sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Tipos Abstratos de Dados

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação

UFSJ
Universidade Federal de São João del-Rei

Introdução

Muitos programas precisam trabalhar com coleções de dados.

Assim surgiram na computação diversos tipos que permitem armazenar e manipular os dados de forma eficiente.

Um **Tipo Abstrato de Dado (TAD)** é um encapsulamento que inclui a representação de um tipo de dado e as operações para mainpulá-lo, ocultando os detalhes de implementação.

Um TAD também pode ser entendido como o conceito que descreve o dado e suas operações.

Introdução

As pessoas trabalham com dados, necessitando manipulados e armazená-los em estruturas. Ex: agenda de telefone.

Uma **Estrutura de dados (ED)** é uma organização de dados e algoritmos de forma coerente e racional de modo a otimizar o seu uso.

De acordo com o modo que os dados são organizados e como as operações que são efetuadas sobre estes dados pode-se solucionar de forma simples problemas extremamente complexos.

Existem diversos modelos de EDs, e novos modelos são criados constantemente pois acompanham também a evolução dos algoritmos e das linguagens de programação.

Introdução

Estruturas de dados são tipos de dados compostos classificados em:

Estruturas homogêneas: coleções de dados do mesmo tipo.

- Por exemplo: as listas de Haskell

Estruturas heterogêneas: coleções de dados de tipos diferentes.

- Por exemplo: as tuplas e os tipos algébricos de Haskell

Introdução

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 **Tipo Abstrato de Dados (TAD)**.

Por exemplo, considerando agenda telefônica como um TAD:

Armazena uma sequência de nomes e telefones;

Possíveis operações: inserção, exclusão, alteração e busca pessoas.

Introdução

Uma vez definido o TAD, pode-se implementá-lo usando uma linguagem de programação e uma ED.

Portanto, pode-se entender que:

O TAD define o comportamento externo e as operações.

Uma ED pode ser usada para implementar um TAD;

Isso implica que, quando é necessário armazenar um conjunto de dados, é preciso:

1. Identificar qual TAD é o mais adequado;
2. Identificar qual ED foi usada para implementar o TAD.

TADs em Haskell

A maneira mais adequada de se implementar um TAD é criando um módulo.

O TAD é implementado usando um tipo algébrico, contudo, um TAD não é definido pela nomeação de seus valores, mas pela nomeação de suas operações (funções).

Isso significa que a representação dos valores dos TADs não é conhecida, somente as operações para manipulá-los.

TAD Pilha

Pilha é um TAD homogêneo em que os valores são colocados e/ou retirados utilizando uma estratégia ***Last In First Out*** **(LIFO)**.

Segue-se uma implementação para o TAD Pilha (Stack):

```
module Stack (Stack, push, pop, top, stackEmpty) where

push :: t -> Stack t -> Stack t --coloca um item no topo da pilha
pop :: Stack t -> Stack t --retira o item do topo da pilha
top :: Stack t -> t --obtém o item do topo da pilha
stackEmpty :: Stack t -> Bool --verifica se a pilha está vazia

data Stack t = EmptyStk | Stk t (Stack t)
```

TAD Pilha

```
instance (Show t) => Show (Stack t) where
  show (EmptyStk) = "#"
  show (Stk x s) = (show x) ++ " | " ++ (show s)

push x s = Stk x s

pop EmptyStk = error "retirada em uma pilha vazia"
pop (Stk _ s) = s

top EmptyStk = error "topo de uma pilha vazia"
top (Stk x _) = x

stackEmpty EmptyStk = True
stackEmpty _ = False
```

TAD Pilha

O TAD criado pode ser utilizado por outros módulos:

```
module UsaStack where
import Stack

listaParaPilha :: [t] -> Stack t
listaParaPilha [ ] = EmptyStk
listaParaPilha (x : xs) = push x (listaParaPilha xs)

pilhaParaLista :: Stack t -> [t]
pilhaParaLista s
  |stackEmpty s = [ ]
  |otherwise = (top s) : (pilhaParaLista (pop s))
```

TAD Fila

Fila é um TAD homogêneo em que os valores são colocados e/ou retirados utilizando uma estratégia ***First In First Out*** **(FIFO)**.

Segue-se uma implementação do TAD Fila (Queue):

```
module Queue (Queue, enqueue, dequeue, front, queueEmpty) where

enqueue :: t -> Queue t -> Queue t --coloca um item no fim da fila
dequeue :: Queue t -> Queue t --retorna a fila sem o item da frente
front :: Queue t -> t --pega o item da frente da fila
queueEmpty :: Queue t -> Bool --testa se a fila está vazia

data Queue t = Que [t]
```

TAD Fila

```
instance (Show t) => Show (Queue t) where
  show (Que [ ]) = ">"
  show (Que (x : xs)) = "<" ++ (show x) ++ (show (Que xs))

enqueue x (Que q) = Que (q ++ [x])

dequeue (Que (x : xs)) = Que xs
dequeue _ = error "Fila de espera vazia"

front (Que (x : _)) = x
front _ = error "Fila de espera vazia"

queueEmpty (Que [ ]) = True
queueEmpty _ = False
```

TAD Fila

O TAD criado pode ser utilizada por outros módulos:

```
module UsaFila where
import Stack
import Queue

filaParaPilha :: Queue t -> Stack t
filaParaPilha q = qts q EmptyStk
where
  qts q s
   | queueEmpty q = s
   | otherwise = qts (dequeue q) (push (front q) s)
```

TAD Árvore Binária

Árvore binária é um TAD homogêneo que pode ser uma folha ou um nó com duas subárvores: esquerda e direita. Nessa definição, ela não pode ser vazia.

Segue-se uma implementação do TAD Árvore Binária não vazia (ArvBin):

```
module ArvBin (ArvBin, tamanho, altura) where

tamanho :: ArvBin t -> Int --calcula a quantidade de elementos
altura :: ArvBin t -> Int --calcula a altura da árvore

data ArvBin t = Folha t | No (ArvBin t) t (ArvBin t)
```

TAD Árvore Binária

```
instance (Show t) => Show (ArvBin t) where
  show (Folha x) = show x
  show (No e x d) = show e ++ " | " ++ show x ++ " | " ++ show d


tamanho (Folha _) = 1
tamanho (No esq _ dir) = 1 + tamanho esq + tamanho dir


altura (Folha _) = 1
altura (No esq _ dir) = 1 + max (altura esq) (altura dir)
```

Exercícios

1. Modifique a implementação da função show de Stack para que não apareça o símbolo # quando uma pilha com um ou mais elementos é impressa. Ex.: show (Stk 5 (Stk 2 EmptyStk)) deve resultar em 5 | 2
2. Reimplemente a TAD Stack de modo que uma pilha contenha uma lista de valores: `data Stack t = Stk [t]`
3. Implemente uma TAD Lista com as seguintes operações:
    1. show, que imprime igual a do Haskell (usando colchetes e vírgula);
    2. read, que transforma de string para lista;
    3. add, adiciona um elemento e na posição i;
    4. remove, retira o elemento da posição i;
    5. len, calcula o tamanho da lista;
    6. vazia, indica se a lista está vazia ou não;
    7. concatena, junta duas listas em uma só.
4. Implemente uma TAD Árvore de busca binária, que pode ser vazia ou ser um nó com duas subárvores. Implemente as funções show, add, remove, tamanho, altura e fromList (cria árvore a partir de uma lista do Haskell).