ED – Estrutura de Dados – 2015/2

T2 – 2o. Trabalho Prático de Implementação

27 de outubro de 2015

1 Objetivo

O objetivo deste trabalho é desenvolver um programa em C para resolver o *puzzle* da Torre de Hanoi usando pilhas alocadas dinamicamente.

2 Descrição do Problema

- Implemente um tipo abstrato de dado (TAD) pilha. A sua pilha não precisa ser heterogênea (isto é, suportar o armazenamento de tipos diferentes) mas ela deve ser *genérica*. Isto é, defina um tipo `info_t` como por exemplo

  ```
  typedef info_t int;
  ```

  e use-o consistentemente ao longo da implementação da sua pilha.

- A sua pilha deve ser implementada com alocação dinâmica de memória e pode ter um tamanho máximo permitido (use uma macro). Você é livre para escolher se quer fazer a sua pilha com implementação encadeada ou sequencial.

- A sua implementação deve prover ao menos as seguintes operações sobre a pilha:

  1. **Create:** Criação (alocação) de pilhas vazias.
  2. **Size:** Tamanho da pilha.
  3. **Push:** Inserir um elemento no topo da pilha.
  4. **Pop:** Remover o elemento do topo da pilha.
  5. **Peek:** Inspecionar o elemento do topo da pilha mas sem remover.
  6. **Print:** Imprimir o conteúdo da pilha no terminal.

- Usando a sua TAD pilha, implemente um programa que resolve o *puzzle* da Torre de Hanoi (detalhes em `http://en.wikipedia.org/wiki/Tower_of_Hanoi`) para QUALQUER número de discos. O número de pinos é fixo em três.

- Chame os pinos de A, B e C, seguindo da esquerda para a direita. Assim, o objetivo do *puzzle* é passar todos os discos do pino A para o pino C.

- Identifique cada disco com um número inteiro que indica o seu tamanho. Para um problema com n discos, o disco 1 é o menor e o disco n é o maior.

- O seu programa deve receber o número de discos como um parâmetro. NÃO use `scanf` para ler esse número, ele é um parâmetro da sua função principal, que deve ter o seguinte cabeçalho:

  ```
  int main (int argc, char *argv[])
  ```

  Veja um exemplo de como usar argumentos em: `http://crasseux.com/books/ctutorial/argc-and-argv.html`.

- Caso o seu programa seja executado com a opção `-v`, ele deve imprimir no terminal a sequência de passos para resolver o *puzzle* (veja exemplo abaixo). Além disso, ele deve imprimir a configuração inicial e final das pilhas. Se a opção `-v` não for usada, o seu programa não precisa imprimir nada na tela, mas ele ainda deve resolver o *puzzle*.

- Exemplo de execução e saída do seu programa:

```
$ ./trab2 4 -v
=> Solving the Tower of Hanoi with 4 disks.
=> Initial configuration:
   A: 4 3 2 1
   B:
   C:
=> Moves:
   Move disk 1 from A to B
   Move disk 2 from A to C
   Move disk 1 from B to C
   Move disk 3 from A to B
   Move disk 1 from C to A
   Move disk 2 from C to B
   Move disk 1 from A to B
   Move disk 4 from A to C
   Move disk 1 from B to C
   Move disk 2 from B to A
   Move disk 1 from C to A
   Move disk 3 from B to C
   Move disk 1 from A to B
   Move disk 2 from A to C
   Move disk 1 from B to C
=> Final configuration:
   A:
   B:
   C: 4 3 2 1
```

- A solução de um problema com n discos requer $2^n - 1$ movimentos. Este é, portanto, um exemplo de um problema com crescimento exponencial. Após terminar a sua implementação, realize experimentos com o seu programa para ver como o desempenho dele varia conforme n aumenta. Para até que valor de n o seu programa consegue rodar num tempo razoável? (*Obs.: Faça todos os testes SEM a opção* `-v`*, pois a impressão na tela mascara o tempo de execução real do programa.*)

- Meça o tempo de execução do seu programa com o utilitário `time` (`http://www.ewhathow.com/2013/10/how-to-measure-program-execution-time-in-linux/`). Para o número de discos variando de 3 até o limite máximo que o seu programa aguentar, monte uma tabela com o tempo real de execução. Coloque tudo em um arquivo chamado `resultados.txt`. Exemplo de um conteúdo desse arquivo:

```
$ cat resultados.txt
Number of disks         Real time
3                       0m0.333s
4                       0m1.333s
5                       0m7.333s
6                       0m100.333s
...
```

3 Regras para Desenvolvimento e Entrega do Trabalho

- **Data da Entrega**: O trabalho deve ser entregue até às 23:55 h do dia 17/11/2015 (3a. feira). Não serão aceitos trabalhos após essa data.
- **Grupo**: O trabalho é *individual*.
- **Linguagem de Programação**: Você deverá implementar o seu trabalho na linguagem C.
- **Ferramentas**:
  - O seu trabalho será corrigido no Linux.
  - Use a ferramenta Valgrind (`http://valgrind.org`) para garantir a ausência de *memory leaks*.
- **Como entregar**: Pela atividade criada no Moodle. Envie um arquivo compactado com todo o seu trabalho. Envie também um arquivo LEIAME com as instruções para compilar o seu trabalho. (Faça isso mesmo que seja necessário apenas um simples comando, tal como `gcc *.c` ou algo similar.) Envie também o seu arquivo de resultados.
- **Recomendações**: Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

4 Avaliação

- O trabalho vale 20 pontos.
- Trabalhos com erros de compilação receberão nota zero.
- Caso seja detectado plágio, todos os envolvidos receberão nota zero.
- Se o seu trabalho tiver *memory leaks* haverá desconto de pontos na nota do trabalho. Tal desconto será proporcional à gravidade das falhas detectadas no código.
- O aluno com a implementação CORRETA mais eficiente ganha um ponto extra na nota. Todos os testes de performances serão refeitos pelo professor na mesma máquina para evitar disparidades de performance devido ao hardware. (*Obs.: o objetivo do ponto extra é estabelecer uma competição "saudável" entre os alunos. Note que ninguém será prejudicado por esse aspecto de competição e que qualquer um é livre para abraçá-lo ou não. Mesmo que o seu programa tenha um desempenho lento em relação aos demais, se ele estiver correto você tem condições de tirar a nota máxima de 20 pontos.*)