CIÊNCIA DA COMPUTAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 9

CQA - COMISSÃO DE QUALIFICAÇÃO E AVALIAÇÃO

CIÊNCIA DA COMPUTAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 9

Christiane Mazur Doi

Doutora em Engenharia Metalúrgica e de Materiais, Mestra em Ciências - Tecnologia Nuclear, Especialista em Língua Portuguesa e Literatura, Engenheira Química e Licenciada em Matemática, com Aperfeiçoamento em Estatística. Professora titular da Universidade Paulista.

Tiago Guglielmeti Correale

Doutor em Engenharia Elétrica, Mestre em Engenharia Elétrica e Engenheiro Elétrico (ênfase em Telecomunicações). Professor titular da Universidade Paulista.

Questão 1

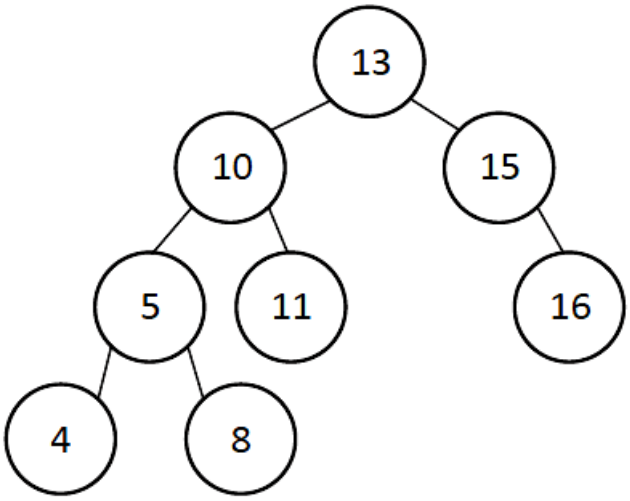
Questão 1.[1]

Leia o texto a seguir.

Uma árvore AVL é um tipo de árvore binária balanceada na qual a diferença entre as alturas de suas subárvores da esquerda e da direita não pode ser maior do que 1 para qualquer nó. Após a inserção de um nó em uma AVL, a raiz da subárvore de nível mais baixo no qual o novo nó foi inserido é marcada. Se a altura de seus filhos diferir em mais de uma unidade, é realizada uma rotação simples ou uma rotação dupla para igualar suas alturas.

LAFORE, R. *Data strutures & algorithms in Java*. Indianapolis: Sams Publishing, 2003 (com adaptações).

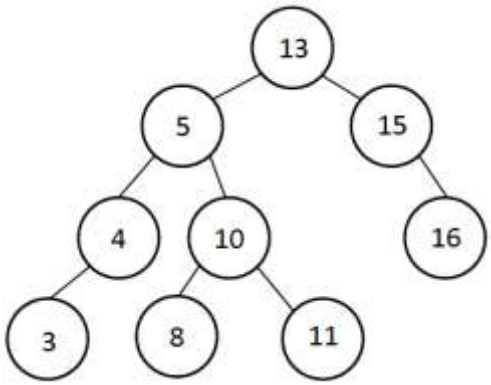
A seguir, é apresentado um exemplo de árvore AVL.



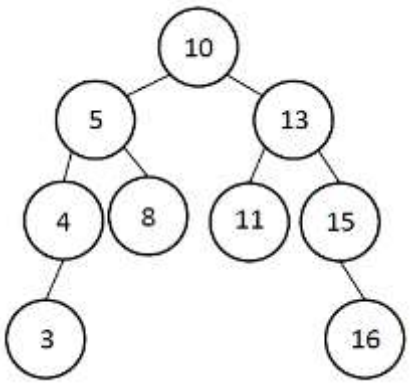
Pelo exposto no texto acima, após a inserção de um nó com valor 3 na árvore AVL exemplificada, ela ficará com a seguinte configuração:

[1]Questão 9 – Enade 2017.

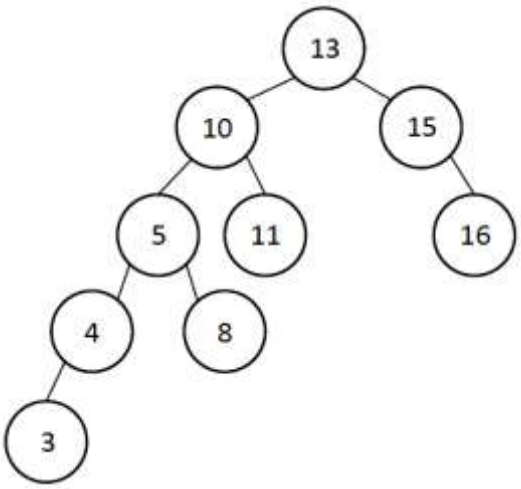
A.



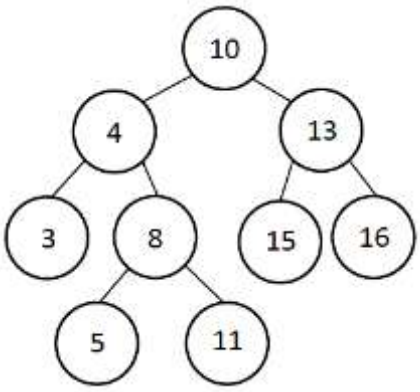
D.



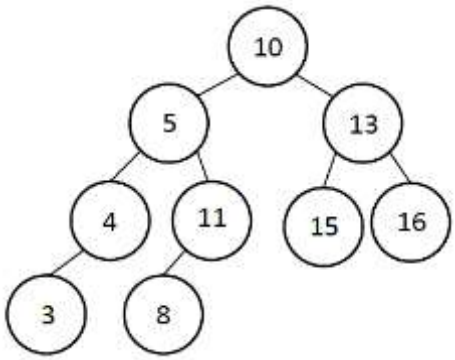
B.



E.



C.



1. Introdução teórica

Estrutura de dados: árvores

Na computação, frequentemente é necessário estruturarmos informações de forma hierárquica, ou seja, seguindo uma estrutura de níveis, de forma similar ao que vemos no organograma de uma organização, como o ilustra a figura 1. Nela, apresentamos uma amostra de um organograma de uma empresa hipotética.

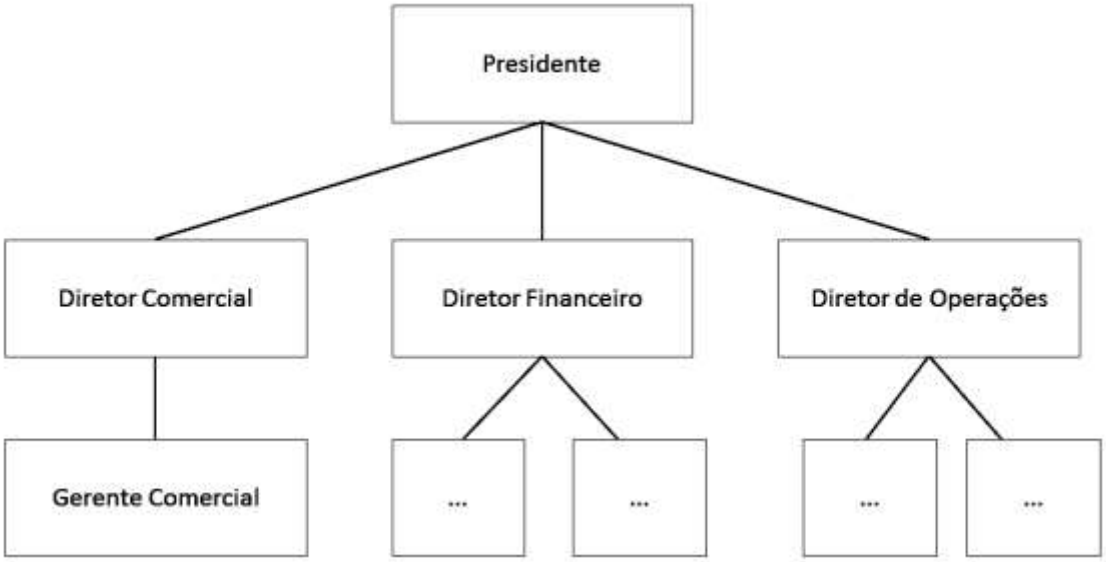


Figura 1. Exemplo de uma árvore: organograma de uma empresa hipotética.

A organização hierárquica da informação é a ideia básica para a estrutura de dados "árvore". Do ponto de vista teórico e formal, definimos uma árvore como um "grafo conectado, acíclico e não direcionado" (CORMEN et al., 2012). Uma das propriedades das árvores é a seguinte: há apenas um único caminho entre dois vértices.

Outra forma de definição de árvore é feita de forma recursiva: uma árvore é composta de um nó especial, chamado de nó raiz, e de um conjunto de subárvores ligadas a esse nó (CELES, CERQUEIRA e RANGEL, 2004). Esses nós são chamados de filhos, e o nó raiz é chamado de pai. Tal estrutura vai se ramificando, com os nós filhos ligando-se a subárvores e formando a estrutura hierárquica. Nas árvores, os nós que não têm filhos são chamados de nós externos ou folhas, enquanto os demais nós são chamados de nós internos (CELES, CERQUEIRA e RANGEL, 2004).

O problema das definições anteriores é que elas permitem grande número de estruturas com características muito diferentes umas das outras, o que pode ser um problema do ponto de vista da construção de algoritmos. Frequentemente, é interessante colocarmos alguma forma de restrição na estruturação da informação a fim de ganharmos algumas propriedades interessantes na construção de algoritmos que utilizam essas estruturas de dados. As árvores binárias seguem exatamente essa abordagem.

Segundo Celes, Cerqueira e Rangel (2004), uma árvore binária é uma "árvore em que cada nó pode ter zero, um ou dois filhos". Vemos que, nessa definição, é imposta uma restrição no número de nós filhos que podem ser descendentes de um dado nó pai. Isso pode ser feito de forma recursiva: uma árvore binária pode ser uma árvore vazia, ou, ainda, outra subárvore.

Em termos de nomenclatura, conforme ilustrado na figura 2, cada nó de uma árvore binária pode ter duas subárvores:

- uma no lado esquerdo, chamada de subárvore da esquerda, ou sae;
- outra no lado direito, chamada de subárvore da direira, ou sad.

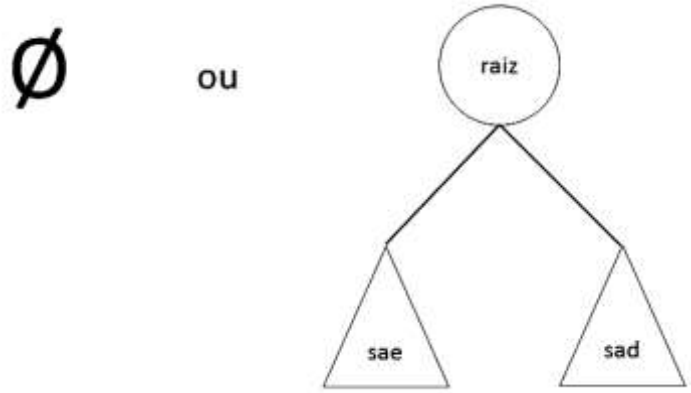


Figura 2. Possibilidades para uma árvore binária.
CELES, CERQUEIRA e RANGEL, 2004 (com adaptações).

Duas conceituações são importantes quando trabalhamos com árvores na computação: altura e profundidade. Dado um nó qualquer n, sua profundidade corresponde ao número de "ancestrais" que esse nó tem (GOODRICH, TAMASSIA e MOUNT, 2011). Por exemplo, observe, na figura 3, o nó F. Esse nó é filho do nó B, que é filho do nó A, que é o nó raiz da árvore. Dessa forma, o nó F tem dois ancestrais, e, portanto, sua profundidade é 2. Vale notar que a profundidade do nó raiz é 0 (zero), já que ele não tem nenhum nó ancestral.

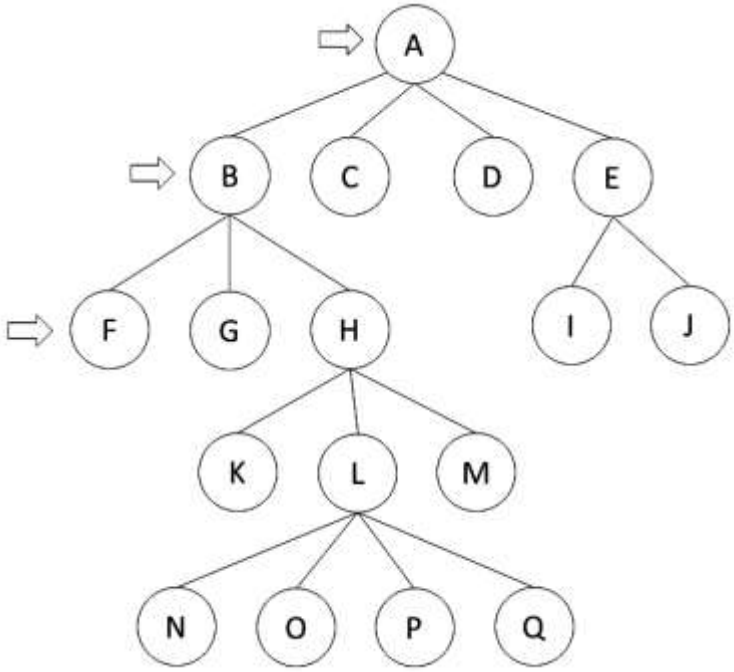


Figura 3. Exemplo de uma árvore.
GOODRICH, TAMASSIA e MOUNT, 2011 (com adaptações).

A altura de um nó é definida de forma recursiva. Para um nó folha, que está embaixo da árvore, a altura é definida como 0 (zero). Para os demais nós (que não são nós folhas), a altura é definida como 1 mais a altura do nó filho que tenha a maior altura. Assim, o nó N da figura 3 tem altura 0. Já o nó L tem altura 1, visto que todos os seus nós filhos (N, O, P e Q) têm altura 0. A altura de uma árvore é definida como a altura do nó raiz. Por exemplo, a altura da árvore na figura 3 é 4 (GOODRICH, TAMASSIA e MOUNT, 2011).

Estruturas de dados como árvores binárias podem ser utilizadas para organizar a informação de maneira que a busca de dados seja eficiente. Para compreendermos como isso é possível, vamos imaginar um caso de uma árvore binária que esteja completa, ou seja, em que todos os seus níveis tenham o máximo número de nós filhos, como ilustra a figura 4.

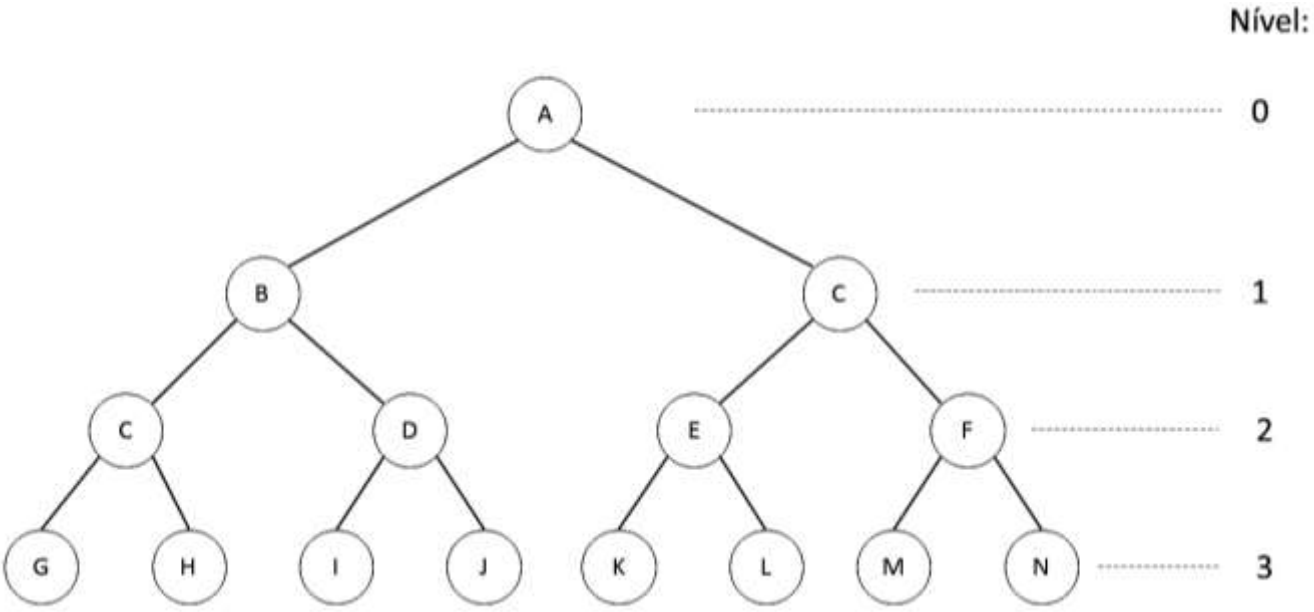


Figura 4. Uma árvore binária cheia.
CELES, CERQUEIRA e RANGEL, 2004 (com adaptações).

Vamos verificar, na figura 4, o número de nós para cada nível da árvore binária:

- para o nível 0, temos apenas 1 nó e podemos expressar isso como $2^0=1$;
- para o nível 1, temos 2 nós e podemos expressar isso como $2^1=2$;
- para o nível 2, temos 4 nós e podemos expressar isso como $2^2=4$;
- para o nível 3, temos 8 nós e podemos expressar isso como $2^3=8$.

De modo geral, dizemos que o número de nós para o nível n é dado por 2^n. Observamos, também, que o nível 3 tem um nó a mais do que o número de nós dado pela soma dos níveis anteriores, que, no caso, é 7: o nível zero tem 1 nó, o nível um tem 2 nós, o nível dois tem 4 nós, e, assim, o total de nós é 1+2+4=7. O nível três tem 8 nós, 1 a mais

do que o nível sete. Com base nesse resultado, podemos calcular o número total de nós de uma árvore binária cheia (ou completa), de altura h, como 2^{h+1}-1.

Na figura 4, a árvore tem altura 3, e o número total de nós é 15:

$$2^{h+1}-1=2^{3+1}-1=2^4-1=15$$

Assim, somos capazes de relacionar o número total de nós n com a altura h da árvore binária cheia:

$$n=2^{h+1}-1$$

Alternativamente, podemos querer obter a relação inversa: a altura h em função do total de nós. Para isso, devemos aplicar o logaritmo na base 2 em ambos os lados da relação $n=2^{h+1}-1$:

$$n=2^{h+1}-1$$

$$n+1=2^{h+1}$$

$$\log_2(n+1)=\log_2(2^{h+1})$$

$$\log_2(n+1)=h+1$$

$$h=\log_2(n+1)-1$$

A última equação mostra que o número de nós, para uma árvore binária completa, cresce como O(log n), utilizando a notação do O-grande. Lembremos que a árvore binária está cheia, ou seja, com o número máximo possível de nós. Portanto, essa relação diz que uma árvore binária de altura h deve ter altura mínima de O(log n), já que esse corresponde ao caso de "máxima compactação" (CELES, CERQUEIRA e RANGEL, 2004).

Tal resultado é importante, pois significa que, se conseguirmos organizar a informação de forma conveniente em uma árvore binária, podemos fazer uma busca de forma eficiente. Em um algoritmo de busca, se realizarmos apenas uma comparação por nível da árvore, o número de comparações vai ser da mesma ordem da altura da h árvore e, portanto, também vai ser da ordem de O(log n).

Isso nos leva à definição de árvore binária de busca. Em uma árvore binária de busca, a informação é organizada de modo que o valor do nó pai seja superior ao valor de qualquer nó da subárvore da esquerda e inferior ao valor de qualquer nó da subárvore da direita (CELES, CERQUEIRA e RANGEL, 2004). Essa propriedade acelera a velocidade com que a

informação pode ser buscada na árvore e é particularmente útil para algoritmos em que uma informação deva ser encontrada de maneira rápida.

Com base nas definições de altura de um nó e de árvores binárias, podemos definir balanceamento. Suponha um nó de uma árvore binária que não seja um nó folha. Imagine que esse nó tenha dois nós filhos. Se o valor absoluto da diferença entre a altura dos seus dois nós filhos for no máximo igual a 1 (em módulo), dizemos que esse nó está balanceado (GOODRICH, TAMASSIA e MOUNT, 2011). Uma árvore de busca binária em que a propriedade de balanceamento vale para todos os nós é chamada de árvore AVL, em homenagem aos seus criadores, Georgy Adelson-Velsky e Yevgeniy Landis.

Contudo, um problema ocorre ao trabalharmos com árvores binárias de busca: a inserção e a remoção de nós podem desbalancear a árvore. Por exemplo, vamos observar as figuras 5 e 6, nas quais é mostrado o efeito da inserção de um nó em uma árvore binária, com o valor 186. A figura 5 retrata a árvore antes da inserção do nó, e a figura 6, após a inserção. Inicialmente, o nó que contém o valor 172 apresenta altura 3 e está balanceado, pois o nó com o valor 188 apresenta altura 1 e o nó com o valor 164 apresenta altura 0, e, portanto, o módulo da diferença de altura é apenas 1. Contudo, ao inserirmos o valor de 186, a árvore torna-se desbalanceada: o nó com o valor de 188 passa a ter altura 2, enquanto o nó com o valor 164 continua tendo altura 0, o que leva a uma diferença de 2 entre as alturas dos nós filhos do nó com o valor de 172 (figura 6).

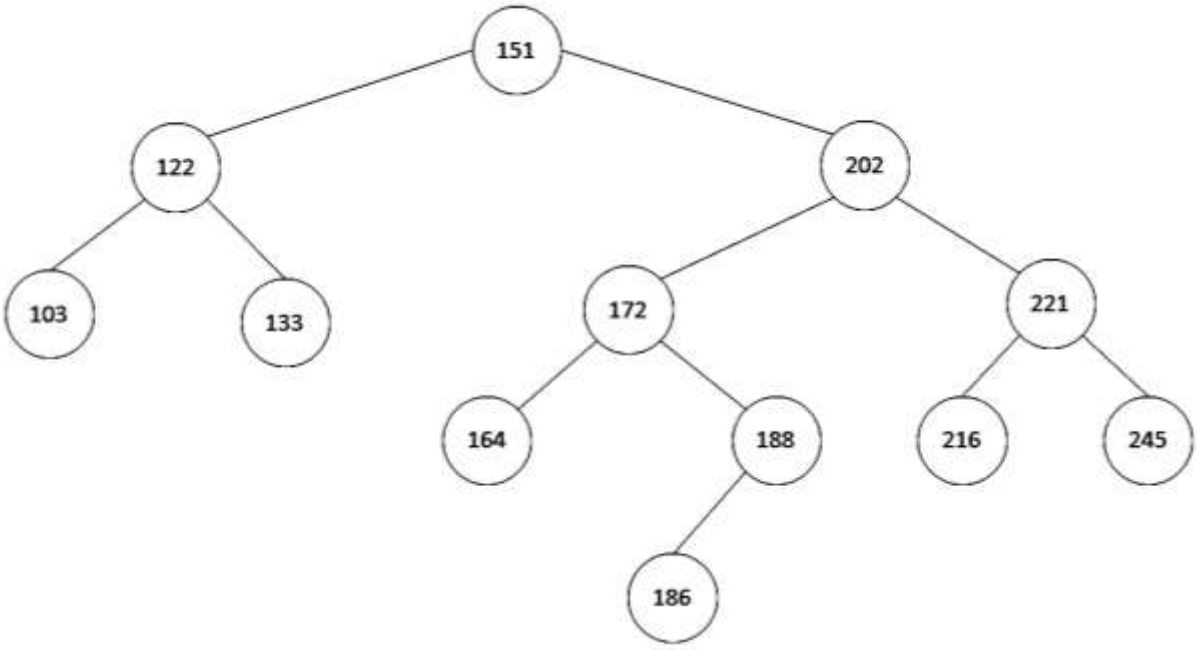


Figura 5. Exemplo de árvore binária antes de uma operação de inserção.

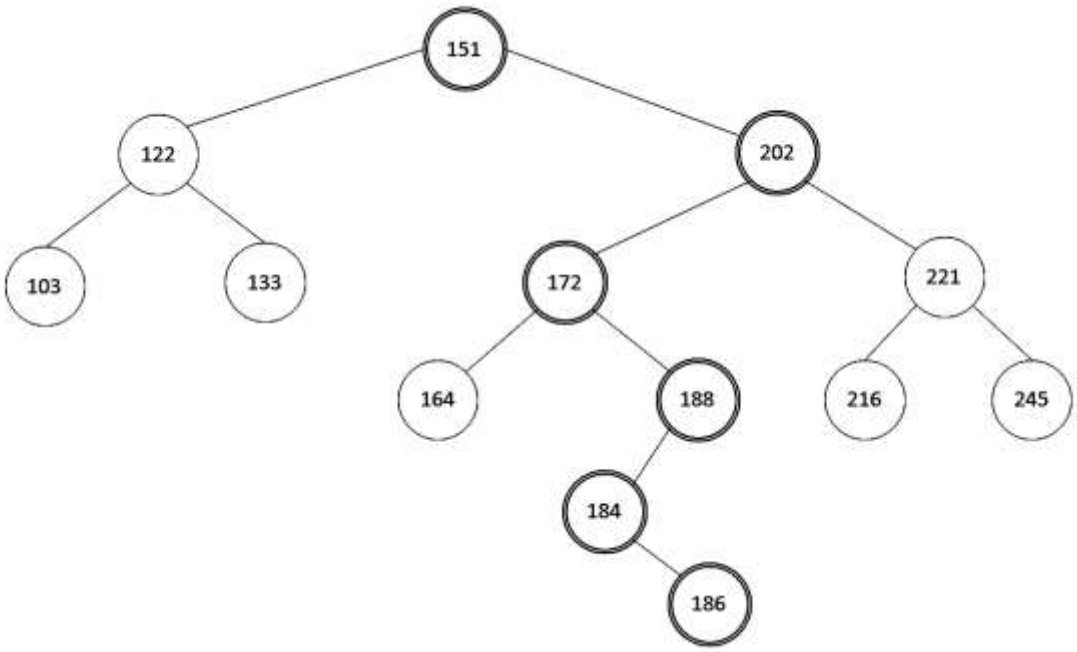


Figura 6. Exemplo anterior após inserção do valor 186.

Para mantermos a propriedade de balanceamento, é necessário modificarmos a árvore. No entanto, devemos manter a característica de uma árvore binária de busca, na qual o nó filho da direita tem valor maior do que o valor do nó pai, e o nó filho da esquerda tem valor menor do que o valor do nó pai. Para conseguirmos isso, utilizamos as operações de rotação simples e de rotação dupla.

Nas figuras 7 e 8, damos um exemplo de rotação à direita. Vemos que a árvore da figura 7 estava originalmente desbalanceada: o nó com o valor 45 tem altura 2, e o nó com o valor 68 tem altura 0, de forma que a diferença entre os dois é superior a 1, e, portanto, o nó com o valor de 63 está desbalanceado.

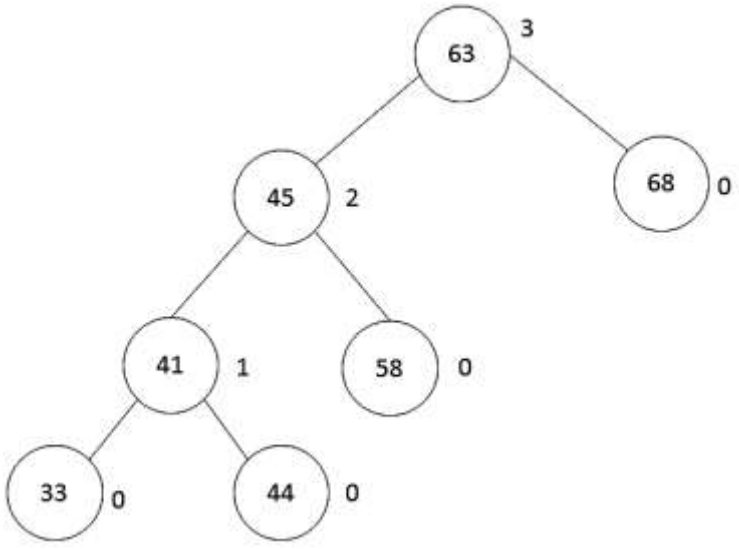


Figura 7. Exemplo de árvore antes da rotação.

Para balancearmos o nó, fizemos uma rotação à direita e obtivemos a árvore apresentada na figura 8. Vemos, agora, que todos os nós estão balanceados, e, portanto, a árvore está balanceada.

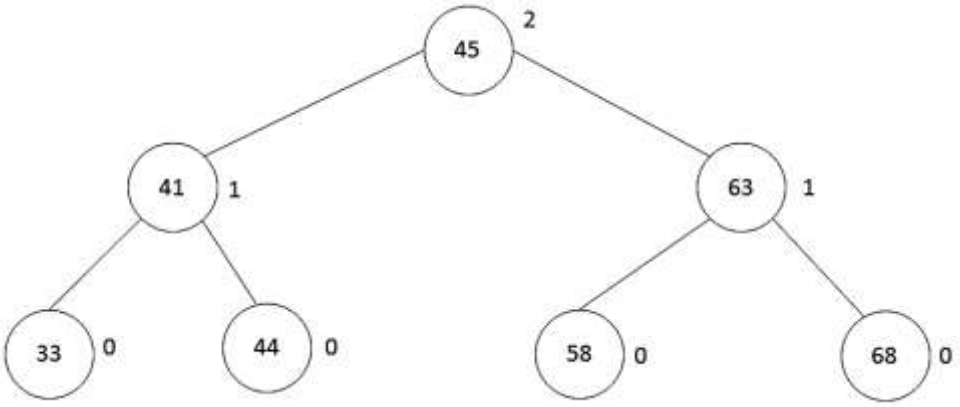


Figura 8. Resultado após a rotação à direita.

2. Análise das alternativas

Antes de solucionarmos o problema, devemos lembrar que o valor 3 deve ser inserido na árvore, a fim de obtermos a árvore da figura 9 (que corresponde à árvore da alternativa B). Essa árvore encontra-se desbalanceada no nó com o valor 10 e no nó raiz.

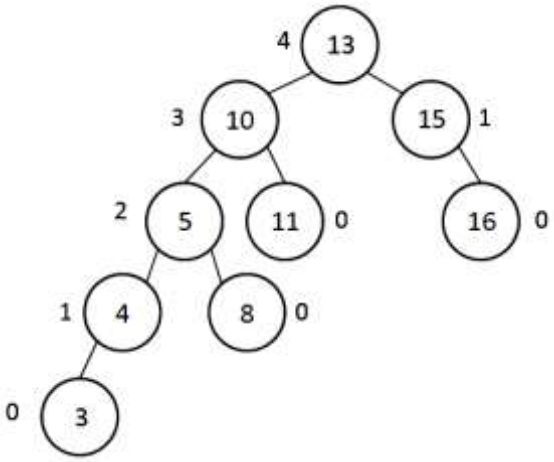


Figura 9. Árvore após inserção do valor 3. Ao lado das circunferências, anotamos as alturas dos nós. Enade, 2017.

A – Alternativa correta.

JUSTIFICATIVA. Observamos que a árvore do enunciado está desbalanceada no nó que contém o valor 10. Logo, devemos fazer uma rotação para balancearmos a árvore. Executando uma rotação simples à direita, obtemos a árvore ilustrada na alternativa.

B – Alternativa incorreta.

JUSTIFICATIVA. Essa árvore corresponde simplesmente à árvore após a inserção do valor 3, sem nenhum procedimento de balanceamento. Portanto, não se trata de uma árvore AVL.

C e E – Alternativas incorretas.

JUSTIFICATIVA. As árvores estão incorretas, pois o valor 15 encontra-se do lado errado do nó com o valor 13 (como ele é "maior", deveria estar no lado direito). Nesses casos, as árvores mostradas não são árvores binárias de busca e, muito menos, árvores do tipo AVL.

D – Alternativa incorreta.

JUSTIFICATIVA. Ainda que a árvore obtida esteja balanceada, a rotação não foi feita no nó correto, uma vez que ela deve ser executada no primeiro nó em que ocorre o desbalanceamento: no caso, trata-se do nó que contém o valor 10 (considerando como referência a figura 9).

3. Indicações bibliográficas

- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução à estrutura de dados com técnicas de programação em C.* Rio de Janeiro: Elsevier, 2004.
- CORMEN, T. et al. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Campus-Elsevier, 2012.
- GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. *Data structures & algorithms in C++*. 2. ed. Hoboken: John Wiley & Sons, 2011.
- SKIENA, S. S. *The algorithm design manual.* 2 ed. Londres: Springer-Verlag, 2008.

Questão 2

Questão 2.[2]

O algoritmo a seguir recebe um vetor v de números inteiros e rearranja esse vetor de tal forma que seus elementos, ao final, estejam ordenados de forma crescente.

```
01.     void ordena(int *v, int n)
02.     {
03.         int i, j, chave;
04.         for(i = 1; i < n; i++)
05.         {
06.             chave = v[i];
07.             j = i - 1;
08.             while(j >= 0 && v[j] < chave)
09.             {
10.                 v[j-1] = v[j];
11.                 j = j - 1;
12.             }
13.             v[j+1] = chave;
14.         }
15.     }
```

Considerando que nesse algoritmo há erros de lógica que devem ser corrigidos para que os elementos sejam ordenados de forma crescente, assinale a opção correta no que se refere às correções adequadas.

A. A linha 04 deve ser corrigida da seguinte forma: `for(i=1;i<n-1;i++);` e a linha 13, do seguinte modo: `v[j-1]=chave;`.

B. A linha 04 deve ser corrigida da seguinte forma: `for(i=1;i<n-1;i++);` e a linha 07, do seguinte modo: `j = j+1;`.

C. A linha 07 deve ser corrigida da seguinte forma: `j = i +1;` e a linha 08, do seguinte modo: `while(j>=0 && v[j]>chave)`.

D. A linha 08 deve ser corrigida da seguinte forma: `while(j>=0 && v[j]>chave);` e a linha 10, do seguinte modo: `v[j+1]=v[j];`.

E. A linha 10 deve ser corrigida da seguinte forma: `v[j+1]=v[j];` e a linha 13, do seguinte modo: `v[j-1]=chave;`.

[2]Questão 18 – Enade 2017.

1. Introdução teórica

O algoritmo de ordenação *insertion-sort*

O algoritmo *insertion sort* é uma das formas mais simples de algoritmo de ordenação. A sua ideia básica é similar à forma como ordenamos cartas de baralho na mão.

Suponha, inicialmente, um vetor de números inteiros, desordenados. Vamos ordenar esse vetor da seguinte forma: vamos dividi-lo em duas partes, escolhendo um elemento desse vetor como o elemento "divisor". Imagine que tiremos uma "foto" desse algoritmo em execução, antes do seu término. Nesse momento, o algoritmo funciona de modo que todos os elementos que vêm "antes" da chave e à sua esquerda estejam ordenados, mas os elementos à direita podem não estar.

A cada iteração do algoritmo, é necessário compararmos todos os elementos anteriores (à esquerda) da chave escolhida com o seu valor. Por exemplo, na figura 1, a chave é o elemento que contém o valor 4 e está marcada com uma flecha em cima. Inicialmente, comparamos a chave com o valor imediatamente anterior, no caso, o número 6. Como esse número é maior do que 4, ele deve ser "empurrado" para frente, e, assim, assume o valor que atualmente é ocupado pelo número 4. Por isso, antes das comparações, devemos fazer uma cópia do valor da chave em outra variável.

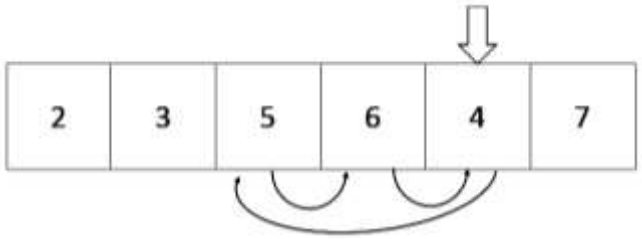


Figura 1. Situação do vetor durante o processo de ordenação.
Cormen et al., 2012 (com adaptações).

Fazemos, novamente, a comparação do valor da chave (que ainda é o número 4) com o valor anterior ao número 6, no caso, o número 5. Esse número também é maior do que a chave, e, portanto, devemos "empurrá-lo" para frente, de forma que ele ocupe o valor que anteriormente era ocupado pelo número 6.

Finalmente, encontramos o número 3, que é menor do que a chave (4). Agora, acabamos de determinar o ponto no qual devemos validar a chave: na posição do número 5 (observe que, temporariamente, ficamos com o valor 5 repetido no vetor, pois temos a posição antiga e o valor empurrado na iteração anterior). Se assumirmos que os valores

anteriores ao número 3 (no caso, apenas o número 2, mas poderíamos ter mais números) estão todos em ordem, acabamos de ordenar a parte esquerda do vetor.

Devemos identificar o procedimento para trabalhar com as chaves. A chave começa como o segundo elemento do vetor, de modo que, na primeira iteração do algoritmo, existe apenas um elemento anterior à chave (apenas um elemento à sua esquerda). Obviamente, por termos apenas um elemento, essa parte já está "ordenada". O máximo que pode acontecer é precisarmos inverter a posição desse elemento com a chave, se ela tiver um valor inferior.

Depois, repetimos o procedimento, mas devemos escolher uma nova chave: o terceiro elemento do vetor. Procedemos conforme descrito anteriormente, comparando a chave com os elementos à sua esquerda, até acharmos o ponto no qual seu valor deveria estar inserido (que pode ser, inclusive, a sua posição atual, se a chave for maior do que todos os outros valores). Isso vai sendo repetido, até chegarmos ao último elemento do vetor, quando finalmente terminamos a ordenação.

É necessário observarmos que esse algoritmo requer uma quantidade constante de memória para o processo de ordenação, já que os valores vão sendo trocados no próprio vetor de entrada e precisamos apenas de algumas variáveis para armazenar os valores da chave e de alguns outros índices auxiliares. Contudo, a performance de execução desse algoritmo é tipicamente ruim: $O(n^2)$ no caso médio e $O(n^2)$ no pior caso. Caso o vetor já se apresente ordenado ou praticamente ordenado, sua performance é linear, O(n). No entanto, tal situação é rara.

A seguir, apresentamos a versão corrigida do algoritmo *insertion-sort* do enunciado, implementado na linguagem C.

```
01.     void ordena(int *v, int n)
02.     {
03.         int i, j, chave;
04.         for(i = 1; i < n; i++)
05.         {
06.             chave = v[i];
07.             j = i - 1;
08.             while(j >= 0 && v[j] > chave)
09.             {
10.                 v[j+1] = v[j];
```

```
11.                 j = j - 1;
12.             }
13.             v[j+1] = chave;
14.         }
15.     }
```

Observamos que foi criada uma função do tipo *void*, ou seja, que não retorna nenhum valor (há ausência do comando *return* na função). O vetor de inteiros é passado como um ponteiro para números inteiros (a variável v), juntamente com o número de elementos do vetor, n, que é um número inteiro. Podemos verificar que essa função é capaz de ordenar vetores de qualquer tamanho e faz uso de apenas três variáveis inteiras adicionais: os índices i e j e a variável que contém a chave.

2. Análise das alternativas

A – Alternativa incorreta.
JUSTIFICATIVA. Os limites do laço "for" no enunciado estão corretos e não devem ser modificados. Da forma proposta na alternativa, o último elemento do vetor não seria ordenado. A linha 13 também está correta e não deve ser modificada.

B – Alternativa incorreta.
JUSTIFICATIVA. O laço "for" da linha 4 está correto, uma vez que ele deve percorrer o vetor a partir do segundo elemento. A linha 7 também está correta, já que o laço interno do algoritmo deve percorrer o vetor de "trás para frente", começando a partir do elemento imediatamente anterior à chave. Assim, essas linhas não devem ser modificadas.

C – Alternativa incorreta.
JUSTIFICATIVA. A modificação da linha 8 está correta e deve ser feita, mas a linha 7 não deve ser modificada.

D – Alternativa correta.
JUSTIFICATIVA. Para que os números sejam ordenados em ordem crescente, a linha 8 deve ser modificada para while(j >= 0 && v[j] > chave), e a linha 10, que corresponde à cópia dos elementos para a frente no vetor de entrada, deve ser v[j+1]=v[j];.

E – Alternativa incorreta.

JUSTIFICATIVA. A correção da linha 10 deve ser feita, mas a linha 13 não deve ser modificada.

3. Indicações bibliográficas

- CORMEN, T. et al. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Campus-Elsevier, 2012.
- GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. *Data structures & algorithms in C++*. 2. ed. Hoboken: John Wiley & Sons, 2011.

Questões 3, 4 e 5

Questão 3.[3]

A sequência de Fibonacci é uma sequência de números inteiros que começa em 1, a que se segue 1, e na qual cada elemento subsequente é a soma dos dois elementos anteriores. A função fib a seguir calcula o n-ésimo elemento da sequência de Fibonacci:

```
unsigned int fib (unsigned int n)
{
     if( n < 2 )
          return 1;
     return fib(n-2) + fib(n-1);
}
```

Considerando a implementação acima, avalie as afirmativas.

I. A complexidade no tempo da função fib é exponencial no valor de n.

II. A complexidade de espaço da função fib é exponencial no valor de n.

III. É possível implementar uma versão iterativa da função fib com complexidade de tempo linear no valor de n e complexidade de espaço constante.

É correto o que se afirma em

A. I, apenas.

B. II, apenas.

C. I e III, apenas.

D. II e III, apenas.

E. I, II e III.

Questão 4.[4]

Considere a função recursiva F a seguir, que, em sua execução, chama a função G.

```
1   void F(int n) {
2      if(n > 0) {
3         for(int i = 0; i < n; i++) {
4            G(i);
5         }
6         F(n/2);
7      }
8   }
```

Com base nos conceitos da teoria da complexidade, avalie as afirmativas.

[3]Questão 25 – Enade 2017.
[4]Questão 33 – Enade 2017.

I. A equação de recorrência que define a complexidade da função F é a mesma do algoritmo clássico de ordenação *mergesort*.
II. O número de chamadas recursivas da função F é Θ(log n).
III. O número de vezes que a função G da linha 4 é chamada é O(n log n).

É correto o que se afirma em

A. I, apenas.

B. II, apenas.

C. I e III, apenas.

D. II e III, apenas.

E. I, II e III.

Questão 5.[5]

Listas lineares armazenam uma coleção de elementos. A seguir, é apresentada a declaração de uma lista simplesmente encadeada.

```
struct ListaEncadeada{
     int dado;
     struct ListaEncadeada *proximo;
};
```

Para imprimir os seus elementos da cauda para a cabeça (do final para o início) de forma eficiente, um algoritmo pode ser escrito da seguinte forma:

```
void mostrar (struct ListaEncadeada *lista){
     if( lista != NULL ) {
          mostrar(lista->proximo);
          printf("%d", lista->dado);
     }
}
```

Com base no algoritmo apresentado, faça o que se pede nos itens a seguir.

a) Apresente a classe de complexidade do algoritmo, usando a notação θ.
b) Considerando que já existe implementada uma estrutura de dados do tipo pilha de inteiros - com as operações de empilhar, desempilhar e verificar pilha vazia - reescreva o algoritmo de forma não-recursiva, mantendo a mesma complexidade. Seu algoritmo pode ser escrito em português estruturado ou em alguma linguagem de programação, como C, Java ou Pascal.

[5]Questão Discursiva 03 – Enade 2017.

1. Introdução teórica

1.1. Notação assintótica e crescimento dos algoritmos

A notação assintótica é utilizada em Ciência da Computação para classificar algoritmos do ponto de vista da ordem de crescimento do tempo de execução (ou do consumo de memória) para grandes entradas. É importante não confundirmos a notação assintótica com o conceito de assíntota na Matemática, que são diferentes.

Para visualizarmos a ideia da notação assintótica, vamos, inicialmente, pensar de uma forma puramente matemática. Suponha a função f(x)=5x+10. Conforme aumentamos o valor de x, os valores de f(x) crescem de forma linear, pois a equação de f(x) descreve uma reta. Vamos comparar os resultados de f(x) para valores pequenos de x e para valores grandes de n.

Tomemos um valor "pequeno" para x, como, por exemplo, x = 1. Nesse caso, ficamos com f(1)=15, pois f(1)=5x1+10=5+10=15. Observe que o fator 5 (coeficiente angular) é menor do que o termo constante 10 (coeficiente linear). Agora, vamos considerar um valor "grande" para x, como, por exemplo, x = 1000. Nesse caso, ficamos com f(1000)=5010, pois f(1000)=5x1000+10=5000+10=5010.

O termo 5000 é 500 vezes maior do que o valor constante de 10. Observamos que, conforme aumentamos o valor de x, o termo 5x, que contém o coeficiente angular 5, torna-se mais "relevante" no valor calculado de f(x) do que o termo constante, dado pelo coeficiente linear 10.

O mesmo raciocínio pode ser utilizado para outras funções. Vejamos, por exemplo, a função do segundo grau dada por $f(x)=2x^2+1000x+10$. Para valores "pequenos" de x, o termo $2x^2$ pode não ser dominante, mas, conforme os valores de x vão aumentando, o termo quadrático vai tornando-se cada vez mais relevante, pois varia mais rapidamente do que os demais termos. Para x=10000, obtemos:

$$f(10000)=2.(10000)^2+1000(10000)+10=210000010$$

Observe que $2.(10000)^2=2.10^8$ e que $1000(10000)=10^7$, de modo que, nessa situação específica, o termo quadrático é 20 vezes maior do que o termo linear, e muito maior do que o termo constante 10. De modo genérico, podemos dizer que o termo quadrático "domina" os demais termos para grandes valores de x.

A notação assintótica formaliza essa análise da velocidade de crescimento das funções. Por exemplo, de acordo com Cormen et al. (2012), segundo a notação do "teta maiúsculo" ou de "teta grande", no formato Θ(.), para uma função g(n), temos que Θ(g(n)) é um conjunto de funções tais que "Θ(g(n)) = {f(n): existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$}".

Vejamos: se tivermos uma afirmação como $f(n)=\Theta(n^2)$, estamos dizendo que a função f(n), a partir de algum ponto n_0, torna-se limitada inferiormente por $c_1 n^2$ e superiormente $c_2 n^2$, sendo c_1 e c_2 duas constantes. Intuitivamente, a função f(n) fica contida entre duas outras funções, $c_1 n^2$ e $c_2 n^2$, ambas quadráticas.

Adicionalmente, de acordo com Cormen et al. (2012), "f(n) deve ser assintoticamente não negativa, ou seja, a função f(n) deve ser não negativa para valores de n suficientemente grandes".

Outra possibilidade ocorre quando encontramos uma função do tipo c.g(n), que se torna sempre maior ou igual a f(n) a partir de um ponto n_0. Essa é a chamada notação do "O grande", dada por O(.). Formalmente, Cormen et al. (2012) definem "O(g(n)) = {f(n): existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq c.g(n)$ para todo $n \geq n_0$}".

A notação do O-grande é chamada de limite assintótico superior (CORMEN et al., 2012), uma vez que encontramos uma função que, a partir de certo valor de n_0, fica sempre maior do que a função g(n) ou igual à função g(n), lembrando que g(n) é a função que queremos classificar.

Também podemos encontrar o limite assintótico inferior, definido de forma similar ao visto anteriormente por Cormen et al. (2012). Segundo os autores, "Ω(g(n)) = {f(n): existem constantes positivas c e n_0 tais que $0 \leq c.g(n) \leq f(n)$ para todo $n \geq n_0$}". Nas figuras 1 e 2, ilustramos essas possibilidades.

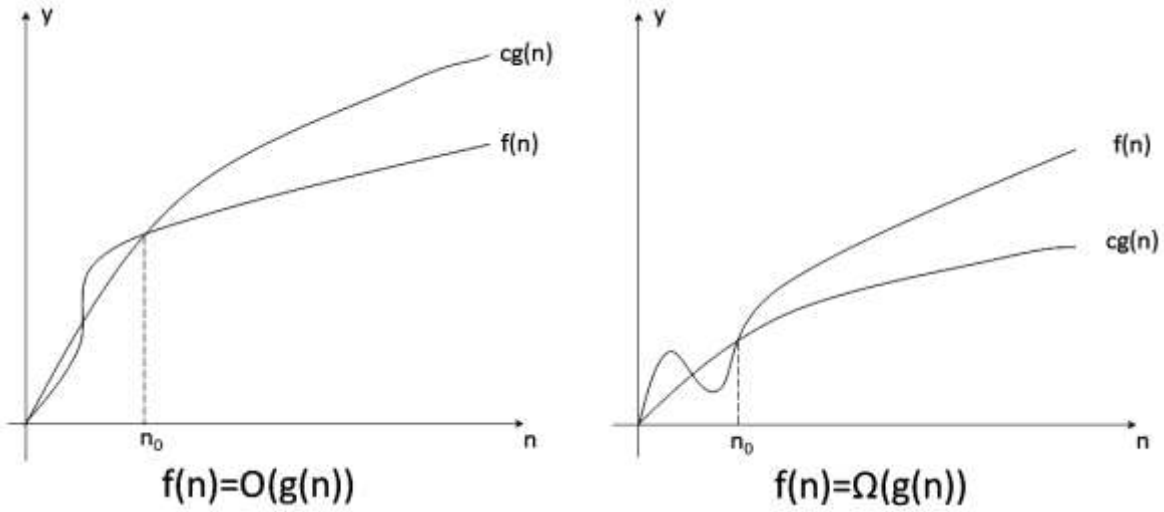


Figura 1. Interpretando graficamente a notação do O-grande e do Ω-grande.
Cormen et al., 2012 (com adaptações).

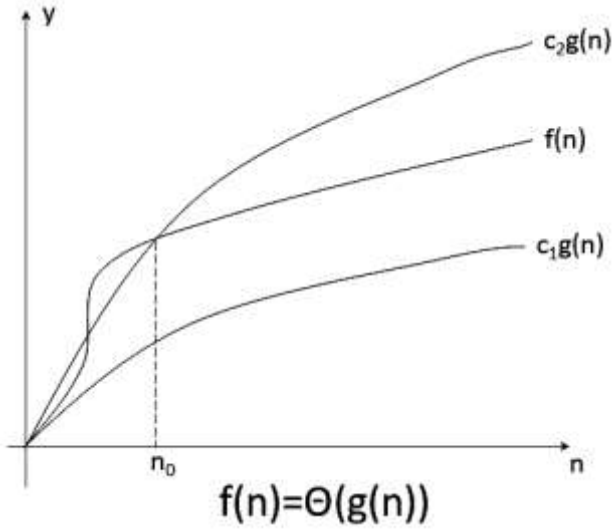


Figura 2. Interpretando graficamente a notação do Θ-grande.
Cormen et al., 2012 (com adaptações).

A notação assintótica é aplicada para termos uma noção da complexidade dos algoritmos. Ela pode ser empregada para avaliarmos:

- a complexidade temporal, ou seja, para medirmos a quantidade de tempo que um algoritmo leva para ser executado;
- a complexidade espacial ou seja, para quantificarmos a memória consumida pelo algoritmo ao longo do tempo.

1.2. Equações de recorrência e teorema mestre

A recursão é uma forma poderosa de resolver problemas na área da computação. Dizemos que uma função é recursiva quando ela "chama a si mesma". Obviamente, deve ser utilizado algum critério que interrompa o processo de recursão com a finalidade de

evitarmos uma recursão "infinita", na qual uma função continuamente chama a si mesma, sem nunca terminar a sua execução.

A análise do tempo de execução de um algoritmo que usa recursão é relativamente complexa e é feita, em geral, por uma "função de recorrência" associada ao algoritmo a ser estudado. Como exemplo, vamos supor algoritmo do tipo "dividir e conquistar", em que um problema maior é dividido em instâncias menores, diversas vezes e de forma recursiva, até que se chegue a um caso de base simples, para o qual a solução é imediata.

Nos algoritmos "dividir e conquistar", a equação de recorrência é dividida em duas partes:

- o primeiro termo tem complexidade constante e corresponde a uma solução simples e imediata;
- o segundo termo é relativo ao tempo para a divisão do problema em partes menores (que chamamos de D(n), sendo n o tamanho da entrada para o algoritmo) e ao tempo para combinar a solução dos problemas menores (que chamamos de C(n)).

Além disso, há o tempo total T(n), que é utilizado nos dois lados da equação de recursão (CORMEN et al., 2012).

O que explicamos pode ser sumarizado pela equação 1 a seguir, em que:

- c é uma constante relacionada ao tamanho do problema;
- a e b são duas constantes relacionadas, respectivamente, ao número de subproblemas e à diminuição do tamanho deles para cada chamada recursiva.

$$T(n) = \begin{cases} \Theta(1) \quad se \quad n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) \quad caso\ contrário \end{cases} \qquad (1)$$

O teorema pode ser utilizado para resolver equações de recorrência no formato mostrado na equação 2 a seguir (CORMEN et al., 2012).

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad (2)$$

Nesse caso, a e b devem ser constantes e satisfazer às condições a≥1 e b>1. O teorema afirma que existem três possíveis soluções para a complexidade assintótica descrita pela equação 2. Devemos, inicialmente, obter a complexidade assintótica de f(n) e, então, analisar se ela pode ser descrita por um dos casos do quadro 1 (CORMEN et al., 2012).

Quadro 1. Condições do teorema.

Caso	Condição	Complexidade de T(n)
1	$f(n) = O(n^{\log_b a - \varepsilon})$ para algum ε > 0	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
3	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algum ε > 0 e $af(n/b) \leq cf(n)$ para alguma constante c < 1 e valores de n suficientemente grandes.	$T(n) = \Theta(f(n))$

Construída com informações de Cormen et al., 2012.

2. Análise das alternativas e solução das questões

Questão 3.

I – Afirmativa correta.

JUSTIFICATIVA. Inicialmente, calculamos um limite superior para a implementação recursiva do algoritmo de Fibonacci. Para isso, desenhamos o processo de computação em uma árvore, como a ilustrada na figura 3. No primeiro nível da árvore (frequentemente chamado de nível zero), que corresponde ao tempo que se leva para o cálculo de fib(n), T(n) faz duas chamadas para fib(n-1) e f(n-2). Assim, o tempo levado para o segundo nível da árvore binária é k+k=2k. O nível inferior tem quatro nós, com tempo total de 4k. Sabemos que uma das propriedades das árvores binárias é que a sua altura h é menor do que t-1, em que t é o número de nós (GOODRICH, TAMASSIA e MOUNT, 2011).

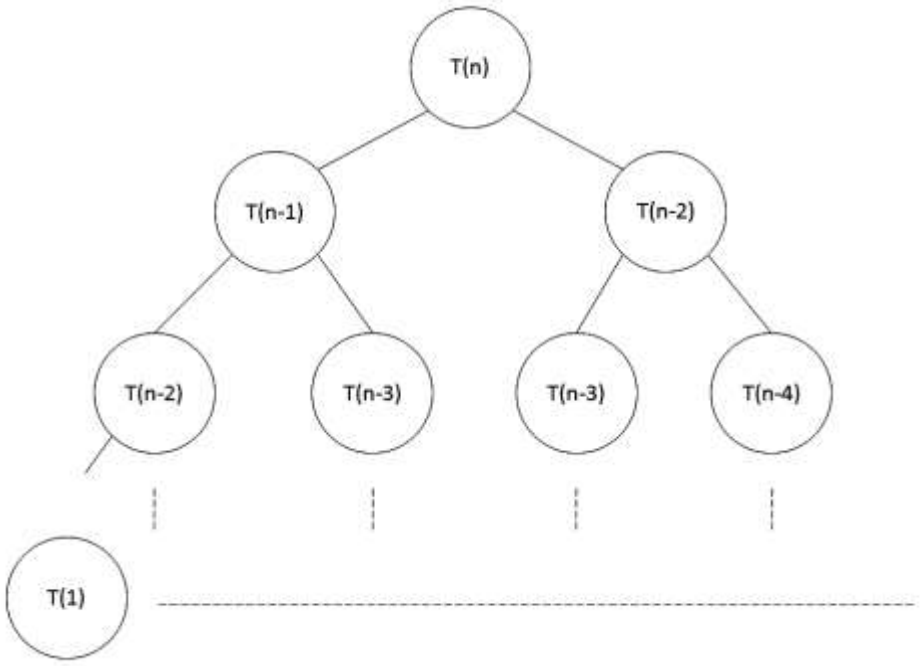


Figura 3. Árvore representando a implementação recursiva do algoritmo de Fibonacci.
Cormen et al., 2012 (com adaptações).

Para calcularmos o número de níveis, observamos, primeiramente, que o nó raiz pode ser escrito como T(n-0). No nível inferior (nível 1), o filho esquerdo é T(n-1). O filho esquerdo desse nó é T(n-2), que é o nível 2. Progredimos até T(n-(n-1))=T(n-n+1)=T(1), que é o nó mais inferior do lado esquerdo da figura 3. Assim, a árvore tem n-1 níveis. Dado um nível m, esse nível tem 2^m nós (considerando uma árvore cheia). Logo, o número total t de nós pode ser calculado como:

$$t=1 + 2 + 2^2+...+2^{n-1}$$

Suponha que a quantidade de tempo para a execução da função fib, sem contar as chamadas recursivas, seja constante e igual a k. Com cada nó levando tempo k para ser executado, sabemos que o tempo de execução total T deve ser:

$$T=k.1 + k.2 + k.2^2+...+k.2^{n-1}$$

Também sabemos que:

$$T=k.1 + k.2 + k.2^2+...+k.2^{n-1} \leq k.2^n$$

Assim, temos $T(n)=O(2^n)$, que é um tempo exponencial. É possível encontrarmos um limite melhor para o tempo de execução, do tipo $O(\Phi^n)$, sendo Φ a proporção áurea. Nesse caso, T(n) continua sendo exponencial. Desse modo, concluímos que a afirmativa I é correta.

II – Afirmativa incorreta.

JUSTIFICATIVA. A afirmativa II não é correta, uma vez que não há alocação de variáveis na implementação mostrada no enunciado. No entanto, isso deve ser visto com cautela. Na maioria das linguagens (não em todas), a chamada de uma função implica a alocação de um espaço na pilha de execução (chamada de *stack*), o que significa que funções recursivas podem causar transbordo de pilha (*stack overflow*) caso sejam chamadas muitas vezes.

III – Afirmativa correta.

JUSTIFICATIVA. Um exemplo de implementação com complexidade linear de tempo e constante de espaço é dado a seguir.

```
unsigned int fib( unsigned int n)
{
    int a, n1 =1, n2=0;
    for(a=0; a < n; a = n1 + n2)
    {
```

```
            printf(" %d ", a);
            n2 = n1;
            n1 = a;
        }
        return a;
    }
```

Observamos que existe apenas um laço "for", que é executado enquanto o número obtido da sequência seja menor do que o valor desejado de n (considerando a complexidade de tempo linear). Além disso, são utilizadas apenas duas das três variáveis para o cálculo (considerando a complexidade de espaço constante).
Desse modo, concluímos que a afirmativa III é correta.

Alternativa correta: C.

Questão 4.

Inicialmente, devemos escrever a relação de recorrência para o algoritmo do enunciado. Observamos que, para cada chamada da função F(n), a função G(i) é executada n vezes no laço da linha 3. A função F é, então, chamada novamente, de forma recursiva, agora com n dividido por dois, como verificamos na linha 6. Dessa forma, se considerarmos o tempo de execução T(n) do pior cenário do algoritmo, a chamada recursiva de F(n/2) deve levar tempo de execução T(n/2). Vemos que, nesse ponto, ainda não sabemos o valor de T(n), mas podemos escrever a relação de recorrência a seguir.

$$T(n) = \begin{cases} \Theta(1) \quad se \quad n = 0 \\ T\left(\frac{n}{2}\right) + \Theta(n) \quad se \quad n \geq 1 \end{cases}$$

A relação de recorrência para o algoritmo *mergesort* é (CORMEN et al., 2012):

$$T(n) = \begin{cases} \Theta(1) \quad se \quad n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) \quad se \quad n > 1 \end{cases}$$

Notamos que as relações são diferentes, uma vez que, no caso do *mergesort*, a função T(n/2) aparece multiplicada por dois, o que invalida a afirmativa I.

Para termos uma ideia de quantas vezes a função F vai ser chamada de forma recursiva, podemos ter em mente que, a cada nova chamada de F, a função é chamada de forma recursiva com n dividido por dois. Assim, vamos supor que a função F seja chamada k vezes. Na primeira vez, F é chamada com F(n/2). Na segunda vez, com F(n/4). Na terceira vez, com F(F/8). Concluímos que temos um formato do tipo $F(n/2^1)$, $F(n/2^2)$, $F(n/2^3)$,..., com $F(n/2^k)$ na k-ésima vez.

Nesse ponto, devemos ser cuidadosos com relação à interpretação de um programa escrito na linguagem C. Se a função trabalhasse com números reais, a divisão de n por dois nunca daria zero: poderíamos ir dividindo n inúmeras vezes e, apenas no limite de k tendendo a infinito, teríamos F(0). Contudo, o tipo da variável n é inteiro. No caso de uma divisão de um inteiro por outro inteiro, apenas a parte inteira da conta é dada como resultado, e a parte fracionária é descartada. Por exemplo, o inteiro 3 dividido pelo inteiro 2 resulta em 1, e não 1,5, já que a parte fracionária (0,5) é descartada. Dessa forma, ao fazermos 1 dividido por 2, obtemos como resultado 0. Podemos observar isso no quadro 2, em que iniciamos o cálculo para n=8.

Quadro 2. Cálculo da função F para vários valores de n.

Valor de n na linha 1	F(8)	F(4)	F(2)	F(1)
Valor de n na linha 6	F(8/2)=F(4)	F(4/2)=F(2)	F(2/2)=F(1)	F(1/2)=F(0)

Vamos analisar o caso para a situação na qual n é uma potência de 2: $n=8=2^3$. Assim, a função F é chamada 5 vezes:

- F(8), na primeira execução (k=1);
- F(4), na segunda execução (k=2);
- F(2), na terceira execução (k=3);
- F(1) na quarta execução (k = 4):
- F(0) na quinta execução (k = 5).

Observe a segunda linha do quadro 2:

- a primeira divisão corresponde a $F(n/2)=F(8/2^1)=F(8/2)=F(4)$;
- a segunda divisão corresponde a $F(n/4)=F(8/2^2)=F(8/4)=F(2)$;
- a terceira divisão corresponde a $F(n/8)=F(8/2^3)=F(8/8)=F(1)$;
- a quarta divisão corresponde a $F(n/8)=F(8/2^4)=F(8/16)=F(1/2)=F(0)$.

Não podemos esquecer que a função F(0) também é executada uma vez, mas, nesse caso, a linha 6 não é executada, já que o "if" da linha 2 não o permite. Assim, a função F é executada 5 vezes, para n igual a 8. Logo, para dado valor de n, a função F é executada k

vezes e, quando obtemos o termo F(8/16)=F(1/2)=F(n/2^k)=F(n/2^4), chegamos à penúltima execução. Na realidade, assim que temos 2^k igual a n, estamos na penúltima execução, pois chamamos a função F(1) e, ainda, será necessário mais uma chamada para F(1/2)=F(0).
Portanto, para dado n (que seja uma potência de 2), ficamos com:

$$n = 2^{k-2}$$

Podemos isolar k em função de n:

$$\log_2 n = \log_2 2^{k-2}$$

$$\log_2 n = (k-2)\log_2 2$$

$$\log_2 n = k - 2$$

$$k = \log_2 n + 2$$

Por exemplo, para n=8, temos:

$$k = \log_2 8 + 2 = 3 + 2 = 5$$

Dessa forma, percebemos que o crescimento de k é do tipo $\Theta(\log_2 n)$: o número de chamadas recursivas da função F é $\Theta(\log_2 n)$, ou seja, de ordem logarítmica.
Podemos fazer uma análise similar caso n não seja uma potência de 2. Trata-se de algo um pouco mais complexo, mas que chega a $\Theta(k)=\Theta(\log_2 n)$.
Assim, concluímos que a afirmativa II é correta.
Para identificarmos o número de vezes que a função G é chamada, temos de resolver a equação de recorrência. Nesse caso, sabemos que cada chamada da função F para determinado número n é de n vezes. Logo, temos uma equação de recorrência na forma:

$$T(n) = \begin{cases} \Theta(1) \quad se \quad n = 1 \\ T\left(\frac{n}{2}\right) + n \quad se \quad n \geq 1 \end{cases}$$

Uma maneira de resolvermos esse problema é utilizar o teorema mestre, que permite a solução de equações de recorrência no formato a seguir (CORMEN et al., 2012).

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Sendo que a ≥ 1 e b > 1, f(n) deve ser uma função assintoticamente positiva. No caso, f(n)=n. Para a equação de recorrência do algoritmo do enunciado, temos a = 1 e b =2.
Assim, calculamos $\log_b a$:

$$\log_b a = \log_2 1 = 0$$

Agora, devemos testar os três casos do teorema mestre (CORMEN et al., 2012), mostrados a seguir.

1. Se $f(n) = O(n^{\log_2 1-\varepsilon}) = O(n^{-\varepsilon})$ para alguma constante ε>0, então $T(n) = \Theta(n^{\log_2 1}) = \Theta(1)$

2. Se $f(n) = \Theta(n^{\log_2 1}) = \Theta(n^0) = \Theta(1)$, então $T(n) = \Theta(n^{\log_2 1} \log_2 n) = \Theta(\log_2 n)$

3. Se 0 $f(n) = \Omega(n^{\log_2 1+\varepsilon}) = \Omega(n^{\varepsilon})$ para alguma constante ε>0 e se f(n/2) ≤ c f(N) para alguma constante c<1 e valores de n suficientemente grandes, então $T(n) = \Theta(f(n))$.

Claramente, os casos 1 e 2 não são verdadeiros, mas o caso 3 é verdadeiro, uma vez que poderíamos supor ε=1 e c maior do que 0,5. Assim, sabemos que T(n)=Θ(f(n))=Θ(n). Quando uma função é do tipo Θ(n), ela pode ser limitada superiormente por c_2g(n) e inferiormente por c_1g(n). Além disso, T(n)=Θ(n) implica T(n)=O(n).
Finalmente, sabemos que O(n log n) representa crescimento mais rápido do que O(n). Portanto, se uma função é O(n), ela também é O(n log n).
Desse modo, concluímos que a afirmativa III é correta.

Alternativa correta: D.

Questão 5.

a) Inicialmente, vamos apresentar uma versão do programa do enunciado com as linhas numeradas.

```
1.   void mostrar(struct ListaEncadeada *lista){
2.       if( lista != NULL ){
3.           mostrar(lista->proximo);
4.           printf("%d", lista->dado);
5.       }
6.   }
```

Na figura 4, vemos a lista mostrada de forma esquemática. Nessa figura, cada elemento da lista é representado por um retângulo, indicado pelo nome da *struct*, ListaEncadeada. No programa, isso corresponde à variável lista, passada como argumento para a função "mostrar". Dentro do retângulo, na parte inferior à esquerda, temos uma divisão que corresponde ao inteiro "dado". No lado inferior à direita, temos o ponteiro para o próximo elemento da lista, chamado de "proximo".

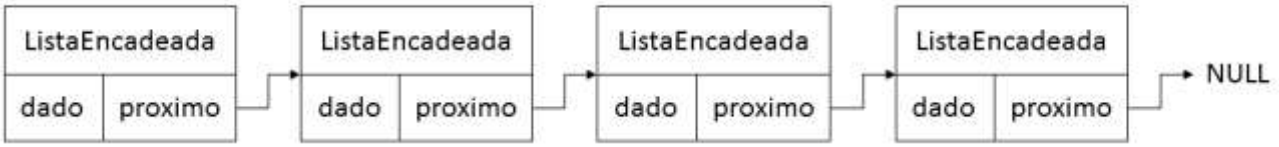


Figura 4. Representação da lista encadeada da questão.

Cada elemento da lista aponta para a posição de memória que contém o próximo elemento pelo uso do ponteiro "proximo". O término da lista é representado por NULL.

Devemos observar que a função "mostrar" percorre a lista linear de forma recursiva, elemento por elemento, como é mostrado na linha 3 do programa, até que, no último elemento da lista, o ponteiro lista->próximo aponta para o valor NULL, o fim da lista encadeada.

Quando a função "mostrar" é chamada com o ponteiro "lista" igual a NULL, ela não entra no *if* da linha 2 e retorna imediatamente. Isso faz com que as chamadas recursivas da função "mostrar" na linha 3 retornem, do último elemento para o primeiro. A próxima linha a ser executada é a linha 4, na qual é impresso o valor de "dado" contido na *struct*, ou seja, em cada elemento (nó) da lista.

Após a linha 4, a função "mostrar" tem apenas dois fechamentos de bloco, nas linhas 4 e 5. A função é chamada uma vez para cada elemento da lista. Se dobrarmos o tamanho da lista, dobraremos o número de vezes com que a função "mostrar" é chamada. Se dividirmos pela metade o tamanho da lista, o número de vezes com que a função mostrar é chamada cai pela metade. Na realidade, a função mostrar é chamada n+1 vezes, em que n é o tamanho da lista encadeada, ou seja, é o seu número de elementos. Dessa forma, a complexidade de tempo da função "mostrar" é linear, ou seja, é do tipo $\theta(n)$.

b) A presença da pilha no enunciado da questão sugere o seu uso para a inversão da ordem de impressão. Assim, a lista é lida elemento por elemento, com cada elemento sendo adicionado à pilha. Como a pilha é uma estrutura de dados do tipo LIFO (*Last In, First Out*),

o último elemento a ser adicionado é o primeiro a ser retirado. Logo, os elementos são retirados na ordem inversa pela qual foram adicionados.

A segunda parte do algoritmo consiste na leitura dos elementos da pilha que são retirados e com o conteúdo impresso na tela. A questão reporta-se a uma pilha de inteiros, mas, na realidade, a pilha deve ser formada por elementos do tipo "struct ListaEncadeada".

O programa completo, que contém, além da função solicitada no enunciado, a implementação da pilha, é mostrada a seguir.

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

struct ListaEncadeada{
   int dado;
   struct ListaEncadeada *proximo;
};

typedef void (*pFpush)(struct ListaEncadeada *lista);
typedef struct ListaEncadeada* (*pFpop)();
typedef int (*pFvazia)(void);

struct ListaEncadeada *topo;

typedef struct Pilha{
   pFpush push;
   pFpop pop;
   pFvazia vazia;
}Pilha;

void push(struct ListaEncadeada *lista)
{
    struct ListaEncadeada *elemento = malloc(sizeof(struct ListaEncadeada));
if( topo != NULL )
    {
        elemento->proximo = topo;
    }else{
        elemento->proximo = NULL;
    }
    elemento->dado = lista->dado;
    topo = elemento;

}

struct ListaEncadeada* pop()
{
    struct ListaEncadeada *ret = topo;
    if( topo != NULL )
    {
        topo = topo->proximo;
    }
    return ret;
}
int vazia()
{
    if( topo == NULL )
        return 1;
    else
        return 0;
```

```
}

void mostrar(struct ListaEncadeada *lista){
   Pilha p;
   p.pop = pop;
   p.push = push;
   p.vazia = vazia;
   while( lista != NULL ){
         p.push(lista);
         lista = lista->proximo;
   }
   while( !p.vazia() ){
         lista = p.pop();
        if( lista != NULL )
        {
            printf("%d", lista->dado);
            free(lista);
        }
        else
            break;
   }
}

int main()
{
    struct ListaEncadeada l1, l2, l3;
    topo = NULL;

    l1.dado = 1;
    l1.proximo = &l2;
    l2.dado = 2;
    l2.proximo = &l3;
    l3.dado = 3;
    l3.proximo = NULL;
    printf("Inicio do programa:\n");
    mostrar(&l1);

    return 0;
}
```

3. Indicações bibliográficas

- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução à estrutura de dados com técnicas de programação em C.* Rio de Janeiro: Elsevier, 2004.
- CORMEN, T. et al. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Campus-Elsevier, 2012.
- GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. *Data structures & algorithms in C++*. 2. ed. Hoboken: John Wiley & Sons, 2011.

Questão 6

Questão 6.[6]

Em um compilador, um analisador sintático descendente preditivo pode ser implementado com o auxílio de uma tabela construída a partir de uma gramática livre de contexto. Essa tabela, chamada tabela LL(k), indica a regra de produção a ser aplicada olhando-se o k-ésimo próximo símbolo lido, chamado *lookahead(k)*. Por motivo de eficiência, normalmente busca-se utilizar k=1. Considere a gramática livre de contexto G=({X,Y,Z},{a,b,c,d,e},P,X), em que P é composto pelas seguintes regras de produção:

$$X \rightarrow aZbXY \mid c$$

$$Y \rightarrow dX \mid \varepsilon$$

$$Z \rightarrow e$$

Considere, ainda, a seguinte tabela LL(1), construída a partir da gramática G, sendo $ o símbolo que representa o fim da cadeia. Essa tabela possui duas produções distintas na célula (Y, d), gerando, no analisador sintático, uma dúvida na escolha da regra de produção aplicada em determinados momentos de análise.

	a	b	c	d	e	$
X	$X \rightarrow aZbXY$		$X \rightarrow c$			
Y				$Y \rightarrow dX$ $Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$
Z					$Z \rightarrow e$	

Considerando que o processo de construção dessa tabela LL(1), a partir da gramática G, foi seguido corretamente, a existência de duas regras de produção distintas na célula (Y,d), neste caso específico, resulta

A. da ausência do símbolo de fim de cadeia ($) nas regras de produção.

B. de um não determinismo causado por uma ambiguidade na gramática.

C. do uso incorreto do símbolo de cadeia vazia (ε) nas regras de produção.

D. da presença de duas regras de produção com um único terminal no corpo.

E. da presença de duas regras de produção com o mesmo não terminal na cabeça.

[6]Questão 30 – Enade 2017.

1. Introdução teórica

Ambiguidade em gramáticas e construção de compiladores

Dizemos que uma gramática é ambígua se ela permitir diferentes derivações à esquerda para uma mesma sentença (COOPER e TORCZON, 2013). O mesmo é válido para diferentes derivações à direita. Ou seja, dada uma disciplina de derivação (mais à esquerda ou mais à direita), se for possível obtermos uma mesma sentença de diferentes formas, dizemos que a gramática é ambígua.

Do ponto de vista da construção de compiladores, isso é um grande problema. Para entendermos o motivo, vamos analisar o processo de funcionamento de um compilador típico.

A primeira etapa de um compilador é chamada de análise léxica. Nessa etapa, o código fonte é varrido (por isso, alguns textos chamam essa etapa de varredura) para encontrar os lexemas, instâncias particulares dos *tokens* que funcionam como categorias. Essas categorias são identificadas por meio de padrões, normalmente expressões regulares, de forma que um lexema em particular que respeitar a esse padrão vai ser enquadrado na categoria correspondente (representada pelo *token*).

A segunda etapa de um compilador é chamada de análise sintática ou léxica. Essa análise fornece um fluxo de *tokens* que devem ser processados pelo analisador sintático, que gera uma árvore sintática a partir dos *tokens*. A árvore sintática é construída de acordo com as regras de produção da gramática de uma linguagem. Ela corresponde a uma derivação ou a uma aplicação de um conjunto de regras de produção, sempre partindo de um símbolo inicial até chegar a uma cadeia específica que pertence à linguagem em questão.

A construção da árvore sintática orienta as etapas seguintes do processo de compilação, que associam um "significado" para a árvore e constroem um binário a partir dessa representação. Assim, uma árvore sintática vai dar origem a um binário ou a um programa em linguagem de máquina. Isso significa que o código-fonte original produz árvore sintática específica, e essa árvore norteia a construção do programa binário, escrito em código de máquina.

Contudo, se houver ambiguidade na gramática, existe a possibilidade de um mesmo código-fonte dar origem a duas ou mais árvores sintáticas diferentes, ou seja, a várias derivações. Como cada uma dessas árvores gera binários diferentes, o compilador "não

sabe" decidir qual é o programa "correto". É por isso que, na construção de compiladores e no projeto de linguagens de programação, devemos tomar cuidado a fim de evitarmos a formação de ambiguidades. Podemos adicionar regras de desambiguação, que vão decidir, posteriormente, qual é o significado correto da expressão, dando origem a um binário específico.

2. Análise das alternativas

A – Alternativa incorreta.
JUSTIFICATIVA. O símbolo de fim de cadeia não precisa ser colocado nas regras de produção.

B – Alternativa correta.
JUSTIFICATIVA. Para ilustrarmos a presença de uma ambiguidade na gramática, vamos fazer uma derivação mais à esquerda de uma mesma cadeia de duas formas diferentes. Inicialmente, vamos enumerar cada uma das regras no quadro 1.

Quadro 1. Enumeração das regras da gramática do enunciado.

Número da regra	Regra
1	$X \rightarrow aZbXY$
2	$X \rightarrow c$
3	$Y \rightarrow dX$
4	$Y \rightarrow \varepsilon$
5	$Z \rightarrow e$

Na derivação (parcial) apresentada a seguir, observamos que foi colocado o número da regra utilizada abaixo da flecha dupla. Essa notação não é universal, mas é bastante comum e facilita o entendimento da derivação. Iniciamos a derivação com o símbolo X, como especificado pela gramática, e chegamos até a forma sentencial aebaebcYY.

$$X \underset{1}{\Rightarrow} aZbXY \underset{5}{\Rightarrow} aebXY \underset{1}{\Rightarrow} aebaZbXYY \underset{5}{\Rightarrow} aebaebXYY \underset{2}{\Rightarrow} aebaebcYY$$

Nesse ponto, poderíamos fazer duas substituições: poderíamos substituir Y pela cadeia vazia (ε), de acordo com a regra 4, ou por dX, de acordo com a regra 3. Vamos utilizar a regra 4, seguida da regra 3 e, finalmente, a regra 2, conforme mostrado a seguir.

$$aebaebcYY \underset{4}{\Rightarrow} aebaebc\varepsilon Y \underset{3}{\Rightarrow} aebaebc\varepsilon dX \underset{2}{\Rightarrow} aebaebc\varepsilon dc$$

Devemos lembrar que a cadeia vazia (ε) desaparece e, assim, obtemos apenas a cadeia aebaebcdc, que é o fim da derivação, uma vez que ela é composta apenas por símbolos terminais.

Outra possibilidade consiste em utilizar a regra 3 em vez de a regra 4 na derivação anterior, seguida da regra 2 e, depois, da regra 4. Desse modo, ficamos com a derivação a seguir.

$$aebaebcYY \underset{3}{\Rightarrow} aebaebcdXY \underset{2}{\Rightarrow} aebaebcdcY \underset{4}{\Rightarrow} aebaebcdc\varepsilon$$

Vemos que derivamos a mesma cadeia aebaebcdc, pois a cadeia vazia (ε) desaparece. Assim, pudemos obter a mesma cadeia de duas formas diferentes, fazendo duas derivações à esquerda diferentes. Logo, a gramática é ambígua. Essa ambiguidade foi evidenciada pelas duas produções na célula (Y,d) na tabela LL(1) mostrada no enunciado.

C – Alternativa incorreta.

JUSTIFICATIVA. Não existe nenhum erro com relação ao uso da cadeia vazia nas regras de produção.

D – Alternativa incorreta.

JUSTIFICATIVA. Não há problemas com a utilização de regras de produção que contenham apenas um único terminal em seu corpo.

E – Alternativa incorreta.

JUSTIFICATIVA. A presença de regras de produção com o mesmo não terminal na cabeça não significa, necessariamente, que uma gramática é ambígua.

3. Indicações bibliográficas

- AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas.* 2. ed. São Paulo: Person Addison Wesley, 2007.
- COOPER, K. D. e TORCZON, L. *Construindo compiladores.* 2. ed. Rio de Janeiro: Campus-Elsevier, 2013.
- SIPSER, M. *Introdução à teoria da computação.* São Paulo: Cengage, 2005.

Questão 7

Questão 7.[7]

Uma relação de equivalência é uma relação binária R em um conjunto A, tal que R é reflexiva, simétrica e transitiva.

Considere as relações binárias apresentadas a seguir.

- R1={(a,b): $a,b \in$ N e $a = b$}
- R2={(a,b): $a,b \in$ N e $a \leq b$}
- R3={(a,b): $a,b \in$ N e $a=b-1$}
- R4={(a,b): $a,b \in$ N e $a+b$ é um número par}

São relações de equivalência apenas o que apresenta em

A. R2 e R3. B. R1 e R3. C. R1 e R4. D. R1, R2 e R4. E. R2, R3 e R4.

1. Introdução teórica

Relações de equivalência

O conceito de relações na Ciência da Computação está intimamente ligado ao conceito matemático de função. A ideia de função é a de "promover um mapeamento" entre elementos de dois conjuntos.

Por exemplo, podemos imaginar um conjunto P formado por três pessoas:

P={João, Maria, José}.

Também podemos imaginar um conjunto T formado por alguns times de futebol:

T={São Paulo, Palmeiras, Vasco, Corinthians}.

Vamos fazer um mapeamento entre as pessoas e os times de futebol, conforme segue.

- João torce para o Palmeiras.
- Maria torce para o Vasco.
- José torce para o Corinthians.

[7]Questão 16 – Enade 2017.

Dessa forma, estamos definindo uma função *f(x)* que associa pessoas (x) aos times de futebol. As pessoas pertencem ao domínio da função, e os times são o contradomínio. Assim, podemos escrever:

$$f: P \rightarrow T$$

Na computação, utilizamos um tipo especial de função, que chamamos de predicado ou propriedade (SIPSER, 2005; BLAUTH, 2013; VIALAR, 2017). Um predicado é uma função cujo contradomínio é apenas o conjunto {VERDADEIRO, FALSO}. Assim, no caso do exemplo anterior, temos que:

- o predicado CORINTHIANO(José) é igual a VERDADEIRO;
- o predicado CORINTHIANO(João) é igual a FALSO.

Logo, um predicado é simplesmente uma função que retorna verdadeiro ou falso para diversos elementos. Obviamente, esse tipo de função é útil em sistemas digitais, em especial nos computadores digitais, já que, nesses casos, trabalhamos com sistemas binários, nos quais sempre operamos com bits, ou seja, zeros ou uns.

Podemos definir predicados (ou propriedades) nos quais o domínio é composto de vários argumentos, e chamamos isso de relação. Por exemplo, suponha a relação "maior", definida sobre dois números inteiros. Nesse caso, temos que:

- maior(3,1)=VERDADEIRO;
- maior(1,5)=FALSO.

Uma relação que recebe dois argumentos é chamada de relação binária. Podemos escrever aRb para denotar uma relação binária R(a,b): por exemplo, costumamos escrever 1<2 e não <(1,2). A notação aRb é chamada de notação infixa (SIPSER, 2005; BLAUTH, 2013; VIALAR, 2017).

Vejamos as situações a seguir, referentes às relações binárias (SIPSER, 2005).

- Se uma relação binária R é tal que, para qualquer x do seu domínio, xRx=VERDADEIRO, então essa relação é reflexiva.
- Se uma relação binária R é tal que, se tomarmos dois x e y quaisquer do seu domínio, xRy=VERDADEIRO e yRx=VERDADEIRO, então essa relação é simétrica.
- Se uma relação binária R é tal que, se tomarmos quaisquer três elementos do seu domínio, x, y e z, e soubermos que xRy=VERDADEIRO e yRz=VERDADEIRO, e isso garantir que xRz=VERDADEIRO (sempre), então dizemos que essa relação é transitiva.

Nem todas as relações binárias gozam dessas três características simultaneamente, mas, quando isso acontecer, dizemos que essa relação binária é uma relação de equivalência (SIPSER, 2005; BLAUTH, 2013; VIALAR, 2017).

2. Resolução da questão

A estratégia para a resolução da questão é simplesmente encontrar quais das relações dadas são simultaneamente reflexivas, simétricas e transitivas. Quando isso acontecer, encontramos as relações de equivalência.
Para classificarmos cada uma das relações, devemos provar que uma propriedade é válida sempre, ou encontrar um contraexemplo no qual a propriedade seja violada.

Identificamos, a seguir, quais das relações dadas no enunciado são reflexivas.

- R1 é uma relação reflexiva, pois $x=x$ (sempre).
- R2 é uma relação reflexiva, pois $x \leq x$ (sempre).
- R3 não é uma relação reflexiva, pois $x \neq x-1$.
- R4 é uma relação reflexiva, pois $x+x=2x$ é um número par (sempre).

Identificamos, a seguir, quais das relações dadas no enunciado são simétricas.

- R1 é uma relação simétrica, pois, se $x=y$, então $y=x$ (sempre)
- R2 não é simétrica, pois, se $x \leq y$, não podemos afirmar que $y \leq x$ (isso é verdadeiro apenas no caso da igualdade).
- R3 não é simétrica, pois, se $x=y+1$, não podemos afirmar que $y=x+1$.
- R4 é simétrica, pois, se $x+y$ é um número par, $y+x$ também é um número par (sempre).

Identificamos, a seguir, quais das relações dadas no enunciado são transitivas.

- R1 é transitiva, pois, se $x=y$ e $y=z$, então $x=z$ (sempre).
- R2 é transitiva, pois se $x \leq y$ e $y \leq z$, então $x \leq z$ (sempre).
- R3 não é transitiva, pois, se $x=y-1$ e $y=z-1$, $x=z-2$ (diferente de $x=z-1$).
- R4 é transitiva, e isso pode ser provado da forma a seguir.
    - Se $x+y$ pertence à relação R4, então $x+y=2k$.
    - Se $y+z$ pertence à relação R4, então $y+z=2l$.
    - Somando os dois resultados, temos $x+y+y+z=x+2y+z=2k+2l$.

- Como x+2y+z=2k+2l, sabemos que x+z=2k+2l-2y e, assim, x+z=2(k+l-y), que é um número par. Logo, x+z é par, e R4 é transitiva, pois xR4y e yR4z implica xR4z (sempre).

Sumarizando, temos as conclusões a seguir.

- R1, R2 e R4 são reflexivas.
- R1 e R4 são simétricas.
- R1, R2 e R4 são transitivas.

Apenas as relações R1 e R4 são simultaneamente reflexivas, simétricas e transitivas; portanto, podem ser consideradas relações de equivalência.

Alternativa correta: C.

3. Indicações bibliográficas

- BLAUTH, P. M. *Matemática discreta para computação e informática.* 4. ed. Porto Alegre: Bookman, 2013.
- SIPSER, M. *Introdução à teoria da computação.* São Paulo: Cengage, 2005.
- VIALAR, T. *Handbook of Mathematics*. Paris: HDBoM, 2017.

Questão 8

Questão 8.[8]

Um país utiliza moedas de 1, 5, 10, 25 e 50 centavos. Um programador desenvolveu o método a seguir, que implementa a estratégia gulosa para o problema do troco mínimo. Esse método recebe como parâmetro um valor inteiro, em centavos, e retorna um *array* no qual cada posição indica a quantidade de moedas de cada valor.

```
public static int[] troco(int valor){
     int[] moedas = new int[5];
     moedas[4] = valor / 50;
     valor = valor % 50;
     moedas[3] = valor / 25;
     valor = valor % 25;
     moedas[2] = valor / 10;
     valor = valor % 10;
     moedas[1] = valor / 5;
     valor = valor % 5;
     moedas[0] = valor;
     return(moedas);
}
```

Considerando o método apresentado, avalie as asserções e a relação proposta entre elas.

I. O método guloso encontra o menor número de moedas para o valor de entrada, considerando as moedas do país.

PORQUE

II. Métodos gulosos sempre encontram a solução global ótima.

A respeito dessas asserções, assinale a opção correta.

A. As asserções I e II são proposições verdadeiras, e a asserção II justifica a I.

B. As asserções I e II são proposições verdadeiras, e a II não justifica a I.

C. A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.

D. A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.

E. As asserções I e II são proposições falsas.

[8]Questão 22 – Enade 2017.

1. Introdução teórica

Algoritmos gulosos

Imagine que você está tentando resolver um problema computacional, de otimização, e que esse problema possa ser dividido em uma série de decisões. A cada iteração do seu algoritmo, você deve tomar uma decisão baseada apenas nas informações disponíveis naquele momento. Não é possível voltar atrás para desfazer uma decisão feita anteriormente. Como saber se o conjunto de decisões tomadas vai dar origem à melhor solução para o problema? Em outras palavras, como saber se encontramos a solução ótima global do problema de otimização?

Vamos ilustrar esse problema com um exemplo. Suponha que você queira fazer um aplicativo de celular que determine a melhor rota possível no trânsito, e que, por melhor rota, entenda-se o menor tempo para chegar ao destino desejado. Imagine que você tenha a posição do carro (com as informações do GPS do celular) e o mapa das ruas, de forma que você sabe exatamente onde o carro está. Você conhece a situação do trânsito local, ou seja, você sabe como é o trânsito nos arredores de onde o carro está, mas não em toda a cidade. A pergunta que fazemos é: será que apenas com essas informações podemos encontrar a melhor rota?

Imagine que o trânsito pode estar "bom" em uma pista vizinha à sua, de forma que você decide mudar para essa pista. Subitamente, a "pista para", e você decide novamente mudar de pista. Então, vendo que uma das ruas laterais está com trânsito menor, você decide alterar a rota, mas percebe que esse trajeto é bem mais longo que o anterior e que o número de semáforos é muito maior, de modo que o tempo do trajeto seria elevado. Adicionalmente, o trânsito mais à frente pode estar até pior do que o anterior. Esse é um cenário no qual simplesmente tomar a melhor decisão com base apenas em informações imediatas (somente olhando para o trânsito ao seu redor, sem levar em consideração outros fatores, como o trânsito mais adiante, o comprimento das rotas alternativas e os números de semáforos) é chamado, em computação, de "algoritmo guloso". Obviamente, para o cenário de trânsito proposto, isso não vai funcionar.

Nos algoritmos gulosos, pressupomos que, tomando decisões ótimas locais (com informação limitada), chegaremos a uma decisão ótima global. Isso, obviamente, não é verdade sempre, mas existem casos em que tal estratégia pode funcionar: nesses casos, alcançamos uma classe de algoritmos que são bastante simples e úteis.

Vamos supor um novo problema, chamado de problema da mochila fracionária. Nesse problema, existe uma mochila com capacidade finita e definida de materiais que podem ser colocados dentro dela (por exemplo, limitada pelo peso máximo). Há uma série de itens que podem ser colocados no interior da mochila, sendo que cada item tem um preço associado e uma disponibilidade associada. O objetivo é transportar a maior quantidade possível de valor na mochila, ou seja, preenchê-la de forma a maximizar o valor transportado. Imagine, também, que os itens podem ser comprados a granel: por exemplo, podemos comprar 5 quilos de arroz, 0,5 quilo de feijão ou 1,5 quilos de milho.

Esse problema pode ser resolvido por um algoritmo do formato descrito a seguir.

- Primeiramente, devemos ordenar os itens disponíveis por ordem crescente da razão entre o valor do item dividido pela unidade de peso (que é normalmente o modo como vemos o preço de itens vendidos a granel, em que se cobra pelo peso).
- Depois, procuramos preencher a mochila com a maior quantidade possível do item de maior valor por quilo.
- Finalmente, quando o item de maior valor por quilo "acabar", vamos para o item seguinte de maior valor agregado, e assim sucessivamente, até completarmos a capacidade da mochila.

Um algoritmo que utiliza esse tipo de estratégia, no qual tomamos decisões localmente ótimas, é chamado de algoritmo guloso.

2. Análise das afirmativas

I – Afirmativa correta.

JUSTIFICATIVA. Como o algoritmo mostrado no enunciado começa com as moedas de maior valor e prossegue para as moedas de menor valor e, também, como o problema é essencialmente similar ao problema da mochila fracionada, ele está correto.

II – Afirmativa incorreta.

JUSTIFICATIVA. Existem muitos problemas de otimização que não podem ser resolvidos por algoritmos gulosos. Segundo Cormen et al. (2012), problemas que podem ser resolvidos por algoritmos gulosos apresentam duas características:

- a "escolha gulosa", na qual uma solução global ótima pode ser construída a partir de escolhas locais ótimas;
- a "subestrutura ótima".

Um problema apresenta uma subestrutura ótima quando a "solução ótima global do problema original contém a solução ótima dos subproblemas" (CORMEN et al., 2012).
Se um problema de otimização não tiver essas características, um algoritmo guloso provavelmente não vai ser capaz de encontrar soluções ótimas globais.

Alternativa correta: C.

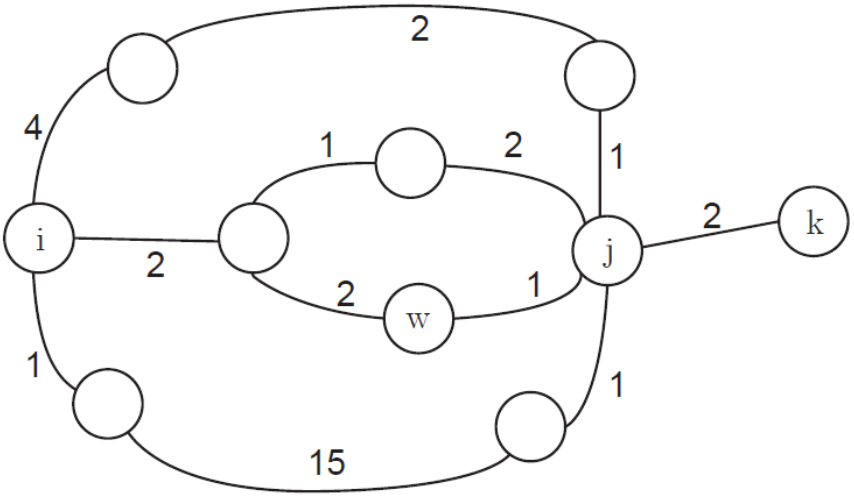
3. Indicações bibliográficas

- CORMEN, T. et al. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Campus-Elsevier, 2012.
- FEOFOLOFF, P. *Algoritmos gulosos*. Disponível em <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/guloso.html>. Acesso em 19 out. 2023.
- FEOFOLOFF, P. *Problema da mochila fracionária*. Disponível em <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-frac.html>. Acesso em 19 out. 2023.
- GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. *Data structures & algorithms in C++*. 2. ed. Hoboken: John Wiley & Sons, 2011.

Questões 9 e 10

Questão 9.[9]

A figura a seguir exibe um grafo que representa um mapa rodoviário, no qual os vértices representam cidades e as arestas representam vias. Os pesos indicam o tempo atual de deslocamento entre duas cidades.



Considerando que os tempos de ida e volta são iguais para qualquer via, avalie as afirmativas a seguir, acerca desse grafo.

I. Dado o vértice de origem i, o algoritmo de Dijkstra encontra o menor tempo de deslocamento entre a cidade i e todas as demais cidades do grafo.

II. Uma árvore geradora de custo mínimo gerada pelo algoritmo de Kruskal contém um caminho de custo mínimo cuja origem é i e cujo destino é k.

III. Se um caminho de custo mínimo entre os vértices i e k contém o vértice w, então o subcaminho de origem w e destino k deve também ser mínimo.

É correto o que se afirma em

A. I, apenas.

B. II, apenas.

C. I e III, apenas.

D. II e III, apenas.

E. I, II e III.

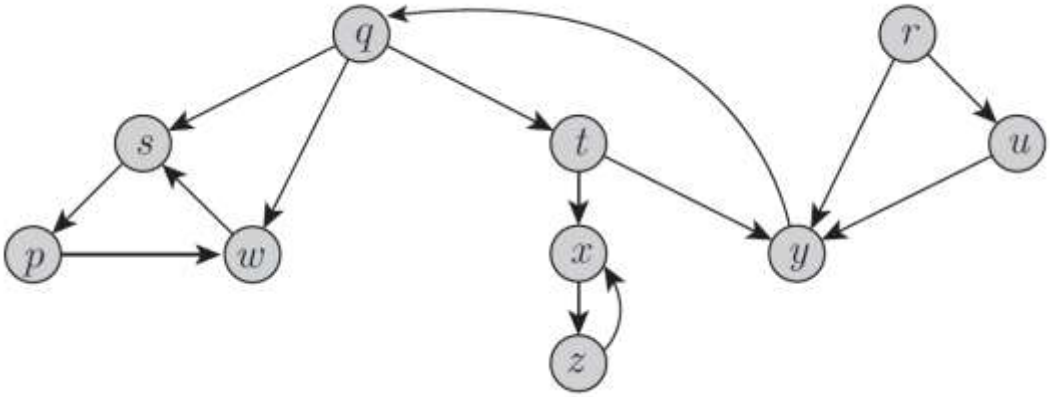
[9]Questão 24 – Enade 2017.

Questão 10.[10]

Leia o texto a seguir.

A busca primeiro em profundidade é um algoritmo de exploração sistemática em grafos, em que as arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda tem arestas inexploradas saindo dele. Quando todas as arestas de v são exploradas, a busca regressa para explorar as arestas que deixam o vértice a partir do qual v foi descoberto. Esse processo continua até que todos os vértices acessíveis a partir do vértice inicial sejam explorados.
CORMEN, T. H. et al. *Introduction to algorithms*. 3. ed. Cambridge, Massachusetts: The MIT Press, 2009 (com adaptações).

Considere o grafo abaixo.



Com base nas informações apresentadas, faça o que se pede nos itens a seguir.

a) Mostre a sequência de vértices descobertos no grafo durante a execução de uma busca em profundidade com controle de estados repetidos. Para isso, utilize o vértice r como inicial. No caso de um vértice explorado ter mais de um vértice adjacente, utilize a ordem alfabética crescente como critério para priorizar a exploração.
b) Faça uma representação da matriz de adjacências desse grafo, podendo os zeros serem omitidos nessa matriz.

1. Introdução teórica

1.1. Algoritmo de Dijkstra

Um dos problemas típicos que temos ao analisarmos um grafo é encontrar o caminho mínimo entre dois vértices. Se pensarmos que cada vértice corresponde a uma cidade e que cada aresta corresponde a uma distância, esse seria o problema equivalente a tentar descobrir o caminho de menor distância entre duas cidades. Existem várias abordagens para

[10]Questão Discursiva 03 – Enade 2017.

esse tipo de questão; entre elas, temos o algoritmo de Dijkstra, que funciona bem com grafos cujos pesos associados às suas arestas são valores não negativos (SKIENA, 2008).

Na primeira etapa, o algoritmo de Dijkstra inicializa as suas estruturas de dados, inclusive aquela que contém os nós conhecidos, para os quais já temos o valor mínimo calculado considerando os nós visitados até dado instante do algoritmo (SKIENA, 2008; CORMEN et al., 2012). O algoritmo de Dijkstra também precisa de um vetor que contenha a distância entre o nó atual que está sendo avaliado em determinada iteração do algoritmo e o nó inicial.

O vetor com as distâncias deve ser inicializado de forma que o cálculo posterior de qualquer distância real seja inferior ao valor inicial, o que pode ser feito por meio de uma *flag* de software ou de um símbolo especial de valor "infinito". O nó anterior que deu origem ao cálculo da distância também deve ser armazenado em alguma estrutura de dados.

O nó inicial, aqui chamado de nó s, é aquele por onde o algoritmo começa a sua execução. Os nós diretamente ligados ao nó inicial são analisados, um a um, com base no peso das suas arestas, o que leva ao cálculo do valor intermediário do custo do caminho total. Depois disso, o nó s é adicionado à lista de nós conhecidos.

Em um momento posterior, o algoritmo escolhe um nó que não esteja na lista de nós conhecidos e que tenha a menor estimativa de custo associado, que chamamos de nó n_2. Aplica-se a mesma abordagem descrita anteriormente, considerando a "distância" (ou peso) entre o nó s e os nós seguintes ao nó n_2. Isso pode levar à atualização da estrutura de dados que contém as diversas distâncias (atualizada a cada iteração do algoritmo). Também devemos atualizar a estrutura de dados que contém os nós predecessores associados aos caminhos parciais calculados até o momento (que serão os caminhos de menor comprimento no final da execução do algoritmo).

O processo precisa ser executado para todos os nós acessíveis pelo nó n_2, sendo que, no final do processo, o nó n_2 é adicionado à estrutura de dados com os nós conhecidos. O processo é, então, repetido, escolhendo-se novamente um outro nó que não esteja nessa lista e que tenha a menor distância estimada. A ideia é que tal processo seja repetido até que todos os nós do grafo estejam na estrutura de dados de nós conhecidos.

Para ilustrar o cálculo da distância, vamos supor dois nós, n_i e n_k, ligados por uma aresta de peso $p(n_i,n_k)$. Imagine que o vetor que contém a estimativa das distâncias seja chamado de dist, de forma que dist[n_i] corresponde à distância mínima estimada entre o nó inicial s e o nó n_i.

Quando o algoritmo de Dijkstra estiver analisando os nós ligados a n_i em dado momento, ele calcula dist[n_i]+p(n_i,n_k). Esse valor deve ser comparado com dist[n_k]. Se, nesse ponto, descobrirmos que dist[n_i]+p(n_i,n_k)<dist[n_k], acabamos de encontrar um caminho mais curto entre o nó inicial s e o nó n_k. Nesse instante, devemos atualizar dist[n_k] fazendo dist[n_k]= dist[n_i]+p(n_i,n_k) e atualizar a lista de nós predecessores (que, agora, é o nó n_i). No entanto, se dist[n_i]+p(n_i,n_k)>dist[n_k], o caminho encontrado é pior do que o anterior, e nada deve ser mudado.

Resta ainda um caso adicional, associado à igualdade se dist[n_i]+p(n_i,n_k)=dist[n_k]. Isso significa que outro caminho de custo mínimo foi encontrado, com o mesmo custo do caminho anterior. Esse caso implica aumento de complexidade na estrutura de dados que armazena os nós predecessores, que deve ser alterada para armazenar não apenas um nó, mas uma lista de nós.

1.2. Algoritmo de Kruskal

Suponha um conjunto de vértices e arestas conectados, formando um grafo G. Um exemplo típico dessa situação é o mapa de uma região, com cada cidade representada por vértices, sendo que as estradas que conectam essas cidades são representadas pelas arestas, como mostrado na figura 1.

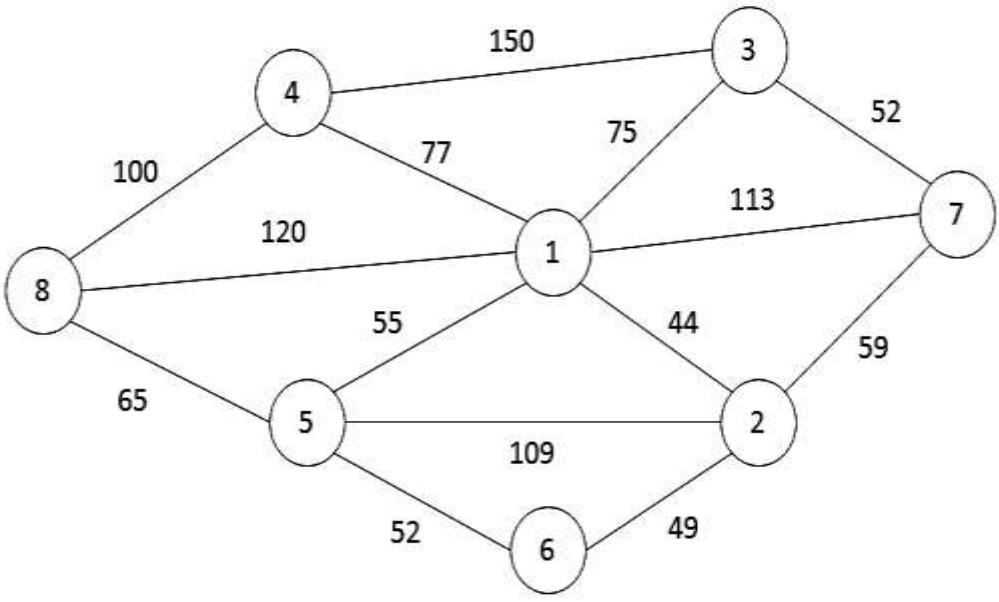


Figura 1. Exemplo de um grafo, nos quais os vértices são cidades (aqui representadas por números) e as arestas são as distâncias entre as cidades (considerando como distância o comprimento das estradas entre as cidades).

Suponha o seguinte problema: queremos encontrar um caminho que ligue todas as cidades da figura 1 com o menor comprimento possível. Observe que desejamos achar um caminho no formato de árvore, e não uma "circulação" entre as cidades. Assim, precisamos de uma lista de cidades ligadas pelo menor comprimento total possível, não sendo necessário terminarmos na mesma cidade em que começamos o trajeto. Na computação, chamamos essa árvore de "árvore geradora de custo mínimo" (no inglês, o termo utilizado é *minimum spanning tree*, ou MST).

O algoritmo de Kruskal resolve esse problema pelo emprego de uma abordagem típica de um algoritmo guloso. Definimos uma série de conjuntos inicialmente contendo apenas um único vértice (uma única cidade do mapa). Depois, organizamos as arestas que ligam as cidades em ordem não decrescente, utilizando a distância entre as cidades como critério de organização. Colocamos tais arestas em uma estrutura de dados que preserve a ordem, como, por exemplo, uma fila (aqui chamada de Q).

Agora, vamos unir os conjuntos de cidades criados anteriormente, produzindo um novo conjunto que vai conter a árvore geradora de custo mínimo (aqui chamada de T). O algoritmo deve construir esse novo conjunto enquanto o número de arestas for menor do que n-1, em que n é o número de vértices do grafo.

Nesse ponto, é importante introduzirmos duas definições usadas em teoria dos grafos. Um grafo que não tem ciclo é chamado de grafo acíclico. Um grafo nos quais todos os vértices estão ligados é chamado de grafo conexo. Se existirem um ou mais vértices que não estão conectados, dizemos que o grafo é não conexo. Se um grafo for acíclico e não conexo, dizemos que ele é uma floresta. Esse nome decorre do seguinte: é como se tivéssemos diversas árvores uma do lado da outra, sem conexão entre os seus vértices, como ocorre em uma floresta de árvores "reais".

A cada iteração, o algoritmo deve unir os conjuntos definidos inicialmente para cada cidade, selecionando uma aresta de custo mínimo entre os conjuntos existentes. Os objetivos são:

- evitar formarmos um grafo que seja uma floresta;
- interligar os grafos existentes por uma aresta de custo mínimo.

Quando selecionamos uma aresta da fila Q, sabemos que ela une dois vértices (pela definição de aresta), mas temos de tomar cuidado para que ela una dois conjuntos de vértices diferentes, ou seja, para que ela una árvores diferentes, e apresente o menor custo possível. Ao fazermos essa união, criamos uma nova árvore maior e com mais vértices. Essa árvore vai sendo armazenada em T e é parte da árvore geradora de custo mínimo. Além

disso, precisamos unir, também, os conjuntos de vértices que correspondem às extremidades da aresta adicionada em um único conjunto, uma vez que fazem parte do mesmo grafo.

Esse processo iterativo é um algoritmo guloso, visto que, a cada iteração, o algoritmo escolhe a aresta de custo mínimo que respeita a condição de que os seus vértices pertençam a conjuntos diferentes. Além disso, a cada iteração o algoritmo constrói um pedaço da árvore, não sendo necessário que o algoritmo "volte para trás" ou reconsidere decisões anteriores. Cada escolha localmente ótima leva à construção de uma solução globalmente ótima no final. Quando a árvore T contiver mais de n-1 arestas, o algoritmo pode parar e, assim, sabemos que T é uma árvore geradora de custo mínimo.

Se executarmos o algoritmo de Kruskal para o grafo da figura 1, obtemos a árvore mostrada na figura 2. Observamos que os nós foram deixados nas suas posições originais para facilitar a identificação, mas a estrutura da figura 2 é uma árvore. O custo dessa árvore é 398, como pode ser visto, somando-se o valor de todas as arestas contidas na árvore.

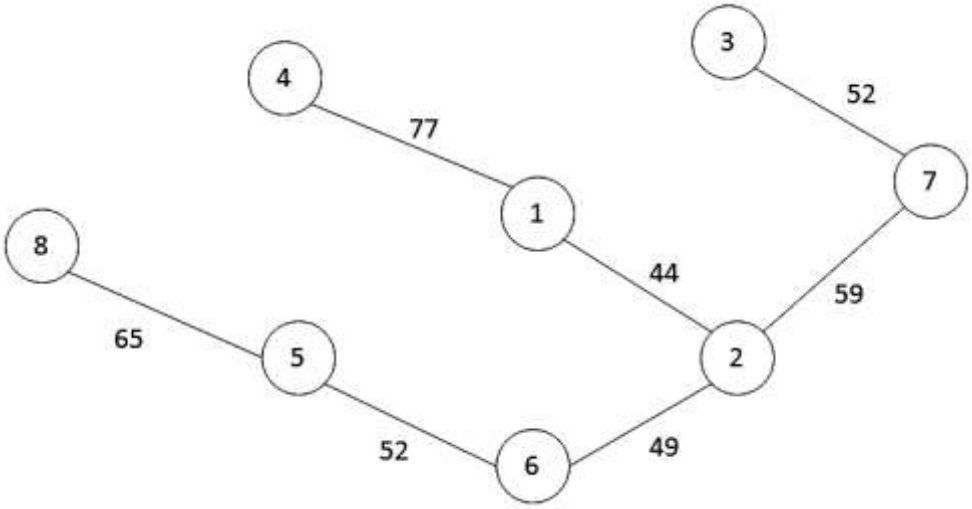


Figura 2. Árvore geradora de custo mínimo para o grafo da figura 1.

1.3. Algoritmo de busca primeiro em profundidade

Diferentemente do algoritmo de Dijkstra, cujo objetivo é encontrar o caminho mínimo entre dois pontos, o algoritmo de busca primeiro em profundidade procura percorrer (ou visitar) todos os nós de um grafo utilizando as arestas como caminhos (FEOFILOFF, 2019).

Vejamos, na Listagem 1, uma possível implementação do algoritmo da busca primeiro em profundidade (CORMEN et al., 2012), apresentado nas listagens 1 e 2. Observe que o algoritmo é apresentado em pseudocódigo e que é dividido em duas funções: a função principal DFS(G) e a função DFS-Visita(G,u).

A função DFS-Visita(G,u) é uma função recursiva chamada inicialmente pela função DFS(G) e que tem o objetivo de visitar cada um dos nós do grafo.

```
DFS(G)
1.    para cada vértice u em G.V:
2.        u.cor = branco
3.        u.pred = NULL
4.    tempo = 0
5.    para cada vértice u em G.V:
6.        se u.cor == branco:
7.            DFS-Visita(G, u)
```

Listagem 1. Função principal do algoritmo de busca em largura.
CORMEN et al., 2012 (com adaptações).

Cada nó tem uma cor associada. O propósito dessa cor é diferenciar os nós de acordo com a função que eles têm, em dado momento, no algoritmo. Nós brancos são aqueles que nunca foram visitados. Assim que eles são visitados alguma vez, a cor associada é mudada para cinza.

Um nó é associado com a cor cinza quando está sendo avaliado pela função DFS-Visita, como mostrado na linha 3 da Listagem 2. Na linha 4 da função DFS-Visita, o algoritmo verifica todos os nós adjacentes ao nó em análise (no caso, o nó u). Na linha 5, apenas os nós adjacentes com a cor branca são levados em consideração, ou seja, apenas aqueles que nunca foram visitados pela função DFS-Visita.

```
DFS-Visita(G,u)
1.     tempo = tempo + 1
2.     u.inicio = tempo
3.     u.cor = cinza
4.     para cada v em G.adj[u]:
5.         se v.cor == branco:
6.             v.pred = u
7.             DFS-Visita(G, v)
8.     u.cor = preto
9.     tempo = tempo + 1
10.    u.fim = tempo
```

Listagem 2. Função para a visita dos nós da busca em largura.
CORMEN et al., 2012 (com adaptações).

Além do atributo de cor, cada nó tem associada uma informação sobre o nó predecessor, como pode ser visto na linha 6. Quando o laço que avalia todos os nós adjacentes termina (laço da linha 4), é necessário mudar a cor do nó para preto (na linha 8), o que indica o fim da análise.

Outra informação a ser armazenada em cada nó está relacionada ao instante no qual os nós são avaliados. Dois momentos devem ser armazenados:

- o instante no qual o nó começa a ser avaliado pela função DFS-Visita, logo antes de sua cor mudar para cinza (aqui chamado de início e associado à linha 2 da listagem 2);
- o instante imediatamente posterior à associação com a cor preta, no fim da função DFS-Visita (linha 10 da listagem 2).

2. Análise das afirmativas e resolução da questão

Questão 9.

I – Afirmativa correta.
JUSTIFICATIVA. Como o algoritmo de Dijkstra permite que achemos o caminho mínimo entre dois vértices em um grafo e no grafo apresentado no enunciado os pesos das arestas correspondem aos tempos de deslocamento, o caminho de menor peso encontrado pelo algoritmo vai encontrar o menor tempo de deslocamento entre as cidades.

II – Afirmativa correta.
JUSTIFICATIVA. Uma vez que o algoritmo de Kruskal encontra uma árvore geradora de custo mínimo para o grafo, essa árvore contém o caminho de custo mínimo com origem em i e com destino em k.

III – Afirmativa correta.
JUSTIFICATIVA. A natureza do problema do caminho mínimo, propícia para a aplicação de algoritmos gulosos, garante que o subproblema de encontrar o subcaminho mínimo com origem em w e com destino em k também esteja no caminho entre i e k (desde que w esteja no caminho mínimo entre i e k).

Alternativa correta: E.

Questão 10.

a) Na resolução, vamos indicar o momento no qual um vértice é descoberto pela impressão do seu nome. Isso corresponde a um comando de saída de resultados, como o comando printf da linguagem C. Como não estamos fazendo uma resolução em nenhuma linguagem

em particular, vamos utilizar apenas o nome genérico "imprime-se x" para indicar a saída de um valor x na tela.

Começando pelo nó r (imprime-se r), devemos explorar seus nós adjacentes com cor branca e em ordem alfabética, visitando inicialmente o nó u (imprime-se u). Chamamos novamente a função DFS-Visita para o nó u e visitamos o nó y (imprime-se y). Chamamos novamente DFS-Visita(G, y) e visitamos o nó q (imprime-se q). O nó q tem três nós adjacentes com cor branca: o nó s, o nó t e o nó w. O nó s é visitado primeiro (imprime-se s). Ele apresenta apenas o nó p como nó adjacente na cor branca (imprime-se p). Isso também ocorre para o nó p e chega-se ao nó w (imprime-se w).

Nesse ponto, a função DFS-Visita(G,w) não pode visitar o nó s, pois ele mudou para a cor cinza anteriormente (a função deve retornar). As chamadas anteriores de DFS-Visita também devem terminar até retornarmos para o nó q.

Agora, voltando ao laço de avaliação de nós da linha 4 da listagem 2, é o momento de visitarmos o nó t (imprime-se t). Isso nos leva à visita do nó x (imprime-se x), seguida da visita ao nó z (imprime-se z). Nesse ponto, o nó z não pode mais visitar o nó x, pois sua cor não é mais branca, mas, sim, cinza. A função DFS-Visita volta, então, para o nó x e, depois, para o nó t. Ainda que o nó y seja adjacente ao nó t, ele também não está mais com a cor branca e não pode ser visitado.

Nesse ponto, voltamos para o nó q, mas não podemos visitar o nó w, que também não está mais com a cor branca. A função DFS-Visita volta para o nó y e, depois, para o nó u, de onde ela foi chamada originalmente. A função DFS-Visita para o nó u retorna para o nó r. Novamente, o nó y não pode ser visitado, pois não está mais na cor branca, e o algoritmo termina.

Assim, a ordem de impressão (visita) dos vértices é: r-u-y-q-s-p-w-t-x-z.

Devemos observar que cada nó é impresso apenas uma vez, logo que é descoberto. Os retornos da função DFS-Visita não devem causar nenhuma impressão.

Podemos encontrar outras respostas para o problema, caso sejam utilizadas diferentes versões do algoritmo. Uma alternativa de resposta envolve utilizar um algoritmo um pouco diferente, que faz uso de uma pilha em ordem lexicográfica.

Nesse caso, a saída seria: r–y–q–w–t–x–z–s–p–u.

Há outras versões, como aquelas nas quais empilhamos um nó mais de uma vez (ou seja, não é feita uma verificação se o nó já teria sido previamente inserido na pilha) e utilizamos ordem lexicográfica.

Nesse caso, a saída seria: r–y–q–w–s–p–t–x–z–u.

Finalmente, invertendo a ordem de inserção dos objetos na pilha, obtemos a seguinte saída: r–u–y–q–s–p–t–x–z–w.

b) Uma matriz de adjacência é uma estrutura de dados utilizada para representar um grafo. No caso de um grafo orientado, em que as arestas são flechas e têm direção, essa matriz deve representar não apenas a existência de uma conexão entre os nós do grafo, mas também a direção da conexão.

Assim, a estratégia da representação é fazermos uma matriz em que cada linha e cada coluna representa um nó do grafo. No caso do problema, precisamos ter uma matriz com 10 linhas e 10 colunas, uma vez que o grafo apresenta 10 nós.

Uma vez que o grafo é direcionado, podemos criar uma matriz na qual cada linha representa o nó do qual uma aresta está "saindo". Colocamos o número 1 na coluna correspondente à aresta que está "chegando". Por exemplo, na primeira linha da matriz, que corresponde às arestas que têm origem no nó p, existe apenas uma coluna com o valor um, a coluna que corresponde ao nó w. Isso representa a única aresta (ou "flecha", na figura) que sai do nó p e chega ao nó w. As demais colunas devem estar com o valor 0 (zero), pois não existe mais nenhuma aresta saindo do nó w. Se aplicarmos esse processo para todas as linhas e para todas as colunas, chegamos à matriz M mostrada a seguir.

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Essa matriz pode ser representada no quadro 1, com o nome dos nós correspondentes mostrados na primeira coluna e na primeira linha.

Quadro 1. Representação da matriz de adjacência para o grafo do enunciado (ENADE 2017).

	p	q	R	s	t	u	w	x	y	z
p	0	0	0	0	0	0	1	0	0	0
q	0	0	0	1	1	0	1	0	0	0
r	0	0	0	0	0	1	0	0	1	0
s	1	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	1	1	0
u	0	0	0	0	0	0	0	0	1	0
w	0	0	0	1	0	0	0	0	0	0
x	0	0	0	0	0	0	0	0	0	1
y	0	1	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	1	0	0

É interessante observarmos, no quadro 1, o que ocorre com o nó y. Ainda que existam quatro arestas conectadas a esse nó, apenas uma aresta tem origem no nó y e destino em outro nó, o nó q. Assim, a linha correspondente ao nó y apresenta apenas um único 1, na coluna do nó q.

Veja, também, que a matriz M e o quadro 1 têm grande quantidade de zeros em várias posições. Em algumas representações, é comum omitirmos esses zeros e mostrarmos somente as posições com 1.

3. Indicações bibliográficas

- CORMEN, T. et al. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Campus-Elsevier, 2012.
- FEOFOLOFF, P. *Busca em profundidade*. Disponível em <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html>. Acesso em 23 out. 2019.
- GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. *Data structures & algorithms in C++*. 2. ed. Hoboken: John Wiley & Sons, 2011.
- SKIENA, S. *The algorithm design manual.* 2. ed. Londres: Springer-Verlag, 2008.

ÍNDICE REMISSIVO

Questão 1	Estruturas de dados. Árvores AVL. Balanceamento de árvores.
Questão 2	Algoritmos de ordenação. Algoritmo *insertion sort*. Linguagem de Programação C.
Questão 3	Complexidade de algoritmos. Recursão. Linguagem de Programação C.
Questão 4	Complexidade de algoritmos. Recursão. Algoritmos de ordenação.
Questão 5	Complexidade de algoritmos. Estruturas de dados. Linguagem de Programação C.
Questão 6	Teoria de compiladores. Gramáticas formais. Ambiguidade em gramáticas formais.
Questão 7	Teoria da computação. Tipos de relações. Relações de equivalência.
Questão 8	Algoritmos Gulosos. Problemas de otimização. Programação.
Questão 9	Teoria dos grafos. Algoritmo de Dijkstra. Algoritmo de Kruskal.
Questão 10	Teoria dos grafos. Busca em profundidade. Matriz de adjacência.