INF 112: Programação II

Aula 11

➔ **Tipo Abstrato de Dados**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

Tipo abstrato de dados

➔ **Toda linguagem de alto nível moderna deve permitir que o programador declare novos tipos de modo a possibilitar uma modelagem mais próxima da realidade. Em C++ um novo tipo é declarado quando usamos a palavra chave `typedef` ou `struct`.**

```
typedef unsigned int Natural; //criando novo tipo
struct Racional // criando novo tipo
{
  int num;
  int den;
};

Natural x; // usando novo tipo
Racional y; // usando novo tipo
```

Tipo abstrato de dados

➔ Esta facilidade permite que programadores possam desenvolver programas mais expressivos. No entanto, este recurso possui limitações, tanto no que se refere à expressividade como na organização de programas:

- Certos tipos de dados não podem ser declarados usando este recurso. Por exemplo, não podemos declarar um tipo que só aceite números pares.

```
typedef int Par;

int main()
{
  Par p;
  p = 1; // Não ocorre erro
  ...
}
```

Tipo abstrato de dados

- **À medida que o programa cresce e várias rotinas passam a utilizar os tipos declarados, torna-se difícil qualquer alteração nos tipos, uma vez que pode afetar todos os trechos de programas onde variáveis dos tipos alterados são utilizadas.**

Tipo abstrato de dados

➔ **Para contornar este problema surgiu um novo conceito denominado de *Tipo Abstrato de Dados* (TAD).**

➔ **O TAD é um modelo independente de qualquer abordagem computacional (não leva em conta possíveis implementações) e que especifica um conjunto de dados e operações que podem ser executadas sobre estes dados.**

➔ **A implementação de TADs em programação tem como objetivo o estabelecimento de uma metodologia que reduza a informação necessária para a criação/programação de um algoritmo através da abstração das variáveis envolvidas em uma única entidade fechada (*encapsulamento*), com operações próprias à sua natureza.**

Tipo abstrato de dados

➔ Um exemplo prático para entender a importância dos TADs é o de representação de dados de estudantes num programa.

➔ Em um projeto anterior à teoria de TAD, um estudante seria representado por variáveis soltas, como seu nome, sua idade e sua matrícula, que seriam utilizadas em separado, sem que houvesse uma ligação lógica entre elas. Além disto, o programador teria que saber que tais variáveis soltas estariam relacionadas à entidade `estudante`.

➔ Com o conceito de TAD, a ideia é que não haja nome, idade e matrícula soltos pelo programa. A abordagem seria pela criação do tipo `estudante` que descrevesse estes dados e ainda que houvesse operações próprias ao tipo `estudante`.

Tipo abstrato de dados

- A ideia é que o programador só faça a manipulação dos dados via operações para este fim e que os detalhes destas operações lhe sejam transparentes (*ocultação de informação*).

- Imaginemos então um TAD Estudante com os dados:
  - nome, idade, matrícula.

- e as operações:
  - `maior_de_idade`: diz se estudante é maior de idade ou não;
  - `valida_matricula`: diz se número de matrícula é válido ou não.

Tipo abstrato de dados

➔ **Vamos agora dar uma ideia da implementação deste modelo com os recursos de programação que conhecemos.**

➔ **Com o que sabemos, TADs são implementados utilizando-se tipos compostos (*structs*) e funções para representar as operações do modelo. Assim, poderíamos ter:**

```
struct Estudante {
  char nome[80];
  short int idade;
  int matricula;
};

bool maiorDeIdade(Estudante &e);
bool validaMatricula(Estudante &e);
```

Tipo abstrato de dados

➔ Um outro exemplo poderia ser a representação de números racionais. Um número racional é representado pela razão de dois números inteiros. Algumas operações comuns para números racionais são soma e teste de igualdade. Para implementar o `TAD número racional` vamos incluir as operações acima mais algumas outras úteis no programa.

Arquivo racional.h

```
struct Racional {
  int num;
  int den;
};

Racional somaRacional(Racional r1, Racional r2);
bool igualRacional(Racional r1, Racional r2);
int getNumRacional(Racional r);
int getDenRacional(Racional r);
Racional criaRacional(int num, int den );
```

Tipo abstrato de dados

Arquivo racional.cpp

```
#include "racional.h"

Racional criaRacional(int num, int den ) {
  Racional r;
  r.num = num;
  r.den = den==0?1:den;
  return r;
}

Racional somaRacional(Racional r1, Racional r2) {
  Racional r;
  r.num = r1.num*r2.den + r2.num*r1.den;
  r.den = r1.den*r2.den;
  return r;
}
          . . .
```

Tipo abstrato de dados

```
        . . .
bool igualRacional(Racional r1, Racional r2){
  if (r1.num*r2.den == r1.den*r2.num)
    return true;
  return false;
}

int getNumRacional(Racional r) {
  return r.num;
}

int getDenRacional(Racional r) {
  return r.den;
}
```

Tipo abstrato de dados

- A questão é que o TAD é somente um modelo. Ao implementá-lo, não há nenhuma segurança de que as operações e regras de operação desejadas para este tipo sejam respeitadas.

- Não existem recursos para ocultar a estrutura de dados de procedimentos que não foram projetados para a sua manipulação.

- Não existe uma forma de relacionar explicitamente as estruturas de dados com os procedimentos que as manipulam.

- Veremos mais à frente como a programação orientada a objetos surge para sanar estes problemas, além de outras vantagens que veremos no devido tempo.

- Por ora, vejamos um outro TAD muito importante em ciência da computação, o TAD lista.

TAD lista

➔ **Implementações do TAD lista compõem provavelmente o tipo de estrutura de dados mais utilizado em programação, uma vez que muitas situações aparecem naturalmente na forma de lista.**

➔ **Uma lista é uma coleção L:[a_1, a_2, ..., a_n], $n \geq 0$, cuja propriedade estrutural baseia-se apenas na posição relativa dos elementos, que são dispostos linearmente.**

➔ **Se n=0, dizemos que a lista é vazia; caso contrário, são válidas as seguintes propriedades:**

- **a_1 é o primeiro elemento de L;**
- **a_n é o último elemento de L;**
- **a_k, $1<k<n$, é precedido pelo elemento a_{k-1} e seguido por a_{k+1} em L.**

TAD lista

➔ **Em outras palavras, a característica fundamental de uma lista linear é o sentido de ordem unidimensional dos elementos que a compõem.**

➔ **Dentre as várias operações comuns de um TAD lista, podemos citar as seguintes como sendo as principais:**

- ***Inserção*** **de um elemento;**
- ***Remoção*** **de um elemento;**
- ***Localização*** **de um elemento.**

➔ **Ao implementar o TAD lista, a operação de inicialização também é importante. Outra operação que poder ser necessária é a de finalização ou destruição (liberação da memória utilizada).**

Listas estáticas contíguas

➔ **Vejamos então uma das implementações mais simples do TAD lista, a estrutura de dados: lista estática contígua.**

➔ **Listas estáticas são tipicamente implementadas através de arranjos.**

➔ **Em uma lista estática contígua, o sucessor de um elemento ocupa posição física subsequente na memória.**

➔ **Então, o arranjo associa o elemento a_i com o índice *i* (mapeamento sequencial).**

- **Características de lista estática contígua:**
  - **Elementos armazenados fisicamente em posições consecutivas;**
  - **A inserção de um elemento na posição *i* causa o deslocamento para a direita do elemento a_i ao último;**
  - **A eliminação do elemento a_i requer o deslocamento para a esquerda do a_{i+1} ao último.**

Listas estáticas contíguas

- **Vantagem:**
  - **Acesso direto indexado a qualquer elemento da lista.**
- **Desvantagem:**
  - **Movimentação quando um elemento é eliminado/inserido;**
  - **Tamanho máximo pré-estimado (esta é uma desvantagem de qualquer lista estática, na verdade).**
- **Quando usar:**
  - **Listas pequenas;**
  - **Tamanho máximo bem definido;**
  - **Inserção/remoção no fim da lista.**

Listas estáticas contíguas

➔ **Mostraremos agora uma implementação em C++ deste tipo de lista. Para fins didáticos, descreveremos uma estrutura de dados para representar lista de inteiros, mas, obviamente, a estrutura ensinada pode ser empregada para qualquer tipo de dado válido, nativo ou criado pelo programador.**

Listas estáticas contíguas

```
#define MAX 50
struct Lista {
   int elem[MAX]; // arranjo para os elementos.
   int posUlt; // posicao do ultimo elemento.
};

// Inicializa lista como vazia
void fazListaVazia(Lista &l) {
   l.posUlt = -1;
}

// Testa se lista estah vazia
bool listaVazia(Lista &l) {
   return (l.posUlt == -1);
}

bool listaCheia(Lista &l) {
   return (l.posUlt == MAX-1);
}
```

Listas estáticas contíguas

```
bool insere(Lista &l, int e, int i) {
// Insere o elemento e na posicao i.
// Retorna true para sucesso e false para falha.
  int j;
  // Se hah espaco e posicao eh valida
  if ( !listaCheia(l) && i >= 0 && i <= l.posUlt+1 ) {
    for (j = l.posUlt+1; j >= i+1; j--) // shiftando
      l.elem[j] = l.elem[j-1];
    l.elem[i] = e;
    l.posUlt++;
    return true;
  }
  return false;
}
```

Listas estáticas contíguas

```
bool remove(Lista &l, int i) {
// Remove o elemento da posicao i.
// Retorna true para sucesso e false para falha.
   int j;
   // se a posicao para remover for valida
   if (i >= 0 && i <= l.posUlt) {
     for (j=i; j <= l.posUlt-1; j++)
        l.elem[j] = l.elem[j+1];
     l.posUlt--;
     return true;
   }
   return false;
}
```

Exercício

➔ **Implemente as operações: `tamanho` (retorna a quantidade de elementos da lista), `localiza` (retorna a posição do elemento procurado na lista) e `imprime` (imprime os elementos da lista).**