CI1057 - Algoritmos e Estruturas de Dados III

Segundo Semestre de 2024 - Trabalho Prático

Entrega: 22/novembro/2024, 23:59h

Neste trabalho, você desenvolverá um programa para implementar o **texto preditivo T9**, um modo de entrada de texto desenvolvido originalmente para telefones celulares e ainda usado em teclados numéricos. Cada número de 2 a 9 no teclado representa três ou quatro letras, o número 0 representa um espaço e 1 representa um conjunto de símbolos, como , . ! ? etc. Os números de 2 a 9 representam letras da seguinte forma:

- 2 → ABC
- 3 → DEF
- 4 → GHI
- 5 → JKL
- 6 → MNO
- 7 → PQRS
- 8 → TUV
- 9 → WXYZ

Como várias letras são mapeadas para um único número, as sequências de teclas podem representar várias palavras. Por exemplo, a entrada 2665 representa “book” (livro) e “cool” (legal), entre outras possibilidades.

2			6			6			5			**OU**	2			6			6			5		
a	B	c	m	n	O	m	n	O	j	K	l		a	b	C	m	n	O	m	n	O	j	k	L
	—				—			—		—					—			—			—			—
	b				o			o		k					c			o			o			l

Para traduzir sequências numéricas para palavras, usaremos uma **trie** n-ária. A estrutura de dados clássica da trie para busca de palavras é baseada em letras para armazenar os prefixos. Mas, para este trabalho, usaremos uma trie compactada que tem apenas 10 ramificações possíveis em cada nó (em vez de 26), já que os dígitos de 0 a 9 representam as 26 letras, espaços e símbolos. Por isso, é necessária uma camada extra de complexidade para descobrir a palavra representada por um caminho.

Na verdade, cada nó precisa apenas de 8 filhos possíveis, numerados de 2 a 9, já que os dígitos 0 e 1 não codificam letras. Mas escrever o código pode ser mais fácil se os nós tiverem 10 filhos numerados de 0 a 9. Assim, cada subárvore n corresponde ao dígito n. Fique à vontade para usar qualquer uma das representações para a trie (com 8 ou 10 filhos), dependendo da que parecer mais simples de implementar.

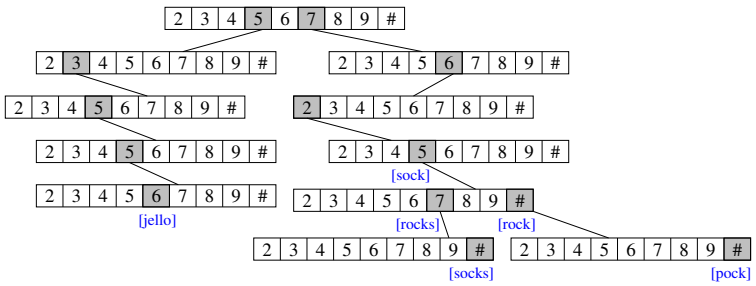
Requisitos técnicos

Implemente um programa chamado `t9`. Você deve usar a linguagem C para desenvolver o trabalho. O comando

```
$ ./t9 dicionario.txt < consulta.txt > saida.txt
```

deve ler um arquivo de dicionário (`dicionario.txt`) que contém uma lista de palavras. Traduza cada palavra do dicionário em sua sequência de teclas numéricas e, em seguida, adicione a sequência de teclas à sua trie, com a palavra no final do caminho correspondente aos dígitos. Se uma palavra com a mesma sequência numérica já existir na trie, adicione a nova palavra à trie como um link para um novo nó com o caracter rotulado como ’#’ ao invés de um dos dígitos 2-9. As palavras vinculadas a um nó pelo caracter ’#’ formam essencialmente uma “lista ligada” de palavras que têm o mesmo código numérico, mas usamos nós da árvore adicionais para vincular às palavras ao invés de definir um tipo separado de nó de lista vinculada somente para essa situação.

Por exemplo, se o programa ler o conjunto de palavras “jello”, “rocks”, “socks”, “sock”, “rock” e “pock” do dicionário e adicioná-lo a uma trie vazia, a trie resultante tem a seguinte forma:



Depois que o programa tiver lido o dicionário e criado a trie, ele deverá ler da entrada padrão sequências numéricas (terminadas ou não com '#') e para cada sequência escrever a palavra correspondente na saída padrão. Seu programa deve usar os números digitados pelo usuário para percorrer a trie que já foi criada, recuperar a palavra e imprimi-la na saída padrão. Se o usuário digitar '#', o programa deverá escrever a próxima palavra na trie que tenha o mesmo valor numérico, e assim por diante. A sequência numérica pode também ser seguida de um ou mais caracteres '#'. Neste caso, o resultado corresponde à palavra que seria encontrada ao digitar o número e os caracteres '#' individuais em linhas de entrada separadas.

Por exemplo, se executarmos o programa usando a trie acima, com o arquivo `consultas.txt` contendo as seguintes linhas:

```
76257
#
53556
#
76257#
76257##
7625
##
9102
013
0
```

A saída do programa deve ser:

```
rocks
socks
jello
palavra nao encontrada
socks
palavra nao encontrada
sock
pock
entrada invalida
entrada invalida
```

Observe que o programa termina quando a linha de entrada contiver o valor 0 (zero).

Certifique-se de que o seu programa trate os casos em que "#" é digitado, mas não há mais palavras, bem como com os casos em que a sequência contém caracteres que não são [2-9,#] ou padrões sem nenhuma palavra associada a eles. Observe que as mensagens na saída para cada um dos casos deve ser exatamente como apresentado no exemplo.

São fornecidos dois arquivos de texto, `miniDicionario.txt` e `dicionario.txt`. Cada um desses arquivos de texto contém uma lista de palavras a serem usadas na construção da trie - a pequena, principalmente para testes, e a grande, para o programa final. Você também pode criar outros dicionários para teste com apenas uma ou duas palavras.

Traduza cada palavra do arquivo em sua sequência de teclas T9 associada e adicione a palavra à trie. No caso de várias palavras com a mesma sequência de teclas (k), deixe que a primeira palavra encontrada no

arquivo de texto seja representada pela sequência de teclas (k), a próxima encontrada seja representada por k#, a próxima por k##, etc. Por exemplo, 2273 pode representar acre, bard, bare, base, cape, card, care ou case. Para não haver ambiguidade, acre seria representado por 2273, bard por 2273#, bare por 2273## e assim por diante.

Além da especificação geral fornecida acima, seu programa deve atender aos seguintes requisitos:

1. Você deve criar um `Makefile` e usar o `make` para compilar o programa. Seu `Makefile` deve recompilar apenas a(s) parte(s) necessária(s) do programa depois que as alterações forem feitas. Seu `Makefile` deve incluir a opção `clean` para remover arquivos gerados anteriormente. Talvez você também queira incluir a opção `teste` para seu próprio uso durante o desenvolvimento.

2. Use `malloc` para alocar dinamicamente os nós, as cadeias de caracteres e quaisquer outros dados que compõem sua trie. Certifique-se de liberar toda a memória no final da execução. O `valgrind` será usado para avaliar se o seu código gerencia bem a memória. Se você precisar criar uma cópia de uma string ou de outros dados de tamanho variável, deverá alocar dinamicamente uma quantidade apropriada de armazenamento usando `malloc` e liberar o armazenamento com `free` quando terminar de usá-lo. A quantidade alocada deve se basear no tamanho real necessário, e não em algum tamanho arbitrário que se presume ser "grande o suficiente". Uma exceção a isso é que você pode assumir que nenhuma palavra excede 50 caracteres de comprimento. Embora não deva criar nós de trie com cadeias de caracteres com o comprimento padrão de 50, você pode usar esse valor ao ler do arquivo de entrada.

3. Use funções da biblioteca C padrão sempre que possível; não reimplemente as operações disponíveis nas bibliotecas padrão. Você deve verificar o status de retorno (código de resultado) de cada função de biblioteca que chamar para ter certeza de que não ocorreu nenhum erro. Em particular, o `malloc` retornará NULL se não conseguir alocar o espaço em memória. Se ocorrer um erro ao abrir ou ler um arquivo, ou devido a uma chamada de função com falha, o programa deverá escrever uma mensagem de erro apropriada no `stderr` e encerrar se não houver mais trabalho a ser feito.

4. Antes de o programa ser encerrado, todos os dados alocados dinamicamente devem ser devidamente liberados (ou seja, liberar tudo o que foi adquirido com `malloc`). Isso deve ser feito explicitamente, sem depender do sistema operacional para fazer a limpeza após o término do programa. Lembre-se de fechar todos os arquivos que tiver aberto.

5. Seu código deve ser compilado e executado sem erros ou avisos quando compilado com `gcc` nas servidoras do DInf. Seu programa deve ser compilado sem erros quando o `make` for executado.

6. Seu programa deve ser encerrado de forma limpa, sem vazamentos de memória ou outros erros de memória relatados quando for executado usando o valgrind.
   Aviso: o `valgrind` torna a execução consideravelmente mais lenta. Levará vários minutos para carregar o arquivo `dicionario.txt` completo e, em seguida, liberar a árvore resultante no valgrind. Sugerimos que você use arquivos de entrada menores durante o desenvolvimento para testar se há problemas de memória com o `valgrind`. Se forem detectados vazamentos de memória, a compilação com a opção `-g` e o uso da opção `--leak-check=full` do `valgrind` gerarão mensagens mais extensas com informações sobre os vazamentos de memória.

Requisitos de qualidade do código

Como em qualquer programa que você escreve, seu código deve ser legível e compreensível para qualquer pessoa que conheça C. Em particular, seu código deve observar os seguintes requisitos:

1. Seu programa deve ser dividido em módulos razoáveis que interajam.

2. O arquivo de cabeçalho (.h) para a trie (e quaisquer outros arquivos de cabeçalho) deve declarar apenas os itens que são compartilhados entre os programas clientes que usam o cabeçalho e o(s) arquivo(s) que o implementam. Não inclua no arquivo de cabeçalho detalhes de implementação que deveriam estar ocultos. Utilize a diretiva do pré-processador #ifndef para que seus arquivos de cabeçalho funcionem corretamente se forem incluídos mais de uma vez em um arquivo de código-fonte, direta ou indiretamente. Você deverá observar atentamente o que está incluído no arquivo de cabeçalho.

3. Certifique-se de incluir protótipos de função apropriados perto do início de cada arquivo de código-fonte para funções definidas nesse arquivo cujas declarações não estejam incluídas em um arquivo de cabeçalho.

4. Comente de forma sensata, mas não excessiva. Você não deve usar comentários para repetir o óbvio ou explicar como a linguagem C funciona. Seu código deve, no entanto, incluir os seguintes comentários mínimos:

   - Toda função deve incluir um comentário que explique o que a função faz (e não como ela faz), incluindo o significado de todos os parâmetros. Não deve ser necessário ler o código da função para determinar como chamá-la ou o que acontece quando ela é chamada.
   - Se sua função pressupõe algo sobre um argumento, certifique-se de deixar isso claro nos comentários.
   - Toda variável significativa deve incluir um comentário que seja suficiente para entender as informações na variável e como elas são armazenadas. Não deve ser necessário ler o código que inicializa ou usa uma variável para entender isso.
   - Todo arquivo de código-fonte deve começar com um comentário que identifique o arquivo, o autor e a finalidade (ou seja, o exercício ou o projeto).
   - Use nomes apropriados para variáveis e funções
   - Não use variáveis globais. Use parâmetros (especialmente ponteiros) de forma adequada.
   - Não faça cálculos desnecessários ou uso excessivo de `malloc` ou `free` - esses recursos são caros. Não faça cópias desnecessárias de estruturas de dados grandes, mas use ponteiros. Lembre que cópias de ints, ponteiros e coisas semelhantes são baratas; mas cópias de arrays e structs grandes são caras.

Dicas de implementação

1. Projete a estrutura da trie e os structs (e campos de struct) de que você precisa. Descubra como adicionar uma única palavra à trie antes de implementar a lógica para processar todas as palavras do dicionário. Descubra como adicionar algumas palavras que tenham códigos numéricos diferentes antes de lidar com palavras que tenham os mesmos códigos. Implemente o código para percorrer a trie e traduzir uma sequência de teclas de entrada na palavra correspondente depois de construir a trie para os 8/10 dígitos, e não antes. Ou seja, desenvolva a TAD trie antes do cliente.

2. Antes de começar a digitar o código no computador, dedique algum tempo para esboçar as estruturas de dados e códigos (especialmente structs de nós de árvore). Certifique-se de que entendeu o que está tentando fazer antes de começar a digitar.

3. Toda vez que adicionar algo novo ao seu código (consulte a dica número 1), teste-o. Agora mesmo! É muito mais fácil encontrar e corrigir problemas se você puder isolar o possível bug em uma pequena seção do código que acabou de adicionar ou alterar.

4. gdb e printf são seus amigos para examinar valores durante a depuração.

5. Você provavelmente achará útil incluir uma função que imprima o conteúdo da árvore em algum formato compreensível. Isso não é obrigatório, mas como você pode ter certeza de que seu código está correto se não puder ver a trie que foi criada para um pequeno conjunto de dados de entrada?

6. Comece com um pequeno arquivo de dados e descubra antecipadamente como deve ser a aparência da trie resultante. Em seguida, verifique se o programa cria, de fato, a estrutura desejada.

7. Para criar a trie, você precisa de uma forma de traduzir os caracteres (letras) das palavras no dicionário para os dígitos correspondentes do teclado. Uma possibilidade é incluir em seu código uma função que recebe um caractere como argumento e retorne o dígito correspondente. Isso pode ser implementado com uma série de instruções if-elseif-else, mas outra maneira de fazer isso é ter um vetor com uma

entrada para cada caractere possível. Na entrada de cada caractere, armazene o código do dígito correspondente. Assim, você pode procurar o código de um caractere sem precisar de uma sequência complicada de instruções if. (Lembre-se de que em C, um caractere pode ser usado como um número inteiro. Isso é particularmente útil exatamente para esse tipo de aplicativo, em que podemos querer usar um valor de caractere como um índice em um vetor.

8. Certifique-se de verificar se há erros, como tentar abrir um arquivo inexistente, para ver se o tratamento de erros está funcionando corretamente.

Sequências de teste:
As sequências abaixo podem ser usadas para testar seu programa utilizando o arquivo `dicionario.txt` fornecido.

- 22737: acres, bards, barer, bares, barfs, baser, bases, caper, capes, cards, carer, cares, cases
- 46637: goner, goods, goofs, homer, homes, honer, hones, hoods, hoofs, inner
- 2273: acre, bard, bare, barf, base, cape, card, care, case
- 729: paw, pax, pay, raw, rax, ray, saw, sax, say
- 76737: popes, pores, poser, poses, roper, ropes, roses, sords, sorer, sores

Detalhes de Entrega
O trabalho pode ser feito em grupos de até 2 pessoas. A entrega deve ser feita por email, com o assunto "CI1057-trabalho-2s2024", para a professora da turma:

- carmemhara@ufpr.br

O prazo de entrega é **22/11/2024, às 23:59h**. Entregas fora do prazo terão desconto na nota por dia de atraso.

- Deve ser enviado um arquivo compactado `tar.gz` com, no mínimo, os seguintes arquivos:
  - `t9.c`: programa cliente
  - `trie.c`: implementação do TAD trie
  - `trie.h`: interface do TAD trie
  - `Makefile`
  - `LEIAME`: detalhes do trabalho que achar interessante, dificuldades que teve na implementação e bugs conhecidos. Descreva a estrutura de dados utilizada para implementar o trabalho.
- o trabalho deverá poder ser compilado no ambiente computacional do DInf
- a compilação deverá ser feita com `make`. O arquivo `Makefile` deve possuir opção `clean` (apaga todos os arquivos objeto .o)
- o trabalho deverá ser executado com a seguinte linha de comando
  `./t9 dicionario.txt < consulta.txt > saida.txt`
- a verificação do resultado será feita com
  `diff saida.txt saida_padrao.txt`
- O arquivo compactado deve possuir o nome como `login.tar.gz`. Em caso de trabalho em grupo, deve possuir o login de todos `login1-login2.tar.gz`. Após descompactar deve **gerar uma pasta** com o nome do compactado (login ou login1-login2) com todos os arquivos necessários, sem subdiretórios.

Qualquer dúvida, enviem um email para carmemhara@ufpr.br.