Universidade Federal de Mato Grosso do Sul
Facom - Faculdade de Computação
Bacharelado Ciência da Computação

Estruturas de Dados e Programação

B-Árvores

Nos programas de bancos de dados, onde a maioria da informação está guardada em dispositivos de armazenamento secundário, a penalidade do tempo para acessar a esses dispositivos pode ser significativamente reduzida pela pela escolha apropriada das estruturas de dados utilizadas.

Uma B-árvore opera junto com o meio de armazenamento secundário e pode ser sintonizada para reduzir a quantidade de acessos a esses dispositivos lentos. Uma propriedade importante das B-árvore é o tamanho de cada nó, que pode ser tão grande quanto o tamanho de um bloco de disco. O número de chaves em um nó pode variar dependendo do tamanho das chaves, da organização dos dados e, naturalmente, do tamanho de um bloco. O tamanho do bloco varia para cada sistema: pode ter 512 bytes, 4KB ou mais. Usualmente, o tamanho dos nós de uma B-árvore é o mesmo tamanho de um bloco de disco.

Uma *B-árvore* de ordem m é uma árvore de busca com as seguintes propriedades:

1. a raiz possui $k-1$ chaves ordenadas e k ponteiros para subárvores, onde $2 \leq k \leq m$. Se a raiz é uma folha, os k ponteiros são nulos.
2. os nós folhas contêm $k-1$ chaves ordenadas $\lceil m/2 \rceil \leq k \leq m$
3. os demais nós contêm $k-1$ chaves ordenadas e k ponteiros não nulos para subárvores, onde $\lceil m/2 \rceil \leq k \leq m$
4. todas as folhas estão no mesmo nível

De acordo com essas condições, uma árvore B sempre está cheia até a metade, pelo menos, tem poucos níveis e está perfeitamente balanceada.

Usualmente o valor de m é grande, de modo que a informação armazenada em um bloco de disco caiba em um nó. Na figura 1 temos um exemplo de uma B-árvore de ordem 5. Para fins didáticos, definimos a B-árvore e mostramos exemplos contendo apenas campos *chave* que armazenam valores inteitos. No entanto, nos casos práticos os nós armazenam registros inteiros contendo vários campos de diversos tipos e cada registro é identificado por uma chave única.

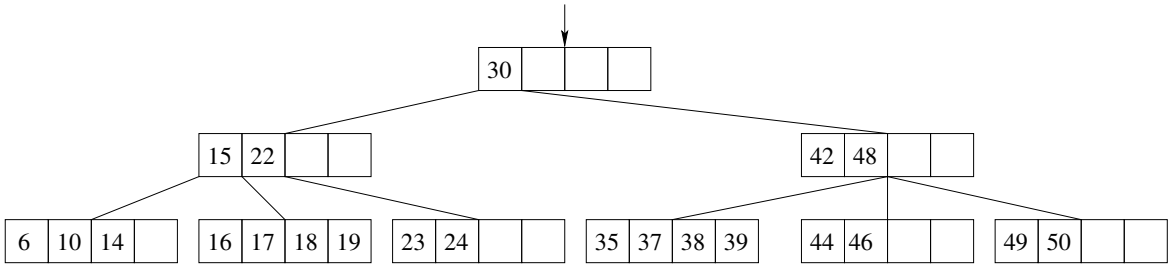

Figura 1: Exemplo de uma B-árvore de ordem 5.

1 Altura de uma B-árvore de ordem m

Vamos estudar a relação entre o número de nós de uma B-árvore e a sua altura. É fácil ver que quanto menor a quantidade de chaves nos nós da árvore, maior será a sua altura. O pior caso ocorre quando cada nó possui a quantidade mínima $q = \lceil m/2 \rceil$ de ponteiros não nulos para subárvores (consequentemente, cada nó possui $q-1$ chaves). Nesse caso, em uma B-árvore de altura h, existem

$$\begin{gathered}
1 \text{ chave no nível 1 (raiz)}+ \\
2(q-1) \text{ chaves no nível 2 } + \\
2q(q-1) \text{ chaves no nível 3 } + \\
2q^2(q-1) \text{ chaves no nível 4 } + \\
\ldots \\
2q^{h-2}(q-1) \text{ chaves nas folhas (nível } h) \\
= \\
1 + (\textstyle\sum_{i=0}^{h-2} 2q^i)(q-1).
\end{gathered}$$

Considerando que o somatório dos primeiros n elementos em uma progressão geométrica é

$$\textstyle\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1} \ ,$$

o número de chaves na B-árvore, no pior caso, pode ser expresso como

$$1 + 2(q-1)\textstyle\sum_{i=0}^{h-2} q^i = 1 + 2(q-1)\frac{q^{h-1}-1}{q-1} = -1 + 2q^{h-1}$$

A relação entre o número de chaves em qualquer B-árvore e a sua altura é, então, expressa como

$$n \geq -1 + 2q^{h-1}.$$

Resolver essa desigualdade para a altura h resulta em:

$$h \leq log_q \frac{n+1}{2} + 1$$

Isso significa que par auma ordem m suficientemente grande, a altura h é pequena mesmo para um grande número de chaves armazenadas na árvore. Por exemplo, considere $m = 200$ e $n = 2000000$, então $h \leq 4$. No pior caso, encontrar uma chave nessa árvore exige quatro buscas (uma para cada nó do caminho da raiz até uma folha). Se a raiz pode ser mantida na memória durante todo o tempo, esse número pode ser reduzido somente a três acessos ao meio de armazenamento secundário.

2 Classes BTreeNode e BTree

Um nó de uma B-árvore é usualmente implementado como uma classe contendo um vetor de $m - 1$ inteiros para armazenamento das chaves, um vetor com m ponteiros para outros nós e possivelmente outras informações para controle e manutenção da árvore. Segue abaixo a definição base de um nó de uma B-árvore e em seguida o código referente aos métodos que operam sobre um nó da B-árvore.

```
class BTreeNode {
public:
  int n;
  int* keys;
  bool leaf;
  BTreeNode** pointers;
  BTreeNode* parent;

  BTreeNode();
  ~BTreeNode();

  int searchPos(int);
  void moveOn(int, int);
};


BTreeNode::BTreeNode() {
  n = 0;
  leaf = true;
  keys = new int[M - 1];
  pointers = new BTreeNode*[M];
  for(int i = 0; i < M; i++)
    pointers[i] = 0;
```

```
}

BTreeNode::~BTreeNode() {
  delete keys;

  delete pointers;
}
```

O método `searchPos(int value)` encontra a posição da chave com valor `value` no nó objeto receptor da mensagem no momento da invocação.

```
int BTreeNode::searchPos(int value) {
  int pos = 0;

  while(pos < n && value > keys[pos])
    pos++;

  return pos;
}
```

O método `moveOn(int initial,int final)` é responsável por movimentar (copiar) as chaves e os ponteiros da posição `initial` até a posição `final` uma posição para frente, a fim de abrir espaço para entrada da nova chave e novo ponteiro, cujas posições já foram calculadas anteriormente (antes da invocação desta método). O código referente a essa operação segue abaixo:

```
void BTreeNode::moveOn(int initial, int final) {
  for(int i = initial; i > final; i--) {
    keys[i] = keys[i - 1];
    pointers[i + 1] = pointers[i];
  }
}
```

Uma B-árvore é, então, definida basicamente por um ponteiro para um nó raiz (objeto da classe `BTreeNode` e um inteiro para guardar a quantidade de nós da árvore. A classe que define e controla a B-árvore é denominada `BTree`. O método construtor desta classe recebe como argumento o valor inteiro "order" representando a ordem da árvore a ser criada. Segue abaixo a descrição da classe `BTree`.

```
class BTree {
private:
  int numberOfNodes;
```

```
  BTreeNode* root;

  BTreeNode* search(int, BTreeNode*);
  int add(BTreeNode*, int, int*, BTreeNode*&);
  void print(BTreeNode*, int);

public:
  BTree(int order) {
    M = order;
    numberOfNodes = 0;
    root = 0;
  }

  ~BTree() {
    numberOfNodes = 0;
    delete root;
  }

  bool search(int value) {
    return search(value, root) != 0;
  }

  bool add(int);

  void print() {
    print(root, 0);
  }

};
```

3 Busca de uma chave em uma B-árvore

O método de busca de uma chave em uma B-árvore é definido em função de uma busca sequencial no vetor `keys` de cada nó da árvore, no caminho da raiz até uma folha, no pior caso. Inicialmente, o método é invocado tendo como argumento o ponteiro para a raiz da árvore e segue, recursivamente, por um caminho bem definido da raiz até uma folha, no pior caso, ou até encontrar a chave desejada. Como as chaves dentro de um nó estão ordenadas, a escolha de qual subárvore seguir para continuar a busca é simples. O método descrito abaixo ilustra a operação de busca de uma chave `value` em uma B-árvore.

```
BTreeNode* BTree::search(int value, BTreeNode* node){
```

```
  int i;
  if(node != 0) {
    for(i = 0; i < node->n && node->keys[i] < value; i++);

    if(i < node->n && node->keys[i] == value)
      return node;

    return search(value, node->pointers[i]);
  }

  return node;
}
```

4 Inserção de uma chave na B-árvore

Tanto as operações de inserção como as de remoção parecem ser desafiantes se lembramos que todas as folhas têm que estar no último nível. Implementar a inserção se torna mais fácil quando a estratégia de construir uma árvore é mudada. Uma árvore binária de busca é construída do topo para a base, resultando em árvores desbalanceadas. Uma B-árvore é construída de baixo para cima, de modo que a raiz seja uma entidade sempre em fluxo, e somente no fim de todas as inserções podemos conhecer com certeza o conteúdo da raiz.

Dada uma chave a ser inserida na B-árvore, o primeiro passo é seguir um percurso da raiz até a folha que deverá receber a nova chave. Se há espaço nesta folha, a inserção é concluída com sucesso e sem custo adicional. Caso não haja espaço na folha em questão, outra folha é criada, as chaves são divididas entre essas folhas e uma chave é promovida para o nó ascendente. Se o ascendente está cheio, o processo é repetido até que a raiz seja atingida e uma nova raiz seja criada. O processo de criar um novo nó folha, redistribuir de chaves (entre os nós folhas em questão-a folha cheia e a nova folha criada) e promover uma chave ao nó ascendente é conhecido como *cisão* de um nó.

Os métodos descritos a seguir implementam o processo de inserção em B-árvores. Inicialmente, o método *public* `bool BTree::add(int key)` é invocado e este realiza a invocação do método *private* com o mesmo que contém os argumentos necessários para controle da inserção. O valor inteiro retornado pelo `add` é testado a fim de finalizar a operação: se valor de retorno é -1 significa que a inserção não ocorreu porque a chave `key` já existia na árvore; se o retorno foi 1, significa que a inserção ocorreu com sucesso e nenhuma atualização na raiz deve ser realizada; finalmente, se o retorno foi 0, uma cisão da raiz será realizada e o atributo *root* será atualizado.

```
bool BTree::add(int key) {
```

```
  BTreeNode* newNode;
  BTreeNode* cur;
  int upKey;
  int ret = add(root, key, &upKey, newNode);

  switch(ret) {
    case -1:
      return false;

    case 0:
      cur = root;
      root = new BTreeNode();
      root->n = 1;
      root->keys[0] = upKey;
      root->pointers[0] = cur;
      root->pointers[1] = newNode;

    default:
      return true;
  }
}



int BTree::add(BTreeNode* node, int value, int* upValue, BTreeNode*& newNode) {
  BTreeNode* lastNode;

  int newValue, lastValue;

  // caso onde um ponteiro nulo é encontrado
  // as variáveis utilizadas como parâmetros de saída são inicializadas
  if(node == 0) {
    newNode = 0;
    *upValue = value;

    return 0;
  }

  // deve-se verificar se a chave procurada está armazenada no nó corrente
  int nKeys = node->n;
  int position = node->searchPos(value);

  // caso encontre a chave, a busca termina
  if(position < nKeys && value == node->keys[position])
    return -1;
```

```
// caso contrário, realiza chamada recursiva de add
int ret = add(node->pointers[position], value, &newValue, newNode);

// se add retornou -1 ou 1, não há nada a fazer além de retonar
if(ret != 0)
  return ret;

// caso contrário, é necessário tratar uma cisão que ocorreu na chamada
// anterior
// se ha espaco para a chave promovida, esta é inserida no nó corrente
// e a inserção é finalizada com sucesso

if(nKeys < M - 1) {
  position = node->searchPos(newValue);

  node->moveOn(node->n, position);

  node->keys[position] = newValue;
  node->pointers[position + 1] = newNode;
  node->n++;

  return 1;
}
// caso contrário, é necessário realizar a cisão do nó corrente
if(position == M - 1) {
  lastValue = newValue;
  lastNode = newNode;
}
else {
  lastValue = node->keys[M - 2];
  lastNode = node->pointers[M - 1];

  node->moveOn(M - 2, position);

  node->keys[position] = newValue;
  node->pointers[position + 1] = newNode;
}
// um novo nó é criado e as chaves do nó corrente são redistribuídas
// a chave medina é promovida e add deve retornar 0 para indicar
// que essa cisão deve ser tratada no nível acima da árvore
int mid = (M - 1)/2;
*upValue = node->keys[mid];

newNode = new BTreeNode();
```

```
  node->n = mid;
  newNode->n = M - 1 - mid;

  for(int i = 0; i < newNode->n; i++) {
    newNode->pointers[i] = node->pointers[i + mid + 1];

    if(i < newNode->n - 1)
      newNode->keys[i] = node->keys[i + mid + 1];
    else
      newNode->keys[i] = lastValue;
  }

  newNode->pointers[newNode->n] = lastNode;

  return 0;
}
```

A figura 2 ilustra a inserção de dois elementos. No primeiro (inserção do elemento 45) não há necessidade de cisão visto que o nó folha onde a chave foi inserida não estava cheia antes da inserção. No segundo caso (inserção do elemento 36), as chaves 35,37,38 e 39 que pertencia ao nó onde o novo elemento deveria ser inserido, juntamente com a chave 36 (nova chave) foram divididos de acordo com um valor mediano (no caso, o elemento 37): as chaves menores que a mediana permanecem no mesmo nó; a mediana é inserida no nó pai; os elementos maiores que a mediana são movidos para um novo nó criado durante o processo de cisão. Seja i a posição da mediana no vetor `keys` do nó pai, o ponteiro `pointers[i+1]` passará a apontar para o novo nó alocado.

5 Remoção de uma chave na B-árvore

A remoção é em grande parte o inverso da inserção, embora tenha mais casos especiais. É preciso tomar cuidado para evitar que qualquer nó esteja com menos do que o mínimo de chaves permitido depois de uma remoção. Isso significa que em alguns casos haverá redistribuição de chaves entre nós e quando for inevitável, a *fusão* de dois nós.

Na remoção existem dois casos principais: remover uma chave de uma folha e remover uma chave de um nó não folha. Segue a descrição dos casos da remoção:

1. Remover uma chave de uma folha:

   (a) Se depois de remover uma chave k, a folha satisfaz todas das propriedades da B-árvore, apenas uma reorganização das chaves dentro do nó será necessária para "ocupar" o espaço deixado pela chave removida.

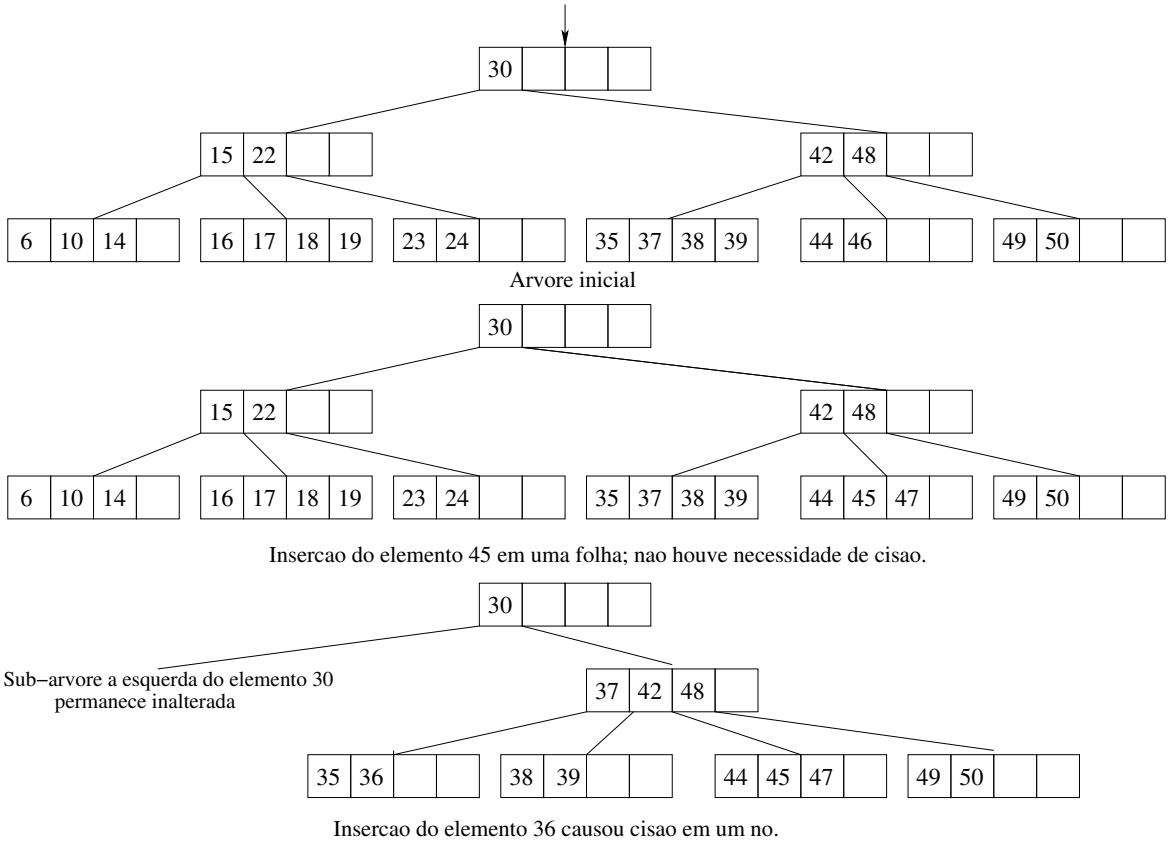

Figura 2: Exemplos de operações de inserção de chaves em uma B-árvore de ordem 5.

(b) Senão, dizemos que há uma *subutilização* do nó (ele possui menos do que o mínimo de chaves estipulado).

i. Se existe um irmão (à esquerda ou à direita) com o número de chaves maior do que o mínimo exigido, então todas as chaves da folha que continha o nó k e do seu irmão são *redistribuídas* entre eles. Move-se uma chave do irmão para o pai e a chave separadora do pai até o nó folha, que passará a ter o mínimo de chaves exigido.

ii. Se não existe irmão (à esquerda ou à direita) com o número de chaves maior do que o mínimo exigido, será necessário realizar *fusão* de nós. As chaves do nó folha de onde k foi removido, as chaves de um irmão e a chave separadora do pai são unidos em uma folha e a outra é descartada. A remoção de uma chave do pai pode ter causado subutilização deste nó e os passos para tratamento são mais uma vez executados a fim de corrigir o problema.

- Caso particular da raiz: no caso em que a raiz tenha uma única chave e que seja necessário realizar fusão dos seus dois filhos, a fusão "englobará" a raiz, ou seja, os nós filhos e a raiz serão fundidos em um único nó que passará a ser a raiz da árvore.

2. Remover uma chave de um nó não-folha: O nó removido será substituído por seu sucessor imediato na árvore, que certamente estará em uma folha. Uma vez encontrado o sucessor, este sobrescreverá a chave para a qual foi solicitada remoção e esse sucessor será removido da folha. Esse processo resume-se, então, nos mesmos casos de tratamento anterior (remoção de uma chave em um nó folha).

A figura 3 mostra remoção de três chaves. No primeiro caso (remoção do elemento 17), foi necessário apenas reorganizar as chaves dentro do nó para que o espaço deixado pelo chave removida seja preenchido. No segundo caso (remoção do elemento 49), a remoção deixou o nó com menos do que a quantidade mínima de chaves. Como o seu nó irmão possuia mais do que a quantidade mínima de chaves, foi possível redistribuir as chaves entre o nó irmão e o nó pai de tal forma a acrescentar mais uma chave ao nó onde foi feita a remoção sem a necessidade de modificar a estrutura dos ponteiros dos nós envolvidos. No último caso (remoção do elemento 50), foi necessário realizar a fusão do nó que continha o valor 50 com o seu nó irmão (que tinha exatamente o mínimo possível de chaves). Essa fusão utilizou a chave separadora (47) do nó pai e, sendo assim, fez com que o nó que continha o valor 47 ficasse com menos do que o número mínimo de nós. Foi necessário, então, realizar a fusão deste nó com seu irmão. Essa fusão utilizou a única chave da raiz, fazendo com que o nó criado fosse considerado a nova raiz da árvore.

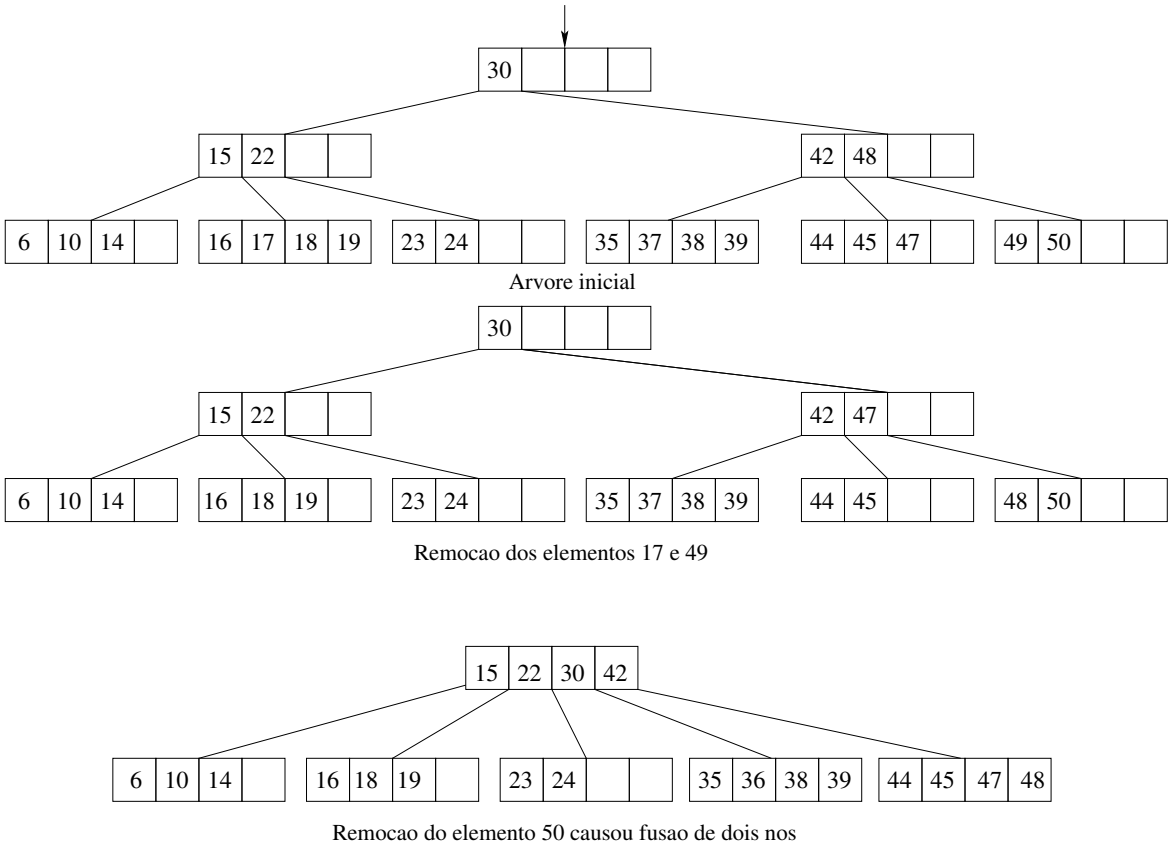

Figura 3: Exemplos de operações de remoção de chaves em uma B-árvore de ordem 5.