Biblioteca de Classes para Implementar as Estruturas de Dados Básicas como Tipo Abstrato de Dados

LLP001/1995

Projeto de Iniciação Científica

Francisco Sapori Junior

Orientadora: Mariza Andrade da Silva Bigonha
DCC - ICEX - UFMG
Belo Horizonte, 20 de abril de 1995.

1 Motivação

Um Tipo Abstrato de Dados (TAD) pode ser definido como um modelo matemático com um conjunto
operações definidas sobre o modelo. O conjunto de inteiros mais as operações de adição, subtração, e
caracterizam um tipo abstrato de dados. Para implementar o modelo matemático utilizamos estrutu
de dados [1, 12, 13].

Existem diversas vantagens em se utilizar Tipos Abstratos de Dados (TADs). A principal é o aume
do grau de modularidade do programa. Com TADs pode-se: (1) Permitir que se "esconda" como u
determinada estrutura é implementada. (2) Permitir que no caso de se alterar a implementação de u
determinada estrutura de dados as interfaces, ou seja, as declarações dos cabeçalhos dos procedimento
funções não sejam alteradas. (3) Permitir uma maior reutilização de programas. (4) Permitir organiz
programas maiores e mais complexos de uma forma ordenada e controlada.

Estas vantagens ficam mais evidentes à medida que mais programas, sejam eles pequenos ou grand
simples ou complexos, são desenvolvidos e têm um reflexo direto na **qualidade** do produto final, ou se
do seu programa. Portanto, a motivação básica para se implementar qualquer Tipo Abstrato de Dad
(TAD) decorre naturalmente das vantagens descritas acima.

2 Objetivos

O objetivo deste projeto é prover uma biblioteca de classes [7] para implementar as estruturas de dad
básicas [5, 6, 1, 12, 13], listas lineares, árvores e grafos como tipo abstrato de dados. A filosofia a s
adotada baseia-se no conceito de máquina de estados [7].

O projeto de estruturas de dados como estados explícitos permite uma forma fácil de documentar int
faces. Os tipos mais comuns de estruturas de dados podem ser caracterizados por estados internos e u
posição corrente.

Este enfoque pode, a primeira vista parecer contradizer a "abstração" da abordagem de tipo abstra
de dados, na qual a programação orientada é baseada. Mas não é verdade. A teoria de tipos abstratos
dados sugere que estruturas de dados devam ser dadas por uma descrição abstrata baseada em operaçõ
aplicáveis e nas propriedades formais destas operações. Isto de forma alguma caracteriza estruturas
dados como um simples local para deixar os dados. É exatamente o contrário: a introdução de estad
e operações sobre o estado, torna a especificação do tipo abstrato de dados mais poderosa à medida q
ele possui mais funções e mais propriedades. Note que o estado é uma abstração, ele nunca é acessa
diretamente mas somente manipulado através de comandos e `queries`. Objetos vistos como máquina
estados reflete em tipos abstratos de dados os quais são mais operacionais, mas nem por isto os torna
menos abstratos.

3 Abordagens Existentes

3.1 Implementação de TADs usando Linguagens tipo Pascal

Existem na literatura algumas propostas de implementação de estruturas de dados básicas como tipo a
strato de dados. Uma delas [1] mostra uma implementação usando uma linguagem de programação pr
cedural (Pascal), para atingir seu objetivo. Esta abordagem apresenta alguns problemas sérios. Tomem
como exemplo, o tipo abstrato de dados Lista. Para criar um **tipo abstrato de dados Lista**, é necessá
definir um conjunto de operações sobre os objetos do tipo Lista. Por exemplo:
(1) Criar uma lista linear vazia.
(2) Inserir um novo item imediatamente após o i-ésimo item.
(3) Retirar o i-ésimo item.
(4) Localizar o i-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
(5) Combinar duas ou mais listas lineares em uma lista única.
(6) Partir uma lista linear em duas ou mais listas.
(7) Fazer uma cópia da lista linear.
(8) Ordenar os itens da lista em ordem ascendente ou descendente
(9) Pesquisar a ocorrência de um item com um valor particular em algum componente.

```
const
   InícioArranjo = 1;
   MaxTam      = 1000;
type
   Apontador = integer;
   TipoItem  = record
                 Chave : TipoChave;
                 {outros componentes}
               end;
   TipoLista = record
                 Item     : array [1..MaxTam] of TipoItem;
                 Primeiro : Apontador;
                 Último   : Apontador;
               end;
```

Se observarmos o modo como foi declarado o objeto `TipoItem` acima vemos que para o usuário utiliz
este TAD Lista ele tem que ter pleno conhecimento dos detalhes da representação dos tipos declarad
Este fato não está associado com a linguagem usada, mas sim com a forma como o TAD foi definid
Neste exemplo, o usuário tem que saber que o registro `TipoItem` deve ter um campo `Chave` e q
é permitido acrescenter *outros componentes* no mesmo. O item *outros componentes* é dependente
programa do usuário. Este fato contrasta com as principais vantagens sobre o uso de TADs citadas
Seção 1, pois permite que o indivíduo insira dentro da declaração de tipos, outros componentes. É fa
que linguagens tipo Pascal não oferecem o recurso para implementar TADs, tudo fica em aberto. Embo
a implementação seja um pouco complicada, ela poderia ser feita de tal forma que a informação dent
da declaração de tipos não fosse usada.

3.2 Implementação de TADs usando Linguagens Orientadas por Objetos

Outra abordagem [7] encontrada na literatura utiliza-se dos conceitos de programação orientada a objet para implementar uma classe que engloba as estruturas de dados básicas como tipo abstrato de dad Exploraremos mais esta abordagem por estar mais próxima da proposta de nosso trabalho. Um exemp desta abordagem utiliza uma linguagem orientada por objetos `Eiffel` [7] e define a classe de listas linea como representações de seqüências. No exemplo a seguir, cada elemento de uma seqüência, denomina célula, é chamado de *Linkable*. Cada *Linkable* contem um valor e um apontador para outro elemer conforme ilustra a Figura 1:

ed2-projeto-ic1.eps

Neste modelo, T é o tipo dos valores (*value*). A lista é representada por uma célula separada chama cabeça. A célula cabeça contém o endereço da primeira célula *linkable*. Esta célula pode também poss outros itens, como por exemplo, um contador do número de elementos da lista (`nb-elements`).

ed2-projeto-ic2.eps

A vantagem desta representação é que tanto a inserção quanto a remoção é rápida se houver um apontac para a célula *linkable* imediatamente à esquerda do ponto de inserção ou remoção. Por outro lado, es representação não é muito boa para pesquisar por um elemento dado o seu valor ou posição, porç estas operações requerem um caminhamento seqüencial na lista. Existem duas estruturas de dados q são usadas normalmente para se implementar listas: arranjos e apontadores. Os arranjos são bons pa acessar uma posição, mas ruins para inserções e remoções.

Pela Figura 2, vemos que necessitamos duas classes: uma para listas (*header*) e outra para os elemen da lista (*linkable*). A abordagem descrita utiliza as classes *LINKED-LIST* e *LINKABLE*. Note que noção de *LINKABLE* é fundamental para a implementação, mas não é relevante para o usuário. Nes esquema é permitido ao usuário acesso ao módulo com primitivas para manipular listas mas sem força a se preocupar com detalhes de implementação, como por exemplo a presença de elementos *LINKABLE*

class LINKABLE[T]
células *LINKABLE* para serem usadas com as listas lineares.
export • **features**
value: T;
right: *LINKABLE* [T] **end**

Como no método [1], para definir uma classe TAD, operações devem ser acrescentadas à definição registro. A seguir mostramos como seria definido as classes *LINKABLE* e *LINKED-LIST1* nesta ab dagem.

class LINKABLE[T]
células *LINKABLE* para serem usadas com as listas lineares.
export veja abaixo
features

```
        value: T;
        right: LINKABLE [T]
        Create (initial:T) is
                inicializa com o valor inicial
                do value := initial end;
        change-value (new:T) is
                substitua o valor com new
                do value := new end;
        change-right (other:LINKABLE [T]) is
                coloque other a direita da célula corrente
                do right := other end;
end class LINKABLE
```

class LINKED-LIST1[T]
export
nb-element, empty, value, change-value, insert, delete, search, ···
features
first-element : *LINKABLE* [T];
nb-element: T;
empty : BOOLEAN **is**
a fila está vazia?
do Result := (nb-element = 0) **end**;
value (i:integer): T **is**
valor do i-ésimo elemento
require l ≤ i; i ≤ nb-element
local elem: *LINKABLE* [T]; j: integer
do from j := 1; elem := first-element
invariant j ≤ i **variant** i − j
until j = i
loop j := j + 1; elem := elem.right **end**;
Result := elem.value **end**;
change-value (i:integer; v:T) **is**
substitua por v o valor do i-ésimo elemento da lista
require 1 ≤ i; i ≤ nb-elements
do ··· **ensure** value(i) = v **end**;
insert (i:integer; v:T) **is**
insere um novo elemento de valor v de tal forma
que ele fique sendo o i-ésimo elemento da lista.
require l ≤ i; i ≤ nb-element + 1
local *previous, new*: *LINKABLE* [T]; j: integer
do – cria nova célula
new.Create(v);
if i = 1 then
– insere no inicio da lista
new.change-right(first-element); first-element := new
else from j := 1; *previous* := first-element
invariant j ≥ 1; j ≤ i − 1; **not** *previous.Void* **variant** i − j − 1
until j = i − 1 **loop**
j := j + 1; *previous* := *previous.right* **end**
nb-elements := nb-elements + 1
ensure nb-elements := **old** nb-elements + 1; **not** *empty*
end – *insert*
delete (i:integer) **is** remove o i-ésimo elemento da lista. ··· **end**
search (v:T):integer **is** posição do primeiro elemento de valor v em lista, 0 se não houver ··· **end**
outras *features*
invariant *empty* = (nb-elements = 0) *empty* = first-element.Void
end class *LINKED-LIST1*

Esta abordagem também apresenta alguns problemas. A classe *LINKED-LIST1* mostra que estrutu
que manipulam apontadores podem ser perigosas, principalmente quando combinados com ciclos.
uso de asserções auxilia um pouco, mas a dificuldade que aparece com este estilo de programação é u
forte argumento para encapsular estas operações de uma vez por toda em módulos reusáveis, como
favorecido em abordagens orientadas por objetos. Muito embora o número de elementos representa
por `nb-element` seja um atributo e a operação *empty* uma função, o usuário não precisa ter conhecimen
destes detalhes.

Um outro aspecto preocupante da classe *LINKED-LIST1* é a presença de redundâncias significativas n
operações, por exemplo: *value* e *insert* contêm basicamente o mesmo ciclo, e ciclos similares devem s
incluidos nas demais operações não mostradas. Para uma abordagem que enfatiza o re-uso, este méto
como está implementado não é a melhor opção. Note que este problema é um problema de implementaçã
interno à classe, mesmo assim constitui um fato representativo em um problema de natureza mais sér
que é a interface da classe.

Considere por exemplo a operação *search* , ela retorna o índice no qual um dado elemento foi encontra
na lista, ou senão `nb-element+1` se o elemento não está presente na lista. Como o usuário utiliza es
informação? Provavelmente, ele deve querer efetuar alguma inserção ou remoção na posição encontrad
Mas para qualquer uma destas operações ele deve percorrer a lista desde o início. Em projetos usan
programação orientada por objetos [11], isto é inaceitável. Como resolver este problema? Existe pe
menos duas formas de solucioná-lo.

- Reescrever a operação *search* para que ela retorne a posição corrente *LINKABLE* para a célu
  onde o valor necessário está e não um indice.
- Prover primitivas na classe para permitir a combinação de várias operações, por exemplo, *search*
  então *insert*.

Contudo a primeira solução derruba toda a noção de encapsulamento de estruturas de dados em class
o usuário poderia manipular diretamente as representações, com todos os perigos envolvidos. Com
mencionamos acima, a noção de *LINKABLE* é interna à implementação. Não faz sentido usar clas
para a abstração de dados se o indivíduo tem conhecimento dos apontadores e células da lista. Ele de
apenas pensar em termos de listas e valores de listas.

A segunda solução foi usada na biblioteca de Eiffel [7]. Muito embora esta solução tenha conseguid
manter a representação interna escondida do usuário, ela não teve muito sucesso devido a quantidade
variações de operações envolvidas nas primitivas. Veja os exemplos:

insert-before-by-value
insert-after-by-value
insert-after-by-position
insert-before-by-position
delete-after-position
...

Além deste aspecto, escrever componentes de *software* para ser reusado é uma tarefa muito difícil e n
há garantias de que os módulos estarão prontos para o reuso logo após a primeira implementação. Pa
tornar a tarefa ainda mais árdua, todas as operações básicas são complexas, contendo ciclos simila
aqueles presentes em *insert* . Felizmente existe um outro tipo de solução para este problema. Esta soluç
envolve olhar o tipo abstrato de dados por um ângulo diferente, e será apresentado na próxima seção.

4 Trabalho Proposto

Nos dois métodos apresentados, o maior problema está no modo de tratar listas. Neles, uma lista é vis
como um receptáculo passivo contendo informações. Para proporcionar ao usuário um produto m
adequado à suas aplicações é necessário tornar listas mais ativas, ou seja, fazer com que ela se "lembre
da última operação efetuada. A filosofia do trabalho proposto é olhar os objetos como máquinas co
estados internos e introduzir procedimentos ou comandos que mudam o estado, e funções, ou *quer*
nos estados. Esta abordagem produz uma interface que além de simples é mais eficiente que as outr
mostradas neste texto.

Uma lista será uma máquina com um estado que pode ser trocado explicitamente. O estado de uma lis
inclui o conteúdo da lista e também uma posição corrente ou *cursor*. A partir desta definição podem
prover o usuário com comandos para mover o cursor.

ed2-projeto-ic3.eps

Nesta nova abordagem, exemplos de comandos que podem movimentar o cursor incluirão operaçõ
como *search* , mas a mesma será um procedimento e não uma função. Portanto *search* não retorna
um resultado, mas simplesmente moverá o cursor para a posição onde o elemento pesquisado está. Sen
mais precisa, *l.search (x, i)* deverá mover o cursor para a i-ésima ocorrência de x na lista *l*.

Como no tipo abstrato de dados lista mostrado anteriormente, outros comandos podem ser definidos pa
atuar no cursor:

- Procedimento *start:* move o cursor para a primeira posição; uma pre-condição necessária é que
  lista não esteja vazia.
- Procedimento *finish:* possui a mesma pre-condição do procedimento *start* mas move o cursor pa
  a última posição da lista.
- Procedimento *back:* move o cursor para a posição anterior na lista, ou seja, para *previous*.
- Procedimento *forth:* move o cursor para a posição posterior na lista, ou seja, para *next*.
- Procedimento *go:* move o cursor para a uma posição específica na lista.

A posição do cursor é dada por uma função `query` *position* , a qual pode, na prática, ser implementa como um atributo. Outras funções `query` importantes sobre o cursor que retornam valores `true` ou `fal` são: *isfirst* e *islast*.

No esquema proposto os procedimentos para construir, modificar uma lista, inserir, substituir um val remover são simplificadas porque eles não têm que se preocupar com posições. Eles simplesmente atuar nos elementos nas posições corrente do cursor. Por exemplo, o procedimento para remover (*delete*) n será invocado como *l.delete(i)* e sim como *l.delete* , o qual irá remover o elemento na posição corre do cursor. Note que em cada uma destas operações, deve ser estabelecido uma convenção precisa sob o que acontece com o cursor após a operação.

Nossa proposta para estas convenções são:

- Procedimento *delete* : (sem argumentos), remova o elemento na posição do cursor e posicione cursor no seu vizinho à esquerda, ou seja, o atributo *position* será decrementado de 1.
- Procedimento *insert-right (v:T)* : insere um elemento de valor v à direita do cursor e não mud cursor. O atributo *position* permanece o mesmo.
- Procedimento *insert-left (v:T)* : insere um elemento de valor v à esquerda do cursor e não mud cursor. Neste caso, o atributo *position* deve ser incrementado de 1.
- Procedimento *chang-value (v:T)* substitui o valor do elemento na posição do cursor. O valor de elemento é fornecido pela função `query` *value*, a qual não possui parâmetros e pode ser implementa como um atributo.

Para definir uma interface como a que estamos propondo, baseada na filosofia de estados, é essenc introduzir asserções apropriadas para garantir que o estado está sempre bem definido. Por exemp suponha que o cursor esteja no primeiro elemento da lista, o que aconteceria se uma operação *del* fosse executada? A convenção que apresentamos acima, nos diz para mover o cursor para o vizinho esquerda, mas neste exemplo não existe mais vizinho. Pensando neste caso e em outros que aparecer nos leva a seguinte convenção: *permita ao cursor ir além dos limites da lista no máximo uma posiç esquerda ou direita. Assim o efeito de todas as operações em lista serão definidas da mesma forma.*

Esta propriedade é tipicamente uma *representação invariante*[1].

Neste caso a classe "invariante" inclui a propriedade: **0 ≤ position; position ≤ nb-elements+1** e funções `query` *offleft* e *offright* permitirão que o usuário determine se o cursor está fora dos limites.

A lista vazia também introduz uma nova cláusula na classe invariante: *empty = (offleft* **and** *offrigh* Como esta é uma igualdade de valores booleanos, o sinal "=" deve ser lido como "se e somente se" .

[1]Uma representação invariante expressa a consistência da representação dada pela classe, `vis-à-vis` o tipo abstrato dados em consideração, mesmo que este tipo abstrato de dados não esteje explicitamente especificado.

4.1 O que é Visível ao Usuário

No esquema proposto na seção anterior, as operações "*search* e então *insert* " são feitas através de du
chamada consecutivas, mas sem perda de eficiência. Por exemplo, se *LINKED-LIST1* é o nome da clas
no esquema proposto, o usuário pode efetuar um *search* e então *insert* :

l: LINKED-LIST[INTEGER]; m, n: INTEGER;
. . .
l.search(m,l);
if not *l.offright* **then** *l.insert-right(n)* **end**

Para remover a terceira ocorrência de um certo valor, o usuário irá executar:

l.search (m,3);
if not *l.offright* **then** *l.delete* **end**.

Para inserir um valor na posição i, o usuário irá executar:

l.go (i);
l.insert-left (i)

e assim por diante. Com este esquema, deixando o estado interno explícito e provendo ao usuário
comandos apropriados e `queries` neste estado, podemos obter uma interface fácil e limpa.

4.2 O que é Interno

A solução proposta produz uma boa interface, além disto ela simplifica consideravelmente a imp
mentação. Todas as redundâncias apontadas anteriormente são removidas. Isto porque, os procediment
agora possuem uma especificação mais restrita, concentrando somente em uma tarefa. Por exemplo,
operações para inserir e remover não precisam mais percorrer toda a lista, todas as modificações são feit
localmente. Outras operações, como por exemplo, *back, forth, go, search* já têm o cursor posiciona
no lugar correto. As redundantes travessias atribuidas aos anéis na Seção 3.2 já não são necessárias.
Figura 4 mostra um instante do estado da lista neste esquema e a Figura 5 ilustra uma remoção.

ed2-projeto-ic4.eps

ed2-projeto-ic5.eps

4.3 Ferramentas Utilizadas

A biblioteca será implementada nos ambientes de programação TOOL e C++.

Tool [8] é um sistema de desenvolvimento de *software* baseado no paradigma de programação orienta por objetos. Ele funciona em diferentes ambientes, Microsoft Windows v.3.1, Microsoft DOS v.3.0 versões mais recentes, PC baseados em microprocessadores INTEL 386 ou versões mais recentes. Necess 4MB de memória e 20MB de disco rígido.

TOOL incorpora uma linguagem, TOOL, simples e pequena, com uma boa interface, especialme projetada para uso em aplicações que manipulam dados. A linguagem é orientada por objetos e pos classes, herança, polimorfismo e outras características inerentes à esta técnica de desenvolvimento programas.

Faz parte da ferramenta TOOL uma grande biblioteca de classes *ready-to-use* como por exemplo, edito de textos entre outras que podem ser facilmente incorporadas na aplicação do usuário.

Como TOOL é uma linguagem extensível, é possível construir sua própria classe e incorporá-la no a biente de trabalho proposto. O ambiente de programação é integrado. Portanto é possível editar vár programas ao mesmo tempo, usando um editor que conhece a sintaxe da linguagem e pode mostrar tela este programa em cores. Sem deixar o ambiente é permitido, compilar, construir, depurar e dispa programas; melhorando assim o processo de construção da aplicação.

C++ [9] é uma linguagem de programação de uso geral baseada na linguagem C [4]. Além das fac dades fornecidas em C, as principais características incluidas em C++ são: tipos fortes e estáticos, tip abstratos de dados (classes), suporte para a programação orientada a objetos: classes, objetos e heran etc.

4.4 Cronograma de Desenvolvimento

A duração deste trabalho está prevista por um ano, agosto/1995 até julho/1996. Seu cronograma abra as seguintes etapas:

1. Estudo das ferramentas TOOL, Visual Basic [10], C++.
2. Definição da interface das classes.
3. Especificação do projeto de implementação das classes.
   (a) em Tool;
   (b) em C++.
4. Produção da documentação:
   (a) manual do usuário para a implementação em Tool;
   (b) manual do usuário para a implementação em C++.

5 Metodologia de Acompanhamento

Na primeira etapa deste projeto o aluno se dedicará ao estudo das ferramentas TOOL, C++ e Visu basic. Este estudo será dirigido pelo professor orientador em duas reuniões semanais com a duração uma hora cada uma.

As demais fases do projeto serão acompanhadas pelo professor orientador em uma reunião semar com a duração de 1:30h aproximadamente durante todo o período de duração do projeto. O objeti deste esquema é permitir ao bolsista a possibilidade de desenvolver os estudos e trabalhos de forr independente, contribuindo assim para sua formação.

6 Conclusão

Este trabalho terá como resultado além das bibliotecas de classe para TOOL e C++, para serem usad nos cursos de algoritmos e estruturas de dados, engenharia de software, etc. do departamento de ciên da computação, dois manuais do usuário, um para cada ferramenta utilizada na confecção das bibliotec Eles serão publicados na forma de monografias.

7 Bibliografia

References

[1] Ziviani, N. *Projeto de Algoritmos com Implementação em Pascal e C*, Editora Pioneira, 1992.

[2] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *Data Structure and Algorithms*, Addison-Wesley, 198

[3] Horowitz, Ellis and Sahni, Sartaj., *Fundamentals of Data Structures* Sixth Printing - Compu Science Press, Inc., 1976.

[4] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, second edition - Prent Hall-Software Series, 1988.

[5] Knuth, D., *The Art of Computer Programming, Volume 1: Fundamental Algoritms*, Addison-Wesl Second Edition, 1973.

[6] Knuth, D., *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesl Second Edition, 1973.

[7] Meyer, Bertrand, *Object-oriented Software Construction*, Prentice-Hall International Series in Co puter Science, C.A.R. Hoare Series Editor, 1988.

[8] TOOL – The Object Oriented Language – Programming for $Windows^{TM}$ MADE EASY, Com
cializado por SPA 1995.

[9] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley Publishing Company, Seco
Edition, 1991.

[10] Torgerson, Thomas W., *Visual Basic Professional 3.0 Programming*, Wiley-Oed Enterprise Comp
ing, 1994.

[11] Yourdon, Edward, *Object-Oriented Systems Design – An Integrated Approach*, Yourdon Press Co
puting Systems, 1994.

[12] Wirth, N., *Algorithms and Data Structures*, Prentice-Hall, 1986.

[13] Wirth, N., *Algoritmos e Estruturas de Dados*, Prentice-Hall do Brasil LTDA,1989.