



Universidade do Minho

## Projecto Java de Laboratório de Informática Relatório

Discentes:  
Axel Ferreira - a53064  
João Rua - a41841

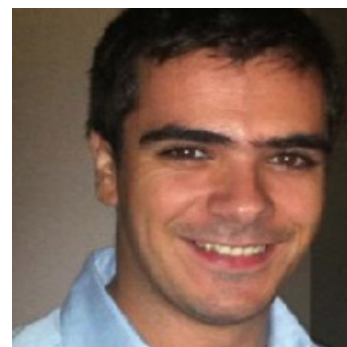
Docentes:  
F. Mário Martins  
João Miguel Fernandes  
João Luís Sobral

June 15, 2013

## Grupo



(a) nome : Axel Ferreira  
número : 53064  
mail : axelferreira@me.com



(b) nome : João Rua  
número : 41841  
mail : joaorua@gmail.com



## Contents

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Estrutura do Relatório . . . . .	3
<b>2</b>	<b>Classes e Estruturas de Dados</b>	<b>3</b>
2.1	Classes Criadas . . . . .	3
<b>3</b>	<b>Consultas Estatísticas</b>	<b>4</b>
3.1	Dados do último ficheiro lido . . . . .	4
3.2	Consultas Interativas . . . . .	4
<b>4</b>	<b>Consultas Interativas</b>	<b>4</b>
4.1	. . . . .	4
<b>5</b>	<b>Medidas de Performance</b>	<b>4</b>
<b>6</b>	<b>Modularidade</b>	<b>4</b>
6.1	parser . . . . .	6
6.2	fread . . . . .	6
6.3	fwrite . . . . .	7
6.4	F . . . . .	7
6.5	main . . . . .	8
<b>7</b>	<b>Conclusão</b>	<b>8</b>
<b>8</b>	<b>Interface</b>	<b>8</b>

### Abstract

Foi desenvolvido um programa no âmbito da U.C. de Laboratório de Informática III , capaz de utilizar o conteúdo processado pelo pelo anterior programa desenvolvido em C, também nesta UC, capaz de ler um ficheiro contendo um conjunto de autores e respetivos co-autores, e responder a alguns queries interativos sobre estes dados. Estes dados de autorias, e co-autorias são retirados do website [DBLP](#).

## 1 Introdução

No âmbito da unidade curricular Laboratórios de Informática III foi proposta a realização de um projeto que dá continuidade ao projeto anteriormente desenvolvido em C nesta mesma UC. Este novo projeto conta com essencialmente duas partes. A primeira diz respeito á leitura de dados de memória secundária e população de estruturas de dados em memória central, gravação destas estruturas de dados em memória persistente em modo binário, bem como a criação de alguns queries de forma a permitir uma consulta interativa aos dados. Desenvolveram-se ainda alguns métodos que permitem consultas sobre as estruturas de dados. A Segunda parte prevê o teste de performance do código e respetivas estruturas de dados criados na 1ª parte, relativamente a estruturas de dados alternativas. De forma a facilitar esta segunda parte, o grupo teve o cuidado de criar uma interface



cada vez que foi utilizada uma estrutura de dados do Java.Collections, de forma a permitir alterar as estruturas utilizadas alterando apenas, e se necessário, esta interface.

### 1.1 Estrutura do Relatório

Este relatório inicia-se com uma capa, incluindo o título do projeto, a data, a identificação dos autores e da equipa docente que acompanhou o projeto. Segue-se o Índice, o Abstract que resume o projeto, a Introdução ao mesmo (onde se explicita o objetivo a atingir) e a Estrutura do Relatório.

## 2 Classes e Estruturas de Dados

### 2.1 Classes Criadas

Na criação deste programa foram desenvolvidas alguns classes

- Anos - Esta classe armazena toda a estrutura de dados. Contém um TreeMap em que são inseridos todos os anos. Cada ano é inserido, utilizando como chave a String contendo a numeração do ano. A escolha do TreeMap deveu-se ao facto de manter a ordem do ano. Esta ordem facilita a impressão e travessia ordenada necessária para algumas queries.

```
private TreeMap<String,Ano> anos;
```

- Ano - Classe criada para guardar o conteúdo de cada Ano, contém uma String com o nome do ano e um HashMap com os Autores. Cada autor é inserido utilizando como chave a String com o nome do mesmo. A utilização do HashMap deve-se ao facto de não haver vantagem associada á ordem de armazenamento.

```
private String ano;  
private HashMap<String,Autor> autores;
```

- Autor - Classe criada para guardar o nome do Autor, numero de publicações, e rede de co-autores. Esta última é armazenada num HashMap de co-autores. Cada co-autor é inserido utilizando como chave a String com o nome do mesmo. A utilização do HashMap deve-se ao facto de não haver vantagem associada á ordem de armazenamento.

```
private String nome;  
private int artigos;  
private HashMap<String,Coautores> coautores;
```

- Coautor - Classe criada para guardar o conteúdo de cada co-autor, resumindo-se ao nome e numero de artigos publicados em comum com o respetivo autor.

```
private String nome;  
private int artigos;
```



- FileInput - Esta classe faz todo o parsing e leitura dos dados de ficheiros com que posteriormente as estruturas de dados são povoadas. Contém apenas como variável de classe o nome do ficheiro que deve ler sempre. Contém ainda um método de classe que devolve o nome do ficheiro. Bem como dois métodos que devolvem o conteúdo dos ficheiros.

```
public static final String ficheiro ="publicx.txt";
```

### 3 Consultas Estatísticas

#### 3.1 Dados do último ficheiro lido

Quando o ficheiro é lido, é apresentado no ecrã o nome do ficheiro, seguido do número total de artigos, e número total de nomes lidos, número total de nomes distintos bem como o intervalo fechado de anos em que os artigos foram lidos. É ainda apresentada alguma informação respeitante aos dados atuais da estrutura de dados, nomeadamente nº total de autores, nº total de artigos, de um só autor, e nº de autores que apenas publicaram a solo. Finalmente é ainda apresentada toda a sequência ordenada de anos seguido do respetivo número de publicações.

#### 3.2 Consultas Interativas

### 4 Consultas Interativas

#### 4.1

### 5 Medidas de Performance

Nesta fase foi efetuado o refactoring e implementado o buffer dinâmico. Garantindo-se desta forma que o fluxo do programa é centrado no main e que o buffer tem sempre tamanho suficiente para ler uma linha.

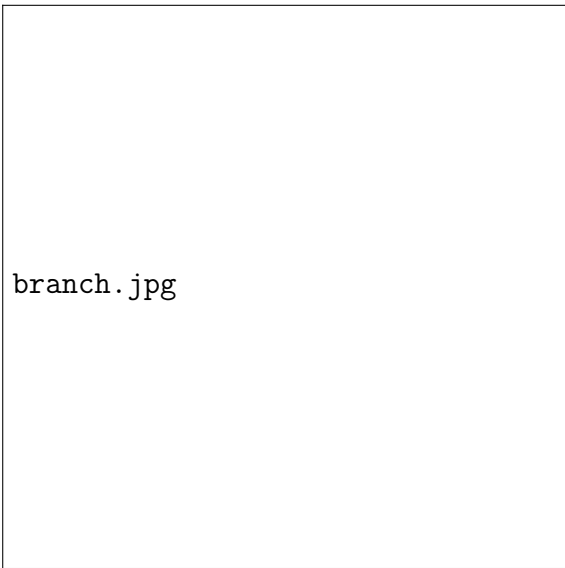
### 6 Modularidade

Após a primeira entrega, de forma a garantir a ocultação de dados e da implementação, a abstração de dados, o encapsulamento e alguma independência contextual, a arquitetura do programa sofreu um processo de refactoring tendo sido reformulados e criados novos módulos. Em cada módulo, os dados foram declarados como "static" tendo sido criados interfaces para mediar o manuseamento dos mesmos. Isto garante independência contextual, ocultação da implementação e dos dados. Estas interfaces foram posteriormente incluídas no programa main de forma a garantir que todo o fluxo do programa é controlado pelo módulo main. é possível ver a nova arquitetura dos módulos no diagrama abaixo.



Arquitetura.jpg

Toda a criação do programa, alteração e implementações experimentais foram feitas com recurso ao git com repositório no Github. O que permitiu a criação de branches para cada parte do projeto, bem como para experiências.



branch.jpg

Para além disto também foi com recurso ao Git que foi possível a compilação de uma versão pré-refactoring para comparação da performance.

### 6.1 parser

O módulo Parser é responsável pelo parsing e validação de cada linha que lhe é passada, devolvendo uma estrutura Stats contendo a informação relevante da linha (nomes de autores, número de autores e ano da publicação) caso seja válida.

### 6.2 fread

O módulo fread (file read) é responsável pela abertura de ficheiros para leitura. Está preparado para trabalhar com um número de ficheiros máximo a definir na interface. Este módulo está preparado para trabalhar com múltiplos ficheiros simultaneamente devido a duas razões. A primeira é para permitir que este módulo possa trabalhar com o ficheiro lista.txt e com um ficheiro de revista ou conferência em simultâneo. A segunda razão deve-se a inicialmente ter sido pensada a possibilidade de otimizar o programa através da criação de várias threads de forma a que cada ficheiro de revista ou conferência pudesse ser tratado num processo separado, implementando assim paralelismo. Contudo no decorrer do projeto, e após medições de tempos de execução tanto no teste 5 (stress) como posteriormente no TESTE1\_FASE2, foi concluído que esta otimização era desnecessária tendo em conta o tempo que a implementação gastaria, bem como o pouco benefício (em tempo de execução) que traria. Esta conclusão é suportada pelo facto dos testes utilizados terem dimensões razoavelmente grandes e o tempo de execução nestes testes anda na ordem das décimas de segundo tal como nos testes mais simples. Inicialmente foi implementado um buffer de 1KB, tendo sido alterado posteriormente para um buffer dinâmico em que o tamanho inicial deste buffer é ligeiramente superior à dimensão da maioria do tamanho das entradas.



grafo.jpg

### 6.3 fwrite

Este módulo é semelhante ao anterior, difere apenas no propósito de escrever ficheiros.

### 6.4 F

O módulo F consiste na implementação da fase 2A, possui uma interface que é utilizada pelo módulo main de forma a inserir os dados relevantes que o main recebe através do módulo parser. O módulo F é responsável pela criação e gestão da estrutura de dados que permite posteriormente imprimir o ficheiro G.



## 6.5 main

Este módulo controla o fluxo do programa, fazendo a ponte entre os módulos anteriores. Nomeadamente faz um pedido de linha do ficheiro lista.txt ao modulo fread, para cada linha devolvida faz um novo pedido a esse módulo com o nome de cada ficheiro, sendo devolvida cada linha (entrada) deste último ficheiro.

Posteriormente o main passa a linha obtida ao parser, de onde é devolvido um resultado que pode ser NULL sendo incrementado o contador de artigos rejeitados (static no main) ou uma estrutura com informação que é passada ao módulo F para inserção. Finalmente, quando todos os ficheiros contidos no ficheiro "lista.txt" forem processados, o módulo main dá ordem de impressão dos ficheiros D.txt, E.txt e G.csv aos respetivos módulos.

## 7 Conclusão

A principal dificuldade que foi ultrapassada, nomeadamente na 2ª Fase, foi a implementação das estruturas de dados uma vez que a função de inserção não funcionava corretamente tendo sido re implementada numa versão recursiva correta. A dificuldade não ultrapassada devido, ao tempo despendido no processo de refactoring, resume-se á falta de tempo para implementação da estrutura de co-autoria.

## 8 Interface

- Parser

```
/**
 * Parses a Line
 * @param buffer the buffer containing the line to be parsed
 * @param ty+he type of the line conference or journal)
 * @return Struct with all counters.
 */
Stats parseLine(char * buffer, char t);
```

- fread

```
/**
 * Initializes the control for the files
 */
void init_file_control();

/**
 * Opens a file.
 * @return file index in control, or -1 if failed.
 */
int openFile(char * file_name, char * mode);
```





```
/**
 * Reads a line from a file growing the buffer as needed.
 * @param a reference to the buffer reference.
 * @param a reference for the size of the buffer.
 * @param file index in control.
 * @return 0 if success -1 otherwise.
 */
int dynamic_read_line(char** buf, int* size, int file);
```

```
/**
 * Closes a file.
 * @param index of the file to be closed.
 */
void closeFile(int index);
```

- fwrite

```
/**
 * This function outputs the number of rejected entries in each file to the "E
 * @param int counter number of invalid entries in the file.
 * @param char * fileN the name of the courent file.
 */
void imprimeE(int bool, int counter, char * file, char * path);
```

```
/**
 * This function creates the file "D.txt" and it's content. If the file already
 * @param nRej the number of rejected entries
 * @param nJou the number of Journals
 * @param nCou the number of Counferences
 * @param path the path of the file to be written to.
 */
void imprimeD(int nRej, int nJou, int nCou, char * path);
```

- F

```
/**
 * Adds data to the structure.
 * @return returns a pointer to a structure with the linked list
 */
struct sList * addList();
```

```
/**
 * This function creates the file "G.csv" and it's content. If the file already
 * @param bool if it's first time opens in "w" mode else in "a".
 */
int imprimeG();
```