

## CAPÍTULO III - PESQUISA

### III.1 INTRODUÇÃO

Nesta parte, veremos como recuperar uma informação a partir de várias outras armazenadas em uma tabela ou arquivo.

A informação é dividida em registros, onde cada um possui uma chave que será usada na pesquisa. Quando encontramos uma ou mais ocorrências de registros com chaves iguais à **chave de pesquisa**, dizemos que a pesquisa ocorreu **com sucesso**. Caso contrário, a pesquisa é **sem sucesso**.

Existem vários métodos de pesquisa que podem ser aplicados. A escolha mais adequada depende:

- ☺ Da quantidade de dados envolvidos
- ☺ Da volatilidade do arquivo (inserções e retiradas freqüentes).

Obs.: 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.

Devemos considerar os arquivos e/ou tabelas como tipos abstratos de dados com um conjunto de operações associadas a esta estrutura de dados, onde as operações 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.

A operação 5 já foi tratada anteriormente e a 6 não será tratada neste capítulo, pois requer a utilização de técnica mais sofisticadas.

Alguns métodos de pesquisa serão vistos com as operações *INICIALIZAR*, *PESQUISAR*, *INSERIR* e *REMOVER*.

Com relação ao algoritmo de *PESQUISA*, devemos considerar como medida relevante de complexidade apenas o número de comparações entre chaves. Já na *INSERÇÃO* e *REMOÇÃO*, o número de movimentações de itens também pode ser importante. Aqui neste capítulo, estudaremos a complexidade apenas dos algoritmos de *PESQUISA*.

Vamos agora, caracterizar a estrutura de dados **tabela** e/ou **arquivo**, sobre a qual os métodos vão operar, primeiro em Pascal e posteriormente em Java.

De uma forma geral, o termo **tabela** identifica um arquivo na memória principal, enquanto o termo **arquivo** é usado para arquivos na memória secundária.

Tanto em Pascal quanto em Java, pelo menos um dos campos (atributos) deve ter um valor diferente para cada entrada (**chave primária**), identificando um elemento da tabela

e/ou do arquivo sem ambiguidades e, normalmente, é usado para organizar a tabela e/ou o arquivo.

Podemos também definir um **item de busca** como um campo (ou atributo) da tabela que pode ser usado para pesquisa, mas que não, necessariamente, seja a chave primária nem identifica de forma única um elemento.

### III.1.1 Estruturas de Dados

As estruturas trabalhadas são formadas por uma coleção de **objetos (Item)**, que, por sua vez, são formados por um conjunto de atributos. A estrutura de dados **Item** é apresentada a seguir:

```
public class Item {
    private tipoChave chave;
    // outros atributos
    // construtor e métodos para manipular os atributos dentre eles:
    public tipoChave getChave ( ){
        return this.chave;
    }
}
```

Onde o campo **chave** da classe **Item** é a chave primária.

Ao armazenar a tabela através de um array, temos:

```
public class Tabela{
    private Item[] vetor; //referência a um vetor de itens
    private int nElem; //número de itens de dados
    public Tabela (int tamanho){
        this.vetor = new Item[tamanho];
        this.nElem = 0;
    } // outros métodos da classe
}
```

Se a pesquisa for implementada com apontadores, teremos a seguinte estrutura:

```
public class No {
    private Item info;
    private No prox;
    public No (Item _info){
        this.info = _info; //prox é automaticamente definido como null
    }
    public No getProx (){
        return this.prox;
    }
    public void setProx(No novo){
        this.prox = novo;
    }
}

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

## III.2 PESQUISA SEQUENCIAL EM TABELA DESORDENADA

A pesquisa sequencial é a técnica mais simples de realizar uma busca. Começa-se a pesquisar a partir do primeiro elemento (ou último) até que se encontre a chave procurada ou até que os elementos da estrutura se esgotem. Para fazer este tipo de pesquisa o conjunto de dados não precisa estar ordenado. Pode-se usar vetor ou lista encadeada.

### III.2.1 Pesquisa sequencial em Vetor

Neste momento, vamos apresentar este tipo de pesquisa usando uma lista contígua.

#### III.2.1.1 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:** elemento pesquisado é o último  $C(n) = 1$  **Pior caso:** elemento não existe no vetor  $C(n) = n$

#### III.2.1.2 Procedimento Insere

Este método retorna **false** se houver erro ao inserir um elemento no vetor (vetor cheio), e **true** se a inserção for bem sucedida.

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

#### III.2.1.3 Procedimento Remove

Este método retorna **false** se houver erro ao remover um elemento no vetor (vetor vazio ou elemento não existe), e **true** se a remoção for bem sucedida.

```
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];
            return true;
        }else
            return false;
    }
}
```

## CONSIDERAÇÕES:

- ✎ Este algoritmo é a melhor solução para problemas de pesquisa em tabelas com até 25 registros
- ✎ A Função Pesquisa retorna o índice do registro que contém a chave procurada. Se retornar zero (Pascal) ou -1 (Java) é porque não encontrou o item.
- ✎ Não suporta mais de um registro com a mesma chave. Para passar a aceitar é necessária uma reformulação.

Vamos ver agora como ficariam os algoritmos se usarmos a estrutura de lista encadeada:

### III.2.2 Pesquisa sequencial em Lista Simplesmente Encadeada (LSE)

Neste momento, veremos a pesquisa sequencial em uma LSE.

#### III.2.2.1 Pesquisa sequencial em lista simplesmente encadeada

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

#### CUSTO:

**Melhor caso:** elemento pesquisado é o primeiro **Pior caso:** elemento não existe na lista

$$C(n) = 1$$

$$C(n) = n$$

#### III.2.2.2 Inserção

```
public void inserePrimeiro(Item elem) {  
    No novoNo = new No (elem);  
    novoNo.setProx(this.prim);  
    this.prim = novoNo;  
}
```

### III.2.2.3 Remoção

```
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;
}
```

## III.3 PESQUISA BINÁRIA

A pesquisa em uma tabela pode ser muito mais eficiente se os registros estiverem ordenados. Uma técnica de busca em tabelas ordenadas, organizadas na forma de vetores, é a Busca Binária, cuja estratégia é a seguinte:

- ✎ compara-se a chave de pesquisa com a chave do registro *que* está no meio do vetor.
  - ✓ Se a chave de busca for menor que a chave deste registro, então o registro procurado deve estar na primeira metade da tabela;
  - ✓ se maior, então deve estar na segunda metade;
  - ✓ se igual, a pesquisa foi realizada com sucesso.
- ✎ O processo deve ser repetido até se encontrar o registro (**pesquisa com sucesso**) ou ficar apenas um registro com chave diferente da chave de pesquisa (**pesquisa sem sucesso**).

### III.3.1 Pesquisa Binária em Vetor

```
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;
}
```

Exemplo: Pesquisando o número 3

Esq										Dir
3	5	6	8	9	12	18	20	24	29	35
1	2	3	4	5	6	7	8	9	10	11

Meio =  $(1 + 11) \div 2 = 6$

3 = V[ meio ] ? NÃO

3 < V[ meio ] ? SIM → Dir = Meio – 1

Esq										Dir
3	5	6	8	9	12	18	20	24	29	35
1	2	3	4	5	6	7	8	9	10	11

Meio =  $(1 + 5) \div 2 = 3$

3 = V[ meio ] ? NÃO

3 < V[ meio ] ? SIM → Dir = Meio – 1

Esq										Dir
3	5	6	8	9	12	18	20	24	29	35
1	2	3	4	5	6	7	8	9	10	11

Meio =  $(1 + 2) \div 2 = 1$

3 = V[ meio ] ? SIM → RETORNA 1

**Análise do algoritmo:**

**Melhor caso:**  $C(n) = 1$  , elemento procurado está no meio do vetor.

**Pior caso:**  $C(n) = \log_2 n$ , elemento não existe no vetor

**Caso médio:**  $O(\log_2 n)$

### III.3.2 Procedimento Insere – Insere ordenadamente

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

### III.3.3 Procedimento Retira – Remove ordenadamente

```
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;
    }
}
```

## 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.

### III.4 ÁRVORES DE BUSCA

São estruturas de dados muito eficientes para armazenar informações. São utilizadas quando há necessidade dos seguintes requisitos:

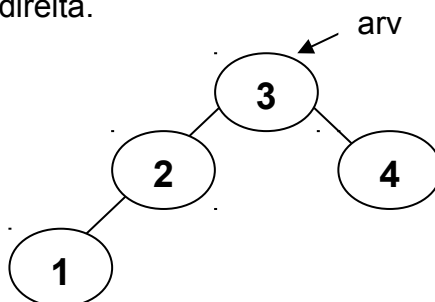
- ✎ Acessos direto e seqüencial eficientes
- ✎ Facilidade de inserção e remoção de elementos
- ✎ Alta taxa de utilização de memória
- ✎ Utilização de memória primária e secundária

Ao analisar esses requisitos de forma separada, podemos achar estruturas de dados melhores, mas ao juntarmos todos ou alguns dos requisitos, as árvores de busca são a melhor solução.

#### III.4.1 Árvore Binária de Busca – ABB

ABB é uma árvore binária onde os elementos menores que a raiz estão à esquerda da mesma e os elementos maiores à direita.

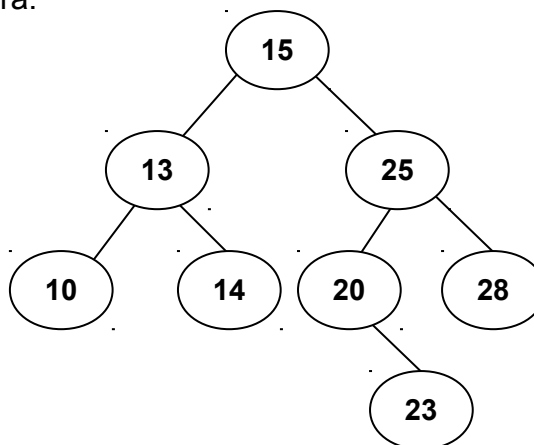
Exemplo:



#### Caminhamento Central:

Ao fazer o caminhamento central em uma ABB obtemos um vetor em ordem crescente. Este caminhamento é feito da seguinte maneira:

1. Percorrer a sub-árvore esquerda
2. Visitar raiz
3. Percorrer sub-árvore direita



Da árvore acima, obtemos:

{10 – 13 – 14 – 15 – 20 – 23 – 25 – 28}

Estrutura de uma árvore de pesquisa:

```
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;
    }
}

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

### III.4.2 Árvore Binária de Busca sem Balanceamento – ABB

Inicialmente vamos trabalhar com uma árvore binária de pesquisa sem realizar balanceamento.

#### III.4.2.1 Pesquisa

```
public boolean pesquisa (int chave){
    NoArvore temp;

    temp = this.pesquisa (chave, this.raiz);
    if (temp != null)
        return true;
    else
        return false;
}
```



```
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;
}
```

#### III.4.2.2 Insere

```
public boolean insere (Item elem){
    boolean existe;

    existe = this.pesquisa (elem.getChave());
    if (existe)
        return false;
    else{
        this.raiz = this.insere (elem, this.raiz);
        return true;
    }
}

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;
        }
    }
}
```

#### III.4.2.3 Remoção

Na remoção é necessário verificar três situações:

1. Remoção de um nó folha: É o caso mais simples. Basta remover o nó.
2. Remoção de um nó com um único filho: Este também é um caso simples. Basta fazer a raiz apontar para este único filho.

3. Remoção de um nó com dois filhos: Neste caso, deve-se reorganizar a árvore para enquadrar este caso em um dos anteriores. Isto é feito trocando-se o nó a ser removido pelo nó mais à direita da sub-árvore esquerda, ou pelo nó mais à esquerda da sub-árvore direita. Assim, cairemos em um dos casos mais simples.

```
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;
    }
}

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;
}
```

☺ **Análise:**

**Depende do formato da árvore.**

**Pior Caso:** Chaves Inseridas em ordem crescente ou decrescente.

Exemplo: 7, 8, 10, 18, 21, 24

$$C(n) = O(n)$$

**Melhor Caso:** Chave de busca na raiz.

$$C(n) = O(1)$$

**Caso Médio :**

$$C(n) = O(\log n)$$

Pode-se evitar o pior caso, Como?

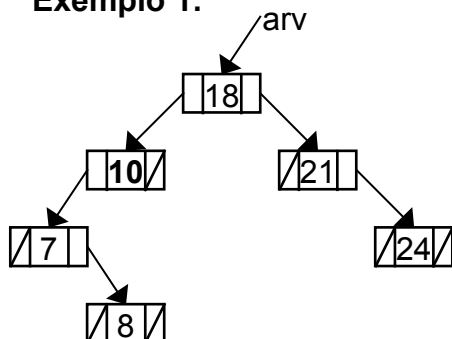
Fazendo o balanceamento da árvore.

### III.4.3 Balanceamento

Como chegar a uma árvore balanceada através de uma árvore não balanceada?

- Fazer um caminhamento central, obtendo um vetor ordenado VO
- Construir um vetor balanceado, VB, a partir de VO.
- Criar a árvore binária de pesquisa balanceada a partir de VB.

#### Exemplo 1:



VO 

7	8	10	18	21	24
---	---	----	----	----	----

Vamos escrever agora dois procedimentos para balancear a árvore:

#### III.4.3.1 Caminhamento Central

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

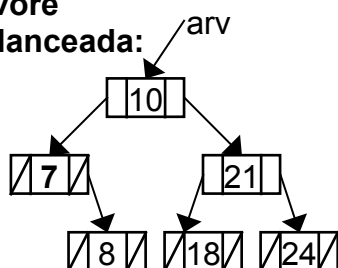
private TabelaOrd FazCamCentral (NoArvore arv, TabelaOrd vetOrdenado){
    if (arv != null) {
        vetOrdenado = this.FazCamCentral (arv.getEsq(), vetOrdenado);
        vetOrdenado.insere (arv.getInfo());
        vetOrdenado = this.FazCamCentral (arv.getDir(), vetOrdenado);
    }
    return vetOrdenado;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

#### III.4.3.2 Construção da Árvore balanceada

Pega-se o registro localizado no meio do vetor ordenado e vai repetindo o processo

#### Árvore

#### Balanceada:



VB 

7	8	10	18	21	24
---	---	----	----	----	----

**Observação :** *inic* é a posição do primeiro elemento armazenado em VO e *fim* é a posição do último elemento em VO.

```
public Arvore ArvoreBalanceada (TabelaOrd vetOrdenado){
    Arvore temp = new Arvore();
    this.Balancear (vetOrdenado, temp, 0, vetOrdenado.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);
    }
}
```

### III.5 ÁRVORES AVL

As árvores binárias de pesquisa têm uma séria desvantagem que pode afetar o tempo necessário para recuperar um item armazenado. A desvantagem é que a estrutura da árvore ***depende da ordem em que os elementos são inseridos***. Sabemos que se inserirmos os elementos já em ordem, a estrutura da árvore será igual a de uma lista encadeada e o tempo médio para recuperar uma informação da árvore aumenta, no pior caso  $O(n)$ . Um meio de evitarmos que a árvore se transforme em um LSE é fazermos o balanceamento da mesma.

Uma técnica foi apresentada seguindo os seguintes passos:

- 1) gerar o vetor ordenado (VO)
- 2) gerar o vetor balanceado (VB)
- 3) gerar a árvore balanceada a partir de VB

Vantagem:

- O balanceamento (dependendo da volatilidade do conjunto de dados) pode ser feito de tempos em tempos (não é feito a todo momento), aumentando eficiência.

Desvantagem:

- Necessidade de memória extra (os vetores)

Uma outra idéia de manter a árvore balanceada: é inserir o elemento deixando a árvore sempre balanceada (insere, balanceia, insere, balanceia).

A idéia de manter uma árvore binária balanceada dinamicamente, ou seja, enquanto os nós estão sendo inseridos foi proposta em 1962 por 2 soviéticos chamados **Adelson-Velskii** e **Landis**. Este tipo de árvore ficou então conhecida como **árvore AVL**, pelas iniciais dos nomes dos seus inventores. Como definimos uma árvore AVL?

Por definição uma árvore AVL é uma árvore binária de pesquisa onde a diferença em altura entre as subárvores esquerda e direita é no máximo 1 (positivo ou negativo).

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

$$\text{FB}(\text{nodo } p) = \text{altura}(\text{subárvore direita } p) - \text{altura}(\text{subárvore esquerda } p)$$

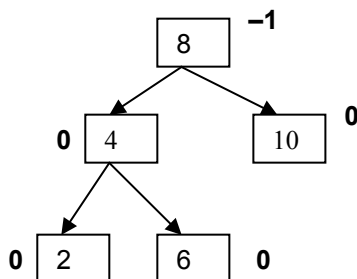
Seja um nó qualquer da árvore, apontado por **P**:

- ✎ se  $\text{FB}(n) = 0$ , as duas subárvores têm a mesma altura;
- ✎ se  $\text{FB}(n) = -1$ , a subárvore esquerda é mais alta que a direita em 1;
- ✎ se  $\text{FB}(n) = +1$ , a subárvore direita é mais alta que a esquerda em 1.

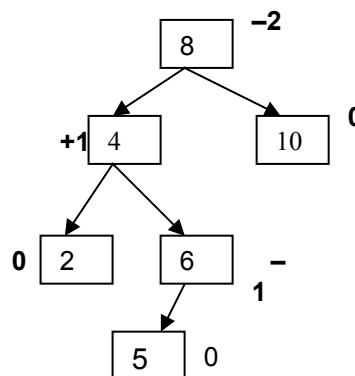
Como, na prática, não podemos prever qual será a ordem de entrada dos valores, e, muito menos, alterar essa ordem para que ela seja conveniente à construção de uma árvore balanceada, a solução é adotar um algoritmo que, a cada inserção, faça as correções necessárias para manter sempre a árvore como uma árvore AVL, ou seja, onde qualquer nó  $n$  tenha  $|\text{FB}(n)| \leq 1$ .

A vantagem de uma árvore AVL sobre uma degenerada (árvore no formato LSE) está na maior eficiência nas suas operações de busca, pois, sendo a altura da AVL bem menor, o número necessário de comparações diminui sensivelmente. 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.

Exemplo de árvore AVL:



Exemplo de árvore binária não balanceada e não AVL:



Objetivo principal: como deixar sempre a árvore balanceada?

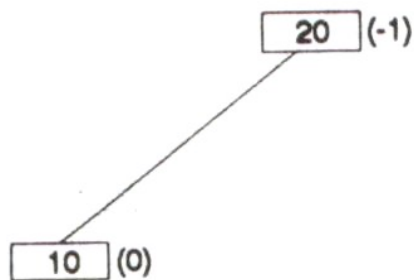
⇒ **Inserção em uma árvore AVL**

A ideia geral da inserção é:

- 1) Inserir um nó na árvore binária de pesquisa
- 2) Procurar, a partir do nó inserido, se existe algum nó desbalanceado.
- 3) A partir do nó desbalanceado corrigir o balanceamento com rotações.
  - 3.1) Escolher entre rotação simples ou dupla

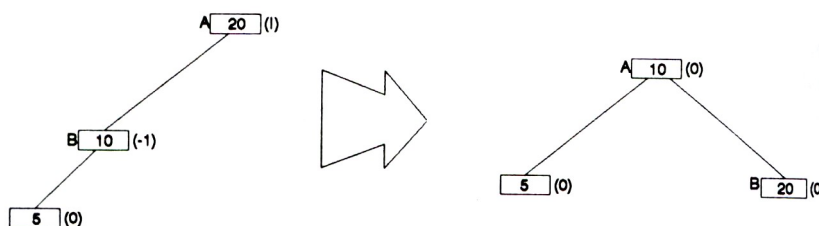
Exemplo:

### INSERÇÃO DAS CHAVES 20 E 10



Não gera desbalanceamento.

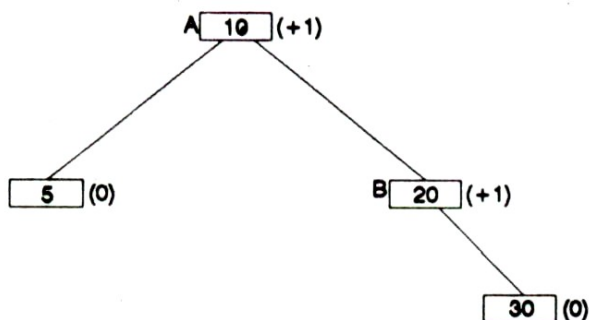
### INSERÇÃO DA CHAVE 5



Esta operação desbalanceou a árvore. O que fazer?

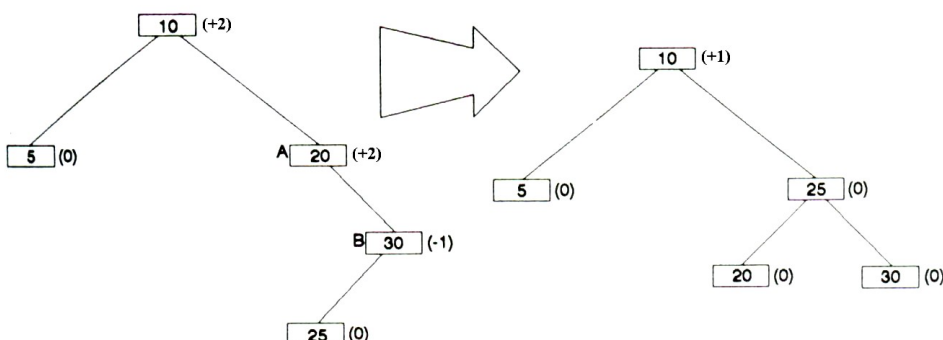
Resposta: uma rotação simples para a direita.

### INSERÇÃO DO 30



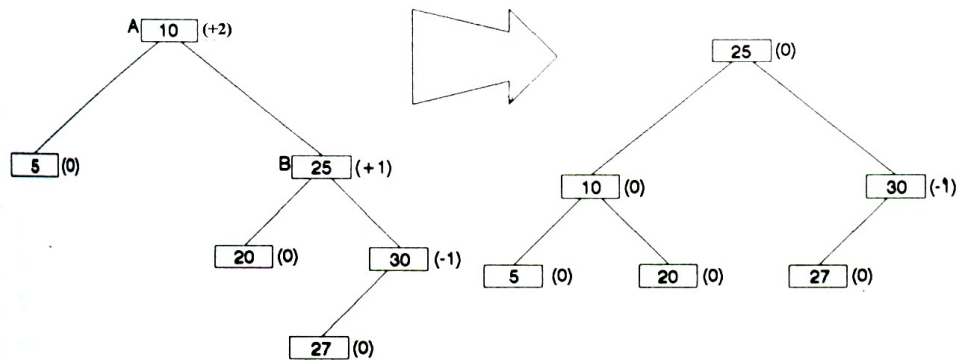
Não causa desbalanceamento.

### INSERÇÃO DO 25



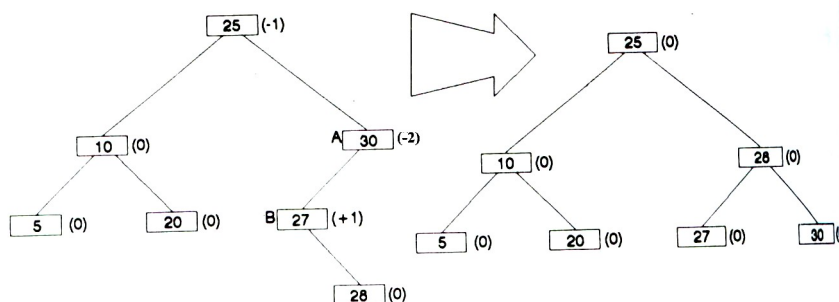
Aqui temos uma rotação simples para a direita entre as chaves 25 e 30 e depois uma rotação para a esquerda envolvendo as chaves 20, 25 e 30. É uma rotação dupla para a esquerda.

### INSERÇÃO DO 27



Aqui temos uma rotação simples para a esquerda envolvendo toda a árvore.

### INSERÇÃO DO 28



Aqui temos uma rotação simples para a esquerda entre as chaves 27 e 28 e depois uma rotação para a direita envolvendo as chaves 30, 27 e 28. É uma rotação dupla para a direita.

⇒ 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 –

## ⇒ Declaração da classe Nodo

```
public class Nodo {  
    private Item info;  
    private Nodo esq, dir;  
    private byte fatorBalanceamento;  
  
    Nodo (Item i){//construtor  
        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 classe árvore AVL

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

### 1) Método para inserir um nó na árvore

```
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()){  
            // Inserir à esquerda e verifica se precisa balancear à direita  
            no.setEsq(this.inserir (elem, no.getEsq()));  
            no = this.balancearDir (no);  
            return no;  
        }  
        else{  
            // Inserir à direita e verifica se precisa balancear à esquerda  
            no.setDir(this.inserir (elem, no.getDir()));  
            no = this.balancearEsq (no);  
            return no;  
        }  
    }  
}  
}
```

## 2) Método para verificar se é necessário o balanceamento para direita do nó

```
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;  
}
```

## 3) Método para verificar se é necessário o balanceamento para esquerda do nó

```
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;  
}
```

#### 4) Método para realizar ROTAÇÃO DIREITA (RD) ou ROTAÇÃO DUPLA DIREITA (RDD)

```
private Nodo rotaçãoDireita (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;
}
```

Faz RD

Faz RDD

Recalcula o FB do nó à direita na nova árvore

Recalcula o FB do nó à esquerda na nova árvore

A raiz terá FB = 0

#### 5) Método para realizar ROTAÇÃO ESQUERDA (RE) ou ROTAÇÃO DUPLA ESQUERDA (RDE)

```
private Nodo rotaçãoEsquerda (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;
    }
```

### III.6 TRANSFORMAÇÃO DE CHAVES (HASHING)

Os métodos de pesquisa apresentados são baseados na comparação entre a chave de pesquisa e as chaves armazenadas na tabela.

O método de transformação de chave (hashing) é completamente diferente, pois os registros são armazenados em uma tabela e podem ser endereçados diretamente através de uma transformação aritmética sobre a chave de pesquisa.

Hash significa:

- Fazer picadinho de carne e vegetais para cozinhar
- Fazer uma bagunça (Webster's New World Dictionary)

Podemos dizer que este método é constituído de duas etapas:

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

#### III.6.1 Função de transformação (Função HASHING)

Um exemplo de função de transformação:  $H(K) = K \bmod m$ .

Onde:

- **K** é o número a ser transformado (pode ser a chave)
- **m** é um número primo maior ou igual a quantidade de registros a serem armazenados.

Esta função é uma das mais utilizadas.

Devemos ter cuidado na escolha do valor de **m**. Ele deve ser um número primo, mas não qualquer primo. Devem ser evitados os números primos obtidos a partir de:

$$b^i \pm j$$

Onde **b** é a base de um conjunto de caracteres. Geralmente, **b** é 64 para BCD; 128 para ASCII; 256 para EBCDIC ou 100 para alguns códigos decimais. **i** e **j** são pequenos inteiros.

**Exemplo: 100 registros, quais números não deveriam ser escolhidos?**

**101** deveria ser evitado pois:  $100^1 + 1 = 101$

**103** deveria ser evitado pois:  $100^1 + 3 = 103$

### III.6.1.1 Se a chave for um valor numérico (Pascal)

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

### III.6.1.2 Se a chave for alfanumérica deve-se transforma-la em numérica:

Uma possibilidade, que não é a melhor, é fazer a transformação a seguir:

```
public int Hashing (String chave){  
    char caract;  
    int i, soma=0;  
  
    for (i=0; i<chave.length(); i++){  
        caract = chave.charAt(i);  
        soma += Character.getNumericValue(caract);  
    }  
    return soma;  
}
```

### III.6.2 Tratamento de colisões

Exemplo 1:

Chaves entre [100 , 500]; quantidade máxima de registros é **N = 12** □ número primo maior ou igual a 12 é **m = 13**

Chaves a serem inseridas: 102; 200; 196; 53

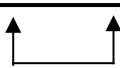
✍  $102 \bmod 13 = 11$

✍  $200 \bmod 13 = 5$

✍  $196 \bmod 13 = 1$

✍  $53 \bmod 13 = 1$

Chave	102	200	196	53
Endereço	11	5	1	1



Pode-se observar que há colisões. Como tratá-las?

O tratamento pode ser feito através do mecanismo de armazenamento:

#### III.6.2.1 Endereçamento Aberto

É um dos métodos mais simples. É possível utilizar este método quando há **N** registros para armazenar em uma tabela de tamanho **M**, onde **M > N**.

Utiliza-se um vetor de índices entre o intervalo  $[0, m-1]$ , as chaves que colidem são armazenadas no próximo índice vazio.

Portanto, o exemplo anterior ficaria:

Chave	102	200	196	53
Endereço	11	5	1	1
End. Efetivo	11	5	1	2

**OBS.:** Uma função hashing, que transforma uma chave  $x$  em um endereço base  $h(x)$  é dita **ideal**, se produzir um número baixo de colisões, se for facilmente computável e se todas as posições tiverem a mesma probabilidade de serem ocupadas (uniforme).

Exemplo 2:

Seja uma tabela com, no máximo, 53 registros, sendo que a chave primária possui 3 dígitos. Para mapear o domínio das possíveis chaves (000 até 999) na tabela de 53 entradas (0 até 52), usaremos a seguinte função Hashing:

$$H(K) = K \bmod 53$$

Assim, teremos o seguinte mapeamento:

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

Portanto, este exemplo ficaria:

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

Neste caso, segundo Knuth (1973), o custo médio de uma pesquisa com sucesso é

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

onde  $n$  = número de registros e

$m$  = primo adotado na função hashing

Custo: **Melhor caso:**  $C(n) = O(1)$

**Pior caso:**  $C(n) = O(n)$

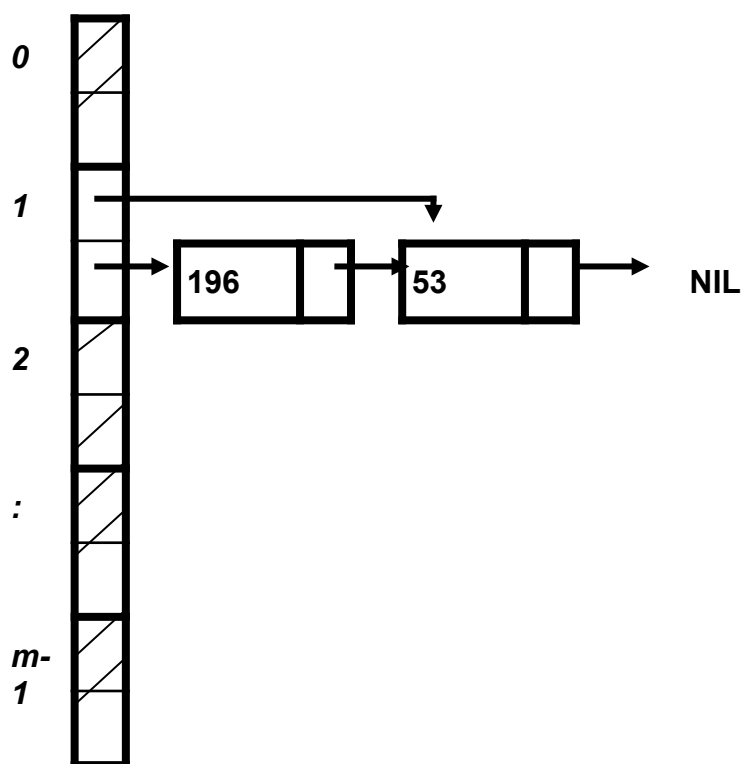
**Caso médio:**  $C(n) = O(1)$

Se os registros não forem espalhados de forma razoável o tempo necessário para novas pesquisas será maior, pois medida que a tabela vai ficando cheia, a nova chave inserida tende a ocupar uma posição na tabela que esteja vizinho a outras posições já ocupadas, deteriorando o tempo necessário para novas pesquisas. Isto pode ser visto na tabela a seguir:

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

### III.6.2.2 Vetor Encadeado

Neste método, todas as chaves transformadas, que colidem no mesmo endereço, são colocadas em uma lista encadeada. Ao ser calculado o endereço, a busca restringe-se à lista encadeada correspondente à lista que está na posição calculada do vetor.



A estrutura de dados lista encadeada utilizada é apresentada abaixo:

Observe que esta estrutura é a mesma para qualquer linguagem.

Custo:

**Melhor caso:**  $C(n) = 1$

**Pior caso:**  $C(n)$  = proporcional ao tamanho da lista

**Caso médio:**  $C(n) = O(1)$

Vantagens do vetor encadeado sobre o endereçamento aberto?

### III.7 EXERCÍCIOS

1) Suponha que:

- a) Exista um conjunto de dados com elevada atividade e pequeno potencial de crescimento.
- b) Exista um conjunto de dados extremamente volátil, com elevado potencial de crescimento.

Deseja-se obter o menor tempo possível de resposta.

Que tipo de algoritmo de pesquisa você indicaria para os casos a e b? Por quê?

2) Entre os seguintes métodos de pesquisa: Pesquisa sequencial em tabela desordenada (vetor), pesquisa binária em tabela ordenada (busca binária) e árvore binária de pesquisa (vista em estrutura de dados), avalie para as situações abaixo, qual o método de pesquisa mais indicado, justificando.

- a) Processamento de consultas a um conjunto de dados com cerca de 500 registros, sendo este conjunto de baixa atividade, pouco volátil e com pequeno potencial de crescimento.
- b) Pesquisa a um conjunto de dados com 1000 registros, elevado potencial de crescimento e média atividade.

3) Para os casos abaixo, diga qual o algoritmo de pesquisa mais adequado, justificando.

- a) Suponha um conjunto de 5.500 elementos, com nenhum potencial de crescimento, baixa volatilidade e alta atividade.
- b) Suponha que haja um conjunto de 10.000 dados com alto potencial de crescimento, e muita atividade.
- c) Suponha que haja um conjunto de 30 elementos muito volátil, nenhum potencial de crescimento e com pouca atividade.
- d) Suponha que haja um conjunto de 5.000 elementos com alta atividade e muito volátil, mas baixo potencial de crescimento.

4) Considere as técnicas de pesquisa sequencial, pesquisa binária e árvore de pesquisa balanceada e não balanceada.

- a) Descreva as vantagens e desvantagens de cada um dos métodos acima. Em que situações você usaria cada um deles?
- b) Dê a ordem do melhor e do pior caso esperado de tempo de execução em cada um deles.

5) Qual a principal propriedade de uma árvore binária de pesquisa?

6) Se as chaves Q U E S T A O F C I L fossem inseridas em uma árvore binária inicialmente vazia:

- Desenhe a árvore binária resultante.
- Desenhe as árvores resultantes das retiradas dos elementos **E** depois **U** da árvore do item (a).
- Desenhe a árvore binária balanceada resultante da questão inicial.

7) Relacione a coluna da direita com a da esquerda.

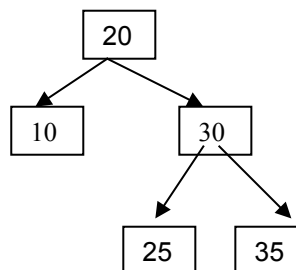
- |                               |  |
|-------------------------------|--|
| [ a ] Árvore Binária de Busca | [   ] Conjunto de dados onde o acesso rápido é essencial. Este conjunto de dados é estático. |
| [ b ] Tabela Hashing          | [   ] Um conjunto de 20 registros.   |
| [ c ] Pesquisa Sequencial     | [   ] Deseja acesso rápido as informações e exibição dos dados ordenados de forma eficiente. |

8) Suponha que um programa deva ser implementado para armazenar 60 registros em um vetor. Este vetor utilizará o método de pesquisa hashing. Defina a função de transformação hashing, (a) endereçamento aberto e (b) vetor encadeado, justificando. Exemplo: chave mod M. Você deverá definir o valor de M.

9) Cite as duas formas de se tratar colisões em uma tabela hashing. Qual das duas é a mais eficiente? Justifique.

10) Construir uma árvore AVL com onde os elementos a seguir são inseridos na ordem em que aparecem: 60, 40, 20, 50, 56, 10, 65, 53, 68, 58, 51.

11) Seja a árvore AVL a seguir, insira as chaves: 23, 15 e 12, 17.



12) Suponha as seguintes chaves: **45, 66, 47, 26, 7, 10**

- Defina a equação hashing, justificando.
- Como eles ficariam armazenados em um vetor considerando a utilização do endereçamento aberto? Suponha que a ordem de inserção na tabela seja a ordem em que as chaves aparecem acima (da esquerda para a direita), ou seja, o primeiro elemento a ser inserido seria o 45, o segundo 66, e assim por diante.
- Considere agora um vetor por encadeamento, como ficaria a sua configuração? (desenhe o vetor encadeado com os elementos inseridos na ordem apresentada acima).



13) Uma tabela Hash Interna tem 30 posições (0..29). As chaves são inteiros e a função hashing  $h(i) = i \bmod 30$  é usada, com endereçamento aberto. Se os elementos com chaves 91, 240, 31, 122, 63, 150, 35 são inseridos, nesta ordem, em uma tabela inicialmente vazia, responda:

- a) Faça o vetor correspondente à tabela Hash acima.
- b) qual o número de comparações que devem ser realizadas para achar o inteiro 150?
- c) Qual o elemento que está na posição 2 do vetor?

14) Supondo que deve ser feita uma tabela Hash com 700 elementos, usando endereçamento aberto. Qual o tamanho ideal do vetor para que diminua o número de colisões sem desperdiçar muita memória? E usando vetor encadeado? Justifique.

## CAPÍTULO IV – ARQUIVO

### IV.1 INTRODUÇÃO

Arquivo é um conjunto de informações que residem em memória secundária ou terciária. Eles podem ser volumosos e duradouros.

O objetivo deste capítulo é estudar as diferentes formas de organização de arquivos, seus mecanismos, como ocorrem pesquisas e ordenações, dentre outros itens.

Um arquivo pode ser utilizado tanto em programação de sistemas de computação quanto em programas de aplicação.

Em sistemas de computação, por exemplo:

- Uso pelas LP's para armazenar o programa fonte, o executável;
- Editores de texto que armazenam arquivos temporários, etc.

Em programas de aplicação:

- Para armazenar informações: editor de texto armazena o texto em um arquivo, uma planilha é armazenada em um arquivo;
- Uma imagem é armazenada em um arquivo;
- Dados de um banco de dados são armazenados em arquivos.

Os sistemas operacionais têm a finalidade de gerenciar os recursos do sistema, como memória, processos, etc. A gerência mais trabalhosa é a de arquivos.

Atualmente existem diversas áreas do conhecimento onde a necessidade de armazenamento de informação é muito grande, ou seja, o espaço requerido é enorme, na ordem mínima de Gbytes ( $2^{30}$ ) a até petabytes ( $2^{50}$ ).

Exemplos:

- **Projeto Genoma**
- **Dados de Satélite**
- **Cientes de banco**

### IV.2 HISTÓRIA

- Os primeiros projetos de estruturas de arquivos assumiam que os dados estariam em fitas: o acesso era seqüencial, e o custo do acesso era diretamente proporcional ao tamanho do arquivo.
- Entretanto, rapidamente os arquivos cresceram de modo inaceitável para o processamento através de acesso seqüencial.
- Dispositivos como discos e tambores apareceram, o que permitiu a associação de **índices** aos arquivos.
- Os índices permitem guardar uma lista de chaves/ponteiros em um arquivo menor, que podem ser pesquisados mais rapidamente. Dada a chave associada ao registro procurado, a busca no índice retorna um ponteiro para o registro no arquivo principal, e a informação pode, então, ser obtida diretamente.

- Índice - é um arquivo auxiliar que facilita a recuperação de informação no arquivo principal, que é o arquivo de dados. Índices simples resultam nos mesmos problemas quando crescem demais e tornam-se difíceis de manter, principalmente no caso de arquivos dinâmicos, nos quais as chaves mudam todo o tempo.
- No início dos anos 60 surgiu a ideia de usar árvores como solução para estruturar os índices. O problema é que as árvores tendem a crescer de maneira desigual à medida que as chaves são inseridas e removidas (ficam desbalanceadas).
- Em 1963 surgiram as árvores AVL, que são ótimas para estruturar dados em RAM. Pesquisadores logo pensaram em utilizar algo parecido para a estrutura de arquivos. Entretanto, mesmo árvores perfeitamente balanceadas exigem muitos acessos para localizar um dado registro. Era, portanto, necessário encontrar um mecanismo de armazenar em cada nó da árvore um bloco do arquivo contendo o maior número possível de registros.
- Mais de dez anos de trabalho foram necessários para chegar à solução, as chamadas *B-trees*, ou árvores-B. Um problema era que a metodologia requerida para arquivos era muito diferente da utilizada em RAM. Árvores AVL crescem de cima para baixo enquanto que árvores-B crescem de baixo para cima!
- As árvores-B permitem um excelente desempenho na busca por registros do arquivo, ao custo de não mais se poder acessar sequencialmente o arquivo de modo eficiente.
- A solução quase imediata foi encontrada com a adição de uma lista encadeada no nível inferior da árvore-B, nesse caso batizada de árvore-B+.
- Em termos práticos, árvores-B+ garantem o acesso a um registro mantido em um arquivo com milhões de entradas com apenas três ou quatro acessos ao disco!!!
- Por mais de dez anos essas árvores formaram a base para a maioria dos sistemas de arquivo comerciais...
- A capacidade de acessar um registro em 3 ou 4 acessos é um aspecto muito bom, mas o objetivo ideal era precisar de apenas 1 acesso....
- A técnica **Hashing** é uma boa maneira de atingir este objetivo se estivermos trabalhando com arquivos cujo tamanho não é alterado. Há muito tempo índices e *hashing* são combinados para garantir acesso eficiente a arquivos estáticos.
- A técnica conhecida por **Dynamic Hashing**, que pode ser aplicada a arquivos dinâmicos eficientemente, é relativamente recente.

Se fossemos pensar em uma linguagem (com limitação de memória):

**640 Kbytes de heap = 655.360**

**1 nó AVL = 2 apontadores (8 bytes) + inteiro:chave (2 bytes) + inteiro : FB (2 bytes)  
+ índice do arquivo (4 bytes) = 16 bytes**

**Conseguiríamos armazenar no máximo 40.960 nós**

Para entendermos melhor como funciona o mecanismo de trabalho com arquivos precisamos entender onde eles são armazenados e como são acessados.

#### IV.3 HIERARQUIA DE MEMÓRIA

- **Memória Cache**

O **cache** significa esconderijo

. Uma memória onde as instruções ou dados mais utilizados pelo processador ficam armazenados. Assim, quando o processador vai executar algo primeiro, vai ao cache, se não encontrar vai à memória ram.

O tempo de acesso ao cache é da ordem de  $10^{-8}$  enquanto que o acesso do processador à memória ram é da ordem de  $10^{-7}$  [GARCIA-MOLINA, ULLMAN e WIDOM, 2001].

Sua capacidade de armazenamento é da ordem de 1Mbyte.

Pode existir o cache onboard localizado internamente ao processador e o cache de nível 2, localizado em outro chip.

### ➤ **Memória principal**

Ordem a capacidade de armazenamento pode chegar, em torno de 10Gbytes [GARCIA-MOLINA, ULLMAN e WIDOM, 2001].

### ➤ **Memória Virtual**

Programa dividido em várias partes, onde cada parte cabe na memória principal. Uma parte é então levada para a memória principal e executada.

Atualmente os computadores de 32 bits conseguem endereçar até  $2^{32}$  endereços diferentes (cada byte possui um endereço). Estas partes ou blocos são conhecidos como páginas e deslocados inteiramente para a memória principal quando solicitados.

### ➤ **Memória secundária**

Discos magnéticos, discos rígidos, cd's, etc..

Muito maior que a memória principal

Mais barata (em 1999 - 5 a 10 centavos de dólar por Mbyte enquanto a principal em torno de 1 a 2 dólares por Mbyte)

Mais lenta [GARCIA-MOLINA, ULLMAN e WIDOM, 2001]:

⇒ **10 a 30 milissegundos para gravar ou ler**

⇒ 1 milhão de instruções são executadas pelo processador neste tempo

### ➤ **Memória Terciária**

Tempo de leitura e escrita muito alto.

Fitas (sequência s)

Jukeboxes de discos óticos

Silos de fitas

O acesso ao meio terciário pode ser cerca de 1000 vezes mais lento que a memória secundária mas sua capacidade pode ser cerca de 1000 vezes maior.

### ▣ Armazenamento Volátil x Não Volátil

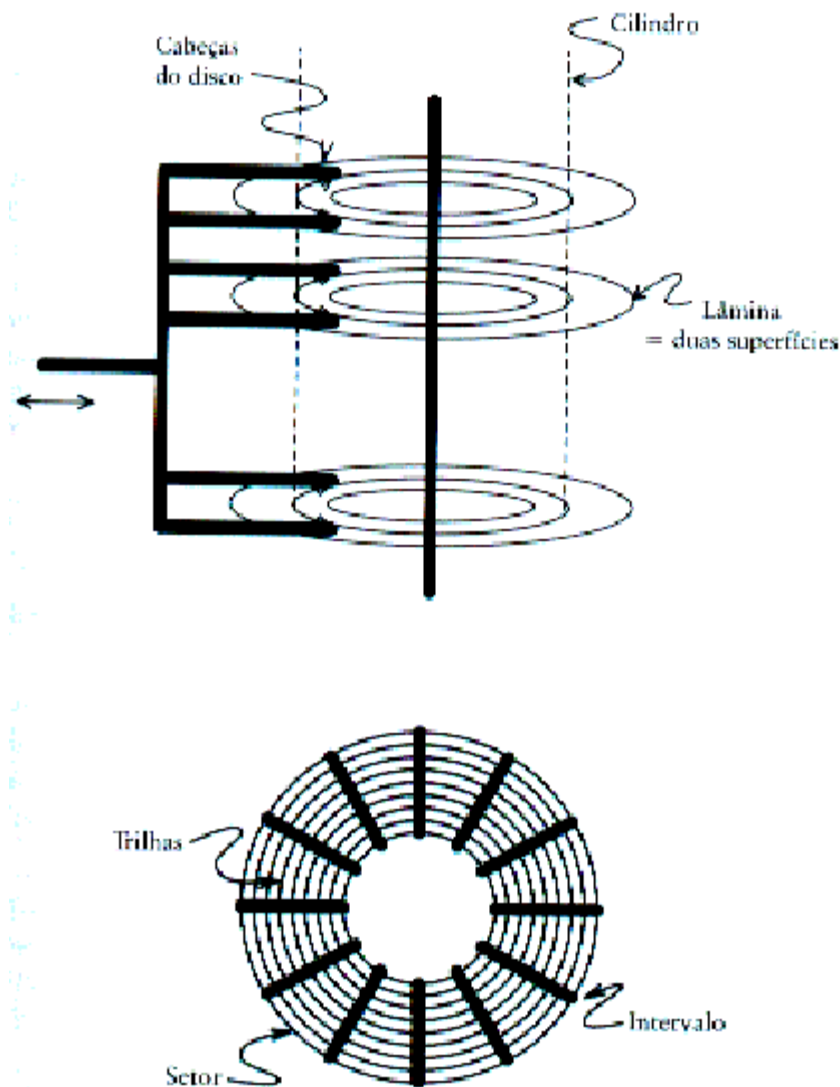
- Memória principal é volátil
- Novos chips de memória chamados de **memória flash** não voláteis.

### ▣ Entendendo melhor os DISCOS (HD)

#### ➤ Algumas características dos HD's

- ✓ Rotação – ex: 5400 rpm: 1 rotação a cada 11 milissegundos.
- ✓ N° de lâminas (ex. 5 a 15 discos – 30 superfícies)
- ✓ N° de trilhas (ex. 10.000 trilhas por superfície)
- ✓ N° de bytes por trilha (ex.  $10^5$ )
  - o 500 setores/trilha
  - o 512 a 4096 bytes

- Gravação ou leitura de dados no HD é feita por blocos.
- Controladores estão mais parecidos com os computadores, tendo processadores e memórias, podendo armazenar uma trilha inteira em sua memória.
- Modificação (alteração) envolve leitura, alteração e gravação-> processo + demorado.
- **Controladora de disco é responsável por:**
  - ✓ Acionamento mecânico
  - ✓ Seleciona uma superfície
  - ✓ Eixo giratório
  - ✓ Transferir os bits
- **Tempos:**
  - ✓ Posicionar na trilha correta: 12 a 14 milissegundos em pc's e 8 ou 9 em servidores.
  - ✓ Localizar e transferir um bloco do disco para a trilha: 12 a 60 milissegundos.



**Figura 1:** (a) Visão interna de um HD. (b) Superfície de um disco.  
 Fonte: [GARCIA-MOLINA, ULLMAN e WIDOM, 2001].

#### IV.4 O QUE É UM BLOCO?

É a unidade de transferência de dados entre o disco e a memória.

Um HD é constituído de discos, onde cada disco está dividido em trilhas e cada trilha está dividida em setores (ou blocos).

- **Setores** → **divisão física**
- **Blocos** → **divisão lógica**

O tamanho de cada bloco é definido pelo SO durante a formatação do disco e depois não pode ser alterado. Existem discos onde não existem setores. Quando se deseja um dado em arquivo, o SO ou gerenciador de BD não busca apenas o dado, o bloco inteiro onde o dado se encontra é levado para a memória primária. Este bloco fica no chamado buffer (área reservada contígua na memória principal). Um bloco é copiado para o buffer ou vice-versa. Quando vários blocos em sequência são transferidos, dá-se o nome de cluster (grupo de blocos).

Um bloco pode variar de 512 a 4096 bytes [ELMASRI e NAVATHE, 2002].

Em se tratando de um BD, um componente do SGBD (gerenciador de buffers) é responsável por particionar a memória principal em buffers. Isto deve ocorrer de tal modo que a quantidade de buffers caiba na memória principal.

### Definições importantes:

- **Organização spanned:** quando é permitido que um registro seja maior que um bloco, neste caso haverá ao final do bloco um ponteiro para o próximo bloco que contém o restante do registro.
- **Organização não-spanned:** quando não é permitido que um arquivo seja maior que o bloco.

## IV.5 ARQUIVOS

Trabalharemos aqui com arquivos que contém uma sequência de registros. Um arquivo é composto de registros de tamanhos fixos ou variáveis.

Exemplos de tamanho variável:

- Um campo nome pode ter um tamanho variável.
- Arquivos de registros variantes.
- E outros.

Um header de arquivo ou descritor de arquivo contém informações sobre o mesmo, como: endereço no disco dos blocos que compõe o arquivo, descrições de registros, etc.

Para pesquisar um registro em disco, por exemplo, um ou mais blocos são copiados para os buffers da memória principal. Os programas então pesquisam o registro ou os registros desejados dentro dos buffers, até que o elemento seja encontrado ou até que os blocos terminem.

**O objetivo** de uma boa organização de arquivos é localizar o bloco que contém um registro desejado com o número mínimo de transferências de blocos.

### IV.5.1 Arquivo Sequencial

Os registros são organizados somente baseando-se em suas localizações físicas sucessivas no arquivo e, são lidos e atualizados na ordem em que aparecem. Ou seja, para acessar o  $n$ -ésimo registro, é necessário acessar os  $(n - 1)$  anteriores.

Se o arquivo não estiver ordenado, as inclusões podem ser feitas no final do arquivo. Mas a inclusão em um arquivo ordenado ou exclusão (no arquivo ordenado ou não) pode implicar na necessidade de fazer uma nova cópia do arquivo. Já as atualizações, quando feitas em disco, não necessitam de nova cópia. Ao contrário do que acontece com arquivos em fita.

Este tipo de organização é utilizado em arquivos para aplicações de baixa atividade e pequena volatilidade.

### IV.5.2 Arquivo Sequencial-Indexado

São aqueles arquivos onde se utiliza uma estrutura de acesso auxiliar chamada de índice, que são utilizados para acelerar a recuperação de registros em resposta a determinadas condições de pesquisa. São projetados para aplicações que requerem acessos seqüencial e direto eficientes.

A entrada do índice é formada por **dois campos principais**: um campo **chave** que conterá o elemento a ser pesquisado e outro campo contendo a **posição** do registro no arquivo ou a posição do bloco que contém o arquivo. Essa entrada ocupa um espaço bem menor do que o registro de dados correspondente, e faz com que a área ocupada pelo índice seja menor do que aquela ocupada pelos dados, com isso a pesquisa sobre um índice pode ser feita com maior rapidez.

Quando o índice contém um registro para cada registro do arquivo ele é chamado de **índice denso**, quando o índice possui um registro que armazena uma entrada para um bloco do arquivo ele é chamado de índice **não denso** ou **esparso**.

Exemplos de organizações para este tipo de arquivo:

- Uso de AVL (já estudado no capítulo anterior) contendo a posição do registro no arquivo.
- Pesquisa Binária (vetor) contendo a posição do registro no arquivo.

Apresentaremos algumas organizações:

#### a. Índices Principais (AIP)

Índice principal é um arquivo ordenado cujos registros possuem dois campos: campo chave e um campo que armazena a posição do bloco onde aquela chave está no arquivo original. O campo chave contém a chave do primeiro registro de cada bloco do arquivo (**registro âncora**).

Para que o AIP seja criado, é necessário que o arquivo original também esteja ordenado.

Como pesquisar?

Como os registros no AIP estão ordenados, a localização de um registro por meio de um argumento de pesquisa passa a ser feita em duas etapas:

1. É pesquisa o índice através de pesquisa binária, descobrindo assim o bloco no qual deve estar o registro
2. O bloco selecionado é acessado e pesquisado, sendo localizado o registro desejado.

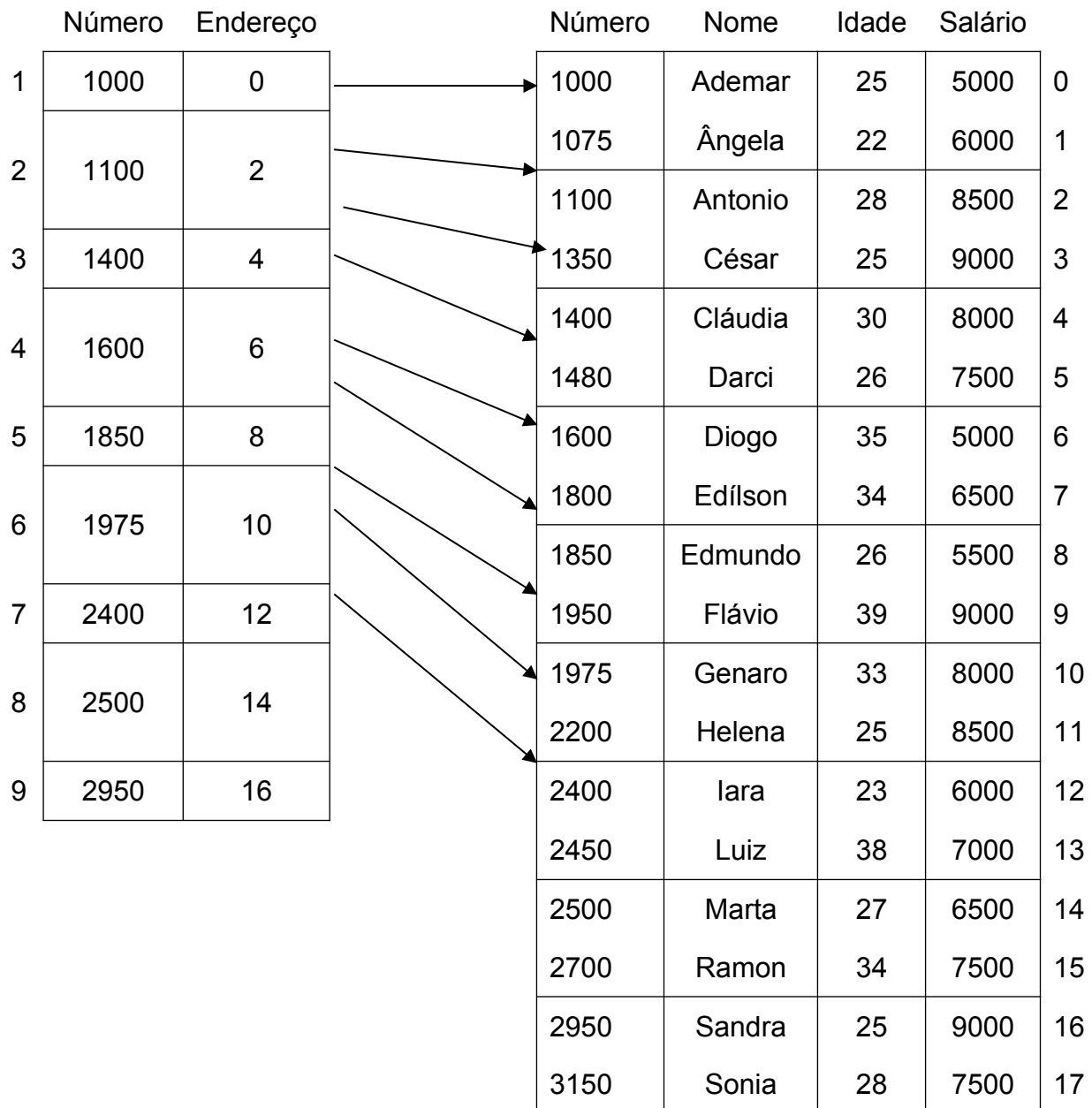
Então, a pesquisa terá custo  $C(n) = \log_2 n + 1$ , onde  $n$  representa o número de blocos do AIP.

Problemas:

- Arquivo original deve ser mantido ordenado, o que torna a inclusão ou exclusão de elementos bastante custosa.



Veja, agora, o diagrama de acesso:



**Figura 2:** Exemplo de índice principal (Diagrama de acesso)

### Exemplo:

1 – Seja um arquivo com 30000 registros em uma organização não-spanned, onde cada registro ocupa 100 bytes. Sabe-se que o armazenamento em memória secundária é feito através de blocos e, nesse caso, cada bloco comporta 2 Kbytes. Se fosse utilizado um arquivo de índices, cada registro (chave + endereço) ocuparia 15 bytes. Com base nesses dados, responda:

- Quantos acessos a disco, no máximo, seriam necessários se fosse feita uma pesquisa sequencial direto no arquivo original?
- Quantos acessos a disco, no máximo, seriam necessários se fosse feita uma pesquisa binária direto no arquivo original?
- Quantos acessos a disco, no máximo, seriam necessários se fosse feita uma pesquisa utilizando um AIP?

## b. Índices de Clustering

Esse tipo de índice é utilizado quando o arquivo original está ordenados por chave secundária, ou seja, a chave de pesquisa pode ser repetida. Um exemplo que podemos dar é quando se deseja pesquisar em um arquivo as pessoas que possuem salário igual a R\$ 1000,00. O campo salário é o de pesquisa e pode estar repetido nos diferentes registros do arquivo.

O Arquivo de Clustering será construído, pegando a primeira ocorrência de cada chave e armazenado seu endereço, ou seja, sua localização no arquivo original.

Exemplo:

Arquivo de Índice de Clustering  
**chave ponteiro**

1	
2	
3	
4	
5	
6	

Arquivo Original  
**chave Outros campos**

1	
1	
2	
2	
3	
4	
5	
5	
5	
6	
6	

**Figura 3:** Exemplo de índice de Clustering.

A pesquisa será feita (Pesquisa Binária) no arquivo de índice, descobrindo em que posição do arquivo original começa a aparecer tal chave.

## c. Índices Secundários (AIS)

Arquivo de Índice Secundário é um arquivo ordenado cujos registros possuem dois campos: campo chave e um campo que armazena a posição do registro que contém aquela chave no arquivo original.

Nesse modelo, o arquivo original estará desordenado, assim o campo chave do AIS contém cada uma das chaves da arquivo original, porém elas estarão ordenadas. Portanto, o AIS será um arquivo denso.

A pesquisa, de forma semelhante à só AIP, será feita em duas etapas:

1. Pesquisa o índice através de pesquisa binária, descobrindo assim sua posição no arquivo original
2. O registro completo é acessado diretamente, dentro do arquivo original.

Então, a pesquisa terá custo  $C(n) = \log_2 n + 1$ , onde  $n$  representa o número de blocos do AIS.

Exemplo:

Arquivo de Índice  
de Clustering  
**chave ponteiro**

1	8
2	3
3	6
4	7
5	0
6	9
7	2
8	4
9	5
10	1

Arquivo Original  
**chave Outros campos**

5	
10	
7	

2	
8	
9	

3	
4	
1	

6	

**Figura 4:** Exemplo de índice secundário.

**Exemplo:**

2 – Com base nos dados do Exercício (1), responda quantos acessos a disco, no máximo, seriam necessários se fosse feita uma pesquisa utilizando um AIS.

Como implementar um arquivo de índice secundário para um arquivo não ordenado, cujo campo a ser considerado para a pesquisa, for uma chave secundária?

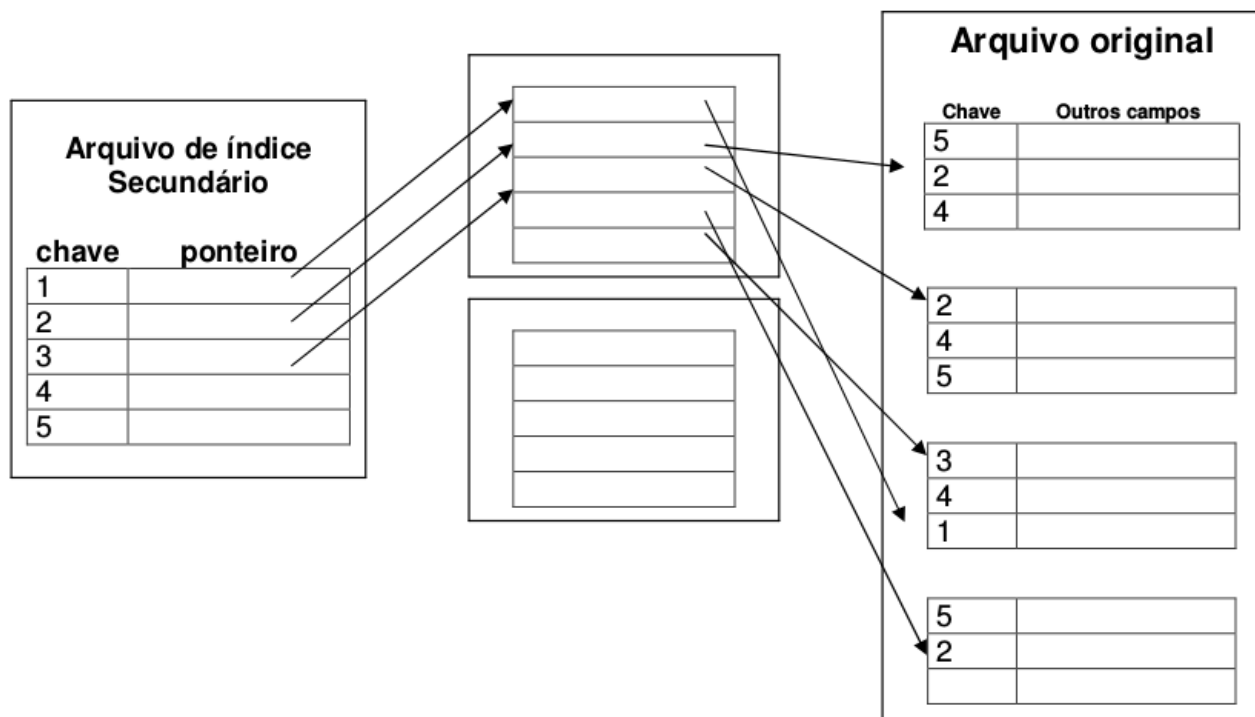


Figura 5: Exemplo de índice secundário para campos repetidos.

#### d. Índices Multinível

Nesta organização utiliza-se mais de um arquivo de índice. A ideia aqui é diminuir o número de acessos a bloco que, na pesquisa binária acessa, em média, cerca de  $O(\log_2 n)$  blocos.

Assim, tem-se a seguinte estrutura, a partir de um arquivo original desordenado:

- 1º nível (Arquivo Original)
- 2º nível (Arquivo Índice Secundário)
- 3º nível em diante (Arquivo Índice Principal)

#### Exemplo:

3 – Com base nos dados do Exercício (1), responda quantos acessos a disco, no máximo, seriam necessários se fosse feita uma pesquisa utilizando um arquivo multinível, sabendo que o arquivo original está desordenado.

- 1º nível (Arquivo Original) → 3.000 blocos (registros desordenados)
- 2º nível (Arquivo Índice Secundário) →  $3000/68 = 442$  blocos
- 3º nível (Arquivo Índice Principal) →  $442/68 = 7$  blocos
- 4º nível (Arquivo Índice Principal) →  $7/68 = 1$  bloco
- Teríamos 4 níveis e, portanto, 4 acessos

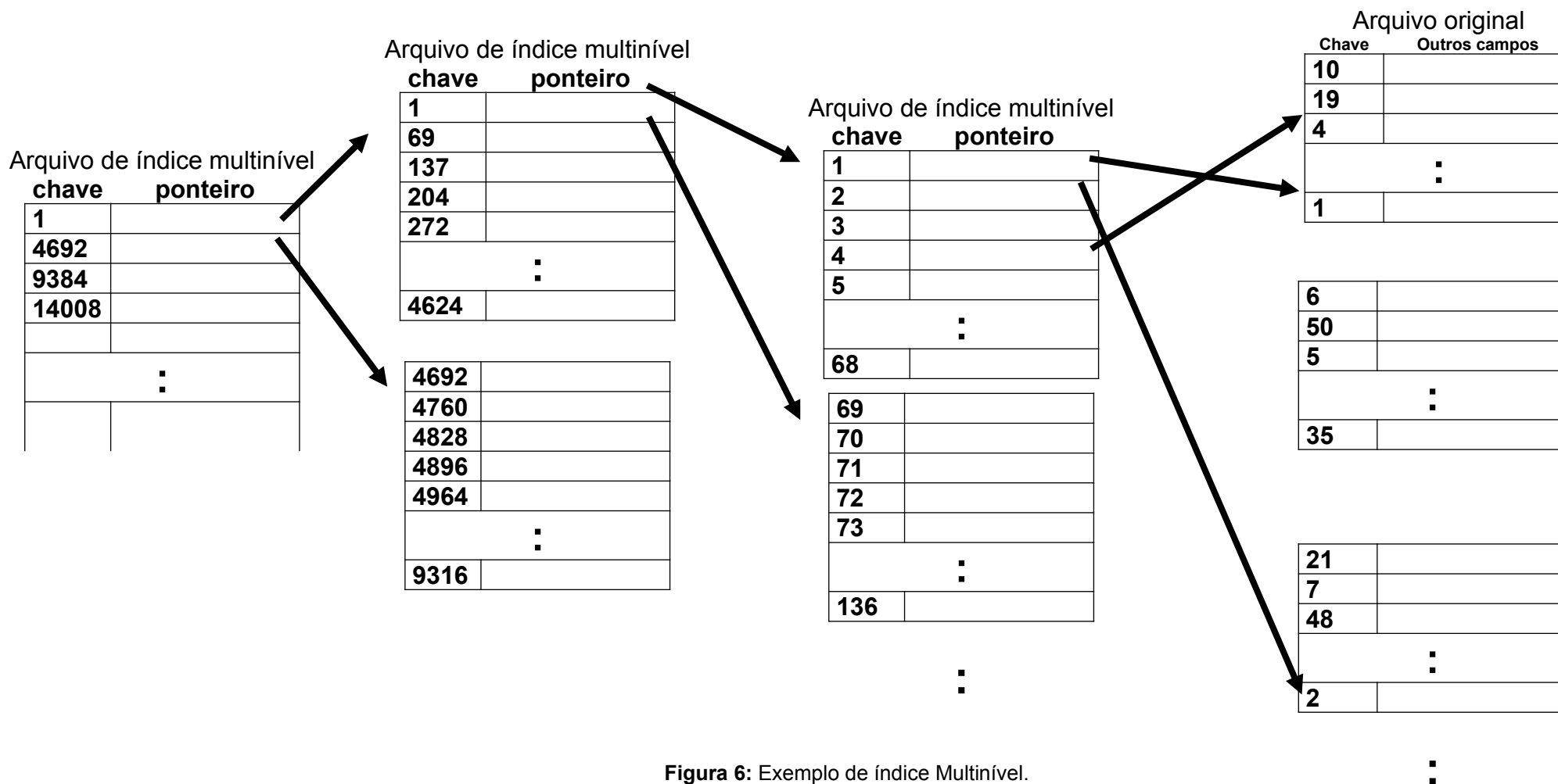


Figura 6: Exemplo de índice Multinível.

**Bibliografia:**

ELMASRI, Ramez, NAVATHE, Sahmkant B..Sistemas de Banco de Dados – Fundamentos e Aplicações. Tradução: Tereza C. P. de Souza. Rio de Janeiro: LTC – Livros Técnicos e Científicos, 2002. 837 pg.

GARCIA-MOLINA, Hector, ULLMAN, Jeffrey D., WIDOM, Jennifer. Implementação de Sistemas de Banco de Dados. Tradução: Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001. 685 pg.

FERRAZ, Inhaúma Neves. Programação com Arquivos. Barueri, SP: Manole, 2003. 443 pg.

**Referências Bibliográficas:**

ELMASRI, Ramez, NAVATHE, Sahmkant B..Sistemas de Banco de Dados – Fundamentos e Aplicações. Tradução: Tereza C. P. de Souza. Rio de Janeiro: LTC – Livros Técnicos e Científicos, 2002. 837 pg.

GARCIA-MOLINA, Hector, ULLMAN, Jeffrey D., WIDOM, Jennifer. Implementação de Sistemas de Banco de Dados. Tradução: Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001. 685 pg.

## IV.6 ÁRVORES B

São árvores de pesquisa balanceadas especialmente projetadas para a pesquisa de informação em discos magnéticos e outros meios de armazenamento secundário.

- Minimizam o número de operações de movimentação de dados (escrita/leitura) numa pesquisa ou alteração.
- O grau de um nó pode ser alto.
- Podem ser consideradas como uma generalização natural das árvores de pesquisa binárias.

### DEFINIÇÃO:

O número máximo de ponteiros que podem ser armazenados em um nó é a **ordem** da árvore, ou seja, a quantidade de filhos de um nó.

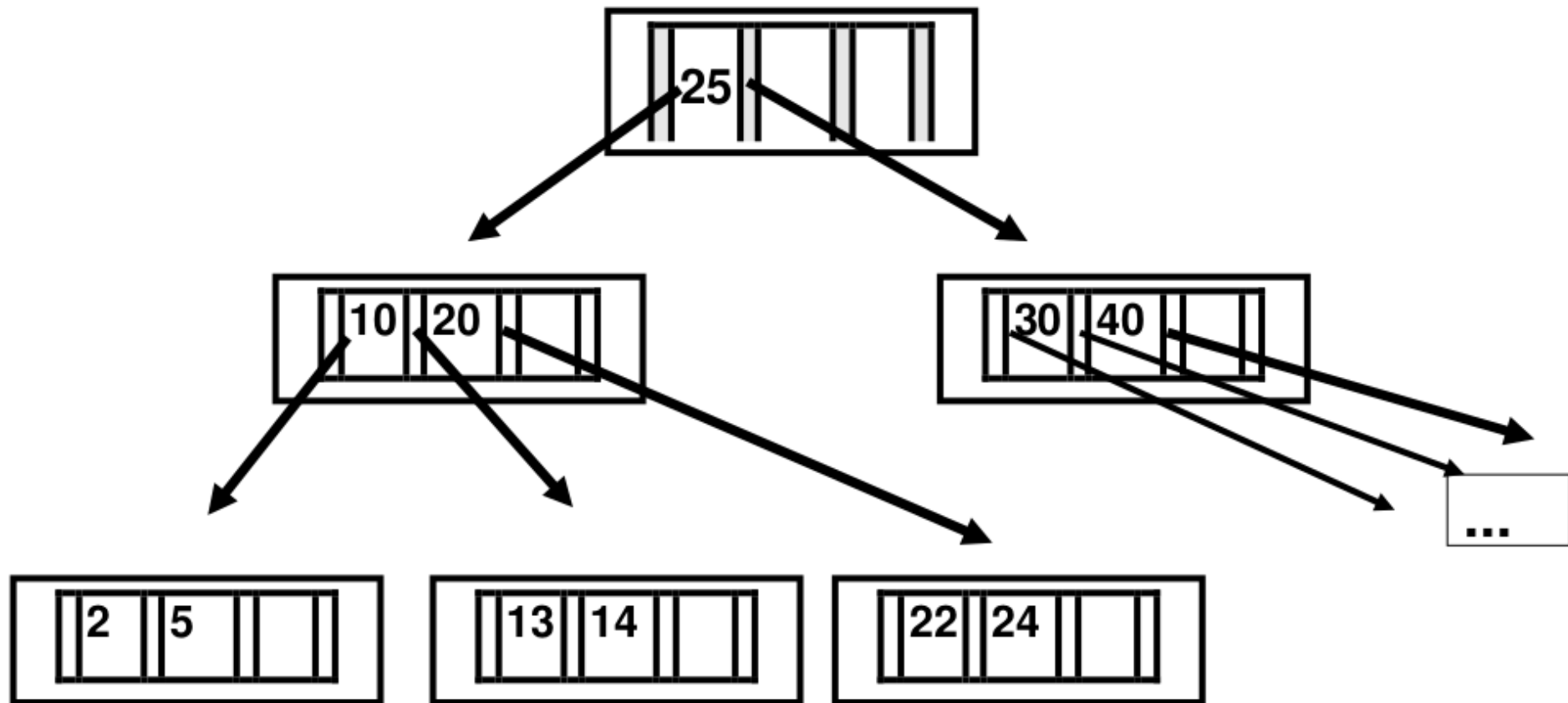
Uma árvore-B de ordem  $m$  é uma árvore que satisfaz às seguintes condições:

1. Cada nó possui  $m$  ou menos sub-árvores (descendentes, filhos). No máximo  $m$ .
2. Todo nó, exceto a raiz, possui  $\lceil m/2 \rceil$  ou mais sub-árvores. No mínimo  $\lceil m/2 \rceil$   
 $\lceil m/2 \rceil \Rightarrow$  menor inteiro maior ou igual a  $m/2$   
 se  $m = 6 \Rightarrow$  mínimo 3. Se  $m = 5 \Rightarrow$  mínimo 3.  
 No caso de nós filhos, estes possuem no mínimo  $(m/2)-1$  componentes nos nós.
3. A raiz possui, no mínimo 2 sub-árvores não vazias, exceto no caso em que é uma folha;
4. todas as folhas estão no mesmo nível e todas as suas sub-árvores são vazias;
5. um nó com  $k$  sub-árvores armazena  $k-1$  chaves ou registros;
6. todos nós de derivação (aqueles que não são folhas) possuem exclusivamente subárvores não vazias.
7. As chaves estão em ordem crescente.

Exemplo:

Árvore de ordem  $m=4$ .

- Cada nó possui no mínimo 2 subárvores e no máximo 4, com exceção da raiz e folhas.
- Cada nó folha possui no mínimo  $(m/2)-1$  elementos.





**Exemplo dado em sala:**

Inserir os elementos em uma árvore B vazia, com  $m=5$ : 20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5, 42, 13, 46, 27, 8, 32, 38, 24, 45, 25.

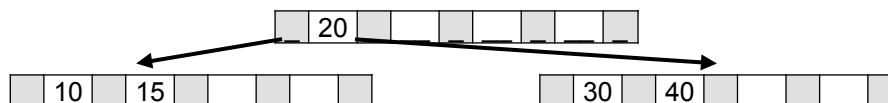
Neste caso cada nó armazenará 4 chaves e 5 apontadores.

Cada nó terá no mínimo 2 chaves (com exceção da raiz). Cada nó não folha terá no mínimo 3 subárvores.

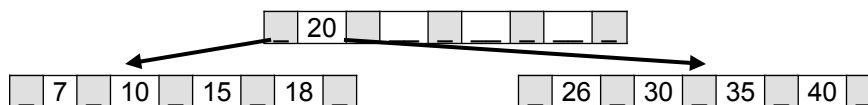
1) Inserir 20, 40, 10, 30:



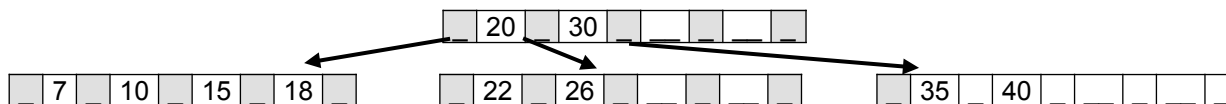
2) Inserir 15: O número 15 deveria ser inserido entre o 10 e o 20 mas não há espaço no nó. Este nó será então quebrado em 2 gerando uma árvore com uma raiz e duas subárvores. A pergunta é: quem será a raiz? Será o elemento central de: 10 15 20 30 40, neste caso é o 20.



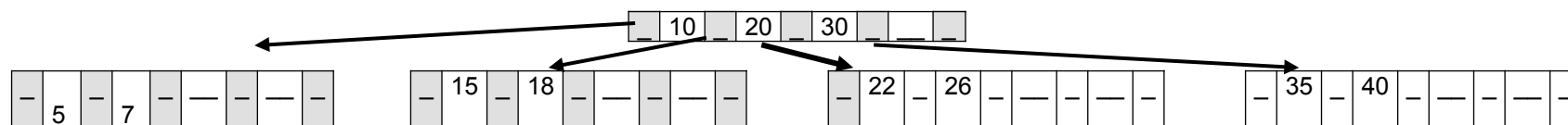
3) Inserir 35, 7, 26, 18, inserção normal:



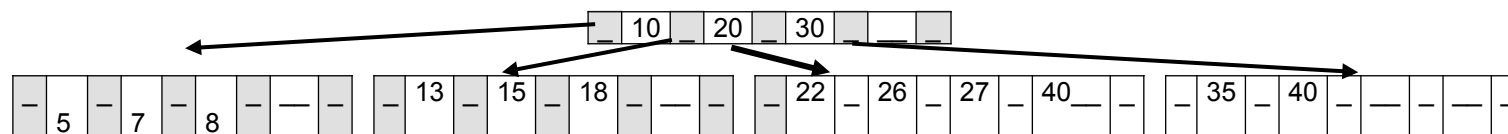
4) Inserir 22: O 22 deveria ser inserido antes do 26 mas não existe espaço no nó. Este nó será quebrado em 2, gerando outro nó. O elemento que subirá para a raiz será o número central entre: 22 26 30 35 40, neste caso é o 30:



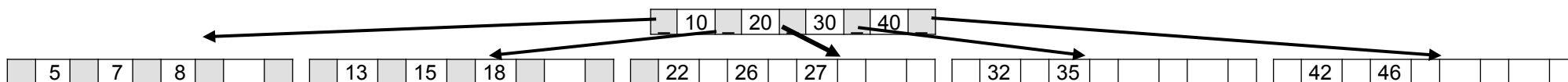
5) Inserir 5: A inserção do 5 segue o mesmo funcionamento do 22:



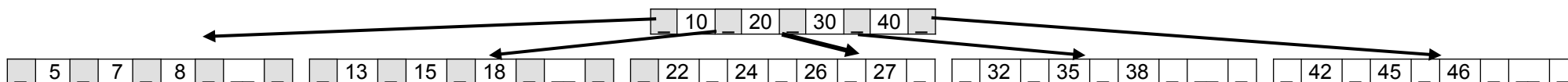
6) Inserir 42, 13, 46, 27, 8



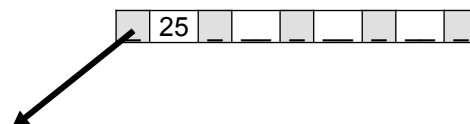
7) Inserir 32:

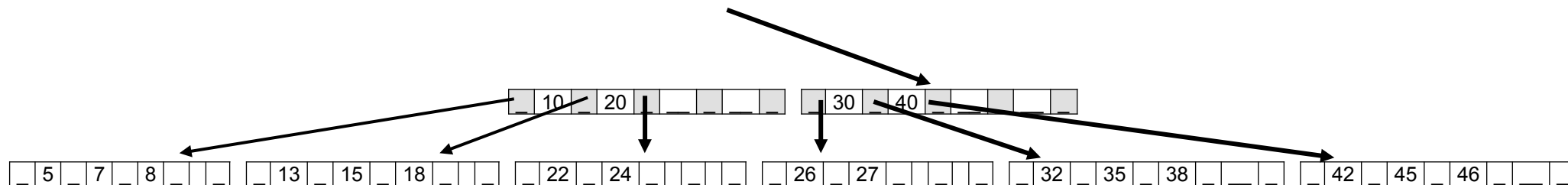


8) Inserir 38, 24, 45:



9) Inserir 25: Para inserir o 25 o nó deverá ser quebrado. O elemento central que deverá subir será: 22 24 25 26 27, o próprio 25. Para inserir o 25 no nó superior não existe espaço, ou seja, este nó deverá ser quebrado também para inserir o 25. O elemento que deverá subir: 10 20 25 30 40, será novamente o 25. Então teremos:



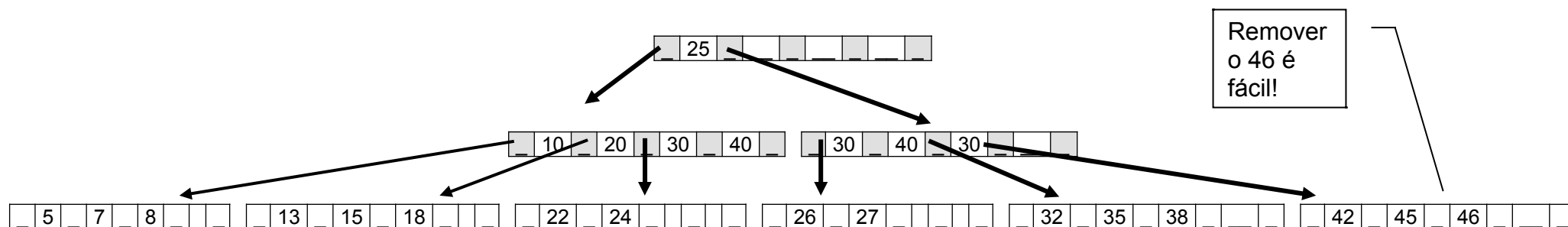


**Exercício: 1** – Insira os elementos abaixo em uma árvore B vazia de  $m=5$ :

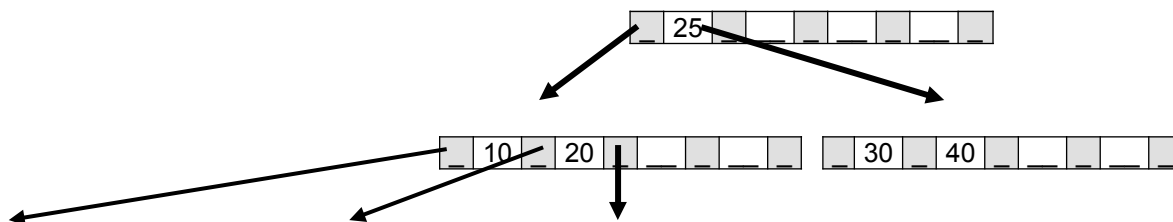
20, 10, 40, 50, 30, 55, 3, 11, 4, 28, 36, 33, 52, 17, 25, 13, 45, 9, 43, 8, 48.

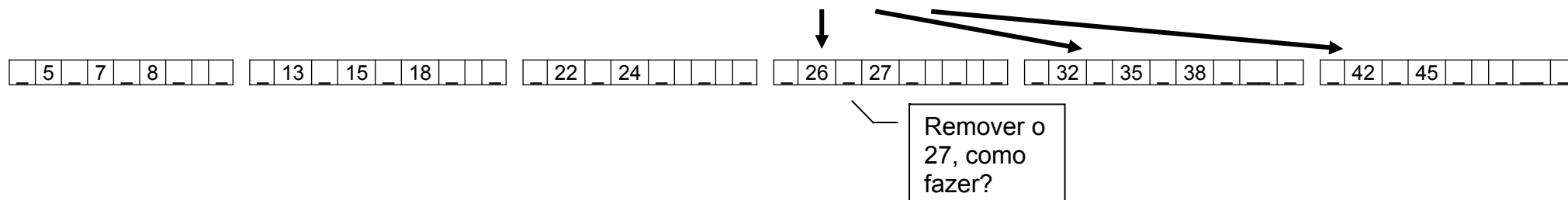
**O processo de remoção é um pouco mais complexo que o de inserção:**

- Remover um elemento de um nó folha com quantidade de elementos  $> (m/2)-1$  é fácil, basta eliminar o elemento mantendo o nó ordenado.



- Remover um elemento de um nó folha com quantidade de elementos  $= (m/2)-1$ , envolve alguns passos.

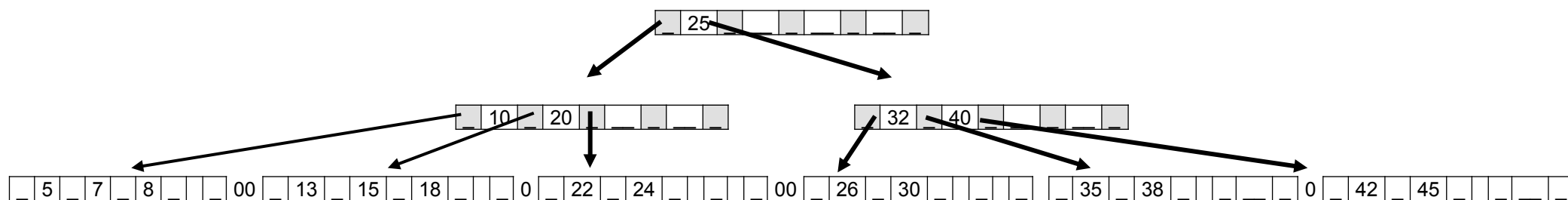




r.1) O nó da direita ou nó da esquerda possui um número de elementos maior que  $(m/2)-1$ .

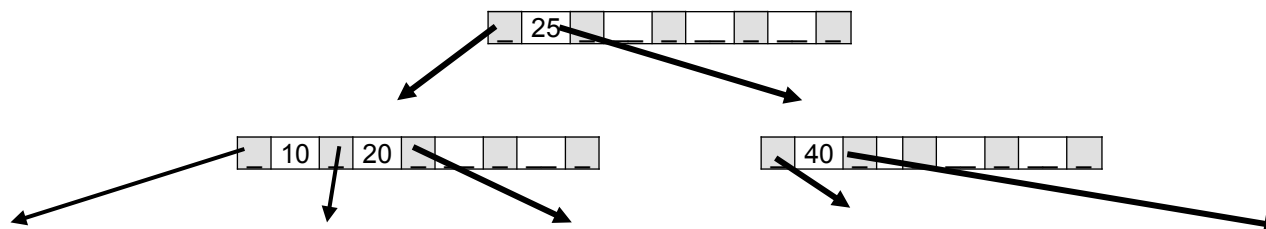
Neste caso se redistribui os elementos pertencentes à raiz do nó onde ocorrerá a remoção (nó esquerdo e direito da raiz).

Neste caso vamos eliminar o 27, deveremos redistribuir: 26 30 32 35 38. O 32 é o elemento central, será a nova raiz:



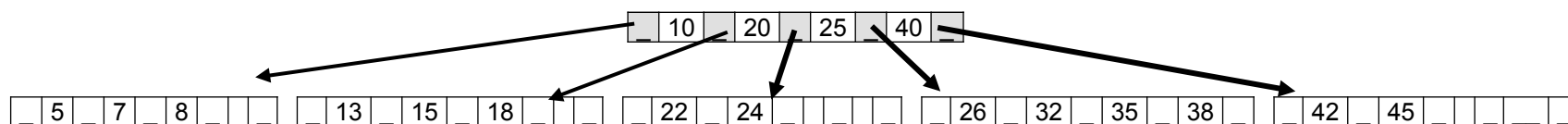
r.2) Uma outra situação é quando o nó da direita ou da esquerda não possui um número maior que  $(m/2)-1$  ou a redistribuição causará em uma árvore não B. Neste caso haverá uma junção dos nós. Um exemplo é a eliminação do 30.

Neste caso o nó da direita possui 2 elementos, e assim não poderemos tirar um visto que a quantidade ficará menor que 2. A solução então é ***juntar os dois nós mais o elemento da raiz***. Ficaria com um nó contendo 26 32 35 38.



[ ] 5 [ ] 7 [ ] 8 [ ] [ ] [ ] 00 [ ] 13 [ ] 15 [ ] 18 [ ] [ ] [ ] 0 [ ] 22 [ ] 24 [ ] [ ] [ ] [ ] 00 [ ] 26 [ ] 32 [ ] 35 [ ] 38 [ ] [ ] 42 [ ] 45 [ ] [ ] [ ] [ ] [ ]

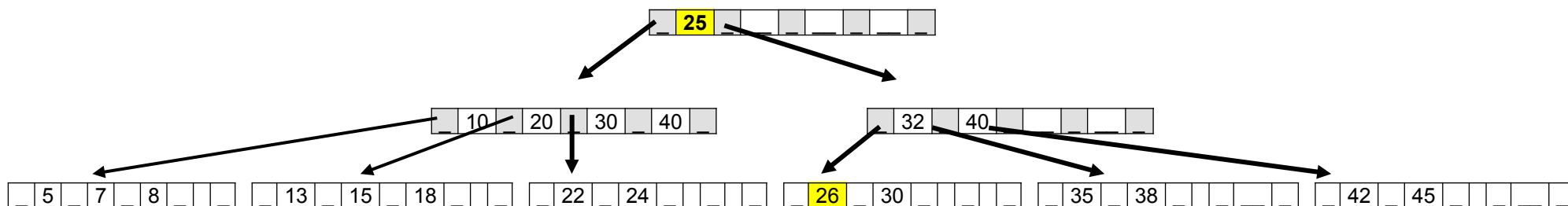
Caso a junção mantenha as restrições da árvore B, tudo bem. Caso não mantenha, como é a situação acima, o processo deve ser propagado para o nó superior, ou seja, os nós do 2º nível serão concatenados mais a raiz.



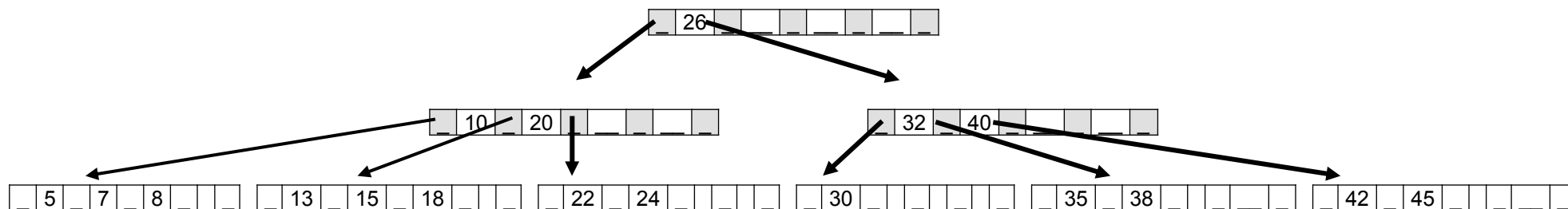
- Remover um elemento da raiz da árvore: caso a raiz seja única, basta remove-lo. Caso a raiz tenha filhos deveremos substituir a raiz pelo elemento mais a esquerda da direita ou mais a direita da esquerda. Se o nó folha onde foi removido o elemento cair em um dos casos citados acima, é só aplicar uma das técnicas mencionadas.

Na remoção do 25, troca-se pelo mais a esquerda da direita:

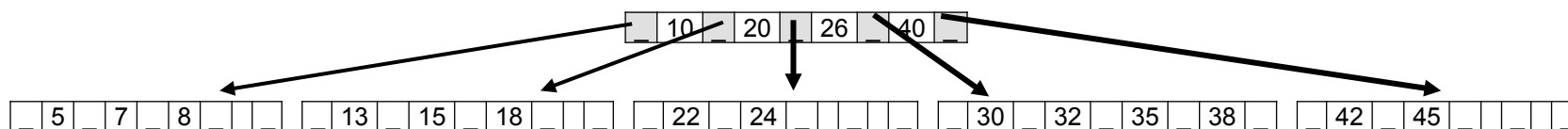
Antes:



Depois:

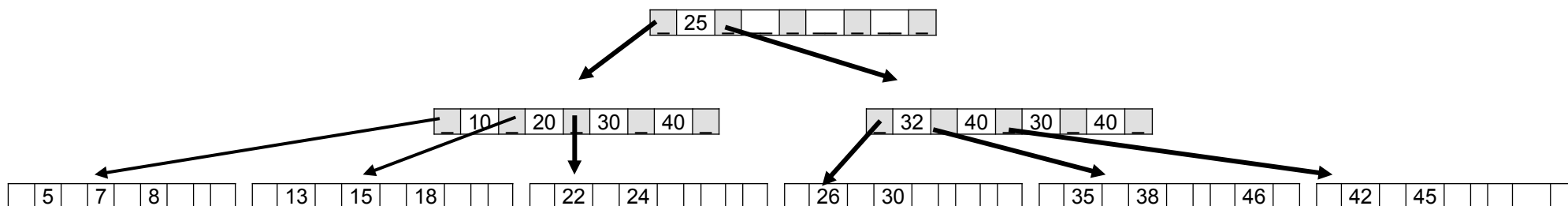


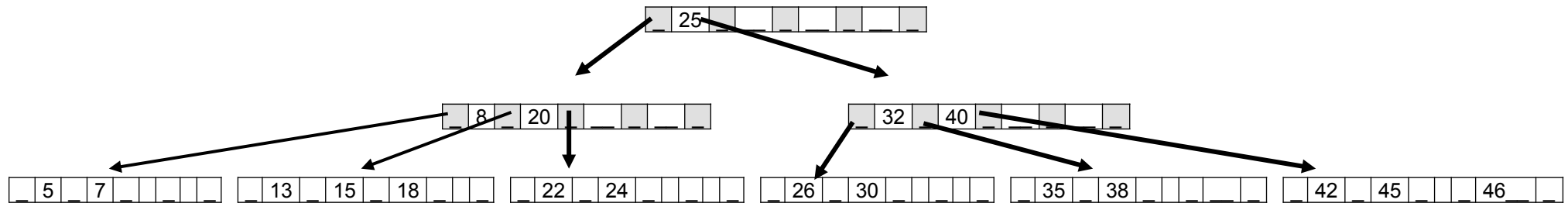
Neste caso a árvore deixou de ser B, e então caiu na situação de nó folha. Como não o nó da direita não pode ceder elementos, haverá uma junção: 30 32 35 38, que deixará a árvore fora da definição também e então deve juntar o nós do nível intermediário.



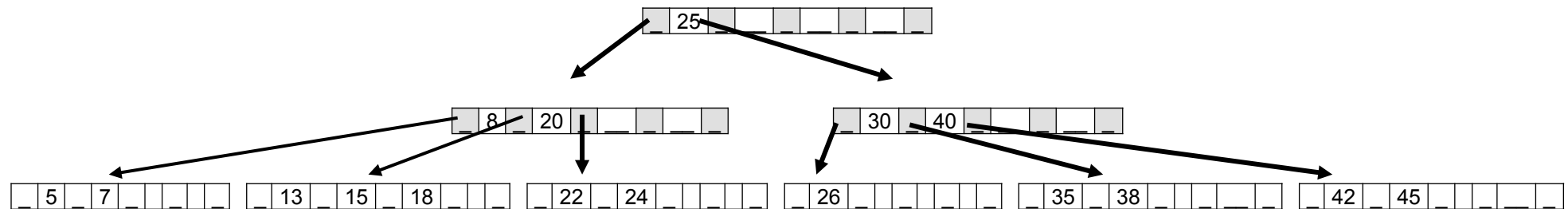
- Remover um elemento de um nó intermediário (não é raiz e nem folha).

Na remoção do 10, por exemplo, pegaria o elemento mais a direita da esquerda ou o elemento mais a esquerda da direita. Neste caso vou pegar o elemento mais a direita da esquerda.



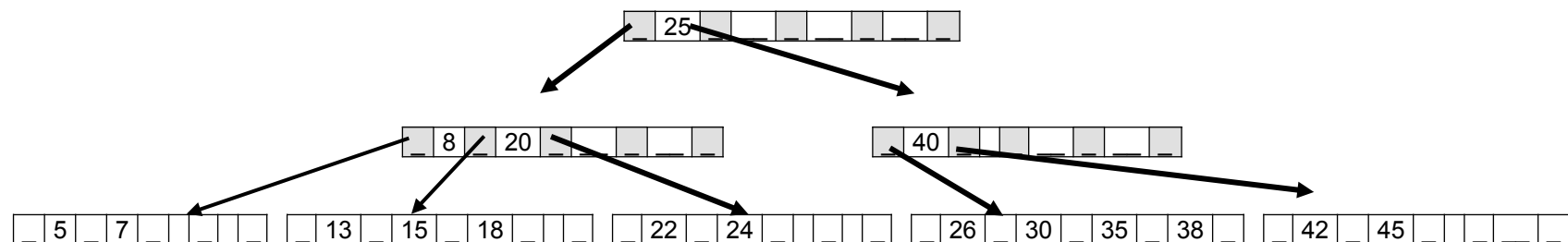


Vamos supor agora que desejo remover o 32. Pego o elemento mais a direita da esquerda:

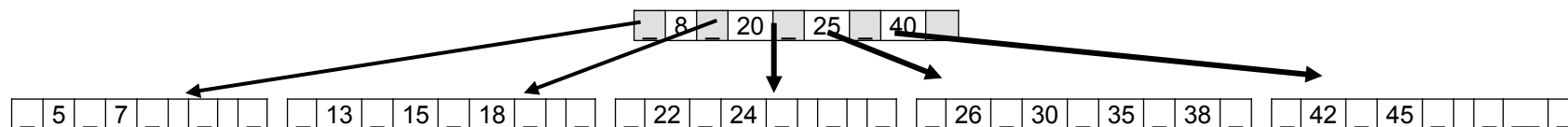


Minha árvore deixa de ser B, e cai em um dos casos de nó folha:

- Primeiro vou juntar, pois o nó a direita possui uma quantidade menor que  $(m/2)-1$ .



Neste caso, a árvore deixou de ser B, então, juntou os nós intermediários:





Resumindo remoção:

Nó folha	Caso simples: apenas remover a inda fica uma árvore B.
	Caso complexo 1: o nó vizinho possui uma quantidade maior que $(m/2)-1$ elementos: só redistribui.
	Caso complexo 2: o nó vizinho não possui exatamente $(m/2)-1$ elementos: <ul style="list-style-type: none"> <li>○ Juntar o nó do elemento removido mais o nó vizinho mais a raiz.</li> <li>○ Se a árvore deixou de ser B, junto os nós superiores (nós superiores + raiz – caso 2) ou redistribuo (caso 1).</li> </ul>
Nó intermediário ou raiz	Substituo o elemento pela chave mais a direita da esquerda ou pela chave mais a esquerda da direita. Se a árvore deixar de ser B, resolvo com uma das soluções de Nó folha.

## IV.7 ÁRVORES B+

Árvore B+ são semelhantes às árvores B com exceção do nível onde estão as folhas.

O nível onde estão as folhas é composto por uma lista encadeada.

O nós intermediários armazenam apenas o campo de pesquisa, os apontadores para os blocos onde estão as informações completas ficam nas folhas.

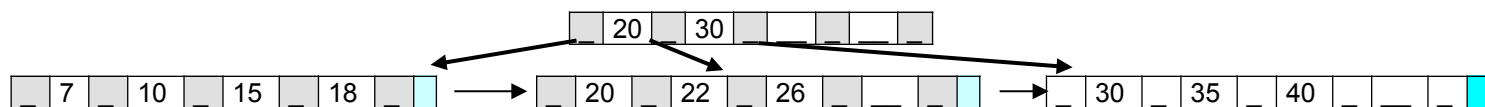
Esta árvore é indicada quando o item de maior impacto for a pesquisa, ou seja, quando ocorre pouca inserção ou remoção.

A facilidade desta árvore sobre a árvore B diz respeito à necessidade de exibir o arquivo ordenado. Neste caso a árvore não precisa ser percorrida, apenas a lista de nós folha.

As árvores B+ também facilitam o acesso concorrente ao arquivo.

Como não há necessidade de usar apontadores nos nós folha, é possível utilizar este espaço para armazenar uma quantidade maior de registros em cada nó folha.

Exemplo de uma árvore B:



A operação de inserção de um registro em uma ÁRVORE B+ é semelhante à de uma Árvore B. A única diferença é que quando uma folha é dividida em duas, o algoritmo faz uma cópia da chave que pertence ao registro do meio para a página pai no nível anterior, restando o registro do meio no nó folha da direita.

A remoção é mais simples, pois o registro a ser retirado está sempre no nó folha, mesmo que uma cópia da chave do registro a ser retirado esteja no índice.

## IV.8 TABELAS HASHING PARA ARMAZENAMENTO SECUNDÁRIO

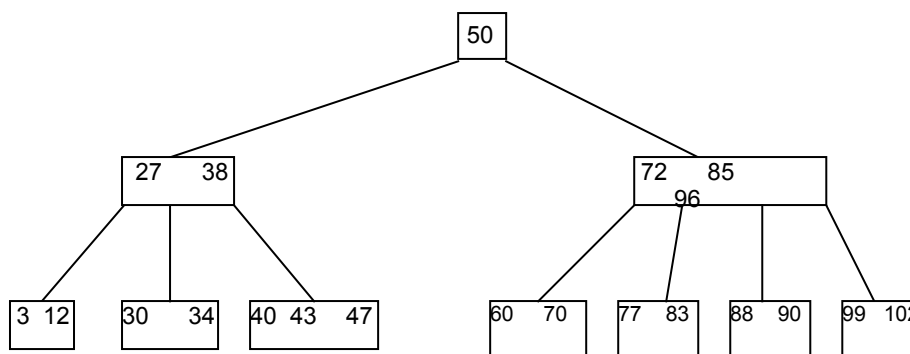
Neste caso podemos ter uma função que transforme uma chave em endereço de bloco onde pode ser armazenado o dado.

Esta forma é uma das mais eficientes, pois no melhor caso necessitaria de apenas 1 acesso para pesquisar ou para inserir.

O problema seria quando o bloco já estiver cheio e não se puder mais inserir elementos nele. Neste caso poderíamos ter um apontador para um bloco de estouro (transbordo), piorando o tempo de pesquisa, ou aplicarmos outra técnica.

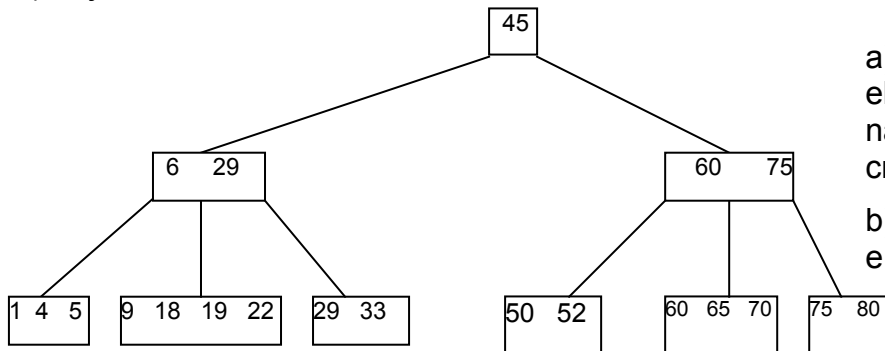
## IV.9 LISTA DE EXERCÍCIOS

- 1) Defina uma árvore B.
- 2) Por que, para uma árvore B, não é permitido um grau mínimo 1?
- 3) Se a árvore abaixo for uma árvore B legal, quais ordens ela pode ter?



- 4) Explique como procurar a menor chave de uma árvore B.
- 5) Insira os dados abaixo em uma árvore B, inicialmente vazia, de ordem 4: 0, 10, 20, 30, 1, 11, 21, 31, 2, 12, 22, 32, 3, 13, 23, 33, 4, 14, 24, 34, 5, 15, 25, 35, 6, 16, 26, 36, 7, 17, 27, 37, 8, 18, 28, 38, 9, 19, 29, 39.
- 6) Remova da árvore anterior, todos os múltiplos de 3.
- 7) Qual a diferença entre uma árvore B e uma árvore B+?

8) Seja a árvore B+ de ordem 5 abaixo:

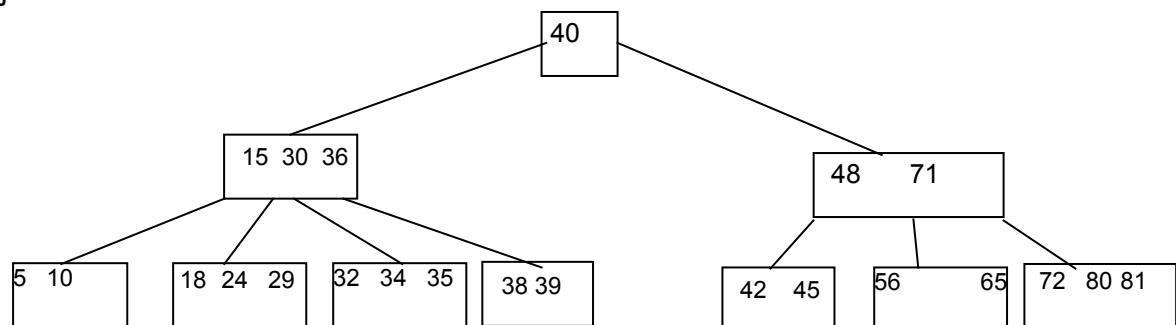


a) Escreva as chaves dos elementos que se encontram na árvore, em ordem crescente.

b) Retire os registros 5, 19, 22 e 60

A seguir, retire o registro 9.

9) Seja a árvore B abaixo:

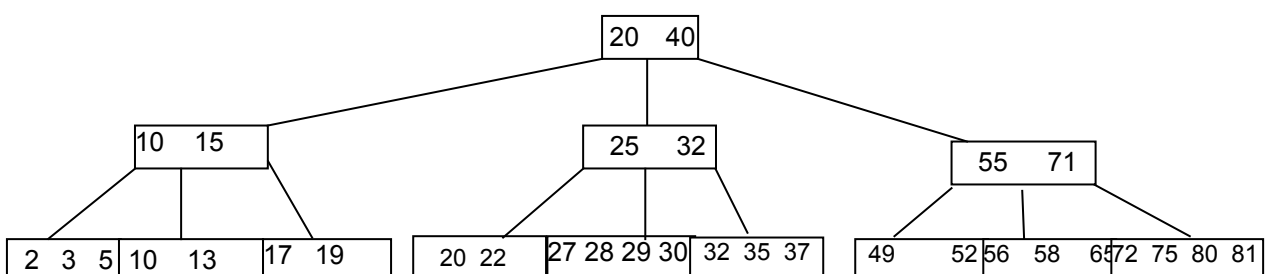
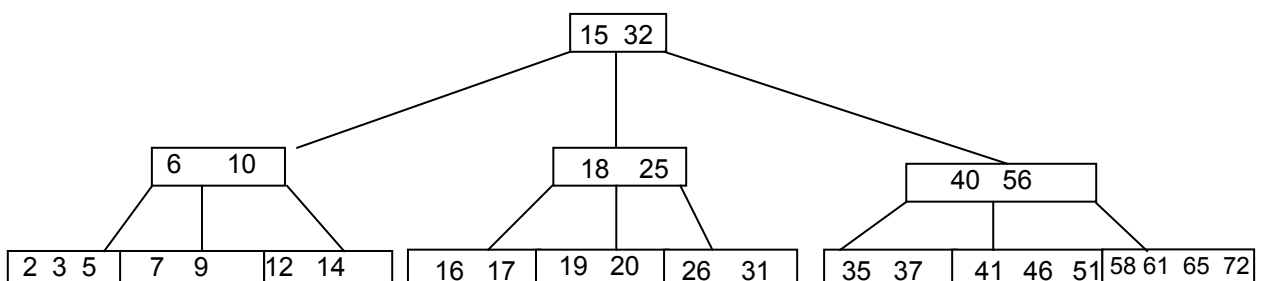


a) Quais as possíveis ordens que esta árvore pode ter?

b) Supondo que ela tenha ordem 5, retire o registro de chave 48.

c) Insira os registros de chave 19, 20, 31 e 33 nesta ordem.

10) Dadas as árvores abaixo, ambas ordem 5, identifique qual delas é B e qual é B+, justificando, retire o elemento 20 da árvore B+ e coloque o elemento 82 na árvore B, nesta ordem.



11) Suponha um arquivo com 100.000 registros em uma organização não-spanned, onde cada registro ocupa 420 bytes. Sabemos que o armazenamento em memória secundária seja feito através de blocos. No nosso caso, cada bloco comporta 4 Kbytes. Se fosse utilizado um arquivo de índices, cada registro (chave + endereço) ocuparia 36

bytes. Com base nesses dados, calcule quantos acessos a disco cada pesquisa abaixo faria:

- a) Pesquisa direto no arquivo original se ele estiver desordenado
- b) Pesquisa direto no arquivo original se ele estiver ordenado
- c) Pesquisa no arquivo de índice principal
- d) Pesquisa no arquivo de índice secundário
- e) Pesquisa usando o método multi-nível e tomando como base o arquivo original desordenado
- f) Pesquisa usando o método multi-nível e tomando como base o arquivo original ordenado

12) Suponha um arquivo com 300.000 registros em uma organização não-spanned, onde cada registro ocupa 150 bytes. Sabemos que o armazenamento em memória secundária é feito através de blocos. No nosso caso, cada bloco comporta 2 Kbytes. Se fosse utilizado um arquivo de índices, cada registro (chave + endereço) ocuparia 20 bytes. Com base nesses dados, calcule quantos acessos a disco cada pesquisa abaixo faria:

- a) Pesquisa direto no arquivo original se ele estiver desordenado
- b) Pesquisa direto no arquivo original se ele estiver ordenado
- c) Pesquisa no arquivo de índice principal
- d) Pesquisa no arquivo de índice secundário
- e) Pesquisa usando o método multi-nível e tomando como base o arquivo original desordenado
- f) Pesquisa usando o método multi-nível e tomando como base o arquivo original ordenado

13) Suponha um arquivo com 25.000 registros em uma organização não-spanned, organizado de forma não ordenada, onde cada registro ocupa 120 bytes. Sabemos que o armazenamento em memória secundária é feito através de blocos. No nosso caso, cada bloco comporta 1024 bytes. Responda:

- a) Quantos registros haverá por bloco?
- b) Quantos blocos este arquivo utiliza?
- c) Se fosse feita uma pesquisa sequencial no arquivo, quantos acessos, em média, seriam necessários?

Supondo que fosse feito um arquivo de índices multi-nível, onde cada registro, chave + ponteiro, utilizasse 15 bytes, responda:

- d) Quantos registros haverá por bloco, em cada nível?
- e) Quantos blocos são utilizados em cada nível?

- f) Se fosse feita uma pesquisa, quantos acessos seriam necessários para acessar o elemento pesquisado?

14) Suponha um arquivo com 70.000 registros não-spanned, organizado de forma não ordenada, onde cada registro ocupa 120 bytes. Sabemos que o armazenamento em memória secundária é feito através de blocos. No nosso caso, cada bloco comporta 2 Kbytes. Responda:

- a) Quantos registros haverá por bloco?
- b) Quantos blocos este arquivo utiliza?
- c) Se fosse feita uma pesquisa sequencial no arquivo, quantos acessos, em média, seriam necessários?

Supondo que fosse feito um arquivo de índices multi-nível, onde cada registro, chave + ponteiro, utilizasse 10 bytes, responda:

- d) Quantos registros haverá por bloco, em cada nível?
- e) Quantos blocos são utilizados em cada nível?
- f) Se fosse feita uma pesquisa, quantos acessos seriam necessários para acessar o elemento pesquisado?