



# *Procura Externa*

---

Algoritmos e Estruturas de Dados  
Inverno 2006

Cátia Vaz



# Procura Externa: *B-trees*

---

- As estruturas de dados também são úteis quando lidamos com dispositivos de armazenamento externo, tais como discos rígidos.
- A troca de dados entre a memória externa e a memória principal é muito dispendiosa.
  - Esta troca é normalmente realizada através de blocos de tamanho igual designados por **páginas**.
- Para estruturas de dados grandes que não possam ser armazenadas na memória principal, pretende-se minimizar estas trocas.



# Procura Externa: *B-tree*

---

- As árvores até agora estudadas apenas contém um item por nó.
- Ler uma página para a memória do disco é dispendioso.
- O acesso à informação numa página em memória é feito em tempo constante.
  - Considerar que cada nó contém mais do que um item (ex: cada nó conter a informação de cada página que é trocada entre um dispositivo de armazenamento externo e a memória principal).
- **Objectivo:** minimizar o número de acessos a páginas.
  - O nó contém  $M$  items = tamanho da página



# B-tree

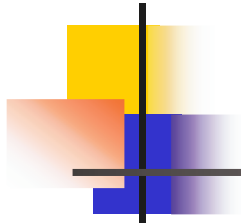
Definição: Uma **B-tree** de ordem  $M$  é uma árvore que ou está vazia ou é composta por  $k$ -nós, com  $k-1$  chaves e  $k$  links a árvores, representando cada um dos  $k$  intervalos delimitados pelas chaves. Uma B-tree tem as seguintes propriedades de estrutura:

- $k$  tem de ser entre 2 e  $M$  na raiz e entre  $M/2$  e  $M$  em cada outro nó;
- todos os links para árvores vazias têm de estar à mesma distância da raiz.

Nota: os algoritmos para estas árvores são construídos de acordo com estas abstracções básicas.

**Nota:** Na implementação, irá ser escolhida uma representação concreta. Nomeadamente, o algoritmo é simplificado ao considerar que:

- cada  $k$ -nó tem  $k$  chaves e  $k$  links



# B-tree

---

- Aplicação Principal: sistemas de ficheiros
- Espaço *versus* tempo:
  - M grande - uma árvore com pouca altura.
  - M pequeno - menos espaço desperdiçado

# Exemplo B-tree

706	→

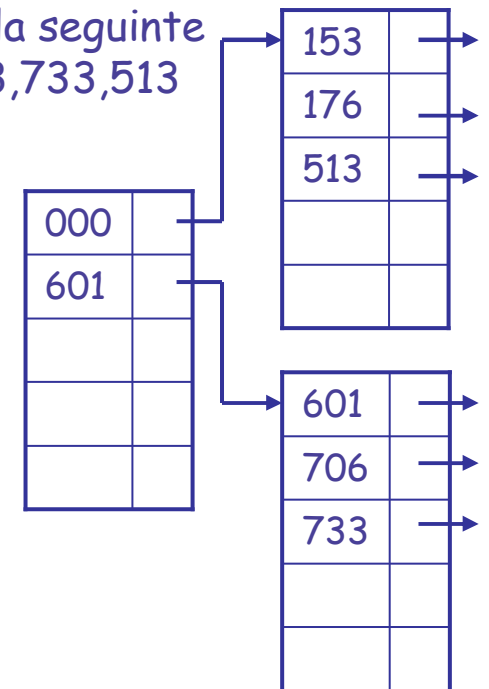
176	→
706	→

Este exemplo mostra inserções de 6 chaves numa B-tree com  $M=5$ . As chaves são inseridas pela seguinte ordem: 706, 176, 601, 153, 733, 513

176	→
601	→
706	→

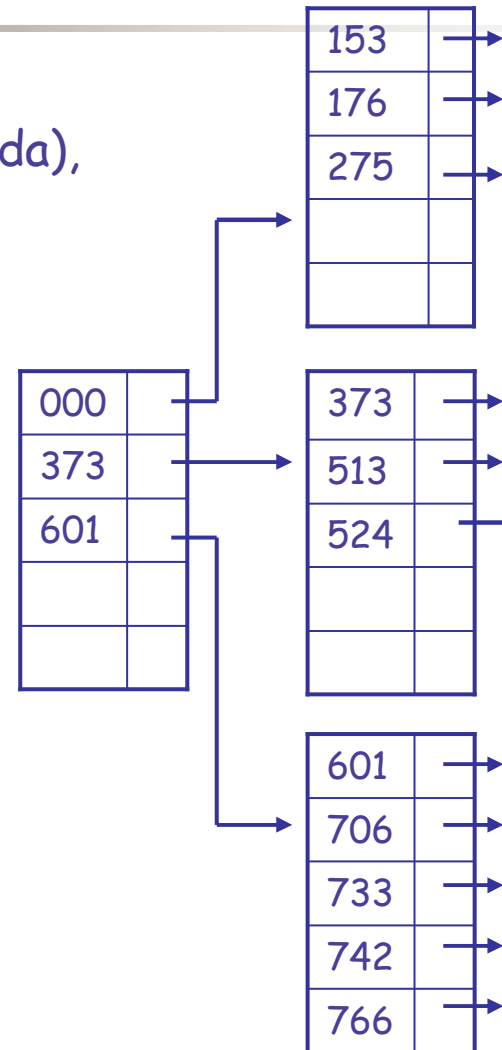
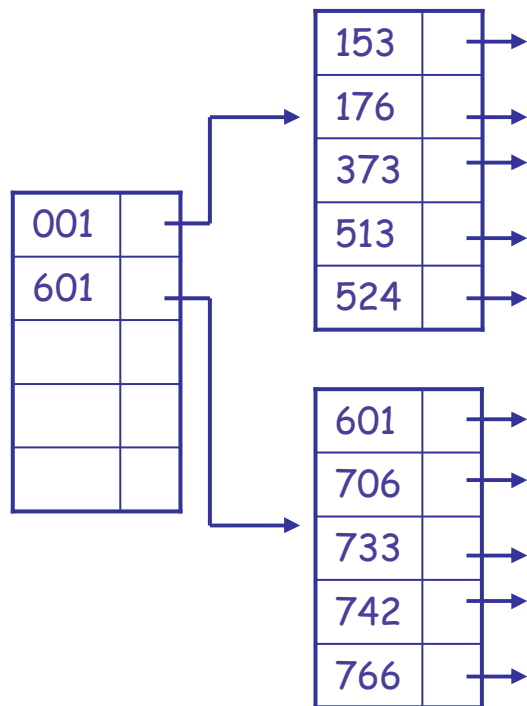
153	→
176	→
601	→
706	→

153	→
176	→
601	→
706	→
733	→



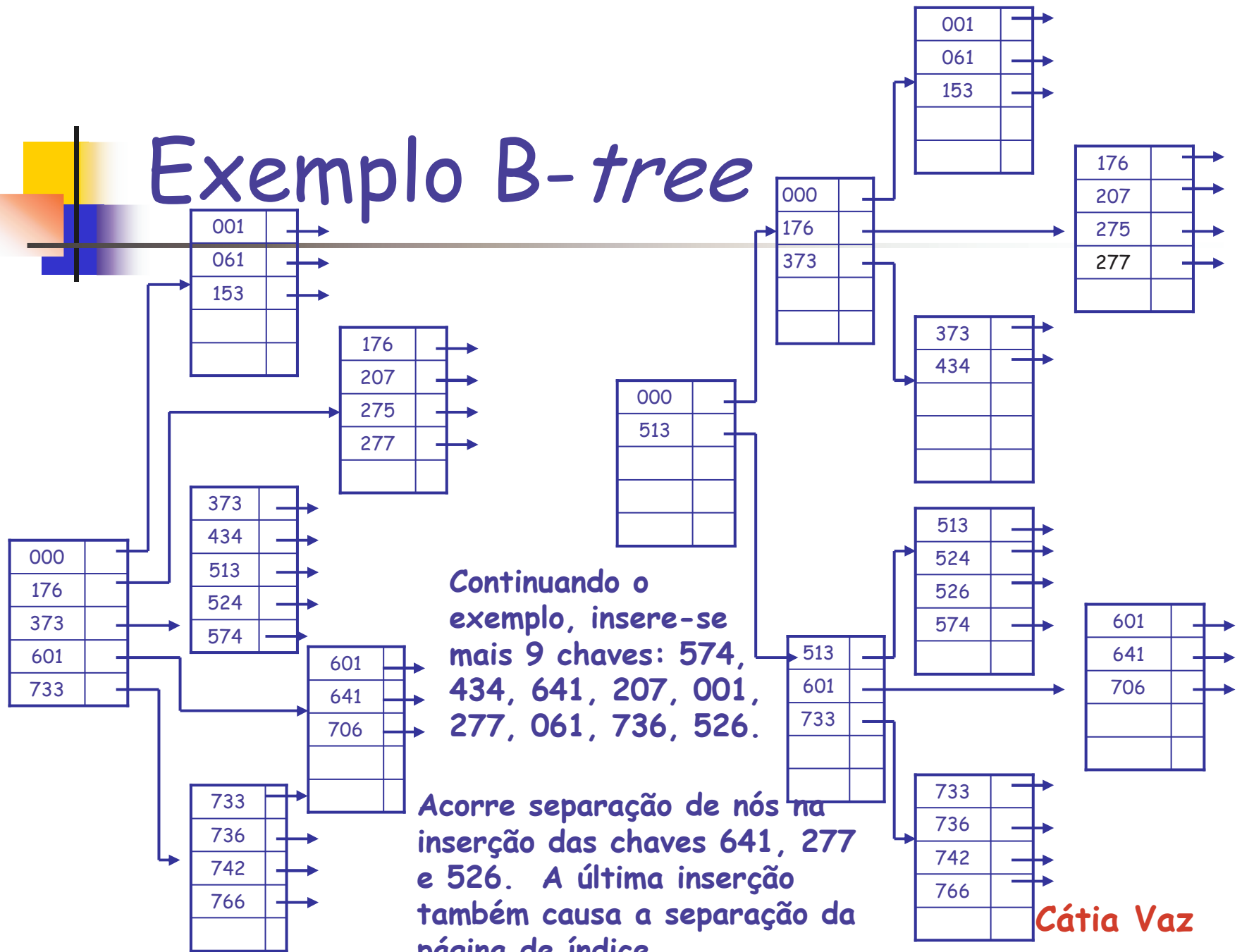
# Exemplo B-tree

Após a inserção das chaves 742, 373, 524 e 766 (à esquerda), insere-se a 275 (à direita).



A página onde devia ser inserido o 275 dividiu-se.

# Exemplo B-tree



Cátia Vaz





# B-tree

```
public class Btree{
    private class entry{
        KEY key; ITEM item; Node next;
        entry(KEY v, ITEM x){key = v; item = x;} /*nós folhas*/
        entry(KEY v, Node u){key = v; next = u;} /*nós não terminais*/
    }
    private class Node{
        int m; entry[] b;
        Node(int k){b = new entry[M]; m = k; }
    }
    private Node root;
    private int HT;
    public Btree(int maxN) { HT = 0; root = new Node(0); }
    public ITEM search(KEY key){ . . . }
    public void insert(ITEM x){ . . . } ...}
```

Cátia Vaz



# B-tree

/\*Para nós não terminais (altura positiva), procura-se pela chave maior do que a chave de procura, e realiza-se uma chamada recursiva na sub-árvore referenciada pelo link anterior. Para as folhas (altura 0), verifica-se se existe um item com chave igual a chave da procura.\*/

```
private ITEM searchR(Node h, KEY v, int ht){
    if(ht == 0)
        for (int j = 0; j < h.m; j++){
            entry e = h.b[j]; if (equals(v, e.key)) return e.item;
        }
    else
        for(int j = 0; j < h.m; j++)
            if ((j+1 == h.m) || less(v, h.b[j+1].key))
                return searchR(h.b[j].next, v, ht-1);
    return null;}

```

```
ITEM search(KEY key) { return searchR(head, key, HT); }
```

Cátia Vaz



# B-tree

```
private Node split(Node h){
    Node t = new Node(M/2);
    h.m = M/2;
    for (int j = M/2; j < M; j++)
        t.b[j-(M/2)] = h.b[j];
    return t; }

public void insert(ITEM x){
    Node u = insertR(root, x, HT);
    if (u == null) return;
    Node t = new Node(2);
    t.b[0] = new entry((root.b[0]).key, root);
    t.b[1] = new entry((u.b[0]).key, u);
    root = t; HT++;
}
```

Cátia Vaz



# B-tree

```
private Node insertR(Node h, ITEM x, int ht){
    int i, j; KEY v = x.key(); Node u; entry t = new entry(v, x);
    if(ht == 0)
        for(j = 0; j < h.m; j++){
            if(less(v, (h.b[j]).key)) break;}
    else
        for(j = 0; j < h.m; j++){
            if((j+1 == h.m) || less(v, (h.b[j+1]).key)){
                u = insertR(h.b[j+1].next, x, ht-1);
                if(u == null) return null;
                t.key = (u.b[0]).key;
                t.next = u; break;
            }
        }
    for (i = h.m; i > j; i--){ h.b[i] = h.b[i-1];}
    h.b[j] = t; h.m++;
    if (h.m < M) return null;
    else return split(h); }
```



# B-tree

---

- **Propriedade:** Uma procura ou uma inserção numa B-tree de ordem  $M$  com  $N$  items requiere entre  $\log_M N$  e  $\log_{M/2} N$  primeiros acessos a páginas.