ESTRUTURAS DE DADOS

prof. Alexandre César Muniz de Oliveira

Sugestão bibliográfica:

- **ESTRUTURAS DE DADOS USANDO C**
Aaron M. Tenenbaum, et alli

- **DATA STRUCTURES, AN ADVANCED APPROACH USING C**
Jeffrey Esakov & Tom Weiss

- **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA (2ED)**
Michael Godrich & Roberto Tamassia

BUSCA

1. Introdução

Arquivo = tabela = conjunto de registros
Chave Interna = campo do registro
Externa = ponteiro para campo chave em outra tabela

Algoritmo de Busca
objetivo:
encontrar registro cuja chave é A
retorna:
registro inteiro
ponteiro para o registro
registro nulo
ponteiro nulo

2. Algoritmo de busca seqüencial

Aplicável a tabela armazenada em vetor, em lista encadeada e até árvore

```
faça (i=0;i<n;i++) {  se(chave==k[i]) retorna(i);  }
retorne(-1); // valor nulo para índices
```

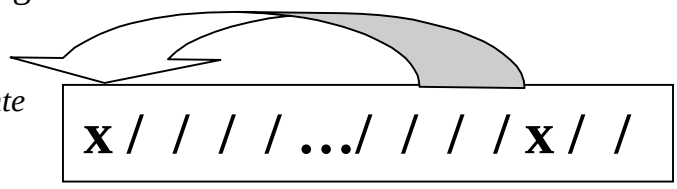
3. Eficiência da busca seqüencial

Melhor Caso: Encontrar na 1ª posição
1 comparação
Pior Caso: Encontrar na N-ésima posição
N comparações
Caso Médio: (n+1)/2 → O(n) (limite superior para uma busca)

4. Ordenar tabela por probabilidade

Força a ocorrência do Caso 1 (melhor caso), sem conhecer as probabilidades de busca dos registros.

```
pesquisa
se encontrar
  move o registro para frente
  retorna valor verdadeiro
se não
  retorna falso
```

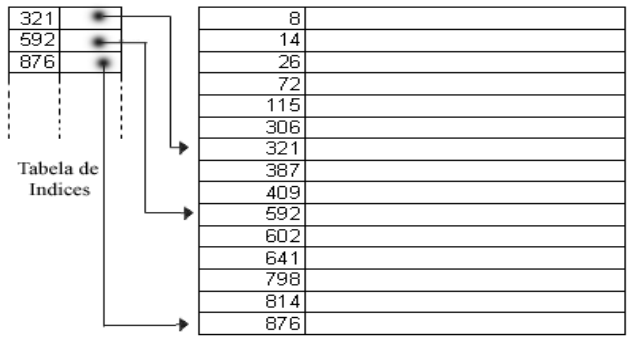


BUSCA

5. Ordenar tabela por chave

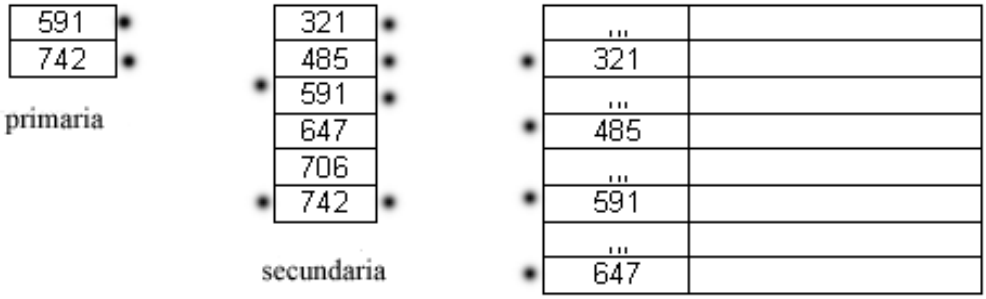
Não ordenada: N comparações para descobrir que não existe na tabela
Ordenada: N/2 comparações em média
Continua sendo O(n)

6. Busca seqüencial indexada



(1) Busca seqüencial sobre o arquivo. Índice pequeno.
(2) Acesso a todos os registros seqüencialmente.
(3) Eficiência depende do tamanho do índice primário.
(4) Necessidade de índice secundário
(5) Inclusão e remoção.

Busca Sequencial Indexada



Pesquisa em índice

Pesquisa até fim ou até chave<=indice[i].k
Se não fim
lim_baixo = indice[i-1].k
lim_alto = indice[i].k
Pesquisa de lim_baixo a lim_alto por chave seqüencialmente
Se encontrou
Recupera registro;
Falha na busca

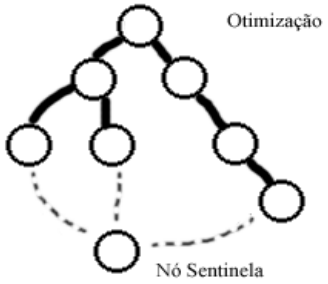
BUSCA

7. Busca binária

- Tabela ordenada de tamanho conhecido
- A tabela é subdividida à razão de 2 em subtabelas até encontrar ou não o elemento
- Nº máximo de comparações é Log_2 n → O(Log n)
- A divisão do problema em partes não depende dos dados, somente o tamanho do conjunto.
- Estruturas com muitas inserções e deleções inviabiliza o uso de vetores e inviabiliza busca binária

8. Busca em árvores binárias

```
p=raiz;
enquanto (p<>nulo && chave<>info(p))
        p=(chave<k(p)?esq(p):dir(p));
retorna(p);
// para evitar as comparações
// com NULL
// antes de iniciar a busca,
// armazenar a chave no nó sentinela
// que é ponto final de todas as
// subárvores
```




```
p=head(raiz);
head(sentinela);
enquanto (chave<>info(p))
p=(chave<k(p)?esq(p):dir(p));
retorna(p);
```

8.1 Inserção

Uma nova chave é sempre inserida como folha.

BUSCA

8.2 Exclusão

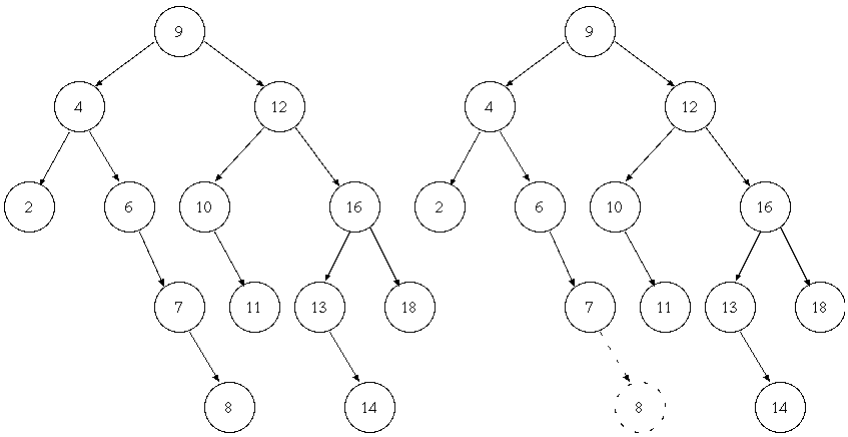
Remover uma chave sem alterar a ordem das demais

OBS:

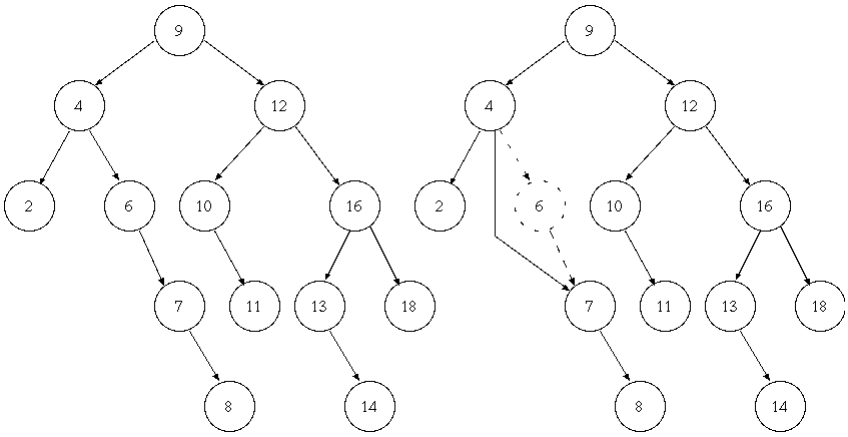
Casos Especiais Para Remoção De Nós

1. o nó a remover é nó folha
2. o nó a remover tem 1 subárvores
3. o nó a remover tem 2 subárvores

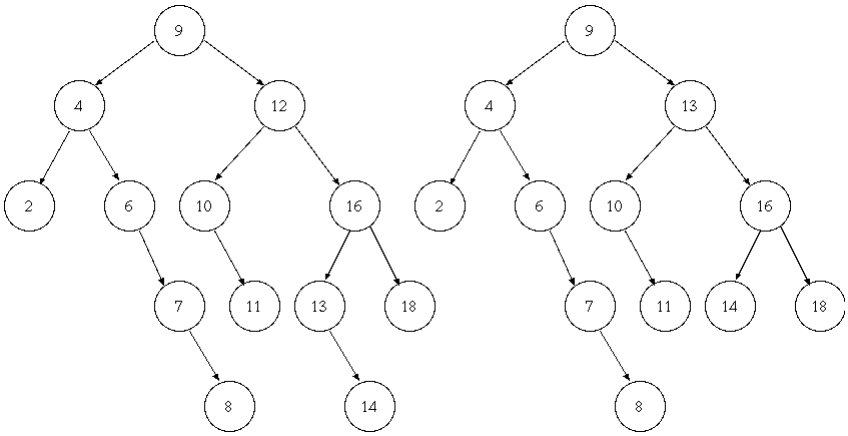
O caso de um nó sem filhos é o mais simples e significa apenas ajustar o ponteiro de seu pai. No caso do nó ter um único filho a mudança na árvore também é simples significa mover o nó filho daquele será removido uma posição para cima. Uma forma mais elegante seria atualizar o ponteiro do nó pai do nó a ser removido para apontar para a subárvore abaixo do nó a ser removido. O caso mais complexo é o do nó com dois filhos. Neste caso, deve-se procurar o sucessor *s* (ou antecessor) do nó deverá ocupar este lugar. Este nó (sucessor) é um descendente que está na subárvore da direita do nó e corresponde ao nó mais à esquerda desta árvore. Ele não tem filhos à esquerda e portanto para remove-lo, cai-se no segundo caso de remoção.



Removendo nó (8) sem filhos.



Removendo nó (6) com um filho.



Removendo nó (12) com dois filhos.

No exemplo acima, *s* (13) é movido para o nó a ser removido (12) e por sua vez o próprio *s* é removido também (caso 2), completando a operação.

BUSCA

Exemplo de código em C

```
tNo *remover (tNo *tree, int num) {
    tNo *p,  /* p aponta para o no a ser
removido */
        *q,  /* q aponta para o pai do no */
        *rp, /* rp aponta que ira substituir o no
p */
        *f,
        *s;  /* sucessor do no p */

   p = tree; q=NULL;

    /* procura o no com a chave num, p
aponta para o no
    e q aponta para o pai do no */
    while ( p && p->info != num) {
         q = p;
         if ( num < p->info)
            p = p->esq;
         else
             p = p->dir;
    } /* fim do while */
    if (!p) return NULL; /* a chave nao existe
na arvore */

    /* agora iremos ver os dois primeiros
casos, o no tem um filho
       no maximo */
    if (p->esq == NULL)   rp = p->dir;
    else
       if (p->dir == NULL)  rp = p->esq;
       else {
           f=p;
           rp = p->dir;
           s = rp->esq;   /* s e sempre o filho
esq de rp */
           while (s != NULL) {
               f = rp;
               rp = s;
               s = rp->esq;  }
           /* neste ponto, rp e o sucessor em
ordem de p */
           if (f != p) {
             /*  p nao e o pai de rp e rp == f-
>left */
              f->esq = rp->dir;
              /* remove o no rp de sua atual
posicao e o
                 substitui pelo filho direito de rp
                 rp ocupa o lugar de p
              */
              rp->dir = p->dir;    }
           /* define o filho esquerdo de rp de
modo que rp
              ocupe o lugar de p
           */
           rp->esq = p->esq;
       }
    /* insere rp na posicao ocupada
anteriormente por p */
    if (q == NULL)
      tree = rp;
    else
       if (p == q->esq)    q->esq = rp;
       else     q->dir = rp;
    free(p);
    return rp;
}
```

8.3 Eficiência

Eficiência:

Tempo de execução de busca
Espaço extra para ponteiros

Tempo de busca = f (grau de balanceamento)

Grau de balanceamento: "Simetria"

* { Seqüência Aleatória
  { Seqüência Classificada ou Inversa

Simetria := f (Ordem de Entrada*) + f (Inserções, Remoções)

Método de Inserção e Remoção que garantam a árvore simetria ou balanceada

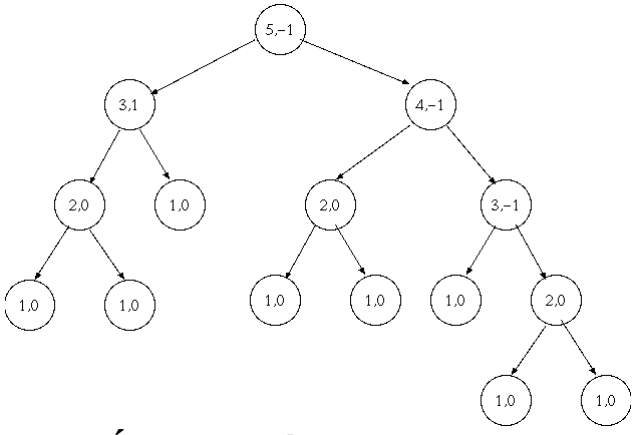
8.4 Árvores Binárias Balanceadas

Uma árvore binária balanceada, chamada de árvore AVL (em homenagem aos russos Adel'son-Vel'skii e Landis, que propuseram o algoritmo de balanceamento), é uma árvore binária na qual as alturas das duas subárvores de cada um dos nós nunca diferem em mais de 1 unidade.

```
int Altura (p)
Begin
        int Alt_Esq, Alt_Dir;
        Begin
        If Nodo = NULL
                Then Altura := -1
        Else Begin
                Alt_Esq := Altura (esq(p));
                Alt_Dir := Altura (dir(p));
                If Alt_Esq > Alt_Dir
                        Then Altura := 1 + Alt_Esq
                Else Altura := 1 + Alt_Dir;
        End;
End;
```

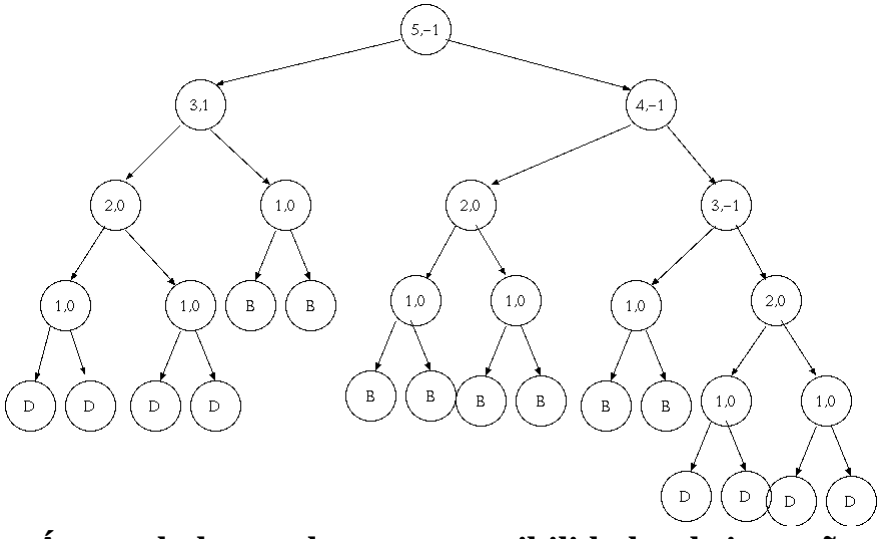
BUSCA

O balanceamento de um nó é igual a diferença entre as suas altura esquerda e direita. Portanto, cada nó de uma árvore balanceada tem balanceamento igual a -1, 0 ou 1, dependendo da comparação entre as alturas esquerda e direita. Lembrando que a altura de um nó *n* da árvore é o número de nós do maior caminho de *n* até um de seus descendentes. As folhas tem altura 1. Uma árvore binária completa com n>0 nós tem altura mínima, que é igual a $1 + \lfloor \log(n) \rfloor$.



Árvore binária balanceada.

Caso a probabilidade de pesquisar uma chave em uma tabela seja a mesma para todas as chaves, uma árvore binária balanceada terá a busca mais eficiente. Cada inserção que mantém a árvore balanceada está indicada por um B e as que desbalanceiam a árvore por um D.

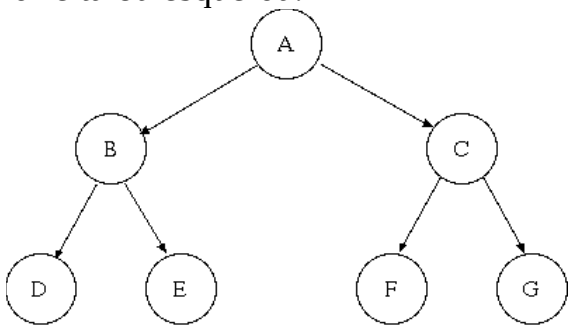


Árvore balanceada e suas possibilidades de inserção

Observe que uma árvore se torna desbalanceada quando o nó inserido se torna descendente esquerdo de um nó que tinha anteriormente um balanceamento de 1 ou se ele se tornar descendente direito de um nó que tinha anteriormente balanceamento de -1. Por exemplo, um nó que tinha balanceamento 1 e recebe um descendente direito aumenta sua altura em 1, portanto aumentando o seu desbalanceamento. Para manter a árvore balanceada é necessário que a transformação na árvore de tal modo que:

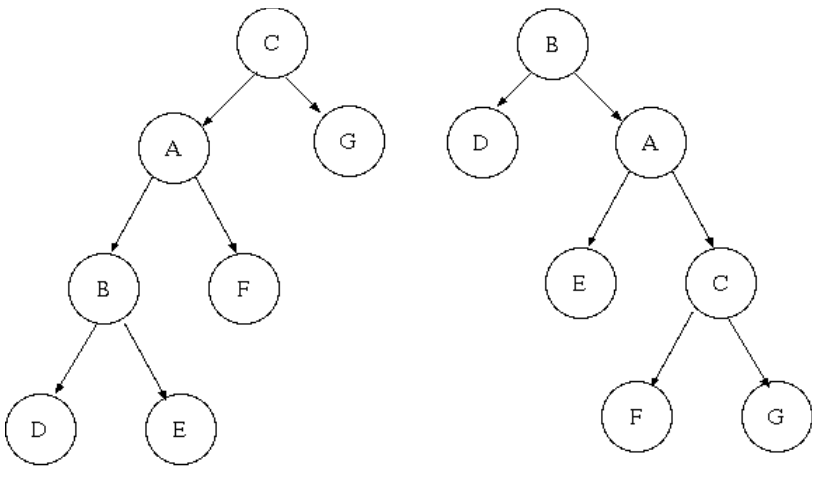
1. a árvore permaneça uma árvore de busca binária;
2. a árvore continue a ser uma árvore balanceada.

O algoritmo de balanceamento consiste em rotacionar subárvores desbalanceadas no sentido contrário ao do desbalanceamento. Uma rotação pode ser à direita ou esquerda.



(a) Árvore original

Árvore original antes da rotação



(a) Rotação à esquerda (b) Rotação à direita

Efeitos das rotações

BUSCA

Um possível algoritmo para rodar para a esquerda uma árvore enraizada em *p* é:

```
RotaçãoEsquerda(p){
        q = dir(p);
        r = esq(q);
        esq(q) = p;
        dir(p) = r;
}
```

- Algoritmo de balanceamento

O algoritmo de pesquisa é responsável por marcar o nó mais jovem que pode ficar desbalanceado, seguindo a trilha de nós visitados até a posição de inserção. Apenas dois nós interessam: o nó *p* mais jovem que ficou desbalanceado e o *filho(p)* (*esq(p) ou dir(p)*, dependendo da subárvore onde foi feita a inclusão). A partir daí surgem dois casos a se considerar:

```
se sinal (p) = sinal(filho(p))
  Rotação*(p) // Rotação no sentido contrário ao desbalanceamento
se não
  Rotação-*(filho(p))
  Rotação*(p)  // Rotações em sentidos inversos
```

BUSCA

9 ÁRVORES MULTIDIRECIONAIS

- **Motivação**

Velocidade de acesso em memória principal (interna) é de microsegundos, enquanto que para memória auxiliar (externa) é de milisegundos.

Necessário minimizar acessos a disco, agrupando os dados em blocos (256 a 1024 *words*)

- **Definição**

São árvores de busca com uma certa ordem *m* que significa o número máximo de nós *filhos* que cada nó pode ter.

Seja, $k \leq m$, o número de *filhos* de um determinado nó, então o número de chaves nesse nó é *k-1*. Isto significa que tem-se, para cada *chave*, uma *sub-árvore* associada e mais uma, chamada de *sub-árvore default.* Exemplos: *topdown-tree,b-tree*

- **Conceitos**

⇒Nó completo - nó com o número máximo de sub-árvores
⇒Árvore *top-down* - em que todo nó incompleto é uma folha
⇒Semi-folha - nó com pelo menos 1 sub-árvore vazia

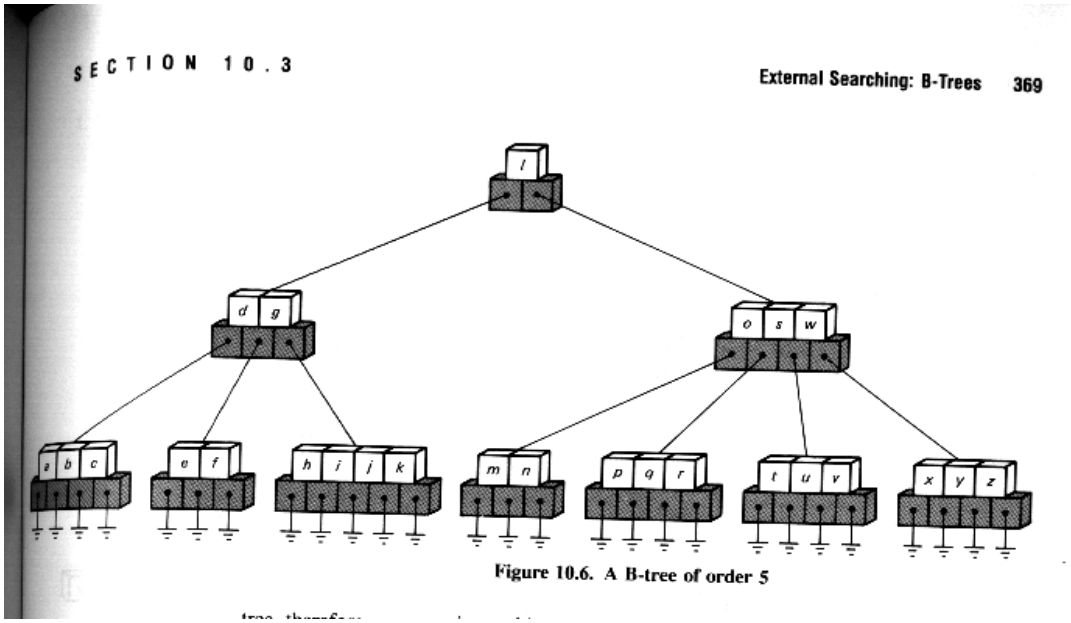
10 ÁRVORES MULTIDIRECIONAIS BALANCEADAS (B-TREES)

- **Definição**

⇒ Seja uma árvore-B (B-tree) de ordem m:
⇒ Todas as folhas estão no mesmo nível;
⇒ Todos os nós internos (não-folhas), exceto a raiz têm no mínimo $\lceil m/2 \rceil$ sub-árvores não vazias.

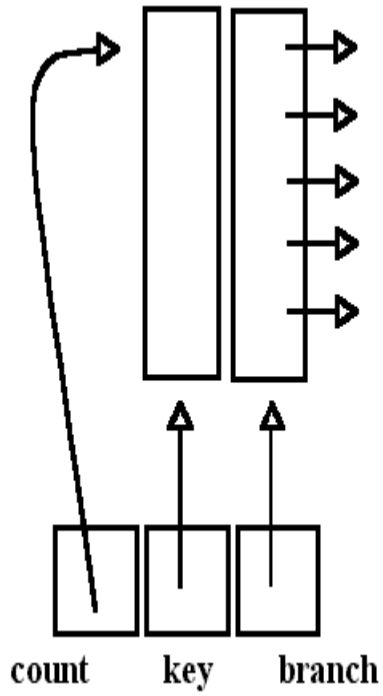
BUSCA

- **Estruturas de dados elementares**




```
Typedef struct node_tag {
        Int count;
        Key_type key [MAX+1];
struct node_tag *branch [MAX+1];
} Node_type;

Node_type * Search (target, root, targetpos)
Key_type target;
Node_type * root;
int * targetpos;
{
if ( root == NULL) return NULL;
else
if (SearchNode(target, root, targetpos))
  return root;
else
 return Search(target, root->branch[*target],
targetpos);
}
```



BUSCA

- **Inserção de elementos**

  Árvores-B crescem de baixo para cima, isto é, à medida que estas vão tornando-se completas, nós são criados em um nível superior. Toda vez que uma folha enche, ocorre o chamado split, isto é, subdivisão em duas folhas, cada uma com a metade das chaves da folha que "estourou" e uma das chaves (mediana) é exportada para o nó pai dessa folha.

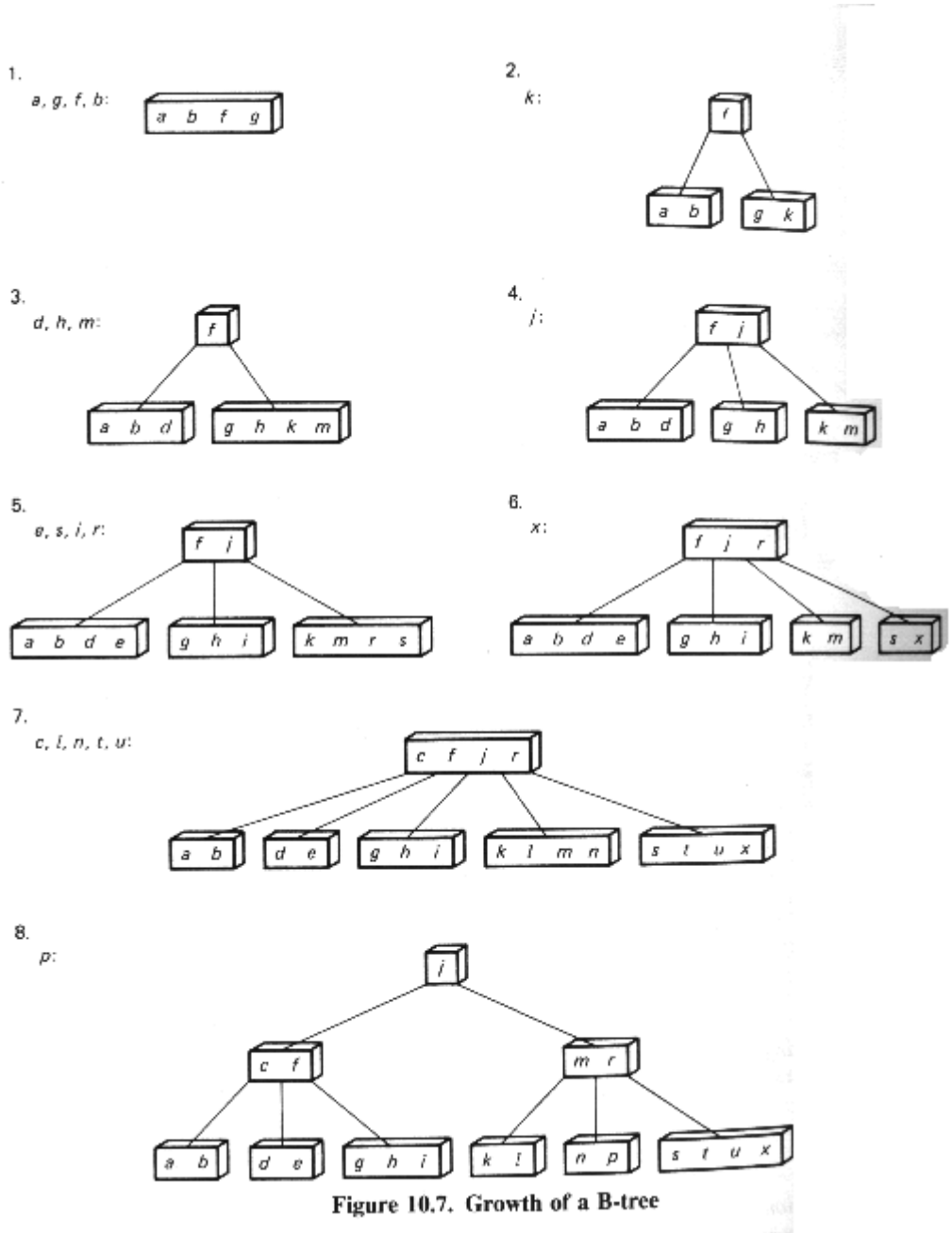


Figure 10.7. Growth of a B-tree

BUSCA

O SPLIT:

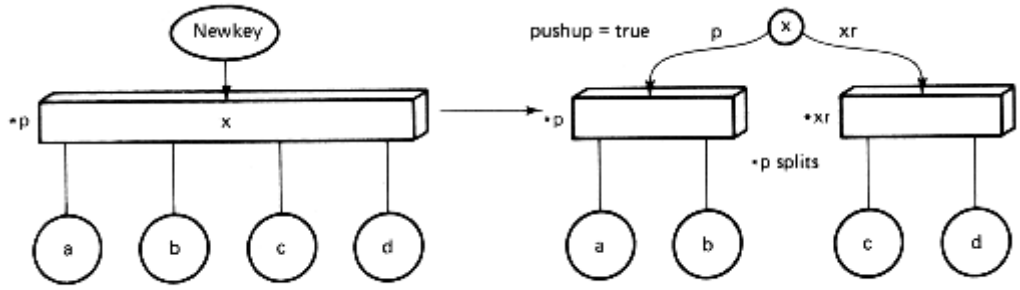


Figure 10.8. Action of PushDown **function**

Cria um novo nó (xr)
Elege a chave mediana (x)
Insere a mediana (x) no nó pai (recursivo, pois pode encher o nó pai também)
Faz a sub-árvore correspondente apontar para o novo nó (xr)

```
/* PushDown: recursively move down tree searching for newkey. */
Boolean_type PushDown(Key_type newkey, Node_type *p, Key_type *x,
                      Node_type **xr)
{
    int k;                              /* branch on which to continue the search
    if (p == NULL) {                    /* cannot insert into empty tree; terminates
        *x = newkey;
        *xr = NULL;
        return TRUE;
    } else {                            /* Search the current node.
        if (SearchNode(newkey, p, &k))
            Error("inserting duplicate key");
        if (PushDown(newkey, p->branch[k], x, xr))
                                        /* Reinsert median key.
            if (p->count < MAX) {
                PushIn(*x, *xr, p, k);
                return FALSE;
            } else {
                Split(*x, *xr, p, k, x, xr);
                return TRUE;
            }
        return FALSE;
    }
}
```

BUSCA

- **Remoção de elementos**
  O método usado para remoção de chaves em uma árvore-B se baseia no fato em que se a chave a ser removida não está numa folha, então o predecessor (ou sucessor) imediato (na ordem natural de chaves) está sempre em uma folha. Assim, promover o sucessor (ou) para a posição da chave a ser removida permite que haja sempre remoções nas folhas.

Algoritmo básico

Se a folha contém mais que o mínimo de chaves,
não é necessário nenhuma outra ação.
Se a folha for mínima, então
procure nas duas folhas imediatamente adjacentes e
filhas do mesmo nó pai.
Se uma dessas folhas não for mínima, então
mova uma chave para o nó pai e
mova do nó pai para a folha onde ocorreu a remoção
(mantendo o número mínimo).
Se não for possível, então
as duas folhas adjacentes e
mais a chave mediana do nó pai
são combinadas em um nó somente.
Se este nó ainda estiver menor que o mínimo então
processo se repete em direção à raiz,
até que fique mínimo ou
então reduzindo um nível da árvore.

BUSCA

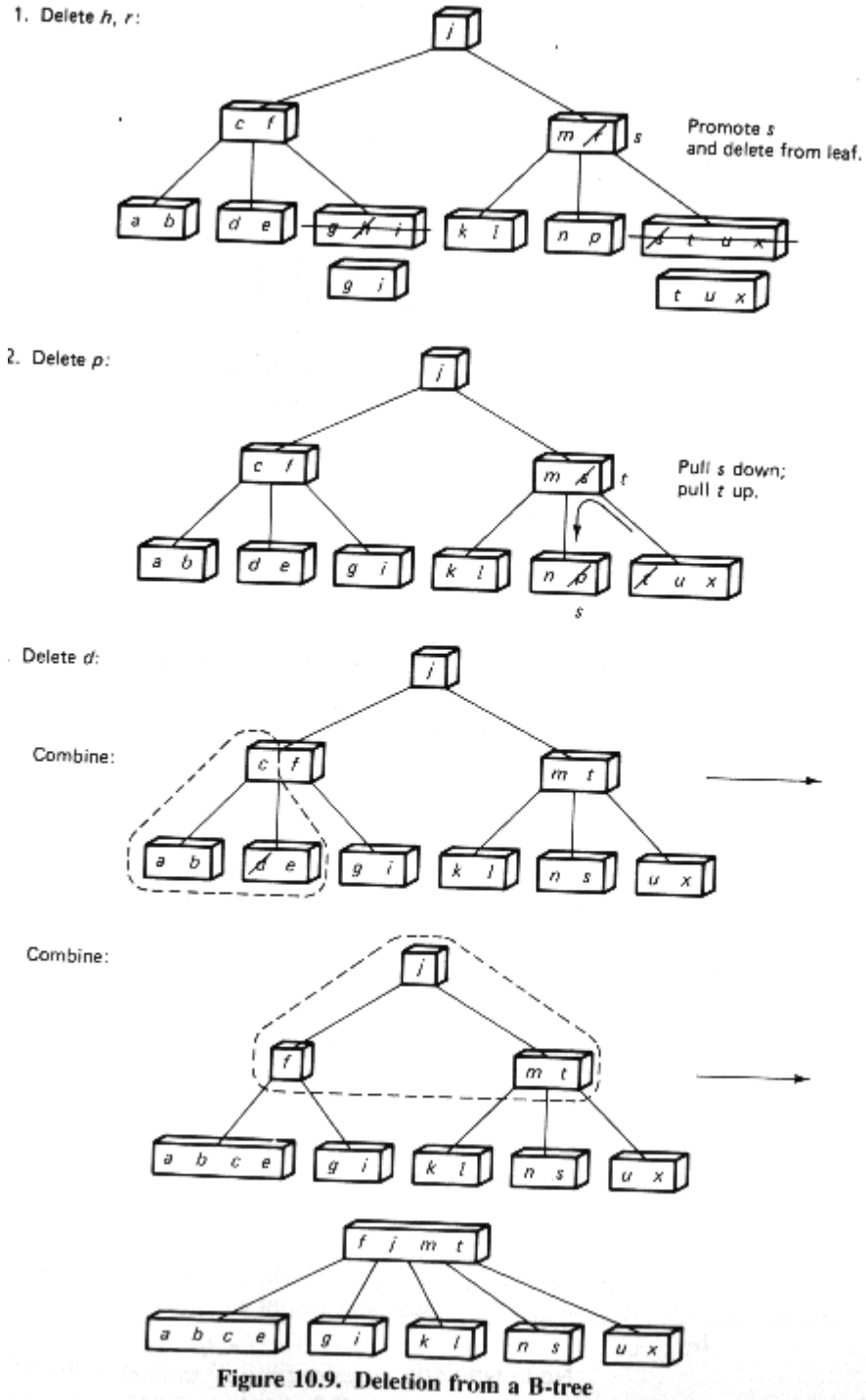


Figure 10.9. Deletion from a B-tree

BUSCA

Código em C

```
/* Delete: deletes the key target from the B-tree with the given root */
Node_type *Delete(Key_type target, Node_type *root)
{
    Node_type *p;                    /* used to dispose of an empty root
    if (!RecDelete(target, root))
        Error("Target was not in the B-tree.");
    else if (root->count == 0) {     /* Root is empty.
        p = root;
        root = root->branch[0];
        free(p);
    }
    return root;
}
```

```
/* RecDelete: look for target to delete. */
Boolean_type RecDelete(Key_type target, Node_type *p)
{
    int k;                   /* location of target or of branch on which to search
    Boolean_type found;
    if (p == NULL)
        return FALSE;                /* Hitting an empty tree is an error.
    else {
        if ((found = SearchNode(target, p, &k)))
            if (p->branch[k-1]) {
                Successor(p, k);  /* replaces key[k] by its successor
                if (!(found = RecDelete(p->key[k], p->branch[k])))
                    /* We know that the new key[k] is in the leaf. */
                    Error("Key not found.");
            } else
                Remove(p, k);  /* removes key from position k of *p
        else                         /* Target was not found in the current node.
            found = RecDelete(target, p->branch[k]);
        /* At this point, the function has returned from a recursive call. */
        if (p->branch[k] != NULL)
            if (p->branch[k]->count < MIN)
                Restore(p, k);
        return found;
    }
}
```

BUSCA

11. Espalhamento

Arquivos organizados (acesso, manutenção)
. seqüencial
. seqüencial indexado
. árvore

Percorre um certo número de chaves, fazendo comparações "desnecessárias"

Alternativa:
. acesso direto

Acesso a arquivo é O(1): o tempo de execução é o mesmo, não importa o quanto o arquivo possa crescer. Essa é a situação ideal, mas há problemas de desempenho para arquivos dinâmicos, cujo tamanho é altamente variável . Por exemplo, árvores-B: acesso $O(\log_k N)$, sendo k a ordem da árvore.

Hashing estático: garante acesso O(1) para arquivos estáticos. Vamos começar com uma descrição de técnicas de hashing estáticas, e depois ver como estender as técnicas para lidar com arquivos dinâmicos.

Hashing
Significa picar, misturar, confundir,...No contexto atual: espalhar. Uma função de espalhamento (função *hash*) h(k) transforma uma chave k em um endereço. Este endereço é usado como a base para o armazenamento e recuperação de registros. É similar a uma indexação, pois associa a chave ao endereço relativo do registro.

Entretanto:

- no espalhamento os endereços parecem ser aleatórios - não existe conexão óbvia entre a chave e o endereço, apesar da chave ser utilizada no cálculo do endereço
- no espalhamento duas chaves podem levar ao mesmo endereço (colisão) - portanto, colisões devem ser tratadas.

BUSCA

Exemplo:

Para armazenar 750 registros em um arquivo para o qual a chave é o campo *sobrenome*. Suponha que foi reservado espaço para manter 1.000 registros (algum desperdício de área). Os registros podem ser "espalhados" através do seguinte procedimento: soma-se, duas a duas, as representações ASCII (código numérico) das letras que formam o sobrenome e usa-se os três dígitos menos significativos do resultado (resto da divisão pelo máximo de registros) para servir de endereço.

```
FUNCTION hash (CHAVE,MAXREGISTROS)
  SUM := 0
  J := 0
  while (J < 12)
    SUM := (SUM+100*CHAVE[J]+CHAVE[J+1])mod 19937
    J := J+ 2
  endwhile
  return (SUM mod MAXREGISTROS)
end FUNTION
```

Para garantir que as sucessivas somas sejam menores que o valor máximo permitido, seria identificando qual a maior parcela que pode aparecer em uma soma (no caso, o inteiro correspondente a ZZ, que é 9.090). Assim, a cada soma parcial, o resultado difere do máximo em pelo menos este valor. Por exemplo, se o valor intermediário máximo permitido para a soma for escolhido como 19.937, este valor difere de 32.767 em mais de 9.090, de forma que podemos garantir que uma nova soma não provoca *overflow*. Isso pode ser feito usando a função MOD na soma.

Registros são colocados em posições aparentemente aleatórias, sem considerar a sua ordem alfabética. Entretanto, sobrenomes diferentes podem produzir o mesmo endereço (**colisão**).

BUSCA

Soluções contra colisões:

- Ideal: usar uma função de espalhamento perfeita, que não produz colisão ou pelo menos uma função, ou algoritmo, que distribua os registros relativamente por igual entre os endereços disponíveis.
- A solução prática é tentar reduzir a um valor aceitável o número de colisões que podem ocorrer. Por exemplo, se uma entre 10 buscas resultar em colisão, então o número médio de acessos a disco necessários para recuperar um registro continua muito pequeno.
- Utilização de mais memória: é relativamente fácil achar um bom algoritmo quando se pode espalhar poucos registros em muitos endereços, em outras palavras, desperdiçar muito espaço (no exemplo: 75/1000=7,5%).
- Utilização de mais de um registro por endereço: Nesse caso, o endereço (chamado de cesto - *bucket*) tem espaço para armazenar vários registros que colidiram. Uma lista de endereço colididos pode ser vista como um cesto.
- Usar uma função de re-espalhamento: o re-espalhamento linear é a mais comum forma de solucionar colisões.