FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Algoritmos e Estruturas de Dados

Ano Letivo 2017/2018

Relatório Final Projeto

HomeAway

2º Ano de MIEI

Docente: Armanda Rodrigues

Discentes: Ana Beatriz Grilo 49930

Joana Cardoso 47738

Índice

Tipos Abstratos de Dados

Home:

A interface Home representa as casas do utilizador.

Cada casa, quando é criada no sistema, recebe um id, o id do dono , um preço para ficar na casa, um número máximo de pessoas, o número de pontos, que começa a zero e é depois incrementado consoante os pontos com que é avaliada, uma localização, uma descrição e o nome do dono.

Engloba todas as operações relativas às casas (Home) que não modificam os seus atributos, métodos get(), e possui também os métodos addTrip(int points) e hasStays().

O método addTrip(int points) permite adicionar uma estadia na casa em questão, recebendo o número de pontos com que a estadia foi avaliada, e o método hasStays() permite verificar se a casa tem ou não estadias.

A classe associada a esta interface (HomeClass) implementa a interface Serializable (marker interface), o que significa que a classe HomeClass pode ser serializada e o estado dos objetos da classe são guardados no disco.

```
public interface Home {

    /**
     * function to get user's identifier
     * @return String variable idUser
     */
    public String getIdUser();

    /**
     * function to get home's identifier
     * @return String variable idHome
     */
    public String getIdHome();

    /**
     * function to get the price of a stay at the home
     * @return int variable price
     */
    public int getPrice();
```

```
    /**
    * function to get the maximum number of people that can stay at the home
     * @return int variable numberOfPeople
     */
    public int getNumberOfPeople();

    /**
     * function to get the total number of points the home received
     * @return int variable points
     */
    public int getPoints();

    /**
     * function to get the location of the home
     * @return String variable location
     */
    public String getLocation();

    /**
     * function to get the description of the home
     * @return String variable description
     */
    public String getDescription();

    /**
     * function to get the address of the home
     * @return String variable address
     */
    public String getAddress();

    /**
     * function to get the owner's name
     * @return String variable owner, the owner's name
     */
    public String getOwner();

    /**
     * method that adds a trip to a specific home.
     * @param points: number of points given by the user to the house.
     */
    public void addTrip(int points);

    /**
     * function to verify if a certain house has been visited
     * @return boolean variable hasStays.
     */
    public boolean hasStays();
}
```

User:

A interface User representa o utilizador.

Cada utilizador, quando é criado no sistema, recebe um id, um email , um número de telefone, um nome, uma nacionalidade e uma morada.

Cada utilizador pode ter um conjunto de casas associado.

Engloba todas as operações relativas ao utilizador (User). Estas operações podem não modificar os atributos do utilizador, como as operações de consulta - get(), boleanos e iteradores - , mas também o podem fazer, como as operações modificadoras – void().

A classe associada a esta interface (UserClass) implementa a interface Serializable (marker interface), o que significa que a classe UserClass pode ser serializada e o estado dos objetos da classe são guardados no disco.

```
public interface User
{
	/**
	 * method to get the name of the user
	 * @return String variable name
	 */
	public String getName();

	/**
	 * method to get the email of the user
	 * @return String variable email
	 */
	public String getEmail();

	/**
	 * method to get the phone number of the user
	 * @return String variable phoneNumber
	 */
	public String getPhoneNumber();

	/**
	 * method to get the identifier of the user
	 * @return String variable idUser
	 */
	public String getIdUser();

	/**
	 * method to get the nationality of the user
	 * @return String variable nationality
```

```
 */
public String getNationality();

/**
 * method to get the address of the user
 * @return String variable address
 */
public String getAddress();

/**
* method to get the stays of a user (in this phase, it is simply a counter, counting the number
* of stays the user has had at the home inserted in the system)
* @return int variable counter
*/
public Iterator<Home> getStays();

/**
 * it updates the user's information
 * @param email : user's email
 * @param phoneNumber : user's phone number
 * @param address : user's address
 */
public void updateUser(String email, String phoneNumber, String address);

/**
* it adds a trip to the user’s information
 */
public void addTrip(Home home);

/**
 *  it finds and removes a user's home
 */
public void removeHome(String idHome);

/**
 * it adds a home to the user's information.
 */
public void addHome(Home home);

/**
 * verifies if the user is a host.
 * @return boolean variable isHost
 */
boolean isHost();

/**
 * function to receive the user's home as an object.
 * @return object home
 */
public Collection<Home> getHomes();

/**
```

```
     * verifies if the user is a traveller.
     * @return boolean variable isTraveller
     */
    public boolean isTraveller();

    /**
     * verifies if the user has travelled.
     * @return true in case the stack of stays is not empty
     */
    public boolean userHasStays();

}
```

As estruturas de dados utilizadas foram:

- ➔ Foi usado um TreeMap<K,V>() porque permite realizar pesquisas por chave - key(K) – que neste caso é o identificador da casa. A complexidade temporal vai ser O(log n) para a inserção (add()), remoção (remove()) e O(1) para a pesquisa – iterador.
- ➔ Foi usada uma StackInList<Home>() porque guarda a estadia mais recente no topo da pilha e a mais antiga no fundo da mesma o que permite que ao ser iterada seja possível listar logo da mais recente para a mais antiga, o que era pretendido. A complexidade para inserção (push()) e para iteração é O(1).

HomeAway:

A interface HomeAway representa o sistema completo e contém todas as operações possíveis de realizar no sistema.

A classe associada a esta interface (HomeAwayClass) implementa a interface Serializable (marker interface), o que significa que a classe HomeAwayClass pode ser serializada e o estado dos objetos da classe são guardados no disco.

```
public interface HomeAway
{
        /**
         * it inserts a new user into the system, with a certain idUser, email,
         *phoneNumber, name, nationality, address
         *  in case there is a user with the same idUser in the system, it
         *throws the exception: UserAlreadyExistsException
         * @param idUser : user's identifier, in the form of a String
         * @param email : user's email
         * @param phoneNumber : user's phone number
         * @param name : user's name
         * @param nationality : user's nationality
         * @param address : user's address
         *@throws UserAlreadyExistsException : exception with the
       *message: Utilizador existente.
         */
        public void insertUser(String idUser, String email, String phoneNumber, String name, String
        nationality, String address) throws UserAlreadyExistsException;

        /**
         * it updates some of the user's information, like the email, phone
         *number and address. In case there isn't a user with the given
         *identifier in the system,
         * it throws the exception : UserDoesNotExistException
         * @param idUser : user's identifier
         * @param email : user's email
         * @param phoneNumber : user's phone number
         * @param address : user's address
         *@throws UserDoesNotExistException : exception with the
         *message: Utilizador inexistente.
         */
        public void updateUser(String idUser, String email, String
        phoneNumber,String address) throws UserDoesNotExistException;

        /**
         * it removes a user with the given identifier from the system. In
         *case there isn't a user with the given idUser in the system, it
         *throws the exception:
         * UserDoesNotExistException. In case the user is an owner of a
         *home in the system it throws the exception:
```

```
     *UserIsOwnerException
     * @param idUser : user's identifier
     * @throws UserDoesNotExistException : exception thrown with
     *the message: Utilizador inexistente.
     * @throws UserIsOwnerException : exception thrown with the
     *message: Utilizador e proprietario.
     */
    public void removeUser(String idUser) throws
UserDoesNotExistException, UserIsOwnerException;

    /**
     * function to get the user with the given identifier. In case there
     *isn't a user with the identifier received as argument, it throws the
     *exception:
     * UserDoesNotExistException
     * @param idUser : user's identifier
     * @return object user from the class UserClass
     * @throws UserDoesNotExistException : exception thrown with
     *the message: Utilizador inexistente.
     */
    public User getUser(String idUser) throws UserDoesNotExistException;

    /**
     * it inserts a home into the system with the arguments: idHome,
     *idUser, price, numberOfPeople, location, description, address. In
     *case the arguments price
     * and/or numberOfPeople is/are negative and/or the argument
     *numberOfPeople is above 20, it throws the exception:  *InvalidDataException. In case there
     * isn't
     * an user with the given identifier in the system the exception
     *UserDoesNotExistException is thrown. Lastly, in case there is
     *already a home with the
     *given identifier idHome in the system, the exception
     *HomeAlreadyExistsException is thrown.
     * @param idHome : home's identifier
     * @param idUser : user's identifier
     * @param price : price of a stay at the hom
     *@param numberOfPeople : maximum number of people the        *home can have
     * @param location : location of the home
     * @param description : description of the property
     * @param address : address of the property
     * @throws InvalidDataException : exception thrown with the
     *message: Dados invalidos.
     * @throws UserDoesNotExistException : exception thrown with the
     *message: Utilizador inexistente.
     * @throws HomeAlreadyExistsException : exception thrown with
     *the message: Propriedade existente.
     */
    public void addHome(String idHome, String idUser, int price, int
    numberOfPeople, String location, String description, String
    address) throws InvalidDataException,
    UserDoesNotExistException,
```

```
HomeAlreadyExistsException;

/**
 *it removes the home with the given identifier received as
 *argument from the system. In case there isn't a home with the
 *given identifier, the exception
 * HomeDoesNotExistException is thrown. In case the home was
 *previously visited, the exception       *HomeWasAlreadyVisitedException is thrown.
 * @param idHome : home's identifier
 * @throws HomeDoesNotExistException : exception thrown with*the  message:  Propriedade
 * inexistente.
 * @throws HomeWasAlreadyVisitedException : exception thrown
 * with the message: Propriedade ja foi visitada.
 */
public void removeHome(String idHome) throws
HomeDoesNotExistException, HomeWasAlreadyVisitedException;

/**
 * function to receive a certain home as an object, with the given
 *identifier. In case there isn't a home with the given identifier in the
 *system,
 * the exception HomeDoesNotExistException is thrown.
 * @param idHome : home's identifier
 * @return object home of class HomeClass
 * @throws HomeDoesNotExistException : exception thrown with
 *the message: Propriedade inexistente.
 */
public Home getHome(String idHome) throws
HomeDoesNotExistException;

/**
 * it adds a trip to a user, in a certain home. It receives idUser,
 * idHome and points as an argument. In case the argument points
 * is negative, the exception
 * InvalidDataException is thrown. In case there isn't a user with
 * the given identifier in the system, the exception
 * UserDoesNotExistException is thrown.
 * In case there isn't a home with the given identifier in the system,
 * the exception HomeDoesNotExistException is thrown. Lastly, in *case the person
 * travelling is also the owner of the home, the exception
 *TravellerIsOwnerException is thrown.
 * @param idUser : user's identifier
 * @param idHome : home's identifier
 * @param points : number of points added to the home (optional)
 * @throws InvalidDataException : exception thrown with the
 * message: Dados invalidos.
 * @throws UserDoesNotExistException : exception thrown with the
 * message: Utilizador inexistente.
 * @throws HomeDoesNotExistException : exception thrown with
 * the message: Propriedade inexistente.
 * @throws TravellerIsOwnerException : exception thrown with the
 * message: Viajante e o proprietario.
```

```
 * @throws TravellerIsNotOwnerException
 */
public void addTrip(String idUser, String idHome, int points) throws
InvalidDataException, UserDoesNotExistException,
HomeDoesNotExistException,
TravellerIsOwnerException, TravellerIsNotOwnerException;

/**
 * it lists every home in the system owned by a certain user. In
 *  case there isn't a user with the given identifier in the system, it
 * throws the exception
 * UserDoesNotExistException. Lastly, in case the user with the
 * given identifier doesn't own a home, the exception
 * UserIsNotOwnerException is thrown.
 * @param idUser : user's identifier
 * @return object home of the class HomeClass
 * @throws UserDoesNotExistException : exception thrown with
 * the message: Utilizador inexistente.
 * @throws UserIsNotOwnerException : exception thrown with the
 * message: Utilizador nao e proprietario.
 */
public java.util.Iterator<Home> listHomes(String idUser) throws
UserDoesNotExistException, UserIsNotOwnerException;

/**
 * it lists homes which are in a certain location and can fit a certain
 * number of people. It throws the exception InvalidDataException
 * when the numberOfPeople
 * is above 20 (and then it's negative). It throws the exception
 * NoResultsException when there isn't a single result found.
 * @param numberOfPeople : number of people the user wishes to
 * stay at a home
 * @param location : location of the desired home
 * @return HomeClass object
 * @throws InvalidDataException : exception thrown with the
 * message: Dados Invalidos.
 * @throws NoResultsException : exception thrown with the
 * message: Pesquisa nao devolveu resultados.
 */
public java.util.Iterator<Entry<Integer, Map<String, Home>>>
searchHome(int numberOfPeople, String location) throws
InvalidDataException, NoResultsException;

/**
* it lists the best home in a given location (with more points). The
* exception NoResultsException is thrown when there are no
* homes in the location.
* @param location : location you wish to search for homes
* @return HomeClass object
* @throws NoResultsException : exception thrown with the
* message: Pesquisa nao devolveu resultados.
 */
```

```
        public java.util.Iterator<Entry<Integer, Map<String, Home>>>
        listBest(String location) throws NoResultsException;

        /**
        * it lists every stay of a given user. The exception
        * UserDoesNotExistException is thrown when there isn't a user with *the given identifier
        * idUser in the
        * system. The exception UserHasNotTravelledException is thrown *when the user hasn't
        * traveller (hadn't had any stays).
        * @param idUser : user's identifier
        * @return iterator of stack stays
        * Therefore, the generated list is the number of times the user has
        * stayed at the only home.
        * @throws UserDoesNotExistException : exception thrown with the
        * message: Utilizador inexistente.
        * @throws UserHasNotTravelledException : exception thrown with
        *the message: Utilizador nao viajou.
         */
        public Iterator<Home> listStays(String idUser) throws
        UserDoesNotExistException, UserHasNotTravelledException;

}
```

As estruturas de dados utilizadas foram:

- ➔ Foi usado um TreeMap<K,V>() porque permite realizar pesquisas por chave - key(K) – que neste caso é o identificador da casa (idHome). A complexidade temporal vai ser O(log n) para a inserção (add()), remoção (remove()) e O(1) para a pesquisa – iterador.
- ➔ Foi usada uma Hashtable<K, V>() em que, a key é o idUser (identificador do utilizador) e o value é o próprio utilizador (objeto da classe user). A complexidade temporal vai ser O(n) para a inserção, remoção e O(1) para a pesquisa – iterador.

Descrição das Operações descritas em 3.3 (efetuadas sobre os TAD)

- **insertUser** - Verifica se já existe um utilizador na tabela com a mesma chave (idUser) que é fornecida. Caso isso aconteça lança a exceção UserAlreadyExistsException com a mensagem “Utilizador existente.”.
  Caso contrário insere um novo utilizador na tabela users (Hashtable<K,V>) com o idUser dado, o email, o número de telefone, o nome, a nacionalidade e a morada. Na tabela a key vai ser o idUser e o value vai ser o próprio user (User).

- **updateUser** - Verifica se existe na tabela de utilizadores algum com a mesma chave (idUser) que é fornecida. Caso não exista lança a exceção UserDoesNotExistException com a mensagem “Utilizador inexistente.”. Caso contrário, procura na tabela users (Hashtable<K,V>) o utilizador com a chave igual ao idUser fornecido e altera a informação desse utilizador.

- **removeUser** - Verifica se existe na tabela de utilizadores algum com a mesma chave (idUser) que é fornecida. Caso não exista lança a exceção UserDoesNotExistException com a mensagem “Utilizador inexistente.”. Depois verifica se o utilizador é o proprietário de alguma das propriedades existentes no sistema. Caso nenhuma das exceções seja lançada, o utilizador vai ser procurado, usando o seu idUser, na tabela users (Hashtable<K,V) e vai ser removido.

- **getUser** - Verifica se existe na tabela de utilizadores algum com a mesma chave (idUser) que é fornecida. Caso não exista lança a exceção UserDoesNotExistException com a mensagem “Utilizador inexistente.”. Caso contrário, procura na tabela users (Hashtable<K,V>) o utilizador com a chave igual ao idUser fornecido e retorna esse utilizador.

- **addHome** - Verifica se o preço e o número de pessoas introduzidos estão de acordo com os parâmetros impostos para que a casa possa ser adicionada, Se não estiverem lança a exceção InvalidDataException com a mensagem “Dados inválidos.”. De seguida, verifica se existe na tabela de utilizadores algum com a mesma chave (idUser) que é fornecida. Caso não exista lança a exceção UserDoesNotExistException com a mensagem “Utilizador inexistente.”. Por último, verifica se no mapa homes (TreeMap<K,V>) já existe alguma propriedade com o idHome fornecido. Caso exista lança a exceção HomeAlreadyExistsException com a mensagem “Propriedade existente.”.
  Se nenhum destes cenários se verificar, vai ser chamado o método getUser

para saber qual é o utilizador que é proprietário da casa, vai ser criado um novo objeto do tipo home com os dados fornecidos e e esse objeto vai ser atribuído ao user e adicionado no mapa homes (TreeMap<K,V>).

- **removeHome** - Verifica se no mapa homes (TreeMap<K,V>) existe alguma propriedade com o idHome fornecido. Caso não exista lança a exceção HomeDoesNotExistException com a mensagem “Propriedade inexistente.”. De seguida, verifica se a casa que se pretende remover já foi visitada alguma vez. Caso tenha sido é lançada a exceção HomeWasAlreadyVisitedException com a mensagem “Propriedade ja foi visitada.”.
  Se nenhum destes cenários se verificar, vai ser procurada no mapa homes a casa com o idHome fornecido, vai ser removida ao proprietário sendo depois removida do mapa homes (TreeMap<K,V>).

- **getHome** - Verifica se no mapa homes (TreeMap<K,V>) existe alguma propriedade com o idHome fornecido. Caso não exista lança a exceção HomeDoesNotExistException com a mensagem “Propriedade inexistente.”. Caso contrário, procura no mapa homes (TreeMap<K,V>) a casa com idHome igual ao fornecido e retorna-a.

- **addTrip** - Verifica se o preço e o número de pessoas introduzidos estão de acordo com os parâmetros impostos para que a casa possa ser adicionada, Se não estiverem lança a exceção InvalidDataException com a mensagem “Dados inválidos.”. De seguida, verifica se existe na tabela de utilizadores algum com a mesma chave (idUser) que é fornecida. Caso não exista lança a exceção UserDoesNotExistException com a mensagem “Utilizador inexistente.”. Depois verifica se no mapa homes (TreeMap<K,V>) existe alguma propriedade com o idHome fornecido. Caso não exista lança a exceção HomeDoesNotExistException com a mensagem “Propriedade inexistente.”. Seguidamente, verifica se o numero de pontos fornecido é maior que zero ou se é igual a zero. Se for maior e o dono da casa com o idHome dado for o utilizador com o idUser dado lança a exceção TravellerIsOwnerException com a mensagem “Viajante e o proprietario.”. Se for igual a zero e o utilizador for dono de alguma casa verifica se essa casa é a casa com o idHome fornecido. Se não for lança a exceção TravellerIsNotOwnerException com a mensagem “Viajante nao e o proprietario.”
  Se nenhum destes cenários se verificar, adiciona a estadia à casa, adicionando o número de pontos aos pontos que a casa já tinha e adiciona a estadia ao viajante, utilizando como parâmetro a casa.

- **listHomes** - Verifica se existe um utilizador registado no sistema com a mesma chave (idUser) que a fornecida. Para isto, faz uma pesquisa na tabela (users). Se não existir um utilizador com a mesma chave então lança a exceção UserDoesNotExistException com a mensagem de erro “Utilizador inexistente”. De seguida, vai verificar se o utilizador com a chave correspondente é proprietário de alguma casa no sistema. Para isto, devolve a String booleana isHost. Caso seja falsa, lança a exceção UserIsNotOwnerExcetion, com a mensagem de erro “Utilizador não é proprietário”. Finalmente, caso nenhuma destas condições se verifique, devolve um iterador do mapa das casas do utilizador (TreeMap<K,V>).

- **listStays -** Verifica se existe um utilizador registado no sistema com a mesma chave (idUser) que a fornecida. Para isto, faz uma pesquisa na tabela (users). Se não existir um utilizador com a mesma chave então lança a exceção UserDoesNotExistException com a mensagem de erro “Utilizador inexistente”. De seguida, vai verificar se o utilizador em questão já realizou estadias nalguma casa do sistema. Para isto, devolve uma variável boolean isTraveller. Se false, então lança a exceção UserHasNotTravelledException com a mensagem de erro “Utilizador não viajou.”. Finalmente, caso nenhuma destas condições se verifique, devolve um iterador da lista de estadias do utilizador (Stack stays).

- **searchHome -** Verifica se os dados inseridos são válidos, ou seja, se o número de pessoas que vão ficar na casa é inferior a vinte e é um valor positivo (maior que zero). Se não for, lança a exceção InvalidDataException com a mensagem de erro “Dados inválidos.”. De seguida, realiza a pesquisa de casas que melhor correspondem às necessidades do utilizador do sistema, através do método auxiliar search(). Finalmente, se o resultado da pesquisa for vazio lança a exceção NoResultsException com a mensagem de erro “Pesquisa não devolveu resultados.”. Caso contrário, o método devolve um iterador do mapa gerado que contém as casas resultantes da pesquisa.

- **listBest -** Este último método vai criar um mapa dentro de um mapa (TreeMap<K,TreeMap<K,V>>) em que o primeiro K vai ser o número de pontos que as casas constituintes do segundo mapa têm enquanto que o segundo K corresponde ao identificador de cada casa (idHome). Isto vai permitir que, quando hajam empates em relação ao número de pontos, consiga-se listar na mesma as diversas casas, algo que não seria possível com apenas um mapa. Como tal, este método vai percorrer todos os elementos do mapa original de casas (homes) e colocá-las, uma a uma, na posição respetiva do mapa. No final deste processo, verifica-se se o mapa gerado está vazio. Se sim, lança a exceção NoResultsException com a mensagem de erro “Pesquisa não devolveu resultados.” Se não, devolve um iterador do mapa gerado como EntrySet().

Estudo das Complexidades

Complexidade Temporal:

nUtilizadores - número de utilizadores registados no sistema e presentes na tabela que armazena os mesmos;
nCasas - número de casas registadas no sistema e presentes no mapa que armazena as mesmas;
nEstadias - número de estadias registadas para cada utilizador no sistema e presentes na lista que armazena as mesmas;

- **insertUser**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log Utilizadores)
Inserir um utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log Utilizadores)
Total	**O(1)**	**O(nUtilizadores)**	**O(log nUtilizadores)**

- **hasUser**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Total	**O(1)**	**O(nUtilizadores)**	**O(log nUtilizadores)**

- **updateUser**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Procurar o utilizador na tabela com a dada chave (idUser)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Total	**O(1)**	**O(nUtilizadores)**	**O(log nUtilizadores)**

- **removeUser**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Procurar na tabela o utilizador com a chave (idUser) correspondente	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Verificar se o utilizador é proprietário (Devolve uma variável booleana - isHost)	O(1)	O(1)	O(1)
Remover o utilizador da tabela	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Total	**O(1)**	**O(nUtilizadores)**	**O(log nUtilizadores)**

- **getUser**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Procurar na tabela o utilizador com a chave (idUser) correspondente.	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Devolve uma String com as características do utilizador	O(1)	O(1)	O(1)
Total	**O(1)**	**O(nUtilizadores)**	**O(log nUtilizadores)**

- **addHome**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar os dados fornecidos acerca da casa	O(1)	O(1)	O(1)
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Verificar a existência da casa no mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Inserir uma casa no mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Total	**O(1)**	**O(nUtilizadores + nCasas)**	**O(log nUtilizadores * nCasas)**

- **removeHome**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência da casa no mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Procurar a casa com a chave (idHome) correspondente	O(1)	O(nCasas)	O(log nCasas)
Remover a casa do mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Total	**O(1)**	**O(nCasas)**	**O(log nCasas)**

- **existsHome**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência da casa no mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Devolve as características da casa	O(1)	O(1)	O(1)
Total	**O(1)**	**O(nCasas)**	**O(log nCasas)**

- **getHome**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência da casa no mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Devolve uma String com as características da casa em questão	O(1)	O(1)	O(1)
Total	**O(1)**	**O(nCasas)**	**O(log nCasas)**

- **addTrip**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar os dados fornecidos	O(1)	O(1)	O(1)
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Verificar a existência da casa no mapa (homes)	O(1)	O(nCasas)	O(log nCasas)
Verificar se o viajante não é o proprietário quando se tenta adicionar uma estadia de proprietário	O(1)	O(1)	O(1)
Adicionar uma estadia à lista do utilizador que a realiza	O(1)	O(1)	O(1)
Total	**O(1)**	**O(nUtilizadores + nCasas)**	**O(log nUtilizadores * nCasas)**

- **listHomes**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Verificar se é proprietário (devolve uma variável boleana isHost)	O(1)	O(1)	O(1)
Devolve um iterador das propriedades do utilizador com a chave (idUser) fornecida	O(1)	O(nCasas)	O(log nCasas)
Total	**O(1)**	**O(nUtilizadores + nCasas)**	**O(log nUtilizadores * nCasas)**

- **listStays**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar a existência do utilizador na tabela (users)	O(1)	O(nUtilizadores)	O(log nUtilizadores)
Verifica se o utilizador já viajou (Devolve uma variável boleana isTraveller)	O(1)	O(1)	O(1)
Total	**O(1)**	**O(nUtilizadores)**	**O(log nUtilizadores)**

- **searchHome**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Verificar os dados fornecidos	O(1)	O(1)	O(1)
Devolve um iterador com a pesquisa realizada (método auxiliar search())	O(1)	O(1)	O(1)
Total	**O(1)**	**O(1)**	**O(1)**

- **listBest**

Ação	Melhor Caso	Pior Caso	Caso Esperado
Devolve um iterador	O(1)	O(1)	O(1)
Total	**O(1)**	**O(1)**	**O(1)**

Complexidade Espacial:

dimA – dimensão da tabela(Hashtable<K,V>) que contém os utilizadores, sendo que cada utilizador tem uma árvore (TreeMap<K,V>) de casas(com a dimensão de nCasas) e uma Stack (StackInList<V>) de estadias(com a dimensão de nEstadias). Logo uma estrutura que depende do número total de utilizadores(O(nUtilizadores)) e duas estruturas que dependem do número de casas/estadias de um certo utilizador(O(nEstadias + nCasas)).
dimB – dimensão do mapa(TreeMap<K,V>) que contém as casas, logo uma estrutura que depende do número total de casas(O(nCasas)).
A complexidade espacial do programa é dada por:

O(nUtilizadores) + O(nEstadias + nCasas) + O(nCasas) <=>
O(dimA) + O(dimB)