Pesquisa em Memória Primária

Algoritmos e Estruturas de Dados II

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

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
  - Pesquisa digital
  - *Hashing*

Tabelas de Símbolos

- Estrutura de dados contendo itens com chaves que suportam duas operações
  - Inserção de um novo item
  - Recuperar um item com uma determinada chave

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

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

- 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 65535

struct registro {
    int chave;
    /* outros componentes */
};

struct tabela {
    struct registro items[MAX+1];
    int tamanho;
};
```

Pesquisa Sequencial

```
struct tabela * cria_tabela(void) {
    struct tabela *T = malloc(sizeof(struct tabela));
    if(!T) { perror(NULL); exit(EXIT_FAILURE); }
    T->tamanho = 0;
    return T;
}

/* retorna 0 se não encontrar um registro com a chave. */
int pesquisa(int chave, struct tabela *T) {
    int i;
    T->items[0].chave = x; /* sentinela */
    for(i = T->tamanho; T->items[i].chave != chave; i--);
    return i;
}
```

Pesquisa Sequencial

```
void insere(struct registro reg, struct tabela *T) {
    if(T->tamanho == MAX)
       printf("Erro: tabela cheia\n");
    else {
       T->tamanho++;
       T->items[T->tamanho] = reg;
    }
}

void remove(int chave, struct tabela *T) {
     int idx = pesquisa(chave, T);
     /* se encontrou o item, troca pelo último e
      * reduz o tamanho: */
     if(idx) {
         T->items[idx] = T->items[T->tamanho];
         T->tamanho -= 1;
     }
}
```

Pesquisa Sequencial

- Análise:
  - Pesquisa com sucesso
    - melhor caso: $C(n) = 1$
    - pior caso: $C(n) = n$
    - caso médio: $C(n) = (n+1) / 2$
  - Pesquisa sem sucesso
    - $C(n) = n + 1$

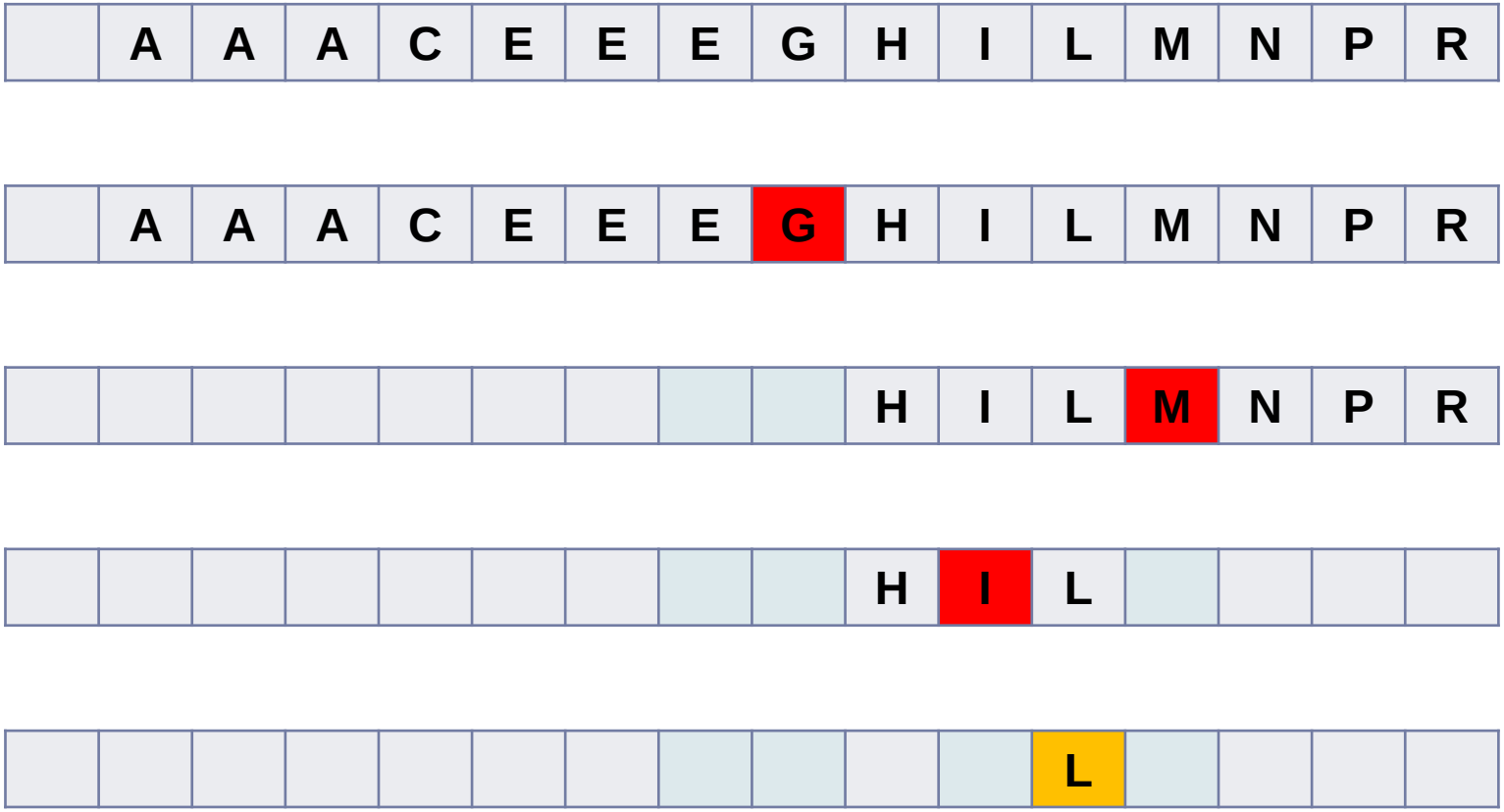
Pesquisa Binária

- Redução do tempo de busca aplicando o paradigma dividir para conquistar
  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

- Restrição: 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
  - Trocas de posições podem reduzir a eficiência

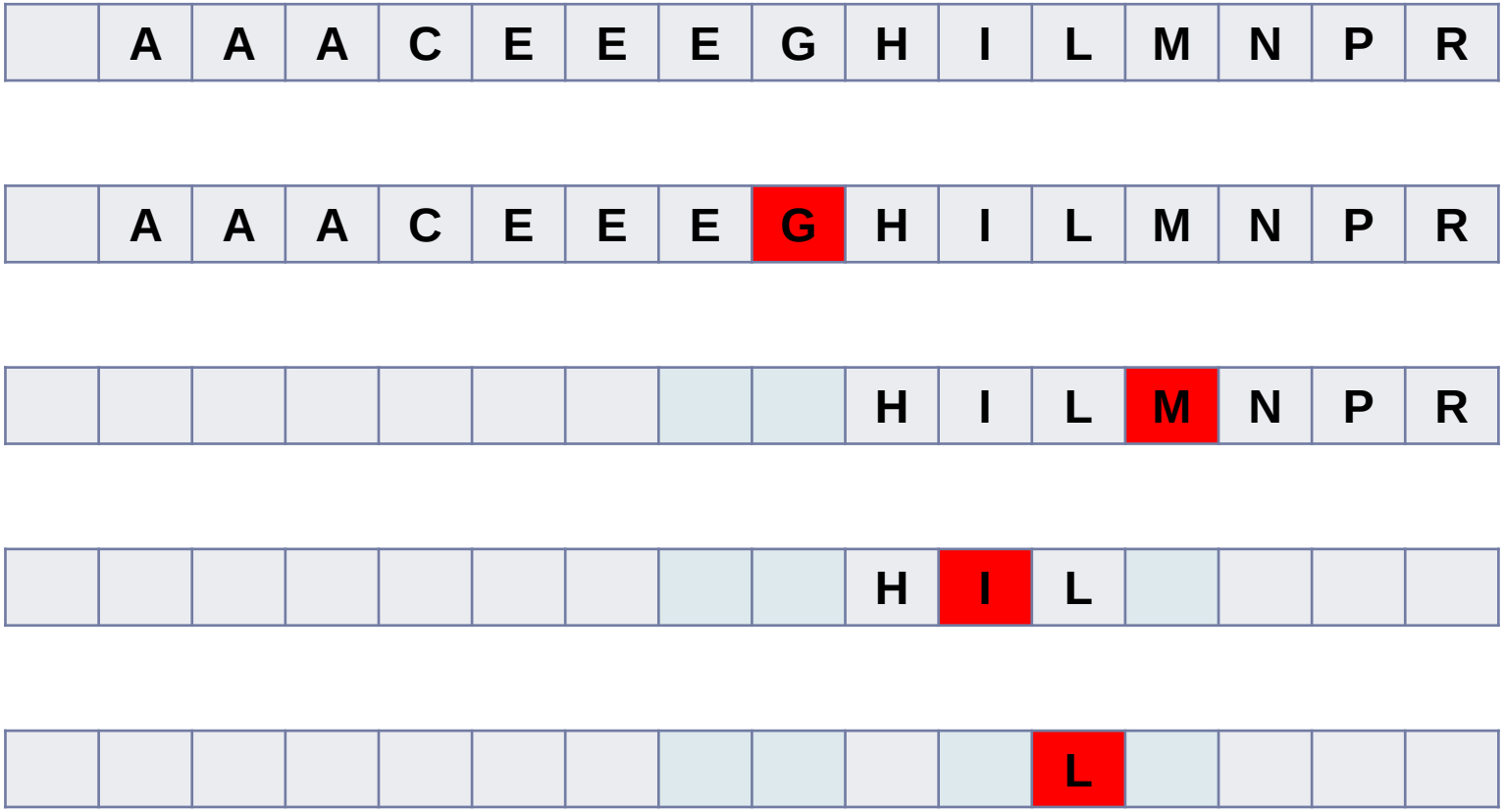
Pesquisa Binária

- Exemplo: pesquisa pela chave L



Pesquisa Binária

- Exemplo: pesquisa pela chave J



Pesquisa Binária

```
int binaria(int chave, struct tabela *T) {
    int i, esq, dir;
    if(T->tamanho == 0) { return 0; }
    esq = 1;
    dir = T->tamanho;
    do {
        i = (esq + dir) / 2;
        if(chave > T->items[i].chave) {
            esq = i + 1; /* procura na partição direita */
        } else {
            dir = i - 1; /* procura na part esquerda */
        }
    } while((chave != T->items[i].chave) && (esq <= dir));
    if (chave == T->items[i].chave) { return i; }
    else { return 0; }
}
```

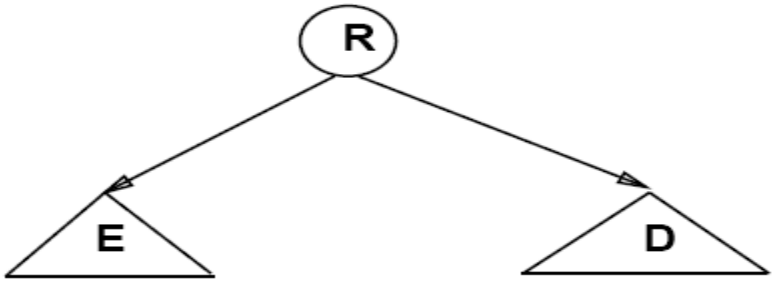
Pesquisa Binária

- Análise
  - 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 $\lg(n)$

- Ressalva
  - Alto custo para manter a tabela ordenada: a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes
  - Portanto, a pesquisa binária não deve ser usada em aplicações muito dinâmicas

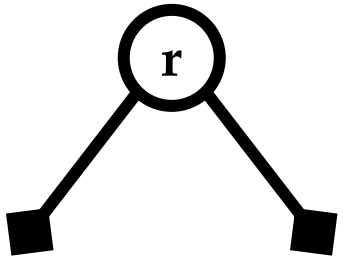
Árvore Binária de Pesquisa

- 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

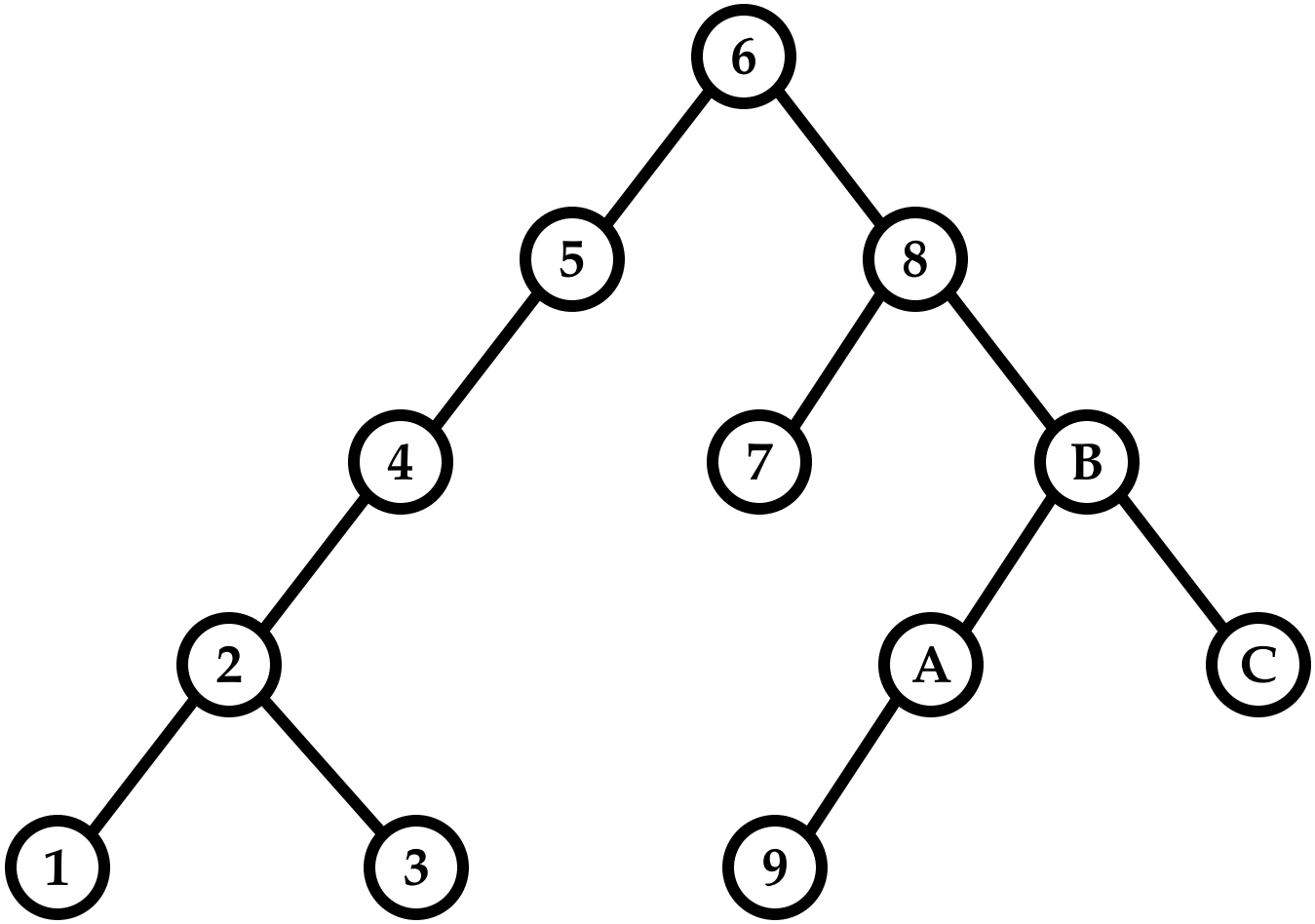
```
struct arvore {
    struct arvore *esq;
    struct arvore *dir;
    struct registro reg;
};

struct arvore *cria_arvore(struct registro reg) {
    struct arvore *novo;
    novo = malloc(sizeof(struct arvore));
    novo->esq = NULL;
    novo->dir = NULL;
    novo->reg = reg;
}
```



Árvore Binária de Pesquisa: Busca

```
void pesquisa(struct registro *r, struct arvore *t) {
    if(t == NULL) {
        printf("Registro não esta presente na árvore\n");
    } else if(r->chave < t->reg.chave) {
        pesquisa(r, t->esq); /* busca no filho esquerdo */
    } else if(r->chave > t->reg.chave)
        pesquisa(r, t->dir); /* busca no filho direito */
    else {
        *x = t->reg; /* retorna dados em r */
    }
}
```

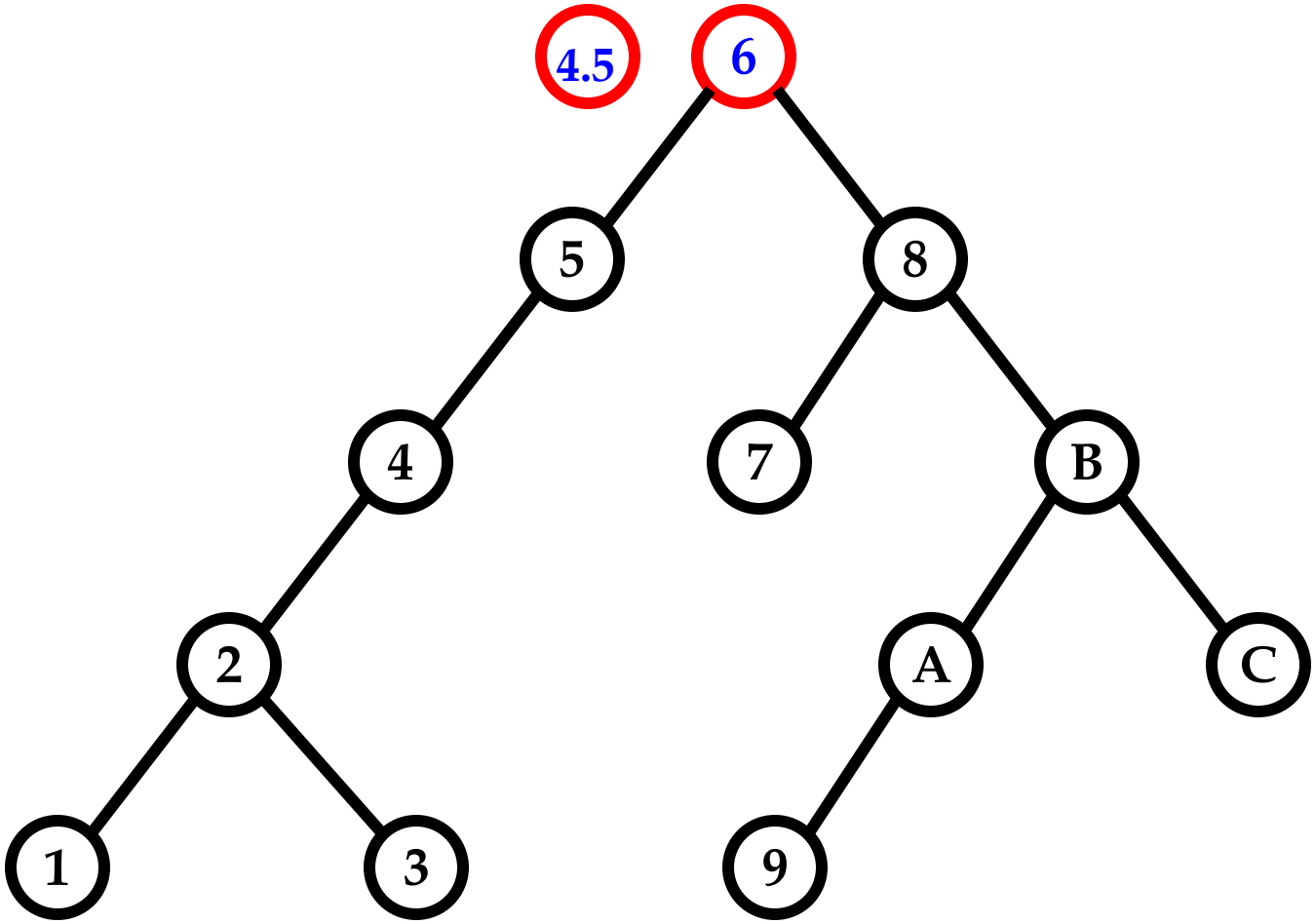
Árvore Binária de Pesquisa: Busca



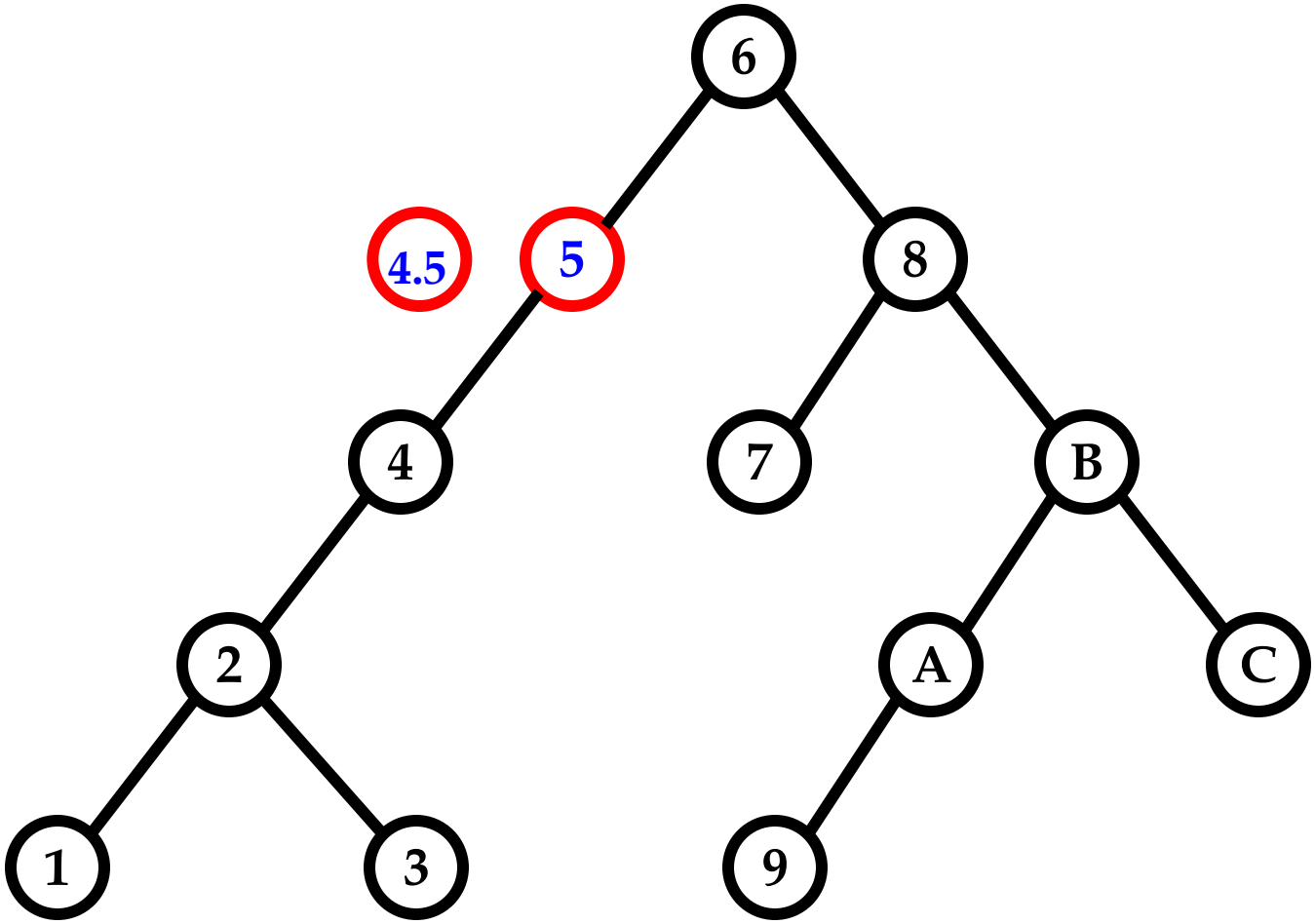
Árvore Binária de Pesquisa: Inserção

- O elemento vai ser inserido como uma *folha* da árvore de busca
- Vamos procurar o lugar de inserção navegando da raiz até a *folha* onde ele será inserido

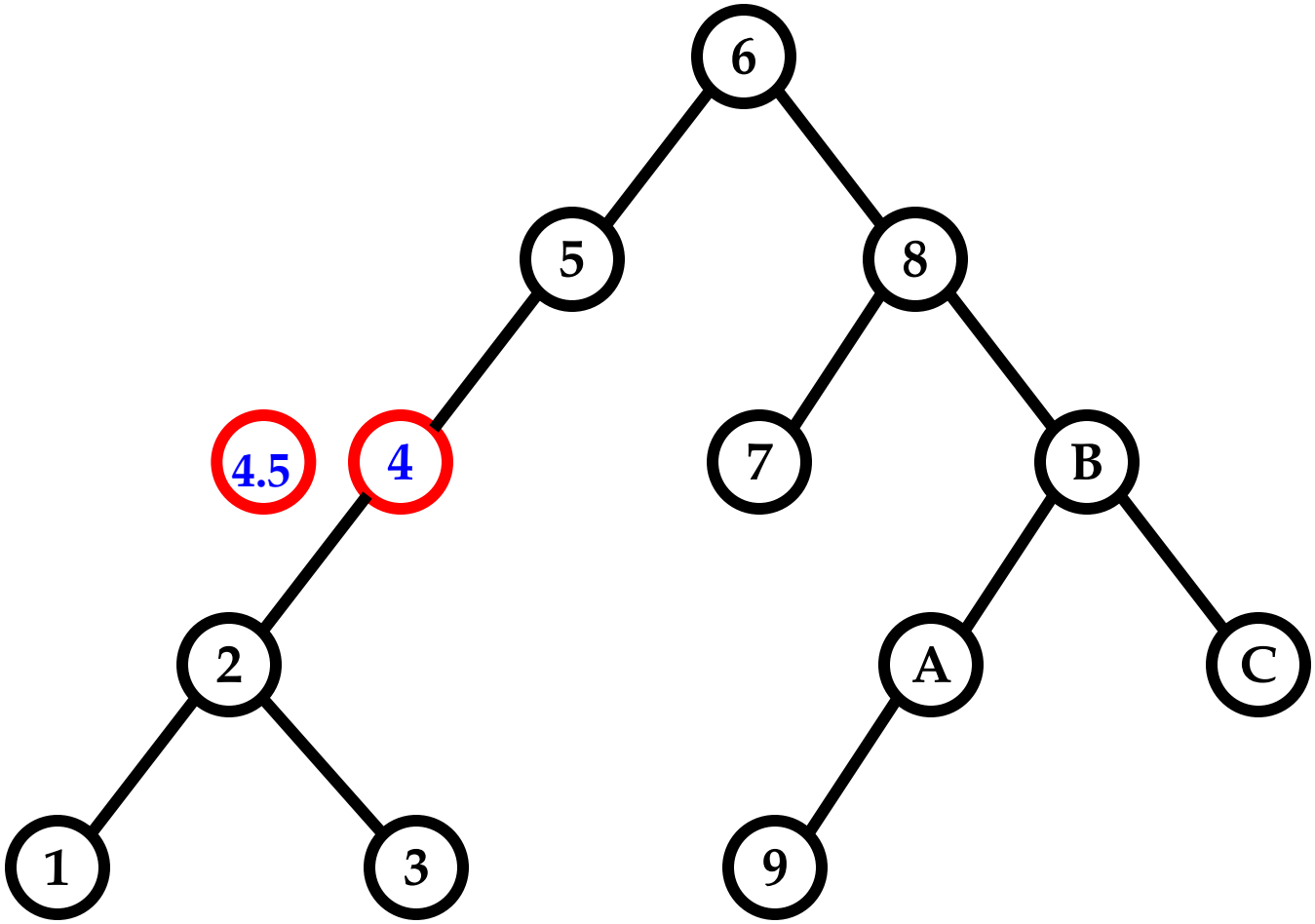
Árvore Binária de Pesquisa: Inserção



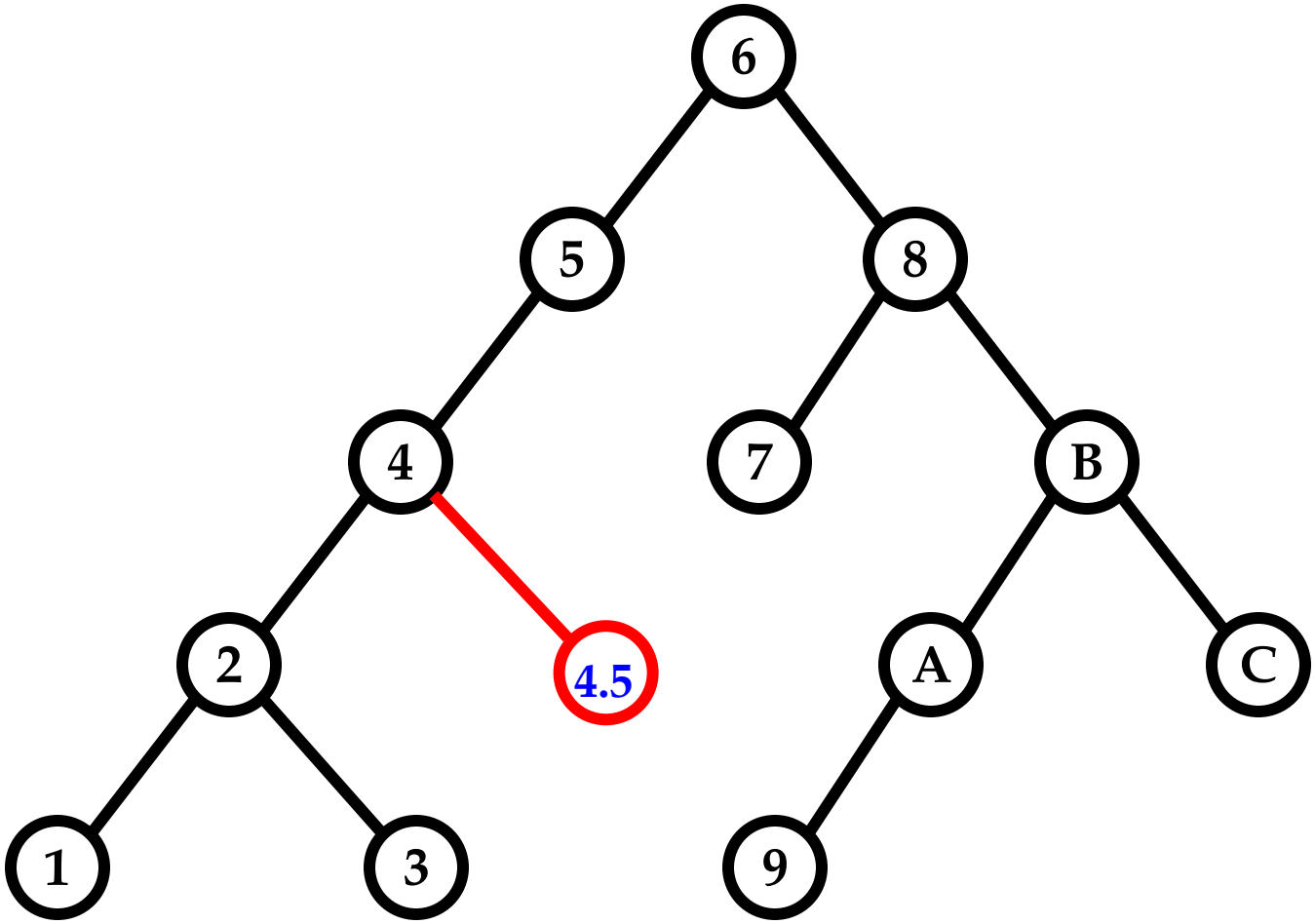
Árvore Binária de Pesquisa: Inserção



Árvore Binária de Pesquisa: Inserção



Árvore Binária de Pesquisa: Inserção

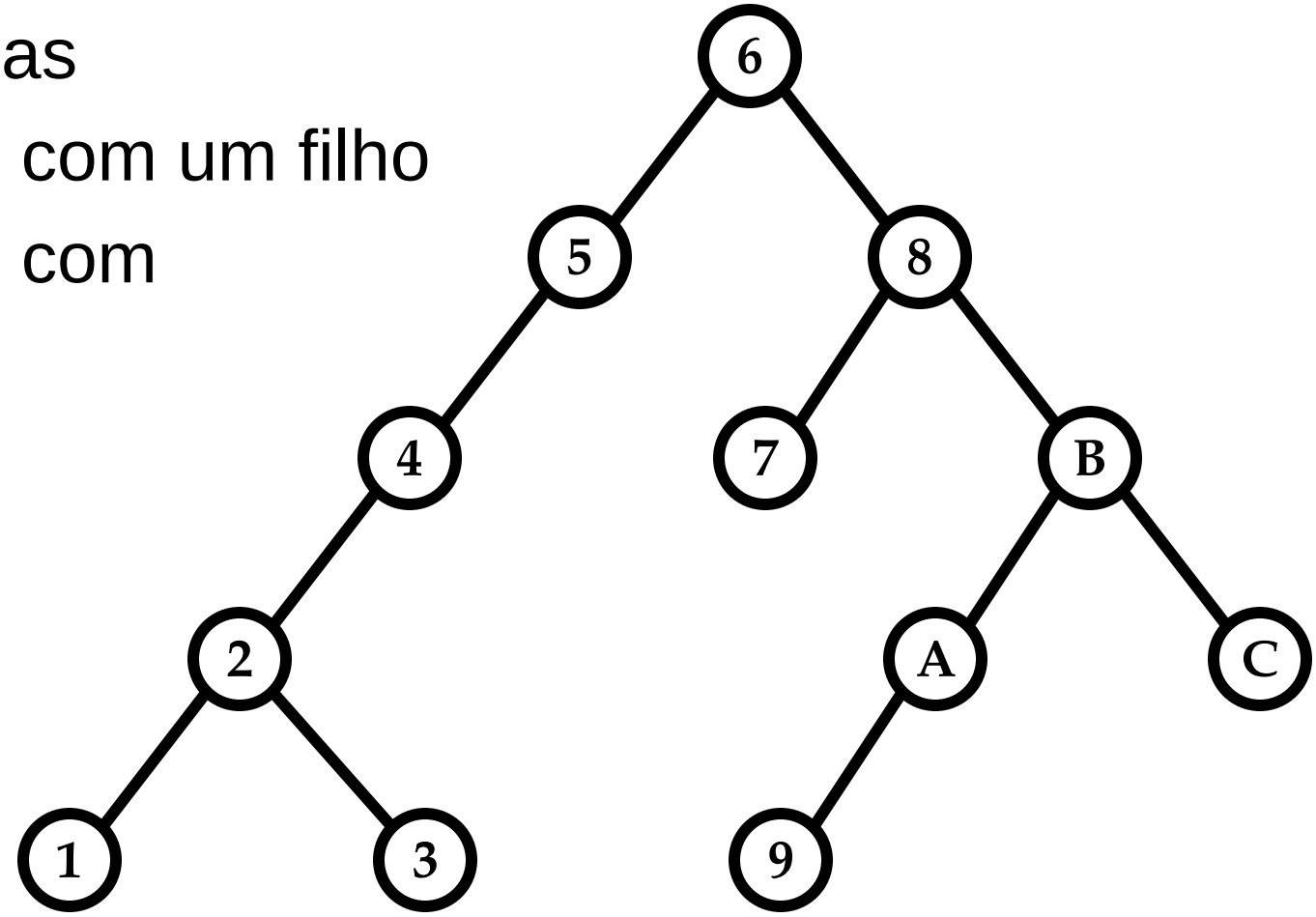


Árvore Binária de Pesquisa: Inserção

```
void insere_elemento(struct arvore *t, struct registro r) {
    if(r.chave < t->reg.chave) {  /* chave menor */
        if(t->esq) { insere_elemento(t->esq, r); }
        else { /* achou local de inserção */
            struct arvore *novo = cria_arvore(r);
            t->esq = novo;
        }
    } else { /* chave maior ou igual ao nodo atual */
        if(t->dir) { insere_elemento(t->dir, r); }
        else {
            struct arvore *novo = cria_arvore(r);
            t->dir = novo;
        }
    }
}
```

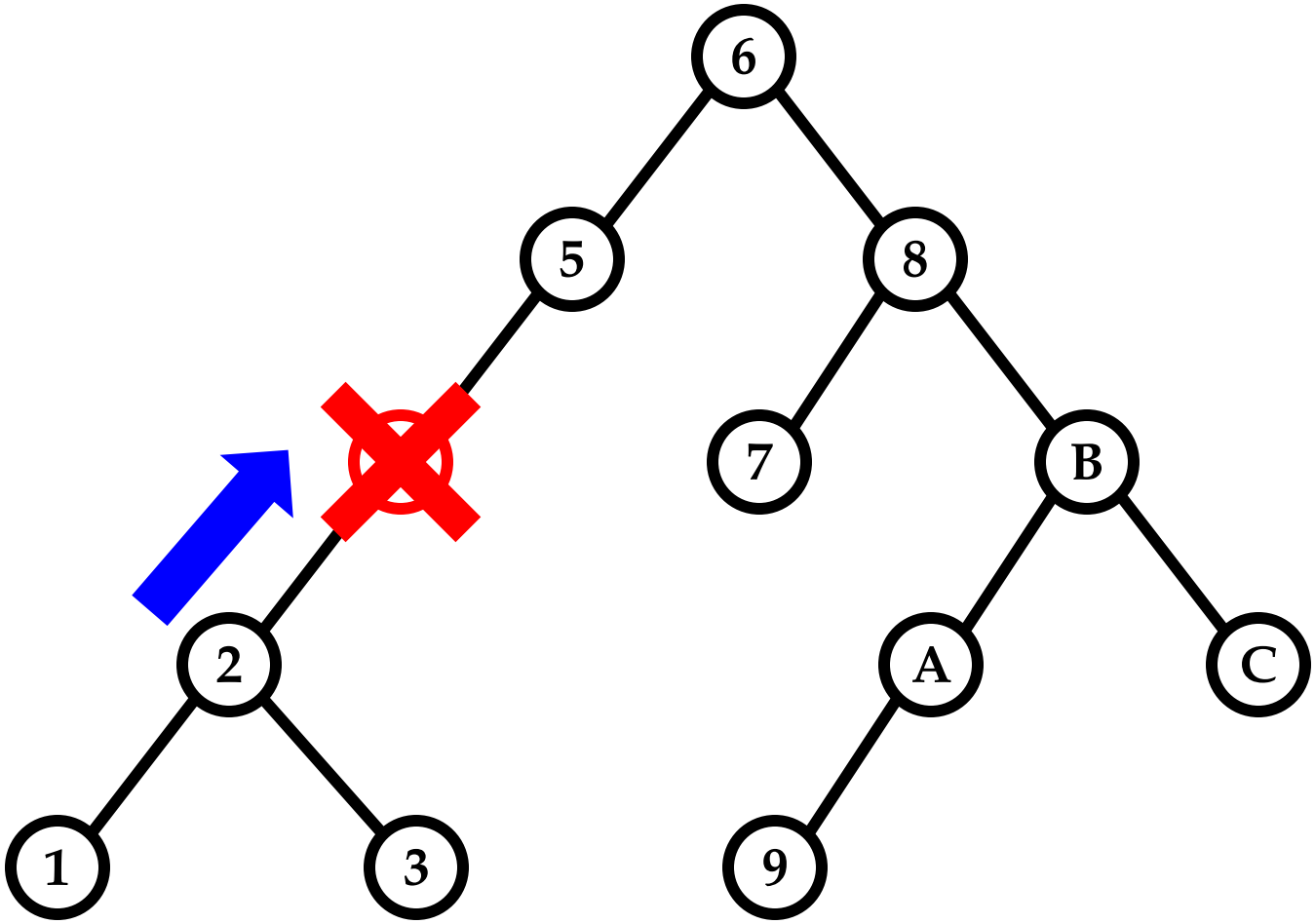
Árvore Binária de Pesquisa: Remoção

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



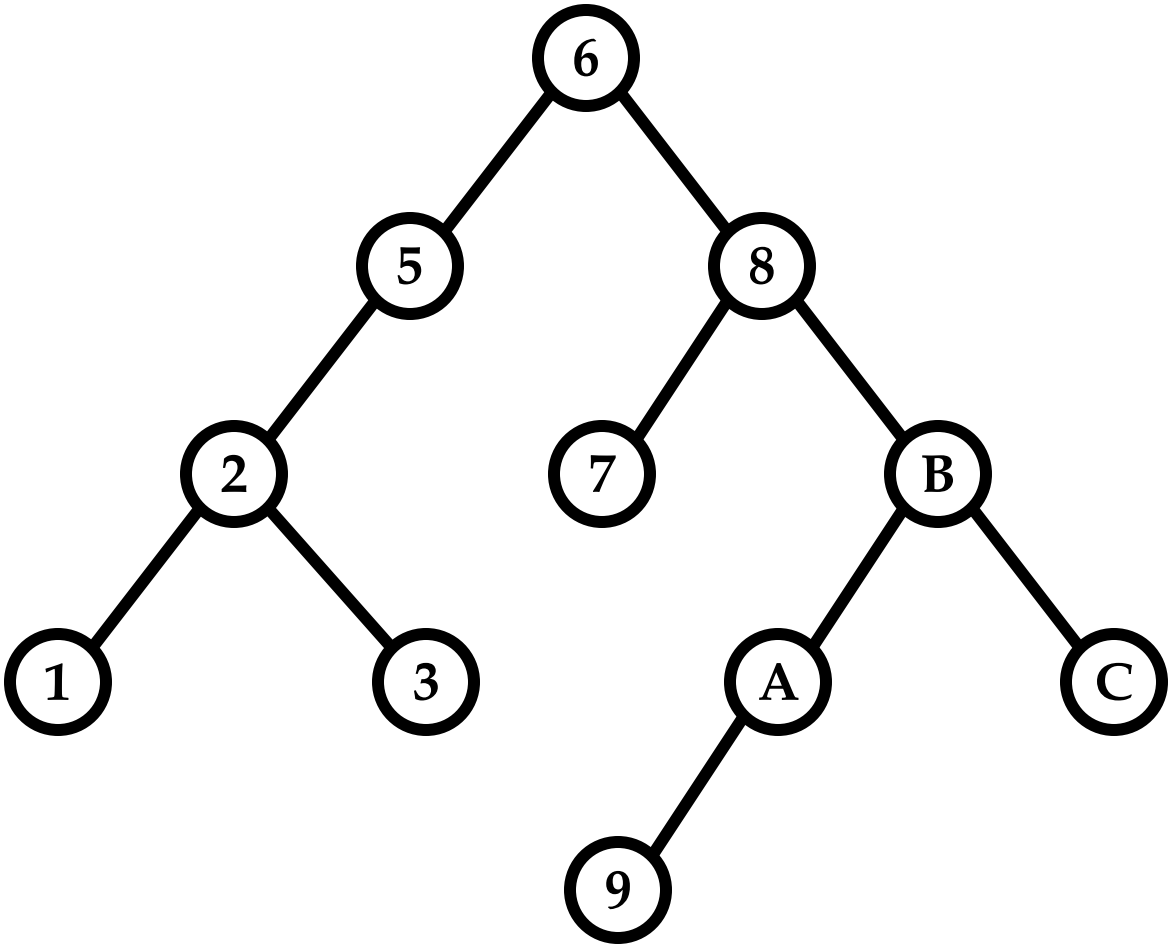
Árvore Binária de Pesquisa: Remoção

- Nó com 1 filho



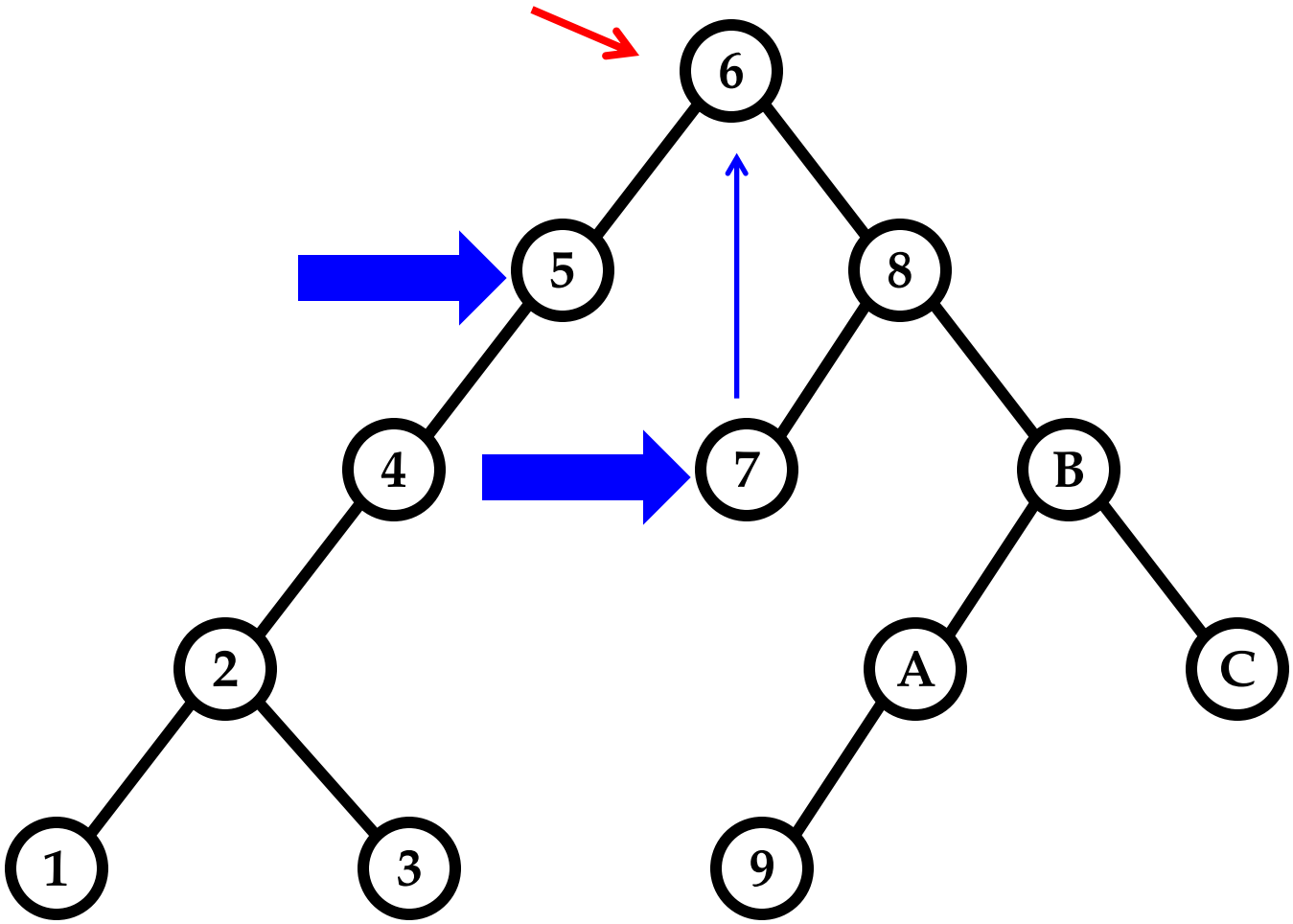
Árvore Binária de Pesquisa: Remoção

- Nó com 1 filho



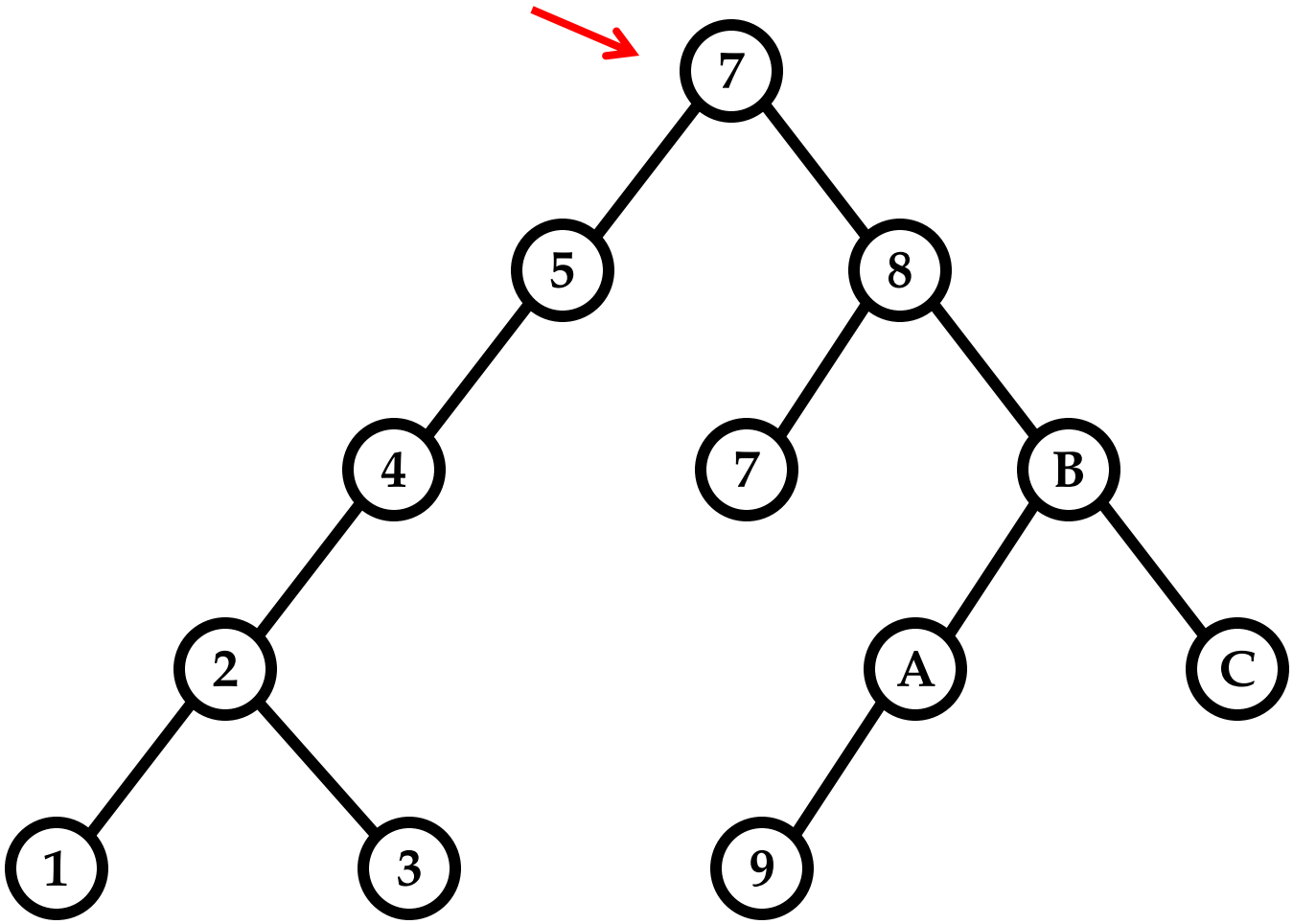
Árvore Binária de Pesquisa: Remoção

- Nó com 2 filhos



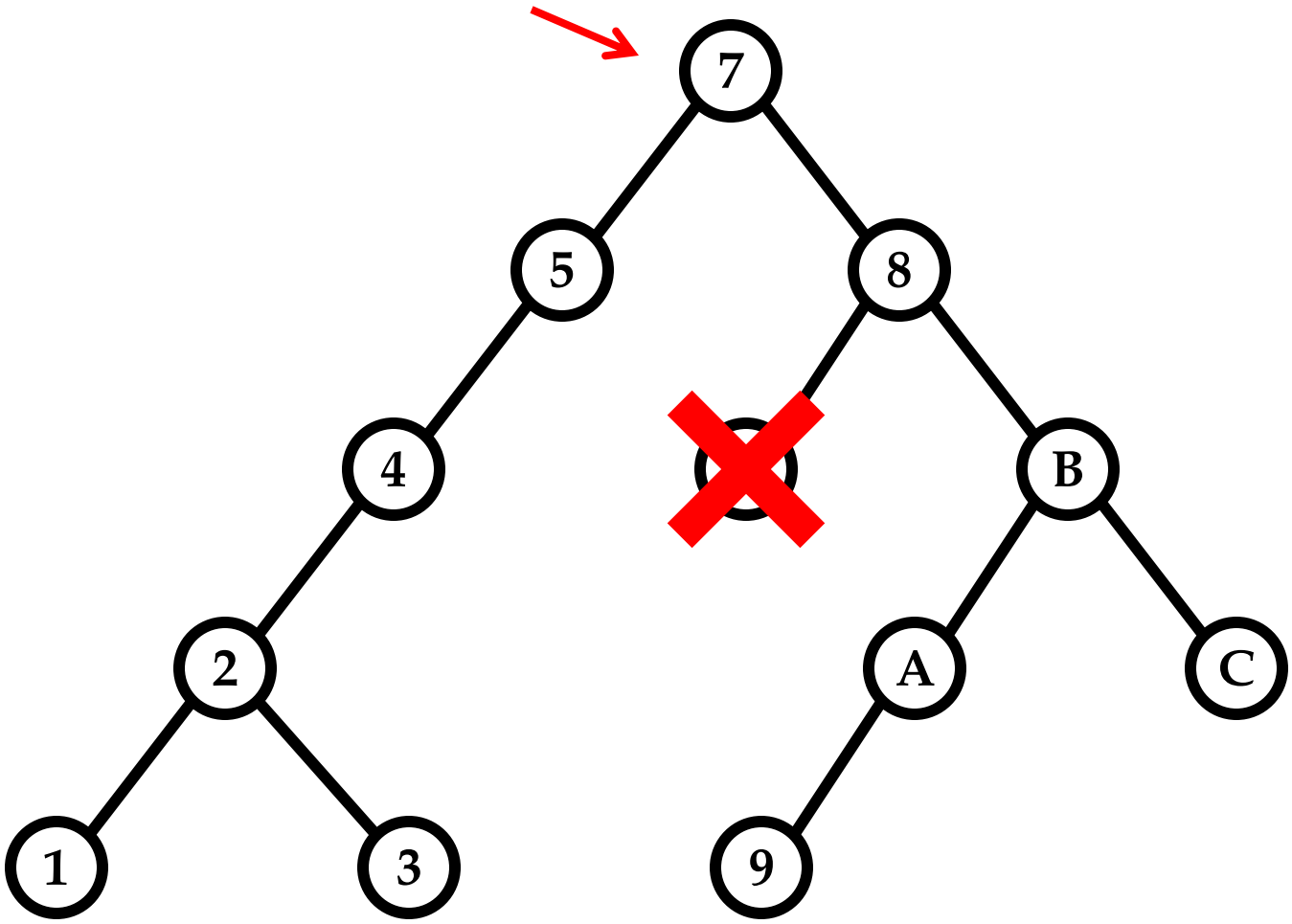
Árvore Binária de Pesquisa: Remoção

- Nó com 2 filhos



Árvore Binária de Pesquisa: Remoção

- Nó com 2 filhos



Árvore Binária de Pesquisa: Remoção

```
struct arvore * remove(struct arvore *t, int chave) {
   struct arvore *aux;
   if(t == NULL) { printf("elemento ausente\n"); }
   else if(chave < t->reg.chave){ t->esq=remove(t->esq, chave); }
   else if(chave > t->reg.chave){ t->dir=remove(t->dir, chave); }
   else if(t->esq == NULL && t->dir == NULL) {
      free(t); return NULL; /* zero filhos */
   } else if(t->esq == NULL) {
      aux = t->dir; free(t); return aux; /* 1 filho direita */
   } else if(t->dir == NULL) {
      aux = t->esq; free(t); return aux; /* 1 filho esquerda */
   } else { /* 2 filhos */
      struct arvore *suc = acha_menor(t->dir);
      t->reg = suc->reg;
      t->dir = remove(t->dir, suc->reg.chave);
      return t;
   }
   return t;
}
```

Funções para árvore de busca

```
void acha_menor(arvore *t) {
    if(t->esq == NULL) { return t; }
    return acha_menor(t->esq);
}
```