Universidade Federal de Minas Gerais
departamento de ciência da computação
Disciplina: Algoritmos e Estrutura de Dados II (DCC004)

Professor: Renato Martins (renato.martins AT dcc.ufmg.br)
https://www.dcc.ufmg.br/~renato.martins/courses/DCC004
2° Semestre de 2018

Lista 2 – Listas, árvores binárias, TADs específicos, herança e polimorfismo

Data de publicação: 28/09/2018
Data de Entrega: 26/10/2018 (via email e com demonstração em aula)

Importante: A linguagem para implementação dos exercícios deve ser C++. Crie um projeto para cada exercício, que deve conter um programa principal que teste a sua solução (e o seu respectivo Makefile para compilação). A pontuação de cada item é indicada entre colchetes antes da questão, totalizando NP = 12 pontos. Você pode resolver todos os exercícios (recomendado) ou escolher um subconjunto de questões que totalizem 10 pontos. A nota final NL será:

$$NL = \begin{cases} \min(NP, 10), & \text{se } NP \geq 10 \\ NP, & \text{caso contrário.} \end{cases}$$

1. **[1pt]** O que será impresso pelos programas em C++ a seguir?

   (a)

```
// Arquivo "src/ClassA.h"
#ifndef _DCC004_CLASS_A
#define _DCC004_CLASS_A
#include <iostream>

class A
{
private:
    int m_n;
public:
    A();
    A(int);
    A(const A&);
};
#endif

// Arquivo "src/ClassA.cpp"
#include "ClassA.h"

A::A()
{
}
A::A(int n = 0)
    : m_n(n)
{
    std::cout << 'd';
}
A::A(const A& a)
    : m_n(a.m_n)
{
    std::cout << 'c';
}
```

```
// Arquivo "main.cpp"
#include <iostream>
#include "src/ClassA.h"

void f(const A &a1, const A &
    a2 = A())
{
}
```

```
8
9  int main()
10 {
11     A a(2), b;
12     const A c(a), &d = c, e =
           b;
13     b = d;
14     A *p = new A(c), *q = &a;
15     static_cast<void>(q);
16     delete p;
17     f(3);
18     std::cout << std::endl;
19     return 0;
20 }
```

(b)

```
1  // Arquivo "src/ClassA.h"
2  #ifndef _DCC004_CLASS_A
3  #define _DCC004_CLASS_A
4  #include <iostream>
5
6  class A {
7  public:
8  int x;
9  };
10 #endif
11
12 // Arquivo "src/ClassB1.h"
13 #ifndef _DCC004_CLASS_B1
14 #define _DCC004_CLASS_B1
15 #include "ClassA.h"
16
17 class B1: public A {
18 public:
19 void f();
20 };
21 #endif
22
23 // Arquivo "src/ClassB1.cpp"
24 #include "ClassB1.h"
25
26 void B1::f() {
27 std::cout << x << std::endl;
28 }
29
30 // Arquivo "src/ClassB2.h"
31 #ifndef _DCC004_CLASS_B2
32 #define _DCC004_CLASS_B2
33 #include "ClassA.h"
34
35 class B2: public A {
36 public:
37 void f();
38 };
39 #endif
40
41 // Arquivo "src/ClassB2.cpp"
42 #include "ClassB2.h"
43
44 void B2::f() {
45 std::cout << x << std::endl;
46 }
```

```
1  // Arquivo "main.cpp"
2  #include <iostream>
3  #include "src/ClassA.h"
4  #include "src/ClassB1.h"
5  #include "src/ClassB2.h"
6
7  int main()
8  {
9       B b;
10      B *bp = &b;
11      A *ap = &b;
12      ap->f();
13      ap->g();
14      b.f();
15      b.g();
16      return 0;
17 }
```

2. **[0.5pt]** Implemente um código para encontrar o k-ésimo elemento de uma lista encadeada.

3. **[0.5pt]** Implemente um código para remover duplicatas de uma lista encadeada não ordenada.

4. **[0.5pt]** Escreve um código para particionar uma lista encadeada em volta de um valor `x`, tal que todos os nós menores que `x` venham antes de todos os nós maiores que ou iguais à `x`. Se `x` estiver contido dentro da lista, os valores de `x` só precisam vir

depois dos elementos menores do que `x`. O elemento `x` pode aparecer em qualquer posição na partição direita, ele não precisa aparecer entre as partições esquerda e direita.

5. [**1pt**] Implemente uma função para checar se uma lista duplamente encadeada é um palíndromo.

6. [**1pt**] Implemente uma função para checar se uma árvore binária é uma árvore binária de pesquisa.

7. [**1pt**] Dado uma lista ordenada de inteiros distintos, escreva um algoritmo para criar uma árvore binária de pesquisa com altura mínima.

8. [**0.5pt**] Explique os principais conceitos da Programação Orientada a Objetos: encapsulamento, herança, composição e polimorfismo. Dê um exemplo de código para cada um deles em C++.

9. [**1pt**] Desenhe uma hierarquia de herança para alunos universitários. Utilize Aluno como a classe básica da hierarquia, então inclua as classes AlunoDeGraduação e AlunoGraduado que derivam de Aluno. Continue a estender a hierarquia o mais profundamente (isto é, com muitos níveis) possível. Por exemplo, Primeiranistas, Segundanistas, Terceiranistas e Quartanistas poderiam derivar de AlunoDeGraduação; e AlunoDeDoutorado e AlunoDeMestrado poderiam derivar de AlunoGraduado. Depois de desenhar a hierarquia, discuta os relacionamentos entre as classes.

10. [**1pt**] Os serviços de correio expresso, como FedEx, DHLe UPS, oferecem várias opções de entrega, cada qual com custos específicos. Crie uma hierarquia de herança para representar vários tipos de pacotes. Utilize `Package` como a classe básica da hierarquia, então inclua as classes `TwoDayPackage` e `OvernightPackage` que derivam de `Package`. A classe básica `Package` deve incluir membros de dados que representam nome, endereço, cidade, estado e CEP tanto do remetente como do destinatário do pacote, além dos membros de dados que armazenam o peso (em quilos) e o custo por quilo para a entrega do pacote. O construtor `Package` deve inicializar esses membros de dados. Assegure que o peso e o custo por quilo contenham valores positivos. `Package` deve fornecer uma função-membro `public calculateCost` que retorna um `double` indicando o custo associado com a entrega do pacote. A função `calculateCost` de `Package` deve determinar o custo multiplicando o peso pelo custo (em quilos). A classe derivada `TwoDayPackage` deve herdar a funcionalidade da classe básica `Package`, mas também incluir um membro de dados que representa uma taxa fixa que a empresa de entrega cobra pelo serviço de entrega de dois dias. O construtor `TwoDayPackage` deve receber um valor para inicializar esse membro de dados. `TwoDayPackage` deve redefinir a função-membro `calculateCost` para que ela calcule o custo de entrega adicionando a taxa fixa ao custo baseado em peso calculado pela função `calculateCost` da classe básica Package. A classe `OvernightPackage` deve herdar diretamente da classe `Package` e conter um membro de dados adicional para representar uma taxa adicional por quilo cobrado pelo serviço de entrega noturno. `OvernightPackage` deve redefinir a função-membro `calculateCost` para que ela acrescente a taxa adicional por quilo ao custo-padrão

por quilo antes de calcular o custo da entrega. Escreva um programa de teste que cria objetos de todos os tipos de `Package` e testa a função-membro `calculateCost`.

11. [**1pt**] Use a hierarquia de herança `Package` criada no exercício anterior para criar um programa que exibe asinformações de endereço e calcula os custos de entrega de vários `Package`s. O programa deve conter um vector de ponteiros `Package` para objetos das classes `TwoDayPackage` e `OvernightPackage`. Faça um loop pelo `vector` para processar o `Package`s polimorficamente. Para cada `Package`, invoque as funções `get` para obter as informações de endereço do remetente e do destinatário, e então imprima os dois endereços da maneira que apareceriam nos pacotes de correio. Além disso, chame a função-membro `calculateCost` de cada `Package` e imprima o resultado. Monitore o custo de entrega total de todos os `Package`s no `vector` e exiba esse total quando o loop terminar.

12. [**1pt**] O mundo das formas é muito rico. Anote todas as formas que puder imaginar — bidimensionais e tridimensionais — e as forme em uma hierarquia `Forma` com o maior número de níveis possível que imaginar. Sua hierarquia deve ter a classe básica `Forma` a partir da qual a classe `FormaBiDimensional` e a `FormaTriDimensional` são derivadas. Implemente a hierarquia `Forma` projetada anteriormente. Cada `FormaBidimensional` deve conter a função `obterArea` para calcular a área da forma bidimensional. Cada `FormaTridimensional` deve ter funções-membro `obterArea` e `obterVolume` para calcular a área do volume e da superfície, respectivamente, da forma tridimensional. Crie um programa que utilize um vector de ponteiros `Forma` para objetos de cada classe concreta na hierarquia. O programa deve imprimir o objeto para o qual cada elemento `vector` aponta. Além disso, no loop que processa todas as formas no `vector`, determine se cada forma é uma `FormaBidimensional` ou `FormaTridimensional`. Se uma forma for uma `FormaBidimensional`, exiba sua área. Se uma forma for uma `FormaTridimensional`, exiba sua área e volume.

13. [**2pt**] Crie uma hierarquia de herança que um banco possa utilizar para representar as contas bancárias dos clientes. Todos os clientes nesse banco podem depositar (isto é, creditar) dinheiro em suas contas e retirar (isto é, debitar) o dinheiro delas. Há também tipos mais específicos de contas. As contas de poupança, por exemplo, recebem juros pelo dinheiro depositado nelas. As contas bancárias, por outro lado, cobram uma taxa por transação (isto é, crédito ou débito).

Crie uma hierarquia de herança contendo classe básica `Account` e classes derivadas `SavingsAccount` e `CheckingAccount` que herdam da classe `Account`. A classe básica `Account` deve incluir um membro de dados do tipo `double` para representar o saldo da conta. A classe deve fornecer um construtor que recebe um saldo inicial e o utiliza para inicializar o membro de dados. O construtor deve validar o saldo inicial para assegurar que ele é maior que ou igual a 0.0. Caso contrário, o saldo deve ser configurado como 0.0 e o construtor deve exibir uma mensagem de erro, indicando que o saldo inicial era inválido. A classe deve fornecer três funções-membro. A função-membro `credit` deve adicionar uma quantia ao saldo atual. A função-membro `debit` deve retirar dinheiro de `Account` e assegurar que o valor do débito não exceda o saldo de `Account`. Se exceder, o saldo deve permanecer inalterado e

a função deve imprimir a mensagem "Debit amount exceeded account balance" [Saldo insuficiente]. A função-membro `getBalance` deve retornar o saldo atual.

A classe derivada `SavingsAccount` deve herdar a funcionalidade de uma `Account`, mas também incluir um membro de dados do tipo `double` para indicar a taxa de juros (porcentagem) atribuída à `Account`. O construtor `SavingsAccount` deve receber o saldo inicial, bem como um valor inicial para a taxa de juros de `SavingsAccount`. `SavingsAccount` deve fornecer uma função-membro `public calculateInterest` que retorna um `double` para indicar os juros auferidos por uma conta. A função-membro `calculateInterest` deve determinar esse valor multiplicando a taxa de juros pelo saldo da conta. Nota: `SavingsAccount` deve herdar as funções-membro `credit` e `debit` exatamente como são sem redefini-las. A classe derivada `CheckingAccount` deve herdar da classe básica `Account` e incluir um membro adicional de dados do tipo `double` que representa a taxa cobrada por transação. O construtor `CheckingAccount` deve receber o saldo inicial, bem como um parâmetro que indica o valor de uma taxa. A classe `CheckingAccount` deve redefinir as funções-membro `credit` e `debit` para que subtraiam a taxa do saldo da conta sempre que qualquer uma das transações for realizada com sucesso. As versões `CheckingAccount` dessas funções devem invocar a versão `Account` da classe básica para realizar as atualizações de saldo de uma conta. A função `debit` de `CheckingAccount` deve cobrar uma taxa somente se o dinheiro for realmente retirado (isto é, o valor do débito não exceder ao do saldo da conta). Dica: Defina a função `debit` de `Account` para que ela retorne um `bool` indicando se houve retirada de dinheiro. Em seguida, utilize o valor de retorno para determinar se uma taxa deve ser cobrada.

Depois de definir as classes nessa hierarquia, escreva um programa que cria objetos de cada classe e testa suas funções-membro. Adicione os juros ao objeto `SavingsAccount` invocando primeiro sua função `calculateInterest` e, então, passando o valor retornado dos juros para a função `credit` do objeto.