Universidade Federal de Goiás – UFG
Instituto de Informática – INF
Bacharelados (Núcleo Básico Comum)

Algoritmos e Estruturas de Dados 1 – 2022/2

Lista de Exercícios nº 03 – Tipo Abstrato de Dados (TAD)

Turmas: INF0286 – Prof. Wanderley de Souza Alencar
adaptado por Prof. Raphael Guedes

Sumário

Observações:

- A resolução de cada um dos exercícios desta lista pressupõe a utilização do conceito de *Tipo Abstrato de Dados* (TAD) durante a implementação utilizando a linguagem de programação $\mathbb{C}$;

1 Conjuntos de Números Naturais

 (++)

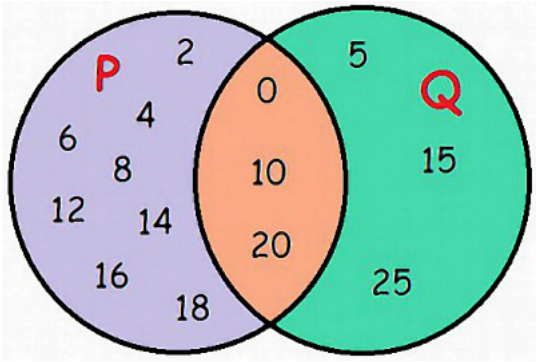

Figura 1: Imagem de dois conjuntos naturais P e Q.

A linguagem ℂ não possui um *tipo de dado* que seja capaz de representar a ideia de *conjunto finito* conforme a acepção Matemática do termo, ou seja, "*Uma coleção finita de elementos (entes ou componentes), na qual a ordem e a repetição destes elementos é irrelevante e, por isso, desconsiderada.*".
Escreva, em ℂ, um programa que seja capaz de representar um *conjunto de números naturais* por meio do uso do conceito de Tipo Abstrato de Dado (TAD).
O programa deve implentar, no mínimo, as seguintes operações (de 1 a 10)fundamentais:

1. criar um conjunto *C*, inicialmente *vazio*:
   `int criaConjunto(C);`
   retornando `SUCESSO` ou `FALHA`.
   A falha ocorre se não for possível, por alguma ocorrência, criar o conjunto *C*.

2. verificar se o conjunto *C* é vazio:
   `int conjuntoVazio(C);`
   retornando `TRUE` ou `FALSE`.

3. incluir o elemento *x* no conjunto *C*:
   `int insereElementoConjunto(x, C);`
   retornando `SUCESSO` ou `FALHA`.
   A falha acontece quando o elemento *x* já está presente no conjunto *C* ou, por algum outro motivo, a inserção não pode ser realizada.

4. excluir o elemento *x* do conjunto *C*:
   `int excluirElementoConjunto(x, C);`
   retornando `SUCESSO` ou `FALHA`.
   A falha acontece quando o elemento *x* não está presente no conjunto *C* ou, por algum outro motivo, a remoção não pode ser realizada.

5. calcular a cardinalidade do conjunto *C*:
   `int tamanhoConjunto(C);`
   retornando a quantidade de elementos em *C*. O valor 0 (zero) indica que o conjunto está vazio.

6. determinar a quantidade de elementos do conjunto *C* que são maiores que *x*:
   `int maior(x, C);`
   O valor 0 (zero) indica que todos os elementos de *C* são maiores que *x*.

7. determinar a quantidade de elementos do conjunto C que são menores que x:
   `int menor(x, C);`
   O valor 0 (zero) indica que todos os elementos de C são menores que x.

8. verificar se o elemento x pertence ao conjunto C:
   `int pertenceConjunto(x, C);`
   retornando `TRUE` ou `FALSE`.

9. comparar se dois conjuntos, C_1 e C_2 são idênticos:
   `int conjuntosIdenticos(C1, C2);`
   retornando `TRUE` ou `FALSE`.

10. identificar se o conjunto C_1 é subconjunto do conjunto C_2:
    `int subconjunto(C1, C2);`
    retornando `TRUE` ou `FALSE`.

Implementações Opcionais

11. gerar o complemento do conjunto C_1 em relação ao conjunto C_2:
    `Conjunto complemento(C1, C2);`
    retornando um *conjunto* que contém os elementos de C_2 que não pertencem a C_1.
    Se todos os elementos de C_2 estão em C_1, então deve retornar um conjunto vazio.

12. gerar a união do conjunto C_1 com o conjunto C_2:
    `Conjunto uniao(C1, C2);`
    retornando um *conjunto* que contém elementos que estão em C_1 ou em C_2.

13. gerar a intersecção do conjunto C_1 com o conjunto C_2:
    `Conjunto interseccao(C1, C2);`
    retornando um *conjunto* que contém elementos que estão em C_1 e, simultaneamente, em C_2.
    Se não houver elementos comuns deverá retornar um conjunto vazio.

14. gerar a diferença entre o conjunto C_1 e o conjunto C_2:
    `Conjunto diferenca(C1, C2);`
    retornando um *conjunto* que contém elementos de C_1 que não pertencem a C_2.
    Se todos os elementos de C_1 estão em C_2 deve retornar um conjunto vazio.

15. mostrar os elementos presentes no conjunto C:
    `void mostraConjunto(C, ordem);`
    Mostrar, no dispositivo de saída, os elementos de C.
    Se `ordem` for igual a CRESCENTE, os elementos de C devem ser mostrados em ordem crescente. Se ordem for igual a DECRESCENTE, os elementos de C devem ser mostrados em ordem decrescente.
    Observação: Como o dispositivo típico de saída é o monitor de vídeo, o(a) programador(a) tem liberdade para definir como os elementos serão dispostos nele. Por exemplo: dez ou vinte elementos por linha. Noutro exemplo: o programa definirá quantos elementos mostrar, por linha, de acordo com o número de elementos existentes no conjunto a ser apresentado.

16. copiar o conjunto C_1 para o conjunto C_2:
    `int copiarConjunto(C1, C2);`
    retornando `SUCESSO` ou `FALHA`.
    A falha acontece quando, por algum motivo, não é possível copiar os elementos do conjunto C_1 para o conjunto C_2.

17. destruir o conjunto *C*:
    `int destroiConjunto(C);`
    retornando `SUCESSO` ou `FALHA`.
    A falha acontece quando, por algum motivo, não é possível eliminar o conjunto *C* da memória.

Observações: Considere que:

- `SUCESSO` = 1; `FALHA` = 0;
- `TRUE` = 1; `FALSE` = 0;
- `CRESCENTE` = 1; `DECRESCENTE` = 0;
- os nomes das funções anteriormente apresentados no texto devem ser obedecidos, ou seja, o código-fonte $\mathbb{C}$ elaborado deverá obrigatoriamente utilizá-los. É claro que outras funções acessórias podem ser criadas livremente pelo(a) programador(a). **Contudo**, as variáveis criadas devem ter nomes representativos para seguir as boas práticas de desenvolvimento

Entradas e Saídas

Não serão fornecidas entradas/saídas para testes, pois o(a) estudante deverá apenas submeter o código-fonte por ele(a) elaborado no *Sharif Judge System* do INF/UFG.
O programa elaborado deverá ter um *menu* que permita ao usuário selecionar cada uma das operações supramencionadas, executá-la e, em seguida, retornar ao *menu* para escolher uma nova opção.
Para *finalizar o programa* o usuário deverá fornecer um entrada especial. Por exemplo, o número 0 (zero) como opção no *menu*.
O(A) estudante terá liberdade para escolher como implementar a funcionalidade de *menu*.