Documentação do Trabalho Prático 1

03/09/2017

Alunos

- **Ruan Gabriel Gato Barros** - 21553690
- **Rúben Jozafá Silva Belém** - 21551560

Esse trabalho teve como objetivo a implementação de programas para armazenamento e pesquisa de dados indexados partindo de uma massa de dados.

---

1. Estrutura do Projeto

A leitura do arquivo artigo.csv *(parsing)*

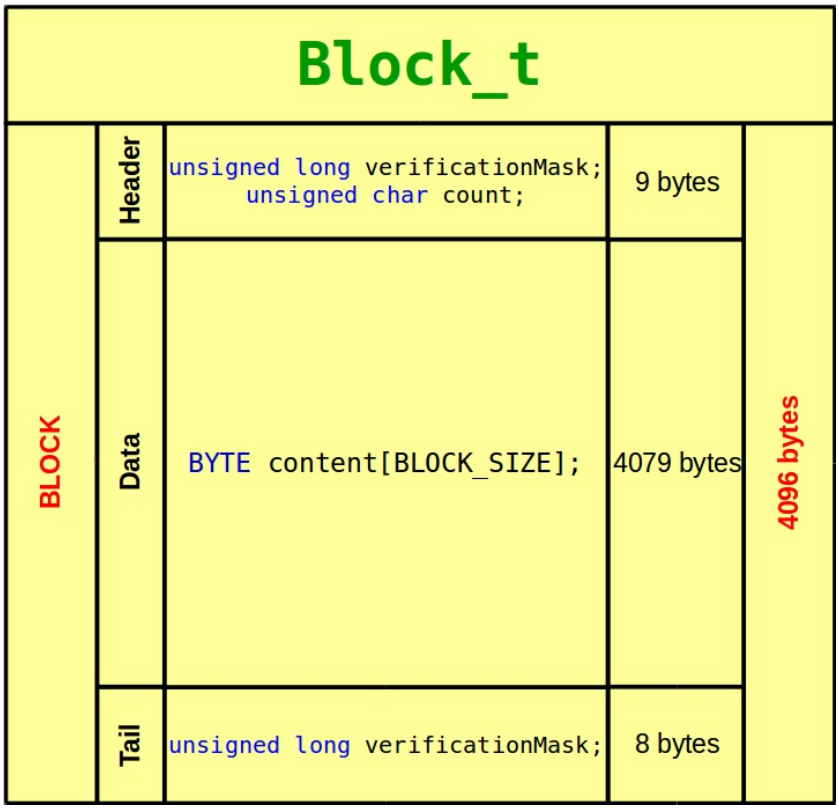
O arquivo artigo.csv foi lido de forma que fosse possível classificar cada par de caracteres, de forma que seja possível ler o arquivo de forma muito mais rápida que um regex, por exemplo; então,decidimos classificar cada par de caracteres na leitura para fazer o parsing dos registros, salvando-os num buffer para tratar os caracteres excedentes e especiais.

O arquivo de dados organizado por hashing

Optamos por implementar o hash perfeito, tendo em mente várias simplicidades que tal implementação traria. Embora tal implementação fosse bastante custosa no que diz respeito à memória secundária, não teve um impacto negativo suficientemente grande para superar os benefícios de tal organização.

A implementação do bloco levou em consideração que os artigos possuem um tamanho suficientemente grande para não ser possível o armazenamento de mais de um artigo por bloco, levando em consideração que a implementação escolhida foi de dados não espalhados.

No que diz respeito à validade do bloco - já que foi utilizado um hash perfeito, então muitos blocos inválidos naturalmente se encontrarão no arquivo - utilizamos uma técnica muito utilizada em sistemas operacionais para indicar que aquela região de memória é o início de um campo válido, forçando a verificação do bloco com uma máscara grande o suficiente para ser praticamente impossível se igualar com lixo de memória.



A estrutura do bloco, como vista acima, conta com 3 subdivisões :

1 . **Header** - responsável por armazenar a máscara de verificação e a contagem de artigos, tendo essa divisão um tamanho total de 9 bytes em uma arquitetura x64.

2 . **Data** - responsável por armazenar os artigos (no caso, artigo; mas foi implementada de forma que seja facilmente adaptado para mais artigos).

3 . **Tail** - responsável por armazenar a segunda parte da máscara de verificação e contagem de artigos, sendo tal divisão sendo um nível a mais de segurança na integridade dos arquivos.

O arquivo de índice primário

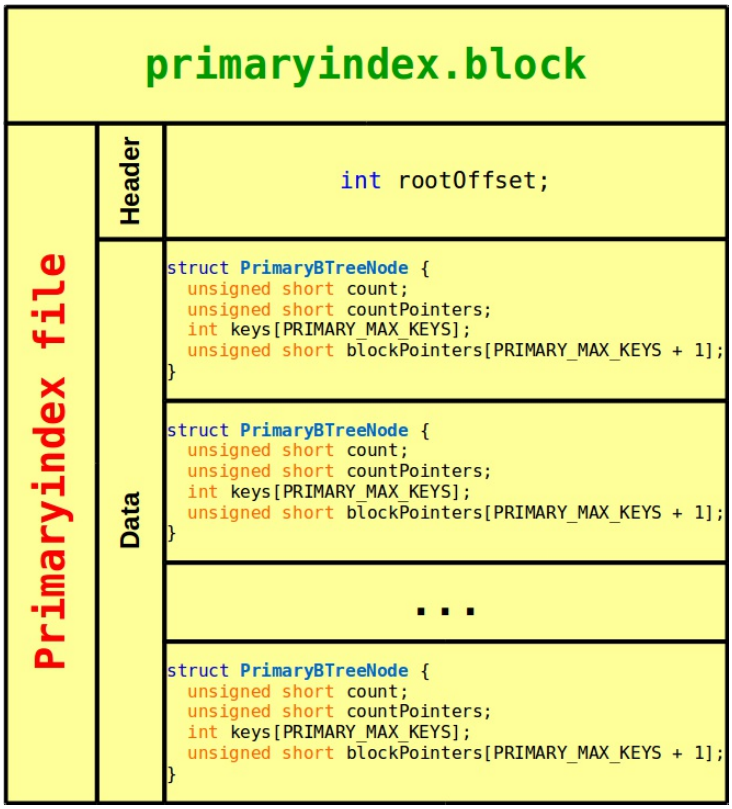
Optamos por indexar o id utilizando o mínimo de espaço possível. A organização por **hash perfeito** no arquivo nos permitiu economizar o espaço de ponteiro para dados, já que **a chave de busca é o próprio ponteiro para dados**. Verificamos que, para cada nó da árvore que comporta 680 elementos, seria possível representar os ponteiros para blocos com uma variável do tipo unsigned short.

A implementação da indexação primária levou em consideração a utilização de um header para indicar o nó raíz, além de indexar os blocos de índice propriamente ditos por meio da struct abaixo :

```
struct PrimaryBTreeNode {
  unsigned short count;
  unsigned short countPointers;
  int keys[PRIMARY_MAX_KEYS];Título [TROCAR]
  unsigned short blockPointers[PRIMARY_MAX_KEYS + 1];

  PrimaryBTreeNode(int order);
  bool isLeaf();
  bool hasRoom();
  unsigned short insert(int key);
};
```

A estrutura do arquivo pode ser averiguada abaixo :



O arquivo de índice secundário

O arquivo de índice secundário se assemelha bastante com o arquivo de índice primário, possuindo este um header para indicar o offset do nó raíz. A principal diferença está no fato de que a chave de busca não é mais o ponteiro para dados, já que a chave de busca se trata de um char[300], diferente do ponteiro para dados.

Um detalhe importante dessa implementação é que, devido ao enorme tamanho da chave de busca, só foi possível armazenar poucos elementos por nó, consequentemente existirão mais nós para serem representados, sugerindo uma mudança no tamanho da variável que armazena o ponteiro para blocos, sendo essa no índice primário um unsigned short e no índice secundário um int, justamente para comportar o número gigantesco de blocos.

A indexação conta com duas estruturas principais utilizadas para representar um nó :

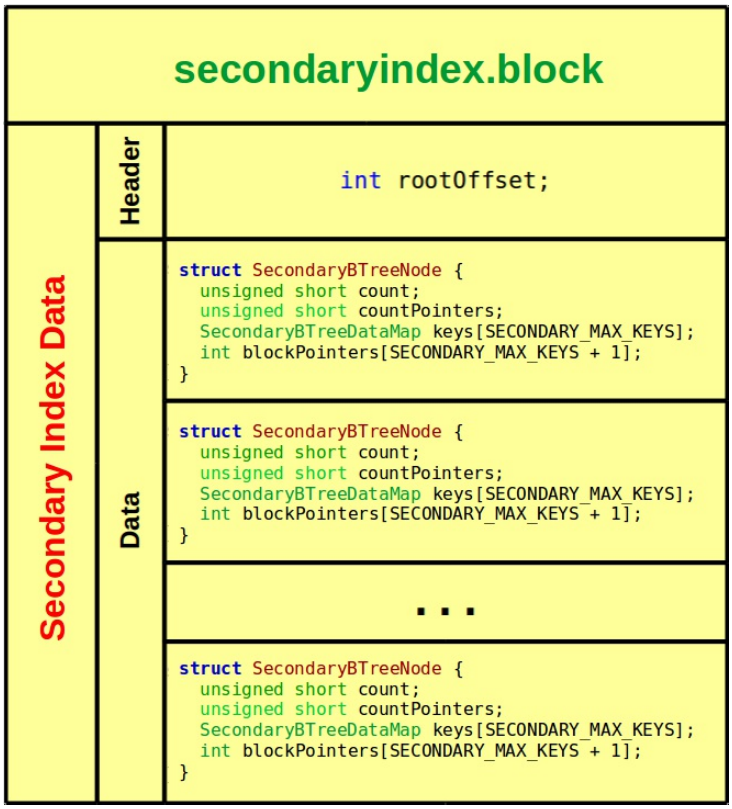
```
struct SecondaryBTreeNode {
  unsigned short count;
  unsigned short countPointers;
  SecondaryBTreeDataMap keys[SECONDARY_MAX_KEYS];por
char[SECONDARY_KEY_LENGTH] para o Título [TROCAR]
  int blockPointers[SECONDARY_MAX_KEYS + 1];

  SecondaryBTreeNode(int order);
  bool isLeaf();
  bool hasRoom();
  int insert(SecondaryBTreeDataMap&);
};
```

```
struct SecondaryBTreeDataMap {
  char key[SECONDARY_KEY_LENGTH];
  int dataPointer;

  bool operator< (const SecondaryBTreeDataMap& other)
const;
  bool operator> (const SecondaryBTreeDataMap& other)
const;
  bool operator==(const SecondaryBTreeDataMap& other)
const;
  void operator=(const SecondaryBTreeDataMap& other);
};
```

Tais structs representam um bloco e seus devidos campos, abaixo é possível averiguar a estrutura do nó em um bloco abaixo :



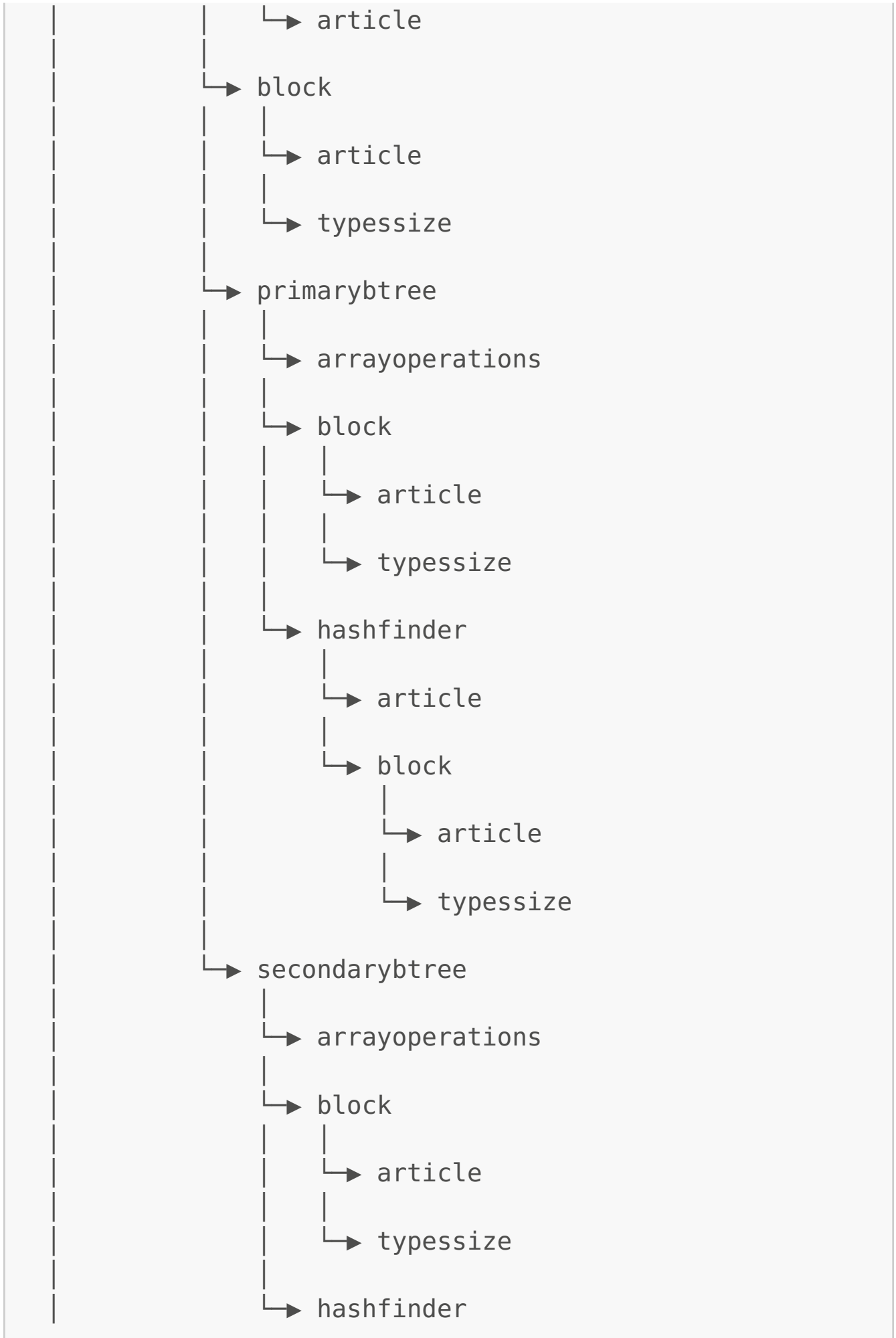
---

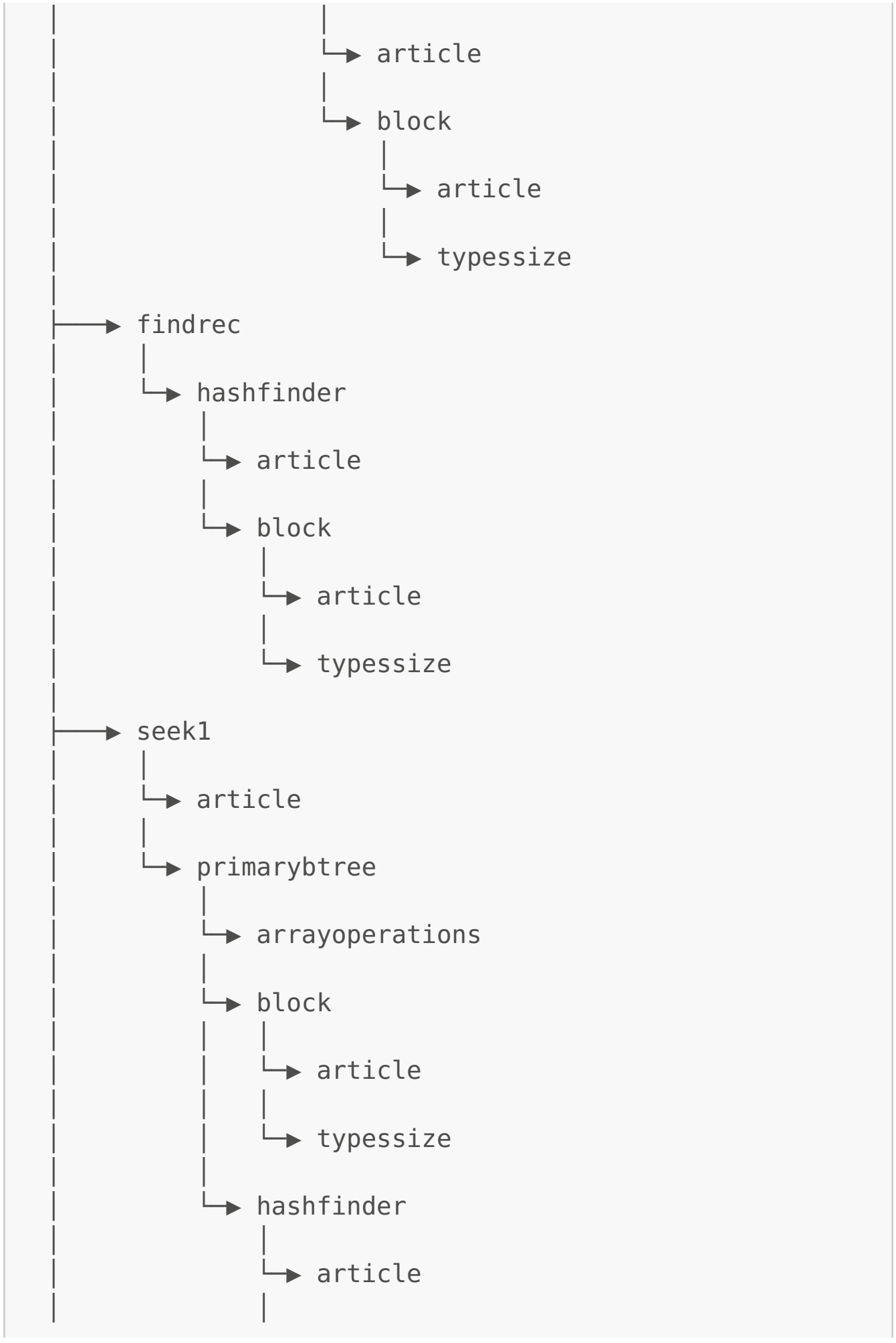
2. Dependência de Fontes

Para uma visualização realmente completa de todas as dependências de cada um dos 4 arquivos-fonte dos programas principais, nós decidimos representá-las no formato de árvore de dependências presente abaixo. Há algumas redundâncias, mas porque procuramos representar da forma mais fiel possível ao que está presente nos arquivos *header*.

- Árvore de Dependências

```
Projeto
|
├──► upload
|      |
|      └──► hashfilefactory
|             |
|             ├──► iohandler
|             |    |
```


article
block
article
typessize
primarybtree
arrayoperations
block
article
typessize
hashfinder
article
block
article
typessize
secondarybtree
arrayoperations
block
article
typessize
hashfinder


article
block
article
typessize
findrec
hashfinder
article
block
article
typessize
seek1
article
primarybtree
arrayoperations
block
article
typessize
hashfinder
article

```
│               └─▶ block
│                   │
│                   └─▶ article
│                   │
│                   └─▶ typessize
│
│
│
│
│
│
└─▶ seek2
      │
      └─▶ article
      │
      └─▶ secondarybtree
          │
          └─▶ arrayoperations
          │
          └─▶ block
          │   │
          │   └─▶ article
          │   │
          │   └─▶ typessize
          │
          └─▶ hashfinder
              │
              └─▶ article
              │
              └─▶ block
                  │
                  └─▶ article
                  │
                  └─▶ typessize
```

---

3. Execução

Documentação

Foi utilizada a ferramenta doxygen para a documentação do código fonte, sendo tal software necessário se desejar gerar a documentação. Além disso, foi utilizada a ferramenta auxiliar moxygen para converter os formatos gerados do doxygen para markdown.

tl;dr : foram utilizadas as ferramentas doxygen e moxygen.

Compilação

Para a compilação, basta executar o comando make na pasta raiz do projeto, serão gerados quatro executáveis : upload,findrec,seek1 e seek1.

Execução

Os arquivos possuem duas maneiras de entrada (exceto o arquivo upload), recebem parâmetros ou da entrada char *argv[] ou da própria stream de entrada cin, eis alguns exemplos :

```
[Via ARGV]

./seek1 123
```

```
[Via CIN]
./seek1

> 123
```

```
[Via ARGV]
(pelo argv, deve ser executado passando a string
desejada entre aspas)

./seek2 "ICAN : efficiently calculate active node set
from searches"
```

```
[Via CIN]
./seek2

> ICAN : efficiently calculate active node set from
searches
```

Documentação do Código

class HashFileFactory

A class to recover raw information in the hashed file

Summary

Members	Descriptions	Author
	Create the hashed file using the file on the first paramether to read the CSV	

public voidcreateBinaryFilePerfectHash(FILE * toRead,FILE * toWrite)	format file and the file on the second parament to write the binary file as a bonus, create the primary index as well xD	Rúben Belém

Members

public voidcreateBinaryFilePerfectHash(FILE * toRead,FILE * toWrite)

Create the hashed file using the file on the first parament to read the CSV format file and the file on the second parament to write the binary file as a bonus, create the primary index as well xD

class IOHandler

A class to read and handle the CSV file, buffering and handleing the fields

Summary

Members	Descriptions	Author
publicIOHandler(FILE *)	Default IOHandler constructor, receiving a file to read	
public boolhasNext()	Verify if there is next record in the buffer	Ruan Gabriel

public voidparseNext()	Prepare the next parsing element	Ruan Gabriel
public voidoperator>> (Article_t&)	Copy the content of the buffer into an article	Ruan Gabriel
public intgetBiggestId()	Parse the next record contained in the buffer	Ruan Gabriel

Members

publicIOHandler(FILE *)

Default IOHandler constructor, receiving a file to read

public boolhasNext()

Verify if there is next record in the buffer

public voidparseNext()

Prepare the next parsing element

public voidoperator>>(Article_t&)

Copy the content of the buffer into an article

public intgetBiggestId()

Parse the next record contained in the buffer

class PrimaryBTree

A class abstracting the btree

Summary

Members	Descriptions	Author
public unsigned shortrootOffset		
public voidinsert(int key,FILE * indexFile)	Insert a key in the tree	Rúben Belém/Ruan Gabriel
public std::pair< bool, int >getArticle(int key,Article_t*,FILE *)	Get an article from the tree	Rúben Belém/Ruan Gabriel
public voidbuildIndex(FILE *)	Build the PrimaryBTree index, writing a new root and its offset	Rúben Belém
public voidreadRoot(FILE * indexFile)	Read the root whence the offset is set	Rúben Belém
publicPrimaryBTree()	PrimaryBTree constructor	

Members

public unsigned shortrootOffset

public voidinsert(int key,FILE * indexFile)

Insert a key in the tree

public std::pair< bool, int >getArticle(int key,Article_t*,FILE *)

Get an article from the tree

public voidbuildIndex(FILE *)

Build the PrimaryBTree index, writing a new root and its offset

public voidreadRoot(FILE * indexFile)

Read the root whence the offset is set

publicPrimaryBTree()

PrimaryBTree constructor

class SecondaryBTree

A class abstracting the btree

Summary

Members	Descriptions	Author
public introotOffset		
public voidinsert(SecondaryBTreeDataMap&,FILE * indexFile)	Insert a key in the tree	Rúben Belém/Rua Gabriel
public std::pair< bool, int >getArticle(SecondaryBTreeDataMap& key,Article_t*,FILE *)	Get an article from the tree	Ruan Gabriel
public voidbuildIndex(FILE *)	Build the PrimaryBTree index, writing a new root and its offset	Rúben Belém

public voidreadRoot(FILE * indexFile)	Read the root whence the offset is set	Rúben Belém
publicSecondaryBTree()	PrimaryBTree constructor	

Members

public introotOffset

public voidinsert(SecondaryBTreeDataMap&,FILE * indexFile)

Insert a key in the tree

public std::pair< bool, int >getArticle(SecondaryBTreeDataMap& key,Article_t*,FILE *)

Get an article from the tree

public voidbuildIndex(FILE *)

Build the PrimaryBTree index, writing a new root and its offset

public voidreadRoot(FILE * indexFile)

Read the root whence the offset is set

publicSecondaryBTree()

PrimaryBTree constructor

struct AbstractBlock_t

Summary

Members	Descriptions
public chardata	

Members

public chardata

struct Article_t

A struct to embbed and abstract an article and its fields

Summary

Members	Descriptions	Author
public intid		
public chartitle		
public intyear		
public charauthors		
public intcitations		
public chardate		
public charsnippet		
public std::stringtoString()	Transform the content of this block into a string	Rúben Belém
publicArticle_t(int,char,int,char,int,char,char)	Constructor including the fields	

publicArticle_t()	Default constructor of an Article

Members

public intid

public chartitle

public intyear

public charauthors

public intcitations

public chardate

public charsnippet

public std::stringtoString()

Transform the content of this block into a string

publicArticle_t(int,char,int,char,int,char,char)

Constructor including the fields

publicArticle_t()

Default constructor of an Article

struct Block_t

A struct to embbed and abstract an block, its head, data and tail

Summary

Members	Descriptions	Author
public BYTEcontent	Ruan Gabriel	
public booltryPutArticle(Article_t&)	Try to put the article into the block, return true if it has been successfull	Ruan Gabriel
public boolhasSpace()	Verify if there is space in the block	Ruan Gabriel
public boolisValid()	Verify if the block is valid	Ruan Gabriel
public voidvalidate()	Validate the block before the insertion so the block can be identifyed	Ruan Gabriel
publicArticle_t*getArticle(unsigned int)	Get an article in the relative position in the block	Ruan Gabriel
publicBlock_t()	Default block constructor	

Members

public BYTEcontent

public booltryPutArticle(Article_t&)

Try to put the article into the block, return true if it has been

successfull

public boolhasSpace()

Verify if there is space in the block

public boolisValid()

Verify if the block is valid

public voidvalidate()

Validate the block before the insertion so the block can be identifyed

publicArticle_t*getArticle(unsigned int)

Get an article in the relative position in the block

publicBlock_t()

Default block constructor

struct Header_Interpretation_t::Header

Abstract header representation

Summary

Members	Descriptions	Author
public unsigned longverificationMask		Ruan Gabriel
		Ruan

public unsigned charcount	Gabriel

Members

public unsigned longverificationMask

public unsigned charcount

struct PrimaryBTreeNode

A struct used for abstract the concept of node

Summary

Members	Descriptions	Author
public unsigned shortcount		
public unsigned shortcountPointers		
public intkeys		
public unsigned shortblockPointers		
publicPrimaryBTreeNode(int order)	PrimaryBTreeNode constructor	
public boolisLeaf()	Verify if a node is a leaf	Rúben Belém
public boolhasRoom()	Verify if a node has room to insert new nodes	Rúben Belém
	Insert a key in a node	

public unsigned shortinsert(int key)	and returns the index where the insertion was made.	Rúben Belém/Ruan Gabriel

Members

public unsigned shortcount

public unsigned shortcountPointers

public intkeys

public unsigned shortblockPointers

publicPrimaryBTreeNode(int order)

PrimaryBTreeNode constructor

public boolisLeaf()

Verify if a node is a leaf

public boolhasRoom()

Verify if a node has room to insert new nodes

public unsigned shortinsert(int key)

Insert a key in a node and returns the index where the insertion was made.

struct PrimaryBTreeRecursionResponse

A struct used for save the response of the recursive insertion method

Summary

Members	Description
public boolhasBeenSplit	
public intpromotedKey	
public unsigned shortnewBlockOffset	
publicPrimaryBTreeRecursionResponse(bool,int,unsigned short)	Build a recursion response from the core

Members

public boolhasBeenSplit

public intpromotedKey

public unsigned shortnewBlockOffset

publicPrimaryBTreeRecursionResponse(bool,int,unsigned short)

Build a recursion response from the core

struct SecondaryBTreeDataMap

A struct used for abstract the keymap and the data block

Summary

Members	Descriptions
public charkey	
public intdataPointer	
public booloperator<(constSecondaryBTreeDataMap& other) const	
public booloperator>(constSecondaryBTreeDataMap& other) const	
public booloperator==(constSecondaryBTreeDataMap& other) const	
public voidoperator=(constSecondaryBTreeDataMap& other)	

Members

public charkey

public intdataPointer

public booloperator<(constSecondaryBTreeDataMap& other) const

public booloperator>(constSecondaryBTreeDataMap& other) const

public booloperator==(constSecondaryBTreeDataMap& other) const

public voidoperator=(constSecondaryBTreeDataMap& other)

struct SecondaryBTreeNode

A struct used for abstract the concept of node

Summary

Members	Descriptions	Author
public unsigned shortcount		
public unsigned shortcountPointers		
publicSecondaryBTreeDataMapkeys		
public intblockPointers		
publicSecondaryBTreeNode(int order)	PrimaryBTreeNode constructor	
public boolisLeaf()	Verify if a node is a leaf	Rúben Belém
public boolhasRoom()	Verify if a node has room to insert new nodes	Rúben Belém
public intinsert(SecondaryBTreeDataMap&)	Insert a key in a node and returns the index where the insertion was made.	Rúben Belém/Ruan Gabriel

Members

public unsigned shortcount

public unsigned shortcountPointers

publicSecondaryBTreeDataMapkeys

public intblockPointers

publicSecondaryBTreeNode(int order)

PrimaryBTreeNode constructor

public boolisLeaf()

Verify if a node is a leaf

public boolhasRoom()

Verify if a node has room to insert new nodes

public intinsert(SecondaryBTreeDataMap&)

Insert a key in a node and returns the index where the insertion was made.

struct SecondaryBTreeRecursionResponse

A struct used for save the response of the recursive insertion method

Summary

Members
public boolhasBeenSplit
publicSecondaryBTreeDataMappromotedKey
public intnewBlockOffset

publicSecondaryBTreeRecursionResponse(bool)

---

publicSecondaryBTreeRecursionResponse(bool,SecondaryBTreeDataMap&

Members

public boolhasBeenSplit

publicSecondaryBTreeDataMappromotedKey

public intnewBlockOffset

publicSecondaryBTreeRecursionResponse(bool)

Build a recursion response from the core

publicSecondaryBTreeRecursionResponse(bool,SecondaryBTreeDataMap&,int)

Build a recursion response from the core

struct Tail_Interpretation_t::Tail

Abstract tail representation

Summary

Members	Descriptions

public unsigned longverificationMask

Members

public unsigned longverificationMask

Generated by Moxygen