DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

# Estruturas de Dados

## Pesquisa em memória primária

Professores: Anisio Lacerda
Wagner Meira Jr.

# Pesquisa em Memória Primária

- Pesquisa:
  - Recuperação de informação em um grande volume de dados.
  - Informação é dividida em registros e cada registro contém uma chave.

- Objetivo:
  - Encontrar itens com chaves iguais a chave dada na pesquisa.

- Aplicações:
  - Contas em um banco
  - Reservas de uma companhia aérea
  - Buscar o histórico escolar de um aluno
  - Etc, etc, etc...

# Pesquisa em Memória Primária

- Escolha do método de busca
  - Quantidade de dados envolvidos.
  - Frequência com que operações de inserção e retirada são efetuadas.

- Métodos de pesquisa:
  - Pesquisa sequencial
  - Pesquisa binária
  - Árvore de pesquisa
    - Árvores binárias de pesquisa sem balanceamento
    - Árvores binárias de pesquisa com balanceamento
  - Hashing

# Tabelas de Símbolos

- Tabelas são também conhecidas como **dicionários**
  - Chaves – palavras
  - Item – entradas associadas as palavras (significado, pronúncia)

- Estrutura de dados contendo itens com chaves que suportam as operações
  - Inserção e Remoção de um item
  - Retorno de um item que contém uma determinada chave.

# Tipo Abstrato de Dados

- Considerar os algoritmos de pesquisa como tipos abstratos de dados (TADs), com um conjunto de operações associado a uma estrutura de dados,
  - Há independência de implementação para as operações.
- Operações:
  - Inicializar a estrutura de dados
  - Pesquisar um ou mais registros com uma dada chave
  - Inserir um novo registro
  - Remover um registro específico
  - Ordenar os registros

# PESQUISA SEQUENCIAL

# Pesquisa Sequencial

- Método de pesquisa mais simples
  - A partir do primeiro registro, pesquisa sequencialmente até encontrar a chave procurada
- Registros ficam armazenados em um vetor (arranjo).
- Inserção de um novo item
  - Adiciona no final do vetor.
- Remoção de um item com chave específica
  - Localiza o elemento, remove-o e coloca o último item do vetor em seu lugar.

# Pesquisa Sequencial

```
#define MAX 1000

class Tabela {
 public:
   typedef int TipoChave;
   struct Registro {
      TipoChave Chave;
      /* Outros campos */
   };
   int Pesquisa(TipoChave x);

 protected:
   int n;
   Registro Item[MAX + 1];
}
```

# Pesquisa Sequencial

```
/* retorna 0 se não encontrar um registro com a chave x */

int Tabela::Pesquisa(TipoChave x) {

  Item[0].Chave = x; /* sentinela */
  int i = n + 1;
  do {
     i--;
  } while (Item[i].Chave != x);
  return i;
}
```

- Complexidade de tempo?
  - *O(n)*

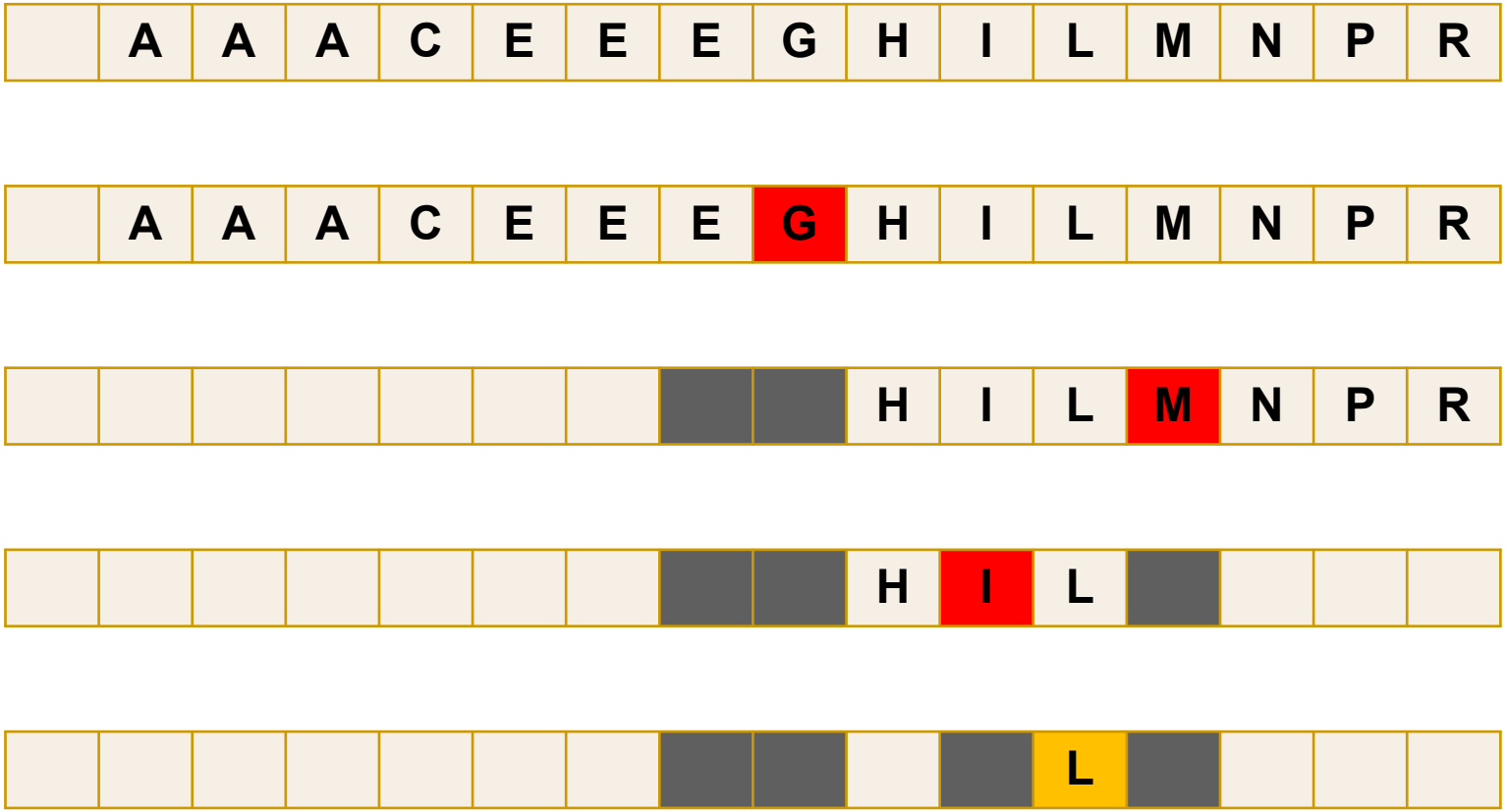
# PESQUISA BINÁRIA

# Pesquisa Binária

- Redução do tempo de busca aplicando o paradigma dividir para conquistar.
- Se aplica à situação específica: chaves estão ordenadas.
- Ideia geral:
  1. Divide o vetor em duas partes
  2. Verifica em qual das partes o item com a chave se localiza
  3. Concentra-se apenas naquela parte

# Pesquisa Binária

- Exemplo: pesquisa pela chave L



# Pesquisa Binária

```
#define MAX 1000

class TabelaBinaria {
 public:
   typedef int TipoChave;
   struct Registro {
      TipoChave Chave;
      /* Outros campos */
   };
   int Pesquisa(TipoChave x);

 protected:
   int n;
   Registro Item[MAX + 1];
}
```

# Pesquisa Binária

```
int TabelaBinaria::Pesquisa(TipoChave x) {
  if (n == 0)return 0;
  int mid;
  int Esq = 1;
  int Dir = n;
  do {
    mid = Dir - ((Esq - Dir) / 2);
    if (x > Item[i].Chave)
      Esq = mid + 1; /* procura na partição direita */
    else
      Dir = mid - 1; /* procura na partição esquerda */
  } while ((x != Item[mid].Chave) && (Esq <= Dir));
  if (x == Item[mid].Chave)
    return mid;
  return 0;
}
```

Compara a chave x com elemento do meio do vetor

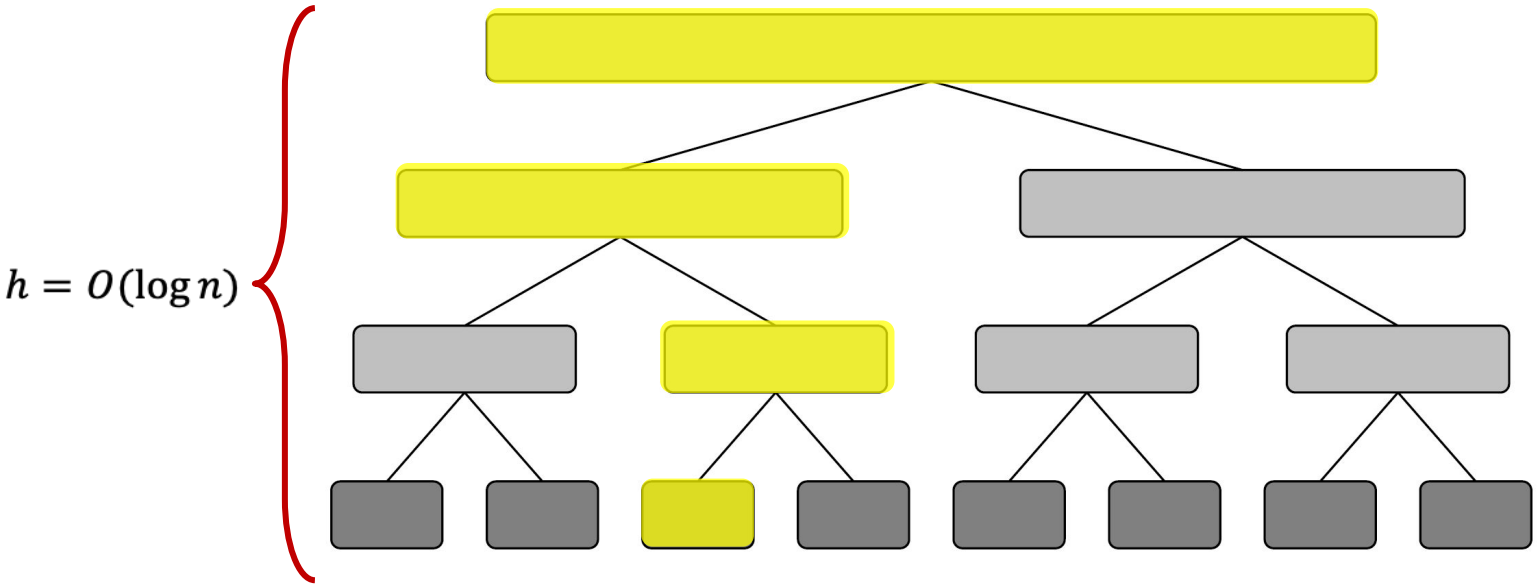
Define em que lado do vetor procurar

Verifica se achou o elemento procurado

Verifica se deve continuar procurando (ainda não achou e ainda tem elementos para procurar)

# Pesquisa Binária - Análise

- **Complexidade**
  - A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.




  - Logo, o número de vezes que o tamanho da tabela é dividido ao meio é cerca de **log n.**

# Pesquisa Binária - Análise

- Ressalva - chaves precisam estar ordenadas
  - Manter chaves ordenadas na inserção pode levar a comportamento quadrático.
  - Se chaves estiverem disponíveis no início, um método de ordenação rápido pode ser usado.
  - Alto custo para manter a tabela ordenada: a cada inserção (e retirada) na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes. O(n)
  - Portanto, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

# Árvore Binária de Pesquisa

# Árvore Binária de Pesquisa

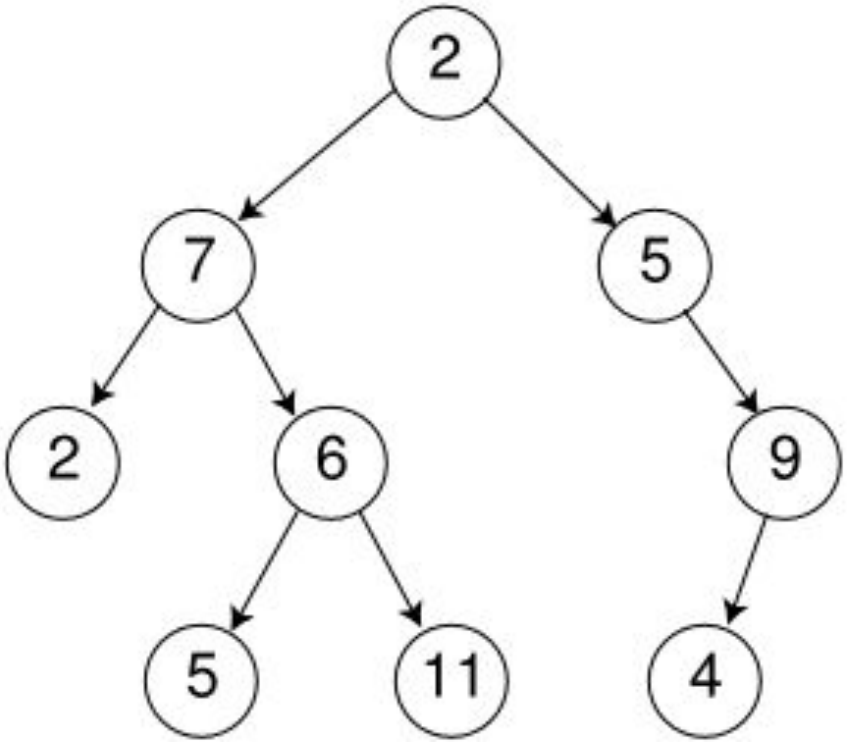
- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.
- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:

1. Acesso direto e seqüencial eficientes.
2. Facilidade de inserção e retirada de registros.
3. Boa taxa de utilização de memória.
4. Utilização de memória primária e secundária.

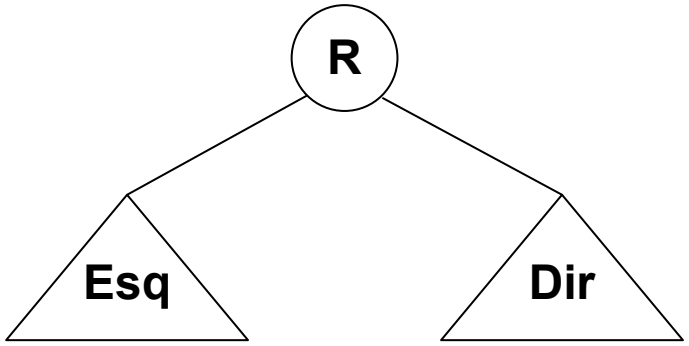
# Árvore Binária de Pesquisa

- Árvore Binária
  - Cada nó tem no máximo dois filhos

- Obs: a árvore ao lado não impõe nenhuma ordenação em seus nodos

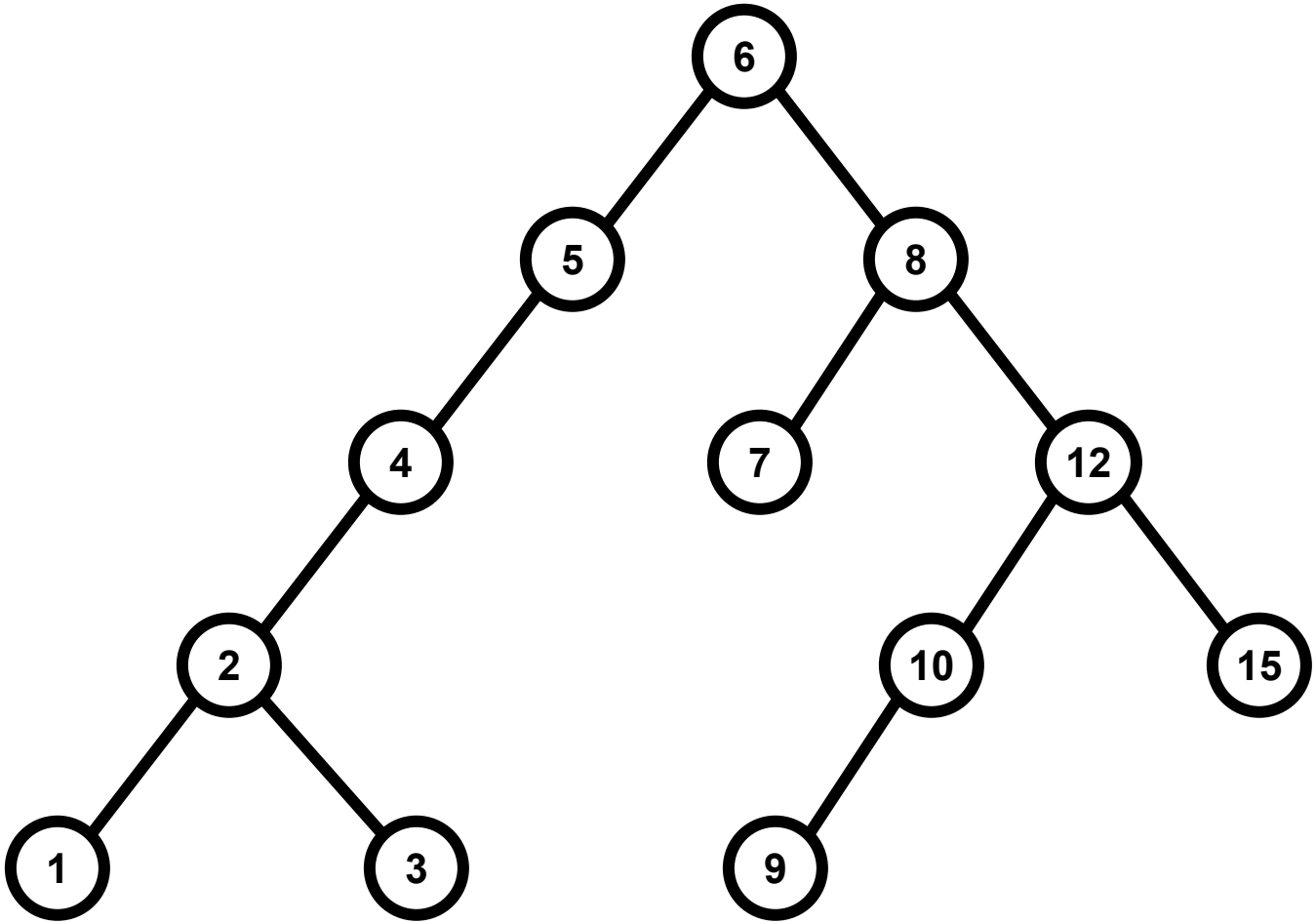


# Árvore Binária de Pesquisa




- Árvores de pesquisa mantêm uma ordem entre seus elementos
  - Raiz é maior que os elementos na árvore à esquerda
  - Raiz é menor que os elementos na árvore à direita

# Árvore Binária de Pesquisa: Exemplo



# Tabela com Árvore Binária

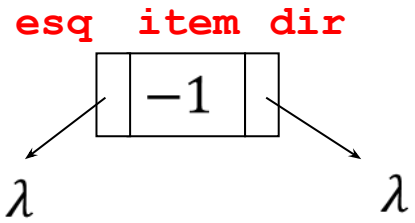
```
class TabelaArvoreBinaria
{
    public:
        TabelaArvoreBinaria();
        ~TabelaArvoreBinaria();
        TipoItem Pesquisa(TipoItem chave);
        void Insere(TipoItem item);
        void Remove(TipoItem chave);

    private:
        TipoItem PesquisaRecursivo(TipoNo* p, TipoChave chave);
        void InsereRecursivo(TipoNo* &p, TipoItem item);
        void ApagaRecursivo(TipoNo* p);
        void Antecessor(TipoNo *q, TipoNo* &r)
        TipoNo *raiz;
};
```

# Mesma Classe de Árvores Binárias

- Classe para representar os Nós da Árvore
  - **TipoItem item**: armazena o item
  - **Esq** e **Dir**: apontadores para as subárvores da direita e esquerda

```
class TipoNo
{
    TipoNo();
    TipoItem item;
    TipoNo *esq;
    TipoNo *dir;
};
```

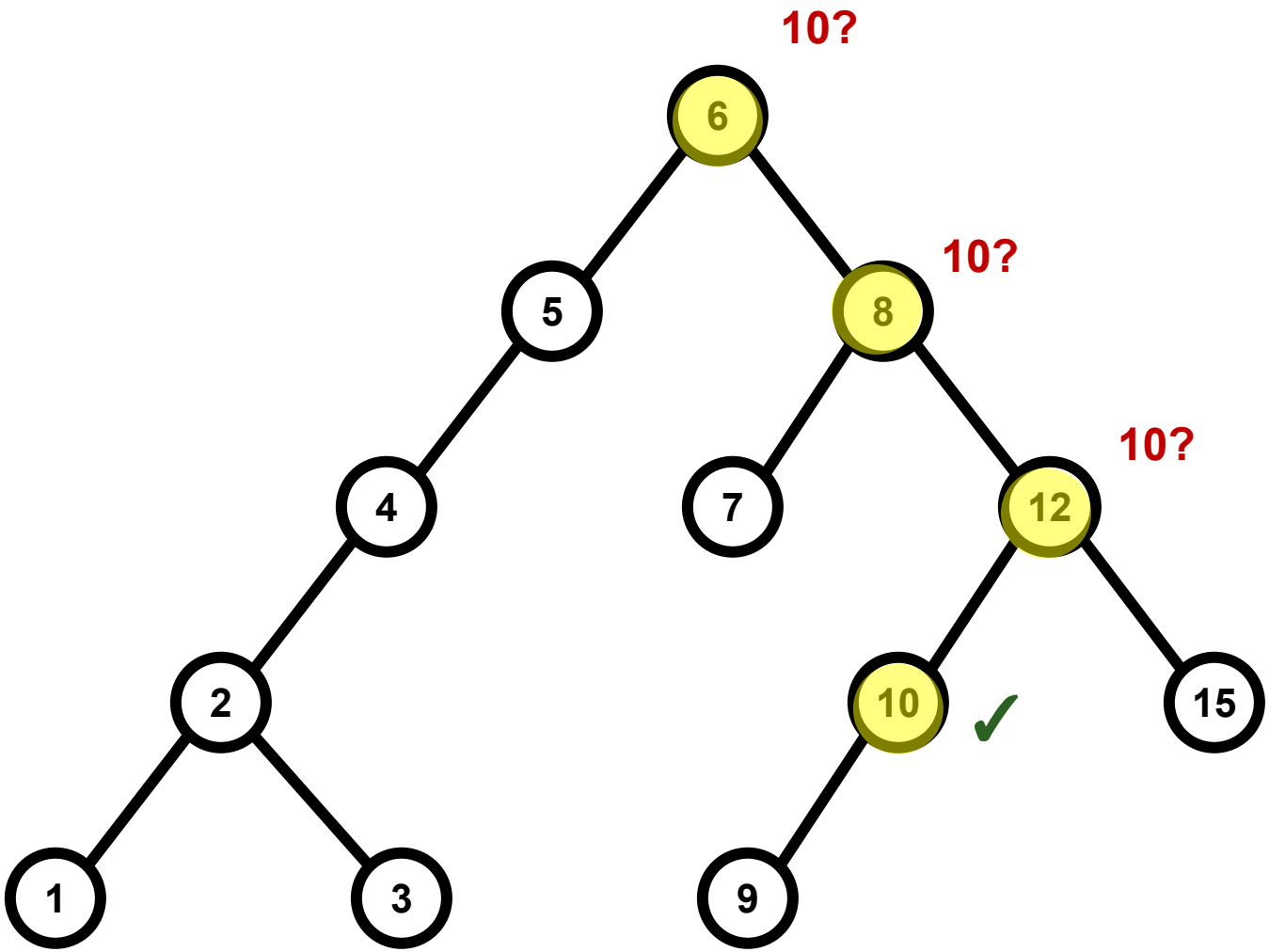
```
TipoNo::TipoNo()
{
    item.SetChave(-1);
    esq = NULL;
    dir = NULL;
}
```



# Árvore Binária de Pesquisa: Pesquisa

- Para encontrar um registro com uma chave x:
  - Compare-a com a chave que está na raiz.
  - Se x é menor, vá para a subárvore esquerda.
  - Se x é maior, vá para a subárvore direita.
  - Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha seja atingido.
  - Se a pesquisa tiver sucesso então o conteúdo do registro retorna no próprio registro x.

# Árvore Binária de Pesquisa: Exemplo



# Árvore Binária de Pesquisa: Pesquisa

```
TipoItem TabelaArvoreBinaria::PesquisaRecursivo(TipoNo *no,
TipoChave chave) {

    TipoItem aux;
    if (no == NULL) {
        aux.SetChave(-1); // Flag para item não presente
        return aux;
    }

    if (chave < no->item.GetChave())
        return PesquisaRecursivo(no->esq, chave);
    else if (chave>no->item.GetChave())
        return PesquisaRecursivo(no->dir, chave);
    else
        return no->item;
}
```

O nó não está na árvore

Chave procurada menor que a chave do nó → procura na sub-árvore da esquerda

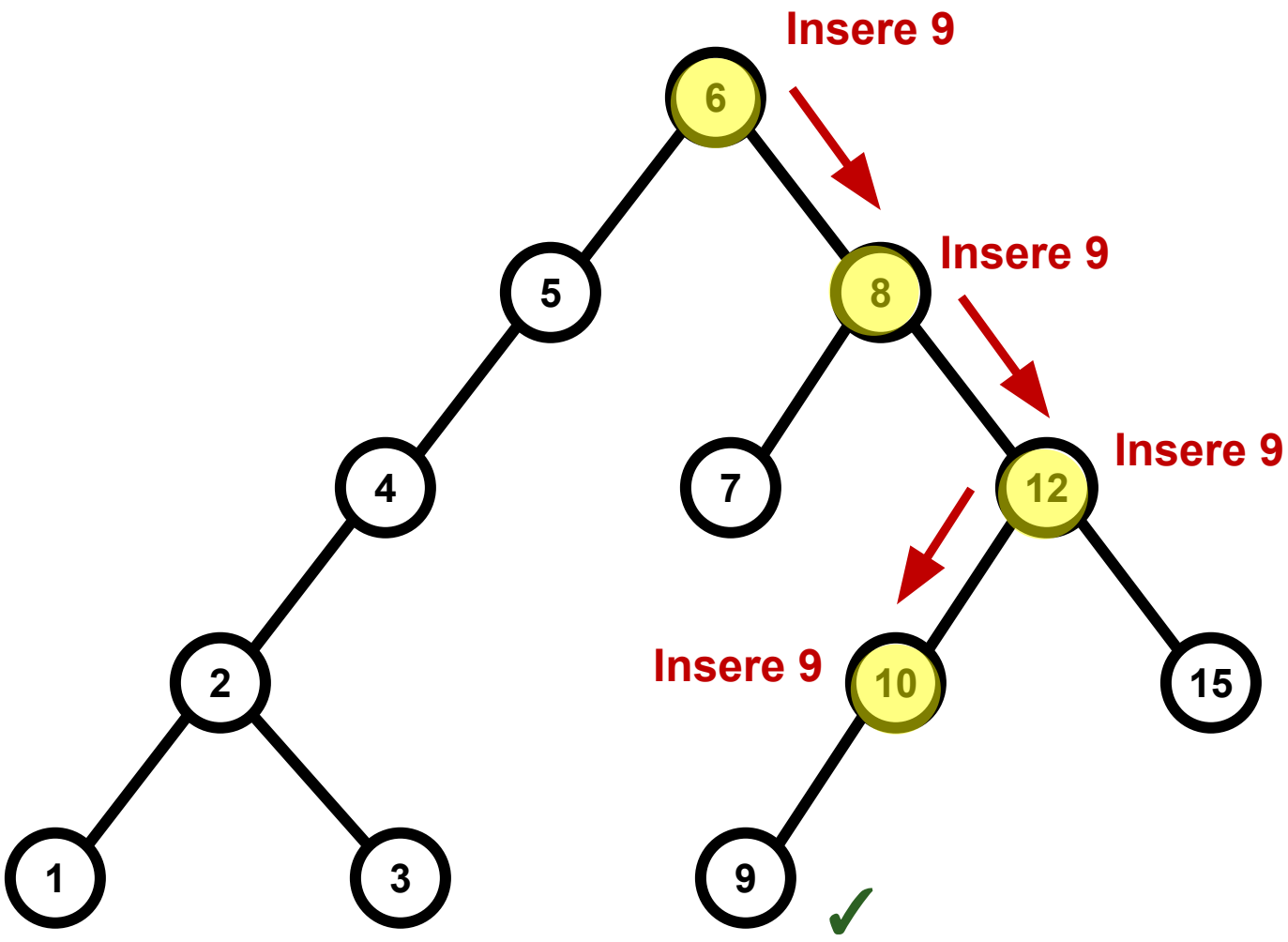
Chave procurada maior que a chave do nó → procura na sub-árvore da direita

Achou! Retorna a chave

# Árvore Binária de Pesquisa: Inserção

- O elemento vai ser inserido como uma *folha* da árvore de busca
- Basicamente, fazemos uma pesquisa e o ponto onde a função encontrar um apontador nulo, será o ponto de inserção
- Método Recursivo para Inserção
  - Compara item com o elemento da Raiz
    - Se menor: insere na subárvore da esquerda
    - Se maior: insere na subárvore da direita
    - Quando a raiz for nula, insere item.

# Árvore Binária de Pesquisa: Exemplo



# Árvore Binária de Pesquisa: Inserção

```
void TabelaArvoreBinaria::InsereRecursivo(TipoNo* &p,
TipoItem item){

    if(p==NULL){
        p = new TipoNo();
        p->item = item;
    }
    else{
        if(item.GetChave() < p->item.GetChave())
            InsereRecursivo(p->esq, item);
        else
            InsereRecursivo(p->dir, item);
    }
}
```

**Achou a posição nula onde o item deveria entrar, insere o item nesta posição**

**Chave a ser inserida é menor que a chave do nó → procura na sub-árvore da esquerda**

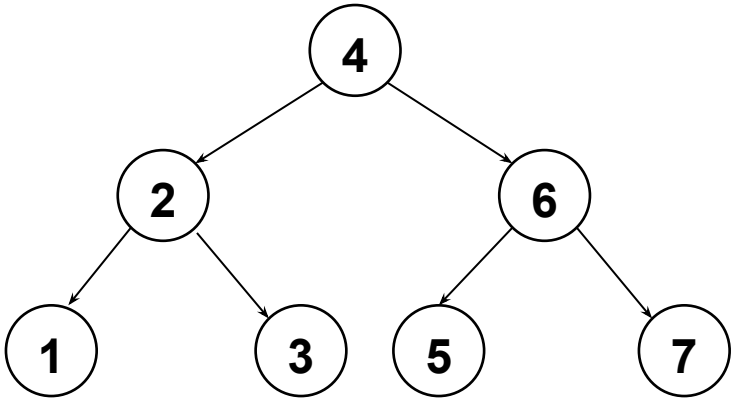
**Senão (chave a ser inserida é maior que a chave do nó) → procura na sub-árvore da direita**

*(Mesmo código apresentado na aula sobre Árvore)*

# Impacto da Entrada na Árvore

- O formato da árvore depende da ordem da entrada dos dados

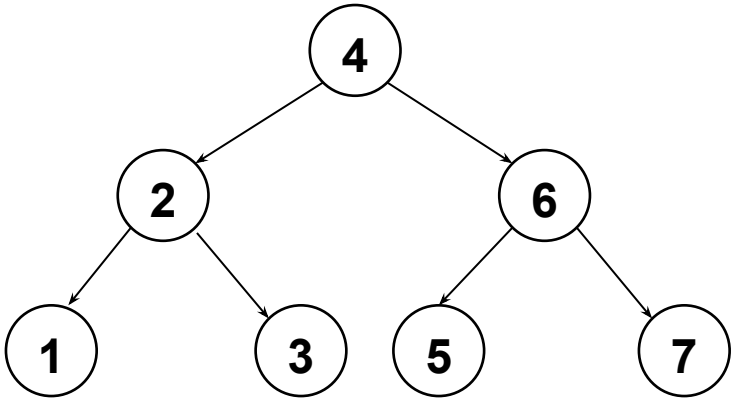
**Entrada: 4, 2, 3, 6, 7, 1, 5**



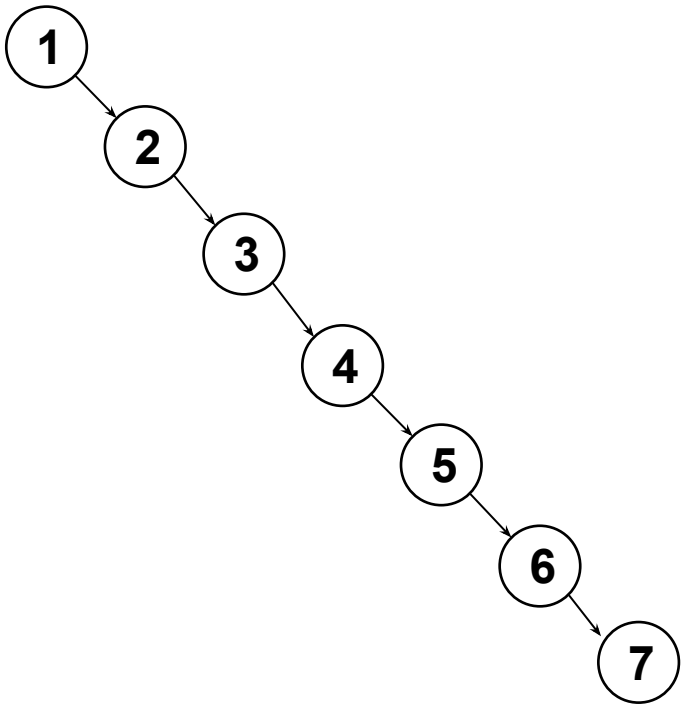
# Impacto da Entrada na Árvore

- O formato da árvore depende da ordem da entrada dos dados.

**Entrada: 4, 2, 3, 6, 7, 1, 5**

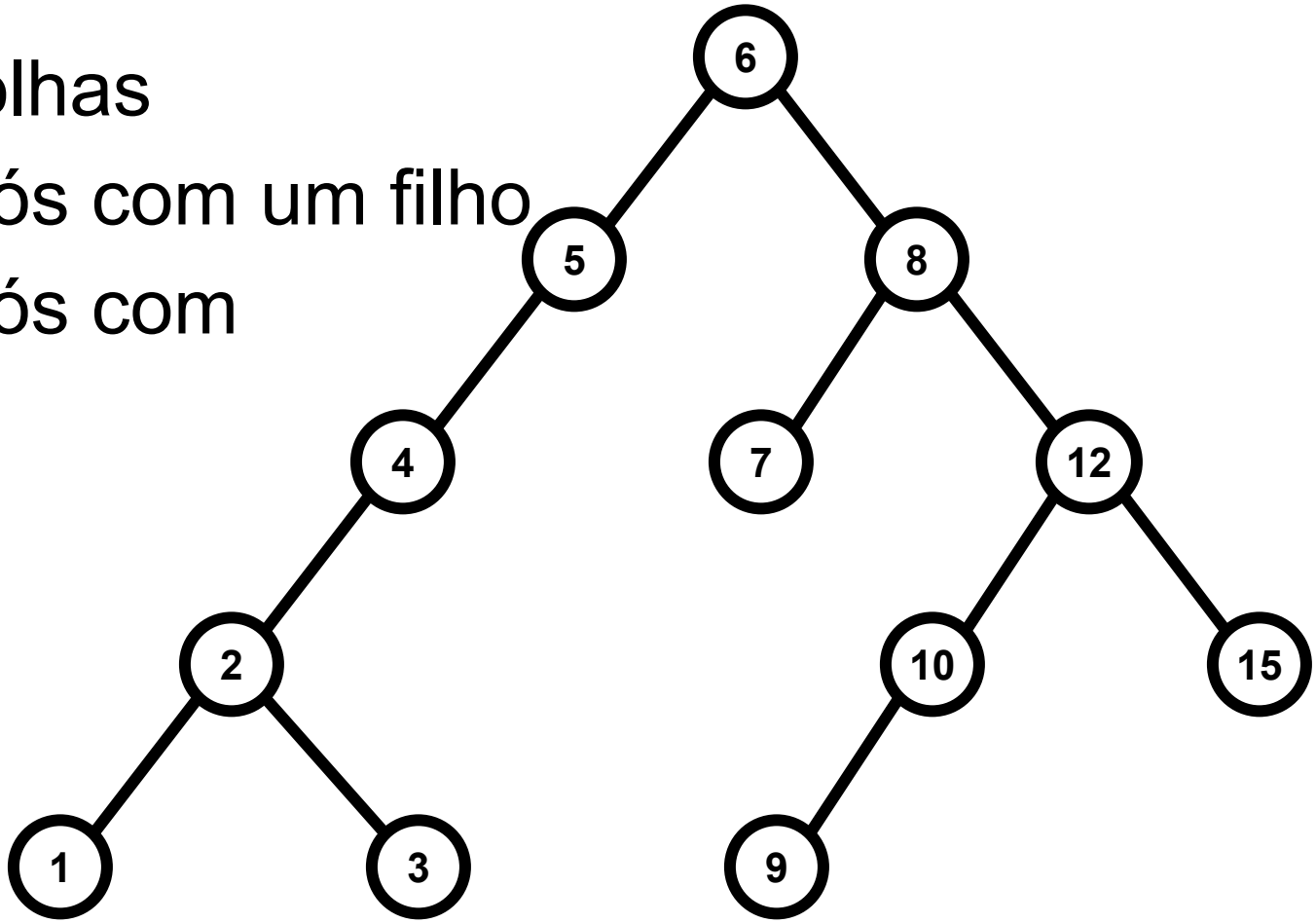



**Entrada: 1, 2, 3, 4, 5, 6, 7**



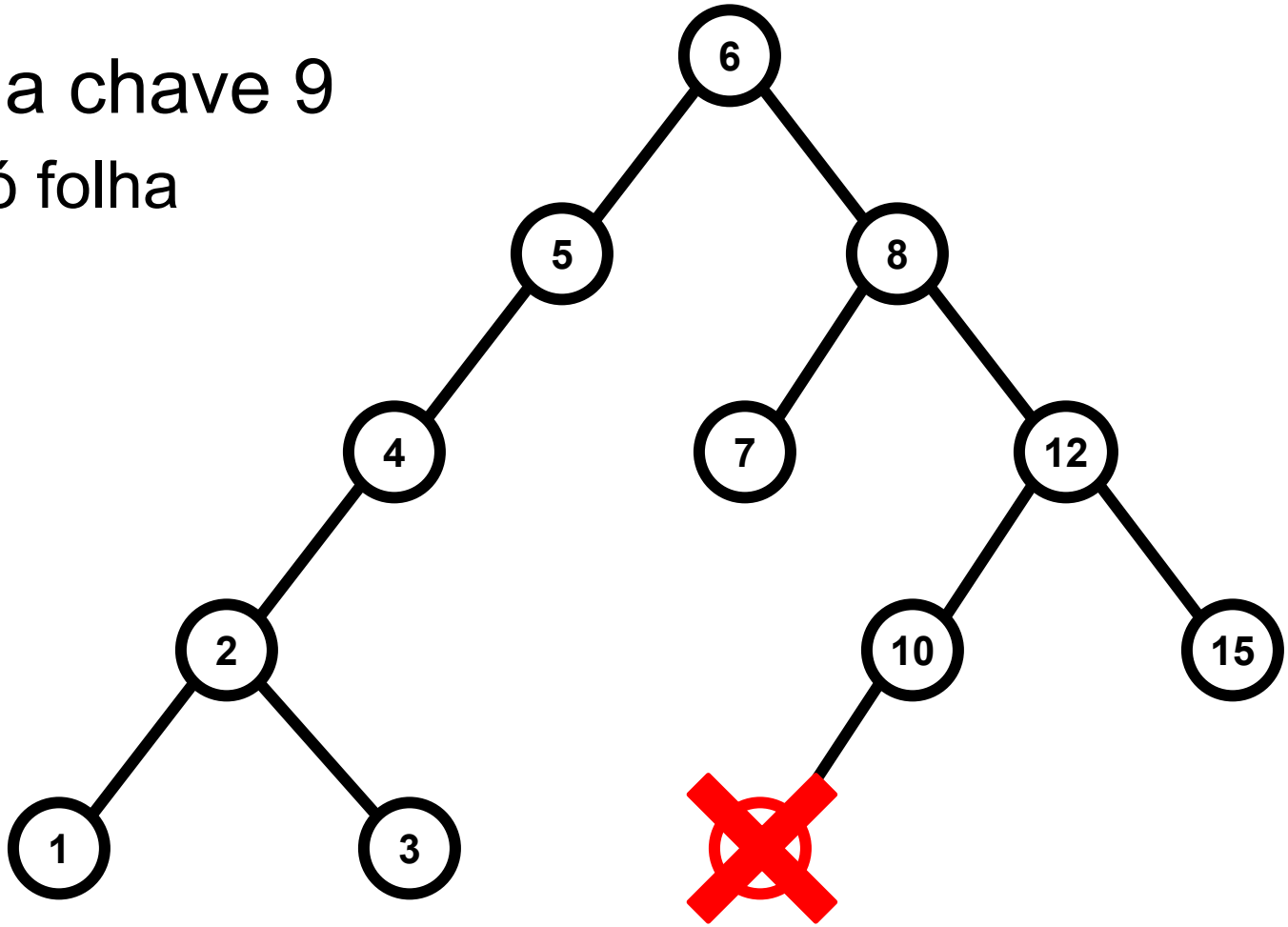
# Árvore Binária de Pesquisa: Remoção

- Remover folhas
- Remover nós com um filho
- Remover nós com dois filhos



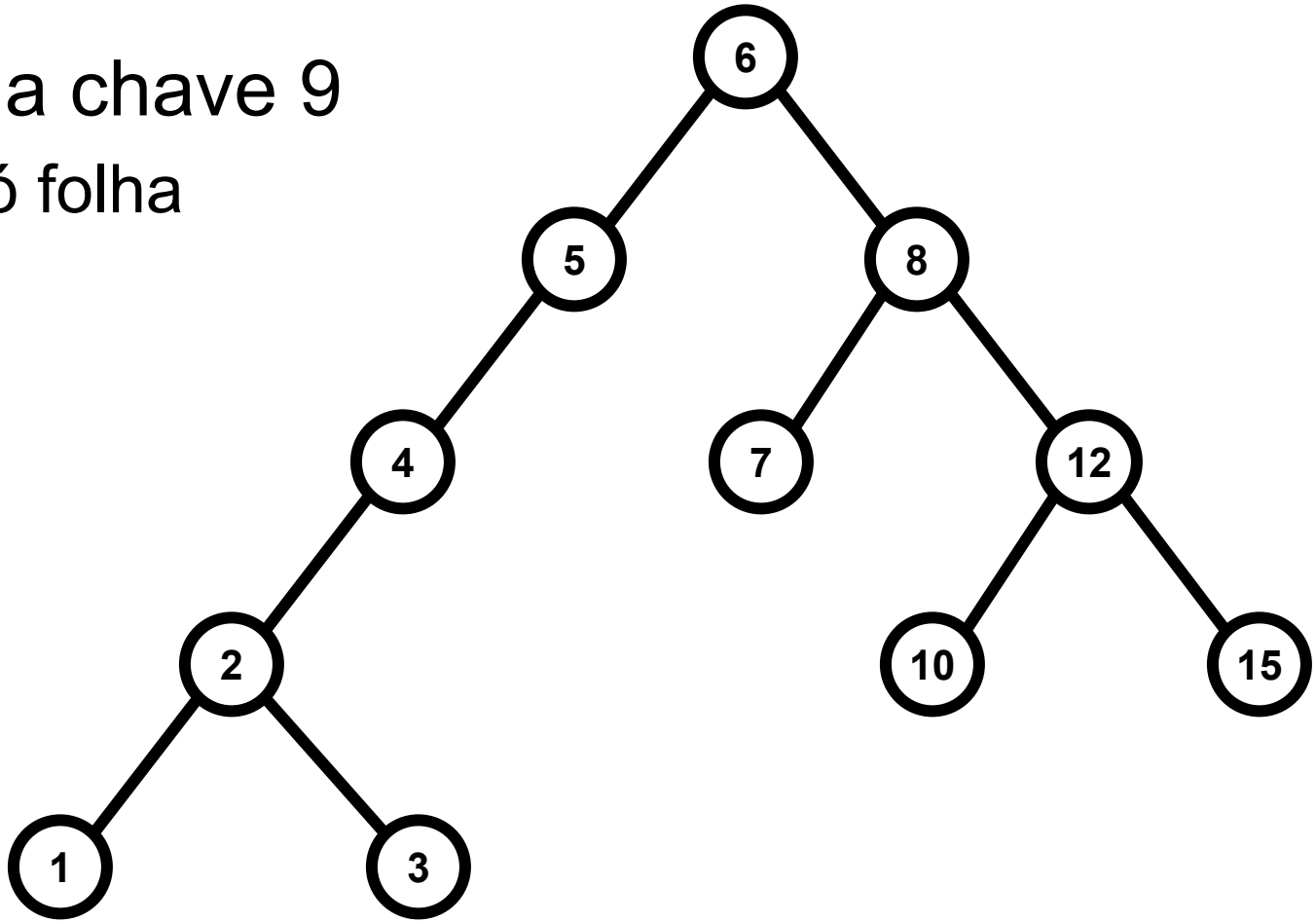
# Remoção: Nó é folha

- Remoção da chave 9
  - Apaga o nó folha



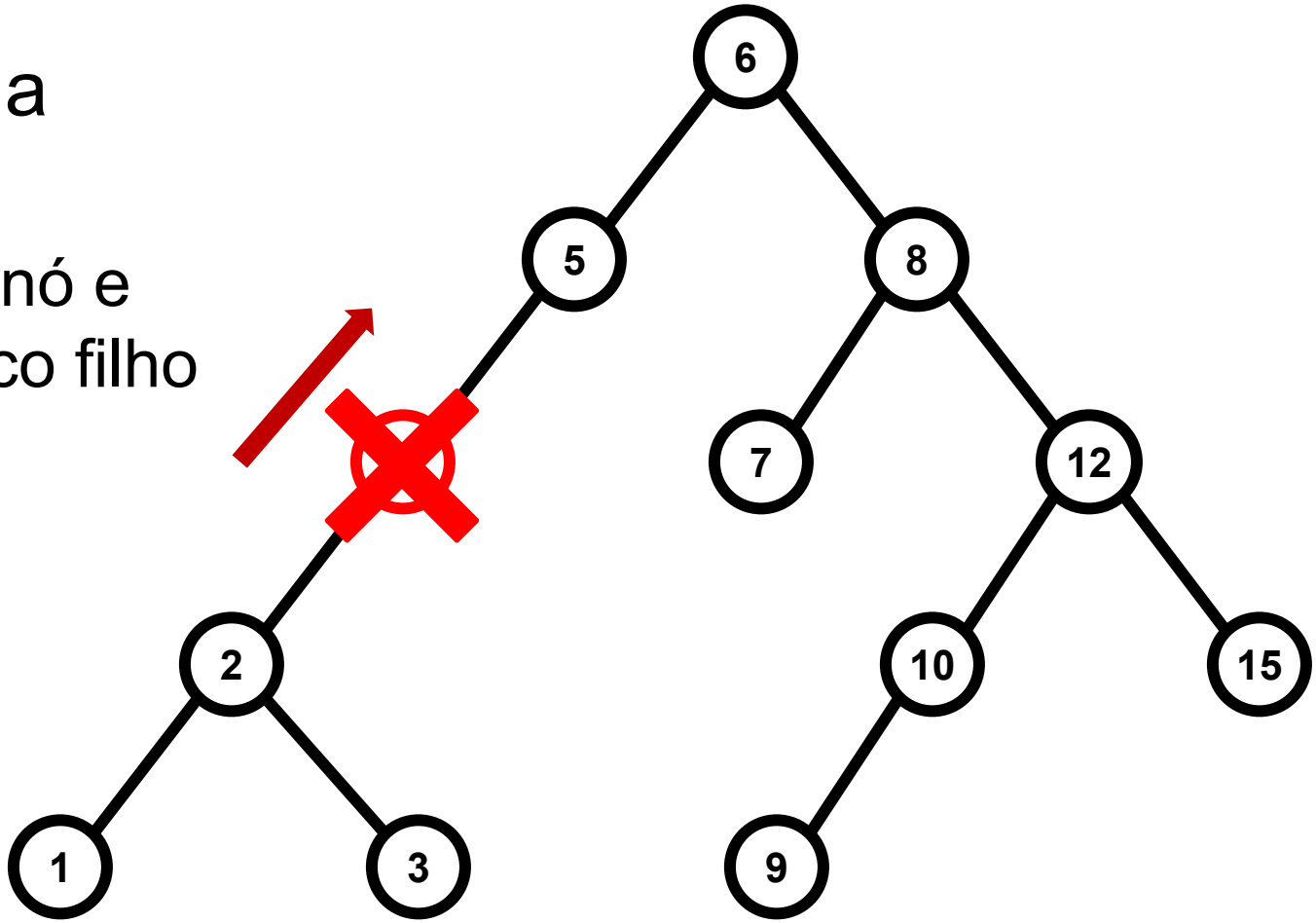
# Remoção: Nó é folha

- Remoção da chave 9
  - Apaga o nó folha



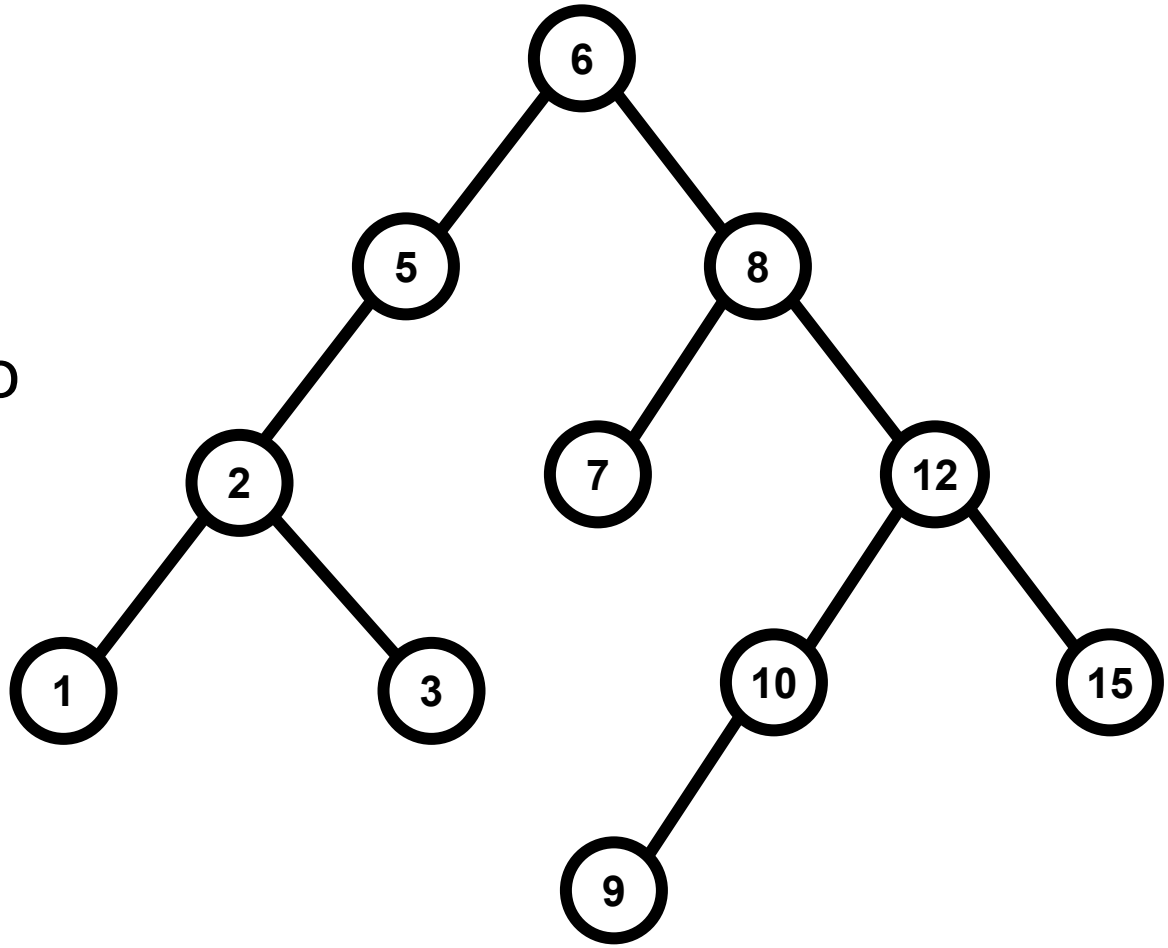
# Remoção: Nó só tem 1 filho

- Remoção da chave 4
  - Remove o nó e sobe o único filho



# Remoção: Nó só tem 1 filho

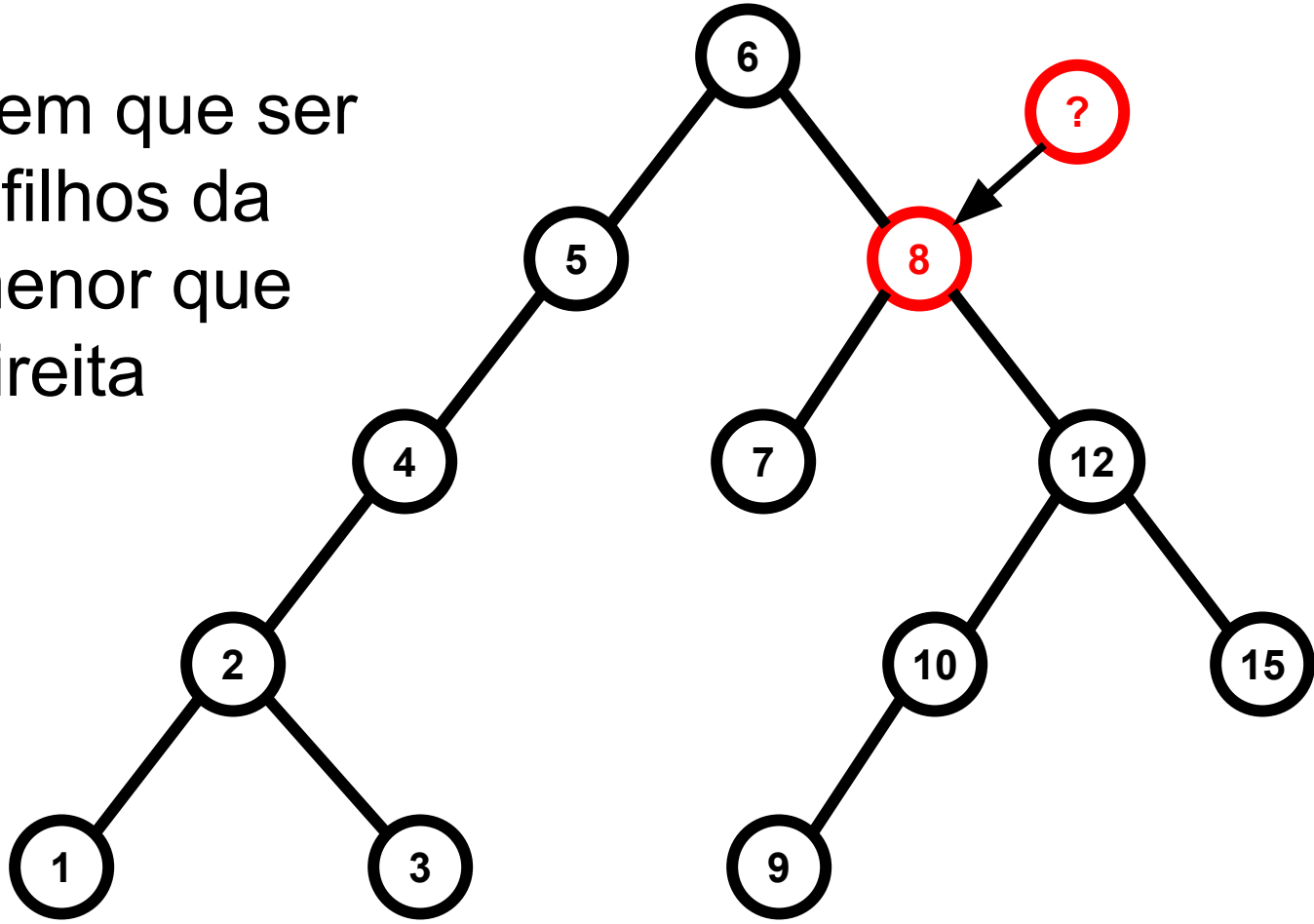
- Remoção da chave 4
  - Remove o nó e sobe o único filho



# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Candidato – tem que ser maior que os filhos da esquerda e menor que os filhos da direita



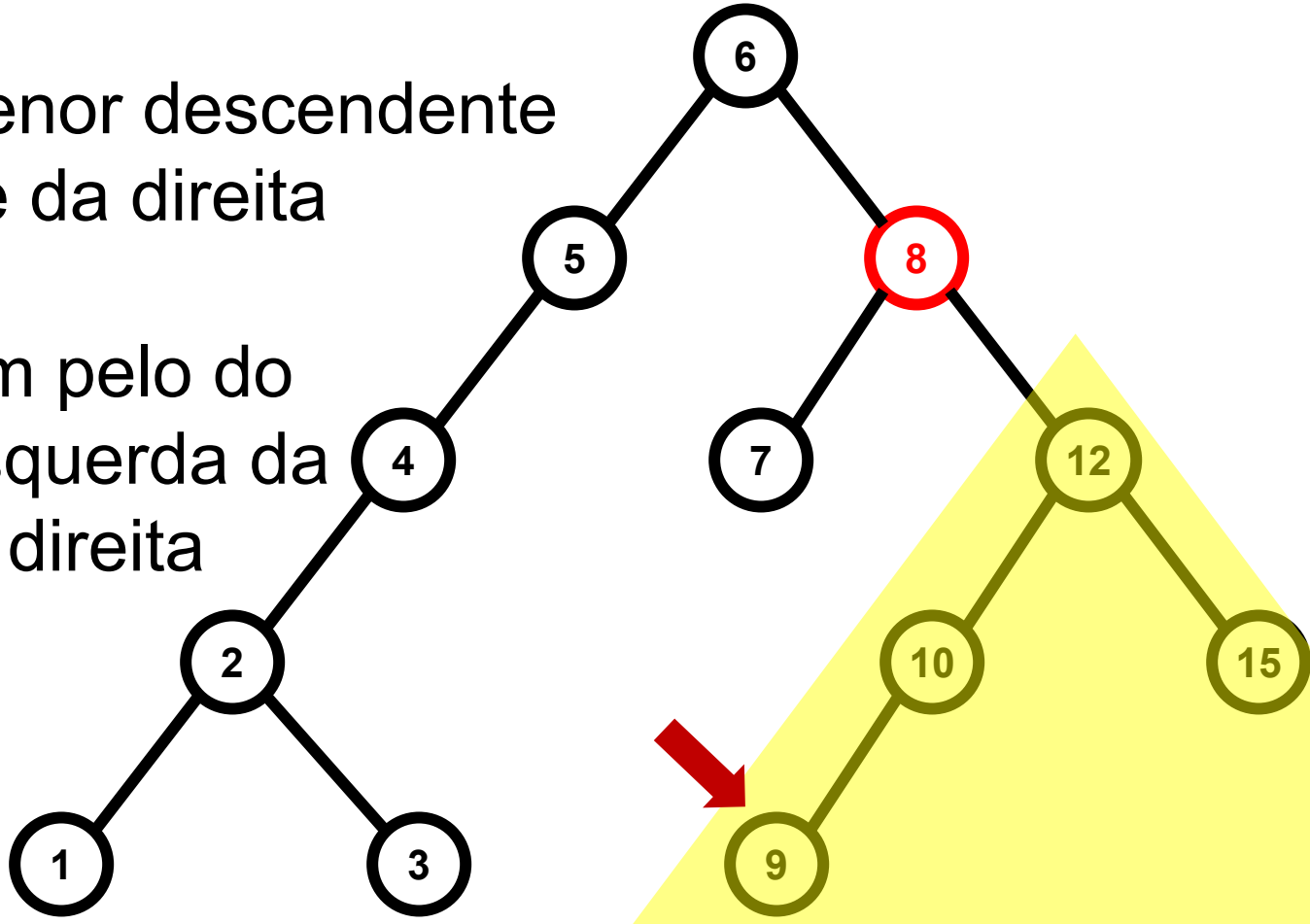
# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Opção 1 – menor descendente da sub-árvore da direita

Substitui o item pelo do filho mais à esquerda da sub-árvore da direita *(sucessor);*

remove este



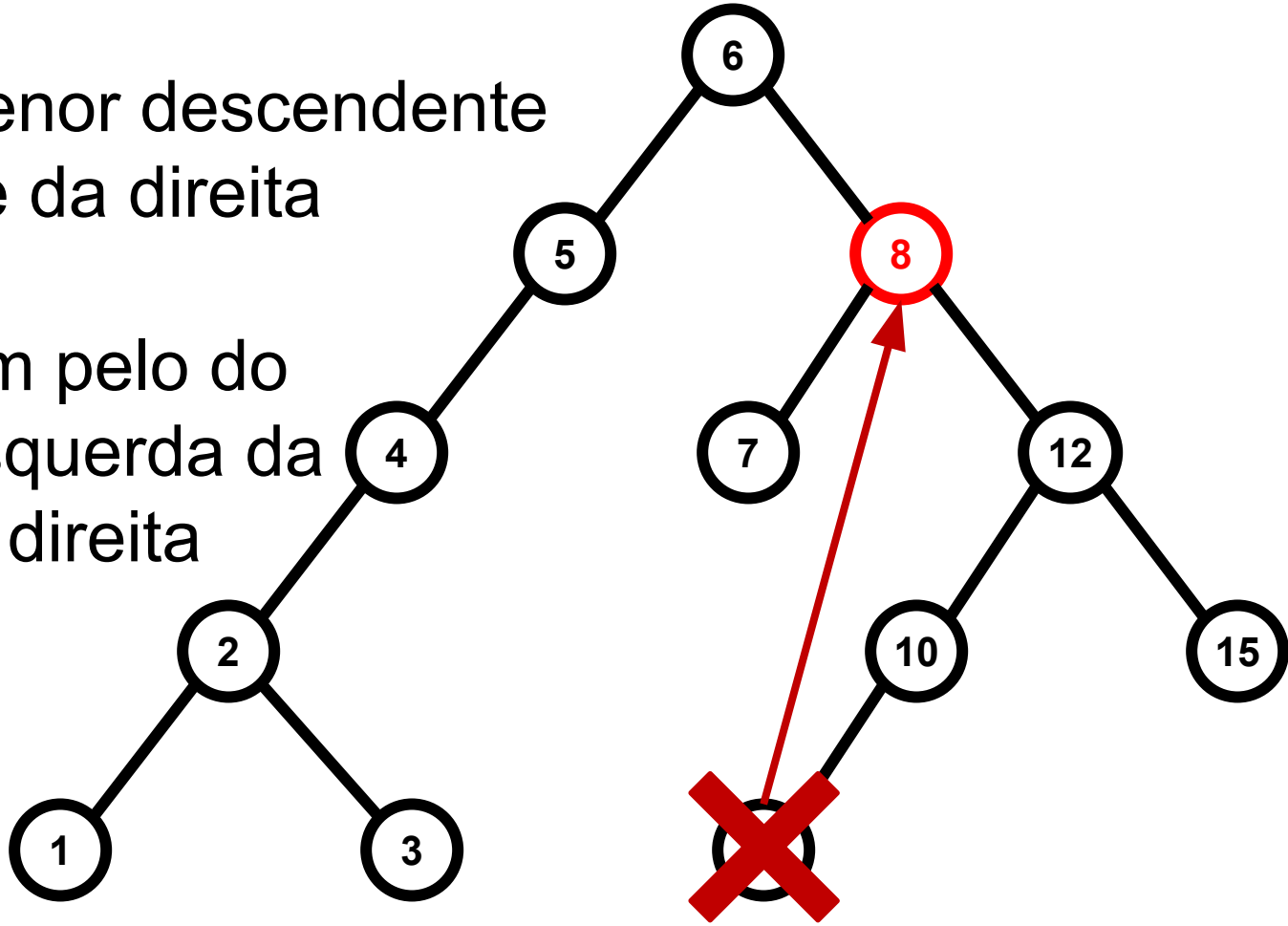
# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Opção 1 – menor descendente da sub-árvore da direita

Substitui o item pelo do filho mais à esquerda da sub-árvore da direita *(sucessor);*

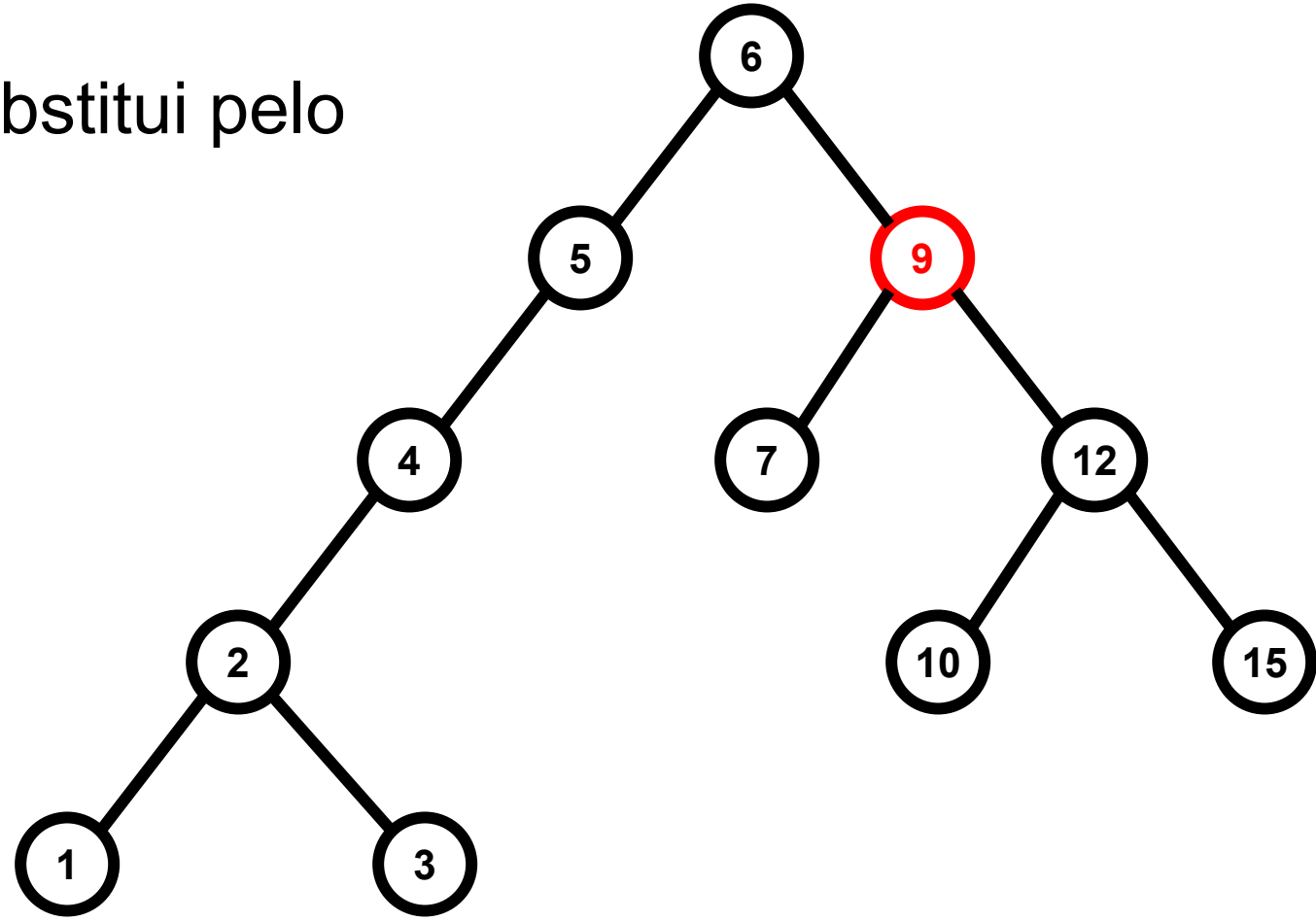
remove este



# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Opção 1 – substitui pelo **sucessor**



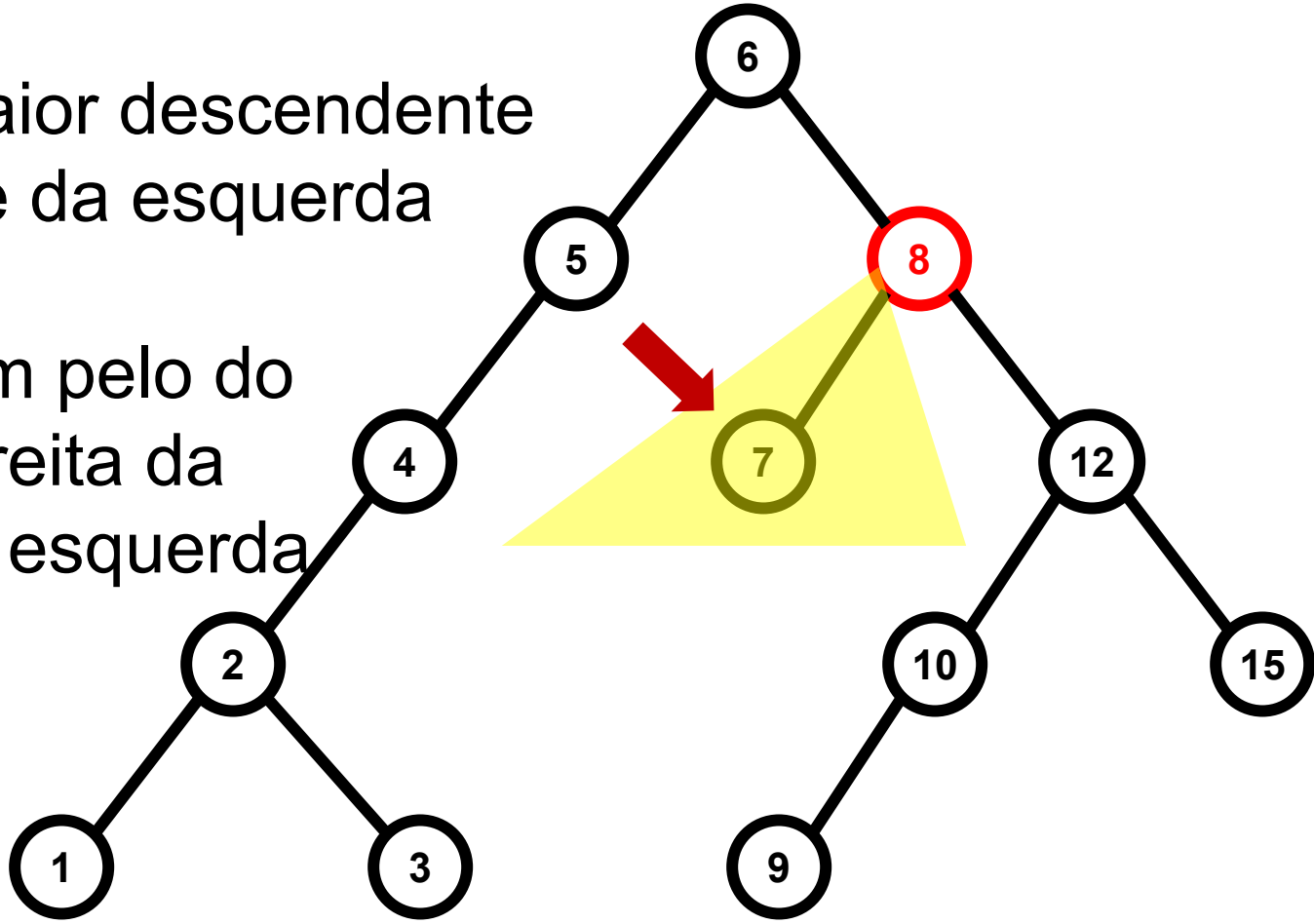
# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Opção 2 – maior descendente da sub-árvore da esquerda

Substitui o item pelo do filho mais à direita da sub-árvore da esquerda *(antecessor);*

remove este



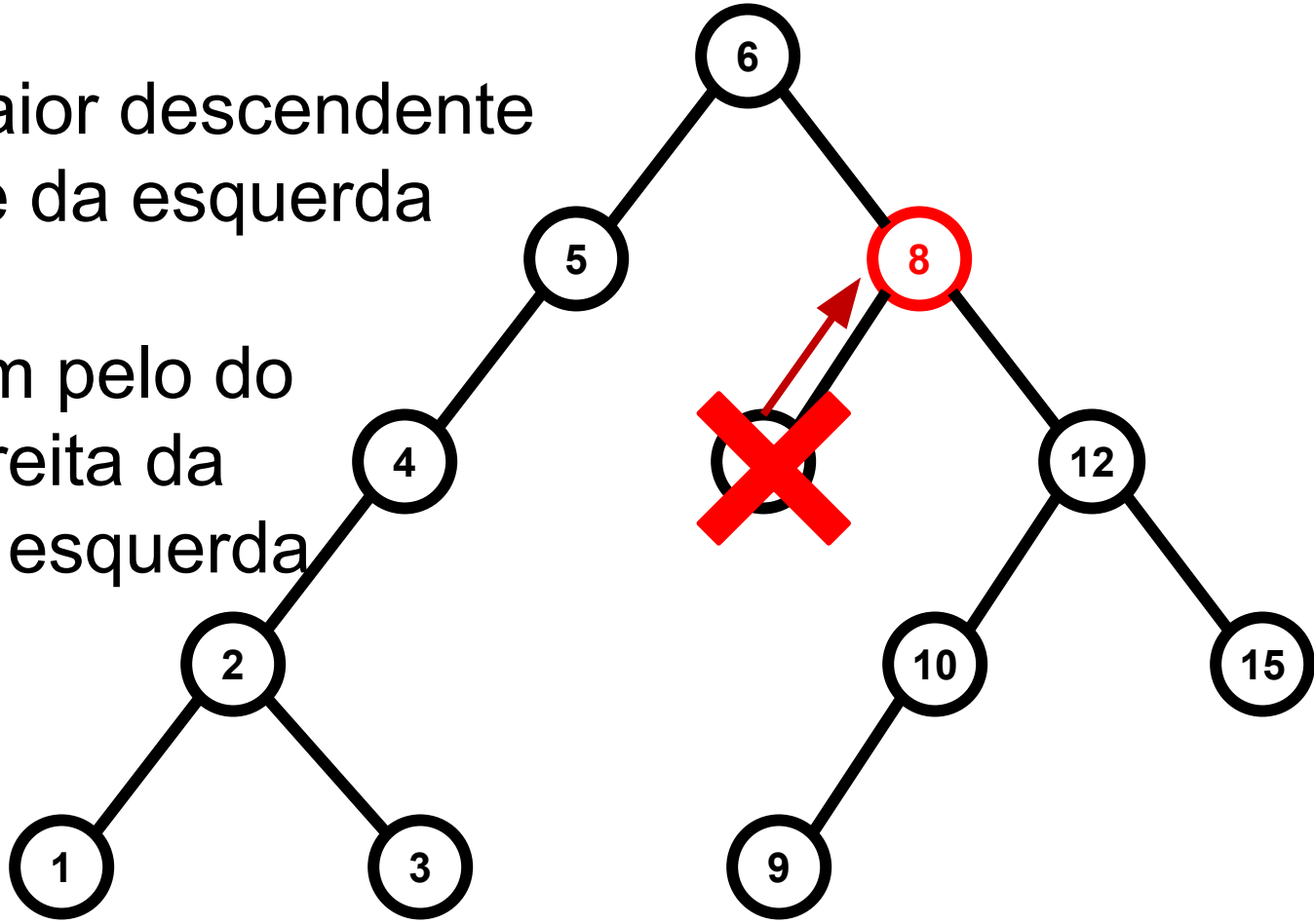
# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Opção 2 – maior descendente da sub-árvore da esquerda

Substitui o item pelo do filho mais à direita da sub-árvore da esquerda *(antecessor);*

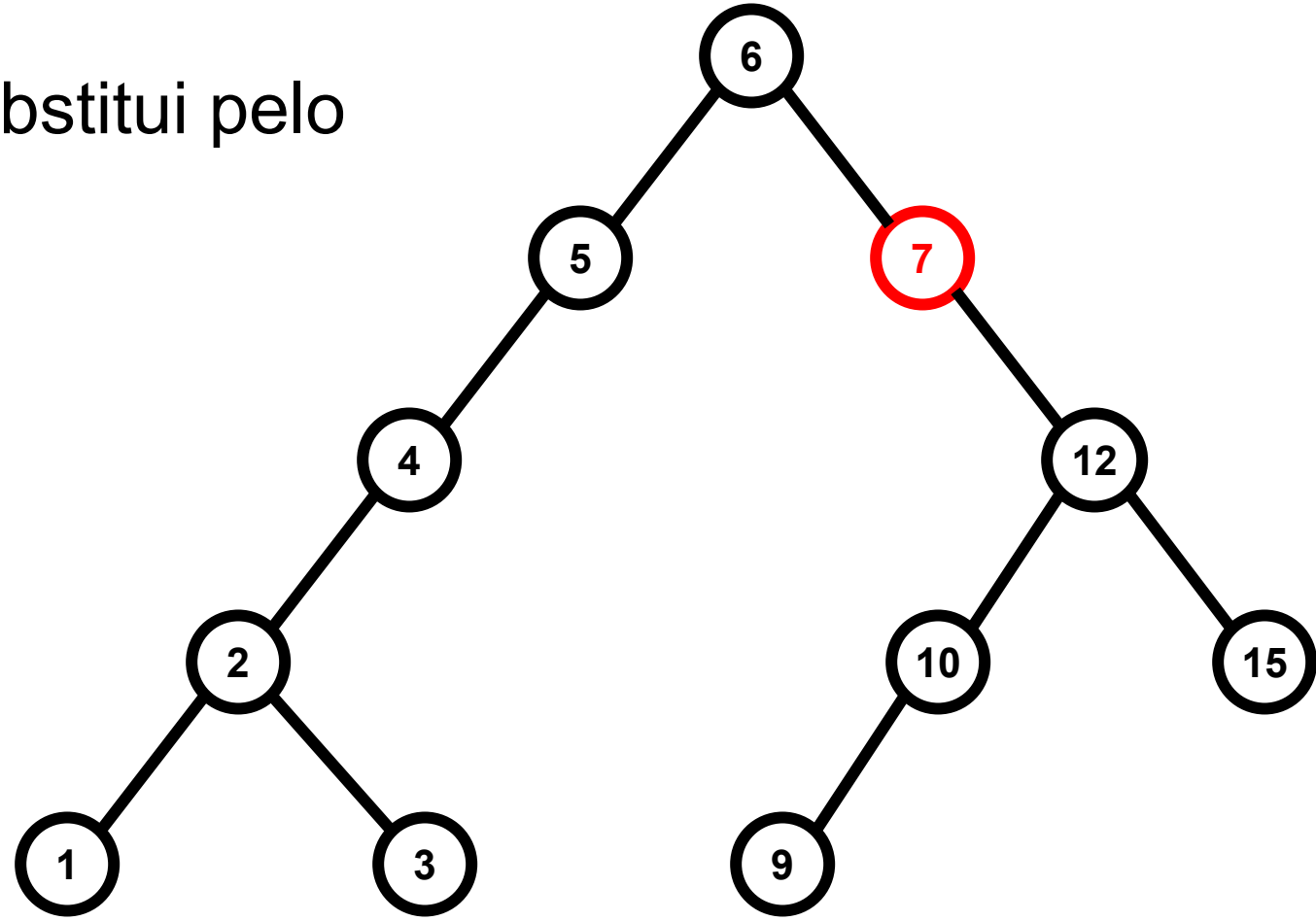
remove este



# Remoção: Nó tem 2 filhos

- Remoção da chave 8

Opção 2 – substitui pelo **antecessor**



# Árvore Binária de Pesquisa: Remoção

```
void TabelaArvoreBinaria::RemoveRecursivo(TipoNo* &no, TipoChave chave){
  TipoNo *aux;

  if (no == NULL) {
     throw("Item nao está presente");
  }

  if (chave < no->item.GetChave())
       return RemoveRecursivo(no->esq, chave);
  else if (chave>no->item.GetChave())
       return RemoveRecursivo(no->dir, chave);
  else {
     if (no->dir == NULL) {
        aux = no;
        no = no->esq;
        free(aux);
     }
     else if(no->esq == NULL) {
        aux = no;
        no = no->dir;
        free(aux);
     }
     else
        Antecessor(no, no->esq);
  }
}
```

# Árvore Binária de Pesquisa: Remoção

```
void TabelaArvoreBinaria::RemoveRecursivo(TipoNo* &no, TipoChave chave){
  TipoNo *aux;
  if (no == NULL) {
     throw("Item nao está presente");
  }
  if (chave < no->item.GetChave())
       return RemoveRecursivo(no->esq, chave);
  else if (chave>no->item.GetChave())
       return RemoveRecursivo(no->dir, chave);
  else {
     if (no->dir == NULL) {
        aux = no;
        no = no->esq;
        free(aux);
     }
     else if(no->esq == NULL) {
        aux = no;
        no = no->dir;
        free(aux);
     }
     else
        Antecessor(no, no->esq);
  }
}
```

**O nó a ser removido não está na árvore**

**Chave procurada menor que a chave do nó → procura na sub-árvore da esquerda**

**Chave procurada maior que a chave do nó → procura na sub-árvore da direita**

# Árvore Binária de Pesquisa: Remoção

```
void TabelaArvoreBinaria::RemoveRecursivo(TipoNo* &no, TipoChave chave){
  TipoNo *aux;

  if (no == NULL) {
     throw("Item nao está presente");
  }

  if (chave < no->item.GetChave())
       return RemoveRecursivo(no->esq, chave);
  else if (chave>no->item.GetChave())
       return RemoveRecursivo(no->dir, chave);
  else {
     if (no->dir == NULL) {
        aux = no;
        no = no->esq;
        free(aux);
     }
     else if(no->esq == NULL) {
        aux = no;
        no = no->dir;
        free(aux);
     }
     else
        Antecessor(no, no->esq);
  }
}
```

**Chave procurada não é menor e nem maior que a chave do nó → logo ACHOU a chave**

**Nó tem 1 ou 0 filhos**

**Se o filho da direita é nulo, então o nó passa a ser o seu filho da esquerda**

**Se o filho da esquerda é nulo, então o nó passa a ser o seu filho da direita**

**Nó tem 2 filhos; substitui nó pelo seu Antecessor**

# Árvore Binária de Pesquisa: Remoção

Chamada no método RemoveRecursivo: Antecessor(no, no->esq);

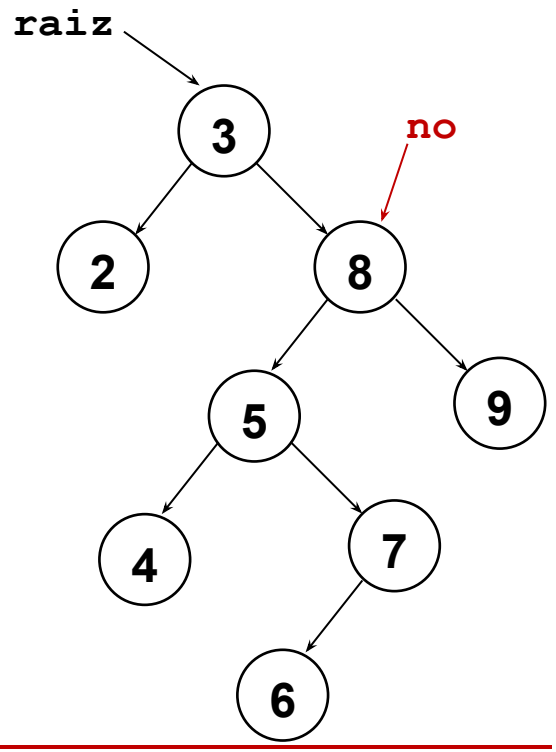
```
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

# Exemplo: Remoção Chave 8

**Retirar 8**

Chamada no método RemoveRecursivo: `Antecessor (no, no->esq);`

```
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

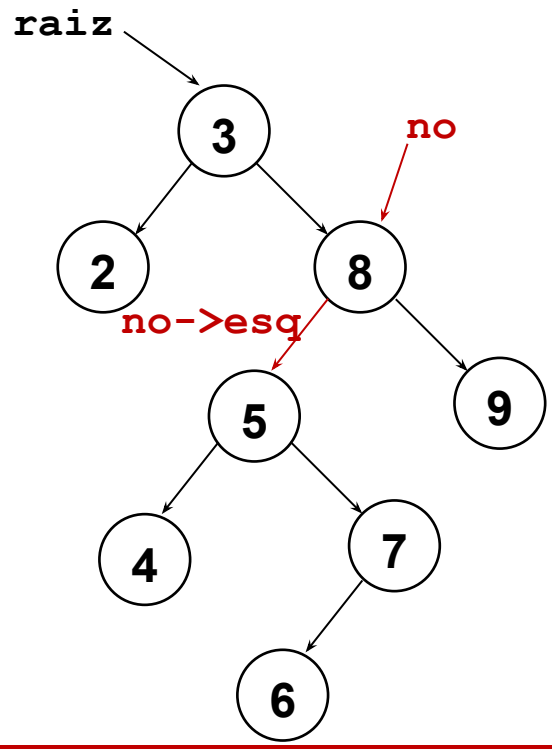


# Exemplo: Remoção Chave 8

**Retirar 8**

Chamada no método RemoveRecursivo: Antecessor (no, no->esq);

```
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```



# Exemplo: Remoção Chave 8

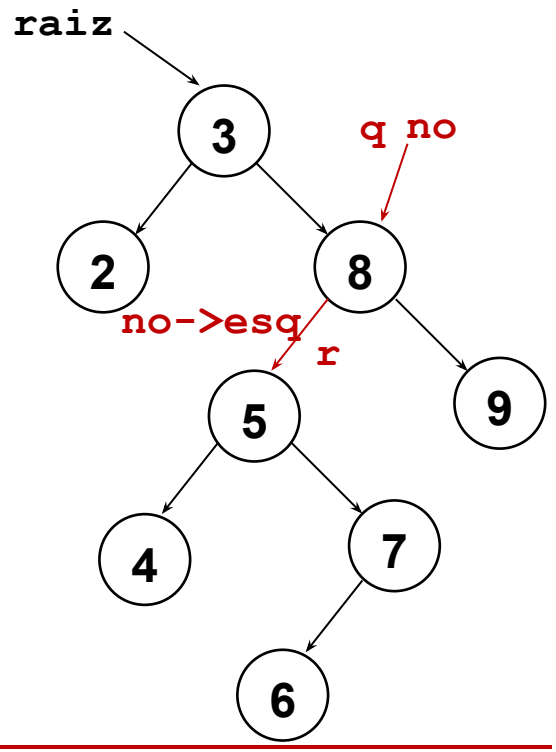
**Retirar 8**

Chamada no método RemoveRecursivo: Antecessor (no, no->esq);

a retirar

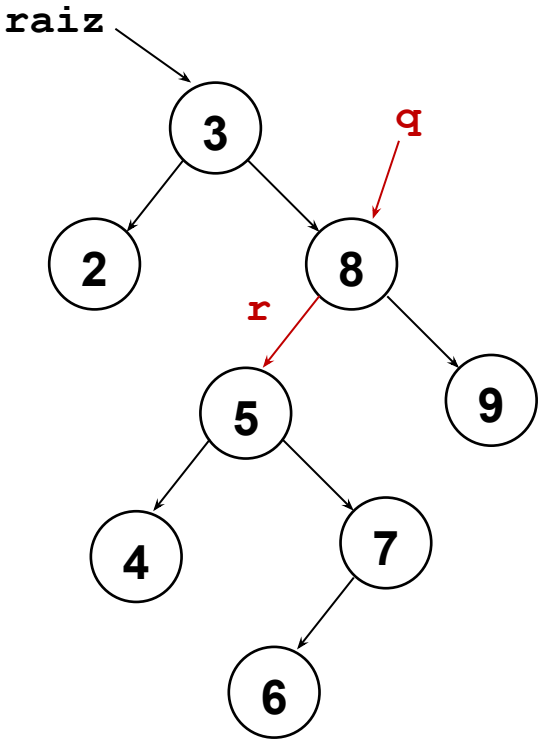
antecessor

```
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

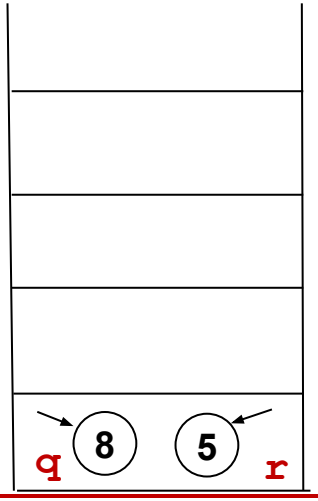


# Exemplo: Remoção Chave 8

```
                                          a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

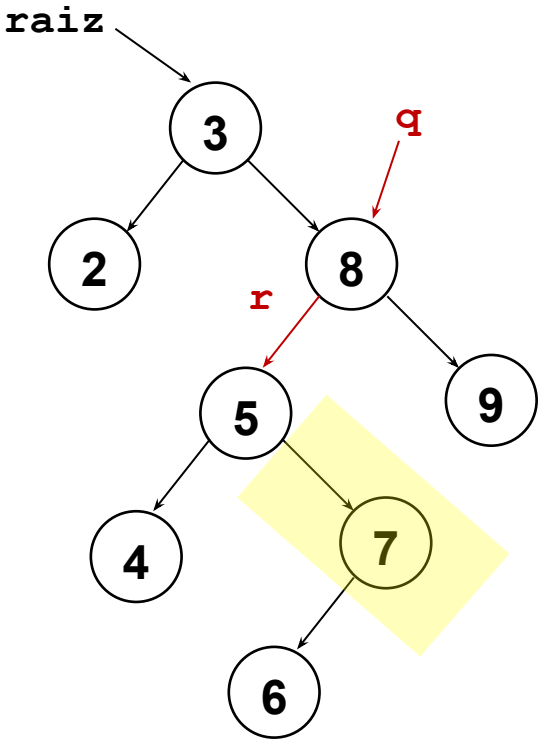



**Pilha de Execução**

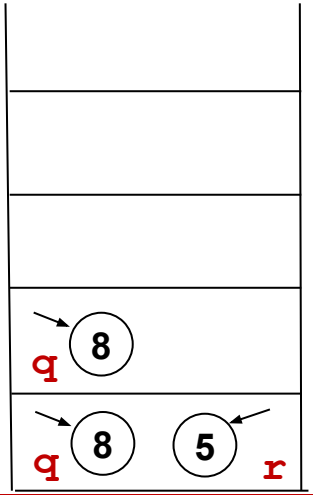


# Exemplo: Remoção Chave 8

```
                                                a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

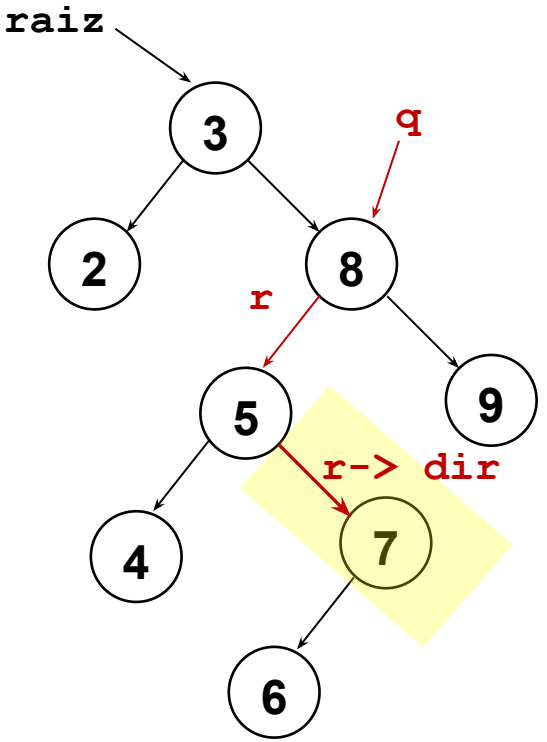



**Pilha de Execução**

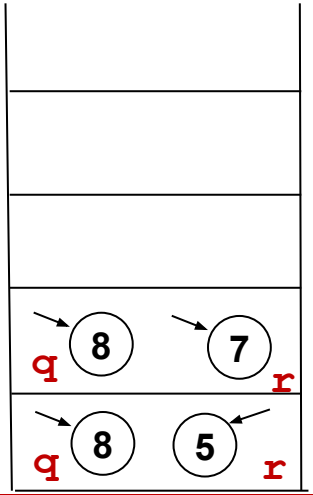


# Exemplo: Remoção Chave 8

```
                                          a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

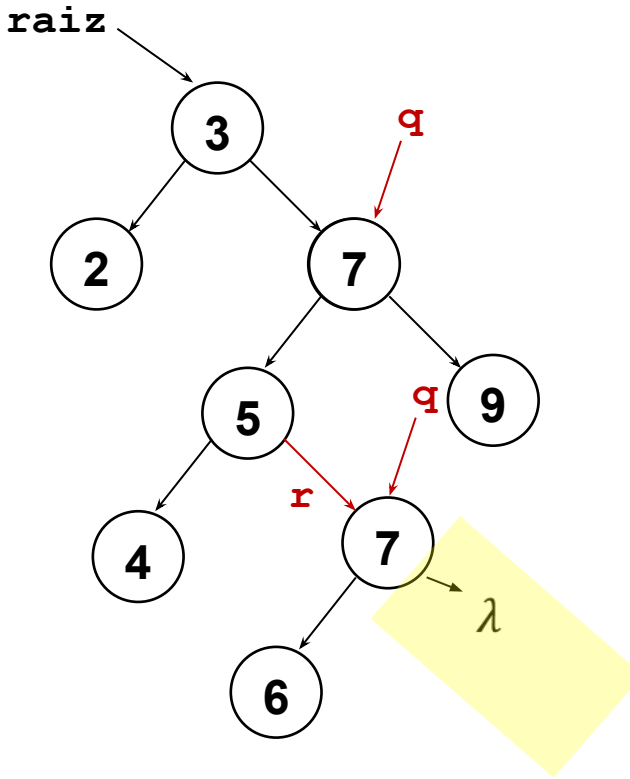



**Pilha de Execução**

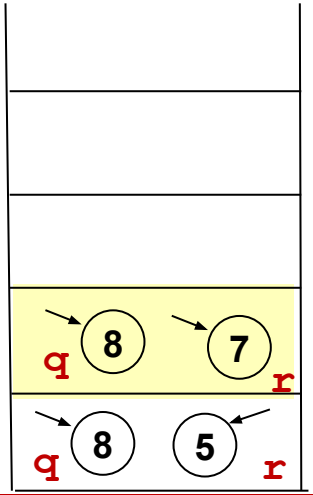


# Exemplo: Remoção Chave 8

```
                                                a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

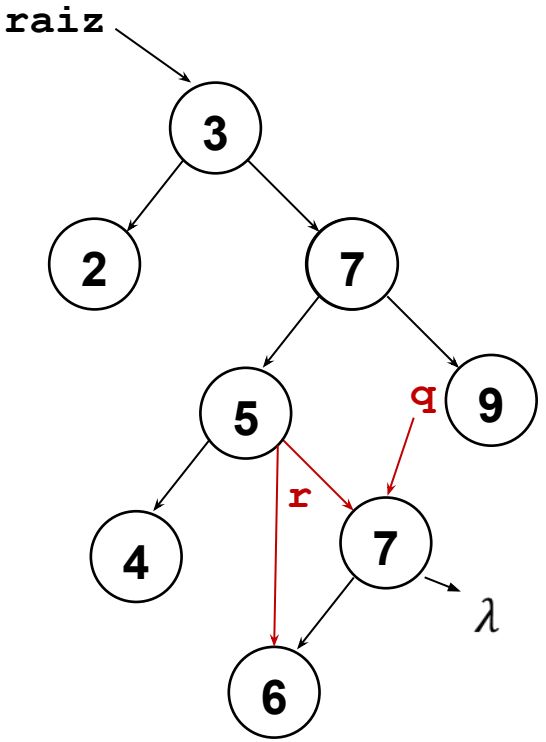



**Pilha de Execução**

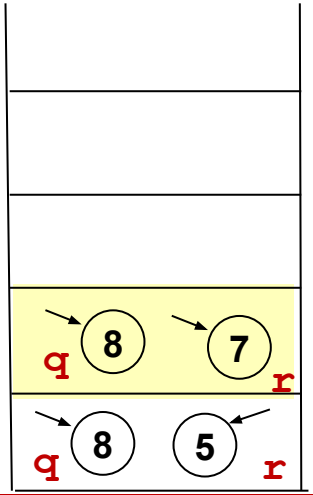


# Exemplo: Remoção Chave 8

```
                                                a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
     if(r->dir != NULL) {
          Antecessor(q, r->dir);
          return;
     }

     q->item = r->item;
     q = r;
     r = r->esq;
     free(q);
}
```

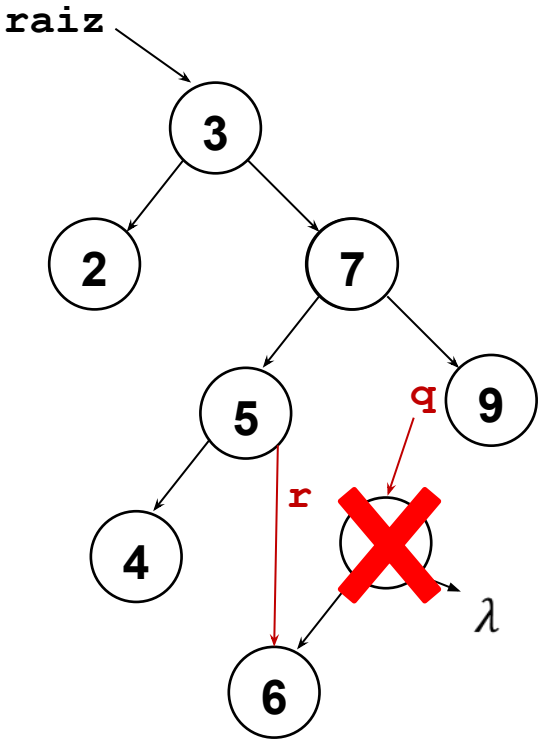



**Pilha de Execução**

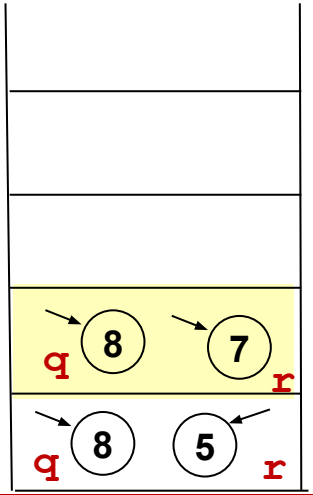


# Exemplo: Remoção Chave 8

```
                                                a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
     if(r->dir != NULL) {
          Antecessor(q, r->dir);
          return;
     }

     q->item = r->item;
     q = r;
     r = r->esq;
     free(q);
}
```

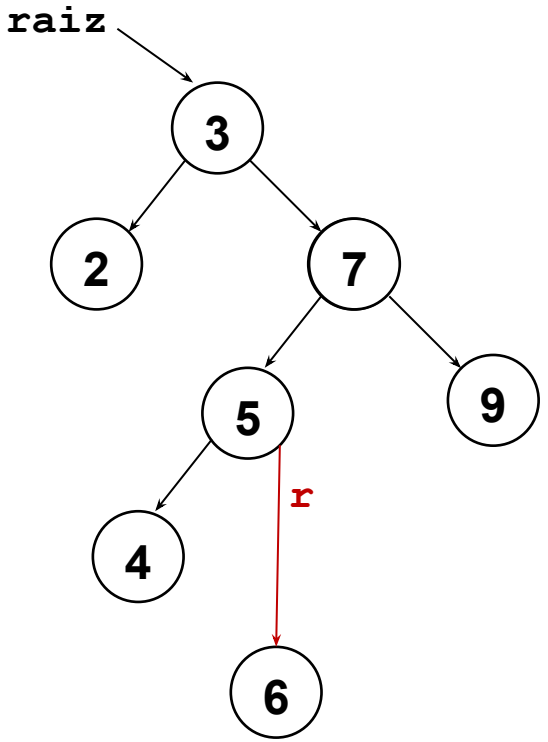



**Pilha de Execução**

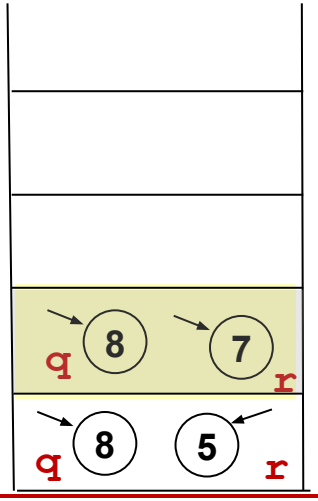


# Exemplo: Remoção Chave 8

```
                                          a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```

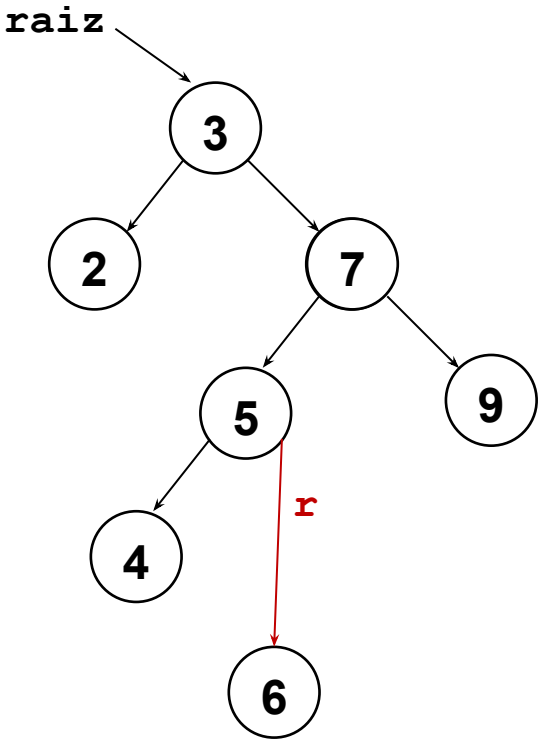



**Pilha de Execução**

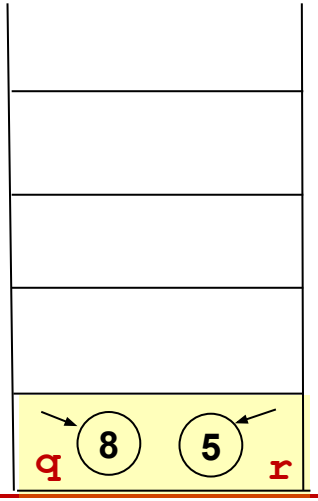


# Exemplo: Remoção Chave 8

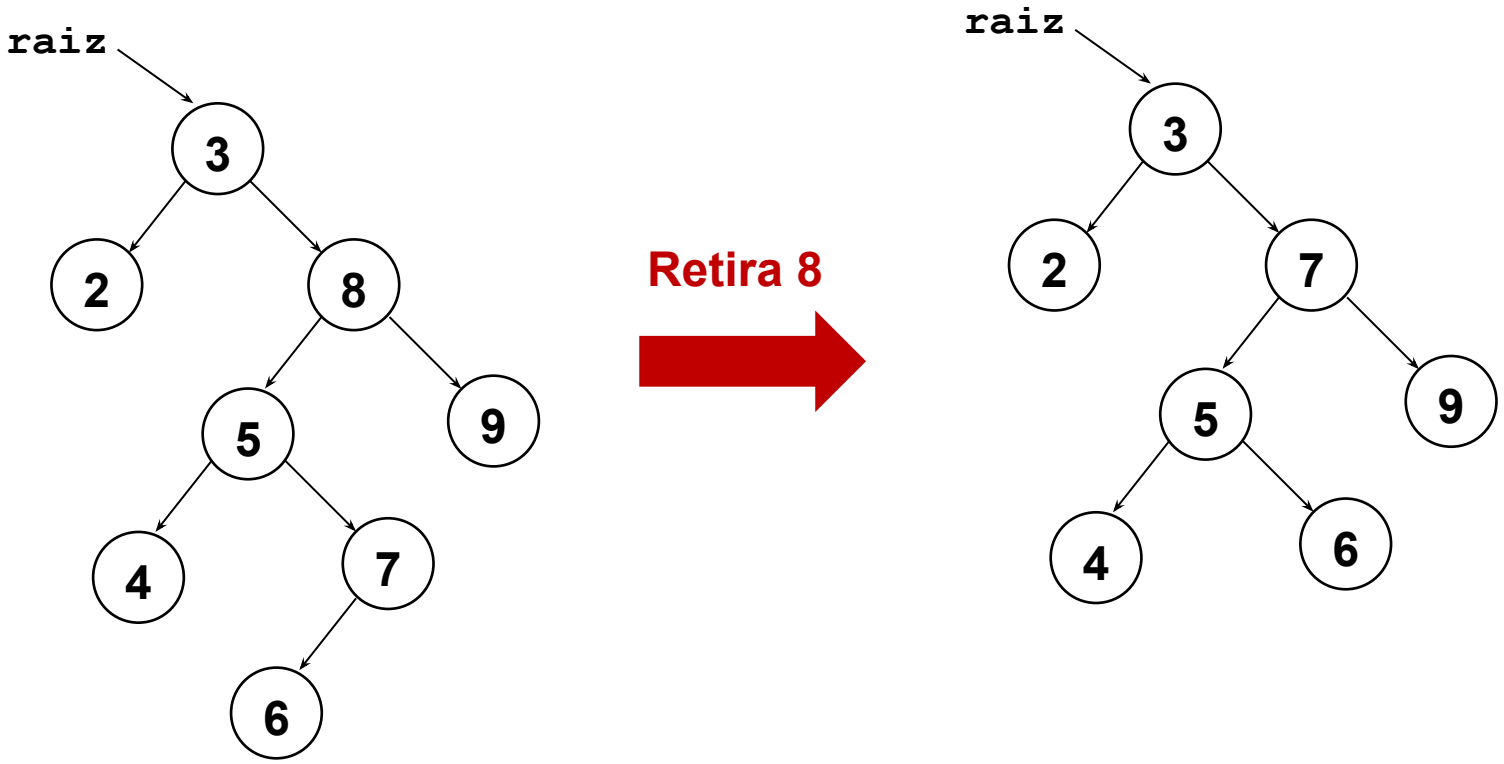
```
                                          a retirar   antecessor
void TabelaArvoreBinaria::Antecessor(TipoNo *q, TipoNo* &r){
    if(r->dir != NULL) {
        Antecessor(q, r->dir);
        return;
    }

    q->item = r->item;
    q = r;
    r = r->esq;
    free(q);
}
```




**Pilha de Execução**



# Exemplo: Remoção Chave 8



# Árvore Binária de Pesquisa: Análise

- Complexidade de tempo?
  - Caso a árvore esteja balanceada: *O(log(n))*
  - Para uma árvore qualquer: *O(n)*

# Tabelas Hash

# Hashing

- Algoritmos vistos efetuam comparações para localizar uma chave.
- Hashing usar outra estratégia: transformação aritmética sobre a chave de pesquisa
  - Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.
  - Busca por meio de operações aritméticas que transformam a chave em endereços em uma tabela.

# Hashing

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:
  1. Computar o valor da **função de transformação**, a qual transforma a chave de pesquisa em um endereço da tabela.
  2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para **lidar com colisões**.

# Função de Transformação

- Uma função de transformação deve mapear chaves em inteiros dentro do intervalo [0...M − 1], onde M é o tamanho da tabela.

- A função de transformação ideal é aquela que:
  - Seja simples de ser computada.
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

# Hashing

- Exemplo:
  - Usa o resto da divisão por M (onde k é a chave)

    F(k) = k % M

  - F(x) = x % 10;

  - F(1) = 1 % 10 = 1
  - F(22) = 22 % 10 = 2
  - F(43) = 43 % 10 = 3
  - F(57) = 57 % 10 = 7
  - F(125) = 125 % 10 = 5

# Hashing – Chaves não numéricas

- Chave[i] corresponde à representação ASCII do i-ésimo caractere da chave.
- Exemplo: Considerando a i-ésima letra do alfabeto representada por *i* e a função de transformação

  h(Chave) = Chave mod M (M=10).

  A = 1 $\longrightarrow$ 1

  L = 12 $\longrightarrow$ 2

  G = 7 $\longrightarrow$ 7

  O = 15 $\longrightarrow$ 5

# Chaves não numéricas

- Se a chave for um string, soma-se o valor dos seus caracteres

  A = 1 $\rightarrow$ 1

  L = 12 $\rightarrow$ 2

  G = 7 $\rightarrow$ 7

  O = 15 $\rightarrow$ 5

  ALGO = 1+12+7+15 = 35 $\rightarrow$ 5

- Como tratar anagramas:

  Exemplo: ALGO, GALO, GOLA e LAGO

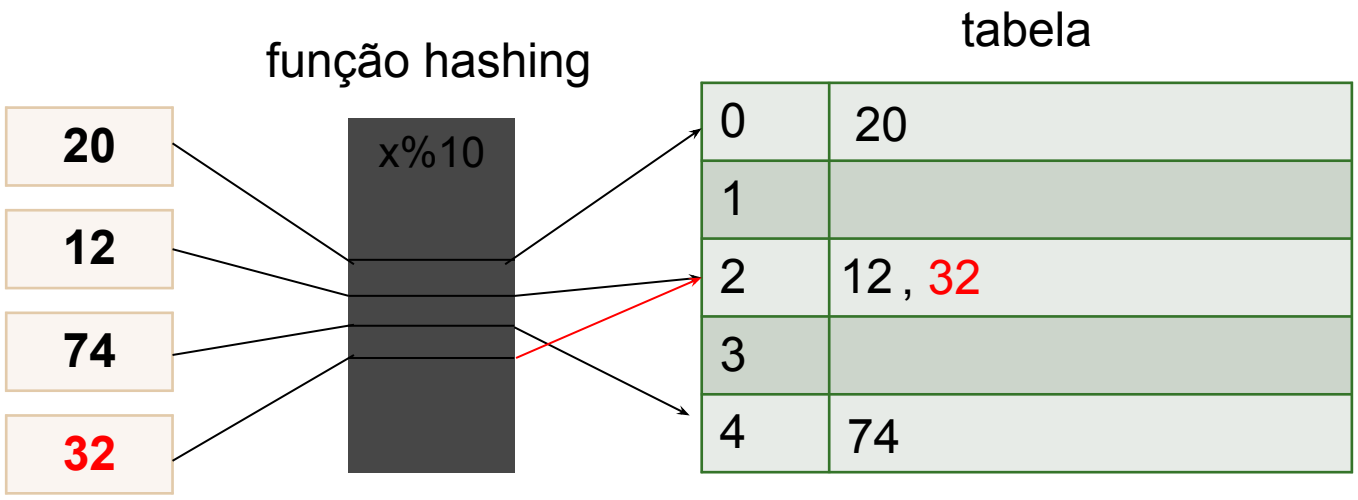
- Pode-se atribuir um peso, *p[i]* à posição da letra na palavra, $1 \leq i \leq n$.

$$H = \sum_{i=1}^{n} Chave[i] \times p[i]$$

# Hashing

- Principal Problema: colisões
  - Exemplo:

    Função hashing: f(x) = x % 10



# Hashing: colisões

- Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm de ser resolvidas de alguma forma.
- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões.

# Hashing: colisões

- Hashing – Paradoxo do Aniversário
  - Chave é a data de nascimento
  - Tabela possui 365 entradas
  - A probabilidade p de se inserir 2 itens sem colisão em uma tabela de tamanho 365 considerando um conjunto com mais de 23 itens é:
    - p ~ 50.7%

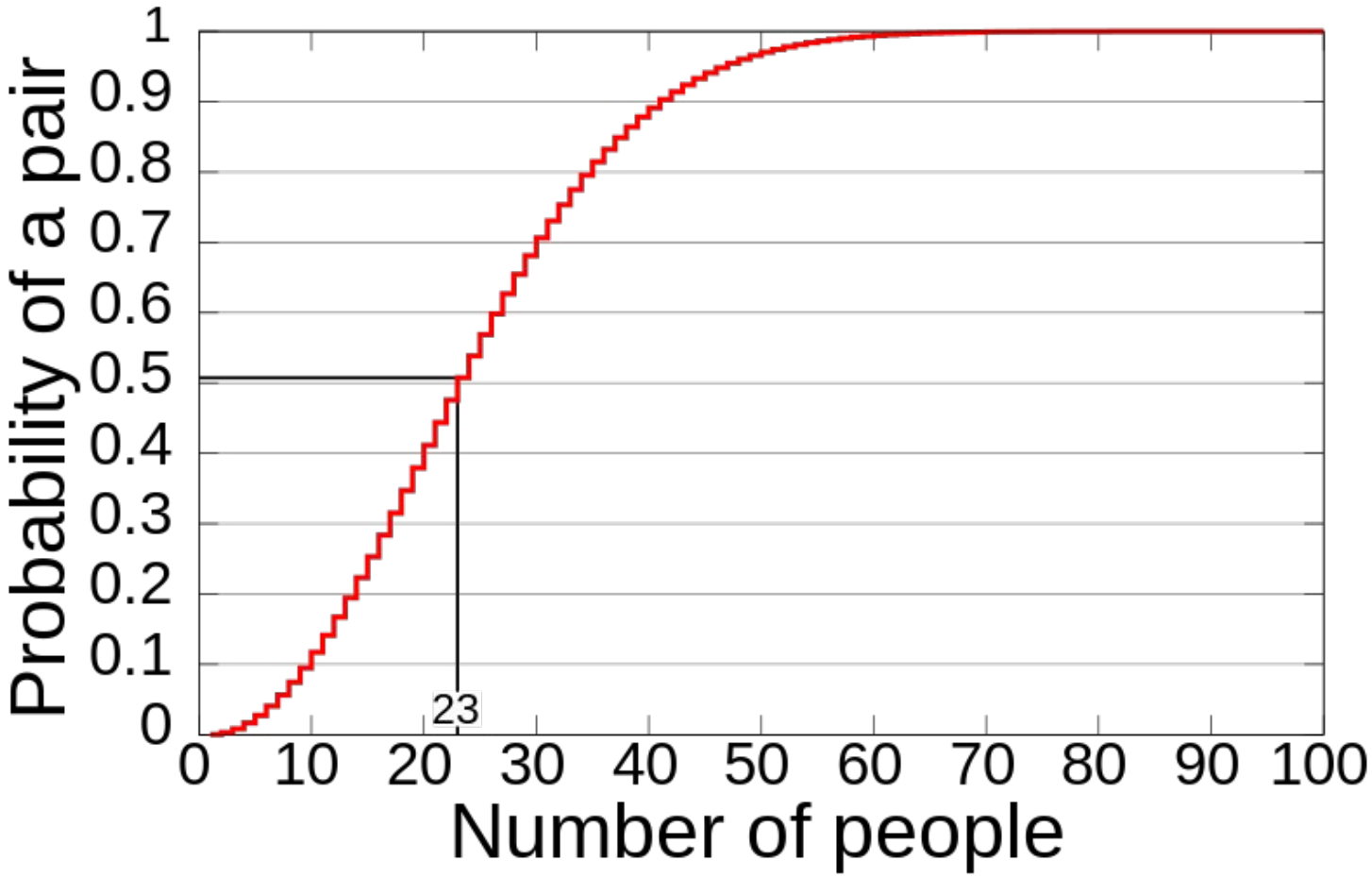
# Hashing: colisões

- O paradoxo do aniversário (Feller,1968, p. 33), diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que o aniversário de duas pessoas seja no mesmo dia.

| n | P(n) |
|---|---|
| 10 | 12% |
| 20 | 41% |
| 23 | 50.7% |
| 30 | 70% |
| 50 | 97% |

# Hashing: colisões

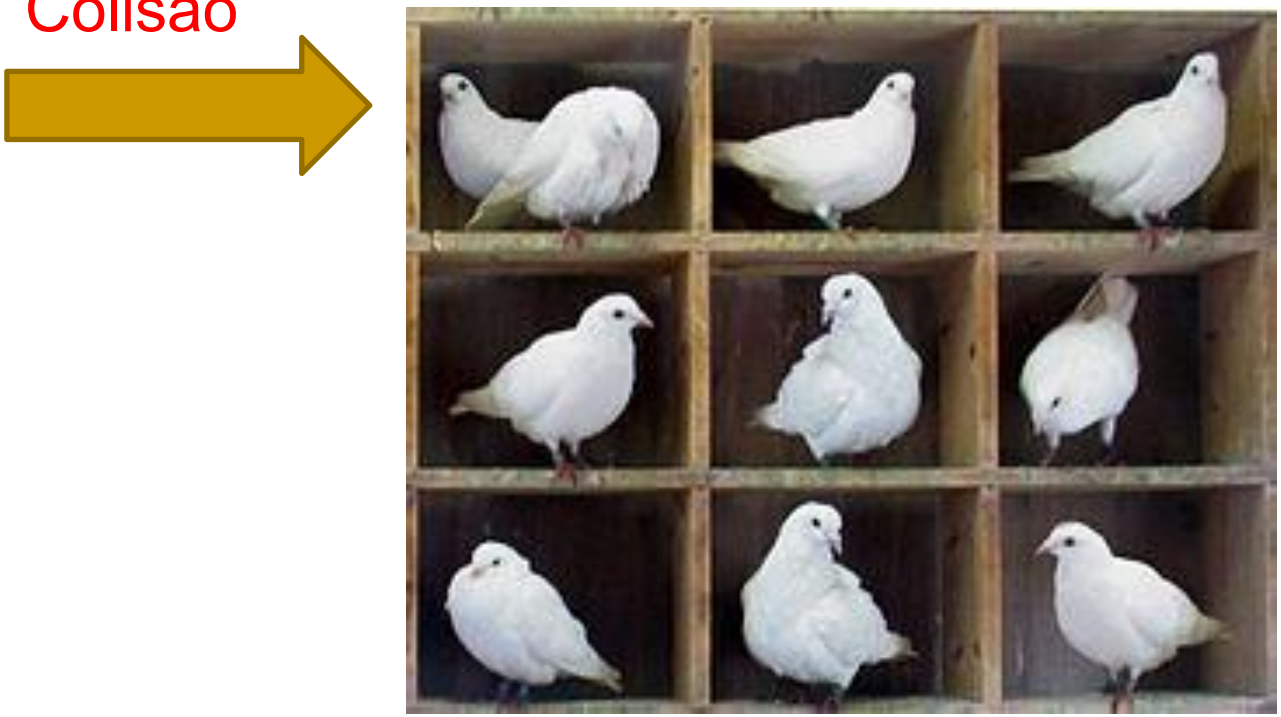
- Paradoxo do aniversário



# Hashing: Colisões

- Seja um conjunto com n pessoas e n > 365, então a probabilidade de colisão é p = 1.0, pelo princípio das casas dos pombos.

Colisão



# Hashing: colisões

- A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:

$$p = \frac{M-1}{M} \times \frac{M-2}{M} \times \ldots \times \frac{M-N+1}{M} = \prod_{i=1}^{N} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$$

# Hashing: colisões

Seja uma  tabela com 50.063.860 de posições (M):

| Chaves (N) | Chance de colisão | Fator de carga (N/M) |
|---|---|---|
| 1000 | 0.995% | 0.002% |
| 2000 | 3.918% | 0.004% |
| 4000 | 14.772% | 0.008% |
| 6000 | 30.206% | 0.012% |
| 8000 | 47.234% | 0.016% |
| 10000 | 63.171% | 0.020% |
| 12000 | 76.269% | 0.024% |
| 14000 | 85.883% | 0.028% |
| 16000 | 92.248% | 0.032% |
| 18000 | 96.070% | 0.036% |
| 20000 | 98.160% | 0.040% |
| 22000 | 99.205% | 0.044% |

# Resolução de Colisões

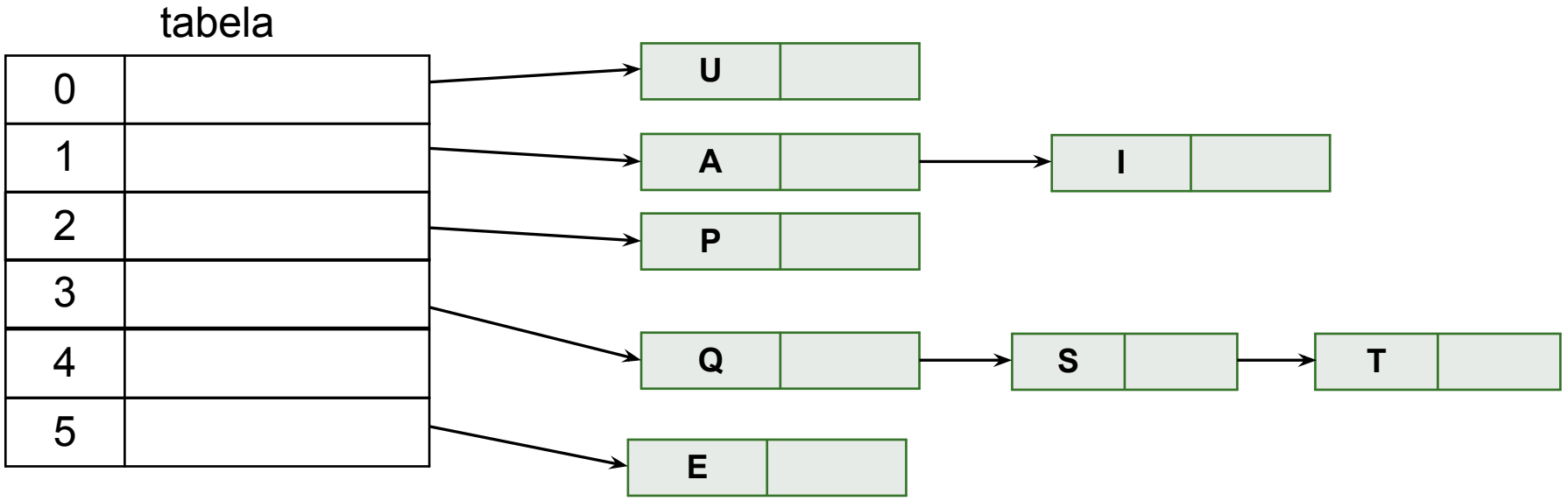
- Encadeamento (Listas Encadeadas)

- Endereçamento aberto

# RESOLUÇÃO DE COLISÕES – LISTA ENCADEADA

# Resolução de Colisões - Encadeamento

- Cria uma lista encadeada para cada endereço da tabela.
- Todas as chaves com mesmo endereço na tabela são encadeadas em uma lista linear.



# Resolução de Colisões - Encadeamento

- Exemplo: Considerando a i-ésima letra do alfabeto representada por i e a função de transformação h(Chave) = Chave mod M (M=10). Insira EXEMPLO na tabela hashing.

  E = 5

  X = 25

  M = 13

  P = 16

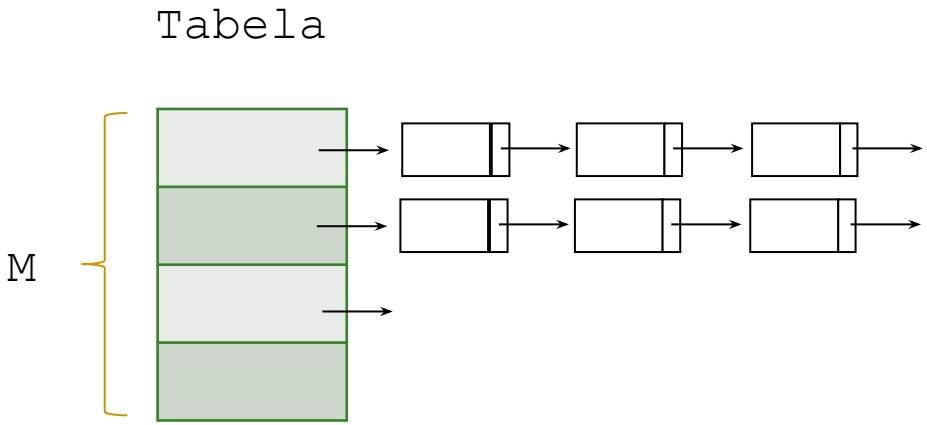
  L = 12

  O = 15

# Resolução de Colisões - Encadeamento

```
class Hash_LE
{
    public:
        Hash_LE();
        TipoItem Pesquisa(TipoChave chave);
        void Insere(TipoItem item);
        void Remove(TipoChave chave);

    private:
        static const int M = 7;
        int Hash(TipoChave Chave);
        ListaEncadeada Tabela[M];
};
```

Observação:
O construtor default já inicializa o vetor com as listas vazias



# Resolução de Colisões - Encadeamento

```
class ListaEncadeada : public Lista {
    public:
        ListaEncadeada();
        ~ListaEncadeada();

        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
***     TipoItem RemoveItem(TipoChave c);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();
        void Limpa();
    private:
        TipoCelula* primeiro;
        TipoCelula* ultimo;
        TipoCelula* Posiciona(int pos, bool antes);
};
```

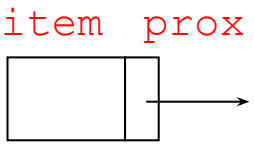
***** Método adicionado para a remoção de um item específico**

# Recapitulando a Classe Lista Encadeada

```
class TipoItem
{
    public:
        TipoItem();
        TipoItem(TipoChave c);
        void SetChave(TipoChave c);
        TipoChave GetChave();
        void Imprime();
***     bool Vazio();

    private:
        TipoChave chave;
        // outros membros
};
```

***** Método adcionado para testar se um item é vazio (chave == -1)**

```
class TipoCelula
{
    public:
        TipoCelula();

    private:
        TipoItem item;
        TipoCelula *prox;

    friend class ListaEncadeada;
};

TipoCelula::TipoCelula()
{
    item.SetChave(-1);
    prox = NULL;
}
```



# Hash Listas Encadeadas - Pesquisa

```
TipoItem Hash_LE::Pesquisa(TipoChave chave){
     int pos;
     TipoItem item;

     pos = Hash(chave);
     item = Tabela[pos].Pesquisa(chave);
     return item;
}
```

Aplica a função hash na chave, para saber em qual lista procurar

Chama o método de pesquisa da classe ListaEncadeada

Retorna o item encontrado ou um item vazio (chave == -1) se não encontrar

# Hash Listas Encadeadas - Pesquisa

```
// Retorna o item encontrado ou um item vazio
// (chave == -1) se não estiver presente

TipoItem  ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux; // construtor seta o item para -1;
    TipoCelula *p;

    p = primeiro->prox;
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }
    return aux;
};
```

Inicializa p com o primeiro da lista

Enquanto p não chegar no fim da lista

Se a chave do item da lista é a procurada

aux recebe o elemento da lista, termina o loop

Se não, passa para o próximo elemento da lista

Retorna aux

# Hash Listas Encadeadas - Insere

```
void Hash_LE::Insere(TipoItem item){
    TipoItem aux;
    int pos;

    aux = Pesquisa(item.GetChave());
    if(!aux.Vazio())
        throw("Erro: Item já está presente");

    pos = Hash(item.GetChave());
    Tabela[pos].InsereFinal(item);
}
```

Verifica se o item já está presente

Aplica a função hash na chave para indicar em qual lista Inserir

Insere no final da lista, chamando o InsereFinal da classe ListaEncadeada

# Hash Listas Encadeadas - Insere

```
void ListaEncadeada::InsereFinal(TipoItem item) {
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = NULL;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```

# Hash Listas Encadeadas - Remove

- Aplica a função hash na chave para indicar em qual lista o item vai estar
- Chama o RemoveItem da classe ListaEncadeada
  - O RemoveItem da classe ListaEncadeada gera uma exceção se o item não estiver presente

```
void Hash_LE::Remove(TipoChave chave){
    int pos;

    pos = Hash(chave);
    Tabela[pos].RemoveItem(chave);
}
```

# Hash Listas Encadeadas - Remove

```
TipoItem  ListaEncadeada::RemoveItem(TipoChave c) {
    TipoItem aux;
    TipoCelula *p, *q;
    // Posiociona p na célula anterior ao item procurado
    p = primeiro;
    while ((p->prox!=NULL) && (p->prox->item.GetChave() != c))
        p = p->prox;

    // remove a célula contendo o item, retornando-o
    if(p->prox == NULL)
        throw "Erro: item não está presente";
    else {
        q = p->prox;
        p->prox = q->prox;
        aux = q->item;
        delete q;
        tamanho--;
        if(p->prox == NULL) ultimo = p;
    }
    return aux;
};
```

Procura o elemento a ser retirado e p aponta para o anterior

Se o elemento não está na lista, lança uma exceção

q é o elemento a ser retirado

retira q da lista

aux recebe o elemento sendo removido

chama o destrutor para q

atualiza os campos da lista: tamanho e último

# Encadeamento: Análise

- Tamanho esperado de cada lista: N/M
  - Assumindo que qualquer item do conjunto tem igual probabilidade endereçado para qualquer entrada de T.
  - N: número de registros, M: tamanho da tabela

- Operações Pesquisa, InsereHashing e RetiraHashing: *O(N/M)*
  - Tempo para encontrar a entrada na tabela: *O(1)*
  - Tempo para percorrer a lista: *O(N/M)*

- M ~ N, tempo se torna constante.

# RESOLUÇÃO DE COLISÕES – ENDEREÇAMENTO ABERTO

# Resolução de Colisões: Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, não há necessidade de se utilizar apontadores.

- **Endereçamento aberto**: chaves são armazenadas na própria tabela.

- Para tabela com tamanho M (M > N), pode-se utilizar os espaços vazios da própria tabela para resolver as colisões.

# Resolução de Colisões: Endereçamento Aberto

- Quando encontra uma colisão, procura localizações alternativas ($h_j$).

- **Hashing linear**

$$h_j = (h(x) + j) \bmod M, \quad \text{para } 1 \leq j \leq M-1$$

- **Hashing quadrático**

$$h_j = (h(x) + j^2) \bmod M$$

# Endereçamento Aberto: Exemplo

- Suponha que  a i-ésima letra do alfabeto é representada pelo número i e a função de transformação abaixo é utilizada:
  - h(Chave) = Chave mod M

- O resultado da inserção das chaves L U N E S na tabela, usando hashing linear (j = 1) para resolver colisões é mostrado a seguir.
  - Considere M = 7

# Endereçamento Aberto: Exemplo

```
h(L) = h(12) = 5
h(U) = h(21) = 0
h(N) = h(14) = 0
h(E) = h(5)  = 5
h(S) = h(19) = 5
```

| | T |
|---|---|
| 0 | U |
| 1 | N |
| 2 | S |
| 3 | |
| 4 | |
| 5 | L |
| 6 | E |

# Endereçamento Aberto

- Pesquisa

| | T |
|---|---|
| 0 | U |
| 1 | N |
| 2 | S |
| 3 | |
| 4 | |
| 5 | L |
| 6 | E |

Procura **S**

**h(S) = 19 = 5**

**OK!**

Retira **N**

Procura **S** **???**

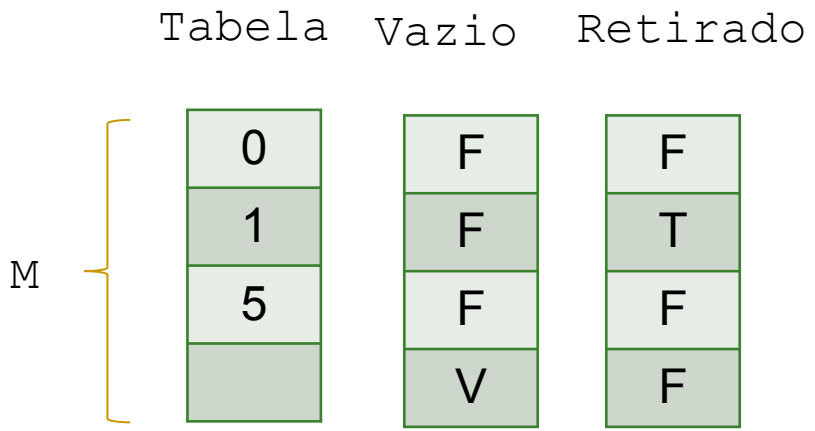
# Endereçamento Aberto

- O que fazer quando um elemento é retirado?
  - Possíveis problemas na busca sequencial após a colisão…
- Solução: Diferenciar o status da posição: "**Vazia**" x "**Retirada**"
- Posição com status de "**Retirada**"
  - Para a pesquisa, posição é considerada ocupada
  - Para a inserção, posição é considerada vazia

# Endereçamento Aberto: Implementação

```
class Hash_EA
{
  public:
    Hash_EA();

    TipoItem Pesquisa(TipoChave chave);
    void Insere(TipoItem item);
    void Remove(TipoChave chave);

  private:
    static const int M = 7;
    int Hash(TipoChave Chave);
    TipoItem Tabela[M];
    bool vazio[M];
    bool retirado[M];
};
```

Vetores de Flags para indicar posições vazias e retiradas



# Endereçamento Aberto: Implementação

```
TipoItem Hash_EA::Pesquisa(TipoChave Chave) {
    TipoItem aux; // construtor seta o item para -1;
    int pos, i;

    pos = Hash(Chave);
    i = 0;
    while ( (i<M) && !vazio[(pos+i)%M] &&
            (Tabela[(pos+i)%M].GetChave() != Chave)  )
        i++;

    if ( (Tabela[(pos+i)%M].GetChave() == Chave) &&
         !retirado[(pos+i)%M] )  {
        aux = Tabela[(pos+i)%M];
    }

    return aux;
}
```

Aplica a função hash na chave, para saber a sua posição na tabela

Enquanto não percorreu toda a tabela e nem achou uma posição vazia

E enquanto não achou a chave

Se achou a chave e não tem o status de retirada

Aux retorna o item procurado

Aux retorna o item

# Endereçamento Aberto: Implementação

```
void Hash_EA::Insere(TipoItem item){
    TipoItem aux; // construtor seta o item para -1;
    int pos, i;

    aux = Pesquisa(item.GetChave());
    if(!aux.Vazio())
        throw("Erro: Item já está presente");

    pos = Hash(item.GetChave());
    i = 0;
    while ( (i<M) && !vazio[(pos+i)%M] && !retirado[(pos+i)%M] )
           i++;

    if(i==M)
        throw("Erro: Tabela está cheia");
    else {
        Tabela[(pos+i)%M] = item;
        vazio[(pos+i)%M] = false;
        retirado[(pos+i)%M] = false;
    }
}
```

Se o elemento já está na tabela, lança uma exceção

Aplica a função hash na chave, para saber a sua posição na tabela

Procura posição disponível para inserção

Se tabela cheia, lança uma exceção

Insere item na Tabela, na posição livre

Indica que a posição está ocupada

# Endereçamento Aberto: Implementação

```
void Hash_EA::Remove(TipoChave Chave){
    int pos, i;

    pos = Hash(Chave);
    i = 0;
    while ( (i<M) && !vazio[(pos+i)%M] &&
            (Tabela[(pos+i)%M].GetChave() != Chave) )
        i++;

    if ( (Tabela[(pos+i)%M].GetChave() == Chave) &&
         !retirado[(pos+i)%M] )
        retirado[(pos+i)%M] = true;
    else
        throw("Erro: Item não está presente");
}
```

Procura posição do elemento na tabela

Se achou o elemento e ele não tem o status de retirado, retira-o.

Se o elemento não está na tabela, lança uma exceção

# Endereçamento Aberto: Análise

- Tempo para Pesquisa, Inserção e Retirada
  - Melhor caso: O(1)
  - Pior caso: O (n)

- Seja α = N / M o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa **com sucesso** é

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

*C(n) é o número de comparações*

# Endereçamento Aberto: Análise

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

| α = N / M | C(n) |
|---|---|
| 0.1000 | 1.0556 |
| 0.2000 | 1.1250 |
| 0.3000 | 1.2143 |
| 0.4000 | 1.3333 |
| 0.5000 | 1.5000 |
| 0.6000 | 1.7500 |
| 0.7000 | 2.1667 |
| 0.8000 | 3.0000 |
| 0.9000 | 5.5000 |
| 0.9500 | 10.5000 |
| 0.9800 | 25.5000 |
| 0.9900 | 50.5000 |

# Endereçamento Aberto: Análise

- O hashing linear sofre de um mal chamado agrupamento.

- Mal do agrupamento
  - Ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas.

- O caso médio é O(1).

# O que fazer quando a tabela enche?

- Em ambas formas de tratar colisões, a complexidade das operações tende a piorar conforme a tabela recebe novos elementos.
- Note que nos dois casos elas estão ligadas ao quão grande é a razão N/M.
- A esta razão damos o nome de **fator de carga**.
- Manter o fator de carga limitado é essencial para garantir eficiência nas operações.

# Fator de carga

- Note que no encadeamento é possível, embora não desejável, que o fator de carga seja maior que 1.
- No endereçamento aberto, ter um fator de carga 1 significa que todos os espaços da tabela estão preenchidos, e ela não pode receber elementos novos.
- Uma alternativa para lidar com esse problema é aumentar o tamanho da tabela quando o fator de carga atingir determinado limiar.

# Reorganizar os elementos da tabela

Imagine uma tabela de tamanho 5 que recebe inteiros e a função de hash é %5. O elemento 7 foi inserido na tabela. Ao aumentar a tabela, onde o 7 deveria ficar?

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 7 |
| 3 | |
| 4 | |

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | ? |
| … | |
| 7 | ? |
| 8 | |
| 9 | |

# Reorganizar os elementos da tabela

Ao alterar o tamanho da tabela para 10, a função de hash se torna %10, então uma pesquisa pelo valor 7 irá começar na chave 7.

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 7 |
| 3 | |
| 4 | |

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | ? |
| … | |
| 7 | ? |
| 8 | |
| 9 | |

# Reorganizar os elementos da tabela

- Isso significa que todos os elementos que estavam presentes na tabela no momento em que seu tamanho aumentou, estão em posições potencialmente incorretas, portanto devemos reposicioná-los. Esse procedimento é chamado de *rehashing*.
- A forma mais simples de se fazer o *rehashing* consiste em iterar por todos os elementos da tabela e reinseri-los na tabela utilizando a nova função de *hashing*.
    - Como isso compromete bastante a eficiência, é interessante sempre aumentar o tamanho da tabela de forma substancial.

# Quando aumentar a tabela?

Decidir qual o fator de carga máximo permitido pela estrutura é crucial. No entanto o valor ideal pode variar de acordo com as peculiaridades da aplicação. No entanto queremos sempre garantir que:

- Se estamos usando encadeamento, queremos um limitante superior pequeno para a quantidade máxima de elementos em cada lista.
- Caso estejamos usando endereçamento aberto, queremos que existam muitos espaços livres para evitar o mal agrupamento.

# Vantagens e Desvantagens do Hashing

- Vantagens:
  - Alta eficiência no custo de pesquisa, que é O(1) para o caso médio.
  - Simplicidade de implementação

- Desvantagens:
  - Custo para recuperar os registros na ordem lexicográfica das chaves é alto, sendo necessário ordenar o arquivo.
  - Pior caso é O(N)