Projeto LI3

Francisco Oliveira Raul Vilas Boas Vitor Peixoto

Grupo 33

1 de Maio de 2017

Laboratórios de Informática III

Mestrado Integrado em Engenharia Informática

Conteúdo

1 Introdução **2**

2 Descrição do Problema **3**

3 Concepção da Solução **4**
3.1 Idealização 4
3.2 Implementação da Estrutura 4
3.3 Operações à Estrutura 7
3.4 Interrogações 7
3.5 Modularidade e Encapsulamento 9

4 Testes **11**

Capítulo 1

Introdução

Este relatório foi realizado no âmbito do projeto proposto na unidade curricular de Laboratórios de Informática III do Mestrado Integrado em Engenharia Informática. Foi-nos então proposto criar um sistema que permitisse analisar diversos artigos presentes nos backups da Wikipedia disponibilizados pela equipa docente. Esse sistema deve também ser capaz de extrair informação útil desses backups, como por exemplo o número de revisões, o número total de artigos, os maiores artigos, etc. Foi nos indicado para realizar este projeto na linguagem *C* e ter como ponto de partida um tutorial em *XML*.

Para chegarmos a resultados concretos adotamos uma postura que se baseia na aprendizagem através do erros. Tudo o que foi desenvolvido foi alcançado através de diversos erros e falhas que fizeram melhorar as nossas capacidades com a linguagem *C*, descobrir outros novos aspetos desta linguagem com os quais não estavamos familiarizados, tais como o encapsulamento de dados, e aprender o funcionamento da estruturação e as capacidades da biblioteca *libxml2*.

Este relatório está orientado por quatro secções, onde serão explorado os problemas a resolver, a nossa abordagem para as suas resoluções e os testes efetuados e respetivos *outputs*.

Capítulo 2

Descrição do Problema

O ponto fulcral deste projeto era desenvolver um sistema que permita analisar backups da Wikipedia de vários meses e conseguir obter informação útil dos mesmos. A informação que deveríamos obter foi-nos dada pela equipa docente sob a forma de interrogações.

O primeiro problema com que nos deparamos foi como organizar a informação de forma a poder ser encontrada de uma maneira rápida, eficaz e fiável. Seria impossível resolver as questões percorrendo diversas vezes os backups porque seria bastante lento (uma vez que o tamanho dos backups somados ronda os 1,7 GB) e complicado (porque iria envolver bastante uso da livraria *XML*).

Ao longo da construção deste projeto fomo-nos deparando diversas vezes com novos problemas, como a necessidade de agrupar os contribuidores num sistema àparte que será falado no próximo capítulo, algumas dificuldades com umas *tags* ocultas do *XML*, entre outros.

Capítulo 3

Concepção da Solução

3.1 Idealização

Para começar achamos necessário aprender mais sobre a livraria *XML* e o seu funcionamento. Após uma larga pesquisa acerca da livraria, do seu funcionamento (tendo como fonte principal o tutorial em *XML*) e de termos criado umas funções de teste para os nossos backups começamos a pensar no próximo obstáculo que seria como organizar a informação de cada backup de modo a ser encontrada rápida e eficazmente.

Após termos conferenciado com colegas de outros grupos e pesquisado na internet, chegamos à conclusão que o método mais eficaz e rápido seria mesmo usar uma tabela de hash (*hashtable*) orientada pelo ID de cada artigo. Esta tabela de hash vai conter diversas informações para cada artigo, especificadas na Figura 3.1.

Esta tabela de hash é *closed addressing* ou seja, em cada posição da hash terá uma lista ligada composta por "caixas" e cada caixa é uma *struct* que vai conter a informação sobre um artigo (ver Figura 3.1).

Este projeto foi iniciado num contexto de *small-scale programming* contudo, conforme o crescimento das funções vimo-nos na necessidade de implementar modularidade, para termos uma melhor organização do código, e também encapsulamento, de modo a garantir um acesso controlado à estrutura.

3.2 Implementação da Estrutura

Após a idealização da estrutura tivemos que implementar a tabela de hash e os backups na mesma. Primeiro tivemos de criar uma simples *struct* demonstrada no Excerto 3.1 para a lista ligada de cada posição da tabela de hash (`DataObject`). Depois foi criado o array de listas ligadas que será a hash (`hashTableID[SIZE]`) dentro da estrutura `TCD_istruct` em que `SIZE=7001`.

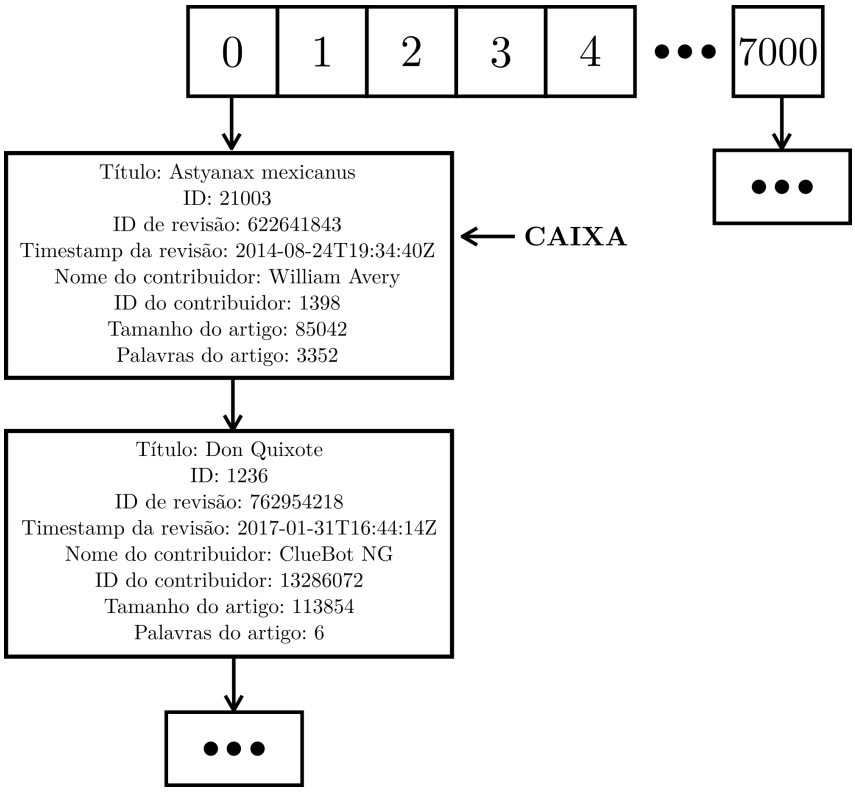

Figura 3.1: *Organização da tabela de hash com closed addressing.*

Excerto 3.1: Implementação da tabela de hash.

```
typedef struct DataObject {
     char * title;
     long id;
     int revid;
     char * revtimestamp;
     char * contributorname;
     int contributorid;
     long tamanho;
     long palavras;
     struct DataObject* next;
} *object;

typedef struct TCD_istruct {
     object hashTableID[SIZE];
     //contributor hashTableRevID[SIZE];
} *TAD_istruct;
```

Após termos a hash criada é necessário adicionar a informação dos artigos à hash. Para o fazer necessitamos dos conhecimentos em *XML* obtidos inicialmente com o tutorial e alguns exemplos que fomos desenvolvendo para termos um melhor entrosamento com a livraria. Para criar uma caixa foi criada a função `getObject` que percorre cada *tag* dentro da *tag* "*page*" e guarda a informação útil nos locais corretos da caixa.

Excerto 3.2: Exemplo da `getObject` para guardar o título do artigo.

```
object getObject (xmlNodePtr node, xmlDocPtr doc) {
     object artigo = (object) malloc (sizeof(struct DataObject));
     node = node->children;

     //Título do artigo
     while(xmlStrcmp(node->name,(const xmlChar *)"title"))
          node = node->next;
     artigo->title = ((char *)xmlNodeListGetString(doc,node->xmlChildren
Node,1));
     ...
}
```

O que a função está a fazer resumidamente (e para este pequeno caso específico - título do artigo) é:

- Enquanto o nome do nó não for "title"avança para o próximo nó;
- Quando encontrar sai do ciclo *while* e guarda o nome do que está dentro desse nó na variável `title` "caixa"da tabela de hash.

Após construída uma caixa esta vai ser inserida na hash. A função responsável é a `insertHashID` (Excerto 3.3) que calcula o resto da divisão por `SIZE=7001` (usando uma função auxiliar) que dará o índice do array da hash onde colocar a caixa e a insere na cabeça da lista ligada, como se pode observar na Figura 3.2.

Excerto 3.3: `insertHashID`.

```
void insertHashID (TAD_istruct qs, object artigo) {
     int hashcode = hashCode(artigo->id);
     artigo->next = qs->hashTableID[hashcode];
     qs->hashTableID[hashcode] = artigo;
}
```

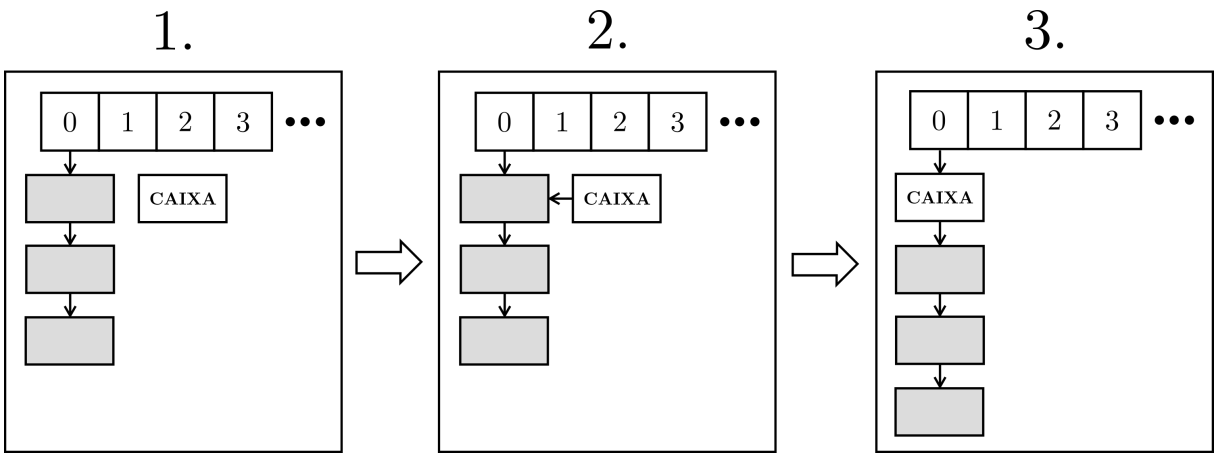

Figura 3.2: *Funcionamento da função* `insertHashID`.

3.3 Operações à Estrutura

Após termos a tabela de hash funcional começamos a escrever as funções definidas no `interface.h`, começando pela `init`, `load` e `clean`.

A `init` tem a função de alocar o espaço necessário para a estrutura e inicializar todas as posições da hash a `NULL`.

A `load` recebe os backups e carrega-os na estrutura. Faz o *parse* do documento e verifica se é válido com a função `parseDoc`. Caso seja avança até à *tag* "*page*", cria uma caixa para aquele artigo invocando a função `getObject` e insere-o na hash pela função `insertHashID`. No fim liberta as variáveis alocadas na memória e retorna a estrutura.

A `clean` liberta todo o espaço alocado pela estrutura (começando por libertar primeiramente as caixas das listas ligadas da hash).

3.4 Interrogações

Por fim chegamos à parte em que podemos começar a desenvolver as interrogações, tendo em conta o funcionamento da tabela de hash implantada.

Funções como a `all_articles`, `article_title`, `contributor_name` e `article_timestamp` foram bastante simples de responder visto que envolve unicamente uma pesquisa pela hash à procura da informação que necessitamos. Deixamos no Excerto 3.4 o exemplo de como foi implantada a função `article_title`.

Excerto 3.4: `article_title`.

```
char* article_title(long article_id, TAD_istruct qs){
     object caixa = qs->hashTableID[hashCode(article_id)];
     while(caixa!=NULL){
          if(caixa->id == article_id){
               char* title = mystrdup(caixa->title);
               return title;
          }
          caixa = caixa->next;
     }
     return NULL;
}
```

A função `article_title` faz uso do argumento `article_id` para descobrir em que posição da hash procurar. Dentro dessa posição vamos andar pela lista ligada procurar quando o ID guardado nessa caixa é igual ao ID fornecido como argumento. Quando encontrar, aloca uma variável `title` para retornar o que está guardado na posição do título dessa caixa. O facto de alocarmos uma variável para ser retornada e não retornar imediatamente o elemento da hash está ligado ao encapsulamento que falaremos mais à frente (o alocamento é feito na função `mystrdup`). Caso chegue ao fim da lista ligada e não encontre o ID que procura retorna `NULL`.

Depois tivemos funções com um grau de dificuldade um bocado maior, mas ainda bastante acessíveis como a `unique_articles` que vai percorrer todas as posições da tabela de hash e para cada posição verifica se o ID do artigo existe num *array* auxiliar criado que guarda os ID's não repetidos. Se existir avança para a próxima "caixa", se não existir adiciona esse ID ao *array* auxiliar e incrementa 1 num contador. No fim retorna esse contador que vai conter o número de artigos sem repetições

A `all_revisions` também foi uma função que mostrou alguma resistência mas foi também resolvida. Para tal recorremos a uma matriz para cada posição da tabela de hash e a uma nova tabela de hash que achamos ser vantajosa para a resolução desta questão e da `top_10_contributors` (e que foi aproveitada também para melhorar o desempenho da `contributor_name`). A nova tabela de hash vai conter o ID do artigo, o ID da revisão, o nome do contribuidor e o seu ID também. A matriz vai conter diversas linhas, uma para cada ID de artigo, cada linha vai ter na 1ª posição o número de ID's de revisão para esse artigo, na 2ª posição o ID do artigo e o resto da linha é ocupada com os ID's de revisão desse artigo. Essa matriz vai ser útil na medida em que para cada revisão vamos procurar o seu ID. Se já estiver registado o ID e a revisão, não fazemos nada. Se o ID já estiver registado mas não o ID de revisão, registamos a revisão nessa linha (linha desse ID de artigo) e incrementamos 1 ao número de revisões desse artigo. Se esse artigo ainda nem sequer estiver na matriz adicionamos esse ID de artigo e o de revisão e iniciamos com 1 na posição 0 da linha (nº de revisões existentes). Quando chegarmos ao fim dessa posição da tabela de hash adicionamos todos os valores das primeiras posições de todas as linhas à variável `count` e iniciamos uma nova matriz para a nova posição. No fim a soma dos `count` dá o número de revisões.

Excerto 3.5: Implementação da nova tabela de hash para revisões.

```
typedef struct DataContributor {
     long id;
     int revid;
     char * contributorname;
     int contributorid;
     struct DataContributor* next;
} *contributor;

typedef struct TCD_istruct {
     object hashTableID[SIZE];
     contributor hashTableRevID[SIZE];
} *TAD_istruct;
```

Outra das funções que nos mostrou dificuldades foi a `top_20_largest_articles` e a `top_N_articles_with_more_words`, mas só iremos abordar a `top_20_largest_articles` devido ao facto de serem muito semelhantes. Para estas funções foram criados mais dois elementos na tabela de hash de artigos, o tamanho e as palavras do texto do artigo (usando para isso a função `strlen` e `contaPalavras` dentro da `getObject`, sendo que a `contaPalavras` foi uma função criada por nós). Para responder a esta questão decidimos criar dois *arrays* que se acompanham (algo como uma matriz), sendo que um *array* contém os ID's dos artigos e o outro contém o tamanho desse ID. Inicialmente é verificado se esse ID já é existente no *array* dos ID's. Se não existir e caso seja maior que o ID de menor tamanho elimina esse ID e assume a sua posição e entra num ciclo verificando sempre se o seguinte é menor (ou se é igual, mas o ID é maior), se for vai avançando (tanto o ID como o tamanho vão avançando simultaneamente para que o índice de cada *array* seja o correspondente ao mesmo conjunto ID - tamanho) até quebrar a condição. Caso o ID já exista no array temos de verificar se o tamanho que queremos por é maior. Caso seja, trocamos o tamanho que tinha no *array* pelo que queremos por e voltamos a iniciar o ciclo para avançar no *array* dependendo do tamanho. No fim iremos obter o *array* dos ID's ordenado pelos de maior tamanho.

A função `top_N_articles_with_more_words` é exatamente igual, mas iremos usar

o elemento palavras da tabela de hash e o tamanho do *array* não é fixo em 20, mas definido pelo argumento "N"dado à função.

A função `titles_with_prefix` inicialmente vai criar um *array* com todos os ID's dos artigos sem repetições. Percorre a lista de ID's e para cada ID obtemos o título correspondente (sendo que esse título é sempre o da revisão mais recente do artigo, devido à tabela de hash ser construída com inserção à cabeça) e verifica se o argumento dado à função é prefixo do título através da função `myStrcmp` e caso seja esse título é guardado num *array* que após percorrer todos os ID's será devolvido como *output* após ser ordenado alfabeticamente.

Por último, para a interrogação `top_10_contributors` foi criada uma lista ligada de *structs*. Cada "caixa"guarda informação acerca do contribuidor (ID, contador de revisões únicas, ID de todas as revisões únicas e uma variável que diz o tamanho máximo alocado para sabermos quando é necessário realocar memória). Verificamos se o contribuidor já existe na lista ligada com uma função auxiliar, caso não exista é criada uma "caixa"com os dados desse contribuidor (o seu ID e a revisão inicial) e insere-se na lista. Caso esse contribuidor já esteja presente na lista é verificada se a revisão que queremos inserir já existe na lista, se já existir ignoramos, se não existir acrescentamos à lista e incrementamos 1 no número de revisões desse contribuidor. Antes de se introduzir uma nova revisão verificamos se a estrutura está "lotada"e caso esteja realocamos mais espaço para esta e para futuras revisões a ser introduzidas. No fim vamos adicionar os contadores de revisões e ID do contribuidor responsável num duplo *array* ordenado pelo número de revisões e de tamanho 10. Após a inserção ordenada é devolvido o *array* que irá conter os 10 maiores contribuidores.

3.5 Modularidade e Encapsulamento

Como tema abordado nesta unidade curricular nas primeiras semanas e indo de encontro ao material disponível na *Blackboard* tentamos ao máximo implementar modularidade e encapsulamento na nossa solução.

Como tal, tivemos preocupação em manter a estrutura num ficheiro ".*c* "ao invés do *header* para obtermos uma estrutura privada, opaca. Por isso criamos um ficheiro *hash.h* com a declaração das estruturas e um *hash.c* onde estas estão definidas.

Excerto 3.6: hash.h

```
typedef struct DataObject *object;
typedef struct DataContributor *contributor;
typedef struct titlesWprefix *TWD;
typedef struct top10contributor *T10C;
```

Excerto 3.7: hash.c

```
typedef struct DataContributor {
     long id;
     int revid;
     char * contributorname;
     int contributorid;
     struct DataContributor* next;
} *contributor;
...
```

De modo a facilitar a correção de erros, a compreensão do código e a reutilização de pedaços de código para outros problemas optamos por implementar o nosso código em vários módulos, visto que o programa acabou por demonstrar uma grande dimensão. Por isso dividimos o código nos seguintes módulos demonstrados na Figura 3.3.

Figura 3.3: *Módulos do sistema.*

O módulo *hash.c* guarda a estrutura geral, as tabelas de hash e outras estruturas de apoio a questões específicas, que estão declaradas no módulo *hash.h* em conjunto com outras funções que precisam de ser globais a vários módulos. O ficheiro *load.c* inclui todas as funções responsáveis por fazer o *parse* dos backups, obter a informação que necessitamos e guardar na estrutura (`load`), bem como iniciá-la (`init`) e limpá-la no fim (`clean`). O módulo *program.c* inclui a `main` e serve como ferramenta de teste às funções, visto que a equipa docente utiliza a sua própria `main`. O *questoes.c* inclui todas as funções necessárias para responder às interrogações, bem como as suas auxiliares, tendo a estrutura já com a informação necessária. Por último temos um *Makefile* que usamos para compilar o programa, não fazendo no entanto parte do programa em si e consequentemente não faz parte da estrutura modular.

Para não permitir o acesso à estrutura tivemos de ter cuidado em como retornar os valores da estrutura para a consola. Para isso não podemos devolver diretamente valores da estrutura, mas sim alocar espaço na memória para o que queremos retornar e no fim retornar esse espaço criado, assim a estrutura está oculta.

Endereço da estrutura transparente.

```
//retorna o título do artigo.
return caixa->title;
```

Endereço da estrutura oculto.

```
//retorna o título do artigo.
char* title;
title = strdup(caixa->title);
return title;
```

Tentamos cobrir qualquer ponto de acesso à estrutura que encontramos, seja através dos *header files* ou através dos *returns* das funções, em conjunto com um programa modularizado de uma maneira, que a nosso ver, é a que mais sentido faz.

NOTA: Apesar de não fazer parte deste capítulo, achamos importante referir que o alocamento de memória foi um dos pontos importantes neste projeto visto que a sua libertação é crucial para obter um programa que funcione numa velocidade aceitável, sem bloquear os nossos computadores (que infelizmente ocorreu bastantes vezes). Evitamos ao máximo atitudes desde recorrer a funções da *libxml2* que alocavam memória até à preocupação em usar a ferramenta *Valgrind* para verificar quantos alocamentos não eram libertados, tendo sempre como objetivo diminuir a diferença entre os alocamentos e libertações. Talvez por isso tenhamos obtido um trabalho com uma velocidade bastante boa sem ter havido necessidade de recorrer a outras livrarias ou a paralelismo do processador (embora fosse uma opção válida).

Capítulo 4

Testes

Nesta secção são executados diversos testes feitos na nossa consola às interrogações, semelhantes aos testes feitos no portal dos resultados da unidade curricular.

Felizmente conseguimos responder a todas as questões e por isso os resultados dos nossos testes são iguais aos esperados.

Excerto 4.1: All Articles.

```
printf("1.  All articles:  %ld\n", all_articles(qs));
1.  All articles:  59593
```

Excerto 4.2: Unique Articles.

```
printf("2.  Unique articles:  %ld\n", unique_articles(qs));
2.  Unique articles:  19867
```

Excerto 4.3: All Revisions.

```
printf("3.  All revisions:  %ld\n", all_revisions(qs));
3.  All revisions:  40131
```

Excerto 4.4: Top 10 Contributors.

```
long* auxTop10 = top_10_contributors(qs);
printf("4.  Top 10 contributors:  \n");
for(i=0; i<10; i++)
     printf("%ld\n", auxTop10[i]);
4.  Top 10 contributors:  (28903366, 13286072, 27823944, 27015025,
194203, 212624, 7852030, 7328338, 7611264, 14508071)
```

Excerto 4.5: Contributor Name.

```
printf("5.  Contributor name:  %s\n", contributor_name(28903366,qs));
5.  Contributor name:  Bender the Bot

printf("5.  Contributor name:  %s\n", contributor_name(194203,qs));
5.  Contributor name:  Graham87

printf("5.  Contributor name:  %s\n", contributor_name(1000,qs));
5.  Contributor name:  (null)
```

Excerto 4.6: Top 20 Largest Articles.

```
long* auxTop20 = top_20_largest_articles(qs);
printf("6.  Top 20 largest articles:  \n");
for(i=0; i<20; i++)
     printf("%ld\n", auxTop20[i]);
6.  Top 20 largest articles:  (15910, 23235, 11812, 28678, 14604,
23440, 26847, 25507, 26909, 18166, 4402, 14889, 23805, 25391, 7023,
13224, 12108, 13913, 23041, 18048)
```

Excerto 4.7: Article Title.

```
printf("7.  Article title:  %s\n", article_title(15910,qs));
7.  Article title:  List of compositions by Johann Sebastian Bach

printf("7.  Article title:  %s\n", article_title(25507,qs));
7.  Article title:  Roman Empire

printf("7.  Article title:  %s\n", article_title(1111,qs));
7.  Article title:  Politics of American Samoa
```

Excerto 4.8: Top N Articles With More Words.

```
int n = 30;
long* auxTopN = top_N_articles_with_more_words(30,qs);
printf("8.  Top %d articles with more words:  \n", n);
for(i=0; i<n; i++)
     printf("%ld\n", auxTopN[i]);
8.  Top 30 articles with more words:  (15910, 25507, 23235, 11812,
13224, 26847, 14889, 7023, 14604, 13289, 18166, 4402, 12157, 13854,
23805, 25401, 10186, 23041, 18048, 16772, 22936, 28678, 27069, 9516,
12108, 13913, 13890, 21217, 23440, 25391)
```

Excerto 4.9: Titles With Prefix.

```
char** auxPrefix = titles_with_prefix("Anax",qs);
printf("9.  Titles with prefix "Anax":\n");
for(i=0; auxPrefix[i]; i++)
     printf("%s\n", auxPrefix[i]);
9.  Titles with prefix "Anax":  (Anaxagoras, Anaxarchus, Anaximander,
Anaximenes of Lampsacus, Anaximenes of Miletus)
```

Excerto 4.10: Article Timestamp.

```
printf("10. Article timestamp: %s\n", article_timestamp(12,763082287,qs));
10.  Article timestamp:  2017-02-01T06:11:56Z

printf("10. Article timestamp: %s\n", article_timestamp(12,755779730,qs));
10.  Article timestamp:  2016-12-20T04:02:33Z

printf("10. Article timestamp: %s\n", article_timestamp(12,4479730,qs));
10.  Article timestamp:  (null)
```