INF1007 - PROGRAMAÇÃO II

LISTA DE EXERCÍCIOS 15

1. Um número racional é expresso por dois inteiros: um numerador e um denominador (este último diferente de zero!). Implemente um TAD para representar números racionais e definir algumas operações simples sobre esse tipo de dado. O tipo Racional deve ser representado pelo seguinte tipo estruturado:

```
struct racional {
  int numerador;
  int denominador;
};
```

O tipo Racional deve ser definido como um ponteiro para a estrutura racional:

```
typedef struct racional *Racional;
```

O TAD Racional deve implementar as seguintes funções:

- `racional_cria` – A função recebe um numerador e denominador, aloca dinamicamente a estrutura que representará o número racional e retorna seu endereço. Caso não seja possível alocar a memória, a função deve exibir uma mensagem e retornar NULL. Essa função tem o seguinte protótipo:

```
Racional racional_cria(int numerador, int denominador);
```

ATENÇÃO: a representação interna do racional deve sempre usar os menores números possíveis. Por exemplo, se o a função receber "4/6", deve-se armazenar "2/3". Para realizar essa transformação utilize o calculo do MDC:

```
static int mdc(int x, int y)
{
  if (y == 0)
    return x;
  return mdc(y, x % y);
}
```

A função mdc deve ser implementada dentro do TAD Racional e estar acessível somente internamente para o TAD.

- `racional_mostra` – A função recebe um número Racional e deve exibir na tela o numerador e o denominador do número racional como uma fração. Exemplo 2/3 (isto é, colocando uma '/' entre eles). Essa função tem o seguinte protótipo:

```
void racional_mostra(Racional r);
```

- `racional_libera` – A função recebe um Racional e libera a memória alocada pela estrutura. Essa função tem o seguinte protótipo:

```
void racional_libera(Racional r);
```

- `racional_multiplica` – A função recebe dois Racionais e retorna um novo Racional, que é o produto dos Racionais recebidos como parâmetros. Note que o novo Racional deve ter os menores valores possíveis para numerador e denominador. Essa função tem o seguinte protótipo:

```
Racional racional_multiplica(Racional r1, Racional r2);
```

- `racional_soma` – A função recebe dois Racionais e retorna um novo Racional, que é a soma dos Racionais recebidos como parâmetros. Note que o novo Racional deve ter os menores valores possíveis para numerador e denominador. Essa função tem o seguinte protótipo:

```
Racional racional_soma(Racional r1, Racional r2);
```

- `racional_compara` – A função recebe dois Racionais e deve retornar um valor menor que zero se o primeiro Racional é menor que o segundo, zero se são iguais e um valor maior que zero se o primeiro Racional é maior que o segundo. Essa função tem o seguinte protótipo:

```
int racional_compara(Racional r1, Racional r2);
```

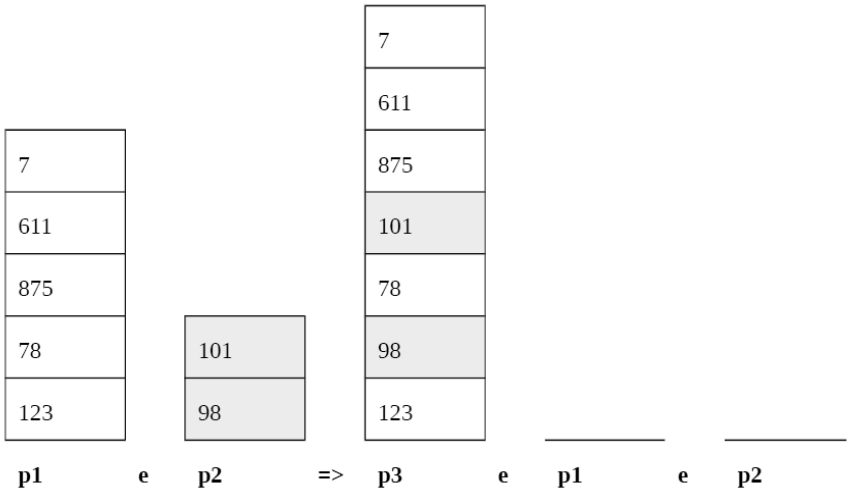
Após implementar as funções, crie o módulo principal do programa para ler dois números racionais informados pelo usuário e realizar as operações de multiplicação, soma e comparação dos números racionais utilizando as funções do TAD Racional.

2. Implemente um TAD Pilha como lista encadeada para armazenar números inteiros. O TAD deve exportar e implementar as seguintes funções:

```
Pilha* pilha_cria();
void pilha_push(Pilha* p, int v);
int pilha_pop(Pilha* p);
int pilha_vazia(Pilha* p);
void pilha_libera(Pilha* p);
```

Utilizando o TAD Pilha, implemente a seguinte função:

- `mesclaPilhas` – A função recebe duas pilhas (p1 e p2) e retorna uma terceira pilha (p3) que é composta pela mescla dos elementos das duas outras, segundo o exemplo a seguir (observe que ao final da execução p1 e p2 deverão estar vazias).



ATENÇÃO: A função mesclaPilhas não pertence ao módulo de implementação do TAD pilha! Ela pode usar apenas as funções definidas na interface deste TAD. A função tem o seguinte protótipo:

```
Pilha* mesclaPilhas(Pilha *p1, Pilha *p2);
```

Continue a implementação do programa, adicionando a ele um TAD Lista, que armazena valores inteiros. Os nós desta lista simplesmente encadeada são representados pela seguinte estrutura:

```
struct noLista {
  int info;
  struct noLista *prox;
};
```

O TAD Lista deve exportar e implementar as seguintes funções:

```
NoLista* lista_insere(NoLista * n, int a);
NoLista* lista_retira(NoLista * n, int a);
void lista_libera(NoLista * n);
void lista_imprime(NoLista * n);
NoLista* lista_acessa_prox(NoLista * n);
int lista_acessa_info(NoLista * n);
void lista_atualiza_info(NoLista * n, int a);
```

Além das funções normais, o TAD Lista também deve implementar e exportar a seguinte função:

> `lista_comparaListasInv` – A função recebe duas listas de inteiros e verifica se uma lista é o inverso da outra, isto é, se a segunda lista contém os mesmos valores inteiros da primeira, mas em ordem contrária. A função deve utilizar o TAD Pilha como estrutura temporária auxiliar. A função deve retornar 1 se uma lista é o inverso da outra e 0 caso contrário. Note que as listas recebidas pela função podem ter tamanhos diferentes! A função tem o seguinte protótipo:
>
> ```
> int comparaListasInv(NoLista *l1, NoLista *l2);
> ```

Após implementar as funções, crie o módulo principal do programa para testar as funções criadas.

3. Implemente um TAD Árvore para armazenar números inteiros com as seguintes funções:

```
NoArvore *arvore_criaVazia(void);
NoArvore *arvore_cria(int info, NoArvore *sae, NoArvore *sad);
int arvore_vazia(NoArvore *a);
void arvore_libera(NoArvore *a);
void arvore_imprime(NoArvore *a, int nivel);
```

Além das funções normais, o TAD Árvore também deve implementar e exportar as seguintes funções:

- `arvore_somaFolhas` – A função recebe uma árvore binária e retorna o a soma dos números inteiros armazenados em nós que são folhas da árvore (isto é, nós que não têm nenhum filho). Essa função tem o seguinte protótipo:

  ```
  int arvore_somaFolhas(NoArvore* a);
  ```

- `arvore_nivel` – Em uma árvore binária, o nível de um nó é definido como sendo igual à distância do caminho da raiz até o nó em questão. Assim, o nó raiz

está no nível 0, as raízes de suas sub-árvores no nível 1, as raízes das sub-árvores das sub-árvores no nível 2, e assim por diante. A função `arvore_nivel` recebe uma árvore binária e retorna o nível do nó cuja informação seja igual a um valor x dado. Se não existir um nó com o valor x, a função retornar o valor –1. Essa função tem o seguinte protótipo:

```
int arvore_nivel(NoArvore* a, int x);
```

- `arvore_maiorSoma` – Considerando todos os possíveis caminhos da raiz até cada folha de uma árvore binária, podemos calcular uma valor referente a soma de todas as informações armazenadas nos nós de cada um destes caminhos. A função `arvore_nivel` recebe uma árvore binária e retorna a maior soma encontrada dentre todos os possíveis caminhos de uma árvore. Essa função tem o seguinte protótipo:

```
int arvore_maiorSoma(NoArvore* a);
```

Após implementar as funções, crie o módulo principal do programa para testar as funções criadas.

4. Implemente um TAD Árvore Binária de Busca (ABB) para armazenar números inteiros. Os nós da ABB são definidos pela seguinte estrutura:

```
struct noABB {
  int info;
  struct noABB *esq;
  struct noABB *dir;
};
```

O TAD ABB deve implementar as seguintes funções:

- `abb_cria` – A função deve criar uma nova ABB vazia retornando NULL. Essa função tem o seguinte protótipo:

```
NoABB *abb_cria();
```

- `abb_insere` – A função deve criar e inserir um novo nó na ABB respeitando a ordenação por valor dos nós. Essa função tem o seguinte protótipo:

```
NoABB *abb_insere(NoABB *a, int valor);
```

- `abb_imprime` – A função de imprimir na tela todas os valores armazenados na ABB em ordem decrescente. Essa função tem o seguinte protótipo:

```
void abb_imprime(NoABB *a);
```

- `abb_libera` – a função deve liberar da memória todos os nós da ABB. Essa função tem o seguinte protótipo:

```
void abb_libera(NoABB *a);
```

Além das funções normais, o TAD ABB também deve implementar e exportar as seguintes funções:

- `abb_valida` – A função recebe uma árvore e verifica se ela é uma árvore binária de busca valida (isto é, se todos os nós estão armazenados ordenadamente).A função deve retornar 1 se a árvore for uma árvore binária de busca, ou 0 caso contrario. Essa função tem o seguinte protótipo:

```
int abb_valida(NoABB *a);
```

- `abb_balanceada` – A função recebe uma árvore e verifica se ela está balanceada (isto é, se para cada nó da árvore, a diferença entre as alturas das suas sub-árvores (direita e esquerda) não é maior do que um).A função deve retornar 1 se a árvore estiver balanceada, ou 0 caso contrario. Essa função tem o seguinte protótipo:

```
int abb_balanceada(NoABB *a);
```

Após implementar as funções, crie o módulo principal do programa para testar as funções criadas.