MAC 121 – Algoritmos e Estruturas de Dados I

Segundo semestre de 2016

Tabelas de símbolos – Entrega: 7 de novembro de 2016

Nome: Eduardo Hashimoto nº USP 6514136

Introdução

Para execução do EP foram criados três arquivos separados do main, um para cada estrutura de dados. A estrutura data visa armazenar a palavra e o contador em um único item.

Textos de Teste

Para o teste de eficiência foram usados 4 arquivos diferentes a saber: t1.txt com 182299 characteres t2.txt com 365284 characteres t3.txt com 731834 characteres t4.txt com 1463592 characteres

Vetores

Os vetores foram feitos com uma estrutura auxiliar que além do próprio vetor (ponteiro para data) continha o espaço máximo destinado na memória e a última posição ocupada.

As duas principais funções são:

```
vetor *insereOrdenadoV(vetor *p, char *c){

    int indice;
    char *c2;

    c2 = malloc(1024);
    strcpy(c2, c);

    indice = buscaBinaria(p, c2, 0, p->ultimo - 1);

    if (p->ultimo == indice || strcmp(p->data[indice].palavra, c2) != 0){
        rearranjaV(p, indice);
        p->data[indice].palavra = c2;
        p->data[indice].contador = 1;
        p->ultimo++;
    }
    else {
        p->data[indice].contador++;
    }
    return p;
}
```

Que busca a palavra a ser inserida no vetor e caso a encontre devolve seu índice e aumenta seu contator. Se não a encontrar, o vetor é rearranjado de modo a dar espaço a nova palavra a ser inserida. Assim, *insereOrdenadoV* tem complexidade linearno pior caso, devido a função *rearranjaV*, que copia todo o vetor para um outro vetor auxiliar. No melhor caso tem complexidade logarítmica, uma vez que a função *buscaBinaria* é a única a ser executada.

```
vetor *insereDesordenadoV(vetor *p, char *c){

    int i;
    char *c2;

    c2 = malloc(1024);
    strcpy(c2, c);

    for (i = 0; i < p->ultimo; i++){
        if (strcmp(p->data[i].palavra, c2) == 0){
            p->data[i].contador++;
            return p;
        }
    }

    if (p->ultimo == p->max - 1)
        rearranjaV(p, i);
    p->data[i].palavra = c2;
    p->data[i].contador = 1;
    p->ultimo++;

    return p;
}
```

Nesse caso, a função performa uma busca linear pela palavra, e caso não a encontre insere como a última palavra do vetor. Assim, *InsereDesordenadoV* tem complexidade sub-linear ou linear, respectivamente.

Listas Ligadas

As listas ligadas foram feitas com estrutura idêntica a estrutura aprendida em sala de aula (a menos que foi considerado o dado ou informação como sendo outra estrutura com dois dados).

Analogamente ao vetor ordenado as duas principais funções são:

```
apontador insereOrdenadoL(apontador inicio, char *c){

    apontador p, ant, novo;

    char* c2 = malloc(1024);
    strcpy(c2, c);

    novo = malloc(sizeof(celula));
    novo->data.palavra = c2;

    p = inicio;
    ant = NULL;

    while (p != NULL && strcmp(p->data.palavra, c2) < 0){
        ant = p;
        p = p->prox;
    }

    if (p != NULL && strcmp(p->data.palavra, c2) == 0){
        free(novo);
        p->data.contador++;
        return inicio;
    }

    if (ant == NULL){
        novo->prox = inicio;
        inicio = novo;
        novo->data.contador = 1;
    }

    else {
        novo->prox = ant->prox;
        ant->prox = novo;
        novo->data.contador = 1;
    }

    return inicio;
}
```

Observa-se que, diferentemente, do vetor ordenado, a lista ligada ordenada tem que performar uma busca linear em seus elementos para verificar se determinada palavra já se encontra lá. Caso a encontre apenas incrementa seu contador, senão a inserção é imediata (pois um novo nó é criado e os ponteiros remanejados). Assim, *insereOrdenadoL* tem complexidade sub-linear, caso a palavra esteja na estrutura e linear caso não esteja.

Listas Ligadas

Diferentemente das outras estruturas, a árvore de busca binária sempre insere a palavra de modo ordenado. A função *insereAB* é tipicamente recursiva, e divide a estrutura pela metade até encontrar (ou fazer toda procura) a palavra.

```
apontadorA insereAB(apontadorA raiz, char *c){

    char *c2 = malloc(1024);
    strcpy(c2, c);

    if (raiz == NULL){
        raiz = novaCelula(c2);
        return raiz;
    }

    else if (strcmp(raiz->data.palavra, c2) > 0)
        raiz->esq = insereAB(raiz->esq, c2);
    else if (strcmp(raiz->data.palavra, c2) < 0)
        raiz->dir = insereAB(raiz->dir, c2);
    else
        raiz->data.contador++;

    return raiz;
}
```

Assim, essa função é logaritimica.

Ordenação

Em todos os casos concluído os modos de inserção na estrutura, procede-se ao modo para organizar essa estrutura por ocorrência ou ordem alfabética dos dados. Para o vetor basta usar a função *qsort* da biblioteca . Nas demais estruturas, no entanto, é preciso copiar a estrutura para um vetor auxiliar, organizar com *qsort* e inserir novamente na estrutura. Assim, a ordenação tem impacto menor no vetor que nas outras estruturas.

Análise dos tempos e conclusão

```
        t1.txt      t2.txt      t3.txt      t4.txt

VO A    0.13        0.23        0.61        0.82
VO O    0.15        0.24        0.62        0.81

VD A    0.76        1.60        6.84        13.98
VD O    0.76        1.65        6.07        13.96

LO A    1.17        3.69        23.34       68.83
LO O    1.10        3.60        24.00       73.04

LD A    1.75        5.69        28.87       88.74
LD O    1.71        5.67        28.78       92.21

AB A    0.23        0.51        1.05        2.38
AB O    0.25        0.48        1.07        2.36
```

Como a leitura das palavras é linear, temos que a complexidade da inserção em cada estrutura deve ser multiplicada por n.

Assim, observa-se que o vetor ordenado e a arvore binaria devido ao seus modos de inserção tem complexidade em algo como Cnlog(n). As demais estruturas tem complexidade em algo como Cnn.

Em relação a estruturação do código, observa-se que este está pouco organizado requerendo um refatoramente urgente, o qual não foi feito por falta de tempo e inteligência. Ademais, infelizmente, a estruturação do código parece não ser abordada nas salas de aula, deixando o aluno "à deriva" para escrever um código limpo. Feita essa ressalva observa-se que o código apresenta muitas funções similares nas estruturas e que, talvez, pudessem ser condensadas. No corpo da função *main.c* o mesmo código é repetido diversas vezes dependendo da escolha do usuário.

Observações sobre trechos dos códigos

1) O trecho:

```
    char* c2 = malloc(1024);
    strcpy(c2, c);
```

repetido em todas as funções de inserção, visa criar uma cópia do string a ser lido no texto. Caso contrário, toda vez que a função fosse chamada o mesmo string seria inserido na estrutura, fornecendo uma tabela de símbolos com um único símbolo.

2)A função *qsort* da foi usada para organizar a estrutura, tanto alfabeticamente (*compareA*) quanto por número de ocorrências (*compareO*), observando que se ela já se encontra ordenada alfabeticamente então não é preciso performar o sort.