Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Informática (DAINF)
Estrutura de Dados I
Professor: Rodrigo Minetto
Lista de exercícios (estrutura de dados pilha)

Exercícios (seleção): necessário entregar TODOS (moodle)!

Exercício 1) Suponha a inserção de seis elementos aleatórios em uma pilha s. Agora suponha que é necessário eliminar todos os elementos pares desta pilha s, de tal forma a manter em s a ordem dos elementos remanescentes. Por exemplo:

```
      ANTES          DEPOIS
peek ->
         0
         1         peek ->
         4                 1
         3         s =     3
         8
s =      6
```

Resolva o problema acima com o uso de pilha auxiliar (se achar necessário), e o uso das operações push, pop, e empty (não use operações de baixo nível com o operador → na resolução deste exercício). Utilize como base o programa incompleto abaixo.

```
#include "stack.h"
int main () {
   int tam = 6;
   Stack *s = create (tam);
   int i;
   for (i = 0; i < tam; i++) {
     push (s, rand()%10);
   }
   print (s);
   /*TERMINAR*/
   destroy (s);
}
```

Utilize o programa **elimina-par.c** para auxiliá-lo neste problema (em **arquivos.zip**).

Exercício 2) Escreva uma função que retorna verdadeiro se uma sequência de parênteses e/ou colchetes em uma string *c* está "bem formada" (ou seja, parênteses e colchetes são fechados na ordem inversa àquela em que foram abertos), ou falso caso contrário. Exemplos:

```
Input                        Output
c = "( ( ) [ ( ) ] )"        true  (bem formada)
c = "( [ ) ]"                false (mal formada)
c = "( ) [ ]"                true  (bem formada)
c = "( [ ]"                  false (mal formada)
c = ") ("                    false (mal formada)
```

Considere o seguinte protótipo para a sua função:

```
int parser (char *c);
```

Não acesse os elementos armazenados na pilha diretamente, utilize as operações definidas pela interface da estrutura de dados.

Utilize o programa **parser.c** para auxiliá-lo neste problema (em **arquivos.zip**).

Exercício 3) A notação Polonesa (reversa) ou pós-fixa foi proposta pelo matemático polonês Jan Lukasiewicz, como uma forma de escrever expressões aritméticas e lógicas sem a necessidade de utilizar parênteses. Posteriormente, ela se mostrou apropriada para a representação dessas expressões em linguagens de programação. Essa notação foi, também, popularizada por ter sido adotada em diversas calculadoras científicas, notadamente as calculadoras da marca HP. Escreva uma função baseada na estrutura de dados **pilha** para converter expressões numéricas para a notação polonesa. Por exemplo, considere que uma string *c* contenha a seguinte expressão numérica

```
( 9 + ( ( 0 + 1 ) * ( 2 * 3 ) ) )
```

após a transformação a expressão é dada por

```
9 0 1 + 2 3 * * +
```

Considere o seguinte protótipo para a sua função:

```
void reversed_polish_notation (char *c);
```

Suponha que a expressão numérica está armazenada em uma string *c* como entrada, e que a função imprime a notação polonesa como saída na tela (não é necessário armazenar). Também considere que cada operando numérico é formado por apenas um inteiro (para critérios de simplificação).

Não acesse os elementos armazenados na pilha diretamente, utilize as as operações definidas pela interface da estrutura de dados.

Utilize o programa **polish.c** para auxiliá-lo neste problema (em **arquivos.zip**).

Dica: utilize apenas uma pilha na resolução desse problema; alguns símbolos que aparecem na string *c* precisam ser fechados com outros distantes, o emparelhamento de quando abrem ou fecham se dá através de operações de push e pop.

Exercícios (aprofundamento): não é necessário entregar mas é importante estudar!

Exercício 4) Discuta se é possível utilizar ou modificar a estrutura de pilha que definimos em aula — sem utilizar um vetor (ou array) adicional e sem deslocar os valores armazenados — para simular **duas** pilhas. Não é necessário programar, apenas apontar qual seriam as modificações caso possível.

Exercício 5) Escreva uma função baseada na estrutura de dados **pilha** para avaliar expressões em notação polonesa (reversa). Por exemplo, considere que uma string c contenha a seguinte expressão

```
9 0 1 + 2 3 * * +
```

após a resolução, a sua função deve retornar como resultado o cálculo do valor da expressão, que é **15** (o resultado de 9+((0+1)*(2*3))). Considere o seguinte protótipo para a sua função:

```
int compute_polish_expression (char *c);
```

Suponha que a expressão numérica em notação polonesa está armazenada em uma string c como entrada. Também considere que cada operando numérico é formado por apenas um inteiro (para critérios de simplificação).

Não acesse os elementos armazenados na pilha diretamente, utilize as as operações definidas pela interface da estrutura de dados.

Dica: utilize apenas uma pilha na resolução desse problema; vefique que operações ou operandos que ativam a chamada da função push ou pop.

Exercício 6) Qual a complexidade, utilizando a notação assintótica $\mathcal{O}(?)$, para as seguintes operações básicas de uma estrutura de dados **pilha**

	Melhor caso	Pior caso
push		
pop		
peek		
empty		

Exercício 7) Escreva uma função que retorna verdadeiro se uma cadeia de caracteres em uma string c é palíndromo, ou falso caso contrário. Uma cadeia de caracteres é palíndromo quando mantém o mesmo sentido quando lida de trás pra frente: radar, asa, renner, etc. Curiosidade: palíndromos são expressões muito utilizadas na publicidade porque acredita-se que são mais fáceis de memorizar, mesmo que o consumidor não perceba. Considere o seguinte protótipo para a sua função:

```
int palindrome (char *c);
```

Não acesse os elementos armazenados na pilha diretamente, utilize as operações definidas pela interface da estrutura de dados, ou seja, utilize apenas as funções **push**, **pop** e **empty** para resolver esse problema.

Exercício 8) Para um dado número inteiro $n > 1$, o menor inteiro $d > 1$ que divide n é chamado de fator primo. É possível determinar a fatoração prima de n achando-se o fator primo d e substituindo n pelo quociente n/d, repetindo essa operação até que n seja igual a 1. Escreva uma função que compute a fatoração prima de um número n e imprima os seus fatores em ordem decrescente. Por exemplo, para $n = 3960$, deverá ser impresso 11 * 5 * 3 * 3 * 2 * 2 * 2. Considere o seguinte protótipo para a sua função:

```
void prime_factorization (int n);
```

Não acesse os elementos armazenados na pilha diretamente, utilize as as operações definidas pela interface da estrutura de dados.

Exercício 9) Escreva uma função **reverse** que inverte os elementos de uma fila q usando uma pilha. A função reverse só pode utilizar as operações definidas pela interface do tipo estruturado pilha (create, push, pop, empty, peek, ...). **Não** acesse os elementos armazenados na pilha senão pelas operações (funções) definidas pela interface (ou seja, suponha que você não tenha acesso ao código fonte que implementa a pilha, portanto não use o operador $\rightarrow$ para acessar os elementos da fila diretamente).

```
void reverse (Queue *q);
```

Utilize o programa **reverse.c** para auxiliá-lo neste problema (em **arquivos.zip**).

Exercício 10) Projete um jogo tal que dois jogadores recebem uma sequência aleatória de n números cada e uma pilha para armazenar esses números. Inicialmente, os jogadores empilham seus respectivos números. A cada jogada, ambos os jogadores desempilham um número (esses números não voltam a ser empilhados no jogo, ou seja, são descartados). Seja n_1 e n_2 os números desempilhados para os jogadores 1 e 2, respectivamente:

- se $n_1 > n_2$, então $(n_1 - n_2)$ números de elementos da pilha do jogador 1 são desempilhados e empilhados na pilha do jogador 2.
- se $n_2 > n_1$, então $(n_2 - n_1)$ números de elementos da pilha do jogador 2 são desempilhados e empilhados na pilha do jogador 1.

- se $n_2 = n_1$, nenhuma ação é realizada e o jogo continua.

Um jogador é declarado vencedor se a sua pilha se torna vazia.

Considere o seguinte protótipo para a sua função:

```
void game (Stack *stack_player_a, Stack *stack_player_b);
```

Exercício 11) Suponha um documento HTML simples, definido da seguinte forma:

```
<html>
 <body>
   <center>
       <h1> The little boat </h1>
   </center>
   <p> The storm tossed the little boat like a cheap sneaker in an
   old washing machine. The three drunker fisherman were used to
   such treatment, of course, but not the three salesman, who even
   as a stowaway now felt that he had overpaid for the voyage. </p>
   <ol>
     <li> Will the salesman die? </li>
     <li> What color is the boat? </li>
     <li> What about Naomi? </li>
   </ol>
 </body>
</html>
```

Note que o modo de iniciar e finalizar um tag em HTML tem a seguinte forma: "<tag>" e "</tag>", respectivamente.

Escreva um programa baseado na estrutura de dados **pilha** para determinar se um programador construiu corretamente (retorna **true**) ou não (retorna **false**) um código HTML válido. Considere o seguinte protótipo para a sua função:

```
int html (FILE *file);
```

Não acesse os elementos armazenados na pilha diretamente, utilize as as operações definidas pela interface da estrutura de dados.

Ps. para simplificar suponha que as **tags** no arquivo html são sempre separadas do restante do arquivo por espaços.

Exercício 12) A estrutura de dados **pilha** é um clássico para encontrar caminhos em um labirinto. Considere neste exercício labirintos representados por matrizes, como exemplo, considere o seguinte labirinto com tamanho 10×10 abaixo, tal que paredes são representadas por '#', vazio como posição possível para movimento, e os caminhos de solução são determinados por '*****', note também que as coordenadas de início e saída do labirinto são determinadas pelo usuário

```
Labirinto                    Ini:(0,0), Fim:(9,9)          Ini:(9,2), Fim:(6,7)
 0 1 2 3 4 5 6 7 8 9
0  #   #   # #              I #   #   # #                #   #   # # * * *
1  #   #   # #   #          * #   #   # #   #              #   #   # # * # *
2                  #        *                 #          * * * * * * * * # *
3  # # # # # # # #          * # # # # # # # # #          * # # # # # # # # *
4  # # # # # # #            * # # # # # # #              * # # # # # # * * *
5  #         #   # #        * # * * *   #   # #          * #         # * # #
6  #   #   # #   # #        * # * # * # #   # #          * #   #   # # F # #
7      #     # # # #        * * * # * * # # # #          * * * #     # # # #
8  #   # #   # # # #          #   # # * # # # #            # * # #   # # # #
9  #   # #                    #   # # * * * * F            # I # #
```

A estrutura de dados pilha pode ser utilizada para resolver este problema de forma elegante, e um conceito relacionado a este problema é denominado **back-tracking**, ou seja, a cada situação em que é possível seguir por dois ou mais caminhos diferentes (por exemplo uma bifurcação), é essencial armazenar na pilha esses possíveis ramos de exploração e caso uma escolha fracasse (caminho sem saída), estes ramos alternativos são desempilhados e explorados.

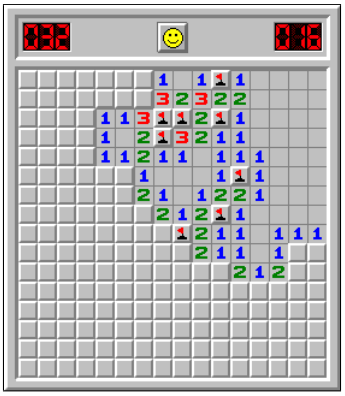
A estratégia para explorar um labirinto com uma pilha se dá através dos seguintes passos: 1) empilhe a posição de partida; 2) da posição de partida analise todos os quatro vizinhos (na direção horizontal e vertical) e empilhe as coordenadas daqueles que estão dentro da matriz e que permitam movimento, espaço vazio, e indique na matriz com algum símbolo que eles estão sendo processados (troque o vazio por um símbolo '**v**' para indicar que foram visitados); 4) a coordenada da posição que estiver no topo da pilha é utilizada para repetir o passo 2); 5) caso uma coordenada seja o ponto final então um caminho foi encontrado.

Implemente a funcionalidade que falta neste problema conforme indicado no programa **maze.c** (em **arquivos.zip**).

Exercício 13) O jogo **campo minado** é um muito popular, e consiste em revelar um campo de minas sem que alguma seja detonada. Foi inventado por Robert Donner em 1989. A área de jogo consiste num campo retangular formado por células. Cada célula é revelada ao escolher aquela coordenada, e se ela contiver uma mina, então o jogo acaba, caso contrário existem duas possibilidades:

- Um número aparece, indicando a quantidade de células adjacentes que contêm minas;
- Nenhum número aparece, e neste caso, o jogo revela automaticamente as células que se encontram adjacentes a célula vazia, já que não podem conter minas;

O jogo termina quando todas células que não têm minas são reveladas.



A estrutura de dados pilha é útil no jogo acima quando o usuário escolhe uma posição vazia no campo, tal que a estratégia para revelar parte do campo minado se dá através dos seguintes passos: (1) empilhe a posição vazia escolhida pelo jogador como semente inicial; (2) enquanto a pilha não estiver vazia desempilhe alguma posição do campo (da primeira vez existirá somente a semente inicial); (3) se a posição desempilhada for vazia (não contiver bombas nem números) então: (3.1) verifique para cada um dos 8 vizinhos da posição desempilhada (vizinhos na horizontal, vertical e diagonais) se ele não é uma bomba, se já não está visível ou se ele não está fora das dimensões do campo minado, se nenhuma das condições for verdadeira então empilhe essa posição vizinha e a marque como visível para o jogador; (3.2) repita o passo (2).

Implemente a funcionalidade que falta neste jogo conforme indicado no programa **minesweeper.c** (em **arquivos.zip**).