Estruturas de Dados - T.332

Capítulo 5

Tipos Abstratos de Dados

Transparência 1

Abstração

- Abstrato tem sido definido como:
  - dissociado de qualquer instância específica,
  - ideal,
  - expressando uma qualidade separadamente do objeto representado.
- Abstração envolve a descoberta de qualidades essenciais de um objeto concreto ou grupo de objetos, dando a estas um nome:
  - descrevemos aspectos essenciais da estrutura de um objeto ou conceito,
  - descrevemos o comportamento de um objeto ou conceito nomeando aspectos importantes deste.

Transparência 2

Exemplo:

2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40

30,40,22,32,36,20,34,4,18,26,6,2,38,16,12,28,8,14,10,40

- É fácil memorizarmos os números da 1ª série, pois precisamos apenas memorizar a sua regra de formação.
- Para a segunda série, necessitamos rememorar os números.
- A abstração facilita descrever fatos: teríamos muita dificuldade de lidar com o mundo, se lembrássemos sempre de todos os detalhes de coisas, ao invés de abstrair e generalizar.

- O nível de abstração com que descrevemos algo é chamado de **granularidade**.

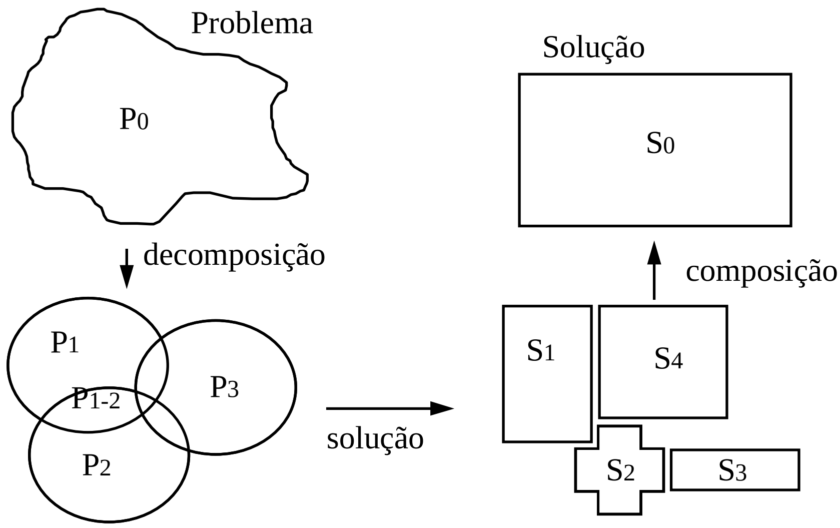
Transparência 3

Abstração em Programação

- A idéia de abstração é aplicada ao idealizarmos um programa de computador.
- Quando queremos construir um programa de computador (sistema) estamos querendo **resolver um problema real**:
  - Descrevemos que **informação** é necessária,
  - Descrevemos, incialmente de forma sucinta, o **comportamento** que o programa deve possuir, para resolver este problema.
  - Estamos **abstraindo** do problema real para tratar de **seus aspectos esseciais**, descrevendo-os.
  - Em programação usamos diversos tipos de abstração:

Transparência 4

Procedimento geral para a solução de um problema:



Transparência 5

Abstração Procedural

- Um programa é um conjunto de instruções que executa uma tarefa:
  - Esta tarefa geralmente vai ser complexa, mas pode ser descrita sucintamente em palavras,
  - É lógico agrupar um conjunto de instruções que realiza uma tarefa especíca dentro do todo sob um nome:
    - Facilita o projeto de um sistema, pois iniciamos com um descrição muito abstrata, que vamos detalhando.
    - Facilita a compreensão de um sistema complexo, se organizarmos conjuntos de intruções para uma tarefa sob um nome.
    - Chamamos a isso de procedimentos, funções (em "C") ou métodos (em OOP).

Transparência 6

Abstração Procedural (definição):

- Nomear um conjunto de ações, instruções ou código de forma que ele possa ser manipulado ou executado simplesmente através de seu nome.
  - Parametrização:
    - Podemos separar o que é fixo ou interno em um procedimento do que é variável, dependente do mundo "exterior", parametrizando a parte variável:
      - normalmente os dados que o procedimento manipula.
    - Através da parametrização podemos utilizar o mesmo procedimento em diferentes contextos, com diferentes dados.
    - Como procedimentos podem ser chamados de outros, temos assim um mecanismo para abstrações de nível mais alto.
    - A parametrização garante que procedimentos não terão efeitos indesejados sobre dados "externos".

Transparência 7

Tipos Abstratos de Dados: ADTs ou TADs

- É muito mais conveniente descrever-se algoritmos e conjuntos de dados em termos das operações efetuadas
  - mais *intuitivo* do que se descrever todos os detalhes de uma implementação,
  - Podemos descrever uma estrutura de dados como uma **pilha** ou **lista** nestes termos, programando-a de acordo com esta idéia,
  - isto é chamado tipo abstrato de dado.
  - Uma característica básica é que nada externo às definições de dados e algoritmos de um TAD deva referenciar a sua estrutura interna e vice-versa.
  - Mecanismo sistemático para facilitar o desenvolvimento de grandes sistemas.

Transparência 8

Tipos Abstratos de Dados:

- Abstração de dados:

  Uma abstração de dados consiste das três partes seguintes:
  - Um conjunto ***S*** de objetos, cuja estrutura de representação é irrelevante,
  - Um conjunto ***P*** de operações defindas sobre os elementos de ***S***,
  - Um conjunto ***R*** de regras que definem as operações e os relacionamentos entre os elementos de ***S***.
- Tipo Abstrato de Dado:
  - Um conjunto de abstrações de dados, cada qual com seus elementos, operações e regras próprios.
  - Exemplo clássico: pilha, lista

Transparência 9

Tipos Abstratos de Dados X OOP

- Conceitos muito semelhantes
- TADs são uma técnica programação concisa e eficiente, semelhante à OOP, só que baseada em linguagens de programação de 3ª geração (estruturadas).
- Diferenças:
  - em TADs, a tarefa de definir quais procedimentos são aplicáveis a quais estruturas de dados, é função do programador.
  - em OOP a sintaxe da linguagem e o compilador garantem que um objeto (como conjunto de dados) "saiba" quais operações são aplicáveis a ele.

Transparência 10

Procedimento organizacional em TADs

- Concepção:
  - Semelhante à OOP
  - Concebemos um Objeto (TAD) atraves da descrição do comportamento que queremos que tenha de de quais dados deve "conter" (Top-Down)
    - Ex.: Uma lista deve poder receber e devolver elementos, deve conter os elementos em um ordem, etc.
    - Através de refinamentos sucessivos detalhamos esta descrição.
- Programação:
  - Organizamos um programa em **Módulos**, cada qual contendo um ou mais TADs relacionados
  - Um módulo contem a declaração de tipo de dados e as funções relativas a um TAD.

Transparência 11

Organização Geral

- Programa principal:
  aplica.c

```
#include <stdio.h>

main ()
{
 char   *a, *b;
 lista  *minhaLista,
        *outraLista;

 minhaLista = criaLista(20);
 .....
 .....
}
```

- Módulo Lista:
  lista.c

```
#include <stdio.h>
typedef lista{
   int elemento;
   lista *proximo;
}

lista *criaLista(int tam)
{
   - - -
}
lista *retiraDaLista()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
```

Transparência 12

Vantagens

- Reutilizabilidade
  - Um TAD bem programado eu posso utilizar em muitos programas.
- Abstração
  - Se eu uso um TAD simplesmente prestando atenção às qualidades e ao comportamento que ele exporta, a minha vida fica mais fácil.
- Manutenibilidade
  - Se eu quiser que as Listas que eu uso no meu programa se comportem diferente, eu só preciso fazer alterações no TAD

Transparência 13

Recursos de"C" para a criação de TADs

- Aspectos procedurais (comportamentais):
  - Funções ou Procedimentos (funções tipo `void`).
- Aspectos de estruturas de dados:
  - Declaração de **tipos derivados estruturados**:
    - estruturas são tipos de dados definidos pelo programador que têm um nome e representam um conjunto de um ou mais tipos de dados.

```
typedef struct Empregado
  {
    char    nome[40];
    char    endereço[100];
    char    função[20];
    int     salário;
    - - -
  }
struct Empregado emp1, emp2, empX;
```

Transparência 14

Usando Estruturas

- Estruturas podem ser declaradas e utilizadas diretamente, como se fossem variáveis:

```
struct {
  char     nome[50];
  char     rua[50];
  char     cidade[30];
  long     cep; } endereço;

endereço.nome = "José Arantes";
endereço.rua = "Estrada Geral Canasvieiras";
```

  - nesse caso não estamos definindo um novo tipo, apenas declarando uma variável com uma estrutura complexa.
  - para a referência e utilização dos **campos** usa-se o ponto "." imediatamente após o nome da variável.

Transparência 15

Usando Estruturas II

- Estruturas são mais úteis quando utilizadas para a definição de **tipos derivados estruturados**

```
typedef struct endereço {                  // Decl. Tipo Deriv.
  char     nome[50];
  char     rua[50];
  char     cidade[30];
  long     cep; };

struct endereço meuEndereço, outroEndereço; // Decl. variáveis
meuEndereço.nome = "Aldo";
meuEndereço.rua = "Rua Lauro linhares";
```

Transparência 16

Usando Ponteiros para Estruturas

- Uma estrutura, sendo uma variável pode ser referenciada normalmente através de um ponteiro:

```
typedef struct endereço {         // Decl. Tipo Deriv.
  char     nome[50];
  char     rua[50]; };

struct endereço end1, end2;       // Decl. variáveis
struct endereço *enderPointer;    // Decl. ponteiro p/endereço
enderPointer = &end1;
end2 = *enderPointer;
```

- Para acessar os campos de uma estrutura usando um ponteiro para a estrutura usamos o operador **->**

```
enderPointer->nome = "Aldo";
end2.cep = enderPointer->cep;
```

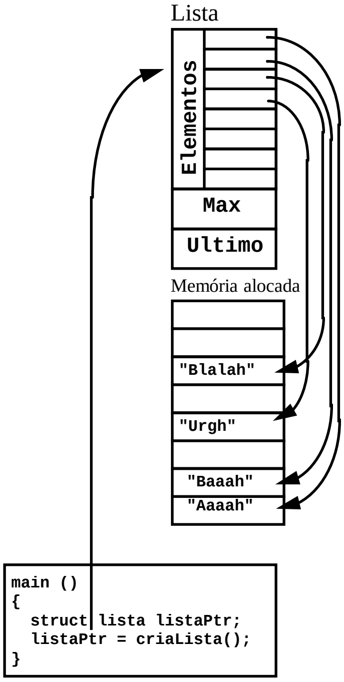
  - **Note que a derreferência "*" aqui não é mais necessária.**

Transparência 17

Exercício-Exemplo

Reimplemente a lista ordenada utilizando um vetor de ponteiros para Strings como um TAD:

- A Lista passa a ser um TAD que possui uma representação interna do nº máximo de elementos e de qual é o último, sob a forma de um **`struct`**.
- Além das funções existentes, crie:
  a) uma que cria uma lista, alocando memória para uma estrutura com o vetor de ponteiros de tamanho max, último e max, devolvendo um ponteiro para esta e
  b) uma que destrói uma lista, dada por um ponteiro para ela, liberando a memória.
- A única variável global que um programa principal usando a lista possui, é um ponteiro apontando para esta.
- O progr.princ. só pode manipular a lista através das funções de acesso.



Transparência 18

■ Módulo lista.h:

```
typedef struct lista {
   char *elemento[30];
   int  ultimo;
   int  max;
}
```

■ Módulo lista.c:

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

lista *criaLista ()
{
 struct lista nova;
 nova = malloc( sizeof(lista) );
 nova->max = 30;
 nova->ultimo = -1;
 return (nova);
}
void destroiLista(lista *morta)
{
 // Libera memória p/os strings
 for (i=0; morta->ultimo; i++)
      free (morta->elemento[i]);
 // Libera memória da lista
 free ( morta );
}
```

Transparência 19