Universidade Federal de Minas Gerais

Programação e Desenvolvimento de Software II - (PDSII)

Alunos: André Luiz Alves Costa e Bruno Henrique Evangelista Pereira

Trabalho Prático: Máquina de Busca

Introdução

O trabalho prático teve o objetivo de desenvolver uma máquina de busca feita a partir da linguagem de programação C++, onde em um banco de dados de arquivos txt, o usuário irá dizer o que deseja consultar e o programa desenvolvido retornará em quais arquivos as palavras dadas pelo usuário aparecem.

Ao início do projeto nos reunimos para decidir a estrutura da aplicação, a modelagem das classes que seriam necessárias e como seria implementado cada funcionalidade. Para isso foi necessário fazer uma abstração das funcionalidades aplicando os conhecimentos aprendidos na disciplina de PDS 2 e isso auxiliou na organização geral. E com as ideias bem organizadas podemos partir para a implementação do projeto de fato.

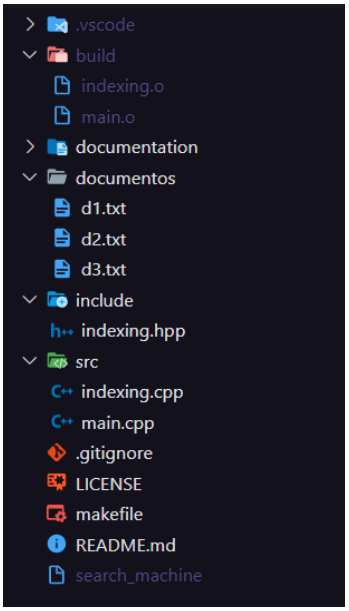
Implementação

Para versionamento do código foi utilizado o github, que funciona baseado em git, e também a utilização de pull requests para que os membros do projeto pudessem validar a implementação dos colegas de trabalho.

Para compilar o código foi utilizado o gcc version 11.3.0 rodando no WLS com o Ubuntu 22.04.1 LTS. E para facilitar os testes foi feito um makefile com os comandos para compilar todos os arquivos do projeto.

A estrutura de pastas utilizadas foi da seguinte forma:

- build: para guardar os arquivos de compilação;
- documentos: para a base de dados lida pelo programa;
- include: para os arquivos de cabeçalho;
- src: para os arquivos de implementação.



Decidimos que só seria necessário implementar uma classe indexing onde conteria as funcionalidades de indexação das palavras da base de dados e de recuperação delas.

```
#ifndef _INDEXING_H_
#define _INDEXING_H_

#include <map>
#include <set>
#include <vector>
#include <string>
#include <iostream>
#include <bits/stdc++.h>
#include <cstdio>
#include <cerrno>
#include <dirent.h>

using namespace std;

class indexing
{
private:
  map<string, set<string>> index;

  void insert(string, string);
  string normalize(string);

public:
  indexing();
  ~indexing();

  void recovery(vector<string>);
};

#endif
```

Para guardar as informações do index decidimos por utilizar um map responsável por manter a string dada pelo base de dados e o set de strings que são os nomes dos arquivos txt onde ela ocorre. Além disso os métodos insert e normalize estão no como privados pois não há necessidade de outros arquivos os acessarem, garantindo mais segurança para eles, enquanto o construtor, destrutor e recuperação serão acessados pela main.cpp para a pesquisa do usuário.

Partindo para implementação do indexing, como só seria necessário ler uma vez a base de dados, pareceu conveniente a funcionalidade de leitura dos arquivos estarem no construtor, então assim que o usuário iniciar o programa essa base de dados será lida.

```
3   indexing::indexing()
4   {
5     // Listando os arquivos do diretório ./Documentos
6     vector<string> files;
7
8     struct dirent **namelist;
9     int n;
10
11    n = scandir("./documentos", &namelist, NULL, alphasort);
12
13    if (n < 0)
14      perror("scandir");
15    else
16    {
17      while (n > 0)
18      {
19        if (namelist[n - 1]->d_name[0] != '.')
20          files.push_back(namelist[n - 1]->d_name);
21        delete namelist[n - 1];
22        n--;
23      }
24
25      delete namelist;
26    }
27
```

Como o nome dos arquivos dentro do diretório ./Documentos não é fixo, foi necessário pesquisar uma forma de saber o nome de todos os arquivos dentro de uma pasta, que é o que foi implementado na imagem acima. Utilizando as bibliotecas cstdio, cerrno e dirent.h, foi possível percorrer o nome de cada arquivo e guardá-los em um container vector.

```
28      // Inserindo as palavras dos arquivos no índice
29      fstream file;
30      string word, filename;
31
32      for (int i = 0; i < files.size(); i++)
33      {
34        filename = files[i];
35        file.open("./documentos/" + filename);
36
37        while (file >> word)
38        {
39          this->insert(word, filename);
40        }
41
42        file.close();
43      }
44    }
```

Agora utilizando a biblioteca bits/stdc++, foi possível abrir cada arquivo com o nome salvo no vector e inserir cada palavra com seu respectivo nome de arquivo com a função insert.

```
50  void indexing::insert(string word, string filename)
51  {
52    string normalized_word = this->normalize(word);
53    this->index[normalized_word].insert(filename);
54  }
```

A função insert, que recebe a palavra e o nome do arquivo onde ela foi encontrada, primeiro normaliza a palavra e depois a insere no index.

```
string indexing::normalize(string word)
{
  string normalized_word = "", aux;

  string comAcentos = "ÄÅÁÂÀÃäáâàãÉÊËÈéêëèÍÎÏÌíîïìÖÓÔÒÕöóôòõÜÚÛüúûùÇç";
  string semAcentos = "AAAAAAaaaaaEEEEeeeeIIIIiiiiOOOOOoooooUUUuuuuCc";

  for (int i = 0; i < word.length(); i++)
  {
    // Verificando se o caractere é maiúsculo
    aux = tolower(word[i]);

    // Verificando se o caractere tem acento
    for (int j = 0; j < comAcentos.length(); j += 2)
    {
      if (word[i] << word[i + 1] == comAcentos[j] << comAcentos[j + 1])
      {
        aux = semAcentos[j / 2];
        aux = tolower(aux[0]);
        break;
      }
    }

    // Verificando se o caractere é um caracter especial
    if (isalpha(aux[0]))
    {
      normalized_word += aux;
    }
  }

  return normalized_word;
}
```

Para permitir uma pesquisa mais precisa utilizamos o método normalize que recebe a palavra que será normalizada, e passando caractere por caractere dela, torna letras maiúsculas em minúsculas, através do tolower(), retira caracteres especiais, como pontuação, exclamação, vírgula, etc, com a função isalpha().

Fora isso temos que percorrer a string comAcentos comparando se ao caractere da palavra é acentuado, e se sim, substituí-lo por um caractere da string semAcentos. Uma dificuldade na implementação dessa função foi que mesmo que os caracteres acentuados pareçam visualmente ter o mesmo tamanho que um sem acentos, eles ocupam duas casas dentro de um vetor. Por isso foi necessário fazer a comparação de duas posições juntas da string word com a string comAcentos, e a posição de semAcentos dividido por dois, já que o laço estava sendo percorrido de dois em dois.

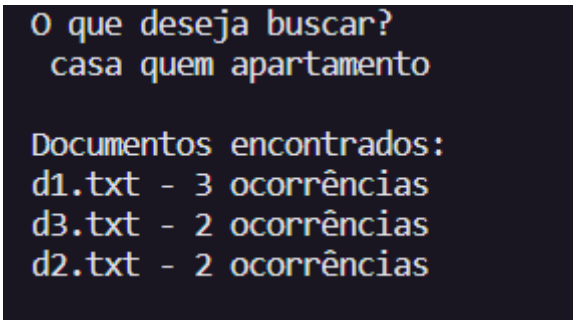
```cpp
void indexing::recovery(vector<string> query)
{
  set<string> normalized_query;
  string aux;

  // Normalizando a query
  for (int i = 0; i < query.size(); i++)
  {
    aux = this->normalize(query[i]);
    if (aux != "")
    {
      normalized_query.insert(aux);
    }
  }

  // Contando a quantidade de vezes que cada arquivo aparece na query
  map<string, int> relevant_files;

  for (auto it = this->index.begin(); it != this->index.end(); it++)
  {
    if (normalized_query.find(it->first) != normalized_query.end())
    {
      for (auto it2 = it->second.begin(); it2 != it->second.end(); it2++)
      {
        relevant_files[*it2]++;
      }
    }
  }
```

Agora para a função de recovery, responsável por retornar os arquivos com a pesquisa feita pelo usuário, precisamos primeiramente normalizar os parâmetros de pesquisa recebidos.

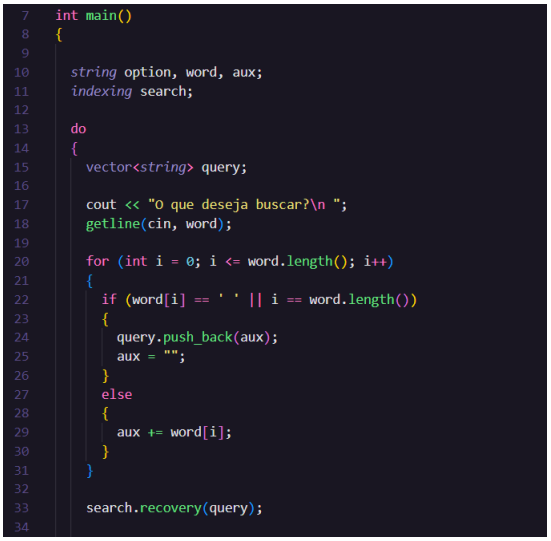
Feito isso, podemos então criar um map responsável por contar quantas vezes a pesquisa do usuário está presente no index. Dentro de um loop que percorre as chaves do index, o programa compara se a palavra do index está presente no container normalized_query, se sim então ele irá percorrer o conteiner set do index, que guarda os arquivos onde a palavra ocorre, e colocar o arquivo no container relevant_files, onde cada vez que ele aparecer na pesquisa irá ser somado mais um ao seu valor.

```cpp
  if (relevant_files.size() == 0)
  {
    cout << "\nNenhum documento encontrado!" << endl;
  }
  else
  {

    // Ordenando os arquivos por ocorrências na query
    set<pair<int, string>> ordered_files;

    for (auto it = relevant_files.begin(); it != relevant_files.end(); it++)
    {
      ordered_files.insert(make_pair(it->second, it->first));
    }

    cout << "\nDocumentos encontrados:" << endl;

    for (auto it = ordered_files.rbegin(); it != ordered_files.rend(); it++)
    {
      cout << it->second << " - " << it->first << " ocorrências" << endl;
    }
  }
```

Para que fosse possível ordenar os arquivos por ordem de relevância, foi necessário transferi-los para um container set, onde o valor de int, que representa a quantidade de vezes que aquele arquivo foi contabilizado na pesquisa, seria o parâmetro de ordenação do set. E no fim, imprimindo o container ordered_files teremos esse resultado no terminal:



Obs: Não conseguimos implementar a ordenação lexicográfica dos documentos que têm a mesma quantidade de ocorrências, pois o set implementado com pair<int, string> só leva em conta uma das chaves para ordenação, que no caso foi a chave int que representa as ocorrências. Pesquisando encontramos algumas soluções envolvendo bubble sort que vão além dos nossos conhecimentos.



```
int main()
{

  string option, word, aux;
  indexing search;

  do
  {
    vector<string> query;

    cout << "O que deseja buscar?\n ";
    getline(cin, word);

    for (int i = 0; i <= word.length(); i++)
    {
      if (word[i] == ' ' || i == word.length())
      {
        query.push_back(aux);
        aux = "";
      }
      else
      {
        aux += word[i];
      }
    }

    search.recovery(query);
```

```
    do
    {
      cout << "\n\nDeseja fazer outra pesquisa? (s/n)" << endl;
      cin >> option;

      if (option != "s" && option != "n")
      {
        cout << "Opção inválida!" << endl;
      }

    } while (option != "s" && option != "n");

    if (option == "s")
    {
      cout << "\n----------------------------------------\n"
           << endl;
    }

    cin.ignore();

  } while (option == "s");

  return 0;
}
```

Por fim, para que o usuário tenha uma interface onde utilizar o programa, no main foi implementado um menu com do/while. O usuário insere a string de pesquisa na variável através do getline, essa string é particionada palavra por palavra dentro de um vector para facilitar a manipulação, e é chamado a função de recovery passando este vector como parâmetro, retornando o resultado que

foi mostrado mais acima. Além disso, o usuário tem a opção de sair do programa ou realizar uma nova consulta.

Conclusão

Ao término da aplicação foi possível desenvolver não apenas conhecimentos a respeito da matéria e da linguagem C++, como também o desenvolvimento prático de um projeto.

A priori os fundamentos discutidos em aula foram fundamentais para o desenvolvimento do trabalho como boas práticas e modularização que permitiram uma boa compreensão e organização do código, bem como a programação orientada a objeto e controle de versão que tornaram possível o desenvolvimento do mesmo projeto em paralelo entre os integrantes.

Além dos conceitos discutidos em aula já citados, houve também a busca por informações mais técnicas da ferramenta como por exemplo as funções isalpha(), que verifica se o caracter é uma letra, tolower(), que transforma letras maiúsculas em minúsculas, além do getline(), que foi necessário utilizar a função cin.ignore() para que funcionasse mais de uma vez no menu do usuário.

Vale destacar que graças a esse TP descobrimos coisas como uma letra acentuada vale por 2 espaços numa string, por exemplo, a string comAcento tem o dobro do tamanho da semAcento, presentes no método normalize, e como um espaço sozinho gera um caractere inválido, necessitando percorrer esse vetor de duas em duas casas.

Concluindo, foi um projeto para treinar nossos conhecimentos básicos, mas também foi desafiador em certos pontos, o que agregou bastante para o aprendizado na disciplina.