Projeto de Estrutura de Dados I e Linguagem de Programação I

Francisco de Assis Barbosa[1], **Gelly Viana Mota** [1], **Manoel Dinab da C. dos S. Junior**[1]

[1]Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN) – Natal – RN – Brasil

assis_25@yahoo.com.br, gellyviana@outlook.com, dinabjunior@hotmail.com

Abstract. *The Data Structuring I and Programming Language I project aims to understand and arrive at a solution like the one behind the GOOGLE researcher that was able to return efficient responses to its users. With this the project tries to resemble itself using the structures that in theory and practice study.*

Resumo. *O projeto de Estrututura de Dados I e Linguagem de Programação I visa compreender e chegar a uma solução como o que está por trás do pesquisador da GOOGLE que consegui retornar respostas eficientes aos seus usuários. Com isso o projeto tenta assemelhar - se usando as estruturas que em teoria e prática estudamos.*

1. Introdução

O pesquisador da Google possui um algoritmo que possibilita busca por algo com rapidez e geralmente efetivas repostas, inicialmente fundado pelos estudantes Larry Page e Sergey Brin em 15 de setembro de 1997, o nome Google é oriundo da palavra Googol (lê-se gugol) denominação dada para o número 10^{100}, uma representação para a grande quantidade de conteúdo existente na Web. Com base nisso foi idealizado o projeto de Estrutura de Dados I e linguagem de Programação I com intuito de assemelhar-se ao maior buscador da atualidade o Google, utilizando as estruturas aprendidas em sala de aula.

2. Metodologia

O projeto foi concebido com base nos conceitos do *Pair Programming*, que é o desenvolvimento em parceria entre os programadores da equipe. Tendo em vista que o trabalho tem diversas etapas a serem desenvolvida, este método foi o que melhor se aplicou a necessidade do grupo, utilizando o controle de versão de arquivos do *Bitbucket* também promoveu colaboração e dinamismo ao trabalho. A análise empírica foi feita em um computador com Processador de Core i3 com 2.13 GHz, memória 2,8 GiB e disco 118,4 GB. Sistema operacional Ubutun versão 16.04 e compilador GNU, GCC/G++ versão 5.4.0. A liguagem utilizada foi a C++ com Orientação ao Objeto.

3. Algoritmos do Projeto

Os algoritmos utilizados neste projeto visam solucionar o problema com também trazer eficiência na execução, já que é um grande volume de dados. A descrição de cada um deles segue nas seções abaixo.

3.1. HashTable

A estrutura da Tabela Hash foi escolhida para realizar as diversas buscas que o programa tem que proporcionar ao usuário, pois ela na maioria das vezes torna o tempo de resposta constante, fazendo com que o programa tenha eficiência diante de outras estruturas estudadas. Denominada **HashTable**, calcula um valor denominado de hash para cada palavra distinta que será o índice de localização, feito isso dentro desse hash cria - se uma **HashObject** que é uma lista, possuindo em seus atributos um ponteiro para outra classe **HashItem** possuidora dos atributos chave e dado.

3.2. LinkedList

É uma classe que se comporta como uma Lista Duplamente Encadeada podendo ser de qualquer tipo, pois é feita com Template e a classe **Node** é responsável em armazenar os valores também de qualquer tipo. Apartir da **LinkedList** outras Listas herdam dela alguns métodos e atributos, dependendo da necessidade, são elas: **LinkedListBase**, **LinkedListRemovable** e **LinkedListInsertable**.

3.3. Buble Sort

O algoritmo de ordenação foi implementado usando lista duplamente encadeada ao invés de vetor. Por ser mais fácil de implementar, tornou - se a melhor solução para o problema de ordenação tendo em vista o tempo para solucionar o problema.

3.4. Outros

A solução para um problema como o proposto pelas disciplinas, tem que buscar estratégias para diminuir a complexidade e assim trabalhar com partes estejam dentro da possibilidade de ser solucionado, com isso, nesta seção está descrito todos os demais algoritmos que promovem esse papel.

3.4.1. Classe Entrada

Caracteriza - se em receber as entradas que o usuário precisa realizar para conseguir executar o que deseja.

3.4.2. Classe Auxiliares

Essa classe possui métodos importantes no sentindo de tratamento do texto, ou seja, permite que após usado seus métodos fique somente o que interessa do arquivo para realizar as buscas que são as palavras dentre outras aplicações.

3.4.3. Classe ControladorArquivos

Nesta classe basicamente todos os arquivos que serão inserido e fornecerá a **base de dados**, terá como funcionalidade leitura e armazenamento do arquivo, como também diminuir alguns problemas que podem surgem caso fosse trabalhar com o texto sem qualquer tipo de tratamento.

3.4.4. Classe Timer

Classe que captura o tempo corrido do sistema usando a biblioteca **sys/time.h** do UNIX, podendo medir o tempo de execução de método como por exemplo.

3.4.5. Classe Serializable

Esta classe é utilizada em muitas outras pois ela consegui transformar em outro objeto, deixando em uma formato de armazenamento.

3.4.6. Main.cpp

É o código responsável em receber as entradas (do usuário) iniciais para inserção, remoção, lista arquivos por ordem de inserção e por ordem alfabética, o fará com seja gerador da base de busca.

3.4.7. Busca.cpp

É o código responsável em receber as entradas (do usuário) para realizar as outras buscas, como: AND, OR, buscar por palavra nos arquivos.

4. Implementações - Imagens

```
/**
    Classe que é a estrutura escolhida para manter constante as buscas.
*/
template <class T>
class HashTable
{
    private:
        typedef HashObject<HashItem<T>>* HashObjectPtr;
        int tamanho;
        int qtdObjetos;
        HashObjectPtr* tabela;

    public:
        HashTable<T>(int tamInicial)
        {
            //Construtor
            this->tamanho = tamInicial;
            this->qtdObjetos = 0;
            this->tabela = new HashObjectPtr[tamInicial];

            for(int i = 0 ; i < tamInicial ; i++)
            {
                this->tabela[i] = NULL;
            }
        }

        ~HashTable<T>()
        {
            // Destrutor da classe
        }
```

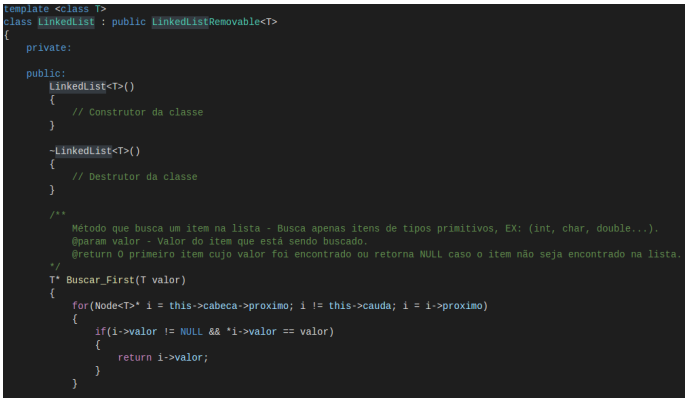
Figura 1. Classe HashTable

```cpp
/**
    Estrutura do objeto armazenado na tabela de dispersão.
*/
template <class T>
class HashObject
{
    public:
        LinkedList<T>* itens;

        HashObject<T>()
        {
            // Construtor
            this->itens = new LinkedList<T>();
        }

        ~HashObject<T>()
        {
            // Destrutor da classe
        }

        /**
            Método para Inserir um item de qualquer tipo da lista.
            @param - Item de qualquer tipo contido na lista.
            @return - não retorna.
        */
        void InserirValor(T* item)
        {
            this->itens->Inserir(item);
        }
```

Figura 2. Classe HashObject

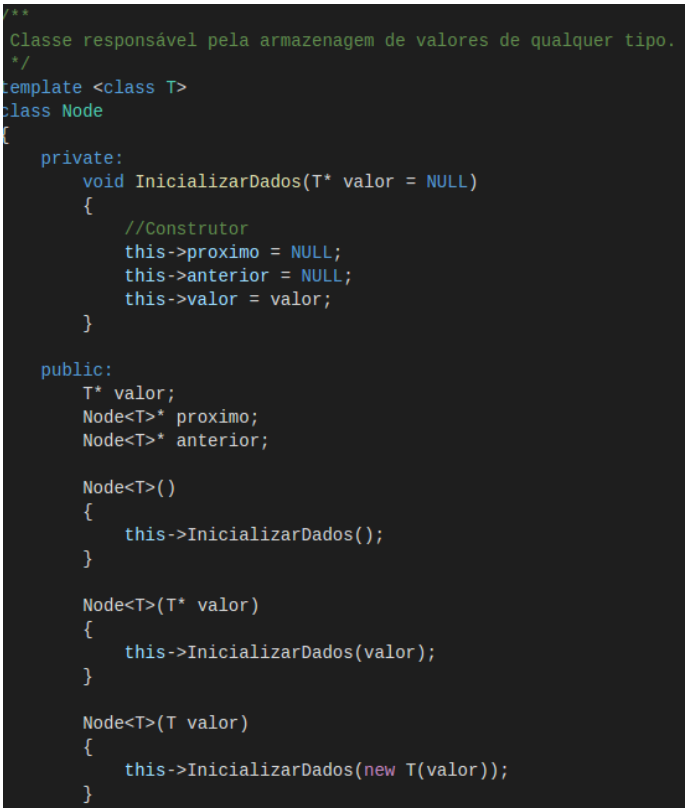
```cpp
/**
    Estrutura do item armazenado no objeto da tabela de dispersão.
*/
template <class T>
class HashItem
{
    public:
        string chave;
        T* dado;

        HashItem<T>(string chave, T* dado)
        {
            // Construtor
            this->chave = chave;
            this->dado = dado;
        }

        ~HashItem<T>()
        {
            // Destrutor da classe
        }
};

#endif
```

Figura 3. Classe HashItem



```
template <class T>
class LinkedList : public LinkedListRemovable<T>
{
    private:

    public:
        LinkedList<T>()
        {
            // Construtor da classe
        }

        ~LinkedList<T>()
        {
            // Destrutor da classe
        }

        /**
            Método que busca um item na lista - Busca apenas itens de tipos primitivos, EX: (int, char, double...).
            @param valor - Valor do item que está sendo buscado.
            @return O primeiro item cujo valor foi encontrado ou retorna NULL caso o item não seja encontrado na lista.
        */
        T* Buscar_First(T valor)
        {
            for(Node<T>* i = this->cabeca->proximo; i != this->cauda; i = i->proximo)
            {
                if(i->valor != NULL && *i->valor == valor)
                {
                    return i->valor;
                }
            }
```

Figura 4. Classe LinkedList



```
/**
 Classe responsável pela armazenagem de valores de qualquer tipo.
 */
template <class T>
class Node
{
    private:
        void InicializarDados(T* valor = NULL)
        {
            //Construtor
            this->proximo = NULL;
            this->anterior = NULL;
            this->valor = valor;
        }

    public:
        T* valor;
        Node<T>* proximo;
        Node<T>* anterior;

        Node<T>()
        {
            this->InicializarDados();
        }

        Node<T>(T* valor)
        {
            this->InicializarDados(valor);
        }

        Node<T>(T valor)
        {
            this->InicializarDados(new T(valor));
        }
```

Figura 5. Classe Node

```
/**
    Método que ordena lista duplamente encadeada
    @param - Duas lista de qualquer tipo e um retorno boleano.
    @return - Não possui retorno.
*/
void Ordenar(bool (*func)(T*, T*))
{
    for(Node<T>* item1 = this->cabeca->proximo; item1 != this->cauda; item1 = item1->proximo)
    {
        Node<T>* temp = item1->proximo;
        for(Node<T>* item2 = temp; item2 != this->cauda; item2 = item2->proximo)
        {
            if(func(item1->valor, item2->valor))
            {
                this->Intercalar(item1, item2);
            }
        }

        item1 = temp->anterior;
    }
}

/**
    Método que intercala valores contido nos Node.
    @param item1 e item2 do tipo Node.
    @return Inverte valores e passa por referência, pois o método é void.
*/
void Intercalar(Node<T>* item1, Node<T>* item2)
{
    T* temp = item1->valor;
    item1->valor = item2->valor;
    item2->valor = temp;
}
```

Figura 6. Buble Sort

```
/**
    Classe que recebe como entrada as escolhas do usúario.
*/

class Entrada
{
    private:
        int _iarg;
        int _argc;
        int _index;
        int _opterr;
        char** _argv;
        Funcao _tipoFuncao;
        LinkedList<string>* _parametros;
        bool Verify(char, char);
        void ValidaParametros();
        void ValidaParametrosBusca();
        void ValidarArgumentos();
        void CarregaArgumentos();
        void ExibeAjuda();
        void MsgParamInvalido();
        void MSGParamObrigNaoInformado();

    public:
        Entrada();
        ~Entrada();
        void ProcessarParametros(int n_argc, char* n_argv[]);
        void ProcessarParametrosBusca(int n_argc, char* n_argv[]);
        LinkedList<string> GetParametros();
        Funcao GetTipoFuncao();
};
```

Figura 7. Classe Entrada

```
/**
    Estrutura para tratamento do tipo da função.
*/
typedef enum tpFuncao{
    FUNC_INSERIR = 1,
    FUNC_REMOVER = 2,
    FUNC_LISTAR = 3,
    FUNC_LISTAR_ORDEM_ALFABETICA = 4,
    FUNC_LISTAR_ORDEM_QTD_PALAVRAS = 5,
    FUNC_BUSCAR = 6,
    FUNC_HELP = 7,
    FUNC_PARAM_INVALIDO = 8,
    FUNC_ARQUIVO_JA_EXISTE = 9,
    FUNC_ARQUIVO_N_EXISTE = 10,
    FUNC_AND = 11,
    FUNC_OR = 12
} Funcao;

/**
    Classe com métodos auxiliares.
*/
class Auxiliares
{
    public:

        static LinkedList<string> Split(const char* delimitador, string texto);
        static string RemoverCaracteresEspeciais(string texto);
        static string ToLowerCase(string texto);
};
```

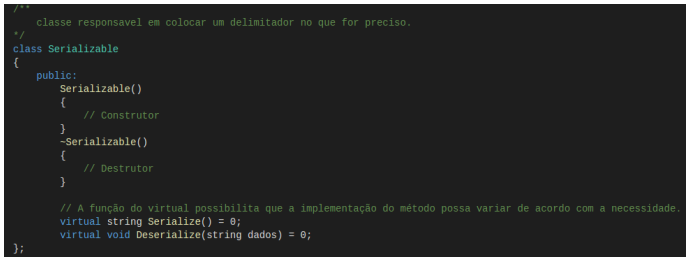
Figura 8. Classe Auxiliares

```
/**
    Constantes para os tipos de arquivos.
*/
#define DADOS_ARQUIVOS_TEMP "DataBase/Arquivos.data.temp"
#define DADOS_ARQUIVOS "DataBase/Arquivos.data"
#define DADOS_PALAVRAS "DataBase/DadosPalavras.data"
#define PASTA_BASE_DE_DADOS "DataBase"

/**
    Classe responsável em tratar os arquivos usados, transformando na base de busca e depois para as buscas.
*/
class ControladorArquivos
{
    private:
        static void ProcessarArquivo(Arquivo arquivo, HashTable<string>* tabela);

    public:
        ControladorArquivos();
        ~ControladorArquivos();
        static void GarantirBaseDeDados();
        static LinkedList<string> LerArquivoTXT(string nomeArquivo);
        static LinkedList<Arquivo> LerArquivo(string nomeArquivo);
        static string LerLinha(string nomeArquivo, unsigned int indexLinha);
        static bool InserirArquivo(string caminhoArquivo);
        static bool RemoverArquivo(string caminhoArquivo);
        static void ProcessarArquivos();
};
```

Figura 9. Classe ControladorArquivos

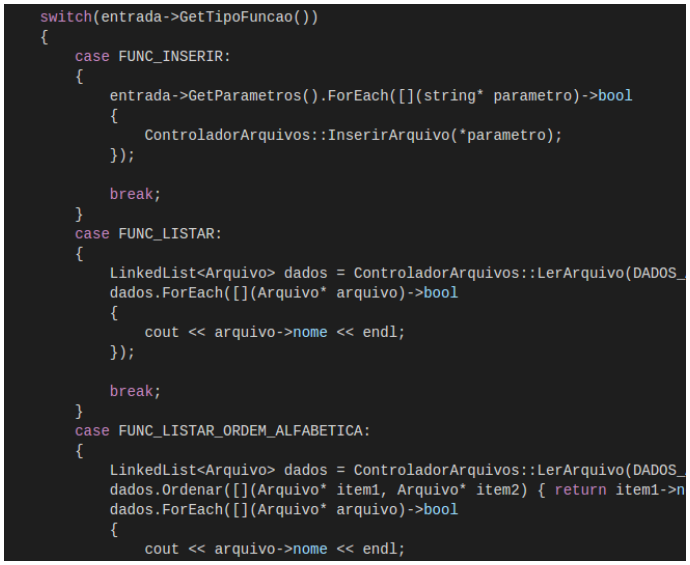
```
/**
    Classe utilizada para controlar as operações que utilizam medição de tempo.
*/

class Timer
{
    private:
        struct timeval _inicio;
        struct timeval _fim;
        int _tempo; // Tempo em milissegundos
    public:
        Timer();
        ~Timer();
        void Start();
        void Stop();
        int GetTime();
        int GetTimePrint();
};
```

Figura 10. Classe Timer



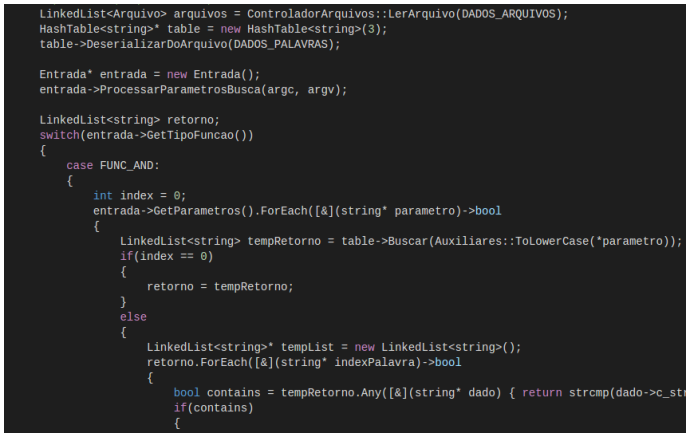
```
/**
    classe responsavel em colocar um delimitador no que for preciso.
*/
class Serializable
{
    public:
        Serializable()
        {
            // Construtor
        }
        ~Serializable()
        {
            // Destrutor
        }

        // A função do virtual possibilita que a implementação do método possa variar de acordo com a necessidade.
        virtual string Serialize() = 0;
        virtual void Deserialize(string dados) = 0;
};
```

Figura 11. Classe Serializable



```
    switch(entrada->GetTipoFuncao())
    {
        case FUNC_INSERIR:
        {
            entrada->GetParametros().ForEach([](string* parametro)->bool
            {
                ControladorArquivos::InserirArquivo(*parametro);
            });

            break;
        }
        case FUNC_LISTAR:
        {
            LinkedList<Arquivo> dados = ControladorArquivos::LerArquivo(DADOS_
            dados.ForEach([](Arquivo* arquivo)->bool
            {
                cout << arquivo->nome << endl;
            });

            break;
        }
        case FUNC_LISTAR_ORDEM_ALFABETICA:
        {
            LinkedList<Arquivo> dados = ControladorArquivos::LerArquivo(DADOS_
            dados.Ordenar([](Arquivo* item1, Arquivo* item2) { return item1->n
            dados.ForEach([](Arquivo* arquivo)->bool
            {
                cout << arquivo->nome << endl;
```

Figura 12. Busca.cpp



```
    LinkedList<Arquivo> arquivos = ControladorArquivos::LerArquivo(DADOS_ARQUIVOS);
    HashTable<string>* table = new HashTable<string>(3);
    table->DeserializarDoArquivo(DADOS_PALAVRAS);

    Entrada* entrada = new Entrada();
    entrada->ProcessarParametrosBusca(argc, argv);

    LinkedList<string> retorno;
    switch(entrada->GetTipoFuncao())
    {
        case FUNC_AND:
        {
            int index = 0;
            entrada->GetParametros().ForEach([&](string* parametro)->bool
            {
                LinkedList<string> tempRetorno = table->Buscar(Auxiliares::ToLowerCase(*parametro));
                if(index == 0)
                {
                    retorno = tempRetorno;
                }
                else
                {
                    LinkedList<string>* tempList = new LinkedList<string>();
                    retorno.ForEach([&](string* indexPalavra)->bool
                    {
                        bool contains = tempRetorno.Any([&](string* dado) { return strcmp(dado->c_str
                        if(contains)
                        {
```

Figura 13. Busca.cpp

5. Análise Empírica

A Tabela Hash é uma estrutura que em toeria mostra que sua complexidade é constante, no entanto ao realizar testes com a estrutura do projeto notou-se que o tempo cresceu

exponencialmente estando diretamente ligada ao tamanho de elementos(palavras). O resultado corrobora com as observações feitas pelo professor César Rennó, que sugeriu em um dos nossos encontros alterar a estrutura adotada que seria a divisão do problema em vários arquivos.

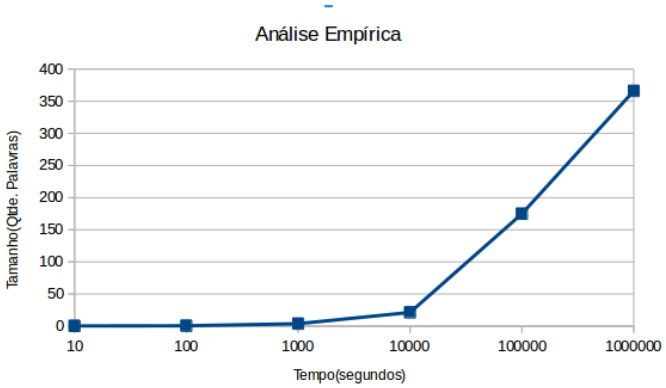


Figura 14. Gráfico de Análise Empírica.

6. Referências

Histórico da Google. Disponível em: <https://www.google.com.br/about/company/history/>. Acessado em 16 de novembro de 2016.

Renato Cardoso Mesquita. Manipulação de arquivos em C++. Disponível em: <http://www.cpdee.ufmg.br/ jramirez/disciplinas/cdtn/cap5-arquivos.pdf>. Acessado em: 28 de setembro de 2016.

Stroustrup, Bjarne. Princípios e práticas de programação em C++. 1ª Edição. Editora Bookman, 2012. 584 p.

<sys/time.h>. Disponível em:<http://pubs.opengroup.org/onlinepubs/7908799/xsh/systime.h.html>. Acessado em: 28 de novembro 2016.