:    ANT ← M[FILA]
6:    ATU ← M[ANT]
```

 ©88-08, Pedro Kantek


```
7:    se (ATU = -1) então
8:       M[FILA] ← -1
9:       devolva ANT + 1
10:   senão
11:      enquanto M[ATU] ≠ -1 faça
12:         ANT ← ATU
13:         ATU ← M[ATU]
14:      fimenquanto
15:      M[ANT] ← -1
16:      devolva ATU + 1
17:   fimse
18: fimse
19: fim função
```

Uma abordagem alternativa e mais eficiente para filas encadeadas, seria manter um apontador para o primeiro que entrou, pois isso evitaria o "loop"que existe no algoritmo 6.4. Vejam-se os algoritmos:

```
1: inteiro função FILTCRI (inteiro TAMANHO)
2: inteiro Z
3: Z ← AMBIOBT(3)
4: se Z ≥ 0 então
5:    M[Z] ← M[Z+1] ← -1
6:    M[Z+2] ← TAMANHO
7:    devolva Z
8: senão
9:    ... erro ...
10: fimse
```

Agora o algoritmo de inserção nesta fila

```
1: função FILTINS (inteiro FILA, DADOS)
2: inteiro TAM, Z
3: TAM ← M[FILA+2]
4: Z ← AMBIOBT(TAM+1)
5: se Z ≥ 0 então
6:    se M[FILA] = -1 ∧ M[FILA+1] = -1 então
7:       M[FILA] ← M[FILA+1] ← Z
8:       M[Z] ← -1
9:       para Y de 0 até T-1 faça
10:         M[Z+Y+1] ← M[DADOS+Y]
11:      fimpara
12:   senão
13:      M[Z] ← -1
14:      M[M[FILA+1]] ← Z
15:      M[FILA+1] ← Z
16:      para Y de 0 até T-1 faça
17:         M[Z+Y+1] ← M[DADOS+Y]
18:      fimpara
19:   fimse
20: senão
21:    ...erro...
22: fimse
```

Algoritmo de exclusão

```
1: inteiro função FILTEXC (inteiro FILA)
```

©88-08, Pedro Kantek

```
2: se M[FILA] = -1 ∧ M[FILA+1] = -1 então
3:    devolva -1 {erro, fila vazia}
4: senão
5:    devolva M[FILA]+1
6:    se M[FILA] = M[FILA+1] então
7:       M[FILA] ← M[FILA+1] ← -1
8:    senão
9:       M[FILA] ← M[M[FILA]]
10:   fimse
11: fimse
```

Filas Circulares em M

Este tipo de fila tem o mesmo conceito da fila linear, mas uma grande vantagem: nunca é necessário fazer o "empurramento"dos dados na direção da saída. Conceitualmente é como se a extremidade de uma lista estivesse conectada à outra extremidade, daí o nome. Note que não há muito sentido em ter-se filas circulares encadeadas, pois estar-se-á contornando o problema (a necessidade de fazer um epurrão nos dados) duas vezes. Não faz sentido. Veremos então as filas circulares seqüenciais.

Haverá um bloco composto por 6 inteiros, a saber:

- endereço limite superior da lista
- endereço atual da entrada (começa igual ao L_i)
- endereço limite inferior da lista
- endereço atual da saída (começa igual ao L_i)
- tamanho de cada elemento
- quantidade de elementos presentes na fila

Note-se que assumir-se-á que o tamanho do elemento, vezes um certo inteiro será igual a quantidade limite-superior menos limite-inferior. Ou seja, não haverá corte de elementos ao passar de uma extremidade a outra. Não que isto seja uma restrição absoluta mas a sua não observância complexifica os algoritmos e devemos fugir disso. Além do que a própria função de criação se encarrega de que esta igualdade seja verificada.

A seguir, a função que cria uma fila circular seqüencial em M.

```
1: inteiro função FILCCRI (inteiro TAMANHO, QTD)
2: inteiro X
3: X ← AMBIOBT (6 + TAMANHO × QTD)
4: se X ≥ 0 então
5:    M[X+1] ← M[X+2] ← M[X+3] ← X + 6
6:    M[X] ← 6 + X + (TAMANHO × QTD)
7:    M[X+4] ← TAMANHO
8:    M[X+5] ← 0
9:    devolve X
10: senão
11:   devolva -1
12: fimse
13: fim função
```

Agora, o algoritmo de inserção em uma fila circular seqüencial em M

```
1: inteiro função FILCINC (inteiro FILA, ENDEREÇO)
2: se M[FILA+5] < (M[FILA]-M[FILA+2]) ÷ M[FILA+4] então
```



```
3:    se M[FILA+1]≥M[FILA] então
4:       M[FILA+1] ← M[FILA+2]
5:    fimse
6:    para Y de 0 até M[FILA+4]-1 faça
7:       M[M[FILA+1]+Y] ← M[ENDEREÇO + Y]
8:    fimpara
9:    M[FILA+1] ← M[FILA + 1] + M[FILA+4]
10:   M[FILA+5]++
11:   devolve 0
12: senão
13:   devolva -1
14: fimse
15: fim função
```

E, a função de exclusão em fila circular seqüencial.

```
1: inteiro função FILCEXC (inteiro FILA)
2: se M[FILA+5] > 0 então
3:    SALVA ← M[FILA+3]
4:    M[FILA+3] ← M[FILA+3] + M[FILA+4]
5:    se M[FILA+3] ≥ M[FILA] então
6:       M[FILA+3] ← M[FILA+2]
7:    fimse
8:    M[FILA+5]–
9:    devolva SALVA
10: senão
11:   devolva -1
12: fimse
13: fim função
```

Filas circulares a elementos variáveis em M

Neste último tópico um refinamento sobre a possibilidade de elementos que tenham tamanho variável em uma fila circular. Note-se que segue verdadeira a regra básica da programação: quanto mais genérica a estrutura ou o programa mais complicados os algoritmos.

Para esta estrutura, vão ser admitidos elementos de tamanho variável. Assim, cada elemento terá no seu início o tamanho dele. Os descritores da estrutura são

- Limite superior da estrutura
- endereço atual da entrada na estrutura
- Limite inferior da estrutura
- endereço atual da saída
- quantidade de inteiros usados na estrutura

Algoritmo de criação

```
1: inteiro função FILVCRI (inteiro TAMANHO)
2: inteiro X
3: X ← AMBIOBT (5 + TAMANHO)
4: se X ≥ 0 então
5:    M[X+1] ← M[X+2] ← M[X+3] ← X + 5
6:    M[X] ← 5 + X + TAMANHO
```

```
7:    M[X+4] ← 0
8:    devolve X
9:  senão
10:   devolva -1
11: fimse
12: fim função
```

Agora, o algoritmo de inserção em uma fila circular seqüencial com elementos de tamanho variável em M Esta função vai receber três parametros a saber:

- O endereço da fila (o inicial dos seus descritores)
- A quantidade de inteiros a serem enfileirados
- O endereço dos dados a serem enfileirados

```
1: inteiro função FILVINC (inteiro FILA, QUANTOS, ENDEREÇO)
2: se QUANTOS + 1 + M[FILA+4] ≤ (M[FILA]-M[FILA+2]) ÷ M[FILA+4] então
3:    M[FILA+4] ← M[FILA+4] + QUANTOS + 1
4:    se M[FILA+1] = M[FILA] então
5:       M[FILA+1] ← M[FILA+2]
6:    fimse
7:    M[M[FILA+1]] ← QUANTOS
8:    M[FILA+1] ← M[FILA+1]+1
9:    para Y de 0 até QUANTOS - 1 faça
10:      se M[FILA+1] = M[FILA] então
11:         M[FILA+1] ← M[FILA+2]
12:      fimse
13:      M[M[FILA+1]] ← M[ENDEREÇO + Y]
14:      M[FILA+1]++
15:   fimpara
16:   devolve 0
17: senão
18:   devolva -1
19: fimse
20: fim {função}
```

E, a função de exclusão em fila circular seqüencial com elementos de tamanho variável. Esta função receberá dois parâmetros,

- endereço da fila
- um endereço qualquer de memória onde os dados deverão ser descarregados

A função devolverá a quantidade de inteiros descarregados no endereço pedido, ou -1 se houver erro. Note que em uma implementação real, antes de chamar esta função, dever-se-ia pedir um GETMAIN, para obter memória e passar o endereço devolvido por GETMAIN para a função de exclusão.

```
1: inteiro função FILVEXC (inteiro FILA, ONDE)
2: inteiro Y
3: Y ← M[M[FILA+3]]
4: se Y≤M[FILA+4] então
5:    M[FILA+4]←M[FILA+4]-(Y+1)
6:    M[FILA+3]++
7:    se M[FILA+3]=M[FILA] então
8:       M[FILA+3] ← M[FILA+2]
```



```
9:    fimse
10:   X ← 0
11:   enquanto X < Y faça
12:      se M[FILA+3]=M[FILA] então
13:         M[FILA+3] ← M[FILA+2]
14:      fimse
15:      M[ONDE+X] ← M[M[FILA+3]]
16:      M[FILA+3]++
17:   fimenquanto
18:   devolva Y
19: senão
20:   devolva -1
21: fimse
```

Desafio

- Implementar os algoritmos de fila circular simulando a memõria M.
- Implementar os algoritmos de fila encadeada simulando a memória M.

6.5 Aplicações de filas

Buffers

Em informática usa-se uma palavra que não tem equivalente em português para representar áreas de memória onde são guardadas informações temporariamente. A palavra inglesa é *buffer* e ela acabou sendo introduzida no jargão e usada

no dia-a-dia. Todo e qualquer processo de *bufferização* tem que ser atendido por uma fila. Veja-se por exemplo 2 programas: o do usuário imprimindo e um componente do windows retirando as páginas impressas do *buffer* e enviando-as à impressora. Note-se que em geral isto é implementado usando- se filas circulares para evitar o overhead de acertar os ponteiros ao final/início da fila.

Simulações

Toda e qualquer simulação de eventos que precisem ser atendidos com base no protocolo FIFO (caixas de banco, supermercado, atendimento de ligações...) Desta idéia nasceu a importante teoria das filas, proposta por Erlang e hoje estudada em qualquer curso de engenharia. Particularmente neste caso também desempenha importante papel a "geração de números pseudo-aleatórios". Um outro exemplo poderia ser simular a atividade nuclear de um reator, verificando qual a grossura e composição da paredes de retenção.

EXERCÍCIO RESOLVIDO 7 Programe em pseudo-código similar a C, uma fila circular de 300 inteiros. Utilize 3 apontadores (E=entrada, S=saida e QTD=quantidade de inteiros usados). Considere as seguintes situações de contorno:

- Fila cheia: QTD $\geq$ 300
- Fila vazia: QTD $\leq$ 0
- Condição de fila circular: 299 + 1 = 0

```
1: estrutura FILA
2: inteiro E, S, QTD
```

```
3: inteiro NUM[300]
4: fim estrutura
5: FILA F
6: função INICA
7: F.E ← F.S ← F.QTD ← 0
8: fim função
1: inteiro função ENFIL (inteiro X)
2: se (QTD < 300) então
3:    F.NUM[F.E] ← X
4:    F.E++
5:    se (F.E = 300) então
6:       F.E ← 0
7:    fimse
8:    F.QTD++
9:    devolva 0
10: senão
11:    devolva -1
12: fimse
13: fim função
1: inteiro função DESENF
2: se (F.QTD > 0) então
3:    devolva F.NUM[F.S] {note que nesta notação algorítmica pode-se devolver algo antes
      de encerrar a função. Em C isto não é possível. Adaptação tem que ser feita...}
4:    F.S++
5:    se F.S = 300 então
6:       F.S ← 0
7:    fimse
8:    F.QTD–
9: senão
10:    devolva -1
11: fimse
12: fim função
```

EXERCÍCIO 66 Crie uma superestrutura chamada PILA, formada por uma pilha e uma fila seqüenciais, ambas com previsão de tamanho de elemento = 3 (constante para as duas estruturas) e ambas com espaço para 10 elementos, cada uma. Apontadores da superestrutura: &P, &F Não é necessário introduzir o tamanho do elemento (3) nem nos apontadores da fila, nem nos da pilha, pois este valor é constante. A função de inserção, receberá um parâmetro que se for > 100 indicará inserção na pilha e se ? 100 indicará inserção na fila. Se a estrutura solicitada estiver cheia, devolver -1 (insucesso). Na retirada, a fila só será usada, se a pilha estiver vazia.

EXERCÍCIO 67 Rescreva a função FILSEXC, de tal modo que a cada exclusão da fila, todos os elementos sejam deslocados em relação à saída (isto é, o apontador S, de saída, terá sempre valor fixo). Em outras palavras, a cada exclusão a fila sofrerá o processo do trocador de ônibus (um passinho pra frente, faz favor).

EXERCÍCIO 68 Defina uma fila seqüencial em C, na qual os elementos terão 3 inteiros cada um. A implementação se dará sobre um vetor com 62 inteiros. V[0] contém o início da fila e V[1] contém seu final. V[2..61] contém espaço para 20 elementos. Defina as funções de criação, enfileiramento e desenfileiramento. As funções de enfileiramento e desenfileiramento receberão como parâmetro (por referência) um vetor de 3 inteiros, que cederá/receberá os

 ©88-08, Pedro Kantek

dados. A condição de deslocamento dos elementos da fila deverá ocorrer quando: A fila se esvaziar Ocorrer overflow na área de entrada, havendo espaço livre na área de saída

EXERCÍCIO 69 Definir uma fila circular, com capacidade para 10 elementos de tamanho igual a 3 inteiros cada um. Escrever a função de criação (apontadores sugeridos: limite mínimo, limite máximo, entrada, saída e quantidade de elementos), a função de inserção e a de retirada.

EXERCÍCIO 70 Definir uma função que informe a quantidade de inteiros disponíveis existentes numa estrutura do tipo fila seqüencial encadeada, com elementos de tamanho variável. Os apontadores devem ser: limite mínimo, limite máximo, entrada, saída e quantidade de elementos.

EXERCÍCIO 71 Escreva uma função que conte quantos elementos existem em uma fila circular a elementos variáveis. Ela receberá o endereço da fila e devolvera a quantidade de elementos (cuidado: não é o número de inteiros que se está pedindo...).

EXERCÍCIO 72
```
Faculdades Positivo
Curso de Bacharelado em Informática
Duração da prova: 60 minutos

1. Escreva uma função de nome PILS50C, que receba o endereço de uma pilha
(implementada sobre M, através da função vista em sala de aula com o nome de
PILSCRI) e devolva .V. se a pilha estiver com mais de 50% do espaço alocado
ocupado e .F. senão.

2. Suponha que você precisa planejar um algoritmo que receba um endereço
de uma pilha seqüencial definida sobre M, e o seu algoritmo deve verificar
se a pilha está íntegra na memória. Que tipo de verificações o algoritmo
poderia fazer ? E que tipo de testes deveriam ser feitos sobre M[0], M[1],
M[2] e M[3], para realizar essas verificações ?

3. Porquê na inserção (empilhamento) de um elemento numa pilha seqüencial
não é necessário fazer a verificação se existe espaço na memória, ou em
outras palavras, porque numa pilha seqüencial nunca vai ocorrer a condição
de "falta de memória ?"
```

Desafio

- Implemente o algoritmo de ordenação topológica usando filas.
- Compare esta implementação de ordenação topológica com outra similar que será feita quando do estudo de grafos.

©88-08, Pedro Kantek

Capítulo 7

Listas

7.1 Introdução

Uma lista é uma estrutura genérica que pode aceitar inclusões e exclusões em qualquer lugar. Nesse sentido ela é mais geral que filas e pilhas, uma vez que uma pilha pode ser implementada usando uma lista. Idem para a lista.

7.1.1 Listas em alocação seqüencial

Seja a seguir um código simples para uma lista seqüencial, de 100 inteiros, sujeita as seguintes condições:

- Lista vazia: QTD=0
- Origem dos índices = 1 (a contagem começa em 1)

A seguir, os algoritmos necessários para manuseio de listas em alocação seqüenciais (nada mais dão do que vetores). Começando com a descrição da lista:

```
1: estrutura LISTA
2: inteiro QTD
3: inteiro NCH[100]
4: fim estrutura
5: LISTA L
```

A seguir, a função de inclusão em lista seqüencial:

```
1: inteiro função INCLUI (inteiro X, K) {inclui X na posição K de L}
2: inteiro J
3: se (K < 0) então
4:    K ← abs(K-1)
5: fimse
6: se (K > QTD) então
7:    L.QTD++
8:    L.NCH[L.QTD] ← X
9: senão
10:   para J de QTD até K+1 faça
11:      L.NCH[J+1] ← L.NCH[J]
12:   fimpara
13:   L.NCH[K+1] ← X
```



```
14:   L.QTD++
15: fimse
16: fim função
```

Agora a função de exclusão em lista seqüencial:

```
1: inteiro função EXCLUI (inteiro K)
2: se (L.QTD ≤ 0) então
3:    devolva -1
4: senão
5:    devolva L.NCH[K]
6:    para J de K até L.QTD faça
7:       L.NCH[J] ← L.NCH[J+1]
8:    fimpara
9:    L.QTD–
10: fimse
11: fim função
```

EXERCÍCIO 73 Suponha uma lista duplamente encadeada em M, com o seguinte formato:

descritores

endereço em M do primeiro nodo	Endereço em M do último nodo

Escreva o algoritmo de uma função de nome PALIN que receba o endereço do bloco de descritores desta lista e devolva .T. se as letras da cadeira formarem um palíndromo, e .F. senão. (Palíndromo: texto que pode ser lido em ordem direta ou em ordem inversa com o mesmo significado, A MENOS DOS ESPAÇOS EM BRANCO). Exemplos de palíndromos:

- OTO COME MOCOTÓ
- ROMA ME TEM AMOR
- SAIRAM O TIO E OITO MARIAS
- SOCORRAM ME SUBI NO ÔNIBUS EM MARROCOS
- ME VÊ SE A PANELA DA MOÇA É DE AÇO MADALENA PAES E VEM
- A DIVA DA VIDA
- LUZ AZUL
- ATO IDIOTA
- O TRECO CERTO
- A BASE DO TETO DESABA
- ÁTILA TOLEDO MATA MODELO TALITA
- ANOTARAM A DATA DA MARATONA
- A DROGA DA GORDA

©88-08, Pedro Kantek

- O ROMANO ACATA AMORES A DAMAS AMADAS E ROMA ATACA O NAMORO
- SÁ DÁ TAPAS E SAPATADAS

Observações:

1. Não considere a acentuação. Ela só foi posta nos exemplos para facilitar a leitura. Na implementação em computador, não use acentos, senão o texto deixa de ser um palíndromo.
2. A memória M só pode conter números, assim, na implementação, ao invés de colocar a letra em M, coloque o código ASCII da letra. Neste caso a letra A é o número 65.
3. Fica sob sua responsabilidade construir uma memória M em condições de testar este algoritmo. Depois que a construir, socialize-a com seus colegas. Quantos mais testes um algoritmo sofrer, mais certeza do seu funcionamento existirá.

Desafio

- Implemente os algoritmos de inclusão e exclusão em listas seqüenciais aqui estudados.
- Dada uma lista encadeada, como a vista na folha ALLA332a (que pode ser fornecida ao programa como uma constante, a fim de não complicar desnecessariamente a tarefa), escreva um programa que faça uma auditoria sobre a lista informando ao final se a mesma está ou não correta.

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Capítulo 8

Listas Duplamente Encadeadas

8.1 Listas Duplamente Encadeadas

Suponhamos uma estrutura contendo dois encadeamentos. Através de um deles a estrutura é percorrida do início ao fim e através do segundo encadeamento é percorrida em ordem inversa. Conseqüentemente os nodos tem que contar com 2 endereços. Cada nodo aponta para o seus dois vizinhos o anterior e o próximo. O primeiro nodo aponta para o segundo e para o terminador. O último nodo aponta para o terminador e para o penúltimo. Acompanhe no desenho

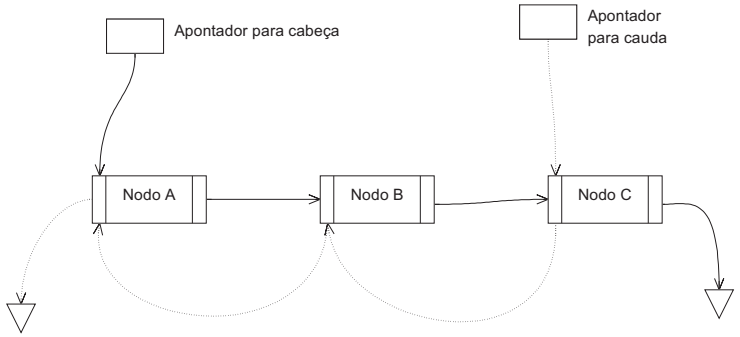


Figura 8.1: Um exemplo de lista duplamente encadeada

No desenho 8.1 verifique que existem 2 encadeamentos distintos: o de linha cheia percorre a cadeia em ordem direta. Começa por um apontador que aponta para o primeiro nodo que por sua vez aponta para o segundo e assim por diante até que o último nodo aponta para terminador.

O encadeamento tracejado por sua vez, percorre a cadeia em ordem inversa. Começa apontando para o último elemento, que aponta para o penúltimo e assim por diante, até

o primeiro que aponta para o terminador.

Genericamente Em situações onde se programa em alto nível, há que se deixarem as coisas mais genéricas. Para isso, haverá o tipo de dados nodo que descreverá o conteúdo do nodo. Haverá a função RLINK (nodo) que devolverá um apontador para o nodo anterior. Haverá LLINK (nodo) que devolverá o apontador para o próximo nodo. Haverá P? NOVONODO(), que devolverá um apontador para o espaço reservado ao novo nodo.

Agora pode-se escrever	como...
Z←AMBIOBT(x)	Z←NOVONODO()
M[Z] ← y	LLINK(Z) ← y
M[Z+1] ← w	RLINK(Z) ← w
Z ← t	Z ← t

Por conta destas convenções, haverá sempre a invariante

RLINK(LLINK(X)) = LLINK(RLINK(X)) = X

Ao se resolver o exercício da folha, devem ser mostrados os seguintes algoritmos:

```
1: estrutura ITEM
2: caracter LETRA
3: inteiro PR, ANT
4: fim{estrutura}
5: TA[100] ITEM {tabela de 100 ocorrências de letras}
6: QTD ← ... o valor correto já usado
7: HEA ← ... o valor correto da cabeça
8: TAI ← ... o valor correto da cauda
```

A seguir, a incusão

```
1: função INCLUILETRA (caracter QUEM, inteiro K)
2: //K indica o local lógico da inclusão. Fisicamente é sempre no final.
3: inteiro QTD, ATU, ANT, C
4: QTD++
5: TA[QTD].LETRA ← QUEM
6: ANT ← ATU ← HEA
7: C ← 1
8: enquanto C ≤ K faça
9:    ANT ← ATU
10:   ATU ← TA[ATU].PR
11:   C++
12: fimenquanto
13: TA[QTD].PR ← ATU
14: TA[QTD].AN ← ANT
15: TA[ANT].PR ← QTD
16: TA[ATU].AN ← QTD
```

A seguir, a exclusão

```
1: função EXCLUILETRA (inteiro K)
2: //K indica quem vai ser excluído
3: inteiro ATU, ANT, C
4: ANT ← ATU ← HEA
5: C ← 1
6: enquanto C ≤ K faça
7:    ANT ← ATU
8:    ATU ← TA[ATU].PR
```

©88-08, Pedro Kantek

9: C++
10: **fimenquanto**
11: TA[ANT].PR ← TA[ATU].PR
12: TA[TA[ATU].PR].AN ← ANT

Observação importante Os algoritmos acima, não estão completos. Eles não implementam as situações limite (inclusão no início ou no final da lista, por exemplo). Faltam os testes que fariam isso. A função só funciona para a inclusão genérica (no meio da lista). A idéia é apenas mostrar a lógica geral. Fica como exercício ao aluno, tornar os algoritmos completos.

Também não estão programados nem na inclusão nem na exclusão o tratamento de erros.

EXERCÍCIO 74 Rescreva as funções de inclusão e exclusão em uma lista duplamente encadeada usando NOVONODO, RLINK e LLINK. Teste em computador.

EXERCÍCIO 75 No desenho 8.2, represente os 4 novos apontadores que devem ser estabelecidos para as inserções respectivamente: a) à esquerda da lista (como primeiro nodo) b) entre o 2 e o terceiro nodos c) à direita da lista (como último nodo)

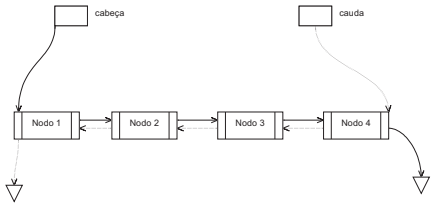


Figura 8.2: Lista para o exercício

EXERCÍCIO 76 Suponha a estrutura em alocação seqüencial:
estrutura inteiro numinsc, pflutuante salario, alfanum[30] nome
registro;
registro area[155];
A variável area é global, e já está definida dentro do seu algoritmo. Suponha também a existência de uma variável global de nome QTDITEM, inteira, e cujo valor presente sempre é a quantidade de itens dentro da estrutura area. Ou seja, QTDITEM = 0 significa que area esta vazia e QTDITEM = 155 indica que area está cheia.

Você deve definir 2 funções de nome INSERE e EXCLUI, que devem inserir e excluir novos itens dentro de area. Note que a estrutura está SEMPRE ordenada por número de inscrição em ordem descendente. Note também, que por ser uma estrutura em alocação seqüencial, deve haver contigüidade física, sem se permitirem espaços ou buracos entre os dados.

A função de INSERE deve receber 3 valores (número de inscrição, salário e nome) e devolver a posição (relativa a zero) em que a inserção foi feita ou -1 caso haja erro. A função EXCLUI deve receber apenas o número de inscrição e devolver o número que o dado originalmente ocupava, ou a informação - 1, que significará que aquela inscrição não existia.

 ©88-08, Pedro Kantek


```
Exemplo. Seja a lista:
(25;100.00;João),(32;250.00;Maria),
(108;110.00;Paulo),(110;136.00;Antônio)
e vier a função de inserção
INSERE (109,200.00,Alexandra), a lista ficará como
(25;100.00;João),(32;250.00;Maria),(108;110.00,Paulo),
(109;200.00;Alexandra),(110;136.00;Antônio)
devolvendo 3.

Nesta mesma lista, (25;100.00;João), (32;250.00;Maria),
(108;110.00,Paulo), (109;200.00;Alexandra),
(110;136.00;Antônio), a função EXCLUI(32), deve deixar a lista como
(25;100.00;João),(108;110.00,Paulo),
(109;200.00;Alexandra),(110;136.00;Antônio), devolvendo 1.
```

- Escreva uma função que percorra a lista e devolva ao final a média de salários
- Escreva função que percorra a lista e ao final informe a quantidade de salários maiores do que 100.

EXERCÍCIO 77 Imagine uma FILA definida em C, através da seguinte estrutura:
struct fulano int entrada, saída; float numbers[200];
struct fulano fila1;
Suponha que a fila já existe em seu algoritmos. Escreva o algoritmo de função que não recebe nenhum parâmetro e devolve o número que está na posição de saída. Se não houver nenhum, devolver -1. A fila não é circular.

EXERCÍCIO 78 Porque se pode dizer que uma pilha é um caso particular de lista encadeada ?

EXERCÍCIO 79 Defina uma função que retire sempre o penúltimo elemento de uma lista duplamente encadeada. (Considere conhecidas – e portanto use – as funções LIDUCRI, LIDUINS, LIDUEXC). Se a lista tiver menos do que 2 elementos, a função deve retornar -1

EXERCÍCIO 80 Imagine uma FILA definida em C, através da seguinte estrutura:
struct fulano int entrada, saída; float numbers[200];
struct fulano fila1;
Suponha que a fila já existe em seu algoritmos e não está vazia (i. é: entrada $>$ saída). Escreva o algoritmo de função que recebe um número float e enfileira-o. Lembre que pode ocorrer (e portanto deve ser programada a condição de falta de espaço na entrada, mas não na saída). A função deve devolver 0 se sucesso e -1 se fracasso. A fila não é circular.

EXERCÍCIO 81 Porque se pode dizer que uma fila é um caso particular de lista encadeada ?

EXERCÍCIO 82 Na programação, a recursividade implica sempre em 3 condições (O nome da função aparece dentro dela, ela sempre é chamada com um universo "menor"e há uma condição de fim da recursão). Escreva uma função recursiva que faça qualquer coisa (mesmo que absurda), e identifique claramente no texto essas 3 condições.

EXERCÍCIO 83 Seja a seguinte pilha definida em um programa escrito na linguagem definida em sala de aula estrutura inteiro ITEM[300], inteiro TOPO PILHA; PILHA A;

©88-08, Pedro Kantek

Escreva a função de desempilhamento, sabendo que a condição de pilha vazia é TOPO = 0. A pilha A é global, e portanto a função não receberá nenhum parâmetro, sendo específica para esta pilha. A função deve devolver o inteiro desempilhado ou a condição de erro que é a devolução do número - 20000.

EXERCÍCIO 84 Em uma clínica de emagrecimento, suponha a existência de uma lista em alocação seqüencial, mantida ordenada por número do cliente

cliente	sexo	idade da pessoa	altura	peso
5	1	18	178	110
7	2	19	145	89
13	1	12	123	78
...	...	...	...	...

O tamanho máximo da estrutura é de 200 ocorrências, e o número real de linhas ocupadas (ou seja, de clientes na estrutura) está guardado numa variável global inteira de nome QCLI.

1. Escreva o algoritmo de uma função, que recebe uma série de 5 números inteiros (cliente, sexo, idade, altura e peso), verifique se esse cliente já não existe na estrutura (se existir, deve retornar erro) e se não existir, deve incluí-lo em seu local correto. Não esquecer de incrementar QCLI.

2. Escreva o algoritmo de uma função que receba o código de um cliente, verifique se este cliente existe (se não existir, deve dar erro), e exclua-o da estrutura. A estrutura não pode ficar com menos do que 10 clientes. Se isto for ocorrer, a função deve devolver erro e não fazer a exclusão. Não esquecer de decrementar QCLI.

EXERCÍCIO 85 Em um aeroporto, existem 12 portões de embarque que são usados em regime de rodízio. Quando um portão está liberado, ele entra em uma fila e só volta a ser reutilizado quando todos os demais já o tiverem sido. Os 12 portões são identificados como 1 a 12. Você pode usar alocação seqüencial ou encadeada a seu critério.

Escreva o algoritmo de uma funcão que usando uma fila de portões livres (pode ser uma variável global), receba um código de voo (inteiro) e devolva o número do portão de embarque que deve ser utilizado por este voo. Defina a fila e todas as variáveis de controle que precisar. Escreva o algoritmo de uma função que usando a fila do aeroporto, receba o número de um portão que está sendo liberado e coloque-o na fila de portões livres.

EXERCÍCIO 86 Suponha uma lista simplesmente encadeada, cujo descritor é

& do primeiro nodo	tamanho do nodo

Os nodos tem a estrutura

& do próximo nodo	dados do nodo

Escreva o algoritmo que recebendo o endereço do descritor devolva a quantidade de nodos na lista.

Escolha como quer responder: pode ser em C ou usando o modelo M de memória. Mas uma vez escolhido o ambiente atenha-se a ele, não usando recursos de um ambiente no outro.

EXERCÍCIO 87 Escreva o algoritmo de uma função que usando a fila do aeroporto, receba o número de um portão que está sendo liberado e coloque-o na fila de portões livres.

Suponha uma lista duplamente encadeada, cujo descritor é

& cabeça	& cauda	Tamanho do nodo

O nodo tem a estrutura:

& próximo	& anterior	dados do nodo

Escreva o algoritmo que recebendo o endereço do descritor devolva a quantidade de nodos na lista.

 ©88-08, Pedro Kantek

EXERCÍCIO 88 Escolha como quer responder: pode ser em C ou usando o modelo M de memória. Mas uma vez escolhido o ambiente atenha-se a ele, não usando recursos de um ambiente no outro.

impolementar uma fila de candidatos a emprego

Implementar uma lista de microcomputadores ordenada por número de série da CPU

EXERCÍCIO 89 Suponha a criação de 2 L.D.E.s em um ambiente programado em C real . O nodo de ambas tem a estrutura

nome	& prox	& ant
8 bytes	4 bytes	4 bytes

e elas são criadas misturadas, pela coleção de comandos:

```
cab1 = cau1 = cab2 = cau2 = NULL;
insere (1, 0,"Aldo");
insere (2, 0,"Maria");
insere (2, 0,"Nelida");
insere (1, 0,"Bene");
insere (1, 1,"Carlos");
insere (1, 3,"Danilo");
insere (2, 1,"Karol");
insere (2,-2,"Wilma");
insere (2, 1,"Lauro");
insere (2, 2,"Trajano");
insere (1, 2,"Edson");
insere (1, 0,"Frida");
insere (1, 4,"Zulmira");
insere (1, -3, "Roberta");
insere (1, 2,"Pipa");
```

a) Apresente o DESENHO da lista 1, na qual você deve incluir um novo nodo com seu nome dentro na posição -3 b) Apresente o DESENHO da lista 2, na qual deve ser excluido o nodo 7

EXERCÍCIO 90 Após a execução do trecho acima, e supondo que a alocação inicial ocorreu em X'0892', complete a linha correspondente a z = Numero

`Cabec1=0942 / Caud1=08E2 / Cabec2=08B2 / Caud2=08A2`

Número linha	Endereço nodo	Nome	& prox	& ant
0		Frida		
1		Bene		
2		Pipa		
3		Roberta		
4		Carlos		
5		Edson		
6		Zulmira		
7		Aldo		
8		Danilo		
9		Nelida		
10		Lauro		
11		Trajano		
12		Wilma		
13		Karol		
14		Maria		

©88-08, Pedro Kantek

```
0 = ( 0942 / Frida / P=08C2 / A=0000 )
1 = ( 08C2 / Bene / P=0972 / A=0942 )
2 = ( 0972 / Pipa / P=0962 / A=08C2 )
3 = ( 0962 / Roberta / P=08D2 / A=0972 )
4 = ( 08D2 / Carlos / P=0932 / A=0962 )
5 = ( 0932 / Edson / P=0952 / A=08D2 )
6 = ( 0952 / Zulmira / P=0892 / A=0932 )
7 = ( 0892 / Aldo / P=08E2 / A=0952 )
8 = ( 08E2 / Danilo / P=0000 / A=0892 )
---------------------------------------
9 = ( 08B2 / Nelida / P=0912 / A=0000 )
10= ( 0912 / Lauro / P=0922 / A=08B2 )
11= ( 0922 / Trajano / P=0902 / A=0912 )
12= ( 0902 / Wilma / P=08F2 / A=0922 )
13= ( 08F2 / Karol / P=08A2 / A=0902 )
14= ( 08A2 / Maria / P=0000 / A=08F2 )
```

EXERCÍCIO 91 Escreva o algoritmo de uma função RECURSIVA que receba 2 números inteiros e positivos A e B, calcule e devolva o resultado A+B. No corpo da função, não é permitido usar a adição e a subtração convencionais, apenas os operadores C ++ e –, que significam "mais um"e "menos um". (Valor 3,0) Exemplo: EX1(5,4) é 9.

EXERCÍCIO 92 Suponha um esquema de alocação de espaço sob UNIX, idêntico ao visto em sala de aula. Neste esquema, cada bloco ocupa 100 bytes. No diretório, há espaço para 5 endereços (Direto1, Direto2, Indireto1, Indireto2, Indireto3). Em cada bloco cabem 4 endereços. Escreva o algoritmo de uma função que receba um inteiro, indicando um tamanho de arquivo e devolva a quantidade de blocos que serão necessários na área de dados para contê-lo. (valor 3,0)

Exemplo: EX2(300) é 4, EX2(50) é 1, EX2(700) é 10.

EXERCÍCIO 93 Escreva o algoritmo de uma função que receba um vetor de 200 inteiros diferentes e positivos, calcule e imprima a soma dos 2 maiores valores do vetor. A complexidade máxima admitida é n. (valor 4,0)

EXERCÍCIO 94 Suponha uma tabela T de 100 números inteiros. Ela representa 100 cadeiras existentes em um salão. Todas as 100 cadeiras estão numeradas de 1 a 100, e portanto, a cadeira número x, corresponde ao x- ésimo elemento da tabela T. Em tempo, o primeiro elemento de T é o T[1], e o último T[100].

Quando T[x] é negativo, a cadeira x está livre Quando uma pessoa chega e senta na cadeira x, automaticamente T[x] vira 0. Finalmente, quando um grupo de pessoas chega junto, elas se sentam nas cadeiras que estiverem livres, só que os T[x] em vez de ficarem com zero, passam a apontar para a próxima cadeira de uma pessoa do grupo, formando um encadeamento.

Exemplo: suponham 10 cadeiras, sendo a 1, 2 5 e 9 livres. A 3, 6 e 7 estão ocupadas por pessoas sozinhas, e as cadeiras 4, 8 e 10 estão ocupadas por 3 pessoas juntas. T seria: T = -1, -1, 0, 8, -1, 0, 0, 10, -1, 4

Note que os grupos formam encadeamentos circulares (sem terminador), nos quais o fim do grupo é reconhecido pela repetição dos números de cadeiras. Por exemplo, note que

```
T[4] = 8
T[8] = 10
T[10] = 4  // este 4 aqui, afirma que este é o último elemento do grupo.
```


Escreva uma função que percorra T e responda:

- quantas cadeiras estão livres ?
- Quantas pessoas isoladas estão sentadas ?
- Quantas pessoas que estão em grupos estão sentadas ?
- Quantos grupos de pessoas há no salão ?
- Quantas pessoas tem o maior grupo ?
- Há mais de um grupo no salão ?
- Definindo o grupo 1 como aquele que ocupa a mais baixa cadeira que é ocupada por alguém que pertença a um grupo, pergunta-se: Há espaço para realocar o grupo 1, de modo que todos se sentem em cadeiras contíguas ?
- Supondo que o grupo 1 comeu muitas batatinhas fritas e seus componentes subitamente engordaram, necessitando 2 cadeiras juntas para sentar, seria possível realocar o grupo inteiro (considere que deve haver espaço para realocação ANTES da liberação do espaço usado atualmente).?

EXERCÍCIO 95 Um pavilhão tipo Barigüi , está vendendo espaços de 3 x 3 m para a Feira de Inutilidades que se realizará proximamente. O pavilhão tem 300 x 300, logo há 10.000 espaços a venda. As empresas podem comprar um único espaço ou quanto quiserem, até o limite do que está disponível para venda. Estes espaços podem ser contíguos ou separados, o freguês escolhe.

Suponha a seguinte estrutura de dados:

A. Há uma tabela seqüencial chamada N, com 1001 linhas, inicialmente preenchida com espaços, e que contém o nome da empresa compradora e o número (de 1 a 10.000) do primeiro espaço que ela comprou. B. Há um vetor numérico de 10.000 ocorrências chamado V (o primeiro elemento é V[1], e o último é V[10.000]). Os valores existentes em V são:

0 bloco livre, pode ser vendido.

<0 bloco reservado para os expositores

n o próximo bloco do comprador que comprou este bloco é o bloco n

10001 este bloco é o último da seqüencia de blocos de um determinado comprador.

Exemplos: Seja N

Empresa	Bloco inicial
Jeca Tatu	4
Palombeta	2
(espaços)	0
...	...

Haverá sempre pelos menos 1 linha com espaços ao final de N, ou seja o número máximo de empresas clientes é 1000. Exemplo, seja V

1	2	3	4	5	6	7	8	9	10	11	12
0	10001	-1	5	8	0	0	10	-2	10001	-2	-2

Atente que a empresa Jeca Tatu comprou o bloco 4, e além deste, o 5, o 8 e o 10. O 10 é o último pois V[10] = 10001. Já a empresa Palombeta, comprou apenas o bloco 2.

Examinando N e V, escreva algoritmo de uma função que :

1. Determine a quantidade de empresas que compraram espaços na feira

2. Determine qual é o espaço ocupado pela empresa que mais espaço comprou
3. Determine quantas empresas nanicas (só compraram 1 bloco) há
4. Qual o maior bloco contíguo que ainda pode ser comprado ? (considere a contiguidade apenas em 1 direção, no sentido de V)
5. Qual o espaço médio ocupado pelas empresas ?
6. Quanas empresas compraram espaços maiores que 1 bloco ?
7. Quantos blocos estão livres ? E se o organizador der um bonus de 1 espaço a cada 5 ou mais blocos comprados por uma mesma empresa, qual será o novo valor de blocos livres ?

Curiosidade

Código C para criar LDE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void crialde \index{crialde }(int k, struct fulano *i,
           struct fulano **cabe,
     struct fulano **caud);
void insere(int kk, char *nm);
void imprime(int jj);

struct fulano {
    char nome[8];
    struct fulano *prox;  //next
    struct fulano *ant;   //prior
} nodo;

struct fulano *cabe;
struct fulano *caud;

void crialde(int k,
 struct fulano *i,
 struct fulano **cabe,
 struct fulano **caud)
  {
    struct fulano *velho, *p;
    int j;
    if (k<0) {
      k = (k * -1) - 1;
    }
    if (*caud==NULL) {   // a lde estava vazia
      i->prox = NULL;
      i->ant = NULL;
      *caud = i;
      *cabe = i;
      return;
    }
    p = *cabe;
    velho = NULL;
```

 ©88-08, Pedro Kantek


```
    j = 0;
    while (p) {  // enquanto p for diferente de nulo
      if (j<k) {
velho = p;
p = p->prox;
j++;
      }
      else {
if (p->ant) {  // testa se o caso e o geral
  p->ant->prox = i;
  i->prox = p;
  i->ant = p->ant;    p->ant = i;
  return;
}
i->prox = p;  // inclusao a esquerda da lista
i->ant = NULL;
p->ant = i;
*cabe = i;
return;
      } // fecha o else
    } // fecha o while
    velho->prox = i;  // se chegou ate aqui, a inclusao
                      // e a direita de tudo
    i->prox = NULL;
    i->ant = velho;
    *caud = i;
  } // fim da funcao crialde

void insere \index{insere }(int kk, char *nm) {
  struct fulano *pp;
  pp = (struct fulano *)malloc(sizeof(nodo));
  if (!pp) {
    printf("Erro de alocacao\n");
    exit(0);
  }
  strcpy(pp->nome,nm);
  crialde(kk,pp,&cabe,&caud);
}

void imprime \index{imprime }(int jj) {
  struct fulano *z;
  z = (jj==0)?cabe:caud;
  while (z) {
    printf("( %p / %s / P=%p / A=%p )\n",z,z->nome,z->prox,z->ant);
    z = (jj==0)?z->prox:z->ant;
  }
}

void main(void) {
  cabe = caud = NULL;
  insere (0,"Aldo");
  insere (0,"Bene");
```

©88-08, Pedro Kantek

```
  insere (1,"Carlos");
  insere (3,"Danilo");
  insere (1,"Edson");
  insere (0,"Frida");
  insere (10,"Zulmira");
  insere (4,"Pipa");
  printf("Cabeca = %p   / Cauda = %p \n",cabe,caud);
  imprime(0);
  printf("----------------------------------------\n");
  imprime(1);
}
```

EXERCÍCIO 96 Escreva os algoritmos de criação, inserção, exclusao em duas pilhas compartilhando o mesmo arranjo A[1..n]. Qualquer uma das duas pilhas somente deve sofrer overflow se a soma dos elementos das duas pilhas for igual ou maior a n. Devem ser escritas as funções:

- Definição e inicialização das duas pilhas
- retorno função empilha (Pilha, elemento), onde Pilha é 1 ou 2 e elemento é um inteiro. O retorno é 0 ou -1 (erro)
- inteiro função desempilha (Pilha) Todas as funções devem ser executadas em tempo O(1) [Cor02, pág 165]

Isto está programado no ws EXERPIFI, funções ex1c, ex1emp e ex1des). Sugestão: usar n=10. Condicoes de inicializacao: BOTTOM=0 e TOP=11

EXERCÍCIO 97 Suponha um ambiente de programação onde as PILHAS são primitivas, bem como suas operações fundamentais:

```
a. inteiro função criapilha (tamanho)   // inteiro e o retorno
b. inteiro função empilha (pilha, elemento) // inteiro é o retorno
c. inteiro função desempilha (pilha) // inteiro é o elemento desempilhado
d. lógico função vazia (pilha)
```

Construa uma FILA e todas as suas operações fundamentais usando 2 destas pilhas e amalise a complexidade de cada operação [Cor02, pág 166]

EXERCÍCIO 98 Suponha um ambiente de programação onde as FILAS são primitivas, bem como suas operações fundamentais:

```
a. inteiro função criafila (tamanho)   // inteiro e o retorno
b. inteiro função enfileira (fila, elemento) // inteiro é o retorno
c. inteiro função desenfileira (fila) // inteiro é o elemento
desenfileirado
d. lógico função vazia (fila)
```

Construa uma PILHA e todas as suas operações fundamentais usando 2 destas filas e analise a complexidade de cada operação [Cor02, pág 166]

EXERCÍCIO 99 Descreva uma pilha e desenvolva as operações fundamentais que aceite elementos de tamanho variável. Cada elemento a ser empilhado terá um número variável de inteiros. Para lidar com esta dificuldade, estipula-se que o primeiro inteiro do elemento conterá a número de inteiros do elemento, que nunca será maior que 10, incluindo-se o descritor do tamanho. Todas as operações se darão usando um buffer de A[1..10].

 ©88-08, Pedro Kantek

EXERCÍCIO 100 existem 2 filas A e B de estrutura similar e que estão carregadas com um número indeterminado de elementos (1 elemento = 1 inteiro) em ordem crescente. O tamanho máximo de cada uma destas filas é de 100 elementos. Podem e devem ser usadas as primitivas: criafila, enfila, desenfila, filavazia.

- Escreva o algoritmo que intercala estas duas filas, criando uma terceira fila mas mantendo o resultado em ordem crescente. Considere que existem e estão definidas as operações fundamentais para filas e para pilhas.
- Considere que a fila A está em ordem crescente e a fila B está em ordem decrescente. Escreva o algoritmo que cria uma terceira fila em ordem crescente.
- Suponha que a o primeiro elemento de B é maior ou igual ao último elemento de A Reescreva a função que cria a terceira fila C em ordem crescente, de modo que ela seja mais eficiente do que a função escrita no caso "a"acima.

EXERCÍCIO 101 Suponha a existência de uma fila contendo até 100 inteiros. Suponha existir a necessidade de que um desses inteiros "fure a fila", passando a ocupar a posição d saída. Escreva a função que executa esta ação. Podem e devem ser usadas as primitivas (criafila, enfila, desenfila e vazia) bem como podem ser usadas tantas filas auxiliares quantas forem necessárias. O elemento que furará a fila será conhecido pela sua identificação. Especular sobre a complexidade desta operação.

EXERCÍCIO 102 Considere a existência de duas listas, denominadas L_1 e L_2 contendo números inteiros e menores do que 9.999.999, ambas ordenadas em ordem crescente, cuja descrição é:

```
inteiro AA[1000]
```

Considere que as listas estão preenchidas até o k-ésimo elemento. Na posição $k+1$ de cada uma das listas existe a constante 9.999.999, que representa o final dessa lista.
Escreva o algoritmo que gera uma lista L_3 cujo tamanho pode chegar a 1999, cujo conteúdo é $L_1 \cap L_2$. L_3 deve estar ordenada em ordem crescente. O custo máximo do algoritmo deve ser $O(n)$, onde $n \leq 1999$.

EXERCÍCIO 103 Considere a existência de duas listas, denominadas L_1 e L_2 contendo números inteiros e menores do que 9.999.999, ambas ordenadas em ordem crescente, cuja descrição é:

```
inteiro AA[1000]
```

Considere que as listas estão preenchidas até o k-ésimo elemento. Na posição $k+1$ de cada uma das listas existe a constante 9.999.999, que representa o final dessa lista.
Escreva o algoritmo que gera uma lista L_3 cujo tamanho pode chegar a 1999, cujo conteúdo é $L_1 \cup L_2$. L_3 deve estar ordenada em ordem crescente. O custo máximo do algoritmo deve ser $O(n)$, onde $n \leq 1999$.

EXERCÍCIO 104 Suponha duas listas, denominadas P_1 e P_2, contendo os coeficientes de 2 polinômios de grau 20. Escreva o algoritmo que gera o polinômio resultado da multiplicação desses dois polinômios.

EXERCÍCIO 105 Esquematize o funcionamento de 3 pilhas em um único array

©88-08, Pedro Kantek

Desafio

- Implemente as funções de criação, inclusão, exclusão e visitação normal e reversa para listas duplamente encadeadas.

8.1.1 Listas duplamente encadeadas em M

Bloco de controle Para efeito da gerência de uma lista duplamente encadeada (LIDU), ter-se-á um bloco composto de 4 inteiros. Os dois primeiros serão o apontador para a cabeça e a cauda respectivamente. O terceiro conterá o tamanho líquido do nodo (em inteiros) e o último conterá o número de nodos presentemente na lista. Note-se que a rigor esta última informação não seria necessária (esta informação sempre se pode buscar na lista, bastando percorrê-la – via indireções sucessivas – em qualquer uma das duas direções até encontrar o terminador). Entretanto o número ajuda os algoritmos e os torna mais eficientes.

Criação de uma lista duplamente encadeada em M:

```
1: inteiro função LIDUCRI (inteiro TAMANHO)
2: inteiro Z
3: Z ← AMBIOBT (4)
4: se Z ≥ 0 então
5:    M[Z] ← M[Z+1] ← -1 {a lista é criada vazia...}
6:    M[Z+2] ← TAMANHO
7:    M[Z+3] ← 0 {zero elementos no início}
8:    devolva Z
9: senão
10:   devolva -1
11: fimse
12: fim função
```

Inserção em lista duplamente encadeada em M

Para inserir em uma lista duplamente encadeada, necessita-se estabelecer um protocolo que diga ONDE a inserção vai ser feita. Lembremo-nos que a lista admite inclusão em qualquer lugar e portanto a função precisará saber do usuário onde a inclusão deve ocorrer.

Usar-se-á uma regra bem simples. O local da inserção será indicado por um inteiro positivo ou negativo e que deve ser assim interpretado: Suponhamos que o local da inclusão é dado por k, para uma lista que tem n nodos.

- Se k = 0, a inserção é no começo da lista.
- Se k ≥ n, a inserção se dá no final da lista.
- Se k = m (m é menor do que n), a inclusão se dá à direita do elemento m.
- Se k = -m (m é menor do que n), a inclusão se dá à esquerda do elemento m.

Note-se que pela formulação da regra, alguns valores distintos indicam o mesmo local. Por exemplo k=0 e k=-1 referem-se ao mesmo lugar de inserção.

EXERCÍCIO 106 Seja a seguinte lista

```
Indique onde vai ser feita a inclusão
       caso a chave de local k for igual a
```

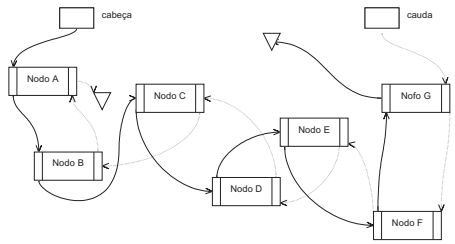



Figura 8.3: exemplo de lista duplamente encadeada

```
a) k = 2
b) k = 8
c) k = 1
d) k = -1
e) k = -3
f) k = 0
g) k = 6
h) k = -5
i) k = -7
j) k = -6
k) k = 12
l) k = 15
m) k = -33
```

O algoritmo vai fazer sempre a seguinte conversão, buscando transformar um k negativo em um positivo:

```
1: se k < 0, então k ← mod(k) - 1 fim se
```

Isto posto, este é o algoritmo de inclusão. Ele recebe 3 inteiros: o endereço dos descritores da lista, o indicado de local (k) e o endereço em M onde estão os dados a inserir e devolve 0 se a inserção for OK e -1 senão.

Descritores para listas duplamente encadeadas em M

& cabeça	& cauda	Tamanho	Quantidade de itens

E a descrição genérica para os elementos a incluir na lista é:

& do próximo	& do anterior	dados1	...	dados n

Inclusão em lista duplamente encadeada em M:

```
1: inteiro função LIDUINS (inteiro LISTA, ONDE, DADOS)
2: se ONDE < 0 então
3:    ONDE ← (abs(ONDE)) - 1
4: fimse
5: Z ← AMBIOBT 2+M[LISTA+2] {obtém tam (M[LISTA+2) + 2 int para apontadores}
6: se Z ≥ 0 então
7:    se se M[LISTA+3] = 0 então
8:       M[Z] ← M[Z+1] ← -1
9:       M[LISTA] ← M[LISTA+1] ← Z
```

```
10:       PREENCHE (Z, DADOS, M[LISTA+2])
11:     senão
12:       se ONDE = 0 então
13:         M[Z+1] ← -1
14:         M[Z] ← M[LISTA]
15:         M[M[LISTA]+1] ← Z
16:         M[LISTA] ← Z
17:         PREENCHE (Z, DADOS, M[LISTA+2])
18:       senão
19:         se  ONDE ≥ M[LISTA+3] então
20:           M[Z] ← -1
21:           M[Z+1] ← M[LISTA + 1]
22:           M[M[LISTA+1]] ← Z
23:           M[LISTA+1] ← Z
24:           PREENCHE (Z, DADOS, M[LISTA+2])
25:         senão
26:           K ← 0
27:           AUX ← M[LISTA]
28:           enquanto K < ONDE faça
29:             AUXANT ← AUX
30:             AUX ← M[AUX]
31:             K++
32:           fimenquanto
33:           M[Z] ← AUX
34:           M[Z+1] ← AUXANT
35:           M[AUXANT] ← Z
36:           M[AUX + 1] ← Z
37:           PREENCHE (Z, DADOS, M[LISTA+2])
38:         fimse
39:       fimse
40:     fimse
41:     M[LISTA+3]++
42:     devolva 0
43: senão
44:     devolva -1
45: fimse
46: fim função
```

```
1: função PREENCHE (inteiro A, E, Q)
2: inteiro Y
3: para Y de 0 até Q-1 faça
4:   M[A+2+y] ← M[E+Y]
5: fimpara
6: fim função
```

Acompanhe os exemplos resolvidos

Seja esta a memória M inicial (vazia)

 ©88-08, Pedro Kantek

M	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

Executando

```
LIDUCRI 3
```

É criada a lista no endereço 0 com tamanho 3. Será a lista 0.

M	0	1	2	3	4	5	6	7	8	9
0	**-1**	**-1**	**3**	**0**	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 0 0 90
```

É feita uma inserção na cabeça da lista 0, com os dados que estão no endereço 90

M	0	1	2	3	4	5	6	7	8	9
0	4	4	3	1	**-1**	**-1**	**11**	**12**	**13**	0
1	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUCRI 2
```

É criada uma nova lista com tamanho 2. Ela está no endereço 9. Portanto daqui para a frente será conhecida como lista 9.

M	0	1	2	3	4	5	6	7	8	9
0	4	4	3	1	-1	-1	11	12	13	**-1**
1	**-1**	**2**	**0**	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 0 1 85
```

Insere na lista 0, à direita do 1. nodo, com os dados do &85

M1	0	1	2	3	4	5	6	7	8	9
0	4	13	3	2	13	-1	11	12	13	-1
1	-1	2	0	**-1**	**4**	**6**	**7**	**8**	0	0
2	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 9 1 84
```

©88-08, Pedro Kantek

Inserção na lista 9, dados de &84 (Como a lista estava vazia, o parâmetro "onde"não tem nenhuma importância

M	0	1	2	3	4	5	6	7	8	9
0	4	13	3	2	13	-1	11	12	13	18
1	18	2	1	-1	4	6	7	8	**-1**	**-1**
2	**5**	**6**	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 0, -2 93
```

Insere na lista 0, à esquerda do 2. elemento (entre o 1. e o 2.) com dados de &93

M	0	1	2	3	4	5	6	7	8	9
0	4	13	3	3	22	-1	11	12	13	18
1	18	2	1	-1	22	6	7	8	-1	-1
2	5	6	**13**	**4**	**14**	**15**	**16**	0	0	0
3	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 0, 2 83
```

Insere na lista 0, à direita do 2 elemento (entre o 2. e o 3.). Dados de &83.

M	0	1	2	3	4	5	6	7	8	9
0	4	13	3	4	22	-1	11	12	13	18
1	18	2	1	-1	27	6	7	8	-1	-1
2	5	6	27	4	14	15	16	**13**	**22**	**4**
3	**5**	**6**	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 0, 2 87
```

Insere na lista 0, à direita do 2 elemento (entre o 2. e o 3.). Dados de &87.

M	0	1	2	3	4	5	6	7	8	9
0	4	13	3	5	22	-1	11	12	13	18
1	18	2	1	-1	27	6	7	8	-1	-1
2	5	6	32	4	14	15	16	13	32	4
3	5	6	**27**	**22**	**8**	**9**	**10**	0	0	0
4	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 0, 2 80
```

Insere na lista 0, à direita do 2 elemento (entre o 2. e o 3.). Dados de &80.

M	0	1	2	3	4	5	6	7	8	9
0	4	13	3	6	22	-1	11	12	13	18
1	18	2	1	-1	27	6	7	8	-1	-1
2	5	6	37	4	14	15	16	13	32	4
3	5	6	27	37	8	9	10	**32**	**22**	**1**
4	**2**	**3**	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

```
LIDUINS 9, 2 81
\
```

Insere na lista 9, na cauda

M	0	1	2	3	4	5	6	7	8	9
0	4	13	3	6	22	-1	11	12	13	18
1	42	2	2	-1	27	6	7	8	42	-1
2	5	6	37	4	14	15	16	13	32	4
3	5	6	27	37	8	9	10	32	22	1
4	2	3	**-1**	**18**	**2**	**3**	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
8	1	2	3	4	5	6	7	8	9	10
9	11	12	13	14	15	16	17	18	19	20

EXERCÍCIO 107 Escreva o algoritmo de uma função sob M que receba um inteiro que é o apontador para um bloco de uma lista duplamente encadeada (bloco = 4 inteiros: cabeça, cauda, tamanho do nodo e quantidade de nodos) e promova uma auditoria sobre os 2 encadeamentos. Eles devem ser percorridos (via indireções sucessivas) em ambos os sentidos e a contagem de nodos deve ser verificada com o número que está no bloco de controle. Se os 3 números (contagem direta, contagem inversa e bloco de controle) estiverem iguais, a função deve retornar .V.. Caso contrário retornar .F..

Exclusão em lista duplamente encadeada em M

Se na inclusão são necessários alterar 4 apontadores, na exclusão são apenas 2. A função de exclusão vai receber 2 inteiros: o ponteiro para o bloco de controle e o número positivo do nodo a excluir. Assim, numa lista de n nodos, uma indicador de 1 significará excluir o primeiro. Indicador $\geq$ n implicará excluir o último e igual a m (m $<$ n) significará que se deve excluir o m-ésimo nodo. A função deve devolver o endereço em M do nodo que foi excluído.

Exclusão em lista duplamente encadeada em M:

```
1: inteiro função LIDUEXC (inteiro LISTA, QUEM)
2: se M[LISTA+3] ≤ 0 então
3:    devolva -1
4: senão
5:    se M[LISTA+3] = 1 então
6:       M[LISTA] ← M[LISTA+1] ← -1
7:       M[LISTA+3] ← 0
8:       devolva M[LISTA] + 2
9:    senão
10:      se QUEM = 1 então
```

```
11:        M[LISTA] ← M[M[LISTA]]
12:        M[M[LISTA]+1] ← -1
13:        M[LISTA+3]–
14:        devolva M[LISTA] + 2
15:      senão
16:        se QUEM ≥ M[LISTA+3] então
17:          M[LISTA+1] ← M[M[LISTA+1]+1]
18:          M[M[LISTA+1]] ← -1
19:          M[LISTA+3]–
20:          devolva M[LISTA+1] + 2
21:        senão
22:          K ← 1
23:          AUX ← M[LISTA]
24:          enquanto (K < QUEM) faça
25:            AUXANT ← AUX
26:            AUX ← M[AUX]
27:            K++
28:          fimenquanto
29:          AUXPRO ← M[AUX] {AUX aponta para o nodo a excluir}
30:          M[AUXANT] ← M[AUX]
31:          M[AUXPRO+1] ← M[AUX+1]
32:          M[LISTA+3]–
33:          devolva AUX + 2
34:        fimse
35:      fimse
36:    fimse
37: fimse
38: fim função
```

EXERCÍCIO 108 Crie uma função de inserção em lista duplamente encadeada, que sempre insira o elemento novo no meio dos elementos já existentes. Não é permitido usar diretamente QTD e você pode considerar conhecidas e portanto usar as funções LIDUINS, LIDUEXC e LIDUCRI. Nesse sentido,

- se a lista estiver vazia: trivial
- se a lista tiver 1 elemento: inserir no final
- se a lista tiver número par (>0) de elementos: inserir no meio exato
- Se a lista tiver número ímpar (>1) de elementos: inserir após o elemento do meio

EXERCÍCIO 109 Defina uma função que retire sempre o penúltimo elemento de uma lista duplamente encadeada. Não é permitido acessar QTD. Se a lista tiver menos do que 2 elementos, a função deve retornar -1

EXERCÍCIO 110 Escreva 3 algoritmos que implementem uma pilha encadeada, usando as funções LIDUCRI, LIDUINS e LIDUEXC sobre M.

EXERCÍCIO 111 Suponha que em M só existe uma lista duplamente encadeada, cujo endereço é 0. Crie uma memória auxiliar de 100 inteiros, idêntica a M, chamada MAUX. Rescreva a lista duplamente encadeada em MAUX, reaproveitando os endereços que tem dados que foram excluídos. Ao final transcreva MAUX para M, não esquecendo de reacertar o PPL.

 ©88-08, Pedro Kantek

EXERCÍCIO 112 Imagine uma lista duplamente encadeada definida em M. Ela pode ter sido criada (por exemplo), pela seqüência de comandos (X ← LIDUCRI 1; LIDUINS X,0,80; LIDUINS X,0,82; LIDUEXC X,0; LIDUINS X,1,85...) Escreva o algoritmo de uma função que imprima os números que fazem parte desta lista (os conteúdos), NA ORDEM INVERSA, isto [e, da frente para o fundo O lay-out do elemento é:

& do próximo	& anterior	número enfileirado

Todos os endereços sempre se referem ao primeiro inteiro do elemento.

EXERCÍCIO 113 A função de criação de pilha seqüencial em M, vista em sala de aula tem a seguinte codificação

```
inteiro função CRIACAO (inteiro QUANTIDADE, inteiro TAMANHO)
   inteiro X
   x  $\gets$  ALOCA(4+QUANTIDADE*TAMANHO)
   se X < 0
      devolva (-1)
   senão
      M[X]  $\gets$  X+4
      M[X+1]  $\gets$  M[X] + TAMANHO*QUANTIDADE
      M[X+2]  $\gets$  TAMANHO
      M[X+3]  $\gets$  M[X]
      devolva (X)
   fim {se}
fim {função}
```

Ela foi codificada pressupondo que os a pilha cresce junto com os endereços crescentes de M. (Isto é, se a base da pilha é no endereço 10, o primeiro elemento ocupará os endereços 10, 11, ... Em outras palavras, o limite até onde os dados podem crescer é sempre maior que a base. Outra pressuposição que foi feita é que os 4 descritores da pilha estão no seu limite inferior, isto é ANTES da área de dados.

Para este exercício, você deve escrever uma nova função de criação de pilha seqüencial em M (chamada CRIACAO2), que faça exatamente a mesma coisa que a função CRIACAO acima, exceto: Os descritores devem ser colocados nos ÚLTIMOS endereços alocados para a pilha Os elementos devem ser empilhados seguindo a direção que começa nos endereços altos (e junto aos descritores) e vai em direção aos endereços pequenos da memória M

EXERCÍCIO 114 Preencha o resto desta tabela

©88-08, Pedro Kantek

Seja a seguinte memória M 4 4 1 1 -1 -1 1 0 0 0 0 0 0 0 0 0 0 0 0	Explicação Existe uma LIDU em 0. Ela possui um único elemento (M[3]) e ele está no endereço 4 (M[0]=M[1])	e o equivalente diagrama de listas vide figura 8.4
4 7 1 2 7 -1 1 -1 4 33 0 0 0 0 0 0 0 0 0 0	Um novo elemento (33) será inserido a direita do 1	vide figura 8.5
4 7 1 3 10 -1 1 -1 10 33 7 4 78 0 0 0 0 0 0 0	Um novo elemento (78) será inserido entre ambos	vide figura 8.6
	Insira um novo elemento (23) a direita do 10	
	Exclua o primeiro elemento da lista	

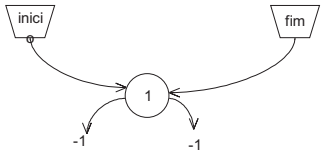


Figura 8.4: Figura para o exercício

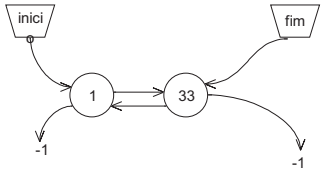


Figura 8.5: Figura para o exercício

EXERCÍCIO 115 Imagine uma lista duplamente encadeada definida em M. Ela pode ter sido criada (por exemplo), pela seqüência de comandos (X ← LIDUCRI 1; LIDUINS X,0,80; LIDUINS X,0,82; LIDUEXC X,0; LIDUINS X,1,85...) Escreva o algoritmo de uma função que

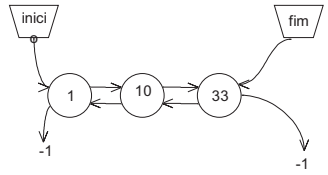


Figura 8.6: Figura para o exercício

imprima os números que fazem parte desta lista (os conteúdos), NA ORDEM DIRETA, isto
[e, da frente para o fundo. O lay-out do elemento é: | & do próximo | & anterior | número enfileirado |
Todos os endereços sempre se referem ao primeiro inteiro do elemento.

Desafio

- Implemente as funções de inclusão e exclusão em listas duplamente encadeadas sobre M.

Capítulo 9

Grafos

9.1 Leonard Euler

Não custa lembrar que as centenas de milhões de computadores que estão por aí, na face da terra, rodam programas. E, um programa nada mais é do que a materialização de um algoritmo. Assim, o conceito de algoritmo adquiriu alguma importância no nosso dia-a-dia. Eis a razão pela qual voltamo-nos para a história da vida daquele que é considerado o criador do conceito.

Fala-se de Leonard Euler, nascido na Basiléia em 1707, filho do pastor calvinista Paul Euler. O pai já havia escolhido a profissão do filho: Teologia. Embora dono de um talento impressionante para a matemática o filho obedientemente foi estudar teologia e hebraico na Universidade da Basiléia. Lá travou contato com dois irmãos Daniel e Niklauss Bernouilli, que vêm a ser membros da famosa família de matemáticos. Nada menos que 8 componentes deste clã deixaram seu nome na matemática em espaço inferior a 100 anos. A família Bernouilli era famosa: Daniel contava que um dia recebera o maior elogio de sua vida. Viajando incógnito, durante uma passeio, travou conversa com um desconhecido. No meio do papo, humildemente, apresentou-se: "Eu sou Daniel Bernouilli", ao que o conhecido fez cara de zombaria e respondeu cheio de pompa "E eu, sou Isaac Newton". Nos dias de hoje se diria "e eu, sou a Madonna".

Pois, voltando aos dois irmãos, logo depois de terem conhecido Euler, chegaram à conclusão de que valia a pena a humanidade perder um pregador medíocre em troca de um grande matemático. Foram convencer Paul Euler a que liberasse o filho. O velho Paul, que fora contemporâneo na escola de Jakob Bernouilli, o pai de todos, e tinha pelos Bernouilli muito respeito, aceitou relutantemente que o filho deixasse a Teologia.

O bacana passou a se interessar por quantos problemas passassem perto dele: estudou a navegação, as finanças, acústica, irrigação, entre outras questões. Cada novo problema levava Euler a criar matemática inovadora e engenhosa. Conseguia escrever diversos trabalhos em um único dia e contava-se que entre a primeira e a segunda chamada para o jantar, era capaz de rascunhar cálculos dignos de serem publicados. Euler tinha memória e intuição e com eles trabalhava. Era capaz de realizar um cálculo completo de cabeça, sem pôr o lápis no papel. Foi conhecido ainda em vida como "a encarnação da análise".

O primeiro algoritmo de que se tem notícia, trabalhado por Euler, é a previsão das fases da lua. Relembrando, a terra atrai a lua e a lua atrai a terra, e o conhecimento deste fato com precisão ajuda a criar tabelas de navegação, fundamentais para um navio descobrir onde está. Não nos esqueçamos que estamos no século XVIII, muito antes da existencias dos GPSs da vida.

O cálculo do comportamento da lua seria quase trivial se não aparecesse na história

o sol. É o assim chamado "problema dos 3 corpos". Euler não achou uma solução, que de resto até hoje não existe, mas trabalhou num algoritmo que permitia calcular um primeiro valor aproximado para a posição da lua. Reintroduzindo essa primeira posição no mesmo algoritmo, era possível obter um novo valor melhor, e agindo sucessivamente dessa forma poder-se-ia chegar ao valor com a precisão desejada. O Almirantado Britânico pagou 300 libras (um dinheirão) a Euler pelo algoritmo. Pensando bem, Euler deve ter sido o primeiro programador profissional da história do mundo.

Quando passou pela Rússia, Euler foi convidado pela czarina Catarina (a grande) a ajudá-la a resolver um imenso abacaxi. Andava pela corte Denis Diderot, francês famoso e ateu convicto. Dididot passava seu tempo tentando convencer as pessoas de que Deus não existia, o que deixava a religiosa Catarina furiosa. Euler, disse "deixa comigo" e imediatamente proclamou ter uma prova algébrica da existência de Deus. Rapidamente, Catarina convidou toda a corte para assistir o dilema teológico entre Euler e Diderot.

No grande dia, Euler levantou-se, pigarreou, dirigiu-se à lousa e escreveu:

"Senhor, $\frac{(a+bn)}{n} = x$, portanto Deus existe, refute!"

O pobre do Diderot que era uma nulidade matemática não conseguiu dizer nada e humilhado, deixou a corte. Não é necessário explicar, que o argumento do Euler é um baita facão. Ele deve ter sido um ótimo jogador de truco.

Outro problema estudado, pelo qual é atribuido a Euler a paternidade da Teoria dos Grafos, é o famoso problema das pontes de Königsberg (atual Kaliningrado). Explica-se: corta a cidade o Rio Pregel, formando o seguinte padrão de 4 regiões (margem esquerda, direita, ilha pequena e ilha grande) e 7 pontes.

Desde a idade média desconfiava-se que não era possível sair de um lugar qualquer, atravessar as 7 pontes uma única vez cada uma e retornar ao ponto de partida. Euler conseguiu mostrar que para este caminho existir cada ponto deve ser ligado por um número par de pontes (ou deve haver apenas 2 lugares com pontes ímpares, se estes lugares forem a saída e a chegada e forem diferentes entre si). Até hoje, na teoria dos grafos, um caminho que goze desta propriedade é chamado caminho Euleriano.

É de Euler a primeira contribuição importante para a solução do Último Teorema de Fermat (não existe n tal que $x^n + y^n = z^n$, para $n > 2$). De fato ele provou que o teorema era verdadeiro para $n = 4$.

Com a idade de 28 anos, Euler perdeu a visão de um olho, por isso grande parte dos retratos dele que foram preservados o mostra como caolho. Longe de incomodá-lo este problema não deixou seqüelas, de fato ele chegou a dizer que agora teria menos distrações com um olho a menos.

Com a idade de 60, surgiu uma catarata no olho bom, e antes que ele ficasse cego, começou a treinar a escrita com os olhos fechados: não queria parar de produzir. Pelos próximos 17 anos continuou criando matemática da melhor qualidade. Seus colegas chegaram a dizer que aparentemente a cegueira ampliara os limites de sua imaginação.

Em 1776 operou-se-lhe a catarata e por alguns dias ele voltou a enxergar. Mas logo depois veio uma infecção (ainda não havia antibióticos) e ele voltou a mergulhar na escuridão. Não se abalou e continuou a trabalhar até a idade de 84, quando segundo Condorcet, deixou de viver e de calcular.

9.2 Grafos

VIDE http://www.dct.ufms.br/ mongelli/disciplinas/graduacao/impleexpalg2003/transp/grafos grafo de relacionamento

Seja o exemplo de uma reunião de amigos para falar de futebol. Estarão convidados André, Bento, Carolina e Dudu. André é Palmeirense e depois simpatiza com o Paraná. Bento aprecia o Atlético e depois o Corintians. Carolina é coxa e também paranista.

©88-08, Pedro Kantek

Figura 9.1: Uma visão de Koenigsberg (atual Kalingrado)

Dudu é corintiano em SP e gosta do Coritiba em Curitiba. Pergunta-se há alguma maneira de sentá-los em um balcao, de modo que para todos os amigos, seus vizinhos tenham alguma afinidade ?

Desenhando um grafo, com 2 fileiras de vértices (pessoas acima e clubes abaixo), e criando arestas com o significado de "gosta de", e conectando todo mundo, fica:

 ©88-08, Pedro Kantek


```
Andre              Bento          Carolina          Dudu


Corintians     Palmeiras        Coritiba          Parana     Atlético
```

Deve-se procurar um caminho que percorra todos os vértices superiores (pessoas), passando pelos clubes, sem que nenhum vértice superior seja repetido. Se tal caminho existir, ele será uma possível distribuição das pessoas:

No exemplo acima, um caminho possível é: André, Paraná, Carolina, Coritiba, Dudu, Corintians, Bento.

Note que se a mesa fosse redonda, teríamos que repetir ao final o primeiro elemento citado. Nesta instância do problema não há esta solução.

EXERCÍCIO 116 Cinco amigas resolvem fazer um chá das 5. Assim, Adriana convida suas 4 amigas Beatriz, Cíntia, Débora e Eliza para sua casa. Adriana manda comprar uma porção de cada um dos doces: quindim, brigadeiro, pavê de morango, ambrosia e gelatina.

- Adriana gosta de quindim, brigadeiro e tudo o que é feito com morango
- Beatriz está de regime e só pode comer gelatina
- Cíntia quer comer brigadeiros, mas como está meio gordinha aceita a gelatina
- Débora aceita quindins, brigadeiros e ambrosia
- Eliza gosta de brigadeiros e de ambrosia

Pergunta-se é possível contentar a todas com estas guloseimas ?

Para responder, cada vértice tem que ser escolhido 1 e só uma vez, sendo a escolha de nome e comida, feita de uma só vez, ajudado por uma aresta ("gosta de...").

Resposta: sim, Beatriz-gelatina, Cíntia-brigadeiro, Adriana-morango, Débora-quindim e eliza-ambrosia.

EXERCÍCIO 117 Suponha a seguinte tabela de tempos de voo entre aeroportos brasileiros:

```
CWB = Curitiba, Afonso Pena
FLN = Florianópolis
POA = Porto Alegre
GRU = São Paulo, Guarulhos
BSB = Brasília
GIG = Rio de Janeiro, Galeão (Tom Jobim)
```

Com estes dados, pode-se construir uma tabela de conexões

	CWB	FLN	POA	GRU	BSB	GIG
CWB	x	115	45	∞	105	110
FLN	25	x	40	∞	∞	∞
POA	160	30	x	90	∞	100
GRU	30	∞	50	x	40	∞
BSB	∞	120	140	65	x	70
GIG	110	200	∞	45	∞	x

Terá como resposta:

```
  0   75   45  135  105  110      1   3   3   3   5   6
 25    0   40  130  130  135      1   2   3   3   1   1
 55   30    0   90  130  100      2   2   3   4   4   6
 30   80   50    0   40  110      1   3   3   4   5   5
 95  120  115   65    0   70      4   2   4   4   5   6
 75  125   95   45   85    0      4   4   4   4   4   6
```

©88-08, Pedro Kantek

- Desenhe um digrafo que mostre as ligações aéreas entre as cidades brasileiras
- Ache o menor tempo entre Curitiba e Florianópolis. Desconsidere os retardos em terra. Infinito indica que não há conexão direta entre as duas cidades.

9.2.1 Introdução

O grafo é uma ferramenta cada dia mais importante em inúmeros ramos do conhecimento. Em uma acepção simples, o grafo pode ser considerado similar a uma rede, e basta olhar onde a palavra "rede"aparece no dia-a-dia (redes de computadores, redes elétricas, redes de informação, redes de difusão, redes de transporte...) para ter uma idéia da importância do assunto. Veja-se por exemplo, um bonito grafo em 9.2

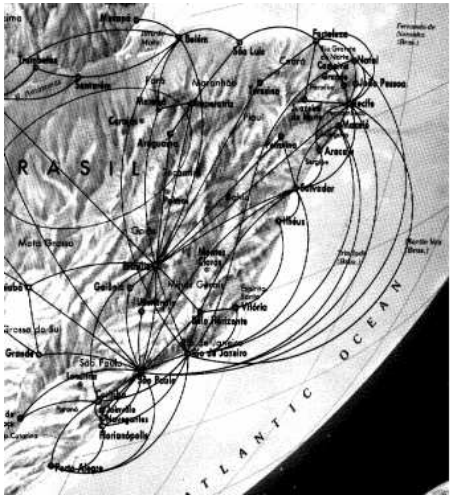


Figura 9.2: A rede de rotas de uma companhia aérea

A primeira menção histórica ao assunto surgiu num trabalho de Euler em 1736, quando ele propôs (e resolveu) o desde então célebre problema chamado "das pontes de Königsberg". Essa cidade alemã é cortada por um rio que faz duas ilhas. Chamando as duas ilhas de A e B, existem 4 pontes ligando A às margens (2 para cada margem) e 2 pontes ligando B às margens (1 cada margem). Além disso existe uma ponte ligando A e B. O problema estudado por Euler perguntava se alguém poderia passar por todas as pontes sem repetir alguma. Euler respondeu que não.

Graficamente falando, um grafo é um conjunto de pontos (chamados vértices) ligados entre si (por arestas), formando um arranjo semelhante a uma rede. Por exemplo, na figura 9.3 O grafo está formado pelos vértices A, B, C, D e E, e pelas arestas 1, 2, 3, 4, 5 e 6.

Alguns exemplos reais de grafos

Rede integrada de transporte de Curitiba Rede de comunicações por microonda no Brasil Rede de distribuidores de alguma coisa Rede de transportes da VARIG/VASP etc.

 ©88-08, Pedro Kantek

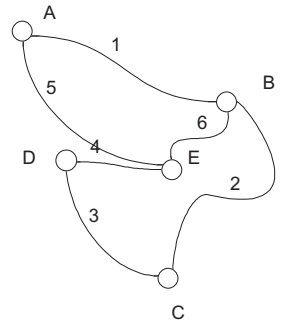


Figura 9.3: Um exemplo de grafo

9.2.2 Definição formal

Formalmente falando, um grafo G é uma tripla $G = (V, A, F)$, onde V = conjunto de vértices do grafo; A = conjunto de arestas do grafo e F = função que relaciona arestas e vértices ($F : A \Longrightarrow V \times V$). O desenho de um gráfico é apenas um instrumento auxiliar (embora muito poderoso) para visualizar as conexões.

Seja o exemplo
$G_1 = (\{A, B, C, D\}, \{1, 2, 3, 4, 6\}, \{1 \rightarrow A \times B, 2 \rightarrow B \times C, 3 \rightarrow B \times D, 4 \rightarrow B \times A, 5 \rightarrow C \times D, 6 \rightarrow D \times A\})$

G_1 seria representado como visto na figura 9.4

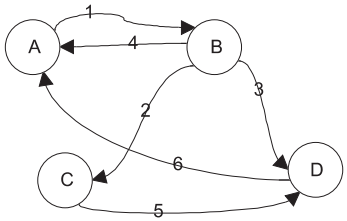


Figura 9.4: O grafo G1

Representações alternativas para o mesmo grafo:
$G_1 = (\{A, B, C, D\}, \{1, 2, 3, 4, 6\},$
$\{1 = (A, B), 2 = (B, C), 3 = (B, D), 4 = (B, A), 5 = (C, D), 6 = (D, A)\})$
ou $G_1 = (\{A, B, C, D\}, \{1, 2, 3, 4, 6\},$
$\{(1, A, B), (2, B, C), (3, B, D), (4, B, A), (5, C, D), (6, D, A)\})$

EXERCÍCIO 118 Faça a representação gráfica do grafo G_2
$G_2 = (\{A, X, Y\}, \{a, b, c, d, e, f\},$
$\{(a, A, X), (b, A, Y), (c, X, Y), (d, Y, X), (e, Y, A), (f, X, A)\})$

EXERCÍCIO 119 Faça a representação gráfica do grafo G_3

©88-08, Pedro Kantek

$G_3 = (\{M, N, O, P, Q, R, S\}, \{abcdef\},$
$\{(a, M, R), (b, R, S), (c, O, P), (d, S, R), (e, S, M), (f, Q, M)\})$

EXERCÍCIO 120 Faça a representação gráfica do grafo G_4
$G_4 = (\{N1, N2, N3\}, \{A1, A2, A5, A8, A9\},$
$\{(A1, N1, N3), (A2, N3, N1), (A5, N3, N2), (A8, N2, N3), (A9, N2, N3)\})$

EXERCÍCIO 121 Faça a representação gráfica do grafo G_5
$G_5 = (\{a, b, c, d\}, \{f, g, h, i\}, \{(f, a, d), (g, a, c), (h, b, c), (i, b, d)\})$

9.2.3 Alguns conceitos

A seguir uma descrição dos principais conceitos associados a grafos. Ressalte-se que não existe uma terminologia universal para o tema, sendo a lista a seguir uma compilação das principais notações utilizadas pelos diversos autores.

- Define-se ordem do grafo G, ao cardinal |V|, que nada mais é que o número de vértices do grafo.
- Uma aresta pode ser definida por um par de vértices, que são conhecidos como extremidades da aresta e são conhecidos como vértices adjacentes
- Um grafo é dito dirigido (ou digrafo) se suas arestas possuem orientação e nesse caso a aresta receberá o nome de arco. Visualmente falando, a orientação é representada por uma seta no destino da aresta.
- Se o grafo não for dirigido ele é dito não dirigido (!). Nesse caso, a representação da aresta ligando os nodos a e b poderá ser representada como (a,b) ou (b,a) indistintamente.
- Definem-se dois arcos como incidentes, quando eles incidem sobre o mesmo vértice. (?)
- Define-se duas arestas como adjacentes quando elas tem um vértice em comum.
- Define-se origem, antecessor ou raiz de um arco ao primeiro vértice no par ordenado que define o arco.
- Define-se destino, sucessor ou extremidade de um arco ao segundo vértice no par ordenado que o define.
- Diz-se que dois arcos ou arestas são paralelos, quando ambos tem mesma origem e mesmo destino.
- Define-se um arco ou aresta laço quando os dois vértices do seu par são iguais.
- Define-se um grafo valorado quando suas linhas possuem um atributo de valor associado.
- Se o grafo não possui laços ou arestas paralelas ele é denominado simples. Se as possui, diz-se que o grafo é multigrafo.
- Um grafo completo é aquele que é simples e no qual cada par de vértices distintos é adjacente.
- Define-se o grafo $\bar{G}$ complementar ao grafo G se eles possuem a mesma ordem e se uma aresta (a,b) $\in G$ implica que (a,b) $\notin \bar{G}$.

 ©88-08, Pedro Kantek

- Define-se grafo bipartite G, quando $G = (\{V_1 \cup V_2\}, E)$, onde $V_1 \cap V_2 = \phi$, e toda aresta (a,b) $\in$ E, tem a característica de a $\in V_1$ e b $\in V_2$.
- Um grafo $G' = (V', A', F')$ é chamado sub-grafo de $G = (V, A, F)$, se V' é subconjunto de V e A' é subconjunto de A.
- Um grafo $G' = (V', A', F')$ é chamado grafo parcial de $G = (V, A, F)$, se $V' \equiv V$ e A' é subconjunto de A.
- Define-se grau de um vértice $v \in V$, o número de vértices adjacentes a V.
- Um grafo é dito regular de grau k, se todos os seus vértices possuem grau k.
- Um grafo regular de grau zero é dito grafo nulo.
- Um vértice que não possui aresta incidente (vértice grau 0) é dito isolado.
- Um vértice que só possui uma única aresta incidente (vértice de grau 1) é dito pendente.
- Dois vértices a e b são ditos adjacentes quando a linha (a, b) existe no grafo.
- Uma seqüência de vértices adjacentes $v_1 ... v_k$ é dita caminho de v_1 até v_k. Esse caminho é formado por $k - 1$ arestas. O valor $k - 1$ é o comprimento do caminho.
- Dado um caminho, se seus vértices forem todos diferentes, o caminho é chamado de caminho elementar.
- Dado um caminho, se todas as suas arestas forem diferentes o caminho denomina-se trajeto.
- Um ciclo é um caminho que tem início e final no mesmo vértice. Se este caminho for elementar, o ciclo é dito elementar.
- Um grafo que não contenha ciclos simples é chamado acíclico.
- Um caminho que contenha cada vértice do grafo uma vez, é chamado hamiltoniano.
- Um caminho que contenha cada aresta do grafo uma vez, é chamado euleriano.
- Um clique de um grafo é um subgrafo desse grafo que é completo.
- Uma árvore é um grafo que não tem ciclos e é conexo.

Desafio

- Descreva 5 grafos que se encontram no nosso dia a dia.

9.2.4 Representação de grafos

Além da representação gráfica, que por motivos óbvios não se presta a uso em algoritmos computacionais, existem os seguintes modelos de representação

©88-08, Pedro Kantek

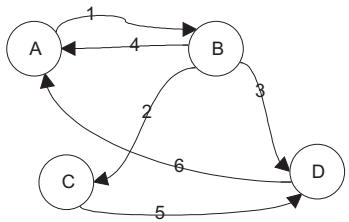


Figura 9.5: Um exemplo de grafo

Matriz de adjacência

Dado um grafo $G = (V, A, F)$, a matriz de adjacência correspondente é a matriz M que tem n linhas por n colunas (onde n é a ordem do grafo, i. é, o número de vértices do grafo. O elemento m_{jk} tem valor zero se v_j e v_k não são adjacentes e tem valor 1 se eles o forem.

Exemplo: seja o grafo dado na figura 9.5

A matriz de adjacência correspondente ao grafo 9.5 é:

0 1 0 0
1 0 1 1
0 0 0 1
1 0 0 0

Que na verdade deve ser interpretada como segue:

	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	0	0	1
D	1	0	0	0

Se o grafo for não orientado, a matriz é simétrica, portanto é suficiente armazenar um dos triângulos (o superior ou o inferior).

Matriz de custo

É uma extensão da matriz de adjacência, usada para grafos valorados. Agora os valores não são apenas 0 e 1, mas sim o valor constante no arco.

Seja o exemplo, representando uma parte da malha rodoviária do sul do País (distâncias fictícias) na figura 9.6

A matriz de adjacência correspondente ao grafo 9.6 é:

	CMP	SPO	CWB	PGR	FLO	BLU	LAG	POA
CMP	0	105	0	380	0	0	0	0
SPO	105	0	400	0	0	0	0	0
CWB	0	400	0	100	290	250	490	0
PGR	380	0	100	0	0	0	0	0
FLO	0	0	290	0	0	135	0	500
BLU	0	0	250	0	135	0	0	0
LAG	0	0	490	0	0	0	0	220
POA	0	0	0	0	500	0	220	0

 ©88-08, Pedro Kantek

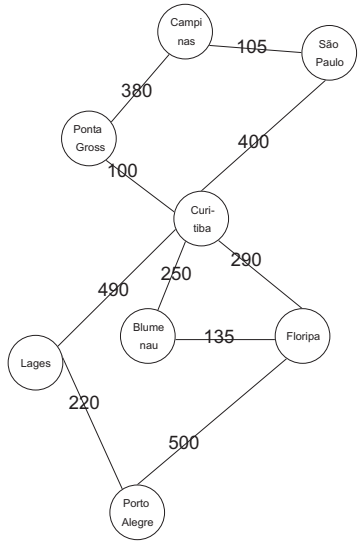


Figura 9.6: Um grafo com distância geográficas

Listas de arestas

Para grafos esparsos (com muitos nodos e poucas arestas), pode-se implementar a representação usando duas listas. A primeira contém os vértices antecessores das arestas e a segunda os sucessores dessas arestas.

Exemplo. Seja novamente o grafo 9.5. Ele poderia ser representado por 2 listas, contendo cada uma 6 valores, e cada valor representando uma aresta, a saber:

A	B	B	B	C	D

e

B	C	D	A	D	A

Matriz de incidência

Dado um grafo $G = (V, A, F)$, a matriz de incidência de G, M, é uma matriz de n linhas (n é o número de vértices) por p colunas (p é o número de arestas). Se o grafo for não dirigido, podem-se usar os valores 1 (se o vértice da linha é extremidade da aresta da coluna) e zero (se não for, ou se esta aresta for um laço). Antevendo a pergunta "Por que usar matriz de incidência se se tem o conceito de adjacência ?"A resposta é simples: como representar 3 arestas entre 2 nodos ? Veja, por exemplo em 9.7

Se o grafo for dirigido, os valores poderão ser 1 (origem neste ponto), 0 (vértice não adjacente ou laço) e -1 (destino neste ponto).

Exercícios 32 (adaptar de Rab92, pag 15)

©88-08, Pedro Kantek

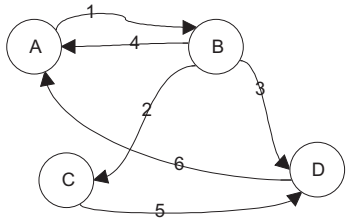


Figura 9.7: Um grafo com 3 arestas entre dois nodos

9.2.5 Conexidade

Um grafo é chamado conexo quando existe um caminho entre cada par de vértices do grafo. Repare que para desenhar a representação geométrica de um grafo desconexo será necessário levantar a caneta do papel. Quando um grafo não for conexo, ele será chamado desconexo. Particularmente, se ele não tiver arestas, será chamado de totalmente desconexo.

EXERCÍCIO RESOLVIDO 8 Dado um grafo não dirigido G_1, definido através de sua matriz de incidência, na qual usou-se o valor 1 para indicar extremidades, zero para vértice não é extremidade e 2 para indicar um laço.

1	0	0	0	0	0	0	0	0	0	0	1
1	1	0	0	0	0	0	1	1	0	0	0
0	1	1	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	1	1	0	1	1	0
0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1	0
0	0	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	0	1

Responder às seguintes perguntas:

1. Quantos vértices tem o grafo ?
   Resposta: 8, já que sua matriz de incidência tem 8 linhas

2. Quantas arestas tem o grafo ?
   R: 12, pois são 12 colunas na matriz.

3. Qual o grau de cada vértice ?
   R: Basta somar a matriz na horizontal, e obter-se-ão 8 valores, correspondentes a quantidade de arestas em cada um dos 8 vértices. O resultado é 2, 4, 3, 5, 2, 2, 3 e 3.

4. Quantas arestas adjacentes possui a aresta número 5 ?
   R: Para responder a esta pergunta, deve-se procurar na matriz, qual a coluna correspondente à aresta. Nessa coluna, estão marcados os 2 vértices que a definem. Tomando as linhas referentes a esses 2 vértices, deve-se somar a quantidade de 1, e subtrair 2 ao final da soma, que correspondem aos dois vértices da aresta 5. A coluna 5, está marcada nas linhas 6 e 7. A linha 6 tem 2 marcações, e a linha 7 tem 3 marcações. A soma é 5, e desse total devem-se retirar os 2 correspondentes à aresta 5. O resultado final é 3 arestas adjacentes.

 ©88-08, Pedro Kantek

5. Quais são elas ?
   R: Deve-se olhar nas linhas 6 e 7, quais arestas estão marcadas, excluindo-se a 5. Portanto, são as arestas 3, 4 e 11.

6. Existem arestas paralelas no grafo ?
   R: Para responder, precisa-se examinar todas as colunas, e verificar se algum par, tem as mesmas marcações. A resposta é não.

7. Existem laços ?
   R: Pela convenção usada, basta procurar pela existência da marcação 2 na matriz. Como não há, a resposta é não.

8. O grafo é simples ou é multigrafo ?
   Já que não existem arestas paralelas ou laços, o grafo é simples.

9. O grafo é completo ?
   R: Para ser completo, ele precisa ser simples (e é), e também cada vértice precisa ser adjacente a todos os demais. Isso significa que se existem 8 vértices, a soma de marcações de cada linha deve ser igual a 7. Como não é, a resposta é não, o grafo não é completo.

10. Existem vértices isolados ?
    R: O vértice isolado é caracterizado pela sua linha não ter marcação. Como não é o caso, a resposta é não existe.

11. Existe caminho entre o vértice 2 e o vértice 7 ?
    R: Saindo do vértice 2 (segunda linha), deve-se mudar de vértice (linha) sempre que houver marcação. O destino da mudança é a outra marcação da aresta (coluna). O processo prossegue até que se alcance o vértice (linha) 7. Existem vários caminhos, a saber: 2, 3, 7; ou 2, 4, 3, 7; ou 2, 8, 4, 5, 7; ou...

12. Os três caminhos acima são elementares ?
    R: Sim, pois não há repetição de vértices dentro de cada caminho.

13. São trajetos ?
    R: Sim, pois não há repetição de arestas

EXERCÍCIO RESOLVIDO 9 Dado o grafo não dirigido G_2, pela sua matriz de adjacência, onde os laços estão representados por 1 na diagonal principal, um número qualquer indica a quantidade de adjacências, e zero não adjacência, responder:

0	1	0	0	0	0	0	1
1	0	1	1	0	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	1	1	0	1
0	0	0	1	0	0	1	0
0	0	0	1	0	0	1	0
0	0	1	0	1	1	0	0
1	1	0	1	0	0	0	0

1. Quantos vértices tem G_2 ?
   R: Tem 8, já que sua matriz de adjacência é 8 x 8.

2. Quantas arestas tem G_2 ?
   R: Tem 12, que é a quantidade de marcações acima (ou abaixo) da diagonal principal.

©88-08, Pedro Kantek

3. Qual o grau de cada vértice ?
   R: Basta somar a matriz na horizontal, (ou na vertical) e obter-se-ão 8 valores, correspondentes a quantidade de arestas em cada um dos 8 vértices. O resultado é 2, 4, 3, 5, 2, 2, 3 e 3.

4. Quantas arestas adjacentes possui a aresta número 5 ?
   R: Para responder a esta pergunta, deve-se estabelecer uma maneira de contar as arestas. Imaginemos que cada marcação acima da diagonal principal será numerada. Com isso, a aresta 5, é a aresta que liga os vértices 2 e 8. (linha 2 e coluna 8). Para responder a pergunta, precisamos somar as marcações dessa linha e dessa coluna e subtrair 2. A linha 2 tem 4 marcações, e a coluna 8 tem 3 marcações (total = 7). Como devemos excluir a marcação original, que foi somada 2 vezes, o resultado é 5.

5. Quais são elas ?
   R: Deve-se olhar nas linhas 2 e coluna 8, e identificar os pares de vértices que definem as arestas adjacentes. Na linha 2, com o vértice 1, 3 e 4, que representam as arestas (2,1), (2,3) e (2,4). Na coluna 8, há marcações em 1 e em 4, que representam as arestas (8,1) e (8,4).

6. Existem arestas paralelas no grafo ?
   R: Para responder, precisa-se examinar a parte superior (ou inferior) à diagonal superior e verificar se existe algum valor superior a 1. A resposta é não.

7. Existem laços ?
   R: Pela convenção usada, basta procurar pela existência da marcação 1 na diagonal principal. Como não há, a resposta é não.

8. O grafo é simples ou é multigrafo ?
   Já que não existem arestas paralelas ou laços, o grafo é simples.

9. O grafo é completo ?
   R: Além de simples, para ser completo, o grafo deveria ter todos os vértices adjacentes. Assim, exceto a diagonal principal, não se admite marcação zero acima (ou abaixo) da diagonal principal. Como há zeros, a resposta é não.

10. Existem vértices isolados ?
    R: O vértice isolado é caracterizado pela sua linha (ou coluna) não ter marcação. Como não é o caso, a resposta é não existe.

9.2.6 Obter a matriz de adjacência a partir da matriz de incidência

Seja o problema de dada uma matriz de incidência obter a de adjacência para um dado grafo simples

```
1: n ← número de linhas de I
2: m ← número de colunas de I
3: A ← matriz n x n contendo zeros
4: para i de 1 até m faça
5:    v1 ← 0
6:    v2 ← 0
7:    para j de 1 até n faça
8:       se I[j;i] ≠ 0 ∧ v1 = 0 então
9:          v1 ← j
10:      senão
11:         v2 ← j
12:      fimse
13:   fimpara
14:   A[v1;v2] ← A[v2;v1] ← 1
15: fimpara
```

Seja o problema inverso: Dada a matriz A de adjacência de um grafo simples, obter a matriz I de incidência

```
1: n ← número de linhas de A
2: qtdcols ← 0
3: para i de 1 até n faça
4:    para j de 1 até n faça
5:       se i < j então
6:          se se A[i;j] ≠ 0 então
7:             qtdcols ← qtdcols + 1
8:          fimse
9:       fimse
10:   fimpara
11: fimpara
12: I ← matriz n x qtdcols contendo zeros
13: coluna ← 1
14: para i de 1 até n faça
15:    para j de 1 até n faça
16:       se i < j então
17:          se A[i;j] ≠ 0 então
18:             I[coluna;i] ← I[coluna;j] ← 1
19:             coluna ← coluna + 1
20:          fimse
21:       fimse
22:   fimpara
23: fimpara
```

9.2.7 Fechamento transitivo

A propriedade do fechamento transitivo garante que todos os nodos são acessíveis a partir de qualquer nodo do gráfico. Vide mais em pág. 672 de [Tene95]

9.2.8 Caminhamento em grafos

Trata-se agora de obter um algoritmo que permita percorrer todos os nodos de um grafo conexo.

Caminhamento em profundidade

Dado um grafo por sua matriz de adjacência, e partindo de um vértice qualquer, o algoritmo deve ser capaz de produzir uma lista de vértices percorridos. A estratégia deste algoritmo, é examinar todos os vértices adjacentes ao vértice que está sendo examinado, colocá-los em uma pilha, e seguir ao primeiro vértice adjacente. É o mesmo princípio do caminhamento em profundidade para as árvores.

Usar-se-á um vetor auxiliar para identificar o estado de cada um dos vértices do grafo. Os valores admitidos nesse vetor são 3. O valor 1 indicará que o vértice ainda não foi pesquisado. O valor 2, indicará que o vértice se encontra na pilha para ser processado, e o valor 3 indicará que o vértice já foi visitado.

0	0	1	0	0	0	1	0	1	1	1
0	0	0	1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	1	0
0	1	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0	1
0	0	0	0	1	0	0	1	0	0	1
1	0	0	1	0	0	0	0	1	0	1
0	1	0	1	0	1	0	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0
1	0	1	0	0	0	0	0	1	0	0
1	0	0	0	1	1	1	0	0	0	0

Entrada: ADJ é a matriz de adjacência de um grafo não direcionado e conexo
Entrada: P1 é uma pilha de vértices
Entrada: FLAG é o vetor auxiliar de estado dos vértices
Entrada: INI é o número do vértice inicial
1: FLAG ← 1 {no início todos os vértices tem estado = 1}
2: P1 ← empilha INI {empilha o vértice inicial}
3: FLAG[INI] ← 2 {o vértice inicial passa a estado = 2}
4: **enquanto** não VAZIA(P1) **faça**
5: X ← desempilha P1
6: imprime (X)
7: FLAG[X] ← 3
8: vizinhos ← nodos adjacentes a X e que tenham estado = 1
9: FLAG[vizinhos] ← 2
10: empilha vizinhos
11: **fimenquanto**

EXERCÍCIO RESOLVIDO 10 Seja o grafo a seguir, através de sua matriz de adjacência:

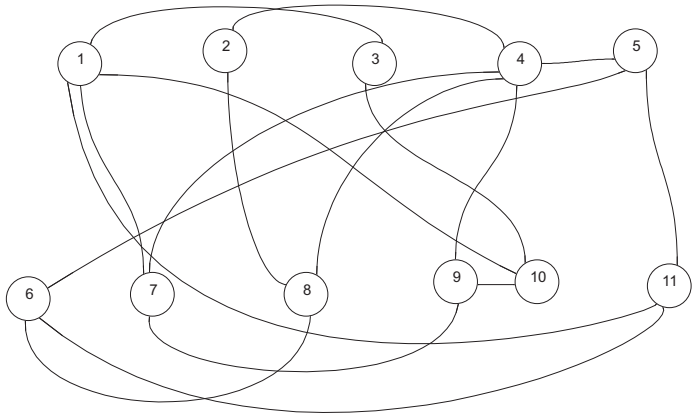


Figura 9.8: Desenho do grafo cuja matriz de adjacência é a 10

Desenhando este grafo, para ter uma idéia de sua distribuição (a matriz pode ser ótima para os computadores, mas é medonha para olhos e cérebro humano, não é?) tem-se 9.8.

Ache a seqüência de nodos visitados no caminhamento em profundidade, e construa a árvore geradora deste caminhamento. Nodo inicial 5.

- Como o problema pede, o nodo inicial é o 5, fica-se FLAG:

1	1	1	1	2	1	1	1	1	1	1

PILHA: 5

Como a PILHA não está vazia, desempilha-se o 5, ele é impresso
SAÍDA: 5
seu flag recebe 3
FLAG:

1	1	1	1	3	1	1	1	1	1	1

e vizinhos recebe 11, 6 e 4, que são empilhados
PILHA: 11, 6, 4
seus flags ficam
FLAG:

1	1	1	2	3	2	1	1	1	1	2

- Como a pilha não está vazia, desempilha-se o 4, ele é impresso
SAÍDA: 5, 4
seu flag recebe 3
FLAG:

1	1	1	3	3	2	1	1	1	1	2

seus vizinhos 9, 8, 7 e 2, que são empilhados
PILHA: 11, 6, 9, 8, 7, 2
e seus flags ficam
FLAG:

1	2	1	3	3	2	2	2	2	1	2

- Como a pilha não está vazia, desempilha-se o 2, ele é impresso
SAÍDA: 5, 4, 2
PILHA: 11, 6, 9, 8, 7
seu flag recebe 3
FLAG:

1	3	1	3	3	2	2	2	2	1	2

2 não tem vizinhos não marcados, logo

- Como a pilha não está vazia, desempilha-se o 7, ele é impresso
SAÍDA: 5, 4, 2, 7
PILHA: 11, 6, 9, 8
seu flag recebe 3
FLAG:

1	3	1	3	3	2	3	2	2	1	2

seu vizinho é 1, que é empilhado
PILHA: 11, 6, 9, 8, 1
os flags ficam
FLAG:

2	3	1	3	3	2	3	2	2	1	2

- Como a pilha não está vazia, desempilha-se o 1, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1
  seu flag recebe 3
  FLAG:

3	3	1	3	3	2	3	2	2	1	2

  seus vizinhos 10, 3, são empilhados
  PILHA: 11, 6, 9, 8, 10, 3
  e seus flags ficam
  FLAG:

3	3	2	3	3	2	3	2	2	2	2

- como a pilha não está vazia, desempilha-se o 3, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1, 3
  PILHA: 11, 6, 9, 8, 10
  seu flag recebe 3
  FLAG:

3	3	3	3	3	2	3	2	2	2	2

  3 não tem vizinhos não marcados, logo

- desempilha-se o próximo, que é o 10, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1, 3, 10
  o flag fica

3	3	3	3	3	2	3	2	2	3	2

  10 não tem vizinhos, logo

- desempilha-se o próximo, que é o 8, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1, 3, 8
  PILHA: 11, 6, 9
  o flag fica

3	3	3	3	3	2	3	3	2	3	2

  8 não tem vizinhos não marcados, logo

- desempilha-se o próximo, que é o 9, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1, 3, 8, 9
  PILHA: 11, 6
  o flag fica

3	3	3	3	3	2	3	3	3	3	2

  9 não tem vizinhos, logo

- desempilha-se o próximo, que é o 6, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1, 3, 8, 9, 6
  PILHA: 11
  o flag fica

3	3	3	3	3	3	3	3	3	3	2

  6 não tem vizinhos, logo

- desempilha-se o próximo que é o 11, ele é impresso
  SAÍDA: 5, 4, 2, 7, 1, 3, 8, 9, 6, 11

 ©88-08, Pedro Kantek

  PILHA: vazia
  o flag fica

3	3	3	3	3	3	3	3	3	3	3

  Como a pilha está vazia, encerra-se o algoritmo.

A árvore de busca gerada neste exercício está mostrada na figura 9.9

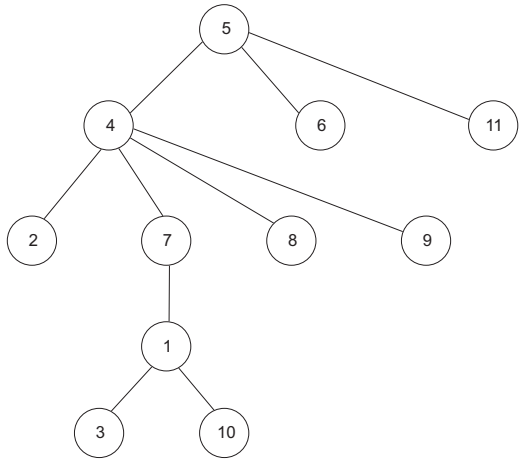


Figura 9.9: Árvore de busca

Resposta: A seqüência de vértices pesquisados na busca em profundidade é 5 4 2 7 1 3 10 8 9 6 11

Pesquisa em largura

O algoritmo para busca em largura é:

Entrada: ADJ é a matriz de adjacência de um grafo não direcionado e conexo
Entrada: F1 é uma fila de vértices
Entrada: FLAG é o vetor auxiliar de estado dos vértices
Entrada: INI é o número do vértice inicial

```
1: FLAG ← 1 {no início todos os vértices tem estado = 1}
2: F1 ← enfileira INI {enfileira o vértice inicial}
3: FLAG[INI] ← 2 {o vértice inicial passa a estado = 2}
4: enquanto VAZIA(F1) faça
5:    X ← desenfileira F1
6:    imprime (X)
7:    FLAG[X] ← 3
8:    vizinhos ← nodos adjacentes a X e que tenham estado = 1
9:    FLAG[vizinhos] ← 2
10:   enfileira vizinhos
11: fimenquanto
```

©88-08, Pedro Kantek

EXERCÍCIO RESOLVIDO 11 Seja o mesmo grafo da pesquisa em profundidade. Queremos agora, obter a seqüência de visitação dos vértices em uma pesquisa em largura. Nodo inicial é o 5

A matriz de adjacência é:

0	0	1	0	0	0	1	0	1	1	1
0	0	0	1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	1	0
0	1	0	0	1	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0	1
0	0	0	0	1	0	0	1	0	0	1
1	0	0	1	0	0	0	0	1	0	1
0	1	0	1	0	1	0	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0
1	0	1	0	0	0	0	0	1	0	0
1	0	0	0	1	1	1	0	0	0	0

FILA é a fila de nodos a processar
FLAG é o vetor de 11 marcadores

- INI é 5
  FLAG é

1	1	1	1	1	1	1	1	1	1	1

  FILA é 5
  FLAG é

1	1	1	1	2	1	1	1	1	1	1

- Como a fila não está vazia, desenfila-se o 5,
  Imprime-se o 5, e
  SAÍDA: 5
  FLAG é

1	1	1	1	3	1	1	1	1	1	1

  os vizinhos de 5 são: 11, 6 e 4.
  FLAG é

1	1	1	2	3	2	1	1	1	1	2	FILA fica: 11, 6, 4

- Como a fila não está vazia, desenfila-se o 4,
  Imprime-se o 4, e
  SAÍDA: 5, 4
  FLAG é

1	1	1	3	3	2	1	1	1	1	2

  FILA: 11, 6
  os vizinhos de 4 são 9, 8, 7 e 2
  FLAG é

1	2	1	3	3	2	2	2	2	1	2

  FILA: 9, 8, 7, 2, 11, 6

- Como a fila não está vazia, desenfila-se o 6
  Imprime-se o 6, e
  SAÍDA: 5, 4, 6
  FLAG é

1	2	1	3	3	3	2	2	2	1	2

  FILA: 9, 8, 7, 2, 11

©88-08, Pedro Kantek

6 não tem vizinhos, logo desenfila-se o 11
Imprime-se o 11, e
SAÍDA: 5, 4, 6, 11
FLAG é

1	2	1	3	3	3	2	2	2	1	3

FILA: 9, 8, 7, 2
O 11 tem como vizinho o 1
FLAG é

2	2	1	3	3	3	2	2	2	1	3

Enfileira-se o vizinho
FILA: 1, 9, 8, 7, 2

- Como a fila não está vazia, desenfila-se o 2
  imprime-se o 2, e
  SAÍDA: 5, 4, 6, 11, 2
  FLAG é

2	3	1	3	3	3	2	2	2	1	3

  FILA: 1, 9, 8, 7

- 2 não tem vizinhos, logo, desenfila-se o 7
  imprime-se o 7, e
  SAÍDA: 5, 4, 6, 11, 2, 7
  FLAG é

2	3	1	3	3	3	3	2	2	1	3

  FILA: 1, 9, 8

- O 7 não tem vizinhos, logo, desenfila-se o 8
  imprime-se o 8, e
  SAÍDA: 5, 4, 6, 11, 2, 7, 8
  FLAG é

2	3	1	3	3	3	3	3	2	1	3

  FILA: 1, 9
  item O 8 não tem vizinhos, logo, desenfila-se o 9
  imprime-se o 9, e
  SAÍDA: 5, 4, 6, 11, 2, 7, 8, 9
  FLAG é

2	3	1	3	3	3	3	3	3	1	3

  FILA: 1
  O 9 tem como vizinho o 10, logo, enfila-se o 10
  FILA: 10, 1,
  FLAG é

2	3	1	3	3	3	3	3	3	2	3

- Como a fila não está vazia, desenfila-se o 1
  imprime-se o 1, e
  SAÍDA: 5, 4, 6, 11, 2, 7, 8, 9, 1
  FLAG é

3	3	1	3	3	3	3	3	3	2	3

  FILA: 10
  O 1 tem como vizinho o 3, enfila-se o 3
  FILA: 3,10
  FLAG é

©88-08, Pedro Kantek

3	3	2	3	3	3	3	3	3	2	3

- Como a fila não está vazia, desenfila-se o 10
  imprime-se o 10, e
  SAÍDA: 5, 4, 6, 11, 2, 7, 8, 9, 1, 10
  FLAG é

3	3	2	3	3	3	3	3	3	3	3

  FILA: 3

- O 10 não tem vizinhos, e desenfila-se o 3
  imprime-se o 3, e
  SAÍDA: 5, 4, 6, 11, 2, 7, 8, 9, 1, 10, 3
  FLAG é

3	3	3	3	3	3	3	3	3	3	3

  FILA: vazia
  Como a fila está vazia, encerra-se o algoritmo.

  A resposta é 5, 4, 6, 11, 2, 7, 8, 9, 1, 10, 3

A árvore gerada, em extensão é está na figura 9.10. Note-se que a árvore tem altura = 3, em contraposição à arvore da pesquisa em profundidade que tem altura = 4.

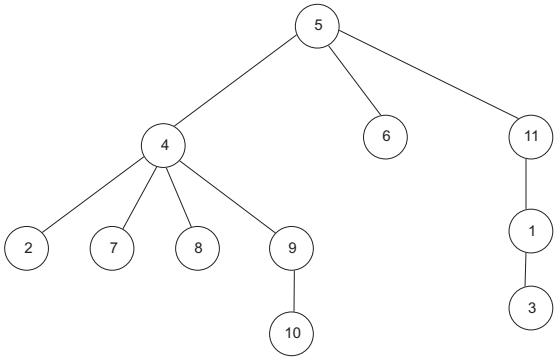


Figura 9.10: Busca em largura

DICA: seja empilhando ou enfileirando, faça-o de tal maneira que para um conjunto empilhado (enfileirado) junto, saia o menor número primeiro.

EXERCÍCIO 122 Escreva uma função que leia uma matriz de adjacência (quadrada), um nodo inicial, e o valor k, e imprima o k-ésimo nodo, obtido num processo de caminhamento em profundidade

EXERCÍCIO 123 Escreva uma função que leia uma matriz de adjacência (quadrada), um nodo inicial, e o valor k, e imprima o k-ésimo nodo, obtido num processo de caminhamento em extensão

EXERCÍCIO 124 Escreva uma matriz de incidência qualquer para um grafo de x nodos, conexo, não direcionado e que contenha y arestas.

EXERCÍCIO 125 Dada uma matriz de incidência, definida como int MAT [y] [y] , escreva um algoritmo que leia a matriz e imprima 1 se a mesma representar um grafo conexo e 0 senão.

EXERCÍCIO 126 Desenhe um grafo com x nodos e y arestas e escreva a sua matriz de adjacência. Os nodos devem ser identificados por letras maiúsculas, a partir de "A", e as arestas devem ser numeradas a partir de 1.

EXERCÍCIO 127 Escreva um algoritmo que leia um grafo não direcionado, sem laços e sem arestas paralelas (através de sua matriz de incidência) de 14 vértices e imprima o número dos vértices que são isolados Utilize a seguinte estrutura: estrutura inteiro M[14][14] GRAF

EXERCÍCIO 128 Usando a mesma estrutura da questão anterior, escreva um algoritmo que leia um grafo direcionado, contendo paralelas e laços e imprima para os pares de vértices vizinhos, qual o menor trajeto entre as duas arestas paralelas. Obviamente, se só houver uma arestas, esta é o menor caminho.

EXERCÍCIO 129 Questão: Porque se deve usar o conceito de hashing ao acessar uma tabela Escreva um algoritmo que leia um grafo não direcionado, sem laços e sem arestas paralelas (através de sua matriz de incidência) de 21 vértices e imprima "SIM"caso o grafo seja completo (isto é, cada vértice é vizinho de todos os demais). Utilize a seguinte estrutura: estrutura inteiro M[21][21] GRAF

EXERCÍCIO 130 Seja X o seu número da chamada. Calcule k (6 + X mod 5 e M ((3 + (X mod 3)) div 2. Desenhe um grafo qualquer que tenha K nodos e M arestas. Identifique os nodos por letras e as arestas por números.

EXERCÍCIO 131 Suponhamos que o que caracteriza a matriz de adjacência de um grafo não dirigido é a presença de um eixo de simetria ao longo da diagonal principal da matriz. Se assim for, escreva um algoritmo que leia uma matriz de adjacência de um grafo de ordem 84 e imprima NÃO se ele for não dirigido e SIM em caso contrário.

EXERCÍCIO 132 Escreva um algoritmo que leia uma matriz de adjacência referente a um grafo dirigido de ordem 127 e gere um relatório impresso, onde para cada aresta, é impressa uma quádrupla formado por: (número-aresta, nodo-origem, nodo-destino, valor-aresta). Tanto nodos como arestas iniciam sua contagem em zero. O zero na matriz de adjacência deve ser interpretado como custo infinito (não há aresta).

EXERCÍCIO 133 Escreva um algoritmo que leia uma matriz de adjacência referente a um grafo dirigido de ordem 701, sem laços. Feito isto, deve ler 1 nodo (estes são numerados de 0 a 700). Finalmente, deve imprimir 2 listas. A primeira é a das arestas que tem este nodo como origem e a segunda a das arestas que tem o nodo como destino. As arestas devem ser impressas na forma do par (origem,destino). O zero na matriz de adjacência deve ser interpretado como custo infinito (não há aresta).

```
Exemplo. Suponha-se a leitura da matriz

0  0  4  3
1  0  4  2
0  3  0  0
0  2  1  0

Se for lido o nodo 1, devem ser impressas as listas ((1,0),(1,2)) e ((2,1),(3,1))
```

EXERCÍCIO 134 Escreva um algoritmo que leia uma matriz de adjacência referente a um grafo dirigido de ordem 538, sem laços. O algoritmo deve imprimir as arestas (na forma origem,destino) que tenham valor (custo) maior do que 10 unidades.

```
Exemplo. Suponha-se a leitura da matriz

0  8 13  4
3  0  5 11
0 10  0 14
0  1  2  0

A resposta deve ser (0,2), (1,4) e (2,4)
```

EXERCÍCIO 135 Dado um grafo orientado no qual existem determinadas arestas, denomina-se a aresta "subida"se ela for do nodo i para o nodo j e i $<$ j. Identicamente denomina-se a aresta de "descida"se ela vai do nodo i para o nodo j e i $>$ j. Note-se que não está definido nenhum nome para os laços pois nestes, tem-se i (j. O zero na matriz de adjacência deve ser interpretado como custo infinito (não há aresta). Escreva um algoritmo que leia a matriz de adjacência de um grafo de 207 nodos e um número de nodo, e depois imprima a lista de subidas e a de descidas que incluem este nodo.

```
Exemplo. Suponha-se a leitura da matriz

0  2  0  5
3  0  1  0
0 10  0  6
0  0  0  0

Se o nodo lido for o "2", a resposta deve ser subidas: (2,3) e (1,2) e descida: (2,1)
```

EXERCÍCIO 136 Define-se o "caminho vai e vem"de um nodo K em uma matriz de adjacência que representa um grafo orientado como aquele formado por 2 arestas, de K até X e de X até K. X varia entre 0 e a ordem da matriz menos um, exceto o valor de K. O caminho só existe se existirem as duas arestas.

```
Seja por exemplo, o seguinte grafo

0  1  3  2
4  0  3  1
2  0  0  4
0  2  2  0

Vamos olhar o "vai e vem" para o nodo 2:
a) de 2 para 0 e de 0 para 2: 2 + 3 = 5
```

 ©88-08, Pedro Kantek


```
b) de 2 para 1 e de 1 para 2: não existe o laço de 2 para 1.
c) de 2 para 3 e de 3 para 2: 4 + 2 = 6
```

EXERCÍCIO 137 Escreva um algoritmo que leia a matriz de adjacência de um grafo de 207 nodos, que contém laços e imprima para os 207 nodos qual o menor valor entre o laço em cada nó e o menor dos caminhos "vai e vem"para este nodo. Lembre-se que zero na matriz significa "não há".

EXERCÍCIO 138 Suponha que um grafo é representado por uma matriz de arestas, de m x 3 números onde m é o número de arestas do grafo e as 3 colunas são:
a) coluna 0: nodo origem da aresta
b) coluna 1: nodo destino da aresta
c) coluna 2: valor da aresta

Escreva um algoritmo que leia uma matriz de arestas de um grafo de ordem 437 e 1000 arestas, gere e imprima a matriz de adjacência do grafo.

EXERCÍCIO 139 Suponha que um grafo é representado por uma matriz de arestas duplas, de m x 4 números onde m é a metade do número de arestas do grafo e as 4 colunas são:
a) coluna 0: nodo A da aresta
b) coluna 1: nodo B da aresta
c) coluna 2: valor da aresta de A para B
d) coluna 3: valor da aresta de B para A

Escreva um algoritmo que leia uma matriz de arestas duplas de um grafo de ordem 288 e 730 arestas, gere e imprima a matriz de adjacência do grafo

EXERCÍCIO 140 Suponha que um grafo é representado por uma matriz de arestas negpos, de m x 3 números onde as 3 colunas são:
a) coluna 0: nodo X
b) coluna 1: nodo Y
c) coluna 2: valor Z
Se Z $>$ 0, esta aresta vai de X a Y com valor Z. Se Z $<$ 0, esta aresta vai de Y a X, com valor -Z. Finalmente se Z (0, esta aresta não existe (ou tem custo infinito, o que é a mesma coisa).

Escreva um algoritmo que leia uma matriz de arestas negpos de um grafo de ordem 300 e matriz de 1000 linhas, gere e imprima a matriz de adjacência do grafo.

EXERCÍCIO 141 Suponha um grafo AMARELO de ordem N. Ele contém laços e é conexo. Os valores das arestas entre os nodos i e j, valem i+j.
a) Escreva a matriz de adjacência de um grafo amarelo de ordem 6
b) Qual o valor da aresta de maior valor no grafo ? (N-1)x2
c) O grafo é não orientado ? (SIM)

EXERCÍCIO 142 Suponha um grafo verde e branco de ordem M. Ele é formado segundo a regra que determina o valor da aresta entre i e j:
a) se (i+j) mod 4 = 0, a aresta não existe
b) se (i+j) mod 4 = 1, a aresta vale i
c) se (i+j) mod 4 = 2, a aresta vale j
d) se (i+j) mod 4 = 3, a aresta vale i+j
Escreva uma matriz de adjacência para um grafo verde e branco de ordem 5

©88-08, Pedro Kantek

EXERCÍCIO 143 Seja um grafo de N nodos e no qual um deles, chamado fornecedor é origem de N-2 arestas que o ligam a todos os demais nodos (exceto ele mesmo e o nodo controle) com custo igual a F. Outro nodo chamado controle (e necessariamente diferente do fornecedor) se liga 2 vezes (ida e volta) ao fornecedor com custo igual a C. Escreva um algoritmo que leia N (ordem do grafo), n1 (fornecedor) e n2 (controle), faça F ← (n1 + n2) + 2 e C ← (n1 + n2) x 3, gere e imprima a matriz de adjacência deste grafo.

EXERCÍCIO 144 Suponha um dominó completo (28 pedras). O problema que se coloca é a de arranjar as pedras juntando metades comuns. Veja um exemplo em 9.11

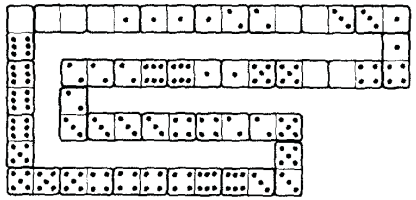


Figura 9.11: Um exemplo de um dominó

O grafo tem 7 vértices e 28 arestas representam os dominós. Qualquer caminho nesse grafo conduzirá a um resultado como este ao lado. Escreva um programa que indique os trajetos [Sch84, pág. 287]

EXERCÍCIO 145 Escreva o algoritmo de uma função que receba uma matriz de adjacência de um grafo de ordem 13 e devolva .V. se o grafo for conexo e .F. senão

EXERCÍCIO 146 Suponha um grafo dirigido K de ordem 17, dado através de sua matriz de adjacência. Dados A e B (2 nodos) escreva uma função que receba A e B e devolva .T. se houver uma ligação de 3 arestas entre A e B.

EXERCÍCIO 147 Escreva uma função que receba uma Matriz de adjacência de um grafo orientado qualquer e também o nodo A e devolva a quantidade de arestas que deixam A

EXERCÍCIO 148 Escreva uma função que receba uma Matriz de adjacência de um grafo orientado qualquer e devolva a quantidade de nodos isolados que há no grafo. (neste caso, isolado = não manda nem recebe aresta).

EXERCÍCIO 149 Definindo a riqueza de um nodo como

$$riq(A) = \sum entrantes(a) - \sum deixantes(A)$$

dado um grafo através de sua matriz de custos, escreva uma função que informe o nodo mais rico.

Desafio

- Escolha 3 exercícios desta aula e implemente-os em computador

 ©88-08, Pedro Kantek

9.2.9 Coloração de grafos

1) coloração de mapas

O problema consiste em colorir o grafo de forma que nenhuma região vizinha tenha a mesma cor.

Quantas cores são necessárias? Resposta: 4, no máximo, para qualquer mapa.

2) problemas de escalonamento

ex.: elaboração do horário de exames escolares

O problema: n disciplinas que devem ter seus exames escalonados de forma a não haver conflitos entre eles, e usando-se o menor número de períodos possível.

O modelo: é associado à coloração de grafos. Cada disciplina é representada por um vértice e o conflito é indicado pela existência de uma aresta entre dois vértices.

A solução: é obtida "colorindo-se"o grafo com o menor número de cores possível.

3) O problema do semáforo x coloração

É um problema comum nas grandes cidades.

Situação real - o sistema

Um cruzamento de avenidas e ruas, umas com mão única, outras permitindo trânsito nos dois sentidos. Define-se quais fluxos de veículos não são permitidos simultaneamente e o objetivo é planejar um semáforo com o menor número de fases possível.

9.3 Algoritmo de Dijskstra

O algoritmo de Floyd que será visto na próxima aula, embora fácil, padece de um defeito grave: tem complexidade $O(n^3)$, pelo que se revela impraticável para instâncias não muito grandes de n ou quando o grafo é dinâmico tendo suas arestas alteração constante de valor.

Mais ainda, em muitos problemas não é necessário saber o menor caminho de todas as origens para todos os destinos, sendo muitas vezes aceitável conhecer apenas todos os menores caminhos de uma única origem para todos os destinos possíveis. O algoritmo que resolve este caso é o seguinte:

Algoritmo DIJSKTRA: menor caminho:

```
1:  // dist é a distância conhecida desde INI até i (onde i ≠ INI)
2:  // perm é um vetor de conhecimento de distâncias
3:  // mínimas permanentes
4:  // (1=sim,0=não)
5:  // n é a número de vértices do grafo
6:  // corrente é o nodo que esta sendo analisado
7:  // INI é o nodo inicial e FIN é o final
8:  // menordist é a menor distância até aqui
9:  // precede é um vetor
10: para i = 0 até n-1 faça
11:    perm [i] ← 0 {0 significa que perm[i] ainda não é conhecida}
12:    dist [i] ← 99999
13: fimpara
14: perm [INI] ← 1 {PERM[INI] é zero (e já é conhecida)}
15: dist [INI] ← 0
16: corrente ← INI
17: enquanto (corrente ≠ FIN) faça
18:    {variação (n != SOMAT(perm)) ***}
19:    menordist ← 9999
20:    dc ← dist[corrente]
```

©88-08, Pedro Kantek

```
21:    para i de 0 até n-1 faça
22:      se perm[i] = 0 então
23:        novadist ← dc + ADJ[corrente,i]
24:        se (novadist < dist [i]  então
25:          dist[i] ← novadist
26:          precede[i] ← corrente
27:        fimse
28:        se (dist[i] < menordist) então
29:          menordist ← dist[i]
30:          k ← i
31:        fimse
32:      fimse
33:    fimpara
34:    corrente ← k
35:    perm[corrente] ← 1
36: fimenquanto
37: devolva (dist[FIN]) // variação devolva(dist) ***
```

Caso se utilize a variação no comando ***, o que se busca não é apenas a distância entre o nodo INI e o nodo FIN, e sim todas as distâncias entre INI e todos os demais vértices. Nota: descrição deste algoritmo em Rabuske: pág 66 e em Tenenbaum: pág. 679. Note-se também que este algoritmo tem complexidade $O(n^2)$.

Um exemplo do algoritmo de Dijkstra. Seja o grafo dado em **??** que tem como matriz

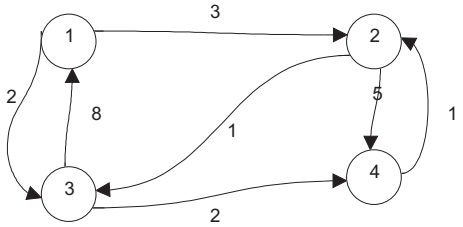


Figura 9.12: Um grafo

Q = 0 3 2 0
0 0 1 5
8 0 0 2
0 1 0 0
Acompanhando o algoritmo
INI ← 1
PERM ← 1 0 0 0
DIST ← 0 ∞ ∞ ∞
CORRENTE ← INI

enquanto 4 ≠ 1
MENOR ← ∞
DC ← 0
I ← 1 ... nada

 ©88-08, Pedro Kantek

I ← 2
NOVA ← 0 + 3
DIST ← 0 3 ∞ ∞
MENOR ← 3
K ← 2
I ← 3
NOVA ← 0 + 2
DIST ← 0, 3, 2, ∞
MENOR ← 2
K ← 3
I ← 4
NOVA ← 0 + ∞
CORRENTE ← 3
PERM ← 1 0 1 0

enquanto 4 ≠ 2
MENOR ← ∞
DC ← 2
I ← 1 ... nada
I ← 2
NOVA ← 2 + ∞
MENOR ← 3
K ← 2
I ← 3 ... nada
I ← 4
NOVA ← 2 + 2
DIST ← 0, 3, 2, 4
CORRENTE ← 2
PERM ← 1, 1, 1, 0

enquanto 4 ≠ 3
MENOR ← ∞
DC ← 3
I ← 1 ... nada
I ← 2 ... nada
I ← 3 ... nada
I ← 4
NOVA ← 3 + 5
MENOR ← 4
K ← 4
CORRENTE ← 4
PERM ← 1, 1, 1, 1
acabou !

Resposta: a distância desde o nodo 1 para os demais é 0, 3, 2 e 4.

Desafio

- Implemente o algoritmo de Dijkstra.

©88-08, Pedro Kantek

9.4 Caminho Mínimo em Grafos

Seja um grafo valorado e direcionado, representado através de sua matriz de adjacência. Como se sabe, a ordem dessa matriz informa a quantidade de vértices do grafo.

A partir da matriz de adjacência, efetuam-se inicialmente duas mudanças:

- a informação "existe caminho do vértice i ao vértice j", que é representada por um "1"na posição i,j da matriz é substituída pelo valor da ligação entre os vértices i e j.
- o valor dos elementos k,k (pertencentes à diagonal principal) indicam o custo de sair de um vértice e chegar nele. Isso pode se dar através de um laço, ou através de um caminho que passe por diversos vértices.
- A inexistência de ligações entre os vértices i e j, que na matriz de adjacência é representada por zero, aqui é representada por ∞.

Com essas modificações, a matriz de adjacência se transforma na matriz K_0. O algoritmo a seguir, prevê o cálculo sucessivo das matriz K_i, até a obtenção da matriz K_n, onde n é a ordem do grafo. Esta última matriz é a de caminhos mínimos. Perceba-se que a obtenção de K_j tem como entrada a matriz K_{j-1}.

Algoritmo Quanto-Menor-Caminho (Alg. de Floyd):

Entrada: matriz de adjacência e transforma-a na de caminhos mínimos

```
1:  inteiro K [n, n]
2:  inteiro infinito 9999999
3:  leia ADJ (de ordem n)
4:  inteiro r,s,t
5:  para r de 0 até n-1 faça
6:     para s de 0 até n-1 faça
7:        se ADJ[r,s] = 0 então
8:           K[r,s] ← ∞
9:        senão
10:          K[r,s] ← ADJ[r,s]
11:       fimse
12:    fimpara
13: fimpara
14: para r de 0 até n-1 faça
15:    para s de 0 até n-1 faça
16:       para t de 0 até n-1 faça
17:          K[s,t] = K[s,t] minimo K[s,r] + K[r,t]
18:       fimpara
19:    fimpara
20: fimpara
21: imprima K
```

Para estudar o algoritmo de caminho mínimo, verifique-se um caso comentado passo-a-passo. Veja a figura 9.13

Este grafo tem a matriz ADJ igual a

$$\begin{matrix} 5 & 0 & 1 \\ 1 & 1 & 3 \\ 8 & 0 & 0 \end{matrix}$$ que dá origem à matriz K_0 valendo $$\begin{matrix} 5 & \infty & 1 \\ 1 & 1 & 3 \\ 8 & \infty & \infty \end{matrix}$$

Para r = 0, s = 0
Q[0,0] ← Q[0,0] mínimo Q[0,0] + Q[0,0]
Q[0,1] ← Q[0,1] mínimo Q[0,0] + Q[0,1]

 ©88-08, Pedro Kantek

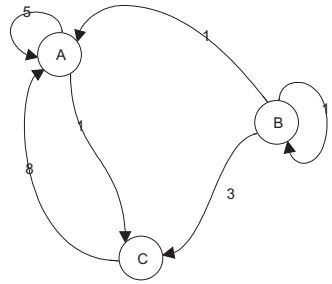


Figura 9.13: Um grafo para achar o caminho mínimo

Q[0,2] ← Q[0,2] mínimo Q[0,0] + Q[0,2]
Para r = 0, s = 1
Q[1,0] ← Q[1,0] mínimo Q[1,0] + Q[0,0]
Q[1,1] ← Q[1,1] mínimo Q[1,0] + Q[0,1]
Q[1,2] ← Q[1,2] mínimo Q[1,0] + Q[0,2]
Para r = 0, s = 2
Q[2,0] ← Q[2,0] mínimo Q[2,0] + Q[0,0]
Q[2,1] ← Q[2,1] mínimo Q[2,0] + Q[0,1]
Q[2,2] ← Q[2,2] mínimo Q[2,0] + Q[0,2]

isto gera a primeira matriz K_1 $$\begin{matrix} 5 & \infty & 1 \\ 1 & 1 & 3 \\ 8 & \infty & 9 \end{matrix}$$

Idêntico procedimento leva a K_2 $$\begin{matrix} 5 & \infty & 1 \\ 1 & 1 & 3 \\ 8 & \infty & 9 \end{matrix}$$

e a K_3 $$\begin{matrix} 5 & \infty & 1 \\ 1 & 1 & 3 \\ 8 & \infty & 9 \end{matrix}$$

Heurística Para a primeira matriz, riscar a primeira linha e a primeira coluna. Valores que estão sob o risco são escritos como estão. Os demais devem ser comparados com a soma dos elementos correspondentes riscados, e o menor dos dois deve ser escrito. Para a segunda matriz riscar a segunda linha e coluna e assim por diante.

Vejamos um exemplo em 9.14

K = 0

$$\begin{matrix} \infty & 4 & 8 & 1 \\ \infty & \infty & \infty & 3 \\ 2 & \infty & \infty & \infty \\ \infty & \infty & 1 & \infty \end{matrix}$$

K = 1

$$\begin{matrix} \infty & 4 & 8 & 1 \\ \infty & \infty & \infty & 3 \\ 2 & 6 & 10 & 3 \\ \infty & \infty & 1 & \infty \end{matrix}$$

K = 2

©88-08, Pedro Kantek

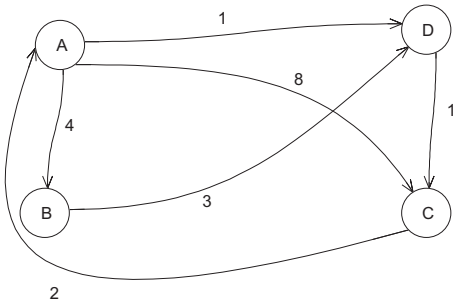


Figura 9.14: Um grafo

∞ 4 8 1
∞ ∞ ∞ 3
2 6 10 3
∞ ∞ 1 (
K = 3
10 4 8 1
∞ ∞ ∞ 3
2 6 10 3
3 7 1 4
K = 4 (resultado final)
4 4 2 1
6 10 4 3
2 6 4 3
3 7 1 4

Exemplos: Seja a seguinte matriz de adjacência:

0 3 4 0 2 0 5
0 4 0 0 0 0 0
0 0 0 5 2 3 7
1 6 1 0 0 3 0
8 4 7 0 0 0 0
0 1 2 0 0 0 8
0 0 0 6 0 0 0

Ela terá como matriz de caminho mínimo, os seguintes resultados (9999 é infinito)

10	3	4	9	2	7	5
9999	4	9999	9999	9999	9999	9999
6	4	5	5	2	3	7
1	4	1	6	3	3	6
8	4	7	12	9	10	13
8	1	2	7	4	5	8
7	10	7	6	9	9	12

EXERCÍCIO 150 Escrever uma matriz indicando em cada elemento (i,j) quantas bolinhas (nodos) eu preciso passar andando pelo grafo para ir de i até j.

9.4.1 Centro de um grafo (orientado)

Define-se o vértice central de um digrafo como sendo aquele vértice que tem a menor excentricidade. Define-se excentricidade de um vértice como segue:

Seja um digrafo G com o conjunto de vértices V e um dado vértice u. A exc(u) é o maior valor existente entre todos os caminhos mínimos que tem u como destino.

Exemplo, seja o grafo dado 9.15

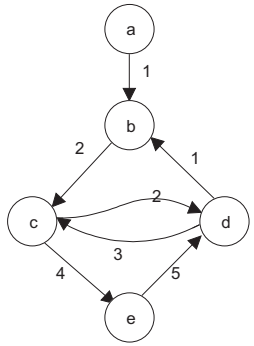


Figura 9.15: Um grafo

As excentricidades de cada vértice são a = ∞, b = 6, c = 8, d = 5 e e = 7, logo o centro do grafo é o vértice d.

Desafio

- Implemente o algoritmo de Floyd.

9.5 Caminho mínimo e por onde passar

Uma pequena modificação no algoritmo acima, permite que se calcule a matriz de caminhos mínimos junto com a matriz que indica por quais vértices tal caminho mínimo passa É o algoritmo de Floyd com um comando a mais:

Algoritmo Qual-Menor-Caminho (Alg. de Floyd modificado):

Entrada: matriz de adjacência e transforma-a na de caminhos mínimos

```
1: inteiro K [n, n], W[n, n]
2: inteiro infinito 9999999
3: leia ADJ (de ordem n)
4: inteiro r,s,t
5: para r de 0 até n-1 faça
6:    para s de 0 até n-1 faça
7:       se ADJ[r,s] = 0 então
8:          K[r,s] ← ∞
9:       senão
10:         K[r,s] ← ADJ[r,s]
11:      fimse
```

```
12:       W[r, s] ← 0
13:     fimpara
14:   fimpara
15:   para r de 0 até n-1 faça
16:     para s de 0 até n-1 faça
17:       para t de 0 até n-1 faça
18:         se (K[s,r] + K[r,t]) < K[s,t] então
19:           K[s,t] ← K[s,r] + K[r,t]
20:           se W[s,r] = 0 {estes são os comandos a mais}
21:           W[s,t] ← r {idem}
22:         senão
23:           W[s,t] ← W[s,r] {idem}
24:         fimse
25:       fimpara
26:     fimpara
27:   fimpara
28:   imprima K
```

Ao final, a matriz W deve ter todos os zeros que ela ainda tiver substituídos pelo número da coluna de cada zero. Note que este algoritmo considera a origem da contagem em 1.

Exemplo passo-a-passo, do grafo visto na figura 9.16

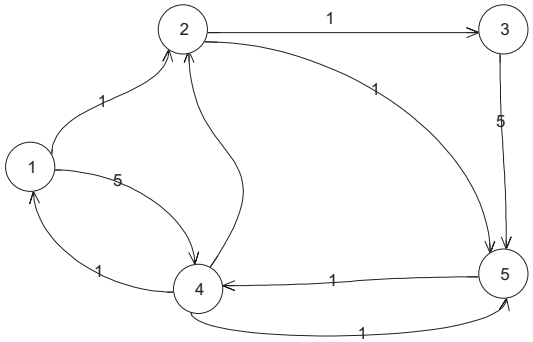


Figura 9.16: Um grafo

K (inicial) =

0	1	x	5	x
x	0	1	x	1
x	x	0	x	5
1	3	x	0	1
x	x	x	1	0

K (final) =

0	1	2	3	2
3	0	1	2	1
7	8	0	6	5
1	2	3	0	1
2	3	4	1	0

W =

1	2	2	2	2
5	2	3	5	5
5	5	3	5	5
1	1	1	4	5
4	4	4	4	5

Para ir do vértice 2 ao vértice 1 faz-se W[2,1] = 5, W[5,1] = 4 e W[4,1] = 1, portanto o caminho é 2 - 5 - 4 - 1.

Por exemplo, seja o grafo dado pela seguinte matriz de adjacência:

0	0	3	0	6	0	0
0	0	3	0	0	0	0
0	0	0	0	0	8	2
0	8	5	0	0	2	0
0	0	0	6	0	2	4
3	1	7	4	0	0	0
1	0	0	0	0	1	0

Ao se usar o algoritmo de Floyd para encontrar os caminhos mínimos, vamos achar os seguintes valores

0	7	3	10	6	6	5
6	0	3	10	12	6	5
3	4	0	7	9	3	2
5	3	5	0	11	2	7
5	3	6	6	0	2	4
3	1	4	4	9	0	6
1	2	4	5	7	1	0

1	***3***	3	3	5	3	3
3	2	3	3	3	3	3
7	***7***	3	7	7	7	7
6	6	3	4	6	6	3
6	6	6	4	5	6	7
1	2	2	4	1	6	2
1	***6***	1	6	1	6	7

A segunda matriz é W (dentro do algoritmo de Floyd modificado) e sua interpretação deve ser como segue: Suponha que deseja-se saber qual o menor caminho entre os nodos 1 e 2. Já se sabe que este caminho vale 7 (linha 1, coluna 2 da matriz K), mas quer-se saber por onde ir ? A resposta, está em W.

Começa-se em W[1,2]. Lá está 3 (em cor vermelha na matriz)
Este 3, nos remete, para a mesma coluna (a 2), mas com a linha = 3. Em W[3,2] está o valor 7 (azul)
Este 7, nos remete para a mesma coluna 2, mas com linha = 7. Em W[7,2] está o valor 6 (verde)
Finalmente, este 6, nos remete para a linha 6, coluna 2. Lá está W[6,2] = 2, que relembrando, era o nodo destino.

Verificar-se-á agora se as duas matrizes estão compatíveis. K diz que a distância entre 1 e 2 vale 7 unidades. W diz que esse caminho passa pelos nodos: 3, 7, 6 e 2. As

distâncias entre:

Nodo origem	Nodo destino	Valor
1	3	3
3	7	2
7	6	1
6	2	1
TOTAL		7

EXERCÍCIO 151 Na mesma matriz acima, encontre os caminhos: a) entre os nodos 2 e 4 (valor 10) b) entre os nodos 4 e 5 (valor 11) c) entre os nodos 1 e 6 (valor 6)

Para que este algoritmo funcione, há que

- Criar uma matriz de custo Q obtida a partir da matriz de adjacência e onde zero foi trocado por infinito.
- A matriz Q deve ter ZEROS na diagonal principal
- Deve ser criada uma matriz de passos P, quadrada, de mesma ordem, originalmente zerada.
- Rodado o algoritmo, ao final, os elementos de P que ainda forem zero, devem ser substituídos pelo número da coluna a que pertencem.

Seja o exemplo dado na figura 9.17

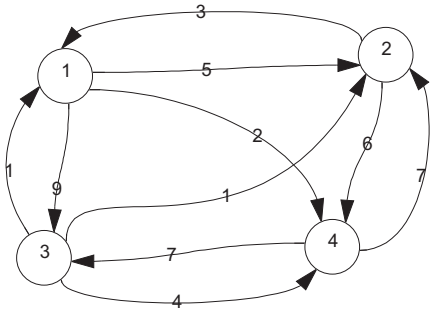


Figura 9.17: Um grafo

Cuja matriz de adjacência é

```
0 5 9 2
3 0 0 6
1 2 0 4
0 7 7 0
```

Que transformada na matriz de custo Q, e com a diagonal zerada

```
0 5 9 2
3 0 X 6
1 2 0 4
X 7 7 0
```


Que aplicado ao algoritmo dá a seguinte matriz final de custos

```
0  5  9  2
3  0 12  5
1  2  0  3
8  7  7  0
```

E a seguinte matriz de passos P

```
1  2  3  4
1  2  1  1
1  2  3  1
2  2  3  4
```

O teste correto acima citado, é necessário. Veja-se o seguinte caso Seja o caso

```
MADJ=0  2  6  5
     3  0  6  1
     8  1  0  8
     4  4  2  0
```

que tem como resposta em termos de Q

```
0  2  5  3
3  0  3  1
4  1  0  2
4  3  2  0
```

e a matriz final é

```
1  2  2  2
1  2  4  4
2  2  3  2
1  3  3  4
```

Para ver o funcionamento da coisa, veja-se o menor caminho de 1 para 3. Olhando a Q0, vê-se que há uma ligação direta custando 6 Se se olhar a matriz P errada, tem-se: $1 \rightarrow 4 \rightarrow 3$ valendo um total de 7 Se se olhar a matriz P certa, tem-se: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$, valendo 5 que é o resultado correto.

Veja-se a construção

0	2	6	5
3	0	6	1
8	1	0	8
4	4	2	0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Começando com K=1

0	2	6	5
3	0	6	1
8	1	0	8
4	4	2	0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

tem-se o seguinte resultado (não houve nenhuma mudança)

0	2	6	5
3	0	6	1
8	1	0	8
4	4	2	0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Fazendo K=2

0	2	6	5
3	0	6	1
8	1	0	8
4	4	2	0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

fica

0	2	6	**3**
3	0	6	1
4	1	0	2
4	4	2	0

0	0	0	**2**
0	0	0	0
2	0	0	**2**
0	0	0	0

Fazendo K=3

0	2	6	3
3	0	6	1
4	1	0	2
4	4	2	0

0	0	0	2
0	0	0	0
2	0	0	2
0	0	0	0

fica

0	2	6	3
3	0	6	1
4	1	0	2
4	**3**	2	0

0	0	0	2
0	0	0	0
2	0	0	2
0	**3**	0	0

Fazendo K=4

0	2	6	3
3	0	6	1
4	1	0	2
4	3	2	0

0	0	0	2
0	0	0	0
2	0	0	2
0	3	0	0

fica

0	2	**5**	3
3	0	3	1
4	1	0	2
4	3	2	0

0	0	**2***	2
0	0	**4**	0
2	0	0	2
0	3	0	0

Finalmente, convertendo a matriz P final, fica

1	2	**2**	**2**
1	2	**4**	4
2	2	3	**2**
1	**3**	3	4

ou

0	2	2	2
1	0	4	4
2	2	0	2
1	3	3	0

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Capítulo 10

Árvores

10.1 Introdução

A árvore é uma das estruturas de informação mais utilizadas e úteis. Como tantas coisas na informática a árvore aproveitou um nome já existente para batizar uma coisa nova quando esta foi inventada, e como sempre buscou-se algo que tivesse semelhança com o conceito novo. A estrutura em árvore é uma estrutura hierárquica, na qual um segmento é o principal e dele se derivam todos os demais. Esse mesmo desenho se repete nos segmentos derivados, e a semelhança que existe com uma árvore tradicional ocorre quando se compara esse segmento básico ao tronco de uma árvore, os demais aos galhos, até que finalmente se chega às folhas.

Árvores ocorrem no nosso dia a dia. Veja-se por exemplo, a tabela das semifinais da copa mundial de futebol de 1994: ==== inserir as finais da copa de 94 ===== Outro exemplo de árvore está no sumário deste livro. Veja às págs 2 a 8, como os assuntos do livro se concatenam e formam um todo hierárquico. Outros exemplos: uma expressão aritmética tipo $((1 + 2) \times (3/4)) - 5$. Ou então uma árvore genealógica. Finalmente, um exemplo vistoso e já estudado é o da alocação de espaços em disco pelo sistema operacional UNIX .

Pode aplicar-se então, a seguinte definição: Uma árvore é uma estrutura de informação que é constituída por nodos, que são os itens elementares dos dados, e onde os nodos se hierarquizam de alguma maneira pré-estabelecida.

Esta hierarquia assume características que podem ser estudadas fazendo-se analogia com a relação "pai-filho"(ou mãe-filho, para as feministas).

Diz-se então que dois nodos estão ligados através de uma relação pai ou filho. Por exemplo, se uma árvore contém dois nodos, A e B, pode-se ter A sendo pai de B ou A sendo filho de B.

Todos os nodos (com exceção de um deles) terão um pai. Já os nodos podem ter ou não ter filhos. O único nodo que não tem pai é conhecido como nodo raiz.

Uma segunda definição, esta recursiva para árvore é: Uma árvore A é uma estrutura definida por a) Uma estrutura vazia OU b) um nodo raiz, ao qual estão vinculadas sub-árvores

10.1.1 Porquê Árvores ?

Do que já se estudou (listas seqüenciais e listas encadeadas), é possível extrair alguns ensinamentos. Relembrando: Uma lista encadeada é um vetor de itens, e que, se mantido ordenado, permite uma consulta muito rápida se for usada a busca binária.

> Por exemplo: Se tivermos um vetor de 1000 elementos ordenados, e for feita uma busca binária, o primeiro acesso dividirá o vetor em 2 segmentos de 500, abandonando-se um deles. O seguinte acesso abandonará 250, o terceiro 125 e assim sucessivamente. Neste caso (1000 elementos), a quantidade máxima de acessos é de 10.

Em outras palavras, uma lista seqüencial tem um tempo de acesso de ordem logarítmica e pode ser expresso por $O(\log_2 n)$ onde n é o tamanho da estrutura.

O problema surge quando a operação desejada não é a consulta (que é rápida, como vimos) e sim a inclusão ou a exclusão. Ambas são muito demoradas, pois para manter a estrutua ordenada, a inclusão deve se realizar em um ponto específico. Como a estrutura é sequencial, buracos no meio dela não são permitidos, e com isso sempre é necessário um esforço de deslocamento de dados para cima ou para baixo. Note-se que esse esforço é puro emphoverhead, pois ele não contribui com nenhuma computação útil.

> Por exemplo, se tivermos o vetor 1, 5, 23, 56, 80, 90, 111, 114, 170, 235, 250, 289, 400, 501, 800 e quisermos incluir o elemento 171, este novo número ocupará o lugar do 235, que para não ser perdido precisará ser deslocado (ele e todos os seus vizinhos até o fim da estrutura) uma posição à direita.

A exclusão sofre do mesmo mal. Quando um determinado elemento deve ser extraído da estrutura o buraco não pode permanecer o que obriga os elementos que ficaram à direita desta posição serem trazidos para ocupar o espaço deixado.

Tanto em um como noutro caso, a complexidade média da operação é de $O(n/2)$, onde n é o tamanho da estrutura (e supondo-se inclusão ou exclusão na sua metade), desprezando-se neste cálculo o esforço necessário para a localização do lugar onde a operação será feita.

Usando outra estrutura já estudada, as listas encadeadas , percebe-se uma situação claramente oposta. Nestas, as operações de inclusão e exclusão são muito rápidas, ao passo que a demora existe na operação de pesquisa. Como vimos, uma inclusão implica em modificar apenas 2 apontadores (ou 4 se a lista for duplamente ligada, ou ainda mais genéricamente $2 \times k$, onde k é o grau de encadeamentos da estrutura), novamente desprezando-se aqui o tempo gasto para a localização do local da inclusão.

Já a exclusão tem a metade do custo acima, pois é apenas 1 apontador para listas simplesmente ligadas,2 para listas duplamente ligadas e genericamente k apontadores para listas k-uplamente ligadas.

custos em listas encadeadas

Estrutura	O(inclusão)	O(exclusão)	O(pesquisa)
lista sequencial	$O(n/2) + P$	$O(n/2) + P$	$P = O(\log_2 n)$
lista encadeada	$O(2) + P$	$O(1) + P$	$P = O(n/2)$

As árvores surgem como resposta a uma estrutura que seja rápida para pesquisar (tipicamente $log_k n$, onde k é a ordem da árvore e n o número de elementos que a formam) e igualmente rápida para inclusão e exclusão, tipicamente também tem complexidade O($log_k n$).

custos em árvores

Estrutura	O(inclusão)	O(exclusão)	O(pesquisa)
árvore	$O(\log_k n)$	$O(\log_k n)$	$O(\log_k n)$

©88-08, Pedro Kantek

10.1.2 Representações em árvores

Nas árvores os segmentos são chamados nós (ou nodos), e portanto uma árvore é uma estrutura de informação formadas por nodos interconectados. O nodo principal, o primeiro a ser criado é chamado nodo raiz. Dele saem ramificações para mais nodos, que por sua vez são nodos raiz de outras sub-árvores. O detalhe é que todas definições válidas para uma árvore também o são para as sub-árvores, o que caracteriza uma definição recursiva por excelência [Vide em Niklaus Wirth]. Os nodos que não são raiz nem folha são chamados nodos intermediários. A representação gráfica de uma árvore é através de um esquema do tipo:

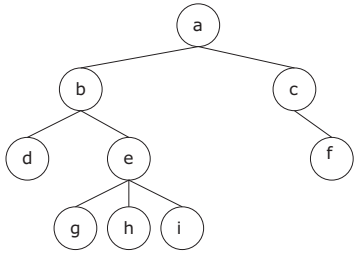


Figura 10.1: Um exemplo de árvore

Na árvore representada pela figura 10.1, a raiz é o nodo a, os nodos intermediários são b, c e e, e as folhas são os nodos d, f, g, h e i. Note-se que todos os nós são raízes de uma sub-árvore. O nodo b é raiz da sub-árvore que começa nele e assim por diante. Note também que um nodo folha, por exemplo f, também é uma árvore (ou uma sub-árvore. Tanto faz, sub-árvores são árvores), neste caso uma árvore vazia.

Além dessa representação gráfica mais usual (já que é a que se assemelha a uma árvore, só que de ponta cabeça), existem outras representações que também podem ser usadas. Vejamos algumas delas, e usando sempre a mesma árvore vista acima.

Na figura 10.2, cada elipse representa uma árvore. O elemento que está isolado na elipse é a raiz da mesma, e as elipses que estão dentro da elipse são as sub-árvores.

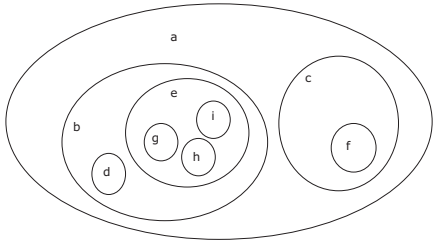


Figura 10.2: árvores através de diagramas de Venn

Outra representação é a "por identação", na qual a árvore é representada na forma de um texto (ótimo quando não se tem o recurso ao desenho), como no exemplo:

Note que o primeiro símbolo (a) é a raiz da árvore. Os elementos que estão no mesmo nível da árvore se encontram na mesma vertical na representação.

 ©88-08, Pedro Kantek


```
a : b : d
        e : g
          : h
          : i
    c : f
```

Figura 10.3: árvores representadas por identação

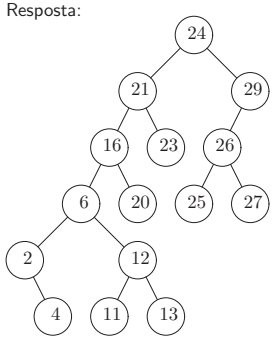
Outra representação, mais adequada às máquinas de escrever é a "por parênteses"

```
a(b(d,e(g,h,i)),c(f))
```

Figura 10.4: árvores representadas por parênteses

Note que cada árvore tem a seguinte representação: Árvore = raiz (sub-árvore 1, sub-árvore2, ...). Esta mesma representação sendo seguida recursivamente dentro da árvore, representa inequivocamente toda a estrutura de maneira correta.

EXERCÍCIO RESOLVIDO 12 Desenhe a seguinte árvore binária, representada através de parênteses:
(24(21(16(6(2()(4))(12(11)(13)))(20))(23))(29(26(25)(27))()))
Resposta:



10.1.3 Conceitos básicos em árvores

Para efeito de entendimento destes conceitos, acompanhe-os pela figura 10.1.

Grau de um nodo é o número de sub-árvores que nascem a partir deste nó. No exemplo acima, os graus são: grau(a) = 2, grau(b)=2, grau(c)=1, grau(d)=0, grau(e)=3, grau(f)=0, grau(g)=0, grau(h)=0, grau(i)=0.

Raiz é o nodo principal, de onde se originam todos os demais. É o único nodo na árvore que não tem pai.

Folhas nodos de grau igual a zero.

Nós de derivação ou intermediários: nós que não são nem raiz nem folhas.

©88-08, Pedro Kantek

Nível de um nodo é a quantidade de arestas existentes entre ele e a raiz. No exemplo: nível(a)=0, nível(b)=1, nível(c)=1, nível(d)=2, nível(e)=2, nível(f)=2, nível(g)=3, nível(h)=3, nível(i)=3.

Altura da árvore é o valor do maior nível existente na árvore.

Ascendente de um nodo nó anterior (também chamado pai). Trata-se da raiz de uma árvore da qual este nodo é raiz de uma sub-árvore.

Descendente de um nó São as raízes das sub-árvores que compõe a árvore da qual este nodo é o raiz.

Nodos irmãos os outros descendentes do mesmo ascendente, ou os outros filhos do mesmo pai.

Árvore ordenada é aquela onde a ordem das sub-árvores é importante.

Visitação é o processo de percorrer os nodos de uma árvore. A visitação é feita com um objetivo qualquer: localizar um nodo, somar algum valor de cada nodo, imprimí-lo, etc. Apenas percorrer um certo caminho na árvore já é considerada uma visitação .

Percorrer a árvore trata-se de visitar todos os nodos de uma árvore segundo uma ordem qualquer dada. Existem diversos percursos em uma árvore.

Chave de um nodo é o valor que permite acessar um nodo e que permite ordenar uma árvore. Quando se desenha uma árvore é o valor chave que é escrito dentro dos círculos.

10.2 Alocação de árvores

Como sempre, árvores podem ser alocadas de maneira seqüencial ou encadeada. No primeiro caso, há a contigüidade física e não podem ficar buracos dentro da estrutura. Neste caso, incluir um novo nodo, em geral, implicar em refazer completamente a árvore. Exatamente por essa razão, em geral só se aplica a árvores estáticas e ainda assim não é muito usado.

Já a alocação encadeada vai ser aqui estudada segundo duas vertentes. A primeira, chamada alocação encadeada absoluta será feita com blocos reais de memória usando-se endereços absolutos de memória nas alocações e acessos. E uma segunda vertente, chamada indireta que se utilizará de uma matriz (uma estrutura seqüencial) para simular uma alocação encadeada. Embora o desempenho não seja o mesmo da alocação absoluta, há vantagens no entendimento dos algoritmos na sua depuração e correção.

10.2.1 Alocação seqüencial

Esta não é a modalidade ideal de representação, já que dificulta a manutenção da árvore, ou seja as inclusões e exclusões de nodos, pode ser utilizada para a representação de uma árvore.

Naturalmente, apenas a imaginação do programador limita as possibilidades de representação, mas eis algumas possibilidades.

 ©88-08, Pedro Kantek

a	2	b	2	c	1	d	0	e	3	f	0	g	0	h	0	i	0

Figura 10.5: árvore seqüencial com explicitação de filhos

Explicitando a quantidade de filhos de cada nó

Reservam-se pares de posições contíguas de memória, a primeira para o nome (ou número, ou endereço, ou qualquer outro identificador), seguido pelo número de filhos que ele contém. A representação da árvore 10.1 nesta modalidade, ficaria assim:

A representação da figura 10.5 deve ser assim interpretada: a tem 2 filhos, que vem a seguir e são o b e o c. Por sua vez, b tem 2 filhos que vem a seguir e são d e e. Note que o na análise dos filhos de b, c é pulado, uma vez que já foi analisado como filho de a. O nodo d não tem filhos e o nodo e tem 3 filhos, e aqui se termina a lista de filhos de b. É hora de retornar e analisar c, que ficou pendente. Ele tem um único filho que é f, que por sua vez, não tem filhos. Finalmente, vem os filhos de e, que são g, h e i, nenhum deles com filhos.

Com delimitadores, usando notação pré-ordem

Continua sendo a mesma árvore, a 10.1.

a	b	d	*	e	g	*	h	*	i	*	*	*	c	f	*	*	*

Figura 10.6: árvore seqüencial com filhos, pré-ordem

A representação da figura 10.6 deve ser assim interpretada: A raiz é a, que tem filho b, que por sua vez tem filho d. O outro filho de b é e, que por sua vez tem os filhos g (sem filhos), h (idem) e i (idem). O segundo asterisco depois do i, encerra a família de e, e o terceiro, encerra a família de b. Depois vem c, que é filho de a (a única que não foi encerrada), que tem como filho apenas f (1º asterisco). O segundo "*" encerra a lista de filhos de c, e o terceiro, a lista de filhos de a, que era a raiz.

Com delimitadores, usando notação pós-ordem

Ainda a mesma árvore, a 10.1.

a	c	f	*	*	b	e	i	*	h	*	g	*	*	d	*	*	*

Figura 10.7: árvore seqüencial filhos, pós-ordem

A raiz é a, que tem filho c, que por sua vez tem filho f. O primeiro asterisco encerra a lista de filhos de f, e o segundo a de c. Depois vem o filho (de a) chamado b, que tem o filho e, que tem o filho i, que não tem filhos (por causa do asterisco a seguir). Seus irmãos são h e g que não tem filhos. O segundo asterisco após o g encerra a lista de e, e a seguir vem o seu irmão d, que não tem filhos (1º asterisco). O segundo encerra a lista de b e o terceiro encerra a.

Variante para economia de memória

A figura 10.8 é uma simplificação da anterior, apenas economizando algumas posições de delimitadores. Será mostrada apenas a notação pré-ordem, já que a pós fixada seguiria o mesmo caminho.

©88-08, Pedro Kantek

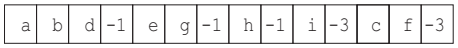


Figura 10.8: árvore seqüencial com economia de memória

Repare que ao invés de usar várias posições para delimitadores, usou-se uma apenas, tomando o cuidado de negar o número, para diferenciá-lo de possíveis conteúdos numéricos.

Embora difícil de enxergar a árvore acima é exatamente a mesma árvore mostrada na figura 10.1.

10.2.2 Alocação encadeada absoluta

Usando uma lista encadeada para cada nodo, poderíamos ter o que se vê na figura 10.9.

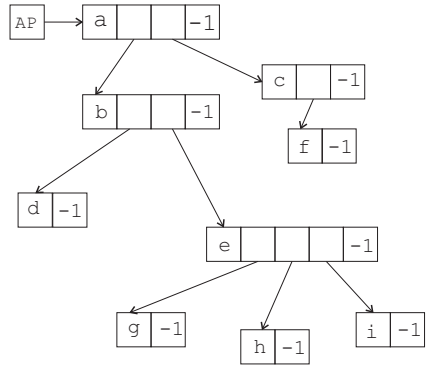


Figura 10.9: árvore usando listas encadeadas

A dificuldade dessa representação é que o número de filhos de cada nó é variável, o que nos força a montar uma lista de listas.

Uma solução para esse problema poderia ser a utilização de nodos de tamanho fixo para os nodos de tamanho intermediário, como vemos na figura a seguir.

O problema que ocorre aqui, é que os nodos podem ter tamanhos diversos. Imagine por exemplo, que os nodos de dados (identificados no desenho por uma letra) sejam compostos na verdade por 30 ou 40 inteiros. Isso obrigaria a utilização de dois tipos de nodos, ou de duas áreas de listas separadas. Uma solução para este problema poderia ser a utilização de nodos contendo 3 valores, sendo que o primeiro é o nodo propriamente dito, o segundo é o apontador da cadeia de filhos e o terceiro é o apontador da cadeia de irmãos, como se pode ver em 10.10.

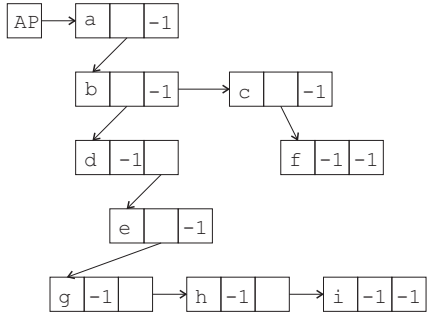
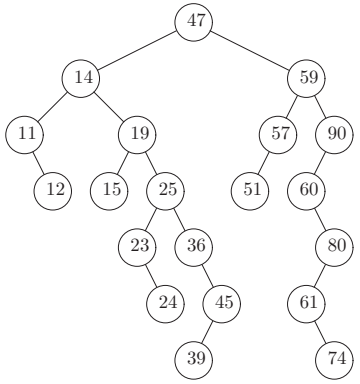


Figura 10.10: árvore usando nodos de tamanho fixo

EXERCÍCIO RESOLVIDO 13 Exemplo em alocação absoluta. Considere a M, que contém uma árvore, cuja raiz está apontada por M[0] (neste caso, 2) e onde cada nodo está formado por 3 inteiros a saber: endereço do filho esquerdo (representado por & fe), endereço do filho direito (&fd) e conteúdo da chave.

```
Procure desenhar a árvore
M|  0  1  2  3  4  5  6  7  8  9
-+------------------------------
0|  2 53  5 16 47  8 13 14 -1 23
1| 11 98 94 44 19 19 26 33 59 36
2| 29 25 40 -1 -1 12 47 -1 57 -1
3| 39 36 74 50 -1 90 -1 54 23 65
4| -1 45 35 13 -1 -1 15 -1 -1 51
5| -1 57 60 99 -1 -1 24 60 -1 80
6| -1 68 61 42 61 -1 -1 39 -1 -1
7| 74 63 91 51 73 97 85 57 23 74
8| 35 92 40 33 26  7 27 37 95 46
9| 91 27 38 33 65 88 57 24 86 24
```

Resposta:



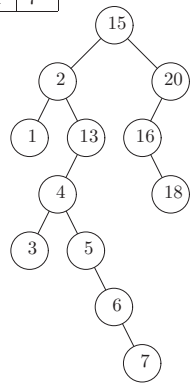
10.2.3 Alocação Encadeada relativa

A alocação encadeada relativa, na sua versão mais simples, para árvores binárias, pode ser entendida como uma matriz, composta ao mínimo de 3 colunas: uma contém o endereço relativo do filho esquerdo, a outra o do filho direito e a terceira o conteúdo do nodo.

Acompanhe no exemplo: Supondo que a raiz reside na primeira linha da matriz.

1	3	2	15
2	5	-1	20
3	8	4	2
4	6	-1	13
5	-1	11	16
6	10	7	4
7	-1	9	5
8	-1	-1	1
9	-1	12	6
10	-1	-1	3
11	-1	-1	18
12	-1	-1	7

A resposta:



Para você fazer, agora:

EXERCÍCIO 152 Para fazer este exercício, você deve interpretar a estrutura de dados, desenhar a árvore, obter os vetores de nodos resultados dos caminhamentos pedidos e finalmente, informar os elementos pedidos desses vetores.

1. Alocação encadeada relativa (usando cursores): árvore a

	filho esquerdo	filho direito	valor
1	5	2	3
2	3	6	11
3	7	4	7
4	-1	8	8
5	-1	-1	1
6	9	-1	19
7	-1	-1	4
8	-1	-1	10
9	11	10	13
10	12	-1	16
11	-1	-1	12
12	-1	-1	15

2. Alocação encadeada absoluta (usando o modelo M): árvore b

M	0	1	2	3	4	5	6	7	8	9
0	2	36	8	5	69	12	54	95	18	33
1	11	7	47	15	76	36	29	86	-1	23
2	1	4	35	44	26	6	-1	-1	8	-1
3	-1	87	10	-1	39	25	-1	50	78	60
4	64	52	56	39	57	-1	5	-1	-1	72
5	-1	-1	82	73	-1	-1	97	-1	-1	4
6	-1	-1	30	44	-1	67	58	-1	-1	68
7	6	55	24	55	77	35	33	41	35	62
8	90	46	75	89	93	64	45	52	4	46
9	61	33	72	25	9	15	32	25	96	81

3. Usando parênteses: árvore c

```
(3(1()(2))(8(6)(26(19(18(9()(12))())(21(20)(24)))
      (29(27)()))))
```

Responda agora:

1. : Árvore a, caminhamento em pré-ordem, 9^o elemento:
2. : Árvore a, caminhamento em em-ordem, 9^o elemento:
3. : Árvore a, caminhamento em pós-ordem, 8^o elemento:
4. : Árvore b, caminhamento em pré-ordem, 14^o elemento:
5. : Árvore b, caminhamento em em-ordem, 11^o elemento:
6. : Árvore b, caminhamento em pós-ordem, 11^o elemento:
7. : Árvore c, caminhamento em pré-ordem, 15^o elemento:
8. : Árvore c, caminhamento em em-ordem, 12^o elemento:
9. : Árvore c, caminhamento em pós-ordem, 11^o elemento:
10. : Árvore b, caminhamento em pós-ordem, 19^o elemento:

EXERCÍCIO 153 Mesma definição do exemplo anterior.

1. Alocação encadeada relativa: árvore a

	filho esquerdo	filho direito	valor
1	2	3	3
2	-1	-1	1
3	5	4	8
4	-1	7	9
5	-1	6	4
6	10	-1	7
7	-1	8	10
8	-1	9	11
9	11	12	18
10	-1	-1	6
11	-1	-1	13
12	-1	-1	19

 ©88-08, Pedro Kantek

2. Alocação absoluta usando M: árvore b

M	0	1	2	3	4	5	6	7	8	9
0	4	75	56	74	7	40	90	50	10	12
1	20	14	30	3	17	30	72	27	24	56
2	58	47	20	46	-1	44	63	34	55	40
3	37	69	80	45	66	-1	32	-1	61	73
4	-1	-1	97	67	-1	-1	71	-1	-1	28
5	-1	-1	8	45	91	-1	-1	42	-1	-1
6	15	-1	-1	77	29	37	-1	-1	31	-1
7	-1	88	83	35	12	6	77	58	68	21
8	90	8	67	91	54	54	73	19	82	43
9	65	23	42	50	95	77	40	82	62	34

3. Usando parênteses: árvore c

```
(29(20(16(13(10(5(1()(4))(7()(8)))(12))(14))
  (19))(21()(23)))())
```

Responda agora:

1. : Árvore a, caminhamento em pré-ordem, 8^o elemento:
2. : Árvore a, caminhamento em em-ordem, 7^o elemento:
3. : Árvore a, caminhamento em pós-ordem, 9^o elemento:
4. : Árvore b, caminhamento em pré-ordem, 15^o elemento:
5. : Árvore b, caminhamento em em-ordem, 13^o elemento:
6. : Árvore b, caminhamento em pós-ordem, 17^o elemento:
7. : Árvore c, caminhamento em pré-ordem, 14^o elemento:
8. : Árvore c, caminhamento em em-ordem, 15^o elemento:
9. : Árvore c, caminhamento em pós-ordem, 10^o elemento:
10. : Árvore b, caminhamento em pós-ordem, 19^o elemento:

EXERCÍCIO 154 Mesma definição do exercício anterior.

1. Alocação encadeada relativa: árvore a

	filho esquerdo	filho direito	valor
1	11	2	2
2	3	4	15
3	5	7	12
4	-1	6	16
5	8	-1	11
6	-1	-1	18
7	-1	-1	13
8	12	9	6
9	10	-1	10
10	-1	-1	9
11	-1	-1	1
12	-1	-1	3

©88-08, Pedro Kantek

2. Alocação absoluta usando M: árvore b

M	0	1	2	3	4	5	6	7	8	9
0	4	4	47	32	7	45	91	10	-1	90
1	14	17	39	4	20	27	34	65	24	51
2	34	30	29	52	40	61	81	-1	37	35
3	55	-1	33	19	51	-1	28	-1	-1	38
4	-1	48	65	55	40	-1	-1	97	-1	-1
5	67	58	-1	25	33	-1	-1	32	-1	68
6	16	-1	-1	84	98	-1	-1	42	-1	-1
7	19	14	34	63	2	67	56	83	31	72
8	62	22	59	70	81	60	25	8	36	49
9	32	92	25	74	99	58	48	20	43	28

3. Usando parênteses: árvore c

```
(24(23(8(7(1()(3()(4)))())(21(16(12)(19(17)()))
(22)))())(26()(29)))
```

Responda agora:

1. : Árvore a, caminhamento em pré-ordem, 8^o elemento:
2. : Árvore a, caminhamento em em-ordem, 7^o elemento:
3. : Árvore a, caminhamento em pós-ordem, 9^o elemento:
4. : Árvore b, caminhamento em pré-ordem, 11^o elemento:
5. : Árvore b, caminhamento em em-ordem, 17^o elemento:
6. : Árvore b, caminhamento em pós-ordem, 11^o elemento:
7. : Árvore c, caminhamento em pré-ordem, 10^o elemento:
8. : Árvore c, caminhamento em em-ordem, 11^o elemento:
9. : Árvore c, caminhamento em pós-ordem, 10^o elemento:
10. : Árvore b, caminhamento em pós-ordem, 16^o elemento:

EXERCÍCIO 155 Mesma definição do exercício anterior

1. Alocação encadeada relativa: árvore a

	filho esquerdo	filho direito	valor
1	2	6	14
2	3	11	11
3	5	4	5
4	12	-1	9
5	8	-1	3
6	7	9	17
7	-1	-1	15
8	10	-1	2
9	-1	-1	20
10	-1	-1	1
11	-1	-1	12
12	-1	-1	8

 ©88-08, Pedro Kantek

2. Alocação absoluta usando M: árvore b

M	0	1	2	3	4	5	6	7	8	9
0	4	81	50	43	-1	7	5	25	10	19
1	15	28	93	2	48	18	31	60	21	36
2	35	-1	49	22	3	-1	63	6	-1	-1
3	95	39	-1	77	60	95	46	42	48	52
4	57	72	-1	-1	54	27	-1	68	39	-1
5	60	28	-1	-1	61	73	32	-1	-1	73
6	-1	71	32	-1	-1	7	9	22	-1	-1
7	40	-1	-1	34	8	51	64	20	61	41
8	17	80	18	17	84	25	17	98	71	29
9	18	3	81	3	17	22	2	76	75	68

3. Usando parênteses: árvore c

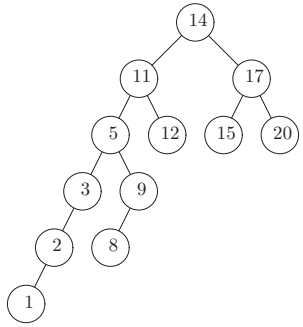
```
(1()(10(9(6()(7))())(12(11)(16(13)(30(17()
  (27(25(22))())(29)))()))))))
```

Responda agora:

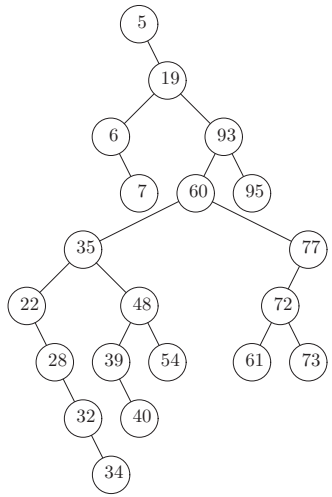
1. : Árvore a, caminhamento em pré-ordem, 10^o elemento:
2. : Árvore a, caminhamento em em-ordem, 7^o elemento:
3. : Árvore a, caminhamento em pós-ordem, 12^o elemento:
4. : Árvore b, caminhamento em pré-ordem, 18^o elemento:
5. : Árvore b, caminhamento em em-ordem, 15^o elemento:
6. : Árvore b, caminhamento em pós-ordem, 13^o elemento:
7. : Árvore c, caminhamento em pré-ordem, 9^o elemento:
8. : Árvore c, caminhamento em em-ordem, 14^o elemento:
9. : Árvore c, caminhamento em pós-ordem, 9^o elemento:
10. : Árvore b, caminhamento em pós-ordem, 12^o elemento:

Respostas dos exercícios acima.
152 - 13 13 16 76 68 11 27 24 27 95
153 - 10 9 10 71 71 30 21 23 19 90
154 - 10 11 18 32 84 42 12 22 21 81
155 - 17 11 14 61 61 73 13 29 27 48
Desenha-se a seguir as árvores A, B e C do exercicio 4 árvore a:

©88-08, Pedro Kantek

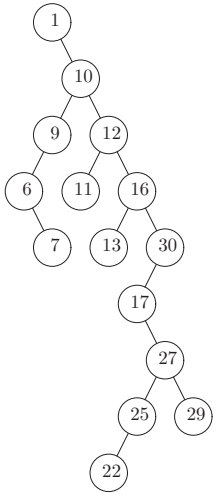


árvore b:



 ©88-08, Pedro Kantek

árvore c:



EXERCÍCIO 156 Desenhe a árvore representada por esta tabela (em encadeamento relativo).

filho esquerdo	filho direito	pai	conteúdo	
2	3	-1	52	1
4	5	1	21	2
6	7	1	70	3
8	9	2	7	4
10	11	2	27	5
-1	-1	3	56	6
-1	12	3	90	7
-1	13	4	3	8
-1	-1	4	15 *	9
-1	-1	5	25	10
-1	14	5	30	11
15	-1	7	99	12
-1	-1	8	4	13
-1	-1	11	49	14
16	-1	12	97	15
-1	-1	15	95	16

Se o número marcado com * for igual a 35, onde estará o erro ?

Desafio

- Escreva um programa que receba uma árvore na representação parentizada e imprima a identificação dos nodos folha que existem na árvore.
- Escreva um programa que receba uma árvore na representação usando cursores e imprima uma visualização da árvore no monitor de vídeo. Considere uma árvore com altura máxima de 4 e com grau 2.

©88-08, Pedro Kantek

10.3 Árvores binárias

A árvore binária é uma estrutura em árvore na qual os nodos podem ter 0, 1 ou 2 filhos no máximo. Eis uma definição recursiva de árvore binária "uma árvore binária é vazia ou é formada por uma raiz e duas sub-árvores binárias".

Um nodo em árvore binária pode ser encarado como uma estrutura contígua formada por 3 informações (filho esquerdo, conteúdo do nodo e filho direito) devidamente apontados pelo apontador do nó. Veja no exemplo:

apontador para o nodo	apontador para o filho esquerdo	conteúdo do nó	apontador para o filho direito

Como se verá a seguir, qualquer árvore pode ser representada através de uma árvore binária, e por essa razão ela é muito usada. Eis como ficaria o nosso exemplo, da figura 10.1 representado através de árvores binárias:

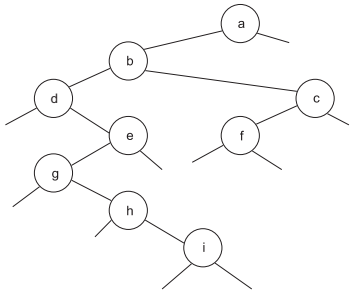


Figura 10.11: Exemplo de uma árvore binária

10.3.1 Transformando árvores em árvores binárias

Sempre é possível transformar qualquer árvore em árvore binária equivalente. É claro que a árvore terá outra forma, ela ficará mais "alta", mas nenhum dado é perdido.

A regra que é usada para transformar quaisquer árvores em uma árvore binária é bastante simples. No ramo esquerdo de um nodo colocam-se os filhos do nodo original, e no ramo direito, os irmãos. Note portanto que uma árvore genérica quando traduzida para árvore binária terá como raiz um nodo sem filhos à direita (já que por definição, a raiz nunca tem irmãos). Esta regra pode ser visualizada na figura 10.12

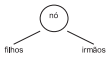


Figura 10.12: Regra para converter árvores

Obviamente quando essa transformação é feita o número de níveis da árvore muda, a árvore binária tem mais nodos para compensar a imposição no número máximo de filhos ser igual a 2. No exemplo acima, a árvore genérica do início do capítulo tinha 4 níveis, enquanto que a transformada em binária tem 7 níveis. Note na figura 10.3 como devem ser compostos os nodos de uma árvore binária para representar os dados de uma árvore de grau qualquer.

De fato, essa regra (a possibilidade de representar qualquer árvore como árvore binária) é absolutamente geral. Até mesmo uma lista encadeada pode ser representada como árvore. Por exemplo a lista constante na figura 10.13

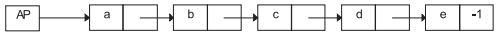


Figura 10.13: Uma lista encadeada

Pode ser representada pela árvore binária mostrada na figura 10.14.

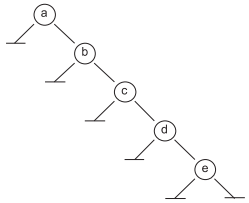


Figura 10.14: Uma árvore binária degenerada

Este tipo de árvore mostrada na figura 10.14 é conhecida como árvore degenerada, uma vez que cada nodo tem um único irmão e não tem filhos. Formalmente falando, uma árvore é degenerada quando tem altura n, n-1 nós tem grau 1 e um nó tem grau 0. Este fato prova que uma lista linear é um caso particular de árvore.

EXERCÍCIO RESOLVIDO 14 Suponha uma árvore binária de pesquisa construída a partir de dados de alunos de uma determinada instituição de ensino superior. A informação principal de cada nodo e que serve para acessar a árvore é o código de matrícula de cada aluno, que por hipótese é único, ou seja, não há dois alunos com o mesmo código. Supondo que a árvore esteja criada na memória e esteja sendo acessada por programas. Cada acesso é feito com um determinado número de matrícula e ao final o algoritmo que está manuseando a árvore informa se este número corresponde a um aluno existente naquela instituição ou se ao contrário é de um aluno inexistente.

Supondo, mais ainda, que a árvore contém N nodos, correspondendo a N alunos presentemente matriculados nessa instituição, pede-se quais as expressões (em função de N) que traduzem as complexidades mínima e máxima, e em que situação tais complexidades são atingidas, a saber: (indique qual a única alternativa correta):
a) A complexidade mínima é $O(n)$, atingida quando a árvore é degenerada e a complexidade máxima é $O(n^2)$ alcançada quando a árvore é completamente balanceada.
b) A complexidade mínima é $O(\log_2 n)$ atingida quando a árvore é completamente balanceada e a complexidade máxima é $O(n^2)$ quando a árvore está degenerada
c) A complexidade mínima é $O(\log_2 n)$ atingida quando a árvore está degenerada e a complexidade máxima é $O(n)$ atingida quando a árvore está completamente balanceada.
d) A complexidade mínima é $O(\log_2 n/n)$ atingida quando a árvore está completamente balanceada e a complexidade máxima é $O(n)$ quando a árvore está degenerada
e) n.d.a

Resposta: A complexidade mínima é $O(\log_2 n)$ quando a árvore está completamente balanceada e a máxima é $O(n)$ quando a árvore está degenerada, logo, resposta E.

Desafio

- Implementar um programa de computador que receba uma árvore de grau = 10 e imprima a árvore binária equivalente.

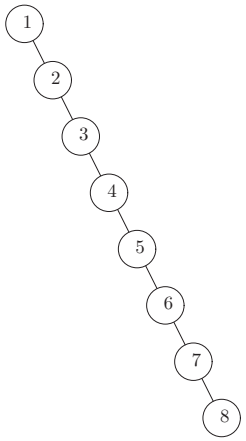
10.4 Árvores balanceadas

Lembrando que o custo de acesso a uma árvore é função do número de operações de acesso (e comparações) deve-se manter a altura da árvore como mínima a fim de que a estrutura permaneça otimizada.

Se a árvore for estática (isto é, não sofrer inclusões e exclusões) esta estrutura otimizada pode ser garantida simplesmente organizando as inserções. Por exemplo, criando-se uma ABP, com os valores (na ordem)

```
1,2,3,4,5,6,7,8
```

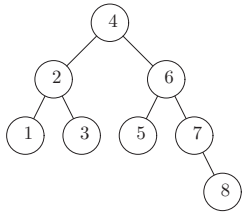
A árvore ficaria como



Para que esta árvore tivesse altura mínima, bastava organizar os dados de entrada:

```
4,2,6,1,3,5,7,8
```

e a árvore ficaria



Agindo assim, pode-se manter a complexidade do acesso em $O(\log_2 n)$, ao passo que se a árvore for degenerada a complexidade será $O(n)$.

Entretanto, se houverem inclusões e exclusões, a árvore pode ir degenerando. Para manter sua altura mínima (e complexidade $\approx O(\log_2 n)$, duas coisas podem ser feitas:

1. a cada inclusão/exclusão verificar se a altura da árvore não perdeu a característica de minimalidade. Esta é a solução que melhor desempenho apresenta em termos de acessos à árvore, MAS, este procedimento torna os algoritmos de inclusão / exclusão mais complicados (e demorados, logo caros).

2. De tempos em tempos, dar uma arrumada na árvore de maneira a garantir que sua altura seja de fato a mínima.

Quando a altura de uma árvore é mínima, diz-se que a mesma está balanceada. Entretanto, no pior caso, a operação de balanceamento pode ter complexidade $O(n)$, o que inviabiliza sua utilização a cada inserção / exclusão.

10.4.1 Árvores AVL

Um relaxamento dessa condição é dado pelas árvores AVL, introduzidas por Adelson, Velkskii e Landis em 1962. A definição de uma árvore AVL é tal que: Uma árvore binária T é AVL quando para qualquer nodo de T, as alturas das duas sub-árvores esquerda e direita diferem (em módulo) em no máximo 1 unidade.

Um nodo que satisfaça essa condição é dito regulado e se não satisfizer é dito não regulado. Uma árvore é AVL quando TODOS os seus nodos forem regulados.

Toda árvore completa é AVL, mas a recíproca não é necessariamente verdadeira.

Toda vez que uma inclusão é feita, verifica-se se a AVL não deixou de sê-lo e se for o caso a característica é restaurada mediante 4 operações (pág. 135 do livro verde) Note-se que a restauração da característica de AVL não se propaga para cima, apenas para baixo.

Esta degradação de não aumenta a complexidade da pesquisa ao mesmo tempo que diminui a complexidade de inclusões / exclusões. O critério de balanceamento é obtido mediante 4 operações básicas: rotação simples à direita, rotação simples à esquerda, rotação dupla à direita e rotação dupla à esquerda.

Acompanhe: Sejam as seguintes inclusões em árvore AVL inicialmente vazia:

```
INSERE(A,10)
INSERE(A,20)
INSERE(A,30)
INSERE(A,40)
INSERE(A,50)
INSERE(A,60)
INSERE(A,70)
```

Acompanhando o processo de criação e balanceamento da árvore AVL teríamos

10.4.2 Árvores rubro-negras

Nas árvores rubro-negras , cada nodo é colorido com preto ou com vermelho e a cor do nodo é o instrumento para garantir o balanceamento. Durante a inserção e a exclusão os nodos podem precisar ser rotacionados para preservar o balanceamento.

Uma árvore rubro-negra é uma árvore binária de pesquisa balanceada com as seguintes propriedades:

1. Todo nodo é preto ou vermelho

2. Toda folha é preta

3. Se um nodo é vermelho, seus dois filhos são pretos

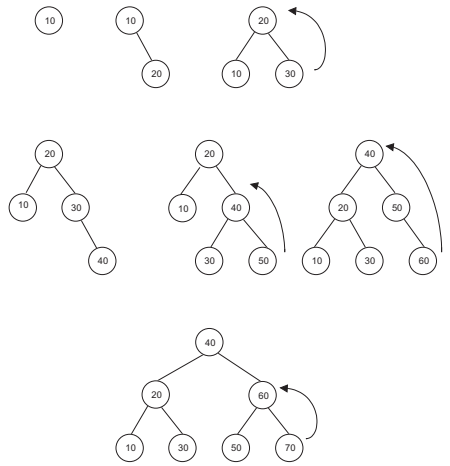


Figura 10.15: Criação de uma árvore AVL

4. Qualquer caminho entre um nodo qualquer e as suas folhas descendentes contém o mesmo número de nodos negros.

O número de nodos negros entre dois nodos quaisquer é conhecido como a altura negra. Particularmente entre a raiz e a folha de maior altura, como a altura negra da árvore. Esta propriedade garante que qualquer caminho entre um nodo e suas folhas é no máximo duas vezes maior do que qualquer outro caminho.

Desafio

- Escreva uma lista de passos (usando português mesmo) do que é necessário para construir uma árvore balanceada a partir dos seus dados constituintes.
- Escreva e implemente um programa de computador que seja capaz de receber uma árvore binária (na forma de parênteses ou na forma de cursores, tanto faz) e informe ao final se a árvore é balanceada ou não.

EXERCÍCIO RESOLVIDO 15 Escreva uma ABP considerando a ordem alfabetica usual. A raiz é a primeira letra da frase. Só inclua as letras na primeira vez em que elas ocorrerem. Melhor dizendo, despreze a repetição de letras.

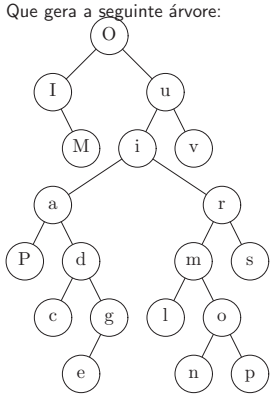
Desenhe a árvore, considerando a seguinte seqüência e desprezando brancos

```
1234567890ABCDEFGHIJKLMNOPQ
(menor)


RSTUVWXYZabcdefghijklmnopqrstuvwxyz
                                    (maior)
```


Seja a frase exemplo: 'Ouviram do Ipiranga as Margens Placidas de um Povo'

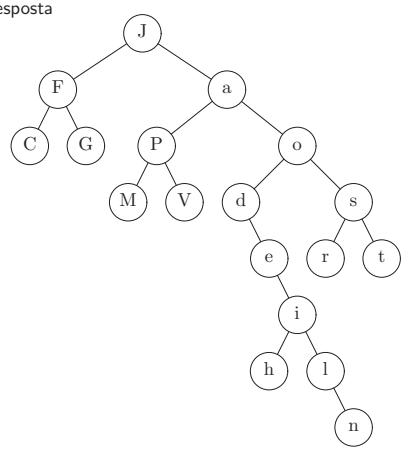
Que gera a seguinte árvore:



Considere a seguinte frase:

```
JaPodeisdaPatriaFilhosVerContenteaMaeGentil
```

Resposta



10.4.3 Caminhamento em árvores com encadeamento absoluto

Por operações em árvores entendem-se as funções que vão permitir manusear a estrutura, e estabelecer a possibilidade de usar estas ferramentas para implementação em um sem número de problemas. As operações que veremos são: o caminhamento em uma árvore, a criação de uma árvore, a inclusão de um nodo como filho direito ou esquerdo de um nodo, a exclusão de um nodo, a contagem de nodos da árvore, a função para determinar o pai, o irmão, e os filhos de um determinado nodo, as operações lógicas para identificar

um nodo como filho esquerdo ou direito, a contagem da profundidade da árvore, o estabelecimento do grau de um nodo.

Entende-se pelo processo de caminhamento em árvores, a pesquisa exaustiva em todos os nodos que compõe uma determinada árvore em uma determinada seqüência. Como veremos nos próximos capítulos, existem inúmeros algoritmos que exigem esse caminhamento. O objetivo aqui é estudar como proceder a isso do ponto de vista da árvore, deixando para o que fazer sobre cada um dos nodos, quando se estudar um algoritmo em particular. Ou seja, o que interessa nesse momento não é o que fazer com cada um dos nós, e sim como fazer para ter acesso ordenado a todos eles. No momento oportuno ver-se-á o que fazer em cada um deles, e nesse instante, o processo de caminhar sobre a árvore, será considerado conhecido.

Como em muitos outros momentos da programação, existem duas possibilidades de se fazer isso. A primeira é a iterativa, através do uso de uma pilha auxiliar. A segunda, quiçá mais elegante, é com a utilização de processos recursivos. Veremos as duas. Em função da ressalva vista anteriormente, vamos definir uma função chamada PROCESSA(X), que consistirá em processar o nodo X. Nos algoritmos deste capítulo, nada se falará sobre PROCESSA, que ficará para ser detalhada quando estudarmos as aplicações de árvores. No momento o que nos interessa é ter acesso a todos os nodos X que compõe uma determinada árvore binária

Relembrando, uma árvore é caracterizada por um apontador para o nodo raiz. Se esse apontador, já contiver o terminador (-1, por hipótese), isso significará que a árvore está vazia. Quando seu conteúdo não for vazio, no endereço indicado nesse apontador estará o nodo raiz. Esse por sua vez, conterá um apontador para o filho esquerdo (por hipótese o primeiro inteiro), um apontador para o filho direito (o segundo inteiro) e a área de dados do nodo, que pode ser formada de 1 a n inteiros, dependendo da aplicação.

caminhamento em pré-ordem

Este algoritmo não recursivo, provê uma maneira de ter acesso a todos os nodos que compõe a árvore.

Visitação de uma árvore binária em pré-ordem (sem recursividade)

```
1: inteiro função PROCESSA(inteiro X) {Antes de executar esta função, precisa ser
   criada uma pilha com capacidade para conter nodos da árvore. Vamos chamá-la de
   PILARV, com endereço P}
2: se M[X] ≠ -1 então
3:    EMPILHA(P,M[X])
4: fimse
5: enquanto ∼ PILHAVAZIA(P) faça
6:    X← DESEMPILHA (P)
7:    PROCESSA (M[X])
8:    se M[X+1]≠ -1 então
9:       EMPILHA (P,M[X+1]) {o filho direito do nodo X}
10:   fimse
11:   se M[X]≠-1 então
12:      EMPILHA (P,M[X]) {o filho esquerdo do nodo X}
13:   fimse
14: fimenquanto
15: fim {função}
```

Note que essa função faz o processamento (também chamada visitação dos nodos) na modalidade pré-ordem. Vejamos em um teste de mesa, aplicado sobre a árvore 10.4.3 para ver como os nodos são visitados.

 ©88-08, Pedro Kantek

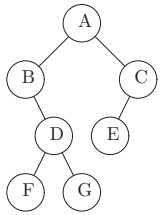


Figura 10.16: Árvore para o chinês do algoritmo 10.4.3

Passos

1. Como M[X]≠-1, empilha-se o nodo A (pilha: A)
2. Como P não é vazia, desempilha-se o nodo A (pilha vazia)
3. processa-se o nodo A (nodos processados: A)
4. Como o filho direito de A não é vazio, empilha-se o nodo C (pilha: C)
5. Como o filho esquerdo de A não é vazio, empilha-se B (pilha: CB)
6. Retorno ao enquanto: como a pilha não é vazia, desempilha-se o nodo B (pilha: C)
7. O nodo B é processado (nodos processados: AB)
8. Como o filho direito de B não é vazio, empilha-se D (pilha: CD)
9. Como o filho esquerdo de B é vazio, nada se empilha
10. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o D (pilha: C)
11. Processa-se D (nodos processados: ABD)
12. Como o filho direito de D não é vazio, empilha-se o nodo G (pilha: CG)
13. Como o filho esquerdo de D não é vazio, empilha-se F (pilha: CGF)
14. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o F (pilha: CG)
15. Processa-se F (nodos processados: ABDF)
16. Como o filho direito de F é vazio, nada é feito
17. Como o filho esquerdo de F é vazio, nada é feito
18. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o G (pilha: C)
19. Processa-se G (nodos processados: ABDFG)
20. Como o filho direito de G é vazio, nada é feito
21. Como o filho esquerdo de G é vazio, nada é feito
22. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o C (pilha vazia)

©88-08, Pedro Kantek

23. Processa-se C (nodos processados: ABDFGC)
24. Como o filho direito de C é vazio, nada é feito
25. Como o filho esquerdo de C não é vazio, empilha-se o nodo E (pilha: E)
26. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o E (pilha vazia)
27. Processa-se E (nodos processados: ABDFGCE)
28. Como o filho direito de E é vazio, nada é feito
29. Como o filho esquerdo de E é vazio, nada é feito
30. Retorno ao enquanto: como a pilha está vazia, termina-se o processamento.

Percebe-se que este algoritmo implementa o algoritmo de visitação na ordem pré-ordem, com o nodo esquerdo visitado antes do direito. Apenas como curiosidade, verifique-se como se fariam as outras 5 ordens de visitação:

Caminhamento em-ordem

Este caminhamento percorre primeiro o nodo esquerdo, depois a raiz e depois o nodo direito, recursivamente por toda a árvore. Usando uma notação mais simples que a do exemplo anterior, indicando-se uma (sub) árvore de raiz k como (k).

```
              (a)
           (b) a          (c)
  (d)       b  a (e)       c
 f d  (g)  b  a  e  (h)  c
 f d i g j b  a  e k h l c
```

Veja-se agora um algoritmo de visitação em em-ordem não recursivo.

```
1: inteiro função PROCESSA(inteiro X) {Antes de executar esta função, precisa ser
   criada uma pilha com capacidade para conter nodos da árvore. Vamos chamá-la de
   PILARV, com endereço P}
2: se M[X] ≠ -1 então
3:    X ← X
4:    empilha M[X]
5: fimse
6: enquanto enquanto ∼ PILHAVAZIA (P) faça
7:    enquanto M[X] ≠ -1 faça
8:       X ← M[X]
9:       empilha M[X]
10:   fimenquanto
11:   se PILHAVAZIA(P) então
12:      ... abandone ...
13:   fimse
14:   X ← DESEMPILHA (P)
15:   PROCESSA X
16:   se M[X+1] ≠ -1 então
17:      empilha M[X+1]
18:      X ← M[M[X+1]]
19:   fimse
20: fimenquanto
21: se (M[X] ≠ -1) então
```



```
22:    EMPILHA(P,M[X])
23: fimse
24: enquanto ∼ PILHAVAZIA(P) faça
25:    X ← DESEMPILHA (P)
26:    se (M[X] ≠ -1) então
27:       EMPILHA (P,M[X])
28:    senão
29:       PROCESSA (X)
30:    fimse
31:    se (X+1 ≠ -1) então
32:       EMPILHA (P,X+1)
33:    fimse
34: fimenquanto
35: fim função
```

Um teste de mesa do algoritmo acima.

1. Como X ≠ -1, empilha-se o nodo A (pilha: A)
2. Como P não é vazia, desempilha-se o nodo A (pilha vazia)
3. Como o filho direito de A não é vazio empilha-se C (pilha: C)
4. Empilha-se A (pilha: CA)
5. Como o filho esquerdo de A não é vazio, empilha-se B (pilha: CAB)
6. Retorno ao enquanto: como a pilha não é vazia, desempilha-se o nodo B (pilha: CA)
7. Como o filho direito de B não é vazio, empilha-se D (pilha: CAD)
8. Empilha-se B (pilha: CADB)
9. Como o filho esquerdo de B é vazio, desempilha-se B (pilha: CAD)
10. B é processado (nodos processados: B)
11. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o D (pilha: CA)
12. Como o filho direito de D não é vazio, empilha-se o nodo G (pilha: CAG)
13. Empilha-se D (pilha: CAGD)
14. Como o filho esquerdo de D não é vazio, empilha-se F (pilha: CAGDF)
15. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o F (pilha: CAGD)
16. Como o filho direito de F é vazio, nada é feito
17. Empilha-se o F (pilha: CAGDF)
18. Como o filho esquerdo de F é vazio, desempilha-se F (pilha: CAGD)
19. F é processador (nodos processados: BF)
20. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o D (pilha: CAG)
21. Processa-se G (nodos processados: ABDFG)

22. Como o filho direito de G é vazio, nada é feito
23. Como o filho esquerdo de G é vazio, nada é feito
24. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o C (pilha vazia)
25. Processa-se C (nodos processados: ABDFGC)
26. Como o filho direito de C é vazio, nada é feito
27. Como o filho esquerdo de C não é vazio, empilha-se o nodo E (pi-lha: E)
28. Retorno ao enquanto: como a pilha é não vazia, desempilha-se o E (pilha vazia)
29. Processa-se E (nodos processados: ABDFGCE)
30. Como o filho direito de E é vazio, nada é feito
31. Como o filho esquerdo de E é vazio, nada é feito
32. Retorno ao enquanto: como a pilha está vazia, termina-se o processamento.

EXERCÍCIO 157 Defina uma função que receba o endereço em "M"de uma árvore representada em alocação seqüencial. Cada nodo será identificado por seu número, seguido pelo número de filhos que ele tem. A função deve devolver a altura da árvore.

Caminhamento em pós-ordem

Neste tipo de caminhamento, visita-se primeiro o nodo esquerdo, depois o nodo direito e depois a raiz. Um exemplo de uso deste caminhamento é dado pelo pacote Tree do ambiente LATEX para desenho de árvore, que por sinal foi usado para desenhar a maioria das árvores deste texto.

Caminhamento Horizontal

Este tipo de caminhamento horizontal sugere que a árvore seja "fatiada"a começar pela raiz, e considerando todos os nodos que se encon-tram em um determinado nível antes de se passar para o próximo.

Em resumo

```
Alguns caminhamentos
PRÉ-ORDEM: RAIZ - ESQUERDA - DIREITA
EM-ORDEM: ESQUERDA - RAIZ - DIREITA
PÓS-ORDEM: ESQUERDA - DIREITA - RAIZ
HORIZONTAL: a árvore é cortada em "fatias"
```

EXERCÍCIO RESOLVIDO 16 Suponha a seguinte árvore

Pergunta-se:

1. Qual a seqüência de caminhamento em pré-ordem ? Resposta:

   ```
   57 26 16 12  7 20 40 68 72 78 99 83
   ```

2. Qual a seqüência de caminhamento em em-ordem ? Resposta:

   ```
   7 12 16 20 26 40 57 68 72 78 83 99
   ```

 ©88-08, Pedro Kantek

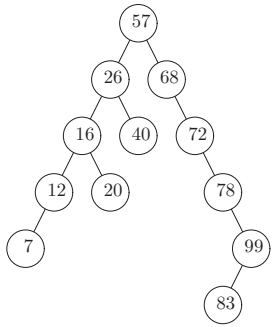


Figura 10.17: Árvore do exercício 16

3. Qual a seqüência de caminhamento em pós-ordem ? Resposta:

   ```
   7 12 20 16 40 26 83 99 78 72 68 57
   ```

4. Qual a seqüência de caminhamento em largura (horizontal)? Resposta:

   ```
   57 26 68 16 40 72 12 20 78  7 99 83
   ```

EXERCÍCIO RESOLVIDO 17 Suponha a seguinte árvore Pergunta-se:

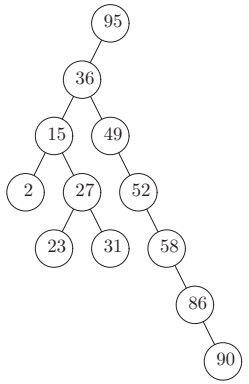


Figura 10.18: Árvore do exercício 17

1. Qual a seqüência de caminhamento em pré-ordem ? Resposta:

   ```
   95 36 15  2 27 23 31 49 52 58 86 90
   ```

©88-08, Pedro Kantek

2. Qual a seqüência de caminhamento em em-ordem ? Resposta:

   2 15 23 27 31 36 49 52 58 86 90 95

3. Qual a seqüência de caminhamento em pós-ordem ? Resposta:

   2 23 31 27 15 90 86 58 52 49 36 95

4. Qual a seqüência de caminhamento em largura (horizontal)? Resposta:

   95 36 15 49 2 27 52 23 31 58 86 90

EXERCÍCIO RESOLVIDO 18 Suponha a seguinte árvore

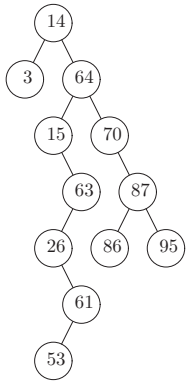


Figura 10.19: Árvore do exercício 18

Pergunta-se:

1. Qual a seqüência de caminhamento em pré-ordem ? Resposta:

   14 3 64 15 63 26 61 53 70 87 86 95

2. Qual a seqüência de caminhamento em em-ordem ? Resposta:

   3 14 15 26 53 61 63 64 70 86 87 95

3. Qual a seqüência de caminhamento em pós-ordem ? Resposta:

   3 53 61 26 63 15 86 95 87 70 64 14

4. Qual a seqüência de caminhamento em largura (horizontal)? Resposta:

   14 3 64 15 70 63 87 26 86 95 61 53

EXERCÍCIO RESOLVIDO 19 Suponha a seguinte árvore Pergunta-se:

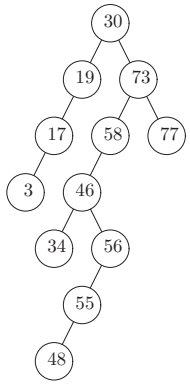


Figura 10.20: Árvore do exercício 19

1. Qual a seqüência de caminhamento em pré-ordem ? Resposta:

   30 19 17 3 73 58 46 34 56 55 48 77

2. Qual a seqüência de caminhamento em em-ordem ? Resposta:

   3 17 19 30 34 46 48 55 56 58 73 77

3. Qual a seqüência de caminhamento em pós-ordem ? Resposta:

   3 17 19 34 48 55 56 46 58 77 73 30

4. Qual a seqüência de caminhamento em largura (horizontal)? Resposta:

   30 19 73 17 58 77 3 46 34 56 55 48

10.4.4 Caminhamento usando encadeamento relativo

Outra maneira de identificar esta técnica é "caminhamento de árvores representadas através de cursores". Vamos supor uma árvore binária de pesquisa sendo representada através de uma matriz de $n \times 3$ (onde n é o número de nodos da árvore e 3 são as colunas: endereço do filho esquerdo, endereço do filho direito e conteúdo do nodo). A raiz da árvore será indicada por um campo chamado RAIZ que será passada às funções, mas nenhuma generalidade é perdida se se considerar que a raiz estará sempre na primeira linha da matriz.

Caminhamento Pré-ordem

Como sempre tem-se duas alternativas de implementação, a iterativa e a recursiva. Começando pela iterativa:

```
1: função PREORDEMI (árvore TAB, inteiro RAIZ)
2: inteiro X
```

```
3: P ← criapilha(1) {guardará números de linha da matriz}
4: empilha(P,RAIZ)
5: enquanto ∼ vazia(P) faça
6:    X ← desempilha(P)
7:    imprime TAB[X,3]
8:    se TAB[X,2] ≠ -1 então
9:       empilha(P,TAB[X,2])
10:   fimse
11:   se TAB[X,1] ≠ -1 então
12:      empilha(P,TAB[X,1])
13:   fimse
14: fimenquanto
15: fim função
```

Já a função recursiva para o caminhamento em pré-ordem é

```
1: função PREORDEMR (árvore TAB, inteiro RAIZ)
2: imprima TAB[X,3]
3: se TAB[X,1] ≠ -1 então
4:    PREORDEMR (TAB, TAB[RAIZ,1])
5: fimse
6: se TAB[X,2] ≠ -1 então
7:    PREORDEMR (TAB, TAB[RAIZ,2])
8: fimse
9: fim função
```

Note-se que a função recursiva é muito mais simples e elegante do que a versão não recursiva. A explicação para este fato está em que a função recusriva se aproveita da definição recusriva de árvore.

Caminhamento Em-ordem

Eis uma versão iterativa

```
1: função EMORDEMI (árvore TAB, inteiro RAIZ)
2: inteiro X
3: lógico FIM, CHA
4: FIM ← FALSO
5: P criapilha(1) {guardará números de linha da matriz}
6: X ← RAIZ
7: empilha(P,X)
8: enquanto ∼ FIM faça
9:    enquanto TAB[X,1] ≠ -1 faça
10:      X ← TAB[X,1]
11:      empilha(P,X)
12:   fimenquanto
13:   CHA ← VERDADEIRO
14:   enquanto CHA faça
15:      se ∼ vazia(P) então
16:         X ← desempilha (P)
17:         imprime TAB[X,3]
18:         se TAB[X,2] ≠ -1 então
19:            empilha TAB[X,2]
20:            X ← TAB[X,2]
21:            CHA ← FALSO
22:         fimse
```

 ©88-08, Pedro Kantek


```
23:      senão
24:         FIM ← VERDADEIRO
25:      fimse
26:   fimenquanto
27: fimenquanto
28: fim função
```

Uma segunda versão desta função, talvez mais simples, ainda que desperezando o conceito de programação estruturada é

```
1: função EMORDEMITERAT2 (árvore TAB, inteiro RAIZ)
2: inteiro X
3: P ← criapilha(1) {guardará números de linha da matriz}
4: X ← RAIZ
5: empilha(P,X)
6: INICIO:
7: enquanto TAB[X,1] ≠ -1 faça
8:    X ← TAB[X,1]
9:    empilha(P,X)
10: fimenquanto
11: CONTINUA:
12: se vazia(p) então
13:    encerre a função
14: fimse
15: X ← desempilha(P)
16: imprima TAB[X,3]
17: se TAB[X,2] ≠ -1 então
18:    empilha TAB[X,2]
19:    vapara INICIO
20: senão
21:    vapara CONTINUA
22: fimse
23: fim função
```

Já a função recursiva para o caminhamento em em-ordem é

```
1: função EMORDEMR (árvore TAB, inteiro RAIZ)
2: se TAB[X,1] ≠ -1 então
3:    EMORDEMR (TAB, TAB[RAIZ,1])
4: fimse
5: imprima TAB[X,3]
6: se TAB[X,2] ≠ -1 então
7:    EMORDEMR (TAB, TAB[RAIZ,2])
8: fimse
9: fim função
```

Note que a diferença das funções PREORDERR e EMORDEMR é a posição do comando imprima.

Caminhamento PÓS-ORDEM

Para este caminhamento, a função não recursiva é

```
1: função POSORDEMI (árvore TAB, inteiro RAIZ)
2: inteiro X
3: P ← criapilha(1) {guardará números de linha da matriz}
4: X ← RAIZ
```

©88-08, Pedro Kantek

```
empilha(P,X)
INICIO:
se TAB[X,2] ≠ -1 então
   empilha(P,TAB[X,2])
   troca o sinal algébrico de TAB[TAB[X,2],3]
fimse
X ← TAB[X,1]
se X ≠ -1  então
   vapara INICIO
fimse
CONTINUA:
se vazia(p)  então
   encerre a função
fimse
Y ← desempilha(P)
se TAB[Y,3] > 0 então
   imprima TAB[Y,3]
   vapara CONTINUA
senão
   inverta o sinal algébrico de TAB[Y,3]
   X ← Y
   vapara INICIO
fimse
fim função
```

Já a função recursiva para o caminhamento em pós-ordem é

```
função POSORDEMR (árvore TAB, inteiro RAIZ)
se TAB[X,1] ≠ -1 então
   POSORDEMR (TAB, TAB[RAIZ,1])
fimse
se TAB[X,2] ≠ -1 então
   POSORDEMR (TAB, TAB[RAIZ,2])
fimse
imprima TAB[X,3]
fim função
```

10.5 Aplicações de árvores

10.5.1 Encontrar repetições em listas

Seja um conjunto de números no qual se deseja conhecer quais são os números repetidos. A maneira trivial de resolver este problema é pesquisar cada número lido com todos os anteriores, através de uma comparação. Este algoritmo funciona, mas é desnecessariamente extenso em número de comparações. Uma maneira mais eficiente é usar uma árvore binária.

O primeiro número será colocado como raiz da árvore. O próximo será comparado com a raiz. Se igual, será desprezado por duplo. Se diferente e menor será colocado como filho esquerdo da raiz. Se maior será o filho direito. Com todos os demais números aplica-se a mesma regra.

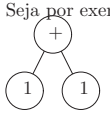
> Seja como exemplo a verificação de quais os repetidos em 8, 5, 2, 5, 9, 3, 8, 1, 3, 2 e 5. O primeiro número é lido: 8. Cria-se uma árvore que contém 8 como raiz. O segundo número é 5. Como é menor que 8 fica sendo seu filho esquerdo. Depois vem o 2 e ele fica sendo o filho esquerdo de 5. O quarto é o 5, que comparado com 8 desvia para a esquerda e comparado com 5 é desprezado por igualdade. O quinto é 9 que é colocado a direita de 8. A seguir vem o 3 que é colocado como filho direito de 2.... E assim por diante.

 ©88-08, Pedro Kantek

10.5.2 Representação de expressões aritméticas

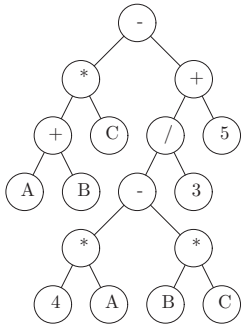
Árvores binárias são muito adequadas para representar expressões aritméticas que envolvem operadores binários (adição, subtração, multiplicação, divisão e potência, por exemplo). Para cada (sub) árvore que existe na estrutura pode-se entendê-la como uma expressão onde o filho esquerdo é o primeiro operando, o filho direito o segundo e a raiz é o operador que vai ser aplicado.

Seja por exemplo, a expressão 1+1 poderia ser



Outro exemplo: a expressão

```
((A + B) * C) - ((((4 * A) - (B * C)) / 3) + 5)
```



10.5.3 Árvores de jogos

Este tipo de aplicação, conhecida como "árvore de jogos"visa estudar a todo momento no desenvolvimento de um jogo, quais são as alternativas possíveis a cada oponente

Seja um jogo bastante trivial: "soma8"[For76, pág. 376]. Jogam 2 oponentes. Cada um pode escolher 1, 2 ou 3 pontos, que vão sendo somados a cada jogada. Um oponente não pode escolher o número anteriormente jogado pelo adversário. Quem somar 8 ganha, quem passar de 8 perde. Não há empate. Uma possível árvore poderia ser:

```
%OBS: O desenho acima precisa ser refeito
%em COREL, pois está ilegivel e gasta 13MB
```

EXERCÍCIO 158 (FOR76) - Suponha o jogo dos 6 peões. Usa-se um tabuleiro quadriculado 3 x 3. Na primeira linha são colocados 3 peões vermelhos e na linha três, 3 peões pretos. As regras:

©88-08, Pedro Kantek

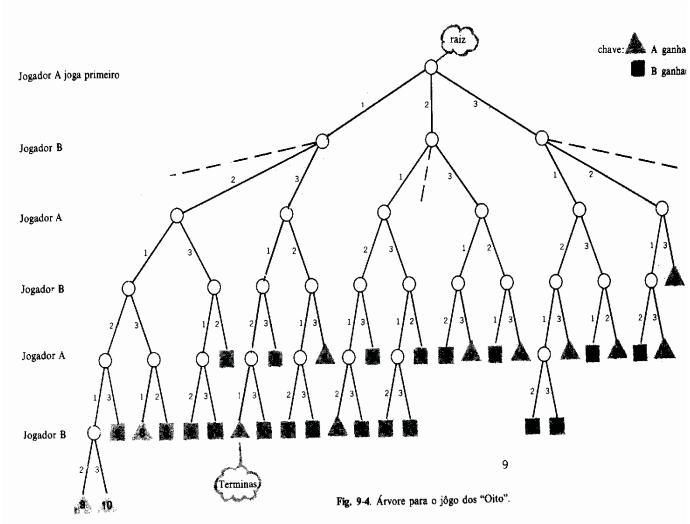


Figura 10.21: Árvore do jogo 8

- Cada peça pode mover-se 1 posição para a frente
- Ao capturar um adversário o peão se move em diagonal e o adversário é retirado do tabuleiro
- Ganha-se de uma de 3 maneiras: ou chegando a linha do adversário com 1 peão, OU bloqueando o adversário de maneira que ele não possa jogar OU capturando todas as peças do adversário.

Invente um esquema econômico de rotular as arestas da árvore, de maneira que seja possível visualizar e entender a árvore e o que ela representa e desenhe a árvore de todos os jogos possíveis, dando em cada folha quem venceu aquele trajeto.

EXERCÍCIO 159 Um jogo de crianças chamado "joquenpô"é usado às vezes como substituto para o par-ou-impar. Cada jogador pode escolher um destes objetos: tesoura, papel, pedra. Dependendo das duas opções feitas ao mesmo tempo, são aplicadas regras para determinar que foi o vencedor. Por exemplo, papel perde de tesoura (porque pode ser furado), mas ganha de pedra (porque pode embrulhá-la). Pedra ganha de tesoura.

- Descubra o jogo completo de opções possíveis

- Descubra as regras que governam o jogo
- Desenhe a árvore de todas as opções, indicando em cada folha quem foi o vencedor.

EXERCÍCIO 160 Um jogo de cartas muito comum é o 21. Suas regras:

- Jogam quantas pessoas quiserem, até o limite de 8. Uma delas é o banqueiro
- Usam-se 2 baralhos de 52 cartas.
- O baralho é embaralhado e cortado como se queira.
- Antes de começar o ciclo, cada jogador (exceto o banqueiro) faz suas apostas. Vamos estipular um mínimo de 3 fichas e um máximo de 10.
- Na seqüência, o banqueiro entrega a cada jogador 1 carta fechada e entrega uma para si aberta. Depois entrega uma segunda carta fechada, incluindo a si. Portanto, neste ponto todos tem 2 cartas. Os jogadores tem as 2 fechadas e o banqueiro tem 1 fechada e uma aberta.
- Quem ganha: o AS vale 1 ou 11, o jogador escolhe. As figuras valem 10 e as demais cartas valem o seu número. Quem fizer 21 ganha, quem passar perde, e se ninguém fizer, o mais próximo de 21 ganha.
- Natural: a seqüência; AS + figura ou 10 é chamada natural. Se um jogador faz natural e o banqueiro não faz, este paga imediatamente 1,5 vezes a aposta do jogador. Se o banqueiro faz natural (mas nenhum jogador faz) o banqueiro recolhe tudo. Se os 2 fazem natural, cada um pega o seu.
- Resolvidos os naturais, o banqueiro pergunta jogador a jogador se quer mais cartas. Cada um pode pedir quantas quiser parando quando quiser. Entretanto, se estourar 21, deve pagar as fichas ao banqueiro e cair fora.
- Atendidos os jogadores o banqueiro deve virar sua carta. Se o total for maior que 16 ele deve parar. Se tiver um AS e com ele valendo 11, ele passar de 17 (mas ficar aquém de 21), ele vale 11.
- Apostas recolhidas não voltam mais. As que ainda estiverem na mesa continuam valendo. Se o banqueiro estourar, paga a cada jogador o equivalente ao que este apostou. Se o banqueiro parou antes de 21, paga o equivalente aos que tiveram mais, recolhendo a aposta dos que tiveram menos. Havendo empate, cada um pega o seu.

Agora

1. Projete um programa de computador capaz de fazer o papel de banqueiro e jogar 21 com quantos jogadores houverem (1 a 7).
2. Cada jogador comprará fichas ao programa (banqueiro). O programa anotará (e mostrará na tela) a conta corrente de cada jogador
3. O programa mostrará as jogadas se desenvolvendo.
4. Será interrompido quando nenhum jogador quiser mais jogar.
5. Não há necessidade de trabalhar com naipes, pois estes não tem nenhum valor no jogo.
6. Não vale roubar !

PS: as regras completas do jogo estão no apêndice B de [Emm87]

EXERCÍCIO 161 Desenhe a árvore que mostre o jogo da velha em 3 x 3, identificando em cada folha, o vencedor.

Desafio

- Implemente o jogo 21 e proceder a 10 rodadas descrevendo qual o comportamento do jogo como banqueiro e depois como jogador.
- Implemente um programa que construa uma ABP com qualquer frase. Teste-a com seu nome. Para estar completo, o programa deverá imprimir a árvore criada.

10.6 Algoritmos de árvores usando alocação encadeada absoluta

Seguindo nossa estratégia de apresentar os algoritmos na sua forma mais simples na memória M, usaremos de um artifício para simplificar mais ainda os algoritmos. Cada nodo, neste tópico, será representado por 3 inteiros, a saber:

& filho esquerdo	Conteúdo do nodo (1 inteiro)	& filho direito

Mais ainda, o terminador como sempre será o inteiro -1. Finalmente, todos os endereços de nodos em M sempre apontarão para o primeiro inteiro do nodo, devendo-se deslocar tal endereço em 1 unidade para obter o conteúdo e em 2 unidades para obter o endereço do filho direito.

Também por simplicidade dos algoritmos aqui em M, não vamos considerar a existência do cabeçalho da árvore. Isto é, uma árvore será conhecida pelo endereço do seu nodo raiz. Naturalmente, em implementações mais sofisticadas ou mais realistas (que serão vistas a seguir) este cabeçalho deverá ter mais informações, tipicamente o tamanho do conteúdo, a quantidade de nodos, dados sobre inclusão, alteração e pesquisa e muita, mas muita coisa mais.

criação de uma árvore

A função de criação de uma árvore binária, pressupõe apenas a chamada da função seguinte, com endereço do PAI igual a -1.

```
1: inteiro função ARBICRI
2: X ← OBTEMEM (3)
3: se X ≠ 0 então
4:    retorne (-1)
5: senão
6:    M[X] ← M[X+1] ← M[X+2] ← -1
7:    retorne (X)
8: fimse
9: fim função
```

Caracteriza-se assim, uma árvore vazia, como aquela contendo um nodo cujos três valores são iguais a -1.

inclusão de um nodo

Esta função receberá o endereço do nodo pai, um inteiro qualquer que será o conteúdo do nodo e um indicador de esquerda (1) ou direita (2) onde deverá se fazer a inclusão. Assume-se que o endereço do nodo pai que será fornecido é válido (isto é, aponta para um nodo real). A resposta será -1 se não houver espaço para a criação do nodo, ou -2 se o nodo pai já tiver o filho solicitado preenchido. Caso não haja erro, a resposta será o endereço do novo nodo.

```
1: inteiro função INCLNODO (PAI, CONTEUDO, INDICADOR)
2: inteiro PAI, CONTEUDO, INDICADOR
```



```
3: inteiro X, RESP
4: se M[PAI+1] = -1 então
5:    M[X+1] ← CONTEUDO {a árvore estava vazia. Este passará a ser o raiz}
6:    retorna (PAI)
7: senão
8:    RESP ← 0 {RESP informará mais tarde se ocorreu erro}
9:    X ← OBTEMEM (3) {espaço para o nodo}
10:   se X ≤ 0 então
11:      RESP ← -1
12:   senão
13:      se INDICADOR = 1 então
14:         se M[PAI] = -1 então
15:            M[PAI] ← X {indicador=1 significa incluir o filho esquerdo. M[PAI]=-1
               indica que este pai não tinha filho esquerdo}
16:         senão
17:            RESP ← -2 {o pai já tinha filho esquerdo}
18:         fimse
19:      senão
20:         se M[PAI+2] = -1 então
21:            M[PAI+2] ← X {Incluir o filho como direito do pai}
22:         senão
23:            RESP ← -2 {pai já tinha filho direito}
24:         fimse
25:      fimse
26:      se RESP = 0 então
27:         M[X] ← M[X+2] ← -1
28:         M[X+1] ← CONTEUDO
29:         RESP ← X
30:      fimse
31:   fimse
32: fimse
33: fim {função}
```

Agora vamos supor uma situação diferente. Uma nova função de inclusão que admitirá que a árvore em questão está ordenada em ordenação central. Em vez de receber o endereço do pai e o conteúdo a inserir (como na anterior), ela apenas receberá o conteúdo - além do endereço da árvore, claro - e a própria função deverá:

- certificar-se de que este conteúdo não existe na árvore (por hipótese não se admitem duplos)
- localizar o ponto correto da inserção
- executar a inserção

Portanto precisa-se de uma função de que receba uma árvore ordenada centralmente e um valor e devolva .V. se o valor está na árvore e .F. se não está.

```
1: lógico função VALOREXISTE (ÁRVORE, VALOR)
2: inteiro ÁRVORE, VALOR
3: lógico RESPOSTA
4: RESPOSTA ← .F. {inicializa-se a resposta como "NÃO ESTÁ"}
5: se M[ÁRVORE+1] ≠ -1 então
6:    enquanto ÁRVORE ≠ -1 faça
7:       se VALOR = M[ÁRVORE+1] então
```

```
8:          RESPOSTA ← .V. {assinala-se a resposta correta e abandona-se o laço}
9:          saia
10:       senão
11:         se VALOR < M[ÁRVORE+1] então
12:           ÁRVORE ← M[ÁRVORE] {escolhe-se a sub-árvore esquerda}
13:         senão
14:           ÁRVORE ← M[ÁRVORE+2] {escolhe-se a sub-árvore direita}
15:         fimse
16:       fimse
17:     fimenquanto
18: fimse
19: retorne (RESPOSTA)
20: fimfunção
```

De posse da função VALOREXISTE, que é pré-requisito para ter certeza que o valor a ser incluído não existe ainda na árvore em questão, pode-se programar a inclusão. Como vimos a função recebe o endereço da árvore, e o conteúdo a inserir. Devolve -1 se a inserção não se puder fazer (ou por falta de espaço ou por duplicidade de valor), ou então devolve o endereço do nodo que contém o conteúdo recém-incluído.

```
1: inteiro função ARBIINS (ÁRVORE, CONTEUDO)
2: inteiro ÁRVORE, CONTEUDO
3: inteiro X, ESQDIR, APONTADOR
4: se M[ÁRVORE+1] = -1 então
5:   M[ÁRVORE+1] ← CONTEUDO
6:   retorne ÁRVORE
7: senão
8:   X ← OBTEMEM(3)
9:   se (X ≤ 0) ∨ (VALOREXISTE(CONTEUDO)) então
10:    retorne -1
11:  senão
12:    M[X] ← M[X+2] ← -1
13:    M[X+1] ← CONTEUDO
14:    enquanto ÁRVORE ≠ -1 faça
15:      APONTADOR ← ÁRVORE
16:      se CONTEUDO < M[ÁRVORE+1] então
17:        ÁRVORE ← M[ÁRVORE]
18:        ESQDIR ← 0
19:      senão
20:        ÁRVORE ← M[ÁRVORE + 2]
21:        ESQDIR ← 2
22:      fimse
23:    fimenquanto
24:    M[APONTADOR + ESQDIR] ← X
25:    retorne (X)
26:  fimse
27: fimse
28: fim função
```

EXERCÍCIO 162 Escreva nas caixinhas da figura 162, quais os números poderiam ocupá-las, pressupondo que a árvore sempre estaria "in order". Resposta:

 ©88-08, Pedro Kantek

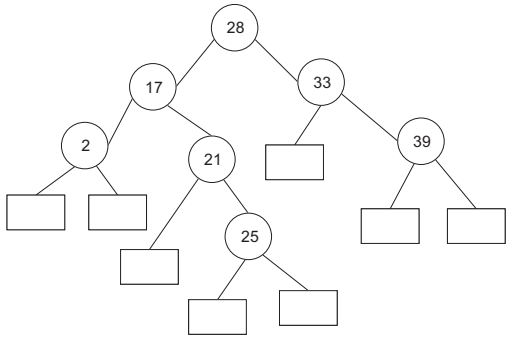


Figura 10.22: Árvore do exercício acima

fe2: 1
fd2: 3..16
fe21: 18..20
fe25: 22..24
fd25: 26 e/ou 27
fe33: 29..32
fe39: 34..38
fd39: 40..∞

exclusão de um nodo

Esta função receberá o endereço de uma árvore e um valor a excluir. Se o valor não existir, ela devolverá -1. Se existir ela o retirará da árvore, e devolverá o endereço do nodo retirado. Três casos distintos precisam ser considerados aqui, já que a exclusão não poderá deixar a árvore desordenada.

Caso 1 o nodo a excluir é uma folha. É o caso mais simples, bastará colocar o terminador no pai que aponta para este nodo.

Caso 2 O nodo tem 1 único filho. A alteração na árvore também é razoavelmente simples: bastará substituir no pai do nodo, o apontador que era para o nodo a excluir pelo endereço do único filho do nodo a excluir.

Caso 3 O nodo a excluir tem 2 filhos. Este é o caso complicado. Vamos chamar de E o nodo a excluir, P(E) seu pai, e S(E) o sucessor dele na ordem natural da árvore (Note que S(E) pode estar bem longe de E). Antes de prosseguir, deve-se notar que S(E) nunca terá filho direito, pois se o tivesse, este é que seria o seu antecessor e não o nodo E.

Esta exclusão se dá em duas etapas: Primeiro é excluído S(E), usando os casos 1 ou 2 acima, graças à regra acima enunciada. Finalmente, S(E) substitui E, e com isso E fica excluído.

©88-08, Pedro Kantek

Graças a esta característica de ter que executar duas vezes a mesma tarefa (no caso 3, uma para excluir S(E) e em seguida para excluir E), vamos optar por escrever uma rotina recursiva.

```
1: inteiro função ARBIEXC (ÁRVORE, CONTEUDO)
2: inteiro ÁRVORE, CONTEUDO
3: se (M[ÁRVORE+1] = -1 ∨ (¬ VALOREXISTE(ÁRVORE,CONTEUDO)) então
4:    retorne (-1)
5: senão
6:    PAI ← ÁRVORE
7:    ÁRVOREAUX ← ÁRVORE
8:    ESQDIR ← 0
9:    enquanto M[ÁRVOREAUX+1] ≠ CONTEUDO faça
10:      se CONTEUDO < M[ÁRVOREAUX + 1] então
11:         ESQDIR ← 0
12:         PAI ← ÁRVOREAUX
13:         ÁRVOREAUX ← M[ÁRVOREAUX]
14:      senão
15:         ESQDIR ← 2
16:         PAI ← ÁRVOREAUX
17:         ÁRVOREAUX ← M[ÁRVOREAUX + 2]
18:      fimse
19:   fimenquanto
20:   ELEMEXC ← ÁRVOREAUX
21:   se M[ELEMEXC] = -1 ∧ M[ELEMEXC+2] = -1 então
22:      M[PAI+ESQDIR] ← -1
23:      retorna (ELEMEXC)
24:   senão
25:      se M[ELEMEXC] = -1 então
26:         M[PAI+ESQDIR] ← M[ELEMEXC+2]
27:         retorna (ELEMEXC)
28:      senão
29:         se M[ELEMEXC+2] = -1 então
30:            M[PAI+ESQDIR] ← M[ELEMEXC]
31:            retorna (ELEMEXC)
32:         senão
33:            SAVE ← ELEMEXC
34:            ELEMEXC ← M[ELEMEXC + 2]
35:            enquanto M[ELEMEXC] ≠ -1 faça
36:               SAVE ← ELEMEXC
37:               ELEMEXC ← M[ELEMEXC]
38:            fimenquanto
39:            TEMPO ← RETIRANODO (ÁRVORE,M[ELEMEXC+2])
40:            M[ÁRVOREAUX+1] ← M[TEMPO + 1]
41:            retorne (ÁRVOREAUX)
42:         fimse
43:      fimse
44:   fimse
45: fimse
46: fim função
```

 ©88-08, Pedro Kantek

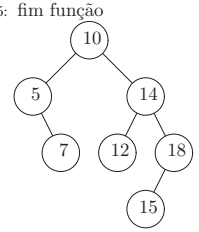
Exclusão Recursiva

Uma versão recursiva da função de exclusão em árvores binárias é apresentada a seguir, extraída de [Aho83, pág. 158 e 159]. São necessárias 2 funções: DMIN(inteiro A) que exclui o menor nodo da árvore apontada por A e devolve o seu conteúdo e DARV(inteiro C,A) que exclui o nodo de valor C da árvore A. A árvore está em pré-ordem.

```
1: inteiro função DMIN (inteiro A)
2: se M[A] = -1 então
3:    devolva M[A+1]
4:    A ← M[A+2]
5: senão
6:    devolva DMIN M[A]
7: fimse
8: fim função
```

A seguir a função de complementação da exclusão recursiva.

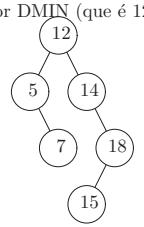
```
1: função DARV(inteiro C, A)
2: se A ≠ -1 então
3:    se C < M[A+1] então
4:       DARV (C, M[A])
5:    senão
6:       se C > M[A+1] então
7:          DARV (C, M[A+2])
8:       senão
9:          se M[A] = -1 ∧ M[A+2] = -1 então
10:            A ← -1
11:         senão
12:            se M[A] = -1 então
13:               A ← M[A+2]
14:            senão
15:               se M[A+2] = -1 então
16:                  A ← M[A]
17:               senão
18:                  M[A+1] ← DMIN M[A+2]
19:               fimse
20:            fimse
21:         fimse
22:      fimse
23:   fimse
24: fimse
25: fim função
```



Suponha-se que se quer excluir o nodo 10 na árvore acima. Chamando ADEL(10,A), executa-se a última linha da função (A tem os 2 filhos) na forma DMIN(apontador para

©88-08, Pedro Kantek

14). Esta chamada resulta em outra chamada a DMIN, agora apontando para o filho esquerdo de 14 que é 12. 12 não tem filho esquerdo, então 12 é retirado e o filho esquerdo de 14 passa a ser o filho direito de 12 que é nil (ou -1). ADEL recebe o valor devolvido por DMIN (que é 12), tira o 10 e coloca o 12 em seu lugar. A árvore fica:



10.6.1 Algoritmos de árvores usando encadeamento relativo (cursores)

As árvores vão residir agora em tabelas de 3 (ou mais) colunas. Neste caso em particular, as árvores terão a seguinte definição:

```
1: inteiro TABARV[tamanho][3]
```

O significado das colunas é:

coluna 1 Endereço da linha onde está o filho esquerdo deste nodo.

coluna 2 Endereço da linha onde está o filho direito deste nodo.

coluna 3 Conteúdo do nodo (supostamente, neste caso mais simples, apenas 1 inteiro).

Inclusão de nodo

Para esta função, considere-se a variável TABARV com número de linhas mais do que suficiente, além de uma variável ULTIMO que contém o endereço da última linha usada. Finalmente, há uma variável chamada RAIZ que contém o número da linha de TABARV que aponta para a raiz da árvore. Sem perda de generalidade pode-se pensar nesta variável contendo 1 e apontando para o início da tabela TABARV. Note que as 3 variáveis são globais.

```
 1: função INCLUIABP (inteiro ARV,VAL)
 2: ULTIMO ← ULTIMO+1
 3: TABARV[ULTIMO;]← -1,-1,VALOR
 4: ONDE ← RAIZ
 5: enquanto ONDE ≠ -1 faça
 6:    se VALOR > TABARV[ONDE;3] então
 7:       ESQDIR ← 1
 8:       PAI ← ONDE
 9:       ONDE ← TABARV[ONDE;2]
10:    senão
11:       ESQDIR ← 0
12:       PAI ← ONDE
13:       ONDE ← TABARV[ONDE;1]
14:    fimse
15: fimenquanto
16: TABARV[PAI][1+ESQDIR] ← ULTIMO
17: fimfunção
```


Exclusão de nodo

Usando a mesma lógica anteriormente exposta:

```
 1: inteiro função EXCLUIABP (inteiro VAL, RAIZ)
 2: ESQDIR ← 0
 3: AAUX ← PAI ← RAIZ
 4: enquanto VAL ≠ TABARV[AAUX;3] faça
 5:    se VAL > TABARV[AAUX;3] então
 6:       ESQDIR ← 1
 7:       PAI ← AAUX
 8:       AAUX ← TABARV[AAUX;2]
 9:    senão
10:       ESQDIR ← 0
11:       PAI ← AAUX
12:       AAUX ← TABARV [AAUX;1]
13:    fimse
14: fimenquanto
15: ELEMX ← AAUX
16: se TABARV[ELEMX;1]=-1∧TABARV[ELEMX;2]=-1 então
17:    TABARV[PAI;1+ESQDIR] ← -1
18:    devolva ELEMX
19: senão
20:    se TABARV[ELEMX;1] = -1 então
21:       TABARV[PAI;1+ESQDIR]← TABARV[ELEMX;2]
22:       devolva ELEMX
23:    senão
24:       se TABARV[ELEMX;2]=-1 então
25:          TABARV[PAI;1+ESQDIR]←TABARV[ELEMX;1]
26:          devolva ELEMX
27:       senão
28:          SAVE ← ELEMX
29:          ELEMX ← TABARV[ELEMX;2]
30:          enquanto TABARV[ELEMX;1] ≠ -1 faça
31:             ELEMX ← TABARV[ELEMX;1]
32:          fimenquanto
33:          TEMPO ← EXCLUIABP (TABARV[ELEMX;3], RAIZ)
34:          TABARV[AAUX;3] ← TABARV[TEMPO;3]
35:          devolva AAUX
36:       fimse
37:    fimse
38: fimse
39: fimfunção
```

EXERCÍCIO RESOLVIDO 20 Suponha uma árvore binária de pesquisa, montada em uma tabela cuja descrição é

```
estrutura NODO
   inteiro FE,    inteiro FD,   inteiro CONTEUDO
fim{estrutura}
NODO ARVORE[100]
```

já corretamente preenchida, e na qual a raiz ocupa a primeira posição. Note que a árvore pode conter lixo o que inviabiliza uma busca sequencial. Escreva um algoritmo que pesquise esta árvore e imprima o maior valor que existe na árvore.

Para exercitar e testar a sua resposta, use a árvore A abaixo (a raiz está na primeira linha e a matriz está "deitada"para facilitar...) Note que as linhas em negrito são lixo, não fazem parte da árvore, nem devem ser consideradas. Elas estão aí para atrapalhar os malandros que pretendem fazer uma busca seqüencial na estrutura.

Por conforto, substitui-se -1"por xx. É a mesma coisa.

FE	5	12	xx	**1**	6	xx	9	**4**	10	xx	xx	xx	**xx**	7
FD	3	xx	2	**2**	14	xx	xx	**5**	11	xx	xx	xx	**xx**	xx
C	37	80	42	**3**	20	2	27	**6**	23	21	26	63	**91**	28
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Uma possível resposta

```
1: inteiro função EX1 (arvore A)
2: inteiro X
3: X ← 1
4: enquanto A[X;2] ≠ -1 faça
5:    X ← A[X;2]
6: fimenquanto
7: imprima A[X;3]
8: fim
```

o algoritmo acima pedido, deve responder 80.

A seguir, a versão do mesmo algoritmo, agora usando uma matriz de registros

```
1: inteiro função EX1A (arvore A)
2: inteiro X
3: X ← 1
4: enquanto A[X].FD ≠ -1 faça
5:    X ← A[X].FD
6: fimenquanto
7: imprima A[X].CONTEUDO
8: fim
```

EXERCÍCIO RESOLVIDO 21 Suponha uma árvore binária de pesquisa, montada em uma tabela cuja descrição é igual à da questão anterior, já corretamente preenchida, e na qual a raiz ocupa a primeira posição. Note que a árvore pode conter lixo o que inviabiliza uma busca seqüencial. Escreva um algoritmo que pesquise esta árvore e imprima a quantidade de nodos que existem na árvore. Note que as linhas em negrito são lixo, não fazem parte da árvore, nem devem ser consideradas. Elas estão aí para atrapalhar os malandros que pretendem fazer uma busca seqüencial na estrutura.

Por conforto, substitui-se -1"por xx. É a mesma coisa.

FE	5	12	xx	**1**	6	xx	9	**4**	10	xx	xx	xx	**xx**	7
FD	3	xx	2	**2**	14	xx	xx	**5**	11	xx	xx	xx	**xx**	xx
C	37	80	42	**3**	20	2	27	**6**	23	21	26	63	**91**	28
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

o algoritmo acima pedido, deve responder 11.

Duas possíveis respostas: Alternativa 1 (solução recursiva)

```
1: função CONTA (inteiro RAIZ)
2: se A[RAIZ;1] ≠ -1 então
3:    SOMA++
4:    CONTA (A[RAIZ;1])
5: fimse
6: se A[RAIZ;2] ≠ -1 então
```

 ©88-08, Pedro Kantek


```
7:    SOMA++
8:    CONTA (A[RAIZ;2])
9: fimse
10: fimfunção
```

```
1: função PRINCIPAL
2: inteiro SOMA
3: SOMA ← 0
4: CONTA(1)
5: imprima SOMA
6: fimfunção
```

Alternativa 2: (sem recursividade)

```
1: função CONTA2 (arvore A)
2: inteiro SOMA ← 0
3: inteiro VETX[100]
4: inteiro COMECO, FIM
5: VETX[1] ← 1
6: COMECO ← 2
7: FIM ← 1
8: enquanto FIM < COMECO faça
9:    se A[VETX[FIM];1] ≠ -1 então
10:      VETX[COMECO]← A[VETX[FIM];1]
11:      COMECO++
12:   fimse
13:   se A[VETX[FIM];2] então
14:      VETX[COMECO]← A[VETX[FIM];2]
15:      COMECO++
16:   fimse
17:   FIM++
18: fimenquanto
19: imprima FIM
20: fimfunção
```

EXERCÍCIO RESOLVIDO 22 Suponha uma árvore binária de pesquisa, montada em uma tabela cuja descrição é

```
estrutura NODO
   inteiro FE,    inteiro FD,    inteiro CONTEUDO
fim{estrutura}
NODO ARVORE[100]
```

já corretamente preenchida, e na qual a raiz ocupa a primeira posição. Note que a árvore pode conter lixo o que inviabiliza uma busca seqüencial. Escreva um algoritmo que pesquise esta árvore e imprima a altura da árvore.

Para a árvore A abaixo (a raiz está na primeira linha e a matriz está "deitada"para facilitar...). Note que as linhas em negrito são lixo, não fazem parte da árvore, nem devem ser consideradas. Elas estão aí para atrapalhar os malandros que pretendem fazer uma busca seqüencial na estrutura. Nesta árvore, o algoritmo acima pedido, deve responder 5.

Por conforto, substitui-se -1"por xx. É a mesma coisa.

FE	5	12	xx	**1**	6	xx	9	**4**	10	xx	xx	xx	**xx**	7
FD	3	xx	2	**2**	14	xx	xx	**5**	11	xx	xx	xx	**xx**	xx
C	37	80	42	**3**	20	2	27	**6**	23	21	26	63	**91**	28
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

©88-08, Pedro Kantek

Uma possível resposta:

```
1: função ALTU (inteiro ALTURA, inteiro RAIZ)
2: se  A[RAIZ;1] ≠ -1 então
3:    se ALTURA+1 > MAIOR então
4:       MAIOR ← ALTURA + 1
5:    fimse
6:    ALTU (ALTURA+1, A[RAIZ;1])
7: fimse
8: se A[RAIZ;2] ≠ -1 então
9:    se ALTURA+1 > MAIOR então
10:      MAIOR← ALTURA + 1
11:   fimse
12:   ALTU (ALTURA+1, A[RAIZ;2])
13: fimse
14: fimfunção
```

```
1: função PRINCIPAL
2: inteiro MAIOR
3: MAIOR ← 0
4: CONTA(0, 1)
5: imprima MAIOR
6: fimfunção
```

EXERCÍCIO RESOLVIDO 23 Suponha uma árvore binária de pesquisa, montada em uma tabela cuja descrição é

```
estrutura NODO
   inteiro FE,    inteiro FD,    inteiro CONTEUDO
fim{estrutura}
NODO ARVORE[100]
```

já corretamente preenchida e na qual a raiz ocupa a primeira posição. Note que a árvore pode conter lixo o que inviabiliza uma busca seqüencial. Escreva um algoritmo que receba um valor que está na árvore. Se o nodo tiver irmão, a função deve devolver qual o conteúdo do irmão.

Para a árvore A abaixo (a raiz está na primeira linha e a matriz está "deitada"para facilitar...)

Por conforto, substitui-se -1"por xx. É a mesma coisa.

FE	5	12	xx	**1**	6	xx	9	**4**	10	xx	xx	xx	**xx**	7
FD	3	xx	2	**2**	14	xx	xx	**5**	11	xx	xx	xx	**xx**	xx
C	37	80	42	**3**	20	2	27	**6**	23	21	26	63	**91**	28
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

o algoritmo acima pedido, se chamado com 2, deve responder 28. Se chamado com 80, não deve responder nada.

Uma possível resposta

```
1: inteiro função IRMAO (arvore A, inteiro X)
2: inteiro R, PAI
3: R ← 1
4: PAI ← -1
5: enquanto X ≠ A[R;3] faça
6:    PAI ← R
7:    se X > A[R;3] então
```

 ©88-08, Pedro Kantek


```
8:       R← A[R;2]
9:    senão
10:      R← A[R;1]
11:   fimse
12: fimenquanto
13: se A[PAI;1] = X ∧ A[PAI;2] ≠ -1 então
14:   devolva A[A[PAI;2];3]
15: senão
16:   se A[PAI;2] = X ∧ A[PAI;1] ≠ -1 então
17:      devolva A[A[PAI;1];3]
18:   fimse
19: fimse
20: fimfunção
```

EXERCÍCIO RESOLVIDO 24 Suponha uma árvore binária de pesquisa, montada em uma tabela cuja descrição é

```
estrutura NODO
   inteiro FE,    inteiro FD,    inteiro PAI
   inteiro CONTEUDO
fim{estrutura}
NODO ARVORE[100]
```

já corretamente preenchida, e na qual a raiz ocupa a primeira posição. Note que a árvore pode conter lixo o que inviabiliza uma busca seqüencial. Escreva um algoritmo que pesquise esta árvore e imprima a amplitude dos dados que compõe a árvore (o maior dado menos o menor dado)

Para a árvore A abaixo (a raiz está na primeira linha e a matriz está "deitada"para facilitar...)

Por conforto, substitui-se -1"por xx. É a mesma coisa.

FE	5	12	xx	**1**	6	xx	9	**4**	10	xx	xx	xx	**xx**	7
FD	3	xx	2	**2**	14	xx	xx	**5**	11	xx	xx	xx	**xx**	xx
C	37	80	42	**3**	20	2	27	**6**	23	21	26	63	**91**	28
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

o algoritmo acima pedido, deve responder 78 (o hachuriado é lixo)

Uma possível resposta

```
1: função AMPLITUDE (arvore A)
2: inteiro X, MAI, MEN
3: X← 1
4: enquanto A[X;2] ≠ -1 faça
5:    X← A[X;2]
6: fimenquanto
7: MAI← A[X;4]
8: X ← 1
9: enquanto A[X;1] ≠ -1 faça
10:   X ← A[X;1]
11: fimenquanto
12: MEN ← A[X;4]
13: imprima MAI-MEN
14: fim função
```

©88-08, Pedro Kantek

EXERCÍCIO RESOLVIDO 25 Suponha uma árvore binária de pesquisa, montada em uma tabela cuja descrição é

```
estrutura NODO
   inteiro FE,    inteiro FD,    inteiro PAI
   inteiro CONTEUDO
fim{estrutura}
NODO ARVORE[100]
```

já corretamente preenchida, e na qual a raiz ocupa a primeira posição. Note que a árvore pode conter lixo o que inviabiliza uma busca seqüencial. Escreva um algoritmo que receba um certo número de linha (que contém um valor válido, que pertence a árvore) e imprima os netos deste nodo.

Por conforto, substitui-se -1"por xx. É a mesma coisa.

FE	5	12	xx	**1**	6	xx	9	**4**	10	xx	xx	xx	**xx**	7
FD	3	xx	2	**2**	14	xx	xx	**5**	11	xx	xx	xx	**xx**	xx
PAI	xx	3	1	**xx**	1	5	14	**xx**	7	9	9	2	**xx**	5
C	37	80	42	3	20	2	27	6	23	21	26	63	91	28
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

o algoritmo acima pedido, se chamado com 20, deve imprimir 27. Se chamado com 37, deve imprimir 28 e 80. Se chamado com 27, deve imprimir 21 e 26.

Uma possível resposta

```
1: inteiro função NETOS (arvore A, inteiro V)
2: inteiro I
3: I← 1
4: enquanto A[I;4] ≠ V faça
5:    se A[I;4] < V então
6:       I← A[I;2]
7:    senão
8:       I← A[I;1]
9:    fimse
10: fimenquanto
11: se A[I;1] ≠ -1 então
12:    se A[A[I;1];1] ≠ -1 então
13:       imprima A[A[A[I;1];1];4]
14:    fimse
15:    se A[A[I;1];2] ≠ -1 então
16:       imprima A[A[A[I;1];2];4]
17:    fimse
18: fimse
19: se A[I;2] ≠ -1 então
20:    se A[A[I;2];1] ≠ -1 então
21:       imprima A[A[A[I;2];1];4]
22:    fimse
23:    se A[A[I;2];2] ≠ -1 então
24:       imprima A[A[A[I;2];2];4]
25:    fimse
26: fimse
27: fim função
```

Desafio

- Implementar o algoritmo de pesquisa de um valor em uma ABP
- Implementar o algoritmo de criação de árvore binária de pesquisa vazia
- Implementar o algoritmo de inclusão de novos nodos em ABPs. Usar a função de pesquisa acima projetada a fim de evitar nodos duplos na árvore.
- Implementar de maneira recursiva o algoritmo de exclusão de nodos.

Exercícios de árvores ____ / ____ / ____

10.6.2 Memória M

Sempre que a memória M for citada nos exercícios, considere a memória sendo simulada em um vetor de inteiros, assim definido:

```
int M [100]
```

O endereço inicial de M é sempre 0 (afinal, é uma memória) e o último é o endereço 99.

EXERCÍCIO 163 Escreva o algoritmo de uma função, de nome AVO que usando M, receba 3 parâmetros (o endereço em M do descritor de uma árvore binária de pesquisa - ABP; e dois valores V1 e V2, que são valores que estão presentes na dita árvore) e devolva .T. se V1 e V2 forem netos do mesmo nodo avô, e .F. senão. O formato dos nodos é

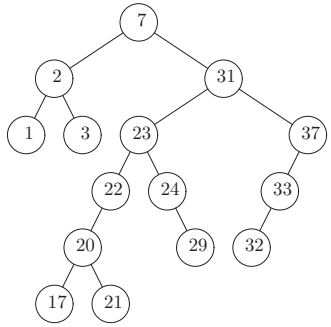
Endereço em M do filho esquerdo	Idem, do filho direito	Valor inteiro do nodo

O descritor da árvore está em `M[0]`, como sempre. Lembre que NÃO é possível qualquer tipo de processamento seqüencial em M, pois por definição ela é uma memória compartilhada e nenhuma suposição pode ser feita sobre seu conteúdo sem que os apontadores explicitamente o informem.

Para efeito de teste de sua função, considere o seguinte caso exemplo

M	0	1	2	3	4	5	6	7	8	9
0	4	0	0	0	27	7	7	17	10	31
1	14	-1	37	0	30	-1	33	20	34	23
2	24	-1	22	0	45	40	20	51	48	2
3	-1	-1	32	0	-1	37	24	-1	-1	29
4	-1	-1	21	0	0	-1	-1	17	-1	-1
5	3	-1	-1	1	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0

Que, devidamente desenhado nos mostra a árvore:



Aqui, se a função fosse chamada com 0,20,29, ela deveria responder com .V., pois os nodos 20 e 29 tem o mesmo avô (que é o 23). Se chamada com 0, 21 e 22, deveria responder com .F., pois os nodos 21 e 22 não têm o mesmo avô.

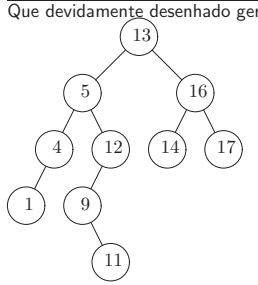
Preocupe-se (como sempre) com as condições de contorno: Pode ser que algum dos valores fornecidos, embora estejam presentes na árvore, NÃO tenham avô.

EXERCÍCIO 164 Pergunta teórica: No que uma ABP completamente degenerada difere de uma lista simplesmente encadeada ?

EXERCÍCIO 165 Escreva o algoritmo de uma função de nome FOLHAS que receba o endereço do descritor de uma ABP implementada sobre M e imprima os valores de todos os nós folha. Para teste desta função, considere o exemplo:

M	0	1	2	3	4	5	6	7	8	9
0	4	0	0	0	7	21	13	15	10	5
1	18	-1	12	0	0	29	-1	4	-1	32
2	9	26	37	16	0	0	-1	-1	14	-1
3	-1	1	-1	-1	11	0	0	-1	-1	17
4	0	0	0	0	0	0	0	0	0	0

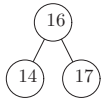
Que devidamente desenhado gera a árvore:



Se a função fosse chamada assim `FUNÇÃO(0)`, deveria imprimir os valores 1, 11, 14 e 17 que são as folhas da árvore.

Se a mesma função fosse chamada na forma `FUNÇÃO(21)` deveria imprimir os valores 14 e 17, pois neste caso a árvore pequisada seria apenas:

 ©88-08, Pedro Kantek

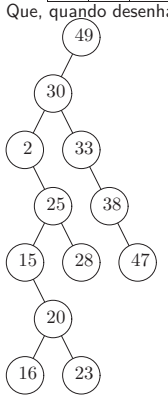


EXERCÍCIO 166 Escreva o algoritmo de uma função, de nome SOBRINHO que receba 3 parâmetros (uma tabela de descritores contendo a árvore, com a raiz na primeira linha e dois valores V1 e V2, que são valores que estão presentes na dita árvore) e devolva .T. se V1 for sobrinho de V2, e .F. senão.

Para testar seu algoritmo, considere a tabela CASO1:

1	2	-1	49
2	4	3	30
3	-1	7	33
4	-1	5	2
5	8	6	25
6	-1	-1	28
7	-1	12	38
8	-1	9	15
9	11	10	20
10	-1	-1	23
11	-1	-1	16
12	-1	-1	47

Que, quando desenhamos a árvore gera:



Neste caso, chamando `SOBRINHO(CASO1,25,33)` é .V. (25 é sobrinho de 33), enquanto `SOBRINHO(CASO1,20,15)` é .F. (20 não é sobrinho de 15).

EXERCÍCIO 167 Pergunta teórica: Suponha duas estruturas que tem o mesmo número de nodos. A primeira, é uma árvore binária COMPLETAMENTE DEGENERADA, de altura H. A segunda é uma lista simplesmente encadeada de K nodos. Como pode-se relacionar os valores de H e K ?

EXERCÍCIO 168 Escreva o algoritmo de uma função de nome DESCEND que receba o endereço do descritor de uma ABP implementada sobre M, e também o valor V de um

©88-08, Pedro Kantek

determinado nodo que está presente na árvore e imprima os valores de todos os nós que descendem do nodo cujo valor é V.

Implemente também o mesmo algoritmo para um caso em que a função receba uma tabela de cursores contendo a árvore (como sempre, a raiz está na linha 1 da tabela).

Por exemplo, sejam os seguintes casos:

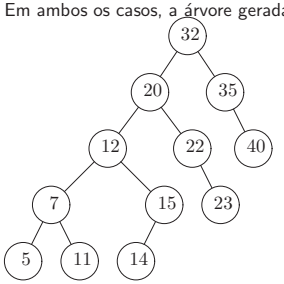
Usando cursores

1	2	6	32
2	4	3	20
3	-1	5	22
4	9	7	12
5	-1	-1	23
6	-1	10	35
7	8	-1	15
8	-1	-1	14
9	11	12	7
10	-1	-1	40
11	-1	-1	5
12	-1	-1	11
13	22	-1	12
14	1	2	1

Usando memória M

M	0	1	2	3	4	5	6	7	8	9
0	5	0	0	0	0	8	22	32	16	11
1	20	-1	19	22	0	0	32	26	12	-1
2	-1	23	-1	37	35	0	29	-1	15	-1
3	-1	14	40	43	7	0	0	-1	-1	40
4	-1	-1	5	-1	-1	11	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0

Em ambos os casos, a árvore gerada é



Se a função fosse chamada com 32, deveria imprimir todos os valores da árvore (exceto o 32). Se fosse chamada com 7, deveria imprimir o 5 e o 11. Se chamada com o valor 40, não deveria imprimir nada.

EXERCÍCIO 169 Suponha uma lista duplamente encadeada em M, com o seguinte formato:

 ©88-08, Pedro Kantek

descritores

endereço em M do primeiro nodo	Endereço em M do último nodo

nodos

endereço em M do próximo nodo	Endereço em M do nodo anterior	uma valor inteiro qualquer

Escreva o algoritmo de uma função de nome ACHARAP que receba um valor x qualquer, e devolva .T. se ele estiver presente na lista ou .F. senão. Os valores que estão na lista são únicos e estão em ordem crescente. Você TEM que fazer uso da característica de duplo encadeamento para MINIMIZAR a quantidade necessária de acessos à estrutura antes de concluir pela existência ou não de M na lista.

EXERCÍCIO 170 Defina uma função que receba o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, e devolva a altura da árvore. A função deve devolver -1 se a árvore estiver vazia, 0 se a árvore contiver apenas o nodo raiz, 1 se tiver altura 1 e assim por diante.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 171 Defina uma função que receba o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, e devolva o valor do 6^o elemento obtido no caminhamento em ordem central (in-order). Se este elemento não existir, a função deve devolver -1.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 172 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o valor de um dado nodo que com certeza existe na árvore (não precisa testar) e devolva o endereço do nodo sucessor ao nodo que contém este valor. (DICA: se o nodo em questão tem filho direito, a busca deve se dar por aí. Se ele não tem, o sucessor é o pai deste nodo)

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 173 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, Uma determinada altura intermediária, e imprima todos os valores dos nodos que estão nessa altura intermediária. Se essa altura não existir, a função não imprimirá nada. A altura deve se medir a partir da raiz. Assim, se o parâmetro for zero, o valor da raiz deve ser impresso. Se for 1, imprimir os valores dos filhos da raiz. Se for 2, imprimir os valores dos netos da raiz,... e assim por diante.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 174 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o valor de um dado nodo que com certeza existe na árvore (não precisa testar) e devolva o valor do nodo predecessor ao nodo que contém este valor. (DICA: se o nodo em questão tem filho esquerdo, a busca deve se dar descendo uma vez à esquerda e depois sempre à direita, até achar o terminador. Se não tem, seu predecessor é o nodo pai).

Implemente também a versão usando uma matriz de cursores.

©88-08, Pedro Kantek

EXERCÍCIO 175 Defina uma função que receba o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, e imprima, para cada valor de nodo da árvore, uma lista de todos os valores possíveis para filhos esquerdo e direito ainda não existentes. A lista de nodos deve ser obtida pelo caminhamento in-order, isto é, em ordem crescente.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 176 Qual a complexidade esperada, em função de N (N é o número de nodos da árvore) esperada para a função que localiza o endereço de um nodo que contenha um dado valor V, para uma árvore de altura mínima, ou seja completamente preenchida ?

EXERCÍCIO 177 Qual a complexidade esperada, em função de N (N é o número de nodos da árvore) esperada para a função que localiza o endereço de um nodo que contenha um dado valor V, para uma árvore de altura máxima, ou seja degenerada ?

EXERCÍCIO 178 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o endereço de um dado nodo nessa árvore, a quem chamaremos PAI Um valor V a ser inserido como filho desse pai, e Insira esse valor V como filho de PAI, SE O LUGAR JÁ NÃO ESTIVER OCUPADO. Se estiver, devolver-1.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 179 Qual a organização (isto é, como os dados deveriam ser dispostos na seqüência de entrada), que os dados fornecidos à função de inserção de ABP para gerar uma árvore de altura mínima ?

EXERCÍCIO 180 Qual a organização (isto é, como os dados deveriam ser dispostos na seqüência de entrada), que os dados fornecidos à função de inserção de ABP para gerar uma árvore de altura máxima, ou seja, completamente degenerada ?

EXERCÍCIO 181 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o endereço de um dado nodo que com certeza existe na árvore (não precisa testar) e devolva o grau desse nodo.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 182 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o valor de um dado nodo que com certeza existe na árvore (não precisa testar) e devolva o valor do irmão desse nodo, se existir.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 183 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o valor de um dado nodo que com certeza existe na árvore (não precisa testar) e imprima todos os valores dos descendentes do nodo dado, em in-order.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 184 Defina uma função que receba: o endereço do descritor de uma ABP (árvore binária de pesquisa), definida sobre a memória M, o valor de um dado nodo que com certeza existe na árvore (não precisa testar) e imprima todos os valores dos seus ascendentes.

Implemente também a versão usando uma matriz de cursores.

 ©88-08, Pedro Kantek

EXERCÍCIO 185 Defina uma função que receba o endereço do descritor de uma ABP definida sobre M, e dois valores conhecidos como V1 e V2, sendo que ambos existem sempre na árvore (Não precisa testar) e:

- imprima SIM, se V1 for pai de V2
- imprima SIM, se V1 for irmão direito de V2
- imprima SIM, se V1 for irmão esquerdo de V2
- imprima SIM, se V1 for ascendente de V2
- imprima SIM, se V1 for descendente de V2
- imprima SIM, se V1 for filho de V2
- imprima SIM, se V1 for neto de V2
- imprima SIM, se V1 for raiz e V2 for folha

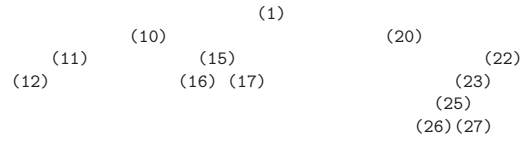
Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 186 Defina uma função que receba o endereço do descritor de uma ABP definida sobre M, e gere outra completamente degenerada contendo os mesmos nodos.

Implemente também a versão usando uma matriz de cursores.

EXERCÍCIO 187 Qual a regra que permite caracterizar uma árvore binária como sendo uma ABP ?

EXERCÍCIO 188 Suponha uma ABP, a quem chamaremos ABPP (de pre-order) construída de tal maneira que seu caminhamento em pré-order coloque todos os valores dos nodos em ordem ascendente, por exemplo:



Note que esta árvore gera o seguinte caminhamento em pre-order 1-10-11-12-15-16-17-20-22-23-25-26-27 Que regra poderia ser definida para certificar que uma AB é de fato uma ABPP ?

EXERCÍCIO 189 Suponha uma árvore implementada sequencialmente na forma de uma matriz de cursores, numerada a partir de zero, como segue

	filhoesquerdo	filhodireito	pai	Co
0	1	-1	-1	A
1	2	3	0	B
2	-1	-1	1	C
3	4	5	1	D
4	-1	-1	3	E
5	6	7	3	F
6	-1	-1	5	G
7	-1	-1	5	H

Pede-se:

1. Qual é o nodo raiz ? Porquê ?
2. Desenhe a árvore na forma convencional (com nodos e arestas)

©88-08, Pedro Kantek

3. Qual a seqüência de conteúdos obtidos numa visitação in-order ?
4. Idem, para pré-order
5. Idem, para pós-order
6. Quem são as folhas da árvore ?
7. Quem são os nodos de altura = 2 ?
8. E os de altura = 4 ?
9. E e G são irmãos ? Porquê ?
10. B é avô de G ? Porquê ?
11. Qual o grau da árvore ?
12. Qual a altura da árvore ?

EXERCÍCIO 190 Qual a única condição em que uma árvore de Huffman não é completa ? (ou seja, quando há ao menos um nodo que não tenha os dois filhos).

EXERCÍCIO 191 Imaginando a árvore do exercício anterior em pé-ordem, qual o algoritmo que localiza o nodo mais à esquerda da árvore ? (Aquele que numa visitação in-order seria o primeiro nodo a ser visitado).

EXERCÍCIO 192 Supondo a seguinte árvore de Huffman, numerada a partir de zero

	filhoesquerdo	filhodireito	pai	Conteúdo
0	1	2	-1	-5
1	3	4	0	-4
2	-1	-1	0	68
3	5	6	1	-3
4	7	8	1	-2
5	-1	-1	3	65
6	-1	-1	3	66
7	-1	-1	4	67
8	9	10	4	-1
9	-1	-1	8	69
10	-1	-1	8	70

Na qual os números de conteúdo que sãop negativos correspondem a números de nodos intermediários e os que são positivos correspondem ao código ASCII das folhas, por exemplo o conteúdo 65 corresponde à letra "A", pede-se

1. Desenhe a árvore
2. Quais os valores que serão codificados usando esta árvore para compressão correspondentes às letras A, B, C, D, E e F
3. Qual o algoritmo que recebendo o número de linha de uma letra (por exemplo, linha nº 7, correspondendo à letra C), devolve o seu string de bits comprimidos ?
4. Qual o algoritmo que dado um string de n bits (n fornecido) devolve a seqüência de letras que o compõe ?

EXERCÍCIO 193 Suponha uma árvore, usando cursores, numerada a partir de zero:

 ©88-08, Pedro Kantek

	filhoesquerdo	filhodireito	Conteúdo
0	1	2	A
1	3	4	B
2	-1	5	D
3	-1	-1	I
4	-1	-1	H
5	6	-1	C
6	7	8	E
7	-1	-1	G
8	-1	-1	F

Pede-se qual o algoritmo que leia esta árvore e gere outra, na forma de uma tabela, agora com 4 colunas, na qual a coluna adicional contém o pai de cada nodo. Obs: o nodo de linha zero é o raiz.

EXERCÍCIO 194 Suponha uma árvore representada sequencialmente na forma de uma matriz de N × 4. N é uma variável que já está pré-definida no algoritmo, e as 4 colunas representam o fe, fd, pai e conteúdo, respectivamente. Não há linhas não ocupadas na matriz e a raiz está na linha 0.

1. Escreva o algoritmo de uma função que insere um novo nodo (cujo conteúdo é dado ao algoritmo), como último filho à direita da árvore.
2. Escreva o algoritmo de uma função que dado um número de linha de um nodo folha (não é necessário testar) exclua-o da árvore
3. Escreva o algoritmo de uma função que imprima os conteúdos de todos os nodos que não são folhas.

EXERCÍCIO 195 Qual seria o tamanho necessário em linhas de uma matriz (de 4 colunas) para a representação sequencial de huffman para uma árvore de 10 folhas ?

EXERCÍCIO 196 Escreva o algoritmo de uma função que receba uma matriz sequencial contendo a representação de uma árvore binária e exclua o conteúdo do nodo raiz (este estava e precisará ficar na linha 0 da matriz).

EXERCÍCIO 197 Seja uma árvore de pesquisa binária cujo lay-out é filhoesquerdo, filhodireito, Conteúdo. Escreva o algoritmo de uma função que receba um número que não está na árvore e devolva o número da linha que contém o pai que deverá receber este novo filho.

EXERCÍCIO 198 Seja a árvore binária

	filhoesquerdo	filhodireito	pai	Conteúdo
0	1	-1	-1	P
1	2	3	0	R
2	-1	-1	1	T
3	4	5	5	G
4	-1	-1	3	L
5	6	7	3	U
6	-1	-1	5	W
7	-1	-1	5	S

Que erro existe nesta tabela acima ?

EXERCÍCIO 199 Escreva sobre o método de Huffman. Descreva o algoritmo. Diga porque ele pode diminuir o tamanho dos dados a representar. Comente sobre o fato dele tambem ser um criptografador de dados. Porquê ?

©88-08, Pedro Kantek

EXERCÍCIO 200 Suponha uma árvore em representação sequencial, como segue

```
estrutura {inteiro fe, fd, pai, conteudo} mapaarv
mapaarv ARVORE[30]
```

Suponha que as 30 posições da tabela estão ocupadas, isto é, a árvore possui 30 nodos. O nodo raiz deve ser pesquisado (não se sabe a priori onde ele está, mas se sabe que é o único nodo que não tem pai). Escreva o algoritmo de uma função que devolva o número da linha correspondente ao nodo que tiver MAIOR ALTURA, isto é, aquele que estiver mais longe da raiz. Por hipótese, só haverá um nodo com esta altura.

EXERCÍCIO 201 Suponha a seguinte estrutura de dados

```
estrutura {alfanum[40] NOME
           pflutuante SALARIO
           inteiro IDADE} PESSOA
PESSOA ARQUIVO[100]
```

O conjunto originalmente está ordenado por ordem alfabética de NOME. Escreva um algoritmo QUALQUER que classifique o conjunto em ordem DECRESCENTE por salário. Ao final, comente:

1. O algoritmo que você escreveu é estável ? Porque ?
2. Qual a complexidade do algoritmo ?

EXERCÍCIO 202 Suponha uma árvore binária descrita pela seguinte estrutura ARV1

```
estrutura {inteiro filhoesq, filhodir, conteudo} ARV1
```

E suponha uma segunda árvore binária descrita pela estrutura ARV2

```
estrutura {inteiro filhoesq, filhodir, prof, conteudo} ARV2
```

Defina uma função que leia uma arvore no formato ARV1 e imprima a mesma árvore no formato ARV2. Ambas as estruturas ocorrem 100 vezes, e o número real de linhas ocupadas está na variável global NL. O que a função deve calcular é o campo PROF, que significa a profundidade (nível) de cada nodo. O nodo raiz está na linha cujo número é o conteúdo da variavel global RAIZ. (Nota NL e RAIZ já estão definidas, com valor correto, e devem ser usadas à vontade).

EXERCÍCIO 203 Suponha um conjunto de registros ordenados por nome cujo conteúdo é aquele definido pela estrutura estrutura

```
estrutura ALUNO
alfanum NOME[35], pflutuante NOTA
fim-estrutura
```

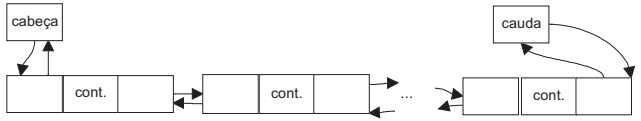
ALUNO ocorre 240 vezes. Escreva uma função que leia tal conjunto de dados e imprima a MEDIANA do conjunto de notas.

Def: MEDIANA em uma distribuição ordenada é o valor central da distribuição. Se houverem 2 valores centrais, a mediana é a média entre os 2.

EXERCÍCIO 204 44) Escreva à vontade sobre árvores como estruturas de dados.

EXERCÍCIO 205 Suponha uma estrutura de listas duplamente encadeadas, cujo esquema é o que segue Descreva o que deve acontecer com os 4 apontadores envolvidos quando há

 ©88-08, Pedro Kantek



1. uma inserção na cabeça
2. uma insersão na cauda
3. uma inserção no meio da lista
4. uma retirada na cabeça
5. uma retirada na cauda
6. uma retirada no meio

Não se esqueça que a ordem das operações é importante, ou seja, você deve indicar em que ordem os apontadores serão modificados.

EXERCÍCIO 206 Seja o seguinte código de uma função que faz pesquisa binária

```
1: estrutura inteiro chave, ... resto ...  TAB
2: inteiro função PESQBIN (TAB tabela, inteiro chave, início, fim)
3: inteiro meio
4: repita
5:    meio ← (inicio + fim ) div 2
6:    se chave > tabela [meio] então
7:       inicio ← meio + 1
8:    senão
9:       fim ← meio - 1
10:   fimse
11: até inicio > fim ∨ chave = tab[meio]
12: se chave = tab[meio] então
13:   devolva meio
14: senão
15:   devolva -1
16: fimse
```

Reescreva esta função de maneira a que ela fique recursiva

EXERCÍCIO 207 Seja uma função que pesquisa a existência de um determinado valor em uma Árvore Binária de Pesquisa definida sobre M. A função devolve o endereço em M do nodo, se este existir e -1 se ele não existir.

Considere o nodo assim definido

Endereço do filho esq	Conteúdo (1 número)	Endereço do filho dir

```
1: inteiro função PESQABP (inteiro arvore, valor)
2: inteiro ACHOU ← -1
3: se M[ARVORE+1] ≠ -1 então
4:    enquanto (ARVORE ≠ -1) faça
5:       se VALOR = M[ARVORE+1] então
6:          ACHOU ← ARVORE
7:          saia
8:       senão
9:          se VALOR < M[ARVORE+1]  então
```

©88-08, Pedro Kantek

```
10:            ARVORE ← M[ARVORE]
11:          senão
12:            ARVORE ← M[ARVORE+2]
13:          fimse
14:        fimse
15:      fimenquanto
16:  fimse
17:  devolva ACHOU
18:  fim função
```

Escreva esta mesma função de modo recursivo.

10.7 Códigos de Huffman

O algoritmos mais famoso, para esta compressão é o algoritmo de Huffman, descrito a seguir, através de um exemplo de construção.

Este algoritmo começa pela análise das freqüências de ocorrência de cada um dos caracteres do arquivo a comprimir. Suponhamos, a título de exemplo, um arquivo que contenha

```
ABCDEFABCDEABCABAABD
```

Contabilizando as ocorrências de cada caractere, ficamos com as se-guintes freqüências:

```
B=25\%, E=10\%, A=30\%, C=15\%, D=15\%, e F=5\%.
```

A seguir, o algoritmo prevê a colocação desta tabela em ordem de-crescente de ocorrências. No exemplo, fica:

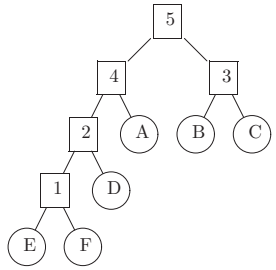
```
A=30\%, B=25\%, C=15\%, D=15\%, E=10\% e F=5\%.
```

Na terceira etapa, é criada uma árvore binária, onde as folhas repre-sentam os caracteres a comprimir, e o número de nós é o número de folhas diminuído de uma unidade. A regra para a montagem da árvore é a seguinte: cada nó terá dois sub-nós, sendo que o caractere mais freqüente ficará sempre à esquerda. Se houver freqüências iguais en-tre nó e folha, colocamos à esquerda o nó. Vejamos no exemplo:

Comparam-se os primeiros (de baixo para cima, ou seja os menores), dois elementos, E (10%) e F (5%). Então o E fica à esquerda e o F à direita. Agrupamos o E e o F sob um nó, a quem chamaremos 1, que passará a ter a freqüência dos dois somados (15%). Continuamos a buscar os 2 menores valores, que agora são o nó 1 e o caractere D. Agrupamos os 2, formando o nó 2, ao qual estão subordinados o nó 1 (à esquerda) e a letra D (à direita). O nó 2 tem freqüência 30%.

Os próximos 2 menores são o B e o C. Cria-se o nó 3 que os agrega, com freqüência 40%. Os próximos são o nó 2 e a letra A, que criam o nó 4, com freqüência 60%. Finalmente encerra a árvore o nó 5, que agrega os nós 4 e 3, com freqüência 100%. Visualmente, a árvore ficará como:

 ©88-08, Pedro Kantek



O que resultaria nos seguintes códigos de bit

A	01
B	10
C	11
D	001
E	0000
F	0001

A partir desta árvore, cada caractere a transmitir será enviado pelos bits que permitem o caminhamento pela árvore, entendendo-se o bit zero como "esquerda"e o bit 1 como "direita". No nosso exemplo, ao transmitir a mensagem ABAEC, enviaríamos 01.10.01.0000.11, com os pontos colocados apenas por clareza. Note-se que os caracteres de maior freqüência terão comprimento menor, o que caracteriza a com-pressão e o conseqüente ganho em termos de transmissão. Por outro lado, olhando a mensagem, sem conhecer a árvore original é muito difícil determinar onde começa e termina cada caractere, o que torna a sua quebra quase impossível.

Para determinarmos o comprimento do arquivo a mandar, e que em última análise determinará a economia pela compressão, usamos a regra do comprimento ponderado da árvore gerada. Isto significa que cada nó deverá gerar uma multiplicação entre o comprimento em bits do caractere representado no nó pela sua freqüência de ocorrência.

No nosso exemplo, isto ficaria:

A	2 x 0.30	0.60
B	2 x 0.25	0.50
C	2 x 0.15	0.30
D	3 x 0.15	0.45
E	4 x 0.10	0.40
F	4 x 0.05	0.20

A somatória destes valores é, em bits, o comprimento médio de cada caractere neste arquivo. No nosso exemplo, a soma deu 2.45, e como são 20 caracteres no total, o comprimento do arquivo comprimido (2.45 x 20) é 49 bits.

Existe um teorema, que não será demonstrado, que informa ser a ár-vore construída segundo o critério visto acima, a árvore de menores caminhos ponderados, ou seja a árvore de Huffman é a solução ótima para este problema.

Desafio

- Escrever um programa que faça a criptografia e a decriptografia de um conjunto de dados binários. A árvore pode ser fornecida na forma de uma constante, não precisando ser dinamicamente construída.

©88-08, Pedro Kantek

10.8 Huffman dinâmico

As duas grandes desvantagens do método de Huffman (estático) são

1. Se as freqüências dos caracteres do arquivo de entrada não são conhecidas *a priori*, é necessário ler o arquivo de entrada duas vezes. Se for um arquivo sendo transmitido em tempo real, é impossível lê-lo duas vezes.

2. A árvore usada para a compressão dos dados precisa ser anexada a eles.

Consegue-se evitar estas desvantangens por um método dinâmico em que a árvore é atualizada cada vez que um caracter é lido na entrada. Em um determinado momento, a árvore existente é a árvore de Huffman para os dados lidos até aquele momento. O procedimento de descompressão segue exatamente a mesma idéia, agora em ordem inversa. A eficiência deste método se baseia em uma característica das árvores de Huffman conhecida como propriedade do parentesco.

Propriedade do parentesco Seja T uma árvore de Huffman com n folhas, todas com pesos positivos. Então, os nodos de T podem ser arranjados em uma seqüência $(x_0, x_1, ..., x_{2n-2})$ tal que:

1. A seqüência de pesos $(peso(x_0), peso(x_1), ..., peso(x_{2n-2}))$ está em ordem decrescente

2. Para qualquer i, onde $(0 \leq i \leq n-2)$ os nodos consecutivos x_{2i+1} e x_{2i+2} são aparentados (ou seja eles têm o mesmo pai).

Tanto a compressão quanto a descompressão inicializam a árvore de Huffman dinâmica com um único nodo que contém um caractere artificial que será denotado por ART. O peso deste nodo único é sempre 1.

10.8.1 Codificação

Cada vez que o símbolo a é lido no arquivo de entrada, sua palavra de código na árvore é enviada. Entretanto, isto só pode acontecer se a já tiver aparecido no fluxo de entrada antes. Em caso contrário, ART é enviado seguido pelo código ASCII de a. A seguir, a árvore é modificada. Se a nunca havia ocorrido antes, um novo nodo interno é criado e seus nodos filhos são identificados como a e como ART. Depois disso, a árvore é atualizada para obter uma árvore de Huffman correta até aqui.

Cada nodo será identificado por um número n, sendo a raiz o nodo 0. Uma invariante da compressão e da descompressão é que se esta árvore tem m nodos, a seqüência de nodos $(m-1, ..., 1, 0)$ satisfaz à propriedade do parentesco. A árvore é armazenada em uma tabela, com as seguintes propriedades

pai(n) é o pai do nodo n, lembrando que o pai(raiz)=indefinido

filho(n) é o filho esquerdo de n, se n for um nodo interno. Caso contrário filho(n) é indefinido. O filho direito não é definido na árvore, já que pela propriedade do parentesco, será sempre a próxima linha da matriz, ou no caso filho(n)+1.

peso(n) é a freqüência de símbolo(n) quando n é uma folha.

A correspondência entre símbolos e folhas da árvore é dada por uma tabela chamada folha. Para cada caracter $a \in \Sigma \cup \{END\} \cup \{ART\}$. folha[$a$] é a correspondente folha de a na árvore. No exemplo que faremos, ART=* e END=.

 ©88-08, Pedro Kantek

A árvore começa com uma árvore com um único nodo, identificado como ART. Para as árvores a seguir desenhadas, considere o seguinte padrão.

(peso) número do nodo
caracter

Com isso, eis o nodo inicial:

(1) 0
ART

O começo de tudo é a inicialização. Eis o algoritmo

```
1: função HD-INICIALIZAÇÃO
2: raiz ← 0
3: pai[raiz] ← -1
4: filho[raiz] ← -1
5: peso[raiz] ← 1
6: para cada caracter a ∈ Σ ∪ {END} da entrada faça
7:    folha[a] ← -1
8: fimpara
9: folha[ART] ← raiz
```

A seguir, o algoritmo básico da codificação:

```
1: função HD-CODIFICAÇÃO(entrada,saída)
2: HD-INICIALIZAÇÃO
3: enquanto houver dados na entrada ∧ a é o próximo símbolo faça
4:    HD-CODIFICA-SIMBOLO(a,saída)
5:    HD-ATUALIZA-ARVORE(a)
6: fimenquanto
7: HD-CODIFICA-SIMBOLO(ART,saída)
```

A codificação da entrada é uma sucessão de 3 etapas: ler o símbolo a da entrada, codificar o símbolo a a partir da árvore corrente e atualizar a árvore. Codificar o símbolo a já existente na árvore consiste em acessar o nodo a e então percorrer a árvore da raiz até a gerando os bits associados (0 à esquerda e 1 à direita). Cada vez que um nodo n (e $n \neq$ raiz) é encontrado, se n é ímpar, 1 é enviado (n é filho direito de seu pai) e se n é par, um 0 é enviado. Como o código de a é lido em ordem reversa, uma pilha S é usada para armazenar os bits e envia-los apropriadamente.

Se a não apareceu ainda, o código de ART é enviado, seguido pelos 9 bits (bit zero, mais os 8 bits do código ASCII do caracter) e uma nova folha é criada para a.

```
1: função HD-CODIFICA-SIMBOLO(a,saída)
2: S ← pilha vazia
3: n ← folha[a]
4: se n = −1 então
5:    n ← folha[ART]
6: fimse
7: enquanto n ≠ raiz faça
8:    se n é impar então
9:       Empilhe (S,1)
10:   senão
11:      Empilhe (S,0)
12:   fimse
13:   n ←pai[n]
```

©88-08, Pedro Kantek

14: **fimenquanto**
15: Envie(S,saída)
16: **se** folha[a]= -1 **então**
17: Escreva em saída um zero mais os 8 bits de a em ASCII
18: HD-ADICIONE(a)
19: **fimse**

Agora, a função de inclusão do nodo a

1: função HD-ADICIONE(a)
2: transforme folha[ART] em
3: um nodo interno de peso 1 com
4: um filho esquerdo de peso 0 para a folha[a]
5: um filho direito de peso 1 para folha[ART]

Finalmente, é necessária a função

1: função HD-ATUALIZA-ARVORE(a)
2: $n \leftarrow$ folha[a]
3: **enquanto** $n \neq$ raiz **faça**
4: peso[n] $\leftarrow$ peso[n]+1
5: $m \leftarrow n$
6: **enquanto** peso[m-1] $<$ peso[n] **faça**
7: $m \leftarrow m - 1$
8: **fimenquanto**
9: HD-TROQUE-NODOS(m, n)
10: $n \leftarrow$ pai[m]
11: **fimenquanto**
12: peso[raiz] $\leftarrow$ peso[raiz]+1

A função HD-TROQUE-NODOS tem a função de inverter a posição de 2 nodos dentro da estrutura (tabela) que representa a árvore. Alguns cuidados devem ser tomados nesta função:

- Na troca de 2 nodos apenas as colunas 2 e 3 (endereço do filho e peso) são trocados. O endereço 1 (endereço do pai) permanece com o valor antigo, para que a propriedade do parentesco não se perca. Depois da inversão, os valores são trocados globalmente em toda a coluna 1 da tabela.

- Eventualmente, o vetor que aponta para as linhas da tabela, fazendo a ligação entre caracter e nodo, também precisa ser alterado para refletir a troca entre nodos.

Procure entender estas duas observações olhando para o exemplo a seguir feito em etapas.

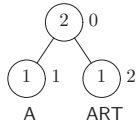
10.8.2 Exemplo

Seja comprimir $y = ACAGAATAGAGA$

1. Árvore inicial

2. O primeiro símbolo é A. O código ASCII é enviado com 9 bits
Bits enviados: 0.0100.0001

 ©88-08, Pedro Kantek



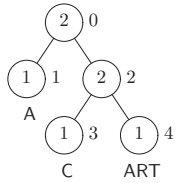
Eis como fica a árvore na forma de tabela

end	pai	filho	peso
0	-1	1	2
1	0	-1	1
2	0	-1	1

Eis a vinculação dos nodos

A	C	G	T	ART
1	-1	-1	-1	2

3. O próximo símbolo é C. São enviados o código ART e a seguir o código ASCII em 9 bits de C.
Bits enviados 1..0.0100.0011. Fica



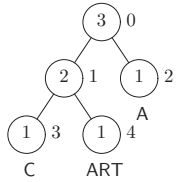
A árvore na forma de tabela, antes da troca é

end	pai	filho	peso
0	-1	1	2
1	0	-1	1
2	0	3	2
3	2	-1	1
4	2	-1	1

Eis a vinculação dos nodos

A	C	G	T	ART
1	3	-1	-1	4

Os nodos 1 e 2 devem ser trocados. Porquê _____?



A árvore na forma de tabela, depois da troca é

©88-08, Pedro Kantek

end	pai	filho	peso
0	-1	1	2
1	0	3	2
2	0	-1	1
3	1	-1	1
4	1	-1	1

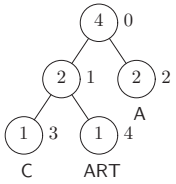
Eis a vinculação dos nodos

A	C	G	T	ART
2	3	-1	-1	4

Após a atualização dos pesos, a árvore fica

end	pai	filho	peso
0	-1	1	3
1	0	3	2
2	0	-1	1
3	1	-1	1
4	1	-1	1

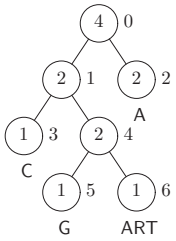
4. O próximo símbolo é A. O código de A é enviado.
   Bit enviado: 1



A tabela

end	pai	filho	peso
0	-1	1	4
1	0	3	2
2	0	-1	2
3	1	-1	1
4	1	-1	1

5. O próximo símbolo é G. São enviados o código ART e o ASCII de 9 bits de G.
   Bits enviados: 01..0.0100.0111.
   Fica



A tabela fica (antes da troca)

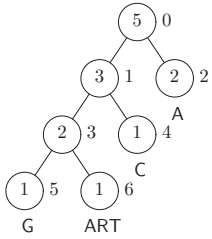
 ©88-08, Pedro Kantek

end	pai	filho	peso
0	-1	1	4
1	0	3	2
2	0	-1	2
3	1	-1	1
4	1	5	2
5	4	-1	1
6	4	-1	1

Eis a vinculação dos nodos

A	C	G	T	ART
2	3	5	-1	6

Nodos 3 e 4 são trocados



A tabela fica (depois da troca)

end	pai	filho	peso
0	-1	1	4
1	0	3	2
2	0	-1	2
3	1	5	2
4	1	-1	1
5	3	-1	1
6	3	-1	1

Eis a vinculação dos nodos

A	C	G	T	ART
2	4	5	-1	6

Depois, a atualização dos pesos

end	pai	filho	peso
0	-1	1	5
1	0	3	3
2	0	-1	2
3	1	5	2
4	1	-1	1
5	3	-1	1
6	3	-1	1

6. A seqüência inteira enviada é (pontos apenas para clareza)

$$\overbrace{0.0100.0001}^{A}\ \overbrace{1}^{ART}\ \overbrace{0.00100.0011}^{C}\ \overbrace{1}^{A}\ \overbrace{01}^{ART}\ \overbrace{0.0100.0111}^{G}\ \overbrace{1}^{A}$$

$$\overbrace{1}^{A}\ \overbrace{101}^{ART}$$

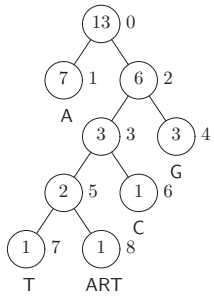
©88-08, Pedro Kantek

$$\overbrace{0.0101.0100}^{T}\ \overbrace{0}^{A}\ \overbrace{0100}^{G}\ \overbrace{0}^{A}\ \overbrace{100}^{G}\ \overbrace{0}^{A}\ \overbrace{1001}^{ART}\ \overbrace{1.0000.0000}^{FIM}$$

O tamanho total do texto enviado é de 67 bits, quando sem compressão esta mensagem ocuparia ($12 \times 8 =$) 96 bits.

Observação: No livro do Attalah, capítulo 12, de onde esta aula foi extraída, o exemplo acima informa que o ART é 111. Na verdade, tudo aponta para estarmos diante de um erro do livro, já que o valor correto parece ser 1001.

A última árvore deste exercício é



A tabela final fica

end	pai	filho	peso
0	-1	1	13
1	0	-1	7
2	0	3	6
3	2	5	3
4	2	-1	3
5	3	7	2
6	3	-1	1
7	5	-1	1
8	5	-1	1

Eis a vinculação dos nodos

A	C	G	T	ART
1	6	4	7	8

10.8.3 Decodificação

Lembrando que, neste método o receptor não sabe nada sobre o emissor, exceto o fluxo de bits que chega. Não há árvore, nem combinação prévia nem nada. O único que ambos compartilham é este algoritmo. A árvore também começa vazia e vai sendo igualmente construída a cada caracter que é decodificado.

Eis os algoritmos empregados

```
1: função HD-DECODIFICAÇÃO(entrada,saída)
2: HD-INICIALIZAÇÃO
3: a ← HD-DECODIFICA-SÍMBOLO (entrada)
4: enquanto a ≠ END faça
5:    Escreva a na saída
6:    HD-ATUALIZA-ARVORE(a)
7:    a ← HD-DECODIFICA-SÍMBOLO(entrada)
```



```
8: fimenquanto
```

A seguir, a função que serve para decodificar cada um dos caracteres da entrada. Note que as funções de construção da árvore (DH-ADICIONA-NODO e HD-ATUALIZA-ARVORE) são rigorosamente as mesmas usadas na fase de codificação.

```
1: função HD-DECODIFICA-SIMBOLO(entrada)
2: n ← RAIZ
3: enquanto filho[n] ≠ −1 faça
4:    Leia o bit b da entrada
5:    n ← filho[n] + b
6: fimenquanto
7: a ← caracter associado ao nodo n
8: se a = ART então
9:    a ← símbolo correspondente aos próximos 9 bits da entrada
10:   DH-ADICIONA-NODO(a)
11: fimse
12: Devolva(a)
```

Desafio

- Escrever um programa que implemente estas funções de Huffman dinâmico. Teste o programa de codificação com um conteudo qualquer, depois decodifique-o e compare os dois arquivos (o primeiro e o último) através do utilitário FC (file comparer) ou similar. Não se preocupe em comprimir nem em otimizar. Deixe que os arquivos cresçam, por exemplo, usando BYTES 0 e 1 ao invés de BITS 0 e 1. Gere feed-backs abundantes para depurar o programa. Diversão garantida para 2 ou 3 tardes chuvosas.

10.9 Árvores B

Por mais que se sofistiquem os gerenciadores de bases de dados e os modelos relacionais de bancos de dados, não tem jeito: a máquina só aceita ler e gravar dados de duas maneiras: ou seqüencialmente, em que um dado vem comportadamente seguindo ao anterior, ou diretamente, onde cada registro tem associado a sí o endereço dele, relativamente ao início do arquivo. É notória a pobreza (poder-se-ia dizer "franciscana") dos métodos de acesso básicos de qualquer computador.

A questão é: como é possível tamanha riqueza de alternativas de acesso e recuperação a dados tendo-se disponíveis apenas o acesso seqüêncial e direto ? A resposta é sempre a mesma: graças ao software.

Uma das maneiras para organizar acessos por conteúdo (e não por endereço – que relembrando é o único acesso que os computadores conhecem), é organizar índices que associem certo conteúdo a certo endereço. Em outras palavras, o arquivo original é gravado seqüencialmente, e neste cada registro tem um endereço físico associado. Um segundo arquivo (índice) é construído na forma de uma tabela que associa conteúdos, por exemplo, número do CPF, a endereços físicos do arquivo original.

A questão que permanece, é como organizar este segundo arquivo. Valem para ele as mesmas restrições existentes para qualquer arquivo: só é possível acessá-lo seqüêncial ou diretamente.

Definição

Em 1972, Bayer e MacCreight introduziram uma estrutura de dados bastante adequada para resolver este problema: denominaram-na B-árvore, mas abstiveram-se de explicar

este nome. A B-árvore é uma árvore n-ária, isto é, cada nodo pode ter n filhos, sendo que em geral este valor de n é escolhido de tal maneira a otimizar a blocagem física do arquivo de índices.

Relembrando o conceito de árvores, estas são estruturas encadeadas, compostas por inúmeros nós, para os quais se estabelece uma relação de hierarquia, poder-se-ia dizer familiar. O uso da analogia familiar para descrever árvores é cômodo e tem sido largamente usado [1] O nodo original é chamado "raiz"e é o único que não tem pai, mas pode ter de 0 até n filhos. Cada um dos filhos, por sua vez, pode ter de 0 a n filhos (que serão netos do nodo raiz) e assim sucessivamente. Este valor de n, é denominado grau da árvore, e as árvores B tem este valor na casa das dezenas ou centenas. Cria-se assim uma estrutura piramidal, com vaga semelhança com as árvores: daí o nome.

A árvore B de grau t é aquela que cada nodo tem as seguintes informações:

- o número k de chaves guardados naquele nodo ($t-1 \leq k \leq 2t-1$, com exceção da raiz que pode ter $1 \leq k \leq 2t-1$)
- as k chaves armazenadas em ordem crescente, junto com todas as demais informações necessárias à recuperação dos dados ligados a esta chave.
- Se o nodo é intermediário (isto é não é folha), além das k chaves tem-se k +1 endereços das folhas subordinadas a esta.

Diz-se portanto, que uma árvore B de grau 5 é aquela que pode conter desde 4 até 9 chaves e de 5 até 10 endereços por nodo. Um caso particular ocorre quando t = 2 e a árvore é chamada árvore 2-3-4.

A árvore B é equilibrada, ou seja qualquer caminho desde uma folha até a raiz, terá o mesmo comprimento.

O número de acessos requeridos para acessar uma determinada chave em uma árvore-B é proporcional à sua altura h. E, a sua altura h é proporcional a $\log_k n$, onde n é o número de chaves na árvore e k é o grau da árvore.

Como, usualmente o grau da árvore é escolhido de maneira a que um nodo completo caiba em um bloco físico de disco, k costuma ter valores elevados, o que faz com que árvores com grande número de nodos tenham baixa altura. Acompanhemos no exemplo:

t	k_{min}	k_{max}	nodos para $h=4$
10	9	19	21.000
20	19	39	460.000
40	39	79	7.300.000
100	99	199	304.000.000

Para o cálculo acima, considerou-se 2/3 de k_{max}.

Funcionamento da árvore B

Suponhamos uma B-árvore X com $t = 3$. Cada nodo poderá conter de 2 a 5 chaves e de 3 a 6 endereços associados. Para efeito desta demonstração, cada nodo V de X, conterá as seguintes informações:

- V[1] = valor booleano indicando se este nodo é folha (1) ou nodo intermediário (0)

[1] Se bem que nem sempre dá muito certo. Veja-se o texto abaixo, publicado no início do século em um jornal alemão: "...eu me casei com uma viúva que tinha uma filha crescida. Meu pai que nos visitava com freqüência apaixonou-se pela minha enteada e se casou com ela. Assim meu pai se tornou meu genro e minha enteada se tornou minha mãe. Al-guns meses mais tarde minha esposa deu a luz a um filho que se tornou cunhado de meu pai e ao mesmo tempo meu tio. A esposa de meu pai que é minha enteada também teve um filho. Desta maneira eu ganhei um irmão e ao mesmo tempo um neto. Minha esposa tornou-se agora minha avó já que ela é mãe de minha mãe. Assim sendo eu sou o marido de minha esposa e ao mesmo tempo seu enteado-neto. Em outras palavras, tornei-me meu proprio avó."

©88-08, Pedro Kantek

- V[2] = quantidade de chaves efetivamente usadas neste nodo
- V[3..7] = as 5 chaves
- V[8..13] = os 6 endereços associados às chaves

Ao incluir a primeira chave, digamos 33, a B-árvore X fica (A raiz está indicada por um *)

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
* 1:  1  1   33    0    0    0    0
```

Incluindo-se o número 45, fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
* 1:  1  2   33   45    0    0    0
```

Incluindo-se o 76, o 2 e o 20, fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
* 1:  1  5    2   20   33   45   76
```

Incluindo-se o 15, fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   15   20    0    0
* 2:  0  1   33    0    0    0    0   1   3
  3:  1  2   45   76    0    0    0
```

Incluindo-se o 50, 53 e 56 fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   15   20    0    0
* 2:  0  1   33    0    0    0    0   1   3
  3:  1  5   45   50   53   56   76
```

Incluindo-se o 59, fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   15   20    0    0
* 2:  0  2   33   53    0    0    0   1   3   4
  3:  1  2   45   50    0    0    0
  4:  1  3   56   59   76    0    0
```

Incluindo-se o 17 e o 25

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  5    2   15   17   20   25
* 2:  0  2   33   53    0    0    0   1   3   4
  3:  1  2   45   50    0    0    0
  4:  1  3   56   59   76    0    0
```

Incluindo-se o 10, fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   10   15    0    0
* 2:  0  3   17   33   53    0    0   1   5   3   4
  3:  1  2   45   50    0    0    0
  4:  1  3   56   59   76    0    0
  5:  1  2   20   25    0    0    0
```

Incluindo-se o 80, 85 e 90 fica

©88-08, Pedro Kantek

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   10   15    0    0
* 2:  0  4   17   33   53   76    0   1   5   3   4   6
  3:  1  2   45   50    0    0    0
  4:  1  2   56   59    0    0    0
  5:  1  2   20   25    0    0    0
  6:  1  3   80   85   90    0    0
```

Incluindo-se o 92, 94 e 96 fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   10   15    0    0
* 2:  0  5   17   33   53   76   90   1   5   3   4   6   7
  3:  1  2   45   50    0    0    0
  4:  1  2   56   59    0    0    0
  5:  1  2   20   25    0    0    0
  6:  1  2   80   85    0    0    0
  7:  1  3   92   94   96    0    0
```

Inserindo-se o 91 fica

```
      F QTD  K1   K2   K3   K4   K5  F1  F2  F3  F4  F5  F6
      - ---  --   --   --   --   --  --  --  --  --  --  --
  1:  1  3    2   10   15    0    0
  2:  0  2   17   33    0    0    0   1   5   3
  3:  1  2   45   50    0    0    0
  4:  1  2   56   59    0    0    0
  5:  1  2   20   25    0    0    0
  6:  1  2   80   85    0    0    0
  7:  1  4   91   92   94   96    0
* 8:  0  1   53    0    0    0    0   2   9
  9:  0  2   76   90    0    0    0   4   6   7
```

e assim por diante.

10.9.1 Busca em B-árvores

A busca, como em qualquer árvore começa na raiz e se extende até a localização da chave buscada, ou de uma folha onde ela deveria estar. Neste segundo caso a conclusão da busca é pela não existência da chave procurada.

Dentro de cada um dos nodos do caminho, há que se localizar a chave mais próxima da chave que se busca. Isso é facilitado pelo fato das chaves dentro dos nodos estarem ordenadas: cabe aqui como uma luva a busca binária.

Este algoritmo (busca), além da varredura, estão programados no WS VIVO441.

Desafio

- Implementar a função de pesquisa de um determinado valor em uma árvore B. A mesma não precisa ser construída dinamicamente, podendo ser fornecida ao programa na forma de uma constante. (No caso, copiar qualquer um dos exemplos de VIVO441a.

10.10 Algoritmos de Árvores B

Suponhamos uma árvore formada por uma matriz de BARV de 100 linhas por 13 colunas. Existe uma variável global denominada ULTIMA que contém a última linha usada (0 se ainda não se usou nada).

```
1: inteiro função ALOCABTREE
```



```
2: x ← ULTIMA+1
3: BARV[x;1]← 1
4: ULTIMA← ULTIMA + 1
5: devolve x
6: fimfunção
```

Função de pesquisa em uma B-árvore

```
1: inteiro função PESQBARV (inteiro NFOL, CHAVE)
2: I← 1
3: enquanto (I≤ BARV[NFOL,2]) ∧ (CHAVE > BARV[NFOL,7+I]) faça
4:    I++
5: fimenquanto
6: se (I ≤ BARV[NFOL,2]) ∧ (CHAVE = BARV[NFOL,7+I]) então
7:    devolva (CHAVE,I)
8: senão
9:    se BARV[NFOL,1] = 1  então
10:      devolva -1
11:   senão
12:      AUX← BARV[NFOL,7+I]
13:      devolva PESQBARV(BARV[NFOL,AUX],CHAVE)
14:   fimse
15: fimse
16: fim função
```

Inserção em uma B-árvore, cujo grau seja "t"

```
1: inteiro função INSBARV (CHAVE)
2: R← RAIZ
3: se BARV[R,2] = (2*t) - 1 então
4:    S← ULTIMA← ULTIMA + 1
5:    RAIZ← S
6:    BARV[S,1]← 0
7:    BARV[S,8]← R
8:    QUEBRABARV(S,1,R)
9:    INSNCBARV(S,CHAVE)
10: senão
11:   INSNCBARV(R,CHAVE)
12: fimse
13: fim função
```

[1]

```
1: inteiro função QUEBRABARV(inteiro PAI, I, FILHO)
2: z← ULTIMA← ULTIMA + 1
3: BARV[z,1]← BARV[FILHO,1]
4: BARV[z,2]← t - 1
5: para J de 1 até t-1 faça
6:    BARV[Z,2+J]← BARV[FILHO,2+t+J]
7: fimpara
8: se BARV[FILHO,1] = 0 então
9:    para J de 1 até t faça
10:      BARV[Z,7+J]← BARV[FILHO,7+t+J]
11:   fimpara
12: fimse
13: BARV[FILHO,2] ← t - 1
14: para J de BARV[PAI,2] até I passo -1 faça
```

```
15:    BARV[PAI,3+J]← BARV[PAI,2+J]
16: fimpara
17: BARV[PAI,2+I]← BARV[FILHO,t]
18: BARV[PAI,2]← BARV[PAI,2] + 1
 1: inteiro função INSNCBARV (inteiro NODO, CHAVE)
 2: I← BARV[NODO,2]
 3: se BARV[NODO,1] = 1 então
 4:    enquanto (I ≥ 1) ∧ (CHAVE < BARV[NODO,2+I]) faça
 5:       BARV[NODO,I+1]← BARV[NODO,I]
 6:       I← I - 1
 7:    fimenquanto
 8:    BARV[NODO, I+1]← CHAVE
 9:    BARV[NODO,2]← BARV[NODO,2] + 1
10: senão
11:    enquanto (I ≥ 1) ∧ (CHAVE < BARV[NODO,2+I]) faça
12:       I← I - 1
13:    fimenquanto
14:    I← I + 1
15:    se BARV[BARV[NODO,7+I],2] = 2×t - 1 então
16:       QUEBRABARVB (NODO, I, BARV[NODO,7+I])
17:       se CHAVE > BARV[NODO,2+I] então
18:          I← I + 1
19:       fimse
20:    fimse
21:    INSNCBARVB (BARV[NODO,7+I], CHAVE)
22: fimse
23: fim função
```

10.10.1 Uma esperteza

Para eliminar o comportamento probabilístico da inclusão em uma árvore B, o algoritmo acima usa um truque bem interessante. Ao invés de esperar para fazer o split na hora em que não houver mais espaço em todos os nodos, o algoritmo, enquanto faz a busca de cima para baixo procurando o lugar correto de uma inserção, vai cortando ("splittando") todos os nodos CHEIOS que estão no caminho entre a raiz e o local da inserção. Com isso, o número máximo de splits em qualquer momento é de apenas 1.

Se ele esperasse até o momento realmente necessário, o número de splits poderia ser enorme, causando – para um único registro, é verdade – uma demora inaceitável.

Assim troca-se o split bottom up (subindo na medida da necessidade) e chamado quando não há mais espaço, por um split top-down que ocorre durante a descida na árvore e sempre que um nodo cheio é encontrado no caminho.

Perceba isso no exemplo: Seja uma b-arvore com t=3 e os primeiros 18 números inseridos em ordem. Eis o aspecto da árvore

```
     1 2    3   4   5   6   7  8  9 10 11 12 13
   +-----------------------------------------
  1| 1 2    1   2   0   0   0
* 2| 0 5    3   6   9  12  15  1  3  4  5  6  7
  3| 1 2    4   5   0   0   0
  4| 1 2    7   8   0   0   0
  5| 1 2   10  11   0   0   0
  6| 1 2   13  14   0   0   0
  7| 1 3   16  17  18   0   0
```

 ©88-08, Pedro Kantek

Na inserção do elemento 19, percebe-se haver lugar ainda no nodo 7 (à direita do 18). Se não houvesse a esperteza, a inclusão seria feita aqui sem nenhum estardalhaço, adiando o split. Quando ele ocorresse na folha, ocorreria também no pai da folha, totalizando dois splits.

Entretanto, graças à esperteza, durante a busca na direção raiz → folha, encontrou-se um nodo (o 2) completo. Ele foi cortado incontinente. Só depois a inclusão do 19 foi feita. Assim, um dos dois splits acima descritos já ocorreu. O segundo só vai ocorrer quando uma folha estiver cheia, e neste caso, com certeza haverá espaço no pai para receber a chave que "sobe". Eis como ficou a árvore acima após a inclusão da chave 19.

```
     1 2    3   4   5   6   7  8  9 10 11 12 13
   +-----------------------------------------
  1| 1 2    1   2   0   0   0
  2| 0 2    3   6   0   0   0  1  3  4
  3| 1 2    4   5   0   0   0
  4| 1 2    7   8   0   0   0
  5| 1 2   10  11   0   0   0
  6| 1 2   13  14   0   0   0
  7| 1 4   16  17  18  19   0
* 8| 0 1    9   0   0   0   0  2  9
  9| 0 2   12  15   0   0   0  5  6  7
```

Em resumo, sem a esperteza, o split quando ocorrer, poderá causar n splits recursivamente, desde a folha até tão alto quanto se queira. Com a esperteza, os splits ocorrem com mais freqüência, mas sempre limitados a 1 operação por inclusão. Esta explicação está também em Cormen et alli, edição em português, pág. 356.

10.10.2 Exclusão em B-árvores

Para remover uma chave da árvore B, duas hipóteses devem ser consideradas

1. A chave está em uma folha
2. A chave está em um nodo intermediário

No segundo caso, o sucessor da chave a ser eliminada, estará em uma folha. Simplesmente, ele deve sair desta folha e ocupar o lugar que o original a ser excluído, ocupava.

Recaímos assim no primeiro caso. Ao excluir a chave, deve-se verificar se o limite inferior de $t-1$ chaves não é ultrapassado. Se não for, a exclusão é trivial. Se a folha ficará com menos do que $t-1$chaves, deve-se reorganizar esta folha. Para tanto, deve-se olhar os irmãos à esquerda e à direita da folha. Se um deles tiver mais do que $t-1$ chaves, basta tomar "emprestada" a chave. Agora, a chave do pai que separa os irmãos é trazida para a folha e a chave emprestada do irmão vai para o pai.

Se os dois irmãos da folha contiverem exatamente $t-1$ chaves, não é possível emprestar nada. Assim, o nodo e um de seus irmãos são concatenados em um único nodo que também contém a chave separadora do pai.

Se o pai também só contiver $t-1$ chaves, deve-se olhar os irmãos do pai, como no caso anterior. Se necessário proceder recursivamente. No pior caso, quando todos estão no limite inferior das chaves, a retirada da última chave da raiz, força a diminuição da altura da árvore.

Desafio

- Implementar o algoritmo de criação de árvore B e de inclusão de chaves.

©88-08, Pedro Kantek

- Estudar os algoritmos em anexo e comentá-los informando qual a razão de cada um dos comandos que os compõe.

10.11 SYBASE

O SYBASE usa árvores B para indexar as tabelas. Existem 2 tipos de árvores B: as granuladas e as não granuladas (clustered). As granuladas são árvores esparsas (nem todo registro de dados tem entrada nos índices) nas quais os registros (na tabela original) são mantidos em ordem dos valores de índice e apenas o primeiro registro em cada página de dados tem uma entrada de índice. Árvores não granuladas são densas e cada um dos registros da tabela tem 1 entrada de índice.

Uma tabela SYBASE com índice granulado exige que seus elementos sejam mantidos em ordem dentro da tabela. Assim, inclusões podem se dar em qualquer página, a depender do conteúdo do campo de índice. Tabelas SYBASE que tem apenas índices não granulados fazem as inclusões de novos elementos sempre ao final das tabelas (na última página).

Tabelas SYBASE podem ter apenas 1 índice granulado, mas podem ter até 250 índices não granulados.

Na criação de um índice granulado, os dados da tabela devem estar colocados nas páginas em ordem.

Por exemplo, seja criar uma tabela com os campos 1, 120, 33, 95, 4, 67, 160, 72, 141, 80, 11, 21, 133, 30, 42, 100, 97, 18, 17 e 3.

Página 1	Página 2	Página 3	Página 4	Página 5
1	17	33	80	120
3	18	42	95	133
4	21	67	97	141
11	30	72	100	160

Apenas o primeiro registro da tabela será indexado, ou neste caso: 1, 17, 33, 80 e 120. Supondo que as páginas de índices comportem apenas 4 entradas, seriam necessárias 2:

1	17	33	80

120			

Sendo, uma árvore B, haveria uma nova entrada, acima desta contendo

1	120		

O nodo raiz se encontra num arquivo chamado sysindexes.

- granulado = clustered = esparsas
- não granuladas = denso

Cada entrada no índice de um indice granulado (esparso) é um apontador para página. Um apontador de página é composto por 5 bytes, assim uma entrada do índice é um valor de chave e um apontador de página.

Nodo raiz do índice

65

1:60	24:55		

Nodo do nível seguinte

60

1:20	6:100	12:45	18:223

Idem

55

24:80			

©88-08, Pedro Kantek

Página de dados

20 100 45 223 80

1	6	12
2	7	14
3	8	16

Indices não granulados (denso)

Nestes índices, todos os componentes da área de dados fazem parte do índice, mas os dados não precisam estar ordenados.

Perceba-se que há uma diferença grande na quantidade de acessos quando se busca um range (intervalo) de registros e estes estão ordenados por indices granulados (esparsos) ou não granulados (densos).

Eis como ficaria

Nodo raiz

150

1,200
24,300

Nível intermediário

200

1,80,5:1
6,70,15:2
12,75,100:2
18,140,30:1

300

24,225,30:4

Folhas

80 70 75 140 225

1,5:1	6,15:2	12,100:2	18,30:1
2,20:3	7,15:4	14,100:4	19,15:1
3,5:3	8,20:2	16,5:2	20,20:3
4,30:3	11,30:2	17,5:4	22,20:1

Páginas de dados

5 20 15 30 100

1	22	19	18
16	8	6	11
3	2	28	4
17	20	7	24

Cada entrada de índice em sybase pode ser composta por até 16 campos e ter até 256 bytes. Estes campos não precisam estar contiguos. Só não podem ser partes de campos e resultados de cálculos.

O tamanho da página varia, mas o tamanho mais comum é 2K. Existem 5 tipos de páginas em sybase, a saber:

Páginas de Dados contém registros de dados ou registros de log. Ambos são estruturalmente diferentes, mas são construídos do mesmo jeito.

Páginas de índices contém registros de índice

Páginas de texto ou imagem contém BLOBS

Páginas de alocação contém estruturas de dados usadas para geremciar o processo de alocacão de páginas

páginas de Estatísticas contém estatística de distribuição e uso para os índices.

©88-08, Pedro Kantek

Página de Dados Se os registros tem tamanho fixo, não haverá tabela de deslocamento na página e o endereço de cada registro é obtido mediante simples cálculo. Neste caso, o tamanho do registro (fixo) é guardado no campo TAMANHO MÍNIMO DE REGISTRO do header.

O header da página é um conjunto de 32 bytes contendo (entre outros): número lógico da página, anterior e próxima página lógica, identificação do objeto ao qual esta página pertence, primeira entrada de registro disponível nesta página, deslocamento do espaço livre, tamanho mínimo do registro

Depois vem a área de registros, que contém um número inteiro de slots para guardar registros. Um registro não atravessa limites de páginas

Tabela de deslocamento: Um conjunto de deslocamentos desde o início da página dando o início de cada um dos registros. Este conjunto cresce do fim da página para o início e só existe quando o tamanho do registro é variável.

Cada registro na página recebe um número de registro. É um campo de 1 byte e assim restringe-se o número de registros em uma página a 256. O número do registro usado nos índices não granulados (densos) é uma combinação do número da página com o número do registro na página.

O registro pode ocupar a página inteira (2048-32=2016). Não há tamanho mínimo para o registro, mas só pode haver 256 registros na página.

Um registro em SYBASE Veja-se o lay-out básico de um registro genérico em SYBASE.

1 byte: número de colunas de tamanho variável	1 byte: número do registro (0 a 255)	dados de tamanho fixo	2 bytes: tamanho total do registro	dados de tamanho variável	locais de início das colunas de tamanho variável

Seja por exemplo, um registro assim definido

```
create tabela1 (a int, b char(20))
```

que gerará um layout como segue:

0	1	2	6 a 25
0	32	campo a	campo b

Já a tabela

```
create tabela2 (a int, b varchar(20), c char(30), d varchar(10))
```

gerará a o seguinte lay-out

0	1	2	6	36	38	48	54	55	56
2	40	AAA	CCC	57	BBB	DDD	1	38	48

Note que os dados de tamanho variável são todos juntados ao final do registro. Tem-se:

- 2=número de campos de tamanho variável
- 40=número deste registro dentro desta página campos fixos
- 57=tamanho deste registro campos de tamanho variável
- 38=o campo B começa na posição 38

 ©88-08, Pedro Kantek

- 48=o campo D começa na posição 48

A busca dentro da página é feita por pesquisa binária. Quando o tamanho do registro é fixo, a coisa é trivial. Quando o tamanho é variável, há uma tabela de deslocamento dos registros no fim da página. Eis como ficaria uma página de dados com registros de tamanho variável:

End	Conteudo
32	registro 0
50	registro 4
80	registro 2
90	registro 7
120	registro 8
150	registro 9
160	registro 10

165	150	120	90	0	0	50	0	80	0	32

A entrada na tabela de deslocamento contém o endereço real do registro na área de dados que, lembrando, começa no &32. Note que a tabela de deslocamento é ideal para uma pesquisa binária dentro da página. O deslocamento 0 no registro 1 indica que o mesmo foi excluído. A próxima inclusão nesta página reaproveitará o número 1 presentemente livre, ainda que o registro seja gravado fisicamente ao final da página. Quando os registros são excluídos, os que sobram são reagrupados no início da página de maneira a manter um único bloco de espaço livre ao final da página.

Desafio

- Achar uma instalação SYBASE. Nela criar um arquivo idêntico ao do exercício feito em sala de aula. Obtem um *dump* de todos os arquivos envolvidos e nele localizar cada uma das informações aqui estudadas.

10.12 xBASE

O dbase foi o primeiro produto de banco de dados para uso em micro-computadores. Lançado em 1980 como dbase2 (nunca existiu um dbase 1), tinha limitações que hoje fariam corar o mais circunspecto analista de sistemas, mas na época foi uma revolução.

De 82 a 87 reinou soberano, e acabou criando uma genealogia de produtos e metodologias (dbase 2, 3, 3plus, 4, 5, Visualdbase 7, FoxPro, FoxBaseParadox, Clipper, só para ficar nos mais conhecidos).

Sua importância hoje reside no fato de que 100% dos produtos que iteragem com algum tipo de banco de dados em microcomputadores aceitam o padrão dbase. Assim, por exemplo, o AutoCad aceita dados em formato dbase. O MSWord aceita dados para mala direta no formato dbase e assim por diante.

Também é importante estudá-lo pela simplicidade do esquema e pelo fato de que funciona e bem. Finalmente, por questões de propriedade de marca, o nome dbase foi abandonado (já que o nome – mas não os formatos – é registrado pelos proprietários, originalmente a Ashton Tate e depois a Borland) e em seu lugar usa-se o termo xbase.

Um banco de dados XBASE é um conjunto de diversos arquivos, cujos principais são

©88-08, Pedro Kantek

Tipo de arquivo	Conteúdo
DBF	Data Base File: o arquivo que contém os dados. Tem registros de tamanho fixo, e as inclusões sempre se dão ao final do arquivo. Exclusões mantém o registro no seu lugar, até que ocorra um PACK. Atualizações se dão no mesmo lugar.
NDX	Index File. O arquivo organizado na forma de uma árvore B+. Os apontadores para o número do registro (no DBF equivalente) só existem nas folhas.
DBT	Data Base Text. Arquivo que contém os campos MEMORANDO que porventura sejam definidos no DBF. Lá fica apenas um apontador de 10 bytes para o endereço do memorando equivalente que está no arquivo DBT.

O dbase armazena os dados brutos em arquivos cuja extensão é DBF (data base file). Cada registro que é incluído o é no fim do arquivo e ele é numerado sequencialmente no instante em que entra no arquivo.

No começo do arquivo, vem o lay-out dos dados. Seria o header, com o seguinte formato

Posição	Significado
0	Assinatura
1-3	Data da última atualização
4-7	Número de registros no arquivo
8-9	Tamanho da estrutura de cabeçalho
10-11	Tamanho de cada registro
12-31	Reservado
...	campo−1
...	
...	Campo_n
...	X"0D"(terminador dos dados)

Cada campo, por sua vez, está assim descrito (ocupando 32 bytes cada)

Posição	Significado
0-10	Nome do campo em ASCII
11	Tipo de campo (A, N, L, D, M)
12-15	Reservado
16	Tamanho do campo
17	Quantidade de decimais
18-31	Reservado

Notas

- Os tipos de campo aceitáveis são: Alfanumérico com tamanho de até 256 bytes, numérico podendo ou não ter decimais, lógico, data e memorando.
- Algumas assinaturas importantes: x'02' é foxbase, h'03' é dbase sem arquivos memo, x'30' é VisualFoxPro, x'83' é dbase com campos memo, etc.
- O tamanho de cada registro é a soma dos tamanhos dos campos do arquivo, mais 1 (que é usado como sinalizador de exclusão.
- As páginas em dbase ocupam 512 bytes.

 ©88-08, Pedro Kantek

- Depois do header, vem os dados, gravados sequencialmente, um depois do outro. Estrutura dos índices

Os índices em dbase são guardados em arquivos separados, o que – como tudo na vida – tem vantagens e desvantagens. Vantagem: por definição, índices são descartáveis, e não precisam ser guardados. Desvantagem: eles podem ser ignorados em operações de inclusão / exclusão, o que diminui a confiabilidade deles.

Os índices são criados através de um comando dbase que recebe os parâmetros que ajudarão a montar o índice, lê o arquivo de dados e constrói o índice. Ele usa uma estrutura de árvore B+, que vem a ser uma árvore B, na qual as informações de chave estão apenas nas folhas, o que permite a leitura "sequencial"do arquivo de dados em qualquer ordem determinada por um arquivo de índices.

O arquivo NDX tem um header que contém:

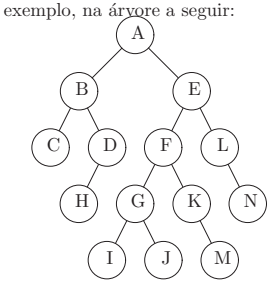
1. O número da página inicial do índice
2. O número total de páginas do índice
3. O comprimento da chave que construiu este índice
4. O número de chaves por página
5. O tipo da chave (numérica ou alfa)
6. Tamanho das entradas de índice (4 para a próxima entrada + 4 para o número do registro no arquivo DBF + expressão da chave arredondada para múltiplo de 4)
7. A string que define a chave deste arquivo de índices

Cada uma das demais páginas, tem um campo de 4 bytes com o número de entradas válidas na página corrente e a partir do byte 4, um array de entradas de chaves. Cada elemento deste array é um apontador para o nível inferior, o número do registro no arquivo DBF e o valor da chave.

10.13 Para ler depois...

10.13.1 Árvores com costura

Em uma árvore binária convencional, percebe-se a existência de um grande número de nodos nos quais os endereços (a esquerda e a direi-ta) são iguais ao terminador. Por exemplo, na árvore a seguir:



Nesta árvore, existem 14 nodos. Destes, apenas 5 nodos (A, B, E, F e G) contém informação útil sem nenhum desperdício. Isto significa que além do conteúdo do nodo,

©88-08, Pedro Kantek

existem os endereços à esquerda e à direita válidos. Já 3 nodos (D, L e K) contém 1 endereço válido e outro inválido, já que é igual a terminador. Finalmente, 6 nodos (C, H, I, J, M e N) contém dos dois endereços igual ao terminador.

Em uma conta simplista, se cada nodo tivesse 3 bytes, a árvore como um todo ocuparia $14 \times 3 = 52$ bytes, dos quais 14 seriam de conteúdos. Sobrariam 28 bytes para apontadores, dos quais 13 estariam ocupados e 15 conteriam o terminador. O número 13 se explica porque há 5 nodos com endereços válidos à esquerda e à direita (10 bytes) e 3 nodos com informação à direita OU à esquerda (3 bytes). Da soma de 10 com 3 chega-se ao número acima mostrado.

Uma maneira de usar este espaço desperdiçado está no conceito de árvores com costura. Como sempre, o que se ganha de um lado se perde de outro (e vice versa), mas a idéia é bem interessante. Este conceito pode ser usado – por exemplo – quando há necessidade de se melhorar o tempo de visitação da árvore, ainda que a despeito de um aumento na dificuldades dos algoritmos de inclusão e exclusão de nodos na árvore.

Talvez seja indicado em árvores que não sofram muitas modificações e que ao contrário tenham muitas operações de visitação.

Quando o endereço do filho (esquerda ou direita ou ambos) de um nodo for igual a terminador, substitui-se este valor por alguma outra coisa que seja mais interessante para a finalidade da árvore. Naturalmente aqui surge uma dificuldade adicional que é guardar a informação de que o valor que está lá significa um endereço válido ou alguma outra coisa (era o terminador e como a árvore é com costura, outra coisa tem lá).

Eis a primeira dificuldade: cada nodo terá que ter 2 indicadores binários dizendo se o valor do endereço é válido ou é uma costura.

Assim, o nodo que era:

& filho esquerdo	conteúdo do nodo	& filho direito

Passa a ser agora:

0=endereço 1=costura	endereço ou costura	conteúdo do nodo	endereço ou constura	0=endereço 1=costura

Note-se que os campos indicadores (a quem chamaremos costuraD e costuraE) são apenas binários e portanto ocupam apenas 1 bit.

Qualquer coisa pode ser colocada nos campos de costura, lembrando apenas que nem todo nodo terá estes campos. Apenas os nodos cujos endereços de filho a esquerda, a direita ou ambos tinham o valor igual ao terminador. Assim, não pode ser informação única (que não exista em outro local da estrutura, ou que não possa ser recalculada). Se assim fosse, apenas alguns nodos a teriam, e a estrutura estaria incompleta.

Costuma-se colocar informação que deva ser calculada, e que – se presente – acelera os algoritmos, em geral, de busca. Este é o exemplo que vai ser aqui desenvolvido, colocando-se em cada endereço esquerdo que era igual a terminador a informação do endereço do antecessor do nodo em questão. Já se o terminador estava no endereço direito, colocar-se-á o endereço do sucessor do nodo.

Desta maneira, os algoritmos de busca de sucessor e antecessor, terão que estar preparados para seguir um de dois caminhos: se o endereço é de filho (costuraX = 0), deve-se usar a indireção para procurar o antecessor/sucessor. Entretanto, se o endereço é de costura (costuraX = 1), basta recuperar o endereço e devolvê-lo. Não há necessidade de nenhuma indireção.

Veja-se uma árvore exemplo, classificada em em-ordem:

Note-se que esta árvore é a convencional, em-ordem, para a seqüên-cia de nodos: 3, 5, 8, 10, 12, 14, 15, 20 e 30.

Veja-se agora, como ficaria a mesma árvore na modalidade costurada:

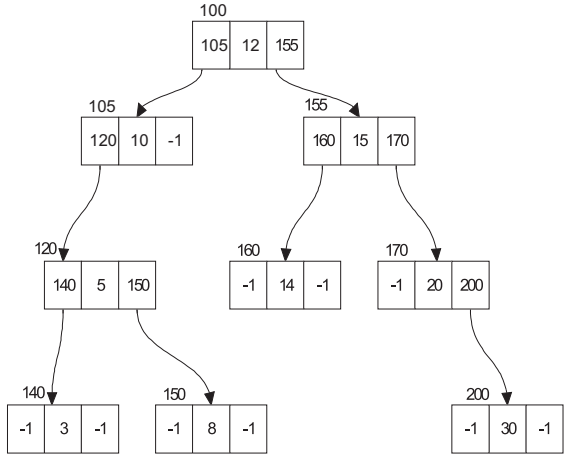


Figura 10.23: Uma árvore com costura

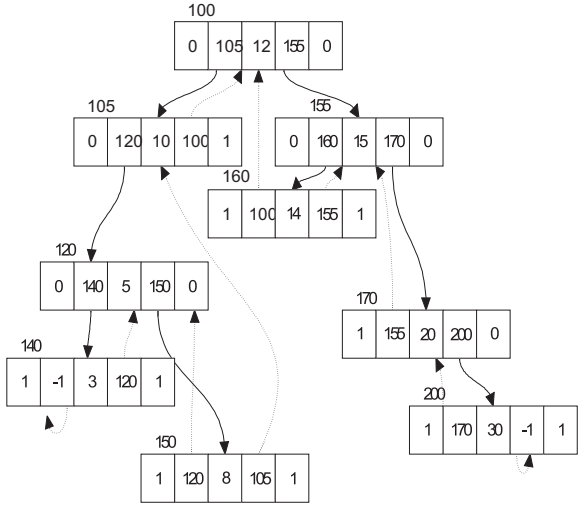


Figura 10.24: Mais uma árvore com costura

Algoritmos de árvore com costura

Os algoritmos agora são Esta função devolve o endereço em M do sucessor de NODOX. NDODOX é um endereço de nodo.

```
1: inteiro função SUC(inteiro NODOX)
2: ptx← direita(NODOX)
3: se ∼ costura-direita(NODOX) então
4:    enquanto ∼ costura-esquerda(ptx) faça
5:       ptx← esquerda(ptx)
6:    fimenquanto
7: fimse
8: retorne (ptx)
9: fim função
```

A função a seguir devolve o endereço em M do antecessor de NODOX, que é um endereço de nodo

```
1: inteiro função ANTEC(inteiro NODOX)
2: ptx← esquerdo(NODOX)
3: se ∼ costura-esquerda(NODOX) então
4:    enquanto ∼ costura-direita(ptx) faça
5:       ptx← direita(ptx)
6:    fimenquanto
7: fimse
8: retorne (ptx)
9: fim função
```

Sorte e Azar do aluno atrasado Um dia George Dantzig (um dos pais da Pesquisa Operacional) chegou atrasado a uma aula. A sala estava vazia e havia 2 problemas no quadro. Achando que eles eram a lição de casa, Dantzig os copiou e trabalhou duro sobre eles. Eram difíceis. Alguns dias depois ele os entregou ao professor desculpando-se pela demora. O professor grunhiu e guardou os problemas na pasta. Diversas semanas depois, Dantzig recebeu uma visita do professor. Era pela manhã e o professor estava assustado. Queria ver como Dantzig havia resolvido os problemas. Eles não eram lição de casa. Eles eram 2 problemas famosos na literatura por não terem sido ainda resolvidos! Mais tarde, quando Dantzig entrou no doutorado e começou a procurar um assunto para sua tese, o professor tranquilizou-o: aqueles dois problemas resolvidos eram a tese.

EXERCÍCIO 208 Pegue uma folha de papel e escreva o nome de seus 3 vizinhos da esquerda, dos seus 3 vizinhos da direita e de 4 vizinhos que estejam sentados à sua frente. Com essa lista de 10 nomes escreva uma ABP cujo chave de acesso seja o nome. O primeiro nome que você escreveu é o nodo raiz.

EXERCÍCIO 209 Suponha uma ABP de 200 nodos. Estime a complexidade de acesso (em termos de número de comparações) caso a ABP esteja balanceada e degenerada. Idem para uma árvore de 10 e outra de 10.000 nodos.

EXERCÍCIO RESOLVIDO 26 Seja um problema extraído de [For76, pág 375]. Existem 8 moedas. Uma delas é mais leve ou mais pesada que as outras, todas as outras 7 tem o mesmo peso. Há uma balança de pratos que pode ser usada apenas 3 vezes. Seu objetivo é identificar qual a moeda diferente e qual a diferença (mais leve ou mais pesada). Dica: use uma árvore...

Parece óbvio que se o que se dispõe é uma balança de pratos, devemos começar a testar um conjunto de moedas contra outro conjunto de moedas. A questão é o que colocar no prato 1 e o que colocar no prato 2?

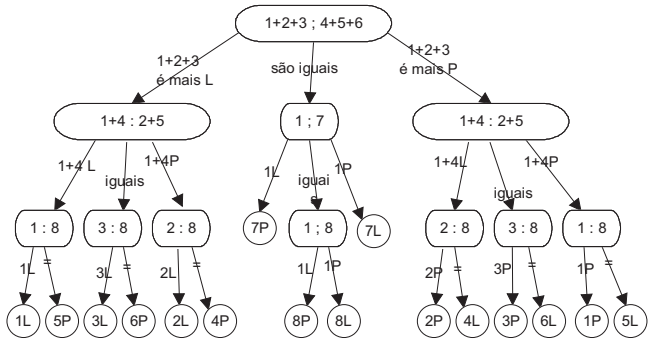


Figura 10.25: Árvore do problema das 8 moedas

Desafio

- Construir um arquivo qualquer usando o gerenciador dBase de banco de dados. Popular o arquivo com poucos registros. Construir diversos índices. Listar através de *dump* todos os arquivos envolvidos. Localizar as características importantes.
- Usando um utilitário de manipulação binária (DEBUG sob DOS ou HEXEDIT sob windows), abrir os arquivos acima criados e com muito cuidado alterar as diversas informações localizadas, percebendo em seguida qual o efeito disso quando o arquivo voltar a ser manipulado pelo dBase. Se feito direitinho, este desafio é diversão na certa.

Capítulo 11

Heap

11.1 Heap

Seja uma lista de prioridades. Para ela, podemos ter inúmeros usos no dia a dia: jobs esperando CPU, pessoas querendo falar com alguém, lista de espera de transplante.

As operações fundamentais neste tipo de estrutura podem ser:

1. Quem tem mais prioridade ?
2. Chegou novo elemento (acompanhado de sua prioridade)
3. Remoção de elemento mais prioritário
4. Alteração de prioridade

A seguir, tem-se uma implementação usando uma lista convencional (um vetor) ainda porcima, não ordenado. Defina-se uma tabela X contendo 2 colunas, onde a coluna 1 é a identificação do elemento e a coluna 2 é a prioridade do mesmo. Neste atributo considerar-se-á 0 a prioridade mais baixa e 9 a mais alta. Defina-se também uma variável global de nome ULTIMO que contém a última linha usada na estrutura, abaixo da qual não há informação relevante na matriz. Uma possível definição seria:

```
inteiro X[nn,2]
inteiro ULTIMO
```

Seja agora alguns algoritmos necessários para responder a questões que se possam formular sobre essa estrutura.

1. Quem é mais prioritário ?

```
1: inteiro função QUEMTEMPRTY
2: inteiro MAX ← 9999
3: inteiro QUEMMAX
4: inteiro K ← 1
5: enquanto (K < ULTIMO) faça
6:    se X[K,2] > MAX então
7:       MAX ← X[K,2]
8:       QUEMMAX ← K
9:    fimse
10:   K++
11: fimenquanto
```



```
12: devolva QUEMMAX
13: fim função
```

A complexidade associada ao algoritmo que localiza o item mais prioritário é $O(n)$.

2. Chegou novo elemento

```
1: inteiro função CHEGOUNOVO (inteiro ID, NP)
2: ULTIMO++
3: X[ULTIMO] [1] ← ID
4: X[ULTIMO] [2] ← NP
5: devolva ULTIMO
6: fim função
```

A complexidade associada é $O(1)$.

3. Remoção do mais prioritário

```
1: inteiro função REMOVEMAIS
2: inteiro QUEM,J
3: QUEM ← QUEMTEMPRTY
4: devolva (X[QUEM] []
5: para J de QUEM até ULTIMO-1 faça
6:    X[J] [] ← X[J+1] []
7: fimpara
8: ULTIMO–
9: fim função
```

A complexidade associada $O(n) + 1/2$ ou $O(n) = O(n)$.

4. Alteração de prioridade

```
1: inteiro função ALTERAPRTY (inteiro FULANO,NOVAPRTY)
2: inteiro K
3: K ← 1
4: logico AINDANAO ← .V.
5: enquanto AINDANAO ∧ K ≤ ULTIMO faça
6:    se X[K] [1] = FULANO então
7:       X[K] [2] ← NOVAPRTY
8:       AINDANAO ← .F.
9:    fimse
10:   K++
11: fimenquanto
12: devolva ¬ AINDANAO
13: fim função
```

A Complexidade Associada: $O(n)$

11.2 Implementação usando lista ordenada

Agora será usada uma lista em ordem descendente por prioridade. Assim, o mais prioritário é o primeiro da lista.

1. Quem é mais prioritário ?

```
1: inteiro função QUEMTEMPRTY
2: devolva X[1] [1]
3: fim função
```

©88-08, Pedro Kantek

A complexidade associada: O(1)

2. Chegou novo elemento

```
1: inteiro função CHEGOUNOVO (inteiro ID, NP)
2: inteiro K
3: ULTIMO++
4: NAOACHOU ← .V.
5: enquanto (NAOACHOU ∧ K ≤ ULTIMO) faça
6:    se X[K] [2] < NP então
7:       NAOACHOU ← .F.
8:    fimse
9:    K++
10: fimenquanto
11: para J de K até ULTIMO-1 faça
12:    X[J+1][] ← X[J][]
13: fimpara
14: X[K] [1] ← ID
15: X[K] [2] ← NP
16: devolva K
17: fim função
```

Complexidade associada: O(n)

3. Remoção do mais prioritário

```
1: inteiro função REMOVEMAIS
2: inteiro QUEM,J
3: devolva (X[1] []
4: para J de 1 até ULTIMO-1 faça
5:    X[J] [] ← X[J+1] []
6: fimpara
7: ULTIMO–
8: fim função
```

Complexidade associada O(n)

4. Alteração de prioridade

```
1: inteiro função ALTERAPRTY (inteiro FULANO,NOVAPRTY)
2: inteiro K
3: K ← 1
4: logico AINDANAO ← .V.
5: enquanto AINDANAO ∧ K ≤ ULTIMO faça
6:    se X[K] [1] = FULANO então
7:       X[K] [2] ← NOVAPRTY
8:       AINDANAO ← .F.
9:    fimse
10:   K++
11: fimenquanto
12: devolva ¬ AINDANAO
13: ORDENATABELAX {⇐= a tabela X deve ser reordenada}
14: fimfunção
```

Complexidade Associada: O(n) + O(log n) = O(n)

 ©88-08, Pedro Kantek

Em resumo

Operação	Lista não ordenada	Lista ordenada
Seleção	O(n)	O(1)
Inserção	O(1)	O(n)
Remoção	O(n)	O(n)
Alteração prioridade	O(n)	O(n)

11.3 heap binário

E uma estrutura SEQÜENCIAL (vizinhança), que permite implementar listas de prioridade com desempenho MELHOR do que lista (ordenada e não ordenada) Trata-se de um vetor (ou LISTA) na qual a propriedade a seguir sempre é válida:

PRTY(S_i) $\leq PRTY(S_{rfloor(i/2)}) \forall i, 1 \leq i \leq tamanho(S)$

Para efeito de entendimento da propriedade acima, podemos dispor os elementos do vetor em uma árvore binária completa. Neste caso, a proprie- dade acima poderá ser expressa como:

> Para qualquer elemento desta árvore, a prioridade do pai terá que ser maior do que a prioridade dos dois filhos.

Exemplos:
95, 60, 78, 39, 28, 66, 70 e 33 é ou não é um heap binário ? (sim)
19, 15, 9, 11, 5, 8, 4, 5, 1, 3, 6, 7, 6, 3, 2 é heap binário ? (não)

Para esta estrutura, valem as seguintes complexidades:

Seleção $O(1)$ - trata-se apenas consultar o primeiro elemento

Inserção $O(\log_2 n)$ - A inserção se dá sempre ao final da lista (vetor), ainda que depois, um novo local para o elemento recém incluído precise ser achado.

Remoção $O(\log_2 n)$ - exclui o primeiro, e preenche o espaço liberado.

Alteração de prioridade O(log n). Aqui mora o problema todo (seja a inclusão, seja a exclusão)...

Para mudar a prioridade, há duas operações: SUBIR e DESCER. A SUBIR pega o elemento no final da lista e vem subindo com ele até achar o lugar dele. A DESCER, pega o primeiro elemento e vem descendo até ele encontrar seu lugar.

Exemplo: Como incluir o 79 no heap binário acima ? (insere atrás e SOBE)

Algoritmos básicos SUBIR: permutar o indivíduo com seu pai, enquanto a PRTY(pai) for menor. Quando isto for FALSO, acabou.
DESCER: permutar o pai com o filho de MAIOR PRTY, sempre que a PRTY do pai for menor. Quando isto for falso para os 2 filhos, acabou...

EXERCÍCIO 210 Desenhe a árvore correspondente a lista heap binário: 90, 85, 84, 52, 10, 60, 71, 25, 31, 2, 7, 40. A propriedade (1) está atendida ?

EXERCÍCIO 211 Faça a inclusão do elemento 99, primeiro como filho do nodo 60 (e irmão do 40) e depois coloque-o em seu local definitivo. Transcreva como ficou a lista heap binário

EXERCÍCIO 212 A lista: 93, 70, 68, 55, 53, 52, 31, 27, 25, 23, 24, 13, 12, 11, 10 é uma lista heap binário ?

©88-08, Pedro Kantek

EXERCÍCIO 213 A lista: 93, 70, 68, 55, 53, 52, 11, 27, 25, 24, 23, 13, 12, 31, 10 não é uma lista heap binário. Onde está o furo ?

EXERCÍCIO 214 Uma lista em rigorosa ordem decrescente, como por exemplo 90, 80, 70, 60, 58, 55, 41, 11, 9, 8 é sempre uma lista heap binário ? Porque ?

EXERCÍCIO 215 Dada a lista heap binária: 28, 26, 24, 22, 20, 18, 16, 14, 12, 11, 10, 9, 6, 3
Modifique a prioridade do individuo 28, para 8. Como fica a lista heap binária?

EXERCÍCIO 216 Modifique a prioridade de 26 para 27. Mudou alguma coisa ? Porquê ?

EXERCÍCIO 217 Modifique a prioridade do 22 para 2. Como ficou a nova lista heap binária ?

Algoritmos

Para esta coleção de algoritmos, vao-se usar as seguintes estruturas de dados e variáveis: a estrutura elemento define os dados que fazem parte de cada nodo da estrutura. PRIOR é a prioridade. Assume-se que a lista de heap binário pode ter ate 1000 elementos e ULTIMO indica a última linha usada em TABELA. A função PERMUTA(i,j) troca os elementos i e j de lugar, um com o outro.

```
inteiro ULTIMO {indica a a ultima linha usada. Comeca com 0}
inteiro HE[1000] [2] {coluna 1 = ID, coluna 2 = PRTY}
```

Subir

Eleva a prioridade de um determinado indivíduo, ate ele alcançar o seu lugar correto, mantendo a estrutura como uma lista heap binário. O valor i e a posição do individuo a ter a sua prioridade alterada, que já foi colocada em HE.

```
1:  função SUBIR(inteiro I)
2:  inteiro J
3:  J ← ⌊ (I / 2) {j contém a metade de i, a menos das decimais}
4:  se (J ≥ 1) então
5:     se HE[I][2] > HE[J][2] então
6:        PERMUTA(I,J) {os elementos de ordem i e j trocam de lugar}
7:        SUBIR(J) {veja-se que esta função e recursiva}
8:     fimse
9:  fimse
10: fimfunção
```

Descer

Diminui a prioridade do elemento i, até ele encontrar seu local correto.

```
1:  função DESCER(inteiro I)
2:  inteiro J
3:  J ← 2 * I
4:  se J ≤ ULTIMO então
5:     se J < ULTIMO então
6:        se HE[J+1][2] > HE[J][2] então
7:           J++
8:        fimse
9:     fimse
10:    se HE[I][2] < HE[J][2] então
11:       PERMUTA(I,J)
12:       DESCER(J) {função recursiva}
13:    fimse
14: fimse
15: fimfunção
```

Como se pode observar nos algoritmos as complexidades de subir e descer equivalem a percorrer um caminho por uma árvore binária, donde eles tem complexidade O(log n).

Insere

Insere o novo elemento NOVO na lista de prioridades.

```
1:  função INSEREHEAP(inteiro ID, PR)
2:  se ULTIMO < 1001 então
3:     ULTIMO++
4:     HE[ULTIMO][1] ← ID
5:     HE[ULTIMO][2] ← PR
6:     subir(ULTIMO)
7:  senão
8:     "estouro do heap binário"
9:  fimse
10: fimfunção
```

Exclusão

A retirada do elemento de maior prioridade na lista (na lista: o primeiro, na árvore: o raiz) deixa um buraco vazio na mesma. Este buraco deve ser preenchido com o último elemento da lista (a fim de manter a compactação dos dados dentro da tabela), e após essa inversão ser feita, o elemento que ocupa agora a primeira posição precisa descer (algoritmo descer) ate encontrar seu local correto.

```
1:  função EXCLUIHEAP()
2:  se ULTIMO > 0 então
3:     devolva HE[1][1] {devolve o ID}
4:     HE[1][1] ← HE[ULTIMO][1]
5:     HE[1][2] ← HE[ULTIMO][2]
6:     ULTIMO–
7:     descer(1)
8:  senão
9:     "tabela vazia"
10: fimse
11: fimfunção
```

A exclusão tem complexidade $O(\log_2 n)$. Se todas as linhas do heap binário tiverem que ser excluídas, isto terá complexidade $O(n.\log_2 n)$.

Criação

Dada uma lista qualquer, para que esta seja transformada em um heap binário, basta rearranjar seus elementos para que a propriedade (1) seja satisfeita para todos eles. Isto pode ser providenciado a partir de algumas considerações. Se pensarmos que uma lista ordenada é uma lista heap binário, uma alternativa seria ordenar a dita lista. Ao final deste processo ela é uma lista heap binária.

Como sabemos a ordenação mais barata, custa $O(n.\log_2 n)$. Bom, mas ainda caro. Por outro lado, se pensarmos que:

1. Quem não tem filhos, já está no seu lugar definitivo
2. Apenas a primeira metade da lista heap binário tem filhos

Bastará que para os elementos que tem filhos (a primeira metade) se faça a descida deles até o seu local correto. Ficaria

```
1: função CRIAR(N) {N é o tamanho da lista}
2: inteiro I,J
3: I ← ⌊ (N / 2)
4: para J de I até 1 passo -1 faça
5:    descer (J)
6: fimpara
7: fimfunção
```

Alternativamente, usando subir, o mesmo algoritmo pode ser escrito como

```
1: função CRIARX(N) {N é o tamanho da lista}
2: inteiro J
3: para J de 1 até N faça
4:    subir (J)
5: fimpara
6: fimfunção
```

Note entretanto que CRIARX tem complexidade $O(n.\log_2 n)$, enquanto que CRIAR tem complexidade $O(n)$. A demonstração deste fato se encontra em [Szwa94, pag. 185].

11.3.1 Aplicações para heap binário

Encontre o k-ésimo maior/menor elemento em um conjunto de números Solução 1

- Leia n elementos em um array (ainda desordenado)
- Crie um heap binário para os n elementos
- Execute k vezes a operação de exclusão
- O último elemento excluido é o maior/menor
- Para excluir o menor, mudar a ordem de criação do heap binário
- Complexidade: $O(n.\log_2 n)$

Solução 2:

- Construa um heap binário usando os primeiros k elementos
- Compare os elementos que sobraram com o topo do heap binário
- Substitua o heap binário se o novo elemento é maior do que o topo
- O topo do heap binário ao final é o k-ésimo maior
- Complexidade: $O(n.\log_2 k)$

 ©88-08, Pedro Kantek

Suponha que você precisa estudar um sistema de filas Sejam elas de supermercado, banco, aeroporto, central telefônica, jobs em um SO, bilheteria de estádio, metrô, etc. Em todos estes sistemas, entidades chegam, entram na fila, vão sendo atendidos e vão embora. Deseja-se conhecer o funcionamento do sistema que é estudado a partir de probabilidades (chegada, atendimento, etc etc). O que se quer saber é tempo médio de espera na fila, tamanho médio da fila, influência do número de atendentes e da modalidade de fila etc.

Uma possibilidade é trabalhar com tempo contínuo e com funções de probabilidade. Isto fica difícil para valores maiores de filas/clientes/tempos.

A solução é trabalhar com sistemas discretos e simular o sistema todo. Uma simulação comeca com a criação de um tique de relógio que passa a ser a unidade de tempo.

Uma possibilidade é
TIQUE ← 0
enquanto (TIQUE < NNNN)
chegou alguém ?
se sim: calcule disponibilidade de atendedor, encaminhe para
fila ou para atendedor, atualize contadores e tamanhos
de fila etc
se não:
saiu alguém ?
se sim: calcule tempo de atendimento, atualize as filas,
atualize contadores e tamanhos, gere estatísticas
se não:
incremente TIQUE
fim(enquanto)

Esta abordagem funciona, mas tem um problema: em grande parte dos TIQUES nao ocorrerá nenhum evento. Essa situação piora se usarmos MILITIQUES. Aí a simulação arrisca a ficar 1000 vezes mais demorada.

Melhor será, a cada instante, incrementar o TIQUE não em 1 unidade e sim em tantas unidades quantas faltam para o próximo evento (chegada ou partida) Para isto é conveniente manter a fila de chegadas e de saídas como dois heaps binários ordenados por tempo (em tiques). Assim, o programa tem acesso imediato aos 2 próximos eventos no tempo.

EXERCÍCIO 218 Escreva em C++ os programas capazes de implementar os algoritmos acima estudados. Faça testes de mesa com um universo pequeno.

EXERCÍCIO 219 Escreva programa C++ que desenhe na tela uma árvore a partir de uma lista fornecida. Com este programa será fácil verificar se uma dada lista é heap binário. Estabeleça como limite máximo de tamanho 15 elementos. Use a seguinte localização de (linha,coluna) para escrever os elementos:

```
raiz: (0,39)
filhos: (6,19) e (6,59)
netos: (13,9), (13,29), (13,49) e (13,69)
bisnetos: (20,4),(20,14), (20,24),(20,34), (20,44),(20,54), (20,64) e (20,74).
```

EXERCÍCIO 220 usando o programa de D2 acima, implemente uma verificação da propriedade 1 e imprima embaixo da árvore desenhada, "É heap binário"ou "NÃO É heap binário".

EXERCÍCIO 221 Reescreva e implemente todas as funções de heap binário sem usar recursividade

©88-08, Pedro Kantek

EXERCÍCIO 222 Reescreva todos os algoritmos vistos usando origem dos índices=0

EXERCÍCIO 223 Reescreva todos os algoritmos aqui estudados para 3-heaps (heaps trinários)

Desafio

- Escreva e implemente os algoritmos de subir e descer. Teste-os até ter certeza de que funcionam bem.
- Escreva uma função de criação de heap. Faça uma encarnação usando subir e outra usando descer. Compare o tempo gasto por ambas na criação de um heap com 10.000 números aleatórios.

©88-08, Pedro Kantek

©88-08, Pedro Kantek

Capítulo 12

Ordenação

12.1 Ordenação

Importância do estudo da ordenação

- Na década de 70, James Martin sugeriu que 40% do tempo de TODOS os computadores é gasto neste algoritmo.
- O conceito de ordem é fundamental em programação
- Excelente tópico para estudar algoritmos e estudar comportamentos e complexidades

Antes de começarmos a estudar os diversos algoritmos de classificação, vamos definir o problema:

Seja X um conjunto composto por i elementos entre os quais se pode estabecer uma relação de ordem. Dados X_i e X_j com $i \neq j$, sempre pode-se estabelecer $X_i > X_j$ ou $X_i = X_j$ ou $X_i < X_j$.

- X estará em ordem crescente se e somente se $X_i \leq X_j, \forall i < j$.
- X estará em ordem estritamente crescente se e somente se $X_i < X_j \forall i < j$.
- X estará em ordem decrescente se e somente se $X_i \geq X_j, \forall i < j$.
- X estará em ordem estritamente decrescente se e somente se $X_i > X_j \forall i < j$.

Um algoritmo de ORDENAÇÃO é aquele que recebe um X qualquer (possivelmente desordenado), permuta seus elementos e devolve X em ordem.

Tipicamente fazem parte de X_i um conjunto de informações. Neste caso, haverá uma parte de X_i denominada CHAVE e identificada por k (k=key) pela qual se fará a ordenação. Embora devamos ter em mente a existência dos outros campos, apenas a chave será tratada nos algoritmos.

Existem 2 classes de algoritmos de ordenação: os que trazem todos os dados para a memória (sort interno) e os que ordenam dados em mídia magnética seqüencial (sort externo). Note que neste caso o algoritmo apenas acessa uma parte do subconjunto total.

Um algoritmo de ordenação é estável se no caso particular em que se k[i] = k[j], a ordem original dos dados é preservada. É sempre bom que um método de ordenação seja estável.

Antes de prosseguir, vale uma lembrança. Para conjunto de dados pequenos (quanto?) não há necessidade de grandes pesquisas e deve-se usar o algoritmo mais simples possível.

Primeiro critério: buscar efetuar as trocas in-situ. Por isso, o algoritmo a seguir deixa a desejar. (Ele movimentam os dados de um vetor para o outro)

Algoritmo de movimentação (Alg. 1) O algoritmo a seguir é um que apresenta uma ordenação ineficiente

```
1: algoritmo MOVIMENTA (inteiro V[1000])
2: inteiro Y[1000]
3: inteiro MAIOR, QUAL
4: inteiro I,J
5: leia X
6: para I de 1 até 1000 faça
7:    MAIOR ← −∞
8:    para J de 1 até 1000 faça
9:       se V[J] > MAIOR então
10:         MAIOR ← X[J]
11:         QUAL ← J
12:      fimse
13:   fimpara
14:   Y[I] ← X[QUAL]
15:   V[QUAL] ← −∞
16: fimpara
17: devolva Y
18: fim{algoritmo}
```

PERGUNTAS: 1. É estável ? SIM
2. Qual a complexidade deste algoritmo ? $O(n^2)$

EXERCÍCIO 224 Exercício: faça um chinês com 4, 12, 8, 3, 9, 1, 5

Método de inserção direta (alg. 3) Conceitualmente este método mantém 2 seqüências lado a lado. No início a ordenada está vazia e a desordenada cheia. Elementos são retirados de uma e inseridos na outra.

Em alto nível:
para i de 2 até tamanho(n)
x ← a[i]
insira x no local certo em a[1]...a[i]

Ordenação por inserção:

```
1: Algoritmo INSERÇÃO (inteiro V[1000])
2: inteiro I, J, X
3: I ← 2
4: enquanto (I ≤ 1000) faça
5:    X ← V[I]
6:    J ← I - 1
7:    enquanto ((J≥ 1) ∧ (V[J]>X) faça
8:       V[J+1] ← V[J]
9:       J–
10:   fimenquanto
11:   V[J+1] ← X
12:   I++
```

©88-08, Pedro Kantek

```
13: fimenquanto
14: devolva V
15: fim algoritmo
```

Este método é bom, se o vetor estiver quase-ordenado. A ordenação é estável. Um melhoramento evidente, é a utilização da busca binária para a parte que já está ordenada Ordenação por inserção usando busca binária

```
1:  Algoritmo INSERÇÃOBIN (inteiro V[1000])
2:  inteiro I,J,X,L,R,M,IND
3:  I ← 2
4:  enquanto (I ≤ 1000) faça
5:     X ← V[I]
6:     L ← 1
7:     R ← I
8:     enquanto (L < R) faça
9:        M ← ⌊ (L+R) ÷ 2
10:       se (X > V[M]) então
11:          L ← M+1
12:       senão
13:          R ← M
14:       fimse
15:    fimenquanto
16:    IND ← R
17:    J ← I-1
18:    enquanto (J ≥ IND) faça
19:       V[J+1] ← V[J]
20:       J–
21:    fimenquanto
22:    V[J+1] ← X
23:    I++
24: fimenquanto
25: devolva (X)
26: fim algoritmo
```

Ambos os métodos acima tem complexidade $O(n^2)$ a menos de algumas constantes

Método de seleção direta (Alg. 4)

```
Em alto nível
para i de 1 até (tamanho n) - 1
   ache o índice do menor elemento entre a[i] e a[k]
   troque a[i] com a[k]
fim{para}
```

Ordenação por seleção direta:

```
1:  Algoritmo SELEÇÃO(inteiro V[1000])
2:  inteiro I,J,COR,INUI
3:  I ← 1
4:  enquanto (I ≤ 1000) faça
5:     COR ← V[I]
6:     INUI ← I
7:     J ← I + 1
8:     enquanto (J ≤ 1000) faça
9:        se V[J] < COR então
```

 ©88-08, Pedro Kantek


```
10:          COR ← V[J]
11:          INUI ← J
12:       fimse
13:       J++
14:    fimenquanto
15:    TROCA (I, INUI)
16:    I++
17: fimenquanto
18: devolva (V)
19: fim algoritmo
```

Este método tem desempenho menos dependente da ordem original do que o de inserção. É ligeiramente superior que aquele para conjuntos desordenados.

Método de Permutação (bolha, alg. 5) Ordenação por bolha

```
1:  Algoritmo BOLHA (inteiro V[1000])
2:  inteiro I, J
3:  I ← 1000
4:  enquanto (I ≥ 2) faça
5:     J ← 2
6:     enquanto (J ≤ I) faça
7:        se (V[J-1] > V[J]) então
8:           TROCA (J, J-1)
9:        fimse
10:       J++
11:    fimenquanto
12:    I–
13: fimenquanto
14: devolva V
15: fim algoritmo
```

Um melhoramento especial pode ser implementado testando-se quando houve a inversão indicativa de que os elementos estavam fora de ordem

Ordenação por bolha melhorado

```
1:  Algoritmo BOLHAMELHORADA (inteiro V[1000])
2:  inteiro I, J
3:  lógico MUDOU ← VERDADEIRO
4:  I ← 1000
5:  enquanto (I ≥ 2) ∧ MUDOU faça
6:     J ← 2
7:     MUDOU ← FALSO
8:     enquanto (J ≤ I) faça
9:        se (V[J-1] > V[J]) então
10:          TROCA (J, J-1)
11:          MUDOU ← VERDADEIRO
12:       fimse
13:       J++
14:    fimenquanto
15:    I–
16: fimenquanto
17: devolva (V)
18: fim algoritmo
```

©88-08, Pedro Kantek

Um outro melhoramento sugere a mudança de sentido nos testes, a fim de levar o maior e o menor a cada passada, e não apenas um deles, como acima. Surge então o SHAKER SORT

Algoritmo shaker sort

```
1: Algoritmo SHAKER (inteiro V[1000]) (alg. 5B)
2: inteiro L,K,R,J
3: L ← 2
4: K ← R ← 1000
5: repita
6:    J ← R
7:    enquanto (J ≥ L) faça
8:       se (V[J-1] > V[J]) então
9:          TROCA (J-1, J)
10:         K ← J
11:      fimse
12:      J–
13:   fimenquanto
14:   L ← K + 1
15:   J ← L
16:   enquanto (J ≤ R) faça
17:      se (V[J-1] > V[J]) então
18:         TROCA (J-1, J)
19:         K ← J
20:      fimse
21:      J++
22:   fimenquanto
23:   R ← K - 1
24: até (L > R)
25: devolva (V)
26: fim algoritmo
```

12.2 Métodos Particulares

Denominei método particular a aquele que – embora muito rápido – não se aplica a conjuntos genéricos de chaves. Assim, para poder usar estes algoritmos, há que se submeter a alguma restrição. Entretanto, se isso for possível, o resultado é o melhor possível em termos de complexidade.

12.2.1 Ordenação por contagem

A principal restrição deste método é que a chave tem que ser inteira, e presumivelmente, deva ter uma chave máxima que seja "pequena". A razão destas restrições é que a própria chave vai ser usada como indexador temporário dentro do algoritmo. Assim, uma chave não inteira impede o uso do algoritmo. Da mesma maneira, uma chave inteira mas muito grande, pode inviabilizar o array temporário que é necessário construir. A complexidade associada é excelente, e igual a $O(n+k)$, onde n é o número de inteiros a ordenar e k é o inteiro de maior valor do conjunto. A complexidade é portanto linear. A requisição de espaço para este método é de dois vetores com tamanho n e um vetor com tamanho k. Uma vantagem deste método é que a ordenação é estável.

O algoritmo vai ser apresentado através de um exemplo: Seja o seguinte conjunto de chaves a ordenar:

1	2	3	4	5	6	7	8	9	10	11
1	8	5	12	3	4	3	5	2	5	5

Passos do algoritmo

Contagem das chaves Deve-se percorrer o conjunto a ordenar, contando as ocorrências de cada chave e armazenando-as em um vetor auxiliar. Fica:

1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	1	4	0	0	1	0	0	0	1

O significado deste vetor é que, por exemplo, há uma chave valendo 1, uma chave valendo 2, duas chaves valendo 3, uma chave valendo 4, quatro chaves valendo 5 e assim por diante.

Note que o vetor de contagens terá o número de elementos igual à maior chave do conjunto de entrada (neste exemplo, 12). Pode, portanto, ser muito maior do que o vetor original a classificar.

acumulação A seguir, o vetor de contagens deve ser acumulado. Fica

1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	5	9	9	9	10	10	10	10	11

Aqui, cada elemento do vetor acumulado é igual a soma do elemento atual do vetor de contagens com o elemento anterior do vetor acumulado (considerando-se a existência virtual de um zero antes do primeiro elemento do vetor acumulado).

alocação dos elementos Agora, a partir das informações obtidas no vetor acumulado, cada chave original deve ser colocada em seu lugar definitivo. Acompanhe:

1	2	3	4	5	6	7	8	9	10	11
1	8	5	12	3	4	3	5	2	5	5

O elemento 5 deve ser retirado do seu local original (o último do vetor de entrada) e colocado na posição indicada pela 5^a ocorrência do vetor acumulado que é 9, e o vetor final fica:

1	2	3	4	5	6	7	8	9	10	11
								5		

Perceba que após esta operação, o índice correspondente ao elemento movido (que foi o 5) deve ser substraído de uma unidade, ficando assim o novo vetor acumulado:

1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	5	8	9	9	10	10	10	10	11

mais um ciclo Façamos o mesmo para o penúltimo número:

1	2	3	4	5	6	7	8	9	10	11
1	8	5	12	3	4	3	5	2	5	5

O elemento 5 deve ser retirado do seu local original (o penúltimo) e colocado na posição indicada pela 5^a ocorrência do vetor acumulado que é 8, e o vetor final fica:

1	2	3	4	5	6	7	8	9	10	11
							5	5		

Perceba que após esta operação, o índice correspondente ao elemento movido (que foi o 5) deve ser substraído de uma unidade, ficando assim o novo vetor acumulado:

1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	5	7	9	9	10	10	10	10	11

Resultado final Eis como ficaria o vetor original devidamente ordenado:

1	2	3	4	5	6	7	8	9	10	11
1	2	3	3	4	5	5	5	5	8	12

Também eis como ficaria o vetor de contagens ao final:

1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	4	5	9	9	9	10	10	10	10

12.2.2 Ordenação por raiz

Este método, conhecido na literatura como *radix sort* era originalmente usado nas classificadoras de cartões, equipamento padrão de qualquer computador até a década de 70. Mais modernamente ele pode ser usado em chaves que tenham todas o mesmo tamanho (físico), como por exemplo, um campo de 10 bytes.

A regra do algoritmo é começar a ordenação pelo byte de menor significância e ordenar o conjunto apenas por esta sub-chave. Encerrada esta parte, os conjuntos resultantes são "empilhados"com os resultados de menor valor sobre os de maior valor. O conjunto assim obtido é submetido a novo radix sort, sendo que agora a chave é o segundo dígito. Procede-se desta maneira até o esgotamento das sub-chaves, quando então o conjunto está ordenado. Acompanhe no exemplo a seguir:

8	3	8		2	0	**0**		2	**0**	0		**2**	0	0
4	0	1		4	0	**1**		4	**0**	1		**3**	1	2
7	2	7		5	1	**1**		5	**1**	1		**4**	0	1
3	1	2		3	1	**2**		3	**1**	2		**5**	1	1
5	1	1	→	9	9	**3**	→	8	**1**	5	→	**6**	7	6
6	7	6		8	1	**5**		7	**2**	7		**7**	2	7
8	1	5		6	7	**6**		7	**2**	7		**8**	1	5
9	9	3		7	2	**7**		6	**7**	6		**8**	3	8
2	0	0		8	3	**8**		9	**9**	3		**9**	9	3

Note-se que este algoritmo pode ser construído passando n vezes por um algoritmo de contagem. n é o número de sub-campos da chave. No exemplo, n seria o número de dígitos do campo numérico. Note-se também que o algoritmo usado em cada uma das etapas precisa ser estável, sob pena de desordenar completamente o conjunto a cada passada.

12.2.3 Algoritmo de Cook-Kim

Este algoritmo é indicado para situações onde o conjunto a ordenar já está quasi-ordenado. Nesta situação ele é o mais rápido algoritmo existente. Define-se uma entrada quasi-ordenada quando apenas alguns elementos estão fora de ordem, enquanto a grande maioria das chaves restantes já está ordenada. Eis as etapas do algoritmo:

Busca de pares desordenados A entrada é percorrida na busca de pares de elementos desordenados. Assim, se a entrada já está quasi-ordenada em ordem crescente, a busca é por vizinhos nos quais $V[k] > V[k+1]$. Cada vez que um par deste tipo é encontrado ele é removido para outro vetor. Após esta remoção o novo par a ser examinado é o antecessor e o sucessor do par retirado.

Ordenação do vetor auxiliar Ao final da etapa anterior, a entrada que permaneceu está 100% ordenada, e como sabemos comporta a grande maioria dos dados da entrada. Agora, o vetor auxiliar produzido é ordenado. (Na proposta original se este vetor tivesse mais de 30 elementos ele seria ordenado via QuickSort e senão usar-se-ia a ordenação por inserção).

Intercalação Com o vetor auxiliar ordenado, agora deve-se apenas intercalar ambos os vetores, para obter a entrada completamente ordenada.

 ©88-08, Pedro Kantek

Desafio

- Implementar os algoritmos aqui estudados.
- Reproduzir a tabela de tempos para as condições particulares de sua implementação, com 100, 1.000 e 10.000 registros aleatórios e quasi-ordenados.

12.3 Ordenação - II parte

Todos os métodos de permutação tem desempenho $O(n^2)$. Um método de ordenação por permutação bastante interessante é o chamado SHELLSORT, inventado por Donald Shell, em 1959, o qual é parecido com o BOLHA, mas utiliza vizinhos distantes e não próximos. Define-se uma seqüência de incrementos, que pode ser qualquer uma, desde que termine com 1. Há seqüências melhores e piores.

Acompanhe no exemplo: Seja ordenar o vetor

```
89, 58, 55, 1, 81, 24, 44, 63, 79
com os índices {6, 4, 1}
vetor original   89, 58, 55,  1, 81, 24, 44, 63, 79
fica em ordem    aa  bb  cc              aa  bb  cc
com incr= 6      44, 58, 55,  1, 81, 24, 89, 63, 79
fica em ordem    aa  bb  cc  dd  aa  bb  cc  dd  aa
com incr=4       44, 24, 55,  1, 79, 58, 89, 63, 81
fica em ordem    aa  aa  aa  aa  aa  aa  aa  aa  aa
com incr=1        1, 24, 44, 55, 58, 63, 79, 81, 89
```

Seqüências Conhecidas

Original do Dr Shell:

$$h_t = \lfloor (N/2), h_k = \lfloor (h_{k+1}/2) ... h_1 = 1$$

(ruim) Exemplo: para $N = 30 \Longrightarrow 15, 7, 3, 1$, mas para $N = 32 \Longrightarrow 16, 8, 4, 2, 1$.

Seqüência de Hibbard

$$h_t = (2^k) - 1, ..., 7, 3, 1$$

Exemplo: para $N = 30 \Longrightarrow 1, 3, 7, 15, 31$, para qualquer N.

Seqüência de Knuth

$$h_i = \frac{1}{2}.[(3^i) + 1]$$

Exemplo: 1, 5, 14, 41...

Seqüência de Gonnet's

$$h_t = \lfloor (N/2.2),\ h_k = \lfloor (h_k + \frac{1}{2.2}),\ com\ h_1 = 1, h_2 = 2 =$$

Exemplo: $N = 30 \Longrightarrow 30/2.2 = 13.6,\ \lfloor 13 = 13, 13/2.2 = 5.9, \lfloor 5 = 5, 5/2.2 = 2.27 \rfloor 2 = 2, h_1 = 1$

Seqüência de Sedgewick: 1, 5, 19, 41, 109, fórmula completa em [Wei97]

Todas estas seqüências geram complexidades $O(n^{5/4})$ ou $O(n^{7/6})$ ou $O(n^{3/2})$ ou $O(n^{4/3})$... Para que o leitor possa entender estas complexidades, imagine-se um vetor de n=100.000 itens que devem ser ordenados. O valor base em $O(n)$ é portanto 100.000

n elevado a 7/6	681.292	(7/6=1,16)
n elevado a 5/4	1.778.279	(5/4=1,25)
n elevado a 4/3	4.641.588	(4/3=1,33)
n elevado a 3/2	31.622.776	(3/2=1,50)
n elevado a 2	10.000.000.000	

©88-08, Pedro Kantek

Regras para a seqüência do shelsort As regras para a seqüência são

1. Qualquer número de elementos
2. O último número tem que ser 1
3. cada número tem que ser maior do que o anterior. Preferencialmente, que estes números sejam primos entre si (melhora o desempenho)
4. Não tem sentido usar número maior que o tamanho do conjunto

Note que a seqüência simples {1} é aceitável (atende às 3 regras acima), mas olhando o algoritmo percebe-se que o desempenho é pífio, pois o algoritmo transforma-se em um quase INSERÇÃO DIRETA

	seq={1}	23.104 comparações
	seq={1,5}	6.269
Eis algumas comparações em um vetor de 300 aleatórios:	seq={1,5,19}	3.548
	seq={1,5,19,41}	3.272
	seq={1,5,19,41,109}	3.131

Algoritmo ShellSort Ordenação por shellsort

```
1: função SHELLSORT (inteiro V[1000])
2: inteiro k, h, i, x, j
3: DECR ← seqüência desejada
4: para k de 1 até tamanho(DECR) faça
5:    h ← DECR[k]
6:    i ← h
7:    enquanto (i < 1000) faça
8:       x ← V[i+1]
9:       j ← i
10:      enquanto (j ≥ h ∧ V[1+j-h] > x) faça
11:         CONTADOR++ {para saber o esforco (complexidade)}
12:         V[1+j] ← V[1+j-h]
13:         j ← j-h
14:      fimenquanto
15:      V[1+j] ← x
16:      i++
17:   fimenquanto
18: fimpara
19: devolva V
20: fim função
```

O shellsort foi o primeiro algoritmo a romper (para baixo) a barreira de $O(n^2)$. É muito simples e muito rápido.

(um bom local de pesquisa http://www.cs.princeton.edu/∼rs/shell/animate.html)

Método HEAPSORT (alg. 2) Considerando conhecida criação de um heap binário (criar, subir, descer, incluir, excluir) deve-se:

Ordenação usando heap

```
1: função HEAPSORT(inteiro V[1000])
2: Y ← criaheap binário (V)
3: para I de 1000 até 1 passo -1 faça
4:    V[I] ← I excluiheap Y {note que excluiheap recebe I para aproveitar o mesmo
      espaço}
5: fimpara
6: devolva V
7: fim função {usa-se V e Y para não embaralhar. Dá para ser 1 só}
```

Este método é muito bom, sendo apenas ligeiramente pior que o QUICKSORT por uma constante. Sua complexidade é $O(n.\log_2 n)$.

Método de Particionamento Parte da premissa da permutação, mas em vez de trocar vizinhos, troca elementos distantes entre si. Muito eficiente. O melhor exemplo é o QuickSort (C.A.R. Hoare, 1962)

Ordenação por particionamento - quicksort

```
1: Algoritmo QUICKSORT (inteiro V[1000])
2: função RAP(inteiro I, F)
3: inteiro ANT, DEP, MEI, AUX
4: ANT ← I
5: DEP ← F
6: MEI ← V[⌊ (I+F) ÷ 2⌋]
7: repita
8:    enquanto (V[ANT] < MEI) faça
9:       ANT++
10:   fimenquanto
11:   enquanto (V[DEP] > MEI) faça
12:      DEP- -
13:   fimenquanto
14:   se (ANT ≤ DEP) então
15:      TROCA (ANT,DEP)
16:      ANT++
17:      DEP–
18:   fimse
19: até (ANT > DEP)
20: se (I < DEP) então
21:   RAP (I, DEP)
22: fimse
23: se (ANT < F) então
24:   RAP (ANT, F)
25: fimse
26: fim função
27: RAP (1,1000)
28: devolva (V)
29: fim algoritmo
```

12.3.1 Comparações

A tabela a seguir descreve os tempos de processamento para ordenar um vetor Aleatório (valores diferentes aleatórios).

Tabela de Desempenhos Comparativos medidos em segundos

Tam	Bol	B*	Sha	Ins	InB	Sel	Heap	Quick
100	0.6	0.7	0.4	0.3	0.2	0.4	0.1	0.06
500	16	17	12	7	6	10	0.8	0.5
1000	68	70	51	31	25	41	2	1
2500	429	436	324	198	153	255	7	3

Já a próxima tabela contém os tempos de processamento para um vetor quasi-ordenado, ou seja um vetor que contém todos os seus elementos já em ordem, com

a exceção de um único deles, incluído no meio do vetor.

Tempos de processamento para um vetor quasi-ordenado

Tam	Bol	B*	Sha	Ins	InB	Sel	Heap	Quick
100	0.4	0.06	0.05	0.01	0.01	0.4	0.1	0.06
500	11	0.1	0.1	0.1	0.5	10	0.9	0.3
1000	47	0.3	0.2	0.2	1.1	40	2	0.8
2500	294	0.6	0.6	0.5	3.2	255	6	2

Medidas feitas em um Pentium 500 com W2000 usando APL*PLUS 6.4

12.3.2 Em resumo

Os métodos $O(n.\log_2 n)$ tem constantes de complexidade MAIORES do que os melhores métodos de $O(n^2)$. Assim, a regra parece ser a seguinte:
Para vetores $< 1000 \sim 10000 \Longrightarrow$ usar o método shellsort, porque o mesmo é simples de programar e muito eficiente.
Para vetores maiores do que o limite acima, usar QUICKSORT, ou opcionalmente HEAPSORT.
Se você não lembrar de nenhum algoritmo, use o método 1, o horrível.

12.4 Ordenação de seqüências (Sort Externo)

A restrição básica nesta classe de algoritmos é que não é possível ter todos os dados na memória ao mesmo tempo. Em um determinado instante só é possível ter acesso a um pequeno conjunto de dados. Por exemplo, imagine ter que ordenar um arquivo por RG (10.500.000 registros no estado do Paraná).

Vamos supor arquivos SEQÜENCIAIS (disquete, winchester, fita) arquivos $S_1, ...S_n$ (S de seqüenciais).

Suponha a existência de 4 arquivos seqüenciais ($S_1, S_2, S_3 e S_4$), sendo que os dados a classificar estão em S_1. Supondo que a memória consiga ordenar K registros, o primeiro passo é ler S_1 de B em B blocos, ordená-los internamente e gravá-los em S_3 e S_4 alternadamente.

Por exemplo, suponhamos a entrada, e seja B=3

```
S1= 44, 27, 78, 12, 81, 17, 23, 90, 46, 73, 29, 15, 20, fica:
S2= vazia
S3= 27, 44, 78   |   23, 46, 90   |   20
S4= 12, 17, 81   |   15, 29, 73
```

Agora, S_3 e S_4 vão ser intercalados, guardando-se o resultado em S_1 e S_2

```
S1=12, 17, 27, 44, 78, 81  |  20
S2=15, 23, 29, 46, 73, 90
```

S_1 e S_2 são intercalados guardando-se o resultado em S_3 e S_4

```
S3=12, 15, 17, 23, 27, 29, 44, 46, 73, 78, 81, 90
S4=20
```

Finalmente, S_3 e S_4 são intercalados

```
S2=12, 15, 17, 20, 23, 27, 29, 44, 46, 73, 78, 81, 90
```

 ©88-08, Pedro Kantek

Lembre que o conceito de intercalar é similar ao do balance line Grava o menor e volta a ler o arquivo que foi menor.

Se mais do que 2 arquivos seqüenciais podem ser usados, o algoritmo fica mais curto. Em vez do algoritmo 2-way merge, passa-se a ter o multi-way merge. Os dados serão gravados em k locais, e na hora de fazer a intercalação, todos os registros lidos terão que ser comparados na busca do menor. Usa-se para isso um HEAP. A escolha do menor é uma operação de EXCLUSAOHEAP.

Gravado o menor, o arquivo que o continha é lido e este novo elemento sofre um INCLUSAOHEAP.

A complexidade deste algoritmo EM PASSADAS é $O(\lceil \log_k N/B)$, onde
k = quantidade de fitas de entrada
N = quantidade de registros na entrada
B = quantidade de registros em cada bloco de classificação.

EXERCÍCIO RESOLVIDO 27 Quantas passadas serão necessárias para classificar 10.500.000 usando-se 8 fitas (4 de entrada), com bloco de 2.000 registros ?

Resposta: $O(\lceil \log_4 10.500.000/2.000) = \lceil \log_4 5250 = \rceil 6.17 = 7$

[Wei97] = Data Structures and Algorithm Analysis in C, Mark Allen Weiss

Desafio

- Implementar o algoritmo QUICK sort na forma de um utilitário de uso genérico
- Inventar uma seqüência nova para o algoritmo do Dr Shell e batizá-la com um nome criativo. Comparar o desempenho desta nova seqüência com os já conhecidos.

©88-08, Pedro Kantek

Capítulo 13

Tabelas de dispersão

13.1 Tabelas de dispersão e Hashing

Esta técnica foi descrita pela primeira vez por Arnold Dumey em 1956, mas o nome HASH só apareceu em 1968. O método também é conhecido com os nomes de aleatorização, randomização e dispersão, além do consagrado *hashing*.

Em casos muito raros, o espaço de valores de chave coincide com o espaço de índices do vetor. No exemplo considerado, isto aconteceria se as chaves dos alunos fossem o seu número de chamada.

Mais usualmente, o espaço de chaves é muito maior do que o espaço de índices. São exemplos deste espaço muito maior: número de carteira de identidade, número de matrícula, nome, CPF, etc.

Por conta desta diferença de dimensões, há que se usar uma função de transformação que receba a chave em questão e devolva um número de índice para aquela chave.

Por exemplo, a função receberia o código de matrícula 940309871 e devolveria 27, sugerindo que o lugar adequado para este aluno seria a 27. posição do vetor.

13.1.1 Espaço de chaves e de índices

Para entender melhor este assunto, precisa-se definir o que seja o espaço de chaves e o de índices. Denomina-se $S(k)$ = espaço de chaves, ao conjunto de todos os valores possíveis que a chave K possa assumir. Por exemplo, se uma chave é a placa de um carro, qual o seu espaço de chaves ? Resposta = são 3 letras e 4 números, ou seja 26 x 26 x 26 x 10 x 10 x 10 x 10 = 175.000.000

$S(i)$=Espaço de índices é o conjunto de todos os valores que o índice possa assumir. Neste caso, S(i) coincide com o tamanho do vetor.

Por conta do usa da função de hash, esta técnica também é conhecida como "transformação de chaves". Como o universo de chaves é muito maior do que o de índices, claramente a função mapeia diversas chaves em um mesmo índice.

Por exemplo, se tivéssemos um vetor para guardar 1000 nomes, contendo cada um 30 caracteres, a nossa função de mapeamento levaria 26^{30} em 10^3.

O processo é realizado em algumas etapas:

1. Dada uma chave K, a função F(K) deve apontar um índice j.
2. Com este índice j, deve-se verificar se a posição j do vetor corresponde a K está livre ou em caso negativo, deve-se prever um processo que gere um j alternativo.

A função F(k) é conhecida como função de HASH e deve ter 2 características:

- ser facilmente computável
- minimizar a ocorrência de colisões

A etapa 2 acima, deve ser considerada, levando em conta a possibilidade de 2 chaves gerarem o mesmo índice. Neste caso, a segunda chave deve poder gerar um índice alternativo.

Dá-se a esta ocorrência o nome de "colisão".

13.1.2 Colisão

Uma característica importante da função de mapeamento é que ela gere uma distribuição tão uniforme quanto possível no espaço de índices. Esta característica minimiza a ocorrência de colisões.

Supondo um vetor de tamanho W, contendo elementos cuja chave é K, uma função boa candidata a espalhar adequadamente é K mod (o primeiro número primo menor que W).

Por exemplo, no caso de termos 500 alunos identificados por seus códigos de matrícula (W), uma função de mapeamento poderia ser W mod 497 + 1

Quando um elemento a ser incluído no vetor gerar um índice j qualquer e o elemento j do vetor já estiver ocupado, está-se diante de uma colisão. O segundo indivíduo deve ser colocado em um outro local, sem que se perca seu acesso a ele.

Conceitualmente falando, a ocorrência de colisões é bastante frequente. Fazer a experiência das datas de nascimento (acima de 23 pessoas é jogo). Isto significa que se mapearmos mais de 23 chaves em uma tabela com 365 entradas usando dia+mes como função de hashing, a P(colisão) é maior do que 0,5 (na verdade 0,5073). Este paradoxo foi descoberto em 1939 por R Mises.

Uma das primeiras propostas de tratar colisão foi: Se a chave 9765, já encontrar o slot ocupado, tentar o 976, depois o 97 e depois o 9, sendo todos estes slots SECUNDÁRIOS. (cerca de 1954).

Olhando graficamente para um problema:

Suponhamos uma aplicação de um vetor com 12 posições e as chaves numeradas entre 1 e 100. A função de transformação vai ser j = 1+K mod 11.

Eis a tabela

1	2	3	4	5	6	7	8	9	10	11	12

Suponhamos que deve ser incluído o elemento de chave 67. Calculando j = 67 mod 11 + 1 = 2.

E portanto, usa-se o índice 2, e o vetor fica:

	67										
1	2	3	4	5	6	7	8	9	10	11	12

Agora vem o número 93. Calculando j = 93 mod 11 + 1 = 6, e fica

	67				93						
1	2	3	4	5	6	7	8	9	10	11	12

Supondo a chegada do número 4. Tem-se j = 4 mod 11 + 1 = 5, e fica

	67			4	93						
1	2	3	4	5	6	7	8	9	10	11	12

Agora ocorre a colisão: Suponha-se a chegada do número 37. Calculando-se j = 37 mod 11 + 1 = 6. Percebe-se que o elemento 6 já está preenchido pelo 93, e obviamente 37 <> 93.

O ponto agora é onde colocar o 37, sem que se perca a estruturação da tabela ?

Sempre que ocorre uma colisão, uma segunda tentativa tem que ser feita. De imediato podem-se usar 2 estratégias. A primeira faz uso da própria área da tabela (ou do vetor) para guardar colisões, usando – por exemplo – o próximo espaço disponível.

Neste caso o 37, seria colocado no próximo espaço vago:

©88-08, Pedro Kantek

Tabela 13.1: Fator de carga versus colisões em tabela hash

Fator de Carga	Média de Colisões
10	1.04100000
20	1.12750000
30	1.20200000
40	1.32950000
50	1.46420000
60	1.67483333
70	2.04785714
80	2.74200000
90	3.90144444
92	4.29565217
95	5.02431579
97	5.59762887

```
      67                  4     93    37
 1    2     3     4       5     6     7     8     9     10    11    12
```

Neste caso, opta-se por testar todas as posições subsequentes a um determinado local. A busca se encerra até que o elemento seja encontrado (sucesso), ou uma posição em branco seja encontrada (insucesso). O mesmo vale se for encontrado o fim da tabela. Este método é chamado de endereçamento aberto.

Eis uma aplicação prática de um endereçamento aberto. Criou-se uma tabela de HASH com espaco para 100 elementos. A função de hash é: i = 1 + k mod 97. O valer da chave foi obtido por um gerador randômico (situação IDEAL). A seguir uma tabela que descreve para os diversos fatores de carga qual a quantidade média de acessos na inclusão. Cada fator de carga foi testado em 100 rodadas. O valor apresentado é a média.

Define-se o fator de carga como FC=numero de elementos incluidos/tamanho total da estrutura.

Aqui, um acesso=1 significa que não houve nenhuma colisão. Um valor de 1.10 significa que em 10 inclusões houve 11 acessos (1 foi colião).

```
 1: inteiro função INCHASHABE(inteiro V[1000], inteiro K)
 2: inteiro I
 3: I ← (K mod 997) + 1
 4: se V[I] = 0 então
 5:    V[I] ← K
 6:    devolva I
 7: senão
 8:    enquanto V[I] ≠ 0 ∧ I ≤ 1000 faça
 9:       I++
10:    fimenquanto
11:    se I ≤ 1000 então
12:       V[I] ← K
13:       devolva I
14:    senão
15:       devolva -1
16:    fimse
17: fimse
18: fim função
```

 ©88-08, Pedro Kantek


```
 1: inteiro função ACHAHASHABE (inteiro V[1000], inteiro K)
 2: inteiro I
 3: I ← (K mod 997) + 1
 4: se V[I] = K então
 5:    devolva I
 6: senão
 7:    enquanto V[I] ≠ K ∧ V[I] ≠ 0 faça
 8:       I++
 9:       se I = 1000 então
10:          devolva -1
11:          abandone
12:       fimse
13:    fimenquanto
14:    se V[I] = K então
15:       devolva I
16:    senão
17:       devolva -1
18:    fimse
19: fimse
20: fimfunção
```

EXERCÍCIO 225 Escreva a função que exclui uma chave K no vetor de HASH V.

EXERCÍCIO 226 Escreva o algoritmo de uma função que receba V e K e devolva o número de testes necessários para descobrir que K existe em V ou não.

Uma segunda estratégia é abrir uma outra área de armazenamento, ou outra tabela, especificamente para guardar os itens que sofreram colisão. Quando isto for feito, dar-se-á o nome de tabela primária a que armazena os elementos que não sofreram colisão, enquanto a outra recebe o nome de tabela secundária.

No exemplo, ficaria

```
(área primária)
      67                  4     93
 1    2     3     4       5     6     7     8     9     10    11    12

(área secundária)
37
1         2         3         4
```

Atente-se que, em geral a área secundária é menor do que a primária.

Tanto em um como noutro caso, há que se guardar a informação da ocorrência da colisão. Novamente, várias abordagens podem ser empregadas.

A mais óbvia é fazer uma ligação entre a área primária e a secundária. Neste caso, na primária, e após a colisão é colocado o endereço da secundária que contém o elemento colidido.

No exemplo, ficaria

```
(área primária)
      67                  4     93(1)
 1    2     3     4       5     6     7     8     9     10    11    12

(área secundária)
37
1         2         3         4
```

Outra alternativa é usar a área secundária seqüencialmente, o que aumenta a complexidade da pesquisa, diminuindo a probabilidade de ocorrência de colisão.

Outra ainda é usar uma área primária maior. Por exemplo, se há 12 chaves usar uma área primária de 24 ou 36 elementos.

©88-08, Pedro Kantek

Um problema ocorre quando se deseja deletar um elemento na tabela primária. Como excluí-lo sem perder o encadeamento ? É necessário colocar lá um flag que diga isso.

Outro problema ocorre quando uma pesquisa falha. Aqui, sabe-se apenas que o registro procurado não está. Nada se sabe sobre as distâncias para os vizinhos maior e menor (isso pode ser importante, por exemplo em interpolação). Igualmente, para um processamento sequencial de chaves.

Finalmente, para usar HASH há que ter fé nas probabilidades, pois se o caso médio é bom, o pior caso pode ser terrível.

Falar sobre Fator de Carga... (ver pag. 219 do CLR)

EXERCÍCIO 227 Simule uma área primária de 9 elementos e uma secundária de 5, para a inclusão das seguintes chaves: 8, 12, 23, 3, 4, 49, 50. Faça o esquema com ligação da primária e da secundária e também com esquema de endereçamento aberto.

EXERCÍCIO 228 Idem, para área primária de 15 e secundária de 10. Chaves: 1, 5, 11, 16, 21, 22, 23, 12, 13, 90, 12, 13, 14, 88. Faça o esquema com ligação da primária e da secundária e também com esquema de endereçamento aberto.

EXERCÍCIO 229 Descreva em palavras o processo de busca de um determinado elemento em uma tabela de dispersão, tanto no caso de ligação, quanto no caso do

Desafio

- Implementar um programa de computador que implemente uma tabela hash usando o critério 1 da avaliação 521a.
- Implementar um programa de computador que implemente uma tabela hash usando o critério 2 da avaliação 521a.

13.2 Busca em tabelas

A questão fundamental em informática é: Dado um atributo de um dado qualquer, preciso localizar os demais atributos desse mesmo dado. Por exemplo:

- dada uma placa, recuperar o nome e endereço do proprietário
- Dado um número de matrícula, recuperar as notas e faltas de um aluno
- Dado um código de cidade, recuperar a cidade e o estado
- Dado um número de voo, descobrir companhia, origem, destino, etc.

Tudo em Informática, são variações sobre este mesmo tema.

13.2.1 Conceitos

Uma tabela é uma série ordenada de itens (ou registros) sendo formado – cada um – por um conjunto de atributos ou campos. Normalmente, cada um destes itens armazenada dados referentes a um único elemento de dados. Todos os itens da tabela têm o mesmo formato. Cada um dos itens é conhecido como uma "entrada"da tabela. Veja na tabela **??** uma explicação destes conceitos.

 ©88-08, Pedro Kantek

	CÓDIGO ALUNO	ANO EN-TRADA	NOME	CURSO
1	1657	1999	Maria Silva	veterinária
2	1708	1999	Antônio Pereira	culinária
3	1954	2000	José Brito	engenharia civil
4	2109	2000	Beatriz Costa	medicina
5	2210	2001	Bernardo Ari	psicologia
6	2761	2002	Wilson Souza	decoração
7	2763	2002	Edison Sá	história
8	2857	2002	Mário Murilo	contábeis
9	2900	2003	José Costa	pedagogia

Aqui, cada uma das linhas da tabela corresponde a um item de dados (neste caso a um aluno). Cada coluna da tabela corresponde a uma lista de atributos de todos os itens (no exemplo, a primeira coluna é o código individual de cada um dos alunos).

Um dos atributos da tabela (muitas vezes também a união de mais de um atributo, contíguos ou não) é designado para ser a chave da mesma. É a "porta de entrada" para acessar os demais dados de cada item. Quando este valor é único, ou seja, não há dois itens com o mesmo valor de chave, diz-se que a chave é primária. É costume, para efeitos de otimização dos algoritmos, que a tabela esteja ordenada por uma chave primária. No exemplo acima, a chave primária para a tabela de alunos é o **CÓDIGO ALUNO**. Ela é primária, já que não existem 2 alunos com o mesmo código. Note também que a tabela está ordenada em ordem crescente por código de aluno.

Nesse sentido, pesquisar uma tabela, é receber um valor de chave e devolver a localização do primeiro (ou único) item da tabela que tem aquela chave, ou devolver uma informação pré-combinada de que não existe item que tenha a chave pesquisada.

Assim, no exemplo acima, ao pesquisar a chave <2857>, o algoritmo pesquisador devolverá a informação **8**, que por sua vez permite o acesso ao conjunto de dados <2857,2002,Mário Murilo,contábeis>. Por outro lado, ao pesquisar a chave <2200>, a resposta poderia ser (por exemplo) **-1**. Ora, como não existe endereço negativo, este valor poderia ser interpretado como não existe item que tenha chave 2200.

Quando a chave não identifica um único item (isto é ela se repete em mais de um item), ela é conhecida como chave secundária. Um exemplo disto pode ser visto na conta de telefone que mensalmente recebemos. Lá constam todas as ligações efetuadas no mês, ordenadas por dia e hora. Assim a chave **número de telefone que originou a chamada** na tabela que contém todas as ligações efetuadas por todos os telefones da região é secundária. Para que ela possa se tornar primária, teria que ser concatenada (juntada) com a informação dia + hora. O conjunto formado por telefone+dia+hora pode ser caracterizado como chave primária.

13.2.2 Operações

Consulta Esta em geral é a mais freqüente operação que sofre uma tabela. Esta é a razão pela qual vale a pena construir a tabela de maneira a que a consulta seja a mais otimizada possível. Uma consulta é receber uma chave e devolver a localização do item pedido ou a informação de não existência.

©88-08, Pedro Kantek

Inclusão Algumas tabelas são construídas uma única vez e muito raramente (ou nuca) sofrem atualização. Um exemplo é a tabela de estados da federação. Outras tabelas sofrem permanente manutenção. A inclusão é a operação que agrega um novo item à tabela. Ela recebe um conjunto completo de dados, localiza uma posição na tabela, inclui o item e devolve um indicativo de sucesso. Pode também devolver um insucesso, e isto pode ter diversas razões, por exemplo:

- falta de espaço na tabela (por exemplo, para tabelas alocadas seqüencialmente quando estas esgotam seu espaço)
- a chave primária fornecida já existe na tabela. Neste caso a inclusão duplicaria a chave o que vimos ser proibido para chaves primárias.
- algum dado eventualmente definido como obrigatório não foi fornecido.
- algum dado fornecido não foi aprovado na consistência, se esta houver (por exemplo, a data de admissão poderia ser 31/02/2002)

Exclusão Esta é a operação inversa à operação da inclusão. Ela consiste em receber uma chave e excluir o item completo da tabela, liberando o espaço anteriormente ocupado por ele, e eventualmente reorganizando a tabela. Ao final devolve um indicativo de sucesso ou de insucesso (o elemento informado não estava na tabela, por exemplo).

Alteração de linha Esta operação visa substituir um ou vários conteúdos de um item. É usual tratar de maneira distinta as duas possibilidades de alteração: de chaves e de não chaves. Muitas vezes os algoritmos não permitem alterar chaves. Neste caso o caminho para corrigir uma imperfeição na tabela seria excluir o item e reincluí-lo.

Alteração de coluna Esta operação refere-se a alterar uma coluna da tabela (ao invés de uma linha, o que seria o caso anterior). Por exemplo, em uma tabela de itens de venda, esta alteração poderia ser substituir todos os preços de venda por um novo valor, talvez preço-antigo $\times$ 1.10, significando um aumento linear nos preços de 10%.

Criação Para tabelas criadas dinamicamente, esta operação prevê a alocação de espaço e inicialização de seus conteúdos, bem como dos descritores da tabela.

Destruição Para tabelas criadas dinamicamente, esta operação é o inverso da criação e prevê a liberação das áreas ocupadas pela tabela e pelos descritores.

Como se viu acima, muitas tabelas apresentam-se ordenadas por uma chave. A maneira como essa ordenação ocorre pode apresentar variações. A maneira mais usual é a chamada *ordenação física*. Aqui, o endereço físico dos itens segue um critério qualquer de ordenação da chave. Na tabela **??** esta maneira foi seguida. Note que cada linha (aluno) tem código de aluno maior do que a linha (aluno) anterior. Formalmente falando, para uma tabela ordenada fisicamente, por exemplo em ordem crescente de chave, teremos: se $chave[i] > chave[j]$ então obrigatoriamente teremos $i > j, \forall\, i, j$.

Outra maneira de manter uma tabela ordenada, é usar encadeamentos. Lembrando, para isto, é necessário que cada item tenha um atributo adicional (denominado *próximo*) e que a tabela tenha um descritor adicional (denominado *início*). Veja no exemplo, na tabela **??**.

início
2

 ©88-08, Pedro Kantek

	ALU	**AN**	**NOME**	**CURSO**	próx
1	1657	1999	Maria Silva	veterinária	8
2	1708	1999	Antônio Pereira	culinária	4
3	1954	2000	José Brito	engenharia civil	9
4	2109	2000	Beatriz Costa	medicina	5
5	2210	2001	Bernardo Ari	psicologia	7
6	2761	2002	Wilson Souza	decoração	-1
7	2763	2002	Edison Sá	história	3
8	2857	2002	Mario Murilo	contábeis	6
9	2900	2003	José Costa	pedagogia	1

Para entender como a tabela **??** está ordenada deve-se seguir o encadeamento começando em *início*. Este nos leva ao elemento 2 da tabela (Antônio). Depois, o próximo é o elemento 4 (Beatriz). Seguindo o encadeamento, vai-se a 5 (Bernardo), depois a 7 (Edison), depois a 3 (José Brito), 9 (José Costa), 1 (Maria), 8 (Mario), 6 (Wilson). Note que Wilson é o último elemento do encadeamento, logo não tem próximo. Isto é representado por um *terminador* no atributo *próximo*. Neste caso usou-se -1.

Este exemplo acabou de mostrar uma tabela cuja ordem física é por código de aluno e que tem um encadeamento para ordenar a mesma tabela por ordem alfabética de nome de aluno.

Vale a referência de uma dada tabela só pode estar em ordem física por um único aributo, mas pode estar em ordem por quantos atributos quiser desde que estas ordens sejam providas por encadeamentos.

> Uma encarnação disto que se fala está nas listas telefônicas: Usualmente temos 2 em casa: uma ordenada por assinante e outra lista ordenada por endereço. Seria uma grande economia de papel se se pudesse ter um único volume com ambas as ordenações mas isto é impossível quando se fala em ordenação física que é a usada ao se construir uma lista telefônica.

Vejamos na tabela **??** um exemplo da mesma tabela já vista, com 2 ordenações por encadeamentos:

início
nome
2

início
curso
8

©88-08, Pedro Kantek

	AL	AN	NOME	CURSO	próx nome	próx curso
1	1657	1999	Maria Silva	veterinária	8	-1
2	1708	1999	Antônio Pereira	culinária	4	6
3	1954	2000	José Brito	engenharia civil	9	7
4	2109	2000	Beatriz Costa	medicina	5	9
5	2210	2001	Bernardo Ari	psicologia	7	1
6	2761	2002	Wilson Souza	decoração	-1	3
7	2763	2002	Edison Sá	história	3	4
8	2857	2002	Mario Murilo	contábeis	6	2
9	2900	2003	José Costa	pedagogia	1	5

Outra possibilidade de ordenação, é utilizando um vetor de cursores (VC). Esta abordagem é usada quando se pretende uma ordenação física sem que haja necessidade de mover fisicamente os dados que compõe cada item. O uso do VC também permite que se desconsidere a regra segundo a qual só é possível uma única ordenação física. Usando VC podem-se ter quantas ordenações se quiserem. Acompanhe na tabela **??** o mesmo exemplo ainda, sendo ordenado por VCs.

	AL	AN	NOME	CURSO		VC nome		VC curso
1	1657	1999	Maria Silva	veterinária	1	2	1	8
2	1708	1999	Antônio Pereira	culinária	2	4	2	2
3	1954	2000	José Brito	engenharia civil	3	5	3	6
4	2109	2000	Beatriz Costa	medicina	4	7	4	3
5	2210	2001	Bernardo Ari	psicologia	5	3	5	7
6	2761	2002	Wilson Souza	decoração	6	9	6	4
7	2763	2002	Edison Sá	história	7	1	7	9
8	2857	2002	Mario Murilo	contábeis	8	8	8	5
9	2900	2003	José Costa	pedagogia	9	6	9	1

Aqui, se se quiser processar a tabela por nome de aluno, deve-se seguir a ordenação indicada pelo *VC de curso*, que indica ser o primeiro item aquele residente no endereço 2 da tabela. O segundo é o 4, depois o 5, 7, 3 e assim por diante. A mesma consideração pode ser feita para processar a tabela por curso.

Apenas a título de recordação das complexidades associadas a cada método de

pesquisa a ser estudado a seguir, eis um resumo e uma proposta de capítulo que vem a seguir.

Um vetor V dado um índice i, recupera-se o elemento V[i], ao custo de $O(1)$. Se esta busca for pela chave K, binária, e o vetor estiver fisicamente ordenado pela chave K, o custo é $\log_2 n$.

Uma lista qualquer de comprimento n dado um atributo A, recupera-se o elemento ao qual A pertence, ao custo de $O(n)$.

Uma árvore binária de pesquisa de n elementos dado uma chave K, recupera-se o nodo ao qual K pertence, ao custo de $O(\log_2 n)$.

Uma árvore B de grau t contendo n elementos dada uma chave... custo de $O(\log_t n)$.

Um arquivo seqüencial (fita) contendo n registros dado um atributo A... a um custo de $O(n)$ [eventualmente 1/2].

13.3 Busca Linear

Dado um vetor V composto de registros (identificados por K), escreva o algoritmo que devolve i que localiza K, ou -1 se K não estiver em V.

Exemplo Vetor de dias uteis em setembro de 2001: V=3,4,5,6,10,11,12,13,14,17, 18, 19,20,21,24,25,26,27,28. Dado um certo dia, por exemplo 17, qual é o seu índice ? Resposta: 10.

```
1: inteiro função ACHAI (inteiro V[1000], inteiro KEY) {ULTIMO contém o último
   elemento usado}
2: inteiro I, RESP
3: I ← 1
4: RESP ← -1
5: enquanto (I≤ ULTIMO) faça
6:    se V[I] = KEY então
7:       RESP ← I
8:       I ← 1001
9:    fimse
10:   I++
11: fimenquanto
12: devolva RESP
13: fim função
```

Note-se que esta função faz 2 testes a cada iteração (I $\leq$ ULTIMO e V[I] $\neq$ KEY), será que dá para melhorar isso ?

Sim, usando-se o conceito de SENTINELA. Define-se sentinela como o elemento buscado, que é colocado ao final do local de busca.

```
1: inteiro função ACHACOMSENT (inteiro V[1000], inteiro KEY) {ULTIMO contém
   o último elemento usado}
2: inteiro I, RESP
3: V[ULTIMO+1] ← KEY {note que ULTIMO não tem seu valor alterado}
4: I ← 1
5: enquanto (V[I] ≠ KEY) faça
6:    I++
7: fimenquanto
8: se I = ULTIMO+1 então
```

```
 9:    devolva -1
10: senão
11:    devolva I
12: fimse
13: fim função
```

Se o vetor estiver ordenado, a busca pode ser acelerada, abandonando a busca quando um maior (menor) for encontrado. Outra possibilidade é usar um salto maior (digamos 50, ao invés de 1). Ao localizar o primeiro maior, volta-se 1 passo (50) e daí usa-se o passo unitário.

Ordenar ou não ? Em caso de grandes arquivos seqüenciais, esta não é uma decisão trivial. Ela depende do número de vezes que as consultas vão ser feitas. Se um número grande de consultas está previsto, vale o esforço de ordenar (e manter ordenado) o arquivo. Se por outro lado as pesquisas são muito poucas, talvez seja melhor varrer seqüencialmente o arquivo quando a pesquisa chegar.

> Por exemplo, qual o assinante do telefone número 342-7675, sem ligar para lá ? Apenas se uma única consulta precisar ser feita, é melhor processar sequencialmente do que ordenar.

EXERCÍCIO 230 Escreva o algoritmo de uma função que receba um vetor V[1000] DESORDENADO de inteiros e uma chave K e devolva qual o índice do elemento de V que mais se aproxima de K (V se encerra no valor de ULTIMO). Se houver empate na aproximação para mais e para menos, responda qualquer uma. Exemplo: Seja o vetor 7, 80, 65, 40, 34, 76, 77, 78 e K = 33. A resposta deverá ser 5.

EXERCÍCIO 231 Escreva o mesmo algoritmo para um vetor ORDENADO, com desempenho superior ao exercício anterior

A complexidade associada a este método, considerando que todas as chaves pesquisadas se encontram no vetor e que todas tem igual probabilidade de serem pesquisadas é de $O(\frac{n+1}{2})$ para um vetor de n elementos. Esta fórmula foi obtida na hipótese de cada uma das chaves do vetor ser consultada uma vez. Então, teremos 1 acesso para a primeira chave, 2 para a segunda, 3 para a terceira, ..., n para a n-ésima. Para localizar todas as chaves, serão $1+2+3+...+n$ acessos. O número de acessos médio é então

$$O(\frac{1+2+3+...+n}{n} = \frac{n \times \frac{n+1}{2}}{n} = \frac{n+1}{2})$$

.

Uma hipótese que pode melhorar este desempenho é ordenar a tabela pela chave que se está buscando. Neste caso, a pesquisa positiva (a que encontra a chave pesquisada) não é melhorada, mas a negativa (quando a chave não está) fica mais rápida, uma vez que para concluir que a chave não está presente não é mais necessário ir até o fim da tabela. Pode-se parar ao encontrar uma chave maior do que aquela que se busca.

Outra maneia de melhorar o desempenho aqui, é ordenar a tabela pla freqüencia com que cada item é pesquisado. De fato, em aplicações reais é comum existirem itens que são muito mais pesquisados do que outros. Os economistas chama isto de princípio de Pareto. Se forem descobertos quais os itens mais pesquisados, eles podem ser deslocados para o início da tabela, melhorando significativamente o desempenho das consultas.

> Vejamos um exemplo de tabela de estados das regiões sul e sudeste do Brasil. Pela regra de Pareto, a probabilidade de acesso poderia ser: PR 35%, SC 20%, SP 14%, RJ 11%, MG 8%, RS 6%, MS 4% e ES 2%.

> Se a tabela estivesse ordenada em ordem alfabética de estado, o número médio de acessos seria

estado	% de pesquisa
ES	0.02
MG	0.08
MS	0.04
PR	0.35
RJ	0.11
RS	0.06
SC	0.20
SP	0.14

> $O(1 \times 0.02 + 2 \times 0.08 + 3 \times 0.04 + 4 \times 0.35 + 5 \times 0.11 + 6 \times 0.06 + 7 \times 0.20 + 8 \times 0.14) = 5.13$ Já se a tabela for ordenada por freqüência de consulta (ordem descendente)

estado	% de pesquisa
PR	0.35
SC	0.20
SP	0.14
RJ	0.11
MG	0.08
RS	0.06
MS	0.04
ES	0.02

> o número médio de acessos será: $O(1 \times 0.35 + 2 \times 0.20 + 3 \times 0.14 + 4 \times 0.11 + 5 \times 0.08 + 6 \times 0.06 + 7 \times 0.04 + 8 \times 0.02) = 2.81$
>
> Como observação, se a tabela for invertida (o pior caso), a complexidade média chega a 6.19 acessos.

A dificuldade está em conhecer previamente como se distribuirão as pesquisas. Para remediar este fato, existem algumas estratégias, por exemplo:

- A cada vez que um item é pesquisado ele é levado ao topo da tabela. Esta estratégia só pode ser pensada para tabelas encadeadas, sob pena de piorar ao invés de melhorar o desempenho.
- A cada vez que um item é pesquisado ele sobe uma (ou mais) posições na tabela.
- Acrescentar um contador de acessos a cada item. De tempos em tempos, ordenar a tabela segundo esse contador.

EXERCÍCIO 232 Escreva o algoritmo de uma função de pesquisa em tabela encadeada, na qual após cada acesso o elemento acessado sobe uma posição.

EXERCÍCIO 233 Escreva o algoritmo de uma função de pesquisa em tabela encadeada, na qual após cada acesso o elemento acessado vai para o início da tabela.

EXERCÍCIO 234 Escreva o algoritmo de uma função de pesquisa em tabela seqüencial, na qual após cada 1000 acessos a tabela é reordenada segundo o número de acessos a cada elemento até então registrado.

13.4 Pesquisa Binária

Mencionada pela primeira vez por John Mauchly em 1946. Provavelmente este foi o primeiro algoritmo não numérico conhecido.

Para poder usar esta pesquisa a tabela PRECISA estar ordenada. Se não estiver os resultados da técnica são imprevisíveis.

Em resumo é o seguinte:

- A tabela é dividida ao meio (daí o nome binária)
- A chave é comparada com o elemento que dividiu a tabela. Se este for menor, a primeira parte da tabela é abandonada. Se for maior abandona-se a segunda parte.
- A pesquisa prossegue pela divisão ao meio da parte que ficou.
- Rapidamente as sucessivas divisões ao meio esgotam a tabela.
- O processo termina quando a chave for encontrada, ou quando a parte que restou da divisão for nula.

Vejamos no algoritmo abaixo

```
1: inteiro função PESQBIN( inteiro V[1000], inteiro KEY)
2: inteiro INIC,METADE,FIM
3: INIC ← 1 {campo delimita o limite inferior da pesquisa}
4: FIM ← 1000 {delimita o limite superior da pesquisa}
5: repita
6:    METADE ← [| (INIC + FIM)]/2
7:    se KEY ≤ V[METADE] então
8:       FIM ← METADE - 1
9:    senão
10:      INIC ← METADE + 1
11:   fimse
12: até V[METADE] = KEY ∨ INIC > FIM
13: se V[METADE] = KEY  então
14:   devolva METADE
15: senão
16:   devolva -1
17: fimse
18: fim função
```

Teste de mesa: Seja o vetor: 1 7 12 20 40 41 47 49 60 88, buscando 77.

1. primeira iteração: INIC = 1, FIM = 10, METADE = 5. Como V[5] é 40, e 77 não é ≤ 40, temos: INIC ← METADE + 1 ou INIC ← 6.
2. segunda iteração: INIC = 6, FIM = 10, METADE = 8. Como V[8] é 49, e 77 não é ≤ 77, temos: INIC ← METADE + 1 ou INIC ← 9.
3. terceira iteração: INIC = 9, FIM = 10, METADE = 9. Como V[9] é 60 e 77 não é ≤ 77, temos: INIC ← METADE + 1 ou INIC ← 10.
4. quarta iteração: INIC = 10, FIM = 10, METADE = 10. Como V[10] é 88 e 88 > 77, temos FIM ← METADE - 1 ou FIM ← 9.

Neste ponto o ciclo se encerra, pois INÍCIO > FIM. Finalmente descobre-se que a chave não está presente pois, V [10] é 88 que é diferente de 77.

Note-se que o número médio de acessos em uma pesquisa binária é da ordem de $\log_2 N$, onde N é o número de elementos da tabela.

Por exemplo: Se o tamanho da tabela for de 1000 elementos, na primeira divisão, desprezamos 500 e ficamos apenas com 500. Na segunda com 250, na terceira com 125, na quarta com 63, na quinta com 33, 17, 9, 5, 3, 2, 1. No total foram 11 comparações.

Duas observações importantes

A para que a pesquisa binária possa ser feita, é necessário que o vetor ESTEJA ordenado

B Embora este algoritmo seja por natureza recursivo, por questões de desempenho muitas vezes ele é programado como iterativo.

Seja um exemplo que usa a tabela **??**. Supondo 2 tentativas, sendo a primeira bem sucedida e a segunda mal sucedida, e supondo que o elemento buscado será encontrado após 50das pesquisas (para o caso da busca seqüencial) compara-se a seguir o desempenho das 2 técnicas para vetores de diversos tamanhos

Tamanho do vetor	Pesquisa Binária sucesso	Pesquisa Binária insucesso	Pesquisa Sequencial sucesso	Sequencial insucesso
10	4 - δ	4	5	10
100	7 - δ	7	50	100
1.000	10 - δ	10	500	1.000
10.000	14 - δ	14	5.000	10.000
1.000.000	20 - δ	20	500.000	1.000.000
100.000.000	27 - δ	27	50.000.000	100.000.000
n	$\log_2 n - \delta$	$\log_2 n$	$n/2$	n

Bibliografia: Tennenbaum, ED usando C, página 501. Pode-se fazer a pergunta: Porque δ ?

EXERCÍCIO RESOLVIDO 28 Determine qual o número médio de acessos em uma pesquisa binária para uma tabela de:

- 5 elementos:
- 10 elementos:
- 20 elementos:
- 50 elementos:
- 100 elementos:
- 1200 elementos:
- 15000 elementos:

Respostas: 2,3 3,3 4,3 5,6 6,6 10,2 e 13,3

EXERCÍCIO 235 Dado o vetor: 1 4 5 6 8 10 13 18 21 60, realizar o chinês para descobrir a presença ou não das chaves, através do algoritmo de pesquisa binária:

- 8

- 9
- 50
- 200

EXERCÍCIO 236 Escreva o algoritmo de uma função que faz busca binária em uma tabela que está ordenadada usando a técnica de VC

Desafio

- Implementar em um programa os 6 métodos acima estudados. Realizar 10.000 inclusões em cada um. Anotar e comparar os tempos obtidos em cada um.

 ©88-08, Pedro Kantek