Universidade Federal do Paraná

Setor de Ciências Exatas

Departamento de Informática

ALGORITMOS E ESTRUTURAS DE DADOS II (CI056)

Segundo Trabalho Prático

Rudolf Copi Eckelberg

Professor - David Menotti

Curitiba
10 de maio de 2016

Sumário

Lista de Figuras

Lista de Programas

Lista de Tabelas

1 Introdução

Este trabalho documenta a geração de um TAD para administração de um espaço de memória, simulando um gerenciador de memória real. Serão aplicados conceitos de listas encadeadas e cursores.

1.1 Especificação do problema

O desafio é criar um TAD TMemoria capaz de armazenar dados simulando a memória real de um computador. Ele deve ser implementado a partir de uma lista duplamente encadeada por cursores. Uma lista auxiliar dentro do mesmo vetor deve organizar os espaços livres de memória.

A memória ocupada deve ser ordenada pelos cursores (mas não necessariamente dentro do vetor, por otimização). A memória livre não seguirá nenhuma ordem em particular, mas será organizada em uma lista.

As funções deste TAD devem incluir:

- Criação de uma memória interna vazia;
- Obtenção do número de células ocupadas;
- Inserção de novo dado, mantendo os dados ordenados;
- Remoção do primeiro ítem;
- Remoçào do último ítem;
- Impressão do conteúdo do TAD.

2 Projeto e implementação dos algoritmos

Como há poucos erros possíveis para cada função desse TAD, não foi necessária a definição de um padrão de retorno rebuscado. Por simplificação, funções que podem falhar retornam 1 em caso de sucesso e 0 (número zero) em caso de falha, para simplificar a leitura das invocações.

A memória livre se comporta, nessa aplicação, como uma pilha. Cada vez que um elemento de memória é liberado (removido), é adicionado ao início da lista de memória livre como primeira célula desocupada, que será a primeira a ser utilizada quando um novo ítem for inserido.

Por conta desse comportamento, a memória livre, após utilização prolongada, não estará ordenada de acordo com nenhum critério em particular, mas pode ser utilizada normalmente em sua totalidade.

A ordem desta pilha utilizará os cursores para se estabelecer, e os cursores "ant"serão tratados como refugo de memória.

2.1 Tipo TMemoria

O tipo TMemoria é declarado em um arquivo de cabeçalho de mesmo nome, TMemoria.h.

A parte relevante às aplicações externas (ou seja, o dado propriamente dito) é definido em uma estrutura chamada TItem.

```c
#ifndef _TMEMORIA_
#define _TMEMORIA_

#define MAX_SIZE 10

typedef struct{
    int chave;
    int data;
} TItem;

typedef struct{
    TItem item;
    int prox, ant;
} TCelula;

typedef struct{
    TCelula items[MAX_SIZE];
    int priCelulaDisp;
    int primeiro, ultimo;
    int numOcupadas;
} TMemoria;

void zeraMemoria(TMemoria *);
int memoriaVazia(TMemoria *);
int numeroOcupadas(TMemoria *);
int encontraAnterior(TMemoria *, int);
int insereItem(TMemoria *, TItem *);
int removePrimeiro(TMemoria *, TItem *);
int removeUltimo(TMemoria *, TItem *);
void imprimeSequencia(TMemoria *);
void rawPrint(TMemoria *);

#endif
```

Programa 1: TMemoria.h

Como não há especificação do tipo de dado a ser armazenado, o tipo escolhido, por simplicidade, foi o de um inteiro acompanhado de chave de ordenação. Para alterar esse comportamento, ou até mesmo para definir tipos mais complexos para TItem, o arquivo de cabeçalho pode ser estendido sem que as funções do TAD se quebrem.

2.1.1 Código fonte: TMemoria.c

As funções do TAD desenvolvido estão no arquivo TMemoria.c. A discussão das funções isoladamente se dá na seção seguinte: **Funções do tipo TMemoria**

```c
/* Consideracoes so far: talvez remover a cabeca seja melhor */
#include<stdio.h>
#include "TMemoria.h"

void zeraMemoria(TMemoria* pMemoria){
    int i;

    // Inicializando parametros da lista
    pMemoria->priCelulaDisp = 0;
    pMemoria->primeiro = -1;
```

```
    pMemoria->ultimo = -1;
    pMemoria->numOcupadas = 0;

    // Inicializando indices vazios
    for (i = 0; i < MAX_SIZE-1; i++) {
        pMemoria->items[i].prox = i+1;
        pMemoria->items[i].ant = i-1;
    }
    pMemoria->items[MAX_SIZE-1].prox = -1;
    pMemoria->items[MAX_SIZE-1].ant = MAX_SIZE-2;
    //pMemoria->ultCelulaDisp = MAX_SIZE-1;
}

int memoriaVazia(TMemoria* pMemoria){
    if(!pMemoria->numOcupadas) return 1;
    return 0;
}

int numeroOcupadas(TMemoria* pMemoria){
    return pMemoria->numOcupadas;
}

/* encontraLugar e uma funcao auxiliar para encontrar o lugar
   de um Item na sequencia de chaves. O indice retornado corresponde
   ao item depois do qual a sequencia deve ser deslocada para a direita
   . */
int encontraAnterior(TMemoria* pMemoria, int chave){
    int i, ichave, anti;

    if(memoriaVazia(pMemoria)) return -1;
    anti = -1;
    i = pMemoria->primeiro;
    ichave = pMemoria->items[i].item.chave;

    while((chave > ichave) && (i != -1)){
        anti = i;
        i = pMemoria->items[i].prox;
        if(i!=-1) ichave = pMemoria->items[i].item.chave;
    }
    return anti;
}

int insereItem(TMemoria* pMemoria, TItem* pItem){
    int idx, antidx, proxidx;
    if(pMemoria->numOcupadas==MAX_SIZE) return 0; // Uma das formas de
        verificar se ha espaco livre

    idx = pMemoria->priCelulaDisp;
    pMemoria->items[idx].item = *pItem;
    pMemoria->priCelulaDisp = pMemoria->items[idx].prox;
    if(!(pMemoria->priCelulaDisp==-1)) pMemoria->items[pMemoria->
        priCelulaDisp].ant = -1;

    if(memoriaVazia(pMemoria)){
        pMemoria->primeiro = idx;
        pMemoria->ultimo = idx;
        pMemoria->items[idx].prox = -1;
        pMemoria->items[idx].ant = -1;
```

```
            pMemoria->numOcupadas++;

            return 1;
        }

        // Encontra lugar na sequencia de chaves
        antidx = encontraAnterior(pMemoria, pItem->chave);
        pMemoria->items[idx].ant = antidx;
        // Caso o item ocupe a primeira posicao
        if(antidx == -1){
            proxidx = pMemoria->primeiro;
            pMemoria->primeiro = idx;
        } else{
            proxidx = pMemoria->items[antidx].prox;
            pMemoria->items[antidx].prox = idx;
        }
        pMemoria->items[idx].prox = proxidx;
        if(!(proxidx == -1)) pMemoria->items[proxidx].ant = idx;
        if(pItem->chave>pMemoria->items[pMemoria->ultimo].item.chave)
            pMemoria->ultimo = idx;
        pMemoria->numOcupadas++;

        return 1;
    }

    int removePrimeiro(TMemoria* pMemoria, TItem* out_item){
        int pri_idx, novopri_idx;

        pri_idx = pMemoria->primeiro; // Indice do item a ser removido
        if(pri_idx==-1) return 0;
        novopri_idx = pMemoria->items[pri_idx].prox; // Indice do novo
            primeiro item

        // Atualizando primeiro
        pMemoria->primeiro = novopri_idx;
        if(!novopri_idx==-1) pMemoria->items[novopri_idx].ant = -1;

        // Atualizando lista de memoria livre
        pMemoria->items[pri_idx].prox = pMemoria->priCelulaDisp;
        pMemoria->priCelulaDisp = pri_idx;

        // Setando valor de retorno em out_item
        *out_item = pMemoria->items[pri_idx].item;

        pMemoria->numOcupadas--;
        return 1;
    }

    int removeUltimo(TMemoria* pMemoria, TItem* out_item){
        int ult_idx, novoult_idx;

        ult_idx = pMemoria->ultimo; // Indice do item a ser removido
        if(ult_idx==-1) return 0;
        novoult_idx = pMemoria->items[ult_idx].ant; // Indice do novo
            primeiro item

        // Atualizando ultimo
        pMemoria->ultimo = novoult_idx;
```

```
    if(novoult_idx!=-1) pMemoria->items[novoult_idx].prox = -1;

    // Atualizando lista de memoria livre
    pMemoria->items[ult_idx].prox = pMemoria->priCelulaDisp;
    pMemoria->priCelulaDisp = ult_idx;

    // Setando valor de retorno em out_item
    *out_item = pMemoria->items[ult_idx].item;

    pMemoria->numOcupadas--;
    return 1;
}

/* Fancy output:
   Imprime sequencia da lista.
   Para fins de ser lindo. Ou quase. */
void imprimeSequencia(TMemoria *pMemoria){
    int i;

    if(memoriaVazia(pMemoria)){
        printf("Memoria toda vazia.\n");
        return;
    }
    i=pMemoria->primeiro;
    printf("Tamanho total: %d\n", pMemoria->numOcupadas);
    printf("addr  key  value\n");
    while(i!=-1){
        printf("%4d %4d %4d\n", i, pMemoria->items[i].item.chave,
                                   pMemoria->items[i].item.data);
        i = pMemoria->items[i].prox;
    }
}

/* Raw output:
   Imprime na sequencia da memoria, sem diferenciar livre de ocupado.
   Para fins de depuracao. */
void rawPrint(TMemoria *pMemoria){
    int i;

    for(i=0; i<MAX_SIZE; i++){
        printf("%4d %4d %4d %4d %4d\n", i, pMemoria->items[i].item.
            chave,
                                        pMemoria->items[i].item.data,
                                        pMemoria->items[i].ant,
                                        pMemoria->items[i].prox);
    }
}
```

Programa 2: TMemoria.c

2.1.2 Funções do tipo TMemoria

As funções referentes ao tipo TMemoria, implementadas no arquivo TMemoria.c, são as seguintes:

- zeraMemoria:

Recebe um ponteiro para um TMemoria (já alocado) e o inicializa para corresponder a um espaço vazio de memória. Como a função não realiza por si a alocação de memória, não há necessidade de verificação de sua saída, então seu retorno é vazio.

Nessa função também é definido o estado inicial da lista de células desocupadas do tipo TMemoria, ou seja, uma lista com todos os espaços do vetor disponível.

Antes que uma instância de TMemoria seja utilizada para qualquer fim, essa função DEVE ser invocada sobre ela para garantir a coerência dos parâmetros internos.

Formato:

```
int zeraMemoria(TMemoria* pMemoria)
```

Retorno: vazio.

- memoriaVazia:

  Recebe um ponteiro para um TMemoria e avisa se a instância está vazia ou não. A verificação é feita a partir da variálve numCelOcupadas.

  Formato:

```
int memoriaVazia(TMemoria* pMemoria)
```

  Retorno: 1 caso a instância de TMemoria esteja vazia, 0 caso contrário.

- numeroOcupadas:

  Similar à função zeraMemoria, mas retorna o número de celulas ocupadas da instância de TMemoria.

  Formato:

```
int numeroOcupadas(TMemoria* pMemoria)
```

  Retorno: Número (inteiro) de células ocupadas no espaço da memória.

- encontraAnterior:

  Função auxiliar para garantir a ordem dos cursores no momento da inserção de um novo ítem na lista.

  Recebe um ponteiro para um TMemoria e um inteiro correspondente a uma chave de ordenação.

  A função varre a instância de TMemoria comparando as chaves com a chave fornecida. A função retorna o índice no vetor da última célula cuja chave é menor que a chave fornecida.

  A forma correta de interpretar o retorno dessa função é de que seu retorno é o valor após o qual o ítem com aquela chave deve ser inserido.

  Formato:

```
int encontraAnterior(TMemoria* pMemoria, int chave)
```

  Retorno: Número inteiro do índice do elemento anterior ao ítem sendo inserido na lista.

- insereItem:

  Recebe um ponteiro para um TMemoria e outro para um TItem. A função insere o TItem na primeira posição de memória livre, então invoca encontraAnterior para saber qual ítem da lista é a chave imediatamente anterior à sua, então atualiza os cursores para manter a ordem.

  Formato:

```
int insereItem(TMemoria* pMemoria, TItem* pItem)
```

  Retorno: 1 em caso de sucesso, 0 caso o vetor não tenha espaço livre.

- removePrimeiro e removeUltimo:

  Funções gêmeas para remoçõa de ítens da lista de memória ocupada. Funcionam como uma forma de pop: extraem o conteúdo do elemento para retorno e então o removem da lista.

  A função adiciona o espaço desocupado como primeira célula desocupada na lista de memória livre, respeitando o comportamento de pilha proposto.

  O ítem removido é armazenado no espaço do ponteiro out_item, de forma que possa ser recuperado pela aplicação.

  Formato:

```
int removePrimeiro(TMemoria* pMemoria, TItem* out_item)
int removeUltimo(TMemoria* pMemoria, TItem* out_item)
```

  Retorno: 1 em caso de sucesso, 0 em caso de lista vazia.

- imprimeSequencia:

  Imprime a lista de ítens ocupados na ordem da lista, e não na ordem da memória. Após a impressão de cada elemento, o cursor para o próximo é carregado, até que o cursor -1 seja encontrado.

```
void imprimeSequencia(TMemoria* pMemoria)
```

  Retorno: vazio.

- rawPrint:

  Função similar a imprimeSequencia, mas para fins de depuração. Os ítens do vetor são impressos na ordem em que se encontram na memória, ignorando a ordem dos cursores e a diferença entre memória livre ou ocupada.

  Ao lado de cada ítem, os cursores "ant"e "prox"são exibidos nessa ordem.

```
void rawPrint(TMemoria* pMemoria)
```

  Retorno: vazio.

Tabela 1: Complexidade das funções em TMemoria.c

Função	F(n)	F(n) em O
zeraMemoria	N	$O(N)$
memoriaVazia	c	$O(1)$
numeroOcupadas	c	$O(1)$
encontraAnterior	$2n$	$O(n)$
insereItem	n	$O(n)$
removePrimeiro	c	$O(1)$
removeUltimo	c	$O(1)$
imprimeSequencia	n	$O(n)$
rawPrint	N	$O(N)$

2.2 Análise de complexidade dos algoritmos para TVetor

Nas funções tratadas, as variáveis de complexidade são os tamanhos das listas tratadas (na maioria dos casos, o tamanho da lista de ocupados).

Por convenção, as listas de memória ocupada e memória livre terão tamanho n, enquanto o vetor de memória total terá tamanho N (MAX_SIZE).

O resumo das funções complexidade está na tabela 1. A discussão pormenorizada se dará adiante.

2.2.1 zeraMemoria

A função zeraMemoria começa inicializando as variáveis de controle de TMemoria, e, em seguida, preenche a lista de células vazias.

Ocorre aqui uma série de acessos a memória, que serão o elemento relevante para definir o desemnho dessa função. Em cada iteração, é realizada uma escrita em ant e uma em prox. Assim, de acordo com o tamanho N do vetor:

$$F(N) = 2N + c \tag{1}$$

$$F(N) = O(N)$$

2.2.2 memoriaVazia e numeroOcupadas

As duas funções de verificação da situação da memória realizam uma verificação simples do valor referente à memória ocupada. Por simplicidade:

$$F(n) = c \tag{2}$$

Em notação O:

$$F(n) = O(n)$$

2.2.3 encontraAnterior

A função realiza a verificação ao longo da lista à procura da última chave menor que a fornecida. No pior caso, o número de iterações será o tamanho da lista de ocupadas.

A cada iteração, são realizados dois acessos à memória para leitura. Assim, a complexidade dessa função será:

$$F(n) = 2n + c \tag{3}$$

logo,

$$F(n) = O(n)$$

2.2.4 insereItem

Para a função insereItem, o melhor caso ocorre quando o ítem inserido é o primeiro a ser incluído na lista. Nessa situação, os valores dos cursores são pré definidos e serão atribuídos sem maiores verificações.

No pior caso, a posição em que o novo ítem será inserido na lista deve ser verificado a partir de comparações, com a função encontraAnterior. Após a obtenção do ponto de inserção do elemento na lista, todas as demais operações são fixas.

$$F(n) = (2n + c) + c_2 = 2n + c \tag{4}$$

logo,

$$F(n) = O(n)$$

2.2.5 removePrimeiro e removeUltimo

As duas funções são similares, variando apenas o índice a ser trocado. Não haverá dependência no tamanho da lista ou do vetor. Dessa forma:

$$F(n) = c \tag{5}$$

logo,

$$F(n) = O(1)$$

2.2.6 imprimeSequencia

O elemento relevante da função será a invocação de printf. O número de iterações será o tamanho da lista de memória ocupada, e o número de invocações de printf por iteração será 1. Juntando às outras duas invocações:

$$F(n) = n + 2 \tag{6}$$

assim,

$$F(n) = O(n)$$

2.2.7 rawPrint

Similar a imprimeSequencia, o número de invocações de printf será o fator relevante, exceto que aqui, o número de iterações será o tamanho do vetor de memória.

$$F(N) = N \tag{7}$$

logo,

$$F(N) = O(N)$$

3 Testes

O programa main.c foi escrito para realizar rotinas de teste com as funções de TMemoria. Sua invocação pode ser realizada com -v para uma interface de usuário mais amigável (com alguns printf para solicitação de dados).

Uma função auxiliar para realizar a entrada de dados dummy foi descrita nesse arquivo para criação de TItems. Para TItems diferentes, uma função diferente haveria de ser desenvolvida. Ela solicita o valor da chave e o valor do dado a ser inserido na lista.

A rotina main de main.c solicita inicialmente o número de entradas que se deseja realizar. A cada input do usuário, um ítem é criado e inserido na unidade de memória.

São realizadas em seguida duas chamadas de removePrimeiro para criar uma fragmentação na organização da memória para que um novo ítem seja inserido em seguida. Por fim, duas invocações de removeUltimo são realizadas.

A última etapa da rotina de testes é a impressão dos resultados. É realizada uma invocação de imprimeSequencia e outra de rawPrint, a primeira para conferir se a estrutura da lista de ocupadas está coerente, a segunda para que as duas listas possam ser analisadas em detalhe.

A última etapa do programa é a liberação do espaço de memória alocado.

Diferentes situações podem ser simuladas criando conjuntos de dados com diferentes disposições.

Para compilar o main, basta executar invocar make na pasta raíz do projeto.

Diferentes situações de inserção e remoção de dados na memória podem ser simulados com arquivos de input. O arquivo in_data.txt está incluso na raíz do projeto como exemplo: ele cria uma lista com 8 elementos. Como a função main realiza todas as inserções listadas seguidas de remoção e uma inserção extra, são fornecidos 9 conjuntos de chave e dado. Arquivos com essa mesma finalidade devem sempre ter $n + 1$ conjuntos de dados, onde n é o número de elementos de memória.

```
$ ./out < in_data.txt
```

Programa 3: Invocação do teste

Na versão acima, o parâmetro -v não é utilizado uma vez que a verbosidade de input são irrelevantes, já que todos os dados necessários estão em in_data.txt.

Uma segunda versão de dados de teste foi fornecida simulando um overflow de memória: tenta-se inserir 11 elementos na lista, sendo que ela tem capacidade para 10. O objetivo era verificar que não ocorria estouro de pilha. O arquivo era in_overflow.txt.

4 Conclusão

O trabalho serviu para demonstrar conceitos interessantes para controle de memória a partir de listas. Foi possível verificar que é possível controlar o acesso a memória sem que se mantenham grandes volumes de variáveis de controle desde que os conceitos das listas sejam respeitados ao longo da implementação.

O modelo final foi capaz de lidar com fragmentação e quantidades "ilegais"de dados.

Uma pilha, mesmo não ordenada linearmente dentro da memória, demonstrou ser uma forma interessante de organizar a memória livre para acesso por outras funções.

Referências

[1] Aho, Alfred V., Hopcroft, John E. e Ullman, Jeffrey D., *Data Structures and Algorithms*. Addison Wesley, 1983 (reimpresso com recorreções em 1987).

[2] Schildt, H., *C Completo e Total*, 3ed. Editora Makron, 1997.

[3] Ziviani, Nivio, *Projeto de Algoritmos com implementações em PASCAL e C*, 2ed, Thomson, 2004.