INF 1010
Estruturas de Dados Avançadas

Árvores B

árvores B

Motivação

Tornar a busca mais eficiente

considerando também os dispositivos de armazenamento de memória

Limitação da árvore binária de busca

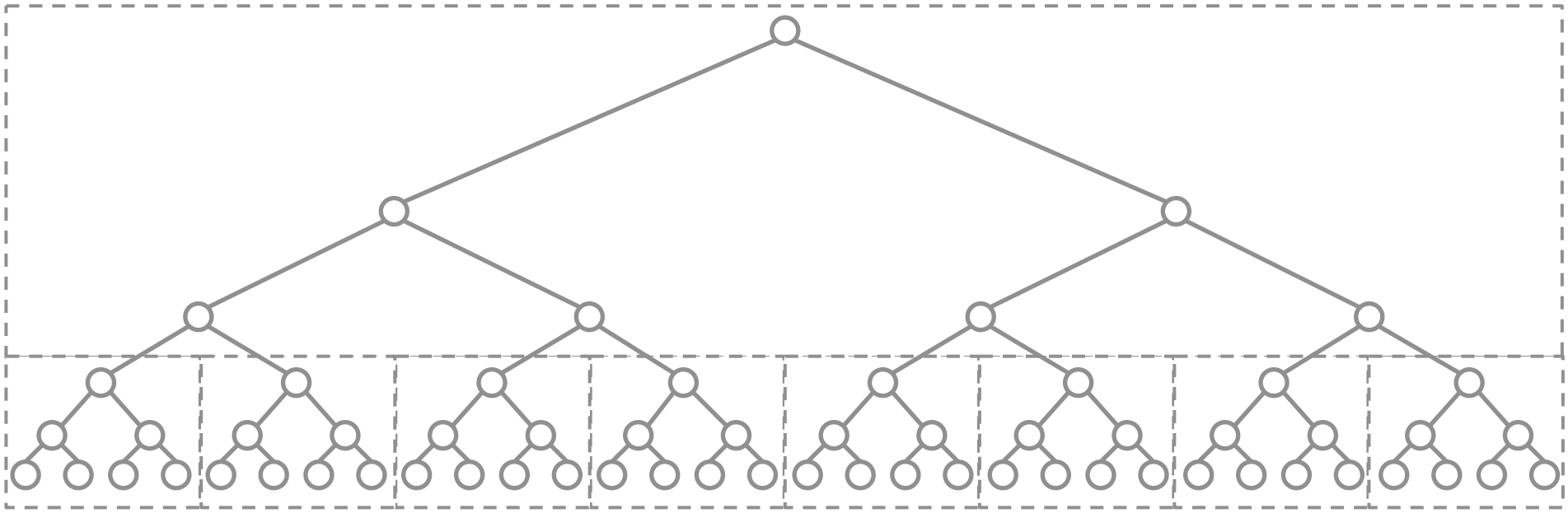
cada nó é lido individualmente

(e o acesso a memória secundária é lento)

Árvore B - definição

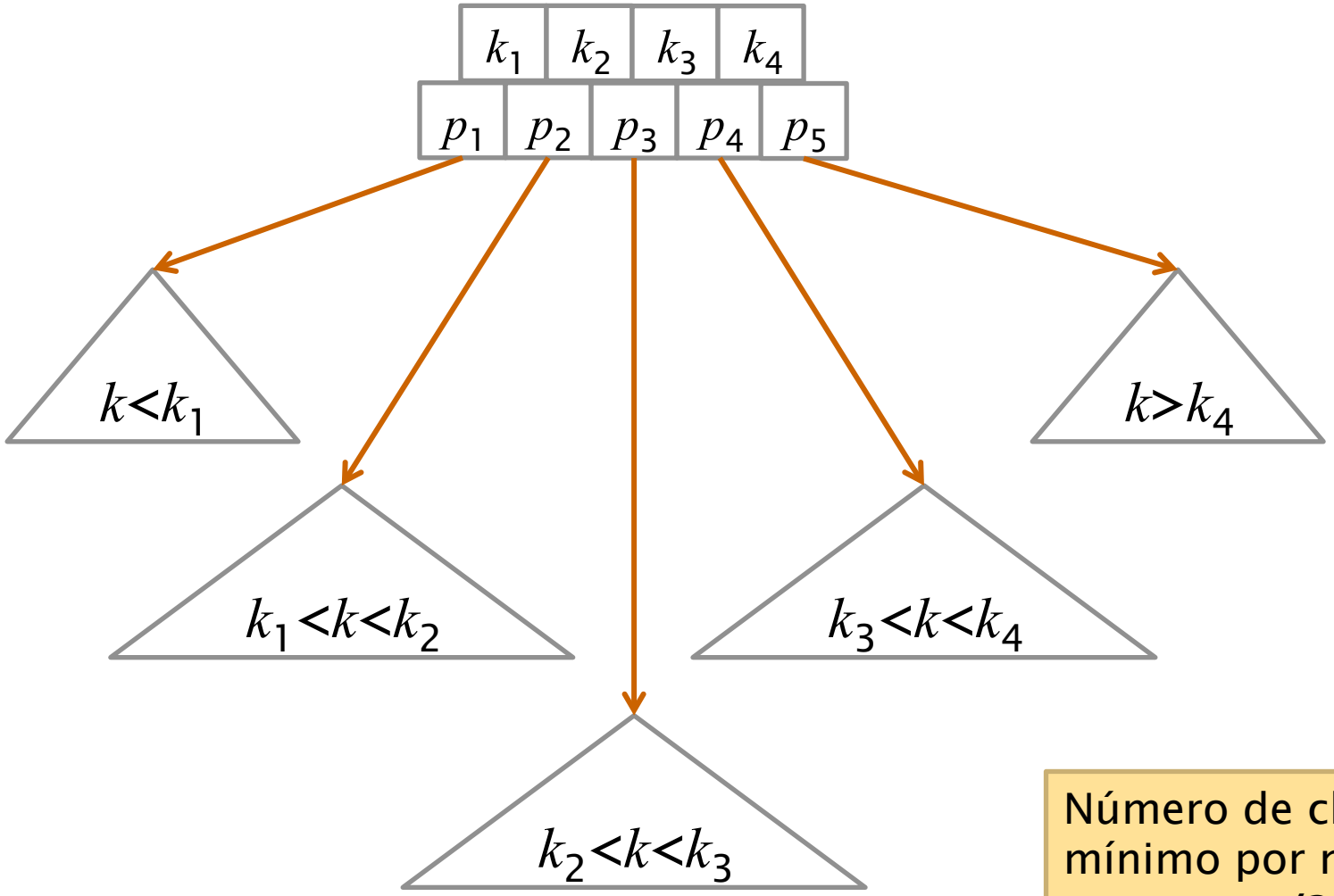
cada acesso à memória secundária traz um grupo de elementos

sub-árvores são divididas em **páginas**



Árvore B – idéia básica

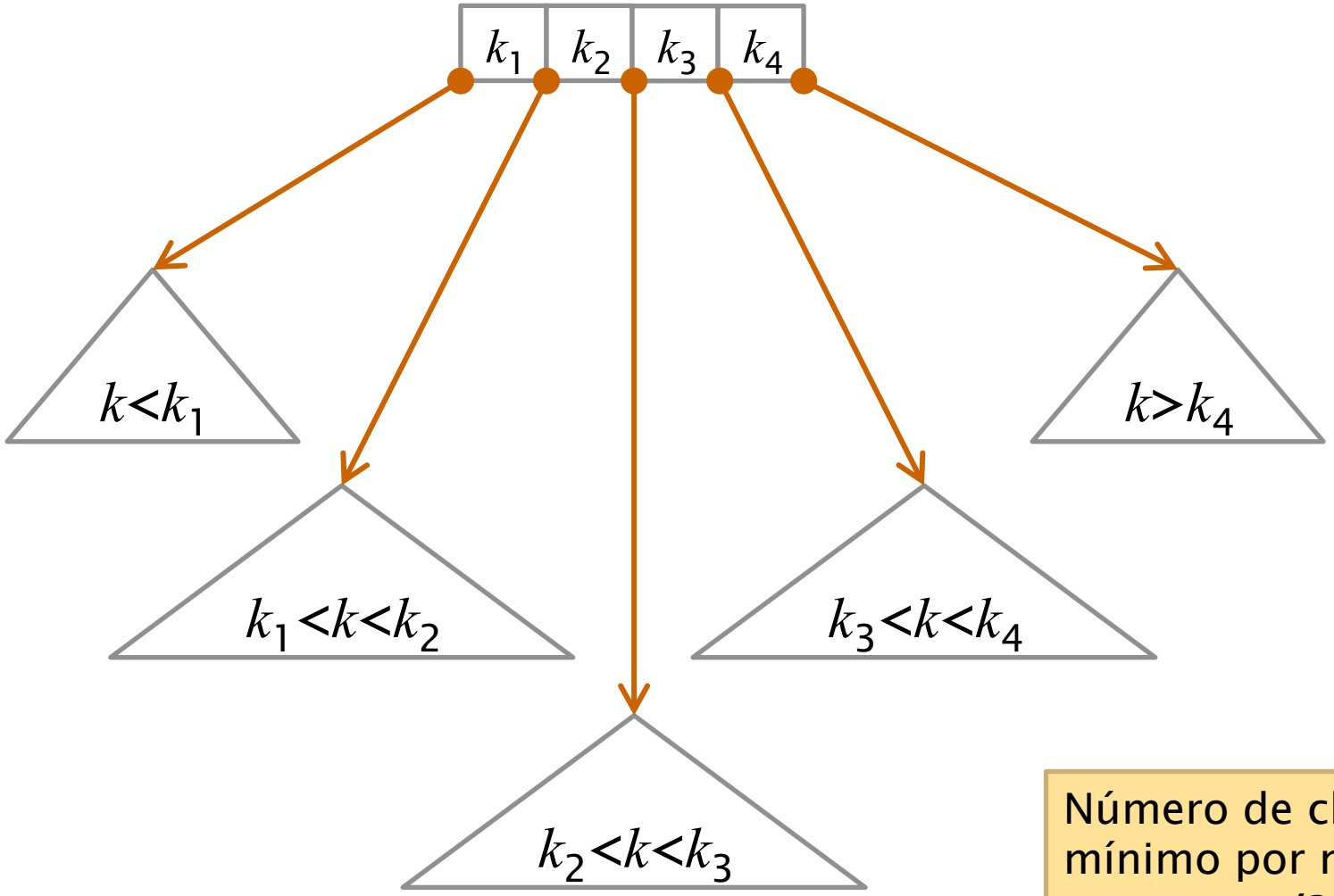
Árvore n-ária com chaves de busca nos nós



Número de chaves mínimo por nó:
n/2

Árvore B – idéia básica

Árvore n-ária com chaves de busca nos nós

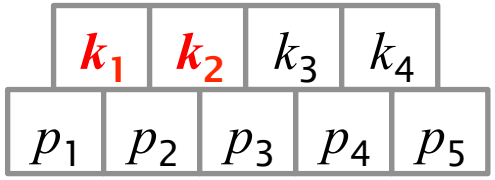


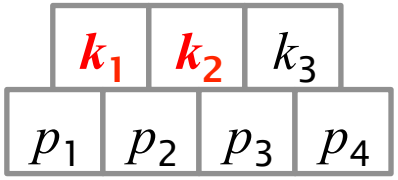
Número de chaves mínimo por nó:
n/2

Árvore B - definição
(Bayer & McCreight, 1972; Comer, 1979)

árvore B de ordem p

- todo nó tem no máximo 2p+1 filhos
- cada nó (exceto a raiz e as folhas) possui no mínimo p+1 filhos
- todas as folhas aparecem no mesmo nível





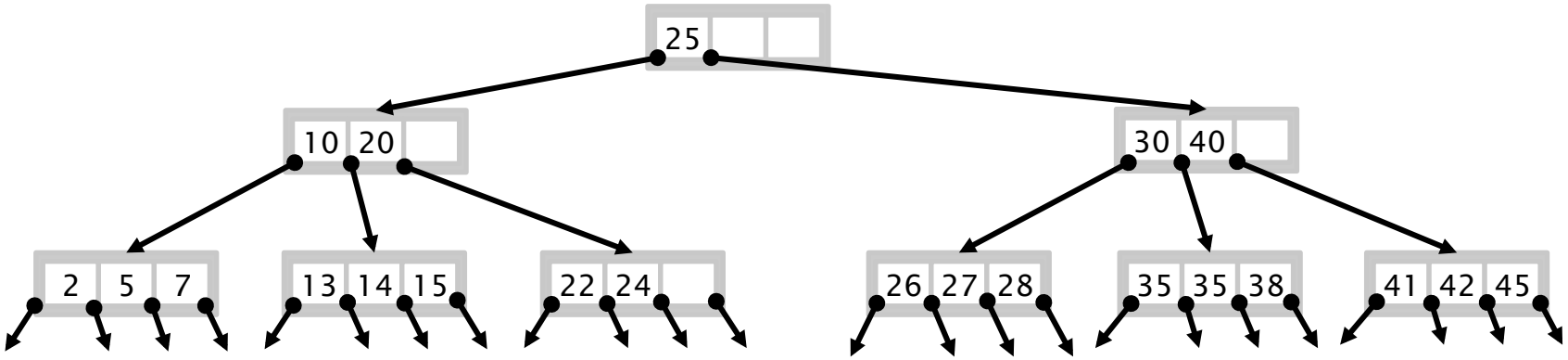
Árvore B – definição (Knuth, 1997)

Os slides a seguir levam em consideração esta definição.

árvore B de ordem m

- todo nó (página) tem no máximo m filhos
- cada nó (exceto a raiz e as folhas) possui no mínimo m/2 filhos
- a raiz possui ao menos 2 filhos (a menos que seja folha)
- um nó não terminal de k filhos possui k-1 chaves
- todas as folhas aparecem no mesmo nível

exemplo: árvore B de ordem 4 (árvore 2-3-4)



Árvore B – busca

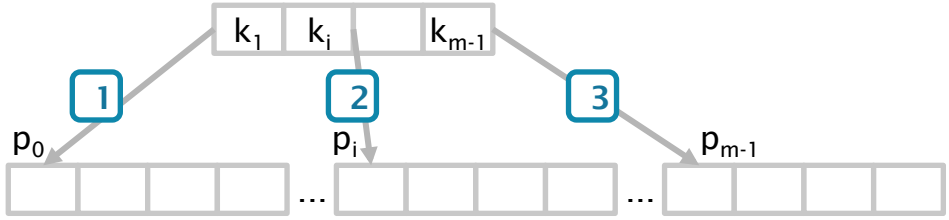
busca entre as chaves de uma página

k_1 ... k_{m-1} (se m for grande: busca binária)

se não for encontrada na página:

1. $x < k_1$ → busca deve continuar na página p_0
2. $k_i < x < k_{i+1}$ para $1 \leq i < m-1$ → busca deve continuar na página p_i
3. $k_{m-1} < x$ → busca deve continuar na página p_{m-1}

se não houver páginas abaixo da atual, a chave não existe



Árvore B de ordem m - inserção

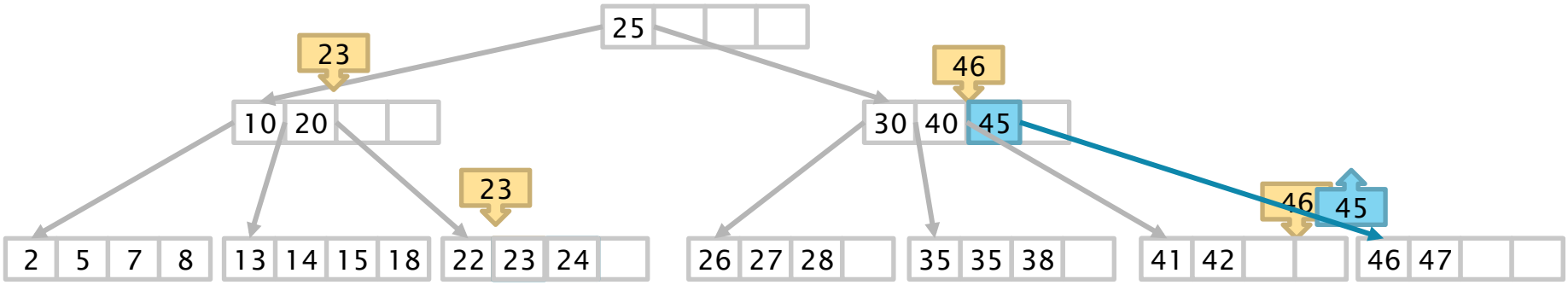
seja p_i a página onde x deverá ser inserido

se p_i tiver menos de m-1 elementos

1. insere em p_i, na posição adequada

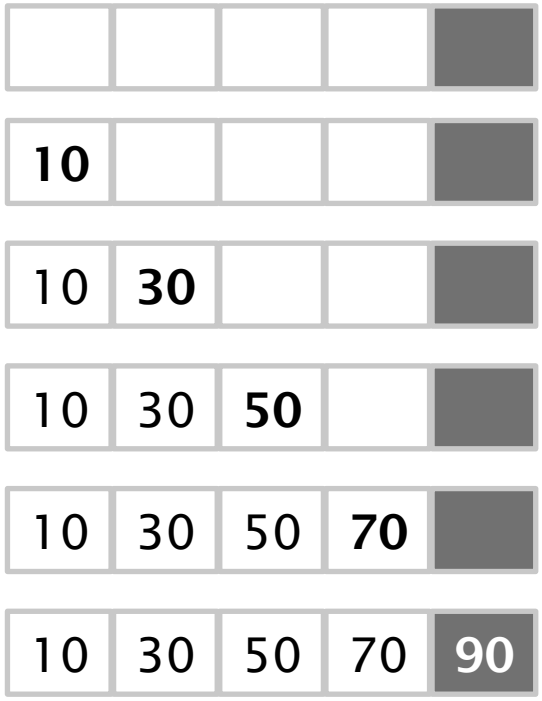
se página p_i já estiver lotada:

1. aloca uma nova página p_k
2. distribui as m chaves da seguinte maneira:
   1. $\lceil m/2 \rceil$-1 menores chaves em p_i
   2. m-$\lceil m/2 \rceil$ maiores chaves em p_k
   3. insere a chave mediana (em $\lceil m/2 \rceil$) na página superior (se página p_i for raiz: cria nova raiz com a mediana)

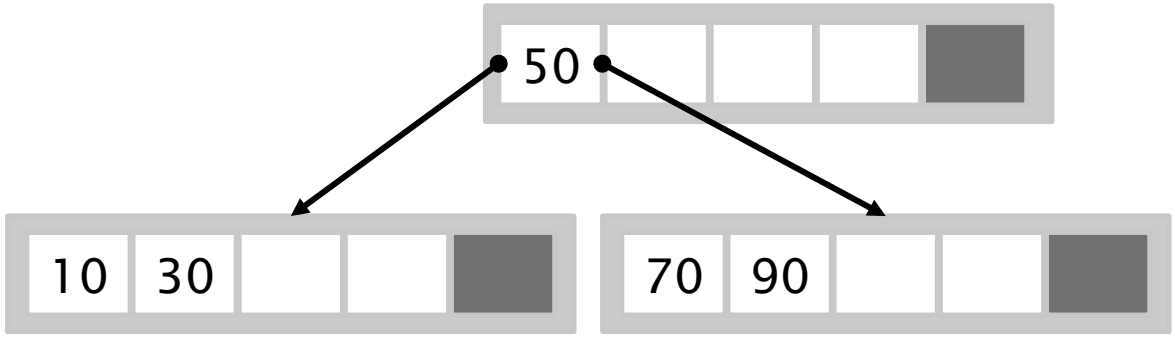


exemplo
(split em caso de overflow)

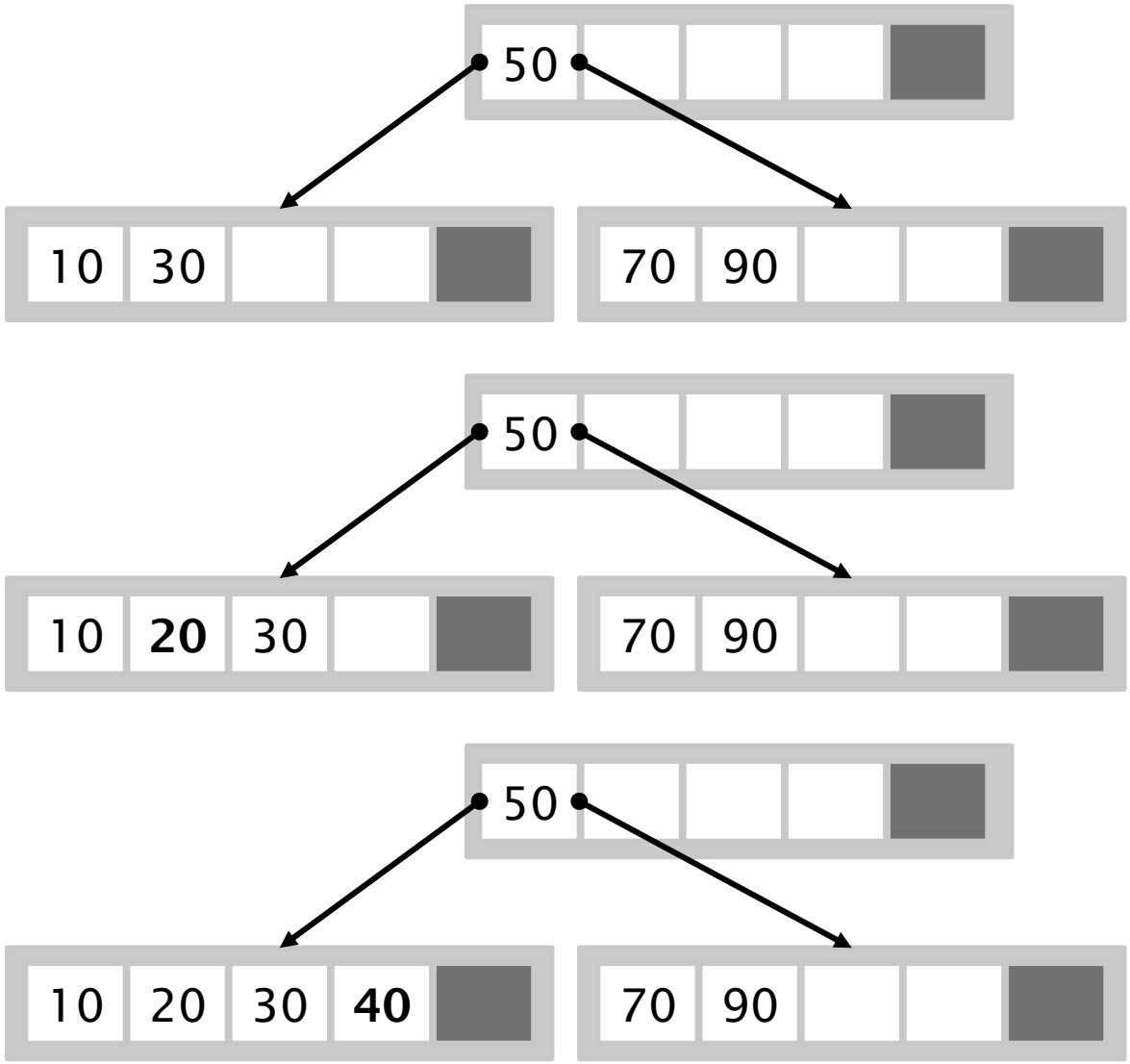
insere 10, 30, 50, 70, 90 (ordem 5)



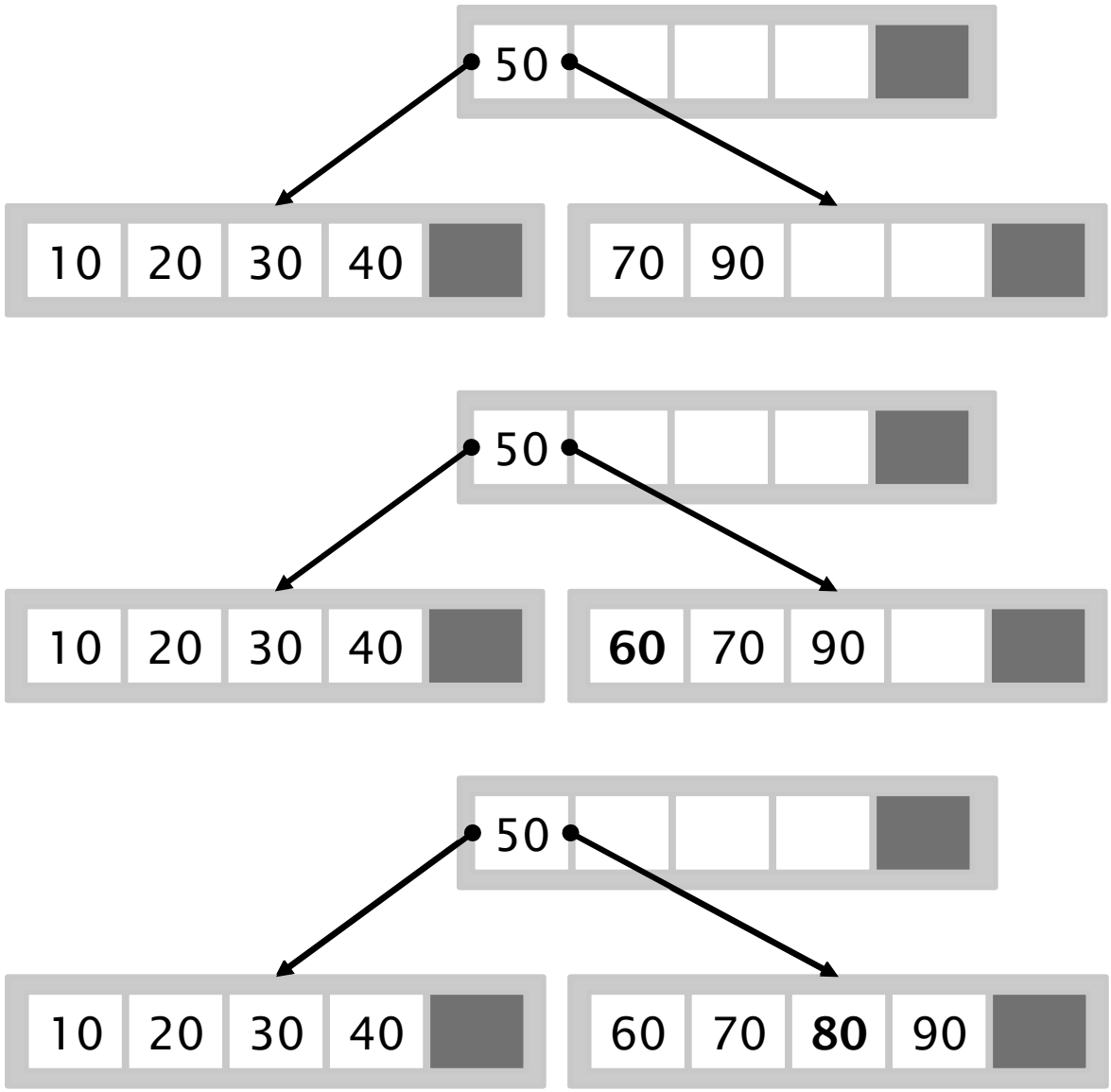
overflow: split



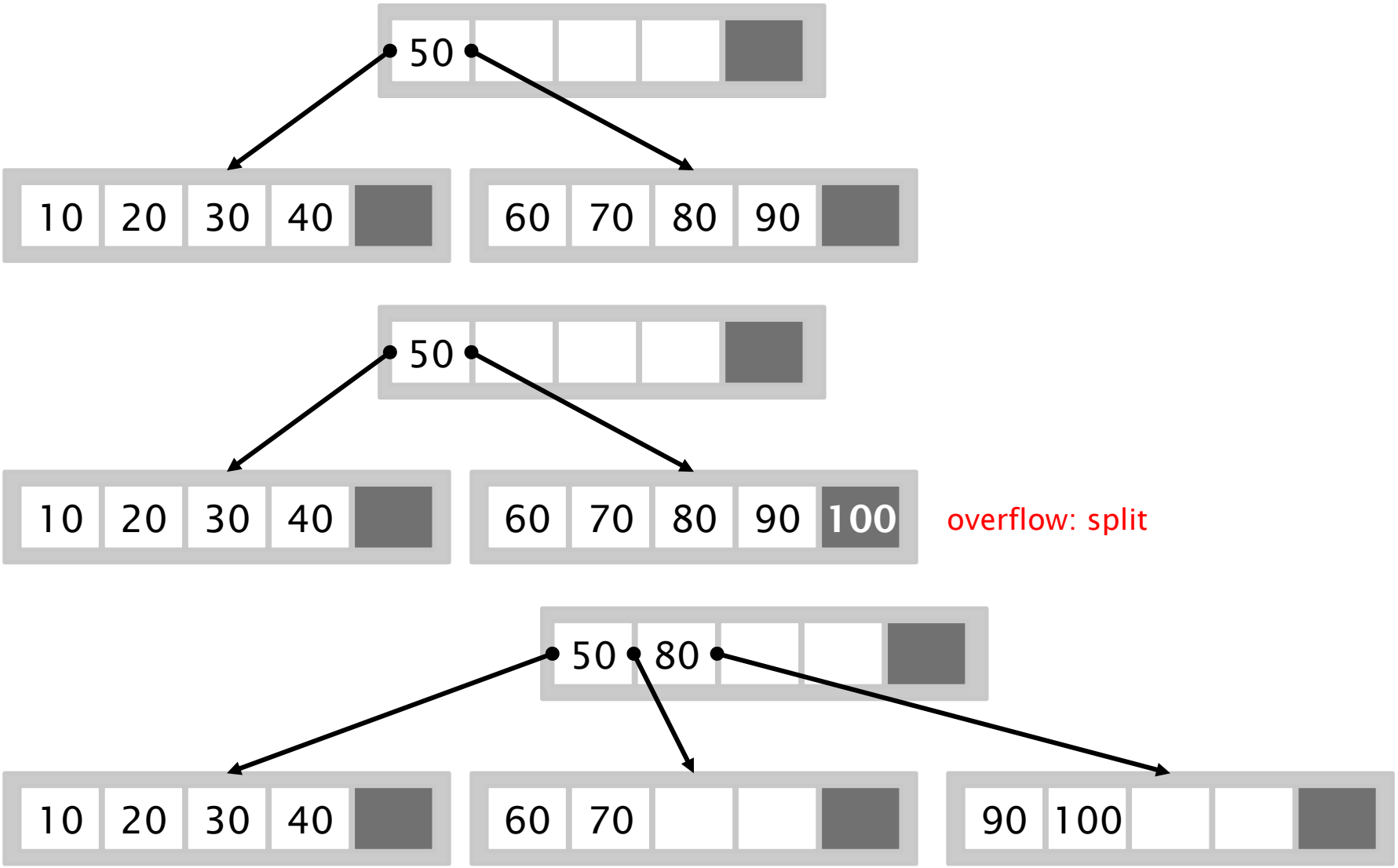
insere 20, 40 (ordem 5)



insere 60, 80 (ordem 5)



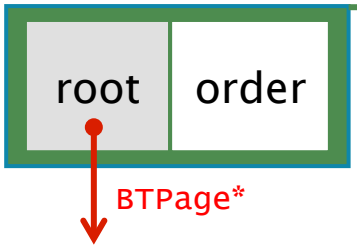
insere 100 (ordem 5)



dúvidas?

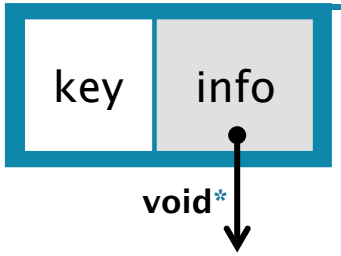
Árvore B – Estruturas

BTree




```
struct _btree {
   BTPage* root;
   int order;
};
typedef struct _btree BTree;
```

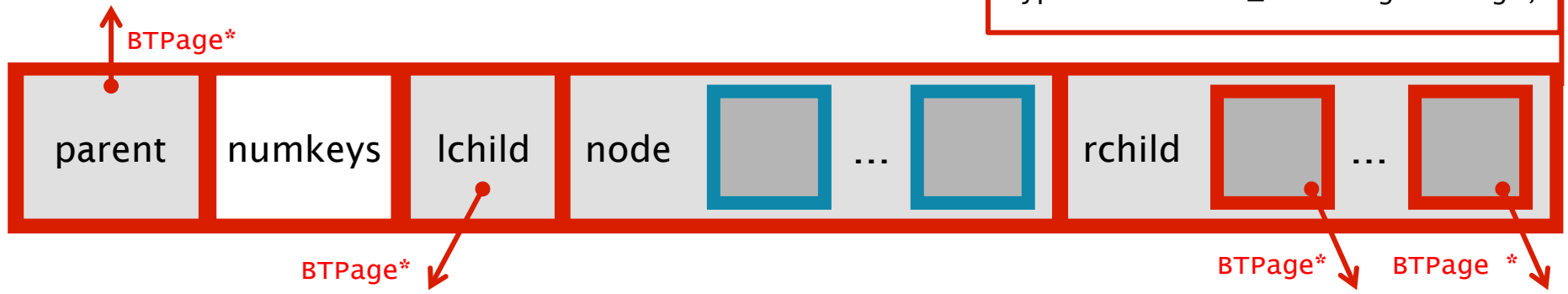
BTNode




```
struct _btreeNode {
   int key;
   void* info;
};
typedef struct _btreeNode BTNode;
```

```
struct _btreePage {
   BTPage*  parent;
   int      numkeys;
   BTPage*  lchild;
   BTNode** node;
   BTPage** rchild;
};
typedef struct _btreePage BTPage;
```

BTPage



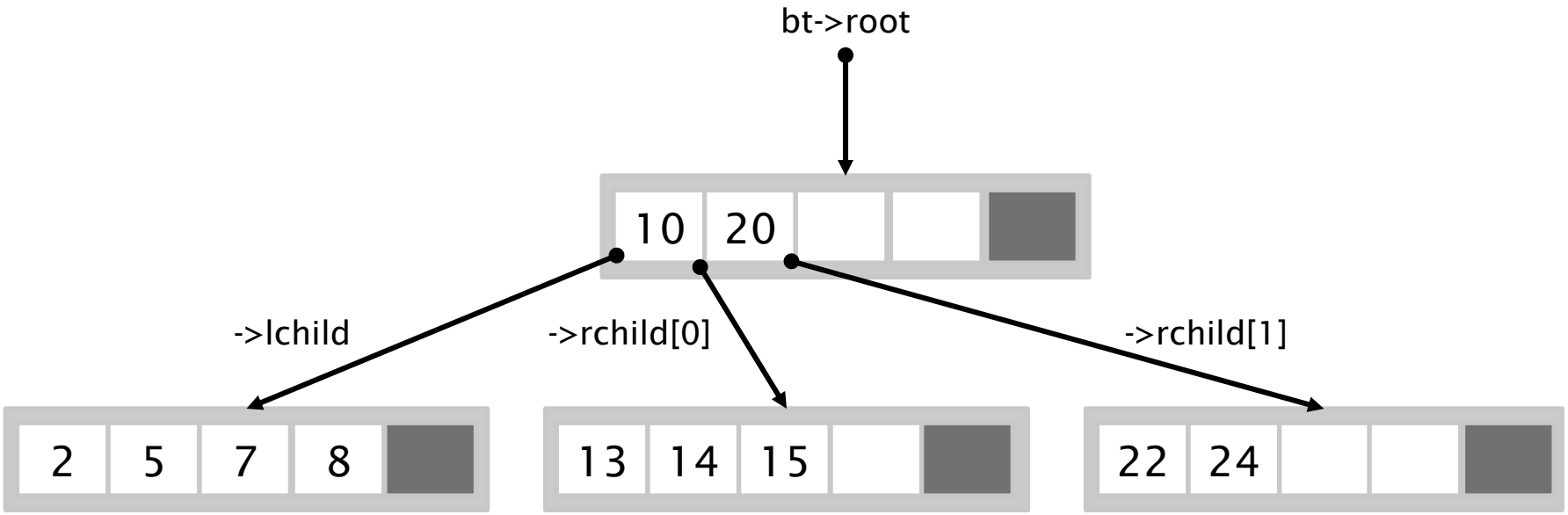
btree.h

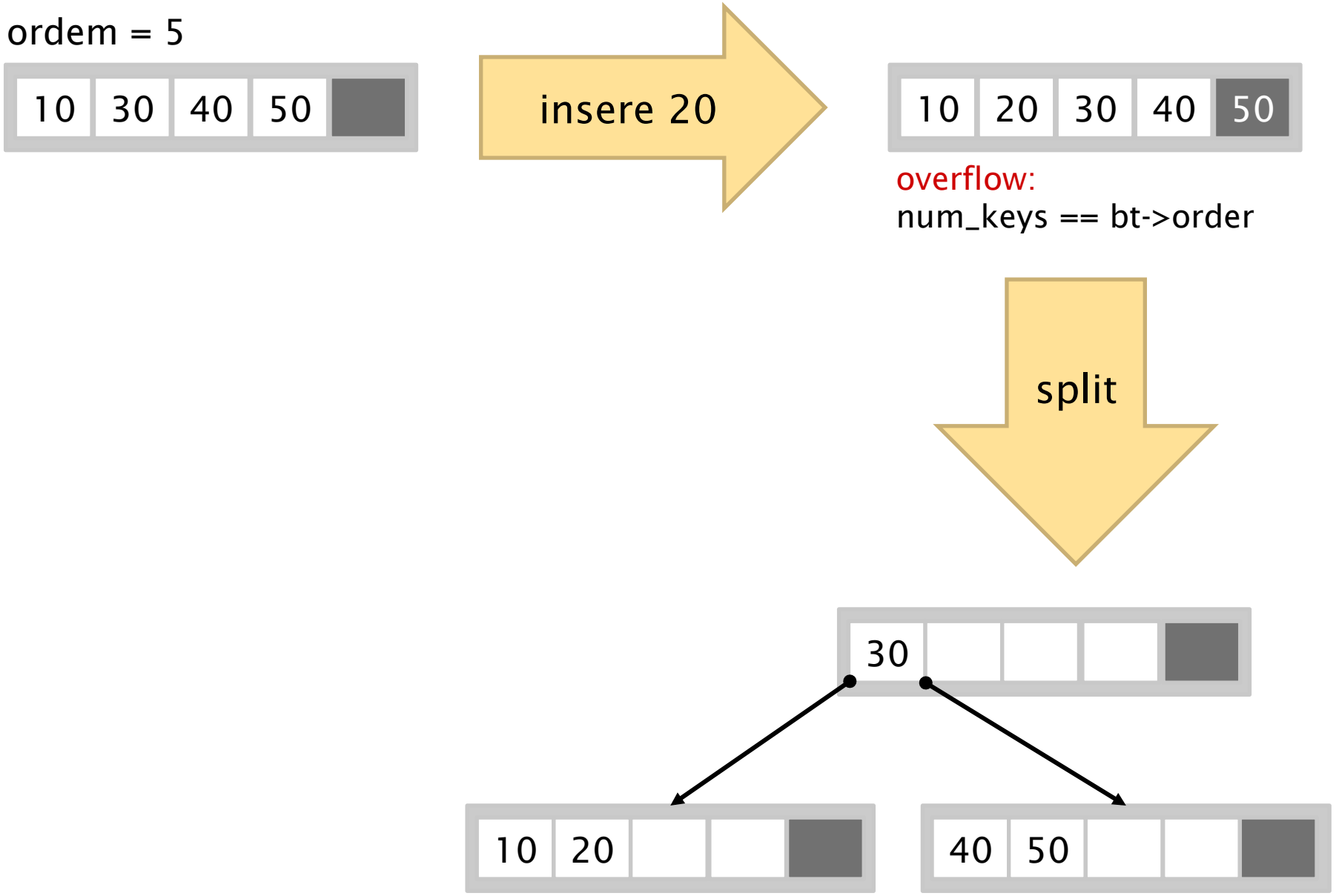
```
typedef struct _btree BTree;


BTree* btree_create(int order);
void   btree_destroy(BTree* bt, void(*cb_destroy)(void*));
void   btree_insert(BTree* bt, int key, void* info);
void   btree_remove(BTree* bt, int key);
void   btree_print_indent(BTree* bt, void (*cb_print)(const void*));
void*  btree_find(BTree* bt, int key);
```

```
static BTPage* btree_page_create_empty(BTree* bt, BTPage* pgParent)
{
   int i;
   /* allocate memory for page structure */
   BTPage* pgCurrent = (BTPage*)malloc(sizeof(BTreePage));
   /* initialize empty page (with 0 elements) */
   pgCurrent->numkeys = 0;
   /* set page parent */
   pgCurrent->parent = pgParent;
   /* allocate memory for nodes (key + info) and children
      (allocate additional node and pointer to rchild to hold temporary overflow) */
   pgCurrent->node = (BTNode**) malloc(bt->order * sizeof(BTNode*));
   pgCurrent->rchild = (BTPage**) malloc((bt->order) * sizeof(BTPage*));
   /* initialize nodes and children pointers */
   pgCurrent->lchild = NULL;
   for (i=0; i < bt->order; ++i)
   {
      pgCurrent->node[i] = NULL;
      pgCurrent->rchild[i] = NULL;
   }
   return pgCurrent;
}
```

Para facilitar, alocamos espaço a mais para usar a mesma página quando houver overflow.


bt->root
10
20
->lchild
->rchild[0]
->rchild[1]
2
5
7
8
13
14
15
22
24


ordem = 5
10
30
40
50
insere 20
10
20
30
40
50
overflow:
num_keys == bt->order
split
30
10
20
40
50

```
static BTPage* btree_insert_aux(BTree* bt, BTPage* pgCurrent, BTNode* btNode) {
   int i;
   if (pgCurrent == NULL) { /* create new page */
      BTPage* pgNew = btree_page_create(bt, NULL, btNode);
      return pgNew;
   }
   /* find out where info should be inserted
      (if order is large, a binary search should be used) */
   for (i=0; i < pgCurrent->numkeys; ++i)
      if (btNode->key == pgCurrent->node[i]->key) /* if key exists, update info */
      {
         pgCurrent->node[i]->info = btNode->info; /* memory leak */
         return pgCurrent;
      }
      else if (btNode->key < pgCurrent->node[i]->key)
         break;
   if (pgCurrent->lchild == NULL)  /* if leaf, insert here */
      btree_page_insert_node_at(pgCurrent, btNode, NULL, i);
   else if (i == 0)
      pgCurrent = btree_insert_aux(bt,  pgCurrent->lchild, btNode);
   else
      pgCurrent = btree_insert_aux(bt,  pgCurrent->rchild[i-1], btNode);
   if (btree_page_overflowed(bt, pgCurrent)) {
         pgCurrent = btree_page_split(bt, pgCurrent);
   }
   return pgCurrent;
}

void btree_insert(BTree* bt, int key, void* info)
{
   BTNode* btNode = btree_node_create(key, info);
   BTPage* pgCurrent = btree_insert_aux(bt, bt->root, btNode);
   if (bt->root != pgCurrent)
      bt->root = pgCurrent;
}
```

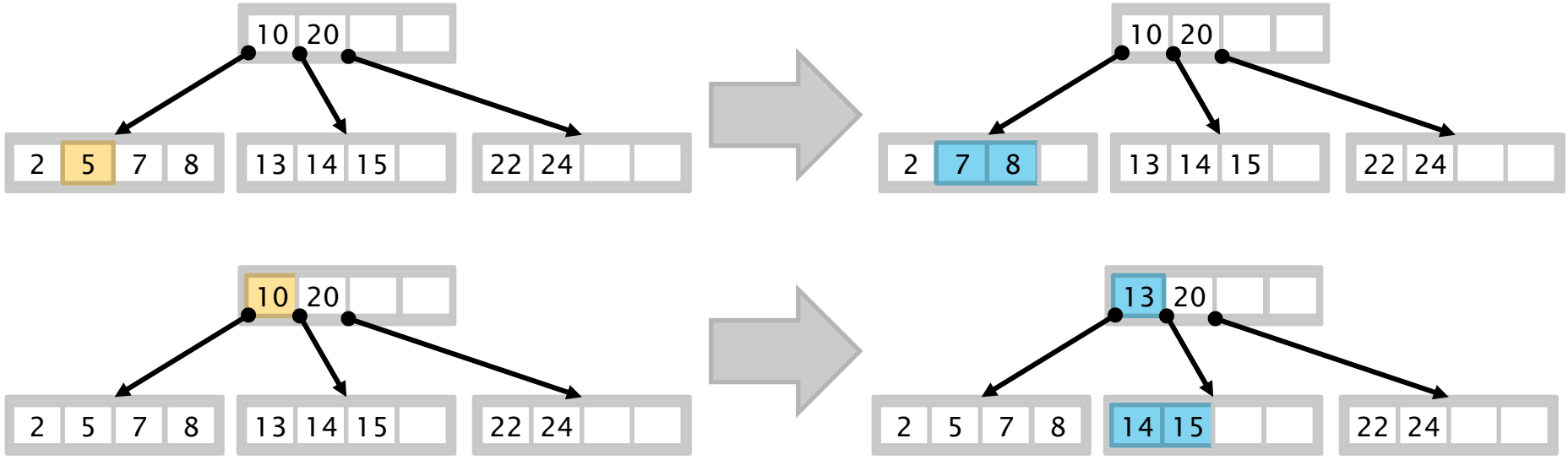
```
static BTPage* btree_page_split(BTree* bt, BTPage* pgCurrent) {
   BTPage* pgParent = pgCurrent->parent;
   BTPage* pgNew = btree_page_create_empty(bt, pgParent);
   int i, j;
   int idxMedian = (bt->order+1)/2 - 1;
   BTNode* nodeMedian = pgCurrent->node[median];
   pgNew->lchild = pgCurrent->rchild[median];
   pgNew->parent = pgParent;
   if (pgNew->lchild) pgNew->lchild->parent = pgNew;
   if (pgParent == NULL) { /* create new root */
      pgParent = btree_page_create(bt, NULL, nodeMedian);
      pgParent->lchild = pgCurrent;
      pgParent->rchild[0] = pgNew;
      pgCurrent->parent = pgParent;
      bt->root = pgParent;
   }
   else { /* insert median node into parent */
      btree_page_insert_node_at(pgParent, nodeMedian, pgNew,
         btree_page_find_insertion_point(pgParent, nodeMedian));
   }
   /* move nodes right of median to new page */
   for (j=0, i = idxMedian + 1; i < pgCurrent->numkeys; ++i, ++j)    {
      pgNew->node[j] = pgCurrent->node[i];
      pgNew->rchild[j] = pgCurrent->rchild[i];
      if (pgNew->rchild[j]) pgNew->rchild[j]->parent = pgNew;
      ++(pgNew->numkeys);
      pgCurrent->node[i] = NULL;
      pgCurrent->rchild[i] = NULL;
   }
   /* remove median node and fix number of keys*/
   pgCurrent->node[median] = NULL;
   pgCurrent->rchild[median] = NULL;
   pgCurrent->numkeys = idxMedian;
   return pgParent;
}
```

dúvidas?

Árvore B de ordem m - remoção

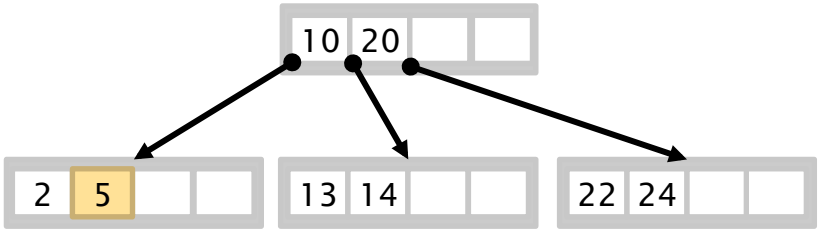
deve ser realizada em um nó folha

1. se o item a ser removido não estiver em um nó folha, substitua-o pelo maior item da sua sub-árvore à esquerda, ou pelo menor da sua sub-árvore à direita.



Remoção de chave em uma folha

2. Quando a chave é uma folha, ela será removida e deverá verificar se a folha ficará com menos de m/2 chaves. Se isso acontecer, deverá ser feita uma concatenação ou uma redistribuição.

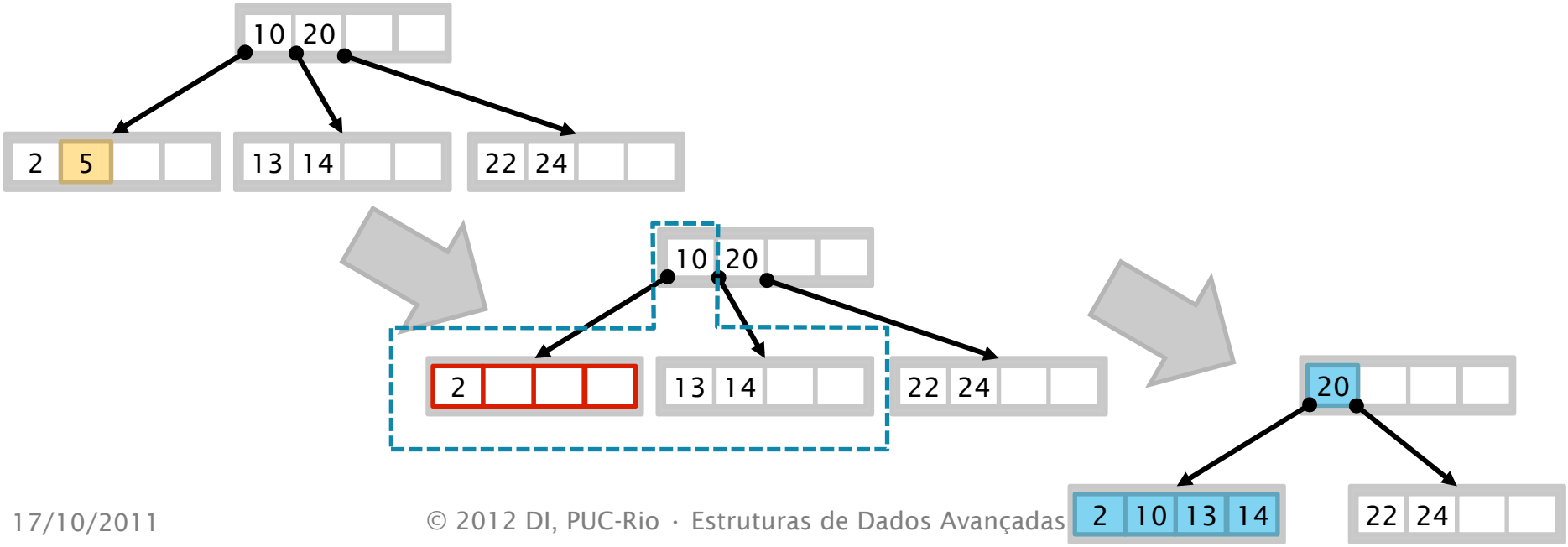


Concatenação

Acontece quando, após a remoção, a página onde a chave foi removida e uma página adjacente possuem em conjunto menos de m chaves.

Concatene essa página com uma adjacente. A chave do pai que estava entre elas fica na página que foi concatenada.

Se esse procedimento resultar em uma página com menos de m/2 chaves, faça novamente o mesmo procedimento, podendo chegar até a raiz.

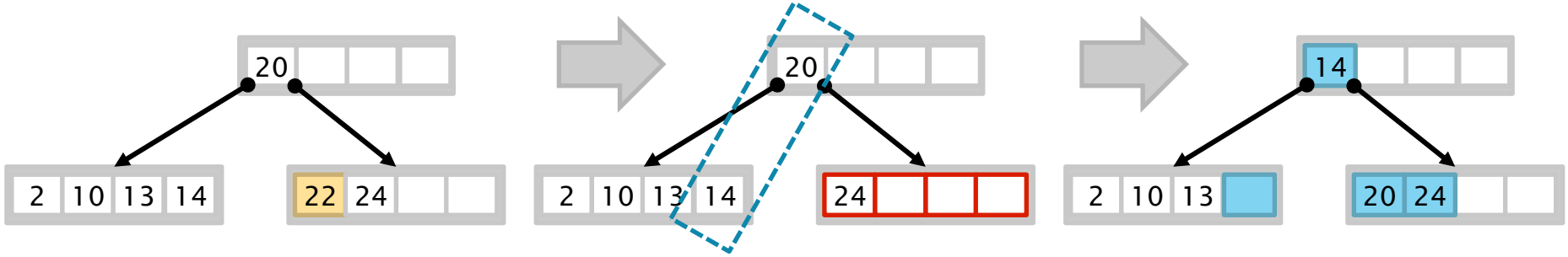


Redistribuição

Acontece quando, após a remoção, a página onde a chave foi removida e uma página adjacente possuem em conjunto m chaves ou mais.

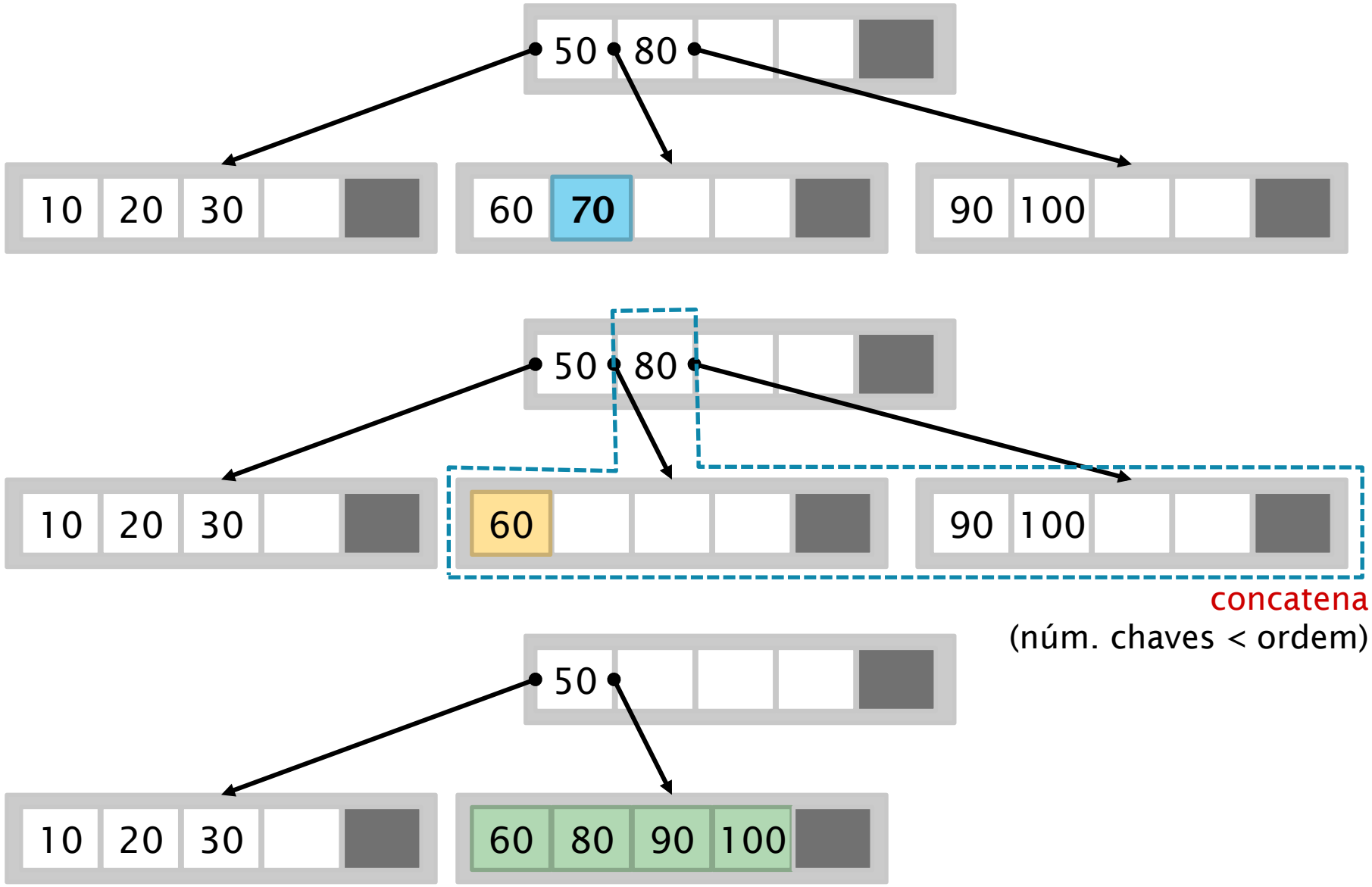
Mova a chave da página pai (“entre” as páginas adjacentes) para a página deficiente, e a chave da página adjacente* para a página pai.

Não é propagável, pois o número de chaves do pai não muda.

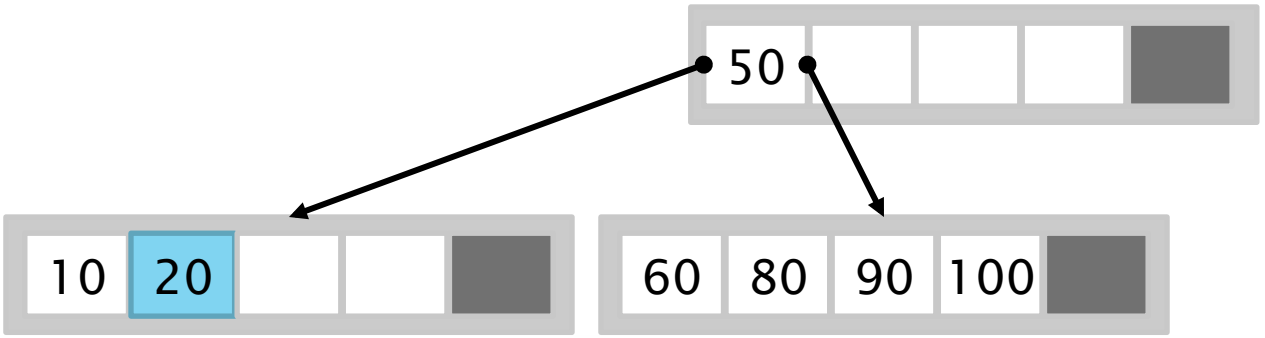


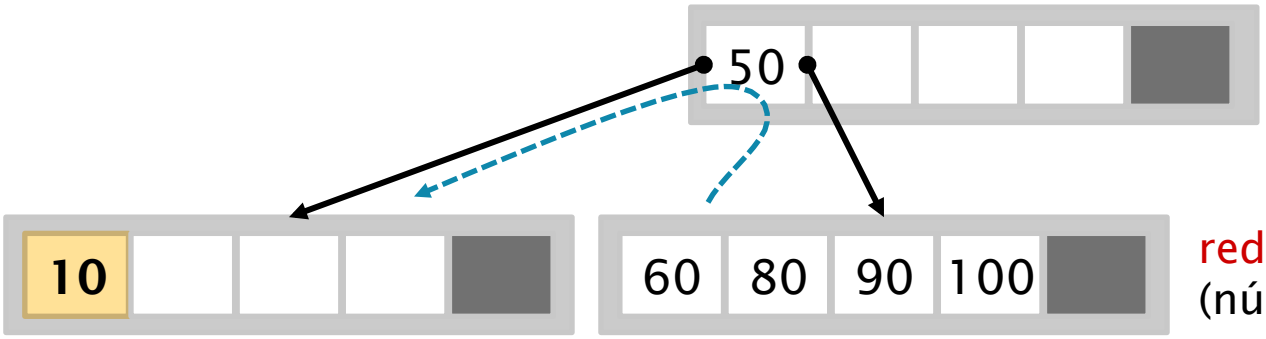
* Se a página adjacente estiver à esquerda da página deficiente, a chave movida é a maior daquela página (*borrow from left*).
Se a página adjacente estiver à direita da página deficiente, a chave movida é a menor daquela página (*borrow from right*).

remove 70 (ordem 5)

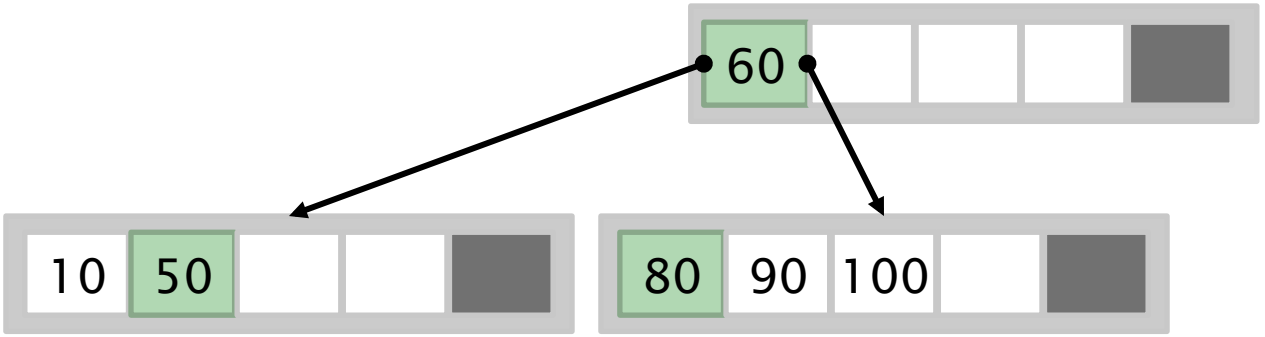


remove 20 (ordem 5)





redistribui
(núm. chaves dir > ordem/2)



dúvidas?

árvores B+

Árvores B+ Definição

número *n* de filhos de um nó de ordem *m*

$$\lceil m/2 \rceil \leq n \leq m$$

nós internos

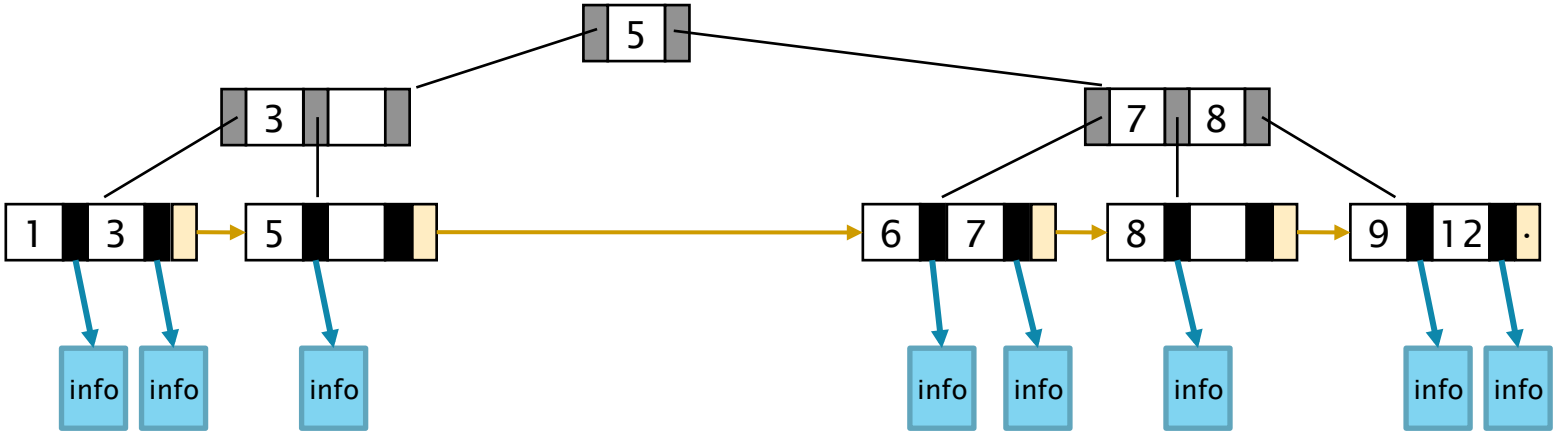
possuem apenas chaves

nós-folha

possuem chaves + dados

formam uma lista (duplamente) encadeada

Árvores B+ Exemplo



dúvidas?