UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
DISCIPLINA: Estrutura De Dados - INE5609
PROFESSOR: José Eduardo De Lucca

PROJETO FINAL: UMA IMPLEMENTAÇÃO COM LISTA INVERTIDA PARA INDEXÇÃO DE UMA TABELA DE DADOS

André Camargo
Lucas Pereira

Florianópolis
Julho de 2011

SUMÁRIO

1 INTRODUÇÃO

Como projeto final recebemos a tarefa de utilizar uma **listar invertida** ou uma **multilista** para realizar a indexação de um conjunto de dados a partir de uma tabela de dados. Foi necessário também, escolher um tipo de conjunto de dados para se trabalhar. Um outro objetivo do trabalho é que fosse possível realizar operações a partir da tabela de dados. Operações como: inserir, remover, buscar, exibir e carregar dados.

Assim, decidimos utilizar a lista invertida como estrutura para realizar a indexação dos dados que serão compostos por dados de pessoas. Os motivos do porque da escolha em usar lista invertida explicaremos logo mais a frente. Já nosso tipo de dado escolhido foi algo bem simples e de certa forma bem genérico, mas é interessante destacar que qualquer outro tipo de dado poderia ser utilizado sem maiores problemas e/ou modificações no projeto.

Temos então um tipo de dado que denominamos **Pessoa** e esta tem os seguintes campos de dados: CPF, Nome, Idade, Salário, Escolaridade e Cidade. Com esses dados poderemos criar nossa tabela e utilizar três tipos diferentes de diretórios de lista invertida: um **diretório discreto** para indexação de dados como idade e escolaridade, um **diretório continuo** para indexação de dados como salário e por fim um diretório que denominamos de **flexível** para indexação de dados como cidade.

Para a realização deste projeto decidimos usar como linguagem de programação a linguagem Java. Essa decisão foi devido ao fato do Java ser orientado a objetos e de já possuirmos uma certa intimidade com a linguagem. Nesse último aspecto pensamos em uma linguagem já conhecida por nós, pois assim, não perdemos tempo com as especifidades da linguagem e podemos nos concentrar apenas nos aspectos das estruturas de dados e da lista invertida.

Teremos neste documento uma seção que falaremos das decisões de projeto, onde será explicado o porque da escolha do uso de cada estrutura e teremos uma outra seção que falaremos da estruturação do projeto, apresentando suas classes e o uso das mesmas.

2 DECISÕES DE PROJETO

Falaremos aqui das principais decisões de projetos tomadas e explicaremos os motivos delas. Nesta parte nos focaremos a falar das decisões relacionadas as estruturas de dados utilizadas, bem como da utilização da **lista invertida** e outros aspectos relacionados a isso.

2.1 Lista Invertida

Nossa escolha se deu entre utilizar lista invertida ou multilista para implementação da indexação dos dados para busca. Escolhemos lista invertida por ser mais flexível devido ao fato de não precisar estar diretamente acoplada com a tabela de dados, mas apenas apontar para linhas da tabela. Utilizando lista invertida também ganhamos um outro conceito para se trabalhar que são os diretórios. Acreditamos ser mais fácil e organizado se trabalhar com diretórios ao invés de multilista.

Em suma escolhemos lista invertida por três aspectos: ser mais flexível, ser mais fácil de se trabalhar e o principal para nós que é o desacoplamento da tabela de dados, podendo trabalhar apenas nos diretórios e somente ir à tabela de dados quando precisarmos inserir, pegar ou remover um dado.

2.2 Tabela De Dados

Para a tabela de dados utilizamos uma **lista encadeada dinamicamente**. Nossa escolha se deu na busca pela simplicidade. Assim, escolhemos uma lista **simplesmente** encadeada e que tem **inserção no inicio**.

Escolhemos o fato de ser simplesmente encadeada porque economizamos espaço, já que para cada elemento precisamos saber apenas o próximo. Escolhemos essa opção pensando no fato que devido a existência dos diretórios para indexação dos dados não precisaríamos percorrer com muita frequência a lista. Também nos baseamos no fato de que se no decorrer do projeto começássemos a precisar andar muito pela lista poderíamos implementar um encadeamento duplo, mas isto não foi necessário.

E escolhemos a característica de possuir inserção no inicio por ser a forma mais rápida nesse tipo de lista encadeada. Precisamos ter apenas o primeiro elemento da lista e assim a inserção é feita em O(1).

Com esse esquema o ponto fraco da nossa lista é a remoção de dados já que a inserção é rápida e a busca é feita nos diretórios. Com isso, tentamos pensar em duas soluções:

- A primeira foi utilizar uma **árvore binária** para indexar as chaves primárias e assim poder remover o elemento. Chegamos a implementar parcialmente esta ideia, porém percebemos no meio do caminho que seria mais custoso manter a árvore do que realizar a remoção diretamente na lista. Seria mais custoso pois a cada inserção na lista era necessário uma inserção na árvore com o ônus de ter que verificar o balanceamento da árvore com uma certa periodicidade.
- A segunda solução foi utilizar uma **lista ordenada**. Porém, logo percebemos que como o número de inserções muito provavelmente superaria consideravelmente o número de remoções, então isto não valeria a pena já que a cada inserção é necessário ordenar o elemento inserido, fator este seria muito mais custoso do que a remoção na nossa lista.

Assim, percebemos que o melhor a fazer seria maximizar a eficiência do caso mais comum em deterioração do caso menos comum. Ou seja demos preferência a uma inserção e buscas rápidas aceitando uma remoção não tão rápida assim.

Também pensamos no uso de outras estruturas ao invés da lista encadeada como o **Hash**. A dificuldade seria em garantir um fator de carga baixo. E devido a esse fato precisaríamos ter uma preocupação adicional que seria analisar

de tempos em tempos o fator de carga e ter que aumentar a tabela de espalhamento, se necessário. Além disso, partimos do pressuposto que não se sabe inicialmente a quantidade média de dados que terá nossa tabela, então ficaria complicado de se definir o tamanho da tabela de espalhamento. Sendo assim mantemos com a nossa lista simplesmente encadeada.

2.3 Diretórios

Criamos três opções de diretórios diferentes para se adaptar a cada caso. Precisamos de um **diretório discreto** para indexar dados discretos ou enumeráveis. Precisamos do **diretório continuo** para indexar dados que possuem grupos de valores e cada valor do diretório deve ser posto dentro de um grupo. E por fim o **diretório flexível** que criamos para dados onde o tamanho do diretório pode variar bastante e não se têm uma noção exata de todos os valores que o diretório conterá, como é o caso da cidade.

Nos diretórios possuímos agrupamento de dados que combinam com um campo em especifico. A esse agrupamento escolhemos o nome de **partições**. Por exemplo, no diretório discreto de idade temos um agrupamento de todos os elementos com idade 50, então chamados esse agrupamento de partição, bem como para todos os outros agrupamentos de idade.

2.3.1 Diretório Discreto

Neste tipo de diretório partimos do pressuposto que teremos uma sequência continua de dados discretos. Assim, precisamos apenas do valor minimo e do valor máximo que se pode assumir e criamos uma partição para cada um desses valores dentro do intervalo de mínimo e máximo. Por exemplo, para idade

criamos um diretório discreto que possui como valor minimo 0 (a pessoa pode ser um recém-nascido) e um valor máximo de 120. Teremos então um diretório com 121 partições diferentes.

Para valores enumeráveis como é o caso da escolaridade utilizamos um enum e como valor mínimo e máximo utilizamos o número ordinal da primeira e da última enumeração respectivamente. Por exemplo, na nossa escolaridade declaramos o enum como: SEM_ESCOLARIDADE, FUNDAMENTAL, MEDIO, SUPERIOR_INCOMPLETO e SUPERIOR. Assim os valores discretos vão de 0 (SEM_ESCOLARIEDADE) a 4 (SUPERIOR) tendo um total de 5 partições.

Para este tipo de diretório discreto criamos em nossa implementação uma **lista array** que guarda **listas encadeadas** de pessoas. A lista array seria o diretório em si e cada lista encadeada de pessoa seria uma partição.

A decisão de se usar uma lista array para guardar as partições foi porque no caso do diretório discreto o mesmo não crescerá “verticalmente”, já que as partições são bem definidas e como nesse caso não precisamos de flexibilidade de crescimento, então podemos usar uma estrutura mais simples, tanto na implementação quanto no acesso. O interessante aqui é que no caso do diretório discreto, como são dados inteiros então o próprio campo é a chave para selecionar a partição desejada. Por exemplo, se quisermos pegar todas as pessoas com idade 50, então basta usar como chave para selecionar a partição o próprio 50. Na nossa implementação isso funcionará mesmo que o valor minimo for diferente de 0, o que seria o caso se a idade minima fosse 1 ao invés de 0.

A escolha em se usar lista encadeada dinamicamente para as partições de pessoas é devido ao fato de que nosso diretório crescerá “horizontalmente”, uma vez que o número de inserções tente a aumentar e assim cada partição também. Dessa forma, como a partição precisa ser flexível quanto ao número de elementos decidimos usar a lista encadeada dinamicamente.

2.3.2 Diretório Continuo

O diretório continuo foi criado para indexar valores não inteiros e não

enumeráveis e que possuem faixas de intervalos onde um dado é inserido em uma das faixas. Este é o caso do salário. Não poderíamos criar um diretório para cada tipo de valor de salário pois não seria eficiente. O que fizemos foi criar intervalos e agrupar os valores por intervalo. Para o salário estipulamos os intervalos de 0 à R$ 1000, R$ 1000 à R$ 2000, R$ 2000 a R$ 3000, …, R$ 9000 a R$ 10000 e mais de R$ 10000, fechando com um total de 11 partições.

Como no caso do diretório discreto nosso diretório continuo não crescerá “verticalmente” já que as faixas de valores são bem estipuladas inicialmente, então podemos usar uma lista array. Para o caso das partições contendo os dados usamos uma lista encadeada dinamicamente pelo mesmo motivo do diretório discreto.

Uma diferença para com os diretórios discretos é que não podemos usar o campo diretamente como chave para acessar a partição. Para este proposito temos uma lista encadeada ordenada auxiliar que serve para mapear cada uma das partições.

2.3.3 Diretório Flexível

O diretório flexível foi criado para indexar campos que não podem ser previamente definidos ou enumeráveis, como é o caso das cidades. Como no Brasil existem mais de 5500 municípios, então seria ineficiente se criássemos uma partição para cada município já no começo. O que acontece no diretório flexível é que somente é criado uma partição para um determinado tipo de campo quando o mesmo é inserido no diretório pela primeira vez.

Por exemplo, consideremos que criamos um diretório para cidades e inserimos um dado que contém o campo Florianópolis. Nesse momento a partição Florianópolis será criada e o dado inserido nela. Mais tarde se inserirmos outro dado que tem como campo Florianópolis, então esse dado será colocado na partição já criada anteriormente. Desta forma não se desperdiçará espaço com partições que nunca contiveram nenhum elemento.

Devido a isso no diretório flexível usamos uma lista encadeada dinamicamente para armazenar as partições que contêm os dados. Isso porque ao

contrário dos diretórios discreto e continuo, o diretório flexível não tem suas partições estipuladas no momento de sua criação e desta forma, o diretório precisará crescer “verticalmente”. Para cada partição contendo as pessoas usamos também uma lista encadeada dinamicamente pelo mesmo motivo dos outros dois diretórios: o tamanho de cada partição precisa ser flexível.

Assim como no diretório contínuo, o diretório flexível não é diretamente mapeável para uma posição na lista encadeada de partições. Por isso, também usamos uma lista auxiliar contendo as chaves onde a partição é buscada.

3 ESTRUTURAÇÃO DO PROJETO

Apresentaremos aqui as classes criadas no projeto e buscaremos discutir a implementação de algumas delas. O projeto está dividido em três pacotes principais sendo eles **modelo**, **visão** e **controle**. Nosso foco principal aqui está no modelo, por isso vamos aprofundá-lo mais.

3.1 Modelo

Temos no modelo os pacotes: **diretório**, **estruturaDeDados**, **conjuntoDeDados** e **excecao**. Ainda dentro do modelo temos as classes **ListaDePessoas** e **GerenciadorDeDados**. Vamos apresentar primeiramente as classes dentro de modelo e depois partiremos para as classes dentro dos subpacotes do modelo.

- **ListaDePessoas**: Esta classe nada mais é que nossa tabela de dados e foi implementada com uma lista encadeada dinamicamente. Na verdade ela apenas contém uma ListaEncadeada e provê métodos de acesso à mesma. Decidimos utilizar na ListaDePessoas o padrão *singleton* já que ela é usada em várias partes do programa e com o uso deste padrão temos mais “segurança” de que em todas as partes está se trabalhando com a mesma lista.
- **GerenciadorDeDados**: É nessa classe que estão os diretórios e é nela que as operações sobre os dados são realizadas. Apesar desta classe ter um aspecto centralizador foi interessante fazer desta forma para isolar os diretórios e os dados do resto do programa. Devido a isso, se uma operação sobre o conjunto de dados é necessária, essa operação será feita utilizando a comunicação com o GerenciadorDeDados. Entre as operações

estão: carregar, inserir, remover e buscar dados. Sendo que a busca pode ser tanto simples quanto composta.

3.1.1 Pacote diretorio

Como explicado na seção decisões de projetos, decidimos criar três tipos de diretórios. Neste pacote temos as seguintes classes:

- **TipoDiretorio**: Interface que especifica como deve ser um diretório. Desta forma os três tipos de diretórios utilizados devem seguir esta especificação.
- **DiretorioDiscreto**: Utilizado para indexar valores inteiros ou enumeráveis.
- **DiretorioContinuo**: Diretório que indexa dados que se encaixam em grupos específicos. Nesse caso, cada valor diferente não possui seu próprio diretório, mas compartilha um mesmo diretório com valores semelhantes.
- **DiretorioFlexivel**: Assim como no DiretorioDiscreto, o DiretorioFlexivel também possui um grupo para cada valor especifico. Porém, a diferença é que no DiretorioFlexivel os valores não são previamente e cada valor tem seu grupo criado quando o valor em questão é inserido pela primeira vez.

3.1.2 Pacote estruturaDeDados

Este pacote contém as estruturas de dados utilizadas para a realização do projeto, bem como classes auxiliares à essas estruturas.

Porém, antes de mostrar as classes deste pacote é interessante explicar

como é feita a busca (simples ou composta) no nosso projeto. Primeiramente na nossa busca vemos quais campos deseja-se buscar e obtemos nos respectivos diretórios o grupo que combina com o campo fornecido. Por exemplo, se desejarmos obter todas pessoas com idade 50 e que sejam de Florianópolis, então pega-se no diretório idade o grupo com as pessoas de idade 50 e pega-se no diretório cidade o grupo das pessoas de Florianópolis. Após isso, guardamos na lista ordenada todos esses grupos buscados e que estão na forma de lista encadeada. Nesse caso, o critério de ordenação é o tamanho da lista que contém os dados. Por exemplo, se existir mais pessoas de Florianópolis do que pessoas com idade 50, então o primeiro elemento da lista ordenada será a lista das pessoas com idade 50.

A utilização da lista ordenada que contêm as listas com os campos buscados é feita, pois, ao buscar primeiramente os elementos das listas menores precisa-se fazer menos comparações com as próximas listas para verificar se a pessoa em questão possui todas as condições da busca.

Explicamos como funciona nossa busca, apenas para que se justifique o motivo de se ter uma ListaEncadeadaOrdenada e de fazer a ListaEncadeada implementar a interface Comparavel. Vamos agora às classes deste pacote:

- **Comparavel**: Interface que especifica um objeto que pode ser comparado. Entre os métodos que devem ser implementados estão: éMaior(), éMenor() e éIgual().
- **ListaEncadeada**: Lista que utiliza encadeamento simples e dinâmico. Esta lista implementa a Interface Comparavel e a condição de comparação é quantidade de elementos em cada lista. A lista encadeada possui também como classe interna o IteradorDeLista que implementa o TipoIterador.
- **ListaEncadeadaOrdenada**: É uma extensão da ListaEncadeada, com a diferença que possui o método inserirOrdenado().
- **ListaArray**: Lista que não é encadeada e utiliza-se de índice para acesso aos elementos. Essa lista é usada pelos diretórios discreto e continuo.
- **TipoIterador**: Interface que específica uma forma de caminhar pelos elementos de uma determinada estrutura de dados.
- **CaixaSimples**: Utilizada pelas listas encadeadas para guardar o

elemento e o próximo elemento.

3.1.3 Pacote excecao

Contém as exceções que podem ocorrer no projeto.

- **ExecaoDeFaltaDeElementos**: Exceção lançada quando durante o uso de um TipoIterador se usa o método fornecerProximo(), sendo que não possuem mais elementos.
- **ExecaoDeRemocaoIlegal**: É lançada quando durante o uso de um TipoIterador se aplica o método remover() duas vezes na mesma iteração ou quando não há mais elementos na estrutura de dados.

3.1.4 Pacote conjuntoDeDados

Aqui estão as classes que representam os dados que escolhemos para a realização do trabalho que consistem em dados de pessoas.

- **Pessoa**: Objeto que guarda todos os campos de uma determinada Pessoa.
- **Escolaridade**: Enumeração representando todas as possíveis escolaridades para serem colocadas em uma Pessoa.

3.2 Visão

Contém a interface gráfica do projeto, possuindo o pacote **atualizacao** e

outras duas classes apresentadas abaixo:

- **FrameInicial**: Frame contendo toda a interface gráfica do projeto onde são mostrados os dados.
- **FrameInserirDado**: É um frame á parte utilizado para o preenchimento de um novo dado inserido.

3.2.1 Pacote atualizacao

Pacote que contém classes auxiliares para a interface gráfica sendo elas:

- **AtualizadorDeNomes**: Tem a função de atualizar as pessoas mostradas na lista de remoção.
- **AtualizadorDeTabela**: Atualiza a tabela que apresenta os dados cada vez que alguma operação sobre a tabela de dados é realizada.

3.3 Controle

Comunica-se com o modelo e com a visão para responder as ações solicitadas. Ou seja quando uma operação é solicitada o controle avisa ao modelo e responde a interface gráfica com o resultado.

- **ControleDeBuscas**: Este controlador é responsável por dizer ao GerenciadorDeDados quais campos estão sendo buscados e espera pela resposta da busca para apresentá-la na interface gráfica.
- **ControleDeCarregamento**: Quando o botão carregar dados é pressionado, um conjunto de dados contidos nessa classe é carregado na ListaDePessoas e após isso os dados carregados na

lista são inseridos nos diretórios através do GerenciadorDeDados.

- **ControleDeInsercao**: Requisita ao Gerenciador de dados a inserção de um determinado dado e atualiza a interface gráfica.
- **ControleDeRemocao**: Requisita ao Gerenciador de dados a eliminação de um determinado dado e atualiza a interface gráfica.
- **ControleDoAtualizadorDeTabela**: Controlador utilizado pelo AtualizadorDeTabela.
- **ControleDoFrameInserirDado**: Quando o botão inserir dado é clicado, o controlador chama o frame de inserção de dado que é mostrado na tela.