Projeto e Análise de Algoritmos

PCC 104- TP1 – 2020-1

Universidade Federal de Ouro Preto

Prof. Haroldo Gambini Santos

Elaborado por: Daniela Costa Terra

1. Tema

Sistema para auto completar textos, desenvolvido em linguagem C (C90), com emprego da estrutura de dados árvore Trie (Trie comprimida).

2. Árvores Trie: pesquisa digital

Pesquisa digital é empregada em aplicação que armazenam um texto e permite a busca por frases do texto. As características nesse caso são que as chaves de busca são frases e o tamanho da chave é, portanto, variável. Outro detalhe é que as chaves não cabem em uma palavra de computador.

Em pesquisa digital considera-se que uma chave é uma **sequência de dígitos** que podem ser usados na indexação. Ao invés de comparar a chave inteira, a **comparação é feita dígito a dígito.**

Alguns exemplos de estruturas para este problema são:

- **Árvore digital**: *R-way trie*, TST (*ternary search trie*)
- **Árvore digital binária**: *trie* binária
- **Árvore PATRICIA** (implementação eficiente de tries binárias)

Árvores **trie Tries** foram concebidas por Edward Fredkin, em 1960 quando estava trabalhando em recuperação de informação (*information retrieval*). Edward extraiu "*trie*" da palavra "*retrieval*". Uma **Trie** (pronuncie "trái") é um tipo de árvore usado para implementar uma tabela de símbolos (TS) de cadeias de caracteres.

Uma TS de *strings* nada mais é que um **dicionário** contendo pares (**chave**, **valor**), onde as chaves são do tipo cadeias de caracteres (*string*). Sedgewick e Wayne (2010) nomeiam árvores digitais de ***R-way trie.*** **R** é o tamanho do alfabeto das chaves e, portanto, cada nó na árvore pode ter até **R filhos.**

A Figura 1 abaixo ilustra uma árvore Trie. Os nós armazenam, além de links, o registro (**valor** ou *value*) corresponde à sua chave na tabela de símbolos. O *value* será **nulo** se o nó não corresponder a uma entrada válida.

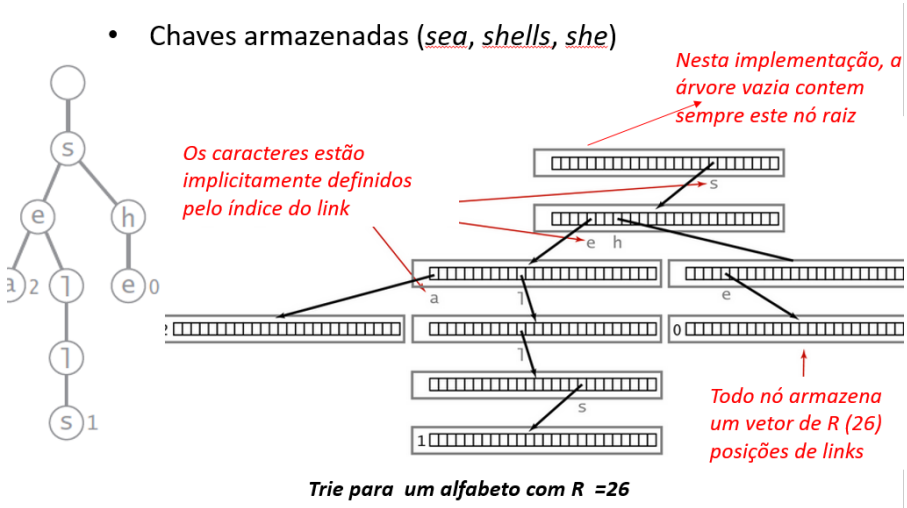

Figura 1: esquema de uma árvore trie
Fonte: adaptado de Feofillof (2018).

As chaves não são armazenadas na árvore, estão implícitas nos índices dos links que fazem o caminho da raiz até o nó que armazena o registro correspondente.

- Nesta estrutura, os nós com valores nulos* (*value* = *null*) não correspondem a chaves, existem para facilitar as pesquisas na *trie*

. Assim, com base em um alfabeto R de 26 dígitos, cada nó armazena, além do registros (Value) um vetor de R posições para os nós filhos. Essas árvores tem aplicações conhecidas como:

- Casamento de cadeias
- Dicionários, ferramentas do tipo *autocomplete*
- Pesquisas sobre DNA
- Análise de códigos

3. Complexidade dos algoritmos

A vantagem das árvores trie é localizar todas as ocorrências de uma determinada cadeia no texto, com tempo de resposta **O(*n*)** onde n é o tamanho do texto. Isso ocorre já que cada caractere da chave é extraído e comparado uma vez. Portanto, o tempo de execução de operações na árvore (inserção, retirada, busca) é **O(n)** e este tempo não depende do número de elementos da árvore. Também, a altura da árvore é igual ao comprimento da chave mais longa no texto.

Complexidade: **O (n)**. Onde **n** o tamanho da chave.

Deve-se observar que a utilização de uma TRIE só compensa se o acesso aos componentes individuais das chaves for bastante rápido. Outro aspecto é relativo ao espaço. Quanto maior for a estrutura, a quantidade de palavras armazenados na árvore, mais eficiente o uso do espaço. Para enfrentar o desperdício de espaço outras estruturas são indicadas como as árvores PATRÍCIA.

4. Estrutura e algoritmos

A estrutura definida nos códigos segue o **tipo abstrato de dados** adaptado de Sedgewick e Wayne(2011) e Feofillof (2018). A seguir a definição da estrutura de dados uma Trie e de cada nó da árvore , em C:

```c
#define R 256  //base (tamanho do alfateto)

// Define Estrutura para árvore trie
struct trie{
    struct noTrie *root;
};
struct noTrie {
    int isReg;   //1 se o nó tem value ou 0 caso contário
    struct linkedlist *value;  //ponteiro para lista de ocorrências de palavras
    struct noTrie* next[R];  // R apontadores um para cada caractere do alfabeto das chaves
};
```

Figura 2: definição de structs em linguagem C para uma Trie e seus nós
Fonte: elaborado pela autora

Observe que cada nó possui um campo booleano (isReg) o qual indica se o nó contém um ou não um registro. A entrada value é na verdade um lista ligada de palavras e o número de ocorrências da palavras no texto. R é a base do alfabeto, ou seja, 256, para cada dígito da tabela ASCII. Embora a sugestão fosse R = 26, para letras do alfabeto Inglês, optou-se por 256 tendo o vetor indexado pelo próprio caractere. Assim para o caractere 'e' a posição em **next** é dada por R[e].

As principais operações definidas são descritas abaixo. Outras operações de apoio estão inclusas no código em anexo.

- **struct noTrie getTrie(char *key, struct trie* arvore)**: pesquisa a chave ***key*** na árvore e retorna o registro (*Value*) em noTrie ou *null* se ***key*** não for encontrada
- **struct noTrie void putTrie(char* key, struct linkedlist* value, struct trie* arvore)**: insere a entrada (***key***, ***value***) na árvore. Se a ***key*** já existir **value** atualizará o número de ocorrências de Key
- **int keysWithPrefix(char* pre, struct linkedlist* value, char*** ocorrencias)**: retorna um vetor com todas as ocorrências de chaves da *trie* que têm o prefixo *pre.*

Para a função keysWithPrefix() é importante ressaltar que todas as listas ligadas de todos os nós descendentes do prefixo dado serão concatenadas em uma única lista (linkedList). Em seguida, esta lista é ordenada (método bolha melhorado) em um vetor e os elementos retornados em *ocorrências*.

A definição da lista ligada usada para representar o registro (valeu) dos nós da Trie é ilustrada abaixo. Esta é uma estrutura sem célula cabeçalho. Cada ocorrência a mais de uma palavra no texto faz incrementar o campo **ocorrencias** na estrutura noList da Figura 2.

```
// Define uma linkedlista para armazenar as ocorrência de palavras em um nó
struct linkedlist{
    int numPalavras;
    struct noList *inicio;
};

// Define um nó da lista de ocorrência de palavras
struct noList {
    int ocorrencias;
    char* palavra;
    struct noList *prox;
};
```

Figura 3: definição de structs para uma lista ligada que armazena as ocorrências de chaves (value dos nós da trie)
Fonte: elaborado pela autora

5. Código e experimentos

O programa aceita os argumentos de linha de comando para indicar o caminho do arquivo e o modo de execução (interativo ou experimentos). A função main foi escrita da seguinte maneira:

1. Leitura dos argumentos de linha de comando
2. Se o argumento de modo de operação é igual a '-interactive'
    a. O arquivo é aberto, cada linha é lida e processada para extrair caracteres especiais: /*@()_-!?:%%[]{}<>.,;'"
    b. Cada palavra extraída é inserida na trie usando a função **putTrie**()
        i. Se chave não for encontrada um novo nó irá armazenar uma lista ligada de 1 elemento contendo (chave, nº ocorrência)
        ii. Se chave já existir, o número de ocorrências na lista ligada para a chave existente é incrementado de 1
    c. Após construção da árvore um laço é iniciado para oferecer sugestões de **auto complete** para o prefixo informado, até que o usuário digite ***0***.
3. Se o argumento de modo de operação é igual a ' -exp' são feitos 3 experimentos computando respectivos tempos de execução:
    a. A montagem da árvore para leitura do arquivo uma vez
    b. A montagem da árvore para leitura do mesmo arquivo 10 vezes
    c. A montagem da árvore para leitura do mesmo arquivo 100 vezes

A Figura 4 abaixo mostra as telas de saída para o arquivo “baskervilles.txt” no modo interativo.

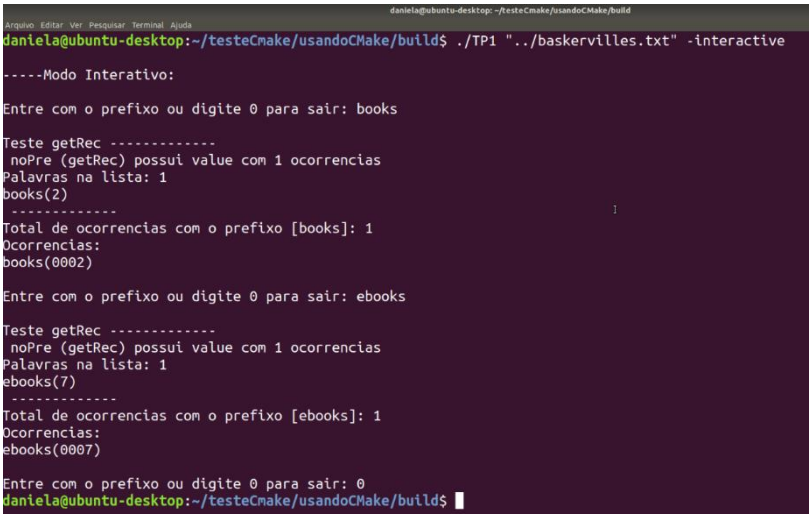

Figura 4: sugestão de palavras a partir dos prefixos com ocorrências de palavras no texto
Fonte: elaborado pela autora

A Figura 5 abaixo apresenta os experimentos feitos par ao mesmo arquivo “baskervilles.txt”:

- Experimento 1: tempo de execução para 1 leitura, extração das palavras e registro das ocorrências e prefixos na trie
- Experimento 2: tempo de execução para 10 leituras do mesmo arquivo. Repetição de abertura e fechamento do mesmo arquivo 10 vezes, extração das palavras e registro das ocorrências e prefixos na trie
- Experimento 3: tempo de execução para 100 leituras do mesmo arquivo. Repetição de abertura e fechamento do mesmo arquivo 100 vezes, extração das palavras e registro das ocorrências e prefixos na trie

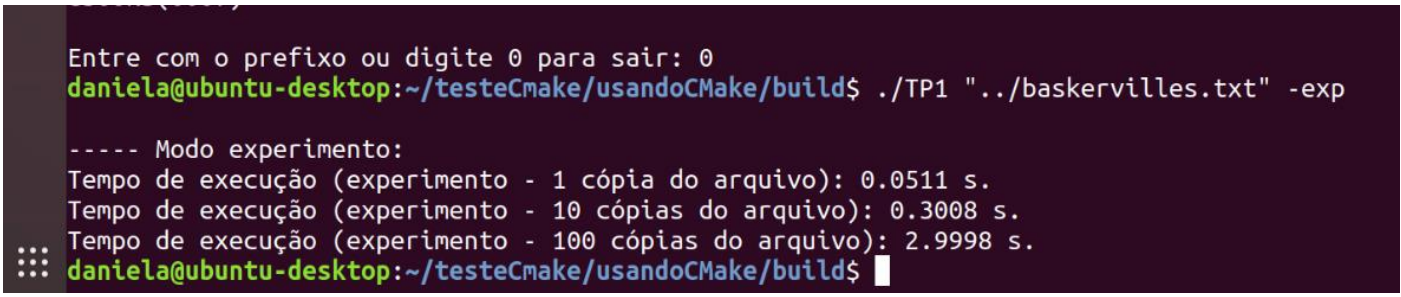

Figura 5: tempos de execução para 1, 10 e 100 cópias do arquivo “baskervilles.txt”
Fonte: elaborado pela autora

A seguir o código das principais operações da trie para os testes acima (vide Figura 6, 7 e 8).

```
//Preenche todas as ocorrencias de palabras com o prefixo 'pre' informado
//no vetor ocorrencias e retorna o tamanho do vetor, ou -1 se 'pre' não encontrado
int keysWithPrefix(char *pre, struct trie* arvore, char*** ocorrencias){
    //criando uma linkedlist para lista geral vazia para armazenar todas as listas do nós trie que são registros (noTrie->isReg = 1)
    struct linkedlist* q = (struct linkedlist*) malloc(sizeof(struct linkedlist));
    q->numPalavras = 0;
    q->inicio = NULL;

    struct noTrie* noPre = getRec(arvore->root, pre, d: 0);

    //Teste para prefixo encontrado!
    if (noPre == NULL)
        return -1;
    // Teste
    printf( format: "\nTeste getRec -------------");
    printf( format: "\n noPre (getRec) possui value com %d ocorrencias", noPre->value->numPalavras);
    imprime (noPre->value);
    printf( format: "\n -------------");

    collect(noPre, pre, q);

    struct noList** v;
    v = (struct noList**) malloc( size: q->numPalavras * sizeof(struct noList*));
    int tam = ordenaOcorrencias(q, v);

    *ocorrencias = (char**) malloc( size: tam * sizeof(char*));
    geraVetorOcorrencias(v, *ocorrencias, tam);

    free(q);
    return tam;
}
```

Figura 6: função keyWithPrefix() que retorna as sugestões a partir de determinado prefixo, em ordem de ocorrências no texto
Fonte: elaborado pela autora

As funções putTrie() e getTrie() chamam, cada uma outra função de apoio recursiva, como ilustrado nas Figuras 7 e 8, a seguir.

```
//Função para inserção de uma chave (palavra) na trie
void putTrie(char *key, struct trie* arvore){
    arvore->root = putRec(arvore->root, key, d: 0);
}

//Função recursiva de apoio à putTrie()
struct noTrie* putRec(struct noTrie* x, char* key, int d){
    if (x == NULL){
        //cria novo noTrie com uma lista de ocorrências vazia
        x = (struct noTrie*) malloc(sizeof(struct noTrie));
        x->isReg = 0;   //a principio um novo nó não contem registro
        x->value = (struct linkedlist*) malloc(sizeof(struct linkedlist)); //cria lista vazia (0 nós)
        // x->next terá 256 ponteiros iguais a (nil)
    }
    if (d == strlen(key)){
        x->isReg = 1;
        insere(x->value, key, numOcor: 1);
        return x;
    }
    char c = key[d];
    x->next[c] = putRec(x->next[c], key, d: d + 1);
    return x;
}
```

Figura 7: funções getTrie() para inclusão e busca de prefixo na árvore trie
Fonte: elaborado pela autora

```c
//Retorna a lista de ocorrências da chave (palavra)  na árvore.
//ou NULL se palavra não encontrada
struct linkedlist * getTrie(char * chave, struct trie* arvore){
    struct noTrie *no = getRec(arvore->root, chave,  d: 0);
    if (no == NULL)
        return NULL;
    return no->value;
}

//Função recursiva de apoio à getTrie()
struct noTrie* getRec(struct noTrie* x, char* key, int d){
     if(x == NULL)
         return NULL;  //key não encontrada na árvore trie;
     if(d == strlen(key))
         return x;
     char c = key[d]; //obtem o d-ésimo caractere da chave
     return getRec(x->next[c], key,  d: (d + 1));
}
```

Figura 8: funções getTrie() para inclusão e busca de prefixo na árvore trie
Fonte: elaborado pela autora

A compressão da trie como ilustra a Figura 9 da direita pode ser feita no retorno das chamadas recursivas a partir dos nós folha. Todo nó cujo pai possuir apenas 1 filho devem ser eliminados e ter suas ocorrências recolhidas. Essas darão origem a uma única linkedlist que ficará armazenada no nó imediatamente abaixo do pai que possuir mais de um filho. Essa busca do pai é feita no caminho de retorno da recursão, para a busca dos nós folha.

A vantagem da compressão é a otimização dos tempos de busca. Uma desvantagem é a necessidade de armazenar outra estrutura (como uma lista) para conter todas as palavras de mesmo prefixo, no mesmo nó trie.

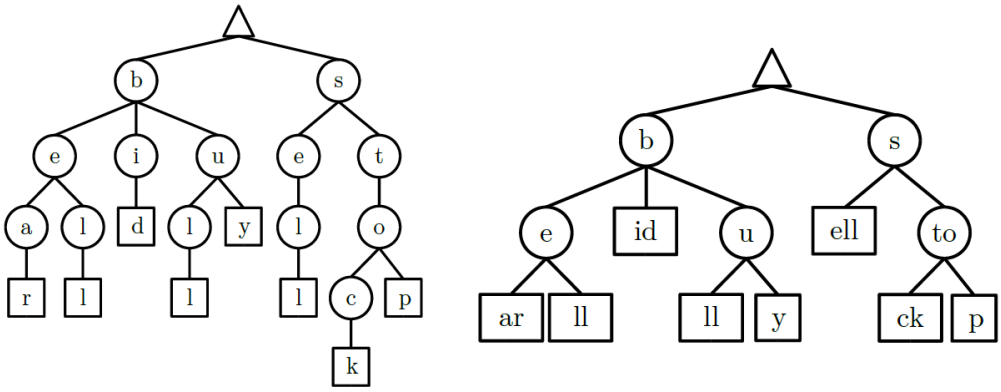

Figura 9: arvore trie convencional (à esquerda). Versão comprimida à direita.
Fonte: materiais de aula da disciplina

6. Referências bibliográficas

R. Sedgewick and K. Wayne, *Algorithms*, 4th Edition, Addison-Wesley, 2011.

www.ime.usp.br/~pf/estruturas-de-dados/ Last modified: Wed Jul 18 09:56:03 BRT 2018. *Paulo Feofiloff*