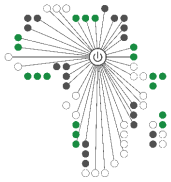


Universidade Virtual Africana

INFORMÁTICA APLICADA: ITI 2300

ESTRUTURA DE DADOS E ALGORITMOS

Arlete Maria Ferrao

Prefácio

A Universidade Virtual Africana (AVU) orgulha-se de participar do aumento do acesso à educação nos países africanos através da produção de materiais de aprendizagem de qualidade. Também estamos orgulhosos de contribuir com o conhecimento global, pois nossos Recursos Educacionais Abertos são acessados principalmente de fora do continente africano.

Este módulo foi desenvolvido como parte de um diploma e programa de graduação em Ciências da Computação Aplicada, em colaboração com 18 instituições parceiras africanas de 16 países. Um total de 156 módulos foram desenvolvidos ou traduzidos para garantir disponibilidade em inglês, francês e português. Esses módulos também foram disponibilizados como recursos de educação aberta (OER) em oer.avu.org.

Em nome da Universidade Virtual Africana e nosso patrono, nossas instituições parceiras, o Banco Africano de Desenvolvimento, convido você a usar este módulo em sua instituição, para sua própria educação, compartilhá-lo o mais amplamente possível e participar ativamente da AVU Comunidades de prática de seu interesse. Estamos empenhados em estar na linha de frente do desenvolvimento e compartilhamento de recursos educacionais abertos.

A Universidade Virtual Africana (UVA) é uma Organização Pan-Africana Intergovernamental criada por carta com o mandato de aumentar significativamente o acesso a educação e treinamento superior de qualidade através do uso inovador de tecnologias de comunicação de informação. Uma Carta, que estabelece a UVA como Organização Intergovernamental, foi assinada até agora por dezenove (19) Governos Africanos - Quênia, Senegal, Mauritânia, Mali, Costa do Marfim, Tanzânia, Moçambique, República Democrática do Congo, Benin, Gana, República da Guiné, Burkina Faso, Níger, Sudão do Sul, Sudão, Gâmbia, Guiné-Bissau, Etiópia e Cabo Verde.

As seguintes instituições participaram do Programa de Informática Aplicada: (1) Université d'Abomey Calavi em Benin; (2) Université de Ougagadougou em Burkina Faso; (3) Université Lumière de Bujumbura no Burundi; (4) Universidade de Douala nos Camarões; (5) Universidade de Nouakchott na Mauritânia; (6) Université Gaston Berger no Senegal; (7) Universidade das Ciências, Técnicas e Tecnologias de Bamako no Mali (8) Instituto de Administração e Administração Pública do Gana; (9) Universidade de Ciência e Tecnologia Kwame Nkrumah em Gana; (10) Universidade Kenyatta no Quênia; (11) Universidade Egerton no Quênia; (12) Universidade de Addis Abeba na Etiópia (13) Universidade do Ruanda; (14) Universidade de Dar es Salaam na Tanzânia; (15) Universite Abdou Moumouni de Niamey no Níger; (16) Université Cheikh Anta Diop no Senegal; (17) Universidade Pedagógica em Moçambique; E (18) A Universidade da Gâmbia na Gâmbia.

Bakary Diallo

O Reitor

Universidade Virtual Africana

Créditos de Produção

Autor

Arlete Maria Ferrao

Par revisor(a)

Célio Sengo

UVA - Coordenação Académica

Dr. Marilena Cabral

Coordenador Geral Programa de Informática Aplicada

Prof Tim Mwololo Waema

Coordenador do módulo

Jules Degila

Designers Instrucionais

Elizabeth Mbasu

Benta Ochola

Diana Tuel

Equipa Multimédia

Sidney McGregor

Barry Savala

Edwin Kiprono

Kelvin Muriithi

Victor Oluoch Otieno

Michal Abigael Koyier

Mercy Tabi Ojwang

Josiah Mutsogu

Kefa Murimi

Gerisson Mulongo

Direitos de Autor

Apoiado por

Projeto Multinacional II da UVA financiado pelo Banco Africano de Desenvolvimento..

Tabela de conteúdo

Descrição Geral do Curso

Bem-vindo(a) a Estrutura de Dados e Algoritmos

O facto de ter conhecimentos de linguagens de programação não é condição suficiente para que um profissional se sinta capacitado a desenvolver sistemas é necessário, saber usar os recursos de programação de forma adequada. O desenvolvimento de um programa, envolve diversas etapas incluindo a identificação das propriedades dos dados e suas características funcionais. Para que um programa possa atender eficientemente as funcionalidades para as quais foi desenhado, é necessário ter conhecimentos das técnicas para organizar de maneira estruturada os dados a serem manipulados. Sendo assim, para além de uma linguagem de programação, é necessário conhecer as principais técnicas de estruturação de dados. O curso em epígrafe introduz aos estudantes à estrutura de dados e algoritmos e a forma como elas podem ser criadas e utilizadas. Aborda aspectos relacionados com a organização de dados, e certas estruturas de dados serão discutidas, bem como as operações a elas aplicadas. Serão igualmente discutidas as noções de algoritmo e sua complexidade. (Falbo, 2012)

Pré-requisitos

- Introdução à Informática Aplicada
- Principios de Programação

Materiais

Os materiais necessários para completar este curso incluem:

1. Adam Drozdek , Data Structures and Algorithms in Java, 2ª edição.
2. Barnett, Granville; Del Tongo, Luca. Data Structures and Algorithms: Annotated Reference with Examples 1st Edition, 2008.
3. Bjarne Stroustrup, Programming: Principles and Practice Using C++, ISBN 0321992784, Publisher: Addison Wesley, 2014
4. Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
5. Edelweiss, Nina; Galante, Renata; Estrutura de Dados, Porto Alegre: Bookman, 2009. ISBN 978-85-7780-381-1
6. Harry H Chaudhary, Practical Data Structures Using C .: Beginner s Easy Edition 2014. ISBN 1500136972; Publisher: Createspace, United States, 2014
7. Malik, D.S., Data Structures Using C++, Second Edition, , 2010
8. Nivio Ziviani, Projecto de Algoritmos com Implementação em Java e C++.. 2006.

Editora Thomson, ISBN 8522105251

9. Robert Sedgewick, ALGORITHMS IN C : FUNDAMENTALS, DATA STRUCTURES, SORTING, SEARCHING, PARTS 1-4, ISBN 8131713059; Publisher: Pearson Education/AW Professional, 2012

10. Robert Sedgewick, Algorithms in C: Parts 1-5: Fundamentals Data Structures, Sorting, Searching, and Graph Algorithms, ISBN 8178082497; Publisher: Addison Wesley, 2001

11. Rocha, A. M. A, Estruturas de Dados e Algoritmos em C, 2008 FCA Editora Informática. Coleção:Tecnologias de Informação

12. SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999

13. SEYMOUR LIPSCHUTZ, G. A. VIJAYALAKSHMI PAI. Data Structures. Tata McGraw-Hill Publishing Company Limited. New Delhi, 2006.

14. Tenenbaum, A. M.;. Langsam, Y. ; Moshe J. A. Estruturas de dados usando C São Paulo : MAKRON Books, 1995.

15. Szwarcfiter, Jaime Luiz; Markenzo n, Lilian; Estruturas de Dados e Seus Algoritmos 3ª edição, Rio de Janeiro 2012.

16. V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.

17. Vic Broquard, Beginning Data Structures in C++, ISBN 1941415547, Publisher: Broquard eBooks, United States, 2014

18. Walmar, Celes; Cerqueira, Renato; Rangel, J. L.; Introdução a Estrutura de Dados, Rio de Janeiro, 2004. Elsevier.

Recursos da Internet

http://people.cs.vt.edu/~shaffer/Book/C++3e20130328.pdf

http://dotnetslackers.com

www.inf.ufes.br/~falbo

- Ligações para recursos da web
- Apontamentos de palestras
- Computadores e ligação à Internet

Outros recursos incluem sebentas elaboradas pelo docente da cadeira, materiais em power point, não discorando também que a internet pode ser também uma fonte de consulta e leitura.

Objetivos do Curso

Após concluir este curso, o(a) aluno(a) deve ser capaz de :

- Entender a importância dos algoritmos na resolução de problemas e determinar quais as estruturas de dados que são mais eficazes para vários cenários e problemas
- Análisar o tempo de execução de vários algoritmos associados com estruturas de dados
- Escolher e aplicar a estrutura de dados apropriada para a modelação de um determinado problema
- Familiarizar-se com a hierarquia de memória e árvores B [B-trees].
- Compreender os conceitos fundamentais associados aos vários tipos de estruturas de dados sequênciais (listas ligadas, pilhas e filas) e não sequênciais (árvores, tabelas de dispersão), e dos algoritmos passíveis de aplicação a cada estrutura de dados.
- Implementar as operações sobre estruturas de dados (criar, modificar, apagar e combinar)

Unidades

- Unidade 0: Introdução à Computação e Programação
- Unidade 1: Resolução de Problemas Algoritmicos
- Unidade 2: Recursão
- Unidade 3: Estruturas de Dados Básicas e Tipo de Dados Abstractos
- Unidade 4: Algoritmos de Busca e de Ordenação

Avaliação

Em cada unidade encontram-se incluídos instrumentos de avaliação formativa a fim de verificar o progresso do(a)s aluno(a)s.

No final de cada módulo são apresentados instrumentos de avaliação sumativa, tais como testes e trabalhos finais, que compreende os conhecimentos e as competências estudadas no módulo.

A implementação dos instrumentos de avaliação sumativa fica ao critério da instituição que oferece o curso. A estratégia de avaliação sugerida é a seguinte:

1	Avaliação do tempo de estudo	30%
2	Avaliação do trabalho independente	30%
3	Exame semestral	40%

Calendarização

Unidade	Temas e Atividades	Estimativa do tempo
0	Diagnóstico 1. Introdução á Informática Aplicada 2. Princípios de Programação	6 horas
1	Resolução de Problemas Algoritmicos 1. Introdução aos algoritmos 2. Características de algoritmos 3. Formas de representação de algoritmos 4. Complexidade de algoritmos	30 horas
2	Recursão 1. Noção de Recursão 2. Aplicações de recursão	9 horas
3	Estrutura de dados básicos e tipos de dados abstratos 1. Arrays 2. Listas 3. Listas ligadas 4. Pilhas 5. Filas 6. Tabelas de dispersão 7. Árvores	60 horas
4	Algoritmos de pesquisa e ordenação	15 horas

Leituras e outros Recursos

As leituras e outros recursos deste curso são:

- https://www.caelum.com.br/apostila-java-estrutura-dados
- www.inf.ufpr.br/cursos/ci055/apostila.pdf
- www.uspleste.usp.br/digiampietri/ACH2023/ACH2023.pdf
- https://www.youtube.com/watch?v=e8tdtnDmUpE

Unidade 0

Leituras e outros recursos obrigatórios:

- Bjarne Stroustrup, Programming Principles and Practice Using C and C++, Addison Wesley, 2014, ISBN0321992784
- Introduction to Computer Science, Pearson Education
- Da Rocha, António Adrego, Estruturas de Dados e Algoritmos em C - 3ª Ed. FCA Editora. Revista e Aumentada 2014

Unidade 1

Leituras e outros recursos obrigatórios:

- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999
- Nivio Ziviani, Projecto de Algoritmos com Implementação em Java e C++.. 2006. Editora Thomson, ISBN 8522105251

Leituras e outros recursos opcionais:

- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Rocha, A. M. A, Estruturas de Dados e Algoritmos em C, 2008 FCA Editora Informática. Coleção:Tecnologias de Informação
- SEYMOUR LIPSCHUTZ, G. A. VIJAYALAKSHMI PAI. Data Structures. Tata McGraw-Hill Publishing Company Limited. New Delhi, 2006.

Unidade 2

Leituras e outros recursos obrigatórios:

- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999

- Nivio Ziviani, Projecto de Algoritmos com Implementação em Java em C++. 2006. Editora Thomson, ISBN 8522105251

Leituras e outros recursos opcionais:

- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Adam Drozdek, Data Structures and Algorithms in Java, 2ª edição.

Unidade 3

Leituras e outros recursos obrigatórios:

- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999
- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Data Structures Using C++, Second Edition, D.S. Malik, 2010

Leituras e outros recursos opcionais:

- MARK ALLEN WEISS. Data Structures and Algorithms Analysis (2nd Ed)1994
- Robert Sedgewick, Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms", 3rd Edition, 2001

Unit 4

Leituras e outros recursos obrigatórios:

- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- Robert Sedgewick, Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms", Parts 1-4, 2012.
- Adam Drozdek, Data Structures and Algorithms in Java, 2ª edição.

Leituras e outros recursos opcionais:

- Robert Sedgewick, Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms", 3rd Edition, Addison Wesley, 2001.
- ROCHA, A. Estrutura de dados e algoritmos em C, FCA Lisboa 2008
- Carvalho A, Exercicios de Java; Algoritmia e Programação Estruturada, FCA, Lisboa 2013

Unidade 0. Diagnóstico

Introdução à Unidade

O propósito desta unidade é verificar a compreensão dos conhecimentos que possui relacionados com este curso. Na presente unidade, você deverá demonstrar as habilidades adquiridas nas disciplinas de Princípios de Programação e na Introdução à Informática Aplicada, que são os pré-requisitos necessários para frequentar esta disciplina, recapitulando os conteúdos essenciais para esse fim.

Objectivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Demonstrar conhecimentos essenciais sobre dados, dados de entrada , dados de saída, processamento, armazenamento.
- Traduzir a solução de um problema em fluxograma e/ou pseudo-código.
- Identificar estratégias necessárias para a resolução de problemas em estruturas de dados e algoritmos.

Termos-chave

Dados: Em informática designa-se por dados os elementos de partida que servem de base para o tratamento e sobre os quais o computador efectua as operações necessárias à tarefa em questão. Os dados são uma representação dos factos, conceitos ou instruções de uma maneira normalizada que se adapte à comunicação, interpretação e processamento pelo ser humano ou através de máquinas automáticas. Eles são representados por símbolos como, por exemplo, as letras do alfabeto : a, b, c , etc, mas não são em si a informação desejada.

Dados de entrada: é usado para descrever o processo de captura ou colecção de dados brutos no início de um sistema de informação baseado em computador.

Dados de saída: é a obtenção de resultados sob a forma de informação significativa para as pessoas a quem se destina.

Processamento: O termo processamento de dados (ou de informações) engloba qualquer trabalho de manipulação de dados que tenha como finalidade obter resultados previamente estabelecidos, ou seja, consiste numa série de actividades ordenadamente realizadas, com o objectivo de produzir um arranjo determinado de informações a partir de outras obtidas inicialmente.

Esse grupo de atividades envolve a transmissão, o armazenamento, a recuperação, a comparação e a combinação de informações.

Dispositivos de Armazenamento: é qualquer hardware de computador que é usado para o armazenamento, e extração de ficheiros de dados e objectos. Ele pode conter e armazenar informação tanto temporária como permanentemente, e pode ser interna ou externa a um computador, servidor ou qualquer dispositivo de computação semelhantes.

Sistema Operativo: é um programa que actua como intermediário entre o usuário e o hardware de um computador. O sistema operativo tem como propósito fornecer um ambiente no qual o usuário possa executar seus programas, actuando como um conjunto de ferramentas necessárias para que um computador possa ser utilizado de forma adequada.

Cada sistema operativo pode ter uma linguagem de máquina própria e distinta, por isso é comum que softwares feitos para um sistema operativo não funcionem em outro.

Constante é um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um programa. Conforme o seu tipo, a constante é classificada como sendo numérica, lógica e literal.

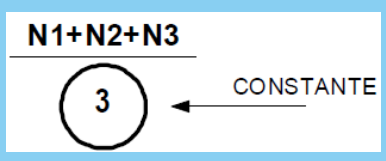


Figura 1: Reprsentação gráfica de uma constante

Variáveis são os elementos básicos que um programa manipula, são espaços reservados na memória do computador para armazenar um tipo de dado determinado. Elas devem receber um nome para que possam ser referenciadas e modificadas sempre que for for necessário pelo programa. Sendo assim, um programa deve conter declarações que especificam qual é o tipo de variáveis que este utilizará.

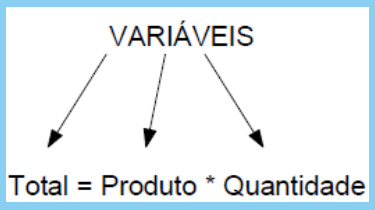


Figura 2: Representação gráfica de uma variável

Exemplo: analogia de uma caixa e uma variável

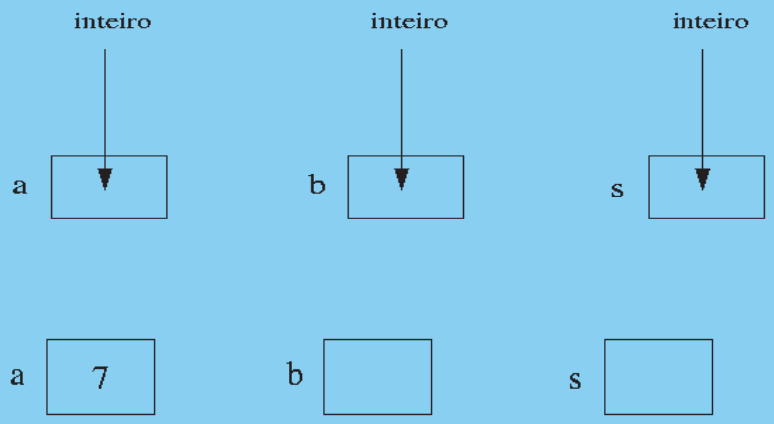


Figura 3: Exemplo Ilustrativo de uma variável

Expressões: são combinações de variáveis, constantes e operadores. Quando se escreve expressões tem de se levar em consideração a ordem com que os operadores são executados, conforme a tabela de precedências de uma determina linguagem de programação. São elas,

1. Expressões aritméticas

Expressões aritméticas são aquelas que apresentam como resultado um valor numérico que pode ser um número inteiro ou real, dependendo dos operandos e operadores.

2. Expressões lógicas

Expressões lógicas são aquelas cujo resultado pode somente assumir os valores verdadeiro ou falso.

3. Expressões literais

Operadores: são meios pelo qual incrementa-se, decrementa-se, compara-se e avalia-se dados dentro do computador.

Existem quatro tipos de operadores:

- Operadores Aritméticos
- Operadores Relacionais
- Operadores Lógicos
- Operadores bit-a-bit

Função: é um bloco de código de programa que pode ser usado diversas vezes em sua execução. A utilização de funções, permite com que o programa fique mais legível, mais bem estruturado. Um programa em C por exemplo, consiste basicamente por várias funções colocadas juntas.

Estruturas de Repetição: Utilizadas quando se deseja executar um determinado conjunto de instruções por um número definido ou indefinido de vezes, enquanto um determinado estado prevalecer ou até que seja alcançado um determinado estado. São eles; For, While, Do ...While.

Estruturas de Decisão: é um bloco de código que utiliza as expressões condicionais para tomar decisões que orientam qual código deve ser executado. São elas; Decisão simples (if), Decisão composta (if… else) , Decisão múltipla (Case) e Operador condicional ternário (?).

Avaliação da Unidade

Verifique a sua compreensão!

Avaliação Sistemática 1: Teste Diagnóstico

Instruções

As seguintes questões estão relacionadas com o que aprendeu nas disciplinas de princípios de programação e introdução à informática aplicada. Responda as seguintes questões com clareza, dando exemplos sempre que for necessário.

Critérios de Avaliação

Pergunta	Pontuação (máximo 20 valores)
1-a)	3
1-b)	3
2-a)	2
2-b)	2
2-c)	2
3	2.5
4	2.5
5	3
Total	20

Avaliação

1. Explique os seguintes conceitos:
   a. Computador
   b. Paradígmas de programação
2. Descreva a arquitectura do sistema computacional quanto a :
   a. Memória
   b. conjunto de instruções
   c. comunicação entre várias unidades do sistema computacional
3. Enuncie as técnicas de descrição de algoritmos e explique pelo menos uma delas.
4. Explique como é feita a gestão dinâmica da memória.
5. Elabore um quadro comparativo dos comandos de repetição.

Respostas:

1. Explique o seguintes conceitos:

   a). Computador: É um equipamento constituido por componentes mecânicos e electrónicos capaz de receber, armazenar e enviar dados e de efectuar sobre estes sequências previamente programadas de operações aritméticas (como cálculos) e lógicas (como comparações) com o objectivo de resolver problemas.

   b). Paradigmas de programação: É o conjunto de características que determinam o ponto de vista da realidade e como se actua sobre ela, os quais são classificados quanto ao seu conceito de base. Cada qual determina uma forma particular de abordar os problemas e de formular respectivas soluções.

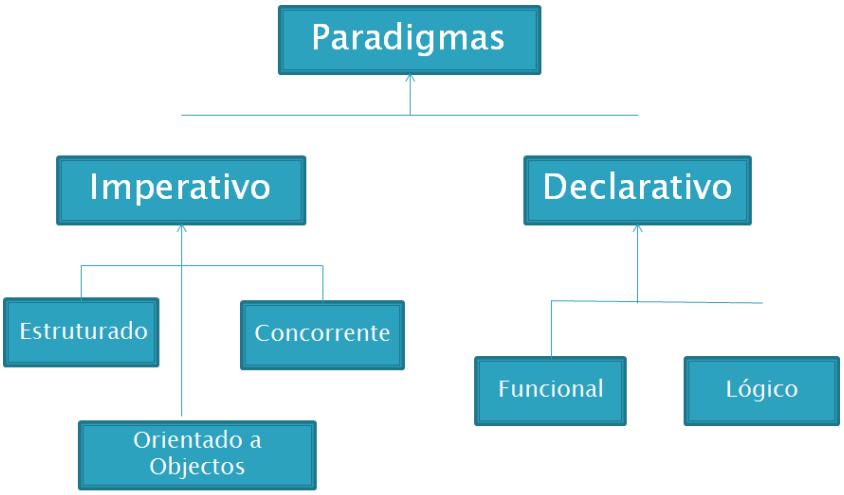


Figura 4: Paradigmas de programação

2. 2. Descreva a arquitectura do sistema computacional quanto a :

   a. Memória

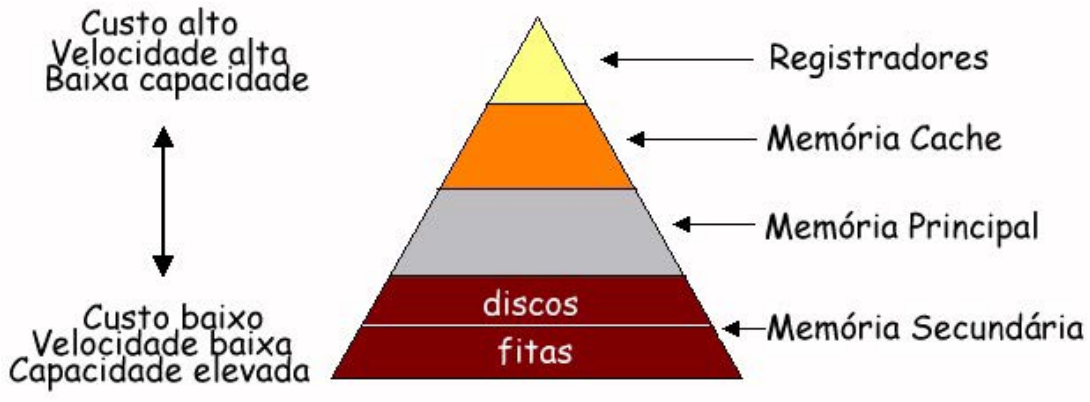


Figura 5: Modelo de barramento de sistema.Fonte: http://www.di.ufpb.br acesso em 10/03/2016

É no interior da CPU que os programas são executados e geralmente não cabem integralmente na memória interna do chip (registradores +Cache interna). Então, num determinado instante, os dados podem ser copiados entre dois níveis adjacentes da hierarquia da memória.

b) Quanto á comunicação, ela é estabelecida entre várias unidades do sistema computacional, através de uma via compartilhada que se denomina barramento de sistema (bus). Este barramento é constituído do barramento de dados, do barramento de endereços e do barramento de controle. Existe também um barramento de energia e algumas arquiteturas podem ter um barramento de I/O separado.

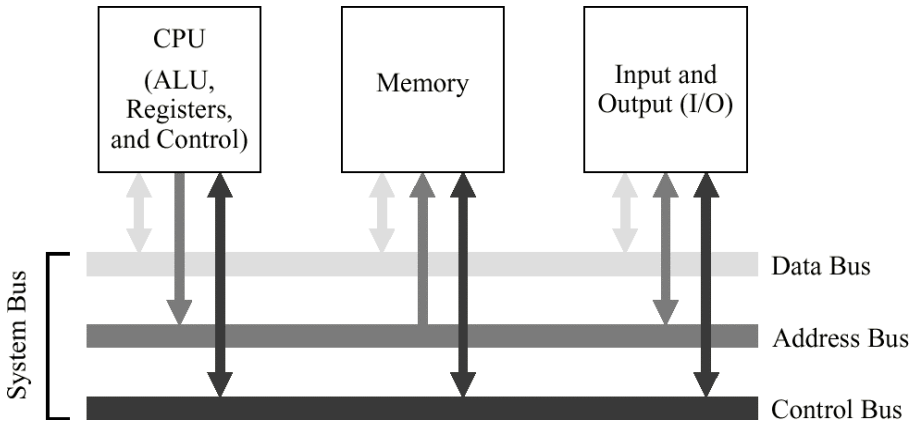


Figura 6: Modelo de barramento de sistema. Fonte: (Murdocca & Heuring, 2001)

3. Enuncie as técnicas de descrição de algoritmos e explique pelo menos uma delas.

- Descrição narrativa: Consiste em analisar o enunciado do problema e escrever, utilizando uma linguagem natural (no nosso caso, a língua portuguesa), os passos a serem seguidos para sua resolução
- Fluxograma: é uma forma gráfica para a expressão do fluxo de execução de um programa. Um problema pode ser analisado e escrever-se com uso de símbolos gráficos, e com passos a serem seguidos para sua resolução. O entendimento de elementos gráficos é mais simples que o entendimento de textos.
- Pseudo-codigo: é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem o escreve, de forma a ser entendida por qualquer pessoa) sem necessidade de conhecer a sintaxe de nenhuma linguagem de programação.

4. Explique como é feita a gestão dinâmica da memória.

A gestão dinâmica da memória é feita através das funções *malloc* e *calloc* para a atribuição da memória, e para libertá-la utiliza-se a função *free*.

5. Elabore um quadro comparativo dos comandos de repetição.

	Utilidade	Forma

Comando while	Permite que um certo trecho de programa seja executado ENQUANTO uma certa condição for verdadeira.	while (condição)
		{
		// comandos a serem repetidos
		// comandos a serem repetidos
		}
		// comandos após o "while"

Comando do - while	Permite que um certo trecho de programa seja executado ENQUANTO uma certa condição for verdadeira	do
		{
		// comandos a serem repetidos
		// comandos a serem repetidos
		} while (condição);
		// comandos após o "do-while"

Comando for	Permite que um certo trecho de programa seja executado um determinado número de vezes.	for (comandos de inicialização;condição de teste;incremento/decremento)
		{
		// comandos a serem repetidos
		// comandos a serem repetidos
		}
		// comandos após o 'for'

Tabela 1: Quadro comparativo de comandos de repetição

Leituras e Outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de "Leituras e Outros Recursos do curso".

- http://www.inf.ufrgs.br acesso em 10/03/2016
- Lorenzi, F. de Mattos, P. N.; de Carvalho, T. P. Estrutura de Dados, Thomson Learning, 2006.
- http://www.di.ufpb.br acesso em 10/03/2016
- Murdoca, M.; Heuring, V. Introdução à Arquitectura de Computadores. Editora Campus, 2001.

Unidade 1. Resolução de Problemas Algorítmicos

Introdução à Unidade

Esta unidade faz-se uma introdução sobre às estruturas de dados e algoritmos. Na unidade discute-se sobre as diferentes estruturas de dados e os seus algoritmos que podem ajudar a implementar diferentes estruturas de dados no computador. A aplicação das diferentes estruturas de dados é apresentada com exemplos de algoritmos e que não estão confinadas a uma determinada linguagem de programação de computador.

Objectivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Calcular a complexidade de algoritmos polinomiais ou exponenciais, iterativos ou recursivos.
- Representar algoritmos em forma de pseudo-códigos e/ ou fluxogramas
- Entender a importância dos algoritmos na resolução de problemas e determinar quais as estruturas de dados que são mais eficazes para vários cenários e problemas.
- Avaliar um algoritmo para verificar se ele é eficiente ou não
- caracterizar um algoritmo

Termos-chave

Estrutura de Dados: é uma organização ou disposição de dados de forma coerente e estruturada para o processamento.

Algoritmico: é uma seqüência finita e bem definida de passos que, quando executados, realizam uma tarefa específica ou resolve um problema.

Complexidade de Algoritmos: A Complexidade de um Algoritmo consiste na quantidade de "trabalho" necessária para a sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados.

Pseudo-código: Escrita abreviada baseada numa "Linguagem Natural". Cada passo é expresso de forma clara e concisa. Permite expressar as ideias sem se preocupar com os detalhes sintácticos da linguagem de programação.

Fluxograma: Descrições gráficas, que permitem visualizar a estrutura do algoritmo. Existem vários símbolos, cada um representa um tipo de instrução.

Actividades de Aprendizagem

Actividade 1.1 - Introdução a resolução de problemas e algoritmos

Introdução

As actividades de aprendizagem desta secção, incluem o desenho de algoritmos, representação de solução de um problema usando fluxogramas e pseudo-códigos e ainda calcular a complexidade de algoritmos.

Detalhes da actividade

Um algoritmo é um conjunto finito de regras não ambíguas que fornece uma sequência de operações para resolver um problema específico. É algo como uma receita, ou uma rotina.

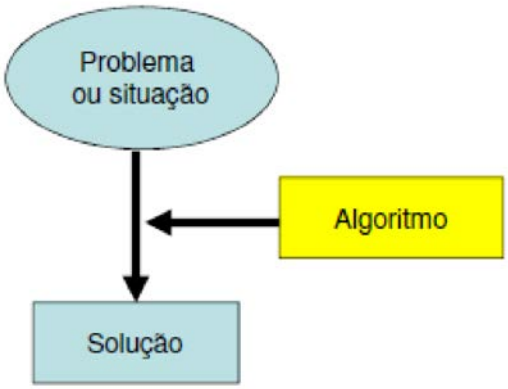


Figura 7: ilustração de algoritmo

O problema pode ser solucionado de várias maneiras, o importante é chegar à uma solução. Melhor ainda, se essa solução for atingida de forma eficiente e eficaz.

Os algoritmos fazem parte do dia-a-dia das pessoas.

Exemplos de algoritmos:

- Um algoritmo para trocar o pneu de um carro

1. Retirar o macaco e o estepe na bagageira do carro
2. Colocar o macaco de baixo do carro
3. Levantar o carro usando o macaco
4. Retirar o pneu furado
5. Colocar o estepe de volta
6. Abaixar o carro
7. Guardar o macaco e o pneu furado

- instruções para o uso de medicamentos
- indicações de como instalar um electrodoméstico ou uma receita de culinária.

Vários algoritmos podem resolver o mesmo problema, isto é, poderão existir várias formas para resolver o mesmo problema. A única diferença entre os algoritmos escolhidos, refere-se ao tempo necessário para executar a tarefa e o espaço de memória necessário. Portanto, é preciso que ao escolher o algoritmo, tenha em conta a sua eficiência e eficácia.

Componentes de um algoritmo:

Estado inicial – quais os dados de entrada do problema.

Estado final – o que se pretende obter.

Transformação – os passos necessários para transformar o estado inicial no final.

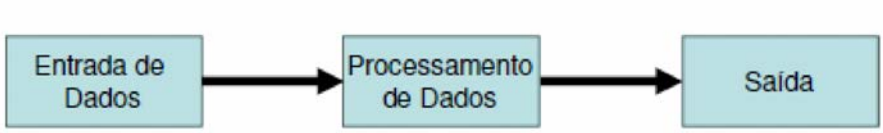


Figura 8: Componentes de um algoritmo

Método para a construção de algorítmos

Definir o processamento, ou seja, quais cálculos que serão efectuados. O processamento é responsável pela transformação dos dados de entrada em dados de saída.

- Definir os dados de saída
- Construir o algoritmo
- Testar o algoritmo realizando simulações

Técnicas de representação de algoritmos:

- Descrição narrativa
- Fluxograma
- Pseudo-codigo ou linguagem estruturada

Descrição Narrativa - Consiste em analisar o enunciado do problema e escrever, utilizando uma linguagem natural (no nosso caso, a língua portuguesa), os passos a serem seguidos para sua resolução. No entanto, o uso da linguagem natural pode provocar alguns problemas de interpretação do funcionamento do algoritmo se este tiver um grau de complexidade relativamente elevado. O grande número de linhas utilizado para descrever a solução de um problema, se apresentado de forma linear, pode representar um grande obstáculo ao entendimento do problema.

Vantagens: Não é necessário aprender nenhum conceito novo, pois é a linguagem natural.

Desvantagens:

- a línguagem natural pode ser interpretada de diferentes maneiras
- para a linguagem de programação, a linguagem natural é abstracta, imprecisa e pouco confiável

Exemplo:

Fluxograma: é uma forma gráfica para a expressão do fluxo de execução de um programa. Um problema pode ser analisado e escrever-se com uso de símbolos gráficos, e com passos a serem seguidos para sua resolução. O entendimento de elementos gráficos é mais simples que o entendimento de textos.

O uso de símbolos especiais e a combinação destes símbolos para formar as estruturas mais clássicas de controlo, como aquelas apresentadas anteriormente podem eliminar a ambiguidade eventualmente provocada pelo uso do texto escrito. Há muitos anos, o fluxograma tem aparecido como uma ferramenta interessante de representação do comportamento de programas, permitindo expressar, além do fluxo lógico da execução e, as operações envolvidas no processamento dos dados e as entradas e saídas. Os fluxogramas são construídos a partir do uso de símbolos padronizados que expressam classes de operações comumente utilizadas nos programas.

Símbolos utilizados na representação de um algoritmo por meio de um fluxograma:

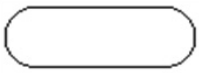
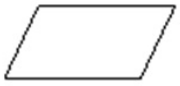
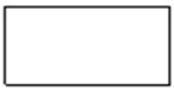
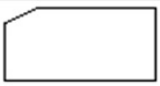
	Início ou fim do algoritmo
	Indica o sentido do fluxo de execução do algoritmo. Conecta os objetos gráficos
	Representa a entrada de dados
	Indica cálculos e atribuições de valores (processamento)
	Indica desvios ou tomadas de decisões (Por exemplo: SE isso, ENTÃO aquilo)
	Representa a saída de dados, no Portugol IDE
	Também representa a saída de dados

Figura 9: simbologia de fluxogramas

Exemplo de cálculo de média de um aluno usando fluxogramas

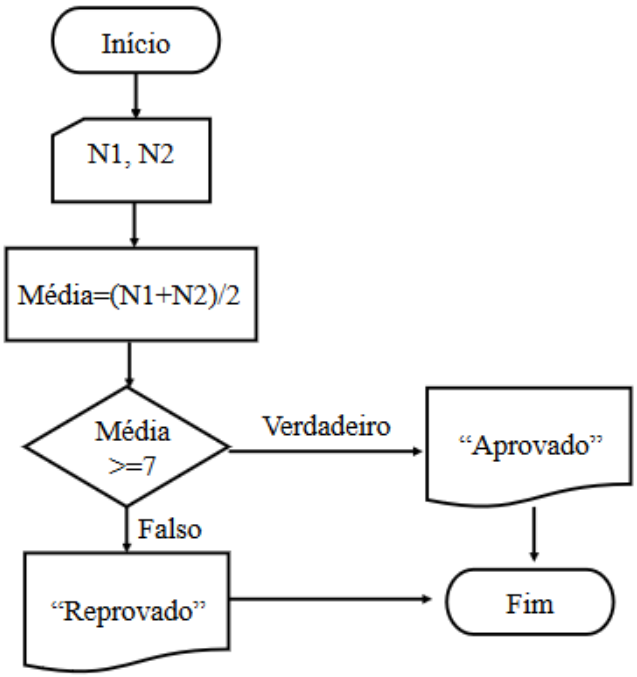


Figura 10: cálculo de média de 4 números, fonte: Autor

Vantagens de utilização de fluxogramas:

- É mais fácil perceber um conteúdo descrito de forma gráfica de que conteúdos descritos em forma textual;
- Quanto a simbologia, uma vez que obedecem a um padrão mundial, evita a ambiguidade.

Desvantagens:

- Os dados podem não ser suficientemente detalhados, dificultando, assim, a transcrição do algoritmo para o programa a ser desenvolvido;
- é necessário aprender a simbologia dos fluxogramas
- para algoritmos mais extensos, a construção do fluxograma pode se tornar mais complicada.

Pseudo-código: é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem o escreve, de forma a ser entendida por qualquer pessoa) sem necessidade de conhecer a sintaxe de nenhuma linguagem de programação.

Vantagens: A passagem do algoritmo para qualquer linguagem de programação équase imediata, bastando conhecer as palavras reservadas dessa linguagem queserão utilizadas.

Desvantagens:

- necessidade de aprender as regras dessa forma de representação

- a não padronização de sua estruturação
- poderá encontrar um mesmo termo descrito de formas diferentes em diferentes literaturas

Exemplo de cálculo da média

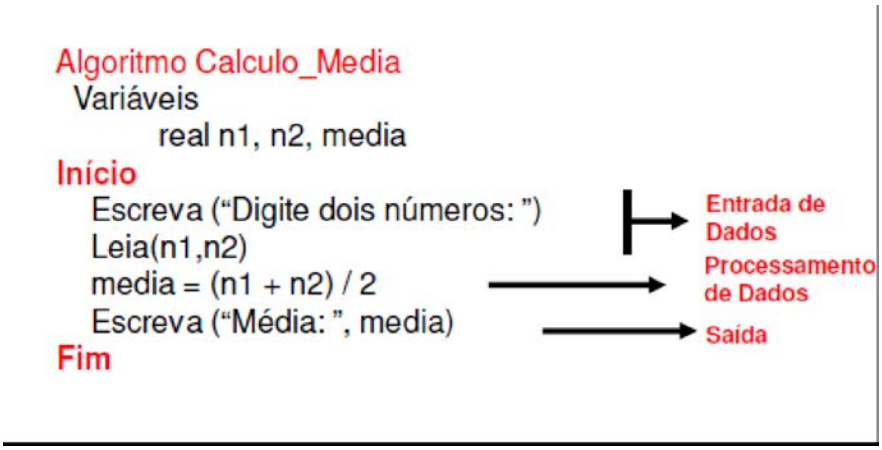


Figura 11:Pseudocódigo de cálculo de média

Caractererísticas de um algorítmo:

Finitude: Um algoritmo deve sempre terminar após um número finito de passos.

Definição: Cada passo de um algoritmo deve ser precisamente definido. As ações devem ser definidas rigorosamente e sem ambiguidades.

Entradas: Um algoritmo deve ter zero ou mais entradas, isto é quantidades que lhe são fornecidas antes do algoritmo iniciar.

Saídas: Um algoritmo deve ter pelo menos uma saída, isto é quantidades que tem uma relação específica com as entradas.

Efectividade: Um algoritmo deve ser efectivo. Isto significa que todas as operações devem ser suficientemente básicas de modo que possam ser em princípio executadas com precisão num tempo finito por um humano usando papel e lápis.

Para manipular adequadamente os dados do problema num algoritmo, tem que identifica-los correctamente dentro deste. Para isso,deve-se atribuir nomes para eles, estes nomes são chamados de identificadores.

As seguintes regras são válidas para a formação de identificadores:

- Devem começar por um caracter alfabético;
- Podem ser seguidos por mais caracteres alfabéticos e/ou numéricos;
- Não é permitido o uso de caracteres especiais, como: @, #, &, *, +, ? etc. (excepto o underscore).

Exemplos de identificadores válidos:

a) X b) X3 c) nomed) altura1 e) teste_11 f) a1b2c3

Exemplos de identificadores inválidos:

a) 1X b) X 3 c) A%1 d) B-2 e) maior que 10 f) >10

Tipos de Dados

Os algoritmos irão manipular dados, que normalmente são fornecidos pelos usuários, e passar os resultados para estes usuários. Uma pergunta importante neste momento é: que tipo de dados se poderem manipular? As linguagens de programação normalmente estabelecem regras precisas para definir que tipos de dados elas irão manipular. A pseudo-linguagem também estabelece, ainda que informalmente, algumas regras que limitam o conjunto de dados existentes na natureza e que poderão ser manipulados pelos algoritmos.

Existem três tipos básicos de dados que a linguagem irá manipular:

Dados numéricos: os dados numéricos que os algoritmos podem manipular são de dois tipos:

- Dados inteiros
- Dados reais

Dados alfa numéricos: servem para tratamento de textos e normalmente são compostos por uma seqüência de caracteres contendo letras, algarismos e caracteres de pontuação. Nos algoritmos são normalmente representados por uma seqüência de caracteres entre aspas, por exemplo:

- “Linguagem de programação”
- “Onde moras?”
- “12345”

Dados Lógicos: muito utilizado durante o processo de tomada de decisões que o computador frequentemente é obrigado a fazer. Em muitos textos este tipo de dados também é chamado de dados booleanos, devido a George Boole, matemático que deu

o nome à álgebra (álgebra booleana) que manipula este tipo de dados. Os dados deste tipo somente podem assumir dois valores: verdadeiro ou falso. Computadores tomam decisões, durante o processamento de um algoritmo, baseados nestes dois valores.

Exemplo:

```
Se raiz >= 0
        imprima “Existe raiz”
caso contrário
```

imprima "Não existe raiz real."

Conclusão

O aluno aprendeu a noção de algoritmo, de componentes que são necessários para escrevê-lo e ainda suas formas de representação e características. Estes conceitos são muito importantes especialmente para os iniciantes em matéria de Estrutura de Dados e Algoritmos uma vez que este capítulo, enfatiza estes termos que são a base para o aprendizado desta disciplina.

Avaliação

1. Usando um diagrama, descreva os passos necessários de algoritmos que possam ser utilizados na resolução de uma determinada tarefa.
2. Usando a linguagem natural, determinar se um determinado número, lido do teclado, é par ou ímpar.

Actividade 1.2 - Complexidade de algoritmos

Introdução

Pode-se construir vários algoritmos para resolver um dado problema. No entanto, esses algoritmos podem variar na forma de pesquisar os dados de entrada, processar e imprimir os dados de saída. Sendo assim, estes podem ter diferenças significativas em termos de performance e utilização de espaço. Entender como analisar algoritmos e como medir a eficiência algorítmica ajuda bastante na escolha do melhor algoritmo a ser usado (melhor optimizado).

Uma característica muito importante de qualquer algoritmo é o seu tempo de execução.

Analisar um algoritmo significa antever os recursos computacionais que o algoritmo requer a quando da sua execução, tais como o espaço de memória e tempo de execução. São vários algoritmos que existem para resolver um problema e a análise deverá identificar qual o mais eficiente. Nesta actividade, os estudantes aprendem a analisar e calcular a complexidade de algoritmos para verificar a sua eficiencia, de modo a escolher o melhor algoritmo a implementar.

Detalhes da atividade

A Complexidade de um Algoritmo consiste na quantidade de "trabalho" necessária para a sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados.

Em geral escreve-se o tempo de execução de um algoritmo como uma função f(n), onde

n é o tamanho da entrada. O tamanho da entrada, por sua vez, depende do problema computacional em questão. Tal função f deve expressar o número de operações fundamentais, ou passos, executados pelo algoritmo.

Para medir o custo de execução de um algoritmo, define-se uma função de custo ou função de complexidade f. f(n) é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n.

Exemplo: Considere os problemas de determinar as matrizes soma C e produto D de duas matrizes dadas e , ambas . C e D também são de dimensão e seus elementos são e respectivamente, podem ser calculados:

$$c_{ij} = a_{ij} + b_{ij} \quad e \quad d_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}.$$

O Algoritimo 1 descreve a computação da matriz soma de duas matrizes

```
Algoritmo Soma ( A , B )
1.      para i← 1 até n faça
2.          para j← 1 até n faça
3.              c_ij ← a_ij + b_ij ;
fim-algoritmo
```

O algoritmo 2 mostra a computação da matriz produto de duas matrizes

```
Algoritmo Mult ( A , B )
1.      para i← 1 até n faça
2.          para j← 1 até n faça
3.              d_ij ← 0;
4.              para k ← 1 até n faça
5.                  d_ij ← d_ij + a_ik · b_ij ;
fim-algoritmo
```

Ambos os algoritmos de soma e de produto, efectuam as mesmas operações sempre que A e B forem matrizes n x n. A variável independente é o parâmetro n. Cada passo do algoritmo 1 corresponde a execução da soma enquanto que no algoritmo 2, ao produto . O número total de passos é, pois igual ao número total da soma e produto respectivamente para cada caso. Ou seja o algoritmo 1 executa passos e o algoritmo 2 executa passos.

A seguir descreve-se a noção de complexidade temporal:

Seja A um algoritmo, ,o conjunto de todas as entradas possíveis de A. Denote por , o número de passos efectuados por A, quando a entrada for

Definem-se:

Complexidade de pior caso = ,

Complexidade de melhor caso =

Complexidade de pior caso =

Onde , é a probabilidade de ocorrência da entrada .

Analogamente pode-se definir a complexidade espacial de um algoritmo.

O objectivo das complexidades é de avaliar a eficiência de tempo ou espaço de um algoritmo. Interessa contar o número de passos que o algoritmo executa em seu pior caso, ou seja, para a entrada mais desfavorável. Esta complexidade fornece um llimite superior para o número de passos que o algoritmo pode efectuar em qualquer caso. Por isso tal medida é a mais utilizada. O termo complexidade será, então empregado com o significado de complexidade de pior caso.

A complexidade de melhor caso é de uso bem menos frequente, é usada em situações específicas. A complexidade de caso médio, apesar de importante, é menos utilizada do que a do caso médio, porque ela exige o conhecimento prévio das probabilidades das diferentes entradas do algoritmo e por outro lado em diferentes casos, essa distribuição é desconhecida. Além disso o cálculo de seu valor , frequentemente é de tratamento matemático complexo.

Exemplo: Cálculo da complexidade algorítmica

```
for(i=1;i<n;i++)
{
    for(j=1;j<n,j++)
     k= k+1;
   }
Solução:
for(i=1;i<n;i++)          O(n)   : executado n vezes
{
    for(j=1;j<n,j++)       O(n): executado n vezes
     k= k+1;     O(1): execução constante
   }
```

Multiplica-se desta forma: O(n)* O(n) = O(n * n) = O(n2), portanto complexidade quadrada.

Conclusão

A análise de algoritmos é uma tarefa muito importante em ciências de computação. Para comparar algoritmos, temos que ter alguns critérios para medir a eficiência dos algoritmos.

Avaliação

1. Dados dois algoritmos X e Y com complexidade n7 e 2n, respectivamente. Em que casos se utilizariam os algoritmo X ou Y? Dê exemplos.
2. Em que consiste a complexidade de um algorítmo? Enuncie as diferentes regras para análise de algorítmos

Actividade 1.3 - Notação assintótica

Introdução

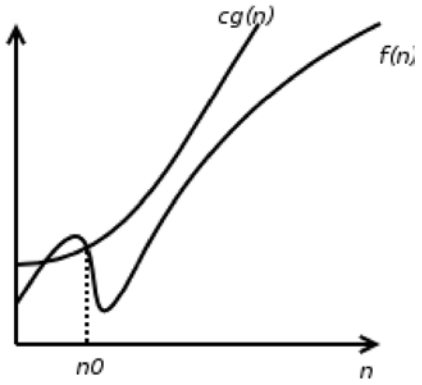
O mais importante na determinação do custo de um algoritmo é a identificação do termo dominante da expressão que descreve a sua complexidade. Este termo dominante, sem as constantes multiplicativas e as parcelas menores, descreve a ordem de crescimento assintótico desta expressão, quando consideramos o tamanho da entrada relativamente grande.

Detalhes da actividade

Um valor de número de passos igual a 3n será aproximado para n. Além disso, como o interesse é restrito a valores assintóticos, termos de menor grau também podem ser desprezíveis. Assim, um valor de número de passos igual a será aproximado para . É deveras importante descrever operadores matemáticos que sejam capazes de representar situações como essas. As notações serão utilizadas com essa finalidade, isto é, interessa é em encontrar uma função que é um limitante superior assintótico.

Definição: Dada uma função g(n), denotamos por O(g(n)) o conjunto das funções

Graficamente:



Exemplo 1:

Aquí interessa mostrar que existem constantes positivas c e tais que , . Isto vale para c = 2 e .

Portanto, denota-se)

Exemplo 2:

Aquí interessa mostrar que existem constantes positivas c e tais que , . Isto vale para c = 1 e .

Portanto,)

Mais alguns exemplos da notação O (O grande)

As seguintes propriedades são úteis para manipular expressões em notação O, elas decorrem directamente da definição e foram utilizadas nos exemplos anteriores.

a)

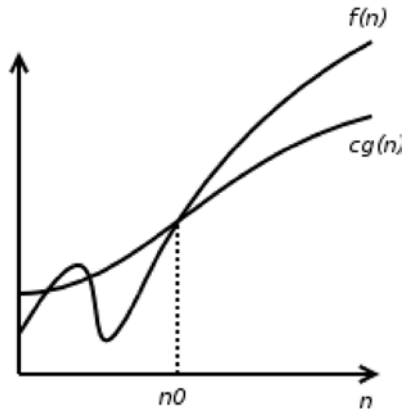
b) =O(g)

Conclusão: A notação O grande é útil para descrever limitantes superiores assintóticos.

Notação (ômega)

Esta notação é utilizada para limitantes inferiores assintóticos

Definição: Para uma dada função g(n), denota-se por (g(n)) o conjunto das funções



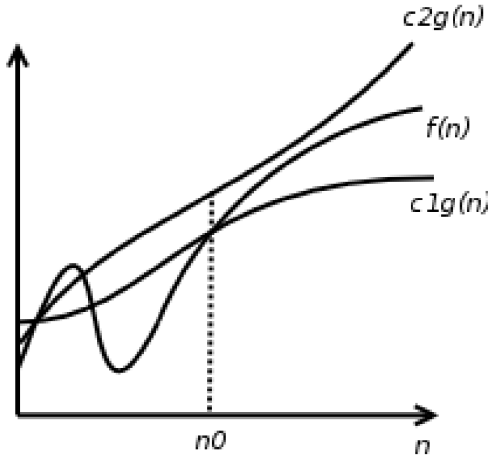
Exemplo: Deve-se mostrar que existem constantes positivas e tais que Isto é válido para .
Portanto,.

Notação

Esta notação é útil para exprimir limites superioes justos

Definição: Sejam f e g funções reais positivas da variável inteira n. Se existirem constantes positivas k, c1 e c2 tais que e então diz-se que e escreve-se quando ambas as condições e forem verificadas. A notação expressa o facto de que duas funções possuem a mesma ordem de grandeza assintótica.

Graficamente:



Exemplo: , e , então , , , mas h não é O(f). Consequentemente,), mas não é .

Da mesma forma, se e g=n, então , porém f não é , mas sim

Complexidade de Algoritmos Iterativos

A complexidade de algoritmos iteractivos é expressa através de somatórios, para encontrar a complexidade siga os seguintes passos:

Escolha uma unidade para medir o tamanho da entrada,

Identifique a operação básica do algoritmo,

Expresse o número de execuções da operaçãoo bássica por um somatório,

Ache uma forma fechada para o somatório,

De (4) derive a ordem de grandeza do algoritmo.

identidades úteis envolvendo somatórios

$$\sum_{i=l}^{u} cx_i = c\sum_{i}^{u} x_i$$

$$\sum_{i=l}^{u}(x_i \pm y_i) = \sum_{i=l}^{u} x_i \pm \sum_{i=l}^{u} y_i$$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

$$\sum_{i=1}^{u} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{u-1} i = 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

Exemplo: Determine a complexidade, no pior caso, do algoritmo abaixo que verifica se um array possui elementos distintos entre si.

Elementos distintos (A, n)

Tamanho da entrada = número de elementos do array A.

Operação básica = comparação (linha 4).

Contagem de operações:

$$C(n) := \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}[n - (i+1) + 1] = \sum_{i=1}^{n-1}(n-i)$$

$$:= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n\sum_{i=1}^{n-1} 1 - \frac{n(n-1)}{2}$$

$$:= n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Complexidade em algorítmos recursivos

A complexidade é expressa através de recorrências. Existem três formas de resolver

recorrências; método de substituição, árvore de recorrência e método mestre.

Análise de complexidade

Escolha uma unidade para medir o tamanho da entrada,

Identifique a operação básica do algoritmo,

Expresse o número de execuções da operação básica por uma recorrência, Ache uma forma fechada para a recorrência

De (4) derive a ordem de grandeza do algoritmo.

Método da Substituição

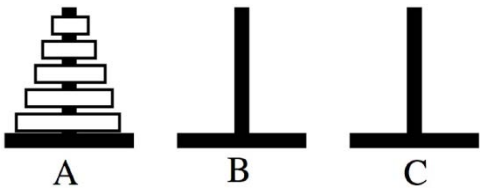
O método da substituição é um método eficiente quando for fácil pressupor a solução, e envolve dois passos:

- Pressupor a solução da recorrência
- Provar que a suposição é correcta por indução

Exemplo:

Considere um procedimento recursivo para resolver o problema da Torre de Hanoi (movendo os discos de A a C).

1. Resolva recursivamente o problema de mover os top n-1 discos do pino A para o pino B (usando o C como auxiliar). Resposta: (T(n - 1) movimentos)
2. Mova o disco n ao pino C. (1 movimento)
3. Resolva recursivamente o problema de mover os n - 1 discos do pino B para o pino C (usando A como auxiliar).



Quantos movimentos são realizados para n discos?

```
Hanoi(n, origem, destino, aux)
if n == 1
    print ( "Disco moveu de origem para destino")
else
```

Hanoi(n – 1, origem, aux, destino)

print (“Disco moveu de origem para destino”)

Hanoi(n – 1, aux, destino, origem)

Como para 1 disco realizamos 1 movimento, o número total de movimentos

pode ser dado pela seguinte relação de recorrência:

T(n) :=

Para encontrar a forma fechada da recorrência usa-se a abordagem expandir, conjecturar e verificar. Então, expandindo e aplicando a definição para n, n - 1, n - 2, etc.:

A expansão termina quando n-k=1, ou seja quando k=n-1 e então temos:

Resta verificar se a conjectura é verdadeira e se for , fica então provada a relação de recorrência.

Verificação por meio de indução

Base: , que é verdade pela recorrência

Hipótese:

Mostrar:

Algoritmos eficientes

Um algoritmo é chamado eficiente para um problema de tamanho n se o seu tempo de execução for limitado por um polinómio p(n). Diz-se então que o algoritmo é polinomial em n. Um algoritmo de (2n) não é eficiente, pois a função 2n cresce mais rapidamente que qualquer polinómio em n.

Os algoritmos de soma e multiplicação de matrizes vistos na secção anterio são, portanto, eficientes. Existem, no entanto, alguns problemas para os quais não se conhece algoritmo eficiente capaz resolvê-los.

Algoritmos óptimos

A função de complexidade está relacionada a um dado algoritmo, visa determinar o número de passos efectuados por um algoritmo específico sem levar em conta a possível existência de outros algoritmos que resolvem o mesmo problema. Seja P um problema. Um limitante inferior para P é uma função tal que a complexidade de pior caso de qualquer algoritmo que resolve P é . Isto é, todo algoritmo que resolve P, executa no mínimo passos. Assim, se existir algum algoritmo A cuja a complexidade seja , então A é denominada algoritmo óptimo para P. Nesse caso o limite é o melhor (maior) possível.

Intuitivamente, um algoritmo óptimo é aquele que apresenta a menor complexidade dentre todos aqueles que resolvem o mesmo problema. Assim como a notação O é conveniente para exprimir complexidades, a notação é utilizada para limites inferiores.

Existem limites inferiores naturais para determinados problemas. Qualquer algoritmo utilizado para somar duas matrizes n × n, por exemplo, deverá antes de qualquer coisa, ler as matrizes. Assim, um limite inferior para este problema é (n2). Portanto, pode-se concluir que o algoritmo soma visto na anteriormente é óptimo para este problema.

Conclusão

Neste capítulo, o aluno adquiriu conhecimentos sobre complexidade de algoritmos, como avaliar a eficiência de um algoritmo, bem assim, o aluno foi introduzido à notações predefinidas para comparar a complexidade de um algoritmo. Estes conhecimentos são necessários pois, antes de escolher que algoritmo a utilizar, é preciso saber analisar e avaliar quanto à sua eficiência.

Avaliação

1. Para duas funcoes g(n) e f(n) temos, f(n)=”teta”(n) se somente se f(n)= Ω(g(n)) e f(n)=O(g(n)). Explique o teorema acima.
2. Calcule a complexidade dos seguintes algoritmos que se seguem:

a) for (int i = 1; i < n; i ++)

```
{
min = i;
for (int j = i; j< m; j++)
{
if a[j] < a[min]
 {
min = j;
x = a[min];
a[min] = a[i];
a[i] = x;
}
```

```
        }
        }
b)      for ( i = 1; j< n; i++)
        for ( j = 1; j< =i; j++)
        soma=soma+1;
```

Resumo da Unidade

A presente unidade definiu os principais conceitos tais como Estruturas de Dados, Algoritmos, complexidade de um algorítimo, debruçou-se sobre as caracteristicas dos algoritmos, vantagens e desvantagens de cada técnica de representação de algoritmos. Vários exemplos ilustrativos também foram mostrados. A noção de complexidade de algoritmos foi abordada, as notações big O, Ômega e Theta, complexidade de algoritmos recursivos e o conceito de algoritmos óptimos.

Avaliação da Unidade

Verifique a sua compreensão!

Avaliação Sistemática 2: Análise e Cálculo da complexidade de algoritmos

Instruções

Nas perguntas que se seguem, deverá responder todas elas com clareza:

Avaliação:

1. Acabou de aprender as diferentes formas de representação de um algoritmo.

   a. Compare as várias formas e explique em que situações usaria cada uma delas

2. Represente um fluxograma para o cálculo da média de um aluno que fez três avaliações numa determinada disciplina.

3. Existem cinco candidatos a uma vaga para o professor associado. Durante a eleição (única chamada), os votos são registados em urna electrônica contendo o voto do eleitor, codificado pelo número do candidato (1, 2, 4 ou 5).

4. Escreva um algoritmo que leia os votos e determine as seguintes informações:

   a. O número de votos que cada candidato obteve;

b. O número de votos nulos;

c. O número de votos em branco

5. Escreva um algoritmo que permita a um usuário entrar com 3 números inteiros e os imprima em ordem crescente.

6. Refira-se às propriedades da notação Big O (O grande).

7. Quais das seguintes afirmações sobre o crescimento assintótico de funções não é verdadeira:

a. $2n^2 + 3n + 1 = O(n^2)$

b. Se $f(n) = O(g(n))$ $então\ g(n) = O(f(n))$

c. $logn^2 = O(log\ n)$

d. $se\ f(n) = O(g(n))\ e\ g(n) = O(h(n))\ então\ f(n) = O(h(n))$

e. $2^{n+1} = O(2^n)$

8. Calcule a complexidade algorítmica dos seguintes problemas:

a.
```
int a = 2;
    a = (b + 1);
    printf("Escreva um número");
```

b. $g(n) = (n^2 + 1)$

Critérios de Avaliação

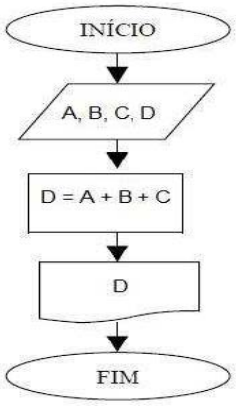
Pergunta	Pontuação (máximo 20 valores)
1-a)	2
2	2.5
3-a), b), c)	3
4	2.5
5-a)	2
5-b)	2

6-a)….e)	5 x 0.5
7-a)	2.0
7-b)	1.5
Total	20

Respostas:

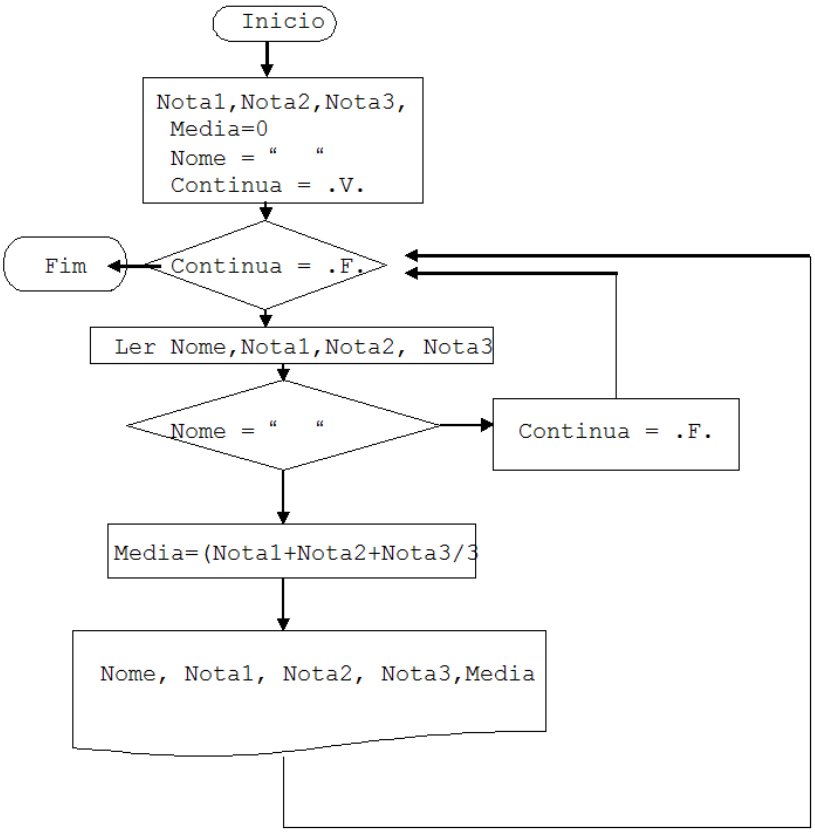
1. Acabou de aprender as diferentes formas ou técnicas de representação de um algoritmo.

   a. Compare as várias formas e explique em que situações usaria cada uma delas

Técnica	Conceito	Exemplo
Descrição narrativa	Consiste em analisar o enunciado do problema e escrever, utilizando uma linguagem natural (para o nosso caso, a língua portuguesa), os passos a serem seguidos para sua resolução.	Soma de três Números Recebe os três números 2. Somar os três números 3. Mostrar o resultado obtido
Fluxograma	Utiliza símbolos gráficos predefinidos para a resolução do problema	

Pseudo-código	Consiste em analisar o problema e escrever, por meio de regras predefinidas, os passos a serem seguidos para a resolução.	ALGORITMO Soma var A, B, C, D: inteiro inicio escreva (“Digite o valor de A: ”) leia (A) escreva (“Digite o valor de B: ”) leia (B) escreva (“Digite o valor de C: ”) leia (C) D <- A + B + C escreva (“D= ”, D) fimalgoritmo

2. Represente um fluxograma para o cálculo da média de um aluno que fez três avaliações numa determinada disciplina.



3. Existem cinco candidatos a uma vaga para o professor associado. Durante a eleição (única chamada), os votos são registados em urna eletrônica contendo o voto do eleitor, codificado pelo número do candidato (1, 2, 3, 4 ou 5). Como finalizador de entrada de dados, considere o valor zero (valor zero como voto).

Escreva um algoritmo que leia os votos e determine as seguintes informações:

a. total de votos para cada candidato;

b. total de votos nulos;

c. total de votos em branco

```
var
    cand1, cand2, cand3, cand4, cand5, votNulo, votBranco, voto: inteiro;
       cand1=0,  cand2=0,  cand3=0,  cand4=0,  cand5=0,  votNulo=0,
votBranco=0
 Ler
    escreva (“Informe o voto do eleitor: “)
    ler voto
    enquanto voto <> 0 faça
         inicio
       se voto<>1 e  voto<>2  e voto<>3 e voto<> 4 e voto<>5 e voto<>6 e
voto<>7
                  entao mostrar “Voto invalido!”
                  se voto =1 entao cand1=cand1+1
                  se voto =2 entao cand2=cand+1
                  se voto =3 entao cand3=cand3+1
                  se voto =4 entao cand4=cand4+1
                 se voto =5 entao cand5=cand5+1
                  se voto =6 entao votNulo=votNulo+1
                  se voto =7 entao votBranco=votBranco +1
           mostrar “Informe o voto do próximo eleitor: “
           ler voto
                fim
          escreva (“O total de votos para o candidato 1 é “), cand1
```

escreva (“O total de votos para o candidato 2 é “), cand2

escreva (“O total de votos para o candidato 3 é “), cand3

escreva (“O total de votos para o candidato 4 é “), cand4

escreva (“O total de votos para o candidato 5 é “), cand5

escreva (“O total de votos nulos é”), votNulo

escreva (“O total de votos em branco é “), votBranco

fim

4. Escreva um algoritmo que permita que o usuário introduza 3 números inteiros e os imprima em ordem crescente.

```
var a,b,c: inteiro
inicio
escreva(“Escreva três números inteiros: “)
leia(a,b,c)
se (a>b)e (b>=c) então
  escreva(c,b,a)
senao
  se (b>=a)e(b>c)e(a>=c)então
    escreva(c,a,b)
  senao
    se(b>a)e(b>=c)e(a<=c)então
      escreva(a,c,b)
    senao
      se (b>a)e(b<=c)então
        escreva(a,b,c)
      senao
        se(b<a)e(a<c)então
          escreva(b,a,c)
        senao
          se(b<c)e(c<=a)então
            escreva(b,c,a)
          senao
          escreval(a,b,c)
          fimse
        fimse
      fimse
    fimse
  fimse
fimse
fimalgoritmo
```

5. Refira-se às propriedades da notação Big O (O grande).

- f(n) = O(f(n))
- c. f(n) = O(f(n)), c=constante
- O(O(f(n))) = O(f(n))
- O(f(n)) + O(g(n)) = O(max(f(n),g(n))
- O(f(n)) * O(g(n)) = O (f(n) * g(n))

6. Quais das seguintes afirmações sobre o crescimento assintótico de funções não é verdadeira:

a. $2n^2 + 3n + 1 = O(n^2)$
Verdadeira
b. Se $f(n) = O(g(n))$ *então* $g(n) = O(f(n))$
Falsa
c. $logn^2 = O(log\ n)$
Falsa
d. *se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$
Verdadeira
e. $2^{n+1} = O(2^n)$
Verdadeira

7. Calcule a complexidade algorítmica dos seguintes problemas:

a. $int\ a = 2;$ ----------O (1)
$a = (b + 1);$ ---------- O (1)
$printf(\text{“}Escreva\ \ um\ número\text{”});$ -----O(1)

b. $g(n) = (n^2 + 1)$
Logo $g(n)$é $O(n^2)$quando $m=1$ e $c=4$. Isso porque $(n^2 + 1) <= 4n^2$ para $n>=1$.

Leituras e outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso.

- DERICK WOOD. Data Structures, Algorithms and Performance , Addison Wesley, 1993.
- https://www.youtube.com/watch?v=P-I1-WJSly4
- www.inf.pucrs.br/~pinho/LaproI/Recursividade/Recursividade.htm

Unidade 2. Recursividade

Introdução à Unidade

Esta unidade introduz ao aluno uma importante ferramenta, ensinando-o, o conceito de recursividade e outros conceitos a ele relacionados. Recursão é um conceito importante em informática. Muitos algoritmos podem ser bem descritos em termos recursivos. . Aborda ao aluno a forma como se pode resolver um problema, reduzindo-o a instância do mesmo problema com dados de entrada menores.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Descrever algoritmos de forma recursiva
- Implementar procedimentos recursivos
- Implementar funções recursivas

Termos-chave

Recursão: é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de deinição circular. Assim, pode-se dizer que o conceito de algo recursivo está dentro de si, que por sua vez está dentro de si e assim sucessivamente, infinitamente.

Algoritmos recursivos: Um algoritmo recursivo faz chamadas a si próprio (para resolver a instância menor)

Caso base: dada uma instância do problema P, testa-se o parâmetro n. Quando n é o menor possível, resolve-se essa instância diretamente. É o caso base.

Actividades de Aprendizagem

Actividade 2.1 - Conceitos básicos de recursão

Introdução

Recursão é um tema importante em ciências de computação. Muitos algoritmos podem ser melhor descritos em termos de recursão. Esta unidade, introduz esta poderosa ferramenta.

Detalhes da actividade

Um objecto é denominado recursivo quando a sua definição é parcialmente feita em termos dele mesmo. A recursividade (ou recursão) é encontrada principalmente na matemática, mas está presente em algumas situações do quotidiano. Por exemplo, quando um objecto é colocado entre dois espelhos planos paralelos e frente a frente surge uma imagem recursiva, porque a imagem do objeto refletida num espelho passa a ser o objeto a ser refletido no outro espelho e, assim, sucessivamente. Em programação, a recursividade é um mecanismo útil e poderoso que permite a uma função chamar a si mesma direta ou indiretamente, ou seja, uma função é dita recursiva se ela contém pelo menos uma chamada explícita ou implícita a si própria.

Suponha que P é um procedimento que contém uma expressão de chamada a si próprio ou uma expressão para um segundo procedimento que porventura poderá ter chamada de volta ao procedimento original P. Então P é um procedimento recursivo. Para que o programa não seja executado indefinidamente, o procedimento recursivo deverá possuir as seguintes propriedades:

Deverá existir um critério denominado base, para o qual, o procedimento não chame a si próprio.

Em cada instância em que o procedimento chama a si próprio, deverá estar próximo ao critério base.

Exemplo 1: Função que chama a si própria

int função(int valor)

```
{
   se (valor < 1)
      retorna;
   função(valor - 1);
   escreva("valor");
}
```

Exemplo 2: Função que chama outra função e esta por sua vez volta a chamar a função de chamada

```
int função(int valor)

{
```

```
        se (valor < 1)
        retorna;
        função(valor -1);
      escreva( "valor");
}
```

Propriedades:

Uma função recursiva pode correr infinitamente como um loop. Para evitar esta situação, existem duas propriedades que uma função recursiva deve possuir:

Critério comum- Deve haver pelo menos um critério de base ou condição de parada, isto é, quando esta condição for acautelada, a função pára de fazer chamadas de si própria recursivamente.

Abordagem progressiva - As chamadas recursivas devem evoluir de tal modo que cada vez que haja uma chamada recursiva, haja também uma aproximação ao critério base.

Implemetação

Várias linguagens de programação implementam as funções recursivas através de pilhas. De um modo geral, sempre que uma função (chamador) chama outra função (chamada) ou, chama a si própria (chamada), a função chamador, transfere o controlo de execução do fluxo para a função chamada. Isto é, a função chamador tem de suspender temporariamente a sua execução e retomar mais tarde, quando o controlo de execução retornar de função chamada. No entanto, a função chamador precisa de começar exatamente do ponto de execução em que se colocou em espera. Para além disso, precisa também dos mesmos dados com que estava trabalhando no momento de espera. Para este efeito, um registro de activação (ou quadro de pilha) é criado para a função de chamada.

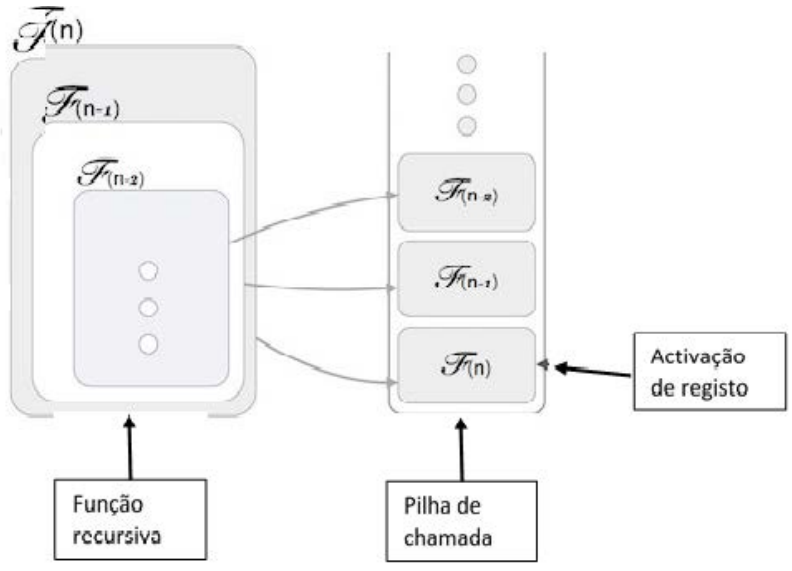


Figura: Implementação de recursão em pilhas

Nota: A activação de registo mantém as informações sobre variáveis locais, parâmetros formais, endereço do remetente e todas as informações passadas á função chamada.

Análise de recursão

Pode-se questionar a razão do uso de recursão se, a mesma tarefa pode ser realizada através de iteração. A resposta é simples pois, o uso de recursão, torna os programas mais legíveis por um lado, e por outro lado, com a tendência actual de , melhoramento de sistemas de CPU, a recursão é mais eficiente que a iteração.

Complexidade temporal

Em caso de iterações, tomamos o número de iterações para contar o tempo de complexidade. Em caso de funções recursivas, assumimos que tudo é constante, sendo assim, tentamos descobrir o número de vezes de uma chamada recursiva. Uma chamada feita para uma função é 0 (1), daí (n) vezes que uma chamada recursiva é feita torna a função recursiva O(n).

Complexidade de espaço

A complexidade de espaço refere-se a quantidade de espaço extra que é necessário para que um módulo seja executado. Em caso de iterações, o compilador dificilmente requere qualquer espaço extra. O compiler mantém atualizando os valores das variáveis utilizadas nas iterações. Mas no caso de recursão, o sistema precisa de armazenar o registro de ativação cada vez que uma chamada recursiva é feita. Assim, considera-se que a complexidade do espaço numa função recursiva pode ser maior do que de uma função com iteração.

Conclusão:

A sequência de Fibonacci nada mais é que uma sucessão de números, e para obter cada número da sequência deve-se somar os dois últimos algarismos para obter o próximo. O triângulo de Pascal, o filme de ficção "O código da Vinci", são algumas das aplicações da série de Fibonacci.

Actividade 2.2 - Séries de Fibonacci

Introdução

Sequência de Fibonacci é uma sucessão de números que, misteriosamente, aparece em muitos fenômenos da natureza, em configurações biológicas, como por exemplo, na disposição dos ramos das árvores ou das folhas em uma haste, no arranjo do cone da alcachofra, ou de um ananás. As aplicações das séris de Fibonacci são vastas desde a análise de mercados financeiros, a ciência da computação e a teoria de jogos.

Detalhes da actividade

A série de Fibonacci foi descrita no final do século 12 pelo italiano Leonardo Fibonacci, ela é infinita e começa com 0 e 1. Os números seguintes são sempre a soma dos dois números anteriores. Portanto, depois de 0 e 1, vêm 1, 2, 3, 5, 8, 13, 21, 34…

Os números de Fibonacci ligam-se facilmente à natureza. É possível encontrá-los no arranjo das folhas do ramo de uma planta, nas copas das árvores ou até mesmo no número de pétalas das flores. Pode-se também encontrar a espiral de Fibonacci nas sementes das flores, em frutos e pinhas.

Ao transformar esses números em quadrados e dispô-los de maneira geométrica, é possível traçar uma espiral perfeita, que também aparece em diversos organismos vivos.

Exemplo 1: Coelhos

Um homem pôs um casal de coelhos em um lugar totalmente cercado. Quantos casais de coelhos podem ser gerados por esse casal em um ano se por hipótese em cada mês cada casal gera um novo casal, o qual começa a se reproduzir a partir do segundo mês de vida ?

Usando uma tabela para demonstrar essa hipótese:

Mês	Casais Maduros	Casais Novos
1	1	0

No primeiro mês não houve nenhum casal novo

Segundo mês

Mês	Casais Maduros	Casais Novos
1	1	0
2	1	1

No segundo mês houve um casal novo

Terceiro mês

Mês	Casais Maduros	Casais Novos
1	1	0
2	1	1
3	2	1

No terceiro mês esses dois casais já estão na fase reprodutiva e há ainda um novo casal, nascido do primeiro par.

Para continuar a encontrar os o nº de novos casais, podemos fazer da seguinte forma:

- Calculamos o número de casais maduros somando os casais que já eram maduros antes com aqueles que eram casais novos no mês anterior.
- O número de novos casais a cada mês é exatamente igual ao número de casais maduros no mês anterior.

Pode-se escrever o mesmo exemplo em forma numérica, de modo a construir o algoritmo:

Uma série de Fibonacci é uma seqüência de valores definida da seguinte maneira:

Os dois primeiros termos são iguais a unidade, ou seja,

Cada termo, a partir do terceiro, é igual à soma dos dois termos anteriores, isto é:

$$T_N = T_{N-2} + T_{N-1}$$

Se $T_{18} = 2584$ e $T_{21} = 10946$ então qual será o valor de T_{22}?

Se e então qual será o valor de ?

Resolução

$$T_{22} = T_{21} + T_{20}$$

$$T_{21} = T_{20} + T_{19}$$

$$T_{20} = T_{19} + T_{18}$$

Então:

$$T_{21} - T_{20} = T_{20} - T_{18}$$
$$T_{21} + T_{18} = 2T_{20}$$
$$T_{22} = T_{21} + \frac{T_{21} + T_{18}}{2}$$
$$T_{22} = 10946 + \frac{10946 + 2584}{2} = \frac{13530}{2} + 10946 = 6765 + 10946 = 17711$$

Nota: A fórmula acima, é utilizada para obter o valor de um certo elemento na Sequência de Fibonacci, para que não precisemos ficar calculando número a número. De um ponto de vista computacional, podemos dizer que a fórmula acima é um algoritmo, ou seja, quando aplicada em uma linguagem computacional, obtemos o resultado desejado de forma muito rápida..

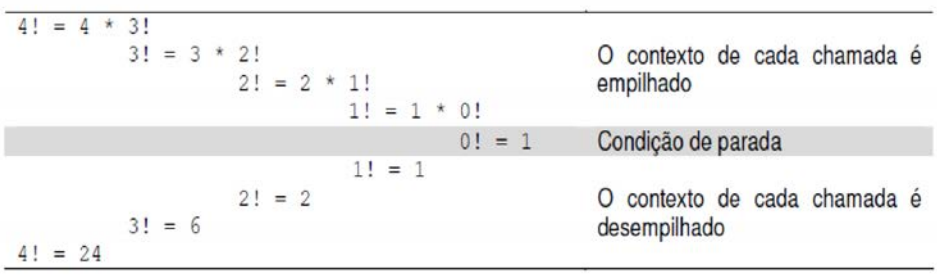
Exemplo 2: FACTORIAL(n)

$$se\ n\ =\ 0$$
$$então\ retorne\ 1;$$
$$senão\ retorne\ n\ *\ FACTORIAL(n-1);$$

Por definição, o valor de é calculado da seguinte maneira:

$$4! = 4 * 3! = 4 * (3 * 2!) = 4 * (3 * (2 * 1!)) = 4 * (3 * (2 * (1 * 0!)))$$
$$= 4 * (3 * (2 * (1 * 1))) = 24$$

Note que a função é chamada recursivamente com argumento decrescente até chegar ao caso trivial (0!), cujo valor é 1. Este caso trivial (condição de parada) encerra a sequência de chamadas recursivas. A sequência de chamadas é melhor ilustrada abaixo:



Exemplo 3: A sequência [0, 1, 1, 2, 3, 5, 8, 13, 21, ...] é conhecida como seqüência ou série de

Fibonacci tem aplicações teóricas e práticas, na medida em que alguns padrões na natureza parecem segui-la.

Pode ser obtida através da definição recursiva:

$$Fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{se } n > 1 \end{cases},$$

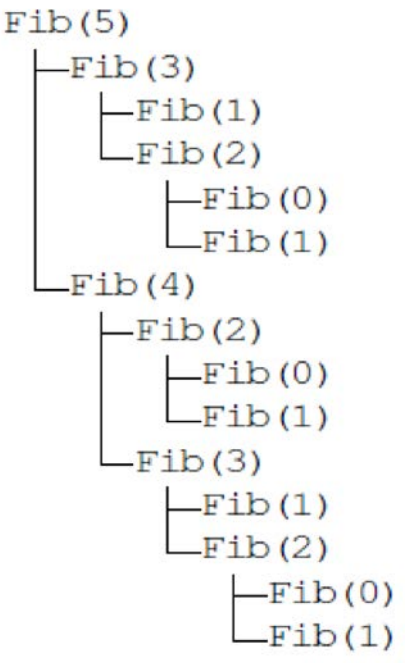
E pode ser implementada de forma seguinte:

Fibonacci(n)

```
se n <= 2
então returna 1;
senão
        retorna Fibonacci (n − 1) + Fibonaccii (n − 2)
```

Note que, para n > 1, cada chamada causa 2 novas chamadas de Fib, isto é, o número total de chamadas cresce exponencialmente. Observe no diagrama de execução abaixo que para Fib(5), são feitas 14 chamadas da função. Além disso, Fib(0) e Fib(2) são chamadas 3 vezes; Fib(1) é chamada 5 vezes. Para Fib(25) são feitas 242784 chamadas recursivas!

No caso da sequência de Fibonacci é relativamente simples implementar um algoritmo iteractivo com complexidade O(n), que tire proveito dos valores já calculados.



Conclusão:

A sequência ou série de Fibonacci nada mais é que uma sucessão de números,e para obter cada número da sequência deve-se somar os dois últimos algarismos para obter o próximo. O triângulo de Pascal, o filme de ficção "O código da Vinci", são alguns dos exemplos de aplicação da série de Fibonacci.

Avaliação

1. Os dois primeiros números são 1 e 1, e o resto é a soma dos dois anteriores. Imprima os 10 primeiros.
2. Faça um programa que calcule o n-ésimo termo da sequência de fibonacci.

, ,

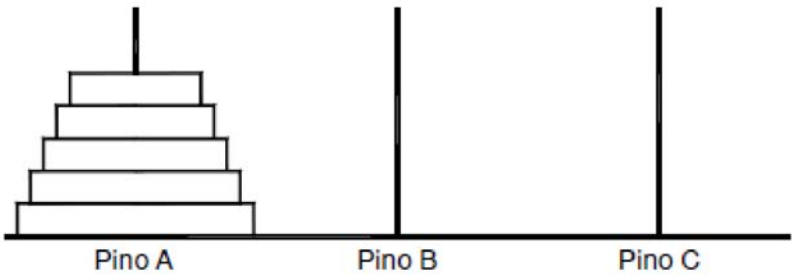
Actividade 2.3 - Torre de Hanói

Introdução

Na presente actividade discutiremos alguns exemplos de procedimentos recursivos, mostrando como as funções recursivas podem ser usadas como ferramentas no desenvolvimento de algoritmos para resolver um determinado problema

Detalhes da actividade

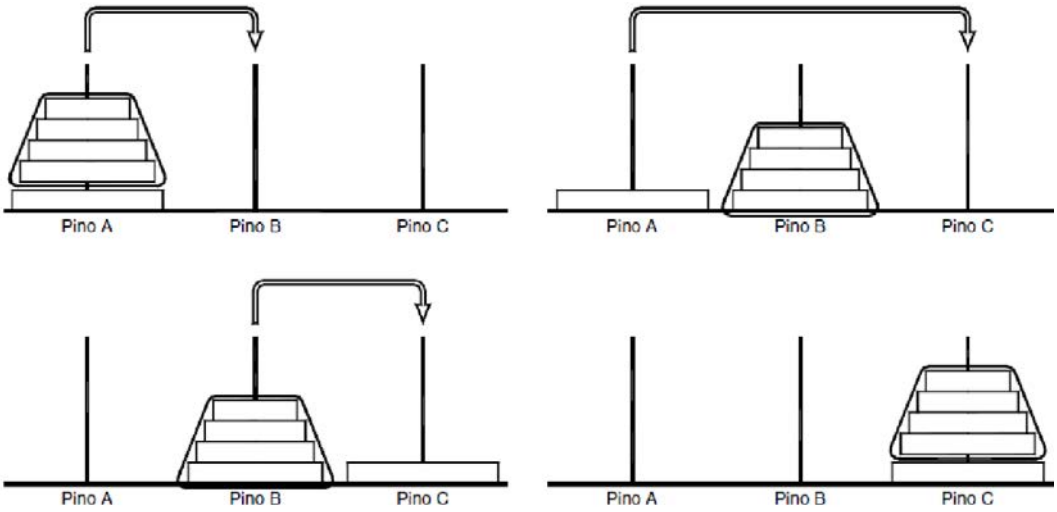
O problema ou quebra-cabeça conhecido como torre de Hanói foi publicado em 1883 pelo matemático francês Edouard Lucas, também conhecido por seus estudos com a série Fibonacci. Consiste em transferir, com o menor número de movimentos, a torre composta por N discos do pino A (origem) para o pino C (destino), utilizando o pino B como auxiliar. Somente um disco pode ser movimentado de cada vez e um disco não pode ser colocado sobre outro disco de menor diâmetro.



Solução: Transferir a torre com N-1 discos de A para B, mover o maior disco de A para C e transferir a torre com N-1 de B para C. Embora não seja possível transferir a torre com N-1 de uma só vez, o problema torna-se mais simples: mover um disco e transferir duas torres com N-2 discos. Assim, cada transferência de torre implica em mover um disco e transferir de duas

torres com um disco a menos e isso deve ser feito até que torre consista de um único disco.

Solução do problema graficamente:



Usando pseudocódigo para a solução do problema

```
TRANSFIRADISCOS (N, origem, destino, auxiliar)
{transfere n discos contidos na torre A para a torre C auxiliando-se do pino C, se for necessario}
se  N=1
        então move o disco da origem para o destino
        senão inicio
        TRANSFIRADISCOS(N-1, origem, auxiliar, destino)
        mover disco de origem para o destino
        TRANSFIRADISCOS(N-1, auxiliar, destino, origem )
fim
No main
se N>0
então TRANSFEREDISCOS (N, origem, destino, auxiliar)
fim
```

Vantagens

Os algoritmos normalmente são mais compactos, mais legíveis e mais fáceis de serem compreendidos. Algoritmos para resolver problemas de natureza recursiva são fáceis de serem implementados em linguagens de programação de alto nível.

Desvantagens

Por usarem intensamente a pilha, o que requer alocações e desalocações de memória, os algoritmos recursivos tendem a ser mais lentos que os equivalentes iteractivos, porém pode valer a pena sacrificar a eficiência em benefício da clareza. Algoritmos recursivos são mais difíceis de serem depurados durante a fase de desenvolvimento.

Aplicações

Nem sempre a natureza recursiva do problema garante que um algoritmo recursivo seja a melhor opção para resolvê-lo. O algoritmo recursivo para obter a seqüência de Fibonacci é um óptimo exemplo disso.

Os algoritmos recursivos são aplicados em diversas situações como em:

1. problemas envolvendo manipulações de árvores;
2. analizadores léxicos recursivos de compiladores; e
3. problemas que envolvem tentativa e erro (“Backtracking”).

Vejamos o pseudocódigo do algoritmo recursivo

```
função fatorial(n)
  se n=1 entao
    fatorial = 1
  senao
    fatorial = n * fatorial(n-1)
fim funcao
```

Conclusão

O problema da torre de Hánoi ilustrou o poder da recursão na solução de vários problemas algorítmicos. Mostrou também como se pode utilizar as pilhas para implementar a recursão.

Resumo da Unidade

Nesta unidade o aluno foi introduzido ao conceito de recursividade e de algoritmos recursivos. Nele aprendeu que os algoritmos, funções e procedimentos podem chamar-se a si próprios e, que deverá haver um mecanismo para que não haja chamadas infinitas.

Um algoritmo recursivo deve fazer pelo menos uma chamada a si mesmo, de forma direta (pode-se ver o algoritmo sendo chamado dentro dele mesmo) ou indireta (o algoritmo chama um outro algoritmo, que por sua vez invoca uma chamada ao primeiro).

Um algoritmo recursivo deve ter pelo menos uma condição de parada, para que não seja invocado indefinidamente. Esta condição de parada corresponde a instâncias suficientemente pequenas ou simples do problema original, que podem ser resolvidas diretamente.

Para todo algoritmo recursivo existe pelo menos um algoritmo iterativo correspondente e vice-versa. Todavia, muitas vezes pode ser difícil encontrar essa correspondência.

Ao escrever funções recursivas, deve-se ter um comando se (if) em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se não existir, a função nunca retornará quando chamada (equivalente a um loop infinito).

Avaliação da Unidade

Avaliação Sistemática 3: Recursividade

Critérios de Avaliação

Pergunta	Pontuação (Máximo 10 valores)
1	3.5
2	3.5
3	3.0

Avaliação

1. Explique o conceito de algoritmos recursivos.
2. Qual é a importância de utilização de algoritmos recursivos. Dê exemplos.
3. O que se deve ter em conta na aplicação de algoritmos recursivos?

Respostas

1. Explique o conceito de algoritmos recursivos.

   A idéia principal de um algoritmo recursivo consiste em diminuir sucessivamente o problema em um problema menor ou mais simples, até que o tamanho ou a simplicidade do problema reduzido permita resolvê-lo de forma direta, sem recorrer a si mesmo. Quando isso ocorre, diz-se que o algoritmo atingiu uma condição de parada, a qual deve estar presente em pelo menos um local dentro algoritmo.

2. Qual é a importância de utilização de algoritmos recursivos. Dê exemplos.

   A idéia principal de um algoritmo recursivo consiste em diminuir sucessivamente o problema em um problema menor ou mais simples, até que o tamanho ou a simplicidade do problema reduzido permita resolvê-lo de forma directa, sem recorrer a si mesmo.

   Exemplo:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

   Função Fat(n : natural) : natural

   início

   se n = 0 então

   retorna 1

   senão

   retorna n * Fat(n - 1)

   fim

3. O que se deve ter em conta na aplicação de algoritmos recursivos?

   Ao desenhar algoritmos recursivos, é necessário ter em conta a condição de parada para que o algoritmo não faça chamadas indefinitivas.

Leituras e outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso.

http://www.desarrolloweb.com/manuales/manual-iniciacion-programacion.html

https://www.dcc.fc.up.pt/~nam/aulas/9900/pi/slides/slipi9916/

Unidade 3. Estrutura de Dados Básicas e Tipo de Dados Abstractos

A presente secção introduz ao aluno aos conceitos relacionados às Estruturas de Dados (ED) e Tipos de Dados Abstractos (TAD). Explica as vantagens e desvantagens de estruturas de dados.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Definir os conceitos Estrutura de Dados, Tipos de Dados e Tipos Abstractos de Dados.
- Descrever os conceitos Estrutura de Dados, Tipos de Dados e Tipos Abstractos de Dados.
- Comparar Estrutura de Dados e Tipos Abstratos de Dados
- Implementar estruturas de dados, em particular, recorrendo a gestão dinâmica de memória.
- Escrever algoritmos para implementar os TAD.

Termos-chave

Estrutura de Dados: é um modelo lógico ou matemático de uma organização particular de dados.

Tipo de Dados Abstratos: é um conjunto de valores e uma colecção de operadores que actuam sobre esses valores. As operações devem ser consistentes com os tipos de valores.

Vector: é uma estrutura de dados linear utilizado para armazenar uma lista de valores do mesmo tipo de dados e, necessita somente de um índice para que seus elementos sejam endereçados.

Matriz: é um arranjo bidimensional ou multidimensional de alocação estática e sequencial, uma estrutura de dados que necessita de um índice para referenciar a linha e outro para referenciar a coluna para que seus elementos sejam endereçados.

Listas ligadas:é uma estrutura de dados onde cada elemento contém um elo (endereço) para o próximo elemento da lista. Desta forma a lista pode ser acessada aleatoriamente, podendo ser removidos, adicionados ou inseridos elementos no meio da lista de forma dinâmica.

Pilha: é um conjunto ordenado de itens, no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados itens de uma extremidade, chamada topo da pilha.

Fila:é um conjunto ordenado de elementos a partir do qual pode-se remover elementos numa extremidade chamada início da fila, e no qual se podem inserir elementos na outra extremidade denotado, final da fila.

Tabela de dispersão: é uma estrutura de dados que associa chaves com valores.

Actividades de Aprendizagem

Actividade 3.1 –Tipos de dados

Introdução

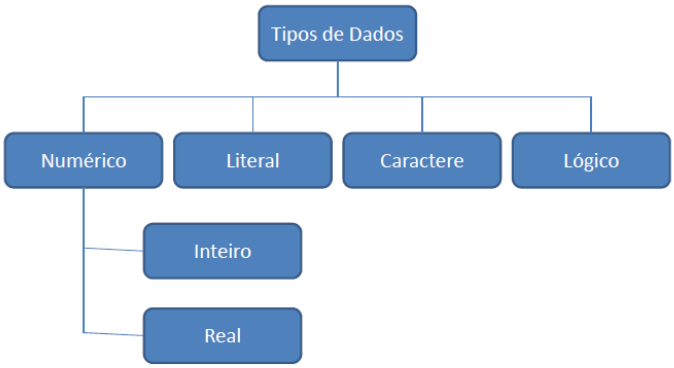
Uma aplicação em Ciências da Computação é basicamente um programa de computador que manipula dados. A representação dos dados manipulados por uma aplicação pode ser feita por diferentes estruturas de dados. Um factor que determina o papel dessas estruturas no processo de programação de aplicações é a identificação do quão bem as estruturas de dados coincidem com o domínio do problema a ser tratado. Portanto, é essencial que as linguagens de programação tenham suporte a uma variedade de tipos e estruturas, para que representem adequadamente dados manipulados pelas aplicações. Embora os termos tipos de dados e estruturas de dados sejam semelhantes, eles têm significados diferentes.

Detalhes da actividade

Um tipo de dados consiste basicamente da definição do conjunto de valores (denominado domínio) que uma variável pode assumir ao longo da execução de um programa e do conjunto de operações que podem ser sobre ele. Por exemplo, o tipo de dado inteiro pode assumir valores inteiros (negativos, zero, positivos), as operações possíveis sobre este tipo de dado são adição, subtracção, multiplicação, divisão, entre outras. As principais linguagens de programação oferecem uma grande variedade de tipo de dados, classificados em básicos e estruturados.

Tipos de dados básicos também denominados tipos primitivos

Os tipos de dados primitivos não possuem uma estrutura sobre os seus vaçoresou seja, não é possível decompor o tipo primitivo em partes menores isto é, são indivisíveis. As principais linguagens de programação, possuem um elenco de tipos de dados básicos, algumas linguagens ainda subdividem esses tipos de dados em outros de acordo com a capacidade de memória necessária para a variável. Mas de modo geral, os tipos de dados primitivos são:



Numérico sendo inteiro ou decimal:

INTEIRO: representa valores numéricos negativos ou positivos sem casa decimal, ou seja, valores inteiros.

DECIMAL: representa valores numéricos negativo ou positivo com casa decimal, ou seja, valores reais. Também são chamados de ponto flutuante.

LÓGICO: representa valores booleanos, assumindo apenas dois estados, VERDADEIRO ou FALSO. Pode ser representado apenas um bit (que aceita apenas 1 ou 0).

LITERAL: representa uma sequência de um ou mais de caracteres, colocamos os valores do tipo TEXTO entre " " (aspas duplas).

Para a resolução de algoritmos complexos, os tipos de dados primitivos tornam-se insuficientes para a resolução de problemas, e para tal, existem outros tipos de dados como por exemplo os tipos de dados estruturados.

Tipos de dados estruturados

Os tipos de dados estruturados permitem agregar mais do que um valor em uma variável, existindo uma relação estrutural entre os seus elementos. As linguagens de programação oferecem mecanismos para estruturar dados complexos, quando esses dados são compostos por diversos campos. Os principais tipos de dados estruturados fornecidos pelas linguagens de programação são: arranjos (também denominados vectores e matrizes, utilizados para agregar componentes do mesmo tipo, com um tamanho máximo predefinido), registos (para agregar componentes de tipos diferentes), sequências (para colecções ordenadas de componentes do mesmo tipo) e conjuntos (para definir, para um tipo básico, uma faixa de valores que seus componentes podem assumir), entre outros.

Vector

É uma estrutura de dados muito utilizada. É importante notar que vectores são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Este tipo de dados é utilizado para armazenar uma lista de valores do mesmo tipo, ou seja, permite armazenar mais de um valor na mesma variável. Um dado vector é definido como tendo um número fixo de células idênticas (seu conteúdo é dividido em posições). Cada célula armazena um e somente um dos valores de dados do vector, cada célula do vector possui seu próprio endereço, ou índice, através do qual pode ser referenciada.

Forma geral: tipo_da_variável nome_da_variável [tamanho];

Características:

Alocação estática (no momento da declaração deve-se especificar o tamanho do vector)

Estrutura homogênea

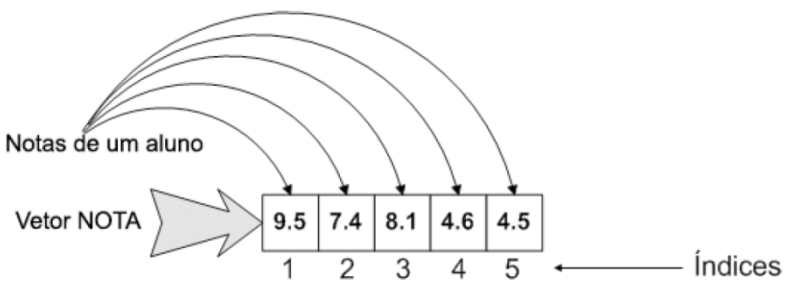
Alocação sequêncial (bytes contíguos)

Inserção/Exclusão

Realocação dos elementos

Não libertação da memória alocada

Figura 3.1: vector referente às notas de um aluno



Algoritmo 3.1

Endereço Elemento 1 ⟸ endereço inicial

Endereço Elemento 2 ⟸ endereço inicial + tamanho Elemento

Endereço Elemento 3 ⟸ endereço inicial + (2 * tamanho Elemento)

Endereço Elemento 2 ⟸ endereço inicial + (3 * tamanho Elemento)

A partir do endereço do primeiro elemento é possível determinar a localização dos demais elementos do vector, porque os elementos do vector estão dispostos na memória um ao lado do outro e cada elemento tem o seu tamanho fixo.

A partir do algoritmo 3.1 é possível deduzir a fórmula genérica para o cálculo de posição na memória de um elemento qualquer. Sendo n o elemento, a fórmula se dá por

Posn = endereço inicial + ((n - 1) * tamanho do tipo do elemento).

A figura 3.1 mostra um vetor de notas de alunos, a referência NOTA[4]

indica o valor 4.6 que se encontra na coluna indicada pelo índice 4.

Matrizes

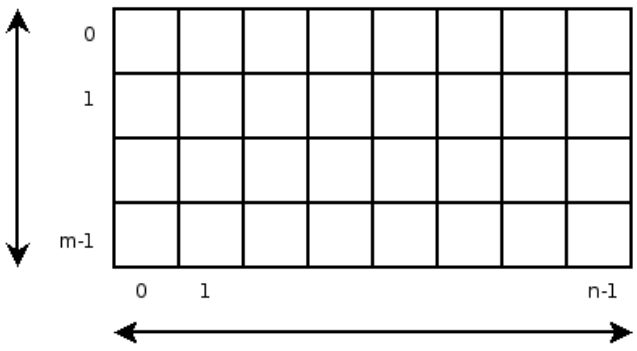
São tipos de dados estruturados definidas com um tamanho fixo, todos os elementos são do mesmo tipo, cada célula contém somente um valor que necessita de dois índices, um para referenciar a linha e outro para referenciar a coluna, para que os seus elementos sejam endereçados.

Os elementos ocupam posições contíguas na memória. A alocação dos elementos da matriz na memória pode ser feita colocando os elementos linha-por linha ou coluna-por-coluna.

Forma geral

tipo_da_variável nome_da_variável [dimensão1][dimensão2];

Figura 3.2: Ilustração esquemática da matriz de duas dimensões ou bidimensional



Algoritmo 3.2

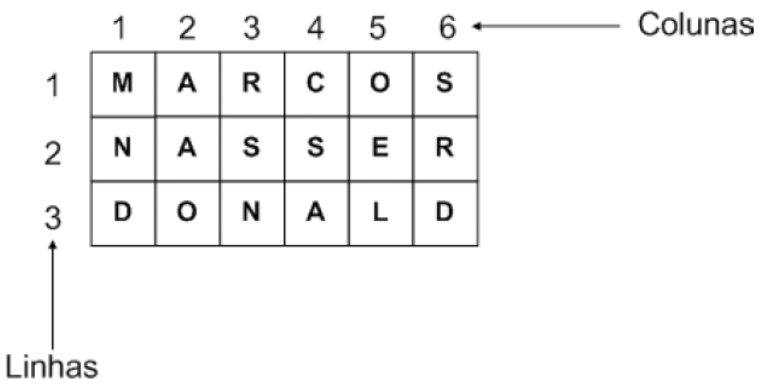
M00 ⟵ endereço inicial

M01 ⟵ endereço inicial + 1 * tamanho Elemento

M10 ⟵ endereço inicial + i * C * tamanho Elemento

M11 ⟵ endereço inicial + i* C * tamanho Elemento + j * tamanho Elemento

Uma matriz consiste em dois ou mais vectores definidos por um conjunto de elementos. Cada dimensão de uma matriz é um vector. O primeiro conjunto (dimensão) é considerado o primeiro vector, o segundo conjunto o segundo vector e assim sucessivamente.



Exemplo 3.3: matriz bidimensional de letras

Registo:

Registos são estruturas de dados logicamente relacionados, que comportam tipos diferentes: numérico, literal, lógico. O estudo até então se restringia às estruturas que comportavam dados de um único tipo.

Exemplo: suponha que queira armazenar os dados de funcionários de uma empresa constantes em uma ficha, contendo: INSCRIÇÃO, NOME, RUA, NÚMERO, CEP, NUIT, SEXO, DATA DE NASCIMENTO, DEPENDENTES, HORAS TRABALHADAS. Com as estruturas de dados vistas até agora, não seria muito fácil armazenar as informações acima. O conceito de registo visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que guardam estreita relação lógica. Na variável composta homogênea, a individualização de um elemento é feita através de índices. Já no registo, cada componente é individualizado pela explicitação de seu identificador. Registos correspondem a conjuntos de posições de memória conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições. Os elementos do conjunto não precisam ser, necessariamente, do mesmo tipo. O registo é constituído por componentes. Cada tipo de dado armazenado em um registo é chamado de campo.

Exemplo: declarar um registo para armazenar as informações da ficha de um funcionário descrita anteriormente.

declare FUNCIONARIO registo (INSCRICAO, NUMERO, DEPENDENTES, HORAS_TRAB numérico; NOME, RUA, CEP, CPF, SEXO, DATA_NASC literal);

Uma vez criada a variável estrutura FUNCIONÁRIO, seus membros podem ser acedidos por meio do ponto:

FUNCIONARIO.INSCRICAO ← 7213;

FUNCIONARIO.NOME ← "Pedro Ferrão";

FUNCIONARIO.RUA ← "Rua das Acácias";

FUNCIONARIO.NUMERO ← 100;

FUNCIONARIO.NUIT ← "358.982.153-00";

FUNCIONARIO.SEXO ← "M";

FUNCIONARIO.DATA_NASC ← "02/12/1964"; FUNCIONARIO.DEPENDENTES 2;

FUNCIONARIO.HORAS_TRAB ← 40;

Conjuntos:

Um conjunto é uma colecção que não possui elementos duplicados, onde não há noção de «ordem dos elementos». Ele pode ser mantido ordenado ou não sendo implementado normalmente em tabela Hash" ou "Árvore" As operações mais importantes de uma coleção do tipo Conjunto são:

Adição de elementos

Adicionar um objeto no conjunto (descartando duplicações)

Remoção de elementos

Remover um objeto presente no conjunto

Acesso aos elemento

Iterar sobre os elementos

Pesquisa de elementos

Descobrir se um certo elemento está na colecção

Indagar sobre atributos

Obter o número de elementos

Conclusão:

Um tipo de dados consiste basicamente da definição do conjunto de valores (denominado domínio) que uma variável pode assumir ao longo da execução de um programa e do conjunto de operações que podem ser sobre ele. Para além do tipo de dados primitivos existem também outros tipos de dados definidos pelo usuário e estruturados tais como matrizes unidimensionais e de várias dimensões, com alocação estática e sequencial, conjuntos e registos.

Avaliação

1. Elabore uma síntese dos tipos de dados.
2. Explique a diferença entre matrizes e vectores.

Actividade 3.2 Estruturas de dados

Introdução

O estudo de estruturas de dados é parte fundamental para o desenvolvimento de programas e algoritmos. Assim como um número pode ser representado em vários sistemas diferentes, também um conjunto de dados relacionados entre si pode ser descrito através de várias estruturas de dados distintas. Quando o programador cria um algoritmo para solucionar um problema, ele também cria uma estrutura de dados que é manipulada pelo algoritmo. A escolha de uma determinada estrutura pode afectar substancialmente a quantidade de área de armazenamento requerida para o processamento bem como o tempo deste processamento.

Detalhes da actividade:

Em computação, as estruturas de dados são uma organização ou disposição de dados de forma coerente e estruturada para o processamento. Um tipo estruturado é um exemplo de estrutura de dados já pré definidas na linguagem de programação. O programador pode definir outras estruturas de dados para armazenar as informações que o seu programa precisa de manipular.

As estruturas de dados especificam conceitualmente os dados, de forma a reflectir um relacionamento lógico entre os dados e o domínio do problema a ser considerado. Além disso, elas incluem operações para a manipulação dos seus dados, que também desempenham o papel de caracterização do domínio de problema considerado. É importante notar que o nível conceitual de abstracção não é fornecido directamente pelas linguagens de programação, as quais fornecem os tipos de dados e os operadores que permitem a construção de uma estrutura de dados flexível para o problema que está sendo definido.

As estruturas de dados consistem em:

Listas lineares

Pilhas

Filas

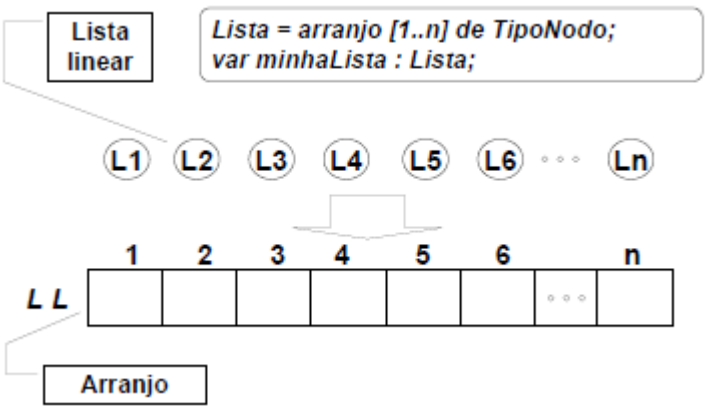
Árvores

Tabelas de dispersão

1.2.1. Listas Lineares

Refere-se a uma colecção de elementos do mesmo tipo denominados nós como por exemplo listas de compras, que contém um primeiro elemento, segundo, terceiro, até ao último. O relacionamento entre os nós de uma lista linear é definido somente por sua posição em relação aos outros nós. Note que a relação linear entre elementos de uma matriz é reflectida por meio de relações físicas de dados na memória, o que torna fácil computar o endereço de um elemento numa matriz por um lado. Por outro lado, as matrizes possuem certas desvantagens, isto é, torna-se mais caro efectuar as operações de inserção e remoção de elementos. A outra forma de guardar elementos na memória, é de possuir para cada elemento um campo denominado ligação ou ponteiro, que contém o endereço do elemento seguinte na lista.

Formalmente uma lista linear pode ser definida como sendo uma colecção de nós todos do mesmo tipo, cujas propriedadades estruturais relevantes envolvem apenas as posições relativas lineares entre os nós.



Exemplo: representação de lista linear

O conjunto de operações a ser definido depende de cada aplicação. Um conjunto de operações necessário a uma maioria de aplicações é:

Criação de uma lista

Inserção de um nó

Remoção de um nó

Localização de um nó

Destruição de uma lista

Combinar duas ou mais listas ligada numa uma lista única.

1.2.2. Listas Lineares implementadas através de contiguidade física

Listas lineares implementadas através de contiguidade física utilizam a sequencialidade da memória do computador para representar a ordem dos nós na lista. Endereços físicos adjacentes na memória representam nós logicamente adjacentes na lista. Deste modo, o relacionamento lógico representado pela posição de um nó na lista não precisa ser explicitamente representado.

A forma mais usual de implementar uma lista linear é através de contiguidade física, isto é através de utilização de uma matriz de uma dimensão (vector). Cada elemento da vetor representa um nó da lista. Qualquer nó da lista pode ser acedido directamente através do índice do vector, que representa a sua posição na lista.

Criação de uma lista

Algoritmo: InicializarLL

entradas: IA (inteiro)

saídas: IL, FL (inteiro)

início

IL FL IA-1

fim

O procedimento seguinte insere um nó em qualquer posição da lista

Para a inserção de um elemento numa lista linear deve-se conhecer :

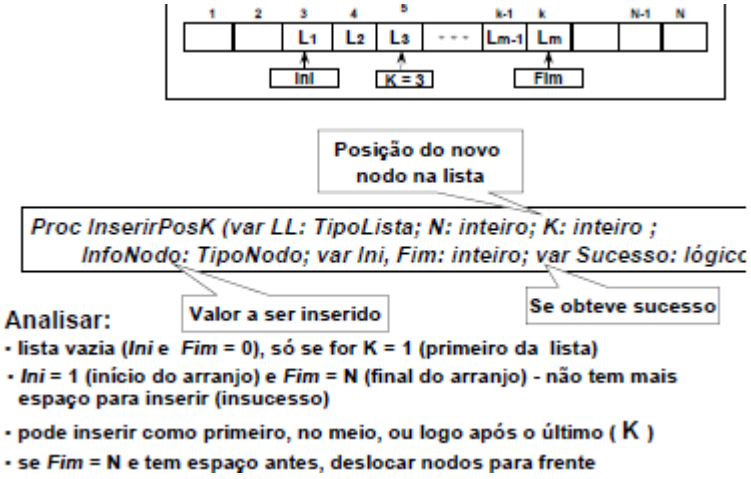
a posição do novo nó na lista

o valor do campo de informação deste nó

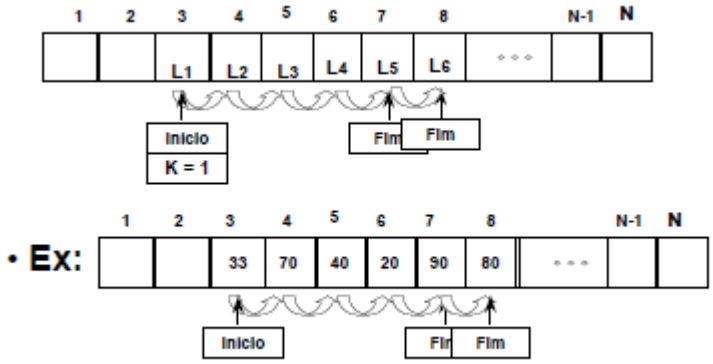
Quando a posição informada não for coerente com a lista actualmente existente, por exemplo a lista tem 8 nós e se pretende inserir o 10º nó

Quando o espaço que a lista pode ocupar no vector estiver totalmente preenchido.

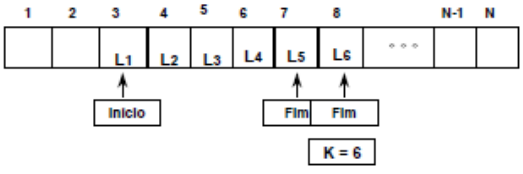
O algoritmo que implementa a operação de inserção deve verificar estas condições e, caso não seja possível realizar a operação, comunicar este facto à aplicação. Em seguida estão descritas as três formas de inserção de um nó.



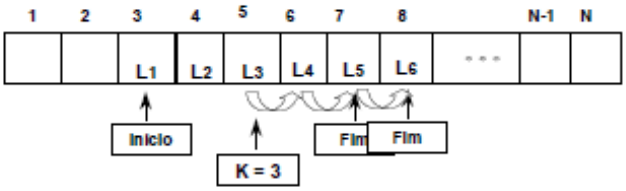
Inserção como primeiro nó na lista



Inserção como último nó na lista



Inserção no meio da lista



Remoção de um nó numa lista linear

A remoção de um nó numa lista linear, requer a identificação em primeiro lugar do nó a ser removido da lista através de alguma informação contida no próprio nó. Para o presente estudo, será considerado o caso em que a identificação do nó a ser removido é feita através da sua ordem na lista isto é será removido o K-ésimo nó, ou seja o nó de ordem k a partir do início da lista. Dado o sequenciamento dos nós, o nó procurado é facilmente localizado assim que for identificada a posição onde inicia a lista, bastando para isso somar a sua ordem ao índice da primeira posição ocupada.

É importante lembrar que a posição do nó procurado não é contada a partir do início do vector mas sim a partir do primeiro nó na lista que pode estar localizado em qualquer posição do vector.

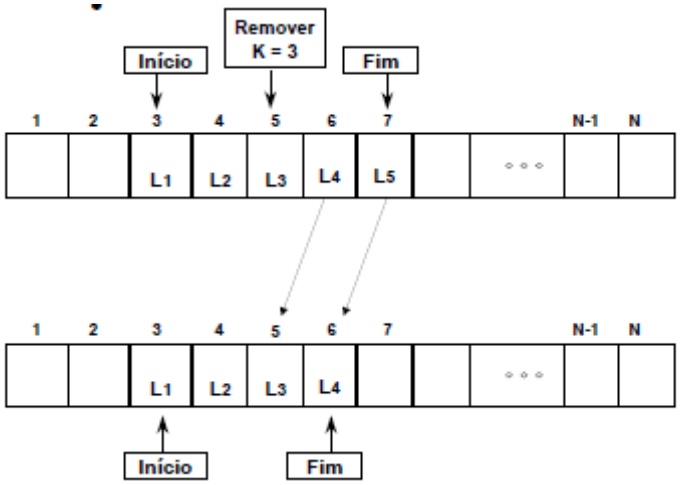
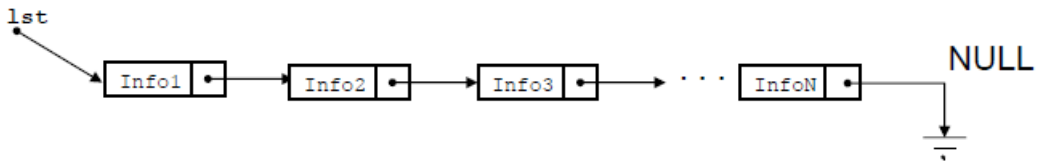


Figura: Remoção de um nó em qualquer posição

1.2.3. Lista Linear Ligada ou encadeada

Uma lista ligada é uma estrutura de dados que pode mudar durante a execução de um programa, isto é, crescer ou diminuir o seu tamanho, de acordo com a demanda e não desperdiça espaço de memória. Os nós que são elementos da lista são representados por uma estrutura que contém, conceitualmente, dois campos: a informação armazenada e o ponteiro para o próximo elemento da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó). A partir deste primeiro elemento, pode-se alcançar o segundo, seguindo o encadeamento, e assim por diante. O último elemento da lista armazena,como próximo elemento, um ponteiro inválido, com valor NULL, e sinaliza assim que não existe um próximo

elemento.



Vantagens

- A inserção e remoção de elementos podem ser feitas sem deslocar os itens seguintes da lista
- Não há necessidade de previsão da quantidade de elementos da lista, espaço necessário é alocado em tempo de execução
- Facilita a gestão de várias listas (fusão, divisão,etc.)

Desvantagens

- Acesso indirecto aos elementos
- Tempo variável para aceder os elementos (depende da posição do elemento)
- Gasto de memória maior pela necessidade de um novo campo para o ponteiro

Listas Lineares em alocação ligada

Na seção anterior foi visto que se pode utilizar uma matriz para representar um conjunto de dados contíguos. A matriz é uma das formas de representação de listas que aproveita a sequencialidade da memória, ou seja, ocupa um espaço contíguo na memória e permite aceder qualquer um de seus elementos. No entanto, não é uma estrutura flexível, pois é necessário fazer uma estimativa do número máximo de nós da lista. Uma forma de implementar as estruturas dinâmicas é através do encadeamento onde os nós são ligados entre si para indicar a ordem existente entre eles.

Criação de uma lista linear ligada

Antes de iniciar uma lista, o ponteiro que indica seu primeiro elemento deve ser inicializado com um endereço nulo, neste caso denomina-se PtLista.

Algoritmo 1.2.2.1

Entradas: -

Saídas: PtLista (tipoPtNó)

início

PtLista ←---- nulo

fim

Inserção de um nó

Para inserir um novo nó numa lista encadeada deve-se em primeiro lugar alocar o novo nó e preenchê-lo.com valor correspondente. Caso não se consiga alocar um novo nó por falta de espaço físico, deve se informar ao usuário. Em seguida, o novo nó deve ser inserido na posição solicitada na lista, o que requer a adequação dos campos de elo dos nós que vão ficar antes e depois deste novo nó: o campo de elo do nó anterior

deverá apontar para o novo nó, e o campo de elo do novo nó deverá conter o endereço do próximo na lista. Nunca haverá necessidade de deslocar nós da sua posição física para efectuar a inserção do novo nó, como no caso de implementação através de alocação sequencial.

A inserção de um nó pode ser feita no início da lista, no fim e no meio da lista.

Inserção no início da lista ligada

Para inserir um nó no início de um lista ligada deve-se após alocar o nó e preenchê-lo com o valor correspondente, apontar seu campo de elo para o endereço daquele que era o primeiro e actualizar o ponteiro de início da lista para o novo nó. Caso a lista esteja vazia, este passará a ser o seu único nó.

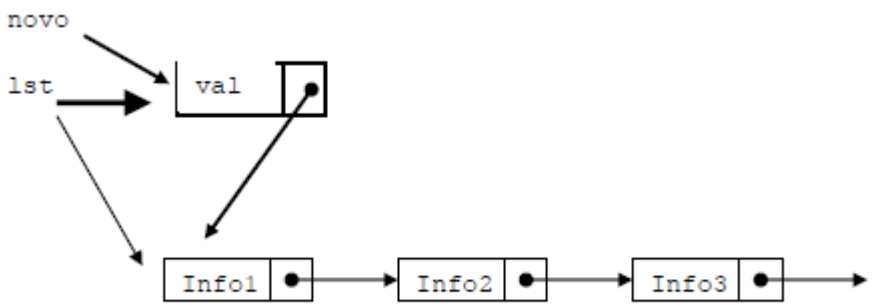


Figura: Inserção de um nó no início da lista

Algoritmo 1.2.2.1: InsertInLista

entrada: PtLista (tipoPtNo)

Dados (tipoInfNo)

saídas: PtLista (tipoPtNo)

sucesso (lógico)

variável auxiliar: Pt (tipoPtNo)

inicio

alocar(Pt)

se Pt=nulo

então sucesso ←falso

senão inicio

sucesso←verdadeiro

PtInfo Dados

PtElo PtLista

PtInfo Pt

fim

fim

Inserção de um nó no final da lista ligada

A inserção de um nó no final da lista requer somente fazer a ligação daquele que era o último nó da lista com o novo nó. Isto é fazer com que o seu campo de elo aponte para o endereço onde foi alocado o novo nó, conforme mostra a figura seguinte. O ponteiro que guarda o início da lista somente será afectado no caso em que a lista era vazia, quando então apontará para este nó.

Algoritmo 1.2.2.2 InsertFimLista

Entrada: PtLista (tipoPtNo)

Dados (tipoInfofNo)

saídas: PtLista (tipoPtNo)

sucesso (lógico)

variáveis auxiliares: P1, P2 (tipoPtNo)

```
inicio
alocar(P1)
        se P1=nulo
        então sucesso ←falso
senão inicio
        sucesso←verdadeiro
        P1Info Dados
P1Elo nulo
se PtLista = nulo
então PtListaP1
senão início
P2PtLista
enquanto P2.Elo nulo
faça P2 P2.Elo
P2.Elo P1
fim
fim
fim
```

Inserção de um nó no meio da lista ligada

Em primeiro lugar, deve-se localizar o elemento da lista que irá preceder o elemento novo a ser inserido, desta feita adequando os campos de elo dos nós anterior e posterior ao novo nó. O novo apontará para o próximo elemento na lista e o elemento precedente apontará para o novo.

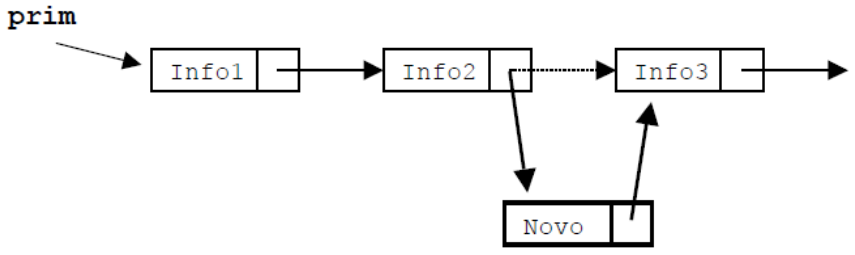


Figura: Inserção de um nó em qualquer posição da lista

Algoritmo 1.2.2.3, InsertQualPosLista

```
Entrada:          PtLista (tipoPtNo)
                  Dados (tipoInfofNo)
                  K (inteiros)
saídas:  PtLista (tipoPtNo)
                  sucesso (lógico)
variáveis auxiliares: PtAnt, PtNovo  (tipoPtNo)
inicio
alocar(Ptnovo)
        se PtNovo=nulo
        então sucesso ←falso
senão  se ((PtNovo=nulo) e (k1)) ou (k<1))
então inicio
libertar (PtNovo)
        sucesso←falso
        fim
senão se k=1
        então início
                PtNovo.InfoDados
                PtNovo.EloPtLista
                PtListaPtNovo
                Sucesso verdadeiro
fim
senão início
        PtAnt PtLista
enquanto  (PtAnt.Elo nulo) e (k>2)
faça   início
PAntPAnt.Elo
kk-1
fim
```

se k> 2

então início

libertar (PtNovo)

Sucesso falso

fim

senão início

ptNovo.InfoDados

ptNovo.EloPtAnt.Elo

PtAnt.EloPtNovo

sucesso verdadeiro

fim

fim

fim

Remoção de nós de uma lista ligada

A remoção de um nó de uma lista ligada é feita simplesmente mudando a ligação dos nós anterior (PtAnt) e posterior (PtPost) ao nó a ser removido: o nó imediatamente anterior apontará para aquele que era o seguinte do nodo excluído da lista. Caso o nó libertado seja o primeiro, o endereço do segundo deverá ser copiado para o ponteiro do início da lista. Caso seja o último, o anterior deverá ficar com o campo do elo nulo. Após esta ligação, que garante a continuidade da lista, a posição ocupada pelo nó removido será libertada. Para implementar esta operação, a lista deve ser percorrida a partir do seu primeiro nó, indicado pelo ponteiro PtLista, com a finalidade de localizar o nó a ser removido.

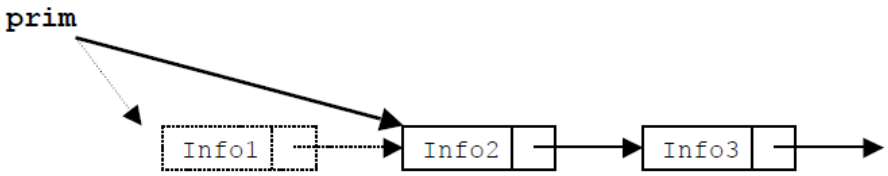


Figura: remoção do primeiro nó da lista ligada

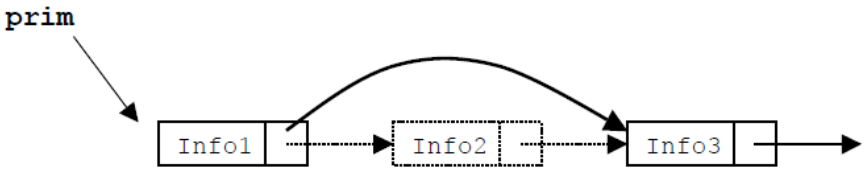


Figura: Remoção de um nó em qualquer posição na lista

Algoritmo : remover QualPosLista

Entrada: PtLista (tipoPtNo)

Dados (tipoInfofNo)

K (inteiros)

saídas: PtLista (tipoPtNo)

sucesso (lógico)

variáveis auxiliares: PtAnt, PtK (tipoPtNo)

inicio

se k<1

então sucesso ←falso

senão início

PtK ←PtLista

PtAnt ←nulo

enquanto ((PtKnulo) e (K>1)

faça início

K←K-1

PtAnt←PtK

PtK←PtK.Elo

fim

se PtK= nulo

então sucesso falso

senão início

```
        se PtK=PtLista
        então   PtListaPtLista.Elo
senão  PtAnt.Elo PtK.Elo
libertar (PtK)
Sucesso verdadeiro
fim
fim
fim
```

Localização de um nó numa lista ligada

O acesso a um determinado nó de uma lista ligada requer que a lista seja percorrida, a partir de seu primeiro nó, até ao nó procurado. Este nó pode ser identificado através de alguma informação nele contida, ou pela sua ordem na lista. Diferentemente do caso de alocação de uma lista sobre um vector, não existe, no caso de lista ligada, a possibilidade de acessar directamente algum nó, sem que ele seja alcançado percorrendo a lista a partir de seu primeiro nó. O algoritmo seguinte localiza um determinado nó de uma lista ligada, identificando-o pela sua ordem na lista (K), devolvendo o endereço físico deste nó (PtK) ao programa que o accionou. Caso a posição do nó não for compatível com o tamanho da lista, ou alista esteja vazia, PtK retorna o endereço nulo.

```
Algoritmo: Aceder  QualquerNo
Entrada:        PtLista (tipoPtNo)
                K (inteiros)
saída:   PtK (tipoPtNo)

inicio
se  (K<1) ou (PtLista=nulo)
então PtK←nulo
senão inicio
        PtK←PtLista
        enquanto (PtK ) e (K>1)
faça  início
```

K K-1

PtKPtK.Elo

fim

se K>1

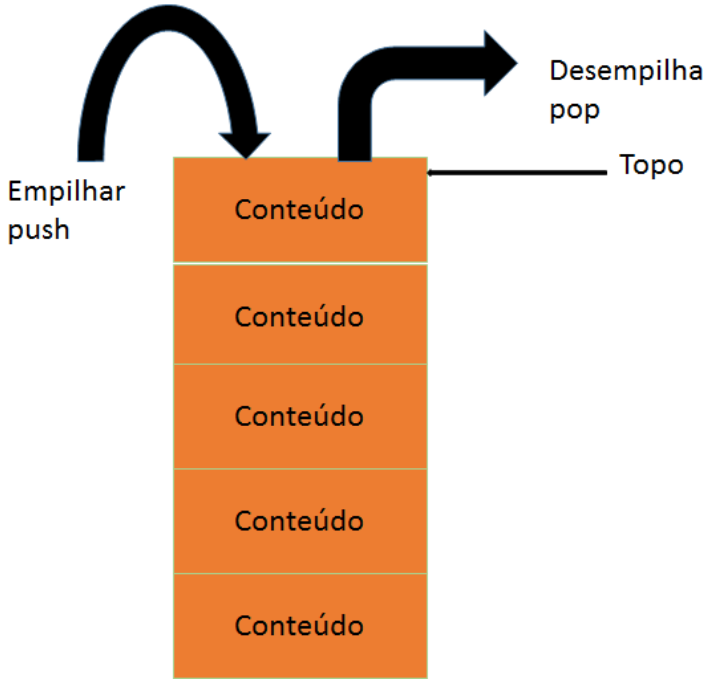
então PtK

fim

fim

1.2.4. Pilhas

É uma lista linear em que todas as inserções, retiradas e, geralmente, todos os acessos são feitos em apenas um extremo da lista. Os elementos são colocados um sobre o outro. O item inserido mais recentemente está no topo e o inserido menos recentemente no fundo. O modelo intuitivo é o de um monte de pratos numa prateleira, sendo conveniente retirar ou adicionar pratos na parte superior.



Propriedades e Aplicações das Pilhas

O último item inserido é o primeiro item que pode ser retirado da lista. São chamadas listas lifo ("last-in, first-out") e existe uma ordem linear para pilhas, do "mais recente para o menos recente", ideal para processamento de estruturas aninhadas de profundidade imprevisível. Uma pilha contém uma sequência de obrigações adiadas. A ordem de remoção garante que as estruturas mais internas serão processadas antes das mais externas.

estrutura em pilha tem os seguintes métodos ou funções:

push - coloca uma informação na pilha (empilha)

pop - retira uma informação da pilha (desempilha)

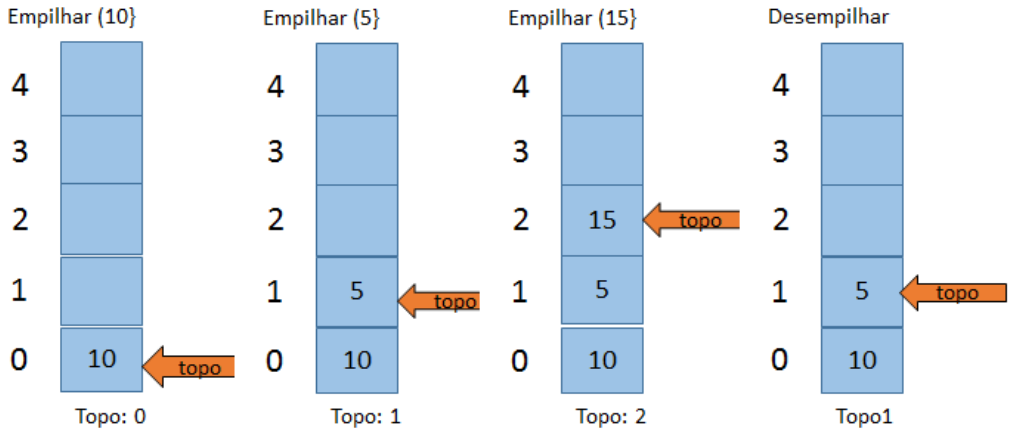
size - retorna o tamanho da pilha

stackpop - retorna o elemento superior da pilha sem removê-lo (equivalente às operações de pop e um push)

empty - verifica se a pilha está vazia.

Há situações em que a pilha se encontra cheia ou vazia. Quando a pilha estiver cheia e, houver algum elemento para inserir, então denomina-se overflow. E se estiver vazia e, houver algum item por remover, denomina-se underflow.

Exemplo: operação de empilhar e desempilhar numa pilha



Nota: As pilhas podem ser implementadas através de contiguidade física ou através de encadeamento.

Implementação de Pilhas em Listas Ligadas

```
void criapilha (void) {
   pi = mallocc (sizeof (celula)); // cabeça
   pi->prox = NULL;
}

void empilha (char y) {
   celula *nova;
   nova = mallocc (sizeof (celula));
   nova->conteudo = y;
   nova->prox  = pi->prox;
   pi->prox = nova;
```

```
}

char desempilha (void) {
   char x;
   celula *p;
   p = pi->prox;
   x = p->conteudo;
   pi->prox = p->prox;
   free (p);
   return x;
```

1.2.5. Fila

Uma fila é um conjunto ordenado de itens a partir do qual se podem remover elementos

numa extremidade, denominado início da fila e ao qual se pode adicionar elementos na outra extremidade chamada final da fila. Nas pilhas existe o princípio de que o primeiro que entra, é o primeiro que sai - first in, first out (FIFO).

O conceito de fila existe no mundo real, os exemplos de filas de ATM, restaurantes , etc.

Propriedades e Aplicações das Filas

As operações básicas de uma fila são:

- insert ou enqueue - insere elementos no final duma fila,
- remove ou dequeue são usadas para retirar os itens de uma fila (primeiro elemento)
- empty - verifica se a fila está vazia
- size - retorna o tamanho da fila.
- front - retorna o próximo nó da fila sem retirar o mesmo da fila

A operação insert ou enqueue sempre pode ser executada, uma vez que teoricamente uma fila não tem limite. A operação remove ou dequeue só pode ser aplicado se a fila não estiver vazia, causando um erro de underlow ou fila vazia se esta operação for realizada nesta situação.

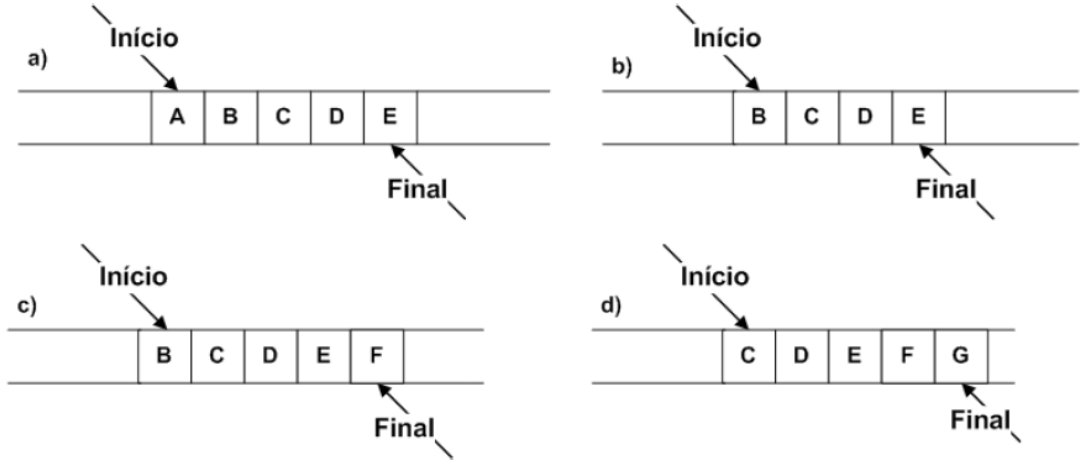
Exemplo de operações numa fila

dequeue() - Retorna o item A (a fila resultante é representada pelo item B)

enqueue(F) - O item F é armazenado ao final da ila (a fila resultante é representada pelo item C)

dequeue() - Retirado o item B da fila

enqueue(G) - Colocado o item G ao final da fila (item D)



Implementação de Filas com uso de Matrizes

Algoritmo Fila

var

varTipo fila_reg = registo

fila_reg = reginicio: inteiro

Tipo fila_reg = fim: inteiro

fila_reg =elemento: vetor [1.50] de inteiro

_reg = fim

vartotal: inteiro

varfila: fila_reg

inicio

varfila.inicio R0

varfila.fim R0

vartotal R0

Função vazia ( ): lógica

varinicio

varvaSe(total = 0) entao

varvaSe(return .v.

varvaSenão

varvaSe(return .f.

varvafim-se

```
varfim
Função cheia ( ): lógica
varinicio
varvaSe(total >=50) então
varvaSe(return .v.
varvaSenão
varvaSe(return .f.
varvafim-se
varfim
Procedimento enfileirar (elem: inteiro)
varinicio
varvaSe(cheia ( ) = .f..) então
arvaSe( fila.elemento[inicio] R elem
arvaSe( fila.fim R fila.fim + 1
arvaSe( total R total + 1
varvaSe(fila.fim >= 50) então
varvaSe( fila.fim = 0
varvafim-se
varva Senão
varva enMostre(“Fila cheia”)
var vafim-se
v rfim
Funcao desenfileirar ( ): inteiro
varvar
varv excluido: inteiro
varinicio
varva Se (vazia ( ) = .f.) então
arva enexcluido R fila.elemento[inicio]
arva enfila.inicio R fila.inicio + 1
arva enSe (fila.inicio > = tamanho) então
```

```
arva enfila.inicio R 0
arv a fim-se
arva total R total – 1 retorne excluido
varva Senão
varva excluído R nulo
varva retorne excluído
varva fim-se
varfim
Procedimento exibefila ( )
varvar
Procedi: inteiro
varinicio
Proced Para (i R 0 até total) faça
Proced ParMostre (“Posição ”, i, ” valor ”, elemento [i] )
Proced fim-para
varfim
fim
Função vazia ( ): lógica
varinicio
varvaSe(total = 0) entao
varvaSe(return .v.
varvaSenão
varvaSe(return .f.
 varvafim-se
varfim
Função cheia ( ): lógica
varinicio
varvaSe(total >=50) então
varvaSe(return .v.
varvaSenão
```

varvaSe(return .f.

varvafim-se

varfim

Procedimento enfileirar (elem: inteiro)

varinicio

varvaSe(cheia ( ) = .f..) então

arvaSe(fila.elemento[inicio] R elem

arvaSe(fila.fim R fila.fim + 1

arvaSe(total R total + 1

varvaSe(fila.fim >= 50) então

varvaSe(fila.fim = 0

varvafim-se

varva Senão

varva enMostre("Fila cheia")

var vafim-se

v rfim

Funcao desenfileirar ( ): inteiro

varvar

varv excluido: inteiro

varinicio

varva Se (vazia ( ) = .f.) então

arva enexcluido fila.elemento[inicio]

arva enfila.inicio fila.inicio + 1

arva enSe (fila.inicio > = tamanho) então

arva enfila.inicio 0

arv a fim-se

arva total total – 1 retorne excluido

varva Senão

varva excluído nulo

varva retorne excluído

```
varva fim-se

varfim

Procedimento exibefila ( )

varvar

Procedi: inteiro

varinicio

Proced Para (i 0 até total) faça

Proced ParMostre (“Posição ”, i, ” valor ”, elemento [i] )

Proced fim-para

varfim

fim
```

Nota: As filas podem ser implementadas através de contiguidade física ou através de encadeamento.

1.2.6. Tabela de dispersão

As tabelas de dispersão (hash tables) são estruturas de dados que associam chaves a valores. A cada chave é transformada pela função de dispersão num número que é utilizado para indexar num vector para encontrar os valores associados aquela chave. As tabelas de dispersão utilizam tabela base de indexação, funções de dispersão (hash functions) e algoritmos para resolução de colisões (se necessário).

Tabela Base

Existem dois tipos de dispersão nomeadamente, dispersão com índices livres, este tipo ocorre quando o número de elementos pode ser estimado à partida em N, neste caso para a tabela de M elementos (M > N). E o outro tipo denomina-se dispersão por separação em listas, quando o número de elementos a guardar (N) é desconhecido, portanto a tabela de M listas de elementos (M < N)

Função de Dispersão

Uma função de dispersão transforma a chave num inteiro [0;M-1] (M tamanho do vector); e deve distribuir as chaves de forma uniforme e quase aleatória, ser capaz de fazer cálculos rápidos Para diferentes funções devem ser usadas para diferentes tipos de dados.

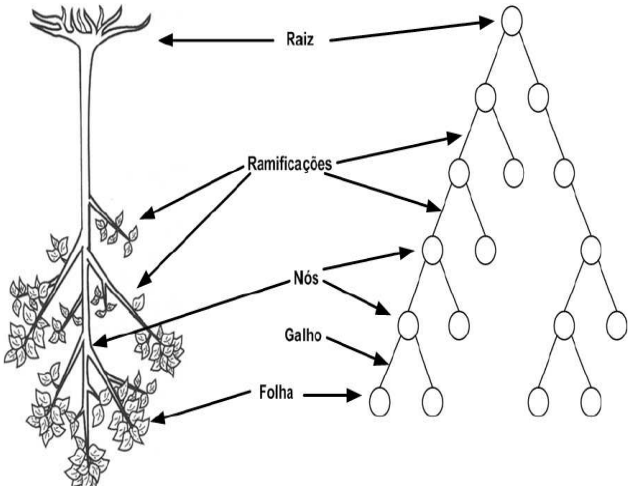
Definição: Colisão - ocorre quando a função de dispersão devolve o mesmo valor para chaves distintas

1.2.7. Árvores

No contexto da programação e ciência da computação, é definida como sendo uma estrutura de dados que herda as características das topologias em árvore onde os dados estão dispostos de forma hierárquica (um conjunto de dados é hierarquicamente subordinado a outro.

Árvore é uma estrutura de dados não linear adequada para representar hierarquias. A forma mais natural de definir uma estrutura de árvore é usanda recursividade. Uma árvore é composta por um conjunto finito de nós. Desse conjunto, há um nó r denominado de raiz, que contém zero ou mais sub-árvores, cujas raízes são ligadas diretamente a r. Esses nós raízes das sub-árvores são chamados filhos do nó pai, r. Nós com filhos são denominados nós internos e os que não os têm são chamados nós externos ou folhas.

Analogia com árvores:



Terminologia

A terminologia utilizada para referenciar conceitos envolvidos nas estruturas de dados denominadas árvores, não é padronizada, sendo utilizados nomes diferentes para os mesmos conceitos em diferentes publicações. Em seguida a terminologia utilizada neste curso:

Raiz: é um nó diferenciado, presente em todas as árvores, ao qual são subordinados todos os outros nós da árvore. O acesso é feito a partir de sua raiz.

Nós descendente: são os nós que apresentam alguma relação de dependência com um nó mais acima na hierarquia representada pela árvore. Usualmente para indicar os graus de dependência entre os nós de uma árvore é o chamar descendentes directos de um nó de filhos ou sucessores deste nó, e o nó em questão, de pai ou mãe, ascendente ou ainda antecessor de seus descendentes directos. Assim, todos os descendentes directos de um nó são designados irmãos.

Subárvore: É um conjunto de nós, sendo todos eles subordinados a um único nó, externo a esta subárvore.

Grau de um nó: É número de subárvores que são subordinadas directamente a este nó, ou seja à quantidade de subárvores para as quais este nó é raiz.

Grau de uma árvore: É o maior valor dentre os graus de todos os seus nós.

Folha ou terminal (externo): Os nós de grau zero (que não tem descendentes) são chamados folhas ou terminais.

Nó de derivação (interno): Os nós de grau maior do que zero e que, consequentemente, apresentam alguma subárvore, são denominados nós de derivação ou nós internos.

Nível de um nó: Corresponde ao número de ligações entre este nó e a raiz da árvore, acrescido de uma unidade. A raiz de uma árvore tem sempre nível 1.

Caminho: Consiste numa sequência de nós consecutivos distintos entre dois nós.

Comprimento do caminho: Dado um determinado caminho entre dois nós, o comprimento deste caminho é determinado pelo número de níveis entre estes dois nós, diminuído de uma unidade.

Altura ou profundidade: É o número de nós do maior caminho deste nó até um dos descendentes- folha. A altura ou profundidade da árvore é igual ao maior nível de seus nós. Todos os nós folha possuem altura 1.

Floresta: É formada por um conjunto de zero ou mais árvores disjuntas.

Árvore ordenada: Uma árvore denomina-se ordenada quando a ordem de suas subárvores é relevante para a aplicação que está sendo representada através desta estrutura de dados.

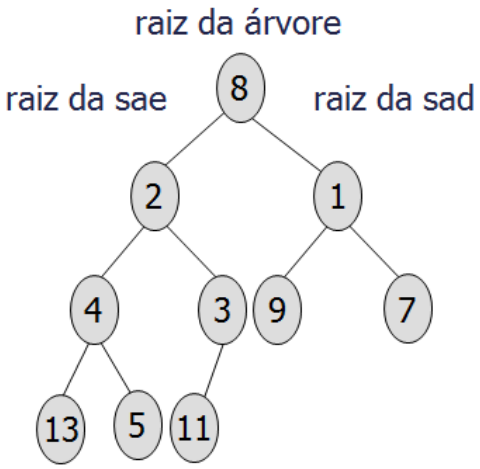
Árvore cheia: Uma árvore denomina-se cheia se possui o número máximo de nós, isto é, todos os nós tem número máximo de filhos excepto as folhas, e todas as folhas estão na mesma altura.

Árvores isomorfas: Duas árvore não ordenadas são isomorfas quando é possível que se tornem coincidentes através de uma permutação na ordem das subárvores de seus nós.

Árvores balanceada: é aquela na qual existe uma distribuição equilibrada entre os nós da árvore, ou seja existe uma diferença mínima entre todas as folhas e a raiz.

Árvore binária: Uma árvore é dita binária quando apresenta no máximo grau 2 em cada nó. De modo semelhante, uma árvore é denominada n-ária quando apresenta no máximo grau "n" em cada nó.

Árvores Binárias: Uma árvore binária é constituída por um conjunto finito de nós. Cada nó pode ter no máximo dois filhos. De maneira recursiva, pode-se definir uma árvore binária como sendo uma árvore vazia ou um nó raiz tendo duas sub-árvores, identificadas como a sub-árvore da direita (sad) e a sub-árvore da esquerda (sae).



Operações sobre árvores

Com base na terminologia apresentada, uma árvore é definida formalmente como sendo um conjunto finito T de zero ou mais nós, podendo ocorrer duas situações:

- Caso o número de nós seja diferente de zero.
- existe sempre um nó denominado raiz da árvore;
- os demais nós formam m>0 conjuntos disjuntos S1, S2, … , Sn, onde cada um destes conjuntos é, por sua vez, uma árvore, sendo as árvores Si desnominadas subárvores.
- Caso o número de nós da árvore seja zero, a árvore é denominada vazia.

As principais operações sobre árvores são:

- Criação da árvore
- Inserção de um novo nó em uma determinada posição
- Exclusão de um nó

- Acesso a um nó
- destruição de uma árvore

Outras operações:

- pai: dado um determinado nó de uma árvore, esta operação retorna o endereço do nó imediatamente superior a ele na hierarquia
- tamanho: operação que retorna o número total de nós de uma árvore
- altura: retorna a altura da árvore em estudo

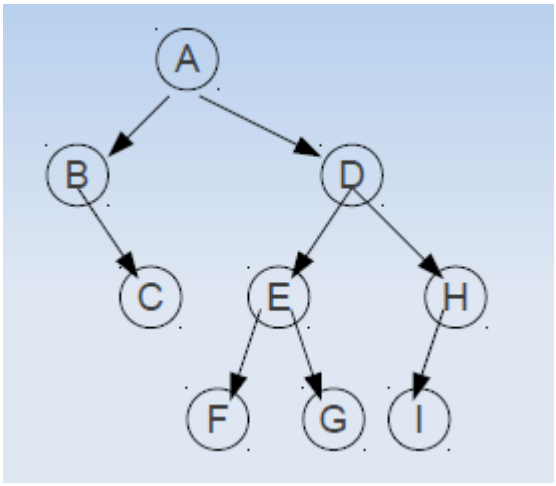
Para além destas operações, é possível fazer outras operações denominadas de percurso em árvores binárias. É comum percorrer uma árvore numa das seguintes ordens:

Pré-Ordem: Imprime os valores presentes nos nós da árvore abaixo, segundo a condição pré-ordem (tratar raiz, percorrer sae, percorrer sad).

Em-Ordem (ordem simétrica): Imprime os valores presentes nos nós da árvore, segundo a condição ordem simétrica (percorrer sae, tratar raiz, percorrer sad).

Pós-Ordem: Imprime os valores presentes nos nós da árvore, segundo a condição pós-ordem (percorrer sae, percorrer sad, tratar raiz).

Exemplo:



Pré-ordem: A,B,C,D,E,F,G,H,I

Ordem: B,C,A,F,E,G,D,I,H

Pós-ordem: C,B,F,G,E,I,H,D,A

Largura: A,B,D,C,E,H,F,G,I

Conclusão

As estruturas de dados são formas de distribuir e relacionar os dados disponíveis, de modo a tornar mais eficientes os algoritmos que manipulam esses dados. O estudo de estrutura de dados é parte fundamental para o desenvolvimento de programas e algoritmos. Assim como um número pode ser representado em vários sistemas diferentes, também um conjunto de dados relacionados entre si pode ser descrito através de várias estruturas de dados distintas.

Quando o programador cria um algoritmo para solucionar um problema, ele também cria uma estrutura de dados que é manipulada pelo algoritmo. A escolha de uma determinada estrutura pode afetar substancialmente a quantidade de área de armazenamento requerida para o processamento bem como o tempo deste processamento. Portanto, é de grande importância o estudo de diferentes estruturas que possam ser utilizadas na resolução de um problema, de forma a simplificar a sua implementação prática.

Avaliação

1. Escreva um algoritmo para implementar a inserção e remoção de um elemento na 3ª posição de uma lista encadeada.
2. Escreva um algoritmo para remoção e inserção de um nó em pilhas.
3. Explique como são realizadas as operações em árvores binárias. Exemplifique.
4. Qual é a aplicação de tabelas de dispersão

Actividade 3.3 –Tipo Abstracto de Dados

Introdução

Um tipo abstrato de dados (TAD) pode ser visto como um modelo matemático que encapsula um modelo de dados e um conjunto de procedimentos que actuam com exclusividade sobre os dados encapsulados. Em nível de abstração mais baixo, associado à implementação, esses procedimentos são implementados por subprogramas denominados operações, métodos ou procedimentos.

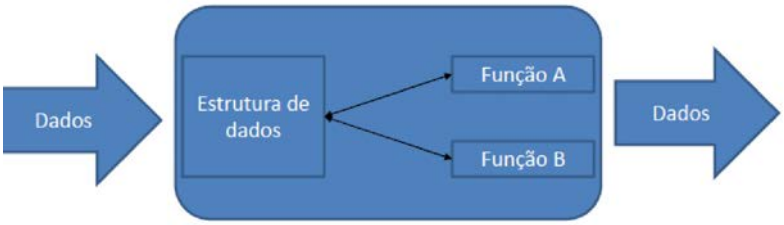
Detalhes da actividade

Qualquer processamento a ser realizada sobre os dados encapsulados num TAD só poderá ser executada por intermédio dos procedimentos definidos no modelo matemático do TAD, sendo esta restrição a característica operacional mais útil dessa estrutura.

Nesses casos, um programa baseado em TAD deverá contêr algoritmos e estruturas de dados que implementam, em termos da linguagem de programação adoptada, os procedimentos e os modelos de dados dos TADs utilizados pelo programa.

Assim, a implementação de cada TAD pode ocupar porções bem definidas do programa: uma para a definição das estruturas de dados e outra para a definição do conjunto de algoritmos.

Nessas condições, quaisquer alterações realizadas na estrutura de um dado TAD não afectarão as partes do programa que utilizam esse TAD.



Características dos TAD´s

A característica essencial de uma TAD é a separação entre conceito e implementação, ou seja existe uma distinção entre a distinção do tipo e a sua representação, e a implementação das operações. Um TAD é uma forma de definir um novo tipo de dados juntamente com as operações que manipulam esse novo tipo de dado. As aplicações que utilizam TAD são denominadas clientes do tipo de dado.

Formalmente define.se a TAD como: Definido como um modelo matemático por meio de um par (V,O) em que V é um conjunto de valores e O é um conjunto de operações sobre esses valores

Exemplo: Conjunto de números reais

V= R

O= {+, -, *, /, =, <, >, <=, >=}

Exemplo TAD data

Data=registo

Dia: inteiro

Mês: inteiro

Ano: inteiro

fim registo.

Vantagens de utilização de TAD:

Reutiização : uma vez definido, implementado e testado, o TAD pode ser acdido por diferentes programas

Manutenção: mudanças na implementação do TAD não põem em causa o código fonte dos

programas que o utilizam (decorrência do ocultamento de informação), isto é, os módulos do TAD são compilados separadamente, uma alteração força somente a recompilação do arquivo envolvido e uma nova link-edição do programa que aceda o TAD e, o programa mesmo não precisa ser recompilado!

Correção: TAD foi testado e funciona correctamente.

Tipo Abstracto de Dados (TAD) que representa árvores binárias

Há duas formas de definir um tipo abstracto de dados do tipo árvore, através de implementação por contiguidade física e por encadeamento.

Conclusão

Tipo Abstracto de dados (TAD) é uma estrutura de programação que visa implementar o domínio de um modelo matemático de dados um conjunto de operações básicas que actuam com exclusividade sobre os valores desse domínio. Qualquer operação a ser realizada sobre dados definidos por meio dessa estrutura só poderá ser executada através dos algoritmos definidos no TAD. Um TAD deverá ocupar uma porção bem definida do programa, a qual conterá uma estrutura com dois componentes nomeadamente um dos componentes está associado à definição das estruturas de dados que implementam a representação dos valores que constituem o domínio do modelo matemático de dados considerado e outro, associado à definição do conjunto de algoritmos que implementam as operações que actuam com exclusividade sobre essas estruturas de dados.

Nessas condições, quaisquer alterações realizadas na implementação da estrutura de um dado TAD não afectarão as partes do programa que utilizam esse TAD.

Avaliação

1. Faça uma comparação entre TAD e Estruturas de Dados.

Resumo da Unidade

Nesta unidade o aluno aprendeu diferentes estruturas de dados, tais como vectores, listas ligadas, pilhas, filas, tabelas de dispersão e árvores. As formas de armazenar fisicamente as estruturas vistas também foram abordadas como por exemplo alocação sequêncial que é estática e a alocação encadeada que é um tipo de alocação dinâmica.

Estruturas de dados e algoritmos estão intimamente ligados, não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas assim como, a escolha dos algoritmos em geral depende da representação e da estrutura dos dados. É inquestionável a importância de estruturas unidimensionais ou lineares (vetores e listas), porém, estas estruturas

não são adequadas para representar dados que devem ser dispostos de maneira hierárquica, daí a introdução de outro tipo de dados árvore capaz de representar dados de forma hierárquica.

As TADs são utilizadas extensivamente como base para o projeto de algoritmos. A implementação do algoritmo em uma linguagem de programação específica exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.

Avaliação da Unidade

Verifique a sua compreensão!

Avaliação 3:

Instruções

Chegados ao final desta unidade, é tempo de verificar se percebemos o que foi abordado aqui, respondendo as seguintes questões com clareza.

Avaliação

1. O que entende por Tipo de Dados Abstractos?
2. Faça um quadro comparativo entre vectores e listas ligadas.
3. Refira-se às vantagens e desvantagens de implementação de filas e pilhas por contiguidade física.
4. Escreva um algoritmo para implementar a inserção e remoção de um elemento na 3ª posição de uma lista encadeada.
5. Escreva um algoritmo para remoção e inserção de um nó em pilhas.
6. Escreva um algoritmo para remoção e inserção de um nó em filas.
7. Explique como são realizadas as operações em árvores binárias. Dê exemplos.
8. Qual é a aplicação de tabelas de dispersão

Critérios de Avaliação

Pergunta	Pontuação (Maximo 20 valorers)
1	1.5
2	2

3	2.5
4	3
5	3
6	3.5
7	2.5
8	2

Respostas:

1. O que entende por Tipo de Dados Abstratos?

   R: um tipo abstrato de dados (TAD) pode ser visto como um modelo matemático que encapsula um modelo de dados e um conjunto de procedimentos que actuam com exclusividade sobre os dados encapsulados. Em nível de abstração mais baixo, associado à implementação, esses procedimentos são implementados por subprogramas denominados operações, métodos ou serviços.

2. Faça um quadro comparativo entre vectores e listas ligadas.

Vectores	Listas Ligadas
estruturas de dados lineares e estáticas, compostas por um número fixo (finito) de elementos de um determinado tipo de dados estrutura recomendada para casos em que os dados armazenados não mudarão, ou pouco mudarão, através do tempo.	estrutura de dados abstrata que implementa uma colecção ordenada de valores, o mesmo valor pode ocorrer mais de uma vez. Uma instância de uma lista é uma representação computacional do conceito matemático de uma sequência finita, que é, uma tupla.

3. Refira-se às vantagens e desvantagens de implementação de filas e pilhas por contiguidade física.
4. Escreva um algoritmo para implementar a inserção e remoção de um elemento na 3ª posição de uma lista encadeada.
5. Escreva um algoritmo para remoção e inserção de um nó em pilhas.
6. Escreva um algoritmo para remoção e inserção de um nó em filas.
7. Explique como são realizadas as operações em árvores binárias. Dê exemplos. Qual é a aplicação de tabelas de dispersão

Leituras e outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso

- http://www.inf.puc-rio.br
- http://homepages.dcc.ufmg.br
- http://www.caelum.com.br

Unidade 4. Algoritmos de Ordenação e de Pesquisa

Introdução à Unidade

Bancos de dados existem para que, de tempos em tempos, um usuário possa localizar o dado de um registo simplesmente digitando sua chave. Há apenas um método para se encontrar informações em um arquivo (matriz) desordenado e um outro para um arquivo (matriz) ordenado. Neste capítulo, o aluno irá aprender as técnicas utilizadas em Informática para pesquisar dados e ainda ordenar elementos num determinado arquivo.

Ordenação e Pesquisa são operações fundamentais na ciência de computação. Ordenação refere-se a operação de rearranjar os dados em uma determinada ordem, enquanto que pesquisa é a operação de procurar a localização de um item numa colecção de dados.

Objectivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Descrever os processos de ordenação e pesquisa
- Escrever algoritmos para ordenação de items num dado conjunto
- Escrever algoritmos para pesquisar um certo item
- Descrever e implementar os algoritmos de ordenação e de pesquisa de dados

Termos-chave

Ordenação: é o processo de arranjar um conjunto de informações semelhantes em uma ordem crescente ou decrescente (numérica ou lexicográfica).

Pesquisa: é a operação de procurar a localização de um item num conjunto de dados

Actividades de Aprendizagem

Actividade 1.1 - Algoritmo de Ordenação

Introdução

O presente capítulo aborda vários algoritmos padrões adequados para triagem em uma colecção de dados que cabem na memória principal do computador. O aluno irá aprender os algoritmos de ordenação que na essência é o processo de ordenar os elementos de um dado conjunto em ordem crescente ou decrescente (numérica ou lexicográfica).

Detalhes da actividade

O algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de um dado conjunto em uma certa ordem quer dizer, efectua a sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica.

Existem vários algoritmos de ordenação, os principais são:

Algoritmo de Ordenação por Bolha (bubble sort): é um algoritmo de ordenação dos mais simples. A idéia básica por trás da ordenação por bolha é percorrer o arquivo sequencialmente várias vezes. Cada passagem consiste em comparar cada elemento no arquivo com seu sucessor (x[i] com x[i + 1]) e trocar os dois elementos se eles não estiverem na ordem correta.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vector.

No pior caso, são feitas n^2 operações. A complexidade desse algoritmo é de Ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operam com quantidade elevada de dados.

Exemplo: Tabela 1 que ilustra o método numa primeira varredura

troca	**L[1]**	**L[2]**	**L[3]**	**L[4]**	**L[5]**
1 com 2	10	9	7	13	5
2 com 3	9	**10**	7	13	5
4 com 5	9	7	**10**	13	5
fim da varredura	9	7	10	5	**13**

Após a primeira varredura ilustrada na tabela 1, o maior elemento encontra-se alocado em sua posição definitiva na lista ordenada. Logo, a ordenação pode continuar no restante da lista sem considerar o último elemento.

Tabela 2: Segunda varredura do método

troca	L[1]	L[2]	L[3]	L[4]	L[5]
troca 1 com 2	9	7	10	5	13
troca 3 com 4	7	**9**	10	5	13
fim da varredura	7	9	5	**10**	13

Na segunda varredura, o segundo maior elemento encontra-se na sua posição definitiva e o restante da ordenação é realizada considerando apenas os últimos elementos (7, 9 e 5). Logo são necessárias elementos - 1 varreduras, pois cada varredura leva um elemento para sua posição definitiva.

Algoritmo 4.1: Bubble Sort

```
para i=1,... n faça
        para j=1,...n-1 faça
                se L[j].chve > l[j+1].chave então
                        trocar (L[j], L[j+1])
```

Este algoritmo não é muito bom, uma vez que a sua complexidade de pior caso é igual a de melhor caso, O(n2), devido a percursos estipulados para i e j. Existem outros critérios de parada que levariam em consideração comparações desnecessárias, isto é, comparações executadas em partes da tabela sabidamente já ordenadas.

O primeiro critério de parada resulta de que se a tabela estiver ordenada, sendo assim não há necessidade de executar o algoritmo. Introduz-se uma variável lógica alterou é introduzida com o fim de sinalizarse pelo menos uma troca foi realizada. Caso isso não ocorra então o algoritmo pode ser encerrado. Essa operação afecta a complexidade do melhor caso, que diminui para O(n), uma vez que se a tabela já foi ordenada, um percurso apenas é necessário.

O segundo critério decorre do próprio método: como, a cada passo, um elemento é garantidamente transportado para o fim da tabela, o número de elementos da mesma pode ser diminuído. Isto é, a posição da última troca (armazenada, no algoritmo, na variável guarda) indica que todos os elementos posteriores já estão ordenados. O algoritmo então pode utilizar esta posição para actualizar o limite superior da tabela, que inicialmente é o próprio número de elementos.

$$
\begin{aligned}
&trocou := V;\ n' := n;\ guarda := n \blacksquare \\
&enquanto\ trocou\ faça \\
&j := 1 : trocou := F \\
&enquanto\ j < n'\ faça \\
&se\ L[j].chve\ >\ l[j+1].chave\ então \\
&trocar\ (L[j], L[j+1]) \\
&trocou := V \\
&guarda := j \\
&j := j + 1 \\
&n' := guarda
\end{aligned}
$$

Exemplo de aplicação do algoritmo

Iteração	Tabela	Trocas
Tabela dada	9 7 10 5 13	
após 1ª iteração	7 9 5 10 13	2
após 2ª iteração	7 5 9 10 13	1
após 3ª iteração	5 7 9 10 13	1

Algoritmo de Ordenação rápida (Quicksort): é um método de ordenação muito rápido e eficiente. O algoritmo QuickSort é do tipo divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves «menores» precedam as chaves «maiores». Em seguida o Quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. Dada uma tabela L com n elementos, o procedimento recursivo para ordenar L consiste nos seguintes passos:

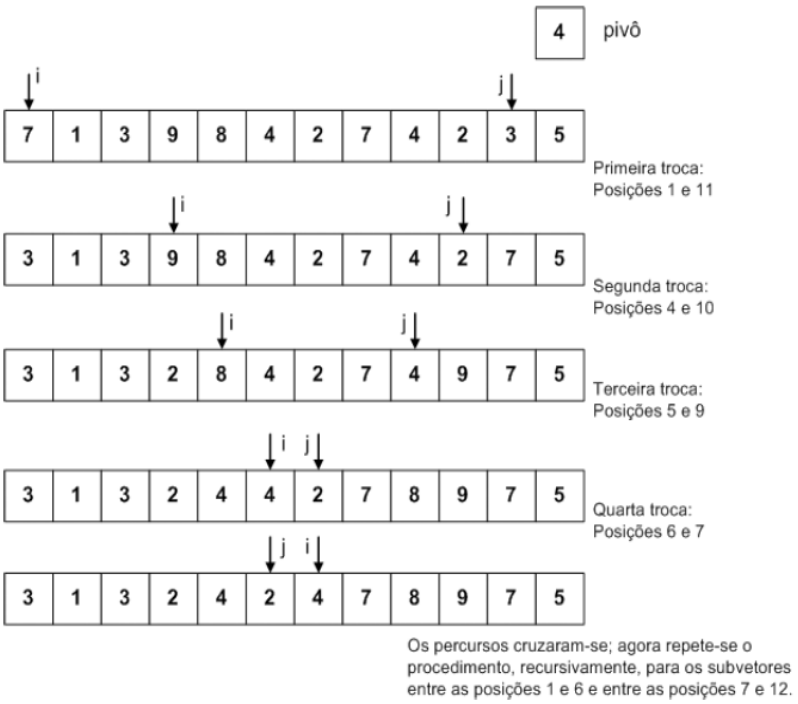
então a tabela está ordenada

escolha qualquer elemento x em L e denomina-se pivô

separe L - {x} em dois conjuntos de elementos disjuntos: e , o procedimento de ordenação é chamado de recursivo para e

recebe a concatenação de seguido de

Exemplo de aplicação



Algoritmo: quicksort(ini, fim)

```
se fim − ini < 2 então
    se fim − ini = 1 então
        se L[ini].key > L[fim].key então
            trocar (L[j], L[j + 1]);
senão PIVO (ini, fim, mediana)
        trocar (L[mediana], L[fim])
        i: = ini; j: = fim − 1
        key: = L[fim].chave
                                    enquanto j ≥ i faça
                enquanto L[i].chave < key ) faça
                    i: = i + 1
                enquanto L[j].chave < key ) faça
                    j = j − 1
                                        se j ≥ i então
                trocar (L[i], L[j])
                i: = i + 1 ; j = j − 1

        trocar (L[i], L[fim])
        quicksort (i + 1, fim)
quicksort (1, n)
```

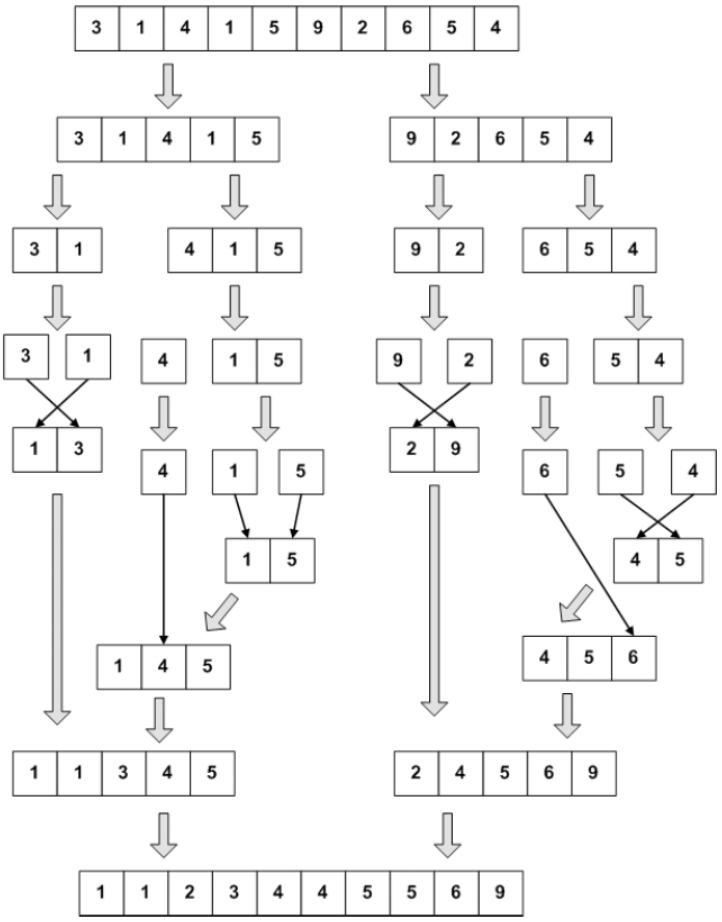
Nota: A escolha do pivô e o particionamento da tabela, são dois pontos decisivos para o bom desempenho do algoritmo.

Algoritmo de Ordenação por intercalação (MergeSort) é outro exemplo de algoritmo do tipo divisão e conquista, é um algoritmo de ordenação por intercalação ou segmentação. A idéia básica é a facilidade de criar uma seqüência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes. Como o algoritmo do Merge Sort usa a recursividade, em alguns problemas esta técnica não é muito eficiente devido ao alto consumo de memória e tempo de execução. Apresenta-se em seguida os passos do algoritmo:

- Dividir uma seqüência em duas novas seqüências.
- Ordenar, recursivamente, cada uma das seqüências (dividindo novamente, quando possível).
- Combinar (merge) as subsequências para obter o resultado final.

A desvantagem deste algoritmo é precisar de uma lista (vector) auxiliar para realizar a ordenação, ocasionando em gasto extra de memória, já que a lista auxiliar deve ter o mesmo tamanho da lista original.

Exemplo ilustrativo:



Algoritmo: mergesort

$procedimento\ mergesort\ ((esq, dir)$

$se\ esq\ <\ dir$ então

$centro := (esq + dir)/2;$

$mergesort(esq, centro);$

$mergesort(cenro\ + 1, dir);$

$intercalar(esq, centro + 1, dir);$

$procedimento\ intercalat\ (L, tmp, ini1, ini2, fim2);$

$fim1 := ini2 - 1;\ nro := 0$

$ind := ini1$

enquanto $(ini1 \leq fim1)$ e $ini2 \leq fim2)$ faça

se $L[ini1].chave\ < L[ini2].chave$ então

$tmp[ind] := L[ini1]$

$ini1 := ini1 + 1$

senão $tmp[ind] := L[ini2]$

$ini2 := ini2 + 1$

$ind := ind + 1\ ;\ nro := nro + 1$

enquanto $\{ini1 \leq fim1\}$ faça

$tmp[ind] := L[ini]$

$ini1 := ini1 + 1;\ ind := ind + 1; nro := nro + 1;$

enquanto $\{ini2 \leq fim2\}$ faça

$tmp[ind] := L[ini2]$

$ini2 := ini2 + 1;\ ind := ind + 1; nro := nro + 1$

para $i := 1, \ldots, nro$ faça

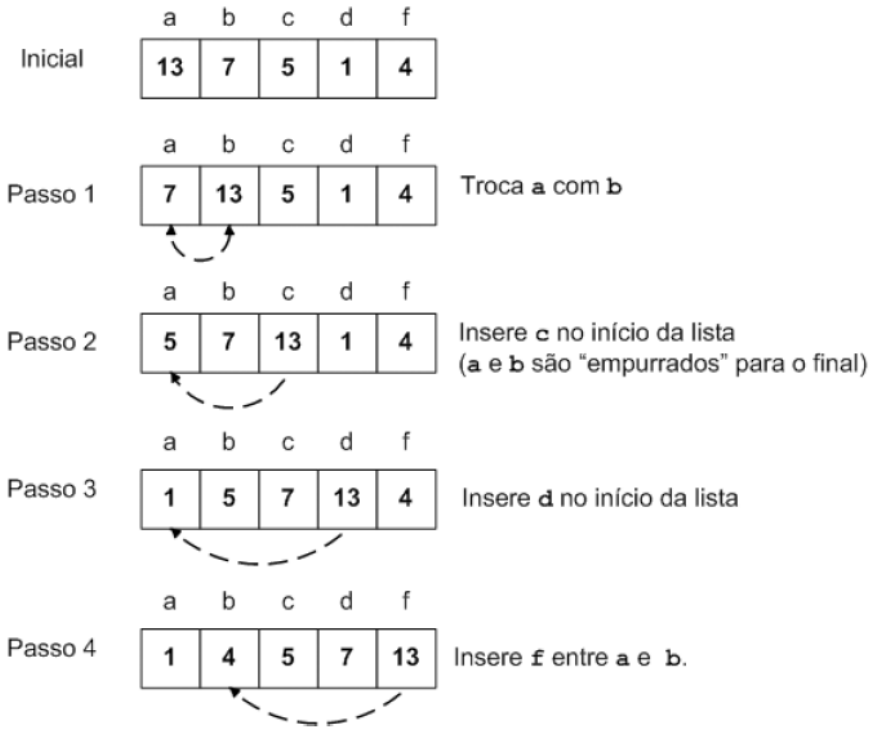
$L[i + ini1 - 1] := tmp[i + ini1 - 1]$

$mergesort\ (1, n)$

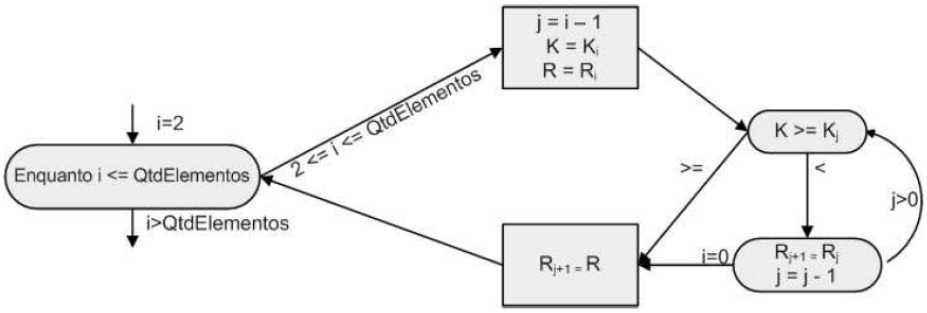
Algoritmo de Ordenação por Inserção (Insertion Sort): a ordenação por inserção é um algoritmo simples e indicado para listas pequenas de valores a serem ordenados. Em termos gerais, ele percorre um conjunto de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados. Possui menor número de trocas e comparações entre os algoritmos de ordenação.

Exemplo de aplicação

inicial	13	7	5	1	4
passo 1	**7**	**13**	5	1	4
passo 2	**5**	7	13	1	4
passo 3	**1**	5	7	13	4
passo 4	1	**4**	5	7	13



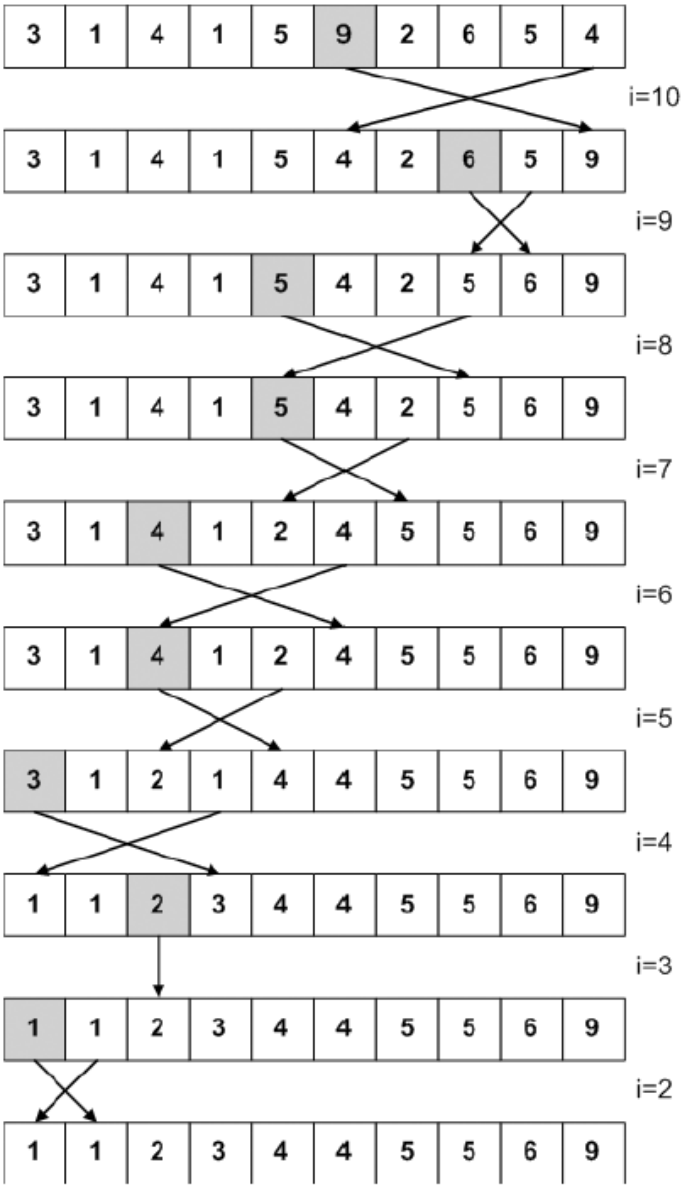
Algoritmo Insertion Sort



Algoritmo de Ordenação por Selecção (Selection Sort): ordenação por selecção consiste em trocar o menor elemento (ou maior) de uma lista com o elemento posicionado no início da lista, depois o segundo menor elemento para a segunda posição e assim sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos. A complexidade deste algoritmo é (n - 1) + (n -) + … + 2 + 1 = n(n −1)/2 = n2 comparações.

Exemplo de aplicação do algoritmo

inicial	13	7	5	1	4
passo 1	**1**	7	5	**13**	4
passo 2	1	**4**	5	13	**7**
passo 3	1	4	5	13	7
passo 4	1	4	5	**7**	**13**



Conclusão

A maior parte dos algoritmos de ordenação funciona pela comparação dos dados que estão sendo ordenados. Determina-se um pedaço de dados e denota-se por chave e, é a partir da chave que os dados são comparados e consequentemente ordenados.

Algoritmos de ordenação normalmente são julgados por sua eficiência.

Neste caso, refere-se a eficiência, como sendo a eficiência algorítmica à medida que o tamanho de dados de entrada cresce. A maior parte dos algoritmos em uso tem uma eficiência algorítmica de O (n 2) ou de O(n *log (n)).

Avaliação

1. Compare os algoritmos de ordenação por selecção e por inserção.
2. Explique o que significa algoritmo eficiente.

Actividade 1.2 - Algoritmos de Pesquisa

Introdução

Organizar e recuperar informação está no centro da maioria das aplicações informáticas, e pesquisa é certamente a mais realizada de entre todas as tarefas de computação. Nesta secção, o aluno irá aprender os algoritmos de pesquisa.

Detalhes da actividade

Entende-se por pesquisa como sendo a operação que permite encontrar ou concluir que não existe, um dado elemento num dado conjunto. A pesquisa de um elemento pode ser feita num conjunto ordenado ou não. Quando o conjunto não está ordenado, o método usado é o exaustivo, que consiste em percorrer sequencialmente todo o conjunto (desde o primeiro) até se encontrar o elemento desejado ou, não o encontrando, se concluir que não existe. Quando o conjunto está ordenado, pode-se utilizar uma pesquisa binária, o que ajuda a localizar o dado mais rapidamente. Existem dois tipos de pesquisa; pesquisa sequencial e pesquisa binária.

Pesquisa Sequêncial

Este é o método mais simples de pesquisa, e consiste em uma varredura

serial da tabela (vector, matriz ou arquivo), durante a qual o argumento de pesquisa é comparado com a chave de cada entrada até ser encontrada uma igual, ou ser atingido o final da tabela, caso a chave procurada não exista. A pesquisa sequencial é fácil de ser codificada.

A função de pesquisa pode ser implementada por duas formas:

Retornar o próprio elemento encontrado;

Retornar o índice do elemento (no caso de um vector)

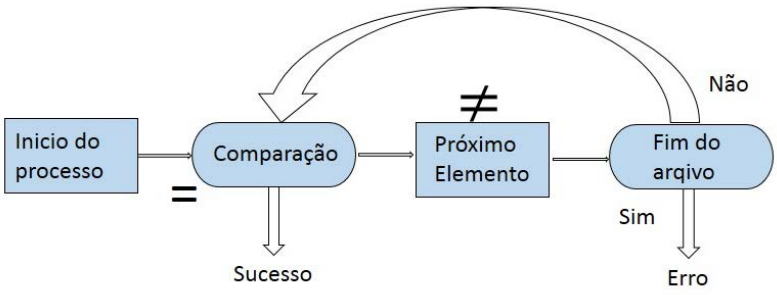


Figura : Pesquisa sequêncial

Exemplo: Pesquisa sequencial

Procurar um elemento X num vector V de tamanho Max

```
k ← -1 // denota que X ainda não foi encontrado em V
i ← 0 // índice dos elementos do vector V
enquanto  (i < Max)  e   (k = -1)  faça
se  (V[i] = X) então
k ← i
senão
se  (V[i] < Max) então
i ← i + 1
senão
k ← -2; // denota que X não está em V
se  (k ≥ 0) then
X está  na posição k
senão
X não está  em V
```

Pesquisa Binária

Se os elementos de um dado vector estiverem ordenados, pode ser utilizado um método muito superior e mais eficiente para encontrar o elemento procurado. Esse método é a pesquisa binária que utiliza a abordagem dividir e conquistar. Ele primeiro verifica o elemento central, se esse elemento é maior que a chave, ele testa o elemento central da primeira

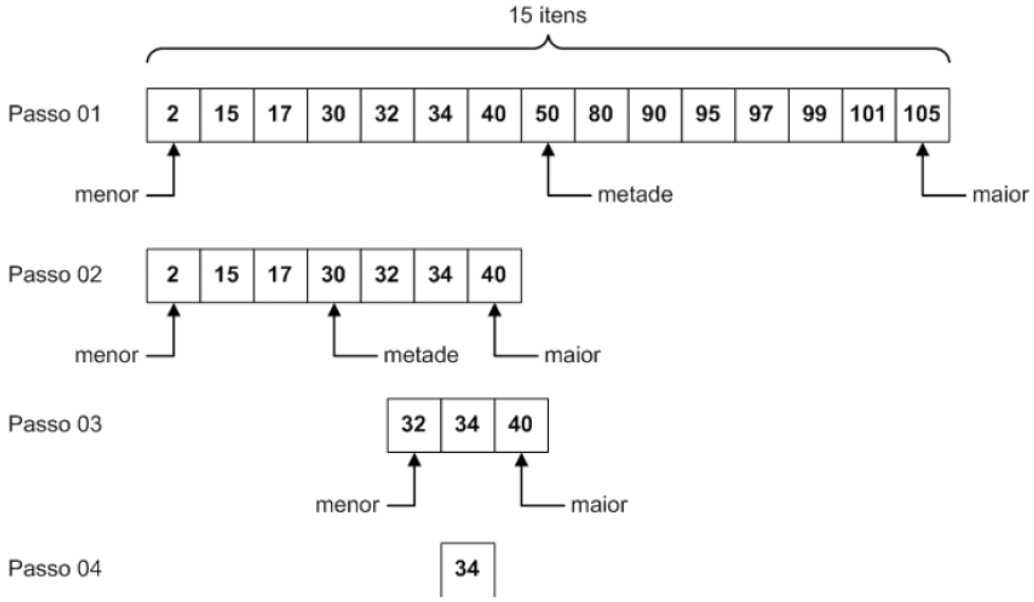
metade; caso contrário, ele testa o elemento central da segunda metade. Esse procedimento é repetido até que o elemento seja encontrado ou que não haja mais elementos a testar.

Por exemplo:

Uma matriz contêm os elementos 1 2 3 4 5 6 7 8 9 para encontrar o número 4, uma pesquisa binária primeiro testa o elemento médio, nesse caso. Visto que é maior que 4, a pesquisa continua com a primeira metade isto é 1 2 3 4 5. O elemento central agora é 3, que é menor que 4, então a primeira metade é descartada.

A pesquisa continua com 4 5 e, nesse momento o elemento é encontrado.

Figura: Pesquisa binária



Algoritmo: Pesquisa binária

Procurar o elemento X no vector V de tamanho Max

```
inicio ← 0 //inicializar o início e o fim
fim ← tam - 1
k ← -1 // k recebe a posição de X
while ( (inicio ≤ fim) and (k = -1) ) do
meio ← (inicio + fim) / 2
if (X = V[meio]) then
k ← meio
else
```

```
            if (X < V[meio]) então
        fim ← meio - 1
        else
        inicio ← meio + 1
        if (k ≥ 0) então X encontra-se em V na posição k
        else
        X não se encontra em V
```

Actividade 1.3.

Conclusão

Nesta unidade os alunos aprenderam as técnicas para a resolução de um problema que aparece como pré-processamento em várias aplicações que envolvem o uso de tabelas. Apresentou-se muitos algoritmos de ordenação e de pesquisa. Destes algoritmos, alguns são mais eficientes como é o caso de ordenação rápida e outros não, a exemplo de ordenação por bolhas. Seleccionar o algoritmo de ordenação correcto é usualmente denotado puramente por eficiência. Alguns algoritmos são muito bem expressos de uma forma recursiva, no entanto estes algoritmos deve ser bastante e eficientes, por exemplo, implementação de uns algoritmos linear, quadrático, ou os mais lentos usando recursão não seria uma boa idéia.

Avaliação

1. Calcule o número médio de comparações necessário para localizar uma entrada em tabelas com 15 e 20 entradas, para a pesquisa sequencial e para a pesquisa binária.
2. Refira-se aos conceitos de Algoritmos de Ordenação e Pesquisa dando exemplos.

Resumo da Unidade

Na presente unidade, a discussão centrou-se na implementação de dois tipos de algoritmos amplamente utilizados na elaboraçao de programas nomeadamente ordenação e pesquisa. Em diversas aplicações, os dados devem ser armazenados segundo uma determinada ordem, pois muitos algoritmos podem explorar essa ordenação dos dados para operar de maneira mais eficiente do ponto de vista de desempenho computacional. O algoritmo de busca por exemplo pode tirar proveito da ordenação dos dados. A operação de busca é tão frequente em aplicações que diversas estruturas de dados são projectadas especificamente para oferecer suporte a essa operação.

Avaliação da Unidade

Verifique a sua compreensão!

Instruções

Leia o enunciado do teste e responda as perguntas que se seguem:

Critérios de Avaliação

Pergunta	Pontuação
1	2.5
2	2.0
3	2.5
4	3.0
Maximo	10 valores

Avaliação

1. Implemente os algoritmos de busca sequencial e de inserção em C para vectores e listas ligadas.
2. Considere que a ordenação para n números leve t segundos. Calcule o tempo de ordenação para 10, 100, 1.000, 10.000 e 100.000 elementos para todos os algoritmos vistos.

Leituras e outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso.

- http://www.di.ubi.pt
- Tenenbaum, A. M.;. Langsam, Y. ; Moshe J. A. Estruturas de dados usando C São Paulo : MAKRON Books, 1995.

Avaliação do Curso

Exame Final 1

Instruções

Das perguntas que se seguem, leia e responda-as de forma clara e concisa, dando exemplos sempre lhe for pedido.

Critérios de avaliação

Pergunta	Pontuação
1	2.0
2	2.0
3	2.0
4	2.0
5	4.0
6	4.0
7	2.0
8	2.0
Máximo	20.0

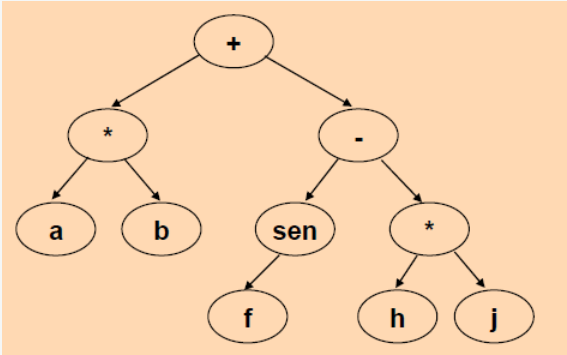
Avaliação

1. Responda se é verdadeiro ou não:

   - Todo o procedimento recursivo deve incorporar terminações sem chamadas recursivas, caso contrário ele será executado um número infinito de vezes.
   - Se e são funções tais que

2. Imagine um vector A de inteiros. Apresente um algoritmo recursivo para calcular o elemento máximo do vector.
3. Refira-se às vantagens e desvantagens de representar um grupo de itens como um vector versus uma lista ligada ?
4. Que conjunto de condições é necessário e suficiente para que uma sequência de operações de inserir e remover sobre uma única fila vazia deixe a fila vazia sem provocar underflow?
5. Implemente os procedimentos empty, push, pop usando as implementações de alocamento sequencial e dinâmica de uma pilha ligada.

6. Crie uma árvore vazia e mostre através da árvore seguinte a implementação dos percursos pré-ordem, em ordem e pós ordem.



7. Explique os conceitos de tipos abstractos de dados e qual é a relação com estruturas de dados.

8. Analise a eficiencia do algoritmo de ordenação por bolhas.

Respostas:

1. Responda se é verdadeiro ou não:

- Todo o procedimento recursivo deve incorporar terminações sem chamadas recursivas, caso contrário ele será executado um número infinito de vezes.

Verdadeiro

Se f e g são funções tais que $f = O(g)$ e $g = \Omega(f)$, então $f = \Theta(g)$.

Verdadeiro

2. Imagine um vector A de inteiros. Apresente um algoritmo recursivo para calcular o elemento máximo do vector.

```
MaxA ( n, v[] ): natural
  se n:= 1
        retorna  v[0]
  senão
        x: natural ;
        x = MaxA(n − 1, v)
        if (x > v[n − 1])
              retorna x
        senão
              retorna  v[n − 1];
```

3. Refira-se às vantagens e desvantagens de representar um grupo de itens como um vector versus uma lista ligada?

	Vantagens	Desvantagens
Vectores	Inserção e eliminação de um elemento no final do vector Aleatoriamente aceder a qualquer elemento Pesquisar a lista para um elemento particular	não é uma estrutura de dados muito flexível, não existe uma maneira simples e barata (computacionalmente) para alterar a dimensão do vector em tempo de execução.
Listas Ligadas	Inserção de um elemento. Exclusão de um elemento. Aplicações onde o acesso sequencial é necessário Situações em que o número de elementos não pode ser previsto de antemão	não há acesso directo aos elementos da lista Cada nó precisa de, no mínimo, uma referencia para o próximo, usando memória para um ponteiro por cada nó. O acesso de cada elemento é feito percorrendo a lista

4. Que conjunto de condições é necessário e suficiente para que uma sequência de operações de inserir e remover sobre uma única fila vazia deixe a fila vazia sem provocar underflow?

   R: É necessário que se verifique se a pilha contêm algum elemento ainda por remover e informar a aplicação sobre esse estado.

5. Implemente os procedimentos empty, push, pop.

   Procedimento empty

   Booleano FUNÇÃO pilhaVazia()

   início

   SE (topo := -1) ENTÃO

   RETORNE(Verdade)

   SENÃO

   RETORNE(Falso);

   fim;

```
Procedimento pop
Inteiro FUNÇÃO desempilha()
início
SE (pilhaVazia) ENTÃO
RETORNE("A pilha está vazia" );
SENÃO
topo ←- topo - 1;
RETORNE(topo);
FIM SE
fim;
Procedimento push
Constantes
PilhaCheia = -1;
PilhaVazia = -2;
Inteiro FUNÇÃO empilha(inteiro dado)
início
SE (pilhaCheia) ENTÃO
RETORNE("A pilha está cheia");
SENÃO
topo ←-- topo + 1.
dados[topo] ←--dado;
RETORNE(topo);
FIM SE
fim;
```

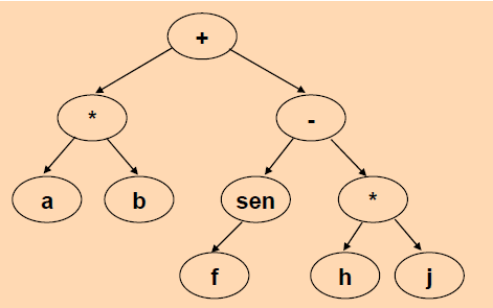
6. Crie uma árvore vazia e mostre através da árvore seguinte a implementação dos percursos pré-ordem, em ordem e pós ordem.

```
criarArvore
Entradas: --
Saídas: Arvore (tipoArvore)
```

inicio

Arvore ←--nulo

fim



Pré-ordem	* a b - sen f * h j
Em-ordem	a * b + f sen - h + j
Pós-ordem	a b * f sen h j * - +

7. Explique os conceitos de tipos abstractos de dados e qual é a relação com estruturas de dados.

—R: Um Tipo Abstrato de Dados (TAD) é um modelo matemático de um conjunto de dados, — basicamente, a definição de um TAD inclui —o tipo dos dados, as operações que agem sobre esses dados e, não —inclui detalhes sobre o formato de armazenamento dos dados e de implementação das operações. — A definição de um TAD indica, o que faz cada operação, e não como se faz. Enquanto que as —Estruturas de dados incluem detalhes de implementação dos algoritmos e materializam as TAD´s .

8. Analise a eficiencia do algoritmo de ordenação por bolhas.

R: O tempo computacional para a execução deste algoritmo varia de forma quadrática com o tamanho do problema isto é, de ordem O(No melhor caso, quando o vector fornecido estiver quase ordenado, o processo pode ser capaz de ordenar em uma única passagem.

Exame final 2

Instruções

Das perguntas que se seguem, leia e responda-as de forma clara e concisa, dando exemplos sempre lhe for pedido.

Critérios de avaliação

Pergunta	Pontuação
1	2.0
2	2.0
3	2.0
4-a)	2.0
4-b)	2.0
5	3.0
6	3.0
7	2.0
8	2.0
Máximo	20.0

Avaliação

1. Faça uma comparação entre Estruturas de Dados e Tipos Abstratos de Dados.
2. Em que consiste a complexidade de um algorítmo? Enuncie as diferentes regras para análise de algoritmos.
3. Enuncie as propriedades do O grande.
4. Determine a ordem de complexidade para os algoritmos que se seguem:

```
a.  int i, j, k, count = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
        Count++;
b.  for (int i = 1; i < n; i++)
```

```
{
        min = i;
       for (int j = i; j< m; j++)
        {
         if a[j] < a[min]
         {
                min = j;
                x = a[min];
                a[min] = a[i];
                a[i] = x;
                }
        }
}
```

5. Escreva um algorítmo para inserir um elemento num array na 3ª posição.
6. Escreva um algorítmo para apagar um elemento na última posição de um array.
7. Escreva um programa para percorrer um array.
8. Explique com detalhes, esquematizando se possível o processo de inserção e remoção de um nó numa lista ligada.

Referências do Curso

- Adam Drozdek , Data Structures and Algorithms in Java, 2ª edição.
- Barnett, Granville; Del Tongo, Luca. Data Structures and Algorithms: Annotated Reference with Examples 1st Edition, 2008.
- Bjarne Stroustrup, Programming: Principles and Practice Using C++, ISBN 0321992784, Publisher: Addison Wesley, 2014
- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Edelweiss, Nina; Galante, Renata; Estrutura de Dados, Porto Alegre: Bookman, 2009. ISBN 978-85-7780-381-1
- Harry H Chaudhary, Practical Data Structures Using C .: Beginner s Easy Edition 2014. ISBN 1500136972; Publisher: Createspace, United States, 2014
- Malik, D.S., Data Structures Using C++, Second Edition, , 2010
- Nivio Ziviani, Projecto de Algoritmos com Implementação em Java e C++.. 2006. Editora Thomson, ISBN 8522105251
- Robert Sedgewick, ALGORITHMS IN C : FUNDAMENTALS, DATA STRUCTURES, SORTING, SEARCHING, PARTS 1-4, ISBN 8131713059; Publisher: Pearson Education/AW Professional, 2012
- Robert Sedgewick, Algorithms in C: Parts 1-5: Fundamentals Data Structures, Sorting, Searching, and Graph Algorithms, ISBN 8178082497; Publisher: Addison Wesley, 2001
- Rocha, A. M. A, Estruturas de Dados e Algoritmos em C, 2008 FCA Editora Informática. Coleção:Tecnologias de Informação
- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999
- SEYMOUR LIPSCHUTZ, G. A. VIJAYALAKSHMI PAI. Data Structures. Tata McGraw-Hill Publishing Company Limited. New Delhi, 2006.
- Tenenbaum, A. M.;. Langsam, Y. ; Moshe J. A. Estruturas de dados usando C São Paulo : MAKRON Books, 1995.
- Szwarcfiter, Jaime Luiz; Markenzo n, Lilian; Estruturas de Dados e Seus Algoritmos 3ª edição, Rio de Janeiro 2012.
- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- Vic Broquard, Beginning Data Structures in C++, ISBN 1941415547, Publisher: Broquard eBooks, United States, 2014
- Walmar, Celes; Cerqueira, Renato; Rangel, J. L.; Introdução a Estrutura de Dados, Rio de Janeiro, 2004. Elsevier.

Sede da Universidade Virtual africana

The African Virtual University
Headquarters

Cape Office Park

Ring Road Kilimani

PO Box 25405-00603

Nairobi, Kenya

Tel: +254 20 25283333

contact@avu.org

oer@avu.org

Escritório Regional da Universidade Virtual Africana em Dakar

Université Virtuelle Africaine

Bureau Régional de l'Afrique de l'Ouest

Sicap Liberté VI Extension

Villa No.8 VDN

B.P. 50609 Dakar, Sénégal

Tel: +221 338670324

bureauregional@avu.org