Tipos Abstratos de Dados em Machĭna

Letícia Decker de Sousa
Roberto da Silva Bigonha

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Belo Horizonte

março de 2006

Conteúdo

1 Introdução **3**

1.1 Objetivo do Projeto 3

1.2 Breve Comentário 3

2 Visão Geral de Machĭna **5**

2.1 Estrutura do Programa 6

2.1.1 Módulo 6

2.1.2 Álgebra 7

2.1.3 Abstrações 7

2.1.4 Regra de Transição 8

2.1.5 Invariante 8

2.2 Entradas 8

2.3 Agentes 8

2.4 Detalhes de Machĭna 9

3 Tipos Abstratos de Dados **11**

3.1 Tipos Abstratos de Dados em Machĭna 11

3.1.1 Recursos para a Implementação de Tipos Abstratos de Dados 11

4 Listas **13**

4.1 Definição de Tipo Abstrato de Dados Lista Simplesmente Encadeada e sua Representação: 13

4.2 Interface do TAD Lista: 15

4.3 Implementação: 17

4.3.1 Operações: 17

4.3.2 Operação insertLeft: 27

4.3.3 finish: 28

4.4 Conclusão: 31

5 Fila **33**

5.1 Definição do Tipo Abstrato de Dados Fila(T) e sua Representação: 33

5.2 Definição do Tipo Abstrato de Dados Fila Circular e sua Representação: . . . 34

5.3 Interface das Operações do Tipo Abstrato de Dados Fila(T) 35

5.4 Implementação 36

5.4.1 Operações: 36

5.5 Conclusão: 38

6 Pilha **39**

6.1 Definição do Tipo Abstrato de Dados Pilha e sua Representação: 39

6.2 Implementação 40

6.2.1 Interface das Operações em Pilha 41

6.2.2 Operações: 41

6.3 Conclusão: 43

7 Árvore **45**

7.1 Definição de TAD Árvore Binária de Pesquisa e sua Representação: 46

7.1.1 Interface das Operações do Tipo Abstrato de Dados Árvore Binária de Pesquisa: 49

7.2 Implementação: 49

7.2.1 Operações: 50

7.3 Conclusão: 59

8 Árvore Digital **61**

9 Árvore Patricia **63**

9.1 Definição de Árvore Patricia 63

9.2 Tipo Abstrato de Dados Árvore Patricia 64

9.3 Implementação e Representações: 64

9.3.1 Interfaces das Operações em Patricia 67

9.4 Operações: 67

9.4.1 inicialize: 67

9.4.2 pesquisa: 68

9.4.3 insira: 69

9.4.4 retira: 73

9.5 Conclusão: 75

10 Árvore SBB **77**

10.1 Tipo Abstrato de Dados Árvore SBB: 77

10.2 Implementação e sua Representação: 78

10.3 Interfaces das Operações de SBB: 79

10.3.1 Operações: 79

10.4 Conclusão: 91

11 Tabela Hash **93**

11.1 Tipo Abstrato de Tabela Hash e sua Representação: 93

11.2 Interface das Operações em Tabela Hash: 95

11.3 Implementação do tipo abstrato de dado Tabela Hash: 95

11.3.1 Operações: 96

11.4 Implementação das Operações de Listas Especiais para Tabela Hash: 98

11.4.1 Operações: 99

11.5 Conclusão: 100

12 Conclusão: **101**

Capítulo 1

Introdução

1.1 Objetivo do Projeto

O foco deste trabalho é definir, em Machĭna, tipos abstratos de dados com o intuito de avaliar a linguagem em suas facilidades de implementação e abstração.

1.2 Breve Comentário

A facilidade de escrever programas em uma determinada linguagem está intimamente relacionada com os recursos nela disponíveis. Esses recursos são os instrumentos que aproximam os problemas do mundo real e sua implementação. É desejável que a linguagem tenha uma coleção mínima de tipos de dados que possibilitem esse mapeamento entre problemas e implementação de maneira sucinta.

Em Machĭna, temos tipos de dados primitivos (básicos), compostos, genéricos e de arquivo. Os tipos de dados básicos são aqueles que não são definidos em termos de outros tipos [**?**]. Int, Bool, Char, Real e String, respectivamente o inteiro, booleano, caracter, real e cadeias de caracteres, são os tipos de dados primitivos em Machĭna. Esse conjunto de tipos quase não varia de linguagem para linguagem.

O tipo composto é aquele que é formado a partir de um tipo ou mais um associados de maneira especial. Em Machĭna temos listas, agentes, tuplas, uniões disjuntas, conjuntos e tipos que podem definidos pelo próprio programador.

Os tipos genéricos são tipos compostos em que o tipo formador não é definido, como em listas, agentes genéricos e união disjunta que pode ser de qualquer tipo primitivo. Há também os tipos de arquivo que podem ser Stream, Input ou Output.

Esses tipos são utilizados para a manipulação de dados e também para a definição de outros tipos, abstrações de dados e tipos abstratos de dados.

Os outros tipos que podem ser implementados são renomeações de tipos preexistentes ou criação novos tipos, a partir de uniões disjuntas de quaisquer outros tipos de dados. As tuplas, tipo já definido em Machĩna, comporta-se como um recurso que fornece a possibilidade de formação de uma estrutura nova, tal como `struct` em linguagem `C`.

Define-se tipo abstrato de dado como uma estrutura de um programa que satisfaz às seguintes condições:

- É caracterizados por um conjunto bem definidos de operações.
- É uma encapsulação que define a representação dos valores do tipo sobre os quais as operações descritas dentro da referida encapsulação atuam.
- Existe um controle de visibilidade: a estrutura interna do tipo não é visível ao usuário, que só poderá acessá-la por meio das operações que forem descritas internamente ao módulo. Mas o nome do tipo é público.
- Operações e definição de tipo são descritas dentro de uma única unidade sintática, sendo que outras unidades de programa tem permissão de criar variáveis do tipo citado [**?**].

Capítulo 2

Visão Geral de Machĭna

Um programa em Machĭna é um conjunto de regras de transição, podendo conter declarações de funções e definições de tipos e abstrações. As regras de transição, por sua vez, são um conjunto de comandos que são processados paralelamente em cada passada. Uma passada é a execução de toda a regra de transição, voltando o ponto de execução para o início desta mesma regra. Assume-se o tempo de uma passada é o intervalo de tempo entre o início da execução da regra de transição até o seu fim. Comandos são todas expressões semânticas que expressam alguma ação, mundando assim, o estado da máquina. Para o melhor entendimento do conceito de estado de uma máquina, assim como funções, sugere-se a consulta à `Especificação da Linguagem Formal Machĭna 2.0`. Mas em linhas gerais, pode-se interpretar função como entidade que é mapeada para um valor dentro do intervalo no qual ela foi declarada. Se na Matemática temos variáveis, em Machĭna temos funções.

Em Machĭna, pode-se definir regras de transições como procedimentos que são chamados em outras regras de transição. Esses procedimentos são chamados de abstrações e podem ser definidos no módulo que os chama ou podem ser importados de outros módulos que os definam.

Para suprir a necessidade de execução de comandos seqüenciais, usa-se a palavra-chave `step`, seguido de um número inteiro. Começa-se com `step 1`, em seguida `step 2` e assim sucessivamente, sendo essa seqüência de `step` crescente. Em frente de cada `step` seguido do número e de dois pontos, coloca-se a regra de transição que será executada quando for a passada de execução desse `step`. Em cada passada, apenas um `step` será executado, reposicionando o ponto de execução da próxima passada para o `step` enumerado imediatamente superior ao `step` que foi executado.

Pode-se alterar a ordem de execução das passadas, quando se usa palavra-chave `step`. Nesse caso, pode-se redefinir qual será o próximo `step` a ser executado, utilizando-se a palavra-chave `next` e atribuindo-se a ela, como se essa fosse uma função do tipo inteiro, o número referente ao proximo `step` a ser executado.

A regra de transição de um programa é executada continuamente, em inúmeras passadas, até que um comando de encerramento da execução seja encontrado, ou seja, a palavra-chave `stop`. No caso da regra de transição pertencer a uma ação (abstração), a palavra-chave de parada é `return`.

2.1 Estrutura do Programa

2.1.1 Módulo

Cada agente é associado a um módulo e é responsável pela sua execução. Esse agente pode ser criado através da operação `create`, que pode sido solicitada por um módulo principal ou por qualquer outro agente. Os agentes comunicam-se através de chamadas de abstração que são anunciadas na interface do módulo principal dos agentes. O agente é criado em um módulo que possui a denominação de `machina` ou em qualquer outro módulo quando se utiliza o comando `create`. A diferença da criação de agentes em módulos denominados `machina` é o fato que, nesse caso, ocorre a criação inicial do agente do módulo e nos outros casos criam-se novos agentes através de um que foi criado anteriormente por meio de módulo `machina`.

No módulo `machina`, cria-se o agente e associa-o a um módulo, por meio da seqüência-chave `agent of` seguido pelo nome do módulo, o qual o agente será associado. Pode-se criar quantos agentes se quiser em um único modulo do tipo `machina`.

O módulo é dividido nas seguintes sessões:

- `import`: coloca no escopo do módulo os agentes que o agente deste precisa se comunicar.
- `include`: define quais os módulos secundários e seus elementos que precisam ser incorporados ao módulo em questão.
- `algebra`: define as funções e tipos referentes a esse módulo.
- `abstractions`: define regras de transições que podem ter uma passada ou um número ilimitado, teoricamente, de passadas. Essas podem ser usadas localmente ou serem exportadas.
- `initial state`: local onde se inicializa funções dinâmicas.
- `transition`: é constituído pelas regras de transição, alterando assim o estado da máquina através da ação do agente.
- `invariant`: define a condição que não pode ser alterada durante as mudanças de estado da máquina.

2.1.2 Álgebra

É nessa sessão do módulo em que se declaram as funções e onde se definem os tipos que serão utilizados no módulo. Pode-se declarar as funções como sendo estáticas, dinâmicas, derivadas ou externas. Caso a função seja estática, seu valor não pode ser modificado. O único momento em que se atribui um valor para uma função classificada como estática é no momento de inicialização. Já a função dita dinâmica pode ter seu valor alterado a qualquer momento pelas operações do módulo em que ela está vinculada ou que são incluídas nele. As funções declaradas como derivadas são funções que não podem sofrer alterações de valor, mas podem acessar funções dinâmicas, derivadas ou externas. As funções externas são definidas e atualizadas em um ambiente externo, podendo ser implementadas em outra linguagem de programação. As palavras-chaves para diferenciar esses vários tipos de classificação de funções são respectivamente `static`, `dynamic`, `derived` e `external`.

Pode-se ainda classificar cada tipo de função como sendo interna ao módulo ou acessível a módulos externos. Nesse último caso, antes da palavra-chave que especifica a qualificação da função, introduz-se a palavra-chave `public`. Na ausência desta última palavra-chave, a função é considerada interna ao módulo.

2.1.3 Abstrações

São dois os tipos de ações, caracterizados pelo número de passadas que podem executar. Todas as ações são identificáveis pela palavra-chave `action` antes do seu nome, seguida pela especificação de parâmetros de entradas e saídas que são identificadas respectivamente pelas palavras-chaves `in` e `out` seguidas do nome do parâmetro, dois pontos e o seu tipo. Na ausência destas palavras-chaves de especificação de parâmetros, presume-se que a função possua ambas características. Uma ação pode ter uma única passada em sua regra de transição, sendo que aparece, nesse caso, a palavra-chave `begin` depois das declarações internas da ação; ou pode ter inúmeras passadas até encontrar o comando de parada `return` ou ter chegado ao final da série de `step`. Nesse último caso, usa-se, após a declaração interna de funções, a palavra-chave `loop`.

Pode-se fazer declarações locais em uma ação, mas ao fazê-las torna-se obrigatório o uso de comandos de marcação `begin` ou `loop`. Estes agem como separadores de declarações locais e da regra de transição das ações. É optativo o uso da palavra-chave `begin` quando não forem declaradas localmente as funções.

Assim como as funções, as ações podem ser públicas ou privadas, definindo-se mecanismos de visibilidade. Para que as ações possam ser públicas, ou seja, possam ser visíveis fora do módulo de definição, deve aparecer antes da palavra-chave `action` a palavra-chave `public`. Caso não apareça nada, a ação é dita privada e não pode ser acessada fora do módulo em que fora criada.

2.1.4 Regra de Transição

Em uma regra de transição, todos os comandos são executados concomitantemente na mesma passada, respeitando-se, casos em que apareça `step`, o que torna os comandos em `step` diferentes, executados em série.

A regra de transição é constituída por um conjunto de comandos que mudam o estado da máquina. As regras de transição são efetuadas continuamente até que se encontre o comando de parada `stop`. Assim encerra-se a execução desse módulo e a atuação do seu respectivo agente nele.

2.1.5 Invariante

O invariante é uma condição que nunca deve ser violada. A verificação do valor do invariante se passa no intervalo entre duas passadas e caso seu valor mude, uma mensagem de erro é gerada e a execução é interrompida.

2.2 Entradas

Para utilizar alguma ação de um determinado módulo em outro módulo, deve-se incluir o módulo no qual ela foi definida no módulo em que a ação é necessária. Todas as ações do módulo incluído que são públicas ficam, então, disponíveis no módulo que o inclui. Para tal, utiliza-se a palavra-chave `include` seguida pelo nome do módulo.

O uso da palavra-chave `import` seguindo de um nome, coloca o escopo do agente referente a esse nome, comunicável com o módulo que o importou.

2.3 Agentes

Cada módulo contendo uma regra de transição é executado por um agente em que a seqüência-chave de criação do tipo agentes é `create agent of`. O agente pode ser criado em um módulo especial denominado `machina`, em que se criam tantos agentes quantos os que forem requisitados. Pode-se criar outros agentes em módulos comuns por meio da palavra-chave `create`.

Podem ser executadas sobre o agente as seguintes operações: criação, eliminação, atribuição, passagem por parâmetro, disparo de execução, retorno de função, comparação por igualdade e passagem de parâmetro.

2.4 Detalhes de Machĭna

Como Machina é baseada em máquinas de estados abstratos (ASM), não contém o tipo apontador e com isso, limita-se a implementação que por ventura tenha sido baseada em apontadores. Também não permite a implementação de algoritmos com recursividade.

`Machĭna` é uma linguagem que se difere das outras pela ausência de loops. No entanto a regra de transição é executada exaustivamente até que não se modifique mais o estado da máquina. Desta maneira pode-se simular um loop geral, do tipo "enquanto o estado da máquina se modificar faça"que engloba toda regra de transição. Loops mais internos não são possíveis diretamente, pois como já foi dito a linguagem não suporta loops. Mas esse problema pode ser facilmente transposto com a utilização de ações que são regras de transições encapsuladas como procedimento e são executadas também até que não haja mudança no estado da máquina.

Capítulo 3

Tipos Abstratos de Dados

Foram implementados em Machina os tipos abstratos de dados correspondentes as seguintes estruturas: `Lista Simplesmente Encadeada`, `Fila de Arranjo Circular`, `Pilha`, `Árvore Binária de Pesquisa`, `Árvore Patricia`, `Árvore SBB` e `Tabela Hash`. Todos esses tipos de dados têm sua implementação baseada em implementações na linguagem Java [**?**].

3.1 Tipos Abstratos de Dados em Machĭna

Para a implementação de um tipos abstratos deve concentrar em um único módulo o tipo que sedeseja manipular e torná-lo público através da palavra-chave `public` imediatamente anterior à palavra-chave `type` e posteriormente denomina-se o tipo seguido pela palavra-chave `is`. Torna-se apenas público o nome do tipo, deixando com que os seus constituíntes sejam privados ao módulo criador, e portanto, podendo ser manipulados somente através de operações públicas do mesmo módulo.

Tendo declarado o tipo e as operações que atuam sobre ele e sendo esses públicos a outros módulos, este módulo contem um `tipo abstrato de dados` (`TAD`) de acordo com as definições, ou seja, um tipo que tem um conjunto único de operações que atua sobre ele, todos implementados em um único módulo.

3.1.1 Recursos para a Implementação de Tipos Abstratos de Dados

Tem-se o controle de visibilidade através da palavra-chave `public` que torna o que assim foi declarado visível fora do módulo de implementação. Caso ocorra a ausência dessa palavra-chave, o tipo ou função ou ação é somente manipulável dentro do módulo de criação.

Pode-se criar as operações que atuem sobre o tipo declarado como público utilizando-se ações.

Essas ações devem ser públicas para serem acessadas por módulos fora do módulo de criação. O objetivo destas operações é o de manipular o tipo, modificando os valores dos constituintes do tipo, já que esses não são visiveis fora do módulo de criação.

Capítulo 4

Listas

Tipo abstrato de dados `Lista` é um recipiente contendo os elementos formadores de uma lista, sendo que podem ser de qualquer tipo. A retirada ou inserção não seguem uma regra específica: podem ser feitas sobre qualquer elemento em qualquer ordem.

Lista linear é uma estrutura de dados dinâmica na qual seus elementos estão organizados de maneira seqüêncial, definindo-se para cada elemento uma posição de ordem dentro da seqüência.

4.1 Definição de Tipo Abstrato de Dados Lista Simplesmente Encadeada e sua Representação:

Nesse tipo abstrato de dados, os elementos formadores da lista contêm uma referência ao próximo elemento da lista.

O tipo abstrato de dados `Lista(T)` foi implementado como uma lista ordenada constituídas de elementos que têm como referência para o próximo elemento da lista um inteiro que é mapeado para um elemento da lista. Tem-se uma função especial que indica em que elemento da lista está atuando a operação que está sendo executada. Esse elemento é chamado de elemento corrente da lista e é nomeado de `active` nessa implementação. Para atuar em qualquer elemento da lista, tem-se que torná-lo o elemento corrente (`active`).

Como a lista é simplesmente encadeada, ou seja, tem-se somente a referência ao elemento imediatamente posterior ao elemento corrente, para atuar em um elemento anterior a ele, deve-se posicionar como elemento corrente um elemento anterior ao de interesse. Para tal, guarda-se a informação de qual é o primeiro elemento da lista (`firstElement`), pois esse é anterior a qualquer elemento dela.

Um elemento de uma lista guarda a informação de qual é o elemento imediatamente posterior

a ele na ordem da `Lista Simplesmente Encadeada` e possui entidades de armazenamento de outras informações quaisquer, de interesse particular para cada caso.

Caminha-se na lista através das referências aos elementos posteriores e com uma operação especial que posiciona o primeiro elemento da lista como o elemento corrente. Assim, o `tipo abstrato de dados Lista` possui a informação de qual é o elemento corrente e qual é o primeiro elemento da lista.

O tipo `Lista (T)` é uma tupla genérica. Possui os seguintes campos: `nElements`, `position`, `firstElement`, `active`, `previous`, `prox`, `status`, `vetor` e `max`. O `nElements` designa o número de elementos contidos na lista; `position` indica qual é a posição ordinal do elemento corrente na lista; `firstElement` guarda um inteiro que é mapeado para o primeiro elemento da lista; `active` é o campo que guarda um inteiro que é mapeado para o elemento corrente da lista; `previous` e `prox` guardam inteiros que mapeiam repectivamente para o elemento imediatamente anterior e posterior ao elemento corrente; `status` guarda uma das seis possibilidade de estados gerado pela última operação realizada sobre a lista. O campo `vetor` é uma função que dado um inteiro, devolve os nodos armazenados na `Lista` e a função `max` designa o tamanho máximo que a `Lista(T)` pode ter.

```
public type Lista(T) is( nElements: Int, position: Int,
                         firstElement: Int, active: Int,
                         previous: Int, prox: Int,
                         status: Erro, vetor: Element(T), max: Int );
```

Essa estrutura é ilustrada abaixo:

Esse módulo define tipos auxiliares a `Lista(T)`, como `Element(T)`, `Erro` e `Nodo(T)`.

O tipo `Element(T)` é um mapeamento do tipo inteiro para o tipo `Nodo(T)`.

```
type Element(T) is Int -> Nodo(T);
```

O tipo `Erro` enumera todas as possibilidades de estados que podem ser gerados durante as operações do tipo abstrato de dados `Lista (T)`.

```
public type Erro is enum ( NoError, ErrorOutOfLimits, ErrorEmptyList,
                           ErrorOffRight,ErrorOffLeft, ErrorNoPosition );
```

A figura abaixo ilustra a implementação do TAD Lista(T), que é uma lista simplesmente encadeada.

O tipo `Nodo(T)` representa o tipo de cada elemento da lista, contendo os campos `info`, `right` e `uso`. O campo `info` guarda a informação armazenada no elemento referente a esse nodo e possui o tipo genérico `T`. O campo `right` guarda um inteiro que é mapeado para o elemento imediatamente à direita do elemento em questão. O campo `uso` indica se o nodo está sendo usado ou não, ou seja, informa se um dado inteiro, que será mapeado para um nodo, está disponível para uma nova inserção na lista.

```
type Nodo(T) is (info:T, right: Int, uso: Bool);
```

4.2 Interface do TAD Lista:

As operações que caracterizam TAD `Lista` nessa implementação são `status, checkRight, checkLeft`, `checkEmpty, numOfElements, value`, `changeValue, checkFirst, checkLast`,`start, forth, go, back, search`, `insertRight, insertLeft, finish, delete` e `inicializaLista`.

As funções `status, numOfElements` e `value` são função que acessam as informações contidas na estrutura interna de `Lista`.

Seguem abaixo a descrição das operações e o relacionamento de dependência que, por ventura, alguma operação possua com outra desta mesma coleção.

- `status`(in z: Lista(T), out erro: Bool): informa se houve algum erro durante a execução da última operação sobre a lista `z`.
- `checkRight`(in z:Lista(T), out offRight: Bool): informa se o elemento corrente está além do último elemento da lista `z`.

- `checkLeft`(in z:Lista(T), out offLeft: Bool): informa se o elemento corrente está aquém do primeiro elemento da lista `z`.
- `checkEmpty`(in z: Lista(T), out empty: Bool): verifica se a lista `z` se encontra vazia.
- `numOfElements`(in z: Lista(T), out nElement: Int): devolve a quantidade de elementos que a lista `z` possui.
- `getValue`(in z: Lista(T), out v: T): devolve o valor da informação de interesse do elemento corrente da lista `z` em `v`. Testa antes se a posição corrente se encontra dentro dos limites da lista, caso contrário gera um estado de erro.
- `changeValue`(z:Lista(T), in v: T): recebe uma lista `z` e um valor válido `v` para elementos da lista, substituindo o valor do elemento corrente pelo o valor dado.
- `checkFirst`(in z: Lista(T), out first: Bool): dada uma lista `z`, informa se o elemento corrente é o primeiro elemento da lista.
- `checkLast`(in z: Lista(T), out last: Bool): informa se o elemento corrente é o último elemento da lista `z`.
- `start`(z:Lista(T)): marca o primeiro elemento da lista `z` como sendo o elemento corrente.
- `forth`(z: Lista(T)): caminha uma posição para frente na lista `z`, caso seja possível, senão gera um estado de erro.
- `go`(z: Lista(T), in pos: Int): dada a posição desejada `pos`, marca como elemento corrente da lista `z`, o elemento correspondente a essa posição, caso seja possível, senão gera um estado de erro.
- `back`(z: Lista(T)): marca como corrente o elemento anterior ao atual corrente da lista `z`, se possível, senão gera um estado de erro.
- `search`(z: Lista(T), in v: T, out achou): dada uma informação válida `v`, caminha na lista `z` até encontrar o elemento contendo o valor `v`, devolvendo na função `achou` o resultado da pesquisa. Marca como elemento corrente, o elemento que foi encontrado ou o último elemento da lista, caso a informação passada como parâmetro não esteja na lista.
- `insertRight`(z: Lista(T), in v: T): recebe uma informação válida `v` e insere-a à direita do elemento corrente da lista `z`. Faz o elemento inserido o novo elemento corrente.
- `insertLeft`(z: Lista(T), in v: T): dada uma informação válida `v` e insere-a à esquerda do elemento corrente da lista `z`. Faz o elemento inserido o novo elemento corrente.
- `finish`(z: Lista(T)): marca o elemento corrente como sendo o último elemento da lista `z`.
- `delete`(z: Lista(T)): retira o elemento corrente da lista `z`.
- `inicializaLista(z: Lista(T), in max: Int)`: cria uma lista `z`, iniciando seus valores.

4.3 Implementação:

A lista linear pode ser implementada como uma estrutura em que os elementos encontram-se indexados tal como em um arranjo. Como a inserção e retirada de elementos em uma `Lista Linear` podem ser feitas em quaisquer posições, a cada uma dessas operações em uma implementação do tipo arranjo, deve-se reorganizar os elementos de modo a não deixar elementos nulos entre elementos não-nulos. Apenas são permitidos elementos nulos após a última posição da lista. Certamente, a implementação de listas através de arranjos não é a possibilidade mais eficiente se a lista não é estável, já que a reorganização citada poderia ser necessária frequentemente e o custo se tornaria alto. Define-se uma lista estável, a lista em que a taxa de pesquisa é significativamente maior do que a taxa de retiradas e inserções juntas. No caso em que não temos uma `Lista` estável, uma alternativa seria a implementação de `Listas Encadeadas`.

Para o caso de uma lista estável seria uma boa solução a lista linear. Neste caso, a constituição da lista não se altera muito ao longo do tempo, sendo sua manutenção, por esse motivo, não muito alta. O peso da pesquisa, no entanto, representa a maior fatia do custo da `Lista`, minimizando-o se fosse a `Lista` implementada como linear, ou seja, através de arranjos. A pesquisa em uma `Lista Linear` é direta através da indexação de arranjo, possuindo custo O(1).

No caso de listas instáveis, o custo das retiradas e inserções representam uma fatia maior do custo de manutenção da `Lista` em relação ao custo de pesquisa. Nesses casos, a melhor implementação é utilizando-se de `Lista Encadeada`. O encadeamento de elementos formadores de listas, geralmente está relacionada à existência de ponteiros. Uma estrutura que representa uma informação e um apontador para a próxima estrutura de mesmo significado é um elemento da lista. No entanto, não existe em Machĭna o tipo apontador. Esse problema pode ser contornado utilizando-se mapeamento. Associa-se cada elemento da lista a um inteiro e o que se guarda em um elemento da tt Lista é o inteiro que é mapeiado para o próximo elemento dela.

4.3.1 Operações:

Status:

Essa operação dá acesso à informação contida no campo status do tipo `Lista(T)`. Esse campo guarda a informação do estado gerado pela última operação realizada sobre a lista. Todas as operações da TAD `Lista(T)` atualizam o valor de status quando chamadas. O acesso ao valor de `status` só é possível através desta operação, já que o programador não tem acesso à estrutura interna da lista. O valor de `status` denota as seguintes ocorrências:

- NoError: ausência erros;
- ErrorOutOfLimits: erro gerado por referência a elementos fora dos limites da lista;

- ErrorEmptyList: erro gerado ao tentar acessar algum elemento em uma lista vazia;
- ErrorOffRight: erro gerado quando se acessa um elemento que ultrapassa limite superior;
- ErrorOffLeft: erro gerado quando se acessa um elemento que ultrapassa limite inferior;
- ErrorNoPosition: erro gerado em acessos à posições inexistentes;

```
public action status( in z: Lista(T), out status: Erro) is
     status:= z.status;
end Status
```

checkRight e checkLeft:

Operações que testam se o elemento corrente da lista se encontra fora dos limites `Lista`.

Depois do último elemento:

```
public action  checkRight( in z: Lista(T), out offRight: Bool) is
     offRight:= (z.nElements = 0 | z.position = z.nElements + 1);
end  checkRight
```

Antes do primeiro elemento:

```
public action checkLeft( in z: Lista(T), out offLeft: Bool) is
     offLeft:= (z.position = 0);
end checkLeft
```

Operação checkEmpty:

Essa operação informa se a lista está vazia.

```
public action checkEmpty(in z: Lista(T), out vazio: Bool) is
     vazio:= (z.nElements = 0);
end checkEmpty
```

numOfElements:

Somente com essa operação pode-se ter acesso à quantidade de elementos que estão contidos na lista.

```
public action numOfElements(in z: Lista(T), out num: Int) is
    num:= z.nElements;
end  numOfElements
```

getValue e changeValue:

A operação `getValue` devolve a informação contida no elemento corrente da lista e a função `changeValue` torna possível a mudança dessa informação.

`getValue:`

```
public action getValue(in z: Lista(T), out valueElem: T) is
    if not  DentroLimitesLista then
        PegaValor;
    else  GeraErro;
    end
end getValue
```

DentroLimitesLista:= (z.position = 0 — z.position = nElements + 1)

PegaValor:

```
valueElem:= vetor(z.active).info;
z.status:= "NoError";
```

GeraErro:

```
z.status:= "ErrorOutOfLimits";
```

`changeValue:`

```
public action changeValue(z:Lista(T), out v: T) is
    if not  DentroLimitesLista then
        MudaInfo;
    else  GeraErro;
    end
end changeValue
```

MudaInfo:

```
vetor(z.active).info:= v;
z.status:= NoError;
```

checkFirst e checkLast:

A operação `checkFirst` verifica se o elemento corrente é o primeiro elemento da lista e a operação `checkLast` verifica se o elemento corrente é o último elemento da lista.

`checkFirst:`

```
public action checkFirst(in z:Lista(T), out first: Bool) is
     first:= (z.position = 1);
end checkFirst
```

`checkLast:`

```
public action checkLast(in z: Lista(T), out last: Bool) is
     last:= (z.position = z.nElements);
end isLast
```

start:

Operação que fornece valores iniciais para os campos da lista, marcando o elemento corrente como sendo o primeiro da elemento da lista e as informações dos outros campos coerentes com essa informação.

```
public action start(z: Lista(T)) is
    vazia: Bool;
loop:
    step 1:   TestaListaVazia(z);
    step 2: if not vazia  then
                    FazPrimeiroElementoCorrente(z)
              else  GeraErroListaVazia(z);
              end
end start
```

TestaListaVazia(z: Lista(T):

```
checkEmpty(z, vazia);
```

FazPrimeiroElementoCorrente(z: Lista(T)):

```
z.previous:= 0;
```

```
z.position:= 1;
z.active:= z.firstElement;
z.prox:= Element(z.active).right;
z.status:= NoError;
```

GeraErroListaVazia(z: Lista(T)):

```
z.status:= ErrorEmptyList;
```

forth, go e back:

A operação `forth` que marca como elemento corrente o próximo elemento da lista e trata casos especiais em que o elemento corrente ou o próximo elemento corrente não existiriam, isto é, o corrente é anterior ao primeiro ou é o último elemento.

```
public action forth (z: Lista(T)) is
     vazia: Bool;
loop
     step 1:  TestaListaVazia(z);
     step 2: if vazia then
                    GeraErroListaVazia(z);
                   return;
               end
               if (z.position = nElements + 1) then
                    GeraErroLimiteSuperior(z);
               end
               if ( z.position = 0 ) then
                    InicializaLista(z);
               else  FazCorrenteProximoElemento(z);
               end
end forth
```

TestaListaVazia(z: Lista(T)):

```
checkEmpty(z, vazia);
```

GeraErroListaVazia(z: Lista(T)):

```
z.status:= ErrorEmptyList;
```

GeraErroLimiteSuperior(z: Lista(T)):

```
z.status:= ErrorOffRight;
```

InicializaLista(z: Lista(T)):

```
start(z);
z.status:= NoError;
```

FazCorrenteProximoElemento(z: Lista(T)):

```
z.previous:= z.active;
z.active:= z.prox;
 CondicaoModificacaoProx(z: Lista(T)):
z.position:= z.position + 1;
z.status:= NoError;
```

CondicaoModificacaoProx(z: Lista(T)):

```
if (z.prox!= 0) then
     z.prox:= vetor(z.prox).right;
end
```

A operação `go` faz corrente o elemento da lista na posição passada por `i`, tratando casos em que esse não se encontra entre os limites da lista. O parâmetro `i` é um inteiro que representa a posição ordinal de um elemento da `Lista(T)`.

Caso esse seja maior que a posição do elemento corrente, o `forth` é chamado repetidamente até chegar na posição desejada. Caso contrário, inicializa a lista, chamando posteriormente o `forth` repetidas vezes até que se alcance a posição `i` requerida.

```
public action go (z: Lista(T), in i: Int) is
    offRight: Bool ;
loop:
    step 1: if  VerificaLimites then
                    GeraErroForaLimites(z);
                   return;
             end
    step 2: if (i < z.position) then
                   InicializaLista(z);
            end
    step 3:  TestaLimiteInferior(z);
    step 4: if not  ChegouNaPosicao  and not UltrapassouLimiteSuperior then
                    FazCorrenteProximoElemento(z);
                   next:= 3;
            end
end go
```

VerificaLimites:=

```
( i < 1 |  i > nElements )
```

GeraErroForaLimites(z: Lista(T)):

```
z.status:= ErrorOutOfLimits;
```

TestaLimiteInferior(z: Lista(T):

```
checkRight(z, offRight);
```

ChegouNaPosicao(z: Lista(T)):= (z.position = i)

UltrapassouLimiteSuperior(z: Lista(T)):= offRight

InicializaLista(z: Lista(T)):

```
start(z);
```

FazCorrenteProximoElemento(z: Lista(T)):

```
forth(z);
z.status:= NoError;
```

E a última operação que trata do caminhamento na lista é a `back`. Esta faz corrente o elemento anterior ao elemento ativo. Lembrando-se de que temos como campos de lista o elemento anterior e posterior ao corrente.

```
public action back (z: Lista(T)) is
     vazia: Bool ;
loop:
     step 1:  VerificaListaVazia(z);
     step 2: if (vazia) then
                    GeraErroListaVazia(z);
                   return;
               end
               if (z.position = 0) then
                     GeraErroForaLimiteInferior(z);
                    return;
               end
               if (z.position = 1) then
                     FazProximoElementoPrimeiroElemento(z);
               else  VaParaElementoAnterior(z);
               end
end back
```

VerificaListaVazia(z: Lista(T)):

```
checkEmpty(z, vazio);
```

GeraErroListaVazia(z: Lista(T)):

```
z.status:= ErrorEmptyList;
```

GeraErroForaLimiteInferior(z: Lista(T)):

```
z.status:= ErrorOffLeft;
```

FazPróximoElementoPrimeiroElemento(z: Lista(T)):

```
z.active:= 0;
z.previous:= 0;
z.prox:= z.firstElement;
z.position:= 0;
```

VaParaElementoAnterior(z: Lista(T)):

```
go(z,z.position-1);
```

search:

Essa operação pesquisa por um elemento da lista que possua uma dada informação `v` passada como parâmetro. A priori, marca o corrente como sendo o primeiro elemento. Depois começa a percorrer a lista até alcançar o elemento cuja informação é a passada no referido parâmetro `v`. A operação `search` percorre a lista utilizando a operação `forth`. Garante-se que a procura pelo elemento se mantenha dentro dos limites da lista.

```
public action search (z:Lista(T), in v: T, out achou: Bool) is
     offRight, vazia: Bool;
loop:
     step 1:  VerificaListaVazia(z);
     step 2: if (vazia) then
                    GeraErroListaVazia(z);
                   return;
              end
              start(z);
              achou:= false;
     step 3:  VerificaLimiteSuperior(z);
     step 4: if  AchouInfomacaoNaLista(z) and not
```

```
                                        UltrapassouLimiteSuperior then
                        achou:= true;
                        return;
                    elseif (off_right) then
                         GeraErroForaLimiteSuperior(z);
                        achou:= false;
                        return;
                    end
            step 5:  FazCorrenteProximoElemento(z);
                     next:= 3;

    end search
```

VerificaLimiteSuperior(z:Lista(T)):

```
    checkRight(z, offRight);
```

UltrapassouLimiteSuperior:= offRight

AchouInfomacaoNaLista(z:Lista(T)):= z.active.info = v

GeraErroForaLimiteSuperiorz:Lista(T)):

```
    z.status:= ErrorOffRight;
```

FazCorrenteProximoElemento(z: Lista(T)):

```
    forth(z);
```

insertRight:

Operação que insere um elemento imediatamente antes do elemento corrente e faz do elemento inserido o próximo corrente. Para a criação da estrutura do elemento a ser inserido, chama-se `alocaCel`, que devolve um inteiro que não esteja associado a qualquer estrutura presente na `Lista`. Atualizam-se os campos do novo nodo com a informação passada como parâmetro. O elemento corrente, agora, é o valor do inteiro gerado pelo `alocaCel`. Atualizam-se os valores `prox` e `previous` da `Lista`.

```
    public action insertRight(z: Lista(T), in v: T) is
        vazia: Bool;
    loop:
        step 1:  VerificaListaVazia(z, vazia);
        step 2:  AlocaNodo(z, num);
```

```
        step 3:  InsereInfoNodo(z, num, v);
        step 4: if (vazia) then
                      TrataInsercaoListaVazia(z);
                else if (  z.position = 0  |  z.position = nElements + 1 ) then
                           GeraErroForaLimitesLista(z);
                          return;
                      end
                       InsercaoDireita(z):
                end
         IncrementaNumElementosLista(z);
    end insertRight
```

VerificaListaVazia(z: Lista(T), vazia: Bool):

```
checkEmpty(z, vazia);
```

AlocaNodo(z: Lista(T), num: Int)

```
alocaCel(z, num);
```

InsereInfoNodo(z: Lista(T), num: Int, v: T):

```
z.vetor(num).info:= v;
```

GeraErroForaLimitesLista(z: Lista(T)):

```
z.status:= "ErrorOutOfLimits";
```

TrataInsercaoListaVazia(z: Lista(T)):

```
z.firstElement:= num;
z.active:= num;
z.previous:= 0;
z.prox:= 0;
z.position:= 1;
z.firstElement.right:= 0;
```

InsercaoDireita(z: Lista(T)):

```
vetor(num).right:= z.prox;
vetor(z.active).right:= num;
z.previous:= z.active;
z.position:= z.position + 1;
z.active:= num;
```

IncrementaNumElementosLista(z: Lista(T)):

```
z.nElements:= z.nElements +1;
```

alocaCel:

Operação local que fornece um inteiro entre os limites do tamanho da lista que ainda não representa uma estrutura dela para ser usado no mapeamento de inteiro para uma estrutura da `Lista`. Utilizada em operações de inserção de elemento em lista.

```
action alocaCel(z: Lista(T), num: Int) is
     choose x > 0 and x < max satisfying z.vetor(x).uso = false do
          num:= x;
          z.vetor(x).uso:= true;
     end
end alocaCel
```

4.3.2 Operação insertLeft:

Ação que insere um elemento imediatamente após o elemento corrente e marca o elemento inserido como corrente. A geração do elemento a ser inserido é igual a da função insertRight. O campo `right` do elemento a ser inserido é o `right` do elemento corrente e o valor deste último é atualizado para o valor gerado por `alocaCel`. Tanto essa função como a anterior garantem que as operações se processem dentro dos limites da lista.

```
public action insertLeft(z: Lista(T), in v: T) is
     vazia: Bool;
loop:
     step 1:  VerificaListaVazia(z, vazia);
     step 2:  AlocaNodo(z, num);
     step 3:  InsereInfoNodo(z, num, v);
     step 4: if (vazia) then
                    TrataInsercaoListaVazia(z);
               else if ( z.position = 0 |  z.position = nElements + 1 ) then
                         GeraErroForaLimitesLista(z);
                        return;
                    end
                    if (z.position = 1) then
                         TrataInsercaoInicioLista(z);
                    else                                   InserçãoEsquerda(z);
                    end
     step 5: z.active:= num;
               z.nElements:= z.nElements + 1;
end insertLeft
```

TrataInsercaoInicioLista(z: Lista(T)):

```
z.firtElement:= num;
vetor(num).right:= z.active;
z.prox:= z.active;
```

InserçãoEsquerda(z: Lista(T)):

```
vetor(num).right:= z.active;
vetor(z.previous).right:= num;
z.prox:= z.active;
```

Atualiza-se o elemento corrente como sendo o elemento que acabou de ser inserido e incrementa-se o número total de elementos na `Lista` nas duas operações.

4.3.3 finish:

Faz do último elemento da lista, o elemento corrente dela.

Verifica, previamente se o elemento corrente se encontra dentro dos limites da lista. Se fora, posiciona como sendo o corrente o primeiro elemento da lista. Então desloca-se na `Lista` até chegar no último elemento da lista.

```
public action finish(z: Lista(T)) is
     offRight: Bool;
     offLeft: Bool;
     vazia: Bool;
loop:
     step 1:  VerificaSituacaoCorrente(z);
     step 2: if (vazia) then
                       GeraErroListaVazia(z);
                      return;
              end
               ForaLimitesInicializaLista(offRight, offLeft, z);
     step 3: if not  HaProximoNodo then
                   ProximoElementoCorrente(z);
                  next:= 3;
             end
end finish
```

VerificaSituacaoCorrente(z: Lista(T)):

```
checkRight(z, off_right);
checkLeft(z, off_left);
checkEmpty(z, vazio);
```

GeraErroListaVazia(z: Lista(T)):

```
z.status:= ErrorEmptyList;
```

ForaLimitesInicializaLista(offRight: Bool, offLeft: Bool, z: Lista(T)):

```
if ( offRight | offLeft) then
     start(z);
end
```

HaProximoNodo:= vetor(z.active).right = 0 ProximoElementoCorrente(z: Lista(T)):

```
forth(z);
```

delete:

Essa operação remove o elemento corrente da lista, marca o próximo como sendo o novo elemento corrente, atualizando o valor de número de elementos que a lista contém.

```
public action delete(z: Lista(T)) is
loop:
     step 1:  SituacaoLista(z);
     step 2:  LiberaCorrente(z: Lista(T));
     step 3:  CondicaoMaisDeUmElementoLista(z: Lista(T));
     step 4:  CondicaoRetiradaPrimeiroElemento(z: Lista(T));
     step 5: if (z.position = 1) then
                    TrataRetiradaListaVazia(z);
              else  ModificaElementoAnteriorCorrente(z);
              end
     step 6:  DecrementaNumeroElementos(z);
end  delete
```

SituacaoLista(z: Lista(T)):

```
if ( z.position = 0 | z.position = z.nElements + 1 )  then
     z.status:= ErrorOutOfLimits;
     return;
end
```

CondicaoMaisDeUmElementoLista(z: Lista(T)):

```
if not ( z.active = 0 ) then
     z.prox:= vetor(z.active).right;
end
```

LiberaCorrente(z: Lista(T)):

```
     liberaCel(z, z.active);
     z.active:= z.prox;
```

CondicaoRetiradaPrimeiroElemento(z: Lista(T)):

```
if (z.position = 1) then
     z.firstElement:= z.active;
end
```

TrataRetiradaListaVazia(z: Lista(T)):

```
     if (z.firstElement = 0) then
          z.position:= 0;
     end
```

ModificaElementoAnteriorCorrente(z: Lista(T)):

```
     z.vetor(z.previous).right:= z.active;
```

DecrementaNumeroElementos(z: Lista(T)):

```
     z.nElements:= z.nElements - 1;
```

liberaCel:

```
     action liberaCel(z: Lista(T), num: Int) is
          z.vetor(num).uso:= false;
     end liberaCel
```

Operação local que torna o inteiro que representava uma estrutura na lista em um inteiro não associado a nenhuma estrutura e portanto possível de ser mapeado para alguma outra no futuro. Esta é utilizada em operações de retirada de estruturas da lista.

inicializaLista:

Operação que contrói a `lista`, sendo com que todos os campos da tupla que descrevem a lista, zerados. Faz com que todo inteiro que pertence ao intervalo de 1 ao valor máximo `max` que a lista pode ter, não esteja associado a nenhum elemento de `Lista`.

```
public action inicializaLista(z: Lista(T), in max: Int) is
     z.max:= Int;
     z.previous:= 0;
     z.prox:= 0;
     z.active:= 0;
     z.position := 0
     z.nElements:= 0;
     forall x: Index do
          z.vetor(x).uso:= false;
     end
end inicializaLista
```

4.4 Conclusão:

O tipo abstrato de dados `Lista` possui algumas dificuldades de implementação que são superadas na implementação dos tipos abstratos de dados `Fila` e `Pilha` que serão apresentados a seguir. Na `Lista` tem-se O um dos problemas da implementação de uma lista é saber se, em tempo de compilação, a `Lista` é estável ou instável. Dependendo dessa característica da `Lista`, a implementação segue dois rumos diferentes: lista de tamanho fixo, seguindo o estilo de um "vetor", ou caso contrário, seguindo o estilo de uma lista encadeada. Ambas implementações tem suas vantagens e desvantagens. Se a lista é instável, fica muito caro tratá-la como um "vetor"porque é alta a taxa de inserção e retirada. Nesse caso, tem-se que reorganizar os elementos da `Lista` de modo que não se tenha nenhum elemento nulo entre elementos não-nulo. Para uma lista generalizada, em que não se tem características que definem a `Lista`, é mais prudente a implementação de uma `Lista Encadeada`, pois não é necessário definir, em tenpo de complilação, o tamanho da lista e inserção e retiradas são facilitadas. No entanto, esse tipo de implementação, prejudica a pesquisa na `Lista`, já que tem-se que percorre-la até o elemento de interesse. Se na implementação da `Lista` , utiliza-se a idéia de vetor tem-se uma complexidade do algoritmo da ordem de 1, na implementação da `Lista Encadeada` a pesquisa possui uma complexidade da ordem de n. No entanto, tem a vantagem da flexibilização do tamanho, que generaliza o tipo abstrato de dados `Lista` para casos em que se torna necessário o uso de `lista` de tamanho diferente. Há, também, o fato desse tipo abstrato de dados conter muitas operações básicas, o que torna a implementação extensa e cansativa, além de tornar a didática da explicação das operações difícil.

Capítulo 5

Fila

`Fila` é um tipo abstrato de dados derivado de `Lista`, na qual se impõe uma restrição em relação ao elemento que deve ser retirado de cada vez, isto é, a fila é uma lista na qual o elemento a ser retirado é sempre o que está na fila há mais tempo.

5.1 Definição do Tipo Abstrato de Dados Fila(T) e sua Representação:

A maneira mais simples de se implementar o tipo fila é através de uma lista linear com operações de inserção e retiradas modificadas. Nesse caso, mantém-se a ordem de entrada na fila utilizando o próprio indexamento do arranjo da lista linear. O elemento de índice menor foi inserido anteriormente que um de índice maior. Como a ordem é mantida de forma sistemática, as retiradas só poderão ser efetuadas no início da fila, ou seja, somente quando não existem na fila elementos cujos índices sejam menores que o que se retira. As inserções, portanto, nessa implementação, devem ser sempre feitas depois do último elemento não-nulo.

Não é necessário o encadeamento de elementos para esse tipo abstrato de dados, já que não temos o infortúnio de retiradas de elementos entre elementos não nulos e esse tipo de implementação é menos eficiente, nesse caso, que o uso de arranjo. A não ser que o número máximo de elementos seja de difícil previsão.

Por ser um tipo de lista, possui uma sequência de elementos como a descrita para o caso do TAD `Lista(T)`. Em `Lista`, não há restrições quanto a retirada de elementos. No entanto, para o caso de fila, é permitida somente a retirada do elemento que foi inserido na `Lista` a mais tempo, assim como o conceito popular de fila: o primeiro a entrar na fila é o primeiro a sair(FIFO - first in, first out). Como se pode notar, haverá uma grande simplificação do tipo se mantivermos, na implementação, a ordem de chegada; tornando, também, a manutenção da `Fila` mais fácil e simples que a da `Lista`.

5.2 Definição do Tipo Abstrato de Dados Fila Circular e sua Representação:

Ao implementar fila usando arranjo, permitindo a inserção somente no fim da fila, mantém-se a ordem de inserção, facilitando a retirada de elementos. No entanto, tem-se um problema quando se implementa fila com arranjo simples: arranjos possuem tamanho limitado e como as retiradas são somente do primeiro elemento e as inserções sempre depois do último, a fila caminha da posição inicial à final do arranjo. Com isso, depois de um determinado período inserindo e retirando, chega-se ao final do arranjo e não se pode mais inserir ou retirar elementos mesmo que ainda existam posições vagas. Para solucionar esse percalço, define-se fila de arranjo circular.

Para evitar o problema, de mesmo tendo posições livres não poder inserir elementos na fila, redefine-se fila como sendo circular, ou seja, de alguma forma ao chegar ao final da fila redireciona o corrente para o início dela.

A estrutura de dados `Fila(T)` possui campos especiais que guardam as informaçôes que indicam qual é o primeiro elemento e qual é o último elemento da `Fila(T)`. Essas são as funções dos campos `frente` e `tras` respectivamente. Assim pode-se inserir após o último elemento da `Fila(T)` utilizando-se da informação do campo `tras` e pode-se retirar o elemento da `Fila(T)` através da informação contida em `frente`.

O tipo Fila(T) é uma tupla que contém os seguintes campos: `itens`, `frente`, `tras` e `ultimaOp`. O campo `itens` é uma função que mapeia inteiros para um tipo genérico `T` que representa o tipo da informação interna do nodo. O campo `frente` e `tras` designam, respectivamente, os inteiros que são mapeiados para o primeiro e para o último elemento da `Fila(T)`. O campo `ultimaOp` informa o estado gerado pela última operação sobre a fila. A função `max` estabelece qual o valor máximo para o tamanho da `Fila`.

```
public type Fila(T) is tuple ( max: Int, itens: Int -> T, frente: Int,
                               tras: Int, ultimaOp: Status );
```

Essa estrutura é ilustrada abaixo:

Junto com o tipo `Fila` define-se o seguinte tipo:

```
type Status is enum {OK, ERRO };
```

`Status` é utilizado na geração de uma representação do estado da máquina baseado na última operação sobre esta.

As operações que caracterizam o TAD `Fila(T)` são `ffVazia, enfileira, desenfileira, houveErro, vazia` e `iniciaFila`.

5.3 Interface das Operações do Tipo Abstrato de Dados Fila(T)

As operações do tipo abstrato de dados Fila(T) apresentam a seguinte interface com o usuário:

- `ffVazia`(f: Fila(T)): cria e devolve uma fila vazia.
- `enfileira`(f: Fila(T), in x: T): dada uma informação `x`, insere-a no final da `Fila(T)` e devolve a `Fila(T)` atualizada.
- `desenfileira`(f: Fila(T), out x: T): devolve o primeiro elemento da `Fila` e a `Fila(T)` sem o primeiro elemento que foi revolvido.

- `houveErro`(in f: Fila(T), out temErro: Bool): informa se a última operação que atuou na `Fila(T)` gerou algum erro.
- `vazia` (in f: Fila(T), out vaziaF: Bool): informa se a fila está ou não vazia.

5.4 Implementação

5.4.1 Operações:

ffvazia:

Essa operação cria uma fila vazia. Considera-se uma `Fila` vazia caso o primeiro e o último elementos sejam iguais a um.

```
public action ffVazia (f: Fila(T), in max: Int) is
      f.max:= max;
      f.frente:= 1;
      f.tras:= 1;
      f.ultimaOp:= OK;
end ffVazia
```

enfileira:

A operação enfileira insere um elemento depois do último elemento da `Fila`. Verifica se há espaço para a inserção do elemento antes de inseri-lo.

```
public action enfileira(f: Fila(T), in x: T) is
     if ( f.tras mod (max+1) = f.frente ) then
          GeraErro(f);
     else
          InsereFila(f);
     end
end enfileira
```

GeraErro(f: Fila(T)):

```
f.ultimaOp:= ERRO;
```

InsereFila(f: Fila(T)):

```
f.itens(f.tras):= x;
f.tras:= f.tras mod ( max + 1 );
f.ultimaOp:= OK;
```

desenfileira:

Verifica se a `Fila` não se encontra vazia. Caso não esteja, retira o elemento que se encontra na posição marcada pela função `frente` que denota o primeiro elemento dela. Posteriormente, atualiza-se a função `frente`.

```
public action desenfileira(f: Fila(T), out x: T) is
     vaziaF: Bool;
loop:
     step 1:  VerificaFilaVazia(f, vaziaF);
     step 2: if ( vaziaF ) then
                    GeraErro(f);
             else
                    RetiraPrimeiroElemento(f);
             end
end desenfileira
```

VerificaFilaVazia(f: Fila(T), vaziaF: Bool):

```
vazia(f, vaziaF);
```

RetiraPrimeiroElemento(f: Fila(T)):

```
x:= f.itens(f.frente);
f.frente:= f.frente mod ( max + 1 );
f.ultimaOp:= OK;
```

houveErro:

Essa operação retorna qual o estado gerado pela última operação efetuada na `Fila`, ou seja, se a operação gerou ou não algum erro.

```
public action houveErro(f: Fila(T), temErro: Bool) is
     temErro:= ( f.ultimaOp!= OK );
end houveErro
```

vazia:

Operação que verifica se a fila está vazia, devolvendo o resultado em `vaziaF`. A verificação se dá com a comparação das funções `frente` e `tras`, que marcam respectivamente o início e o final da `Fila`. Caso os dois valores forem iguais, a `Fila` está vazia; caso contrário, possui pelo menos um elemento.

```
public action vazia ( in f: Fila(T), out vaziaF: Bool) is
     vaziaF:= ( f.frente = f.tras );
end vazia
```

5.5 Conclusão:

A implementação do tipo abstrato de dados `Fila` é uma versão especificada de `Lista`, em que, tem-se uma obrigatoriedade na ordem de retirada. Mas, ao contrário do que possa aparentar, isso facilita a implementação do `TAD Fila`. Foi escolhida, como implementação de `Fila`, um "vetor circular", em que se aproveita melhor o espaço de memória reservado para a `Fila`. A facilidade se encontra na marcação do primeiro elemento da fila, que será sempre o próximo elemento que será retirado. Assim, não precisa-se percorrer a `Fila` toda para se achar o elemento a ser retirado, pois é sempre o primeiro. E a inserção também é facilitada, pois possui só uma possibilidade de localização para inserção, depois do elemento da `Fila`. No entanto, esse tipo abstrato de dados, não possui a flexibilidade da `TAD Lista`, mas possui uma implementação muito mais simplificada, como se pode perceber.

Capítulo 6

Pilha

Assim como a `Fila`, a `Pilha` pode ser implementada como um caso especial de `Lista`. Como no caso de `Fila`, `Pilha` se diferencia por sua restrição de retirada de elementos. Neste caso, a restrição é que o elemento a ser retirado é o mais recentemente colocado na `Pilha`.

6.1 Definição do Tipo Abstrato de Dados Pilha e sua Representação:

O tipo `Pilha(T)` possui um elemento especial que indica qual é o elemento mais recente na estrutura, `top`. O campo `top` de pilha facilita a retirada dos elementos e sempre é atualizado quando há uma retirada ou uma inserção.

Assim como nos casos anteriores, tem-se que os nodos constituíntes da `PIlha` são mapeados através de inteiros, assim como o `top` que também é um inteiro e será mapeado da mesma forma.

O estado gerado pela última operação feita sobre a `Pilha` é armazenado como uma condição binária de erro ou não erro, sem especificações de tipos de erros.

Em `Pilha` só pode ser retirado o último elemento inserido, ou seja, de acordo com a regra: o último a entrar é o primeiro a sair, assim como em uma pilha de pratos em que podemos retirar apenas o prato de cima da pilha, isto é, o último a ser colocado na pilha. Tipo abstrato de dados `Pilha` foi implementado por meio do tipo `Stack(T)` e as operações que atuam sobre ele.

O tipo `Stack(T)` pode receber qualquer tipo de elementos, ou seja, os elementos são do tipo genérico `T`. `Stack(T)` é definido como uma tupla em que seu primeiro campo é a função `elements` que mapeia inteiros em informações do tipo T contida no nodo referenciado por esse inteiro e o segundo campo é um inteiro que referencia o topo da pilha, `top`. O terceiro

campo da tupla é uma função que guarda o estado gerado na máquina após a última operação realizada sobre `Stack(T)` e o quarto campo indica o tamanho de `Stack(T)`

```
public type Stack(T) is tuple ( elements: Int -> T, top: Int,
                                erro: Bool, max: Int);
```

Essa estrutura é ilustrada abaixo:

O tipo abstrato de dados `Stack(T)` implementado por meio de lista simplesmente encadeada e possui as seguintes operações: `initStack`, `push`, `pop`, `empty` e `status`.

6.2 Implementação

Para podermos retirar sempre o último que foi colocado na `Pilha` temos que, assim como na `Fila`, mantem de alguma forma a ordem de inserção dos elementos. Uma maneira simples de implementar a `Pilha` seria como um lista linear, pois não temos a necessidade de retirada de elemento entre elementos não-nulos, que estejam no interior da pilha. Controla-se a ordem de inserção de elementos como foi proposto fazer para fila, ou seja, a ordem crescente de índices designa a ordem cronológica de inserção na pilha. Um elemento só pode ser retirado quando todos os elementos que possuem índices menores do que o dele já tiverem sido retirados da pilha.

Como pode-se notar, nessa implementação, assim como no caso da `Fila`, não é conveniente

o uso de lista encadeada, já que não se pode retirar elementos entre dois elementos não-nulos. A conveniência se aplica na quantidade de elementos que uma pilha pode conter.Caso a pilha tenha que ser tão grande quanto a necessidade em tempo de execução, ou seja, a informação referente ao tamanho da pilha não esteja disponível em tempo de compilação, a pilha implementada usando lista encadeada é mais adequada. Pode-se perceber com isso que a pilha torna-se mais geral quando for implementada usando-se lista encadeada.

6.2.1 Interface das Operações em Pilha

As operações que atuam sobre a `Pilha` têm as seguintes interfaces com o usuário:

- `initStack`(s: Stack(T)): cria e inicializa uma pilha.
- `push`(s: Stack(T), in x: T): Insere o elemento que é passado pelo parâmetro `x` no topo `top` da pilha.
- `pop` (s: Stack(T), out x: T): Retira o elemento do topo da pilha e devolvê-o em `x`.
- `empty` (in s: Stack(T), out vazia: Bool): Verifica se a pilha em questão está ou não vazia, devolvendo o resultado da verificação.
- `status` (in s: Stack(T), out houveErro: Bool): Retorna a informação sobre o estado gerado pela última operação sobre a pilha.

6.2.2 Operações:

initStack:

Essa operação instancia uma nova estrutura `Stack(T)` que é referenciada por meio da função `s`. É passado também o tamanho de `Stack(T)` em `max`.

```
public action initStack(s: Stack(T), max: Int) is
     s.max:= max;
     s.top:= 0;
     s.erro:= false;
end initStack
```

push:

A operação `push` acrescenta um novo elemento a `Stack(T)`. Esse elemento é identificado através de um inteiro que é obtido somando-se um ao topo da pilha. Marca-se esse inteiro

como sendo o `top` de `Stack(T)`. O limite de inserção é o tamanho máximo no qual a pilha foi criada.

```
public action push (s: Stack(T), in x: T) is
begin
     s.elements(s.top + 1):= x;
     s.top:= s.top + 1;
end push
```

pop:

A operação `pop` é a operação de retirada de elementos de `Stack(T)`. O elemento que pode ser retirado é aquele que é referenciado por `top`. Após a retirada atualiza-se o topo da pilha. Devolve-se o elemento do topo de `Stack(T)` em `x`.

Testa-se, nessa ação, a existência de pelo menos um elemento em `Stack(T)`.

```
public action pop(s: Stack(T), out x: T) is
     if (s.top > 0) then
          RetiraUltimoElementoInserido(s);
          s.erro:= false;
     else s.erro:= true;
     end
end pop
```

RetiraUltimoElementoInserido(s: Stack(T)):

```
x:=  s.elements(s.top);
s.top:= s.top -1;
```

empty:

A operação `empty` devolve a resposta questão de Stack(T) está vazia.

```
public action empty(in s: Stack(T), out isEmpty: Bool) is
     isEmpty:= (s.top = 0);
end empty
```

status:

Como a informação da ocorrência de erro encontra-se escondida dentro do tipo `Stack(T)`, a única maneira de disponibilizar essa informação é através da ação `status`, que responde se foi gerado um estado de erro durante a última operação em `Stack`.

```
public action status(in s: Stack(T), out erro: Bool) is
     erro:= s.erro;
end status
```

6.3 Conclusão:

Os mesmos comentários efetuados para o `TAD Fila` podem ser adaptados para o `TAD Pilha`. Implementação simplificada, bem mais que o `TAD Lista`, em complexidade, no nível de `TAD Fila`. A diferença entre a implementação de `Fila` e `Pilha` é o elemento que será retirado. Se na `Fila`, o elemento que será retirado é o elemento que está a mais tempo nela, a `Pilha` será retirado o elemento que foi mais recentemente inserido. Assim tem-se, como na `Fila`, operações do `TAD Pilha` com complexidade menor que a da `Lista`. O `TAD Pilha` é uma especiliadade de `Lista` em que, como decisão de implementação, decidiu-se em utilizar um " vetor circular". Assim para voltar da última posição do "vetor"para a primeira posição dele, usa-se uma função chamada `mod` que dado uma posição no "vetor"devolve o resto da divisão desta com o tamanho do "vetor", propondo, dessa maneira, a circularidade na estrutura. A limitação utilizando essa implementação é justamente a limitação do tamanho da `Pilha`, já que o vetro circular possui tamanho determinado em tempo de compilação.

Capítulo 7

Árvore

Pode-se definir árvore como um conjunto de nós interconectados de acordo com o relacionamento de origem-descendência. Detentora de um nó especial, denominado nó raiz, o qual não possui nodo de origem e é a origem de todas as ramificações da árvore. Cada nodo da árvore pode ou não possuir descendentes, mas cada nodo possui apenas um nodo de origem direta.

Podemos definir, também, de forma recursiva o tipo abstrato de dados `Árvore` como sendo um conjunto de árvores conectadas em que cada nó da árvore dá origem a uma subárvore de mesmas características, em que cada nó descreve um caminho único entre raiz e qualquer nodo da árvore.

Árvore é um tipo estruturado de dados que possui uma composição hierárquica constituídas de estruturas denominados nós ou nodos. Com a exceção do nodo raiz, que dá origem à árvore, todos os nodos pertencentes a uma determinada árvore possuem um único nodo de origem direta e zero ou mais nodos que dele se originam, ou seja, nodos filhos. O nodo de origem, também chamado de nodo-pai, é o nodo que gera uma ramificação ao nodo descendente. Esse nodo-pai é o nodo hierarquiamente mais alto, mais próximo de um determinado nó. O encadeamento dessas relações diretas entre nodos produzem caminhos da raiz até todo nó pertencente à árvore, sendo esses caminhos únicos. Nós que não extendem caminhos até outro nós são denominados nodos-folha ou nós externos da árvore. Os nós que possuem ao menos um nó descendente são denominados nós internos.

A maneira mais comum de ilustrar uma árvore é através de grafo direcionado. A relação entre os nodos é direcionada do nodo pai para o nodo filho. O nó raiz é desenhado mais ao alto, seguido pelos nós conectados mais abaixo, num encadeamento hierárquico.

Não há limites a quantidade de nodos que podem se originar num determinado nó. Há uma denominação especial para árvore que tem seu número de nodos descendentes limitado para cada nó. No caso da quantidade se restringir a, no máximo, dois nós para cada nó, temos para essa árvore, a classificação de árvore binária. Em caso de, no máximo, três, árvore ternária, e assim por diante.

Um exemplo de árvore que faz parte do nosso cotidiano são árvore genealógicas e estruturas hieráquicas funcionais que designam os relacionamentos empregado-patrão em empresas.

7.1 Definição de TAD Árvore Binária de Pesquisa e sua Representação:

Em uma árvore de pequisa, cada nó possui uma chave que é única e é usada na pesquisa dentro da árvore. Em um determinado nodo, se for preciso inserir ou pesquisar por algum nodo, percorre-se a árvore utilizando-se a seguinte regra: "se a chave do nodo a ser introduzido for menor que do nó corrente, siga para o caminho da esquerda, caso contrário, para o da direita. Faça isso até chegar em um nodo que não possua mais nodos descendentes no caminho que se escolheu". Assim, mantem-se uma regra que torna a pesquisa mais eficiente.

Árvores são ordenadas, se existe uma ordem preestabelecida linear que torna possível a identificação dos nodos-filhos de um nó, isto é, sabe-se qual é o primeiro filho, segundo filho e assim por diante.

Em uma árvore binária, os nodos-filhos de um determinado nó são especificados como `nodo direito` e `nodo esquerdo`, de acordo com sua posição na árvore, a direita ou a esquerda do nodo corrente. O `nodo esquerda é raiz de uma subárvore da árvore binária, chamada de subárvore da esquerda e o mesmo acontece com o nodo direito em relação a subárvore da direita. As subárvores de uma árvore binária possui as mesmas características formadoras da árvore binária principal. Árvore é uma estrutura de dados direcionada para o armazenamento de informações que demandam uma grande eficiência na busca e inserção de dados.`

`Uma árvore binária é própria se cada nodo possui zero ou dois nodos filhos, isto é, todos os nodos internos possuem dois nodos filhos e todo nó externo não possui nós filhos.`

`Trata-se, nesse tipo estruturado de dados, os casos de retirada e de inserção de novos elementos, mas as operações mais usuais em uma Árvore Binária de Pesquisa é justamente a busca e obtenção de dados contidos na Árvore. Para tal, define-se três modos de caminhamento em uma Árvore Binária: caminhamento em pré-ordem, em ordem central e pós-ordem.`

`No caminhamento em pré-ordem, primeiramente, visita-se a raiz, em seguida a subárvoreda direita e depois a subárvore da esquerda. No caminhamento em ordem central, visita-se a subárvore da direita, a raiz e em seguida a subárvore da esquerda. E por último, no caminhamento em pós-ordem, visita-se a subárvore da direita, depois a subárvore da esquerda e só depois a raiz.`

`Em árvore, nodo corrente é o nodo em que atuará uma a chamada de ação. Assim, para poder atuar em qualquer nodo da árvore, antes é necessário torná-lo o nodo`

corrente. Apenas um nodo pode ser corrente em cada instante.

O tipo abstrato de dados Árvore Binária possui o seguinte tipo definido:

- Byte: é uma enumeração que é utilizada na definição do tipo subsequente.

  type Byte **is enum** (1, 2, 3);

- Par: é uma tupla, contendo os seguintes campos: n, que recebe uma estrutura do tipo Nodo(T), definida posteriormente; e um campo vez que recebe um valor do tipo Byte que foi definido acima. O tipo Par é utilizado no caminhamento pós-ordem.

  type Par **is tuple** (n: Nodo(T), vez: Byte);

- Nodo: O tipo seguinte define a estrutura de cada nodo da árvore. É uma tupla que possui como campos reg, esq e dir, que designam respectivamente a informação, que é do tipo Registo(T), sendo esse T de mesmo tipo do T recebido em Arvore(T), um campo esq e outro dir que são do tipo inteiro e armazenam um valor que é usado no mapeamento para o nodo correspondente esquerdo e direito.

  type Nodo(T) **is tuple** (reg: Registro(T), esq: Int, dir: Int);

Os três últimos tipos são tipos suportes do único tipo que é visível fora do módulo, o tipo Arvore(T). O tipo T, que árvore recebe, é um tipo genérico, ou seja, pode ser substituído por qualquer outro tipo, desde que tenha sua definição visível no módulo.

O tipo Arvore(T) é uma tupla que contem os seguintes campos:

- raiz: recebe um inteiro que é mapeiado para o nodo raiz.
- a: recebe um inteiro que é mapeiado para o nodo corrente da árvore.
- vetor: é uma função que dado um inteiro mapeia para o nodo correspondente.
- nodoLib: é uma pilha que recebe os inteiros que anteriormente mapeavam para nodos existentes na árvore, mas formam retirados dela.
- pre:pilha que recebe inteiros no caminhamento em pré-ordem, marcando o caminho percorrido.

- in:pilha que recebe inteiros no caminhamento em ordem central, marcando o caminho percorrido.

- pos:pilha que recebe inteiros no caminhamento em pós-ordem, marcando o caminho percorrido.

- livre:inteiro utilizado como contador. Representa o menor inteiro que está disponível para ser utilizado no mapeamento para nodo.

- status:pode ser um dos elementos da enumeração erro ou ok, definindo assim o estado gerado pela última operação realizada sobre a Arvore(T).

```
public type Arvore(T) is tuple ( raiz: Int, a: Int, vetor: Int-> Nodo(T),
                                 nodoLib: Stack(Int), pre: Stack(Int),
                                 in: Stack(Int), pos: Stack(Par),
                                 livre: Int, status: enum erro, ok );
```

Essa estrutura é ilustrada abaixo:

O tipo estruturado de dados Arvore(T) possui as seguintes operações: fEsq, fDir, corrente, inicializa, reiniciaIn, proximoIn, reiniciaPre, proximoPre, preencheParPos, reiniciaPos, proximoPos, insira, remova e pesquise.

7.1.1 Interface das Operações do Tipo Abstrato de Dados Árvore Binária de Pesquisa:

As operações do tipo abstrato de dados Árvore Binária apresentam as seguintes interfaces com o usuário:

- fEsq (in arv: Arvore(T), out no: Nodo(T)): devolve o nodo-esquerdo do nó corrente da árvore arv em no e o marca como corrente.
- fDir(in arv: Arvore(T), out no: Nodo(T)): devolve o nodo-direito do nó-corrente da árvore arv em no e o marca como corrente.
- corrente (in arv: Arvore(T), out no: Nodo(T)): devolve o nodo-corrente da árvore arv em no.
- inicializa(arv: Arvore(T)): construtora de uma árvore binária de pesquisa arv. Inicializa os campos formadores da tupla Arvore(T).
- proximoIn(arv: Arvore(T), out f: Bool, out r: Registro(T)): faz com que o próximo passo em encaminhamento em ordem central na árvore arv seja dado.
- proximoPre(arv: Arvore(T),out f: Bool, out r: Registro(T)): faz com que o próximo passo em encaminhamento em pre-ordem na árvore arv seja dado.
- proximoPos(arv: Arvore(T), out f: Bool,out r: Registro(T)): faz com que o próximo passo em encaminhamento em pos-ordem na árvore arv seja dado.
- insira(arv: Arvore(T), item: Registro(T)): insere um novo nodo, cujo o registro é dado por item, na árvore arv.
- remova(arv: Arvore(T), chv: Int): retira o nodo da árvore arv cuja chave de pesquisa seja chv passada como parâmetro.
- pesquise(arv: Arvore(T), chv: Int): procura pelo nodo que possua a chave de pesquisa chv na árvore arv, tornando esse nodo, o nodo corrente.

7.2 Implementação:

Na implementação do tipo abstrato de dados Arvore(T), referenciou-se os nodos descendentes de um determinado nodo por meio de inteiros que posteriormente, poderiam ser mapeiados em nodos. Isso se torna necessário para a simplificação da estrutura da Arvore(T), visto que, se cada nodo armazenasse os nodos descendentes, a estrutura da árvore seria inviável, por ser uma estrutura tipicamente recursiva.

É necessário guardar o valor da raiz da árvore pois todos os encaminhamentos começam a partir do nodo raiz. Sem o conhecimento de seu valor, haveriam dificuldades de se fazer uma pesquisa confiável.

As pilhas auxiliares de encaminhamento são necessárias para tornar os encaminhamentos viáveis, uma vez que guardam o caminho percorrido segundo a regra especificada para cada caso.

7.2.1 Operações:

fEsq:

Devolve o filho-esquerdo do corrente a da árvore arv em no, fazendo deste o novo corrente.

```
public action fEsq (arv: Arvore(T), no: Nodo(T)) is
    pegaNodoEsqCorrente;
end fEsq
```

pegaNodoEsqCorrente:

```
no:= arv.vetor(arv.vetor(arv.a).esq);
arv.a:= arv.vetor(arv.a).esq;
```

fDir:

Devolve o filho-direito do corrente a da árvore arv em no, fazendo deste o novo corrente.

```
public action fDir (arv: Arvore(T), no: Nodo(T)) is
    pegaNodoDirCorrente;
end fDir
```

pegaNodoDirCorrente:

```
no:= arv.vetor(arv.vetor(arv.a).dir);
arv.a:= arv.vetor(arv.a).dir;
```

corrente:

Devolve o corrente a da árvore arv em no.

```
public action corrente (arv: Arvore(T), no: Nodo(T))is
    no:= arv.vetor(arv.a);
end corrente
```

inicializa:

Inicializa uma árvore. É a construtora de árvores, inicializando os campos da tupla.

```
public action inicializa(arv: Arvore(T)) is
    num: Int;
loop:
    step 1: alocaNodo(arv, num);
    step 2: inicializaComponentesArv;
end inicializa
```

inicializaComponentesArv:

```
arv.vetor(num).esq:= 0;
arv.vetor(num).dir:= 0;
arv.raiz:= num;
arv.a:= num;
constructor(arv.nodoLib);
arv.livre:= 1;
arv.status:= ok;
```

alocaNodo:

Devolve um inteiro que pode ser associado a um nodo, dando preferência aos menores inteiros que estão em na lista arv.nodoLib. alocaNodo(in arv: Arvore(T), out num: Int): fornece um número inteiro que pode ser associado a um nodo, por meio de um mapeamento. Ou seja, um inteiro que não está sendo usado em algum mapeamento. Utilizado internamente nas ações de inicialização e de inserção de nodos na árvore.

```
action alocaNodo(arv: Arvore(T), num: Int) is
    vazia: Bool;
loop:
    step 1: empty(arv.nodoLib, vazia);
    step 2: sc pegaIntParaMap;
end alocaNodo
```

sc pegaIntParaMap:

```
if (vazia) then
    num:= arv.livre;
    arv.livre:= arv.livre + 1;
else delete(arv.nodoLib);
end
```

proximoIn:

Permite que o nodo corrente arv.a da árvore arv dê um passo no caminhamento em ordem central. Assim que não for mais possível dar passos neste caminhamento, o booleano retornado em f será false, caso contrário, será validado com o valor verdadeiro.

```
public action proximoIn(arv: Arvore(T), f: Bool, r: Registro(T)) is
    vazia: Bool;
loop:
    step 1: reiniciaIn(arv);
    step 2:  SePilhaVaziaVaParaPasso4(arv);
    step 3: arv.a:= arv.vetor(arv.a).esq;
            next:= 2;
    step 4: empty(arv.in, vazia);
    step 5:  SePilhaVaziaValorValido(vazia);
    step 6: pop(arv.in, arv.a);
    step 7: r:= arv.vetor(arv.a).reg;
    step 8: arv.a:= arv.vetor(arv.a).dir;
            f:= false;
end proximoIn
```

SePilhaVaziaVaParaPasso4(arv: Arvore(T)):

```
if (arv.a != 0) then
    push(arv.in, arv.a);
else next:= 4;
end
```

SePilhaVaziaValorValido(vazia: Bool):

```
if (vazia) then
    f:= true;
    return;
end
```

reiniciaIn:

Prepara a árvore arv para iniciar o caminhamento em ordem central. Utilizada em proximoIn.

```
action reiniciaIn(arv: Arvore(T))) is
     initStack(arv.in);
     arv.a:= arv.raiz;
end reiniciaIn
```

proximoPre:

Permite que o nodo corrente arv.a da árvore arv dê um passo no caminhamento em pre-ordem. Assim que não for mais possível dar passos neste caminhamento, o booleano retornado em f será false, caso contrário, será validado.

```
public action proximoPre(arv: Arvore(T),f: Bool, r: Registro(T)) is
     vazia: Bool;
loop:
     step 1: reiniciaPre(arv);
     step 2: empty(arv.pre, vazia);
     step 3: EmpilhaCrtSeEsteEhNuloEPilhaNaoVaziaOuVaParaPasso5(arv);
     step 4: a:= devNodo(arv.a).dir;
              next:= 3;
     step 5: SeCrtNaoNuloFazDeleDisponivelSenaoRetorna(arv, r);
     step 6: arv.a:= arv.vetor(arv.a).esq;
             f:= false;
end proximoPre
```

EmpilhaCrtSeEsteEhNuloEPilhaNaoVaziaOuVaParaPasso5(arv: Arvore(T)):

```
if ((arv.a = 0) & !(vazia)) then
     pop(arv.pre, arv.a);
else next:= 5;
end
```

SeCrtNaoNuloFazDeleDisponivelSenaoRetorna(arv: Arvore(T), r: Registro(T)):

```
if (arv.a!= 0) then
     r:= arv.vetor(arv.a).reg;
     push(arv.pre, arv.a);
```

```
     else f:= true;
          return;
     end
```

reiniciaPre:

Prepara a árvore arv para iniciar o caminhamento em pre-ordem. Utilizado em proximoPre.

```
     action reiniciaPre(arv: Arvore(T)) is
          initStack(arv.pre);
          arv.a:= arv.raiz;
     end reiniciaPre
```

proximoPos:

Permite que o nodo corrente arv.a da árvore arv dê um passo no caminhamento em pos-ordem. Assim que não for mais possível dar passos neste caminhamento, o booleano retornado em f será false, caso contrário, será validado.

```
     public action proximoPos(arv: Arvore(T), f: Bool,r: Registro(T)) is
          vazia: Bool;
          par: Par;
          aux: Bool;
          vez: Int;
     loop:
          step 1: reiniciaPos(arv);
          step 2: empty(arv.pos, vazia);
          step 3: SeCrtNuloEmpilheParEPosVaziaRetorneTrue(arv, vazia, f);
          step 4: SeCrtNuloPreenchaPar(arv, vez, par);
          step 5: TratamentoDiferentesParaDiferentesValoresDeVez(arv, vez, par);
          step 6: EmpilhaParPosContinuacaoDerpenteDoPassoAnterior(arv, aux);
          step 7: arv.a:= arv.vetor(arv.a).esq;
                  next:= 3;
          step 8: arv.a:= arv.vetor(arv.a).dir;
                  vez:= 1;
                  next:= 3;
     end proximo
```

SeCrtNuloEmpilheParEPosVaziaRetorneTrue(arv: Arvore(T), vazia: Bool, f: Bool):

```
if (arv.a = 0) then
     if (vazia) then
          f:= true;
          return;
     end
     pop(arv.pos, par);
end
```

SeCrtNuloPreenchaPar(arv: Arvore(T), vez: Int, par: Par):

```
if (arv.a = 0) then
     arv.a:= par.n.reg.chv;
     vez:= par.vez;
end
```

TratamentoDiferentesParaDiferentesValoresDeVez (arv: Arvore(T), vez: Int, par: Par):

```
case vez of
     1=> preenchePar(arv.a, 2, par);
         aux:= true;
     2=> preenchePar(arv.a, 3, par);
         aux:= false;
     3=> r:= arv.vetor(arv.a).reg;
         f:= false;
         arv.a:= 0;
         return;
end
```

EmpilhaParPosContinuacaoDerpenteDoPassoAnterior(arv: Arvore(T), aux: Bool):

```
push(arv.pos, par);
if !(aux) then
     next:= 8;
end
```

preencheParPos:

Preenche o par passado como parâmetro com os respectivos valores também passados como parâmetros. Essa operação é utilizada no encaminhamento pos-ordem.

```
action preencheParPos(x: Int, y: Num, par:Par) is
```

```
        par.n:= x;
        par.vez:= y;
    end preencheParPos
```

reiniciaPos:

Prepara a árvore arv para iniciar o caminhamento em pos-ordem. Utilizado em proximoPos.

```
    action reiniciaPos(arv: Arvore(T)) is
        initStack(arv.pos);
        arv.a:= arv.raiz;
        vez:= 1;
    end reiniciaPos
```

insira:

Dado um elemento a ser inserido item na árvore arv, pesquisa e insere o elemento no lugar apropriado de acordo com a chave do elemento.

```
    public action insira(arv: Arvore(T), item: Registro(T)) is
        v, p, num: Int;
    loop:
        step 1: pesquise(arv, chv, achou, p, v);
        step 2: SeAchouElemErro(arv, achou);
        step 3: alocaNodo(arv, num);
        step 4: PreencheOsCamposDoNodo(arv, num);
        step 5: InsereComoFilhoDeAcordoComChave(arv, item);
    end insira
```

SeAchouElemErro(arv: Arvore(T), achou: Bool):

```
    if (achou) then
        arv.status:= erro;
        return;
    end
```

PreencheOsCamposDoNodo(arv: Arvore(T), num: Int):

```
    arv.vetor(num).reg:= item;
    arv.vetor(num).dir:= 0;
    arv.vetor(num).esq:= 0;
```

InsereComoFilhoDeAcordoComChave(arv: Arvore(T), item: Registro(T)):

```
if (item.ch > arv.vetor(p).reg.ch) then
     arv.vetor(p).dir:= num;
else arv.vetor(p).esq:= num;
end
```

remove:

Remove o elemento cuja chave chv foi passada como parâmetro, da árvore arv.

```
public action remova(arv: Arvore(T), chv: Int) is
     achou: Bool;
     p: Int;
loop:
     step 1: pesquise(arv, chv, achou);
     step 2: SeNaoAchouErro(arv, achou);
     step 3: TrataCasoDeNoMaximoUmFilho(arv);
     step 4: antecessor(arv);
end remova
```

SeNaoAchouErro(arv: Arvore(T), achou: Bool):

```
if !(achou) then
     arv.status:= erro;
     return;
end
```

TrataCasoDeNoMaximoUmFilho(arv: Arvore(T)):

```
if (arv.vetor(arv.a).dir = 0 & arv.vetor(p).dir = arv.a) then
     arv.vetor(p).dir:= arv.vetor(arv.a).esq;
elseif (arv.vetor(arv.a).esq = 0 & arv.vetor(p).dir = arv.a) then
     arv.vetor(p).dir:= arv.vetor(arv.a).dir;
elseif (arv.vetor(arv.a).dir = 0 & arv.vetor(p).esq = arv.a) then
     arv.vetor(p).esq:= arv.vetor(arv.a).esq;
elseif (arv.vetor(arv.a).esq = 0 & arv.vetor(p).esq = arv.a) then
     arv.vetor(p).esq:= arv.vetor(arv.a).esq;
end
if (arv.vetor(arv.a).dir = 0 | arv.vetor(arv.a).esq = 0 ) then
     return;
end
```

Busca o elemento o menor elemento da subárvore da esquerda para substituir o elemento que possua dois filhos.

ANTECESSOR(arv: Arvore(T)):

```
action antecessor (arv: Arvore(T)) is
    p,v: Int;
loop:
    step 1: p:= arv.a;
    step 2: v:= arv.vetor(arv.a).esq;
    step 3: AchaElemMaisDireita(arv, v, p);
    step 4: arv.vetor(arv.a):= arv.vetor(v);
            arv.vetor(p).dir:= arv.vetor(v).esq;
end antecessor
```

AchaElemMaisDireita(arv: Arvore(T), v: Int, p: Int):

```
if !(arv.vetor(v) = 0) then
    v:= arv.vetor(v).dir;
    p:= v;
    next:= 2;
end
```

liberaNodo:

Desassocia um inteiro num do nodo que mapeava na árvore arv. Utilizado em ações de retiradas de nodos da árvore.

```
action liberaNodo(arv: Arvore(T), num: Int) is
    insertLeft(arv.nodoLib, num);
    arv.vetor(num).reg.ch:= 0;
end liberaCel
```

pesquisa:

Procura pelo elemento que possua a chave chv, devolve um booleano informando o resultado da pesquisa, devolvendo também os inteiros que mapeiam para o nodo de origem do nodo (pai do corrente) e o nodo de origem deste último (avô do corrente).

```
public action pesquise(in arv: Arvore(T), in chv: Int, out achou: Bool, out p: Int, o
loop:
    step 1: InicializaCrtPaiAvo(arv, p, v);
```

```
        step 2: SeCrtNuloErro (arv);
        step 3: ProcuraNodo(arv, p, v);
    end pesquise
```

SeCrtNuloErro(arv: Arvore(T)):

```
    if (arv.a = 0) then
        achou:= false;
        return;
    end
```

InicializaCrtPaiAvo(arv: Arvore(T), p: Int, v: Int):

```
    arv.a:= arv.raiz;
    p:= 0;
    v:= 0;
```

ProcuraNodo(arv: Arvore(T), p: Int, v: Int):

```
    if (arv.vetor(arv.a).reg.ch < chv) then
        v:=p;
        p:= arv.a;
        arv.a:= arv.vetor(arv.a).dir
        next:= 2;
    elseif (arv.vetor(arv.a).reg.ch > chv) then
        v:=p;
        p:= arv.a;
        arv.a:= arv.vetor(arv.a).esq
        next:= 2;
    else achou:= true;
    end
```

7.3 Conclusão:

No TAD Arvore representa uma árvore binária de pesquisa. Nesse TAD tem-se o inconveniente de ser uma estrutura tipicamente recursiva. O nodo-raiz dá origem a um nodo-esquerdo e um nodo-direito e cada nodo citado dá origem a mais um par de nodos esquerdo e direito e assim sucessivamente. Isso torna-se um problema, pois em Machĩna não se têm tipos recursivos.

Superou-se esse problema, por meio de mapeamento de inteiro para nodo. Tem-se assim uma tabela que associa inteiro e nodo. Assim a estrutura Nodo que possui dois nodos descendentes fica simplicado e conciso, tornando a estrutura mais legível.

Como esse TAD é conceituamente mais complexo que os anteriores: as retiradas são mais complicadas, pois existem vários casos que podem ocorrer e em cada caso deve-se ter um procedimento diferente. Casos em que temos um único nodo descendente e caso em que existem dois nodos descendentes. No primeiro caso, o nodo que dá origem ao nodo que se quer retirar, toma como nodo descendente (seja esquerdo ou direito) como nodo diretamente descendente. O caso de o nodo a ser retirado possuir dois nodos, é um caso mais complexo, em que acha-se o nodo anterior a todos os nodos da subárvore da direita ou o nodo posterior na subárvore da esquerda. No caso, dessa implementação, optou-se por achar o nodo anterior.

No caso de inserção na Árvore Binária de Pesquisa, pesquisa-se, primeiramente a localização em que deve ser inserido o nodo. Como é uma árvore de pesquisa, segue a regra dos posicionamentos das chaves de acordo com seu tamanho, a inserção é uma operação muito mais simples que a retirada, possuindo apenas um caso de inserção.

Capítulo 8

Árvore Digital

A árvore digital caracteriza-se por possuir chaves que são formadas por sequência de dígitos binários ou caracteres. Esse tipo de árvore binária é preferível quando temos chaves de tamanho grande ou variável. Os exemplos mais marcantes de árvore digital são as árvores do tipo trie e Patricia.

Na árvore digital, é necessário que se tenha, no caminho da raiz a um determinado nó, a descrição da chave inteira. Todo nodo no nível i contem a mesma subsequência de tamanho i da chave.

A árvore Patricia será mais detalhadamente expicada abaixo, pois foi implementada em Machĩna. A definição de árvore trie ajuda no maior entendimento de árvore Patricia, pois ela é um caso particular de Trie.

Árvore Trie, também conhecida como radix searching tree, é uma árvore n-ária em que o fator de subdivisão, ou número máximo de filhos por nó é igual ao número de símbolos do alfabeto, sendo que o alfabeto pode ser qualquer conjunto de símbolos. Cada filho de um nó representa um símbolo do alfabeto que se torna necessário naquela posição. A sequência de caracteres gerada pelo caminhamento na árvore desde a raiz até o nodo que se deseja é a chave do nó. Cada nó possui uma chave que é única, mas pode ocorrer uma subsequência que seja comum a um ou mais nodos. Na inserção de um nodo, não é necessário que o caminho da raíz até o nodo descreva o valor completo da chave, mas pelo menos um subconjunto da sequência de caracteres da chave.

A inserção em uma árvore trie se dá pela formação do menor caminho que se pode gerar da raiz até a folha aonde irá se situar o novo nodo. Com isso, se tivermos na árvore trie algum nó que tenha na chave uma subsequência de caracteres inicial igual ao do novo nó, gera-se o caminho: a sequência de caracteres desde o primeiro caracter até o primeiro caracter diferente. Em certos casos, isso pode gerar uma árvore desbalanceada, pois o caracter diferente pode estar distante do caracter

gerado até então. Esse problema é superado com a árvore Patricia.

Árvore Binária Patricia é um caso especial de Árvore Digital, mais especificamente, um caso especial de Árvore Trie. A diferença entre Árvore Patricia e Árvore Trie é o caminho até o caracter da diferença. No caso de Árvore Patricia não precisa aparecer o caminho completo da raiz até o nodo de interesse, ao contrário da Trie. Nesse caso o caminho mínimo tem um significado diferente. Caminho mínimo, em Árvore Patricia, é o caminho que contem apenas os caracteres de diferença e não os caracteres até os que se diferenciam como não caso da Árvore Trie. Com isso, corrige-se os casos em que temos longas sequências de caracteres iguais em nodos distintos até a posição de interesse para a realização de alguma operação, tal como pesquisa, inserção ou retirada.

Capítulo 9

Árvore Patricia

9.1 Definição de Árvore Patricia

Árvore Patricia é uma árvore de pesquisa n-ária, em um caso particular, é uma árvore de pesquisa binária, pois tem-se chaves, que são sequência de dois tipos de dígitos: 0 ou 1. Para alfabetos maiores que esse, tem-se tantos nodos descendentes para cada nodo quantos são os símbolos que formam o alfabeto.

Na Árvore Patricia Binária tem-se dois tipos de nodos: os nodos internos e os externos. Os nodos internos servem para armazenar a posição na sequência de dígito da chave. No caso de ser uma árvore binária, esse dígito pode ser 0, que dará origem ao nodo descendente à esquerda ou 1, que dará origem ao nodo descendente à direita. O nodo descendente pode ser interno ou externo. Se na árvore houver mais de um nodo com a mesma sequência de dígitos estabelecida até então, tem-se mais um nodo interno que poderá dar origem a mais um nodo interno ou externo, de acordo com essa regra estabelecida. O nodo interno guarda a posição do dígito binário na sequência de dígitos que formam a chave. Assim, se em uma determinada sequência de digitos binários (bits), tiver o valor 0, o nodo seguirá como um nodo-esquerd caso contrário, como um nodo-direito. Os nodos externos são os que efetivamente guardam a informação de interesse na pesquisa. Os nodos internos não contem informações resgatáveis na pesquisa, apenas mostram o caminho para chegar à elas, atráves da chave e da sequência de dígitos até um nodo externo.

Os nodos internos contêm a informação de qual é o caracter da diferença das chaves envolvidas que dá origem as ramificações. No caso da chave ser uma sequência de dígitos binários, os nodos internos informam qual o bit da diferença entre as chaves de dois nodos. A chave, em que numa determinada posição contiver o valor 1 é o nodo direito e caso contrário é o nodo esquerdo de um dado nó interno. Mas pode-se ter casos em que os caracteres das chaves não são digitos binários, podendo ter, então, mais de dois nodos-filhos. Isso depende, portanto, do alfabeto utilizado

na construção das chaves.

9.2 Tipo Abstrato de Dados Árvore Patricia

O tipo abstrato de dados Patricia é uma árvore de pesquisa digital binária, com o alfabeto { 0 ,1 }, onde nodos internos guardam a posição de digito de diferença entre chaves e os nodos externos guerdam as informações de interesse de pesquisa. As operações que atuam sobre o tipo abstrato de dados Patricia é a criação de uma árvore Patricia, a inserção de nodos, a pesquisa para o resgate de informações contidas na árvore e a retirada de nodos.

O ArvPatricia é um módulo que contém os tipos definidos que dão suporte ao tipo Patricia conjugados com as operações que atuam nesse tipo, que serão depois descridos.

Há nesse módulo a noção de nodo corrente na pesquisa na árvore, ou seja, o nodo corrente é o nodo em que se tem a informação, seja interno ou externo. O nodo corrente é o nodo em que se atuam as operações que estão sendo processadas.

A posição armazenada no nodo interno se refere à posição do dígito que difere entre dois nodos externos que possuem a mesma sequência de digitos nas respectivas posições já inseridas na árvore. Já a informação contida no nodo externo é genérica, dependendo somente de como for criada a árvore, isto é, se for criada uma árvore que contenha nodos externos que armazenam informações do tipo inteiro, tipo caracteres e assim por diante.

9.3 Implementação e Representações:

Para a criação do módulo que contém o tipo abstrato de dados Patricia, que representa a implementação de uma árvore Patricia, foram criadas as seguintes funções e tipos internos que dão suporte ao tipo público Patricia:

Funções Externas:

- ord: Char -> Int: função que dado um caracter devolve seu valor em códico ASCII.
- bit: (Int, Chave) -> Int: função que retorna o i-ésimo bit da chave k.
- difBit: (String, String)-> Int: função que dados dois strings, devolve zero em caso dos strings serem iguais ou devolve o primeiro bit diferente da esquerda para a direita.

Funções Estáticas:

- nodoExt(p: Patricia): Bool:= with patNo(p) as
  PatNodoExt=> ehExt:= true;
  otherwise=> ehExt:= false;
  end:

  função que dado um inteiro, devolve um valor booleano que indica se o nodo correspondente a esse inteiro é um nodo externo ou interno.

- max: Int:= 30: função que fornece o valor de caracteres que as chaves da árvore Patricia devem ter.

- d:= 8*max: função que fornece o número de bits que as chaves da árvore Patricia pode conter.

- chaveOk(k: String) : Bool:= length(k) = max: função que verifica se o tamanho de uma determinada chave é igual ao valor estipulado para ela nesta árvore, validando-a.

Tipos Declarados como Públicos:

- Patricia is Int: o tipo Patricia é definido como um inteiro que mapeia para um nodo da Árvore Patricia, sendo este seu nodo corrente.

- Chave is String: o tipo Chave é utilizado em todos os nodos externos, sendo cada função única para cada nodo para ser possível a pesquisa em Árvore Patricia.

Essas estruturas são ilustradas abaixo:

Tipos declarados como internos ao módulo:

- Nodos is PatNodoInt | PatNodoExt: o tipo Nodos pode representar tanto um nodo interno quanto um nodo externo.
- PatNo is Int -> Nodos: o tipo PatNo dado um inteiro mapeia-o para o nodo correspondente, sendo ele interno ou externo.
- PatNodoExt(T) is tuple(ch: Chave, //outras informações: T): o tipo PatNodoExt representa um nodo do tipo externo, sendo uma tupla que possui o campo ch do tipo Chave, para pesquisa e outros campos que podem ser modificados de acordo com a aplicação.
- PatNodoInt is tuple(index: Int, esq: Patricia, dir: Patricia): o tipo PatNodoInt representa um nodo que é do tipo interno, sendo uma tupla que possue os campos index, que representa o bit da diferença entre duas chaves, e os campos esq e dir que são do tipo Patricia e podem ser mapeiados para intidades do tipo Nodos.

Funções dinâmicas:

- livre: Int:= 1: livre é uma função inteira usada na aloção de nodos.
- nodoLib: Stack(Int): é uma pilha de inteiros, sendo utilizada na desalocação de nodos.

- erro: Bool:= false: função do tipo booleano que é utilizada em todo módulo na verificação de ocorrência de erros.

9.3.1 Interfaces das Operações em Patricia

O tipo estruturado Patricia possui as seguintes operações básicas:

- inicialize(r: Patricia, in chvEsp: Chave): constrói uma árvore Patricia, alocando um nodo interno e outro externo e inicializando os campos referentes a eles.
- pesquisa(in k: Chave, in t:Patricia, q: Patricia): dada a chave de pesquisa k, pesquisa-se na árvore Patricia t e devolve o resultado da pesquisa em q.
- insira(in k: Chave, t: Patricia): insere chave k passada como parâmetro na árvore Patricia t.
- retira(k: Chave, t: Patricia): retira a chave k passada como parâmetro da árvore Patricia t.

9.4 Operações:

9.4.1 inicialize:

Aloca par de nodos para que um nodo seja inserido na Árvore Patricia. O par de nodos é composto por um nodo do tipo interno ligado a um nó do tipo externo.

```
public action inicialize(r: Patricia, chvEsp: Chave) is
    q: Patricia;
    x1: PatNodoInt; x2: PatNodoExt;
    num, num1: Int;
loop:
    step 1: AlocaNodos(r, q);
    step 2: PreencheNodos(x1, x2, chvEsp, r, q);
end inicialize
```

AlocaNodos(r: Patricia, q: Patricia):

```
alocaNodo(r);
alocaNodo(q);
```

PreencheNodos(x1: PatNodoInt, x2: PatNodoExt, chvEsp: Chave, r: Patricia, q:Patricia):

```
x1.index:= 0; x1.esq:= q;
x1.dir:= 0; x2.ch:= chvEsp;
patNo(r):= x1;patNo(q):= x2;
```

alocaNodo:

Aloca um inteiro que representará uma Árvore Patricia. Essa operação interna ao módulo é utilizanda tanto na operação inicialize quanto na insira

```
action alocaNodo(p: Patricia) is
     vazia: Bool;
loop:
     step 1: VerificaPilhaVazia;
     step 2: if (vazia) then
                    PegaInteiro(p);
             else  Desempilha (p);
             end
end alocaNodo
```

VerificaPilhaVazia:

```
empty(nodoLib, vazia);
```

PegaInteiro(p: Patricia):

```
p:= livre;
livre:= livre + 1;
```

Desempilha (p: Patricia):

```
 pop(nodoLib, p);
```

9.4.2 pesquisa:

Dada uma chave, a operação pesquisa procura pela árvore Patricia t e devolve o inteiro que é mapeado para o nodo procurado em q.

```
public action pesquisa(k:Chave, in t:Patricia, q:Patricia) is
     y: Int;
loop:
     step 1: TestaChave(k);
     step 2: ProcuraNodoExt(t, k);
     step 3: DevolveResultadoPesquisa(k, t, q);
end pesquisa
```

TestaChave(k: Chave):

```
if !(chaveOk(k)) then
     erro:= true;
     return;
else erro:= false;
end
```

ProcuraNodoExt(t: Patricia, k: Chave):

```
if !(nodoExt(t)) then
     if (bit(patNo(t).index, k) = 0) then
          t:= patNo(t).esq;
     else t:= patNo(t).dir;
     end
     next:= 2;
end
```

DevolveResultadoPesquisa(k: Chave, t: Patricia, q: Patricia):

```
if (equals(k, patNo(t).ch) then
     q:= t;
else q:= 0;
end
```

9.4.3 insira:

Essa operação acha a localização em que o nodo cuja chave k, que foi passada como parâmetro,em que deve ser inserido na árvore Patricia t. Essa operação cria o par de nodos interno-externo que deverá será inserido.

```
public action insira(k: Chave, t: Patricia) is
     p: Stack(Patricia);
     erro: Bool;
     i: Int:= 0;
```

```
        y, x: Int;
        v, r, q: Patricia;
        h, ehExt: Bool;
        x1: PatNoInt;
        x2: PatNoExt;
    loop:
        step 1: localizaExterno(k, t, q, p, i, achou);
        step 2: TestaNodoPresente(achou);
        step 3: crieNodos(k, patNo(q).ch, i+1, v, h);
                pop(p, q);
        step 4: RetornaSeErro;
        step 5: AcharLugarInsercao(p, q, v);
        step 6: InserePar(q,r,v);
        step 7: ReconstitueArvorePosInsercao(k, v, r);
    end insira
```

TestaNodoPresente(achou: Bool):

```
    if (achou) then
        return;
        erro:= true;
    end
```

RetornaSeErro:

```
    if (erro) then
        return;
    end
```

AcharLugarInsercao(p:Stack(Patricia), q: Patricia, v: Patricia):

```
    if patNo(q).index >= patNo(v).index then
        pop(p, q);
        next:= 5;
    end
```

InserePar(q: Patricia,r: Patricia,v: Patricia):

```
    if bit(patNo(q).index, k) = 0 then
        r:= patNo(q).esq;
        patNo(q).esq:= v;
    else r:= patNo(q).dir;
        patNo(q).dir:= v;
    end
```

ReconstitueArvorePosInsercao(k: Chave, v: Patricia, r: Patricia):

```
if (bit(patNo(v).index, k) = 0) then
     patNo(v).dir:= r;
else patNo(v).esq:= r;
end
```

localizaExterno:

Dada um determinada árvore Patricia t, localiza o nodo externo cuja chave k foi passada como parâmetro. Essa operação não é pública, não fazendo parte, pois, da coleção de operações do tipo abstrato de dados Árvore Patricia.

```
action localizaExterno(k: Chave, t: Patricia, q: Patricia,p: Stack(Patricia),
                                                        i: Int, achou: Bool) is
loop:
     step 1:  INICIALIZACOES(p, k, i, q, t);
              achou: false;
     step 2: if !(nodoExt(q)) then
                  BUSCANODOEXTERNO(p: Patricia,
                                   q: Patricia, t: Patricia, i: Int, k: Chave);
                next:= 2;
              else achou:= true;
             end
end localizaExterno
```

INICIALIZACOES(p: Stack(Patricia), k: Chave, i: Int, q: Patricia, t: Patricia):

```
initStack(p);
i:= 0;
 TESTACHAVE(k);
q:= t;
```

TESTACHAVE(k: Chave):

```
if !(chaveOk(k))then
     erro:= true;
     return;
end
```

BUSCANODOEXTERNO(p: Patricia, q: Patricia, t: Patricia, i: Int, k: Chave):

```
push (p, q);
```

```
// ultimo indice de prefixo iguais
if (patNo(q).index = i+1) then
     i= i+ 1;
end
if (bit(patNo(q).index, k) = 0) then
     t:= patNo(q).esq;
else t:= patNo(q).dir;
end
```

crieNodos:

Cria par de nodos interno-externo para a inserção em uma árvore Patricia e devolve em v o inteiro que mapeia para o nodo interno. É utilizado na ação de inserção de nodos.

```
action crieNodos(k:Chave, ka:Chave, i:Int, v:Patricia, h:Bool) is
     p, q: Patricia;
     b: Int;
     x1: PatNoInt;
     x2: PatNoExt;
loop:
     step 1: TestaChaves(k, ka);
     step 2: AchaBitDiferenca(b, k, ka);
     step 3: SeChavesDifAlocaParNodos(b, h, p, q);
     step 4: PreencheNodos(x1, x2, b, k, q, p, v);
end crieNodos
```

TestaChaves(k: Chave, ka: Chave):

```
if !( chaveOk(k) and chaveOk(ka)) then
     erro:= true;
     return;
end
```

AchaBitDiferenca(b: Int, k: Chave, ka: Chave):

```
b: difBit(k, ka);
```

SeChavesDifAlocaParNodos(b: Int, h: Bool, p: Patricia, q: Patricia):

```
if (b = 0) then
     h:= true;
     return;
```

```
     else alocaNodo(p);
          alocaNodo(q);
          h:= false;
     end
```

PreencheNodos(x1: PatNoInt, x2: PatNoExt, b: Int, k: Chave, q: Patricia, p: Patricia, v: Patricia):

```
     patNo(p):= x1;
     patNo(q):= x2;
     x1.index:= b;
     x2.ch:= k;
     if ( bit( b, k) = 1) then
          x1.dir:= q;
          x1.esq:= 0;
     else x1.dir:= 0;
          x1.esq:= q;
     end
     v:= p;
```

9.4.4 retira:

Essa operação retira o nodo, cuja chave k foi passada como parâmetro, da árvore Patricia t. Trata dos quatro caso de remoção de nodos em uma árvore Patricia: casos em que se tem um filho ou nenhum, sendo que ele pode ser esquerdo ou direito, e o caso em que se tem dois filhos.

```
     public action retira(k: Chave, t: Patricia) is
          a, p, r: Patricia;
          ehExt: Bool;
     loop:
          step 1: pesquisaRet(a, p, t, r, k);
          step 2: SeNaoAchouErro(r);
          step 3: TrataRetiradas(a, p, q, r);
     end retira
```

SeNaoAchouErro(r: Int):

```
     if (r = 0) then
          erro:= true;
          return;
     else erro:= false;
     end
```

TrataRetiradas(a: Patricia, p: Patricia, q: Patricia, r: Patricia):

```
if (r = patNo(p).esq) then
     q:= patNo(p).dir;
else  q:= patNo(p).esq;
end
if (p = patNo(a).esq) then
     patNo(a).esq:= q;
else patNo(a).dir:= q;
end
liberaNodo(r);
liberaNodo(p);
```

liberaNodo:

Essa operação desassocia o inteiro com o seu respectivo nodo, tornando-o disponível para outra inserção.

```
public action liberaNodo(p: Patricia) is
     push(nodoLib, p);
end liberaCel
```

pesquisaRet:

Essa operação não faz parte da coleção de operações do tipo abstrato de dados Árvore Patricia, portanto é usada apenas internamente ao módulo de criação do TAD.

```
action pesquisaRet(k: Chave, in t: Patricia, a: Patricia,p: Patricia,
                                                    r: Patricia) is
loop:
     step 1:  Inicializacoes(a, t, p, r, k);
     step 2: nodoExt(t, ehExt);
     step 3: if (! ehExt) then
                    ProcuraPelaChv(a, p, t, k);
             end
     step 4:  DevolveResultadoPesquisa(k, t, r);
end pesquisaRet
```

Inicializacoes(a: Patricia, t: Patricia, p: Patricia, r: Patricia, k: Chave):

```
a:= t; p:= t; r:= 0;
```

```
 TestaChave(k);
erro:= false;
```

TestaChave(k: Chave):

```
if !(chaveOk(k)) then
     erro:= true;
     return;
end
```

ProcuraPelaChv(a: Patricia, p: Patricia, t: Patricia, k: Chave):

```
a:= p;
p:= t;
if (bit(patNo(t).index, k) = 0) then
     t:= patNo(t).esq;
else t:= patNo(t).dir;
end
next:= 2;
```

DevolveResultadoPesquisa(k: Chave, t: Patricia, r: Patricia):

```
if (equals(k, patNo(t).ch)) then
     r= t;
     erro:= false
else erro:= true;
end
```

9.5 Conclusão:

Na implementação do tipo abstrato de dados Patricia, que representa uma árvore digital de pesquisa binária Patricia. Tem-se dois tipo de estruturas de nodos: internos e externos. Os nodos internos são apenas o caminho para o nodo em que se tem interesse de se resgatar as informações, ou seja, nodos externos. A necessidade de ter dois tipos de nodos implica na dificuldade da implementação da árvore Patricia. Por vezes, identificar de qual tipo de nodo que se trata passa a ser um problema, superado com a função que identifica se o nodo é do tipo externo.

A operação que se refere a pesquisa se diferencia da operação de pesquisa referente à retirada de nodos, pois precisa-se de mais informação sobre o caminho até um determinado nodo no momento em que se quer retirar um nodo do que quando a questão é apenas achar o nodo. Por isso que foi necessário formular duas operação de pesquisa específica para a retirada e tratar casos de retiradas de nodos de maneira mais

simplificadas que os casos de retirada em árvore de pesquisa binária. No entanto, a inserção de um nodo em uma Árvore de Pesquisa Binária Patricia é mais complicada que em uma Árvore Binaria de Pesquisa, pois nessa árvore tem-se que introduzir um par de nodos interno-externo, além de achar o ponto de inserção do par em que fique coerente com as outras chaves já presentes na Árvore Binária Patricia.

Capítulo 10

Árvore SBB

Árvore SBB é uma árvore binária de pesquisa extendida em que os seus apontadores de subárvores podem ser de dois tipos: horizontal e vertical. Com isso, entende-se que cada nodo pode ter no máximo dois descendentes e que a referência a esses nodos pode ser dada através de um apontadores vertical ou horizontal, limitando-se o uso dos apontadores da seguinte maneira:

- Todos os caminhos que levam algum nodo folha até a raíz devem conter o mesmo número de apontadores verticais.
- Não se pode ter mais de dois apontadores horizontais consecutivos.

Essas regras garantem um melhor balanceamento de nodos do em que árvores binárias comuns. Esse balanceamento é mantido através de manutenções nos algoritmos de retirada e inserção de nodos. Após retiradas ou inserções podem aparecere apontadores horizontais sucessivos que devem ser manipulados de modo a manter as regras expostas acima.

A inserção ou a retirada de um nó é sempre feita após o apontador vertical mais baixo da árvore.

10.1 Tipo Abstrato de Dados Árvore SBB:

O tipo abstrato de dados SBB é formado pelo tipo SBB e as operações de criação, inserção, retiradas e pesquisa de nodos. O nodo desta árvore é mais complexo que os outros tipos de nodos vistos, já que além da informação guardada e a chave única de pesquisa do nodo, tem-se as inclinações dos ‘‘apontadores"que fazem referencia aos nodos-filhos.

Ao introduzir as regras de utilização dos tipos de apontadores, dificulta-se a inserção e retirada de nodos, pois a cada modificação Árvore SBB, tem-se manter as regras estabelecidas. Para tal, será mostrado todos os casos de retirada, inserção e possíveis situações resultantes, resolvendo o balanceamento dos nodos em relação aos tipos de ‘‘apontadores".

10.2 Implementação e sua Representação:

O módulo de criação do tipo abstrato de dados SBB inclui do módulo Registros, o tipo Registro(T), que será utilizado na composição do tipo Nodo(T).

São definidos os seguintes tipos, sendo todos acessíveis ao usuário:

- Inclinacao is public enum(vertical, horizontal): tipo que é uma enumeração das opções de inclinação de ‘‘ponteiros".
- Nodo(T) is tuple(reg: Registro(T), esq: Int, dir: Int, bitE: Inclinacao, bitD: Inclinacao): tipo que é formado por uma tupla que contem os campos reg, que contem o registro do nodo do tipo Registro(T), o campo esq e dir que são do tipo Int e são mapeiados para Nodo(T); e bitD e bitE que informam qual é o tipo de inclinação dos apontadores para os nodos direito e esquerdo respectivamente.
- NoSBB(T) is Int -> Nodo(T): tipo que mapeia um dado inteiro para um Nodo(T).
- SBB is Int: tipo que representa o nodo corrente de uma árvore SBB, sendo do tipo inteiro.
- Dicionario is SBB: tipo que é expressado como uma árvore SBB.

Essas estruturas são ilustradas abaixo:

10.3 Interfaces das Operações de SBB:

As operações do tipo abstrato de dados SBB são as que se seguem:

- inicializa(dicionario: Dicionario):iniciaza dicionario que é do tipo SBB.
- insere(in x: Registro(T), ap: SBB, iap: Incinacao, out fim : Bool): insere um registro x ma árvore SBB ap.
- retira (x: Chave, ap: SBB): retira o nodo cuja chave x foi passada como parâmetro na árvore SBB ap
- pesquisa(in k: Chave, ap: SBB, achou: Bool): verifica se o nodo cuja chave k está na árvore SBB ap, tornando esse nodo o corrente em caso afirmativo. Caso contrário não modifica o nodo corrente da árvore antes da pesquisa.

10.3.1 Operações:

inicializa:

Essa operação iniciaza a Árvore SBB como dicionario.

```
public action inicializa (dicionario: Dicionario) is
    dicionario:= 0;
end inicializa
```

As próximas quatro operações são operações referentes a transformações em que temos casos em que a Árvore SBB possui dois apontadores horizontais consecutivos. Esses casos podem aparecer depois de uma inserção ou uma retirada, quando há a possibilidade da árvore ficar desbalanceada.

transEE

: Trata o caso em que temos dois apontadores horizontais à esquerda, ou seja, a esquerda de um nodo e a esquerda do nodo anteriormente citado.

```
action transEE (ap: SBB) is
    ap1: SBB;
loop:
    step 1: ap1:= NoSBB(ap).esq;
    step 2: SobeNodoCentral(ap, ap1);
end transEE
```

SobeNodoCentral(ap: SBB, ap1: SBB):

```
NoSBB(ap).esq:= NoSBB(ap1).dir;
NoSBB(ap1).dir:= ap;
NoSBB(ap1).bitE:= vertical;
NoSBB(ap).bitE:= vertical;
ap:= ap1;
```

transED:

Operação de balanceamento de Árvore SBB quando temos o caso em que temos um apontador horizontal a esquerda de um nodo e o filho direito desse nodo também é horizontal.

```
action transED (ap: SBB) is
    ap1: SBB;
    ap2: SBB;
loop:
    step 1: ap1:= NoSBB(ap).esq;
    step 2: ap2:= NoSBB(ap1).dir;
            VerticalizaApontadores(ap, ap2, ap1);
```

```
        step 3:  ManipulacaoPonteiros(ap, ap1, ap2);
        step 4:  ManApontFazCentralCrt(ap, ap2);
    end transED
```

VerticalizaApontadores(ap: SBB, ap2: SBB, ap1: SBB):

```
    NoSBB(ap1).bitD:= vertical;
    NoSBB(ap).bitE:= vertical;
    NoSBB(ap1).dir:= NoSBB(ap2).esq;
```

ManipulacaoPonteiros(ap: SBB, ap1: SBB, ap2: SBB):

```
    NoSBB(ap2).esq:= ap1;
    NoSBB(ap).esq:= NoSBB(ap2).dir;
```

ManApontFazCentralCrt(ap: SBB, ap2: SBB):

```
    NoSBB(ap2).dir:= ap;
    ap:= ap2;
```

transDD:

Operação que balanceia a Árvore SBB no caso em que temos um apontador horizontal a direita do nodo corrente e outro apontador horizontal no filho direito desse nodo corrente.

```
    action transDD (ap: SBB) is
        ap1: SBB;
    loop:
        step 1: ap1:= NoSBB(ap).dir;
        step 2:  SobeNodoCentralManApont(ap, ap1);
    end transDD
```

SobeNodoCentralManApont(ap: SBB, ap1: SBB):

```
    NoSBB(ap).dir:= NoSBB(ap1).esq;
    NoSBB(ap1).esq:= ap;
    NoSBB(ap1).bitD:= vertical;
    NoSBB(ap).bitD:= vertical;
    ap:= ap1;
```

transDE:

Operação que trata do caso em que temos no filho direito do nodo corrente, um apontador horizontal e o filho esquerdo desse nodo-filho também horizontal.

```
action transDE (ap: SBB) is
      ap1: SBB;
      ap2: SBB;
loop:
      step 1: ap1:= NoSBB(ap).dir;
      step 2: ap2:= NoSBB(ap1).esq;
                VertilizaApont(ap, ap1, ap2);
      step 3:  SobeCentralManApont(ap, ap1, ap2);
 end transDE
```

VertilizaApont(ap: SBB, ap1: SBB, ap2: SBB):

```
NoSBB(ap1).bitE:= vertical;
NoSBB(ap).bitD:= vertical;
```

SobeCentralManApont(ap: SBB, ap1: SBB, ap2: SBB):

```
NoSBB(ap1).esq:= NoSBB(ap2).dir;
NoSBB(ap2).dir:= ap1;
NoSBB(ap).dir:= NoSBB(ap2).esq;
NoSBB(ap2).esq:= ap;
ap:= ap2;
```

insere:

Essa operação insere um nodo na Árvore SBB e trata dos casos em que aparecem dois apontadores horizontais consecutivos.

```
public action insere(in x: Registro(T), ap: SBB, iap: Incinacao, out fim : Bool) is
     nodo: Nodo(T);
     num: Int;
loop:
     step 1: if (ap = 0) then
                   InsereComoFilhoAP(nodo, num, iap);
             end
     step 2: if (ap = 0) then
                   ContInsercaoComoFilhoAP(nodo, fim);
             end
```

```
        step 3: if (ap != 0) then
                    ProcuraLugarInserirEsquerda(ap, iap, x);
                end
        step 4: if (ap != 0) then
                    PossivelTransfEEouEDEsqAP( ap, iap, x);
                end
        step 5: if (ap != 0) then
                    ProcuraLugarInserirDireita(ap, iap, x);
                end
        step 6: if (ap != 0) then
                    PossivelTransfDEouDDDirAP( ap, iap, x);
                end
    end insere
```

InsereComoFilhoAP(nodo: Nodo(T), num: Int, iap: Inclinacao):

```
    alocaNodo(nodo, num);
    iap:= horizontal;
    erro:= false;
```

ContInsercaoComoFilhoAP(nodo: Nodo(T), fim: Bool):

```
    nodo.reg: x;
    nodo.bitE:= vertical;
    nodo.bitD:= vertical;
    nodo.esq:= 0;
    nodo.dir:= 0;
    fim:= false;
```

ProcuraLugarInserirEsquerda(ap: SBB, iap: Inclinacao, x: Registro(T)):

```
    if (x.ch < NoSBB(ap).reg.ch) then
        ap:= NoSBB(ap).esq;
        iap:= NoSBB(ap).bitE;
        next:= 1;
    end
```

PossivelTransfEEouEDEsqAP(ap: SBB, iap: Inclinacao, x: Registro(T)):

```
    if (x.ch < NoSBB(ap).reg.ch) then
        transEEEDEsqAp(fim, ap, iap);
    end
```

ProcuraLugarInserirDireita(ap: SBB, iap: Inclinacao, x: Registro(T)):

```
if !(x.ch < NoSBB(ap).reg.ch) then
     if (x.ch > NoSBB(ap).reg.ch) then
          ap:= NoSBB(ap).dir;
          iap:= NoSBB(ap).bitD;
          next:=1;
     end
end
```

PossivelTransfDEouDDDirAP(ap: SBB, iap: Inclinacao, x: Registro(T)):

```
if !(x.ch < NoSBB(ap).reg.ch) then
     if (x.ch > NoSBB(ap).reg.ch) then
          transDDDEDirAp(fim, ap, iap);
     else fim:= true;
            erro:= true; //Ja esta na arvore;
     end
end
```

transEEEDEsqAp:

Operação que pode fazer uma ação do tipo EE ou ED à esquerda de ap, que é o apontador do nodo corrente da árvore SBB.

```
action transEEEDEsqAp( out fim: Bool, ap: SBB, iap: Inclinacao) is
     if !(fim) then
          EnquantoNaoTerminarTransEEouED;
     end
end transEEEDEsqAp
```

EnquantoNaoTerminarTransEEouDE(fim: Bool, ap: SBB, iap: Inclinacao):

```
if (NoSBB(ap).bitE = horizontal) then
      ChamaTransEEouED(ap, iap);
else fim:= true;
end
```

ChamaTransEEouED(ap: SBB, iap: Inclinacao):

```
if ( NoSBB(NoSBB(ap).esq).bitE = horizontal) then
     transEE(ap);
     iap:= horizontal;
else if (NoSBB(NoSBB(ap).esq).bitD = horizontal)then
```

```
        trasED(ap);
        iap:= horizontal;
       end
    end
```

transDDDEDirAp:

Operação que pode fazer uma operação do tipo DD ou DE à direita de ap, que é o apontador do nodo corrente da Árvore SBB.

```
action transDDDEDirAp(out fim: Bool, ap: SBB, iap: Inclinacao) is
      if !(fim) then
          EnquantoNaoTerminarTransEEouED(fim: Bool, ap: SBB, iap: Inclinacao);
      end
end transDDDEDirAp
```

EnquantoNaoTerminarTransDDouDE(fim: Bool, ap: SBB, iap: Inclinacao):

```
if (NoSBB(ap).bitD = horizontal) then
     ChamaTransDDouDE(ap, iap);
else fim:= true;
end
```

ChamaTransDDouDE(ap: SBB, iap: Inclinacao):

```
if ( NoSBB(NoSBB(ap).dir).bitD = horizontal) then
     transDD(ap);
     iap:= horizontal;
else if (NoSBB(NoSBB(ap).dir).bitE = horizontal)then
     trasDE(ap);
     iap:= horizontal;
     end
end
```

esqCurto:

Segue a operação que trata os quatro casos que podem surgir quando temos uma retirada de nodo, em que o lado esquerdo fica mais curto, em relação ao caminho de apontadores verticais do lado direito.

```
action esqCurto(ap: SBB, out fim: Bool) is
loop:
     step 1: if (NoSBB(ap).bitE = horizontal) then
                    TrataCasos1e2(ap, fim);
                   return;
             end
     step 2: if (NoSBB(ap).bitD = vertical) then
                    TrataCaso3(ap, fim);
                   return;
             end
     step 3: trataCaso4(ap, fim);
end esqCurto
```

TrataCasos1e2(ap: SBB, fim: Bool): Possibilidade tornar um lado mais curto maior, tornando o apontador refenrente a esse lado vertical.

```
NoSBB(ap).bitE:= vertical;
fim:= true;
```

TrataCaso3(ap: SBB, fim: Bool): Possibilidade de tornar o lado maior, mais curto, tornando o apontador referente a esse lado horizontal.

```
NoSBB(ap).bitD:= horizontal;
if (NoSBB(NoSBB(ap).dir).bitE = horizontal) then
     transDE(ap);
else if (NoSBB(NoSBB(ap).dir).bitD = horizontal) then
     transDD(ap);
     end
end
fim:= true;
```

Operação auxiliar que trata do caso 4 de esquerdo curto:

Alterar a posição dos nodos para que os dois lados tenham o mesmo tamanho em termos de número de apontadores verticais.

```
action trataCaso4(ap, out fim)is
     ap1: SBB;
loop:
     step 1: ap1:= NoSBB(ap).dir;
     step 2: NoSBB(ap).dir:= NoSBB(ap1).esq;
     step 3: NoSBB(ap).esq:= ap;
     step 4: ap:= ap1;
     step 5:  TrataTransfDEouDD(ap, fim);
end trataCaso4
```

TrataTransfDEouDD(ap: SBB, fim: Bool):

```
if (NoSBB(NoSBB(NoSBB(ap).esq).dir).bitE = horizontal) then
     transDE(NoSBB(ap).esq);
     NoSBB(ap).bitE:= vertical;
else if (NoSBB(NoSBB(NoSBB(ap).esq).dir)bitD = horizontal) then
          transDD(NoSBB(ap).esq);
          NoSBB(ap).bitE:= vertical;
       end
       fim:= true;
end
```

dirCurto:

Essa operação trata dos quatro casos em que a subárvore do lado direito pode estar mais curta que a do lado esquerdo.

```
action dirCurto(ap: SBB, out fim: Bool) is
loop:
     step 1: if (NoSBB(ap).bitD = horizontal) then
                   TrataCaso1e2(ap, fim);
                   return;
             end
     step 2: if (NoSBB(ap).bitE = horizontal) then
                   trataCaso4(ap, fim);
                   return;
             end
     step 3:  TrataCaso3(ap, fim);
end dirCurto
```

TrataCaso1e2(ap: SBB, fim: Bool):

```
NoSBB(ap).bitD:= vertical;
fim:= true;
```

Função auxiliar que trata do caso em que temos que modificar as posições dos nodos para que possamos manter a Árvore SBB balanceada.

```
action trataCaso4(ap, out fim) is
     ap1: SBB;
loop:
     step 1: ap1:= NoSBB(ap).esq;
     step 2: NoSBB(ap).esq:= NoSBB(ap1).dir;
```

```
          step 3: NoSBB(ap1).dir:= ap;
          step 4: ap:= ap1;
          step 5:  TrataTransEDouEE(ap, fim);
     end trataCaso4
```

TrataTransEDouEE(ap: SBB, fim: Bool):

```
     if (NoSBB(NoSBB(NoSBB(ap).dir).esq).bitD = horizontal) then
          transED(NoSBB(ap).dir);
          NoSBB(ap).bitD:= vertical;
     else if (NoSBB(NoSBB(NoSBB(ap).dir).esq).bitE = horizontal) then
                transEE(NoSBB(ap).dir);
                NoSBB(ap).bitD:= vertical;
            end
            fim:= true;
     end
```

TrataCaso3(ap: SBB, fim: Bool):

```
     NoSBB(ap).bitE:= horizontal;
     if (NoSBB(NoSBB(NoSBB(ap).esq).bitD = horizontal) then
          transED(ap);
          fim:= true;
     else if (NoSBB(NoSBB(NoSBB(ap).esq).bitE = horizontal) then
                transEE(ap);
                fim:= true;
            end
     end
```

pesquisa:

Pesquisa se o nodo cuja chave k dada está presente na Árvore SBB.

```
     public action pesquisaSBB ( in k: Chave, ap: SBB, out achou: Bool, out caminho:
                                                                    Stack(Int) ) is
          r: SBB;
     loop:
          step 1:  Inicializa(r, ap, achou);
          step 2:  AchaNodo(ap, r, achou, caminho);
     end pesquisaSBB
```

Inicializa(r: SBB, ap: SBB, achou: Bool):

```
r:= ap;
achou:= false;
```

AchaNodo(ap: SBB, r: SBB, achou: Bool, caminho: Stack(Int)):

```
if !(r = 0) then
     if (k < r) then
          r:= NoSBB (r).esq;
          pop(caminho, r);
     else if (k > ap) then
          r:= NoSBB (r).dir;
          pop(caminho, r);
     else achou:= true;
          return;
     end
     next: 2;
else if ( achou ) then
     ap:=r;
end
```

retira:

Retira o nodo cuja chave é k da Árvore SBB ap.

```
public action retiraSBB ( in k: Chave, ap: SBB) is
     caminho, camAnt: Stack(Int);
     a, p: SBB;
     fim: Bool;
loop:
     step 1:  LocalizaNodo(k, ap, achou, caminho);
     step 2: if !(achou) then
                 return;
             end
     step 3:  CriaPilhas(caminho, camAnt);
     step 4: push(caminho, r);
     step 5:  CasoFilhoUnico(ap);
     step 6:  AntecessorNodo(ap, camAnt);
     step 7: push(caminho, p);
     step 8: push(caminho, p);
              a:= p;
     step 9:  TransformaçõesEsqOuDirCurto(ap);
              next:=8;
end retiraSBB
```

LocalizaNodo(k: Chave, ap: SBB, achou: Bool):

```
pesquisaSBB(k, ap, achou, caminho);
```

CriaPilhas(caminho, camAnt);

```
 initStack(caminho);
initStack(camAnt);
```

CasoFilhoUnico(ap: SBB):

```
 if (NoSBB(ap).dir = 0 and NoSBB(r).dir = ap ) then
           NoSBB (r).dir:= NoSBB(ap).esq;
else if (NoSBB(ap).dir = 0 and NoSBB(r).esq = ap ) then
           NoSBB (r).esq:= NoSBB(r).esq;
else if (NoSBB(ap).esq = 0  and NoSBB(r).dir = ap ) then
           NoSBB (r).dir:= NoSBB(r).esq;
else if (NoSBB(ap).esq = 0 and NoSBB(r).esq = ap ) then
           NoSBB (r).esq:= NoSBB(r).dir;
 end
 return;
```

AntecessorNodo(ap: SBB, campAnt):

```
antecessor ( k, ap, camAnt ) ;
```

TransformaçõesEsqOuDirCurto(s: SBB, r, SBB):

```
if (NoSBB(a).esq = p) them
     esqCurto(p,  fim);
else
     dirCurto(p,  fim);
end
if (a = 0) then
     return;
end
```

antecessor:

Operação interna que acha o nodo antecessor de um dado nodo cuja chave k foi dada. Utilizado na operação retira.

```
action antecessor ( in k: Chave, ap: SBB, caminho: Stack(Int) ) is
     r: SBB;
     fim, vazia: Bool;
loop:
     step 1:  Inicializa(r, ap, caminho, fim);
     step 2:  AchaAntecessor(r, caminho, ap);
     step 3: empty( caminho, vazia );
     step 4: if ! ( fim and vazia) then
                     dirCurto (ap, fim);
                     next:= 3;
             end
end antecessor
```

```
Inicializa(r:  SBB, ap:  SBB, caminho caminho:  Stack(Int), fim:  Bool):

     r:= NoSBB(ap).esq;
     pop ( caminho, ap);
     fim:= false;
```

```
AchaAntecessor(r:  SBB, caminho:  Stack(Int), ap:  SBB):

      if  !( r = 0 ) then
           r:= NoSBB(r).dir;
           pop(caminho, r);
           next:= 2;
      else NoSBB(ap.reg):= NoSBB(r.reg);
           liberaNodo(r);
      end
```

10.4 Conclusão:

Das Árvores de Pesquisa, a Árvore de Pesquisa SBB é a que possui a manutenção mais complexa, pela suas próprias características que a define. Para manter características da Árvore de Pesquisa SBB, manipula-se a inclinações dos ponteiros após uma inserção ou uma retirada, nos diversos tipos de situações que podem ser geradas após essas ações. Como são muitos os casos que podem surgir em que a subárvore da direita é mais curta ou a subárvore da esquerda mais curta, todos esses casos foram tratados, para manter as características da Árvore SBB.

Por meio dessas características da Árvore SBB, tem-se uma árvore mais balanceada facilitando a busca e recuperação da informação.

Capítulo 11

Tabela Hash

Tabela Hash é uma tabela em que, por meio de uma chave transformada de acordo com uma determinada regra, fornece a localização do elemento referente a essa chave na tabela. A principal intenção do uso desse tipo de tabela é ter uma tabela em que as localizações dos elementos adquiridas por meio da transformação da chave sejam as mais dispersas possível sobre as posições na tabela.

No entanto, sempre haverá o problema de colisões de elementos, em que a localização na Tabela Hash após a transformação da chave, coincidem. O importante, então, é encontrar uma função de transformação que tenha a propriedade de dispersar o máximo as localizações dos elementos inseridos na tabela. Isso, obviamente, depende do tamanho da tabela que será necessária.

11.1 Tipo Abstrato de Tabela Hash e sua Representação:

O tipo abstrato de dados Tabela Hash é uma tabela, em que para solucionar o problema de colisões, já citado, em cada posição da tabela associa-se uma Lista Encadeada. Se a função de transformação de chave for adequada ao tamanho da Tabela Hash teremos uma boa dispersão dos elementos inseridos nas posições da tabela. Mas, como não há como ter uma transformação em que não há colisões, Lista Encadeada foi a solução escolhida implementação. Caso a transformação da chave não seja adequada ao caso, a pesquisa numa Tabela Hash será da ordem de O(n), assemelhando-se a de uma Lista Encadeada. Uma pesquisa em uma Tabela Hash em que a transformada da chave do registro é adequada é da ordem de O(1).

A transformada da chave tem justamente o intuito de tornar as colisões menos freqüêntes e aumentar a eficiência de pesquisa nela. O tipo abstrato de dados Tabela Hash possui a função estática max que informa o tamanho da Tabela Hash.

```
static max is Int:= 30;
```

A função externa ord é uma função que dado um caracter devolve o inteiro correspondente em código ASCII.

```
external ord is Char -> Int;
```

Os tipos internos ao módulo de criação da Tabela Hash são os tipos Index) e (Indice ambos do tipo inteiro, mas que representam intervalos possíveis de inteiros diferentes.

```
type Index is Int (0..max-1);
```

```
type Indice is Int (1.. max);
```

O tipo Chave é uma função que mapeia um inteiro para seu correspondente caracter, ou seja, porta-se como o tipo String da Linguagem C , um vetor de caracteres.

```
public type Chave is Int -> Char (Indice);
```

O tipo Reg é uma tupla que guarda em um de seus campos a chave de pesquisa de um elemento, o qual representa.

```
public type Reg is tuple ( ch: Chave, // outros componente );
```

O tipo Dicionario é a representação propriamente dita de uma lista de listas encadeadas, ou seja, o modo de implementação utilizado nesse modo de implementação da tabela Hash.

```
public type Dicionario is Int -> Lista(T) (Index);
```

Essa estrutura é ilustrada abaixo:

11.2 Interface das Operações em Tabela Hash:

- h(in ch: Chave, ind: Int): transforma a chave ch em uma posição da tabela Hash e a retorna em $\hat{ind}$.
- inicialize(t: Dicionario): cria e torna disponível a tabela hash em t.
- pesquise(in ch: Chave,t: Dicionario, x: Int, out achou: Bool): pesquisa na tabela hash t dada a chave de pesquisa ch e devolve o inteiro que representa o registro em x e retorna em achou se a informação em x é válida ou não.
- insira(in x: Reg, t: Dicionario, out erro: Bool): insere na tabela hash t o registro x e retorna se houve algum erro durante a inserção.
- retire(in ch: Chave, t: Dicionario, out x: Int, out erro: Bool): dada uma chave de pesquisa ch na tabela hash t, retira o registro correspondente a chave, devolvendo em x o inteiro que representa o registro desta chave ch e devolve em achou a verificação se o registro foi ou não encontrado.

11.3 Implementação do tipo abstrato de dado Tabela Hash:

Na implementação da Tabela Hash foi implementado um módulo de ações auxiliares de lista especialmente para atender o caso de Tabela Hash. A interface dessas

operações serão mostradas logo em seguir, incluindo no fim do capítulo as suas implementações.

- fLvazia(z: Lista(T)): cria uma lista vazia e a torna disponível em z.
- pesquiseLista(in ch: Chave, z: Lista(T), out achou: Bool): pesquisa na lista z a existência da chave ch e devolve o resultado da pesquisa em achou.
- pegueCorrente(in z: Lista(T), out x: Int): dada a lista z, retorna o elemento corrente em x.
- houveErroLista(in z: Lista(T), out erro: Bool): verifica se houve erro na última operação efetuada na lista z.
- insiraInicio(in x: Reg, z: Lista(Int)): insere a informação dada em x na lista z.
- retireCorrente(z: Lista(T)): retira da lista z, o elemento corrente.

11.3.1 Operações:

h:

```
action h( ch: Chave, ind: Int) is
     i, soma: Int;
loop:
     step 1: soma:= 0;
     step 2: SomaCodigosNumericoChave(ch);
     step 3: DevolveInd(ind, soma);
end h
```

SomaCodigosNumericoChave(ch: Chave):

```
if (i = 1 | i <= n) then
     soma:= soma + ord (ch(i));
     next:= 2
end
```

DevolveInd(ind: Int, soma: Int):

```
ind:= soma mod max;
```

inicialize:

```
public action inicialize(t: Dicionario) is
     i: Int;
     f: Lista(T);
loop:
     step 1:  FazCadaPosicaoHashLista(f, t, i);
     step 2:  CriaListasEncParaCadaPosicaoHash(f, i);
end inicialize
```

```
IDentro0..Max-1:= i > 0 or i=0 or i < max FazCadaPosicaoHashLista(f:  LIsta(T),
t:  Dicionario, i:  Int):

     if  IDentro0..Max-1 then
          f:= t(i);
     end
```

```
CriaListasEncParaCadaPosicaoHash(f:  Lista(T), i:  Int):

     if (i = 0 & i <= (max - 1)) then
          fLvazia(f);
          i:= i+ 1;
          next:= 1;
     end
```

pesquise:

```
public action pesquise( in ch: Chave, t: Dicionario, out x: Reg, out achou: Bool) is
     k: Int;
loop:
     step 1:  TransformaChaveEmPosicao(ch, k);
     step 2: pesquiseLista(ch, t(k), achou);
     step 3: if (achou) then
                    pegueCorrente(t(k), x);
             end
end pesquise
```

```
TransformaChaveEmPosicao(ch:  Chave, k:  Int):

     h(ch, k);
```

insira:

```
public action insira(in x: Reg, t: Dicionario, out erro: Bool) is
     achou, erroLista: Bool;
     k: Int;
loop:
     step 1:  TransformaChaveEmPosicao(x.ch, k);
     step 2: pesquiseLista(x.ch, t(k), achou);
     step 3: houveErroLista(t(k), erroLista);
     step 4: if !(achou) then
                  insiraInicio(x,t(k));
             end
             erro:= achou | erroLista;
end insira
```

TransformaChaveEmPosicao(ch: Chave, k: Int):

```
h(x.ch, k);
```

retire:

```
public action retire( in ch: Chave, t: Dicionario, out x: Reg, out erro: Bool) is
     achou: Bool;
loop:
     k: Int;
     step 1: h( ch, k);
     step 2: pesquiseLista( ch, t(k), achou);
     step 3: if (achou) then
                  retireCorrente(t(k), x);
             end
             erro:= ! achou;
end retire
```

TransformaChaveEmPosicao(ch: Chave, k: Int):

```
h( ch, k);
```

11.4 Implementação das Operações de Listas Especiais para Tabela Hash:

As ações aqui utilizadas estão disponíveis na implementação de Lista Encadeada no capítulo 4.

11.4.1 Operações:

fLvazia:

```
public action fLvazia(z: Lista(T)) is
    inicializaLista(z);
end fLvazia
```

pesquiseLista:

```
public action pesquiseLista(ch: Chave, z: Lista(T), achou: Bool) is
     search(ch, z, achou);
end pesquiseLista
```

pegueCorrente:

```
public action pegueCorrente(z: Lista(T), x: Int) is
     go(z, x);
end pegueCorrente
```

houveErroLista:

```
public action houveErroLista(z: Lista(T), erro: Bool) is
     houveErro(z.status, erro);
end houveErroLista
```

insiraInicio:

```
public action insiraInicio(x: Int, z: Lista(Int)) is
     step 1: start(z);
     step 2: insertLeft(z, x);
end insiraInicio
```

retireCorrente:

```
public action retireCorrente(z: Lista(T)) is
     delete(z);
end retireCorrente
```

11.5 Conclusão:

A Tabela Hash possui muitas vantagens se for utilizada uma função que se idequalize com o tamanho da tabela, caso contrário a pesquisa em uma tabela pode se tornar uma pesquisa em uma Lista Encadeada, o que seria algo indesejado e perderia completamente o sentido da Tabela Hash.

A intensão da Tabela Hash é de dispersar ao máximo as entradas pela tabela de tal forma que a pesquisa fique próxima da ordem de um (O(1)). Sendo assim, a maior dificuldade da implementação de uma Tabela Hash é justamente encontrar uma função ideal para cada tamanho de tabela necessária para que essa dispersão seja máxima. No entanto, achar essa função pode ser uma tarefa muito complicada e dependeria de uma uma longa dissertação para descrever modos de achá-la.

Em síntese, Tabela Hash é uma Lista Linear em que cada entrada funciona como um armazenamento para uma Lista Simplesmente Encadeada. Por isso mesmo que é tão importante a dispersão das entradas na Tabela Hash.

Capítulo 12

Conclusão:

O trabalho de pesquisa desenvolvido tinha como principal objetivo a programação de Tipos Abstratos de Dados em Machĩna tanto para testar a linguagem e seu compilador, ainda em desenvolvimento, quanto para o crescimento acadêmico da bolsista na área de linguagem de computadores. Nessa monografia vê-se o reultado dos esforços, tornando-a possível. A monografia atingiu os objetivos proposto pelo projeto, mas como há muitos tipos abstratos de dados, escolheu-se os mais freqüentemente usados e de mais simples aplicação. Listas, filas, pilhas, árvores binárias de pesquisa, árvore Patricia, árvore SBB e tabela Hash foram os TADs que foram implementados e amplamente discutidos na monografia.

O trabalho não se restringe apenas a programação, mas na didática de apresentação dos resultados, utilizando a metodologia científica. A boa apresentação dos resultados da maneira mais inteligível possivel para fins acadêmicos.

A principal dificuldade imposta foi a mudança de estilo de programação e tipo de linguagem, a qual é baseada em máquinas de estados abstratos.