Estruturas de Dados e Algoritmos Período: 2o./2017
Prof.: Fernando W Cruz *Data entrega: 30/11/2017 (postado no Moodle)*

2º. Projeto de pesquisa

1. Título: Comparando o desempenho de estruturas de dados

2. Objetivos do projeto e descrição Geral

Verificadores ortográficos são ferramentas que analisam um texto em linguagem natural e destacam erros de ortografias. De forma simplista, um verificador ortográfico pode ser visto como uma ferramenta que contém um dicionário contendo todas as palavras de um certo idioma. Para encontrar se uma palavra está errada ou não, bastaria verificar se a palavra existe ou não no dicionário. Computacionalmente, o problema de verificar ortografia pode ser custoso, uma vez que existem milhares de palavras em um idioma e, para um texto com 100 palavras, são necessárias até 100 consultas no dicionário.

Neste projeto, é solicitado que os alunos investiguem o uso de estruturas de dados para construção de corretores ortográficos com o objetivo de entender como elas influenciam na eficiência/desempenho do programa.

3. Recursos disponibilizados para execução do projeto

Para alcançar o objetivo proposto, foi disponibilizada, no site da disciplina (Moodle), uma pasta zipada contendo: *(i)* o código fonte, em linguagem C/GCC Linux, de um corretor ortográfico, *(ii)* um dicionário em lingua inglesa (dicioPadrao), e *(iii)* uma pasta contendo documentos (arquivos-texto escritos em lingua inglesa) que podem ser usados para teste do corretor ortográfico.

Em linhas gerais, o corretor ortográfico (*corretorOrtografico.c*) recupera as palavras de um arquivo-texto e as submete ao processo de conferência no dicionário. Ao final, o programa mostra cada erro de ortografia no arquivo-texto junto com estatísticas de execução do processo.

De um modo geral, o programa *corretorOrtografico.c* só aceita dois parâmetros: um arquivo com um dicionário e um arquivo-texto para correção. Após compilado, a forma de chamar o programa é algo como está descrito abaixo:

```
./corretorOrtografico [dicionario] arquivo-texto <enter>
```

Onde:

- *corretorOrtografico* – é o nome do binário compilado à partir do código fonte fornecido
- *dicionario* – deve ser um arquivo contendo uma lista de palavras, em letras minúsculas, uma por linha. Na pasta principal está o arquivo dicioPadrao que contempla o dicionário que deve ser usado nos experimentos do projeto.
- *arquivo-texto* – é um arquivo a ser corrigido ortograficamente. Os arquivos de teste estão disponíveis na pasta textos

Como sugerem os colchetes, o uso do dicionário é opcional (se esse parâmetro for omitido, o corretor usará o arquivo *dicioPadrao* como dicionário). Dentro desse arquivo estão as palavras a serem carregadas na memória.

4. O que deve ser feito?

Embora o corretor ortográfico disponibilizado esteja funcional, ele não tem a implementação das quatro funções mais importantes do código: `carregaDicionario`, `conferePalavra`, `contaPalavrasDic` e `descarregaDicionario`. O objetivo do projeto é construir essas funções considerando as subseções a seguir. Antes de continuar, sugere-se que os alunos analisem o arquivo `corretorOrtografico.c` a fim de compreenderem essas funções (os comentários no código ajudam a compreender cada uma dessas funções).

4.1 Atendendo requisitos mínimos

O requisito mínimo desse projeto é a implementação das 4 funções citadas usando árvores e *hash tables*. Nesse caso, os alunos devem gerar duas versões do corretor ortográfico, considerando:

(i) Uma versão com uso de árvore AVL, e

(ii) Uma versão com uso de *hash table*. Nesse caso os alunos podem escolher uma das funções de *hash* que estão listadas na Seção 7 desse documento. Antes da escolha, sugere-se testar pelo menos três das funções de *hash* e escolher uma delas como solução para o problema.

Após implementar as versões AVL e *hash table*, os alunos devem realizar testes e anotar as estatísticas de execução – apresentação de gráfico de desempenho ou tabela de valores a partir dos testes –, os quais devem ser apresentados no relatório do projeto (especificado mais adiante) .

4.2 Em busca de uma solução ótima (melhor desempenho medido em segundos)

Uma vez tendo valores comparativos entre as soluções AVL e uma das funções *hash table* (dentre as apresentadas na seção 7), os alunos devem implementar uma terceira versão do corretor, procurando por uma solução diferente e com mais *performance* do que as duas versões construídas como requisitos mínimos.

Nesse caso, é permitido usar outras estruturas de dados (*tries*, variantes de árvore balanceada, funções de *hash* que tenham formas distintas de tratamento de colisões, etc.) desde que a estrutura usada seja devidamente explicada. Os passos de pesquisa para se chegar à solução apresentada, incluindo os *sites* visitados e a explicação completa da solução (se possível de forma gráfica) devem ser apresentadas no relatório do projeto.

5. Dicas e restrições

Requisitos para resolver o problema:

- Não alterar a estrutura principal do código *corretorOrtografico.c*, a não ser que haja algum erro de compilação (não percebido pelo professor). Observe como, utilizando a função *getrusage*, é possível capturar o tempo de execução das funções principais do programa.
- Para resolver o escopo das funções citadas (*carregaDicionario*, *conferePalavra*, *contaPalavrasDic* e *descarregaDicionario*), o aluno pode criar funções auxiliares a fim de modularizar a solução apresentada. No entanto, é vedado mexer nas demais funções que chamam as funções citadas.
- A solução construída deve ser feita de modo a NÃO alterar as declarações das funções *carregaDicionario*, *conferePalavra*, *contaPalavrasDic* e *descarregaDicionario*.
- A implementação de *conferePalavra* deve ser *case-insensitive*. Em outras palavras, se a palavra *foo* está no *dicionario*, a função *conferePalavra* deve retornar *true* para qualquer possibilidade de

capitalização. Ou seja, nenhuma variação da palavra (tais como foo, foO, fOo, fOO, fOO, Foo, FoO, FOo e FOO) deve ser considerada incorreta.

- A função *conferePalavra* só deve retornar *true* quando uma palavra realmente se encontra no dicionario e não é permitido colocar palavras comuns *hard-coded* no código. Além disso, as palavras devem ser exatamente como as encontradas no dicionario para que *conferePalavra* retorne *true*. Em outras palavras, mesmo que foo esteja no dicionario, *conferePalavra* deve retornar *false* para um argumento foo's se foo's não está também no dicionário.
- Assumir que, para a função *conferePalavra*, só serão passadas *strings* com caracteres alfabéticos e/ou apóstrofos.
- Assumir que, na correção do projeto, qualquer dicionario passado para o programa será estruturado exatamente como o arquivo *dicioPadrao*. Ou seja, ordenado alfabeticamente de cima para baixo com uma palavra por linha, cada uma das quais terminando com \n.
- Assumir que as palavras manipuladas são menores do que TAM_MAX (uma constante definida em corretorOrtografico.c), que nenhuma palavra vai aparecer mais de uma vez, e que cada palavra irá conter apenas caracteres alfabéticos minúsculos e, possivelmente, apóstrofos.

6. Regras para entrega do trabalho

Este trabalho deve ser feito obedecendo as seguintes regras:

- O trabalho pode ser feito por grupos de até 3 alunos.
- Datas de entrega: *(i)* relatório escrito e códigos – devem ser postados no Moodle até às 08h do dia 30/11/2017, *(ii)* apresentação oral – em 30/11/2017 (trazer relatório impresso p avaliação).
- O grupo deve apresentar as três versões do corretor ortográfico funcionando no laboratório (e entregá-las em mídia digital – arquivo zipado – junto com relatório via Moodle).
- Todo o código deve estar identado e devidamente documentado, inclusive com uma descrição das funções e dos parâmetros usados.
- O grupo deve entregar um relatório, contendo:
  i) Os objetivos do trabalho (incluindo uma declaração sobre quais requisitos de implementação citados foram resolvidos e se foram resolvidos integralmente).
  ii) Um diário de atividades, relatando o que foi resolvido ou estudado em que dia até chegar a versão final do trabalho. Nesse diário, relatar uma descrição resumida sobre os problemas encontrados e as soluções adotadas ao longo dos estudos para atender o que foi especificado nas seções 4.1 e 4.2.
  iii) A explicação das soluções (4.1 e 4.2) devem ser apresentadas de forma clara, se possível com apresentação de gráficos e figuras elucidativas. Sobre a parte que explora melhorias de desempenho (além de AVL e *hash*), o relatório pode conter: *(i)* explicação da estrutura/função utilizada (com desenhos e exemplos, além de citação da fonte), *(ii)* gráficos com os resultados dos índices de colisões, no caso de soluções com *hash*; *(iii)* Tabelas ou gráficos com a comparação entre os tempos e gasto de memória.
  iv) O relatório deve prover uma análise do grau de falhas e limitação das soluções apresentada e as melhorias que poderiam ser providas, se for o caso.
  v) Opinião de cada aluno do grupo sobre o projeto, em especial sobre o uso das estruturas de melhoria de *performance.*

- Os alunos do grupo devem demonstrar conhecimento e aprendizado satisfatório com o projeto. Outros requisitos como cumprimento das datas e trabalho coordenado em grupo também serão consideradas para emissão da nota final.
- As notas serão atribuídas com base *(i)* na qualidade e originalidade da implementação do código, *(ii)* na qualidade do relatório e *(iii)* da apresentação oral (demonstração de conhecimento). Além disso, o grupo que cumprir as regras especificadas e apresentar a solução que encontrar o menor tempo de resposta terá uma pontuação extra.
- Ao final das avaliações, será divulgada um *ranking* da turma, com nomes dos grupos, o tempo encontrado no experimento e as respectivas notas obtidas[1].

7. Funções *hash*

Nessa seção estão apresentadas as funções de *hash* propostas por Partow, Arash (2002) *General Purpose Hash Function Algorithms Library*. Para compreender o funcionamento básico dessas funções sugere-se visitar o *site* http://www.partow.net/programming/hashfunctions/index.html.

```
unsigned int RSHash(const char* str, unsigned int len) {
   unsigned int b    = 378551;
   unsigned int a    = 63689;
   unsigned int hash = 0;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++) {
      hash = hash * a + (*str);
      a    = a * b;
   }
   return hash;
} /* End of RSHash */
```

```
unsigned int PJWHash(const char* str, unsigned int len) {
   const unsigned int BitsInUnsignedInt = (unsigned int)(sizeof(unsigned int) *
8);
   const unsigned int ThreeQuarters     = (unsigned int)((BitsInUnsignedInt  *
3) / 4);
   const unsigned int OneEighth         = (unsigned int)(BitsInUnsignedInt / 8);
   const unsigned int HighBits          = (unsigned int)(0xFFFFFFFF) <<
(BitsInUnsignedInt - OneEighth);
   unsigned int hash              = 0;
   unsigned int test              = 0;
   unsigned int i                 = 0;

   for(i = 0; i < len; str++, i++) {
      hash = (hash << OneEighth) + (*str);

      if((test = hash & HighBits)  != 0) {
         hash = (( hash ^ (test >> ThreeQuarters)) & (~HighBits));
      }
   }
   return hash;
} /* End of PJWhash */
```

[1] O requisito de espaço será avaliado se houver empate com relação ao desempenho das aplicações.

```
unsigned int JSHash(const char* str, unsigned int len) {
   unsigned int hash = 1315423911, i    = 0;

   for(i = 0; i < len; str++, i++) {
      hash ^= ((hash << 5) + (*str) + (hash >> 2));
   }
   return hash;
} /* End of JSHash */
```

```
unsigned int ELFHash(const char* str, unsigned int len) {
   unsigned int hash = 0, x= 0, i= 0;

   for(i = 0; i < len; str++, i++) {
      hash = (hash << 4) + (*str);
      if((x = hash & 0xF0000000L) != 0) {
         hash ^= (x >> 24);
      }
      hash &= ~x;
   }
   return hash;
} /* End of ELFHash */
```

```
unsigned int BKDRHash(const char* str, unsigned int len) {
   unsigned int seed = 131; /* 31 131 1313 13131 131313 etc.. */
   unsigned int hash = 0, i = 0;

   for(i = 0; i < len; str++, i++) {
      hash = (hash * seed) + (*str);
   }
   return hash;
} /* End Of BKDR Hash Function */
```

```
unsigned int SDBMHash(const char* str, unsigned int len) {
   unsigned int hash = 0;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++) {
      hash = (*str) + (hash << 6) + (hash << 16) - hash;
   }
   return hash;
} /* End Of SDBM Hash Function */
```

```
unsigned int DJBHash(const char* str, unsigned int len) {
   unsigned int hash = 5381;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++) {
      hash = ((hash << 5) + hash) + (*str);
   }
   return hash;
} /* End of DJBHash */
```

```
unsigned int DEKHash(const char* str, unsigned int len) {
   unsigned int hash = len;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++) {
      hash = ((hash << 5) ^ (hash >> 27)) ^ (*str);
   }
   return hash;
} /* End Of DEK Hash Function */
```

```
unsigned int BPHash(const char* str, unsigned int len) {
   unsigned int hash = 0;
   unsigned int i    = 0;
   for(i = 0; i < len; str++, i++) {
      hash = hash << 7 ^ (*str);
   }
   return hash;
} /* End Of BP Hash Function */

unsigned int FNVHash(const char* str, unsigned int len) {
   const unsigned int fnv_prime = 0x811C9DC5;
   unsigned int hash      = 0;
   unsigned int i         = 0;

   for(i = 0; i < len; str++, i++) {
      hash *= fnv_prime;
      hash ^= (*str);
   }
   return hash;
} /* End Of FNVHash Function */
```

```
unsigned int APHash(const char* str, unsigned int len) {
   unsigned int hash = 0xAAAAAAAA;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++) {
      hash ^= ((i & 1) == 0) ? (  (hash <<  7) ^ (*str) * (hash >> 3)) :
                               (~((hash << 11) + ((*str) ^ (hash >> 5))));
   }
   return hash;
} /* End Of APHash Function */
```