Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

ALGORITMOS E ESTRUTURAS DE DADOS

Relatório da 3ª Série de Problemas

Docente

Prof.ª Cátia Vaz

Alunos

36368 Mihail Cirja

43552 Samuel Costa

Março 2018

Índice

Introdução

Este relatório, referente ao terceiro de três trabalhos práticos, foi elaborado no âmbito da unidade curricular de Algoritmos e Estruturas de Dados e acompanha a resolução de uma série de exercícios. Esta série constituiu uma aplicação de técnicas de algoritmia, a implementação de iteradores "lazy", algoritmos em árvores binárias, tries e pesquisa em grafos.

Ao longo do documento apresenta-se a descrição das soluções propostas para dar resposta aos exercícios e, por fim, uma breve nota em que explicitam as conclusões sobre o trabalho realizado.

Exercício 1

Exercício 1.1

Foi possível a resolução do exercício por recurso a uma flag que indique se o próximo a obter é um número par, a um elemento inteiro *curr* e a um iterador *it*.

A lógica de obtenção do elemento seguinte do iterador *lazy* está contida no método *hasNext()* em que se avança para o próximo elemento. Sempre que o valor da *flag* for igual à paridade do elemento *curr*, quer dizer que esse é o próximo elemento a retornar.

No método *next()* é consumido o resultado e o elemento *curr* é colocado a NULL.

Exercício 1.2

Importa guardar referências para o elemento anterior e o actual. Sempre que se verifique que o resultado da comparação entre o anterior e o actual for menor que 0, então o elemento actual é um elemento válido da sequência.

No método *next()* o resultado é consumido, atribuindo-se a *prev* o valor de *curr*, a *curr* o valor *NULL*, e retornando *prev*.

Exercício 1.3

Recorrendo a dois iteradores *it* e *subIt* que iteram a lista e as sub-listas respetivamente, realizou-se o *flattern* do iterador, permitindo apresentar os elementos das sublistas como se pertencessem a uma única lista. O método *hasNext()* retorna true sempre que a string actual contiver a sub-string passada como parâmetro. O método *next()* consome o resultado e coloca *curr* a *null*.

Exercício 2

Exercício 2.1

```
public static Integer sumIf(Node<Integer> root, Predicate<Integer> predicate)
```

A soma dos elementos de uma árvore binária que verificam dado predicado é dada pela soma da raíz da árvore (no caso de este verificar o predicado), dos elementos da sub-árvore direita e da sub-árvore esquerda que o verificam.

Assim, identificou-se um caso base, que é aquele em que o nó root é igual a NULL. Nesse caso, o valor a retornar é zero. Nos demais casos, o valor a retornar é o valor item ou 0 mais o valor retornado pela chamada recursiva da função para a sub-árvore esquerda mais o valor retornado pela chamada recursiva da função para a sub-árvore direita.

Exercício 2.2

Trata-se de aplicar o algoritmo do predecessor ao máximo, i.e, o nó mais à direita na árvore. Aplicar o algoritmo do predecessor sucessivamente até o valor de retorno deste ser NULL, ie até ao mínimo, que é o elemento mais à esquerda na árvore.

Exercício 2.3

Uma árvore diz-se balanceada se a altura máxima entre as alturas dos seus nós folha é, no máximo 1. Recorde-se que, por definição, uma árvore vazia está balanceada. Assim, importa chamar recursivamente o algoritmo isBalanced para a sua sub-árvore direita e para a sub-árvore esquerda e verificar que o módulo da altura da sua subárvore esquerda menos a altura da sub-árvore direita é menor ou igual a um. Uma vez que o algoritmo é chamado recursivamente, o retorno propagado das chamadas à função nos nós de níveis inferiores resolve o valor de retorno final da função.

Exercício 3

Definiu-se a classe *TrieNode* com um campo booleano que indica que aquele nó é a terminação de um fragmento, e um inteiro *fragmentCount*, que armazena o número de fragmentos que têm como prefixo o fragmento terminado em *TrieNode*. A classe tem ainda um campo que é um array de TrieNode com quatro posições. Convencionou-se que a ordem de indexação é a seguinte:

- [0] diz respeito a 'a';
- [1] diz respeito a 'c';
- [2] diz respeito a 't';
- [3] diz respeito a 'g'.

Definiu-se a classe DNACollection, com uma referência para a raiz de uma Trie. Esta classe oferece suporte às operações add e prefixCount.

A operação **add** consiste em, relativamente aos nós da trie, avançar internamente nos arrays de children na posição apontada pelo primeiro caracter do fragmento e obtendo uma *sub-string* que diz respeito ao fragmento excluindo o primeiro caracter.

Para implementar a operação **prefixCount**, "navega-se" na trie, retornando-se o campo fragmentCount do nó terminal desse fragmento.

Exercício 4

$$\begin{array}{c} \boldsymbol{Stack} \\ [A] \\ A \end{array}$$

$$\begin{array}{cc} & \boldsymbol{Stack} \\ [A \quad C] & C \\ & A \end{array}$$

$$\begin{array}{cc} & \boldsymbol{Stack} \\ [A \quad C \quad D] & D \\ & C \\ & A \end{array}$$

$$[A \quad C \quad D \quad E\,] \qquad \begin{matrix}\boldsymbol{Stack}\\ E\\ D\\ C\\ A\end{matrix}$$

$$[A \quad C \quad D \quad E \quad G\,] \qquad \begin{matrix}\boldsymbol{Stack}\\ G\\ E\\ D\\ C\\ A\end{matrix}$$

$$[A \quad C \quad D \quad E \quad G \quad H\,] \qquad \begin{matrix}\boldsymbol{Stack}\\ H\\ D\\ C\\ A\end{matrix}$$

$$[A \quad C \quad D \quad E \quad G \quad H \quad F\,] \qquad \begin{matrix}\boldsymbol{Stack}\\ F\\ H\\ D\\ C\\ A\end{matrix}$$

$$[A \quad C \quad D \quad E \quad G \quad H \quad F \quad B\,] \qquad \begin{matrix}\boldsymbol{Stack}\\ B\\ F\\ H\\ D\\ C\\ A\end{matrix}$$

Exercício 5

A pesquisa em largura primeiro (*breadth-first-search*) num grafo implica a definição de um nó originário. A cada nó visitado são colocados numa queue os nós adjacentes. O próximo nó visitado é retirado da queue.

$$[A] \qquad \begin{matrix}\boldsymbol{Queue}\\ B\\ C\end{matrix}$$

$$[A \quad B] \qquad \begin{matrix}\boldsymbol{Queue}\\ C\\ F\end{matrix}$$

$$[A \quad B \quad C] \qquad \begin{matrix}\boldsymbol{Queue}\\ F\\ G\end{matrix}$$

$$[A \quad B \quad C \quad F\,] \qquad \begin{matrix}\boldsymbol{Queue}\\ G\end{matrix}$$

$$[A \quad B \quad C \quad F \quad G\,] \qquad \begin{matrix} \boldsymbol{Queue} \\ E \end{matrix}$$

$$[A \quad B \quad C \quad F \quad G \quad E\,] \qquad \begin{matrix} \boldsymbol{Queue} \\ - \\ - \end{matrix}$$

Conclusão

Durante a elaboração do trabalho (série e relatório), foi possível observar que a utilização de iteradores "*lazy*" possui a vantagem de percorrer uma única vez uma coleção sem necessitar de espaço adicional.

Foi também possível observar as vantagens de transformação numa abordagem que faça uso de operações em árvores, pois na maioria das vezes estas têm custo logarítmico.