```
main()
{
   int n1, n2;
   char op;
   RACIONAL r1, r2, r3;
   printf ("A operacao: \n");
   scanf ("%d/%d %c ", &n1, &n2, &op);
   criar_racional (n1, n2, &r1);
   scanf ("%d/%d", &n1, &n2);
   criar_racional (n1, n2, &r2);
   switch (op)
  {
     case '+':  somar_racionais(&r1, &r2, &r3);
          break;
     case '*':  multiplicar_racionais(&r1, &r2, &r3);
          break;
     case '=':  printf ("%s",equivalencia_racionais(&r1,
    &r2)?"iguais":"diferentes");
          exit();
  }
   printf ("= %d/%d", r3.num, r3.den);
}
```

Tipos Abstratos de Dados

A forma apresentada para manipulação de racionais é satisfatória?

Não. Devido ao limite de representatividade dos inteiros. Os racionais deveriam ser representados em seus números mínimos.

Uma rotina conhecida como algoritmo de Euclides pode ser usada para reduzir qualquer fração da forma numerador/ denominador a seus termos mínimos.

UNIVASF

Tipos Abstratos de Dados

Essa rotina pode ser descrita como:

1. Seja a o maior entre o numerador e o denominador e b o menor.
2. Divida a por b, encontrando um quociente q e um resto r (isto é, $a = q * b + r$).
3. Defina $a = b$ e $b = r$.
4. Repita os passos 2 e 3 até que b seja igual a 0.
5. Divida tanto o numerador quanto o denominador pelo valor de a.

Implemente esta rotina em C.

```
void reduzir_racional (RACIONAL *inrat, RACIONAL *outrat)
{
  int a, b, rem;
  if (inrat->num > inrat->den)
  {
    a = inrat->num;
    b = inrat->den;
  }
  else
  {
    a = inrat->den;
    b = inrat->num;
  }
  while (b)
  {
    rem = a%b;
    a = b;
    b = rem;
  }
  outrat->num = inrat->num/a;
  outrat->den = inrat->den/a;
}
```

Vetores

Um vetor é um exemplo de uma estrutura de dados trivial.

Logo, uma string em C é um exemplo de uma estrutura de dados vetor.

Contudo, devemos nos lembrar que uma estrutura de dados só é funcional com um conjunto de operações associadas.

UNIVASF

Vetores

Para uma melhor compreensão dos conceitos apresentados veremos um exemplo.

Vamos definir um TAD matriz com seu grupo de operações.

A estrutura a seguir permite o armazenamento de uma matriz e as operações criar, inicializar, imprimir, somar, subtrair e multiplicar matrizes são definidas.

UNIVASF

```
typedef struct
{
   int nl;
   int nc;
   int **elementos;
}MATRIZ;
void criar_matriz (int, int, MATRIZ *);
void inicializar_matriz (MATRIZ *);
void imprimir_matriz (MATRIZ *);
void somar_matrizes (MATRIZ *, MATRIZ *,
 MATRIZ *);
void subtrair_matrizes (MATRIZ *, MATRIZ *,
 MATRIZ *);
void multiplicar_matrizes (MATRIZ *, MATRIZ *,
 MATRIZ *);
```

```
void criar_matriz (int nl, int nc, MATRIZ *m) {
   int i, j;
   m->elementos = (int **) malloc (sizeof(int *)*nl);
   if (!m->elementos) {
      printf("Nao foi possivel reservar memoria para a
    matriz!\n");
      exit(1);
   }
   for (i=0; i<nl; i++) {
      m->elementos[i] = (int *) malloc (sizeof(int)*nc);
      if (!(m->elementos[i])) {
         printf("Nao foi possivel reservar memoria para a
    matriz!\n");
         exit(2);
      }
   }
```

```
    m->nl = nl;
    m->nc = nc;
    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
            m->elementos[i][j] = 0;
}
```

```
void inicializar_matriz (MATRIZ *m)
{
   int i, j;
   for (i=0; i<m->nl; i++)
      for (j=0; j<m->nc; j++)
      {
         printf ("\nEntre com matriz[%d][%d]=",i+1,j+1);
          scanf ("%d",&(m->elementos[i][j]));
      }
}
```

```
void imprimir_matriz (MATRIZ *m)
{
   int i, j;
   for (i=0;i<m->nl;i++)
   {
      printf("\n|");
      for (j=0;j<m->nc;j++)
          printf ("%5d", m->elementos[i][j]);
      printf(" |");
   }
}
```

```
void somar_matrizes (MATRIZ *m1, MATRIZ *m2, MATRIZ
   *m3)
{
   if (m1->nl==m2->nl && m1->nc==m2->nc)
   {
      int i, j;
      criar_matriz (m1->nl, m1->nc, m3);
      for (i=0; i<m3->nl; i++)
         for (j=0; j<m3->nc; j++)
            m3->elementos[i][j] = m1->elementos[i][j] + m2-
   >elementos[i][j];
   }
   else
   {
      printf ("A soma nao eh possivel!\n");
   }
}
```

```
void subtrair_matrizes (MATRIZ *m1, MATRIZ *m2,
   MATRIZ *m3)
{
  if (m1->nl==m2->nl && m1->nc==m2->nc)
  {
    int i, j;
    criar_matriz (m1->nl, m1->nc, m3);
    for (i=0; i<m3->nl; i++)
      for (j=0; j<m3->nc; j++)
        m3->elementos[i][j] = m1->elementos[i][j] - m2-
  >elementos[i][j];
  }
  else
  {
    printf ("A subtracao nao eh possivel!\n");
  }
}
```

```
void multiplicar_matrizes (MATRIZ *m1, MATRIZ *m2,
    MATRIZ *m3) {
   if (m1->nc==m2->nl) {
      int i, j, z;
      criar_matriz (m1->nl, m2->nc, m3);
      for (i=0; i<m3->nl; i++)
         for (j=0; j<m3->nc; j++) {
            m3->elementos[i][j] = 0;
            for (z=0; z<m1->nc; z++)
               m3->elementos[i][j] += m1->elementos[i][z] * m2-
    >elementos[z][j];
         }
   }
   else {
      printf ("A multiplicacao nao eh possivel!\n");
   }
}
```

Vetores

É importante salientar que o TAD permite uma liberdade na definição de como os dados serão armazenados, por exemplo, o TAD MATRIZ poderia ser representado pela estrutura abaixo:

```
typedef struct
{
   int nl;
   int nc;
   int *elementos;
}MATRIZ;
```

Vetores – Exercícios

Implemente as operações definidas anteriormente para o TAD MATRIZ considerando a nova estrutura apresentada.

UNIVASF