Métodos de Programação II
Ficha de Revisões para as aulas Teórico-Práticas

Manuel Bernardo Barbosa @ di.uminho.pt

February 19, 2003

1 Introdução

- O que é preciso saber para começar?
  - A sintaxe básica do C.
  - O que são variáveis estáticas e dinâmicas.
  - O que é um apontador.
  - Como é que se trabalha com apontadores em C (`*`, `->`, `NULL`, `malloc`, `sizeof`, `free`, etc).
  - Vantagens de utilizar alocação dinâmica de memória: nomeadamente em ambientes de multi-tasking é vital que a memória alocada a cada processo seja o mínimo necessário, para aumentar a eficiência global do sistema.
  - Diferenciar entre um tipo de dados abstracto (lista, fila, pilha) e os detalhes da sua implementação.
  - A definição de um tipo de dados abstracto consiste apenas numa especificação dos elementos desse tipo, de como eles se relacionam e das operações que se podem efectuar sobre esse tipo abstracto.
  - No desenvolvimento de uma aplicação deve primeiro considerar-se que tipo abstracto de dados é mais apropriado, e só depois pensar nos detalhes da sua implementação (aproximação top-down).

2 Stack (Pilha)

- Estrutura de dados que é uma lista na qual todas as inserções e remoções são feitas num dos extremos, denominado topo.
- Assim, o ultimo elemento a ser colocado numa pilha é sempre o primeiro a ser removido: é uma lista Last-In-First-Out (LIFO).
- Uma das analogias mais comuns é a de uma caixa com mola, em que para se introduzir um novo elemento se tem de pressionar (push) e para se retirar um elemento se tem de libertar o ultimo elemento colocado na pilha (pop).

- No entanto, esta analogia não é muito correcta, uma vez que quando se insere ou remove um elemento, todos os outros elementos são movimentados. Uma implementação baseada nesta filosofia seria pouco eficiente.
- Uma analogia mais correcta é a de uma pilha de objectos, e.g. livros, em que um elemento permanece no local onde é colocado, até ser removido. O que se altera é a posição em que se efectuam as inserções e remoções.
- Uma aplicação muito simples de uma pilha é a inversão de um texto: basta fazer push de todos os caracteres para a pilha, e depois fazer pop para os recuperar por ordem inversa. Outra aplicação típica das pilhas é a implementação de máquinas de calcular com base na notação Reverse Polish.
- Operações básicas sobre a estrutura de dados stack:
  - Inserir (Push)
  - Remover (Pop)
- Operações de gestão:
  - Criar
  - Esvaziar
  - Testar se está cheia
  - Testar se está vazia
  - Obter o tamanho
- Outras operações:
  - Top – Obter o valor do primeiro elemento da stack, sem o remover e.g. para um teste.
  - Percorrer (Traverse) – Percorrer todos os elementos da stack, aplicando-lhes uma determinada operação de processamento.
- Implementação contígua

```
#define MAXSTACK 10
typedef char StackEntry;
typedef struct stack {
   int top;
   StackEntry entry[MAXSTACK];
} Stack;
```

- Implementação ligada

```
typedef struct node {
   StackEntry entry;
   struct node *next;
} Node;

typedef struct stack {
   Node *top;
} Stack;
```

  - A melhor maneira de proceder à inserção e remoção é fazer corresponder o topo da pilha com o primeiro elemento da estrutura ligada. Assim não é preciso percorrer todos os elementos antes de se proceder a essa operação. A solução resultante é muito eficiente.
  - Porquê definir os dois tipos? Manter uma distinção lógica e permitir uma depuração mais rigorosa.

- Comparação entre implementações:

  - Como é que se sabe o tamanho da pilha? Na implementação contígua é imediato. Na implementação ligada há que percorrer a estrutura toda, ou manter um contador.
  - Na implementação ligada, o tamanho máximo da pilha é apenas limitado pela disponibilidade de memória.
  - Além disso, o espaço ocupado pela estrutura de dados é proporcional ao número de elementos. No caso da implementação contígua isto não acontece.
  - Na implementação ligada há um overhead de espaço de memória devido ao armazenamento dos apontadores.

- Exercícios:
  - Implemente as seguintes funções em C, correspondentes às operações sobre Stacks definidas acima. Faça-o para as duas implementações apresentadas.

    ```
    Stack *CreateStack();
    void Push(StackEntry entry, Stack *s);
    StackEntry Pop(Stack *s);
    void ClearStack(Stack *s);
    int StackSize(Stack s);
    int StackEmpty(Stack s);
    int StackFull(Stack s);
    StackEntry StackTop(Stack s);
    ```

  - Explique porque é que a passagem do parâmetro *s* varia de umas funções para as outras.
  - Utilize as duas implementações de pilhas que construiu para desenvolver duas pequenas aplicações que invertam um array de caracteres. Repare que o código que utiliza a pilha é independente da implementação que está a utilizar.
  - Utilize uma das implementações de pilhas que desenvolveu para implementar uma pequena máquina de calcular baseada na notação Reverse-Polish. Este tipo de calculadora mantém uma pilha de operandos, que pode ser alterada pelo utilizador. As operações são efectuadas sobre os elementos no topo da pilha, tantos quantos os necessários para o operador indicado pelo utilizador. Exemplo:

    ```
    PILHA VAZIA
    > 1
    INSERIU
    TOP=1
    > 2
    INSERIU
    TOP=2
    > +
    ADICIONOU DOIS ELEMENTOS
    TOP=3
    > +/-
    ```

```
CALCULOU O SIMETRICO DE UM ELEMENTO
TOP=-3
```

3 Queue (Fila)

- Estrutura de dados que é uma lista na qual todas as inserções são feitas num dos extremos (rear ou tail) e as remoções são feitas no outro (front).
- Assim, o primeiro elemento a ser colocado numa fila é sempre o primeiro a ser removido: é uma lista First-In-First-Out (FIFO).
- Uma das analogias mais comuns é a de uma fila de espera num qualquer serviço público, e.g. uma bilheteira. É nesta analogia que têm origem os nomes rear e front.
- As filas são (se é que isso é possível) ainda mais utilizadas que as pilhas em aplicações computacionais. De facto, é muito frequente haver a necessidade de implementar o conceito de fila de espera, seja para controlar os acessos a um determinado recurso, para processar pedidos, etc.
- Operações básicas sobre a estrutura de dados queue:
  - Inserir (Append)
  - Remover (Serve)
- Operações de gestão:
  - Criar
  - Esvaziar
  - Testar se está cheia
  - Testar se está vazia
  - Obter o tamanho
- Outras operações:
  - Front – Obter o valor do primeiro elemento da fila, sem o remover e.g. para um teste.

  - Percorrer (Traverse) – Percorrer todos os elementos da queue, aplicando-lhes uma determinada operação de processamento.

- Implementação contígua

```
#define MAXQUEUE 10
typedef char QueueEntry;

typedef struct queue {
   int count;
   int front;
   int rear;
   QueueEntry entry[MAXQUEUE];
} Queue;
```

  - Agora não basta guardar o índice do último elemento a ser inserido: é também necessário saber qual é o próximo elemento a ser removido.
  - Se se seguir o modelo físico à letra , o facto de os índices serem sempre incrementados, e nunca decrementados, coloca um problema: eventualmente chega-se ao fim do espaço reservado para a fila.
  - A solução é a adopção de uma implementação circular, em que quando se chega ao fim do espaço reservado, se volta ao início.
  - Neste caso, há que ter cuidado com os casos extremos i.e. usar contadores, flags, para detectar os casos em que a fila está cheia e vazia.

- Implementação ligada

```
typedef struct node {
   QueueEntry entry;
   struct node *next;
} Node;

typedef struct queue {
   Node *front;
   Node *rear;
} Queue;
```

  - Ao contrário do que acontece com as implementações contíguas, uma implementação ligada de uma fila não é mais complicada do que uma implementação ligada de uma pilha: as operações de remoção e inserção na fila são de complexidade equivalente às da pilha.

- Comparação entre implementações:

  - Na implementação ligada, o tamanho máximo da fila é apenas limitado pela disponibilidade de memória.
  - Além disso, o espaço ocupado pela estrutura de dados é proporcional ao número de elementos. No caso da implementação contígua isto não acontece.
  - Na implementação ligada há um overhead de espaço de memória devido ao armazenamento dos apontadores.

- Exercícios:

  - Implemente as seguintes funções em C, correspondentes às operações sobre Queues definidas acima. Faça-o para as duas implementações apresentadas.

    ```
    Queue *CreateQueue();
    void Append(QueueEntry entry, Queue *q);
    QueueEntry Serve(Queue *q);
    void ClearQueue(Queue *queue);
    int QueueSize(Queue q);
    int QueueEmpty(Queue q);
    int QueueFull(Queue q);
    QueueEntry QueueFront(Queue *q);
    ```

  - Modifique as funções que escreveu de forma a que retornem sempre um valor que indica o sucesso ou uma falha na operação. Os valores que são retornados nas versões acima, deverão ser convertidos em parâmetros passados por referência (usando apontadores).
  - Utilize as duas implementações de queues que construiu para desenvolver duas pequenas aplicações que imprimam os passos inermédios no calculo do factorial de um número inteiro de forma

recursiva. Para isso deverá escrever uma função recursiva que calcule o factorial de um número inteiro, e que adicione o valor de cada resultado parcial à queue. Uma possível declaração dessa função seria:

```
int factorial(int n, Queue *q);
```

Repare que o código que utiliza a queue é independente da implementação que está a utilizar. Repare também que a utilização da implementação contígua pode trazer alguns problemas.

4 Lista generalizada

- As pilhas e filas são casos particulares de uma estrutura de dados chamada lista.
- Uma lista é uma sequência finita de elementos de um determinado tipo, sobre as quais se definem um determinado conjunto de operações.
- Operações básicas sobre a estrutura de dados lista:
  - Inserir na posição n
  - Remover da posição n
  - Substituir na posição n
  - Ler da posição n
- perações de gestão:
  - Criar
  - Esvaziar
  - Testar se está cheia.
  - Testar se está vazia.
  - Obter o tamanho
- Outras operações:
  - Percorrer (Traverse) – Percorrer todos os elementos da lista, aplicando-lhes uma determinada operação de processamento.

- Implementação contígua:

```
#define MAXLIST 10
typedef char ListEntry;
typedef struct list {
   int count;
   ListEntry entry[MAXLIST];
} List;
```

  - As operações de inserção e remoção tornam-se bastante ineficientes devido á necessidade de deslocar elementos.
  - O seu tempo de execução é proporcional ao número de elementos da lista.
  - As restantes operações básicas demoram tempo constante.

- Implementação simplesmente ligada:

```
typedef struct node {
   ListEntry entry;
   struct node *next;
} Node;

typedef struct list {
   int count;
   Node *head;
} List;
```

  - A operação de ir para um elemento da lista torna-se proporcional ao tamanho da lista.
  - À parte desta componente, todas as outras operações básicas passam a ser de tempo de execução constante, mas é pouco eficiente.
  - Algumas variações podem ser utilizadas para melhorar este aspecto e.g. manter uma variável de estado a apontar para a posição actual na lista.
  - Note-se que isto apenas acelera movimentos num dos sentidos, uma vez que nunca conseguimos obter o elemento anterior.

- Implementação duplamente ligada:

```
typedef struct node {
   ListEntry entry;
   struct node *next;
   struct node *previous;
} Node;

typedef struct list {
   int count;
   int currentpos;
   Node *current;
} List;
```

  - Mantendo dois apontadores em cada nó, passamos a conseguir mover-nos de forma eficiente em ambos os sentidos na lista.
  - Desta forma não faz sentido guardar sempre o primeiro elemento, é muito mais eficiente guardar sempre o ultimo a ser acedido.
  - As operações de inserção e remoção tornam-se um pouco mais complexas devido à manipulação de apontadores.
  - Há algum overhead adicional devido ao armazenamento de mais um apontador por nó.

- Um problema genérico das implementações ligadas é não serem adequadas a acessos aleatórios.

- Em geral, as implementações ligadas são preferíveis quando os registos são grandes, quando não se sabe antecipadamente o tamanho da lista, quando é necessária mais flexibilidade na inserção, remoção e reorganização de entradas.

- Exercícios:

  - Implemente as seguintes funções em C, correspondentes às operações sobre Listas definidas acima. Faça-o para as três implementações apresentadas.

```
void InsertList(int position, ListEntry entry, List *list};
ListEntry DeleteList(int position, List *list};
ListEntry RetrieveList(int position, List *list};
ListEntry ReplaceList(int position, ListEntry entry, List *list};
List *CreateList();
void ClearList(List *list);
int ListEmpty(List list);
int ListFull(List list);
int ListSize(List list);
```

- Utilize as três implementações de listas que construiu para desenvolver três pequenas aplicações que apliquem o algoritmo de ordenação Merge Sort a uma lista de números inteiros. Repare que o código que utiliza a lista é independente da implementação que está a utilizar.
- Compare os tempos de execução das aplicações que desenvolveu para listas com um número significativo de elementos aleatórios. Retire conclusões.

5 Referências

"Data Structures and Program Design in C", 2^{nd} Edition
Robert Kruse, C.L.Tondo and Bruce Leung
Prentice Hall 1997