GRADUAÇÃO
Unicesumar

ESTRUTURA DE DADOS II

Professor Me. Pietro Martins de Oliveira
Professor Me. Rogério de Leon Pereira

UNICESUMAR
Av. Guedner, 1610 - Jardim Aclimação
Cep 87050-900 - MARINGÁ - PARANÁ
unicesumar.edu.br
44 3027.6360

UNICESUMAR EDUCAÇÃO A DISTÂNCIA
NEAD - Núcleo de Educação a Distância
Bloco 4 - MARINGÁ - PARANÁ
unicesumar.edu.br
0800 600 6360

as imagens utilizadas neste livro foram obtidas a partir do site SHUTTERSTOCK.COM

FICHA CATALOGRÁFICA

C397 **CENTRO UNIVERSITÁRIO DE MARINGÁ.** Núcleo de Educação a Distância; **OLIVEIRA**, Pietro Martins de; **PEREIRA**, Rogério de Leon.

Estrutura de Dados II. Pietro Martins de Oliveira; Rogério de Leon Pereira.
Maringá-Pr.: Unicesumar, 2019.
152 p.
"Graduação - EaD".

1. Análise de Sistemas. 2. Estrutura de Dados. 3. EaD. I. Título.

ISBN 978-85-459-1767-0

CDD - 22 ed. 005.73
CIP - NBR 12899 - AACR/2

Ficha catalográfica elaborada pelo bibliotecário
João Vivaldo de Souza - CRB-8 - 6828

Impresso por:

Professor
Wilson de Matos Silva
Reitor

Em um mundo global e dinâmico, nós trabalhamos com princípios éticos e profissionalismo, não somente para oferecer uma educação de qualidade, mas, acima de tudo, para gerar uma conversão integral das pessoas ao conhecimento. Baseamo-nos em 4 pilares: intelectual, profissional, emocional e espiritual.

Iniciamos a Unicesumar em 1990, com dois cursos de graduação e 180 alunos. Hoje, temos mais de 100 mil estudantes espalhados em todo o Brasil: nos quatro campi presenciais (Maringá, Curitiba, Ponta Grossa e Londrina) e em mais de 300 polos EAD no país, com dezenas de cursos de graduação e pós-graduação. Produzimos e revisamos 500 livros e distribuímos mais de 500 mil exemplares por ano. Somos reconhecidos pelo MEC como uma instituição de excelência, com IGC 4 em 7 anos consecutivos. Estamos entre os 10 maiores grupos educacionais do Brasil.

A rapidez do mundo moderno exige dos educadores soluções inteligentes para as necessidades de todos. Para continuar relevante, a instituição de educação precisa ter pelo menos três virtudes: inovação, coragem e compromisso com a qualidade. Por isso, desenvolvemos, para os cursos de Engenharia, metodologias ativas, as quais visam reunir o melhor do ensino presencial e a distância.

Tudo isso para honrarmos a nossa missão que é promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária.

Vamos juntos!

Janes Fidélis Tomelin
Pró-Reitor de Ensino de EaD

Kátia Solange Coelho
Diretoria de Graduação e Pós

Débora do Nascimento Leite
Diretoria de Design Educacional

Leonardo Spaine
Diretoria de Permanência

Seja bem-vindo(a), caro(a) acadêmico(a)! Você está iniciando um processo de transformação, pois quando investimos em nossa formação, seja ela pessoal ou profissional, nos transformamos e, consequentemente, transformamos também a sociedade na qual estamos inseridos. De que forma o fazemos? Criando oportunidades e/ou estabelecendo mudanças capazes de alcançar um nível de desenvolvimento compatível com os desafios que surgem no mundo contemporâneo.

O Centro Universitário Cesumar mediante o Núcleo de Educação a Distância, o(a) acompanhará durante todo este processo, pois conforme Freire (1996): "Os homens se educam juntos, na transformação do mundo".

Os materiais produzidos oferecem linguagem dialógica e encontram-se integrados à proposta pedagógica, contribuindo no processo educacional, complementando sua formação profissional, desenvolvendo competências e habilidades, e aplicando conceitos teóricos em situação de realidade, de maneira a inseri-lo no mercado de trabalho. Ou seja, estes materiais têm como principal objetivo "provocar uma aproximação entre você e o conteúdo", desta forma possibilita o desenvolvimento da autonomia em busca dos conhecimentos necessários para a sua formação pessoal e profissional.

Portanto, nossa distância nesse processo de crescimento e construção do conhecimento deve ser apenas geográfica. Utilize os diversos recursos pedagógicos que o Centro Universitário Cesumar lhe possibilita. Ou seja, acesse regularmente o Studeo, que é o seu Ambiente Virtual de Aprendizagem, interaja nos fóruns e enquetes, assista às aulas ao vivo e participe das discussões. Além disso, lembre-se que existe uma equipe de professores e tutores que se encontra disponível para sanar suas dúvidas e auxiliá-lo(a) em seu processo de aprendizagem, possibilitando-lhe trilhar com tranquilidade e segurança sua trajetória acadêmica.

Professor Me. Pietro Martins de Oliveira

Possui graduação em Engenharia de Computação pela Universidade Estadual de Ponta Grossa (2011). É mestre em Ciência da Computação na área de Visão Computacional pela Universidade Estadual de Maringá (2015). Atuou como analista de sistemas e programador nas empresas Siemens Enterprise Communications (Centro Internacional de Tecnologia de Software - CITS, 2011-2012), e Benner Saúde Maringá (2015). Experiência de dois anos como coordenador dos cursos de Bacharelado em Engenharia de Software, Sistemas de Informação e Engenharia de Produção. Docente no ensino superior.

Professor Me. Rogério de Leon Pereira

Possui graduação em Tecnologia em Processamento de Dados pelo Centro de Ensino Superior de Maringá (1999) e mestrado em Ciência da Computação pela Universidade Estadual de Maringá (2006). Atualmente é analista de informática da Universidade Estadual de Maringá. Tem experiência na área de Ciência da Computação, com ênfase em desenvolvimento de sistemas Web.

ESTRUTURA DE DADOS II

SEJA BEM-VINDO(A)!

As estruturas de dados são o alicerce principal das aplicações computacionais. Sem elas, de que maneira os desenvolvedores teriam ferramentas para organizar os dados de maneira eficaz e eficiente? Neste livro, teremos a oportunidade de ter contato com estruturas e métodos avançados que são muito utilizados na computação, do mais baixo ao mais alto nível de abstração.

Na Unidade 1, teremos contato com as árvores. Essa é uma estrutura de dados muito conhecida pelos cientistas da computação, especialmente por aproveitar a memória de um sistema de maneira bastante otimizada. Veremos como podemos implementar esse tipo de estrutura em linguagem C.

Dando continuidade ao tema relacionado a árvores, na Unidade 2 teremos contato com operações que podem ser realizadas em árvores binárias. Em especial, veremos três técnicas utilizadas para visitar ou percorrer os nós de uma árvore binária. Além disso, veremos como utilizar árvores para realizar buscas de dados, bem como utilizar as técnicas de rotação para compor árvores balanceadas (AVL).

Adentrando na Unidade 3, damos início à investigação sobre técnicas para realizar a ordenação em vetores, arranjos, tabelas etc. Inicialmente conhecemos métodos mais simples, como o Bubblesort e o Selectionsort. Em seguida, tentamos algo mais otimizado, estudando os algoritmos Insertionsort e Shellsort.

Na Unidade 4, dando continuidade aos estudos de técnicas para ordenação de vetores, veremos métodos mais avançados e de implementação mais elaborada, como é o caso do Mergesort, Heapsort e o Quicksort. Teremos uma boa noção de como tais algoritmos utilizam o conceito de "dividir para conquistar" durante a resolução do problema de ordenação.

A ordenação de vetores é especialmente importante em situações nas quais é preciso realizar buscas nesse tipo de estrutura de dados. Por isso, ao final, a Unidade 5 aborda técnicas de busca em vetores estáticos. Tudo isso é feito de maneira bastante intuitiva e facilitada, para que tenhamos a compreensão necessária para identificar, analisar e implementar estruturas de dados.

SUMÁRIO

UNIDADE I

ÁRVORES BINÁRIAS

UNIDADE II

OPERAÇÕES SOBRE ÁRVORES BINÁRIAS

SUMÁRIO

SUMÁRIO

UNIDADE V

OPERAÇÕES DE BUSCA

Professor Me. Rogério de Leon Pereira

ÁRVORES BINÁRIAS

UNIDADE I

Objetivos de Aprendizagem

- Conhecer a estrutura de árvore binária.
- Aprender sobre árvore estritamente binária.
- Reconhecer uma árvore binária completa.
- Implementar uma árvore binária em linguagem C.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Árvore Binária
- Árvore Estritamente Binária
- Árvore Binária Completa
- Implementando Árvore Binária em C
- Uma Árvore Binária Diferente

Unicesumar

INTRODUÇÃO

Nesta unidade, estudaremos uma estrutura de dados útil para várias aplicações: a árvore binária. Definiremos algumas formas dessa estrutura de dados e mostraremos como podem ser implementadas na linguagem C. A árvore como estrutura é muito utilizada para organizar informações armazenadas tanto na memória principal como na secundária. Isso se dá devido ao fato de ser fácil e rápida a pesquisa de dados em árvores. Daremos foco no entendimento da estrutura e não na sua definição matemática.

ÁRVORE BINÁRIA

Segundo Tenenbaum (1995, p. 303),

> Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos disjuntos. O primeiro subconjunto contém um único elemento, chamado ***raiz*** da árvore. Os outros dois subconjuntos são em si mesmos árvores binárias, chamadas ***subárvores esquerda*** e ***direita*** da árvore original. Uma subárvore esquerda ou direita pode estar vazia. Cada elemento de uma árvore binária é chamado *nó* da árvore.

A Figura 1 apresenta um exemplo de árvore binária. Essa árvore possui 9 nós, sendo **A** o nó raiz da árvore. Ela possui uma subárvore esquerda com raiz em **B** e uma direita com raiz em **C**. A subárvore com raiz em **B** possui uma subárvore esquerda com raiz em **D,** e a direita com raiz em **E**. A subárvore com raiz em **C** possui uma subárvore esquerda **vazia** e uma subárvore direita com raiz em **F**. As árvores binárias com raiz em **D**, **G**, **H** e **I** possuem subárvore direita e esquerda vazias.

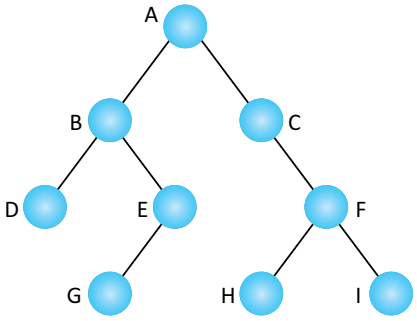


Figura 1 - Exemplo de árvore binária
Fonte: o autor.

Se **A** é o nó raiz de uma árvore binária e tem uma subárvore com raiz em **B**, então se diz que **A** é pai de **B** e **B** é filho esquerda de **A**. Na mesma forma, se **A** tem uma subárvore direita com raiz em **C**, diz-se que **A** é pai de **C** e **C** é filho direita de **A**. Um nó sem filhos é chamado de **folha**. Na árvore representada pela Figura 1 os nós **D**, **G**, **H** e **I** são considerados folhas.

Os nós **D** e **E** são ambos filhos de **B**, dessa forma podemos considerá-los como nós **irmãos**. Também podemos classificar os nós de acordo com a sua hereditariedade. O nó **B** é ancestral dos nós **D**, **E** e **G**, da mesma forma que esses três nós são descendentes de **B**.

REFLITA

Uma árvore binária é um tipo de grafo que tem regras específicas na sua construção. Cada nó tem no máximo dois filhos e um único pai, excetuando-se o nó raiz da árvore principal, que é órfão.

Quando se percorre uma árvore a partir da raiz em direção às folhas, diz-se que se está caminhando **para baixo,** ou **descendo** na árvore. De forma análoga, quando se está percorrendo uma árvore a partir de uma de suas folhas em direção à raiz, diz-se que se está caminhando **para cima,** ou **subindo** na árvore. Se fôssemos fazer uma analogia entre uma árvore na informática e uma árvore real (planta), na computação as árvores seriam representadas de cabeça para baixo.

ÁRVORE ESTRITAMENTE BINÁRIA

Uma árvore é considerada **estritamente binária** se todo nó que não for **folha** tiver sempre subárvores direita e esquerda não vazias. Na Figura 2, temos um exemplo de árvore estritamente binária. São considerados **folhas** os nós **C**, **D**, **F** e **G**. Os nós **A**, **B** e **E** possuem subárvores esquerda e direita não vazias.

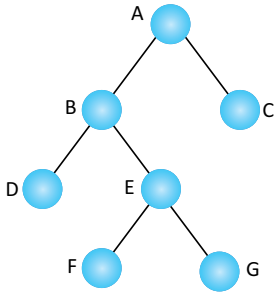


Figura 2 - Exemplo de árvore estritamente binária
Fonte: o autor.

A árvore apresentada na Figura 2 não pode ser considerada como uma árvore estritamente binária já posto que o nó **C** possui uma subárvore esquerda vazia, e o nó **E** possui uma subárvore direita vazia.

A quantidade de nós presentes numa árvore estritamente binária se dá pela fórmula $n = (2 * f) - 1$ em que f é o número de folhas na árvore. Vamos testar essa fórmula a partir da árvore representada pela Figura 2.

A árvore possui 4 folhas, quais sejam os nós: **C, D, F** e **G**. Aplicando-se a fórmula, temos:

$$n = (2 * f) - 1$$

Como $f = 4$, substituímos o valor de f na fórmula:

$$n = (2 * 4) - 1$$

Realizamos a multiplicação e em seguida a subtração.

$$n = (8) - 1$$

Temos então que:

$$n = 7$$

A árvore possui exatamente 7 nós: **A**, **B**, **C**, **D**, **E**, **F** e **G**.

Também podemos calcular a quantidade de folhas em uma árvore estritamente binária pela dedução da fórmula : $n = (2 * f) - 1$

$$n = (2 * f) - 1$$

Adicionando 1 em ambos os lados da igualdade.

$$n + 1 = (2 * f) - 1 + 1$$

Efetuando-se o cálculo:

$$n + 1 = (2 * f)$$

Dividindo ambos os lados da igualdade por 2:

$$\frac{n+1}{2} = \frac{(2 * f)}{2}$$

Efetuando-se o cálculo:

$$\frac{n+1}{2} = f$$

Vamos inverter os lados da igualdade para deixar a variável isolada à esquerda:

$$f = \frac{n+1}{2}$$

No caso da árvore estritamente binária da Figura 2, sabemos que possui 7 nós. Colocando na fórmula:

$$f = \frac{7+1}{2}$$

Realizando a soma e em seguida a divisão:

$$f = \frac{8}{2}$$

Temos que:

$$f = 4$$

A árvore possui exatamente 4 folhas: **C**, **D**, **F** e **G**.

ÁRVORE BINÁRIA COMPLETA

O nó **raiz** de uma árvore binária é considerado como de nível 0. A partir dele, cada nó possui um nível a mais do que o seu pai. Por exemplo, na árvore binária da Figura 3, o nó **C** está no nível 1, **F** está no nível 2 e o nó **M,** no nível 3. A *profundidade* ou *altura* de uma árvore binária é dada pelo maior nível de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até uma folha qualquer. Dessa forma, a profundidade da árvore da Figura 3 é 3.

Quando uma árvore estritamente binária possui todas as folhas no último nível, ela é chamada de árvore binária completa. A Figura 3 demonstra uma árvore binária completa de profundidade 3.

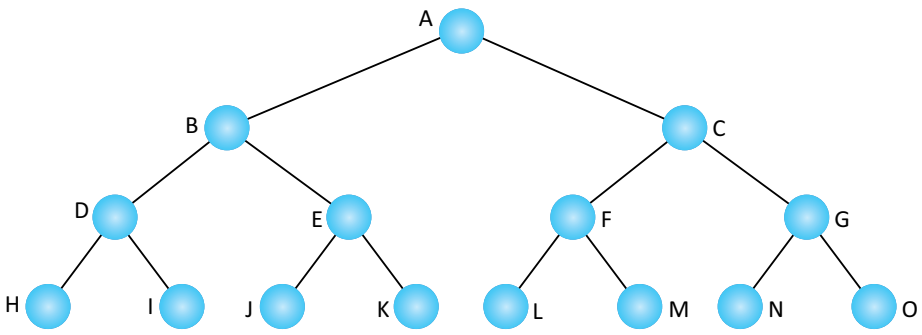


Figura 3 - Exemplo de árvore binária completa
Fonte: o autor.

IMPLEMENTANDO ÁRVORE BINÁRIA EM C

Existem várias formas de implementar uma árvore binária. A mais simples delas é usar um vetor de **nós**. Cada nó possui pelo menos três valores: **pai**, **esquerda** e **direita**. O atributo **pai** vai apontar para a posição no vetor do **pai** do **nó**. O atributo **esquerda** vai armazenar a posição da raiz da subárvore **esquerda,** e o atributo **direita** guarda a posição da raiz da subárvore **direita**. Vamos adicionar mais um atributo, **dado**, que irá armazenar o valor do **nó**.

Programa 1.1 – Estrutura do nó

```
//Estrutura
struct str_no {
  char dado;
  int esquerda;
  int direita;
  int pai;
};
```

Com a estrutura já definida no Programa 1.1, vamos criar uma variável do tipo **vetor** de **str_no**. Esse vetor terá o tamanho definido por uma constante chamada **tamanho**. Precisaremos também de uma variável do tipo **inteiro** que servirá de índice para o nosso vetor.

Programa 1.2 – Variáveis

```
//Constantes
#define tamanho 100

//Variáveis
struct str_no arvore[tamanho];
int indice=0;
```

Para inserir um **nó** na árvore, eu preciso saber o seu valor, quem é o seu **pai** e se ele é um filho **esquerda** ou **direita**. Mesmo sabendo quem é o pai, antes de fazer a inserção no vetor eu preciso encontrar a sua localização. Vou criar uma função chamada **arvore_procura**, que retorna um valor **inteiro** (posição no vetor) e tem como parâmetro o nome do **pai**.

Programa 1.3 – Procurando um nó na árvore

```
//Procura nó
int arvore_procura(char dado){
  if (indice != 0){
    for (int i = 0; i<indice; i++){
      if (arvore[i].dado == dado) {
        return (i);
      }
    }
  }
  else {
    return (0);
  }
}
```

Note que o Programa 1.3 faz uma verificação no valor da variável **indice**. Se o valor for 0 significa que a árvore está vazia e o valor a ser inserido será a raiz da árvore. A função leva em conta que o **pai** está presente na árvore, já posto que não foi previsto nenhum tipo de retorno de erro para o caso do **pai** não ser encontrado.

Já fizemos a leitura do valor a ser inserido e já sabemos quem é o seu pai e qual a sua descendência. Por meio da função demonstrada no Programa 1.3, passamos o valor do pai como parâmetro e obtivemos como retorno a sua posição no vetor. Agora estamos prontos para a inserção do novo nó na árvore.

Vamos criar uma função nova, chamada **arvore_insere**. Ela terá como parâmetro um valor inteiro que representa a posição do **pai** no vetor, o valor **dado** digitado pelo usuário e a sua posição de descendência (se é filho do **lado** esquerdo ou direito).

Programa 1.4 – Inserindo um nó na árvore

```
 1 //Inserir nó
 2 void arvore_insere(int pai, char dado, int lado){
 3   switch (lado){
 4     case E:
 5        arvore[pai].esquerda = indice;
 6        arvore[indice].dado = dado;
 7        arvore[indice].pai = pai;
 8        arvore[indice].esquerda = -1;
 9        arvore[indice].direita = -1;
10        indice++;
11     break;
```

```
12      case D:
13         arvore[pai].direita = indice;
14         arvore[indice].dado = dado;
15         arvore[indice].pai = pai;
16         arvore[indice].esquerda = -1;
17         arvore[indice].direita = -1;
18         indice++;
19       break;
20    }
21 }
```

No Programa 1.2, criamos uma variável chamada **indice,** que guarda a primeira posição livre da **arvore**. O parâmetro **pai** recebido na função indica qual a posição do nó pai. Se o novo nó for filho esquerdo, atribuiremos o valor de **indice** ao atributo **esquerdo** na **arvore,** na posição **pai** (linha 5, Programa 1.4). No caso de filho direito, colocamos o valor de **indice** no atributo **direito** (linha 13, Programa 1.4).

O próximo passo é guardar o nome do nó no atributo **dado** e a referência do **pai**. Vamos marcar com -1 os valores de **esquerda** e **direita** para identificar que ambos os ponteiros não apontam para uma subárvore. Esses passos estão demonstrados no Programa 1.4 nas linhas 6 a 10 para filho esquerda e 14 a 18 para filho direita.

O Programa 1.5 traz uma implementação completa de uma Árvore Binária em linguagem C. Ele traz a declaração de bibliotecas, constantes e variáveis auxiliares, função principal, função para desenhar o menu de opções, dentre muitas outras linhas de código.

Programa 1.5 – Implementação de Árvore Binária em linguagem C

```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>

//Constantes
#define tamanho 100
#define E 0
#define D 1
#define R -1
```

```
//Estrutura
struct str_no {
  char dado;
  int esquerda;
  int direita;
  int pai;
};

//Variáveis
struct str_no arvore[tamanho];
int lado,indice=0;
int opt=-1;
char pai, no;

//Prototipação
void arvore_insere(int pai, char dado, int lado);
int arvore_procura(char dado);
void menu_mostrar(void);

//Função principal
int main(void){
  int temp;
  do {
    menu_mostrar();
    scanf("%d", &opt);
    switch (opt){
       case 1:
         printf("\nDigite o valor do PAI: ");
         scanf("%c",&pai);
         scanf("%c",&pai);
         printf("Digite o valor do NO: ");
         scanf("%c",&no);
         scanf("%c",&no);
        printf("Digite o lado da subarvore (E=%d/D=%d/R=%d): ",
                E,D,R);
         scanf("%d",&lado);
         temp = arvore_procura(pai);
         arvore_insere(temp,no,lado);
       break;
      case 2:
         printf("Digite o valor do NO: ");
         scanf("%c",&no);
        scanf("%c",&no);
        temp = arvore_procura(no);
```

```
        printf("No %c\nFilho Esquerda: %c
                \nFilho Direita: %c\n\n",
                 arvore[temp].dado,
                 arvore[arvore[temp].esquerda].dado,
                 arvore[arvore[temp].direita].dado);
        system("pause");
      break;
    }
  }while (opt!=0);
  system("pause");
  return(0);
}

//Inserir nó
void arvore_insere(int pai, char dado, int lado){
  switch (lado){
    case E:
       arvore[pai].esquerda = indice;
       arvore[indice].dado = dado;
       arvore[indice].pai = pai;
       arvore[indice].esquerda = -1;
       arvore[indice].direita = -1;
       indice++;
    break;
    case D:
       arvore[pai].direita = indice;
       arvore[indice].dado = dado;
       arvore[indice].pai = pai;
       arvore[indice].esquerda = -1;
       arvore[indice].direita = -1;
       indice++;
     break;
     case R:
       arvore[indice].dado = dado;
       arvore[indice].pai = -1;
       arvore[indice].esquerda = -1;
       arvore[indice].direita = -1;
       indice++;
     break;
  }
}

//Procura nó
int arvore_procura(char dado){
 if (indice != 0){
```

```
    for (int i = 0; i<indice; i++){
      if (arvore[i].dado == dado) {
        return (i);
      }
    }
  }
  else {
    return (0);
  }
}

//Desenha o menu na tela
void menu_mostrar(void){
  system("cls");
  for (int i = 0; i < indice; i++){
        printf("| %c ",arvore[i].dado);
  }
  printf("\n1 - Inserir um NO na arvore");
  printf("\n2 - Pesquisar um NO na arvore");
  printf("\n0 - Sair...\n\n");
}
```

Acabamos de ver uma forma de armazenar uma árvore binária em um vetor. Essa implementação é bem simples e rápida, mas longe do ideal. Voltamos ao mesmo problema de estruturas anteriores em que muita memória é alocada na execução do programa para uma aplicação que pode ou não precisar de todo o espaço reservado na memória.

O Programa 1.5 procura sempre a primeira posição livre no vetor para posicionar um novo nó na árvore. Dessa forma, a ordenação da estrutura estará diretamente ligada à ordem que os nós foram adicionados.

SAIBA MAIS

Esse material criado pelo Prof. Paulo Feofiloff, da USP, traz um resumo dos principais termos relacionados a árvores binárias, além de exemplos ilustrados e trechos de código em linguagem C. Acesse: <http://www.ime.usp.br/~pf/algoritmos/aulas/bint.html>.

Fonte: o autor.

UMA ÁRVORE BINÁRIA DIFERENTE

Outra forma de armazenar uma árvore binária em um vetor é reservar as posições de acordo com o nível e descendência de cada nó. O primeiro nó a ser armazenado é a raiz da árvore e ele ficará na primeira posição do vetor, lembrando que os vetores em C começam na posição 0.

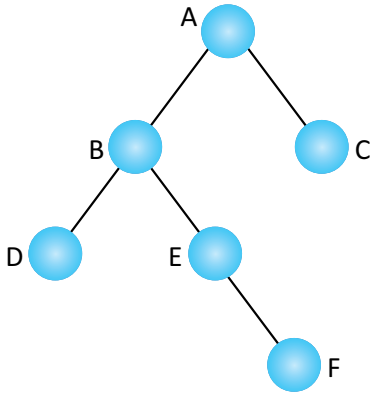


Figura 4 - Árvore binária com 6 nós e profundidade 3
Fonte: o autor.

Vamos simular a inserção da árvore representada pela Figura 4 num vetor de 16 posições. O primeiro nó a ser inserido será a raiz da árvore **A** e ocupará a posição **0** do vetor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A															

Como nosso objetivo é manter a árvore indexada dentro do vetor, vamos reservar a primeira posição **p** logo após o **nó** para armazenar o filho **esquerdo** e a segunda posição **p** para o filho **direito**. Assim, para um nó armazenado numa posição **p** qualquer, seu filho esquerdo estará na posição **p+1** e seu filho direito na posição **p+2**. Vamos inserir os nós **B** e **C** no vetor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C													

O nó **A** está na posição **p=0**, então é esperado que seu filho esquerdo esteja na posição **p+1=1** e o filho direito na posição **p+2=2** do vetor, o que é verdade. Vamos inserir agora os nós **D** e **E**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E											

Usando a fórmula que foi proposta, para encontrar os filhos de **B** na posição **1** do vetor, o filho esquerdo estará na posição **p+1=2** e o direito na posição **p+2=3**, o que não é verdade, logo a fórmula que propusemos não serve para resolver o problema de ordenação de árvore binária em vetor de dados.

Uma árvore binária cresce de forma geométrica posto que cada nó tem dois filhos, que por sua vez, são duas subárvores que podem estar vazias ou não. Independentemente de o filho existir, seu espaço precisa ser reservado no vetor.

Como todo nó em uma árvore binária tem dois filhos, vamos modificar a fórmula para **2*p+1**, para o filho esquerdo, e **2*p+2**, para o filho direito, sendo **p** a posição do nó no vetor.

Aplicando as novas fórmulas para o nó **A** que está na posição **0,** temos que **2*p+1=1** é a posição de **B** (filho esquerdo de **A**) e **2*p+2=2** é a posição de **C** (filho direito de **A**).

Para **B,** que está na posição **1**, temos que **2*p+1=3,** que é a posição de **D** (filho esquerdo de **B**) e **2*p+2=4** é a posição de **D** (filho direito de **B**).

O nó **C** é uma folha, podemos dizer que ele não tem filhos ou que seus filhos são árvores vazias. Como o nosso objetivo é manter o vetor ordenado, a posição dos filhos de **C** será reservada aplicando a fórmula proposta. Assim a posição **2*p+1=5** ficará disponível para o filho **esquerdo** de **C** e **2*p+2=6,** para o seu filho **direito**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-									

O próximo será **D**, que também é uma folha. Então será reservado no vetor as posições **2*p+1=7** e **2*p+2=8**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	-	-							

O último nó de nível 2 (**E**) possui um filho direito (**F**), que será armazenado na posição **2*p+2=10** do vetor criado para a nossa árvore binária.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	-	-	-	F					

No primeiro momento, podemos pensar que essa não é uma boa abordagem. Se a árvore não for binária completa, existirão vários espaços vazios no vetor, lembrando uma memória fragmentada. Por outro lado, estaremos ocupando exatamente a mesma quantidade de memória que a implementação anterior.

A principal diferença é que nesse modelo os nós da árvore estarão indexados, assim é possível obter as informações de forma rápida e precisa, utilizando-se do índice ao invés de percorrer toda a estrutura.

REFLITA

Uma árvore binária também pode ser criada dinamicamente. Basta ter um ponteiro **esquerdo** e **direito** que apontará para os filhos e um ponteiro para o **pai**.

CONSIDERAÇÕES FINAIS

Nesta unidade, foi apresentada ao aluno uma nova estrutura de dados: a árvore binária. Ela lembra muito um grafo, porém essas duas estruturas se diferem pelos seguintes motivos:

1) O grafo pode não ter nenhuma aresta e contém no mínimo um único vértice.

2) Uma árvore pode ser vazia e todos os nós podem ser de no máximo grau 3, sendo que cada nó tem um único pai e dois filhos.

Vimos duas formas de representar essa poderosa estrutura pelo uso de vetores. No primeiro modelo o vetor vai sendo preenchido à medida que a árvore é lida. Cada nó entra na árvore na primeira posição livre da variável.

A segunda implementação possui uma abordagem focada em indexar os nós dentro do vetor, posicionando-os de acordo com a sua ascendência e descendência.

A importância da estrutura de árvore binária e da sua representação indexada ficará clara na próxima unidade, quando tratarmos dos mais usuais métodos de pesquisa em memória.

ATIVIDADES

1. Faça a implementação em linguagem C de uma estrutura de árvore binária para a criação de uma árvore dinâmica seguindo os seguintes passos:

   a) Crie um atributo para armazenar o valor do nó.

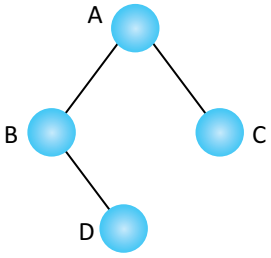
   b) Crie um ponteiro para o pai.

   c) Crie um ponteiro para o filho esquerdo.

   d) Crie um ponteiro para o filho direito.

2. Para cada árvore abaixo, identifique os seus componentes seguindo o exemplo da letra a.

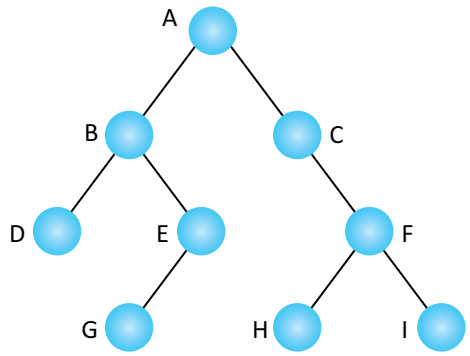
   a)



Raiz	A
Nós	A, B, C, D
Folhas	C, D
Com um filho	B
Com dois filhos	A
Nível 0	A
Nível 1	B, C
Nível 2	D
Nível 3	
Profundidade	2

ATIVIDADES

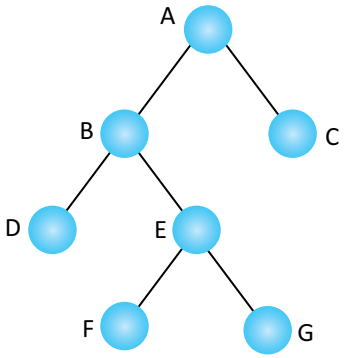
b)



Raiz	
Nós	
Folhas	
Com um filho	
Com dois filhos	
Nível 0	
Nível 1	
Nível 2	
Nível 3	
Profundidade	

ATIVIDADES

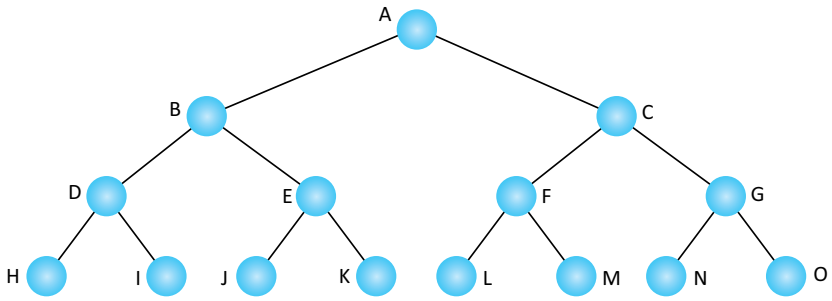
c)



Raiz	
Nós	
Folhas	
Com um filho	
Com dois filhos	
Nível 0	
Nível 1	
Nível 2	
Nível 3	
Profundidade	

ATIVIDADES

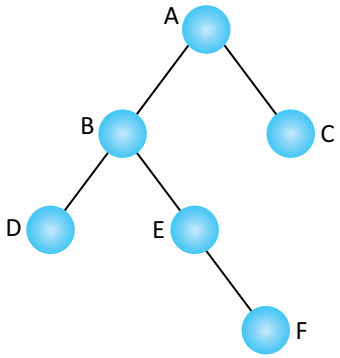
d)



Raiz	
Nós	
Folhas	
Com um filho	
Com dois filhos	
Nível 0	
Nível 1	
Nível 2	
Nível 3	
Profundidade	

ATIVIDADES

e)



Raiz	
Nós	
Folhas	
Com um filho	
Com dois filhos	
Nível 0	
Nível 1	
Nível 2	
Nível 3	
Profundidade	

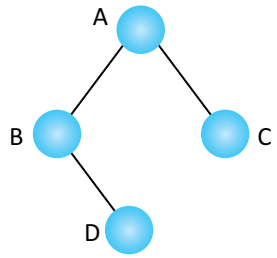
3. O que uma árvore precisa para ser considerada uma árvore estritamente binária?

4. Quais são as características de uma árvore binária completa?

ATIVIDADES

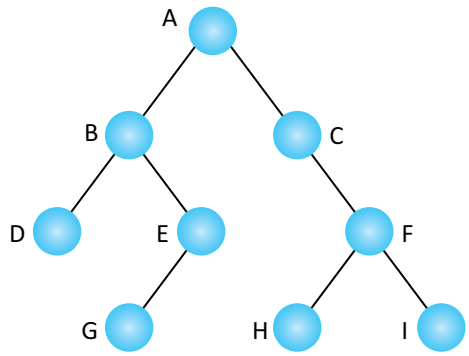
5. Para cada árvore dada, preencha o seu vetor de forma indexada, conforme o exemplo da letra a.

a)



0	1	2	3	4	5	6	7	8	9
A	B	C		D					

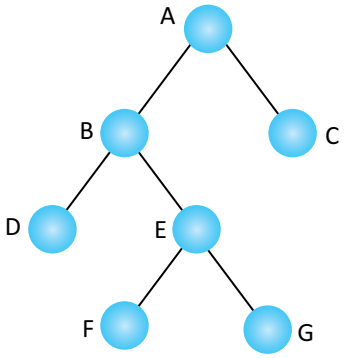
b)



0	1	2	3	4	5	6	7	8	9
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

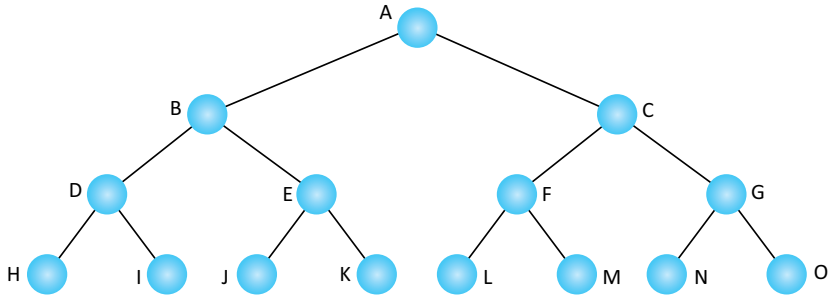
ATIVIDADES

c)



0	1	2	3	4	5	6	7	8	9
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

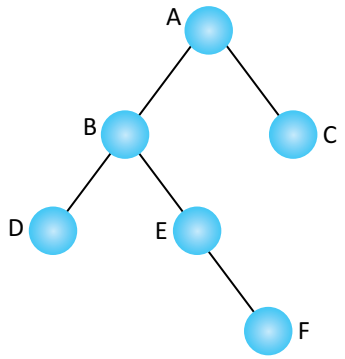
d)



0	1	2	3	4	5	6	7	8	9
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

ATIVIDADES

e)



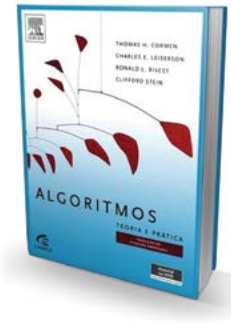
0	1	2	3	4	5	6	7	8	9
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

MATERIAL COMPLEMENTAR

LIVRO

Algoritmos

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein



Editora: Elsevier

Sinopse: O livro apresenta um texto abrangente sobre o moderno estudo de algoritmos para computadores. É uma obra clássica, cuja primeira edição tornou-se amplamente adotada nas melhores universidades em todo o mundo, bem como padrão de referência para profissionais da área.

Nesta terceira edição, totalmente revista e ampliada, as mudanças são extensivas e incluem novos capítulos, exercícios e problemas; revisão de pseudocódigos e um estilo de redação mais claro. A edição brasileira conta ainda com nova tradução e revisão técnica do Prof. Arnaldo Mandel, do Departamento de Ciência da Computação do Instituto de Matemática e Estatística da Universidade de São Paulo.

Elaborado para ser ao mesmo tempo versátil e completo, o livro atende a alunos dos cursos de graduação e pós-graduação em algoritmos ou estruturas de dados.

Professor Me. Pietro Martins de Oliveira
Professor Me. Rogério de Leon Pereira

OPERAÇÕES SOBRE ÁRVORES BINÁRIAS

UNIDADE II

Objetivos de Aprendizagem

- Conhecer técnicas de caminhamento em árvores binárias.
- Aprender a diferenciar os diferentes percursos em árvores binárias.
- Ter contato com técnicas para se balancear uma árvore binária.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Caminhamento em Árvores Binárias
- Percurso Pré-ordem
- Percurso Em-ordem
- Percurso Pós-ordem
- Busca em Árvores Binárias
- Árvores AVL

Unicesumar

INTRODUÇÃO

Nesta unidade, veremos como podemos realizar certas operações em árvores binárias para solucionar problemas clássicos. Primeiramente, veremos como é possível gerar diferentes ordens de visitação nos nós de uma árvore por meio das técnicas de caminhamento em árvores binárias, a saber: caminhamento Pré-ordem; Em-ordem: e Pós-ordem.

Em seguida, veremos como podemos criar uma árvore binária formatada especificamente para realizar a busca de elementos. Também teremos uma noção de como podemos manter tal árvore balanceada por meio do algoritmo de Adelson-Velskii e Landis.

CAMINHAMENTO EM ÁRVORES BINÁRIAS

Em determinadas ocasiões, dependendo dos requisitos de uma aplicação, pode ser preciso percorrer todos os elementos de uma árvore para, por exemplo, exibir todo o seu conteúdo ao usuário. De acordo com a ordem de visitação dos nós, o usuário pode ter visões distintas de uma mesma árvore.

Imagine que, para percorrer uma árvore, tomemos o nó raiz como nó inicial e, a partir dele, comecemos a visitar todos os nós adjacentes a ele para, só então, começar a investigar os outros nós da árvore. Por outro lado, imagine que tomemos um nó folha como ponto de partida e caminhemos em direção à raiz, visitando apenas o ramo da árvore que leva o nó folha à raiz. São maneiras distintas de se visualizar a mesma árvore.

Dito isso, vamos a alguns algoritmos clássicos que ditam as regras de visitação de nós individuais em uma árvore. Ao final da execução de cada um dos algoritmos a seguir, é esperado que sequências de visitação distintas sejam geradas. Para isso, considere que nossas árvores são compostas por registros dinâmicos em C, nos quais cada nó possui ao menos um ponteiro para a subárvore esquerda e outro ponteiro para a subárvore direita.

PERCURSO PRÉ-ORDEM

O caminhamento pré-ordem, também conhecido por caminhamento pré-fixado, marca primeiramente a raiz como visitada, e só depois visitamos as subárvores esquerda e direita, respectivamente. O Programa 2.1, a seguir, apresenta um código-fonte no qual a função **preOrdem()** implementa a lógica semântica necessária para fazer com que, a partir do parâmetro raiz, o programa realize o respectivo caminhamento em uma árvore binária.

Programa 2.1 - Função preOrdem()

```
1 void preOrdem(struct NO* raiz){
2       if(raiz){
3             printf("%d \t", raiz->dado); //visita o nó atual
4             preOrdem(raiz->esq);
5             preOrdem(raiz->dir);
6       }
7 }
```

Considere o caminhamento pré-ordem na árvore da Figura 1, na qual a raiz é o nó **F**.

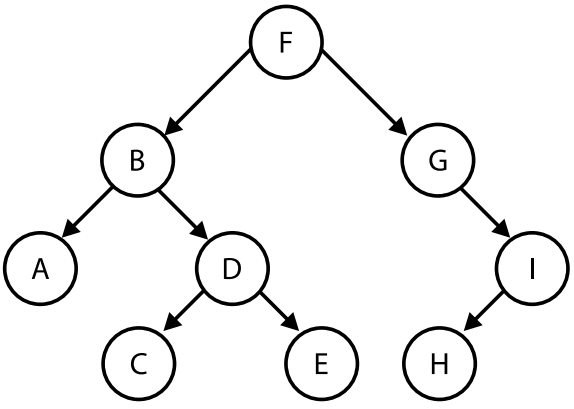


Figura 1 - Árvore binária
Fonte: o autor.

A ordem de visitação produzida pela função **preOrdem()**, do Programa 2.1, levando em conta o nó inicial **F**, seria a seguinte: **F, B, A, D, C, E, G, I, H**.

PERCURSO EM-ORDEM

No caminhamento em-ordem, também conhecido por caminhamento interfixado, primeiramente visitamos toda a subárvore esquerda e, só então, a raiz é marcada como visitada. Em seguida, o método em-ordem faz a visitação de toda a subárvore direita. O Programa 2.2, a seguir, apresenta um código-fonte no qual a função **emOrdem()** implementa a lógica para fazer com que, a partir da raiz, o programa execute o caminhamento em-ordem em uma árvore binária.

Programa 2.2 - Função emOrdem()

```
1  void emOrdem(NO* raiz){
2         if(raiz){
3                emOrdem(raiz->esq);
4                printf("%d \t", raiz->dado); //visita o nó atual
5                emOrdem(raiz->dir);
6         }
7  }
```

Considere o caminhamento em-ordem na mesma árvore da Figura 1, vista anteriormente, na qual a raiz é o nó **F**. A ordem de visitação produzida pela função **emOrdem()**, do Programa 2.2, levando em conta o nó inicial **F**, seria a seguinte: **A, B, C, D, E, F, G, H, I**.

PERCURSO PÓS-ORDEM

O último caminhamento que veremos é o método pós-ordem, também conhecido por caminhamento pós-fixado. Nesse caso, primeiramente visitamos toda a subárvore esquerda, depois, toda a subárvore direita. Só após ter visitado as duas **subárvores**, é que marcamos o nó corrente como visitado. O Programa 2.3 a seguir apresenta a função **posOrdem()**, que faz com que, a partir da raiz, o programa execute o caminhamento pós-ordem em uma árvore binária.

Programa 2.3 - Função posOrdem()

```
1  void posOrdem(NO* raiz){
2         if(raiz){
3                posOrdem(raiz->esq);
4                posOrdem(raiz->dir);
5                printf("%d \t", raiz->dado); //visita o nó atual
6         }
7  }
```

Considerando o caminhamento pós-ordem na árvore já conhecida, da Figura 1, a ordem de visitação produzida pela função **posOrdem()**, do Programa 2.3, levando em conta o nó inicial **F**, seria a seguinte: **A, C, E, D, B, H, I, G, F**.

BUSCA EM ÁRVORES BINÁRIAS

Na unidade passada foi dito que árvores binárias são muito utilizadas para organizar informações na memória devido ao seu grande potencial de busca em um tempo relativamente curto. Para isso precisamos criar uma árvore binária de busca.

Você já sabe como criar um vetor a partir de uma árvore dada. Vamos fazer exatamente o contrário. A partir de um vetor vamos construir uma árvore binária com regras específicas para que ela se torne uma árvore binária de busca.

Uma árvore binária ou é vazia ou tem pelo menos um nó **raiz**. Todo nó tem um **pai** (menos a raiz) e no máximo dois filhos, um **esquerdo** e um **direito**. Tanto o filho esquerdo como direito podem ser **subárvores** vazias.

Agora vamos adicionar uma nova regra para a construção da árvore. O nó raiz será o valor que estiver na posição do meio de um vetor (ordenado ou não). Ao adicionar um novo nó na árvore, verificamos se ele é menor do que a raiz. Caso seja verdade, ele será adicionado na subárvore esquerda, caso contrário, na subárvore direita. Faremos uma simulação usando como base um vetor ordenado representado na Figura 2. A primeira linha da figura é o índice do vetor, que vai de 0 a 9. A segunda linha possui os valores de cada posição do vetor.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Figura 2 - Vetor para construção de uma árvore binária

A posição central do vetor é encontrada pela fórmula:

$$meio = \frac{(menor + maior)}{2}$$

Substituindo-se os valores temos:

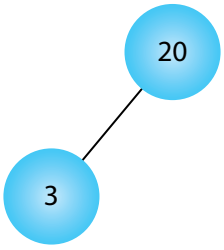
$$meio = \frac{(0+9)}{2}$$

$$meio = \frac{(9)}{2}$$

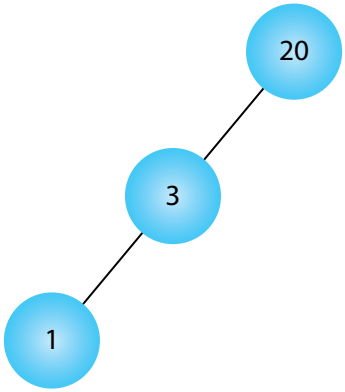
$$meio = 4,5$$

Como os valores de índices de um vetor são inteiros, **meio** será **4**. O valor do vetor na posição 4 (**vec[4]**) será a raiz da árvore.

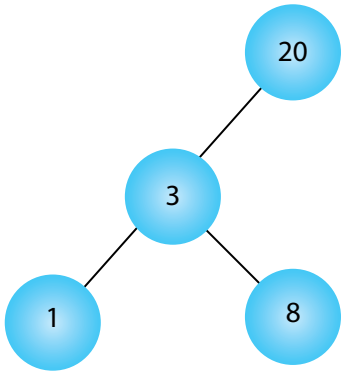
Agora que a nossa árvore possui uma raiz, vamos adicionar o primeiro elemento do vetor na árvore. Como **vec[0]** é menor do que a raiz, seu valor será adicionado na subárvore esquerda.



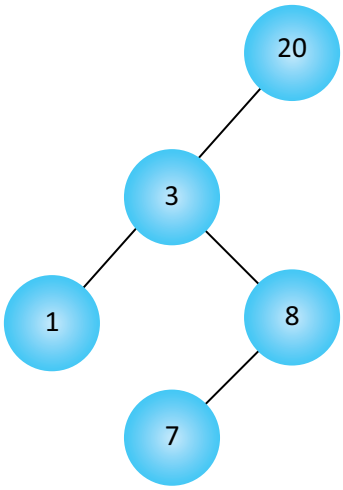
O próximo valor **vec[1]** também é menor do que a **raiz**, mas a raiz já possui uma subárvore esquerda. Fazemos então uma nova comparação com a raiz da subárvore esquerda. Como **vec[1]** é menor do que a raiz dessa subárvore, ele será adicionado como seu filho esquerdo.



Dando sequência, vamos verificar o valor contido em **vec[2]**. Seu valor é menor do que **20**, então iremos verificar com o filho esquerdo da raiz. O valor de **vec[2]** é maior do que **3**, então ele entrará na árvore como seu filho direito.



Agora é a vez de **vec[3]**, que tem valor **7**. Percorreremos a árvore em formação respeitando as regras da árvore binária de busca e um novo nó será adicionado como filho esquerdo de **8**.



Chegamos agora à metade do vetor da Figura 2. Precisamos de mais cinco interações para finalizar a construção da árvore. A Figura 3 apresenta a árvore final após a inserção dos nós de valores 21, 31, 40, 30 e 0.

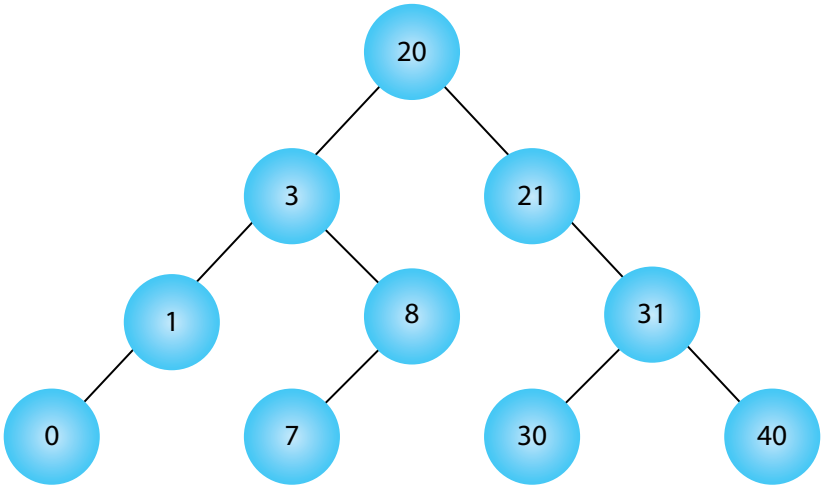


Figura 3 - Uma Árvore Binária de Busca

Agora vem o que nos interessa, que é a realização da busca. Dado um argumento qualquer, se ele for menor do que a **raiz**, ou ele não existe, ou ele se encontra na sua subárvore **esquerda**. Se o valor for maior do que a **raiz**, ou ele não existe ou está na sua subárvore **direita**. A partir da **raiz** vamos descendo pela árvore binária de busca até que o valor seja encontrado ou que encontremos uma **folha** ou uma **subárvore vazia**.

A Figura 4 apresenta uma busca na árvore pelo valor **7**. A **raiz** tem valor **20**, então a busca prossegue na sua subárvore **esquerda**. Como **7 > 3**, a busca continua pela subárvore **direita** do nó **3**. Como **7 < 8,** a busca desce pela subárvore **esquerda** do nó **8** até que o valor **7** seja encontrado.

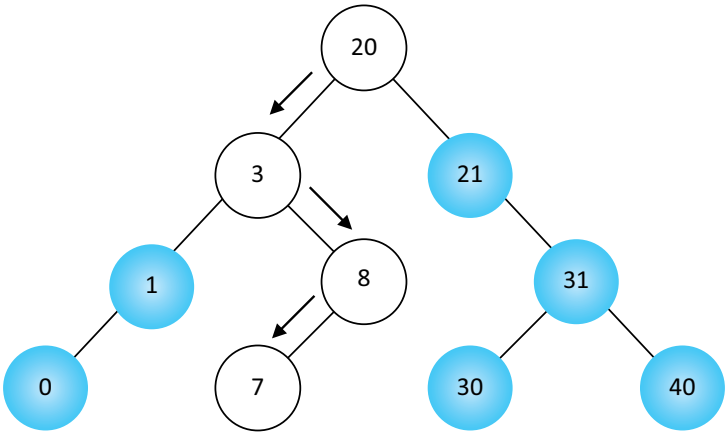


Figura 4 - Busca do valor 7 na árvore binária

Essa técnica se assemelha muito à **busca binária**. A principal diferença é que a árvore binária de busca pode ser implementada dinamicamente na memória, pois não precisamos descobrir onde fica a metade da tabela. Ao invés de dividir o vetor pela metade, ignoramos uma das subárvores para continuar a pesquisa. Esse método é extremamente rápido e eficiente em arquivos com grandes quantidades de registros.

O Programa 2.4 demonstra um algoritmo em linguagem C que encontra um valor por meio de uma árvore binária de busca em um vetor de dados. A função **buscaArvoreBinaria()** recebe três parâmetros: o vetor **vec** a ser pesquisado, o argumento **arg** a ser encontrado e o tamanho **tam** do vetor.

O laço principal tem duas regras de parada. O laço continua até o valor ser encontrado, que se dá no momento em que a variável **achou** for diferente de **-1** ou quando chegar ao final do vetor.

Para ficar mais simples a implementação, estamos utilizando um vetor que guarda os nós de forma ordenada de acordo com o nível de cada nó. Assim, a busca começa na raiz e segue para **2*p+1** na direção da árvore esquerda ou **2*p+2** na árvore direita, onde **p** é a posição do nó no vetor.

Programa 2.4 – Árvore de busca binária em C

```
//Função de árvore binária de busca
int buscaArvoreBinaria(int vec[], int arg, int tam){
  int no, achou, fim;
  fim = 0;
  no = 0;
  achou = -1;
  while((achou == -1) && (fim <= tam)){
    if (arg == vec[no]){
      achou = no;
    }
    if (arg < vec[no]){
      no = (2 * no) + 1;
    }
    else {
      no = (2 * no) + 2;
    }
    fim++;
  }
  return(achou);
}
```


ÁRVORES AVL

A forma como os elementos são inseridos em uma árvore binária de busca pode fazer com que a busca se torne altamente ineficiente. Considere o conjunto de dados a seguir:

2 - 5 - 10 - 17 - 25 - 32

Se inserirmos tais dados, do primeiro até o último, em uma árvore de busca binária, teremos a árvore estruturada da seguinte forma:

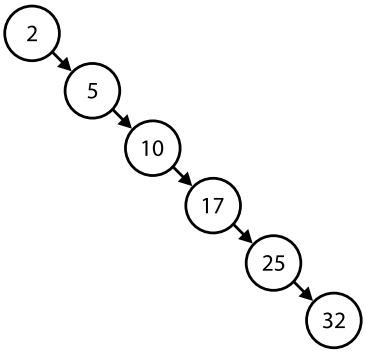


Figura 5 - Árvore de busca binária desbalanceada
Fonte: o autor.

Repare que a estrutura de dados se assemelha muito mais a uma lista linear encadeada, do que com uma árvore binária de busca. Em consequência, operações de busca nessa estrutura de dados serão feitas de maneira sequencial, ao invés de aproveitarem-se da capacidade de ignorar subárvores, cortando caminho na busca. Dizemos que a árvore da Figura 5, acima, encontra-se desbalanceada.

A partir daí, surge o conceito de balanceamento. Dizemos que uma árvore balanceada tende a manter sua altura tão pequena quanto possível, à medida em que são realizadas novas inserções ou remoções de dados. Considerando o mesmo conjunto de dados que vimos anteriormente, poderíamos tentar balancear a árvore binária de busca, resultando em algo parecido com o seguinte:

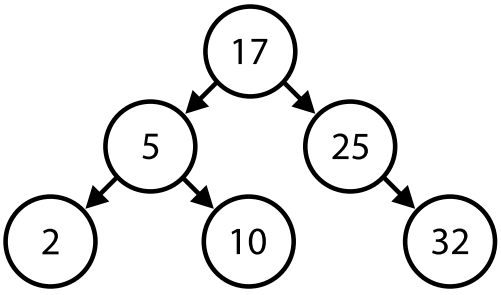


Figura 6 - Árvore de busca binária balanceada
Fonte: o autor.

O conceito de balanceamento está relacionado à altura das subárvores que compõem uma árvore binária. É importante lembrar que a altura (ou profundidade) de uma subárvore é igual ao número de nós visitados desde a raiz até o nó folha mais distante (uma subárvore vazia possui altura -1, por definição). Dizemos que um nó está balanceado caso o valor absoluto da diferença entre as alturas das subárvores esquerda e direita seja menor ou igual a 1. Chamaremos de fator de balanceamento (Fb) o resultado da diferença entre a altura da subárvore esquerda (He) de um nó pela altura da subárvore direita (Hd) do mesmo nó, de acordo com a seguinte fórmula:

$$Fb = He - Hd$$

Vamos a alguns exemplos. A árvore da Figura 7 encontra-se balanceada. Nessa árvore, ilustramos a altura (H) de cada subárvore na porção superior esquerda de cada nó, e o respectivo fator de balanceamento (Fb) no canto superior direito de cada nó. Além disso, as raízes de subárvores vazias estão destacadas como quadrados pontilhados. Repare como cada fator de balanceamento (destacados ao lado superior direito de cada nó) é o resultado do valor da altura da subárvore esquerda menos a altura da subárvore direita.

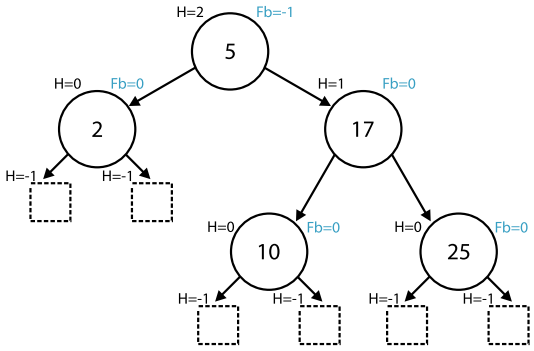


Figura 7 - Fatores de balanceamento com todos os nós balanceados
Fonte: o autor.

Todavia existem situações nas quais uma árvore de busca possui nós desbalanceados. Por exemplo, imagine que adicionemos um nó cujo valor é igual a 32. Seguindo as regras de inserção para uma árvore binária de busca, tal nó seria adicionado como filho à direita do nó de valor igual a 25. Com isso, temos uma nova árvore, como ilustra a Figura 8.

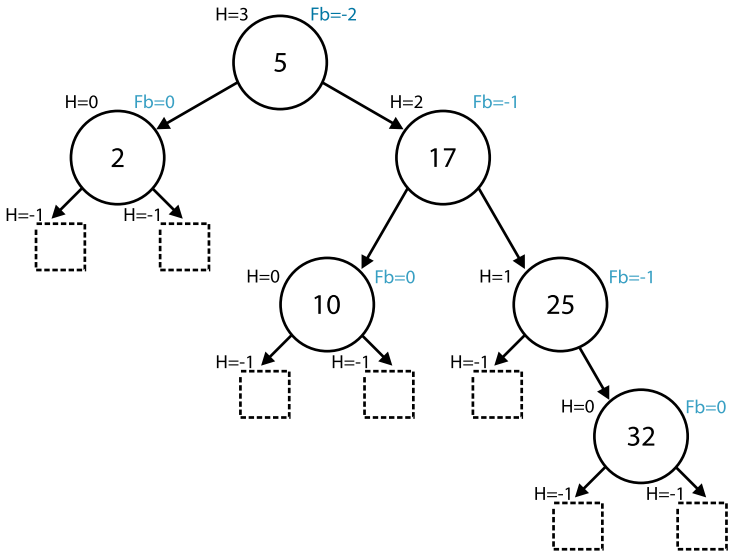


Figura 8 - Fatores de balanceamento com um nó desbalanceado
Fonte: o autor.

Analisando a Figura 8, notamos que a inserção do nó de valor 32 fez com que a raíz da árvore perdesse seu balanceamento. Note que a altura da subárvore esquerda, em relação à raiz, é igual a 0, porém a altura da subárvore direita é igual a 2. Com isso, o fator de balanceamento da raiz é igual a -2 que, em valor absoluto, é maior do que 1. Assim, pela definição, dizemos que o nó de valor igual a 5 encontra-se desbalanceado.

Para resolver o problema do desbalanceamento de árvores binárias de busca, os pesquisadores Adelson-Velskii e Landis, em 1962, criaram um algoritmo que leva as iniciais de seus nomes. As árvores AVL são, nesse sentido, árvores nas quais todos os nós encontram-se balanceados. Para cada nó, a diferença entre as alturas de suas subárvores não pode ser igual ou superior a 2, em valor absoluto.

Como vimos, uma árvore balanceada pode perder essa característica quando um novo elemento é inserido. Isso também pode ocorrer quando um elemento é removido da árvore binária. Assim, quando ocorrem operações de inserção ou remoção em árvores AVL, recalcula-se os fatores de balanceamento de cada nó para, assim, poder executar rotações nos nós problemáticos, na tentativa de restabelecer seu balanceamento. Existem, basicamente, quatro tipos de rotações: dois tipos de rotações simples e dois tipos de rotações duplas.

De acordo com Adam Drozdek (2008), quando uma árvore AVL se desbalanceia inserindo-se um nó na subárvore que se encontra à direita do filho direito, é necessário realizar uma rotação simples. Simetricamente, se inserimos um nó na subárvore esquerda do filho esquerdo, é preciso realizar uma rotação simples no sentido oposto ao anterior.

Dessa forma, podemos observar que, no exemplo da Figura 8, foi realizada uma inserção na subárvore direita do filho direito em relação ao nó desbalanceado. Ou seja, o nó 32 foi inserido na subárvore direita do nó 17, que é filho à direita do nó 5. Para tornar a árvore balanceada, precisamos realizar uma rotação, conforme o ilustrado na Figura 9.

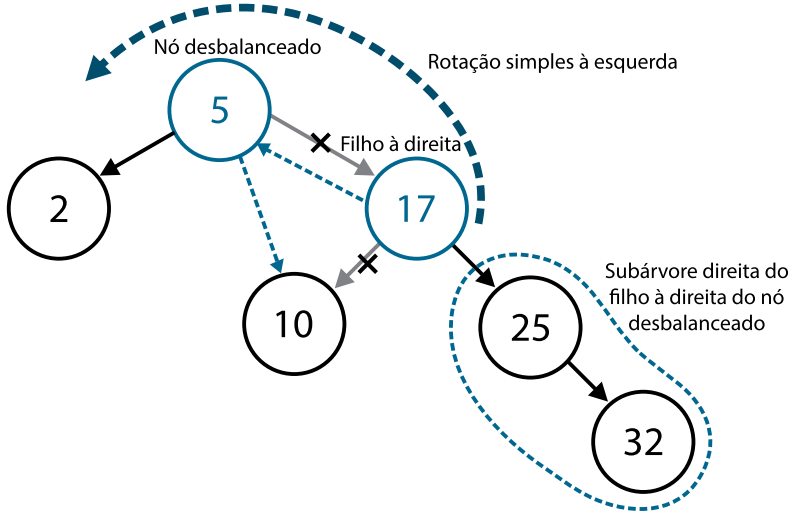


Figura 9 - Corrigindo uma inserção na subárvore direita do filho à direita com rotação simples à esquerda
Fonte: o autor.

Para realizarmos a rotação à esquerda, precisamos "puxar" o nó 17 para cima, fazendo com que o nó 5 passe a ser o filho à esquerda do nó 17. Além disso, a subárvore com raiz no nó 10 é "deserdada" pelo nó 17, e o nó 5 a "adota", fazendo com que o nó 10 passe a ser o filho à direita do nó 5, deixando de ser o filho à esquerda do nó 17. Dessa forma, após realizada a rotação, devemos recalcular a altura de cada subárvore, bem como os fatores de balanceamento de cada nó. Nesse caso, a árvore balanceada resultante pode ser visualizada na Figura 10.

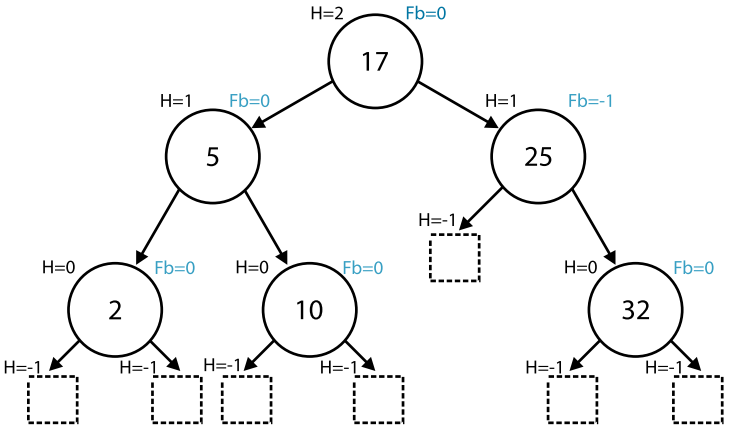


Figura 10 - Árvore AVL balanceada após uma inserção
Fonte: o autor.

A partir dessa nova árvore balanceada, iremos adicionar um nó de valor igual a 1. Para isso, devemos continuar seguindo a regra de inserção em árvore de busca binária. Por isso, temos de percorrer da raiz 17 em direção a uma folha, até poder inserir o 1 à esquerda do nó 2. Dessa forma, temos uma nova árvore, com novas alturas e fatores de balanceamento, como podemos observar na Figura 11. Repare que a árvore não está desbalanceada em nó algum.

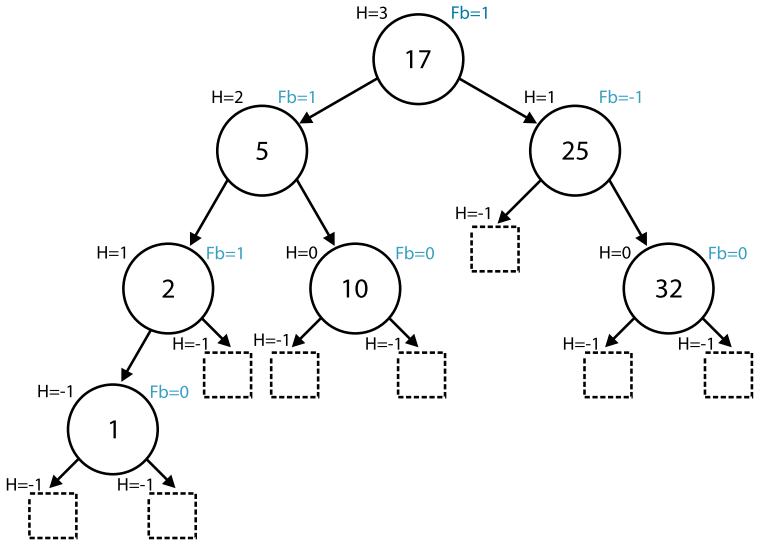


Figura 11 - Inserindo o nó 1
Fonte: o autor.

Sempre que inserimos um nó, devemos checar se todos os nós percorridos desde a raíz até o nó folha estão balanceados. Como podemos observar na Figura 11, os nós percorridos foram, nesta ordem: 17, 5 e 2. É possível reparar que, após a inserção, foram recalculados os fatores de balanceamento Fb de tais nós e nenhum deles se tornou desbalanceado. Repare, ainda, que somente os nós 17, 5 e 2 tiveram seus fatores de balanceamento alterados. Ou seja, isso reforça o fato de que apenas os nós percorridos devem ser checados à procura de desbalanceamentos.

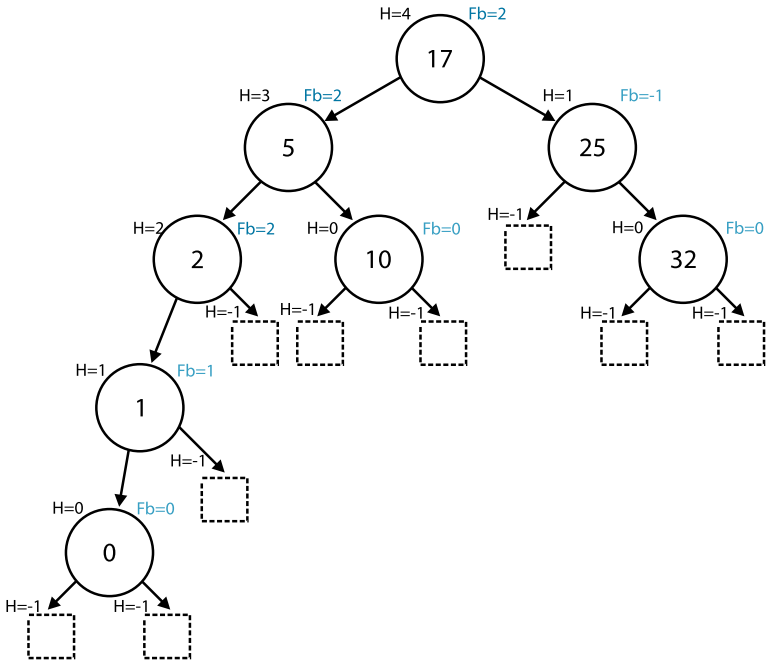


Figura 12 - Inserindo o nó 0

Note, agora, que na Figura 12 temos a inserção do nó de valor igual a 0. Para adicionar o nó 0, tivemos de percorrer o nó 17, o 5, o 2 e o 1, para finalmente posicionar o 0 à esquerda do nó 1. Ao recalcular os fatores de balanceamento, chegamos à conclusão de que os nós 2, 5 e 17 estão desbalanceados. A partir dessas informações temos condições de tentar consertar nossa árvore, de baixo para cima. Novamente, devemos dar atenção ao fato de que apenas os nós percorridos durante a inserção tiveram chance de se tornar desbalanceados, pois somente suas subárvores tiveram as respectivas alturas alteradas após a inserção.

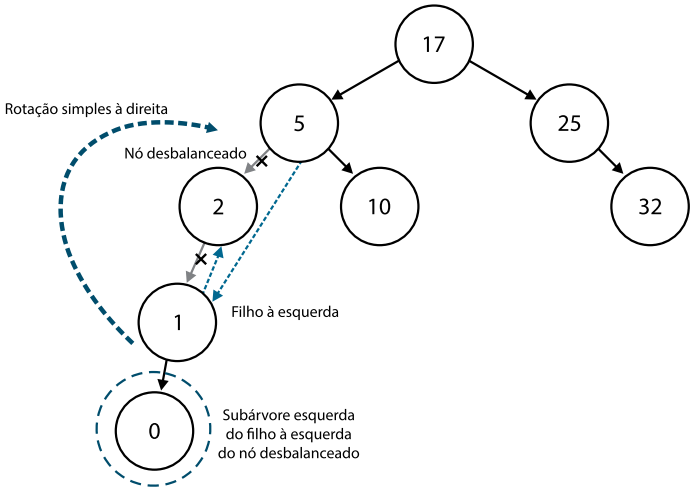


Figura 13 - Corrigindo uma inserção na subárvore esquerda do filho à esquerda com rotação simples à direita

De maneira simétrica à situação que levou à rotação feita anteriormente, temos o caso oposto. Agora realizamos a inserção de um nó na subárvore esquerda do filho à esquerda do nó desbalanceado. Para resolver esse desbalanceamento, devemos realizar uma rotação simples para a direita, fazendo com que o nó 5 passe a ser o pai do nó 1 que, por sua vez, passa a ser o filho à esquerda do nó 5. O nó 2 deixa de ser filho do nó 5 e passa a ser o filho à direita do nó 1. Caso o nó 1 tenha alguma subárvore à direita, a raiz dessa subárvore passa a ser filha à esquerda do nó 2 (nó 2 "adota" o filho órfão do nó 1). O resultado dessa rotação pode ser observado na Figura 14.

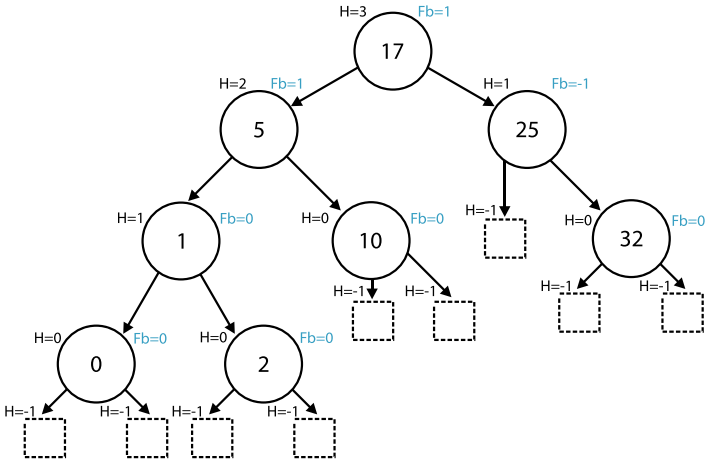


Figura 14 – Balanceamento de árvore AVL após rotação simples à direita
Fonte: o autor.

A Figura 14 nos mostra que, realizando uma rotação simples no primeiro nó desbalanceado, partindo do nó recém inserido em direção à raiz, balanceou-se toda a árvore. Todavia em situações nas quais os nós ascendentes continuam desbalanceados após uma rotação, é preciso continuar corrigindo a árvore, até que o todos os nós da folha até a raiz sejam balanceados.

Agora, considere outra árvore, na qual os nós 17, 32 e 25 foram inseridos na árvore de busca binária, na respectiva ordem. Com isso, temos a raiz 17 se tornando desbalanceada, como podemos ver na Figura 15. Contudo, a quebra no balanço foi realizada por meio de uma inserção na subárvore esquerda do filho direito em relação ao nó desbalanceado. Ou seja, o número 25 foi inserido à esquerda do nó 32 que, por sua vez, está à direita do nó 17.

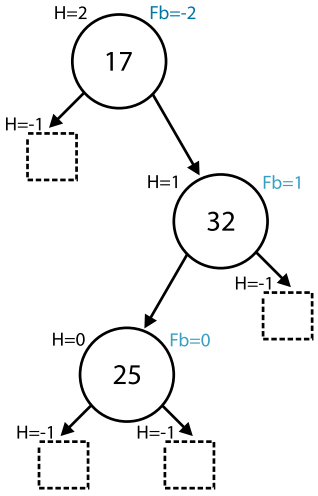


Figura 15 – Inserção na subárvore esquerda do filho à direita do nó desbalanceado
Fonte: o autor.

Em uma situação como essas, uma única rotação simples em torno do nó desbalanceado não resolve o problema. Isso ocorre pois, ao tentar realizar uma única rotação à esquerda, continuaríamos tendo o nó raiz, 17, desbalanceado. Nessas situações é preciso empregar a rotação dupla direita-esquerda, que é feita em duas etapas. Nesse caso: primeiramente rotaciona-se à direita a subárvore com raiz em 25, em direção ao nó 32 (Figura 16 (a) e (b)); num segundo momento, rotacionamos à esquerda a subárvore com raiz em 25, em direção ao nó 17 (Figura 16 (c) e (d)), como podemos ver na Figura 16.

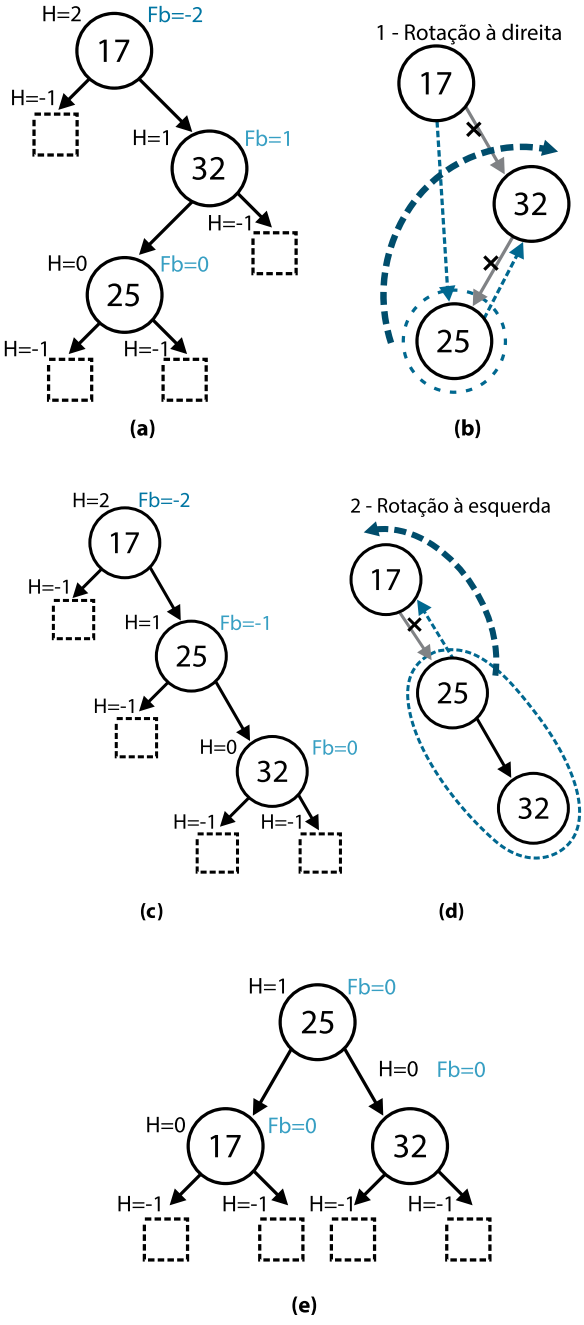


Figura 16 - Corrigindo uma inserção na subárvore esquerda do filho à direita com rotação dupla esquerda-direita

Fonte: o autor.

Por outro lado, temos uma situação simétrica à que foi recém apresentada, que também é resolvida com uma rotação dupla. Considere, agora, que inserimos os nós 10 e 13, um em seguida do outro, levando à situação ilustrada pela Figura 17.

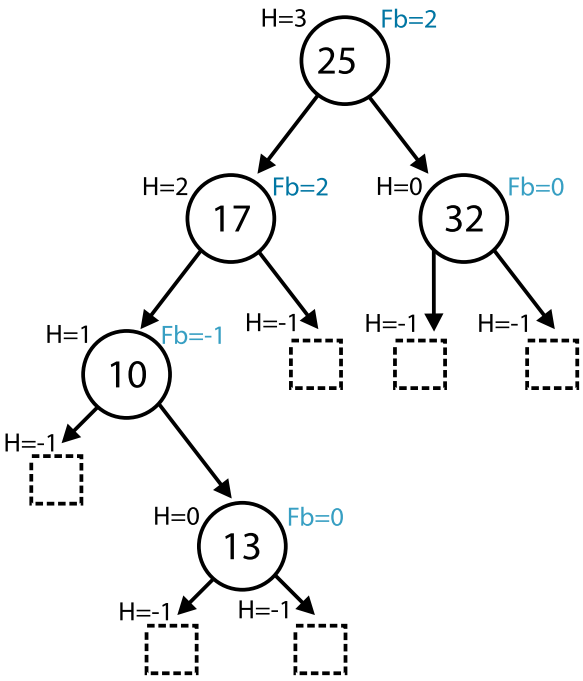


Figura 17 - Inserção na subárvore à direita do filho à esquerda
Fonte: o autor.

Após a inserção do 10 à esquerda do 17, e do 13 à direita do 10, temos uma situação na qual os fatores de balanceamento dos nós 17 e 25 quebram a regra da árvore AVL. Nesse caso, temos uma inserção na subárvore direita do filho esquerdo em relação ao nó desbalanceado. Assim, precisamos nesse caso, realizar rotação dupla esquerda-direita, em duas etapas. Primeiramente fazemos com que o nó 10 se torne o filho à esquerda do nó 13, como ilustrado na Figura 18.

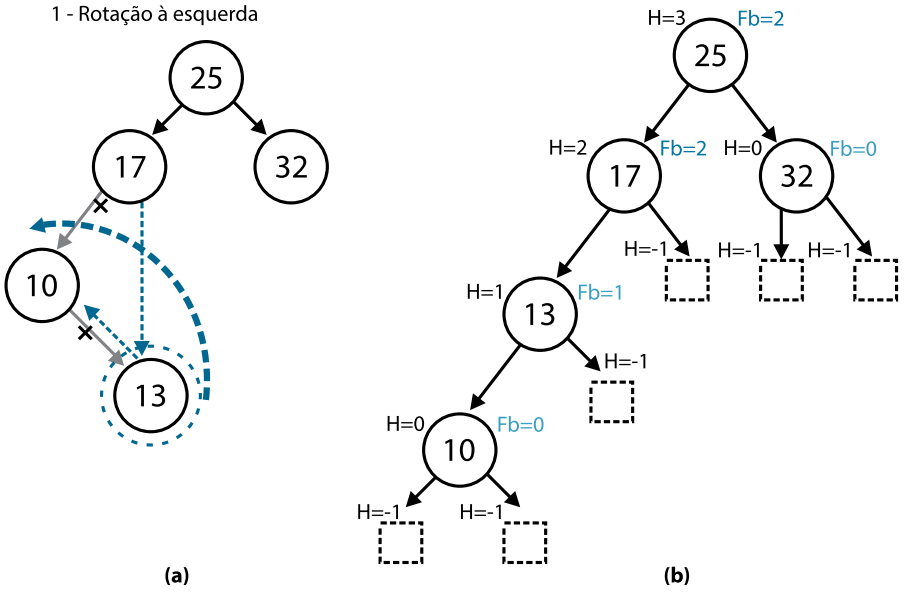


Figura 18 - Rotação dupla - 1 etapa: rotação à esquerda
Fonte: a autor.

Depois, para balancear o nó 17, realizamos uma rotação à direita a partir do nó 13, como podemos visualizar na Figura 19.

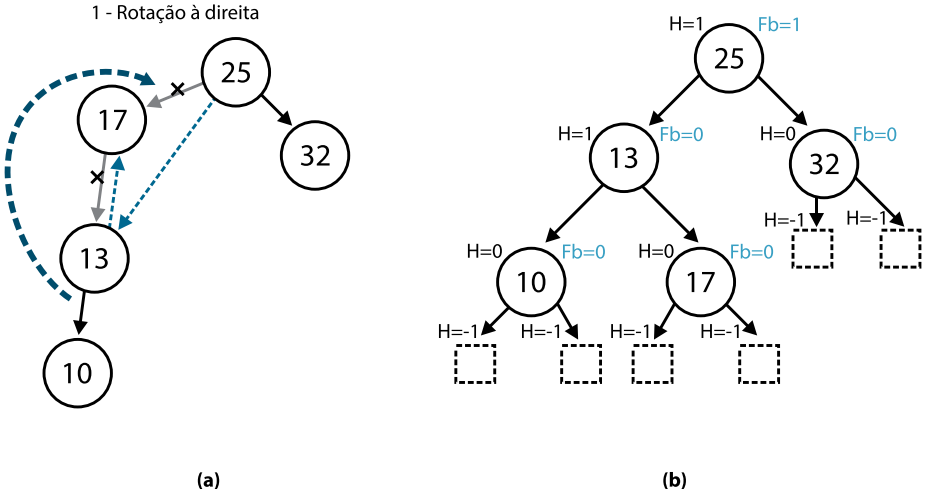


Figura 19 - Rotação dupla - 2 etapa: rotação à direita
Fonte: a autor.

Com isso, temos nossa árvore AVL balanceada novamente. Basta observar os fatores de balanceamento do resultado final.

Dessa forma, abordamos as quatro situações nas quais precisamos balancear nossa árvore AVL, a saber: duas rotações simples e duas rotações duplas. Podemos resumir as quatro situações que exigem as respectivas rotações como o descrito a seguir.

1) Inserção na subárvore direita do filho à direita: solução com rotação simples à esquerda.
2) Inserção na subárvore esquerda do filho à esquerda: solução com rotação simples à direita.
3) Inserção na subárvore esquerda do filho à direita: solução com rotação dupla direita-esquerda.
4) Inserção na subárvore direita do filho à esquerda: solução com rotação dupla esquerda-direita.

Em suma, uma árvore AVL é uma árvore binária de busca na qual o fator de balanceamento de cada um de seus nós não pode ser maior que 1 ou menor que -1. Caso o fator de balanceamento de um ou mais nós seja maior ou igual a 2, em valores absolutos, é preciso analisar a ordem das inserções para aplicar a rotação mais adequada para corrigir o balanceamento de cada um dos nós. Isso faz com que buscas em uma árvore AVL sejam mais eficientes.

CONSIDERAÇÕES FINAIS

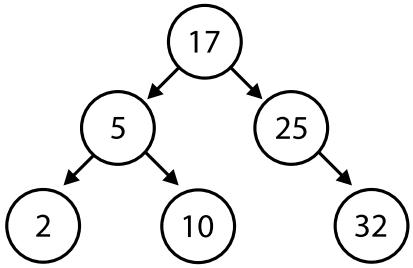
Como vimos, uma árvore binária pode ser percorrida de diversas formas. No percurso pré-ordem, primeiramente visitamos a raiz da arvore, depois tentamos visitar a subárvore esquerda para, só então, visitar a subárvore direita. No percurso em-ordem, primeiro a subárvore esquerda é visitada por completo para, só então, a raiz da árvore ser visitada e, ao fim, a subárvore direita é visitada. O último caminhamento que vimos, o pós-ordem, faz a visitação da subárvore esquerda, depois da subárvore direita para, finalmente, visitar a raiz da árvore.

Também visualizamos como é possível estabelecer regras para criar árvores binárias ordenadas e balanceadas, para que seja possível realizar buscas de maneira otimizada. Para criar uma árvore binária de busca, podemos estabelecer que elementos menores ou iguais à raiz só podem ser inseridos na subárvore esquerda, e elementos maiores na subárvore direita. A partir disso podemos realizar buscas com a mesma lógica.

Além disso, as árvores AVL nos dão ferramentas para rotacionar nós desbalanceados em busca de uma árvore binária tão balanceada quanto possível. Isso otimiza ainda mais os tempos de busca nesse tipo de estrutura de dados.

ATIVIDADES

1. Considere a árvore binária que se segue:



   a) Qual a ordem de visitação dos nós ao se aplicar o percurso pré-ordem na árvore binária acima?

   b) Qual a ordem de visitação dos nós ao se aplicar o percurso em-ordem na árvore binária acima?

   c) Qual a ordem de visitação dos nós ao se aplicar o percurso pós-ordem na árvore binária acima?

2. Considere o seguinte conjunto de dados: 13, 70, 60, 39, 66, 55, 50, 42. Levando em conta a ordem dos elementos recém apresentados, monte uma árvore binária de busca.

   a) A partir da árvore de busca montada, realize uma busca pelo elemento 42 e informe quantos nós foram visitados para se encontrar tal elemento, considerando que a busca parte sempre da raiz da árvore.

3. Considerando os mesmos elementos: 13, 70, 60, 39, 66, 55, 50, 42. Levando em conta a ordem dos elementos recém apresentados, monte uma árvore AVL. Se necessário, realize as rotações simples ou duplas após cada inserção, para garantir o balanceamento.

   a) A partir da árvore de AVL montada, realize uma busca pelo elemento 42 e informe quantos nós foram visitados para se encontrar tal elemento, considerando que a busca parte sempre da raiz da árvore.

MATERIAL COMPLEMENTAR

LIVRO

Estrutura de Dados e Algoritmos em C++
Adam Drozdek

Editora: Cencage Learning
Sinopse: o livro é muito útil para quem gosta de se aprofundar no tema estrutura de dados. A partir dessa obra, o leitor é capaz de ter contato com as mais simples estruturas de dados, até as mais elaboradas, tendo exemplos visuais e também com código-fonte em linguagem C++

NA WEB

Busca em árvore binária
<http://www.youtube.com/watch?v=uAZ1TcZinsk>.

Compreendendo melhor as árvores AVL
<https://www.youtube.com/watch?v=YkF76cOgtMQ&feature=youtu.be>.

Professor Me. Rogério de Leon Pereira

TÉCNICAS DE ORDENAÇÃO

UNIDADE III

Objetivos de Aprendizagem

- Aprender diferentes técnicas de ordenação de dados.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Preparando o ambiente de testes
- Ordenação por Bubblesort (Método da Bolha)
- Ordenação por Selectionsort
- Ordenação por Insertionsort
- Ordenação por Shellsort

Unicesumar

INTRODUÇÃO

A partir daqui, teremos mais contato com vetores que árvores. Em particular, veremos nesta unidade alguns algoritmos que são capazes de organizar vetores de maneira crescente ou decrescente. As motivações para se ordenar um vetor são variadas.

Por exemplo, imagine que a agenda de seu celular não apresenta função de busca por contatos e, para piorar, seus dados estão completamente desordenados. Não seria muito mais fácil organizar seus contatos em ordem alfabética para que nós, humanos, possamos utilizar a agenda de maneira mais adequada?

Imagine, agora, que você tem a lista de várias compras e vendas realizadas diariamente em uma organização. Sua lista encontra-se ordenada de acordo com as datas nas quais cada compra ou venda ocorreu. Seu líder lhe solicita que organize a lista de forma a considerar o nome do comprador/vendedor, em ordem alfabética. Como faríamos para ordenar sua lista?

Por isso, veremos agora alguns algoritmos de ordenação de implementação mais simplificada. Todos os métodos apresentados aqui são capazes de ordenar um conjunto de dados armazenados em um vetor, de maneira exata. Nesta unidade, veremos os algoritmos Bubblesort, Selectionsort, Insertionsort e Shellsort.

PREPARANDO O AMBIENTE DE TESTES

Antes de abordamos as técnicas e algoritmos de ordenação, vamos preparar o nosso ambiente de testes. O objetivo deste livro não o de ensinar a programação em linguagem C. Todo conteúdo parte do pressuposto de que o(a) aluno(a) já saiba programar e tenha conhecimento de nível intermediário da linguagem, saiba conceitos de variáveis e ponteiros, conheça estruturas de dados como listas, pilhas e filas.

Porém, para que o aluno possa colocar na prática o que veremos a seguir, que são as técnicas de ordenação, faz-se necessário que seja criado um programa em que esses algoritmos sejam executados.

Com isso em mente, preparamos o Programa 3.1, a seguir, que traz as ferramentas necessárias para que o aluno possa verificar o conteúdo dessa unidade de forma quase imediata.

Programa 3.1 – Programa criado para servir de ambiente de teste

```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//Constantes
#define tamanho 10

//Variáveis
int lista[tamanho];
int ordenado[tamanho];
int opt=-1;
int qtd;

//Prototipação
void menu_mostrar(void);
void lista_mostrar(void);
void lista_gerar(void);
void lista_ler(void);
void lista_limpar(void);
void lista_mostrar_ordenado(void);

//Função Principal
int main(void){
  srand(time(NULL));
  do {
    system("cls");
    lista_mostrar();
    lista_mostrar_ordenado();
    menu_mostrar();
    scanf("%d",&opt);
    switch (opt){
      case 1:
        lista_gerar();
```

```
      break;
      case 2:
        lista_ler();
      break;
    }
  }while(opt!=0);
  system("pause");
  return(0);
}

//Mostra o conteúdo da lista
void lista_mostrar(void){
  printf("[  ");
  for (int i = 0; i < tamanho; i++ ){
    printf("%d  ",lista[i]);
  }
  printf("]\n\n");
}

//Mostra o menu
void menu_mostrar(void){
  printf("1 - Gerar lista aleatoriamente\n");
  printf("2 - Criar lista manualmente\n");
  printf("0 - Sair...\n\n");
}

//Gera uma lista aleatória
void lista_gerar(void){
  for (int i = 0; i < tamanho; i++){
    lista[i] = rand()%50;
  }
}

//Permite que o usuário entre com os valores da lista
void lista_ler(void){
  for (int i = 0; i < tamanho; i++){
    system("cls");
    lista_mostrar();
    printf("\nDigite o valor para a posicao %d: ", i);
    scanf("%d", &lista[i]);
  }
}
```

```
//Preparar a lista para ordenação
void lista_limpar(void){
  for (int i = 0; i < tamanho; i++)      {
    ordenado[i] = lista[i];
  }
}

//Mostra o conteúdo da lista ordenada
void lista_mostrar_ordenado(void){
  printf("[  ");
  for (int i = 0; i < tamanho; i++ ){
    printf("%d  ",ordenado[i]);
  }
  printf("] Tempo = %d iteracoes\n\n", qtd);
}
```

Eu dividi o programa em vários blocos e vou explicá-los um a um. Além de entender o seu funcionamento, você entenderá onde deverá adicionar as funções de ordenação para poder usar essa ferramenta de testes durante os seus estudos.

O Programa 3.1.a mostra a declaração das bibliotecas, constantes e variáveis. As bibliotecas **stdio** e **stdlib** são velhas conhecidas de qualquer estudante ou programador habituado na linguagem C. A diferença está na biblioteca **time**, ela foi incluída para que possamos fazer uso da máquina de geração de números aleatórios que veremos adiante.

Existe uma única constante chamada **tamanho**. Ela servirá de parâmetro em praticamente todas as funções de ordenação. Ela define o tamanho máximo do vetor em que os nossos dados estarão armazenados. Para ficar mais fácil, fixamos o tamanho em 10, mas você pode e deve alterar esse número para um valor maior durante as suas investigações nesta matéria.

Criamos dois vetores, a variável **lista** guardará os dados na ordem original, e a variável **ordenado** conterá os valores após a aplicação das técnicas de ordenação. Com o vetor original intacto, podemos aplicar diferentes técnicas uma após a outra sem precisar "bagunçar" os dados no vetor antes de testar um novo algoritmo.

A variável **opt** é usada para fazer o controle do menu e a variável **qtd** é usada para calcular "o esforço computacional" despendido pela técnica de ordenação.

Programa 3.1.a – Bibliotecas, Constantes e Variáveis

```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//Constantes
#define tamanho 10

//Variáveis
int lista[tamanho];
int ordenado[tamanho];
int opt=-1;
int qtd;
```

O próximo bloco é representado pelo Programa 3.1.b, com a região de prototipação. Essa área é usada para colocar a lista de funções existentes no programa. Isso permite que as funções possam ser chamadas de qualquer parte do arquivo. Quando você for adicionar um algoritmo de ordenação desse programa, adicione nesse bloco o nome da função que irá executar a técnica escolhida.

Programa 3.1.b – Prototipação

```
//Prototipação
void menu_mostrar(void);
void lista_mostrar(void);
void lista_gerar(void);
void lista_ler(void);
void lista_limpar(void);
void lista_mostrar_ordenado(void);
```

A função principal está representada no Programa 3.1.c. O comando **srand** está incluído na biblioteca **time** e serve para inicializar a **máquina geradora de números aleatórios**, que usaremos mais adiante.

O laço principal limpa a tela, desenha o vetor original, o vetor ordenado e mostra ao usuário quais são as opções disponíveis no programa. Por meio da entrada na variável **opt,** a estrutura **case** chama a função desejada.

Quando estiver incluindo uma nova função programa, adicione um novo caso na estrutura **case** para que o usuário possa estar aplicando a técnica de ordenação desejada.

Programa 3.1.c – Função principal

```
//Função Principal
int main(void){
  srand(time(NULL));
  do {
    system("cls");
    lista_mostrar();
    lista_mostrar_ordenado();
    menu_mostrar();
    scanf("%d",&opt);
    switch (opt){
      case 1:
        lista_gerar();
      break;
      case 2:
        lista_ler();
      break;
    }
  }while(opt!=0);
  system("pause");
  return(0);
}
```

O Programa 3.1.d mostra três funções criadas para desenhar a tela do programa. Você precisará alterar apenas a **menu_mostrar**. Originalmente ela contém três opções: **1)** gerar lista aleatoriamente; **2)** criar uma lista manualmente; e **0)** sair. Para cada algoritmo de ordenação incluído no programa, uma nova opção deverá ser criada na função **menu_mostrar**.

Programa 3.1.d – Desenho do conteúdo da lista e do menu de opções

```
//Mostra o conteúdo da lista
void lista_mostrar(void){
  printf("[  ");
  for (int i = 0; i < tamanho; i++ ){
    printf("%d  ",lista[i]);
  }
  printf("]\n\n");
}
```

```
//Mostra o menu
void menu_mostrar(void){
  printf("1 - Gerar lista aleatoriamente\n");
  printf("2 - Criar lista manualmente\n");
  printf("0 - Sair...\n\n");
}

//Mostra o conteúdo da lista ordenada
void lista_mostrar_ordenado(void){
  printf("[  ");
  for (int i = 0; i < tamanho; i++ ){
    printf("%d  ",ordenado[i]);
  }
  printf("] Tempo = %d iteracoes\n\n", qtd);
}
```

No caso de o usuário não querer digitar o conteúdo de um vetor para que possa ser ordenado, criamos a função **lista_gerar** apresentada no Programa 3.1.e. Ela percorre todo o vetor **lista** e para cada posição sorteia um número entre **0** e **50** por meio da função **rand() % 50**.

Programa 3.1.e – Gera uma lista aleatoriamente

```
//Gera uma lista aleatória
void lista_gerar(void){
  for (int i = 0; i < tamanho; i++){
    lista[i] = rand()%50;
  }
}
```

Por fim, mais duas funções. A primeira, no Programa 3.1.f, solicita que o usuário preencha uma lista com valores escolhidos por ele. Em alguns casos poderá ser interessante testar os diversos algoritmos em vetores ordenados ou parcialmente ordenados, verificando o desempenho nessas situações peculiares. A função **lista_limpar** prepara o vetor auxiliar para a aplicação da técnica de ordenação escolhida pelo usuário. Essa função deve ser incluída no laço do menu principal junto com a função de ordenação em cada uma das novas entradas na estrutura **case**.

Programa 3.1.f – Leitura dos dados e preparação para a ordenação

```
//Permite que o usuário entre com os valores da lista
void lista_ler(void){
  for (int i = 0; i < tamanho; i++){
    system("cls");
    lista_mostrar();
    printf("\nDigite o valor para a posicao %d: ", i);
    scanf("%d", &lista[i]);
  }
}

//Preparar a lista para ordenação
void lista_limpar(void){
  for (int i = 0; i < tamanho; i++)     {
    ordenado[i] = lista[i];
  }
}
```

ORDENAÇÃO POR BUBBLESORT (MÉTODO DA BOLHA)

A técnica de ordenação **Bubblesort** também é conhecida por **ordenação por flutuação** ou por **método da bolha**. Ela é de simples implementação e de alto custo computacional. Começando na primeira posição do vetor, compara-se o valor dela com todos os demais elementos, trocando caso o valor da posição atual seja maior do que o valor verificado. Os valores mais altos vão **flutuando** para o final do vetor, criando a ordenação da estrutura. Esse processo se repete para cada uma das posições da tabela.

Programa 3.2 – Implementação do método Bubblesort

```
//Aplica o método do bubbleSort
int bubbleSort(int vec[]){
  int qtd, i, j, tmp;
  qtd = 0;
  for (i = 0; i < tamanho -1; i++){
    for (j = i+1; j < tamanho; j++){
      if (vec[i] > vec[j]){
        troca(&vec[i], &vec[j]);
      }
      qtd++;
    }
  }
  return(qtd);
}

//Função genérica de troca de valores
void troca(int* a, int* b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

O Programa 3.2 traz também uma função chamada **troca**. Ela recebe como parâmetro dois ponteiros e tem como objetivo trocar os seus valores de lugar. Essa função também será utilizada em outros algoritmos de ordenação.

Vamos simular mentalmente esse algoritmo com base no vetor **vec[]** desordenado de testes apresentado na Figura 1.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Figura 1 - Vetor vec[] desordenado para teste

Vamos fixar na primeira posição **vec[0]=3** e compará-lo com o próximo **vec[1]=1**. Como **3 > 1**, os valores são trocados no vetor.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

Continuamos fixo no **vec[0]**, que agora possui valor **1,** e comparamos com a próxima posição **vec[2]=8**. Como **1 < 8**, nada acontece.

O programa continua comparando **vec[0]** com todas as demais posições do vetor de dados até encontrar **vec[9]=0,** onde haverá nova troca.

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
0	3	8	7	20	21	31	40	30	1

Depois fixamos em **vec[1]** e novamente comparamos com todos os demais. Como **vec[1]=3**, acontecerá troca apenas em **vec[9]=1.**

0	1	2	3	4	5	6	7	8	9
0	3	8	7	20	21	31	40	30	1

0	1	2	3	4	5	6	7	8	9
0	1	8	7	20	21	31	40	30	3

Depois fixamos **vec[2]** e, fazemos comparação com o restante do vetor e assim por diante, até que encontremos a última posição e o arquivos estejam ordenados. Ao mesmo tempo em que os valores maiores são empurrados para a direita, os menores são puxados para a esquerda. No final de cada iteração do laço externo do algoritmo, a parte inicial da tabela fica mais e mais ordenada.

REFLITA

O algoritmo Bubblesort irá comparar todos os valores, de dois em dois, do primeiro ao último valor do vetor, mesmo se já estiver ordenado. Ele só é indicado a vetores pequenos devido à grande quantidade de repetições necessárias para a sua execução.

ORDENAÇÃO POR SELECTIONSORT

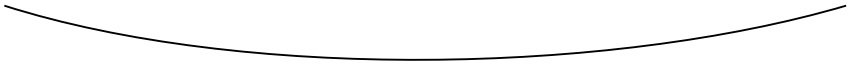
A técnica também é de simples implementação e de alto consumo computacional. A partir da primeira posição, procura-se o menor valor em todo o vetor. Chegando no final da estrutura, trocamos o menor valor encontrado com a primeira posição. Em seguida, ele parte para a segunda posição e passa a procurar o segundo menor valor do vetor até o final da tabela, fazendo a troca de posição dos valores. O algoritmo repete até que a lógica seja aplicada a cada uma das posições da tabela.

Programa 3.3 – Implementação do método Selectionsort

```
//Aplica o modo selectionSort
int selectionSort(int vec[], int tam){
  int i, j, min, qtd=0;
  for (i = 0; i < (tam-1); i++)
  {
     min = i;
     for (j = (i+1); j < tam; j++) {
       if(vec[j] < vec[min]) {
         min = j;
       }
       qtd++;
     }
     if (i != min) {
       troca(&vec[i], &vec[min]);
     }
  }
  return(qtd);
}
```

Vamos simular a aplicação dessa técnica com base no vetor **vec**[] da Figura 1. A partir da primeira posição, o algoritmo vai percorrer o vetor até o seu final armazenando numa variável temporária o menor valor encontrado, que no nosso caso é **vec**[**9**]=**0**. Mais uma vez usaremos a função **troca** vista no Programa 3.2. Ela irá trocar os valores da primeira posição **vec**[**0**]=**3** com **vec**[**9**]=**0**.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0



0	1	2	3	4	5	6	7	8	9
0	1	8	7	20	21	31	40	30	3

Agora, a partir da segunda posição, o algoritmo percorrerá todo o vetor buscando o segundo menor valor, que é ele mesmo **vec**[**1**]=**1**. Nada acontece. O próximo passo é procurar a partir da terceira posição o terceiro menor valor, que é **vec**[**9**]=**3**. O valor é trocado com o de **vec**[**2**]=**8**.

0	1	2	3	4	5	6	7	8	9
0	1	8	7	20	21	31	40	30	3

0	1	2	3	4	5	6	7	8	9
0	1	3	7	20	21	31	40	30	8

O algoritmo continua até que o processo seja repetido para cada uma das posições da tabela.

REFLITA

O Selectionsort irá comparar o elemento da posição atual com todos os valores da tabela mesmo que o valor atual seja o menor valor do vetor. O tempo computacional é o mesmo para um vetor ordenado, não ordenado e parcialmente ordenado.

Enquanto o Bubblesort faz uma troca sempre que a posição atual fixa é maior que a posição visitada, o Selectionsort faz a troca apenas quando tem certeza que o menor valor foi encontrado para a atual posição.

ORDENAÇÃO POR INSERTIONSORT

A ordenação **Insertionsort** também é conhecida como ordenação por inserção. É de implementação simples e traz bons resultados. A técnica consiste em remover o primeiro elemento da lista, e procurar sua posição ideal no vetor e reinseri-lo na tabela. O processo é repetido para todos os elementos.

O Programa 3.4 apresenta uma implementação em linguagem C do método de ordenação por inserção. Ele possui dois laços de repetição, o primeiro é executado inteiro e o segundo um número aleatório, dependendo da distância que o elemento está da sua posição ideal. Esse algoritmo também utiliza a função **troca** estudada anteriormente.

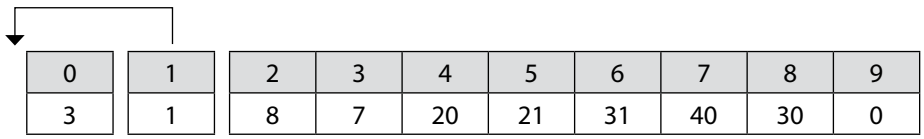
A princípio o Insertionsort é muito parecido com o Bubblesort e o Selectionsort, já que todos os três trazem dois laços de repetição aninhados, porém os dois últimos percorrem sempre os dois laços por inteiro. Esse é o motivo do Insertionsort ser mais rápido do que os outros dois.

Isso pode ser facilmente comprovado aplicando os três métodos em um vetor já ordenado e observando a quantidade de vezes que cada um efetua trocas de posições.

Programa 3.4 – Implementação do método Insertionsort

```
//Aplicando o insertionSort
int insertionSort(int vec[], int tam)
{
  int i, j, qtd=0;
  for(i = 1; i < tam; i++){
    j = i;
    while((vec[j] < vec[j - 1]) && (j!=0)){
      troca(&vec[j], &vec[j-1]);
      j--;
      qtd++;
    }
  }
  return(qtd);
}
```

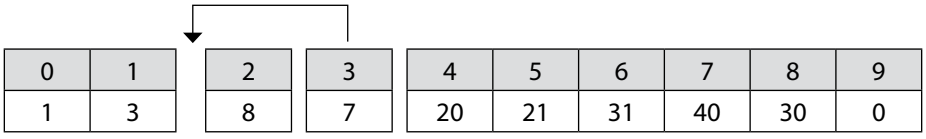
Novamente com base no vetor **vec** da Figura 1, vamos simular a ordenação por inserção. O primeiro elemento **vec[0]=3**, vamos removê-lo do vetor e inseri-lo novamente do lado do primeiro número que encontrarmos que for menor do que ele.



0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

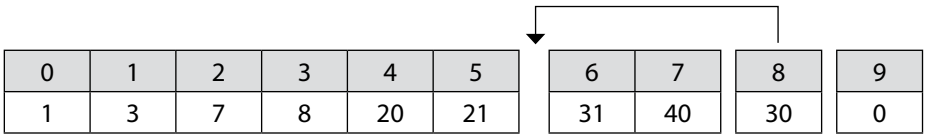
Os valores de **vec[0]=1** e **vec[1]=3** se encontram corretamente posicionados no vetor. O próximo a ser analisado será **vec[2]=8**. Ele será removido e então reinserido assim que for encontrado um valor menor do que ele, que no caso será **vec[3]=7**.



0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
1	3	7	8	20	21	31	40	30	0

O valor de **vec[4]** já se encontra ao lado de um número maior que ele, assim como **vec[5]** e **vec[6]**, não havendo nenhuma alteração. O próximo valor que será removido e reinserido no vetor é **vec[8]=30**.



0	1	2	3	4	5	6	7	8	9
1	3	7	8	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
1	3	7	8	20	21	30	31	40	0

Agora falta fazer o mesmo com o último elemento **vec[9]** e o vetor estará ordenado.

ORDENAÇÃO POR SHELLSORT

O algoritmo **Shellshort** de ordenação não tem nada a ver com uma concha (*shell,* em inglês). Ele tem esse nome em homenagem ao seu criador Donald Shell, publicado pela Universidade de Cincinnat em 1959.

Segundo Wirth (1989, p. 61 a 63), ele é o mais eficiente dentro dos algoritmos classificados como de complexidade quadrática. Ele é uma técnica refinada do método de ordenação por inserção.

REFLITA

Um algoritmo é considerado de complexidade quadrática se houver nele dois laços aninhados.

Ao invés de tratar o arquivo como um todo, ele divide a tabela em segmentos menores e em cada um deles é aplicado o **Insertionsort**. Ele faz isso diversas vezes, dividindo grupos maiores em menores até que todo o vetor esteja ordenado.

O Programa 3.5 apresenta a implementação do algoritmo de Shellsort. Ele possui uma variável chamada **gap**. O **gap** determina a distância entre os elementos que serão removidos do vetor original. Ao subvetor aplica-se o algoritmo de **Insertionsort**, e o subvetor é novamente inserido no vetor original. O processo se repete até atingir todos os elementos.

O valor de **gap** sofre um decremento e uma nova quantidade de grupos é criada no vetor parcialmente ordenado. Aplica-se o processo de ordenação por Insertionisort em cada um dos subvetores. O processo se repete até que **gap** seja igual a **1**, então uma nova sequência de insertionsort é realizada e o vetor termina por estar ordenado.

Programa 3.5 – Implementação do método Shellsort

```
//Aplica o shellSort
int shellSort(int vec[], int tam) {
  int i , j , valor, qtd=0;
  int gap = 1;
  do {
    gap = 3*gap+1;
  } while(gap < tam);
  do {
    gap /= 3;
    for(i = gap; i < tam; i++) {
      valor = vec[i];
      j = i - gap;
      while (j >= 0 && valor < vec[j]) {
        vec[j + gap] = vec[j];
        j -= gap;
        qtd++;
      }
      vec[j + gap] = valor;
    }
  } while ( gap > 1);
  return (qtd);
}
```

Imagine um vetor **vec** de **12** posições e a variável **gap** com valor **3**. O primeiro subgrupo terá os valores:

vec[0], vec[3], vec[6], vec[9], vec[12].

Nesse subvetor de quatro elementos é aplicado o Insertionsort e seus dados serão inseridos de volta no vetor original. Depois é a vez dos valores:

vec[1], vec[4], vec[7], vec[10].

Aplica-se o Insertionsort e devolve os valores ao vetor. O processo continua agora para os valores:

vec[2], vec[5], vec[8], vec[11].

O novo subvetor é ordenado por Insertionsort, seu resultado é inserido de volta ao vetor original. Nesse momento, finaliza a primeira passagem e **gap** sofre um decremento. Vamos supor para o nosso caso que o decremento é de **1**, então **gap** agora será **2**.

Com o novo valor de gap, um novo subvetor é criado com os seguintes valores:

vec[0], vec[2], vec[4], vec[6], vec[8], vec[10], vec[12]

Esse subvetor é ordenado por Insertionsort. O processo continua até que todos os valores tenham sido escolhidos e o gap sofre um novo incremento. Quando o valor de **gap** for 1, será aplicado o algoritmo de Insertionsort no vetor original parcialmente ordenado e o processo é finalizado.

CONSIDERAÇÕES FINAIS

No início da unidade, o (a) aluno (a) foi apresentado (a) a um ambiente de testes pronto para uso. Ele traz o programa completo com desenho de tela, geração automática ou manual de vetores. Com essa ferramenta em mãos, é possível partir diretamente para o estudo e execução das técnicas de pesquisa aqui apresentados.

O primeiro algoritmo apresentado foi o Bubblesort, que é de fácil compreensão e implementação. Devido ao seu alto custo computacional, seu uso não é indicado para a ordenação de grandes quantidades de informação.

Em seguida, vimos o Selectionsort, que lembra um pouco o algoritmo anterior em simplicidade, complexidade e entendimento. Enquanto o Bubblesort vai empurrando os valores grandes para o final do vetor, o Selectionsort vai puxando os menores para o início da tabela.

Por fim, verificamos como o Insertionsort é um algoritmo intuitivo que pode utilizar um laço interno mais otimizado, que evita comparações desnecessárias ao se encontrar a posição correta de um elemento ordenado. Como constatamos, o conceito de gap pode ser associado ao Insertionsort para compor um algoritmo ainda mais otimizado, chamado Shellsort.

Na próxima unidade, veremos mais três técnicas de ordenação e fecharemos o conteúdo fazendo algumas comparações e considerações entre os algoritmos apresentados no livro.

ATIVIDADES

1. Qual é a ideia do Bubblesort, suas vantagens e desvantagens?

2. Qual a diferença entre o Insertionsort e o Bubblesort?

3. Quais as semelhanças e diferenças entre o Insertionsort, Selectionsort e Bubblesort?

4. Como o algoritmo proposto por Donald Shell pode ser mais eficiente que o Insertionsort se ele mesmo utiliza-se do Insertionsort durante o processo de ordenação?

MOTORES DE BUSCA

por Paulo Peixoto, Faculdade de Economia, Universidade de Coimbra - adaptado por Pietro Martins de Oliveira

O que são Motores de Busca?

Os motores de busca utilizam software conhecido como ''aranhas" ou "robots" que percorrem "toda" a Internet em busca da informação (documentos ou endereços de páginas web) que se pretende. Os dados são recolhidos para o índice dos motores de busca, que cria uma base de dados com essa informação. A forma como a informação é indexada depende de cada motor de busca, podendo ser feita por palavras, títulos, URL's ou por directorias. Assim, sempre que se introduz uma palavra ou um conjunto de palavras que se pretende pesquisar, as bases de dados são percorridas em busca de documentos ou sites que lhe correspondam. O resultado da busca é dado em *hyperlinks*, podendo clicar-se em cada uma das entradas para aceder à informação.

Na internet encontramos centenas de motores de busca. Alguns deles são acessíveis a partir desta página, que os distingue entre directórios, indexantes e metapesquisadores.

Uma mesma pesquisa efetuada nesses diversos motores de busca terá resultados muito diferentes. Mesmo pesquisas efetuadas num mesmo instrumento (motor de busca) poderão dar resultados muito diferentes de um dia para o outro. Qualquer internauta com alguma experiência sabe que, ao utilizar os instrumentos de pesquisa, está a "lançar uma garrafa ao mar" com uma mensagem lá dentro. Se a mensagem "chega a bom porto" é sempre uma incógnita. Entre os resultados da pesquisa efetuada só alguns são pertinentes, pois não é possível ter um controlo total da estratégia de pesquisa (pois não depende só do internauta). Também não é a performance informática dos motores de busca que está em causa, embora ela tenha alguma importância, mas a performance do ponto de vista documental (diz-se que, hoje em dia, está tudo na internet, mas nem na internet é possível encontrar documentos que não existem).

Uma pesquisa traduz uma boa performance quando permite encontrar os documentos pesquisados e apenas os documentos pesquisados. De acordo com esse critério, a pesquisa na internet revela uma grande taxa de "silêncio" (os documentos pertinentes não são encontrados) ou de ruído (encontramos frequentemente um vasto conjunto de documentos não pertinentes). Por isso, só muito dificilmente podemos efetuar uma pesquisa na internet que permita uma boa performance. O investimento em tempo exigido para efetuar a pesquisa e o exame e análise dos resultados obtidos são frequentemente exigentes e morosos considerando os benefícios que se retiram da pesquisa.

Qual utilizar?

Existem dois tipos de motores de busca: os que indexam, por título e URL, toda a informação onde encontram a palavra ou o conjunto de palavras que pediu, conhecidos como *webcrawlers* (como é o caso do Google) e aqueles que funcionam com base em diretórios (o mais conhecido é o Yahoo). Enquanto que os primeiros devem ser utilizados para pesquisas mais específicas, os motores de busca baseados em diretórios deverão ser utilizados para encontrar informação mais genérica, dado que os resultados da pesquisa são um conjunto de sites onde se poderá encontrar o que se pretende. Exemplificando, se o objectivo é encontrar o site do Sport Lisboa e Benfica deve utilizar-se o Yahoo, mas se pretende encontrar a biografia do Eusébio será mais fácil pesquisar no Google.

Um dos pontos negativos de sites como o Google é que poderão devolver informação irrelevante para o que se procura, dado que indexam páginas individuais que fazem referência à pesquisa efectuada, mas que não têm nada a ver com o que se pretende. No entanto, esse tipo de motor de busca é melhor em termos de atualização da informação.

Atualmente, com a evolução da própria Internet, a maior parte dos motores de busca já apresenta um misto de directorias com indexantes, aumentando as possibilidades de encontrar o que se deseja.

Dicas de pesquisa

Para facilitar a pesquisa pode-se utilizar operadores booleanos, tais como o AND, OR e NOT, ou usar parêntesis, efetuando pesquisas booleanas mais complexas (pesquisa avançada). Um exemplo poderia ser uma pesquisa por (Estádio da Luz AND Eusébio), que resultava em informação relacionada com o Estádio da Luz e Eusébio. Os operadores booleanos podem ser utilizados em simultâneo (Estádio da Luz AND NOT Eusébio) - resultava em informação relacionada com Estádio da Luz excluindo o Ex-jogador Eusébio - ou substituídos por sinais "-" ou "+" (+estádio da luz -eusébio). Para confirmar estas situações, deve ler-se com atenção as ajudas dadas pelos motores de busca, dado que cada motor de busca tem a sua linguagem específica para pesquisas avançadas.

Outra recomendação é a utilização de mais do que um motor de busca. É impossível que apenas um tenha toda a informação disponível na Internet. Assim, pesquisar em vários aumenta as possibilidades de encontrar o que se procura. Nesse caso dever-se-á utilizar os metapesquisadores, motores de busca que procuram diretórios e índices a partir de outros motores de busca. Os mais conhecidos são o MetaCrawler e o Ixquick.

Pesquisar na Internet não é tão fácil como parece, mas conferindo alguma atenção às ajudas dadas pelos motores de busca, ser capaz de aprender com as pesquisas efetuadas e, sobretudo, não desistir, ajuda a encontrar a informação exata que se pretende.

Fonte: Peixoto (2008).

MATERIAL COMPLEMENTAR

NA WEB

Dança simulando uma ordenação por Bubblesort
<http://www.youtube.com/watch?v=lyZQPjUT5B4>.

Dança simulando uma ordenação por Shellsort
<http://www.youtube.com/watch?v=CmPA7zE8mx0>.

Dança simulando uma ordenação por Insertionsort
<http://www.youtube.com/watch?v=ROalU379l3U&NR=1>.

Dança simulando uma ordenação por Selectionsort
<http://www.youtube.com/watch?v=Ns4TPTC8whw>.

Professor Me. Pietro Martins de Oliveira
Professor Me. Rogério de Leon Pereira

ALGORITMOS DE ORDENAÇÃO AVANÇADOS

UNIDADE IV

Objetivos de Aprendizagem

- Estudar técnicas mais eficientes de ordenação de tabelas.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Ordenação por Mergesort
- Ordenação por Quicksort
- Ordenação por Heapsort

Unicesumar

INTRODUÇÃO

Na unidade anterior, estudamos quatro algoritmos de ordenação. Os dois primeiros, Bubblesort e Selectionsort, eram de implementação mais simplificada. O terceiro, Insertionsort, ligeiramente mais complexo, utiliza um laço de repetição interno para tentar otimizar a ordenação. Já o Shellsort, quarto método, se apropria do Insertionsort para realizar a ordenação de um vetor considerando o conceito de gaps.

Agora veremos mais três técnicas de ordenação, sua complexidade será crescente, assim com o seu desempenho. No final, faremos um fechamento debatendo sobre todos os métodos estudados até então.

ORDENAÇÃO POR MERGESORT

A técnica de ordenação Mergesort utiliza um conceito conhecido por **dividir para conquistar**. Esse conceito sugere que um problema complexo possa ser dividido em dois problemas menores, e cada um desses sejam divididos novamente em partes menores ainda, até que se encontre uma parte pequena e simples suficiente para que seja resolvido.

O algoritmo Mergesort faz isso de forma recursiva. Assim que o vetor é dividido, cada uma das metades é passada como parâmetro a uma nova chamada da função Mergesort. Essa recursividade desce até o ponto em que o vetor tem apenas um único valor. Nesse momento, inicia-se o retorno da recursividade, e os vetores unitários são comparados e unidos já ordenados.

Essa abordagem é bem mais complexa do que as duas anteriores, porém o seu esforço computacional é reduzido. Para cada divisão, porém, faz-se necessária a criação de um novo vetor na memória e, no caso de ordenação de arquivos muito grandes, a utilização de memória pode ser excessiva.

O Programa 4.1 traz uma implementação em linguagem C da técnica de ordenação Mergesort. A primeira função é bem simples e mostra a recursividade da técnica. Ela recebe como parâmetro o vetor **vec** a ser ordenado, o tamanho **tam** do vetor e uma variável **qtd** inteira usada pra medir o esforço computacional do algoritmo.

Se o tamanho do vetor for maior do que um, o programa procura o meio do vetor e aplica a recursão duas vezes, uma para o início até a metade do vetor atual e outra da metade até o final do vetor.

Depois que o vetor for transformado em partes unitárias, a recursividade volta chamando a função junta. Ela irá verificar o valor das partes antes de realizar a junção de forma ordenada.

Programa 4.1 – Implementação do método Mergesort

```
//Aplica o modo mergeSort
int mergeSort(int vec[], int tam, int qtd) {
  int meio;

  if (tam > 1) {
    meio = tam / 2;
    qtd = mergeSort(vec, meio, qtd);
    qtd = mergeSort(vec + meio, tam - meio, qtd);
    junta(vec, tam);
  }
  return (qtd+1);
}

//Junta os pedaços num novo vetor ordenado
void junta(int vec[], int tam) {
  int i, j, k;
  int meio;
  int* tmp;

  tmp = (int*) malloc(tam * sizeof(int));
  if (tmp == NULL) {
    exit(1);
  }
```

```
  meio = tam / 2;
  i = 0;
  j = meio;
  k = 0;

  while (i < meio && j < tam) {
    if (vec[i] < vec[j]) {
      tmp[k] = vec[i];
      ++i;
    }
    else {
      tmp[k] = vec[j];
      ++j;
    }
    ++k;
  }

  if (i == meio) {
    while (j < tam) {
      tmp[k] = vec[j];
      ++j;
      ++k;
    }
  }
  else {
    while (i < meio) {
      tmp[k] = vec[i];
      ++i;
      ++k;
    }
  }

  for (i = 0; i < tam; ++i) {
    vec[i] = tmp[i];
  }
  free(tmp);
}
```

Já com a ideia central do Mergesort em mente, vamos fazer uma breve simulação em cima do vetor apresentado na Figura 1 da Unidade 3. A primeira coisa que o algoritmo faz é dividir o vetor em dois a aplicar recursividade em cada uma das metades.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Cada uma das chamadas ao Mergesort irá dividir novamente o vetor, recursivamente.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

O processo se repete até que o cada vetor contenha apenas um valor.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Nesse momento não há mais chamadas recursivas e começa o retorno para a chamada original aplicando a função **junta** nos pares de vetores, já ordenados.

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	0	3

E o procedimento se repete até que tenhamos apenas um vetor e o mesmo se encontrará totalmente ordenado.

0	1	2	3	4	5	6	7	8	9
0	1	3	7	8	20	21	30	31	40

Figura 1 - Vetor vec[] ordenado pelo Mergesort

REFLITA

O Mergesort foi criado em 1945 pelo matemático húngaro chamado John Von Newmann. Apesar de apresentar bom desempenho em vetores não muito grandes, sua implementação e ideia são complexos se comparado com o Bubblesort e Selectionsort.

ORDENAÇÃO POR QUICKSORT

A segunda técnica de ordenação que veremos nesta unidade é o **Quicksort**. Segundo Cormen (2012), esse método também é conhecido por **classificação por troca de partição**. Criado em 1960 pelo cientista da computação britânico Sir Charles Antony Richard Hoare, ele é considerado o algoritmo de ordenação mais utilizado no mundo. Sua publicação ocorreu em 1962 após uma série de refinamentos.

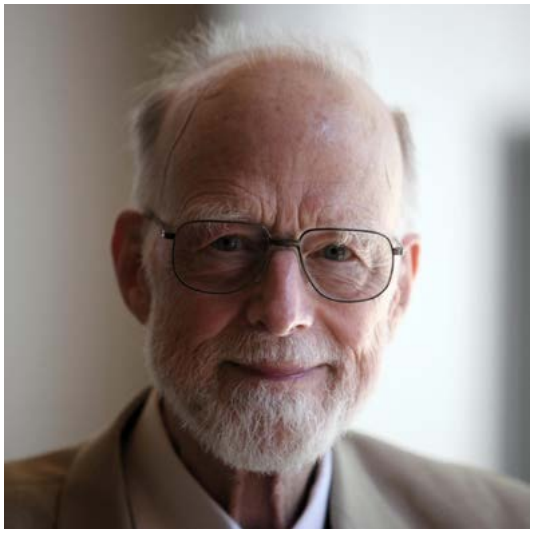


Figura 2 - Sir Charles Antony Richard Hoare em uma conferência na Escola Politécnica Federal de Lausanne, em 20/06/2011
Fonte: <http://upload.wikimedia.org/wikipedia/commons/2/2c/Sir_Tony_Hoare_IMG_5125.jpg>.

Essa técnica também utiliza a estratégia de dividir para conquistar. O primeiro passo é escolher um elemento qualquer que será denominado de pivô. A partir desse elemento, a lista será dividida em três sublistas, uma para o pivô, uma para os valores menores e outra para os valores maiores do que o próprio pivô.

Isso garante que as chaves menores precedam as chaves maiores e que o pivô esteja na sua correta posição dentro do vetor. Essa técnica é muito parecida com a árvore de busca binária. As duas sublistas (partições) ainda não ordenadas são chamadas de forma recursiva até que cada uma das inúmeras sublistas criadas tenha apenas um elemento e o vetor se encontre ordenado.

O Programa 4.2 apresenta duas funções. A primeira é o **Quicksort** propriamente dito a sua chamada recursiva. A cada iteração ele invoca a função **Particiona,** que vai escolher o pivô e criar duas novas listas a serem ordenadas.

Programa 4.2 – Implementação do método Quicksort

```
//Aplica o modo do quickSort
int quickSort(int vec[], int left, int right, int qtd) {
  int r;
  if (right > left) {
    r = particiona(vec, left, right);
    qtd = quickSort(vec, left, r - 1, qtd);
    qtd = quickSort(vec, r + 1, right, qtd);
  }
  return (qtd +1);
}

//Divide o vetor em pedaços menores
int particiona(int vec[], int left, int right) {
  int i, j;
  i = left;
  for (j = left + 1; j <= right; ++j) {
    if (vec[j] < vec[left]) {
      ++i;
      troca(&vec[i], &vec[j]);
    }
  }
  troca(&vec[left], &vec[i]);
  return i;
}
```

Esse algoritmo também se assemelha ao Mergesort. A principal diferença é que o Quicksort trabalha com um pivô numa posição aleatória e, durante o processo de partição, o pivô já estará na sua posição final do vetor. O Mergesort divide a estrutura sempre pela metade e inicia o processo de ordenação apenas no final do processo durante o retorno da recursividade.

Vamos fazer uma simulação do Quicksort no vetor **vec** desordenado apresentado na Figura 3.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Figura 3 - Vetor vec[] desordenado para teste

Qualquer elemento pode ser escolhido como pivô. Pense num número de 0 a 9. Pronto, pensou? Isso mesmo, você acertou, escolhi começar por **vec[0]=3**. Vamos separar a lista agora em três partes, uma com o **pivô**, uma com os elementos menores que **3** e outra com elementos maiores que **3**.

0	1	2	3	4	5	6	7	8	9
1	0	3	8	7	20	21	31	40	30

O valor escolhido para o pivô (**3**) já se encontra na sua devida posição na lista, e à sua esquerda está a sublista com valores menores que **3** e à direita outra sublista com valores maiores que **3**. Aplicaremos a recursividade em cada uma dessas sublistas.

Para ficar mais claro o entendimento, vamos tratar as duas chamadas recursivas separadamente, primeiro a da sublista com valores menores que o pivô. Escolheremos nela um elemento qualquer para ser o novo pivô na recursão. Vamos pegar **vec[0]=1**. Os valores da sublista serão divididos novamente, ficando os valores menores à esquerda e os maiores à direita.

0	1	2	3	4	5	6	7	8	9
0	1	3	8	7	20	21	31	40	30

O valor do pivô (**1**) já se encontra na sua devida posição na lista. Como sobrou apenas um elemento na sublista (**0**), o mesmo já se encontra ordenado. Agora vamos tratar da recursão do outro lado do primeiro pivô.

Faremos diferente e vamos escolher **vec[9]=30** como novo pivô. Dividiremos a lista em duas sublistas e aplicaremos novamente a recursão. Uma das listas terá apenas valores menores do que **30** e a outra apenas valores maiores.

0	1	2	3	4	5	6	7	8	9
0	1	3	8	7	20	21	30	31	40

O algoritmo ainda não sabe, mas a parte superior da lista já se encontra ordenada. Mesmo assim aquela parte também sofrerá recursão e em mais uma interação estará pronta. A sublista com os valores menores também está quase ordenada, e a quantidade de passos necessários para a finalização depende da escolha do pivô.

Se for escolhido **7** ou **8,** o vetor já ficará ordenado. Se for escolhido **20** ou **21,** será necessário ainda mais uma iteração para encontrar o vetor original devidamente ordenado.

0	1	2	3	4	5	6	7	8	9
0	1	3	8	7	20	21	30	31	40

Figura 4 - Vetor vec[] ordenado pelo Quicksort

ORDENAÇÃO POR HEAPSORT

Para compreender como o Heapsort realiza a ordenação de um arranjo, devemos remeter a outra estrutura de dados: as filas de prioridade. Uma fila de prioridades agrupa elementos de forma que cada um dos elementos pode ter maior ou menor importância para a aplicação. Em suma, nesse tipo de fila é possível inserir elementos a qualquer instante e em qualquer posição do arranjo, de acordo com sua prioridade. Já a remoção é sempre feita no elemento de maior prioridade.

A implementação de uma fila de prioridades eficiente advém da estrutura de dados *heap*. Uma *heap* permite a inserção e remoção de elementos em filas de prioridade em tempo logarítmico, o que é algo bastante eficiente. Tamanha eficiência é alcançada a partir da transformação de um vetor linear em uma estrutura similar a uma árvore binária. Todavia devemos lembrar que o algoritmo Heapsort não implementa uma fila de prioridades, ou seja, são coisas distintas.

Podemos definir a estrutura de dados *heap* como uma árvore binária com algumas propriedades adicionais. Considere uma árvore binária com N níveis, que vão de 0 até N-1:

- A *heap* deve ser uma árvore binária eficiente, por isso é preciso que ela seja uma árvore completa até o nível N-2. Isto é, a *heap* é, obrigatoriamente, uma árvore binária completa até o penúltimo nível.
- Por convenção a *heap* deve fazer com que os nós do nível N-1 (último nível) estejam tão à esquerda quanto possível.
- A chave de cada nó deve ser comparada ao seu nó pai. Ou seja, o conteúdo de nós x e y, cujas subárvores são enraizadas em z, devem respeitar a seguinte regra:
- No caso de uma *max-heap*, o nó raiz deve ser maior ou igual aos nós filhos x e y.
- Já em uma *min-heap*, o nó raiz deve ser menor ou igual aos nós filhos x e y.

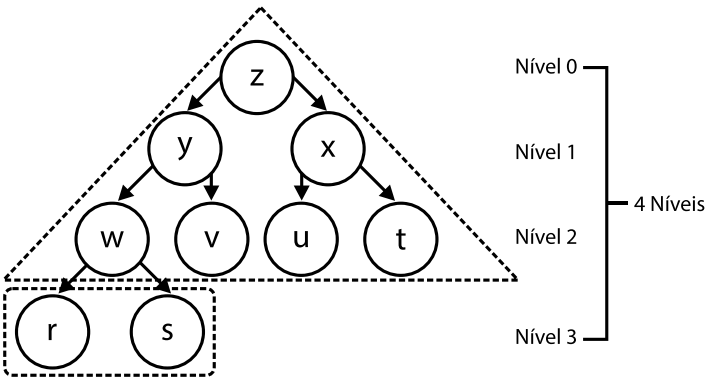


Figura 5 – Representação visual de uma *heap*
Fonte: o autor.

Tais propriedades nos auxiliam a armazenar a *heap* em um vetor, ao invés de ter de trabalhar com alocação dinâmica de memória. Lembra-se de como usar fórmulas matemáticas de posicionamento para armazenar uma árvore em um vetor, que vimos anteriormente? Apenas para relembrar: se um nó pai está na posição **p** do vetor, então seu filho esquerdo estará na posição **2*p+1** e seu filho direito na posição **2*p+2**. Dessa forma, observe como a *heap* representada visualmente na Figura 6 pode ser armazenada em um vetor **v**.

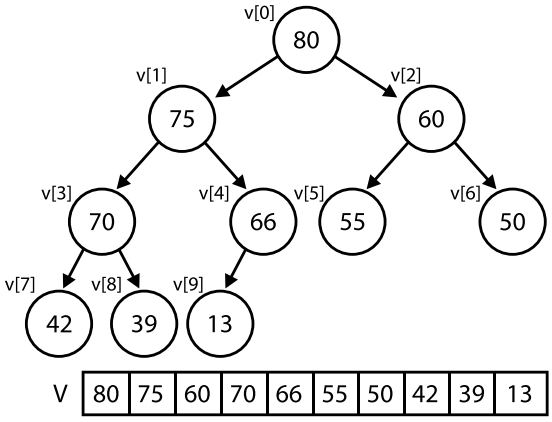


Figura 6 - Exemplo de heap em um vetor
Fonte: o autor.

Repare que as propriedades da *heap* garantem um fato importante: o maior elemento entre todos sempre estará armazenado na raiz, isto é, na posição inicial do vetor **(v[0])**. Dessa forma, podemos pensar em um algoritmo para se aproveitar dessa característica para realizar a ordenação em um vetor. Daremos a esse algoritmo o nome de **Heapsort**.

Primeiramente, precisamos garantir que o vetor esteja formatado como uma *heap*, de acordo com as fórmulas de posicionamento apresentadas anteriormente. Damos o nome de **constroiHeap** ao método que realiza essa façanha (em inglês, ***Build-Max-Heap***). Além de construir uma árvore binária quase completa dentro do vetor, o método constroiHeap é responsável por garantir que cada nó pai seja maior ou igual aos nós filhos.

Em seguida, devemos nos concentrar nas extremidades do vetor de forma a considerar que, conforme o Heapsort vai sendo executado, nas partes iniciais do vetor, temos os dados da *heap*, e nas partes finais do vetor, temos o arranjo

ordenado. Em suma, durante o processo de ordenação, dividimos o vetor logicamente em duas porções: a *heap* e a porção ordenada do vetor.

Uma vez que o vetor desordenado foi transformado em *heap*, podemos dar sequência. Na primeira posição do vetor (raiz da heap), temos o maior elemento de todos. Se nossa intenção é ordenar o vetor em ordem não-decrescente (de modo geral, crescente), podemos simplesmente trocar o maior elemento da raiz pelo elemento que se encontra ao final da *heap*.

Quando trocamos o elemento da raiz da *heap* com o elemento do final do vetor, estamos posicionando o maior elemento em sua posição ordenada final. Nesse instante, devemos desconsiderar tal elemento como um nó da *heap* de forma que, agora, ele passe a pertencer à porção ordenada do vetor. Observe como o nó 80, raiz do vetor **v** da Figura 6, foi trocado com o nó de chave igual a 13, resultando na Figura 7 a seguir.

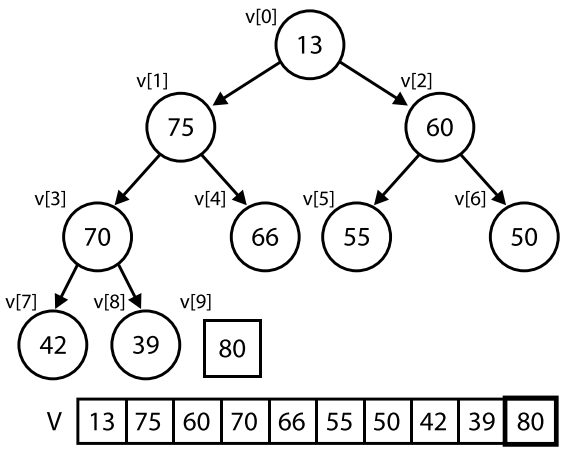


Figura 7 - Trocando o 80 com o 13

Observe que o elemento 80, de fato, é o maior de todos e, após a troca, foi posicionado no último índice de **v**. Assim, o 80 já se encontra ordenado em sua posição final. Todavia, após a troca, nossa árvore perdeu a propriedade de *heap*, pois a raiz 13 não é maior que seus filhos, quebrando as regras. Dessa forma, precisamos consertar a *heap*, fazendo com que a nova raiz "escorregue" até uma posição que restaure nossa árvore binária para ser enquadrada enquanto uma *heap*. Fazemos isso por meio do método que chamamos de **heapifica** (em inglês, ***heapify***).

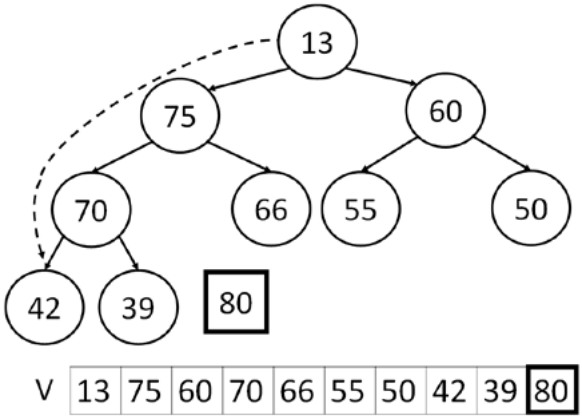
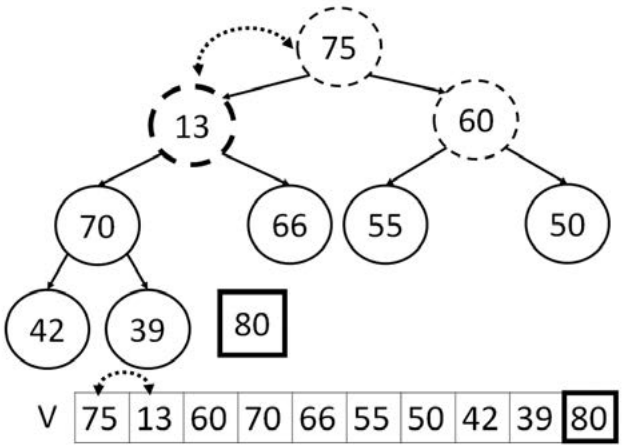


Figura 8 - Comportamento do método heapifica aplicado sobre a raiz
Fonte: o autor.

O método heapifica, quando invocado, vai comparando um nó pai aos respectivos nós filhos. Caso algum dos filhos seja maior que o nó pai, então realizamos a troca entre o maior filho e o pai, de forma que, após essa operação, o nó pai seja, de fato, maior ou igual aos nós filhos para manter a propriedade da *heap*. Todavia essa troca pode fazer com que o novo nó filho quebre as propriedades de *heap*, isto é, o nó filho, recém trocado, pode ter novos filhos que não se categorizam enquanto *heap*. Por isso, é preciso invocar o método heapifica recursivamente, até que todos os nós necessários sejam corrigidos. Observe o passo a passo executado em método heapifica, na Figura 9.



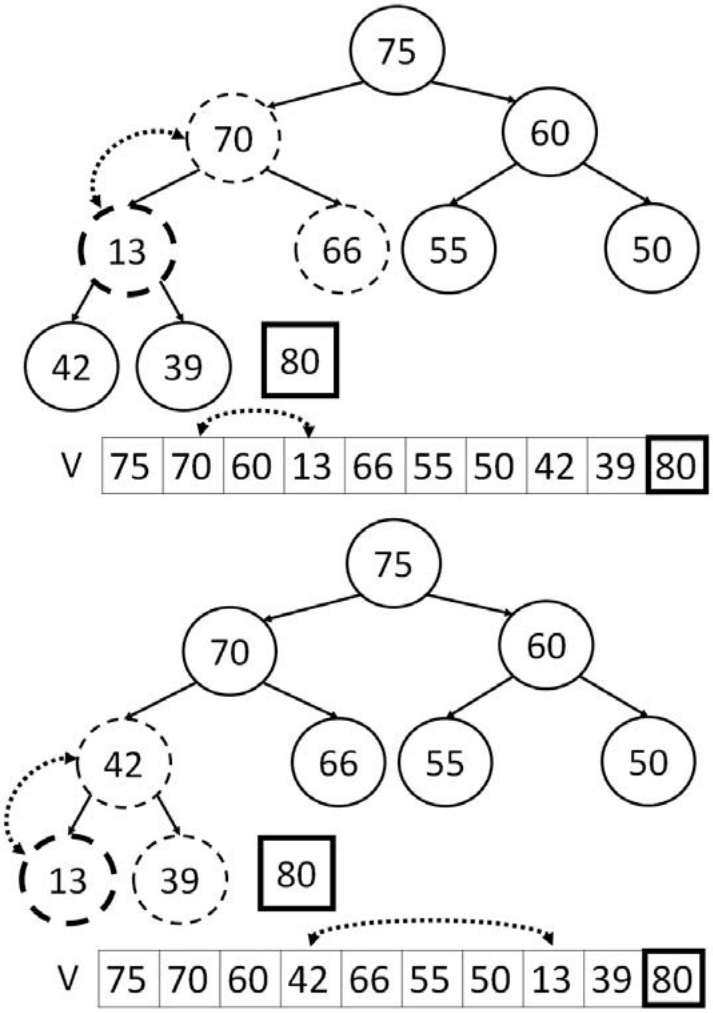


Figura 9 - Passo a passo do método heapifica
Fonte: o autor.

Na Figura 9 podemos notar como o método heapifica troca a raiz de uma subárvore com seu maior filho, à medida em que é executado. Para cada nó trocado, invoca-se o método recursivamente, até que a propriedade de *heap* seja garantida a todos os nós envolvidos no processo. Além disso, podemos observar que, ao final, temos novamente uma *heap* na qual o maior entre todos os elementos se encontra na raiz.

Agora, o nó, cuja chave é igual a 75, encontra-se na raiz. Assim, repetimos o processo realizado anteriormente, no qual a raiz da *heap* era trocada com o "último" elemento do vetor. Por último elemento, entenda a última posição da porção desordenada do vetor, ou seja, a última posição da *heap*. Nesse caso, vamos trocar o 75 com o 13, fazendo com que o 75 se encaixe em sua posição ordenada final, de acordo com o que podemos visualizar na Figura 10.

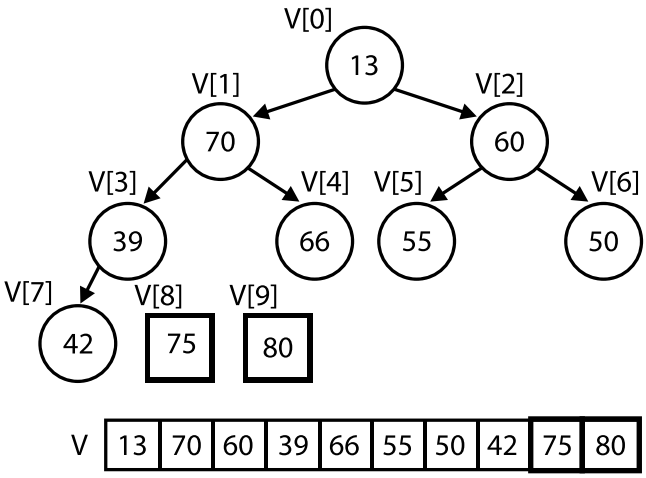


Figura 10 - Trocando o 75 com o 13
Fonte: o autor.

A partir daqui, podemos perceber que, repetindo todo o processo descrito até aqui, o Heapsort posiciona os maiores elementos ao final do arranjo, de maneira ordenada. Na Figura 10, percebemos que o valor 75 é o segundo maior entre todos os elementos de v e, corretamente, está alocado à penúltima posição do vetor.

Programa 4.3 - Heapsort

```
1  //Garante as propriedades de heap a um nó
2  int heapifica(int vec[], int tam, int i){
3         int e, d, maior, qtd;
4         qtd = 1;
5         e = 2*i+1;
6         d = 2*i+2;
7         if(e<tam && vec[e] > vec[i]){
8               maior = e;
9         }
10        else {
11              maior = i;
12        }
13        if(d<tam && vec[d] > vec[maior]){
14              maior = d;
15        }
16        if(maior != i){
17              troca(&vec[i], &vec[maior]);
18              qtd += heapifica(vec, tam, maior);
19        }
20        return qtd;
21 }
22
23 //Transforma o vetor em uma heap
24 int constroiHeap(int vec[], int tam){
25        int i, qtd;
26        qtd = 0;
27        for(i=tam/2;i>=0;i--){
28              qtd += heapifica(vec, tam, i);
29        }
30        return qtd;
31 }
32
33 //Ordena com base na estrutura heap
34 int heapSort(int vec[], int tam){
35        int n, i, qtd;
36        qtd = 0;
37        qtd += constroiHeap(vec, tam);
38        n = tam;
39        for(i=tam-1;i>0;i--){
40              troca(&vec[0], &vec[i]);
41              n--;
42              qtd += heapifica(vec, n, 0);
43        }
44        return qtd;
45 }
```

CONSIDERAÇÕES FINAIS

Vimos até então diversos algoritmos de ordenação. Como já foi dito, os processos de pesquisa e ordenação são duas das operações mais utilizadas na computação.

Agora faremos algumas comparações e análise de desempenho dentre seis das sete técnicas vistas.

Para o nosso exercício de raciocínio, vamos definir o **esforço computacional**. Nessa análise, estaremos considerando como esforço computacional a quantidade de vezes que o laço mais interno de um algoritmo é repetido ou a quantidade de vezes que uma chamada recursiva é realizada.

De fato são duas grandezas distintas, porém estamos trabalhando com diversos algoritmos e não há uma única relação que seja comum a todos eles. Pensando no conceito de complexidade de algoritmos, estaremos avaliando em cada caso qual a parte mais complexa da sua execução.

Em todas as funções de ordenação descritas neste livro foi incluída uma variável chamada **qtd**. Ela será responsável por contar o esforço computacional aplicado na ordenação de cada um dos vetores.

A primeira leva de dados será obtido por meio da execução dos seis algoritmos em quatro vetores de dez posições mostrados na Figura 11. O primeiro a) terá uma ordenação totalmente aleatória; b) trará um vetor parcialmente ordenado, c) é um vetor totalmente ordenado; e d) um vetor ordenado de forma inversa.

a)	21	48	22	44	40	16	21	10	17	12
b)	0	1	2	3	4	16	21	10	17	12
c)	0	1	2	3	4	5	6	7	8	9
d)	9	8	7	6	5	4	3	2	1	0

Figura 11 - Vetores para a primeira bateria de testes de ordenação

Na primeira bateria de testes (Figura 12), observamos que o Bubblesort e o Selectionsort obtiveram o mesmo desempenho. Ambos executam o laço interno até o final, independentemente da situação do vetor em cada passagem. O Shellsort usou metade do esforço que o Insertionsort, o que é muito interessante, pois sabemos que o Shellsort faz várias chamadas do Insertionsort durante a sua execução.

Algoritmo	Esforço Computacional
Bubblesort	45
Selectionsort	45
Mergesort	19
Quicksort	13
Insertionsort	34
Shellsort	15

Figura 12 – Testes realizados com os dados do vetor a)

A segunda bateria de testes foi realizada com dados parcialmente ordenados (Figura 13). Tanto o Shellsort com o Insertionsort apresentam desempenho superior aos demais. O Bubblesort e o Selectionsort foram realizados com o mesmo esforço do teste anterior.

Algoritmo	Esforço Computacional
BubbleSort	45
SelectionSort	45
MergeSort	19
QuickSort	19
InsertionSort	6
ShellSort	6

Figura 13 – Testes realizados com os dados do vetor b)

O resultado é ainda mais impressionante com os dados totalmente ordenados (Figura 14). Tanto o Insertionsort como o Shellsort fazem apenas uma única passagem pelo vetor de dados sem realizar nenhuma troca. Você deve ter percebido que o Bubblesort e o Selectionsort apresentam sempre o mesmo desempenho, independente da forma como os dados estão armazenados.

Algoritmo	Esforço Computacional
Bubblesort	45
Selectionsort	45
Mergesort	19
Quicksort	21
Insertionsort	0
Shellsort	0

Figura 14 – Testes realizados com os dados do vetor c)

O último teste traz o vetor com os dados ordenados de forma decrescente (Figura 15), nesse momento vemos novamente a superioridade do Shellsort em relação ao seu parente próximo, Insertionsort.

Algoritmo	Esforço Computacional
Bubblesort	45
Selectionsort	45
Mergesort	19
Quicksort	19
Insertionsort	45
Shellsort	13

Figura 15 - Testes realizados com os dados do vetor d)

O Quicksort, teve um desempenho bem variado, mostrando-se muito eficiente em alguns casos e de desempenho médio em outros.

Vamos agora "engrossar um pouco o caldo". Escolhemos vetores pequenos com tamanho 10 para que o aluno possa reproduzir o teste com os mesmos valores no seu computador. Agora vamos fazer uma nova bateria um pouco mais ousada. A Figura 16 traz uma relação de massas de dados maiores. Usaremos o nosso programa de ambiente de testes para gerar vetores aleatórios nas quantidades descritas e, aplicaremos cada um dos algoritmos e mediremos os seus respectivos esforços.

Teste	Tamanho
Teste a	100
Teste b	1.000
Teste c	10.000

Figura 16 - Testes com maiores quantidades de dados

Analisando os dados das Figuras 17, 18 e 19 podemos fazer diversas conclusões. A primeira é que os algoritmos Bubblesort e Selectionsort são os mais lentos e têm o seu consumo computacional exponencial. Quanto maior a massa de dados, maior a quantidade de esforço necessário para a ordenação.

Algoritmo	Esforço Computacional
Bubblesort	4.950
Selectionsort	4.950
Mergesort	199
Quicksort	141
Insertionsort	2.261
Shellsort	328

Figura 17 - Testes para 100 valores aleatórios

O Shellsort mostra desempenho superior ao do Insertionsort mesmo que o primeiro faça inúmeras chamadas ao segundo durante o seu funcionamento.

Algoritmo	Esforço Computacional
Bubblesort	499.500
Selectionsort	499.500
Mergesort	1.999
Quicksort	1.903
Insertionsort	241.493
Shellsort	5.959

Figura 18 - Testes para 1.000 valores aleatórios

O Quicksort, que se mostrou mediano nos vetores de dez posições, obteve o melhor desempenho em todos os testes com grandes quantidades de dados. O Mergesort, que também utiliza o conceito de dividir para conquistar, não é tão rápido quanto o Quicksort, mas tem desempenho superior aos outros métodos de ordenação aqui apresentados.

Algoritmo	Esforço Computacional
Bubblesort	49.995.000
Selectionsort	49.995.000
Mergesort	19.999
Quicksort	19.903
Insertionsort	24.722.325
Shellsort	71.682

Figura 19 - Testes para 10.000 valores aleatórios

Quando é preciso ordenar um pequeno volume de dados, você pode se dar ao luxo de fazer rapidamente implementações mais simples, porque o tempo de execução é tão pequeno que compensa o tempo economizado na hora de codificar um método mais complexo.

Já para o caso de estarmos ordenando índices de arquivos grandes, como o usado no sistema operacional para localizar arquivos, ou por bancos de dados para aperfeiçoar a busca em suas tabelas, a implementação de algoritmos mais rápidos se torna crucial.

ATIVIDADES

1. Explique o funcionamento do Mergesort.

2. Qual a semelhança entre o Quicksort, Mergesort e Árvore Binária de Busca?

3. Qual a relação entre a estrutura de dados Heap e o algoritmo Heapsort?

Ordenação digital de strings de mesmo comprimento

por Paulo Fileoff

Suponha dado um vetor v[0..n-1] de strings, todas do mesmo comprimento, e considere o problema de rearranjar o vetor em ordem lexicográfica (você pode imaginar que todas as strings são ASCII, mas isso não é essencial). No contexto desse problema, os elementos (bytes) das strings são tradicionalmente chamados dígitos, mesmo que não pertençam no conjunto 0..9.

Exemplo. Suponha que as strings são placas de licenciamento de automóveis. (Poderíamos nos restringir aos quarenta e três caracteres que vão de '0' a 'Z'), segue um exemplo com 10 placas de 7 dígitos cada:

```
FON1723
EAD3312
CDA7891
FAJ4021
DOG1125
BAT7271
GIZ1234
BAT7328
BIG8733
CAT9955
```

Para colocar esse vetor em ordem lexicográfica, podemos usar ordenação por contagem repetidas vezes: primeiro para ordenar pelo último dígito, depois para ordenar pelo penúltimo dígito, e assim por diante:

CDA7891	EAD3312	FAJ4021	DOG1125	CDA7891	EAD3312	BAT7271
FAJ4021	FAJ4021	DOG1125	GIZ1234	EAD3312	FAJ4021	BAT7328
BAT7271	FON1723	GIZ1234	FON1723	DOG1125	BAT7271	BIG8733
EAD3312	DOG1125	BAT7271	EAD3312	BIG8733	BAT7328	CAT9955
FON1723	BAT7328	EAD3312	FAJ4021	FAJ4021	CAT9955	CDA7891
BIG8733	BIG8733	BAT7328	BAT7271	FON1723	CDA7891	DOG1125
GIZ1234	GIZ1234	FON1723	BAT7328	BAT7271	BIG8733	EAD3312
DOG1125	CAT9955	BIG8733	CDA7891	BAT7328	GIZ1234	FAJ4021
CAT9955	BAT7271	CDA7891	BIG8733	CAT9955	DOG1125	FON1723
BAT7328	CDA7891	CAT9955	CAT9955	GIZ1234	FON1723	GIZ1234

Como a ordenação por contagem é estável, o vetor de strings acaba ficando em ordem lexicográfica. Observe que é essencial aplicar a contagem da direita para a esquerda, ou seja, começando pelo último dígito.

Algoritmo. O código a seguir implementa o algoritmo sugerido no exemplo. O parâmetro W representa o comprimento comum de todas as strings (7 no exemplo acima) e o parâmetro R (menor que 256) especifica o tamanho do universo de dígitos:

```
 1  typedef unsigned char byte;
 2  // Rearranja em ordem lexicográfica um vetor v[0..n-1]
 3  // de strings. Cada v[i] é uma string v[i][0..W-1]
 4  // cujos elementos pertencem ao conjunto 0..R-1.
 5  void
 6  ordenacaoDigital (byte **v, int n, int W, int R)
 7  {
 8     int *fp;
 9     byte **aux;
10     fp = malloc ((R+1) * sizeof (int));
11     aux = malloc (n * sizeof (byte *));
12     for (int d = W-1; d >= 0; --d) {
13        int r;
14        for (r = 0; r <= R; ++r)
15           fp[r] = 0;
16        for (int i = 0; i < n; ++i) {
17           r = v[i][d];
18           fp[r+1] += 1;
19        }
20        for (r = 1; r <= R; ++r)
21           fp[r] += fp[r-1];
22        for (int i = 0; i < n; ++i) {
23           r = v[i][d];
24           aux[fp[r]] = v[i];
25           fp[r]++;
26        }
27        for (int i = 0; i < n; ++i)
28           v[i] = aux[i];
29     }
30     free (fp);
31     free (aux);
32  }
```

O código ignora o byte nulo, \0, que marca o fim de cada string e portanto pode ser aplicado a qualquer vetor de vetores de bytes (desde que o valor de cada byte esteja no intervalo 0..R-1).

Esse algoritmo de ordenação é digital porque ordena as strings dígito-a-dígito. O algoritmo também é conhecido como:

- ordenação digital da direita para a esquerda,
- ordenação digital a partir do dígito menos significativo, e
- LSD (l*east significant digit*) Radixsort.

Fonte: Feofiloff (2018, on-line[1]).

MATERIAL COMPLEMENTAR

NA WEB

Dança simulando uma ordenação por Mergesort
<http://www.youtube.com/watch?v=XaqR3G_NVoo>.

Dança simulando uma ordenação por Quicksort
<http://www.youtube.com/watch?v=ywWBy6J5gz8>.

Comparação da eficiência e tempo computacional do Bubblesort e Quicksort
<http://www.youtube.com/watch?v=F-7kk4lY_mQ>.

Vídeo mostrando várias técnicas de ordenação
<http://www.youtube.com/watch?v=PbuuRLD_tg4>.

Professor Me. Pietro Martins de Oliveira
Professor Me. Rogério de Leon Pereira

UNIDADE V

OPERAÇÕES DE BUSCA

Objetivos de Aprendizagem

- Aprender os conceitos básicos de busca.
- Estudar diferentes algoritmos de busca.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Conceitos básicos
- Operação de busca sequencial
- Busca sequencial indexada
- A busca binária
- Busca por interpolação
- Tabelas de Dispersão (*Hash*)

Unicesumar

INTRODUÇÃO

Nesta unidade, estudaremos alguns métodos de como pesquisar grandes quantidades de dados em busca de uma determinada informação. Veremos como a organização dos dados torna o processo de busca mais eficiente. Como a operação de busca é uma tarefa muito comum na ciência da computação, o conhecimento desses métodos é de suma importância para todo o profissional que deseja se tornar um bom programador.

CONCEITOS BÁSICOS

Antes de nos aprofundarmos no estudo de algoritmos de busca, vamos tratar de alguns termos que utilizaremos nesta unidade. Uma **tabela** ou **arquivo** é um conjunto de **registros**. Um registro é um conjunto de **dados**.

Existe uma **chave** associada a cada registro, se ela estiver dentro do próprio registro ela é chamada de **chave interna**, **chave embutida** ou **incorporada**. No caso da chave ficar fora do registro ela é chamada de **chave externa**.

Para todos os registros dentro de uma tabela, pelo menos um dos campos precisa ter um valor único para assim diferenciar cada um dos registros. Esse campo é chamado de **chave primária**.

Por exemplo, vamos imaginar que temos uma tabela de clientes. Cada registro dessa tabela representa um cliente e cada cliente possui um conjunto de dados, conforme a Tabela 1.

Tabela: clientes	
Registro	CPF
	Nome
	Rua
	CEP
	Telefone

Tabela 1 - Exemplo da Tabela clientes e os dados de um Registro

Posso identificar os clientes, de maneira única, pelo **nome**? Não pode, porque existem homônimos, ou seja, pessoas com o mesmo nome, como José Maria da Silva, Roberto Carlos dos Santos etc.

Também não posso usar a **Rua** ou o **CEP**, pois vários clientes podem morar na mesma localidade. Se eu e a minha esposa formos clientes dessa empresa, nosso **Telefone** também aparecerá mais de uma vez na tabela.

Para esse caso específico, sobrou o **CPF** (Cadastro de Pessoa Física), que é realmente único, não existem duas pessoas no Brasil com o mesmo CPF. Se nenhum dos campos do registro fosse potencial para servir de chave primária, poderíamos criar um novo campo, chamado **codigo** ou **id**, e atribuir um número único para cada um dos registros.

Ainda pensando nesse exemplo, imagine que a empresa dona dessas informações pretenda fazer uma ação de marketing e tenha como objetivo atingir apenas os clientes que residam numa determinada cidade, bairro ou numa rua específica. Posso estar fazendo uma busca na Tabela clientes pelos campos Rua ou CEP. Quando um atributo com informações não únicas é utilizado numa pesquisa ele é chamado de **chave secundária**.

REFLITA

Todo campo de um registro em uma tabela pode servir como chave em potencial, dependendo da informação desejada ou da regra de busca utilizada.

Como você deve ter aprendido, as grandes massas de dados ficam armazenados em dispositivos de memória secundária, como disco rígido, disco óptico, fita magnética, memória flash, dentre outros.

Entretanto todo dado, para ser manipulado, precisa necessariamente estar na memória principal. Como a relação entre as duas memórias é desproporcional, ou seja, um computador tem Gb de memória RAM enquanto possui Tb de memória de armazenamento, somente uma pequena parcela dos dados pode ser analisada por vez.

Dessa forma, é muito mais inteligente carregar apenas uma tabela de chaves do que todos os registros de um arquivo. Encontrada a referência desejada, busca-se no disco apenas a porção desejada de informação.

Alguns algoritmos são especializados em realizar buscas em arquivos de chaves primárias, outros por chaves secundárias. Alguns são mais rápidos em arquivos menores, outros ideais para grandes quantidades de informações.

Um algoritmo de busca tem como objetivo, a partir de um argumento **x** recebido, encontrar numa tabela um registro que possua uma chave **x** equivalente. Ele irá retornar todos os dados do registro, ou o índice da sua posição num vetor ou ainda um ponteiro para uma posição na memória.

Se não houver no arquivo um registro compatível com o argumento pesquisado, o algoritmo não obtém sucesso na sua busca e retorna uma mensagem de erro, um índice inválido ou um ponteiro nulo. Segundo Tenenbaum (1995), uma busca com sucesso é chamada **recuperação**.

OPERAÇÃO DE BUSCA SEQUENCIAL

A **busca sequencial** é de longe a forma mais simples de pesquisa. Ela pode ser utilizada tanto para o caso em que a tabela está armazenada num vetor como numa lista ligada.

A partir da primeira posição no vetor (ou do primeiro nó da lista), compara-se o valor atual da estrutura com o argumento **x** passado. Repete-se esse procedimento com cada um dos valores até que seja encontrado o valor desejado ou o final da tabela.

O Programa 5.1 traz um exemplo de busca sequencial em C. A função **buscaSequencial** recebe três parâmetros: o vetor **vec** a ser pesquisado, o argumento **arg** a ser procurado e o tamanho **tam** do vetor.

São criadas duas variáveis locais. A variável **i** serve de índice para percorrer o vetor, enquanto a variável **achou** é usada para identificar se a busca foi concluída antes do final do vetor.

Como os vetores em C começam pela posição **0**, a variável **i** também foi inicializada com esse valor, para que a pesquisa realmente dê início pelo começo do vetor. A variável **achou** foi inicializada com **-1**.

Durante o laço de repetição é verificado se a variável de controle **i** atingiu o tamanho máximo **tam** recebido como parâmetro. O valor de **i** é incrementando sempre no final do laço, garantindo que a busca percorra o vetor até o final. Adicionalmente o laço possui outra regra de parada que verifica o valor da variável **achou**. Ela foi inicializada com **-1**, porém, se o valor for encontrado, a sua posição no vetor será atribuída ao valor de **achou**.

Ao final a função retorna o valor da variável **achou**. Ele será **-1** se a busca tiver falhado ou, no caso contrário, a posição do registro no vetor.

Programa 5.1 - Busca Sequencial

```
//Função de Busca Sequencial
int buscaSequencial(int vec[], int arg, int tam){
  int i = 0, achou = -1;
  while ((i < tam) && (achou == -1)){
    if (vec[i]==arg){
      achou = i;
    }
    i++;
  }
  return(achou);
}
```

Se considerarmos que os registros no vetor não estão organizados de forma alguma, a pesquisa poderá encontrar o valor desejado na primeira ou na última posição da estrutura. Imagine que a tabela seja muito grande e buscar alguma informação procurando em todos os registro pode ter um custo computacional muito elevado. Uma maneira de melhorar isso é a técnica de busca sequencial indexada.

BUSCA SEQUENCIAL INDEXADA

Não é possível prever qual será o valor buscado numa pesquisa e nem a posição onde o registro está armazenado no vetor, menos ainda se ele está ou não presente na tabela.

Tudo depende do que está armazenado e o que se costuma pesquisar. Se numa tabela de clientes as buscas são feitas por pessoas que habitem na mesma região demográfica para efetuar ações mais precisas e eficazes de marketing, uma boa ideia seria criar um índice específico ordenado por cidade, estado ou CEP.

Se o síndico do prédio onde você mora entrega o boleto do condomínio começando pelos apartamentos do último andar até o térreo, seria interessante que houvesse um índice com o número do apartamento ordenado de forma decrescente.

Em uma loja de departamentos especializada em produtos de moda feminina é muito provável que o usuário busque uma cliente do sexo feminino ao invés de um homem, por isso os dados devem ser ordenados primeiro por gênero.

Se for observado que alguns argumentos específicos sejam procurados com mais frequência, esses registros podem ser movidos para uma região mais próxima do início da tabela, fazendo com que futuras buscas sejam cada vez mais rápidas. Tenenbaum (1995) fala de um método em particular, muito simples e que traz bons resultados, chamado método da **transposição,** no qual o registro recuperado com sucesso é trocado pelo registro imediatamente anterior.

Imagine um vetor de valores inteiros que esteja ordenado de forma crescente. A busca sequencial procuraria o valor desejado em cada uma das posições até encontrá-lo ou até chegar ao final do arquivo, o que ocorrer primeiro. Mesmo no caso do registro não existir na tabela, todo o arquivo será percorrido.

Entretanto como sabemos que a estrutura está ordenada de forma ascendente, no momento em que for encontrado um número maior do que o que está sendo procurado, não há mais necessidade de continuar a pesquisa, porque certamente o valor procurado não estará no restante da tabela. Nessa linha de raciocínio, podemos dizer que compensa classificar um arquivo antes de pesquisar informações dentro dele.

A técnica de **busca sequencial indexada** consiste em criar uma tabela auxiliar ao arquivo de dados. Nessa tabela, deve constar pelo menos dois campos: a chave de busca e o endereço do registro na tabela original. A tabela de índice pode ser ordenada de forma ascendente, descendente ou de qualquer outra forma que seja mais compatível com as principais buscas que utilizarão a chave contida na tabela auxiliar.

Mais uma vez mencionando uma suposta tabela de clientes como exemplo, fisicamente os registros estão gravados em disco pela ordem como os clientes são cadastrados no sistema. Se a busca for efetuada pelo número do CPF do cliente, uma nova tabela é criada contendo o CPF e a posição original do registro na tabela de clientes.

REFLITA

Se todos os valores de um vetor estiverem ordenados de forma crescente, uma pesquisa pode ser interrompida assim que um valor maior do que o procurado for visitado. O mesmo acontece de forma análoga para um vetor ordenado de forma decrescente.

A BUSCA BINÁRIA

A forma mais eficiente de efetuar pesquisa em um arquivo ordenado sem a necessidade de tabelas auxiliares é a **busca binária**. A estratégia consiste em comparar o argumento chave ao elemento do meio da tabela. Se forem iguais, a busca terá terminado com sucesso. No caso contrário, o vetor será divido em duas metades, e a pesquisa será repetida na metade inferior, se o argumento for menor do que o valor do meio da tabela, ou na parte superior, se o argumento for maior.

A cada iteração, a busca binária reduz a quantidade de possíveis candidatos pela metade. Vale ressaltar que essa estratégia funciona apenas em estruturas cujos dados se encontram ordenados. Sua implementação em vetores não é muito complexa e um exemplo em C é apresentado no Programa 5.2.

Esse algoritmo não pode ser utilizado com listas dinâmicas devido às características da sua estrutura. Mesmo criando um ponteiro especial que aponte para o meio da lista, toda vez que o problema for dividido, a lista precisará ser percorrida novamente para achar o meio da nova metade da lista, o que torna o tempo de execução do algoritmo muito extenso.

Programa 5.2 - Busca Binária

```
//Função de Busca Binária
int buscaBinaria(int vec[], int arg, int tam){
  int menor, maior, meio;
  menor = 0;
  maior = tam-1;
  while (menor <= maior){
    meio = (menor + maior)/2;
    if (arg == vec[meio]){
      return(meio);
    }
    if(arg < vec[meio]){
      maior = meio - 1;
    }
    else {
      menor = meio + 1;
    }
  }
  return(-1);
}
```

A função **buscaBinaria** do Programa 5.2 recebe três parâmetros: o vetor **vec** que será pesquisado, o argumento **arg** que será procurado e o tamanho **tam** do vetor. São criadas três variáveis internas, uma para marcar a posição de início (**menor**) da pesquisa, o limite a ser pesquisado (**maior**) e o **meio** do vetor.

A variável **menor** é inicializada com valor **0**, que na linguagem C representa o início do vetor. A variável **maior** é inicializada com o valor **tam-1,** posto que um vetor de tamanho **n** vai de **0** até **n-1**.

O laço será repetido enquanto a variável **menor** for menor ou igual a variável **maior**. Nele será calculada a posição do **meio** entre **menor** e **maior**. Se o argumento **arg** estiver na posição **meio** do vetor **vec,** a função retorna o valor de **meio** e o algoritmo concluiu a **recuperação**.

Se a recuperação ainda não foi concluída, é preciso atualizar o valor da variável **meio**. Como sabemos que o vetor se encontra ordenado, se o argumento **arg** for menor do que o conteúdo no vetor **vec** na posição **meio**, então a busca deverá ser feita entre a posição **menor** e a metade. Caso contrário, a busca deve continuar entre a metade e o final do vetor.

Vamos simular uma busca binária num vetor ordenado de 10 posições, demonstrado na Figura 1. O valor do argumento a ser encontrado é **5**.

a) A variável **menor** será inicializada com **0** e **maior** com **9** (**tamanho – 1**).

b) Para o cálculo do meio do vetor fazemos:

$$meio = \frac{(menor + maior)}{2}$$

$$meio = \frac{(0+9)}{2}$$

$$meio = \frac{(9)}{2}$$

$$meio = 4,5$$

Como a variável **meio** é do tipo inteiro, a posição que indica a metade do vetor é **4**. O valor contido no vetor na posição **meio** é **16**, que é maior do que o argumento pesquisado **5**.

c) Como o valor não foi encontrado, o laço é repetido e a busca se dá entre as posições 0 a 3.

$$meio = \frac{(menor + maior)}{2}$$

$$meio = \frac{(0+3)}{2}$$

$$meio = \frac{(3)}{2}$$

$$meio = 1,5$$

Como a variável **meio** é do tipo inteira, a posição que indica a metade do vetor é **1**. O valor contido no vetor na posição **meio** é **7**, que é maior do que o argumento pesquisado **5**.

d) Como o valor não foi encontrado, o laço é repetido e a busca se dá na parte restante do vetor, que é a posição 0.

O vetor na posição **0** contém o valor procurado **5**, o algoritmo finaliza com sucesso recuperando a posição no vetor onde o argumento está armazenado.

a)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	16	20	21	30	31	33

b)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	16	20	21	30	31	33

c)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	16	20	21	30	31	33

d)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	16	20	21	30	31	33

Figura 1 - Simulação de uma Busca Binária pelo elemento 5 num vetor de dez posições

BUSCA POR INTERPOLAÇÃO

A busca por interpolação é outra forma de pesquisar em vetores com dados ordenados. Se os valores estiverem distribuídos de forma uniforme entre **vec[menor]** e **vec[maior],** esse método pode ser ainda mais eficiente do que a busca binária.

Sua implementação é muito parecida com a busca binária que acabamos de estudar. A variável **menor** é definida com **0** e a **maior** é definida com **tamanho – 1**. Sabendo que os valores estão distribuídos de forma uniforme e tomando por verdade que em toda a execução do algoritmo o argumento **arg** estará contido entre os valores **vec[menor]** e **vec[maior],** podemos esperar que **arg** esteja aproximadamente na posição:

$$meio = menor + (maior - menor) * \left(\frac{\arg - vec[menor]}{vec[maior] - vec[menor]} \right)$$

A busca deve continuar até que o valor seja encontrado ou que **menor > maior**. Se **arg** for menor que **vec[meio]**, o valor da variável **maior** será atualizada com **meio -1**, fazendo com que a busca aconteça na parte inferior do vetor. Caso contrário, a variável **menor** será atualizada com **meio + 1,** para que a busca prossiga na parte superior da estrutura. O Programa 5.3 demonstra uma implementação de busca por interpolação em linguagem C.

Programa 5.3 - Busca por Interpolação

```
//Função de busca por interpolação
int buscaInterpol(int vec[], int arg, int tam){
  int menor, maior, meio, achou;
  menor = 0;
  maior = tam-1;
  achou = -1;
  while((menor <= maior) && (achou == -1)){
    meio = menor + (maior - menor) *
    ((arg - vec[menor])/vec[maior] - vec[menor]);
    if (arg == vec[meio]){
      achou = meio;
    }
    if(arg < vec[meio]){
       maior = meio - 1;
    }
```

```
else {
        menor = meio + 1;
    }
  }
  return(achou);
}
```

REFLITA

A busca por interpolação pode ser mais rápida do que uma busca binária se os valores no vetor estiverem distribuídos de forma uniforme. Caso contrário, ela pode ser tão lenta como uma busca sequencial.

TABELA DE DISPERSÃO

Todos os métodos de busca que vimos até então realizam a comparação de elementos do arranjo, dois a dois, para procurar a chave de busca. Todavia, existe uma estrutura de dados pensada para eliminar a necessidade de realizar comparações durante as buscas por dados. Existe uma maneira de realizar um cálculo que indica exatamente (ou quase) a posição do elemento chave dentro do arranjo de dados, sem realizar comparações. É como se, a partir da chave de busca, fosse possível aferir o índice da posição no qual o elemento igual àquela chave se encontra em um vetor, por exemplo.

Drozdek (2008), comenta que é preciso desenvolver uma função **h()** que processe uma chave particular **k** em um índice que possa ser utilizado para armazenar itens em um vetor que seja do mesmo tipo de **k** (int, float, char, etc). Chama-se h(), função de dispersão, função de escrutínio, ou função *hash*. Em suma, para nós, a função *hash* deve ser capaz de converter um dado, ou um conjunto de dados, em um índice, que é a posição na qual tais dados serão armazenados (DROZDEK, 2008). Em outras ocasiões, uma função *hash* pode gerar

um código utilizado em aplicações que envolvam criptografia, todavia esse não é o foco de nosso estudo nesse momento.

Por exemplo, imagine um vetor com dez posições, que vão de 0 a 9. Suponha que queiramos armazenar dados do tipo inteiro em nossa tabela de busca. Assim, poderíamos pensar em uma função h() tal que, realizando cálculos matemáticos sob o elemento a ser armazenado (ou buscado), cheguemos a um valor inteiro que varie entre 0 e 9, assim como o índice do vetor.

Considere, a título de ilustração, que nossa função h() recebe um parâmetro de entrada **x**, do tipo int, e retorna o cálculo da expressão "**(x*x) % 10**". Ou seja, nossa função *hash* retorna o resto de divisão do quadrado do parâmetro de entrada. Assim, para inserir o número 72 em nosso vetor, temos de realizar o cálculo "72 * 72 % 10", de acordo com a expressão de retorno de nossa função *hash*. Assim sendo, ao realizar a conta, descobrimos que h(72) é igual a 4. Por isso, o elemento 72 deve ser armazenado na posição 4 do vetor. Matematicamente, podemos dizer que $\mathbf{h(x) = x^2 \bmod 10}$, nesse caso.

Observe, na Figura 2, como ficou o vetor após a inserção dos elementos 11, 23, 26 e 72. O vetor que podemos visualizar na porção direita dessa imagem funciona como uma tabela, na qual os dados são armazenados de acordo com o resultado do processamento de uma chave pela função *hash*. Damos o nome a esse tipo de estrutura de dados de Tabela *Hash*, Tabela de Dispersão ou Tabela de Escrutínio.

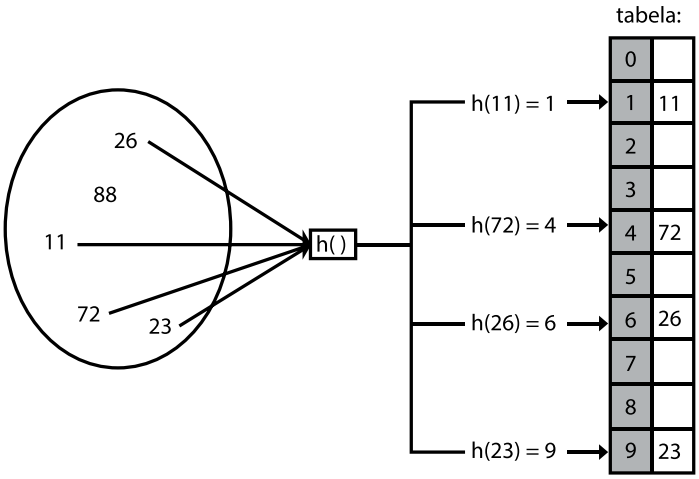


Figura 2 - Inserção de dados em uma Tabela *Hash*.

Para compreender melhor, tomemos o caso da inserção do elemento 23. Durante o processamento, a função h() encontra o quadrado de 23 que é igual a 529. Em seguida, a função *hash* divide 529 por 10, encontrando um quociente igual a 52 (que não nos interessa, nesse caso), e um resto de divisão igual a 9. Assim, o resultado da execução da função h(), passando o argumento 23, é 9. Por isso, o elemento 23 se encontra armazenado na posição 9 da tabela. Esse mesmo processo é repetido para todo elemento que é inserido.

Após os elementos terem sido inseridos em nosso vetor, para realizar uma busca, basta utilizar a função *hash*, novamente, de maneira similar ao que foi feito no momento da inserção. Ou seja, se quisermos realizar uma busca pelo elemento 11, por exemplo, devemos invocar a função *hash* para que, nesse caso, saibamos que o elemento 11 se encontra na posição 1 da tabela.

Por outro lado, se tentassémos buscar um elemento ausente, digamos, o 35, o que aconteceria? Bem, ao processar o elemento 35 de acordo com nossa função *hash*, descobriremos que o 35 deveria estar armazenado na posição 5. Todavia, como vemos na Figura 2, o 35 não se encontra armazenado nessa posição. Assim, o algoritmo de busca conclui que o elemento está ausente.

Apesar disso tudo, será que não há algum problema com nossa tabela? Repare que, na Figura 2, o elemento 88 não foi armazenado no vetor, propositalmente. Ao tentar inserir o elemento 88, verificaríamos que o resultado de h(88) é igual a 4. Com isso teríamos de sobrescrever o dado que se encontra armazenado na posição 4. Damos o nome de colisão a fatos como esse. Sempre que duas ou mais chaves, depois de processadas pela função *hash*, apresentam um resultado igual, temos uma colisão.

Existem diversas técnicas para resolver o problema das colisões em tabelas *hash*. Pense que se, quando houver uma colisão, decidimos procurar pela próxima posição livre no vetor, para armazenar o dado conflitante. Dessa forma, quando for fazer a busca, ao calcular o *hash* da chave, não bastaria apenas verificar a posição indicada pela função *hash*, mas também as posições subsequentes. Observe a Figura 3, a seguir.

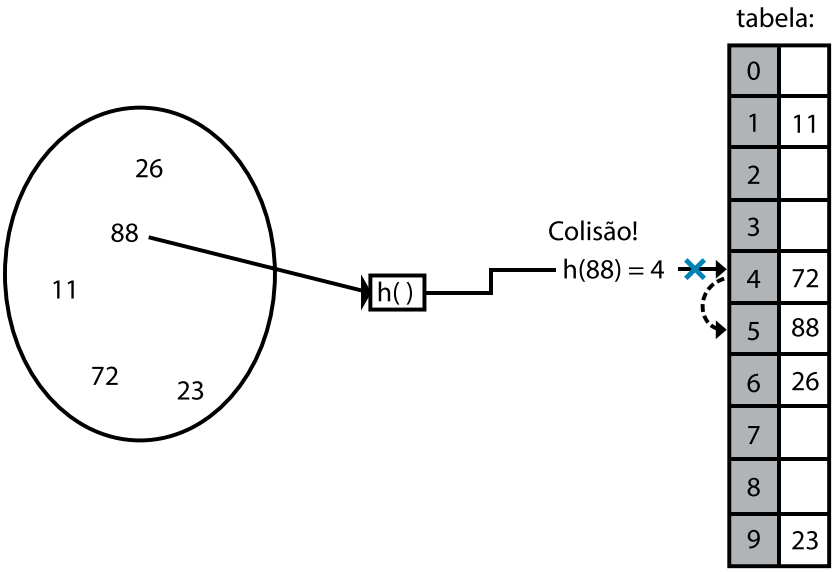


Figura 3 - Resolvendo o problema de colisão em uma inserção

Essa é uma solução interessante, na medida em que a quantidade de dados a serem armazenados é menor que o tamanho do vetor estático. Quando tentamos inserir o 88 na posição 4 por meio da função *hash*, tivemos uma colisão. Nesse instante, nosso algoritmo pode tentar procurar pela próxima posição livre dentro do arranjo, após a colisão, para armazenar o elemento em questão.

Dessa forma, ao tentar realizar uma busca pelo elemento 88, a função *hash* retornaria à posição 4. Ao verificar a posição 4, notamos que quem se encontra lá é o 72. Todavia, nosso algoritmo sabe que pode ter havido alguma colisão. Por isso, a busca continua sequencialmente, verificando se existe algum elemento posterior ao 72 que seja igual ao 88. Para nossa sorte, o 88 está armazenado exatamente na posição seguinte, encerrando a busca com sucesso, na posição 5. Por outro lado, o algoritmo pode interromper a busca sequencial caso atinja alguma posição na qual não haja elemento armazenado.

Considere, ainda, que os elementos em uma tabela *hash* podem ser removidos, além de inseridos. Imagine que inserimos o 88 na posição 5 para solucionar a colisão porém, em seguida, o elemento 72 é removido. Após isso, ao tentar buscar pelo elemento 88, tentaremos acessar a posição 4 devido à função *hash*

e perceberemos que é uma posição vazia. A busca não pode ser interrompida quando um elemento está ausente por causa de uma remoção, pois o elemento alvo pode ter sido inserido em um momento que ocasionava uma colisão. Por isso, nesse tipo de solução de colisões, é interessante que todo o vetor possua flags que sinalizam se uma posição vazia é resultado de uma remoção ou não.

Outro método de solução de colisões consiste em combinar o caráter estático de vetores com o potencial de alocação dinâmico dos ponteiros. Assim, caso haja uma colisão, podemos "pendurar" mais elementos naquela posição do vetor de maneira dinâmica. Ou seja, o vetor passa a ser uma tabela na qual cada posição do vetor pode dar origem a uma lista encadeada distinta. Observe o problema da colisão do número 88 solucionada por meio de ponteiros, na Figura 4.

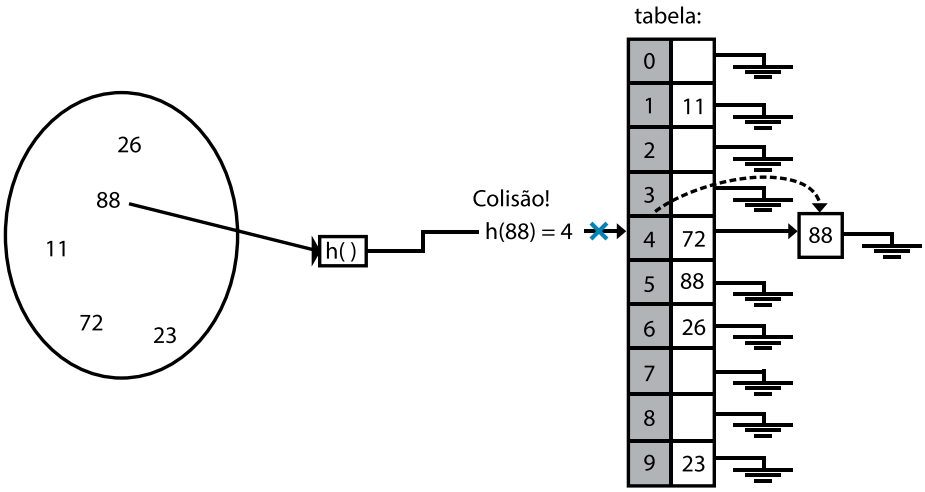


Figura 4 - Solucionando colisões através de alocação dinâmica

A Figura 4 ilustra de maneira clara o que pode ser feito em caso de colisão. Antes de tudo, cada elemento deve ser um registro que contém ao menos um campo de "dados" e outro campo ponteiro para apontar para o próximo elemento (em caso de colisões). Assim, a Tabela *Hash* passa a ser um vetor de struct que pode ser flexível em seu tamanho, fazendo com que colisões resultem na criação de listas encadeadas.

CONSIDERAÇÕES FINAIS

Nesta unidade, aprendemos sobre vários métodos de busca. Começamos definindo o que é uma tabela (ou arquivo), registros e dados. Falamos também sobre conceito de chave, chaves internas e externas, chaves primárias e secundárias.

Vimos a busca sequencial, que é de simples implementação, mas de alto custo computacional. No pior caso, quando o valor pesquisado não se encontra na tabela ou quando este está localizado na última posição da mesma, o algoritmo terá que percorrer mandatoriamente todos os registros ali armazenados.

Estudamos também a busca sequencial indexada. Ela é um pouco melhor do que a busca sequencial, pois podemos prever se o valor está ou não contido na tabela mesmo antes de percorrê-la por inteiro.

Foi apresentada ao aluno a busca binária que traz o conceito **dividir para conquistar**. A cada passo do algoritmo, a região de busca é dividida pela metade, fazendo com que a pesquisa seja muito rápida. Essa técnica só é possível de ser implementada em estruturas estáticas e precisa que os dados estejam previamente ordenados.

O algoritmo de busca por interpolação pode melhorar ainda mais a busca binária se os dados estiverem distribuídos de forma uniforme no vetor.

Por último, vimos a Tabela *Hash*, que é um tipo de estrutura de dados semelhante a um vetor. Nesse tipo de estrutura de dados, temos o conceito de função de dispersão, também conhecida como função *hash*. Essa função é responsável por processar a chave de busca e retornar, de maneira direta, à posição do elemento indicado pela chave de busca. As principais formas e as técnicas mais rápidas são baseadas em indexação e ordenação prévia dos dados.

ATIVIDADES

1. Como funciona e quais as características da Busca Sequencial?

2. Quais as vantagens de realizar o método da Busca Sequencial Indexada?

3. Por que a Busca Binária é mais eficiente quando aplicada em estruturas com valores indexados?

4. Quais as dificuldades de implementar uma Busca Binária utilizando estrutura de listas dinâmicas?

5. Explique a Busca por Interpolação.

6. Por que é possível implementar a Busca em Árvore utilizando estruturas alocadas dinamicamente posto que o processo é semelhante à Busca Binária?

MATERIAL COMPLEMENTAR

LIVRO

Projeto de Algoritmos
Nivio Ziviani

Editora: Cengage Learning
Sinopse: a obra apresenta os principais algoritmos conhecidos. Algoritmos e estruturas de dados formam o núcleo da ciência da computação, sendo os componentes básicos de qualquer software. Aprender algoritmos é crucial para qualquer pessoa que deseja desenvolver um software de qualidade.
Destaques

- As técnicas de projeto de algoritmos são ensinadas por meio de refinamentos sucessivos até o nível de um programa em Pascal.
- Todo programa Pascal tem um programa C correspondente no apêndice.
- Mais de 163 exercícios propostos, dos quais 80 com soluções; 221 programas em Pascal e 221 programas em C; 164 figuras ilustrativas.

NA WEB

Busca Binária
<http://www.youtube.com/watch?v=GEXVka4GIXU>.

REFERÊNCIAS

CORMEN, T. H. *et al*. **Algoritmos**. Rio de Janeiro: Elsevier, 2012.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. São Paulo: Cencage Learning, 2008.

PEIXOTO, P. **Motores de Busca**. Faculdade de Economia, Universidade de Coimbra. 2008. Disponível em: <http://www4.fe.uc.pt/fontes/pesquisa_na_internet/motores_busca/motores_de_busca.htm>. Acesso em: 15 jan. 2019.

TENENBAUM, A. M. *et al.* **Estruturas de dados usando C**. São Paulo: Makron Books, 1995.

WIRTH, N. **Algoritmos e Estruturas de Dados**. Rio de Janeiro: Prentice-Hall do Brasil, 1989.

REFERÊNCIA ON-LINE

[1]Em: <https://www.ime.usp.br/~pf/algoritmos/aulas/radix.html>. Acesso em: 14 jan. 2019.

GABARITO

UNIDADE I

1.

```
//Estrutura
struct str_no {
  char dado;
  struct str_no *pai;
  struct str_no *esquerda;
  struct str_no *direita;
};
```

2.

a)

Raiz	A
Nós	A, B, C, D
Folhas	C, D
Com um filho	B
Com dois filhos	A
Nível 0	A
Nível 1	B, C
Nível 2	D
Nível 3	
Profundidade	2

b)

Raiz	A
Nós	A, B, C, D, E, F, G, H, I
Folhas	D, G, H, I
Com um filho	C, E
Com dois filhos	A, B, F
Nível 0	A
Nível 1	B, C
Nível 2	D, E, F
Nível 3	G, H, I
Profundidade	3

c)

Raiz	A
Nós	A, B, C, D, E, F, G
Folhas	C, D, F, G
Com um filho	
Com dois filhos	A, B, E
Nível 0	A
Nível 1	B, C
Nível 2	D, E
Nível 3	F, G
Profundidade	3

d)

Raiz	A
Nós	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O
Folhas	H, I, J, K, L, M, N, O
Com um filho	
Com dois filhos	A, B, C, D, E, F, G
Nível 0	A
Nível 1	B, C
Nível 2	D, E, F, G
Nível 3	H, I, J, K, L, M, N, O
Profundidade	3

e)

Raiz	A
Nós	A, B, C, D, E, F
Folhas	C, D, F
Com um filho	E
Com dois filhos	A, B
Nível 0	A
Nível 1	B, C
Nível 2	D, E
Nível 3	F
Profundidade	3

3. Uma árvore é considerada estritamente binária se todo nó que não for folha tiver sempre subárvores direita e esquerda não vazias. Exemplo: letra **c** do exercício **2**.

4. Uma árvore é dita binária completa quando for uma árvore estritamente binária e todas as suas folhas estiverem no último nível da estrutura. Exemplo: letra **d** do exercício **2**.

5.

a)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	B	C		D					

b)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	B	C	D	E		F			G
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
			H	I					

c)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	B	C	D	E					F
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
G									

d)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	B	C	D	E	F	G	H	I	J
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
K	L	M	N	O					

e)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	B	C	D	E					
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
F									

UNIDADE II

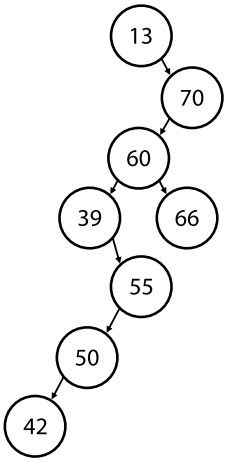
1.

a) 17 – 5 – 2 – 10 – 25 - 32

b) 2 – 5 – 10 – 17 – 25 - 32

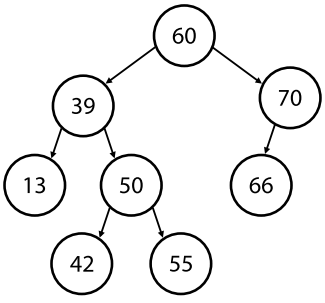
c) 2 – 10 – 5 – 32 – 25 - 17

2.



a) A ordem de visitação para chegar até o 42 é a seguinte: 13, 70, 60, 39, 55, 50, 42. Ou seja, sete nós foram visitados.

3.



a) A ordem de visitação para chegar até o 42 é a seguinte: 60, 39, 50, 42. Ou seja, quatro nós foram visitados. É quase a metade de visitas, quando comparada à árvore binária de busca simples que vimos no exercício anterior.

GABARITO

UNIDADE III

1. A vantagem do algoritmo está na sua simplicidade e facilidade de programação. Tem como desvantagem o alto tempo computacional. Não obstante, a técnica é sempre aplicada de forma completa, mesmo se o vetor já estiver ordenado. A ideia central é de comparar os valores aos pares e ir empurrando o maior valor para o final do vetor.

2. Enquanto o Bubblesort compara os valores de dois a dois e empurra o maior valor para o final, o Insertionsort busca o menor valor em todo o vetor e posiciona-o no início da tabela, em seguida repete o processo para o segundo menor valor, e o terceiro, e assim por diante, puxando os menores valores para o início do arquivo.

3. Todos os três não possuem recursividade e têm dois laços de repetição aninhados. A principal diferença é que o Bubblesort e o Selectionsort repetem os laços por inteiro, quando o Insertionsort tem potencial para executar um número muito menor de repetições.

4. Ambos são de complexidade quadrática, então o tempo computacional será o mesmo. A principal diferença está em como o Shellsort divide repetidamente o vetor em subvetores menores fazendo com que a quantidade de trocas seja menor do que na utilização pura do Insertionsort. Isso acontece porque o Shellsort possibilita a troca de valores entre posições distantes dentro da tabela, e o Insertionsort realiza troca sempre

GABARITO

UNIDADE IV

1. O algoritmo usa a técnica de dividir para conquistar. Ele pega um vetor muito grande e divide-o em dois vetores menores. Aplica o resultado novamente no algoritmo, dividindo-o em partes cada vez menores. Quando não houver mais possibilidade de divisão, a técnica começa a remontar o vetor, dessa vez já ordenado.

2. A partir da raiz da árvore, a árvore binária de busca posiciona os valores menores da raiz na sua subárvore esquerda e os maiores na subárvore direita. O Quicksort, após escolher um elemento pivô, particiona o vetor em dois subvetores, um com valores menores que o pivô e outro com os maiores.

   O Mergesort usa a técnica de dividir para conquistar, dividindo a lista pela metade até que existam apenas listas de um único elemento e, durante o retorno da recursividade, os dados são ordenados. No Quicksort a divisão é feita durante o processo de particionamento, em que os valores menores que o pivô ficam numa sublista e os maiores em outra. Isso garante que o pivô já estará na sua posição final e que os elementos menores precedam os elementos maiores.

3. A heap é uma árvore binária completa (ou quase completa) que sempre armazena em sua raiz o elemento de maior valor entre todos os dados armazenados. Assim, no Heapsort, podemos nos aproveitar dessa propriedade para realizar a ordenação dos dados de modo a trocar a raiz com a última posição não ordenada do vetor.

GABARITO

UNIDADE V

1. A busca sequencial tem um funcionamento simples. Ela percorre toda uma estrutura em busca de um argumento dado até que o valor seja encontrado ou o arquivo termine. Apesar da facilidade de implementação, seu custo computacional é elevado.

2. O método de busca sequencial indexada permite que a busca seja interrompida prevendo se o argumento está ou não armazenado na tabela sem precisar percorrê-la até o final.

3. A única forma de aplicar a busca binária é em tabelas ordenadas ou em índices de valores indexados. A estratégia pressupõe que o valor está na metade superior ou inferior do arquivo, o que não seria possível se os dados não estiverem ordenados. A partir daí ela vai dividindo a região de busca, fazendo com que o tempo computacional seja reduzido.

4. Não é possível implementar a busca binária em listas dinâmicas. A técnica consiste em dividir a região de pesquisa sempre pela metade, e como os dados não estão armazenados de forma sequencial não é possível pular para a posição desejada durante a execução do algoritmo.

5. A busca por interpolação é parecida com a busca binária. A diferença está na forma de dividir a região de busca. Enquanto a busca binária divide o vetor pela metade, a busca por interpolação divide a região usando uma fórmula matemática mais complexa. Essa técnica só funciona se os dados no arquivo estiverem distribuídos de maneira uniforme.

6. A busca binária precisa dividir a região de busca pela metade, pois ela consegue prever se o valor está acima ou abaixo da atual posição no arquivo. Essa navegação se dá de forma simples em uma estrutura baseada em vetor, mas não é possível em listas encadeadas.

   Com a árvore binária de busca a pesquisa é feita de forma parecida, mas diferente. A pesquisa se inicia na raiz da árvore e segue pelo seu filho esquerda ou direita. Essa informação pode estar armazenada tanto num vetor como numa estrutura alocada dinamicamente, posto que a partir da raiz é possível percorrer a árvore inteira seguindo os ponteiros armazenados nos seus filhos.

   A busca em árvore binária também divide a região de busca, mas não exatamente pela metade. Como sabemos que todos os valores da subárvore esquerda são menores do que o valor da raiz, ou maiores se na árvore direita, é possível ir eliminando toda uma parte da árvore durante o processo, o que é muito eficiente em grandes massas de dados.

CONCLUSÃO

Chegamos novamente no final de mais um árduo trabalho, mas que sem dúvida trará frutos que compensarão todo o esforço dispensado por meio da pesquisa bibliográfica, codificação e teste de algoritmos e na redação dos conceitos teóricos explicados da forma mais acessível possível, sem deixar de apresentar a notação formal quando necessário.

Vimos a estrutura de Arvore Binária, sem dúvida a minha preferida (além dos grafos, se bem que toda árvore binária é um grafo). Estrutura essa muito utilizada na construção de sistemas operacionais e de bancos de dados, por ser uma estrutura que permite armazenamento e busca de dados eficientes.

Em seguida, estudamos diversos algoritmos de ordenação. Em sua maioria são algoritmos bastante antigos. E mesmo assim são técnicas utilizadas até hoje e serão utilizadas por muito mais tempo. Existem vários outros algoritmos de ordenação além desses, porém os apresentados aqui são os mais conhecidos.

Aprendemos alguns dos principais métodos de buscas. Desde o mais simples, como a busca sequencial, passando por buscas mais complexas, como a surpreendente busca binária e a apaixonante tabela *hash*.

Entretanto o estudo não para por aí. Agora depende de você, existem muitas outras técnicas e algoritmos na literatura para o seu deleite. Alguns consomem menos memória, outros demandam menos processamento. Você já tem a faca na mão, agora basta procurar o queijo certo e decidir como cortá-lo.

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES