Universidade Federal do Espírito Santo
Centro de Ciências Exatas, Naturais e de Saúde – CCENS-UFES
Departamento de Computação

Busca e Ordenação em Memória Secundária
Estrutura de Dados II

COM10078 - Estrutura de Dados II

Prof. Marcelo Otone Aguiar
marcelo.aguiar@ufes.br
www.marceloaguiar.pro.br

Conteúdo Programático da Disciplina

- Busca e Ordenação em memória Secundária
  - Mergesort Externo;
  - Acesso Indexado;
  - Árvores n-árias;
  - Indexação por espalhamento;

C.H. Prevista
12 horas

Ordenação Externa

Ordenação Externa

- A ordenação externa consiste em ordenar arquivos de tamanho maior que a memória interna disponível.
- Os métodos de ordenação externa são diferentes dos de ordenação interna.
  - Os algoritmos devem diminuir o número de acesso às unidades de memória externa.
- Nas memórias externas, os dados ficam em um arquivo sequencial.
  - Apenas um registro pode ser acessado em um dado momento.

Ordenação Externa

- Fatores que determinam as diferenças das técnicas de ordenação externa em relação à ordenação interna:
  - Maior custo de acesso à memória para transferências dos dados entre a interna e externa
  - Restrições de acesso a dados, como o acesso apenas sequencial ou o custo do acesso
  - Os métodos são dependentes do estado atual da tecnologia

Ordenação Externa

- O método mais importante de ordenação externa é o de ordenação por intercalação
  - Intercalar é combinar dois ou mais blocos ordenados em um único bloco ordenado
  - A intercalação é utilizada como operação auxiliar na ordenação
- O foco dos algoritmos de ordenação externa é reduzir o número de acessos ao arquivo.
  - Uma boa medida de complexidade de um algoritmo de ordenação por intercalação é o número de vezes que um item é lido ou escrito na memória interna.
  - Os bons métodos de ordenação externa geralmente envolvem menos do que dez passadas sobre o arquivo.

Ordenação Externa

- Estratégia geral dos métodos de ordenação externa:
  1. Quebre o arquivo em blocos do tamanho da memória interna disponível.
  2. Ordene cada bloco na memória interna.
  3. Intercale os blocos ordenados, fazendo várias passadas sobre o arquivo.
     - A cada acesso são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

Mergesort Externo

Mergesort Externo

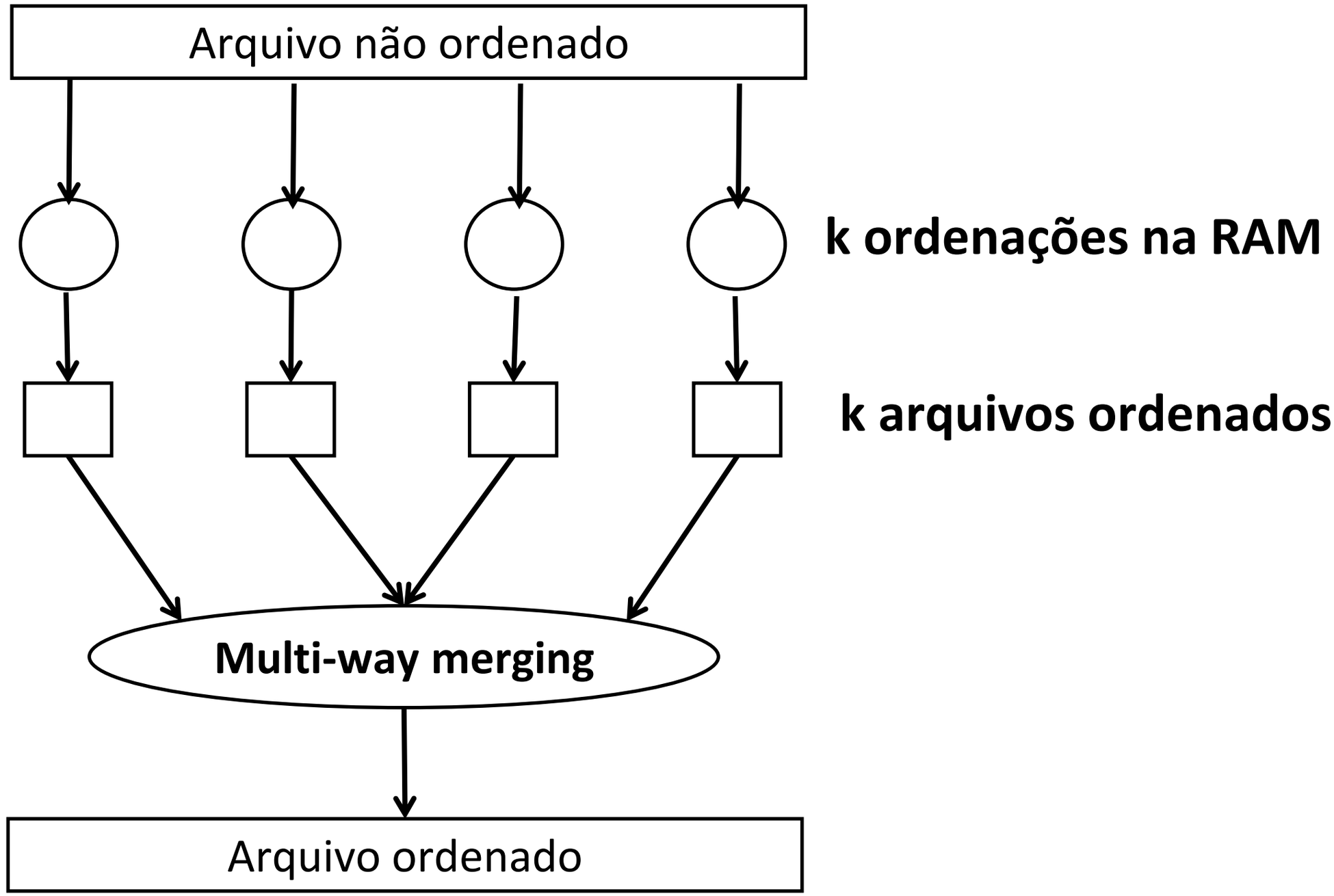
- Método mais comum da ordenação externa.
- Utiliza a técnica de intercalação: combina dois ou mais blocos ordenados em um único bloco, maior, ordenado.
- Funcionamento:
  - RAM comporta **“n”** registros de dados
  - Carregar parte do arquivo na RAM
  - Ordenar os dados na RAM com um algoritmo
  - In-place: ex. **Quicksort**
  - Salvar os dados ordenados em um arquivo separado

Mergesort Externo

- Funcionamento (continuação):
  - Repetir os anteriores até terminar o arquivo original
  - Ao final, temos **“k”** arquivos ordenados
  - “Multi-way merging”
    - Criar **“k+1”** buffers de tamanho **“n/(k+1)”** (**“1”** de saída, **“k”** de entrada)
    - Carregar parte dos arquivos ordenados nos “buffers de entrada”, intercalar no “buffer de saída”
    - Buffers de entrada “vazio”: carregar mais dados
    - Buffers de saída “cheio”: salvar dados

Tamanho máximo de dados na memória

Mergesort Externo



```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 100
//quantidade de valores que é possível ter na memória

void criArquivoTeste(char *nome)
{
    int i;
    FILE *f = fopen(nome, "w");
    srand(time(NULL));
    for (i=1; i < 1000; i++)
    {
        fprintf(f, "%d\n", rand());
    }
    fprintf(f, "%d", rand());
    fclose(f);
}

int main()
{
    criArquivoTeste("dados.txt");
    mergeSortExterno("dados.txt");
    return 0;
}
```

```
void mergeSortExterno(char *nome)
{
     char novo[20];
     int numArqs = criaArquivosOrdenados(nome);
     int i, k = N / (numArqs + 1);

     remove(nome);
     merge(nome, numArqs, K);

     for (i=0; i<numArqs; i++) {
          sprintf(novo, "Temp%d.txt", i+1);
          remove(novo);
     }
}
```

```
int criaArquivosOrdenados(char *nome)
{
    int V[N], cont = 0, total = 0;
    char novo[20];
    FILE *f = fopen(nome, "r");
    while (!feof(f)) {
        fscanf(f, "%d", &V[total]);
        total++;
        if (total == N)
        {
            cont++;
            sprintf(novo, "Temp%d.txt", cont);
            qsort(V, total, sizeof(int), compara);
            salvaArquivo(novo, V, total, 0);
            total = 0;
        }
    }
    if (total > 0) {
        cont++;
        sprintf(novo, "Temp%d.txt", cont);
        qsort(V, total, sizeof(int), compara);
        salvaArquivo(novo, V, total, 0);
    }
    fclose(f);
    return cont;
}
```

Buffer cheio: Salva em disco

Sobraram dados no B uffer: Salva em disco

Nº arquivos gerados

```c
void salvaArquivo(char *nome, int *V, int tam, int mudaLinhaFinal)
{
    int i;
    FILE *f = fopen(nome, "a");
    for (i=0; i < tam-1; i++)
        fprintf(f, "%d\n", V[i]);

    if (mudaLinhaFinal == 0)
        fprintf(f, "%d", V[tam-1]);
    else
        fprintf(f, "%d\n", V[tam-1]);

    fclose(f);
}
```

Controla a mudança d e linha no final do arq uivo

```
struct arquivo {
    FILE *f;
    int pos, MAX, *buffer;
};
```

struct para gerenciar os buffers

```
void merge(char *nome, int numArqs, int K) {
    char novo[20];
    int i;
    int *buffer = (int*)malloc(K*sizeof(int));

    struct arquivo* arq;
    arq = (struct arquivo*)malloc(numArqs*sizeof(struct arquivo));

    for (i=0; i<numArqs; i++) {
        sprintf(novo, "Temp%d.txt", i+1);
        arq[i].f = fopen(novo, "r");
        arq[i].MAX = 0;
        arq[i].pos = 0;
        arq[i].buffer = (int*)malloc(K*sizeof(int));
        preencheBuffer(&arq[i], K);
    }
}
```

Quantidade de elementos que pode carregar na memória para cada um dos buffers

Existe menor elemento?
Coloca no buffer de saída.
Salvar se buffer cheio.

Se verdadeiro, então buffer de saída está cheio.

Sobraram dados no buffer?
Salvar em arquivo.

```
void merge(char *nome, int numArqs, int K) {
    //continuação
    //enquanto houver arquivos para processar
    int menor, qtdBuffer = 0;
    while (procuraMenor(arq, numArqs, K, &menor) == 1) {
        buffer[qtdBuffer] = menor;
        qtdBuffer++;
        if (qtdBuffer == K) {
            salvaArquivo(nome, buffer, K, 1);
            qtdBuffer = 0;
        }
    }
    //salva dados ainda no buffer
    if (qtdBuffer != 0)
        salvaArquivo(nome, buffer, qtdBuffer, 1);

    for (i=0; i<numArqs; i++)
        free(arq[i].buffer);

    free(arq);
    free(buffer);
}
```

```
int procuraMenor(struct arquivo* arq, int numArqs, int K, int* menor) {
    int i, idx = -1;
    for (i=0; i<numArqs; i++) {
        if (arq[i].pos < arq[i].MAX) {
            if (idx == -1)
                idx = i;
            else {
                if (arq[i].buffer[arq[i].pos] < arq[idx].buffer[arq[idx].pos])
                    idx = i;
            }
        }
    }

    if (idx != -1) {
        *menor = arq[idx].buffer[arq[idx].pos];
        arq[idx].pos++;
        if (arq[idx].pos == arq[idx].MAX)
            preencheBuffer(&arq[idx], K);
        return 1;
    }
    else
        return 0;
}
```

Procura menor valor na primeira posição de cada buffer

Achou menor. Atualiza posição do buffer.
Encher se estiver vazio.

Tem dados no arquivo?
Lê e coloca no buffer

```
void preencheBuffer(struct arquivo* arq, int K) {
    int i;
    if (arq->f == NULL)
        return;

    arq->pos = 0;
    arq->MAX = 0;
    for (i=0; i<K; i++) {
        if (!feof(arq->f)) {
            fscanf(arq->f, "%d", &arq->buffer[arq->MAX]);
            arq->MAX++;
        }
        else
        {
            fclose(arq->f);
            arq->f = NULL;
            break;
        }
    }
}
```

Acabou os dados.
Fecha arquivo.

Árvores n-árias

Árvores n-árias

- As árvores binárias de pesquisa são estruturas muito eficientes quando se deseja trabalhar com tabelas que caibam inteiramente na memória principal do computador.
- Satisfazem condições e requisitos diversificados e conflitantes, tais como acesso direto e sequencial, facilidade de inserção e retirada de registros e boa utilização de memória.

- Uma forma simplista de resolver o problema de recuperar informação em grandes arquivos de dados é:
  - Armazenar os **nós** da árvore no disco,
  - e os **apontadores** à **esquerda** e à **direita** de cada nó se tornam os **endereços** de disco em vez de endereços memória principal.

Árvores n-árias

- Para diminuir o número de acessos a disco, os nós da árvore podem ser agrupados em páginas.
- A forma de organizar os nós da árvore dentro de páginas é muito importante sob o ponto de vista do número esperado de páginas lidas quando se realiza uma pesquisa na árvore.
- Um método de alocação de nós em páginas que leva em consideração a relação de proximidade dos nós dentro da estrutura da árvore foi proposto por Muntz e Uzgalis (1970).
- Neste método o **novo nó** a ser inserido é sempre colocado na mesma página do nó pai. Se o **nó pai** estiver em uma página **cheia**, então uma **nova página é criada** e o **novo nó** é colocado no **início** da nova página.

Árvores n-árias

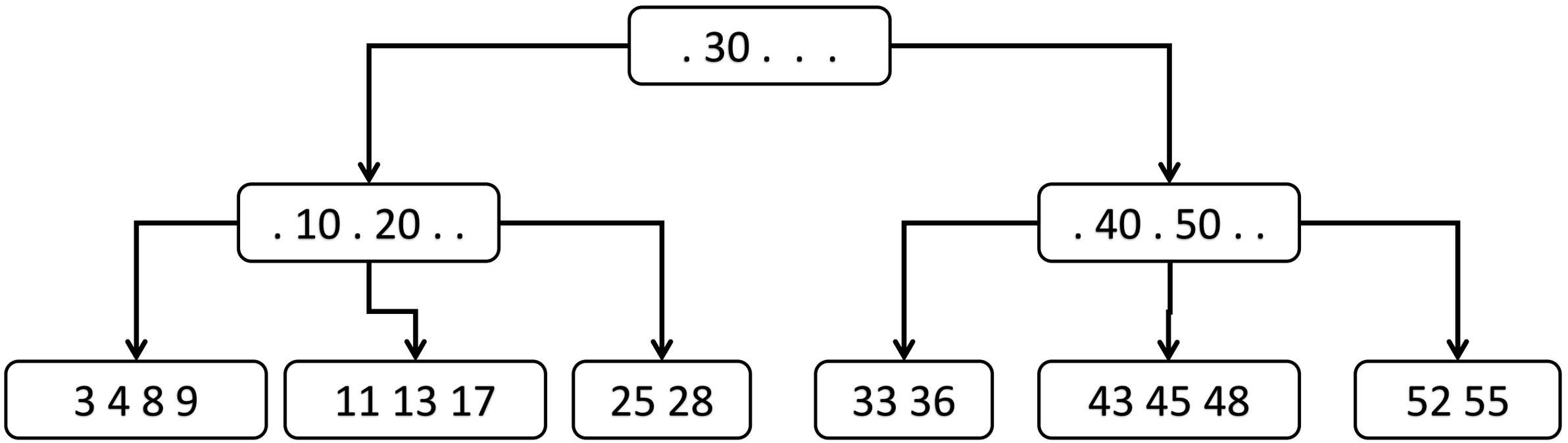
- Knuth (1973) mostrou que o número esperado de acessos a páginas em uma pesquisa na árvore é muito próximo do ótimo.
- Contudo, a ocupação média das paginas é extremamente baixa, da ordem de 10%, o que torna o algoritmo inviável para aplicações práticas.
- Uma solução brilhante para esse problema, simultaneamente a uma proposta para manter equilibrado o crescimento da árvore é permitir inserções e retiradas a vontade.

Árvores B

- Quando uma árvore de pesquisa possui mais de um registro por nó ela deixa de ser **binária**.
- Essas árvores são chamadas de **n-árias**, pelo fato de possuírem mais de dois descendentes por nó.
- Neste caso, os nós são comumente chamados de páginas.
- A árvore B é n-ária. Em uma **árvore B** de ordem ***m***, temos que:
  - Cada página contém no mínimo ***m*** registros e no máximo 2***m*** registros, exceto a página raiz, que pode conter entre 1 e 2***m*** registros. (Cormen, 2001)
  - Ou no mínimo ***m*** + 1 descendentes e no máximo 2***m*** + 1 descendente. (Knuth, 1978)
  - Todas as páginas folha aparecem no mesmo nível.

Árvores B

- Uma árvore B de ordem m = 2 com três níveis pode ser vista abaixo.
- Todas as páginas contêm (entre *m* e 2*m* chaves) dois, três ou quatro registros, exceto a raiz, que pode conter entre 1 e 2*m* chaves.
- Os registros aparecem em ordem crescente da esquerda para a direita.
- O número de filhos (exceto folha) deve ser o número de chaves + 1
- Todas as folhas estão no mesmo nível

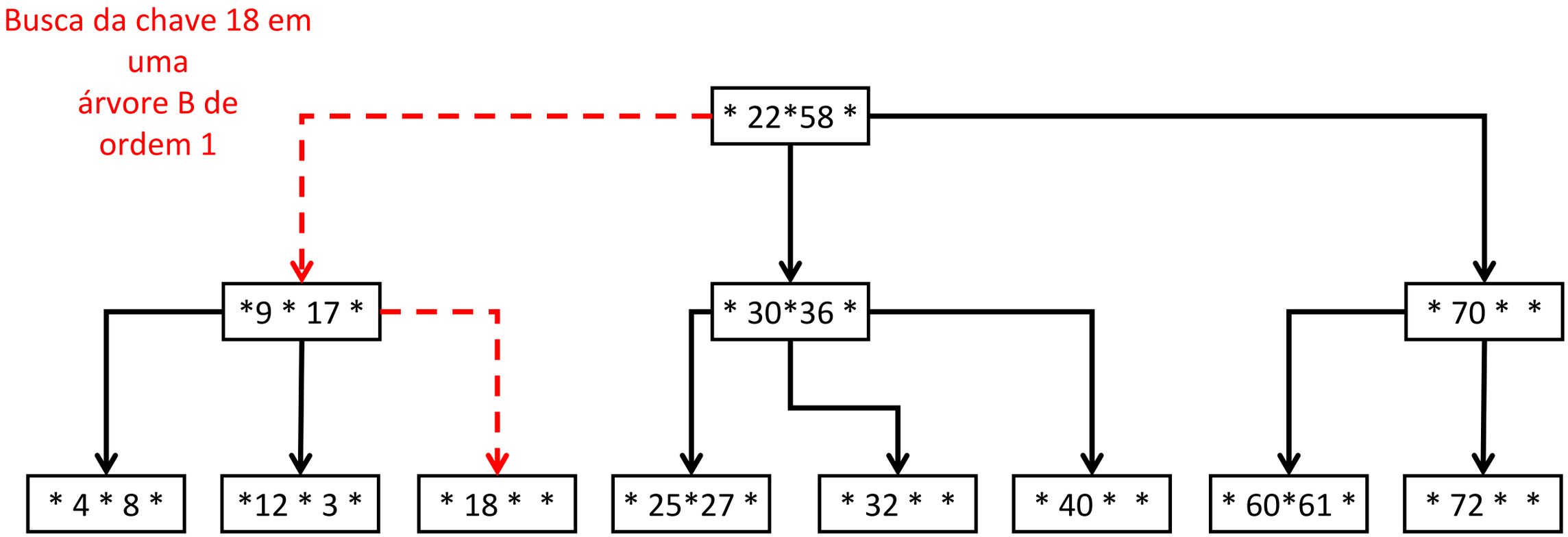


Operações Básicas com Árvore B

- Criação de uma árvore B vazia
- Busca na árvore B
- Inserção na árvore B
- Remoção na árvore B

Busca em Árvore B

- A busca na árvore B é semelhante à busca na árvore binária
- É executado o mesmo caminho
- Contudo, é necessário acrescentar verificações relativas às chaves existentes em cada página, que determinam qual o próximo passo do percurso



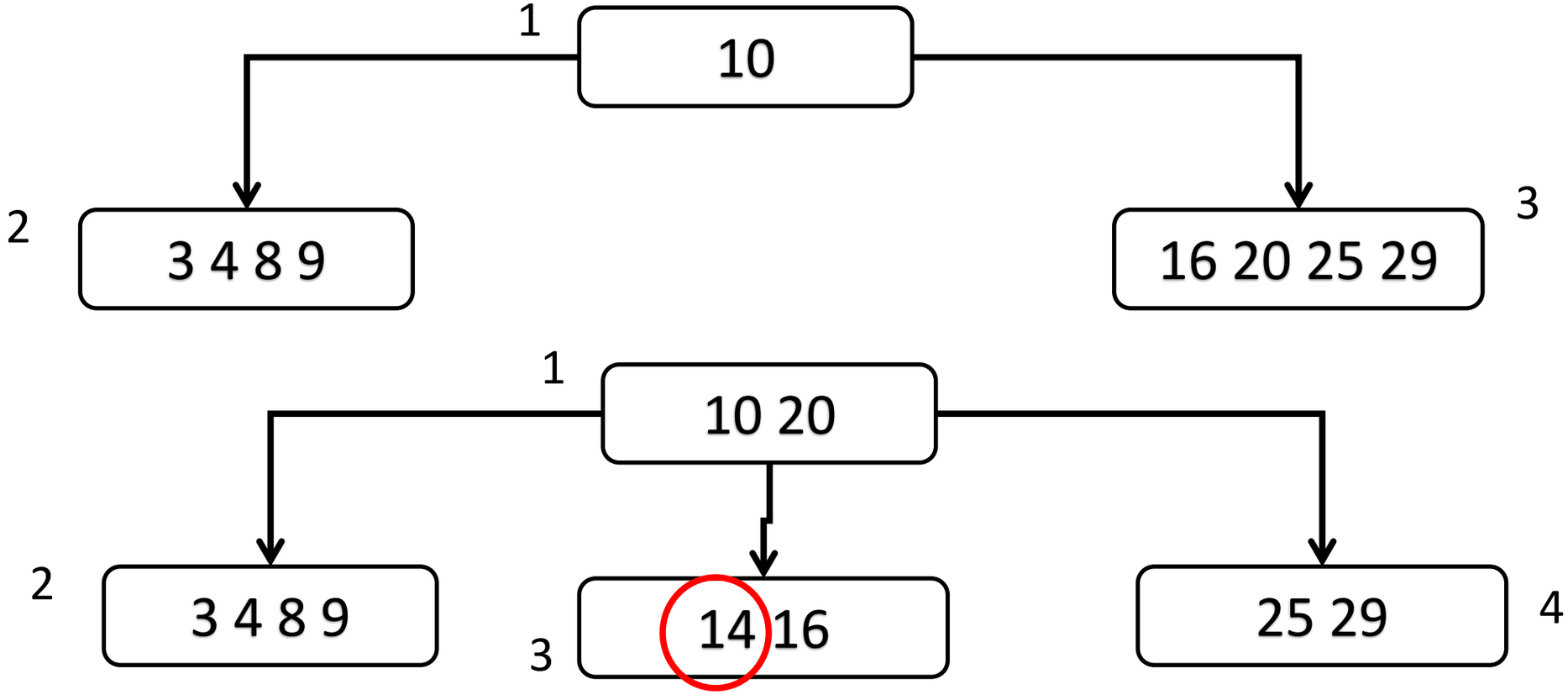
Busca em Árvore B

- O algoritmo de busca compara a chave com as chaves do nó raiz.
- Se não encontrar, então a busca prossegue em um certo filho dessa página, observando a seguinte propriedade:
  - Todas as chaves armazenadas no filho apontado por P_j em uma página são menores que a chave k_j armazenada nessa página. Todas as chaves armazenadas no filho apontado por P_j+1 são maiores que a chave k_j.

Inserir Registro

- O primeiro passo é localizar a página apropriada em que o novo registro deve ser inserido, se o registro já existir o registro não deve ser incluído novamente.
- Se o registro couber na página, basta inclui-lo de forma ordenada.
- Se não couber, a página deve ser dividida em duas e o elemento do meio deve ser promovido.
- Se nesta página não houver espaço para armazenamento da chave promovida, a divisão é repetida até que o problema seja resolvido.

Inserir Registro



- Registro 14 não encontrado e página 3 cheia
- Página 3 dividida em duas páginas, gerando página 4

Exemplo

Ordem m=2

$2 \leq NE \leq 4$

Inserir 11

Inserir 53

Inserir 36

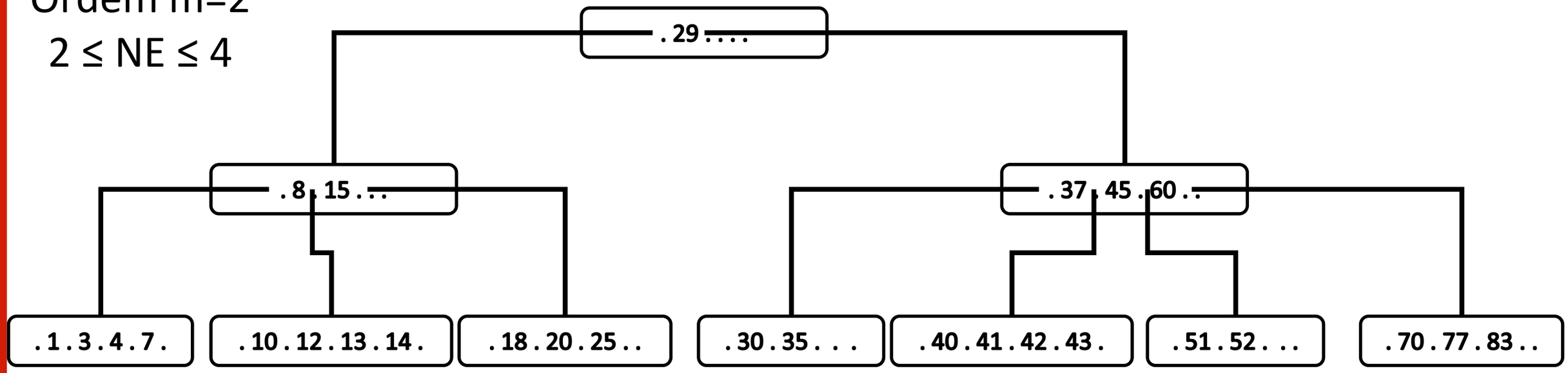
Inserir 95

Inserir 8

Exemplo 2

Ordem m=2

$2 \leq NE \leq 4$



Inserir 11

Inserir 35

Inserir 65

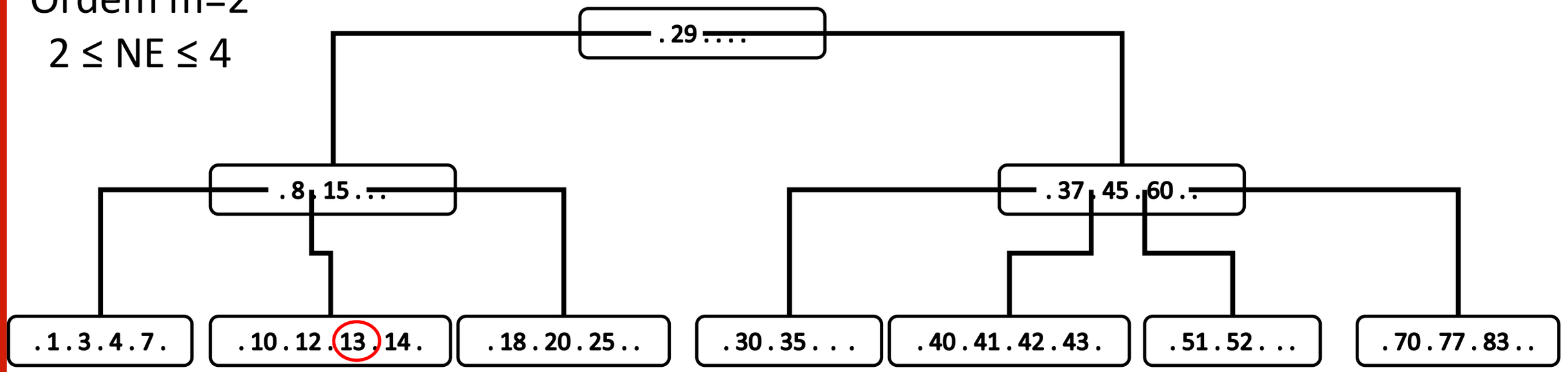
Inserir 88

Remover em Árvore B

Caso 1: o elemento está na folha e a folha manterá 50% de ocupação, basta remover

Ordem m=2

$2 \leq NE \leq 4$



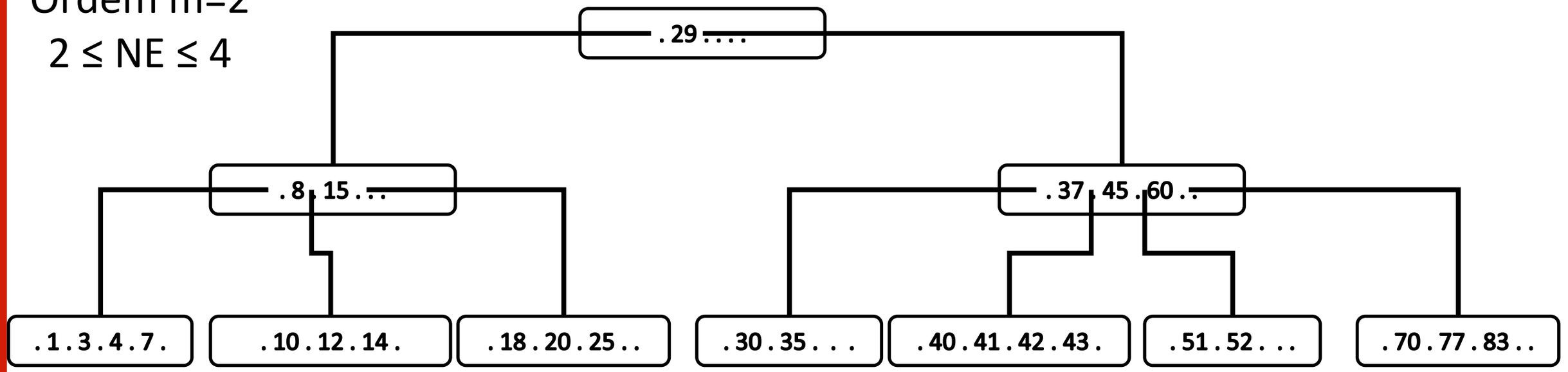
Remover 13

Remover em Árvore B

Caso 1: o elemento está na folha e a folha manterá 50% de ocupação, basta remover

Ordem m=2

$2 \leq NE \leq 4$



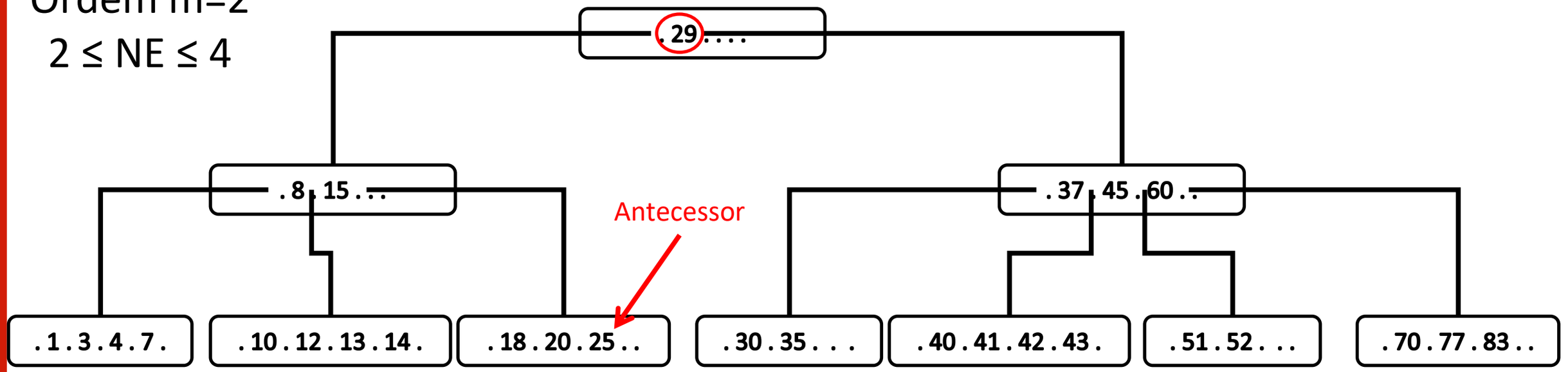
13 removido

Remover em Árvore B

Caso 2: se o elemento não está na folha troque-o pelo seu antecessor (elemento mais a direita da sub-árvore esquerda)

Ordem m=2

$2 \leq NE \leq 4$



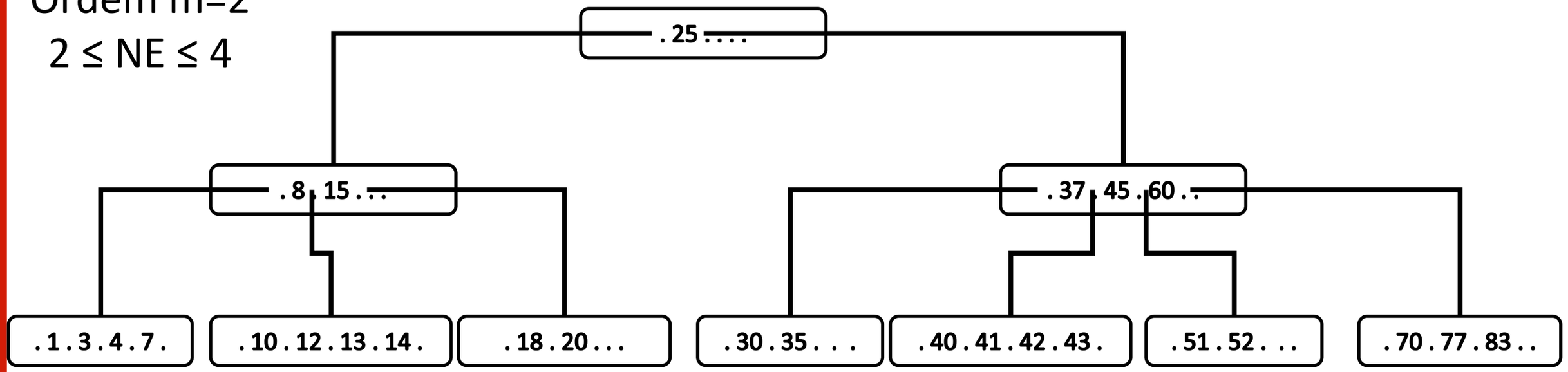
Remover 29

Remover em Árvore B

Caso 2: se o elemento não está na folha troque-o pelo seu antecessor (elemento mais a direita da sub-árvore esquerda)

Ordem m=2

$2 \leq NE \leq 4$



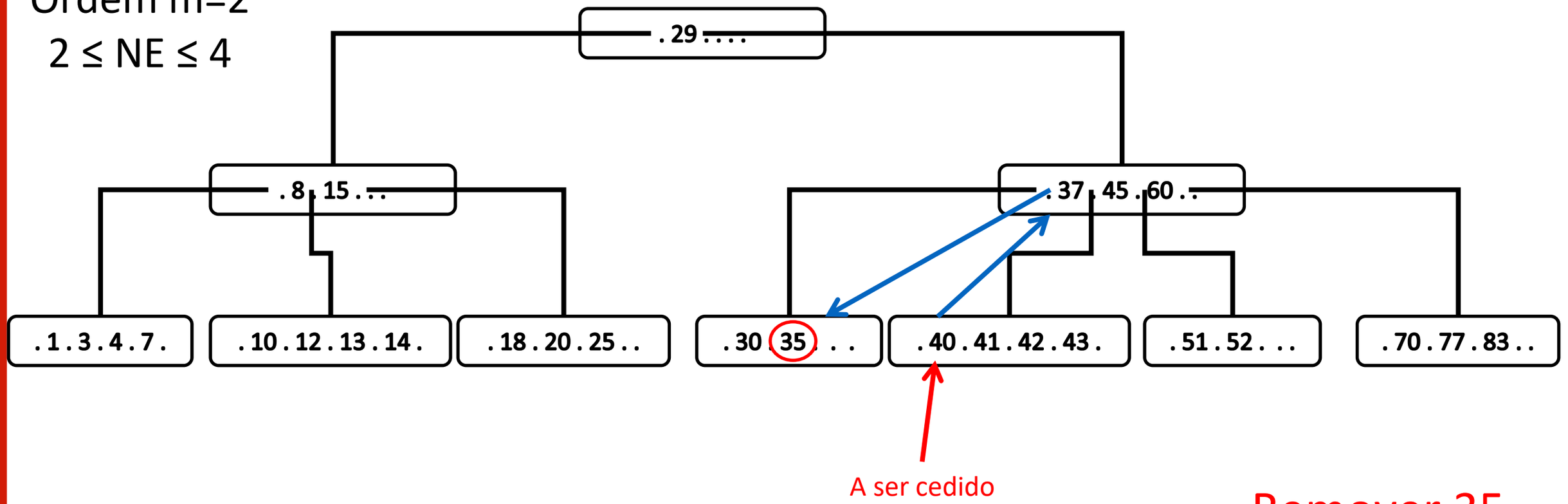
29 removido

Remover em Árvore B

Caso 3: se a folha ficar com menos de 50% de ocupação, mas a página irmã puder ceder uma chave

Ordem m=2

$2 \leq NE \leq 4$



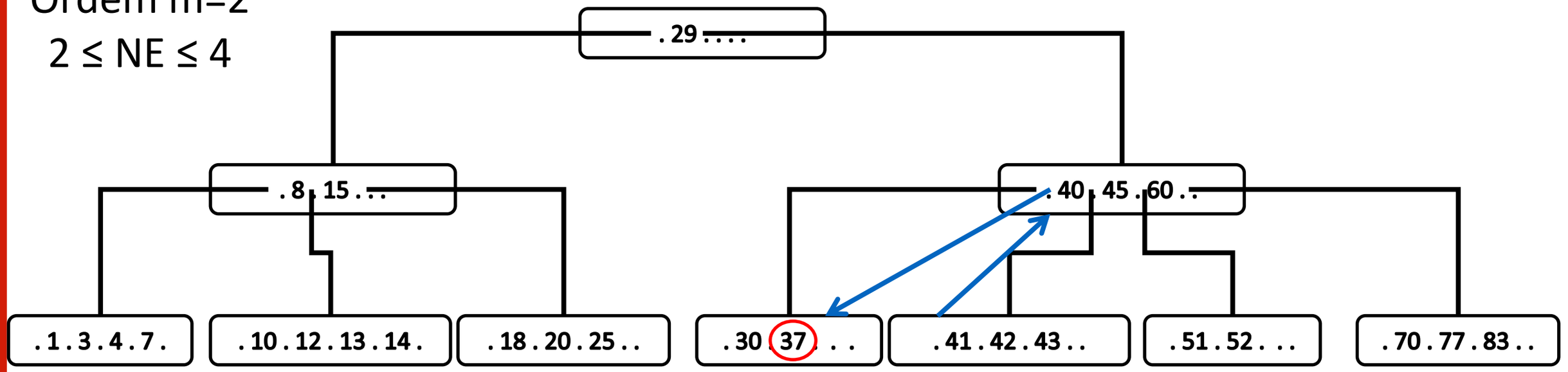
Remover 35

Remover em Árvore B

Caso 3: se a folha ficar com menos de 50% de ocupação, mas a página irmã puder ceder uma chave

Ordem m=2

$2 \leq NE \leq 4$



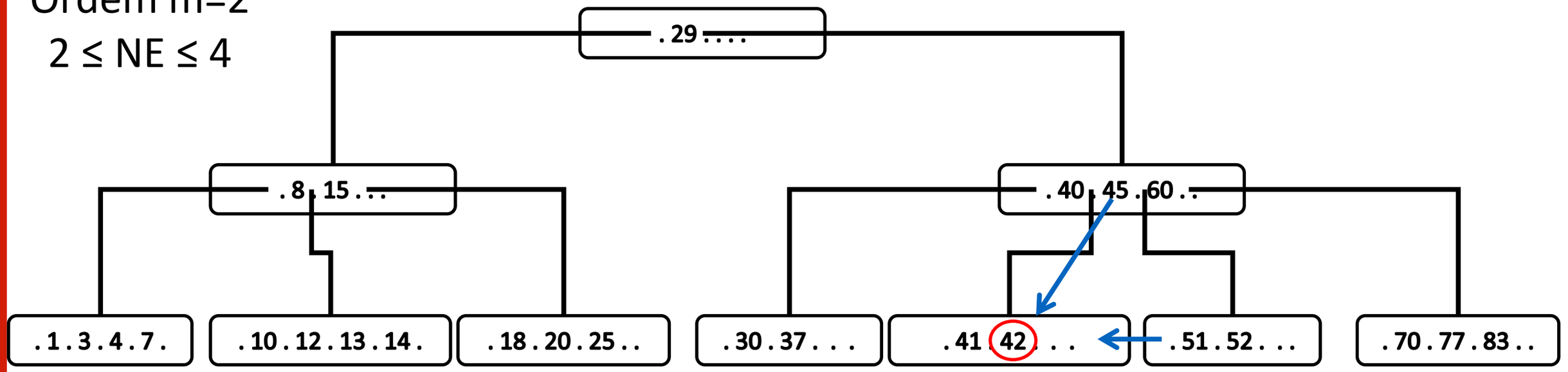
35 removido

Remover em Árvore B

Caso 4: se a folha ficar com menos de 50% de ocupação e as páginas irmãs não puderem ceder uma chave

Ordem m=2

$2 \leq NE \leq 4$



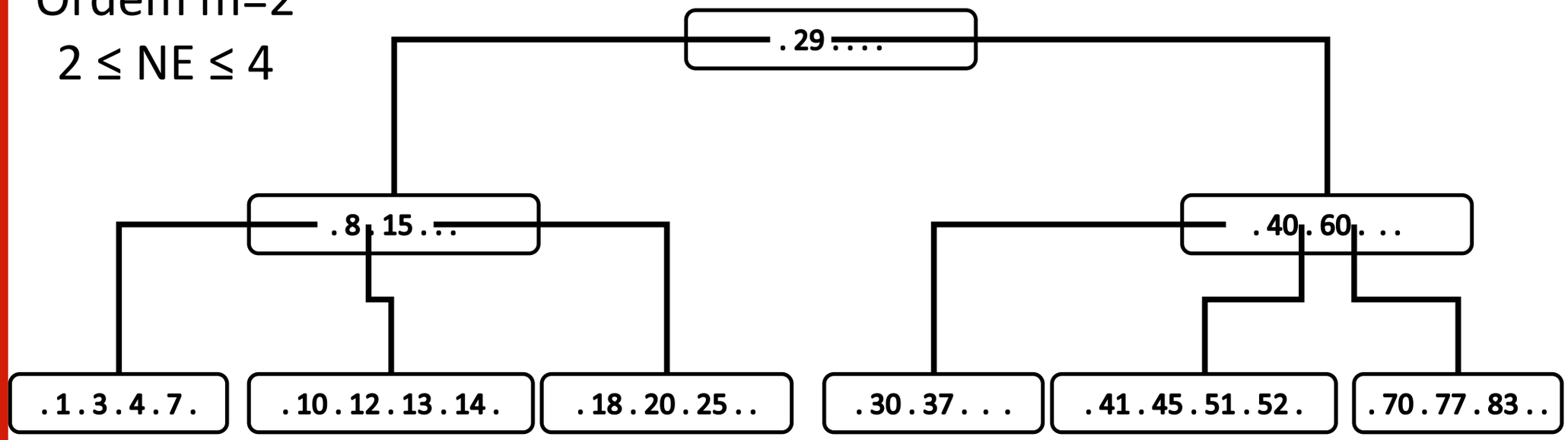
Remover 42

Remover em Árvore B

Caso 4: se a folha ficar com menos de 50% de ocupação e as páginas irmãs não puderem ceder uma chave

Ordem m=2

$2 \leq NE \leq 4$



Remover 42

Estrutura da página

N	P_0	C_0D_0	P_1	C_1D_1	P_2	C_2D_2	P_3	C_3D_3	...	P_{n-1}	$C_{n-1}D_{n-1}$	P_n

Em que:

N – número de elementos presentes na página

C_i – chave do registro (geralmente um código)

D_i – dados (ex.: endereço do registro no arquivo)

P_i – ponteiro para o i-ésimo filho

Fim do conteúdo