Universidade Federal de Viçosa - Campus Florestal
Algoritmos e Estruturas de Dados II - CCF 212
Trabalho Prático 1

Trabalho Prático 01
Algoritmos e Estrutura de Dados II

Gabriel Miranda - 3857
Mariana Souza - 3898

13 de Setembro de 2021

Gabriel Miranda 3857
Mariana Souza 3898

Primeiro Trabalho Prático da disciplina de Algoritmos e Estrutura de Dados II

Primeiro trabalho prático da disciplina Algoritmos e Estrutura de Dados II.
O trabalho tem o intuito de implementar a busca de palavras-chave armazenadas em textos, em duas árvores, TST e PATRICIA.

Professora: Gláucia Braga

13 de Setembro de 2021

Sumário

1. Introdução

Neste trabalho prático foi desenvolvido uma máquina de busca, ou seja, pesquisar palavras-chave em textos, similar ao que ocorre quando realizamos uma pesquisa na Web, e para essa implementação foram usadas árvores digitais, sendo elas a patrícia e TST, que foram implementadas na linguagem C, ambas têm a característica de serem muito eficientes e rápidas em suas pesquisas por chaves, onde a patrícia e a TST estão armazenando palavras em suas chaves.

O trabalho prático desta disciplina também tem como objetivo executar um índice invertido, dado um arquivo, o índice invertido é uma estrutura que conta a quantidade de vezes que cada palavra irá se repetir dado um arquivo, fornecendo mais especificamente a saída da forma: <qtde, idDoc>, onde “qtde” é a quantidade de palavras que se repetem e idDoc é o número de identificação do documento que foi lido. Para o desenvolvimento dos códigos foram utilizados o compilador GCC, a IDE VSCode, e a plataforma GitHub para cooperação dos integrantes do grupo no processo de criação.

2. Objetivo

O objetivo do trabalho é aplicar os conhecimentos adquiridos em aula na disciplina Algoritmos e Estrutura de Dados II, implementando as árvores digitais Patrícia e TST, realizando a inserção de palavras e um índice invertido para árvore Patricia, onde é utilizado o auto complete inspirado na forma de pesquisa utilizada na web, e assim, realizando a leitura de um dicionário de palavras.

3. Desenvolvimento

Para o desenvolvimento do presente trabalho tomamos a decisão da divisão entre os integrantes do grupo, como: Gabriel realizando a implementação da patrícia, Gtk e os cálculos de relevância, Mariana implementando a TST e o auto complete, ambos para debugar e realizar a documentação. Realizando reuniões semanais para decidir caminhos de implementação para o projeto.

3.1 PATRICIA

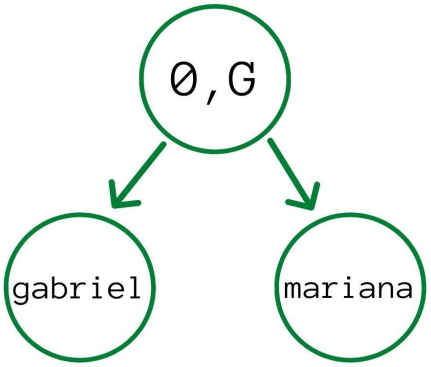
Para a construção do algoritmo árvore patrícia, foi criada uma estrutura que no nó interno tem-se dois campos que se fazem essenciais para a execução deste trabalho prático, que é o campo letra e índice, essa árvore PATRICIA é a mesma proposta pelo Ziviani.

```
typedef struct TipoPatNo *TipoArvorePat;

typedef struct TipoPatNo
{
  TipoNo nt;
  union
  {
    //nó interno
    struct
    {
      char letra; //armazena o caracter a ser comparado
      int indice; //armazena o índice a ser comparado
      TipoArvorePat Esq, Dir;
    } Ninterno;
    Tipochave chave;
  } NO;
} TipoPatNo;
```

Em linhas gerais no arquivo “*Pat.h*”, foi criada a struct “*TipoPatNo*”, nela tem-se os campos “*TipoNo*”, que é o tipo do nó da árvore se é externo ou interno, o tipo chave que contém a palavra que será armazenada e um campo para a lista de índices invertidos, no union, existem os campos letra e índice. A letra, armazena o carácter a ser comparado com as palavras quando inseridas e o índice demonstra qual posição deve ser comparada.

Exemplo:



Neste exemplo, primeiro foi inserido a palavra gabriel e em seguida a palavra mariana, no algoritmo só se tem nós internos quando se insere o segundo nó externo.

3.2 Funções da PATRICIA

Função "*patInsere*"

Depois que a palavra "*gabriel*" foi inserida na patrícia, a palavra "*mariana*" será inserida também. Isto acontece da seguinte forma. Primeiro é chamada a função "*patInsere*", que tem como parâmetros "*tipoArvore*", e a string que será passada no caso será "*mariana*", ao entrar na função tem-se a condição perguntando se a árvore é vazia, como no exemplo já foi inserida a palavra "*gabriel*" então a árvore não está vazia.

Em seguida existe uma parte da função que percorre os nós da árvore, e enquanto não é encontrado um nó externo, percorre-se a árvore comparando as palavras em determinado índice com o carácter salvo naquele nó interno, chamando a operação recursivamente para direita caso o caracter possua um valor maior, ou para a esquerda, caso possua um valor menor que o comparado. Adiante, achado o nó externo verifica-se se ele é igual ao passado, se for não será adicionado na PATRICIA.

Ademais, nesta função é criada um tupla que vai definir como será o nó interno. Para isso foi criada a função "*get_char*", que tem como parâmetros a palavra do nó externo que foi achado quando estava-se percorrendo a PATRICIA a fim de saber para qual lado a string passada iria, e a própria string passada. Nesta função é definido qual o tamanho das duas strings passadas e nela se verifica em qual posição essas duas palavras se diferem, retornando para a tupla a posição que difere e a menor letra da diferença. No exemplo de inserção da palavra "*mariana*" e "*gabriel*" essa tupla retornaria a posição 0, letra "G", por assim já se definindo como será o nó interno.

Função "*patInsereEntre*"

Na função "*patInsere*", é chamada a árvore, a palavra a ser inserida, e a tupla, nesta parte do código será definido onde a palavra será inserida, para isto existem duas verificações. A primeira é se o no passado é um nó externo ou o índice que está na tupla é menor que o índice que está no "*no.interno.indice*". Será criado um nó externo com a palavra passada, criando-se este nó externo em um apontador auxiliar, verifica-se agora onde a palavra será inserida na árvore.

```
//Escolhe se a string será inserida a esquerda ou à direita do novo nó interno

if (k[tupla.indice] > tupla.letra){
      return CriaNoInt(no, &Aux, tupla); //Aux para a Dir
    }
    else{
      return CriaNoInt(&Aux, no, tupla); //Aux para a esq
    }
```

Nesta figura, se a string na posição "*tupla.indice* > *tupla.letra*", esta palavra será inserida a Direita do nó interno criado, e o nó passado naquela posição será inserido à esquerda, e se não respeitar a condição o contrário. Passadas essas informações para o nó que será interno, retorna-se da função criar nó interno um nó interno com dois filhos.

A segunda parte importante desta função é se este nó passado não for um nó externo e o índice da tupla for maior que o índice do nó interno passado. Então teremos um "*else*" que percorrerá a PATRICIA até achar a posição em que o nó será inserido. Como no exemplo abaixo:

```
else{
    if (k[(*no)->NO.Ninterno.indice] > (*no)->NO.Ninterno.letra){
      (*no)->NO.Ninterno.Dir = patInsereEntre(&(*no)->NO.Ninterno.Dir, k,   tupla, id);
    }
    else{
      (*no)->NO.Ninterno.Esq = patInsereEntre(&(*no)->NO.Ninterno.Esq, k, tupla, id);
    }
    return (*no); //Retorna PATRICIA com nova palavra
  }
```

Esta parte da função chama recursivamente para a direita ou esquerda, até achar uma posição para inserir a palavra. Sempre chamando recursivamente para a mesma função, e assim que verifica que o nó da vez tem o índice maior que o da tupla ou se é um nó externo insere a palavra na patricia.

3.3 TST

Para a Ternary Search Tree (TST), é uma estrutura de dados que os nós filhos são ordenados como uma árvore de pesquisa binária, cada nó da TST contém três ponteiros, sendo assim cada nó da árvore TST pode ter até três filhos, o menor (esquerda), o igual (do meio) e o maior (direito), assim, o ponteiro esquerdo aponta para o nó que o valor é menor que o atual, o ponteiro igual aponta para o nó que o valor é igual ao nó atual, o ponteiro direito aponta para o valor que é maior que o valor do nó atual. Além disso, cada nó possui um campo para indicar os dados, e outro campo para marcar o final de uma string.

```
typedef struct node {
    char c; //letra
    unsigned ehOFimDaString: 1;
    int flag: 1;
    struct node *Esquerda;
    struct node *Igual;
    struct node *Direita;
} TipodoNOTST;
```

Dado os ponteiros apresentados acima na implementação foi criada uma struct do "*TipodoNOTST*", tendo armazenado uma estrutura node para os ponteiros, esquerda, igual e direita, que são usados nas funções para inicializar uma TST e inserir uma palavra na TST.

3.4 Funções da TST

Função "*TSTInsert*"

Depois da implementação da struct para a TST e a função para inicializar um árvore tst, temos a função apresentada na figura abaixo que mostra a função que insere uma palavra na tst, ela funciona com chamadas recursivas da função de inserção ("*TSTInsert*"), onde é

iniciada alocando dinamicamente um espaço para o "*NO*", que tem o "*TipodoNOTST*", e ele aponta para todos os ponteiros sendo direcionados para NULL, ou seja caso a palavra acabe, e uma "*flag*" que será verdadeira se o caractere for o último caractere de uma das palavras.

```
void TSTInsert(TipodoNOTST **NO, char *Letra){
    if (*NO == NULL){
        MALLOC(*NO, TipodoNOTST);
        (*NO)->c = *Letra;
        (*NO)->flag = 0;
        (*NO)->Esquerda = NULL;
        (*NO)->Igual = NULL;
        (*NO)->Direita = NULL;
    }
    if (*Letra < (*NO)->c)
        TSTInsert(&((*NO)->Esquerda), Letra);
    else if (*Letra > (*NO)->c)
        TSTInsert(&((*NO)->Direita), Letra);
    else if (*(Letra + 1))
        TSTInsert(&((*NO)->Igual), Letra + 1);
    else
        (*NO)->flag = 1;
}
```

Para imprimir a TST foi implementado a função "imprimeTSTAux" e "imprimeTST", na função que está sendo mostrada na figura acima temos a imprime auxiliar, que é um auxiliar para a função que imprime TST, essa função tem como objetivo percorrer a árvore e vai guardando no vetor "buffer" as palavras, caso seja o fim da palavra na árvore, o que está dentro do buffer é mostrado, é usado um define para definir o tamanho do buffer, sendo assim, dependendo do tamanho da árvore é necessário aumentar esse tamanho predefinido no buffer. A função que imprime a TST, contém a chamada da função auxiliar de impressão e também o tamanho do buffer que foi definido pelo define.

```
void imprimeTSTAux(TipodoNOTST *No, char *buffer, int depth){
    if (No){

        imprimeTSTAux(No->Esquerda, buffer, depth);
        buffer[depth] = No->c;

        if (No->flag){
            buffer[depth + 1] = '\0';
            printf("%s\n", buffer);
        }
```

```
        imprimeTSTAux(No->Igual, buffer, depth + 1);
        imprimeTSTAux(No->Direita, buffer, depth);
    }
}
```

Função "*AutoComplete*"

A função "*autocomplete*" está no arquivo "*tst.c*" onde, vai verificar o inicio do termo de acordo com o dicionário que foi inserido, optamos por escolher o dicionário com palavras em inglês para evitar possíveis erros, então após o termo ser digitado pelo usuário a função impressão retornará todas as possíveis palavras do arquivo que tem aquele termo que foi digitado. No código apresentado abaixo tem apontadores para a esquerda, direita e igual, onde "c" é a letra, no caso o termo a ser inserido, chamado recursivamente a função do autocomplete, assim, a flag que foi declarada no início, caso ela seja igual a -1 vai percorrer a palavra do dicionário e printar o termo que contém ela.

```
int auto_complite(TipodoNOTST *No, char *Letra, char *imprimir, int *i)

//verifica o inicio do termo de acordo com o dicionário inserido
{
    if (No == NULL)
        return 0;
    else if (*Letra < No->c)
        return auto_complite(No->Esquerda, Letra, imprimir, i);
    else if (*Letra > No->c)
        return auto_complite(No->Direita, Letra, imprimir, i);
    else{
        if (*(Letra + 1)){
            imprimir[*i] = No->c;
            *i += 1;
            if (No->flag == -1){
                int j;
                for (j = 0; j < *i; j++){
                    printf("%c", imprimir[j]);
                }
                printf("\n");
            }
            auto_complite(No->Igual, Letra + 1, imprimir, i);
            return 1;
        }
        else{
            imprimir[*i] = No->c;
            *i += 1;
            if (No->flag == -1){
```

```
                int j;
                for (j = 0; j < *i; j++){
                    printf("%c", imprimir[j]);
                }
                printf("\n");
            }
            impressao(No->Igual, imprimir, i);
            return 1;
        }
    }
}
```

3.5 Índice Invertido

Para fazer o índice invertido, foi necessária a criação da tad "*LIndice_invertido*", neste foi criada uma lista encadeada, com os campos "*qtd*" (representa a quantidade de palavras repetidas), e "*idDoc*" (representa a o id do arquivo lido). Para este trabalho era necessário que cada palavra tivesse um lista de índices, então esta lista foi inicializada quando se cria um nó externo, esta lista está dentro do campo chave, onde existe ela e a palavra que será armazenada, isto dentro da estrutura patricia. Em linhas gerais, quando se cria um nó externo é passado o identificador de um arquivo a palavra e a árvore, lá é inicializada a lista, é criada uma célula, onde nela é armazenada um id, e conta 1 no campo "*qtd*", já que apenas uma palavra foi armazenada.

E como saber quando criar uma nova célula para um novo id, ou somar mais um para um mesmo arquivo passado que tem mais de uma palavra repetida. Dentro da função "*patInsere*" existe um campo que verifica se a palavra passada já existe na árvore, neste campo é passado o id de um determinado arquivo para a função "*contapalavras*" que está no "*LIndice_invertido*", como mostrado abaixo:

```
if (strcmp(aux->NO.chave.palavra, k) == 0)
{
  Tlista *lista = aux->NO.chave.lista;
  verifica = ContaPalavras(lista, id);
  return (*no);
}
```

Dentro da função "*Contapalavras"*, é passada a lista de determinada palavra e o id do arquivo

da vez, se este id já estiver na lista da palavra, se soma mais 1 na sua quantidade, no caso na sua célula, e se ele não estiver chama a função para criar um novo índice, no caso uma nova célula com o id passado, como mostrado na figura abaixo:

```
int ContaPalavras(Tlista *lista, int id){
    TcelulaInvertida *Aux = lista->primeiro->prox;
    while (Aux != NULL){
        if (Aux->inver.idDoc == id){
            (Aux->inver.qtd)++;
            break;
        }
        else{
            Aux = Aux->prox;
        }
    }
    if (Aux == NULL){
        Insere_Iarquivo(lista, id);
        return 0;
    }
    return 1;
}
```

Lembrando que todos os índices invertidos estão sendo inseridos ordenadamente na lista.

3.6 Lista de documentos

No desenvolvimento do programa, para se efetuar o cálculo de relevância, foi desenvolvido o Tad "*listadocs*", para efetuar o cálculo era necessário saber a quantidade de arquivos inseridos no algoritmo, essa lista serviu bem. Na criação de sua estrutura de dados, foi criada uma célula com os campos "*id*" como sendo identificador do arquivo, o "*qtdpalavras*" que representa a quantidade de palavras que existem naquele arquivo, o "*nome*" referenciando o nome do arquivo, e um campo "*relevância*", para o cálculo de relevância. Já na estrutura "*Ldocs*" que representa a lista, existem dois ponteiros, o primeiro e o último, ademais, ainda existe o campo "*qtddoc*" que representa a quantidade de arquivos inseridos no programa. Segue abaixo a estrutura de dados desse tad:

```
typedef struct docs *Apontadodocs;

typedef struct docs
{
```

```
    int id;
    int qtdpalavras;
    char nome[50];
    double relevancia;
    struct docs *prox;
} Cdocs;

typedef struct
{
    int qtddoc;
    Cdocs *primeiro;
    Cdocs *ultimo;
} Ldocs;
```

Outrossim, na função "*main*" são declaradas duas lista de documentos a "*LArquivos*", que armazena todos os arquivos que são inseridos na PATRICIA, e os ordena por id, para isto existe a função "*insereDoc*" que fica dentro da função "*LeituraPat*". E a lista "*LArquivosPrio*" que recebe os arquivos e suas devidas relevância na função "*relevancia*" que fica no tad "*Pat.c/h*", todos os arquivos inseridos nesta lista são ordenados em ordem crescente pela suas, respectivas relevâncias calculadas, estes são ordenados na função "*insereRelevancia*".

3.7 Cálculo de Relevância

Para se efetuar o cálculo de relevância, primeiramente foi criada uma estrutura de dados chamada "*palavraprio*", que foi declarada no arquivo "*Pat.h*". Esta estrutura contém os campos "*nome*" que é o nome que será procurado na PATRICIA, o campo "*contf*" que é um ponteiro que representa o número de ocorrências do termo "*tj*" no documento *"i"*. Existe também o campo "*dj*" que representa o número de documentos na coleção, e por fim o campo "*w*" que é um ponteiro que terá em suas devidas posições o peso do termo tj no documento i. Segue abaixo a estrutura de dados:

```
typedef struct
{
  char nome[50]; //palavra
  int *contf;    //número de ocorrência do termo tj no documento i
  int dj;        //numero de documento na coleção
  double *W;     // peso do termo tj no documento i
} palavraprio;
```

Em decorrência da complexidade deste algoritmo será explicado passo a passo do que está acontecendo. De início na função principal "*main*" foi declarada uma variável do tipo "*palavraprio*" a variável que é um ponteiro chamasse "*palavras*". Ao decorrer do programa quando no menu é escolhida a opção 5, é perguntado ao usuário quantas palavras ele deseja buscar na PATRICIA para efetuar o cálculo de relevância. Dessa forma, é alocado para o ponteiro "*palavras*" que é do tipo "*palavrasprio*" a quantidade de palavras que será inseridas pelo usuário. Veja abaixo o trecho de código que exemplifica isso

```
printf("Digite a quantidade de palavras que deseja buscar:\n");
           scanf("%d", &qtd);
           palavras = (palavraprio *)malloc(qtd * sizeof(palavraprio));
```

Ou seja, a estrutura das palavras será um vetor, em que cada posição "i" representará um termo passado pelo usuário. Dessa forma, com os termos passados é chamada a função "*relevancia*". Nesta função são passados as seguintes variáveis, a própria PATRICIA (*no), a struct palavras, o "*Larquivo*" que contém todos os arquivos armazenados no programa, o "*Lprio*" que terá todos os arquivos armazenados por relevância, e por último a "*qtd*", que representa a quantidade de palavras passadas. Logo de início são criadas duas variáveis a (relevância) que será um vetor de tamanho da quantidade de arquivos inseridos no programa, e o peso que também será o vetor de tamanho quantidade de arquivos inseridos. Também é alocado na posição "i" de cada palavra o "*contf*" e o "*W*", que terão tamanho quantidade de arquivos inseridos na PATRICIA.

Depois da alocação dessas variáveis na posição "i" representando cada termo passado na estrutura palavras, é verificado em cada "*contf[j]*" quantas vezes aquele termo aparece na patricia em determinado "*id*". Exemplo: se foi inserido dois arquivos na PATRICIA o primeiro e o segundo, "*larquivo*", os insere ordenadamente em sua estrutura, então para uma determinada palavra na posição "j" do "*contf*" será armazenada quantas vezes ela aparece em determinado arquivo, além do for passado terá o ponteiro aux da "*larquivo*" que vai seguindo junto com o for passando o id corretamente. Então é chamada a função "*contpalavras*" que passa um id de arquivo a palavra na posição "i", e retorna quantas vezes ela apareceu naquele arquivo na posição "*palavra[i].contf[j]",* além disso como cada "*contf*" representa uma quantidade de palavras que se tem em um arquivo em determinado id, verifica-se se na

posição “j” do “*contf*” se existe essa palavra em determinado id, se existe soma 1 no campo “*dj*” da palavra.

Agora que já sabemos o número de ocorrências de cada termo passado em determinado arquivo, precisamos calcular o peso “*W*” do termo “tj” no documento “i”. Nesta parte da função foi percorrida a estrutura “*palavras*”, onde em cada posição da palavra para uma determinada posição “*W*”, são feitos os cálculos de peso como o valores que já foram achados.

E no fim da função é calculada a relevância, como todos os dado já estão armazenados na estrutura “*palavras*” fica bem fácil de fazer isto, como já dito foi criado um vetor de tamanho quantidade de documento inseridos, em que cada posição ele recebe a relevância de um determinado arquivo, e passa essa relevância e o nome do arquivo, pois a lista de arquivos está sendo percorrida também, dessa forma, ele cria uma nova lista de documentos ordenados por relevância.

```
cont = 0;
  Aux = larquivo->primeiro;
  for (int i = 0; i < larquivo->qtddoc; i++)
  {
    Aux = Aux->prox;
    contaTermos(no, Aux->id, &cont);
    // printf("Quantidade de termos:%d\n", cont);
    r[i] = (double)(1 / ((double)cont)) * (peso[i]);
    printf("r = 1/%d * %lf\n", cont, peso[i]);
    printf("r:%lf\n", r[i]);
    insereRelevancia(Lprio, r[i], Aux->nome, cont);
    cont = 0;
  }
```

3.8 GTK

Para a criação da interface do algoritmo foi usada a biblioteca <*gtk/gtk.h*>, aliada a ela ainda foi usado o *Glade* que deixou o desenvolvimento um pouco mais fácil já que se podia editar a interface diretamente. No mais, foi feita a interface usando-se funções do glade, o mais complicado disso tudo era puxar os botões para o código, e printar o conteúdo das árvores na interface.

4. Execução

Para a execução é necessário ter o gcc instalado em sua máquina para compilar o código que foi feito na linguagem C, são oferecidas duas opções para a compilação, executar no terminal (prompt de comando), basta executar o executável gerado com o comando . Será apresentado um Menu com as seguintes opções como está sendo representado abaixo pela imagem.

```
printf("|------------------------------MENU-----------------------------------|\n");
printf("| 1 - Construir o indice invertido usando a Patricia.                 |\n");
printf("| 2 - Inserir as palavras do dicionario na arvore TST.                |\n");
printf("| 3 - Imprimir o Indice Invertido.                                    |\n");
printf("| 4 - Imprimir as palavras da TST.                                    |\n");
printf("| 5 - Buscar por uma palavra na PATRICIA, a partir do indice construido.|\n");
printf("| 6 - Auto Complete.                                                  |\n");
printf("| 7 - Sair.                                                           |\n");
printf("|---------------------------------------------------------------------|\n");
```

Posteriormente basta inserir a opção desejada de 1 a 7, como está sendo apresentado na figura cada um dos números possui uma função relacionada.

Para executar utilizando uma interface mais amigável implementamos o GTK, que é uma ferramenta multiplataforma que permite a criação de interfaces gráficas, vale atentar que a biblioteca do gtk está disponível apenas para sistemas Linux, sendo assim, para executar utilizando a interface gráfica do gtk é preciso inserir o comando: *"gcc `pkg-config --cflags gtk+-3.0` -o programa main.c IndiceInvertido/indiceInvertido.c Patricia/pat.c TST/tst.c Lista/listadoc.c `pkg-config --libs gtk+-3.0`"*

5. Conclusão

Portanto, neste trabalho prático foi criado um algoritmo de pesquisa, nele foram implementados as árvores Patricia e TST, a TST sendo utilizada como dicionário e a Patrícia como armazenadora de texto, em que cada palavra armazenada existe uma lista de índices que mostram de qual arquivo a palavra é, e quantas vezes ela se repete neste arquivo.

Outrossim, foi criada a função autocomplete que seria capaz de auxiliar o usuário com palavras que já estão armazenadas no algoritmo. Em linha gerais esta função ajuda o usuário a autocompletar as palavras, basta digitar um prefixo que o algoritmo retorna as palavras com aquele prefixo passado.

Além disso, é importante ressaltar a estrutura de dados PATRICIA que se mostrou muito eficiente para armazenar as palavras dos textos passados e se mostrou uma ótima árvore de pesquisa.

6. Referências

visualizador TST:
https://www.cs.usfca.edu/~galles/visualization/TST.html
Implementação TST:
https://www.geeksforgeeks.org/ternary-search-tree/
Playlist GTK:
https://www.youtube.com/watch?v=9rBkGHJlGWg&list=PLV2Ns32vi3lBfkhsF5Ha3S8akVUazKS6K
Gtk com o vcpkg:
https://vcpkg.io/en/getting-started.html