Nota Importante: No final da parte prática do exame, deve submeter um ficheiro .zip com todos os source files que produziu, assim como o ficheiro "results.txt" gerado durante a execução do projeto. A não submissão destes ficheiros implicará uma nota nula.

Um grupo de *alumni* da FEUP conseguiu assegurar financiamento para um projeto de desenvolvimento de software para gerir agências imobiliárias. Este sistema será responsável por auxiliar os vendedores de cada agência a gerir os seus imóveis e fazer recomendações a potenciais clientes. Uma agência é representada pela classe **REAgency**.

Os clientes da agência, representados pela classe **Client**, são identificados pelo seu nome (*name*) e um contacto de e-mail (*eMail*). Um cliente pode estar interessado em um ou mais imóveis, *propertyListing*, representado pelo tuplo (morada, codigo postal, tipologia, preço), que são *strings*. A lista de imóveis nos quais o cliente está interessado é guardado num registo que assume a forma dum vetor *visitedProperties*, de tuplos (morada, codigo postal, tipologia, preço).

A classe **Property** define um imóvel e tem os membros-dado *ownerName, address, postalCode, typology e price,* representando o nome do proprietário, morada, código postal, tipologia e preço de venda do imóvel, respetivamente. É possível que vários imóveis tenham a mesma morada (devido a dados incompletos nos números de porta ou andar), sendo diferenciados pelo seu código postal e nome do proprietário, tipologia e preço (por essa ordem). É também possível que vários imóveis tenham o mesmo proprietário, devendo ser diferenciados pelo seu código postal e preço, por esta ordem. O membro-dado *reservation*, que é um tuplo contendo um apontador para um objeto da classe **Client** e um inteiro *reservationPrice* representando a proposta atual, guarda uma referência para o cliente que possa estar a negociar o preço de um imóvel num dado momento, tornando-o indisponível para ser visitado por outros clientes.

A classe **ClientRecord** encapsula um apontador para um objeto da classe Client, a fim de facilitar a gestão dos registos dos clientes da agência.

Um objeto da classe **PropertyTypeItem** é um item de catálogo para um determinado tipo de imóvel *typology* e código postal e mantém um registo de todos os imóveis na agência que obedeçam a estes parâmetros.

A classe **REAgency** mantém, naturalmente, referência para todos os imóveis que de momento estão anunciados/na base de dados da agência – reservados ou não – na Tabela de Dispersão *listingRecords*.

Para facilitar a consulta dos imóveis da agência, os objetos da classe **PropertyTypeItem** são mantidos numa Árvore Binária de Pesquisa (BST) (*typeItems*), enquanto a gestão dos imóveis é feita a partir de uma Tabela de Dispersão (*listingRecords*), de objetos da classe **ClientRecord**.

Numa tentativa de aumentar o volume de vendas e reter os clientes mais abastados, a agência segmenta os seus clientes usando uma Fila de Prioridade (*clientProfiles*) e, com uma determinada regularidade, recalcula a mesma e oferece sugestões de negócio aos clientes mais propensos a comprar.

As classes **Client**, **ClientRecord**, **Property**, **PropertyTypeItem** e **REAgency**, estão parcialmente definidas a seguir.

```
class Client {
   string name;
   string eMail;
   tuple<string, string, string, string>
                                 propertyListing;
   vector<tuple<string, string, string, string> >
                                  visitedProperties;
public:
   friend class ClientRecord;
   Client(string name, string eMail);
   bool operator<(const Client& c1) const;
};


class ClientRecord {
   Client* clientPtr;
public:
   ClientRecord(Client* client);
};


class Property {
   const string address;
   const string ownerName;
   const string postalCode;
   const string typology;
   const int price;
   tuple<Client*, int> reservation;
public:
   Property(string a, string o, string pc,
                          string t, int pr);
};
```

```
class PropertyTypeItem {
     string address;
     string postalCode;
     string typology;
     int maxPrice;
     vector<Property*> items; //all properties with same
               //postalCode typology and up to maxPrice
public:
     PropertyTypeItem(string address, string postalCode,
                           string typology, int maxPrice);
…
    bool operator<(const PropertyTypeItem &pti1) const;
    bool operator==(const PropertyTypeItem &pti1) const;
};


class REAgency {
     vector<Property*> properties;
     BST<PropertyTypeItem> catalogItems;
     HashTabClientRecord listingRecords;
     priority_queue<Client> clientProfiles;
public:
     REAgency();
     REAgency(vector<Property*> properties);
};
```

Nota importante! A correta implementação das alíneas seguintes, referentes à utilização de Árvores Binárias de Pesquisa, Tabelas de Dispersão e Filas de Prioridade, pressupõe a implementação dos operadores adequados nas classes e estruturas apropriadas.

a) [3 valores] Implemente na classe ***REAgency*** o membro-função

```
void REAgency::generateCatalog()
```

que adiciona ao catálogo *catalogItems* novos itens de catálogo para as propriedades do vector *properties*. Os itens do catálogo (objetos da classe *PopertyTypeItem*) estão organizados na BST alfabeticamente, pela morada, pelo código postal, pela tipologia e por um preço máximo. Pode haver propriedades com a mesma morada mas com código postal diferente – isto implicará itens distintos no catálogo. Cada item do catálogo agrupa no vector *items*, da classe *PropertyTypeItem*, todas as propriedades de mesma morada, código postal e tipologia.

b) [4 valores] Implemente na classe ***REAgency*** o membro-função

```
vector<Book*> REAgency::getAvailableProperties(Property* property) const
```

que retorna um vector com apontadores para todos as propriedades na mesma morada, mesmo codigo postal e mesma tipologia do argumento *property*, que estejam disponíveis. Uma propriedade da lista *items* de um *PropertyTypeItem* está disponível quando não tem associado qualquer entrada ao seu membro-dado *reservation*.

c) [3 valores] Implemente na classe ***REAgency*** o membro-função

```
void REAgency::reservePropertyFromCatalog(Property* property, Client* client,
int* percentage)
```

que procura através do catálogo *catalogItems* se haverá algum exemplar disponível na mesma morada, código postal e tipologia da propriedade *property*, passado como argumento. Se houver um exemplar disponível (o primeiro encontrado), o cliente *client* reserva aquele imóvel em particular com um valor inferior em percentagem *percentage*,passada por argumento ao preço pedido. Esta propriedade passará a estar indisponível. Se houver um imóvel disponível e a operação for bem-sucedida, a função retorna ***true***; caso contrário, a função retornará ***false***. Use o membro-função ***addVisiting***, da classe **Client**, para estabelecer a associação da propriedade ao cliente.

d) [2 valores] A fim de gerir melhor os clientes da agência, pretende-se guardar os seus registos (objetos da classe ***ClientRecord***) numa Tabela de Dispersão (membro-dado *listingRecords*), em que os utilizadores são reconhecidos pelos seus e-mails, que são únicos para cada utilizador (diferentes utilizadores poderão ter o mesmo nome, mas terão e-mails distintos). Implemente na classe ***REAgency*** o membro-função

```
void REAgency::addClientRecord(Client* client)
```

que insere na Tabela de Dispersão *listingRecords* um novo registo para o cliente *client* passado como argumento.

e) [1.5 valores] Com o objetivo de manter os registos dos clientes sempre atualizados, implemente na classe ***REAgency*** o membro-função

```
void REAgency::deleteClients()
```

que elimina da tabela de dispersão *listingRecords,* todos os clientes que nunca tenham visitado uma propriedade.

f) [2.5 valores] A agência pretende determinar quais deverão ser considerados os seus melhores clientes, ou seja, aqueles que possuírem o maior rácio de reservas por número de visitas. Para ordenar os candidatos, o clube utiliza uma *fila de prioridades* (membro-dado *clientProfiles*). Implemente na classe ***REAgency*** o membro-função

```
void REAgency::addBestClientProfiles(const vector<User>& candidates, int min)
```

que insere na *heap clientProfiles* apenas os candidatos aptos a receberem uma recomendação. Um candidato está apto a receber uma sugestão quando o seu rácio de reservas por número de visitas for superior a um valor *min, passado como argumento*. *O*rdene pelo preço da última propriedade visitada (decrescente).

g) [4 valores] Implemente na classe ***REAgency*** o membro-função

```
vector<Property*> REAgency::suggestProperties()
```

que devolve uma sugestão para cada cliente da fila de prioridades. A sugestão deve ser o imóvel que mais se assemelhe ao último imóvel que ele visitou a nível de código postal (menor diferença entre códigos postais) e que não esteja reservado. Caso o cliente não tenha ainda visitado nenhum imóvel, não deve fazer qualquer sugestão para esse mesmo cliente.