Os estudantes da FEUP organizaram-se para prestar serviços de consultadoria. A classe principal usada na implementação do sistema de informação para suportar essa atividade é **FEUPConsulting**.

A classe **Expertize** representa uma área em que os estudantes podem prestar serviços, registando a designação (*description*), valor por hora (*cost*) e também todos os estudantes com competências para prestar serviços nessa área e a esse custo/hora (*consultants*).

Um estudante é representado pela classe **Student**, é identificado pelo seu nome (*name*) e um contato de e-mail (*eMail*). Caso um estudante esteja envolvido num projeto, o nome desse projeto é registado em *currentProject*. Cada estudante pode estar, em cada momento, alocado a apenas um projeto. Os projetos em que o estudante já terminou a participação também são registados (*pastProjects*).

A classe **Project** representa um projeto encomendado à organização, com os membros-dado *title*, *expertize*, e *cost*, representando o título, a expertise que se pretende contratar e o valor por hora oferecido por hora, respetivamente. O membro dado *consultant*, que é um apontador para um objeto da classe **Student**, guarda a referência para o estudante atribuído ao projeto. Cada projeto é executado por um único estudante.

A classe **FEUPConsulting** contém no vetor *projects* uma lista de todos os projetos em curso ou terminados. Para facilitar a consulta das *expertizes* da organização, os objetos respetivos são mantidos numa Árvore Binária de Pesquisa (BST) (*expertizes*). A gestão dos estudantes é feita a partir de uma Tabela de Dispersão (*students*), de objetos da classe **StudentPtr**. A organização mantém um registo dos estudantes com um nível mínimo de atividade numa Fila de Prioridade (*activeStudents*).

As classes **Student**, **StudentPtr**, **Project**, **Expertize** e **FEUPConsulting**, estão parcialmente definidas a seguir.

```
class Expertize {
  string name;
  unsigned cost;
  vector<Student*> consultants;
public:
  Expertize(string name, unsigned cost);
  string getName() const;
  unsigned getCost() const;
  vector<Student*> getConsultants() const;
  void setConsultants(vector<Student*>
students);
  void addConsultant(Student* consultant);
    bool operator<(const Expertize &exp1)
const;
    bool operator==(const Expertize &exp1)
const;
};

class StudentPtr {
  Student* student;
public:
  StudentPtr(Student* student);
  string getName() const;
  void setEMail(string eMail);
  string getEMail() const;
};

class Project {
  const string title;
  const unsigned cost;
  const string expertize;
  Student* consultant;
public:
  Project(string title, string expertize,
unsigned price);
  string getTitle() const;
  string getExpertize() const;
  unsigned getCost() const;
  void setConsultant(Student* student);
  Student* getConsultant() const;
};
```

```
class Student {
  string name;
  string eMail;
  string currentProject;
  vector<string> pastProjects;
public:
  friend class StudentPtr;
  Student(string name, string eMail);
  string getName() const;
  string getEMail() const;
  void setEMail(string eMail);
  void addProject(string title);
  void closeProject();
  string getCurrentProject() const;
  vector<string> getPastProjects() const;
};

class FEUPConsulting {
  vector<Project*> projects;
  BST<Expertize> expertizes;
  HashTabStudentPtr students;
  priority_queue<Student> activeStudents;
public:
  FEUPConsulting();
  FEUPConsulting(vector<Project*> projects);
  void addProjects(vector<Project*> projects);
  vector<Project*> getProjects() const;
  // Part I   - BST
  void addAvailability(Student* student, string
expertize, unsigned cost);
  vector<Student*> getCandidateStudents(Project*
book) const;
  bool assignProjectToStudent(Project* project,
Student* student);
  // Part II  - Hash Table
  void addStudent(Student* user);
  void changeStudentEMail(Student* student, string
newEMail);
  // Part III - Piority Queue
  void addActiveStudents(const vector<Student>&
candidates, int min);
  int mostActiveStudent(Student& studentMaximus);
};
```

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 ***FEUPConsulting*** o membro-função

```
void FEUPConsulting::addAvailability(Student* s, string e, unsigned c)
```

que atualiza a BST *expertizes*, acrescentando o estudante *s* ao vetor de estudantes (*consultants*) da expertise de nome *e* e de custo *c*. Se não existir uma expertise com esse nome e custo, esta deve ser criada e acrescentada à BST. As expertises estão ordenadas na BST pelo seu nome (alfabeticamente) e pelo valor.

b) [**3 valores**] Implemente na classe ***FEUPConsulting*** o membro-função

```
vector<Student*> FEUPConsulting::getCandidateStudents(Project* project) const
```

que retorna um vetor com apontadores para todos os estudantes que estão disponíveis para projetos com as competências e preço oferecido no projeto *project*. Um estudante da lista *consultants* de uma *Expertize* está disponível quando não tem associado qualquer projeto ao seu membro-dado *currentProject*. Por simplicidade, assumimos que cada estudante só está disponível para trabalhar numa área por um único custo/hora, nem mais, nem menos.

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

```
bool FEUPConsulting:: assignProjectToStudent(Project* project, Student* student)
```

que atribui o estudante *student* ao projeto *project*. No entanto, a atribuição só pode ser feita se o projeto ainda não tiver um estudante associado e se o estudante estiver disponível e interessado (isto é, se estiver disponível para projetos nessa expertise e pelo valor respetivo). Esta última validação é feita na BST *expertizes*. Se a operação for bem-sucedida, a função retorna ***true***; caso contrário, a função retornará ***false***. Use o membro-função ***addProject*** da classe **Student** e o membro-função ***setConsultant*** da classe **Project** para estabelecer a associação entre o projeto e o estudante.

d) [**3 valores**] Os estudantes envolvidos na organização são registados como objetos da classe ***StudentPtr*** numa Tabela de Dispersão, *students*. A identificação dos estudantes é feita pelo endereço de e-mail (naturalmente único para cada estudante), sendo possível que estudantes diferentes tenham o mesmo nome. Implemente na classe ***FEUPConsulting*** o membro-função

```
void FEUPConsulting::addStudent(Student* student)
```

que insere na Tabela de Dispersão *students* um novo registo para o estudante *student*.

e) [**2,5 valores**] Implemente na classe ***FEUPConsulting*** o membro-função

```
void FEUPConsulting::changeStudentEMail(Student* student, string newEMail)
```

que permite ao estudante *student* atualizar o seu endereço eletrónico para *newEMail*.

f) [**2,5 valores**] Para manter o número de alunos envolvidos num número razoável, a organização mantém um registo daqueles que atingiram um número mínimo de projetos. Esse registo é guardado numa *heap* (membro-dado *activeStudents*), devendo estar no topo da *heap* o estudante com maior número de projetos terminados. Implemente na classe ***FEUPConsulting*** o membro-função

```
void FEUPConsulting::addActiveStudents(const vector<Student>& candidates, int min)
```

que insere na *heap activeStudents* os estudantes da lista de candidatos fornecida que terminou a participação num número de projetos maior do que o mínimo passado como parâmetro *min*. É importante notar que os projetos em curso não contam para esta definição de estudante ativo.

g) [**3 valores**] Implemente na classe ***FEUPConsulting*** o membro-função

```
int FEUPConsulting::mostActiveStudent(Student& studentMaximus) const
```

que identifica o estudante com participação terminada no maior número de projetos entre todos os estudantes. A função passa este estudante para o argumento *studentMaximus*, passado por referência, e retorna o número de estudantes ativos em competição. No caso de haver mais do que um estudante a partilhar a posição, o argumento *studentMaximus* não é modificado, e a função retorna 0 (zero).