NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO
DE INFORMÁTICA

DEPARTMENT OF
COMPUTER SCIENCE

Relatório de Projeto

"*Uni Bedrooms* - Residências de estudantes"

Cadeira de Algoritmos e Estruturas de Dados, 2022/2023

Realizado por:

Rodrigo Santos, 63263
Gonçalo Amorim, 63470

Índice

1. Introdução

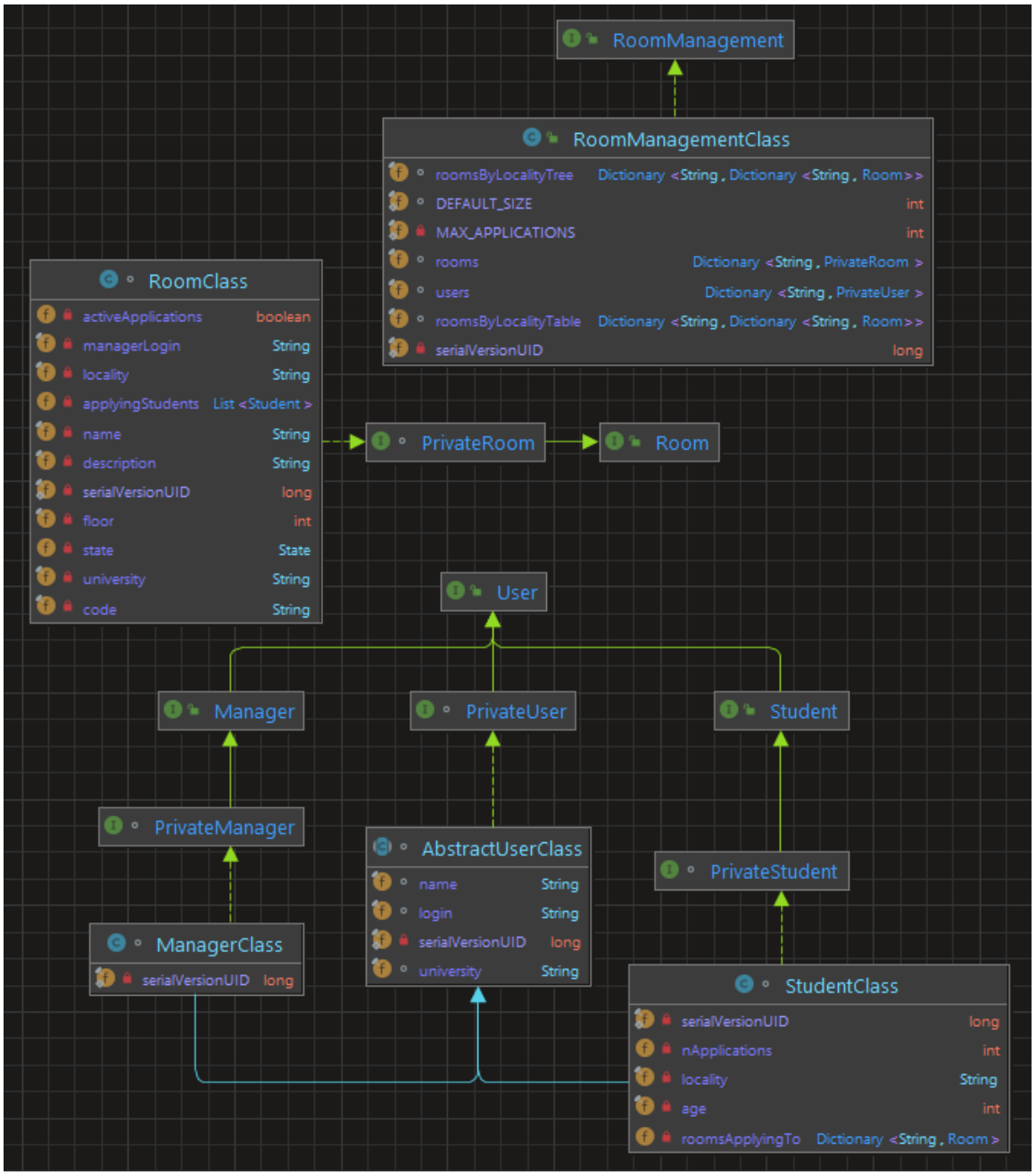
No âmbito da disciplina de Algoritmos e Estruturas de Dados do 2º ano de Licenciatura Informática, foi-nos proposta a realização do projeto **"Uni Bedrooms - Residências de Estudantes"** de forma a aplicar os conhecimentos adquiridos ao longo do decorrer das aulas práticas e teóricas.

Este trabalho tinha como objetivo principal o desenvolvimento de um sistema de arrendamento de quartos em residências de estudantes de universidades da União Europeia. O programa destina-se a ser usado por dois tipos de utilizadores: os **estudantes** universitários **(*Students*)** e os **gerentes** das residências **(*Managers)***. Os estudantes podem candidatar-se a um **quarto** em várias residências da sua escolha e não estão limitados a quartos de residências da sua universidade. Os gerentes usam o sistema para introduzir quartos (que são de residências das suas universidades), introduzir vagas nesses quartos, e escolher a qual dos estudantes candidatos quer atribuir um determinado quarto.

O presente relatório destina-se à descrição detalhada de todo o processo de desenvolvimento do projeto, bem como à justificação de algumas das decisões tomadas relativamente à maximização da eficiência temporal e espacial do programa. Inclui ainda uma breve conclusão sobre a utilidade deste trabalho na melhor compreensão dos temas relevantes à realização do mesmo.

2. Estrutura, hierarquia, e funcionamento do projeto

Abaixo encontra-se um diagrama de classes que visa dar um melhor entendimento da hierarquia das classes usadas no projeto, bem como da forma como estas se interligam. Como é possível interpretar pelo mesmo, todas as interfaces, à exceção da Interface ***RoomManagement***, são estendidas por uma subinterface com o prefixo "***Private***". Este procedimento tem como objetivo a melhor divisão dos métodos declarados pelas interfaces, evitando a partilha desnecessária de informação com o exterior das classes que as implementam. Os métodos "***set***", ou seja, que alteram de alguma forma os valores da classe, estão declarados na interface privada e só podem ser acedidos pela classe de topo, ficando assim inacessíveis diretamente pela classe main. Todos os restantes métodos estão declarados na classe pública e podem ser acedidos livremente.



2.1. Tipos Abstratos de Dados

2.1.1. Interface RoomManagement

É implementada pela classe ***RoomManagementClass***, que funciona como o sistema central do programa. Faz o tratamento dos dados e a comunicação entre a main e as restantes classes. Define 4 estruturas de dados e 2 constantes *int*:

1. `Dictionary<String, PrivateUser> users`
2. `Dictionary<String, PrivateUser> rooms`

   Definem dicionários implementados por classes ***SepChainHashTable***. São utilizados para guardar, respetivamente, todos os utilizadores e quartos registados no sistema.

3. `Dictionary<String,Dictionary<String,Room>>roomsByLocalityTable`
4. `Dictionary<String,Dictionary<String,Room>>roomsByLocalityTree`

   Definem dicionários, porém são implementados por classes ***SepChainHashTable*** e ***BinarySearchTree***, respetivamente. São ambas também usadas para guardar todos os quartos registados no sistema, mas organizados por localidade. O motivo para a criação de várias estruturas para guardar os "mesmos" dados será explicada em detalhe mais abaixo no documento.

5. `final int DEFAULT_SIZE = 50`

   Define o *default size* (tamanho pré-definido) das tabelas de dispersão mencionadas acima.

6. `final int MAX_APPLICATIONS = 10`

   Define o número máximo de candidaturas ativas que um estudante pode ter.

2.1.1.1. Estruturas de dados da classe

Muitas das funcionalidades suportadas pelo programa requerem a inserção, remoção, pesquisa e modificação do estado de quartos e utilizadores várias vezes, não sendo necessário considerar a sua organização no sistema. Assim, achámos conveniente utilizar tabelas de dispersão para guardar os dados descritos acima nas alíneas **1.**, **2.**, uma vez que a complexidade temporal da inserção, remoção, e pesquisa nesta estrutura é baixa e sempre constante ($O_{(1)}$). Nestas usamos como ***key*** o login do utilizador ou código de quarto e como ***value*** uma instância da classe ***StudentClass***, ***ManagerClass***, ***RoomClass***.

Sabemos ainda que outras duas funcionalidades do programa permitem listar os quartos em função da sua localidade:

- **Caso 1**: o utilizador escolhe uma localidade e o programa lista todos os quartos disponíveis nessa mesma localidade.
- **Caso 2**: o programa lista todos os quartos de todas as localidades organizados por ordem lexicográfica da localidade onde estes se encontram.

Note-se, ainda, que em ambos os casos é possível que exista mais de um quarto numa dada localidade, e pretende-se que estes sejam então listados por ordem lexicográfica dos respectivos códigos. Para dar resposta a essa necessidade, estas estruturas têm uma particularidade: Em ambas, cada entrada da tabela (alínea **3.**) ou nó da árvore (alínea **4.**) tem como ***key*** uma localidade e como ***value*** uma outra árvore de pesquisa binária cujos nós, por sua vez, contêm os quartos desta localidade como ***values*** e os respectivos códigos como ***keys***.

Procedemos desta forma aplicando a mesma lógica das alíneas **1.** e **2.** ao **Caso 1**: Como pretendemos apenas listar os quartos de uma única localidade, é irrelevante a forma como estas estão organizadas no sistema - interessa-nos apenas aceder à entrada que contém os quartos da localidade desejada o mais rápido possível. Usamos então a estrutura da alínea **3.**: uma tabela de dispersão, cuja complexidade temporal de pesquisa é baixa e constante ($O_{(1)}$).

Ao **Caso 2** aplicamos a lógica inversa: pretendemos listar os quartos por ordem lexicográfica da localidade em que se encontram, pelo que a organização dos elementos é crucial. Assim, usamos árvores de pesquisa binária para guardar os dados, uma vez que esta estrutura permite organizar os mesmos da forma pretendida. O mesmo acontece quando, independentemente do caso, existam vários quartos numa localidade e se pretenda listar os mesmos por ordem lexicográfica dos respectivos códigos de quarto.

Nas restantes alíneas estão definidas constantes correspondentes ao tamanho das tabelas de dispersão e ao número máximo de candidaturas ativas que um estudante pode ter, como descrito acima. Servem apenas para facilitar a interpretação do código e, caso seja necessário, uma eventual mudança dos valores definidos sem ter que alterar todos os métodos que os utilizam.

2.1.2. Interface User

É estendida pelas sub-interfaces ***PrivateUser***, ***Student*** e ***Manager***.

2.1.2.1. Interface PrivateUser

É implementada pela classe abstrata ***AbstractUserClass***, que é a classe mãe das subclasses ***StudentClass*** e ***ManagerClass***. Cada ***User***, seja ele ***Student*** ou ***Manager***, é um utilizador individual registado no sistema. Este possui um identificador único, um nome, e o nome da universidade que frequenta. Estes 3 elementos são definidos por constantes *String*:

1. `final String login`
   Inicializada no construtor como o identificador único do utilizador registado;

2. `final String name`
   Inicializada no construtor como o nome do utilizador registado;

3. `final String university`
   Inicializada no construtor como a universidade frequentada pelo utilizador registado.

2.1.2.2. Interface Student

É estendida pela interface ***PrivateStudent***, que por sua vez é implementada pela classe ***StudentClass***. Cada ***Student*** é um tipo específico de ***User*** com características particulares. Para além dos dados definidos na sua superclasse, possui ainda uma localidade de residência, uma idade, e um conjunto de quartos aos quais é atualmente candidato. Estes são definidos por uma constante *String*, uma constante e uma variável *int*, e uma estrutura de dados:

1. `final String locality`
   Inicializada no construtor como a localidade de residência do estudante registado.

2. `final int age`
   Inicializada no construtor como a idade do estudante registado.

3. `int nApplications`
   Variável usada para guardar o número de candidaturas ativas que o estudante tem.

4. `Dictionary<String, Room> roomsApplyingTo`
   Define um dicionário implementado por uma classe ***SepChainHashTable.*** É utilizada para guardar todos os quartos aos quais o estudante é atualmente candidato.

2.1.2.2.1. Estruturas de Dados da classe

Para fazer a gestão das candidaturas ativas de um estudante, é necessário guardá-las numa estrutura de dados. Uma vez que estas precisam apenas de ser inseridas, removidas, e pesquisadas, sem considerar a sua ordem de inserção, escolhemos usar uma ***SepChainHashTable.***

2.1.2.3. Interface Manager

É estendida pela interface ***PrivateManager***, que por sua vez é implementada pela classe ***ManagerClass***. Cada ***Manager*** é um tipo específico de ***User***, porém não precisa de mais dados para além daqueles que foram definidos na sua superclasse.

2.1.3. Interface Room

É estendida pela interface ***PrivateRoom***, que por sua vez é implementada pela classe ***RoomClass***. Cada ***Room*** representa um quarto registado no sistema e possui um código único, um gerente associado, um nome de residência, a universidade a que pertence, a localidade onde se encontra, uma quantidade indefinida de estudantes candidatos, um andar e uma breve descrição, definidos através de 6 constantes *String*, uma estrutura de dados, uma constante *int*, uma variável *boolean*, e um *enum*.

1. `final String code`
   Definida no construtor como o código único do quarto.

2. `final String managerLogin`
   Definida no construtor como o login do gerente associado ao quarto.

3. `final String residenceName`
   Definida no construtor como nome do quarto.

4. `final String university`
   Definida no construtor como a universidade a que o quarto pertence.

5. `final String locality`
   Definida no construtor como a localidade onde o quarto se encontra.

6. `final String description`
   Definida no construtor como uma breve descrição do quarto.

7. `List<Student> applyingStudents`
   Define uma lista implementada por uma classe ***DoubleList***. É utilizada para guardar todos os estudantes atualmente candidatos ao quarto.

8. `final int floor`

Definida no construtor como o andar do quarto.

9. `boolean hasActiveApplications`

   Variável utilizada para verificar se o quarto tem algum estudante com uma candidatura ativa ao quarto.

10. `State state`

    Enum que guarda os dois estados possíveis do quarto: “OCUPADO” ou “LIVRE”.

2.1.3.1. Estruturas de dados da classe

Pretende-se que o programa nos permita listar os estudantes que são atualmente candidatos a um determinado quarto por ordem cronológica da data de criação da candidatura. Como tal, optámos por usar uma ***DoubleList***, uma vez que esta estrutura nos permite organizar os dados pela ordem pertendida.

3. Estudo da Complexidade Temporal

3.1. Inserir estudante

Este comando insere um novo estudante no sistema desde que não exista outro usuário com o mesmo login. Para fazer esta verificação fazemos um find no dicionário ***users*** para averiguar a existência de um ***User*** com o mesmo login, caso não se verifique criamos um novo objeto com os dados fornecidos pela Main e inserimo-lo no dicionário.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג)	O(n)
Criação	O(1)	O(1)	O(1)
Inserção	O(1)	O(1 + ג)	O(n)

Total	**O(1)**	**O(1 + ג)**	**O(n)**

Legenda :

- n - Número total de usuários no dicionário. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.

3.2. Consultar dados do estudante

Este comando lista as informações de um estudante, desde que o estudante esteja inserido no sistema. Para verificar se o estudante está inserido no sistema, fazemos uma pesquisa no dicionário users. Se encontrarmos o estudante, retornamos o mesmo para a Main como um objeto de interface pública.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג)	O(n)

Total	**O(1)**	**O(1 + ג)**	**O(n)**

Legenda :

- n - Número total de usuários no dicionário. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.

3.3. Inserir gerente

Este comando insere um novo gerente no sistema desde que não exista outro usuário com o mesmo login. Para fazer esta verificação fazemos um find no dicionário ***users*** para averiguar a existência de um utilizador com o mesmo login. Caso tal não se verifique, criamos um novo objeto com os dados fornecidos pelo utilizador e inserimo-lo no dicionário.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג)	O(n)
Criação	O(1)	O(1)	O(1)
Inserção	O(1)	O(1 + ג)	O(n)

Total	**O(1)**	**O(1 + ג)**	**O(n)**

Legenda:

- n - Número total de usuários no dicionário. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.

3.4. Consultar dados de gerente

Este comando lista as informações de um gerente, desde que o gerente esteja inserido no sistema. Para verificar se tal acontece, fazemos uma pesquisa no dicionário ***users***. Se encontrarmos o gerente, retornamos o mesmo para a Main como um objeto de interface pública.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג)	O(n)

Total	**O(1)**	**O(1 + ג)**	**O(n)**

Legenda:

- n - Número total de usuários no dicionário. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.

3.5. Inserir novo quarto

Este comando insere um novo quarto no sistema, desde que:

- não exista outro quarto com o mesmo código.
- o login pertença a um gerente inserido no sistema
- a universidade fornecida coincidir com a universidade a qual o gerente está associado.

Para fazer esta verificação, fazemos uma pesquisa no dicionário ***rooms*** para averiguar a existência de um ***Room*** com o mesmo código, comparamos o login do gerente associado a este com o login do gerente dado pelo utilizador, e o nome da universidade fornecido com o nome da universidade a que o gerente pertence. Após as verificações, criamos um novo objeto com os dados fornecidos e inserimo-lo nos dicionários rooms, roomsByLocalityTree, roomByLocalityTable.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג) = z	O(n) = c
Criação	O(1)	O(1)	O(1)
Inserção rooms	O(1)	O(1 + ג) = z	O(n) = c
Pesquisar a localidade no dicionário roomsByLocalityTable	O(1)	O(1 + ג) = z	O(n1) = d
criação de uma BST	O(1)	O(1)	O(1)
inserção no resultado da pesquisa acima	O(1)	O(log(n2) = x	O(log(n2) = a
Inserção no roomsByLocalityTree	O(1)	O(log(n1) = y	O(log(n1) = b
Inserção no dicionário roomsByLocalityTable	O(1)	O(1 + ג) = z	O(n1) = d

Total	**O(1)**	**O(max(x,y,z))**	**O(max(a,b,c,d)**

Legenda:

- n - Número total de quartos no dicionário. (número de entradas dos 3 dicionários.)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.
- n1 - número de localidades.
- n2 - número de quartos na localidade

3.6. Consultar dados de quarto

Este comando lista as informações de um quarto, desde que o quarto esteja inserido no sistema. Para verificar se o quarto está inserido no sistema, fizemos uma pesquisa no dicionário rooms. Caso este seja encontrado, retornamo-lo para a Main como um objeto de interface pública.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג)	O(n)

Total	**O(1)**	**O(1 + ג)**	**O(n)**

Legenda:

- n - Número total de usuários no dicionário. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.

3.7. Modificar estado de um quarto

Este comando modifica o estado de um quarto, desde que

- o quarto exista no sistema,
- o login fornecido seja coincidente com o login do gerente do quarto,
- o quarto não tenha candidaturas ativas
- o novo estado pretendido seja “OCUPADO”.

Para fazer estas verificações, começamos por fazer uma pesquisa no dicionário ***rooms***. Em seguida, averiguamos se o login do gerente do quarto coincide com o login fornecido pelo utilizador, e por fim fazemos um *if statement* que averigua se o novo estado é “OCUPADO” e se as candidaturas estão ativas. Após concluirmos as verificações, mudamos o estado do quarto.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג)	O(n)
Mudança de estado	O(1)	O(1)	O(1)

Total	**O(1)**	**O(1 + ג)**	**O(n)**

Legenda:

- n - Número total de usuários no dicionário. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.

3.8. Remoção de um quarto

Este comando remove um quarto do sistema. Para proceder a esta operação, temos que verificar:

- se o quarto está no sistema,
- se o login inserido pelo utilizador corresponde ao login do gerente do quarto
- se o quarto tem candidaturas ativas.

Para tal, fazemos uma pesquisa no dicionário rooms, seguido de uma comparação entre o login inserido pelo utilizador e o login guardado no objeto room, e por fim verificando a variável *boolean* que nos diz se o quarto tem candidaturas ativas. Se tudo correr como esperado, o comando passa à execução: o quarto obtido na primeira pesquisa é removido das tabelas e árvores que guardam os quartos do sistema.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa	O(1)	O(1 + ג) = z	O(n) = c
Remoção do quarto de cada candidato.	O(1)	O(n1) = w	O(n1)
Remoção do quarto do dicionário rooms	O(1)	O(1 + ג)	O(n)
Pesquisar o quarto no dicionário roomsByLocalityTable	O(1)	O(1 + ג)	O(n2)
Remover o quarto da árvore binária (o resultado da pesquisa acima)	O(1)	O(log(n3)) = x	O(log(n3) =a
Remoção no roomsByLocalityTree	O(1)	O(log(n2) = y	O(log(n2) = b
Remoção no dicionário roomsByLocalityTable	O(1)	O(1 + ג)	O(n2)

Total	**O(1)**	**O(max (z,x,y,w))**	**O(max(a,b,c))**

Legenda:

- n - Número total de rooms no dicionário rooms. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.
- n1 - número de candidatos ao quarto.
- n2 - número de localidades.
- n3 - número de quartos com a mesma localidade que o quarto que vai se removido

3.9. Inserir candidatura

Este comando permite inserir uma nova candidatura de um estudante a um quarto. É feita uma pesquisa no dicionário ***users*** e ***rooms*** para encontrar o estudante e o quarto pretendidos, mas para o comando ser executado é necessário efetuar algumas verificações:

- se o estudante existe no sistema;
- se este já excedeu o limite máximo de candidaturas ativas;
- se o quarto existe no sistema;
- se este já se encontra ocupado;
- se o estudante já tem uma candidatura ativa a este quarto;

Caso alguma destas situações se verifiquem, é lançada a exceção correspondente ao erro, e caso contrário, o comando passa à execução. O estudante é inserido na lista de estudantes atualmente candidatos ao quarto, e o quarto é inserido na lista de quartos aos quais o estudante é atualmente candidato.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa pelo quarto	O(1)	O(1 + ג)	O(n)
Pesquisa pelo estudante	O(1)	O(1 + ג)	O(n1)
Inserção do estudante na lista de candidatos ao quarto	O(1)	O(1)	O(1)
Inserção do quarto na lista de quartos aos quais o estudante é candidato	O(1)	O(1)	O(n2)

Total	**O(1)**	**O(1 +** ג)	**O(max(n, n1, n2))**

Legenda:

- n - Número total de rooms no dicionário ***rooms***. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.
- n1 - Número total de users no dicionário ***users***. (número de entradas da tabela)
- n2 - Número de candidaturas de um estudante.

3.10. Aceitar candidatura

Este comando aceita uma candidatura a um quarto. Para que este comando tenha sucesso temos que verificar:

- se o quarto existe,
- se o login fornecido pelo utilizador corresponde ao login do gerente do quarto,
- se o estudante existe,
- se o estudante tem uma candidatura aquele quarto,

Para tal, fazemos pesquisa no dicionário ***rooms***, uma comparação entre o login dado e o login do gerente associado ao quarto, uma pesquisa no dicionário ***users***, e uma pesquisa na lista de quartos do estudante para verificar se este já é candidato ao quarto. Se as condições necessárias estiverem reunidas, o comando procede com a execução: Remove todas as candidaturas a outros quartos que o estudante possa ter e associa o estudante ao quarto pretendido mudando o estado para “OCUPADO” e removendo todas as candidaturas de outros estudantes.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa pelo quarto	O(1)	O(1 + ג)	O(n)
Pesquisa pelo estudante	O(1)	O(1 + ג)	O(n) = e
Remoção do estudante na lista de candidaturas dos quartos.	O(1)	O(n1 * n2) = a	O(n1 * n2) = c
Remoção do quarto da lista de candidaturas dos candidatos ao quarto.	O(1)	O(n2) = b	O(n2 * n1) = d

Total	**O(1)**	**O(a)**	**O(max(c,d, e))**

Legenda:

- n - Número total de rooms no dicionário rooms. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.
- n1 - número de candidaturas de um estudante.
- n2 - número de candidatos ao quarto.

3.11. Listar candidaturas ativas a um quarto

Este comando lista as candidaturas ativas a um quarto, devolvendo um iterador de quartos para a Main. Para o fazer temos que:

- verificar se o quarto existe,
- verificar se o login introduzido corresponde ao login do manager do quarto,
- verificar se o quarto tem alguma candidatura ativa.

Para tal é feita uma pesquisa no dicionário de quartos, uma comparação entre o login dado e o login do gerente associado ao quarto, e a verificação da variável *boolean* que nos indica se o quarto tem candidaturas ativas. Caso não existam problemas, o comando procede com a execução.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa pelo quarto	O(1)	O(1 + ג) = a	O(n) = c
Criação do iterador	O(1)	O(1)	O(1)
Iteração de quartos	O(n1)	O(n1) = b	O(n1) = d

Total	**O(n1)**	**O(max (a,b))**	**O(max(c,d))**

Legenda:

- n - Número total de rooms no dicionário rooms. (número de entradas da tabela)
- ג - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.
- n1 - número de candidatos ao quarto.

3.12. Listar todos os quartos

Este comando lista todos os quartos do sistema por ordem lexicográfica da localidade dos mesmos. Para o comando ter sucesso tem que haver quartos inseridos no sistema.

Operação	Melhor Caso	Caso Esperado	Pior Caso.
Criação do iterador	O(log(n))	O(log(n))	O(log(n))
Iteração de quartos	O(n)	O(n)	O(n)

Total	**O(n)**	**O(n)**	**O(n)**

Legenda:

- n - Número total de quartos.

3.13. Listar todos os quartos vagos numa localidade

Este comando lista os quartos vagos de uma dada localidade. Para tal acontecer a localidade tem que existir e tem que existir quartos vagos na localidade pretendida. Para chegarmos a essa conclusão executamos uma busca no dicionário roomsByLocalityTable.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**
Pesquisa pela localidade	O(1)	O(1 + ƛ)	O(n)
Criação do iterador	O(log(n2)	O(log(n2))	O(log(n2)
Iteração de quartos	O(n2)	O(n2)	O(n2)

Total	**O(n2)**	**O(n2)**	**O(n2)**

Legenda:

- n - Número total de localidades no dicionário roomsByLocalityTable. (número de entradas da tabela)
- ƛ - Fator de ocupação do dicionário, dado pela divisão do número de entradas da tabela pela dimensão da tabela.
- n2 - número de quartos com a localidade introduzidas.

3.14. Terminar a execução

Este comando termina a execução do programa.

Operação	**Melhor Caso**	**Caso Esperado**	**Pior Caso.**

Total	**O(1)**	**O(1)**	**O(1)**

4. Estudo da Complexidade Espacial

A complexidade espacial é a soma do tamanho ocupado por todas as estruturas de dados:

O(t1) + O(t2) + 2 * O(n1 * t5) + O(n2 * t4) + O(n3 * t3)

Legenda:

- t1 - tamanho da ***SepChainHashTable*** “users”.
- t2 - tamanho da ***SepChainHashTable*** “rooms”.
- t3 - tamanho da ***SepChainHashTable*** “roomsApplyingTo”.
- t4 - tamanho da ***DoubleList*** “applyingStudents”.
- t5 - tamanho da ***BinarySearchTree*** que guarda os quartos.
- n1 - número de localidades
- n2 - número de quartos
- n3 - número de pessoas

5. Conclusão

Tendo em conta a ideia base da disciplina - a aprendizagem das formas de utilização e funcionamento das várias estruturas de dados - sabíamos que o objetivo principal deste trabalho seria exatamente isso: fazer-nos pensar e refletir sobre a melhor forma de implementar as diferentes funcionalidades do programa, tendo sempre em mente a eficiência do código e poupança de tempo e recursos. Acreditamos que este objetivo foi alcançado na totalidade, e que somos agora capazes de desenvolver programas com melhor eficiência de tempo e memória.