



MÉTODOS DE PESQUISA




Pesquisa

- Os arquivos e/ou tabelas são tipos abstratos de dados com um conjunto de operações associadas. As mais comuns são:
 1. Inicializar a estrutura de dados;
 2. Pesquisa um ou mais registros com determinada chave;
 3. Inserir um novo registro;
 4. Remover um determinado registro
 5. Ordenar um arquivo
 6. Unir dois arquivos, formando um maior.



Pesquisa

- **Como recuperar uma informação a partir de várias outras armazenadas em uma tabela ou arquivo?**
 - A informação é dividida em registros
 - Cada um possui uma chave que será usada na pesquisa.
 - A pesquisa pode ser com sucesso. Caso contrário, a pesquisa é sem sucesso.



Medida Relevante de Complexidade

- Na *PESQUISA*, vamos considerar apenas o número de comparações entre chaves.
- Na *INSERÇÃO* e *REMOÇÃO*, o número de movimentações de itens também pode ser importante.
- Neste capítulo, estudaremos a complexidade apenas dos algoritmos de *PESQUISA*.



Pesquisa

- **Existem vários métodos de pesquisa. A escolha depende:**
 - Da quantidade de dados envolvidos
 - Da volatilidade do arquivo (inserções e retiradas freqüentes).
- **Se o arquivo é praticamente estável, o importante é minimizar o tempo de pesquisa, sem se preocupar com o tempo gasto para estruturar o arquivo.**



Pesquisa

- O termo **tabela** identifica um arquivo na memória principal,
- O termo **arquivo** é usado para arquivos na memória secundária.
- Os algoritmos apresentados trabalharão sobre **chave primária**, ficando a cargo do aluno pensar em soluções para **chave secundária**.
- **Item de busca** é definido como um campo (ou atributo) da tabela que pode ser usado para pesquisa, mas que não, necessariamente, é uma chave primária

Classe Item

```
public class Item {
    private tipoChave chave;
    // outros atributos
    // construtor e métodos
    public tipoChave getChave ( ){
        return this.chave;
    }
}
```

Classe ListaEnc

```
public class ListaEnc {
    private No prim;
    //Pode ter outros atributos,
    // como ult, nElem.
    public ListaEnc(){
        this.prim = null;
    }
}
```

Classe Tabela

```
public class Tabela{
    private Item[] vetor;
    private int nElem;
    public Tabela (int tamanho){
        this.vetor = new Item[tamanho];
        this.nElem = 0;
    } // métodos da classe
}
```

PESQUISA SEQUENCIAL EM TABELA DESORDENADA

- É a técnica mais simples de realizar uma busca.
- Para fazer este tipo de pesquisa o conjunto de dados não precisa estar ordenado.
- Pode-se usar vetor ou lista encadeada.

Classe No

```
public class No {
    private Item info;
    private No prox;
    public No(Item _info){
        this.info = _info;
    }
    public No getProx () {
        return this.prox;
    }
    public void setProx(No novo) {
        this.prox = novo;
    }
}
```

Pesquisa Sequencial em Vetor

```
public int pesquisaSequencial (int chave){
    int pos = this.nElem-1;
    while ((pos >= 0) &&
        (this.vetor[pos].getChave() != chave))
        pos--;
    }
    return pos;
}
```

Custo

- Melhor caso:

$$C(n) = 1$$

- Pior caso

$$C(n) = n$$

Considerações

- É a melhor solução para problemas com até 25 registros
- A Função Pesquisa retorna o índice do registro que contém a chave procurada. Se retornar -1 é porque não encontrou o item.
- Não suporta mais de um registro com a mesma chave. Para passar a aceitar é necessária uma reformulação.

Inserir no Vetor

```
public boolean inserir(Item elemento){
    if (this.nElem == this.vetor.length)
        return false;
    else{
        this.vetor[this.nElem]= elemento;
        this.nElem++;
        return true;
    }
}
```

Pesquisa Seqüencial em LSE

```
public No pesqSeq (int chave){
    No atual = this.prim;
    while ((atual != null) &&
        (atual.getInfo().getChave() != chave))
        atual = atual.getProx();
    return atual;
}
```

Remove do Vetor

```
public boolean remove (int chave){
    int pos;
    if (this.nElem == 0)
        return false;
    else{
        pos = pesquisaSequencial (chave);
        if (pos >= 0){
            this.nElem--;
            this.vetor[pos] = this.vetor[this.nElem];
            this.vetor[this.nElem] = null;
            return true;
        }
        else
            return false;
    }
}
```

Custo

- Melhor caso:

$$C(n) = 1$$

- Pior caso

$$C(n) = n$$

Insere na Lista

```
public void inserePrimeiro(Item elem){
    No novoNo = new No (elem);

    novoNo.setProx(this.prim);
    this.prim = novoNo;
}
```

Pesquisa Binária

meio = (0 + 4) div 2 = 2
 3 = V[meio] ? NÃO
 3 < V[meio] ? SIM ⇒ Dir = meio - 1
 Esq

3	5	6	8	9	12	18	20	24	29	35
0	1	2	3	4	5	6	7	8	9	10

Meio = (0 + 1) div 2 = 0
 3 = V[meio] ? SIM ⇒ RETORNA 0

Remove da Lista

```
public boolean remove (int chave){
    No atual = this.prim;
    No ant = this.prim;
    if (atual == null)
        return false;
    while (atual.getInfo().getChave() != chave){
        if (atual.getProx() == null)
            return false;
        else{
            ant = atual;
            atual = atual.getProx();
        }
    }
    if (atual == this.prim)
        this.prim = this.prim.getProx();
    else
        ant.setProx(atual.getProx());
    return true;
}
```

Classe TabelaOrd

```
public class TabelaOrd{
    private Item[] vetor;
    private int nElem;
    public TabelaOrd (int tamanho){
        this.vetor = new Item[tamanho];
        this.nElem = 0;
    } // métodos da classe
}
```

Pesquisa Binária

Exemplo: Pesquisando o número 3

Esq										Dir
3	5	6	8	9	12	18	20	24	29	35
0	1	2	3	4	5	6	7	8	9	10

meio = (0 + 10) div 2 = 5
 3 = V[meio] ? NÃO
 3 < V[meio] ? SIM ⇒ Dir = meio - 1
 Esq

3	5	6	8	9	12	18	20	24	29	35
0	1	2	3	4	5	6	7	8	9	10

Pesquisa Binária

```
public int pesqBinaria (int chave){
    int meio, esq, dir;
    esq = 0;
    dir = this.nElem-1;
    while (esq <= dir){
        meio = (esq + dir)/2;
        if (chave == this.vetor[meio].getChave())
            return meio;
        else{
            if (chave < this.vetor[meio].getChave())
                dir = meio - 1;
            else
                esq = meio + 1;
        }
    }
    return -1;
}
```

Custo

■ Melhor caso:

$$C(n) = 1$$

■ Pior caso

$$C(n) = 2 \log_2 n$$

■ Caso Médio

$$O(\log_2 n)$$

Considerações

✍ O custo para manter a tabela ordenada é alto, pois a cada inserção de forma ordenada ou a cada exclusão, há um número alto movimentações.

✍ Logo, **não é** um método indicado para arquivos voláteis.

Inserir no Vetor Ordenado

```
public boolean insere (Item elem){
    if (this.nElem == this.vetor.length)
        return false;
    else{
        this.vetor[this.nElem] = elem;
        this.nElem++;
        inserçãoDireta();//ordenação
        return true;
    }
}
```

Árvore de Busca

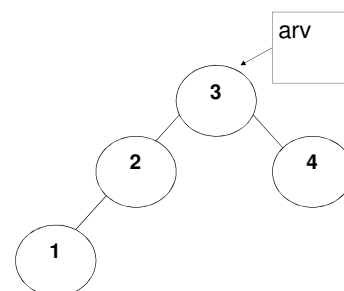
■ São utilizadas quando há necessidade dos seguintes requisitos:

- ☐ Acessos direto e seqüencial eficientes
- ☐ Facilidade de inserção e remoção de elementos
- ☐ Alta taxa de utilização de memória
- ☐ Utilização de memória primária e secundária

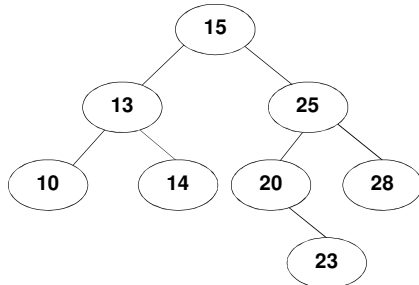
Remove Ordenado do Vetor

```
public boolean remove (int chave){
    int i, pos;
    if (this.nElem == 0)
        return false;
    else{
        pos = pesqBinaria(chave);
        if (pos >= 0){
            for (i = pos + 1; i < this.nElem; i++)
                this.vetor[i-1] = this.vetor[i];
            this.nElem--;
            return true;
        }
        else
            return false;
    }
}
```

Árvore Binária de Busca



Caminhamento Central



Árvore Binária de Busca sem Balanceamento – ABB – Pesquisa

```
public NoArvore pesquisa (int chave){
    return this.pesquisa (chave, this.raiz);
}
```

Estrutura de uma Árvore de Busca

```
public class NoArvore {
    private Item info;
    private NoArvore dir, esq;
    public NoArvore (Item _info){ this.info = _info; }
    public NoArvore getDir() { return dir; }
    public void setDir(NoArvore dir) { this.dir = dir; }
    public NoArvore getEsq() { return esq; }
    public void setEsq(NoArvore esq) { this.esq = esq; }
    public Item getInfo() { return info; }
    public void setInfo(Item novo) { this.info = novo; }
}
```

Árvore Binária de Busca sem Balanceamento – ABB – Pesquisa

```
private NoArvore pesquisa (int chave, NoArvore no){
    NoArvore temp;
    temp = no;
    if (temp != null){
        if (chave < temp.getInfo().getChave()){
            temp = this.pesquisa (chave, temp.getEsq());
        }
        else{
            if (chave > temp.getInfo().getChave()){
                temp = this.pesquisa (chave, temp.getDir());
            }
        }
    }
    return temp;
}
```

Estrutura de uma Árvore de Busca

```
public class Arvore {
    private NoArvore raiz;
    public Arvore(){
        this.raiz = null;
    } //métodos (inclusive os de inserção,
    //remoção e pesquisa)
}
```

Custo

Melhor caso: Chave de busca na raiz

$$C(n) = 1$$

■ **Pior caso:** Chaves inseridas em ordem crescente ou decrescente

$$C(n) = n$$

■ **Caso Médio**

$$C(n) = \log_2 n$$

Árvore Binária de Busca sem Balanceamento – ABB – Inserção

```
public boolean insere (Item elem){
    if (this.pesquisa (elem.getChave())!=null)
        return false;
    else{
        this.raiz = this.insere (elem, this.raiz);
        return true;
    }
}
```

Árvore Binária de Busca sem Balanceamento – ABB – Remoção

```
private NoArvore remove (int chave, NoArvore arv){
    if (arv == null)
        return arv;
    else
        if (chave < arv.getInfo().getChave())
            arv.setEsq(this.remove (chave, arv.getEsq()));
        else
            if (chave > arv.getInfo().getChave())
                arv.setDir(this.remove (chave, arv.getDir()));
            else
                if (arv.getDir() == null)
                    return arv.getEsq();
                else
                    if (arv.getEsq() == null)
                        return arv.getDir();
                    else
                        arv.setEsq(this.arruma (arv, arv.getEsq()));
    return arv;
}
```

Árvore Binária de Busca sem Balanceamento – ABB – Inserção

```
private NoArvore insere (Item elem, NoArvore no){
    NoArvore novo;
    if (no == null){
        novo = new NoArvore(elem);
        return novo;
    }
    else {
        if (elem.getChave() < no.getInfo().getChave()){
            no.setEsq(this.insere (elem, no.getEsq()));
            return no;
        }
        else{
            no.setDir(this.insere (elem, no.getDir()));
            return no;
        }
    }
}
```

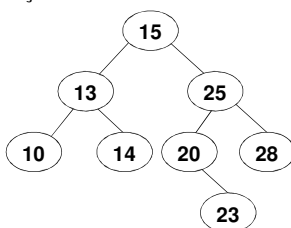
Árvore Binária de Busca sem Balanceamento – ABB – Remoção

```
private NoArvore arruma (NoArvore Q, NoArvore R){
    if (R.getDir() != null)
        R.setDir(this.arruma (Q, R.getDir()));
    else{
        Q.setInfo(R.getInfo());
        R = R.getEsq();
    }
    return R;
}
```

Árvore Binária de Busca sem Balanceamento – ABB – Remoção

- Na remoção é necessário verificar três situações:

1. Remoção de um nó folha
2. Remoção de um nó com um único filho
3. Remoção de um nó com dois filhos



Balanceamento

- Como chegar a uma árvore balanceada através de uma árvore não balanceada?
 - a) Fazer um caminhamento central, obtendo um vetor ordenado VO
 - b) Criar a árvore binária de pesquisa balanceada a partir de VO.

Caminhamento Central

```
public TabelaOrd CamCentral (Tabela vetOrd){
    return (this.FazCamCentral (this.raiz, vetOrd));
}

private TabelaOrd FazCamCentral (NoArvore arv,
                                TabelaOrd vetOrd){
    if (arv != null) {
        vetOrd = this.FazCamCentral (arv.getEsq(), vetOrd);
        vetOrd.insere (arv.getInfo());
        vetOrd = this.FazCamCentral (arv.getDir(), vetOrd);
    }
    return vetOrd;
}
```

Considerações

- ✍ O custo para manter a árvore balanceada é alto.
- ✍ Por isso não se deve estar sempre balanceando a árvore
- ✍ Logo, **não é** um método indicado para arquivos com alto potencial de crescimento.
- ✍ Mesmo para arquivo voláteis ou com baixo potencial de crescimento, esse método não deve balancear a árvore a todo momento.

Construção da Árvore Balanceada

- Pega-se o registro localizado no meio do vetor ordenado e insere em uma árvore vazia
- Repete o processo até pegar todos os elementos do vetor

Árvores AVL

- Como manter uma árvore binária balanceada?
- inserir o elemento deixando a árvore sempre balanceada
- A idéia de manter uma árvore binária balanceada dinamicamente, foi proposta em 1962 por 2 soviéticos chamados **Adelson-Velskii** e **Landis**.
- Uma árvore AVL é uma árvore binária de pesquisa onde a diferença em altura entre as subárvores esquerda e direita é -1, 0 ou 1.

Construção da Árvore Balanceada

```
public Arvore ArvoreBalanceada (TabelaOrd vetOrd){
    Arvore temp = new Arvore();
    this.Balancear (vetOrd, temp, 0, vetOrd.getnElem()-1);
    return temp;
}

private void Balancear (TabelaOrd vet, Arvore temp,
                        int inic, int fim){
    int meio;
    if (fim >= inic){
        meio = (inic+fim)/2;
        temp.insere(vet.getElemVetor(meio));
        this.Balancear (vet, temp, inic, meio - 1);
        this.Balancear (vet, temp, meio + 1, fim);
    }
}
```

Árvores AVL

- Assim, para cada nó podemos definir um **fator de balanceamento (FB)**, que vem a ser um número inteiro igual a:

$FB(nodo\ p) = altura(sub\grave{a}rvore\ direita\ p) - altura(sub\grave{a}rvore\ esquerda\ p)$

- Seja um nó qualquer da árvore **n**:
 - se $FB(n) = 0$, as duas subárvores têm a mesma altura;
 - se $FB(n) = -1$, a subárvore esquerda é mais alta que a direita em 1;
 - se $FB(n) = +1$, a subárvore direita é mais alta que a esquerda em 1.

Árvores AVL

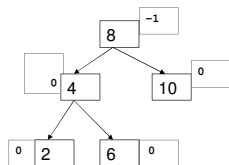
- Adotar um algoritmo que, a cada inserção, faça as correções necessárias para manter sempre a árvore como uma árvore AVL, ou seja, onde qualquer nó n tenha $|FB(n)| \leq 1$.
- A vantagem de uma árvore AVL sobre uma degenerada
 - maior eficiência nas suas operações de busca.
 - Por exemplo
 - numa árvore degenerada de 10.000 nós, são necessárias, em média, 5.000 comparações, numa busca;
 - numa árvore AVL, com o mesmo número de nós, essa média baixa para 14.

Rotação

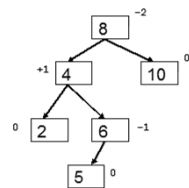
- Quando fazer rotação simples?
 - Fator de balanceamento com mesmo sinal
 - Rotação para esquerda -> sinais do FB positivos
 - Rotação para direita -> sinais do FB negativos
- Quando fazer rotação dupla?
 - Fator de balanceamento com sinais trocados
 - Rotação dupla para direita -> pai - e o filho +
 - Rotação dupla para esquerda -> pai + e o filho -

Árvores AVL

Exemplo de árvore AVL:



Exemplo de árvore binária não balanceada e não AVL:



Exemplo

- Construa uma árvore AVL inserindo as chaves: 32, 40, 48, 55, 44, 46, 22, 11, 36, 42, 39, 45 e 02, nessa ordem.

Inserção em uma árvore AVL

- Inserir um nó na árvore binária de pesquisa
- Procurar, a partir do nó inserido, se existe algum nó desbalanceado.
- A partir do nó desbalanceado corrigir o balanceamento com rotações.
- Inserir os nós 20, 10, 5, 30, 25, 27 e 28 em uma árvore inicialmente vazia.

Declaração do nó da árvore AVL

```

public class Nodo {
    private Item info;
    private Nodo esq, dir;
    private byte fatorBalanceamento;

    Nodo (Item i) {
        this.info = i;
        this.fatorBalanceamento = 0;
    }

    public Nodo getDir() { return this.dir; }
    public void setDir(Nodo dir) { this.dir = dir; }
    public Nodo getEsq() { return this.esq; }
    public void setEsq(Nodo esq) { this.esq = esq; }
    public byte getFatorBalanceamento() {
        return this.fatorBalanceamento;
    }
    public void setFatorBalanceamento(byte fatorBalanceamento) {
        this.fatorBalanceamento = fatorBalanceamento;
    }
    public Item getInfo() { return this.info; }
}
  
```

Declaração da árvore AVL

```
public class ArvoreAVL {
    private Nodo raiz;
    private boolean h;

    public ArvoreAVL(){
        this.raiz = null;
        this.h = true;
    }
    // Outros métodos
}
```

Verificar Balanceamento à Esquerda

```
private Nodo balancearEsq (Nodo no){
    if (this.h)
        switch (no.getFatorBalanceamento()){
            case -1: no.setFatorBalanceamento((byte)0);
                    this.h = false;
                    break;
            case 0 : no.setFatorBalanceamento((byte)1);
                    break;
            case 1 : no = this.rotacaoEsquerda (no);
                    return no;
        }
}
```

Inserção

```
public void insereRaiz (Item elem){
    this.raiz = this.insere (elem, this.raiz);
}

private Nodo insere (Item elem, Nodo no){
    if (no == null){
        Nodo novo = new Nodo(elem);
        this.h = true;
        return novo;
    }
    else{
        if (elem.getChave() < no.getInfo().getChave()){
            no.setEsq(this.insere (elem, no.getEsq()));
            no = this.balancearDir (no);
            return no;
        }
        else{
            no.setDir(this.insere (elem, no.getDir()));
            no = this.balancearEsq (no);
            return no;
        }
    }
}
```

Rotação à Direita

```
private Nodo rotacaoDireita (Nodo no){
    Nodo temp1, temp2;
    temp1 = no.getEsq();
    if (temp1.getFatorBalanceamento() == -1){
        no.setEsq(temp1.getDir());
        temp1.setDir(no);
        no.setFatorBalanceamento((byte)0);
        no = temp1;
    }
    else {
        temp2 = temp1.getDir();
        temp1.setDir(temp2.getEsq());
        temp2.setEsq(temp1);
        no.setEsq(temp2.getDir());
        temp2.setDir(no);
        if (temp2.getFatorBalanceamento() == -1)
            no.setFatorBalanceamento((byte)1);
        else
            no.setFatorBalanceamento((byte)0);
        if (temp2.getFatorBalanceamento() == 1)
            temp1.setFatorBalanceamento((byte)-1);
        else
            temp1.setFatorBalanceamento((byte)0);
        no = temp2;
    }
    no.setFatorBalanceamento((byte)0);
    this.h = false;
    return no;
}
```

Verificar Balanceamento à Direita

```
private Nodo balancearDir (Nodo no){
    if (this.h)
        switch (no.getFatorBalanceamento()){
            case 1 : no.setFatorBalanceamento((byte)0);
                    this.h = false;
                    break;
            case 0 : no.setFatorBalanceamento((byte)-1);
                    break;
            case -1: no = this.rotacaoDireita (no);
                    return no;
        }
    return no;
}
```

Rotação à Esquerda

```
private Nodo rotacaoEsquerda (Nodo no){
    Nodo temp1, temp2;
    temp1 = no.getDir();
    if (temp1.getFatorBalanceamento() == 1){
        no.setDir (temp1.getEsq());
        temp1.setEsq(no);
        no.setFatorBalanceamento((byte)0);
        no = temp1;
    }
    else {
        temp2 = temp1.getEsq();
        temp1.setEsq(temp2.getDir());
        temp2.setDir(temp1);
        no.setDir(temp2.getEsq());
        temp2.setEsq(no);
        if (temp2.getFatorBalanceamento() == 1)
            no.setFatorBalanceamento((byte)-1);
        else
            no.setFatorBalanceamento((byte)0);
        if (temp2.getFatorBalanceamento() == -1)
            temp1.setFatorBalanceamento((byte)1);
        else
            temp1.setFatorBalanceamento((byte)0);
        no = temp2;
    }
    no.setFatorBalanceamento((byte)0);
    this.h = false;
    return no;
}
```

HASHING

- Hash significa:
 - Fazer picadinho de carne e vegetais para cozinhar
 - Fazer uma bagunça
- Os registros são armazenados em uma tabela e podem ser endereçados diretamente através de uma transformação aritmética sobre a chave de pesquisa.

Função de Transformação

Se a chave for um valor numérico

```
public int Hashing (int chave, int M){
    return chave%M;
}
```

Se a chave não for um valor numérico

```
public int Hashing (String chave){
    char carac;
    int i, soma=0;

    for (i=0; i<chave.length(); i++){
        carac = chave.charAt(i);
        soma += Character.getNumericValue(carac);
    }
    return soma;
}
```

HASHING

- Constituído de duas etapas:
 - Computar o valor da função de transformação (FUNÇÃO HASHING) que transforma a chave de pesquisa em um endereço da tabela.
 - Se duas ou mais chaves forem transformadas em um mesmo endereço da tabela (**colisão**), é necessário um método para tratar esse problema.

Endereçamento Aberto

- Exemplo:
 - Chaves entre [50 , 500];
 - quantidade de registros **N = 11** ⇒ número primo **m = 13**
 - Chaves a serem inseridas: 102; 200; 196; 53

$102 \bmod 13 = 11$
 $200 \bmod 13 = 5$
 $196 \bmod 13 = 1$
 $53 \bmod 13 = 1$

	196	53			200			102	
0	1	2	3	4	5	6	...	11	m-1=12

Função de Transformação

- $H(K) = K \bmod m$
 - **K** é o número a ser transformado
 - **m** é um número primo maior ou igual a quantidade de registros a serem armazenados.
 - Devem ser evitados os números primos obtidos a partir de $b^i \pm j$
 - Onde **b** é a base de um conjunto de caracteres. (64 para BCD; 128 para ASCII; 256 para EBCDIC ou 100 para alguns códigos decimais)
- Exemplo com $n = 100$

Endereçamento Aberto

- Exemplo:
 - Seja uma tabela com, no máximo, 53 registros, sendo que a chave primária possui 3 dígitos. Devemos mapear chaves desse tipo (000 até 999) na tabela de 53 entradas (0 até 52).

Chave	383	487	235	527	510	320	203	108	563	646	063
Endereço	12	10	23	50	33	2	44	2	33	10	10
End. Efetivo	12	10	23	50	33	2	44	3	34	11	13

Chave	383	487	235	527	510	320	203	108	563	646	063
Endereço	12	10	23	50	33	2	44	2	33	10	10

Diagrama de endereçamento aberto mostrando colisões e resolução:

- Chave 383 → 12
- Chave 487 → 10
- Chave 235 → 23
- Chave 527 → 50
- Chave 510 → 33
- Chave 320 → 2
- Chave 203 → 44
- Chave 108 → 2 (colisão com 320) → 3
- Chave 563 → 33 (colisão com 510) → 34
- Chave 646 → 10 (colisão com 487) → 11
- Chave 063 → 10 (colisão com 487) → 13

Endereçamento Aberto

- Segundo Knuth (1973), o custo médio de uma pesquisa com sucesso é

$$C(n) = \frac{1}{2} \left(1 + \frac{1}{1 - \frac{n}{m}} \right)$$

- n = número de registros
- m = primo adotado na função hashing

Vetor Encadeado

- Custo:
 - Melhor caso: $C(n) = O(1)$
 - Pior caso: $C(n) = O(1)$
 - Caso médio: $C(n) = O(1)$
- Vantagens do vetor encadeado sobre o endereçamento aberto?

Endereçamento Aberto

- Custo:
 - Melhor caso: $C(n) = O(1)$
 - Pior caso: $C(n) = O(n)$
 - Caso médio: $C(n) = O(1)$

N/M	C(n)
0.10	1.06
0.25	1.17
0.50	1.50
0.75	2.50
0.90	5.50
0.95	10.50

Vetor Encadeado

