Universidade de Brasília - Estrutura de Dados

Professor: Cristhian Ivan Riaño Jaimes

Alunos : Maria Claudia Campos Martins ( 17/0109968 )

Brando Wilha Galdino Da Silva (17/0161579)

**Trabalho Prático 01**

**01-**

A eficiência de um algoritmo vai além de sua corretude, por esse motivo, mesmo que haja várias estruturas de dados que garantem sua funcionalidade é adequado escolher uma estrutura que minimize, na medida do possível, os custos de tempo e espaço durante a execução do algoritmo. Uma escolha inadequada pode ocasionar num gasto de tempo (e esforço) da máquina que poderia ser facilmente contornado se utilizar outra estrutura, como, por exemplo, se quiser garantir uma busca de um elemento numa estrutura, procura-se meios para encontrar o mais rápido possível o endereço desse elemento na memória; por isso, nesses casos é melhor utilizar um vetor – estrutura com tamanho fixo durante a execução do algoritmo. Outro problema que uma escolha inadequada causa é o desperdício ou falta de memória, por exemplo, em um vetor, pode-se retirar elementos sem desalocar a memória reservada para tal, ou seja, há espaços inutilizados na memória que poderiam servir para outros usos de processamentos.

**02-**

(1) f(n) = n – 100 ; g(n) = n – 200 ;  
  
 n – 100 <= c(n – 200) ; para n >= 300 e c = 2 ; logo f(n) = O(g(n))

n – 100 >= c(n – 200) ; para n >= 0 e c = 1; logo f(n) = Ω(g(n))

e como f(n) = O(g(n)) e f(n) = Ω(g(n)) ; conclui-se que f(n) = Θ(g(n))

(2) f(n) = n¹/² ; g(n) = n²/³ ;

n¹/² < cn²/³ ; para n >= 2 e c >= 1 ; logo f(n) = O(g(n))

não existe c para que n¹/² > c(n²/³) ; f(n) ≠ Ω(g(n))

(3) f(n) = 100n + logn ; g(n) = n + log(2n)

100n + logn <= c(n + log(2n)) ; para n >= 1000 e c = 1000 ; f(n) = O(g(n))

100n + logn > c(n + log(2n)) ; para n >= 1 e c = 1 ; f(n) = Ω(g(n))

e como f(n) = O(g(n)) e f(n) = Ω(g(n)) ; conclui-se que f(n) = Θ(g(n))

(4) f(n) = nlog(n) ; g(n) = 10nlog(10n) ;

nlog(n) < c10nlog(10n) ; para n >= 1 e c = 1 ; f(n) = O(g(n))

nlog(n) > c10nlog(10n) ; não é verdade para n > 0 e c > 1 ; f(n) ≠ Ω(g(n))

(5) f(n) = log(2n) ; g(n) = log(3n) ;

log(2n) < clog(3n) ; para n >= 1 e c = 1 ; f(n) = O(g(n))

log(2n) > cnlog(3n) ; não é verdade para n > 0 e c > 1 ; f(n) ≠ Ω(g(n))

(6) f(n) = 10log(n) ; g(n) = log(n²) = 2log(n) ;

10log(n) < c2log(n) ; para n >= 3 e c = 6 ; f(n) = O(g(n))

10log(n) > c2log(n) ; para n >= 1 e c = 1 ; f(n) = Ω(g(n))

e como f(n) = O(g(n)) e f(n) = Ω(g(n)) ; conclui-se que f(n) = Θ(g(n))

(7) f(n) = (n²)/log(n) ; g(n) = nlog(n)² ;

(n²)/log(n) < cnlog(n)² ; para n >= 1 e c = 1 ; f(n) = O(g(n))

(n²)/log(n) > cnlog(n)² ; não é verdade para n > 0 e c > 1 ; f(n) ≠ Ω(g(n))

(8) f(n) = √n ; g(n) = log(n)³ ;

n¹/² < cnlog(n)³ ; para n >= 3 e c = 1 ; f(n) = O(g(n))

n¹/² > cnlog(n)³ ; não é verdade para n > 0 e c > 1 ; f(n) ≠ Ω(g(n))

(9) f(n) = √n ; g(n) = 5^(log(n)) ;

n¹/² <= c5^(log(n)) ; para n >= 1 e c = 1 ; f(n) = O(g(n))

n¹/² > c5^(log(n)) ; não é verdade para n > 0 e c > 1 ; f(n) ≠ Ω(g(n))

(10) f(n) = n(2^n) ; g(n) = 3^n ;

n(2^n) < c(3^n) ; para n >= 3 e c = 1 ; f(n) = O(g(n))

n(2^n) > c(3^n) ; não é verdade para n > 0 e c > 1 ; f(n) ≠ Ω(g(n))

**03-**

Para um vetor de tamanho n, e um elemento na posição n desse vetor, fazer n – 1 comparações com o vetor de tamanho n – 1 que já está ordenado, para um elemento na posição n – 1 fazer n – 2 comparações com o vetor de tamanho n – 2 já ordenado...

T(n) = T(n – 1) + n -1

T(n – 1) = T(n – 2) + n – 2

T(n – 2) = T(n – 3) + n – 3…

T(1) = 1;

∑ni=0 (n – i) = n² – (n(n + 1))/2 = n² – (n²)/2 – n/2 = n²/2 – n/2

**04 -**

Para implementar uma pilha usando duas filas: Nomeie as filas como,por exemplo, fila01 e fila02. O próximo passo é organizar a inserção de elementos na pilha, para tal, a cada elemento adicionado é necessário transferir todos os elementos de fila01 para fila02. Em seguida, adiciona-se o novo elemento a fila01 para,então, retornar os elementos de fila02 para fila01. A retirada de um elemento é feita apenas retirando um elemento de fila01, caracterizando desse modo uma pilha com o comportamento de ‘last in, first out’.

O tempo de execução desse algoritmo é da ordem de O(n). Na retirada, como apenas um elemento é retirado de cada vez, a ordem é de O(1).

**05 -**

Para implementar uma filha usando duas pilhas: Nomeie as pilhas, por exemplo, pilha01 e pilha 02. Depois, mantenha os elementos que são adicionados na pilha01. Quando retirar algum elemento, basta transferir todos os elementos de pilha 01 para pilha 02 para,então, retirar o elemento desejado da pilha02. Dessa forma, vê-se o comportamento de uma fila de “first in,first out”.

Ao acrescentar um elemento, o tempo de execução tem a ordem de O(1), já para retirar, é O(n).

**06,07,08 -**

*[Foram feitas em linguagem C. São os arquivos ‘.c’.]*

**10,11-**

Como ambas as questões envolvem o mesmo tipo de lista, o mesmo algoritmo foi utilizado para resolver seus problemas.

O tipo ‘t\_lista’, utilizado no algoritmo que se segue, é dado por:

typedef struct lista{

t\_elemento\* primeiro;

t\_elemento\* ultimo;

}

E o tipo ‘t\_elemento’, por:

typedef struct elemento{

int dado;

struct elemento\* proximo;

}t\_elemento;

Desse modo, segue-se o algoritmo:

t\_elemento\* reverse (t\_lista\*lista){

t\_elemento\* atual = lista->primeiro;

t\_elemento \*t, \*r = NULL;

while(atual != NULL){

t = atual->proximo;

atual->proximo = r;

r = atual;

atual = t;

atual = atual->proximo;

}

lista->primeiro = r;

return lista->primeiro;

}

**12-**

Os procedimentos feitos nessa questão foram estruturados em linguagem C.

Para concatenar duas listas encadeadas simples, com seus elementos estruturados a seguir:

typedef struct elemento{

int dado;

struct elemento\* proximo;

}t\_elemento;

E uma estrutura auxiliar para manipulação:

typedef struct lista{

t\_elemento\* primeiro;

t\_elemento\* ultimo;

}t\_lista;

tem-se o seguinte algoritmo:

void concatena\_simples(t\_lista\* l\_1,t\_lista\* l\_2){

t\_elemento\* atual = l\_1->primeiro;

int i;

if(atual != NULL){

for(i=0;atual->proximo != NULL;i++){

atual = atual->proximo;

}

atual->proximo = l\_2->primeiro;

}

}

Para concatenar duas listas duplamente encadeadas, considera-se que seus elementos tem a seguinte forma:

typedef struct elemento{

int dado;

struct elemento\* anterior;

struct elemento\* proximo;

}t\_elemento;

Com estrutura auxiliar para a manipulação da lista idêntica a utilizada no procedimento anterior. Assim, tem-se o algoritmo de concatenação de listas duplamente encadeadas a seguir:

void concatena\_dupla(t\_lista\* l\_1,t\_lista\* l\_2){

t\_elemento\* atual = l\_1->primeiro;

int i;

if(atual != NULL){

for(i=0;atual->proximo != NULL;i++){

atual = atual->proximo;

}

atual->proximo = l\_2->primeiro;

l\_2->primeiro->anterior = atual;

}

}

**13 -**

Para o algoritmo de insertion sort, a escolha mais adequada de uma estrutura de dado é um vetor, já que o algoritmo envolve dados homogêneos e precisa de um tamanho fixo para que funcione. Outro exemplo, é um algoritmo de busca em que a melhor estrutura é aquela em que é possível saber o endereço do elemento no qual se busca, como, por exemplo, uma lista sequencial ou um vetor. Por fim, um terceiro exemplo, é um dos algoritmos de inserção, em que é melhor trabalhar com uma lista encadeada pois permite maior flexibilidade com os elementos além de evitar desperdício de memória.

**14 -**

A lista encadeada é uma estrutura alocada dinamicamente, cujos elementos encontram-se dispersos na memória. Dessa forma, é possível efetuar operações de inserção,remoção e busca de um elemento, além de ser possível manipular seu conteúdo de forma flexível - ou seja, pode-se inserir um elemento cujo espaço não estava reservado antes ou retirá-lo e liberar a memória alocada para tal. Já o vetor é uma estrutura sequencial cuja alocação implica que seus elementos estejam sequencialmente na memória. São possíveis as mesmas operações que são feitas com a lista encadeada porém ao inserir e remover um elemento, a memória alocada inicialmente continua sendo reservada para o vetor.