Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Informática (DAINF)
Estrutura de Dados I
Professor: Rodrigo Minetto
Lista de exercícios (lista com encadeamento duplo)

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

Exercício 1) Qual o propósito da função abaixo?

```
List* mistério (List *l, int elem) {
  Lista *no = search (lista, elem);
  if (no == NULL) { return lista; }
  if (lista == no) { lista = no->next; }
  if (no->next != NULL) { no->next->prev = no->prev; }
  if (no->prev != NULL) { no->prev->next = no->next; }
  free(no)
  return l;
}
```

Exercício 2) Escreva uma função para inserir na cauda da lista (última posição), **sem recursão**. Observe que o ponteiro na função main aponta para a cabeça da lista, ou seja, é necessário percorrer a lista para determinar a posição correta. Considere o seguinte protótipo para a sua função:

```
List* insert_back (List *l, int elem);
```

Qual a complexidade dessa operação em termos da notação $\mathcal{O}(?)$

Para resolver este exercício utilize o código **list.c** (em '**arquivos.zip**').

Exercício 3) Projete uma função que recebe como argumento de entrada uma lista com encadeamento duplo e transforma esta mesma lista em uma **lista circular** com encadeamento duplo, **sem recursão**. Por exemplo:

```
L = |1|<->|2|<->|3|       L = |1|<->|2|<->|3|
                                ^           ^
                                |___________|
```

Considere o seguinte protótipo para a sua função:

```
List* create_circular (List *l);
```

Para ter certeza que você conseguiu implentar esse exercício, use a função imprimir para verificar se os números são mostrados de forma repetida (em loop infinito), pois não existe mais o NULL como critério da parada quando a cauda se conecta com a cabeça. Por exemplo, a saída da função imprimir para o exemplo acima seria:

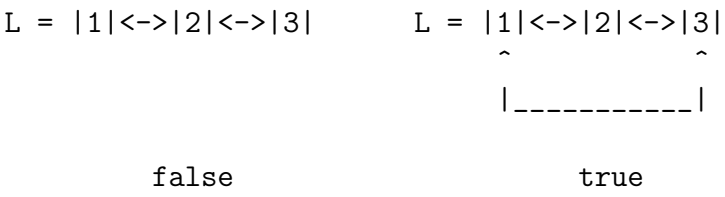
```
1 2 3 1 2 3 1 2 3 1 2 3 ...
```

Para resolver este exercício utilize o código **circular.c** (em '**arquivos.zip**').

Ps. verificar se a lista circular mantém o duplo encadeamento.

Exercícios (aprofundamento): escolha UM para entregar!

Exercício 4) Projete uma função que recebe como argumento de entrada uma lista com encadeamento duplo — cujo endereço não se situa necessariamente na cabeça da lista, por exemplo, o endereço pode ser o resultado da função search — e determina se ela **é ou não circular**. Por exemplo:



Considere o seguinte protótipo para a sua função:

```
int is_circular (List *l);
```

Para resolver este exercício utilize o código **circular.c** (em '**arquivos.zip**').

Exercício 5) Qual a complexidade, utilizando a notação assintótica $\mathcal{O}(?)$, para as seguintes operações abaixo, implementadas em uma lista com encadeamento duplo.

	Melhor caso	Pior caso
inserir cabeça		
inserir cauda		
inserir qq posição		
remover cabeça		
remover cauda		
remover qq posição		
busca (search)		

Exercício 6) Suponha que x e y sejam dois endereços de nós de uma lista com encadeamento duplo ℓ, suponha também que $x \neq \ell$, $x \neq y$, todos os elementos de ℓ distintos, e que x aparece antes de y na lista. Escreva uma função que corta o pedaço que vai de x a y em ℓ, e retorna esse trecho como uma nova lista ℓ_s. Por exemplo, suponha uma lista $\ell = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ e que $x = 3$ e $y = 7$, então $\ell = \{0, 1, 2, 8, 9\}$ e $\ell_s = \{3, 4, 5, 6, 7\}$. Considere o seguinte protótipo para a sua função:

```
List* split (List *l, List *x, List *y);
```

Para resolver este exercício utilize o código **split.c** (em '**arquivos.zip**').

Exercício 7) Escreva uma função que replica cada um nós armazenados na lista k vezes. Uma réplica é um novo nó adicionado na mesma lista com conteúdo igual ao original. Por exemplo, se a lista é representada por $\ell = \{1, 2, 3\}$ e $k = 3$, então o resultado esperado é dado por $\ell = \{1, 1, 1, 2, 2, 2, 3, 3, 3\}$. Formalmente, se $\ell = \{e_1, \ldots, e_n\}$ então a função réplica produz como resultado $\ell = \{e_1, e_1, e_1, \ldots, e_n, e_n, e_n, \ldots\}$ tal que cada elemento aparece exatamente k vezes. Considere o seguinte protótipo para a sua função:

```
List* replicate (List *l, int k);
```

Para resolver este exercício utilize o código **replicate.c** (em '**arquivos.zip**').

Exercício 8) O crivo de Eratóstenes é um algoritmo simples e prático para encontrar números primos até um certo valor limite. Foi criado pelo matemático grego Eratóstenes (285-194 A.C.), o terceiro bibliotecário chefe da biblioteca de Alexandria. O algoritmo é exemplificado abaixo.

- Defina um intervalo de $1 \ldots n$.
- Determine $\sqrt{n}$ (maior número a ser verificado)
- Crie uma lista de inteiros de 2 até n: {2,3,4,5,6,7,8,9,...,n}
- Remova todos os múltiplos do primeiro item da lista (número 2).
    * {2,3,5,7,9,...,n}
- Com o próximo primo da lista repita o procedimento (número 3)
    * {2,3,5,7,...,n}
- A busca para esse exemplo se encerra no número 5.

Se escolher este problema para resolver então utilize o código **crivo.c** (em '**arquivos.zip**') e o seguinte protótipo para a sua função:

```
void crivo_eratostenes (int n);
```

Exercício 9) O objetivo deste exercício é aplicar os conceitos de um tipo abstrato de dados conhecido como **deque**, que vem do inglês double-ended queue (abreviado como deque e pronunciado como "Deck"). O deque é a generalização de uma fila, ou seja, os elementos podem ser adicionados ou removidos da frente (cabeça) ou de trás (cauda). Deques são geralmente implementados como listas duplamente ligadas.

O jogo Snake, de 1976, consiste em controlar uma serpente que se move por um campo. Na versão mais básica a serpente não pode colidir com as paredes que cercam a área do jogo. Cada vez que a serpente se alimenta a sua cauda cresce, aumentando a dificuldade do jogo. O usuário controla a direção da serpente (para cima, para baixo, esquerda e direita). Utilize os códigos em "**snake**" (dentro de "**arquivos.zip**" em anexo ao material de aula no moodle) e implemente as seguintes operações para simular um TAD Deck:

```
typedef struct {
  int x;
  int y;
} Point;

typedef struct deque {
  Point p;
  struct deque *next;
  struct deque *prev;
} Deque;

Deque* insert_front (Deque *d, Point p);  /*Adiciona um item na cabeça da lista.*/
Deque* insert_back (Deque *d, Point p); /*Adiciona um item na cauda da lista.*/
Deque* delete_front (Deque *d); /*Remove um item da cabeça da lista.*/
Deque* delete_back (Deque *d); /*Remove um item da cauda da lista.*/
Point get_front (Deque *d); /*Retorna sem remover o item na cabeça da lista.*/
Point get_back (Deque *d); /*Retorna sem remover o item na cauda da lista.*/
```

O programa "**snake.c**" já tem algumas funcionalidades implementadas como a criação do campo do jogo, leitura de teclas para realizar o movimento da serpente pelo campo — esquerda (tecla 'l'), direita (tecla 'r'), para cima (tecla 'u') e para baixo (tecla 'd'). A funcionalidade fundamental a ser implementar nesse exercício é o movimento da serpente. Se for possível, também implemente a ocorrência de recompensas em posições aleatórias no campo e o crescimento da serpente ao consumir as recompensas.