Relatório EP2 - MAC0323

Anahí Coimbra Maciel | N°USP: 11809127

Maio de 2023

Esse relatório é sobre o EP2 da disciplina Algoritmos e Estruturas de Dados II, lecionada pelo professor Carlos Ferreira no primeiro semestre de 2023. O programa foi escrito em C++. O programa respeita todas as restrições apresentadas na proposta.

1 Como compilar e executar o EP

Para compilar o EP simplesmente rode no diretório do EP o comando:

```
g++ EP2.cpp -o EP2
```

Em seguida, para executá-lo, utilize:

```
./EP2
```

2 Estruturas e funções implementadas

Para esse EP, foram implementadas as estruturas vetor dinâmico ordenado, árvore binária de busca, treap, árvore rubro-negra e árvore 2-3. Para representar as chaves, foi implementado o *struct Key*, e para os valores, o *struct Item*. Cada *Key* contém uma *string* k que é a chave do nó, enquanto cada *Item* contém três inteiros: *freq*, que contém a frequência da chave no texto, *length*, que contém o comprimento da chave, e *vowels*, que contém o número de vogais diferentes na chave.

Para armazenar os dados nas estruturas, foram implementadas as classes *Node, NodeARN, NodeA23* e *NodeVDO*.

Node foi usada para as estruturas árvore binária de busca e Treap. Cada objeto *Node* contém uma *Key*, um *Item*, um inteiro *height* (que representa a altura do nó e é usado apenas na treap) e dois ponteiros, um para o filho esquerdo e um para o filho direito.

NodeARN foi usada para a estrutura árvore rubro-negra. Cada objeto *NodeARN* contém uma *Key*, um *Item*, um valor booleano *color* (que representa a cor do nó, sendo true se vermelha e false se preta) e três ponteiros, um para o filho esquerdo, um para o filho direito e um para o nó "pai".

NodeA23 foi usada para a estrutura árvore 2-3. Cada objeto *NodeA23* contém duas *Key*, dois *Item*, um valor booleano *is2node* (que representa se o nó é 2-nó (true) ou 3-nó(false)), e três ponteiros, um para o filho esquerdo, um para o filho do meio e um para o filho direito(no caso dos 3-nós).

NodeVDO foi usada para a estrutura vetor dinâmico ordenado. Cada objeto *NodeVDO* contém uma *Key* e um *Item*.

Todas as estruturas implementadas contêm as seguintes funções:

```
void add (Key key, Item value);
Item value(Key key);
vector<string> mostFrequent();
vector<string> longestWords();
vector<string> longestNonRepeat();
vector<string> shortestNonRepeatVowels();
```

O função *add(Key key, Item value)* insere na estrutura o nó com chave *key* e valor *value* mantendo as propriedades da estrutura. A função *value(Key key)* retorna o item de chave *key*. As funções *vector<string> mostFrequent(), vector<string> longestWords(), vector<string> longestNonRepeat(), vector<string> shortestNonRepeatVowels()* são utilizadas nos testes e retornam, respectivamente, um vetor de strings com as palavras mais frequentes, mais longas, mais longas sem repetição de letras e mais curtas com maior número de vogais diferentes.

2.1 Vetor dinâmico ordenado

Para essa estrutura, foi utilizado um vetor de *NodeVDO*, estrutura que como já dito contém uma *Key* e um *Item*. Para adicionar elementos à estrutura, usei a função $lower\_bound()$, e para fazer a busca por chave, utilizei uma busca binária.

2.2 Árvore binária de busca, treap e árvore 2-3

Nessas estruturas, segui as implementações apresentadas pelo professor.

2.3 Árvore rubro-negra

Nota-se que na implementação da árvore rubro-negra considerei que a raiz deveria ser sempre preta.

2.4 Funções auxiliares

Foram implementadas as seguintes funções auxiliares:

```
bool hasRepeat(string word);
int difVowels(string word);
```

Respectivamente, as funções:

1. Retorna se a *string word* tem palavras repetidas;
2. Retorna o número de vogais diferentes na palavra.

3 Notas sobre o funcionamento do EP

O programa recebe um arquivo .txt especificado na função *main()*, que pode ser mudado conforme o teste que se deseja realizar. O arquivo segue as diretivas apresentadas na proposta do EP (na primeira linha a estrutura a ser utilizada, na segunda o número de palavras do texto etc). Para contar o número de palavras nos textos, usei o .cpp:

```
string filename = "dickens.txt";
ifstream input(filename);
int cnt=0;
string word;

while(input>>word){
   word.erase(remove_if(word.begin(),word.end(),::ispunct),word.end());
   transform(word.begin(), word.end(), word.begin(), ::tolower);

   if(!word.empty()){
        cnt++;
   }
}
    cout<<cnt;
```

O código acima utiliza as mesmas convenções que utilizei no EP. Dessa forma, o número de palavras que ele indica é o mesmo que o EP. Dado que a contagem correta de palavras é imprescindível para o funcionamento do EP, recomenda-se contar o número de palavras de cada arquivo com esse código, que será disponibilizado como count.cpp.

Neste EP, tudo que não é uma pontuação é considerada uma palavra, inclusive números. Por exemplo, uma sequência de pontos de exclamação (' !!!! ') separada por espaços de palavras não é considerada uma palavra, não sendo inserida na estrutura. Além disso, para simplificar as consultas, retirei toda a pontuação das palavras (por exemplo, 'olá,' se torna 'olá' e 'guarda-roupa' se torna 'guardaroupa' e é interpretada como uma palavra) e converti todas letras maiúsculas para minúsculas. Por fim, é importante notar que caracteres fora da tabela ASCII padrão podem ser mal-interpretados pelo programa.

4 Testes

Preliminarmente, para verificar se as estruturas estavam funcionando, realizei os testes apresentados na proposta do EP para todas as estruturas. Após obter respostas corretas com todas as estruturas, realizei testes com textos aleatórios do tipo lorem ipsum (do site lipsum) de tamanhos variados (o maior com 160 mil palavras) e um texto do site Projeto Gutenberg com cerca de 130 mil palavras. Os resultados (médias calculadas a partir de 10 testes) em segundos para os maiores arquivos estão apresentados abaixo.

dickens.txt				
A23	ARN	ABB	TR	VO
0.5176	0.6096	0.5186	0.5700	1.1822

lorem160k.txt				
A23	ARN	ABB	TR	VO
0.4882	0.5760	0.5240	0.5220	0.5520

Nota-se que os tempos médios de execução com o arquivo dickens.txt foram menores que os tempos médios com o arquivo lorem160k.txt, apesar do tamanho deste ser maior. Provavelmente, isso ocorreu por causa da maior complexidade do arquivo de texto não aleatório.

Além disso, a árvore rubro-negra teve a pior performance com o arquivo lorem160k.txt e a segunda pior com o arquivo dickens.txt, o que foi um fato surpreendente. Por outro lado, a árvore 2-3 obteve a melhor performance em todos os testes, como esperado.

Os arquivos de texto utilizados nos testes e o código count.cpp podem ser encontrados aqui.