Pesquisa Externa

🎓 **Aluno**: Lucas de Araújo
Matrícula: 18.2.4049
Professor Doutor: Guilherme Tavares de Assis

Introdução a Pesquisa Externa e Acesso Sequencial Indexado

Diferente da pesquisa interna onde nós possuíamos uma quantidade de memória principal suficiente para armazenar nossos dados, é estudado no tópico de pesquisa externa algoritmos capazes de realizar buscas de forma eficiente em memória não principal do computador.

Porém existem métricas diferentes que serão abordadas quando comparadas a da pesquisa realizada em memória principal. Dentre estas métricas, a **Medida de Complexidade** é um das mais importantes para realizar a comparação entre métodos de pesquisa externa a fim de entender sua eficiência, sendo até mais importante que puramente apenas o número de comparações.

- A Medida de Complexidade é o número de transferências de dados entre memória principal e secundária

Características que compõe a arquitetura e o sistema operacional são responsáveis também por tornar os métodos dependentes de parâmetros que afetam seu desempenho. Temos como exemplo, a diferença de velocidade com qual os dados são manipulados em um HD (Disco Rígido) e em um SSD (Solid State Drive), onde, o SSD por ser elétrico, consegue manipular volumes de dados me maneira muito mais rápida quando comparado ao HD

Sistema de Paginação

O sistema de paginação, em suma, é uma abordagem (estratégia) que tem como objetivo promover a implementação eficiente de métodos de pesquisa externa e também qualquer outro método que envolva um grande volume de dados em memória secundária através da criação de *"páginas"* durante o processo de pesquisa, com isto, reduzindo o número de transferências entre dados da memória principal e da memória secundária (Medida de Complexidade)

Vamos imaginar o seguinte exemplo:

✏️ 1) Um arquivo possui uma quantidade de 20 registros indexados dentro de si

2) Caso eu precisa acessar o arquivo de registro com index de número 15, será necessário realizar sequencialmente a busca um a um de cada registro, desta forma trazendo um registro por vez a memória principal e em seguida desalocando o mesmo para dar espaço ao registro seguinte

3) Neste cenário, nosso pior caso se dá por $O(n)$, onde n é o número de registros indexados dentro do arquivo

4) Com o intuito de promover eficiência, o sistema de paginação resolve isto através da criação das mencionadas **páginas**, que nada mais são que *"blocos"* de dados divididos de maneira homogênea e que caibam na memória principal

5) Imagine nesse mesmo exemplo mencionado no item 1, se a memória principal suportar blocos de de até 5 registros, podemos gerar 4 blocos (cada um com 5 registros) e desta forma o número de transferências de dados é reduzido e não é mais dependente de n

Dito isto, podemos dividir as páginas em dois tipos:

- Páginas Ativas: Páginas que foram estão presentes na memória principal
- Páginas Inativas: Páginas restantes que se encontram residentes na memória secundária

Mecanismo do Sistema de Paginação

O mecanismo do sistema de paginação possui duas funções:

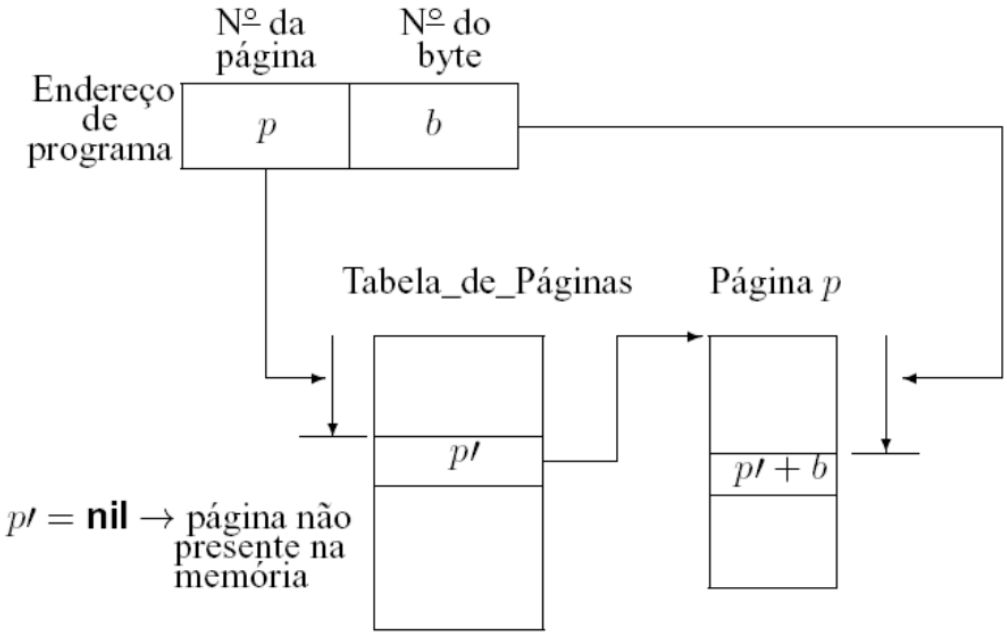
- **Mapeamento de Endereços**: Determinar qual página da memória secundária está sendo endereçada por um programa e desta forma, encontrar a moldura de determinada página na memória principal, caso venha a existir. Com isto, caso necessário precisar-se de um item da memória secundária que já esteja presente na memória principal, não será necessário realizar uma nova transferência (redução da medida de complexidade)

- **Transferência de Páginas**: Realizar a transferência de páginas da memória secundária para memoŕia primária, quando necessário, e também transferi-las de volta para a memória secundária caso não estejam mais sendo utilizadas na memória principal

Mapeamento de Endereços

Quando se é endereçado uma página, é utilizado uma parcela dos bits do endereço para representar o número da página e a outra parte o número do *byte* do item presente dentro dá página. Este mapeamento de endereços que envolve a memória principal e memória secundária é realizado por meio de uma **Tabela de Páginas**

Uma **Tabela de Páginas** pode ser imaginada como um vetor que armazena em cada posição, um determinado endereço para uma referente página na memória principal.
Com isto, em um cenário onde é necessário utilizar-se novamente uma determinada página em uma determinada parte do programa, será feito primeiramente a consulta nesta tabela de páginas para conferir se página já não se encontra na memória principal.
Caso não se encontre, ela é transferida da memória secundária e a tabela será atualizada com um novo endereço referente a está página



Exemplo ilustrativo do funcionamento do mapeamento de endereços (retirado do slide)

- A p-ésima entrada da tabela contém a localização p' da moldura da página que contém a página número p, caso a mesma esteja na memória principal
- Caso a página número p não esteja em uma moldura na memória principal, p' não terá valor algum, ou seja, p' = NULL

> "Mas e se caso a memória principal estiver cheia e for necessário trazer algo da memória secundária? O que deverá ser feito?"

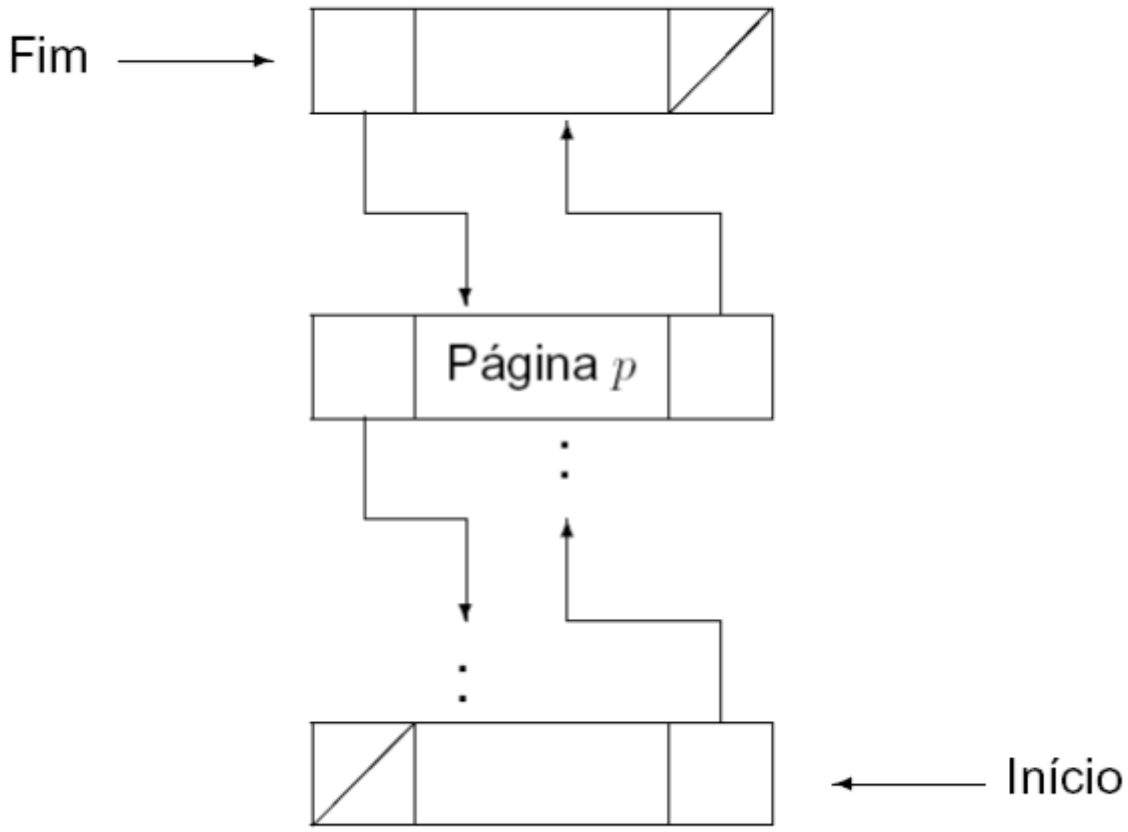
Neste cenário, uma página deverá ser removida da memória principal a fim de liberar espaço para uma nova pagina. Mas qual página deverá ser removida?

Políticas de Remoção de Páginas da Memória Principal

Menos Recentemente Utilizada (Least Recently Used)

Essa política consiste em remover a página menos recentemente utilizada através do principio de analise do seu comportamento futuro com base no seu comportamento passado.

1. Toda vez que uma página é utilizada, ela é colocada no fim da fila
2. A página que está no inicio da fila é a página LRU
3. A nova página trazida da memória secundária deve ser colocada na moldura que contém a página LRU



Exemplo ilustrativo do funcionamento da política LRU (retirado do slide)

Menos Frequentemente Utilizada (Least Frequently Used)

Tem como ideia remover a pagina com menor frequência de utilização através de um contator embutido nas páginas. Dessa forma, com cada acesso feito na página, o contator é acrescido de uma unidade (simbolizando assim a frequência de uso)

- **Desvantagem**: Uma página recentemente trazida da memória secundária tem um baixo número de acessos e pode ser brevemente removida
- A fim de sanar essa desvantagem, é utilizada uma média para iniciar o contador de uma página, evitando assim sua saída precocemente

Ordem de Chegada (First In First Out)

Tem como ideia remover a página que se encontra na memória principal há mais tempo

- É o algoritmo mais simples e barato de se manter
- **Desvantagem**: Ignora a importância que envolve as páginas, levando em consideração apenas a ordem de chegada

Métodos de Pesquisa Externa

Acesso Sequencial Indexado

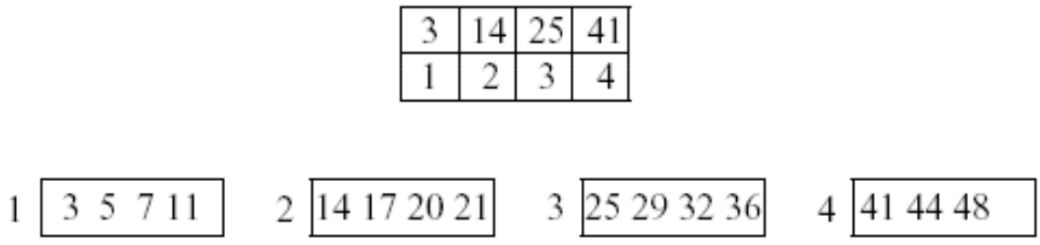
Este método de pesquisa externa utiliza o princípio da pesquisa sequencial, ou seja, cada item será lido sequencialmente até que se encontre uma chave maior ou igual a chave de pesquisa

Este método possui alguns pré-requisitos necessários para ter maior eficiência, sendo eles:

- O arquivo deve estar ordenado pelo campo chave do item
- Um arquivo de índice de páginas, contendo os pares de valores $< x, p >$ deve ser criado, onde x representa a chave de um item e p representa o endereço da página na qual o primeiro item contém a chave x

Exemplo Ilustrativo

Vamos considerar um arquivo sequencial indexado com 15 itens no total. Cada página tem a capacidade de armazenar quatro itens do disco e cada entrada do índice de páginas armazena a chave do **primeiro** item de cada página e também o endereço de tal página no disco



Exemplo de arquivo sequencial indexado (retirado dos slides)

- Na linha de baixo podemos ver as páginas numeradas de 1 a 4 com até quatro itens armazenados (no caso, números)
- Na linha de cima temos a criação do índice de páginas (vetor) que se localiza na memória principal. Cada posição do vetor armazena duas informações:
  - O número da página
  - A chave de menor valor que se encontra nesta página

Processo de Pesquisa

"Vamos supor que se deseja recuperar o registro de chave 29, como será feita a pesquisa?"

1) Primeiro devemos localizar através do índice de páginas, a página que pode conter o item desejado (nosso caso, o valor 29) de acordo com sua chave de pesquisa

- Nesse caso, devemos comparar com cada valor do índice de páginas e notar que, caso o valor seja maior que o índice anterior e menor que índice sucessor, ele estará na página de índice anterior
- No nosso exemplo, 29 é maior que o o menor valor armazenado na página de índice 3 (que no caso é 25), porém é menor que o menor valor armazenado na página de índice 4 (no caso, 41). Portanto, 29 irá estar em alguma posição da página de índice 3.

2) Trazemos a página de índice 3 para memória principal e realizamos uma pesquisa sequencial

- Iremos realizar este carregamento através do reposicionamento do ponteiro dentro do arquivo
- Este reposicionamento se dará através da função `fseek`, que possibilita a manipulação do ponteiro dentro do arquivo.
- Para encontrar o valor que devemos apontar com o ponteiro, devemos levar em consideração três itens:
  1. Capacidade de Cada página (4 itens nesse caso)
  2. Índice da página que contém nosso valor desejado (no caso, 3, portanto devemos "pular" 2 páginas)

3. Tamanho de cada item dentro de cada página (obtido através da função `sizeof`)

Por fim, teremos uma fórmula de procura parecida com esta:

$$Position = 4 * 2 * sizeof(item)$$

Após localizar a página, utilizamos o `fread` para executar a leitura da página e transportá-la para memória principal

- É possível aumentar a eficiência da pesquisa dentro da página através da utilização de uma pesquisa binária ($O(\log n)$) ao invés de uma pesquisa sequencial ($O(n)$)

Implementação

```
#include <iostream>
#include <stdio.h>
using namespace std;

#define ITENSPAGINA 4
#define MAXTABELA 100

// definição de uma entrada da tabela de índice das páginas
typedef struct {
   int posicao;
   int chave;
} tipoindice;

// definição de um item do arquivo de dados
typedef struct {
   char titulo[31];  int chave;  float preco;
} tipoitem;

                                                // continuando ...
```

- Definição do `tipoitem`
- Definição do `tipoindice`

```
int main () {

   tipoindice tabela[MAXTABELA];
   FILE *arq;   tipoitem x;   int pos, cont;

   // abre o arquivo de dados
   if ((arq = fopen("livros.bin","rb")) == NULL) {
      cout << "Erro na abertura do arquivo\n"; return 0;
   }

   // gera a tabela de índice das páginas
   cont = 0; pos = 0;
   while (fread(&x, sizeof(x), 1, arq) == 1) {
      cont++;
      if (cont%ITENSPAGINA == 1) {
         tabela[pos].chave = x.chave;
         tabela[pos].posicao = pos+1;
         pos++;
      }
   }                                          // continuando ...
```

15

- Abertura do arquivo `livros.bin` (fazendo a devida verificação de existência)
- Geração da tabela de índices das páginas
- Dentro do `while` juntamente ao `if` será feito a separação de páginas dentro da memória através da operação de módulo
- O código não é eficiente pois é a realizada a leitura um a um dos itens dentro do arquivo (linear). Uma forma de solucionar isso, seria mais eficiente posicionar o ponteiro em cada uma das devidas posições necessárias através da utilização da função `fseek`
- Outra forma de solução seria através da leitura de quatro itens por vez e armazená-los na variável `x`.

```
   fflush (stdout);
   cout << "Código do livro desejado:";  cin >> x.chave;

   // ativa a função de pesquisa
   if (pesquisa (tabela, pos, &x, arq))
      printf ("Livro %s (codigo %d) foi localizado",
                                          x.titulo, x.chave);
   else
      printf ("Livro de código %d nao foi localizado",x.chave);

   fclose (arq);
   return 0;

}
```

- Ativação da função de pesquisa

```
int pesquisa (tipoindice tab[], int tam,
              tipoitem* item, FILE *arq) {

  tipoitem pagina[ITENSPAGINA];
  int i, quantitens;
  long desloc;

  // procura pela página onde o item pode se encontrar
  i = 0;
  while (i < tam && tab[i].chave <= item->chave) i++;

  // caso a chave desejada seja menor que a 1a chave, o item
  // não existe no arquivo
  if (i == 0) return 0;
  else {
          (*)                                  // continuando ...
```

- Nessa função será implementada a lógica de pesquisa descrita nesta parte

```
    // a ultima página pode não estar completa
    if (i < tam) quantitens = ITENSPAGINA;
    else {
       fseek (arq, 0, SEEK_END);
       quantitens = (ftell(arq)/sizeof(tipoitem))%ITENSPAGINA;
    }

    // lê a página desejada do arquivo
    desloc = (tab[i-1].posicao-1)*ITENSPAGINA*sizeof(tipoitem);
    fseek (arq, desloc, SEEK_SET);
    fread (&pagina, sizeof(tipoitem), quantitens, arq);

    // pesquisa sequencial na página lida
    for (i=0; i < quantitens; i++)
       if (pagina[i].chave == item->chave) {
          *item = pagina[i];    return 1;
       }
    return 0;
  }
}                                          // continuando ...
```

(*)

- Realiza verificação se a última página está ou não completa (com 4 itens armazenados)
- **Atenção no seguinte cenário**: Caso a ultima página esteja preenchida (quatro itens armazenados), `quantitens` será igual a zero e dessa forma o `fread` não irá executar a função de leitura corretamente
  - Para resolver basta alimentar o `else` com uma condicional a mais: Caso `quantitens` seja igual a zero, `quantitens` irá receber `ITENSPAGINA`

Árvores de Pesquisa

Árvores Binárias de Pesquisa

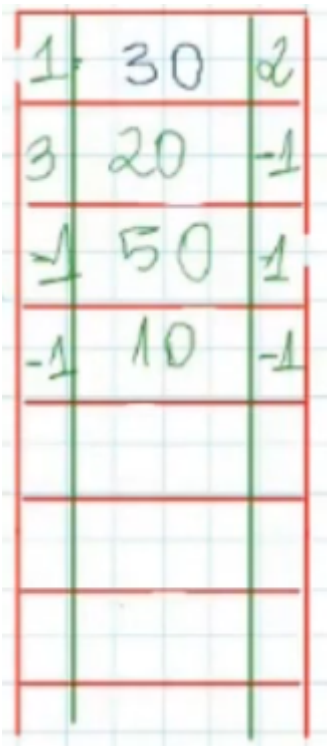
Uma árvore binária de pesquisa é uma estrutura de dados de árvore binária baseada em nós, onde os nós localizados a esquerda possuem valor numérico inferior ao nó raiz e os nós localizados a direita possuem valor numérico superior ao nó raiz. São muito eficientes quando a memória principal armazena todos os itens.
Além disso, proporcionam:

- Acesso direto e sequencial
- Facilidade de inserção e remoção de itens

- Utilização eficiente de memória

Para pesquisar um item dentro de grandes arquivos de dados que estejam armazenados em memória secundária, as árvores binarias de pesquisa podem ser usadas de forma simplista

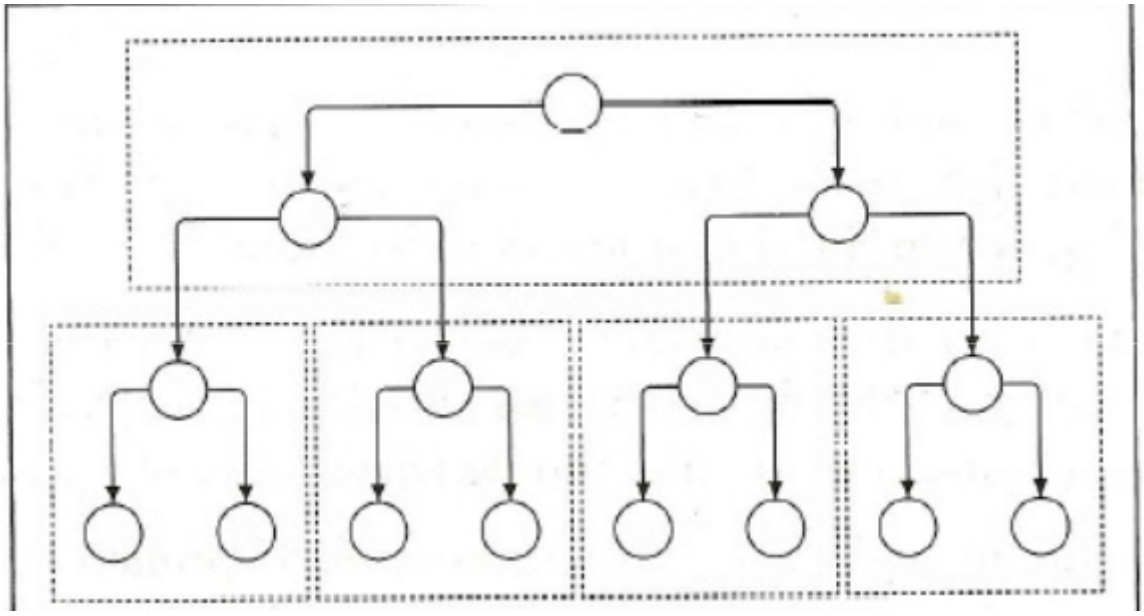
- Os nós das árvores são armazenados em disco e os seus apontadores à esquerda e à direita armazenam endereços de disco ao invés de endereços de memória principal



- São necessários cerca de $\log_2 n$ acessos ao disco para se encontrar um item
  - **Exemplo**:

$$n = 1024$$
$$log_2 1024 = 10$$

A fim de diminuir o número de acessos a disco, os nós de uma árvore podem ser agrupados em páginas. Dessa forma geramos uma redução no número de acessos a disco e dessa forma reduzimos o tempo de execução. Como também como visto anteriormente, acesso ao disco é muito custoso e por isso deve ser evitado de forma repetitiva



Exemplo de árvore agrupada em páginas (retirado do slide)

No caso ótimo, em termos de acessos a disco, o formato da árvore irá mudar de binário ($n = 2$) para quartenário ($n = 4$), o que reduz pela metade o número de acesos ao disco no pior caso.

- **Exemplo**:

$$n = 1024$$
$$log_4 1024 = 5$$

A organização dos nós dentro das páginas tem uma relação muito importante com o número esperado de páginas lidas quando se realiza uma pesquisa. Uma organização "ótima" dificilmente será obtida durante a construção da árvore pois é um problema complexo de otimização.

O algoritmo simples de "*alocação sequencial*" armazena de forma consecutiva os nós na página a medida que vão surgindo e não considerando o formato físico da árvore. Com isso podemos listar suas vantagens e desvantagens:

- **Vantagem**: Utiliza todo o espaço disponível na página
- **Desvantagem**: Os nós de uma pagina estarão relacionados pela ordem de entrada dos itens e não pela localidade na árvore e com isso piorando bastante o tempo de pesquisa

Em 1970, foi proposto um método de alocação de nós em paginas por Muntz e Uzgalis levando em consideração a proximidade dos nós dentro dá arvore. Sendo assim, após a realização da leitura de um item, serão seguidas as regras:

1. O novo nó é sempre colocado na mesma página do nó pai
2. Caso a página do nó pai esteja cheia, o novo nó será colocado no início de uma nova página criada

Com isso, temos algumas vantagens e desvantagens também:

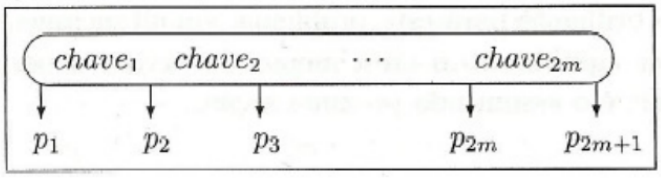
- **Vantagem**: O número de acessos na pesquisa é próximo do ótimo
- Desvantagem: A ocupação média das páginas é baixa (muitas páginas estarão incompletas), com ordem de 10%

Árvore B

A árvore B foi proposta em 1972 por Bayers e McCreight como uma solução para o problema relacionada a ocupação das páginas, dessa forma permitindo o crescimento da árvore de forma equilibrada e permitindo inserções e retiradas

Como suas características, podemos citar:

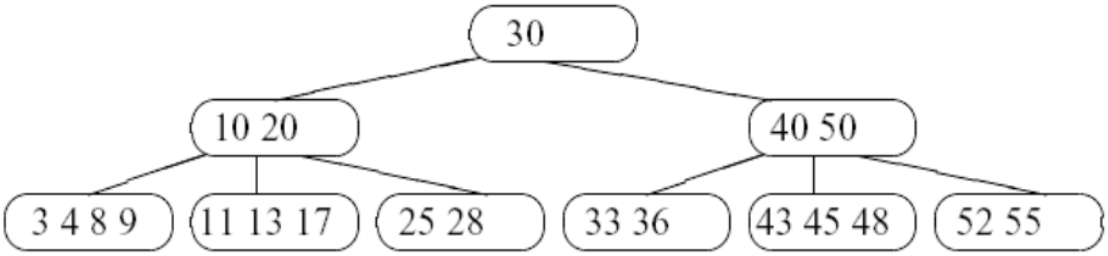
- É uma árvore n-ária, ou seja, permite mais de dois descendentes por nó (comumente chamados de páginas por conta da sua maior capacidade de armazenamento)
- Uma árvore B de ordem possui:
  - Página Raiz: Contém entre 1 e $2m$ itens
  - Demais Páginas: Contém, no mínimo, m itens e $m + 1$ descendentes, com no máximo $2m$ itens e $2m + 1$ descendentes
  - Páginas Folhas: Aparecem todas no mesmo nível
  - Itens: Aparecem dentro de uma página em ordem crescente, de acordo com suas chaves, da esquerda para direita



A forma geral de uma página de uma árvore B de ordem m (exemplo retirado do slide)

Exemplo Ilustrativo

Suponhamos uma árvore B de ordem $m = 2$ com três níveis. Dessa forma, todas as páginas irão conter de 2 a 4 itens, exceto a raiz que pode conter de 1 a 4 itens



O esquema representa uma extensão natural da árvore binária de pesquisa (exemplo retirado do slide)

- Complexidade Média: $\mathcal{O}(\log n)$
- Cada página guarda dentro de si:
  - Um vetor capaz de armazenar $2m$ itens (Tipo Registro)
  - Um vetor capaz de armazenar $2m + 1$ apontadores (Tipo Apontador)
  - Campo tamanho que guarda o número total de itens presentes no vetor do Tipo Registro

Código - Tipos

```
typedef long TipoChave;

typedef struct TipoRegistro {
    TipoChave Chave;
    /* outros componentes */
} TipoRegistro;

typedef struct TipoPagina* TipoApontador;

typedef struct TipoPagina {
    short n;
    TipoRegistro r[MM];
    TipoApontador p[MM + 1];
} TipoPagina;
```

```
void Inicializa (TipoApontador Arvore)
{
   Arvore = NULL;
}
```

Pesquisa na Árvore B

A operação de pesquisa realizada em uma árvore B é semelhante a operação de pesquisa em uma árvore binária de pesquisa (mencionada anteriormente aqui). Para

encontrar o item de valor x, deve-se executar as seguintes operações:

1. Comparar a chave do item x com as chaves que estão presentes na página raiz até encontrar a chave desejada ou o intervalo no qual ela se encaixa
2. Caso não se tenha localizado a chave desejada, devemos seguir o apontador para a subárvore do intervalo encontrado
3. Repetir o processo de maneira recursiva até encontrar a chave ou atingir uma página folha (apontador de valor nulo)

- **Observação**: O intervalo desejado pode ser encontrado por meio de uma pesquisa sequencial ou algum outro método que seja mais eficiente

Código - Pesquisa

```
void Pesquisa(TipoRegistro *x, TipoApontador Ap)
{ long i = 1;
  if (Ap == NULL)
  { printf("TipoRegistro nao esta presente na arvore\n");
    return;
  }
  while (i < Ap->n && x->Chave > Ap->r[i-1].Chave) i++;
  if (x->Chave == Ap->r[i-1].Chave)
  { *x = Ap->r[i-1];
    return;
  }
  if (x->Chave < Ap->r[i-1].Chave)
  Pesquisa(x, Ap->p[i-1]);
  else Pesquisa(x, Ap->p[i]);
}
```

Pesquisa sequencial para se encontrar o intervalo desejado

Verifica se a chave desejada foi localizada

Ativação recursiva da Pesquisa em uma das subárvores (esquerda ou direita)

```
void Imprime(TipoApontador arvore){
    int i = 0;

    if (arvore == NULL) return;

    while (i <= arvore->n) {
        Imprime(arvore->p[i]);
        if (i != arvore->n)
            cout << arvore->r[i].Chave << " ";
        i++;
    }

}
```

Podemos realizar algumas melhorias neste método de pesquisa:

- Ao invés do método `Pesquisa()` "retornar" um `void`, podemos pedir para que o mesmo retorne um `int` indicando se determinado valor foi encontrado (1) ou determinado valor não foi encontrado (0) (seria interessante retornar um `bool`, porém o C não possui esse tipo)

Inserção na Árvore B

A fim de **manter a complexidade logarítmica**, a **árvore B precisa sempre estar balanceada**. Dessa forma, não irá existir um nó filho isolado em um nível separado dos demais e sim sempre o mesmo número de nós por nível da árvore.

"Suponhamos que eu deseje inserir o nó com valor x na respectiva árvore B demonstrada anteriormente aqui, como faria isto?"

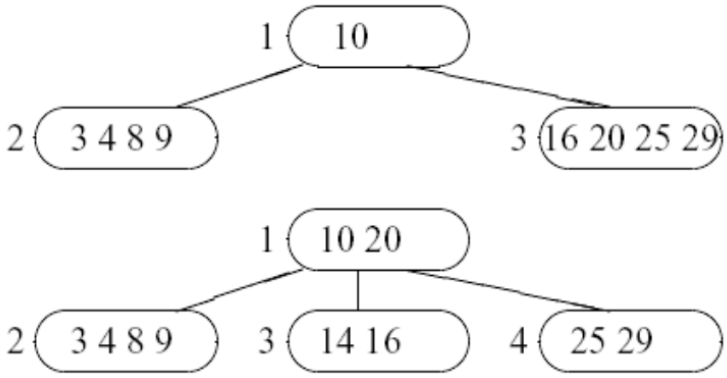
1. Primeiramente realizamos uma pesquisa para localizar qual a página o item deve ser inserido
2. Se o item a ser inserido encontra uma página que possua menos de $2m$ itens, o processo de inserção fica limitado para está página

3. Se o item a ser inserido se deparar com uma página cheia, será criada uma nova página para divisão dos itens envolvendo a nova página, a página onde o item seria inserido e a página pai de ambas

- Se a página pai estiver cheia, o processo de divisão se propaga
- No pior caso, o processo de divisão pode propagar-se até a raiz da árvore, e neste caso, sua altura irá aumentar (a única forma de aumentar a altura de uma árvore B é pela divisão de raiz)

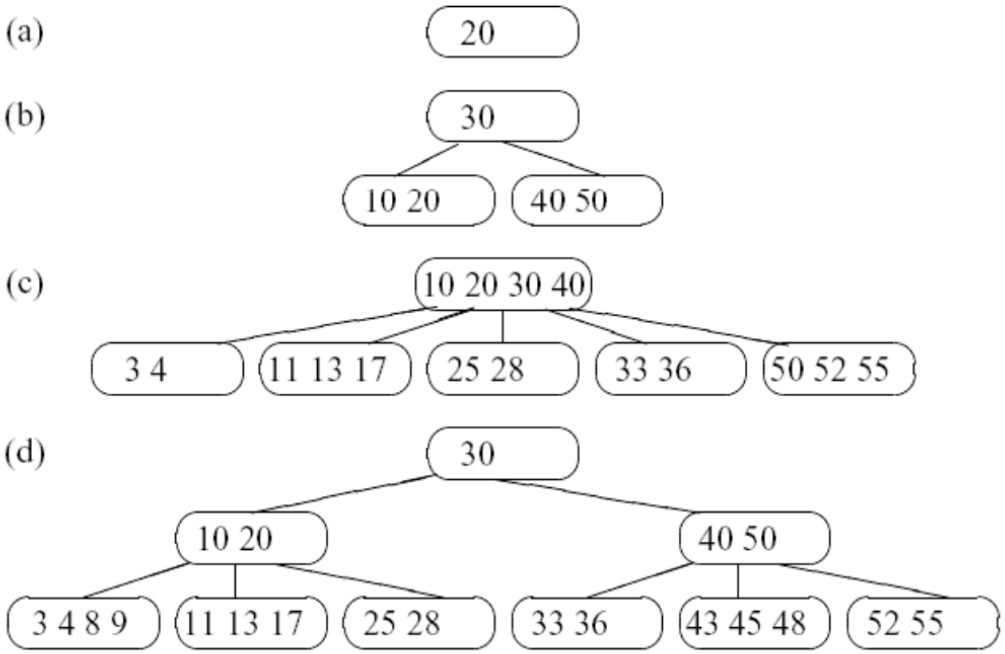
Inserção na Árvore B - Exemplo Ilustrativo

- Ex.: inserção do item com chave 14 em uma árvore B.
  - Item contendo a chave 14 não se encontra na árvore.
  - Página 3, onde o item deveria ser inserido, está cheia.
  - Página 4 é criada, por meio da divisão da página 3.
    - Os **2m+1** itens são distribuídos igualmente entre as páginas 3 e 4, e o item do meio (chave 20) é movido para a página pai.



Exemplo de inserção do item com chave 14 na árvore B (retirado do slide)

- Ex.: inserção das chaves (a) 20; (b) 10, 40, 50, 30; (c) 55, 3, 11, 4, 28, 36, 33, 52, 17, 25, 13; (d) 45, 9, 43, 8, 48.



Exemplo de inserção de múltiplos itens na árvore B (retirado do slide)

Inserção na Árvore B - Código

```
void InsereNaPagina(TipoApontador Ap,
                    TipoRegistro Reg, TipoApontador ApDir)
{ short NaoAchouPosicao;
  int k;
  k = Ap->n;  NaoAchouPosicao = (k > 0);
  while (NaoAchouPosicao)
    { if (Reg.Chave >= Ap->r[k-1].Chave)
      { NaoAchouPosicao = FALSE;
        break;
      }
      Ap->r[k] = Ap->r[k-1];
      Ap->p[k+1] = Ap->p[k];
      k--;
      if (k < 1) NaoAchouPosicao = FALSE;
    }
  Ap->r[k] = Reg;
  Ap->p[k+1] = ApDir;
  Ap->n++;
}
```

```
void Ins(TipoRegistro Reg, TipoApontador Ap, short *Cresceu,
         TipoRegistro *RegRetorno, TipoApontador *ApRetorno)
{ long i = 1; long j;
  TipoApontador ApTemp;
  if (Ap == NULL)
  { *Cresceu = TRUE; (*RegRetorno) = Reg; (*ApRetorno) = NULL;
    return;
  }
  while (i < Ap->n && Reg.Chave > Ap->r[i-1].Chave)  i++;
  if (Reg.Chave == Ap->r[i-1].Chave)
  { printf(" Erro: Registro ja esta presente\n"); *Cresceu = FALSE;
    return;
  }
```

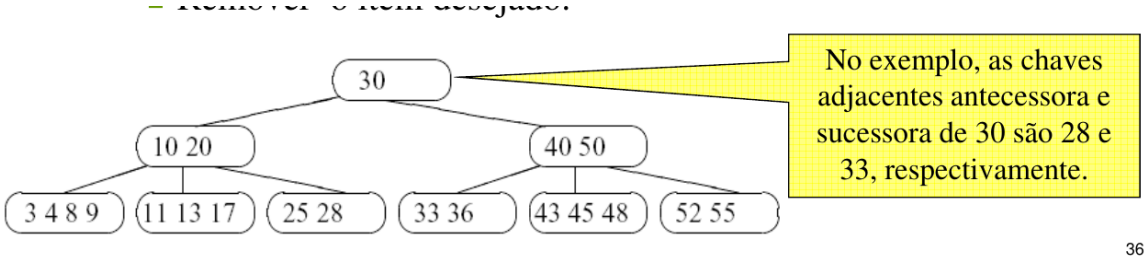
```
  if (Reg.Chave < Ap->r[i-1].Chave) i--;
  Ins(Reg, Ap->p[i], Cresceu, RegRetorno, ApRetorno);
  if (!*Cresceu) return;
  if (Ap->n < MM)    /* Pagina tem espaco */
    { InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
      *Cresceu = FALSE;
      return;
    }
  /* Overflow: Pagina tem que ser dividida */
  ApTemp = (TipoApontador)malloc(sizeof(TipoPagina));
  ApTemp->n = 0;  ApTemp->p[0] = NULL;


  if (i < M + 1)
  { InsereNaPagina(ApTemp, Ap->r[MM-1], Ap->p[MM]);
    Ap->n--;
    InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
  }
  else InsereNaPagina(ApTemp, *RegRetorno, *ApRetorno);
  for (j = M + 2; j <= MM; j++)
    InsereNaPagina(ApTemp, Ap->r[j-1], Ap->p[j]);
  Ap->n = M;  ApTemp->p[0] = Ap->p[M+1];
  *RegRetorno = Ap->r[M];  *ApRetorno = ApTemp;
}
```

```
void Insere(TipoRegistro Reg, TipoApontador *Ap)
{ short Cresceu;
  TipoRegistro RegRetorno;
  TipoPagina *ApRetorno, *ApTemp;
  Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
  if (Cresceu)  /* Arvore cresce na altura pela raiz */
  { ApTemp = (TipoPagina *)malloc(sizeof(TipoPagina));
    ApTemp->n = 1;
    ApTemp->r[0] = RegRetorno;
    ApTemp->p[1] = ApRetorno;
    ApTemp->p[0] = *Ap;   *Ap = ApTemp;
  }
}
```

Remoção na Árvore B

O processo de remoção de um item presente na árvore B sempre será feito com um item presente em uma página folha. Caso o item esteja presente já em uma página folha, o processo de remoção será efetuado diretamente, caso não, alguns processos deverão ser feitos:

- Devemos substituir o item desejado pelo item que contenha adjacente antecessora, ou seja, o item mais a direta da subárvore à esquerda, ou podemos substituir pela chave adjacente sucessora, ou seja, o item mais à esquerda da subárvore à direita
- Após isso, o item será removido



Exemplo de remoção na árvore B (exemplo retirado dos slides)

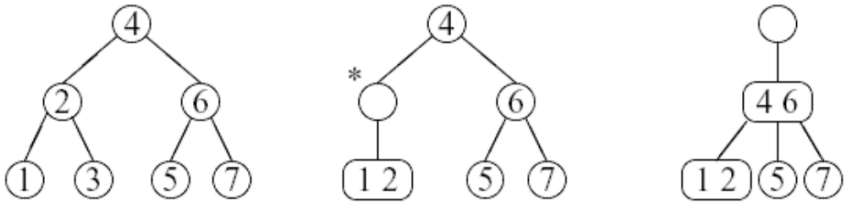
Após a remoção do item da página folha, **deve-se verificar se esta página contém, pelo menos, m itens, ou seja, se a propriedade da árvore B não foi violada**.

Caso ela tenha sido violada, é necessário reconstruir a propriedade da árvore B utilizando de um empréstimo de um item da página vizinha

Existem duas possibilidades para realizar esta operação de empréstimo:

1. Página vizinha possui m itens
    - Já que o número total de itens nas duas páginas é $2m - 1$, as mesmas devem ser fundidas em uma só, tomando emprestado da página pai o item do meio e permitindo liberar uma das páginas.
    - Este processo pode propagar até a página raiz e, ficando a mesma vazia, a página raiz será eliminada, dessa forma causando redução na altura da árvore

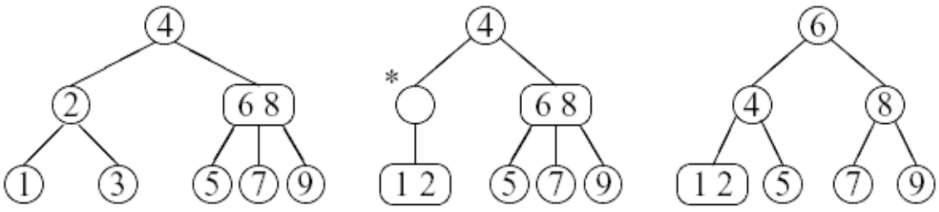
        - Ex.: remoção da chave 3 em uma árvore B de ordem m=1.



38

2. O número de itens na página vizinha é maior que m
    - Deve-se tomar emprestado um item da página vizinha e trazê-lo para a página em questão via página pai

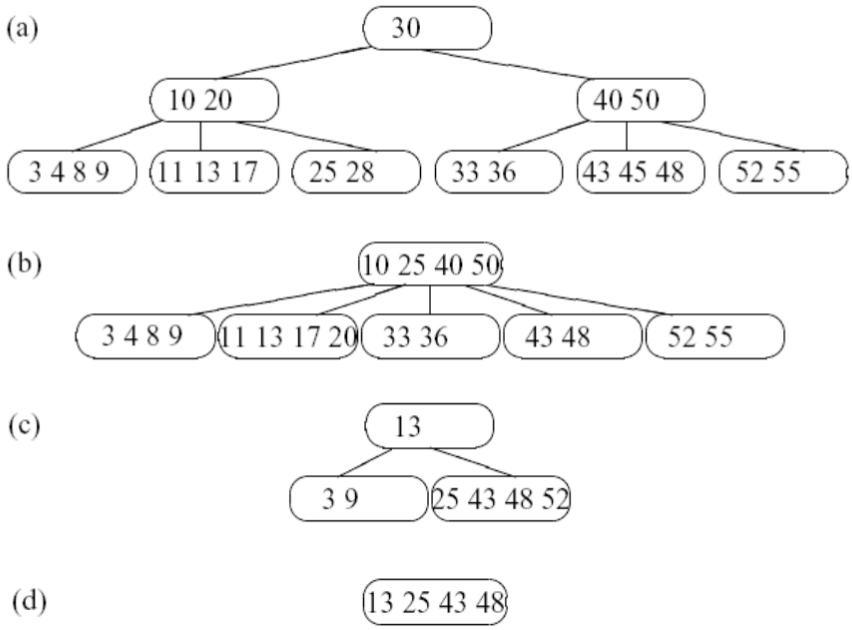
        - Ex.: remoção da chave 3 em uma árvore B de ordem m=1.



> 📌 **Observação**: A fusão de itens é mais eficiente a longo pois favorece a redução da altura da árvore

Exemplo de Remoção Múltipla na Árvore B

- Ex.: remoção das chaves (a) 45, 30, 28; (b) 50, 8, 10, 4, 20, 40, 55, 17, 33, 11, 36; (c) 3, 9, 52.

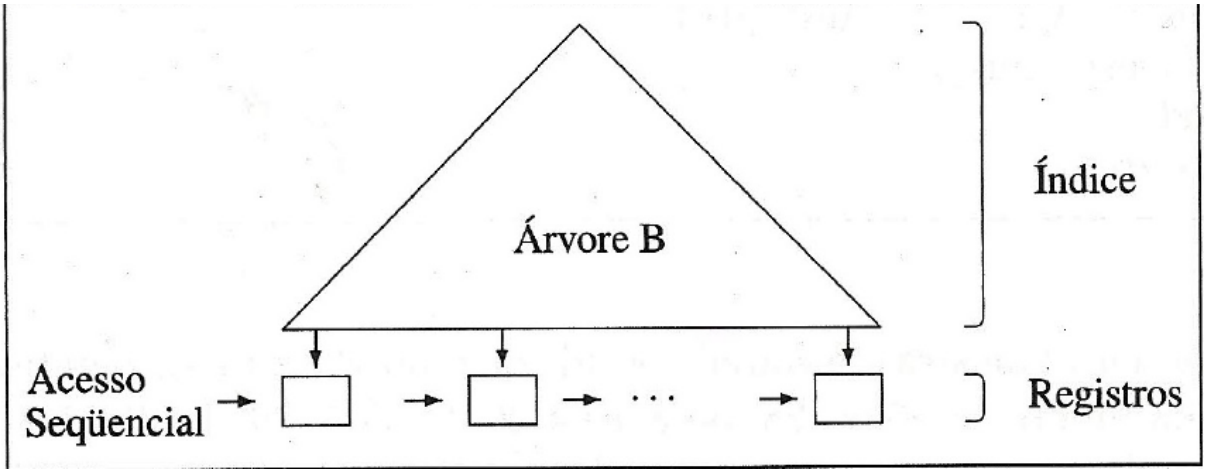


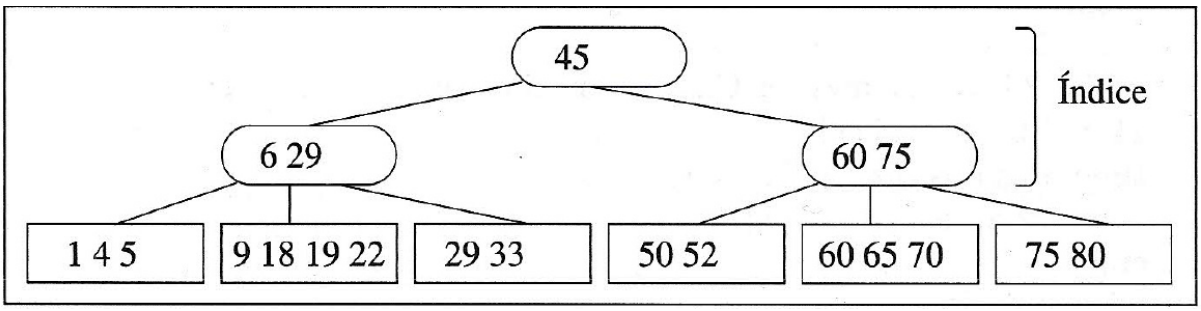
Árvore B* (Estrela)

Conhecida como árvore B estrela (B*), é uma das alternativas para a implementação da árvore B com o intuito de se obter mais eficiência.

Possui como caracteristicas adicionais:

- Todos os itens estão armazenados no último nível, ou seja, o nível das páginas folhas.
- Os níveis acima do último nível constituem um índice cuja a organização é a de uma árvore B





📌 Observação: Os índices são apenas valores de indexação, **não** são chaves "úteis" que serão buscadas em um processo de pesquisa dentro do arquivo

Em uma árvore B*, existe uma separação lógica entre o índice (referente a árvore B) e os itens que constituem o arquivo propriamente dito.

- Estão presente nos índices apenas as chaves dos itens
- Os itens completos encontram-se somente nas páginas folhas

As pesquisas se encerram somente nos nós folhas, diferente da árvore B onde é encerrado assim que a chave buscada é encontrada.
As páginas folhas podem ou não estar conectadas da esquerda para a direita. Dessa forma, permitindo um acesso sequencial mais eficiente do que o acesso via índice.

Devida a esta diferença entre as páginas de índice e as páginas folhas, na árvore B* é necessário que se faça uma categorização das páginas como sendo internas ou externas.

Como não existe a necessidade do uso de ponteiros nas páginas folhas, é possível armazenar uma quantidade superior de itens nas páginas, ou seja, a ordem m é maior.

Isto **não** promove problemas para o algoritmo de inserção, pois as metades de cada página particionada permanecem no mesmo nível da página original antes da partição

Árvore B* (Estrela) - Código das Páginas

```
typedef long TipoChave;

typedef struct TipoRegistro {
    TipoChave Chave;
    /* outros componentes */
} TipoRegistro;

typedef enum {Interna, Externa} TipoIntExt;

typedef struct TipoPagina* TipoApontador;



typedef struct TipoPagina {
    TipoIntExt Pt;
    union {
        struct {
            int ni;
            TipoChave ri[MM];
            TipoApontador pi[MM + 1];
        } U0;
        struct {
            int ne;
            TipoRegistro re[MM2];
        } U1;
    } UU;
} TipoPagina;
```

📌 O `TipoPagina` é chamado de **Registro Seletivo**, onde a estrutura pode se alterar dependendo da situação
Quem indica a estrutura que iremos trabalhar (interna ou externa) é a variável `Pt`
Quem permite o acesso às variáveis internas das estruturas é `UU`

Deve-se notar que é possível se utilizar ordens diferentes (MM ou $MM2$) quando se trabalha com a árvore B* por conta desta diferença entre as estruturas das páginas

Pesquisa na Árvore B* (Estrela)

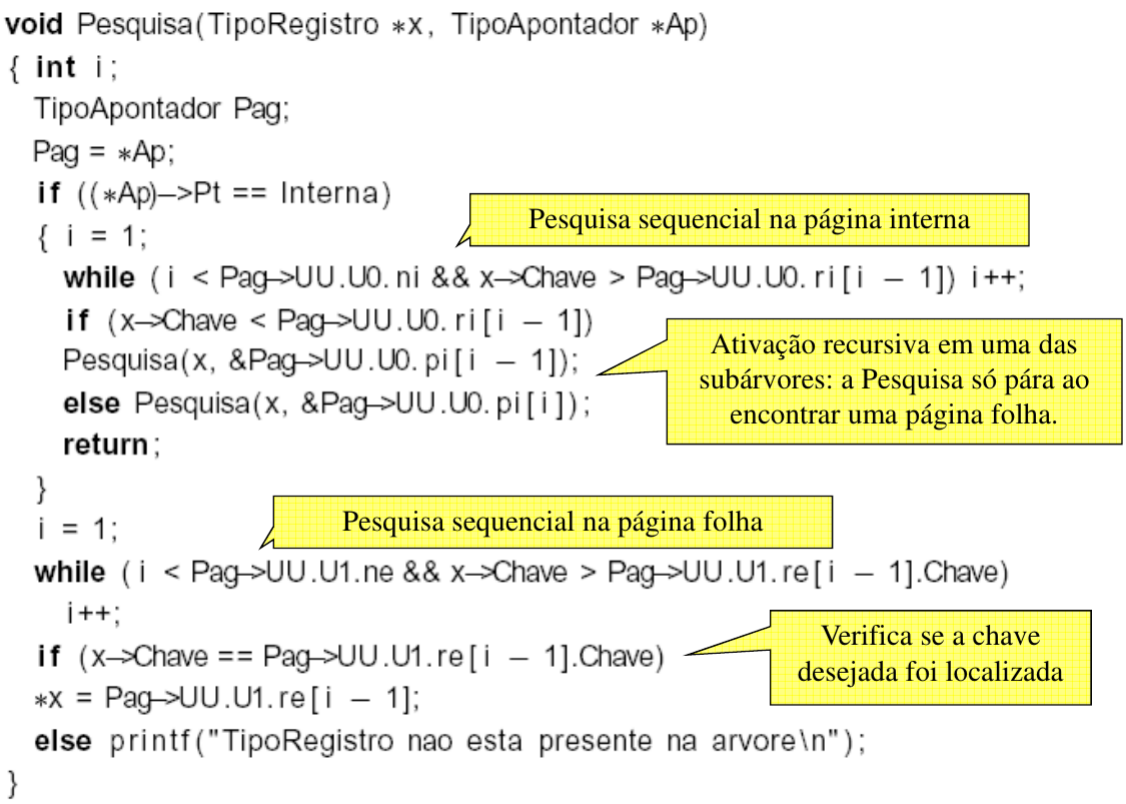
A operação de pesquisa na árvore B* é relativamente semelhante à pesquisa em uma árvore B, sendo que:

- A pesquisa sempre leva até uma página folha
- Os valores encontrados ao longo do caminho são irrelevantes desde de que conduzam à página folha correta

Com diferença que:

- Ao encontrar a chave buscada em uma página do índice, a pesquisa não finaliza. No caso, o apontador da direita é seguido e a pesquisa continua até que se encontre uma página folha.

Pesquisa na Árvore B* (Estrela) - Código



Código de pesquisa na árvore B* (exemplo retirado dos slides)

Inserção na Árvore B* (Estrela)

A operação de inserção na árvore B* é muito similar à inserção de um item na árvore B no geral. Porém com a seguinte diferença:

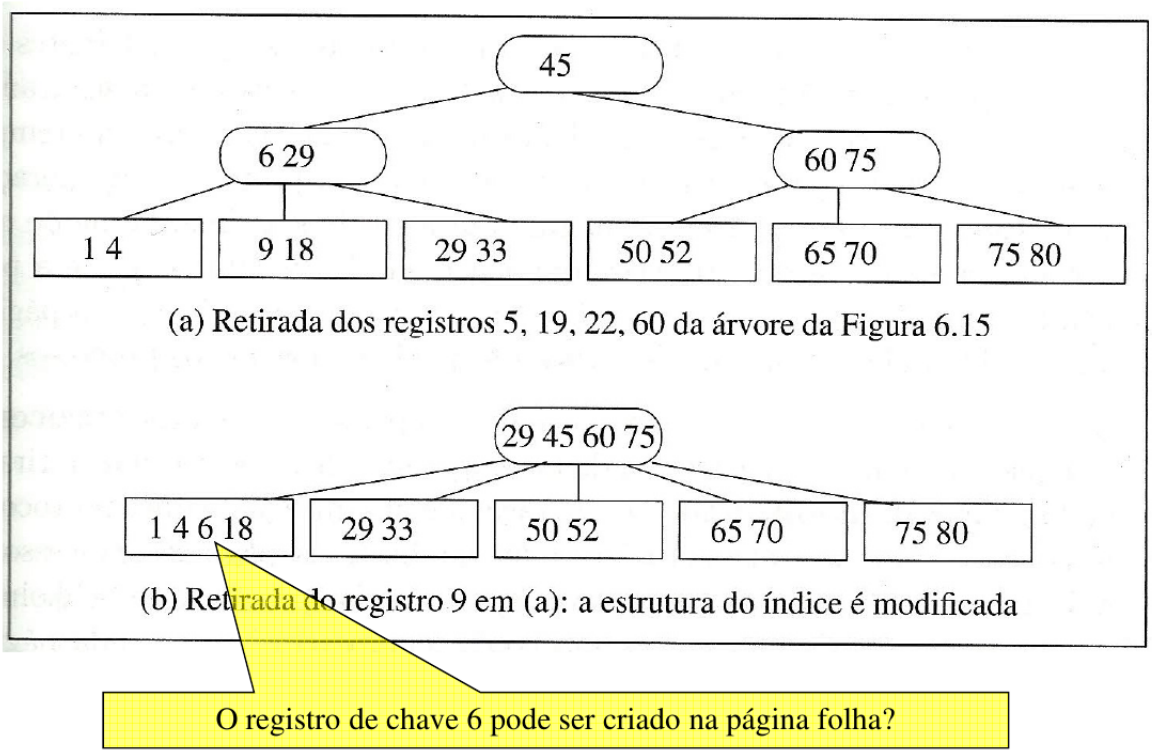
- Quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave que pertence ao item do meio para página pai no nível inferior, retendo o próprio item do meio na página folha da direita

Remoção na Árvore B* (Estrela)

A operação de remoção em uma árvore B* é relativamente mais simples do que em uma árvore B.

- O item a ser removido reside sempre em uma página folha, não havendo necessidade de se localizar o item com a chave antecessora
- Caso a folha fique com pelo menos m itens, as páginas do índice não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao item a ser removido esteja no
  índice

- As páginas do índice precisarão ser modificadas apenas se a folha ficar com uma quantidade de itens menor que m



Exemplo de remoção em uma árvore B* (exemplo retirado dos slides)