Tipos de Dados, Tipos Abstratos de Dados Estruturas de Dados

Tipo de dados, tipo abstrato de dados, estruturas de dados

- Termos parecidos, mas com significados diferentes

Tipo de dado

- Em linguagens de programação o tipo de dado de uma variável, constante ou função define o conjunto de valores que a variável, constante ou função podem assumir
  - p.ex., variável *boolean* pode assumir valores *true* ou *false*
- Programador pode definir novos tipos de dados em termos de outros já definidos
  - Tipos estruturados, p.ex., *arrays*, *records*

Estrutura de Dados

- Um tipo estruturado é um exemplo de estrutura de dados
  - Tipos estruturados são estruturas de dados já pré-definidas na linguagem de programação
  - O programador pode definir outras estruturas de dados para armazenar as informações que seu programa precisa manipular
  - Vetores, registros, listas encadeadas, pilhas, filas, árvores, grafos, são exemplos de estruturas de dados típicas utilizadas para armazenar informação em memória principal

Perspectivas para Tipos de Dados

- Tipos de dados podem ser vistos como métodos para interpretar o conteúdo da memória do computador
- Mas, podemos interpretar o conceito de tipo de dado sob outra perspectiva
  - não em termos do que um computador pode fazer (interpretar os bits...), mas em termos do que os usuários (programadores) desejam fazer (p.ex., somar dois inteiros...)
  - O programador não se importa muito com a representação no hardware, mas sim com o conceito matemático de inteiro
  - Um tipo inteiro ´suporta´ certas operações...

Tipo Abstrato de Dados (TAD)

- Os tipos e estruturas de dados existem para serem usados pelo programa para acessar informações neles armazenadas, por meio de operações apropriadas
- Do ponto de vista do programador, muitas vezes é conveniente pensar nas estruturas de dados em termos das operações que elas suportam, e não da maneira como elas são implementadas
- Uma estrutura de dados definida dessa forma é chamada de um Tipo Abstrato de Dados (TAD)

Tipo Abstrato de Dados (TAD)

- TAD, portanto, estabelece o conceito de tipo de dado divorciado da sua representação
- Definido como um modelo matemático por meio de um par (*v*,*o*) em que
  - *v* é um conjunto de valores
  - *o* é um conjunto de operações sobre esses valores
  - Ex.: tipo *real*
    - $v = \Re$
    - *o* = {+, -, *, /, =, <, >, <=, >=}

Definição de TAD

- Requer que operações sejam definidas sobre os dados sem estarem atreladas a uma representação específica
  - ocultamento de informação (*information hiding*)
- Programador que usa um tipo de dado *real, integer, array* não precisa saber como tais valores são representados internamente
  - mesmo princípio pode ser aplicado a listas, pilhas, ...
  - se existe uma implementação disponível de uma lista, p. ex., um programador pode utilizá-la como se fosse uma ´caixa preta´, acessá-la por meio das operações que ela suporta

Definição de TAD

- O conceito de TAD é suportado por algumas linguagens de programação procedimentais
  - Ex. Modula 2, Ada
- Para definir um TAD
  - programador descreve o TAD em dois módulos separados
  - Um módulo contém a definição do TAD: representação da estrutura de dados e implementação de cada operação suportada
  - Outro módulo contém a interface de acesso: apresenta as operações possíveis
  - Outros programadores podem, por meio da interface de acesso, usar o TAD sem conhecer os detalhes representacionais e sem acessar o módulo de definição

Definição de TAD

- Os módulos (ou units) são instalados em uma biblioteca e podem ser reutilizados por vários programas
  - A execução do programa requer a linkedição dos módulos de definição (que podem ser mantidos já pré-compilados em uma biblioteca) junto com o programa
  - Mas o programador não precisa olhar o código do módulo de definição para usar o TAD!
  - Basta conhecer a interface de acesso

Implementação de um TAD

- Uma vez definido um TAD e especificadas as operações associadas, ele pode ser implementado em uma linguagem de programação
- Uma estrutura de dados pode ser vista, então, como uma implementação de um TAD
  - implementação do TAD implica na escolha de uma ED para representá-lo, a qual é acessada pelas operações que ele define
- ED é construída a partir dos tipos básicos (*integer*, *real*, *char*) ou dos tipos estruturados (*array*, *record*) de uma linguagem de programação

Características de um TAD

- Característica essencial de TAD é a separação entre a definição conceitual – par (v, o) – e a implementação (ED específica)
  - O programa só acessa o TAD por meio de suas operações, a ED **nunca** é acessada diretamente
  - "ocultamento de informação"

Características de um TAD

- Programador tem acesso a uma descrição dos valores e operações admitidos pelo TAD
- Programador não tem acesso à implementação
  - Idealmente, a implementação é ´invisível´ e inacessível
  - Ex. pode criar uma lista de clientes e aplicar operações sobre ela, mas não sabe como ela é representada internamente
- Quais as vantagens?

Vantagens do uso de TADs

- Reuso: uma vez definido, implementado e testado, o TAD pode ser acessado por diferentes programas
- Manutenção: mudanças na implementação do TAD não afetam o código fonte dos programas que o utilizam (decorrência do ocultamento de informação)
  - módulos do TAD são compilados separadamente
  - uma alteração força somente a recompilação do arquivo envolvido e uma nova link-edição do programa que acessa o TAD
  - O programa mesmo não precisa ser recompilado!
- Correção: TAD foi testado e funciona corretamente

TADs em C: Exemplo

```
/* TAD: Matriz m por n */

/* Tipo Exportado */
typedef struct matriz Matriz;

/* Funções Exportadas */
/* Função cria - Aloca e retorna matriz m por n   */
Matriz* cria (int m, int n);
/* Função libera - Libera a memória de uma matriz */
void libera (Matriz* mat);


                                    /* Continua... */
```

- Note que a *composição* da estrutura Matriz (struct matriz) não está explicita.
- Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos dessa estrutura.
- Os clientes desse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas.

Arquivo matriz.h

TADs em C: Exemplo

```
/* Continuação... */

/* Função acessa - Retorna o valor do elemento [i][j]  */
float acessa (Matriz* mat, int i, int j);
/* Função atribui - Atribui valor ao elemento [i][j]   */
void atribui (Matriz* mat, int i, int j, float v);
/* Função linhas - Retorna o no. de linhas da matriz   */
int linhas (Matriz* mat);
/* Função colunas - Retorna o no. de colunas da matriz */
int colunas (Matriz* mat);
```

Arquivo matriz.h

TADs em C: Exemplo

```
#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include "matriz.h"

struct matriz {
        int lin;
        int col;
        float* v;
};

void libera (Matriz* mat){
   free(mat->v);
   free(mat);
}
```

Arquivo matriz.c

TADs em C: Exemplo

```
/* Continuação... */

Matriz* cria (int m, int n) {
   Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
   if (mat == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);
   }
   mat->lin = m;
   mat->col = n;
   mat->v = (float*) malloc(m*n*sizeof(float));
   return mat;
}
```

Arquivo matriz.c

TADs em C: Exemplo

```
/* Continuação... */

float acessa (Matriz* mat, int i, int j) {
   int k; /* índice do elemento no vetor */
   if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
   }
   k = i*mat->col + j; {armazenamento por linha}
   return mat->v[k];
}

int linhas (Matriz* mat) {
   return mat->lin;
}
```

Arquivo matriz.c

TADs em C: Exemplo

```
/* Continuação... */

void atribui (Matriz* mat, int i, int j, float v) {
   int k; /* índice do elemento no vetor */
   if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
   }
   k = i*mat->col + j;
   mat->v[k] = v;
}

int colunas (Matriz* mat) {
   return mat->col;
}
```

Arquivo matriz.c

Programa cliente – que usa o TAD

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

int main(int argc, char *argv[])
{
  float a,b,c,d;
  Matriz *M;

  // criação de uma matriz
  M = cria(5,5);

  // inserção de valores na matriz
  atribui(M,1,2,40);
  atribui(M,2,3,3);
  atribui(M,3,0,15);
  atribui(M,4,1,21);
```

```
/* Continuação... */


// verificando se a inserção foi feita corretamente
  a = acessa(M,1,2);
  b = acessa(M,2,3);
  c = acessa(M,3,0);
  d = acessa(M,4,1);

  printf ("M[1][2]: %4.2f \n", a);
  printf ("M[2][3]: %4.2f \n", b);
  printf ("M[3][0]: %4.2f \n", c);
  printf ("M[4][1]: %4.2f \n", d);



  system("PAUSE");
  return 0;
}
```

Exercício: TAD Conjuntos (SET)

- Um conjunto é uma coleção de elementos
- Todos os elementos são diferentes entre si e a ordem entre eles é irrelevante
- Ex:

{1,4} = {4, 1} ok

{1,4, 1} não ok

Operações básicas: união, interesecção e diferença

- Se A e B são conjuntos então $A \cup B$ é o conjunto de elementos que são membros de A ou B ou ambos
- Se A e B são conjuntos então $A \cap B$ é o conjuntos de elementos que estão em A e B
- Se A e B são conjuntos então A – B é o conjunto de elementos em A que não estão em B.
- Ex: A = {a,b,c} B = {b,d}
- $A \cup B = \{a,b,c,d\}$
- $A \cap B = \{b\}$
- A – B = {a,c}

Operações usuais

- União(A,B,C)
- Intersecção(A,B,C)
- Diferença(A,B,C)
- Membro(x,A)
- Cria_Conj_Vazio(A)
- Insere(x,A)
- Remove(x,A)
- Atribui(A,B)
- Min(A)
- Max(A)
- Igual(A,B)

Definição das operações

- União(A,B,C): toma os argumentos A e B que são conjuntos e retorna A $\cup$ B à variavel C
- Intersecção(A,B,C): toma os argumentos A e B que são conjuntos e retorna A $\cap$ B à variavel C
- Diferença(A,B,C): toma os argumentos A e B que são conjuntos e retorna A - B à variavel C
- Membro(x,A): toma o conjunto A e o objeto x cujo tipo é o tipo do elemento de A e retorna um valor booleano – true se x $\in$ A e false caso contrário.
- Cria_Conj_Vazio(A): faz o conjunto vazio ser o valor para a variável conjunto A

- Insere(x,A): toma o conjunto A e o objeto x cujo tipo é o tipo do elemento de A e faz x um membro de A. O novo valor de A = A $\cup$ {x}. Se x já é um membro de A, então a operação insere não muda A.
- Remove(x,A): remove o objeto x, cujo tipo é o tipo do elemento de A, de A. O novo valor de A = A – {x}. Se x não pertence a A então a operação remove não altera A.

- Atribui(A,B): Seta o valor da variável conjunto A = ao valor da variável conjunto B.
- Min(A): retorna o valor mínimo no conjunto A. Esta operação pode ser aplicada somente quando os membros do parâmetro A são linearmente ordenados. Por exemplo: Min({2,3,1}) = 1 e Min({‘a’,’b’,’c’}) = ‘a’.
- Max(A): Similar a Min(A) só que retorna o máximo do conjunto.
- Igual(A,B): retorna true se e somente se os conjuntos A e B consistem dos mesmos elementos.

Como Implementar?

Vetor de Booleanos

- Opção para conjuntos de inteiros: 1..N
- i^{th} bit é true se o elemento está no conjunto

- Const N =... ;
- Type set = array [1..N] of boolean;

Exercícios

- Desenvolva:
  - TAD Número Complexo (livro Sincovec & Wiener, Capítulo 1)

- Livros (na biblioteca)
  - Data Structures and algorithms.Addison Wesley. AHO, A.V.; HOPCROFT,J.E.; ULLMAN,J.D., 1982.
  - Data structures using Modula 2. Richard F. Sincovec, Richard S. Wiener, Wiley 1986.
  - Programando em Modula 2. N. Wirth, LTC ed. 1989.