Tipos Abstratos de Dados

Introdução

O que precisamos para ser bons programadores?

- conhecer uma linguagem de programação – só isso?
- usar a linguagem de prog. de maneira eficiente
- identificar as propriedades dos dados
- identificar as características dos dados

Não basta conhecer uma linguagem de programação. Devemos saber programar de maneira eficiente. Mas como? É importante que um programa tenha uma fase de identificação das propriedades dos dados e suas características funcionais. Uma representação adequada, em vista das funcionalidades que devem ser atendidas, dos dados é fundamental para desenvolver programas eficientes e confiáveis.

Programação

Freqüentemente um programador necessita comunicar precisamente o que um procedimento deve realizar, sem qualquer indicação de como deve ser realizada esta tarefa.

Programação - Exemplo

A característica essencial de um TAD é a separação entre conceito e implementação.

Tipos Abstratos de Dados - TADs

Um Tipo Abstrato de Dados é uma coleção bem definida de dados e um grupo de operadores que podem ser aplicados sobre esses dados. Portanto, um TAD é constituído de:

- Uma ESTRUTURA para armazenamento de valores
- Um conjunto de OPERADORES para manipulação dos valores armazenados

A especificação de um TAD descreve quais dados podem ser armazenados (características do TAD), como eles podem ser utilizados (operações), mas não descreve como isso é implementado no programa. Assim, os projetistas de software podem especificar os dados de um programa (o que fazer e manipular os dados) sem fornecer detalhes de implementação.

TAD

- é descrito pela finalidade do tipo e de suas operações e não pela forma como é implementado
- pode ser considerado generalização de tipos primitivos de dados
- quem usa TAD precisa apenas conhecer a funcionalidade que ele implementa
- facilidade manutenção e reutilização de código

A implementação de um algoritmos em uma linguagem de programação exige que se encontre alguma forma de representar TADs, em termos dos tipos de dados e dos operadores suportados pela linguagem considerada. Da mesma forma que um procedimento é usado para encapsular partes de um algoritmo, o TAD pode ser usado para encapsular tipos de dados. Neste caso, a definição do tipo e todas as operações definidas sobre ele podem ser localizadas em uma única seção do programa.

Exemplo - loja de brinquedos

- Composta de: funcionários, clientes, brinquedos, horário de funcionamento, ...
- Operações: classificar os brinquedos, verificação de estoque, pedido de compras, venda, ...

Uma loja de brinquedos pode ser vista como um tipo abstrato de dados. O usuário dessa TAD não se preocupará com quais funcionários estão trabalhando, quantos brinquedos estão disponíveis (como os dados são armazenados) ou como classificar os brinquedos (operações). Certamente, modelar uma loja de brinquedos como uma TAD não é simples, mas o mais importante neste momento é a compreensão do conceito de TADs.

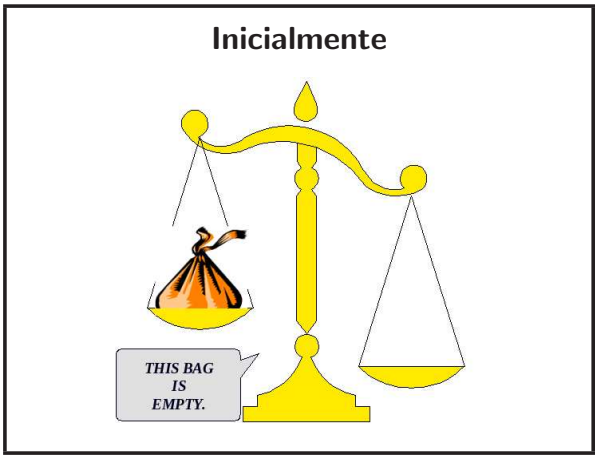
TAD

Acesso aos dados armazenados

- apenas via operações implementadas, isto é, não é possível acessar os dados diretamente
- oculta os detalhes de representação e implementação, apenas funcionalidade é conhecida
- Encapsulam dados e comportamento

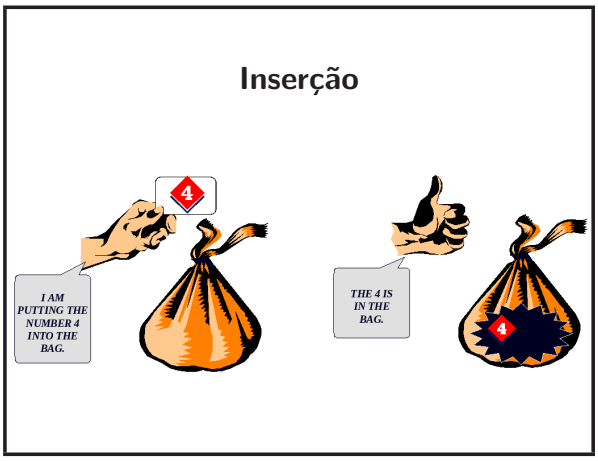
Exemplo: Sacolas (Bags)

Este exemplo é baseado em uma sacola. Esta sacola contém alguns números.

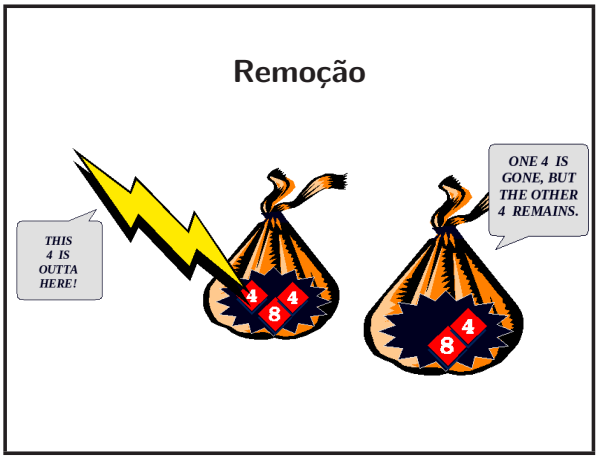


Inicialmente, a sacola está vazia. Este é o estado inicial de qualquer sacola utilizada.

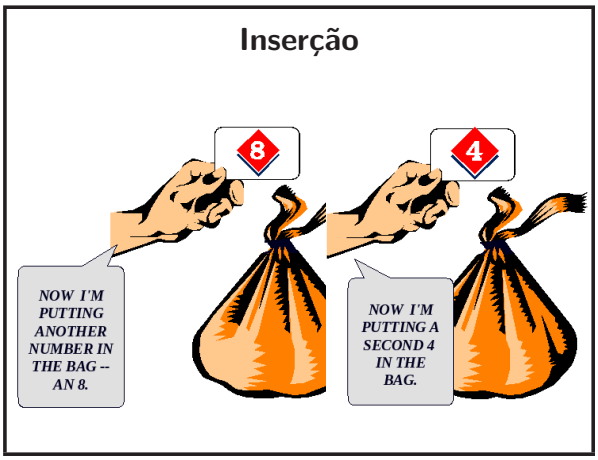
Podemos efetuar uma operação para verificar quantas vezes um determinado número existe dentro da sacola.



Na sacola, podemos inserir números inteiros. No exemplo, o inteiro 4 foi inserido na sacola.



Outra operação possível é remover números da sacola. No exemplo, um número 4 foi removido. Note que o outro número 4 ainda permanece na sacola. Podemos verificar que apenas um número de cada vez é removido.



Podemos verificar no exemplo acima a inserção dos números 8 e 4, nessa ordem. Podemos notar que a sacola suporta diversos números. Além disso, é possível inserir números repetidos.

Podemos definir uma operação para determinar quantos números existem na sacola.

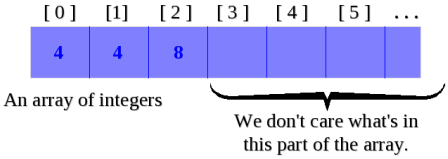
Resumindo

Operações na sacola:

- estado inicial - vazio
- inserção - inserção de um número
- verificação - verificação da existência de um número
- remoção - remoção de um número
- quantidade - quantidade de números (tamanho)

Detalhes de Implementação

- The entries may appear in any order. This represents the same bag as the previous one. . .



An array of integers

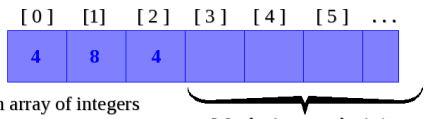
We don't care what's in this part of the array.

As operações que podemos efetuar com a sacola são:

- pode ser colocada no seu estado inicial – vazio;
- números podem ser inseridos;
- pode-se verificar quantas vezes um determinado número se repete;
- números podem ser removidos;
- pode-ser verificar quantos números existem na sacola.

Detalhes de Implementação

- The entries of a bag will be stored in the front part of an array, as shown in this example.



An array of integers

We don't care what's in this part of the array.

Implementação

Veja o código: bag.c e bag.h

Veja também o código: numeros.c

Implementação

Vamos alterar a estrutura Bag para:

```
typedef struct b {
  int item[BAGSIZE];
  int used; // number of elements in item
} Bag;
```

Podemos agora alterar bag.c e utilizar `used` ao invés da função `size`

A reimplementação do TAD bag pode ser realizada sem que os programas que a utilizam sejam alterados. No nosso exemplo, podemos alterar o código bag.c e bag.h sem alterar numeros.c, desde que a interface de bag, isto é, nomes, parâmetros e retorno de funções, não sejam modificados.

Implemente também uma função que verifica se a sacola está cheia.

TAD - Vantagens

- Mais fácil programar
- Não há preocupação com detalhes de implementação
- Logicamente mais claro
- Mais seguro programar
- Apenas os operadores podem mexer nos dados
- Maior independência, portabilidade e facilidade de manutenção do código
- Maior potencial de reutilização de código
- Abstração

Consequência: custo menor de desenvolvimento

TAD- Resumo

- novo tipo de dado com operações que manipulam este tipo de dado
- colocado em uma *unit* separada
- pode ser utilizado por qualquer programa
- sua modificação não afeta os programas que o utilizam

Listas Estáticas

Automação de uma Biblioteca

- todos os livros devem ser cadastrados;
- sistema deve informar se um determinado livro está ou não disponível nas estantes;
- se livro não disponível, o usuário poderá aguardar pela liberação do livro se cadastrando em uma fila de espera;
- quando o livro for devolvido e liberado, o primeiro da fila deve ser contatado para vir buscá-lo.

Solução 2

- alocar espaço para 1000 elementos
- todas as 120.000 filas compartilham o mesmo espaço

Problema: como 120.000 filas podem compartilhar a memória reservada a elas?

Dados

- 120.000 livros
- 1 fila de espera para cada livro
- máximo de 1000 pessoas ficam a espera por um dos livros da biblioteca
- até 30 pessoas ficam a espera de um mesmo livro

O que precisamos?

- interligar os elementos de um conjunto
- definir operações de inserir, retirar e localizar
- flexível para alterar seu tamanho (de acordo com a demanda)

Solução 1

- reservar espaço para 120.000 filas (uma para cada livro), com capacidade para 30 pessoas
- 120.000 vetores de tamanho 30
- espaço reservado para 3.600.000 pessoas

$\Rightarrow$ **muito espaço reservado não é utilizado**

Listas Lineares

- Sequência de zero ou mais itens $x_1, x_2, \ldots, x_n$, na qual x_i é de um determinado tipo e n representa o tamanho da lista linear.
- Assumindo $n \geq 1$:
  - x_1 é o primeiro elemento
  - x_n é o último elemento
  - x_i precede x_{i+1} para $i = 1, 2, \ldots, n$
  - x_i sucede x_{i-1} para $i = 1, 2, \ldots, n$
  - x_i é dito estar na i-ésima posição

Exemplos - listas

- telefônica
- clientes de uma agência bancária
- setores de disco a serem acessados por um SO
- pacotes a serem transmitidos por uma rede
- variáveis globais e outra para variáveis locais
- funções do programa
- parâmetros formais de uma função
- parâmetros reais de uma chamada de função

Operações com Lista

- cria uma lista vazia
- verifica se a lista está vazia
- determina o tamanho da lista
- localiza/modifica o nó que contém um determinado valor
- insere novo item imediatamente após (ou antes) o i-ésimo item
- retira o i-ésimo item
- localiza/modifica o i-ésimo item
- combina duas listas em uma única
- exibe os elementos da lista

O conjunto de operações a ser definido sobre os objetos do tipo Lista depende de cada aplicação, não existindo um conjunto adequado a todas as aplicações. No slide acima descrevemos algumas operações que normalmente são utilizadas pela maioria de aplicações.

Implementação – representações

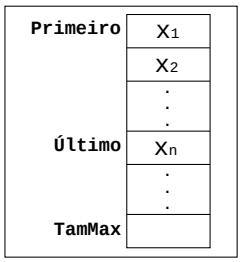
- sequencial (arranjos)
- encadeada (apontadores)

A escolha de uma destas formas dependerá da frequência com que determinadas operações serão executadas sobre a lista.

Existem várias estruturas de dados que podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares. As representações mais utilizadas são as implementações por meio de arranjos e de apontadores. A implementação por cursores (Aho, Hopcroft e Ullman, 1983 p. 48) pode ser útil em algumas aplicações

Lista Sequencial

- itens são armazenados em posições contíguas de memória:



Explora a sequencialidade da memória do computador, de tal forma que os nós de uma lista sejam armazenados em endereços sequenciais, ou igualmente distanciados um do outro. Pode ser representado por um vetor na memória principal ou um arquivo seqüencial em disco.

Características

- inserção após o último item
- inserção na i-ésima posição requer um deslocamento à direita do i-ésimo elemento ao último
- eliminação i-ésimo elemento requer o deslocamento à esquerda do $i+1$-ésimo ao último

Vantagens

- economia de memória – apontadores implícitos
- acesso direto indexado a qualquer elemento da lista
- tempo constante para acessar o elemento i – dependerá somente do índice

Quando usar?

- listas pequenas
- inserção/eliminação no fim da lista
- tamanho máximo bem definido

Desvantagens

- custo para inserir/eliminar itens (deslocamento)
- tamanho máximo pré-determinado – ruim em aplicações em que não existe previsão sobre o crescimento da lista

Implementação

- `void cria(Lista *p_l);`
- `int vazia(Lista *p_l);`
- `void insere_inicio(Lista *p_l, elem_t e);`
- `void insere_fim(Lista *p_l, elem_t e);`
- `int insere_ordenado(Lista *p_l, elem_t e);`
- `int ordenada(Lista *p_l);`
- `void ordena(Lista *p_l);`
- `int remove_inicio(Lista *p_l, elem_t *p_e);`
- `int remove_fim(Lista *p_l, elem_t *p_e);`

- `int remove_valor(Lista *p_l, elem_t e);`
- `void inverte(Lista *p_l);`
- `void libera(Lista *p_l);`
- `void exibe(Lista *p_l);`

Listas Dinâmicas

Listas Lineares

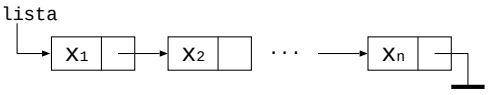
Listas sequenciais:

- algumas operações exigem um grande esforço computacional
- baixo desempenho
- estimativa do tamanho máximo
- se aplicação requer ultrapassar o tamanho máximo?

Na aula anterior vimos que algumas operações sobre listas lineares sequenciais exigem um grande esforço computacional, o que pode ser um fator determinante do baixo desempenho do programa, caso estas operações sejam frequentemente executadas. Além disso, não é possível alterar dinamicamente o tamanho máximo da lista.

Lista Encadeada

- os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor
- posições não contíguas de memória



Cada item da lista é encadeada com o seguinte mediante uma variável do tipo apontador. Este tipo de implementação permite utilizar posições não contíguas de memória, sendo possível inserir e eliminar elementos sem haver a necessidade de deslocar os itens seguintes da lista.

Características

- inserção e eliminação sem deslocamento
- cada nó (registro ou célula) contém um item da lista e um apontador para o próximo nó
- um apontador armazena o endereço do primeiro nó da lista

Estrutura

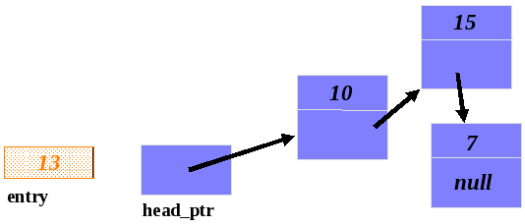
```
typedef int elem_t;
typedef struct no_lista {
  elem_t v;
  struct no_lista *prox;
} No_lista;

typedef No_lista* Lista;
```

Inicialmente

- `head_ptr` é um ponteiro que possui o endereço o primeiro elemento da lista
- `head_ptr` inicialmente deve conter `null`, que implica em lista vazia

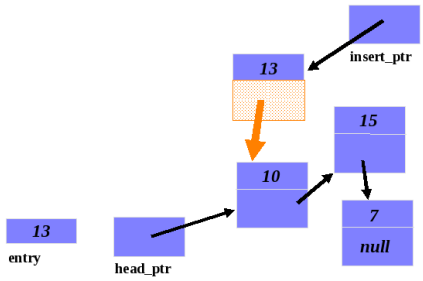
Inserção no início

- vamos inserir 13 no início da lista



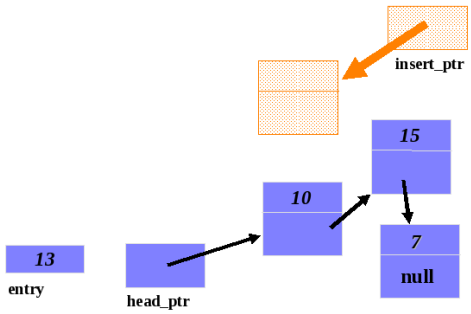
Inserção no início

- novo nó aponta para o primeiro elemento da lista



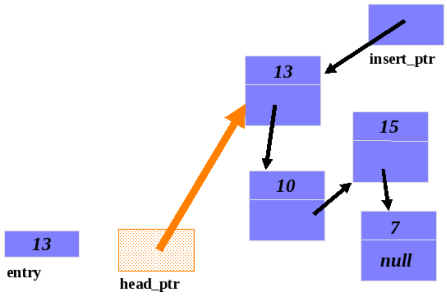
Inserção no início

- cria um novo nó



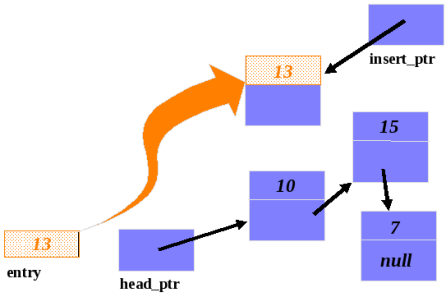
Inserção no início

- `head_ptr` aponta para novo nó



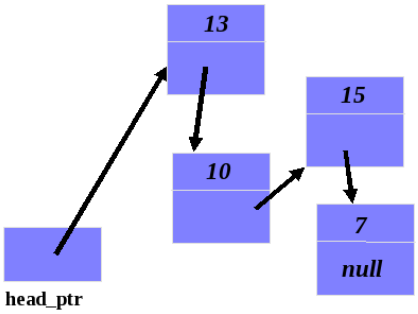
Inserção no início

- insere o valor 13 no nó criado



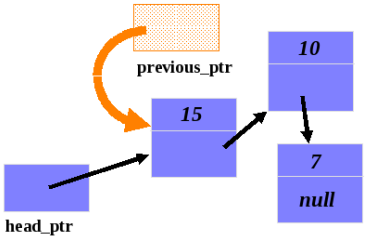
Inserção no início

- após a inserção:



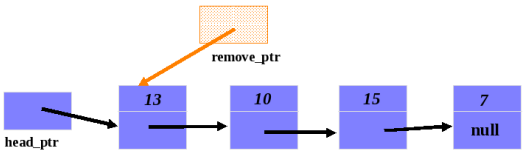
Inserção

- inserir 13 após o 15
- `previou_ptr` ponteiro auxiliar



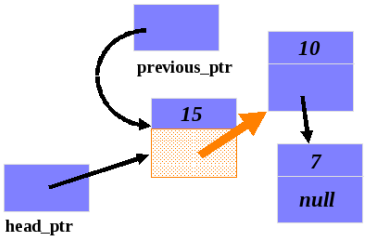
Eliminação do primeiro elemento

- `remove_ptr` aponta para o início da lista



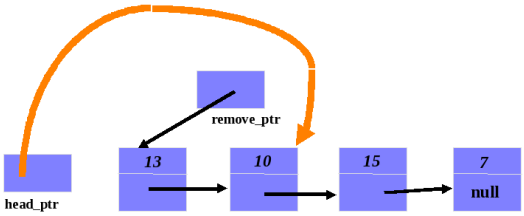
Inserção

- verifica conteúdo do auxiliar: `*previou_ptr`
- `previou_ptr`: aponta p/ a sublista com 10 e 7



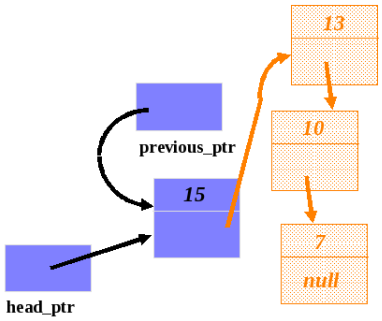
Eliminação do primeiro elemento

- `head_ptr` deve apontar para o próximo do `remove_ptr`



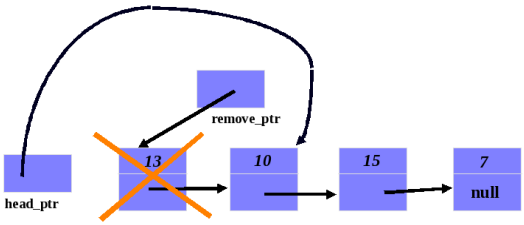
Inserção

- insere o elemento no início da sublista



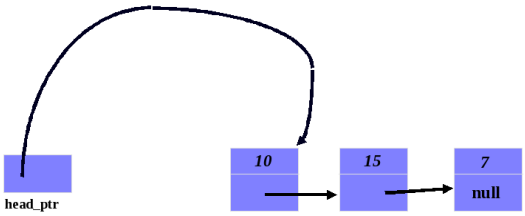
Eliminação do primeiro elemento

- libera espaço de memória



Eliminação do primeiro elemento

- após a eliminação:



Vantagens

- não é necessário pré-definir um tamanho máximo para a lista
  - limite é o tamanho da memória
  - não há alocação desnecessária de espaço
- economia de memória

Desvantagens

- acesso não é indexado – necessário percorrer i nós para encontrar o i-ésimo elemento
- consumo de tempo para alocação e liberação de memória em operações de inserção e remoção

Implementação

- `void cria(Lista *p_l);`
- `int vazia(Lista *p_l);`
- `void insere_inicio(Lista *p_l, elem_t e);`
- `void insere_fim(Lista *p_l, elem_t e);`
- `int insere_ordenado(Lista *p_l, elem_t e);`
- `int ordenada(Lista *p_l);`
- `void ordena(Lista *p_l);`
- `int remove_inicio(Lista *p_l, elem_t *p_e);`
- `int remove_fim(Lista *p_l, elem_t *p_e);`

- `int remove_valor(Lista *p_l, elem_t e);`
- `void inverte(Lista *p_l);`
- `void libera(Lista *p_l);`
- `void exibe(Lista *p_l);`

Bibliografia

- Michael Main and Walter Savitch, *Data Structures and Other Objects Using C++*, 2. edição, Addison Wesley, 2004.
- Roberto Ferrari, *Curso de estruturas de dados*, São Carlos, 2006. Apostila disponível em: `http://www2.dc.ufscar.br/~ferrari/ed/ed.html`