Documentação Trabalho Prático 3 da Disciplina Estrutura de Dados - TW1

Samuel Assis Vieira - 2018109736

Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG - Brasil

samuelassis@dcc.ufmg.br

1. Introdução

Mais uma vez Rick Sanchez propoẽum problema para resolvermos computacionalmente. Desta vez Rick gostaria de comprimir suas mensagens de texto de forma a gastar o menos dados com o envio das mesmas. O problema se dividia em duas partes: contar as frequências das palavras e as guardar numa Estrutura de Dados, resolvi essa parte implementando uma tabela Hash. Já a segunda parte era a compressão de dados que devia ser feita usando uma Arvore de Huffman que iria codificar cada palavra com um código binário que correspondia a frequência da palavra no texto. Quanto mais frequente, menor o código.

2. Implementação

2.1. Primeira Parte

A primeira parte consistia em contar e armazenar as palavras do texto, como dito anteriormente implementei uma tabela Hash para resolver o problema, que implementei como um vetor de listas duplamente encadeadas, a qual havia sido implementada no primeiro TP da disciplina, para tratar as colisões. Declarei o tamanho como 128 após testar alguns valores e observar a distribuição não vetor. Criei alguns métodos na lista para busca de elementos e implementei uma função hashing. Uma descrição mais detalhada:

- **Lista::incremental():** Recebe uma palavra e verifica se ela está na lista, caso esteja o contador dessa palavra é incrementado e caso a palavra não esteja na lista. A função de adicionar elemento a lista é chamada com a palavra.
- **Lista::search():** Recebe uma palavra e uma por referencia Celula, a palavra é procurada na lista e os atributos da Celula em que a palavra se encontra são copiados para a Celula que foi passada como parâmetro.
- **Hashing():** A função recebe uma palavra e retorna uma posição para o vetor hash. Para diminuir as colisões multipliquei cada letra da palavra por um peso.

2.2. Segunda Parte

A segunda parte era onde era feito a compressão e codificação dos dados a qual deveria ser feita se usando uma arvore de Huffman. Para montar a arvore implementei um Min Heap para facilitar a extração das menores arvores e juntá-las ate formar a arvore de Huffman. Implementei o Heap a partir de um vetor então primeiro precisei de transferir os dados da Hash para um vetor de nós onde cada posição era uma arvore, então fui extraindo as menores arvores, as juntando e colocando novamente no vetor ate ter apenas duas arvores. Junto então as duas arvores formando assim a arvore de Huffman.

Uma vez a arvore pronta a percorro e armazeno o código de cada palavra no Hash para facilitar o acesso do código. Implementei diversas funções para fazer estas operações:

- **n_words():** A função percorre o hash e retorna a quantidade de palavras na tabela. Usada para criar o vetor de arvores necessário.
- **data_transfer():** A função recebe um vetor de Nos e a tabela Hash, ela vai percorrendo a tabela e transferindo os dados para o vetor de Nos.
- **ExtractMin():** A função recebe o vetor de nos e dois nos para armazenar os mínimos. Ela faz um heap a partir do vetor, extrai sua raiz e a copia para um dos nós como parâmetro, então retira o menor nó do vetor e repete o processo armazenando agora no segundo nó.
- **Swap():** Procedimento que recebe dois Nos e os valores deles um pelo outro.
- **CopyNode()**: A função recebe dois nos e copia os valore de um nó para o outro.
- **MakeHeap():** A função recebe como parâmetro um vetor de nós e percorre metade dele chamando Heapify(), que garante as propriedades do Min Heap.
- **Heapify():** Função que mantêm as propriedades do Min Heap toda vez que o nó pai for maior que o nó filho é feita a troca e Heapify é chamada novamente repetindo o processo recursivamente até que a raiz seja o menor nó do vetor.
- **MergeNode():** A função recebe três nós como parâmetro, ela soma os valores dos dois primeiros nós e armazena no terceiro. Fazendo ainda o terceiro nó apontar para os outros dois. Criando assim uma arvore, seguindo algumas exigências dadas na especificação.
- **EncodeWords():** A função recebe uma árvore, uma string e uma tabela hash. Ela fara a codificação das palavras, a arvore recebida tem as palavras de interesse nas folhas, a função então percorre recursivamente toda a arvore em caminhamento Pré-Ordem, adicionando ao código ‘1’ a cada vez que anda para a esquerda e ‘0’ para caminhamento a direita. Isso é realizado ate se chegar a uma folha da arvore que tem seu respectivo código associado a ela pela tabela hash que foi passada como parâmetro.

2.3. Main

A função main organiza todos os procedimentos e funções explicitados acima. Primeiro recebe o texto como entrada do usuário, e insere cada palavra na Tabela Hash. Em seguida Os dados da tabela são transferidos para um vetor de arvores que vai sendo reduzido ate sobrar duas arvores que são combinadas formando assim a arvore de Huffman. A arvore é então percorrida e os códigos armazenados. Em seguida o programa lê do usuário uma entrada para retornar a contagem de uma palavra ou o código de uma palavra.

3. Instruções de compilação e execução

Foi criado um Makefile para facilitar a compilação do programa. Todo o código foi feito em C++ e é compilado na versão 11 da linguagem. O código foi desenvolvido e testado numa maquina com processador Intel Core 2 Duo, 4GB RAM e SO Linux Mint 19.2 Cinnamon. O código também foi testado nos Laboratórios do DCC com sistema Ubuntu.

Para realizar a execução do make file e compilação do programa basta baixar o diretório *samuel_assis* acessar o mesmo pelo terminal e então digitar o comando ‘make’, o programa será compilado e o executável terá o nome ‘tp3’, para executá-lo basta digitar ‘./tp3’.

4. Analise Complexidade

4.1. Complexidade de Espaço

- **Main():** Declara um vetor de listas de tamanho *t*, um *inteiro*, um *char* e um struct Cell. Além disso declara dinamicamente um vetor de Node de tamanho *d* e dois s*tructs* Node em cada iteração do laço. Sendo assim t(2+a+b) + 3 + (2+a+b) + *d*(a + 4) + d-2(a+4) bytes. Sendo *a* tamanho da string que armazena a palavra e *b* o tamanho da string que armazena o código.

Todas as outras funções tem custos constantes de memória.

4.2. Complexidade de Tempo

- **Main():** A função tem 3 laços, o todos tem custos lineares. Para uma entrada de *n* palavras das quais *p* palavras são independentes o custo do primeiro no pior caso é n. E do segundo é p-2. Ou seja, O(n).
- **Heapify():** A função trabalha com metade do vetor, no melhor caso o menor já esta na raiz então é O(1). Já no pior caso por ser um arvore binaria é O(log n).
- **data_transfer():** A função trabalha com 2 laços um dentro do outro. O melhor caso é um vetor de tamanho 1 sem elementos na lista, executando assim somente o *for* de fora uma vez, logo O(n). O pior caso é um vetor de n posições e todas as listas das posições com *p* elementos, com complexidade n*p. Aproximadamente $O(n^2)$.

Podemos concluir então que o programa tem complexidade de aproximadamente $O(n^2)$.

5. Conclusão

Esse Trabalho foi sem duvidas o mais desafiador da disciplina, achei a implementação complexa, trabalhosa e o prazo insuficiente. Tive muita dificuldade em abstrair a segunda parte, a montagem da arvore. Tive que aprender de forma consistente a implementar um heap e me custou um valioso tempo não sobrando muito tempo pra me dedicar montagem da arvore. Minha arvore acabou não saindo da forma como deveria e eu não consegui achar onde tinha errado em tempo hábil (creio ser na comparação de palavras na hora de juntar os nós ou no Heap). Sendo assim a parte de retornar o código acabou por não funcionar pois a arvore estava montada errada e não consegui colocar em pratica da foram correta minha ideia de concatenar strings para ser o código. Mas já não tinha mais tempo pra consertar os problemas do código. Não consegui concluir o trabalho mas ainda sim aprendi muitas coisas, como manipulação de arvores, manipulação de ponteiros, implementação de um Heap e mais sobre alocação de memória.

6. Bibliografia

Slides da Disciplina (Arvores; Hash)

http://www.cplusplus.com/

https://www.geeksforgeeks.org/binary-heap/

https://www.geeksforgeeks.org/building-heap-from-array/