FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Algoritmos e Estruturas de dados

Ano Lectivo 2012/2013

RELATÓRIO DE PROJECTO:

SchoolCARD – Cartão electrónico da escola

Docente: Prof. Vasco Amaral

Turno: P2

Autores:

Hugo António, nº 34334

Ricardo Gaspar, nº 35277

ÍNDICE

TIPOS ABSTRACTOS DE DADOS

Card e CardWritable

As interfaces **Card** e **CardWritable** representam o cartão.

Card contém todas as operações relativas ao cartão que não modificam os seus atributos. Esta interface só possui métodos de consulta, por forma a proteger o código, uma vez que este tipo de objectos é devolvido para o programa principal (main).

```
public interface Card {

	/**
	 * Devolve o código do cartão.
	 *
	 * @return Código do cartão.
	 */
	int getCode();

	/**
	 * Devolve o nome do aluno.
	 *
	 * @return Nome do aluno.
	 */
	String getName();

	/**
	 * Devolve o saldo do cartão.
	 *
	 * @return Saldo do cartão.
	 */
	int getBalance();

	/**
	 * Devolve a turma à qual o aluno pertence.
	 *
	 * @return Turma do aluno.
	 */
	String getStudentClass();

	/**
	 * Devolve a morada do aluno.
	 *
	 * @return Morada do aluno.
	 */
	String getAddress();

	/**
	 * Devolve o contacto do aluno.
	 *
	 * @return Contacto do aluno.
	 */
	String getContact();

	/**
	 * Devolve um produto favorito existente indicando o seu nome.
	 *
	 * @param productName
	 *            Nome do produto a devolver.
	 * @return Produto favorito com o nome indicado.
	 * @throws CardWithoutFavouritesException
	 *             Caso o cartão não tenha produtos favoritos associados.
	 * @throws FavouriteProductInexistenceException
	 *                              Caso o cartão não tenha o produto favorito com o nome indicado
	 */
	Product getFavouriteProduct(String productName)
```

```
        /**
         * Devolve os movimentos realizados com o cartão.
         *
         * @return Iterador de movimentos realizados com o cartão.
         * @throws CardWithoutTransactionsException
         *             Caso não existam movimentos realizados pelo cartão.
         */
        Iterator<Transaction> getTransactions()
                        throws CardWithoutTransactionsException;

        /**
         * Devolve todos os produtos favoritos associados ao cartão.
         *
         * @return Conjunto de entradas na árvore dos produtos favoritos existentes no cartão.
         * @throws CardWithoutFavouritesException
         *             Caso o cartão não tenha produtos favoritos associados.
         */
        Set<Map.Entry<String, Product>> getFavourites() throws CardWithoutFavouritesException;
}
```

CardWritable contém as operações relativas ao cartão que modificam os seus atributos.

```
public interface CardWritable extends Card {

	/**
	 * Recarrega o cartão.
	 *
	 * @param amount
	 *            Valor a adicionar ao saldo.
	 */
	void increaseBalance(int amount);


	/**
	 * Adiciona um movimento ao cartão.
	 *
	 * @param product
	 *            Produto.
	 * @param totalItems
	 *            Número de itens.
	 */
	void addTransaction(Product product, int totalItems);

	/**
	 * Adiciona um produto aos favoritos do cartão.
	 *
	 * @param p
	 *            Produto a adicionar.
	 * @throws FavouriteProductAlreadyExistsException
	 *             Caso o produto indicado já exista nos favoritos do cartão.
	 */
	void addFavourite(Product product) throws FavouriteProductAlreadyExistsException;

	/**
	 * Remove um produto dos favoritos do cartão.
	 *
	 * @param productName
	 *            Nome do produto.
	 * @throws FavouriteProductInexistenceException
	 *             Caso o produto indicado não exista nos favoritos do cartão.
	 */
	void removeFavourite(String productName)
			throws FavouriteProductInexistenceException;

	/**
	 * Remove todos os movimentos do cartão.
	 */
	void clearTransactions();
}
```

A estrutura de dados escolhida para armazenar um conjunto de cartões foi um dicionário, implementado utilizando uma tabela de dispersão aberta, devido ao facto de permitir pesquisas (por chave), inserções e remoções com complexidade temporal constante (no caso esperado).

A estrutura de dados escolhida para armazenar os produtos favoritos associados a um cartão foi um dicionário ordenado, implementado por uma árvore AVL. Esta é utilizada para permitir o armazenamento dos produtos favoritos ordenados por ordem alfabética, bem como a sua listagem.

A sua inserção, remoção e pesquisa têm complexidade temporal (1.44 log n).

Product e ProductWritable

As interfaces **Product e ProductWritable** representam o produto.

Product contém todas as operações relativas ao produto que não modificam os seus atributos. Esta interface só possui métodos de consulta, por forma a proteger o código, uma vez que este tipo de objectos é devolvido para o programa principal (main).

```
public interface Product {

/**
* Devolve o código do produto.
*
* @return Código do produto.
*/
int getCode();

/**
* Devolve o nome do produto.
*
* @return Nome do produto.
*/
String getName();

/**
* Devolve o preço do produto.
*
* @return Preço do produto.
*/
int getPrice();

/**
* Devolve o total monetário de vendas.
*
* @return Valor total monetário de vendas.
*/
int getTotalSales();
}
```

ProductWritable contém as operações relativas ao cartão que modificam os seus atributos.

```
public interface ProductWritable extends Product{

        /**
         * Adiciona um valor ao total monetário de vendas do produto.
         *
         * @param totalItems
         *            Numero de items a adicionar ao total monetário de vendas.
         */
        void increaseTotalSales(int totalItems);

        /**
         * Inicializa o valor total monetário de vendas do produto.
         */
        void clearTotalSales();

}
```

A estrutura de dados escolhida para armazenar um conjunto de produtos foi um dicionário, implementado utilizando uma tabela de dispersão aberta, devido ao facto de permitir pesquisas (por chave), inserções e remoções com complexidade temporal constante(no caso esperado). Além do dicionário, foi utilizada um dicionário ordenado implementado utilizando uma árvore AVL. Esta é utilizada para as listagens por ordem alfabética dos produtos.

Transaction

A interface **Transaction** representa um movimento.

```
public interface Transaction {

	/**
	 * Devolve o código do produto.
	 *
	 * @return Código do produto.
	 */
	int getProductCode();

	/**
	 * Devolve o número total de itens do produto.
	 *
	 * @return Número total de intens do produto.
	 */
	int getTotalItems();

	/**
	 * Devolve o custo total do movimento.
	 *
	 * @return Custo total do movimento.
	 */
	int getTotalCost();
}
```

A estrutura de dados escolhida para armazenar um conjunto de movimentos foi uma lista, implementada utilizando uma lista duplamente ligada, devido ao facto de ser iterável. Esta é utilizada segundo uma disciplina de pilha de forma a manter a ordem de inserção. Assim sendo, a complexidade da inserção é constante.

SchoolSystem

A interface **SchoolSystem** representa o Sistema de Gestão de Cartões Eletrónicos escolares. Esta contém todas as operações possíveis de realizar no sitema.

```
public interface SchoolSystem {

	/**
	 * Insere um novo cartão no sistema.
	 *
	 * @param cardCode
	 *            Código do cartão.
	 * @param studentName
	 *            Nome do aluno.
	 * @param balance
	 *            Saldo inicial do cartão.
	 * @param studentClass
	 *            Turma do aluno.
	 * @param address
	 *            Morada do aluno.
	 * @param contact
	 *            Contacto do aluno.
	 * @throws CardAlreadyExistsException
	 *             Caso já exista um cartão com o mesmo código.
	 */
	void insertCard(int cardCode, String studentName, int balance,
			String studentClass, String address, String contact)
			throws CardAlreadyExistsException;

	/**
	 * Remove um cartão do sistema.
	 *
	 * @param cardCode
	 *            Código do cartão.
	 * @throws CardInexistenceException
	 *             Caso não exista nenhum cartão com o código indicado.
	 */
	void removeCard(int cardCode) throws CardInexistenceException;

	/**
	 * Devolve um cartão existente no sistema.
	 *
	 * @param cardCode
	 *            Código do cartão.
	 * @return Cartão com o código indicado.
	 * @throws CardInexistenceException
	 *             Caso não exista nenhum cartão com o código indicado.
	 */
	Card getCard(int cardCode) throws CardInexistenceException;

	/**
	 * Carrega um cartão dado o seu código e o valor a carregar.
	 *
	 * @param cardCode
	 *            Código do cartão.
	 * @param amount
	 *            Valor a carregar.
	 * @throws CardInexistenceException
	 *             Caso não exista nenhum cartão com o código indicado.
	 */
	void increaseCardBalance(int cardCode, int amount)
			throws CardInexistenceException;
```

```java
/**
         * Insere um produto no sistema.
         *
         * @param productCode
         *            Código do produto.
         * @param productName
         *            Nome do produto.
         * @param productPrice
         *            Preço do produto.
         * @throws ProductAlreadyExistsException
         *             Caso já exista um produto no sistema com o mesmo código.
         * @throws ProductNameAlreadyExistsException
         *             Caso já exista um produto no sistema com o mesmo nome.
         */
        void insertProduct(int productCode, String productName, int productPrice)
                        throws ProductAlreadyExistsException,
                        ProductNameAlreadyExistsException;

        /**
         * Devolve um produto existente no sistema.
         *
         * @param productCode
         *            Código do produto.
         * @return Produto com o código indicado.
         * @throws ProductInexistenceException
         *             Caso o produto com o código indicado não exista.
         */
        Product getProduct(int productCode) throws ProductInexistenceException;

        /**
         * Compra de um produto.
         *
         * @param cardCode
         *            Código do cartão.
         * @param productCode
         *            Código do produto.
         * @param totalItems
         *            Número de itens a comprar do produto indicado.
         * @throws CardInexistenceException
         *             Caso o cartão com o código indicado não exista.
         * @throws ProductInexistenceException
         *             Caso o produto com o código indicado não exista.
         * @throws InsufficientBalanceException
         *             Caso o saldo do cartão indicado não tenha saldo insuficiente
         *             para a compra a realizar.
         */
        void purchaseProduct(int cardCode, int productCode, int totalItems)
                        throws CardInexistenceException, ProductInexistenceException,
                        InsufficientBalanceException;

        /**
         * Adiciona um produto aos favoritos de um cartão.
         *
         * @param cardCode
         *            Código do cartão.
         * @param productCode
         *            Código do produto.
         * @throws CardInexistenceException
         *             Caso o cartão com o código indicado não exista.
         * @throws ProductInexistenceException
         *             Caso o produto com o código indicado não exista.
         * @throws FavouriteProductAlreadyExistsException
         *             Caso o produto já exista nos favoritos do cartão.
         */
        void addFavouriteProduct(int cardCode, int productCode)
                        throws CardInexistenceException, ProductInexistenceException,
                        FavouriteProductAlreadyExistsException;
```

```
/**
         * Compra de um produto favorito existente no cartão.
         *
         * @param cardCode
         *            Código do cartão.
         * @param productName
         *            Nome do produto.
         * @param totalItems
         *            Número de itens a comprar do produto indicado.
         * @throws CardInexistenceException
         *             Caso o cartão com o código indicado não exista.
         * @throws FavouriteProductInexistenceException
         *             Caso o produto não exista nos favoritos do cartão.
         * @throws InsufficientBalanceException
         *             Caso o saldo do cartão indicado não tenha saldo insuficiente
         *             para a compra a realizar.
         */
        void purchaseFavouriteProduct(int cardCode, String productName,
                        int totalItems) throws CardInexistenceException,
                        FavouriteProductInexistenceException, InsufficientBalanceException;

        /**
         * Remove um produto existente nos favoritos de um cartão.
         *
         * @param cardCode
         *            Código do cartão.
         * @param productName
         *            Nome do produto.
         * @throws CardInexistenceException
         *             Caso o cartão com o código indicado não exista.
         * @throws FavouriteProductInexistenceException
         *             Caso o produto não exista nos favoritos do cartão.
         */
        void removeFavouriteProduct(int cardCode, String productName)
                        throws CardInexistenceException,
                        FavouriteProductInexistenceException;

        /**
         * Lista os produtos favoritos do cartão indicado.
         *
         * @param cardCode
         *            Código do cartão.
         * @return Iterador de entries de produtos favoritos existentes no cartão.
         * @throws CardInexistenceException
         *             Caso o cartão com o código indicado não exista.
         * @throws CardWithoutFavouritesException
         *             Caso o cartão não tenha produtos favoritos.
         */
        Iterator<Entry<String, Product>> listFavouriteProducts(int cardCode)
                        throws CardInexistenceException, CardWithoutFavouritesException;

        /**
         * Lista todos os movimentos do cartão indicado.
         *
         * @param cardCode
         *            Código do cartão.
         * @return Iterador dos movimentos realizados com o cartão indicado.
         * @throws CardInexistenceException
         *             Caso o cartão com o código indicado não exista.
         * @throws CardWithoutTransactionsException
         *             Caso o cartão não tenha produtos favoritos.
         */
        Iterator<Transaction> listTransactions(int cardCode)
                        throws CardInexistenceException, CardWithoutTransactionsException;

        /**
         * Muda de mês do sistema. Faz com que o valor de vendas de todos os
         * produtos seja inicializado e os movimentos de todos os cartões existentes
         * sejam removidos.
         */
        void changeMonth();
```

```
/**
         * Lista todos os produtos existentes no sistema.
         *
         * @return Iterador de entries de produtos existentes no sistema.
         * @throws ProductInexistenceException
         *             Caso não existam produtos no sistema.
         */
        Iterator<Entry<String, Product>> listProducts() throws ProductInexistenceException;
}
```

ESTUDO DAS COMPLEXIDADES

Complexidade temporal

Nesta secção assume-se que todas as operações descritas nas tabelas ocorrem com sucesso.

Legenda:

λc – factor de ocupação da tabela de dispersão dos cartões.

nCartões – número de elementos da tabela de dispersão dos cartões. No pior caso a complexidade é linear devido à possibilidade de realizar o rehash da tabela.

λp – factor de ocupação da tabela de dispersão dos produtos.

nProdutos – número de elementos existentes na estrutura de dados que armazena produtos.

nFavoritos – número de elementos existentes na AVL que armazena produtos favoritos.

nMovimentos - número de elementos existentes na lista que armazena movimentos.

insertCard

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Inserir cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
TOTAL	**O(1)**	**O(nCartões)**	**O(1+λc)**

removeCard

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Remover o cartão da tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
TOTAL	**O(1)**	**O(nCartões)**	**O(1+λc)**

getCard

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
TOTAL	**O(1)**	**O(nCartões)**	**O(1+λc)**

increaseCardBalance

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
TOTAL	**O(1)**	**O(nCartões)**	**O(1+λc)**

insertProduct

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do produto na tabela de dispersão dos produtos	O(1)	O(nProdutos)	O(1+λp)
Verificar a existência do produto na AVL	O(1)	(1.44) log nProdutos	log nProdutos
Inserir o produto na tabela de dispersão dos produtos	O(1)	O(nProdutos)	O(1+λp)
Inserir o produto na AVL	O(1)	O(1.44 log nProdutos)	O(log nProdutos)
TOTAL	**O(1)**	**O(nProdutos)**	**log nProdutos**

getProduct

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do produto na tabela de dispersão dos produtos	O(1)	O(nProdutos)	O(1+λp)
TOTAL	**O(1)**	**O(nProdutos)**	**O(1+λp)**

purchaseProduct

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Verificar a existência do produto na tabela de dispersão dos produtos	O(1)	O(nProdutos)	O(1+λp)
Adicionar transacção ao cartão	O(1)	O(1)	O(1)
TOTAL	**O(1)**	**O(nCartões) ou O(nProdutos)**	**O(1+λc) ou O(1+λp)**

Neste método a complexidade no pior caso e no caso esperado dependem do número de produtos e cartões. Ou seja, aquela que for maior determina a complexidade desta operação.

addFavouriteProduct

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Verificar a existência do produto na tabela de dispersão dos produtos	O(1)	O(nProdutos)	O(1+λp)
Inserir produto favorito na AVL dos favoritos do cartão	O(1)	O(1.44log nFavoritos)	O(log nFavoritos)
TOTAL	**O(1)**	**O(nCartões) ou O(nProdutos)**	**O(log nFavoritos)**

Neste método a complexidade no pior caso dependem do número de produtos e cartões. Ou seja, aquela que for maior determina a complexidade desta operação.

purchaseFavouriteProduct

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Verificar a existência do produto favorito na tabela de dispersão dos produtos	O(1)	O(nProdutos)	O(1+λp)
Adicionar transacção ao cartão	O(1)	O(1)	O(1)
TOTAL	**O(1)**	**O(nCartões) ou O(nProdutos)**	**O(1+λc) ou O(1+λp)**

Neste método a complexidade no pior caso e no caso esperado dependem do número de produtos e cartões. Ou seja, aquela que for maior determina a complexidade desta operação.

removeFavouriteProduct

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Remover produto favorito da AVL dos favoritos do cartão	O(1)	O(1.44log nFavoritos)	O(log nFavoritos)
TOTAL	**O(1)**	**O(nCartões)**	**O(log nFavoritos)**

changeMonth

Acção	Melhor caso	Pior caso	Caso esperado
Remover os movimentos de todos os cartões	O(1)	O(nCartões)	O(nCartões)
Inicializar as vendas de todos os produtos	O(1)	O(nProdutos)	O(nProdutos)
TOTAL	**O(1)**	**O(nCartões) ou O(nProdutos)**	**O(nCartões) ou O(nProdutos)**

Neste método a complexidade no pior caso e no caso esperado dependem do número de produtos e cartões. Ou seja, aquela que for maior determina a complexidade desta operação.

listFavouriteProducts

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Devolver iterador da AVL dos favoritos de um cartão	O(1)	O(1)	O(1)
TOTAL	**O(1)**	**O(nCartões)**	**O(1+λc)**

listTransactions

Acção	Melhor caso	Pior caso	Caso esperado
Verificar a existência do cartão na tabela de dispersão dos cartões	O(1)	O(nCartões)	O(1+λc)
Devolver iterador da lista dos movimentos de um cartão	O(1)	O(1)	O(1)
TOTAL	**O(1)**	**O(nCartões)**	**O(1+λc)**

listProducts

Acção	Melhor caso	Pior caso	Caso esperado
Verificar se a AVL está vazia	O(1)	O(1)	O(1)
Devolver iterador da AVL dos produtos	O(1)	O(1)	O(1)
TOTAL	**O(1)**	**O(1)**	**O(1)**

Complexidade espacial

A complexidade espacial do programa é dada pela seguinte fórmula:

O(dim1) + O(dim2) + O(dim3)

Dim1 – dimensão do vector da tabela de dispersão dos cartões. Sendo que cada cartão tem uma lista de movimentos (com a dimensão de *nMovimentos*) e uma árvore AVL de produtos favoritos (com a dimensão de *nFavoritos*).

Dim2 – dimensão do vector da tabela de dispersão dos produtos.

Dim3 – dimensão da árvore AVL dos produtos

ANEXOS

DIAGRAMA DE CLASSES

CÓDIGO FONTE

ESTRUTURAS IMPLEMENTADAS