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

Algoritmos e Estruturas de Dados 1 – 2023/2

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

Turmas: INF0286 – Prof. Wanderley de Souza Alencar)

## 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 $\mathbb{C}$ ou $\mathbb{C}++$;
- Tendo em vista que, por característica de construção do ambiente *Sharif Judge System* do INF/UFG utilizado nesta disciplina, **apenas um único arquivo** pode ser enviado como proposta de solução para um problema, tal arquivo deverá ter a estrutura apresentada na figura a seguir;
- Detalhando: o único arquivo com a extensão `.c` deverá conter o que *deveria estar* em dois arquivos: o `tad.h` e o próprio `tad.c`;
- O uso do arquivo `tad.h` significa que a função `main()` elaborada como proposta de solução para o problema deve somente utilizar operações presentes neste arquivo.

```
#include <stdio.h>
#include <stdlib.h>

<TADs: conteúdo do(s) arquivo(s) .h>

<TADs: conteúdo do(s) arquivo(s) .c , sem #include "TAD.h">

int main () {
        <seu código>
}
```

Veja o exemplo a seguir:

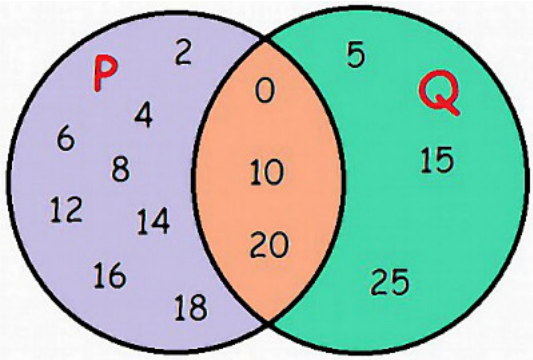
```
//=========================================================================
// Arquivo ponto.h
//=========================================================================
//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>

typedef struct ponto Ponto;
Ponto* ponto_cria(float x, float y, bool visibilidade);
void   ponto_libera(Ponto* p);
void   ponto_acessa(Ponto* p, float* x, float* y);
void   ponto_atribui(Ponto* p, float x, float y);
float  ponto_distancia(Ponto* p1, Ponto* p2);
void   ponto_oculta(Ponto* p);
void   ponto_mostra(Ponto* p);
void   ponto_move(Ponto* p, float x, float y);
//
//=========================================================================
// Arquivo ponto.c
//=========================================================================
//
struct ponto {
  float x;
  float y;
  bool visibilidade;
};
//
// Cria um ponto
//
Ponto* ponto_cria (float x, float y, bool visibilidade) {
   Ponto* p = (Ponto*) malloc(sizeof(Ponto));
   if (p != NULL) {
      p->x = x;
      p->y = y;
      p->visibilidade = visibilidade;
   }

   return (p);
}
//
// Libera (desaloca) um ponto...
//
void ponto_libera (Ponto* p) {
   if (p != NULL) {
      free(p);
   }
}
//
// Acessa um ponto, coletando suas coordenadas
//
void ponto_acessa (Ponto* p, float* x, float* y) {
   if (p != NULL) {
     *x=p->x;
     *y=p->y;
   }
}
//
// Atribui coordenadas a um ponto, modificando-o
//
void ponto_atribui (Ponto* p, float x, float y) {
   if (p != NULL) {
      p->x=x;
      p->y=y;
   }
}
//
// Retorna a distancia entre dois pontos
//
float ponto_distancia (Ponto* p1, Ponto* p2) {
   float dx = p1->x - p2->x;
   float dy = p1->y - p2->y;
   return (sqrt(dx*dx+dy*dy));
}
```

```
76
77 //
78 // Oculta (torna invisivel) o ponto
79 //
80 void ponto_oculta (Ponto* p) {
81
82    p->visibilidade = false;
83 }
84 //
85 // Mostra (torna visivel) o ponto
86 //
87 void ponto_mostra (Ponto* p) {
88
89    p->visibilidade = true;
90 }
91
92 void ponto_move(Ponto * p, float x, float y) {
93    //
94    // Codigo para movimentacao do ponto
95    //
96 }
97 //
98 // Corpo principal
99 //
100
101 int main(){
102    float xp,yp,xq,yq,d;
103    Ponto *p,*q;
104
105    printf("digite as coordenadas x e y para o ponto 1: ");
106    scanf("%f %f",&xp,&yp);
107    printf("digite as coordenadas x e y para o ponto 2: ");
108    scanf("%f %f",&xq,&yq);
109    p = ponto_cria(xp,yp, true);
110    q = ponto_cria(xq,yq, true);
111    d = ponto_distancia(p,q);
112    ponto_acessa(p,&xp,&yp); ponto_acessa(q,&xq,&yq);
113    printf("Distancia entre os pontos (%.2f,%.2f) e (%.2f,%.2f) = %.5f\n",xp,yp,xq,yq,d);
114    ponto_libera(p); ponto_libera(q);
115    return (0);
116 }
```

./programas/pontoCompleto.c




# 1 Conjuntos de Números Naturais

(+++) 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 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`.

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. gerar o conjunto das partes do conjunto $C$:
    `Conjunto conjuntoPartes(C);`

16. 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.

17. 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 $C_2$.

18. 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;
- qualquer conjunto poderá ter no máximo 1.000.000 (um milhão) de elementos, ou seja, esta é a *cardinalidade máxima* de um conjunto. Se qualquer operação resultar num conjunto com cardinalidade maior, então a função correspondente deverá retornar um *conjunto vazio* (se ela retorna um conjunto) ou `FALHA` (se ela retorna `SUCESSO` ou `FALHA`);
- a biblioteca `limits.h` da linguagem ℂ contém duas constantes para denotar quais são o *menor* e o *maior* `long int` que pode ser utilizado no ambiente computacional em que o programa está sendo elaborado. São elas: `LONG_MIN` e `LONG_MAX`. Elas deverão ser, respectivamente, o menor e o maior número que podem ser armazenados num conjunto qualquer do programa;
- os nomes das funções anteriormente apresentados no texto devem ser obedecidos, ou seja, o código-fonte ℂ elaborado deverá obrigatoriamente utilizá-los. É claro que outras funções acessórias podem ser criadas livremente pelo(a) programador(a).

#### 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*.


## 2 Manipulando Datas

 (+++)

É inquestionário que a capacidade de *manipular datas* é de extrema importância em muitas aplicações práticas na área de processamento de dados. Infelizmente nem sempre há, numa determinada linguagem de programação que se está utilizando para o desenvolvimento de aplicações, uma *biblioteca* com variadas funções para realizar a manipulação de datas.

Considere que você está participando do desenvolvimento de uma *biblioteca* para esta finalidade, sendo que ela deverá ser integralmente escrita em $\mathbb{C}$ e conter pelo menos as seguintes funções, expressas por seus cabeçalhos: `data.h`.

1. `Data * criaData (unsigned int dia, unsigned int mes, unsigned int ano);`
   Cria, de maneira dinâmica, uma *data* a partir dos valores para dia, mês e ano fornecidos.

2. `Data * copiaData (Data d);`
   Cria uma *cópia* da data `d`, retornando-a.

3. `void liberaData (Data * d);`
   Destrói a data indicada por `d`.

4. `Data * somaDiasData (Data d, unsigned int dias);`
   Retorna uma data que é `dias` dias posteriores à data `d`.
   Por exemplo, fornecendo a data `d = 16/03/2020` e `dias = 5`, retornará a data `21/03/2020`.

5. `Data * subtrairDiasData (Data d, unsigned int dias);`
   Retorna uma data que é `dias` dias anteriores à data `d`.
   Por exemplo, fornecendo a data `d = 16/03/2020` e `dias = 15`, retornará a data `01/04/2020`.

6. `void atribuirData (Data * d, unsigned int dia, unsigned int mes, unsigned int ano);`
   Atribui, à data `d`, a data `dia/mes/ano` especificada.
   Se não for possível, então faz com que `d` seja alterada para `NULL`.

7. `unsigned int obtemDiaData (Data d);`
   Retorna a componente `dia` da data `d`.

8. `unsigned int obtemMesData (Data d);`
   Retorna a componente `mes` da data `d`.

9. `unsigned int obtemAnoData (Data d);`
   Retorna a componente `ano` da data `d`.

10. `unsigned int bissextoData (Data d);`
    Retorna `TRUE` se a data pertence a um ano bissexto. Do contrário, retorna `FALSE`.

11. `int comparaData (Data d1, Data d2);`
    Retorna MENOR se `d1` < `d2`, retorna IGUAL se `d1` = `d2` ou retorna MAIOR, se `d1` > `d2`.

12. `unsigned int numeroDiasDatas (Data d1, Data d2);`
    Retorna o número de dias que existe entre as datas `d1` e `d2`.
    Se `d1` = `d2`, então o número de dias é igual a 0 (zero). Do contrário, será um número estritamente positivo.

13. `unsigned int numeroMesesDatas (Data d1, Data d2);`
    Se `d1` e `d2` estão no mesmo mês/ano, então o número de meses é igual a 0 (zero). Do contrário, será um número estritamente positivo.

14. `unsigned int numeroAnosDatas (Data d1, Data d2);`
    Se `d1` e `d2` estão no mesmo ano, então o número de anos é igual a 0 (zero). Do contrário, será um número estritamente positivo.

15. `unsigned int obtemDiaSemanaData (Data d);`
    Retorna o *dia da semana* correspodente à data `d`.
    Considerando que `DOMINGO` = 1; `SEGUNDA-FEIRA` = 2; ... ; `SÁBADO` = 7.

16. `char * imprimeData (Data d, char * formato);`
    Retorna uma *string* com a data "*formatada*" de acordo com o especificado em `formato`.
    Se `formato` = "ddmmaaaa", então a *string* retornada deverá apresentar os dois dígitos do dia, os dois dígitos do mês e os quatro dígitos do ano, nesta ordem, e separados por uma (/ – barra). Por exemplo: "12/11/2019".
    Se `formato` = "aaaammdd", então a *string* retornada deverá apresentar os quatro dígitos do ano, os dois dígitos do mês e os dois dígitos do dia, nesta ordem, e separados por uma (/ – barra).
    Por exemplo: "2019/11/12".
    De maneira análoga, são válidas as seguintes *strings* de formatação:

    - "aaaa";
    - "mm";
    - "dd";
    - "ddmm".

**Entrada 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*.

**Observações**

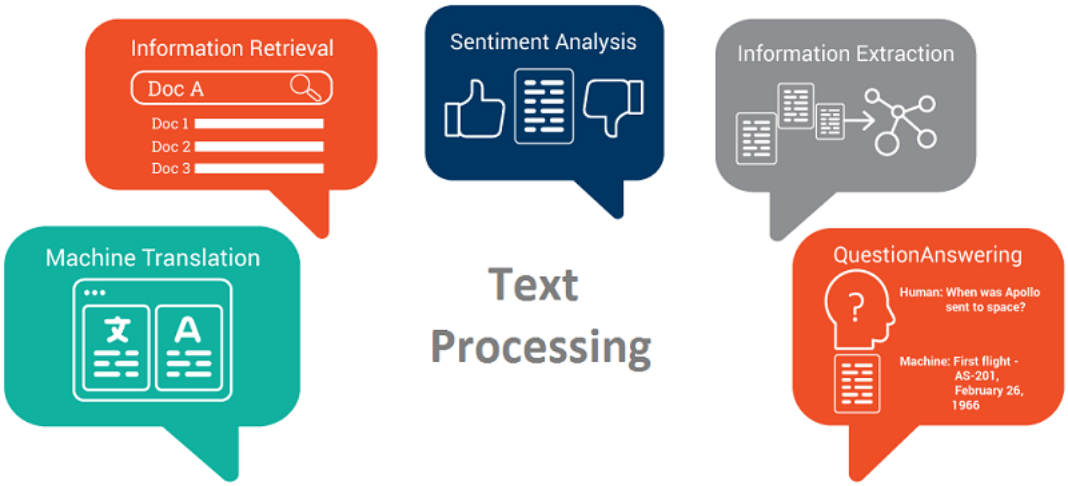
1. Uma `data` é formada por seu `dia`, `mes` e `ano`;
2. Considere que as datas a serem aplicadas ao sistema serão, sempre, no intervalo de 01/01/1900 a 31/12/2200;
3. Fique atento a um importante evento que ocorreu no mês de outubro de 1582 envolvendo o calendário Gregoriano – pesquise sobre isto antes de implementar todas as funções. Em partircular a função `obtemDiaSemanaData (Data d);`
4. `TRUE` = 1; `FALSE` = 0;
5. A função `comparaData (Data d1, Data d2)` deve retornar:

   **`MENOR`** quando d1 < d2;

   **`IGUAL`** quando d1 = d2;

   **`MAIOR`** quando d1 > d2.

   com `MENOR` = -1; `IGUAL` = 0 e `MAIOR` = 1.




# 3 Processamento de Textos

 (+++)

Vamos, agora, elaborar um TAD que seja capaz, de maneira simples, representar um *texto* e disponibilizar diversas funções que sejam capazes de manipulá-lo.
Neste contexto, um *texto* é concebido como sendo "uma sequência de caracteres sem nenhuma formatação especial, ou seja, não existe **negrito**, *itálico* ou qualquer outro atributo especial aplicado sobre os caracteres que formam o texto: há simplesmente o caractere. Entretanto, ele pode ser uma letra maiúscula ou minúscula, um dígito ou um símbolo especial.
Considere que as seguintes operações estão previstas para estarem presentes no TAD `texto.h`:

1. `Texto * criarTexto (char * t);`
2. `void liberarTexto (Texto * t);`
3. `unsigned int tamanhoTexto (Texto * t);`
4. `char * obterTexto (Texto * t;`
5. `void mostrarTexto (Texto *t, unsigned int colunas).`
6. `Texto * copiarTexto (Texto * t);`
7. `void substituirTexto (Texto * t, char * alteracao);`
8. `Texto * concatenarTextos (Texto * t1, Texto * t2);`
9. `char * obterSubtexto (Texto * t, unsigned int inicio, unsigned int tama`
10. `unsigned int encontrarSubtexto (Texto * t, char * subtexto, unsigned int ocorrencia);`
11. `int compararTextos (Texto * t1, Texto * t2).`

**Entrada 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*.

## Observações

1. O procedimento `substituirTexto` substitui o texto presente em `t` pelo texto recebido em `alteracao`, mesmo que eles tenham tamanhos diferentes;

2. A função `obterTexto` deverá retornar uma cadeia de caracteres com o texto armazenado;

3. A função `compararTextos(Texto * t1, Texto * t2)` deve retornar:

   **`MENOR`** quando t1 < t2;

   **`IGUAL`** quando t1 = t2;

   **`MAIOR`** quando t1 > t2.

   com `MENOR` = -1; `IGUAL` = 0 e `MAIOR` = 1.

4. A função `obterSubtexto` deverá retornar uma cadeia de caracteres que se inicia na `inicio`-ésima posição do texto `t` e conter `tamanho` caracteres de extensão.
   A primeira posição do texto `t` é a de número 1 (um).
   Se, a partir da posição `inicio` não for possível obter os `tamanho` caracteres solicitados, a função deverá retornar uma cadeia que se inicia na posição `inicio` do texto `t` e conter até seu último caractere.
   Se a posição `inicio` for inválida, ou seja, menor que 1 (um) ou maior que o tamanho do texto `t`, a função deve retornar uma cadeia *nula*.

5. A função `encontrarSubtexto` deve *procurar* pela ocorrência de número `ocorrência` (1ª, 2ª, 3ª, ..., nª) do `subtexto` em `t`.
   Se encontrar esta ocorrência, deverá retornar a posição do *primeiro caractere* dela em `t`. Do contrário, deverá retornar 0 (zero), pois o primeiro caractere de `t` é considerado como sendo o de número 1 (um);

6. A função `mostrarTexto` deve apresentar `t` no dispositivo de saída do computador (normalmente o monitor de vídeo), de tal maneira que a cada linha do dispositivo de saída sejam apresentados `colunas` caracteres. Por consequência, o texto `t` poderá ocupar uma ou mais linhas em função de seu tamanho total.


# 4 Mundo das Bactérias

 (+++++)

Um estudante do curso de Bacharelado em Ciências Biológicas precisa, durante o seu Trabalho de Conclusão de Curso (TCC), fazer uma sequência de experimentos com uma "*cultura de bactérias*" para comprovar hipóteses a respeito da *movimentação* de cada uma destas bactérias nesta cultura.
O problema do estudante é que para fazer experimentos no "*mundo real*" das bactérias, consome-se muito tempo e, sabendo que você é estudante de um dos cursos de bacharelado do INF/UFG, pediu que você elaborasse um programa de computador que tornasse possível fazer uma *simulação* de culturas de bactérias, o que acelerará a obtenção dos resultados desejados.
Você gostou do desafio e passou algumas horas conversando com o estudante e, ao final, imaginou que uma "*cultura de bactérias*" pode ser representada, computacionalmente, por meio de uma matriz bidimensional $W$ (de *world*) que contém um par de números naturais $(x_i, y_i)$, com $n$ linhas e $m$ colunas. Você considerou que $n, m \in \mathbb{N}$ e $1 \leq n, m \leq 1000$, ou seja, que haverá no máximo 1.000.000 de bactérias numa cultura.
Cada *bactéria* é representada por um par $(x_i, y_i)$, onde:

$x_i$ é a identificação daquela *bactéria*, ou seja, o número associado a ela – que é chamada de *ordem* da bactéria. Sabe-se que $1 \leq x_i \leq (n \cdot m)$;

$y_i$ é a *força* da bactéria em relação às demais bactérias de sua cultura ou de outras culturas.
A *força* é expressa por um número que varia de 1 a 100, inclusive extremos, sendo que a força máxima é 100 (cem) e a mínima é 1 (um).

$z_i$ é a *expectativa de vida* da bactéria, ou seja, o tempo, expresso em número de *épocas*, que a bactéria viverá se não houver uma intercorrência durante sua vida. Você apenas precisará utilizar este conceito se for implementar a porção "*opcional*" desta questão: veja a seção específica ao final deste texto.

Como você está, neste momento, estudando os TADs, se propôs elaborar o programa solicitado utilizando este conceito e a linguagem de programação ℂ, para tornar a aventura mais "*hard*".
Depois de alguns rascunhos, você chegou à conclusão de que as seguintes operações devem ser disponibilizadas para o estudante de Ciências Biológicas por seu programa:

[01] `World * newWorld (unsigned int n, unsigned int m);`

*Cria* uma nova cultura de bactérias, com *n* linhas e *m* colunas de tamanho.
A cultura deverá, após a operação, ficar vazia, ou seja, não conterá nenhuma bactéria.

[02] `World * cloneWorld (World * w):`
Faz a cópia da cultura de bactérias presente em `w`, gerando um *clone* dela, mesmo que esta esteja vazia.

[03] `void freeWorld (World * w):`
Se a cultura de bactérias `w` existe, a destrói. Do contrário ignora a solicitação.

[04] `unsigned int randomWorld (World * w, unsigned int n):`
Insere, em posições aleatórias, livres e distintas da cultura `w`, `n` bactérias.
A ordem a ser utilizada para as bactérias a serem adicionadas se inicia no número natural seguinte ao associado à bactéria que possuir o *maior ordem* na cultura `w`.
A *força* de cada bactéria deverá ser gerada aleatoriamente na faixa de valores permitida, o mesmo acontecendo com o tempo de *vida* de cada bactéria.
Se não for possível inserir as `n` bactérias em `w`, a função deverá retornar `FALHA`. O sucesso será indicado restornando `SUCESSO`.
**Observação**: Se `w` estiver inicialmente vazia, então a ordem das bactérias deve ser iniciada em 1 (um) e, por consequência, terminar em `n`.

[05] `unsigned int addBacterium (World * w,`
`unsigned int n, unsigned int f, unsigned int e):`
Insere, numa posição livre aleatoriamente escolhida da cultura `w`, a bactéria cuja ordem é dada pelo número `n`, cuja *força* é dada por `f` e cuja *expectativa de vida* seja expressa por `e`.
Se a bactéria de ordem `n` já está na cultura ou se a cultura não possui espaço para nenhuma bactéria adicional, a função deve retornar `FALHA`. Na hipótese de sucesso, a função deve retornar `SUCESSO`;

[06] `unsigned int addBacteriumXY (World * w, unsigned int n,`
`unsigned int x, unsigned int y, unsigned int f, unsigned int e):`
Insere, na linha `x` e coluna `y` da cultura `w`, a bactéria cuja ordem é dada pelo número `n`, cuja *força* é `f` e cuja *expectativa de vida* seja expressa por `e`.
Se a bactéria de ordem `n` já está na cultura ou se a cultura não possui espaço para nenhuma bactéria adicional ou se a posição `(x,y)` estiver ocupada ou, ainda, se a expectativa `e` for inválida, a função deve retornar `FALHA`. Na hipótese de sucesso, a função deve retornar `SUCESSO`.

[07] `unsigned int killBacterium (World * w, unsigned int n):`
Mata (destrói) a bactéria cuja ordem é dada pelo número `n`, independentemente de sua localização na cultura.
Se a bactéria de ordem `n` não estiver na cultura, a função deve retornar `FALHA`. Na hipótese de sucesso, a função deve retornar `SUCESSO`.

[08] `unsigned int killBacteriumXY (World * w, unsigned int x, unsigned int y):`
Mata (destrói) a bactéria que está na linha `x` e coluna `y` da cultura `w`.
Se não há bactéria na posição `(x,y)`, a função deve retornar `FALHA`. Na hipótese de sucesso, a função deve retornar `SUCESSO`.

[09] `World * jointWorlds (World * w1, World *w2):`
Realiza a *união* das culturas de bactérias `w1` e `w2`, gerando uma nova cultura.
A operação de união deve ser consensual, ou seja, não pode haver *colisão* entre as posições ocupadas por

nenhuma das bactérias proveniente das culturas originais: é uma *união* pacífica.
Se houver colisão, a função deverá retornar `NULL` para indicar que a união não é possível. Do contrário, retorna a *nova cultura* gerada é retornada.
**Observação**: Lembre-se de que a cultura gerada deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura gerada deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

[10] `World * warWorlds (World * w1, World *w2):`
Coloca as culturas de bactérias `w1` e `w2` em *guerra*, gerando uma nova cultura com as bactérias sobreviventes de acordo com as seguintes regras:

1. se houver duas bactérias que deveriam, na nova cultura, ocupar a mesma posição, a bactéria *mais forte* fagocita a bactéria *mais fraca* e ocupa aquele lugar na nova cultura. Além disso, sua *nova força* é dada pela soma da sua própria *força* e a da bactéria que acabou de fagocitar. Se houver *empate* entre as forças de ambas bactérias originais, escolha aquela proveniente da cultura `w1` para ir para a nova cultura, com *força dobrada*. A *expectativa de vida* da bactéria resultante é a maior expectativa de vida dentre as bactérias originais.
   Se a *força* da bactéria vencedora superar a força máxima estabelecida, a ela deverá ser atribuída a *força máxima*.

2. se não houver disputa, a bactéria deverá permanecer, na nova cultura, na mesma posição em que está na cultura de origem, com sua mesma *força* e *expectativa de vida* originais.

**Observação**: Lembre-se de que a cultura gerada deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura `w` deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

[11] `World * probabilisticWarWorlds (World * w1, World *w2, float p):`
Coloca as culturas de bactérias `w1` e `w2` em guerra, gerando uma nova cultura com as bactérias sobreviventes de acordo com as seguintes regras:

1. se houver duas bactérias que deveriam, na nova cultura, ocupar a mesma posição, a bactéria *mais forte* fagocita a bactéria *mais fraca* de acordo com a probabilidade `p`, sabendo-se que $0 < p \leq 1$, e ocupa aquele lugar na nova cultura. Do contrário, a bactéria *mais fraca* é que fagocita a bactéria *mais forte* e ocupa aquele lugar na nova cultura.
   Além disso, a *nova força* da bactéria vencedora é adicionada à da bactéria derrotada. Se a *força* da bactéria vencedora superar a força máxima estabelecida, a ela deverá ser atribuída a *força máxima*. A *expectativa de vida* da bactéria resultante é a maior expectativa de vida dentre as bactérias originais.

2. se não houver disputa, a bactéria deverá permanecer, na nova cultura, na mesma posição em que está na cultura de origem, com sua mesma *força* e *expectativa de vida* originais.

**Observação**: Lembre-se de que a cultura gerada deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura `w` deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

[12] `unsigned int sizeWorld (World * w):`
Retorna o número de bactérias existentes na cultura `w`.
O valor 0 (zero) indicará que a cultura está vazia.

[13] `unsigned int forceWorld (World * w):`
Retorna a soma de todas as *forças* das bactérias existentes na cultura `w`.
O valor 0 (zero) indicará que a cultura está vazia.

[14] `unsigned int showWorld (World * w):`
Apresenta, no dispositivo de saída (normalmente o monitor de vídeo), uma "imagem" matricial da cultura `w` de acordo com seu tamanho (expressso por $n \cdot m$).
**Observação**: A proposta aqui é que você elabore, livremente, uma maneira de *mostrar* o que está presente na cultura `w` no dispositivo de saída.
Por exemplo, mesmo no modo texto, é possível construir um mapa como o representado abaixo, para uma cultura de tamanho $5 \cdot 5$:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 6,1 | 2,100 | 5,70 | | |
| 2 | | | | 4,32 | 3,3 |
| 3 | 7,4 | | 8,3 | | |
| 4 | | 1,80 | | | 10,80 |
| 5 | | | | 9,45 | |

O conteúdo da linha 1, coluna 1, indica que nesta célula está a bactéria de ordem `6` e cuja *força* é igual a `1`. Na posição (2,4), portanto, está a bactéria de ordem `4` e de *32*.
Apesar do exemplo, você estará livre para criar outras maneiras de mostrar a cultura de bactérias: solte sua imaginação para concebê-la, mesmo que isso faça com que você utilize a "*parte gráfica*" da linguagem.
**Observação**: Se a cultura `w` for *muito grande*, ou seja, a dimensão $n \cdot m$ não cabe inteiramente no dispositivo de saída, você poderá fazer com que o usuário escolha uma *porção* da cultura a ser visualizada. Isto pode ser feito fazendo com que ele selecione a porção das linhas e das colunas a serem mostradas.
Por exemplo, informando (1,10) e (6,14) significa que ele deseja visualizar as linhas de 1 a 10 e as colunas de 6 a 14. Evidentemente esta *porção* a ser visualizada não poderá superar a possibilidade de apresentação do dispositivo de saída. Se superar, avise-o da incorreção e peça para que ele selecione uma área menor.

# IMPLEMENTAÇÃO OPCIONAL

A implementação das funções a seguir é **opcional**, pois elas são um *experimento* contínuo com o que acontecerá durante o processo de sucessivas interações entre duas culturas de bactérias.
Para aqueles(as) que gostam que "*quebra-cabeças*", este é um bom exercício.
[15] `World * continumWarWorlds (World * w1, World *w2):`
Coloca as culturas de bactérias `w1` e `w2` em *guerra contínua*, gerando uma sequência de novas culturas com as bactérias sobreviventes que, novamente, voltam à *guerra*.
Na *guerra contínua* é necessário entender o conceito de *época*: ele é um relógio que será iniciado em 0 (zero) marcando o número de batalhas já realizadas entre culturas de bactérias. Assim, este relógio contará continuamente: 0, 1, 2, 3, 4, ...

Todas as bactérias são consideradas "*nascidas*" na *época* 0 (zero). Assim, se uma bactéria tiver *expectativa vida* igual a 50, significa que ela poderá participar de até 50 batalhas, pois a cada batalha vencida sua *expectativa de vida* é diminuída de uma unidade. Quando a expectativa chegar a 0 (zero), aquela bactéria morre naturalmente, ou seja, sem participar de uma batalha.
É claro que uma bactéria poderá morrer antes do término de sua *expectativa de vida* por ter perdido uma batalha com outra bactéria (mais forte ou mais fraca se, por exemplo, se a disputa for a probabilística).
As regras para uma *batalha* desta guerra são as seguintes:

1. verifique quais as bactérias atingiram sua *expectativa de vida* e as elimine de ambas culturas – `w1` e `w2` – antes da batalha começar.

2. se houver duas bactérias que deveriam, na nova cultura, ocupar a mesma posição, a bactéria *mais forte* fagocita a bactéria *mais fraca* e ocupa aquele lugar na nova cultura. Além disso, sua *nova força* é adicionada à força da bactéria que acabou de fagocitar. Se houver *empate*, escolha a originária da cultura `w1` para ir para a nova cultura, com *força dobrada*. A *expectativa de vida* da bactéria resultante é a maior expectativa de vida dentre as bactérias originais.
   Se a *força* da bactéria vencedora superar a força máxima estabelecida, a ela deverá ser atribuída a *força máxima*.

3. se não houver disputa, a bactéria deverá permanecer, na nova cultura, na mesma posição em que está na cultura de origem, com sua mesma *força* e *expectativa de vida* originais.

Ao terminar uma batalha, a nova cultura (ou seja, a gerada pela batalha das culturas `w1` e `w2`. Vamos chamá-la de `w`.) irá travar uma nova batalha com `w1` ou `w2` originais.

Quem irá para a batalha com `w`?
Aquela que tiver a maior soma das forças de suas bactérias! (Lembre-se que a função:
`forceWorld (World * w)`
lhe fornecerá esta informação).
Assim, a cultura `w` tomará o lugar `w1` na nova batalha e a cultura *mais forte* entre `w1` e `w2` originais tomará o lugar de `w2` nesta nova batalha.

Esta guerra terá fim?
Esta é uma ótima questão para experimentação:

uma guerra terminará depois de uma sequência de `DuracaoGuerra` batalhas ou, se por algum motivo, haverá uma *estabilização* do processo de guerrilha que, por consequência, se tornará uma "Guerra Infinita"?

O valor de `DuracaoGuerra` será fixado por você durante a experimentação: 10, 20, 30, 50, ... 1000.

O que é uma *estabilização* da guerra?
Vamos considerar que a guerra ficou *estável* se após uma sequência de `DuracaoEstabilizacao` batalhas, não há mudança no valor retornado pela função `forceWorld` quando esta é aplicada na cultura vencedora da batalha.
Por exemplo: Depois de uma sequência de 10 batalhas (que seria o valor fixado para `DuracaoEstabilizacao`) a cultura vencedora está sempre com o mesmo valor de `forceWorld` obtido.
O valor `DuracaoEstabilizacao` terá que ser fixado para ser menor que o valor de `DuracaoGuerra` corrente.

Ao terminar a função deve retornar `NULL` se houve algum problema que a impediu de continuar até completar. Do contrário, deve retornar a cultura vencedora da guerra contínua.

**Observação**: Lembre-se de que a cultura gerada em cada batalha deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura `w` deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

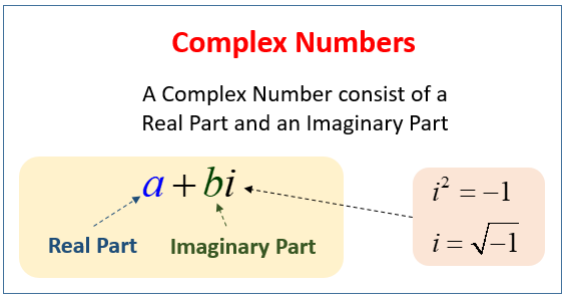
[16] `World * continumProbabilisticWarWorlds (World * w1, World *w2, float p)`:
É a versão probabilística da função:
`World * continumWarWorlds (World * w1, World *w2)`,
ou seja, ela utiliza as regras estabelecida por esta função para a guerra contínua, mas cada batalha entre `w1` e `w2` é realizada de acordo com as regras de probabilidade fixada em
`World * probabilisticWarWorlds (World * w1, World *w2, float p)`.

#### 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*.

#### Observações

- `SUCESSO` = 1; `FALHA` = 0;
- `TRUE` = 1; `FALSE` = 0;
- `DuracaoGuerra` varia entre 1 e 1000, inclusive extremos;
- `DuracaoEstabilizacao` varia entre 1 e o valor de `DuracaoGuerra`, inclusive extremos;
- a *expectativa de vida* de uma bactéria varia de 1 a 1000, inclusive extremos.




# 5 Números Complexos

 (+++)

Na ciência Matemática, um *número complexo* é um número expresso na forma $a+b\cdot i$, onde $a$ e $b$ são números reais ($a,b \in \Re$) e $i$ é um símbolo especial chamado de *unidade imaginária*, que satisfaz à equação $i^2 = -1$. Como nenhum númeor *real* é capaz de satisfazer a esta equação, o número que a satisfaz foi chamado, por René Decartes (1596–1650), de "*número imaginário*" e, atualmente, é representado pela letra $i$.

Para o número complexo $a+b\cdot i$, $a$ é chamado de *parte real* do número, e $b$ de *parte imaginária* do número. O conjunto dos números complexos é denotado por $\mathbb{C}$ o que, em nosso caso, o torna idêntico à grafia utilizada para denotar a linguagem de programação *C*... mera coincidência.

Apesar de sua origem a nomenclatura de *imaginário*, os números complexos se mostram extremamente importantes para a Matemática e para diversas outras ciências, como a Física e a Engenharia, por exemplo. Eles nos auxiliam na resolução de muitos problemas do mundo natural que, sem eles, não teriam solução. Por exemplo: eles permitem encontrar uma solução para qualquer equação polinomial dada, mesmo para aquelas que não possuem solução real, comom, por exemplo, a equação $x^2 - x + 1 = 0$, cujas soluções são:

$$x_1 = \frac{1+\sqrt{3}\cdot i}{2}$$

e

$$x_2 = \frac{1-\sqrt{3}\cdot i}{2}$$

Outro exemplo é $(x+1)^2 = -9$, cujas soluções são $x_1 = -1+3\cdot i$ e $x_2 = -1-3\cdot i$.

O que se deseja é que você, de maneira análoga ao que foi realizado na uestão nº 1 desta lista, elabore, utilizando a linguagem $\mathbb{C}$ um programa que seja capaz de representar um *conjunto de números complexos* por meio do uso do conceito de Tipo Abstrato de Dado (TAD).
O programa deve implentar, no mínimo, as seguintes operações 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. verificar se o elemento $x$ pertence ao conjunto $C$:
   `int pertenceConjunto(x, C);`
   retornando `TRUE` ou `FALSE`.

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

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

9. 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.

10. 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 $C_2$.

11. 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.

12. 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.

13. 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, primeiramente de sua "*parte real*" e, havendo empate, utilizar a "*parte imaginária*". Se ordem for igual a DECRESCENTE, os elementos de $C$ devem ser mostrados em ordem decrescente, primeiramente de sua "*parte real*" e, havendo empate, utilizar a "*parte imaginária*".

**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.

14. 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 $C_2$.

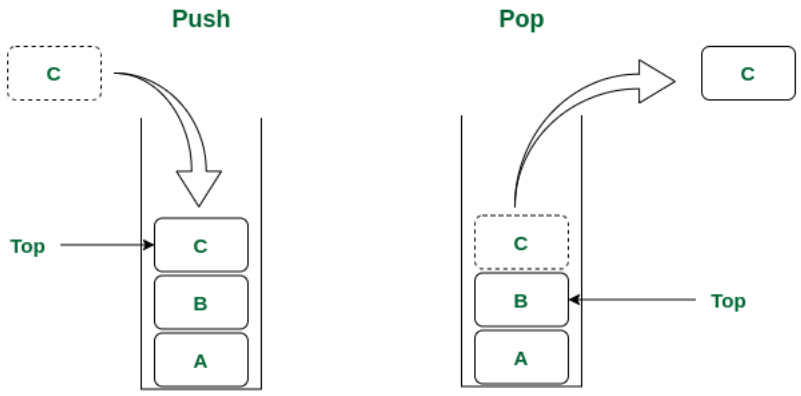
15. destruir o conjunto $C$:
    `int destroiConjunto(C);`
    retornando `SUCESSO` ou `FALHA`.
    A falha acontece quando, por algum motivo, não é possível eliminar o conjunto $C$ da memória.

**Observações**: Considere que:

- `SUCESSO` = 1; `FALHA` = 0;
- `TRUE` = 1; `FALSE` = 0;
- `CRESCENTE` = 1; `DECRESCENTE` = 0;
- qualquer conjunto poderá ter no máximo 1.000.000 (um milhão) de elementos, ou seja, esta é a *cardinalidade máxima* de um conjunto. Se qualquer operação resultar num conjunto com cardinalidade maior, então a função correspondente deverá retornar um *conjunto vazio* (se ela retorna um conjunto) ou `FALHA` (se ela retorna `SUCESSO` ou `FALHA`);
- a biblioteca `limits.h` da linguagem $\mathbb{C}$ contém duas constantes para denotar quais são o *menor* e o *maior* `long int` que pode ser utilizado no ambiente computacional em que o programa está sendo elaborado. São elas: `LONG_MIN` e `LONG_MAX`. Elas deverão ser, respectivamente, o menor e o maior número que podem ser armazenados num conjunto qualquer do programa;
- 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).

#### 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*.




Stack Data Structure

# 6 Pilhas

 (+++)

Uma importante estrutura de dado é denominada de *pilha*.

De maneira simples, pode-se dizer que uma *pilha* é uma coleção de dados em que se impõe a seguinte regra para a realização das operações de inserção e remoção: O último elemento a entrar para a coleção é primeiro a sair da coleção; o penúltimo elemento a entrar é o segundo elemento a sair da coleção; o antepenúltimo elemento a entrar é terceiro a sair da coleção e, assim, sucessivamente, de tal maneira que o primeiro elemento a entrar é o último elemento a sair da coleção.

A operação de inserir um elemento na *pilha* é denominada de **`push`** e a de remover um elemento é chamada de **`pop`**.

Além dessas operações fundamentais, devem ser imaginadas operações:

1. criar uma pilha $P$, inicialmente *vazia*:
   `int create(P);`
   retornando `SUCESSO` ou `FALHA`.
   A falha ocorre se não for possível, por alguma ocorrência, criar a pilha $P$.

2. verificar se a pilha $P$ é vazia:
   `int isEmpty(P);`
   retornando `TRUE` ou `FALSE`.

3. incluir o elemento $x$[1] na pilha $P$:
   `int push(P, x);`
   retornando `SUCESSO` ou `FALHA`.
   A falha acontece quando, por algum outro motivo, a inserção não pode ser realizada.

4. remover um elemento da pilha $P$:
   `unsigned long int pop(P);`
   retornando `SUCESSO` ou `FALHA`.

[1] Considere que os elementos que formam a pilha são números inteiros positivos e *longos* de acordo com a nomenclatura da linguagem $\mathbb{C}$.

A falha acontece quando a pilha está vazia ou por algum outro motivo, a remoção não pode ser realizada.

5. calcular a cardinalidade da pilha $P$:
   `long int size(P);`
   retornando a quantidade de elementos em $P$. O valor 0 (zero) indica que a pilha está vazia.

6. comparar se duas pilhas, $P_1$ e $P_2$, são idênticas:
   `int isEqual(P1, P2);`
   retornando `TRUE` ou `FALSE`.

7. mostrar os elementos presentes na pilha $P$, na ordem especificada.
   `void show(P, ordem);`
   Mostrar, no dispositivo de saída, os elementos de $P$.
   Se `ordem` for igual a TOPO, os elementos de $P$ devem ser mostrados do *topo* para a *base*. Se ordem for igual a BASE, os elementos de $P$ devem ser mostrados da *base* para o *top*.
   **Observação**: Como o dispositivo típico de saída é o monitor de vídeo, o(a) programador(a) deve fazer com que um elemento seja exibido por linha.

8. copiar a pilha $P_1$ para a pilha $P_2$:
   `int copy(P1, P2);`
   retornando `SUCESSO` ou `FALHA`.
   A falha acontece quando, por algum motivo, não é possível copiar os elementos da pilha $P_1$ para a pilha $P_2$.

**Observações**: Considere que:

- `SUCESSO` = 1; `FALHA` = 0;
- `TRUE` = 1; `FALSE` = 0;
- `TOP` = 1; `BASE` = 0;
- qualquer pilha poderá ter no máximo 1.000.000 (um milhão) de elementos, ou seja, esta é a *cardinalidade máxima* de uma pilha. Se qualquer operação resultar numa pilha com cardinalidade maior, então a função correspondente deverá retornar uma *pilha vazia* (se ela retorna uma pilha) ou `FALHA` (se ela retorna `SUCESSO` ou `FALHA`);
- a biblioteca `limits.h` da linguagem $\mathbb{C}$ contém duas constantes para denotar quais são o *menor* e o *maior* `long int` que pode ser utilizado no ambiente computacional em que o programa está sendo elaborado. São elas: `LONG_MIN` e `LONG_MAX`. Elas deverão ser, respectivamente, o menor e o maior número que podem ser armazenados num conjunto qualquer do programa;
- 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).

#### 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*.