

Trabalho Prático - Extrato

Thiago Pedro Ferreira de Moraes

Junho, 2023

1 Introdução

Este trabalho visa em aprofundar os conhecimentos sobre os tipos abstratos de dados em um problema sobre extratos bancários. Para isso, é preciso fazer um "mini sistema" que recebe dados da conta e operações realizadas, produzindo uma saída que é um extrato completo de todas as contas.

2 Método e análise

Em primeiro lugar, foi utilizado, neste trabalho, **listas encadeadas simples sem sentinela**. Criando uma lista de "Conta", cada nodo da lista passa a ter um número de conta (que serve como um "ID") e **uma outra lista, de mesmo tipo (simples sem sentinela), dentro**.

```
public class Conta {  
  
    private int numero_conta;  
    private ListaEncadeada<Extrato> extrato;  
}
```

Figura 1: Atributos de Conta.

A lista mais interna, "extrato" (do tipo "Extrato"), carrega as informações relevantes de cada operação bancária que um usuário faz (operação realizada e valor). Sendo assim, cada **conta possui múltiplas operações**, justificando o uso de uma lista encadeada interna.

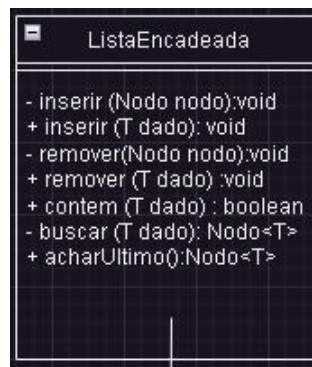
```
public class Extrato {  
    private int operacao;  
    private int valor;  
}
```

Figura 2: Atributos de Extrato.

A partir disso, dispondo da lista encadeada implementada, foi possível armazenar todos os dados em listas, para depois produzir uma saída.

2.1 A Lista encadeada

A lista encadeada é composta por Nodos. Além disso, dispõe de diversos métodos, tais como inserção e remoção. A figura a seguir mostra todos eles. Os *getters* e *setters* foram omitidos.



```
ListaEncadeada  
  
- inserir (Nodo nodo):void  
+ inserir (T dado): void  
- remover(Nodo nodo):void  
+ remover (T dado) :void  
+ contem (T dado) : boolean  
- buscar (T dado): Nodo<T>  
+ acharUltimo():Nodo<T>
```

Figura 3: Métodos da ListaEncadeada

2.1.1 O método público inserir

Este método cria um novo nodo genérico (T). Passa então, para a classe privada "inserir", como argumento.

2.1.2 O método privado inserir

Na lista criada, a ideia é o novo nodo apontar para *null*. Caso a lista não tenha nodos ainda, o primeiro nodo aponta para *null*. Assim, o primeiro aponta para o segundo, o segundo para o terceiro, seguindo esse padrão para n nodos (n-1 aponta para n). Essa abordagem facilita imprimir as operações e seus respectivos valores de modo temporal, como foi exigido pelo problema.

O tempo de execução desta função é linear, não sendo constante por chamar a função "achaUltimo", a qual possui um *while*.

Dentro desta função, duas variáveis são criadas (cabecaAtual e ultimoAtual), consumindo memória. O restante das variáveis são apenas redefinidas, não havendo mais gasto de memória.

```
private void inserir(Nodo nodo) {
    if (cabeca == null) { //c1
        Nodo cabecaAtual = cabeca; // c2
        cabeca = nodo; // c3
        nodo.prox = cabecaAtual; // c4 -> c1+c2+c3+c4 = A
    } else { //c5
        Nodo ultimoAtual = acharUltimo(); // X = an + b
        ultimoAtual.prox = nodo; // c6
        nodo.prox = null; // c7 -> c4+c5+c6+c7 = B
    } // no total: A*0+B+X,
} // Tempo linear
```

Figura 4: Análise do método inserir

2.1.3 Os métodos buscar e contém

O método "contem" chama o método buscar, passando como argumento o dado que recebeu, comparando com null. Se for *null*, retorna *false*.

Em buscar, percorre-se a lista começando da cabeça. Utilizando-se do método equals do tipo T da lista, compara-se se o nodo de cada iteração é igual ao nodo passado como argumento. Caso não ache, é retornado *null* para "contem", caso contrário, o nodo achado é retornado.

Desse modo, essa abordagem requer que cada classe nova (como Conta e Valores) instanciada como tipo de dado da coleção (neste caso, lista encadeada) tenha um método equals.

O tempo de execução deste método é linear. Isso ocorre por existir apenas um *while* dentro do método, não havendo métodos com estrutura de repetição aninhadas.

Além disso, o consumo de memória se dá por uma única variável auxiliar, a única a ser declarada aqui. As demais tem apenas seu valor atualizado.

```

private Nodo<T> buscar(T dado){
    Nodo<T> aux = cabeca;           // c1
    while(aux != null){             // c2.n
        if(aux.dado.equals(obj:dado)){ // c3. (n-1)
            return aux;              // c4 . 0
        }
        aux = aux.prox;              // c5
    }

    return null;                    // c6
}                                   // c1+c2*n+(c3*(n-1)) + c5 = 2 + n
                                   // Tempo Linear

```

Figura 5: Análise do método buscar

2.1.4 Método público remover

Este método utiliza o método buscar para encontrar o nodo, passando este para a sua versão privada.

2.1.5 Método privado remover

Este método remove um nodo alterando o valor da referência do elemento anterior para o próximo nodo.

A respeito de seu tempo de execução, este se dá de forma linear, havendo apenas um *while* e nenhuma função aninhada dentro. Aqui, um nodo é declarado. Dessa forma, duas variáveis são criadas, *nodo.dado* e *nodo.prox*.

```

private void remover(Nodo nodo){
    if(cabeca == null) throw new IllegalStateException(s:"underflow");

    if (nodo == cabeca){           // c1 * 0
        cabeca = cabeca.prox;     // c2 * 0
        return;                   // c3 * 0
    }

    // procura o no anterior
    Nodo ant = cabeca;             // c4
    while(ant.prox != null && ant.prox != nodo){ // c5 * n
        ant = ant.prox;           // c6 * (n-1)
    }

    if (ant.prox != nodo) throw new IllegalStateException(s:"nodo nao existe"); // 0

    ant.prox = nodo.prox; // c7
    nodo.prox = null;      // c8          c5*n + c6 *(n-1) + c7 + c8 = 2n +1
                                   // Tempo Linear
}

```

Figura 6: Análise do método remover

2.1.6 Método acharUltimo

De maneira objetiva, este método acha o elemento da lista que aponta para *null*, o último elemento, portanto.

Este método também é linear. Não há estruturas de repetição aninhadas.

```
public Nodo<T> acharUltimo(){
    Nodo ultimo = cabeca;           // c1
    while(ultimo.prox != null){      // c2*n
        ultimo = ultimo.prox;       // c3*(n-1)
    }
    return ultimo;                   // c4          Tempo Linear : 2n + 1
}
```

Figura 7: Análise do método acharUltimo

Como há um nodo sendo declarado, este gasta, na memória, duas variáveis.

2.2 A classe "HelperClass"

Aqui, são agregados todos métodos úteis. Alguns de seus métodos são essenciais para o povoamento da lista e a criação de uma saída, com o formato de um extrato. A seguir estão os métodos da lista e uma breve descrição dos principais.

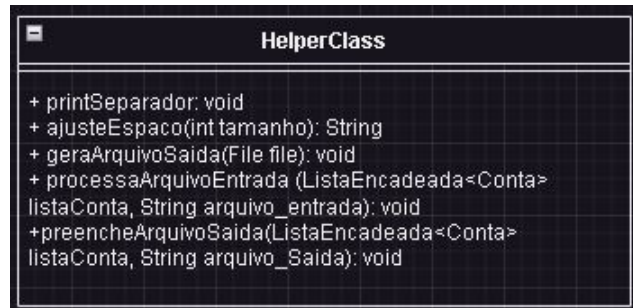


Figura 8: Métodos de HelperClass

2.2.1 O método "processaArquivoEntrada"

Este método captura o arquivo de entrada - cujo nome é especificado pelo usuário - e o lê, linha por linha. Desse modo, é possível atribuir, dentro deste método, os valores relacionados às classes Conta e Extrato. Após isso, é inserido na lista a classe Conta.

O tempo de execução deste método não é linear. Aqui, por haver estruturas de repetição aninhadas, o tempo passa a ser exponencial. A chamada do método inserir transforma torna a função cúbica, sem esse, seria apenas quadrática.

Este método declara várias variáveis, consumindo muita memória. Uma forma de evitar mais gasto de memória é declarar as variáveis fora do *while* presente, evitando mais inicializações.

2.2.2 O método "preencheArquivoSaida"

Por fim, tem-se um método muito importante. Este método captura os dados presentes na lista, iterando todos os nodos através de um ***foreach***, imprimindo-os em um arquivo externo. Essa estrutura de repetição só é executável aqui graças ao Iterator previamente implementado.

Da mesma forma que o método anterior, o pior caso gasta um tempo cúbico. O gasto de memória também se dá do mesmo modo que o anterior.

2.3 Esquema UML do projeto

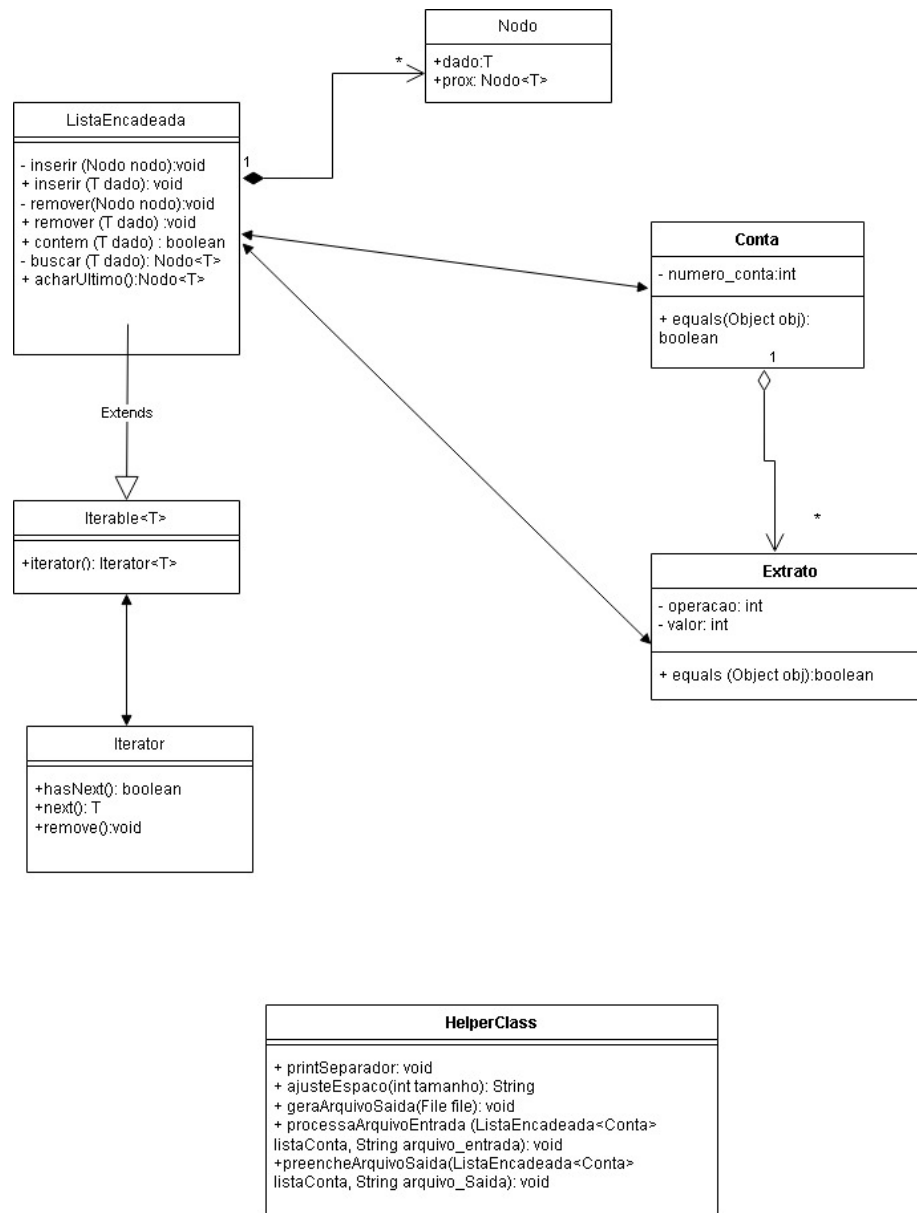


Figura 9: Esquema UML do sistema

3 Conclusão

Neste trabalho foi feito um pequeno sistema de exportação de extrato bancário, utilizando um tipo abstrato de dados, uma lista encadeada simples sem sentinela.

Assim, foi possível aprender mais profundamente sobre a implementação de listas encadeadas, bem como o uso de APIs. A implementação em Java proporcionou um maior entendimento da Orientação a Objetos, graças, também, à implementação de classes não nativas.

Um outro aspecto importante foi a liberdade de escolha quanto ao tipo de coleção. Isso instigou a refletir qual seria o melhor tipo de dado abstrato a ser utilizado: pilhas, filas ou listas.