Relatório EP 1 [ED2]

- Interface (interface.cpp) :

Interface para utilizarmos o programa, fortemente baseada no *unit_test.cpp*.

Para executar o programa, o usuário deve digitar, além do **./ep**, o nome do arquivo (.txt) que será lido e a sigla implementação escolhida (as mesmas do enunciado do EP). Caso o usuário não digite os 3 termo, ele interrompe a execução e exibe a mensagem de modo de uso (a mesma do unit_test.cpp). A execução também é interrompida se o usuário digita uma sigla de estrutura não válida.

Criamos a tabela de símbolos na implementação desejada e contamos o tempo necessário. Adaptamos a função **testeOperacoes,** presente no *unit_test*. As operações na tabela seguiram a nomenclatura do *unit_test* (*minST, delminST, getST <chave>, rankST <chave>, deleteST <chave>, selectST <int>*).

Quando executa operações que necessitam de outro input do usuário (como o *rankST,* que necessita da chave), o programa aguarda o usuário digitar o outro termo. Para encerrar o programa, basta digitar '*END*'.

Para compilar o EP, basta digitar 'make ep'.

Definimos:

-> o *ranking* de uma palavra que não está na tabela é -1
-> a função *seleciona*, se aplicada a um rank que não está na tabela, devolve ""

- Tabela de Hash (hs.hpp) :

Implementamos uma Tabela de Hashing em um vetor e, para lidar com as colisões, usamos encadeamento.

Temos 2 constantes nesse arquivo, **TAM** (tamanho da tabela) e **PRIMO**, ambas usadas na função de hashing. Adotamos TAM igual a 5000, mas pode ser facilmente alterada para analisar textos maiores.

Cada célula da tabela e da lista é um **cel_HS** que guarda a chave, valor, apontador pro próximo da lista e um bool para indicar se a posição está vazia ou não. A classe da tabela de hash é **HS**.

O construtor recebe o nome do arquivo como argumento e chama a função **leArquivo** que insere as palavras do arquivo na tabela.

A função de hash, itera sobre cada letra da palavra, pegando o hash até então, multiplicando pelo PRIMO, somando com a letra atual e tirando mod TAM do resultado.

As outras funções implementadas são as requisitadas pelo EP.

- Lista ordenada (lo.hpp)

Implementamos uma lista ordenada pelas chaves, ou seja, pelas palavras. Cada elemento da lista é um **cel_LO** que guarda a chave, o valor (número de ocorrências da palavra), o rank dela e ponteiros para o anterior e o próximo da lista. Temos uma função auxiliar, **leArquivo**, que funciona da mesma forma da descrita na Tabela de Hash.

As outras funções são as requisitadas no enunciado, além se uma função **show** que mostra a tabela, ou seja a lista de palavras, o número de ocorrências e o rank delas.

- Lista desordenada (ld.hpp)

Implementamos uma lista desordenada, em que cada elemento é um **cel_LD** (bem semelhante ao **cel_LO** da lista ordenada, apenas sem guardar o ranking. As outras funções são semelhantes às da lista ordenada.

- Árvore Binária (ab.hpp)

Nesse arquivo implementamos uma árvore binária na qual cada nó é um **No** que guarda a chave, o valor e ponteiros para o nó da direita e da esquerda. A classe da árvore binária é **ABB** e guardamos nela um ponteiro para a raiz.

Temos 5 funções auxiliares:

- **put:** chamada pela **insere**, adiciona um elemento recursivamente
- **get:** chamada pela **devolve**, usa recursão para procurar e devolver o valor de uma chave
- **del:** chama pela **remove**, usa recursão para remover um elemento
- **soma:** chamada pela **rank**, usa recursão para devolver o rank de uma chave
- **delABB:** chamada pelo **~ABB()** deleta a árvore
- **getRaiz:** devolve a raiz (usada apenas para mostrar a árvore na função **show**)

As demais funções são as exigidas pelo ep, além da **show** e **leArquivo** descritas anteriormente.

- Vetor desordenado e ordenado (vetor.hpp):

Aqui implementamos um vetor desordenado e um vetor ordenado usando *vector* do STL (apenas para não precisar me preocupar com o tamanho nem dar resize). Cada elemento do vetor é um **cel** que guarda a chave e o valor.

Além das funções exigidas, estão implementadas as funções **show** e **leArquivo** descritas acima.

- Treap (treap.hpp):

Implementamos uma treap, em que cada nó é um **No_TR** que guarda a chave, o valor e a prioridade de cada elemento. Para definir a prioridade do nó usamos a função *rand* e tiramos o módulo de **MAX_TR,** definida como 10169 no início do arquivo, mas que pode ser alterada para fazer simulações maiores.

Implementamos funções auxiliares similares às da **ABB**, além delas temos as funções **rotDir** e **rotEsq** que rotacionam um ramo e devolvem um ponteiro para a nova raiz do ramo.

- Árvore Rubro Negra (rb.hpp) :

Implementamos uma árvore rubro-negra como explicada em aula, cada nó da árvore é um **No_RB**, que guarda a chave, a cor, o valor, além de ponteiros para o pai, além dos filhos esquerdo e direito. A árvore pertence à classe **RB** e possui algumas funções auxiliares:

- **rotDir:** rotaciona um ramo para a direita
- **rotEsq:** rotaciona um ramo para a esquerda
- **delRB:** deleta a árvore
- **put:** adiciona uma palavra na árvore
- **del:** remove uma palavra
- **soma:** auxilia a função rank

As outras funções implementadas são as exigidas no enunciado, além de uma função que mostra a árvore e outra que retorna a raiz.

- Árvore 2-3 (a23.hpp) :

Nesse arquivo, implementamos uma árvore 2-3, em que cada nó é um **No_23**, que guarda duas chaves e dois valores, um bool que indica se é um *doisNo* e outro que indica se é folha (*ehFolha)* ponteiro para o pai, além de ponteiros para os 3 filhos.

A árvore pertence à classe **A23**, implementamos funções auxiliares para ler o arquivo, adicionar uma palavra, devolver uma palavra, encontrar uma palavra com um rank específico, deletar a árvore e remover um elemento.