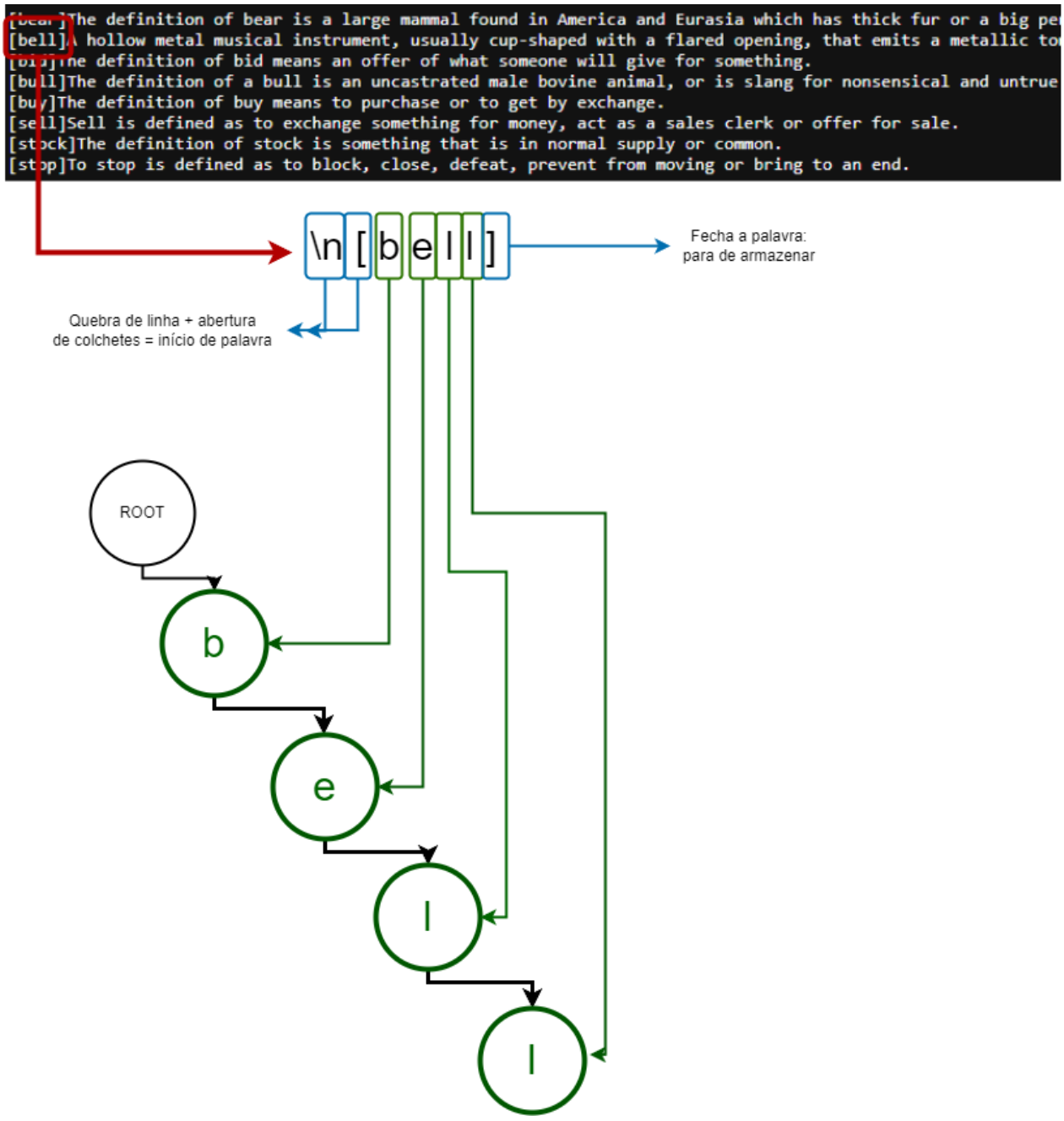
Nome: Tális Breda

Disciplina: INE5408 - Estruturas de dados

Relatório do projeto I

Explicação resumida

O algoritmo para resolver o problema consiste, basicamente, na classe TrieNode com um vetor de TrieNode*, que são os filhos. Na primeira parte, é feita a leitura do arquivo e o armazenamento na árvore. Na segunda parte, é feita a análise das palavras que foram colocadas como input



Explicação detalhada

Iniciando pelo main:

```
int main() {

    std::string filePath;
    std::cin >> filePath;

    std::string fileContents = readFileToString(filePath);

    TrieNode *root = new TrieNode();

    readAndStoreFileContents(fileContents, root);
    analyzeInputAndPrintResult(root);

    delete root;

    return 0;
}
```

As tarefas explícitas no main são bastante simples:

- Recebe o nome do arquivo pelo input
- Obtém a string com o conteúdo do arquivo
- Cria um ponteiro TrieNode que representa a raiz da árvore
- Lê o arquivo e armazena os dados na árvore
- Analisa os inputs do programa e exibe o resultado
- Deleta a raiz da memória ao fim do programa

readFileToString()

```
std::string readFileToString(const std::string& filePath) {
   std::ifstream file(filePath);

   if (!file.is_open()) {
       std::cerr << "Error opening file: " << filePath << std::endl;
       return "";
   }

   std::stringstream buffer;
   buffer << file.rdbuf();

   std::string fileContents = buffer.str();

   file.close();
   return fileContents;
}
```

Essa é uma função simples, que lê o arquivo e retorna seus conteúdos em formato string.

TrieNode

```
class TrieNode {
    public:

    char letra;
    TrieNode* filhos[26];
    int posicao;
    int comprimento;

    /* Construtor para nodos da lista */
    explicit TrieNode(char letra, int posicao) {
        this->letra = letra;
        this->posicao = posicao;
        this->comprimento = -1;
        initializeFilhos();
    }

    /* Construtor para o nodo raiz */
    TrieNode() {
        this->letra = '\0';
        this->comprimento = -1;
        initializeFilhos();
    }

    /* Destrutor recursivo */
    ~TrieNode() {
        for (auto i = 0u; i < 26; i++) {
            if (this->filhos[i] != nullptr) delete filhos[i];
        }
    }
```

Aqui estão os atributos, construtores e destrutor da classe TrieNode.
Atributos:

- **letra**: a letra que o nodo representa. A raiz não tem letra.
- **filhos**: a lista de filhos de cada nodo. É uma lista de ponteiros de TrieNode, de tamanho 26, representando, de forma ordenada, cada uma das letras do alfabeto
- **posicao**: o índice daquela letra dentro do arquivo
- **comprimento**: comprimento da linha a qual aquela letra pertence. Por padrão é -1, e apenas será alterado em nodos que representam a última letra de uma palavra.

A classe tem dois construtores. Um deles é para nodos que não são a raiz, recebendo letra e posição, e o outro é para a raiz que não representa uma letra. O destrutor é recursivo e tem a função de limpar a memória após o fim do programa.

TrieNode.initializeFilhos()

```
void initializeFilhos() {
    for (auto i = 0u; i < 26; i++) {
        this->filhos[i] = nullptr;
    }
}
```

Esse método da classe TrieNode tem a função de garantir que cada um dos elementos do vetor de filhos seja inicializado com um ponteiro nulo. Apenas para prevenir inconsistências no código.

TrieNode.contains()

```
bool contains(char value) {
    return this->filhos[value - 'a'] != nullptr;
}
```

Esse método checa se um determinado caractere existe na lista de filhos daquele nodo. A maneira que isso é feito é **value** - ‘a’, pois o tipo char é um número, com o ‘a’ representando 97 e as subsequentes letras do alfabeto representando os próximos números.
Ao reduzir o valor desejado de ‘a’, obtém-se a posição da lista onde a letra será armazenada, e as letras ficarão ordenadas. Isso aumenta a eficiência do algoritmo pois previne que seja necessário percorrer vetores.

TrieNode.countPrefixes()

```
int countPrefixes() {
    int count = 0;
    if (this->comprimento != -1) {
        count++;
    }
    for (auto i = 0u; i < 26; i++) {
        if (this->filhos[i] != nullptr) {
            count += this->filhos[i]->countPrefixes();
        }
    }
    return count;
}
```

Esse método serve para contar a quantidade de prefixos que uma palavra tem no dicionário. Isso é feito de forma recursiva, percorrendo toda a lista de filhos do nodo atual e chamando o mesmo método novamente. Quando o comprimento do nodo é diferente de -1, significa, pela lógica do programa, que o nodo atual é o final de uma palavra, e é adicionado 1 ao contador.

TrieNode.get()

```
TrieNode* get(char value) {
    return this->filhos[value - 'a'];
}
```

Esse método apenas retorna o elemento da lista de filhos que contém a letra passada no parâmetro.

readAndStoreFileContents()

```
void readAndStoreFileContents(std::string fileContents, TrieNode *root) {
    TrieNode *currentNode = root;

    bool inWord;
    int position;
    int lineLength = 0;
    char previousLetter = '\0';
    char letter = '\0';

    for (auto i = 0u; i < fileContents.size(); i++) {
        previousLetter = letter;
        letter = fileContents[i];

        /* Se a letra atual for um '\n', ou for a última do arquivo, resetamos o
comprimento da linha atual e voltamos para o nodo raiz */
        if (letter == '\n' || i == fileContents.size() - 1) {
            currentNode->comprimento = lineLength;
            lineLength = 0;
            currentNode = root;
            continue;
        } else { /* Se não, incrementamos o comprimento da linha atual */
            lineLength++;
        }

        /* Se estivermos na primeira letra ou tivermos um '\n' seguido de um
'[', iniciamos a leitura de uma palavra */
        if (i == 0 || (letter == '[' && previousLetter == '\n')) {
            inWord = true;
            position = i;
            continue;
        }

        /* Se estivermos dentro de uma palavra, adicionamos a letra atual ao
nodo atual */
        if (inWord) {
            if (letter == ']') {
                inWord = false;
            } else if (!currentNode->contains(letter)) {
                TrieNode* newnode = new TrieNode(letter, position);
                currentNode->filhos[letter - 'a'] = newnode;
```

```
                currentNode = newnode;
            } else {
                currentNode = currentNode->get(letter);
            }
        }
    }
}
```

Essa é a função relativamente mais complicada do programa, pois é responsável por analisar o arquivo caracter por caracter, e realizar as operações necessárias. Temos algumas variáveis para auxiliar nesse processo. São elas:

- **inWord**: boolean que, quando true, significa que a letra atual está dentro de uma palavra do dicionário, e deve ser armazenada.
- **position**: inteiro que recebe o índice da letra atual, utilizado no momento de instanciar um novo nodo.
- **lineLength**: comprimento da linha, incluindo palavra e significado.
- **letter**: letra atual
- **previousLetter**: letra anterior, utilizada para definir quando começa uma palavra.

Essa função funciona com um loop for, delimitado pelo tamanho da string do arquivo.

- A primeira operação é definir as variáveis **letter** e **previousLetter**.

- Depois, vamos conferir se a letra atual é a uma quebra de linha ou se é a última do arquivo. A conferência da última letra é por conta de não haver uma quebra de linha na última linha. Caso alguma dessas condições seja verdadeira, atribuímos **lineLength** ao **comprimento** do nodo atual, sinalizando que ele é o último de uma palavra, e voltamos **lineLength** e **currentNode** para os valores padrão, 0 e root, respectivamente.

- A segunda checagem é para quando a letra for a primeira do arquivo, ou quando for um ‘[‘ seguido de quebra de linha, significando que uma palavra está iniciando. Nesse caso, ativamos a flag **inWord**, sinalizando que estamos dentro de uma palavra, e atribuímos o índice a **position**, indicando a posição de início da palavra no arquivo.

- A terceira checagem é em relação à flag **inWord**. Caso ela seja true, faremos o tratamento das letras dentro de uma palavra. Dentro dessa condição, a primeira checagem é se a letra é um ‘]’, pois caso seja, indica que a palavra está terminando. Nesse caso, apenas desativamos a flag. Depois, utilizamos o método **contains()** para conferir se a letra existe entre os filhos do nodo atual. Caso não exista, instanciamos um novo nodo e atribuímos ele ao **currentNode**. Caso exista, apenas vamos até ele.

analyzeInputAndPrintResult()

```
void analyzeInputAndPrintResult(TrieNode *root) {
    while (true) {
        TrieNode* currentNode = root;
        std::string word;

        std::cin >> word;

        /* Quebra a execução caso o input seja 0 */
        if (word.compare("0") == 0) {
            break;
        }

        /* Viaja pelos nodos até chegar ao fim da palavra atual */
        bool isNotPrefix = false;
        for (auto i = 0u; i < word.size(); i++) {
            if (currentNode->contains(word[i])) {
                currentNode = currentNode->get(word[i]);
            } else {
                isNotPrefix = true;
                break;
            }
        }

        /* Conta a quantidade de prefixos que a palavra atual tem */
        int prefixCount = currentNode->countPrefixes();

        /* Imprime o resultado */
        if (isNotPrefix) {
            printf("%s is not prefix\n", word.c_str());
        } else if (currentNode->comprimento == -1) {
            printf("%s is prefix of %d words\n", word.c_str(), prefixCount);
        } else {
            printf("%s is prefix of %d words\n", word.c_str(), prefixCount);
            printf("%s is at (%d,%d)\n", word.c_str(), currentNode->posicao,
currentNode->comprimento);
        }
    }
}
```

Essa função trabalha com um loop while.

- A primeira operação é atribuir a palavra recebida à string **word**.
- Após isso, vamos checar se a palavra atual é um prefixo. Para isso, percorremos cada uma das letras da palavra, e caso a sequência esteja presente na árvore, iremos até o último nodo dela. Caso não esteja presente, ativamos a flag **isNotPrefix** e quebramos o loop para evitar problemas.
- Depois, utilizamos o método **countPrefixes()** para contar quantas palavras têm a palavra atual como prefixo.

- Após isso, exibiremos o resultado. Caso a flag **isNotPrefix** esteja ativada, significa que a palavra não é prefixo. Caso contrário, se o comprimento do nodo atual for -1, significa que a palavra atual não está no dicionário, e é apenas um prefixo. Caso nenhuma das condições anteriores seja verdadeira, a palavra está presente no dicionário, e é exibida a quantidade de prefixos e a posição da palavra no dicionário.

Conclusão

O trabalho acabou sendo mais complicado do que parecia, pois enfrentei diversos problemas com segmentation fault e gerenciamento de memória, por estar trabalhando com diversos ponteiros. Tive um problema onde o código funcionava no Valgrind mas não no GDB, e fiquei horas até descobrir um problema de memória que um programa conseguia lidar e outro não.

Referências

https://stackoverflow.com/questions/43297814/segfault-but-not-in-valgrind-or-gdb

https://stackoverflow.com/questions/4035769/invalid-read-of-size-8-valgrind-c

https://stackoverflow.com/questions/2456086/order-of-execution-for-an-if-with-multiple-conditionals