Trabalho Prático 2
Algoritmos e Estruturas de Dados II

Lucas Caetano Lopes Rodrigues

Belo Horizonte
2017

1. Introdução

A utilização de estrutura de dados que facilitem a pesquisa pelos dados e a inserção dos dados é fundamental para áreas da computação que precisam ser otimizadas e que trabalham com uma grande quantidade de dados e para aplicações reais que têm necessidade de rodar em tempo hábil.

O objetivo deste trabalho é implementar as estruturas ***árvore binária*** e ***hashing*** para armazenamento das palavras de um arquivo *.txt*, assim como implementar uma *lista dinamicamente encadeada* para armazenamento das linhas nas quais essas palavras se encontram.

Com isso, espera-se praticar os conceitos de manipulação de apontadores e espaços dinâmicos da memória, assim como leitura e escrita em arquivos e a lógica por trás das implementações.

2. Implementação

2.1 Estruturas de dados

- qItem

A estrutura qItem representa o item da *lista dinâmica* que armazena as linhas nas quais as palavras se encontram. Portanto, sua composição é apenas um inteiro, que indica o número da linha e é também sua chave.

```
typedef int Key;

typedef struct{
    Key key; //Linha da palavra
} qItem;
```

- qCell

A estrutura qCell é um nó da *lista dinâmica* que armazena as linhas nas quais as palavras se encontram. Ela é composta por um qItem e um apontador, que indica o próximo nó na lista.

```c
typedef struct QCell_str *qPointer;

typedef struct QCell_str{
    qItem content;
    qPointer next;
} qCell;
```

- Queue

A estrutura Queue é a própria *lista dinâmica* que armazena as linhas nas quais as palavras se encontram. Para isso, ela possui um apontador para o primeiro nó (ou célula *cabeça*) e um apontador para o último nó.

```c
typedef struct{
    qPointer first, last; //Ponteiros para nós das filas
} Queue;

void createEmptyQueue(Queue*);
int isQueueEmpty(const Queue*);
void insertInQueue(int, Queue*);
int deleteFromQueue(Queue*, qItem*);
```

- Funções e procedimentos:

1. `createEmptyQueue(Queue *q)`: cria uma fila vazia.
2. `isQueueEmpty(const Queue *q)`: checa se a fila recebida por parâmetro está vazia.
3. `insertInQueue(int item, Queue *q)`: enfileira o item *iitem* na fila *q*.
4. `deleteFromQueue(Queue *q, qItem *item)`: desenfileira o item *q*, armazenando seu valor no item *item*.
5. `void printQueue(Queue q, FILE *outputFile)`: Imprime a lista no arquivo especificado.

- Item

A estrutura Item representa o conteúdo do nó da árvore, e contém uma string e uma *Queue* das linhas nas quais a palavra pode ser encontrada.

```
typedef struct{
    char *word;
    Queue lines;
} Item;
```

- Tree

A estrutura Tree representa o nó da *árvore binária*, e contém um *Item* e um ponteiro para o nó da esquerda e um para o nó da direita.

```
typedef struct Tree_str *Pointer;

typedef struct Tree_str{
    Item content;
    Pointer left, right;
} Tree;
```

- Library

A estrutura Library representa a *árvore binária*, e, para isso, contém apenas um ponteiro para o nó raíz da árvore.

```
typedef Pointer Library;

void createTree(Pointer*);
int searchFor(char*, Pointer, Item*);
void insertIntoTree(char*, Pointer*, int);
void previous(Pointer, Pointer*);
int removeFromTree(char*, Pointer*);
void centralWalk(Pointer, FILE *);
```

- Funções e procedimentos:

1. `createTree(Pointer*)`: inicializa a árvore.
2. `searchFor(char*, Pointer, Item*)`: procura um item dentro da árvore.
3. `insertIntoTree(char*, Pointer*, int)`: insere um item na árvore.
4. `previous(Pointer, Pointer*)`: função utilizada na remoção de um item na árvore.
5. `removeFromTree(char*, Pointer*)`: remove um item da árvore.
6. `centralWalk(Pointer, FILE *)`: imprime os itens da árvore utilizando caminhamento central.

2.2 Programa principal

O programa principal lê os argumentos passados para decidir se ele vai chamar a implementação da estrutura **árvore binária** ou da estrutura **hashing**. Para tal, ele lê o parâmetro **`argv[3]`** e decide qual função chamar. Se o argumento for "**TREE**", ele chama a função **`_callTreeFunction`**, que então abre os arquivos de acordo com o caminho especificado em **`argv[1]`** e chama a função **`startTree`**.

A função **`startTree`** faz a leitura do arquivo *.txt*, passando as palavras para o diminutivo, contando as linhas e inserindo, uma a uma, na *árvore binária*. Após finalizar o arquivo e a inserção, a função **`centralWalk`** é chamada, para imprimir a árvore em ordem alfabética no arquivo de saída especificado.

Caso o parâmetro passado seja "**HASH**", o programa não fará nada, visto que a implementação do hashing não foi feita! (desculpa, professor).

3. Análise de complexidade

- `createEmptyQueue`: Não faz nenhum loop, apenas aloca dinamicamente a fila. Complexidade O(1).
- `isQueueEmpty`: compara o ponteiro inicial com o final da fila, para saber se ela está vazia ou não. Complexidade O(1).
- `insertInQueue`: Faz operações simples de alocação de memória e mudança de apontadores. Complexidade O(1).

- `deleteFromQueue`: Faz operações simples de liberação de memória e mudança de apontadores. Complexidade O(1).

- `printQueue`: Imprime no arquivo de saída os elementos daquela fila. Complexidade O(n), onde n é o número de elementos na fila.

- `createTree`: Inicializa o ponteiro da biblioteca como *null*. Complexidade O(1).

- `searchFor`: Busca recursivamente na árvore pelo registro, caminhando de acordo com a ordem pré-estabelecida.
  Complexidade: - Melhor caso: O(1).
  - Pior caso: O(n).
  - Caso médio: O(log n).

- `insertIntoTree`: Busca a posição na qual o nó “deveria” estar e, se não estiver presente, o insere na árvore.
  Complexidade: - Melhor caso: O(1).
  - Pior caso: O(n).
  - Caso médio: O(log n).

- `centralWalk`: Percorre todos os nós da árvore, imprimindo-os no arquivo de saída. Complexidade O(n).

- `_callTreeFunction`: Abre os arquivos e chama a inicialização da árvore. Complexidade O(1).

- `_callHashFunction`: Não faz nada. Complexidade O(1).

- `startTree`: Percorre o arquivo palavra por palavra, até o final, chamando as funções de passar as palavras para minúsculo e de inserir na árvore.
  Complexidade: - Melhor caso: O(n).
  - Pior caso: $O(n^2)$.
  - Caso médio: O(log n).

- `lowercase`: A função passa todas as letras de uma dada palavra para minúsculo. Sua complexidade é O(k), onde k é o número de letras da palavra.

4. Testes

Arquivo de entrada:

```
Beautiful Day - U2

The heart is a bloom
Shoots up through the stony ground
There's no room
No space to rent in this town

You're out of luck
And the reason that you had to care
The traffic is stuck
And you're not moving anywhere

You thought you'd found a friend
To take you out of this place
Someone you could lend a hand
In return for grace

It's a beautiful day
Sky falls, you feel like
It's a beautiful day
Don't let it get away

You're on the road
But you've got no destination
You're in the mud
In the maze of her imagination
```

Arquivo de saída:

```
after 49
all 30 31 49
and 9 11 31 42 48
anywhere 11
away 21 34 52
beautiful 1 18 20 33 35 51 53 64
bedouin 46
been 30 31
bird 48
bloom 3
blue 42
broken 44
but 24
came 49
can 61
canyons 44
care 9
case 40 58
china 43
clearing 45
cloud 44
colors 49
could 15
day 1 18 20 33 35 51 53 64 64
destination 24
doesn't 29
don't 21 34 52 60 60 61 62 62 63
even 29
falls, 19
feel 19 61
```

5. Conclusão

A implementação da **árvore binária** ocorreu como esperado. As maiores dificuldades foram: leitura correta das palavras no arquivo de entrada, inserção dos registros na árvore binária e tratamento dos ponteiros para os nós da árvore e para as listas de cada nó. O trabalho foi bem efetivo para por em prática os conceitos aprendidos em sala e exercitar a lógica de programação.

6. Referências

https://stackoverflow.com/

Anexos

Código no GitHub:

https://github.com/lucasclopesr/tp2

Listagem dos programas:

- Item.h
- Library.h
- qCell.h
- qItem.h
- Queue.h
- Tree.h
- Library.c
- Queue.c
- TP2.c

Item.h

```
#ifndef ITEM_H
#define ITEM_H

typedef struct{
      char *word;
      Queue lines;
} Item;

#endif
```

Library.h

```
#ifndef LIBRARY_H
#define LIBRARY_H

typedef Pointer Library;

void createTree(Pointer*);
int searchFor(char*, Pointer, Item*);
void insertIntoTree(char*, Pointer*, int);
void previous(Pointer, Pointer*);
int removeFromTree(char*, Pointer*);
void centralWalk(Pointer, FILE *);

#endif
```

qCell.h

```
#ifndef QCELL_H
#define QCELL_H

typedef struct QCell_str *qPointer;

typedef struct QCell_str{
      qItem content;
      qPointer next;
} qCell;

#endif
```

qItem.h

```
#ifndef QITEM_H
#define QITEM_H

typedef int Key;

typedef struct{
      Key key; //Linha da palavra
} qItem;

#endif
```

Queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

typedef struct{
     qPointer first, last; //Ponteiros para nós das filas
} Queue;

void createEmptyQueue(Queue*);
int isQueueEmpty(const Queue*);
void insertInQueue(int, Queue*);
int deleteFromQueue(Queue*, qItem*);

#endif
```

Tree.h

```
#ifndef TREE_H
#define TREE_H

typedef struct Tree_str *Pointer;

typedef struct Tree_str{
     Item content;
     Pointer left, right;
} Tree;

#endif
```

Library.c

```
#include "../lib/qItem.h"
#include "../lib/qCell.h"
#include "../lib/Queue.h"
#include "../lib/Item.h"
#include "../lib/Tree.h"
#include "../lib/Library.h"
#include <string.h>
#include <stdio.h>

#include "../src/Queue.c"

void createTree(Pointer *library){
     *library = NULL;
}

int searchFor(char *word, Pointer p, Item *i){
     if(p == NULL) return -1;

     if(strcmp(word, p->content.word) < 0){
          searchFor(word, p->left, i);
     } else if(strcmp(word, p->content.word) > 0){
          searchFor(word, p->right, i);
     } else {
          *i = p->content;
     }
}

void insertIntoTree(char *word, Pointer *p, int line){
     if(*p == NULL){
          *p = (Pointer) malloc(sizeof(Tree));
          (*p)->content.word = (char*) malloc(sizeof(strlen(word)));
          strcpy((*p)->content.word, word);
          (*p)->left         = NULL;
          (*p)->right        = NULL;
          createEmptyQueue(&((*p)->content.lines));
          insertInQueue(line, &((*p)->content.lines));
     } else if(strcmp(word, (*p)->content.word) < 0){
          insertIntoTree(word, &((*p)->left), line);
     } else if(strcmp(word, (*p)->content.word) > 0){
          insertIntoTree(word, &((*p)->right), line);
     } else {
```

```
            insertInQueue(line, &((*p)->content.lines)); //Registro já
existe na árvore;
      }
}

void previous(Pointer q, Pointer *r){
      if((*r)->right != NULL){
            previous(q, &(*r)->right);
            return;
      }

      q->content = (*r)->content;
      q = *r;
      *r = (*r)->left;
      free(q);
}

int removeFromTree(char *word, Pointer *p){
      Pointer aux;

      if(*p == NULL){
            return 0; //Registro não está na árvore;
      } else if(strcmp(word, (*p)->content.word) < 0){
            removeFromTree(word, &(*p)->left);
      } else if(strcmp(word, (*p)->content.word) > 0){
            removeFromTree(word, &(*p)->right);
      } else if((*p)->right == NULL){
            aux = *p;
            *p = (*p)->left;
            free(aux);
      } else if((*p)->left == NULL){
            aux = *p;
            *p = (*p)->right;
            free(aux);
      } else {
            previous(*p, &(*p)->left);
      }
      return 1;
}

void centralWalk(Pointer p, FILE *outputFile){
      if(p == NULL) return;
      centralWalk(p->left, outputFile);
```

```
        fprintf(outputFile, "%s ", p->content.word);
        printQueue(p->content.lines, outputFile);
        fprintf(outputFile, "\n");
        centralWalk(p->right, outputFile);
}
```

Queue.c

```
#include "../lib/qItem.h"
#include "../lib/qCell.h"
#include "../lib/Queue.h"
#include <stdlib.h>
#include <stdio.h>

void createEmptyQueue(Queue *q){
     q->first       = (qPointer) malloc(sizeof(qCell));
     q->last        = q->first;
     q->first->next = NULL;
}

int isQueueEmpty(const Queue *q){
     return (q->first == q->last);
}

void insertInQueue(int item, Queue *q){
     q->last->next        = (qPointer) malloc(sizeof(qCell));
     q->last              = q->last->next;
     q->last->content.key = item;
     q->last->next        = NULL;
}

int deleteFromQueue(Queue *q, qItem *item){
     qPointer p;

     if(isQueueEmpty(q)){
          return 0;
     }

     p = q->first;
     q->first = q->first->next;
     free(p);
     *item = q->first->content;
     return 1;
}

void printQueue(Queue q, FILE *outputFile){
     qPointer p = q.first;
     if(!isQueueEmpty(&q)){
```

```
            // printf("Lines: ");
            while(p != NULL){
                  p = p->next;
                  if(p == NULL) break;
                  fprintf(outputFile, "%d ", p->content.key);
            }
      }
}
```

TP2.c

```
/* ************************************************
*   Autor: Lucas Caetano Lopes Rodrigues           *
*   Disciplina: AEDS 2                             *
*   Curso: Ciência da Computação                   *
*   Raise your words, not your voice.              *
************************************************** */

/*#include "Queue.c"*/
#include "Library.c"
#include <ctype.h>

const char *TREE_CONSTANT = "TREE";
const char *HASH_CONSTANT = "HASH";

void lowercase(char*, int);
void startTree(FILE*, FILE*);
void _callTreeFunction(char*, char*);
void _callHashFunction(char*, char*);

void main(int argc, char **argv){
     if(argc == 4){
          if(strcmp(argv[3], TREE_CONSTANT) == 0){
               _callTreeFunction(argv[1], argv[2]);
          } else if(strcmp(argv[3], HASH_CONSTANT) == 0){
               _callHashFunction(argv[1], argv[2]);
          } else {
               printf("Parametro incorreto.\n");
          }
     } else {
          printf("Numero incorreto de parametros.\n");
     }
}

void _callTreeFunction(char *inputFilePath, char *outputFilePath){
     FILE *inputFile = fopen(inputFilePath, "r");
     FILE *outputFile = fopen(outputFilePath, "w");

     if(inputFile == NULL){
          //File not found
          printf("Arquivo não encontrado.");
     } else {
```

```
            startTree(inputFile, outputFile);
      }
}

void _callHashFunction(char *inputFilePath, char *outputFilePath){
      printf("Professor, nao tive tempo de fazer implementacao por
HASH. Peco perdao!\n");
}

void startTree(FILE *input, FILE *outputFile){
      char word[1024], testBreakLine1, testBreakLine2;
      int size, countLines = 1;
      Library tree;
      Item i;

      createTree(&tree);

      while((testBreakLine1 = fgetc(input)) != EOF){
            if(testBreakLine1 == 10){
                  if((testBreakLine2 = fgetc(input)) == 10){
                        countLines++;
                  }
                  countLines++;
                  ungetc(testBreakLine2, input);
            }
            ungetc(testBreakLine1, input);

            fscanf(input, "%1023s", word);

            size = strlen(word);

            if(size >= 3){
                  lowercase(word, size);
                  insertIntoTree(word, &tree, countLines);
            }
      }

      centralWalk(tree, outputFile);
}

void lowercase(char *word, int size){
      int i;
```

```
	for(i = 0; i < size; i++){
		word[i] = tolower(word[i]);
	}
}
```