Relatório do EP4

Juliano Garcia de Oliveira Nº USP: 9277086

14 de Novembro, 2016

1. Especificação do EP

O EP4 consiste em implementar 5 tabelas de símbolos usando diferentes estruturas de dados, sendo elas: Vetor desordenado, vetor ordenado, lista .ligada desordenada, lista ligada ordenada e uma árvore de busca binária. Deve-se criar um arquivo principal que irá receber instruções da linha de comando, ler um arquivo e contar a frequência das palavras do arquivo usando as tabelas de símbolos implementadas, e em seguida imprimir os pares {palavra : frequência} na ordem especificada pela entrada.

2. Algoritmo(s) e estruturas de dados

Os algoritmos implementados são os algoritmos básicos de cada estrutura de dados. Cada tabela de símbolo implementa as operações com os algoritmos corretos para manipular as respectivas estruturas. As operações de cada estrutura são explicadas adiante, e as funções específicas do programa podem ser vistas no código-fonte onde são tratadas com mais detalhe.

Para a leitura dos arquivos, usei um "Buffer" para armazenar os caracteres. O Buffer é uma estrutura de dados que implementamos na matéria de Técnicas de Programação I (MAC216). A implementação de um Buffer é fácil e bastante óbvia, basta ver o código em *buffer.c*. As operações do Buffer são as seguintes:

- ***create***: Cria um Buffer vazio e o retorna;
- ***destroy***: Libera a memória e destrói o Buffer recebido como argumento;
- ***reset***: Aloca novamente o Buffer, apagando o conteúdo antigo;
- ***push_back***: Adiciona o caractere recebido como parâmetro no final do Buffer;
- ***lower***: Trasforma todos os elementos do Buffer para minúsculo;
- ***read_line***: Lê uma linha do arquivo recebido como parâmetro e armazena no Buffer.

A implementação das tabelas de símbolos usa a interface que aprendemos em Técnicas de programação I, só que um pouco modificada. No EP em questão, uma tabela de símbolos contém as seguintes operações, cujo comportamento é igual para todas as diferentes implementações:

- ***create***: Cria uma tabela de símbolos vazia e a retorna;
- ***destroy***: Libera a memória e destrói a tabela de símbolos recebida como argumento;

- ***insert***: Insere uma chave se ela não estiver na tabela de símbolos. Retorna sempre uma estrutura que contém um ponteiro para o valor da chave recebida como argumento, junto com um indicador para determinar se a chave é nova ou não;
- ***apply***: Aplica uma determinada função recebida como *callback* em todos os elementos da tabela de símbolo, ou até que seja interrompida pelo *callback* (com exceção da tabela usando árvore binária, que não permite a interrupção).

2.1 Leitura

A leitura das palavras é feita através do *Buffer*, usando a operação *read_line*. O algoritmo é bastante simples: Enquanto houver caracteres lidos pelo Buffer, separa-se as palavras da linha segundo as especificações do EP, e a palavra é inserida na tabela de símbolos usando a operação *insert*. Caso a palavra seja nova, o valor correspondente da chave inserida recebe **1**, caso contrário este valor é incrementado. Na leitura também é calculado o tamanho da maior palavra lida, para fazer a impressão corretamente. Este tamanho é armazenado na variável *wide*, que é usada nas funções de impressão.

2.2 Tabela de símbolos

As estruturas de dados das tabelas de símbolos mais importantes são a ***stable_s***, ***InsertionResult*** e ***EntryData***.
A ***stable_s*** define a implementação em si da tabela de símbolos, e é diferente pra cada implementação das tabelas de símbolos. Para usar um padrão de desenvolvimento mais correto, as tabelas de símbolo de cada tipo são criadas como um ponteiro para a estrutura *stable_s*, o que não acontece em uma estrutura simples como o *Buffer*, no qual explicitamente declara-se um ponteiro para a estrutura.

O ***InsertionResult*** foi o modo mais simples de permitir com que o usuário mude o valor de determinada chave na tabela de símbolos, e foi apresentado na matéria de Técnicas de Programação I (MAC216). O *InsertionResult* contém a variável *new*, que (após a operação de inserção) tem valor ***1*** caso a chave foi inserida na tabela, ou 0 caso contrário (a chave já estava na tabela). O outro campo do *InsertionResult* contém um ponteiro para o valor da entrada da tabela, isto é, um ponteiro para a *struct EntryData*.

Cada valor da tabela é do tipo ***EntryData***, que é um *union* de alguns tipos básicos para valores. Assim, temos uma implementação genérica da tabela de símbolos, sendo que ela pode ser usada para vários outros propósitos, e não apenas para mapear a chave para um inteiro. Ou seja, é uma implementação *agnóstica* de tabela de símbolos, que pode ser usada no futuro para outras funcionalidades.

Cada implementação da tabela de símbolos possui alguma particularidade, que são explicadas a seguir.

- **Vetor desordenado**

  A implementação do vetor desordenado é a mais simples possível. É simplesmente criado dois vetores, um para as chaves e outro para os valores. Para inserir um elemento, simplesmente procura-se linearmente no vetor de chaves a chave em questão, e se não estiver no vetor, o novo elemento é inserido na próxima posição livre da tabela de símbolo, que é acessada através da varíavel i.

  Busca: $\mathcal{O}(n)$

  Inserção: $\mathcal{O}(1)$

- **Vetor ordenado**

  A implementação do vetor ordenado é mais sofisticada. É igualmente criado dois vetores, um para as chaves e outro para os valores. Para inserir um elemento, ele é procurado usando uma busca binária no vetor de chaves, e se não estiver no vetor, o novo elemento é inserido na posição correta da tabela de símbolos, e todos os outros elementos devem ser deslocados para frente.

Busca: $\mathcal{O}(\log n)$

Inserção:$\mathcal{O}(k)$, onde k é a quantidade de elementos que se deve deslocar. Se a entrada fosse totalmente aleatória, teríamos $\mathcal{O}(\frac{n}{2})$.

- **Lista ligada desordenada**

  A implementação da lista ligada desordenada armazena cada elemento em uma *struct* que representa um par *{chave : valor}* na tabela de símbolos. Como cada um desses pares não estão armazenados em nenhuma ordem específica na lista ligada, então para inserir um elemento é necessário comparar item a item, e se o elemento não estiver na lista ligada, é criada uma nova *struct* e o elemento em questão é adicionado no final da lista ligada.

  Busca: $\mathcal{O}(n)$

  Inserção: $\mathcal{O}(1)$

- **Lista ligada ordenada**

  A implementação da lista ligada ordenada armazena os elementos da tabela de símbolos igualmente a desordenada, com execeção de que os elementos da lista ligada estão ordenados pelas chaves. A diferença é que, na inserção, não é necessário comparar todos os elementos da lista ligada, já que a busca pode ser parada assim que um elemento maior que a chave em questão foi encontrado na lista. Em seguida, o programa verifica se a chave na qual a busca parou é igual a chave a ser inserida. Se for diferente, cria-se uma nova *struct* para armazenar a nova entrada da tabela de símbolos, e ela é encaixada na sua posição correta, de modo que a tabela de símbolos permaneça ordenada.

  Busca: $\mathcal{O}(\frac{n}{2})$ no caso médio, considerando que a entrada é totalmente aleatória. Isso quase nunca acontece se pegarmos um texto 'normal', pois em praticamente nenhum idioma a frequência das palavras segue uma distribuição uniforme (algumas palavras tem frequência bem maior que outras).

  Inserção: $\mathcal{O}(1)$

- **Árvore de busca binária**

  A implementação da árvore de busca binária é parecida com a de listas ligadas. Cada elemento da tabela de símbolos é representado por uma *struct* que contém o par *{ chave : valor}*, além de um ponteiro para a subárvore direita e outro para a subárvore esquerda. A maioria das operações é implementada recursivamente, sendo que a inserção faz uma recursão na estrutura de árvore até inserir o elemento no seu devido lugar, caso ainda não esteja na árvore.

  Busca: $\mathcal{O}(\log n)$ no caso médio.

  Inserção: $\mathcal{O}(\log n)$ no caso médio.

3.Padrão de Desenvolvimento

Este EP é um ótimo exemplo das vantagens de se usar um paradigma de programação de Orientação a Objetos. Ao invés de criar várias funções que são praticamente iguais (cuja única diferença é o tipo da tabela de símbolos usada), o ideal e mais limpo seria criar um classe base para a tabela de símbolo, e então 5 classes filhas que herdam da classe base, aplicando polimorfismo para cada operação da tabela de símbolos. Assim, o código ficaria mais conciso e coerente. Uma alternativa (não tão boa) seria usar um tipo *union* cujos membros contém todas as 5 implementações diferentes das tabelas de símbolos, e passar as operações corretas de acordo com a tabela utilizada como *callbacks*, o que reduziria a quantidade de funções similares mas redurizia bastante a legibilidade do código, pois não ficaria explícita a chamada das funções de cada tipo diferente. Porém, como a especificação e a linguagem C

não permitem nada muito longe do que uma implementação imperativa, muito código que poderia ser reutilizado teve de ser reescrito.

Uma outra escolha do padrão de desenvolvimento foi tentar manter as implementações *agnósticas*, ou seja, são implementações de tabelas de símbolos que servem para armazenar praticamente qualquer informação e funcionar como uma tabela de símbolos em qualquer programa, não como uma tabela de símbolos específica que calcula frequência. Por este motivo, é necessário usar generalizações e considerar o escopo de cada função e operação da tabela de símbolos e da parte principal do EP.

Com relação a otimização, não há muito o que fazer quando as estruturas de dados já são definidas, a não ser no jeito como será a implementação, que obviamente utilizei implementação. Por exemplo, para inserir um elemento na tabela de símbolos LD parece ser mais rápido simplesmente colocar o elemento no início, e ligar o ponteiro do novo elemento ao resto da lista ligada, porém isso é mais devagar (contrariando o senso comum), em partes por causa da frequência das palavras nas línguas que eu testei (português, inglês, e francês), e também pelo fato de que após fazer as buscas, é mais fácil (para o programa) inserir diretamente no último ponteiro. Outras otimizações são bastante simples, como fazer o *quick sort* para ordenar o vetor por ordem de ocorrência, e outras coisas do gênero.

Não separei os arquivos *header* e *.c* em pastas diferentes (*include* e *src*) por causa das orientações do EP4.

4. Resultados

Inicialmente, com arquivos relativamente pequenos o cálculo é quase instantâneo, não sendo possível diferenciar bem qual o tempo de cada implementação.

As entradas dos testes abaixo são:

- Bíblia: *King James Bible*, versão da bíblia em inglês;
- Aleatório: *Strings* geradas aleatoriamente (usando um script em *Python*). O texto é composto de palavras com 3 a 10 letras, no total de 111645 palavras diferentes no arquivo;
- Francês: É uma compilação da Divina Comédia, *Histoire de Jane Grey* e os dois volumes de *Histoire de la République de Venise*, em francês. O arquivo tem 2637732 caracteres, e foram retirados os acentos do texto;
- Ordenado: Arquivo texto com as palavras da *King James Bible*, porém em ordem alfabética.

O resultado dos testes estão dispostos na tabela abaixo (lembrar que o tempo específico varia de acordo com o computador). O teste é feito com a ordenação **alfabética**, sendo que a ordenação por frequência levaria pouco tempo a mais devido ao *quicksort* que é feito nas tabelas **VO**, **LO** e **AB**, que não é executado quando há ordenação alfabética.

Tipo	Bíblia	Aleatório	Francês	Bíblia - Ordenado
VD	5.375s	53.353s	12.613s	43.025s
VO	0.536s	6.608s	0.753s	0.409s
LD	5.496s	1m13.824s	13.572s	48.647s
LO	1m2.879s	2m12.004s	1m47.146s	50.175s
AB	0.402s	0.379s	0.183s	1m30.166s

Os testes apresentados acima são suficiente para discutir os efeitos de cada tabela de símbolo em cada texto.

Primeiramente, vemos que na Bíblia em Inglês a diferença entre as tabelas **LD** e **LO** é bastante marcante. Como na bíblia as palavras com mais frequência são '*the*', '*of*', '*to*','*that*', '*in*', '*he*' e '*shall*', que são palavras que ficam no final da ordem alfabética, logo a lista ligada ordenada é percorrida várias vezes até o final, para encontrar essas palavras. Se a tabela de símbolos fosse só ler arquivos em inglês, poderíamos consertar isso armazenando na ordem reversa (o problema é que '*a*' também é muito frequente, o que não ajudaria muito).

No caso da entrada aleatória, percebe-se que o vetor desordenado demorou bem mais tempo. Com a bíblia, as palavras com maior frequência geralmente são inseridas no início do vetor se o texto estiver bem "uniforme". Mas quando há uma entrada aleatória, isso já não é verdade, então é percorrido várias vezes o vetor inteiro para encontrar certa palavra. O tempo do vetor ordenado também aumentou, já que as palavras possuem uma distribuição mais uniforme (mais deslocamentos). O comportamento da lista ligada ordenada e desordenada foi similar ao da bíblia, porém o tempo de execução para a tabela de símbolos implementada usando a Árvore de busca binária foi menor que a execução com o texto da Bíblia, o que já é de se esperar quando a entrada é aleatória e uniforme, o que permite com que a inserção fique próxima de $\mathcal{O}(\log n)$.

Com o texto em francês o comportamento é parecido com o da Bíblia, e percebemos melhor como o vetor desordenado se saiu melhor em comparação com palavras aleatórias, devido ao fato das palavras com maior frequência serem introduzidas já no início do vetor.

E finalmente na execução do programa com o texto da Bíblia com as palavras ordenadas alfabeticamente, fica claro que é o pior caso para a tabela de símbolos **AB**. Quando o texto já está ordenado, a tabela de símbolos **AB** cria praticamente uma lista ligada comum, porém percorrendo usando recursão e outras operações comuns a árvore. Outro fato interessante é que neste caso a performance das tabelas **LD** e **LO** ficaram praticamente iguais, o que não ocorreu em nenhum dos outros casos, quando a tabela **LD** estava sempre com bastante vantagem em relação a tabela **LO**. Como a lista já está ordenada, é claro que a tabela **LD** se transforma praticamente na tabela **LO**.

4.1 Conclusão

Na média, a melhor implementação da tabela de símbolos é a que usa uma árvore de busca binária. Em segundo lugar temos a implementação usando vetor ordenado, seguido pelo vetor desordenado e por fim a lista ligada desordenada e a ordenada.

É complicado estudar o tempo em função da língua, já que é difícil achar textos exatamente iguais da mesma edição porém em línguas diferentes. No entanto, é fácil de estimar dependendo das palavras mais frequentes dessa língua, que influenciam mais a performance da tabela **LO**. E também há o fato de que se o texto estiver mais ordenado, a performance dele será ruim usando a tabela **AB**, e se estiver aleatório e com poucas palavras repetidas, a performance diminui no **VO**, por causa dos deslocamentos no vetor (O que provamente também ocorre com textos na ordem reversa).

Fontes:

Buffer e tabela de Símbolos, disponível em https://www.ime.usp.br/~fmario/cursos/mac216-15/parte1.html

Word Frequency Counter, disponível em http://www.writewords.org.uk/word_count.asp

Tabelas de Símbolos (TSs), disponível em https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st.html