Relatório do Projeto 2: Verificador de palavras

SCC0223 - Estruturas de Dados I

Dezembro de 2020

Hiago Américo, nUSP: 11218469

Francisco Dias, nUSP: 4402962

Objetivo: Este relatório tem como objetivo explicar os algoritmos e estruturas de dados utilizados na criação de um programa para avaliar trechos utilizando dicionários pré-fornecidos, retornando as palavras do trecho que não estão contidas no dicionário e as palavras mais frequentes do trecho que são palavras-chave.

Descrição do problema

A classificação de conteúdo textual é um dos grandes desafios atuais da área conhecida como processamento de linguagem natural (PLN). Neste trabalho, entendemos como **classificação** a técnica de identificar um conjunto de significados ou conceitos de um texto.

Uma das maneiras de criar algoritmos que sejam capazes de reconhecer o conteúdo de um texto é filtrando as palavras mais pertinentes ao domínio do tema objetivo, e excluindo palavras "comuns" da linguagem, ou seja, que não agreguem significado semântico para a análise. Dessa forma, o assunto pode ser indicado pela alta presença de palavras relevantes que não façam parte de um dicionário pré-estabelecido.

O programa criado tem como principal objetivo verificar trechos de texto a partir de um dicionário e retornar as palavras não encontradas e as n palavras do dicionário mais frequentes no trecho, com suas respectivas frequências.

Conforme a especificação do projeto, serão apresentadas cinco opções ao usuário, sendo elas:

1. Criar dicionário

- Um dicionário é criado, se for possível,
- O usuário digita as palavras a serem adicionadas ao dicionário,
- **Operações**: Inicialização, seguida de número arbitrário de inserções no dicionário.

2. Atualizar dicionário

- O usuário informa o dicionário a ser alterado,
- É informado (1) para uma inserção ou (0) para uma remoção
- A palavra é atualizada no dicionário.
- **Operações**: número arbitrário de inserções ou remoções em um dicionário.

3. Apagar dicionário

- É informado a ID de um dicionário, que é em seguida removido.
- **Operações**: Exclusão de um dicionário.

4. Verificar texto

- É informada a ID de um dicionário e um inteiro inteiro n de palavras mais frequentes a ser retornado,
- O trecho a ser analisado é informado,
- O programa retorna as palavras não contidas no dicionário e as n mais frequentes, em ordem alfabética e de frequência.
- **Operações**:
  - busca dos termos em um dicionário existente,
  - criação de um dicionário de apoio com os termos não encontrados,
  - impressão de ambos na saída padrão.

5. Encerrar aplicação

- Todas as estruturas são excluídas da memória e a aplicação é finalizada.
- **Operações**: Exclusão de todos os dicionários.

Modelagem

Vamos definir um **dicionário** como sendo um conjunto de termos únicos ordenados lexicograficamente. Para escolher qual estrutura de dados será mais adequada para resolver o problema, primeiro pensemos em algumas peculiaridades que estarão envolvidas em tempo de execução.

Temos que o final da entrada em cada opção é somente indicado pelo caractere '#'. Dessa forma, não é sabido pelo programador o tamanho prévio de um dicionário, sendo assim, é mandatória a utilização de TAD's com alocacão dinâmica de memória.

Por não haver a ocorrência de termos em duplicidade, é necessário que uma busca seja feita antes da inserção de um novo termo, que já deve ser inserido na posição correta. Como temos um conjunto ordenado em que gostaríamos que as operações - principalmente a de busca - sejam o mais eficiente possíveis, é natural que recorramos a uma **árvore binária de busca** (ABB).

Porém, dentre as especificidades de dicionários, os termos podem acabar sendo inseridos já em ordem alfabética, o que arruinaria a eficiência da busca em uma ABB, que neste pior caso poderia tornar-se de complexidade $O(n)$. Dessa forma, prezando pela eficiência, é necessário que utilizemos algum processo de **balanceamento** que garanta um menor tempo computacional para nossas primitivas.

Uma **árvore AVL** (assim chamada devido ao nome de seus inventores, Adelson-Velsky and Landis) é uma árvore binária de busca auto-balaceada. A vantagem do balanceamento é possibilitar que a busca seja de complexidade $O(logn)$, porém uma desvantagem é que as operações de inserção e remoção terão o mesmo custo.

As inserções e exclusões possuem custo similar ao da busca pois podem acabar desbalanceando a árvore, que deve então ser rotacionada de forma a assegurar que o balanceamento seja restaurado. Contudo, mesmo assim optamos pela adoção de árvores AVL's como estrutura de dados para armazenar nossos dicionários, por considerar que o ganho de eficiência dado pelas buscas compensa a maior complexidade de inserções e remoções.

Durante a execução, o usuário pode criar até três dicionários. Iremos então criar um vetor de apontadores que armazenará cada um deles para acesso oportuno das outras opções. Após a criação, os termos são lidos da entrada padrão e inseridos em sua respectiva árvore.

Também, criamos um dicionário auxiliar para armazenar os termos inéditos que possam surgir ao comparar um texto com o dicionário. Além disso, é solicitado que sejam impressos na saída padrão os **termos mais frequentes**.

No caso destes, temos um tipo diferente de ordenação: aqui são impressos primeiro os termos que tenham maior ocorrência, sendo que a ordem alfabética é secundária, apenas utilizada entre termos com igual ocorrência. Essa característica nos fez optar pelo uso de outro TAD para armazenar as palavras mais frequentes, a **Fila de Prioridade**.

Como os termos mais frequentes serão recuperados de acordo com sua frequência de ocorrência, e não mais somente em ordem lexicográfica como em nosso dicionário, a adoção desse tipo de estrutura nos favorece na medida em que permite a leitura sequencial dos elementos armazenados no vetor em ordem de prioridade, diminuindo a quantidade de chamados a função de alocação de memória.

Neste projeto, armazenamos nossa fila em um heap, em que seu tamanho é dinamicamente alocado, e tem como tamanho máximo o número de termos no dicionário. Como um heap binário é uma árvore de busca completa, as operações primitivas de busca, inserção e remoção também são realizadas em tempo logarítmico.

Como desvantagem, assim como nas AVL's precisamos garantir que a integridade do heap sejam mantidas após cada operação - o que neste caso envolve trocar elementos de posição para que a prioridade de um nó seja sempre menor ou igual que a de seu ancestral, no caso do *heap máximo*.

Adicionalmente, fizemos duas adaptações no TAD Fila de Prioridade: a primeira delas envolveu incrementar a prioridade de um determinado elemento sempre que ele fosse buscado, de forma a refletir sua frequência de ocorrência num texto, e também demos um tratamento especial no caso de nós com prioridades iguais - neste caso, há uma ordenação lexicofráfica (alfabética).

Assim, o programa, ao receber um texto para verificação, lê uma palavra por vez e procura-a no dicionário escolhido pelo usuário. Caso a encontre, sua frequência é contabiliada dentro da fila de prioridade; caso a palavra não esteja no dicionário, ela é armazenada em um dicionário auxiliar criado com os termos inéditas.

Após todas as palavras serem lidas, o programa então imprime na saída padrão todas as palavras únicas, assim como os n termos mais frequentes, que poderão, no máximo, serem iguais a quantidade de elementos que o dicionário escolhido possui, a depender do valor de n dado pelo usuário.

Implementação

Utilizamos o tipo `ITEM`, que armazena o endereço da string contendo uma *palavra*, e um inteiro que contém a frequência daquele termo. Optamos por gastar um pouco mais de espaço armazenado a frequência mesmo quando ela não é utilizada, no caso dos dicionários, pois em contrapartida a string que armazenará o termo será alocada dinamicamente, e totalmente inicializada e operacionalizada dentro deste módulo.

Tal estratégia proveu um ganho na legibilidade do código, pois acabamos por utilizar sempre os mesmos tipo no chamado das duas funções de maneira consistente - favorecendo também o encapsulamento e a ocultação de informação característicos de um TAD. Abaixo, as principais funções implementadas juntamente com os TAD's Árvore AVL e Fila de Prioridade, presentes nos módulos:

- **heap.h**
  - `criar`: criação da fila;
  - `cheia`: recebe uma fila, verifica se ela está cheia e retorna 1 caso esteja ou 0 caso contrário;
  - `vazia`: recebe uma fila, verifica se ela está vazia e retorna 1 caso esteja ou 0 caso contrário;
  - `inserir`: recebe uma fila e um item para inserir;
  - `remover`: remove e retorna o item com maior prioridade da fila.
- **avl.h**
  - `avl_criar`: criação da AVL e retorno do seu ponteiro;
  - `avl_apagar`: apaga todo o conteudo da lista e desaloca memória;
  - `avl_inserir`: recebe uma AVL e um item para inseri-lo na árvore e retorna 1 caso consiga ou 0 caso contrário;
  - `avl_remover`: recebe uma AVL e um item para removê-lo, retorna 1 caso consiga ou 0 caso contrário;
  - `avl_buscar`: recebe uma AVL e uma chave para buscar um elemento na árvore, retornará caso exista ou retorna ERRO caso não exista;
  - `avl_imprimir`: recebe uma AVL para imprimi-la ordenamente por ordem de frequência dos termos ;
  - `avl_altura`: recebe uma AVL e retorna a altura da mesma;
  - `avl_vazia`: recebe uma AVL e retorna 1 caso esteja vazia, 0 caso contrário ou ERRO caso não tenha sido alocada.

O programa pode ser compilado atráves da diretiva `make all` e executado através do comando `make run`. Para limpar os arquivos gerados pela compilação, utilize o comando `make clean`, conforme os padrões vigentes da GNU Coding Standards.

Conclusão

Este relatório procurou explicar as escolhas e estratégias de implementação por nós utilizadas para a criação de um programa de verificação de texto e contagem de palavras. Utilizamos Árvores AVL para armazenar aos dicionários, e Filas de Prioridades para contabilizar a

ocorrência de cada termo presente no trecho avaliado. Através destes, pudemos modelar de forma que consideramos satisfatória, utilizando algoritmos que trabalham em tempo computacional consideravelmente menor do que seria obtido por estruturas de dados lineares.

Essa escolha também nos favoreceu pelo fato do programa ser capaz avaliar grandes dicionários, pois uma AVL que armazene n termos, tem algura máxima $1,44log_2 n$. Ou seja, temos que uma árvore de altura 30 é capaz de armazenar na ordem de 2^{30} termos, que já é na ordem de um bilhão.

Referências

- AVL tree, https://en.wikipedia.org/w/index.php?title=AVL_tree&oldid=991991252 (acesso em Dec. 08, 2020).
- Priority queue, https://en.wikipedia.org/w/index.php?title=Priority_queue&oldid=992018991 (acesso em Dec. 08, 2020).