Algoritmos e Estruturas de Dados

2013/2014

MIEI

Relatório Fase Final

SocialNet

Realizado Por:

Pedro Trindade - Nº 41661

Paulo Martins - Nº 41982

P3 - Prof. Sofia Cavaco

Índice

*Excepções Omitidas

Introdução

Este projecto representa uma rede social, que gere um conjunto de grupos e de contactos, que por sua vez podem aderir a grupos, criar amizades com outros utilizadores e postar mensagens.

Ao longo deste relatório vamos abordar os **Tipos Abstractos de Dados** utilizados na realização do projecto, as estruturas de dados usadas e as razões da sua escolha, tal como a implementação dos métodos descritos em 3.3. e as suas respectivas complexidades temporais. Será, também, apresentado o diagrama de classes e interfaces do programa.

Na versão de entrega no Moodle vem ainda anexado o código fonte do projecto.

Tipos Abstractos de Dados

SocialNet

Representa a rede social. Gere toda a informação associada à mesma (todos os contactos e grupos registados no sistema). Esta **TAD** implementa directamente os métodos da **Main**, descritos no enunciado do projecto na secção 3.3., chamando, para isso, os métodos das **TADs** *Contact* e *Group*.

Tanto *Contact* como *Group* são representados através de dicionários (implementados pela classe *SepChainHashTable*), pois estes permitem a procura (criação e remoção) de objectos usando uma chave identificadora (neste caso o *login* e o *nome do grupo*, respectivamente) com uma complexidade temporal reduzida.

Contact

Representa um utilizador, ou contacto. Gere toda a informação associada ao próprio utilizador (as variáveis que representam o seu *login*, *nome*, *localidade*, *profissão* e *idade*), tal como o conjunto dos seus amigos (representado por uma *BinarySearchTree*), o conjunto dos grupos a que o contacto aderiu (representado por uma *DoublyLinkedList*) e o conjunto dos posts que o contacto publicou (também representado por uma *DoublyLinkedList*).

Ora, para o conjunto dos amigos do utilizador utilizamos uma *BinarySearchTree* pois esta estrutura de dados facilita a listagem dos amigos por ordem lexicográfica do login, garantindo uma complexidade temporal reduzida nas operações usadas, dado que os nós constituintes da árvore são representados de forma ordenada (os nós da sub-árvore esquerda contém um valor inferior ao nó da raíz, enquanto que os da sub-árvore direita possuem um valor superior). Para esta listagem foi implementado o iterador *BSTInOrderIterator*, que possibilita a iteração dos elementos da árvore.

Já para o conjunto dos grupos a que o contacto aderiu e dos posts que o contacto enviou usamos *DoublyLinkedLists* para representar ambos. A *DoublyLinkedList* é uma classe que implementa a estrutura de dados List, guardando a informação em nós que contém apontadores para o nó que precedente e para o nó seguinte. Esta classe contém, também, um iterador que permite percorrer todos os elementos presentes na lista, tornando-se eficiente na listagem dos posts enviados pelo contacto.

Group

Representa um grupo. Gere toda a informação associada ao próprio grupo (as variáveis que representam o seu *nome* e a sua *descrição*), tal como o conjunto dos membros do grupo (representados por uma *BinarySearchTree*) e o conjunto dos posts publicados no grupo (representado por uma *DoublyLinkedList*).

Tal como a lista de amigos do contacto, a lista de membros do grupo é representada usando uma *BinarySearchTree* pelas razões indicadas posteriormente, e, da mesma forma, a lista dos posts publicados no grupo é representada usando uma *DoublyLinkedList* (exactamente como a lista de posts submetidos por um contacto).

Post

Representa um post. Gere toda a informação associada ao próprio post (as variáveis que representam o seu *título*, o *texto* e o *URL de uma foto* que lhe está associada).

Operações descritas em 3.3. e as suas respectivas *Complexidades Temporais*

• Inserção de um contacto:

1. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

2. Caso não exista, cria o objecto Contact e insere o mesmo na tabela de disperção.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Insert (HT)	O(1)	O(n)	O(1+λ)
Total	O(1)	O(n)	O(1+λ)

• Consulta dos dados de um contacto:

1. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

2. Caso exista, devolve o utilizador encontrado na tabela de disperção.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
Total	O(1)	O(n)	O(1+λ)

• Inserção de amizade:

1. Verifica se ambos os utilizadores com os logins passados por argumento existem na tabela de dispersão.

2. Verifica se um dos utilizadores existe na árvore binária do outro utilizador, ou seja, se existe amizade entre eles.

3. Caso não exista, é inserido o primeiro utilizador na árvore binária do segundo utilizador e vice-versa.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
3. Insert (HT)	O(1)	O(n)	O(1+λ)
3. Insert (HT)	O(1)	O(n)	O(1+λ)
Total	O(1)	O(n)	O(1+λ)

• Remoção de amizade:

1. Verifica se ambos os utilizadores com os logins passados por argumento existem na tabela de dispersão.

2. Verifica se o um dos utilizadores existe na árvore binária do outro utilizador, ou seja, se existe amizade entre eles.

3. Caso exista, é removido o primeiro utilizador da árvore binário do segundo utilizador e vice-versa.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
3. Remove (HT)	O(1)	O(n)	O(1+λ)
3. Remove (HT)	O(1)	O(n)	O(1+λ)
Total	O(1)	O(n)	O(1+λ)

• Inserção de grupo:

1. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

2. Caso não exista, cria o objecto Group e insere o mesmo na tabela de disperção.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Insert (HT)	O(1)	O(n)	O(1+λ)
Total	O(1)	O(n)	O(1+λ)

• Consulta dos dados de um grupo:

1. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

2. Caso exista, devolve o grupo encontrado na tabela de disperção.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
Total	O(1)	O(n)	O(1+λ)

• Remoção de grupo:

1. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

2. Verifica se o grupo tem algum membro, ou seja, se a sua árvore binária de membros não está vazia.

3. Cria e corre o iterador da árvore binaria, chamando a operação de remoção do grupo a eliminar na lista duplamente ligada de grupos do contacto retornado por next().

4. Remove o grupo da tabela de dispersão.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. IsEmpty (BST)	O(1)	O(1)	O(1)
3. Iterator (BST)	O(n)	O(n)	O(n)
4. Remove (HT)	O(1)	O(n)	O(1+λ)
Total	O(n)	O(n)	O(n)

• Inserção de aderente no grupo:

1. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

2. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

3. Verifica se o grupo existe na lista duplamente ligada de grupos no contacto e se o contacto existe na árvore binária de membros no grupo, ou seja, se existe aderência.

4. Caso não exista, insere o grupo na lista duplamente ligada de grupos do contacto e insere o contacto na árvore binária de membros do grupo.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
3. Find (DLL)	O(1)	O(n)	O(n)
3. Find (BST)	O(1)	O(n)	O(lg(n))
4. AddFirst (DLL)	O(1)	O(1)	O(1)
4. Insert (BST)	O(1)	O(n)	O(lg(n))
Total	O(1)	O(n)	O(n)

• Remoção de aderente do grupo:

1. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

2. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

3. Verifica se o grupo existe na lista duplamente ligada de grupos no contacto e se o contacto existe na árvore binária de membros no grupo, ou seja, se existe aderência.

4. Caso exista, remove o grupo da lista duplamente ligada de grupos no contacto e remove o contacto da árvore binária de membros no grupo.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
3. Find (DLL)	O(1)	O(n)	O(n)
3. Find (BST)	O(1)	O(n)	O(lg(n))
4. Remove (DLL)	O(1)	O(n)	O(n)
4. Remove (BST)	O(1)	O(n)	O(lg(n))
Total	O(1)	O(n)	O(n)

• Inserção de post por contacto:

1. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

2. Verifica se a árvore binária dos amigos do utilizador está vazia, ou seja, se o utilizador tem ou não amigos.

3. Caso tenha, é criado e corrido um iterador da árvore binária que chama a operação de inserção de post em cada contacto retornado por next();

4. Verifica se a lista duplamente ligada dos grupos do utilizador está vazia, ou seja, se o utilizador pertence ou não a um grupo.

5. Caso pertença, é criado e corrido um iterador da lista duplamente ligada que chama a operação de inserção de post em cada grupo retornado por next();

6. Por fim, é inserido um post na lista de posts do utilizador.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. IsEmpty (BST)	O(1)	O(1)	O(1)
3. Iterator (BST)	O(n)	O(n)	O(n)
4. IsEmpty (DLL)	O(1)	O(1)	O(1)
5. Iterator (DLL)	O(1)	O(n)	O(n)
6. AddFirst (DLL)	O(1)	O(1)	O(1)
Total	O(n)	O(n)	O(n)

• Listagem de amigos de um contacto:

1. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

2. Verifica se a árvore binária dos amigos do utilizador está vazia, ou seja, se o utilizador tem ou não amigos.

3. Caso tenha, é retornado um iterador da árvore binária de amigos, que é corrido na Main e em cada valor de next(), que representa um amigo, é feito um print das variáveis pedidas.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. IsEmpty (BST)	O(1)	O(1)	O(1)
3. Iterator (BST)	O(n)	O(n)	O(n)
Total	O(n)	O(n)	O(n)

• Listagem de aderentes de um grupo:

1. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

2. Verifica se o grupo tem algum membro, ou seja, se a sua árvore binária de membros não está vazia.

3. Caso tenha, é retornado um iterador da árvore binária de membros, que é corrido na Main e em cada valor de next(), que representa um membro, é feito um print das variáveis pedidas.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. IsEmpty (BST)	O(1)	O(1)	O(1)
3. Iterator (BST)	O(n)	O(n)	O(n)
Total	O(n)	O(n)	O(n)

• Listagem de posts de um contacto:

1. Verifica se ambos os utilizadores com os logins passados por argumento existem na tabela de dispersão.

2. Verifica se a lista duplamente ligada de posts do primeiro utilizador está vazia, ou seja, se o utilizador tem posts.

3. Verifica se um dos utilizadores existe na árvore binária do outro utilizador, ou seja, se existe amizade entre eles.

4. Caso exista, é retornado um iterador da lista duplamente ligada de posts do utilizador, que é corrido na Main e em cada valor de next(), que representa um post, é feito um print das variáveis pedidas.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. IsEmpty (DLL)	O(1)	O(1)	O(1)
3. Find (HT)	O(1)	O(n)	O(1+λ)
3. Iterator (DLL)	O(1)	O(n)	O(n)
Total	O(1)	O(n)	O(n)

• Listagem de posts de um grupo:

1. Verifica se existe um grupo na tabela de dispersão com o nome de grupo passado por argumento.

2. Verifica se existe um utilizador na tabela de dispersão com o login passado por argumento.

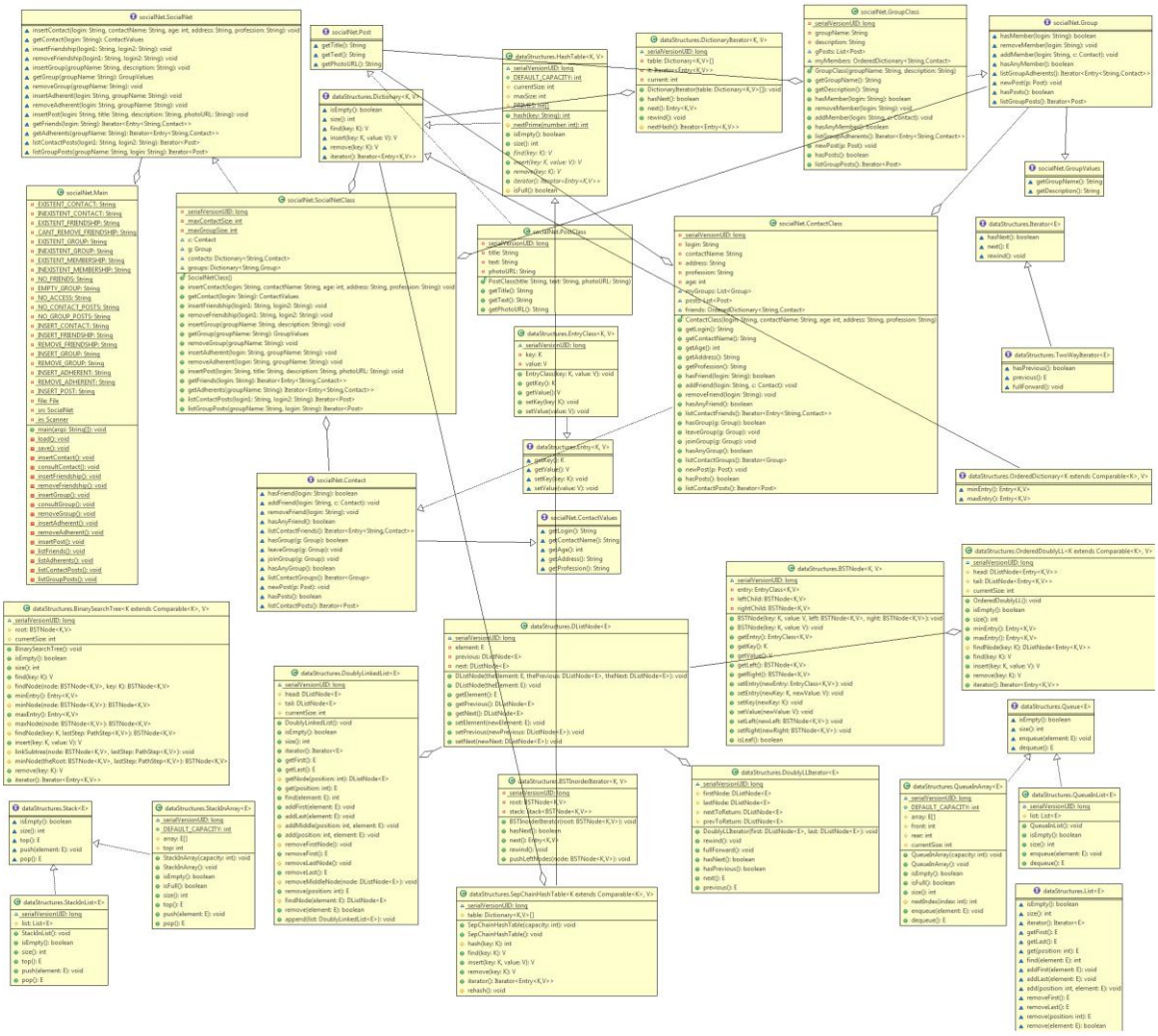
3. Verifica se o grupo existe na lista duplamente ligada de grupos no contacto e se o contacto existe na árvore binária de membros no grupo, ou seja, se existe aderência.

4. Verifica se a lista duplamente ligada de posts do grupo está vazia, ou seja, se o grupo tem posts.

5. Caso tenha, é retornado um iterador da lista duplamente ligada de posts do grupo, que é corrido na Main e em cada valor de next(), que representa um post, é feito um print das variáveis pedidas.

	Melhor Caso	**Pior Caso**	**Caso Esperado**
1. Find (HT)	O(1)	O(n)	O(1+λ)
2. Find (HT)	O(1)	O(n)	O(1+λ)
3. Find (DLL)	O(1)	O(n)	O(n)
3. Find (BST)	O(1)	O(n)	O(lg(n))
4. IsEmpty (DLL)	O(1)	O(1)	O(1)
5. Iterator (DLL)	O(1)	O(n)	O(n)
Total	O(1)	O(n)	O(n)

Diagrama de Classes e Interfaces



Código Fonte do Projecto

Main

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Scanner;

import dataStructures.*;
import socialNet.*;
import exceptions.*;

/**
 * @author Pedro Trindade Nº 41661 e Paulo Martins Nº 41982
 */

public class Main {

	//Comandos utilizados pelo utilizador.
	private static final String IC = "IC";
	private static final String CC = "CC";
	private static final String IA = "IA";
	private static final String RA = "RA";
	private static final String IG = "IG";
	private static final String CG = "CG";
	private static final String RG = "RG";
	private static final String ID = "ID";
	private static final String RD = "RD";
	private static final String IP = "IP";
	private static final String LA = "LA";
	private static final String LD = "LD";
	private static final String LC = "LC";
	private static final String LG = "LG";

	//Mensagens de erro apresentadas ao utilizador.
	private static final String EXISTENT_CONTACT = "Existencia de contacto
referido.";
	private static final String INEXISTENT_CONTACT = "Inexistencia de
contacto referido.";
	private static final String EXISTENT_FRIENDSHIP = "Existencia de amizade
referida.";
	private static final String INEXISTENT_FRIENDSHIP = "Inexistencia de
amizade referida.";
	private static final String CANT_REMOVE_FRIENDSHIP = "Amizade nao pode
ser removida.";
	private static final String EXISTENT_GROUP = "Existencia de grupo
referido.";
	private static final String INEXISTENT_GROUP = "Inexistencia de grupo
referido.";
```

```
	private static final String EXISTENT_MEMBERSHIP = "Existencia de
aderencia referida.";
	private static final String INEXISTENT_MEMBERSHIP = "Inexistencia de
aderencia referida.";
	private static final String NO_FRIENDS = "Contacto nao tem amigos.";
	private static final String EMPTY_GROUP = "Grupo nao tem aderentes.";
	private static final String NO_ACCESS = "Contacto nao tem permissao de
leitura de posts.";
	private static final String NO_CONTACT_POSTS = "Contacto nao tem posts.";
	private static final String NO_GROUP_POSTS = "Grupo nao tem posts.";

	//Mensagens de sucesso apresentadas ao utilizador.
	private static final String INSERT_CONTACT = "Insercao de contacto com
sucesso.";
	private static final String INSERT_FRIENDSHIP = "Insercao de amizade com
sucesso.";
	private static final String REMOVE_FRIENDSHIP = "Remocao de amizade com
sucesso.";
	private static final String INSERT_GROUP = "Insercao de grupo com
sucesso.";
	private static final String REMOVE_GROUP = "Remocao de grupo com
sucesso.";
	private static final String INSERT_ADHERENT = "Insercao de aderencia a
grupo com sucesso.";
	private static final String REMOVE_ADHERENT = "Remocao de aderencia com
sucesso.";
	private static final String INSERT_POST = "Insercao de post com
sucesso.";

	public static void main(String[] args) {
		Scanner in = new Scanner(System.in);

		File file = new File("data.txt");

		SocialNet sn = load(file);

		//Interpretador de comandos.
		do {
			String commands = in.next().toUpperCase();
			if(commands.equals(IC))
				insertContact(in, sn);
			else if(commands.equals(CC))
				consultContact(in, sn);
			else if(commands.equals(IA))
				insertFriendship(in, sn);
			else if(commands.equals(RA))
				removeFriendship(in, sn);
			else if(commands.equals(IG))
				insertGroup(in, sn);
			else if(commands.equals(CG))
				consultGroup(in, sn);
			else if(commands.equals(RG))
				removeGroup(in, sn);
			else if(commands.equals(ID))
				insertAdherent(in, sn);
```

```
                else if(commands.equals(RD))
                        removeAdherent(in, sn);
                else if(commands.equals(IP))
                        insertPost(in, sn);
                else if(commands.equals(LA))
                        listFriends(in, sn);
                else if(commands.equals(LD))
                        listAdherents(in, sn);
                else if(commands.equals(LC))
                        listContactPosts(in, sn);
                else if(commands.equals(LG))
                        listGroupPosts(in, sn);
                System.out.println();
                } while(in.hasNext());
        save(sn, file);
        }

        private static SocialNet load(File file) {
                SocialNet sn = null;

                try {
                        FileInputStream fileIn = new FileInputStream(file);
                        ObjectInputStream in = new ObjectInputStream(fileIn);
                        sn = (SocialNet) in.readObject();
                        in.close();
                        fileIn.close();
                } catch (IOException i) {
                        sn = new SocialNetClass();
                } catch (ClassNotFoundException c) {
                        c.printStackTrace();
                }

                return sn;
        }

        private static void save(SocialNet sn, File file) {
                try {
                        FileOutputStream fileOut = new FileOutputStream(file);
                        ObjectOutputStream out = new
ObjectOutputStream(fileOut);
                        out.writeObject(sn);
                        out.close();
                        fileOut.close();
                } catch (IOException i) {
                        i.printStackTrace();
                }
        }

        //Command methods
        private static void insertContact(Scanner in, SocialNet sn) {
                String login = in.next().trim().toUpperCase();
                String contactName = in.nextLine().trim();
                int age = in.nextInt();
                String address = in.nextLine().trim();
                String profession = in.nextLine();
```

```
                        in.nextLine();

                        try {
                                sn.insertContact(login, contactName, age, address,
profession);
                                System.out.println(INSERT_CONTACT);
                        }
                        catch (ExistentContactException ex) {
                                System.out.println(EXISTENT_CONTACT);
                        }
                }

                private static void consultContact(Scanner in, SocialNet sn) {
                        String login = in.nextLine().trim().toUpperCase();
                        in.nextLine();

                        try {
                                ContactValues c = sn.getContact(login);
                                System.out.println(c.getLogin().toUpperCase() + " " +
c.getContactName().toUpperCase() + " " + c.getAge());
                                System.out.println(c.getAddress().toUpperCase() + " "
+ c.getProfession().toUpperCase());
                        }
                        catch (InexistentContactException ex) {
                                System.out.println(INEXISTENT_CONTACT);
                        }
                }

                private static void insertFriendship(Scanner in, SocialNet sn) {
                        String login1 = in.next().trim().toUpperCase();
                        String login2 = in.nextLine().trim().toUpperCase();
                        in.nextLine();

                        try {
                                sn.insertFriendship(login1, login2);
                                System.out.println(INSERT_FRIENDSHIP);
                        }
                        catch (InexistentContactException ex) {
                                System.out.println(INEXISTENT_CONTACT);
                        }
                        catch (ExistentFriendshipException ex) {
                                System.out.println(EXISTENT_FRIENDSHIP);
                        }
                }

                private static void removeFriendship(Scanner in, SocialNet sn) {
                        String login1 = in.next().trim().toUpperCase();
                        String login2 = in.nextLine().trim().toUpperCase();
                        in.nextLine();

                        try {
                                sn.removeFriendship(login1, login2);
                                System.out.println(REMOVE_FRIENDSHIP);
                        }
                        catch (InexistentContactException ex) {
```

```
                System.out.println(INEXISTENT_CONTACT);
        }
        catch (SelfFriendException ex) {
                System.out.println(CANT_REMOVE_FRIENDSHIP);
        }
        catch (InexistentFriendshipException ex) {
                System.out.println(INEXISTENT_FRIENDSHIP);
        }
}

private static void insertGroup(Scanner in, SocialNet sn) {
        String groupName = in.nextLine().trim().toUpperCase();
        String description = in.nextLine();
        in.nextLine();

        try {
                sn.insertGroup(groupName, description);
                System.out.println(INSERT_GROUP);
        }
        catch (ExistentGroupException ex) {
                System.out.println(EXISTENT_GROUP);
        }
}

private static void consultGroup(Scanner in, SocialNet sn) {
        String groupName = in.nextLine().trim().toUpperCase();
        in.nextLine();

        try {
                GroupValues g = sn.getGroup(groupName);
                System.out.println(g.getGroupName().toUpperCase());
                System.out.println(g.getDescription().toUpperCase());
        }
        catch (InexistentGroupException ex) {
                System.out.println(INEXISTENT_GROUP);
        }
}

private static void removeGroup(Scanner in, SocialNet sn) {
        String groupName = in.nextLine().trim().toUpperCase();
        in.nextLine();

        try {
                sn.removeGroup(groupName);
                System.out.println(REMOVE_GROUP);
        }
        catch (InexistentGroupException ex) {
                System.out.println(INEXISTENT_GROUP);
        }
}

private static void insertAdherent(Scanner in, SocialNet sn) {
        String login = in.next().trim().toUpperCase();
        String groupName = in.nextLine().trim().toUpperCase();
        in.nextLine();
```

```
		try {
			sn.insertAdherent(login, groupName);
			System.out.println(INSERT_ADHERENT);
		}
		catch (InexistentContactException ex) {
			System.out.println(INEXISTENT_CONTACT);
		}
		catch (InexistentGroupException ex) {
			System.out.println(INEXISTENT_GROUP);
		}
		catch (ExistentMembershipException ex) {
			System.out.println(EXISTENT_MEMBERSHIP);
		}
	}

	private static void removeAdherent(Scanner in, SocialNet sn) {
		String login = in.next().trim().toUpperCase();
		String groupName = in.nextLine().trim().toUpperCase();
		in.nextLine();

		try {
			sn.removeAdherent(login, groupName);
			System.out.println(REMOVE_ADHERENT);
		}
		catch (InexistentContactException ex) {
			System.out.println(INEXISTENT_CONTACT);
		}
		catch (InexistentGroupException ex) {
			System.out.println(INEXISTENT_GROUP);
		}
		catch (InexistentMembershipException ex) {
			System.out.println(INEXISTENT_MEMBERSHIP);
		}
	}

	private static void insertPost(Scanner in, SocialNet sn) {
		String login = in.nextLine().trim().toUpperCase();
		String title = in.nextLine();
		String description = in.nextLine();
		String photoURL = in.nextLine();
		in.nextLine();

		try {
			sn.insertPost(login, title, description, photoURL);
			System.out.println(INSERT_POST);
		}
		catch (InexistentContactException ex) {
			System.out.println(INEXISTENT_CONTACT);
		}
	}

	private static void listFriends(Scanner in, SocialNet sn) {
		String login = in.nextLine().trim().toUpperCase();
		in.nextLine();
```

```
                    try {
                          Iterator<Entry<String, Contact>> it =
sn.getFriends(login);
                          ContactValues f = null;
                          while(it.hasNext()) {
                                f = it.next().getValue();
                                System.out.println(f.getLogin().toUpperCase() +
" " + f.getContactName().toUpperCase());
                          }
                    }
                    catch (InexistentContactException ex) {
                          System.out.println(INEXISTENT_CONTACT);
                    }
                    catch (NoFriendsException ex) {
                          System.out.println(NO_FRIENDS);
                    }
              }

              private static void listAdherents(Scanner in, SocialNet sn) {
                    String groupName = in.nextLine().trim().toUpperCase();
                    in.nextLine();

                    try {
                          Iterator<Entry<String, Contact>> it =
sn.getAdherents(groupName);
                          ContactValues a = null;
                          while(it.hasNext()) {
                                a = it.next().getValue();
                                System.out.println(a.getLogin().toUpperCase() +
" " + a.getContactName().toUpperCase());
                          }
                    }
                    catch (InexistentGroupException ex) {
                          System.out.println(INEXISTENT_GROUP);
                    }
                    catch (EmptyGroupException ex) {
                          System.out.println(EMPTY_GROUP);
                    }
              }

              private static void listContactPosts(Scanner in, SocialNet sn) {
                    String login1 = in.next().trim().toUpperCase();
                    String login2 = in.nextLine().trim().toUpperCase();
                    in.nextLine();

                    try {
                          Iterator<Post> it = sn.listContactPosts(login1,
login2);
                          Post p = null;
                          while(it.hasNext()) {
                                p = it.next();
                                System.out.println(p.getTitle().toUpperCase());
                                System.out.println(p.getText().toUpperCase());
```

```
	System.out.println(p.getPhotoURL().toUpperCase());
                                if(it.hasNext()) //O último post não necessita
de imprimir esta linha extra.
                                        System.out.println();
                                }
                        }
                        catch (InexistentContactException ex) {
                                System.out.println(INEXISTENT_CONTACT);
                        }
                        catch (NoPostsException ex) {
                                System.out.println(NO_CONTACT_POSTS);
                        }
                        catch (InexistentFriendshipException ex) {
                                System.out.println(NO_ACCESS);
                        }
                }

                private static void listGroupPosts(Scanner in, SocialNet sn) {
                        String groupName = in.next().trim().toUpperCase();
                        String login = in.nextLine().trim().toUpperCase();
                        in.nextLine();

                        try {
                                Iterator<Post> it = sn.listGroupPosts(groupName,
login);
                                Post p = null;
                                while(it.hasNext()) {
                                        p = it.next();
                                        System.out.println(p.getTitle().toUpperCase());
                                        System.out.println(p.getText().toUpperCase());

	System.out.println(p.getPhotoURL().toUpperCase());
                                if(it.hasNext()) //O último post não necessita
de imprimir esta linha extra.
                                        System.out.println();
                                }
                        }
                        catch (InexistentGroupException ex) {
                                System.out.println(INEXISTENT_GROUP);
                        }
                        catch (InexistentContactException ex) {
                                System.out.println(INEXISTENT_CONTACT);
                        }
                        catch (InexistentMembershipException ex) {
                                System.out.println(NO_ACCESS);
                        }
                        catch(NoPostsException ex) {
                                System.out.println(NO_GROUP_POSTS);
                        }
                }

}
```

DataStructures

BinarySearchTree

```
package dataStructures;

public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K, V> {

    static final long serialVersionUID = 0L;

    // The root of the tree.
    protected BSTNode<K, V> root;

    // Number of entries in the tree.
    protected int currentSize;

    protected static class PathStep<K, V> {

        // The parent of the node.
        public BSTNode<K, V> parent;

        // The node is the left or the right child of parent.
        public boolean isLeftChild;

        public PathStep(BSTNode<K,V> theParent, boolean toTheLeft) {
            parent = theParent;
            isLeftChild = toTheLeft;
        }

        public void set(BSTNode<K,V> newParent, boolean toTheLeft) {
            parent = newParent;
            isLeftChild = toTheLeft;
        }
    }

    public BinarySearchTree() {
        root = null;
        currentSize = 0;
    }

    // Returs true if the dictionary contains no entries.
    public boolean isEmpty() {
        return root == null;
    }

    // Returns the number of entries in the dictionary.
    public int size() {
        return currentSize;
    }

    // If there is an entry in the dictionary whose key is the specified key,
    // returns its value; otherwise, returns null.
    public V find(K key) {
```

```
    BSTNode<K, V> node = this.findNode(root, key);
    if(node == null)
        return null;
    else
        return node.getValue();
}

// Returns the node whose key is the specified key;
// or null if no such node exists.
protected BSTNode<K, V> findNode(BSTNode<K, V> node, K key) {
    if(node == null)
        return null;
    else {
        int compResult = key.compareTo(node.getKey());
        if(compResult == 0)
            return node;
        else if(compResult < 0)
            return this.findNode(node.getLeft(), key);
        else
            return this.findNode(node.getRight(), key);
    }
}

// Returns the entry with the smallest key in the dictionary.
public Entry<K, V> minEntry() throws EmptyDictionaryException {
    if(this.isEmpty())
        throw new EmptyDictionaryException();

    return this.minNode(root).getEntry();
}

// Returns the node with the smallest key
// in the tree rooted at the specified node.
// Precondition: node != null.
protected BSTNode<K, V> minNode(BSTNode<K, V> node) {
    if(node.getLeft() == null)
        return node;
    else
        return this.minNode(node.getLeft());
}

// Returns the entry with the largest key in the dictionary.
public Entry<K, V> maxEntry() throws EmptyDictionaryException {
    if(this.isEmpty())
        throw new EmptyDictionaryException();

    return this.maxNode(root).getEntry();
}

// Returns the node with the largest key
// in the tree rooted at the specified node.
// Precondition: node != null.
protected BSTNode<K, V> maxNode(BSTNode<K, V> node) {
    if(node.getRight() == null)
        return node;
```

```
    else
        return this.maxNode(node.getRight());
}

// Returns the node whose key is the specified key;
// or null if no such node exists.
// Moreover, stores the last step of the path in lastStep.
protected BSTNode<K, V> findNode(K key, PathStep<K, V> lastStep) {
    BSTNode<K, V> node = root;
    while(node != null) {
        int compResult = key.compareTo(node.getKey());
        if(compResult == 0)
            return node;
        else if(compResult < 0) {
            lastStep.set(node, true);
            node = node.getLeft();
        }
        else {
            lastStep.set(node, false);
            node = node.getRight();
        }
    }
    return null;
}

// If there is an entry in the dictionary whose key is the specified key,
// replaces its value by the specified value and returns the old value;
// otherwise, inserts the entry (key, value) and returns null.
public V insert(K key, V value) {
    PathStep<K, V> lastStep = new PathStep<K, V>(null, false);
    BSTNode<K, V> node = this.findNode(key, lastStep);
    if(node == null) {
        BSTNode<K, V> newLeaf = new BSTNode<K, V>(key, value);
        this.linkSubtree(newLeaf, lastStep);
        currentSize++;
        return null;
    }
    else {
        V oldValue = node.getValue();
        node.setValue(value);
        return oldValue;
    }
}

// Links a new subtree, rooted at the specified node, to the tree.
// The parent of the old subtree is stored in lastStep.
protected void linkSubtree(BSTNode<K, V> node, PathStep<K, V> lastStep) {
    if(lastStep.parent == null)
        // Change the root of the tree.
        root = node;
    else
        // Change a child of parent.
        if(lastStep.isLeftChild)
            lastStep.parent.setLeft(node);
        else
```

```
                lastStep.parent.setRight(node);
    }

    // Returns the node with the smallest key
    // in the tree rooted at the specified node.
    // Moreover, stores the last step of the path in lastStep.
    // Precondition: theRoot != null.
    protected BSTNode<K, V> minNode(BSTNode<K, V> theRoot, PathStep<K, V>
lastStep) {
        BSTNode<K, V> node = theRoot;
        while(node.getLeft() != null) {
            lastStep.set(node, true);
            node = node.getLeft();
        }
        return node;
    }

    // If there is an entry in the dictionary whose key is the specified key,
    // removes it from the dictionary and returns its value;
    // otherwise, returns null.
    public V remove(K key) {
        PathStep<K, V> lastStep = new PathStep<K, V>(null, false);
        BSTNode<K, V> node = this.findNode(key, lastStep);
        if(node == null)
            return null;
        else {
            V oldValue = node.getValue();
            if(node.getLeft() == null)
                // The left subtree is empty.
                this.linkSubtree(node.getRight(), lastStep);
            else if(node.getRight() == null)
                // The right subtree is empty.
                this.linkSubtree(node.getLeft(), lastStep);
            else {
                // Node has 2 children. Replace the node's entry with
                // the 'minEntry' of the right subtree.
                lastStep.set(node, false);
                BSTNode<K, V> minNode = this.minNode(node.getRight(), lastStep);
                node.setEntry(minNode.getEntry());
                // Remove the 'minEntry' of the right subtree.
                this.linkSubtree(minNode.getRight(), lastStep);
            }
            currentSize--;
            return oldValue;
        }
    }

    // Returns an iterator of the entries in the dictionary
    // which preserves the key order relation.
    public Iterator<Entry<K, V>> iterator() {
        return new BSTInorderIterator<K, V>(root);
    }

}
```

BSTInOrderIterator

```
package dataStructures;

public class BSTInorderIterator<K, V> implements Iterator<Entry<K, V>> {

	private static final long serialVersionUID = 0L;

	// The root of the BST.
	private BSTNode<K, V> root;

	// The stack used to save the nodes.
	private Stack<BSTNode<K, V>> stack;

	public BSTInorderIterator(BSTNode<K, V> root) {
		this.root = root;
		this.rewind();
	}

	public boolean hasNext() {
		return (!stack.isEmpty());
	}

	public Entry<K, V> next() throws NoSuchElementException {
		if(!this.hasNext()) {
			throw new NoSuchElementException();
		}

		BSTNode<K, V> next = stack.pop();
		pushLeftNodes(next.getRight());

		return next.getEntry();
	}

	public void rewind() {
		stack = new StackInList<BSTNode<K, V>>();
		pushLeftNodes(root);
	}

	public void pushLeftNodes(BSTNode<K, V> node) {
		while(node != null) {
			stack.push(node);
			node = node.getLeft();
		}
	}

}
```

BSTNode

```
package dataStructures;

import java.io.Serializable;

class BSTNode<K, V> implements Serializable {

    static final long serialVersionUID = 0L;

    // Entry stored in the node.
    private EntryClass<K, V> entry;

    // (Pointer to) the left child.
    private BSTNode<K, V> leftChild;

    // (Pointer to) the right child.
    private BSTNode<K, V> rightChild;

    public BSTNode(K key, V value, BSTNode<K, V> left, BSTNode<K, V> right) {
        entry = new EntryClass<K, V>(key, value);
        leftChild = left;
        rightChild = right;
    }

    public BSTNode(K key, V value) {
        this(key, value, null, null);
    }

    public EntryClass<K, V> getEntry() {
        return entry;
    }

    public K getKey() {
        return entry.getKey();
    }

    public V getValue() {
        return entry.getValue();
    }

    public BSTNode<K, V> getLeft() {
        return leftChild;
    }

    public BSTNode<K, V> getRight() {
        return rightChild;
    }

    public void setEntry(EntryClass<K, V> newEntry) {
        entry = newEntry;
    }

    public void setEntry(K newKey, V newValue) {
        entry.setKey(newKey);
```

```
        entry.setValue(newValue);
    }

    public void setKey(K newKey) {
        entry.setKey(newKey);
    }

    public void setValue(V newValue) {
        entry.setValue(newValue);
    }

    public void setLeft(BSTNode<K, V> newLeft) {
        leftChild = newLeft;
    }

    public void setRight(BSTNode<K, V> newRight) {
        rightChild = newRight;
    }

    // Returns true if the node is a leaf.
    public boolean isLeaf() {
        return leftChild == null && rightChild == null;
    }

}
```

Dictionary

```
package dataStructures;

import java.io.Serializable;

public interface Dictionary<K, V> extends Serializable {

    //Returns true if the dictionary contains no entries.
    boolean isEmpty();

    //Returns the number of entries in the dictionary.
    int size();

    //If there is an entry in the dictionary whose key is the specified key,
    //returns its value; otherwise, returns null.
    V find(K key);

    //If there is an entry in the dictionary whose key is the specified key,
    //replaces its value by the specified value and returns the old value;
    //otherwise, inserts the entry (key, value) and returns null.
    V insert(K key, V value);

    //If there is an entry in the dictionary whose key is the specified key,
    //removes it from the dictionary and returns its value;
```

```
    //otherwise, returns null.
    V remove(K key);

    //Returns an iterator of the entries in the dictionary.
    Iterator<Entry<K, V>> iterator();

}
```

DictionaryIterator

```
package dataStructures;

public class DictionaryIterator<K, V> implements Iterator<Entry<K, V>> {

	static final long serialVersionUID = 0L;

	private Dictionary<K, V>[] table;
	private Iterator<Entry<K, V>> it;
	private int current;

	public DictionaryIterator(Dictionary<K, V>[] table) {
		this.table = table;
		this.rewind();
	}

	public boolean hasNext() {
		return (current < table.length);
	}

	public Entry<K, V> next() throws NoSuchElementException {
		Entry<K, V> next = it.next();

		if(!this.hasNext())
			throw new NoSuchElementException();

		if(!it.hasNext()) {
			current++;
			it = this.nextHash();
		}

		return next;
	}

	public void rewind() {
		current = 0;
		it = this.nextHash();
	}

	protected Iterator<Entry<K, V>> nextHash() {
		it = null;
```

```
            while(this.hasNext() && table[current].isEmpty()) {
                  current++;
            }

            if(current < table.length)
                  it = table[current].iterator();

            return it;
      }

}
```

DListNode

```
package dataStructures;

import java.io.Serializable;

class DListNode<E> implements Serializable {

    static final long serialVersionUID = 0L;

    //Element stored in the node.
    private E element;

    //(Pointer to) the previous node.
    private DListNode<E> previous;

    //(Pointer to) the next node.
    private DListNode<E> next;

    public DListNode(E theElement, DListNode<E> thePrevious,
             DListNode<E> theNext) {
        element = theElement;
        previous = thePrevious;
        next = theNext;
    }

    public DListNode(E theElement) {
        this(theElement, null, null);
    }

    public E getElement() {
        return element;
    }

    public DListNode<E> getPrevious() {
        return previous;
    }

    public DListNode<E> getNext() {
```

```
        return next;
    }

    public void setElement(E newElement) {
        element = newElement;
    }

    public void setPrevious(DListNode<E> newPrevious) {
        previous = newPrevious;
    }

    public void setNext(DListNode<E> newNext) {
        next = newNext;
    }

}
```

DoublyLinkedList

```
package dataStructures;

public class DoublyLinkedList<E> implements List<E> {

    static final long serialVersionUID = 0L;

    //Node at the head of the list.
    protected DListNode<E> head;

    //Node at the tail of the list.
    protected DListNode<E> tail;

    //Number of elements in the list.
    protected int currentSize;

    public DoublyLinkedList() {
        head = null;
        tail = null;
        currentSize = 0;
    }

    //Returns true if the list contains no elements.
    public boolean isEmpty() {
        return currentSize == 0;
    }

    //Returns the number of elements in the list.
    public int size() {
        return currentSize;
    }

    //Returns an iterator of the elements in the list (in proper sequence).
```

```java
public Iterator<E> iterator() {
    return new DoublyLLIterator<E>(head, tail);
}

//Returns the first element of the list.
public E getFirst() throws EmptyListException {
    if(this.isEmpty())
        throw new EmptyListException();

    return head.getElement();
}

//Returns the last element of the list.
public E getLast() throws EmptyListException {
    if(this.isEmpty())
        throw new EmptyListException();

    return tail.getElement();
}

//Returns the node at the specified position in the list.
//Pre-condition: position ranges from 0 to currentSize-1.
protected DListNode<E> getNode(int position) {
    DListNode<E> node;

    if(position <= (currentSize - 1) / 2) {
        node = head;
        for(int i = 0; i < position; i++)
            node = node.getNext();
    }
    else {
        node = tail;
        for(int i = currentSize - 1; i > position; i--)
            node = node.getPrevious();
    }

    return node;
}

//Returns the element at the specified position in the list.
//Range of valid positions: 0, ..., size()-1.
//If the specified position is 0, get corresponds to getFirst.
//If the specified position is size()-1, get corresponds to getLast.
public E get(int position) throws InvalidPositionException {
    if(position < 0 || position >= currentSize)
        throw new InvalidPositionException();

    return this.getNode(position).getElement();
}

//Returns the position of the first occurrence of the specified element
//in the list, if the list contains the element.
//Otherwise, returns -1.
public int find(E element) {
    DListNode<E> node = head;
```

```
        int position = 0;

        while(node != null && !node.getElement().equals(element)) {
            node = node.getNext();
            position++;
        }

        if(node == null)
            return -1;
        else
            return position;
    }

    //Inserts the specified element at the first position in the list.
    public void addFirst(E element) {
        DListNode<E> newNode = new DListNode<E>(element, null, head);

        if(this.isEmpty())
            tail = newNode;
        else
            head.setPrevious(newNode);

        head = newNode;
        currentSize++;
    }

    //Inserts the specified element at the last position in the list.
    public void addLast(E element) {
        DListNode<E> newNode = new DListNode<E>(element, tail, null);

        if(this.isEmpty())
            head = newNode;
        else
            tail.setNext(newNode);

        tail = newNode;
        currentSize++;
    }

    //Inserts the specified element at the specified position in the list.
    //Pre-condition: position ranges from 1 to currentSize-1.
    protected void addMiddle(int position, E element) {
        DListNode<E> prevNode = this.getNode(position - 1);
        DListNode<E> nextNode = prevNode.getNext();
        DListNode<E> newNode = new DListNode<E>(element, prevNode, nextNode);
        prevNode.setNext(newNode);
        nextNode.setPrevious(newNode);
        currentSize++;
    }

    //Inserts the specified element at the specified position in the list.
    //Range of valid positions: 0, ..., size().
    //If the specified position is 0, add corresponds to addFirst.
    //If the specified position is size(), add corresponds to addLast.
    public void add(int position, E element) throws InvalidPositionException {
```

```
    if(position < 0 || position > currentSize)
        throw new InvalidPositionException();

    if(position == 0)
        this.addFirst(element);
    else if(position == currentSize)
        this.addLast(element);
    else
        this.addMiddle(position, element);
}

//Removes the first node in the list.
//Pre-condition: the list is not empty.
protected void removeFirstNode() {
    head = head.getNext();

    if(head == null)
        tail = null;
    else
        head.setPrevious(null);

    currentSize--;
}

//Removes and returns the element at the first position in the list.
public E removeFirst() throws EmptyListException {
    if(this.isEmpty())
        throw new EmptyListException();

    E element = head.getElement();
    this.removeFirstNode();
    return element;
}

//Removes the last node in the list.
//Pre-condition: the list is not empty.
protected void removeLastNode() {
    tail = tail.getPrevious();

    if(tail == null)
        head = null;
    else
        tail.setNext(null);

    currentSize--;
}

//Removes and returns the element at the last position in the list.
public E removeLast() throws EmptyListException {
    if(this.isEmpty())
        throw new EmptyListException();

    E element = tail.getElement();
    this.removeLastNode();
    return element;
```

```
}

//Removes the specified node from the list.
//Pre-condition: the node is neither the head nor the tail of the list.
protected void removeMiddleNode(DListNode<E> node) {
    DListNode<E> prevNode = node.getPrevious();
    DListNode<E> nextNode = node.getNext();
    prevNode.setNext(nextNode);
    nextNode.setPrevious(prevNode);
    currentSize--;
}

//Removes and returns the element at the specified position in the list.
//Range of valid positions: 0, ..., size()-1.
//If the specified position is 0, remove corresponds to removeFirst.
//If the specified position is size()-1, remove corresponds to removeLast.
public E remove(int position) throws InvalidPositionException {
    if(position < 0 || position >= currentSize)
        throw new InvalidPositionException();

    if(position == 0)
        return this.removeFirst();
    else if(position == currentSize - 1)
        return this.removeLast();
    else {
        DListNode<E> nodeToRemove = this.getNode(position);
        this.removeMiddleNode(nodeToRemove);
        return nodeToRemove.getElement();
    }
}

//Returns the node with the first occurrence of the specified element
//in the list, if the list contains the element.
//Otherwise, returns null.
protected DListNode<E> findNode(E element) {
    DListNode<E> node = head;

    while(node != null && !node.getElement().equals(element))
        node = node.getNext();

    return node;
}

//Removes the first occurrence of the specified element from the list
//and returns true, if the list contains the element.
//Otherwise, returns false.
public boolean remove(E element) {
    DListNode<E> node = this.findNode(element);

    if(node == null)
        return false;
    else {
        if(node == head)
            this.removeFirstNode();
        else if(node == tail)
```

```
                this.removeLastNode();
            else
                this.removeMiddleNode(node);
            return true;
        }
    }

    //Removes all of the elements from the specified list and
    //inserts them at the end of the list (in proper sequence).
    public void append(DoublyLinkedList<E> list) { //Comentado
      if(list.isEmpty()) //Se a lista recebida por argumento estiver vazia, o
método não faz nada.
            return;

      if(this.isEmpty()) //Se a lista estiver vazia, a head da lista passa a
ser a head da lista recebida por argumento.
            this.head = list.head;
      else {
            //Junta a tail da lista à head da lista recebida por argumento.
            this.tail.setNext(list.head);
            list.head.setPrevious(this.tail);
      }

      //A tail da lista passa a ser a tail da lista recebida por argumento,
sendo o tamanho da lista aumentado adequadamente.
      this.tail = list.tail;
      this.currentSize += list.currentSize;

      //Apaga a lista recebida por argumento.
      list.tail = null;
      list.head = null;
      list.currentSize = 0;
    }

}
```

DoublyLLIterator

```
package dataStructures;

class DoublyLLIterator<E> implements TwoWayIterator<E> {

    static final long serialVersionUID = 0L;

    //Node with the first element in the iteration.
    protected DListNode<E> firstNode;

    //Node with the last element in the iteration.
    protected DListNode<E> lastNode;

    //Node with the next element in the iteration.
```

```
protected DListNode<E> nextToReturn;

//Node with the previous element in the iteration.
protected DListNode<E> prevToReturn;

public DoublyLLIterator(DListNode<E> first, DListNode<E> last) {
    firstNode = first;
    lastNode = last;
    this.rewind();
}

//Restarts the iteration.
//After rewind, if the iteration is not empty,
//next will return the first element in the iteration.
public void rewind() {
    nextToReturn = firstNode;
    prevToReturn = null;
}

//Restarts the iteration in the reverse direction.
//After fullForward, if the iteration is not empty,
//previous will return the last element in the iteration.
public void fullForward() {
    prevToReturn = lastNode;
    nextToReturn = null;
}

//Returns true if the iteration has more elements.
//In other words, returns true if next would return an element
//rather than throwing an exception.
public boolean hasNext() {
    return nextToReturn != null;
}

//Returns true if the iteration has more elements
//in the reverse direction.
//In other words, returns true if previous would return an element
//rather than throwing an exception.
public boolean hasPrevious() {
    return prevToReturn != null;
}

//Returns the next element in the iteration.
public E next() throws NoSuchElementException {
    if(!this.hasNext())
        throw new NoSuchElementException();

    E element = nextToReturn.getElement();
    prevToReturn = nextToReturn.getPrevious();
    nextToReturn = nextToReturn.getNext();
    return element;
}

//Returns the previous element in the iteration.
public E previous() throws NoSuchElementException {
```

```
        if(!this.hasPrevious())
            throw new NoSuchElementException();

        E element = prevToReturn.getElement();
        nextToReturn = prevToReturn.getNext();
        prevToReturn = prevToReturn.getPrevious();
        return element;
    }

}
```

Entry

```
package dataStructures;

import java.io.Serializable;

public interface Entry<K, V> extends Serializable {

    //Returns the key in the entry.
    K getKey();

    //Returns the value in the entry.
    V getValue();

    //Sets the key in the entry.
    void setKey(K key);

    //Sets the value in the entry.
    void setValue(V value);

}
```

EntryClass

```
package dataStructures;

public class EntryClass<K, V> implements Entry<K, V> {

       static final long serialVersionUID = 0L;

       private K key;
       private V value;

       public EntryClass(K key, V value) {
              this.key = key;
```

```
            this.value = value;
        }

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
        }

        public void setKey(K key) {
            this.key = key;
        }

        public void setValue(V value) {
            this.value = value;
        }

}
```

HashTable

```
package dataStructures;

public abstract class HashTable<K, V> implements Dictionary<K, V> {

    static final long serialVersionUID = 0L;

    //Default size of the hash table.
    public static final int DEFAULT_CAPACITY = 50;

    //Number of entries in the hash table.
    protected int currentSize;

    //Maximum number of entries.
    protected int maxSize;

    //
    //Public Static Methods
    //

    //Returns the hash code of the specified key,
    //which is an integer in the range 0, ..., b-1.
    public static int hash(String key) {
        int a = 127;          // a is a prime number.
        int b = 2147483647;   // b is a prime number.
        int hashCode = 0;

        for(int i = 0; i < key.length(); i++)
            hashCode = (hashCode * a + key.charAt(i)) % b;
```

```
    return hashCode;
}

//
//Protected Static Methods
//

//Returns a prime number that is not less than the specified number;
//or zero if all such primes are greater than Integer.MAX_VALUE.
protected static int nextPrime(int number) {
    for(int i = 0; i < PRIMES.length; i++)
        if(PRIMES[i] >= number)
            return PRIMES[i];

    return 0;
}

protected static final int[] PRIMES =
{
    11, 19, 31, 47, 73,
    113, 181, 277, 421, 643, 967,
    1451, 2179, 3299, 4951, 7433,
    11173, 16763, 25163, 37747, 56671, 85009,
    127529, 191299, 287003, 430517, 645787, 968689,
    1453043, 2179571, 3269377, 4904077, 7356119,
    11034223, 16551361, 24827059, 37240597, 55860923, 83791441,
    125687173, 188530777, 282796177, 424194271, 636291413, 954437161,
    1431655751, 2147483647
};

//
//Public Instance Methods
//

//Returns true if the dictionary contains no entries.
public boolean isEmpty() {
    return currentSize == 0;
}

//Returns the number of entries in the dictionary.
public int size() {
    return currentSize;
}

//If there is an entry in the dictionary whose key is the specified key,
//returns its value; otherwise, returns null.
public abstract V find(K key);

//If there is an entry in the dictionary whose key is the specified key,
//replaces its value by the specified value and returns the old value;
//otherwise, inserts the entry (key, value) and returns null.
public abstract V insert(K key, V value);

//If there is an entry in the dictionary whose key is the specified key,
//removes it from the dictionary and returns its value;
```

```
    //otherwise, returns null.
    public abstract V remove(K key);

    //Returns an iterator of the entries in the dictionary.
    public abstract Iterator<Entry<K, V>> iterator();

    //
    //Protected Instance Methods
    //

    //Returns true if the hash table cannot contain more entries.
    protected boolean isFull() {
        return currentSize == maxSize;
    }

}
```

Iterator

```
package dataStructures;

import java.io.Serializable;

public interface Iterator<E> extends Serializable {

    //Returns true if the iteration has more elements.
    //In other words, returns true if next would return an element
    //rather than throwing an exception.
    boolean hasNext();

    //Returns the next element in the iteration.
    E next() throws NoSuchElementException;

    //Restarts the iteration.
    //After rewind, if the iteration is not empty,
    //next will return the first element in the iteration.
    void rewind();

}
```

List

```
package dataStructures;

import java.io.Serializable;

public interface List<E> extends Serializable {

    //Returns true if the list contains no elements.
    boolean isEmpty();

    //Returns the number of elements in the list.
    int size();

    //Returns an iterator of the elements in the list (in proper sequence).
    Iterator<E> iterator();

    //Returns the first element of the list.
    E getFirst() throws EmptyListException;

    //Returns the last element of the list.
    E getLast() throws EmptyListException;

    //Returns the element at the specified position in the list.
    //Range of valid positions: 0, ..., size()-1.
    //If the specified position is 0, get corresponds to getFirst.
    //If the specified position is size()-1, get corresponds to getLast.
    E get(int position) throws InvalidPositionException;

    //Returns the position of the first occurrence of the specified element
    //in the list, if the list contains the element.
    //Otherwise, returns -1.
    int find(E element);

    //Inserts the specified element at the first position in the list.
    void addFirst(E element);

    //Inserts the specified element at the last position in the list.
    void addLast(E element);

    //Inserts the specified element at the specified position in the list.
    //Range of valid positions: 0, ..., size().
    //If the specified position is 0, add corresponds to addFirst.
    //If the specified position is size(), add corresponds to addLast.
    void add(int position, E element) throws InvalidPositionException;

    //Removes and returns the element at the first position in the list.
    E removeFirst() throws EmptyListException;

    //Removes and returns the element at the last position in the list.
    E removeLast() throws EmptyListException;

    //Removes and returns the element at the specified position in the list.
    //Range of valid positions: 0, ..., size()-1.
    //If the specified position is 0, remove corresponds to removeFirst.
```

```
    //If the specified position is size()-1, remove corresponds to removeLast.
    E remove(int position) throws InvalidPositionException;

    //Removes the first occurrence of the specified element from the list
    //and returns true, if the list contains the element.
    //Otherwise, returns false.
    boolean remove(E element);

}
```

OrderedDictionary

```
package dataStructures;

public interface OrderedDictionary<K extends Comparable<K>, V>
       extends Dictionary<K, V> {

       //Returns the entry with the smallest key in the dictionary.
       Entry<K, V> minEntry() throws EmptyDictionaryException;

       //Returns the entry with the largest key in the dictionary.
       Entry<K, V> maxEntry() throws EmptyDictionaryException;

       //Returns an iterator of the entries in the dictionary
       //which preserves the key order relation.
       //Iterator<Entry<K, V>> iterator();

}
```

OrderedDoublyLL

```
package dataStructures;

public class OrderedDoublyLL<K extends Comparable<K>, V>
       implements OrderedDictionary<K, V> {

       static final long serialVersionUID = 0L;

       //Node at the head of the list.
       protected DListNode<Entry<K, V>> head;

       //Node at the tail of the list.
       protected DListNode<Entry<K, V>> tail;

       //Number of elements in the list.
       protected int currentSize;
```

```
	public OrderedDoublyLL() {
		head = null;
		tail = null;
		currentSize = 0;
	}

	public boolean isEmpty() {
		return currentSize == 0;
	}

	public int size() {
		return currentSize;
	}

	public Entry<K, V> minEntry() throws EmptyDictionaryException {
		if(this.isEmpty())
			throw new EmptyDictionaryException();

		return head.getElement();
	}

	public Entry<K, V> maxEntry() throws EmptyDictionaryException {
		if(this.isEmpty())
			throw new EmptyDictionaryException();

		return tail.getElement();
	}

	protected DListNode<Entry<K, V>> findNode(K key) {
		DListNode<Entry<K, V>> node = head;

		while(node != null && node.getElement().getKey().compareTo(key) <
0) {
			node = node.getNext();
		}

		return node;
	}

	public V find(K key) {
		DListNode<Entry<K, V>> node = this.findNode(key);

		if(node == null || node.getElement().getKey().compareTo(key) != 0)
			return null;

		return node.getElement().getValue();
	}

	public V insert(K key, V value) {
		DListNode<Entry<K, V>> node;
		Entry<K, V> entry = new EntryClass<K, V>(key, value);
		V val = this.find(key);

		if(this.isEmpty()) {
```

```
                        node = new DListNode<Entry<K, V>>(entry, null, null);
                        head = node;
                        tail = node;
                        currentSize++;
                }
                else if(val != null) {
                        this.findNode(key).getElement().setValue(value);
                }
                else if(val == null) {
                        if(this.minEntry().getKey().compareTo(key) > 0) {
                                node = new DListNode<Entry<K, V>>(entry, null, head);
                                head.setPrevious(node);
                                head = node;
                        }
                        else if(this.maxEntry().getKey().compareTo(key) < 0) {
                                node = new DListNode<Entry<K, V>>(entry, tail, null);
                                tail.setNext(node);
                                tail = node;
                        }
                        else {
                                DListNode<Entry<K, V>> previous =
this.findNode(key).getPrevious();
                                DListNode<Entry<K, V>> next = this.findNode(key);
                                node = new DListNode<Entry<K, V>>(entry, previous,
next);
                                previous.setNext(node);
                                next.setPrevious(node);
                        }
                        currentSize++;
                }

                return val;
        }

        public V remove(K key) {
                DListNode<Entry<K, V>> node = this.findNode(key);

                if(this.isEmpty() || this.find(key) == null)
                        return null;

                if(node == head) {
                        head = node.getNext();
                        if(head == null)
                                tail = null;
                        else
                                head.setPrevious(null);
                }
                else if(node == tail) {
                        tail = node.getPrevious();
                        if(tail == null)
                                head = null;
                        else
                                tail.setNext(null);
                }
                else {
```

```
                        DListNode<Entry<K, V>> previous = node.getPrevious();
                        DListNode<Entry<K, V>> next = node.getNext();
                        previous.setNext(next);
                        next.setPrevious(previous);
                }

                currentSize--;

                return node.getElement().getValue();
        }

        public Iterator<Entry<K, V>> iterator() {
                return new DoublyLLIterator<Entry<K, V>>(head, tail);
        }

}
```

SepChainHashTable

```
package dataStructures;

public class SepChainHashTable<K extends Comparable<K>, V>
        extends HashTable<K, V> {

    static final long serialVersionUID = 0L;

    //The array of dictionaries.
    protected Dictionary<K, V>[] table;

    @SuppressWarnings("unchecked")
       public SepChainHashTable(int capacity) {
         int arraySize = HashTable.nextPrime((int) (1.1 * capacity));
         //Compiler gives a warning.
         table = (Dictionary<K, V>[]) new Dictionary[arraySize];
         for(int i = 0; i < arraySize; i++)
             table[i] = new OrderedDoublyLL<K, V>();

         maxSize = capacity;
         currentSize = 0;
    }

    public SepChainHashTable() {
        this(DEFAULT_CAPACITY);
    }

    //Returns the hash value of the specified key.
    protected int hash(K key) {
        return Math.abs(key.hashCode()) % table.length;
    }

    //If there is an entry in the dictionary whose key is the specified key,
```

```
    //returns its value; otherwise, returns null.
    public V find(K key) {
        return table[this.hash(key)].find(key);
    }

    //If there is an entry in the dictionary whose key is the specified key,
    //replaces its value by the specified value and returns the old value;
    //otherwise, inserts the entry (key, value) and returns null.
    public V insert(K key, V value) {
       if(this.isFull())
             this.rehash();

        V val = table[this.hash(key)].insert(key, value);

        if(val == null)
             currentSize++;

        return val;
    }

    //If there is an entry in the dictionary whose key is the specified key,
    //removes it from the dictionary and returns its value;
    //otherwise, returns null.
    public V remove(K key) {
       V val = table[this.hash(key)].remove(key);

        if(val != null)
             currentSize--;

        return val;
    }

    //Returns an iterator of the entries in the dictionary.
    public Iterator<Entry<K,V>> iterator() {
       return new DictionaryIterator<K, V> (table);
    }

    @SuppressWarnings("unchecked")
       protected void rehash() { //Test!
       //Duplica o tamanho máximo da tabela.
       maxSize = maxSize * 2;
       int newSize = HashTable.nextPrime((int) (1.1 * maxSize));

       //Compiler gives a warning.
       //Cria uma nova tabela com o novo tamanho máximo.
       Dictionary<K, V>[] newTable = (Dictionary <K, V>[]) new
Dictionary[newSize];

       //Inicializa o valor de cada posição da tabela, ou seja, as
OrderedDoublyLL.
       for(int i = 0; i < newSize; i++) {
             newTable[i] = new OrderedDoublyLL<K, V>();
       }
```

```
      //Cria e corre um iterador para inicializar os valores de cada posição em
todas as OrderedDoublyLL da tabela.
      Iterator<Entry<K, V>> it = this.iterator();
      Entry<K, V> entry = null;
      while(it.hasNext()) {
            entry = it.next();
            newTable[this.hash(entry.getKey())].insert(entry.getKey(),
entry.getValue());
      }

      //Iguala a tabela anterior à tabela criada com o dobro do tamanho.
      table = newTable;
    }

}
```

Stack

```
package dataStructures;

import java.io.Serializable;

public interface Stack<E> extends Serializable {

    // Returns true if the stack contains no elements.
    boolean isEmpty();

    // Returns the number of elements in the stack.
    int size();

    // Returns the element at the top of the stack.
    E top() throws EmptyStackException;

    // Inserts the specified element onto the top of the stack.
    void push(E element);

    // Removes and returns the element at the top of the stack.
    E pop() throws EmptyStackException;

}
```

StackInList

```
package dataStructures;

public class StackInList<E> implements Stack<E> {
```

```
    static final long serialVersionUID = 0L;

    // Memory of the stack: a list.
    protected List<E> list;

    public StackInList() {
        list = new DoublyLinkedList<E>();
    }

    // Returns true if the stack contains no elements.
    public boolean isEmpty() {
        return list.isEmpty();
    }

    // Returns the number of elements in the stack.
    public int size() {
        return list.size();
    }

    // Returns the element at the top of the stack.
    public E top() throws EmptyStackException {
        if(list.isEmpty())
            throw new EmptyStackException("Stack is empty.");

        return list.getFirst();
    }

    // Inserts the specified element onto the top of the stack.
    public void push(E element) {
        list.addFirst(element);
    }

    // Removes and returns the element at the top of the stack.
    public E pop() throws EmptyStackException {
        if(list.isEmpty())
            throw new EmptyStackException("Stack is empty.");

        return list.removeFirst();
    }

}
```

TwoWayIterator

```
package dataStructures;

public interface TwoWayIterator<E> extends Iterator<E> {

    //Returns true if the iteration has more elements
    //in the reverse direction.
```

```
    //In other words, returns true if previous would return an element
    //rather than throwing an exception.
    boolean hasPrevious();

    //Returns the previous element in the iteration.
    E previous() throws NoSuchElementException;

    //Restarts the iteration in the reverse direction.
    //After fullForward, if the iteration is not empty,
    //previous will return the last element in the iteration.
    void fullForward();

}
```

SocialNet

Contact

```
package socialNet;

import dataStructures.*;

public interface Contact extends java.io.Serializable, ContactValues {

	/**
	* Verifica se o utilizador tem o contacto identificado pelo login recebido por argumento no seu conjunto de amigos.
	* @param login - o login do contacto.
	* @return - true se os utilizadores forem amigos, false caso contrário.
	*/
	boolean hasFriend(String login);

	/**
	* Adiciona um amigo, representado pelo login e pelo objecto Contact, ambos recebidos por argumento.
	* @param login - o login do contacto.
	* @param c - o objecto Contact.
	*/
	void addFriend(String login, Contact c);

	/**
	* Remove um amigo do conjunto, identificado pelo login recebido por argumento.
	* @param login - o login do contacto.
	*/
	void removeFriend(String login);

	/**
	 * Verifica se o utilizador tem amigos, ou seja, se a árvore binária que representa o conjunto de amigos está ou não vazia.
	 * @return - true se o utilizador tiver amigos, false caso contrário.
	 */
	boolean hasAnyFriend();

	/**
	* Retorna um iterador da árvore binária de amigos do utilizador.
	* @return - o iterador da árvore binária de amigos.
	*/
	Iterator<Entry<String, Contact>> listContactFriends();

	/**
	* Verifica se o utilizador pertence ao grupo recebido por argumento.
	* @param g - o objecto grupo.
	* @return - true se o utilizador pertencer ao grupo, false caso contrário.
	*/
	boolean hasGroup(Group g);
```

```
	/**
	* Remove a aderência do utilizador ao grupo recebido por argumento.
	* @param g - o objecto grupo.
	*/
	void leaveGroup(Group g);

	/**
	* O utilizador adere ao grupo recebido por argumento.
	* @param g - o objecto Group.
	*/
	void joinGroup(Group g);

	/**
	 * Verifica se o utilizador pertence a um grupo.
	 * @return - true se o utilizador pertencer a um grupo, false caso
contrário.
	 */
	boolean hasAnyGroup();

	/**
	* Retorna um iterador da lista de grupos do utilizador.
	* @return - o iterador da lista de grupos.
	*/
	Iterator<Group> listContactGroups();

	/**
	* Insere um novo post na lista de posts, recebido por argumento.
	* @param post - o objecto Post.
	*/
	void newPost(Post p);

	/**
	* Verifica se o utilizador contém posts, ou seja, se a sua lista de posts
não está vazia.
	* @return - true se o utilizador tiver posts, false caso contrário.
	*/
	boolean hasPosts();

	/**
	* Retorna um iterador da lista de posts do utilizador.
	* @return - o iterador da lista de posts.
	*/
	Iterator<Post> listContactPosts();

}
```

ContactClass

```
package socialNet;

import dataStructures.*;
```

```java
public class ContactClass implements Contact, java.io.Serializable {

	private static final long serialVersionUID = 0L;

	private String login, contactName, address, profession; //Variáveis primitivas da classe, o login, o nome, a localidade, a profissão
	private int age; //e a idade do utilizador.

	List<Group> myGroups; //Lista dos grupos a que o utilizador aderiu.
	List<Post> posts; //Lista dos posts submetidos pelo utilizador.

	OrderedDictionary<String, Contact> friends; //Árvore binária dos amigos do utilizador.

	public ContactClass(String login, String contactName, int age, String address, String profession) {
		this.login = login;
		this.contactName = contactName;
		this.age = age;
		this.address = address;
		this.profession = profession;
		myGroups = new DoublyLinkedList<Group>(); //Max size = 10
		posts = new DoublyLinkedList<Post>(); //Max size = 200
		friends = new BinarySearchTree<String, Contact>();
	}

	public String getLogin() {
		return login;
	}

	public String getContactName() {
		return contactName;
	}

	public int getAge() {
		return age;
	}

	public String getAddress() {
		return address;
	}

	public String getProfession() {
		return profession;
	}

	public boolean hasFriend(String login) {
		return (friends.find(login) != null);
	}

	public void addFriend(String login, Contact c) {
		friends.insert(login, c);
	}
```

```
	public void removeFriend(String login) {
		friends.remove(login);
	}

	public boolean hasAnyFriend() {
		return (!friends.isEmpty());
	}

	public Iterator<Entry<String, Contact>> listContactFriends() {
		return friends.iterator();
	}

	public boolean hasGroup(Group g) {
		return (myGroups.find(g) != -1);
	}

	public void leaveGroup(Group g) {
		myGroups.remove(g);
	}

	public void joinGroup(Group g) {
		myGroups.addFirst(g);
	}

	public boolean hasAnyGroup() {
		return (!myGroups.isEmpty());
	}

	public Iterator<Group> listContactGroups() {
		return myGroups.iterator();
	}

	public void newPost(Post p) {
		posts.addFirst(p);
	}

	public boolean hasPosts() {
		return (!posts.isEmpty());
	}

	public Iterator<Post> listContactPosts() {
		return posts.iterator();
	}

}
```

ContactValues

```
package socialNet;

public interface ContactValues {

	/**
	* Consulta o login do utilizador.
	* @return - o login do utilizador.
	*/
	String getLogin();

	/**
	* Consulta o nome do utilizador.
	* @return - o nome do utilizador.
	*/
	String getContactName();

	/**
	* Consulta a idade do utilizador.
	* @return - a idade do utilizador.
	*/
	int getAge();

	/**
	* Consulta a localidade do utilizador.
	* @return - a localidade do utilizador.
	*/
	String getAddress();

	/**
	* Consulta a profissão do utilizador.
	* @return - a profissão do utilizador.
	*/
	String getProfession();

}
```

Group

```
package socialNet;

import dataStructures.*;

public interface Group extends java.io.Serializable, GroupValues {

	/**
	* Verifica se o utilizador representado pelo login recebido por argumento
pertence ao grupo.
	* @param login - o login do utilizador.
```

```
	 * @return - true se o utilizador pertencer ao grupo, false caso
contrário.
	 */
	boolean hasMember(String login);

	/**
	 * Remove o membro do grupo identificado por login.
	 * @param login - o login do utilizador.
	 */
	void removeMember(String login);

	/**
	 * Adiciona o utilizador ao grupo, sendo este representado pelo login e
pelo objecto Contact, ambos recebidos por argumento.
	 * @param login - o login do utilizador.
	 * @param c - o objecto Contact.
	 */
	void addMember(String login, Contact c);

	/**
	 * Verifica se o grupo tem membros.
	 * @return - true se o grupo tiver membros, false caso contrário.
	 */
	boolean hasAnyMember();

	/**
	 * Retorna um iterador da árvore binária de aderentes do grupo.
	 * @return - o iterador da árvore binária de aderentes.
	 */
	Iterator<Entry<String, Contact>> listGroupAdherents();

	/**
	 * Insere um novo post na lista de posts, recebido por argumento.
	 * @param post - o objecto Post.
	 */
	void newPost(Post p);

	/**
	 * Verifica se o grupo contém posts, ou seja, se a sua lista de posts não
está vazia.
	 * @return - true se o grupo tiver posts, false caso contrário.
	 */
	boolean hasPosts();

	/**
	 * Retorna um iterador da lista de posts do grupo.
	 * @return - o iterador da lista de posts.
	 */
	Iterator<Post> listGroupPosts();

}
```

GroupClass

```
package socialNet;

import dataStructures.*;

public class GroupClass implements Group, java.io.Serializable {

	private static final long serialVersionUID = 0L;

	private String groupName, description; //Variáveis primitivas da classe, o nome e a descrição do grupo.

	List<Post> gPosts; //Lista de posts presentes na zona de comunicação do grupo.

	OrderedDictionary<String, Contact> myMembers; //Árvore binária dos membros do grupo.

	public GroupClass(String groupName, String description) {
		this.groupName = groupName;
		this.description = description;
		gPosts = new DoublyLinkedList<Post>(); //Max size = 200
		myMembers = new BinarySearchTree<String, Contact>();
	}

	public String getGroupName() {
		return groupName;
	}

	public String getDescription() {
		return description;
	}

	public boolean hasMember(String login) {
		return (myMembers.find(login) != null);
	}

	public void removeMember(String login) {
		myMembers.remove(login);
	}

	public void addMember(String login, Contact c) {
		myMembers.insert(login, c);
	}

	public boolean hasAnyMember() {
		return (!myMembers.isEmpty());
	}

	public Iterator<Entry<String, Contact>> listGroupAdherents() {
		return myMembers.iterator();
	}

	public void newPost(Post p) {
```

```
			gPosts.addFirst(p);
		}

		public boolean hasPosts() {
			return !gPosts.isEmpty();
		}

		public Iterator<Post> listGroupPosts() {
			return gPosts.iterator();
		}

}
```

GroupValues

```
package socialNet;

public interface GroupValues {

		/**
		* Consulta o nome do grupo.
		* @return - o nome do grupo.
		*/
		String getGroupName();

		/**
		* Consulta a descrição do grupo.
		* @return - a descrição do grupo.
		*/
		String getDescription();

}
```

Post

```
package socialNet;

public interface Post extends java.io.Serializable {

		/**
		* Consulta o título do post.
		* @return - o título do post.
		*/
		String getTitle();
```

```
	/**
	* Consulta o texto do post.
	* @return - o texto do post.
	*/
	String getText();

	/**
	* Consulta o url de uma foto relacionada com o post.
	* @return - o url de uma foto relacionada com o post.
	*/
	String getPhotoURL();

}
```

PostClass

```
package socialNet;

//Do another Constructor!
public class PostClass implements Post, java.io.Serializable {

	private static final long serialVersionUID = 0L;

	private String title, text, photoURL; //Variáveis primitivas da classe, o
título, o texto e o URL de uma foto relacionada com o post.

	public PostClass(String title, String text, String photoURL) {
		this.title = title;
		this.text = text;
		this.photoURL = photoURL;
	}

	public String getTitle() {
		return title;
	}

	public String getText() {
		return text;
	}

	public String getPhotoURL() {
		return photoURL;
	}

}
```

SocialNet

```
package socialNet;


import dataStructures.*;

import exceptions.*;


public interface SocialNet extends java.io.Serializable {


	/**

	 * Insere um utilizador no sistema, identificado por login, nome, idade, localidade a profissão.

	 * @param login - o login do utilizador.

	 * @param contactName - o nome do utilizador.

	 * @param age - a idade do utilizador.

	 * @param address - a localidade do utilizador.

	 * @param profession - a profissão do utilizador.

	 * @throws ExistentContactException - quando existe um utilizador com o mesmo login.

	 */

	void insertContact(String login, String contactName, int age, String address, String profession) throws ExistentContactException;


	/**

	 * Consulta os dados de um utilizador, identificado por login.

	 * @param login - o login do utilizador.

	 * @return - o objecto Contact.

	 * @throws InexistentContactException - quando o utilizador não existe.

	 */

	ContactValues getContact(String login) throws InexistentContactException;
```

```
	/**
	 * Estabelece uma relação de amizade entre os utilizadores idenficados pelos logins 1&2.
	 * @param login1 - o login do utilizador 1.
	 * @param login2 - o login do utilizador 2.
	 * @throws InexistentContactException - quando um dos utilizadores não existe.
	 * @throws ExistentFriendshipException - quando a relação de amizade já existe.
	 */
	void insertFriendship(String login1, String login2) throws InexistentContactException, ExistentFriendshipException;

	/**
	 * Remove uma relação de amizade entre os utilizadores identificados pelos logins 1&2.
	 * @param login1 - o login do utilizador 1.
	 * @param login2 - o login do utilizador 2.
	 * @throws InexistentContactException - quando um dos utilizadores não existe.
	 * @throws SelfFriendException - quando o amigo do utilizador é ele mesmo.
	 * @throws InexistentFriendshipException - quando a relação de amizade não existe.
	 */
	void removeFriendship(String login1, String login2) throws InexistentContactException, SelfFriendException, InexistentFriendshipException;

	/**
	 * Insere um grupo no sistema, identificado por nome e descrição.
	 * @param groupName - o nome do grupo.
```

```
        * @param description - a descrição do grupo.
        * @throws ExistentGroupException - quando existe um grupo com o mesmo
nome.
        */
       void insertGroup(String groupName, String description) throws
ExistentGroupException;

       /**
        * Consulta os dados de um grupo, identificado por nome.
        * @param groupName - o nome do grupo.
        * @return - o objecto Group.
        * @throws InexistentGroupException - quando o grupo não existe.
        */
       GroupValues getGroup(String groupName) throws InexistentGroupException;

       /**
        * Remove o grupo identificado pelo nome.
        * @param groupName - o nome do grupo.
        * @throws InexistentGroupException - quando o grupo não existe.
        */
       void removeGroup(String groupName) throws InexistentGroupException;

       /**
        * Estabelece uma aderência do utilizador identificado por login ao grupo
identificado por nome.
        * @param login - o login do utilizador.
        * @param groupName - o nome do grupo.
        * @throws InexistentContactException - quando o utilizador não existe.
        * @throws InexistentGroupException - quando o grupo não existe.
```

```
        * @throws ExistentMembershipException - quando o utilizador já pertence
ao grupo.

        */

       void insertAdherent(String login, String groupName) throws
InexistentContactException, InexistentGroupException,
ExistentMembershipException;



       /**

        * Remove a aderência do utilizador identificado por login ao grupo
identificado por nome.

        * @param login - o login do utilizador.

        * @param groupName - o nome do grupo.

        * @throws InexistentContactException - quando o utilizador não existe.

        * @throws InexistentGroupException - quando o grupo não existe.

        * @throws InexistentMembershipException - quando o utilizador não
pertence ao grupo.

        */

       void removeAdherent(String login, String groupName) throws
InexistentContactException, InexistentGroupException,
InexistentMembershipException;



       /**

        * Insere um novo post, associado ao utilizador (identificado por login),
identificado por título,

        * descrição e pelo URL de uma foto que lhe está associada.

        * @param login - o login do utilizador.

        * @param title - o título do post.

        * @param description - a descrição do post.

        * @param photoURL - o URL da foto.

        * @throws InexistentContactException - quando o utilizador não existe.

        */
```

```
	void insertPost(String login, String title, String description, String
photoURL) throws InexistentContactException;


	/**
	 * Retorna um iterador de amigos do utilizador identificado por login.
	 * @param login - o login do utilizador.
	 * @return - o iterador de amigos.
	 * @throws InexistentContactException - quando o utilizador não existe.
	 * @throws NoFriendsException - quando o utilizador não tem amigos.
	 */
	Iterator<Entry<String, Contact>> getFriends(String login) throws
InexistentContactException, NoFriendsException;


	/**
	 * Retorna um iterador de aderentes do grupo identificado por nome.
	 * @param groupName - o nome do grupo.
	 * @return - o iterador de aderentes.
	 * @throws InexistentGroupException - quando o grupo não existe.
	 * @throws EmptyGroupException - quando o grupo está vazio (não tem
membros).
	 */
	Iterator<Entry<String, Contact>> getAdherents(String groupName) throws
InexistentGroupException, EmptyGroupException;


	/**
	 * Retorna um iterador de posts submetidos pelo utilizador identificado
pelo login 2, na zona de comunicação do utilizador identificado pelo login 1.
	 * @param login1 - o login do utilizador.
	 * @param login2 - o login do utilizador.
	 * @return - o iterador de posts.
```

```
        * @throws InexistentContactException - quando o utilizador não existe.

        * @throws NoPostsException - quando o utilizador não tem posts.

        * @throws InexistentFriendshipException - quando a relação de amizade
não existe.

        */

       Iterator<Post> listContactPosts(String login1, String login2) throws
InexistentContactException, NoPostsException, InexistentFriendshipException;



       /**

        * Retorna um iterador de posts do grupo identificado por nome,
submetidos pelo utilizador identificado por login.

        * @param groupName - o nome do grupo.

        * @param login - o login do utilizador.

        * @return - o iterador de posts.

        * @throws InexistentGroupException - quando o grupo não existe.

        * @throws InexistentContactException - quando o utilizador não existe.

        * @throws InexistentMembershipException - quando o utilizador não
pertence ao grupo.

        * @throws NoPostsException - quando o grupo não tem posts.

        */

       Iterator<Post> listGroupPosts(String groupName, String login) throws
InexistentGroupException, InexistentContactException,
InexistentMembershipException, NoPostsException;



}
```

SocialNetClass

```
package socialNet;


import dataStructures.*;

import exceptions.*;


public class SocialNetClass implements SocialNet, java.io.Serializable {


	private static final long serialVersionUID = 0L;


	private static final int maxContactSize = 12000; //Tamanho máximo do
dicionário de contactos.

	private static final int maxGroupSize = 3000; //Tamanho máximo do
dicionário de grupos.


	Contact c; //Objecto contacto.

	Group g; //Objecto grupo.


	Dictionary<String, Contact> contacts; //Dicionário de contactos.

	Dictionary<String, Group> groups; //Dicionário de grupos.


	public SocialNetClass() {

		c = null;

		g = null;

		contacts = new SepChainHashTable<String, Contact>(maxContactSize);

		groups = new SepChainHashTable<String, Group>(maxGroupSize);

	}
```

```
	public void insertContact(String login, String contactName, int age,
String address, String profession) throws ExistentContactException {

		if(contacts.find(login) != null)

			throw new ExistentContactException();


		c = new ContactClass(login, contactName, age, address, profession);

		contacts.insert(login, c);

	}


	public ContactValues getContact(String login) throws
InexistentContactException {

		if(contacts.find(login) == null)

			throw new InexistentContactException();


		ContactValues cv = contacts.find(login);


		return cv;

	}


	public void insertFriendship(String login1, String login2) throws
InexistentContactException, ExistentFriendshipException {

		if(contacts.find(login1) == null || contacts.find(login2) == null)

			throw new InexistentContactException();


		c = contacts.find(login1);


		if(login1.equalsIgnoreCase(login2))

			throw new ExistentFriendshipException();
```

```
            if(c.hasFriend(login2))
                  throw new ExistentFriendshipException();

            c.addFriend(login2, contacts.find(login2));
            contacts.find(login2).addFriend(login1, c);
      }

      public void removeFriendship(String login1, String login2) throws
InexistentContactException, SelfFriendException, InexistentFriendshipException {
            if(contacts.find(login1) == null || contacts.find(login2) == null)
                  throw new InexistentContactException();

            c = contacts.find(login1);

            if(login1.equalsIgnoreCase(login2))
                  throw new SelfFriendException();

            if(!c.hasFriend(login2))
                  throw new InexistentFriendshipException();

            c.removeFriend(login2);
            contacts.find(login2).removeFriend(login1);
      }

      public void insertGroup(String groupName, String description) throws
ExistentGroupException {
            if(groups.find(groupName) != null)
                  throw new ExistentGroupException();
```

```
            g = new GroupClass(groupName, description);

            groups.insert(groupName, g);

      }


      public GroupValues getGroup(String groupName) throws
InexistentGroupException {

            if(groups.find(groupName) == null)

                  throw new InexistentGroupException();


            GroupValues gv = groups.find(groupName);


            return gv;

      }


      public void removeGroup(String groupName) throws InexistentGroupException
{

            if(groups.find(groupName) == null)

                  throw new InexistentGroupException();


            g = groups.find(groupName);


            if(g.hasAnyMember()) {

                  Iterator<Entry<String, Contact>> it =
g.listGroupAdherents();

                  Contact c = null;

                  while(it.hasNext()) {

                        c = it.next().getValue();

                        c.leaveGroup(g);

                  }
```

```
		}


		groups.remove(groupName);

	}


	public void insertAdherent(String login, String groupName) throws
InexistentContactException, InexistentGroupException,
ExistentMembershipException {

		if(contacts.find(login) == null)

			throw new InexistentContactException();


		if(groups.find(groupName) == null)

			throw new InexistentGroupException();


		c = contacts.find(login);

		g = groups.find(groupName);


		if(c.hasGroup(g) && g.hasMember(login))

			throw new ExistentMembershipException();


		c.joinGroup(g);

		g.addMember(login, c);

	}


	public void removeAdherent(String login, String groupName) throws
InexistentContactException, InexistentGroupException,
InexistentMembershipException {

		if(contacts.find(login) == null)

			throw new InexistentContactException();
```

```
		if(groups.find(groupName) == null)
			throw new InexistentGroupException();

		c = contacts.find(login);
		g = groups.find(groupName);

		if(!c.hasGroup(g) || !g.hasMember(login))
			throw new InexistentMembershipException();

		c.leaveGroup(g);
		g.removeMember(login);
	}

	public void insertPost(String login, String title, String description,
String photoURL) throws InexistentContactException {
		if(contacts.find(login) == null)
			throw new InexistentContactException();

		Post p = null;
		c = contacts.find(login);
		p = new PostClass(title, description, photoURL);

		if(c.hasAnyFriend()) {
			Iterator<Entry<String, Contact>> it =
c.listContactFriends();
			Contact f = null;
			while(it.hasNext()) {
				f = it.next().getValue();
```

```
                        f.newPost(p);
                }
        }

        if(c.hasAnyGroup()) {
                Iterator<Group> it = c.listContactGroups();
                Group g = null;
                while(it.hasNext()) {
                        g = it.next();
                        g.newPost(p);
                }
        }

        c.newPost(p);
    }

    public Iterator<Entry<String, Contact>> getFriends(String login) throws
InexistentContactException, NoFriendsException {
        if(contacts.find(login) == null)
                throw new InexistentContactException();

        c = contacts.find(login);

        if(!c.hasAnyFriend())
                throw new NoFriendsException();

        return c.listContactFriends();
    }
```

```
	public Iterator<Entry<String, Contact>> getAdherents(String groupName)
throws InexistentGroupException, EmptyGroupException {

		if(groups.find(groupName) == null)

			throw new InexistentGroupException();


		g = groups.find(groupName);


		if(!g.hasAnyMember())

			throw new EmptyGroupException();


		return g.listGroupAdherents();

	}


	public Iterator<Post> listContactPosts(String login1, String login2)
throws InexistentContactException, NoPostsException,
InexistentFriendshipException {

		if(contacts.find(login1) == null || contacts.find(login2) == null)

			throw new InexistentContactException();


		c = contacts.find(login1);


		if(!c.hasPosts())

			throw new NoPostsException();


		if(!login1.equalsIgnoreCase(login2)) {

			if(!c.hasFriend(login2))

				throw new InexistentFriendshipException();

		}
```

```
            return c.listContactPosts();

      }


      public Iterator<Post> listGroupPosts(String groupName, String login)
throws InexistentGroupException, InexistentContactException,
InexistentMembershipException, NoPostsException {

            if(groups.find(groupName) == null)

                  throw new InexistentGroupException();



            if(contacts.find(login) == null)

                  throw new InexistentContactException();



            g = groups.find(groupName);

            c = contacts.find(login);



            if(!g.hasMember(login) || !c.hasGroup(g))

                  throw new InexistentMembershipException();



            if(!g.hasPosts())

                  throw new NoPostsException();



            return g.listGroupPosts();

      }


}
```