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

Projeto de Iniciação Científica

Francisco Sapori Júnior

Orientadora: Mariza Andrade da Silva Bigonha

DCC - ICEX - UFMG

Resumo

O objetivo deste projeto é prover uma biblioteca de classes para implementar as estruturas de dados básicas, lista linear, pilha, fila e árvore como tipo abstrato de dados, usando as linguagens C^{++} , TOOL e Ita. A filosofia a ser adotada baseia-se no conceito de máquina de estados, isto é, objetos são vistos como máquinas contendo estados internos, procedimentos ou comandos são introduzidos para mudarem o estado. Esta abordagem baseia-se nas idéias propostas em “Object Oriented Constructions”, contudo produz uma interface mais simples e mais eficiente.

Sumário

1. Motivação

Um Tipo Abstrato de Dados (TAD) pode ser definido como um modelo matemático com um conjunto de operações definidas sobre o modelo. Por exemplo, o conjunto de inteiros mais as operações de adição, subtração, etc, caracterizam um tipo abstrato de dados.

Existem diversas vantagens em se utilizar Tipos Abstratos de Dados (TADs). Dentre elas, enumeramos algumas que consideramos as mais importantes. Com TADs pode-se:

1) Permitir que se "esconda" como uma determinada estrutura é˝implementada.

2) Permitir que, no caso de se alterar a implementação de uma determinada estrutura de dados, as interfaces, ou seja, as declarações dos cabeçalhos dos procedimentos e funções não sejam alteradas.

3) Permitir uma maior reutilização de programas.

4) Permitir o aumento da modularidade, organizando programas maiores e mais complexos de uma forma ordenada e controlada.

Estas vantagens ficam evidentes à medida que mais programas, sejam eles pequenos ou grandes, simples ou complexos, são desenvolvidos e têm reflexo direto na qualidade do programa final.

2. Objetivos

O objetivo deste projeto é prover bibliotecas de classes para implementar as estruturas de dados básicas [5, 6, 1, 12, 13], listas lineares e árvores como tipo abstrato de dados. A filosofia a ser adotada baseia-se no conceito de máquinas de estados [7]. Nesta abordagem, o projeto de estruturas de dados como estados explícitos permite uma forma fácil de documentar interfaces. Os tipos mais comuns destas estruturas são caracterizados por estados internos e uma posição˝corrente.

O ambiente a que se destina as bibliotecas de classes projetadas são: C^{++} [9,10], TOOL [8], Ita [14], e portanto as mesmas foram implementadas em˝C^{++}, TOOL e Ita.

O enfoque de máquinas de estados pode, a primeira vista, contradizer a "abstração" da abordagem tipo abstrato de dados, na qual a programação orientada é baseada. Mas não é verdade. A teoria de tipos abstratos de dados sugere que estruturas de dados devam ser dadas por uma descrição abstrata baseada em operações aplicáveis e nas propriedades formais destas operações. Isto de forma alguma caracteriza estruturas de dados como um simples local para armazenar os dados. É exatamente o contrário: a introdução de estados e operações sobre o estado, torna a especificação do tipo abstrato de dados mais poderosa à medida que ele possui mais funções e propriedades. É bom salientar que estado, neste contexto, é uma abstração e nunca será acessado diretamente, somente manipulado através de comandos e queries.

3. Abordagens Existentes

3.1. Implementação de TADs usando Linguagem tipo Pascal

Existem, na literatura, algumas propostas de implementação de estruturas de dados básicas como tipo abstrato de dados. Uma delas [1] mostra uma implementação usando uma linguagem de programação procedural, Pascal, para atingir seu objetivo. Esta abordagem apresenta alguns problemas sérios. Tomemos como exemplo o tipo abstrato de dados Lista. Para criar um tipo abstrato de dados Lista, é necessário 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 cópia da lista linear.
8) Ordenar dois 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, veremos que, para o usuário utilizar este TAD Lista, ele tem que ter pleno conhecimento dos detalhes de representação de tipos declarados. Este fato não está associado com a linguagem usada, mas sim com a forma como o TAD foi definido. Neste exemplo, o usuário tem que saber que o registro TipoItem deve ter um campo Chave e que é permitido acrescentar *outros componentes* no mesmo. O item outros componentes é dependente do programa do usuário. Este fato contrasta com as principais vantagens sobre o uso de TADs citadas na Seção 1, pois permite que o indivíduo insira outros componentes dentro da declaração de tipos. É fato que linguagens tipo Pascal não oferecem o recurso para implementar TADs. Tudo fica em aberto. Contudo, muito embora a implementação seja um pouco complicada, ela poderia ser feita de tal forma que a informação dentro da declaração de tipos não˝fosse usada.

3.2. Implementação de TADs usando Linguagens Orientadas por Objetos

Outra abordagem [7], presente na literatura, utiliza-se dos conceitos de programação orientada por objetos (LOO) para implementar uma classe que engloba as estruturas de dados básicas como tipo abstrato de dados.

Exploraremos mais esta abordagem por estar mais próxima da proposta de nosso trabalho. Por questão de clareza, vamos apresentá-la por meio de um exemplo usando a linguagem Eiffel [7] orientada por objetos. Mas antes vamos estabelecer algumas convenções:

1) O exemplo define uma classe de listas lineares como representações de seqüências, onde cada elemento de uma seqüência, denominado célula, é chamado de *Linkable*. Cada *Linkable* contém um valor e um apontador para outro elemento conforme ilustra a Figura 1.

2) Value representa o tipo de valores dado por T.

3) A lista é representada por uma célula separada chamada cabeça. Esta célula contém o endereço da primeira célula *Linkable*. Esta pode, também, possuir outros itens como por exemplo, um contador do número de elementos da lista (*nb-elements*).

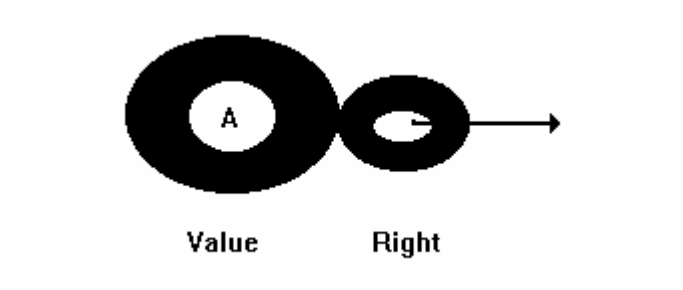

Figura 1 - Uma célula

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

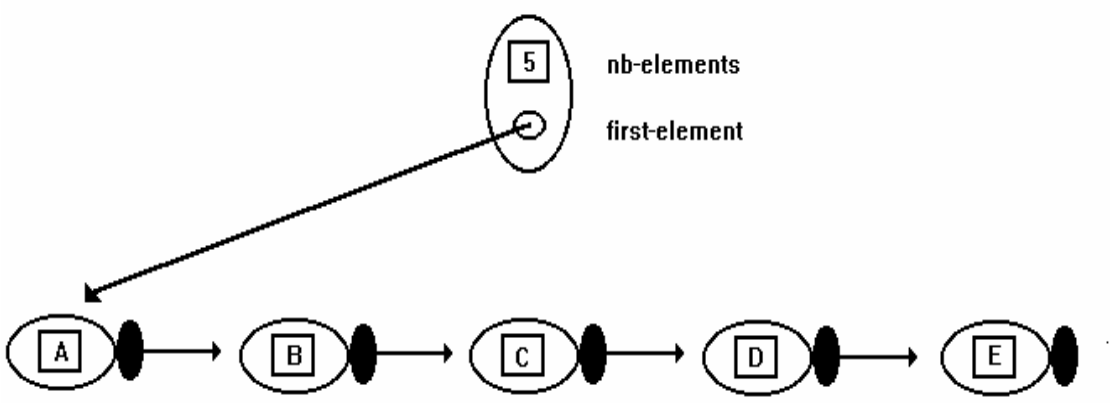

Figura 2 - Lista Encadeada

```
class LINKABLE[T]
        células LINKABLE para serem usadas como 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 do registro. A seguir, é mostrado como seria definido as classes *LINKABLE* e *LINKED-LIST1* nesta abordagem.

```
class LINKABLE[t]
        células LINKABLE para serem usadas como listas lineares.
export veja abaixo
features
        value: T;
        right: LINKABLE[T]
        Create (initial: T) is
                inicializa com 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, 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 1 ≤ 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 o valor do i-ésimo elemento na lista.
                require 1 ≤ i; i ≥ nb-element
                do ... ensure value(i) = v end;
        insert (i: integer) is
                insere um novo elemento do valor v de tal forma que
                ele fique sendo o i-ésimo elemento da lista.
                require 1 ≤ i; i ≥ nb-element + 1
                local previous, new: LINKABLE[T]; j: integer
                        do - cria uma nova célula.
                        new.Create(v);
                        if i = 1 then
                        - insere no início 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 = 1 loop
                                j:= j + 1; previous:= previous.right end
                        nb-element:= nb-element + 1
                ensure nb-element:= old nb-element + 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 uma lista, 0 se não houver. ... end
        outras features
        invariant empty = (nb-element = 0) empty = first-element.Void
end class LINKED-LIST1
```

A vantagem desta representação é que tanto a inserção quanto a remoção é rápida se houver um apontador para a célula *Linkable* imediatamente à esquerda do ponto de inserção ou remoção. Por outro lado, esta representação não é muito boa para pesquisar por um elemento dado o seu valor de posição, pois estas operações requerem um caminhamento seqüencial na lista.

Existem duas estruturas de dados que são usadas normalmente para se implementar listas: arranjos e apontadores. Os arranjos são bons para acessar uma posição, mas ruins para inserções e retiradas.

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

Um outro aspecto preocupante da classe *LINKED-LIST1* é a presença de redundâncias significativas nas operações: *value* e *insert*, elas contêm basicamente o mesmo ciclo, e ciclos similares devem ser incluídos nas demais operações não mostradas. Para uma abordagem que enfatiza o reuso, este método, como está implementado, não é a melhor opção. É bom lembrar que este problema é um problema de implementação, interno à classe, mas, mesmo assim, constitui um fato representativo em um problema de natureza mais séria, que é a interface da classe.

Considere, por exemplo, a operação *search* que retorna o índice no qual um dado elemento foi encontrado na lista ou nb-element + 1 se o elemento não está presente na lista. Como o usuário utiliza esta informação?˝Provavelmente, ele deve efetuar alguma inserção ou remoção na posição encontrada. Mas, para qualquer uma destas operações, ele deve percorrer a lista deste o início. Em projetos usando programação orientada por objetos [11], isto é inaceitável. Como resolver este problema? Existe pelo menos duas formas de solucioná-lo:

1) Rescrevendo a operação *search* para que ela retorne a posição corrente *LINKABLE* para a célula onde o valor necessário está e não um˝índice.

2) Prover primitivas na classe para permitir a combinação de várias operações, por exemplo, *search* e então *insert*.

Contudo, a primeira solução derruba toda a noção de encapsulamento de estruturas de dados em classes: o usuário poderia manipular diretamente as representações, com todos os perigos envolvidos. Como mencionamos acima, a noção de *LIKABLE* é interna a implementação. Não faz sentido usar esta classe para a abstração de dados se o indivíduo tem conhecimento de apontadores e células na lista. Ele deve pensar em termos de listas e valores de listas.

A segunda solução foi usada na biblioteca Eiffel [7]. Muito embora esta solução tenha conseguido manter a representação interna escondida do usuário, ela não teve muito sucesso devido a quantidade de variações de operações evolvidas nas primitivas. Veja alguns 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 não há garantias de que todos os módulos estarão prontos para o reuso logo após a primeira implementação. Para tornar a tarefa ainda mais árdua, todas as operações básicas são complexas, contendo ciclos similares aqueles presentes em *insert*. Felizmente, existe um outro tipo de solução para este problema. Esta solução

envolve olhar o tipo abstrato de dados por um ângulo diferente, e será apresentado na próxima seção.

4. Trabalho Proposto

Nos dois métodos apresentados [1, 7], o maior problema está no modo de tratar listas. Neles, uma lista é vista como um receptáculo passivo contendo informações. Para proporcionar ao usuário um produto mais adequado à suas aplicações, é necessário tornar listas mais ativas, fazendo com que elas se "lembrem" da última operação efetuada. A filosofia do trabalho proposto é olhar os objetos como máquinas contendo estados internos e introduzir procedimentos ou comandos para mudarem o estado. Ela baseia-se nas idéias encontradas em [7]. Esta abordagem produz uma interface simples e é mais eficiente que as outras mostradas neste texto.

Uma lista agora é uma máquina com um estado que pode ser trocado explicitamente. O estado de um lista inclui o conteúdo da lista e também uma posição corrente do *cursor*. A partir desta definição, podemos prover o usuário com comandos para mover o cursor.

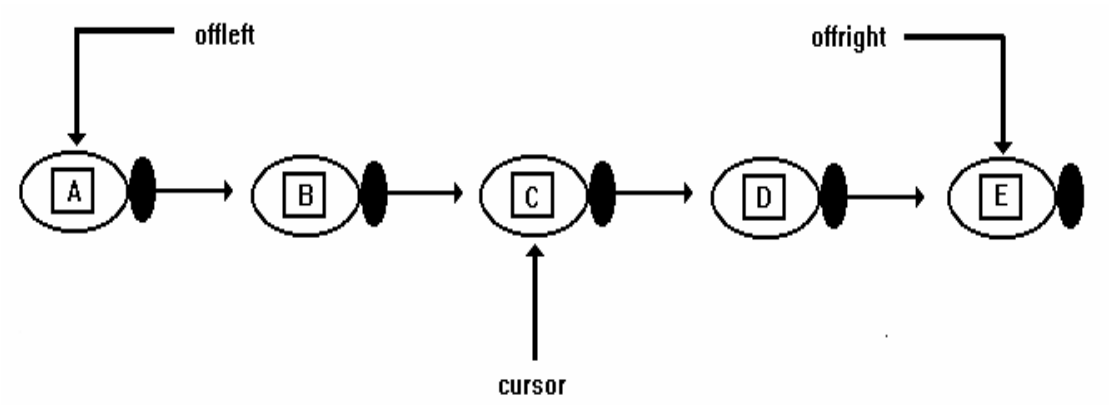

Figura 3 - Lista com cursor

Nesta abordagem, exemplos de comandos que podem movimentar o cursor incluem operações como *search*. É interessante notar que como *search* é um procedimento e não uma função. Ele não retornará um resultado, mas simplesmente moverá o cursor para a posição onde o elemento pesquisado está. Exemplo, *l.search(x,i)* move o cursor para a i-ésima ocorrência de *x* na lista *l*.

Como no tipo abstrato de dados lista mostrado na Seção 3, outros comandos podem ser definidos para atuar no cursor:

- Procedimento *start*: movimenta o cursor para a primeira posição, uma pré-condição necessária é que a lista não esteja˝vazia.

- Procedimento *finish*: possui a mesma pré-condição do procedimento *start*, mas movimenta o cursor para a última posição da lista.

- Procedimento *back*: movimenta o cursor para a posição anterior na lista, ou seja, para *previous*.

- Procedimento *forth*: movimenta o cursor para a posição posterior na lista, ou seja, para *next*.

- Procedimento *go*: movimenta o cursor para uma posição específica na˝lista.

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

No esquema proposto, os procedimentos para construir, modificar uma lista, inserir um valor, remover são simplificadas porque eles não têm que se preocupar com posições. Eles atuam nos elementos nas posições corrente do cursor. Por exemplo, o procedimento para remover, *delete*, não será invocado como *l.delete(i)* e sim como *l.delete*, que remove o elemento na posição corrente do cursor. Note que, em cada uma destas operações, deve ser estabelecido uma convenção precisa sobre o que acontece com o cursor após a operação. Nossa proposta para estas˝convenções são:

- Procedimento *delete*: (sem argumentos) remove o elemento na posição do cursor e posiciona o cursor no seu vizinho à direita, ou seja, o atributo *position* é decrementado de 1.

- Procedimento *insert-right(v: T)*: insere um elemento de valor v à direita do cursor e não muda o cursor. O atributo *position* permanece o mesmo.

- Procedimento *insert-left(v: T)*: insere um elemento de valor v à esquerda do cursor e não muda o cursor. Neste caso, o atributo *position* deve ser incrementado de 1.

- Procedimento *change-value(v: T)*: substitui o valor do elemento na posição do cursor. O valor deste elemento é fornecido pela função query *value*, a qual não possui parâmetros e pode ser implementada como um atributo.

Para definir uma interface, baseada na filosofia de estados, é essencial introduzir asserções apropriadas para garantir que o estado esteja sempre bem definido. Por exemplo, suponha que o cursor esteja no primeiro elemento da lista, o que aconteceria se uma operação *delete* fosse executada? A convenção apresentada acima nos diz para mover o cursor para o vizinho à esquerda, mas neste exemplo, não existe mais vizinho. Pensando neste caso e em outros que aparecerão, adotou-se a seguinte convenção: *é permitido ao cursor ir além dos limites da lista no máximo uma posição, esquerda ou direita. Com isto, o efeito de todas as operações em lista serão definidas da mesma forma.*

Esta convenção é tipicamente uma representação invariante. Uma representação invariante expressa a consistência da representação dada pela classe, vis-à-vis o tipo abstrato de dados em consideração, mesmo que este tipo de dados não esteja explicitamente especificado.

Neste caso, a classe "invariante" inclui a propriedade: $0 \leq$ position; position $\leq$ nb-element + 1 e as 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 *offright*). Como esta é uma grande igualdade de valores "booleanos", o sinal "=" deve ser lido como "se e somente se".

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

4.1. Ferramentas Utilizadas

A biblioteca será implementada nos ambientes de programação TOOL [8], C++[9, 10] e Ita [14]

4.1.1. TOOL

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

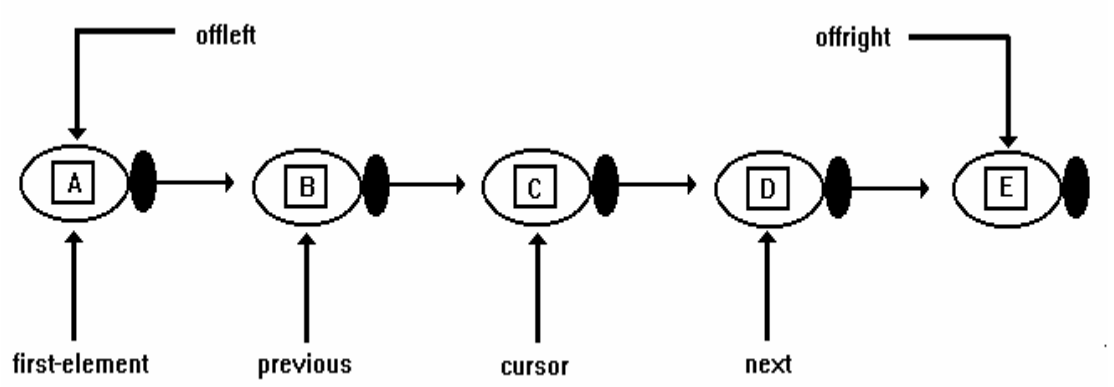

Figura 4 - Lista com cursor

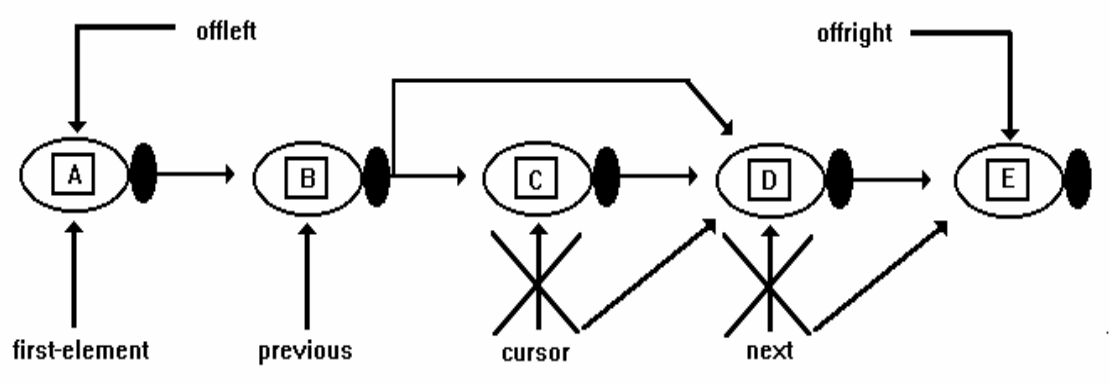

Figura 5 - Remoção em uma lista encadeada com cursor

TOOL incorpora uma linguagem, TOOL, simples e pequena, com uma boa interface, especialmente projetada para uso em aplicações que manipulam dados. A linguagem é orientada por objetos e possui classe herança, polimorfismos e outras características inerentes à esta técnica de˝desenvolvimento de programas.

Faz parte da ferramenta TOOL uma grande biblioteca de classes *ready-to-use* como, por exemplo, editores 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 ambiente de trabalho proposto. O ambiente de programação é integrado. Portanto, é possível editar vários programas ao mesmo tempo, usando um editor que conhece a sintaxe da linguagem e pode mostrar na tela este programa em cores. Sem deixar o ambiente é permitido compilar, construir, depurar e disparar programas, melhorando assim o processo de construção da˝aplicação.

4.1.2. C++

C++ [9, 10] é uma linguagem de programação de uso geral baseada na linguagem C [4]. Além das facilidades fornecidas em C, as principais˝características incluídas em C++ são: tipos fortes e estáticos, tipos abstratos de dados (classes), suporte para a programação orientada por objetos: classes, objetos e˝herança, etc.

4.1.3. Ita

Ita [12] é uma linguagem de programação inspirada em C, cujo objetivo é lhe introduzir conceitos de programação orientada por objetos,˝como herança, polimorfismo, adotando-se a filosofia de programação Eiffel, porém no estilo de C++. Estes conceitos incentivam a produção de *software* com alto grau de reusabilidade. Além de reusabilidade, incentiva-se também a produção de *software* correto, isto é, de acordo com sua especificação, através de um estilo de programação por contrato. Dentre os recursos de Ita que procuram suportar programação por contrato estão pré e pós-condições, invariantes e tratamento de exceções. Objetos em Ita são tratados uniformemente por meio de uma semântica de referência, por exemplo, no caso da passagem de parâmetro, assim como em C, é passado por valor. Nos caso de parâmetros do tipo classe, passa-se, por valor, uma referência para um˝objeto.

5. Biblioteca de Classes como Máquinas de Estados

5.1. Implementação no Ambiente de Programação C++

Esta seção mostra o desenvolvimento de bibliotecas de classes: lista encadeada, pilha, fila, árvore binária, usando o ambiente de˝programação C++.

5.1.1. Lista

Como proposto na Seção 4, uma lista será uma máquina com um estado que pode ser trocado explicitamente. Dentro desta filosofia, é possível torná-la capaz de lembrar da última operação efetuada.

5.1.1.1. Representação das Estruturas de Dados

Para esta implementação, foram definidas duas estruturas˝básicas:

1. Classe Cell: tipo da célula que compõe a lista duplamente encadeada. Nela, estão definidas:

    a) a posição da célula que está à direita, Next.
    b) a posição da célula que está à esquerda, Previous.
    c) um valor, Value, que pode ser inteiro real, etc.

2. Classe LinkList: possui as seguintes funções:

    a) guardar a posição da primeira célula da lista, Left.
    b) guardar a posição da célula corrente, Cursor.
    c) armazenar o número de células que compõem a lista, Nb_Elements.
    d) armazenar o número da célula para onde o Cursor está apontando, Current.

Estas estruturas são representadas como uma lista duplamente encadeada. A especificação em C^{++} das classes Cell e LinkList são apresentadas a seguir e a Figura 6 mostra a forma interna da lista.

```
template <class V> class Cell{

        friend class LinkList<V>;

        Cell<V> *Next;              // Next -> aponta para a˝próxima célula.
        Cell<V> *Previous;          // Previous -> aponta para a˝célula anterior.
        V Value;                    // Value -> elemento da célula do tipo V.


public:

                Cell::Cell(V Element);
};
```

```
template <class V> class LinkList{

        Cell<V> *Left,              // Left (Aponta para primeira˝célula da lista)
                *Cursor;        // Cursor (Posicão corrente na lista)
        int Nb_Elements;
        int Current;



        public:

        // Operações:

        LinkList();
        int Is_Ok(int Position);
        int Go(int Position);
        int Back();
        int Forth();
        void InsertRight(V Element);
        void InsertLeft(V Element);
        void InsertFirst(V Element);
        int Insert(V Element,int Position);
        int DelLeft();
        int DelRight();
        int Del(int Position);
        void DelLast();
        int ChangeValue(V Element);
        void Start();
        void Finish();
        int NumberElements();
        V ReturnValue();
        int Search(V Element);
};
```

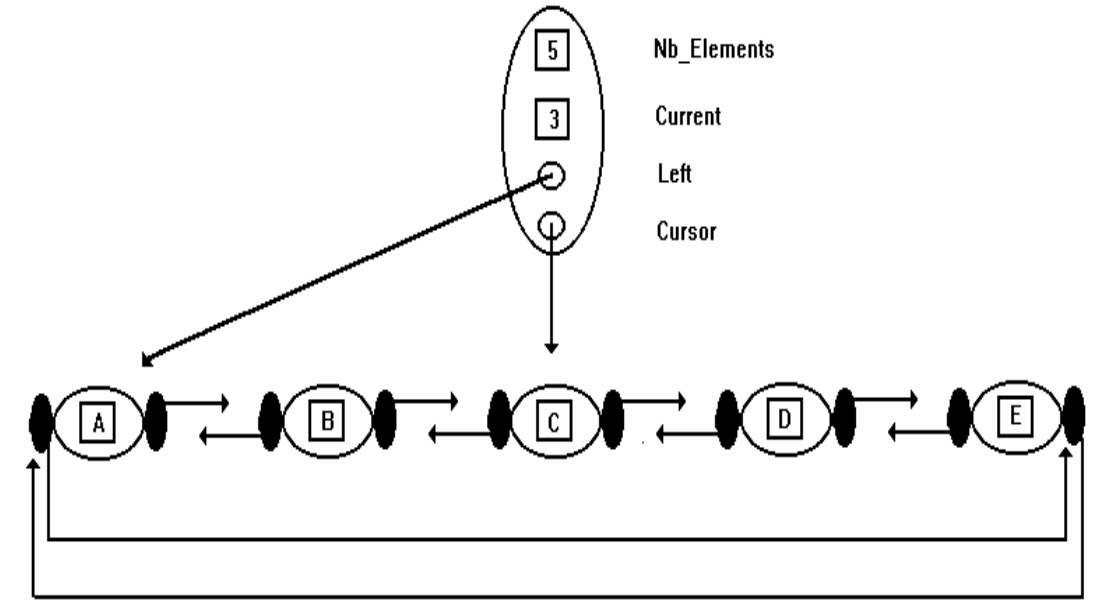

Figura 6 - Lista duplamente encadeada

5.1.1.2. Operações Oferecidas no TAD Lista

Classe Cell:

- Cell(V Element): função construtora responsável pela˝inicialização de Value.

Classe LinkList:

- LinkList(): função construtora cujo objetivo é inicializar a lista duplamente encadeada. Cursor e Left recebem NULL e Current e Nb_Elements˝recebem zero.

- int Is_Ok(int Position): operação cuja função é retornar verdadeiro se a posição de inserção de uma nova célula é válida, posição que estiver entre 1 e Nb_Elements + 1; ou falso, caso contrário.

- int Go(int Position): operação responsável pelo posicionamento do Cursor em uma determinada posição. Essa função retorna verdadeiro se esta operação for efetuada ou falso, caso isto não aconteça.

- int Back(): move o Cursor para a célula anterior. Mas isto só acontece se existir mais de um elemento na célula. Neste caso, retorna verdadeiro se esta função foi executada com sucesso ou falso, caso contrário.

- int Forth(): move o Cursor para a próxima célula. Mas isto só acontece se existir mais de um elemento na célula. Neste caso, retorna verdadeiro se esta função foi executada com sucesso ou falso, caso contrário.

- void InsertRight(V Element): insere um nova célula à direita da célula para onde o Cursor está apontando. Agora, para inserir o primeiro elemento na lista esta operação faz a chamada da operação InsertFirst.

- void InsertLeft(V Element): insere um nova célula à esquerda da célula para onde o Cursor está apontando. Agora, para inserir o primeiro elemento na lista esta operação faz a chamada da operação InsertFirst.

- void InsertFirst(V Element): insere a primeira célula que irá compor a lista. Uma particularidade desta operação é que ela não pode ser usada diretamente pelo usuário, pois ela é interna a classe, isto é, ela só˝pode ser acessada por operações que estão definidas na classe.

- int Insert(V Element, int Position): inserção de uma nova célula em uma determinada posiçao da lista. O procedimento adotado foi, primeiramente, verificar se a posição onde se deseja colocar o novo objeto é válida. Se for, verifica se é desejado colocar a célula em uma posição diferente da primeira. Caso seja verdade, posiciona-se o Cursor na célula em uma posição anterior à desejada, usando a operação Go, e faz-se a inserção do novo elemento, usando a operação InsertRight. Se esta operação, como um todo, tiver sucesso retorna-se verdadeiro, ou falso caso˝contrário.

- int DelRight(): apaga a célula para onde o cursor está apontando e, depois, movimenta o cursor para a próxima célula, ou seja, célula que estava à direita da que foi eliminada. Esta operação só se realiza se existir pelo menos dois elementos na lista duplamente encadeada. Retorna verdadeiro se a operação foi realizada e falso, caso contrário.

- int DelLeft(): apaga a célula para onde o cursor está apontando e, depois, movimenta o cursor para a célula anterior, ou seja, célula que estava à esquerda da que foi eliminada. Esta operação só se realiza se existir pelo menos dois elementos na lista duplamente encadeada. Retorna verdadeiro se esta operação foi realizada e falso, caso contrário.

- void DelLast(): é responsável por certas atribuições quando se apaga o último elemento da lista.

- int Del(int Position): retira um elemento da lista em uma determinada posição especificada pelo usuário. Nesta operação, primeiramente, verifica se a posição especificada é válida. Se for, usa-se a operação˝DelRight. Note que poderia ser usada a operação DelLeft para a retirada deste objeto. Retorna verdadeiro se houve a retirada do elemento e falso, caso contrário.

- int ChangeValue(V Element): operação responsável pela alteração do valor da célula para onde o cursor aponta. Isto só acontece se existir, pelo menos, um elemento na lista. Retorna verdadeiro se houve a troca do elemento e falso,˝caso contrário.

- int Start(): operação cuja função é posicionar o cursor na primeira posição da lista caso haja algum elemento. Retorna-se verdadeiro se esta operação foi executada com sucesso e falso, caso contrário.

- int Finish(): operação cuja função é contrária a Start, pois posiciona o cursor na última posição da lista. Retorna-se verdadeiro se houve sucesso na execução desta operação e falso, caso contrário.

- int NumberElements(): função que retorna o número de˝elementos que há na lista.

- V ReturnValue(): função que retorna o elemento da˝célula apontada pelo cursor.

- int Search(V Element): verifica se um determinado elemento se encontra na lista. Caso ele esteja, retorna-se a sua posição, senão,˝retorna-se zero.

5.1.2. Pilha

Como se sabe, uma pilha é uma lista linear nas quais inserções e retiradas ocorrem sempre em um dos extremos da lista. As pilhas possuem a seguinte propriedade: o último item a ser inserido é o primeiro item que pode ser retirado - “last-in, first-out” [1].

Para definir a classe pilha, tem-se a classe LinkList, juntamente com algumas de suas operações.

5.1.2.1. Representação das Estruturas de Dados

```
template <class V> class Stack{

        LinkList<V> S;

public:

        void Push(V);
        int Pop(V&);
        void Print();
};
```

5.1.2.2. Operações Oferecidas no TAD Pilha

- void Stack::Push(V Element): operação usada para empilhar. Coloca o último elemento a ser inserido na primeira posição da lista. Usa-se a operação de inserção da Classe LinkList, Insert.

- int Stack::Pop::Push(V& Element): operação usada para desempilhar. Retira o elemento que está na primeira posição da lista. Retorna verdadeiro se existir pelo menos um elemento e falso, caso contrário. Usa as operações NumberElements, Go, ReturnValue e DelRight, nesta ordem.

5.1.3. Fila

Uma fila é uma lista linear em que todas as inserções são realizadas em um extremo da lista, e todas as retiradas são realizadas no outro extremo da lista. Por esta razão, as filas são chamadas de listas **fifo**, termo formado a partir de “first-in, first-out” [1].

Para definir a classse fila, tem-se a Classe LinkList, juntamente com algumas de suas operações.

5.1.3.1. Representação das Estruturas de dados

```
template <class V> class Queue{

        LinkList<V> Q;

public:

        void Insert(V);
        int Del(V&);
};
```

5.1.3.2. Operações Oferecidas no TAD Fila

- void Insert(V): inserção de um elemento na última posição da lista. Usa-se a operação Insert da Classe LinkList.

- V Del(): retirada do primeiro elemento da lista. Retorna-se verdadeiro se existir pelo menos um elemento e falso, caso contrário. Usa as operações NumberElements, Go, ReturnValue e DelRight da Classe LinkList.

5.1.4. Árvore Binária

Uma árovre binária é formada a partir de um conjunto finito de nodos, consistindo de um nodo raiz mais zero ou duas subárvores binárias distintas. Também definida como um conjunto de nodos onde cada nodo tem exatamente zero ou dois filhos, chamados de filho à esquerda e filhos à direita do nodo. A árvore binária tem a seguinte propriedade: os elementos dos nodos que estão á direita de algum nodo pai são maiores e os que estão à esquerda são menores [1].

5.1.4.1. Representação das estruturas de dados

Para esta implementação, tem-se duas estruturas˝básicas:

1. Classe Cell: é o tipo da célula que compõe a árvore binária. Nela, estão definidas:

    a) a posição da subárvore à esquerda, Left.
    b) a posição da subárvore à direita, Right.
    c) um valor, Value, que pode ser inteiro real, etc.

2. Classe Tree: tem as seguintes funções:

    a) guardar a posição do primeiro nodo da árvore, nodo raiz,˝Root.
    b) armazenar o número de nodos que compõem a árvore, Nb_Elements.

A Figura 7 mostra a forma interna da árvore binária e o código em C^{++} mostra sua especificação.

```
template <class V> class Cell{

        friend Tree<V>;

        Cell <V>*Left;          // Left -> aponta para a˝subárvore à esquerda.
        Cell<V> *Right;         //  Right -> aponta para a˝subárove à direita.
        V Value;            // Value -> elemento da celula do˝tipo V.


        // Operação

        Cell(V Element)
};
```

```
template <class V> class Tree{

        Cell<V> *Root;          // Aponta para o primeiro nodo da˝arvore.
        int Nb_Elements;    // Numero de elementos da arvore.

        public:

        Tree();
        int Insert(V);
        int Search(V);
        int Del(V);
};
```

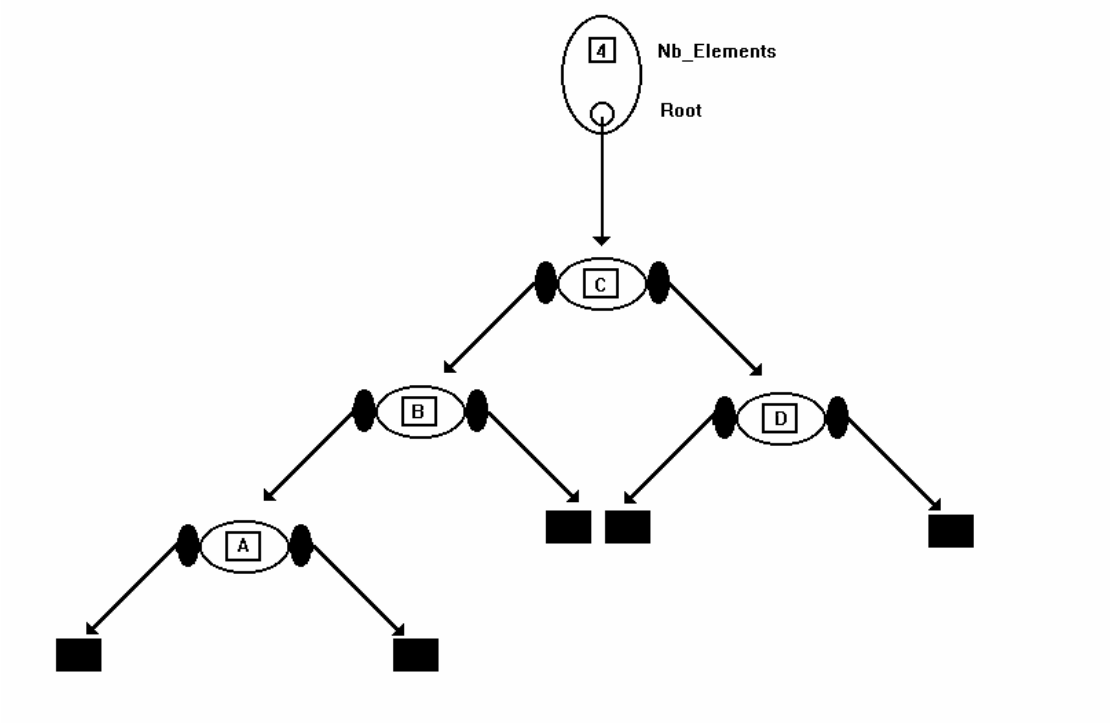

Figura 7 - Árvore Binária

5.1.4.2. Operações Oferecidas no TAD Árvore

Classe Cell:

- Cell(V Element): função construtora responsável pela˝inicialização de Value.

Classe Tree:

- Tree(): inicialização da árvore. Root recebe NULL e Nb_Elements é inicializado com 0.

- int Insert(V Element): operação responsável pela inserção de um nodo na ávore. Para inserir um elemento na árvore é necessário fazer uma pesquisa para determinar a localidade do novo nodo. Verifica se o elemento do novo nodo é menor ou maior que o elemento do nodo que está sendo pesquisado, começando

pelo nodo raiz. Se ele for menor, então caminha-se para a subárvore à esquerda. Se ele for maior, então caminha-se para a subárvore à direita. Este processo é repetido até que se encontra um nodo folha. Neste ponto pode ser constatado duas situações: o nodo é nulo, NULL ou o elemento a ser inserido já existe na árvore. Se for nulo, o novo nodo é inserido [1]. Esta operação retorna verdadeiro se houve a inserção de um novo nodo ou falso, caso este nodo já esteja presente na árvore.

- int Search(V Element): operação responsável pela procura de um determinado elemento na árvore binária. Verifica se o elemento é menor ou maior que um dado elemento da árvore começando pelo nodo raiz. Se ele for menor, então caminha-se na subárvore à direita. Caso contrário, caminha-se na subárvore à esquerda. Este processo é repetido até que se encontre o elemento procurado ou se encontre um nodo folha [1]. Ao encontrar o elemento procurado, a operação retorna verdadeiro e falso, caso contrário.

- int Del(V Element): operação responsável pela retirada de um determinado elemento da árvore. Usa-se o mesmo algoritmo da operação Search. Ao encontrar o elemento a ser retirado, verifica se o nodo onde ele está localizado possui algum descendente. Caso não tenha descendente, retira-se o nodo. Se houver um descendente, então este nodo passa a ser substituído pelo seu descendente. Se houver mais de um descendente, o elemento a ser retirado é substituído pelo elemento do nodo mais à direita na subárvore esquerda e depois este nodo mais à direita é retirado da árvore [1]. Se o elemento foi retirado da árvore, então a operação retorna verdadeira e falso, caso contrário.

5.2. Implementação no Ambiente de Programação TOOL

Devido a existência dos TADs Lista, Pilha, Fila na linguagem TOOL, implementou-se a Árvore Binária na linguagem TOOL. Para ela definiu-se duas estruturas e três operações básicas, Insert, Search e Del, com as mesmas características daquelas definidas na Seção 5.1.4.1. Foi constatado durante a inserção dos elementos na árvore que a partir do 3º nível, não se conseguia fazer a inserção corretamente dos elementos. Então, depois de muitos testes, chegou-se a conclusão de que a ferramenta não estava fazendo a alocação de memória corretamente para este problema especificamente. Portanto o usuário deve estar atento para este tipo de problema.

5.3. Implementação no Ambiente de Programação Ita

No ambiente de programação Ita, implementou-se somente a estrutura de dados árvore. As demais estruturas básicas, listas, pilhas e fila já foram implementadas e podem ser encontradas em [14].

5.3.1. Árvore Binária

5.3.1.1. Representação das Estruturas de Dados

Assim como na Seção 5.1.4.1 definiu-se duas estruturas básicas:

1. Classe Cell: é o tipo da célula que compõe a árvore binária. Nela, estão definidas:

    a) a posição da subárvore à esquerda, Left.
    b) a posição da subárvore à direita, Right.
    c) um valor, Value, que pode ser inteiro real, etc.

2. Classe Tree: tem as seguintes funções:

    a) guardar a posição do primeiro nodo da árvore, nodo raiz,˝Root.
    b) armazenar o número de nodos que compõem a árvore, Nb_Elements.

```
class Cell<V> friend Tree{


        Cell<V> Left;
        Cell<V> Right;
        V Value;

public:

        init(V Element);
        void insert_right (Cell <V> other);
        void insert_left (Cell <V> other);
        void Change(V Element);

}
```

```
class Tree<V>{

        Cell<V> Root;
        int Nb_Elements;

public:
        init()
        int Insert(V *Element);
        int Search(V *Element);
        int Del(V *Element);

}
```

5.3.1.2. Operações Oferecidadas no TAD Árvore

As operações oferecidadas e suas descrições são as˝mesmas da Seção 5.1.4.2.

6. Conclusão

Neste projeto, foi construída uma biblioteca de classes para implementar as estruturas de dados básicas, lista linear, pilha, fila e árvore usando o conceito de Tipo Abstrato de Dados. Isso é importante, pois o usuário dessas bibliotecas não precisa conhecer o funcionamento das estruturas de dados. Para ele só é necessário saber como usá-las. Para esta implementação usamos as linguagens C^{++}, TOOL e Ita, pois elas proporcionam o uso da Orientação por Objetos.

Estas estruturas foram devidamente implementadas em C^{++}, usando a ferramenta Borland C^{++} 4.0. Primeiramente, construímos a lista linear, optando por uma lista duplamentente encadeada. A vantagem do uso deste tipo de lista é que ela nos proporciona uma maior facilidade de caminhamento. O ponto negativo é o uso de mais apontadores acarrentando uma queda no número de elementos. Na próxima etapa, implementou-se a pilha e fila usando as operações da lista linear. Este fato demonstra a idéia de usar as estruturas definidas sem ter conhecimento de como elas funcionam, mas sim do que elas oferecem. Por último, criou-se a árvore binária, retirando totalmente sua recursividade. Essa remoção por um lado, faz com que o número de elementos da árvore aumente.

Este projeto previa também a implementação das estruturas de dados básicas em TOOL, mas devido a existência da implementação das estruturas lista linear, pilha e fila na biblioteca de TOOL, implementou-se somente a árvore binária. Com isto decidimos incluir em nosso projeto a implementação de árvore binária para complementar a biblioteca das estruturas de dados básicas em Ita. A dificuldade encontrada neste ponto está relacionado com a falta de manuais que proporcionaria uma maior rapidez no desenvolvimento destas estruturas. Um ponto negativo que gostaríamos de registrar foi a impossibilidade de transpor o 3^{o} nível da árvore implementada na linguagem TOOL, o que limita bastante o número de elementos na árvore.

7. Bibliografia

Referências

[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, 1993.

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

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

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

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

[7] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice-Hall International Series in Computer Science, C.A.R. Hoere Series Editor, 1988.

[8] TOOL - The Object Oriented Language - Programming for Windows™ MADE EASY, Comercializado por SPA, 1995.

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

[10] Pappas, Chris H. & Murray, Willian H. *Borland C++ 4.0*. McGraw-Hill Ltda, 1994.

[11] Yourdon, Edward, *Object-Oriented Systems Design - An Integrated Approach*, Yourdon Press Computing 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.

[14] A Linguagem de Programação Ita - Marco Túlio de Oliveira Valente e Roberto da Silva Bigonha, 1996.

Apêndice A: Implementação em C++

Lista Duplamente Encadeada

```
#include <alloc.h>
#include <iostream.h>

//---------------------------------//
//- Celula que compoe a lista -//
//---------------------------------//

template <class V> class Cell{

        friend class LinkList<V>;

        Cell<V> *Next;              // Next -> aponta para a proxima˝celula.
        Cell<V> *Previous;          // Previous -> aponta para a celula˝anterior.
        V Value;                    // Value -> elemento da˝celula do tipo V.


public:

                Cell::Cell(V Element)
                {
                        Value = Element;
                }

};


#include <alloc.h>
#include <iostream.h>

//-------------------------------------------------------------------------//
//- Classe definida para aplicar operacoes em uma lista encadeada˝-//
//-------------------------------------------------------------------------//

template <class V> class LinkList{

        Cell<V> *Left,              // Left (Aponta para a ultima˝celula da lista)
                *Cursor;            // Cursor (Posicao corrente na lista)
        int Nb_Elements;
        int Current;



        public:

        // Operacoes:

        LinkList();
        int Is_Ok(int Position);
        int Go(int Position);
        int Back();
        int Forth();
        void InsertRight(V Element);
        void InsertLeft(V Element);
        void InsertFirst(V Element);
        int Insert(V Element,int Position);
```

```
        int DelLeft();
        int DelRight();
        int Del(int Position);
        void DelLast();
        int ChangeValue(V Element);
        void Start();
        void Finish();
        int NumberElements();
        V ReturnValue();
        int Search(V Element);
        void Print();
};

// Responsavel pela "incializacao" da lista duplamente encadeada.˝//

template <class V> LinkList<V>::LinkList()
{
        Left = Cursor = NULL;
        Nb_Elements = Current = 0;
        int DelRight();
}

// Verifica se a posicao de insercao de um novo novo objeto e˝valida. //
// Parametro: Position -> indica a posicao de insercao de um˝novo //
/ / e l e m e n t o n a l i s t a . ˝ / /
// Esta operacao retorna verdadeiro se a posicao e valida e falso˝se //
/ / n a o f o r . ˝ / /

template <class V> int LinkList<V>::Is_Ok(int Position)
{
        return ((1 <= Position) && (Nb_Elements+1 >= Position));
}

// Operacao que posiciona o Cursor em uma determinada localidade˝da lista. //
// Parametro: Position -> posicao para onde se deseja mover o˝cursor. //
// Primeiramente, verifica-se se a posicao e valida. Se for,˝move-se o //
// cursor para frente ou para tras, dependendo da posicao que se˝deseja e //
// retorna verdadeiro para indicar que a operacao foi efetuada˝com sucesso.//
// Caso contrario, retorna falso para indicar que esta posicao e˝invalida. //

template <class V> int LinkList<V>::Go(int Position)
{
        if ((Position >= 1) && (Position <= Nb_Elements))
        {
                if (Position > Current)
                {
                        do
                        {
                                Forth();
                        }while (Current!=Position);
                }
                else
                if (Position < Current)
                {
                        do
                        {
                                Back();
                        }while (Current!=Position);
                }
                return 1;
```

```
	}
	else
		return 0;
}

// Move o cursor para a celula anterior da lista. Esta operacao″//
// so e efetuada caso haja mais de um elemento na lista, pois″//
// nao faz sentido a execucao desta operacao. Entao ela retorna″//
// falso para controle do usuario. Agora, tendo mais de um ele-″//
// mento, ela e executada da seguinte forma: o cursor aponta″//
// para a celula anterior daquela que estava sendo apontada an-″//
// teriormente e a variavel Current e decrementada se o cursor″//
// nao estiver na primeira posicao da lista ou assume o valor″//
// de Nb_Elements que e a posicao do ultimo objeto da lista du-″//
// plamente encadeada. Fazendo isso, a funcao retorna verdadei-″//
// ro para indicar o sucesso da operacao.                    //

template <class V> int LinkList<V>::Back()
{
	if (Nb_Elements > 1)
	{
		if (Current == 1)
			Current = Nb_Elements;
		else
			Current--;
		Cursor = Cursor->Previous;
		return 1;
	}
	else
		return 0;
}

// Move o cursor para a celula posteror da lista. Esta operacao″//
// so e efetuada caso haja mais de um elemento na lista, pois″//
// nao faz sentido a execucao desta operacao. Entao ela retorna″//
// falso para controle do usuario. Agora, tendo mais de um ele-″//
// mento, ela e executada da seguinte forma: o cursor aponta″//
// para a celula posterior daquela que estava sendo apontada″//
// anteriormente e a variavel Current e incrementada se o cur-″//
// sor nao estiver na ultima posicao da lista ou assume o valor″//
// unitario que e a posicao do primeiro objeto da lista dupla-″//
// mente encadeada. Fazendo isso, a funcao retorna verdadeiro″//
// para indicar o sucesso da operacao.″//


template <class V> int LinkList<V>::Forth()
{
	if (Nb_Elements > 1)
	{
		if (Current == Nb_Elements)
			Current = 1;
		else
			Current++;
		Cursor = Cursor->Next;
		return 1;
	}
	else
		return 0;
}
// Insere uma nova celula a direita do cursor.            //
```

```
// Parametro: Element -> Tipo da celula que compoe a lista. //
// Primeiramente verifica-se se ha mais de um elemento na   //
// lista. Caso haja usa-se uma variavel Pointer que aponta  //
// para o novo objeto criado na lista. Entao inserimos este //
// objeto a direita do objeto para onde cursor aponta e re- //
// torna-se verdadeiro para indicar o sucesso da operacao.  //
// Caso contrario, retorna-se falso.                        //

template <class V> void LinkList<V>::InsertRight(V Element)
{
        if (Nb_Elements >= 1)
        {
                Cell<V> *Pointer = new Cell<V>(Element);
                Pointer->Previous = Cursor;
                Pointer->Next = Cursor->Next;
                Cursor->Next->Previous = Pointer;
                Cursor->Next = Pointer;
                Nb_Elements++;
                Forth();
        }
        else
                InsertFirst(Element);
}

// Insere uma nova celula a esquerda do cursor.             //
// Parametro: Element -> Tipo da celula que compoe a lista. //
// Primeiramente verifica-se se ha mais de um elemento na   //
// lista. Caso haja usa-se uma variavel Pointer que aponta  //
// para o novo objeto criado na lista. Entao inserimos este //
// objeto a esquerda do objeto para onde o cursor aponta e  //
// retorna-se verdadeiro para indicar o sucesso da opera-   //
// cao. Caso contrario, retorna-se falso.                   //

template <class V> void LinkList<V>::InsertLeft(V Element)
{
        if (Nb_Elements >= 1)
        {
                Cell<V> *Pointer = new Cell<V>(Element);
                Pointer->Previous = Cursor->Previous;
                Pointer->Next = Cursor;
                Cursor->Previous->Next = Pointer;
                Cursor->Previous = Pointer;
                Nb_Elements++;
                Current++;
                if (Cursor == Left)
                        Left = Cursor->Previous;
                Back();
        }
        else
                InsertFirst(Element);
}

// Insere a primeira celula que vai compor a lista.         //
// Parametro: Element -> Tipo da celula que compoe a lista. //
// O procedimento adotado e criar a primeira celula que ira //
// compor a lista e faz as devidas atribuicoes.             //

template <class V> void LinkList<V>::InsertFirst(V Element)
{
                Cursor = new Cell<V>(Element);
```

```
                    Cursor->Previous = Cursor;
                    Cursor->Next = Cursor;
                    Left = Cursor;
                    Nb_Elements = Current = 1;
}

// Insercao de uma celula em uma determinada posicao da lista. //
// Parametros: Element -> tipo que compoe a celula da lista;   //
//             Position -> posicao em que se deseja colocar a  //
//                         celula na lista.                    //
// O procedimento adotado foi, primeiramente, verificar se a   //
// posicao onde se deseja colocar o novo objeto e valida. Se   //
// for, verificamos se e desejado colocar a celula em uma po-  //
// sicao diferente da primeira da lista. Caso seja verdadeiro, //
// posicionamos o Cursor em uma celula antes da posicao dese-  //
// jada - usando a operacao Go - e faz-se a insercao deste     //
// objeto - usando a operacao InsertRight. Se a posicao do     //
// objeto for a primeira, posicionamos o Cursor na primeira"//
// posicao da lista e chamamos a operacao InsertLeft para efe-"//
// tuar sua insercao. Mas, existe uma particularidade que e a"//
// insercao do primeiro elemento que ira compor a lista. Entao"//
// chamamos a operacao InsertFirst para faze-la. Caso haja su-"//
// cesso nesta opercao de insercao, retorna-se verdadeiro,"//
// senao, retorna-se falso.                                    //

template <class V> int LinkList<V>::Insert(V Element,int"Position)
{
        if (Is_Ok(Position))
        {
                if (Position != 1)
                {
                        Go(Position-1);
                        InsertRight(Element);
                }
                else
                {
                        Go(Position);
                        InsertLeft(Element);
                }
                return 1;
        }
        else
                return 0;
}

// Apaga a celula para onde o cursor esta apontando e, depois,"//
// ele se move para a direita desta celula. Para que isso"//
// ocorra,primeiramente, verificamos se existe mais de um ele-"//
// mento na lista. Se isso for verdadeiro, verificamos se ha"//
// apenas um elemento nesta lista para, entao, usamos a opera-"//
// cao DelLast. Caso tenha mais de um elemento na lista, usa-"//
// mos um conjunto de instrucoes para apagar esta celula. Ago-"//
// ra, se esta operacao tiver sucesso, retorna-se verdadeiro,"//
// caso contrario, falso.                                      //

template <class V> int LinkList<V>::DelRight()
{
        Cell<V> *Pointer;

        if (Nb_Elements >= 1)
```

```
	{
		Pointer = Cursor;
		if (Nb_Elements == 1)
			DelLast();
		else
		{
			if (Cursor == Left)
				Left = Left->Next;
			Cursor = Cursor->Next;
			Pointer->Previous->Next = Pointer->Next;
			Pointer->Next->Previous = Pointer->Previous;
			if (Current == Nb_Elements)
				Current = 1;
			Nb_Elements--;
		}
		delete Pointer;
		return 1;
	}
	else
		return 0;
}

// Apaga a celula para onde o cursor esta apontando e, depois,"//
// ele se move para a esquerda desta celula. Para que isso"//
// ocorra,primeiramente, verificamos se existe mais de um ele-"//
// mento na lista. Se isso for verdadeiro, verificamos se ha"//
// apenas um elemento nesta lista para, entao, usamos a opera-"//
// cao DelLast. Caso tenha mais de um elemento na lista, usa-"//
// mos um conjunto de instrucoes para apagar esta celula. Ago-"//
// ra, se esta operacao tiver sucesso, retorna-se verdadeiro,"//
// caso contrario, falso.                                    //


template <class V> int LinkList<V>::DelLeft()
{
	Cell<V> *Pointer;

	if (Nb_Elements >= 1)
	{
		Pointer = Cursor;
		if (Nb_Elements == 1)
			DelLast();
		else
		{
			if (Cursor == Left)
				Left = Left->Next;
			Cursor = Cursor->Previous;
			Pointer->Previous->Next = Pointer->Next;
			Pointer->Next->Previous = Pointer->Previous;
			Nb_Elements--;
			if (Current == 1)
				Current = Nb_Elements;
			else
				Current--;
		}
		delete Pointer;
		return 1;
	}
	else
		return 0;
```

```
}

// Nesta operacao, faz-se atribuicoes quando se apaga o ultimo //
// objeto da lista.                                            //

template <class V> void LinkList<V>::DelLast()
{
        Cursor = Left = NULL;
        Current = Nb_Elements = 0;
}

// Apaga um elemento da lista em uma determinada posicao. //
// Parametro: Position -> posicao do objeto que compoe a  //
//                        lista dupalemente encadeada.    //
// Nesta operacao, primeiramente, verificamos se a existe //
// alguma celula naquela posicao. Caso haja, usamos a ope-//
// racao DelRight - poderia usar DelLeft tambem - para a  //
// retirada do objeto.

template <class V> int LinkList<V>::Del(int Position)
{
        if (Go(Position))
        {
                DelRight();
                return 1;
        }
        else
                return 0;
}

// Operacao responsavel pela troca do "valor" da celula   //
// para onde o cursor aponta.                             //
// Parametro: Element -> novo valor da celula.            //
// Esta troca do elemento da celula so acontece caso haja //
// pelo menos um elemento na lista.                       //

template <class V> int LinkList<V>::ChangeValue(V Element)
{
        if (Nb_Elements >= 1)
        {
                Cursor->Value = Element;
                return 1;
        }
        else
                return 0;
        }

// Operacao que posiciona o cursor na primeira posicao da lista.˝//

template <class V> void LinkList<V>::Start()
{
        if (Nb_Elements > 1)
        {
                Cursor = Left;
                Current = 1;
        }
}

// Operacao que posiciona o cursor na ultima posicao da lista.˝//
```

```
template <class V> void LinkList<V>::Finish()
{
	if (Nb_Elements > 1)
	{
		Cursor = Left->Previous;
		Current = Nb_Elements;
	}
}

// Operacao que retorna o numero de elementos da lista. //

template <class V> int LinkList<V>::NumberElements()
{
	return Nb_Elements;
}

// Operacao que retorna o "valor" da celula apontada pelo //
// Cursor.                                                //

template <class V> V LinkList<V>::ReturnValue()
{
	if (Nb_Elements > 0)
		return Cursor->Value;
}

// Operacao que retorna a posicao do elemento procurado na lista.˝//
// Parametro: Element -> Elemento procurado.˝//
// Este operacao consiste de procurar a primeira ocorrencia de um˝//
// determinado elemento. Quando ele e encontrado, retorna-se a˝//
// posicao onde ele se localiza. Mas, se este elemento nao esti-˝//
// na lista, retorna-se zero.˝//

template <class V> int LinkList<V>::Search(V Element)
{
	int i;

	i = 1;
	while ((i != Nb_Elements+1) && (Cursor->Value != Element))
	{
		Forth();
		i++;
	}
	if (Cursor->Value == Element)
		return Current;
	else
		return 0;

}


// Impressao da lista. //

template <class V> void LinkList<V>::Print()
{
	int i;

	Go(1);
	if (Cursor != NULL)
	{
		for (i = 1; i<=Nb_Elements; i++)
```

```
                {
                        cout << "  ";
                        cout << Cursor->Value;
                        Forth();
                }
                cout << "\n";
        }
}
```

Pilha

```
template <class V> class Stack{

        LinkList<V> S;

public:

        void Push(V);
        int Pop(V&);
        void Print();
};

template <class V> void Stack<V>::Push(V Element)
{
        S.Insert(Element,1);
}

template <class V> int Stack<V>::Pop(V& Element)
{
        if (S.NumberElements() > 0)
        {
                S.Go(1);
                Element = S.ReturnValue();
                S.DelRight();
                return 1;
        }
        else
                return 0;

}
template <class V> void Stack<V>::Print()
{
        S.Go(1);
        S.Print();
}
```

Fila

```
template <class V> class Queue{

        LinkList<V> Q;

public:

        void Insert(V);
        int Del(V&);
```

```
	void Print();
};

template <class V> void Queue<V>::Insert(V Element)
{
	Q.Insert(Element,Q.NumberElements()+1);
}

template <class V> int Queue<V>::Del(V& Element)
{
	if (Q.NumberElements() > 0)
	{
		Q.Go(1);
		Element = Q.ReturnValue();
		Q.DelRight();
		return 1;
	}
	else
		return 0;
}

template <class V> void Queue<V>::Print()
{
	Q.Go(1);
	Q.Print();
}
```

Árvore Binária

```
#include <alloc.h>
#include <iostream.h>

//--------------------------------------//
//- Celula que compoe a arvore -//
//--------------------------------------//

template <class V> class Cell{

	friend Tree<V>;

	Cell<V> *Left;		// Left -> aponta para a subarvore˝a esquerda.
	Cell<V> *Right;		 // Previous -> aponta para a˝subarvore a direita.
	V Value;		// Value -> elemento da celula do tipo˝V.


		Cell::Cell(V Element)
		{
			Value = Element;
			Left = NULL;
			Right = NULL;
		}

};

#include <alloc.h>
#include <iostream.h>

template <class V> class Tree{
```

```
        Cell<V> *Root;          // Aponta para o primeiro nodo da arvore.
        int Nb_Elements;     // Numero de elementos da arvore.

        // Operacoes

        public:

        Tree();
        int Insert(V Element);
        int Search(V Element);
        int Del(V Element);


// Inicializacao da arvore.
};

template <class V> Tree<V>::Tree()
{
                Root = NULL;
                Nb_Elements = 0;
}

// Operacao usada para inserir um elemento na arvore. //
// Parametro: Element -> elemento a ser inserido.     //

template <class V> int Tree<V>::Insert(V Element)
{

// Result -> e usado para determinar se ja existe ou nao o
//          elemento que desejamos inserir na arvore. 0
//          indica que o elemento ja esta presente na arvore
//          e 1, o elemento nao esta presente.
// Direction -> determina a direcao do ultimo caminhamento
//              na arvore. Se o seu valor for 0, caminhou-se
//              para a subarvore a esquerda e 1, caminhou-se
//              para a subarvore a direita.
// Pointer ->  apontador usado para fazer o caminhamento na
//             arvore.
// Temp -> apontador usado receber o endereco do  nodo pai do
//         elemento que sera inserido.

        int Result, Direction;
        Cell<V> *Pointer, *Temp;

// Pointer recebe o endereco de Root para fazer o caminhamento na˝arvore.

        Pointer = Root;
        Result = 1;

// Se o numero de elementos da arvore for 0, entao ha a insercao˝do
// primeiro elemento e mais nada e feito.

        if (Nb_Elements == 0)
        {
                Root = new Cell<V>(Element);
                Nb_Elements++;
        }

// Se o numero de elementos for maior que 0, entao ocorre a˝procura da
```

```
// posicao que ocorrera a insercao da nova celula na arvore. Para˝isso,
// segui-se o seguinte criterio: chaminha-se para a subarvore a˝direita
// se o elemento  inserido for maior do que aquele que esta no˝nodo para
// onde Pointer aponta ou caminha-se para a subarvore a esquerda˝se o
// elemento inserido for menor do que aquele que esta no nodo para˝onde
// Pointer aponta. Este processo e repetido ate que Pointer aponte˝para
// um nodo folha ou que ja tenha um elemento igual aquele que˝estamos
// inserindo.

        else
        {
                while (Pointer != NULL && Result != 0)
                {
                if (Pointer->Value >  Element)
                {

// Condicao que antecipa o alcance de um nodo folha.

                        if (Pointer->Left == NULL)
                        {
                                Temp = Pointer;
                                Direction = 0;
                        }
                        Pointer = Pointer->Left;

                }
                else
                        if (Pointer->Value < Element)
                        {

// Condicao que antecipa o alcance de um nodo folha.

                                if (Pointer->Right == NULL)
                                {
                                        Temp = Pointer;
                                        Direction = 1;
                                }
                                Pointer = Pointer->Right;
                        }
                        else
                                Result = 0;

                }

// Caso nao exista nehum elemento que estamos tentando inserir,˝entao e feita
// a insercao deste novo elemento. A variavel Direction˝determinara se o novo
// nodo inserido ficara a direita ou a esquerda do nodo pai.

                if (Result == 1)
                {
                        if (Direction == 0)
                                Temp->Left = new Cell<V>(Element);
                        else
                                Temp->Right = new Cell<V>(Element);
                        Nb_Elements++;
                }

        }
        return Result;
}
```

```
// Operacao usada para pesquisar se existe ou nao um elemento na˝arvore. //
// Parametro: Element -> elemento a ser pesquisado.˝       //

template <class V> int Tree<V>::Search(V Element)
{

// Result -> e usado para determinar se existe ou nao o
//          elemento que desejamos pesquisar na arvore. 1
//          indica que o elemento esta presente na arvore
//          e 0, o elemento nao esta presente.
// Pointer ->  apontador usado para fazer o caminhamento na
//             arvore.


          int Result;
          Cell<V> *Pointer;

// Pointer recebe o endereco de Root para fazer o caminhamento na˝arvore.

          Pointer = Root;
          Result = 0;

// Faz o caminhamento na arvore usando o mesmo criterio usado na˝insercao.
// Este caminhamento e interrompido quando encontramos um nodo˝folha ou
// quando encontramos o elemento que estamos pesquisando.

          while (Pointer != NULL && Result != 1)
          {
                    if (Pointer->Value >  Element)
                              Pointer = Pointer->Left;
                    else
                              if (Pointer->Value < Element)
                                        Pointer = Pointer->Right;
                              else
                                        Result = 1;
          }
          return Result;
}

// Operacao responsavel pela retirada de um nodo da arvore. //
// Parametro: elemento a ser retirado da arvore.           //

template <class V> int Tree<V>::Del(V Element)
{

// Result -> e usado para determinar se ja ha ou nao o
//          elemento que desejamos retirar da arvore. 1
//          indica que o elemento foi apagado da arvore
//          e 0, o elemento nao esta na arvore.
// Direction -> determina a direcao do ultimo caminhamento
//              na arvore. Se o seu valor for 0, caminhou-se
//              para a subarvore a esquerda e 1, caminhou-se
//              para a subarvore a direita. Se for igual a 2,
//              e porque o elemento a ser retirado e o nodo
//              raiz.
// Control ->  verifica a posicao do novo nodo que sera˝retirado.
// Pointer ->  apontador usado para fazer o caminhamento na
//             arvore.
// Temp -> apontador usado receber o endereco do ultimo nodo
```

```
//        apontado por Pointer, isto e, aponta para a ultima
//        subarvore apontada por Pointer.
// AuxPointer -> apontador para auxiliar na retirada de um nodo
//               da arvore.


        int Result, Direction, Control;
        Cell<V> *Pointer, *Temp, *AuxPointer;

// Inicializacao de Pointer  e Temp.

        Pointer = Temp = Root;
        Result = 0;
        Direction = 2;

// Primeiramente, fazemos o caminhamento na arvore, usando o mesmo˝criterio
// adotado na insercao, ate encontrarmos o nodo que desejamos˝encontra-lo ou
// ate encontrarmos um nodo folha.

        while (Pointer != NULL && Result != 1)
        {
                if (Pointer->Value >  Element)
                {
                        Temp = Pointer;
                        Pointer = Pointer->Left;
                        Direction = 0;
                }
                else
                        if (Pointer->Value < Element)
                        {
                                Temp = Pointer;
                                Pointer = Pointer->Right;
                                Direction = 1;
                        }
                        else
                                Result = 1;
        }

// Result = 1 indica que encontramos o nodo a ser retirado.

        if (Result == 1)
        {
                Nb_Elements--;
// Verificamos se existe uma subarvore a direita do nodo que sera˝retirado.
// Se nao existir retiramos este nodo. Senao, verificamos se˝existe uma
// subarvore a esquerda do nodo que sera retirado. Se nao exitir,˝retiramos
// este nodo. Agora, se existir uma subarvore tanto a direita˝quanto a
// esquerda do nodo que desejamos retirar, entao devemos˝substituir o
// elemento deste nodo pelo elemento de um nodo que esta mais a˝direita
// na subarvore a esquerda. Depois, retira-se o nodo que contem o˝elemento
// que foi copiado para o nodo que seria retirado.

                if (Pointer->Right == NULL)
                {
                        if (Direction==0)
                                Temp->Left = Pointer->Left;
                        else
                                if (Direction == 1)
                                        Temp->Right = Pointer->Left;
                                else
```

```
                                        Root = Pointer->Left;
                        }
                        else
                                if (Pointer->Left == NULL)
                                {
                                        if (Direction == 0)
                                                Temp->Left = Pointer->Right;
                                        else
                                                if (Direction == 1)
                                                        Temp->Right = Pointer->Right;
                                                else
                                                        Root = Pointer->Right;
                                }
                                else
                                {
                                        Control = 0;
                                        AuxPointer = Pointer;

// Busca do nodo mais a direita na subarvore a esquerda do nodo a˝ser retirado.

                                        Pointer = Pointer->Left;
                                        while (Pointer->Right != NULL)
                                        {
                                                Control = 1;
                                                AuxPointer = Pointer;
                                                Pointer = Pointer->Right;
                                        }
                                        if (Direction == 0)
                                                Temp->Left->Value = Pointer->Value;
                                        else
                                                if (Direction == 1)
                                                        Temp->Right->Value = Pointer->Value;
                                                else
                                                        Root->Value = Pointer->Value;
                                        if (Control == 0)
                                                AuxPointer->Left = Pointer->Left;
                                        else
                                                AuxPointer->Right = Pointer->Left;
                                }
                        delete Pointer;
                }
                return Result;
}
```

Apêndice B: Implementação em TOOL

Árvore Binária

```
-- Classe que representa o nodo da árvore.

XCLASS DoubleLink;

  PUBLIC REPRESENTATION
        DoubleLink      POLY    left;   -- Aponta para a esquerda do nodo.
        DoubleLink      POLY    right;  -- Aponta para a direita do nodo.
     INT object;                -- Tipo do nodo, no caso˝é inteiro.
  END REPRESENTATION

-- Inicialização da nodo

  METHOD Initialize
  BEGIN
        SELF.left <-NOW_IS(SELF);
        SELF.right <- NOW_IS(SELF);
 END METHOD
END XCLASS

-- Classe que representa a árvore.

XCLASS Tree;
        REPRESENTATION
                DoubleLink      POLY    root; -- Aponta para o nodo raiz.
                INT nb_elements;            -- Armazena o número de˝elementos.
        END REPRESENTATION

-- Inicialização da árvore.

        METHOD Initialize;
        BEGIN
                SELF.root <- NOW_IS(NULL);
                SELF.nb_elements:=0;
        END METHOD

-- Inserção dos elementos na árvore. Retorna falso se˝isso não ocorrer.

        METHOD Insert(IN  int info) RETURNS BOOLEAN result;
                DoubleLink      POLY    Pointer;
                DoubleLink      POLY    Temp;
                INT               Direction;
        BEGIN
                Pointer<-NOW_IS(root);
                Temp<-NOW_IS(NULL);
                result:=TRUE;
                IF SELF.nb_elements = 0
                THEN
                        root<-CREATE;
                        root<-Initialize;
                        root.object:= info;
                        SELF.nb_elements:=SELF.nb_elements+1;
                ELSE
                        LOOP
```

```
                        EXIT WHEN  Pointer<- IS_SAME(Temp)  OR NOT result;
                        IF (Pointer.object > info)
                        THEN
                                Temp<-NOW_IS(Pointer);
                                Pointer<-NOW_IS(Pointer.left);
                                Direction := 0;

                        ELSIF (Pointer.object < info)
                           THEN
                                        Temp<-NOW_IS(Pointer);
                                        Pointer<-Now_IS(Pointer.right);
                                        Direction := 1;

                           ELSE
                                result:=FALSE;
                        END IF

                END LOOP
                IF (result)
                THEN
                        if Direction = 0
                        then

                                Temp.left<-CREATE;
                                Temp.left<-Initialize;
                                Temp.left.object:=info;
                        ELSE

                                Temp.right<-CREATE;
                                Temp.right<-Initialize;
                                Temp.right.object:=info;

                        END IF
                        SELF.nb_elements:=SELF.nb_elements + 1;
                END IF
        END IF
END METHOD

-- Operação que pesquisa se um determinado elemento se˝encontra na árvore.
-- Retorna falso se ele não for encontrado e verdadeiro, caso˝contrário.

        METHOD Search(IN  INT info) RETURNS BOOLEAN result;
                DoubleLink      POLY    Pointer;
                DoubleLink      POLY    Temp;

        BEGIN
                Pointer<-NOW_IS(SELF.root);
                Temp<-NOW_IS(NULL);
                result:=FALSE;

                LOOP
                        EXIT WHEN  Pointer <- IS_SAME(Temp)  or result;
                        IF (Pointer.object > info)
                        THEN
                                Temp<-NOW_IS(Pointer);
                                Pointer<-NOW_IS(Pointer.left);
                        ELSE IF (Pointer.object < info)
                                THEN
                                        Temp<-NOW_IS(Pointer);
                                        Pointer<-NOW_IS(Pointer.right);
```

```
                ELSE
                        result:=TRUE;
                END IF
            END IF
        END LOOP

END METHOD

-- Operação que retira um nodo com um determinado valor da˝árvore.
-- Esta função retorna verdadeiro se houve a retirada do˝elemento e falso, caso contrário.

METHOD Del(IN INT info) RETURNS BOOLEAN result;
        int Result, Direction, Control;
        DoubleLink POLY Pointer, POLY Temp, POLY AuxPointer, POLY Temp2;
BEGIN
        Pointer<-NOW_IS(root);
        Temp<-NOW_IS(NULL);
        Result:=0;
        result:=false;
        Direction:=2;

        LOOP
                EXIT WHEN  Pointer <- IS_SAME(Temp)  or Result = 1;
                IF (Pointer.object > info)
                THEN
                        Temp<-NOW_IS(Pointer);
                        Pointer<-NOW_IS(Pointer.left);
                ELSIF (Pointer.object < info)
                THEN
                        Temp<-NOW_IS(Pointer);
                        Pointer<-NOW_IS(Pointer.right);
                ELSE
                        Result:=1;
                        result:=TRUE;
                END IF
        END LOOP

        IF (Result = 1)
        THEN
                SELF.nb_elements := SELF.nb_elements - 1;
                IF (Pointer.right<-IS_SAME(Pointer))
                THEN
                        IF (Direction = 0)
                        THEN
                                Temp.left<-NOW_IS(Pointer.left);
                        ELSE
                                root<-NOW_IS(Pointer.left);
                        END IF
                ELSIF (Pointer.left<-IS_SAME(Pointer))
                        THEN
                                IF (Direction = 0)
                                THEN
                                        Temp.left<-NOW_IS(Pointer.right);
                                ELSIF (Direction = 1)
                                        THEN
                                                Temp.right<-NOW_IS(Pointer.right);
                                        ELSE
                                                root<-NOW_IS(Pointer.right);
                                END IF
                ELSE
```

```
                        Control:=0;
                        AuxPointer<-NOW_IS(Pointer);
                        Pointer<-NOW_IS(Pointer.left);
                        LOOP
                                EXIT WHEN (Pointer.right<-IS_SAME(Pointer));
                                Control:=1;
                                AuxPointer<-NOW_IS(Pointer);
                                Pointer<-NOW_IS(Pointer.right);
                        END LOOP
                        IF (Direction = 0)
                        THEN
                                Temp.left.object:=Pointer.object;
                        ELSE
                                IF (Direction = 2)
                                THEN
                                        Temp.right.object:=Pointer.object;
                                ELSE
                                        root.object:=Pointer.object;
                                END IF
                        END IF
                        IF (Control = 0)
                        THEN
                                AuxPointer.left<-NOW_IS(Pointer.left);
                        ELSE
                                AuxPointer.right<-NOW_IS(Pointer.left);
                        END IF
                END IF
                Pointer<-DISPOSE;
                result:=TRUE;
        END IF

    END METHOD
END XCLASS
```

Apêndice C: Implementação em Ita

Árvore Binária

```
#include <italib.h>

class Tree;

// Celula que compoe a arvore.

class Cell<V> friend Tree{


        Cell<V> Left;
        Cell<V> Right;
        V Value;

public:

                // Operacao do nodo da arvore.

                init(V Element)
                {
                        Value = Element;
                        Left = NULL;
                        Right = NULL;
                }

                // Operacao de insercao a direita do nodo correspondente.

                void insert_right (Cell <V> other)
                { Right= other; }

                // Operacao de insercao a esquerda do nodo correspondete.

                void insert_left (Cell <V> other)
                { Left= other; }

                // Operacao de alteracao do elemento que compoe a celula˝correspondete.


                void Change(V Element)
                {Value = Element;}

}


class Tree<V>{

        Cell<V> Root;
        int Nb_Elements;

        public:

// Inicializacao da arvore.

init()
{
                Root = NULL;
```

```
                Nb_Elements = 0;
}


// Operacao usada para inserir um elemento na arvore.
// Parametro: Element -> elemento a ser inserido.

int Insert(V *Element)
{

// Result -> e usado para determinar se ja existe ou nao o
//           elemento que desejamos inserir na arvore. 0
//           indica que o elemento ja esta presente na arvore
//           e 1, o elemento nao esta presente.
// Direction -> determina a direcao do ultimo caminhamento
//              na arvore. Se o seu valor for 0, caminhou-se
//              para a subarvore a esquerda e 1, caminhou-se
//              para a subarvore a direita.
// Pointer ->  apontador usado para fazer o caminhamento na
//             arvore.
// Temp -> apontador usado receber o endereco do  nodo pai do
//         elemento que sera inserido.

            int Result, Direction;
            Cell<V> Pointer,Temp,Aux;

// Pointer recebe o endereco de Root para fazer o caminhamento na˝arvore.

            Pointer = Root;
            Result = 1;

// Se o numero de elementos da arvore for 0, entao ha a insercao˝do
// primeiro elemento e mais nada e feito.

            if (Nb_Elements == 0)
            {
                        Root = new Cell<V>(*Element);
                        Nb_Elements++;
            }

// Se o numero de elementos for maior que 0, entao ocorre a˝procura da
// posicao que ocorrera a insercao da nova celula na arvore. Para˝isso,
// segui-se o seguinte criterio: chaminha-se para a subarvore a˝direita
// se o elemento  inserido for maior do que aquele que esta no˝nodo para
// onde Pointer aponta ou caminha-se para a subarvore a esquerda˝se o
// elemento inserido for menor do que aquele que esta no nodo˝para onde
// Pointer aponta. Este processo e repetido ate que Pointer˝aponte para
// um nodo folha ou que ja tenha um elemento igual aquele que˝estamos
// inserindo.

            else
            {
                        while (Pointer != NULL && Result != 0)
                        {
                        if (Pointer.Value > *Element)
                        {

// Condicao que antecipa o alcance de um nodo folha.

                                    if (Pointer.Left == NULL)
```

```
                        {
                                Temp = Pointer;
                                Direction = 0;
                        }
                        Pointer = Pointer.Left;

                }
                else
                        if (Pointer.Value < *Element)
                        {

// Condicao que antecipa o alcance de um nodo folha.

                                if (Pointer.Right == NULL)
                                {
                                        Temp = Pointer;
                                        Direction = 1;
                                }
                                Pointer = Pointer.Right;
                        }
                        else
                                Result = 0;


                }


// Caso nao exista nehum elemento que estamos tentando inserir,˝entao e feita
// a insercao deste novo elemento. A variavel Direction˝determinara se o novo
// nodo inserido ficara a direita ou a esquerda do nodo pai.

                if (Result == 1)
                {
                        if (Direction == 0)
                        {
                                Aux = new Cell<V>(*Element);
                                Temp.insert_left(Aux);
                        }
                        else
                        {
                                Aux = new Cell<V>(*Element);
                                Temp.insert_right(Aux);
                        }
                        Nb_Elements++;
                }

        }
        return Result;
}

// Operacao usada para pesquisar se existe ou nao um elemento na˝arvore.
// Parametro: Element -> elemento a ser pesquisado.

int Search(V *Element)
{

        int Result;
        Cell<V>  Pointer;

// Result -> e usado para determinar se existe ou nao o
```

```
//          elemento que desejamos pesquisar na arvore. 1
//          indica que o elemento esta presente na arvore
//          e 0, o elemento nao esta presente.
// Pointer ->  apontador usado para fazer o caminhamento na
//             arvore.


            Pointer = Root;
            Result = 0;

// Faz o caminhamento na arvore usando o mesmo criterio usado na˝insercao.
// Este caminhamento e interrompido quando encontramos um nodo˝folha ou
// quando encontramos o elemento que estamos pesquisando.

            while (Pointer != NULL && Result != 1)
            {
                        if (Pointer.Value > *Element)
                                    Pointer = Pointer.Left;
                        else
                                    if (Pointer.Value < *Element)
                                                Pointer = Pointer.Right;
                                    else
                                                Result = 1;
            }
            return Result;
}

// Operacao responsavel pela retirada de um nodo da arvore.
// Parametro: elemento a ser retirado da arvore.


int Del(V *Element)
{

// Result -> e usado para determinar se ja ha ou nao o
//          elemento que desejamos retirar da arvore. 1
//          indica que o elemento foi apagado da arvore
//          e 0, o elemento nao esta na arvore.
// Direction -> determina a direcao do ultimo caminhamento
//              na arvore. Se o seu valor for 0, caminhou-se
//              para a subarvore a esquerda e 1, caminhou-se
//              para a subarvore a direita. Se for igual a 2,
//              e porque o elemento a ser retirado e o nodo
//              raiz.
// Control ->  verifica a posicao do novo nodo que sera˝retirado.
// Pointer ->  apontador usado para fazer o caminhamento na
//             arvore.
// Temp -> apontador usado receber o endereco do ultimo nodo
//         apontado por Pointer, isto e, aponta para a ultima
//         subarvore apontada por Pointer.
// AuxPointer -> apontador para auxiliar na retirada de um nodo
//               da arvore.

            int Result, Direction, Control;
            Cell<V> Pointer, Temp, AuxPointer;

            Pointer = Temp = Root;
            Result = 0;
            Direction = 2;
```

```
// Primeiramente, fazemos o caminhamento na arvore, usando o mesmo˝criterio
// adotado na insercao, ate encontrarmos o nodo que desejamos˝encontra-lo ou
// ate encontrarmos um nodo folha.

        while (Pointer != NULL && Result != 1)
        {
                if (Pointer.Value > *Element)
                {
                        Temp = Pointer;
                        Pointer = Pointer.Left;
                        Direction = 0;
                }
                else
                        if (Pointer.Value < *Element)
                        {
                                Temp = Pointer;
                                Pointer = Pointer.Right;
                                Direction = 1;
                        }
                        else
                                Result = 1;
        }

// Result = 1 indica que encontramos o nodo a ser retirado.

        if (Result == 1)
        {
                Nb_Elements--;

// Verificamos se existe uma subarvore a direita do nodo que sera˝retirado.
// Se nao existir retiramos este nodo. Senao, verificamos se˝existe uma
// subarvore a esquerda do nodo que sera retirado. Se nao exitir,˝retiramos
// este nodo. Agora, se existir uma subarvore tanto a direita˝quanto a
// esquerda do nodo que desejamos retirar, entao devemos˝substituir o
// elemento deste nodo pelo elemento de um nodo que esta mais a˝direita
// na subarvore a esquerda. Depois, retira-se o nodo que contem o˝elemento
// que foi copiado para o nodo que seria retirado.

                if (Pointer.Right == NULL)
                {
                        if (Direction==0)
                                Temp.insert_left(Pointer.Left);
                        else
                                if (Direction == 1)
                                        Temp.insert_right(Pointer.Left);
                                else
                                        Root = Pointer.Left;
                }
                else
                        if (Pointer.Left == NULL)
                        {
                                if (Direction == 0)
                                        Temp.insert_left(Pointer.Right);
                                else
                                        if (Direction == 1)
                                                Temp.insert_right(Pointer.Right);
                                        else
                                                Root = Pointer.Right;
                        }
                        else
```

```
                        {
                                Control = 0;
                                AuxPointer = Pointer;

// Busca do nodo mais a direita na subarvore a esquerda do nodo a˝ser retirado.

                                Pointer = Pointer.Left;
                                while (Pointer.Right != NULL)
                                {
                                        Control = 1;
                                        AuxPointer = Pointer;
                                        Pointer = Pointer.Right;
                                }
                                if (Direction == 0)
                                        Temp.Left.Change(Pointer.Value);
                                else
                                    if (Direction == 1)
                                            Temp.Right.Change(Pointer.Value);
                                   else
                                            Root.Change(Pointer.Value);
                                if (Control == 0)
                                    AuxPointer.insert_left(Pointer.Left);
                                else
                                    AuxPointer.insert_right(Pointer.Left);
                        }
                delete Pointer;
        }
        return Result;
}

}
```

Apêndice D: Exemplos em C++

Para todos os programas, primeiramente deve-se declarar a lista, pilha, fila ou árvore, definindo o seu tipo e depois usar as operações definidas para cada TAD. Abaixo, terá exemplos utilizando os TADs definidos e a utilização de algumas de suas operações.

Lista Duplamente Encadeada

```
#include <stdlib.h>

main()
{
        LinkList<char> L;
        int j;
        char i;
        cout << "\nTeste de Lista de Caracteres\n";
        cout << "\n----------------------------\n";

        i = 'a';
        while (i!='g')
        {
           L.InsertRight(i);
           i++;
        }
        L.Print();
        cout << "\nIr para o ultimo elemento da lista\n\n";
        L.Finish();
        cout << "\nApagar o ultimo elemento da lista\n";
        L.DelRight();
        L.Print();
        cout << "\nIr para o 3o. elemento da lista\n";
        L.Go(3);
        cout << "\nTroca o 3o. elemento por j\n";
        L.ChangeValue('j');
        L.Print();
        cout << "\nApagar o 10o. elemento da lista\n";
        j = L.Del(10);
        if (!j)
                cout << "\nNao existe o 10o. elemento na lista\n";
        cout << "\nApagar o 1o. elemento da lista\n";
        L.Del(1);
        L.Print();
        cout << "\nO numero de elementos da lista e ";
        cout << L.NumberElements();

}
```

Pilha

```
#include <stdlib.h>

main()
{
        Stack<int> S;
        int i,j;
        cout << "\nTeste de Pilha de Inteiros\n";
        cout << "\n----------------------------\n";

        i = 1;
        while (i!=9)
        {
           S.Push(i);
           i++;
        }
        S.Print();

        cout << "\nRetirada de 2 elementos da pilha\n";

        S.Pop(i);
        S.Pop(j);
        cout << "\n Sao eles: ";
        cout << i;
        cout << " e ";
        cout << j;
        cout << "\n";
        cout << "\nAgora a pilha so tem os elementos:\n";
        S.Print();

}
```

Fila

```
#include <stdlib.h>

main()
{
        Queue<int> Q;
        int i,j;
        cout << "\nTeste de Fila de Inteiros\n";
        cout << "\n----------------------------\n";

        i = 1;
        while (i!=9)
        {
           Q.Insert(i);
           i++;
        }
        Q.Print();

        cout << "\nRetirada do 1o. elemento da fila\n";

        Q.Del(i);
        Q.Print();
        cout <<"\n\nInsercao de dois novos elementos na fila\n";
```

```
        Q.Insert(50);
        Q.Insert(13);
        Q.Print();

}
```

Árvore Binária

```
#include <stdlib.h>

main()
{       int i[30], j, c, d, k =0;
        Tree<int> T;
        randomize();

        // Gerando os elementos que irao compor a arvore.

        for (j=0; j<=29; j++)
        {
                i[j] = rand() % 30;

                // Contando o numero de 10's e 20's.

                if (i[j] == 20 || i[j] == 10)
                        d++;

                // Inserindo os elementos na arvore.

                T.Insert(i[j]);

                //Conta o numero de elementos repetidos

        }

        // Imprimindo os elementos gerados.

        for (j=0; j<=29; j++)
        {
                cout << i[j];
                cout << " ";
        }
        cout << "\n";

        // Apagando os elementos 10 e 20 da arvore.

        T.Del(10);
        T.Del(20);

        cout << "\n";
        k=0;

        // Contagem dos elementos que ainda estao na arvore.

        for (j=0; j<=29; j++)
        {
                c=T.Search(i[j]);
                if (c)
                        k++;
```

```
        }

        cout << "\nO numero de elementos que forma pesquisados com´sucesso sao: ";
        cout << k;


        cout << "\nO numero de 10's e 20's sao: ";
        cout << d;
}
```

Apêndice E: Resultados em C^{++}

Lista Duplamente Encadeada

Teste de Lista de Caracteres

---
 a b c d e f

Ir para o ultimo elemento da lista

Apagar o ultimo elemento da lista
 a b c d e

Ir para o 3o. elemento da lista

Troca o 3o. elemento por j
 a b j d e

Apagar o 10o. elemento da lista

Nao existe o 10o. elemento na lista

Apagar o 1o. elemento da lista
 b j d e

O numero de elementos da lista e 4

Pilha

Teste de Pilha de Inteiros

---
 8 7 6 5 4 3 2 1
Retirada de 2 elementos da pilha

 Sao eles: 8 e 7

Agora a pilha so tem os elementos:
 6 5 4 3 2 1

Fila

Teste de Fila de Inteiros

---
 1 2 3 4 5 6 7 8
Retirada do 1o. elemento da fila
 2 3 4 5 6 7 8

Insercao de dois novos elementos na fila
 2 3 4 5 6 7 8 50 13

Árvore Binária

7 20 12 17 6 3 4 18 2 3 29 22 19 25 29 14 10 3 9 7 22 15 17 27 22˝25 29 14 17 12

O numero de elementos que forma pesquisados com sucesso sao: 28
O numero de 10's e 20's sao: 2

Apêndice F: Exemplo em TOOL

Árvore Binária

```
XCLASS VisualTeste FROM Program;

  REPRESENTATION
        DialogBox POLY dialog;
        Static POLY obj1;
        Tree  POLY Arvore;
        BOOLEAN result;
  END REPRESENTATION

  MESSAGE
    Destroyed;
  END MESSAGE;

  METHOD Main;
  BEGIN

        SELF <- Dialog( NULL );
        Arvore<-Create;
        Arvore<-Initialize;

        obj1 <- SetText("Insercao dos elementos 5, 6, 4, 3.");
        obj1 <- Show;
        Arvore<-Insert(5);
        Arvore<-Insert(6);
        Arvore<-Insert(4);
        Arvore<-Insert(3);

        Arvore<-Del(4);
        result:=Arvore<-Search(4);

        if (NOT result)
        tHEN
                obj1 <- SetText("O elemento 4 foi retirado com sucesso.");
                obj1 <- Show;
        END IF
  END METHOD -- Main

  METHOD Dialog( IN VAR Window parent );
    VisualStyle v;
    Color color;
    Cursor new_cursor;
    Cursor old_cursor;
    StaticStyle static_style;
    Color st_fore_color,st_back_color;
  BEGIN
    new_cursor<- CREATE;
    new_cursor<- CreatePredefinedCursor( WAIT );
    old_cursor<- SAME( new_cursor <- SetCursor );
    new_cursor<- ShowCursor( TRUE );
    dialog <- CREATE;
    v <- Add( BORDER );
    v <- Add( DLG_FRAME );
    v <- Add( SYS_MENU );
    dialog <- DialogInitialize( 255, 193, 114, 300, 300,˝"Teste", v, parent );
    color <- Set(15269887);
```

```
    dialog <- SetBkColor( color );
    obj1 <- CREATE;
    v := CHILD;
    static_style := CENTER;
    obj1 <- StaticInitialize( v, static_style, "", 106, 49, 89,˝45, dialog );
    dialog <- AttachToDialogBox( obj1, FALSE, '\0' );
    st_back_color <- Set(12632256)
    obj1 <- SetBkColor(st_back_color);
    st_fore_color <- Set(0)
    obj1 <- SetForeColor(st_fore_color);
    SELF <- VisualInitialize;
    dialog <- Dialog;
    old_cursor<- SetCursor;
  END METHOD -- Dialog

  METHOD VisualInitialize;
  BEGIN
  END METHOD -- VisualInitialize

  METHOD LastWishes;
  BEGIN
    IF NOT OWNER <- IS_NULL
    THEN OWNER <<- Destroyed ;
    END IF
    dialog <- LastWishes;
  END METHOD -- LastWishes
END XCLASS
```

Apêndice G: Resultado em TOOL

Árvore Binária

Insercao dos elementos 5, 6, 4, 3.

O elemento 4 foi retirado com sucesso.

Apêndice H: Exemplo em Ita

Árvore Binária

```
main()
{
      int i, j, Element;
      Tree<int> T = new Tree<int>;


      // Insercao dos elemtos na lista.

      Element = 5;
      T.Insert(&Element);
      Element = 7;
      T.Insert(&Element);
      Element = 6;
      T.Insert(&Element);
      Element = 3;
      T.Insert(&Element);
      Element = 4;
      T.Insert(&Element);
      Element = 2;
      T.Insert(&Element);
      Element = 8;
      T.Insert(&Element);
      printf("\nOs elementos presentes na arvore sao: ");
      for (j = 2;j < 9;j++)
      {
            printf("%d ",j);
      }
     printf("\n\nRetirada do elemento 5 da arvore\n");

      Element = 5;
      T.Del(&Element);
      printf("\nAgora, os elementos presentes sao: ");
      for (j = 2;j < 9;j++)
      {
          i = T.Search(&j);
            if (i)
            {
                printf("%d ",j);
            }
      }
}
```

Apêndice I: Resultado em Ita

Árvore Binária

Os elementos presentes na arvore sao: 2 3 4 5 6 7 8

Retirada do elemento 5 da arvore

Agora, os elementos presentes sao: 2 3 4 6 7 8