Algoritmos e Estruturas de Dados – Projeto Individual

Implementação de tabelas de símbolos com árvores binárias de pesquisa

Descrição

As tabelas de símbolos são estruturas de dados para armazenamento de elementos que a cada elemento faz corresponder uma **chave** (***key***) e um **valor** (***value***). Podemos pensar nesta estrutura de dados como implementando precisamente uma tabela usual, e.g. uma tabela de *DNS lookup* que associe a cada URL o IP correspondente:

URL (*key*)	Endereço IP (*value*)
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

Tabela 1 - Exemplo de tabela com associação entre "chaves" e "valores"

As tabelas de símbolos estão munidas de, pelo menos, duas operações elementares:

- **put** – insere um novo par *(key, value)* na tabela, ou substitui o valor associado à chave caso esta já esteja presente na tabela;
- **get** – devolve o valor associado a uma chave da tabela, ou *null* caso esta não esteja presente.

Implementações eficientes de tabelas de símbolos geralmente assentam sobre a representação interna das mesmas utilizando **árvores binárias de pesquisa**, uma representação de estruturas de dados que considera a invariante de **cada nó ser maior do que cada nó na sua sub-árvore esquerda e menor do que cada nó na sua sub-árvore direita**. Por exemplo:

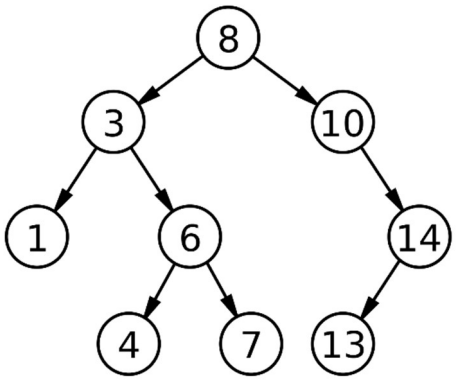

Figura 1 - Exemplo de árvore binária de pesquisa com números inteiros

As vantagens em termos de desempenho de uma representação em árvore binária de pesquisa advêm de que a eficiência temporal de operações na árvore é, em média, diretamente proporcional à altura da árvore.

O projeto individual de Algoritmos e Estruturas de Dados prevê a implementação eficiente de uma tabela de símbolos utilizando uma representação interna em árvore binária de pesquisa.

API

A implementação deve seguir a seguinte API.

```
public class ST<Key extends Comparable<Key>, Value>
       ST()                                 // Initialise an empty ordered symbol table
       void put(Key key, Value val)         // Put the key-value pair into this table
       Value get(Key key)                   // Get the value paired with key (or null)
       void delete(Key key)                 // Remove the pair that has this key
       boolean contains(Key key)            // Is there a value paired with the key?
       boolean isEmpty()                    // Is this symbol table empty?
       int size()                           // Number of key-value pairs in this table
       Key min()                            // Smallest key
       Key max()                            // Largest key
       Key floor(Key key)                   // Largest key less than or equal to key
       Key ceiling(Key key)                 // Smallest key greater than or equal to key
       int rank(Key key)                    // Number of keys less than key
       Key select(int k)                    // Get a key of rank k
       void deleteMin()                     // Delete the pair with the smallest key
       void deleteMax()                     // Delete the pair with the largest key
       int size(Key lo, Key hi)             // Number of keys in [lo, hi]
       Iterable<Key> keys(Key lo, Key hi)   // Keys in [lo, hi] in sorted order
       Iterable<Key> keys()                 // All keys in the table, in sorted order
```

Figura 2 - API de tabelas de símbolos com árvores de pesquisa binárias

A classe não deve estar contida em nenhum *package* ou, caso esteja, a declaração do *package* deve ser removida antes do momento da submissão.

Cada classe deve estar contida num ficheiro cujo nome é exatamente o mesmo da classe. Por exemplo, a classe MyClass está definida num ficheiro MyClass.java. (Isto é uma regra do próprio Java.)

Os modificadores de visibilidade, os nomes, o tipo dos parâmetros, e a ordem dos parâmetros das classes e métodos da API implementada devem corresponder **exatamente** aos apresentados na API acima. Isto é, a classe e todos os métodos da API devem ser **públicos** e ter exatamente os mesmos nomes que os apresentados acima, bem como os parâmetros definidos exatamente na mesma ordem e com os mesmos nomes. A assinatura de quaisquer métodos auxiliares não contemplados na API acima fica ao critério do aluno.

A representação dos nós internos da árvore binária deve ser feita através de uma classe interna ***Node*** que guarda uma chave ***key*** e um valor ***value***, bem como referências ***left*** e ***right*** aos nós filhos da esquerda e da direita, respetivamente. Deve ser guardada uma referência ***root*** ao nó raíz da árvore.

O não-cumprimento das indicações descritas acima pode implicar a classificação automática de 0 valores num método ou em toda a classe se a assinatura não corresponder à apresentada.

Critérios de Avaliação

O projeto individual será classificado com as notas A, B, C, ou D, sendo que a classificação obtida pode limitar a classificação máxima que poderá ser obtida na UC:

- **A**: a nota final não está limitada pelo projeto, podendo o aluno obter até 20 valores.
- **B**: a nota final fica limitada a 17 valores.
- **C**: a nota final fica limitada a 13 valores.
- **D**: o aluno fica automaticamente reprovado à avaliação periódica, e terá de ir a exame.

Cada submissão será testada e avaliada tendo em conta:

- A implementação dinâmica, i.e. através de nós ligados. A não-inclusão da classe ***Node*** pode implicar a automática classificação da submissão com 0 valores.
- A implementação genérica, i.e. deve ser possível criar e manipular tabelas de símbolos com vários tipos de dados, e.g. *<Integer, String>*, *<Double, Integer>*, *<String, Double>*, etc.
- A natureza comparável das chaves. As chaves não serem comparáveis entre si pode implicar a automática classificação da submissão em 0 valores.
- O correto resultado das invocações de todos os métodos da API em diferentes estados da tabela de símbolos, bem como a concordância entre os mesmos. Por exemplo (mas não só):
    - Se a tabela está vazia, *isEmpty* deve ser *true* e *size* deve ser 0, e vice-versa;
    - A invocação de *size(lo, hi)* deve produzir o mesmo resultado de *size()* quando *lo* e *hi* correspondem à menor e maior chaves, respetivamente;
    - Qualquer resultado de *select(k)* deve ter *rank* k, i.e. *rank(select(k)) == k*.

Submissão

Parte 1

A submissão da parte 1 do projeto corresponde à submissão do exercício de programação 10, na secção reservada para a mesma. A submissão 10 prevê a entrega do código apenas com a implementação das funções *min* e *max*. **Atenção: para testar estes métodos, devem estar implementados o construtor e o método *put*!**

Parte 2

A submissão deverá ser feita através da plataforma Moodle na secção reservada e corretamente identificada para a submissão do projeto individual.

Qualquer submissão deve seguir as seguintes regras:

- Devem ser apenas submetidos os ficheiros .java contendo o código-fonte relevante ao projeto.
- Caso o aluno necessite de submeter mais do que 1 ficheiro .java, estes devem ser agrupados na raíz de uma pasta .zip. Não são suportados ficheiros comprimidos noutros formatos, e não deve ser forçosamente alterada a extensão do ficheiro de outra para .zip.

O não-cumprimento das regras de submissão acima referidas pode implicar a imediata classificação do projeto individual em 0 valores.

Avisos

Para assegurar uma avaliação justa do seu código, tenha em conta que:

- Submissões que apresentem erros de compilação serão automaticamente classificadas com 0 valores, sem oportunidade de re-submissão, pelo que devem **sempre** tentar compilar e executar o código antes deste ser submetido.
- O lançamento de exceções inesperadas pelo código submetido e não contempladas nas implementações estudadas em aula implica uma penalização negativa na classificação do projeto.
- Quaisquer indícios de plágio serão apropriadamente investigados e, caso se confirme que um aluno cometeu plágio, este estará automaticamente reprovado à UC.